PostgreSQL 11.2 手册

PostgreSQL 全球开发组

翻译：彭煜玮[1]， 瀚高软件[2]及其他贡献者

[1] http://www.pengyuwei.net
[2] http://www.highgo.com/

PostgreSQL 11.2 手册

PostgreSQL 全球开发组
翻译：彭煜玮[1]， 瀚高软件[2]及其他贡献者

摘要

《PostgreSQL 11.2手册》基于上一版本的《PostgreSQL 10.1手册》[3]翻译， 并且合并了彭煜玮老师翻译的《PostgreSQL 11.0文档》[4]中的大量内容。 其余翻译工作由志愿者Steve888888,jingsam,chegong18,sizhitu,sunshinerxu,ChenHuajun(排名不分先后)等完成。 详细请参考PostgreSQL 11中文手册的翻译[5]。 如果发现中文手册中存在问题，请向Github源码仓库[6]提交问题报告(Issue)或修订Patch(PR)。

中文手册版本:1.1 最后更新时间:2020-03-14

版权声明

[1] http://www.pengyuwei.net
[2] http://www.highgo.com/
[3] http://www.postgres.cn/docs/10
[4] http://www.pengyuwei.net/PGDOC/110/index.html
[5] https://github.com/postgres-cn/pgdoc-cn/wiki/pg11
[6] https://github.com/postgres-cn/pgdoc-cn

目录

插图清单

表格清单

范例清单

前言

本书是PostgreSQL的官方文档。 它是PostgreSQL开发人员和其它志愿者与PostgreSQL的开发并行编写的。它描述了当前版本的PostgreSQL官方支持的所有功能。

为了能够管理有关PostgreSQL的大量信息，本书被组织成了几个部分。每个部分都是针对不同层次的用户，或者说针对具有不同阶段PostgreSQL体验的用户：

- 第 I 部是一个给新用户的非正式介绍。
- 第 II 部说明了SQL查询语言环境， 包括数据类型和函数，以及用户级别的性能调优。每个 PostgreSQL用户都应该阅读这些内容。
- 第 III 部描述服务器的安装和管理。每个运行PostgreSQL服务器的人，不管是个人使用还是为别人维护，都应该阅读这部分内容。
- 第 IV 部描述PostgreSQL客户端程序的编程接口。
- 第 V 部包含为高级用户准备的信息， 比如服务器的扩展能力等。其中的内容包括用户定义数据类型和函数等。
- 第 VI 部包含有关 SQL 命令、客户端和服务器程序的参考信息。这部分以按照命令或程序排序的结构化信息支持其它部分。
- 第 VII 部包含可用于PostgreSQL开发人员的各类信息。

1. 何为PostgreSQL?

PostgreSQL是以加州大学伯克利分校计算机系开发的POSTGRES， 版本 4.2[1]为基础的对象关系型数据库管理系统（ORDBMS）。POSTGRES 领先的许多概念在很久以后才出现在一些商业数据库系统中。

PostgreSQL是最初的伯克利代码的开源继承者。它支持大部分 SQL 标准并且提供了许多现代特性：

- 复杂查询
- 外键
- 触发器
- 可更新视图
- 事务完整性
- 多版本并发控制

同样，PostgreSQL可以用许多方法扩展，比如， 通过增加新的：

- 数据类型
- 函数
- 操作符
- 聚集函数
- 索引方法
- 过程语言

并且，因为自由宽大的许可证，任何人都可以以任何目的免费使用、修改和分发PostgreSQL， 不管是私用、商用还是学术研究目的。

2. PostgreSQL简史

[1] http://db.cs.berkeley.edu/postgres.html

现在被称为PostgreSQL的对象-关系型数据库管理系统是从加州大学伯克利分校写的POSTGRES软件包发展而来的。经过二十多年的发展，PostgreSQL是世界上可以获得的最先进的开源数据库。

2.1. 伯克利的POSTGRES项目

由Michael Stonebraker教授领导的POSTGRES项目是由防务高级研究项目局（DARPA）、陆军研究办公室（ARO）、国家科学基金（NSF）以及ESL, Inc共同赞助的。POSTGRES的实现始于1986年。该系统最初的概念详见[ston86]。 最初的数据模型定义见[rowe87]。当时的规则系统设计在[ston87a]里描述。存储管理器的理论基础和体系结构在[ston87b]里有详细描述。

从那以后，POSTGRES经历了几次主要的版本更新。第一个“演示性”系统在1987年便可使用了，并且在1988年的ACM-SIGMOD大会上展出。在1989年6月发布了版本1（见[ston90a]）给一些外部的用户使用。为了回应用户对第一个规则系统（[ston89]）的批评，规则系统被重新设计了（[ston90b]），在1990年6月发布了使用新规则系统的版本 2。 版本 3 在1991年出现，增加了多存储管理器的支持， 并且改进了查询执行器、重写了规则系统。直到Postgres95发布前（见下文）的后续版本大多把工作都集中在移植性和可靠性上。

POSTGRES已经被用于实现很多不同的研究和生产应用。这些应用包括： 一个财务数据分析系统、一个喷气引擎性能监控软件包、一个小行星跟踪数据库、一个医疗信息数据库和一些地理信息系统。POSTGRES还被许多大学用于教学用途。最后，Illustra Information Technologies（后来并入Informix[2]， 而Informix[3]现在被IBM[4]所拥有） 拿到代码并使之商业化。在 1992 年末POSTGRES成为Sequoia 2000科学计算项目[5]的主要数据管理器。

在 1993 年间，外部用户社区的数量几乎翻番。随着用户的增加， 用于源代码维护的时间日益增加并占用了太多本应该用于数据库研究的时间，为了减少支持的负担，伯克利的POSTGRES项目在版本 4.2 时正式终止。

2.2. Postgres95

在 1994 年，Andrew Yu 和 Jolly Chen 向POSTGRES中增加了 SQL 语言的解释器。并随后用新名字Postgres95将源代码发布到互联网上供大家使用， 成为最初POSTGRES伯克利代码的开源继承者。

Postgres95的源代码都是完全的 ANSI C，而且代码量减少了25%。许多内部修改提高了性能和可维护性。Postgres95的1.0.x版本在进行 Wisconsin Benchmark 测试时大概比POSTGRES的版本4.2 快 30-50%。除了修正了一些错误，下面的是一些主要提升：

- 原来的查询语言 PostQUEL 被SQL取代（在服务器端实现）。接口库libpq被按照PostQUEL命名。在PostgreSQL之前还不支持子查询（见下文），但它们可以在Postgres95中由用户定义的SQL函数模拟。聚集函数被重新实现。同时还增加了对GROUP BY 查询子句的支持。
- 新增加了一个利用GNU的Readline进行交互 SQL 查询的程序（psql）。这个程序很大程度上取代了老的monitor程序。
- 增加了新的前端库（libpgtcl），用以支持基于Tcl的客户端。一个样本 shell（pgtclsh），提供了新的 Tcl 命令用于Tcl程序和Postgres95服务器之间的交互。
- 彻底重写了大对象的接口。保留了将大对象倒转（Inversion ）作为存储大对象的唯一机制（去掉了倒转（Inversion ）文件系统）。
- 去掉了实例级的规则系统。但规则仍然以重写规则的形式存在。

[2] https://www.ibm.com/analytics/informix
[3] https://www.ibm.com/analytics/informix
[4] https://www.ibm.com/
[5] http://meteora.ucsd.edu/s2k/s2k_home.html

- 在发布的源码中增加了一个介绍SQL和Postgres95特性的简短教程。
- 用GNU的make（取代了BSD的make）来编译。Postgres95可以使用不打补丁的GCC编译（修正了双精度数据对齐问题）。

2.3. PostgreSQL

到了 1996 年， 很明显“Postgres95”这个名字已经跟不上时代了。于是我们选择了一个新名字PostgreSQL来反映与最初的POSTGRES和最新的具有SQL能力的版本之间的关系。同时版本号也从 6.0 开始， 将版本号放回到最初由伯克利POSTGRES项目开始的序列中。

很多人会因为传统或者更容易发音而继续用“Postgres”来指代PostgreSQL（现在很少用全大写字母）。这种用法也被广泛接受为一种昵称或别名。

Postgres95的开发重点放在标识和理解后端代码的现有问题上。PostgreSQL的开发重点则转到了一些有争议的特性和功能上面，当然各个方面的工作同时都在进行。

自那以来，PostgreSQL发生的变化可以在附录 E中找到。

3. 约定

下面的约定被用于命令的大纲：方括弧（[和]）表示可选的部分（在 Tcl 命令里，使用的是问号 （?），就像通常的 Tcl 一样）。 花括弧（{和}）和竖线（|）表示你必须选取一个候选。 点（...）表示它前面的元素可以被重复。

如果能提高清晰度，那么 SQL 命令前面会放上提示符=>， 而 shell 命令前面会放上提示符$。不过，提示符通常不被显示。

一个管理员通常是一个负责安装和运行服务器的人员。 一个用户可以是任何使用或者需要使用PostgreSQL系统的任何部分的人员。 我们不应该对这些术语的概念理解得太狭隘；这份文档集在系统管理过程方面没有固定的假设。

4. 进一步的信息

除了本文档（即本书），还有其他关于PostgreSQL的资源：

维基

PostgreSQL wiki[6]包含项目的FAQ[7]（常见问题）列表、TODO[8]列表以及更多主题的详细信息。

网站

PostgreSQL网站[9]中有最新发布的详细信息，以及其他让你能够以更高效使用PostgreSQL的信息。

邮件列表

邮件列表是一个提问的好地方，也可以与其他用户分享经验，还可以用来联系开发者。详情请参考PostgreSQL网站。

你自己!

PostgreSQL是一个开源项目。就其本身而言，它依赖于用户社区提供不间断的支持。如果你刚开始使用PostgreSQL，你将需要从其他人那里得到帮助，或者通过文档或者通过

[6] https://wiki.postgresql.org
[7] https://wiki.postgresql.org/wiki/Frequently_Asked_Questions
[8] https://wiki.postgresql.org/wiki/Todo
[9] https://www.postgresql.org

邮件列表。请考虑把你的知识贡献出来回馈社区。阅读邮件列表并且回答问题。如果你学到某些不在文档中的东西，请把它写下来并且贡献出来。如果你为代码增加了特性，也请把它们贡献出来。

5. 缺陷报告指南

当你在PostgreSQL中找到一个缺陷时，我们希望听到关于它的事情。你的缺陷报告在使得PostgreSQL更可靠的工作上扮演了重要的角色，因为再谨慎也不能保证PostgreSQL的每一部分都能在每一个平台上、每一种环境中工作。

下列建议目的是协助你把缺陷形成便于高效处理的形式。没有人被要求遵循它们，但是这样做对每个人都好。

我们无法保证马上修复每一个缺陷。如果缺陷是明显的、严重的或者影响很多用户，那么这是个好机会让某人去检查它。也可能我们会告诉你升级到一个新的版本来查看缺陷是否在该版本里也存在。或者我们可能决定在我们计划中的某些重大重写完成之前缺陷无法被修复。或者也许就是修复该缺陷太困难并且在日程表上还有更多重要的事情要做。如果你需要立即得到帮助，考虑联系一个商业支持。

5.1. 标识缺陷

在你报告一个缺陷之前，请阅读再阅读文档来验证你真地可以做你正在尝试的任何东西。如果从文档中无法清楚地知道你是否能做某事，也请把它报告给我们；这是一个文档中的缺陷。如果发现一个程序做的事情和文档说的不一样，那就是一个缺陷。那可能包括（但不限于）下列情况：

- 一个程序带有一个致命信号或者一个操作系统错误消息终止，它们会指出程序中的一个问题（一个反例是一个“磁盘满”消息，因为你只能自己修复它）。
- 一个程序对任何给定的输入产生错误的输出。
- 一个程序拒绝接受合法的输入（按文档定义）。
- 一个程序接受非法输入，并且没有提示或者错误消息。但是记住你对非法输入的概念可能是我们认为的一个扩展或兼容性。
- PostgreSQL根据在被支持平台上的指导无法编译、建立或者安装。

这里的“程序”指任何可执行文件，不仅仅是后台进程。

很慢或者很占资源不一定是一个缺陷。阅读文档或者在邮件列表中寻求帮助来调优你的应用。无法符合SQL标准也不一定是一个缺陷，除非文档中已经明确地声明指定特性是兼容的。

在你继续之前，检查 TODO 列表和 FAQ 来看你的缺陷是否为已知。如果你不能理解 TODO 列表中的信息，那就报告你的问题。至少我们可以让 TODO 列表更清晰。

5.2. 报告什么

关于缺陷报告要记住的最重要的事情是说明事实并且只说明事实。不要推断你觉得什么出错了、“它看起来在做”什么或者程序的哪一部分出错了。如果你不熟悉实现，你可能猜错并且不会帮到我们。而且即使你熟悉实现，受过训练的解释是巨大的补充但是也无法替代事实。如果我们想要修复缺陷，我们还是需要首先重现它。报告最基本的事实相对直接（你可以直接从屏幕上拷贝和粘贴它们）但是可能有太多重要的细节被略去，因为某些人可能认为它不重要（所以没有包含在报告中）或者报告可能以其他方式被理解。

下列项应该被包含在每一个缺陷报告中：

- 从程序启动开始的准确步骤序列，我们需要它们来重现问题。这应该是自包含的，如果输出是依赖于表中的数据，那么只发送一个裸的SELECT语句而不发送前面的CREATE

TABLE和INSERT语句是不够的。我们没有时间来对你的数据库模式做逆向工程，而且即使我们能够通过我们自己的数据来弥补，我们也很可能会错过问题。

SQL 相关问题的一个测试样例的最佳格式是一个可以通过psql前端运行并展示该问题的文件（注意不要在你的~/.psqlrc启动文件中包含任何东西）。一种简单的创建该文件的方法是使用pg_dump来转储出表声明和设置场景的数据，然后增加问题查询。我们鼓励你最小化你的例子的尺寸，但是这不是绝对必要的。如果缺陷是可重现的，我们以两种方法都可以找到它。

如果你的应用使用某些其他客户端接口（例如PHP），那么请尝试隔离出错的查询。我们将可能无法设置一个网页服务器来重现你的问题。在任何情况中记住提供准确的输入文件，不要猜测问题是因为“大文件” 或“中等大小的数据库”等而发生，因为这些信息用起来太不准确。

- 你得到的输出。请不要说它“无法工作”或者“崩溃了”。如果有一个错误消息，请展示它，即使你不理解它。如果程序带着一个操作系统错误而终止，请说出它。如果根本没有发生任何事情，也说出来。虽然你的测试样例的结果是一次程序崩溃，但是显然它不一定会在我们的平台上发生。如果可能的话，最简单的事情就是从终端上复制你的输出。

> **注意**
>
> 如果你正在报告一个错误消息，请获得消息的最冗长的形式。在psql中，预先执行\set VERBOSITY verbose。如果你在从服务器日志抽取消息，设置运行时参数log_error_verbosity为verbose，这样所有的细节将被记录在日志中。

> **注意**
>
> 在致命错误的情况中，客户端报告的错误消息可能不会包含所有可用的信息。请也看看数据库服务器的日志输出。如果你没有保留你的服务器的日志输出，这将是一个最好的时机开始记录它。

- 你所期望的输出也是需要说明的很重要的内容。如果你只写“这个命令给我那个输出。”或“这不是我所期待的。”，我们可能自己运行它、扫描输出并且认为它看起来OK 并且正是我们期望的。我们不会花时间来理解你的命令背后的准确语义。特别是避免仅仅说“这不是 SQL 所说的/Oracle 所作的。”从SQL中挖掘出正确的行为不是一个有趣的工作，我们也没法去了解所有的其他关系型数据库的行为（如果你的问题是一个程序崩溃，你显然可以忽略这一项）。

- 任何命令行选项和其他启动选项，包括任何相关的环境变量或者你从默认修改过的配置文件。再次，请提供准确的信息。如果你在使用一个预打包的发布并且它在系统启动时自动开始数据库服务器，你应当尝试找出它是怎样启动的。

- 任何你做的和安装指导上不同的地方。

- PostgreSQL的版本。你可以运行命令SELECT version();来找出你连接到的服务器的版本。大部分可执行程序也支持一个--version选项，至少postgres --version和psql --version可以工作。如果该函数或选项不存在，那么你的版本就已经老得无法保证可以升级了。如果你运行的是一个预打包的版本（如 RPM），请说明并且包括该包的任何子版本。如果你在谈论一个 Git 快照，也请说明并且包括提交哈希值。

  如果你的版本比 11.2 更老，我们将几乎肯定会告诉你进行升级。在每一个新的发布中都包含很多缺陷修复和改进，因此很有可能你在旧版本PostgreSQL中碰到的一个缺陷已经被修复了。我们对使用旧版本PostgreSQL的站点只提供有限的支持。如果你需要更多支持，请考虑咨询一个商业支持。

- 平台信息。这包括内核名称和版本、C 库、处理器、内存信息等等。在大部分情况中，提供厂家和版本就足够了，但是不要假定每个人都了解“Debian”中到底包含什么或者每个人都运行在 i386 上。如果你碰到的是安装问题，那么你机器上的工具链（编译器、make 等等）的信息也是需要被报告的。

不要担心你的缺陷报告变得太长。这就是生活。最好是第一次就报告所有的东西，而不是让我们去从你那里询问。在另一方面，如果你的输入文件太大，最好先问问是否有人有兴趣去看它。这里有一篇文章[10]勾勒了一些报告缺陷的提示。

不要把你所有的时间花费在指出输入中的哪些改变让问题消失。这可能对解决问题没有什么帮助。如果发现缺陷不能被立马解决，你将还有时间去寻找和分享你的解决方法。同样，不要浪费你的时间去猜测为什么缺陷会存在。我们将尽快找出原因。

在编写一份缺陷报告时，请避免使用含糊的术语。这个软件包从整体上被称为“PostgreSQL”，有时会简称为“Postgres”。如果你要谈论特定的后端进程，不要只说“PostgreSQL 崩溃了”。一个单一后端进程的崩溃和其父“postgres”进程崩溃是完全不同的；当你想说一个单一后端进程垮掉时，请不要说“服务器崩溃了”，反之亦然。还有，如交互式前端“psql”等的客户端程序和后端是完全独立的。请尽量确定问题是发生在客户端还是服务器端。

5.3. 向哪里报告缺陷

通常，请把缺陷报告发送到缺陷报告邮件列表<pgsql-bugs@lists.postgresql.org>。你会被要求给你的电子邮件消息加上一个描述性的主题，可以用错误消息的一部分。

另一种方法是填写项目网站[11]上的缺陷报告网页表格。以这种方式输入的缺陷报告会被发送到<pgsql-bugs@lists.postgresql.org>邮件列表。

如果你的缺陷报告牵涉到安全并且你不想让它立刻变得公众可见，不要把它发送到pgsql-bugs。安全问题可以被私下报告给<security@lists.postgresql.org>。

不要将缺陷报告发送到任何一个用户邮件列表，例如<pgsql-sql@lists.postgresql.org>或<pgsql-general@lists.postgresql.org>。这些邮件列表是用来回答用户问题的，并且它们的订阅者通常不希望收到缺陷报告。更重要的是，他们不可能去修复缺陷。

另外，请不要把报告发送给开发者邮件列表<pgsql-hackers@lists.postgresql.org>。这个列表是用来讨论PostgreSQL开发的地方，并且把缺陷报告隔离开对我们会比较好。如果问题需要更多的审查，我们可能选择在pgsql-hackers上对你的缺陷报告展开一次讨论。

如果你有一个关于文档的问题，报告它最好的地方是文档邮件列表<pgsql-docs@postgresql.org>。请指出你对文档的哪个部分不爽。

如果你的缺陷是一个在非被支持平台上的移植性问题，请发送邮件到<pgsql-hackers@postgresql.org>，这样我们（和你）就可以做些工作把PostgreSQL移植到你的平台。

> **注意**
>
> 为了防止出现大量的垃圾邮件，除非您订阅，否则上述所有列表都将被审核。这意味着在邮件被发送之前，将有一些延迟。 如果您想订阅列表，请访问https://lists.postgresql.org/获取说明。

[10] https://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[11] https://www.postgresql.org/

部分 I. 教程

欢迎来到PostgreSQL教程。下面的几章将为那些新接触PostgreSQL、关系数据库概念和 SQL 语言的读者给出一个简单介绍。我们只假定读者拥有关于如何使用计算机的一般知识。读者不需要特殊的 Unix 或编程经验。这一部分主要希望给你一些关于PostgreSQL系统的重要方面的手把手的体验。我们并不准备把它写成一份能覆盖这些主题所有内容的文档。

当你完成了这份教程之后，你可能希望继续去阅读第 II 部分得到 SQL 语言更正式的知识，或者阅读第 IV 部分了解如何开发PostgreSQL应用程序。希望搭建并且管理自己的服务器的用户还应该阅读第 III 部分

目录

第 1 章 从头开始

1.1. 安装

自然，在你能开始使用PostgreSQL之前， 你必须安装它。PostgreSQL很有可能已经安装到你的节点上了， 因为它可能包含在你的操作系统的发布里， 或者是系统管理员已经安装了它。如果是这样的话， 那么你应该从操作系统的文档或者你的系统管理员那里获取有关如何访问PostgreSQL的信息。

如果你不清楚PostgreSQL是否已经安装， 或者不知道你能否用它（已经安装的）做自己的实验，那么你就可以自己安装。 这么做并不难，并且是一次很好的练习。PostgreSQL可以由任何非特权用户安装， 并不需要超级用户 （root）的权限。

如果你准备自己安装PostgreSQL， 那么请参考第 16 章获取安装的有关信息， 安装之后再回到这个指导手册来。一定要记住要尽可能遵循有关设置合适的环境变量章节里的信息。

如果你的站点管理员没有按照缺省的方式设置各项相关参数， 那你还有点额外的活儿要干。比如，如果数据库服务器机器是一个远程的机器， 那你就需要把PGHOST环境变量设置为数据库服务器的名字。环境变量PGPORT也可能需要设置。总而言之就是： 如果当你试着启动一个应用而该应用报告说不能与数据库建立联接时， 你应该马上与你的数据库管理员联系，如果你就是管理员， 那么你就要参考文档以确保你的环境变量得到正确的设置。 如果你不理解随后的几段，那么先阅读下一节。

1.2. 架构基础

在我们继续之前，你应该先了解PostgreSQL的系统架构。 对PostgreSQL的部件之间如何相互作用的理解将会使本节更易理解。

在数据库术语里，PostgreSQL使用一种客户端/服务器的模型。一次PostgreSQL会话由下列相关的进程（程序）组成：

- 一个服务器进程，它管理数据库文件、接受来自客户端应用与数据库的联接并且代表客户端在数据库上执行操作。 该数据库服务器程序叫做postgres。
- 那些需要执行数据库操作的用户的客户端（前端）应用。 客户端应用可能本身就是多种多样的：可以是一个面向文本的工具， 也可以是一个图形界面的应用，或者是一个通过访问数据库来显示网页的网页服务器，或者是一个特制的数据库管理工具。 一些客户端应用是和 PostgreSQL发布一起提供的，但绝大部分是用户开发的。

和典型的客户端/服务器应用（C/S应用）一样，这些客户端和服务器可以在不同的主机上。这时它们通过 TCP/IP 网络联接通讯。 你应该记住的是，在客户机上可以访问的文件未必能够在数据库服务器机器上访问（或者只能用不同的文件名进行访问）。

PostgreSQL服务器可以处理来自客户端的多个并发请求。 因此，它为每个连接启动（“forks”）一个新的进程。 从这个时候开始，客户端和新服务器进程就不再经过最初的 postgres进程的干涉进行通讯。 因此，主服务器进程总是在运行并等待着客户端联接， 而客户端和相关联的服务器进程则是起起停停（当然，这些对用户是透明的。我们介绍这些主要是为了内容的完整性）。

1.3. 创建一个数据库

看看你能否访问数据库服务器的第一个例子就是试着创建一个数据库。 一台运行着的PostgreSQL服务器可以管理许多数据库。 通常我们会为每个项目和每个用户单独使用一个数据库。

你的站点管理员可能已经为你创建了可以使用的数据库。 如果这样你就可以省略这一步，并且跳到下一节。

要创建一个新的数据库，在我们这个例子里叫mydb，你可以使用下面的命令：

```
$ createdb mydb
```

如果不产生任何响应则表示该步骤成功，你可以跳过本节的剩余部分。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

那么就是PostgreSQL没有安装好。或者是根本没安装， 或者是你的shell搜索路径没有设置正确。尝试用绝对路径调用该命令试试：

```
$ /usr/local/pgsql/bin/createdb mydb
```

在你的站点上这个路径可能不一样。和你的站点管理员联系或者看看安装指导获取正确的位置。

另外一种响应可能是这样：

```
createdb: could not connect to database postgres: could not connect to server:
 No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

这意味着该服务器没有启动，或者没有按照createdb预期地启动。同样， 你也要查看安装指导或者咨询管理员。

另外一个响应可能是这样：

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not
 exist
```

在这里提到了你自己的登录名。如果管理员没有为你创建PostgreSQL用户帐号， 就会发生这些现象。（PostgreSQL用户帐号和操作系统用户帐号是不同的。） 如果你是管理员，参阅第 21 章获取创建用户帐号的帮助。 你需要变成安装PostgreSQL的操作系统用户的身份（通常是 postgres）才能创建第一个用户帐号。 也有可能是赋予你的PostgreSQL用户名和你的操作系统用户名不同； 这种情况下，你需要使用-U选项或者使用PGUSER环境变量指定你的PostgreSQL用户名。

如果你有个数据库用户帐号，但是没有创建数据库所需要的权限，那么你会看到下面的信息：

```
createdb: database creation failed: ERROR:  permission denied to create database
```

并非所有用户都被许可创建新数据库。 如果PostgreSQL拒绝为你创建数据库， 那么你需要让站点管理员赋予你创建数据库的权限。出现这种情况时请咨询你的站点管理员。 如果你自己安装了PostgreSQL， 那么你应该以你启动数据库服务器的用户身份登录然后参考手册完成权限的赋予工作。 [1]

你还可以用其它名字创建数据库。PostgreSQL允许你在一个站点上创建任意数量的数据库。数据库名必须是以字母开头并且小于 63 个字符长。 一个方便的做法是创建和你当前用户名

[1] 为什么这么做的解释：PostgreSQL用户名是和操作系统用户账号分开的。 如果你连接到一个数据库时，你可以选择以何种PostgreSQL用户名进行联接； 如果你不选择，那么缺省就是你的当前操作系统账号。 如果这样，那么总有一个与操作系统用户同名的PostgreSQL用户账号用于启动服务器， 并且通常这个用户都有创建数据库的权限。如果你不想以该用户身份登录， 那么你也可以在任何地方声明一个-U选项以选择一个用于连接的PostgreSQL用户名。

同名的数据库。 许多工具假设该数据库名为缺省数据库名，所以这样可以节省你的敲键。要创建这样的数据库，只需要键入：

```
$ createdb
```

如果你再也不想使用你的数据库了，那么你可以删除它。 比如，如果你是数据库mydb的所有人（创建人）， 那么你就可以用下面的命令删除它：

```
$ dropdb mydb
```

（对于这条命令而言，数据库名不是缺省的用户名，因此你就必须声明它） 。这个动作将在物理上把所有与该数据库相关的文件都删除并且不可取消， 因此做这中操作之前一定要考虑清楚。

更多关于createdb和dropdb的信息可以分别在createdb和dropdb中找到。

1.4. 访问数据库

一旦你创建了数据库，你就可以通过以下方式访问它：

- 运行PostgreSQL的交互式终端程序，它被称为psql， 它允许你交互地输入、编辑和执行SQL命令。
- 使用一种已有的图形化前端工具，比如pgAdmin或者带ODBC或JDBC支持的办公套件来创建和管理数据库。这种方法在这份教程中没有介绍。
- 使用多种绑定发行的语言中的一种写一个自定义的应用。这些可能性在第 IV 部分将有更深入的讨论。

你可能需要启动psql来试验本教程中的例子。 你可以用下面的命令为mydb数据库激活它：

```
$ psql mydb
```

如果你不提供数据库名字，那么它的缺省值就是你的用户账号名字。在前面使用createdb的小节里你应该已经了解了这种方式。

在psql中，你将看到下面的欢迎信息：

```
psql (11.2)
Type "help" for help.

mydb=>
```

最后一行也可能是：

```
mydb=#
```

这个提示符意味着你是数据库超级用户，最可能出现在你自己安装了 PostgreSQL实例的情况下。 作为超级用户意味着你不受访问控制的限制。 对于本教程的目的而言， 是否超级用户并不重要。

如果你启动psql时碰到了问题，那么请回到前面的小节。诊断createdb的方法和诊断psql的方法很类似， 如果前者能运行那么后者也应该能运行。

psql打印出的最后一行是提示符，它表示psql正听着你说话，这个时候你就可以敲入 SQL查询到一个psql维护的工作区中。试验一下下面的命令：

```
mydb=> SELECT version();
                                                  version
-------------------------------------------------------------------------------------------
 PostgreSQL 11.2 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10)
 4.9.2, 64-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

psql程序有一些不属于SQL命令的内部命令。它们以反斜线开头，“\”。 欢迎信息中列出了一些这种命令。比如，你可以用下面的命令获取各种PostgreSQL的SQL命令的帮助语法：

```
mydb=> \h
```

要退出psql，输入：

```
mydb=> \q
```

psql将会退出并且让你返回到命令行shell。 （要获取更多有关内部命令的信息，你可以在psql提示符上键入\?。） psql的完整功能在psql中有文档说明。在这份文档里，我们将不会明确使用这些特性，但是你自己可以在需要的时候使用它们。

第 2 章 SQL语言

2.1. 引言

本章提供一个如何使用SQL执行简单操作的概述。本教程的目的只是给你一个介绍，并非完整的SQL教程。有许多关于SQL的书籍，包括[melt93]和[date97]。你还要知道有些PostgreSQL语言特性是对标准的扩展。

在随后的例子里，我们假设你已经创建了名为mydb的数据库，就象在前面的章里面介绍的一样，并且已经能够启动psql。

本手册的例子也可以在PostgreSQL源代码的目录src/tutorial/中找到（二进制PostgreSQL发布中可能没有编译这些文件）。要使用这些文件，首先进入该目录然后运行make：

```
$ cd ..../src/tutorial
$ make
```

这样就创建了那些脚本并编译了包含用户定义函数和类型的 C 文件。接下来，要开始本教程，按照下面说的做：

```
$ cd ..../tutorial
$ psql -s mydb

...


mydb=> \i basics.sql
```

\i命令从指定的文件中读取命令。psql的-s选项把你置于单步模式，它在向服务器发送每个语句之前暂停。 在本节使用的命令都在文件basics.sql中。

2.2. 概念

PostgreSQL是一种关系型数据库管理系统 （RDBMS）。这意味着它是一种用于管理存储在关系中的数据的系统。关系实际上是表的数学术语。 今天，把数据存储在表里的概念已经快成了固有的常识了， 但是还有其它的一些方法用于组织数据库。在类 Unix 操作系统上的文件和目录就形成了一种层次数据库的例子。 更现代的发展是面向对象数据库。

每个表都是一个命名的行集合。一个给定表的每一行由同一组的命名列组成，而且每一列都有一个特定的数据类型。虽然列在每行里的顺序是固定的， 但一定要记住 SQL 并不对行在表中的顺序做任何保证（但你可以为了显示的目的对它们进行显式地排序）。

表被分组成数据库，一个由单个PostgreSQL服务器实例管理的数据库集合组成一个数据库集簇。

2.3. 创建一个新表

你可以通过指定表的名字和所有列的名字及其类型来创建表：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- 最低温度
    temp_hi         int,           -- 最高温度
```

```
    prcp            real,           -- 湿度
    date            date
);
```

你可以在psql输入这些命令以及换行符。psql可以识别该命令直到分号才结束。

你可以在 SQL 命令中自由使用空白（即空格、制表符和换行符）。 这就意味着你可以用和上面不同的对齐方式键入命令，或者将命令全部放在一行中。两个划线（“--”）引入注释。 任何跟在它后面直到行尾的东西都会被忽略。SQL 是对关键字和标识符大小写不敏感的语言，只有在标识符用双引号包围时才能保留它们的大小写（上例没有这么做）。

varchar(80)指定了一个可以存储最长 80 个字符的任意字符串的数据类型。int是普通的整数类型。real是一种用于存储单精度浮点数的类型。date类型应该可以自解释（没错，类型为date的列名字也是date。 这么做可能比较方便或者容易让人混淆 — 你自己选择）。

PostgreSQL支持标准的SQL类型int、smallint、real、double precision、char(N)、varchar(N)、date、time、timestamp和interval，还支持其他的通用功能的类型和丰富的几何类型。PostgreSQL中可以定制任意数量的用户定义数据类型。因而类型名并不是语法关键字，除了SQL标准要求支持的特例外。

第二个例子将保存城市和它们相关的地理位置：

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

类型point就是一种PostgreSQL特有数据类型的例子。

最后，我们还要提到如果你不再需要某个表，或者你想以不同的形式重建它，那么你可以用下面的命令删除它：

```
DROP TABLE tablename;
```

2.4. 在表中增加行

INSERT语句用于向表中添加行：

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

请注意所有数据类型都使用了相当明了的输入格式。那些不是简单数字值的常量通常必需用单引号（'）包围，就象在例子里一样。date类型实际上对可接收的格式相当灵活，不过在本教程里，我们应该坚持使用这种清晰的格式。

point类型要求一个座标对作为输入，如下：

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

到目前为止使用的语法要求你记住列的顺序。一个可选的语法允许你明确地列出列：

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

如果你需要，你可以用另外一个顺序列出列或者是忽略某些列， 比如说，我们不知道降水量：

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

许多开发人员认为明确列出列要比依赖隐含的顺序是更好的风格。

请输入上面显示的所有命令，这样你在随后的各节中才有可用的数据。

你还可以使用COPY从文本文件中装载大量数据。这种方式通常更快，因为COPY命令就是为这类应用优化的， 只是比 INSERT少一些灵活性。比如：

```
COPY weather FROM '/home/user/weather.txt';
```

这里源文件的文件名必须在运行后端进程的机器上是可用的， 而不是在客户端上，因为后端进程将直接读取该文件。你可以在COPY中读到更多有关COPY命令的信息。

2.5. 查询一个表

要从一个表中检索数据就是查询这个表。SQL的SELECT语句就是做这个用途的。 该语句分为选择列表（列出要返回的列）、表列表（列出从中检索数据的表）以及可选的条件（指定任意的限制）。比如，要检索表weather的所有行，键入：

```
SELECT * FROM weather;
```

这里*是“所有列”的缩写。 [1] 因此相同的结果应该这样获得：

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

而输出应该是：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

你可以在选择列表中写任意表达式，而不仅仅是列的列表。比如，你可以：

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

这样应该得到：

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

请注意这里的AS子句是如何给输出列重新命名的（AS子句是可选的）。

[1] 虽然SELECT *对于即席查询很有用，但我们普遍认为在生产代码中这是很糟糕的风格，因为给表增加一个列就改变了结果。

一个查询可以使用WHERE子句“修饰”，它指定需要哪些行。WHERE子句包含一个布尔（真值）表达式，只有那些使布尔表达式为真的行才会被返回。在条件中可以使用常用的布尔操作符（AND、OR和NOT）。 比如，下面的查询检索旧金山的下雨天的天气：

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

结果：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

你可以要求返回的查询结果是排好序的：

```
SELECT * FROM weather
    ORDER BY city;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

在这个例子里，排序的顺序并未完全被指定，因此你可能看到属于旧金山的行被随机地排序。但是如果你使用下面的语句，那么就总是会得到上面的结果：

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

你可以要求在查询的结果中消除重复的行：

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

再次声明，结果行的顺序可能变化。你可以组合使用DISTINCT和ORDER BY来保证获取一致的结果：[2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. 在表之间连接

[2] 在一些数据库系统里，包括老版本的PostgreSQL，DISTINCT的实现自动对行进行排序，因此ORDER BY是多余的。但是这一点并不是 SQL 标准的要求，并且目前的PostgreSQL并不保证DISTINCT会导致行被排序。

到目前为止，我们的查询一次只访问一个表。查询可以一次访问多个表，或者用这种方式访问一个表而同时处理该表的多个行。　一个同时访问同一个或者不同表的多个行的查询叫连接查询。举例来说，比如你想列出所有天气记录以及相关的城市位置。要实现这个目标，我们需要拿 weather表每行的city列和cities表所有行的name列进行比较，　并选取那些在该值上相匹配的行对。

> **注意**
>
> 这里只是一个概念上的模型。连接通常以比实际比较每个可能的行对更高效的方式执行，　但这些是用户看不到的。

这个任务可以用下面的查询来实现：

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

```
     city      | temp_lo | temp_hi | prcp |    date    |     name      |
 location
---------------+---------+---------+------+------------+---------------
+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco |
 (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco |
 (-194,53)
(2 rows)
```

观察结果集的两个方面：

- 没有城市Hayward的结果行。这是因为在cities表里面没有Hayward的匹配行，所以连接忽略 weather表里的不匹配行。我们稍后将看到如何修补它。
- 有两个列包含城市名字。这是正确的，　因为weather和cities表的列被串接在一起。不过，实际上我们不想要这些，　因此你将可能希望明确列出输出列而不是使用*：

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

练习：.　看看这个查询省略WHERE子句的语义是什么

因为这些列的名字都不一样，所以规划器自动地找出它们属于哪个表。如果在两个表里有重名的列，你需要限定列名来说明你究竟想要哪一个，如：

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

人们广泛认为在一个连接查询中限定所有列名是一种好的风格，这样即使未来向其中一个表里添加重名列也不会导致查询失败。

到目前为止，这种类型的连接查询也可以用下面这样的形式写出来：

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

这个语法并不象上文的那个那么常用，我们在这里写出来是为了让你更容易了解后面的主题。

现在我们将看看如何能把Hayward记录找回来。我们想让查询干的事是扫描weather表， 并且对每一行都找出匹配的cities表行。如果我们没有找到匹配的行，那么我们需要一些“空值”代替cities表的列。 这种类型的查询叫外连接 （我们在此之前看到的连接都是内连接）。这样的命令看起来象这样：

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      |
 location
---------------+---------+---------+------+------------+---------------
+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco |
 (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco |
 (-194,53)
(3 rows)
```

这个查询是一个左外连接， 因为在连接操作符左部的表中的行在输出中至少要出现一次，而在右部的表的行只有在能找到匹配的左部表行时才被输出。 如果输出的左部表的行没有对应匹配的右部表的行，那么右部表行的列将填充空值（null）。

练习：. 还有右外连接和全外连接。试着找出来它们能干什么。

我们也可以把一个表和自己连接起来。这叫做自连接。 比如，假设我们想找出那些在其它天气记录的温度范围之外的天气记录。这样我们就需要拿 weather表里每行的temp_lo和temp_hi列与weather表里其它行的temp_lo和temp_hi列进行比较。我们可以用下面的查询实现这个目标：

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

在这里我们把weather表重新标记为W1和W2以区分连接的左部和右部。你还可以用这样的别名在其它查询里节约一些敲键，比如：

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

你以后会经常碰到这样的缩写的。

2.7. 聚集函数

和大多数其它关系数据库产品一样，PostgreSQL支持聚集函数。 一个聚集函数从多个输入行中计算出一个结果。 比如，我们有在一个行集合上计算count（计数）、sum（和）、avg（均值）、max（最大值）和min（最小值）的函数。

比如，我们可以用下面的语句找出所有记录中最低温度中的最高温度：

```
SELECT max(temp_lo) FROM weather;
```

```
 max
-----
  46
(1 row)
```

如果我们想知道该读数发生在哪个城市，我们可以用：

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     错误
```

不过这个方法不能运转，因为聚集max不能被用于WHERE子句中（存在这个限制是因为WHERE子句决定哪些行可以被聚集计算包括；因此显然它必需在聚集函数之前被计算）。不过，我们通常都可以用其它方法实现我们的目的；这里我们就可以使用子查询：

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```
     city
---------------
 San Francisco
(1 row)
```

这样做是 OK 的，因为子查询是一次独立的计算，它独立于外层的查询计算出自己的聚集。

聚集同样也常用于和GROUP BY子句组合。比如，我们可以获取每个城市观测到的最低温度的最高值：

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

这样给我们每个城市一个输出。每个聚集结果都是在匹配该城市的表行上面计算的。我们可以用HAVING 过滤这些被分组的行：

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
```

```
    HAVING max(temp_lo) < 40;
```

```
  city    | max
----------+-----
 Hayward |  37
(1 row)
```

这样就只给出那些所有temp_lo值曾都低于 40的城市。最后，如果我们只关心那些名字以“S”开头的城市，我们可以用：

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'            -- 1
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

1 LIKE操作符进行模式匹配，在第 9.7 节有解释。

理解聚集和SQL的WHERE以及HAVING子句之间的关系对我们非常重要。WHERE和HAVING的基本区别如下：WHERE在分组和聚集计算之前选取输入行（因此，它控制哪些行进入聚集计算），而HAVING在分组和聚集之后选取分组行。因此，WHERE子句不能包含聚集函数；因为试图用聚集函数判断哪些行应输入给聚集运算是没有意义的。相反，HAVING子句总是包含聚集函数（严格说来，你可以写不使用聚集的HAVING子句，但这样做很少有用。同样的条件用在WHERE阶段会更有效）。

在前面的例子里，我们可以在WHERE里应用城市名称限制，因为它不需要聚集。这样比放在HAVING里更加高效，因为可以避免那些未通过 WHERE检查的行参与到分组和聚集计算中。

2.8. 更新

你可以用UPDATE命令更新现有的行。假设你发现所有 11 月 28 日以后的温度读数都低了两度，那么你就可以用下面的方式改正数据：

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

看看数据的新状态：

```
SELECT * FROM weather;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. 删除

数据行可以用DELETE命令从表中删除。假设你对Hayward的天气不再感兴趣，那么你可以用下面的方法把那些行从表中删除：

```
DELETE FROM weather WHERE city = 'Hayward';
```

所有属于Hayward的天气记录都被删除。

```
SELECT * FROM weather;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

我们用下面形式的语句的时候一定要小心

```
DELETE FROM tablename;
```

如果没有一个限制，DELETE将从指定表中删除所有行，把它清空。做这些之前系统不会请求你确认！

第 3 章 高级特性

3.1. 简介

在之前的章节里我们已经涉及了使用SQL在PostgreSQL中存储和访问数据的基础知识。现在我们将要讨论SQL中一些更高级的特性，这些特性有助于简化管理和防止数据丢失或损坏。最后，我们还将介绍一些PostgreSQL扩展。

本章有时将引用第 2 章的例子并对其进行改变或改进以便于阅读本章。本章中的某些例子可以在教程目录的advanced.sql文件中找到。该文件也包含一些样例数据，在这里就不在赘述（查看第 2.1 节解如何使用该文件）。

3.2. 视图

回想一下第 2.6 节的查询。假设天气记录和城市位置的组合列表对我们的应用有用，但我们又不想每次需要使用它时都敲入整个查询。我们可以在该查询上创建一个视图，这会给该查询一个名字，我们可以像使用一个普通表一样来使用它：

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

对视图的使用是成就一个好的SQL数据库设计的关键方面。视图允许用户通过始终如一的接口封装表的结构细节，这样可以避免表结构随着应用的进化而改变。

视图几乎可以用在任何可以使用表的地方。在其他视图基础上创建视图也并不少见。

3.3. 外键

回想第2章中的weather和cities表。考虑以下问题：我们希望确保在cities表中有相应项之前任何人都不能在weather表中插入行。这叫做维持数据的引用完整性。在过分简化的数据库系统中，可以通过先检查cities表中是否有匹配的记录存在，然后决定应该接受还是拒绝即将插入weather表的行。这种方法有一些问题且并不方便，于是PostgreSQL可以为我们来解决：

新的表定义如下：

```
CREATE TABLE cities (
        city     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

现在尝试插入一个非法的记录：

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');


ERROR:  insert or update on table "weather" violates foreign key constraint
 "weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

外键的行为可以很好地根据应用来调整。我们不会在这个教程里更深入地介绍，读者可以参考第 5 章的信息。正确使用外键无疑会提高数据库应用的质量，因此强烈建议用户学会如何使用它们。

3.4. 事务

事务是所有数据库系统的基础概念。事务最重要的一点是它将多个步骤捆绑成了一个单一的、要么全完成要么全不完成的操作。步骤之间的中间状态对于其他并发事务是不可见的，并且如果有某些错误发生导致事务不能完成，则其中任何一个步骤都不会对数据库造成影响。

例如，考虑一个保存着多个客户账户余额和支行总存款额的银行数据库。假设我们希望记录一笔从Alice的账户到Bob的账户的额度为100.00美元的转账。在最大程度地简化后，涉及到的SQL命令是：

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

这些命令的细节在这里并不重要，关键点是为了完成这个相当简单的操作涉及到多个独立的更新。我们的银行职员希望确保这些更新要么全部发生，或者全部不发生。当然不能发生因为系统错误导致Bob收到100美元而Alice并未被扣款的情况。Alice当然也不希望自己被扣款而Bob没有收到钱。我们需要一种保障，当操作中途某些错误发生时已经执行的步骤不会产生效果。将这些更新组织成一个事务就可以给我们这种保障。一个事务被称为是原子的：从其他事务的角度来看，它要么整个发生要么完全不发生。

我们同样希望能保证一旦一个事务被数据库系统完成并认可，它就被永久地记录下来且即便其后发生崩溃也不会被丢失。例如，如果我们正在记录Bob的一次现金提款，我们当然不希望他刚走出银行大门，对他账户的扣款就消失。一个事务型数据库保证一个事务在被报告为完成之前它所做的所有更新都被记录在持久存储（即磁盘）。

事务型数据库的另一个重要性质与原子更新的概念紧密相关：当多个事务并发运行时，每一个都不能看到其他事务未完成的修改。例如，如果一个事务正忙着总计所有支行的余额，它不会只包括Alice的支行的扣款而不包括Bob的支行的存款，或者反之。所以事务的全做或全不做并不只体现在它们对数据库的持久影响，也体现在它们发生时的可见性。一个事务所做的更新在它完成之前对于其他事务是不可见的，而之后所有的更新将同时变得可见。

在PostgreSQL中，开启一个事务需要将SQL命令用BEGIN和COMMIT命令包围起来。因此我们的银行事务看起来会是这样：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
```

```
COMMIT;
```

如果，在事务执行中我们并不想提交（或许是我们注意到Alice的余额不足），我们可以发出ROLLBACK命令而不是COMMIT命令，这样所有目前的更新将会被取消。

PostgreSQL实际上将每一个SQL语句都作为一个事务来执行。如果我们没有发出BEGIN命令，则每个独立的语句都会被加上一个隐式的BEGIN以及（如果成功）COMMIT来包围它。一组被BEGIN和COMMIT包围的语句也被称为一个事务块。

> **注意**
>
> 某些客户端库会自动发出BEGIN和COMMIT命令，因此我们可能会在不被告知的情况下得到事务块的效果。具体请查看所使用的接口文档。

也可以利用保存点来以更细的粒度来控制一个事务中的语句。保存点允许我们有选择性地放弃事务的一部分而提交剩下的部分。在使用SAVEPOINT定义一个保存点后，我们可以在必要时利用ROLLBACK TO回滚到该保存点。该事务中位于保存点和回滚点之间的数据库修改都会被放弃，但是早于该保存点的修改则会被保存。

在回滚到保存点之后，它的定义依然存在，因此我们可以多次回滚到它。反过来，如果确定不再需要回滚到特定的保存点，它可以被释放以便系统释放一些资源。记住不管是释放保存点还是回滚到保存点都会释放定义在该保存点之后的所有其他保存点。

所有这些都发生在一个事务块内，因此这些对于其他数据库会话都不可见。当提交整个事务块时，被提交的动作将作为一个单元变得对其他会话可见，而被回滚的动作则永远不会变得可见。

记住那个银行数据库，假设我们从Alice的账户扣款100美元，然后存款到Bob的账户，结果直到最后才发现我们应该存到Wally的账户。我们可以通过使用保存点来做这件事：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

当然，这个例子是被过度简化的，但是在一个事务块中使用保存点存在很多种控制可能性。此外，ROLLBACK TO是唯一的途径来重新控制一个由于错误被系统置为中断状态的事务块，而不是完全回滚它并重新启动。

3.5. 窗口函数

一个窗口函数在一系列与当前行有某种关联的表行上执行一种计算。这与一个聚集函数所完成的计算有可比之处。但是窗口函数并不会使多行被聚集成一个单独的输出行，这与通常的非窗口聚集函数不同。取而代之，行保留它们独立的标识。在这些现象背后，窗口函数可以访问的不仅仅是查询结果的当前行。

下面是一个例子用于展示如何将每一个员工的薪水与他/她所在部门的平均薪水进行比较：

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM
 empsalary;


  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

最开始的三个输出列直接来自于表empsalary，并且表中每一行都有一个输出行。第四列表示对与当前行具有相同depname值的所有表行取得平均值（这实际和非窗口avg聚集函数是相同的函数，但是OVER子句使得它被当做一个窗口函数处理并在一个合适的窗口帧上计算。）。

一个窗口函数调用总是包含一个直接跟在窗口函数名及其参数之后的OVER子句。这使得它从句法上和一个普通函数或非窗口函数区分开来。OVER子句决定究竟查询中的哪些行被分离出来由窗口函数处理。OVER子句中的PARTITION BY子句指定了将具有相同PARTITION BY表达式值的行分到组或者分区。对于每一行，窗口函数都会在当前行同一分区的行上进行计算。

我们可以通过OVER上的ORDER BY控制窗口函数处理行的顺序（窗口的ORDER BY并不一定要符合行输出的顺序。）。下面是一个例子：

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;


  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

如上所示，rank函数在当前行的分区内按照ORDER BY子句的顺序为每一个可区分的ORDER BY值产生了一个数字等级。rank不需要显式的参数，因为它的行为完全决定于OVER子句。

一个窗口函数所考虑的行属于那些通过查询的FROM子句产生并通过WHERE、GROUP BY、HAVING过滤的“虚拟表”。例如，一个由于不满足WHERE条件被删除的行是不会被任何窗口函数所见的。在一个查询中可以包含多个窗口函数，每个窗口函数都可以用不同的OVER子句来按不同方式划分数据，但是它们都作用在由虚拟表定义的同一个行集上。

我们已经看到如果行的顺序不重要时ORDER BY可以忽略。PARTITION BY同样也可以被忽略，在这种情况下会产生一个包含所有行的分区。

这里有一个与窗口函数相关的重要概念：对于每一行，在它的分区中的行集被称为它的窗口帧。 一些窗口函数只作用在窗口帧中的行上，而不是整个分区。默认情况下，如果使用ORDER BY，则帧包括从分区开始到当前行的所有行，以及后续任何与当前行在ORDER BY子句上相等的行。如果ORDER BY被忽略，则默认帧包含整个分区中所有的行。 [1] 下面是使用sum的例子：

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

如上所示，由于在OVER子句中没有ORDER BY，窗口帧和分区一样，而如果缺少PARTITION BY则和整个表一样。换句话说，每个合计都会在整个表上进行，这样我们为每一个输出行得到的都是相同的结果。但是如果我们加上一个ORDER BY子句，我们会得到非常不同的结果：

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

这里的合计是从第一个（最低的）薪水一直到当前行，包括任何与当前行相同的行（注意相同薪水行的结果）。

窗口函数只允许出现在查询的SELECT列表和ORDER BY子句中。它们不允许出现在其他地方，例如GROUP BY、HAVING和WHERE子句中。这是因为窗口函数的执行逻辑是在处理完这些子句之后。另外，窗口函数在非窗口聚集函数之后执行。这意味着可以在窗口函数的参数中包括一个聚集函数，但反过来不行。

如果需要在窗口计算执行后进行过滤或者分组，我们可以使用子查询。例如：

```
SELECT depname, empno, salary, enroll_date
```

[1] 还有些选项用于以其他方式定义窗口帧，但是这不包括在本教程内。详见第 4.2.8 节

```
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

上述查询仅仅显示了内层查询中rank低于3的结果。

当一个查询涉及到多个窗口函数时，可以将每一个分别写在一个独立的OVER子句中。但如果多个函数要求同一个窗口行为时，这种做法是冗余的而且容易出错的。替代方案是，每一个窗口行为可以被放在一个命名的WINDOW子句中，然后在OVER中引用它。例如：

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

关于窗口函数的更多细节可以在第 4.2.8 节第 9.21 节第 7.2.5 节以及SELECT参考页中找到。

3.6. 继承

继承是面向对象数据库中的概念。它展示了数据库设计的新的可能性。

让我们创建两个表：表cities和表capitals。自然地，首都也是城市，所以我们需要有某种方式能够在列举所有城市的时候也隐式地包含首都。如果真的聪明，我们会设计如下的模式：

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,     -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int      -- (in ft)
);

CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

这个模式对于查询而言工作正常，但是当我们需要更新一些行时它就变得不好用了。

更好的方案是：

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int      -- (in ft)
);
```

```
CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

在这种情况下，一个capitals的行从它的父亲cities继承了所有列（name、population和altitude）。列name的类型是text，一种用于变长字符串的本地PostgreSQL类型。州首都有一个附加列state用于显示它们的州。在PostgreSQL中，一个表可以从0个或者多个表继承。

例如，如下查询可以寻找所有海拔500尺以上的城市名称，包括州首都：

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

它的返回为：

```
   name     | altitude
------------+----------
 Las Vegas  |     2174
 Mariposa   |     1953
 Madison    |      845
(3 rows)
```

在另一方面，下面的查询可以查找所有海拔高于500尺且不是州首府的城市：

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;
```

```
   name     | altitude
------------+----------
 Las Vegas  |     2174
 Mariposa   |     1953
(2 rows)
```

其中cities之前的ONLY用于指示查询只在cities表上进行而不会涉及到继承层次中位于cities之下的其他表。很多我们已经讨论过的命令 — SELECT、UPDATE 和DELETE — 都支持这个ONLY记号。

> **注意**
>
> 尽管继承很有用，但是它还未与唯一约束或外键集成，这也限制了它的可用性。更多详情见第 5.9 节

3.7. 小结

PostgreSQL中有很多特性在这个面向SQL新用户的教程中并未触及。有关这些特性的更多详情将在本书的后续部分进行讨论。

如果需要更多介绍材料，请访问 PostgreSQL 官方网站[2]来获得更多资源链接。

[2] https://www.postgresql.org

部分 II. SQL 语言

这部份描述在PostgreSQL中SQL语言的使用。我们从描述SQL的一般语法开始，然后解释如何创建保存数据的结构、如何填充数据库以及如何查询它。中间的部分列出了在SQL命令中可用的数据类型和函数。剩余的部分则留给对于调优数据性能的重要方面。

这部份的信息被组织成让一个新用户可以从头到尾跟随它来全面理解主题，而不需要多次参考后面的内容。这些章都是自包含的，这样高级用户可以根据他们的选择阅读单独的章。这一部分的信息被以一种叙事的风格展现。需要查看一个特定命令的完整描述的读者应该去看看第 VI 部分

这一部分的阅读者应该知道如何连接到一个PostgreSQL数据库并且发出SQL命令。我们鼓励不熟悉这些问题的读者先去阅读第 I 部分SQL通常使用PostgreSQL的交互式终端psql输入，但是其他具有相似功能的程序也可以被使用。

目录

第 4 章 SQL语法

这一章描述了SQL的语法。它构成了理解后续具体介绍如何使用SQL定义和修改数据的章节的基础 。

我们同时建议已经熟悉SQL的用户仔细阅读本章，因为本章包含一些在SQL数据库中实现得不一致的以及PostgreSQL中特有的规则和概念。

4.1. 词法结构

SQL输入由一个命令序列组成。一个命令由一个记号的序列构成，并由一个分号（“;”）终结。输入流的末端也会标志一个命令的结束。具体哪些记号是合法的与具体命令的语法有关。

一个记号可以是一个关键词、一个标识符、一个带引号的标识符、一个literal（或常量）或者一个特殊字符符号。记号通常以空白（空格、制表符、新行）来分隔，但在无歧义时并不强制要求如此（唯一的例子是一个特殊字符紧挨着其他记号）。

例如，下面是一个（语法上）合法的SQL输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这是一个由三个命令组成的序列，每一行一个命令（尽管这不是必须地，在同一行中可以有超过一个命令，而且命令还可以被跨行分割）。

另外，注释也可以出现在SQL输入中。它们不是记号，它们和空白完全一样。

根据标识命令、操作符、参数的记号不同，SQL的语法不很一致。最前面的一些记号通常是命令名，因此在上面的例子中我们通常会说一个“SELECT”、一个“UPDATE”和一个“INSERT”命令。但是例如UPDATE命令总是要求一个SET记号出现在一个特定位置，而INSERT则要求一个VALUES来完成命令。每个命令的精确语法规则在第 VI 部分介绍。

4.1.1. 标识符和关键词

上例中的SELECT、UPDATE或VALUES记号是关键词的例子，即SQL语言中具有特定意义的词。记号MY_TABLE和A则是标识符的例子。它们标识表、列或者其他数据库对象的名字，取决于使用它们的命令。因此它们有时也被简称为“名字”。关键词和标识符具有相同的词法结构，这意味着我们无法在没有语言知识的前提下区分一个标识符和关键词。一个关键词的完整列表可以在附录 C中找到。

SQL标识符和关键词必须以一个字母（a-z，也可以是带变音符的字母和非拉丁字母）或一个下划线（_）开始。后续字符可以是字母、下划线（_）、数字（0-9）或美元符号（$）。注意根据SQL标准的字母规定，美元符号是不允许出现在标识符中的，因此它们的使用可能会降低应用的可移植性。SQL标准不会定义包含数字或者以下划线开头或结尾的关键词，因此这种形式的标识符不会与未来可能的标准扩展冲突 。

系统中一个标识符的长度不能超过 NAMEDATALEN-1 字节，在命令中可以写超过此长度的标识符，但是它们会被截断。默认情况下，NAMEDATALEN 的值为64，因此标识符的长度上限为63字节。如果这个限制有问题，可以在src/include/pg_config_manual.h中修改 NAMEDATALEN 常量。

关键词和不被引号修饰的标识符是大小写不敏感的。因此：

```
UPDATE MY_TABLE SET A = 5;
```

可以等价地写成：

```
uPDaTE my_TabLE SeT a = 5;
```

一个常见的习惯是将关键词写成大写，而名称写成小写，例如：

```
UPDATE my_table SET a = 5;
```

这里还有第二种形式的标识符：受限标识符或被引号修饰的标识符。它是由双引号（"）包围的一个任意字符序列。一个受限标识符总是一个标识符而不会是一个关键字。因此"select"可以用于引用一个名为“select”的列或者表，而一个没有引号修饰的select则会被当作一个关键词，从而在本应使用表或列名的地方引起解析错误。在上例中使用受限标识符的例子如下：

```
UPDATE "my_table" SET "a" = 5;
```

受限标识符可以包含任何字符，除了代码为0的字符（如果要包含一个双引号，则写两个双引号）。这使得可以构建原本不被允许的表或列的名称，例如包含空格或花号的名字。但是长度限制依然有效。

一种受限标识符的变体允许包括转义的用代码点标识的Unicode字符。这种变体以U&（大写或小写U跟上一个花号）开始，后面紧跟双引号修饰的名称，两者之间没有任何空白，如U&"foo"（注意这里与操作符&似乎有一些混淆，但是在&操作符周围使用空白避免了这个问题）。在引号内，Unicode字符可以以转义的形式指定：反斜线接上4位16进制代码点号码或者反斜线和加号接上6位16进制代码点号码。例如，标识符"data"可以写成：

```
U&"d\0061t\+000061"
```

下面的例子用斯拉夫语字母写出了俄语单词 “slon”（大象）：

```
U&"\0441\043B\043E\043D"
```

如果希望使用其他转义字符来代替反斜线，可以在字符串后使用UESCAPE子句，例如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

转义字符可以是除了16进制位、加号、单引号、双引号、空白字符之外的任意单个字符。注意转义字符是被写在单引号而不是双引号内。

为了在标识符中包括转义字符本身，将其写两次即可。

Unicode转义语法只有在服务器编码为UTF8时才起效。当使用其他服务器编码时，只有在ASCII范围内（最高到\007F）的编码点才能被使用。4位和6位形式都可以被用来定义UTF-16代理对来组成代码点大于U+FFFF的字符，尽管6位形式的存在使得这种做法变得不必要（代理对并不被直接存储，而是被被绑定到一个单独的代码点然后被编码到UTF-8）。

将一个标识符变得受限同时也使它变成大小写敏感的，反之非受限名称总是被转换成小写形式。例如，标识符FOO、foo和"foo"在PostgreSQL中被认为是相同的，而"Foo"和"FOO"则互不相同且也不同于前面三个标识符（PostgreSQL将非受限名字转换为小写形式与SQL标准是不兼容 的，SQL标准中要求将非受限名称转换为大写形式。这样根据标准， foo应该和 "FOO"而不是"foo"相同。如果希望写一个可移植的应用，我们应该总是用引号修饰一个特定名字或者 从不使用 引号修饰）。

4.1.2. 常量

在PostgreSQL中有三种隐式类型常量：字符串、位串和数字。常量也可以被指定显示类型，这可以使得它被更精确地展示以及更有效地处理。这些选择将会在后续小节中讨论。

4.1.2.1. 字符串常量

在SQL中，一个字符串常量是一个由单引号（'）包围的任意字符序列，例如'This is a string'。为了在一个字符串中包括一个单引号，可以写两个相连的单引号，例如'Dianne''s horse'。注意这和一个双引号（"）不同。

两个只由空白及至少一个新行分隔的字符串常量会被连接在一起，并且将作为一个写在一起的字符串常量来对待。例如：

```
SELECT 'foo'
'bar';
```

等同于：

```
SELECT 'foobar';
```

但是：

```
SELECT 'foo'      'bar';
```

则不是合法的语法（这种有些奇怪的行为是SQL指定的，PostgreSQL遵循了该标准）。

4.1.2.2. C风格转义的字符串常量

PostgreSQL也接受“转义”字符串常量，这也是SQL标准的一个扩展。一个转义字符串常量可以通过在开单引号前面写一个字母E（大写或小写形式）来指定，例如E'foo'（当一个转义字符串常量跨行时，只在第一个开引号之前写E）。在一个转义字符串内部，一个反斜线字符（\）会开始一个 C 风格的反斜线转义序列，在其中反斜线和后续字符的组合表示一个特殊的字节值（如表 4.1中所示）。

表 4.1. 反斜线转义序列

反斜线转义序列	解释
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
\o, \oo, \ooo (o = 0 - 7)	八进制字节值
\xh, \xhh (h = 0 - 9, A - F)	十六进制字节值
\uxxxx, \Uxxxxxxxx (x = 0 - 9, A - F)	16 或 32-位十六进制 Unicode 字符值

跟随在一个反斜线后面的任何其他字符被当做其字面意思。因此，要包括一个反斜线字符，请写两个反斜线（\\）。在一个转义字符串中包括一个单引号除了普通方法''之外，还可以写成\'。

你要负责保证你创建的字节序列由服务器字符集编码中合法的字符组成，特别是在使用八进制或十六进制转义时。如果服务器编码为 UTF-8，那么应该使用 Unicode 转义或替代的

Unicode 转义语法（在第 4.1.2.3 节解释）。替代方案可能是手工写出 UTF-8 编码字节，这可能会非常麻烦。

只有当服务器编码是UTF8时，Unicode 转义语法才能完全工作。当使用其他服务器编码时，只有在 ASCII 范围（低于\u007F）内的代码点能够被指定。4 位和 8 位形式都能被用来指定 UTF-16 代理对，用来组成代码点超过 U+FFFF 的字符，不过 8 位形式的可用从技术上使得这种做法不再是必须的（当服务器编码为UTF8并使用代理对时，它们首先被结合到一个单一代码点，然后会被用 UTF-8 编码）。

> **小心**
>
> 如果配置参数standard_conforming_strings为off，那么PostgreSQL对常规字符串常量和转义字符串常量中的反斜线转义都识别。不过，从PostgreSQL 9.1开始，该参数的默认值为on，意味着只在转义字符串常量中识别反斜线转义。这种行为更兼容标准，但是可能打断依赖于历史行为（反斜线转义总是会被识别）的应用。作为一种变通，你可以设置该参数为off，但是最好迁移到符合新的行为。如果你需要使用一个反斜线转义来表示一个特殊字符，为该字符串常量写上一个E。
>
> 在standard_conforming_strings之外，配置参数escape_string_warning和backslash_quote也决定了如何对待字符串常量中的反斜线。

代码零的字符不能出现在一个字符串常量中。

4.1.2.3. 带有 Unicode 转义的字符串常量

PostgreSQL也支持另一种类型的字符串转义语法，它允许用代码点指定任意 Unicode 字符。一个 Unicode 转义字符串常量开始于U&（大写或小写形式的字母 U，后跟花号），后面紧跟着开引号，之间没有任何空白，例如U&'foo'（注意这产生了与操作符&的混淆。在操作符周围使用空白来避免这个问题）。在引号内，Unicode 字符可以通过写一个后跟 4 位十六进制代码点编号或者一个前面有加号的 6 位十六进制代码点编号的反斜线来指定。例如，字符串'data'可以被写为

```
U&'d\0061t\+000061'
```

下面的例子用斯拉夫字母写出了俄语的单词“slon”（大象）：

```
U&'\0441\043B\043E\043D'
```

如果想要一个不是反斜线的转义字符，可以在字符串之后使用UESCAPE子句来指定，例如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

转义字符可以是出一个十六进制位、加号、单引号、双引号或空白字符之外的任何单一字符。

只有当服务器编码是UTF8时，Unicode 转义语法才能完全工作。当使用其他服务器编码时，只有在 ASCII 范围（低于\u007F）内的代码点能够被指定。4 位和 8 位形式都能被用来指定 UTF-16 代理对，用来组成代码点超过 U+FFFF 的字符，不过 8 位形式的可用从技术上使得这种做法不再是必须的（当服务器编码为UTF8并使用代理对时，它们首先被结合到一个单一代码点，然后会被用 UTF-8 编码）。

还有，只有当配置参数standard_conforming_strings被打开时，用于字符串常量的 Unicode 转义语法才能工作。这是因为否则这种语法将迷惑客户端中肯地解析 SQL 语句，进

而会导致 SQL 注入以及类似的安全性问题。如果这个参数被设置为关闭，这种语法将被拒绝并且报告一个错误消息。

要在一个字符串中包括一个表示其字面意思的转义字符，把它写两次。

4.1.2.4. 美元引用的字符串常量

虽然用于指定字符串常量的标准语法通常都很方便，但是当字符串中包含了很多单引号或反斜线时很难理解它，因为每一个都需要被双写。要在这种情形下允许可读性更好的查询，PostgreSQL提供了另一种被称为“美元引用”的方式来书写字符串常量。一个美元引用的字符串常量由一个美元符号（$）、一个可选的另个或更多字符的“标签”、另一个美元符号、一个构成字符串内容的任意字符序列、一个美元符号、开始这个美元引用的相同标签和一个美元符号组成。例如，这里有两种不同的方法使用美元引用指定字符串“Dianne's horse”：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

注意在美元引用字符串中，单引号可以在不被转义的情况下使用。事实上，在一个美元引用字符串中不需要对字符进行转义：字符串内容总是按其字面意思写出。反斜线不是特殊的，并且美元符号也不是特殊的，除非它们是匹配开标签的一个序列的一部分。

可以通过在每一个嵌套级别上选择不同的标签来嵌套美元引用字符串常量。这最常被用在编写函数定义上。例如：

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列q[\t\r\n\v\\]q表示一个美元引用的文字串[\t\r\n\v\\]，当该函数体被PostgreSQL执行时它将被识别。但是因为该序列不匹配外层的美元引用的定界符$function$，它只是一些在外层字符串所关注的常量中的字符而已。

一个美元引用字符串的标签（如果有）遵循一个未被引用标识符的相同规则，除了它不能包含一个美元符号之外。标签是大小写敏感的，因此tagString contenttag是正确的，但是TAGString contenttag不正确。

一个跟着一个关键词或标识符的美元引用字符串必须用空白与之分隔开，否则美元引用定界符可能会被作为前面标识符的一部分。

美元引用不是 SQL 标准的一部分，但是在书写复杂字符串文字方面，它常常是一种比兼容标准的单引号语法更方便的方法。当要表示的字符串常量位于其他常量中时它特别有用，这种情况常常在过程函数定义中出现。如果用单引号语法，上一个例子中的每个反斜线将必须被写成四个反斜线，这在解析原始字符串常量时会被缩减到两个反斜线，并且接着在函数执行期间重新解析内层字符串常量时变成一个。

4.1.2.5. 位串常量

位串常量看起来像常规字符串常量在开引号之前（中间无空白）加了一个B（大写或小写形式），例如B'1001'。位串常量中允许的字符只有0和1。

作为一种选择，位串常量可以用十六进制记号法指定，使用一个前导X（大写或小写形式），例如X'1FF'。这种记号法等价于一个用四个二进制位取代每个十六进制位的位串常量。

两种形式的位串常量可以以常规字符串常量相同的方式跨行继续。美元引用不能被用在位串常量中。

4.1.2.6. 数字常量

在这些一般形式中可以接受数字常量：

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

其中digits是一个或多个十进制数字（0 到 9）。如果使用了小数点，在小数点前面或后面必须至少有一个数字。如果存在一个指数标记（e），在其后必须跟着至少一个数字。在该常量中不能嵌入任何空白或其他字符。注意任何前导的加号或减号并不实际被考虑为常量的一部分，它是一个应用到该常量的操作符。

这些是合法数字常量的例子：

```
42
3.5
4.
.001
5e2
1.925e-3
```

如果一个不包含小数点和指数的数字常量的值适合类型integer（32 位），它首先被假定为类型integer。否则如果它的值适合类型bigint（64 位），它被假定为类型bigint。再否则它会被取做类型numeric。包含小数点和/或指数的常量总是首先被假定为类型numeric。

一个数字常量初始指派的数据类型只是类型转换算法的一个开始点。在大部分情况中，常量将被根据上下文自动被强制到最合适的类型。必要时，你可以通过造型它来强制一个数字值被解释为一种指定数据类型。例如，你可以这样强制一个数字值被当做类型real（float4）：

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

这些实际上只是接下来要讨论的一般造型记号的特例。

4.1.2.7. 其他类型的常量

一种任意类型的一个常量可以使用下列记号中的任意一种输入：

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

字符串常量的文本被传递到名为type的类型的输入转换例程中。其结果是指定类型的一个常量。如果对该常量的类型没有歧义（例如，当它被直接指派给一个表列时），显式类型造型可以被忽略，在那种情况下它会被自动强制。

字符串常量可以使用常规 SQL 记号或美元引用书写。

也可以使用一个类似函数的语法来指定一个类型强制：

```
typename ( 'string' )
```

但是并非所有类型名都可以用在这种方法中，详见第 4.2.9 节

如第 4.2.9 节讨论的，::、CAST()以及函数调用语法也可以被用来指定任意表达式的运行时类型转换。要避免语法歧义，type 'string'语法只能被用来指定简单文字常量的类型。type 'string'语法上的另一个限制是它无法对数组类型工作，指定一个数组常量的类型可使用::或CAST()。

CAST()语法符合 SQL。type 'string'语法是该标准的一般化：SQL 指定这种语法只用于一些数据类型，但是PostgreSQL允许它用于所有类型。带有::的语法是PostgreSQL的历史用法，就像函数调用语法一样。

4.1.3. 操作符

一个操作符名是最多NAMEDATALEN-1（默认为 63）的一个字符序列，其中的字符来自下面的列表：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

不过，在操作符名上有一些限制：

- -- and /*不能在一个操作符名的任何地方出现，因为它们将被作为一段注释的开始。
- 一个多字符操作符名不能以+或-结尾，除非该名称也至少包含这些字符中的一个：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，@-是一个被允许的操作符名，但*-不是。这些限制允许PostgreSQL解析 SQL 兼容的查询而不需要在记号之间有空格。

当使用非 SQL 标准的操作符名时，你通常需要用空格分隔相邻的操作符来避免歧义。例如，如果你定义了一个名为@的左一元操作符，你不能写X*@Y，你必须写X* @Y来确保PostgreSQL把它读作两个操作符名而不是一个。

4.1.4. 特殊字符

一些不是数字字母的字符有一种不同于作为操作符的特殊含义。这些字符的详细用法可以在描述相应语法元素的地方找到。这一节只是为了告知它们的存在以及总结这些字符的目的。

- 跟随在一个美元符号（$）后面的数字被用来表示在一个函数定义或一个预备语句中的位置参数。在其他上下文中该美元符号可以作为一个标识符或者一个美元引用字符串常量的一部分。
- 圆括号（()）具有它们通常的含义，用来分组表达式并且强制优先。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号（[]）被用来选择一个数组中的元素。更多关于数组的信息见第 8.15 节
- 逗号（,）被用在某些语法结构中来分割一个列表的元素。
- 分号（;）结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个被引用的标识符中。
- 冒号（:）被用来从数组中选择“切片”（见第 8.15 节。在某些 SQL 的“方言”（例如嵌入式 SQL）中，冒号被用来作为变量名的前缀。
- 星号（*）被用在某些上下文中标记一个表的所有域或者组合值。当它被用作一个聚集函数的参数时，它还有一种特殊的含义，即该聚集不要求任何显式参数。
- 句点（.）被用在数字常量中，并且被用来分割模式、表和列名。

4.1.5. 注释

一段注释是以双斜线开始并且延伸到行结尾的一个字符序列，例如：

```
-- This is a standard SQL comment
```

另外，也可以使用 C 风格注释块：

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

这里该注释开始于/*并且延伸到匹配出现的*/。这些注释块可按照 SQL 标准中指定的方式嵌套，但和 C 中不同。这样我们可以注释掉一大段可能包含注释块的代码。

在进一步的语法分析前，注释会被从输入流中被移除并且实际被替换为空白。

4.1.6. 操作符优先级

表 4.2显示了PostgreSQL中操作符的优先级和结合性。大部分操作符具有相同的优先并且是左结合的。操作符的优先级和结合性被硬写在解析器中。

此外，当使用二元和一元操作符的组合时，有时你将需要增加圆括号。例如：

```
SELECT 5 ! - 6;
```

将被解析为：

```
SELECT 5 ! (- 6);
```

因为解析器不知道 — 知道时就为时已晚 — !被定义为一个后缀操作符而不是一个中缀操作符。在这种情况下要得到想要的行为，你必须写成：

```
SELECT (5 !) - 6;
```

只是为了扩展性必须付出的代价。

表 4.2. 操作符优先级（从高到低）

操作符/元素	结合性	描述
.	左	表/列名分隔符
::	左	PostgreSQL-风格的类型转换
[ ]	左	数组元素选择
+ -	右	一元加、一元减
^	左	指数
* / %	左	乘、除、模
+ -	左	加、减
（任意其他操作符）	左	所有其他本地以及用户定义的操作符
BETWEEN IN LIKE ILIKE SIMILAR		范围包含、集合成员关系、字符串匹配

操作符/元素	结合性	描述
< > = <= >= <>		比较操作符
IS ISNULL NOTNULL		IS TRUE、IS FALSE、IS NULL、IS DISTINCT FROM等
NOT	右	逻辑否定
AND	左	逻辑合取
OR	左	逻辑析取

注意该操作符有限规则也适用于与上述内建操作符具有相同名称的用户定义的操作符。例如，如果你为某种自定义数据类型定义了一个“+”操作符，它将具有和内建的“+”操作符相同的优先级，不管你的操作符要做什么。

当一个模式限定的操作符名被用在OPERATOR语法中时，如下面的例子：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

OPERATOR结构被用来为“任意其他操作符”获得表 4.2中默认的优先级。不管出现在OPERATOR()中的是哪个指定操作符，这都是真的。

> **注意**
>
> 版本 9.5 之前的PostgreSQL使用的操作符优先级 规则略有不同。特别是，<=、>= 和<>习惯于被当作普通操作符，IS 测试习惯于具有较高的优先级。并且在一些认为NOT比 BETWEEN优先级高的情况下，NOT BETWEEN 和相关的结构的行为不一致。为了更好地兼容 SQL 标准并且减少对 逻辑上等价的结构不一致的处理，这些规则也得到了修改。在大部分情况下， 这些变化不会导致行为上的变化，或者可能会产生“no such operator” 错误，但可以通过增加圆括号解决。不过在一些极端情况中，查询可能在 没有被报告解析错误的情况下发生行为的改变。如果你发觉这些改变悄悄地 破坏了一些事情，可以打开operator_precedence_warning 配置参数，然后测试你的应用看看有没有一些警告被记录。

4.2. 值表达式

值表达式被用于各种各样的环境中，例如在SELECT命令的目标列表中、作为INSERT或UPDATE中的新列值或者若干命令中的搜索条件。为了区别于一个表表达式（是一个表）的结果，一个值表达式的结果有时候被称为一个标量。值表达式因此也被称为标量表达式（或者甚至简称为表达式）。表达式语法允许使用算数、逻辑、集合和其他操作从原始部分计算值。

一个值表达式是下列之一：

- 一个常量或文字值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用
- 一个下标表达式
- 一个域选择表达式
- 一个操作符调用
- 一个函数调用

- 一个聚集表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 另一个在圆括号（用来分组子表达式以及重载优先级）中的值表达式

在这个列表之外，还有一些结构可以被分类为一个表达式，但是它们不遵循任何一般语法规则。这些通常具有一个函数或操作符的语义并且在第 9 章的合适位置解释。一个例子是IS NULL子句。

我们已经在第 4.1.2 节讨论过常量。下面的小节会讨论剩下的选项。

4.2.1. 列引用

一个列可以以下面的形式被引用：

```
correlation.columnname
```

correlation是一个表（有可能以一个模式名限定）的名字，或者是在FROM子句中为一个表定义的别名。如果列名在当前索引所使用的表中都是唯一的，关联名称和分隔用的句点可以被忽略（另见第 7 章。

4.2.2. 位置参数

一个位置参数引用被用来指示一个由 SQL 语句外部提供的值。参数被用于 SQL 函数定义和预备查询中。某些客户端库还支持独立于 SQL 命令字符串来指定数据值，在这种情况中参数被用来引用那些线外数据值。一个参数引用的形式是：

```
$number
```

例如，考虑一个函数dept的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

这里$1引用函数被调用时第一个函数参数的值。

4.2.3. 下标

如果一个表达式得到了一个数组类型的值，那么可以抽取出该数组值的一个特定元素：

```
expression[subscript]
```

或者抽取出多个相邻元素（一个“数组切片”）：

```
expression[lower_subscript:upper_subscript]
```

（这里，方括号[ ]表示其字面意思）。每一个下标自身是一个表达式，它必须得到一个整数值。

通常，数组表达式必须被加上括号，但是当要被加下标的表达式只是一个列引用或位置参数时，括号可以被忽略。还有，当原始数组是多维时，多个下标可以被连接起来。例如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个例子中的圆括号是必需的。详见第 8.15 节

4.2.4. 域选择

如果一个表达式得到一个组合类型（行类型）的值，那么可以抽取该行的指定域

```
expression.fieldname
```

通常行表达式必须被加上括号，但是当该表达式是仅从一个表引用或位置参数选择时，圆括号可以被忽略。例如：

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

（因此，一个被限定的列引用实际上只是域选择语法的一种特例）。一种重要的特例是从一个组合类型的表列中抽取一个域：

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要圆括号来显示compositecol是一个列名而不是一个表名，在第二种情况中则是显示mytable是一个表名而不是一个模式名。

你可以通过书写.*来请求一个组合值的所有域：

```
(compositecol).*
```

这种记法的行为根据上下文会有不同，详见第 8.16.5 节

4.2.5. 操作符调用

对于一次操作符调用，有三种可能的语法：

```
expression operator expression（二元中缀操作符）
operator expression（一元前缀操作符）
expression operator（一元后缀操作符）
```

其中operator记号遵循第 4.1.3 节的语法规则，或者是关键词AND、OR和NOT之一，或者是一个如下形式的受限定操作符名：

OPERATOR(schema.operatorname)

哪个特定操作符存在以及它们是一元的还是二元的取决于由系统或用户定义的那些操作符。第 9 章描述了内建操作符。

4.2.6. 函数调用

一个函数调用的语法是一个函数的名称（可能受限于一个模式名）后面跟上封闭于圆括号中的参数列表：

```
function_name ([expression [, expression ... ]] )
```

例如，下面会计算 2 的平方根：

```
sqrt(2)
```

当在一个某些用户不信任其他用户的数据库中发出查询时，在编写函数调用时应遵守第 10.3 节的安全防范措施。

内建函数的列表在第 9 章。其他函数可以由用户增加。

参数可以有选择地被附加名称。详见第 4.3 节

> **注意**
>
> 一个采用单一组合类型参数的函数可以被有选择地称为域选择语法，并且反过来域选择可以被写成函数的风格。也就是说，记号col(table)和table.col是可以互换的。这种行为是非 SQL 标准的但是在PostgreSQL中被提供，因为它允许函数的使用来模拟“计算域”。详见第 8.16.5 节

4.2.7. 聚集表达式

一个聚集表达式表示在由一个查询选择的行上应用一个聚集函数。一个聚集函数将多个输入减少到一个单一输出值，例如对输入的求和或平均。一个聚集表达式的语法是下列之一：

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER
 ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER
 ( WHERE filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER
 ( WHERE filter_clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause )
 [ FILTER ( WHERE filter_clause ) ]
```

这里aggregate_name是一个之前定义的聚集（可能带有一个模式名限定），并且expression是任意自身不包含聚集表达式的值表达式或一个窗口函数调用。可选的order_by_clause和filter_clause描述如下。

第一种形式的聚集表达式为每一个输入行调用一次聚集。第二种形式和第一种相同，因为ALL是默认选项。第三种形式为输入行中表达式的每一个可区分值（或者对于多个表达式是值的可区分集合）调用一次聚集。第四种形式为每一个输入行调用一次聚集，因为没有特定的输入值被指定，它通常只对于count(*)聚集函数有用。最后一种形式被用于有序集聚集函数，其描述如下。

大部分聚集函数忽略空输入，这样其中一个或多个表达式得到空值的行将被丢弃。除非另有说明，对于所有内建聚集都是这样。

例如，count(*)得到输入行的总数。count(f1)得到输入行中f1为非空的数量，因为count忽略空值。而count(distinct f1)得到f1的非空可区分值的数量。

一般地，交给聚集函数的输入行是未排序的。在很多情况中这没有关系，例如不管接收到什么样的输入，min总是产生相同的结果。但是，某些聚集函数（例如array_agg 和string_agg）依据输入行的排序产生结果。当使用这类聚集时，可选 的order_by_clause可以被用来指定想要的顺序。order_by_clause与查询级别的ORDER BY子句（如第 7.5 所述）具有相同的语法，除非它的表达式总是仅有表达式并且不能是输出列名称或编号。例如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚集函数时，注意ORDER BY出现在所有聚集参数之后。例如，要这样写：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不能这样写：

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- 不正确
```

后者在语法上是合法的，但是它表示用两个ORDER BY键来调用一个单一参数聚集函数（第二个是无用的，因为它是一个常量）。

如果在order_by_clause之外指定了DISTINCT，那么所有的ORDER BY表达式必须匹配聚集的常规参数。也就是说，你不能在DISTINCT列表没有包括的表达式上排序。

> **注意**
>
> 在一个聚集函数中指定DISTINCT以及ORDER BY的能力是一种PostgreSQL扩展。

按照到目前为止的描述，如果一般目的和统计性聚集中 排序是可选的，在要为它排序输入行时可以在该聚集的常规参数 列表中放置ORDER BY。有一个聚集函数的子集叫 做有序集聚集，它要求一个 order_by_clause，通常是因为 该聚集的计算只对其输入行的特定顺序有意义。有序集聚集的典 型例子包括排名和百分位计算。按照上文的最后一种语法，对于 一个有序集聚集， order_by_clause被写在 WITHIN GROUP (...)中。 order_by_clause中的表达式 会像普通聚集参数一样对每一个输入行计算一次，按照每个 order_by_clause的要求排序并 且交给该聚集函数作为输入参数（这和非 WITHIN GROUP order_by_clause的情况不同，在其中表达 式的结果不会被作为聚集函数的参数）。如果有在 WITHIN GROUP之前的参数表达式，会把它们称 为直接参数以便与列在 order_by_clause中的 聚集参数相区分。与普通聚集参数不同，针对 每次聚集调用只会计算一次直接参数，而不是为每一个输入行 计算一次。这意味着只有那些变量被GROUP BY 分组时，它们才能包含这些变量。这个限制同样适用于根本不在 一个聚集表达式内部的直接参数。直接参数通常被用于百分数 之类的东西，它们只有作为每次聚集计算用一次的单一值才有意 义。直接参数列表可以为空，在这种情况下，写成() 而不是(*)（实际上 PostgreSQL接受两种拼写，但是只有第一 种符合 SQL 标准）。

有序集聚集的调用例子：

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
```

```
 50489
```

这会从表households的 income列得到第 50 个百分位或者中位的值。 这里0.5是一个直接参数，对于百分位部分是一个 在不同行之间变化的值的情况它没有意义。

如果指定了FILTER，那么只有对filter_clause计算为真的输入行会被交给该聚集函数，其他行会被丢弃。例如：

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

预定义的聚集函数在第 9.20 节描述。其他聚集函数可以由用户增加。

一个聚集表达式只能出现在SELECT命令的结果列表或是HAVING子句中。在其他子句（如WHERE）中禁止使用它，因为那些子句的计算在逻辑上是在聚集的结果被形成之前。

当一个聚集表达式出现在一个子查询中（见第 4.2.11 和第 9.22 节，聚集通常在该子查询的行上被计算。但是如果该聚集的参数（以及filter_clause，如果有）只包含外层变量则会产生一个异常：该聚集则属于最近的那个外层，并且会在那个查询的行上被计算。该聚集表达式从整体上则是对其所出现于的子查询的一种外层引用，并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或HAVING子句的限制适用于该聚集所属的查询层次。

4.2.8. 窗口函数调用

一个窗口函数调用表示在一个查询选择的行的某个部分上应用一个聚集类的函数。和非窗口聚集函数调用不同，这不会被约束为将被选择的行分组为一个单一的输出行 — 在查询输出中每一个行仍保持独立。不过，窗口函数能够根据窗口函数调用的分组声明（PARTITION BY列表）访问属于当前行所在分组中的所有行。一个窗口函数调用的语法是下列之一：

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause
 ) ] OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause
 ) ] OVER ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition
 )
```

其中window_definition的语法是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
[ frame_clause ]
```

可选的frame_clause是下列之一

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
```

{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [frame_exclusion]

其中frame_start和frame_end可以是下面形式中的一种

UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING

而frame_exclusion可以是下列之一

EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS

这里，expression表示任何自身不含有窗口函数调用的值表达式。

window_name是对定义在查询的WINDOW子句中的一个命名窗口声明的引用。还可以使用在WINDOW子句中定义命名窗口的相同语法在圆括号内给定一个完整的window_definition，详见SELECT参考页。值得指出的是，OVER wname并不严格地等价于OVER (wname ...)，后者表示复制并修改窗口定义，并且在被引用窗口声明包括一个帧子句时会被拒绝。

PARTITION BY选项将查询的行分组成为分区，窗口函数会独立地处理它们。PARTITION BY工作起来类似于一个查询级别的GROUP BY子句，不过它的表达式总是只是表达式并且不能是输出列的名称或编号。如果没有PARTITION BY，该查询产生的所有行被当作一个单一分区来处理。ORDER BY选项决定被窗口函数处理的一个分区中的行的顺序。它工作起来类似于一个查询级别的ORDER BY子句，但是同样不能使用输出列的名称或编号。如果没有ORDER BY，行将被以未指定的顺序被处理。

frame_clause指定构成窗口帧的行集合，它是当前分区的一个子集，窗口函数将作用在该帧而不是整个分区。帧中的行集合会随着哪一行是当前行而变化。在RANGE、ROWS或者GROUPS模式中可以指定帧，在每一种情况下，帧的范围都是从frame_start到frame_end。如果frame_end被省略，则末尾默认为CURRENT ROW。

UNBOUNDED PRECEDING的一个frame_start表示该帧开始于分区的第一行，类似地UNBOUNDED FOLLOWING的一个frame_end表示该帧结束于分区的最后一行。

在RANGE或GROUPS模式中，CURRENT ROW的一个frame_start表示帧开始于当前行的第一个平级行（被窗口的ORDER BY子句排序为与当前行等效的行），而CURRENT ROW的一个frame_end表示帧结束于当前行的最后一个平级行。在ROWS模式中，CURRENT ROW就表示当前行。

在offset PRECEDING以及offset FOLLOWING帧选项中，offset必须是一个不包含任何变量、聚集函数或者窗口函数的表达式。offset的含义取决于帧模式：

- 在ROWS模式中，offset必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行之前或者之后指定数量的行。
- 在GROUPS模式中，offset也必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行的平级组之前或者之后指定数量的平级组，这里平级组是在ORDER BY顺序中等效的行集合（要使用GROUPS模式，在窗口定义中就必须有一个ORDER BY子句）。
- 在RANGE模式中，这些选项要求ORDER BY子句正好指定一列。offset指定当前行中那一列的值与它在该帧中前面或后面的行中的列值的最大差值。offset表达式的数据类型会随着排序列的数据类型而变化。对于数字的排序列，它通常是与排序列相同的类型，但对于日期时间排序列它是一个interval。例如，如果排序列是类型date或者timestamp，我们可

以写RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING。offset仍然要求是非空且非负，不过“非负”的含义取决于它的数据类型。

在任何一种情况下，到帧末尾的距离都受限于到分区末尾的距离，因此对于离分区末尾比较近的行来说，帧可能会包含比较少的行。

注意在ROWS以及GROUPS模式中，0 PRECEDING和0 FOLLOWING与CURRENT ROW等效。通常在RANGE模式中，这个结论也成立（只要有一种合适的、与数据类型相关的“零”的含义）。

frame_exclusion选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW会把当前行排除在帧之外。EXCLUDE GROUP会把当前行以及它在顺序上的平级行都排除在帧之外。EXCLUDE TIES把当前行的任何平级行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的这种默认行为。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。如果使用ORDER BY，这会把该帧设置为从分区开始一直到当前行的最后一个ORDER BY平级行的所有行。如果不使用ORDER BY，就意味着分区中所有的行都被包括在窗口帧中，因为所有行都成为了当前行的平级行。

限制是frame_start不能是UNBOUNDED FOLLOWING、frame_end不能是UNBOUNDED PRECEDING，并且在上述frame_start和frame_end选项的列表中frame_end选择不能早于frame_start选择出现 — 例如不允许RANGE BETWEEN CURRENT ROW AND offset PRECEDING，但允许ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING，虽然它不会选择任何行。

如果指定了FILTER，那么只有对filter_clause计算为真的输入行会被交给该窗口函数，其他行会被丢弃。只有是聚集的窗口函数才接受FILTER 。

内建的窗口函数在表 9.57中介绍。用户可以加入其他窗口函数。此外，任何内建的或者用户定义的通用聚集或者统计性聚集都可以被用作窗口函数（有序集和假想集聚集当前不能被用作窗口函数）。

使用*的语法被用来把参数较少的聚集函数当作窗口函数调用，例如count(*) OVER (PARTITION BY x ORDER BY y)。星号（*）通常不被用于窗口相关的函数。窗口相关的函数不允许在函数参数列表中使用DISTINCT或ORDER BY。

只有在SELECT列表和查询的ORDER BY子句中才允许窗口函数调用。

更多关于窗口函数的信息可以在第 3.5 节第 9.21 节以及第 7.2.5 节找到。

4.2.9. 类型转换

一个类型造型指定从一种数据类型到另一种数据类型的转换。PostgreSQL接受两种等价的类型造型语法：

```
CAST ( expression AS type )
expression::type
```

CAST语法遵从 SQL，而用::的语法是PostgreSQL的历史用法。

当一个造型被应用到一种未知类型的值表达式上时，它表示一种运行时类型转换。只有已经定义了一种合适的类型转换操作时，该造型才会成功。注意这和常量的造型（如第 4.1.2.7 节所示）使用不同。应用于一个未修饰串文字的造型表示一种类型到一个文字常量值的初始赋值，并且因此它将对任意类型都成功（如果该串文字的内容对于该数据类型的输入语法是可接受的）。

如果一个值表达式必须产生的类型没有歧义（例如当它被指派给一个表列），通常可以省略显式类型造型，在这种情况下系统会自动应用一个类型造型。但是，只有对在系统目录中被

标记为“OK to apply implicitly”的造型才会执行自动造型。其他造型必须使用显式造型语法调用。这种限制是为了防止出人意料的转换被无声无息地应用。

还可以用像函数的语法来指定一次类型造型：

```
typename ( expression )
```

不过，这只对那些名字也作为函数名可用的类型有效。例如，double precision不能以这种方式使用，但是等效的float8可以。还有，如果名称interval、time和timestamp被用双引号引用，那么由于语法冲突的原因，它们只能以这种风格使用。因此，函数风格的造型语法的使用会导致不一致性并且应该尽可能被避免。

> **注意**
>
> 函数风格的语法事实上只是一次函数调用。当两种标准造型语法之一被用来做一次运行时转换时，它将在内部调用一个已注册的函数来执行该转换。简而言之，这些转换函数具有和它们的输出类型相同的名字，并且因此“函数风格的语法”无非是对底层转换函数的一次直接调用。显然，一个可移植的应用不应当依赖于它。详见CREATE CAST。

4.2.10. 排序规则表达式

COLLATE子句会重载一个表达式的排序规则。它被追加到它适用的表达式：

```
expr COLLATE collation
```

这里collation可能是一个受模式限定的标识符。COLLATE子句比操作符绑得更紧，需要时可以使用圆括号。

如果没有显式指定排序规则，数据库系统会从表达式所涉及的列中得到一个排序规则，如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE子句的两种常见使用是重载ORDER BY子句中的排序顺序，例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

以及重载具有区域敏感结果的函数或操作符调用的排序规则，例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

注意在后一种情况中，COLLATE子句被附加到我们希望影响的操作符的一个输入参数上。COLLATE子句被附加到该操作符或函数调用的哪个参数上无关紧要，因为被操作符或函数应用的排序规则是考虑所有参数得来的，并且一个显式的COLLATE子句将重载所有其他参数的排序规则（不过，附加非匹配COLLATE子句到多于一个参数是一种错误。详见第 23.2 节。因此，这会给出和前一个例子相同的结果：

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

但是这是一个错误：

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试把一个排序规则应用到>操作符的结果，而它的数据类型是非可排序数据类型boolean。

4.2.11. 标量子查询

一个标量子查询是一种圆括号内的普通SELECT查询，它刚好返回一行一列（关于书写查询可见第 7 章。SELECT查询被执行并且该单一返回值被使用在周围的值表达式中。将一个返回超过一行或一列的查询作为一个标量子查询使用是一种错误（但是如果在一次特定执行期间该子查询没有返回行则不是错误，该标量结果被当做为空）。该子查询可以从周围的查询中引用变量，这些变量在该子查询的任何一次计算中都将作为常量。对于其他涉及子查询的表达式还可见第 9.22 节

例如，下列语句会寻找每个州中最大的城市人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. 数组构造器

一个数组构造器是一个能构建一个数组值并且将值用于它的成员元素的表达式。一个简单的数组构造器由关键词ARRAY、一个左方括号[、一个用于数组元素值的表达式列表（用逗号分隔）以及最后的一个右方括号]组成。例如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

默认情况下，数组元素类型是成员表达式的公共类型，使用和UNION或CASE结构（见第 10.5 节相同的规则决定。你可以通过显式将数组构造器造型为想要的类型来重载，例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

这和把每一个表达式单独地造型为数组元素类型的效果相同。关于造型的更多信息请见第 4.2.9 节

多维数组值可以通过嵌套数组构造器来构建。在内层的构造器中，关键词ARRAY可以被忽略。例如，这些语句产生相同的结果：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
```

```
(1 row)
```

因为多维数组必须是矩形的，处于同一层次的内层构造器必须产生相同维度的子数组。任何被应用于外层ARRAY构造器的造型会自动传播到所有的内层构造器。

多维数组构造器元素可以是任何得到一个正确种类数组的任何东西，而不仅仅是一个子-ARRAY结构。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

你可以构造一个空数组，但是因为无法得到一个无类型的数组，你必须显式地把你的空数组造型成想要的类型。例如：

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

也可以从一个子查询的结果构建一个数组。在这种形式中，数组构造器被写为关键词ARRAY后跟着一个加了圆括号（不是方括号）的子查询。例如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                                array
----------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

子查询必须返回一个单一列。如果子查询的输出列是非数组类型， 结果的一维数组将为该子查询结果中的每一行有一个元素， 并且有一个与子查询的输出列匹配的元素类型。如果子查询的输出列 是一种数组类型，结果将是同类型的一个数组，但是要高一个维度。 在这种情况下，该子查询的所有行必须产生同样维度的数组，否则结果 就不会是矩形形式。

用ARRAY构建的一个数组值的下标总是从一开始。更多关于数组的信息，请见第 8.15 节

4.2.13. 行构造器

一个行构造器是能够构建一个行值（也称作一个组合类型）并用值作为其成员域的表达式。一个行构造器由关键词ROW、一个左圆括号、用于行的域值的零个或多个表达式（用逗号分隔）以及最后的一个右圆括号组成。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

当在列表中有超过一个表达式时，关键词ROW是可选的。

一个行构造器可以包括语法rowvalue.*，它将被扩展为该行值的元素的一个列表，就像在一个顶层SELECT列表（见第 8.16.5 节中使用.*时发生的事情一样。例如，如果表t有列f1和f2，那么这些是相同的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **注意**
>
> 在PostgreSQL 8.2 以前，.*语法不会在行构造器中被扩展，这样写ROW(t.*, 42)会创建一个有两个域的行，其第一个域是另一个行值。新的行为通常更有用。如果你需要嵌套行值的旧行为，写内层行值时不要用.*，例如ROW(t, 42)。

默认情况下，由一个ROW表达式创建的值是一种匿名记录类型。如果必要，它可以被造型为一种命名的组合类型 — 或者是一个表的行类型，或者是一种用CREATE TYPE AS创建的组合类型。为了避免歧义，可能需要一个显式造型。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 不需要造型因为只有一个 getf1() 存在
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 现在我们需要一个造型来指示要调用哪个函数：
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

行构造器可以被用来构建存储在一个组合类型表列中的组合值，或者被传递给一个接受组合参数的函数。还有，可以比较两个行值，或者用IS NULL或IS NOT NULL测试一个行，例如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

详见第 9.23 节如第 9.22 节所讨论的，行构造器也可以被用来与子查询相连接。

4.2.14. 表达式计算规则

子表达式的计算顺序没有被定义。特别地，一个操作符或函数的输入不必按照从左至右或其他任何固定顺序进行计算。

此外，如果一个表达式的结果可以通过只计算其一部分来决定，那么其他子表达式可能完全不需要被计算。例如，如果我们写：

```
SELECT true OR somefunc();
```

那么somefunc()将（可能）完全不被调用。如果我们写成下面这样也是一样：

```
SELECT somefunc() OR true;
```

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

因此，在复杂表达式中使用带有副作用的函数是不明智的。在WHERE和HAVING子句中依赖副作用或计算顺序尤其危险，因为在建立一个执行计划时这些子句会被广泛地重新处理。这些子句中布尔表达式（AND/OR/NOT的组合）可能会以布尔代数定律所允许的任何方式被重组。

当有必要强制计算顺序时，可以使用一个CASE结构（见第 9.17 节。例如，在一个WHERE子句中使用下面的方法尝试避免除零是不可靠的：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

但是这是安全的：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

一个以这种风格使用的CASE结构将使得优化尝试失败，因此只有必要时才这样做（在这个特别的例子中，最好通过写y > 1.5*x来回避这个问题）。

不过，CASE不是这类问题的万灵药。上述技术的一个限制是， 它无法阻止常量子表达式的提早计算。如第 38.7 节中所述，当查询被规划而不是被执行时，被标记成 IMMUTABLE的函数和操作符可以被计算。因此

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

很可能会导致一次除零失败，因为规划器尝试简化常量子表达式。即便是 表中的每一行都有x > 0（这样运行时永远不会进入到 ELSE分支）也是这样。

虽然这个特别的例子可能看起来愚蠢，没有明显涉及常量的情况可能会发生 在函数内执行的查询中，因为因为函数参数的值和本地变量可以作为常量 被插入到查询中用于规划目的。例如，在PL/pgSQL函数 中，使用一个IF-THEN-ELSE语句来 保护一种有风险的计算比把它嵌在一个CASE表达式中要安全得多。

另一个同类型的限制是，一个CASE无法阻止其所包含的聚集表达式 的计算，因为在考虑SELECT列表或HAVING子句中的 其他表达式之前，会先计算聚集表达式。例如，下面的查询会导致一个除零错误， 虽然看起来好像已经这种情况加以了保护：

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

min()和avg()聚集会在所有输入行上并行地计算，　　因此如果任何行有employees等于零，在有机会测试　　min()的结果之前，就会发生除零错误。取而代之的是，可以使用　　一个WHERE或FILTER子句来首先阻止有问题的输入行到达 一个聚集函数。

4.3.　调用函数

PostgreSQL允许带有命名参数的函数被使用位置或命名记号法调用。命名记号法对于有大量参数的函数特别有用，因为它让参数和实际参数之间的关联更明显和可靠。在位置记号法中，书写一个函数调用时，其参数值要按照它们在函数声明中被定义的顺序书写。在命名记号法中，参数根据名称匹配函数参数，并且可以以任何顺序书写。对于每一种记法，还要考虑函数参数类型的效果，这些在第　10.3　节介绍。

在任意一种记号法中，在函数声明中给出了默认值的参数根本不需要在调用中写出。但是这在命名记号法中特别有用，因为任何参数的组合都可以被忽略。而在位置记号法中参数只能从右往左忽略。

PostgreSQL也支持混合记号法，它组合了位置和命名记号法。在这种情况中，位置参数被首先写出并且命名参数出现在其后。

下列例子将展示所有三种记号法的用法：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT
 false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数concat_lower_or_upper有两个强制参数，a和b。此外，有一个可选的参数uppercase，其默认值为false。a和b输入将被串接，并且根据uppercase参数被强制为大写或小写形式。这个函数的剩余细节对这里并不重要（详见第　38　章。

4.3.1.　使用位置记号

在PostgreSQL中，位置记号法是给函数传递参数的传统机制。一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

所有参数被按照顺序指定。结果是大写形式，因为uppercase被指定为true。另一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World');
```

```
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

这里，uppercase参数被忽略，因此它接收它的默认值false，并导致小写形式的输出。在位置记号法中，参数可以按照从右往左被忽略并且因此而得到默认值。

4.3.2. 使用命名记号

在命名记号法中，每一个参数名都用=> 指定来把它与参数表达式分隔开。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

再次，参数uppercase被忽略，因此它被隐式地设置为false。使用命名记号法的一个优点是参数可以用任何顺序指定，例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

为了向后兼容性，基于 ":=" 的旧语法仍被支持：

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. 使用混合记号

混合记号法组合了位置和命名记号法。不过，正如已经提到过的，命名参数不能超越位置参数。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

在上述查询中，参数a和b被以位置指定，而uppercase通过名字指定。在这个例子中，这只增加了一点文档。在一个具有大量带默认值参数的复杂函数中，命名的或混合的记号法可以节省大量的书写并且减少出错的机会。

注意

命名的和混合的调用记号法当前不能在调用聚集函数时使用（但是当聚集函数被用作窗口函数时它们可以被使用）。

第 5 章 数据定义

本章包含了如何创建用来保存数据的数据库结构。在一个关系型数据库中，原始数据被存储在表中，因此本章的主要工作就是解释如何创建和修改表，以及哪些特性可以控制何种数据会被存储在表中。接着，我们讨论表如何被组织成模式，以及如何将权限分配给表。最后，我们将将简短地介绍其他一些影响数据存储的特性，例如继承、表分区、视图、函数和触发器。

5.1. 表基础

关系型数据库中的一个表非常像纸上的一张表：它由行和列组成。列的数量和顺序是固定的，并且每一列拥有一个名字。行的数目是变化的，它反映了在一个给定时刻表中存储的数据量。SQL并不保证表中行的顺序。当一个表被读取时，表中的行将以非特定顺序出现，除非明确地指定需要排序。这些将在第 7 章介绍。此外，SQL不会为行分配唯一的标识符，因此在一个表中可能会存在一些完全相同的行。这是SQL之下的数学模型导致的结果，但并不是所期望的。稍后在本章中我们将看到如何处理这种问题。

每一列都有一个数据类型。数据类型约束着一组可以分配给列的可能值，并且它为列中存储的数据赋予了语义，这样它可以用于计算。例如，一个被声明为数字类型的列将不会接受任何文本串，而存储在这样一列中的数据可以用来进行数学计算。反过来，一个被声明为字符串类型的列将接受几乎任何一种的数据，它可以进行如字符串连接的操作但不允许进行数学计算。

PostgreSQL包括了相当多的内建数据类型，可以适用于很多应用。用户也可以定义他们自己的数据类型。大部分内建数据类型有着显而易见的名称和语义，所以我们将它们的详细解释放在第 8 章。一些常用的数据类型是：用于整数的integer；可以用于分数的numeric；用于字符串的text，用于日期的date，用于一天内时间的time以及可以同时包含日期和时间的timestamp。

要创建一个表，我们要用到CREATE TABLE命令。在这个命令中 我们需要为新表至少指定一个名字、列的名字及数据类型。例如：

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

这将创建一个名为my_first_table的表，它拥有两个列。第一个列名为first_column且数据类型为text；第二个列名为second_column且数据类型为integer。表和列的名字遵循第 4.1.1 节解释的标识符语法。类型名称通常也是标识符，但是也有些例外。注意列的列表由逗号分隔并被圆括号包围。

当然，前面的例子是非常不自然的。通常，我们为表和列赋予的名称都会表明它们存储着什么类别的数据。因此让我们再看一个更现实的例子：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

（numeric类型能够存储小数部分，典型的例子是金额。）

> **提示**
>
> 当我们创建很多相关的表时，最好为表和列选择一致的命名模式。例如，一种选择是用单数或复数名词作为表名，每一种都受到一些理论家支持。

一个表能够拥有的列的数据是有限的，根据列的类型，这个限制介于250和1600之间。但是，极少会定义一个接近这个限制的表，即便有也是一个值的商榷的设计。

如果我们不再需要一个表，我们可以通过使用DROP TABLE命令来移除它。例如：

```
DROP TABLE my_first_table;
DROP TABLE products;
```

尝试移除一个不存在的表会引起错误。然而，在SQL脚本中在创建每个表之前无条件地尝试移除它的做法是很常见的，即使发生错误也会忽略之，因此这样的脚本可以在表存在和不存在时都工作得很好（如果你喜欢，可以使用DROP TABLE IF EXISTS变体来防止出现错误消息，但这并非标准SQL）。

如果我们需要修改一个已经存在的表，请参考本章稍后的第 5.5 节

利用到目前为止所讨论的工具，我们可以创建一个全功能的表。本章的后续部分将集中于为表定义增加特性来保证数据完整性、安全性或方便。如果你希望现在就去填充你的表，你可以跳过这些直接去第 6 章

5.2. 默认值

一个列可以被分配一个默认值。当一个新行被创建且没有为某些列指定值时，这些列将会被它们相应的默认值填充。一个数据操纵命令也可以显式地要求一个列被置为它的默认值，而不需要知道这个值到底是什么（数据操纵命令详见第 6 章。

如果没有显式指定默认值，则默认值是空值。这是合理的，因为空值表示未知数据。

在一个表定义中，默认值被列在列的数据类型之后。例如：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

默认值可以是一个表达式，它将在任何需要插入默认值的时候被实时计算（不是表创建时）。一个常见的例子是为一个timestamp列指定默认值为CURRENT_TIMESTAMP，这样它将得到行被插入时的时间。另一个常见的例子是为每一行生成一个“序列号” 。这在PostgreSQL可以按照如下方式实现：

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

这里nextval()函数从一个序列对象第 9.16 节。还有一种特别的速写：

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

SERIAL速写将在第 8.1.4 进一步讨论。

5.3. 约束

数据类型是一种限制能够存储在表中数据类别的方法。但是对于很多应用来说，它们提供的约束太粗糙。例如，一个包含产品价格的列应该只接受正值。但是没有任何一种标准数据类型只接受正值。另一个问题是我们可能需要根据其他列或行来约束一个列中的数据。例如，在一个包含产品信息的表中，对于每个产品编号应该只有一行。

到目前为止，SQL允许我们在列和表上定义约束。约束让我们能够根据我们的愿望来控制表中的数据。如果一个用户试图在一个列中保存违反一个约束的数据，一个错误会被抛出。即便是这个值来自于默认值定义，这个规则也同样适用。

5.3.1. 检查约束

一个检查约束是最普通的约束类型。它允许我们指定一个特定列中的值必须要满足一个布尔表达式。例如，为了要求正值的产品价格，我们可以使用：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

如你所见，约束定义就和默认值定义一样跟在数据类型之后。默认值和约束之间的顺序没有影响。一个检查约束有关键字CHECK以及其后的包围在圆括号中的表达式组成。检查约束表达式应该涉及到被约束的列，否则该约束也没什么实际意义。

我们也可以给与约束一个独立的名称。这会使得错误消息更为清晰，同时也允许我们在需要更改约束时能引用它。语法为：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

要指定一个命名的约束，请在约束名称标识符前使用关键词CONSTRAINT，然后把约束定义放在标识符之后（如果没有以这种方式指定一个约束名称，系统将会为我们选择一个）。

一个检查约束也可以引用多个列。例如我们存储一个普通价格和一个打折后的价格，而我们希望保证打折后的价格低于普通价格：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

前两个约束看起来很相似。第三个则使用了一种新语法。它并没有依附在一个特定的列，而是作为一个独立的项出现在逗号分隔的列列表中。列定义和这种约束定义可以以混合的顺序出现在列表中。

我们将前两个约束称为列约束，而第三个约束为表约束，因为它独立于任何一个列定义。列约束也可以写成表约束，但反过来不行，因为一个列约束只能引用它所依附的那一个列（PostgreSQL并不强制要求这个规则，但是如果我们希望表定义能够在其他数据库系统中工作，那就应该遵循它）。上述例子也可以写成：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

甚至是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

这只是口味的问题。

表约束也可以用列约束相同的方法来指定名称：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

需要注意的是，一个检查约束在其检查表达式值为真或空值时被满足。因为当任何操作数为空时大部分表达式将计算为空值，所以它们不会阻止被约束列中的控制。为了保证一个列不包含控制，可以使用下一节中的非空约束。

5.3.2. 非空约束

一个非空约束仅仅指定一个列中不会有空值。语法例子：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
```

```
);
```

一个非空约束总是被写成一个列约束。一个非空约束等价于创建一个检查约束CHECK (column_name IS NOT NULL)，但在PostgreSQL中创建一个显式的非空约束更高效。这种方式创建的非空约束的缺点是我们无法为它给予一个显式的名称。

当然，一个列可以有多于一个的约束，只需要将这些约束一个接一个写出：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

约束的顺序没有关系，因为并不需要决定约束被检查的顺序。

NOT NULL约束有一个相反的情况：NULL约束。这并不意味着该列必须为空，进而肯定是无用的。相反，它仅仅选择了列可能为空的默认行为。SQL标准中并不存在NULL约束，因此它不能被用于可移植的应用中（PostgreSQL中加入它是为了和某些其他数据库系统兼容）。但是某些用户喜欢它，因为它使得在一个脚本文件中可以很容易的进行约束切换。例如，初始时我们可以：

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

然后可以在需要的地方插入NOT关键词。

> **提示**
>
> 在大部分数据库中多数列应该被标记为非空。

5.3.3. 唯一约束

唯一约束保证\在一列中或者一组列中保存的数据在表中所有行间是唯一的。写成一个列约束的语法是：

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

写成一个表约束的语法是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

要为一组列定义一个唯一约束，把它写作一个表级约束，列名用逗号分隔：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

这指定这些列的组合值在整个表的范围内是唯一的，但其中任意一列的值并不需要是（一般也不是）唯一的。

我们可以通常的方式为一个唯一索引命名：

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

增加一个唯一约束会在约束中列出的列或列组上自动创建一个唯一B-tree索引。只覆盖某些行的唯一性限制不能被写为一个唯一约束，但可以通过创建一个唯一的部分索引来强制这种限制。

通常，如果表中有超过一行在约束所包括列上的值相同，将会违反唯一约束。但是在这种比较中，两个空值被认为是不同的。这意味着即便存在一个唯一约束，也可以存储多个在至少一个被约束列中包含空值的行。这种行为符合SQL标准，但我们听说一些其他SQL数据库可能不遵循这个规则。所以在开发需要可移植的应用时应注意这一点。

5.3.4. 主键

一个主键约束表示可以用作表中行的唯一标识符的一个列或者一组列。这要求那些值都是唯一的并且非空。因此，下面的两个表定义接受相同的数据：

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

主键也可以包含多于一个列，其语法和唯一约束相似：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

增加一个主键将自动在主键中列出的列或列组上创建一个唯一B-tree索引。并且会强制这些列被标记为NOT NULL。

一个表最多只能有一个主键（可以有任意数量的唯一和非空约束，它们可以达到和主键几乎一样的功能，但只能有一个被标识为主键）。关系数据库理论要求每一个表都要有一个主键。但PostgreSQL中并未强制要求这一点，但是最好能够遵循它。

主键对于文档和客户端应用都是有用的。例如，一个允许修改行值的 GUI 应用可能需要知道一个表的主键，以便能唯一地标识行。如果定义了主键，数据库系统也有多种方法来利用主键。例如，主键定义了外键要引用的默认目标列。

5.3.5. 外键

一个外键约束指定一列（或一组列）中的值必须匹配出现在另一个表中某些行的值。我们说这维持了两个关联表之间的引用完整性。

例如我们有一个使用过多次的产品表：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

让我们假设我们还有一个存储这些产品订单的表。我们希望保证订单表中只包含真正存在的产品的订单。因此我们在订单表中定义一个引用产品表的外键约束：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

现在就不可能创建包含不存在于产品表中的product_no值（非空）的订单。

我们说在这种情况下，订单表是引用表而产品表是被引用表。相应地，也有引用和被引用列的说法。

我们也可以把上述命令简写为：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

因为如果缺少列的列表，则被引用表的主键将被用作被引用列。

一个外键也可以约束和引用一组列。照例，它需要被写成表约束的形式。下面是一个例子：

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

当然，被约束列的数量和类型应该匹配被引用列的数量和类型。

按照前面的方式，我们可以为一个外键约束命名。

一个表可以有超过一个的外键约束。这被用于实现表之间的多对多关系。例如我们有关于产品和订单的表，但我们现在希望一个订单能包含多种产品（这在上面的结构中是不允许的）。我们可以使用这种表结构：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

注意在最后一个表中主键和外键之间有重叠。

我们知道外键不允许创建与任何产品都不相关的订单。但如果一个产品在一个引用它的订单创建之后被移除会发生什么？SQL允许我们处理这种情况。直观上，我们有几种选项：

- 不允许删除一个被引用的产品
- 同时也删除引用产品的订单
- 其他？

为了说明这些，让我们在上面的多对多关系例子中实现下面的策略：当某人希望移除一个仍然被一个订单引用（通过order_items）的产品时 ，我们组织它。如果某人移除一个订单，订单项也同时被移除：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

限制删除或者级联删除是两种最常见的选项。RESTRICT阻止删除一个被引用的行。NO ACTION表示在约束被检察时如果有任何引用行存在，则会抛出一个错误，这是我们没有指定

任何东西时的默认行为（这两种选择的本质不同在于NO ACTION允许检查被推迟到事务的最后，而RESTRICT则不会）。CASCADE指定当一个被引用行被删除后，引用它的行也应该被自动删除。还有其他两种选项：SET NULL和SET DEFAULT。这些将导致在被引用行被删除后，引用行中的引用列被置为空值或它们的默认值。注意这些并不会是我们免于遵守任何约束。例如，如果一个动作指定了SET DEFAULT，但是默认值不满足外键约束，操作将会失败。

与ON DELETE相似，同样有ON UPDATE可以用在一个被引用列被修改（更新）的情况，可选的动作相同。在这种情况下，CASCADE意味着被引用列的更新值应该被复制到引用行中。

正常情况下，如果一个引用行的任意一个引用列都为空，则它不需要满足外键约束。如果在外键定义中加入了MATCH FULL，一个引用行只有在它的所有引用列为空时才不需要满足外键约束（因此空和非空值的混合肯定会导致MATCH FULL约束失败）。如果不希望引用行能够避开外键约束，将引用行声明为NOT NULL。

一个外键所引用的列必须是一个主键或者被唯一约束所限制。这意味着被引用列总是拥有一个索引（位于主键或唯一约束之下的索引），因此在其上进行的一个引用行是否匹配的检查将会很高效。由于从被引用表中DELETE一行或者UPDATE一个被引用列将要求对引用表进行扫描以得到匹配旧值的行，在引用列上建立合适的索引也会大有益处。由于这种做法并不是必须的，而且创建索引也有很多种选择，所以外键约束的定义并不会自动在引用列上创建索引。

更多关于更新和删除数据的信息请见第 6 章外键约束的语法描述请参考CREATE TABLE。

5.3.6. 排他约束

排他约束保证如果将任何两行的指定列或表达式使用指定操作符进行比较，至少其中一个操作符比较将会返回否或空值。语法是：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

详见CREATE TABLE ... CONSTRAINT ... EXCLUDE。

增加一个排他约束将在约束声明所指定的类型上自动创建索引。

5.4. 系统列

每一个表都拥有一些由系统隐式定义的系统列。因此，这些列的名字不能像用户定义的列一样使用（注意这种限制与名称是否为关键词没有关系，即便用引号限定一个名称也无法绕过这种限制）。 事实上用户不需要关心这些列，只需要知道它们存在即可。

oid

一行的对象标识符（对象ID）。该列只有在表使用WITH OIDS创建时或者default_with_oids配置变量被设置时才存在。该列的类型为oid（与列名一致），该类型详见第 8.19 节

tableoid

包含这一行的表的OID。该列是特别为从继承层次（见第 5.9 节中选择的查询而准备，因为如果没有它将很难知道一行来自于哪个表。tableoid可以与pg_class的oid列进行连接来获得表的名称。

xmin

插入该行版本的事务身份（事务ID）。一个行版本是一个行的一个特别版本，对一个逻辑行的每一次更新都将创建一个新的行版本。

cmin

插入事务中的命令标识符（从0开始）。

xmax

删除事务的身份（事务ID），对于未删除的行版本为0。对于一个可见的行版本，该列值也可能为非零。这通常表示删除事务还没有提交，或者一个删除尝试被回滚。

cmax

删除事务中的命令标识符，或者为0。

ctid

行版本在其表中的物理位置。注意尽管ctid可以被用来非常快速地定位行版本，但是一个行的ctid会在被更新或者被VACUUM FULL移动时改变。因此，ctid不能作为一个长期行标识符。OID或者最好是一个用户定义的序列号才应该被用来标识逻辑行。

OID是32位量，它从一个服务于整个集簇的计数器分配而来。在一个大型的或者历时长久的数据库中，该计数器有可能会出现绕回。因此，不要总是假设OID是唯一的，除非你采取了措施来保证。如果需要在一个表中标识行，推荐使用一个序列生成器。然而，OID也可以被使用，但是是要采取一些额外的预防措施：

- 如果要将OID用来标识行，应该在OID列上创建一个唯一约束。当这样一个唯一约束（或唯一索引）存在时，系统会注意不生成匹配现有行的OID（当然，这只有在表的航数目少于2^{32}（40亿）时才成立。并且在实践中表的尺寸最好远比这个值小，否则将会牺牲性能）。
- 绝不要认为OID在表之间也是唯一的，使用tableoid和行OID的组合来作为数据库范围内的标识符。
- 当然，问题中的表都必须是用WITH OIDS创建。在PostgreSQL 8.1中，WITHOUT OIDS是默认形式。

事务标识符也是32位量。在一个历时长久的数据库中事务ID同样会绕回。但如果采取适当的维护过程，这不会是一个致命的问题，详见第 24 章但是，长期（超过10亿个事务）依赖事务ID的唯一性是不明智的。

命令标识符也是32位量。这对一个事务中包含的SQL命令设置了一个硬极限： 2^{32}（40亿）。在实践中，该限制并不是问题 — 注意该限制只是针对SQL命令的数目而不是被处理的行数。同样，只有真正 修改了数据库内容的命令才会消耗一个命令标识符。

5.5. 修改表

当我们已经创建了一个表并意识到犯了一个错误或者应用需求发生改变时，我们可以移除表并重新创建它。但如果表中已经被填充数据或者被其他数据库对象引用（例如有一个外键约束），这种做法就显得很不方便。因此，PostgreSQL提供了一族命令来对已有的表进行修改。注意这和修改表中所包含的数据是不同的，这里要做的是对表的定义或者说结构进行修改。

利用这些命令，我们可以：

- 增加列
- 移除列
- 增加约束
- 移除约束
- 修改默认值
- 修改列数据类型
- 重命名列
- 重命名表

所有这些动作都由ALTER TABLE命令执行，其参考页面中包含更详细的信息。

5.5.1. 增加列

要增加一个列，可以使用这样的命令：

```
ALTER TABLE products ADD COLUMN description text;
```

新列将被默认值所填充（如果没有指定DEFAULT子句，则会填充空值）。

也可以同时为列定义约束，语法：

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

事实上CREATE TABLE中关于一列的描述都可以应用在这里。记住不管怎样，默认值必须满足给定的约束，否则ADD将会失败。也可以先将新列正确地填充好，然后再增加约束（见后文）。

> **提示**
>
> 增加一个带默认值的列需要更新表中的每一行（来存储新列值）。然而，如果不指定默认值，PostgreSQL可以避免物理更新。因此如果我们准备向列中填充的值大多是非默认值，最好是增加列的时候不指定默认值，增加列后用UPDATE填充正确的数据并且增加所需要的默认值约束。

5.5.2. 移除列

为了移除一个列，使用如下的命令：

```
ALTER TABLE products DROP COLUMN description;
```

列中的数据将会消失。涉及到该列的表约束也会被移除。然而，如果该列被另一个表的外键所引用，PostgreSQL不会安静地移除该约束。我们可以通过增加CASCADE来授权移除任何依赖于被删除列的所有东西：

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

关于这个操作背后的一般性机制请见第 5.13 节

5.5.3. 增加约束

为了增加一个约束，可以使用表约束的语法，例如：

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
 product_groups;
```

要增加一个不能写成表约束的非空约束，可使用语法：

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

该约束会立即被检查，所以表中的数据必须在约束被增加之前就已经符合约束。

5.5.4. 移除约束

为了移除一个约束首先需要知道它的名称。如果在创建时已经给它指定了名称，那么事情就变得很容易。否则约束的名称是由系统生成的，我们必须先找出这个名称。psql的命令\d 表名将会对此有所帮助，其他接口也会提供方法来查看表的细节。因此命令是：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（如果处理的是自动生成的约束名称，如$2，别忘了用双引号使它变成一个合法的标识符。）

和移除一个列相似，如果需要移除一个被某些别的东西依赖的约束，也需要加上CASCADE。一个例子是一个外键约束依赖于被引用列上的一个唯一或者主键约束。

这对除了非空约束之外的所有约束类型都一样有效。为了移除一个非空约束可以用：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（回忆一下，非空约束是没有名称的，所以不能用第一种方式。）

5.5.5. 更改列的默认值

要为一个列设置一个新默认值，使用命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

注意这不会影响任何表中已经存在的行，它只是为未来的INSERT命令改变了默认值。

要移除任何默认值，使用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这等同于将默认值设置为空值。相应的，试图删除一个未被定义的默认值并不会引发错误，因为默认值已经被隐式地设置为空值。

5.5.6. 修改列的数据类型

为了将一个列转换为一种不同的数据类型，使用如下命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有当列中的每一个项都能通过一个隐式造型转换为新的类型时该操作才能成功。如果需要一种更复杂的转换，应该加上一个USING子句来指定应该如何把旧值转换为新值。

PostgreSQL将尝试把列的默认值转换为新类型，其他涉及到该列的任何约束也是一样。但是这些转换可能失败或者产生奇特的结果。因此最好在修改类型之前先删除该列上所有的约束，然后在修改完类型后重新加上相应修改过的约束。

5.5.7. 重命名列

要重命名一个列：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. 重命名表

要重命名一个表：

```
ALTER TABLE products RENAME TO items;
```

5.6. 权限

一旦一个对象被创建，它会被分配一个所有者。所有者通常是执行创建语句的角色。对于大部分类型的对象，初始状态下只有所有者（或者超级用户）能够对该对象做任何事情。为了允许其他角色使用它，必须分配权限。

有多种不同的权限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、及USAGE。可以应用于一个特定对象的权限随着对象的类型（表、函数等）而不同。PostgreSQL所支持的不同类型的完整权限信息请参考GRANT。下面的章节将简单介绍如何使用这些权限。

修改或销毁一个对象的权力通常是只有所有者才有的权限。

一个对象可以通过该对象类型相应的ALTER命令来重新分配所有者，例如ALTER TABLE。超级用户总是可以做到这点，普通角色只有同时是对象的当前所有者（或者是拥有角色的一个成员）以及新拥有角色的一个成员时才能做同样的事。

要分配权限，可以使用GRANT命令。例如，如果joe是一个已有角色，而accounts是一个已有表，更新该表的权限可以按如下方式授权：

```
GRANT UPDATE ON accounts TO joe;
```

用ALL取代特定权限会把与对象类型相关的所有权限全部授权。

一个特殊的名为PUBLIC的“角色”可以用来向系统中的每一个角色授予一个权限。同时，在数据库中有很多用户时可以设置“组”角色来帮助管理权限。详见第 21 章

为了撤销一个权限，使用REVOKE命令：

```
REVOKE ALL ON accounts FROM PUBLIC;
```

对象拥有者的特殊权限（即执行DROP、GRANT、REVOKE等的权力）总是隐式地属于拥有者，并且不能被授予或撤销。但是对象拥有者可以选择撤销他们自己的普通权限，例如把一个表变得对他们自己和其他人只读。

一般情况下，只有对象拥有者（或者超级用户）可以授予或撤销一个对象上的权限。但是可以在授予权限时使用“with grant option”来允许接收人将权限转授给其他人。如果后来授予选项被撤销，则所有从接收人那里获得的权限（直接或者通过授权链获得）都将被撤销。更多详情请见GRANT和REVOKE参考页。

5.7. 行安全性策略

除可以通过GRANT使用 SQL 标准的 特权系统之外，表还可以具有 行安全性策略，它针对每一个用户限制哪些行可以 被普通的查询返回或者可以被数据修改命令插入、更新或删除。这种 特性也被称为行级安全性。默认情况下，表不具有 任何策略，这样用户根据 SQL 特权系统具有对表的访问特权，对于 查询或更新来说其中所有的行都是平等的。

当在一个表上启用行安全性时（使用 ALTER TABLE ... ENABLE ROW LEVEL SECURITY），所有对该表选择行或者修改行的普通访问都必须被一条 行安全性策略所允许（不过，表的拥有者通常不服从行安全性策略）。如果 表上不存在策略，将使用一条默认的否定策略，即所有的行都不可见或者不能 被修改。应用在整个表上的操作不服从行安全性，例如TRUNCATE和 REFERENCES。

行安全性策略可以针对特定的命令、角色或者两者。一条策略可以被指定为 适用于ALL命令，或者SELECT、 INSERT、UPDATE或者DELETE。 可以为一条给定策略分配多个角色，并且通常的角色成员关系和继承规则也 适用。

要指定哪些行根据一条策略是可见的或者是可修改的，需要一个返回布尔结果 的表达式。对于每一行，在计算任何来自用户查询的条件或函数之前，先会计 算这个表达式（这条规则的唯一例外是leakproof函数， 它们被保证不会泄露信息，优化器可能会选择在行安全性检查之前应用这类 函数）。使该表达式不返回true的行将不会被处理。可以指定 独立的表达式来单独控制哪些行可见以及哪些行被允许修改。策略表达式会作 为查询的一部分运行并且带有运行该查询的用户的特权，但是安全性定义者函数 可以被用来访问对调用用户不可用的数据。

具有BYPASSRLS属性的超级用户和角色在访问一个表时总是 可以绕过行安全性系统。表拥有者通常也能绕过行安全性，不过表拥有者 可以选择用ALTER TABLE ... FORCE ROW LEVEL SECURITY来服从行安全性。

启用和禁用行安全性以及向表增加策略是只有表拥有者具有的特权。

策略的创建可以使用CREATE POLICY命令，策略的修改 可以使用ALTER POLICY命令，而策略的删除可以使用 DROP POLICY命令。要为一个给定表启用或者禁用行 安全性，可以使用ALTER TABLE命令。

每一条策略都有名称并且可以为一个表定义多条策略。由于策略是表相 关的，一个表的每一条策略都必须有一个唯一的名称。不同的表可以拥有 相同名称的策略。

当多条策略适用于一个给定的查询时，会把它们用OR（对宽容性策略，默认的策略类型）或者AND（对限制性策略）组合在一起。这和给定角色拥有它作为成员的所有角色的特权的规则类似。宽容性策略和限制性策略在下文将会进一步讨论。

作为一个简单的例子，这里是如何在account关系上 创建一条策略以允许只有managers角色的成员能访问行， 并且只能访问它们账户的行：

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

上面的策略隐含地提供了一个与其该约束适用于被一个命令选择的行（这样一个经理不能SELECT、UPDATE或者DELETE属于其他经理的已有行）以及被一个命令修改的行（这样属于其他经理的行不能通过INSERT或者UPDATE创建）。

如果没有指定角色或者使用了特殊的用户名PUBLIC， 则该策略适用于系统上所有的用户。要允许所有用户访问users 表中属于他们自己的行，可以使用一条简单的策略：

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

这个例子的效果和前一个类似。

为了对增加到表中的行使用与可见行不同的策略，可以组合多条策略。这一对策略将允许所有用户查看users表中的所有行，但只能修改他们自己的行：

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
```

```
    USING (user_name = current_user);
```

在一个SELECT命令中，这两条规则被用OR组合在一起，最终的效应就是所有的行都能被选择。在其他命令类型中，只有第二条策略适用，这样其效果就和以前相同。

也可以用ALTER TABLE命令禁用行安全性。禁用行安全性 不会移除定义在表上的任何策略，它们只是被简单地忽略。然后该表中的所有 行都是可见的并且可修改，服从于标准的 SQL特权系统。

下面是一个较大的例子，它展示了这种特性如何被用于生产环境。表 passwd模拟了一个Unix 口令文件：

```
-- 简单的口令文件例子
CREATE TABLE passwd (
  user_name              text UNIQUE NOT NULL,
  pwhash                 text,
  uid                    int  PRIMARY KEY,
  gid                    int  NOT NULL,
  real_name              text NOT NULL,
  home_phone             text,
  extra_info             text,
  home_dir               text NOT NULL,
  shell                  text NOT NULL
);

CREATE ROLE admin;  -- 管理员
CREATE ROLE bob;    -- 普通用户
CREATE ROLE alice;  -- 普通用户

-- 填充表
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- 确保在表上启用行级安全性
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- 创建策略
-- 管理员能看见所有行并且增加任意行
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- 普通用户可以看见所有行
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- 普通用户可以更新它们自己的记录，但是限制普通用户可用的 shell
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- 允许管理员有所有普通权限
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- 用户只在公共列上得到选择访问
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
```

```
  ON passwd TO public;
-- 允许用户更新特定行
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

对于任意安全性设置来说，重要的是测试并确保系统的行为符合预期。 使用上述的例子，下面展示了权限系统工作正确：

```
-- admin 可以看到所有的行和域
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info |
 home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------+------------
+-------------+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root
      | /bin/dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/
bob   | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/
alice | /bin/zsh
(3 rows)

-- 测试 Alice 能做什么
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for relation passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from
 passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for relation passwd
-- Alice 被允许更改她自己的 real_name，但不能改其他的
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for relation passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for relation passwd
-- Alice 可以更改她自己的口令；行级安全性会悄悄地阻止更新其他行
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

目前为止所有构建的策略都是宽容性策略，也就是当多条策略都适用时会被适用“OR”布尔操作符组合在一起。而宽容性策略可以被用来仅允许在预计情况中对行的访问，这比将宽容

性策略与限制性策略（记录必须通过这类策略并且它们会被“AND”布尔操作符组合起来）组合在一起更简单。在上面的例子之上，我们增加一条限制性策略要求通过一个本地Unix套接字连接过来的管理员访问passwd表的记录：

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

然后，由于这条限制性规则的存在，我们可以看到从网络连接进来的管理员将无法看到任何记录：

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir
 | shell
-----------+--------+-----+-----+-----------+------------+------------
+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

参照完整性检查（例如唯一或逐渐约束和外键引用）总是会绕过行级安全性以　保证数据完整性得到维护。在开发模式和行级安全性时必须小心避免　“隐通道”通过这类参照完整性检查泄露信息。

在某些环境中确保行安全性没有被应用很重要。例如，在做备份时，如果　　行安全性悄悄地导致某些行被从备份中忽略掉，这会是灾难性的。在这类　　情况下，你可以设置row_security配置参数为 off。这本身不会绕过行安全性，它所做的是如果任何结果会 被一条策略过滤掉，就会抛出一个错误。然后错误的原因就可以被找到并且 修复。

在上面的例子中，策略表达式只考虑了要被访问的行中的当前值。这是最简　单并且表现最好的情况。如果可能，最好设计行安全性应用以这种方式工作。　如果需要参考其他行或者其他表来做出策略的决定，可以在策略表达式中通过　使用子-SELECT或者包含SELECT的函数来实现。不过要注意这类访问可能会导致竞争条件，在不小心的情况下这可能　会导致信息泄露。作为一个例子，考虑下面的表设计：

```
-- 特权组的定义
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);
```

```
INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice 是管理员
GRANT SELECT ON groups TO public;

-- 用户的特权级别的定义
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- 保存要被保护的信息的表
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- 对于安全性 group_id 大于等于一行的 group_id 的用户，
-- 这一行应该是可见的/可更新的
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name =
 current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name =
 current_user));

-- 我们只依赖于行级安全性来保护信息表
GRANT ALL ON information TO public;
```

现在假设alice希望更改“有一点点秘密”　的信息，但是觉得mallory不应该看到该行中的新 内容，因此她这样做：

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

这看起来是安全的，没有窗口可供mallory看到 “对 mallory 保密”的字符串。不过，这里有一种 竞争条件。如果mallory正在并行地做：

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

并且她的事务处于READ COMMITTED模式，她就可能看到 “s对 mallory 保密”的东西。如果她的事务在alice 做完之后就到达信息行，这就会发生。它会阻塞等待 alice的事务提交，然后拜FOR UPDATE子句所赐 取得更新后的行内容。不过，对于来自users的隐式 SELECT，它不会取得一个已更新的行， 因为子-SELECT没有FOR UPDATE，相反 会使用查询开始时取得的快照读取users行。因此， 策略表达式会测试mallory的特权级别的旧值并且允许她看到被更新的行。

有多种方法能解决这个问题。一种简单的答案是在行安全性策略中的 子-SELECT里使用SELECT ... FOR SHARE。 不过，这要求在被引用表（这里是users）上授予 UPDATE特权给受影响的用户，这可能不是我们想要的（ 但是另一条行安全性策略可能被应用来阻止它们实际使用这个特权，或者 子-SELECT可能被嵌入到一个安全性定义者函数中）。 还有，在被引用的表上过多并发地使用行共享锁可能会导致性能问题， 特别是表更新比较频繁时。另一种解决方案（如果被引用表上的更新 不频繁就可行）是在更新被引用表时对它取一个排他锁，这样就没有 并发事务能够检查旧的行值了。或者我们可以在提交对被引用表的更新之后、在做依赖于新安全性情况的更改之前等待所有并发事务结束。

更多细节请见CREATE POLICY 和ALTER TABLE。

5.8. 模式

一个PostgreSQL数据库集簇中包含一个或更多命名的数据库。用户和用户组被整个集簇共享，但没有其他数据在数据库之间共享。任何给定客户端连接只能访问在连接中指定的数据库中的数据。

> **注意**
>
> 一个集簇的用户并不必拥有访问集簇中每一个数据库的权限。用户名的共享意味着不可能在同一个集簇中出现重名的不同用户，例如两个数据库中都有叫joe的用户。但系统可以被配置为只允许joe访问某些数据库。

一个数据库包含一个或多个命名模式，模式中包含着表。模式还包含其他类型的命名对象，包括数据类型、函数和操作符。相同的对象名称可以被用于不同的模式中二不会出现冲突，例如schema1和myschema都可以包含名为mytable的表。和数据库不同，模式并不是被严格地隔离：一个用户可以访问他们所连接的数据库中的所有模式内的对象，只要他们有足够的权限。

下面是一些使用方案的原因：

- 允许多个用户使用一个数据库并且不会互相干扰。
- 将数据库对象组织成逻辑组以便更容易管理。
- 第三方应用的对象可以放在独立的模式中，这样它们就不会与其他对象的名称发生冲突。

模式类似于操作系统层的目录，但是模式不能嵌套。

5.8.1. 创建模式

要创建一个模式，可使用CREATE SCHEMA命令，并且给出选择的模式名称。例如：

```
CREATE SCHEMA myschema;
```

在一个模式中创建或访问对象，需要使用由模式名和表名构成的限定名，模式名和表名之间以点号分隔：

schema.table

在任何需要一个表名的地方都可以这样用，包括表修改命令和后续章节要讨论的数据访问命令（为了简洁我们在这里只谈到表，但是这种方式对其他类型的命名对象同样有效，例如类型和函数）。

事实上，还有更加通用的语法：

database.schema.table

也可以使用，但是目前它只是在形式上与SQL标准兼容。如果我们写一个数据库名称，它必须是我们正在连接的数据库。

因此，如果要在一个新模式中创建一个表，可用：

```
CREATE TABLE myschema.mytable (
 ...
);
```

要删除一个为空的模式（其中的所有对象已经被删除），可用：

```
DROP SCHEMA myschema;
```

要删除一个模式以及其中包含的所有对象，可用：

```
DROP SCHEMA myschema CASCADE;
```

有关于此的更一般的机制请参见第 5.13 节

我们常常希望创建一个由其他人所拥有的模式（因为这是将用户动作限制在良定义的名字空间中的方法之一）。其语法是：

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

我们甚至可以省略模式名称，在此种情况下模式名称将会使用用户名，参见第 5.8.6 节

以pg_开头的模式名被保留用于系统目的，所以不能被用户所创建。

5.8.2. 公共模式

在前面的小节中，我们创建的表都没有指定任何模式名称。默认情况下这些表（以及其他对象）会自动的被放入一个名为“public”的模式中。任何新数据库都包含这样一个模式。因此，下面的命令是等效的：

```
CREATE TABLE products ( ... );
```

以及：

```
CREATE TABLE public.products ( ... );
```

5.8.3. 模式搜索路径

限定名写起来很冗长，通常最好不要把一个特定模式名拉到应用中。因此，表名通常被使用非限定名来引用，它只由表名构成。系统将沿着一条搜索路径来决定该名称指的是哪个表，搜索路径是一个进行查看的模式列表。　搜索路径中第一个匹配的表将被认为是所需要的。如果在搜索路径中没有任何匹配，即使在数据库的其他模式中存在匹配的表名也将会报告一个错误。

在不同方案中创建命名相同的对象的能力使得编写每次都准确引用相同对象的查询变得复杂。这也使得用户有可能更改其他用户查询的行为，不管是出于恶意还是无意。由于未经限定的名称在查询中以及在PostgreSQL内部的广泛使用，在search_path中增加一个方案实际上是信任所有在该方案中具有CREATE特权的用户。在你运行一个普通查询时，恶意用户可以在你的搜索路径中的以方案中创建能够夺取控制权并且执行任意SQL函数的对象，而这些事情就像是你在执行一样。

搜索路径中的第一个模式被称为当前模式。除了是第一个被搜索的模式外，如果CREATE TABLE命令没有指定模式名，它将是新创建表所在的模式。

要显示当前搜索路径，使用下面的命令：

```
SHOW search_path;
```

在默认设置下这将返回：

```
 search_path
--------------
 "$user",public
```

第一个元素说明一个和当前用户同名的模式会被搜索。如果不存在这个模式，该项将被忽略。第二个元素指向我们已经见过的公共模式。

搜索路径中的第一个模式是创建新对象的默认存储位置。这就是默认情况下对象会被创建在公共模式中的原因。当对象在任何其他没有模式限定的环境中被引用（表修改、数据修改或查询命令）时，搜索路径将被遍历直到一个匹配对象被找到。因此，在默认配置中，任何非限定访问将只能指向公共模式。

要把新模式放在搜索路径中，我们可以使用：

```
SET search_path TO myschema,public;
```

（我们在这里省略了$user，因为我们并不立即需要它）。然后我们可以删除该表而无需使用方案进行限定：

```
DROP TABLE mytable;
```

同样，由于myschema是路径中的第一个元素，新对象会被默认创建在其中。

我们也可以这样写：

```
SET search_path TO myschema;
```

这样我们在没有显式限定时再也不必去访问公共模式了。公共模式没有什么特别之处，它只是默认存在而已，它也可以被删除。

其他操作模式搜索路径的方法请见第　9.25　节

搜索路径对于数据类型名称、函数名称和操作符名称的作用与表名一样。数据类型和函数名称可以使用和表名完全相同的限定方式。如果我们需要在一个表达式中写一个限定的操作符名称，我们必须写成一种特殊的形式：

```
OPERATOR(schema.operator)
```

这是为了避免句法歧义。例如：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

实际上我们通常都会依赖于搜索路径来查找操作符，因此没有必要去写如此“丑陋”的东西。

5.8.4. 模式和权限

默认情况下，用户不能访问不属于他们的方案中的任何对象。要允许这种行为，模式的拥有者必须在该模式上授予USAGE权限。为了允许用户使用方案中的对象，可能还需要根据对象授予额外的权限。

一个用户也可以被允许在其他某人的模式中创建对象。要允许这种行为，模式上的CREATE权限必须被授予。注意在默认情况下，所有人都拥有在public模式上的CREATE和USAGE权限。这使得用户能够连接到一个给定数据库并在它的public模式中创建对象。回收这一特权的使用模式调用：

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

（第一个“public”是方案，第二个“public”指的是“每一个用户”。第一种是一个标识符，第二种是一个关键词，所以两者的大小写不同。请回想第 4.1.1 节的指导方针。）

5.8.5. 系统目录模式

除public和用户创建的模式之外，每一个数据库还包括一个pg_catalog模式，它包含了系统表和所有内建的数据类型、函数以及操作符。pg_catalog总是搜索路径的一个有效部分。如果没有在路径中显式地包括该模式，它将在路径中的模式之前被搜索。这保证了内建的名称总是能被找到。然而，如果我们希望用用户定义的名称重载内建的名称，可以显式的将pg_catalog放在搜索路径的末尾。

由于系统表名称以pg_开头，最好还是避免使用这样的名称，以避免和未来新版本中 可能出现的系统表名发生冲突。系统表将继续采用以pg_开头的方式，这样它们不会 与非限制的用户表名称冲突。

5.8.6. 使用模式

有一些默认配置可以轻易支持的使用模式，当数据库用户不信任其他数据库用户时，使用其中之一就足够了：

- 将普通用户约束在其私有的方案中。要实现这一点，发出REVOKE CREATE ON SCHEMA public FROM PUBLIC，并且为每一个用户创建一个用其用户名命名的方案。如果受影响的用户在做这些之前就已经登入，应该对与方案pg_catalog中对象命名相似的对象审计public方案。回忆一下，默认的搜索路径开始于$user，它会被解析为用户名。因此，如果每个用户都有一个单独的方案，默认他们访问他们自己的方案。

- 使用ALTER ROLE user SET search_path = "$user"从每个用户的默认搜索路径中去掉public方案。每个人都保留着在public方案中创建对象的能力，但是只有限定的名称才能选择那些对象。虽然限定的表引用是好的，但对public方案中函数的调用将是不安全或不

可靠的。此外，持有CREATEROLE特权的用户可以撤销这种设置并且以依赖于这种设置的用户的身份发出任意查询。如果你在public方案中创建方案或扩展或者把CREATEROLE授予给不能正当使用这种几乎是超级用户能力的用户，应该使用第一种模式。

- 从postgresql.conf中的search_path去掉public方案。接下来的用户体验符合前一种模式。除了上一中模式对函数和CREATEROLE的暗示之外，这种模式像CREATEROLE那样信任数据库拥有者。如果你在public方案中创建函数或扩展，或者向不能正当使用几乎是超级用户访问的用户授予CREATEROLE特权、CREATEDB特权或数据库的拥有关系，请使用第一种模式。

- 保持默认。所有用户都隐式地访问public模式。这模拟了方案根本不可用的情况，可以用于从没有方案的世界平滑过渡。不过，任何用户都能以任何无法自我保护的用户的身份发出任意查询。只有当数据库仅有单个用户或者少数相互信任的用户时，这种模式才可接受。

对于任何一种模式，为了安装共享的应用（所有人都要用其中的表，第三方提供的额外函数，等等），可把它们放在单独的方案中。记住授予适当的特权以允许其他用户访问它们。然后用户可以通过以方案名限定名称的方式来引用这些额外的对象，或者他们可以把额外的方案放在自己的搜索路径中。

5.8.7. 可移植性

在SQL标准中，在由不同用户拥有的同一个模式中的对象是不存在的。此外，某些实现不允许创建与拥有者名称不同名的模式。事实上，在那些仅实现了标准中基本模式支持的数据库中，模式和用户的概念是等同的。因此，很多用户认为限定名称实际上是由user_name.table_name组成的。如果我们为每一个用户都创建了一个模式，PostgreSQL实际也是这样认为的。

同样，在SQL标准中也没有public模式的概念。为了最大限度的与标准一致，我们不应使用（甚至是删除）public模式。

当然，某些SQL数据库系统可能根本没有实现方案，或者提供允许跨数据库访问的名字空间。如果需要使用这样一些系统，最好不要使用方案。

5.9. 继承

PostgreSQL实现了表继承，这对数据库设计者来说是一种有用的工具（SQL:1999及其后的版本定义了一种类型继承特性，但和这里介绍的继承有很大的不同）。

让我们从一个例子开始：假设我们要为城市建立一个数据模型。每一个州有很多城市，但是只有一个首府。我们希望能够快速地检索任何特定州的首府城市。这可以通过创建两个表来实现：一个用于州首府，另一个用于不是首府的城市。然而，当我们想要查看一个城市的数据（不管它是不是一个首府）时会发生什么？继承特性将有助于解决这个问题。我们可以将capitals表定义为继承自cities表：

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- in feet
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

在这种情况下，capitals表继承了它的父表cities的所有列。州首府还有一个额外的列state用来表示它所属的州。

在PostgreSQL中，一个表可以从0个或者多个其他表继承，而对一个表的查询则可以引用一个表的所有行或者该表的所有行加上它所有的后代表。默认情况是后一种行为。例如，下面的查询将查找所有海拔高于500尺的城市的名称，包括州首府：

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

对于来自PostgreSQL教程（见第 2.1 节的例子数据，它将返回：

```
   name     | altitude
------------+----------
 Las Vegas  |     2174
 Mariposa   |     1953
 Madison    |      845
```

在另一方面，下面的查询将找到海拔超过500尺且不是州首府的所有城市：

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name     | altitude
------------+----------
 Las Vegas  |     2174
 Mariposa   |     1953
```

这里的ONLY关键词指示查询只被应用于cities上，而其他在继承层次中位于cities之下的其他表都不会被该查询涉及。很多我们已经讨论过的命令（如SELECT、UPDATE和DELETE）都支持ONLY关键词。

我们也可以在表名后写上一个*来显式地将后代表包括在查询范围内：

```
SELECT name, altitude
    FROM cities*
    WHERE altitude > 500;
```

写*不是必需的，因为这种行为总是默认的。不过，为了兼容可以修改默认值的较老版本，现在仍然支持这种语法。

在某些情况下，我们可能希望知道一个特定行来自于哪个表。每个表中的系统列tableoid可以告诉我们行来自于哪个表：

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

将会返回：

```
 tableoid |   name     | altitude
----------+------------+----------
   139793 | Las Vegas  |     2174
   139793 | Mariposa   |     1953
   139798 | Madison    |      845
```

（如果重新生成这个结果，可能会得到不同的OID数字。）通过与pg_class进行连接可以看到实际的表名：

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

将会返回：

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

另一种得到同样效果的方法是使用regclass别名类型， 它将象征性地打印出表的 OID：

```
SELECT c.tableoid::regclass, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

继承不会自动地将来自INSERT或COPY命令的数据传播到继承层次中的其他表中。在我们的例子中，下面的INSERT语句将会失败：

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

我们也许希望数据能被以某种方式被引入到capitals表中，但是这不会发生：INSERT总是向指定的表中插入。在某些情况下，可以通过使用一个规则（见第 41 章来将插入动作重定向。但是这对上面的情况并没有帮助，因为cities表根本就不包含state列，因而这个命令将在触发规则之前就被拒绝。

父表上的所有检查约束和非空约束都将自动被它的后代所继承，除非显式地指定了NO INHERIT子句。其他类型的约束（唯一、主键和外键约束）则不会被继承。

一个表可以从超过一个的父表继承，在这种情况下它拥有父表们所定义的列的并集。任何定义在子表上的列也会被加入到其中。如果在这个集合中出现重名列，那么这些列将被“合并”，这样在子表中只会有一个这样的列。重名列能被合并的前提是这些列必须具有相同的数据类型，否则会导致错误。可继承的检查约束和非空约束会以类似的方式被合并。例如，如果合并成一个合并列的任一列定义被标记为非空，则该合并列会被标记为非空。如果检查约束的名称相同，则他们会被合并，但如果它们的条件不同则合并会失败。

表继承通常是在子表被创建时建立，使用CREATE TABLE语句的INHERITS子句。一个已经被创建的表也可以另外一种方式增加一个新的父亲关系，使用ALTER TABLE的INHERIT变体。要这样做，新的子表必须已经包括和父表相同名称和数据类型的列。子表还必须包括和父表相同的检查约束和检查表达式。相似地，一个继承链接也可以使用ALTER TABLE的 NO INHERIT变体从一个子表中移除。动态增加和移除继承链接可以用于实现表划分（见第 5.10 节。

一种创建一个未来将被用做子女的新表的方法是在CREATE TABLE中使用LIKE子句。这将创建一个和源表具有相同列的新表。如果源表上定义有任何CHECK约束，LIKE的INCLUDING CONSTRAINTS选项可以用来让新的子表也包含和父表相同的约束。

当有任何一个子表存在时，父表不能被删除。当子表的列或者检查约束继承于父表时，它们也不能被删除或修改。如果希望移除一个表和它的所有后代，一种简单的方法是使用CASCADE选项删除父表（见第 5.13 节。

ALTER TABLE将会把列的数据定义或检查约束上的任何变化沿着继承层次向下传播。同样，删除被其他表依赖的列只能使用CASCADE选项。ALTER TABLE对于重名列的合并和拒绝遵循与CREATE TABLE同样的规则。

继承的查询仅在附表上执行访问权限检查。例如，在cities表上授予UPDATE权限也隐含着通过cities访问时在capitals表中更新行的权限。这保留了数据（也）在父表中的样子。但是如果没有额外的授权，则不能直接更新capitals表。以类似的方式，父表的行安全性策略（见第 5.7 节适用于继承查询期间来自于子表的行。只有当子表在查询中被明确提到时，其策略（如果有）才会被应用，在那种情况下，附着在其父表上的任何策略都会被忽略。

外部表（见第 5.11 节也可以是继承层次 中的一部分，即可以作为父表也可以作为子表，就像常规表一样。如果 一个外部表是继承层次的一部分，那么任何不被该外部表支持的操作也 不被整个层次所支持。

5.9.1. 警告

注意并非所有的SQL命令都能工作在继承层次上。用于数据查询、数据修改或模式修改（例如SELECT、UPDATE、DELETE、大部分ALTER TABLE的变体，但INSERT或ALTER TABLE ... RENAME不在此列）的命令会默认将子表包含在内并且支持ONLY记号来排除子表。负责数据库维护和调整的命令（如REINDEX、VACUUM）只工作在独立的、物理的表上并且不支持在继承层次上的递归。每个命令相应的行为请参见它们的参考页（SQL 命令）。

继承特性的一个严肃的限制是索引（包括唯一约束）和外键约束值应用在单个表上而非它们的继承子女。在外键约束的引用端和被引用端都是这样。因此，按照上面的例子：

- 如果我们声明cities.name为UNIQUE或者PRIMARY KEY，这将不会阻止capitals表中拥有和cities中城市同名的行。而且这些重复的行将会默认显示在cities的查询中。事实上，capitals在默认情况下是根本不能拥有唯一约束的，并且因此能够包含多个同名的行。我们可以为capitals增加一个唯一约束，但这无法阻止相对于cities的重复。

- 相似地，如果我们指定cities.name REFERENCES某个其他表，该约束不会自动地传播到capitals。在此种情况下，我们可以变通地在capitals上手工创建一个相同的REFERENCES约束。

- 指定另一个表的列REFERENCES cities(name)将允许其他表包含城市名称，但不会包含首府名称。这对于这个例子不是一个好的变通方案。

这些不足可能还将存在于某些未来的发布中，但是同时在决定继承是否对我们的应用有用时需要相当小心。

5.10. 表分区

PostgreSQL支持基本的表划分。本小节介绍为何以及怎样把划分实现为数据库设计的一部分。

5.10.1. 概述

划分指的是将逻辑上的一个大表分成一些小的物理上的片。划分有很多益处：

- 在某些情况下查询性能能够显著提升，特别是当那些访问压力大的行在一个分区或者少数几个分区时。划分可以取代索引的主导列、减小索引尺寸以及使索引中访问压力大的部分更有可能被放在内存中。

- 当查询或更新访问一个分区的大部分行时，可以通过该分区上的一个顺序扫描来取代分散到整个表上的索引和随机访问，这样可以改善性能。

- 如果批量操作的需求是在分区设计时就规划好的，则批量装载和删除可以通过增加或者去除分区来完成。执行ALTER TABLE DETACH PARTITION或者使用DROP TABLE删除一个分区远快于批量操作。这些命令也完全避免了批量DELETE导致的VACUUM开销。

- 很少使用的数据可以被迁移到便宜且较慢的存储介质上。

当一个表非常大时，划分所带来的好处是非常值得的。一个表何种情况下会从划分获益取决于应用，一个经验法则是当表的尺寸超过了数据库服务器物理内存时，划分会为表带来好处。

PostgreSQL对下列分区形式提供了内建支持：

范围划分

表被根据一个关键列或一组列划分为“范围”，不同的分区的范围之间没有重叠。例如，我们可以根据日期范围划分，或者根据特定业务对象的标识符划分。

列表划分

通过显式地列出每一个分区中出现的键值来划分表。

哈希分区

通过为每个分区指定模数和余数来对表进行分区。每个分区所持有的行都满足：分区键的值除以为其指定的模数将产生为其指定的余数。

如果你的应用需要使用上面所列之外的分区形式，可以使用诸如继承和UNION ALL视图之类的替代方法。这些方法很灵活，但是却缺少内建声明式分区的一些性能优势。

5.10.2. 声明式划分

PostgreSQL提供了一种方法指定如何把一个表划分成称为分区的片段。被划分的表被称作分区表。这种说明由分区方法以及要被用作分区键的列或者表达式列表组成。

所有被插入到分区表的行将被基于分区键的值路由到分区中。每个分区都有一个由其分区边界定义的数据子集。当前支持的分区方法是范围、列表以及哈希。

分区本身也可能被定义为分区表，这种用法被称为子分区。分区可以有自己的与其他分区不同的索引、约束以及默认值。创建分区表及分区的更多细节请见CREATE TABLE。

无法把一个常规表转换成分区表，反之亦然。不过，可以把一个包含数据的常规表或者分区表作为分区加入到另一个分区表，或者从分区表中移走一个分区并且把它变成一个独立的表。有关ATTACH PARTITION和DETACH PARTITION子命令的内容请见ALTER TABLE。

个体分区在内部以继承的方式链接到分区表，不过无法对声明式分区表或其分区使用继承的某些一般特性（下文讨论）。例如，分区不能有除其所属分区表之外的父表，一个常规表也不能从分区表继承使得后者成为其父表。这意味着分区表及其分区不会参与到与常规表的继承关系中。由于分区表及其分区组成的分区层次仍然是一种继承层次，所有第 5.9 节所述的继承的普通规则也适用，不过有一些例外，尤其是：

- 分区表的CHECK约束和NOT NULL约束总是会被其所有的分区所继承。不允许在分区表上创建标记为NO INHERIT的CHECK约束。

- 只要分区表中不存在分区，则支持使用ONLY仅在分区表上增加或者删除约束。一旦分区存在，那样做就会导致错误，因为当分区存在时是不支持仅在分区表上增加或删除约束的。不过，分区表本身上的约束可以被增加（如果它们不出现在父表中）和删除。

- 由于分区表并不直接拥有任何数据，尝试在分区表上使用TRUNCATE ONLY将总是返回错误。

- 分区不能有在父表中不存在的列。在使用CREATE TABLE创建分区时不能指定列，在事后使用ALTER TABLE时也不能为分区增加列。只有当表的列正好匹配父表时（包括任何oid列），才能使用ALTER TABLE ... ATTACH PARTITION将它作为分区加入。

- 如果NOT NULL约束在父表中存在，那么就不能删除分区的列上的对应的NOT NULL约束。

分区也可以是外部表，不过它们有一些普通表没有的限制，详情请见CREATE FOREIGN TABLE。

更新行的分区键可能导致它满足另一个不同的分区的分区边界，进而被移动到那个分区中。

5.10.2.1. 例子

假定我们正在为一个大型的冰激凌公司构建数据库。该公司每天测量最高温度以及每个区域的冰激凌销售情况。概念上，我们想要一个这样的表：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

我们知道大部分查询只会访问上周的、上月的或者上季度的数据，因为这个表的主要用途是为管理层准备在线报告。为了减少需要被存放的旧数据量，我们决定只保留最近3年的数据。在每个月的开始我们将去除掉最早的那个月的数据。在这种情况下我们可以使用分区技术来帮助我们满足对measurement表的所有不同需求。

要在这种情况下使用声明式分区，可采用下面的步骤：

1. 通过指定PARTITION BY子句把measurement表创建为分区表，该子句包括分区方法（这个例子中是RANGE）以及用作分区键的列列表。

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```

   你可能需要决定在分区键中使用多列进行范围分区。当然，这通常会导致较大数量的分区，其中每一个个体都比较小。另一方面，使用较少的列可能会导致粗粒度的分区策略得到较少数量的分区。如果条件涉及这些列中的一部分或者全部，访问分区表的查询将不得不扫描较少的分区。例如，考虑一个使用列lastname和firstname（按照这样的顺序）作为分区键进行范围分区的表。
2. 创建分区。每个分区的定义必须指定对应于父表的分区方法和分区键的边界。注意，如果指定的边界使得新分区的值会与已有分区中的值重叠，则会导致错误。向父表中插入无法映射到任何现有分区的数据将会导致错误，这种情况下应该手工增加一个合适的分区。

   分区以普通PostgreSQL表（或者可能是外部表）的方式创建。可以为每个分区单独指定表空间和存储参数。

   没有必要创建表约束来描述分区的分区边界条件。相反，只要需要引用分区约束时，分区约束会自动地隐式地从分区边界说明中生成。

   ```
   CREATE TABLE measurement_y2006m02 PARTITION OF measurement
       FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

   CREATE TABLE measurement_y2006m03 PARTITION OF measurement
       FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');
   ```

```
...
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

为了实现子分区，在创建分区的命令中指定PARTITION BY子句，例如：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

在创建了measurement_y2006m02的分区之后，任何被插入到measurement中且被映射到measurement_y2006m02的数据（或者直接被插入到measurement_y2006m02的数据，假定它满足这个分区的分区约束）将被基于peaktemp列进一步重定向到measurement_y2006m02的一个分区。指定的分区键可以与父亲的分区键重叠，不过在指定子分区的边界时要注意它接受的数据集合是分区自身边界允许的数据集合的一个子集，系统不会尝试检查事情情况是否如此。

3. 在分区表的键列上创建一个索引，还有其他需要的索引（键索引并不是必需的，但是大部分场景中它都能很有帮助）。这会自动在每个分区上创建一个索引，并且后来创建或者附着的任何分区也将会包含索引。

```
CREATE INDEX ON measurement (logdate);
```

4. 确保enable_partition_pruning配置参数在postgresql.conf中没有被禁用。如果被禁用，查询将不会按照想要的方式被优化。

在上面的例子中，我们会每个月创建一个新分区，因此写一个脚本来自动生成所需的DDL会更好。

5.10.2.2. 分区维护

通常在初始定义分区表时建立的分区并非保持静态不变。移除旧分区的数据并且为新数据周期性地增加新分区的需求比比皆是。分区的最大好处之一就是可以通过操纵分区结构来近乎瞬时地执行这类让人头痛的任务，而不是物理地去除大量数据。

移除旧数据最简单的选择是删除掉不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

这可以非常快地删除数百万行记录，因为它不需要逐个删除每个记录。不过要注意上面的命令需要在父表上拿到ACCESS EXCLUSIVE锁。

另一种通常更好的选项是把分区从分区表中移除，但是保留它作为一个独立的表：

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

这允许在它被删除之前在其数据上执行进一步的操作。例如，这通常是一种使用COPY、pg_dump或类似工具备份数据的好时候。这也是把数据聚集成较小的格式、执行其他数据操作或者运行报表的好时机。

类似地，我们可以增加一个新分区来处理新数据。我们可以在分区表中创建一个空分区，就像上面创建的初始分区那样：

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

另外一种选择是，有时候在分区结构之外创建新表更加方便，然后将它作为一个合适的分区。这允许先对数据进行装载、检查和转换，然后再让它们出现在分区表中：

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

在运行ATTACH PARTITION命令之前，推荐在要被挂接的表上创建一个CHECK约束来描述想要的分区约束。采用这种方法，系统将能够跳过验证隐式分区约束的扫描。如果没有这样一个约束，则将会扫描表来验证分区约束，其间会持有一个父表上的ACCESS EXCLUSIVE锁。人们可能在ATTACH PARTITION完成之后删除该约束，因为之后就不再需要它了。

5.10.2.3. 限制

分区表有下列限制：

- 没有办法创建跨越所有分区的排除约束，只可能单个约束每个叶子分区。
- 虽然在分区表上支持主键，但引用分区表的外键不受支持（但支持从分区表到某个其他表的外键引用）。
- 当一个UPDATE导致一行从一个分区移动到另一个分区时，另一个并发的UPDATE或DELETE可能会产生一个串行化错误。假设会话1正在执行一个分区键上的UPDATE，同时一个并发的能看见这个行的会话2执行了对该行的UPDATE或者DELETE操作。在这种情况下，会话2的UPDATE或者DELETE会检测到行的移动，并抛出一个串行化的错误(将总是会返回一个SQLSTATE '40001')。 如果发生这种情况，应用程序可能希望重试该事务。 在没有分区表或没有行移动的通常情况下， 会话2将识别新更新的行并在新行上执行UPDATE/DELETE。
- 如果必要，必须在个体分区上定义BEFORE ROW触发器，分区表上不需要。
- 不允许在同一个分区树中混杂临时关系和持久关系。因此，如果分区表是持久的，则其分区也必须是持久的，反之亦然。在使用临时关系时，分区数的所有成员都必须来自于同一个会话。

5.10.3. 使用继承实现

虽然内建的声明式分区适合于大部分常见的用例，但还是有一些场景需要更加灵活的方法。分区可以使用表继承来实现，这能够带来一些声明式分区不支持的特性，例如：

- 对声明式分区来说，分区必须具有和分区表正好相同的列集合，而在表继承中，子表可以有父表中没有出现过的额外列。

- 表继承允许多继承。
- 声明式分区仅支持范围、列表以及哈希分区，而表继承允许数据按照用户的选择来划分（不过注意，如果约束排除不能有效地剪枝子表，查询性能可能会很差）。
- 在使用声明式分区时，一些操作比使用表继承时要求更长的持锁时间。例如，向分区表中增加分区或者从分区表移除分区要求在父表上取得一个ACCESS EXCLUSIVE锁，而在常规继承的情况下一个SHARE UPDATE EXCLUSIVE锁就足够了。

5.10.3.1. 例子

我们使用上面用过的同一个measurement表。为了使用继承实现分区，可使用下面的步骤：

1. 创建“主”表，所有的“子”表都将从它继承。这个表将不包含数据。不要在这个表上定义任何检查约束，除非想让它们应用到所有的子表上。同样，在这个表上定义索引或者唯一约束也没有意义。对于我们的例子来说，主表是最初定义的measurement表。
2. 创建数个“子”表，每一个都从主表继承。通常，这些表将不会在从主表继承的列集合之外增加任何列。正如声明性分区那样，这些表就是普通的PostgreSQL表（或者外部表）。

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. 为子表增加不重叠的表约束来定义每个分区允许的键值。

   典型的例子是：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

   确保约束能保证不同子表允许的键值之间没有重叠。设置范围约束的常见错误：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

   这是错误的，因为不清楚键值200属于哪一个子表。

   像下面这样创建子表会更好：

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
```

```
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 对于每个子表，在键列上创建一个索引，以及任何想要的其他索引。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. 我们希望我们的应用能够使用INSERT INTO measurement ...并且数据将被重定向到合适的分区表。我们可以通过为主表附加一个合适的触发器函数来实现这一点。如果数据将只被增加到最后一个分区，我们可以使用一个非常简单的触发器函数：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

完成函数创建后，我们创建一个调用该触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

我们必须在每个月重新定义触发器函数，这样它才会总是指向当前的子表。而触发器的定义则不需要被更新。

我们也可能希望插入数据时服务器会自动地定位应该加入数据的子表。我们可以通过一个更复杂的触发器函数来实现之，例如：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the
 measurement_insert_trigger() function!';
    END IF;
```

```
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

触发器的定义和以前一样。注意每一个IF测试必须准确地匹配它的子表的CHECK约束。

当该函数比单月形式更加复杂时，并不需要频繁地更新它，因为可以在需要的时候提前加入分支。

> **注意**
>
> 在实践中，如果大部分插入都会进入最新的子表，最好先检查它。为了简洁，我们为触发器的检查采用了和本例中其他部分一致的顺序。

把插入重定向到一个合适的子表中的另一种不同方法是在主表上设置规则而不是触发器。例如：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

规则的开销比触发器大很多，但是这种开销是每个查询只有一次，而不是每行一次，因此这种方法可能对批量插入的情况有优势。不过，在大部分情况下，触发器方法将提供更好的性能。

注意COPY会忽略规则。如果想要使用COPY插入数据，则需要拷贝到正确的子表而不是直接放在主表中。COPY会引发触发器，因此在使用触发器方法时可以正常使用它。

规则方法的另一个缺点是，如果规则集合无法覆盖插入日期，则没有简单的方法能够强制产生错误，数据将会无声无息地进入到主表中。

6. 确认constraint_exclusion配置参数在postgresql.conf中没有被禁用，否则将会不必要地访问子表。

如我们所见，一个复杂的表层次可能需要大量的DDL。在上面的例子中，我们可能为每个月创建一个新的子表，因此编写一个脚本来自动生成所需要的DDL可能会更好。

5.10.3.2. 继承分区的维护

要快速移除旧数据，只需要简单地去掉不再需要的子表：

```
DROP TABLE measurement_y2006m02;
```

要从继承层次表中去掉子表，但还是把它当做一个表保留：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

要增加一个新子表来处理新数据，可以像上面创建的原始子表那样创建一个空的子表：

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

或者，用户可能想要创建新子表并且在将它加入到表层次之前填充它。这可以允许数据在被父表上的查询可见之前对数据进行装载、检查以及转换。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.10.3.3. 提醒

下面的提醒适用于用继承实现的分区：

- 没有自动的方法啊验证所有的CHECK约束之间是否互斥。编写代码来产生子表以及创建和修改相关对象比手写命令要更加安全。
- 这里展示的模式假定行的键列值从不改变，或者说改变不足以让行移动到另一个分区。由于CHECK约束的存在，尝试那样做的UPDATE将会失败。如果需要处理那种情况，可以在子表上放置适当的更新触发器，但是那会使对结构的管理更加复杂。
- 如果使用手工的VACUUM或者ANALYZE命令，不要忘记需要在每个子表上单独运行它们。这样的命令：

  ```
  ANALYZE measurement;
  ```

  将只会处理主表。
- 带有ON CONFLICT子句的INSERT语句不太可能按照预期工作，因为只有在指定的目标关系而不是其子关系上发生唯一违背时才会采取ON CONFLICT行动。
- 将会需要触发器或者规则将行路由到想要的子表中，除非应用明确地知道分区的模式。编写触发器可能会很复杂，并且会比声明式分区在内部执行的元组路由慢很多。

5.10.4. 分区剪枝

分区剪枝是一种提升声明式分区表性能的查询优化技术。例如：

```
SET enable_partition_pruning = on;                 -- the default
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

如果没有分区剪枝，上面的查询将会扫描measurement表的每一个分区。如果启用了分区剪枝，规划器将会检查每个分区的定义并且检验该分区是否因为不包含符合查询WHERE子句的行而无需扫描。当规划器可以证实这一点时，它会把分区从查询计划中排除（剪枝）。

通过使用EXPLAIN命令和enable_partition_pruning配置参数，可以展示剪枝掉分区的计划与没有剪枝的计划之间的差别。对这种类型的表设置，一种典型的未优化计划是：

```
SET enable_partition_pruning = off;
```

```
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

某些或者全部的分区可能会使用索引扫描取代全表顺序扫描，但是这里的重点是根本不需要扫描较老的分区来回答这个查询。当我们启用分区剪枝时，我们会得到一个便宜很多的计划，而它能给出相同的答案：

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Append  (cost=0.00..36.21 rows=617 width=0)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617
 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

注意，分区剪枝仅由分区键隐式定义的约束所驱动，而不是由索引的存在驱动。因此，没有必要在键列上定义索引。是否需要为一个给定分区创建索引取决于预期的查询扫描该分区时会扫描大部分还是小部分。后一种情况下索引的帮助会比前者大。

不仅在给定查询的规划期间可以执行分区剪枝，在其执行期间也能执行分区剪枝。这非常有用，因为如果子句中包含查询规划时值未知的表达式时，这可以剪枝掉更多的分区，例如在PREPARE语句中定义的参数会使用从子查询拿到的值或者嵌套循环连接内侧关系上的参数化值。执行期间的分区剪枝可能在下列任何时刻执行：

- 在查询计划的初始化期间。对于执行的初始化阶段就已知值的参数，可以在这里执行分区剪枝。这个阶段中被剪枝掉的分区将不会显示在查询的EXPLAIN或EXPLAIN ANALYZE结果中。通过观察EXPLAIN输出的“Subplans Removed”属性，可以确定被剪枝掉的分区数。

- 在查询计划的实际执行期间。这里可以使用只有在实际查询执行时才能知道的值执行分区剪枝。这包括来自子查询的值以及来自执行时参数的值（例如来自于参数化嵌套循环连接的参数）。由于在查询执行期间这些参数的值可能会改变多次，所以只要分区剪枝使用到的执行参数发生改变，就会执行一次分区剪枝。要判断分区是否在这个阶段被剪枝，需要仔细地观察EXPLAIN ANALYZE输出中的loops属性。 对应于不同分区的子计划可以具有不同的值，这取决于在执行期间每个分区被修剪的次数。 如果每次都被剪枝，有些分区可能会显示为(never executed)。

可以使用enable_partition_pruning设置禁用分区剪枝。

> **注意**
>
> 当前，UPDATE　or　DELETE命令规划时的分区剪枝采用约束排除方法实现（但是，它是由enable_partition_pruning而不是constraint_exclusion控制　— 其细节及相应的提醒请参考接下来的小节。
>
> 此外，执行时的分区剪枝当前仅发生在Append节点类型上，对MergeAppend则不会。
>
> 这些行为都很可能在未来的PostgreSQL发行中被改变。

5.10.5.　分区和约束排除

约束排除是一种与分区剪枝类似的查询优化技术。虽然它主要被用于使用传统继承方法实现的分区上，但它也可以被用于其他目的，包括用于声明式分区。

约束排除以非常类似于分区剪枝的方式工作，不过它使用每个表的CHECK约束 — 这也是它得名的原因　— 而分区剪枝使用表的分区边界，分区边界仅存在于声明式分区的情况中。另一点不同之处是约束排除仅在规划时应用，在执行时不会尝试移除分区。

由于约束排除使用CHECK约束，这导致它比分区剪枝要慢，但有时候可以被当作一种优点加以利用：因为甚至可以在声明式分区的表上（在分区边界之外）定义约束，约束排除可能可以从查询计划中消去额外的分区。

constraint_exclusion的默认（也是推荐的）设置不是on也不是off，而是一种被称为partition的中间设置，这会导致该技术仅被应用于可能工作在继承分区表上的查询。on设置导致规划器检查所有查询中的CHECK约束，甚至是那些不太可能受益的简单查询。

下列提醒适用于约束排除：

- 约束排除仅适用于查询规划期间，和分区剪枝不同，它不适用于查询执行期间。
- 只有查询的WHERE子句包含常量（或者外部提供的参数）时，约束排除才能有效果。例如，针对一个非不变函数（如CURRENT_TIMESTAMP）的比较不能被优化，因为规划器不知道该函数的值在运行时会落到哪个子表中。
- 保持分区约束简单化，否则规划器可能无法验证哪些子表可能不需要被访问。如前面的例子所示，对列表分区使用简单的等值条件，对范围分区使用简单的范围测试。一种好的经验规则是分区约束应该仅包含分区列与常量使用B-树的可索引操作符的比较，因为只有B-树的可索引列才允许出现在分区键中。
- 约束排除期间会检查父表的所有子表上的所有约束，因此大量的子表很可能明显地增加查询规划时间。因此，传统的基于继承的分区可以很好地处理上百个子表，不要尝试使用上千个子表。

5.11.　外部数据

PostgreSQL实现了部分的SQL/MED规定，允许我们使用普通SQL查询来访问位于PostgreSQL之外的数据。这种数据被称为外部数据（注意这种用法不要和外键混淆，后者是数据库中的一种约束）。

外部数据可以在一个外部数据包装器的帮助下被访问。一个外部数据包装器是一个库，它可以与一个外部数据源通讯，并隐藏连接到数据源和从它获取数据的细节。在contrib模块中有一些外部数据包装器，参见附录　F 其他类型的外部数据包装器可以在第三方产品中找

到。如果这些现有的外部数据包装器都不能满足你的需要，可以自己编写一个，参见第 57 章

要访问外部数据，我们需要建立一个外部服务器对象，它根据它所支持的外部数据包装器所使用的一组选项定义了如何连接到一个特定的外部数据源。接着我们需要创建一个或多个外部表，它们定义了外部数据的结构。一个外部表可以在查询中像一个普通表一样地使用，但是在PostgreSQL服务器中外部表没有存储数据。不管使用什么外部数据包装器，PostgreSQL会要求外部数据包装器从外部数据源获取数据，或者在更新命令的情况下传送数据到外部数据源。

访问远程数据可能需要在外部数据源的授权。这些信息通过一个用户映射提供，它基于当前的PostgreSQL角色提供了附加的数据例如用户名和密码。

更多信息请见 CREATE FOREIGN DATA WRAPPER、 CREATE SERVER、 CREATE USER MAPPING、CREATE FOREIGN TABLE以及 IMPORT FOREIGN SCHEMA。

5.12. 其他数据库对象

表是一个关系型数据库结构中的核心对象，因为它们承载了我们的数据。但是它们并不是数据库中的唯一一种对象。有很多其他种类的对象可以被创建来使得数据的使用和刮泥更加方便或高效。在本章中不会讨论它们，但是我们在会给出一个列表：

- 视图
- 函数、过程和操作符
- 数据类型和域
- 触发器和重写规则

这些主题的详细信息请见第 V 部分

5.13. 依赖跟踪

当我们创建一个涉及到很多具有外键约束、视图、触发器、函数等的表的复杂数据库结构时，我们隐式地创建了一张对象之间的依赖关系网。例如，具有一个外键约束的表依赖于它所引用的表。

为了保证整个数据库结构的完整性，PostgreSQL确保我们无法删除仍然被其他对象依赖的对象。例如，尝试删除第 5.3.5 节的产品表会导致一个如下的错误消息，因为有订单表依赖于产品表：

```
DROP TABLE products;

ERROR:  cannot drop table products because other objects depend on it
DETAIL:  constraint orders_product_no_fkey on table orders depends on table
 products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

该错误消息包含了一个有用的提示：如果我们不想一个一个去删除所有的依赖对象，我们可以执行：

```
DROP TABLE products CASCADE;
```

这样所有的依赖对象将被移除，同样依赖于它们的任何对象也会被递归删除。在这种情况下，订单表不会被移除，但是它的外键约束会被移除。之所以在这里会停下，是因为没有什

么依赖着外键约束（如果希望检查DROP ... CASCADE会干什么，运行不带CASCADE的DROP并阅读DETAIL输出）。

PostgreSQL中的几乎所有DROP命令都支持CASCADE。当然，其本质的区别随着对象的类型而不同。我们也可以用RESTRICT代替CASCADE来获得默认行为，它将阻止删除任何被其他对象依赖的对象。

> **注意**
>
> 根据SQL标准，在DROP命令中指定RESTRICT或CASCADE是被要求的。但没有哪个数据库系统真正强制了这个规则，但是不同的系统中两种默认行为都是可能的。

如果一个DROP命令列出了多个对象，只有在存在指定对象构成的组之外的依赖关系时才需要CASCADE。例如，如果发出命令DROP TABLE tab1, tab2且存在从tab2到tab1的外键引用，那么就不需要CASCADE即可成功执行。

对于用户定义的函数，PostgreSQL会追踪与函数外部可见性质相关的依赖性，例如它的参数和结果类型，但不追踪检查函数体才能知道的依赖性。例如，考虑这种情况：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

（SQL 元函数的解释见第 38.5 节。PostgreSQL将会注意到get_color_note函数依赖于rainbow类型：删掉该类型会强制删除该函数，因为该函数的参数类型就无法定义了。但是PostgreSQL不会认为get_color_note依赖于my_colors表，因此即使该表被删除也不会删除这个函数。虽然这种方法有缺点，但是也有好处。如果该表丢失，这个函数在某种程度上仍然是有效的，但是执行它会导致错误。创建一个同名的新表将允许该函数重新有效。

第 6 章 数据操纵

前面的章节讨论了如何创建表和其他结构来保存你的数据。现在是时候给表填充数据了。本章涉及如何插入、更新和删除表数据。在接下来的一章将最终解释如何把你丢失已久的数据从数据库中抽取出来。

6.1. 插入数据

当一个表被创建后，它不包含数据。在数据库可以有点用之前要做的第一件事就是向里面插入数据。数据在概念上是以每次一行地方式被插入的。你当然可以每次插入多行，但是却没有办法一次插入少于一行的数据。即使你只知道几个列的值，那么你也必须创建一个完整的行。

要创建一个新行，使用INSERT命令。这条命令要求提供表的名字和其中列的值。例如，考虑第 5 章的产品表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

一个插入一行的命令将是：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据的值是按照这些列在表中出现的顺序列出的，并且用逗号分隔。通常，数据的值是文字（常量），但也允许使用标量表达式。

上面的语法的缺点是你必须知道表中列的顺序。要避免这个问题，你也可以显式地列出列。例如，下面的两条命令都有和上文那条 命令一样的效果：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为明确列出列的名字是个好习惯。

如果你没有获得所有列的值，那么你可以省略其中的一些。在这种情况下，这些列将被填充为它们的缺省值。例如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是PostgreSQL的一个扩展。它从使用给出的值从左开始填充列，有多少个给出的列值就填充多少个列，其他列的将使用缺省值。

为了保持清晰，你也可以显式地要求缺省值，用于单个的列或者用于整个行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

你可以在一个命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

也可以插入查询的结果（可能没有行、一行或多行）：

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

这提供了用于计算要插入的行的SQL查询机制（第 7 章的全部功能。

> **提示**
>
> 在一次性插入大量数据时，考虑使用COPY命令。它不如INSERT命令那么灵活，但是更高效。 参考第 14.4 获取更多有关批量装载性能的信息。

6.2. 更新数据

修改已经存储在数据库中的数据的行为叫做更新。你可以更新单个行，也可以更新表中所有的行，还可以更新其中的一部分行。 我们可以独立地更新每个列，而其他的列则不受影响。

要更新现有的行，使用UPDATE命令。这需要提供三部分信息：

1. 表的名字和要更新的列名
2. 列的新值
3. 要更新的是哪（些）行

我们在第 5 章说过，SQL 通常并不为行提供唯一标识符。因此我们无法总是直接指定需要更新哪一行。但是，我们可以通过指定一个被更新的行必须满足的条件。只有在表里面存在主键的时候（不管你声明它还是不声明它），我们才能可靠地通过选择一个匹配主键的条件来指定一个独立的行。图形化的数据库访问工具就靠这允许我们独立地更新某些行。

例如，这条命令把所有价格为5的产品的价格更新为10：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这样做可能导致零行、一行或者更多行被更新。如果我们试图做一个不匹配任何行的更新，那也不算错误。

让我们仔细看看这个命令。首先是关键字UPDATE， 然后跟着表名字。和平常一样，表名字也可以是用模式限定的， 否则会从路径中查找它。然后是关键字SET， 后面跟着列名、一个等号以及新的列值。新的列值可以是任意标量表达式， 而不仅仅是常量。例如，如果你想把所有产品的价格提高 10%，你可以用：

```
UPDATE products SET price = price * 1.10;
```

如你所见，用于新值的表达式也可以引用行中现有的值。我们还忽略了WHERE子句。如果我们忽略了这个子句， 那么就意味着表中的所有行都要被更新。如果出现了WHERE子句， 那么只有匹配它后面的条件的行被更新。请注意在SET子句中的等号是一个赋值， 而在WHERE子句中的等号是比较，不过这样并不会导致任何歧义。当然WHERE条件不一定非得是等值测试。许多其他操作符也都可以使用（参阅第 9 章。但是表达式必须得出一个布尔结果。

你还可以在一个UPDATE命令中更新更多的列， 方法是在SET子句中列出更多赋值。例如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. 删除数据

到目前为止我们已经解释了如何向表中增加数据以及如何改变数据。剩下的是讨论如何删除不再需要的数据。和前面增加数据一样，你也只能从表中整行整行地删除数据。在前面的一节里我们解释了 SQL 不提供直接访问单个行的方法。因此，删除行只能是通过指定被删除行必须匹配的条件进行。如果你在表上有一个主键，那么你可以指定准确的行。但是你也可以删除匹配条件的一组行，或者你可以一次从表中删除所有的行。

可以使用DELETE命令删除行，它的语法和UPDATE命令非常类似。例如，要从产品表中删除所有价格为 10 的产品，使用：

```
DELETE FROM products WHERE price = 10;
```

如果你只是写：

```
DELETE FROM products;
```

那么表中所有行都会被删除！程序员一定要注意。

6.4. 从修改的行中返回数据

有时在修改行的操作过程中获取数据很有用。INSERT、 UPDATE和DELETE命令都有一个支持这个的可选的 RETURNING子句。使用RETURNING 可以避免执行额外的数据库查询来收集数据，并且在否则难以可靠地识别修改的行时尤其有用。

所允许的RETURNING子句的内容与SELECT命令的输出列表相同 （请参阅第 7.3 节。它可以包含命令的目标表的列名， 或者包含使用这些列的值表达式。一个常见的简写是RETURNING *， 它按顺序选择目标表的所有列。

在INSERT中，可用于RETURNING的数据是插入的行。 这在琐碎的插入中并不是很有用，因为它只会重复客户端提供的数据。 但依赖于计算出的默认值时可以非常方便。例如，当使用serial列来提供唯一标识符时， RETURNING可以返回分配给新行的ID：

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

RETURNING子句对于INSERT ... SELECT也非常有用。

在UPDATE中，可用于RETURNING的数据是被修改行的新内容。 例如：

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

在DELETE中，可用于RETURNING的数据是删除行的内容。例如：

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
```

```
  RETURNING *;
```

如果目标表上有触发器(第 39 章 可用于RETURNING 的数据是被触发器修改的行。因此，检查由触发器计算的列是 RETURNING的另一个常见用例。

第 7 章 查询

前面的章节解释了如何创建表、如何用数据填充它们 以及如何操纵那些数据。现在我们终于可以讨论如何从数据库中检索数据了。

7.1. 概述

从数据库中检索数据的过程或命令叫做查询。在 SQL 里SELECT命令用于指定查询。SELECT命令的一般语法是

```
[WITH with_queries] SELECT select_list FROM table_expression
 [sort_specification]
```

下面几个小节描述选择列表、表表达式和排序声明的细节。WITH查询等高级特性将在最后讨论。

一个简单类型的查询的形式：

```
SELECT * FROM table1;
```

假设有一个表叫做table1，这条命令将table1中检索所有行和所有用户定义的列（检索的方法取决于客户端应用。例如，psql程序将在屏幕上显示一个 ASCII 形式的表格， 而客户端库将提供函数来从检索结果中抽取单个值）。 选择列表声明*意味着所有表表达式提供的列。 一个选择列表也可以选择可用列的一个子集或者在使用它们之前对列进行计算。例如，如果table1有叫做a、b和c的列（可能还有其他），那么你可以用下面的查询：

```
SELECT a, b + c FROM table1;
```

（假设b和c都是数字数据类型）。 参阅第 7.3 获取更多细节。

FROM table1是一种非常简单的表表达式：它只读取了一个表。通常，表表达式可以是基本表、连接和子查询组成的复杂结构。 但你也可以省略整个表表达式而把SELECT命令当做一个计算器：

```
SELECT 3 * 4;
```

如果选择列表里的表达式返回变化的结果，那么这就更有用了。例如，你可以用这种方法调用函数：

```
SELECT random();
```

7.2. 表表达式

表表达式计算一个表。该表表达式包含一个FROM子句，该子句后面可以根据需要选用WHERE、GROUP BY和HAVING子句。最简单的表表达式只是引用磁盘上的一个表，一个所谓的基本表，但是我们可以用更复杂的表表达式以多种方法修改或组合基本表。

表表达式里可选的WHERE、GROUP BY和HAVING子句指定一系列对源自FROM子句的表的转换操作。所有这些转换最后生成一个虚拟表，它提供行传递给选择列表计算查询的输出行。

7.2.1. FROM子句

FROM 子句从一个用逗号分隔的表引用列表中的一个或更多个其它表中生成一个表。

```
FROM table_reference [, table_reference [, ...]]
```

表引用可以是一个表名字（可能有模式限定）或者是一个生成的表， 例如子查询、一个JOIN结构或者这些东西的复杂组合。如果在FROM子句中引用了多于一个表， 那么它们被交叉连接（即构造它们的行的笛卡尔积，见下文）。FROM列表的结果是一个中间的虚拟表，该表可以进行由WHERE、GROUP BY和HAVING子句指定的转换，并最后生成全局的表表达式结果。

如果一个表引用是一个简单的表名字并且它是表继承层次中的父表，那么该表引用将产生该表和它的后代表中的行，除非你在该表名字前面放上ONLY关键字。但是，这种引用只会产生出现在该命名表中的列 — 在子表中增加的列都会被忽略。

除了在表名前写ONLY，你可以在表名后面写上*来显式地指定要包括所有的后代表。没有实际的理由再继续使用这种语法，因为搜索后代表现在总是默认行为。不过，为了保持与旧版本的兼容性，仍然支持这种语法。

7.2.1.1. 连接表

一个连接表是根据特定的连接类型的规则从两个其它表（真实表或生成表）中派生的表。目前支持内连接、外连接和交叉连接。一个连接表的一般语法是：

```
T1 join_type T2 [ join_condition ]
```

所有类型的连接都可以被链在一起或者嵌套：T1和T2都可以是连接表。在JOIN子句周围可以使用圆括号来控制连接顺序。如果不使用圆括号，JOIN子句会从左至右嵌套。

连接类型

交叉连接

```
T1 CROSS JOIN T2
```

对来自于T1和T2的行的每一种可能的组合（即笛卡尔积），连接表将包含这样一行：它由所有T1里面的列后面跟着所有T2里面的列构成。如果两个表分别有 N 和 M 行，连接表将有 N * M 行。

FROM T1 CROSS JOIN T2等效于FROM T1 INNER JOIN T2 ON TRUE（见下文）。它也等效于FROM T1,T2。

> **注意**
>
> 当多于两个表出现时，后一种等效并不严格成立，因为JOIN比逗号绑得更紧。例如FROM T1 CROSS JOIN T2 INNER JOIN T3 ON condition和FROM T1,T2 INNER JOIN T3 ON condition并不完全相同，因为第一种情况中的condition可以引用T1，但在第二种情况中却不行。

条件连接

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column
 list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNER和OUTER对所有连接形式都是可选的。INNER是缺省；LEFT、RIGHT和FULL指示一个外连接。

连接条件在ON或USING子句中指定， 或者用关键字NATURAL隐含地指定。连接条件决定来自两个源表中的哪些行是“匹配”的，这些我们将在后文详细解释。

可能的条件连接类型是：

INNER JOIN

对于 T1 的每一行 R1，生成的连接表都有一行对应 T2 中的每一个满足和 R1 的连接条件的行。

LEFT OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。因此，生成的连接表里为来自 T1 的每一行都至少包含一行。

RIGHT OUTER JOIN

首先，执行一次内连接。然后，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。因此，生成的连接表里为来自 T2 的每一行都至少包含一行。

FULL OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。同样，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。

ON子句是最常见的连接条件的形式：它接收一个和WHERE子句里用的一样的布尔值表达式。 如果两个分别来自T1和T2的行在ON表达式上运算的结果为真，那么它们就算是匹配的行。

USING是个缩写符号，它允许你利用特殊的情况：连接的两端都具有相同的连接列名。它接受共享列名的一个逗号分隔列表，并且为其中每一个共享列构造一个包含等值比较的连接条件。例如用USING (a, b)连接T1和T2会产生连接条件ON T1.a = T2.a AND T1.b = T2.b。

更进一步，JOIN USING的输出会废除冗余列：不需要把匹配上的列都打印出来，因为它们必须具有相等的值。不过JOIN ON会先产生来自T1的所有列，后面跟上所有来自T2的列；而JOIN USING会先为列出的每一个列对产生一个输出列，然后先跟上来自T1的剩余列，最后跟上来自T2的剩余列。

最后，NATURAL是USING的缩写形式：它形成一个USING列表， 该列表由那些在两个表里都出现了的列名组成。和USING一样，这些列只在输出表里出现一次。如果不存在公共列，NATURAL JOIN的行为将和JOIN ... ON TRUE一样产生交叉集连接。

> **注意**
>
> USING对于连接关系中的列改变是相当安全的，因为只有被列出的列会被组合成连接条件。NATURAL的风险更大，因为如果其中一个关系的模式改变会导致出现一个新的匹配列名，就会导致连接将新列也组合成连接条件。

为了解释这些问题，假设我们有一个表t1：

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

和t2：

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

然后我们用不同的连接方式可以获得各种结果：

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
```

```
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

用ON指定的连接条件也可以包含与连接不直接相关的条件。这种功能可能对某些查询很有用，但是需要我们仔细想清楚。例如：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

注意把限制放在WHERE子句中会产生不同的结果：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

这是因为放在ON子句中的一个约束在连接之前被处理，而放在WHERE子句中的一个约束是在连接之后被处理。这对内连接没有关系，但是对于外连接会带来麻烦。

7.2.1.2. 表和列别名

你可以给一个表或复杂的表引用指定一个临时的名字，用于剩下的查询中引用那些派生的表。这被叫做表别名。

要创建一个表别名，我们可以写：

```
FROM table_reference AS alias
```

或者

```
FROM table_reference alias
```

AS关键字是可选的。别名可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识符， 好让连接子句更易读。例如：

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON
 s.id = a.num;
```

到这里，别名成为当前查询的表引用的新名称 — 我们不再能够用该表最初的名字引用它了。因此，下面的用法是不合法的：

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- 错误
```

表别名主要用于简化符号，但是当把一个表连接到它自身时必须使用别名，例如：

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id =
 child.mother_id;
```

此外，如果一个表引用是一个子查询，则必须要使用一个别名（见第 7.2.1.3 节。

圆括弧用于解决歧义。在下面的例子中，第一个语句将把别名b赋给my_table的第二个实例，但是第二个语句把别名赋给连接的结果：

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种给表指定别名的形式是给表的列赋予临时名字，就像给表本身指定别名一样：

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

如果指定的列别名比表里实际的列少，那么剩下的列就没有被重命名。这种语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个JOIN子句的输出附加了一个别名， 那么该别名就在JOIN的作用下隐去了其原始的名字。例如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL，但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：表别名a在别名c外面是看不到的。

7.2.1.3. 子查询

子查询指定了一个派生表，它必须被包围在圆括弧里并且必须被赋予一个表别名（参阅第 7.2.1.2 节。例如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个例子等效于FROM table1 AS alias_name。更有趣的情况是在子查询里面有分组或聚集的时候， 子查询不能被简化为一个简单的连接。

一个子查询也可以是一个VALUES列表：

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

再次的，这里要求一个表别名。为VALUES列表中的列分配别名是可选的，但是选择这样做是一个好习惯。更多信息可参见第 7.7 节

7.2.1.4. 表函数

表函数是那些生成一个行集合的函数，这个集合可以是由基本数据类型（标量类型）组成，也可以是由复合数据类型（表行）组成。它们的用法类似一个表、视图或者在查询的FROM子句里的子查询。表函数返回的列可以像一个表列、视图或者子查询那样被包含在SELECT、JOIN或WHERE子句里。

也可以使用ROWS FROM语法将平行列返回的结果组合成表函数； 这种情况下结果行的数量是最大一个函数结果的数量，较小的结果会用空值来填充。

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias
 [(column_alias [, ... ])]]
```

如果指定了WITH ORDINALITY子句，一个额外的 bigint类型的列将会被增加到函数的结果列中。这个列对 函数结果集的行进行编号，编号从 1 开始（这是对 SQL 标准语法 UNNEST ... WITH ORDINALITY的一般化）。默认情 况下，序数列被称为ordinality，但也可以通过使用一个 AS子句给它分配一个不同的列名。

调用特殊的表函数UNNEST可以使用任意数量的数组参数， 它会返回对应的列数，就好像在每一个参数上单独调用 UNNEST（第 9.18 节并且使用 ROWS FROM结构把它们组合起来。

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias
 [(column_alias [, ... ])]]
```

如果没有指定table_alias，该函数名将被用作 表名。在ROWS FROM()结构的情况中，会使用第一个函数名。

如果没有提供列的别名，那么对于一个返回基数据类型的函数，列名也与该函数 名相同。对于一个返回组合类型的函数，结果列会从该类型的属性得到名称。

例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
```

```
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

有时侯，定义一个能够根据它们被调用方式返回不同列集合的表函数是很有用的。为了支持这些，表函数可以被声明为返回伪类型record。 如果在查询里使用这样的函数，那么我们必须在查询中指定所预期的行结构，这样系统才知道如何分析和规划该查询。这种语法是这样的：

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ] )
```

在没有使用ROWS FROM()语法时， column_definition列表会取代无法附着在 FROM项上的列别名列表，列定义中的名称就起到列别名的作用。 在使用ROWS FROM()语法时， 可以为每一个成员函数单独附着一个 column_definition列表；或者在只有一个成员 函数并且没有WITH ORDINALITY子句的情况下，可以在 ROWS FROM()后面写一个 column_definition列表来取代一个列别名列表。

考虑下面的例子：

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

dblink函数（dblink模块的一部分）执行一个远程的查询。它被声明为返回record，因为它可能会被用于任何类型的查询。 实际的列集必须在调用它的查询中指定，这样分析器才知道类似*这样的东西应该扩展成什么样子。

7.2.1.5. LATERAL子查询

可以在出现于FROM中的子查询前放置关键词LATERAL。这允许它们引用前面的FROM项提供的列（如果没有LATERAL，每一个子查询将被独立计算，并且因此不能被其他FROM项交叉引用）。

出现在FROM中的表函数的前面也可以被放上关键词LATERAL，但对于函数该关键词是可选的，在任何情况下函数的参数都可以包含对前面的FROM项提供的列的引用。

一个LATERAL项可以出现在FROM列表顶层，或者出现在一个JOIN树中。在后一种情况下，如果它出现在JOIN的右部，那么它也可以引用 在JOIN左部的任何项。

如果一个FROM项包含LATERAL交叉引用，计算过程如下：对于提供交叉引用列的FROM项的每一行，或者多个提供这些列的多个FROM项的行集合，LATERAL项将被使用该行或者行集中的列值进行计算。得到的结果行将和它们被计算出来的行进行正常的连接。对于来自这些列的源表的每一行或行集，该过程将重复。

LATERAL的一个简单例子：

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

这不是非常有用，因为它和一种更简单的形式得到的结果完全一样：

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

在必须要使用交叉引用列来计算那些即将要被连接的行时，LATERAL是最有用的。一种常用的应用是为一个返回集合的函数提供一个参数值。例如，假设vertices(polygon)返回一个多边形的顶点集合，我们可以这样标识存储在一个表中的多边形中靠近的顶点：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

这个查询也可以被写成：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

或者写成其他几种等价的公式（正如以上提到的，LATERAL关键词在这个例子中并不是必不可少的，但是我们在这里使用它是为了使表述更清晰）。

有时候也会很特别地把LEFT JOIN放在一个LATERAL子查询的前面，这样即使LATERAL子查询对源行不产生行，源行也会出现在结果中。例如，如果get_product_names()返回一个制造商制造的产品的名字，但是某些制造商在我们的表中目前没有制造产品，我们可以找出哪些制造商是这样：

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. WHERE子句

WHERE 子句的语法是

```
WHERE search_condition
```

这里的search_condition是任意返回一个boolean类型值的值表达式（参阅第 4.2 节。

在完成对FROM子句的处理之后，生成的虚拟表的每一行都会对根据搜索条件进行检查。 如果该条件的结果是真，那么该行被保留在输出表中；否则（也就是说，如果结果是假或空）就把它抛弃。搜索条件通常至少要引用一些在FROM子句里生成的列；虽然这不是必须的，但如果不引用这些列，那么WHERE子句就没什么用了。

> **注意**
>
> 内连接的连接条件既可以写在WHERE子句也可以写在JOIN子句里。例如，这些表表达式是等效的：
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```

和：

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

或者可能还有：

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

你想用哪个只是一个风格问题。FROM子句里的JOIN语法可能不那么容易移植到其它SQL数据库管理系统中。 对于外部连接而言没有选择：它们必须在FROM子句中完成。 外部连接的ON或USING子句不等于WHERE条件，因为它导致最终结果中行的增加（对那些不匹配的输入行）和减少。

这里是一些WHERE子句的例子：

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
 AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

在上面的例子里，fdt是从FROM子句中派生的表。 那些不符合WHERE子句的搜索条件的行会被从fdt中删除。请注意我们把标量子查询当做一个值表达式来用。 和任何其它查询一样，子查询里可以使用复杂的表表达式。同时还请注意fdt在子查询中也被引用。只有在c1也是作为子查询输入表的生成表的列时，才必须把c1限定成fdt.c1。但限定列名字可以增加语句的清晰度，即使有时候不是必须的。这个例子展示了一个外层查询的列名范围如何扩展到它的内层查询。

7.2.3. GROUP BY和HAVING子句

在通过了WHERE过滤器之后，生成的输入表可以使用GROUP BY子句进行分组，然后用HAVING子句删除一些分组行。

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

GROUP BY 子句被用来把表中在所列出的列上具有相同值的行分组在一起。 这些列的列出顺序并没有什么关系。其效果是把每组具有相同值的行组合为一个组行，它代表该组里的所有行。 这样就可以删除输出里的重复和/或计算应用于这些组的聚集。例如：

```
=> SELECT * FROM test1;
 x | y
---+---
```

```
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

在第二个查询里，我们不能写成SELECT * FROM test1 GROUP BY x， 因为列y里没有哪个值可以和每个组相关联起来。被分组的列可以在选择列表中引用是因为它们在每个组都有单一的值。

通常，如果一个表被分了组，那么没有在GROUP BY中列出的列都不能被引用，除非在聚集表达式中被引用。 一个用聚集表达式的例子是：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

这里的sum是一个聚集函数，它在整个组上计算出一个单一值。有关可用的聚集函数的更多信息可以在第 9.20 节

> **提示**
>
> 没有聚集表达式的分组实际上计算了一个列中可区分值的集合。我们也可以用DISTINCT子句实现（参阅第 7.3.3 节。

这里是另外一个例子：它计算每种产品的总销售额（而不是所有产品的总销售额）：

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

在这个例子里，列product_id、p.name和p.price必须在GROUP BY子句里， 因为它们都在查询的选择列表里被引用到（但见下文）。列s.units不必在GROUP BY列表里，因为它只是在一个聚集表达式（sum(...)）里使用，它代表一组产品的销售额。对于每种产品，这个查询都返回一个该产品的所有销售额的总和行。

如果产品表被建立起来，例如product_id是主键，那么在上面的例子中用product_id来分组就够了，因为名称和价格都是函数依赖于产品ID，并且关于为每个产品ID分组返回哪个名称和价格值就不会有歧义。

在严格的 SQL 里，GROUP BY只能对源表的列进行分组，但PostgreSQL把这个扩展为也允许GROUP BY去根据选择列表中的列分组。也允许对值表达式进行分组，而不仅是简单的列名。

如果一个表已经用GROUP BY子句分了组，然后你又只对其中的某些组感兴趣， 那么就可以用HAVING子句，它很象WHERE子句，用于从结果中删除一些组。其语法是：

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

在HAVING子句中的表达式可以引用分组的表达式和未分组的表达式（后者必须涉及一个聚集函数）。

例子：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

再次，一个更现实的例子：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

在上面的例子里，WHERE子句用那些非分组的列选择数据行（表达式只是对那些最近四周发生的销售为真）。 而HAVING子句限制输出为总销售收入超过 5000 的组。请注意聚集表达式不需要在查询中的所有地方都一样。

如果一个查询包含聚集函数调用，但是没有GROUP BY子句，分组仍然会发生：结果是一个单一行（或者根本就没有行，如果该单一行被HAVING所消除）。它包含一个HAVING子句时也是这样，即使没有任何聚集函数调用或者GROUP BY子句。

7.2.4. GROUPING SETS、CUBE和ROLLUP

使用分组集的概念可以实现比上述更加复杂的分组操作。由 FROM和WHERE子句选出的数据被按照每一个指定 的分组集单独分组，按照简单GROUP BY子句对每一个分组计算 聚集，然后返回结果。例如：

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)
```

```
=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS
 ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

GROUPING SETS的每一个子列表可以指定一个或者多个列或者表达式， 它们将按照直接出现在GROUP BY子句中同样的方式被解释。一个空的 分组集表示所有的行都要被聚集到一个单一分组（即使没有输入行存在也会被输出） 中，这就像前面所说的没有GROUP BY子句的聚集函数的情况一样。

对于分组列或表达式没有出现在其中的分组集的结果行，对分组列或表达式的引用会 被空值所替代。要区分一个特定的输出行来自于哪个分组，请见 表 9.56

PostgreSQL 中提供了一种简化方法来指定两种常用类型的分组集。下面形式的子句

```
ROLLUP ( e1, e2, e3, ... )
```

表示给定的表达式列表及其所有前缀（包括空列表），因此它等效于

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

这通常被用来分析历史数据，例如按部门、区和公司范围计算的总薪水。

下面形式的子句

```
CUBE ( e1, e2, ... )
```

表示给定的列表及其可能的子集（即幂集）。因此

```
CUBE ( a, b, c )
```

等效于

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
```

)

CUBE或ROLLUP子句中的元素可以是表达式或者　圆括号中的元素子列表。在后一种情况中，对于生成分组集的目的来说，子列 表被当做单一单元来对待。例如：

```
CUBE ( (a, b), (c, d) )
```

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

并且

```
ROLLUP ( a, (b, c), d )
```

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

CUBE和ROLLUP可以被直接用在 GROUP BY子句中，也可以被嵌套在一个 GROUPING SETS子句中。如果一个 GROUPING SETS子句被嵌套在另一个同类子句中， 效果和把内层子句的所有元素直接写在外层子句中一样。

如果在一个GROUP BY子句中指定了多个分组项，那么最终的 分组集列表是这些项的叉积。例如：

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

等效于

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

> **注意**
>
> 在表达式中，结构(a, b)通常被识别为一个 a 行构造器。在 GROUP BY子句中，这不会在表达式的顶层应用，并且 (a, b)会按照上面所说的被解析为一个表达式的列表。如果出于　某种原因你在分组表达式中需要一个行构造器，请使用 ROW(a, b)。

7.2.5. 窗口函数处理

如果查询包含任何窗口函数（见第 3.5 节第 9.21 和第 4.2.8 节，这些函数将在任何分组、聚集和HAVING过滤被执行之后被计算。也就是说如果查询使用了任何聚集、GROUP BY或HAVING，则窗口函数看到的行是分组行而不是来自于FROM/WHERE的原始表行。

当多个窗口函数被使用，所有在窗口定义中有句法上等效的PARTITION BY和ORDER BY子句的窗口函数被保证在数据上的同一趟扫描中计算。因此它们将会看到相同的排序顺序，即使ORDER BY没有唯一地决定一个顺序。但是，对于具有不同PARTITION BY或ORDER BY定义的函数的计算没有这种保证（在这种情况中，在多个窗口函数计算之间通常要求一个排序步骤，并且并不保证保留行的顺序，即使它的ORDER BY把这些行视为等效的）。

目前，窗口函数总是要求排序好的数据，并且这样查询的输出总是被根据窗口函数的PARTITION BY/ORDER BY子句的一个或者另一个排序。但是，我们不推荐依赖于此。如果你希望确保结果以特定的方式排序，请显式使用顶层的ORDER BY子句。

7.3. 选择列表

如前面的小节说明的那样， 在SELECT命令里的表表达式构造了一个中间的虚拟表， 方法可能有组合表、视图、消除行、分组等等。这个表最后被选择列表传递下去处理。选择列表判断中间表的哪个列是实际输出。

7.3.1. 选择列表项

最简单的选择列表类型是*，它发出表表达式生成的所有列。否则，一个选择列表是一个逗号分隔的值表达式的列表（和在第 4.2 章定义的一样）。 例如，它可能是一个列名的列表：

```
SELECT a, b, c FROM ...
```

列名字a、b和c要么是在FROM子句里引用的表中列的实际名字，要么是像第 7.2.1.2 章解释的那样的别名。在选择列表里可用的名字空间和在WHERE子句里的一样， 除非你使用了分组，这时候它和HAVING子句一样。

如果超过一个表有同样的列名，那么你还必须给出表名字，如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

在使用多个表时，要求一个特定表的所有列也是有用的：

```
SELECT tbl1.*, tbl2.a FROM ...
```

更多有关table_name.*记号的内容请参考第 8.16.5 节

如果将任意值表达式用于选择列表，那么它在概念上向返回的表中增加了一个新的虚拟列。值表达式为结果的每一行进行一次计算，对任何列引用替换行的值。 不过选择列表中的这个表达式并非一定要引用来自FROM子句中表表达式里面的列，例如它也可以是任意常量算术表达式。

7.3.2. 列标签

选择列表中的项可以被赋予名字，用于进一步的处理。 例如为了在一个ORDER BY子句中使用或者为了客户端应用显示。例如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用AS指定输出列名，那么系统会分配一个缺省的列名。对于简单的列引用， 它是被引用列的名字。对于函数调用，它是函数的名字。对于复杂表达式，系统会生成一个通用的名字。

只有在新列无法匹配任何PostgreSQL关键词（见附录 C 时，AS关键词是可选的。为了避免一个关键字的意外匹配，你可以使用双引号来修饰列名。例如，VALUE是一个关键字，所以下面的语句不会工作：

```
SELECT a value, b + c AS sum FROM ...
```

但是这个可以：

```
SELECT a "value", b + c AS sum FROM ...
```

为了防止未来可能的关键词增加，我们推荐总是写AS或者用双引号修饰输出列名。

> **注意**
>
> 输出列的命名和在FROM子句里的命名是不一样的 （参阅第 7.2.1.2 节。 它实际上允许你对同一个列命名两次，但是在选择列表中分配的名字是要传递下去的名字。

7.3.3. DISTINCT

在处理完选择列表之后，结果表可以可选的删除重复行。我们可以直接在SELECT后面写上DISTINCT关键字来指定：

```
SELECT DISTINCT select_list ...
```

（如果不用DISTINCT你可以用ALL关键词来指定获得的所有行的缺省行为）。

显然，如果两行里至少有一个列有不同的值，那么我们认为它是可区分的。空值在这种比较中被认为是相同的。

另外，我们还可以用任意表达式来判断什么行可以被认为是可区分的：

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

这里expression是任意值表达式，它为所有行计算。如果一个行集合里所有表达式的值是一样的， 那么我们认为它们是重复的并且因此只有第一行保留在输出中。请注意这里的一个集合的“第一行”是不可预料的， 除非你在足够多的列上对该查询排了序，保证到达DISTINCT过滤器的行的顺序是唯一的（DISTINCT ON处理是发生在ORDER BY排序后面的）。

DISTINCT ON子句不是 SQL 标准的一部分， 有时候有人认为它是一个糟糕的风格，因为它的结果是不可判定的。 如果有选择的使用GROUP BY和在FROM中的子查询，那么我们可以避免使用这个构造， 但是通常它是更方便的候选方法。

7.4. 组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是

```
query1 UNION [ALL] query2
```

```
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1和query2都是可以使用以上所有特性的查询。集合操作也可以嵌套和级连，例如

```
query1 UNION query2 UNION query3
```

实际执行的是：

```
(query1 UNION query2) UNION query3
```

UNION有效地把query2的结果附加到query1的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，它将删除结果中所有重复的行， 就象DISTINCT做的那样，除非你使用了UNION ALL。

INTERSECT返回那些同时存在于query1和query2的结果中的行，除非声明了INTERSECT ALL，否则所有重复行都被消除。

EXCEPT返回所有在query1的结果中但是不在query2的结果中的行（有时侯这叫做两个查询的差）。同样的，除非声明了EXCEPT ALL，否则所有重复行都被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，也就意味着它们都返回同样数量的列， 并且对应的列有兼容的数据类型，如第 10.5 节描述的那样。

7.5. 行排序

在一个查询生成一个输出表之后（在处理完选择列表之后），还可以选择性地对它进行排序。如果没有选择排序，那么行将以未指定的顺序返回。 这时候的实际顺序将取决于扫描和连接计划类型以及行在磁盘上的顺序，但是肯定不能依赖这些东西。一种特定的顺序只能在显式地选择了排序步骤之后才能被保证。

ORDER BY子句指定了排序顺序：

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

排序表达式可以是任何在查询的选择列表中合法的表达式。一个例子是：

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

当多于一个表达式被指定，后面的值将被用于排序那些在前面值上相等的行。每一个表达式后可以选择性地放置一个ASC或DESC关键词来设置排序方向为升序或降序。ASC顺序是默认值。升序会把较小的值放在前面，而“较小”则由<操作符定义。相似地，降序则由>操作符定义。 [1]

NULLS FIRST和NULLS LAST选项将可以被用来决定在排序顺序中，空值是出现在非空值之前或者出现在非空值之后。默认情况下，排序时空值被认为比任何非空值都要大，即NULLS FIRST是DESC顺序的默认值，而不是NULLS LAST的默认值。

注意顺序选项是对每一个排序列独立考虑的。例如ORDER BY x, y DESC表示ORDER BY x ASC, y DESC，而和ORDER BY x DESC, y DESC不同。

[1] 事实上，PostgreSQL为表达式的数据类型使用默认B-tree操作符类来决定ASC和DESC的排序顺序。照惯例，数据类型将被建立，这样<和>操作符负责这个排序顺序，但是一个用户定义的数据类型的设计者可以选择做些不同的设置。

一个sort_expression也可以是列标签或者一个输出列的编号，如：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

两者都根据第一个输出列排序。注意一个输出列的名字必须孤立，即它不能被用在一个表达式中 — 例如，这是不正确的：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- 错误
```

该限制是为了减少混淆。如果一个ORDER BY项是一个单一名字并且匹配一个输出列名或者一个表表达式的列，仍然会出现混淆。在这种情况中输出列将被使用。只有在你使用AS来重命名一个输出列来匹配某些其他表列的名字时，这才会导致混淆。

ORDER BY可以被应用于UNION、INTERSECT或EXCEPT组合的结果，但是在这种情况中它只被允许根据输出列名或编号排序，而不能根据表达式排序。

7.6. LIMIT和OFFSET

LIMIT和OFFSET允许你只检索查询剩余部分产生的行的一部分：

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

如果给出了一个限制计数，那么会返回数量不超过该限制的行（但可能更少些，因为查询本身可能生成的行数就比较少）。LIMIT ALL的效果和省略LIMIT子句一样，就像是LIMIT带有 NULL 参数一样。

OFFSET说明在开始返回行之前忽略多少行。OFFSET 0的效果和省略OFFSET子句是一样的，并且LIMIT NULL的效果和省略LIMIT子句一样，就像是OFFSET带有 NULL 参数一样。

如果OFFSET和LIMIT都出现了， 那么在返回LIMIT个行之前要先忽略OFFSET行。

如果使用LIMIT，那么用一个ORDER BY子句把结果行约束成一个唯一的顺序是很重要的。否则你就会拿到一个不可预料的该查询的行的子集。你要的可能是第十到第二十行，但以什么顺序的第十到第二十？除非你指定了ORDER BY，否则顺序是不知道的。

查询优化器在生成查询计划时会考虑LIMIT，因此如果你给定LIMIT和OFFSET，那么你很可能收到不同的规划（产生不同的行顺序）。因此，使用不同的LIMIT/OFFSET值选择查询结果的不同子集将生成不一致的结果，除非你用ORDER BY强制一个可预测的顺序。这并非bug， 这是一个很自然的结果，因为 SQL 没有许诺把查询的结果按照任何特定的顺序发出，除非用了ORDER BY来约束顺序。

被OFFSET子句忽略的行仍然需要在服务器内部计算；因此，一个很大的OFFSET的效率可能还是不够高。

7.7. VALUES列表

VALUES提供了一种生成“常量表”的方法，它可以被使用在一个查询中而不需要实际在磁盘上创建一个表。语法是：

```
VALUES ( expression [, ...] ) [, ...]
```

每一个被圆括号包围的表达式列表生成表中的一行。列表都必须具有相同数据的元素（即表中列的数目），并且在每个列表中对应的项必须具有可兼容的数据类型。分配给结果的每一列的实际数据类型使用和UNION相同的规则确定（参见第 10.5 节。

一个例子：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

将会返回一个有两列三行的表。它实际上等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

在默认情况下，PostgreSQL将column1、column2等名字分配给一个VALUES表的列。这些列名不是由SQL标准指定的，并且不同的数据库系统的做法也不同，因此通常最好使用表别名列表来重写这些默认的名字，像这样：

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t
 (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

在句法上，后面跟随着表达式列表的VALUES列表被视为和

```
SELECT select_list FROM table_expression
```

一样，并且可以出现在SELECT能出现的任何地方。例如，你可以把它用作UNION的一部分，或者附加一个sort_specification（ORDER BY、LIMIT和/或OFFSET）给它。VALUES最常见的用途是作为一个INSERT命令的数据源，以及作为一个子查询。

更多信息请见VALUES。

7.8. WITH查询（公共表表达式）

WITH提供了一种方式来书写在一个大型查询中使用的辅助语句。这些语句通常被称为公共表表达式或CTE，它们可以被看成是定义只在一个查询中存在的临时表。在WITH子句中的每一个辅助语句可以是一个SELECT、INSERT、UPDATE或DELETE，并且WITH子句本身也可以被附加到一个主语句，主语句也可以是SELECT、INSERT、UPDATE或DELETE。

7.8.1. WITH中的SELECT

WITH中SELECT的基本价值是将复杂的查询分解称为简单的部分。一个例子：

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
```

```
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

它只显示在高销售区域每种产品的销售总额。WITH子句定义了两个辅助语句regional_sales和top_regions，其中regional_sales的输出用在top_regions中而top_regions的输出用在主SELECT查询。这个例子可以不用WITH来书写，但是我们必须要用两层嵌套的子SELECT。使用这种方法要更简单些。

可选的RECURSIVE修饰符将WITH从单纯的句法便利变成了一种在标准SQL中不能完成的特性。通过使用RECURSIVE，一个WITH查询可以引用它自己的输出。一个非常简单的例子是计算从1到100的整数合的查询：

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

一个递归WITH查询的通常形式总是一个非递归项，然后是UNION（或者UNION ALL），再然后是一个递归项，其中只有递归项能够包含对于查询自身输出的引用。这样一个查询可以被这样执行：

递归查询求值

1. 计算非递归项。对UNION（但不对UNION ALL），抛弃重复行。把所有剩余的行包括在递归查询的结果中，并且也把它们放在一个临时的工作表中。

2. 只要工作表不为空，重复下列步骤：

    a. 计算递归项，用当前工作表的内容替换递归自引用。对UNION（不是UNION ALL），抛弃重复行以及那些与之前结果行重复的行。将剩下的所有行包括在递归查询的结果中，并且也把它们放在一个临时的中间表中。

    b. 用中间表的内容替换工作表的内容，然后清空中间表。

> **注意**
>
> 严格来说，这个处理是迭代而不是递归，但是RECURSIVE是SQL标准委员会选择的术语。

在上面的例子中，工作表在每一步只有一个行，并且它在连续的步骤中取值从1到100。在第100步，由于WHERE子句导致没有输出，因此查询终止。

递归查询通常用于处理层次或者树状结构的数据。一个有用的例子是这个用于找到一个产品的直接或间接部件的查询，只要给定一个显示了直接包含关系的表：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

在使用递归查询时，确保查询的递归部分最终将不返回元组非常重要，否则查询将会无限循环。在某些时候，使用UNION替代UNION ALL可以通过抛弃与之前输出行重复的行来达到这个目的。不过，经常有循环不涉及到完全重复的输出行：它可能只需要检查一个或几个域来看相同点之前是否达到过。处理这种情况的标准方法是计算一个已经访问过值的数组。例如，考虑下面这个使用link域搜索表graph的查询：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果link关系包含环，这个查询将会循环。因为我们要求一个“depth”输出，仅仅将UNION ALL 改为UNION不会消除循环。反过来在我们顺着一个特定链接路径搜索时，我们需要识别我们是否再次到达了一个相同的行。我们可以项这个有循环倾向的查询增加两个列path和cycle：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[g.id],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || g.id,
      g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了阻止环，数组值对于它们自己的工作显示到达任何特定行的“path”也有用。

在通常情况下如果需要检查多于一个域来识别一个环，请用行数组。例如，如果我们需要比较域f1和f2：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[ROW(g.f1, g.f2)],
      false
```

```
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || ROW(g.f1, g.f2),
      ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **提示**
>
> 在通常情况下只有一个域需要被检查来识别一个环，可以省略ROW()语法。这允许使用一个简单的数组而不是一个组合类型数组，可以获得效率。

> **提示**
>
> 递归查询计算算法使用宽度优先搜索顺序产生它的输出。你可以通过让外部查询ORDER BY一个以这种方法构建的“path”，用来以深度优先搜索顺序显示结果。

当你不确定查询是否可能循环时，一个测试查询的有用技巧是在父查询中放一个LIMIT。例如，这个查询没有LIMIT时会永远循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

这会起作用，因为PostgreSQL的实现只计算WITH查询中被父查询实际取到的行。不推荐在生产中使用这个技巧，因为其他系统可能以不同方式工作。同样，如果你让外层查询排序递归查询的结果或者把它们连接成某种其他表，这个技巧将不会起作用，因为在这些情况下外层查询通常将尝试取得WITH查询的所有输出。

WITH查询的一个有用的特性是在每一次父查询的执行中它们只被计算一次，即使它们被父查询或兄弟WITH查询引用了超过一次。因此，在多个地方需要的昂贵计算可以被放在一个WITH查询中来避免冗余工作。另一种可能的应用是阻止不希望的多个函数计算产生副作用。但是，从另一方面来看，优化器不能将来自父查询的约束下推到WITH查询中而不是一个普通子查询。WITH查询通常将会被按照所写的方式计算，而不抑制父查询以后可能会抛弃的行（但是，如上所述，如果对查询的引用只请求有限数目的行，计算可能会提前停止）。

以上的例子只展示了和SELECT一起使用的WITH，但是它可以被以相同的方式附加在INSERT、UPDATE或DELETE上。在每一种情况中，它实际上提供了可在主命令中引用的临时表。

7.8.2. WITH中的数据修改语句

你可以在WITH中使用数据修改语句（INSERT、UPDATE或DELETE）。这允许你在同一个查询中执行多个而不同操作。一个例子：

```
WITH moved_rows AS (
```

```
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

这个查询实际上从products把行移动到products_log。WITH中的DELETE删除来自products的指定行，以它的RETURNING子句返回它们的内容，并且接着主查询读该输出并将它插入到products_log。

上述例子中好的一点是WITH子句被附加给INSERT，而没有附加给INSERT的子SELECT。这是必需的，因为数据修改语句只允许出现在附加给顶层语句的WITH子句中。不过，普通WITH可见性规则应用，这样才可能从子SELECT中引用到WITH语句的输出。

正如上述例子所示，WITH中的数据修改语句通常具有RETURNING子句（见第 6.4 节。它是RETURNING子句的输出，不是数据修改语句的目标表，它形成了剩余查询可以引用的临时表。如果一个WITH中的数据修改语句缺少一个RETURNING子句，则它形不成临时表并且不能在剩余的查询中被引用。但是这样一个语句将被执行。一个非特殊使用的例子：

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

这个例子将从表foo和bar中移除所有行。被报告给客户端的受影响行的数目可能只包括从bar中移除的行。

数据修改语句中不允许递归自引用。在某些情况中可以采取引用一个递归WITH的输出来操作这个限制，例如：

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

这个查询将会移除一个产品的所有直接或间接子部件。

WITH中的数据修改语句只被执行一次，并且总是能结束，而不管主查询是否读取它们所有（或者任何）的输出。注意这和WITH中SELECT的规则不同：正如前一小节所述，直到主查询要求SELECT的输出时，SELECT才会被执行。

The sub-statements in WITH中的子语句被和每一个其他子语句以及主查询并发执行。因此在使用WITH中的数据修改语句时，指定更新的顺序实际是以不可预测的方式发生的。所有的语句都使用同一个snapshot执行（参见第 13 章，因此它们不能“看见”在目标表上另一个执行的效果。这减轻了行更新的实际顺序的不可预见性的影响，并且意味着RETURNING数据是在不同WITH子语句和主查询之间传达改变的唯一方法。其例子

```
WITH t AS (
    UPDATE products SET price = price * 1.05
```

```
    RETURNING *
)
SELECT * FROM products;
```

外层SELECT可以返回在UPDATE动作之前的原始价格，而在

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

外部SELECT将返回更新过的数据。

在一个语句中试图两次更新同一行是不被支持的。只会发生一次修改，但是该办法不能很容易地（有时是不可能）可靠地预测哪一个会被执行。这也应用于删除一个已经在同一个语句中被更新过的行：只有更新被执行。因此你通常应该避免尝试在一个语句中尝试两次修改同一个行。尤其是防止书写可能影响被主语句或兄弟子语句修改的相同行。这样一个语句的效果将是不可预测的。

当前，在WITH中一个数据修改语句中被用作目标的任何表不能有条件规则、ALSO规则或INSTEAD规则，这些规则会扩展成为多个语句。

第 8 章 数据类型

PostgreSQL有着丰富的本地数据类型可用。用户可以使用CREATE TYPE命令为 PostgreSQL增加新的数据类型。

表 8.1显示了所有内建的普通数据类型。大部分在“别名”列里列出的可选名字都是因历史原因 被PostgreSQL在内部使用的名字。另外，还有一些内部使用的或者废弃的类型也可以用，但没有在这里列出。

表 8.1. 数据类型

名字	别名	描述
bigint	int8	有符号的8字节整数
bigserial	serial8	自动增长的8字节整数
bit [(n)]		定长位串
bit varying [(n)]	varbit [(n)]	变长位串
boolean	bool	逻辑布尔值（真/假）
box		平面上的普通方框
bytea		二进制数据（“字节数组”）
character [(n)]	char [(n)]	定长字符串
character varying [(n)]	varchar [(n)]	变长字符串
cidr		IPv4或IPv6网络地址
circle		平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数（8字节）
inet		IPv4或IPv6主机地址
integer	int, int4	有符号4字节整数
interval [fields] [(p)]		时间段
json		文本 JSON 数据
jsonb		二进制 JSON 数据，已分解
line		平面上的无限长的线
lseg		平面上的线段
macaddr		MAC（Media Access Control）地址
macaddr8		MAC（Media Access Control）地址（EUI-64格式）
money		货币数量
numeric [(p, s)]	decimal [(p, s)]	可选择精度的精确数字
path		平面上的几何路径
pg_lsn		PostgreSQL日志序列号
point		平面上的几何点
polygon		平面上的封闭几何路径
real	float4	单精度浮点数（4字节）
smallint	int2	有符号2字节整数

名字	别名	描述
smallserial	serial2	自动增长的2字节整数
serial	serial4	自动增长的4字节整数
text		变长字符串
time [(p)] [without time zone]		一天中的时间（无时区）
time [(p)] with time zone	timetz	一天中的时间，包括时区
timestamp [(p)] [without time zone]		日期和时间（无时区）
timestamp [(p)] with time zone	timestamptz	日期和时间，包括时区
tsquery		文本搜索查询
tsvector		文本搜索文档
txid_snapshot		用户级别事务ID快照
uuid		通用唯一标识码
xml		XML数据

兼容性

下列类型（或者及其拼写）是SQL指定的：bigint、bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time（有时区或无时区）、timestamp（有时区或无时区）、xml。

每种数据类型都有一个由其输入和输出函数决定的外部表现形式。许多内建的类型有明显的格式。不过，许多类型要么是PostgreSQL所特有的（例如几何路径），要么可能是有几种不同的格式（例如日期和时间类型）。 有些输入和输出函数是不可逆的，即输出函数的结果和原始输入比较时可能丢失精度。

8.1. 数字类型

数字类型由2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。表 8.2列出了所有可用类型。

表 8.2. 数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 to +32767
integer	4字节	整数的典型选择	-2147483648 to +2147483647
bigint	8字节	大范围整数	-9223372036854775808 to +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位

名字	存储尺寸	描述	范围
numeric	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确	15位十进制精度
smallserial	2字节	自动增加的小整数	1到32767
serial	4字节	自动增加的整数	1到2147483647
bigserial	8字节	自动增长的大整数	1到9223372036854775807

数字类型常量的语法在第 4.1.2 节描述。数字类型有一整套对应的数学操作符和函数。相关信息请参考 第 9 章下面的几节详细描述这些类型。

8.1.1. 整数类型

类型smallint、integer和bigint存储各种范围的全部是数字的数，也就是没有小数部分的数字。试图存储超出范围以外的值将导致一个错误。

常用的类型是integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。一般只有在磁盘空间紧张的时候才使用 smallint类型。而只有在integer的范围不够的时候才使用bigint。

SQL只声明了整数类型integer（或int）、smallint和bigint。类型int2、int4和int8都是扩展，也在许多其它SQL数据库系统中使用。

8.1.2. 任意精度数字

类型numeric可以存储非常多位的数字。我们特别建议将它用于货币金额和其它要求计算准确的数量。numeric值的计算在可能的情况下会得到准确的结果，例如加法、减法、乘法。不过，numeric类型上的算术运算比整数类型或者下一节描述的浮点数类型要慢很多。

在随后的内容里，我们使用了下述术语：一个numeric的precision（精度）是整个数中有效位的总数，也就是小数点两边的位数。numeric的scale（刻度）是小数部分的数字位数，也就是小数点右边的部分。因此数字 23.5141 的精度为6而刻度为4。可以认为整数的刻度为零。

numeric列的最大精度和最大比例都是可以配置的。要声明一个类型为numeric的列，你可以用下面的语法：

```
NUMERIC(precision, scale)
```

精度必须为正数，比例可以为零或者正数。另外：

```
NUMERIC(precision)
```

选择比例为 0 。如果使用

```
NUMERIC
```

创建一个列时不使用精度或比例，则该列可以存储任何精度和比例的数字值，并且值的范围最多可以到实现精度的上限。一个这种列将不会把输入值转化成任何特定的比例，而带有比例声明的numeric列将把输入值转化为该比例（SQL标准要求缺省的比例是 0，即转化成整数

精度。我们觉得这样做有点没用。如果你关心移植性，那你最好总是显式声明精度和比例）。

> **注意**
>
> 显式指定类型精度时的最大允许精度为 1000，没有指定精度的NUMERIC受到表 8.2中描述的限制所控制。

如果一个要存储的值的比例比列声明的比例高，那么系统将尝试圆整（四舍五入）该值到指定的分数位数。 然后，如果小数点左边的位数超过了声明的精度减去声明的比例，那么抛出一个错误。

数字值在物理上是以不带任何前导或者后缀零的形式存储。 因此，列上声明的精度和比例都是最大值，而不是固定分配的 （在这个方面，numeric类型更类似于varchar(n)， 而不像char(n)）。 实际存储要求是每四个十进制位组用两个字节，再加上三到八个字节的开销。

除了普通的数字值之外，numeric类型允许特殊值NaN， 表示“不是一个数字”。任何在NaN上面的操作都生成另外一个NaN。 如果在 SQL 命令里把这些值当作一个常量写，你必须在其周围放上单引号，例如UPDATE table SET x = 'NaN'。在输入时，字串NaN被识别为大小写无关。

> **注意**
>
> 在“不是一个数字”概念的大部分实现中，NaN被认为不等于任何其他数字值（包括NaN）。为了允许numeric值可以被排序和使用基于树的索引，PostgreSQL把NaN值视为相等，并且比所有非NaN值都要大。

类型decimal和numeric是等效的。两种类型都是SQL标准的一部分。

在对值进行圆整时，numeric类型会圆到远离零的整数，而（在大部分机器上）real和double precision类型会圆到最近的偶数上。例如：

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. 浮点类型

数据类型real和double precision是不准确的、变精度的数字类型。实际上，这些类型是IEEE标准 754 二进制浮点算术（分别对应单精度和双精度）的一般实现， 一直到下层处理器、操作系统和编译器对它的支持。

不准确意味着一些值不能准确地转换成内部格式并且是以近似的形式存储的，因此存储和检索一个值可能出现一些缺失。 处理这些错误以及这些错误是如何在计算中传播的主题属于数学和计算机科学的一个完整的分支， 我们不会在这里进一步讨论它，这里的讨论仅限于如下几点：

- 如果你要求准确的存储和计算（例如计算货币金额），应使用numeric类型。
- 如果你想用这些类型做任何重要的复杂计算，尤其是那些你对范围情况（无穷、下溢）严重依赖的事情，那你应该仔细评估你的实现。
- 用两个浮点数值进行等值比较不可能总是按照期望地进行。

在大部分平台上，real类型的范围是至少 -1E+37 到 +1E+37，精度至少是 6 位小数。double precision类型通常有 -1E+308 到 +1E+308 的范围，精度是至少 15 位数字。太大或者太小的值都会导致错误。 如果输入数字的精度太高，那么可能发生园整。太接近零的数字，如果无法与零值的表现形式相区分就会产生下溢错误。

> **注意**
>
> extra_float_digits设置控制当一个浮点值被转换为文本输出时要包括的额外有效数字的数目。其默认值为0，在每一个PostgreSQL支持的平台上输出都相同。增加该设置将产生能更精确表示存储值的输出，但是可能无法移植。

除了普通的数字值之外，浮点类型还有几个特殊值：

```
Infinity
-Infinity
NaN
```

这些值分别表示 IEEE 754 特殊值“正无穷大”、“负无穷大”以及“不是一个数字”（在不遵循 IEEE 754 浮点算术的机器上，这些值的含义可能不是预期的）。如果在 SQL 命令里把这些数值当作常量写，你必须在它们周围放上单引号，例如UPDATE table SET x = '-Infinity'。 在输入时，这些串是以大小写无关的方式识别的。

> **注意**
>
> IEEE754指定NaN不应该与任何其他浮点值（包括NaN）相等。为了允许浮点值被排序或者在基于树的索引中使用，PostgreSQL将NaN值视为相等，并且比所有非NaN值要更大。

PostgreSQL还支持 SQL 标准表示法float和float(p)用于声明非精确的数字类型。在这里，p指定以二进制位表示的最低可接受精度。 在选取real类型的时候，PostgreSQL接受float(1)到float(24)，在选取double precision的时候，接受float(25)到float(53)。在允许范围之外的p值将导致一个错误。没有指定精度的float将被当作是double precision。

> **注意**
>
> 认为real和double precision分别有 24 和 53 个二进制位的假设对 IEEE 标准的浮点实现来说是正确的。在非 IEEE 平台上，这个数值可能略有偏差，但是为了简化，我们在所有平台上都用了同样的p值范围。

8.1.4. 序数类型

> **注意**
>
> 这一节描述了PostgreSQL特有的创建一个自增列的方法。另一种方法是使用SQL标准的标识列特性，它在CREATE TABLE中描述。

smallserial、serial和bigserial类型不是真正的类型，它们只是为了创建唯一标识符列而存在的方便符号（类似其它一些数据库中支持的AUTO_INCREMENT属性）。 在目前的实现中，下面一个语句：

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等价于以下语句：

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

因此，我们就创建了一个整数列并且把它的缺省值安排为从一个序列发生器取值。应用了一个NOT NULL约束以确保空值不会被插入（在大多数情况下你可能还希望附加一个UNIQUE或者PRIMARY KEY约束避免意外地插入重复的值，但这个不是自动发生的）。最后，该序列被标记为“属于”该列，这样当列或表被删除时该序列也会被删除。

> **注意**
>
> 因为smallserial、serial和bigserial是用序列实现的，所以即使没有删除过行，在出现在列中的序列值可能有“空洞”或者间隙。如果一个从序列中分配的值被用在一行中，即使该行最终没有被成功地插入到表中，该值也被“用掉”了。例如，当插入事务回滚时就会发生这种情况。更多信息参见第 9.16 节的nextval()。

要使用serial列插入序列的下一个数值到表中， 请指定serial列应该被赋予其缺省值。我们可以通过在INSERT语句中把该列排除在列列表之外来实现，也可以通过使用DEFAULT关键字来实现。

类型名serial和serial4是等效的： 两个都创建integer列。类型名bigserial和serial8也一样，只不过它们创建一个 bigint列。如果你预计在表的生存期中使用的标识符数目超过 2^{31} 个，那么你应该使用bigserial。类型名smallserial和serial2也以相同方式工作，只不过它们创建一个smallint列。

为一个serial列创建的序列在所属的列被删除的时候自动删除。你可以在不删除列的情况下删除序列，但是这会强制删除该列的默认值表达式。

8.2. 货币类型

money类型存储固定小数精度的货币数字，参阅表 8.3 小数的精度由数据库的lc_monetary设置决定。表中展示的范围假设有两个小数位。可接受的输入格式很多，包括整数和浮点数文字，以及常用的货币格式，如'$1,000.00'。 输出通常是最后一种形式，但和区域相关。

表 8.3. 货币类型

名字	存储尺寸	描述	范围
money	8 bytes	货币额	-92233720368547758.08 到 +92233720368547758.07

由于这种数据类型的输出是区域敏感的，因此将money数据装入到一个具有不同lc_monetary设置的数据库是不起作用的。为了避免这种问题，在恢复一个转储到一个新数据库中之前，应确保新数据库的lc_monetary设置和被转储数据库的相同或者具有等效值。

数据类型numeric、int和bigint的值可以被造型成money。从数据类型real和double precision的转换可以通过先造型成numeric来实现，例如：

```
SELECT '12.34'::float8::numeric::money;
```

但是，我们不推荐这样做。浮点数不应该被用来处理货币，因为浮点数可能会有圆整错误。

一个money值可以在不损失精度的情况下被造型成numeric。转换到其他类型可能会丢失精度，并且必须采用两个阶段完成：

```
SELECT '52093.89'::money::numeric::float8;
```

一个money值被一个整数值除的除法结果会被截去分数部分。要得到圆整的结果，可以除以一个浮点值，或者在除法之前把money转换成numeric然后在除法之后转回money（如果要避免精度丢失的风险则后者更好）。当一个money值被另一个money值除时，结果是double precision（即一个纯数字，而不是金额），在除法中货币单位被约掉了。

8.3. 字符类型

表 8.4. 字符类型

名字	描述
character varying(n), varchar(n)	有限制的变长
character(n), char(n)	定长，空格填充
text	无限变长

表 8.4显示了在PostgreSQL里可用的一般用途的字符类型。

SQL定义了两种基本的字符类型： character varying(n)和character(n)， 其中n是一个正整数。两种类型都可以存储最多n个字符长的串。试图存储更长的串到这些类型的列里会产生一个错误， 除非超出长度的字符都是空白，这种情况下该串将被截断为最大长度（这个看上去有点怪异的例外是SQL标准要求的）。 如果要存储的串比声明的长度短，类型为character的值将会用空白填满；而类型为character varying的值将只是存储短些的串。

如果我们明确地把一个值造型成character varying(n)或者character(n)， 那么超长的值将被截断成n个字符，而不会抛出错误（这也是SQL标准的要求）。

varchar(n)和char(n)的概念分别是character varying(n)和character(n)的别名。没有长度声明词的character等效于character(1)。如果不带长度说明词使用character varying，那么该类型接受任何长度的串。后者是一个PostgreSQL的扩展。

另外，PostgreSQL提供text类型，它可以存储任何长度的串。尽管类型text不是SQL标准，但是许多其它 SQL 数据库系统也有它。

类型character的值物理上都用空白填充到指定的长度n， 并且以这种方式存储和显示。不过，拖尾的空白被当作是没有意义的，并且在比较两个 character类型值时不会考虑它们。在空白有意义的排序规则中，这种行为可能会 产生意料之外的结果，例如SELECT 'a'::CHAR(2) collate "C" < E'a\n'::CHAR(2)会返回真（即便C区域会认为一个空格比新行更大）。当把一个character值转换成其他 字符串类型之一时，拖尾的空白会被移除。请注意，在character varying和text值里， 结尾的空白语意上是有含义的，并且在使用模式匹配（如LIKE和正则表达式）时也会被考虑。

这些类型的存储需求是 4 字节加上实际的字串，如果是 character 的话再加上填充的字节。长的字串将会自动被系统压缩， 因此在磁盘上的物理需求可能会更少些。长的数值也会存储在后台表里面，这样它们就不会干扰对短字段值的快速访问。 不管怎样，允许存储的最长字串大概是 1 GB。 （允许在数据类型声明中出现的的 n 的最大值比这还小。 修改这个行为没有甚么意义，因为在多字节编码下字符和字节的数目可能差别很大。 如果你想存储没有特定上限的长字串，那么使用 text 或者没有长度声明词的 character varying， 而不要选择一个任意长度限制。） 一个短串（最长126字节）的存储要求是1个字节外加实际的串，该串在character情况下包含填充的空白。长一些的串在前面需要4个字节而不是1个字节。长串会被系统自动压缩，这样在磁盘上的物理需求可能会更少。非常长的值也会被存储在背景表中，这样它们不会干扰对较短的列值的快速访问。在任何情况下，能被存储的最长的字符串是1GB（数据类型定义中n能允许的最大值比这个值要小。修改它没有用处，因为对于多字节字符编码来说，字符的数量和字节数可能完全不同。如果你想要存储没有指定上限的长串，使用text或没有长度声明的character varying，而不是给出一个任意长度限制）。

> **提示**
>
> 这三种类型之间没有性能差别，只不过是在使用填充空白的类型的时候需要更多存储尺寸，以及在存储到一个有长度约束的列时需要少量额外CPU周期来检查长度。虽然在某些其它的数据库系统里，character(n)有一定的性能优势，但在PostgreSQL里没有。事实上，character(n)通常是这三种类型之中最慢的一个，因为它需要额外的存储开销。在大多数情况下，应该使用text或者character varying。

请参考第 4.1.2.1 节取关于串文本的语法的信息，以及参阅第 9 章取关于可用操作符和函数的信息。 数据库的字符集决定用于存储文本值的字符集；有关字符集支持的更多信息，请参考第 23.3 节。

例 8.1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
```

```
   b    | char_length
-------+-------------
 ok     |           2
 good   |           5
 too l |           5
```

1 函数char_length在第 9.4 节讨论。

在PostgreSQL里另外还有两种定长字符类型，在表 8.5里显示。 name类型只用于在内部系统目录中存储标识符并且不是给一般用户使用的。该类型长度当前定为 64 字节（63 可用字符加结束符）但在C源代码应该使用常量 NAMEDATALEN引用。这个长度是在编译的时候设置的（因而可以为特殊用途调整），缺省的最大长度在以后的版本可能会改变。类型"char"（注意引号）和 char(1)是不一样的，它只用了一个字节的存储空间。它在系统内部用于系统目录当做简化的枚举类型用。

表 8.5. 特殊字符类型

名字	存储尺寸	描述
"char"	1字节	单字节内部类型
name	64字节	用于对象名的内部类型

8.4. 二进制数据类型

bytea数据类型允许存储二进制串，参见表 8.6

表 8.6. 二进制数据类型

名字	存储尺寸	描述
bytea	1或4字节外加真正的二进制串	变长二进制串

二进制串是一个八位位组（或字节）的序列。 二进制串和字符串的区别有两个： 首先，二进制串明确允许存储零值的字节以及其它“不可打印的”字节（通常是位于十进制范围32到126之外的字节）。 字符串不允许零字节，并且也不允许那些对于数据库的选定字符集编码是非法的任何其它字节值或者字节值序列。 第二，对二进制串的操作会处理实际上的字节，而字符串的处理和取决于区域设置。 简单说，二进制字串适用于存储那些程序员认为是“裸字节”的数据，而字符串适合存储文本。

bytea类型支持两种用于输入和输出的格式：“十六进制”格式和PostgreSQL的历史的“转义”格式。在输入时这两种格式总是会被接受。输出格式则取决于配置参数bytea_output，其默认值为十六进制（注意十六进制格式是在PostgreSQL 9.0中被引入的，早期的版本和某些工具无法理解它）。

SQL标准定义了一种不同的二进制串类型， 叫做BLOB或者BINARY LARGE OBJECT。其输入格式和bytea不同，但是提供的函数和操作符大多一样。

8.4.1. bytea的十六进制格式

“十六进制”格式将二进制数据编码为每个字节2个十六进制位，最高有效位在前。整个串以序列\x开头（用以和转义格式区分）。在某些情景中，开头的反斜线可能需要通过双写来转义，详见(see 第 4.1.2.1 节。 作为输入，十六进制位可以是大写也可以是小写，在位对之间可以有空白（但是在位对内部以及开头的\x序列中不能有空白）。十六进制格式和很多外部应用及协议相兼容，并且其转换速度要比转义格式更快，因此人们更愿意用它。

例子：

```
SELECT '\xDEADBEEF';
```

8.4.2. bytea的转义格式

“转义”格式是bytea类型的传统PostgreSQL格式。它采用将二进制串表示成ASCII字符序列的方法，而将那些无法用ASCII字符表示的字节转换成特殊的转义语句。从应用的角度来看，如果将字节表示为字符有意义，那么这种表示将很方便。但是在实际中，这常常是令人困扰的，因为它使二进制串和字符串之间的区别变得模糊，并且这种特别的转义机制也有点难于处理。因此这种格式可能会在大部分新应用中避免使用。

在转义模式下输入bytea值时，某些值的字节必须被转义，而所有的字节值都可以被转义。通常，要转义一个字节，需要把它转换成与它的三位八进制值， 并且前导一个反斜线。反斜线本身（十进制字节值92）也可以用双写的反斜线表示。表 8.7显示了必须被转义的字符，并给出了可以使用的替代转义序列。

表 8.7.bytea文字转义字节

十进制字节值	描述	转义输入表示	例子	十六进制表示
0	0字节	'\000'	SELECT '\000'::bytea;	\x00
39	单引号	''''或'\047'	SELECT ''''::bytea;	\x27
92	反斜线	'\\'或'\134'	SELECT '\\'::bytea;	\x5c
0到31和127到255	“不可打印的”字节	'\xxx'（八进制值）	SELECT '\001'::bytea;	\x01

转义“不可打印的”字节的要求取决于区域设置。在某些实例中，你可以不理睬它们，让它们保持未转义的状态。

如表 8.7中所示，要求单引号必须写两次的原因对任何SQL命令中的字符串常量都是一样的。 文字解析器消耗最外层的单引号，并缩减成对的单引号为一个普通数据字符。 bytea输入函数看到的只是一个单引号，它将其视为普通数据字符。 但是，bytea输入函数将反斜杠视为特殊字符，表 8.7中显示的其他行为由该函数实现。

在某些情况下，反斜杠必须加倍，如上所示，因为通用的字符串文字解析器也会 将一对反斜杠减少为一个数据字符;请参阅第 4.1.2.1 节。

Bytea字节默认被输出为hex格式。如果你把bytea_output改为escape，“不可打印的”字节会被转换成与之等效的三位八进制值并且前置一个反斜线。大部分“可打印的”字节被输出为它们在客户端字符集中的标准表示形式，例如：

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

十进制值为92（反斜线）的字节在输出时被双写。详情请见表 8.8。

表 8.8.bytea输出转义字节

十进制字节值	描述	转义的输出表示	例子	输出结果
92	反斜线	\\	SELECT '\134'::bytea;	\\

十进制字节值	描述	转义的输出表示	例子	输出结果
0到31和127到255	“不可打印的”字节	\xxx（八进制值）	SELECT '\001'::bytea;	\001
32到126	“可打印的”字节	客户端字符集表示	SELECT '\176'::bytea;	~

根据你使用的PostgreSQL前端，你在转义和未转义bytea串方面可能需要做额外的工作。例如，如果你的接口自动翻译换行和回车，你可能也不得不转义它们。

8.5. 日期/时间类型

PostgreSQL支持SQL中所有的日期和时间类型，如表 8.9所示。这些数据类型上可用的操作如第 9.9 节所述。日期根据公历来计算，即使对于该历法被引入之前的年份也一样（见第 B.5 节。

表 8.9. 日期/时间类型

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [(p)] [without time zone]	8字节	包括日期和时间（无时区）	4713 BC	294276 AD	1微秒
timestamp [(p)] with time zone	8字节	包括日期和时间，有时区	4713 BC	294276 AD	1微秒
date	4字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1日
time [(p)] [without time zone]	8字节	一天中的时间（无日期）	00:00:00	24:00:00	1微秒
time [(p)] with time zone	12字节	仅仅是一天中的时间（没有日期），带有时区	00:00:00+1459	24:00:00-1459	1微秒
interval [fields] [(p)]	16字节	时间间隔	-178000000年	178000000年	1微秒

> **注意**
>
> SQL要求只写timestamp等效于timestamp without time zone，并且PostgreSQL鼓励这种行为。timestamptz被接受为timestamp with time zone的一种简写，这是一种PostgreSQL的扩展。

time、timestamp和interval接受一个可选的精度值 p，这个精度值声明在秒域中小数点之后保留的位数。缺省情况下，在精度上没有明确的边界。p允许的范围是从 0 到 6。

interval类型有一个附加选项，它可以通过写下面之一的短语来限制存储的fields的集合：

YEAR
MONTH
DAY

HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND

注意如果fields和p被指定，fields必须包括SECOND，因为精度只应用于秒。

类型time with time zone是 SQL 标准定义的，但是该定义显示出了一些会影响可用性的性质。在大多数情况下， date、time、timestamp without time zone和timestamp with time zone的组合就应该能提供任何应用所需的全范围的日期/时间功能。

类型abstime和reltime是低精度类型，它们被用于系统内部。 我们不鼓励你在应用里面使用这些类型，这些内部类型可能会在未来的版本里消失。

8.5.1. 日期/时间输入

日期和时间的输入可以接受几乎任何合理的格式，包括 ISO 8601、SQL-兼容的、传统POSTGRES的和其他的形式。 对于一些格式，日期输入里的日、月和年的顺序会让人混淆， 并且支持指定所预期的这些域的顺序。把DateStyle参数设置为MDY，就是选择“月－日－年”的解释，设置为DMY就是 “日－月－年”，而YMD是 “年－月－日”。

PostgreSQL在处理日期/时间输入上比SQL标准要求的更灵活。 参阅附录 B获取关于日期/时间输入的准确的分析规则和可识别文本域，包括月份、星期几和时区。

请记住任何日期或者时间的文字输入需要由单引号包围，就象一个文本字符串一样。参考第 4.1.2.7 获取更多信息。SQL要求下面的语法

```
type [ (p) ] 'value'
```

其中p是一个可选的精度声明，它给出了在秒域中的小数位数目。精度可以被指定给time、timestamp和interval类型，并且可以取从0到6的值。这允许前文所述的值。如果在一个常数声明中没有指定任何精度，它将默认取文字值的精度（但不能超过6位）。

8.5.1.1. 日期

表 8.10显示了date类型可能的输入方式。

表 8.10. 日期输入

例子	描述
1999-01-08	ISO 8601; 任何模式下的1月8日 （推荐格式）
January 8, 1999	在任何datestyle输入模式下都无歧义
1/8/1999	MDY模式中的1月8日；DMY模式中的8月1日
1/18/1999	MDY模式中的1月18日；在其他模式中被拒绝
01/02/03	MDY模式中的2003年1月2日； DMY模式中的2003年2月1日； YMD模式中的2001年2月3日
1999-Jan-08	任何模式下的1月8日
Jan-08-1999	任何模式下的1月8日

例子	描述
08-Jan-1999	任何模式下的1月8日
99-Jan-08	YMD模式中的1月8日，否则错误
08-Jan-99	1月8日，除了在YMD模式中错误
Jan-08-99	1月8日，除了在YMD模式中错误
19990108	ISO 8601；任何模式中的1999年1月8日
990108	ISO 8601；任何模式中的1999年1月8日
1999.008	年和一年中的日子
J2451187	儒略日期
January 8, 99 BC	公元前99年

8.5.1.2. 时间

当日时间类型是time [(p)] without time zone和time [(p)] with time zone。 只写time等效于time without time zone。

这些类型的有效输入由当日时间后面跟着可选的时区组成（参阅表 8.11和表 8.12 ）。 如果在time without time zone的输入中指定了时区，那么它会被无声地忽略。你也可以指定一个日期但是它会被忽略，除非你使用了一个涉及到夏令时规则的时区，例如America/New_York。在这种情况下，为了判断是应用了标准时间还是夏令时时间，要求指定该日期。适当的时区偏移被记录在time with time zone值中。

表 8.11. 时间输入

例子	描述
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	和04:05一样，AM并不影响值
04:05 PM	和16:05一样，输入的小时必须为 <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	缩写指定的时区
2003-04-12 04:05:06 America/New_York	全名指定的时区

表 8.12. 时区输入

例子	描述
PST	缩写（太平洋标准时间）
America/New_York	完整时区名
PST8PDT	POSIX风格的时区声明
-8:00	PST的ISO-8601偏移
-800	PST的ISO-8601偏移
-8	PST的ISO-8601偏移

例子	描述
zulu	UTC的军方缩写
z	zulu的短形式

参考第 8.5.3 节以了解如何指定时区。

8.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和时间的串接组成，后面跟着一个可选的时区，一个可选的AD或者BC（另外，AD/BC可以出现在时区前面，但这个顺序并非最佳）。 因此：

1999-01-08 04:05:06

和：

1999-01-08 04:05:06 -8:00

都是有效的值，它遵循ISO 8601 标准。另外，使用广泛的格式：

January 8 04:05:06 1999 PST

也被支持。

SQL标准通过“+”或者“-”符号的存在以及时间后面的时区偏移来区分timestamp without time zone和timestamp with time zone文字。因此，根据标准，

TIMESTAMP '2004-10-19 10:23:54'

是一个timestamp without time zone， 而

TIMESTAMP '2004-10-19 10:23:54+02'

是一个timestamp with time zone。PostgreSQL从来不会在确定文字串的类型之前检查其内容，因此会把上面两个都看做是 timestamp without time zone。因此要保证把上面的文字当作timestamp with time zone看待， 就要给它正确的显式类型：

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

如果一个文字已被确定是timestamp without time zone，PostgreSQL将不声不响忽略任何其中指出的时区。 即，结果值是从输入值的日期/时间域衍生出来的，并且没有就时区进行调整。

对于timestamp with time zone，内部存储的值总是 UTC （全球统一时间，以前也叫格林威治时间GMT）。如果一个输入值有明确的时区声明， 那么它将用该时区合适的偏移量转换成 UTC。如果在输入串里没有时区声明， 那么它就被假设是在系统的TimeZone参数里的那个时区，然后使用这个 timezone时区的偏移转换成 UTC。

如果一个timestamp with time zone值被输出，那么它总是从 UTC 转换成当前的timezone时区，并且显示为该时区的本地时间。要看其它时区的时间，要么修改timezone，要么使用AT TIME ZONE构造（参阅第 9.9.3 节。

在timestamp without time zone和timestamp with time zone之间的转换通常假设timestamp without time zone值应该以timezone本地时间的形式接受或者写出。为该转换指定一个不同的可以用AT TIME ZONE。

8.5.1.4. 特殊值

为了方便，PostgreSQL支持一些特殊日期/时间输入值，如表 8.13所示。这些值中infinity和-infinity被在系统内部以特殊方式表示并且将被原封不动地显示。但是其他的仅仅只是概念上的速写，当被读到的时候会被转换为正常的日期/时间值（特殊地，now及相关串在被读到时立刻被转换到一个指定的时间值）。在作为常量在SQL命令中使用时，所有这些值需要被包括在单引号内。

表 8.13. 特殊日期/时间输入

输入串	合法类型	描述
epoch	date, timestamp	1970-01-01 00:00:00+00（Unix系统时间0）
infinity	date, timestamp	比任何其他时间戳都晚
-infinity	date, timestamp	比任何其他时间戳都早
now	date, time, timestamp	当前事务的开始时间
today	date, timestamp	当日午夜
tomorrow	date, timestamp	明日午夜
yesterday	date, timestamp	昨日午夜
allballs	time	00:00:00.00 UTC

下列SQL-兼容的函数可以被用来为相应的数据类型获得当前时间值：CURRENT_DATE、CURRENT_TIME、 CURRENT_TIMESTAMP、LOCALTIME、 LOCALTIMESTAMP。后四种接受一个可选的亚秒精度声明（参见第 9.9.4 节。注意这些是SQL函数并且在数据输入串中不被识别。

8.5.2. 日期/时间输出

时间/日期类型的输出格式可以设成四种风格之一： ISO 8601、SQL（Ingres）、传统的POSTGRES（Unix的date格式）或 German 。缺省是ISO格式（ISO标准要求使用 ISO 8601 格式。ISO输出格式的名字是历史偶然）。表 8.14显示了每种输出风格的例子。date和time类型的 输出通常只有日期或时间部分和例子中一致。不过，POSTGRES风格输出的是ISO格式的只有日期的值。

表 8.14. 日期/时间输出风格

风格声明	描述	例子
ISO	ISO 8601, SQL标准	1997-12-17 07:37:16-08
SQL	传统风格	12/17/1997 07:37:16.00 PST
Postgres	原始风格	Wed Dec 17 07:37:16 1997 PST
German	地区风格	17.12.1997 07:37:16.00 PST

> **注意**
>
> ISO 8601指定使用大写字母T来分隔日期和时间。PostgreSQL在输入上接受这种格式，但是在输出时它采用一个空格而不是T，如上所示。和一些其他数据库系统一样，这是为了可读性以及与RFC 3339的一致性。

SQL和POSTGRES风格中，如果DMY域顺序被指定，“日”将出现在“月”之前，否则“月”出现在“日”之前（有关该设置如何影响输入值的解释，请参考第 8.5.1 节。表 8.15给出了例子。

表 8.15. 日期顺序习惯

datestyle设置	输入顺序	例子输出
SQL, DMY	日/月/年	17/12/1997 15:37:16.00 CET
SQL, MDY	月/日/年	12/17/1997 07:37:16.00 PST
Postgres, DMY	日/月/年	Wed 17 Dec 07:37:16 1997 PST

日期/时间风格可以由用户使用SET datestyle命令选取，在postgresql.conf配置文件里的参数DateStyle设置或者在服务器或客户端的PGDATESTYLE环境变量里设置。

格式化函数to_char（见第 9.8 节也可以作为一个更灵活的方式来格式化日期/时间输出。

8.5.3. 时区

时区和时区习惯不仅仅受地球几何形状的影响，还受到政治决定的影响。 到了19世纪，全球的时区变得稍微标准化了些，但是还是易于遭受随意的修改，部分是因为夏时制规则。PostgreSQL使用广泛使用的 IANA (Olson) 时区数据库来得到有关历史时区规则的信息。对于未来的时间，我们假设关于一个给定时区的最新已知 规则将会一直持续到无穷远的未来。

PostgreSQL努力在典型使用中与SQL标准的定义相兼容。但SQL标准在日期和时间类型和功能上有一些奇怪的混淆。两个显而易见的问题是：

- 尽管date类型与时区没有联系，而time类型却可以有。 然而，现实世界的时区只有在与时间和日期都关联时才有意义， 因为偏移（时差）可能因为实行类似夏时制这样的制度而在一年里有所变化。
- 缺省的时区会指定一个到UTC的数字常量偏移（时差）。因此，当跨DST边界做日期/时间算术时， 我们根本不可能适应于夏时制时间。

为了克服这些困难，我们建议在使用时区的时候，使用那些同时包含日期和时间的日期/时间类型。我们不建议使用类型 time with time zone （尽管PostgreSQL出于遗留应用以及与SQL标准兼容性的考虑支持这个类型）。 PostgreSQL假设你用于任何类型的本地时区都只包含日期或时间。

在系统内部，所有时区相关的日期和时间都用UTC存储。它们在被显示给客户端之前会被转换成由TimeZone配置参数指定的本地时间。

PostgreSQL允许你使用三种不同形式指定时区：

- 一个完整的时区名字，例如America/New_York。能被识别的时区名字被列在pg_timezone_names视图中（参见第 52.90 节。PostgreSQL用广泛使用的 IANA 时区数据来实现该目的，因此相同的时区名字也可以在其他软件中被识别。
- 一个时区缩写，例如PST。这样一种声明仅仅定义了到UTC的一个特定偏移，而不像完整时区名那样指出整套夏令时转换日期规则。能被识别的缩写被列在pg_timezone_abbrevs视图中（参见第 52.89 节。你不能将配置参数TimeZone或log_timezone设置成一个时区缩写，但是你可以在日期/时间输入值和AT TIME ZONE操作符中使用时区缩写。
- 除了时区名和缩写，PostgreSQL将接受POSIX-风格的 时区声明，形式为STDoffset或STDoffsetDST， 其中STD是一个区域缩写、offset是从UTC西 起的以小时计的数字偏移量、DST是一个可选的夏令时区域缩 写（被假定为给定偏移量提前一小时）。例如，如果EST5EDT还不是一 个被识别的区域名，它可以被接受并且可能和美国东海岸时间的功效相同。在这种语法中， 一个时区缩写可以是一个字母的字符串或者由尖括号（<>）包围 的任意字符串。当一个夏令时区域缩写出现时，会假定根据 IANA 时区数据库的posixrules条目中使用的同一个夏令时转换规则使用它。 在一个标准的PostgreSQL安装中， posixrules和US/Eastern相同， 因此POSIX-风格的时区声明遵循美国的夏令时规则。如果需要，你可以通过替换 posixrules文件来调整这种行为。

简而言之，在缩写和全称之间是有不同的：缩写表示从UTC开始的一个特定偏移量， 而很多全称表示一个本地夏令时规则并且因此具有两种可能的UTC偏移量。例如， 2014-06-04 12:00 America/New_York表示纽约本地时间的中午， 这个特殊的日期是东部夏令时间（UTC-4）。因此2014-06-04 12:00 EDT 指定的是同一个时间点。但是2014-06-04 12:00 EST指定东部标准时间的 中午（UTC-5），不管在那个日期夏令时是否生效。

更要命的是，某些行政区已经使用相同的时区缩写在不同的时间表示不同的 UTC 偏移量。例如， 在莫斯科MSK在某些年份表示 UTC+3 而在另一些年份表示 UTC+4。 PostgreSQL 会根据在指定的日期它们到底表示什么（或者最近表示什么） 来解释这种缩写。但是，正如上面的EST例子所示，这并不是必须和那一天的本地 标准时间相同。

你应该注意到POSIX-风格的时区特性可能导致伪造的输入被接受，因为它没有对区域缩写合理性的检查。例如SET TIMEZONE TO FOOBAR0将会正常工作，让系统实际使用一个相当奇怪的UTC缩写。另一个需要记住的问题是在POSIX时区名中，正值的偏移量被用于格林威治以西的位置。在其他情况下，PostgreSQL将遵循 ISO-8601 惯例，认为正值的时区偏移量是格林威治以东。

在所有情况下，时区名及其缩写都是大小写不敏感的（这是对PostgreSQL 8.2之前版本的一个修改，在这些版本中某些环境下时区名是大小写敏感的而在另外一些环境中却是大小写不敏感的）。

时区名和缩写都不是硬写在服务器中的，它们是从存储在安装目录下的.../share/timezone/和.../share/timezonesets/子目录中获取的（参见第 B.4 节）。

TimeZone配置参数可以在文件postgresql.conf中被设置，或者使用第 19 章描述的任何一种标准方法设置。同时也有一些特殊的方法来设置它：

- SQL命令SET TIME ZONE为会话设置时区。它是SET TIMEZONE TO的另一种拼写，它更加符合SQL的语法。
- libpq客户端使用PGTZ环境变量来通过连接发送一个SET TIME ZONE命令给服务器。

8.5.4. 间隔输入

interval值可以使用下列语法书写：

```
[@] quantity unit [quantity unit...] [direction]
```

其中quantity是一个数字（很可能是有符号的）； unit是毫秒、 millisecond、second、minute、hour、day、 week、month、year、 decade、century、millennium 或者缩写或者这些单位的复数； direction可以是ago或者为空。At符号（@）是一个可选的噪声。不同单位的数量通过合适的符号计数被隐式地添加。ago对所有域求反。如果IntervalStyle被设置为postgres_verbose，该语法也被用于间隔输出。

日、小时、分钟和秒的数量可以不适用显式的单位标记指定。例如，'1 12:59:10'被读作'1 day 12 hours 59 min 10 sec'。同样，一个年和月的组合可以使用一个横线指定，例如'200-10'被读作'200年10个月'（这些较短的形式事实上是SQL标准唯一许可的形式，并且在IntervalStyle被设置为sql_standard时用于输出）。

间隔值也可以被写成 ISO 8601 时间间隔，使用该标准4.4.3.2小节的“带标志符的格式”或者4.4.3.3小节的“替代格式”。带标志符的格式看起来像这样：

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

该串必须以一个P开始，并且可以包括一个引入当日时间单位的T。可用的单位缩写在表 8.16中给出。单位可以被忽略，并且可以以任何顺序指定，但是小于一天的单位必须出现在T之后。特别地，M的含义取决于它出现在T之前还是之后。

表 8.16. ISO 8601 间隔单位缩写

缩写	含义
Y	年
M	月（在日期部分中）
W	周
D	日
H	小时
M	分钟（在时间部分中）
S	秒

如果使用替代格式：

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

串必须以P开始，并且一个T分隔间隔的日期和时间部分。其值按照类似于 ISO 8601日期的数字给出。

在用一个域声明书写一个间隔常量时，或者为一个用域声明定义的间隔列赋予一个串时，对于为标记的量的解释依赖于域。例如INTERVAL '1' YEAR被解读成1年，而INTERVAL '1'表示1秒。同样，域声明允许的最后一个有效域“右边”的域值会被无声地丢弃掉。例如书写INTERVAL '1 day 2:03:04' HOUR TO MINUTE将会导致丢弃秒域，而不是日域。

根据SQL标准，一个间隔值的所有域都必须由相同的符号，这样一个领头的负号将会应用到所有域；例如在间隔文字'-1 2:03:04'中的负号会被应用于日、小时、分钟和秒部分。PostgreSQL允许域具有不同的符号，并且在习惯上认为以文本表示的每个域具有独立的符号，因此在这个例子中小时、分钟和秒部分被认为是正值。如果IntervalStyle被设置为sql_standard，则一个领头的符号将被认为是应用于所有域（但是仅当没有额外符号出现）。否则将使用传统的PostgreSQL解释。为了避免混淆，我们推荐在任何域为负值时为每一个域都附加一个显式的符号。

在冗长的输入格式中，以及在更紧凑输入格式的某些域中，域值可以有分数部分；例如'1.5 week'或'01:02:03.45'。这样的输入被转换为合适的月数、日数和秒数用于存储。当这样会导致月和日中的分数时，分数被加到低序域中，使用的转换因子是1月=30日和1日=24小时。例如，'1.5 month'会变成1月和15日。只有秒总是在输出时被显示为分数。

表 8.1展示了一些有效interval输入的例子。

表 8.17. 间隔输入

例子	描述
1-2	SQL标准格式：1年2个月
3 4:05:06	SQL标准格式：3日4小时5分钟6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	传统Postgres格式：1年2个月3日4小时5分钟6秒钟
P1Y2M3DT4H5M6S	“带标志符的”ISO 8601 格式：含义同上
P0001-02-03T04:05:06	ISO 8601 的“替代格式”：含义同上

在内部，interval值被存储为months、days以及seconds。之所以这样做是因为一个月中的天数是变化的，并且在涉及到夏令时调整时一天可以有23或者25个小时。months以及days域是整数，而seconds域可以存储分数。因为区间通常是从常量字符串或者timestamp减法创建而来，这种存储方法在大部分情况下都很好，但是也可能导致预料之外的结果：

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

函数justify_days和justify_hours可以用来调整溢出其正常范围之外的days和hours。

8.5.5. 间隔输出

间隔类型的输出格式可以被设置为四种风格之一：sql_standard、postgres、postgres_verbose或iso_8601，设置方法使用SET intervalstyle命令。默认值为postgres格式。表 8.18展示了每种输出风格的例子。

如果间隔值符合SQL标准的限制（仅年-月或仅日-时间，没有正负值部分的混合），sql_standard风格为间隔文字串产生符合SQL标准规范的输出。否则输出将看起来像一个标准的年-月文字串跟着一个日-时间文字串，并且带有显式添加的符号以区分混合符号的间隔。

当DateStyle参数被设置为ISO时，postgres风格的输出匹配PostgreSQL 8.4版本以前的输出。

当DateStyle参数被设置为非ISO输出时，postgres_verbose风格的输出匹配PostgreSQL 8.4版本以前的输出。

iso_8601风格的输出匹配在ISO 8601标准的4.4.3.2小节中描述的“带标志符的格式”。

表 8.18. 间隔输出风格例子

风格声明	年-月间隔	日-时间间隔	混合间隔
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. 布尔类型

PostgreSQL提供标准的SQL类型boolean，参见表 8.19 boolean可以有多个状态：“true（真）”、“false（假）”和第三种状态“unknown（未知）”，未知状态由SQL空值表示。

表 8.19. 布尔数据类型

名字	存储字节	描述
boolean	1字节	状态为真或假

“真”状态的有效文字值是：

```
TRUE
```

't'
'true'
'y'
'yes'
'on'
'1'

而对于“假”状态，你可以使用下面这些值：

FALSE
'f'
'false'
'n'
'no'
'off'
'0'

前导或者末尾的空白将被忽略，并且大小写也无关紧要。使用TRUE和FALSE这样的关键词比较好（SQL兼容）。

例 8.2显示了使用字母t和f输出boolean值的例子。

例 8.2. 使用boolean类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

8.7. 枚举类型

枚举（enum）类型是由一个静态、值的有序集合构成的数据类型。它们等效于很多编程语言所支持的enum类型。枚举类型的一个例子可以是一周中的日期，或者一个数据的状态值集合。

8.7.1. 枚举类型的声明

枚举类型可以使用CREATE TYPE命令创建，例如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

一旦被创建，枚举类型可以像很多其他类型一样在表和函数定义中使用：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
```

```
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. 排序

一个枚举类型的值的排序是该类型被创建时所列出的值的顺序。枚举类型的所有标准的比较操作符以及相关聚集函数都被支持。例如：

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. 类型安全性

每一种枚举数据类型都是独立的并且不能和其他枚举类型相比较。看这样一个例子：

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果你确实需要做这样的事情，你可以写一个自定义的操作符或者在查询中加上显式造型：

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. 实现细节

枚举标签是大小写敏感的，因此'happy'与'HAPPY'是不同的。标签中的空格也是有意义的。

尽管枚举类型的主要目的是用于值的静态集合，但也有方法在现有枚举类型中增加新值和重命名值（见ALTER TYPE）。不能从枚举类型中去除现有的值，也不能更改这些值的排序顺序，如果要那样做可以删除并且重建枚举类型。

一个枚举值在磁盘上占据4个字节。一个枚举值的文本标签的长度受限于NAMEDATALEN设置，该设置被编译在PostgreSQL中，在标准编译下它表示最多63字节。

从内部枚举值到文本标签的翻译被保存在系统目录pg_enum中。可以直接查询该目录。

8.8. 几何类型

几何数据类型表示二维的空间物体。表 8.20展示了PostgreSQL中可以用的几何类型。

表 8.20. 几何类型

名字	存储尺寸	表示	描述
point	16字节	平面上的点	(x,y)
line	32字节	无限长的线	{A,B,C}
lseg	32字节	有限线段	((x1,y1),(x2,y2))
box	32字节	矩形框	((x1,y1),(x2,y2))
path	16+16n字节	封闭路径（类似于多边形）	((x1,y1),...)
path	16+16n字节	开放路径	[(x1,y1),...]
polygon	40+16n字节	多边形（类似于封闭路径）	((x1,y1),...)
circle	24字节	圆	<(x,y),r>（中心点和半径）

我们有一系列丰富的函数和操作符可用来进行各种几何操作， 如缩放、平移、旋转和计算相交等 它们在第 9.11 节解释。

8.8.1. 点

点是几何类型的基本二维构造块。用下面的语法描述point类型的值：

```
( x , y )
  x , y
```

其中x和y分别是坐标，都是浮点数。

点使用第一种语法输出。

8.8.2. 线

线由线性方程Ax + By + C = 0 表示，其中A和B都不为零。类型line 的值采用以下形式输入和输出：

```
{ A, B, C }
```

另外，还可以用下列任一形式输入：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中 (x1,y1) 和 (x2,y2) 是线上不同的两点。

8.8.3. 线段

线段用一对线段的端点来表示。lseg类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中(x1,y1) 和 (x2,y2) 是线段的端点。

线段使用第一种语法输出。

8.8.4. 方框

方框用其对角的点对表示。box类型的值使用下面的语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中(x1,y1) 和 (x2,y2) 是方框的对角点。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被按顺序记录为右上角和左下角。

8.8.5. 路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有被连接起来；也可能是封闭的，这时认为第一个和最后一个点被连接起来。

path类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成路径的线段的端点。方括弧（[]）表示一个开放的路径，圆括弧（()）表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被忽略时，路径将被假定为封闭。

路径的输出使用第一种或第二种语法。

8.8.6. 多边形

多边形由一系列点代表（多边形的顶点）。多边形和封闭路径很像，但是存储方式不一样而且有自己的一套支持例程。

polygon类型的值用下列语法声明：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

8.8.7. 圆

圆由一个圆心和一个半径代表。circle类型的值用下面的语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

其中(x,y)是圆心，而r是圆的半径。

圆的输出用第一种语法。

8.9. 网络地址类型

PostgreSQL提供用于存储 IPv4、IPv6 和 MAC 地址的数据类型，如表 8.2所示。 用这些数据类型存储网络地址比用纯文本类型好，因为这些类型提供输入错误检查以及特殊的操作符和函数（见第 9.12 节

表 8.21. 网络地址类型

名字	存储尺寸	描述
cidr	7或19字节	IPv4和IPv6网络
inet	7或19字节	IPv4和IPv6主机以及网络
macaddr	6字节	MAC地址
macaddr8	8 bytes	MAC地址（EUI-64格式）

在对inet或者cidr数据类型进行排序的时候， IPv4 地址将总是排在 IPv6 地址前面，包括那些封装或者是映射在 IPv6 地址里 的 IPv4 地址，例如 ::10.2.3.4 或者 ::ffff::10.4.3.2。

8.9.1. inet

inet在一个数据域里保存一个 IPv4 或 IPv6 主机地址，以及一个可选的它的子网。 子网由主机地址中表示的网络地址位数表示（“网络掩码”）。 如果网络掩码为 32 并且地址是 IPv4 ，那么该值不表示任何子网，只是一台主机。在 IPv6 中地址长度是 128 位，因此 128 位指定一个唯一的主机地址。 请注意如果你想只接受网络地址，你应该使用cidr类型而不是inet。

该类型的输入格式是地址/y，其中地址是一个 IPv4 或者 IPv6 地址，y是网络掩码的位数。如果/y部分缺失， 则网络掩码对 IPv4 而言是 32，对 IPv6 而言是 128，所以该值表示只有一台主机。在显示时，如果/y部分指定一个单台主机，它将不会被显示出来。

8.9.2. cidr

cidr类型保存一个 IPv4 或 IPv6 网络地址声明。其输入和输出遵循无类的互联网域路由（Classless Internet Domain Routing）习惯。声明一个网络的格式是地址/y，其中address是 IPv4 或 IPv6 网络地址而y是网络掩码的位数。如果省略y， 那么掩码部分用旧的有类的网络编号系统进行计算，否则它将至少大到足以包括写在输入中的所有字节。声明一个在其指定的掩码右边置了位的网络地址会导致错误。

表 8.22展示了一些例子。

表 8.22.cidr类型输入例子

cidr输入	cidr输出	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet vs. cidr

inet和cidr类型之间的本质区别是inet接受右边有非零位的网络掩码， 而cidr不接受。例如，192.168.0.1/24对inet是有效的，但对cidr是无效的。

> **提示**
>
> 如果你不喜欢inet或cidr值的输出格式，可以尝试函数host、text和abbrev。

8.9.4. macaddr

macaddr类型存储 MAC 地址，也就是以太网卡硬件地址 （尽管 MAC 地址还用于其它用途）。可以接受下列格式的输入：

'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'

这些例子指定的都是同一个地址。对于位a到f，大小写都可以接受。输出总是使用展示的第一种形式。

IEEE Std 802-2001 指定第二种展示的形式（带有连字符）作为MAC地址的标准形式，并且指定第一种形式（带有分号）作为位翻转的记号，因此 08-00-2b-01-02-03 = 01:00:4D:08:04:0C。这种习惯目前已经被广泛地忽略，并且它只与废弃的网络协议（如令牌环）相关。PostgreSQL 没有对位翻转做任何规定，并且所有可接受的格式都使用标准的LSB顺序。

剩下的五种输入格式不属于任何标准。

8.9.5. macaddr8

macaddr8类型以EUI-64格式存储MAC地址，例如以太网卡的硬件地址（尽管MAC地址也被用于其他目的）。这种类型可以接受6字节和8字节长度的MAC地址，并且将它们存储为8字节长度的格式。以6字节格式给出的MAC地址被存储为8字节长度格式的方式是吧第4和第5字节分别设置为FF和FE。 注意IPv6使用一种修改过的EUI-64格式，其中从EUI-48转换过来后的第7位应该被设置为一。函数macaddr8_set7bit被用来做这种修改。 一般而言，任何由16进制数（字节边界上）对构成的输入（可以由':'、'-'或者'.'统一地分隔）都会被接受。16进制数的数量必须是16（8字节）或者12（6字节）。前导和拖尾的空格会被忽略。 下面是可以被接受的输入格式的例子：

'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'

这些例子都指定相同的地址。数字a到f的大小写形式都被接受。输出总是以上面显示的第一种形式。 上述的后六种输入格式不属于任何标准。 要把EUI-48格式的传统48位MAC地址转换成修改版EUI-64格式（包括在IPv6地址中作为主机部分），可以使用下面的macaddr8_set7bit：

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. 位串类型

位串就是一串 1 和 0 的串。它们可以用于存储和可视化位掩码。我们有两种类型的 SQL 位类型：bit(n)和bit varying(n)，其中 n是一个正整数。

bit类型的数据必须准确匹配长度n;　　试图存储短些或者长一些的位串都是错误的。bit varying数据是最长n的变长类型，更长的串会被拒绝。写一个没有长度的bit等效于bit(1)，没有长度的bit varying意味着没有长度限制。

> **注意**
>
> 如果我们显式地把一个位串值转换成bit(n)，　那么它的右边将被截断或者在右边补齐零，直到刚好n位，　而且不会抛出任何错误。类似地，如果我们显式地把一个位串数值转换成bit varying(n)，如果它超过了n位，　那么它的右边将被截断。

请参考第　4.1.2.5　节获取有关位串常量的语法的信息。还有一些位逻辑操作符和串操作函数可用，请见第　9.6　节。

例　8.3.　使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

一个位串值对于每8位的组需要一个字节，外加总共5个或8个字节，这取决于串的长度（但是长值可能被压缩或者移到线外，如第　8.3　节对字符串的解释一样）。

8.11.　文本搜索类型

PostgreSQL提供两种数据类型，它们被设计用来支持全文搜索，全文搜索是一种在自然语言的文档集合中搜索以定位那些最匹配一个查询的文档的活动。tsvector类型表示一个为文本搜索优化的形式下的文档，tsquery类型表示一个文本查询。第　12　章提供了对于这种功能的详细解释，并且第　9.13　节总结了相关的函数和操作符。

8.11.1.　tsvector

一个tsvector值是一个排序的可区分词位的列表，词位是被正规化合并了同一个词的不同变种的词（详见第　12　章。排序和去重是在输入期间自动完成的，如下例所示：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

要表示包含空白或标点的词位，将它们用引号包围：

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

（我们在这个例子中使用美元符号包围的串文字并且下一个用来避免在文字中包含双引号记号产生的混淆）。嵌入的引号和反斜线必须被双写：

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

可选的，整数位置可以被附加给词位：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

一个位置通常表示源词在文档中的定位。位置信息可以被用于邻近排名。位置值可以从 1 到 16383，更大的数字会被 16383。对于相同的词位出现的重复位置将被丢弃。

具有位置的词位可以进一步地被标注一个权重，它可以是A、 B、C或D。 D是默认值并且因此在输出中不会显示：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

权重通常被用来反映文档结构，例如将主题词标记成与正文词不同。文本搜索排名函数可以为不同的权重标记器分配不同的优先级。

了解tsvector类型本身并不执行任何词正规化这一点很重要，它假定给它的词已经被恰当地为应用正规化过。例如，

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

对于大部分英语文本搜索应用，上面的词将会被认为是非正规化的，但是tsvector并不在乎这一点。原始文档文本通常应该经过to_tsvector以恰当地为搜索正规化其中的词：

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

再次地，详情请参阅第 12 章

8.11.2. tsquery

一个tsquery值存储要用于搜索的词位，并且使用布尔操作符&（AND）、|（OR）和!（NOT）来组合它们，还有短语搜索操作符<->（FOLLOWED BY）。也有一种 FOLLOWED BY 操作符的变体<N>，其中N是一个整数常量，它指定要搜索的两个词位之间的距离。<->等效于<1>。

圆括号可以被用来强制对操作符分组。如果没有圆括号，!（NOT）的优先级最高，其次是<->（FOLLOWED BY），然后是&（AND），最后是|（OR）。

这里有一些例子：

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

可选地，一个tsquery中的词位可以被标注一个或多个权重字母，这将限制它们只能和具有那些权重之一的tsvector词位相匹配：

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

此外，一个tsquery中的词位可以被标注为*来指定前缀匹配：

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

这个查询将匹配一个tsvector中以“super”开头的任意词。

词位的引号规则和之前描述的tsvector中的词位相同；并且，正如tsvector，任何请求的词正规化必须在转换到tsquery类型之前完成。to_tsquery函数可以方便地执行这种正规化：

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

注意to_tsquery将会以和其他词同样的方式处理前缀，这也意味着下面的比较会返回真：

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
```

```
----------
 t
```

因为postgres会被处理成postgr：

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

这会匹配postgraduate被处理后的形式。

8.12. UUID类型

数据类型uuid存储由RFC 4122、ISO/IEC 9834-8:2005以及相关标准定义的通用唯一标识符（UUID）（某些系统将这种数据类型引用为全局唯一标识符GUID）。这种标识符是一个128位的量，它由一个精心选择的算法产生，该算法能保证在已知空间中任何其他使用相同算法的人能够产生同一个标识符的可能性非常非常小。因此，对于分布式系统，这些标识符相比序列生成器而言提供了一种很好的唯一性保障，序列生成器只能在一个数据库中保证唯一。

一个UUID被写成一个小写十六进制位的序列，该序列被连字符分隔成多个组：首先是一个8位组，接下来是三个4位组，最后是一个12位组。总共的32位（十六进制位）表示了128个二进制位。一个标准形式的UUID类似于：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL也接受另一种输入形式： 使用大写位、标准格式被花括号包围、忽略某些或者全部连字符、在任意4位组后面增加一个连字符。例如：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

输出总是采用标准形式。

PostgreSQL为UUID提供了存储和比较函数，但是核心数据库不包含任何用于产生UUID的函数，因为没有一个单一算法能够很好地适应每一个应用。uuid-ossp模块提供了实现一些标准算法的函数。 pgcrypto模块也为随机 UUID 提供了一个生成函数。 此外，UUID可以由客户端应用产生，或者由通过服务器端函数调用的其他库生成。

8.13. XML类型

xml数据类型可以被用来存储XML数据。它比直接在一个text域中存储XML数据的优势在于，它会检查输入值的结构是不是良好，并且有支持函数用于在其上执行类型安全的操作，参见第 9.14 节使用这种数据类型要求在安装时用configure --with-libxml选项编译。

xml类型可以存储结构良好（如XML标准所定义）的“文档”，以及“内容”片段，它们由XML标准中的XMLDecl? content产品所定义。粗略地看，这意味着内容片段中可以有多于一个的顶层元素或字符节点。表达式xmlvalue IS DOCUMENT可以被用来评估一个特定的xml值是一个完整文档或者仅仅是一个文档片段。

8.13.1. 创建XML值

要从字符数据中生成一个xml类型的值，可以使用函数xmlparse:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

例子：

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</
title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

然而根据SQL标准这是唯一将字符串转换为XML值的方法，PostgreSQL特有的语法：

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

也可以被使用。

即便输入值指定了一个文档类型声明（DTD），xml类型也不根据DTD来验证输入值。目前也没有内建的支持用于根据其他XML模式语言（如XML模式）来进行验证。

作为一个逆操作，从xml产生一个字符串可以使用函数xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

type可以是 character、character varying或 text（或者其中之一的一个别名）。再次地，根据SQL标准，这也是在xml类型和字符类型间做转换的唯一方法，但是PostgreSQL也允许你简单地造型这些值。

当一个字符串不是使用XMLPARSE造型成xml或者不是使用XMLSERIALIZE从xml造型得到，对于DOCUMENT和CONTENT两者的选择是根据“XML option” 会话配置参数决定的，它可以使用标准命令来设置：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

或者是更具有PostgreSQL风格的语法

```
SET xmloption TO { DOCUMENT | CONTENT };
```

默认值是CONTENT，因此所有形式的XML数据都被允许。

> **注意**
>
> 在使用默认XML option设置时，如果字符串包含一个文档类型声明，你就不能直接将字符串造型成类型xml，因为XML内容片断的定义不接受它们。如果你需要这样做，要么使用XMLPARSE，要么修改XML option。

8.13.2. 编码处理

在客户端、服务器以及其中流过的XML数据上处理多字符编码时必须要注意。在使用文本模式向服务器传递查询以及向客户端传递查询结果（在普通模式）时，PostgreSQL将所有在客户端和服务器之间传递的字符数据转换为目标端的字符编码，参见第 23.3 节这也包括了表示XML值的串，正如上面的例子所述。这也通常意味着由于字符数据会在客户端和服务器之间传递时被转换成其他编码，包含在XML数据中的编码声明可能是无效的，因为内嵌的编码声明没有被改变。为了处理这种行为，包含在表示xml类型输入的字符串中包含的编码声明会被忽略，并且其内容被假定为当前服务器的编码。接着，为了正确处理，XML数据的字

符串必须以当前客户端编码从客户端发出。客户端负责在把文档发送给服务器之前将它们转换为当前客户端编码，或者适当地调整客户端编码。在输出时，xml类型的值将不会有一个编码声明，并且客户端将会假设所有数据都是当前客户端编码。

在使用二进制模式传送查询参数给服务器以及传回查询结果给客户端时，不会执行编码转换，因此情况就有所不同。在这种情况下，XML数据中的编码声明将被注意到，并且如果缺少编码声明时该数据会被假定为UTF-8（由于XML标准的要求，注意PostgreSQL不支持UTF-16）。在输出时，数据将会有一个编码声明来指定客户端编码，除非客户端编码为UTF-8（这种情况下编码声明会被忽略）。

不用说，在PostgreSQL中处理XML数据产生错误的可能性更小，并且在XML数据编码、客户端编码和服务器编码三者相同时效率更高。因为XML数据在内部是以UTF-8处理的，如果服务器编码也是UTF-8时，计算效率将会最高。

> **小心**
>
> 当服务器编码不是UTF-8时，某些XML相关的函数可能在非ASCII数据上完全无法工作。尤其在xmltable()和xpath()上，这是一个已知的问题。

8.13.3. 访问XML值

xml数据类型有些不同寻常，因为它不提供任何比较操作符。这是因为对于XML数据不存在良定义的和通用的比较算法。这种状况造成的后果就是，你无法通过比较一个xml和一个搜索值来检索行。XML值因此通常应该伴随着一个独立键值域，如一个ID。另一种比较XML值的方案是将它们先转换为字符串，但注意字符串比较对于XML比较方法没有什么帮助。

由于没有可以用于xml数据类型的比较操作符，因此无法直接在这种类型上创建索引。如果需要在XML中快速的搜索，可能的解决方案包括将表达式造型为一个字符串类型然后索引之，或者在一个XPath表达式上索引。当然，实际的查询必须被调整为使用被索引的表达式。

PostgreSQL中的文本搜索功能也可以被用来加速XML数据的全文搜索。但是，所需的预处理支持目前在PostgreSQL发布中还不可用。

8.14. JSON 类型

根据RFC 7159[1] 中的说明，JSON 数据类型是用来存储 JSON（JavaScript Object Notation）数据的。这种数据也可以被存储为text，但是 JSON 数据类型的 优势在于能强制要求每个被存储的值符合 JSON 规则。也有很多 JSON 相关的函 数和操作符可以用于存储在这些数据类型中的数据，见 第 9.15 节

有两种 JSON 数据类型：json 和 jsonb。它们 几乎接受完全相同的值集合作为输入。主要的实际区别之一是 效率。json数据类型存储输入文本的精准拷贝，处理函数必须在每 次执行时必须重新解析该数据。而jsonb数据被存储在一种分解好的 二进制格式中，它在输入时要稍慢一些，因为需要做附加的转换。但是 jsonb在处理时要快很多，因为不需要解析。jsonb也支 持索引，这也是一个令人瞩目的优势。

由于json类型存储的是输入文本的准确拷贝，其中可能会保留在语法 上不明显的、存在于记号之间的空格，还有 JSON 对象内部的键的顺序。还有， 如果一个值中的 JSON 对象包含同一个键超过一次，所有的键/值对都会被保留（ 处理函数会把最后的值当作有效值）。相反，jsonb不保留空格、不 保留对象键的顺序并且不保留重复的对象键。如果在输入中指定了重复的键，只有 最后一个值会被保留。

通常，除非有特别特殊的需要（例如遗留的对象键顺序假设），大多数应用应该 更愿意把 JSON 数据存储为jsonb。

[1] https://tools.ietf.org/html/rfc7159

PostgreSQL对每个数据库只允许一种 字符集编码。因此 JSON 类型不可能严格遵守 JSON 规范，除非数据库编码 是 UTF8。尝试直接包括数据库编码中无法表示的字符将会失败。反过来，能 在数据库编码中表示但是不在 UTF8 中的字符是被允许的。

RFC 7159 允许 JSON 字符串包含\uXXXX 所标记的 Unicode 转义序列。在json类型的输入函数中，不管数据库 编码如何都允许 Unicode 转义，并且只检查语法正确性（即，跟在\u 后面的四个十六进制位）。但是，jsonb的输入函数更加严格：它不允 许非 ASCII 字符的 Unicode 转义（高于U+007F的那些），除非数据 库编码是 UTF8。jsonb类型也拒绝\u0000（因为 PostgreSQL的text类型无法表示 它），并且它坚持使用 Unicode 代理对来标记位于 Unicode 基本多语言平面之外 的字符是正确的。合法的 Unicode 转义会被转换成等价的 ASCII 或 UTF8 字符进 行存储，这包括把代理对折叠成一个单一字符。

> **注意**
>
> 很多第 9.15 中描述的 JSON 处理函数将把 Unicode 转义转换成常规字符，并且将因此抛出和刚才所描述的同样类型的错误（即使它们 的输入是类型json而不是jsonb）。json的 输入函数不做这些检查是由来已久的，不过它确实允许将 JSON Unicode 转义简单 的（不经处理）存储在一个非 UTF8 数据库编码中。通常，最好尽可能避免在一个非 UTF8 数据库编码的 JSON 中混入 Unicode 转义。

在把文本 JSON 输入转换成jsonb时，RFC 7159描述 的基本类型会被有效地映射到原生的PostgreSQL类型（如 表 8.23中所示）。因此，在合法 jsonb数据的组成上有一些次要额外约束，它们不适合 json类型和抽象意义上的 JSON，这些约束对应于有关哪些东西不 能被底层数据类型表示的限制。尤其是，jsonb将拒绝位于 PostgreSQL numeric数据类型范 围之外的数字，而json则不会。这类实现定义的限制是 RFC 7159 所允许的。不过，实际上这类问题更可能发生在其他实 现中，因为把 JSON 的number基本类型表示为 IEEE 754 双精度浮点是很常见的（这也是RFC 7159 明确期待和允许的）。当在这类系 统间使用 JSON 作为一种交换格式时，应该考虑丢失数字精度的风险。

相反地，如表中所述，有一些 JSON 基本类型输入格式上的次要限制并不适用于相 应的PostgreSQL类型。

表 8.23. JSON 基本类型和相应的PostgreSQL类型

JSON 基本类型	PostgreSQL类型	注释
string	text	不允许\u0000，如果数据库编码不是 UTF8，非 ASCII Unicode 转义也是这样
number	numeric	不允许NaN 和 infinity值
boolean	boolean	只接受小写true和false拼写
null	(无)	SQL NULL是一个不同的概念

8.14.1. JSON 输入和输出语法

RFC 7159 中定义了 JSON 数据类型的输入/输出语法。

下列都是合法的json（或者jsonb）表达式：

```
-- 简单标量/基本值
-- 基本值可以是数字、带引号的字符串、true、false或者null
SELECT '5'::json;

-- 有零个或者更多元素的数组（元素不需要为同一类型）
```

```
SELECT '[1, 2, "foo", null]'::json;

-- 包含键值对的对象
-- 注意对象键必须总是带引号的字符串
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- 数组和对象可以被任意嵌套
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

如前所述，当一个 JSON 值被输入并且接着不做任何附加处理就输出时， json会输出和输入完全相同的文本，而jsonb 则不会保留语义上没有意义的细节（例如空格）。例如，注意下面的不同：

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

值得一提的一种语义上无意义的细节是，在jsonb中数据会被按照底层 numeric类型的行为来打印。实际上，这意味着用E记号 输入的数字被打印出来时就不会有该记号，例如：

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

不过，如这个例子所示，jsonb将会保留拖尾的小数点后的零，即便这 对于等值检查等目的来说是语义上无意义的。

8.14.2. 有效地设计 JSON 文档

将数据表示为 JSON 比传统关系数据模型要灵活得多，在需求不固定时 这种优势更加令人感兴趣。在同一个应用里非常有可能有两种方法共存 并且互补。不过，即便是在要求最大灵活性的应用中，我们还是推荐 JSON 文档有固定的结构。该结构通常是非强制的（尽管可能会强制一 些业务规则），但是有一个可预测的结构会使书写概括一个表中的 “文档”（数据）集合的查询更容易。

当被存储在表中时，JSON 数据也像其他数据类型一样服从相同的并发 控制考虑。尽管存储大型文档是可行的，但是要记住任何更新都在整行 上要求一个行级锁。为了在更新事务之间减少锁争夺，可考虑把 JSON 文档限制到一个可管理的尺寸。理想情况下，JSON 文档应该每个表示 一个原子数据，业务规则命令不会进一步把它们划分成更小的可独立修 改的数据。

8.14.3. jsonb 包含和存在

测试包含是jsonb的一种重要能力。对 json类型没有平行的功能集。包含测试会测试一个jsonb文档是否被包含在另一个文档中。除了特别注解 之外，这些例子都会返回真：

```
-- 简单的标量/基本值只包含相同的值：
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- 右边的数字被包含在左边的数组中：
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- 数组元素的顺序没有意义，因此这个例子也返回真：
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- 重复的数组元素也没有关系：
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- 右边具有一个单一键值对的对象被包含在左边的对象中：
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @>
 '{"version": 9.4}'::jsonb;

-- 右边的数组不会被认为包含在左边的数组中，
-- 即使其中嵌入了一个相似的数组：
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- 得到假

-- 但是如果同样也有嵌套，包含就成立：
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- 类似的，这个例子也不会被认为是包含：
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- 得到假

-- 包含一个顶层键和一个空对象：
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

一般原则是被包含的对象必须在结构和数据内容上匹配包含对象，这种匹配　可以是从包含对象中丢弃了不匹配的数组元素或者对象键值对之后成立。但　是记住做包含匹配时数组元素的顺序是没有意义的，并且重复的数组元素实 际也只会考虑一次。

结构必须匹配的一般原则有一种特殊情况，一个数组可以包含一个基本值：

```
-- 这个数组包含基本字符串值：
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- 反之不然，下面的例子会报告“不包含”：
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- 得到假
```

jsonb还有一个存在操作符，它是包含的一种　变体：它测试一个字符串（以一个text值的形式给出）是否出 现在jsonb值顶层的一个对象键或者数组元素中。除非特别注解，　下面这些例子返回真：

```
-- 字符串作为一个数组元素存在：
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- 字符串作为一个对象键存在：
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- 不考虑对象值：
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- 得到假

-- 和包含一样，存在必须在顶层匹配：
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- 得到假
```

```
-- 如果一个字符串匹配一个基本 JSON 字符串，它就被认为存在：
SELECT '"foo"'::jsonb ? 'foo';
```

当涉及很多键或元素时，JSON 对象比数组更适合于做包含或存在测试， 因为它们不像数组，进行搜索时会进行内部优化，并且不需要被线性搜索。

> **提示**
>
> 由于 JSON 的包含是嵌套的，因此一个恰当的查询可以跳过对子对象的显式选择。 例如，假设我们在顶层有一个doc列包含着对象，大部分对象包含着tags域，其中有子对象的数组。这个查询会找到其中出现了 同时包含"term":"paris"和"term":"food"的子对象 的项，而忽略任何位于tags数组之外的这类键：
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
> ```
>
> 可以用下面的查询完成同样的事情：
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
> ```
>
> 但是后一种方法灵活性较差，并且常常也效率更低。
>
> 在另一方面，JSON 的存在操作符不是嵌套的：它将只在 JSON 值的顶层 查找指定的键或数组元素。

第 9.15 节记录了多个包含和存在操作符，以及 所有其他 JSON 操作符和函数。

8.14.4. jsonb 索引

GIN 索引可以被用来有效地搜索在大量jsonb文档（数据）中出现 的键或者键值对。提供了两种 GIN “操作符类”，它们在性能和灵活 性方面做出了不同的平衡。

jsonb的默认 GIN 操作符类支持使用@>、 ?、?&以及?|操作符的查询（这些 操作符实现的详细语义请见表 9.44 。 使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxgin ON api USING gin (jdoc);
```

非默认的 GIN 操作符类jsonb_path_ops只支持索引 @>操作符。使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxginp ON api USING gin (jdoc jsonb_path_ops);
```

考虑这样一个例子：一个表存储了从一个第三方 Web 服务检索到的 JSON 文档，并且有一个模式定义。一个典型的文档：

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
```

```
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

我们把这些文档存储在一个名为api的表的名为 jdoc的jsonb列中。如果在这个列上创建一个 GIN 索引，下面这样的查询就能利用该索引：

```
-- 寻找键 "company" 有值 "Magnafone" 的文档
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company":
 "Magnafone"}';
```

不过，该索引不能被用于下面这样的查询，因为尽管操作符? 是可索引的，但它不能直接被应用于被索引列jdoc：

```
-- 寻找这样的文档：其中的键 "tags" 包含键或数组元素 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

但是，通过适当地使用表达式索引，上述查询也能使用一个索引。 如果对"tags"键中的特定项的查询很常见，可能值得 定义一个这样的索引：

```
CREATE INDEX idxgintags ON api USING gin ((jdoc -> 'tags'));
```

现在，WHERE 子句 jdoc -> 'tags' ? 'qui' 将被识别为可索引操作符?在索引表达式jdoc -> 'tags' 上的应用（更多有关表达式索引的信息可见第 11.7 节。

另一种查询的方法是利用包含，例如：

```
-- 寻找这样的文档：其中键 "tags" 包含数组元素 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

jdoc列上的一个简单 GIN 索引就能支持这个查询。 但是注意这样一个索引将会存储jdoc列中每一个键 和值的拷贝，然而前一个例子的表达式索引只存储tags 键下找到的数据。虽然简单索引的方法更加灵活（因为它支持有关任 意键的查询），定向的表达式索引更小并且搜索速度比简单索引更快。

尽管jsonb_path_ops操作符类只支持用 @>操作符的查询，但它比起默认的操作符类jsonb_ops有更客观的性能优势。一个 jsonb_path_ops索引通常也比一个相同数据上的jsonb_ops要小得多，并且搜索的专一性更好，特 别是当查询包含频繁出现在该数据中的键时。因此，其上的搜索操作 通常比使用默认操作符类的搜索表现更好。

jsonb_ops和jsonb_path_ops GIN 索引之间的技术区别是前者为数据中的每一个键和值创建独立的索引项， 而后者值为该数据中的每个值创建索引项。[2] 基本上，每一个jsonb_path_ops索引项是其所对应的值和 键的哈希。例如要索引{"foo": {"bar": "baz"}}，将创建一个 单一的索引项，它把所有三个foo、bar、 和baz合并到哈希值中。因此一个查找这个结构的包含查询可能 导致极度详细的索引搜索。但是根本没有办法找到foo是否作为 一个键出现。在另一方面，一个jsonb_ops会创建三个索引 项分别表示foo、bar和baz。那么要 做同样的包含查询，它将会查找包含所有三个项的行。虽然 GIN

[2] 对于这种目的，术语“值”包括数组元素，尽管 JSON 的术语有时 认为数组元素与对象内的值不同。

索引能够相当 有效地执行这种 AND 搜索，它仍然不如等效的 jsonb_path_ops搜索那样详细和快速（特别是如果有大量 行包含三个索引项中的任意一个时）。

jsonb_path_ops方法的一个不足是它不会为不包含任何值 的 JSON 结构创建索引项，例如{"a": {}}。如果需要搜索包 含这样一种结构的文档，它将要求一次全索引扫描，那就非常慢。 因此jsonb_path_ops不适合经常执行这类搜索的应用。

jsonb也支持btree和hash索引。 这通常值用于检查完整 JSON 文档等值非常重要的场合。jsonb 数据的btree顺序很少有人关系，但是为了完整性其顺序是：

对象 > 数组 > 布尔 > 数字 > 字符串 > 空值

带有 n 对的对象 > 带有 n - 1 对的对象

带有 n 个元素的数组 > 带有 n - 1 个元素的数组

具有相同数量对的对象这样比较：

key-1, value-1, key-2 ...

注意对象键被按照它们的存储顺序进行比较，特别是由于较短的键被存储在 较长的键之前，这可能导致结果不直观，例如：

{ "aa": 1, "c": 1} > {"b": 1, "d": 1}

相似地，具有相同元素数量的数组按照以下顺序比较：

element-1, element-2 ...

基本 JSON 值的比较会使用低层PostgreSQL 数据类型相同的比较规则进行。字符串的比较会使用默认的数据库排序规则。

8.14.5. 转换

有一些附加的扩展可以为不同的过程语言实现jsonb类型的转换。

PL/Perl的扩展被称作jsonb_plperl和jsonb_plperlu。如果使用它们，jsonb值会视情况被映射为Perl的数组、哈希和标量。

PL/Python的扩展被称作jsonb_plpythonu、jsonb_plpython2u和jsonb_plpython3u（PL/Python命名习惯请见第 46.1 节。如果使用它们，jsonb值会视情况被映射为Python的词典、列表和标量。

8.15. 数组

PostgreSQL允许一个表中的列定义为变长多维数组。可以创建任何内建或用户定义的基类、枚举类型、组合类型或者域的数组。

8.15.1. 数组类型的定义

为了展示数组类型的使用，我们创建这样一个表：

```
CREATE TABLE sal_emp (
    name            text,
```

```
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

如上所示，一个数组数据类型可以通过在数组元素的数据类型名称后面加上方括号（[]）来命名。上述命令将创建一个名为sal_emp的表，它有一个类型为text的列（name），一个表示雇员的季度工资的一维integer类型数组（pay_by_quarter），以及一个表示雇员每周日程表的二维text类型数组（schedule）。

CREATE TABLE的语法允许指定数组的确切大小，例如：

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

然而，当前的实现忽略任何提供的数组尺寸限制，即其行为与未指定长度的数组相同。

当前的实现也不会强制所声明的维度数。一个特定元素类型的数组全部被当作是相同的类型，而不论其尺寸或维度数。因此，在CREATE TABLE中声明数组的尺寸或维度数仅仅只是文档而已，它并不影响运行时的行为。

另一种符合SQL标准的语法是使用关键词ARRAY，可以用来定义一维数组。pay_by_quarter可以这样定义：

```
    pay_by_quarter  integer ARRAY[4],
```

或者，不指定数组尺寸：

```
    pay_by_quarter  integer ARRAY,
```

但是和前面一样，PostgreSQL在任何情况下都不会强制尺寸限制。

8.15.2. 数组值输入

要把一个数组值写成一个文字常数，将元素值用花括号包围并用逗号分隔（如果你懂C，这和初始化结构的C语法没什么两样）。在任意元素值周围可以使用双引号，并且在元素值包含逗号或花括号时必须这样做（更多细节如下所示）。因此，一个数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

这里delim是类型的定界符，记录在类型的pg_type项中。在PostgreSQL发行提供的标准数据类型中，所有的都使用一个逗号（,），除了类型box使用一个分号（;）。每个val可以是数组元素类型的一个常量，也可以是一个子数组。一个数组常量的例子是：

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

该常量是一个二维的，3乘3数组，它由3个整数子数组构成。

要设置一个数组常量的一个元素为NULL，在该元素值处写NULL（任何NULL的大写或小写变体都有效）。如果你需要一个真正的字符串值“NULL”，你必须在它两边放上双引号。

（这些种类的数组常数实际是第 4.1.2.7 节讨论的一般类型常量的一种特殊形式。常数最初被当做一个字符串，然后被传给数组的输入转换例程。有必要时可能需要一个显式的类型指定。）

现在我们可以展示一些INSERT语句：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

前两个插入的结果看起来像这样：

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多维数组的每一维都必须有相匹配的长度。不匹配会造成错误，例如：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching
 dimensions
```

ARRAY构造器语法也可以被用于：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

注意数组元素是普通SQL常数或表达式，例如，字符串文字使用单引号而不是双引号包围，因为双引号可以出现在一个数组文字中。ARRAY构造器语法的详细讨论请见第 4.2.12 节

8.15.3. 访问数组

现在，我们可以在该表上运行一些查询。首先，我们展示如何访问一个数组中的一个元素。下面的查询检索在第二季度工资发生变化的雇员的名字：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
```

```
 Carol
(1 row)
```

数组下标写在方括号内。默认情况下，PostgreSQL为数组使用了一种从1开始的编号习惯，即一个具有n个元素的数组从array[1]开始，结束于array[n]。

下面的查询检索所有员工第三季度的工资：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

我们也可以访问一个数组的任意矩形切片或者子数组。一个数组切片可以通过在一个或多个数组维度上指定下界:上界来定义例如，下面的查询检索Bill在本周头两天日程中的第一项：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果任何维度被写成一个切片，即包含一个冒号，那么所有的维度都被看成是切片对待。其中任何只有一个数字（无冒号）的维度被视作是从1到指定的数字。例如，下面例子中的[2]被认为是[1:2]：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

为了避免和非切片情况搞混，最好在所有的维度上都使用切片语法，例如[1:2][1:1]而不是[2][1:1]。

可以省略一个切片说明符的lower-bound或者 upper-bound（亦可两者都省略），缺失的 边界会被数组下标的上下限所替代。例如：

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
```

```
(1 row)
```

如果数组本身为空或者任何一个下标表达式为空，访问数组下标表达式将会返回空值。如果下标超过了数组边界，下标表达式也会返回空值（这种情况不会抛出错误）。例如，如果schedule目前具有的维度是[1:3][1:2]，那么引用schedule[3][3]将得到NULL。相似地，使用错误的下标号引用一个数组会得到空值而不是错误。

如果数组本身或者任何一个下标表达式为空，则一个数组切片表达式也会得到空值。但是，在其他情况例如选择一个完全位于当前数组边界之外的切片时，一个切片表达式会得到一个空（零维）数组而不是空值（由于历史原因，这并不符合非切片行为）。 如果所请求的切片和数组边界重叠，那么它会被缩减为重叠的区域而不是返回空。

任何数组值的当前维度可以使用array_dims函数获得：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

array_dims产生一个text结果，它便于人类阅读但是不便于程序读取。 Dimensions can also be retrieved with 也可以通过array_upper和array_lower来获得维度，它们将分别返回一个指定数组的上界和下界：

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

array_length将返回一个指定数组维度的长度：

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

cardinality返回一个数组中在所有维度上的元素总数。 这实际上是调用unnest将会得到的行数：

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. 修改数组

一个数组值可以被整个替换：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

或者使用ARRAY表达式语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

一个数组也可以在一个元素上被更新：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

或者在一个切片上被更新：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

也可以使用省略lower-bound或者 upper-bound的切片语法，但是只能用于 更新一个不是NULL 或者零维的数组值（否则无法替换现有的下标界线）。

一个已存储的数组值可以被通过为其还不存在的元素赋值来扩大之。任何位于之前已存在的元素和新元素之间的位置都将被空值填充。例如，如果数组myarray目前有4个元素，在用一个更新对myarray[6]赋值后它将有6个元素，其中myarray[5]为空值。目前，采用这种方式扩大数组只允许使用在一维数组上。

带下标的赋值方式允许创建下标不是从1开始的数组。例如，我们可以为myarray[-2:7]赋值来创建一个下标值从-2到7的数组。

新的数组值也可以通过串接操作符||构建：

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

串接操作符允许把一个单独的元素加入到一个一维数组的开头或末尾。它也能接受两个N维数组，或者一个N维数组和一个N+1维数组。

当一个单独的元素被加入到一个一维数组的开头或末尾时，其结果是一个和数组操作数具有相同下界下标的新数组。例如：

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
```

```
 array_dims
------------
 [1:3]
(1 row)
```

当两个具有相同维度数的数组被串接时，其结果保留左操作数的外维度的下界下标。结果将是一个数组，它由左操作数的每一个元素以及紧接着的右操作数的每一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

当一个N维数组被放在另一个N+1维数组的前面或者后面时，结果和上面的例子相似。每一个N维子数组实际上是N+1维数组外维度的一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

一个数组也可以通过使用函数array_prepend、array_append或array_cat构建。前两个函数仅支持一维数组，但array_cat支持多维数组。 一些例子：

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)
```

```
SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

在简单的情况中，上面讨论的串接操作符比直接使用这些函数更好。不过，由于 串接操作符需要服务于所有三种情况，所以它的负担比较重，在有些情况下使用 这些函数之一有助于避免混淆。例如：

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- 没有指定类型的文字被当做一个数组
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                  -- 这个也是
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;                 -- 未修饰的 NULL 也是如此
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);     -- 这可能才是想要的意思
 array_append
--------------
 {1,2,NULL}
```

在上面的例子中，解析器看到在串接操作符的一遍看到了一个整数数组，并且在 另一边看到了一个未确定类型的常量。它用来决定该常量类型的启发式规则是假 定它和该操作符的另一个输入具有相同的类型 — 在这种情况中是整数数 组。因此串接操作符表示array_cat而不是 array_append。如果这样做是错误的选择，它可以通过将该常 量造型成数组的元素类型来修复。但是显式地使用array_append 可能是一种最好的方案。

8.15.5. 在数组中搜索

要在一个数组中搜索一个值，每一个值都必须被检查。这可以手动完成，但是我们必须知道数组的尺寸。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

但是这对于大型数组来说太过冗长，且在数组尺寸未知时无法使用。一种可选的方法可见第 9.23 节上面的查询可以被替换为：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

此外，我们还可以查找所有元素值都为10000的数组所在的行：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

另外，generate_subscripts函数也可以用来完成类似的查找。例如：

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

该函数的描述见表 9.59。

我们也可以使用&&操作符来搜索一个数组，它会检查左操作数是否与右操作数重叠。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

该操作符和其他数组操作符的进一步描述请见第 9.18 节如第 11.2 所述，它可以使用一个合适的索引来提速。

你也可以使用array_position和array_positions在一个数组中搜索特定值。前者返回值在数组中第一次出现的位置的下标。后者返回一个数组，其中有该值在数组中的所有出现位置的下标。例如：

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

> **提示**
>
> 数组不是集合，在其中搜索指定数组元素可能是数据设计失误的表现。考虑使用一个独立的表来替代，其中每一行都对应于一个数组元素。这将更有利于搜索，并且对于大量元素的可扩展性更好。

8.15.6. 数组输入和输出语法

一个数组值的外部文本表现由根据数组元素类型的I/O转换规则解释的项构成，并在其上加上修饰用于指示数组结构。修饰包括数组值周围的花括号（{和}）以及相邻项之间的定界字符。定界字符通常是一个逗号（,），但是也可能是别的：它由数组元素类型的typdelim设置决定。在PostgreSQL发行版提供的标准数据类型中，除了box类型使用分号（;）之外，其他都是用逗号。在一个多维数组中，每一个维度（行、平面、方体等）都有其自己的花括号层次，且同层的被花括号限定的相邻实体之间也必须有定界符。

如果元素值是空字符串、包含花括号、包含定界字符、包含双引号、包含反斜线、包含空白或者匹配词NULL，数组输出例程将在元素值周围放上双引号。嵌在元素值中的双引号以及反斜线将被反斜线转义。对于数字数据类型可以安全地假设双引号绝不会出现，但是对于文本数据类型我们必须准备好处理可能出现亦可能不出现的引号。

默认情况下，一个数组的一个维度的下界索引值被设置为1。要表示具有其他下界的数组，数组下标的范围应在填充数组内容之前被显式地指定好。这种修饰包括在每个数组维度上下界周围的方括号（[]），以及上下界之间的一个冒号（:）定界符。数组维度修饰后面要跟一个等号（=）。例如：

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
```

```
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

只有当数组的维度中有一个或多个的下界不为1时，数组输出例程才会在结果中包括维度。

如果为一个元素给定的值是NULL（或者是任何变体），该元素将被设置为NULL。任何引号或反斜线的存在将阻止这种行为，而允许为元素值输入“NULL”的字面意思。为了向后兼容PostgreSQL的8.2之前的版本，可将array_nulls配置参数设置为off来阻止将NULL识别为NULL。

如前所示，在写一个数组值时我们可以在任何单独数组元素周围使用引号。如果元素值可能混淆数组值分析器时，我们必须 这样做。例如，包含花括号、逗号（或者数据类型的定界符）、双引号、反斜线或首尾有空白的元素必须使用双引号。空字符串和匹配单词NULL的字符串也必须使用双引号。要把一个双引号或反斜线放在一个使用了双引号的数组元素值中，需要在它前面放一个反斜线。作为一种选择，我们可以免去使用引号而使用反斜线转义的方式来保护可能被认为是数组语法的所有数据字符。

我们可以在左括号前面或右括号后面增加空白。我们也可以在任何单独的项之前或之后加上空白。在所有这些情况中空白将被忽略。但是，在被使用了双引号的元素中的空白以及周围有其他非空白字符的空白不会被忽略。

> **提示**
>
> 在SQL命令中写数组值时，ARRAY构造器语法（见第 4.2.12 节常常比数组文字语法要更容易使用。在ARRAY中，单独的元素值可以使用不属于数组成员时的方式来书写。

8.16. 组合类型

一个组合类型表示一行或一个记录的结构，它本质上就是一个域名和它们数据类型的列表。PostgreSQL允许把组合类型用在很多能用简单类型的地方。例如，一个表的一列可以被声明为一种组合类型。

8.16.1. 组合类型的声明

这里有两个定义组合类型的简单例子：

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

该语法堪比CREATE TABLE，不过只能指定域名和类型，当前不能包括约束（例如NOT NULL）。注意AS关键词是必不可少的，如果没有它，系统将认为用户想要的是一种不同类型的CREATE TYPE命令，并且你将得到奇怪的语法错误。

定义了类型之后，我们可以用它们来创建表：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

or functions:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

只要你创建了一个表，也会自动创建一个组合类型来表示表的行类型，它具有和表一样的名称。例如，如果我们说：

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

那么和上面所示相同的inventory_item组合类型将成为一种副产品，并且可以按上面所说的进行使用。不过要注意当前实现的一个重要限制：因为没有约束与一个组合类型相关，显示在表定义中的约束不会应用于表外组合类型的值（要解决这个问题，可以在该组合类型上创建一个域，并且把想要的约束应用为这个域上的CHECK约束）。

8.16.2. 构造组合值

要把一个组合值写作一个文字常量，将该域值封闭在圆括号中并且用逗号分隔它们。你可以在任何域值周围放上双引号，并且如果该域值包含逗号或圆括号则必须这样做（更多细节见下文）。这样，一个组合常量的一般格式是下面这样的：

```
'( val1 , val2 , ... )'
```

一个例子是：

```
'("fuzzy dice",42,1.99)'
```

这将是上文定义的inventory_item类型的一个合法值。要让一个域为 NULL，在列表中它的位置上根本不写字符。例如，这个常量指定其第三个域为 NULL：

```
'("fuzzy dice",42,)'
```

如果你写一个空字符串而不是 NULL，写上两个引号：

```
'("",42,)'
```

这里第一个域是一个非 NULL 空字符串，第三个是 NULL。

（这些常量实际上只是第 4.1.2.7 节讨论的一般类型常量的特殊类型。该常量最初被当做一个字符串并且被传递给组合类型输入转换例程。有必要用一次显式类型说明来告知要把该常量转换成何种类型。）。

ROW表达式也能被用来构建组合值。在大部分情况下，比起使用字符串语法，这相当简单易用，因为你不必担心多层引用。我们已经在上文用过这种方法：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

只要在表达式中有多于一个域，ROW 关键词实际上就是可选的，因此这些可以被简化成：

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

第 4.2.13 节更加详细地讨论了ROW表达式语法。

8.16.3. 访问组合类型

要访问一个组合列的一个域，可以写成一个点和域的名称，更像从一个表名中选择一个域。事实上，它太像从一个表名中选择，这样我们不得不使用圆括号来避免让解析器混淆。例如，你可能尝试从例子表on_hand中选取一些子域：

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

这不会有用，因为名称item会被当成是一个表名，而不是on_hand的一个列名。你必须写成这样：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

或者你还需要使用表名（例如在一个多表查询中），像这样：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

现在加上括号的对象就被正确地解释为对item列的引用，然后可以从中选出子域。

只要你从一个组合值中选择一个域，相似的语法问题就适用。例如，要从一个返回组合值的函数的结果中选取一个域，你需要这样写：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括号，这将生成一个语法错误。

特殊的域名称*表示“所有的域”，第 8.16.5 节有进一步的解释。

8.16.4. 修改组合类型

这里有一些插入和更新组合列的正确语法的例子。首先，插入或者更新一整个列：

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

第一个例子忽略ROW，第二个例子使用它，我们可以用两者之一完成。

我们能够更新一个组合列的单个子域：

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

注意这里我们不需要（事实上也不能）把圆括号放在正好出现在SET之后的列名周围，但是当在等号右边的表达式中引用同一列时确实需要圆括号。

并且我们也可以指定子域作为INSERT的目标：

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

如果我们没有为该列的所有子域提供值，剩下的子域将用空值填充。

8.16.5. 在查询中使用组合类型

对于查询中的组合类型有各种特殊的语法规则和行为。这些规则提供了有用的捷径，但是如果你不懂背后的逻辑就会被此困扰。

在PostgreSQL中，查询中对一个表名（或别名）的引用实际上是对该表的当前行的组合值的引用。例如，如果我们有一个如上所示的表inventory_item，我们可以写：

```
SELECT c FROM inventory_item c;
```

这个查询产生一个单一组合值列，所以我们会得到这样的输出：

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

不过要注意简单的名称会在表名之前先匹配到列名，因此这个例子可行的原因仅仅是因为在该查询的表中没有名为c的列。

普通的限定列名语法table_name.column_name可以理解为把字段选择应用在该表的当前行的组合值上（由于效率的原因，实际上不是以这种方式实现）。

当我们写

```
SELECT c.* FROM inventory_item c;
```

时，根据SQL标准，我们应该得到该表展开成列的内容：

```
    name     | supplier_id | price
-------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

就好像查询是

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

尽管如上所示，PostgreSQL将对任何组合值表达式应用这种展开行为，但只要.*所应用的值不是一个简单的表名，你就需要把该值写在圆括号内。例如，如果myfunc()是一个返回组合类型的函数，该组合类型由列a、b和c组成，那么这两个查询有相同的结果：

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

> **提示**
>
> PostgreSQL实际上通过将第一种形式转换为第二种来处理列展开。因此，在这个例子中，用两种语法时对每行都会调用myfunc()三次。如果它是一个开销很大的函数，你可能希望避免这样做，所以可以用一个这样的查询：
>
> ```
> SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
> ```
>
> 把该函数放在一个LATERAL FROM项中会防止它对每一行被调用超过一次。m.*仍然会被展开为m.a, m.b, m.c，但现在那些变量只是对这个FROM项的输出的引用（这里关键词LATERAL是可选的，但我们在这里写上它是为了说明该函数从some_table中得到x）。

当composite_value.*出现在一个SELECT输出列表的顶层中、INSERT/UPDATE/DELETE中的一个RETURNING列表中、一个VALUES子句中或者一个行构造器中时，该语法会导致这种类型的列展开。在所有其他上下文（包括被嵌入在那些结构之一中时）中，把.*附加到一个组合值不会改变该值，因为它表示“所有的列”并且因此同一个组合值会被再次产生。例如，如果somefunc()接受一个组合值参数，这些查询是相同的：

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

在两种情况中，inventory_item的当前行被传递给该函数作为一个单一的组合值参数。即使.*在这类情况中什么也不做，使用它也是一种好的风格，因为它说清了一个组合值的目的是什么。特别地，解析器将会认为c.*中的c是引用一个表名或别名，而不是一个列名，这样就不会出现混淆。而如果没有.*，就弄不清楚c到底是表示一个表名还是一个列名，并且在有一个名为c的列时会优先选择按列名来解释。

另一个演示这些概念的例子是下面这些查询，它们表示相同的东西：

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

所有这些ORDER BY子句指定该行的组合值，导致根据第 9.23.6 节介绍的规则对行进行排序。不过，如果inventory_item包含一个名为c的列，第一种情况会不同于其他情况，因为它表示仅按那一列排序。给定之前所示的列名，下面这些查询也等效于上面的那些查询：

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

（最后一种情况使用了一个省略关键字ROW的行构造器）。

另一种与组合值相关的特殊语法行为是，我们可以使用函数记法来抽取一个组合值的字段。解释这种行为的简单方式是记法field(table)和table.field是可以互换的。例如，这些查询是等效的：

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
```

```
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

此外，如果我们有一个函数接受单一的组合类型参数，我们可以以任意一种记法来调用它。这些查询全都是等效的：

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

这种函数记法和字段记法之间的等效性使得我们可以在组合类型上使用函数来实现“计算字段”。　一个使用上述最后一种查询的应用不会直接意识到somefunc不是一个真实的表列。

> **提示**
>
> 由于这种行为，让一个接受单一组合类型参数的函数与该组合类型的任意字段具有相同的名称是不明智的。出现歧义时，如果使用了字段名语法，则字段名解释将被选择，而如果使用的是函数调用语法则会选择函数解释。不过，PostgreSQL在版本11之前总是选择字段名解释，除非该调用的语法要求它是一个函数调用。在老的版本中强制函数解释的一种方法是用方案限定函数名，也就是写成schema.func(compositevalue)。

8.16.6. 组合类型输入和输出语法

一个组合值的外部文本表达由根据域类型的 I/O 转换规则解释的项，外加指示组合结构的装饰组成。装饰由整个值周围的圆括号（(和)），外加相邻项之间的逗号（,）组成。圆括号之外的空格会被忽略，但是在圆括号之内空格会被当成域值的一部分，并且根据域数据类型的输入转换规则可能有意义，也可能没有意义。例如，在

```
'(  42)'
```

中，如果域类型是整数则空格会被忽略，而如果是文本则空格不会被忽略。

如前所示，在写一个组合值时，你可以在任意域值周围写上双引号。如果不这样做会让域值迷惑组合值解析器，你就必须这么做。特别地，包含圆括号、逗号、双引号或反斜线的域必须用双引号引用。要把一个双引号或者反斜线放在一个被引用的组合域值中，需要在它前面放上一个反斜线（还有，一个双引号引用的域值中的一对双引号被认为是表示一个双引号字符，这和 SQL 字符串中单引号的规则类似）。另一种办法是，你可以避免引用以及使用反斜线转义来保护所有可能被当作组合语法的数据字符。

一个全空的域值（在逗号或圆括号之间完全没有字符）表示一个　NULL。要写一个空字符串值而不是 NULL，可以写成""。

如果域值是空串或者包含圆括号、逗号、双引号、反斜线或空格，组合输出例程将在域值周围放上双引号（对空格这样处理并不是不可缺少的，但是可以提高可读性）。嵌入在域值中的双引号及反斜线将被双写。

> **注意**
>
> 记住你在一个　SQL　命令中写的东西将首先被解释为一个字符串，然后才会被解释为一个组合。这就让你所需要的反斜线数量翻倍（假定使用了转义字符串语法）。例如，要在组合值中插入一个含有一个双引号和一个反斜线的text域，你需要写成：
>
> ```
> INSERT ... VALUES ('("\"\\")');
> ```

字符串处理器会移除一层反斜线，这样在组合值解析器那里看到的就会是("\"\\")。接着，字符串被交给text数据类型的输入例程并且变成"\（如果我们使用的数据类型的输入例程也会特别处理反斜线，例如bytea，在命令中我们可能需要八个反斜线用来在组合域中存储一个反斜线）。美元引用（见第 4.1.2.4 节可以被用来避免双写反斜线。

提示

当在 SQL 命令中书写组合值时，ROW构造器语法通常比组合文字语法更容易使用。在ROW中，单个域值可以按照平时不是组合值成员的写法来写。

8.17. 范围类型

范围类型是表达某种元素类型（称为范围的subtype）的一个值的范围的数据类型。例如，timestamp的范围可以被用来表达一个会议室被保留的时间范围。在这种情况下，数据类型是tsrange（“timestamp range”的简写）而timestamp是 subtype。subtype 必须具有一种总体的顺序，这样对于元素值是在一个范围值之内、之前或之后就是界线清楚的。

范围类型非常有用，因为它们可以表达一种单一范围值中的多个元素值，并且可以很清晰地表达诸如范围重叠等概念。用于时间安排的时间和日期范围是最清晰的例子；但是价格范围、一种仪器的量程等等也都有用。

8.17.1. 内建范围类型

PostgreSQL 带有下列内建范围类型：

- int4range — integer的范围
- int8range — bigint的范围
- numrange — numeric的范围
- tsrange — 不带时区的 timestamp的范围
- tstzrange — 带时区的 timestamp的范围
- daterange — date的范围

此外，你可以定义自己的范围类型，详见CREATE TYPE。

8.17.2. 例子

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 包含
SELECT int4range(10, 20) @> 3;

-- 重叠
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 抽取上界
```

```
SELECT upper(int8range(15, 25));

-- 计算交集
SELECT int4range(10, 20) * int4range(15, 25);

-- 范围为空吗?
SELECT isempty(numrange(1, 5));
```

范围类型上的操作符和函数的完整列表可见表 9.50以及表 9.51

8.17.3. 包含和排除边界

每一个非空范围都有两个界限，下界和上界。这些值之间的所有点都被包括在范围内。一个包含界限意味着边界点本身也被包括在范围内，而一个排除边界意味着边界点不被包括在范围内。

在一个范围的文本形式中，一个包含下界被表达为“[”而一个排除下界被表达为“(”。同样，一个包含上界被表达为“]”而一个排除上界被表达为“)”（详见第 8.17.5 节。

函数lower_inc和upper_inc分别测试一个范围值的上下界。

8.17.4. 无限（无界）范围

一个范围的下界可以被忽略，意味着所有小于上界的点都被包括在范围中。同样，如果范围的上界被忽略，那么所有比上界大的的都被包括在范围中。如果上下界都被忽略，该元素类型的所有值都被认为在该范围中。

这等效于把下界当作“负无穷”，或者把上界当作“正无穷”。但是注意这些无穷值绝不是范围的元素类型的值，并且绝不是范围的一部分（因此没有所谓的包含无限界限 — 如果你尝试写一个，它将被自动转换成一个排除界限）。

还有，有一些元素类型具有一种“无限”概念，但是那只是范围类型机制所关心的之外的另一种值。例如，在时间戳范围中，[today,]意味着与[today,)相同的东西。但是[today,infinity]意味着与[today,infinity)不同的某种东西 — 后者排除了特殊的timestamp值infinity。

函数lower_inf和upper_inf分别测试一个范围的无限上下界。

8.17.5. 范围输入/输出

一个范围值的输入必须遵循下列模式之一：

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

圆括号或方括号指示上下界是否为排除的或者包含的。注意最后一个模式是empty，它表示一个空范围（一个不包含点的范围）。

lower-bound可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有下界。同样，upper-bound可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有上界。

每个界限值可以使用"（双引号）字符引用。如果界限值包含圆括号、方括号、逗号、双引号或反斜线时，这样做是必须的，因为否则那些字符会被认作范围语法的一部分。要把一个双引号或反斜线放在一个被引用的界限值中，就在它前面放一个反斜线（还有，在一个双引号引用的界限值中的一对双引号表示一个双引号字符，这与 SQL 字符串中的单引号规则类

似）。此外，你可以避免引用并且使用反斜线转义来保护所有数据字符，否则它们会被当做返回语法的一部分。还有，要写一个是空字符串的界限值，则可以写成""，因为什么都不写表示一个无限界限。

范围值前后允许有空格，但是圆括号或方括号之间的任何空格会被当做上下界值的一部分（取决于元素类型，它可能是也可能不是有意义的）。

> **注意**
>
> 这些规则与组合类型文字中书写域值的规则非常相似。更多注解请见第 8.16.6 节

例子：

```
-- 包括 3，不包括 7，并且包括 3 和 7 之间的所有点
SELECT '[3,7)'::int4range;

-- 既不包括 3 也不包括 7，但是包括之间的所有点
SELECT '(3,7)'::int4range;

-- 只包括单独一个点 4
SELECT '[4,4]'::int4range;

-- 不包括点（并且将被标准化为 '空'）
SELECT '[4,4)'::int4range;
```

8.17.6. 构造范围

每一种范围类型都有一个与其同名的构造器函数。使用构造器函数常常比写一个范围文字常数更方便，因为它避免了对界限值的额外引用。构造器函数接受两个或三个参数。两个参数的形式以标准的形式构造一个范围（下界是包含的，上界是排除的），而三个参数的形式按照第三个参数指定的界限形式构造一个范围。第三个参数必须是下列字符串之一：“()”、 “(]”、 “[)”或者 “[]”。 例如：

```
-- 完整形式是：下界、上界以及指示界限包含性/排除性的文本参数。
SELECT numrange(1.0, 14.0, '(]');

-- 如果第三个参数被忽略，则假定为 '[)'。
SELECT numrange(1.0, 14.0);

-- 尽管这里指定了 '(]'，显示时该值将被转换成标准形式，因为 int8range 是一种离散范围类型（见下文）。
SELECT int8range(1, 14, '(]');

-- 为一个界限使用 NULL 导致范围在那一边是无界的。
SELECT numrange(NULL, 2.2);
```

8.17.7. 离散范围类型

一种范围的元素类型具有一个良定义的“步长”，例如integer或date。在这些类型中，如果两个元素之间没有合法值，它们可以被说成是相邻。这与连续范围相反，连续范围中总是（或者几乎总是）可以在两个给定值之间标识其他元素值。例如，numeric类型之上的一个范围就是连续的，timestamp上的范围也是（尽管timestamp具有有限的精度，并且在理论上可以被当做离散的，最好认为它是连续的，因为通常并不关心它的步长）。

另一种考虑离散范围类型的方法是对每一个元素值都有一种清晰的“下一个”或“上一个”值。了解了这种思想之后，通过选择原来给定的下一个或上一个元素值来取代它，就可以在一个范围界限的包含和排除表达之间转换。例如，在一个整数范围类型中，[4,8]和(3,9)表示相同的值集合，但是对于 numeric 上的范围就不是这样。

一个离散范围类型应该具有一个正规化函数，它知道元素类型期望的步长。正规化函数负责把范围类型的相等值转换成具有相同的表达，特别是与包含或者排除界限一致。如果没有指定一个正规化函数，那么具有不同格式的范围将总是会被当作不等，即使它们实际上是表达相同的一组值。

内建的范围类型int4range、int8range和daterange都使用一种正规的形式，该形式包括下界并且排除上界，也就是[)。不过，用户定义的范围类型可以使用其他习惯。

8.17.8. 定义新的范围类型

用户可以定义他们自己的范围类型。这样做最常见的原因是为了使用内建范围类型中没有提供的 subtype 上的范围。例如，要创建一个 subtype float8的范围类型：

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

因为float8没有有意义的“步长”，我们在这个例子中没有定义一个正规化函数。

定义自己的范围类型也允许你指定使用一个不同的子类型 B-树操作符类或者集合， 以便更改排序顺序来决定哪些值会落入到给定的范围中。

如果 subtype 被认为是具有离散值而不是连续值，CREATE TYPE命令应当指定一个canonical函数。正规化函数接收一个输入的范围值，并且必须返回一个可能具有不同界限和格式的等价的范围值。对于两个表示相同值集合的范围（例如[1, 7]和[1, 8)），正规的输出必须一样。选择哪一种表达作为正规的没有关系，只要两个具有不同格式的等价值总是能被映射到具有相同格式的相同值就行。除了调整包含/排除界限格式外，假使期望的补偿比 subtype 能够存储的要大，一个正规化函数可能会舍入边界值。例如，一个timestamp之上的范围类型可能被定义为具有一个一小时的步长，这样正规化函数可能需要对不是一小时的倍数的界限进行舍入，或者可能直接抛出一个错误。

另外，任何打算要和 GiST 或 SP-GiST 索引一起使用的范围类型应当定一个 subtype 差异或subtype_diff函数（没有subtype_diff时索引仍然能工作，但是可能效率不如提供了差异函数时高）。subtype 差异函数采用两个 subtype 输入值，并且返回表示为一个float8值的差（即X减Y）。在我们上面的例子中，可以使用常规float8减法操作符之下的函数。但是对于任何其他 subtype，可能需要某种类型转换。还可能需要一些关于如何把差异表达为数字的创新型想法。为了最大的可扩展性，subtype_diff函数应该同意选中的操作符类和排序规则所蕴含的排序顺序，也就是说，只要它的第一个参数根据排序顺序大于第二个参数，它的结果就应该是正值。

subtype_diff函数的一个不那么过度简化的例子：

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);
```

```
SELECT '[11:10, 23:00]'::timerange;
```

更多关于创建范围类型的信息请参考CREATE TYPE。

8.17.9. 索引

可以为范围类型的表列创建 GiST 和 SP-GiST 索引。例如，要创建一个 GiST 索引：

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

一个 GiST 或 SP-GiST 索引可以加速涉及以下范围操作符的查询： =、 &&、 <@、 @>、 <<、 >>、 -|-、 &<以及 &> （详见表 9.50 。

此外，B-树和哈希索引可以在范围类型的表列上创建。对于这些索引类型，基本上唯一有用的范围操作就是等值。使用相应的< 和 >操作符，对于范围值定义有一种 B-树排序顺序，但是该顺序相当任意并且在真实世界中通常不怎么有用。范围类型的 B-树和哈希支持主要是为了允许在查询内部进行排序和哈希，而不是创建真正的索引。

8.17.10. 范围上的约束

虽然UNIQUE是标量值的一种自然约束，它通常不适合于范围类型。反而，一种排除约束常常更加适合（见CREATE TABLE ... CONSTRAINT ... EXCLUDE）。排除约束允许在一个范围类型上说明诸如“non-overlapping”的约束。例如：

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

该约束将阻止任何重叠值同时存在于表中：

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint
 "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

你可以使用btree_gist扩展来在纯标量数据类型上定义排除约束，然后把它和范围排除结合可以得到最大的灵活性。例如，安装btree_gist之后，只有会议室号码相等时，下列约束将拒绝重叠的范围：

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);
```

```
INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint
 "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01
 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01
 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. 域类型

域是一种用户定义的数据类型，它基于另一种底层类型。根据需要，它可以有约束来限制其有效值为底层类型所允许值的一个子集。如果没有约束，它的行为就和底层类型一样 — 例如，任何适用于底层类型的操作符或函数都对该域类型有效。底层类型可以是任何内建或者用户定义的基础类型、枚举类型、数组类型、组合类型、范围类型或者另一个域。

例如，我们可以在整数之上创建一个域，它只接受正整数：

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

当底层类型的一个操作符或函数适用于一个域值时，域会被自动向下造型为底层类型。因此，mytable.id - 1的结果会被认为是类型integer而不是posint。我们可以写成(mytable.id - 1)::posint来把结果转换回posint，这会导致域的约束被重新检查。在这种情况下，如果该表达式被应用于一个值为1的id就会错误。把底层类型的值赋给域类型的一个字段或者变量不需要写显式的造型，但是域的约束将会被检查。

更多信息请参考CREATE DOMAIN。

8.19. 对象标识符类型

对象标识符（OID）被PostgreSQL用来在内部作为多个系统表的主键。OID不会被添加到用户创建的表中，除非在创建表时指定了WITH OIDS或者default_with_oids配置变量被启用。类型oid表示一个对象标识符。也有多个oid的别名类型：regproc、regprocedure、regoper、regoperator、regclass、regtype、regrole、regnamespace、regc 8.24显示了一个概览。

oid类型目前被实现为一个无符号4字节整数。因此，在大型数据库中它并不足以提供数据库范围内的唯一性，甚至在一些大型的表中也无法提供表范围内的唯一性。于是，我们不鼓励使用一个用户定义表的OID列作为主键。OID最好只被用于引用系统表。

oid类型本身除了比较之外只有很少的操作。不过，它可以被造型成整数，并且接着可以使用标准的整数操作符进行操纵（这样做时要注意有符号和无符号之间可能出现的混乱）。

OID的别名类型除了特定的输入和输出例程之外没有别的操作。这些例程可以接受并显示系统对象的符号名，而不是类型oid使用的原始数字值。别名类型使查找对象的OID值变得简单。例如，要检查与一个表mytable有关的pg_attribute行，你可以写：

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

而不是：

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

虽然从它本身看起来并没有那么糟，它仍然被过度简化了。如果有多个名为mytable的表存在于不同的模式中，就可能需要一个更复杂的子选择来选择右边的OID。regclass输入转换器会根据模式路径设置处理表查找，并且因此它会自动地完成这种“右边的事情”。类似地，对于一个数字OID的符号化显示可以很方便地通过将表OID造型成regclass来实现。

表 8.24. 对象标识符类型

名字	引用	描述	值示例
oid	任意	数字形式的对象标识符	564182
regproc	pg_proc	函数名字	sum
regprocedure	pg_proc	带参数类型的函数	sum(int4)
regoper	pg_operator	操作符名字	+
regoperator	pg_operator	带参数类型的操作符	*(integer,integer) or -(NONE,integer)
regclass	pg_class	关系名字	pg_type
regtype	pg_type	数据类型名字	integer
regrole	pg_authid	角色名	smithee
regnamespace	pg_namespace	名字空间名称	pg_catalog
regconfig	pg_ts_config	文本搜索配置	english
regdictionary	pg_ts_dict	文本搜索字典	simple

所有用于由名字空间组织的对象的 OID 别名类型都接受模式限定的名字，如果没有被限定的对象在当前搜索路径中无法找到时，将会在输出时显示模式限定的名字。regproc和regoper别名类型将只接受唯一的（非重载的）输入名字，因此它们的使用是受限的；对于大多数使用，regprocedure或regoperator更合适。对于regoperator，通过使用NONE来替代未使用的操作数可以标识一元操作符。

大部分 OID 别名类型的一个附加性质是依赖性的创建。如果这些类型之一的一个常量出现在一个存储的表达式（如一个列默认值表达式或视图）中，它会在被引用的对象上创建一个依赖。例如，如果一个列有一个默认值表达式nextval('my_seq'::regclass)，PostgreSQL会理解该默认值表达式是依赖于序列my_seq的，在删除该默认值表达式之前系统将不允许删除该序列。regrole是这个性质的唯一例外。这种类型的常量不允许出现在这类表达式中。

> **注意**
>
> OID 别名类型不完全遵循事务隔离规则。规划器也把它们当做简单常量， 这可能会导致次优的规划。

另一种系统中使用的标识符类型是xid，或者称为事务（简写为xact）标识符。这是系统列xmin和xmax使用的数据类型。事务标识符是32位量。

系统使用的第三种标识符类型是cid，或者称为命令标识符。这是系统列cmin和cmax使用的数据类型。命令标识符也是32位量。

系统使用的最后一种标识符类型是tid，或者称为元组标识符（行标识符）。这是系统列ctid使用的数据类型。一个元组ID是一个（块号，块内元组索引）对，它标识了行在它的表中的物理位置。

（这些系统列在第 5.4 节有进一步的解释）。

8.20. pg_lsn 类型

pg_lsn数据类型可以被用来存储 LSN（日志序列号）数据，LSN 是一个指向WAL中的位置的指针。这个类型是XLogRecPtr的一种表达并且是 PostgreSQL的一种内部系统类型。

在内部，一个 LSN 是一个 64 位整数，表示在预写式日志流中的一个字节位置。它被打印成两个最高 8 位的十六进制数，中间用斜线分隔，例如16/B374D848。 pg_lsn类型支持标准的比较操作符，如=和 >。两个 LSN 可以用-操作符做减法， 结果将是分隔两个预写式日志位置的字节数。

8.21. 伪类型

PostgreSQL类型系统包含了一些特殊目的的项，它们被统称为伪类型。一个伪类型不能被用作一个列的数据类型，但是它可以被用来定义一个函数的参数或者结果类型。每一种可用的伪类型都有其可以发挥作用的情况，这些情况的特点是一个函数的行为并不能符合于简单使用或者返回一种特定SQL数据类型的值。表 8.25列出了现有的伪类型。

表 8.25. 伪类型

名字	描述
any	表示一个函数可以接受任意输入数据类型。
anyelement	表示一个函数可以接受任意数据类型（参见第 38.2.5 节。
anyarray	表示一个函数可以接受任意数组数据类型（参见第 38.2.5 节
anynonarray	表示一个函数可以接受任意非数组数据类型（参见第 38.2.5 节。
anyenum	表示一个函数可以接受任意枚举数据类型（参见第 38.2.5 和第 8.7 节。
anyrange	表示一个函数可以接受任意范围数据类型（参见第 38.2.5 和第 8.17 节。
cstring	表示一个函数接受或者返回一个非空结尾的C字符串。
internal	表示一个函数接受或返回一个服务器内部数据类型。
language_handler	一个被声明为返回language_handler的过程语言调用处理器。
fdw_handler	一个被声明为返回fdw_handler的外部数据包装器处理器。
index_am_handler	一个被声明为返回index_am_handler索引访问方法处理器。
tsm_handler	一个被声明为返回tsm_handler的表采样方法处理器。
record	标识一个接收或者返回一个未指定的行类型的函数。
trigger	一个被声明为返回trigger的触发器函数。

名字	描述
event_trigger	一个被声明为返回event_trigger的事件触发器函数。
pg_ddl_command	标识一种对事件触发器可用的 DDL 命令的表达。
void	表示一个函数不返回值。
unknown	标识一种还未被解析的类型，例如一个未修饰的字符文本。
opaque	一种已被废弃的类型名称，以前它用于实现以上的很多种目的。

用C编写的函数（不管是内建的还是动态载入的）可以被声明为接受或返回这些为数据类型的任意一种。函数的作者应当保证当一个伪类型被用作一个参数类型时函数的行为是安全的。

用过程语言编写的函数只有在其实现语言允许的情况下才能使用伪类型。目前大部分过程语言都禁止使用伪类型作为一种参数类型，并且只允许使用void和record作为结果类型（如果函数被用于一个触发器或者事件触发器， trigger或者event_trigger也被允许作为结果类型）。某些过程语言也支持在多态函数中使用类型anyelement、anyarray、anynonarray、anyenum和anyrange。

internal伪类型用于定义只在数据库系统内部调用的函数，这些函数不会被SQL直接调用。如果一个函数拥有至少一个internal类型的参数，则它不能从SQL中被调用。为了保持这种限制的类型安全性，遵循以下编码规则非常重要：不要创建任何被声明要返回internal的函数，除非它有至少一个internal参数。

第 9 章 函数和操作符

PostgreSQL为内建的数据类型提供了大量的函数和操作符。 用户也可以定义它们自己的函数和操作符， 如第 V 部所述。psql命令\df和\do可以分别被用于显示所有可用的函数和操作符的列表。

如果你关心移植性，那么请注意，我们在本章描述的大多数函数和操作符， 除了最琐碎的算术和比较操作符以及一些做了明确标记的函数以外，都没有在SQL标准里声明。某些这种扩展的功能也出现在许多其它SQL数据库管理系统中，并且在很多情况下多个实现的这种功能是相互兼容的和一致的。本章也并没有穷尽一切信息；一些附加的函数在本手册的相关小节里出现。

9.1. 逻辑操作符

常用的逻辑操作符有：

AND
OR
NOT

SQL使用三值的逻辑系统，包括真、假和null，null表示“未知”。观察下面的真值表：

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符AND和OR是可交换的，也就是说，你可以交换左右操作数而不影响结果。但是请参阅第 4.2.14 获取有关子表达式计算顺序的更多信息。

9.2. 比较函数和操作符

常见的比较操作符都可用，如表 9.1所示。

表 9.1. 比较操作符

操作符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于

操作符	描述
<> or !=	不等于

> **注意**
>
> !=操作符在分析器阶段被转换成<>。不能把!=和<>操作符实现为做不同的事。

比较操作符可以用于所有可以比较的数据类型。所有比较操作符都是双目操作符，它们返回boolean类型；类似于1 < 2 < 3的表达式是非法的（因为没有<操作符可以比较一个布尔值和3）。

如表 9.2所示，也有一些比较谓词。它们的行为和操作符很像，但是具有 SQL 标准所要求的特殊语法。

表 9.2. 比较谓词

谓词	描述
a BETWEEN x AND y	在x和y之间
a NOT BETWEEN x AND y	不在x和y之间
a BETWEEN SYMMETRIC x AND y	在对比较值排序后位于x和y之间
a NOT BETWEEN SYMMETRIC x AND y	在对比较值排序后不位于x和y之间
a IS DISTINCT FROM b	不等于，空值被当做一个普通值
a IS NOT DISTINCT FROM b	等于，空值被当做一个普通值
expression IS NULL	是空值
expression IS NOT NULL	不是空值
expression ISNULL	是空值（非标准语法）
expression NOTNULL	不是空值（非标准语法）
boolean_expression IS TRUE	为真
boolean_expression IS NOT TRUE	为假或未知
boolean_expression IS FALSE	为假
boolean_expression IS NOT FALSE	为真或者未知
boolean_expression IS UNKNOWN	值为未知
boolean_expression IS NOT UNKNOWN	为真或者为假

BETWEEN谓词可以简化范围测试：

a BETWEEN x AND y

等效于

a >= x AND a <= y

注意BETWEEN认为终点值是包含在范围内的。 NOT BETWEEN可以做相反比较：

a NOT BETWEEN x AND y

等效于

```
a < x OR a > y
```

BETWEEN SYMMETRIC和BETWEEN相似，不过BETWEEN SYMMETRIC不要求AND左边的参数小于或等于右边的参数。如果左参数不是小于等于右参数，这两个参数会自动被交换，这样总是会应用一个非空范围。

当有一个输入为空时，普通的比较操作符会得到空（表示“未知”），而不是真或假。例如，7 = NULL得到空，7 <> NULL也一样。如果这种行为不合适，可以使用IS [NOT] DISTINCT FROM谓词：

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

对于非空输入，IS DISTINCT FROM和<>操作符一样。不过，如果两个输入都为空，它会返回假。而如果只有一个输入为空，它会返回真。类似地，IS NOT DISTINCT FROM对于非空输入的行为与=相同，但是当两个输入都为空时它返回真，并且当只有一个输入为空时返回假。因此，这些谓词实际上把空值当作一种普通数据值而不是“unknown”。

要检查一个值是否为空，使用下面的谓词：

```
expression IS NULL
expression IS NOT NULL
```

或者等效，但并不标准的谓词：

```
expression ISNULL
expression NOTNULL
```

不要写expression = NULL，因为NULL是不“等于”NULL的（控制代表一个未知的值，因此我们无法知道两个未知的数值是否相等）。

> **提示**
>
> 有些应用可能要求表达式expression = NULL在expression得出空值时返回真。我们强烈建议这样的应用修改成遵循 SQL 标准。但是，如果这样修改不可能完成，那么我们可以使用配置变量transform_null_equals。如果打开它，PostgreSQL将把x = NULL子句转换成x IS NULL。

如果expression是行值，那么当行表达式本身为非空值或者行的所有域为非空时IS NULL为真。由于这种行为，IS NULL和IS NOT NULL并不总是为行值表达式返回反转的结果，特别是，一个同时包含 NULL 和非空值的域将会对两种测试都返回假。在某些情况下，写成row IS DISTINCT FROM NULL或者row IS NOT DISTINCT FROM NULL会更好，它们只会检查整个行值是否为空而不需要在行的域上做额外的测试。

布尔值也可以使用下列谓词进行测试：

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

这些谓词将总是返回真或假，从来不返回空值，即使操作数是空也如此。空值输入被当做逻辑值“未知”。 请注意实际上IS UNKNOWN和IS NOT UNKNOWN分别与IS NULL和IS NOT NULL相同， 只是输入表达式必须是布尔类型。

如表 9.3中所示，也有一些比较相关的函数可用。

表 9.3. 比较函数

函数	描述	例子	例子结果
num_nonnulls(VARIADIC "any")	返回非空参数的数量	num_nonnulls(1, NULL, 2)	2
num_nulls(VARIADIC "any")	返回空参数的数量	num_nulls(1, NULL, 2)	1

9.3. 数学函数和操作符

PostgreSQL为很多类型提供了数学操作符。对于那些没有标准数学表达的类型（如日期/时间类型），我们将在后续小节中描述实际的行为。

表 9.4展示了所有可用的数学操作符。

表 9.4. 数学操作符

操作符	描述	例子	结果
+	加	2 + 3	5
-	减	2 - 3	-1
*	乘	2 * 3	6
/	除（整数除法截断结果）	4 / 2	2
%	模（取余）	5 % 4	1
^	指数（从左至右结合）	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘	5 !	120
!!	阶乘（前缀操作符）	!! 5	120
@	绝对值	@ -5.0	5
&	按位与	91 & 15	11
|	按位或	32 | 3	35
#	按位异或	17 # 5	20
~	按位求反	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

按位操作操作符只能用于整数数据类型，而其它的操作符可以用于全部数字数据类型。按位操作的操作符还可以用于位串类型bit和bit varying， 如表 9.13所示。

表 9.5显示了可用的数学函数。在该表中，dp表示double precision。这些函数中有许多都有多种不同的形式，区别是参数不同。除非特别指明，任何特定形式的函数都返回和它的参

数相同的数据类型。 处理double precision数据的函数大多数是在宿主系统的 C 库基础上实现的；因此，边界情况下的准确度和行为是根据宿主系统而变化的。

表 9.5. 数学函数

函数	返回类型	描述	例子	结果
abs(x)	（和输入相同）	绝对值	abs(-17.4)	17.4
cbrt(dp)	dp	立方根	cbrt(27.0)	3
ceil(dp or numeric)	（和输入相同）	不小于参数的最近的整数	ceil(-42.8)	-42
ceiling(dp or numeric)	（和输入相同）	不小于参数的最近的整数（ceil的别名）	ceiling(-95.3)	-95
degrees(dp)	dp	把弧度转为角度	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	y/x的整数商	div(9,4)	2
exp(dp or numeric)	（和输入相同）	指数	exp(1.0)	2.71828182845905
floor(dp or numeric)	（和输入相同）	不大于参数的最近的整数	floor(-42.8)	-43
ln(dp or numeric)	（和输入相同）	自然对数	ln(2.0)	0.693147180559945
log(dp or numeric)	（和输入相同）	以10为底的对数	log(100.0)	2
log(b numeric, x numeric)	numeric	以b为底的对数	log(2.0, 64.0)	6.0000000000
mod(y, x)	（和参数类型相同）	y/x的余数	mod(9,4)	1
pi()	dp	“π”常数	pi()	3.14159265358979
power(a dp, b dp)	dp	求a的b次幂	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	求a的b次幂	power(9.0, 3.0)	729
radians(dp)	dp	把角度转为弧度	radians(45.0)	0.785398163397448
round(dp or numeric)	（和输入相同）	圆整为最接近的整数	round(42.4)	42
round(v numeric, s int)	numeric	圆整为s位小数数字	round(42.4382, 2)	42.44
scale(numeric)	integer	参数的精度（小数点后的位数）	scale(8.41)	2
sign(dp or numeric)	（和输入相同）	参数的符号（-1, 0, +1）	sign(-8.4)	-1
sqrt(dp or numeric)	（和输入相同）	平方根	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	（和输入相同）	截断（向零靠近）	trunc(42.8)	42
trunc(v numeric, s int)	numeric	截断为s位小数位置的数字	trunc(42.4382, 2)	42.43

函数	返回类型	描述	例子	结果
width_bucket(op dp, b1 dp, b2 dp, count int)	int	返回一个桶号，这个桶是在一个柱状图中operand将被分配的那个桶，该柱状图有count个散布在范围b1到b2上的等宽桶。对于超过该范围的输入，将返回0或者count+1	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(op numeric, b1 numeric, b2 numeric, count int)	int	返回一个桶号，这个桶是在一个柱状图中operand将被分配的那个桶，该柱状图有count个散布在范围b1到b2上的等宽桶。对于超过该范围的输入，将返回0或者count+1	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(operand anyelement, thresholds anyarray)	int	返回一个桶号，这个桶是在给定数组中operand将被分配的桶，该数组列出了桶的下界。对于一个低于第一个下界的输入返回0。thresholds数组必须被排好序， 最小的排在最前面，否则将会得到意想不到的结果	width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])	2

表 9.6 展示了用于产生随机数的函数。

表 9.6. 随机函数

函数	返回类型	描述
random()	dp	范围 0.0 <= x < 1.0 中的随机值
setseed(dp)	void	为后续的random()调用设置种子（值为于 -1.0 和 1.0 之间，包括边界值）

random()返回的值的特征取决于系统实现。 它不适合用于加密应用，如果需要用于加密应用请参考pgcrypto模块。

最后，表 9.7 显示了可用的三角函数。所有三角函数都有类型为double precision的参数和返回类型。每一种三角函数都有两个变体，一个以弧度度量角，另一个以角度度量角。

表 9.7. 三角函数

函数（弧度）	函数（角度）	描述
acos(x)	acosd(x)	反余弦
asin(x)	asind(x)	反正弦
atan(x)	atand(x)	反正切
atan2(y, x)	atan2d(y, x)	y/x的反正切
cos(x)	cosd(x)	余弦
cot(x)	cotd(x)	余切
sin(x)	sind(x)	正弦
tan(x)	tand(x)	正切

> **注意**
>
> 另一种使用以角度度量的角的方法是使用早前展示的单位转换函数radians()和degrees()。不过，使用基于角度的三角函数更好，因为这类方法能避免sind(30)等特殊情况下的舍入偏差。

9.4. 字符串函数和操作符

本节描述了用于检查和操作字符串值的函数和操作符。在这个环境中的串包括所有类型character、character varying和text的值。除非另外说明，所有下面列出的函数都可以处理这些类型，不过要小心的是，在使用character类型的时候， 它有自动填充空白的潜在影响。有些函数还可以处理位串类型。

SQL定义了一些字符串函数，它们使用关键字，而不是逗号来分隔参数。详情请见表 9.8。PostgreSQL也提供了这些函数使用正常函数调用语法的版本（见表 9.9 ）。

> **注意**
>
> 由于存在从那些数据类型到text的隐式强制措施，在PostgreSQL 8.3之前，这些函数也可以接受多种非字符串数据类型。这些强制措施在目前的版本中已经被删除，因为它们常常导致令人惊讶的行为。不过，字符串串接操作符（||）仍然接受非字符串输入，只要至少一个输入是一种字符串类型，如表 9.8所示。对于其他情况，如果你需要复制之前的行为，可以为text插入一个显式强制措施。

表 9.8. SQL字符串函数和操作符

函数	返回类型	描述	例子	结果
string || string	text	串接	'Post' || 'greSQL'	PostgreSQL
string || non-string or non-string || string	text	使用一个非字符串输入的串接	'Value: ' || 42	Value: 42
bit_length(string)	int	串中的位数	bit_length('jose')	32
char_length(string)	int	串中字符数	char_length('jose')	4

函数	返回类型	描述	例子	结果
or character_length(string)				
lower(string)	text	将字符串转换为小写形式	lower('TOM')	tom
octet_length(string)	int	串中的字节数	octet_length('jose')	4
overlay(string placing string from int [for int])	text	替换子串	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(substring in string)	int	定位指定子串	position('om' in 'Thomas')	3
substring(string [from int] [for int])	text	提取子串	substring('Thomas' from 2 for 3)	hom
substring(string from pattern)	text	提取匹配POSIX正则表达式的子串。模式匹配详情见第 9.7 节	substring('Thomas' from '...$')	mas
substring(string from pattern for escape)	text	提取匹配SQL正则表达式的子串。模式匹配详情见第 9.7 节	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [characters] from string)	text	从string的开头、结尾或者两端（both是默认值）移除只包含characters（默认是一个空格）中字符的最长字符串	trim(both 'xyz' from 'yxTomxx')	Tom
trim([leading | trailing | both] [from] string [, characters])	text	trim()的非标准版本	trim(both from 'xTomxx', 'x')	Tom
upper(string)	text	将字符串转换成大写形式	upper('tom')	TOM

还有额外的串操作函数可以用，它们在表 9.9 中列出。它们有些在内部用于实现表 9.8 中列出的SQL标准字符串函数。

表 9.9. 其他字符串函数

函数	返回类型	描述	例子	结果
ascii(string)	int	参数第一个字符的ASCII代码。对于UTF8返回该字符的Unicode代码点。对于其他多字节编码，该参	ascii('x')	120

函数	返回类型	描述	例子	结果
		数必须是一个ASCII字符。		
btrim(string text [, characters text])	text	从string的开头或结尾删除最长的只包含characters（默认是一个空格）的串	btrim('xyxtrimyyx', 'xyz')	trim
chr(int)	text	给定代码的字符。对于UTF8该参数被视作一个Unicode代码点。对于其他多字节编码该参数必须指定一个ASCII字符。NULL (0) 字符不被允许，因为文本数据类型不能存储这种字节。	chr(65)	A
concat(str "any" [, str "any" [, ...]])	text	串接所有参数的文本表示。NULL参数被忽略。	concat('abcde', 2, NULL, 22)	abcde222
concat_ws(sep text, str "any" [, str "any" [, ...]])	text	将除了第一个参数外的其他参数用分隔符串接在一起。第一个参数被用作分隔符字符串。NULL 参数被忽略。	concat_ws(',', 'abcde', 2, NULL, 22)	abcde,2,22
convert(string bytea, src_encoding name, dest_encoding name)	bytea	将字符串转换为dest_encoding。原始编码由src_encoding指定。string在这个编码中必须可用。转换可以使用CREATE CONVERSION定义。也有一些预定义的转换。可用的转换请见表 9.10	convert('text_in_utf8', 'UTF8', 'LATIN1')	用LATIN1 encoding (ISO 8859-1) 表示的text_in_utf8
convert_from(string bytea, src_encoding name)	text	将字符串转换为数据库编码。原始编码由src_encoding指定。string在这个编码中必须可用。	convert_from('text_in_utf8', 'UTF8')	用当前数据库编码表示的text_in_utf8
convert_to(string text,	bytea	将字符串转换为dest_encoding。	convert_to('some text', 'UTF8')	用UTF8编码表达的some text

函数	返回类型	描述	例子	结果
dest_encoding name)				
decode(string text, format text)	bytea	从string中的文本表达解码二进制数据。format的选项和encode中的一样。	decode('MTIzAAE=', 'base64')	\x3132330001
encode(data bytea, format text)	text	将二进制数据编码成一个文本表达。支持的格式有：base64、hex、escape。escape将零字节和高位组字节转换为八进制序列（\nnn）和双写的反斜线。	encode('123\000\001', 'base64')	MTIzAAE=
format(formatstr text [, formatarg "any" [, ...]])	text	根据一个格式字符串格式化参数。该函数和C函数sprintf相似。见第 9.4.1 节	format('Hello %s, %1$s', 'World')	Hello World, World
initcap(string)	text	将每一个词的第一个字母转换为大写形式并把剩下的字母转换为小写形式。词是由非字母数字字符分隔的字母数字字符的序列。	initcap('hi THOMAS')	Hi Thomas
left(str text, n int)	text	返回字符串中的前n个字符。当n为负时，将返回除了最后|n|个字符之外的所有字符。	left('abcde', 2)	ab
length(string)	int	string中的字符数	length('jose')	4
length(string bytea, encoding name)	int	string在给定编码中的字符数。string必须在这个编码中有效。	length('jose', 'UTF8')	4
lpad(string text, length int [, fill text])	text	将string通过前置字符fill（默认是一个空格）填充到长度length。如果string已经长于length，则它被（从右边）截断。	lpad('hi', 5, 'xy')	xyxhi

函数	返回类型	描述	例子	结果
ltrim(string text [, characters text])	text	从string的开头删除最长的只包含characters（默认是一个空格）的串	ltrim('zzzytest', 'xyz')	test
md5(string)	text	计算string的MD5 哈希，返回十六进制的结果	md5('abc')	900150983cd24fb0d6963f7d28e17f72
parse_ident(qualified_identifier text [, strictmode boolean DEFAULT true])	text[]	把qualified_identifier分解成一个标识符数组，移除单个标识符上的任何引号。默认情况下，最后一个标识符后面的多余字符会被当做错误。但是如果第二个参数为false，那么这一类多余的字符会被忽略（这种行为对于解析函数之类的对象名称有用）。注意这个函数不会截断超长标识符。如果想要进行截断，可以把结果转换成name[]。	parse_ident('"SomeSchema".someTable')	{SomeSchema,sometable}
pg_client_encoding()	name	当前的客户端编码名字	pg_client_encoding()	SQL_ASCII
quote_ident(string text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个标识符。只有需要时才会加上引号（即，如果字符串包含非标识符字符或可能是大小写折叠的）。嵌入的引号会被正确地双写。参见例 43.1。	quote_ident('Foo bar')	"Foo bar"
quote_literal(string text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个字符串文字。嵌入的引号会被正确地双写。注	quote_literal(E'O\'Reilly')	'O''Reilly'

函数	返回类型	描述	例子	结果
		意quote_literal对空输入返回空；如果参数可能为空，quote_nullable通常更合适。参见例 43.1		
quote_literal(value anyelement)	text	强迫给定值为文本并且接着将它用引号包围作为一个文本。嵌入的单引号和反斜线被正确的双写。	quote_literal(42.5)	'42.5'
quote_nullable(string text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个字符串文字；或者，如果参数为空，返回NULL。嵌入的引号会被正确地双写。请参见例 43.1	quote_nullable(NULL)	NULL
quote_nullable(value anyelement)	text	强迫给定值为文本并且接着将它用引号包围作为一个文本；或者，如果参数为空，返回NULL。嵌入的单引号和反斜线被正确的双写。	quote_nullable(42.5)	'42.5'
regexp_match(string text, pattern text [, flags text])	text[]	返回一个POSIX正则表达式与string的第一个匹配得到的子串。更多信息请见第 9.7.3 节	regexp_match('foobarbequebaz', '(bar)(beque)')	{bar,beque}
regexp_matches(string text, pattern text [, flags text])	setof text[]	返回一个POSIX正则表达式与string匹配得到的子串。更多信息请见第 9.7.3 节	regexp_matches('foobarbequebaz', 'ba.', 'g')	{bar} {baz} (2 rows)
regexp_replace(string text, pattern text, replacement text [, flags text])	text	替换匹配一个POSIX正则表达式的子串。详见第 9.7.3 节	regexp_replace('Thomas', '.[mN]a.', 'M')	ThM

函数	返回类型	描述	例子	结果
regexp_split_to_array(text, pattern text [, flags text])	text[] array(string	使用一个POSIX正则表达式作为分隔符划分string。详见第 9.7.3 节	regexp_split_to_array('hello world', '\s+')	{hello,world}
regexp_split_to_table(text, pattern text [, flags text])	setof text table(string	使用一个POSIX正则表达式作为分隔符划分string。详见第 9.7.3 节	regexp_split_to_table('hello world', '\s+')	hello world (2 rows)
repeat(string text, number int)	text	重复string指定的number次	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	将string中出现的所有子串from替换为子串to	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
reverse(str)	text	返回反转的字符串。	reverse('abcde')	edcba
right(str text, n int)	text	返回字符串中的最后n个字符。如果n为负，返回除最前面的|n|个字符外的所有字符。	right('abcde', 2)	de
rpad(string text, length int [, fill text])	text	将string通过增加字符fill（默认为一个空格）填充到长度length。如果string已经长于length则它会被截断。	rpad('hi', 5, 'xy')	hixyx
rtrim(string text [, characters text])	text	从string的结尾删除最长的只包含characters（默认是一个空格）的串	rtrim('testxxzx', 'xyz')	test
split_part(string text, delimiter text, field int)	text	按delimiter划分string并返回给定域（从1开始计算）	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(string, substring)	int	指定子串的位置（和position(substring in string)相同，但是注意相反的参数顺序）	strpos('high', 'ig')	2
substr(string, from [, count])	text	提取子串（与substring(string from from for count)相同）	substr('alphabet', 3, 2)	ph

函数	返回类型	描述	例子	结果
starts_with(string, prefix)	bool	如果string以prefix开始则返回真。	starts_with('alphabet', 'alph')	t
to_ascii(string text [, encoding text])	text	将string从另一个编码转换到ASCII（只支持从LATIN1、LATIN2、LATIN9和WIN1250编码的转换）	to_ascii('Karel')	Karel
to_hex(number int or bigint)	text	将number转换到它等效的十六进制表示	to_hex(2147483647)	7fffffff
translate(string text, from text, to text)	text	string中任何匹配from集合中一个字符的字符会被替换成to集合中的相应字符。如果from比to长，from中的额外字符会被删除。	translate('12345', '143', 'ax')	a2x5

concat、concat_ws和format函数是可变的，因此可以把要串接或格式化的值作为一个标记了VARIADIC关键字的数组进行传递（见第 38.5.5 节。数组的元素被当作函数的独立普通参数一样处理。如果可变数组参数为 NULL，concat和concat_ws返回 NULL，但format把 NULL 当作一个零元素数组。

还可以参阅第 9.20 节的string_agg。

表 9.10. 内建转换

转换名 [a]	源编码	目标编码
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8

转换名 [a]	源编码	目标编码
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R

转换名 [a]	源编码	目标编码
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253

转换名 [a]	源编码	目标编码
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] 转换名遵循一种标准命名模式：将全部非字母数字字符替换为下划线的源编码的官方名称，后面跟上_to_，最后是按照相似方式处理过的目标编码名称。因此，名称可能会不同于习惯的编码名称。

9.4.1. format

函数format根据一个格式字符串产生格式化的输出，其形式类似于 C 函数sprintf。

```
format(formatstr text [, formatarg "any" [, ...] ])
```

formatstr是一个格式字符串，它指定了结果应该如何被格式化。格式字符串中的文本被直接复制到结果中，除了使用格式说明符的地方。格式说明符在字符串中扮演着占位符的角色，它定义后续的函数参数如何被格式化及插入到结果中。每一个formatarg参数会被根据其数据类型的常规输出规则转换为文本，并接着根据格式说明符被格式化和插入到结果字符串中。

格式说明符由一个%字符开始并且有这样的形式

%[position][flags][width]type

其中的各组件域是：

position（可选）

一个形式为n$的字符串，其中n是要打印的参数的索引。索引 1 表示formatstr之后的第一个参数。如果position被忽略，默认会使用序列中的下一个参数。

flags（可选）

控制格式说明符的输出如何被格式化的附加选项。当前唯一支持的标志是一个负号（-），它将导致格式说明符的输出会被左对齐（left-justified）。除非width域也被指定，否者这个域不会产生任何效果。

width（可选）

指定用于显示格式说明符输出的最小字符数。输出将被在左部或右部（取决于-标志）用空格填充以保证充满该宽度。太小的宽度设置不会导致输出被截断，但是会被简单地忽略。宽度可以使用下列形式之一指定：一个正整数；一个星号（*）表示使用下一个函数参数作为宽度；或者一个形式为*n$的字符串表示使用第n个函数参数作为宽度。

如果宽度来自于一个函数参数，则参数在被格式说明符的值使用之前就被消耗掉了。如果宽度参数是负值，结果会在长度为abs(width)的域中被左对齐（如果-标志被指定）。

type（必需）

格式转换的类型，用于产生格式说明符的输出。支持下面的类型：

- s将参数值格式化为一个简单字符串。一个控制被视为一个空字符串。
- I将参数值视作 SQL 标识符，并在必要时用双写引号包围它。如果参数为空，将会是一个错误（等效于quote_ident）。
- L将参数值引用为 SQL 文字。一个空值将被显示为不带引号的字符串NULL（等效于quote_nullable）。

除了以上所述的格式说明符之外，要输出一个文字形式的%字符，可以使用特殊序列%%。

下面有一些基本的格式转换的例子：

```
SELECT format('Hello %s', 'World');
结果：Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
结果：Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
结果：INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
结果：INSERT INTO locations VALUES(E'C:\\Program Files')
```

下面是使用width域和-标志的例子：

```
SELECT format('|%10s|', 'foo');
结果：|       foo|

SELECT format('|%-10s|', 'foo');
结果：|foo       |
```

```
SELECT format('|%*s|', 10, 'foo');
结果：|       foo|

SELECT format('|%*s|', -10, 'foo');
结果：|foo       |

SELECT format('|%-*s|', 10, 'foo');
结果：|foo       |

SELECT format('|%-*s|', -10, 'foo');
结果：|foo       |
```

这些例子展示了position域的例子：

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
结果：Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
结果：|       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
结果：|       foo|
```

不同于标准的 C 函数sprintf，PostgreSQL的format函数允许将带有或者不带有position域的格式说明符被混在同一个格式字符串中。一个不带有position域的格式说明符总是使用最后一个被消耗的参数的下一个参数。另外，format函数不要求所有函数参数都被用在格式字符串中。例如：

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
结果：Testing three, two, three
```

对于安全地构造动态 SQL 语句，%I和%L格式说明符特别有用。参见例 43.1

9.5. 二进制串函数和操作符

本节描述那些检查和操作类型为bytea的值的函数和操作符。

SQL定义了一些使用关键字而不是逗号来分割参数的串函数。详情请见表 9.11。PostgreSQL也提供了这些函数使用常规函数调用语法的版本（参阅表 9.12 。

> **注意**
>
> 本页中显示的示例结果假设服务器参数bytea_output被设置为escape（传统PostgreSQL格式）。

表 9.11. SQL二进制串函数和操作符

函数	返回类型	描述	例子	结果
string || string	bytea	串连接	'\\Post'::bytea || '\047gres\\000'::bytea	\\Post'gres\000
octet_length(string)	int	二进制串中的字节数	octet_length('jo\000se'::bytea)	5

函数	返回类型	描述	例子	结果
overlay(string placing string from int [for int])	bytea	替换子串	overlay('Th\000omas'::bytea placing '\002\003'::bytea from 2 for 3)	T\\002\\003mas
position(substring in string)	int	指定子串的位置	position('\000om'in 'Th\000omas'::bytea)	3
substring(string [from int] [for int])	bytea	提取子串	substring('Th\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from string)	bytea	从string的开头或结尾删除只包含出现在bytes中字节的最长串	trim('\000\001'::bytea from '\000Tom\001'::bytea)	Tom

还有一些二进制串处理函数可以使用，在表 9.12列出。 其中有一些是在内部使用，用于实现表 9.11列出的 SQL 标准串函数。

表 9.12. 其他二进制串函数

函数	返回类型	描述	例子	结果
btrim(string bytea, bytes bytea)	bytea	从string的开头或结尾删除只由出现在bytes中字节组成的最长串	btrim('\000trim\001'::bytea, '\000\001'::bytea)	trim
decode(string text, format text)	bytea	从string中的文本表示解码二进制数据。format的参数和在encode中一样。	decode('123\000456', 'escape')	123\000456
encode(data bytea, format text)	text	将二进制数据编码为一个文本表示。支持的格式有：base64、hex、escape。escape将零字节和高位组字节转换为八进制序列（\nnn）和双反斜线。	encode('123\000456'::bytea, 'escape')	123\000456
get_bit(string, offset)	int	从串中抽取位	get_bit('Th\000omas'::bytea, 45)	1
get_byte(string, offset)	int	从串中抽取字节	get_byte('Th\000omas'::bytea, 4)	109
length(string)	int	二进制串的长度	length('jo\000se'::bytea)	5
md5(string)	text	计算string的MD5哈希码，以十六进制形式返回结果	md5('Th\000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1

函数	返回类型	描述	例子	结果
set_bit(string, offset, newvalue)	bytea	设置串中的位	set_bit('Th\000omas'::bytea, 45, 0)	Th\000omAs
set_byte(string, offset, newvalue)	bytea	设置串中的字节	set_byte('Th\000omas'::bytea, 4, 64)	Th\000o@as
sha224(bytea)	bytea	SHA-224哈希	sha224('abc')	\x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
sha256(bytea)	bytea	SHA-256哈希	sha256('abc')	\xba7816bf8f01cfea414140de5daeb00361a396177a9cb410ff61f20015
sha384(bytea)	bytea	SHA-384哈希	sha384('abc')	\xcb00753f45a35e8bb5a03d699ac6272c32ab0eded1631a8b605a43ff5b8086072ba1e7cc2358baeca134c825
sha512(bytea)	bytea	SHA-512哈希	sha512('abc')	\xddaf35a193617abacc417349ae2012e6fa4e89a97ea20a9eeee64b55d32192992a274fc1a836ba3c23a3feeb454d4423643ce80e2a9ac94fa54ca4

get_byte和set_byte把一个二进制串中的一个字节计数为字节 0。get_bit和set_bit在每一个字节中从右边起计数位；例如位 0 是第一个字节的最低有效位，而位 15 是第二个字节的最高有效位。

注意由于历史原因，函数md5返回的是一个十六进制编码的text值，而SHA-2函数返回类型bytea。可以使用函数encode和decode在两者之间转换，例如encode(sha256('abc'), 'hex')可以得到一个十六进制编码的文本表示。

参见第 9.20 节的聚集函数string_agg以及第 35.4 节的大对象函数。

9.6. 位串函数和操作符

本节描述用于检查和操作位串的函数和操作符，也就是操作类型为bit和bit varying的值的函数和操作符。除了常用的比较操作符之外，还可以使用表 9.13里显示的操作符。&、|和#的位串操作数必须等长。在移位的时候，保留原始的位串的的长度，如例子所示。

表 9.13. 位串操作符

操作符	描述	例子	结果
||	连接	B'10001' || B'011'	10001011
&	按位与	B'10001' & B'01101'	00001
|	按位或	B'10001' | B'01101'	11101
#	按位异或	B'10001' # B'01101'	11100
~	按位求反	~ B'10001'	01110
<<	按位左移	B'10001' << 3	01000
>>	按位右移	B'10001' >> 2	00100

下面的SQL标准函数除了可以用于字符串之外，也可以用于位串： length、 bit_length、 octet_length、 position、 substring、 overlay。

下面的函数除了可以用于二进制串之外，也可以用于位串： get_bit、 set_bit。 当使用于一个位串时，这些函数将串的第一（最左）位计数为位 0。

另外，我们可以在整数和bit之间来回转换。一些例子：

```
44::bit(10)                     0000101100
44::bit(3)                      100
cast(-44 as bit(12))            111111010100
'1110'::bit(4)::integer         14
```

请注意，如果只是转换为“bit”，意思是转换成bit(1)，因此只会转换整数的最低有效位。

> **注意**
>
> 把一个整数转换成bit(n)将拷贝整数的最右边的n位。 把一个整数转换成比整数本身长的位串，就会在最左边扩展符号。

9.7. 模式匹配

PostgreSQL提供了三种独立的实现模式匹配的方法：SQL LIKE操作符、更近一些的SIMILAR TO操作符（SQL:1999 里添加进来的）和POSIX-风格的正则表达式。除了这些基本的“这个串匹配这个模式吗？”操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

> **提示**
>
> 如果你的模式匹配的要求超出了这些，请考虑用 Perl 或 Tcl 写一个用户定义的函数。

> **小心**
>
> 虽然大部分的正则表达式搜索都能被很快地执行，但是正则表达式仍可能被人为地弄成需要任意长的时间和任意量的内存进行处理。要当心从不怀好意的来源接受正则表达式搜索模式。如果必须这样做，建议加上语句超时限制。
>
> 使用SIMILAR TO模式的搜索具有同样的安全性危险， 因为SIMILAR TO提供了很多和 POSIX-风格正则表达式相同的能力。
>
> LIKE搜索比其他两种选项简单得多，因此在使用 不怀好意的模式来源时要更安全些。

9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

如果该string匹配了提供的pattern，那么LIKE表达式返回真（和预期的一样，如果LIKE返回真，那么NOT LIKE表达式返回假， 反之亦然。一个等效的表达式是NOT (string LIKE pattern)）。

如果pattern不包含百分号或者下划线，那么该模式只代表它本身的串；这时候LIKE的行为就象等号操作符。在pattern里的下划线 （_）代表（匹配）任何单个字符； 而一个百分号（%）匹配任何零或更多个字符的序列。

一些例子：

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE模式匹配总是覆盖整个串。因此，要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

要匹配文本的下划线或者百分号，而不是匹配其它字符，在pattern里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜线，但是你可以用ESCAPE子句指定一个不同的逃逸字符。要匹配逃逸字符本身，写两个逃逸字符。

> **注意**
>
> 如果你关掉了standard_conforming_strings，你在文串常量中写的任何反斜线都需要被双写。详见第 4.1.2.1 节

请注意反斜线在串文本里已经有特殊含义了，所以如果你写一个包含反斜线的模式常量，那你就要在 SQL 语句里写两个反斜线。因此，写一个匹配单个反斜线的模式实际上要在语句里写四个反斜线。你可以通过用 ESCAPE 选择一个不同的逃逸字符来避免这样；这样反斜线就不再是 LIKE 的特殊字符了。但仍然是字符文本分析器的特殊字符，所以你还是需要两个反斜线。）我们也可以通过写ESCAPE ''的方式不选择逃逸字符，这样可以有效地禁用逃逸机制，但是没有办法关闭下划线和百分号在模式中的特殊含义。

关键字ILIKE可以用于替换LIKE，它令该匹配根据活动区域成为大小写无关。这个不属于SQL标准而是一个PostgreSQL扩展。

操作符~~等效于LIKE，而~~*对应ILIKE。还有 !~~和!~~*操作符分别代表NOT LIKE和NOT ILIKE。所有这些操作符都是PostgreSQL特有的。

在仅需要从字符串的开始部分搜索的情况，还有前缀操作符^@和相应的starts_with函数可以使用。

9.7.2. SIMILAR TO正则表达式

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

SIMILAR TO操作符根据自己的模式是否匹配给定串而返回真或者假。它和LIKE非常类似，只不过它使用 SQL 标准定义的正则表达式理解模式。SQL 正则表达式是在LIKE标记和普通的正则表达式标记的奇怪的杂交。

类似LIKE，SIMILAR TO操作符只有在它的模式匹配整个串的时候才能成功；这一点和普通的正则表达式的行为不同，在普通的正则表达式里，模式匹配串的任意部分。和LIKE类似的地方还有，SIMILAR TO使用_和%作为分别代表任意单个字符和任意串的通配符（这些可以比得上 POSIX 正则表达式里的.和.*）。

除了这些从LIKE借用的功能之外，SIMILAR TO支持下面这些从 POSIX 正则表达式借用的 模式匹配元字符：

- |表示选择（两个候选之一）。
- *表示重复前面的项零次或更多次。
- +表示重复前面的项一次或更多次。

- ?表示重复前面的项零次或一次。
- {m}表示重复前面的项刚好m次。
- {m,}表示重复前面的项m次或更多次。
- {m,n}表示重复前面的项至少m次并且不超过n次。
- 可以使用圆括号()把多个项组合成一个逻辑项。
- 一个方括号表达式[...]声明一个字符类，就像 POSIX 正则表达式一样。

注意点号（.）不是SIMILAR TO的一个元字符。

和LIKE一样，反斜线禁用所有这些元字符的特殊含义；当然我们也可以用ESCAPE指定一个不同的逃逸字符。

一些例子：

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

带三个参数的substring，即substring(string from pattern for escape-character)，提供了抽取一个匹配 SQL 正则表达式的子串的方法。和SIMILAR TO一样，声明的模式必须匹配整个数据串，否则函数失败并返回空值。为了标识在成功的时候应该返回的模式部分，模式 必须包含逃逸字符的两次出现，并且后面要跟上双引号（"）。匹配这两个标记之间的模式的文本将被返回。

一些例子，使用#"定界返回串：

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. POSIX正则表达式

表 9.14列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

表 9.14. 正则表达式匹配操作符

操作符	描述	例子
~	匹配正则表达式，大小写敏感	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，大小写不敏感	'thomas' ~* '.*Thomas.*'
!~	不匹配正则表达式，大小写敏感	'thomas' !~ '.*Thomas.*'
!~*	不匹配正则表达式，大小写不敏感	'thomas' !~* '.*vadim.*'

POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的含义。许多 Unix 工具，例如egrep、sed或awk使用一种与我们这里描述的类似的模式匹配语言。

正则表达式是一个字符序列，它是定义一个串集合 （一个正则集）的缩写。 如果一个串是正则表达式描述的正则集中的一员时， 我们就说这个串匹配该正则表达式。 和LIKE一样，

模式字符准确地匹配串字符， 除非在正则表达式语言里有特殊字符 — 不过正则表达式用的 特殊字符和LIKE用的不同。 和LIKE模式不一样的是，正则表达式允许匹配串里的任何位置，除非该正则表达式显式地挂接在串的开头或者结尾。

一些例子：

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

POSIX模式语言的详细描述见下文。

带两个参数的substring函数，即substring(string from pattern)，提供了抽取一个匹配POSIX 正则表达式模式的子串的方法。如果没有匹配它返回空值，否则就是文本中匹配模式的那部分。 但是如果该模式包含任何圆括号，那么将返回匹配第一对子表达式（对应第一个左圆括号的） 的文本。如果你想在表达式里使用圆括号而又不想导致这个例外，那么你可以在整个表达式外边放上一对圆括号。 如果你需要在想抽取的子表达式前有圆括号，参阅后文描述的非捕获性圆括号。

一些例子：

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

regexp_replace函数提供了将匹配 POSIX 正则表达式模式的子串替换为新文本的功能。 它的语法是 regexp_replace(source, pattern, replacement [, flags])。 如果没有匹配pattern，那么返回不加修改的source串。 如果有匹配，则返回的source串里面的匹配子串将被replacement串替换掉。replacement串可以包含\n， 其中\n是 1 到 9， 表明源串里匹配模式里第n个圆括号子表达式的子串应该被插入， 并且它可以包含\&表示应该插入匹配整个模式的子串。如果你需要放一个文字形式的反斜线在替换文本里，那么写\\。flags参数是一个可选的文本串，它包含另个或更多单字母标志，这些标志可以改变函数的行为。标志i指定大小写无关的匹配，而标志g指定替换每一个匹配的子串而不仅仅是第一个。支持的标志（但不是g）在表 9.22中描述。

一些例子：

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

regexp_match返回一个文本数组，它包含一个POSIX正则表达式模式与一个字符串第一个匹配所得到的子串。其语法是regexp_match(string, pattern [, flags])。如果没有匹配，则结果为NULL。如果找到一个匹配并且pattern不包含带括号的子表达式，那么结果是一个单一元素的文本数组，其中包含匹配整个模式的子串。如果找到一个匹配并且pattern含有带括号的子表达式，那么结果是一个文本数组，其中第n个元素是与pattern的第n个圆括号子表达式匹配的子串（“非捕获”圆括号不计入在内，详见下文）。flags参数是一个可选的文本字符串，它包含零个或者更多个可以改变该函数行为的单字母标志。所支持的标志在表 9.22中介绍。

一些例子：

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
```

```
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

在通常情况下，人们只是想要的大整个匹配的子串或者NULL（没有匹配），可以写成这样

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

regexp_matches函数返回一个文本数组的集合，其中包含着一个POSIX正则表达式模式与一个字符串匹配得到的子串。它和regexp_match具有相同的语法。如果没有匹配，这个函数不会返回行。如果有一个匹配并且给定了g标志，则返回一行。如果有N个匹配并且给定了g标志，则返回N行。每一个返回的行都是一个文本数组，其中含有整个匹配的子串或者匹配pattern的圆括号子表达式的子串，这和上面对regexp_match的介绍一样。regexp_matches接受表 9.22中展示的所有标志，外加令它返回所有匹配而不仅仅是第一个匹配的g标志。

一些例子：

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **提示**
>
> 在大部分情况下，regexp_matches()应该与g标志一起使用，因为如果只是想要第一个匹配，使用regexp_match()会更加简单高效。不过，regexp_match()仅存在于PostgreSQL版本10以及更高的版本中。当在较老的版本中使用时，一种常用的技巧是把regexp_matches()调用放在子选择中，例如：
>
> ```
> SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
> ```
>
> 如果有一个匹配，则这个语句会产生一个文本数组，否则返回NULL，这和regexp_match()的做法一样。如果没有子选择，这个查询对于没有匹配的表行根本不会产生输出，显然那不是想要的行为。

regexp_split_to_table把一个 POSIX 正则表达式模式当作一个定界符来分离一个串。它的语法形式是regexp_split_to_table(string, pattern [, flags])。如果没有与pattern的匹配，该函数返回string。如果有至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。flags参数是一个可选的文本串，它包含零个或更多单字母标志，这些标识可以改变该函数的行为。regexp_split_to_table能支持的标志在表 9.2中描述。

regexp_split_to_array函数的行为和regexp_split_to_table相同，不过regexp_split_to_array会把它的结果以一个text数组的形式返回。它的语法是regexp_split_to_array(string, pattern [, flags])。这些参数和regexp_split_to_table的相同。

一些例子：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy
 dog', '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s
+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

正如上一个例子所示，正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者由regexp_match和regexp_matches实现，但是通常前者是实际中最常用的行为。其他软件系统如Perl也使用相似的定义。

9.7.3.1. 正则表达式细节

PostgreSQL的正则表达式是使用 Henry Spencer 写的一个包来实现的。下面的正则表达式的大部分描述都是从他的手册页中逐字拷贝过来的。

正则表达式（RE），在POSIX 1003.2 中定义， 它有两种形式：扩展的RE或者是ERE（大概地说就是那些在egrep里的）， 基本的RE或者是BRE（大概地说就是那些在ed里的）。PostgreSQL支持两种形式，并且还实现了一些POSIX标准中没有但是在类似 Perl 或者 Tcl 这样的语言中得到广泛应用的一些扩展。使用了那些非POSIX扩展的RE叫高级RE， 或者本文档里说的ARE。ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容（以及更多的限制）。我们首先描述 ARE 和 ERE 形式， 描述那些只适用于 ARE 的特性，然后描述 BRE 的区别是什么。

> **注意**
>
> PostgreSQL初始时总是推测一个正则表达式遵循 ARE 规则。但是，可以通过为 RE 模式预置一个embedded option来选择限制更多的 ERE 或 BRE 规则，如第 9.7.3.4 节所述。这对为期望准确的POSIX 1003.2 规则的应用提供兼容性很有用。

一个正则表达式被定义为一个或更多分支，它们之间被|分隔。只要能匹配其中一个分支的东西都能匹配正则表达式。

一个分支是一个或多个量化原子或者约束连接而成。一个原子匹配第一个，然后后面的原子匹配第二个， 以此类推；一个空分支匹配空串。

一个量化原子是一个原子， 后面可能跟着一个量词。没有量词的时候，它匹配一个原子，有量词的时候，它可以匹配若干个原子。一个原子可以是在表 9.15里面显示的任何可能。可能的量词和它们的含义在表 9.16里显示。

一个约束匹配一个空串，但只是在满足特定条件下才匹配。 约束可以在能够使用原子的地方使用，只是它不能跟着量词。简单的约束在表 9.17里显示； 更多的约束稍后描述。

表 9.15. 正则表达式原子

原子	描述
(re)	（其中re是任何正则表达式） 匹配一个对re的匹配，匹配将为可能的报告被记下
(?:re)	同上，但是匹配不会为了报告而被记下 （一个“非捕获”圆括号集） （只对 ARE）
.	匹配任意单个字符
[chars]	一个方括号表达式， 匹配chars中的任意一个（详见第 9.7.3.2 节
\k	（其中k是一个非字母数字字符） 匹配一个被当作普通字符看待的特定字符， 例如，\\匹配一个反斜线字符
\c	其中c是一个字母数字 （可能跟着其它字符），它是一个逃逸， 参阅第 9.7.3.3 节仅对 ARE； 在 ERE 和 BRE 中，它匹配c）

原子	描述
{	如果后面跟着一个字符，而不是数字， 那么就匹配左花括弧{；如果跟着一个数字， 那么它是range的开始（见下文）
x	其中x是一个没有其它意义的单个字符，则匹配该字符

RE 不能以反斜线（\）结尾。

> **注意**
>
> 如果你关掉了standard_conforming_strings，任何你写在文字串常量中的反斜线都需要被双写。详见第 4.1.2.1 节

表 9.16. 正则表达式量词

量词	匹配
*	一个由原子的 0 次或更多次匹配组成的序列
+	一个由原子的 1 次或更多次匹配组成的序列
?	一个由原子的 0 次或 1 次匹配组成的序列
{m}	一个由原子的正好m次匹配组成的序列
{m,}	一个由原子的m次或更多次匹配组成的序列
{m,n}	一个由原子的从m次到n次（包括）匹配组成的序列；m不能超过n
*?	*的非贪婪版本
+?	+的非贪婪版本
??	?的非贪婪版本
{m}?	{m}的非贪婪版本
{m,}?	{m,}的非贪婪版本
{m,n}?	{m,n}的非贪婪版本

使用{...}的形式被称作范围。 一个范围内的数字m和n都是无符号十进制整数， 允许的数值从 0 到 255（包含）。

非贪婪的量词（只在 ARE 中可用）匹配对应的正常 （贪婪）模式，区别是它寻找最少的匹配，而不是最多的匹配。详见第 9.7.3.5 节

> **注意**
>
> 一个量词不能紧跟在另外一个量词后面，例如**是非法的。量词不能作为表达式或者子表达式的开头，也不能跟在^或者|后面。

表 9.17. 正则表达式约束

约束	描述
^	串开头的匹配
$	串末尾的匹配

约束	描述
(?=re)	在匹配re的子串开始的任何点的positive lookahead匹配（只对 ARE）
(?!re)	在匹配re的子串开始的任何点的negative lookahead匹配（只对 ARE）
(?<=re)	只要有一个点上有一个子串匹配re端，positive lookbehind就在这个点上匹配（只对 ARE）
(?<!re)	只要有一个点上没有子串匹配re端，negative lookbehind就在这个点上匹配（只对 ARE）

lookahead 和 lookbehind 约束不能包含后引用 （参阅第 9.7.3.3 节，并且其中的所有圆括号 都被认为是非捕获的。

9.7.3.2. 方括号表达式

方括号表达式是一个包围在[]中的字符列表。它通常匹配列表中的任意单个字符（但见下文）。 如果列表以^开头，它匹配任意单个不在该列表参与部分中的字符。如果该列表中两个字符用-隔开， 那它就是那两个字符（包括在内）之间的所有字符范围的缩写，例如，在ASCII中[0-9]匹配任何十进制数字。两个范围共享一个端点是非法的，例如，a-c-e。范围与字符集关系密切， 可移植的程序应该避免依靠它们。

想在列表中包含文本]，可以让它做列表的首字符（如果使用了^，需要放在其后）。 想在列表中包含文本-，可以让它做列表的首字符或者尾字符，或者一个范围的第二个端点。 想在列表中把文本-当做范围的起点， 把它用[.和.]包围起来，这样它就成为一个排序元素（见下文）。 除了这些字符本身、一些用[的组合（见下段）以及逃逸（只在 ARE 中有效）以外，所有其它特殊字符 在方括号表达式里都失去它们的特殊含义。特别是，在 ERE 和 BRE 规则下\不是特殊的， 但在 ARE 里，它是特殊的（引入一个逃逸）。

在一个方括号表达式里，一个排序元素（一个字符、一个被当做一个单一字符排序的多字符序列或者一个表示上面两种情况的排序序列名称） 包含在[.和.]里面的时候表示该排序元素的字符序列。该序列被当做该方括号列表 的一个单一元素。这允许一个包含多字符排序元素的方括号表达式去匹配多于一个字符，例如，如果排序序列包含一个ch排序元素， 那么 RE [[.ch.]]*c匹配chchcc的头五个字符。

> **注意**
>
> PostgreSQL当前不支持多字符排序元素。这些信息描述了将来可能有的行为。

在方括号表达式里，包围在[=和=]里的排序元素是一个等价类， 代表等效于那一个的所有排序元素的字符序列，包括它本身（如果没有其它等效排序元素，那么就好象封装定界符是[.和 .]）。例如，如果o和^是一个等价类的成员，那么[[=o=]]、[[=^=]]和[o^]都是同义的。一个等价类不能是一个范围的端点。

在方括号表达式里，在[:和:]里面封装的字符类的名字代表属于该类的所有字符的列表。标准的字符类名字是：alnum、 alpha、blank、 cntrl、digit、 graph、lower、print、punct、 space、upper、 xdigit。 它们代表在ctype中定义的字符类。 一个区域可以会提供其他的类。字符类不能用做一个范围的端点。

方括号表达式里有两个特例：方括号表达式[[:<:]]和[[:>:]]是约束，分别匹配一个单词开头和结束的空串。 单词定义为一个单词字符序列，前面和后面都没有其它单词字符。单词字符是一个alnum字符（和ctype中定义的一样） 或者一个下划线。这是一个扩展，兼容 POSIX 1003.2， 但那里面并没有说明， 而且在准备移植到其他系统里去的软件里一定要小心使用。通常下文描述的约束逃逸更好些（它们并非更标准，但是更容易键入）。

9.7.3.3. 正则表达式逃逸

逃逸是以\开头，后面跟着一个字母数字字符得特殊序列。 逃逸有好几种变体：字符项、类缩写、约束逃逸以及后引用。在 ARE 里， 如果一个\后面跟着一个字母数字，但是并未组成一个合法的逃逸， 那么它是非法的。在 ERE 中没有逃逸：在方括号表达式之外，一个后面跟着字母数字字符的\只是表示该字符是一个普通的字符，而且在一个方括号表达式里，\是一个普通的字符（后者实际上在 ERE 和 ARE 不兼容）。

字符项逃逸用于便于我们在 RE 中声明那些不可打印的或其他习惯的字符。它们显示在表 9.18中。

类缩写逃逸用来提供一些常用的字符类缩写。它们显示在表 9.19中。

约束逃逸是一个约束，如果满足特定的条件，它匹配该空串。它们显示在表 9.20中。

后引用（\n）匹配数字\n指定的被前面的圆括号子表达式匹配的同一个串 （参阅表 9.21） 。例如， ([bc])\1匹配bb或者cc， 但是不匹配bc或者cb。RE 中子表达式必须完全在后引用前面。子表达式以它们的先导圆括号的顺序编号。非捕获圆括号并不定义子表达式。

表 9.18. 正则表达式字符项逃逸

逃逸	描述
\a	警告（响铃）字符，和 C 中一样
\b	退格，和 C 中一样
\B	反斜线（\）的同义词，用来减少双写反斜线
\cX	（其中X是任意字符）低序5位和X相同的字符，它的其他位都是零
\e	排序序列名为ESC的字符，如果无法做到该字符为八进制值 033
\f	换页，和 C 中一样
\n	新行，和 C 中一样
\r	回车，和 C 中一样
\t	水平制表符，和 C 中一样
\uwxyz	（其中wxyz正好是四个十六进制位）十六进制值为0xwxyz的字符
\Ustuvwxyz	（其中stuvwxyz正好是八个十六进制位）十六进制值为0xstuvwxyz的字符
\v	垂直制表符，和 C 中一样
\xhhh	（其中hhh是十六进制位的任意序列）十六进制值为0xhhh的字符（一个单一字符，不管用了多少个十六进制位）
\0	值为0（空字节）的字符
\xy	（其中xy正好是两个八进制位，并且不是一个后引用）八进制值为0xy的字符
\xyz	（其中xyz正好是三个八进制位，并且不是一个后引用）八进制值为0xyz的字符

十六进制位是0-9、a-f和A-F。八进制位是0-7。

指定 ASCII 范围（0-127）之外的值的数字字符项转义的含义取决于数据库编码。 当编码是 UTF-8 时，转义值等价于 Unicode 代码点，例如 \u1234表示字符U+1234。对于其他多字节

编码， 字符项转义通常只是指定该字符的字节值的串接。如果该转义值不对应数据库编码中的任何合法字符，将不会发生错误，但是它不会匹配任何数据。

字符项逃逸总是被当作普通字符。例如，\135是 ASCII 中的]， 但\135并不终止一个方括号表达式。

表 9.19. 正则表达式类缩写逃逸

逃逸	描述
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_]（注意下划线是被包括的）
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] （注意下划线是被包括的）

在方括号表达式里，\d、\s和\w会失去它们的外层方括号，而\D、\S和 \W是非法的（也就是说，例如[a-c\d]等效于[a-c[:digit:]]。同样[a-c\D]等效于 [a-c^[:digit:]]的，也是非法的）。

表 9.20. 正则表达式约束逃逸

逃逸	描述
\A	只在串开头匹配（与^的不同请参见第 9.7.3.5 节
\m	只在一个词的开头匹配
\M	只在一个词的末尾匹配
\y	只在一个词的开头或末尾匹配
\Y	只在一个词的不是开头或末尾的点上匹配
\Z	只在串的末尾匹配（与$的不同请参见第 9.7.3.5 节

一个词被定义成在上面[[:<:]]和[[:>:]]中的声明。在方括号表达式里，约束逃逸是非法的。

表 9.21. 正则表达式后引用

逃逸	描述
\m	（其中m是一个非零位）一个到第m个子表达式的后引用
\mnn	（其中m是一个非零位，并且nn是一些更多的位，并且十六进制值mnn不超过目前能看到的封闭捕获圆括号的数目）一个到第mnn个子表达式的后引用

> **注意**
>
> 在八进制字符项逃逸和后引用之间有一个历史继承的歧义存在，这个歧义是通过下面的启发式规则解决的，像上面描述地那样。前导零总是表示这是一个八进制逃逸。 而单个非零数字，如果没有跟着任何其它位，那么总是被认为后引用。 一个多位的非零开头的序列也被认为是后引用，只要它出现在合适的子表达式后面 （也就是说，在后引用的合法范围中的数），否则就被认为是一个八进制。

9.7.3.4. 正则表达式元语法

除了上面描述的主要语法之外，还有几种特殊形式和杂项语法。

如果一个 RE 以***:开头，那么剩下的 RE 都被当作 ARE（这在PostgreSQL中通常是无效的，因为 RE 被假定为 ARE，但是如果 ERE 或 BRE 模式通过flags参数被指定为一个正则表达式函数时，它确实能产生效果）。如果一个 RE 以***=开头， 那么剩下的 RE 被当作一个文本串，所有的字符都被认为是一个普通字符。

一个 ARE 可以以嵌入选项开头：一个序列(?xyz)（这里的xyz是一个或多个字母字符）声明影响剩余 RE 的选项。 这些选项覆盖任何前面判断的选项 — 特别地，它们可以覆盖一个正则表达式操作符隐含的大小写敏感的行为，或者覆盖flags参数中的正则表达式函数。可用的选项字母在表 9.22中显示。注意这些同样的选项字母也被用在正则表达式函数的flags参数中。

表 9.22. ARE 嵌入选项字母

选项	描述
b	RE的剩余部分是一个BRE
c	大小写敏感的匹配（覆盖操作符类型）
e	RE的剩余部分是一个ERE
i	大小写不敏感的匹配（见第 9.7.3.5 节（覆盖操作符类型）
m	n的历史原因的同义词
n	新行敏感的匹配（见第 9.7.3.5 节
p	部分新行敏感的匹配（见第 9.7.3.5 节
q	RE的剩余部分是一个文字（“quoted”）串，全部是普通字符
s	非新行敏感的匹配（默认）
t	紧语法（默认，见下文）
w	逆部分新行敏感（“怪异”）的匹配（见第 9.7.3.5 节
x	扩展语法（见下文）

嵌入选项在)终止序列时发生作用。它们只在 ARE 的开始处起作用 （在任何可能存在的***:控制器后面）。

除了通常的（紧）RE 语法（这种情况下所有字符都有效）， 还有一种扩展语法，可以通过声明嵌入的x选项获得。在扩展语法里，RE 中的空白字符被忽略，就像那些在#和其后的新行（或 RE 的末尾）之间的字符一样。这样就允许我们给一个复杂的 RE 分段和注释。不过这个基本规则有三种例外：

- 空白字符或前置了\的#将被保留
- 方括号表达式里的空白或者#将被保留
- 在多字符符号里面不能出现空白和注释，例如(?:

为了这个目的，空白是空格、制表符、新行和任何属于空白字符类的字符。

最后，在 ARE 里，方括号表达式外面，序列(?#ttt)（其中ttt是任意不包含一个))的文本）是一个注释， 它被完全忽略。同样，这样的东西是不允许出现在多字符符号的字符中间的，例如 (?:。这种注释更像是一种历史产物而不是一种有用的设施，并且它们的使用已经被废弃；请使用扩展语法来替代。

如果声明了一个初始的***=控制器，那么所有这些元语法扩展都不能使用，因为这样表示把用户输入当作一个文字串而不是 RE 对待。

9.7.3.5. 正则表达式匹配规则

在 RE 可以在给定串中匹配多于一个子串的情况下， RE 匹配串中最靠前的那个子串。如果 RE 可以匹配在那个位置开始 的多个子串，要么是取最长的子串，要么是最短的，具体哪种， 取决于 RE 是贪婪的还是非贪婪的。

一个 RE 是否贪婪取决于下面规则：

- 大多数原子以及所有约束，都没有贪婪属性（因为它们毕竟无法匹配个数变化的文本）。
- 在一个 RE 周围加上圆括号并不会改变其贪婪性。
- 带一个固定重复次数量词 （{m}或者{m}?） 的量化原子和原子自身具有同样的贪婪性（可能是没有）。
- 一个带其他普通的量词（包括{m,n}中m等于n的情况）的量化原子是贪婪的（首选最长匹配）。
- 一个带非贪婪量词（包括{m,n}?中m等于 n的情况）的量化原子是非贪婪的（首选最短匹配）。
- 一个分支 — 也就是说，一个没有顶级|操作符的 RE — 和它里面的第一个有贪婪属性的量化原子有着同样的贪婪性。
- 一个由|操作符连接起来的两个或者更多分支组成的 RE 总是贪婪的。

上面的规则所描述的贪婪属性不仅仅适用于独立的量化原子， 而且也适用于包含量化原子的分支和整个 RE。这里的意思是， 匹配是按照分支或者整个 RE 作为一个整体匹配最长或者最短的可能子串。 一旦整个匹配的长度确定，那么匹配任意特定子表达式的部分就基于该子表达式的贪婪属性进行判断，在 RE 里面靠前的子表达式的优先级高于靠后的子表达式。

一个相应的例子：

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
结果：123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
结果：1
```

在第一个例子里，RE 作为整体是贪婪的，因为Y*是贪婪的。它可以匹配从Y开始的东西，并且它匹配从这个位置开始的最长的串， 也就是，Y123。输出是这里的圆括号包围的部分，或者说是123。在第二个例子里， RE 总体上是一个非贪婪的 RE，因为Y*?是非贪婪的。它可以匹配从Y开始的最短的子串，也就是说Y1。子表达式[0-9]{1,3}是贪婪的，但是它不能修改总体匹配长度的决定； 因此它被迫只匹配1。

简而言之，如果一个 RE 同时包含贪婪和非贪婪的子表达式，那么总的匹配长度要么是尽可能长，要么是尽可能短，这取决于给整个 RE 赋予的属性。给子表达式赋予的属性只影响在这个匹配里，各个子表达式之间相互允许“吃掉”的多少。

量词{1,1}和{1,1}?可以分别用于在一个子表达式 或者整个 RE 上强制贪婪或者非贪婪。当需要整个 RE 具有不同于从其元素中 推导出的贪婪属性时，这很有用。例如，假设我们尝试将一个包含一些数字的 字符串分隔成数字以及在它们之前和之后的部分，我们可能会尝试这样做：

```
SELECT regexp_matches('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}
```

这不会有用：第一个.*是贪婪的，因此它会“吃掉” 尽可能多的字符而留下\d+去匹配在最后一个可能位置上的最 后一个数字。我们可能会通过让它变成非贪婪来修复：

```
SELECT regexp_matches('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}
```

这也不会有用：因为现在 RE 作为整体来说是非贪婪的，因此它会尽快结束 全部的匹配。我们可以通过强制 RE 整体是贪婪的来得到我们想要的：

```
SELECT regexp_matches('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}
```

独立于 RE 的组件的贪婪性之外控制 RE 的整体贪婪性为处理变长模式提供了 很大的灵活性。

在决定更长或者更短的匹配时，匹配长度是以字符衡量的，而不是排序元素。一个空串会被认为比什么都不匹配长。例如：bb*匹配abbbc的中间三个字符；(week|wee)(night|knights)匹配weeknights的所有十个字符； 而(.*).*匹配 abc的时候，圆括号包围的子表达式匹配所有三个字符；当(a*)*被拿来匹配bc时，整个 RE 和圆括号 子表达式都匹配一个空串。

如果声明了大小写无关的匹配，那么效果就好像所有大小写区别在字母表中消失了。如果在多个情况中一个字母以一个普通字符的形式出现在方括号表达式外面，那么它实际上被转换成 一个包含大小写的方括号表达式，也就是说，x 变成 [xX]。 如果它出现在一个方括号表达式里面，那么它的所有大小写的同族都被加入 方括号表达式中，也就是说，x变成[xX]。当它出现在一个方括号表达式内时，它的所有大小写副本都被加入到方括号表达式中，例如， [x]会变成[xX]，而[^x]会变成[^xX]。

如果指定了新行敏感的匹配，.和使用^的方括号表达式 将永远不会匹配新行字符（这样，匹配就绝对不会跨越新行，除非 RE 显式地安排了这样的情况）并且^和$除了分别匹配串开头和结尾之外，还将分别匹配新行后面和前面的空串。但是 ARE 逃逸\A和\Z仍然只匹配串的开头和结尾。

如果指定了部分新行敏感的匹配，那么它影响.和方括号表达式， 这个时候和新行敏感的匹配一样，但是不影响^和$。

如果指定了逆新行敏感匹配，那么它影响^和$，其作用和在新行敏感的匹配里一样，但是不影响.和方括号表达式。这个并不是很有用，只是为了满足对称性而提供的。

9.7.3.6. 限制和兼容性

在这个实现里，对 RE 的长度没有特别的限制。但是，那些希望高移植性的程序应该避免使用长度超过 256 字节的 RE，因为 POSIX 兼容 的实现可以拒绝接受这样的 RE。

ARE 实际上和 POSIX ERE 不兼容的唯一的特性是在方括号表达式里\并不失去它特殊的含义。所有其它 ARE 特性都使用在 POSIX ERE 里面是非法或者是未定义、未声明效果的语法；指示器的***就是在 POSIX 的 BRE 和 ERE 之外的语法。

许多 ARE 扩展都是从 Perl 那里借来的（但是有些被做了修改来清理它们），以及一些 Perl 里没有出现的扩展。要注意的不兼容性包括\b、\B、对结尾的新行缺乏特别的处理、对那些被新行敏感匹配的东西附加的补齐方括号表达式、在 lookahead/lookbehind 约束里对圆括号和后引用的限制以及最长/最短 匹配（而不是第一匹配）的语义。

PostgreSQL 7.4 之前的版本中识别的 ARE 和 ERE 语法存在两个非常明显的不兼容：

- 在 ARE 中，后面跟着一个字母数字字符的\要么是一个逃逸要么是一个错误， 但是在以前的版本里，它只是写该字母数字字符的另外一种方法。这个应该不是什么问题， 因为在以前的版本里没有什么理由会让我们写这样的序列。
- 在 ARE 里，\在[]里还是一个特殊字符， 因此在方括号表达式里的一个文本\必须被写成\\。

9.7.3.7. 基本正则表达式

BRE 在几个方面和 ERE 不太一样。在 BRE 中，|、+和?都是普通字符并且没有与它们功能等价的东西。范围的定界符是\{和\}， 因为 {和}本身是普通字符。嵌套的子表达式的圆括号是\(和\)，因为(和)自身是普通字符。除非在 RE 开头或者是圆括号子表达式开头，^都是一个普通字符。 除非在 RE 结尾或者是圆括号子表达式的结尾，$是一个普通字符。如果*出现在 RE 开头或者是圆括号封装的子表达式开头 （前面可能有^），那么它是个普通字符。最后，可以用单数字的后引用，\<和\>分别是[[:<:]]和[[:>:]]的同义词；在 BRE 中没有其它可用的逃逸。

9.8. 数据类型格式化函数

PostgreSQL格式化函数提供一套强大的工具用于把各种数据类型 （日期/时间、整数、浮点、数字） 转换成格式化的字符串以及反过来从格式化的字符串转换成 指定的数据类型。表 9.23列出了这些函数。这些函数都遵循一个公共的调用规范： 第一个参数是待格式化的值，而第二个是一个定义输出或输入格式的模板。

表 9.23. 格式化函数

函数	返回类型	描述	例子
to_char(timestamp, text)	text	把时间戳转成字符串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	把间隔转成字符串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	把整数转成字符串	to_char(125, '999')
to_char(double precision, text)	text	把实数或双精度转成字符串	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	把数字转成字符串	to_char(-125.8, '999D99S')
to_date(text, text)	date	把字符串转成日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	把字符串转成数字	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	把字符串转成时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

> **注意**
>
> 还有一个单一参数的to_timestamp函数，请见表 9.30

> **提示**
>
> to_timestamp和to_date存在的目的是为了处理无法用简单造型转换的输入格式。对于大部分标准的日期/时间格式，简单地把源字符串造型成所需的数据类型是可以的，并且简单很多。类似地，对于标准的数字表示形式，to_number也是没有必要的。

在一个to_char输出模板串中，一些特定的模式可以被识别并且被替换成基于给定值的被恰当地格式化的数据。任何不属于模板模式的文本都简单地照字面拷贝。同样，在一个输入模板串里（对其他函数），模板模式标识由输入数据串提供的值。如果在模板字符串中有不

是模板模式的字符，输入数据字符串中的对应字符会被简单地跳过（不管它们是否等于模板字符串字符）。

表 9.2展示了可以用于格式化日期和时间值的模版。

表 9.24. 用于日期/时间格式化的模板模式

模式	描述
HH	一天中的小时 （01-12）
HH12	一天中的小时 （01-12）
HH24	一天中的小时 （00-23）
MI	分钟 （00-59）minute (00-59)
SS	秒（00-59）
MS	毫秒（000-999）
US	微秒（000000-999999）
SSSS	午夜后的秒（0-86399）
AM, am, PM or pm	正午指示器（不带句号）
A.M., a.m., P.M. or p.m.	正午指示器（带句号）
Y,YYY	带逗号的年（4 位或者更多位）
YYYY	年（4 位或者更多位）
YYY	年的后三位
YY	年的后两位
Y	年的最后一位
IYYY	ISO 8601 周编号方式的年（4 位或更多位）
IYY	ISO 8601 周编号方式的年的最后 3 位
IY	ISO 8601 周编号方式的年的最后 2 位
I	ISO 8601 周编号方式的年的最后一位
BC, bc, AD或者ad	纪元指示器（不带句号）
B.C., b.c., A.D.或者a.d.	纪元指示器（带句号）
MONTH	全大写形式的月名（空格补齐到 9 字符）
Month	全首字母大写形式的月名（空格补齐到 9 字符）
month	全小写形式的月名（空格补齐到 9 字符）
MON	简写的大写形式的月名（英文 3 字符，本地化长度可变）
Mon	简写的首字母大写形式的月名（英文 3 字符，本地化长度可变）
mon	简写的小写形式的月名（英文 3 字符，本地化长度可变）
MM	月编号（01-12）
DAY	全大写形式的日名（空格补齐到 9 字符）
Day	全首字母大写形式的日名（空格补齐到 9 字符）
day	全小写形式的日名（空格补齐到 9 字符）
DY	简写的大写形式的日名（英语 3 字符，本地化长度可变）

模式	描述
Dy	简写的首字母大写形式的日名（英语 3 字符，本地化长度可变）
dy	简写的小写形式的日名（英语 3 字符，本地化长度可变）
DDD	一年中的日（001-366）
IDDD	ISO 8601 周编号方式的年中的日（001-371，年的第 1 日时第一个 ISO 周的周一）
DD	月中的日（01-31）
D	周中的日，周日（1）到周六（7）
ID	周中的 ISO 8601 日，周一（1）到周日（7）
W	月中的周（1-5）（第一周从该月的第一天开始）
WW	年中的周数（1-53）（第一周从该年的第一天开始）
IW	ISO 8601 周编号方式的年中的周数（01 - 53；新的一年的第一个周四在第一周）
CC	世纪（2 位数）（21 世纪开始于 2001-01-01）
J	儒略日（从午夜 UTC 的公元前 4714 年 11 月 24 日开始的整数日数）
Q	季度（to_date和to_timestamp会忽略）
RM	大写形式的罗马计数法的月（I-XII；I 是 一月）
rm	小写形式的罗马计数法的月（i-xii；i 是 一月）
TZ	大写形式的时区缩写（仅在to_char中支持）
tz	小写形式的时区缩写（仅在to_char中支持）
TZH	时区的小时
TZM	时区的分钟
OF	从UTC开始的时区偏移（仅在to_char中支持）

修饰语可以被应用于模板模式来修改它们的行为。例如，FMMonth就是带着FM修饰语的Month模式。表 9.25展示了可用于日期/时间格式化的修饰语模式。

表 9.25. 用于日期/时间格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式（抑制前导零和填充的空格）	FMMonth
TH suffix	大写形式的序数后缀	DDTH, e.g., 12TH
th suffix	小写形式的序数后缀	DDth, e.g., 12th
FX prefix	固定的格式化全局选项（见使用须知）	FX Month DD Day
TM prefix	翻译模式（基于lc_time打印本地化的日和月名）	TMMonth
SP suffix	拼写模式（未实现）	DDSP

日期/时间格式化的使用须知：

- FM抑制前导的零或尾随的空白，否则会把它们增加到输入从而把一个模式的输出变成固定宽度。在PostgreSQL中，FM只修改下一个声明，而在 Oracle 中，FM影响所有随后的声明，并且重复的FM修饰语将触发填充模式开和关。

- TM不包括结尾空白。to_timestamp和to_date会忽略TM修饰语。

- 如果没有使用FX选项，to_timestamp和to_date会跳过输入字符串中的多个空白。例如，to_timestamp('2000 JUN', 'YYYY MON')是正确的，但to_timestamp('2000 JUN', 'FXYYYY MON')会返回一个错误，因为to_timestamp只期望一个空白。FX必须被指定为模板中的第一个项。

- 在to_char模板里可以有普通文本，并且它们会被照字面输出。你可以把一个子串放到双引号里强迫它被解释成一个文本，即使它里面包含模板模式也如此。例如，在 '"Hello Year "YYYY'中，YYYY将被年份数据代替，但是Year中单独的Y不会。在to_date、to_number以及to_timestamp中，文本和双引号字符串会导致跳过该字符串中所包含的字符数量，例如"XX"会跳过两个输入字符（不管它们是不是XX）。

- 如果你想在输出里有双引号，那么你必须在它们前面放反斜线，例如 '\"YYYY Month\"'。不然，在双引号字符串外面的反斜线就不是特殊的。在双引号字符串内，反斜线会导致下一个字符被取其字面形式，不管它是什么字符（但是这没有特殊效果，除非下一个字符是一个双引号或者另一个反斜线）。

- 在to_timestamp和to_date中，如果年份格式声明少于四位（如YYY）并且提供的年份少于四位，年份将被调整为最接近于 2020 年，例如95会变成 1995。

- 在to_timestamp和to_date中，在处理超过4位数的年份时，YYYY转换具有限制。你必须在YYYY后面使用一些非数字字符或者模板，否则年份总是被解释为 4 位数字。例如（对于 20000 年）：to_date('200001131', 'YYYYMMDD')将会被解释成一个 4 位数字的年份，而不是在年份后使用一个非数字分隔符，像to_date('20000-1131', 'YYYY-MMDD')或to_date('20000Nov31', 'YYYYMonDD')。

- 在to_timestamp和to_date中，CC（世纪）字段会被接受，但是如果有YYY、YYYY或者Y,YYY字段则会忽略它。如果CC与YY或Y一起使用，则结果被计算为指定世纪中的那一年。如果指定了世纪但是没有指定年，则会假定为该世纪的第一年。

- 在to_timestamp和to_date中，工作日名称或编号（DAY、D以及相关的字段类型）会被接受，但会为了计算结果的目的而忽略。季度（Q）字段也是一样。

- 在to_timestamp和to_date中，一个 ISO 8601 周编号的日期（与一个格里高利日期相区别）可以用两种方法之一被指定为to_timestamp和to_date：

  - 年、周编号和工作日：例如to_date('2006-42-4', 'IYYY-IW-ID')返回日期2006-10-19。如果你忽略工作日，它被假定为 1（周一）。

  - 年和一年中的日：例如to_date('2006-291', 'IYYY-IDDD')也返回2006-10-19。

  尝试使用一个混合了 ISO 8601 周编号和格里高利日期的域来输入一个日期是无意义的，并且将导致一个错误。在一个 ISO 周编号的年的环境下，一个“月”或“月中的日”的概念没有意义。在一个格里高利年的环境下，ISO 周没有意义。用户应当避免混合格里高利和 ISO 日期声明。

> **小心**
>
> 虽然to_date将会拒绝混合使用格里高利和 ISO 周编号日期的域，to_char却不会，因为YYYY-MM-DD (IYYY-IDDD)这种输出格式也会有用。但是避免写类似IYYY-MM-DD的东西，那会得到在起始年附近令人惊讶的结果（详见第 9.9.1 节。

- 在to_timestamp中，毫秒（MS）和微秒（US）域都被用作小数点后的秒位。例如to_timestamp('12.3', 'SS.MS')不是 3 毫秒， 而是 300，因为该转换把它看做 12 + 0.3 秒。这意味着对于格式SS.MS而言，输入值12.3、12.30和12.300指定了相同数目的毫秒。要得到三毫秒，你必须使用 12.003，转换会把它看做 12 + 0.003 = 12.003 秒。

  下面是一个更复杂的例子：to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')是 15 小时、12 分钟和 2 秒 + 20 毫秒 + 1230微秒 = 2.021230 秒。

- to_char(..., 'ID')的一周中日的编号匹配extract(isodow from ...)函数，但是to_char(..., 'D')不匹配extract(dow from ...)的日编号。

- to_char(interval)格式化HH和HH12为显示在一个 12 小时的时钟上，即零小时和 36 小时输出为12，而HH24会输出完整的小时值，对于间隔它可以超过 23.

表 9.26展示了可以用于格式化数字值的模版模式。

表 9.26. 用于数字格式化的模板模式

模式	描述
9	数位（如果无意义可以被删除）
0	数位（即便没有意义也不会被删除）
. (period)	小数点
, (comma)	分组（千）分隔符
PR	尖括号内的负值
S	带符号的数字（使用区域）
L	货币符号（使用区域）
D	小数点（使用区域）
G	分组分隔符（使用区域）
MI	在指定位置的负号（如果数字 < 0）
PL	在指定位置的正号（如果数字 > 0）
SG	在指定位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH or th	序数后缀
V	移动指定位数（参阅注解）
EEEE	科学记数的指数

数字格式化的用法须知：

- 0指定一个总是被打印的数位，即便它包含前导/拖尾的零。9也指定一个数位，但是如果它是前导零则会被空格替换，而如果是拖尾零并且指定了填充模式则它会被删除（对于to_number()来说，这两种模式字符等效）。

- 模式字符S、L、D以及G表示当前locale定义的负号、货币符号、小数点以及数字分隔符字符（见lc_monetary和lc_numeric）。不管locale是什么，模式字符句号和逗号就表示小数点和数字分隔符。

- 对于to_char()的模式中的一个负号，如果没有明确的规定，将为该负号保留一列，并且它将被锚接到（出现在左边）那个数字。如果S正好出现在某个9的左边，它也将被锚接到那个数字。

- 使用SG、PL或MI格式化的符号并不挂在数字上面； 例如，to_char(-12, 'MI9999')生成'- 12'而to_char(-12, 'S9999')生成 ' -12'（Oracle 里的实现不允许在9前面使用MI，而是要求9在MI前面。）

- TH不会转换小于零的数值，也不会转换小数。

- PL、SG和TH是PostgreSQL扩展。
- 在to_number中，如果没有使用L或TH之类的非数据模板模式，相应数量的输入字符会被跳过，不管它们是否匹配模板模式，除非它们是数据字符（也就是数位、负号、小数点或者逗号）。例如，TH会跳过两个非数据字符。
- 带有to_char的V会把输入值乘上10^n，其中n是跟在V后面的位数。带有to_number的V以类似的方式做除法。to_char和to_number不支持使用结合小数点的V（例如，不允许99.9V99）。
- EEEE（科学记数法）不能和任何其他格式化模式或修饰语（数字和小数点模式除外）组合在一起使用，并且必须位于格式化字符串的最后（例如9.99EEEE是一个合法的模式）。

某些修饰语可以被应用到任何模板来改变其行为。例如，FM99.99是带有FM修饰语的99.99模式。表 9.27中展示了用于数字格式化模式修饰语。

表 9.27. 用于数字格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式（抑制拖尾零和填充的空白）	FM99.99
TH suffix	大写序数后缀	999TH
th suffix	小写序数后缀	999th

表 9.28展示了一些使用to_char函数的例子。

表 9.28. to_char例子

表达式	结果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'

表达式	结果
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. 时间/日期函数和操作符

表 9.30 展示了可用于处理日期/时间值的函数，其细节在随后的小节中描述。表 9.29 演示了基本算术操作符 （+、*等）的行为。 而与格式化相关的函数，可以参考第 9.8 节你应该很熟悉第 8.5 节的日期/时间数据类型的背景知识。

所有下文描述的接受time或timestamp输入的函数和操作符实际上都有两种变体： 一种接收time with time zone或timestamp with time zone， 另外一种接受time without time zone或者 timestamp without time zone。为了简化，这些变种没有被独立地展示。此外，+和*操作符都是可交换的操作符对（例如，date + integer 和 integer + date）；我们只显示其中一个。

表 9.29. 日期/时间操作符

操作符	例子	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'

操作符	例子	结果
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

表 9.30. 日期/时间函数

函数	返回类型	描述	例子	结果
age(timestamp, timestamp)	interval	减去参数，生成一个使用年、月（而不是只用日）的“符号化”的结果	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 年 9 月 27 日
age(timestamp)	interval	从current_date（在午夜）减去	age(timestamp '1957-06-13')	43 years 8 mons 3 days
clock_timestamp()	timestamp with time zone	当前日期和时间（在语句执行期间变化）；见第 9.9.4 节		
current_date	date	当前日期；见第 9.9.4 节		
current_time	time with time zone	当前时间（一天中的时间）；见第 9.9.4 节		
current_timestamp	timestamp with time zone	当前日期和时间（当前事务开始时）；见第 9.9.4 节		

函数	返回类型	描述	例子	结果
date_part(text, timestamp)	double precision	获得子域（等价于extract）；见第 9.9.1 节	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	获得子域（等价于extract）；见第 9.9.1 节	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	截断到指定精度；另见第 9.9.2 节	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
date_trunc(text, interval)	interval	截断到指定精度；另见第 9.9.2 节	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00
extract(field from timestamp)	double precision	获得子域；见第 9.9.1 节	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double precision	获得子域；见第 9.9.1 节	extract(month from interval '2 years 3 months')	3
isfinite(date)	boolean	测试有限日期（不是+/-无限）	isfinite(date '2001-02-16')	true
isfinite(timestamp)	boolean	测试有限时间戳（不是+/-无限）	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	测试有限间隔	isfinite(interval '4 hours')	true
justify_days(interval)	interval	调整间隔这样30天时间周期可以表示为月	justify_days(interval '35 days')	1 mon 5 days
justify_hours(interval)	interval	调整间隔这样24小时时间周期可以表示为日	justify_hours(interval '27 hours')	1 day 03:00:00
justify_interval(interval)	interval	使用justify_days和justify_hours调整间隔，使用额外的符号调整	justify_interval(interval '1 mon -1 hour')	29 days 23:00:00
localtime	time	当前时间（一天中的时间）；见第 9.9.4 节		
localtimestamp	timestamp	当前日期和时间（当前事务的开始）；见第 9.9.4 节		
make_date(year	date	从年、月、日域创建日期	make_date(2013, 7, 15)	2013-07-15

函数	返回类型	描述	例子	结果
int, month int, day int)				
make_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)	interval	从年、月、周、日、时、分、秒域创建 interval	make_interval(days => 10)	10 days
make_time(hour int, min int, sec double precision)	time	从时、分、秒域创建时间	make_time(8, 15, 23.5)	08:15:23.5
make_timestamp(year int, month int, day int, hour int, min int, sec double precision)	timestamp	从年、月、日、时、分、秒域创建时间戳	make_timestamp(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5
make_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [timezone text])	timestamp with time zone	从年、月、日、时、分、秒域创建带时区的时间戳。如果没有指定timezone， 则使用当前时区。	make_timestamptz(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5+01
now()	timestamp with time zone	当前日期和时间（当前事务的开始）；见第 9.9.4 节		
statement_timestamp()	timestamp with time zone	当前日期和时间（当前事务的开始）；见第 9.9.4 节		
timeofday()	text	当前日期和时间（像clock_timestamp，但是作为一个text字符串）；见第 9.9.4 节		
transaction_timestamp()	timestamp with time zone	当前日期和时间（当前事务的开		

函数	返回类型	描述	例子	结果
		始）； 见第 9.9.4 节		
to_timestamp(double precision)	timestamp with time zone	把 Unix 时间（从 1970-01-01 00:00:00+00 开始的秒）转换成 timestamp	to_timestamp(1284352323)	2010-09-13 04:32:03+00

除了这些函数以外，还支持 SQL 操作符OVERLAPS：

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

这个表达式在两个时间域（用它们的端点定义）重叠的时候得到真，当它们不重叠时得到假。端点可以用一对日期、时间或者时间戳来指定；或者是用一个后面跟着一个间隔的日期、时间或时间戳来指定。当一对值被提供时，起点或终点都可以被写在前面，OVERLAPS会自动地把较早的值作为起点。每一个时间段被认为是表示半开的间隔start <= time < end，除非start和end相等，这种情况下它表示单个时间实例。例如这表示两个只有一个共同端点的时间段不重叠。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果：true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果：false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
结果：false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
结果：true
```

当把一个interval值添加到timestamp with time zone上（或从中减去）时， days 部分会按照指定的天数增加或减少timestamp with time zone的日期。 对于横跨夏令时的变化（当会话的时区被设置为可识别DST的时区时），这意味着interval '1 day'并 不一定等于interval '24 hours'。例如，当会话的时区设置为CST7CDT时，timestamp with time zone '2005-04-02 12:00-07' + interval '1 day' 的结果是timestamp with time zone '2005-04-03 12:00-06'，而将interval '24 hours'增加到相同的初始timestamp with time zone的结果 则是timestamp with time zone '2005-04-03 13:00-06'， 因为CST7CDT时区在2005-04-03 02:00有一个夏令时变更。

注意age返回的月数域可能有歧义，因为不同的月份有不同的天数。 PostgreSQL的方法是当计算部分月数时，采用两个日期中较早的月。例如：age('2004-06-01', '2004-04-30')使用4月份得到1 mon 1 day，而用5月分时会得到1 mon 2 days，因为5月有31天，而4月只有30天。

日期和时间戳的减法也可能会很复杂。执行减法的一种概念上很简单的方法是，使用EXTRACT(EPOCH FROM ...)把每个值都转换成秒数，然后执行减法， 这样会得到两个值之间的秒数。这种方法将会适应每个月中天数、 时区改变和夏令时调整。使用“-”操作符的日期或时间 戳减法会返回值之间的天数（24小时）以及时/分/秒，也会做同样的调整。 age函数会返回年、月、日以及时/分/秒，执行按域的减法，然后对 负值域进行调整。下面的查询展示了这些方法的不同。例子中的结果由 timezone = 'US/Eastern'产生，这使得两个使用的日期之间存在着夏令 时的变化：

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Result: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Result: 121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Result: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01
 12:00:00');
Result: 4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

extract函数从日期/时间值中抽取子域，例如年或者小时等。source必须是一个类型timestamp、time或interval的值表达式（类型为date的表达式将被造型为 timestamp，并且因此也可以被同样使用）。field是一个标识符或者字符串，它指定从源值中抽取的域。extract函数返回类型为double precision的值。 下列值是有效的域名字：

century

世纪

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
结果：20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：21
```

第一个世纪从 0001-01-01 00:00:00 AD 开始， 尽管那时候人们还不知道这是第一个世纪。这个定义适用于所有使用格里高利历法的国家。其中没有 0 世纪，我们直接从公元前 1 世纪到公元 1 世纪。 如果你认为这个不合理，那么请把抱怨发给：罗马圣彼得教堂，梵蒂冈，教皇收。

day

对于timestamp值，是（月份）里的日域（1-31）；对于interval值，是日数

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
结果：40
```

decade

年份域除以10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
结果：200
```

dow

一周中的日，从周日（0）到周六（6）

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
结果：5

请注意，extract的一周中的日和to_char(..., 'D')函数不同。

doy

一年的第几天（1 -365/366）

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：47

epoch

对于timestamp with time zone值， 是自 1970-01-01 00:00:00 UTC 以来的秒数（结果可能是负数）； 对于date and timestamp值，是自本地时间 1970-01-01 00:00:00 以来的描述；对于interval值，它是时间间隔的总秒数。

SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40.12-08');
结果：982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
结果：442800

不能用to_timestamp把一个 epoch 值转换回成时间戳：

SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00

hour

小时域（0 - 23）

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
结果：20

isodow

一周中的日，从周一（1）到周日（7）

SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
结果：7

除了周日，这和dow相同。这符合ISO 8601 中一周中的日的编号。

isoyear

日期所落在的ISO 8601 周编号的年（不适用于间隔）

SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
结果：2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
结果：2006

每一个ISO 8601 周编号的年都开始于包含1月4日的那一周的周一，在早的1月或迟的12月中ISO年可能和格里高利年不同。更多信息见week域。

这个域不能用于 PostgreSQL 8.3之前的版本。

microseconds

秒域，包括小数部分，乘以 1,000,000。请注意它包括全部的秒

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
结果：28500000
```

millennium

千年

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
结果：3
```

19xx的年份在第二个千年里。第三个千年从 2001 年 1 月 1 日开始。

milliseconds

秒域，包括小数部分，乘以 1000。请注意它包括完整的秒。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
结果：28500
```

minute

分钟域（0 - 59）

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
结果：38
```

month

对于timestamp值，它是一年里的月份数（1 - 12）； 对于interval值，它是月的数目，然后对 12 取模（0 - 11）

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
结果：2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
结果：3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
结果：1
```

quarter

该天所在的该年的季度（1 - 4）

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
结果：1
```

second

秒域，包括小数部分（0 - 59[1]）

[1]如果操作系统实现了闰秒，则为60

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
结果：40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
结果：28.5

timezone

与 UTC 的时区偏移，以秒记。正数对应 UTC 东边的时区，负数对应 UTC 西边的时区（从技术上来看，PostgreSQL不使用 UTC，因为其中不处理闰秒）。

timezone_hour

时区偏移的小时部分。

timezone_minute

时区偏移的分钟部分。

week

该天在所在的ISO 8601 周编号的年份里是第几周。根据定义， 一年的第一周包含该年的 1月 4 日并且 ISO 周从星期一开始。换句话说，一年的第一个星期四在第一周。

在 ISO 周编号系统中，早的 1 月的日期可能位于前一年的第五十二或者第五十三周，而迟的 12 月的日期可能位于下一年的第一周。例如， 2005-01-01位于 2004 年的第五十三周，并且2006-01-01位于 2005 年的第五十二周，而2012-12-31位于 2013 年的第一周。我们推荐把isoyear域和week一起使用来得到一致的结果。

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
结果：7

year

年份域。要记住这里没有0 AD，所以从AD年里抽取BC年应该小心处理。

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
结果：2001

> **注意**
>
> 当输入值为 +/-Infinity 时，extract对于单调增的域（epoch、julian、year、isoyear、decade、century以及millennium）返回 +/-Infinity。对于其他域返回 NULL。PostgreSQL 9.6 之前的版本对所有输入无穷的情况都返回零。

extract函数主要的用途是做计算性处理。对于用于显示的日期/时间值格式化，参阅第 9.8 节

在传统的Ingres上建模的date_part函数等价于SQL标准函数extract：

date_part('field', source)

请注意这里的field参数必须是一个串值，而不是一个名字。有效的date_part域名和extract相同。

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');

结果：16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
结果：4

9.9.2. date_trunc

date_trunc函数在概念上和用于数字的trunc函数类似。

date_trunc('field', source)

source是类型timestamp或interval的值表达式（类型date和 time的值都分别被自动转换成timestamp或者interval）。field选择对输入值选用什么样的精度进行截断。返回的值是timestamp类型或者所有小于选定的 精度的域都设置为零（或者一，对于日期和月份）的interval。

field的有效值是：

microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium

例子：

SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-01-01 00:00:00

9.9.3. AT TIME ZONE

AT TIME ZONE把时间戳without time zone转换成时间戳with time zone或者反过来，并且把time值转换成不同的时区。表 9.3展示了它的变体。

表 9.31.AT TIME ZONE变体

表达式	返回类型	描述
timestamp without time zone AT TIME ZONE zone	timestamp with time zone	把给定的不带时区的时间戳当作位于指定时区的时间对待
timestamp with time zone AT TIME ZONE zone	timestamp without time zone	把给定的带时区的时间戳转换到新的时区，不带时区指定
time with time zone AT TIME ZONE zone	time with time zone	把给定的带时区的时间转换到新时区

在这些表达式里，我们需要的时区zone可以指定为文本串（例如，'America/Los_Angeles'）或者一个间隔 （例如，INTERVAL '-08:00'）。 在文本情况下，可用的时区名字可以用第 8.5.3 节描述的任何方式指定。

例子（假设本地时区是America/Los_Angeles）：

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40
```

第一个例子给缺少时区的值加上了时区，并且显示了使用当前TimeZone设置的值。第二个例子把带有时区值的时间戳移动到指定的时区，并且返回不带时区的值。这允许存储和显示不同于当前TimeZone设置的值。第三个例子把东京时间转换成芝加哥时间。把time值转换成其他时区会使用当前活跃的时区规则，因为没有提供日期。

函数timezone(zone, timestamp)等效于 SQL 兼容的结构timestamp AT TIME ZONE zone。

9.9.4. 当前日期/时间

PostgreSQL提供了许多返回当前日期和时间的函数。这些 SQL 标准的函数全部都按照当前事务的开始时刻返回值：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIME和CURRENT_TIMESTAMP传递带有时区的值；LOCALTIME和LOCALTIMESTAMP传递的值不带时区。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME和 LOCALTIMESTAMP可以有选择地接受一个精度参数， 该精度导致结果的秒域被园整为指定小数位。如果没有精度参数，结果将被给予所能得到的全部精度。

一些例子：

```
SELECT CURRENT_TIME;
结果：14:39:53.662522-05

SELECT CURRENT_DATE;
结果：2001-12-23

SELECT CURRENT_TIMESTAMP;
结果：2001-12-23 14:39:53.662522-05
```

```
SELECT CURRENT_TIMESTAMP(2);
结果：2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
结果：2001-12-23 14:39:53.662522
```

因为这些函数全部都按照当前事务的开始时刻返回结果，所以它们的值在事务运行的整个期间内都不改变。 我们认为这是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念， 这样在同一个事务里的多个修改可以保持同样的时间戳。

> **注意**
>
> 许多其它数据库系统可能会更频繁地推进这些值。

PostgreSQL同样也提供了返回当前语句开始时间的函数， 它们会返回函数被调用时的真实当前时间。这些非 SQL 标准的函数列表如下：

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

transaction_timestamp()等价于CURRENT_TIMESTAMP，但是其命名清楚地反映了它的返回值。statement_timestamp()返回当前语句的开始时刻（更准确的说是收到 客户端最后一条命令的时间）。statement_timestamp()和transaction_timestamp()在一个事务的第一条命令期间返回值相同，但是在随后的命令中却不一定相同。 clock_timestamp()返回真正的当前时间，因此它的值甚至在同一条 SQL 命令中都会变化。timeofday()是一个有历史原因的PostgreSQL函数。和clock_timestamp()相似，timeofday()也返回真实的当前时间，但是它的结果是一个格式化的text串，而不是timestamp with time zone值。now()是PostgreSQL的一个传统，等效于transaction_timestamp()。

所有日期/时间类型还接受特殊的文字值now，用于指定当前的日期和时间（重申，被解释为当前事务的开始时刻）。 因此，下面三个都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- 对于和 DEFAULT 一起使用是不正确的
```

> **提示**
>
> 在创建表期间指定一个DEFAULT子句时，你不会希望使用第三种形式。系统将在分析这个常量的时候把now转换为一个timestamp， 这样需要默认值时就会得到创建表的时间！而前两种形式要到实际使用缺省值的时候才被计算， 因为它们是函数调用。因此它们可以给出每次插入行的时刻。

9.9.5. 延时执行

下面的这些函数可以用于让服务器进程延时执行：

```
pg_sleep(seconds)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

pg_sleep让当前的会话进程休眠seconds 秒以后再执行。seconds是一个double precision类型的值，所以可以指定带小数的秒数。pg_sleep_for是针对用 interval指定的较长休眠时间的函数。pg_sleep_until 则可以用来休眠到一个指定的时刻唤醒。例如：

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

> **注意**
>
> 有效的休眠时间间隔精度是平台相关的，通常 0.01 秒是通用值。休眠延迟将至少持续指 定的时长， 也有可能由于服务器负荷而比指定的时间长。特别地， pg_sleep_until并不保证能刚好在指定的时刻被唤醒，但它不会 在比指定时刻早的时候醒来。

> **警告**
>
> 请确保在调用pg_sleep或者其变体时，你的会话没有持有不必要 的锁。否则其它会话可能必须等待你的休眠会话，因而减慢整个系统速度。

9.10. 枚举支持函数

对于枚举类型(在第 8.7 节描述)， 有一些函数允许更清洁的编码，而不需要为一个枚举类型硬写特定的值。它们被列在表 9.32中。本例假定一个枚举类型被创建为：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue',
 'purple');
```

表 9.32. 枚举支持函数

函数	描述	例子	例子结果
enum_first(anyenum)	返回输入枚举类型的第一个值	enum_first(null::rainbow)	red
enum_last(anyenum)	返回输入枚举类型的最后一个值	enum_last(null::rainbow)	purple
enum_range(anyenum)	将输入枚举类型的所有值作为一个有序的数组返回	enum_range(null::rainbow)	{red,orange,yellow,green,blue,purple}
enum_range(anyenum, anyenum)	以一个数组返回在给定两个枚举值之间的范围。值必须来自相同的枚举类型。 如果第一个参数为空，其结果将从枚举类型的第一个值开始。如果第二参数为空，其结果将以枚举类型的最后一个值结束。	enum_range('orange'::rainbow, 'green'::rainbow)	{orange,yellow,green}
		enum_range(NULL, 'green'::rainbow)	{red,orange,yellow,green}
		enum_range('orange'::rainbow, NULL)	{orange,yellow,green,blue,purple}

请注意，除了双参数形式的enum_range外， 这些函数忽略传递给它们的具体值，它们只关心声明的数据类型。 空值或类型的一个特定值可以通过，并得到相同的结果。这些函数更多地被用于一个表列或函数参数，而不是一个硬写的类型名，如例子中所建议。

9.11. 几何函数和操作符

几何类型point、box、 lseg、line、path、 polygon和circle有一大堆本地支持函数和操作符，如表 9.33、表 9.34和表 9.35中所示。

> **小心**
>
> 请注意“same as”操作符（~=），表示point、box、polygon和circle类型的一般相等概念。这些类型中的某些还有一个=操作符，但是=只比较相同的面积。其它的标量比较操作符 （<=等等）也是为这些类型比较面积。

表 9.33. 几何操作符

操作符	描述	例子
+	平移	box '((0,0),(1,1))' + point '(2.0,0)'
-	平移	box '((0,0),(1,1))' - point '(2.0,0)'
*	缩放/旋转	box '((0,0),(1,1))' * point '(2.0,0)'
/	缩放/旋转	box '((0,0),(2,2))' / point '(2.0,0)'
#	相交的点或方框	box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'
#	路径或多边形中的点数	# path '((1,0),(0,1),(-1,0))'
@-@	长度或周长	@-@ path '((0,0),(1,0))'
@@	中心	@@ circle '((0,0),10)'
##	第二个操作数上最接近第一个操作数的点	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	距离	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	是否重叠？（只要有一个公共点这就为真）	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	是否严格地在左侧？	circle '((0,0),1)' << circle '((5,0),1)'
>>	是否严格地在右侧？	circle '((5,0),1)' >> circle '((0,0),1)'
&<	没有延展到右边？	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	没有延展到左边？	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	严格在下？	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	严格在上？	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	没有延展到上面？	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'

操作符	描述	例子
|&>	没有延展到下面？	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	在下面（允许相切）？	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	在上面（允许相切）？	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	相交？	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	水平？	?- lseg '((-1,0),(1,0))'
?-	水平对齐？	point '(1,0)' ?- point '(0,0)'
?|	垂直？	?| lseg '((-1,0),(1,0))'
?|	垂直对齐？	point '(0,1)' ?| point '(0,0)'
?-|	相互垂直？	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	平行？	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	包含？	circle '((0,0),2)' @> point '(1,1)'
<@	包含在内或在上？	point '(1,1)' <@ circle '((0,0),2)'
~=	相同？	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

> **注意**
>
> 在PostgreSQL之前，包含操作符@>和<@被分别称为~和@。 这些名字仍然可以使用，但是已被废除并且最终将被移除。

表 9.34. 几何函数

函数	返回类型	描述	例子
area(object)	double precision	面积	area(box '((0,0),(1,1))')
center(object)	point	中心	center(box '((0,0),(1,2))')
diameter(circle)	double precision	圆的直径	diameter(circle '((0,0),2.0)')
height(box)	double precision	方框的垂直尺寸	height(box '((0,0),(1,1))')
isclosed(path)	boolean	一个封闭路径？	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	一个开放路径？	isopen(path '[(0,0),(1,1),(2,0)]')

函数	返回类型	描述	例子
length(object)	double precision	长度	length(path '((-1,0),(1,0))')
npoints(path)	int	点数	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	点数	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	将路径转换成封闭的	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	将路径转换成开放	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	圆的半径	radius(circle '((0,0),2.0)')
width(box)	double precision	方框的水平尺寸	width(box '((0,0),(1,1))')

表 9.35. 几何类型转换函数

函数	返回类型	描述	例子
box(circle)	box	圆到方框	box(circle '((0,0),2.0)')
box(point)	box	点到空方框	box(point '(0,0)')
box(point, point)	box	点到方框	box(point '(0,0)', point '(1,1)')
box(polygon)	box	多边形到方框	box(polygon '((0,0),(1,1),(2,0))')
bound_box(box, box)	box	方框到外包框	bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))')
circle(box)	circle	方框到圆	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	中心和半径到圆	circle(point '(0,0)', 2.0)
circle(polygon)	circle	多边形到圆	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	点到线	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	方框对角线到线段	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	点到线段	lseg(point '(-1,0)', point '(1,0)')

函数	返回类型	描述	例子
path(polygon)	path	多边形到路径	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	构造点	point(23.4, -44.5)
point(box)	point	方框的中心	point(box '((-1,0),(1,0))')
point(circle)	point	圆的中心	point(circle '((0,0),2.0)')
point(lseg)	point	线段的中心	point(lseg '((-1,0),(1,0))')
point(polygon)	point	多边形的中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	方框到4点多边形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	圆到12点多边形	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	点到npts点多边形	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	路径到多边形	polygon(path '((0,0),(1,1),(2,0))')

我们可以把一个point的两个组成数字当作具有索引 0 和 1 的数组访问。例如，如果t.p是一个point列，那么SELECT p[0] FROM t检索 X 座标而 UPDATE t SET p[1] = ...改变 Y 座标。同样，box或者lseg类型的值可以当作两个point值的数组值看待。

函数area可以用于类型box、circle和path。area函数操作path数据类型的时候， 只有在path的点没有交叉的情况下才可用。例如，path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH是不行的， 而下面的视觉上相同的 path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH就可以。 如果交叉和不交叉的path概念让你疑惑，那么把上面两个path都画在一张图纸上，你就明白了。

9.12. 网络地址函数和操作符

表 9.36展示了可以用于cidr和 inet类型的操作符。 操作符<<、<<=、 >>、>>=和 &&测试用于子网包含。它们只考虑两个地址的网 络部分（忽略任何主机部分），然后判断其中一个网络部分是等于另外一个或者是 另外一个的子网。

表 9.36.cidr和inet操作符

操作符	描述	例子
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于等于	inet '192.168.1.5' >= inet '192.168.1.5'

操作符	描述	例子
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	被包含在内	inet '192.168.1.5' << inet '192.168.1/24'
<<=	被包含在内或等于	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	包含	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	包含或者被包含contains or is contained by	inet '192.168.1/24' && inet '192.168.1.80/28'
~	按位 NOT	~ inet '192.168.1.6'
&	按位 AND	inet '192.168.1.6' & inet '0.0.0.255'
|	按位 OR	inet '192.168.1.6' | inet '0.0.0.255'
+	加	inet '192.168.1.6' + 25
-	减	inet '192.168.1.43' - 36
-	减	inet '192.168.1.43' - inet '192.168.1.19'

表 9.37展示了所有可以用于cidr和inet类型的函数。函数abbrev、host和text主要是为了提供可选的显示格式用的。

表 9.37.cidr和inet函数

函数	返回类型	描述	例子	结果
abbrev(inet)	text	缩写显示格式文本	abbrev(inet '10.1.0.0/16')	10.1.0.0/16
abbrev(cidr)	text	缩写显示格式文本	abbrev(cidr '10.1.0.0/16')	10.1/16
broadcast(inet)	inet	网络广播地址	broadcast('192.168.1.5/24')	192.168.1.255/24
family(inet)	int	抽取地址族；4为IPv4， 6为 IPv6	family('::1')	6
host(inet)	text	抽取 IP 地址为文本	host('192.168.1.5/24')	192.168.1.5
hostmask(inet)	inet	为网络构造主机掩码	hostmask('192.168.23.20/30')	0.0.0.3
masklen(inet)	int	抽取网络掩码长度	masklen('192.168.1.5/24')	24
netmask(inet)	inet	为网络构造网络掩码	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	抽取地址的网络部分	network('192.168.1.5/24')	192.168.1.0/24

函数	返回类型	描述	例子	结果
set_masklen(inet, int)	inet	为inet值设置网络掩码长度	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	为cidr值设置网络掩码长度	set_masklen('192.168.0.0/24'::cidr, 16)	192.168.0.0/16
text(inet)	text	抽取 IP 地址和网络掩码长度为文本	text(inet '192.168.1.5')	192.168.1.5/32
inet_same_family(inet, inet)	boolean	地址是来自于同一个家族吗?	inet_same_family('192.168.1.5/24', '::1')	f
inet_merge(inet, inet)	cidr	包括给定网络的最小网络	inet_merge('192.168.1.5/24', '192.168.2.5/24')	192.168.0.0/22

任何cidr值都能够被隐式或显式地转换为inet值， 因此上述能够操作inet值的函数也同样能够操作cidr值（也有独立的用于inet和cidr的函数，因为它的行为应当和这两种情况不同）。inet值也可以转换为cidr值。完成时，该网络掩码右侧的任何位都将无声地转换为零以获得一个有效的cidr值。另外，你还可以使用常规的造型语法将一个文本字符串转换为inet或cidr值：例如，inet(expression)或colname::cidr。

表 9.38展示了可以用于macaddr类型的函数。 函数trunc(macaddr)返回一个 MAC 地址，该地址的最后三个字节设置为零。这样可以把剩下的前缀与一个制造商相关联。

表 9.38. macaddr函数

函数	返回类型	描述	例子	结果
trunc(macaddr)	macaddr	设置最后3个字节为零	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

macaddr类型还支持标准关系操作符 （>、<=等） 用于编辑次序，并且按位算术操作符（~、&和|）用于 NOT、AND 和 OR。

表 9.39中展示了可以用于macaddr8类型的函数。函数trunc(macaddr8)返回一个后五个字节设置为零的MAC地址。这可以被用来为一个制造商关联一个前缀。

表 9.39. macaddr8函数

函数	返回类型	描述	例子	结果
trunc(macaddr8)	macaddr8	设置最后五个字节为零	trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')	12:34:56:00:00:00:00:00
macaddr8_set7bit(macaddr8)	macaddr8	设置第7位为一，也被称为修改版的EUI-64，用于内含在IPv6地址中	macaddr8_set7bit('00:34:56:ab:cd:ef')	02:34:56:ff:fe:ab:cd:ef

macaddr8类型也支持用于排序的标准关系操作符（>、<=等）以及用于NOT、AND和OR的位运算操作符（~、&和|）。

9.13. 文本搜索函数和操作符

表 9.40、 表 9.41和 表 9.42总结了为全文搜索提供的函数和操作符。PostgreSQL的文本搜索功能的详细解释可参考第 12 章

表 9.40. 文本搜索操作符

操作符	返回类型	描述	例子	结果
@@	boolean	tsvector匹配tsquery吗?	to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')	t
@@@	boolean	@@的已废弃同义词	to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')	t
||	tsvector	连接tsvector	'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector	'a':1 'b':2,5 'c':3 'd':4
&&	tsquery	将tsquery用 AND 连接起来	'fat | rat'::tsquery && 'cat'::tsquery	('fat' | 'rat') & 'cat'
||	tsquery	将tsquery用 OR 连接起来	'fat | rat'::tsquery || 'cat'::tsquery	('fat' | 'rat') | 'cat'
!!	tsquery	对一个tsquery取反	!! 'cat'::tsquery	!'cat'
<->	tsquery	tsquery后面跟着tsquery	to_tsquery('fat') <-> to_tsquery('rat')	'fat' <-> 'rat'
@>	boolean	tsquery包含另一个?	'cat'::tsquery @> 'cat & rat'::tsquery	f
<@	boolean	tsquery被包含?	'cat'::tsquery <@ 'cat & rat'::tsquery	t

> **注意**
>
> tsquery的包含操作符只考虑两个查询中的词位，而忽略组合操作符。

除了显示在表中的操作符，还定义了tsvector和tsquery类型的普通B-tree比较操作符（=、<等）。它们对于文本搜索不是很有用，但是允许使用。例如，建在这些类型列上的唯一索引。

表 9.41. 文本搜索函数

函数	返回类型	描述	例子	结果
array_to_tsvector(text[])	tsvector	把词位数组转换成tsvector	array_to_tsvector('{fat,cat,rat}'::text[])	'cat' 'fat' 'rat'
get_current_ts_config()	regconfig	获得默认文本搜索配置	get_current_ts_config()	english

函数	返回类型	描述	例子	结果
length(tsvector)	integer	tsvector中的词位数	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
numnode(tsquery)	integer	tsquery中词位外加操作符的数目	numnode('(fat & rat) | cat'::tsquery)	5
plainto_tsquery([config regconfig ,] query text)	tsquery	产生tsquery但忽略标点符号	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
phraseto_tsquery([config regconfig ,] query text)	tsquery	产生忽略标点搜索短语的tsquery	phraseto_tsquery('english', 'The Fat Rats')	'fat' <-> 'rat'
websearch_to_tsquery([config regconfig ,] query text)	tsquery	从一个Web搜索风格的查询产生tsquery	websearch_to_tsquery('english', '"fat rat" or rat')	'fat' <-> 'rat' | 'rat'
querytree(query tsquery)	text	获得一个tsquery的可索引部分	querytree('foo & ! bar'::tsquery)	'foo'
setweight(vector tsvector, weight "char")	tsvector	为vector的每一个元素分配权重	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
setweight(vector tsvector, weight "char", lexemes text[])	tsvector	为lexemes中列出的vector的元素分配权重	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}')	'cat':3A 'fat':2,4 'rat':5A
strip(tsvector)	tsvector	从tsvector中移除位置和权重	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([config regconfig ,] query text)	tsquery	规范化词并转换成tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
to_tsvector([config regconfig ,] document text)	tsvector	缩减文档文本成tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
to_tsvector([config regconfig ,] document json(b))	tsvector	把该文档中的每个字符串值缩减成一个tsvector，然后将它们按在文档中的顺序串接起来形成一个tsvector	to_tsvector('english', '{"a": "The Fat Rats"}'::json)	'fat':2 'rat':3

函数	返回类型	描述	例子	结果
json(b)_to_tsvector([config regconfig,] document json(b), filter json(b))	tsvector	把filter指定的文档中的每个值缩减为一个tsvector，然后把它们按照文档中的顺序串接起来形成一个tsvector。filter是一个jsonb数组，它列举哪些种类的元素需要被包括在结果tsvector中。filter的可能值是"string"（包括所有字符串值）、"numeric"（包括所有字符串格式的数字值）、"boolean"（包括所有字符串格式"true"/"false"的布尔值）、"key"（包括所有键）或者"all"（包括上述所有）。这些值可以被组合在一起，例如用来包括所有的字符串和数字值。	json_to_tsvector('{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')	'123':5 'fat':2 'rat':3
ts_delete(vector tsvector, lexeme text)	tsvector	从vector中移除给定的lexeme	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')	'cat':3 'rat':5A
ts_delete(vector tsvector, lexemes text[])	tsvector	从vector中移除lexemes中词位的任何出现	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])	'cat':3
ts_filter(vector tsvector, weights "char"[])	tsvector	从vector中只选择带有给定权重的元素	ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}')	'cat':3B 'rat':5A
ts_headline([config regconfig,] document text, query tsquery [, options text])	text	显示一个查询匹配	ts_headline('x y z', 'z'::tsquery)	x y z
ts_headline([config regconfig,]	text	显示一个查询匹配	ts_headline('{"a": "x y z"}'::json, 'z'::tsquery)	{"a": "x y z"}

函数	返回类型	描述	例子	结果
document json(b), query tsquery [, options text])				
ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])	float4	为查询排名文档	ts_rank(textsearch, query)	0.818
ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])	float4	使用覆盖密度为查询排名文档	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(query tsquery, target tsquery, substitute tsquery)	tsquery	在查询内用substitute替换target	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(query tsquery, select text)	tsquery	使用来自一个SELECT的目标和替换者进行替换	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsquery_phrase(query1 tsquery, query2 tsquery)	tsquery	制造搜索后面跟着query2的query1的查询（和<->操作符相同）	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))	'fat' <-> 'cat'
tsquery_phrase(query1 tsquery, query2 tsquery, distance integer)	tsquery	制造查询来搜索在query1后面最大距离distance上跟着query2 的情况	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)	'fat' <10> 'cat'
tsvector_to_array(tsvector)	text[]	把tsvector转换为词位数组	tsvector_to_array('a:1 cat:3 rat:5A'::tsvector)	{a,cat,rat}
tsvector_update_trigger()	trigger	用于自动tsvector列更新的触发器函数	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)	
tsvector_update_trigger_column()	trigger	用于自动tsvector列更新的触发器函数	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)	

函数	返回类型	描述	例子	结果
unnest(tsvector, OUT lexeme text, OUT positions smallint[], OUT weights text)	setof record	把一个 tsvector 扩展成一组行	unnest('fat:2,4 cat:3 rat:5A'::tsvector)	(cat,{3}, {D}) ...

> **注意**
>
> 所有接受一个可选的regconfig参数的文本搜索函数在该参数被忽略时，使用由default_text_search_config指定的配置。

表 9.42中的函数被单独列出，因为它们通常不被用于日常的文本搜索操作。 它们有助于开发和调试新的文本搜索配置。

表 9.42. 文本搜索调试函数

函数	返回类型	描述	例子	结果
ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])	setof record	测试一个配置	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The, {english_stem},english_stem, {}) ...
ts_lexize(dict regdictionary, token text)	text[]	测试一个字典	ts_lexize('english_stem', 'stars')	{star}
ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)	setof record	测试一个解析器	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)	setof record	测试一个解析器	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type(parser_name text, OUT tokid integer, OUT alias text,	setof record	获得解析器定义的记号类型	ts_token_type('default')	(1,asciiword,"Word, all ASCII") ...

函数	返回类型	描述	例子	结果
OUT description text)				
ts_token_type(parser_oid, OUT tokid integer, OUT alias text, OUT description text)	setof record	获得解析器定义的记号类型	ts_token_type(3722)	(22,asciiword,"Word, all ASCII") ...
ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)	setof record	获得一个tsvector列的统计	ts_stat('SELECT vector from apod')	(foo,10,15) ...

9.14. XML 函数

本节中描述的函数以及类函数的表达式都在类型xml的值上操作。类型xml的详细信息请参见第 8.13 节用于在值和类型xml之间转换的类函数的表达式xmlparse和xmlserialize就不在这里重复介绍。使用大部分这些函数要求安装时使用了configure --with-libxml进行编译。

9.14.1. 产生 XML 内容

有一组函数和类函数的表达式可以用来从 SQL 数据产生 XML 内容。它们特别适合于将查询结果格式化成 XML 文档以便于在客户端应用中处理。

9.14.1.1. xmlcomment

```
xmlcomment(text)
```

函数xmlcomment创建了一个 XML 值，它包含一个使用指定文本作为内容的 XML 注释。该文本不包含“--”或者也不会以一个“-”结尾，这样结果的结构是一个合法的 XML 注释。如果参数为空，结果也为空。

例子：

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.14.1.2. xmlconcat

```
xmlconcat(xml[, ...])
```

函数xmlconcat将由单个 XML 值组成的列表串接成一个单独的值，这个值包含一个 XML 内容片断。空值会被忽略，只有当没有参数为非空时结果才为空。

例子：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

如果 XML 声明存在，它们会按照下面的方式被组合。如果所有的参数值都有相同的 XML 版本声明，该版本将被用在结果中，否则将不使用版本。如果所有参数值有独立声明值“yes”，那么该值将被用在结果中。如果所有参数值都有一个独立声明值并且至少有一个为“no”，则“no”被用在结果中。否则结果中将没有独立声明。如果结果被决定要要求一个独立声明但是没有版本声明，将会使用一个版本 1.0 的版本声明，因为 XML 要求一个 XML 声明要包含一个版本声明。编码声明会被忽略并且在所有情况中都会被移除。

例子：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1"
 standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. xmlelement

```
xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [,
 content, ...])
```

表达式xmlelement使用给定名称、属性和内容产生一个 XML 元素。

例子：

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

不是合法 XML 名字的元素名和属性名将被逃逸，逃逸的方法是将违反的字符用序列_xHHHH_替换，其中HHHH是被替换字符的 Unicode 代码点的十六进制表示。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));
```

```
            xmlelement
---------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用，则不需要指定一个显式的属性名，在这种情况下列的名字将被默认用于属性的名字。在其他情况下，属性必须被给定一个显式名称。因此这个例子是合法的：

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但是下面这些不合法：

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

如果指定了元素内容，它们将被根据其数据类型格式化。如果内容本身也是类型xml，就可以构建复杂的 XML 文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

其他类型的内容将被格式化为合法的 XML 字符数据。这意味着字符 <, >, 和 & 将被转换为实体。二进制数据（数据类型bytea）将被表示成 base64 或十六进制编码，具体取决于配置参数xmlbinary的设置。为了将 SQL 和 PostgreSQL 数据类型和 XML 模式声明对齐，我们期待单独数据类型的特定行为能够改进，到那时将会出现一个更为精确的描述。

9.14.1.4. xmlforest

```
xmlforest(content [AS name] [, ...])
```

表达式xmlforest使用给定名称和内容产生一个元素的 XML 森林（序列）。

例子：

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                        xmlforest
-----------------------------------------------------------------------------------------
```

```
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

如我们在第二个例子中所见，如果内容值是一个列引用，元素名称可以被忽略，这种情况下默认使用列名。否则，必须指定一个名字。

如上文xmlelement所示，非法 XML 名字的元素名会被逃逸。相似地，内容数据也会被逃逸来产生合法的 XML 内容，除非它已经是一个xml类型。

注意如果 XML 森林由多于一个元素组成，那么它不是合法的 XML 文档，因此在xmlelement中包装xmlforest表达式会有用处。

9.14.1.5. xmlpi

```
xmlpi(name target [, content])
```

表达式xmlpi创建一个 XML 处理指令。如果存在内容，内容不能包含字符序列?>。

例子：

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
------------------------------
 <?php echo "hello world";?>
```

9.14.1.6. xmlroot

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

表达式xmlroot修改一个 XML 值的根结点的属性。如果指定了一个版本，它会替换根节点的版本声明中的值；如果指定了一个独立设置，它会替换根节点的独立声明中的值。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. xmlagg

```
xmlagg(xml)
```

和这里描述的其他函数不同，函数xmlagg是一个聚集函数。它将聚集函数调用的输入值串接起来，非常像xmlconcat所做的事情，除了串接是跨行发生的而不是在单一行的多个表达式上发生。聚集表达式的更多信息请见第 9.20 节

例子：

```
CREATE TABLE test (y int, x xml);
```

```
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

为了决定串接的顺序，可以为聚集调用增加一个ORDER BY子句，如第 4.2.7 节所述。例如：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

我们推荐在以前的版本中使用下列非标准方法，并且它们在特定情况下仍然有用：

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.2. XML 谓词

这一节描述的表达式检查xml值的属性。

9.14.2.1. IS DOCUMENT

```
xml IS DOCUMENT
```

如果参数 XML 值是一个正确的 XML 文档，则IS DOCUMENT返回真，如果不是则返回假（即它是一个内容片断），或者是参数为空时返回空。文档和内容片断之间的区别请见第 8.13 节

9.14.2.2. IS NOT DOCUMENT

```
xml IS NOT DOCUMENT
```

如果参数中的XML值是一个正确的XML文档，那么表达式IS NOT DOCUMENT返回假，否则返回真（也就是说它是一个内容片段），如果参数为空则返回空。

9.14.2.3. XMLEXISTS

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

如果第一个参数中的 XPath 表达式返回任何结点，则函数xmlexists返回真，否则返回假（如果哪一个参数为空，则结果就为空）。

例子：

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');
```

```
 xmlexists
------------
 t
(1 row)
```

BY REF子句在 PostgreSQL 中没有作用，但是为了和其他实现保持 SQL 一致性和兼容性还是允许它出现。每一种 SQL 标准，第一个BY REF是被要求的，而第二个则是可选的。也要注意 SQL 标准指定xmlexists结构来将一个 XQuery 表达式作为第一个参数，但 PostgreSQL 目前只支持 XPath，它是 XQuery的一个子集。

9.14.2.4. xml_is_well_formed

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

这些函数检查一个text串是不是一个良构的 XML，返回一个布尔结果。xml_is_well_formed_document检查一个良构的文档，而xml_is_well_formed_content检查良构的内容。如果xmloption配置参数被设置为DOCUMENT，xml_is_well_formed会做第一个函数的工作；如果配置参数被设置为CONTENT，xml_is_well_formed会做第二个函数的工作。这意味着xml_is_well_formed对于检查一个到类型xml的简单造型是否会成功非常有用，而其他两个函数对于检查XMLPARSE的对应变体是否会成功有用。

例子：

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</my:foo>');
 xml_is_well_formed_document
```

```
------------------------------
 f
(1 row)
```

最后一个例子显示了这些检查也包括名字空间是否正确地匹配。

9.14.3. 处理 XML

要处理数据类型xml的值， PostgreSQL 提供了函数xpath和xpath_exists，它们计算 XPath 1.0 表达式以及XMLTABLE表函数。

9.14.3.1. xpath

```
xpath(xpath, xml [, nsarray])
```

函数xpath在 XML 值 xml上计算 XPath 表达式xpath (a text value)。它返回一个 XML 值的数组，该数组对应于该 XPath 表达式产生的结点集合。如果该 XPath 表达式返回一个标量值而不是一个结点集合，将会返回一个单一元素的数组。

第二个参数必须是一个良构的 XML 文档。特殊地，它必须有一个单一根结点元素。

该函数可选的第三个参数是一个名字空间映射的数组。这个数组应该是一个二维text数组，其第二轴长度等于2（即它应该是一个数组的数组，其中每一个都由刚好 2 个元素组成）。每个数组项的第一个元素是名字空间的名称（别名），第二个元素是名字空间的 URI。并不要求在这个数组中提供的别名和在 XML 文档本身中使用的那些名字空间相同（换句话说，在 XML 文档中和在xpath函数环境中，别名都是本地的）。

例子：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

要处理默认（匿名）命名空间，做这样的事情：

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></
a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.14.3.2. xpath_exists

```
xpath_exists(xpath, xml [, nsarray])
```

函数xpath_exists是xpath函数的一种特殊形式。这个函数不是返回满足 XPath 的单一 XML 值，它返回一个布尔值表示查询是否被满足。这个函数等价于标准的XMLEXISTS谓词，不过它还提供了对一个名字空间映射参数的支持。

例子：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</
my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.14.3.3. xmltable

```
xmltable( [XMLNAMESPACES(namespace uri AS namespace name[, ...]), ]
          row_expression PASSING [BY REF] document_expression [BY REF]
          COLUMNS name { type [PATH column_expression]
 [DEFAULT default_expression] [NOT NULL | NULL]
                        | FOR ORDINALITY }
                  [, ...]
)
```

xmltable函数基于给定的XML值产生一个表、一个抽取行的XPath过滤器以及一个可选的列定义集合。

可选的XMLNAMESPACES子句是一个逗号分隔的名字空间列表。它指定文档中使用的XML名字空间极其别名。当前不支持默认的名字空间说明。

必需的row_expression参数是一个XPath表达式，该表达式会根据所提供的XML文档进行计算来得到一个有序的XML节点序列。xmltable会把这个序列转变成输出行。

document_expression提供要操作的XML文档。BY REF子句在PostgreSQL中没有效果，允许它的目的是为了遵守SQL以及与其他实现相兼容。该参数必须是一个结构良好的XML文档，不接受XML片段或者森林。

强制需要的COLUMNS子句指定输出表中的列列表。如果COLUMNS子句被省略，结果集合中的行包含类型为xml的单一列，列中包含匹配row_expression的数据。如果指定了COLUMNS，则每一项描述一个列。格式请见上面的语法综述。列名和类型是必需的，路径、默认值以及为空性子句是可选的。

被标记为FOR ORDINALITY的列将被行号填充，它们匹配输出列出现在原始的输入XML文档中的顺序。最多只能有一个列被标记为FOR ORDINALITY。

一个列的column_expression是一个要针对每行（与row_expression的结果有关）计算的XPath表达式，它用来得到该列的值。如果没有给出column_expression，则把列名用作一种隐式路径。

如果一个列的XPath表达式返回多个元素，则会发生错误。如果该表达式匹配一个空标记，则结果是一个空字符串（不是NULL）。任何xsi:nil属性都会被忽略。

被column_expression匹配上的XML的文本主体被用作该列的值。一个元素中的多个text()节点会按照顺序串接起来。任何子元素、处理指令以及注释都会被忽略，但是子元素的文本内容会被串接到结果中。注意，两个非文本元素之间的仅有空格的text()节点会被保留，并且text()节点上的前导空格不会被平面化。

如果路径表达式不匹配一个给定行但制定有default_expression，则会使用从该表达式计算出的结果值。如果没有对该列给出DEFAULT子句，则该字段将被设置为NULL。default_expression可以引用在列列表中出现在它前面的输出列值，因此一列的默认值可能会基于另一列的值。

列可能会被标记为NOT NULL。如果一个NOT NULL列的column_expression不匹配任何东西并且没有DEFAULT或者default_expression也计算为空，则会报告一个错误。

和常规的PostgreSQL函数不同，column_expression和default_expression在调用前不会被计算为简单值。column_expression通常为每一个输入行计算正好一
次，default_expression则在每当一个字段需要默认值时都会被计算。如果表达式是稳定的或者不变的，则重复计算可以被跳过。实际上xmltable的行为更像一个子查询而不是函数调用。这意味着你可以在default_expression中使用易变函数，并且column_expression可以基于XML文档的其他部分。

例子：

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!
="sq_km"]/@unit)',
                        premier_name text PATH 'PREMIER_NAME' DEFAULT 'not
 specified') ;

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  |
 premier_name
----+------------+--------------+------------+------------+--------------
+---------------
  1 |          1 | Australia    | AU         |            |              | not
 specified
  5 |          2 | Japan        | JP         |            | 145935 sq_mi |
 Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not
 specified
```

接下来的例子展示了多个text()节点的串接、列名用作XPath过滤器的用法以及对空格、XML注释和处理指令的处理：

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </
element>
  </root>
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
       element
----------------------
   Hello2a2   bbbCC
```

下面的例子展示了如何使用XMLNAMESPACES子句指定用在XML文档以及XPath表达式中的名字空间列表：

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
              '/x:example/x:item'
                 PASSING (SELECT data FROM xmldata)
                 COLUMNS foo int PATH '@foo',
                   bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

9.14.4. 将表映射到 XML

下面的函数将会把关系表的内容映射成 XML 值。它们可以被看成是 XML 导出功能：

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

每一个函数的返回值都是xml。

table_to_xml映射由参数tbl传递的命名表的内容。regclass类型接受使用常见标记标识表的字符串，包括可选的模式限定和双引号。query_to_xml执行由参数query传递的查询并且映射结果集。cursor_to_xml从cursor指定的游标中取出指定数量的行。如果需要映射一个大型的表，我们推荐这种变体，因为每一个函数都是在内存中构建结果值的。

如果tableforest为假，则结果的 XML 文档看起来像这样：

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

如果tableforest为真，结果是一个看起来像这样的 XML 内容片断：

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

如果没有表名可用，在映射一个查询或一个游标时，在第一种格式中使用串table，在第二种格式中使用row。

这几种格式的选择由用户决定。第一种格式是一个正确的 XML 文档，它在很多应用中都很重要。如果结果值要被重组为一个文档，第二种格式在cursor_to_xml函数中更有用。前文讨论的产生 XML 内容的函数（特别是xmlelement）可以被用来把结果修改成符合用户的要求。

数据值会被以前文的函数xmlelement中描述的相同方法映射。

参数nulls决定空值是否会被包含在输出中。如果为真，列中的空值被表示为：

```
<columnname xsi:nil="true"/>
```

其中xsi是 XML 模式实例的 XML 名字空间前缀。一个合适的名字空间声明将被加入到结果值中。如果为假，包含空值的列将被从输出中忽略掉。

参数targetns指定想要的结果的 XML 名字空间。如果没有想要的特定名字空间，将会传递一个空串。

下面的函数返回 XML 模式文档，这些文档描述上述对应函数所执行的映射：

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns
 text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns
 text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean,
 targetns text)
```

最重要的是相同的参数被传递来获得匹配的 XML 数据映射和 XML 模式文档。

下面的函数产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档（或森林）中。在要求自包含和自描述的结果是它们非常有用：

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean,
 targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean,
 targetns text)
```

另外，下面的函数可用于产生相似的整个模式或整个当前数据库的映射：

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns
 text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean,
 targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

注意这些函数可能产生很多数据，它们都需要在内存中被构建。在请求大型模式或数据库的内容映射时，可以考虑分别映射每一个表，甚至通过一个游标来映射。

一个模式内容映射的结果看起来像这样：

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

其中一个表映射的格式取决于上文解释的tableforest参数。

一个数据库内容映射的结果看起来像这样：

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

其中的模式映射如上所述。

作为一个使用这些函数产生的输出的例子，图 9.1展示了一个 XSLT 样式表，它将table_to_xml_and_xmlschema的输出转换为一个包含表数据的扁平转印的 HTML 文档。以一种相似的方式，这些函数的结果可以被转换成其他基于 XML 的格式。

图 9.1. 转换 SQL/XML 输出到 HTML 的 XSLT 样式表

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/
xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/
xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. JSON 函数和操作符

表 9.4展示了可以用于两种 JSON 数据类型（见第 8.14 节的操作符。

表 9.43. json和jsonb 操作符

操作符	右操作数类型	描述	例子	例子结果
->	int	获得 JSON 数组元素（索引从 0 开始，负整数从末尾开始计）	'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2	{"c":"baz"}
->	text	通过键获得 JSON 对象域	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	以text形式获得 JSON 数组元素	'[1,2,3]'::json->>2	3
->>	text	以text形式获得 JSON 对象域	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	获取在指定路径的 JSON 对象	'{"a": {"b": {"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	以text形式获取在指定路径的 JSON 对象	'{"a": [1,2,3],"b": [4,5,6]}'::json#>>'{a,2}'	3

> **注意**
>
> 对json和jsonb类型，这些操作符都有其并行变体。 域/元素/路径抽取操作符返回与其左手输入（json或jsonb） 相同的类型，不过那些被指定为返回text的除外，它们的返回值会被强制 为文本。如果该 JSON 输入没有匹配请求的正确结构（例如那样的元素不存在），这些 域/元素/路径抽取操作符会返回 NULL 而不是失败。 接受整数 JSON 数组下标的 域/元素/路径抽取操作符都支持表示从数组末尾开始的负值下标形式。

表 9.1中展示的标准比较操作符只对 jsonb有效，而不适合json。它们遵循在第 8.14.4 节给出的 B 树操作规则。

如表 9.4中所示，还存在一些只适合 jsonb的操作符。这些操作符中的很多可以用jsonb 操作符类索引。jsonb包含和存在语义的完整描述可参见第 8.14.3 节第 8.14.4 描述了如何 用这些操作符来有效地索引jsonb数据。

表 9.44. 额外的jsonb操作符

操作符	右操作数类型	描述	例子
@>	jsonb	左边的 JSON 值是否在顶层包含右边的 JSON 路径/值项?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	左边的 JSON 路径/值项是否被包含在右边的 JSON 值的顶层?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	键/元素字符串是否存在于 JSON 值的顶层?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	这些数组字符串中的任何一个是否做为顶层键存在?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']

操作符	右操作数类型	描述	例子
?&	text[]	是否所有这些数组字符串都作为顶层键存在?	'["a", "b"]'::jsonb ?& array['a', 'b']
||	jsonb	把两个jsonb值串接成一个新的jsonb值	'["a", "b"]'::jsonb || '["c", "d"]'::jsonb
-	text	从左操作数删除键/值对或者string 元素。键/值对基于它们的键值来匹配。	'{"a": "b"}'::jsonb - 'a'
-	text[]	从左操作数中删除多个键/值对或者string元素。键/值对基于它们的键值来匹配。	'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]
-	integer	删除具有指定索引（负值表示倒数）的数组元素。如果 顶层容器不是数组则抛出一个错误。	'["a", "b"]'::jsonb - 1
#-	text[]	删除具有指定路径的域或者元素（对于 JSON 数组，负值 表示倒数）	'["a", {"b":1}]'::jsonb #- '{1,b}'

> **注意**
>
> ||操作符将其每一个操作数的顶层的元素串接起来。它不会递归 操作。例如，如果两个操作数都是具有公共域名称的对象，结果中的域值将 只是来自右手操作数的值。

表 9.45展示了可用于创建 json 和 jsonb值的函数（没有用于 jsonb的与row_to_json和array_to_json等价的函数。不过，to_jsonb函数 提供了这些函数的很大一部分相同的功能）。

表 9.45. JSON 创建函数

函数	描述	例子	例子结果
to_json(anyelement) to_jsonb(anyelement)	把该值返回为json或者jsonb。数组和组合会被（递归）转换成数组和对象；对于不是数组和组合的值，如果有 从该类型到json的造型，造型函数将被用来执行该转换；否则将产生一个标量值。对于任何不是数字、布尔、空值的标 量类型，将使用文本表达，在这种	to_json('Fred said "Hi."'::text)	"Fred said \"Hi.\""

函数	描述	例子	例子结果
	风格下它是一个合法的 json或者jsonb值。		
array_to_json(anyarray [, pretty_bool])	把数组作为一个 JSON 数组返回。一个 PostgreSQL 多维数组会成为一个数组 的 JSON 数组。如果pretty_bool为真，将在 第 1 维度的元素之间增加换行。	array_to_json('{{1,5},{99,100}}'::int[])	[[1,5],[99,100]]
row_to_json(record [, pretty_bool])	把行作为一个 JSON 对象返回。如果pretty_bool为真，将在第1层元素之间增加换行。	row_to_json(row(1,'foo'))	{"f1":1,"f2":"foo"}
json_build_array(VARIADIC "any") jsonb_build_array(VARIADIC "any")	从可变参数列表构造一个可能包含异质类型的 JSON 数组。	json_build_array(1,2,'3',4,5)	[1, 2, "3", 4, 5]
json_build_object(VARIADIC "any") jsonb_build_object(VARIADIC "any")	从可变参数列表构造一个 JSON 对象。通过转换，该参数列表由交替 出现的键和值构成。	json_build_object('foo',1,'bar',2)	{"foo": 1, "bar": 2}
json_object(text[]) jsonb_object(text[])	从一个文本数组构造一个 JSON 对象。该数组必须可以是具有偶数个成员的 一维数组（成员被当做交替出现的键/值对），或者是一个二维数组（每一个 内部数组刚好有 2 个元素，可以被看做是键/值对）。	json_object('{a, 1, b, "def", c, 3.5}') json_object('{{a, 1},{b, "def"},{c, 3.5}}')	{"a": "1", "b": "def", "c": "3.5"}
json_object(keys text[], values text[]) jsonb_object(keys text[], values text[])	json_object的这种形式从两个独立的数组得到键/值对。在其他方面和一个参数的形式相同。	json_object('{a, b}', '{1,2}')	{"a": "1", "b": "2"}

注意

array_to_json和row_to_json与to_json 具有相同的行为，不过它们提供了更好的打印选项。针对to_json所描述 的行为同样也适用于由其他 JSON 创建函数转换的每个值。

> **注意**
>
> hstore扩展有一个从hstore到json 的造型，因此通过 JSON 创建函数转换的hstore值将被表示为 JSON 对象，而不是原始字符串值。

表 9.46展示了可用来处理json 和jsonb值的函数。

表 9.46. JSON 处理

函数	返回值	描述	例子	例子结果
json_array_length(json) jsonb_array_length(jsonb)	int	返回最外层 JSON 数组中的元素数量。	json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')	5
json_each(json) jsonb_each(jsonb)	setof key text, value json setof key text, value jsonb	扩展最外层的 JSON 对象成为一组键/值对。	select * from json_each('{"a":"foo", "b":"bar"}')	key | value -----+------- a | "foo" b | "bar"
json_each_text(json) jsonb_each_text(jsonb)	setof key text, value text	扩展最外层的 JSON 对象成为一组键/值对。返回值将是text类型。	select * from json_each_text('{"a":"foo", "b":"bar"}')	key | value -----+------- a | foo b | bar
json_extract_path(from_json json, VARIADIC path_elems text[]) jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])	json jsonb	返回由path_elems指向的 JSON 值（等效于#>操作符）。	json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4')	{"f5":99,"f6":"foo"}
json_extract_path_text(from_json json, VARIADIC path_elems text[]) jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])	text	以text返回由path_elems指向的 JSON 值（等效于#>>操作符）。	json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4','f6')	foo
json_object_keys(json) jsonb_object_keys(jsonb)	setof text	返回最外层 JSON 对象中的键集合。	json_object_keys('{"f1":"abc","f2":{"f3":"a","f4":"b"}}')	json_object_keys ------------------ f1 f2
json_populate_record(base anyelement, from_json json) jsonb_populate_record(base anyelement,	anyelement	扩展from_json中的对象成一个行，它的列匹配由base定义的记录类型（见下文的注释）。	select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}')	a | b | c ---+---+---------- +------------

函数	返回值	描述	例子	例子结果
from_json jsonb)				1 | {2,"a b"} | (4,"a b c")
json_populate_recordset(base anyelement, from_json json) jsonb_populate_recordset(base anyelement, from_json jsonb)	setof anyelement	扩展from_json中最外的对象数组为一个集合，该集合的列匹配由base定义的记录类型。	select * from json_populate_recordset '[{"a":1,"b":2}, {"a":3,"b":4}]')	cordset(null::myrowtype, ---+--- 1 | 2 3 | 4
json_array_elements(json) jsonb_array_elements(jsonb)	setof json setof jsonb	把一个 JSON 数组扩展成一个 JSON 值的集合。	select * from json_array_elements('[1,true, [2,false]]')	value --- 1 true [2,false]
json_array_elements_text(json) jsonb_array_elements_text(jsonb)	setof text	把一个 JSON 数组扩展成一个text值集合。	select * from json_array_elements_text('["foo", "bar"]')	value --- foo bar
json_typeof(json) jsonb_typeof(jsonb)	text	把最外层的 JSON 值的类型作为一个文本字符串返回。可能的类型是： object、array、string、number、boolean以及null。	json_typeof('-123.4')	number
json_to_record(json) jsonb_to_record(jsonb)	record	从一个 JSON 对象（见下文的注解）构建一个任意的记录。正如所有返回record的函数一样，调用者必须用一个AS子句显式地定义记录的结构。	select * from json_to_record('{"a":1,"b": [1,2,3],"c": [1,2,3],"e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)	a | b c | d | ---+---------+--- +---------+--- 1 | [1,2,3] | {1,2,3} | | (123,"a b c")
json_to_recordset(json) jsonb_to_recordset(jsonb)	setof record	从一个 JSON 对象数组（见下文的注解）构建一个任意的记录集合。正如所有返回record 的函数一样，调用者必须用一个AS子句显式地定义记录的结构。	select * from json_to_recordset('[{"a":1,"b":"foo"}, {"a":"2","c":"bar"}]') as x(a int, b text);	a | b ---+--- 1 | foo 2 |
json_strip_nulls(from_json json) jsonb_strip_nulls(from_json jsonb)	json jsonb	返回from_json，其中所有具有空值的 对象域都被省略。其他空值不动。	json_strip_nulls('[{"f1":1,"f2":null},2,null,3]	[{"f1":1},2,null,3]

函数	返回值	描述	例子	例子结果
jsonb_set(target jsonb, path text[], new_value jsonb[, create_missing boolean])	jsonb	返回target，其中由 path指定的节用new_value替换，如果 path指定的项不存在并且create_missing为真（默认为true）则加上new_value。正如面向路径的 操作符一样，出现在path中的 负整数表示从 JSON数组的末尾开始数。	jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false) jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f3}','[2,3,4]')	[{"f1": [2, 3, 4], "f2": null}, 2, null, 3] [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]
jsonb_insert(target jsonb, path text[], new_value jsonb, [insert_after boolean])	jsonb	返回被插入了new_value的target，如果path指定的target节在一个 JSONB 数组中，new_value将被插入到目标之前（insert_after为默认情况，false）或者之后（insert_after为真）。如果path指定的target节在一个 JSONB 对象内，则只有当target不存在时才插入new_value。对于面向路径的操作符来说，出现在path中的负整数表示从 JSON数组的末尾开始计数。	jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"') jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)	{"a": [0, "new_value", 1, 2]} {"a": [0, 1, "new_value", 2]}
jsonb_pretty(from_json jsonb)	text	把from_json返回成一段 缩进后的JSON 文本。	jsonb_pretty('[{"f1":1,"f2":null},2,null,3]')	[{ "f1": 1, "f2": null }, 2, null, 3]

> **注意**
>
> 很多这些函数和操作符将把 JSON 字符串中的 Unicode 转义转换成合适的单一字符。如果 输入类型是jsonb，这就没有问题，因为该转换已经完成了。但是对于json 输入，这可能会导致抛出一个错误（如第 8.14 所述）。

> **注意**
>
> 虽然函数json_populate_record、json_populate_recordset、json_to_record以及json_to_recordset的例子使用了常量，但常见的用法是引用FROM子句中的表并且使用其json或jsonb列之一作为函数的参数。然后抽取出的键值可以被查询的其他部分引用，例如WHERE子句和目标列表。以这种方式抽取多个值的性能比用以键为单位的操作符单个抽取它们的性能更好。
>
> JSON键被匹配到目标行类型中的相同列名。这些函数的JSON类型强制是一种“尽力而为”的方式并且对于某些类型可能得不到想要的值。不出现在目标行类型中的JSON字段将从输出中忽略，而且不匹配任何JSON字段的目标列将为NULL。

> **注意**
>
> jsonb_set和jsonb_insert的path参数中除最后一项之外的所有项都必须存在于target中。如果create_missing为假，jsonb_set的path参数的所有项都必须存在。如果这些条件不满足，则返回的target不会被改变。
>
> 如果最后的路径项是一个对象键，在它不存在且给定了新值的情况下会创建它。如果最后的路径项是一个数组索引，为正值则表示从左边开始计数，为负值表示从右边开始计数 - -1表示最右边的元素，以此类推。如果该项超过范围 -array_length .. array_length -1 并且 create_missing 为真，则该项为负时把新值加载数组的开始处，而该项为正时把新值加在数组的末尾处。

> **注意**
>
> 不要把json_typeof函数的null返回值与 SQL 的 NULL 弄混。 虽然调用json_typeof('null'::json)将会返回null，但调用json_typeof(NULL::json)将会返回一个 SQL 的 NULL。

> **注意**
>
> 如果json_strip_nulls的参数在任何对象中包含重复的域名称， 结果在语义上可能有所不同，具体取决于它们发生的顺序。这不是 jsonb_strip_nulls的一个问题，因为jsonb值 不能具有重复的对象域名称。

也可参见第 9.20 节解聚集函数json_agg，它可以把记录值聚集成 JSON。还有聚集函数json_object_agg，它可以把值对聚集成一个 JSON 对象。还有它们的jsonb等效体，jsonb_agg和jsonb_object_agg.

9.16. 序列操作函数

本节描述用于操作序列对象的函数，序列对象也被称为序列生成器或者就是序列。序列对象都是用CREATE SEQUENCE创建的特殊的单行表。序列对象通常用于为表的行生成唯一的标识符。表 9.47中列出的这些序列函数，可以为我们从序列对象中获取连续的序列值提供了简单的、多用户安全的 方法。

表 9.47. 序列函数

函数	返回类型	描述
currval(regclass)	bigint	返回最近一次用nextval获取的指定序列的值
lastval()	bigint	返回最近一次用nextval获取的任何序列的值
nextval(regclass)	bigint	递增序列并返回新值
setval(regclass, bigint)	bigint	设置序列的当前值
setval(regclass, bigint, boolean)	bigint	设置序列的当前值以及is_called标志

将要由序列函数调用操作的序列是用一个regclass参数声明的， 它只是序列在pg_class系统表里面的 OID。不过，你不需要手工查找 OID， 因为regclass数据类型的输入转换器会帮你做这件事情。 只要写出用单引号包围的序列名字即可，因此它看上去像文本常量。为了和普通SQL名字处理兼容，这个字串将转换成小写形式， 除非在序列名字周围包含双引号。因此：

```
nextval('foo')      操作序列foo
nextval('FOO')      操作序列foo
nextval('"Foo"')    操作序列Foo
```

必要时序列名可以用模式限定：

```
nextval('myschema.foo')     操作myschema.foo
nextval('"myschema".foo')   同上
nextval('foo')              在搜索路径中查找foo
```

参阅第 8.19 获取有关regclass的更多信息。

注意

在PostgreSQL 8.1 之前，序列函数的参数类型是text， 而不是regclass，并且前文所述的从文本串到 OID 值的转换将在每次调用的时候发生。 为了向后兼容，这个处理仍然存在，但是在内部实际上是通过在函数调用前隐式地将text转换成regclass实现的。

当你把一个序列函数的参数写成一个无修饰的文字串，那么它将变成类型为regclass的常量。因为这只是一个 OID，它将跟踪最初标识的序列，而不管后面是否改名、模式变化等等。 这种“早期绑定”的行为通常是列默认值和视图中引用的序列所需要的。 但是有时候你可能想要“延迟绑定”，其中序列的引用是在运行时解析的。要得到延迟绑定的行为，我们可以强制常量被存储为text常量，而不是regclass：

```
nextval('foo'::text)      foo在运行时查找
```

请注意，延迟绑定是PostgreSQL版本 8.1 之前唯一被支持的行为， 因此你可能需要做这些来保留旧应用的语义。

> 当然，序列函数的参数也可以是表达式。如果它是一个文本表达式，那么隐式的转换将导致运行时的查找。

可用的序列函数有：

nextval

递增序列对象到它的下一个值并且返回该值。这个动作是自动完成的： 即使多个会话并发执行nextval，每个进程也会安全地收到一个唯一的序列值。

如果一个序列对象是用默认参数创建的，连续的nextval调用将会返回从 1 开始的连续的值。其他的行为可以通过在CREATE SEQUENCE命令中使用特殊参数来获得；详见该命令的参考页。

> **重要**
>
> 为了避免阻塞从同一个序列获取序号的并发事务，nextval操作从来不会被回滚。也就是说，一旦一个值被取出就视同被用掉并且不会被再次返回给调用者，即便调用该操作的外层事务后来中止或者调用查询后来没有使用取得的值也是这样。例如一个带有ON CONFLICT子句的INSERT会计算要被插入的元组，其中可能就包括调用nextval，然后才会检测到导致它转向ON CONFLICT规则的冲突。这种情况就会在已分配值的序列中留下未被使用的“空洞”。因此，PostgreSQL的序列对象不能被用来得到“无间隙”的序列。

这个函数要求序列上的USAGE或者UPDATE特权。

currval

在当前会话中返回最近一次nextval取到的该序列的值（如果在本会话中从未在该序列上调用过nextval，那么会报告一个错误）。请注意因为此函数返回一个会话本地的值，不论其它会话是否在当前会话之后执行过nextval，它都能给出一个可预测的回答。

这个函数要求序列上的USAGE或者SELECT特权。

lastval

返回当前会话里最近一次nextval返回的值。 这个函数等效于currval，只是它不用序列名作为参数， 它会引用当前会话里面最近一次被应用的序列的nextval。如果当前会话还没有调用过nextval，那么调用lastval会报错。

这个函数要求上一次使用的序列上的USAGE或者SELECT特权。

setval

重置序列对象的计数器值。双参数的形式设置序列的last_value域为指定值并且将其is_called域设置为 true，表示下一次nextval将在返回值之前递增该序列。currval报告的值也被设置为指定的值。在三参数形式里，is_called可以设置为true或false。true具有和双参数形式相同的效果。如果你把它设置为false，那么下一次nextval将返回指定的值，而从随后的nextval才开始递增该序列。此外，在这种情况中currval报告的值不会被改变。 例如：

```
SELECT setval('foo', 42);           下一次nextval会返回 43
SELECT setval('foo', 42, true);     同上
SELECT setval('foo', 42, false);    下一次nextval将返回 42
```

setval返回的结果就是它的第二个参数的值。

> **重要**
>
> 因为序列是非事务的，setval造成的改变不会由于事务的回滚而撤销。

这个函数要求序列上的UPDATE特权。

9.17. 条件表达式

本节描述在PostgreSQL中可用的SQL兼容的条件表达式。

> **提示**
>
> 如果你的需求超过这些条件表达式的能力，你可能会希望用一种更富表现力的编程语言写一个服务器端函数。

9.17.1. CASE

SQL CASE表达式是一种通用的条件表达式，类似于其它编程语言中的 if/else 语句：

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE子句可以用于任何表达式可以出现的地方。每一个condition是一个返回boolean结果的表达式。如果结果为真，那么CASE表达式的结果就是符合条件的result，并且剩下的CASE表达式不会被处理。如果条件的结果不为真，那么以相同方式搜寻任何随后的WHEN子句。如果没有WHEN condition为真，那么CASE表达式的值就是在ELSE子句里的result。如果省略了ELSE子句而且没有条件为真，结果为空。

例子：

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
```

```
 3 | other
```

所有result表达式的数据类型都必须可以转换成单一的输出类型。 参阅第 10.5 节获取细节。

下面这个“简单”形式的CASE表达式是上述通用形式的一个变种：

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

第一个expression会被计算，然后与所有在WHEN子句中的每一个value对比，直到找到一个相等的。如果没有找到匹配的，则返回在ELSE子句中的result（或者控制）。 这类似于 C 里的switch语句。

上面的例子可以用简单CASE语法来写：

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE表达式并不计算任何无助于判断结果的子表达式。例如，下面是一个可以避免被零除错误的方法：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **注意**
>
> 如第 4.2.14 节所述，在有几种情况中一个表达式的子表达式 会被计算多次，因此“CASE只计算必要的表达式”这 一原则并非不可打破。例如一个常量子表达式1/0通常将会在规划时导致一次 除零错误，即便它位于一个执行时永远也不会进入的CASE分支时也是 如此。

9.17.2. COALESCE

```
COALESCE(value [, ...])
```

COALESCE函数返回它的第一个非空参数的值。当且仅当所有参数都为空时才会返回空。它常用于在为显示目的检索数据时用缺省值替换空值。例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

如果description不为空，这将会返回它的值，否则如果short_description非空则返回short_description的值，如果前两个都为空则返回(none)。

和CASE表达式一样，COALESCE将不会 计算无助于判断结果的参数；也就是说，在第一个非空参数右边的参数不会被计算。这个 SQL 标准函数提供了类似于NVL和IFNULL的能力，它们被用在某些其他数据库系统中。

9.17.3. NULLIF

```
NULLIF(value1, value2)
```

当value1和value2相等时，NULLIF返回一个空值。 否则它返回value1。 这些可以用于执行前文给出的COALESCE例子的逆操作：

```
SELECT NULLIF(value, '(none)') ...
```

在这个例子中，如果value是(none)，将返回空值，否则返回value的值。

9.17.4. GREATEST和LEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

GREATEST和LEAST函数从一个任意的数字表达式列表里选取最大或者最小的数值。 这些表达式必须都可以转换成一个普通的数据类型，它将会是结果类型 （参阅第 10.5 节取细节）。列表中的 NULL 数值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才会是 NULL。

请注意GREATEST和LEAST都不是 SQL 标准，但却是很常见的扩展。某些其他数据库让它们在任何参数为 NULL 时返回 NULL，而不是在所有参数都为 NULL 时才返回 NULL。

9.18. 数组函数和操作符

表 9.48显示了可以用于数组类型的操作符。

表 9.48. 数组操作符

操作符	描述	例子	结果
=	等于	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t

操作符	描述	例子	结果
@>	包含	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	被包含	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重叠（具有公共元素）	ARRAY[1,4,3] && ARRAY[2,1]	t
||	数组和数组串接	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	数组和数组串接	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	元素到数组串接	3 || ARRAY[4,5,6]	{3,4,5,6}
||	数组到元素串接	ARRAY[4,5,6] || 7	{4,5,6,7}

数组比较使用默认的 B-Tree 在元素数据类型上的比较函数对数组内容按元素逐一进行。 多维数组的元素按照行序进行访问（最后的下标变化最快）。如果两个数组的内容相同但维数不等，那么维度信息中的第一个不同将决定排序顺序（这是对PostgreSQL 8.2 之前版本的修改： 老版本认为内容相同的两个数组相等，即使它们的维数或下标范围并不相同）。

参阅第 8.15 获取有关数组操作符行为的更多细节。有关哪些操作符支持被索引的操作，请参阅第 11.2 节。

表 9.49 展示了可以用于数组类型的函数。 参阅第 8.15 获取更多信息以及使用这些函数的例子。

表 9.49. 数组函数

函数	返回类型	描述	例子	结果
array_append(anyarray, anyelement)	anyarray	向一个数组的末端追加一个元素	array_append(ARRAY[1,2], 3)	{1,2,3}
array_cat(anyarray, anyarray)	anyarray	连接两个数组	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_ndims(anyarray)	int	返回数组的维度数	array_ndims(ARRAY[[1,2,3], [4,5,6]])	2
array_dims(anyarray)	text	返回数组的维度的文本表示	array_dims(ARRAY[[1,2,3], [4,5,6]])	[1:2][1:3]
array_fill(anyelement, int[], [, int[]])	anyarray	返回一个用提供的值和维度初始化好的数组，可以选择下界不为 1	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7}
array_length(anyarray, int)	int	返回被请求的数组维度的长度	array_length(array[1,2,3], 1)	3
array_lower(anyarray, int)	int	返回被请求的数组维度的下界	array_lower('[0:2]={1,2,3}'::int[], 1)	0
array_position(anyarray, anyelement [, int])	int	返回在该数组中从第三个参数指定的元素开始或者 第一个元素开始（数组必须是	array_position(ARRAY['sun','mon','tue','wed','mon')	2

函数	返回类型	描述	例子	结果
		一维的）、第二个参数的 第一次出现的下标		
array_positions(anyarray, anyelement)	int[]	返回在第一个参数给定的数组（数组必须是一维的）中， 第二个参数所有出现位置的下标组成的数组	array_positions(ARRAY['A','A','B','A'], 'A')	{1,2,4}
array_prepend(anyelement, anyarray)	anyarray	向一个数组的首部追加一个元素	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_remove(anyarray, anyelement)	anyarray	从数组中移除所有等于给定值的所有元素（数组必须是一维的）	array_remove(ARRAY[1,2,3,2], 2)	{1,3}
array_replace(anyarray, anyelement, anyelement)	anyarray	将每一个等于给定值的数组元素替换成一个新值	array_replace(ARRAY[1,2,5,4], 5, 3)	{1,2,3,4}
array_to_string(anyarray, text [, text])	text	使用提供的定界符和可选的空串连接数组元素	array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')	1,2,3,*,5
array_upper(anyarray, int)	int	返回被请求的数组维度的上界	array_upper(ARRAY[1,8,3,7], 1)	4
cardinality(anyarray)	int	返回数组中元素的总数，如果数组为空则返回 0	cardinality(ARRAY[[1,2], [3,4]])	4
string_to_array(text, text [, text])	text[]	使用提供的定界符和可选的空串将字符串划分成数组元素	string_to_array('xx~^~yy~^~zz', '~^~', 'yy')	{xx,NULL,zz}
unnest(anyarray)	setof anyelement	将一个数组扩展成一组行	unnest(ARRAY[1,2])	1 2 (2 rows)
unnest(anyarray, anyarray [, ...])	setof anyelement, anyelement [, ...]	把多维数组（可能是不同类型）扩展成一个行的集合。 这只允许用在 FROM 子句中，见第 7.2.1.4 节	unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])	1 foo 2 bar NULL baz (3 rows)

在array_position和array_positions中， 每一个数组元素都使用IS NOT DISTINCT FROM 语义与要搜索的值比较。

在array_position中，如果值没有找到则返回 NULL。

在array_positions中，只有当数组为 NULL时才返回NULL，如果该值 没有在该数组中找到则返回一个空数组。

在string_to_array中，如果定界符参数为 NULL，输入字符串中的每一个字符将变成结果数组中的一个独立元素。如果定界符是一个空串，则整个输入字符串被作为一个单一元素的数组返回。否则输入字符串会被在每一个出现定界符字符串的位置分裂。

在string_to_array中，如果空值串参数被忽略或者为 NULL，输入中的子串不会被替换成NULL。在array_to_string中，如果空值串参数被忽略或者为 NULL，该数组中的任何空值元素会被简单地跳过并且不会在输出串中被表示。

> **注意**
>
> string_to_array的行为中有两点与PostgreSQL 9.1之前的版本不同。首先，当输入串的长度为零时，它将返回一个空（无元素）数组而不是 NULL。其次，如果定界符串为 NULL，该函数会将输入划分成独立字符，而不是像以前那样返回 NULL。

也可参见第 9.20 节解用于数组的聚集函数array_agg。

9.19. 范围函数和操作符

范围类型的概述请见第 8.17 节。

表 9.50展示了范围类型可用的操作符。

表 9.50. 范围操作符

操作符	描述	例子	结果
=	等于	int4range(1,5) = '[1,4]'::int4range	t
<>	不等于	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	小于	int4range(1,10) < int4range(2,3)	t
>	大于	int4range(1,10) > int4range(1,5)	t
<=	小于等于	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	大于等于	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	包含范围	int4range(2,4) @> int4range(2,3)	t
@>	包含元素	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	范围被包含	int4range(2,4) <@ int4range(1,7)	t
<@	元素被包含	42 <@ int4range(1,7)	f
&&	重叠（有公共点）	int8range(3,7) && int8range(4,12)	t
<<	严格左部	int8range(1,10) << int8range(100,110)	t

操作符	描述	例子	结果
>>	严格右部	int8range(50,60) >> int8range(20,30)	t
&<	不超过右部	int8range(1,20) &< int8range(18,20)	t
&>	不超过左部	int8range(7,20) &> int8range(5,10)	t
-|-	相邻	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	并	numrange(5,15) + numrange(10,20)	[5,20)
*	交	int8range(5,15) * int8range(10,20)	[10,15)
-	差	int8range(5,15) - int8range(10,20)	[5,10)

简单比较操作符<、 >、<=和 >=首先比较下界，并且只有在下界相等时才比较上界。这些比较通常对范围不怎么有用，但是还是提供它们以便能够在范围上构建 B树索引。

当涉及一个空范围时，左部/右部/相邻操作符总是返回假；即一个空范围被认为不在任何其他范围前面或者后面。

如果结果范围可能需要包含两个分离的子范围，并和差操作符将会失败，因为这样的范围无法被表示。

表 9.51显示可用于范围类型的函数。

表 9.51. 范围函数

函数	返回类型	描述	例子	结果
lower(anyrange)	范围的元素类型	范围的下界	lower(numrange(1.1,2.2))	1.1
upper(anyrange)	范围的元素类型	范围的上界	upper(numrange(1.1,2.2))	2.2
isempty(anyrange)	boolean	范围为空？	isempty(numrange(1.1,2.2))	false
lower_inc(anyrange)	boolean	下界包含在内？	lower_inc(numrange(1.1,2.2))	true
upper_inc(anyrange)	boolean	上界包含在内？	upper_inc(numrange(1.1,2.2))	false
lower_inf(anyrange)	boolean	下界无限？	lower_inf('(,)'::daterange)	true
upper_inf(anyrange)	boolean	上界无限？	upper_inf('(,)'::daterange)	true
range_merge(anyrange, anyrange)	anyrange	包含两个给定范围的最小范围	range_merge('[1,2)'::int4range, '[3,4)'::int4range)	[1,4)

如果范围为空或者被请求的界是无限的，lower和upper函数返回空值。函数lower_inc、upper_inc、lower_inf和upper_inf对一个空范围全部返回假。

9.20. 聚集函数

聚集函数从一个输入值的集合计算出一个单一值。内建的通用聚集函数在表 9.52中列出，而统计性聚集在表 9.53中列出。内建的组内有序集聚集函数在表 9.54中列出，而内建的组内假想集聚集在表 9.55中列出。与聚集函数紧密相关的分组操作在表 9.56中列出。第 4.2.7 节中会解释针对聚集函数的特殊语法考虑。额外的介绍信息请参考第 2.7 节。

表 9.52. 通用聚集函数

函数	参数类型	返回类型	部分模式	描述
array_agg(expression)	任何非数组类型	参数类型的数组	No	输入值（包括空）被连接到一个数组
array_agg(expression)	任意数组类型	和参数数据类型相同	No	输入数组被串接到一个更高维度的数组中 （输入必须都具有相同的维度并且不能为空或者 NULL）
avg(expression)	smallint, int, bigint、real、double precision、numeric或interval	对于任何整数类型参数是numeric，对于一个浮点参数是double precision，否则和参数数据类型相同	Yes	所有输入值的平均值（算术平均）
bit_and(expression)	smallint、int、bigint或bit	与参数数据类型相同	Yes	所有非空输入值的按位与，如果没有非空值则结果是空值
bit_or(expression)	smallint, int, bigint, or bit	与参数数据类型相同	Yes	所有非空输入值的按位或，如果没有非空值则结果是空值
bool_and(expression)	bool	bool	Yes	如果所有输入值为真则结果为真，否则为假
bool_or(expression)	bool	bool	Yes	至少一个输入值为真时结果为真，否则为假
count(*)		bigint	Yes	输入的行数
count(expression)	any	bigint	Yes	expression值非空的输入行的数目
every(expression)	bool	bool	Yes	等价于bool_and
json_agg(expression)	any	json	No	将值聚集成一个 JSON 数组
jsonb_agg(expression)	any	jsonb	No	把值聚合成一个 JSON 数组
json_object_agg(name, value)	(any, any)	json	No	将名字/值对聚集成一个 JSON 对象
jsonb_object_agg(name, value)	(any, any)	jsonb	No	把名字/值对聚合成一个 JSON 对象

函数	参数类型	返回类型	部分模式	描述
max(expression)	任意数组、数字、串、日期/时间、网络或者枚举类型，或者这些类型的数组	与参数数据类型相同	Yes	所有输入值中expression的最大值
min(expression)	任意数组、数字、串、日期/时间、网络或者枚举类型，或者这些类型的数组	与参数数据类型相同	Yes	所有输入值中expression的最小值
string_agg(expression, delimiter)	(text, text) 或 (bytea, bytea)	与参数数据类型相同	No	输入值连接成一个串，用定界符分隔
sum(expression)	smallint、int、bigint、real、double precision、numeric、interval或money	对smallint或int参数是bigint，对bigint参数是numeric，否则和参数数据类型相同	Yes	所有输入值的expression的和
xmlagg(expression)	xml	xml	No	连接 XML 值（参见第 9.14.1.7 节

请注意，除了count以外，这些函数在没有行被选中时返回控制。尤其是sum函数在没有输入行时返回空值，而不是零，并且array_agg在这种情况返回空值而不是一个空数组。必要时可以用coalesce把空值替换成零或一个空数组。

支持部分模式的聚集函数有资格参与到各种优化中，例如并行聚集。

注意

布尔聚集bool_and和bool_or对应于标准的 SQL 聚集every和any或some。而对于any 和some，似乎在标准语法中有一个歧义：

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

如果子查询返回一行有一个布尔值的结果，这里的ANY可以被认为是引入一个子查询，或者是作为一个聚集函数。因而标准的名称不能指定给这些聚集。

注意

在把count聚集应用到整个表上时，习惯于使用其他 SQL 数据管理系统的用户可能会对它的性能感到失望。一个如下的查询：

```
SELECT count(*) FROM sometable;
```

将会要求与整个表大小成比例的工作：PostgreSQL将需要扫描整个表或者整个包含表中所有行的索引。

与相似的用户定义的聚集函数一样，聚集函数array_agg、json_agg、jsonb_agg、json_object_agg、jsonb_object_agg、string_agg和xmlagg会依赖

输入值的顺序产生有意义的不同结果值。这个顺序默认是不用指定的，但是可以在聚集调用时使用ORDER BY子句进行控制，如第 4.2.7 节所示。作为一种选择，从一个排序号的子查询来提供输入值通常会有帮助。例如：

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

注意如果外面的查询层次包含额外的处理（例如连接），这种方法可能会失败，因为这可能导致子查询的输出在计算聚集之前被重新排序。

表 9.53展示了通常被用在统计分析中的聚集函数（这些被隔离出来是为了不和常用聚集混淆）。其中描述提到了N，它表示对应于所有非空输入表达式的输入行的数目。在所有情况中，如果计算是无意义的，将会返回空值，例如当N为零。

表 9.53. 用于统计的聚集函数

函数	参数类型	返回类型	部分模式	描述
corr(Y, X)	double precision	double precision	Yes	相关系数
covar_pop(Y, X)	double precision	double precision	Yes	总体协方差
covar_samp(Y, X)	double precision	double precision	Yes	样本协方差
regr_avgx(Y, X)	double precision	double precision	Yes	自变量的平均值（sum(X)/N）
regr_avgy(Y, X)	double precision	double precision	Yes	因变量的平均值（sum(Y)/N）
regr_count(Y, X)	double precision	bigint	Yes	两个表达式都不为空的输入行的数目
regr_intercept(Y, X)	double precision	double precision	Yes	由（X, Y）对决定的最小二乘拟合的线性方程的y截距
regr_r2(Y, X)	double precision	double precision	Yes	相关系数的平方
regr_slope(Y, X)	double precision	double precision	Yes	由（X, Y）对决定的最小二乘拟合的线性方程的斜率
regr_sxx(Y, X)	double precision	double precision	Yes	sum(X^2) - sum(X)^2/N（自变量的“平方和”）
regr_sxy(Y, X)	double precision	double precision	Yes	sum(X*Y) - sum(X) * sum(Y)/N（自变量乘以因变量的“积之合”）
regr_syy(Y, X)	double precision	double precision	Yes	sum(Y^2) - sum(Y)^2/N（因变量的“平方和”）

函数	参数类型	返回类型	部分模式	描述
stddev(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	stddev_samp的历史别名
stddev_pop(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的总体标准偏差
stddev_samp(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的样本标准偏差
variance(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	var_samp的历史别名
var_pop(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的总体方差（总体标准偏差的平方）
var_samp(expression)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的样本方差（样本标准偏差的平方）

表 9.54展示了一些使用 有序集聚集语法的聚集函数。这些函数有时也被称为 “逆分布”函数。

表 9.54. 有序集聚集函数

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
mode() WITHIN GROUP (ORDER BY sort_expression)		任何可排序类型	与排序表达式相同	No	返回最频繁的输入值（如果有多个频度相同的值就选第一个）
percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)	double precision	double precision或者interval	与排序表达式相同	No	连续百分率：返回一个对应于排序中指定分数的值，如有必要就在相邻的输入项之间插值
percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)	double precision[]	double precision或者interval	排序表达式的类型的数组	No	多重连续百分率：返回一个匹配fractions参数形状的结果数组，其中每一个非空元素都用对应于那个百分率的值替换

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)	double precision	一种可排序类型	与排序表达式相同	No	离散百分率：返回第一个在排序中位置等于或者超过指定分数的输入值
percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)	double precision[]	任何可排序类型	排序表达式的类型的数组	No	多重离散百分率：返回一个匹配fractions参数形状的结果数组，其中每一个非空元素都用对应于那个百分率的输入值替换

所有列在表 9.54中的聚集会忽略它们的已排序输入中的空值。对那些有一个fraction参数的聚集来说，该分数值必须位于 0 和 1 之间，否则会抛出错误。不过，一个空分数值会产生一个空结果。

每个列在表 9.55中的聚集都与一个定义在第 9.21 节的同名窗口函数相关联。在每种情况中，聚集结果的计算方法是：假设根据args构建的“假想”行已经被增加到从sorted_args计算得到的已排序行分组中，然后用相关联的窗口函数针对该行返回的值就是聚集的结果。

表 9.55. 假想集聚集函数

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
rank(args) WITHIN GROUP (ORDER BY sorted_args)	VARIADIC "any"	VARIADIC "any"	bigint	No	假想行的排名，为重复的行留下间隔
dense_rank(args) WITHIN GROUP (ORDER BY sorted_args)	VARIADIC "any"	VARIADIC "any"	bigint	No	假想行的排名，不留间隔
percent_rank(args) WITHIN GROUP (ORDER BY sorted_args)	VARIADIC "any"	VARIADIC "any"	double precision	No	假想行的相对排名，范围从 0 到 1
cume_dist(args) WITHIN GROUP (ORDER BY sorted_args)	VARIADIC "any"	VARIADIC "any"	double precision	No	假想行的相对排名，范围从 1/N 到 1

对于这些假想集聚集的每一个，args中给定的直接参数列表必须匹配sorted_args中给定的聚集参数的数量和类型。与大部分的内建聚集不同，这些聚集并不严格，即它们不会丢弃包含空值的输入行。空值的排序根据ORDER BY子句中指定的规则进行。

表 9.56. 分组操作

函数	返回类型	描述
GROUPING(args...)	integer	整数位掩码指示哪些参数不被包括在当前分组集合中

分组操作用来与分组集合（见第 7.2.4 节共同来 区分结果行。GROUPING操作的参数并不会被实际计算，但是它们必 须准确地匹配在相关查询层次的GROUP BY子句中给定的表达式。 最右边参数指派的位是最低有效位，如果对应的表达式被包括在产生结果行的分组 集合的分组条件中则每一位是 0，否则是 1。例如：

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY
 ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

9.21. 窗口函数

窗口函数提供在与当前查询行相关的行集合上执行计算的能力。有关这个特性的介绍请见第 3.5 节 语法细节则请见第 4.2.8 节

表 9.57列出了内建的窗口函数。注意必须使用窗口函数的语法调用这些函数； 一个OVER子句是必需的。

在这些函数之外，任何内建的或者用户定义的通用或者统计性聚集（即非有序集和假想集聚集）都可以被用作一个窗口函数，内建聚集的列表请见第 9.20 节仅当聚集函数调用后面跟着一个OVER子句时，聚集函数才会像窗口函数那样工作，否则它们会按非窗口聚集的方式运行并且为整个集合返回一个单一行。

表 9.57. 通用窗口函数

函数	返回类型	描述
row_number()	bigint	当前行在其分区中的行号，从1计
rank()	bigint	带间隙的当前行排名； 与该行的第一个同等行的row_number相同
dense_rank()	bigint	不带间隙的当前行排名； 这个函数计数同等组

函数	返回类型	描述
percent_rank()	double precision	当前行的相对排名：（rank-1）/（总行数 - 1）
cume_dist()	double precision	累积分布：（在当前行之前或者平级的分区行数）/ 分区行总数
ntile(num_buckets integer)	integer	从1到参数值的整数范围，尽可能等分分区
lag(value anyelement [, offset integer [, default anyelement]])	和value的类型相同	返回value，它在分区内当前行的之前offset个位置的行上计算；如果没有这样的行，返回default替代（必须和value类型相同）。offset和default都是根据当前行计算的结果。如果忽略它们，则offset默认是1，default默认是空值
lead(value anyelement [, offset integer [, default anyelement]])	和value类型相同	返回value，它在分区内当前行的之后offset个位置的行上计算；如果没有这样的行，返回default替代（必须和value类型相同）。offset和default都是根据当前行计算的结果。如果忽略它们，则offset默认是1，default默认是空值
first_value(value any)	same type as value	返回在窗口帧中第一行上计算的value
last_value(value any)	和value类型相同	返回在窗口帧中最后一行上计算的value
nth_value(value any, nth integer)	和value类型相同	返回在窗口帧中第nth行（行从1计数）上计算的value；没有这样的行则返回空值

在表 9.57中列出的所有函数都依赖于相关窗口定义的ORDER BY子句指定的排序顺序。仅考虑ORDER BY列时不能区分的行被称为是同等行。定义的这四个排名函数（包括cume_dist），对于任何两个同等行的答案相同。

注意first_value、last_value和nth_value只考虑“窗口帧”内的行，它默认情况下包含从分区的开始行直到当前行的最后一个同等行。这对last_value可能不会给出有用的结果，有时对nth_value也一样。你可以通过向OVER子句增加一个合适的帧声明（RANGE或GROUPS）来重定义帧。关于帧声明的更多信息请参考第 4.2.8 节

当一个聚集函数被用作窗口函数时，它将在当前行的窗口帧内的行上聚集。 一个使用ORDER BY和默认窗口帧定义的聚集产生一种“运行时求和”类型的行为，这可能是或者不是想要的结果。为了获取在整个分区上的聚集，忽略ORDER BY或者使用ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。 其它窗口帧声明可以用来获得其它的效果。

注意

SQL 标准为lead、lag、first_value、last_value和nth_value定义了一个RESPECT NULLS或IGNORE NULLS选项。 这在PostgreSQL中没有实现：行为总是与标准的默认相同，即RESPECT NULLS。 同样，标准中用

> 于nth_value的FROM FIRST或FROM LAST选项没有实现： 只有支持默认的FROM FIRST行为（你可以通过反转ORDER BY的排序达到FROM LAST的结果）。

cume_dist计算小于等于当前行及其平级行的分区行所占的分数，而percent_rank计算小于当前行的分区行所占的分数，假定当前行不存在于该分区中。

9.22. 子查询表达式

本节描述PostgreSQL中可用的SQL兼容的子查询表达式。所有本节中成文的表达式都返回布尔值（真/假）结果。

9.22.1. EXISTS

```
EXISTS (subquery)
```

EXISTS的参数是一个任意的SELECT语句， 或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么EXISTS的结果就为“真”； 如果子查询没有返回行，那么EXISTS的结果是“假”。

子查询可以引用来自周围的查询的变量，这些变量在该子查询的任何一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以返回至少一行为止， 而不是等到全部结束。在这里写任何有副作用的子查询都是不明智的（例如调用序列函数）；这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容， 所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用EXISTS(SELECT 1 WHERE ...)的形式写所有的EXISTS测试。不过这条规则有例外，例如那些使用INTERSECT的子查询。

下面这个简单的例子类似在col2上的一次内联接，但是它为每个 tab1的行生成最多一个输出，即使存在多个匹配tab2的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. IN

```
expression IN (subquery)
```

右手边是一个圆括弧括起来的子查询， 它必须正好只返回一个列。左手边表达式将被计算并与子查询结果逐行进行比较。 如果找到任何等于子查询行的情况，那么IN的结果就是“真”。 如果没有找到相等行，那么结果是“假”（包括子查询没有返回任何行的情况）。

请注意如果左手边表达式得到空值，或者没有相等的右手边值， 并且至少有一个右手边行得到空值，那么IN结构的结果将是空值，而不是假。这个行为是遵照 SQL 处理空值的一般规则的。

和EXISTS一样，假设子查询将被完成运行完全是不明智的。

```
row_constructor IN (subquery)
```

这种形式的IN的左手边是一个行构造器，如第 4.2.13 节所述。 右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。 左手边表达式将被计算并与子查询结果逐行进行比较。如果找到任意相等的子查询行，则IN的结果为“真”。如果没有找到相等行， 那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。 如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等； 否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等， 要么是空值，并且至少有一个空值，那么IN的结果是空值。

9.22.3. NOT IN

```
expression NOT IN (subquery)
```

右手边是一个用圆括弧包围的子查询，它必须返回正好一个列。左手边表达式将被计算并与子查询结果逐行进行比较。 如果只找到不相等的子查询行（包括子查询不返回行的情况），那么NOT IN的结果是“真”。 如果找到任何相等行，则结果为“假”。

请注意如果左手边表达式得到空值，或者没有相等的右手边值， 并且至少有一个右手边行得到空值，那么NOT IN结构的结果将是空值，而不是真。这个行为是遵照 SQL 处理空值的一般规则的。

和EXISTS一样，假设子查询会完全结束是不明智的。

```
row_constructor NOT IN (subquery)
```

这种形式的NOT IN的左手边是一个行构造器， 如第 4.2.13 节所述。 右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。 左手边表达式将被计算并与子查询结果逐行进行比较。如果找到不等于子查询行的行，则NOT IN的结果为“真”。如果找到相等行， 那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。 如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等； 否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等， 要么是空值，并且至少有一个空值，那么NOT IN的结果是空值。

9.22.4. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

这种形式的右手边是一个圆括弧括起来的子查询， 它必须返回正好一个列。左手边表达式将被计算并使用给出的 操作符对子查询结果逐行进行比较。如果获得任何真值结果，那么ANY的结果就是“真”。 如果没有找到真值结果，那么结果是“假”（包括子查询没有返回任何行的情况）。

SOME是ANY的同义词。IN等价于= ANY。

请注意如果没有任何成功并且至少有一个右手边行为该操作符结果生成空值， 那么ANY结构的结果将是空值，而不是假。 这个行为是遵照 SQL 处理空值布尔组合的一般规则制定的。

和EXISTS一样，假设子查询将被完全运行是不明智的。

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

这种形式的左手边是一个行构造器，如第 4.2.13 节所述。右手边是一个圆括弧括起来的子查询， 它必须返回和左手边列表给出的表达式一样多的列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果比较为任何子查询行返回真，则ANY的结果为“真”。如果比较对每一个子查询行都返回假，则结果为“假”（包括子查询不返回行的情况）。如果比较不对任何行返回真并且至少对一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 9.23.5 节

9.22.5. ALL

expression operator ALL (subquery)

ALL 的这种形式的右手边是一个圆括弧括起来的子查询， 它必须只返回一列。左手边表达式将被计算并使用给出的 操作符对子查询结果逐行进行比较。该操作符必须生成布尔结果。 如果所有行得到真（包括子查询没有返回任何行的情况），ALL的结果就是“真”。如果没有存在任何假值结果，那么结果是“假”。如果比较为任何行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

NOT IN等价于<> ALL。

和EXISTS一样，假设子查询将被完全运行是不明智的。

row_constructor operator ALL (subquery)

ALL的这种形式的左手边是一个行构造器，如第 4.2.13 节所述。 右手边是一个圆括弧括起来的子查询，它必须返回和左手边行中表达式一样多的列。 左手边表达式将被计算并使用给出的 操作符对子查询结果逐行进行比较。如果对所有子查询行该比较都返回真，那么ALL的结果就是“真”（包括子查询没有返回任何行的情况）。如果对任何子查询行比较返回假，则结果为“假”。如果比较对任何子查询行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 9.23.5 节

9.22.6. 单一行比较

row_constructor operator (subquery)

左手边是一个行构造器，如第 4.2.13 节所述。 右手边是一个圆括弧括起来的子查询，该查询必须返回和左手边行中表达式数目完全一样的列。 另外，该子查询不能返回超过一行的数量（如果它返回零行，那么结果就是空值）。 左手边被计算并逐行与右手边的子查询结果行比较。

关于行构造器比较的详细含义请见第 9.23.5 节

9.23. 行和数组比较

本节描述几个特殊的结构，用于在值的组之间进行多重比较。这些形式语法上和前面一节的子查询形式相关，但是不涉及子查询。 这种形式涉及的数组子表达式是PostgreSQL的扩展； 其它的是SQL兼容的。所有本节记录的表达式形式都返回布尔（Boolean）结果（真/假）。

9.23.1. IN

```
expression IN (value [, ...])
```

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果等于任何右手边表达式中的一个，结果为“真”。它是下面形式的缩写

```
expression = value1
OR
expression = value2
OR
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么IN结构的结果将为空值，而不是假。这符合 SQL 处理空值的布尔组合的一般规则。

9.23.2. NOT IN

```
expression NOT IN (value [, ...])
```

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果不等于所有右手边表达式，结果为“真”。它是下面形式的缩写

```
expression <> value1
AND
expression <> value2
AND
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么NOT IN结构的结果将为空值， 而不是我们可能天真地认为的真值。这符合 SQL 处理空值的布尔组合的一般规则。

> **提示**
>
> x NOT IN y在所有情况下都等效于NOT (x IN y)。但是，在处理空值的时候，用NOT IN比用IN更可能迷惑新手。最好尽可能用正逻辑来表达你的条件。

9.23.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

右手边是一个圆括弧包围的表达式，它必须得到一个数组值。左手边表达式被计算并且使用给出的操作符对数组的每个元素进行比较，这个操作符必须得到布尔结果。如果得到了任何真值结果，那么ANY的结果是“真”。 如果没有找到真值结果（包括数组只有零个元素的情况），那么结果是“假”。

如果数组表达式得到一个空数组，ANY的结果将为空值。如果左手边的表达式得到空值，ANY通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到真值比较结果，ANY的结果将是空值而不是假（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

SOME是ANY的同义词。

9.23.4. ALL (array)

expression operator ALL (array expression)

右手边是一个圆括弧包围的表达式，它必须得到一个数组值。左手边表达式将被计算并使用给出的操作符与数组的每个元素进行比较，这个操作符必须得到一个布尔结果。如果所有比较都得到真值结果，那么ALL的结果是 “真”（包括数组只有零个元素的情况）。如果有任何假值结果，那么结果是“假”。

如果数组表达式得到一个空数组，ALL的结果将为空值。如果左手边的表达式得到空值，ALL通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到假值比较结果，ALL的结果将是空值而不是真（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

9.23.5. 行构造器比较

row_constructor operator row_constructor

每一边都是一个行构造器，如第 4.2.13 所述。两个行值必须具有相同数量的域。每一边被计算并且被逐行比较。当操作符是 =、 <>、 < <=、 >、 >=时，允许进行行构造器比较。每一个行元素必须是具有一个默认 B 树操作符类的类型，否则尝试比较会产生一个错误。

> **注意**
>
> Errors related to the number or types of elements might not occur if the comparison is resolved using earlier columns.

=和<>情况略有不同。如果两行的所有对应成员都是非空且相等则这两行被认为相等；如果任何对应成员是非空但是不相等则这两行不相等；否则行比较的结果为未知（空值）。

对于<、<=、>和>=情况，行元素被从左至右比较，在找到一处不等的或为空的元素对就立刻停下来。如果这一对元素都为空值，则行比较的结果为未知（空值）；否则这一对元素的比较结果决定行比较的结果。例如，ROW(1,2,NULL) < ROW(1,3,0)得到真，而不是空值，因为第三对元素并没有被考虑。

> **注意**
>
> 在PostgreSQL 8.2之前，<、<=、>和>=情况不是按照每个 SQL 声明来处理的。一个像ROW(a,b) < ROW(c,d)的比较会被实现为a < c AND b < d，而结果行为等价于a < c OR (a = c AND b < d)。

row_constructor IS DISTINCT FROM row_constructor

这个结构与<>行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

row_constructor IS NOT DISTINCT FROM row_constructor

这个结构与=行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

9.23.6. 组合类型比较

record operator record

SQL 规范要求在结果依赖于比较两个 NULL 值或者一个 NULL 与一个非 NULL 时逐行比较返回 NULL。PostgreSQL只有在比较两个行构造器（如第 9.23.5 节的结果或者比较一个行构造器与一个子查询的输出时才这样做（如第 9.22 节所述）。在其他比较两个组合类型值的环境中，两个 NULL 域值被认为相等，并且一个 NULL 被认为大于一个非 NULL。为了得到组合类型的一致的排序和索引行为，这样做是必要的。

每一边都会被计算并且它们会被逐行比较。当操作符是 =、 <>、 <、 <=、 >或者 >=时或者具有与这些类似的语义时，允许组合类型的比较（更准确地说，如果一个操作符是一个 B 树操作符类的成员，或者是一个 B 树操作符类的=成员的否定词，它就可以是一个行比较操作符）。 上述操作符的行为与用于行构造器（见第 9.23.5 节的IS [NOT] DISTINCT FROM相同。

为了支持包含无默认 B 树操作符类的元素的行匹配，为组合类型比较定义了下列操作符：*=、 *<>、 *<、 *<=、 *>以及 *>=。 这些操作符比较两行的内部二进制表达。即使两行用相等操作符的比较为真，两行也可能 具有不同的二进制表达。行在这些比较操作符之下的排序是决定性的，其他倒没什么意义。 这些操作符在内部被用于物化视图并且可能对其他如复制之类的特殊功能有用，但是它们 并不打算用在书写查询这类普通用途中。

9.24. 集合返回函数

本节描述那些可能返回多于一行的函数。目前这个类中被使用最广泛的是级数生成函数，如表 9.58和表 9.59所述。其他更特殊的集合返回函数在本手册的其他地方描述。 组合多集合返回函数的方法可见第 7.2.1.4 节

表 9.58. 级数生成函数

函数	参数类型	返回类型	描述
generate_series(start, stop)	int、bigint或者numeric	setof int、setof bigint或者setof numeric（与参数类型相同）	产生一系列值，从start到stop，步长为 1
generate_series(start, stop, step)	int、bigint或者numeric	setof int、setof bigint或者setof numeric（与参数类型相同）	产生一系列值，从start到stop，步长为step
generate_series(start, stop, step interval)	timestamp或timestamp with time zone	setof timestamp或setof timestamp with time zone（和参数类型相同）	产生一系列值，从start到stop，步长为step

当step为正时，如果start大于stop则返回零行。相反，当step为负时，如果start小于stop则返回零行。对于NULL输入也会返回零行。step为零是一个错误。下面是一些例子：

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
```

```
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- 这个例子依赖于日期+整数操作符
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

表 9.59. 下标生成函数

函数	返回类型	描述
generate_subscripts(array anyarray, dim int)	setof int	生成一个级数组成给定数组的下标。
generate_subscripts(array anyarray, dim int, reverse boolean)	setof int	生成一个级数组成给定数组的下标。当reverse为真，级数以逆序返回。

generate_subscripts是一个快捷函数，它为给定数组的指定维度生成一组合法的下标。对于不具有请求维度的数组返回零行，对于 NULL 数组也返回零行（但是会对 NULL 数组元素返回合法的下标）。下面是一些例子：

```
-- 基本使用
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- 表示一个数组，下标和被下标的值需要一个子查询
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- 平面化一个 2D 数组
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

当FROM子句中的一个函数后面有WITH ORDINALITY时，输出中会追加一个bigint列，它的值从1开始并且该函数输出的每一行加1。这在unnest()之类的集合返回函数中最有用。

```
-- set returning function WITH ORDINALITY
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
```

```
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_xact         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_wal          | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
 pg_subtrans     | 19
(19 rows)
```

9.25. 系统信息函数

表 9.60展示了多个可以抽取会话和系统信息的函数。

除了本节列出的函数，还有一些与统计系统相关的函数也提供系统信息。详见第 28.2.2 节

表 9.60. 会话信息函数

名称	返回类型	描述
current_catalog	name	当前数据库名（SQL 标准中称作“目录”）
current_database()	name	当前数据库名
current_query()	text	当前正在执行的查询的文本，和客户端提交的一样（可能包含多于一个语句）
current_role	name	等效于current_user
current_schema[()]	name	当前模式名
current_schemas(boolean)	name[]	搜索路径中的模式名，可以选择是否包含隐式模式
current_user	name	当前执行上下文的用户名
inet_client_addr()	inet	远程连接的地址
inet_client_port()	int	远程连接的端口
inet_server_addr()	inet	本地连接的地址
inet_server_port()	int	本地连接的端口
pg_backend_pid()	int	与当前会话关联的服务器进程的进程 ID
pg_blocking_pids(int)	int[]	阻塞指定服务器进程ID获得锁的进程 ID
pg_conf_load_time()	timestamp with time zone	配置载入时间

名称	返回类型	描述
pg_current_logfile([text])	text	当前日志收集器在使用的主日志文件名或者所要求格式的日志的文件名
pg_my_temp_schema()	oid	会话的临时模式的 OID，如果没有则为 0
pg_is_other_temp_schema(oid)	boolean	模式是另一个会话的临时模式吗？
pg_jit_available()	boolean	这个会话中JIT编译是否可用（见第 32 章？如果jit被设置为假，则返回false。
pg_listening_channels()	setof text	会话当前正在监听的频道名称
pg_notification_queue_usage()	double	异步通知队列当前被占用的分数（0-1）
pg_postmaster_start_time()	timestamp with time zone	服务器启动时间
pg_safe_snapshot_blocking_pids(int)	int[]	阻止指定服务器进程ID获取安全快照的进程ID
pg_trigger_depth()	int	PostgreSQL触发器的当前嵌套层次（如果没有调用则为 0，直接或间接，从一个触发器内部开始）
session_user	name	会话用户名
user	name	等价于current_user
version()	text	PostgreSQL版本信息。机器可读的版本还可见server_version_num。

注意

current_catalog、current_role、current_schema、current_user、session_user和user在SQL里有特殊的语意状态：它们被调用时结尾不要跟着园括号（在 PostgreSQL 中，圆括号可以有选择性地被用于current_schema，但是不能和其他的一起用）。

session_user通常是发起当前数据库连接的用户，不过超级用户可以用SET SESSION AUTHORIZATION修改这个设置。current_user是用于权限检查的用户标识。通常，它总是等于会话用户，但是可以被SET ROLE改变。它也会在函数执行的过程中随着属性SECURITY DEFINER的改变而改变。在 Unix 的说法里，那么会话用户是“真实用户”，而当前用户是“有效用户”。current_role以及user是current_user的同义词（SQL标准在current_role和current_user之间做了区分，但PostgreSQL不区分，因为它把用户和角色统一成了一种实体）。

current_schema返回在搜索路径中的第一个模式名（如果搜索路径是空则返回空值）。 如果创建表或者其它命名对象时没有声明目标模式，那么它将是被用于这些对象的模式。current_schemas(boolean)返回一个在搜索路径中出现的所有模式名的数组。布尔选项决定pg_catalog这样的隐式包含的系统模式是否包含在返回的搜索路径中。

注意

搜索路径可以在运行时修改。命令是：

```
SET search_path TO schema [, schema, ...]
```

inet_client_addr返回当前客户端的 IP 地址，inet_client_port返回它的端口号。inet_server_addr返回接受当前连接的服务器的 IP 地址，而inet_server_port返回对应的端口号。如果连接是通过 Unix 域套接字进行的，那么所有这些函数都返回 NULL。

pg_blocking_pids返回一个进程 ID 的数组，数组中的进程中的会话阻塞了指定进程 ID 所代表的服务器进程，如果指定的服务器进程不存在或者没有被阻塞则返回空数组。如果一个进程持有与另一个进程加锁请求冲突的锁（硬锁），或者前者正在等待一个与后者加锁请求冲突的锁并且前者在该锁的等待队列中位于后者的前面（软锁），则前者会阻塞后者。在使用并行查询时，这个函数的结果总是会列出客户端可见的进程 ID（即pg_backend_pid的结果），即便实际的锁是由工作者进程所持有或者等待也是如此。这样造成的后果是，结果中可能会有很多重复的 PID。还要注意当一个预备事务持有一个冲突锁时，这个函数的结果中它将被表示为一个为零的进程 ID。对这个函数的频繁调用可能对数据库性能有一些影响，因为它需要短时间地独占访问锁管理器的共享状态。

pg_conf_load_time返回服务器配置文件最近被载入的timestamp with time zone（如果当前会话在那时就已经存在，这个值将是该会话自己重新读取配置文件的时间，因此在不同的会话中这个读数会有一点变化。如果不是这样，这个值就是 postmaster 进程重读配置文件的时间）。

pg_current_logfile以text类型返回当前被日志收集器使用的日志文件的路径。该路径包括log_directory目录和日志文件名。日志收集必须被启用，否则返回值为NULL。当多个日志文件存在并且每一个都有不同的格式时，不带参数调用pg_current_logfile会返回这样的文件的路径：在所有的文件中，没有任何文件的格式在列表stderr、csvlog中排在这个文件的格式前面。如果没有任何日志文件有上述格式，则返回NULL。要请求一种特定的文件格式，可以以text将csvlog或者stderr作为可选参数的值。当所请求的日志格式不是已配置的log_destination时，会返回NULL。pg_current_logfile反映了current_logfiles文件的内容。

pg_my_temp_schema返回当前会话临时模式的 OID，如果没有使用临时模式（因为它没有创建任何临时表）则返回零。如果给定的 OID 是另一个会话的临时模式的 OID，则pg_is_other_temp_schema返回真（这是有用的，例如，要将其他会话的临时表从一个目录显示中排除）。

pg_listening_channels返回当前会话正在监听的异步通知频道的名称的集合。pg_notification_queue_usage返回等待处理的通知占可用的通知空间的比例，它是一个 0-1 范围内的double值。详见LISTEN和NOTIFY。

pg_postmaster_start_time返回服务器启动的timestamp with time zone。

pg_safe_snapshot_blocking_pids一个进程ID的数组，它们代表阻止指定进程ID对应的服务器进程获取安全快照的会话，如果没有这类服务器进程或者它没有被阻塞，则会返回一个空数组。一个运行着SERIALIZABLE事务的会话会阻止SERIALIZABLE READ ONLY DEFERRABLE事务获取快照，直到后者确定避免拿到任何谓词锁是安全的。更多有关可序列化以及可延迟事务的信息请参考第 13.2.3 节频繁调用这个函数可能会对数据库性能产生一些影响，因为它需要短时间访问谓词锁管理器的共享状态。

version返回一个描述PostgreSQL服务器版本的字符串。你也可以从server_version或者一个机器可读的版本server_version_num得到这个信息。软件开发者应该使用server_version_num（从 8.2 开始可用）或者 PQserverVersion ，而不必解析文本形式的版本。

表 9.67 列出那些允许用户编程查询对象访问权限的函数。参阅第 5.6 节获取更多有关权限的信息。

表 9.61. 访问权限查询函数

名称	返回类型	描述
has_any_column_privilege(user, table, privilege)	boolean	用户有没有表中任意列上的权限
has_any_column_privilege(table, privilege)	boolean	当前用户有没有表中任意列上的权限
has_column_privilege(user, table, column, privilege)	boolean	用户有没有列的权限
has_column_privilege(table, column, privilege)	boolean	当前用户有没有列的权限
has_database_privilege(user, database, privilege)	boolean	用户有没有数据库的权限
has_database_privilege(database, privilege)	boolean	当前用户有没有数据库的权限
has_foreign_data_wrapper_privilege(user, fdw, privilege)	boolean	用户有没有外部数据包装器上的权限
has_foreign_data_wrapper_privilege(fdw, privilege)	boolean	当前用户有没有外部数据包装器上的权限
has_function_privilege(user, function, privilege)	boolean	用户有没有函数上的权限
has_function_privilege(function, privilege)	boolean	当前用户有没有函数上的权限
has_language_privilege(user, language, privilege)	boolean	用户有没有语言上的权限
has_language_privilege(language, privilege)	boolean	当前用户有没有语言上的权限
has_schema_privilege(user, schema, privilege)	boolean	用户有没有模式上的权限
has_schema_privilege(schema, privilege)	boolean	当前用户有没有模式上的权限
has_sequence_privilege(user, sequence, privilege)	boolean	用户有没有序列上的权限
has_sequence_privilege(sequence, privilege)	boolean	当前用户有没有序列上的权限
has_server_privilege(user, server, privilege)	boolean	用户有没有外部服务器上的权限
has_server_privilege(server, privilege)	boolean	当前用户有没有外部服务器上的权限
has_table_privilege(user, table, privilege)	boolean	用户有没有表上的权限
has_table_privilege(table, privilege)	boolean	当前用户有没有表上的权限
has_tablespace_privilege(user, tablespace, privilege)	boolean	用户有没有表空间上的权限
has_tablespace_privilege(tablespace, privilege)	boolean	当前用户有没有表空间上的权限

名称	返回类型	描述
has_type_privilege(user, type, privilege)	boolean	用户有没有类型的特权
has_type_privilege(type, privilege)	boolean	当前用户有没有类型的特权
pg_has_role(user, role, privilege)	boolean	用户有没有角色上的权限
pg_has_role(role, privilege)	boolean	当前用户有没有角色上的权限
row_security_active(table)	boolean	当前用户是否在表上开启了行级安全性

has_table_privilege判断一个用户是否可以用某种特定的方式访问一个表。 该用户可以通过名字或者 OID （pg_authid.oid） 来指定，也可以用public表示 PUBLIC 伪角色。如果省略该参数，则使用current_user。 该表可以通过名字或者 OID 指定（因此，实际上有六种 has_table_privilege的变体，我们可以通过它们的参数数目和类型来区分它们） 。如果用名字指定，那么在必要时该名字可以是模式限定的。 所希望的权限类型是用一个文本串来指定的，它必须是下面的几个值之一： SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER。WITH GRANT OPTION可以被选择增加到一个权限类型来测试是否该权限是使用转授选项得到。另外，可以使用逗号分隔来列出多个权限类型，在这种情况下只要具有其中之一的权限则结果为真（权限字符串的大小写并不重要，可以在权限名称之间出现额外的空白，但是在权限名内部不能有空白）。一些例子：

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT
 OPTION');
```

has_sequence_privilege检查一个用户是否能以某种特定方式访问一个序列。它的参数可能性和has_table_privilege相似。所希望测试的访问权限类型必须是下列之一：USAGE、SELECT或UPDATE。

has_any_column_privilege检查一个用户是否能以特定方式访问一个表的任意列。其参数可能性和has_table_privilege类似，除了所希望的访问权限类型必须是下面值的某种组合：SELECT、INSERT、UPDATE或REFERENCES。注意在表层面上具有这些权限的任意一个都会隐式地把它授权给表中的每一列，因此如果has_table_privilege对同样的参数返回真则has_any_column_privilege将总是返回真。但是如果在至少一列上有一个该权限的列级授权，has_any_column_privilege也会成功。

has_column_privilege检查一个用户是否能以特定方式访问一个列。它的参数可能性与has_table_privilege类似，并且列还可以使用名字或者属性号来指定。希望的访问权限类型必须是下列值的某种组合：SELECT、INSERT、UPDATE或REFERENCES。注意在表级别上具有这些权限中的任意一种将会隐式地把它授予给表上的每一列。

has_database_privilege检查一个用户是否能以特定方式访问一个数据库。它的参数可能性类似 has_table_privilege。希望的访问权限类型必须是以下值的某种组合：CREATE、CONNECT、TEMPORARY或TEMP（等价于TEMPORARY）。

has_function_privilege检查一个用户是否能以特定方式访问一个函数。其参数可能性类似has_table_privilege。在用一个文本串而不是 OID 指定一个函数时，允许的输入和regprocedure数据类型一样（参阅 第 8.19 节。希望的访问权限类型必须是EXECUTE。一个例子：

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege检查一个用户是否能以特定方式访问一个外部数据包装器。它的参数可能性类似于has_table_privilege。希望的访问权限类型必须是USAGE。

has_language_privilege检查一个用户是否可以以某种特定的方式访问一个过程语言。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是USAGE。

has_schema_privilege检查一个用户是否可以以某种特定的方式访问一个模式。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是CREATE或USAGE。

has_server_privilege检查一个用户是否可以以某种特定的方式访问一个外部服务器。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是USAGE。

has_tablespace_privilege检查一个用户是否可以以某种特定的方式访问一个表空间。其参数可能性类似 has_table_privilege。希望的访问权限类型必须是CREATE。

has_type_privilege检查一个用户是否能以特定的方式访问一种类型。其参数的可能性类同于has_table_privilege。在用字符串而不是 OID 指定类型时，允许的输入和regtype数据类型相同（见第 8.19 节。期望的访问特权类型必须等于USAGE。

pg_has_role检查一个用户是否可以以某种特定的方式访问一个角色。其参数可能性类似 has_table_privilege，除了public不能被允许作为一个用户名。希望的访问权限类型必须是下列值的某种组合：MEMBER或USAGE。MEMBER表示该角色中的直接或间接成员关系（即使用SET ROLE的权力），而USAGE表示不做SET ROLE的情况下该角色的权限是否立即可用。

row_security_active检查在 current_user的上下文和环境中是否为指定的 表激活了行级安全性。表可以用名称或者 OID 指定。

表 9.62 展示了决定是否一个特定对象在当前模式搜索路径中可见的函数。例如，如果一个表所在的模式在当前搜索路径中并且在它之前没有出现过相同的名字，这个表就被说是可见的。这等价于在语句中表可以被用名称引用但不加显式的模式限定。要列出所有可见表的名字：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

表 9.62. 模式可见性查询函数

名称	返回类型	描述
pg_collation_is_visible(collation_oid)	boolean	排序规则在搜索路径中可见吗？
pg_conversion_is_visible(conversion_oid)	boolean	转换在搜索路径中可见吗？
pg_function_is_visible(function_oid)	boolean	函数在搜索路径中可见吗？
pg_opclass_is_visible(opclass_oid)	boolean	操作符类在搜索路径中可见吗？
pg_operator_is_visible(operator_oid)	boolean	操作符在搜索路径中可见吗？
pg_opfamily_is_visible(opclass_oid)	boolean	操作符族在搜索路径中可见吗？
pg_statistics_obj_is_visible(stat_oid)	boolean	是搜索路径中的统计信息对象
pg_table_is_visible(table_oid)	boolean	表在搜索路径中可见吗？
pg_ts_config_is_visible(config_oid)	boolean	文本搜索配置在搜索路径中可见吗？
pg_ts_dict_is_visible(dict_oid)	boolean	文本搜索字典在搜索路径中可见吗？

名称	返回类型	描述
pg_ts_parser_is_visible(parser_oid)	boolean	文本搜索解析器在搜索路径中可见吗？
pg_ts_template_is_visible(template_oid)	boolean	文本搜索模板在搜索路径中可见吗？
pg_type_is_visible(type_oid)	boolean	类型（或域）在搜索路径中可见吗？

每一个函数对一种数据库对象执行可见性检查。注意pg_table_is_visible也可被用于视图、物化视图、索引、序列和外部表，pg_function_is_visible也能被用于过程和聚集，pg_type_is_visible也可以被用于域。对于函数和操作符，如果在路径中更早的地方没有出现具有相同名称和参数数据类型的对象，该对象在搜索路径中是可见的。对于操作符类，名称和相关的索引访问方法都要考虑。

所有这些函数都要求用对象 OID 来标识将被检查的对象。如果你想用名称来测试一个对象，使用 OID 别名类型（regclass、regtype、regprocedure、regoperator、regconfig或regdictionary）将会很方便。例如：

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

注意以这种方式测试一个非模式限定的类型名没什么意义 — 如果该名称完全能被识别，它必须是可见的。

表 9.63列出了从系统目录抽取信息的函数。

表 9.63. 系统目录信息函数

名称	返回类型	描述
format_type(type_oid, typemod)	text	获得一个数据类型的 SQL 名字
pg_get_constraintdef(constraint_oid)	text	获得一个约束的定义
pg_get_constraintdef(constraint_oid, pretty_bool)	text	获得一个约束的定义
pg_get_expr(pg_node_tree, relation_oid)	text	反编译一个表达式的内部形式，假定其中的任何 Var 指向由第二个参数指示的关系
pg_get_expr(pg_node_tree, relation_oid, pretty_bool)	text	反编译一个表达式的内部形式，假定其中的任何 Var 指向由第二个参数指示的关系
pg_get_functiondef(func_oid)	text	获得一个函数或过程的定义
pg_get_function_arguments(func_oid)	text	获得一个函数或过程定义的参数列表（带有默认值）
pg_get_function_identity_arguments(func_oid)	text	获得标识一个函数或过程的参数列表（不带默认值）
pg_get_function_result(func_oid)	text	获得函数的RETURNS子句（对过程返回空）
pg_get_indexdef(index_oid)	text	获得索引的CREATE INDEX命令
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	获得索引的CREATE INDEX命令，或者当column_no为非零时只得到一个索引列的定义

名称	返回类型	描述
pg_get_keywords()	setof record	获得 SQL 关键字的列表及其分类
pg_get_ruledef(rule_oid)	text	获得规则的CREATE RULE命令
pg_get_ruledef(rule_oid, pretty_bool)	text	获得规则的CREATE RULE命令
pg_get_serial_sequence(table_name, column_name)	text	获得一个序列列或标识列使用的序列的名称
pg_get_statisticsobjdef(stat_obj_oid)	text	为扩展的统计信息对象得到CREATE STATISTICS命令
pg_get_triggerdef(trigger_oid)	text	获得触发器的CREATE [CONSTRAINT] TRIGGER命令
pg_get_triggerdef(trigger_oid, pretty_bool)	text	获得触发器的CREATE [CONSTRAINT] TRIGGER命令
pg_get_userbyid(role_oid)	name	获得给定 OID 指定的角色名
pg_get_viewdef(view_name)	text	获得视图或物化视图的底层SELECT命令（已废弃）
pg_get_viewdef(view_name, pretty_bool)	text	获得视图或物化视图的底层SELECT命令（已废弃）
pg_get_viewdef(view_oid)	text	获得视图或物化视图的底层SELECT命令
pg_get_viewdef(view_oid, pretty_bool)	text	获得视图或物化视图的底层SELECT命令
pg_get_viewdef(view_oid, wrap_column_int)	text	获得视图或物化视图的底层SELECT命令；带域的行被包装成指定的列数，并隐含了优质打印
pg_index_column_has_property(index_oid, column_no, prop_name)	boolean	测试一个索引列是否有指定的性质
pg_index_has_property(index_oid, prop_name)	boolean	测试一个索引是否有指定的性质
pg_indexam_has_property(am_oid, prop_name)	boolean	测试一个索引访问方法是否有指定的性质
pg_options_to_table(reloptions)	setof record	获得存储选项的名称/值对的集合
pg_tablespace_databases(tablespace_oid)	setof oid	获得在该表空间中有对象的数据库的 OID 的集合
pg_tablespace_location(tablespace_oid)	text	获得这个表空间所在的文件系统的路径
pg_typeof(any)	regtype	获得任意值的数据类型
collation for (any)	text	获得该参数的排序规则
to_regclass(rel_name)	regclass	得到指定关系的 OID
to_regproc(func_name)	regproc	得到指定函数的 OID
to_regprocedure(func_name)	regprocedure	得到指定函数的 OID
to_regoper(operator_name)	regoper	得到指定操作符的 OID
to_regoperator(operator_name)	regoperator	得到指定操作符的 OID
to_regtype(type_name)	regtype	得到指定类型的 OID

名称	返回类型	描述
to_regnamespace(schema_name)	regnamespace	得到指定模式的 OID
to_regrole(role_name)	regrole	得到指定角色的 OID

format_type返回一个数据类型的 SQL 名称，它由它的类型 OID 标识并且可能是一个类型修饰符。如果不知道相关的修饰符，则为类型修饰符传递 NULL。

pg_get_keywords返回一组记录描述服务器识别的 SQL 关键字。word列包含关键字。catcode列包含一个分类码：U为未被预定，C 为列名，T类型或函数名，R为预留。catdesc列包含一个可能本地化的描述分类的字符串。

pg_get_constraintdef、pg_get_indexdef、pg_get_ruledef、pg_get_statisticsobjdef和pg_get_triggerd别重建一个约束、索引、规则、扩展统计对象或触发器的创建命令（注意这是一个反编译的重构，而不是命令的原始文本）。pg_get_expr反编译一个表达式的内部形式，例如一个列的默认值。在检查系统目录内容时有用。如果表达式可能包含 Var，在第二个参数中指定它们引用的关系的 OID；如果不会出现 Var，第二个参数设置为 0 即可。pg_get_viewdef重构定义一个视图的SELECT查询。这些函数的大部分都有两种变体，一种可以可选地“优质打印”结果。优质打印的格式可读性更强，但是默认格式更可能被未来版本的PostgreSQL以相同的方式解释。在转出目的中避免使用优质打印输出。为优质打印参数传递假将得到和不带该参数的变体相同的结果。

pg_get_functiondef为一个函数返回一个完整的CREATE OR REPLACE FUNCTION语句。pg_get_function_arguments返回一个函数的参数列表，形式按照它们出现在CREATE FUNCTION中的那样。pg_get_function_result类似地返回函数的合适的RETURNS子句。pg_get_function_identity_arguments返回标识一个函数必要的参数列表，形式和它们出现在ALTER FUNCTION中的一样。这种形式忽略默认值。

pg_get_serial_sequence返回与一个列相关联的序列的名称，如果与列相关联的序列则返回NULL。如果该列是一个标识列，相关联的序列是为该标识列内部创建的序列。对于使用序列类型之一（serial、smallserial、bigserial）创建的列，它是为那个序列列定义创建的序列。在后一种情况中，这种关联可以用ALTER SEQUENCE OWNED BY修改或者移除（该函数可能应该已经被pg_get_owned_sequence调用，它当前的名称反映了它通常被serial或bigserial列使用）。第一个输入参数是一个带可选模式的表名，第二个参数是一个列名。因为第一个参数可能是一个模式和表，它不能按照一个双引号包围的标识符来对待，意味着它默认情况下是小写的。而第二个参数只是一个列名，将被当作一个双引号包围的来处理并且会保留其大小写。函数返回的值会被适当地格式化以便传递给序列函数（参见第 9.16 节。一种典型的用法是为标识列或者序列列读取当前值，例如：

```
SELECT currval(pg_get_serial_sequence('sometable', 'id'));
```

pg_get_userbyid抽取给定 OID 的角色的名称。

pg_index_column_has_property、pg_index_has_property和pg_indexam_has_property返回指定的索引列、索引或者索引访问方法是否具有指定性质。如果性质的名称找不到或者不适用于特定的对象，亦或者 OID 或者列号不表示合法的对象，则返回NULL。列的性质可参见表 9.64 索引的性质可参见表 9.65 访问方法的性质可参见表 9.66（注意扩展访问方法可以为其索引定义额外的性质）。

表 9.64. 索引列属性

名称	描述
asc	在向前扫描时列是按照升序排列吗？
desc	在向前扫描时列是按照降序排列吗？
nulls_first	在向前扫描时列排序会把空值排在前面吗？
nulls_last	在向前扫描时列排序会把空值排在最后吗？
orderable	列具有已定义的排序顺序吗？

名称	描述
distance_orderable	列能否通过一个“distance”操作符（例如ORDER BY col <-> constant）有序地扫描？
returnable	列值是否可以通过一次只用索引扫描返回？
search_array	列是否天然支持col = ANY(array)搜索？
search_nulls	列是否支持IS NULL和IS NOT NULL搜索？

表 9.65. 索引性质

名称	描述
clusterable	索引是否可以用于CLUSTER命令？
index_scan	索引是否支持普通扫描（非位图）？
bitmap_scan	索引是否支持位图扫描？
backward_scan	在扫描中扫描方向能否被更改（为了支持游标上无需物化的FETCH BACKWARD）？

表 9.66. 索引访问方法性质

名称	描述
can_order	访问方法是否支持ASC、DESC以及CREATE INDEX中的有关关键词？
can_unique	访问方法是否支持唯一索引？
can_multi_col	访问方法是否支持多列索引？
can_exclude	访问方法是否支持排除约束？
can_include	访问方法是否支持CREATE INDEX的INCLUDE子句？

当传入pg_class.reloptions或pg_attribute.attoptions时，pg_options_to_table返回存储选项名称/值对（option_name/option_value）的集合。

pg_tablespace_databases允许一个表空间被检查。它返回一组数据库的 OID，这些数据库都有对象存储在该表空间中。如果这个函数返回任何行，则该表空间为非空并且不能被删除。为了显示该表空间中的指定对象，你将需要连接到pg_tablespace_databases标识的数据库并且查询它们的pg_class目录。

pg_typeof返回传递给它的值的数据类型的 OID。这在检修或者动态构建 SQL 查询时有用。函数被声明为返回regtype，它是一个 OID 别名类型（见第 8.19 节；这表明它和一个用于比较目的的 OID 相同，但是作为一个类型名称显示。例如：

```
SELECT pg_typeof(33);

 pg_typeof
------------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

表达式collation for返回传递给它的值的排序规则。例子：

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
------------------
 "de_DE"
(1 row)
```

值可能被加上引号并且变成模式限定的。如果从参数表达式得不到排序规则，则返回一个空值。如果参数不是一个可排序的数据类型，则抛出一个错误。

to_regclass、to_regproc、to_regprocedure, to_regoper、to_regoperator、to_regtype、to_regnamespace和to_regrole函数把关系、函数、操作符、类型、模式和角色的名称（以text给出）分别转换成、regclass、regproc、regprocedure、regoper、regoperator、regtype、regnamespace和regrole对象。这些函数与 text 转换的不同在于它们不接受数字 OID，并且在名称无法找到时不会抛出错误而是返回空。对于to_regproc和to_regoper，如果给定名称匹配多个对象时返回空。

表 9.67列出了与数据库对象 标识和定位有关的函数。

表 9.67. 对象信息和定位函数

名称	返回类型	描述
pg_describe_object(catalog_id oid, object_id oid, object_sub_id integer)	text	得到一个数据库对象的描述
pg_identify_object(catalog_id oid, object_id oid, object_sub_id integer)	type text, schema text, name text, identity text	得到一个数据库对象的标识
pg_identify_object_as_address(catalog_id oid, object_id oid, object_sub_id integer)	type text, name text[], args text[]	得到一个数据库对象的地址的外部表示
pg_get_object_address(type text, name text[], args text[])	class_id oid, object_id oid, object_sub_id int32	从一个数据库对象的内部表示得到它的地址

pg_describe_object返回由目录OID、对象OID以及子对象ID（例如表中的一个列号，当子对象引用了一整个对象时其ID为零）指定的数据库对象的文本描述。这种描述是为 了人类可读的，并且可能是被翻译过的，具体取决于服务器配置。这有助于确定一 个存储在pg_depend目录中的对象的标识。

pg_identify_object返回一行，其中包含有足以唯一标识 由目录 OID、对象 OID 和一个（可能为零的）子对象 ID 指定的数据库对象的信息。 该信息是共机器读取的，并且不会被翻译。type标识数据库对象 的类型；schema是该对象所属的模式名，如果对象类型不属于 模式则为NULL；如果名称（加上方案名，如果相关）足以唯一标识对象，则name就是对象的名称（必要时会被加上引号），否则为NULL；identity是完整的对象标识， 它会表现为与对象类型相关的精确格式，并且如有必要，该格式中的每个部分都会 被模式限定。

pg_identify_object_as_address返回一行，其中包含有 足以唯一标识由目录 OID、对象 OID 和一个（可能为零的）子对象 ID 指定的数据 库对象的信息。返回的信息独立于当前服

务器，也就是说，它可以被用来在另一个 服务器中标识一个具有相同命名的对象。type标识数据库对象 的类型；object_names和object_args是文本数组，它们一起 构成了对对象的引用。这三个值可以被传递给 pg_get_object_address以获得该对象的内部地址。这个函数是 pg_get_object_address的逆函数。

pg_get_object_address返回一行，其中包含有足以唯一 标识由类型、对象名和参数数组指定的数据库对象的信息。返回值可以被用在诸如 pg_depend等系统目录中并且可以被传递给pg_identify_object或pg_describe_object等其他 系统函数。class_id是包含该对象的系统目录 OID； objid是对象本身的 OID，而 objsubid是子对象 ID，如果没有则为零。这个函数是 pg_identify_object_as_address的逆函数。

表 9.68中展示的函数抽取注释，注释是由COMMENT命令在以前存储的。如果对指定参数找不到注释，则返回空值。

表 9.68. 注释信息函数

名称	返回类型	描述
col_description(table_oid, column_number)	text	为一个表列获得注释
obj_description(object_oid, catalog_name)	text	为一个数据库对象获得注释
obj_description(object_oid)	text	为一个数据库对象获得注释（已被废弃）
shobj_description(object_oid, catalog_name)	text	为一个共享数据库对象获得注释

col_description为一个表列返回注释，该表列由所在表的 OID 和它的列号指定（obj_description不能被用在表列，因为表列没有自己的 OID）。

obj_description的双参数形式返回一个由其 OID 和所在系统目录名称指定的数据库对象的注释。例如，obj_description(123456,'pg_class')将会检索出 OID 为123456的表的注释。obj_description的单参数形式只要求对象 OID。它已经被废弃，因为无法保证 OID 在不同系统目录之间是唯一的；这样可能会返回错误的注释。

shobj_description用起来就像obj_description，但是前者是用于检索共享对象上的注释。某些系统目录对于一个集簇中的所有数据库是全局的，并且其中的对象的描述也是全局存储的。

表 9.69中展示的函数以一种可导出的形式提供了服务器事务信息。这些函数的主要用途是判断在两个快照之间哪些事务被提交。

表 9.69. 事务 ID 和快照

名称	返回类型	描述
txid_current()	bigint	获得当前事务 ID，如果当前事务没有 ID 则分配一个新的 ID
txid_current_if_assigned()	bigint	与txid_current()相同，但是在事务没有分配ID时是返回空值而不是分配一个新的事务ID
txid_current_snapshot()	txid_snapshot	获得当前快照
txid_snapshot_xip(txid_snapshot)	setof bigint	获得快照中正在进行的事务 ID

名称	返回类型	描述
txid_snapshot_xmax(txid_snapshot)	bigint	获得快照的xmax
txid_snapshot_xmin(txid_snapshot)	bigint	获得快照的xmin
txid_visible_in_snapshot(bigint, txid_snapshot)	boolean	事务 ID 在快照中可见吗？（不能用于子事务 ID）
txid_status(bigint)	text	报告给定事务的状态：committed、aborted、in progress，如果事务ID太老则为空值

内部事务 ID 类型（xid）是 32 位宽并且每 40 亿个事务就会回卷。但是，这些函数导出一种 64 位格式，它被使用一个“世代”计数器，这样在一个安装的生命期内不会回卷。这些函数使用的数据类型txid_snapshot存储了在一个特定时刻有关事务 ID 可见性的信息。它的成分在表 9.70中描述。

表 9.70. 快照成分

名称	描述
xmin	仍然活动的最早的事务 ID （txid）。所有更早的事务要么已经被提交并且可见，要么已经被回滚并且死亡。
xmax	第一个还未分配的 txid。所有大于等于它的 txid 在快照的时刻还没有开始，并且因此是不可见的。
xip_list	在快照时刻活动的 txid。这个列表只包括那些位于xmin和xmax之间的活动 txid；可能有活动的超过xmax的 txid。一个满足xmin <= txid < xmax并且不在这个列表中的 txid 在快照时刻已经结束，并且因此根据其提交状态要么可见要么死亡。该列表不包括子事务的 txid。

txid_snapshot的文本表示是xmin:xmax:xip_list。例如10:20:10,14,15表示xmin=10, xmax=20, xip_list=10, 14, 15。

txid_status(bigint)报告一个近期事务的提交状态。当一个应用和数据库服务器的连接在COMMIT正在进行时断开，应用可以用它来判断事务是提交了还是中止了。一个事务的状态将被报告为in progress、committed或者aborted，前提是该事务的发生时间足够近，这样系统才会保留它的提交状态。如果事务太老，则系统中不会留下对该事务的引用并且提交状态信息也已经被抛弃，那么这个函数将会返回NULL。注意，预备事务会被报告为in progress，如果应用需要判断该txid是否是一个预备事务，应用必须检查pg_prepared_xacts。

表 9.71中展示的函数提供了有关于 已经提交事务的信息。这些函数主要提供有关事务何时被提交的信息。只有当 track_commit_timestamp配置选项被启用时它们才能 提供有用的数据，并且只对已提交事务提供数据。

表 9.71. 已提交事务信息

名称	返回类型	描述
pg_xact_commit_timestamp(xid)	timestamp with time zone	得到一个事务的提交时间戳
pg_last_committed_xact()	xid xid, timestamp timestamp with time zone	得到最后一个已提交事务的事务 ID 和提交时间戳

表 9.72中所展示的函数能打印initdb期间初始化的信息，例如系统目录版本。它们也能显示有关预写式日志和检查点处理的信息。这些信息是集簇范围内的，不与任何特定的一个数据库相关。对于同一种来源，它们返回和pg_controldata大致相同的信息，不过其形式更适合于SQL函数。

表 9.72. 控制数据函数

名称	返回类型	描述
pg_control_checkpoint()	record	返回有关当前检查点状态的信息。
pg_control_system()	record	返回有关当前控制文件状态的信息。
pg_control_init()	record	返回有关集簇初始化状态的信息。
pg_control_recovery()	record	返回有关恢复状态的信息。

pg_control_checkpoint返回一个表 9.73中所示的记录

表 9.73.pg_control_checkpoint列

列名	数据类型
checkpoint_location	pg_lsn
redo_lsn	pg_lsn
redo_wal_file	text
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

pg_control_system返回一个表 9.74中所示的记录

表 9.74.pg_control_system列

列名	数据类型
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

pg_control_init返回一个表 9.75中所示的记录

表 9.75.pg_control_init列

列名	数据类型
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float4_pass_by_value	boolean
float8_pass_by_value	boolean
data_page_checksum_version	integer

pg_control_recovery返回一个表 9.76中所示的记录

表 9.76.pg_control_recovery列

列名	数据类型
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.26. 系统管理函数

这一节描述的函数被用来控制和监视一个PostgreSQL安装。

9.26.1. 配置设定函数

表 9.77展示了那些可以用于查询以及修改运行时配置参数的函数。

表 9.77. 配置设定函数

名称	返回类型	描述
current_setting(setting_name [, missing_ok])	text	获得设置的当前值
set_config(setting_name, new_value, is_local)	text	设置一个参数并返回新值

current_setting得到setting_name设置的当前值。它对应于SQL命令SHOW。一个例子：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

如果没有名为setting_name的设置，除非提供missing_ok并且其值为true，current_setting会抛出错误。

set_config将参数setting_name设置为new_value。如果 is_local设置为true，那么新值将只应用于当前事务。 如果你希望新值应用于当前会话，那么应该使用false。 它等效于 SQL 命令 SET。例如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

9.26.2. 服务器信号函数

在表 9.78中展示的函数向其它服务器进程发送控制信号。默认情况下这些函数只能被超级用户使用，但是如果需要，可以利用GRANT把访问特权授予给其他用户。

表 9.78. 服务器信号函数

名称	返回类型	描述
pg_cancel_backend(pid int)	boolean	取消一个后端的当前查询。如果调用角色是被取消后端的拥有者角色的成员或者调用角色已经被授予pg_signal_backend，这也是允许的，不过只有超级用户才能取消超级用户的后端。
pg_reload_conf()	boolean	导致服务器进程重载它们的配置文件
pg_rotate_logfile()	boolean	切换服务器的日志文件
pg_terminate_backend(pid int)	boolean	中止一个后端。如果调用角色是被取消后端的拥有者角色的成员或者调用角色已经被授予pg_signal_backend，这也是允许的，不过只有超级用户才能取消超级用户的后端。

这些函数中的每一个都在成功时返回true，并且在失败时返回false。

pg_cancel_backend和pg_terminate_backend向由进程 ID 标识的后端进程发送信号（分别是SIGINT或SIGTERM）。一个活动后端的进程 ID可以从pg_stat_activity视图的pid列中找到，或者通过在服务器上列出postgres进程（在 Unix 上使用ps或者在Windows上使用任务管理器）得到。一个活动后端的角色可以在pg_stat_activity视图的usename列中找到。

pg_reload_conf给服务器发送一个SIGHUP信号， 导致所有服务器进程重载配置文件。

pg_rotate_logfile给日志文件管理器发送信号，告诉它立即切换到一个新的输出文件。这个函数只有在内建日志收集器运行时才能工作，因为否则就不存在日志文件管理器子进程。subprocess.

9.26.3. 备份控制函数

表 9.79中展示的函数可以辅助制作在线备份。这些函数不能在恢复期间执行（pg_is_in_backup、pg_backup_start_time和pg_wal_lsn_diff除外）。

表 9.79. 备份控制函数

名称	返回类型	描述
pg_create_restore_point(name text)	pg_lsn	为执行恢复创建一个命名点（默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）
pg_current_wal_flush_lsn()	pg_lsn	得到当前的预写式日志刷写位置
pg_current_wal_insert_lsn()	pg_lsn	获得当前预写式日志插入位置
pg_current_wal_lsn()	pg_lsn	获得当前预写式日志写入位置
pg_start_backup(label text [, fast boolean [, exclusive boolean]])	pg_lsn	准备执行在线备份（默认只限于超级用户或者复制角色，但是可以授予其他用户 EXECUTE 特权来执行该函数）
pg_stop_backup()	pg_lsn	完成执行排他的在线备份（默认只限于超级用户或者复制角色，但是可以授予其他用户 EXECUTE 特权来执行该函数）
pg_stop_backup(exclusive boolean)	setof record	结束执行排他或者非排他的在线备份 （默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）
pg_is_in_backup()	bool	如果一个在线排他备份仍在进行中则为真。
pg_backup_start_time()	timestamp with time zone	获得一个进行中的在线排他备份的开始时间。
pg_switch_wal()	pg_lsn	强制切换到一个新的预写式日志文件（默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）
pg_walfile_name(lsn text)	pg_lsn	转换预写式日志位置字符串为文件名
pg_walfile_name_offset(lsn text)	pg_lsn, integer	转换预写式日志位置字符串为文件名以及文件内的十进制字节偏移
pg_wal_lsn_diff(lsn pg_lsn, lsn pg_lsn)	numeric	计算两个预写式日志位置间的差别

pg_start_backup接受一个参数，这个参数可以是备份的任意用户定义的标签（通常这是备份转储文件将被存储的名字）。当被用在排他模式中时，该函数向数据库集簇的数据目录写入一个备份标签文件（backup_label）和一个表空间映射文件（tablespace_map，如果在pg_tblspc/目录中有任何链接），执行一个检查点，然后以文本方式返回备份的起始预写式日志位置。用户可以忽略这个结果值，但是为了可能需要的场合我们还是提供该值。 当在非排他模式中使用时，这些文件的内容会转而由pg_stop_backup函数返回，并且应该由调用者写入到备份中去。

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

第二个参数是可选的，其类型为boolean。如果为true，它指定尽快执行pg_start_backup。这会强制一个立即执行的检查点，它会导致 I/O 操作的峰值，拖慢任何并发执行的查询。

在一次排他备份中，pg_stop_backup会移除标签文件以及pg_start_backup创建的tablespace_map文件（如果存在）。在一次非排他备份中，backup_label和tablespace_map的内容会包含在该函数返回的结果中，并且应该被写入到该备份的文件中（这些内容不在数据目录中）。有一个可选的boolean类型的第二参数。如果为假，pg_stop_backup将在备份完成后立即返回而不等待WAL被归档。这种行为仅对独立监控WAL归档的备份软件有用。否则，让备份一致所要求的WAL可能会丢失，进而让备份变得毫无用处。当这个参数被设置为真时，在启用归档的前提下pg_stop_backup将等待WAL被归档，在后备服务器上，这意味只有archive_mode = always时才会等待。如果主服务器上的写活动很低，在主服务器上运行pg_switch_wal以触发一次即刻的段切换会很有用。

当在主服务器上执行时，该函数还在预写式日志归档区里创建一个备份历史文件。这个历史文件包含给予pg_start_backup的标签、备份的起始与终止预写式日志位置以及备份的起始和终止时间。返回值是备份的终止预写式日志位置（同样也可以被忽略）。在记录结束位置之后，当前预写式日志插入点被自动地推进到下一个预写式日志文件，这样结束的预写式日志文件可以立即被归档来结束备份。

pg_switch_wal移动到下一个预写式日志文件，允许当前文件被归档（假定你正在使用连续归档）。返回值是在甘冈完成的预写式日志文件中结束预写式日志位置 + 1。如果从上一次预写式日志切换依赖没有预写式日志活动，pg_switch_wal不会做任何事情并且返回当前正在使用的预写式日志文件的开始位置。

pg_create_restore_point创建一个命名预写式日志记录，它可以被用作恢复目标，并且返回相应的预写式日志位置。这个给定的名字可以用于recovery_target_name来指定恢复要进行到的点。避免使用同一个名称创建多个恢复点，因为恢复会停止在第一个匹配名称的恢复目标。

pg_current_wal_lsn以上述函数所使用的相同格式显示当前预写式日志的写位置。类似地，pg_current_wal_insert_lsn显示当前预写式日志插入点，而pg_current_wal_flush_lsn显示当前预写式日志的刷写点。在任何情况下，插入点是预写式日志的“逻辑”终止点，而写入位置是已经实际从服务器内部缓冲区写出的日志的终止点，刷写位置则是被确保写入到持久存储中的日志的终止点。写入位置是可以从服务器外部检查的终止点，对那些关注归档部分完成预写式日志文件的人来说，这就是他们需要的位置。插入和刷写点主要是为了服务器调试目的而存在的。这些都是只读操作并且不需要超级用户权限。

你可以使用pg_walfile_name_offset从任何上述函数的结果中抽取相应的预写式日志文件名称以及字节偏移。例如：

```
postgres=# SELECT * FROM pg_walfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
```

```
 00000001000000000000000D |        4039624
(1 row)
```

相似地，pg_walfile_name只抽取预写式日志文件名。当给定的预写式日志位置正好在一个预写式日志文件的边界，这些函数都返回之前的预写式日志文件的名称。这对管理预写式日志归档行为通常是所希望的行为，因为前一个文件是当前需要被归档的最后一个文件。

pg_wal_lsn_diff以字节数计算两个预写式日志位置之间的差别。它可以和pg_stat_replication或表 9.79中其他的函数一起使用来获得复制延迟。

关于正确使用这些函数的细节，请见第 25.3 节。

9.26.4. 恢复控制函数

表 9.80中展示的函数提供有关后备机当前状态的信息。这些函数可以在恢复或普通运行过程中被执行。

表 9.80. 恢复信息函数

名称	返回类型	描述
pg_is_in_recovery()	bool	如果恢复仍在进行中，为真。
pg_last_wal_receive_lsn()	pg_lsn	获得最后一个收到并由流复制同步到磁盘的预写式日志位置。当流复制在进行中时，这将单调增加。如果恢复已经完成，这将保持静止在恢复过程中收到并同步到磁盘的最后一个 WAL 记录。如果流复制被禁用，或者还没有被启动，该函数返回 NULL。
pg_last_wal_replay_lsn()	pg_lsn	获得恢复过程中被重放的最后一个预写式日志位置。当流复制在进行中时，这将单调增加。如果恢复已经完成，这将保持静止在恢复过程中被应用的最后一个 WAL 记录。如果服务器被正常启动而没有恢复，该函数返回 NULL。
pg_last_xact_replay_timestamp()	timestamp with time zone	获得恢复过程中被重放的最后一个事务的时间戳。这是在主机上产生的事务的提交或中止 WAL 记录的时间。如果在恢复过程中没有事务被重放，这个函数返回 NULL。否则，如果恢复仍在进行这将单调增加。如果恢复已经完成，则这个值会保持静止在恢复过程中最后一个被应用的事务。如果服务器被正常启动而没有恢复，该函数返回 NULL。

表 9.81中展示的函数空值恢复的进程。这些函数只能在恢复过程中被执行。

表 9.81. 恢复控制函数

名称	返回类型	描述
pg_is_wal_replay_paused()	bool	如果恢复被暂停，为真。
pg_wal_replay_pause()	void	立即暂停恢复（默认仅限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）。
pg_wal_replay_resume()	void	如果恢复被暂停，重启之（默认仅限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）。

在恢复被暂停时，不会有进一步的数据库改变被应用。如果在热备模式，所有新的查询将看到数据库的同一个一致快照，并且在恢复被继续之前不会有更多查询冲突会产生。

如果流复制被禁用，暂停状态可以无限制地继续而不出问题。在流复制进行时，WAL 记录将继续被接收，最后将会填满可用的磁盘空间，取决于暂停的持续时间、WAL 的产生率和可用的磁盘空间。

9.26.5. 快照同步函数

PostgreSQL允许数据库会话同步它们的快照。一个快照决定对于正在使用该快照的事务哪些数据是可见的。当两个或者更多个会话需要看到数据库中的相同内容时，就需要同步快照。如果两个会话独立开始其事务，就总是有可能有某个第三事务在两个START TRANSACTION命令的执行之间提交，这样其中一个会话就可以看到该事务的效果而另一个则看不到。

为了解决这个问题，PostgreSQL允许一个事务导出它正在使用的快照。只要导出的事务仍然保持打开，其他事务可以导入它的快照，并且因此可以保证它们可以看到和第一个事务看到的完全一样的数据库视图。但是注意这些事务中的任何一个对数据库所作的更改对其他事务仍然保持不可见，和未提交事务所作的修改一样。因此这些事务是针对以前存在的数据同步，而对由它们自己所作的更改则采取正常的动作。

如表 9.82中所示，快照通过pg_export_snapshot函数导出，并且通过SET TRANSACTION命令导入。

表 9.82. 快照同步函数

名称	返回类型	描述
pg_export_snapshot()	text	保存当前快照并返回它的标识符

函数pg_export_snapshot保存当前的快照并且返回一个text串标识该快照。该字符串必须被传递（到数据库外）给希望导入快照的客户端。直到导出快照的事务的末尾，快照都可以被导入。如果需要，一个事务可以导出多于一个快照。注意这样做只在 READ COMMITTED事务中有用，因为在REPEATABLE READ和更高隔离级别中，事务在它们的生命期中都使用同一个快照。一旦一个事务已经导出了任何快照，它不能使用PREPARE TRANSACTION。

关于如何使用一个已导出快照的细节请见SET TRANSACTION.

9.26.6. 复制函数

表 9.83中展示的函数用于控制以及与复制特性交互。有关底层特性的信息请见第 26.2.5 节 第 26.2.6 节以及 第 50 章这些函数只限于超级 用户使用。

很多这些函数在复制协议中都有等价的命令，见 第 53.4 节

第 9.26.3 节 第 9.26.4 和 第 9.26.5 节中描述的函数也与复制相关。

表 9.83. 复制 SQL 函数

函数	返回类型	描述
pg_create_physical_replication_slot(slot_name name [, immediately_reserve boolean])	(slot_name name, lsn pg_lsn)	创建一个新的名为slot_name的物理复制槽。第二个参数是可选的，当它为true时，立即为这个物理槽指定要被保留的LSN。否则该LSN会被保留在来自一个流复制客户端的第一个连接上。来自一个物理槽的流改变只可能出现在使用流复制协议时 — 见第 53.4 节当可选的第三参数temporary被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。这个函数对应于复制协议命令CREATE_REPLICATION_SLOT...PHYSICAL。
pg_drop_replication_slot(slot_name name)	void	丢弃名为slot_name的物理或逻辑复制槽。 和复制协议命令DROP_REPLICATION_SLOT相同。对于逻辑槽，在连接到在其中创建该槽的同一个数据库时，必须调用这个函数。
pg_create_logical_replication_slot(slot_name name, plugin name)	(slot_name name, lsn pg_lsn)	使用输出插件plugin创建一个名为 slot_name的新逻辑（解码）复制槽。当可选的第三参数temporary被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。对这个函数的调用与复制协议命令CREATE_REPLICATION_SLOT...LOGICAL具有相同的效果。
pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(lsn pg_lsn, xid xid, data text)	返回槽slot_name中的改变，从上一次已经被消费的点开始返回。 如果upto_lsn和upto_nchanges为NULL，逻辑解码将一 直继续到 WAL 的末尾。如果upto_lsn为非 NULL，解码将只包括那些在指 定LSN 之前提交的事务。如果upto_nchanges为非 NULL，解码将在其产生的行数超过指定值后停止。不过要注意， 被返回的实际行数可能更大，因为对这个限制的检查只会在增加了解码每个新

函数	返回类型	描述
		的提交事务产生 的行之后进行。
pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(lsn text, xid xid, data text)	行为就像pg_logical_slot_get_changes()函数， 不过改变不会被消费， 即在未来的调用中还会返回这些改变。
pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(lsn pg_lsn, xid xid, data bytea)	行为就像pg_logical_slot_get_changes()函数， 不过改变会以bytea返回。
pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(lsn pg_lsn, xid xid, data bytea)	行为就像pg_logical_slot_get_changes()函数， 不过改变会以bytea返回并且这些改变不会被消费，即在未来的调用中还会返回这些改变。
pg_replication_slot_advance(slot_name name, upto_lsn pg_lsn)	(slot_name name, end_lsn pg_lsn)	名为slot_name的复制槽的当前确认位置的增长。该槽将不会被反向移动，且它将不会被移动到超过当前插入位置的地方。返回该槽的名称以及它被推进到的真实位置。
pg_replication_origin_create(node_name text)	oid	用给定的外部名称创建一个复制源，并且返回分配给它的内部 id。
pg_replication_origin_drop(node_name text)	void	删除一个之前创建的复制源，包括任何相关的重放进度。
pg_replication_origin_oid(node_name text)	oid	用名称查找复制源并且返回内部 id。如果没有找到则抛出错误。
pg_replication_origin_session_setup(node_name text)	void	把当前会话标记为正在从给定的源进行重放，允许重放进度被跟踪。使用pg_replication_origin_session_reset可以取消 标记。只有之前没有源被配置时才能使用。
pg_replication_origin_session_reset()	void	取消pg_replication_origin_session_setup()的效果。
pg_replication_origin_session_is_setup()	bool	当前会话中是否已经配置了一个复制源？
pg_replication_origin_session_progress(flush bool)	pg_lsn	返回当前会话中配置的复制源的重放位置。参数 flush决定对应的本地事务是否被确保 已经刷入磁盘。
pg_replication_origin_xact_setup(origin_lsn	void	标记当前事务为正在重放一个已经在给定的LSN 和时间戳提交的事务。只有当之前已

函数	返回类型	描述
pg_lsn, origin_timestamp timestamptz)		经用pg_replication_origin_session_setup()配置过 一个复制源时才能被调用。
pg_replication_origin_xact_reset()	void	取消pg_replication_origin_xact_setup()的效果。
pg_replication_origin_advance(node_name text, pos pg_lsn)	void	把给定节点的复制进度设置为给定的位置。这主要用于配置更改或者类似 操作之后设置初始位置或者新位置。注意这个函数的不当使用可能会导致 不一致的复制数据。
pg_replication_origin_progress(node_name text, flush bool)	pg_lsn	返回给定复制元的重放位置。参数 flush决定对应的本地事务是否被确保 已经刷入磁盘。
pg_logical_emit_message(transactional bool, prefix text, content text)	pg_lsn	发出文本形式的逻辑解码消息。这可以被用来通过 WAL向逻辑解码插件传递一般消息。参数transactional指定该消息是否应该是当前事务的一部分或者当逻辑解码读到该记录时该消息是否应该被立刻写入并且解码。prefix是逻辑解码插件用来识别它们感兴趣的消息的文本前缀。content是消息的文本。
pg_logical_emit_message(transactional bool, prefix text, content bytea)	pg_lsn	发出二进制逻辑解码消息。这可以被用来通过WAL向逻辑解码插件传递一般性消息。参数transactional指定该消息是否应该成为当前事务的一部分或者是否应该在逻辑解码过程读到该记录时立刻进行写入和解码。参数prefix是一个逻辑解码插件使用的文本前缀，逻辑解码插件用它来识别感兴趣的消息。参数content是消息的二进制内容。

9.26.7. 数据库对象管理函数

表 9.84中展示的函数计算数据库对象使用的磁盘空间。

表 9.84. 数据库对象尺寸函数

名称	返回类型	描述
pg_column_size(any)	int	存储一个特定值（可能压缩过）所需的字节数

名称	返回类型	描述
pg_database_size(oid)	bigint	指定 OID 的数据库使用的磁盘空间
pg_database_size(name)	bigint	指定名称的数据库使用的磁盘空间
pg_indexes_size(regclass)	bigint	附加到指定表的索引所占的总磁盘空间
pg_relation_size(relation regclass, fork text)	bigint	指定表或索引的指定分叉（'main'、'fsm'、'vm'或'init'）使用的磁盘空间
pg_relation_size(relation regclass)	bigint	pg_relation_size(..., 'main')的简写
pg_size_bytes(text)	bigint	把人类可读格式的带有单位的尺寸转换成字节数
pg_size_pretty(bigint)	text	将表示成一个 64位整数的字节尺寸转换为带尺寸单位的人类可读格式
pg_size_pretty(numeric)	text	将表示成一个数字值的字节尺寸转换为带尺寸单位的人类可读格式
pg_table_size(regclass)	bigint	被指定表使用的磁盘空间，排除索引（但包括 TOAST、空闲空间映射和可见性映射）
pg_tablespace_size(oid)	bigint	指定 OID 的表空间使用的磁盘空间
pg_tablespace_size(name)	bigint	指定名称的表空间使用的磁盘空间
pg_total_relation_size(regclass)	bigint	指定表所用的总磁盘空间，包括所有的索引和TOAST数据

pg_column_size显示用于存储任意独立数据值的空间。

pg_total_relation_size接受一个表或 TOAST 表的 OID 或名称，并返回该表所使用的总磁盘空间，包括所有相关的索引。这个函数等价于pg_table_size + pg_indexes_size。

pg_table_size接受一个表的 OID 或名称，并返回该表所需的磁盘空间，但是排除索引（TOAST 空间、空闲空间映射和可见性映射包含在内）

pg_indexes_size接受一个表的 OID 或名称，并返回附加到该表的所有索引所使用的全部磁盘空间。

pg_database_size以及pg_tablespace_size接受数据库或者表空间的OID或者名称，并且返回它们使用的磁盘空间。要使用pg_database_size，用户必须具有指定数据库上的CONNECT权限（默认情况下已经被授予）或者是pg_read_all_stats角色的一个成员。要使用pg_tablespace_size，用户必须具有指定表空间上的CREATE权限或者是pg_read_all_stats角色的一个成员，除非该表空间是当前数据库的默认表空间。

pg_relation_size接受一个表、索引或 TOAST 表的 OID 或者名称，并且返回那个关系的一个分叉所占的磁盘空间的字节尺寸（注意 对于大部分目的，使用更高层的函数pg_total_relation_size 或者pg_table_size会更方便，它们会合计所有分叉的尺寸）。如果只得到一个参数，它会返回该关系的主数据分叉的尺寸。提供第二个参数 可以指定要检查哪个分叉：

- 'main'返回该关系主数据分叉的尺寸。

- 'fsm'返回与该关系相关的空闲空间映射（见 第 68.3 节）的尺寸。
- 'vm'返回与该关系相关的可见性映射（见 第 68.4 节）的尺寸。
- 'init'返回与该关系相关的初始化分叉（如 果有）的尺寸。

pg_size_pretty可以用于把其它函数之一的结果格式化成一种人类易读的格式，可以根据情况使用字节、kB、MB、GB 或者 TB。

pg_size_bytes可以被用来从人类可读格式的字符串得到其中所表示的字节数。其输入可能带有的单位包括字节、kB、MB、GB 或者 TB，并且对输入进行解析时是区分大小写的。如果没有指定单位，会假定单位为字节。

> **注意**
>
> 函数pg_size_pretty和pg_size_bytes所使用的单位 kB、MB、GB 和 TB 是用 2 的幂而不是 10 的幂来定义，因此 1kB 是 1024 字节，1MB 是 1024^2 = 1048576 字节，以此类推。

上述操作表和索引的函数接受一个regclass参数，它是该表或索引在pg_class系统目录中的OID。你不必手工去查找该 OID，因为regclass数据类型的输入转换器会为你代劳。只写包围在单引号内的表名，这样它看起来像一个文字常量。为了与普通SQL名称的处理相兼容，该字符串将被转换为小写形式，除非其中在表名周围包含双引号。

如果一个 OID 不表示一个已有的对象并且被作为参数传递给了上述函数，将会返回 NULL。

表 9.85中展示的函数帮助标识数据库对象相关的磁盘文件。

表 9.85. 数据库对象定位函数

名称	返回类型	描述
pg_relation_filenode(relation regclass)	oid	指定关系的文件结点号
pg_relation_filepath(relation regclass)	text	指定关系的文件路径名
pg_filenode_relation(tablespace oid, filenode oid)	regclass	查找与给定的表空间和文件节点相关的关系

pg_relation_filenode接受一个表、索引、序列或 TOAST 表的 OID 或名称，返回当前分配给它的“filenode”号。文件结点是关系的文件名的基本组件（详见第 68.1 节）。对于大多数表结果和pg_class.relfilenode相同，但是对于某些系统目录relfilenode为零，并且必须使用此函数获取正确的值。 如果传递一个没有存储的关系（如视图），此函数将返回NULL。

pg_relation_filepath与pg_relation_filenode类似，但是它返回关系的整个文件路径名（相对于数据库集簇的数据目录PGDATA）。

pg_filenode_relation是pg_relation_filenode的反向函数。给定一个“tablespace” OID以及一个 “filenode”，它会返回相关关系的 OID。对于一个在数据库的默认表空间中的表，该表空间可以指定为 0。

表 9.86列出了用来管理排序规则的函数。

表 9.86. 排序规则管理函数

名称	返回类型	描述
pg_collation_actual_version(oid)	text	返回来自操作系统的排序规则的实际版本

名称	返回类型	描述
pg_import_system_collations(schema regnamespace)	integer	导入操作系统排序规则

pg_collation_actual_version返回当前安装在操作系统中的该排序规则对象的实际版本。如果这个版本与pg_collation.collversion中的值不同，则依赖于该排序规则的对象可能需要被重建。还可以参考ALTER COLLATION。

pg_import_system_collations基于在操作系统中找到的所有locale在系统目录pg_collation中加入排序规则。这是initdb会使用的函数，更多细节请参考第 23.2.2 节如果后来在操作系统上安装了额外的locale，可以再次运行这个函数加入新locale的排序规则。匹配pg_collation中现有项的locale将被跳过（但是这个函数不会移除以在操作系统中不再存在的locale为基础的排序规则对象）。schema参数通常是pg_catalog，但这不是一种要求，排序规则也可以被安装到其他的方案中。该函数返回其创建的新排序规则对象的数量。

9.26.8. 索引维护函数

表 9.87展示了可用于 索引维护任务的函数。这些函数不能在恢复期间执行。只有超级用户以及给定索引的拥有者才能是用这些函数。

表 9.87. 索引维护函数

名称	返回类型	描述
brin_summarize_new_values(index regclass)	integer	对还没有建立概要的页面范围建立概要
brin_summarize_range(index regclass, blockNumber bigint)	integer	如果还没有对覆盖给定块的页面范围建立概要，则对其建立概要
brin_desummarize_range(index regclass, blockNumber bigint)	integer	如果覆盖给定块的页面范围已经建立有概要，则去掉概要
gin_clean_pending_list(index regclass)	bigint	把 GIN 待处理列表项移动到主索引结构中

brin_summarize_new_values接收一个 BRIN 索引的 OID 或者名称作为参数并且检查该索引以找到基表中当前还没有被该索引汇总的页面范围。对任意一个这样的范围，它将通过扫描那些表页面创建一个新的摘要索引元组。它会返回被插入到该索引的新页面范围摘要的数量。brin_summarize_range做同样的事情，不过它只对覆盖给定块号的范围建立概要。

gin_clean_pending_list接受一个 GIN 索引的 OID 或者名字，并且通过把指定索引的待处理列表中的项批量移动到主 GIN 数据结构来清理该索引的待处理列表。它会返回从待处理列表中移除的页数。注意如果其参数是一个禁用fastupdate选项构建的 GIN 索引，那么不会做清理并且返回值为 0，因为该索引根本没有待处理列表。有关待处理列表和fastupdate选项的细节请见第 66.4.1 和第 66.5 节

9.26.9. 通用文件访问函数

表 9.88中展示的函数提供了对数据库服务器所在机器上的文件的本地访问。只能访问数据库集簇目录以及log_directory中的文件，除非用户被授予了角色pg_read_server_files。使用相对路径访问集簇目录里面的文件，以及匹配 log_directory配置设置的路径访问日志文件。

注意向用户授予pg_read_file()或者相关函数上的EXECUTE特权，函数会允许他们读取服务器上该数据库能读取的任何文件并且这些读取动作会绕过所有的数据库内特权检查。这意味

着，除了别的之外，具有这种访问的用户能够读取pg_authid表中包含着认证信息的内容，也能读取数据库中的任意文件。因此，授予对这些函数的访问应该要很仔细地考虑。

表 9.88. 通用文件访问函数

名称	返回类型	描述
pg_ls_dir(dirname text [, missing_ok boolean, include_dot_dirs boolean])	setof text	列出目录中的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
pg_ls_logdir()	setof record	列出日志目录中文件的名称、尺寸以及最后修改时间。访问被授予给pg_monitor角色的成员，并且可以被授予给其他非超级用户角色。
pg_ls_waldir()	setof record	列出WAL目录中文件的名称、尺寸以及最后修改时间。访问被授予给pg_monitor角色的成员，并且可以被授予给其他非超级用户角色。
pg_read_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])	text	返回一个文本文件的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
pg_read_binary_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])	bytea	返回一个文件的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
pg_stat_file(filename text[, missing_ok boolean])	record	返回关于一个文件的信息。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。

这些函数中的某些有一个可选的missing_ok参数， 它指定文件或者目录不存在时的行为。如果为true， 函数会返回 NULL （pg_ls_dir除外，它返回一个空 结果集）。如果为false，则发生一个错误。默认是 false。

pg_ls_dir返回指定目录中所有文件（以及目录和其他特殊文件） 的名称。include_dot_dirs指示结果集中是否包括 “.”和“..”。默认是排除它们（false），但是 当missing_ok为true时把它们包括在内是 有用的，因为可以把一个空目录与一个不存在的目录区分开。

pg_ls_logdir返回日志目录中每个文件的名称、尺寸以及最后的修改时间（mtime）。默认情况下，只有超级用户以及pg_monitor角色的成员能够使用这个函数。可以使用GRANT把访问授予给其他人。

pg_ls_waldir返回预写式日志（WAL）目录中每个文件的名称、尺寸以及最后的修改时间（mtime）。默认情况下，只有超级用户以及pg_monitor角色的成员能够使用这个函数。可以使用GRANT把访问授予给其他人。

pg_read_file返回一个文本文件的一部分，从给定的offset开始，返回最多length字节（如果先到达文件末尾则会稍短）。如果offset为负，它相对于文件的末尾。如果offset和length被忽略，整个文件都被返回。从文件中读的字节被使用服务器编码解释成一个字符串；如果它们在编码中不合法则抛出一个错误。

pg_read_binary_file与pg_read_file相似，除了前者的结果是一个bytea值；相应地，不会执行编码检查。通过与convert_from函数结合，这个函数可以用来读取一个指定编码的文件：

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_file返回一个记录，其中包含文件尺寸、最后访问时间戳、最后修改时间戳、最后文件状态改变时间戳（只支持 Unix 平台）、文件创建时间戳（只支持 Windows）和一个boolean指示它是否为目录。通常的用法包括：

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.10. 咨询锁函数

表 9.89中展示的函数管理咨询锁。有关正确使用这些函数的细节请参考第 13.3.5 节。

表 9.89. 咨询锁函数

名称	返回类型	描述
pg_advisory_lock(key bigint)	void	获得排他会话级别咨询锁
pg_advisory_lock(key1 int, key2 int)	void	获得排他会话级别咨询锁
pg_advisory_lock_shared(key bigint)	void	获得共享会话级别咨询锁
pg_advisory_lock_shared(key1 int, key2 int)	void	获得共享会话级别咨询锁
pg_advisory_unlock(key bigint)	boolean	释放一个排他会话级别咨询锁
pg_advisory_unlock(key1 int, key2 int)	boolean	释放一个排他会话级别咨询锁
pg_advisory_unlock_all()	void	释放当前会话持有的所有会话级别咨询锁
pg_advisory_unlock_shared(key bigint)	boolean	释放一个共享会话级别咨询锁
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	释放一个共享会话级别咨询锁
pg_advisory_xact_lock(key bigint)	void	获得排他事务级别咨询锁
pg_advisory_xact_lock(key1 int, key2 int)	void	获得排他事务级别咨询锁
pg_advisory_xact_lock_shared(key bigint)	void	获得共享事务级别咨询锁

名称	返回类型	描述
pg_advisory_xact_lock_shared(key1 int, key2 int)	void	获得共享事务级别咨询锁
pg_try_advisory_lock(key bigint)	boolean	如果可能，获得排他会话级别咨询锁
pg_try_advisory_lock(key1 int, key2 int)	boolean	如果可能，获得排他会话级别咨询锁
pg_try_advisory_lock_shared(key bigint)	boolean	如果可能，获得共享会话级别咨询锁
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	如果可能，获得共享会话级别咨询锁
pg_try_advisory_xact_lock(key bigint)	boolean	如果可能，获得排他事务级别咨询锁
pg_try_advisory_xact_lock(key1 int, key2 int)	boolean	如果可能，获得排他事务级别咨询锁
pg_try_advisory_xact_lock_shared(key bigint)	boolean	如果可能，获得共享事务级别咨询锁
pg_try_advisory_xact_lock_shared(key1 int, key2 int)	boolean	如果可能，获得共享事务级别咨询锁

pg_advisory_lock锁住一个应用定义的资源，可以使用一个单一64位键值或两个32位键值标识（注意这些两个键空间不重叠）。如果另一个会话已经在同一个资源标识符上持有了一个锁，这个函数将等待直到该资源变成可用。该锁是排他的。多个锁请求会入栈，因此如果同一个资源被锁住三次，则它必须被解锁三次来被释放给其他会话使用。

pg_advisory_lock_shared的工作和pg_advisory_lock相同，不过该锁可以与其他请求共享锁的会话共享。只有想要排他的锁请求会被排除。

pg_try_advisory_lock与pg_advisory_lock相似，不过该函数将不会等待锁变为可用。它要么立刻获得锁并返回true，要么不能立即获得锁并返回false。

pg_try_advisory_lock_shared的工作和pg_try_advisory_lock相同，不过它尝试获得一个共享锁而不是一个排他锁。

pg_advisory_unlock将会释放之前获得的排他会话级别咨询锁。如果锁被成功释放，它返回true。如果锁没有被持有，它将返回false并且额外由服务器报告一个 SQL 警告。

pg_advisory_unlock_shared的工作和pg_advisory_unlock相同，除了它释放一个共享的会话级别咨询锁。

pg_advisory_unlock_all将释放当前会话所持有的所有会话级别咨询锁（这个函数隐式地在会话末尾被调用，即使客户端已经不雅地断开）。

pg_advisory_xact_lock的工作和pg_advisory_lock相同，不过锁是在当前事务的末尾被自动释放的并且不能被显式释放。

pg_advisory_xact_lock_shared的工作和pg_advisory_lock_shared相同，除了锁是在当前事务的末尾自动被释放的并且不能被显式释放。

pg_try_advisory_xact_lock的工作和pg_try_advisory_lock相同，不过锁（若果获得）是在当前事务的末尾被自动释放的并且不能被显式释放。

pg_try_advisory_xact_lock_shared的工作和pg_try_advisory_lock_shared相同，不过锁（若果获得）是在当前事务的末尾被自动释放的并且不能被显式释放。

9.27. 触发器函数

当前PostgreSQL提供一个内建的触发器函数suppress_redundant_updates_trigger， 它将阻止任何不会实际更改行中数据的更新发生，这与正常的行为不管数据是否改变始终执行更新相反（这是正常的行为，使得更新运行速度更快，因为不需要检查，并在某些情况下也是有用的）。

理想的情况下，你通常应该避免运行实际上并没有改变记录中数据的更新。 冗余更新会花费大量不必要的时间，尤其是如果有大量索引要改变， 并将最终不得不清理被死亡行占用的空间。但是，在客户端代码中检测这种情况并不总是容易的，甚至不可能做到。 而写表达式来检测它们容易产生错误。作为替代，使用suppress_redundant_updates_trigger可以跳过不改变数据的更新。 但是，你需要小心使用它。触发器需要很短但不能忽略的时间来处理每条记录，所以如果大多数被一个更新影响的记录确实被更改，此触发器的使用将实际上使更新运行得更慢。

suppress_redundant_updates_trigger函数可以像这样被加到一个表：

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

在大部分情况下，你可能希望在最后为每行触发这个触发器。考虑到触发器是按照名字顺序被触发，你需要选择一个位于该表所有其他触发器之后的触发器名字。

有关创建触发器的更多信息请参考CREATE TRIGGER。

9.28. 事件触发器函数

PostgreSQL提供了这些助手函数来从 事件触发器检索信息。

更多有关事件触发器的信息请见第 40 章

9.28.1. 在命令结束处捕捉更改

当在一个ddl_command_end事件触发器的函数中调用时，pg_event_trigger_ddl_commands返回被每一个用户动作 执行的DDL命令的列表。如果在其他任何环境中 调用这个函数，会发生错误。 pg_event_trigger_ddl_commands为每一个被执行的基本 命令返回一行，某些只有一个单一 SQL 句子的命令可能会返回多于一行。 这个函数返回下面的列：

名称	类型	描述
classid	oid	对象所属的目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	对象的子-id（例如列的属性号）
command_tag	text	命令标签

名称	类型	描述
object_type	text	对象的类型
schema_name	text	该对象所属的模式的名称（如果有），如果没有则为NULL。 没有引号。
object_identity	text	对象标识的文本表现形式，用模式限定。如果必要，出现在 该标识中的每一个标识符都会被引用。
in_extension	bool	如果该命令是一个扩展脚本的一部分则为真
command	pg_ddl_command	以内部格式表达的该命令的一个完整表现形式。这不能被直接输出， 但是可以把它传递给其他函数来得到有关于该命令不同部分的信息。

9.28.2. 处理被 DDL 命令删除的对象

pg_event_trigger_dropped_objects返回其sql_drop事件中命令所删除的所有对象的列表。如果在任何其他环境中被调用，pg_event_trigger_dropped_objects将抛出一个错误。pg_event_trigger_dropped_objects返回下列列：

名称	类型	描述
classid	oid	对象所属的目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	对象的子ID（如列的属性号）
original	bool	如果这是删除中的一个根对象则为真
normal	bool	指示在依赖图中有一个普通依赖关系指向该对象的标志
is_temporary	bool	如果该对象是一个临时对象则为真
object_type	text	对象的类型
schema_name	text	对象所属模式的名称（如果存在）；否则为NULL。不应用引用。
object_name	text	如果模式和名称的组合能被用于对象的一个唯一标识符，则是对象的名称；否则是NULL。不应用引用，并且名称不是模式限定的。
object_identity	text	对象身份的文本表现，模式限定的。每一个以及所有身份中出现的标识符在必要时加引号。
address_names	text[]	一个数组，它可以和object_type及address_args 一起通过pg_get_object_address()函

名称	类型	描述
		数在一台包含有　同类相同名称对象的远程服务器上重建该对象地址。
address_args	text[]	上述address_names的补充。

pg_event_trigger_dropped_objects可以被这样用在一个事件触发器中：

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.28.3.　处理表重写事件

表　9.90中所示的函数提供刚刚被调用过table_rewrite 事件的表的信息。如果在任何其他环境中调用，会发生错误。

表　9.90.　表重写信息

名称	返回类型	描述
pg_event_trigger_table_rewrite_oid()	oid	要被重写的表的 OID。
pg_event_trigger_table_rewrite_reason()	int	解释重写原因的原因代码。这些代码的确切含义在单独的文档中。

可以在一个这样的事件触发器中使用 pg_event_trigger_table_rewrite_oid函数：

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
               pg_event_trigger_table_rewrite_oid()::regclass,
               pg_event_trigger_table_rewrite_reason();
END;
$$;
```

```
CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

第 10 章 类型转换

SQL语句可能（有意无意地）要求在同一表达式里混合不同的数据类型。 PostgreSQL在计算混合类型表达式方面有许多功能。

在大多数情况下，用户不需要明白类型转换机制的细节。但是，由PostgreSQL进行的隐式类型转换会对查询的结果产生影响。必要时这些结果可以被使用显式类型转换来调整。

本章介绍PostgreSQL类型转换的机制和习惯。 关于特定的类型和允许的函数及操作符的进一步信息，请参考第 8 章和第 9 章里的相关章节。

10.1. 概述

SQL是一种强类型语言。也就是说，每个数据项都有一个相关的数据类型，数据类型决定其行为和允许的用法。 PostgreSQL有一个可扩展的类型系统，该系统比其它SQL实现更具通用和灵活。因而，PostgreSQL中大多数类型转换行为是由通用规则来管理的，而不是ad hoc启发式规则。这种做法允许使用混合类型表达式，即便是其中包含用户定义的类型。

PostgreSQL扫描器/解析器只将词法元素分解成五个基本种类：整数、非整数数字、字符串、标识符、关键字。大多数非数字类型常量首先被分类为字符串。SQL语言定义允许将类型名指定为字符串， 这个机制被PostgreSQL用于保证解析器沿着正确的方向运行。例如，查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

有两个文字常量，类型分别为text和point。如果一个串文字没有指定类型，初始将被分配一个占位符类型unknown，该类型将在下文描述的后续阶段被解析。

在SQL解析器里，有四种基本的SQL结构要求独立的类型转换规则：

函数调用

PostgreSQL类型系统的大部分建立在一套丰富的函数上。 函数可以有一个或多个参数。由于PostgreSQL允许函数重载， 所以函数名自身并不唯一地标识将要被调用的函数，解析器必须根据提供的参数类型选择正确的函数。

操作符

PostgreSQL允许带有前缀和后缀一元（单目）操作符的表达式，也允许二元（两个参数）操作符。像函数一样，操作符也可以被重载，因此操作符的选择也有同样的问题。

值存储

SQL INSERT和UPDATE语句将表达式的结果放 入表中。语句中的表达式类型必须和目标列的类型一致（或者可以被转换为一致）。

UNION、CASE和相关结构

因为来自一个联合的SELECT语句中的所有查询结果必须在一个列集中显示，所以每个SELECT子句的结果类型必须能相互匹配并被转换成一个统一的集合。类似地，一个CASE结构的结果表达式必须被转换成一种公共的类型，这样CASE表达式作为整体才 有一种已知的输出类型。同样的要求也存在于ARRAY结构以及GREATEST和LEAST函数中。

系统目录存储有关哪些数据类型之间存在哪种转换（或造型）以及如何执行这些转换的相关信息。额外的造型可以由用户通过CREATE CAST命令增加（这个通常和定义一种新的数据类型一起完成。 内建的类型转换集已经经过了仔细的雕琢，最好不要去更改它们）。

解析器提供了一种额外的启发式规则，它允许在具有隐式造型的类型组中恰当造型行为的改进决定。 数据类型被分为几个基本的类型分类，包括boolean、numeric、string、bitstring、datetime、timespan、geometric、network和用户自定义（可参阅表 52.63中的列表；但需要注意的是 也可以创建自定义的类型分类）。在每个分类中，可以有一个或多个首选类型， 当存在类型选择时，这个是更好的选择。利用精心选择的首选类型和可用的隐式造型， 我们可以确保有歧义的表达式（那些有多个候选解析方案的表达式）可以用一种有用的方式来处理。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐式转换决不能有意外的或不可预见的输出。
- 如果一个查询不需要隐式类型转换，解析器或执行器不应该有额外的开销。也就是说，如果一个查询是结构良好的并且类型已经匹配，则查询不应该在解析器里耗费额外的时间执行，也不会在查询中引入不必要的隐式类型转换调用。
- 另外，如果一个查询通常要求为某个函数进行隐式类型转换，而用户定义了一个有正确参数类型的新函数， 解析器应该使用新函数并不再做隐式转换来使用旧函数。

10.2. 操作符

被一个操作符表达式引用的特定操作符由下列过程决定。注意这个过程会被所涉及的操作符的优先级间接地影响，因为这将决定哪些子表达式被用作哪个操作符的输入。详见第 4.1.6 节

操作符类型决定

1. 从系统目录pg_operator中选出要考虑的操作符。如果使用了一个不带模式限定的操作符 名（常见的情况），那么操作符被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的操作符（参见第 5.8.3 节。如果给出一个被限定的操作符名，那么只考虑指定模式中的操作符。

    - (可选的) 如果搜索路径找到了多个有相同参数类型的操作符，那么只考虑最早出现在路径中的那一个。 但是不同参数类型的操作符将被平等看待，而不管它们在路径中的位置如何。

2. 查找一个正好接受输入参数类型的操作符。如果找到一个（在一组被考虑的操作符中，可能只存在一个正好匹配的），则使用之。在通过限定名称（非典型）调用在一个允许不可信用户创建对象的方案中找到的任意操作符时，精确匹配的缺失会导致安全性危害[1] 。在这样的情况下，应该造型参数以便强制一次精确匹配。

    a. (可选的) 如果一个二元操作符调用中的一个参数是unknown类型，则在本次检查中假设它与另一个参数类型相同。 对于涉及两个unknown输入的调用或者带有一个unknown输入的一元操作符，在这一步将永远找不到一个匹配。

    b. (可选的) 如果一个二元操作符调用的其中一个参数是unknown类型 而另一个是一种域类型，下一次检查会看看是否有一个操作符正好在两边都 接受该域的基类型，如果有就使用它。

3. 寻找最优匹配。

[1] 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

a. 抛弃那些输入类型不匹配并且也不能被转换成匹配的候选操作符。 unknown文字被假定为可以为这个目的被转换为 任何东西。如果只剩下一个候选操作符，则使用之，否则继续下一 步。

b. 如果任何输入参数是一种域类型，对所有后续步骤都把它当做是该 域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们 的基类型。

c. 遍历所有候选操作符，保留那些在输入类型上的匹配最准确的。如果没有一个操作符能准确匹配，则保留所有候选。如果只剩下一个候选操作符，则使用之，否则继续下一步。

d. 遍历所有候选操作符，保留那些在最多个需要类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用之， 否则继续下一步。

e. 如果有任何输入参数是unknown类型，检查被剩余候选操作符在那些参数位置上接受的类型分类。 在每一个位置，如果任何候选接受该分类，则选择string分类（这种对字符串的偏爱是合适的， 因为未知类型的文本确实像字符串）。否则，如果所有剩下的候选操作符都接受相同的类型 分类，则选择该分类；否则抛出一个错误（因为在没有更多线索的条件下无法作出正确 的推断）。现在抛弃不接受选定的类型分类的候选操作符。然后，如果任意候选操作符接受那个分类中的首选类型， 则抛弃那些在该参数位置接受非首选类型的候选操作符。如果没有候选操作符能通过这些测试则保留全部候选者。如果只剩下一个候选者，则使用之；否则继续下一步。

f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选操作符可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。

下面是一些例子。

例 10.1. 阶乘操作符类型决定

在标准目录中只有一个被定义的阶乘操作符（后缀!），它接受一个类型为bigint的参数。在下面这个查询表达式中，扫描器会为该参数分配一个初始类型integer：

```
SELECT 40 ! AS "40 factorial";

                  40 factorial
-----------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

因此，解析器在操作数上做了一个类型转换，该查询等价于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

例 10.2. 字符串连接操作符类型决定

一个类字符串的语法被用来处理字符串类型和处理复杂的扩展类型。未指定类型的字符串与可能的候选操作符匹配。

一个未指定参数的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

在这种情况下，解析器查看是否有一个操作符的两个参数都使用text。既然有，那么它假设第二个参数应被解释为text类型。

下面是两个未指定类型的值的连接：

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

在这种情况下，没有对于使用哪种类型的初始提示，因为在查询中没有指定类型。 因此，解析器查找所有的候选操作符并找到候选者同时接受字符串分类和位串分类的输入。 因为字符串分类在可用时是首选的，该分类会被选中，并且接下来字符串的首选类型（text）会被用作解决未知类型文字的指定类型。

例 10.3. 绝对值与否定操作符类型决定

PostgreSQL操作符目录中有几个对于前缀操作符@的条目， 这些都现实了针对各种数字数据类型的绝对值操作。其中之一用于float8类型，它是在数字分类中的首选类型。 因此，PostgreSQL将在遇到一个unknown输入时使用它：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

在这里，系统在应用所选操作符之前已经隐式地解决了将未知类型文字作为float8类型。我们可以验证我们使用的是float8而不是别的类型：

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

另一方面，前缀符~（按位取反）只为整数数据类型定义，而没有为float8定义。因此，如果我们尝试一个与使用~类似的情况，我们会得到：

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

这是因为系统不能决定在几个可能的~符号中应该选择哪一个。我们可以用一个显式造型来帮助它：

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

例 10.4. 数组包含操作符类型决定

这里是另一个决定带有一个已知和一个未知输入的操作符的例子：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

PostgreSQL操作符目录有一些条目用于中缀操作符<@，但是仅有的两个可以在左手边接受一个整数数组的是数组包含（anyarray <@ anyarray）和范围包含（anyelement <@ anyrange）。因为这些多态伪类型（见第 8.21 节中没有一个被认为是首选的，解析器不能以此为基础来解决歧义。不过，步骤 3.f告诉它假定位置类型的文字和其他输入的类型相同，即整数数组。现在这两个操作符中只有一个可以匹配，因此数组包含被选择（如果选择范围包含，我们将得到一个错误，因为该字符串没有成为一个范围文字的正确格式）。

例 10.5. 域类型上的自定义操作符

用户有时会尝试声明只适用于一种域类型的操作符。这是可能的， 但是远非它看起来那么有用，因为操作符解析规则被设计为选择 适用于域的基类型的操作符。考虑这个例子：

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

这个查询将不会使用自定义操作符。解析器将首先看看是否有一个 mytext = mytext操作符（ 步骤 2.a），当然这里没有； 然后它将会考虑该域的基类型text，并且看看是否有一个text = text操作符（ 步骤 2.b），这里也没有；因 此它会把unknown-类型文字解析为text 并使用text = text操作符。 让自定义操作符能被使用的唯一方法是显式地转换改文字：

```
SELECT * FROM mytable WHERE val = text 'foo';
```

这样根据准确匹配规则会立即找到 mytext = text操作符。如果 到达最佳匹配规则，它们会积极地排斥域类型上的操作符。如果它 们没有，这样一个操作符将创建太多歧义操作符失败，因为转换规 则总是认为一个域可以和它的基类型相互转换，并且因此该域操作 符在所有与该基类型上的一个类似命名的操作符相同的情况中都被 认为可用。

10.3. 函数

被一个函数调用引用的特定函数使用下面的过程来决定。

函数类型决定

1. 从pg_proc系统目录中选择要被考虑的函数。 如果使用一个非模式限定的函数名称，那么函数被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的函数（参见第 5.8.3 节。如果给出一个被限定的函数名，那么只考虑指定模式中的函数。

   a. (可选的) 如果搜索路径发现多个参数类型相同的函数，那么只考虑最早在搜索路径中出现的那个。 不同参数类型的函数被平等对待，不受在搜索路径中位置的影响。

   b. (可选的) 如果使用一个VARIADIC数组参数声明一个函数，并且调用不使用关键字VARIADIC， 那么该函数就好像其数组参数被它的元素类型的一次或多次出现所替换，根据需要去匹配调用。 这样的扩展之后，函数可能会有和非可变函数相同的参数类型。在这种情况下，在搜索路径中出现比较早的函数将被使用，或者如果两个函数在相同的模式中时首选非可变的那一个。

   在通过限定名称调用在一个允许不可信用户创建对象的方案中找到的可变函数时，会导致安全性危害 [2]。 恶意用户可以拿到控制权并且执行任意SQL函数（就好像你在执行它们一样）。将涉及VARIADIC关键词的调用替换掉就可以绕过这种危害。涉及到VARIADIC "any"参数的调用通常没有等效的包含VARIADIC关键词的形式。为了安全地发出那些调用，函数的方案必须只允许可信用户创建对象。

   c. (可选的) 考虑使用有默认参数值的函数来匹配任何省略了零个或者多个可默认参数位置的调用。如果有超出一个的这种函数匹配一个调用，那么使用最早出现在搜索路径中的那个。如果同一个模式中在同一个非默认位置上有两个或者更多这样的函数（如果它们有 不同的默认参数设置，这是可能的），系统将不能确定去选择哪一个，并且如果不能找到该调用更好的匹配，将会导致一个“有歧义的函数调用” 错误。

   在通过限定名称[2]调用在一个允许不可信用户创建对象的方案中找到的任意函数时，会导致可用性危害。恶意用户可以用一个已有函数的名称创建一个函数，复制该函数的参数并且追加新的具有默认值的参数。这会妨碍对原始函数的新调用。为了防止这种危害，应将函数放在仅允许可信用户创建对象的方案中。

2. 检查一个函数正好接受输入参数类型。如果存在一个（在所考虑的一组函数中只能有一个准确匹配），则使用之。在通过限定名称[2]调用在一个允许不可信用户创建对象的方案中找到的函数时，精确匹配的缺失会导致安全性危害。在这样的情况下，应该造型参数以便强制一次精确匹配（在该步骤中，涉及unknown的情况将永远找不到一个匹配）。

3. 如果没有发现准确匹配，那么查看函数调用是否作为一个特定的类型转换请求出现。如果函数调用仅有一个参数并且函数名和一些数据类型的（内部）名称相同，那么该情况将会发生。 并且，该函数参数必须是一个未知类型的文字，或者是一个可以被二进制强制转换到命名数据类型的类型， 或者是一个可以通过应用其I/O函数被转换为命名数据类型的类型（也就是，转换是转到标准字符串类型或者从标准字符串类型转来）。当满足这些条件的时候，函数调用被当做CAST声明的一种形式来对待。 [3]

4. 查找最佳匹配。

   a. 如果候选函数的输入类型不匹配并且不能通过转换（使用一个隐式转换）达到匹配，则丢弃它。为了这个目的，unknown文字被假定可被转换成任何东西。如果仅有一个候选项，则使用之；否则继续下一步。

   b. 如果任何输入参数是一种域类型，在所有后续步骤中都把它当做 该域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们 的基类型。

[2] 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

[3] 这一步的原因是在没有一个实际的造型函数的情况下支持函数风格的造型声明。如果有一个造型函数，它被按惯例以其输出类型命名，并且不需要有特殊情况。更多信息请见CREATE CAST。

c. 遍历所有候选函数并保留那些最匹配输入类型的。如果没有准确匹配，则保留所有候选项。 如果仅有一个候选项，则使用之；否则继续下一步。

d. 遍历所有候选函数并保留那些在最多要求类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的候选项。如果没有接受首选类型的候选项，则保留所有候选项。如果仅有一个候选项，则使用之；否则继续下一步。

e. 如果任何输入参数是unknown，那么检查那些被剩余候选项在那些参数位置上接受的类型分类。在每一个位置上，如果任何候选项接受该分类则选择string分类（这个偏向于字符串是恰当的，因为一个未知类型文字看起来像字符）。 否则，如果所有剩余的候选项接受相同的类型分类，那么选择那个分类； 否则将失败，因为缺乏更多线索来推断出正确的选择。现在，丢弃不接受被选中类型分类的候选项。此外，如果任何候选项接受那个分类中的一个首选类型，则丢弃对该参数接受非首选类型的候选项。如果没有候选项能通过这些测试，则保留所有候选项。如果只剩下一个候选项，则使用之；否则继续下一步。

f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选函数可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。

注意，对于操作符和函数类型决定来说“最优匹配”规则是完全相同的。下面是一些例子。

例 10.6. 圆整函数参数类型决定

只有一个带有两个参数的圆整函数； 它采用第一个参数类型为numeric和第二个参数类型为integer。这样下面的查询自动将第一个类型为integer参数转换为numeric：

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

该查询实际上被解析器转换为：

```
SELECT round(CAST (4 AS numeric), 4);
```

因为包含小数点的数字常数初始会被分配类型numeric，下面的查询将不需要类型转换并因此可能会稍稍高效一些：

```
SELECT round(4.0, 4);
```

例 10.7. 可变函数决定

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

这个函数接受（但不要求）VARIADIC关键词。它能同时容忍integer以numeric参数：

```
SELECT public.variadic_example(0),
```

```
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

不过，如果可以，第一个和第二个调用将更喜欢更明确的函数：

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

如果给定默认的配置并且只有第一个函数存在，则第一个和第二个调用是不安全的。任何用户都可以通过创建第二个或者第三个函数来截断它们。通过精确匹配参数类型并且使用VARIADIC关键词，第三个调用是安全的。

例 10.8. 子串函数类型决定

有几个substr函数，其中一个用于text和integer类型。如果使用一个未指定类型的字符常量调用，那么系统选择接受一个首选分类string（也就是text类型）的参数的候选函数。

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

如果字符串被声明为类型varchar（如果它来自于一个表就会这样），那么解析器将尝试转换它为text：

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

解析器所作的转换：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **注意**
>
> 解析器从pg_cast目录中知道text和varchar是二进制可兼容的， 意思是其中一个可以被传递给接受另一种类型的函数而不需要做任何物理转换。因此，在这种情况下不会真正使用类型转换调用。

并且，如果该函数使用一个integer类型的参数调用，那么解析器将尝试将它转换为text：

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

由于integer类型没有到text的一个隐式造型，这将不会工作。但是一次显式造型则可以工作：

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. 值存储

将被插入到一个表的值会按照下列步骤被转换到目标列的数据类型。

值存储类型转换

1. 检查一个与目标的准确匹配。
2. 否则，尝试转换表达式为目标类型。如果在两种类型之间的一个 赋值造型已经被注册在pg_cast 目录（见CREATE CAST）中， 这是可能的。或者，如果该表达式是一个未知类型的文字， 则该文字串的内容将被提供给目标类型的输入转换例程。
3. 检查是否有一个用于目标类型的尺寸调整造型。尺寸调整造型是一个从该类型到其自身的造型。 如果在pg_cast目录中找到一个，那么把表达式存储到目标列中之前把它应用到表达式。这样一个造型的实现函数总是采用一个额外的integer类型的参数，它接收目标列的atttypmod值（通常是它被声明的长度，尽管对于不同数据类型atttypmod有不同的解释），并且它可能采用第三个boolean参数来说明造型是显式的还是隐式的。该造型函数负责应用任何长度相关的语义，例如尺寸检查或截断。

例 10.9.character存储类型转换

对于一个声明为character(20)的目标列，下面的语句展示了被存储的值如何被正确地调整尺寸：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
```

```
 abcdef                 |              20
(1 row)
```

实际发生的事情是两个未知文字被默认决定为text，允许||操作符被决定为text连接。 然后操作符的text结果被转换成bpchar（“空白填充字符”，character数据类型的内部名称）来匹配目标列类型（由于从text到bpchar的转换是二进制强制的，这个转换不会插入任何实际的函数调用）。最后，尺寸调整函数bpchar(bpchar, integer, boolean)被从系统目录中找到并应用到操作符的结果和存储的列长度上。这个类型相关的函数执行必要的长度检查并增加填充的空间。

10.5. UNION、CASE和相关结构

SQL UNION结构必须使可能不相似的类型匹配成为一个单一的结果集。该决定算法被独立地应用到一个联合查询的每个输出列。INTERSECT和EXCEPT采用和UNION相同的方法来决定不相似的类型。CASE、ARRAY、VALUES、GREATEST和LEAST结构使用相同的算法来使它们的组成表达式匹配并选择一种结果数据类型。

UNION、CASE和相关结构的类型决定

1. 如果所有的输入为相同类型，并且不是unknown，那么就决定是该类型。
2. 如果任何输入是一种域类型，在所有后续步骤中都把它当做 该域的基类型。 [4]
3. 如果所有的输入为unknown类型，则决定为text（字符串分类的首选类型）类型。否则，为了剩余规则，unknown输入会被忽略。
4. 如果非未知输入不全是相同的类型分类，则失败。
5. 如果有的话，选择第一个在其分类中作为首选类型的非未知输入类型。
6. 否则，选择最后的非未知输入类型，它允许所有在前面的非未知输入被隐式地转换为它（总有这样的一种类型，因为至少在列表中的第一个类型必须满足这个条件）。
7. 转换所有的输入为选定的类型。如果没有一个从给定输入到选定类型的转换将会失败。

下面是一些例子。

例 10.10. 联合中未指定类型的类型决定

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

这里，未知类型文字'b'将被决定为类型text。

例 10.11. 简单联合中的类型决定

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
```

[4] 多少有些类似于对待用于操作符和函数的域输入的方式，这种行为允许 一种域类型能通过一个UNION或相似的结构保留下来，只要用户小心地确保所有的输入都是（显式地或隐式地）准确类型。否 则会优先选择该域的基类型。

```
---------
       1
     1.2
(2 rows)
```

文字1.2是numeric类型，且integer值1可以被隐式地造型为numeric，因此使用numeric类型。

例 10.12. 可换位联合中的类型决定

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

这里，由于类型real被能被隐式地造型为integer，而integer可以被隐式地造型为real，联合结果类型被决定为real。

例 10.13. 嵌套合并中的类型决定

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

这个失败发生的原因是PostgreSQL把多个UNION当作是成对操作的嵌套，也就是说上面的输入等同于：

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

根据上面给定的规则，内层的UNION被确定为类型text。然后外层的UNION的输入是类型text和integer，这就导致了上面看到的错误。通过确保最左边的UNION至少有一个输入类型为想要的结果类型，就可以修正这个问题。

INTERSECT和EXCEPT操作也被当作成对操作。不过，这一节中描述的其他结构会在一个决定步骤中考虑所有的输入。

10.6. SELECT的输出列

前面的小节中给出的规则将会导致对SQL查询中的所有表达式分配非unknown数据类型，不过作为SELECT命令的简单输出列出现的未指定类型的文本除外。例如，在

```
SELECT 'Hello World';
```

中没有标识该字符串应该取何种类型。在这种情况下，PostgreSQL将会退而求其次将其类型决定为text。

当SELECT处于UNION（或者INTERSECT，或者EXCEPT）结构的一边或者出现在INSERT ... SELECT中时，这条规则就不适用了，因为在前面小节中给出的规则会优先。在第一种情况下未指定类型文本的类型将从UNION的另一边取得，而在第二种情况下未指定类型文本的类型将从目标列取得。

出于这样的目的，RETURNING列表采用和SELECT输出列表同样的方式对待。

> **注意**
>
> 在PostgreSQL　10之前，这条规则还不存在，SELECT输出列表中未指定类型的文本的类型会被留成unknown。这样做会导致各种不好的后果，因此新版本中做出了改变。

第 11 章 索引

索引是提高数据库性能的常用途径。比起没有索引，使用索引可以让数据库服务器更快找到并获取特定行。但是索引同时也会增加数据库系统的日常管理负担，因此我们应该聪明地使用索引。

11.1. 简介

假设我们有一个如下的表：

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

而应用发出很多以下形式的查询：

```
SELECT content FROM test1 WHERE id = constant;
```

在没有事前准备的情况下，系统不得不扫描整个test1表，一行一行地去找到所有匹配的项。如果test1中有很多行但是只有一小部分行（可能是0或者1）需要被该查询返回，这显然是一种低效的方式。但是如果系统被指示维护一个在id列上的索引，它就能使用一种更有效的方式来定位匹配行。例如，它可能仅仅需要遍历一棵搜索树的几层而已。

类似的方法也被用于大部分非小说书籍中：经常被读者查找的术语和概念被收集在一个字母序索引中放在书籍的末尾。感兴趣的读者可以相对快地扫描索引并跳到合适的页而不需要阅读整本书来寻找感兴趣的材料。正如作者的任务是准备好读者可能会查找的术语一样，数据库程序员也需要预见哪些索引会有用。

正如前面讨论的，下列命令可以用来在id列上创建一个索引：

```
CREATE INDEX test1_id_index ON test1 (id);
```

索引的名字test1_id_index可以自由选择，但我们最好选择一个能让我们想起该索引用途的名字。

为了移除一个索引，可以使用DROP INDEX命令。索引可以随时被创建或删除。

一旦一个索引被创建，就不再需要进一步的干预：系统会在表更新时更新索引，而且会在它觉得使用索引比顺序扫描表效率更高时使用索引。但我们可能需要定期地运行ANALYZE命令来更新统计信息以便查询规划器能做出正确的决定。通过第 14 章信息可以了解如何找出一个索引是否被使用以及规划器在何时以及为什么会选择不使用索引。

索引也会使带有搜索条件的UPDATE和DELETE命令受益。此外索引还可以在连接搜索中使用。因此，一个定义在连接条件列上的索引可以显著地提高连接查询的速度。

在一个大表上创建一个索引会耗费很长的时间。默认情况下，PostgreSQL允许在索引创建时并行地进行读（SELECT命令），但写（INSERT、UPDATE和DELETE）则会被阻塞直到索引创建完成。在生产环境中这通常是不可接受的。在创建索引时允许并行的写是可能的，但是有些警告需要注意，更多信息可以参考并发构建索引。

一个索引被创建后，系统必须保持它与表同步。这增加了数据操作的负担。因此哪些很少或从不在查询中使用的索引应该被移除。

11.2. 索引类型

PostgreSQL提供了多种索引类型： B-tree、Hash、GiST、SP-GiST 、GIN 和 BRIN。每一种索引类型使用了 一种不同的算法来适应不同类型的查询。默认情况下， CREATE INDEX命令创建适合于大部分情况的B-tree 索引。

B-tree可以在可排序数据上的处理等值和范围查询。特别地，PostgreSQL的查询规划器会在任何一种涉及到以下操作符的已索引列上考虑使用B-tree索引：

```
<
<=
=
>=
>
```

将这些操作符组合起来，例如BETWEEN和IN，也可以用B-tree索引搜索实现。同样，在索引列上的IS NULL或IS NOT NULL条件也可以在B-tree索引中使用。

优化器也会将B-tree索引用于涉及到模式匹配操作符LIKE和~ 的查询，前提是如果模式是一个常量且被固定在字符串的开头—例如：col LIKE 'foo%'或者col ~ '^foo', 但在col LIKE '%bar'上则不会。但是，如果我们的数据库没有使用C区域设置，我们需要创建一个具有特殊操作符类的索引来支持模式匹配查询，参见下面的第 11.10 节同样可以将B-tree索引用于ILIKE和~*，但仅当模式以非字母字符开始，即不受大小写转换影响的字符。

B-tree索引也可以用于检索排序数据。这并不会总是比简单扫描和排序更快，但是总是有用的。

Hash索引只能处理简单等值比较。不论何时当一个索引列涉及到一个使用了=操作符的比较时，查询规划器将考虑使用一个Hash索引。下面的命令将创建一个Hash索引：

```
CREATE INDEX name ON table USING HASH (column);
```

GiST索引并不是一种单独的索引，而是可以用于实现很多不同索引策略的基础设施。相应地，可以使用一个GiST索引的特定操作符根据索引策略（操作符类）而变化。作为一个例子，PostgreSQL的标准捐献包中包括了用于多种二维几何数据类型的GiST操作符类，它用来支持使用下列操作符的索引化查询：

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

（这些操作符的含义见第 9.11 节表 64.1中给出了标准发布中所包括的 GiST 操作符类。contrib集合中还包括了很多其他GiST操作符类，可见第 64 章

GiST索引也有能力优化“最近邻”搜索，例如：

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

它将找到离给定目标点最近的10个位置。能够支持这种查询的能力同样取决于被使用的特定操作符类。 在表 64.中，“Ordering Operators”列中列出了可以在这种方法中使用的操作符。

和GiST相似，SP-GiST索引为支持多种搜索提供了一种基础结构。SP-GiST 允许实现众多不同的非平衡的基于磁盘的数据结构，例如四叉树、k-d树和radix树。作为一个例 子，PostgreSQL的标准捐献包中包含了一个用于二维点的SP-GiST操作符类，它用于支持使用下列操作符的索引化查询：

<<
>>
~=
<@
<^
>^

（其含义见第 9.11 节表 65.中给出了标准发布中所包括的 SP-GiST 操作符类。更多信息参见第 65 章

GIN 索引是“倒排索引”，它适合于包含多个组成值的数据值，例如数组。倒排索引中为每一个组成值都包含一个单独的项，它可以高效地处理测试指定组成值是否存在的查询。

与 GiST 和 SP-GiST相似， GIN 可以支持多种不同的用户定义的索引策略，并且可以与一个 GIN 索引配合使用的特定操作符取决于索引策略。作为一个例子，PostgreSQL的标准贡献包中包含了用于数组的GIN操作符类，它用于支持使用下列操作符的索引化查询：

<@
@>
=
&&

（这些操作符的含义见第 9.18 节表 66.中给出了标准发布中所包括的 GIN 操作符类。在contrib集合中还有更多其他GIN操作符类，更多信息参见第 66 章

BRIN 索引（块范围索引的缩写）存储有关存放在一个表的连续物理块范围上的值摘要信息。与 GiST、SP-GiST 和 GIN 相似，BRIN 可以支持很多种不同的索引策略，并且可以与一个 BRIN 索引配合使用的特定操作符取决于索引策略。对于具有线性排序顺序的数据类型，被索引的数据对应于每个块范围的列中值的最小值和最大值，使用这些操作符来支持用到索引的查询：

<
<=
=
>=
>

包括在标准发布中的 BRIN 操作符类的文档在表 67.中。更多信息请见第 67 章

11.3. 多列索引

一个索引可以定义在表的多个列上。例如，我们有这样一个表：

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

（即将我们的/dev目录保存在数据库中）而且我们经常会做如下形式的查询：

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

那么我们可以在major和minor上定义一个索引：

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

目前，只有 B-tree、GiST、GIN 和 BRIN 索引类型支持多列索引，最多可以指定32个列（该限制可以在源代码文件pg_config_manual.h中修改，但是修改后需要重新编译PostgreSQL）。

一个B-tree索引可以用于条件中涉及到任意索引列子集的查询，但是当先导列（即最左边的那些列）上有约束条件时索引最为有效。确切的规则是：在先导列上的等值约束，加上第一个无等值约束的列上的不等值约束，将被用于限制索引被扫描的部分。在这些列右边的列上的约束将在索引中被检查，这样它们适当节约了对表的访问，但它们并未减小索引被扫描的部分。例如，在(a, b, c)上有一个索引并且给定一个查询条件WHERE a = 5 AND b >= 42 AND c < 77，对索引的扫描将从第一个具有a = 5和b = 42的项开始向上进行，直到最后一个具有a = 5的项。在扫描过程中，具有c >= 77的索引项将被跳过，但是它们还是会被扫描到。这个索引在原则上可以被用于在b和/或c上有约束而在a上没有约束的查询，但是整个索引都不得不被扫描，因此在大部分情况下规划器宁可使用一个顺序的表扫描来替代索引。

一个多列GiST索引可以用于条件中涉及到任意索引列子集的查询。在其余列上的条件将限制由索引返回的项，但是第一列上的条件是决定索引上扫描量的最重要因素。当第一列中具有很少的可区分值时，一个GiST索引将会相对比较低效，即便在其他列上有很多可区分值。

一个GIN索引可以用于条件中涉及到任意索引列子集的查询。与B-tree和GiST不同，GIN的搜索效率与查询条件中使用哪些索引列无关。

多列 BRIN 索引可以被用于涉及该索引被索引列的任意子集的查询条件。和 GIN 相似且不同于 B-树 或者 GiST，索引搜索效率与查询条件使用哪个索引列无关。在单个表上使用多个 BRIN 索引来取代一个多列 BRIN 索引的唯一原因是为了使用不同的pages_per_range存储参数。

当然，要使索引起作用，查询条件中的列必须要使用适合于索引类型的操作符，使用其他操作符的子句将不会被考虑使用索引。

多列索引应该较少地使用。在绝大多数情况下，单列索引就足够了且能节约时间和空间。具有超过三个列的索引不太有用，除非该表的使用是极端程式化的。第 11.5 节以及第 11.9 节有对不同索引配置优点的讨论。

11.4. 索引和ORDER BY

除了简单地查找查询要返回的行外，一个索引可能还需要将它们以指定的顺序传递。这使得查询中的ORDER BY不需要独立的排序步骤。在PostgreSQL当前支持的索引类型中，只有B-tree可以产生排序后的输出，其他索引类型会把行以一种没有指定的且与实现相关的顺序返回。

规划器会考虑以两种方式来满足一个ORDER BY说明：扫描一个符合说明的可用索引，或者先以物理顺序扫描表然后再显式排序。对于一个需要扫描表的大部分的查询，一个显式的排序很可能比使用一个索引更快，因为其顺序访问模式使得它所需要的磁盘I/O更少。只有在少数行需要被取出时，索引才会更有用。一种重要的特殊情况是ORDER BY与LIMIT n联合使用：一个显式的排序将会处理所有的数据来确定最前面的n行，但如果有一个符合ORDER BY的索引，前n行将会被直接获取且根本不需要扫描剩下的数据。

默认情况下，B-tree索引将它的项以升序方式存储，并将空值放在最后。这意味着对列x上索引的一次前向扫描将产生满足ORDER BY x（或者更长的形式：ORDER BY x ASC NULLS LAST）的结果。索引也可以被后向扫描，产生满足ORDER BY x DESC（ORDER BY x DESC NULLS FIRST， NULLS FIRST是ORDER BY DESC的默认情况）。

我们可以在创建B-tree索引时通过ASC、DESC、NULLS FIRST和NULLS LAST选项来改变索引的排序，例如：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

一个以升序存储且将空值前置的索引可以根据扫描方向来支持ORDER BY x ASC NULLS FIRST或ORDER BY x DESC NULLS LAST。

读者可能会疑惑为什么要麻烦地提供所有四个选项，因为两个选项连同可能的后向扫描可以覆盖所有ORDER BY的变体。在单列索引中这些选项确实有冗余，但是在多列索引中它们却很有用。考虑(x, y)上的一个两列索引：它可以通过前向扫描满足ORDER BY x, y，或者通过后向扫描满足ORDER BY x DESC, y DESC。但是应用可能需要频繁地使用ORDER BY x ASC, y DESC。这样就没有办法从通常的索引中得到这种顺序，但是如果将索引定义为(x ASC, y DESC)或者(x DESC, y ASC)就可以产生这种排序。

显然，具有非默认排序的索引是相当专门的特性，但是有时它们会为特定查询提供巨大的速度提升。是否值得维护这样一个索引取决于我们会多频繁地使用需要特殊排序的查询。

11.5. 组合多个索引

只有查询子句中在索引列上使用了索引操作符类中的操作符并且通过AND连接时才能使用单一索引。例如，给定一个(a, b) 上的索引，查询条件WHERE a = 5 AND b = 6可以使用该索引，而查询WHERE a = 5 OR b = 6不能直接使用该索引。

幸运的是，PostgreSQL具有组合多个索引（包括多次使用同一个索引）的能力来处理那些不能用单个索引扫描实现的情况。系统能在多个索引扫描之间安排AND和OR条件。例如，WHERE x = 42 OR x = 47 OR x = 53 OR x = 99这样一个查询可以被分解成为四个独立的在x上索引扫描，每一个扫描使用其中一个条件。这些查询的结果将被“或”起来形成最后的结果。另一个例子是如果我们在x和y上都有独立的索引，WHERE x = 5 AND y = 6这样的查询的一种可能的实现方式就是分别使用两个索引配合相应的条件，然后将结果“与”起来得到最后的结果行。

为了组合多个索引，系统扫描每一个所需的索引并在内存中准备一个位图用于指示表中符合索引条件的行的位置。然后这些位图会被根据查询的需要“与”和“或”起来。最后，实际的表行将被访问并返回。表行将被以物理顺序访问，因为位图就是以这种顺序布局的。这意味着原始索引中的任何排序都会被丢失，并且如果存在一个ORDER BY子句就需要一个单独的排序步骤。由于这个原因以及每一个附加的索引都需要额外的时间，即使有额外的索引可用，规划器有时也会选择使用单一索引扫描。

在所有的应用（除了最简单的应用）中，可能会有多种有用的索引组合，数据库开发人员必须做出权衡以决定提供哪些索引。有时候多列索引最好，但是有时更好的选择是创建单独的索引并依赖于索引组合特性。例如，如果我们的查询中有时只涉及到列x，有时候只涉及到列y，还有时候会同时涉及到两列，我们可以选择在x和y上创建两个独立索引然后依赖索引组合来处理同时涉及到两列的查询。我们当然也可以创建一个(x, y)上的多列索引。当查询同时涉及到两列时，该索引会比组合索引效率更高，但是正如第 11.3 节讨论的，它在只涉及到y的查询中几乎完全无用，因此它不能是唯一的一个索引。一个多列索引和一个y上的独立索引的组合将会工作得很好。多列索引可以用于那些只涉及到x的查询，尽管它比x上的独立索引更大且更慢。最后一种选择是创建所有三个索引，但是这种选择最适合表经常被执行所有三种查询但是很少被更新的情况。如果其中一种查询要明显少于其他类型的查询，我们可能需要只为常见类型的查询创建两个索引。

11.6. 唯一索引

索引也可以被用来强制列值的唯一性，或者是多个列组合值的唯一性。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

当前，只有B-tree能够被声明为唯一。

当一个索引被声明为唯一时，索引中不允许多个表行具有相同的索引值。空值被视为不相同。一个多列唯一索引将会拒绝在所有索引列上具有相同组合值的表行。

PostgreSQL会自动为定义了一个唯一约束或主键的表创建一个唯一索引。该索引包含组成主键或唯一约束的所有列（可能是一个多列索引），它也是用于强制这些约束的机制。

> **注意**
>
> 不需要手工在唯一列上创建索引，如果那样做也只是重复了自动创建的索引而已。

11.7. 表达式索引

一个索引列并不一定是底层表的一个列，也可以是从表的一列或多列计算而来的一个函数或者标量表达式。这种特性对于根据计算结果快速获取表中内容是有用的。

例如，一种进行大小写不敏感比较的常用方法是使用lower函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

这种查询可以利用一个建立在lower(col1)函数结果之上的索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

如果我们将该索引声明为UNIQUE，它将阻止创建在col1值上只有大小写不同的行。

另外一个例子，如果我们经常进行如下的查询：

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

那么值得创建一个这样的索引：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

正如第二个例子所示，CREATE INDEX命令的语法通常要求在被索引的表达式周围书写圆括号。而如第一个例子所示，当表达式只是一个函数调用时可以省略掉圆括号。

索引表达式的维护代价较为昂贵，因为在每一个行被插入或更新时都得为它重新计算相应的表达式。然而，索引表达式在进行索引搜索时却不需要重新计算，因为它们的结果已经被存储在索引中了。在上面两个例子中，系统将会发现查询的条件是WHERE indexedcolumn = 'constant'，因此查询的速度将等同于其他简单索引查询。因此，表达式索引对于检索速度远比插入和更新速度重要的情况非常有用。

11.8. 部分索引

一个部分索引是建立在表的一个子集上，而该子集则由一个条件表达式（被称为部分索引的谓词）定义。而索引中只包含那些符合该谓词的表行的项。部分索引是一种专门的特性，但在很多种情况下它们也很有用。

使用部分索引的一个主要原因是避免索引公值。由于搜索一个公值的查询（一个在所有表行中占比超过一定百分比的值）不会使用索引，所以完全没有理由将这些行保留在索引中。这可以减小索引的尺寸，同时也将加速使用索引的查询。它也将加速很多表更新操作，因为这种索引并不需要在所有情况下都被更新。例 11.1展示了一种可能的应用：

例 11.1. 建立一个部分索引来排除公值

假设我们要在一个数据库中保存网页服务器访问日志。大部分访问都来自于我们组织内的IP地址，但是有些来自于其他地方（如使用拨号连接的员工）。如果我们主要通过IP搜索来自于外部的访问，我们就没有必要索引对应于我们组织内网的IP范围。

假设有这样一个表：

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

用以下命令可以创建适用于我们的部分索引：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

一个使用该索引的典型查询是：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

一个不能使用该索引的查询：

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

可以看到部分索引查询要求公值能被预知，因此部分索引最适合于数据分布不会改变的情况。当然索引也可以偶尔被重建来适应新的数据分布，但是这会增加维护负担。

例 11.2展示了部分索引的另一个可能的用途：从索引中排除那些查询不感兴趣的值。这导致了上述相同的好处，但它防止了通过索引来访问“不感兴趣的”值，即便在这种情况下一个索引扫描是有益的。显然，为这种场景建立部分索引需要很多考虑和实验。

例 11.2. 建立一个部分索引来排除不感兴趣的值

如果我们有一个表包含已上账和未上账的订单，其中未上账的订单在整个表中占据一小部分且它们是最经常被访问的行。我们可以通过只在未上账的行上创建一个索引来提高性能。创建索引的命令如下：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

使用该索引的一个可能查询是：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

然而，索引也可以用于完全不涉及order_nr的查询，例如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这并不如在amount列上部分索引有效，因为系统必须扫描整个索引。然而，如果有相对较少的未上账订单，使用这个部分索引来查找未上账订单将会更好。

注意这个查询将不会使用该索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

订单3501可能在已上账订单或未上账订单中。

例 11.2也显示索引列和谓词中使用的列并不需要匹配。PostgreSQL支持使用任意谓词的部分索引，只要其中涉及的只有被索引表的列。然而，记住谓词必须匹配在将要受益于索引的查询中使用的条件。更准确地，只有当系统能识别查询的WHERE条件从数学上索引的谓词时，一个部分索引才能被用于一个查询。PostgreSQL并不能给出一个精致的定理证明器来识别写成不同形式在数学上等价的表达式（一方面创建这种证明器极端困难，另一方面即便能创建出来对于实用也过慢）。系统可以识别简单的不等蕴含，例如“x < 1”蕴含“x < 2”；否则谓词条件必须准确匹配查询的WHERE条件中的部分，或者索引将不会被识别为可用。匹配发生在查询规划期间而不是运行期间。因此，参数化查询子句无法配合一个部分索引工作。例如，对于参数的所有可能值来说，一个具有参数“x < ?”的预备查询绝不会蕴含“x < 2”。

部分索引的第三种可能的用途并不要求索引被用于查询。其思想是在一个表的子集上创建一个唯一索引，如例 11.3所示。这对那些满足索引谓词的行强制了唯一性，而对那些不满足的行则没有影响。

例 11.3. 建立一个部分唯一索引

假设我们有一个描述测试结果的表。我们希望保证其中对于一个给定的主题和目标组合只有一个“成功”项，但其中可能会有任意多个“不成功”项。实现它的方式是：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

当有少数成功测试和很多不成功测试时这是一种特别有效的方法。

最后，一个部分索引也可以被用来重载系统的查询规划选择。同样，具有特殊分布的数据集可能导致系统在它并不需要索引的时候选择使用索引。在此种情况下可以被建立，这样它将不会被那些无关的查询所用。通常，PostgreSQL会对索引使用做出合理的选择（例如，它会在检索公值时避开索引，这样前面的例子只能节约索引尺寸，它并非是避免索引使用所必需的），非常不正确的规划选择则需要作为故障报告。

记住建立一个部分索引意味着我们知道的至少和查询规划器所知的一样多，尤其是我们知道什么时候一个索引会是有益的。构建这些知识需要经验和对于PostgreSQL中索引工作方式的理解。在大部分情况下，一个部分索引相对于一个普通索引的优势很小。

关于部分索引的更多信息可以在[ston89b]、[olson93]和[seshadri95]中找到。

11.9. 只用索引的扫描和覆盖索引

PostgreSQL中的所有索引是二级索引,这意味着每个索引都是与表的主数据区（在PostgreSQL术语称为表的堆中）分开存储。这意味着在普通索引扫描中，每行检索都需要从索引和堆中取数据。 此外，虽然匹配给定的可索引WHERE条件的索引条目通常在一起靠近存储，但它们引用的表行可能在堆中的任何地方。 因此索引扫描的堆访问部分涉及到对堆的大量随机访问，这可能很慢，特别是在传统旋转媒介上。如 第 11.5 节中所述，位图扫描尝试通过按排序的顺序进行堆访问来减少成本，但这远远不够）。

为了解决这种性能问题，PostgreSQL支持只用索引的扫描，这类扫描可以仅用一个索引来回答查询而不产生任何堆访问。其基本思想是直接从每一个索引项中直接返回值，而不是去参考相关的堆项。在使用这种方法时有两个根本的限制：

1. 索引类型必须支持只用索引的扫描。B-树索引总是支持只用索引的扫描。GiST 和 SP-GiST 索引只对某些操作符类支持只用索引的扫描。其他索引类型不支持这种扫描。底层的要求是索引必须在物理上存储或者可以重构出每一个索引项对应的原始数据值。GIN 索引是一个不支持只用索引的扫描的反例，因为它的每一个索引项通常只包含原始数据值的一部分。

2. 查询必须只引用存储在该索引中的列。例如，给定的索引建立在表的列x和y上，而该表还有一个列z，这些查询可以使用只用索引的扫描：

   ```
   SELECT x, y FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND y < 42;
   ```

   但是这些查询不能使用只用索引的查询：

   ```
   SELECT x, z FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND z < 42;
   ```

   （如下面所讨论的，表达式索引和部分索引会让这条规则更加复杂）。

如果符合这两个根本要求，那么该查询所要求的所有数据值都可以从索引得到，因此才可能使用只用索引的扫描。但是对PostgreSQL中的任何表扫描还有一个额外的要求：必须验证每一个检索到的行对该查询的 MVCC 快照是“可见的”，如第 13 章讨论的那样。可见性信息并不存储在索引项中，只存储在堆项中。因此，乍一看似乎每一次行检索无论如何都会要求一次堆访问。如果表行最近被修改过，确实是这样。但是，对于很少更改的数据有一种方法可以解决这个问题。PostgreSQL为表堆中的每一个页面跟踪是否其中所有的行的年龄都足够大，以至于对所有当前以及未来的事务都可见。这个信息存储在该表的可见性映射的一个位中。在找到一个候选索引项后，只用索引的扫描会检查对应堆页面的可见性映射位。如果该位被设置，那么这一行就是可见的并且该数据库可以直接被返回。如果该位没有被设置，那么就必须访问堆项以确定这一行是否可见，这种情况下相对于标准索引扫描就没有性能优势。即便是在成功的情况下，这种方法也是把对堆的访问换成了对可见性映射的访问。不过由于可见性映射比它所描述的堆要小四个数量级，所以访问可见性映射所需的物理 I/O 要少很多。在大部分情况下，可见性映射总是会被保留在内存中的缓冲中。

总之，虽然当两个根本要求满足时可以使用只用索引的扫描，但是只有该表的堆页面中有很大一部分的“所有都可见”映射位被设置时这种索引才有优势。不过，有很大一部分行不被更改的表是很常见的，这也让这一类扫描在实际中非常有用。

为了有效利用仅索引扫描功能，您可以选择创建一个覆盖索引，它是一个特别设计的索引，包含经常运行的特殊类型查询所需要的列。由于查询通常需要检索的列不仅仅是他们搜索的列，PostgreSQL允许您创建索引,这个索引中有些列只是“负荷”而不是搜索键的一部分。这可以通过添加INCLUDE来完成子句来列出了额外的列。 例如，如果您通常可以运行这样的查询：

```
SELECT y FROM tab WHERE x = 'key';
```

加快此类查询的传统方法是仅在x上的索引。但是，一个索引定义为

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

可以将这些查询作为仅索引扫描处理，因为y可以从索引中获取而不需要访问堆。

因为列y不是搜索键的一部分，它不必是索引可以处理的数据类型；它只存储在索引中，不由索引机解释。另外，如果索引是唯一的索引，则

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

唯一性条件仅适用于x 列，而不是x和y的组合。（如果使用和在索引中设置的类似语法，一个INCLUDE子句可以写在UNIQUE和PRIMARY KEY约束中。）

保守地将非键负载列添加到索引是明智的，尤其是宽列。 如果索引元组超过索引类型允许的最大大小，数据插入将失败。在任何情况下，非键列都将复制索引表中的数据并放大了索引的大小，从而有可能减慢搜索速度。请记住，除非一个表足够慢以至于仅索引扫描可能不必访问堆，否则没有什么理由在一个索引中包含负载列。无论如何，如果必须访问堆元组，从堆里获取列的值并不会带来更高的开销。其他限制是表达式不被作为包含的来支持。只有B树索引当前支持包含的列。

在 PostgreSQL有INCLUDE特性之前，人们有时会通过写负载列作为普通索引列来制作覆盖索引。它这样写：

```
CREATE INDEX tab_x_y ON tab(x, y);
```

即使他们无意将y用作WHER子句的一部分，只要额外的列是尾列就可以很好的工作。 让它们成为前导字段是不明智的，原因在 第 11.3 节有说明。但是，此方法不支持您希望索引在键列上实施唯一性。 另外，明确标记不可搜索列为INCLUDE 列可让索引稍小一些，因为这样的列不需要存储在B树的上层。

原则上，只用索引的扫描可以被用于表达式索引。例如，给定一个f(x)上的索引（x是一个表列），可以把

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

作为只用索引的扫描执行，如果f()是一个计算代价昂贵的函数，这会非常有吸引力。不过，PostgreSQL的规划器当前面对这类情况时并不是很聪明。只有在索引中有查询所需要的所有列时，规划器才会考虑用只用索引的扫描来执行一个查询。在这个例子中，除了在f(x)环境中之外，查询的其他部分不需要x，但是规划器并不能意识到这一点，因此它会得出不能使用只用索引的扫描的结论。如果只用索引的扫描足够有价值，有一种解决方法是把该索引定义在(f(x), x)上，其中第二个列实际上并不会被使用，它只是用来说服规划器可以使用只用索引的扫描而已。如果目标是避免重复计算f(x)，一个额外的警示是规划器不一定会把不在可索引WHERE子句中对f(x)的使用匹配到索引列。通常在上述那种简单查询中一切正常，但是涉及到连接的查询中就不行了。这些不足将在未来的PostgreSQL版本中修正。

部分索引也和只用索引的扫描之间有着有趣的关系。考虑例 11.3中所展示的部分索引：

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

原则上，我们可以在这个索引上使用只用索引的扫描来满足查询

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

但是有一个问题：WHERE子句引用的是不能作为索引结果列的success。尽管如此，还是可以使用只用索引的扫描，因为在运行时计划不需要重新检查WHERE子句的那个部分：在该索引中找到的所有项必定具有success = true，因此在计划中检查这个部分的需要并不明显。PostgreSQL 9.6 和以后的版本将会识别这种情况，并且允许生成只用索引的扫描，但是旧版本无法这样做。

11.10. 操作符类和操作符族

一个索引定义可以为索引中的每一列都指定一个操作符类。

```
CREATE INDEX name ON table (column opclass [sort options] [, ...]);
```

操作符类标识该列上索引要使用的操作符。例如，一个int4类型上的B树索引会使用int4_ops类，这个操作符类包括用于int4类型值的比较函数。实际上列的数据类型的默认操作符类通常就足够了。存在多个操作符类的原因是，对于某些数据类型可能会有多于一种的有意义的索引行为。例如，我们可能想要对一种复数数据类型按照绝对值排序或者按照实数部分排序。我们可以通过为该数据类型定义两个操作符类来实现，并且在创建一个索引时选择合适的类。操作符类会决定基本的排序顺序（可以通过增加排序选项COLLATE、 ASC/DESC和/或 NULLS FIRST/NULLS LAST来修改）。

除了默认的操作符类，还有一些内建的操作符类：

- 操作符类text_pattern_ops、varchar_pattern_ops和 bpchar_pattern_ops分别支持类型text、varchar和 char上的B树索引。它们与默认操作符类的区别是值的比较是严格按照字符进行而不是根据区域相关的排序规则。这使得这些操作符类适合于当一个数据库没有使用标准“C”区域时被使用在涉及模式匹配表达式（LIKE或POSIX正则表达式）的查询中。一个例子是，你可以这样索引一个varchar列：

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  注意如果你希望涉及到<、<=、>或>=比较的查询使用一个索引，你也应该创建一个使用默认操作符类的索引。这些查询不能使用xxx_pattern_ops操作符类（但是普通的等值比较可以使用这些操作符类）。可以在同一个列上创建多个使用不同操作符类的索引。如果你正在使用C区域，你并不需要xxx_pattern_ops操作符类，因为在C区域中的模式匹配查询可以用带有默认操作符类的索引。

下面的查询展示了所有已定义的操作符类：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

一个操作符类实际上只是一个更大的被称为操作符族的结构的一个子集。在多种数据类型具有相似行为的情况下，常常会定义跨数据类型的操作符并且允许索引使用它们。为了实现该目的，这些类型的操作符类必须被分组到同一个操作符族中。跨类型的操作符是该族的成员，但是并不与族内任意一个单独的类相关联。

前一个查询的扩展版本展示了每个操作符类所属的操作符族：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

这个查询展示所有已定义的操作符族和每一个族中包含的所有操作符：

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.11. 索引和排序规则

一个索引在每一个索引列上只能支持一种排序规则。如果需要多种排序规则，你可能需要多个索引。

考虑这些语句：

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

该索引自动使用下层列的排序规则。因此一个如下形式的查询：

```
SELECT * FROM test1c WHERE content > constant;
```

可以使用该索引，因为比较会默认使用列的排序规则。但是，这个索引无法加速涉及到某些其他排序规则的查询。因此对于下面形式的查询：

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

可以创建一个额外的支持"y"排序规则的索引，例如：

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.12. 检查索引使用

尽管PostgreSQL中的索引并不需要维护或调优，但是检查真实的查询负载实际使用了哪些索引仍然非常重要。检查一个独立查询的索引使用情况可以使用EXPLAIN命令，它应用于这种目的的内容在第 14.1 节有介绍。也可以在一个运行中的服务器上收集有关索引使用的总体统计情况，如第 28.2 所述。

很难明确地表达决定创建哪些索引的通用过程。在之前的小节中的例子里有一些典型的情况。通常需要大量的实验才能决定应该创建哪些索引。本小节剩余的部分将给出一些创建索引的提示：

- 总是先运行ANALYZE。这个命令会收集有关表中值分布情况的统计信息。估计一个查询将要返回的行数需要这些信息，而结果行数则被规划器用来为每一个可能的查询计划分配实际的代价。如果没有任何真实的统计信息，将会假定一些默认值，这几乎肯定是不准确的。在没有运行的情况下检查一个应用的索引使用情况是注定要失败的。详见第 24.1.3 和第 24.1.6 节

- 使用真实数据进行实验。使用测试数据来建立索引将会告诉你测试数据需要什么样的索引，但这并不代表真实数据的需要。

  使用非常小的测试数据集是特别致命的。在从100000行中选出1000行时可能会用到索引，但是从100行里选出1行是很难用到索引的，因为100行完全可能放入到一个磁盘页面中，而没有任何计划能够比得上从一个磁盘页面顺序获取的计划。

  在创建测试数据时也要小心，特别是当应用还没有产生时通常是不可避免的。值非常相似、完全随机或以排好序的方式被插入都将使得统计信息倾斜于真实数据中的值分布。

- 如果索引没有被用到，强制使用它们将会对测试非常有用。有一些运行时参数可以关闭多种计划类型（参见第 19.7.1 节。例如，关闭顺序扫描（enable_seqscan）以及嵌套循环连接（enable_nestloop）将强制系统使用一种不同的计划。如果系统仍然选择使用一个顺序扫描或嵌套循环连接，则索引没有被使用的原因可能更加根本，例如查询条件不匹配索引（哪种查询能够使用哪种索引已经在前面的小节中解释过了）。

- 如果强制索引使用确实使用了索引，则有两种可能性：系统是正确的并且索引确实不合适，或者查询计划的代价估计并没有反映真实情况。因此你应该对用索引的查询和不用索引的查询计时。此时EXPLAIN ANALYZE命令就能发挥作用了。

- 如果发现代价估计是错误的，也分为两种可能性。总代价是用每个计划节点的每行代价乘以计划节点的选择度估计来计算的。计划节点的代价估计可以通过运行时参数调整（如第 19.7.2 所述）。不准确的选择度估计可能是由于缺乏统计信息，可以通过调节统计信息收集参数（见ALTER TABLE）来改进。

  如果你不能成功地把代价调整得更合适，那么你可能必须依靠显式地强制索引使用。你也可能希望联系PostgreSQL开发者来检查该问题。

第 12 章 全文搜索

12.1. 介绍

全文搜索（或者文本搜索）提供了确定满足一个查询的自然语言文档的能力，并可以选择将它们按照与查询的相关度排序。最常用的搜索类型是找到所有包含给定查询词的文档并按照它们与查询的相似性顺序返回它们。查询和相似性的概念非常灵活并且依赖于特定的应用。最简单的搜索认为查询是一组词而相似性是查询词在文档中的频度。

文本搜索操作符已经在数据库中存在很多年了。PostgreSQL对文本数据类型提供了~、~*、LIKE和ILIKE操作符，但是它们缺少现代信息系统所要求的很多基本属性：

- 即使对英语也缺乏语言的支持。正则表达式是不够的，因为它们不能很容易地处理派生词，例如satisfies和satisfy。你可能会错过包含satisfies的文档，尽管你可能想要在对于satisfy的搜索中找到它们。可以使用OR来搜索多个派生形式，但是这样做太罗嗦也容易出错（有些词可能有数千种派生）。
- 它们不提供对搜索结果的排序（排名），这使它们面对数以千计被找到的文档时变得无效。
- 它们很慢因为没有索引支持，因此它们必须为每次搜索处理所有的文档。

全文索引允许文档被预处理并且保存一个索引用于以后快速的搜索。预处理包括：

将文档解析成记号。标识出多种类型的记号是有所帮助的，例如数字、词、复杂的词、电子邮件地址，这样它们可以被以不同的方式处理。原则上记号分类取决于相关的应用，但是对于大部分目的都可以使用一套预定义的分类。PostgreSQL使用一个解析器来执行这个步骤。其中提供了一个标准的解析器，并且为特定的需要也可以创建定制的解析器。

将记号转换成词位。和一个记号一样，一个词位是一个字符串，但是它已经被正规化，这样同一个词的不同形式被变成一样。例如，正规化几乎总是包括将大写字母转换成小写形式，并且经常涉及移除后缀（例如英语中的s或es）。这允许搜索找到同一个词的变体形式，而不需要冗长地输入所有可能的变体。此外，这个步骤通常会消除停用词，它们是那些太普通的词，它们对于搜索是无用的（简而言之，记号是文档文本的原始片段，而词位是那些被认为对索引和搜索有用的词）。PostgreSQL使用词典来执行这个步骤。已经提供了多种标准词典，并且为特定的需要也可以创建定制的词典。

为搜索优化存储预处理好的文档。例如，每一个文档可以被表示为正规化的词位的一个有序数组。与词位一起，通常还想要存储用于近似排名的位置信息，这样一个包含查询词更“密集”区域的文档要比那些包含分散的查询词的文档有更高的排名。

词典允许对记号如何被正规化进行细粒度的控制。使用合适的词典，你可以：

- 定义不应该被索引的停用词。
- 使用Ispell把同义词映射到一个单一词。
- 使用一个分类词典把短语映射到一个单一词。
- 使用一个Ispell词典把一个词的不同变体映射到一种规范的形式。
- 使用Snowball词干分析器规则将一个词的不同变体映射到一种规范的形式。

我们提供了一种数据类型tsvector来存储预处理后的文档，还提供了一种类型tsquery来表示处理过的查询（第 8.11 节）。有很多函数和操作符可以用于这些数据类型（第 9.13 节），其中最重要的是匹配操作符@@，它在第 12.1.2 节中介绍。全文搜索可以使用索引来加速（第 12.9 节）。

12.1.1. 什么是一个文档？

一个document是在一个全文搜索系统中进行搜索的单元，例如，一篇杂志文章或电子邮件消息。文本搜索引擎必须能够解析文档并存储词位（关键词）与它们的父文档之间的关联。随后，这些关联会被用来搜索包含查询词的文档。

对于PostgreSQL中的搜索，一个文档通常是一个数据库表中一行内的一个文本形式的域，或者可能是这类域的一个组合（连接），这些域可能存储在多个表或者是动态获取。换句话说，一个文档可能从用于索引的不同部分构建，并且它可能被作为一个整体存储在某个地方。例如：

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS
 document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

> **注意**
>
> 实际上在这些例子查询中，coalesce应该被用来防止一个单一NULL属性导致整个文档的一个NULL结果。

另一种存储文档的可能性是作为文件系统中的简单文本文件。在这种情况下，数据库可以被用来存储全文索引并执行搜索，并且某些唯一标识符可以被用来从文件系统检索文档。但是，从数据库的外面检索文件要求超级用户权限或者特殊函数支持，因此这种方法通常不如把所有数据放在PostgreSQL内部方便。另外，把所有东西放在数据库内部允许方便地访问文档元数据来协助索引和现实。

对于文本搜索目的，每一个文档必须被缩减成预处理后的tsvector格式。搜索和排名被整个在一个文档的tsvector表示上执行 — 只有当文档被选择来显示给用户时才需要检索原始文本。我们因此经常把tsvector说成是文档，但是当然它只是完整文档的一种紧凑表示。

12.1.2. 基本文本匹配

PostgreSQL中的全文搜索基于匹配操作符@@，它在一个tsvector（文档）匹配一个tsquery（查询）时返回true。哪种数据类型写在前面没有影响：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat &
 rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat
 rat'::tsvector;
 ?column?
----------
 f
```

正如以上例子所建议的，一个tsquery并不只是一个未经处理的文本，顶多一个tsvector是这样。一个tsquery包含搜索术语，它们必须是已经正规化的词位，并且可以使用 AND 、OR、NOT 以及 FOLLOWED BY 操作符结合多个术语（语法详见第 8.11.2 节。有几个函数to_tsquery、plainto_tsquery以及phraseto_tsquery可用于将用户书写的文本转换为正确的tsquery，它们会主要采用正则化出现在文本中的词的方法。相似地，to_tsvector被用来解析和正规化一个文档字符串。因此在实际上一个文本搜索匹配可能看起来更像：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
```

```
 ?column?
----------
 t
```

注意如果这个匹配被写成下面这样它将不会成功：

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

因为这里不会发生词rats的正规化。一个tsvector的元素是词位，它被假定为已经正规化好，因此rats不匹配rat。

@@操作符也支持text输出，它允许在简单情况下跳过从文本字符串到tsvector或tsquery的显式转换。可用的变体是：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

前两种我们已经见过。形式text @@ tsquery等价于to_tsvector(x) @@ y。形式text @@ text等价于to_tsvector(x) @@ plainto_tsquery(y)。

在tsquery中，&（AND）操作符指定它的两个参数都必须出现在文档中才表示匹配。类似地，|（OR）操作符指定至少一个参数必须出现，而!（NOT）操作符指定它的参数不出现才能匹配。例如，查询fat & ! rat匹配包含fat但不包含rat的文档。

在<->（FOLLOWED BY） tsquery操作符的帮助下搜索可能的短语，只有该操作符的参数的匹配是相邻的并且符合给定顺序时，该操作符才算是匹配。例如：

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 f
```

FOLLOWED BY 操作符还有一种更一般的版本，形式是<N>，其中N是一个表示匹配词位位置之间的差。<1>和<->相同，而<2>允许刚好一个其他词位出现在匹配之间，以此类推。当有些词是停用词时，phraseto_tsquery函数利用这个操作符来构造一个能够匹配多词短语的tsquery。例如：

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

一种有时候有用的特殊情况是，<0>可以被用来要求两个匹配同一个词的模式。

圆括号可以被用来控制tsquery操作符的嵌套。如果没有圆括号，|的计算优先级最低，然后从低到高依次是&、<->、!。

值得注意的是，当AND/OR/NOT操作符在一个FOLLOWED BY操作符的参数中时，它们表示与不在那些参数中时不同的含义，因为在FOLLOWED BY中匹配的准确位置是有意义的。例如，通常!x仅匹配在任何地方都不包含x的文档。但如果y不是紧接在一个x后面，!x <-> y就会匹配那个y，在文档中其他位置出现的x不会阻止匹配。另一个例子是，x & y通常仅要求x和y均出现在文档中的某处，但是(x & y) <-> z要求x和y在紧挨着z之前的同一个位置匹配。因此这个查询的行为会不同于x <-> z & y <-> z，它将匹配一个含有两个单独序列x z以及y z的文档（这个特定的查询一点用都没有，因为x和y不可能在同一个位置匹配，但是对于前缀匹配模式之类的更复杂的情况，这种形式的查询就会有用武之地）。

12.1.3. 配置

前述的都是简单的文本搜索例子。正如前面所提到的，全文搜索功能包括做更多事情的能力：跳过索引特定词（停用词）、处理同义词并使用更高级的解析，例如基于空白之外的解析。这个功能由文本搜索配置控制。PostgreSQL中有多种语言的预定义配置，并且你可以很容易地创建你自己的配置（psql的\dF命令显示所有可用的配置）。

在安装期间一个合适的配置将被选择并且default_text_search_config也被相应地设置在postgresql.conf中。如果你正在对整个集簇使用相同的文本搜索配置，你可以使用在postgresql.conf中使用该值。要在集簇中使用不同的配置但是在任何一个数据库内部使用同一种配置，使用ALTER DATABASE ... SET。否则，你可以在每个会话中设置default_text_search_config。

依赖一个配置的每一个文本搜索函数都有一个可选的regconfig参数，因此要使用的配置可以被显式指定。只有当这个参数被忽略时，default_text_search_config才被使用。

为了让建立自定义文本搜索配置更容易，一个配置可以从更简单的数据库对象来建立。PostgreSQL的文本搜索功能提供了四类配置相关的数据库对象：

- 文本搜索解析器将文档拆分成记号并分类每个记号（例如，作为词或者数字）。
- 文本搜索词典将记号转变成正规化的形式并拒绝停用词。
- 文本搜索模板提供位于词典底层的函数（一个词典简单地指定一个模板和一组用于模板的参数）。
- 文本搜索配置选择一个解析器和一组用于将解析器产生的记号正规化的词典。

文本搜索解析器和模板是从低层 C 函数构建而来，因此它要求 C 编程能力来开发新的解析器和模板，并且还需要超级用户权限来把它们安装到一个数据库中（在PostgreSQL发布的contrib/区域中有一些附加的解析器和模板的例子）。由于词典和配置只是对底层解析器和模板的参数化和连接，不需要特殊的权限来创建一个新词典或配置。创建定制词典和配置的例子将在本章稍后的部分给出。

12.2. 表和索引

在前一节中的例子演示了使用简单常数字符串进行全文匹配。本节展示如何搜索表数据，以及可选择地使用索引。

12.2.1. 搜索一个表

可以在没有一个索引的情况下做一次全文搜索。一个简单的查询将打印每一个行的title，这些行在其body域中包含词friend：

```
SELECT title
```

```
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

这将还会找到相关的词例如friends和friendly，因为这些都被约减到同一个正规化的词位。

以上的查询指定要使用english配置来解析和正规化字符串。我们也可以忽略配置参数：

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

这个查询将使用由default_text_search_config设置的配置。

一个更复杂的例子是选择 10 个最近的文档，要求它们在title或body中包含create和table：

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为了清晰，我们忽略coalesce函数调用，它可能需要被用来查找在这两个域之中包含NULL的行。

尽管这些查询可以在没有索引的情况下工作，大部分应用会发现这种方法太慢了，除了偶尔的临时搜索。实际使用文本搜索通常要求创建一个索引。

12.2.2. 创建索引

我们可以创建一个GIN索引（第 12.9 节来加速文本搜索：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector('english', body));
```

注意这里使用了to_tsvector的双参数版本。只有指定了一个配置名称的文本搜索函数可以被用在表达式索引（第 11.7 节中。这是因为索引内容必须是没有被default_text_search_config影响的。如果它们被影响，索引内容可能会不一致因为不同的项可能包含被使用不同文本搜索配置创建的tsvector，并且没有办法猜测哪个是哪个。也没有可能正确地转储和恢复这样的一个索引。

由于to_tsvector的双参数版本被使用在上述的索引中，只有一个使用了带有相同配置名的双参数版to_tsvector的查询引用才能使用该索引。即，WHERE to_tsvector('english', body) @@ 'a & b' 可以使用该索引，但WHERE to_tsvector(body) @@ 'a & b'不能。这保证一个索引只能和创建索引项时所用的相同配置一起使用。

可以建立更复杂的表达式索引，在其中配置名被另一个列指定，例如：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector(config_name, body));
```

这里config_name是pgweb表中的一个列。这允许在同一个索引中有混合配置，同时记录哪个配置被用于每一个索引项。例如，如果文档集合包含不同语言的文档，这就可能会有用。同样，要使用索引的查询必须被措辞成匹配，例如WHERE to_tsvector(config_name, body) @@ 'a & b'。

索引甚至可以连接列：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector('english', title || ' ' ||
 body));
```

另一种方法是创建一个单独的tsvector列来保存to_tsvector的输出。这个例子是title和body的连接，使用coalesce来保证当其他域为NULL时一个域仍然能留在索引中：

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

然后我们创建一个GIN索引来加速搜索：

```
CREATE INDEX textsearch_idx ON pgweb USING GIN(textsearchable_index_col);
```

现在我们准备好执行一个快速的全文搜索了：

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

在使用一个单独的列来存储tsvector表示时，有必要创建一个触发器在title或body改变时保证tsvector列为当前值。第 12.4.3 解释了怎样去做。

单独列方法相对于表达式索引的一个优势在于，它不必为了利用索引而在查询中显式地指定文本搜索配置。如上述例子所示，查询可以依赖default_text_search_config。另一个优势是搜索将会更快，因为它不必重做to_tsvector调用来验证索引匹配（在使用 GiST 索引时这一点比使用 GIN 索引时更重要；见第 12.9 节。表达式索引方法更容易建立，但是它要求更少的磁盘空间，因为tsvector表示没有被显式地存储下来。

12.3. 空值文本搜索

要实现全文搜索必须要有一个从文档创建tsvector以及从用户查询创建tsquery的函数。而且我们需要一种有用的顺序返回结果，因此我们需要一个函数能够根据文档与查询的相关性比较文档。还有一点重要的是要能够很好地显示结果。PostgreSQL对所有这些函数都提供了支持。

12.3.1. 解析文档

PostgreSQL提供了函数to_tsvector将一个文档转换成tsvector数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector把一个文本文档解析成记号，把记号缩减成词位，并且返回一个tsvector，它列出了词位以及词位在文档中的位置。文档被根据指定的或默认的文本搜索配置来处理。下面是一个简单例子：

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面这个例子中我们看到，作为结果的tsvector不包含词a、on或it，词rats变成了rat，并且标点符号-被忽略了。

to_tsvector函数在内部调用了一个解析器，它把文档文本分解成记号并且为每一种记号分配一个类型。对于每一个记号，会去查询一个词典列表（第 12.6 节，该列表会根据记号的类型而变化。第一个识别记号的词典产生一个或多个正规化的词位来表示该记号。例如，rats变成rat是因为一个词典识别到该词rats是rat的复数形式。一些词会被识别为停用词（第 12.6.1 节，这将导致它们被忽略，因为它们出现得太频繁以至于在搜索中起不到作用。在我们的例子中有a、on和it是停用词。如果在列表中没有词典能识别该记号，那它将也会被忽略。在这个例子中标点符号-就属于这种情况，因为事实上没有词典会给它分配记号类型（空间符号），即空间记号不会被索引。对于解析器、词典以及要索引哪些记号类型是由所选择的文本搜索配置（第 12.7 节决定的。可以在同一个数据库中有多种不同的配置，并且有用于很多种语言的预定义配置。在我们的例子中，我们使用用于英语的默认配置english。

函数setweight可以被用来对tsvector中的项标注一个给定的权重，这里一个权重可以是四个字母之一：A、B、C或D。这通常被用来标记来自文档不同部分的项，例如标题对正文。稍后，这种信息可以被用来排名搜索结果。

因为to_tsvector(NULL) 将返回NULL，不论何时一个域可能为空时，我们推荐使用coalesce。下面是我们推荐的从一个结构化文档创建一个tsvector的方法：

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

这里我们已经使用了setweight在完成的tsvector标注每一个词位的来源，并且接着将标注过的tsvector值用tsvector连接操作符||合并在一起（第 12.4.1 节给出了关于这些操作符的细节）。

12.3.2. 解析查询

PostgreSQL提供了函数to_tsquery、plainto_tsquery、phraseto_tsquery以及websearch_to_tsquery用来把一个查询转换成tsquery数据类型。to_tsquery提供了比plainto_tsquery和phraseto_tsquery更多的特性，但是它对其输入要求更加严格。websearch_to_tsquery是to_tsquery的一个简化版本，它使用一种可选择的语法，类似于Web搜索引擎使用的语法。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery从querytext创建一个tsquery值，该值由被tsquery操作符& (AND)、| (OR)、! (NOT) 和<-> (FOLLOWED BY) 分隔的单个记号组成。 这些操作符可以使用圆括号分组。换句话说，to_tsquery的输入必须已经遵循tsquery输入的一般规则，如第 8.11.2 节所述。区别在于基本的tsquery输入把记号当作表面值，而to_tsquery 会使用指定的或者默认的配置把每一个记号正规化成一个词位，并且丢弃掉任何根据配置是停用词的记号。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

和在基本tsquery输入中一样，权重可以被附加到每一个词位来限制它只匹配属于那些权重的tsvector词位。例如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

同样，*可以被附加到一个词位来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

这样一个词位将匹配一个tsvector中的任意以给定字符串开头的词。

to_tsquery也能够接受单引号短语。当配置包括一个会在这种短语上触发的分类词典时就是它的主要用处。在下面的例子中，一个分类词典含规则supernovae stars : sn：

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

在没有引号时，to_tsquery将为那些没有被 AND、OR 或者 FOLLOWED BY 操作符分隔的记号产生一个语法错误。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsquery将未格式化的文本querytext转换成一个tsquery值。该文本被解析并被正规化，很像to_tsvector，然后&（AND）布尔操作符被插入到留下来的词之间。

例子：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

注意plainto_tsquery不会识其输入中的tsquery操作符、权重标签或前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

这里，所有输入的标点都被作为空间符号并且丢弃。

```
phraseto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

phraseto_tsquery的行为很像plainto_tsquery，不过前者会在留下来的词之间插入<-> （FOLLOWED BY）操作符而不是& （AND）操作符。还有，停用词也不是简单地丢弃掉，而是通过插入<N>操作符（而不是<->操作符）来解释。在搜索准确的词位序列时这个函数很有用，因为 FOLLOWED BY 操作符不只是检查所有词位的存在性，还会检查词位的顺序。

例子：

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

和plainto_tsquery相似，phraseto_tsquery函数不会识别其输入中的tsquery操作符、权重标签或者前缀匹配标签：

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

websearch_to_tsquery使用一种可供选择的语法从querytext创建一个tsquery值，这种语法中简单的未格式化文本是一个有效的查询。和plainto_tsquery以及phraseto_tsquery不同，它还识别特定的操作符。此外，这个函数绝不会报出语法错误，这就可以把原始的用户提供的输入用于搜索。支持下列语法：

- 无引号文本：不在引号中的文本将被转换成由&操作符分隔的词，就像被plainto_tsquery处理过那样。
- "引号文本"：在引号中的文本将被转换成由<->操作符分隔的词，就像被phraseto_tsquery处理过那样。
- OR：逻辑或将被转换成|操作符。
- -：逻辑非操作符，被转换成!操作符。

示例：

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
----------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
-----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
         websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <->');
```

```
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. 排名搜索结果

排名处理尝试度量文档和一个特定查询的接近程度，这样当有很多匹配时最相关的那些可以被先显示。PostgreSQL提供了两种预定义的排名函数，它们考虑词法、临近性和结构信息；即，它们考虑查询词在文档中出现得有多频繁，文档中的词有多接近，以及词出现的文档部分有多重要。不过，相关性的概念是模糊的并且与应用非常相关。不同的应用可能要求额外的信息用于排名，例如，文档修改时间。内建的排名函数只是例子。你可以编写你自己的排名函数和/或把它们的结果与附加因素整合在一起来适应你的特定需求。

目前可用的两种排名函数是：

ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer]) returns float4

基于向量的匹配词位的频率来排名向量。

ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer]) returns float4

这个函数为给定文档向量和查询计算覆盖密度排名，该方法在 Clarke、Cormack 和 Tudhope 于 1999 年在期刊 "Information Processing and Management" 上的文章 "Relevance Ranking for One to Three Term Queries" 文章中有描述。覆盖密度类似于ts_rank排名，不过它会考虑匹配词位相互之间的接近度。

这个函数要求词位的位置信息来执行其计算。因此它会忽略tsvector中任何“被剥离的”词位。如果在输入中有未被剥离的词位，结果将会是零（strip函数和tsvector中的位置信息的更多内容请见第 12.4.1 节。

对这两个函数，可选的权重参数提供了为词实例赋予更多或更少权重的能力，这种能力是依据它们被标注的情况的。权重数组指定每一类词应该得到多重的权重，按照如下的顺序：

{D-权重, C-权重, B-权重, A-权重}

如果没有提供权重，那么将使用这些默认值：

{0.1, 0.2, 0.4, 1.0}

通常权重被用来标记来自文档特别区域的词，如标题或一个初始的摘要，这样它们可以被认为比来自文档正文的词更重要或更不重要。

由于一个较长的文档有更多的机会包含一个查询术语，因此考虑文档的尺寸是合理的，例如一个一百个词的文档中有一个搜索词的五个实例而零一个一千个词的文档中有该搜索词的五个实例，则前者比后者更相关。两种排名函数都采用一个整数正规化选项，它指定文档长度是否影响其排名以及如何影响。该整数选项控制多个行为，因此它是一个位掩码：你可以使用|指定一个或多个行为（例如，2|4）。

- 0（默认值）忽略文档长度
- 1 用 1 + 文档长度的对数除排名
- 2 用文档长度除排名
- 4 用长度之间的平均调和距离除排名（只被ts_rank_cd实现）
- 8 用文档中唯一词的数量除排名
- 16 用 1 + 文档中唯一词数量的对数除排名
- 32 用排名 + 1 除排名

如果多于一个标志位被指定，转换将根据列出的顺序被应用。

值得注意的是排名函数并不使用任何全局信息，因此它不可能按照某些时候期望地产生一个公平的正规化，从 1% 或 100%。正规化选项 32 （rank/(rank+1)）可以被应用来缩放所有的排名到范围零到一，但是当然这只是一个外观上的改变；它不会影响搜索结果的顺序。

这里是一个例子，它只选择十个最高排名的匹配：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

这是相同的例子使用正规化的排名：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

排名可能会非常昂贵，因为它要求查询每一个匹配文档的tsvector，这可能会涉及很多I/O因而很慢。不幸的是，这几乎不可能避免，因为实际查询常常导致巨大数目的匹配。

12.3.4. 加亮结果

要表示搜索结果，理想的方式是显示每一个文档的一个部分并且显示它是怎样与查询相关的。通常，搜索引擎显示文档片段时会对其中的搜索术语进行标记。PostgreSQL提供了一个函数ts_headline来实现这个功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text
 ]) returns text
```

ts_headline接受一个文档和一个查询，并且从该文档返回一个引用，在其中来自查询的术语会被加亮。被用来解析该文档的配置可以用config指定；如果config被忽略，将会使用default_text_search_config配置。

如果一个options字符串被指定，它必须由一个逗号分隔的列表组成，列表中是一个或多个option=value对。可用的选项是：

- StartSel、StopSel：用来定界文档中出现的查询词的字符串，这用来把它们与其他被引用的词区分开。如果这些字符串包含空格或逗号，你必须把它们加上双引号。
- MaxWords、MinWords：这些数字决定要输出的最长和最短 headline。
- ShortWord：长度小于等于这个值的词将被从一个 headline 的开头或结尾处丢掉。默认值三消除普通英语文章。
- HighlightAll：布尔标志，如果为true整个文档将被用作 headline，并忽略前面的三个参数。
- MaxFragments：要显示的文本引用或片段的最大数量。默认值零选择一种非片段倾向的 headline 生成方法。一个大于零的值选择基于片段的 headline 生成。这种方法找到有尽可能多查询词的文本片段并且展开查询词周围的那些片段。结果是查询词会靠近每个片段的中间并且在其两侧都有词。每一个片段将是最多MaxWords并且长度小于等于ShortWord的词被从每个片段的开头或结尾丢弃。如果不是所有的查询词都在该文档中找到，文档中第一个MinWords的单一片段将被显示。
- FragmentDelimiter：当多于一个片段被显示时，片段将被这个字符串所分隔。

这些选项名称不区分大小写。任何未指定的选项将收到这些默认值：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

例如：

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                       ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

ts_headline使用原始文档，而不是一个tsvector摘要，因此它可能很慢并且应该被小心使用。

12.4. 额外特性

这一节描述在文本搜索中有用的一些额外的函数和操作符。

12.4.1. 操纵文档

第 12.3.1 展示了未经处理的文本文档如何被转换成tsvector值。PostgreSQL也提供了用于操纵已经为tsvector形式的文档的函数和操作符。

tsvector || tsvector

tsvector连接操作符返回一个向量，它结合了作为参数给出的两个向量的词位和位置信息。位置和权重标签在连接期间被保留。出现在右手向量中的位置被使用左手向量中提到的最大位置进行偏移，这样结果几乎等于在两个原始文档字符串的连接上执行to_tsvector的结果（这种等价不是完全的，因为从左手参数的尾端移除的任何停用词将会影响结果，而如果文本连接被使用它们就影响了右手参数中的词位位置）。

使用向量形式的连接而不是在应用to_tsvector之前连接文本的一个优点是你可以使用不同配置来解析文档的不同小节。此外，因为setweight函数按照相同的方式标记给定向量的所有词位，如果你想把文档的不同部分标注不同的权重，你就有必要解析文本并且在连接之前做setweight。

setweight(vector tsvector, weight "char") returns tsvector

setweight返回输入向量的一个拷贝，其中每一个位置都被标注为给定的权重：A、B、C或D（D是新向量的默认值并且并不会被显示在输出上）。向量被连接时会保留这些标签，允许来自文档的不同部分的词被排名函数给予不同的权重。

注意权重标签是应用到位置而不是词位。如果输入向量已经被剥离了位置，则setweight什么也不会做。

length(vector tsvector) returns integer

返回存储在向量中的词位数。

strip(vector tsvector) returns tsvector

返回一个向量，其中列出了和给定向量相同的词位，不过没有任何位置或者权重信息。其结果通常比未被剥离的向量小很多，但是用处也小很多。和未被剥离的向量一样，相关度排名在已剥离的向量上也不起作用。此外，<-> (FOLLOWED BY) tsquery操作符不会匹配已剥离的输入，因为它无法确定词位之间的距离。

表 9.41中有tsvector相关函数的完整列表。

12.4.2. 操纵查询

第 12.3.2 展示了未经处理的文本形式的查询如何被转换成tsquery值。PostgreSQL也提供了用于操纵已经是tsquery形式的查询的函数和操作符。

tsquery && tsquery

返回用 AND 结合的两个给定查询。

tsquery || tsquery

返回用 OR 结合的两个给定查询。

!! tsquery

返回一个给定查询的反（NOT）。

tsquery <-> tsquery

返回一个查询，它用<->（FOLLOWED BY）tsquery操作符搜索两个紧跟的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
             ?column?
-----------------------------------
 'fat' <-> 'cat' | 'fat' <-> 'rat'
```

tsquery_phrase(query1 tsquery, query2 tsquery [, distance integer]) returns tsquery

返回一个查询，它使用<N> tsquery操作符搜索两个距离为distance个词位的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

numnode(query tsquery) returns integer

返回一个tsquery中的结点数（词位外加操作符）。要确定查询是否有意义或者是否只包含停用词时，这个函数有用，在前一种情况它返回 > 0，后一种情况返回 0。例子：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s),
 ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

querytree(query tsquery) returns text

返回一个tsquery中可以被用来搜索一个索引的部分。这个函数可用来检测不可被索引的查询，例如那些只包含停用词或者只有否定术语的查询。例如：

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. 查询重写

ts_rewrite函数族在一个给定的tsquery中搜索一个目标子查询的出现，并且将每一次出现替换成一个替补子查询。本质上这个操作就是一个tsquery版本的子串替换。一个目标和替补的组合可以被看成是一个查询重写规则。一个这类重写规则的集合可以是一个强大的搜索

助手。例如，你可以使用同义词扩展搜索（如，new york、big apple、nyc、gotham），或者收缩搜索来将用户导向某些特点主题。在这个特性和分类词典（第 12.6.4 节有些功能重叠。但是，你可以随时修改一组重写规则而无需重索引，而更新一个分类词典则要求进行重索引才能生效。

ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery

这种形式的ts_rewrite简单地应用一个单一重写规则：不管target出现在query中的那个地方，它都被substitute替代。例如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

ts_rewrite (query tsquery, select text) returns tsquery

这种形式的ts_rewrite接受一个开始query和一个 SQL select命令，它们以一个文本字符串的形式给出。select必须得到tsquery类型的两列。对于select结果的每一行，在当前query值中出现的第一列值（目标）被第二列值（替补）所替换。例如：

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

注意当多个重写规则被以这种方式应用时，应用的顺序很重要；因此在实际中你会要求源查询按某些排序键ORDER BY。

让我们考虑一个现实的天文学例子。我们将使用表驱动的重写规则扩展查询supernovae：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|
sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

我们可以通过只更新表来改变重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
---------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

当有很多重写规则时，重写可能会很慢，因为它要为为每一个可能的匹配检查每一条规则。要过滤掉明显不符合的规则，我们可以为tsquery类型使用包含操作符。在下面的例子中，我们只选择那些可能匹配原始查询的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. 用于自动更新的触发器

当使用一个单独的列来存储你的文档的tsvector表示时，有必要创建一个触发器在文档内容列改变时更新tsvector列。两个内建触发器函数可以用于这个目的，或者你可以编写你自己的触发器函数。

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name
 [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name
 [, ... ])
```

这些触发器函数在CREATE TRIGGER命令中指定的参数控制下，自动从一个或多个文本列计算一个tsvector列。它们使用的一个例子是：

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

在创建了这个触发器后，在title或body中的任何修改将自动地被反映到tsv中，不需要应用来操心同步的问题。

第一个触发器参数必须是要被更新的tsvector列的名字。第二个参数指定要被用来执行转换的文本搜索配置。对于tsvector_update_trigger，配置名被简单地用第二个触发器参数给出。如上所示，它必须是模式限定的，因此该触发器行为不会因为search_path中的改变而改变。对于tsvector_update_trigger_column，第二个触发器参数是另一个表列的名称，它必须是类型regconfig。这允许做一种逐行的配置选择。剩下的参数是文本列的名称（类型为text、varchar或char）。它们将按给定的顺序被包括在文档中。NULL 值将被跳过（但是其他列仍将被索引）。

这些内建触发器的一个限制是它们将所有输入列同样对待。要对列进行不同的处理 — 例如，使标题的权重和正文的不同 — 就需要编写一个自定义触发器。下面是用PL/pgSQL作为触发器语言的一个例子：

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A')
 ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

记住当在触发器内创建tsvector值时，显式地指定配置名非常重要，这样列的内容才不会被default_text_search_config的改变所影响。如果不这样做很可能导致问题，例如在转储并重新载入之后搜索结果改变。

12.4.4. 收集文档统计数据

ts_stat被用于检查你的配置以及寻找候选的停用词。

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlquery是一个文本值，它包含一个必须返回单一tsvector列的 SQL 查询。ts_stat执行该查询并返回有关包含在该tsvector数据中的每一个可区分词位（词）的统计数据。返回的列是：

- word text — 一个词位的值
- ndoc integer — 词出现过的文档（tsvector）的数量
- nentry integer — 词出现的总次数

如果提供了权重，只有具有其中之一权重的出现才被计算在内。

例如，要在一个文档集合中查找十个最频繁的词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的要求，但是只计算以权重A或B出现的次数：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. 解析器

文本搜索解析器负责把未处理的文档文本划分成记号并且标识每一个记号的类型，而可能的类型集合由解析器本身定义。注意一个解析器完全不会修改文本 — 它简单地标识看似有理的词边界。因为这种有限的视野，对于应用相关的自定义解析器的需求就没有自定义字典那么强烈。目前PostgreSQL只提供了一种内建解析器，它已经被证实对很多种应用都适用。

内建解析器被称为pg_catalog.default。它识别 23 种记号类型，如表 12.所示。

表 12.1. 默认解析器的记号类型

别名	描述	例子
asciiword	单词，所有 ASCII 字母	elephant
word	单词，所有字母	mañana
numword	单词，字母和数字	beta1
asciihword	带连字符的单词，所有 ASCII	up-to-date
hword	带连字符的单词，所有字母	lógico-matemática
numhword	带连字符的单词，字母和数字	postgresql-beta1
hword_asciipart	带连字符的单词部分，所有 ASCII	postgresql in the context postgresql-beta1
hword_part	带连字符的单词部分，所有字母	lógico or matemática in the context lógico-matemática
hword_numpart	带连字符的单词部分，字母和数字	beta1 in the context postgresql-beta1
email	Email 地址	foo@example.com
protocol	协议头部	http://
url	URL	example.com/stuff/index.html
host	主机	example.com
url_path	URL 路径	/stuff/index.html, in the context of a URL
file	文件或路径名	/usr/local/foo.txt, if not within a URL
sfloat	科学记数法	-1.234e56
float	十进制记数法	-1.234
int	有符号整数	-1234
uint	无符号整数	1234
version	版本号	8.3.0
tag	XML 标签	<a href="dictionaries.html">
entity	XML 实体	&
blank	空格符号	（其他不识别的任意空白或标点符号）

> **注意**
>
> 解析器的一个“字母”的概念由数据库的区域设置决定，具体是lc_ctype。只包含基本 ASCII 字母的词被报告为一个单独的记号类型，因为有时可以用来区别它们。在大部分欧洲语言中，记号类型word和asciiword应该被同样对待。
>
> email不支持 RFC 5322 定义的所有合法 email 字符。具体来说，对 email 用户名被支持的非字母数字字符只有句点、破折号和下划线。

解析器有可能从同一份文本得出相互覆盖的记号。例如，一个带连字符的词可能会被报告为一整个词或者多个部分：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

这种行为是值得要的，因为它允许对整个复合词和每个部分进行搜索。这里是另一个例子：

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/
index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. 词典

词典被用来消除不被搜索考虑的词（stop words）、并被用来正规化词这样同一个词的不同派生形式将会匹配。一个被成功地正规化的词被称为一个词位。除了提高搜索质量，正规化和移除停用词减小了文档的tsvector表示的尺寸，因而提高了性能。正规化不会总是有语言上的意义并且通常依赖于应用的语义。

一些正规化的例子：

- 语言学的 - Ispell 词典尝试将输入词缩减为一种正规化的形式；词干分析器词典移除词的结尾
- URL位置可以被规范化来得到等效的 URL 匹配：
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- 颜色名可以被它们的十六进制值替换，例如red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- 如果索引数字，我们可以移除某些小数位来缩减可能的数字的范围，因此如果只保留小数点后两位，例如3.14159265359、3.1415926、3.14在正规化后会变得相同。

一个词典是一个程序，它接受一个记号作为输入，并返回：

- 如果输入的记号对词典是已知的，则返回一个词位数组（注意一个记号可能产生多于一个词位）
- 一个TSL_FILTER标志被设置的单一词位，用一个新记号来替换要被传递给后续字典的原始记号（做这件事的一个字典被称为一个过滤字典）
- 如果字典知道该记号但它是一个停用词，则返回一个空数组
- 如果字典不识别该输入记号，则返回NULL

PostgreSQL为许多语言提供了预定义的字典。也有多种预定义模板可以被用于创建带自定义参数的新词典。每一种预定义词典模板在下面描述。如果没有合适的现有模板，可以创建新的；例子见PostgreSQL发布的contrib/区域。

一个文本搜索配置把一个解析器和一组处理解析器输出记号的词典绑定在一起。对于每一中解析器能返回的记号类型，配置都指定了一个单独的词典列表。当该类型的一个记号被解析

器找到时，每一个词典都被按照顺序查询，知道某个词典将其识别为一个已知词。如果它被标识为一个停用词或者没有一个词典识别它，它将被丢弃并且不会被索引和用于搜索。通常，第一个返回非NULL输出的词典决定结果，并且任何剩下的词典都不会被查找；但是一个过滤词典可以将给定词替换为一个被修改的词，它再被传递给后续的词典。

配置一个词典列表的通用规则是将最狭窄、最特定的词典放在第一位，然后是更加通用的词典，以一个非常通用的词典结尾，像一个Snowball词干分析器或什么都识别的simple。例如，对于一个天文学相关的搜索（astro_en　配置）我们可以把记号类型asciiword（ASCII词）绑定到一个天文学术语的分类词典、一个通用英语词典和一个Snowball英语词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

一个过滤词典可以被放置在列表中的任意位置，除了在最后，因为过滤词典放在最后就等于无用。过滤词典可用于部分正规化词来简化后续词典的工作。例如，一个过滤词典可以被用来从音标字母中移除重音符号，就像unaccent模块所做的。

12.6.1. 停用词

停用词是非常常用、在几乎每一个文档中出现并且没有任何区分度的词。因此，在全文搜索的环境中它们可以被忽略。例如，每一段英语文本都包含a和the等次，因此把它们存储在一个索引中是没有用处的。但是，停用词确实会影响在tsvector中的位置，这进而会影响排名：

```
SELECT to_tsvector('english','in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

缺失的位置 1、2、4 是因为停用词。文档的排名计算在使用和不使用停用词的情况下是很不同的：

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list &
 stop'));
 ts_rank_cd
------------
        0.1
```

如何对待停用词是由指定词典决定的。例如，ispell词典首先正规化词并且查看停用词列表，而Snowball词干分析器首先检查停用词的列表。这种不同行为的原因是一冲降低噪声的尝试。

12.6.2. 简单词典

simple词典模板的操作是将输入记号转换为小写形式并且根据一个停用词文件检查它。如果该记号在该文件中被找到，则返回一个空数组，导致该记号被丢弃。否则，该词的小写形式被返回作为正规化的词位。作为一种选择，该词典可以被配置为将非停用词报告为未识别，允许它们被传递给列表中的下一个词典。

下面是一个使用simple模板的词典定义的例子：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

这里，english是一个停用词文件的基本名称。该文件的全名将是$SHAREDIR/tsearch_data/english.stop，其中$SHAREDIR表示PostgreSQL安装的共享数据目录，通常是/usr/local/share/postgresql（如果不确定，使用pg_config --sharedir）。该文件格式是一个词的列表，每行一个。空行和尾部的空格都被忽略，并且大写也被折叠成小写，但是没有其他对该文件内容的处理。

现在我们能够测试我们的词典：

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

如果没有在停用词文件中找到，我们也可以选择返回NULL而不是小写形式的词。这种行为可以通过设置词典的Accept参数为false来选择。继续该例子：

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

在使用默认值Accept = true，只有把一个simple词典放在词典列表的尾部才有用，因为它将不会传递任何记号给后续的词典。相反，Accept = false只有当至少有一个后续词典的情况下才有用。

> **小心**
>
> 大部分类型的词典依赖于配置文件，例如停用词文件。这些文件必须被存储为UTF-8 编码。当它们被读入服务器时，如果存在不同，它们将被翻译成真实的数据库编码。

> **小心**
>
> 通常，当一个词典配置文件第一次在数据库会话中使用时，数据库会话将只读取它一次。如果你修改了一个配置文件并且想强迫现有的会话取得新内容，可

> 以在该词典上发出一个ALTER TEXT SEARCH DICTIONARY命令。这可以是一次“假”更新，它并不实际修改任何参数值。

12.6.3. 同义词词典

这个词典模板被用来创建用于同义词替换的词典。不支持短语（使用分类词典模板（第 12.6.4 节可以支持）。一个同义词词典可以被用来解决语言学问题，例如，阻止一个英语词干分析器词典把词“Paris”缩减成“pari”。在同义词词典中有一行Paris paris并把它放在english_stem词典之前就足够了。例如：

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary |
 lexemes
-----------+-----------------+-------+---------------------------+------------
+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
 {paris}
```

synonym模板要求的唯一参数是SYNONYMS，它是其配置文件的基本名 — 上例中的my_synonyms。该文件的完整名称将是$SHAREDIR/tsearch_data/my_synonyms.syn（其中$SHAREDIR表示PostgreSQL安装的共享数据目录）。该文件格式是每行一个要被替换的词，后面跟着它的同义词，用空白分隔。空行和结尾的空格会被忽略。

synonym模板还有一个可选的参数CaseSensitive，其默认值为false。当CaseSensitive为false时，同义词文件中的词被折叠成小写，这和输入记号一样。当它为true时，词和记号将不会被折叠成小写，但是比较时就好像被折叠过一样。

一个星号（*）可以被放置在配置文件中一个同义词的末尾。这表示该同义词是一个前缀。当项被用在to_tsvector()中时，星号会被忽略；当它被用在to_tsquery()中时，结果将是一个带有前缀匹配标记器（见第 12.3.2 节的查询项。例如，假设我们在$SHAREDIR/tsearch_data/synonym_sample.syn中有这些项：

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

那么我们将得到这些结果：

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
 synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@
 to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. 分类词典

一个分类词典（有时被简写成TZ）是一个词的集合，其中包括了词与短语之间的联系，即广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。

基本上一个分类词典会用一个首选词替换所有非首选词，并且也可选择地保留原始术语用于索引。PostgreSQL的分类词典的当前实现是同义词词典的一个扩展，并增加了短语支持。一个分类词典要求一个下列格式的配置文件：

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

其中冒号（:）符号扮演了一个短语及其替换之间的定界符。

一个分类词典使用一个子词典（在词典的配置中指定）在检查短语匹配之前正规化输入文本。只能选择一个子词典。如果子词典无法识别一个词，将报告一个错误。在这种情况下，你应该移除该词的使用或者让子词典学会这个词。你可以在一个被索引词的开头放上一个星号（*）来跳过在其上应用子词典，但是所有采样词必须被子词典知道。

如果有多个短语匹配输入，则分类词典选择最长的那一个，并且使用最后的定义打破连结。

由子词典识别的特定停用词不能够被指定；改用?标记任何可以出现停用词的地方。例如，假定根据子词典a和the是停用词：

```
? one ? two : swsw
```

匹配a one the two和the one a two；两者都将被swsw替换。

由于一个分类词典具有识别短语的能力，它必须记住它的状态并与解析器交互。一个分类词典使用这些任务来检查它是否应当处理下一个词或者停止累积。分类词典必须被小心地配置。例如，如果分类词典被分配只处理asciiword记号，则一个形如one 7的分类词典定义将不会工作，因为记号类型uint没有被分配给该分类词典。

> **小心**
>
> 在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重索引。对于大多数其他索引类型，例如增加或移除停用词等小改动都不会强制重索引。

12.6.4.1. 分类词典配置

要定义一个新的分类词典，可使用thesaurus模板。例如：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

这里：

- thesaurus_simple是新词典的名称
- mythesaurus是分类词典配置文件的基础名称（它的全名将是$SHAREDIR/tsearch_data/mythesaurus.ths，其中$SHAREDIR表示安装的共享数据目录）。
- pg_catalog.english_stem是要用于分类词典正规化的子词典（这里是一个 Snowball 英语词干分析器）。注意子词典将拥有它自己的配置（例如停用词），但这里没有展示。

现在可以在配置中把分类词典thesaurus_simple绑定到想要的记号类型上，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. 分类词典例子

考虑一个简单的天文学分类词典thesaurus_astro，它包含一些天文学词组合：

```
supernovae stars : sn
crab nebulae : crab
```

下面我们创建一个词典并绑定一些记号类型到一个天文学分类词典以及英语词干分析器：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
```

```
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

现在我们可以看看它如何工作。ts_lexize对于测试一个分类词典用处不大，因为它把它的输入看成是一个单一记号。我们可以用plainto_tsquery和to_tsvector，它们将把其输入字符串打断成多个记号：

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

原则上，如果你对参数加了引号，你可以使用to_tsquery：

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

注意在thesaurus_astro中supernova star匹配supernovae stars，因为我们在分类词典定义中指定了english_stem词干分析器。该词干分析器移除了e和s。

要和替补一样也索引原始短语，只要将它包含在定义的右手部分中：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-------------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell 词典

Ispell词典模板支持词法词典，它可以把一个词的很多不同语言学的形式正规化成相同的词位。例如，一个英语Ispell词典可以匹配搜索词bank的词尾变化和词形变化，例如banking、banked、banks、banks'和bank's。

标准的PostgreSQL发布不包括任何Ispell配置文件。用于很多种语言的词典可以从Ispell[1]得到。此外，也支持一些更现代的词典文件格式 — MySpell[2]（OO < 2.0.1）和Hunspell[3]（OO >= 2.0.2）。一个很大的词典列表在OpenOffice Wiki[4]上可以得到。

要创建一个Ispell词典，执行这三步：

[1] https://www.cs.hmc.edu/~geoff/ispell.html
[2] https://en.wikipedia.org/wiki/MySpell
[3] https://sourceforge.net/projects/hunspell/
[4] https://wiki.services.openoffice.org/wiki/Dictionaries

- 下载词典配置文件。OpenOffice扩展文件的扩展名是.oxt。有必要抽取.aff和.dic文件，把扩展改为.affix和.dict。对于某些词典文件，还需要使用下面的命令把字符转换成 UTF-8 编码（例如挪威语词典）：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- 拷贝文件到$SHAREDIR/tsearch_data目录
- 用下面的命令把文件载入到 PostgreSQL：

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

这里，DictFile、AffFile和StopWords指定词典、词缀和停用词文件的基础名称。停用词文件的格式和前面解释的simple词典类型相同。其他文件的格式在这里没有指定，但是也可以从上面提到的网站获得。

Ispell　词典通常识别一个有限集合的词，这样它们后面应该跟着另一个更广义的词典；例如，一个 Snowball 词典，它可以识别所有东西。

Ispell的.affix文件具有下面的结构：

```
prefixes
flag *A:
    .           >   RE      # As in enter > reenter
suffixes
flag T:
    E           >   ST      # As in late > latest
    [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
    [AEIOU]Y    >   EST     # As in gray > grayest
    [^EY]       >   EST     # As in small > smallest
```

.dict文件具有下面的结构：

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

.dict文件的格式是：

```
basic_form/affix_class_name
```

在.affix文件中，每一个词缀标志以下面的格式描述：

```
condition > [-stripping_letters,] adding_affix
```

这里的条件具有和正则表达式相似的格式。它可以使用分组[...]和[^...]。例如，[AEIOU]Y表示词的最后一个字母是"y"并且倒数第二个字母是"a"、"e"、"i"、"o"或者"u"。[^EY]表示最后一个字母既不是"e"也不是"y"。

Ispell　词典支持划分复合词，这是一个有用的特性。注意词缀文件应该用compoundwords controlled语句指定一个特殊标志，它标记可以参与到复合格式中的词典词：

```
compoundwords  controlled z
```

下面是挪威语的一些例子：

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpell格式是Hunspell格式的一个子集。Hunspell的.affix文件具有下面的结构：

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
SFX T   0     est        [^ey]
```

一个词缀类的第一行是头部。头部后面列出了词缀规则的域：

- 参数名（PFX 或者 SFX）
- 标志（词缀类的名称）
- 从该词的开始（前缀）或者结尾（后缀）剥离字符
- 增加词缀
- 和正则表达式格式类似的条件。

.dict文件看起来和Ispell的.dict文件相似：

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

> **注意**
>
> MySpell 不支持复合词。Hunspell则对复合词有更好的支持。当前，PostgreSQL只实现了 Hunspell 中基本的复合词操作。

12.6.6. Snowball 词典

Snowball词典模板基于 Martin Porter 的一个项目，他是流行的英语 Porter 词干分析算法的发明者。Snowball 现在对许多语言提供词干分析算法（详见Snowball 站点[5]）。每一个算法懂得按照其语言中的拼写，如何缩减词的常见变体形式为一个基础或词干。一个 Snowball 词典要求一个language参数来标识要用哪种词干分析器，并且可以选择地指定一个stopword文件名来给出一个要被消除的词列表（PostgreSQL的标准停用词列表也是由 Snowball 项目提供的）。例如，有一个内建的定义等效于

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
```

[5] http://snowballstem.org/

```
    StopWords = english
);
```

停用词文件格式和已经解释的一样。

一个Snowball词典识别所有的东西，不管它能不能简化该词，因此它应当被放置在词典列表的最后。把它放在任何其他词典前面是没有用处的，因为一个记号永远不会穿过它而进入到下一个词典。

12.7. 配置例子

一个文本搜索配置指定了将一个文档转换成一个tsvector所需的所有选项：用于把文本分解成记号的解析器，以及用于将每一个记号转换成词位的词典。每一次to_tsvector或to_tsquery的调用都需要一个文本搜索配置来执行其处理。配置参数default_text_search_config指定了默认配置的名称，如果忽略了显式的配置参数，文本搜索函数将会使用它。它可以在postgresql.conf中设置，或者使用SET命令为一个单独的会话设置。

有一些预定义的文本搜索配置可用，并且你可以容易地创建自定义的配置。为了便于管理文本搜索对象，可以使用一组SQL命令，并且有多个psql命令可以显示有关文本搜索对象（第 12.10 节的信息。

作为一个例子，我们将创建一个配置pg，从复制内建的english配置开始：

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

我们将使用一个 PostgreSQL 相关的同义词列表，并将它存储在$SHAREDIR/tsearch_data/pg_dict.syn中。文件内容看起来像：

```
postgres    pg
pgsql       pg
postgresql  pg
```

我们定义同义词词典如下：

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

接下来我们注册Ispell词典english_ispell，它有其自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

现在我们可以在配置pg中建立词的映射：

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

我们选择不索引或搜索某些内建配置确实处理的记号类型：

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

现在我们可以测试我们的配置：

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一个步骤是设置会话让它使用新配置，它被创建在public模式中：

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. 测试和调试文本搜索

一个自定义文本搜索配置的行为很容易变得混乱。本节中描述的函数对于测试文本搜索对象有用。你可以测试一个完整的配置，或者独立测试解析器和词典。

12.8.1. 配置测试

函数ts_debug允许简单地测试一个文本搜索配置。

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

ts_debug显示document的每一个记号的信息，记号由解析器产生并由配置的词典处理过。该函数使用由config指定的配置，如果该参数被忽略则使用default_text_search_config指定的配置。

ts_debug为解析器在文本中标识的每一个记号返回一行。被返回的列是：

- alias text — 记号类型的短名称

- description text — 记号类型的描述
- token text — 记号的文本
- dictionaries regdictionary[] — 配置为这种记号类型选择的词典
- dictionary regdictionary — 识别该记号的词典，如果没有词典能识别则为NULL
- lexemes text[] — 识别该记号的词典产生的词位，如果没有词典能识别则为NULL；一个空数组（{}）表示该记号被识别为一个停用词

这里是一个简单的例子：

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

为了一个更广泛的示范，我们先为英语语言创建一个public.english配置和 Ispell 词典：

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;


SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |
 dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------
+---------------+-------------
```

```
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} |
 english_ispell | {}
 blank     | Space symbols   |             | {}                            |
           |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} |
 english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |
           |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
 english_stem   | {supernova}
```

在这个例子中，词Brightest被解析器识别为一个ASCII 词（别名asciiword）。对于这种记号类型，词典列表是english_ispell和english_stem。该词被english_ispell识别，这个词典将它缩减为名词bright。词supernovaes对于english_ispell词典是未知的，因此它被传递给下一个词典，并且幸运地是，它被识别了（实际上，english_stem是一个 Snowball 词典，它识别所有的东西；这也是为什么它被放置在词典列表的尾部）。

词The被english_ispell词典识别为一个停用词（第 12.6.1 节并且将不会被索引。空格也被丢弃，因为该配置没有为它们提供词典。

你可以通过显式地指定你想看哪些列来缩减输出的宽度：

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. 解析器测试

下列函数允许直接测试一个文本搜索解析器。

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts_parse解析给定的document并返回一系列记录，每一个记录对应一个由解析产生的记号。每一个记录包括一个tokid展示分配给记号的类型以及一个token展示记号的文本。例如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

ts_token_type返回一个表，该表描述指定解析器能够识别的每一种记号类型。对于每一种记号类型，该表给出了解析器用来标注该类型记号的整数tokid，还给出了在配置命令中命名该记号类型的alias，以及一个简短的description。例如：

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |                description
-------+-----------------+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. 词典测试

ts_lexize函数帮助词典测试。

```
ts_lexize(dict regdictionary, token text) returns text[]
```

如果输入的token是该词典已知的，则ts_lexize返回一个词位数组；如果记号是词典已知的但是它是一个停用词，则返回一个空数组；或者如果它对词典是未知词，则返回NULL。

例子：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
```

{}

> **注意**
>
> ts_lexize函数期望一个单一记号而不是文本。下面的情况会让它搞混：
>
> ```
> SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```
>
> 分类词典thesaurus_astro确实知道短语supernovae stars，但是ts_lexize会失败，因为它无法解析输入文本而把它当做一个单一记号。可以使用plainto_tsquery或to_tsvector来测试分类词典，例如：
>
> ```
> SELECT plainto_tsquery('supernovae stars');
>  plainto_tsquery
> -----------------
>  'sn'
> ```

12.9. GIN 和 GiST 索引类型

有两种索引可以被用来加速全文搜索。注意全文搜索并非一定需要索引，但是在一个定期会被搜索的列上，通常需要有一个索引。

```
CREATE INDEX name ON table USING GIN(column);
```

创建一个基于 GIN（通用倒排索引）的索引。column必须是tsvector类型。

```
CREATE INDEX name ON table USING GIST(column);
```

创建一个基于 GiST（通用搜索树）的索引。column可以是tsvector或tsquery类型。

GIN 索引是更好的文本搜索索引类型。作为倒排索引，每个词（词位）在 其中都有一个索引项，其中有压缩过的匹配位置的列表。多词搜索可以找到 第一个匹配，然后使用该索引移除缺少额外词的行。GIN 索引只存储 tsvector值的词（词位），并且不存储它们的权重标签。因此， 在使用涉及权重的查询时需要一次在表行上的重新检查。

一个 GiST 索引是有损的，这表示索引可能产生假匹配，并且有必要检查真实的表行来消除这种假匹配（PostgreSQL在需要时会自动做这一步）。GiST 索引之所以是有损的，是因为每一个文档在索引中被表示为一个定长的签名。该签名通过哈希每一个词到一个 n 位串中的一个单一位来产生，通过将所有这些位 OR 在一起产生一个 n 位的文档签名。当两个词哈希到同一个位位置时就会产生假匹配。如果查询中所有词都有匹配（真或假），则必须检索表行查看匹配是否正确。

有损性导致的性能下降归因于不必要的表记录（即被证实为假匹配的记录）获取。因为表记录的随机访问是较慢的，这限制了 GiST 索引的可用性。假匹配的可能性取决于几个因素，特别是唯一词的数量，因此推荐使用词典来缩减这个数量。

注意GIN索引的构件时间常常可以通过增加maintenance_work_mem来改进，而GiST索引的构建时间则与该参数无关。

对大集合分区并正确使用 GIN 和 GiST 索引允许实现带在线更新的快速搜索。分区可以在数据库层面上使用表继承来完成，或者是通过将文档分布在服务器上并收集外部的搜索结果，例如通过外部数据访问。后者是可能的，因为排名函数只使用本地信息。

12.10. psql支持

关于文本搜索配置对象的信息可以在psql中使用一组命令获得：

```
\dF{d,p,t}[+] [PATTERN]
```

可选的+能产生更多细节。

可选参数PATTERN可以是一个文本搜索对象的名称，可以是模式限定的。如果PATTERN被忽略，则所有可见对象的信息都将被显示。PATTERN可以是一个正则表达式并且可以为模式和对象名称提供独立的模式。下面的例子展示了这些特性：

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |


=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

可用的命令是：

```
\dF[+] [PATTERN]
```

列出文本搜索配置（加上+得到更多细节）。

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
```

```
 numword          | simple
 sfloat           | simple
 uint             | simple
 url              | simple
 url_path         | simple
 version          | simple
 word             | russian_stem
```

\dFd[+] [PATTERN]

列出文本搜索词典（加上+得到更多细节）。

```
=> \dFd
                            List of text search dictionaries
   Schema   |      Name       |                        Description

------------+-----------------
+-----------------------------------------------------------
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check
 for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

\dFp[+] [PATTERN]

列出文本搜索解析器（加上+得到更多细节）。

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
  Token name    |               Description
```

```
-----------------+-----------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

\dFt[+] [PATTERN]

列出文本搜索模板（加上+得到更多细节）。

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description

------------+-----------
+------------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for
 stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. 限制

PostgreSQL的文本搜索特性的当前限制是：

- 每一个词位的长度必须小于 2K 字节
- 一个tsvector（词位 + 位置）的长度必须小于 1 兆字节
- 词位的数量必须小于 2^{64}
- tsvector中的位置值必须大于 0 并且小于 16,383
- <N>（FOLLOWED BY）tsquery操作符中的匹配距离不能超过 16,384
- 每个词位不超过 256 个位置
- 一个tsquery中结点（词位 + 操作符）的个数必须小于 32,768

为了对比，PostgreSQL 8.1 的文档包含 10,441 个唯一词，总数 335,420 个词，并且最频繁的词“postgresql”在 655 个文档中被提到 6,127 次。

另一个例子 — PostgreSQL的邮件列表归档在 461,020 条消息的 57,491,343 个词位中包含 910,989 个唯一词。

第 13 章 并发控制

本章描述PostgreSQL数据库系统在多个会话试图同时访问同一数据时的行为。 这种情况的目标是为所有会话提供高效的访问，同时还要维护严格的数据完整性。每个数据库应用开发人员都应该熟悉本章讨论的话题。

13.1. 介绍

PostgreSQL为开发者提供了一组丰富的工具来管理对数据的并发访问。在内部，数据一致性通过使用一种多版本模型（多版本并发控制，MVCC）来维护。这就意味着每个 SQL 语句看到的都只是一小段时间之前的数据快照（一个数据库版本），而不管底层数据的当前状态。这样可以保护语句不会看到可能由其他在相同数据行上执行更新的并发事务造成的不一致数据，为每一个数据库会话提供事务隔离。MVCC避免了传统的数据库系统的锁定方法，将锁争夺最小化来允许多用户环境中的合理性能。

使用MVCC并发控制模型而不是锁定的主要优点是在MVCC中，对查询（读）数据的锁请求与写数据的锁请求不冲突，所以读不会阻塞写，而写也从不阻塞读。甚至在通过使用革新的可序列化快照隔离（SSI）级别提供最严格的事务隔离级别时，PostgreSQL也维持这个保证。

在PostgreSQL里也有表和行级别的锁功能，用于那些通常不需要完整事务隔离并且想要显式管理特定冲突点的应用。不过，恰当地使用MVCC通常会提供比锁更好的性能。另外，由应用定义的咨询锁提供了一个获得不依赖于单一事务的锁的机制。

13.2. 事务隔离

SQL标准定义了四种隔离级别。最严格的是可序列化，在标准中用了一整段来定义它，其中说到一组可序列化事务的任意并发执行被保证效果和以某种顺序一个一个执行这些事务一样。其他三种级别使用并发事务之间交互产生的现象来定义，每一个级别中都要求必须不出现一种现象。注意由于可序列化的定义，在该级别上这些现象都不可能发生（这并不令人惊讶--如果事务的效果与每个时刻只运行一个的相同，你怎么可能看见由于交互产生的现象？）。

在各个级别上被禁止出现的现象是：

脏读

一个事务读取了另一个并行未提交事务写入的数据。

不可重复读

一个事务重新读取之前读取过的数据，发现该数据已经被另一个事务（在初始读之后提交）修改。

幻读

一个事务重新执行一个返回符合一个搜索条件的行集合的查询， 发现满足条件的行集合因为另一个最近提交的事务而发生了改变。

序列化异常

成功提交一组事务的结果与这些事务所有可能的串行执行结果都不一致。

SQL 标准和 PostgreSQL 实现的事务隔离级别在 表 13.1中描述。

表 13.1. 事务隔离级别

隔离级别	脏读	不可重复读	幻读	序列化异常
读未提交	允许，但不在 PG 中	可能	可能	可能

隔离级别	脏读	不可重复读	幻读	序列化异常
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	允许，但不在 PG 中	可能
可序列化	不可能	不可能	不可能	不可能

在PostgreSQL中，你可以请求四种标准事务隔离级别中的任意一种，但是内部只实现了三种不同的隔离级别，即 PostgreSQL 的读未提交模式的行为和读已提交相同。这是因为把标准隔离级别映射到 PostgreSQL 的多版本并发控制架构的唯一合理的方法。

该表格也显示 PostgreSQL 的可重复读实现不允许幻读。而 SQL 标准允许更严格的行为：四种隔离级别只定义了哪种现像不能发生，但是没有定义哪种现像必须发生。可用的隔离级别的行为在下面的小节中详细描述。

要设置一个事务的事务隔离级别，使用SET TRANSACTION命令。

> **重要**
>
> 某些PostgreSQL数据类型和函数关于事务的行为有特殊的规则。特别是，对一个序列的修改（以及用serial声明的一列的计数器）是立刻对所有其他事务可见的，并且在作出该修改的事务中断时也不会被回滚。
> 见第 9.16 和第 8.1.4 节

13.2.1. 读已提交隔离级别

读已提交是PostgreSQL中的默认隔离级别。 当一个事务运行使用这个隔离级别时， 一个查询（没有FOR UPDATE/SHARE子句）只能看到查询开始之前已经被提交的数据， 而无法看到未提交的数据或在查询执行期间其它事务提交的数据。实际上，SELECT查询看到的是一个在查询开始运行的瞬间该数据库的一个快照。不过SELECT可以看见在它自身事务中之前执行的更新的效果，即使它们还没有被提交。还要注意的是，即使在同一个事务里两个相邻的SELECT命令可能看到不同的数据， 因为其它事务可能会在第一个SELECT开始和第二个SELECT开始之间提交。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样： 它们将只找到在命令开始时已经被提交的行。 不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下， 即将进行的更新将等待第一个更新事务提交或者回滚（如果它还在进行中）。 如果第一个更新事务回滚，那么它的作用将被忽略并且第二个事务可以继续更新最初发现的行。 如果第一个更新事务提交，若该行被第一个更新者删除，则第二个更新事务将忽略该行，否则第二个更新者将试图在该行的已被更新的版本上应用它的操作。该命令的搜索条件（WHERE子句）将被重新计算来看该行被更新的版本是否仍然符合搜索条件。如果符合，则第二个更新者使用该行的已更新版本继续其操作。在SELECT FOR UPDATE和SELECT FOR SHARE的情况下，这意味着把该行的已更新版本锁住并返回给客户端。

带有ON CONFLICT DO UPDATE子句的 INSERT行为类似。在读已提交模式，要插入的 每一行将被插入或者更新。除非有不相干的错误出现，这两种结果之一是肯定 会出现的。如果在另一个事务中发生冲突，并且其效果对于INSERT 还不可见，则UPDATE子句将会 影响那个行，即便那一行对于该命令来说没有惯常的可见版本。

带有ON CONFLICT DO NOTHING子句的 INSERT有可能因为另一个效果对 INSERT快照不可见的事务的结果无法让插入进行 下去。再一次，这只是读已提交模式中的情况。

因为上面的规则，正在更新的命令可能会看到一个不一致的快照： 它们可以看到并发更新命令在它尝试更新的相同行上的作用，但是却看不到那些命令对数据库里其它行的作用。这样的行为令读已提交模式不适合用于涉及复杂搜索条件的命令。不过，它对于更简单的情况是正确的。 例如，考虑用这样的命令更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个这样的事务同时尝试修改帐号 12345 的余额，那我们很明显希望第二个事务从账户行的已更新版本上开始工作。 因为每个命令只影响一个已经决定了的行，让它看到行的已更新版本不会导致任何麻烦的不一致性。

在读已提交模式中，更复杂的使用可能产生不符合需要的结果。例如： 考虑一个在数据上操作的DELETE命令，它操作的数据正被另一个命令从它的限制条件中移除或者加入，例如，假定website是一个两行的表，两行的website.hits等于9和10：

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

即便在UPDATE之前有一个website.hits = 10的行，DELETE将不会产生效果。这是因为更新之前的行值9被跳过，并且当UPDATE完成并且DELETE获得一个锁，新行值不再是10而是11，这再也不匹配条件了。

因为在读已提交模式中，每个命令都是从一个新的快照开始的，而这个快照包含在该时刻已提交的事务， 因此同一事务中的后续命令将看到任何已提交的并行事务的效果。以上的焦点在于单个命令是否看到数据库的绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快并且使用简单。 不过，它不是对于所有情况都够用。做复杂查询和更新的应用可能需要比读已提交模式提供的更严格一致的数据库视图。

13.2.2. 可重复读隔离级别

可重复读隔离级别只看到在事务开始之前被提交的数据；它从来看不到未提交的数据或者并行事务在本事务执行期间提交的修改（不过，查询能够看见在它的事务中之前执行的更新，即使它们还没有被提交）。这是比SQL标准对此隔离级别所要求的更强的保证，并且阻止表 13.中描述的除了序列化异常之外的所有现象。如上面所提到的，这是标准特别允许的，标准只描述了每种隔离级别必须提供的最小保护。

这个级别与读已提交不同之处在于，一个可重复读事务中的查询可以看见在事务中第一个非事务控制语句开始时的一个快照，而不是事务中当前语句开始时的快照。因此，在一个单一事务中的后续SELECT命令看到的是相同的数据，即它们看不到其他事务在本事务启动后提交的修改。

使用这个级别的应用必须准备好由于序列化失败而重试事务。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样： 它们将只找到在事务开始时已经被提交的行。 不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下， 可重复读事务将等待第一个更新事务提交或者回滚（如果它还在进行中）。 如果第一个更新事务回滚，那么它的作用将被忽略并且可重复读事务可以继续更新最初发现的行。 但是如果第一个更新事务提交（并且实际更新或删除该行，而不是只锁住它），则可重复读事务将回滚并带有如下消息

```
ERROR:  could not serialize access due to concurrent update
```

因为一个可重复读事务无法修改或者锁住被其他在可重复读事务开始之后的事务改变的行。

当一个应用接收到这个错误消息，它应该中断当前事务并且从开头重试整个事务。在第二次执行中，该事务将见到作为其初始数据库视图一部分的之前提交的改变，这样在使用行的新版本作为新事务更新的起点时就不会有逻辑冲突。

注意只有更新事务可能需要被重试；只读事务将永远不会有序列化冲突。

可重复读模式提供了一种严格的保证，在其中每一个事务看到数据库的一个完全稳定的视图。不过，这个视图并不需要总是和同一级别上并发事务的某些序列化（一次一个）执行保持一致。例如，即使这个级别上的一个只读事务可能看到一个控制记录被更新，这显示一个批处理已经被完成但是不能看见作为该批处理的逻辑组成部分的一个细节记录，因为它读取空值记录的一个较早的版本。如果不小心地使用显式锁来阻塞冲突事务，尝试用运行在这个隔离级别的事务来强制业务规则不太可能正确地工作。

> **注意**
>
> 在PostgreSQL版本 9.1 之前，一个对于可序列化事务隔离级别的请求会提供和这里描述的完全一样的行为。为了保持可序列化行为，现在应该请求可重复读。

13.2.3. 可序列化隔离级别

可序列化隔离级别提供了最严格的事务隔离。这个级别为所有已提交事务模拟序列事务执行；就好像事务被按照序列一个接着另一个被执行，而不是并行地被执行。但是，和可重复读级别相似，使用这个级别的应用必须准备好因为序列化失败而重试事务。事实上，这个隔离级别完全像可重复读一样地工作，除了它会监视一些条件，这些条件可能导致一个可序列化事务的并发集合的执行产生的行为与这些事务所有可能的序列化（一次一个）执行不一致。这种监控不会引入超出可重复读之外的阻塞，但是监控会产生一些负荷，并且对那些可能导致序列化异常的条件的检测将触发一次序列化失败。

例如，考虑一个表mytab，它初始时包含：

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

假设可序列化事务 A 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

并且接着把结果（3）作为一个新行的value插入，新行的class = 2。同时，可序列化事务 B 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

并得到结果 300，它会将其与class = 1插入到一个新行中。然后两个事务都尝试提交。如果其中一个事务运行在可重复读隔离级别，两者都被允许提交；但是由于没有执行的序列化顺序能在结果上一致，使用可序列化事务将允许一个事务提交并且将回滚另一个并伴有这个消息：

```
ERROR:  could not serialize access due to read/write dependencies among
 transactions
```

这是因为，如果 A 在 B 之前执行，B 将计算得到合计值 330 而不是 300，而且相似地另一种顺序将导致 A 计算出一个不同的合计值。

当依赖可序列化事务来阻止异常时，重要的一点是任何从一个持久化用户表读出数据都不被认为是有效的，直到读它的事务已经成功提交为止。即便是对只读事务也是如此，除了在一个可推迟的只读事务中读取的数据是读出以后立刻有效的，因为这样的一个事务在开始读取任何数据之前会等待，直到它能获得一个快照保证来避免这种问题为止。在所有其他情况下，应用不能依靠在一个后来被中断的事务中读取的结果；相反，它们应当重试事务直到它成功。

要保证真正的可序列化，PostgreSQL使用了谓词锁，这意味着它会保持锁，这些锁让它能够判断在它先运行的情况下，什么时候一个写操作会对一个并发事务中之前读取的结果产生影响。在PostgreSQL中，这些锁并不导致任何阻塞，并且因此不会导致一个死锁。它们被用来标识和标志并发可序列化事务之间的依赖性，这些事务的组合可能导致序列化异常。相反，一个想要保证数据一致性的读已提交或可重复读事务可能需要拿走一个在整个表上的锁，这可能阻塞其他尝试使用该表的用户，或者它可能会使用不仅会阻塞其他事务还会导致磁盘访问的SELECT FOR UPDATE或SELECT FOR SHARE。

像大部分其他数据库系统，PostgreSQL中的谓词锁基于被一个事务真正访问的数据。这些谓词锁将显示在pg_locks系统视图中，它们的mode为SIReadLock。这种在一个查询执行期间获得的特别的锁将依赖于该查询所使用的计划，并且在事务过程中多个细粒度锁（如元组锁）可能和少量粗粒度锁（如页面锁）相结合来防止耗尽用于跟踪锁的内存。如果一个READ ONLY事务检测到不会有导致序列化异常的冲突发生，它可以在完成前释放其 SIRead 锁。事实上，READ ONLY事务将常常可以在启动时确立这一事实并避免拿到任何谓词锁。如果你显式地请求一个SERIALIZABLE READ ONLY DEFERRABLE事务，它将阻塞直到它能够确立这一事实（这是唯一一种可序列化事务阻塞但可重复读事务不阻塞的情况）。在另一方面，SIRead 锁常常需要被保持到事务提交之后，直到重叠的读写事务完成。

坚持使用可序列化事务可以简化开发。成功提交的并发可序列化事务的任意集合将得到和一次运行一个相同效果的这种保证意味着，如果你能证明一个单一事务在独自运行时能做正确的事情，则你可以相信它在任何混合的可序列化事务中也能做正确的事情，即使它不知道那些其他事务做了些什么，否则它将不会成功提交。重要的是使用这种技术的环境有一种普遍的方法来处理序列化失败（总是会返回一个 SQLSTATE 值 '40001'），因为它将很难准确地预计哪些事务可能为读/写依赖性做贡献并且需要被回滚来阻止序列化异常。读/写依赖性的监控会产生开销，如重启被序列化失败中止的事务，但是作为在该开销和显式锁及SELECT FOR UPDATE或SELECT FOR SHARE导致的阻塞之间的一种平衡，可序列化事务是在某些环境中最好性能的选择。

虽然PostgreSQL的可序列化事务隔离级别只允许并发事务在能够证明有一种串行执行能够产生相同效果的前提下提交，但它却不能总是阻止在真正的串行执行中不会发生的错误产生。尤其是可能会看到由于可序列化事务重叠执行导致的唯一约束被违背的情况，这些情况即便在尝试插入键之前就显式地检查过该键不存在也会发生。避免这种问题的方法是，确保所有插入可能会冲突的键的可序列化事务首先显式地检查它们能不能那样做。例如，试想一个要求用户输入新键的应用，它会通过尝试查询用户给出的键来检查键是否已经存在，或者是通过选取现有最大的键并且加一来产生一个新键。如果某些可序列化事务不遵循这种协议而直接插入新键，则也可能会报告唯一约束被违背，即便在并发事务串行执行的情况下不会发生唯一约束被违背也是如此。

当依赖可序列化事务进行并发控制时，为了最佳性能应该考虑一下问题：

- 在可能时声明事务为READ ONLY。
- 控制活动连接的数量，如果需要使用一个连接池。这总是一个重要的性能考虑，但是在一个使用可序列化事务的繁忙系统中这尤为重要。
- 只在一个单一事务中放完整性目的所需要的东西。
- 不要让连接不必要地“闲置在事务中”。配置参数idle_in_transaction_session_timeout可以被用来自动断开拖延会话的连接。

- 在那些由于使用可序列化事务自动提供的保护的地方消除不再需要的显式锁、SELECT FOR UPDATE和SELECT FOR SHARE。
- 当系统因为谓词锁表内存短缺而被强制结合多个页面级谓词锁为一个单一的关系级谓词锁时，序列化失败的比例可能会上升。你可以通过增加max_pred_locks_per_transaction、max_pred_locks_per_relation和max_pred_locks_per_page来避免这种情况。
- 一次顺序扫描将总是需要一个关系级谓词锁。这可能导致序列化失败的比例上升。通过缩减random_page_cost和/或增加cpu_tuple_cost来鼓励使用索引扫描将有助于此。一定要在事务回滚和重启数目的任何减少与查询执行时间的任何全面改变之间进行权衡。

13.3. 显式锁定

PostgreSQL提供了多种锁模式用于控制对表中数据的并发访问。 这些模式可以用于在MVCC无法给出期望行为的情境中由应用控制的锁。 同样，大多数PostgreSQL命令会自动要求恰当的锁以保证被引用的表在命令的执行过程中 不会以一种不兼容的方式删除或修改（例如，TRUNCATE无法安全地与同一表中上的其他操作并发地执行，因此它在表上获得一个排他锁来强制这种行为）。

要检查在一个数据库服务器中当前未解除的锁列表，可以使用pg_locks系统视图。 有关监控锁管理器子系统状态的更多信息，请参考第 28 章

13.3.1. 表级锁

下面的列表显示了可用的锁模式和PostgreSQL自动使用它们的场合。 你也可以用LOCK命令显式获得这些锁。请记住所有这些锁模式都是表级锁，即使它们的名字包含“row”单词（这些名称是历史遗产）。 在一定程度上，这些名字反应了每种锁模式的典型用法 — 但是语意却都是一样的。 两种锁模式之间真正的区别是它们有着不同的冲突锁模式集合（参考表 13.2 。 两个事务在同一时刻不能在同一个表上持有属于相互冲突模式的锁（但是，一个事务决不会和自身冲突。例如，它可以在同一个表上获得ACCESS EXCLUSIVE锁然后接着获取ACCESS SHARE锁）。非冲突锁模式可以由许多事务同时持有。 请特别注意有些锁模式是自冲突的（例如，在一个时刻ACCESS EXCLUSIVE锁不能被多于一个事务持有)而其他锁模式不是自冲突的（例如，ACCESS SHARE锁可以被多个事务持有)。

表级锁模式

ACCESS SHARE

只与ACCESS EXCLUSIVE锁模式冲突。

SELECT命令在被引用的表上获得一个这种模式的锁。通常，任何只读取表而不修改它的查询都将获得这种锁模式。

ROW SHARE

与EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

SELECT FOR UPDATE和SELECT FOR SHARE命令在目标表上取得一个这种模式的锁 （加上在被引用但没有选择FOR UPDATE/FOR SHARE的任何其他表上的ACCESS SHARE锁）。

ROW EXCLUSIVE

与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

命令UPDATE、DELETE和INSERT在目标表上取得这种锁模式（加上在任何其他被引用表上的ACCESS SHARE锁）。通常，这种锁模式将被任何修改表中数据的命令取得。

SHARE UPDATE EXCLUSIVE

与SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发模式改变和VACUUM运行的影响。

由VACUUM（不带FULL）、ANALYZE、CREATE INDEX CONCURRENTLY、CREATE STATISTICS和ALTER TABLE VALIDATE以及其他ALTER TABLE的变体获得。

SHARE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据改变的影响。

由CREATE INDEX（不带CONCURRENTLY）取得。

SHARE ROW EXCLUSIVE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据修改所影响，并且是自排他的，这样在一个时刻只能有一个会话持有它。

由CREATE COLLATION、CREATE TRIGGER和很多 ALTER TABLE的很多形式所获得（见 ALTER TABLE）。

EXCLUSIVE

与ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式只允许并发的ACCESS SHARE锁，即只有来自于表的读操作可以与一个持有该锁模式的事务并行处理。

由REFRESH MATERIALIZED VIEW CONCURRENTLY获得。

ACCESS EXCLUSIVE

与所有模式的锁冲突（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE）。这种模式保证持有者是访问该表的唯一事务。

由ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL和REFRESH MATERIALIZED VIEW（不带CONCURRENTLY）命令获取。ALTER TABLE的很多形式也在这个层面上获得锁（见ALTER TABLE）。这也是未显式指定模式的LOCK TABLE命令的默认锁模式。

> **提示**
>
> 只有一个ACCESS EXCLUSIVE锁阻塞一个SELECT（不带FOR UPDATE/SHARE）语句。

一旦被获取，一个锁通常将被持有直到事务结束。 但是如果在建立保存点之后才获得锁，那么在回滚到这个保存点的时候将立即释放该锁。 这与ROLLBACK取消保存点之后所有的影响的原则保持一致。 同样的原则也适用于在PL/pgSQL异常块中获得的锁：一个跳出块的错误将释放在块中获得的锁。

表 13.2. 冲突的锁模式

请求的锁模式	当前的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X

请求的锁模式	当前的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. 行级锁

除了表级锁以外，还有行级锁，在下文列出了行级锁以及在哪些情境下PostgreSQL会自动使用它们。行级锁的完整冲突表请见表 13.3 注意一个事务可能会在相同的行上保持冲突的锁，甚至是在不同的子事务中。但是除此之外，两个事务永远不可能在相同的行上持有冲突的锁。行级锁不影响数据查询，它们只阻塞对同一行的写入者和加锁者。

行级锁模式

FOR UPDATE

FOR UPDATE会导致由SELECT语句检索到的行被锁定，就好像它们要被更新。这可以阻止它们被其他事务锁定、修改或者删除，一直到当前事务结束。也就是说其他尝试UPDATE、DELETE、SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE这些行的事务将被阻塞，直到当前事务结束。反过来，SELECT FOR UPDATE将等待已经在相同行上运行以上这些命令的并发事务，并且接着锁定并且返回被更新的行（或者没有行，因为行可能已被删除）。不过，在一个REPEATABLE READ或SERIALIZABLE事务中，如果一个要被锁定的行在事务开始后被更改，将会抛出一个错误。进一步的讨论请见第 13.4 节

任何在一行上的DELETE命令也会获得FOR UPDATE锁模式，在某些列上修改值的UPDATE也会获得该锁模式。当前UPDATE情况中被考虑的列集合是那些具有能用于外键的唯一索引的列（所以部分索引和表达式索引不被考虑），但是这种要求未来有可能会改变。

FOR NO KEY UPDATE

行为与FOR UPDATE类似，不过获得的锁较弱：这种锁将不会阻塞尝试在相同行上获得锁的SELECT FOR KEY SHARE命令。任何不获取FOR UPDATE锁的UPDATE也会获得这种锁模式。

FOR SHARE

行为与FOR NO KEY UPDATE类似，不过它在每个检索到的行上获得一个共享锁而不是排他锁。一个共享锁会阻塞其他事务在这些行上执行UPDATE、DELETE、SELECT FOR UPDATE或者SELECT FOR NO KEY UPDATE，但是它不会阻止它们执行SELECT FOR SHARE或者SELECT FOR KEY SHARE。

FOR KEY SHARE

行为与FOR SHARE类似，不过锁较弱：SELECT FOR UPDATE会被阻塞，但是SELECT FOR NO KEY UPDATE不会被阻塞。一个键共享锁会阻塞其他事务执行修改键值的DELETE或

者UPDATE，但不会阻塞其他UPDATE，也不会阻止SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE。

PostgreSQL不会在内存里保存任何关于已修改行的信息，因此对一次锁定的行数没有限制。不过，锁住一行会导致一次磁盘写，例如， SELECT FOR UPDATE将修改选中的行以标记它们被锁住，并且因此会导致磁盘写入。

表 13.3. 冲突的行级锁

要求的锁模式	当前的锁模式			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. 页级锁

除了表级别和行级别的锁以外，页面级别的共享/排他锁被用来控制对共享缓冲池中表页面的读/写。 这些锁在行被抓取或者更新后马上被释放。应用开发者通常不需要关心页级锁，我们在这里提到它们只是为了完整。

13.3.4. 死锁

显式锁定的使用可能会增加死锁的可能性，死锁是指两个（或多个）事务相互持有对方想要的锁。例如，如果事务 1 在表 A 上获得一个排他锁，同时试图获取一个在表 B 上的排他锁， 而事务 2 已经持有表 B 的排他锁，同时却正在请求表 A 上的一个排他锁，那么两个事务就都不能进行下去。PostgreSQL能够自动检测到死锁情况并且会通过中断其中一个事务从而允许其它事务完成来解决这个问题（具体哪个事务会被中断是很难预测的，而且也不应该依靠这样的预测）。

要注意死锁也可能会作为行级锁的结果而发生（并且因此，它们即使在没有使用显式锁定的情况下也会发生）。考虑如下情况，两个并发事务在修改一个表。第一个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这样就在指定帐号的行上获得了一个行级锁。然后，第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一个UPDATE语句成功地在指定行上获得了一个行级锁，因此它成功更新了该行。 但是第二个UPDATE语句发现它试图更新的行已经被锁住了，因此它等待持有该锁的事务结束。事务二现在就在等待事务一结束，然后再继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一试图在指定行上获得一个行级锁，但是它得不到：事务二已经持有了这样的锁。所以它要等待事务二完成。因此，事务一被事务二阻塞，而事务二也被事务一阻塞：一个死锁。PostgreSQL将检测这样的情况并中断其中一个事务。

防止死锁的最好方法通常是保证所有使用一个数据库的应用都以一致的顺序在多个对象上获得锁。在上面的例子里，如果两个事务以同样的顺序更新那些行，那么就不会发生死锁。

我们也应该保证一个事务中在一个对象上获得的第一个锁是该对象需要的最严格的锁模式。如果我们无法提前验证这些，那么可以通过重试因死锁而中断的事务来即时处理死锁。

只要没有检测到死锁情况，寻求一个表级或行级锁的事务将无限等待冲突锁被释放。这意味着一个应用长时间保持事务开启不是什么好事（例如等待用户输入）。

13.3.5. 咨询锁

PostgreSQL提供了一种方法创建由应用定义其含义的锁。这种锁被称为咨询锁，因为系统并不强迫其使用 — 而是由应用来保证其正确的使用。咨询锁可用于 MVCC 模型不适用的锁定策略。例如，咨询锁的一种常用用法是模拟所谓“平面文件”数据管理系统典型的悲观锁策略。虽然一个存储在表中的标志可以被用于相同目的，但咨询锁更快、可以避免表膨胀并且会由服务器在会话结束时自动清理。

有两种方法在PostgreSQL中获取一个咨询锁：在会话级别或在事务级别。一旦在会话级别获得了咨询锁，它将被保持直到被显式释放或会话结束。不同于标准锁请求，会话级咨询锁请求不尊重事务语义：在一个后来被回滚的事务中得到的锁在回滚后仍然被保持，并且同样即使调用它的事务后来失败一个解锁也是有效的。一个锁在它所属的进程中可以被获取多次；对于每一个完成的锁请求必须有一个相应的解锁请求，直至锁被真正释放。在另一方面，事务级锁请求的行为更像普通锁请求：在事务结束时会自动释放它们，并且没有显式的解锁操作。这种行为通常比会话级别的行为更方便，因为它使用一个咨询锁的时间更短。对于同一咨询锁标识符的会话级别和事务级别的锁请求按照期望将彼此阻塞。如果一个会话已经持有了一个给定的咨询锁，由它发出的附加请求将总是成功，即使有其他会话在等待该锁；不管现有的锁和新请求是处在会话级别还是事务级别，这种说法都是真的。

和所有PostgreSQL中的锁一样，当前被任何会话所持有的咨询锁的完整列表可以在pg_locks系统视图中找到。

咨询锁和普通锁都被存储在一个共享内存池中，它的尺寸由max_locks_per_transaction和max_connections配置变量定义。 必须当心不要耗尽这些内存，否则服务器将不能再授予任何锁。这对服务器可以授予的咨询锁数量设置了一个上限，根据服务器的配置不同，这个限制通常是数万到数十万。

在使用咨询锁方法的特定情况下，特别是查询中涉及显式排序和LIMIT子句时，由于 SQL 表达式被计算的顺序，必须小心控制锁的获取。例如：

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

在上述查询中，第二种形式是危险的，因为不能保证在锁定函数被执行之前应用LIMIT。这可能导致获得某些应用不期望的锁，并因此在会话结束之前无法释放。 从应用的角度来看，这样的锁将被挂起，虽然它们仍然在pg_locks中可见。

提供的操作咨询锁函数在第 9.26.10 节描述。

13.4. 应用级别的数据完整性检查

对于使用读已提交事务的数据完整性强制业务规则非常困难，因为对每一个语句数据视图都在变化，并且如果一个写冲突发生即使一个单一语句也不能把它自己限制到该语句的快照。

虽然一个可重复读事务在其执行期间有一个稳定的数据视图，在使用MVCC快照进行数据一致性检查时也有一个小问题，它涉及到被称为读/写冲突的东西。如果一个事务写数据并且一个并发事务尝试读相同的数据（不管是在写之前还是之后），它不能看到其他事务的工作。读取事务看起来是第一个执行的，不管哪个是第一个启动或者哪个是第一个提交。如果就到

此为止，则没有问题，但是如果读取者也写入被一个并发事务读取的数据，现在有一个事务好像是已经在前面提到的任何一个事务之前运行。如果看起来最后执行的事务实际上第一个提交，在这些事务的执行顺序图中很容易出现一个环。当这样一个环出现时，完整性检查在没有任何帮助的情况下将不会正确地工作。

正如第 13.2.3 节提到的，可序列化事务仅仅是可重复读事务增加了对读/写冲突的危险模式的非阻塞监控。当检测到一个可能导致表面的执行顺序中产生环的模式，涉及到的一个事务将被回滚来打破该环。

13.4.1. 用可序列化事务来强制一致性

如果可序列化事务隔离级别被用于所有需要一个一致数据视图的写入和读取，不需要其他的工作来保证一致性。在PostgreSQL中，来自于其他环境的被编写成使用可序列化事务来保证一致性的软件应该“只工作”在这一点上。

当使用这种技术时，如果应用软件通过一个框架来自动重试由于序列化错误而回滚的事务，它将避免为应用程序员带来不必要的负担。把default_transaction_isolation设置为serializable可能是个好主意。通过触发器中的事务隔离级别检查来采取某些动作来保证没有其他事务隔离级别被使用（由于疏忽或者为了破坏完整性检查）也是明智的。

性能建议见第 13.2.3 节

> **警告**
>
> 这个级别的使用可序列化事务的完整性保护还没有扩展到热备份模式（第 26.5 节。由于这个原因，那些使用热备份的系统可能想要在主控机上使用可重复读和显式锁定。

13.4.2. 使用显式锁定强制一致性

当可以使用非可序列化写时，要保证一行的当前有效性并保护它不受并发更新的影响，我们必须使用SELECT FOR UPDATE、SELECT FOR SHARE或一个合适的LOCK TABLE 语句（SELECT FOR UPDATE和SELECT FOR SHARE锁只针对并发更新返回行，而LOCK TABLE会锁住整个表）。当从其他环境移植应用到PostgreSQL时需要考虑这些。

关于这些来自其他环境的转换还需要注意的是SELECT FOR UPDATE不保证一个并发事务将不会更新或删除一个被选中的行。要在PostgreSQL中这样做，你必须真正地更新该行，即便没有值需要被改变。SELECT FOR UPDATE 临时阻塞其他事务，让它们不能获取该相同的锁或者执行一个会影响被锁定行的UPDATE或DELETE，但是一旦正持有该所锁的事务提交或回滚，一个被阻塞的事务将继续执行冲突操作，除非当锁被持有时一个该行的实际UPDATE被执行。

在非可序列化MVCC环境下，全局有效性检查需要一些额外的考虑。例如，一个银行应用可能会希望检查一个表中的所有扣款总和等于另外一个表中的收款总和，同时两个表还会被更新。比较两个连续的在读已提交模式下不会可靠工作的SELECT sum(...)命令， 因为第二个查询很可能会包含没有被第一个查询考虑的事务提交的结果。在一个单一的可重复读事务里进行两个求和则给出在可串行化事务开始之前提交的所有事务产生的准确结果 — 但有人可能会合理地置疑在结果被递交的时候，它们是否仍然相关。 如果可重复读事务本身在尝试做一致性检查之前应用了某些变更，那么检查的有用性就更加值得讨论了， 因为现在它包含了一些（但不是全部）事务开始后的变化。 在这种情况下，一个小心的人可能希望锁住所有需要检查的表，这样才能获得一个无可置疑的当前现状的图像。 一个SHARE模式（或者更高）的锁保证在被锁定表中除了当前事务所作的更改之外，没有未提交的更改。

还要注意如果某人正在依赖显式锁定来避免并发更改，那么他应该使用读已提交模式， 或者是在可重复读模式里在执行命令之前小心地获取锁。 在可重复读事务里获取的锁保证了不会有其它修改该表的事务正在运行，但是如果事务看到的快照在获取锁之前， 那么它可能早于表中一些现在已经提交的更改。 一个可重复读事务的快照实际上是在它的第一个查

询或者数据修改命令（SELECT、INSERT、UPDATE或DELETE）开始的时候冻结的，因此我们可以在快照冻结之前显式地获取锁。

13.5. 提醒

一些 DDL 命令（当前只有TRUNCATE和表重写形式的ALTER TABLE）对于 MVCC 不是安全的。这意味着在截断或者重写提交之后，该表将对并发事务（如果它们使用的快照是在 DDL 命令提交前取得的）呈现出空表的形态。这只对没有在该 DDL 命令开始前访问所讨论的表的事务存在问题 — 任何在 DDL 命令开始前访问过该表的事务将持有至少一个 ACCESS SHARE 表锁，这将阻塞该 DDL 命令直到该事务完成。因此这些命令对于目标表上的连续查询将不会造成任何明显的表内容不一致，但是它们可能导致目标表内容和数据库中其他表内容之间的不一致。

对于可序列化事务隔离级别的支持还没有被加入到热备复制目标（在第 26.5 节描述）中。当前在热备模式中支持的最严格的隔离级别是可重复读。虽然在主控机上用可序列化事务执行所有持久化数据库写入将确保所有后备机将最终达到一个一致的状态，但是运行在后备机上的一个可重复读事务有时可能会看到一个短暂的、与主控机上事务的任何串行执行都不一致的状态。

13.6. 锁定和索引

尽管PostgreSQL提供对表数据访问的非阻塞读/写， 但并非PostgreSQL中实现的每一个索引访问方法当前都能够提供非阻塞读/写访问。 不同的索引类型按照下面方法操作：

B-tree、GiST和SP-GiST索引

> 短期的页面级共享/排他锁被用于读/写访问。每个索引行被取得或被插入后立即释放锁。 这些索引类型提供了无死锁情况的最高并发性。

Hash索引

> Hash 桶级别的共享/排他锁被用于读/写访问。锁在整个 Hash 桶处理完成后释放。Hash 桶级锁比索引级的锁提供了更好的并发性但是可能产生死锁，因为锁持有的时间比一次索引操作时间长。

GIN索引

> 短期的页面级共享/排他锁被用于读/写访问。 锁在索引行被插入/抓取后立即释放。但要注意的是一个 GIN 索引值的插入通常导致对每行产生几个索引键的插入，因此 GIN 可能为了插入一个单一值而做大量的工作。

目前，B-tree 索引为并发应用提供了最好的性能。因为它还有比 Hash 索引更多的特性，在那些需要对标量数据进行索引的并发应用中，我们建议使用 B-tree 索引类型。在处理非标量类型数据的时候，B-tree 就没什么用了，应该使用 GiST、SP-GiST 或 GIN 索引替代。

第 14 章 性能提示

查询性能可能受多种因素影响。其中一些因素可以由用户控制，而其它的则属于系统下层设计的基本原理。 本章我们提供一些有关理解和调节PostgreSQL性能的提示。

14.1. 使用EXPLAIN

PostgreSQL为每个收到查询产生一个查询计划。 选择正确的计划来匹配查询结构和数据的属性对于好的性能来说绝对是最关键的，因此系统包含了一个复杂的规划器来尝试选择好的计划。 你可以使用EXPLAIN命令察看规划器为任何查询生成的查询计划。 阅读查询计划是一门艺术，它要求一些经验来掌握，但是本节只试图覆盖一些基础。

本节中的例子都是从 9.3 开发源代码的回归测试数据库中抽取出来的，并且在此之前做过一次VACUUM ANALYZE。你应该能够在自己尝试这些例子时得到相似的结果，但是你的估计代价和行计数可能会小幅变化，因为ANALYZE的统计信息是随机采样而不是精确值，并且代价也与平台有某种程度的相关性。

这些例子使用EXPLAIN的默认“text”输出格式，这种格式紧凑并且便于人类阅读。如果你想把EXPLAIN的输出交给一个程序做进一步分析，你应该使用它的某种机器可读的输出格式（XML、JSON 或 YAML）。

14.1.1. EXPLAIN基础

查询计划的结构是一个计划结点的树。最底层的结点是扫描结点：它们从表中返回未经处理的行。 不同的表访问模式有不同的扫描结点类型：顺序扫描、索引扫描、位图索引扫描。 也还有不是表的行来源，例如VALUES子句和FROM中返回集合的函数，它们有自己的结点类型。如果查询需要连接、聚集、排序、或者在未经处理的行上的其它操作，那么就会在扫描结点之上有其它额外的结点来执行这些操作。 并且，做这些操作通常都有多种方法，因此在这些位置也有可能出现不同的结点类型。 EXPLAIN给计划树中每个结点都输出一行，显示基本的结点类型和计划器为该计划结点的执行所做的开销估计。 第一行（最上层的结点）是对该计划的总执行开销的估计；计划器试图最小化的就是这个数字。

这里是一个简单的例子，只是用来显示输出看起来是什么样的：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于这个查询没有WHERE子句，它必须扫描表中的所有行，因此计划器只能选择使用一个简单的顺序扫描计划。被包含在圆括号中的数字是（从左至右）：

- 估计的启动开销。在输出阶段可以开始之前消耗的时间，例如在一个排序结点里执行排序的时间。
- 估计的总开销。这个估计值基于的假设是计划结点会被运行到完成，即所有可用的行都被检索。不过实际上一个结点的父结点可能很快停止读所有可用的行（见下面的LIMIT例子）。
- 这个计划结点输出行数的估计值。同样，也假定该结点能运行到完成。
- 预计这个计划结点输出的行平均宽度（以字节计算）。

开销是用规划器的开销参数（参见第 19.7.2 节所决定的捏造单位来衡量的。传统上以取磁盘页面为单位来度量开销； 也就是seq_page_cost将被按照习惯设为1.0，其它开销参数将相对于它来设置。 本节的例子都假定这些参数使用默认值。

有一点很重要：一个上层结点的开销包括它的所有子结点的开销。还有一点也很重要：这个开销只反映规划器关心的东西。特别是这个开销没有考虑结果行传递给客户端所花费的时间，这个时间可能是实际花费时间中的一个重要因素；但是它被规划器忽略了，因为它无法通过修改计划来改变（我们相信，每个正确的计划都将输出同样的行集）。

行数值有一些小技巧，因为它不是计划结点处理或扫描过的行数，而是该结点发出的行数。这通常比被扫描的行数少一些， 因为有些被扫描的行会被应用于此结点上的任意WHERE子句条件过滤掉。 理想中顶层的行估计会接近于查询实际返回、更新、删除的行数。

回到我们的例子：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的产生非常直接。如果你执行：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

你会发现tenk1有358个磁盘页面和10000行。 开销被计算为 （页面读取数*seq_page_cost）+（扫描的行数*cpu_tuple_cost）。默认情况下，seq_page_cost是1.0，cpu_tuple_cost是0.01， 因此估计的开销是 (358 * 1.0) + (10000 * 0.01) = 458。

现在让我们修改查询并增加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

请注意EXPLAIN输出显示WHERE子句被当做一个“过滤器”条件附加到顺序扫描计划结点。这意味着该计划结点为它扫描的每一行检查该条件，并且只输出通过该条件的行。因为WHERE子句的存在，估计的输出行数降低了。不过，扫描仍将必须访问所有 10000 行，因此开销没有被降低；实际上开销还有所上升（准确来说，上升了 10000 * cpu_operator_cost）以反映检查WHERE条件所花费的额外 CPU 时间。

这条查询实际选择的行数是 7000，但是估计的rows只是个近似值。如果你尝试重复这个试验，那么你很可能得到略有不同的估计。 此外，这个估计会在每次ANALYZE命令之后改变，因为ANALYZE生成的统计数据是从该表中随机采样计算的。

现在，让我们把条件变得更严格：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                     QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

这里，规划器决定使用一个两步的计划：子计划结点访问访问一个索引来找出匹配索引条件的行的位置，然后上层计划结点实际地从表中取出那些行。独立地抓取行比顺序地读取它们

的开销高很多，但是不是所有的表页面都被访问，这么做实际上仍然比一次顺序扫描开销要少（使用两层计划的原因是因为上层规划结点把索引标识出来的行位置在读取之前按照物理位置排序，这样可以最小化单独抓取的开销。结点名称里面提到的“位图”是执行该排序的机制）。

现在让我们给WHERE子句增加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

新增的条件stringu1 = 'xxx'减少了估计的输出行计数， 但是没有减少开销，因为我们仍然需要访问相同的行集合。 请注意，stringu1子句不能被应用为一个索引条件，因为这个索引只是在unique1列上。 它被用来过滤从索引中检索出的行。因此开销实际上略微增加了一些以反映这个额外的检查。

在某些情况下规划器将更倾向于一个“simple”索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

在这类计划中，表行被按照索引顺序取得，这使得读取它们开销更高，但是其中有一些是对行位置排序的额外开销。你很多时候将在只取得一个单一行的查询中看到这种计划类型。它也经常被用于拥有匹配索引顺序的ORDER BY子句的查询中，因为那样就不需要额外的排序步骤来满足ORDER BY。

如果在WHERE引用的多个行上有独立的索引，规划器可能会选择使用这些索引的一个 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999
 width=0)
               Index Cond: (unique2 > 9000)
```

但是这要求访问两个索引，所以与只使用一个索引并把其他条件作为过滤器相比，它不一定能胜出。如果你变动涉及到的范围，你将看到计划也会相应改变。

下面是一个例子，它展示了LIMIT的效果：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                      QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10
 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

这是和上面相同的查询，但是我们增加了一个LIMIT这样不是所有的行都需要被检索，并且规划器改变了它的决定。注意索引扫描结点的总开销和行计数显示出好像它会被运行到完成。但是，限制结点在检索到这些行的五分之一后就会停止，因此它的总开销只是索引扫描结点的五分之一，并且这是查询的实际估计开销。之所以用这个计划而不是在之前的计划上增加一个限制结点是因为限制无法避免在位图扫描上花费启动开销，因此总开销会是超过那种方法（25个单位）的某个值。

让我们尝试连接两个表，使用我们已经讨论过的列：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10
 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1
 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个计划中，我们有一个嵌套循环连接结点，它有两个表扫描作为输入或子结点。该结点的摘要行的缩进反映了计划树的结构。连接的第一个（或“outer”）子结点是一个与前面见到的相似的位图扫描。它的开销和行计数与我们从SELECT ... WHERE unique1 < 10得到的相同，因为我们将WHERE子句unique1 < 10用在了那个结点上。t1.unique2 = t2.unique2子句现在还不相关，因此它不影响 outer 扫描的行计数。嵌套循环连接结点将为从 outer 子结点得到的每一行运行它的第二个（或“inner”）子结点。当前 outer 行的列值可以被插入 inner 扫描。这里，来自 outer 行的t1.unique2值是可用的，所以我们得到的计划和开销与前面见到的简单SELECT ... WHERE t2.unique2 = constant情况相似（估计的开销实际上比前面看到的略低，是因为在t2上的重复索引扫描会利用到高速缓存）。循环结点的开销则被以 outer 扫描的开销为基础设置，外加对每一个 outer 行都要进行一次 inner 扫描 （10 * 7.87），再加上用于连接处理一点 CPU 时间。

在这个例子里，连接的输出行计数等于两个扫描的行计数的乘积，但通常并不是所有的情况中都如此， 因为可能有同时提及两个表的 额外WHERE子句，并且因此它只能被应用于连接点，而不能影响任何一个输入扫描。这里是一个例子：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                      QUERY PLAN
```

```
------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10
 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46
 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

条件t1.hundred < t2.hundred不能在tenk2_unique2索引中被测试，因此它被应用在连接结点。这缩减了连接结点的估计输出行计数，但是没有改变任何输入扫描。

注意这里规划器选择了“物化”连接的 inner 关系，方法是在它的上方放了一个物化计划结点。这意味着t2索引扫描将只被做一次，即使嵌套循环连接结点需要读取其数据十次（每个来自 outer 关系的行都要读一次）。物化结点在读取数据时将它保存在内存中，然后在每一次后续执行时从内存返回数据。

在处理外连接时，你可能会看到连接计划结点同时附加有“连接过滤器”和普通“过滤器”条件。连接过滤器条件来自于外连接的ON子句，因此一个无法通过连接过滤器条件的行也能够作为一个空值扩展的行被发出。但是一个普通过滤器条件被应用在外连接条件之后并且因此无条件移除行。在一个内连接中这两种过滤器类型没有语义区别。

如果我们把查询的选择度改变一点，我们可能得到一个非常不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
 width=0)
                     Index Cond: (unique1 < 100)
```

这里规划器选择了使用一个哈希连接，在其中一个表的行被放入一个内存哈希表，在这之后其他表被扫描并且为每一行查找哈希表来寻找匹配。同样要注意缩进是如何反映计划结构的：tenk1上的位图扫描是哈希结点的输入，哈希结点会构造哈希表。然后哈希表会返回给哈希连接结点，哈希连接结点将从它的 outer 子计划读取行，并为每一个行搜索哈希表。

另一种可能的连接类型是一个归并连接，如下所示：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
```

```
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

归并连接要求它的输入数据被按照连接键排序。在这个计划中，tenk1数据被使用一个索引扫描排序，以便能够按照正确的顺序来访问行。但是对于onek则更倾向于一个顺序扫描和排序，因为在那个表中有更多行需要被访问（对于很多行的排序，顺序扫描加排序常常比一个索引扫描好，因为索引扫描需要非顺序的磁盘访问）。

一种查看变体计划的方法是强制规划器丢弃它认为开销最低的任何策略，这可以使用第 19.7.1 节描述的启用/禁用标志实现（这是一个野蛮的工具，但是很有用。另见第 14.3 节。例如，如果我们并不认同在前面的例子中顺序扫描加排序是处理表onek的最佳方法，我们可以尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000
 width=244)
```

这显示规划器认为用索引扫描来排序onek的开销要比用顺序扫描加排序的方式高大约12%。当然，下一个问题是是否真的是这样。我们可以通过使用EXPLAIN ANALYZE来仔细研究一下，如下文所述。

14.1.2. EXPLAIN ANALYZE

可以通过使用EXPLAIN的ANALYZE选项来检查规划器估计值的准确性。通过使用这个选项，EXPLAIN会实际执行该查询，然后显示真实的行计数和在每个计划结点中累计的真实运行时间，还会有一个普通EXPLAIN显示的估计值。例如，我们可能得到这样一个结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                              QUERY PLAN
------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377
 rows=10 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
 (actual time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
```

```
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10
 width=0) (actual time=0.024..0.024 rows=10 loops=1)
              Index Cond: (unique1 < 10)
  ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1
 width=244) (actual time=0.021..0.022 rows=1 loops=10)
        Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

注意“actual time”值是以毫秒计的真实时间，而cost估计值被以捏造的单位表示，因此它们不大可能匹配上。在这里面要查看的最重要的一点是估计的行计数是否合理地接近实际值。在这个例子中，估计值都是完全正确的，但是在实际中非常少见。

在某些查询计划中，可以多次执行一个子计划结点。例如，inner 索引扫描可能会因为上层嵌套循环计划中的每一个 outer 行而被执行一次。在这种情况下，loops值报告了执行该结点的总次数，并且 actual time 和行数值是这些执行的平均值。这是为了让这些数字能够与开销估计被显示的方式有可比性。将这些值乘上loops值可以得到在该结点中实际消耗的总时间。在上面的例子中，我们在执行tenk2的索引扫描上花费了总共 0.220 毫秒。

在某些情况中，EXPLAIN ANALYZE会显示计划结点执行时间和行计数之外的额外执行统计信息。例如，排序和哈希结点提供额外的信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774
 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual
 time=0.711..7.427 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
 (actual time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual
 time=0.659..0.659 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
 width=244) (actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=101 width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

排序结点显示使用的排序方法（尤其是，排序是在内存中还是磁盘上进行）和需要的内存或磁盘空间量。哈希结点显示了哈希桶的数量和批数，以及被哈希表所使用的内存量的峰值（如果批数超过一，也将会涉及到磁盘空间使用，但是并没有被显示）。

另一种类型的额外信息是被一个过滤器条件移除的行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                  QUERY PLAN
```

```
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual
 time=0.016..5.107 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

这些值对于被应用在连接结点上的过滤器条件特别有价值。只有在至少有一个被扫描行或者在连接结点中一个可能的连接对被过滤器条件拒绝时，“Rows Removed”行才会出现。

一个与过滤器条件相似的情况出现在“有损”索引扫描中。例如，考虑这个查询，它搜索包含一个指定点的多边形：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                QUERY PLAN
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual
 time=0.044..0.044 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

规划器认为（非常正确）这个采样表太小不值得劳烦一次索引扫描，因此我们得到了一个普通的顺序扫描，其中的所有行都被过滤器条件拒绝。但是如果我们强制使得一次索引扫描可以被使用，我们看到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                      QUERY PLAN
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32)
 (actual time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

这里我们可以看到索引返回一个候选行，然后它会被索引条件的重新检查拒绝。这是因为一个 GiST 索引对于多边形包含测试是 “有损的”：它确实返回覆盖目标的多边形的行，然后我们必须在那些行上做精确的包含性测试。

EXPLAIN有一个BUFFERS选项可以和ANALYZE一起使用来得到更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 >
 9000;

                                                      QUERY PLAN
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual
 time=0.323..0.342 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
```

```
  ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309
rows=0 loops=1)
        Buffers: shared hit=7
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0) (actual time=0.043..0.043 rows=100 loops=1)
              Index Cond: (unique1 < 100)
              Buffers: shared hit=2
        ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999
width=0) (actual time=0.227..0.227 rows=999 loops=1)
              Index Cond: (unique2 > 9000)
              Buffers: shared hit=5
Planning time: 0.088 ms
Execution time: 0.423 ms
```

BUFFERS提供的数字帮助我们标识查询的哪些部分是对 I/O 最敏感的。

记住因为EXPLAIN ANALYZE实际运行查询，任何副作用都将照常发生，即使查询可能输出的任何结果被丢弃来支持打印EXPLAIN数据。如果你想要分析一个数据修改查询而不想改变你的表，你可以在分析完后回滚命令，例如：

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
time=14.628..14.628 rows=0 loops=1)
  ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
time=0.101..0.439 rows=100 loops=1)
        Recheck Cond: (unique1 < 100)
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0) (actual time=0.043..0.043 rows=100 loops=1)
              Index Cond: (unique1 < 100)
Planning time: 0.079 ms
Execution time: 14.727 ms

ROLLBACK;
```

正如在这个例子中所看到的，当查询是一个INSERT、UPDATE或DELETE命令时，应用表更改的实际工作由顶层插入、更新或删除计划结点完成。这个结点之下的计划结点执行定位旧行以及/或者计算新数据的工作。因此在上面，我们看到我们已经见过的位图表扫描，它的输出被交给一个更新结点，更新结点会存储被更新过的行。还有一点值得注意的是，尽管数据修改结点可能要可观的运行时间（这里，它消耗最大份额的时间），规划器当前并没有对开销估计增加任何东西来说明这些工作。这是因为这些工作对每一个正确的查询计划都得做，所以它不影响计划的选择。

当一个UPDATE或者DELETE命令影响继承层次时， 输出可能像这样：

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
```

```
  ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
        Filter: (f1 = 101)
  ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1
width=14)
        Index Cond: (f1 = 101)
  ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1
width=14)
        Index Cond: (f1 = 101)
  ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1
width=14)
        Index Cond: (f1 = 101)
```

在这个例子中，更新节点需要考虑三个子表以及最初提到的父表。因此有四个输入　　的扫描子计划，每一个对应于一个表。为清楚起见，在更新节点上标注了将被更新　的相关目标表，显示的顺序与相应的子计划相同（这些标注是从 PostgreSQL 9.5 开始新增的，在以前的版本中读者必须通过 观察子计划才能知道这些目标表）。

EXPLAIN ANALYZE显示的 Planning time是从一个已解析的查询生成查询计划并进行优化 所花费的时间，其中不包括解析和重写。

EXPLAIN ANALYZE显示的Execution time包括执行器的启动和关闭时间，以及运行被触发的任何触发器的时间，但是它不包括解析、重写或规划的时间。如果有花在执行BEFORE执行器的时间，它将被包括在相关的插入、更新或删除结点的时间内；但是用来执行AFTER 触发器的时间没有被计算，因为AFTER触发器是在整个计划完成后被触发的。在每个触发器（BEFORE或AFTER）也被独立地显示。注意延迟约束触发器将不会被执行，直到事务结束，并且因此根本不会被EXPLAIN ANALYZE考虑。

14.1.3. 警告

在两种有效的方法中EXPLAIN　　ANALYZE所度量的运行时间可能偏离同一个查询的正常执行。首先，由于不会有输出行被递交给客户端，网络传输开销和　　I/O　　转换开销没有被包括在内。其次，由EXPLAIN　ANALYZE所增加的度量符合可能会很可观，特别是在那些gettimeofday()操作系统调用很慢的机器上。你可以使用pg_test_timing工具来度量在你的系统上的计时开销。

EXPLAIN结果不应该被外推到与你实际测试的非常不同的情况。例如，一个很小的表上的结果不能被假定成适合大型表。规划器的开销估计不是线性的，并且因此它可能为一个更大或更小的表选择一个不同的计划。一个极端例子是，在一个只占据一个磁盘页面的表上，你将几乎总是得到一个顺序扫描计划，而不管索引是否可用。规划器认识到它在任何情况下都将采用一次磁盘页面读取来处理该表，因此用额外的页面读取去查看一个索引是没有价值的（我们已经在前面的polygon_tbl例子中见过）。

在一些情况中，实际的值和估计的值不会匹配得很好，但是这并非错误。一种这样的情况发生在计划结点的执行被LIMIT或类似的效果很快停止。例如，在我们之前用过的LIMIT查询中：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT
 2;

                                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2
 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10
 width=244) (actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
```

```
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

索引扫描结点的估计开销和行计数被显示成好像它会运行到完成。但是实际上限制结点在得到两个行之后就停止请求行，因此实际的行计数只有 2 并且运行时间远低于开销估计所建议的时间。这并非估计错误，这仅仅一种估计值和实际值显示方式上的不同。

归并连接也有类似的现象。如果一个归并连接用尽了一个输入并且其中的最后一个键值小于另一个输入中的下一个键值，它将停止读取另一个输入。在这种情况下，不会有更多的匹配并且因此不需要扫描第二个输入的剩余部分。这会导致不读取一个子结点的所有内容，其结果就像在LIMIT中所提到的。另外，如果 outer （第一个）子结点包含带有重复键值的行，inner（第二个）子结点会被倒退并且被重新扫描来找能匹配那个键值的行。EXPLAIN ANALYZE会统计相同 inner 行的重复发出，就好像它们是真实的额外行。当有很多 outer 重复时，对 inner 子计划结点所报告的实际行计数会显著地大于实际在 inner 关系中的行数。

由于实现的限制，BitmapAnd 和 BitmapOr 结点总是报告它们的实际行计数为零。

通常，EXPLAIN输出将显示查询规划器生成的每个计划节点的详细情况。不过，有一些情况中执行器能够确定特定的节点不是必需的。当前，唯一支持这种行动的节点类型是Append节点。这种节点类型有能力丢弃掉确定不会产生查询所需记录的子节点。可以通过EXPLAIN输出中的“Subplans Removed”属性的存在确定已经被移除的节点。

14.2. 规划器使用的统计信息

14.2.1. 单列统计信息

如我们在上一节所见，查询规划器需要估计一个查询要检索的行数，这样才能对查询计划做出好的选择。 本节对系统用于这些估计的统计信息进行一个快速的介绍。

统计信息的一个部分就是每个表和索引中的项的总数，以及每个表和索引占用的磁盘块数。这些信息保存在pg_class表的reltuples和relpages列中。 我们可以用类似下面的查询查看这些信息：

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

这里我们可以看到tenk1包含 10000 行， 它的索引也有这么多行，但是索引远比表小得多（不奇怪）。

出于效率考虑，reltuples和relpages不是实时更新的 ，因此它们通常包含有些过时的值。它们被VACUUM、ANALYZE和几个 DDL 命令（例如CREATE INDEX）更新。一个不扫描全表的VACUUM或ANALYZE操作（常见情况）将以它扫描的部分为基础增量更新reltuples计数，这就导致了一个近似值。在任何情况中，规划器将缩放它在pg_class中找到的值来匹配当前的物理表尺寸，这样得到一个较紧的近似。

大多数查询只是检索表中行的一部分，因为它们有限制要被检查的行的WHERE子句。 因此规划器需要估算WHERE子句的选择度，即符合WHERE子句中每个条件的行的比例。 用于这个

任务的信息存储在pg_statistic系统目录中。 在pg_statistic中的项由ANALYZE和VACUUM ANALYZE命令更新， 并且总是近似值（即使刚刚更新完）。

除了直接查看pg_statistic之外， 手工检查统计信息的时候最好查看它的视图pg_stats。pg_stats被设计为更容易阅读。 而且，pg_stats是所有人都可以读取的，而pg_statistic只能由超级用户读取（这样可以避免非授权用户从统计信息中获取一些其他人的表的内容的信息。pg_stats视图被限制为只显示当前用户可读的表）。例如，我们可以：

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                          Ramp+
         |           |            | I- 880                          Ramp+
         |           |            | Sp Railroad                        +
         |           |            | I- 580                             +
         |           |            | I- 680                          Ramp
 name    | t         |  -0.284859 | I- 880                          Ramp+
         |           |            | I- 580                          Ramp+
         |           |            | I- 680                          Ramp+
         |           |            | I- 580                             +
         |           |            | State Hwy 13                    Ramp
(2 rows)
```

注意，这两行显示的是相同的列，一个对应开始于road表（inherited=t）的完全继承层次， 另一个只包括road表本身（inherited=f）。

ANALYZE在pg_statistic中存储的信息量（特别是每个列的most_common_vals中的最大项数和histogram_bounds数组）可以用ALTER TABLE SET STATISTICS命令为每一列设置， 或者通过设置配置变量default_statistics_target进行全局设置。 目前的默认限制是100 个项。提升该限制可能会让规划器做出更准确的估计（特别是对那些有不规则数据分布的列）， 其代价是在pg_statistic中消耗了更多空间，并且需要略微多一些的时间来计算估计数值。 相比之下，比较低的限制可能更适合那些数据分布比较简单的列。

更多规划器对统计信息的使用可参阅第 70 章

14.2.2. 扩展统计信息

常常可以看到由于查询子句中用到的多个列相互关联而运行着糟糕的执行计划的慢查询。规划器通常会假设多个条件是彼此独立的，这种假设在列值相互关联的情况下是不成立的。由于常规的统计信息天然的针对个体列的性质，它们无法捕捉到跨列关联的知识。不过，PostgreSQL有能力计算多元统计信息，它能捕捉这类信息。

由于可能的列组合数非常巨大，所以不可能自动计算多元统计信息。可以创建扩展统计信息对象（更常被称为统计信息对象）来指示服务器获得跨感兴趣列集合的统计信息。

统计信息对象可以使用CREATE STATISTICS命令创建。这样一个对象的创建仅仅是创建了一个目录项来表示对统计信息有兴趣。实际的数据收集是由ANALYZE（或者是一个手工命令，或者是后台的自动分析）执行的。收集到的值可以在pg_statistic_ext目录中看到。

ANALYZE基于它用来计算常规单列统计信息的表行样本来计算扩展统计信息。由于样本的尺寸会随着表或者表列的统计信息目标（如前一节所述）增大而增加，更大的统计信息目标通常将会导致更准确的扩展统计信息，同时也会导致更多花在计算扩展统计信息之上的时间。

下面的小节介绍当前支持的扩展统计信息类型。

14.2.2.1. 函数依赖

最简单的一类扩展统计信息跟踪函数依赖，这是在数据库范式定义中使用的概念。如果列a的值的知识足以决定列b的值，即不会有两个行具有相同的a值但是有不同的b值，我们就说列b函数依赖于列a。在一个完全规范化的数据库中，函数依赖应该仅存在于主键和超键上。不过，实际上很多数据集合会由于各种原因无法被完全规范化，常见的例子是为了性能而有意地反规范化。即使在一个完全规范化的数据库中，也会有某些列之间的部分关联，这些可以表达成部分函数依赖。

函数依赖的存在直接影响了特定查询中估计的准确性。如果一个查询包含独立列和依赖列上的条件，依赖列上的条件不会进一步降低结果的尺寸。但是如果没有函数依赖的知识，查询规划器将假定条件是独立的，导致对结果尺寸的低估。

要告知规划器有关函数依赖的信息，ANALYZE可以收集跨列依赖的测度。评估所有列组之间的依赖程度可能会昂贵到不可实现，因此数据收集被限制为针对那些在一个统计信息对象中一起出现的列组（用dependencies选项定义）。建议只对强相关的列组创建dependencies统计信息，以避免ANALYZE以及后期查询规划中不必要的开销。

这里是一个收集函数依赖统计信息的例子：

```
CREATE STATISTICS stts (dependencies) ON zip, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxdependencies
  FROM pg_statistic_ext
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxdependencies
---------+---------+-----------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

这里可以看到列1（邮编）完全决定列5（城市），因此系数为1.0，而城市仅决定42%的邮编，意味着有很多城市（58%）有多个邮编。

在为涉及函数依赖列的查询计算选择度时，规划器会使用依赖系数来调整针对条件的选择度估计，这样就不会产生低估。

14.2.2.1.1. 函数依赖的限制

当前只有在考虑简单等值条件（将列与常量值比较）时，函数依赖才适用。不会使用它们来改进比较两个列或者比较列和表达式的等值条件的估计，也不会用它们来改进范围子句、LIKE或者任何其他类型的条件。

在用函数依赖估计时，规划器假定在涉及的列上的条件是兼容的并且因此是冗余的。如果它们是不兼容的，正确的估计将是零行，但是那种可能性不会被考虑。例如，给定一个这样的查询

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

规划器将会忽视city子句，因为它不改变选择度，这是正确的。不过，即便真地只有零行满足下面的查询，规划器也会做出同样的假设

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

不过，函数依赖统计信息无法提供足够的信息来排除这种情况。

在很多实际情况中，这种假设通常是能满足的。例如，在应用程序中可能有一个GUI仅允许选择兼容的城市和邮编值用在查询中。但是如果不是这样，函数依赖可能就不是一个可行的选项。

14.2.2.2. 多元可区分值计数

单列统计信息存储每一列中可区分值的数量。在组合多个列（例如GROUP BY a, b）时，如果规划器只有单列统计数据，则对可区分值数量的估计常常会错误，导致选择不好的计划。

为了改进这种估计，ANALYZE可以为列组收集可区分值统计信息。和以前一样，为每一种可能的列组合做这件事情是不切实际的，因此只会为一起出现在一个统计信息对象（用ndistinct选项定义）中的列组收集数据。将会为列组中列出的列的每一种可能的组合都收集数据。

继续之前的例子，ZIP代码表中的可区分值计数可能像这样：

```
CREATE STATISTICS stts2 (ndistinct) ON zip, state, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxndistinct AS nd
  FROM pg_statistic_ext
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

这表示有三种列组合有33178个可区分值：ZIP代码和州、ZIP代码和城市、ZIP代码+城市+周（事实上对于表中给定的一个唯一的ZIP代码，它们本来就应该是相等的）。另一方面，城市和州的组合只有27435个可区分值。

建议只对实际用于分组的列组合以及分组数错误估计导致了糟糕计划的列组合创建ndistinct统计信息对象。否则，ANALYZE循环只会被浪费。

14.3. 用显式JOIN子句控制规划器

我们可以在一定程度上用显式JOIN语法控制查询规划器。要明白为什么需要它，我们首先需要一些背景知识。

在一个简单的连接查询中，例如：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按照任何顺序连接给定的表。例如，它可以生成一个使用WHERE条件a.id = b.id连接 A 到 B 的查询计划，然后用另外一个WHERE条件把 C 连接到这个连接表。或者它可以先连接 B 和 C 然后再连接 A 得到同样的结果。 或者也可以连接 A 到 C 然后把结果与 B 连接 — 不过这么做效率不好，因为必须生成完整的 A 和 C 的迪卡尔积，而在WHERE子句中没有可用条件来优化该连接（PostgreSQL执行器中的所有连接都发生在两个输入表之间， 所以它必须以这些形式之一建立结果）。 重要的一点是这些不同的连接可能性给出在语义等效的结果，但在执行开销上却可能有巨大的差别。 因此，规划器会对它们进行探索并尝试找出最高效的查询计划。

当一个查询只涉及两个或三个表时，那么不需要考虑很多连接顺序。但是可能的连接顺序数随着表数目的增加成指数增长。 当超过十个左右的表以后，实际上根本不可能对所有可

能性做一次穷举搜索，甚至对六七个表都需要相当长的时间进行规划。 当有太多的输入表时，PostgreSQL规划器将从穷举搜索切换为一种遗传概率搜索，它只需要考虑有限数量的可能性（切换的阈值用geqo_threshold运行时参数设置）。遗传搜索用时更少，但是并不一定会找到最好的计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑：

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询的约束表面上和前一个非常相似，但它们的语义却不同， 因为如果 A 里有任何一行不能匹配 B 和 C的连接表中的行，它也必须被输出。因此这里规划器对连接顺序没有什么选择：它必须先连接 B 到 C，然后把 A 连接到该结果上。 相应地，这个查询比前面一个花在规划上的时间更少。在其它情况下，规划器就有可能确定多种连接顺序都是安全的。例如，给定：

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN完全约束连接顺序。大多数涉及LEFT JOIN或RIGHT JOIN的实际情况都在某种程度上可以被重新排列。

显式连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的， 因此它不约束连接顺序。

即使大多数类型的JOIN并不完全约束连接顺序，但仍然可以指示PostgreSQL查询规划器将所有JOIN子句当作有连接顺序约束来对待。例如，这里的三个查询在逻辑上是等效的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果我们告诉规划器遵循JOIN的顺序，那么第二个和第三个还是要比第一个花在规划上的时间少。 这个效果对于只有三个表的连接而言是微不足道的，但对于数目众多的表，可能就是救命稻草了。

要强制规划器遵循显式JOIN的连接顺序， 我们可以把运行时参数join_collapse_limit设置为 1（其它可能值在下文讨论）。

你不必为了缩短搜索时间来完全约束连接顺序，因为可以在一个普通FROM列表里使用JOIN操作符。例如，考虑：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置join_collapse_limit = 1，那么这就强迫规划器先把 A 连接到 B， 然后再连接到其它的表上，但并不约束它的选择。在这个例子中，可能的连接顺序的数目减少了 5 倍。

按照这种方法约束规划器的搜索是一个有用的技巧，不管是对减少规划时间还是对引导规划器生成好的查询计划。 如果规划器按照默认选择了一个糟糕的连接顺序，你可以通过JOIN语法强迫它选择一个更好的顺序 — 假设你知道一个更好的顺序。我们推荐进行实验。

一个非常相近的影响规划时间的问题是把子查询压缩到它们的父查询中。例如，考虑：

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能在使用包含连接的视图时出现；该视图的SELECT规则将被插入到引用视图的地方，得到与上文非常相似的查询。 通常，规划器会尝试把子查询压缩到父查询里，得到：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这样通常会生成一个比独立的子查询更好些的计划（例如，outer 的WHERE条件可能先把 X 连接到 A 上，这样就消除了 A 中的许多行， 因此避免了形成子查询的全部逻辑输出）。但是同时，我们增加了规划的时间； 在这里，我们用五路连接问题替代了两个独立的三路连接问题。这样的差别是巨大的，因为可能的计划数的是按照指数增长的。 如果有超过from_collapse_limit个FROM项将会导致父查询，规划器将尝试通过停止提升子查询来避免卡在巨大的连接搜索问题中。你可以通过调高或调低这个运行时参数在规划时间和计划的质量之间取得平衡。

from_collapse_limit和join_collapse_limit的命名相似，因为它们做的几乎是同一件事：一个控制规划器何时将把子查询“平面化”，另外一个控制何时把显式连接平面化。通常，你要么把join_collapse_limit设置成和from_collapse_limit一样（这样显式连接和子查询的行为类似）， 要么把join_collapse_limit设置为 1（如果你想用显式连接控制连接顺序）。 但是你可以把它们设置成不同的值，这样你就可以细粒度地调节规划时间和运行时间之间的平衡。

14.4. 填充一个数据库

第一次填充数据库时可能需要插入大量的数据。本节包含一些如何让这个处理尽可能高效的建议。

14.4.1. 禁用自动提交

在使用多个INSERT时，关闭自动提交并且只在最后做一次提交（在普通 SQL 中，这意味着在开始发出BEGIN并且在结束时发出COMMIT。某些客户端库可能背着你就做了这些，在这种情况下你需要确定在你需要做这些时该库确实帮你做了）。如果你允许每一个插入都被独立地提交，PostgreSQL要为每一个被增加的行做很多工作。在一个事务中做所有插入的一个额外好处是：如果一个行的插入失败则所有之前插入的行都会被回滚，这样你不会被卡在部分载入的数据中。

14.4.2. 使用COPY

使用COPY在一条命令中装载所有记录，而不是一系列INSERT命令。 COPY命令是为装载大量行而优化过的； 它没INSERT那么灵活，但是在大量数据装载时导致的负荷也更少。 因为COPY是单条命令，因此使用这种方法填充表时无须关闭自动提交。

如果你不能使用COPY，那么使用PREPARE来创建一个预备INSERT语句也有所帮助，然后根据需要使用EXECUTE多次。这样就避免了重复分析和规划INSERT的负荷。不同接口以不同的方式提供该功能， 可参阅接口文档中的“预备语句”。

请注意，在载入大量行时，使用COPY几乎总是比使用INSERT快， 即使使用了PREPARE并且把多个插入被成批地放入一个单一事务。

同样的事务中，COPY比更早的CREATE TABLE或TRUNCATE命令更快。 在这种情况下，不需要写 WAL，因为在一个错误的情况下，包含新载入数据的文件不管怎样都将被移除。不过，只有当wal_level设置为minimal（此时所有的命令必须写 WAL）时才会应用这种考虑。

14.4.3. 移除索引

如果你正在载入一个新创建的表，最快的方法是创建该表，用COPY批量载入该表的数据，然后创建表需要的任何索引。在已存在数据的表上创建索引要比在每一行被载入时增量地更新它更快。

如果你正在对现有表增加大量的数据，删除索引、载入表然后重新创建索引可能是最好的方案。 当然，在缺少索引的期间，其它数据库用户的数据库性能将会下降。 我们在删除唯一索引之前还需要仔细考虑清楚，因为唯一约束提供的错误检查在缺少索引的时候会丢失。

14.4.4. 移除外键约束

和索引一样，“成批地”检查外键约束比一行行检查效率更高。 因此，先删除外键约束、载入数据然后重建约束会很有用。 同样，载入数据和约束缺失期间错误检查的丢失之间也存在平衡。

更重要的是，当你在已有外键约束的情况下向表中载入数据时， 每个新行需要一个在服务器的待处理触发器事件（因为是一个触发器的触发会检查行的外键约束）列表的条目。载入数百万行会导致触发器事件队列溢出可用内存， 造成不能接受的交换或者甚至是命令的彻底失败。因此在载入大量数据时，可能需要（而不仅仅是期望）删除并重新应用外键。如果临时移除约束不可接受，那唯一的其他办法可能是就是将载入操作分解成更小的事务。

14.4.5. 增加maintenance_work_mem

在载入大量数据时，临时增大maintenance_work_mem配置变量可以改进性能。这个参数也可以帮助加速CREATE INDEX命令和ALTER TABLE ADD FOREIGN KEY命令。 它不会对COPY本身起很大作用，所以这个建议只有在你使用上面的一个或两个技巧时才有用。

14.4.6. 增加max_wal_size

临时增大max_wal_size配置变量也可以让大量数据载入更快。 这是因为向PostgreSQL中载入大量的数据将导致检查点的发生比平常（由checkpoint_timeout配置变量指定）更频繁。无论何时发生一个检查点时，所有脏页都必须被刷写到磁盘上。 通过在批量数据载入时临时增加max_wal_size，所需的检查点数目可以被缩减。

14.4.7. 禁用 WAL 归档和流复制

当使用 WAL 归档或流复制向一个安装中载入大量数据时，在录入结束后执行一次新的基础备份比处理大量的增量 WAL 数据更快。为了防止载入时记录增量 WAL，通过将wal_level设置为minimal、将archive_mode设置为off以及将max_wal_senders设置为零来禁用归档和流复制。 但需要注意的是，修改这些设置需要重启服务。

除了避免归档器或 WAL 发送者处理 WAL 数据的时间之外，这样做将实际上使某些命令更快， 因为它们被设计为在wal_level为minimal时完全不写 WAL （通过在最后执行一个fsync而不是写 WAL，它们能以更小地代价保证崩溃安全）。这适用于下列命令：

- CREATE TABLE AS SELECT
- CREATE INDEX（以及类似 ALTER TABLE ADD PRIMARY KEY的变体）
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM，当目标表已经被创建或者在同一个事务的早期被截断

14.4.8. 事后运行ANALYZE

不管什么时候你显著地改变了表中的数据分布后，我们都强烈推荐运行ANALYZE。着包括向表中批量载入大量数据。运行ANALYZE（或者VACUUM ANALYZE）保证规划器有表的最新统计信息。 如果没有统计数据或者统计数据过时，那么规划器在查询规划时可能做出很差劲决定，导致在任意表上的性能低下。需要注意的是，如果启用了 autovacuum 守护进程，它可能会自动运行ANALYZE；参阅第 24.1.3 和第 24.1.6 节。

14.4.9. 关于pg_dump的一些注记

pg_dump生成的转储脚本自动应用上面的若干个（但不是全部）技巧。 要尽可能快地载入pg_dump转储，你需要手工做一些额外的事情（请注意，这些要点适用于恢复一个转储，而不是创建它的时候。同样的要点也适用于使用psql载入一个文本转储或用pg_restore从一个pg_dump归档文件载入）。

默认情况下，pg_dump使用COPY，并且当它在生成一个完整的模式和数据转储时， 它会很小心地先装载数据，然后创建索引和外键。因此在这种情况下，一些指导方针是被自动处理的。你需要做的是：

- 为maintenance_work_mem和max_wal_size设置适当的（即比正常值大的）值。
- 如果使用 WAL 归档或流复制，在转储时考虑禁用它们。在载入转储之前，可通过将archive_mode设置为off、将wal_level设置为minimal以及将max_wal_senders设置为零（在录入dump前）来实现禁用。 之后，将它们设回正确的值并执行一次新的基础备份。
- 采用pg_dump和pg_restore的并行转储和恢复模式进行实验并且找出要使用的最佳并发任务数量。通过使用-j选项的并行转储和恢复应该能为你带来比串行模式高得多的性能。
- 考虑是否应该在一个单一事务中恢复整个转储。要这样做，将-1或--single-transaction命令行选项传递给psql或pg_restore。 当使用这种模式时，即使是一个很小的错误也会回滚整个恢复，可能会丢弃已经处理了很多个小时的工作。根据数据间的相关性， 可能手动清理更好。如果你使用一个单一事务并且关闭了 WAL 归档，COPY命令将运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用pg_restore的--jobs选项。这允许并行数据载入和索引创建。
- 之后运行ANALYZE。

一个只涉及数据的转储仍将使用COPY，但是它不会删除或重建索引，并且它通常不会触碰外键。 [1] 因此当载入一个只有数据的转储时，如果你希望使用那些技术，你需要负责删除并重建索引和外键。在载入数据时增加max_wal_size仍然有用，但是不要去增加maintenance_work_mem；不如说在以后手工重建索引和外键时你已经做了这些。并且不要忘记在完成后执行ANALYZE，详见第 24.1.3 和第 24.1.6 节

14.5. 非持久设置

持久性是数据库的一个保证已提交事务的记录的特性（即使是发生服务器崩溃或断电）。然而，持久性会明显增加数据库的负荷，因此如果你的站点不需要这个保证，PostgreSQL可以被配置成运行更快。在这种情况下，你可以调整下列配置来提高性能。除了下面列出的，在数据库软件崩溃的情况下也能保证持久性。当这些设置被使用时，只有突然的操作系统停止会产生数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即RAM磁盘）。这消除了所有的数据库磁盘 I/O，但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭fsync；不需要将数据刷入磁盘。
- 关闭synchronous_commit；可能不需要在每次提交时 强制把WAL写入磁盘。这种设置可能会在 数据库崩溃时带来事务丢失的风险（但是没有数据破坏）。
- 关闭full_page_writes；不需要警惕部分页面写入。
- 增加max_wal_size和checkpoint_timeout； 这会降低检查点的频率，但会 增加/pg_wal的存储要求。

[1] 你可以通过使用--disable-triggers选项的方法获得禁用外键的效果 — 不过你要意识到这么做是消除（而不只是推迟）外键验证。因此如果你使用该选项，就可能插入坏数据。

- 创建不做日志的表 来避免WAL写入，不过这会让表在崩溃时不安全。

第 15 章 并行查询

PostgreSQL能设计出利用多 CPU 让查询更快的查询计划。这种特性被称为并行查询。由于现有实现的限制或者因为没有比连续查询计划更快的查询计划存在，很多查询并不能从并行查询获益。不过，对于那些可以从并行查询获益的查询来说，并行查询带来的速度提升是显著的。很多查询在使用并行查询时比之前快了超过两倍，有些查询是以前的四倍甚至更多的倍数。那些访问大量数据但只返回其中少数行给用户的查询最能从并行查询中获益。这一章介绍一些并行查询如何工作的细节以及哪些情况下可以使用并行查询，这样希望充分利用并行查询的用户可以理解他们能从并行查询得到什么。

15.1. 并行查询如何工作

当优化器判断对于某一个特定的查询，并行查询是最快的执行策略时，优化器将创建一个查询计划。该计划包括一个 Gather或者Gather Merge节点。下面是一个简单的例子：

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1
 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

在所有的情形下，Gather或Gather Merge节点都只有一个子计划，它是将被并行执行的计划的一部分。如果Gather或Gather Merge节点位于计划树的最顶层，那么整个查询将并行执行。如果它位于计划树的其他位置，那么只有查询中在它之下的那一部分会并行执行。在上面的例子中，查询只访问了一个表，因此除Gather节点本身之外只有一个计划节点。因为该计划节点是Gather节点的孩子节点，所以它会并行执行。

使用 EXPLAIN命令， 你能看到规划器选择的工作者数量。当查询执行期间到达Gather节点时，实现用户会话的进程将会请求和规划器选中的工作者数量一样多的后台工作者进程 。规划器将考虑使用的后台工作者的数量被限制为最多max_parallel_workers_per_gather个。任何时候能够存在的后台工作者进程的总数由max_worker_processes和max_parallel_workers限制。因此，一个并行查询可能会使用比规划中少的工作者来运行，甚至有可能根本不使用工作者。最优的计划可能取决于可用的工作者的数量，因此这可能会导致不好的查询性能。如果这种情况经常发生，那么就应当考虑一下提高max_worker_processes和max_parallel_workers的值，这样更多的工作者可以同时运行；或者降低max_parallel_workers_per_gather，这样规划器会要求少一些的工作者。

为一个给定并行查询成功启动的后台工作者进程都将会执行计划的并行部分。这些工作者的领导者也将执行该计划，不过它还有一个额外的任务：它还必须读取所有由工作者产生的元组。当整个计划的并行部分只产生了少量元组时，领导者通常将表现为一个额外的加速查询执行的工作者。反过来，当计划的并行部分产生大量的元组时，领导者将几乎全用来读取由工作者产生的元组并且执行Gather或Gather Merge节点上层计划节点所要求的任何进一步处理。在这些情况下，领导者所作的执行并行部分的工作将会很少。

当计划的并行部分的顶层节点是Gather Merge而不是Gather时，它表示每个执行计划并行部分的进程会产生有序的元组，并且领导者执行一种保持顺序的合并。相反，Gather会以任何方便的顺序从工作者读取元组，这会破坏可能已经存在的排序顺序。

15.2. 何时会用到并行查询？

有几种设置会导致查询规划器在任何情况下都不生成并行查询计划。为了让并行查询计划能够被生成，必须配置好下列设置。

- max_parallel_workers_per_gather必须被设置为大于零的值。这是一种特殊情况，更加普遍的原则是所用的工作者数量不能超过max_parallel_workers_per_gather所配置的数量。
- dynamic_shared_memory_type必须被设置为除none之外的值。并行查询要求动态共享内存以便在合作的进程之间传递数据。

此外，系统一定不能运行在单用户模式下。因为在单用户模式下，整个数据库系统运行在单个进程中，没有后台工作者进程可用。

如果下面的任一条件为真，即便对一个给定查询通常可以产生并行查询计划，规划器都不会为它产生并行查询计划：

- 查询要写任何数据或者锁定任何数据库行。如果一个查询在顶层或者 CTE 中包含了数据修改操作，那么不会为该查询产生并行计划。一种例外是，CREATE TABLE ... AS、SELECT INTO以及CREATE MATERIALIZED VIEW这些创建新表并填充它的命令可以使用并行计划。
- 查询可能在执行过程中被暂停。只要在系统认为可能发生部分或者增量式执行，就不会产生并行计划。例如：用DECLARE CURSOR创建的游标将永远不会使用并行计划。类似地，一个FOR x IN query LOOP .. END LOOP形式的 PL/pgSQL 循环也永远不会使用并行计划，因为当并行查询进行时，并行查询系统无法验证循环中的代码执行起来是安全的。
- 使用了任何被标记为PARALLEL UNSAFE的函数的查询。大多数系统定义的函数都被标记为PARALLEL SAFE，但是用户定义的函数默认被标记为PARALLEL UNSAFE。参见第 15.4 节的讨论。
- 该查询运行在另一个已经存在的并行查询内部。例如，如果一个被并行查询调用的函数自己发出一个 SQL 查询，那么该查询将不会使用并行计划。这是当前实现的一个限制，但是或许不值得移除这个限制，因为它会导致单个查询使用大量的进程。
- 事务隔离级别是可串行化。这是当前实现的一个限制。

即使对于一个特定的查询已经产生了并行查询计划，在一些情况下执行时也不会并行执行该计划。如果发生这种情况，那么领导者将会自己执行该计划在Gather节点之下的部分，就好像Gather节点不存在一样。上述情况将在满足下面的任一条件时发生：

- 因为后台工作者进程的总数不能超过max_worker_processes，导致不能得到后台工作者进程。
- 由于为并行查询目的启动的后台工作者数量不能超过max_parallel_workers这一限制而不能得到后台工作者。
- 客户端发送了一个执行消息，并且消息中要求取元组的数量不为零。执行消息可见扩展查询协议中的讨论。因为libpq当前没有提供方法来发送这种消息，所以这种情况只可能发生在不依赖 libpq 的客户端中。如果这种情况经常发生，那在它可能发生的会话中设置 max_parallel_workers_per_gather为零是一个很好的主意，这样可以避免产生连续运行时次优的查询计划。
- 事务隔离级别是可串行化。这种情况通常不会出现，因为当事务隔离级别是可串行化时不会产生并行查询计划。不过，如果在产生计划之后并且在执行计划之前把事务隔离级别改成可串行化，这种情况就有可能发生。

15.3. 并行计划

因为每个工作者只执行完成计划的并行部分，所以不可能简单地产生一个普通查询计划并使用多个工作者运行它。每个工作者都会产生输出结果集的一个完全拷贝，因而查询并不会比

普通查询运行得更快甚至还会产生不正确的结果。相反，计划的并行部分一定被查询优化器在内部当作一个部分计划，即它必须被构建出来，这样每一个执行该计划的进程将以无重复地方式产生输出行的一个子集，即保证每一个所需要的输出行正好只被一个合作进程生成。通常，这意味着该查询的驱动表上的扫描必须是一种可并行的扫描。

15.3.1. 并行扫描

当前支持下列可并行的表扫描。

- 在一个并行顺序扫描中，表块将在合作进程之间被划分。一次会分发一个块，这样对表的访问还是保持顺序方式。
- 在一个并行位图堆扫描中，一个进程被选为领导者。这个进程执行对一个或者多个索引的扫描并且构建出一个位图指示需要访问哪些表块。这些表块接着会在合作进程之间划分（和并行顺序扫描中一样）。换句话说，堆扫描以并行方式进行但底层的索引扫描不是并行。
- 在一个并行索引扫描或者并行只用索引的扫描中，合作进程轮流从索引读取数据。当前，并行索引扫描仅有B-树索引支持。每一个进程将认领一个索引块并且扫描和返回该索引块引用的所有元组，其他进程可以同时地从一个不同的索引块返回元组。并行B-树扫描的结果会以每个工作者进程内的顺序返回。

其他扫描类型（例如非B-树索引的扫描）可能会在未来支持并行扫描。

15.3.2. 并行连接

正如在非并行计划中那样，驱动表可能被使用嵌套循环、哈希连接或者归并连接连接到一个或者多个其他表。连接的内侧可以是任何类型的被规划器支持的非并行计划，假设它能够安全地在并行工作者中运行。根据连接类型，内侧还可以是一种并行计划。

- 在一个嵌套循环连接中，内侧总是非并行的。尽管它会被完全执行，如果内侧是一个索引扫描也会很高效，因为外侧元组以及在索引中查找值的循环会被划分到多个合作进程。
- 在一个归并连接中，内侧总是一个非并行计划并且因此会被完全执行。这可能是不太高效的，特别是在排序必须被执行时，因为在每一个合作进程中工作数据和结果数据是重复的。
- 在一个哈希连接（没有“并行”前缀）中，每个合作进程都会完全执行内侧以构建哈希表的相同拷贝。如果哈希表很大或者该计划开销很大，这种方式就很低效。在一个并行哈希连接中，内侧是一个并行哈希，它把构建共享哈希表的工作划分到多个合作进程。

15.3.3. 并行聚集

PostgreSQL通过按两个阶段进行聚集来支持并行聚集。首先，每个参与到查询并行部分的进程执行一个聚集步骤，为该进程注意到的每个分组产生一个部分结果。这在计划中反映为一个Partial Aggregate节点。然后，部分结果通过Gather或者Gather Merge被传输到领导者。最后，领导者对来自所有工作者的结果进行重新聚集得到最终的结果。这在计划中反映为一个Finalize Aggregate节点。

因为Finalize Aggregate节点运行在领导者进程上，如果查询产生的分组数相对于其输入行数来说比较大，则查询规划器不会喜欢它。例如，在最坏的情况下，Finalize Aggregate节点看到的分组数可能与所有工作者进程在Partial Aggregate阶段看到的输入行数一样多。对于这类情况，使用并行聚集显然得不到性能收益。查询规划器会在规划过程中考虑这一点并且不太会在这种情况下选择并行聚集。

并行聚集并非在所有情况下都被支持。每一个聚集都必须是对并行安全的并且必须有一个组合函数。如果该聚集有一个类型为internal的转移状态，它必须有序列化和反序列化函数。更多细节请参考CREATE AGGREGATE。如果任何聚集函数调用包含DISTINCT或ORDER BY子句，

则不支持并行聚集。对于有序集聚集或者当查询涉及GROUPING SETS时，也不支持并行聚集。只有在查询中涉及的所有连接也是该计划并行部分的组成部分时，才能使用并行聚集。

15.3.4. 并行Append

只要当PostgreSQL需要从多个源中整合行到一个单一结果集时，它会使用Append或MergeAppend计划节点。在实现UNION ALL或扫描分区表时常常会发生这种情况。就像这些节点可以被用在任何其他计划中一样，它们可以被用在并行计划中。不过，在并行计划中，规划器使用的是Parallel Append节点。

当一个Append节点被用在并行计划中时，每个进程将按照子计划出现的顺序执行子计划，这样所有的参与进程会合作执行第一个子计划直到它被完成，然后同时移动到第二个计划。而在使用Parallel Append时，执行器将把它的子计划尽可能均匀地散布在参与进程中，这样多个子计划会被同时执行。这避免了竞争，也避免了子计划在那些不执行它的进程中产生启动代价。

此外，和常规的Append节点不同（在并行计划中使用时仅有部分子计划），Parallel Append节点既可以有部分子计划也可以有非部分子计划。非部分子计划将仅被单个进程扫描，因为扫描它们不止一次会产生重复的结果。因此涉及到追加多个结果集的计划即使在没有有效的部分计划可用时，也能实现粗粒度的并行。例如，考虑一个针对分区表的查询，它只能通过使用一个不支持并行扫描的索引来实现。规划器可能会选择常规Index Scan计划的Parallel Append。每个索引扫描必须被单一的进程执行完，但不同的扫描可以由不同的进程同时执行。

enable_parallel_append可以被用来禁用这种特性。

15.3.5. 并行计划小贴士

如果我们想要一个查询能产生并行计划但事实上又没有产生，可以尝试减小parallel_setup_cost或者parallel_tuple_cost。当然，这个计划可能比规划器优先产生的顺序计划还要慢，但也不总是如此。如果将这些设置为很小的值（例如把它们设置为零）也不能得到并行计划，那就可能是有某种原因导致查询规划器无法为你的查询产生并行计划。可能的原因可见第 15.2 和第 15.4 节

在执行一个并行计划时，可以用EXPLAIN (ANALYZE,VERBOSE)来显示每个计划节点在每个工作者上的统计信息。这些信息有助于确定是否所有的工作被均匀地分发到所有计划节点以及从总体上理解计划的性能特点。

15.4. 并行安全性

规划器把查询中涉及的操作分类成并行安全、并行受限或者并行不安全。并行安全的操作不会与并行查询的使用产生冲突。并行受限的操作不能在并行工作者中执行，但是能够在并行查询的领导者中执行。因此，并行受限的操作不能出现在Gather或者Gather Merge节点之下，但是能够出现在包含这类节点的计划的其他位置。并行不安全的操作不能在并行查询中执行，甚至不能在领导者中执行。当一个查询包含任何并行不安全操作时，并行查询对这个查询是完全被禁用的。

下面的操作总是并行受限的。

- 公共表表达式（CTE）的扫描。
- 临时表的扫描。
- 外部表的扫描，除非外部数据包装器有一个IsForeignScanParallelSafe API。
- InitPlan所挂接到的计划节点。
- 引用一个相关的SubPlan的计划节点。

15.4.1. 为函数和聚集加并行标签

规划器无法自动判定一个用户定义的函数或者聚集是并行安全、并行受限还是并行不安全，因为这需要预测函数可能执行的每一个操作。一般而言，这就相当于一个停机问题，因此是不可能的。甚至对于可以做到判定的简单函数我们也不会尝试，因为那会非常昂贵而且容易出错。相反，除非是被标记出来，所有用户定义的函数都被认为是并行不安全的。在使用CREATE FUNCTION或者ALTER FUNCTION时，可以通过指定PARALLEL SAFE、PARALLEL RESTRICTED或者PARALLEL UNSAFE来设置标记 。在使用CREATE AGGREGATE时，PARALLEL选项可以被指定为SAFE、RESTRICTED或者 UNSAFE。

如果函数和聚集会写数据库、访问序列、改变事务状态（即便是临时改变，例如建立一个EXCEPTION块来捕捉错误的 PL/pgsql）或者对设置做持久化的更改，它们一定要被标记为PARALLEL UNSAFE。类似地，如果函数会访问临时表、客户端连接状态、游标、预备语句或者系统无法在工作者之间同步的后端本地状态，它们必须被标记为PARALLEL RESTRICTED。例如，setseed和 random由于后一种原因而是并行受限的。

一般而言，如果一个函数是受限或者不安全的却被标记为安全，或者它实际是不安全的却被标记为受限，把它用在并行查询中时可能会抛出错误或者产生错误的回答。如果 C 语言函数被错误标记，理论上它会展现出完全不明确的行为，因为系统中无法保护自身不受任意 C 代码的影响。但是，在最有可能的情况下，结果不会比其他任何函数更糟糕。如果有疑虑，最好还是标记函数为UNSAFE。

如果在并行工作者中执行的函数要求领导者没有持有的锁，例如读该查询中没有引用的表，那么工作者退出时会释放那些锁（而不是在事务结束时释放）。如果你写了一个这样做的函数并且这种不同的行为对你很重要，把这类函数标记为PARALLEL RESTRICTED以确保它们只在领导者中执行。

注意查询规划器不会为了获取一个更好的计划而考虑延迟计算并行受限的函数或者聚集。所以，如果一个被应用到特定表的WHERE子句是并行受限的，查询规划器就不会考虑对处于计划并行部分的表执行一次扫描。在一些情况中，可以（甚至效率更高）把对表的扫描包括在查询的并行部分并且延迟对WHERE子句的计算，这样它会出现在Gather节点之上。不过，规划器不会这样做。

部分 III. 服务器管理

这部份覆盖了PostgreSQL数据库管理员感兴趣的主题。包括软件安装、搭建和配置一个服务器、管理用户和数据库以及维护任务。任何想要运行一个PostgreSQL服务器的人（即使是用于个人用途而不是生产环境），应该熟悉这一部分覆盖的主题。

这部份的信息大致上按照一个新用户会阅读的顺序来安排。但是章节都是自组织的并且可以根据需要独立阅读。这一部分的信息也是按照叙事风格组织的。需要一个特定命令的完整描述的读者应该看看第 VI 部分

最开始的几章主要是为了让读者理解时不需要先导知识，这样需要搭建自己的服务器的新用户可以从这部分开始他们的探索。这一部分剩下的内容是有关调优和管理，这些材料假定读者已经熟悉PostgreSQL数据库系统的一般使用。我们鼓励读者阅读第 I 部分和第 II 部分以获得额外信息。

目录

第 16 章 从源代码安装

本章的内容描述从源代码发布安装PostgreSQL（如果你安装的是打包好的版本如RPM或Debian包，那么请略过这一章并且阅读打包者的指导）。

16.1. 简单版

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

本章剩余部分都是完全版。

16.2. 要求

一般说来，一个现代的与 Unix 兼容的平台应该就能运行PostgreSQL。 到发布为止已经明确测试过的平台的列表在 第 16.6 节列出。在发布的doc子目录里面有许多平台相关的 FAQ 文档，如果你碰到问题你可能会需要参考它们。

编译PostgreSQL需要下列软件包：

- 要求GNU make版本3.80或以上；其他的make程序或更老的GNU make版本将不会工作（GNU make有时以名字gmake安装）。要测试GNU make可以输入：

  ```
  make --version
  ```

- 你需要一个ISO/ANSI C 编译器（至少是 C89兼容的）。我们推荐使用最近版本的GCC，不过，众所周知的是PostgreSQL可以利用许多不同厂商的不同编译器进行编译。

- 除了gzip和bzip2之外，我们还需要tar来解包源代码发布。

- 默认时将自动使用GNU Readline库。它允许psql（PostgreSQL的命令行 SQL 解释器）记住你输入的每一个命令并且允许你使用箭头键来找回和编辑之前的命令。如果你不想用它，那么你必需给configure声明--without-readline选项。作为一种可选方案，你常常可以使用 BSD许可证的libedit库，它最初是在NetBSD上开发的。libedit库是GNU Readline兼容的， 如果没有发现libreadline或者configure使用了--with-libedit-preferred选项，都会使用这个库。如果你使用的是一个基于包的 Linux 发布，那么要注意你需要readline和readline-devel两个包，特别是如果这两个包在你的版本里是分开的时候。

- 默认的时候将使用zlib压缩库。 如果你不想使用它，那么你必须给configure声明--without-zlib选项。使用这个选项关闭了在pg_dump和pg_restore中对压缩归档的支持。

下列包是可选的。在默认配置的时候并不要求它们，但是如果打开了一些编译选项之后就需要它们了，如下文所解释的：

- 要编译服务器端编程语言PL/Perl，你需要一个完整的 Perl安装，包括libperl 库和头文件。 所需的最低版本是Perl 5.8.3。 因为PL/Perl是一个共享库， libperl库在大多数

平台上也必须是一个共享库。最近的版本的 Perl好像已经默认这样做了，但是早先的版本可不是 这样的，而且这总是一种在站点上安装 Perl 的选择。如果选择了编译 PL/Perl 但是它却找不到一个共享的 libperl，那么configure将会失败。 在这种情况下，你将不得不重新手工编译并且安装Perl 以便能编译PL/Perl。在 Perl的配置处理过程中，需要一个共享库。

如果你想更多地使用PL/Perl， 你应当保证Perl安装在编译时启用了 usemultiplicity选项（perl -V将会显示是否是这样）。

- 要编译PL/Python服务器端编程语言， 你需要一个Python的安装，包括头文件和distutils模块。最低的版本要求是Python 2.4。如果是版本3.1或更高版本，则支持Python 3， 如果使用 Python 3 请参考第 46.1 节

  因为PL/Python将以共享库的方式编译， libpython库在大多数平台上也必须是一个共享库。 在默认的从源码安装Python时不是这样的， 而是在很多操作系统发布中有一个共享库可用。如果选择了编译 PL/Python但找不到一个共享的 libpython，configure将 会失败。这可能意味着你不得不安装额外的包或者（部分）重编译 Python安装以提供这个共享库。 在从源码编译时，请用--enable-shared标志运行 Python的配置脚本。

- 如果你想编译PL/Tcl过程语言， 你当然需要安装Tcl，要求的最低版本是 Tcl 8.4。

- 要打开本地语言支持（NLS），也就是说， 用英语之外的语言显示程序的消息，你需要一个Gettext API的实现。有些操作系统内置了这些（例如Linux、NetBSD、Solaris）， 对于其它系统，你可以从http://www.gnu.org/software/gettext/下载一个额外的包。如果你在使用GNU C 库里面的Gettext实现， 那么你就额外需要GNU Gettext包，因为我们需要里面的几个工具程序。对于任何其它的实现，你应该不需要它。

- 如果您想支持加密的客户端连接，则需要OpenSSL。最低要求的版本是0.9.8。

- 如果你想支持使用Kerberos、OpenLDAP和/或PAM服务的认证，那你需要相应的包。

- 要编译PostgreSQL文档，有一些独立的要求集，请见 第 J.2 节

如果你正从Git树而不是使用发布的源代码包进行编译，或者你想做服务器端开发， 那么你还需要下面的包：

- 如果你需要从 Git 检出中编译，或者你修改了实际的扫描器和分析器的定义文件，那么你需要 GNU Flex和Bison。 如果你需要它们，那么确保自己拿到的是Flex 2.5.31 或更新的版本， 以及Bison 1.875 或者更新的版本。不能使用其他lex和yacc程序。

- 如果需要从 Git 检出中编译，或者你修改了任何使用 Perl 脚本的编译步骤的输入文件，那么你需要Perl 5.8.3或以后的版本。如果你在 Windows 上编译，你在任何情况下都需要 Perl。运行一些测试套件时也需要Perl。

如果你需要获取GNU包，你可以在你的本地GNU镜像站点 （看看 https://www.gnu.org/prep/ftp或ftp://ftp.gnu.org/gnu/找到它们。

还要检查一下你是否有足够的磁盘空间。你将大概需要近 100MB 用于存放编译过程中的源码树和大约 20 MB 用于安装目录。 一个空数据库集簇大概需要 35 MB。一个数据库所占的空间大约是存储同样数据的平面文件所占空间的五倍。如果你要运行回归测试，还临时需要额外的 150MB。请用df命令检查剩余磁盘空间。

16.3. 获取源码

PostgreSQL 11.2 源代码可以从我们的官方网站 https://www.postgresql.org/download/的下载区中获得。你将得到一个名为postgresql-11.2.tar.gz或postgresql-11.2.tar.bz2的文件。在你获取文件之后，解压缩它：

```
gunzip postgresql-11.2.tar.gz
tar xf postgresql-11.2.tar
```

（如果你得到的是.bz2文件，请用bunzip2代替gunzip）。这样将在当前目录创建一个目录postgresql-11.2， 里面是PostgreSQL源代码。 进入这个目录完成安装过程的其他步骤。

你也可以直接从版本控制库中获得源代码，参见附录 I

16.4. 安装过程

1. 配置

   安装过程的第一步就是为你的系统配置源代码树并选择你喜欢的选项。这个工作是通过运行configure脚本实现的，对于默认安装，你只需要简单地输入：

   ```
   ./configure
   ```

   该脚本将运行一些测试来决定一些系统相关的变量， 并检测你的操作系统的特殊设置，并且最后将在编译树中创建一些文件以记录它找到了什么。如果你想保持编译目录的独立，你也可以在一个源码树之外的目录中运行configure 。这个过程也被称为一个VPATH编译。做法如下：

   ```
   mkdir build_dir
   cd build_dir
   /path/to/source/tree/configure [options go here]
   make
   ```

   默认设置将编译服务器和辅助程序，还有只需要 C 编译器的所有客户端程序和接口。默认时所有文件都将安装到/usr/local/pgsql。

   你可以通过给出下面的configure命令行选项中的一个或更多的选项来自定义编译和安装过程：

   --prefix=PREFIX

   把所有文件装在目录PREFIX中而不是/usr/local/pgsql中。 实际的文件会安装到数个子目录中；没有一个文件会直接安装到PREFIX目录里。

   如果你有特殊需要，你还可以用下面的选项自定义不同的子目录的位置。 不过，如果你把这些设置保留默认，那么安装将是可重定位的，意思是你可以在安装过后移动目录（man和doc位置不受此影响）。

   对于可重定位的安装，你可能需要使用configure的--disable-rpath选项。 还有，你需要告诉操作系统如何找到共享库。

   --exec-prefix=EXEC-PREFIX

   你可以把体系相关的文件安装到一个不同的前缀下（EXEC-PREFIX），而不是PREFIX中设置的地方。 这样做可以比较方便地在不同主机之间共享体系相关的文件。 如果你省略这些，那么EXEC-PREFIX就会被设置为等于 PREFIX并且体系相关和体系无关的文件都会安装到同一棵目录树下，这也可能是你想要的。

   --bindir=DIRECTORY

   为可执行程序指定目录。默认是EXEC-PREFIX/bin， 通常也就是/usr/local/pgsql/bin。

   --sysconfdir=DIRECTORY

   用于各种各样配置文件的目录，默认为PREFIX/etc。

--libdir=DIRECTORY

设置安装库和动态装载模块的目录。默认是EXEC-PREFIX/lib。

--includedir=DIRECTORY

C 和 C++ 头文件的目录。默认是PREFIX/include。

--datarootdir=DIRECTORY

设置多种只读数据文件的根目录。这只为后面的某些选项设置默认值。默认值为PREFIX/share。

--datadir=DIRECTORY

设置被安装的程序使用的只读数据文件的目录。默认值为DATAROOTDIR。注意这不会对你的数据库文件被放置的位置产生任何影响。

--localedir=DIRECTORY

设置安装区域数据的目录，特别是消息翻译目录文件。默认值为DATAROOTDIR/locale。

--mandir=DIRECTORY

PostgreSQL自带的手册页将安装到这个目录，它们被安装在相应的manx子目录里。默认是DATAROOTDIR/man。

--docdir=DIRECTORY

设置安装文档文件的根目录，“man”页不包含在内。这只为后续选项设置默认值。这个选项的默认值为DATAROOTDIR/doc/postgresql。

--htmldir=DIRECTORY

PostgreSQL的HTML格式的文档将被安装在这个目录中。默认值为DATAROOTDIR。

> **注意**
>
> 为了让PostgreSQL能够安装在一些共享的安装位置（例如/usr/local/include）， 同时又不至于和系统其它部分产生名字空间干扰，我们特别做了一些处理。 首先，安装脚本会自动给datadir、sysconfdir和docdir后面附加上“/postgresql”字符串，除非展开的完整路径名已经包含字符串“postgres”或者“pgsql”。 例如，如果你选择/usr/local作为前缀， 那么文档将安装在/usr/local/doc/postgresql，但如果前缀是/opt/postgres， 那么它将被放到/opt/postgres/doc。客户接口的公共 C 头文件安装到了includedir，并且是名字空间无关的。内部的头文件和服务器头文件都安装在includedir下的私有目录中。参考每种接口的文档获取关于如何访问头文件的信息。最后，如果合适，那么也会在libdir下创建一个私有的子目录用于动态可装载的模块。

--with-extra-version=STRING

把STRING追加到 PostgreSQL 版本号。例如，你可以使用它来标记从未发布的 Git 快照或者包含定制补丁（带有一个如git describe标识符之类的额外版本号或者一个分发包发行号）创建的二进制文件。

--with-includes=DIRECTORIES

DIRECTORIES是一个冒号分隔的目录列表，这些目录将被加入编译器的头文件搜索列表中。 如果你有一些可选的包（例如 GNU Readline）安装在非标准位置， 你就必须使用这个选项，以及可能还有相应的 --with-libraries选项。

例子：--with-includes=/opt/gnu/include:/usr/sup/include.

--with-libraries=DIRECTORIES

DIRECTORIES是一个冒号分隔的目录列表，这些目录是用于查找库文件的。 如果你有一些包安装在非标准位置，你可能就需要使用这个选项（以及对应的--with-includes选项）。

例子：--with-libraries=/opt/gnu/lib:/usr/sup/lib.

--enable-nls[=LANGUAGES]

打开本地语言支持（NLS），也就是以非英文显示程序消息的能力。LANGUAGES是一个空格分隔的语言代码列表， 表示你想支持的语言。例如--enable-nls='de fr' （你提供的列表和实际支持的列表之间的交集将会自动计算出来）。如果你没有声明一个列表，那么就会安装所有可用的翻译。

要使用这个选项，你需要一个Gettext API 的实现。见上文。

--with-pgport=NUMBER

把NUMBER设置为服务器和客户端的默认端口。默认是 5432。 这个端口可以在以后修改，不过如果你在这里声明，那么服务器和客户端将有相同的编译好了的默认值。这样会非常方便些。 通常选取一个非默认值的理由是你企图在同一台机器上运行多个PostgreSQL服务器。

--with-perl

制作PL/Perl服务器端编程语言。

--with-python

制作PL/Python服务器端编程语言。

--with-tcl

制作PL/Tcl服务器编程语言。

--with-tclconfig=DIRECTORY

Tcl 安装文件tclConfig.sh，其中里面包含编译与 Tcl 接口的模块的配置信息。该文件通常可以自动地在一个众所周知的位置找到，但是如果你需要一个不同版本的Tcl，你也可以指定可以找到它的目录。

--with-gssapi

编译 GSSAPI 认证支持。在很多系统上，GSSAPI（通常是 Kerberos 安装的一部分）系统不会被安装在默认搜索位置（例如/usr/include、/usr/lib），因此你必须使用选项--with-includes和--with-libraries来配合该选项。configure将会检查所需的头文件和库以确保你的 GSSAPI 安装足以让配置继续下去。

--with-krb-srvnam=NAME

默认的 Kerberos 服务主的名称（也被 GSSAPI 使用）。默认是postgres。通常没有理由改变这个值，除非你是一个 Windows 环境，这种情况下该名称必须被设置为大写形式POSTGRES。

--with-llvm

支持基于LLVM的JIT编译 （请参阅第 32 章。 这需要安装LLVM库。 当前LLVM的最低要求版本是3.9。

llvm-config 可用于查找所需的编译选项。llvm-config会在PATH 上搜索所有受支持版本的llvm-config-$major-$minor。 如果那还不能找到正确的二进制文件，请使用LLVM_CONFIG指定正确的 llvm-config的路径。例如：

```
./configure ... --with-llvm LLVM_CONFIG='/path/to/llvm/bin/llvm-config'
```

LLVM支持需要兼容的clang编译器 （必要时使用CLANG环境变量指定）和有效的C++编译器 （必要时使用使用CXX环境变量指定）。

--with-icu

支持ICU库。 这需要安装ICU4C软件包。 目前要求的最低ICU4C版本是4.2。

默认的，pkg-config 将被用来查找所需的编译选项。支持ICU4C版本4.6及更高版本。 对于较老版本，或者如果pkg-config不可用， 可以将变量ICU_CFLAGS和ICU_LIBS 指定为configure，就像下面的示例中那样：

```
./configure ... --with-icu ICU_CFLAGS='-I/some/where/include' ICU_LIBS='-L/some/where/lib -licui18n -licuuc -licudata'
```

(如果ICU4C在编译器的默认搜索路径中， 那么你仍然需要指定一个非空的字符串，以避免使用pkg-config， 例如ICU_CFLAGS=' '。)

--with-openssl

编译SSL（加密）连接支持。这个选项需要安装OpenSSL包。configure将会检查所需的头文件和库以确保你的 OpenSSL安装足以让配置继续下去。

--with-pam

编译PAM（可插拔认证模块）支持。

--with-bsd-auth

编译 BSD 认证支持（BSD 认证框架目前只在 OpenBSD 上可用）。

--with-ldap

为认证和连接参数查找编译LDAP支持 （详见第 34.17 和第 20.10 节。在 Unix 上，这需要安装OpenLDAP包。在 Windows 上将使用默认的WinLDAP库。configure将会检查所需的头文件和库以确保你的 OpenLDAP安装足以让配置继续下去。

--with-systemd

编译对systemd 服务通知的支持。如果服务器是在systemd 机制下被启动，这可以提高集成度，否则不会有影响 否则，; 参考 第 18.3 节查看更多信息 。要使用这个选项，必须安装libsystemd 以及相关的头文件。

--without-readline

避免使用Readline库（以及libedit）。这个选项禁用了psql中的命令行编辑和历史， 因此我们不建议这么做。

--with-libedit-preferred

更倾向于使用BSD许可证的libedit库而不是GPL许可证的Readline。这个选项只有在你同时安装了两个库时才有意义，在那种情况下默认会使用Readline。

--with-bonjour

编译 Bonjour 支持。这要求你的操作系统支持 Bonjour。在 macOS 上建议使用。

--with-uuid=LIBRARY

使用指定的 UUID 库编译uuid-ossp模块（提供生成 UUID 的函数）。 LIBRARY必须是下列之一：

- bsd，用来使用 FreeBSD、NetBSD 和一些其他 BSD 衍生系统 中的 UUID 函数
- e2fs，用来使用e2fsprogs项目创建的 UUID 库， 这个库出现在大部分的 Linux 系统和 macOS 中，并且也能找到用于其他平台的 版本
- ossp，用来使用OSSP UUID library[1]

--with-ossp-uuid

--with-uuid=ossp的已废弃的等效选项。

--with-libxml

编译 libxml （启用 SQL/XML 支持）。这个特性需要 Libxml 版本 2.6.23 及以上。

Libxml 会安装一个程序xml2-config，它可以被用来检测所需的编译器和链接器选项。如果能找到，PostgreSQL 将自动使用它。要制定一个非常用的 libxml 安装位置，你可以设置环境变量XML2_CONFIG指向xml2-config程序所属的安装，或者使用选项--with-includes和--with-libraries。

--with-libxslt

编译xml2模块时使用 libxslt。xml2依赖这个库来执行XML的XSL转换。

--disable-float4-byval

禁用 float4 值的“传值”，导致它们只能被“传引用”。这个选项会损失性能，但是在需要兼容使用 C 编写并使用“version 0”调用规范的老用户定义函数时可能需要这个选项。更好的长久解决方案是将任何这样的函数更新成使用“version 1”调用规范。

--disable-float8-byval

禁用 float8 值的“传值”，导致它们只能被“传引用”。这个选项会损失性能，但是在需要兼容使用 C 编写并使用“version 0”调用规范的老用户定义函数时可能需要这个选项。更好的长久解决方案是将任何这样的函数更新成使用“version 1”调用规范。注意这个选项并非只影响 float8，它还影响 int8 和某些相关类型如时间戳。在32位平台上，--disable-float8-byval是默认选项并且不允许选择--enable-float8-byval。

[1] http://www.ossp.org/pkg/lib/uuid/

--with-segsize=SEGSIZE

设置段尺寸，以 G 字节计。大型的表会被分解成多个操作系统文件，每一个的尺寸等于段尺寸。这避免了与操作系统对文件大小限制相关的问题。默认的段尺寸（1G字节）在所有支持的平台上都是安全的。如果你的操作系统有“largefile”支持（如今大部分都支持），你可以使用一个更大的段尺寸。这可以有助于在使用非常大的表时消耗的文件描述符数目。但是要当心不能选择一个超过你将使用的平台和文件系统所支持尺寸的值。你可能希望使用的其他工具（如tar）也可以对可用文件尺寸设限。如非绝对必要，我们推荐这个值应为2的幂。注意改变这个值需要一次 initdb。

--with-blocksize=BLOCKSIZE

设置块尺寸，以 K 字节计。这是表内存储和I/O的单位。默认值（8K字节）适合于大多数情况，但是在特殊情况下可能其他值更有用。这个值必须是2的幂并且在 1 和 32 （K字节）之间。注意修改这个值需要一次 initdb。

--with-wal-segsize=SEGSIZE

设置WAL 段尺寸，以 M 字节计。这是 WAL 日志中每一个独立文件的尺寸。调整这个值来控制传送 WAL 日志的粒度非常有用。默认尺寸为 16 M字节。这个值必须是2的幂并且在 1 到 1024 （M字节）之间。注意修改这个值需要一次 initdb。

--with-wal-blocksize=BLOCKSIZE

设置WAL 块尺寸，以 K 字节计。这是 WAL 日志存储和I/O的单位。默认值（8K 字节）适合于大多数情况，但是在特殊情况下其他值更好有用。这个值必须是2的幂并且在 1 到 64（K字节）之间。注意修改这个值需要一次 initdb。

--disable-spinlocks

即便PostgreSQL对于该平台没有 CPU 自旋锁支持，也允许编译成功。自旋锁支持的缺乏会导致较差的性能，因此这个选项只有当编译终端或者通知你该平台缺乏自旋锁支持时才应被使用。如果在你的平台上要求使用该选项来编译PostgreSQL，请将此问题报告给PostgreSQL的开发者。

--disable-strong-random

即使PostgreSQL不支持平台上的强大随机数，也允许构建成功。 一些认证协议以及pgcrypto 模块中的一些例程需要随机数的来源。--disable-strong-random 禁用需要密码强的随机数的功能，并用弱伪随机数生成器代替验证盐值生成和查询取消密钥。 它可能会使认证安全性降低。

--disable-thread-safety

禁用客户端库的线程安全性。这会阻止libpq和ECPG程序中的并发线程安全地控制它们私有的连接句柄。

--with-system-tzdata=DIRECTORY

PostgreSQL包含它自己的时区数据库，它被用于日期和时间操作。这个时区数据库实际上是和 IANA 时区数据库相兼容的，后者在很多操作系统如 FreeBSD、Linux和Solaris上都有提供，因此再次安装它可能是冗余的。当这个选项被使用时，将不会使用DIRECTORY中系统提供的时区数据库，而是使用包括在 PostgreSQL 源码发布中的时区数据库。DIRECTORY必须被指定为一个绝对路径。/usr/share/zoneinfo在某些操作系统上是一个很有可能的路径。注意安装例程将不会检测不匹配或错误的时区数据。如果你使用这个选项，建议你运行回归测试来验证你指定的时区数据能正常地工作在PostgreSQL中。

这个选项主要针对那些很了解他们的目标操作系统的二进制包发布者。使用这个选项主要优点是不管何时当众多本地夏令时规则之一改变时， PostgreSQL 包不需要被升级。另一个优点是如果时区数据库文件在安装时不需要被编译， PostgreSQL 可以被更直接地交叉编译。

--without-zlib

避免使用Zlib库。这样就禁用了pg_dump和 pg_restore中对压缩归档的支持。这个选项只适用于那些没有这个库的少见的系统。

--enable-debug

把所有程序和库以带有调试符号的方式编译。这意味着你可以通过一个调试器运行程序来分析问题。 这样做显著增大了最后安装的可执行文件的大小，并且在非 GCC 的编译器上，这么做通常还要关闭编译器优化， 这些都导致速度的下降。但是，如果有这些符号的话，就可以非常有效地帮助定位可能发生问题的位置。目前，我们只是在你使用 GCC 的情况下才建议在生产安装中使用这个选项。但是如果你正在进行开发工作，或者正在使用 beta 版本，那么你就应该总是打开它。

--enable-coverage

如果在使用 GCC，所有程序和库都会用代码覆盖率测试工具编译。在运行时，它们会在编译目录中生成代码覆盖率度量的文件。详见第 33.5 节这个选项只用于 GCC 以及做开发工作时。

--enable-profiling

如果在使用 GCC，所有程序和库都被编译成可以进行性能分析。在后端退出时，将会创建一个子目录，其中包含用于性能分析的gmon.out文件。这个选项只用于 GCC 和做开发工作时。

--enable-cassert

打开在服务器中的assertion检查， 它会检查许多“不可能发生”的条件。它对于代码开发的用途而言是无价之宝， 不过这些测试可能会显著地降低服务器的速度。并且，打开这个测试不会提高你的系统的稳定性！ 这些断言检查并不是按照严重性分类的，因此一些相对无害的小故障也可能导致服务器重启 — 只要它触发了一次断言失败。 目前，我们不推荐在生产环境中使用这个选项，但是如果你在做开发或者在使用 beta 版本的时候应该打开它。

--enable-depend

打开自动倚赖性跟踪。如果打开这个选项，那么制作文件（makefile）将设置为在任何头文件被修改的时候都将重新编译所有受影响的目标文件。 如果你在做开发的工作，那么这个选项很有用，但是如果你只是想编译一次并且安装，那么这就是浪费时间。 目前，这个选项只对 GCC 有用。

--enable-dtrace

为PostgreSQL编译对动态跟踪工具 DTrace 的支持。 详见第 28.5 节

要指向dtrace程序，必须设置环境变量DTRACE。这通常是必需的，因为dtrace通常被安装在/usr/sbin中，该路径可能不在搜索路径中。

dtrace程序的附加命令行选项可以在环境变量DTRACEFLAGS中指定。在 Solaris 上，要在一个64位二进制中包括 DTrace，你必须为 configure 指定DTRACEFLAGS="-64"。例如，使用 GCC 编译器：

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

使用 Sun 的编译器：

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace
 DTRACEFLAGS='-64' ...
```

--enable-tap-tests

启用 Perl TAP 工具进行测试。这要求安装了 Perl 以及 Perl 模块IPC::Run。 详见 第 33.4 节

如果你喜欢用那些和configure选取的不同的 C 编译器，那么你可以你的环境变量CC设置为你选择的程序。默认时，只要gcc可以使用，configure将选择它， 或者是该平台的默认（通常是cc）。类似地，你可以用CFLAGS变量覆盖默认编译器标志。

你可以在configure命令行上指定环境变量， 例如：

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

下面是可以以这种方式设置的有效变量的列表：

BISON

Bison程序

CC

C编译器

CFLAGS

传递给 C 编译器的选项

CLANG

clang程序的路径，用于处理使用-with-llvm 进行编译时内联的源代码。

CPP

C 预处理器

CPPFLAGS

传递给 C 预处理器的选项

CXX

C++编译器

CXXFLAGS

传给C++编译器的选项

DTRACE

dtrace程序的位置

DTRACEFLAGS

传递给dtrace程序的选项

FLEX

Flex程序

LDFLAGS

链接可执行程序或共享库时使用的选项

LDFLAGS_EX

只用于链接可执行程序的附加选项

LDFLAGS_SL

只用于链接共享库的附加选项

LLVM_CONFIG

llvm-config程序用于查找 LLVM安装。

MSGFMT

用于本地语言支持的msgfmt程序

PERL

Perl 解释器的全路径名称。这将被用来决定编译 PL/Perl 时的依赖性。

PYTHON

Python 解释器程序。这将被用来决定编译 PL/Python 时的依赖性。另外这里指定的是 Python 2 还是 Python 3 （或者是隐式选择）决定了 PL/Python 语言的哪一种变种将成为可用的。 详见第 46.1 节 如果未设置，则按以下顺序探测：python python3 python2。

TCLSH

Tcl 解释器的程序。这将被用来决定编译 PL/Tcl 时的依赖性，并且它将被替换到 Tcl 脚本中。

XML2_CONFIG

用于定位 libxml 安装的xml2-config程序。

有时候，将编译器标志事后添加到由configure选择的集合中非常有用。 一个重要的例子是，gcc的-Werror 选项不能包含在传递给configure的CFLAGS中， 因为它会中断许多configure的内置测试。要添加这样的标志， 在运行make时将它们包含在COPT环境变量中。 将COPT的内容添加到由configure设置的 CFLAGS和LDFLAGS中。例如，你可以这样做

```
make COPT='-Werror'
```

或者

```
export COPT='-Werror'
make
```

> **注意**
>
> 在开发服务器内部代码时，我们推荐使用配置选项--enable-cassert（它会打开很多运行时错误检查）和--enable-debug（它会提高调试工具的有用性）。
>
> 如果在使用 GCC，最好使用至少-O1的优化级别来编译，因为不使用优化（-O0）会禁用某些重要的编译器警告（例如使用未经初始化的变量）。但是，非零的优化级别会使调试更复杂，因为在编译好的代码中步进通常将不能和源代码行一一对应。如果你在尝试调试优化过的代码时觉得困惑，将感兴趣的特定文件使用-O0编译。一种简单的方式是传递一个选项给make：make PROFILE=-O0 file.o。
>
> COPT和PROFILE环境变量同样由PostgreSQL makefile实际处理。要使用哪个是一个性能问题，但是开发者的共同习惯是将 PROFILE用于一次性的标识调整，而始终保持设置COPT。

2. 编译

   要开始编译，键入：

   ```
   make
   ```

   （一定要记得用GNU make）。依你的硬件而异，编译过程可能需要 5 分钟到半小时。显示的最后一行应该是：

   ```
   All of PostgreSQL successfully made. Ready to install.
   ```

   如果你希望编译所有能编译的东西，包括文档（HTML和手册页）以及附加模块（contrib），这样键入：

   ```
   make world
   ```

   显示的最后一行应该是：

   ```
   PostgreSQL, contrib, and documentation successfully made. Ready to install.
   ```

   如果要调用另一个makefile而不是手动构建，则必须取消设置 MAKELEVEL或将其设置为零，例如这样:

   ```
   build-postgresql:
           $(MAKE) -C postgresql MAKELEVEL=0 all
   ```

   否则可能会导致奇怪的错误消息，通常是有关缺少头文件的消息。

3. 回归测试

   如果你想在安装文件前测试新编译的服务器， 那么你可以在这个时候运行回归测试。 回归测试是一个用于验证PostgreSQL在你的系统上是否按照开发人员设想的那样运行的测试套件。键入：

   ```
   make check
   ```

（这条命令不能以 root 运行；请在非特权用户下运行该命令）。（This won't work as root; do it as an unprivileged user.）详细参考第 33 章关于如何解释测试结果的信息。你可以在以后的任何时间通过执行这条命令来运行这个测试。

4. 安装文件

> **注意**
>
> 如果你正在升级一套现有的系统，请阅读第 18.6 节 其中有关于升级一个集簇的指导。

要安装PostgreSQL，输入：

```
make install
```

这条命令将把文件安装到在步骤 1中指定的目录。确保你有足够的权限向该区域写入。通常你需要用 root 权限做这一步。或者你也可以事先创建目标目录并且分派合适的权限。

要安装文档（HTML和手册页），输入：

```
make install-docs
```

如果你按照上面的方法编译了所有东西，输入：

```
make install-world
```

这也会安装文档。

你可以使用make install-strip代替make install， 在安装可执行文件和库文件时把它们剥离。 这样将节约一些空间。如果你编译时带着调试支持，那么抽取将有效地删除调试支持， 因此我们应该只是在不再需要调试的时候做这些事情。 install-strip力图做一些合理的工作来节约空间， 但是它并不了解如何从可执行文件中抽取每个不需要的字节， 因此，如果你希望节约所有可能节约的磁盘空间，那么你可能需要手工做些处理。

标准的安装只提供客户端应用开发和服务器端程序开发所需的所有头文件，例如用 C 写的定制函数或者数据类型（在PostgreSQL 8.0 之前，后者需要独立地执行一次make install-all-headers命令，不过现在这个步骤已经融合到标准的安装步骤中）。

只安装客户端：. 如果你只想装客户应用和接口，那么你可以用下面的命令：

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
make -C doc install
```

src/bin中有一些服务器专用的二进制文件，但是它们很小。

卸载：. 要撤销安装可以使用命令make uninstall。不过这样不会删除任何创建出来的目录。

清理：. 在安装完成以后，你可以通过在源码树里面用命令make clean删除编译文件。 这样会保留configure程序生成的文件，这样以后你就可以用make命令重

新编译所有东西。 要把源码树恢复为发布时的状态，可用make distclean命令。 如果你想从同一棵源码树上为多个不同平台制作，你就一定要运行这条命令并且为每个编译重新配置（另外一种方法是在每种平台上使用一套独立的编译树，这样源代码树就可以保留不被更改）。

如果你执行了一次制作，然后发现你的configure选项是错误的， 或者你修改了任何configure所探测的东西（例如，升级了软件）， 那么在重新配置和编译之前运行一下make distclean是个好习惯。如果不这样做， 你修改的配置选项可能无法传播到所有需要变化的地方。

16.5. 安装后设置

16.5.1. 共享库

在一些有共享库的系统里，你需要告诉你的系统如何找到新安装的共享库。那些并不是必须做这个工作的系统包括 FreeBSD、HP-UX、Linux、NetBSD、OpenBSD和Solaris。

设置共享库的搜索路径的方法因平台而异， 但是最广泛使用的方法是设置环境变量LD_LIBRARY_PATH，例如在 Bourne shells （sh、ksh、bash、zsh）中：

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

或者在csh或tcsh中：

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

把/usr/local/pgsql/lib换成你在步骤 1时设置的--libdir。 你应该把这些命令放到 shell启动文件，如/etc/profile或~/.bash_profile中。 和这个方法相关的一些注意事项和很好的信息可以在http://xahlee.info/UnixResource_dir/_/ldpath.html找到。

在有些系统上，更好的方法可能是在编译之前设置环境变量LD_RUN_PATH。

在Cygwin上，把库目录放在PATH中或者把.dll文件移动到bin目录。

如果有疑问，请参考你的系统的手册页（可能是ld.so或rld）。 如果稍后你收到下面这样的消息：

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

那么这一步就是必须的了。这个只需关注一下就是了。

如果你用的系统是Linux，并且你还有 root 权限，那么你可以在安装之后运行：

```
/sbin/ldconfig /usr/local/pgsql/lib
```

（或者等效的目录）以便让运行时链接器更快地找到共享库。请参考ldconfig的手册页获取更多信息。在FreeBSD、NetBSD和OpenBSD上，命令是：

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

我们不知道其它的系统有等效的命令。

16.5.2. 环境变量

如果你安装到/usr/local/pgsql或者其他默认不在搜索路径中的地方，那你应该在你的PATH环境变量里面增加一个/usr/local/pgsql/bin（或者是你在步骤1时给选项--bindir设置的任何值）。严格来说，这些都不是必须的，但这么做可以让你使用PostgreSQL更方便。

要做这些事情，把下面几行加到你的 shell 启动文件，如~/.bash_profile（如果想影响所有用户就放在/etc/profile）：

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

如果你用的是csh或者tcsh，那么用这条命令：

```
set path = ( /usr/local/pgsql/bin $path )
```

为了让你的系统找得到man文档，你需要加类似下面的一行到一个shell启动文件里（除非你安装到了默认搜索的位置）：

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

环境变量PGHOST和PGPORT为客户端应用指定了数据库服务器的主机和端口，它们会覆盖编译时的默认项。如果你想从远程运行客户端应用，那么为每个准备使用该数据库的用户都设置PGHOST将会非常方便。但这不是必须的，而且大部分客户端程序也可以通过命令行选项替换这些设置。

16.6. 平台支持

如果代码包含规定要工作在一个平台（即一种 CPU 架构和操作系统的结合）上并且它最近已经被验证能在该平台上编译并通过其回归测试，PostgreSQL开发社区才会认为该平台是被支持的。目前，大部分平台兼容性的测试都是由PostgreSQL编译农场[2]的测试机器自动完成的。如果你对在一个并没有出现在编译农场中的平台上运行PostgreSQL感兴趣，但是代码确实能够工作或者能被修改得工作，我们强烈鼓励你建立一个编译农场成员机器，这样进一步的兼容性可以被确认。

通常，PostgreSQL被期望能在这些CPU架构上工作：x86、x86_64、IA64、PowerPC、PowerPC 64、S/390、S/390x、Sparc、Sparc 64、ARM、MIPS、MIPSEL和PA-RISC。存在对 M68K、M32R 和 VAX 的代码支持，但是这些架构上并没有近期测试的报告。通常也可以在一个为支持的 CPU 类型上通过使用--disable-spinlocks配置来进行编译，但是性能将会比较差。

PostgreSQL被期望能在这些操作系统上工作：Linux（所有最近的发布）、Windows（Win2000 SP4及以上）、 FreeBSD、OpenBSD、NetBSD、macOS、AIX、HP/UX 和 Solaris。其他类 Unix 系统可能也可以工作，但是目前没有被测试。在大部分情况下，一个给定操作系统所支持的所有 CPU 架构都能工作。查找下文的第 16.7 节看是否有与你的操作系统相关的信息，特别是使用一个老的系统时更应该这样做。

如果你在一个平台上有安装问题，并且该平台根据最近的编译农场结果已经可以被支持，请将问题报告给<pgsql-bugs@lists.postgresql.org>。如果你有兴趣将PostgreSQL移植到一个新的平台，<pgsql-hackers@lists.postgresql.org>是一个合适的讨论它的地方。

[2] https://buildfarm.postgresql.org/

16.7. 平台相关的说明

这一节提供了考虑 PostgreSQL 安装和设置的附加平台相关的话题。确保阅读安装指导，特别是第 16.2 节 同样，检查关于回归测试结果解释的第 33 章

这里没有覆盖的平台不存在平台相关的安装问题。

16.7.1. AIX

PostgreSQL 能在 AIX 上工作，但是正确地安装它却富有挑战性。从4.3.3到6.1的 AIX 被认为是可支持的。你可以使用 GCC 或本地 IBM 编译器xlc。通常，使用最新版本的 AIX 和 PostgreSQL 能有所帮助。在编译农场中检查有关已知能工作的 AIX 版本的最新信息。

被支持的 AIX 版本的最小推荐修理级别是：

AIX 4.3.3

Maintenance Level 11 + post ML11 bundle

AIX 5.1

Maintenance Level 9 + post ML9 bundle

AIX 5.2

Technology Level 10 Service Pack 3

AIX 5.3

Technology Level 7

AIX 6.1

Base Level

要检查你当前的修理级别，在AIX 4.3.3 至 AIX 5.2 ML 7中使用 oslevel -r，或者在后面的版本中使用 oslevel -s。

如果你已经在/usr/local中安装了 Readline 或 libz，在你自己的选项之外使用下列configure标志： --with-includes=/usr/local/include --with-libraries=/usr/local/lib.

16.7.1.1. GCC问题

在 AIX 5.3 上，使用 GCC 编译和运行 PostgreSQL 有一些问题。

你将要使用 GCC 继 3.3.2 之后的一个版本，特别是如果你在使用一个打包好的版本。我们在 4.0.1 上获得了成功。早期版本的问题看起来更多地与 IBM 打包的 GCC 有关，而非 GCC 真正的问题，因此如果你自己编译 GCC， 你更有可能使用早期版本的 GCC 取得成功。

16.7.1.2. Unix域套接字崩溃

AIX 5.3 有一个问题是sockaddr_storage定义得不够大。在版本 5.3 中，IBM 增加了sockaddr_un（Unix域套接字的地址结构）的尺寸，但是没有相应地增加sockaddr_storage的尺寸。这样做的结果是在 PostgreSQL 中尝试使用 Unix域套接字会导致 libpq 让该数据结构溢出。 TCP/IP 连接工作正常，但是 Unix域套接字不行，这将使回归测试不能工作。

该问题已经被报告给了 IBM，并且已被记录为缺陷报告 PMR29657。如果你升级到 maintenance level 5300-03 或更新，将会包括这个修复。一种快速的解决方法是把/usr/

include/sys/socket.h中的_SS_MAXSIZE改成 1025。在两种情况中，一旦你得到了修正过的头文件，你都需要重编译 PostgreSQL。

16.7.1.3. Internet地址问题

PostgreSQL 依赖系统的getaddrinfo函数来解析listen_addresses、pg_hba.conf等中的 IP 地址。旧版本的 AIX 在这个函数中有各种各样的缺陷。如果你存在与此有关的问题，更新到上文所示的合适的 AIX fix level 将会解决它。

一个用户报告：

当在 AIX 5.3 上实现 PostgreSQL 版本 8.1 时，我们会周期性地碰到问题，在其中统计收集器会“神秘地”无法成功启动。这似乎是在 IPv6 实现中意外行为的结果。看起来 PostgreSQL 和 IPv6 无法和 AIX 5.3 一起很好地工作。

下面任意一种动作都可以“修复”该问题。

- 删除 localhost 的 IPv6 地址：

```
(as root)
# ifconfig lo0 inet6 ::1/0 delete
```

- 从网络服务删除 IPv6。AIX 上的/etc/netsvc.conf大概等价于 Solaris/Linux 上的/etc/nsswitch.conf。在 AIX 上的默认值因此是：

```
hosts=local,bind
```

将其换成：

```
hosts=local4,bind4
```

来使 IPv6 地址的搜索无效。

> **警告**
>
> 这实际上是对有关 IPv6 支持不成熟性的问题的一种变通方案，这在 AIX 5.3 发布的过程中有了显著地改进。它可以和 AIX 5.3 一起工作，但是不代表对此问题的一种华丽的解决方案。有报告称该变通方案不仅仅是多余的，还会在 AIX 6.1 上导致问题，在 AIX 6.1 中 IPv6 支持已变得更加成熟。

16.7.1.4. 内存管理

AIX 的特别之处在于它的内存管理。你可能有一个装备有好多个吉字节空闲 RAM 的服务器，但是在运行应用时仍然会得到内存不足或者地址空间错误。一个例子是加载扩展会因为罕见的错误失败。例如，作为 PostgreSQL 安装的拥有者运行：

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": A memory address
 is not in the address space for the process.
```

作为拥有 PostgreSQL 安装的组中的非拥有者运行：

```
=# CREATE EXTENSION plperl;
```

ERROR: could not load library "/opt/dbs/pgsql/lib/plperl.so": Bad address

另一个例子是 PostgreSQL 服务器日志中的内存不足错误，每次内存分配接近或者超过 256 MB 时都会失败。

所有这些问题的总体成因是服务器进程所用的寻址空间和内存模型。默认情况下，所有在 AIX 上编译的二进制都是32位。这并不依赖于硬件类型或使用的内核。这些32位进程被限制在 4GB 的内存中，并被使用几种模型之一安排成 256 MB 的段。该默认值允许在堆中低于 256 MB，因为它和栈共享一个单独的段。

在plperl的例子中，检查你的 umask 和你的 PostgreSQL 安装中的二进制的权限。这个例子中涉及的二进制是32位的并且被用模式 750 而不是 755 安装。由于这种方式的权限设置，只有所有者或拥有组的成员可以载入该库。因为它不是所有人可读的，载入器将该对象放在进程的堆中而不是它应该被放入的共享库段中。

这个问题的“理想的”解决方案是使用 PostgreSQL 的64位编译，但是这不是总是实用的，因为有32位处理器的系统可以编译64位二进制但是却不能运行它。

如果想要一个 32 位二进制，在开始 PostgreSQL 服务器之前将LDR_CNTRL设置为MAXDATA=0xn0000000，其中 1 <= n <= 8，并且尝试不同的值以及postgresql.conf设置来找一个能让你满意的配置。这种LDR_CNTRL的使用告诉 AIX 你希望服务器留出MAXDATA字节给堆，以 256 MB 的段分配。当你找到了一个可工作的配置时，ldedit可以被用来修改二进制，这样它们默认使用想要的堆尺寸。PostgreSQL 也可以被重新编译，传递configure LDFLAGS="-Wl,-bmaxdata:0xn0000000"来达到相同的效果。

对于一个 64 位编译，设置OBJECT_MODE为 64 并且传递CC="gcc -maix64"和LDFLAGS="-Wl,-bbigtoc"给configure（给xlc的选项可能不同）。如果你省略OBJECT_MODE的输出，你的编译可能会因为链接器错误而失败。当OBJECT_MODE被设置时，它告诉 AIX 的编译工具（如ar、as和ld）默认要处理哪些对象类型。

默认情况下，过量使用页面空间的情况可能会发生。不过我们还没有看到过，当进程用尽内存并且出现了过量使用时 AIX 会杀死进程。我们见到过的最接近于此的是 fork 失败，其原因是系统觉得已经没有足够的内存给另一个进程。和 AIX 的很多其他部分一样，如果这成为了一个问题，页面空间分配方法和耗尽内存导致的杀死在系统范围或进程范围是可以配置的。

参考和资源

“Large Program Support[1]”. AIX Documentation: General Programming Concepts: Writing and Debugging Programs.

“Program Address Space Overview[2]”. AIX Documentation: General Programming Concepts: Writing and Debugging Programs.

“Performance Overview of the Virtual Memory Manager (VMM)[3]”. AIX Documentation: Performance Management Guide.

“Page Space Allocation[4]”. AIX Documentation: Performance Management Guide.

“Paging-space thresholds tuning[5]”. AIX Documentation: Performance Management Guide.

Developing and Porting C and C++ Applications on AIX[6]. IBM Redbook.

16.7.2. Cygwin

[1] http://publib.boulder.ibm.com/infocenter/pseries/topic/com.ibm.aix.doc/aixprggd/genprogc/lrg_prg_support.htm
[2] http://publib.boulder.ibm.com/infocenter/pseries/topic/com.ibm.aix.doc/aixprggd/genprogc/address_space.htm
[3] http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.doc/aixbman/prftungd/resmgmt2.htm
[4] http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.doc/aixbman/prftungd/memperf7.htm
[5] http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.doc/aixbman/prftungd/memperf6.htm
[6] http://www.redbooks.ibm.com/abstracts/sg245674.html?Open

PostgreSQL 可以使用 Cygwin 来编译，它是用于 Windows 的一个类 Linux 环境，但是这种方法不如原生 Windows 编译见第 17 章并且我们已经不再推荐在 Cygwin 下运行一个服务器。

在从源代码编译时，按照正常安装过程进行（即./configure; make; 等；只要注意下列 Cygwin 相关的区别：

- 将你的路径设置为使用 Cygwin 的 bin 目录并且把它放在 Windows 工具的前面。这将帮助避免很多编译的问题。
- 不支持adduser命令；使用 Windows NT、2000 或 XP 上的用户管理应用来替代。否则，跳过这一步。
- 不支持su命令；在 Windows NT、2000 或 XP 上使用 ssh 来模拟 su。否则，跳过这一步。
- 不支持 OpenSSL。
- 为共享内存支持启动cygserver。要这样做，输入命令/usr/sbin/cygserver &。这个程序在你启动 PostgreSQL 服务器或初始化一个数据集簇（initdb）时的任何时刻都需要被运行。默认的cygserver配置可能需要被更改（例如增加SEMMNS）来防止 PostgreSQL 因为缺少系统资源而失败。
- 在某些不使用 C 区域的系统上编译可能会失败。要修复这个问题，通过在边以前export LANG=C.utf8把区域设置为 C，并且在安装完 PostgreSQL 之后把区域恢复成之前的设置。
- 并行回归测试（make check）可能产生虚假的回归测试错误，这是由于溢出的listen()连接缓冲区，它会导致连接拒绝错误或挂起。你可以使用MAX_CONNECTIONS来限制连接数：

  ```
  make MAX_CONNECTIONS=5 check
  ```

  （在某些系统上你可以有大约 10 个同时连接）。

可以把cygserver PostgreSQL 服务器安装为 Windows NT 服务。关于如何这样做的信息，请参考包含在 Cygwin 上 PostgreSQL 二进制包中的README文档。它被安装在目录/usr/share/doc/Cygwin中。

16.7.3. HP-UX

给定合适的系统补丁级别和编译工具，PostgreSQL 7.3+ 应该可以工作在运行 HP-UX 10.X 或 11.X 的 Series 700/800 PA-RISC 机器上。至少一个开发者例行地在 HP-UX 10.20 上测试过，并且我们有在 HP-UX 11.00 和 11.11 上成功安装的报告。

除了 PostgreSQL 源代码发布，你将需要 GNU make（HP 的 make 不行），并且需要 GCC 或 HP 的 ANSI C 编译器。如果你想从 Git 源编译而不是一个发布包，你还将需要 Flex（GNU lex）和 Bison（GNU yacc）。我们还推荐确认你真的在使用最新的 HP 补丁。最低限度下，如果你在 HP-UX 11.11 上编译 64 位二进制，你可能需要 PHSS_30966 (11.11) 或一个后继补丁，否则initdb可能中止：

```
PHSS_30966  s700_800 ld(1) and linker tools cumulative patch
```

在一般原则上，你应该使用 libc 和 ld/dld 的当前补丁，如果你在使用 HP 的 C 编译器也一样要用当前的编译器补丁。它们最新补丁的免费拷贝请见 HP 的支持站点如ftp://us-ffs.external.hp.com/。

如果你正在一台 PA-RISC 2.0 机器上编译并且项使用 GCC 得到 64 位二进制，你必须使用 GCC 的 64 位版本。

如果你正在一台 PA-RISC 2.0 机器上编译并且想让编译好的二进制运行在 PA-RISC 1.1 机器上，你将需要在CFLAGS中指定+DAportable。

如果你正在一台 HP-UX Itanium 机器上编译，你将需要最新的 HP ANSI C 编译器，以及它的依赖补丁或后继补丁：

```
PHSS_30848  s700_800 HP C Compiler (A.05.57)
PHSS_30849  s700_800 u2comp/be/plugin library Patch
```

如果你同时有 HP 的 C 编译器和 GCC 的编译器，那么在运行configure时你可能希望显式地选择要使用的编译器：

```
./configure CC=cc
```

用于 HP 的 C 编译器，或者

```
./configure CC=gcc
```

用于 GCC。如果你忽略这个设置，configure 在可以选择时会使用gcc。

默认的安装目标位置是/usr/local/pgsql，你可能希望修改它为/opt之下的某个地方。如果是这样，使用configure的--prefix开关。

在回归测试中，在几何测试中可能会有某些低序位差别，这会根据你使用的编译器和数学库版本而变化。任何其他错误都需要怀疑。

16.7.4. macOS

在最新的macOS版本中，有必要将“sysroot” 路径嵌入用于查找某些系统头文件的include选项中。这导致configure 脚本的输出会有所不同，具体取决于在configure期间使用的SDK版本。 在简单的情况下，这应该不会造成任何问题，但是，如果您要尝试在与构建服务器代码不同 的计算机上构建扩展程序，则可能需要强制使用其他sysroot路径。为此，需要设置 PG_SYSROOT，例如：

```
make PG_SYSROOT=/desired/path all
```

要在您的计算机上找到合适的路径，请运行

```
xcodebuild -version -sdk macosx Path
```

请注意，实际上不建议使用与构建核心服务器不同的sysroot版本构建扩展。 在最坏的情况下，它可能导致难以调试的ABI不一致。

您还可以在配置时选择非默认的sysroot路径，通过在 configure中指定PG_SYSROOT：

```
./configure ... PG_SYSROOT=/desired/path
```

macOS的“系统完整性保护”（SIP） 功能破坏了make check，因为它阻止通过 设置所需的DYLD_LIBRARY_PATH传递给被测试的可执行文件。 您可以通过在make check之前执行make install来解决此问题。 不过，大多数Postgres开发人员关闭了SIP。

16.7.5. MinGW/原生 Windows

用于 Windows 的 PostgreSQL 可以使用 MinGW 编译，它是一个用于微软操作系统的类 Unix 的编译环境。也可以使用微软的Visual C++编译器套件来编译。MinGW 编译使用本章中描述的正常编译系统；而 Visual C++ 编译的工作完全不同并且在 第 17 章中描述。后者是一

种完全原生的编译并且没有像 MinGW 那样使用额外软件。在 PostgreSQL 的主网站上有一个现成的安装器可用。

原生 Windows 移植要求一个 Windows 2000 或更高的 32 或 64 位版本。早期的操作系统没有足够的基础设施（但 Cygwin可以用在它们之上）。类 Unix 的编译工具 MinGW 和 MSYS（一个 Unix 工具集合，用于运行如configure之类的 shell 脚本）可以从http://www.mingw.org/下载。运行结果二进制两者都需要，它们只在创建二进制时需要。

要使用 MinGW 编译 64 位二进制，从http://mingw-w64.sourceforge.net/安装 64 位工具。把它放在PATH中的 bin 目录，并且使用--host=x86_64-w64-mingw32选项运行configure.

在你安装完所有的东西之后，我们建议你在CMD.EXE下运行psql，因为 MSYS 控制台有缓冲问题。

16.7.5.1. 在 Windows 上收集崩溃转储

如果 PostgreSQL 在 Windows 上崩溃，它有能力产生minidumps，这可以被用来追踪崩溃发生的原因，这与 Unix 上的核心转储相似。这些转储可以被使用Windows Debugger Tools或Visual Studio读取。要启用在 Windows 上的转储生成，可在集簇数据目录下创建一个名为crashdumps的子目录。转储将被写入到这个目录，转储的名字基于崩溃进程的标识符和崩溃的当前时间来确定。

16.7.6. Solaris

PostgreSQL 在 Solaris 上得到了很好的支持。你的操作系统越新，你将会碰到更少的问题；细节如下。

16.7.6.1. 要求的工具

你可以使用 GCC 或 Sun 的编译器套件进行编译。为了更好的代码优化，我们强烈推荐在 SPARC 架构下使用 Sun 的编译器。我们已经得到一些使用 GCC 2.95.1 时的问题报告；我们推荐 GCC 2.95.3 或之后的版本。如果你正在使用 Sun 的编译器，注意不要选择/usr/ucb/cc；而是使用/opt/SUNWspro/bin/cc。

你可以从https://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/下载 Sun Studio。很多 GNU 工具都被整合到了 Solaris 10，或者它们在 Solaris companion CD 中。如果你喜欢用于老版本 Solaris 的包，你可以在http://www.sunfreeware.com找到这些工具。如果你想要源码，在https://www.gnu.org/prep/ftp上找找。

16.7.6.2. configure 抱怨一个失败的测试程序

如果configure抱怨一个失败的测试程序，可能的情况是运行时链接器无法找到某些库，可能是libz、libreadline或某些其他非标准库如 libssl。要向它指出正确的位置，在configure命令行上设置LDFLAGS环境变量，例如：

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

更多信息可见ld手册页。

16.7.6.3. 64-位编译有时会崩溃

在 Solaris 7 和更老的版本上，64-位版本的 libc 有一个有缺陷的vsnprintf例程，这导致 PostgreSQL 中不稳定的核心转储。最简单的已知解决方案是强制 PostgreSQL 使用它自己的vsnprintf版本而不是库中的拷贝。要这样做，运行configure之后编辑一个由configure产生的文件： 在文件src/Makefile.global中将行

LIBOBJS =

改成

LIBOBJS = snprintf.o

（可能有其他文件已经被列在这个变量中。顺序无影响）。然后正常编译。

16.7.6.4. 为最优性能编译

在 SPARC 架构上，我们强烈推荐使用 Sun Studio来编译。尝试使用-xO5优化标志来生成显著加快的二进制。不要使用任何修改浮点操作和errno处理（例如-fast）行为的标志。这些标志可能会做出某些非标准 PostgreSQL 行为，例如在日期/时间计算中。

如果你没有理由要使用 SPARC 上的 64 位二进制，最好用 32 位版本。64 位操作较慢并且 64 位二进制比其 32 位变体要慢。并且在另一方面，AMD64 CPU 家族上的32 位代码不是原生的，并且这也是问什么在这个 CPU 族中 32 位代码要明显地更慢。

16.7.6.5. 用 DTrace 来跟踪 PostgreSQL

是的，可以使用 DTrace。详见第 28.5 节

如果你看到postgres可执行程序的链接中断并且报出下面的错误消息：

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

说明你的 DTrace 安装太旧，无法处理静态函数中的探测。你需要 Solaris 10u4 或更新的版本。

第 17 章 在Windows上从源代码安装

对于大部分用户，推荐下载Windows的二进制发布，它在PostgreSQL 的网站上作为一个图形化安装包可供下载。从源代码构建的方式只适合于希望开发或者扩展 PostgreSQL的人们。

有多种不同的方式可以在Windows上构建PostgreSQL。 用微软工具进行构建的最简单方式是安装 Visual Studio Express 2017 for Windows Desktop并使用其中包含的编译器。 也可以使用完整的Microsoft Visual C++ 2005到2017来构建。 在某些情况中除了编译器还需要安装Windows SDK。

也可以使用由MinGW提供的GNU编译器工具来构建PostgreSQL，或者使用适合于旧版本Windows的Cygwin。

使用MinGW或Cygwin的构建用到普通构建系统，见第 16 章第 16.7.5 和第 16.7.2 节在这些环境下，要产生原生的64位二进制代码，请使用MinGW-w64中的工具。这些工具同样可以被用于在32位和64位 Windows上交叉编译其他主机上的目标，例如Linux和macOS。不推荐将Cygwin用于一个产品服务器，它只应被用于老版本的Windows，因为在这些系统中原生构建无法工作，例如Windows 98。官方的二进制代码使用Visual Studio构建。

psql的原生构建不支持命令行编辑。Cygwin构建能支持命令行编辑，因此如果需要在Windows上交互式地使用psql可以用到它。

17.1. 使用Visual C++或Microsoft Windows SDK构建

PostgreSQL可以使用来自微软的Visual C++编译器套件构建。这些编译器可以来自于Visual Studio、Visual Studio Express或者Microsoft Windows SDK的某些版本。如果你还没有准备好一个Visual Studio环境设置，最简单的方式是使用Visual Studio Express 2017 for Windows Desktop中的编译器，或者Windows SDK 8.1中的，两者都可以从微软免费下载。

使用微软编译器套件可以编译得到 32 位和 64 位版本。32 位的 PostgreSQL 可以使用Visual Studio 2005至 Visual Studio 2017（包括精简版）编译， 单独的Windows SDK 6.0至8.1亦可。64位PostgreSQL的构建只被 Microsoft Windows SDK版本6.0a至8.1或 Visual Studio 2008及以上版本支持。 使用Visual Studio 2005至 Visual Studio 2013编译时最低支持 Windows XP和 Windows Server 2003。使用 Visual Studio 2015编译时最低支持Windows Vista和Windows Server 2008。 使用Visual Studio 2017编译时最低支持 Windows 7 SP1和Windows Server 2008 R2 SP1。

使用Visual C++或Platform SDK构建的工具在src/tools/msvc目录中。在构建时，请确定在系统路径中没有来自于MinGW或的Cygwin工具。同样，确保所有需要的Visual C++工具都在PATH中。在Visual Studio中，启动Visual Studio Command Prompt。如果你希望构建一个64位版本，你必须使用64位版本的命令，反之亦然。在Microsoft Windows SDK中，启动该SDK在启动菜单中的CMD shell。在最近的SDK版本中你可以使用setenv命令改变目标CPU架构、构建类型以及目标OS，例如setenv /x86 /release /xp会设置为Windows XP或更高版本上的32位发布构建。使用/?来了解setenv的其他选项。所有命令应该从src\tools\msvc目录运行。

在开始构建之前，你还需要编辑文件config.pl来反映任何你想改变的配置选项，或者要使用的任何第三方库目录。完整的配置在第一次读取并解析文件config_default.pl时确定，然后应用config.pl中的任何改变。例如，要制定你的Python安装的位置，将下面的内容放在config.pl中：

```
$config->{python} = 'c:\python26';
```

你只需要指定那些和config_default.pl中不同的参数即可。

如果你希望设置任何其他环境变量，可创建一个名为buildenv.pl的文件并将需要的命令放在其中。例如，要把不在PATH中的bison路径加上，创建一个包含以下内容的文件：

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

传递更多的命令行参数到Visual Studio构建命令(msbuild 或 vcbuild):

```
$ENV{MSBFLAGS}="/m";
```

17.1.1. 要求

构建PostgreSQL时需要下列附加产品。使用config.pl文件来指定这些库所在的目录。

Microsoft Windows SDK

如果你的构建环境中没有一个受支持的Microsoft Windows SDK版本，推荐你升级到最新版（当前版本为7.1，可从https://www.microsoft.com/download下载）。

你必须总是包括SDK中的Windows头文件和库部分。如果你安装的是一个包括Visual C++编译器的Windows SDK，构建时不需要Visual Studio。注意在版本8.0a，Windows SDK中不再包括一个完整的命令行构建环境。

ActiveState Perl

ActiveState Perl被用来运行构建生成脚本。MinGW或Cygwin Perl是不符合要求的。AcitveState Perl也必须存在于PATH中。其二进制文件可以从https://www.activestate.com下载（注意：需要版本5.8.3及以上，免费标准发布就足够了）。

下面的附加产品在开始时并不要求，但是如果要构建完整的包就需要它们。使用config.pl文件来指定这些库所在的目录。

ActiveState TCL

用于构建PL/Tcl（注意：要求版本8.4，免费标准发布即可）。

Bison和 Flex

Bison和Flex用来从Git构建，但使用发行文件构建时可以不要求。只有Bison 1.875或2.2及以上才能正常工作。Flex则必须是版本2.5.31或以上。

Bison和Flex 都包括在msys工具套件中，它作为 MinGW编译器套件的一部分可以从http://www.mingw.org/wiki/MSYS得到。

你将需要把包含flex.exe和bison.exe 的目录加入到buildenv.pl中的PATH环境变量中， 除非它们已经存在于PATH中。在MinGW的情况下， 目录将是MinGW安装目录的\msys\1.0\bin子目录。

> **注意**
>
> 来自GnuWin32的Bison发布似乎有一个故障，它会导致Bison安装于名称中有空格的目录时发生故障，例如英语安装的默认位置C:\Program Files\GnuWin32。考虑将它安装到C:\GnuWin32或者在PATH环境设置中使用NTFS段路径名（例如C:\PROGRA~1\GnuWin32）。

> **注意**
>
> 在PostgreSQL的FTP站点上的以及被旧文档引用的老式winflex二进制程序在64位Windows主机上会出现“flex: fatal internal error, exec failed”的错误。请使用来自 MSYS 的 Flex。

Diff

Diff是回归测试所需要的，可以从http://gnuwin32.sourceforge.net得到。

Gettext

Gettext用于NLS支持，可以从http://gnuwin32.sourceforge.net得到。注意二进制程序、依赖文件以及开发者文件都需要。

MIT Kerberos

用于 GSSAPI 认证支持。MIT Kerberos可以从http://web.mit.edu/Kerberos/dist/index.html下载。

libxml2 and libxslt

用于XML支持。二进制文件可以从http://zlatkovic.com/pub/libxml得到，源代码可以从http://xmlsoft.org得到。注意libxml2需要iconv，后者也可以在相同的下载位置得到。

OpenSSL

用于SSL支持。二进制文件可以从https://slproweb.com/products/Win32OpenSSL.html下载，源代码可以从https://www.openssl.org下载。

ossp-uuid

用于UUID-OSSP支持（contrib only）。源代码可以从http://www.ossp.org/pkg/lib/uuid/下载。

Python

用于构建PL/Python。二进制文件可以从https://www.python.org下载。

zlib

用于pg_dump和pg_restore中的压缩支持。二进制文件可以从http://www.zlib.net下载。

17.1.2. 针对64位Windows的特殊考虑

在64位Windows上，PostgreSQL只能为x64架构构建，因此无法支持安腾处理器。

不支持在同一个构建树中混合32位和64位版本。构建系统会自动检测它运行在32位还是64位环境中，然后相应地构建PostgreSQL。鉴于此，在构建前启动正确的命令提示很重要。

要使用服务器端的第三方库如python或OpenSSL，该库必须也是64位。在一个64位服务器上载入一个32位库是不被支持的。PostgreSQL支持的一些第三方库可能只有32位版本，在这种情况下它们就不能被用于64位PostgreSQL。

17.1.3. 构建

要在发行配置中构建PostgreSQL的所有部分（默认），运行命令：

```
build
```

要在调试配置中构建PostgreSQL的所有部分，运行命令：

```
build DEBUG
```

要构建单独一个对象，例如psql，运行命令：

```
build psql
build DEBUG psql
```

要将默认的构建配置改变成调试，将下面的内容放在buildenv.pl文件中：

```
$ENV{CONFIG}="Debug";
```

也可以在Visual Studio的图形界面中进行构建。在这种情况下，你需要在命令提示符下运行：

```
perl mkvcbuild.pl
```

然后在Visual Studio中打开生成的pgsql.sln（在源代码树的根目录中）。

17.1.4. 清理和安装

在大部分时间里，Visual Studio的自动依赖跟踪会处理发生改变的文件。但是如果发生了大量的改变，你也许需要清理整个安装。为此，只要运行clean.bat命令，它将会自动清除所有生成的文件。你也可以使用dist参数运行它，这种情况下它的效果和make distclean一样，并且会移除flex/bison的输出文件。

默认情况下，所有的文件都被写入到名为debug或release的子目录中。要将这些文件以标准布局进行安装并且生成初始化和使用数据库所需的文件 ，运行命令：

```
install c:\destination\directory
```

如果你想只安装客户端应用和接口库，那么你可以使用这些命令：

```
install c:\destination\directory client
```

17.1.5. 运行回归测试

要运行回归测试，确保你已经完成了所有所需部分的构建。另外，确保载入整个系统所需的DLL（例如Perl和Python过程语言所需的DLL）都在系统路径中。如果它们不在路径中，通过buildenv.pl文件设置。要运行测试，可以从src\tools\msvc目录运行以下命令之一：

```
vcregress check
vcregress installcheck
vcregress plcheck
vcregress contribcheck
vcregress modulescheck
vcregress ecpgcheck
vcregress isolationcheck
vcregress bincheck
```

```
vcregress recoverycheck
vcregress upgradecheck
```

要更改使用的调度方式（默认为并行），在命令行后增加调度方式，如：

```
vcregress check serial
```

关于回归测试详见第 33 章

用vcregress bincheck和vcregress recoverycheck 分别可以运行客户端程序上的回归测试和恢复测试，这要求 安装了额外的 Perl 模块：

IPC::Run

从编写这份文档时起，IPC::Run没有被包括在 ActiveState Perl 安装或者 ActiveState Perl Package Manager (PPM) 库中。要安装，请从 CPAN （在 https://metacpan.org/release/IPC-Run）下载 IPC-Run-<version>.tar.gz源代码归档并且解压。 编辑buildenv.pl文件，并且增加一个 PERL5LIB 变量 指向解压得到的归档中的lib子目录。例如：

```
$ENV{PERL5LIB}=$ENV{PERL5LIB} . ';c:\IPC-Run-0.94\lib';
```

17.1.6. 构建文档

构建HTML格式的PostgreSQL文档需要一些工具和文件。为所有这些文件创建一个根目录，然后将下面列出的它们分别放在相应子目录中。

OpenJade 1.3.1-2

从https://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/download下载并解压到openjade-1.3.1子目录。

DocBook DTD 4.2

从https://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip下载并解压到docbook子目录。

ISO字符实体

从https://www.oasis-open.org/cover/ISOEnts.zip下载并解压到docbook子目录。

编辑buildenv.pl文件，为根目录的位置增加一个变量，例如：

```
$ENV{DOCROOT}='c:\docbook';
```

要构建文档，运行命令builddoc.bat。注意这实际会运行构建两次以生成索引。生成好的HTML文件将在doc\src\sgml中。

第 18 章 服务器设置和操作

本章讨论如何设置和运行数据库服务器，以及它与操作系统的交互。

18.1. PostgreSQL用户账户

和对外部世界可访问的任何服务器守护进程一样，我们也建议在一个独立的用户账户下运行PostgreSQL。这个用户账户应该只拥有被该服务器管理的数据，并且应该不能被其他守护进程共享（例如，使用用户nobody是一个坏主意）。我们不建议把可执行文件安装为属于这个用户，因为妥协系统可能接着修改它们自己的二进制文件。

要在你的系统中增加一个 Unix 用户账户，查看一个命令useradd或adduser。通常会用postgres（本书中也假定用这个账户），但是你可以使用另一个名称。

18.2. 创建一个数据库集簇

在你能做任何事情之前，你必须在磁盘上初始化一个数据库存储区域。我们称之为一个数据库集簇（SQL标准使用的术语是目录集簇）。一个数据库集簇是被一个运行数据库服务器的单一实例所管理的多个数据库的集合。在初始化之后，一个数据库集簇将包含一个名为postgres的数据库，它表示被功能、用户和第三方应用所使用的默认数据库。数据库服务器本身并不要求postgres数据库存在。另一个在初始化过程中为每一个集簇创建的数据库被称为template1。顾名思义，它将被用于创建后续数据库的模板；它不应该被用于实际工作（在集簇内创建新数据库的更多信息请见第 22 章。

在文件系统术语中，一个数据库集簇是一个单一目录，所有数据都将被存储在其中。我们称它为数据目录或数据区域。在哪里存储你的数据完全由你选择。没有默认的位置，不过/usr/local/pgsql/data或/var/lib/pgsql/data位置比较流行。要初始化一个数据库集簇，使用和PostgreSQL一起安装的命令initdb。你的数据库集簇的文件系统位置由-D选项指定，例如：

```
$ initdb -D /usr/local/pgsql/data
```

注意你必须在使用PostgreSQL用户账户（如前一节所示）登录后执行这个命令。

> **提示**
>
> 作为-D选项的一种替换方案，你可以设置环境变量PGDATA。

另一种替代方案是，你可以通过pg_ctl程序来运行initdb：

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

如果你使用pg_ctl来启停服务器（见第 18.3 节，这种方法可能更直观，以为这样pg_ctl将是你用来管理数据库服务器实例的唯一命令。

如果你指定的目录还不存在，initdb将尝试创建它。当然，如果initdb没有在父目录中的写权限，这将会失败。通常推荐让PostgreSQL用户拥有数据目录及其父目录，这样就不存在上面的问题了。如果想要的父目录也不存在，你将需要先创建它，如果父父目录不可写则使用root 特权。因此，该过程可能像这样：

```
root# mkdir /usr/local/pgsql
```

```
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

如果数据目录存在并且已经包含文件，initdb将拒绝运行。这可以避免无意中覆盖一个已有的安装。

因为数据目录包含所有存储在数据库里的数据，所以最重要的是保护这个目录不受未授权的访问。因此，initdb会回收禁止除PostgreSQL用户，也可以选择组，之外所有用户的访问权限。当组访问启用时，是只读的。它允许相同组中未被授权的用户作为集簇属主，备份集簇数据或者执行其他只需要读访问权限的操作。

注意在现有集群启用或禁用组访问时，需要关闭集群，且重新启动PostgreSQL之前设置所有的目录和文件到恰当的模式。否则，数据目录中会存在多种模式。集群仅可以被其属主访问，恰当的模式应该是，其目录设置为0700，普通文件设置为0600。允许集群被组可读，恰当的模式应该是，其目录设置为0750，普通文件设置为0640。

不过，虽然目录的内容是安全的，但默认的客户端认证设置允许任意本地用户连接到数据库甚至成为数据库超级用户。如果你不信任其他本地用户， 我们建议你使用initdb的-W、--pwprompt或--pwfile选项之一给数据库超级用户赋予一个口令。还可以指定-A md5或-A password，这样就不会使用默认的trust 身份认证。或者在执行initdb之后、第一次启动服务器之前修改生成的pg_hba.conf文件（另外一些可行的方法包括peer认证或者用文件系统权限限制连接。更多信息见第 20 章。

initdb同时也为数据库集簇初始化默认区域。 通常，它将只是使用环境中的区域设置并且把它们应用于被初始化的数据库。 可以为数据库指定一个不同的区域；有关于此的更多信息可以在第 23.1 节找到。 特定数据库集簇中使用的默认排序顺序是通过initdb设置的， 虽然你可以创建使用不同排序顺序的新数据库，但在 initdb 创建的模板数据库中使用的顺序不能更改（除非删除并重建它们）。使用非C或POSIX的区域还会对性能造成影响。因此，第一次就正确地选择很重要。

initdb还为数据库集簇设置默认的字符集编码。通常字符集编码应该选择与区域设置匹配。详见第 23.3 节

非C以及非POSIX区域对于字符集排序依赖于操作系统的排序规则库。这控制着索引中存储的键的排序。为此，通过快照恢复、二进制流复制、更换不同的操作系统或者升级操作系统都不能把一个集簇切换到一种不兼容的排序规则库版本。

18.2.1. 二级文件系统的使用

很多安装会在文件系统（卷）而不是机器的“根”卷上创建它们的数据库集簇。如果你选择这样做，我们不建议尝试使用二级卷的顶层目录（挂载点）作为数据目录。最好的做法是在PostgreSQL用户拥有的挂载点目录中创建一个目录，然后在其中创建数据目录。这可以避免权限问题，特别是对于pg_upgrade这类操作，并且它也能在二级卷被断线后确保干净的失败。

18.2.2. 网络文件系统的使用

许多安装会在网络文件系统上创建它们的数据库集簇。有时直接通过NFS， 或通过内部使用NFS的网络附加存储设备（NAS）完成。 PostgreSQL不对 NFS文件系统做特殊处理，即它假定NFS的行为和本地连接的设备完全一样。如果客户端或者服务器NFS没有提供标准的文件系统语义，这将导致可靠性问题 （参阅https://www.time-travellers.org/shane/papers/NFS_considered_harmful.html）。 具体来说，延迟（异步）写入到NFS服务器可以导致数据损坏问题。 如果可能的话，把NFS文件系统挂载为同步（无高速缓存）可以避免这种灾难。还有，我们不推荐软挂载的NFS文件系统。

存储区域网络（SAN）通常使用非NFS的通讯协议，并且可能或者不可能遭受这类灾难。建议咨询供应商的文档来了解数据一致性保证。PostgreSQL无法做到比它所使用的文件系统更可靠。

18.3. 启动数据库服务器

在任何人可以访问数据库前，你必须启动数据库服务器。 数据库服务器程序是postgres，它必须知道在哪里能找到它要用的数据。这是用-D选项实现的。 因此，启动服务器最简单的方法是：

```
$ postgres -D /usr/local/pgsql/data
```

这将把服务器放在前台运行。这个步骤同样必须以PostgreSQL用户帐户登录来操作。如果没有-D选项，服务器将尝试使用环境变量PGDATA命名的目录。如果这个环境变量也没有提供则导致失败。

通常最好在后台启动postgres。要这样做，使用常用的 Unix shell 语法：

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

如上所示，把服务器的stdout和stderr输出存储到某个地方是非常重要的。这将对审计目的和诊断问题有所帮助（更深入的有关日志文件处理的讨论请见（第 24.3 节。

postgres还接受其它一些命令行选项。更多的信息请见postgres参考页 和下面的第 19 章

这些 shell 语法很容易让人觉得无聊。因此我们提供了包装器程序pg_ctl以简化一些任务。例如：

```
pg_ctl start -l logfile
```

将在后台启动服务器并且把输出放到指定的日志文件中。-D选项和postgres中的一样。pg_ctl还可以用于停止服务器。

通常，你会希望在计算机启动的时候启动数据库服务器。自动启动脚本是操作系统相关的。PostgreSQL在contrib/start-scripts目录中提供了几种。安装将需要 root 权限。

不同的系统在引导时有不同的启动守护进程的习惯。许多系统有一个文件/etc/rc.local或/etc/rc.d/rc.local。其他的使用init.d或rc.d目录。不管你做什么，服务器必须由PostgreSQL用户账户而不是 root或任何其他用户启动。因此你可能应该在你的命令中使用su postgres -c '...'这种形式。例如：

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

下面是一些更加与操作系统相关的建议（在每一种情况中要确保在我们展示通用值的地方使用正确的安装目录和用户名）。

- 对于FreeBSD，找找PostgreSQL源码发布中的文件contrib/start-scripts/freebsd。
- 在OpenBSD上， 把下面几行加到/etc/rc.local文件中：

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ];
 then
    su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/
postgresql/log -D /usr/local/pgsql/data'
    echo -n ' postgresql'
fi
```

- 在Linux系统上将

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

加入到/etc/rc.d/rc.local或/etc/rc.local中，还可以在PostgreSQL的源码发布中找找文件contrib/start-scripts/linux。

在使用systemd时，可以使用下面的服务单元文件（例如/etc/systemd/system/postgresql.service）：

```
[Unit]
Description=PostgreSQL database server
Documentation=man:postgres(1)

[Service]
Type=notify
User=postgres
ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
TimeoutSec=0

[Install]
WantedBy=multi-user.target
```

使用Type=notify要求服务器的二进制文件使用configure --with-systemd编译。

要仔细地考虑超时设置。在写作这份文档时，systemd的默认超时时长是 90 秒，并且将会杀死没有在这段时间内报告准备好的进程。但是PostgreSQL服务器可能因为执行崩溃恢复而导致启动过程大大超过这个默认时间。建议的值是 0 禁用超时逻辑。

- 在NetBSD上，你可以根据爱好选择FreeBSD或Linux的启动脚本。
- 在Solaris上，创建一个名为/etc/init.d/postgresql的文件，其中包含下列行：

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
```

然后在/etc/rc3.d中创建一个符号链接S99postgresql指向它。

当服务器在运行时，它的PID被保存在数据目录中的postmaster.pid文件。这样做 可以防止多个服务器实例运行在同一个数据目录中，并且也可以被用来关闭服务器。

18.3.1. 服务器启动失败

有几个常见的原因会导致服务器启动失败。通过检查服务器日志或使用手工启动的方法（不做标准输出或标准错误的重定向）， 就可以看到出现什么错误消息。下面我们详细地解释一些最常见的错误消息。

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few
 seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

正如这个消息所说的，这表示：你试图在一个已经有服务器运行着的端口上再启动另一个服务器。不过，如果核心错误消息不是Address already in use或其变体，那就有可能是别的问题。 例如，试图在一个被保留的端口上启动服务器会收到下面这样的消息：

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few
 seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

像这样的消息：

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

可能意味着你的内核对共享内存区的限制小于PostgreSQL试图创建的工作区域（本例中是4011376640 字节）。或者可能意味着根本就没有 System-V 风格的共享内存支持被配置在你的内核中。作为一种临时的解决方案， 你可以试着以小于正常数量的缓冲区（shared_buffers）启动服务器。 你最终还是会希望重新配置内核以增加共享内存允许的尺寸。 当你试图在同一台机器上启动多个服务器，并且它们所需的总空间超过了内核的限制，也会报这个错。

一个这样的错误：

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

并不意味着你已经用光了磁盘空间。它的意思是你的内核对System V信号量的限制小于PostgreSQL想创建的数量。和上面一样，你可以通过减少允许的连接数（max_connections）来绕开这个限制，但最终你还是会希望提高内核的限制。

如果你收到一个“illegal system call”错误， 那么很有可能是你的内核根本不支持共享内存或者信号量。这种情况下你唯一的选择就是重新配置内核并且把这些特性打开。

关于配置System V IPC功能的细节请见第 18.4.1 节

18.3.2. 客户端连接问题

尽管可能在客户端出现的错误情况范围宽广而且是应用相关的，但的确有几种与服务器的启动方式直接相关。除了下面提到的几种错误之外的问题都应该在相应的客户端应用文档中。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

这是常见的“I couldn't find a server to talk to”失败。上面的情况看起来是发生在尝试 TCP/IP 通信时。常见的错误是忘记把服务器配置成允许 TCP/IP 连接。

另外，当试图通过 Unix 域套接字与本地服务器通信时，你会看到这个：

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

最后一行可以验证客户端是不是尝试连接到正确的位置。如果实际上没有服务器在那里运行，典型的核心错误消息将是Connection refused或No such file or directory（值得注意的是这种环境中的Connection refused并不表示服务器得到了你的连接请求并拒绝了它。那种情况会产生一个不同的消息，如第 20.15 节所示）。其它像Connection timed out这样的消息可能表示更基础的问题，如缺少网络连接。

18.4. 管理内核资源

PostgreSQL某些时候会耗尽操作系统的各种资源限制，当同一个系统上运行着多个拷贝的服务器或在一个非常大的安装中时尤其如此。本节解释了PostgreSQL使用的内核资源以及你可以采取的用于解决内核资源消耗相关问题的步骤。

18.4.1. 共享内存和信号量

PostgreSQL需要操作系统提供进程间通信(IPC)特性， 特别是共享内存和信号量。Unix驱动的系统通常提供 “System V” IPC、 “POSIX” IPC，或者两者都有。 Windows有它自己的这些功能的实现，这里不讨论。

完全缺少这些功能通常表现为服务器启动时的“Illegal system call”错误。这种情况下，除了重新配置内核之外别无选择。PostgreSQL没有它们就不能工作。 不过，在现代操作系统中这种情况是罕见的。

在启动服务器时，PostgreSQL通常分配少量的System V共享内存， 和大量的POSIX (mmap)共享内存。另外， 在服务器启动时会创建大量信号量，这些信号量可以是System V或POSIX风格。 目前，POSIX信号量用于Linux和FreeBSD系统，而其他平台则使用System V信号量。

> **注意**
>
> 在PostgreSQL 9.3之前，只使用了System V共享内存， 所以启动服务器所需的System V共享内存的数量更大一些。 如果你在运行着一个老版本的服务器，请参考该服务器版本的文档。

System V IPC特性通常受系统范围分配限制的限制。 当PostgreSQL超出了这些限制之一时，服务器会拒绝启动并且并且留下一条有指导性的错误消息，其中描述了问题以及应该怎么做（又见第 18.3.1 节。相关的内核参数在不同系统之间的命名方式一致，表 18.1给出了一个概述。不过，设置它们的方法却多种多样。下面给出了对于某些平台的建议：

表 18.1. System V IPC参数

名称	描述	运行一个PostgreSQL实例所需的值值
SHMMAX	共享内存段的最大尺寸（字节）	至少 1kB，但是默认值通常要高一些
SHMMIN	共享内存段的最小尺寸（字节）	1
SHMALL	可用共享内存的总量（字节或页面）	如果是字节，同SHMMAX；如果是页面， 为ceil(SHMMAX/PAGE_SIZE)，加上其他应用程序的空间
SHMSEG	每个进程的最大共享内存段数目	只需要 1 段，但是默认值高很多
SHMMNI	系统范围内的最大共享内存段数目	像SHMSEG外加其他应用的空间
SEMMNI	信号量标识符（即，集合）的最大数目	至少ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)加上其他应用程序的空间
SEMMNS	系统范围内的最大信号量数目	ceil((max_connections + autovacuum_max_workers +

名称	描述	运行一个PostgreSQL实例所需的值值
		max_worker_processes + 5) / 16) * 17外加其他应用的空间
SEMMSL	每个集合中信号量的最大数目	至少 17
SEMMAP	信号量映射中的项数	见文本
SEMVMX	信号量的最大值	至少 1000 （默认值常常是32767，如非必要不要更改）

PostgreSQL要求少量字节的 System V 共享内存（在 64 位平台上通常是 48 字节） 用于每一个服务器拷贝。在大多数现代操作系统上，这个量很容易得到。但是，如果你运行了很多个服务器拷贝， 或者其他应用也在使用 System V 共享内存，可能需要增加SHMALL（系统范围内 System V 共享内存的总量）。注意在很多系统上SHMALL是以页面而不是字节来度量。

不太可能出问题的是共享内存段的最小尺寸（SHMMIN），对PostgreSQL来说应该最多大约是 32 字节（通常只是1）。而系统范围（SHMMNI）或每个进程（SHMSEG）的最大共享内存段数目不太可能会导致问题，除非你的系统把它们设成零。

当使用System V信号量时，PostgreSQL对每个允许的连接（max_connections）、每个允许的自动清理工作者进程（autovacuum_max_workers）和每个允许的后台进程（max_worker_processes）使用一个信号量， 以16个为一个集合。每个这种集合还包含第 17 个信号量， 其中存储一个“magic number”，以检测和其它应用使用的信号量集合的冲突。 系统里的最大信号量数目是由SEMMNS设置的， 因此这个值必须至少和max_connections加autovacuum_max_workers再加max_worker_processes一样大， 并且每 16 个连接外加工作者还要另外加一个（见表 18.4中的公式）。参数SEMMNI 决定系统中同一时刻可以存在的信号量集合的数目限制。因此这个参数必须至少为ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)。降低允许的连接数目是一种临时的绕开失败（来自函数semget）的方法，通常使用让人混乱的措辞“No space left on device”。

在某些情况下可能还有必要增大SEMMAP，使之至少与SEMMNS相近。如果系统有这个参数(很多系统没有)，这个参数定义信号量资源映射的尺寸，在其中每个连续的可用信号量块都需要一项。 每当一个信号量集合被释放，那么它要么会被加入到该与被释放块相邻的一个现有项，或者它会被注册在一个新映射项中。如果映射被填满，被释放的信号量将丢失（直到重启）。因此信号量空间的碎片时间长了会导致可用的信号量比应有的信号量少。

与“semaphore undo”有关的其他各种设置，如SEMMNU和SEMUME 不会影响PostgreSQL。

当使用POSIX信号量时，所需的信号量数量与System V相同， 即每个允许的连接(max_connections)、允许的自动清理工作进程 (autovacuum_max_workers)和允许的后台进程 (max_worker_processes)一个信号量。 在首选此选项的平台上，POSIX信号量的数量没有特定的内核限制。

AIX

至少到版本 5.1 为止，不再需要对这些参数（例如SHMMAX）做任何特殊的配置，这看起来就像是被配置成允许所有内存都被用作共享内存。这是一种通常被用于其他数据库（DB/2）的配置。

但是，可能需要修改/etc/security/limits中的全局ulimit信息，默认的文件尺寸硬限制（fsize）和文件数量（nofiles）可能太低。

FreeBSD

可以使用sysctl或loader接口来改变默认IPC配置。下列参数可以使用sysctl设置：

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
# sysctl kern.ipc.semmap=256
```

要让这些设置在重启之后也保持，请修改/etc/sysctl.conf。

对于sysctl所关心的来说这些信号量相关的设置都是只读的，但是可以在/boot/loader.conf中设置：

```
kern.ipc.semmni=256
kern.ipc.semmns=512
```

修改该配置文件后，需要重启一次让新设置生效。

你可能也希望你的内核将共享内存锁定在 RAM 中并且防止它被换页到交换分区。这可以使用sysctl的设置 kern.ipc.shm_use_phys来完成。

如果通过启用sysctl的security.jail.sysvipc_allowed运行在 FreeBSD jail 中，运行在不同 jail 中的postmaster应当被不同的操作系统用户运行。这可以提高安全性，因为它阻止非 root 用户干涉不同 jail 中的共享内存或信号量，并且它允许 PostgreSQL IPC 清理代码正确地工作（在 FreeBSD 6.0 及其后的版本中，IPC 清理代码不能正确地检测到其他 jail 中的进程，也不能阻止不同 jail 中的 postmaster 运行在相同的端口）。

FreeBSD 4.0 之前的版本的工作与旧版OpenBSD相似（见下文）。

NetBSD

在NetBSD 5.0 及其后的版本中，IPC 参数可以使用sysctl调整。例如：

```
$ sysctl -w kern.ipc.semmni=100
```

要使这些设置在重启后保持，请修改/etc/sysctl.conf。

作为NetBSD的默认设置，你总是会想 调大kern.ipc.semmni和kern.ipc.semmns的值，因为他们实在太小了。

你可能也希望你的内核将共享内存锁定在 RAM 中并且防止它被换页到交换分区。这可以使用sysctl的设置 kern.ipc.shm_use_phys来完成。

NetBSD 5.0 以前的版本的工作与旧版OpenBSD相似（见下文），除了那些内核参数应该用关键词options设置而不是option。

OpenBSD

在OpenBSD3.3及以后版本，使用sysctl命令，IPC参数可以被自动调节，例如：

```
# sysctl kern.seminfo.semmni=100
```

要使这些设置在重启后保持，请修改/etc/sysctl.conf。

作为OpenBSD的默认配置，你总是会想调
大kern.seminfo.semmni和kern.seminfo.semmns的值，因为他们实在太小了。

在较早的OpenBSD 版本中，你需要编译定制化内核来修改这些IPC参数。也要确保SYSVSHM和SYSVSEM选项为启用状态。(这两项默认都是启用状态。) 下面给出一些内核配置文件中如何设置这些参数的例子：

```
option          SYSVSHM
option          SHMMAXPGS=4096
option          SHMSEG=256

option          SYSVSEM
option          SEMMNI=256
option          SEMMNS=512
option          SEMMNU=256
```

HP-UX

默认的设置可以满足正常的安装。在HP-UX 10 上，SEMMNS的出厂默认值是 128，这可能对大型数据库站点太低。

IPC参数可以在Kernel Configuration → Configurable Parameters下的System Administration Manager（SAM）中被设置。当你完成时选择Create A New Kernel。

Linux

默认的最大段尺寸是 32 MB，并且默认的最大总尺寸是 2097152 个页面。一个页面几乎总是 4096 字节，除了在使用少见“huge pages”的内核配置中（使用getconf PAGE_SIZE来验证）。

共享内存尺寸设置可以通过sysctl接口来更改。例如，要允许 16 GB：

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

另外在重启之间这些设置可以被保存在文件/etc/sysctl.conf中。我们强烈推荐这样做。

古老的发型可能没有sysctl程序，但是可以通过操纵/proc文件系统来得到等效的更改：

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

剩下的默认值都被设置得很宽大，并且通常不需要更改。

macOS

在 macOS 中配置共享内存的推荐方法是创建一个名为/etc/sysctl.conf的文件，其中包含这样的变量赋值：

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

注意在某些 macOS 版本中，所有五个共享内存参数必须在/etc/sysctl.conf中设置，否则值将会被忽略。

注意近期的 macOS 版本会忽略把SHMMAX设置成非 4096 倍数值的尝试。

在这个平台上，SHMALL以 4kB 的页面度量。

在更老的 macOS 版本中，你将需要重启来让共享内存参数的更改生效。到了 10.5，可以使用sysctl随时改变除了SHMMNI之外的所有参数。但是最好还是通过/etc/sysctl.conf来设置你喜欢的值，这样重启之后这些值还能被保持。

只有在 macOS 10.3.9 及以后的版本中才遵循/etc/sysctl.conf文件。如果你正在使用 10.3.x 之前的发布，你必须编辑文件/etc/rc并且在下列命令中改变值：

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

注意/etc/rc通常会被 macOS 的系统更新所覆盖，因此你应该在每次更新之后重做这些编辑。

在 macOS 10.2 及更早的版本中，应该在文件/System/Library/StartupItems/SystemTuning/SystemTuning中编辑这些命令。

Solaris 2.6 至 2.9 (Solaris 6 至 Solaris 9)

相似的设置可以在/etc/system中更改，例如：

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

你需要重启来让更改生效。关于更老版本的 Solaris 下的共享内存的信息请见http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html。

Solaris 2.10 (Solaris 10) 及以后
OpenSolaris

在 Solaris 10 及以后的版本以及 OpenSolaris 中，默认的共享内存和信号量设置已经足以应付大部分PostgreSQL应用。Solaris 现在将SHMMAX的默认值设置为系统 RAM的四分之一。要进一步调整这个设置，使用与postgres用户有关的一个项目设置。例如，以root运行下列命令：

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-
memory=(privileged,8GB,deny)" -U postgres -G postgres user.postgres
```

这个命令增加user.postgres项目并且将用于postgres用户的最大共享内存设置为 8GB，并且在下次用户登录进来时或重启PostgreSQL（不是重新载入）时生效。上述假定PostgreSQL是由postgres组中的postgres用户所运行。不需要重新启动服务器。

对于将有巨大数量连接的数据库服务器，我们推荐的其他内核设置修改是：

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

此外，如果你正在在一个区中运行PostgreSQL，你可能也需要提升该区的资源使用限制。更多关于projects 和prctl的信息请见System Administrator's Guide中的"Chapter2: Projects and Tasks"。

18.4.2. systemd RemoveIPC

如果正在使用systemd，则必须注意IPC资源（共享内存和信号量） 不会被操作系统过早删除。从源代码安装PostgreSQL时，这尤其值得关注。 PostgreSQL发布包的用户不太可能受到影响，因为postgres用户通常是作为系统用户创建的。

控制当用户完全退出时是否移除IPC对象。系统用户免除。 此设置在死板的systemd中默认为on， 但某些操作系统分配默认为关闭。

当此设置打开时，典型的观察效果是PostgreSQL服务器使用的信号量对象在明显随机的时间被删除， 从而导致服务器崩溃，并显示日志消息

```
LOG: semctl(1234567890, 0, IPC_RMID, ...) failed: Invalid argument
```

不同类型的IPC对象（共享内存与信号量，System V与POSIX）在systemd 中略有不同，因此可能会发现某些IPC资源不会像其他IPC资源一样被删除。 但依靠这些微妙的差异是不可取的。

“注销用户”可能会作为维护工作的一部分发生，或者当管理员以 postgres用户或类似名称登录时手动发生，所以通常难以防止。

什么是“系统用户”是由/etc/login.defs中的 SYS_UID_MAX设置在systemd编译时确定的。

打包和部署脚本应该小心，通过使用useradd -r、 adduser --system或等价物来创建postgres用户作为系统用户。

或者，如果用户帐户创建不正确或无法更改，建议设置

```
RemoveIPC=no
```

在/etc/systemd/logind.conf或其他适当的配置文件中。

> **小心**
>
> 至少要确保这两件事中的一件，否则PostgreSQL服务器将非常不可靠。

18.4.3. 资源限制

Unix类操作系统强制了许多种资源限制，这些限制可能干扰你的PostgreSQL服务器的操作。尤其重要的是对每个用户的进程数目的限制、每个进程打开文件数目的限制以及每个进程可用的内存的限制。这些限制中每个都有一个“硬”限制和一个“软”限制。实际使用的是软限制，但用户可以自己修改成最大为硬限制的数目。而硬限制只能由root用户修改。系统调用setrlimit负责设置这些参数。 shell的内建命令ulimit（Bourne shells）或limit（csh）被用来从命令行控制资源限制。 在 BSD 衍生的系统上，/etc/login.conf文件控制在登录期间设置的各种资源限制。详见操作系统文档。相关的参数是maxproc、openfiles和datasize。例如：

```
default:\
...
        :datasize-cur=256M:\
```

```
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

（-cur是软限制。增加-max可设置硬限制）。

内核也可以在某些资源上有系统范围的限制。

- 在Linux上，/proc/sys/fs/file-max决定内核可以支持打开的最大文件数。你可以通过往该文件写入一个不同的数值修改此值， 或者通过在/etc/sysctl.conf中增加一个赋值来修改。 每个进程的最大打开文件数限制是在编译内核的时候固定的；更多信息请见/usr/src/linux/Documentation/proc.txt。

PostgreSQL服务器为每个连接都使用一个进程， 所以你应该至少和允许的连接同样多的进程，再加上系统其它部分所需要的进程数目。 通常这个并不是什么问题，但如果你在一台机器上运行多个服务器，资源使用可能就会紧张。

打开文件的出厂默认限制通常设置为“socially friendly”的值， 它允许许多用户在一台机器上共存，而不会导致不成比例的系统资源使用。 如果你在一台机器上运行许多服务器，这也许就是你想要的，但是在专门的服务器上， 你可能需要提高这个限制。

在另一方面，一些系统允许独立的进程打开非常多的文件；如果不止几个进程这么干，那系统范围的限制就很容易被超过。如果你发现这样的现像， 并且不想修改系统范围的限制，你就可以设置PostgreSQL的 max_files_per_process配置参数来限制打开文件数的消耗。

18.4.4. Linux 内存过量使用

在 Linux 2.4 及其后的版本中，默认的虚拟内存行为对PostgreSQL不是最优的。由于内核实现内存过量使用的方法，如果PostgreSQL或其它进程的内存要求导致系统用光虚拟内存，那么内核可能会终止PostgreSQL的 postmaster 进程（主服务器进程）。

如果发生了这样的事情，你会看到像下面这样的内核消息（参考你的系统文档和配置，看看在哪里能看到这样的消息）：

```
Out of Memory: Killed process 12345 (postgres).
```

这表明postgres进程因为内存压力而被终止了。尽管现有的数据库连接将继续正常运转，但是新的连接将无法被接受。要想恢复，PostgreSQL应该被重启。

一种避免这个问题的方法是在一台你确信其它进程不会耗尽内存的机器上运行PostgreSQL。如果内存资源紧张，增加操作系统的交换空间可以帮助避免这个问题，因为内存不足（OOM）杀手（即终止进程这种行为）只有当物理内存和交换空间都被用尽时才会被调用。

如果PostgreSQL本身是导致系统内存耗尽的原因，你可以通过改变你的配置来避免该问题。在某些情况中，降低内存相关的配置参数可能有所帮助，特别是shared_buffers和work_mem两个参数。在其他情况中，允许太多连接到数据库服务器本身也可能导致该问题。在很多情况下，最好减小max_connections并且转而利用外部连接池软件。

在 Linux 2.6 及其后的版本中，可以修改内核的行为，这样它将不会“过量使用”内存。尽管此设置不会阻止OOM 杀手[1]被调用，但它可以显著地降低其可能性并且将因此得到更鲁棒的系统行为。这可以通过用sysctl选择严格的过量使用模式来实现：

```
sysctl -w vm.overcommit_memory=2
```

或者在/etc/sysctl.conf中放置一个等效的项。你可能还希望修改相关的设置vm.overcommit_ratio。 详细信息请参阅内核文档的https://www.kernel.org/doc/Documentation/vm/overcommit-accounting文件。

[1] https://lwn.net/Articles/104179/

另一种方法，可以在改变或不改变vm.overcommit_memory的情况下使用。它将 postmaster 进程的进程相关的OOM score adjustment值设置为-1000，从而保证它不会成为 OOM 杀手的目标。 这样做最简单的方法是在 postmaster 的启动脚本中执行

```
echo -1000 > /proc/self/oom_score_adj
```

并且要在调用 postmaster 之前执行。请注意这个动作必须以 root 完成，否则它将不会产生效果。所以一个被 root 拥有的启动脚本是放置这个动作最容易的地方。如果这样做，你还应该在调用 postmaster 之前在启动脚本中设置这些环境变量：

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

这些设置将导致 postmaster 子进程使用普通的值为零的 OOM score adjustment 运行，所以 OOM 杀手仍能在需要时把它们作为目标。如果你想要子进程用某些其他 OOM score adjustment 值运行，可以为PG_OOM_ADJUST_VALUE使用其他的值（PG_OOM_ADJUST_VALUE也能被省略，那时它会被默认为零）。如果你没有设置PG_OOM_ADJUST_FILE，子进程将使用和 postmaster 相同的 OOM score adjustment 运行，这是不明智的，因为重点是确保 postmaster 具有优先的设置。

更老的 Linux 内核不提供/proc/self/oom_score_adj，但是可能有一个具有相同功能的早期版本，它被称为/proc/self/oom_adj。这种方式工作起来完全相同，除了禁用值是-17而不是-1000。

> **注意**
>
> 有些厂商的 Linux 2.4 内核被报告有着 2.6 过量使用sysctl参数的早期版本。不过，在没有相关代码的 2.4 内核里设置vm.overcommit_memory为 2 将会让事情更糟。我们推荐你检查一下实际的内核源代码（见文件mm/mmap.c中的vm_enough_memory函数），验证一下这个是在你的内核中是被支持的， 然后再在 2.4 安装中使用它。文档文件overcommit-accounting的存在不能当作是这个特性存在的证明。如果有疑问，请咨询一位内核专家或你的内核厂商。

18.4.5. Linux 大页面

当PostgreSQL使用大量连续的内存块时，使用大页面会减少开销， 特别是在使用大shared_buffers时。 要在PostgreSQL中使用此特性，您需要一个包含CONFIG_HUGETLBFS=y和CONFIG_HUGETLB_PAGE=y的内核。 您还必须调整内核设置vm.nr_hugepages。要估计所需的巨大页面的数量， 请启动PostgreSQL，而不启用巨大页面，并使用 /proc文件系统来检查postmaster的匿名共享内存段大小以及系统的巨大页面大小。 这可能看起来像：

```
$ head -1 $PGDATA/postmaster.pid
4170
$ pmap 4170 | awk '/rw-s/ && /zero/ {print $2}'
6490428K
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

6490428 / 2048 大约是3169.154，因此在这个示例中你至少需要 3170个大页面，我们可以设置：

```
$ sysctl -w vm.nr_hugepages=3170
```

如果机器上的其他程序也需要大页面，则更大的设置将是合适的。 不要忘记将此设置添加到/etc/sysctl.conf， 以便在重启后重新应用它。

有时候内核会无法立即分配想要数量的大页面，所以可能有必要重复该命令或者重新启动。（在重新启动之后，应立即将大部分机器的内存转换为大页面。） 要验证巨大的页面分配情况，请使用：

```
$ grep Huge /proc/meminfo
```

可能还需要赋予数据库服务器的操作系统用户权限，让他能通过sysctl 设置vm.hugetlb_shm_group以使用大页面， 和/或赋予使用ulimit -l锁定内存的权限。

PostgreSQL中大页面的默认行为是 尽可能使用它们并且在失败时转回到正常页面。要强制使用大页面，你可 以在postgresql.conf中把huge_pages设置成 on。注意此设置下如果没有足够的大页面可用， PostgreSQL将会启动失败。

Linux大页面特性的详细描述可见https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

18.5. 关闭服务器

有几种关闭数据库服务器的方法。通过给postgres进程发送不同的信号，你就可以控制关闭类型。

SIGTERM

这是智能关闭模式。在接收SIGTERM后， 服务器将不允许新连接，但是会让现有的会话正常结束它们的工作。仅当所有的会话终止后它才关闭。 如果服务器处在线备份模式，它将等待直到在线备份模式不再被激活。当在线备份模式被激活时， 仍然允许新的连接，但是只能是超级用户的连接（这一例外允许超级用户连接来终止在线备份模式）。如果服务器在恢复时请求智能关闭，恢复和流复制只有在所有正常会话都终止后才停止。

SIGINT

这是快速关闭模式。服务器不再允许新的连接，并向所有现有服务器进程发送SIGTERM，让它们中断当前事务并立刻退出。然后服务器等待所有服务器进程退出并最终关闭。 如果服务处于在线备份模式，备份模式将被终止并致使备份无用。

SIGQUIT

这是立即关闭模式。服务器将给所有子进程发送 SIGQUIT并且等待它们终止。如果有任何进程没有在 5 秒内终止，它们将被发送 SIGKILL。主服务器进程将在所有子进程退出之后立刻退出，而无需做普通的数据库关闭处理。这将导致在下一次启动时（通过重放 WAL 日志）恢复。只在紧急 时才推荐这种方式。

pg_ctl程序提供了一个发送这些信号关闭服务器的方便的接口。 另外，你在非 Windows 系统上可以用kill直接发送这些信号。可以用ps程序或者从数据目录的postmaster.pid文件中找到postgres进程的PID。例如，要做一次快速关闭：

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

> **重要**
>
> 最好不要使用SIGKILL关闭服务器。 这样做将会阻止服务器释放共享内存和信号量，那么在开始一个新的服务器之前，可能需要手动完成这些释放。 此

外，使用SIGKILL杀掉postgres进程时，postgres不会有机会将信号传播到它的子进程，所以也必须手工杀掉单个的子进程。

要终止单个会话同时允许其他会话继续，使用pg_terminate_backend()（参阅表 9.78）或发送SIGTERM信号到该会话相关的子进程。

18.6. 升级一个PostgreSQL集簇

本节讨论如何把你的数据库数据从一个PostgreSQL发行升级到一个更新的发行。

当前PostgreSQL版本号由主要版本号和次要版本号组成。 例如，在版本号10.1中，10是主要版本号，1是次要版本号，这意味着这将是主版本10的第一个次要版本。 对于PostgreSQL版本10.0之前的版本，版本号由三个数字组成，例如9.5.3。 在这些情况下，主要版本由版本号的前两个数字组（例如9.5）组成，次要版本是第三个数字， 例如3，这意味着这将是主要版本9.5的第三次要版本。

次要发行从来不改变内部存储格式并且总是向前并向后兼容同一主版本号中的次要发行。例如版本10.1与版本10.0和版本10.6兼容。类似的，例如9.5.3与9.5.0、9.5.1和9.5.6兼容。 要在兼容的版本间升级，你只需要简单地在服务器关闭时替换可执行文件并重启服务器。 数据目录则保持不变 — 次要升级就这么简单。

对于PostgreSQL的主发行， 内部数据存储格式常被改变，这使升级复杂化。传统的把数据移动到 新主版本的方法是先转储然后重新载入到数据库，不过这可能会很慢。 一种更快的方式是pg_upgrade。如下文所讨论的， 复制方法也能被用于升级。

新的主版本也通常会引入一些用户可见的不兼容性，因此可能需要应用程序编程上的改变。所有用户可见的更改都被列在发行注记（附录 E 中，请特别注意标有 "Migration" 的小节。如果你正在跨越几个主版本升级，一定要阅读每个中间版本的发行注记。

小心的用户在完全切换过去之前将希望在新版本上测试他们的客户端应用。因此，建立一个新旧版本的并存安装通常是一个好主意。在测试一个PostgreSQL主要升级时，考虑下列可能的改变类别：

管理

用于管理员监控和控制服务器的功能在每一个主发行中经常会改变和增加。

SQL

通常这包括新的 SQL 命令功能并且在行为上没有更改，除非在发行注记中有特别提到。

库 API

通常libpq等库值增加新功能，除非在发行注记中有特别提到。

系统目录

系统目录改变通常只影响数据库管理工具。

服务器 C-语言 API

这涉及到后端函数 API 中的改变，它使用 C 编程语言编写。这些改变影响引用服务器内部后端函数的代码。

18.6.1. 通过pg_dumpall升级数据

一种升级方法是从PostgreSQL的一个主版本转储数据并将它重新载入到另一个主版本中 — 要这样做，你必须使用pg_dumpall这样的逻辑备份工具，文件系统级别的备份方法将不会有

用（这也阻止你在一个不兼容版本的PostgreSQL中使用一个数据目录，因此在一个数据目录上尝试启动一个错误的服务器版本不会造成很大的危害）。

我们推荐你从较新版本的PostgreSQL中使用pg_dump和pg_dumpall程序，这样可以利用在这些程序中可能存在的改进。当前发行的转储程序可以读取任何 7.0 以上版本服务器中的数据。

这些指令假定你现有的安装位于/usr/local/pgsql目录，并且数据区域在/usr/local/pgsql/data。请用你的路径进行适当的替换。

1. 如果在创建一个备份，确认你的数据库没有在被更新。这不会影响备份的完整性，但是那些更改当然不会被包括在备份中。如果必要，编辑/usr/local/pgsql/data/pg_hba.conf文件中的权限（或等效的方法）来不允许除你之外的任何人使用数据库。关于访问控制的额外信息请见第 20 章

   要备份你的数据库安装，键入：

   ```
   pg_dumpall > outputfile
   ```

   要制作备份，你可以使用你正在运行版本的pg_dumpall命令，详见第 25.1.2 节但是，要得到最好的结果，试试使用PostgreSQL 11.2 的pg_dumpall命令，因为这个版本包含了对旧版本的缺陷修复和改进。虽然这个建议可能看起来很奇怪，因为你还没有安装新版本，但如果你计划平行地安装新版本，遵循这个建议是很明智的。在这种情况下，你可以正常完成安装并且稍后再来传输数据。这也将减少停机时间。

2. 关闭旧服务器：

   ```
   pg_ctl stop
   ```

   在那些自动启动PostgreSQL的系统上，可能有一个启动文件将完成同样的事情。例如，在一个Red Hat Linux系统中，我们会发现这也能用：

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   关于启动和停止服务器的细节请见第 18 章

3. 如果从备份恢复，重命名或删除旧的安装目录（如果它不是针对特定版本的）。重命名该目录是一个好主意，而不是删除它，因为如果你碰到问题并需要返回到它，它还存在。记住该目录可能消耗可观的磁盘空间。要重命名该目录，使用类似的命令：

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   （注意将该目录作为一个单一单元移动，这样相对路径可以保持不变）。

4. 安装新版本的PostgreSQL在第 16.4 节

5. 如果需要，创建一个新的数据库集簇。记住你必须在登录到一个特殊的数据库用户账户（如果你在升级，你就已经有了这个账户）时执行这些命令。

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. 恢复你之前的pg_hba.conf以及任何postgresql.conf修改。

7. 启动数据库服务器，也要使用特殊的数据库用户账户：

   ```
   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
   ```

8. 最后，使用新的 psql从备份恢复你的数据：

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

通过在一个不同的目录中安装新的服务器并且并行地在不同的端口运行新旧两个服务器可以达到最低的停机时间。那么你可以这样用：

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

来转移你的数据。

18.6.2. 通过pg_upgrade升级数据

pg_upgrade模块允许一个安装从一个 PostgreSQL主版本“就地”升级成另一个主版本。 升级可以在数分钟内被执行，特别是使用--link模式时。它 要求和上面的pg_dumpall相似的步骤，例如启动/停止 服务器、运行initdb。pg_upgrade 文档概述了所需的步骤。

18.6.3. 通过复制升级数据

也可以用PostgreSQL的已更新版本逻辑复制来创建一个~ 后备服务器，逻辑复制支持在不同主版本的PostgreSQL之间~ 的复制。后备服务器可以在同一台计算机或者不同的计算机上。一旦它和主服务器（运行旧版本的PostgreSQL）同步好，你可以切换主机并且将后备服~ 务器作为主机，然后关闭旧的数据库实例。这样一种切换使得一次升级的停机时间只有数秒。

这种升级方法可以用内置的逻辑复制工具和外部的逻辑复制系统如
pglogical，Slony，Londiste，和Bucardo。

18.7. 阻止服务器欺骗

服务器在运行时，它不可能让恶意用户取代正常的数据库服务器。然而，当服务器关闭时，一个本地用户可以通过启动它们自己的服务器来欺骗正常的服务器。行骗的服务器可以读取客户端发送的密码和查询语句， 但是不会返回任何数据，因为PGDATA这个目录是安全的（它有目录权限）。 欺骗是可能的，因为任何用户都可以启动一个数据库服务器；客户端无法识别一个无效的服务器，除非它被专门配置。

一种阻止local连接欺骗的方法是使用一个 Unix 域套接字目录（unix_socket_directories），该目录只对一个被信任的本地用户有写权限。 这可以防止恶意用户在该目录中创建自己的套接字文件。如果你担心有些应用程序可能仍然引用/tmp下的套接字文件并且因此容易受到欺骗，可在操作系统启动时创建一个符号链接/tmp/.s.PGSQL.5432指向一个被重定位的套接字文件。你也可能需要修改/tmp清除脚本防止删除这个符号链接。

local连接的另一个选项是对客户端使用requirepeer指定所需的连接到该套接字的服务器进程的拥有者。

要在TCP连接上防止欺骗，最好的解决方案是使用 SSL 证书，并且确保客户检查服务器的证书。 要做到这点，服务器必须配置为仅接受hostssl连接（第 20.1 节，并且有 SSL 密钥和证书文件（第 18.9 节。 TCP 客户端连接必须使用sslmode=verify-ca或verify-full进行连接，并且安装有适当的根证书文件第 34.18.1 节。

18.8. 加密选项

PostgreSQL提供了几个不同级别的加密， 并且在保护数据不会因为数据库服务器偷窃、不道德的管理员、不安全网络等因素而泄漏方面 提供很高的灵活性。加密可能也是保护一些诸如医疗记录或财务交易等敏感数据所要求的。

口令加密

数据库用户的口令都是以哈希(取决于password_encryption配置)的方式存储， 所以管理员不能限定实际的口令赋予用户。如果 SCRAM 或者 MD5 加密算法被用于客户端认证， 那么未加密的口令甚至都不可能出现在服务器上，因为客户端在通过网络发送口令之前，就已经加密过。推荐使用SCRAM，因为它是互联网标准而且相比于PostgreSQL特定的MD5认证协议更安全。

指定列加密

pgcrypto模块允许对特定域进行加密存储。这个功能只对某些敏感数据有用。 客户端提供解密的密钥，然后数据在服务器端解密并发送给客户端。

在数据被解密和在服务器与客户端之间传递时，解密数据和解密密钥将会在服务器端存在短暂的一段时间。 这就给那些能完全访问数据库服务器的人提供了一个短暂的截获密钥和数据的时间，例如系统管理员。

数据分区加密

存储加密可以在文件系统层面或者块层面上执行。Linux 文件系统加密 选项包括 eCryptfs 和 EncFS，而 FreeBSD 使用 PEFS。快层面或者全 盘加密选项包括 Linux 上的 dm-crypt + LUKS 以及 FreeBSD 上的 GEOM 模块 geli 及 gbde。很多其他操作系统也支持这个功能，包括 Windows。

这个机制避免了在整个计算机或者驱动器被盗的情况下， 未加密的数据被从驱动器中读取。它无法防止在文件系统被挂 载时的攻击，因为在挂载之后，操作系统提供数据的解密视图。不过，要想挂载该文件系统，你需要有一些方法把加密密钥传递给操作 系统，并且有时候这个密钥就存储在挂载该磁盘的主机上的某处。

跨网络加密数据

SSL 连接加密所有跨网络发送的数据：口令、查询以及返回的数据。pg_hba.conf文件允许管理员指定哪些主机可以使用 非加密连接（host），以及哪些主机需要使用 SSL 加密的连接（hostssl）。客户端还可以指定它们只通过 SSL 连接到服务器。我们还可以使用Stunnel或SSH加密传输。

SSL 主机认证

客户端和主机都可以提供 SSL 证书给对方。这在两边都需要一些额外的配置， 但是这种方式提供了比仅使用口令更强的身份验证。 它避免一个计算机伪装成服务器，这个时长只要足够读取客户端发送的口令就行了。它还避免了 “中间人”攻击，在其中有一台计算机处于客户端和服务器之间并伪装成服务器读取和传递两者之间的所有数据。

客户端加密

如果服务器所在机器的系统管理员是不可信的，那么客户端加密数据也是必要的。在这种情况下，未加密的数据从来不会在数据库服务器上出现。数据在发送给服务器之前加密，而数据库结果在能使用之前必须在客户端上解密。

18.9. 用 SSL 进行安全的 TCP/IP 连接

PostgreSQL 有一个对使用 SSL 连接加密客户端/服务器通讯的本地支持，它可以增加安全性。这个特性要求在客户端和服务器端都安装 OpenSSL 并且在编译 PostgreSQL 的时候打开这个支持（见第 16 章。

18.9.1. Basic Setup

当SSL支持被编译在PostgreSQL中时，可以通过将postgresql.conf中的 ssl设置为on让PostgreSQL服务器带着SSL支持被启动。 服务器在同一个 TCP 端口监听普通连接和SSL连

接，并且将与任何正在连接的客户端协商是否使用SSL。默认情况下，这是客户端的选项，关于如何设置服务器来要求某些或者所有连接使用SSL请见第 20.1 节

要SSL模式中启动服务器，包含服务器证书和私钥的文件必须存在。默认情况下，这些文件应该分别被命名为server.crt和server.key并且被放在服务器的数据目录中，但是可以通过配置参数ssl_cert_file和ssl_key_file指定其他名称和位置。

在 Unix 系统上，server.key上的权限必须不允许所有人或组的任何访问，通过命令chmod 0600 server.key可以做到。或者，该文件可以由root 所拥有并且具有组读访问（也就是0640权限）。这种设置适用于由操作系统管理证书和密钥文件的安装。用于运行PostgreSQL服务器的用户应该被作为能够访问那些证书和密钥文件的组成员。

如果数据目录允许组读取访问，则证书文件可能需要位于数据目录之外，以符合上面概述的安全要求。通常，启用组访问权限是为了允许非特权用户备份数据库，在这种情况下，备份软件将无法读取证书文件，并且可能会出错。

如果私钥被一个密码保护着，服务器将提示要求这个密码，并且在它被输入前不会启动。使用密码还会禁用在不重启服务器的情况下更改服务器的SSL配置的功能。 此外，密码保护的私钥在Windows上根本无法使用。

server.crt中的第一个证书必须是服务器的证书，因为它必须与服务器的私钥匹配。“intermediate”的证书颁发机构，也可以追加到文件。假设根证书和中间证书是使用v3_ca扩展名创建的，那么这样做避免了在客户端上存储中间证书的必要。这使得中间证书更容易到期。

无需将根证书添加到中server.crt。相反，客户端必须具有服务器证书链的根证书。

18.9.2. OpenSSL配置

PostgreSQL读取系统范围的OpenSSL配置文件。默认情况下，该文件被命名为openssl.cnf并位于openssl version -d报告的目录中。通过将环境变量设置OPENSSL_CONF为所需配置文件的名称，可以覆盖此默认值。

OpenSSL支持各种强度不同的密码和身份验证算法。虽然许多密码可以在OpenSSL的配置文件中被指定，您可以通过修改postgresql.conf配置文件中指定专门针对数据库服务器使用密码的ssl_ciphers 配置。

> **注意**
>
> 使用NULL-SHA或NULL-MD5可以得到身份验~ 证但没有加密开销。不过，中间人能够读取和传递客户端和服务器之间的通信。此外，加~ 密开销相比身份认证的开销是最小的。出于这些原因，我们建议不要使用 NULL 密码。

18.9.3. 使用客户端证书

要求客户端提供受信任的证书，把你信任的根证书颁发机构（CA）的证书放置在数据目录文件中。并且修改postgresql.conf中的参数ssl_ca_file到新的文件名，还要把认证选项clientcert=1加入到pg_hba.conf文件中合适的hostssl行上。然后将在 SSL 连接启动时从客户端请求该证书（一段对于如何在客户端设置证书的描述请见第 34.18 节。服务器将验证客户端的证书是由受信任的证书颁发机构之一签名。

如果希望避免将链接到现有根证书的中间证书显示在ssl_ca_file文件中（假设根证书和中间证书是使用 v3_ca 扩展名创建的），则这些证书也可以显示在ssl_ca_file 文件中。如果参数ssl_crl_file被设置，证书撤销列表（CRL）项也要被检查（显示 SSL 证书用法的图标见http://h41379.www4.hpe.com/doc/83final/ba554_90007/ch04s02.html）。

clientcert认证选项适用于所有的认证方法，但仅适用于pg_hba.conf中用hostssl指定的行。 当clientcert没有指定或设置为 0时，如果配置了 CA 文件，服务器将仍然会根据它验证任何提交的客户端证书 — 但是它将不会坚持要求出示一个客户端证书。

如果你在设置客户端证书，你可能希望用cert认证方法，这样证书控制用户认证以及提供连接安全。详见第 20.12 节在使用cert认证方法时，没有必要显式地指定clientcert=1）。

18.9.4. SSL 服务器文件用法

表 18.2总结了与服务器上 SSL 配置有关的文件（显示的文件名是默认的名称。本地配置的名称可能会不同）。

表 18.2. SSL 服务器文件用法

文件	内容	效果
ssl_cert_file ($PGDATA/server.crt)	服务器证书	发送给客户端来说明服务器的身份
ssl_key_file ($PGDATA/server.key)	服务器私钥	证明服务器证书是其所有者发送的，并不说明证书所有者是值得信任的
ssl_ca_file	可信的证书颁发机构	检查客户端证书是由一个可信的证书颁发机构签名的
ssl_crl_file	被证书授权机构撤销的证书	客户端证书不能出现在这个列表上

服务器在服务器启动时以及服务器配置重新加载时读取这些文件。在Windows系统上，只要为新客户端连接生成新的后端进程，它们也会重新读取。

如果在服务器启动时检测到这些文件中的错误，服务器将拒绝启动。但是，如果在配置重新加载过程中检测到错误，则会忽略这些文件，并继续使用旧的SSL配置。在Windows系统上，如果在后端启动时检测到这些文件中存在错误，则该后端将无法建立SSL连接。在所有这些情况下，错误情况都会在服务器日志中报告。

18.9.5. 创建证书

要为服务器创建一个有效期为365天的简单自签名证书，可以使用下面的OpenSSL命令，将dbhost.yourdomain.com替换为服务器的主机名：

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

然后执行：

```
chmod og-rwx server.key
```

如果文件的权限比这个更自由，服务器将拒绝该文件。要了解更多关于如何创建你的服务器私钥和证书的细节， 请参考OpenSSL文档。

尽管可以使用自签名证书进行测试，但是在生产中应该使用由证书颁发机构（CA）（通常是企业范围的根CA）签名的证书。

要创建其身份可以被客户端验证的服务器证书，请首先创建一个证书签名请求（CSR）和一个公共/专用密钥文件：

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

然后，使用密钥对请求进行签名以创建根证书颁发机构（使用Linux上的默认OpenSSL配置文件位置）：

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

最后，创建由新的根证书颁发机构签名的服务器证书：

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

server.crt和server.key应该存储在服务器上，并且root.crt应该存储在客户端上，以便客户端可以验证服务器的叶证书已由其受信任的根证书签名。root.key应该离线存储以用于创建将来的证书。

也可以创建一个包括中间证书的信任链：

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

server.crt和intermediate.crt应连接成一个证书文件包中并存储在服务器上。server.key还应该存储在服务器上。root.crt应将其存储在客户端上，以便客户端可以验证服务器的叶证书是否已由链接到其受信任根证书的证书链签名。root.key和intermediate.key应离线存储以用于创建将来的证书。

18.10. 使用SSH隧道的安全 TCP/IP 连接

可以使用SSH来加密客户端和PostgreSQL服务器之间的网络连接。如果处理得当，这将提供一个足够安全的网络连接，即使是对那些无 SSL 能力的客户端。

首先确认在PostgreSQL服务器的同一台机器上正确运行着一个SSH服务器，并且你可以使用ssh作为某个用户登入。然后你可以从客户端机器采用下面这种形式的命令建立一个安全的隧道：

```
ssh -L 63333:localhost:5432 joe@foo.com
```

-L参数中的第一个数（63333）是隧道在你那一端的端口号，它可以是任意未用过的端口（IANA 把端口 49152 到 65535 保留为个人使用）。第二个数（5432）是隧道的远端：你的服务器所使用的端口号。在端口号之间的名字或 IP 地址是你准备连接的数据库服务器的主机，至于你是从哪个主机登入的，在这个例子中则由foo.com表示。为了使用这个隧道连接到数据库服务器，你在本地机器上连接到端口 63333：

```
psql -h localhost -p 63333 postgres
```

对于数据库服务器，在这个环境中它将把你看做是连接到localhost的主机foo.com上的真实用户joe，并且它会使用被配置用于来自这个用户和主机的连接的认证过程。注意服务器将不会认为连接是 SSL 加密的，因为事实上SSH服务器和PostgreSQL服务器之间没有加密。只要它们在同一台机器上，这就不会造成任何额外的安全风险。

为了让隧道设置成功，你必须允许通过ssh作为joe@foo.com连接，就像你已经尝试使用ssh来创建一个终端会话。

你应当也已经设定好了端口转发：

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

但是数据库服务器则将会看到连接从它的foo.com接口进来，它没有被默认设置listen_addresses = 'localhost'所打开。这通常不是你想要的。

如果你必须通过某个登录主机“跳”到数据库服务器，一个可能的设置看起来像：

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

注意这种从shell.foo.com到db.foo.com的连接的方法将不会被 SSH 隧道加密。当网络被限制于各种方法时，SSH 提供了相当多的配置可能性。详情请参考 SSH 的文档。

> **提示**
>
> 一些其他的应用可以提供安全隧道，它们使用和刚刚描述的 SSH 概念上相似的过程。

18.11. 在Windows上注册事件日志

要为操作系统注册一个Windows 事件日志库，发出这个命令：

```
regsvr32 pgsql_library_directory/pgevent.dll
```

这会创建被事件查看器使用的注册表项，默认事件源命名为PostgreSQL。

要指定一个不同的事件源名称（见event_source）。使用/n和/i选项：

```
regsvr32 /n /i:event_source_name pgsql_library_directory/pgevent.dll
```

要从操作系统反注册事件日志库，发出这个命令：

```
regsvr32 /u [/i:event_source_name] pgsql_library_directory/pgevent.dll
```

> **注意**
>
> 要启用数据库服务器中的事件日志，在postgresql.conf中修改log_destination来包括eventlog。

第 19 章 服务器配置

有很多配置参数可以影响数据库系统的行为。本章的第一节中我们将描述一下如何与配置参数交互。 后续的小节将详细地讨论每一个参数。

19.1. 设置参数

19.1.1. 参数名称和值

所有参数名都是大小写不敏感的。每个参数都可以接受五种类型之一的值： 布尔、字符串、整数、 浮点数或枚举。该类型决定了设置该参数的语法：

- 布尔: 值可以被写成 on, off, true, false, yes, no, 1, 0 （都是大小写不敏感的）或者这些值的任何无歧义前缀。
- 字符串: 通常值被包括在单引号内，值内部的任何单引号都需要被双写。不过，如果值是一个简单数字或者 标识符，引号通常可以被省略。
- 数字（整数和浮点）: 只对浮点参数允许一个小数点。不要使用千位分隔符。不要求引号。
- 带单位的数字: 一些数字参数具有隐含单位，因为它们描述内存或时间量。单位可能是字节、千字节、块（通常是 8KB）、 毫秒、秒或分钟。这些设置之一的一个未修饰的数字值将使用该设置的默认单位，默认单位可以通 过引用pg_settings.unit来找到。为了方便，也可以 显式地指定一个不同的单位，例如时间值可以是'120 ms'，并且它们将被转换到参数的实际单位。要使用这个特性，注意值必须被写成一个字符 串（带有引号）。单位名称是大小写敏感的，并且在数字值和单位之间可以有空白。
  - 可用的内存单位是B（字节）、kB（千字节）、MB（兆字节）和GB（吉字节）。内存单位的乘数是 1024 而不是 1000。
  - 可用的时间单位是ms（毫秒）、s（秒）、min（分钟）、 h（小时）和d（天）。
- 枚举: 枚举类型的参数以与字符串参数相同的方式指定，但被限制到一组有限的值。 这样一个参数可用的值可以在pg_settings.enumvals 中找到。枚举参数值是大小写无关的。

19.1.2. 通过配置文件影响参数

设置这些参数最基本的方法是编辑postgresql.conf文件， 它通常被保存在数据目录中（当数据库集簇目录被初始化时，一个默认的拷贝将会被安装在那里）。一个该文件的例子看起来是：

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

每一行指定一个参数。名称和值之间的等号是可选的。空白是无意义的（除了在一个引号引用的参数值内）并且空行被忽略。井号（#）指示该行的剩余部分是一个注释。非简单标识符或者数字的参数值必须用单引号包围。要在参数值里嵌入单引号， 要么写两个单引号（首选）或者在引号前放反斜线。

以这种方式设定的参数为集簇提供了默认值。除非这些设置被覆盖，活动会话看到的就是这些设置。 下面的小节描述了管理员或用户覆盖这些默认值的方法。

主服务器进程每次收到SIGHUP信号（最简单的方法是从命令行运行pg_ctl reload或调用 SQL 函数pg_reload_conf()来发送这个信号）后都会重新读取这个配

置 文件。主服务器进程还会把这个信号传播给所有正在运行的服务器进程，这样现有的会话也能采用新 值（要等待它们完成当前正在执行的客户端命令之后才会发生）。另外，你可以直接向一个单一服务 器进程发送该信号。有些参数只能在服务器启动时设置，在配置文件中对这些条目的修改将被忽略， 直到下次服务器重启。配置文件中的非法参数设置也会在SIGHUP处理过程中被 忽略（但是会记录日志）。

除postgresql.conf之外，PostgreSQL 数据目录还包含一个文件 postgresql.auto.conf，它具有和postgresql.conf相同的格式但是不应该被手工编辑。这个 文件保存了通过ALTER SYSTEM命令提供的设置。每当postgresql.conf被读 取时这个文件会被自动读取，并且它的设置会以同样的方式生效。 postgresql.auto.conf中的设置会覆盖postgresql.conf 中的设置。

系统视图pg_file_settings 可以有助于对配置文件中的更改进行提前测试，或者在SIGHUP信号没有达到预期效果时用来诊断问题。

19.1.3. 通过SQL影响参数

PostgreSQL提供了三个SQL命令来建立配置默认值。 已经提到过的ALTER SYSTEM命令提供了一种改变全局默认值的从SQL可 访问的方法；它在功效上等效于编辑postgresql.conf。此外，还有两个命令 可以针对每个数据库或者每个角色设置默认值：

- ALTER DATABASE命令允许针对一个数据库覆盖其全局设置。
- ALTER ROLE命令允许用用户指定的值来覆盖全局设置和数据库设置。

只有当开始一个新的数据库会话时，用ALTER DATABASE和 ALTER ROLE设置的值才会被应用。它们会覆盖从配置文件或服务器命令行 获得的值，并且作为该会话后续的默认值。注意某些设置在服务器启动后不能被更改，并且因此 不能被这些命令（或者下文列举的命令）设置。

一旦一个客户端连接到数据库，PostgreSQL会提供两个额外的SQL命令（ 以及等效的函数）用以影响会话本地的配置设置：

- SHOW命令允许察看所有参数的当前值。对应的函数是 current_setting(setting_name text)。
- SET命令允许修改对于一个会话可以本地设置的参数的当前值， 它对其他会话没有影响。对应的函数是 set_config(setting_name, new_value, is_local)。

此外，系统视图pg_settings可以被用来查看和改变 会话本地的值：

- 查询这个视图与使用SHOW ALL相似，但是可以提供更多细节。它也更加灵活， 因为可以为它指定过滤条件或者把它与其他关系进行连接。
- 在这个视图上使用UPDATE并且指定更新setting 列，其效果等同于发出SET命令。例如，下面的命令

```
SET configuration_parameter TO DEFAULT;
```

等效于：

```
UPDATE pg_settings SET setting = reset_val WHERE name =
 'configuration_parameter';
```

19.1.4. 通过 Shell 影响参数

除了在数据库或者角色层面上设置全局默认值或者进行覆盖，你还可以通过 shell 工具把设置 传递给PostgreSQL。服务器和libpq 客户端库都能通过 shell 接受参数值。

- 在服务器启动期间，可以通过-c命令行参数把参数设置传递给 postgres命令。例如：

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```

  这种方式提供的设置会覆盖通过postgresql.conf或者 ALTER SYSTEM提供的设置，因此除了重启服务器之外无法从全局上改变它们。

- 当通过libpq启动一个客户端会话时，可以使用PGOPTIONS 环境变量指定参数设置。这种方式建立的设置构成了会话生存期间的默认值，但是不会影响 其他的会话。由于历史原因，PGOPTIONS的格式和启动 postgres命令时用到的相似，特别是-c标志必须被指定。 例如：

  ```
  env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
  ```

  通过 shell 或者其他方式，其他客户端和库可能提供它们自己的机制，以便允许用户在不直接 使用SQL命令的前提下修改会话设置。

19.1.5. 管理配置文件内容

PostgreSQL提供了一些特性用于把复杂的 postgresql.conf文件分解成子文件。在管理多个具有相关但不完全相同 配置的服务器时，这些特性特别有用。

除了单个参数设置，postgresql.conf文件可以包含包括指令，它指定要读入和处理的另一个文件，就好像该文件被插入到配置文件的这个点。这个特性允许一个配置文件被划分成物理上独立的部分。包括指令看起来像：

```
include 'filename'
```

如果文件名不是一个绝对路径，它将作为包含引用配置文件的目录的相对位置。包括可以被嵌套。

也有一个include_if_exists指令，它的作用和include指令一样，不过当被引用的文件不存在或者无法被读取时其行为不同。一个通常的include将认为这是一个错误情况，而include_if_exists仅仅记录一个消息并且继续处理引用配置文件。

postgresql.conf文件也可以包含include_dir指令，它指定要被包含的配置文件的一整个目录。它的用法类似：

```
include_dir 'directory'
```

非绝对目录名被当做包含引用配置文件的目录的相对路径。在该指定目录中，只有以后缀名.conf结尾的非目录文件才会被包括。以. 字符开头的文件名也会被忽略，因为在某些平台上它们是隐藏文件。一个包括目录中的多个文件 被以文件名顺序处理（根据 C 区域规则排序，即数字在字母之前并且大写字母在小写字母 之前）。

包括文件或目录可以被用来在逻辑上分隔数据库配置的各个部分，而不是用一个很大的postgresql.conf文件。考虑一个有两台数据库服务器的公司，每一个都有不同的内存量。很可能配置的元素都会被共享，例如用于日志的参数。但是两者关于内存的参数将会不同。并且还可能会有服务器相关的自定义。一种管理这类情况的方法是将你的站点的自定义配置修改分成三个文件。你可以把下面的内容加入到你的postgresql.conf文件末尾来包括它们：

```
include 'shared.conf'
```

```
include 'memory.conf'
include 'server.conf'
```

所有的系统将会有相同的shared.conf。每个有特定内存量的服务器可以共享相同的memory.conf。你可能对所有 8GB 内存的服务器有一个，而对那些 16GB 内存的服务器有另一个。并且最后server.conf可以装有真正服务器相关的配置信息。

另一中可能性是创建一个配置文件目录并把这个信息放到其中的文件里。例如，一个conf.d目录可以在postgresql.conf的末尾被引用：

```
include_dir 'conf.d'
```

然后你可以这样命名conf.d目录中的文件：

```
00shared.conf
01memory.conf
02server.conf
```

这种命名习惯建立了这些文件将被载入的清晰顺序。这是很重要的，因为在服务器读取配置文件时，对于一个特定的参数只有最后碰到的一个设置才会被使用。在这个例子中，conf.d/02server.conf设置的东西将会覆盖在 conf.d/01memory.conf中相同参数的值。

你还可以使用这种配置目录方法，在命名文件时更有描述性：

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

这种形式的安排为每个配置文件变体给定了一个唯一的名称。当多个服务器把它们的配置全部存储在一个位置（例如在一个版本控制仓库中）时，这可以帮助消除歧义（在版本控制下存储数据库配置文件是另一个值得考虑的好方法）。

19.2. 文件位置

除了已经提到过的postgresql.conf文件之外，PostgreSQL还使用另外两个手工编辑的配置文件，它们控制客户端认证（其使用在第 20 章讨论）。默认情况下，所有三个配置文件都存放在数据库集簇的数据目录中。 本节描述的参数允许配置文件放在别的地方（这么做可以简化管理，特别是如果配置文件被独立放置，可以很容易保证它得到恰当的备份）。

data_directory (string)

指定用于数据存储的目录。这个选项只能在服务器启动时设置。

config_file (string)

指定主服务器配置文件（通常叫postgresql.conf）。这个参数只能在postgres命令行上设置。

hba_file (string)

指定基于主机认证配置文件（通常叫pg_hba.conf）。这个参数只能在服务器启动的时候设置。

ident_file (string)

指定用于用户名称映射的配置文件（通常叫pg_ident.conf）。这个参数只能在服务器启动的时候设置。另见第 20.2 节

external_pid_file (string)

指定可被服务器创建的用于管理程序的额外进程 ID（PID）文件。这个参数只能在服务器启动的时候设置。

在默认安装中不会显式设置以上参数。相反，命令行参数-D或者环境变量PGDATA指定数据目录，并且上述配置文件都能在数据目录中找到。

如果你想把配置文件放在别的地方而不是数据目录中，那么postgres -D命令行选项或者环境变量PGDATA必须指向包含配置文件的目录，并且postgresql.conf中（或者命令行上）的data_directory参数必须显示数据目录实际存放的地方。请注意，data_directory将覆盖-D和PGDATA指定的数据目录位置，但是不覆盖配置文件的位置。

如果你愿意，可以使用选项config_file、hba_file和/或ident_file单独指定配置文件名称和位置。config_file只能在postgres命令行上指定，但是其他文件可以在主配置文件中设置。如果所有三个参数外加data_directory被显式地设置，则不必指定-D或PGDATA。

在设置任何这些参数时，相对路径将被解释为相对于postgres启动路径的路径。

19.3. 连接和认证

19.3.1. 连接设置

listen_addresses (string)

指定服务器在哪些 TCP/IP 地址上监听客户端连接。值的形式是一个逗号分隔的主机名和/或数字 IP 地址列表。特殊项*对应所有可用 IP 接口。项0.0.0.0允许监听所有 IPv4 地址并且::允许监听所有 IPv6 地址。如果列表为空，服务器将根本不会监听任何 IP 接口，在这种情况中只能使用 Unix 域套接字来连接它。默认值是localhost，它只允许建立本地 TCP/IP “环回”连接。虽然客户端认证（第 20 章允许细粒度地控制谁能访问服务器，listen_addresses控制哪些接口接受连接尝试，这能帮助在不安全网络接口上阻止重复的恶意连接请求。这个参数只能在服务器启动时设置。

port (integer)

服务器监听的 TCP 端口；默认是 5432 。请注意服务器会同一个端口号监听所有的 IP 地址。这个参数只能在服务器启动时设置。

max_connections (integer)

决定数据库的最大并发连接数。默认值通常是 100 个连接，但是如果内核设置不支持（initdb时决定），可能会比这个 数少。这个参数只能在服务器启动时设置。

当运行一个后备服务器时，你必须设置这个参数等于或大于主服务器上的参数。否则，后备服务器上可能无法允许查询。

superuser_reserved_connections (integer)

决定为PostgreSQL超级用户连接而保留的连接“槽”数。 同时活跃的并发连接最多max_connections个。任何时候，活跃的并发连接数最多为max_connections减去superuser_reserved_connections，新连接就只能由超级用户发起了，并且不会有新的复制连接被接受。

默认值是 3 。这个值必须小于max_connections减去max_wal_senders的值。 这个参数只能在服务器启动时设置。

unix_socket_directories (string)

指定服务器用于监听来自客户端应用的连接的 Unix 域套接字目录。通过列出用逗号分隔的多个目录可以建立多个套接字。项之间的空白被忽略，如果你需要在名字中包括空白或逗号，在目录名周围放上双引号。一个空值指定在任何 Unix 域套接字上都不监听，在这种情况中只能使用 TCP/IP 套接字来连接到服务器。默认值通常是/tmp，但是在编译时可以被改变。这个参数只能在服务器启动时设置。

除了套接字文件本身（名为.s.PGSQL.nnnn，其中nnnn是服务器的端口号），一个名为.s.PGSQL.nnnn.lock的普通文件会在每一个unix_socket_directories目录中被创建。任何一个都不应该被手工移除。

Windows下没有 Unix 域套接字，因此这个参数与 Windows 无关。

unix_socket_group (string)

设置 Unix 域套接字的所属组（套接字的所属用户总是启动服务器的用户）。可以与选项unix_socket_permissions一起用于对 Unix域连接进行访问控制。默认是一个空字符串，表示服务器用户的默认组。这个参数只能在服务器启动时设置。

Windows 下没有 Unix 域套接字，因此这个参数与 Windows 无关。

unix_socket_permissions (integer)

设置 Unix 域套接字的访问权限。Unix 域套接字使用普通的 Unix 文件系统权限集。这个参数值应该是数字的形式，也就是系统调用chmod和umask接受的 形式（如果使用自定义的八进制格式，数字必须以一个0（零）开头）。

默认的权限是0777，意思是任何人都可以连接。合理的候选是0770（只有用户和同组的人可以访问， 又见unix_socket_group）和0700（只有用户自己可以访问）（请注意，对于 Unix 域套接字，只有写权限有麻烦，因此没有对读取和执行权限的设置和收回）。

这个访问控制机制与第 20 章的用户认证没有关系。

这个参数只能在服务器启动时设置。

这个参数与完全忽略套接字权限的系统无关，尤其是自版本10以上的Solaris。 在那些系统上，可以通过把unix_socket_directories指向一个把搜索权限 限制给指定用户的目录来实现相似的效果。 因为 Windows 下没有 Unix 域套接字，因此这个参数也与 Windows 无关。

bonjour (boolean)

通过Bonjour广告服务器的存在。默认值是关闭。 这个参数只能在服务器启动时设置。

bonjour_name (string)

指定Bonjour服务名称。空字符串''（默认值）表示使用计算机名。 如果编译时没有打开Bonjour支持那么将忽略这个参数。这个参数只能在服务器启动时设置。

tcp_keepalives_idle (integer)

指定不活动多少秒之后通过 TCP 向客户端发送一个 keepalive 消息。 0 值表示使用默认值。这个参数只有在支持TCP_KEEPIDLE或等效套接字选项的系统或 Windows 上才可以使用。在其他系统上，它必须为零。在通过 Unix 域套接字连接的会话中，这个参数被忽略并且总是读作零。

> **注意**
>
> 在 Windows 上，值若为 0，系统会将该参数设置为 2 小时，因为 Windows 不支持读取系统默认值。

tcp_keepalives_interval (integer)

指定在多少秒之后重发一个还没有被客户端告知已收到的 TCP keepalive 消息。0 值表示使用系统默认值。这个参数只有在支持TCP_KEEPINTVL或等效套接字选项的系统或 Windows 上才可以使用。在其他系统上，必须为零。在通过 Unix域套接字连接的会话中，这个参数被忽略并总被读作零。

> **注意**
>
> 在 Windows 上，值若为 0，系统会将该参数设置为 1 秒，因为 Windows 不支持读取系统默认值。

tcp_keepalives_count (integer)

指定与客户端的服务器连接被认为死掉之前允许丢失的 TCP keepalive 数量。0 值表示使用系统默认值。这个参数只有在支持TCP_KEEPCNT或等效套接字选项的系统上才可以使用。在其他系统上，必须为零。在通过 Unix 域套接字连接的会话中，这个参数被忽略并总被读作零。

> **注意**
>
> Windows 不支持该参数，且必须为零。

19.3.2. 安全和认证

authentication_timeout (integer)

完成客户端认证的最长时间，以秒计。如果一个客户端没有在这段时间里完成 认证协议，服务器将关闭连接。这样就避免了出问题的客户端无限制地占有一个连接。默认值是 1分钟（1m）。这个参数只能在服务器命令行上或者在postgresql.conf文件中设置。

password_encryption (enum)

当在CREATE ROLE或者ALTER ROLE中指定了口令时，这个参数决定用于加密该口令的算法。默认值是md5，它会将口令存为一个MD5哈希（on也会被接受，它是md5的别名）。将这个参数设置为scram-sha-256将使用SCRAM-SHA-256来加密口令。

注意老的客户端可能缺少对SCRAM认证机制的支持，因此无法使用用SCRAM-SHA-256加密的口令。详情请参考第 20.5 节

krb_server_keyfile (string)

设置Kerberos服务器密钥文件的位置。详情请参考第 20.6 节这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

krb_caseins_users (boolean)

设置是否应该以大小写不敏感的方式对待GSSAPI用户名。默认值是off（大小写敏感）。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

db_user_namespace (boolean)

这个参数启用针对每个数据库的用户名。这个参数默认是关掉的。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

如果这个参数为打开，应该把用户创建成username@dbname的形式。当一个连接客户端传来username时，@和数据库名会被追加到用户名并且服务器会查找这个与数据库相关的用户名。注意在SQL环境中用含有@的名称创建用户时，需要把用户名放在引号内。

在这个参数被启用时，仍然可以创建平常的全局用户。而在客户端中指定这种用户时只需要简单地追加@，例如joe@。在服务器查找该用户名之前，@会被剥离掉。

db_user_namespace会导致客户端和服务器的用户名表达形式不同。认证检查总是会以服务器的用户名表达形式来完成，因此认证方法必须针对服务器用户名而不是客户端用户名来配置。由于md5方法在客户端和服务器两端都使用用户名作为salt，md5不能与db_user_namespace同时使用。

> **注意**
>
> 这种特性的目的是在找到完整的解决方案之前提供一种临时的措施。在找到完整解决方案时，这个选项将被去除。

19.3.3. SSL

有关设置SSL的更多信息请参考第 18.9 节

ssl (boolean)

启用SSL连接。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是off。

ssl_ca_file (string)

指定包含 SSL 服务器证书颁发机构（CA）的文件名。相对路径是相对于数据目录的。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值为空，表示没有载入CA文件，并且客户端证书验证没有被执行。

ssl_cert_file (string)

指定包含 SSL 服务器证书的文件名。相对路径是相对于数据目录的。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是server.crt。

ssl_crl_file (string)

指定包含 SSL 服务器证书撤销列表（CRL）的文件名。其中的相对路径是相对于数据目录的。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是空，表示没有载入CRL文件。

ssl_key_file (string)

指定包含 SSL 服务器私钥的文件名。相对路径是相对于数据目录。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是server.key。

ssl_ciphers (string)

指定一个SSL密码列表，用于安全连接。这个设置的语法和所支持的值列表可以 参见OpenSSL包中的 ciphers手册页。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是 HIGH:MEDIUM:+3DES:!aNULL。默认值通常是一种合理的选择，除非用户有特定的安全性需求。

默认值的解释：

HIGH

使用来自HIGH组的密码的密码组（例如 AES, Camellia, 3DES）

MEDIUM

使用来自MEDIUM组的密码的密码组（例如 RC4, SEED）

+3DES

OpenSSL 对HIGH的默认排序是有问题的，因为它认为 3DES 比 AES128 更高。这是错误的，因为 3DES 提供的安全性比 AES128 低，并且它也更加慢。 +3DES把它重新排序在所有其他HIGH和 MEDIUM密码之后。

!aNULL

禁用不做认证的匿名密码组。这类密码组容易收到中间人攻击，因此不应被使用。

可用的密码组细节可能会随着 OpenSSL 版本变化。可使用命令 openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'来查看 当前安装的OpenSSL版本的实际细节。注意这个列表是根据服务器密钥类型 在运行时过滤过的。

ssl_prefer_server_ciphers (boolean)

指定是否使用服务器的 SSL 密码首选项，而不是用客户端的。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是true。

老的PostgreSQL版本没有这个设置并且总是使用客户端的首选项。这个设置主要用于与那些版本 的向后兼容性。使用服务器的首选项通常会更好，因为服务器更可能会被合适地配置。

ssl_ecdh_curve (string)

指定用在ECDH密钥交换中的曲线名称。它需要被所有连接的客户端支持。 它不需要与服务器椭圆曲线密钥使用的曲线相同。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。默认值是prime256v1。

OpenSSL 命名了最常见的曲线： prime256v1 (NIST P-256)、 secp384r1 (NIST P-384)、secp521r1 (NIST P-521)。 openssl ecparam -list_curves命令可以显示可用曲线的完整列表。不过并不是所有的都在TLS中可用。

ssl_dh_params_file (string)

指定含有用于SSL密码的所谓临时DH家族的Diffie-Hellman参数的文件名。默认值为空，这种情况下将使用内置的默认DH参数。使用自定义的DH参数可以降低攻击者破解众所周知的内置DH参数的风险。可以用命令openssl dhparam -out dhparams.pem 2048创建自己的DH参数文件。

这个参数只能在 postgresql.conf文件中或服务器命令行上进行设置。

ssl_passphrase_command (string)

设置当需要一个密码（例如一个私钥）来解密SSL文件时会调用的一个外部命令。默认情况下，这个参数为空，表示使用内建的提示机制。

该命令必须将密码打印到标准输出并且以代码0退出。在该参数值中，%p被替换为一个提示字符串（要得到文字%，应该写成%%）。注意该提示字符串将可能含有空格，因此要确保加上适当的引号。如果输出的末尾有单一的新行，它会被剥离掉。

该命令实际上并不一定要提示用户输入一个密码。它可以从文件中读取密码、从钥匙链得到密码等等。确保选中的机制足够安全是用户的责任。

这个参数只能在 postgresql.conf文件中或服务器命令行上进行设置。

ssl_passphrase_command_supports_reload (boolean)

这个参数决定在配置重载期间如果一个密钥文件需要口令时，是否也调用ssl_passphrase_command设置的密码命令。如果这个参数为假（默认），那么在重载期间将忽略ssl_passphrase_command，如果在此期间需要密码则SSL配置将不会被重载。对于要求一个TTY（当服务器正在运行时可能是不可用的）来进行提示的命令，这种设置是合适的。例如，如果密码是从一个文件中得到的，将这个参数设置为真可能是合适的。

这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

19.4. 资源消耗

19.4.1. 内存

shared_buffers (integer)

设置数据库服务器将使用的共享内存缓冲区量。默认通常是 128 兆字节（128MB），但是如果你的内核设置不支持（在initdb时决定），那么可以会更少。这个设置必须至少为 128 千字节（BLCKSZ的非默认值将改变最小值）。不过为了更好的性能，通常会使用明显高于最小值的设置。

如果有一个专用的 1GB 或更多内存的数据库服务器，一个合理的shared_buffers开始值是系统内存的 25%。即使更大的shared_buffers有效，也会造成一些工作负载， 但因为PostgreSQL同样依赖操作系统的高速缓冲区，将shared_buffers设置为超过 40% 的RAM不太可能比一个小点值工作得更好。为了能把对写大量新的或改变的数据的处理分布在一个较长的时间段内，shared_buffers更大的设置通常要求对max_wal_size也做相应增加。

如果系统内存小于 1GB，一个较小的 RAM 百分数是合适的，这样可以为操作系统留下足够的空间。

huge_pages (enum)

控制是否为主共享内存区域请求巨型页。有效值是try（默认）、on以及off。如果huge_pages被设置为try，则服务器将尝试请求巨型页，但是如果失败会退回到默认的方式。如果为on，请求巨型页失败将使得服务器无法启动。如果为off，则不会请求巨型页。

当前，只有Linux和Windows上支持这个设置。在其他系统上这个参数被设置为try时，它会被忽略。

巨型页面的使用会导致更小的页面表以及花费在内存管理上的 CPU 时间更少，从而提高性能。更多有关Linux上使用巨型页面的细节请见第 18.4.5 节

巨型页在Windows上被称为大页面。要使用大页面，需要为运行PostgreSQL的Windows用户账号分配Lock Pages in Memory的用户权限。可以使用Windows的组策略工具（gpedit.msc）来分配用户权限Lock Pages in Memory。为了在命令窗口以单进程（而不是Windows服务）的方式启动数据库服务器，命令窗口必须以管理员身份运行或者禁用用户访问控制（UAC）。当UAC被启用时，普通的命令窗口会在启动时收回用户权限Lock Pages in Memory。

注意这种设置仅影响主共享内存区域。Linux、FreeBSD以及Illumos之类的操作系统也能为普通内存分配自动使用巨型页（也被称为“超级”页或者“大”页面），而不需要来自PostgreSQL的显式请求。在Linux上，这被称为“transparent huge pages”（THP，透明巨型页）。已知这种特性对某些Linux版本上的某些用户会导致PostgreSQL的性能退化，因此当前并不鼓励使用它（与huge_pages的显式使用不同）。

temp_buffers (integer)

设置每个数据库会话使用的临时缓冲区的最大数目。这些都是会话的本地缓冲区，只用于访问临时表。默认是 8 兆字节（8MB）。这个设置可以在独立的会话内部被改变，但是只有在会话第一次使用临时表之前才能改变； 在会话中随后企图改变该值是无效的。

一个会话将按照temp_buffers给出的限制根据需要分配临时缓冲区。如果在一个并不需要大量临时缓冲区的会话里设置一个大的数值， 其开销只是一个缓冲区描述符，或者说temp_buffers每增加一则增加大概 64 字节。不过，如果一个缓冲区被实际使用，那么它就会额外消耗 8192 字节（或者BLCKSZ字节）。

max_prepared_transactions (integer)

设置可以同时处于“prepared”状态的事务的最大数目（见PREPARE TRANSACTION）。把这个参数设置 为零（这是默认设置）将禁用预备事务特性。这个参数只能在服务器启动时设置。

如果你不打算使用预备事务，可以把这个参数设置为零来防止意外创建预备事务。如果你正在使用预备事务，你将希望把max_prepared_transactions至少设置为max_connections一样大，因此每一个会话可以有一个预备事务待处理。

当运行一个后备服务器时，这个参数必须至少与主服务器上的一样大。否则，后备服务器上将不会执行查询。

work_mem (integer)

指定在写到临时磁盘文件之前被内部排序操作和哈希表使用的内存量。该值默认为四兆字节（4MB）。注意对于一个复杂查询， 可能会并行运行好几个排序或者哈希操作；每个操作都会被允许使用这个参数指定的内存量，然后才会开始写数据到临时文件。同样，几个正在运行的会话可能并发进行这样的操作。因此被使用的总内存可能是work_mem值的好几倍，在选择这个值时一定要记住这一点。ORDER BY、DISTINCT和归并连接都要用到排序操作。哈希连接、基于哈希的聚集以及基于哈希的IN子查询处理中都要用到哈希表。

maintenance_work_mem (integer)

指定在维护性操作（例如VACUUM、CREATE INDEX和ALTER TABLE ADD FOREIGN KEY）中使用的 最大的内存量。其默认值是 64 兆字节（64MB）。因为在一个数据库会话中，一个时刻只有一个这样的操作可以被执行，并且一个数据库安装通常不会有太多这样的操作并发执行， 把这个数值设置得比work_mem大很多是安全的。 更大的设置可以改进清理和恢复数据库转储的性能。

注意当自动清理运行时，可能会分配最多达这个内存的autovacuum_max_workers倍，因此要小心不要把该默认值设置得太高。 通过独立地设置autovacuum_work_mem可能会对控制这种情况 有所帮助。

autovacuum_work_mem (integer)

指定每个自动清理工作者进程能使用的最大内存量。其默认值为 -1，表示转而使用maintenance_work_mem的值。当运行在其他上下文环境中时， 这个设置对VACUUM的行为没有影响。

max_stack_depth (integer)

指定服务器的执行堆栈的最大安全深度。这个参数的理想设置是由内核强制的实际栈尺寸限制（ulimit -s所设置的或者本地等价物），减去大约一兆字节的安全边缘。需要这个安全边缘是因为在服务器中并非所有例程都检查栈深度，只是在关键的可能递规的例程（例如表达式计算）中才进行检查。默认设置是两兆字节（2MB），这个值相对比较小并且不可能导致崩溃。但是，这个值可能太小了，以至于无法执行复杂的函数。只有超级用户可以修改这个设置。

把max_stack_depth参数设置得高于实际的内核限制将意味着一个失控的递归函数可能会导致一个独立的后端进程崩溃。 在PostgreSQL能够检测内核限制的平台上， 服务器将不允许把这个参数设置为一个不安全的值。不过，并非所有平台都能提供该信息，所以我们还是建议你在选择值时要小心。

dynamic_shared_memory_type (enum)

指定服务器应该使用的动态共享内存实现。可能的值是posix（用于使用 shm_open分配的 POSIX 共享内存）、sysv （用于通过shmget分配的 System V 共享内存）、windows（用于 Windows 共享内存）、mmap （使用存储在数据目录中的内存映射文件模拟共享内存）以及none（禁用 这个特性）。并非所有平台上都支持所有值，平台上第一个支持的选项就是其默认值。 在任何平台上mmap选项都不是默认值，通常不鼓励使用它，因为操作系统会 反复地把修改过的页面写回到磁盘上，从而增加了系统的I/O负载。不过当 pg_dynshmem目录被存储在一个 RAM 盘时或者没有其他共享内存功能可用时， 它还是有用的。

19.4.2. 磁盘

temp_file_limit (integer)

指定一个进程能用于临时文件（如排序和哈希临时文件，或者用于保持游标的存储文件）的最大磁盘空间量。一个试图超过这个限制的事务将被取消。这个值以千字节计，并且-1（默认值）意味着没有限制。只有超级用户能够修改这个设置。

这个设置约束着一个给定PostgreSQL进程在任何瞬间所使用的所有临时文件的总空间。应该注意的是，与在查询执行中在幕后使用的临时文件相反，显式临时表所用的磁盘空间不被这个设置所限制。

19.4.3. 内核资源使用

max_files_per_process (integer)

设置每个服务器子进程允许同时打开的最大文件数目。默认是 1000 个文件。如果内核强制一个安全的针对每个进程的限制，那么你不用操心这个设置。但是在 一些平台上（特别是大多数 BSD 系统），如果很多进程都尝试打开很多文件，内核将允许独立进程打开比个系统真正可以支持的数目大得多得文件数。如果你发现自己看到了“Too many open files”这样的失败，可尝试减小这个设置。这个参数只能在服务器启动时设置。

19.4.4. 基于代价的清理延迟

在VACUUM和ANALYZE命令的执行过程中，系统维持着一个内部计数器来跟踪各种被执行的I/O操作的估算开销。当累计的代价达到一个限制（由vacuum_cost_limit指定），执行这些操作的进程将按照vacuum_cost_delay所指定的休眠一小段时间。然后它将重置计数器并继续执行。

这个特性的出发点是允许管理员降低这些命令对并发的数据库活动产生的I/O影响。在很多情况下，VACUUM和ANALYZE等维护命令能否快速完成并不重要，而非常重要的是这些命令不会对系统执行其他数据库操作的能力产生显著的影响。基于代价的清理延迟提供了一种方式让管理员能够保证这一点。

对于手动发出的VACUUM命令，该特性默认被禁用。要启用它，只要把vacuum_cost_delay变量设为一个非零值。

vacuum_cost_delay (integer)

进程超过代价限制后将休眠的时间长度，以毫秒计。其默认值为0，这将禁用基于代价的清理延迟特性。正值将启用基于代价的清理。注意在很多系统上，实际的休眠延迟单位是10毫秒，将vacuum_cost_delay设置成不为10的倍数的值和将它设置为比该值大的10的倍数的效果相同。

在使用基于代价的清理时，vacuum_cost_delay的合适值通常很小，也许是10或20毫秒。调整清理时资源消耗最好的方法是调整其他清理代价参数。

vacuum_cost_page_hit (integer)

清理一个在共享缓存中找到的缓冲区的估计代价。它表示锁住缓冲池、查找共享哈希表和扫描页内容的代价。默认值为1。

vacuum_cost_page_miss (integer)

清理一个必须从磁盘上读取的缓冲区的代价。它表示锁住缓冲池、查找共享哈希表、从磁盘读取需要的块以及扫描其内容的代价。默认值为10。

vacuum_cost_page_dirty (integer)

当清理修改一个之前干净的块时需要花费的估计代价。它表示再次把脏块刷出到磁盘所需要的额外I/O。默认值为20。

vacuum_cost_limit (integer)

将导致清理进程休眠的累计代价。默认值为200。

> **注意**
>
> 有些操作会保持关键性的锁，这样可以尽快完成。基于代价的清理延迟在这类操作期间不会发生。因此有可能代价会累计至大大超过指定的限制。为了防止在这种情况下的无意义的长时间延迟，实际延迟的计算方式是vacuum_cost_delay * accumulated_balance / vacuum_cost_limit，且最大值是vacuum_cost_delay * 4。

19.4.5. 后台写入器

有一个独立的服务器进程，叫做后台写入器，它的功能就是发出写“脏”（新的或修改过的）共享缓冲区的命令。它写出共享缓冲区，这样让处理用户查询的服务器进程很少或者永不等待写动作的发生。不过，后台写入器确实会增加 I/O 的总负荷，因为虽然在每个检查点间隔中一个重复弄脏的页面可能只会写出一次，但在同一个间隔中后台写入器可能会把它写出好几次。在这一小节讨论的参数可以被用于调节本地需求的行为。

bgwriter_delay (integer)

指定后台写入器活动轮次之间的延迟。在每个轮次中，写入器都会为一定数量的脏缓冲区发出写操作（可以用下面的参数控制）。然后它就休眠 bgwriter_delay毫秒， 然后重复动作。默认值是 200 毫秒（200ms）。注意在许多系统上，休眠延迟的有效解析度是 10 毫秒；因此，为bgwriter_delay设置一个 不是 10 的倍数的值与把它设置为下一个更高的 10 的倍数是一样的效果。这个选项只能在服务器命令行上或者在postgresql.conf文件中设置。

bgwriter_lru_maxpages (integer)

在每个轮次中，不超过这么多个缓冲区将被后台写入器写出。把这个参数设置为零可禁用后台写出（注意被一个独立、专用辅助进程管理的检查点不受影响）。默认值是 100 个缓冲区。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

bgwriter_lru_multiplier (floating point)

每一轮次要写的脏缓冲区的数目基于最近几个轮次中服务器进程需要的新缓冲区的数目。 最近所需的平均值乘以bgwriter_lru_multiplier可以估算下一轮次中将会需要的缓冲区数目。脏缓冲区将被写出直到有很多干净可重用的缓冲区（然而，每一轮次中写出的缓冲区数不超过bgwriter_lru_maxpages）。 因此，设置为 1.0 表示一种“刚刚好

的”策略，这种策略会写出正好符合预测值的数目的缓冲区。 更大大的值可以为需求高峰提供某种缓冲，而更小的值则需要服务进程来处理一些写出操作。默认值是 2.0。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

bgwriter_flush_after (integer)

不管何时 bgwriter 写入了超过bgwriter_flush_after字节，尝试强制 OS 把这些写发送到底层存储上。这样做将限制内核页缓存中脏数据的量，降低了在检查点末尾发出一个 fsync 时或者 OS 在后台大批量写回数据时卡住的可能性。那常常会导致大幅度压缩的事务延迟，但是也有一些情况（特别是负载超过shared_buffers但小于 OS 页面高速缓存）的性能会降低。这种设置可能会在某些平台上没有效果。合法的范围在0（禁用受控写回）和2MB之间。Linux 上的默认值是512kB，其他平台上是0（如果BLCKSZ不是8kB，则默认值和最大值会按比例缩放至这个值）。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

较小的bgwriter_lru_maxpages和bgwriter_lru_multiplier可以降低由后台写入器造成的额外 I/O 开销。但更可能的是，服务器进程将必须自己发出写入操作，这会延迟交互式查询。

19.4.6. 异步行为

effective_io_concurrency (integer)

设置PostgreSQL可以同时被执行的并发磁盘 I/O 操作的数量。调高这个值，可以增加任何单个PostgreSQL会话试图并行发起的 I/O 操作的数目。 允许的范围是 1 到 1000，或 0 表示禁用异步 I/O 请求。当前这个设置仅影响位图堆扫描。

对于磁盘驱动器，这个设置的一个很好的出发点是组成一个被用于该数据库的 RAID 0 条带或 RAID 1 镜像的独立驱动器数量（对 RAID 5 而言，校验驱动器不计入）。但是， 如果数据库经常忙于在并发会话中发出的多个查询，较低的值可能足以使磁盘阵列繁忙。比保持磁盘繁忙所需的值更高的值只会造成额外的 CPU 开销。SSD 以及其他基于内存的存储常常能处理很多并发请求，因此它们的最佳值可能是数百。

异步 I/O 依赖于一个有效的posix_fadvise函数（一些操作系统可能没有）。 如果不存在这个函数，将这个参数设置为除 0 之外的任何东西将导致错误。在一些操作系统上（如Solaris）虽然提供了这个函数，但它不会做任何事情。

在支持的系统上默认值为 1，否则为 0。对于一个特定表空间中的表，可以通过设定该表空间的同名参数（见ALTER TABLESPACE）可以覆盖这个值。

max_worker_processes (integer)

设置系统能够支持的后台进程的最大数量。这个参数只能在服务器启动时设置。默认值为 8。

在运行一个后备服务器时，你必须把这个参数设置为等于或者高于主控服务器上的值。否则， 后备服务器上可能不会允许查询。

在更改这个值时，考虑也对max_parallel_workers、max_parallel_maintenance_workers以及max_parallel_workers_per_gather进行调整。

max_parallel_workers_per_gather (integer)

设置单个Gather或者Gather Merge节点能够开始的工作者的最大数量。并行工作者会从max_worker_processes建立的进程池中取得，数量由max_parallel_workers限制。注意所要求的工作者数量在运行时可能实际无法被满足。如果这种事情发生，该计划将会以比预期更少的工作者运行，这可能会不太高效。默认值是2。把这个值设置为 0（默认值）将会禁用并行查询执行。

注意并行查询可能消耗比非并行查询更多的资源，因为每一个工作者进程时一个完全独立的进程，它对系统产生的影响大致和一个额外的用户会话相同。在为这个设置选择值

时，以及配置其他控制资源利用的设置（例如work_mem）时，应该把这个因素考虑在内。work_mem之类的资源限制会被独立地应用于每一个工作者，这意味着所有进程的总资源利用可能会比单个进程时高得多。例如，一个使用 4 个工作者的并行查询使用的 CPU 时间、内存、I/O 带宽可能是不使用工作者时的 5 倍之多。

并行查询的更多信息请见第 15 章

max_parallel_maintenance_workers (integer)

设置单一工具性命令能够启动的并行工作者的最大数目。当前，唯一一种支持使用并行工作者的工具性命令是CREATE INDEX，并且只有在构建B-树索引时才能并行。并行工作者从由max_worker_processes创建的进程池中取出，数量由max_parallel_workers控制。注意实际在运行时所请求数量的工作者可能不可用。如果发生这种情况，工具性操作将使用比预期数量少的工作者运行。默认值为2。将这个值设置为0可以禁用工具性命令对并行工作者的使用。

注意并行工具性命令不应该消耗比同等数量非并行操作更多的内存。这种策略与并行查询不同，并行查询的资源限制通常是应用在每个工作者进程上。并行工具性命令把资源限制maintenance_work_mem当作对整个工具性命令的限制，而不管其中用到了多少个并行工作者进程。不过，并行工具性命令实际上可能仍会消耗更多的CPU资源和I/O带宽。

max_parallel_workers (integer)

设置系统为并行操作所支持的工作者的最大数量。默认值为8。在增加或者减小这个值时，也要考虑对max_parallel_maintenance_workers以及max_parallel_workers_per_gather进行调整。此外，要注意将这个值设置得大于max_worker_processes将不会产生效果，因为并行工作者进程都是从max_worker_processes所建立的工作者进程池中取出来的。

backend_flush_after (integer)

只要一个后端写入了超过backend_flush_after字节，就会尝试强制 OS 把这些写发送到底层存储。这样做将会限制内核页高速缓存中的脏数据数量，降低在检查点末尾发出fsync时或者 OS 在后台大批写回数据时卡住的可能性。这常常会导致极大降低的事务延迟，但是也有一些情况中（特别是负载超过shared_buffers但低于 OS 的页面高速缓存时），性能可能会下降。这个设置可能在某些平台上没有效果。合法的范围位于0（禁用受控写回）和2MB之间。默认是0（即没有强制写回）。（如果BLCKSZ不是8kB，最大值会按比例缩放到它）。

old_snapshot_threshold (integer)

设置在使用快照时，一个快照可以被使用而没有发生snapshot too old错误风险的最小时间。这个参数只能在服务器启动时设置。

如果超过该阈值，旧数据将被清理掉。这可以有助于阻止长时间使用的快照造成的快照膨胀。为了阻止由于本来对该快照可见的数据被清理导致的不正确结果，当快照比这个阈值更旧并且该快照被用来读取一个该快照建立以来被修改过的页面时，将会产生一个错误。

值为-1会禁用这个特性，并且这个值是默认值。对于生产工作有用的值可能从几个小时到几天。该设置将被转换成分钟粒度，并且小数字（例如0或者1min）被允许只是因为它们有时对于测试有用。虽然允许高达60d的设置，但是请注意很多负载情况下，很短的时间帧里就可能发生极大的膨胀或者事务 ID 回卷。

当这个特性被启用时，关系末尾的被清出的空间不能被释放给操作系统，因为那可能会移除用于检测snapshot too old情况所需的信息。所有分配给关系的空间还将与该关系关联在一起便于重用，除非它们被显式地释放（例如，用VACUUM FULL）。

这个设置不会尝试保证在任何特殊情况下都会生成错误。事实上，如果（例如）可以从一个已经物化了一个结果集的游标中生成正确的结果，即便被引用表中的底层行已经被

清理掉也不会生成错误。某些表不能被过早地安全清除，并且因此将不受这个设置的影响，例如系统目录。对于这些表，这个设置将不能降低膨胀，也不能降低在扫描时产生snapshot too old错误的可能性。

19.5. 预写式日志

参阅第 30.4 获取调节这些设置的额外信息。

19.5.1. 设置

wal_level (enum)

wal_level决定多少信息写入到 WAL 中。默认值是replica，它会写入足够的数据以支持WAL归档和复制，包括在后备服务器上运行只读查询。minimal会去掉除从崩溃或者立即关机中进行恢复所需的信息之外的所有记录。最后，logical会增加支持逻辑解码所需的信息。每个层次包括所有更低层次记录的信息。这个参数只能在服务器启动时设置。

在minimal级别中，某些批量操作的 WAL 日志可以被安全地跳过，这可以使那些操作更快（见第 14.4.7 节。这种优化可以应用的操作包括：

CREATE TABLE AS
CREATE INDEX
CLUSTER
COPY到在同一个事务中被创建或截断的表中

但最少的 WAL 不会包括足够的信息来从基础备份和 WAL 日志中重建数据，因此，要启用 WAL 归档（archive_mode）和流复制，必须使用replica或更高级别。

在logical层，与replica相同的信息会被记录，外加上 允许从 WAL 抽取逻辑修改集所需的信息。使用级别 logical将增加 WAL 容量，特别是如果为了REPLICA IDENTITY FULL配置了很多表并且执行了很多UPDATE和DELETE 语句时。

在 9.6 之前的版本中，这个参数也允许值archive和hot_standby。现在仍然接受这些值，但是它们会被映射到replica。

fsync (boolean)

如果打开这个参数，PostgreSQL服务器将尝试确保更新被物理地写入到磁盘，做法是发出fsync()系统调用或者使用多种等价的方法（见wal_sync_method）。这保证了数据库集簇在一次操作系统或者硬件崩溃后能恢复到一个一致的状态。

虽然关闭fsync常常可以得到性能上的收益，但当发生断电或系统崩溃时可能造成不可恢复的数据损坏。因此，只有在能很容易地从外部数据中重建整个数据库时才建议关闭fsync。

能安全关闭fsync的环境的例子包括从一个备份文件中初始加载一个新数据库集簇、使用一个数据库集簇来在数据库被删掉并重建之后处理一批数据，或者一个被经常重建并却不用于失效备援的只读数据库克隆。单独的高质量硬件不足以成为关闭fsync的理由。

当把fsync从关闭改成打开时，为了可靠的恢复，需要强制在内核中的所有被修改的缓冲区进入持久化存储。这可以在多个时机来完成：在集簇被关闭时或在 fsync 因为运行initdb --sync-only而打开时、运行sync时、卸载文件系统时或者重启服务器时。

在很多情况下，为不重要的事务关闭synchronous_commit可以提供很多关闭fsync的潜在性能收益，并不会有的同时， 关闭fsync可以提供很多潜在的性能优势，而不会有伴随着的数据损坏风险。

fsync只能在postgresql.conf文件中或在服务器命令行上设置。如果你关闭这个参数，请也考虑关闭full_page_writes。

synchronous_commit (enum)

指定在命令返回“success”指示给客户端之前，一个事务是否需要等待 WAL 记录被写入磁盘。合法的值是on、remote_apply、remote_write、local和off。默认的并且安全的设置是on。当设置为off时，在向客户端报告成功和真正保证事务不会被服务器崩溃威胁之间会有延迟（最大的延迟是wal_writer_delay的三倍）。不同于fsync，将这个参数设置为off不会产生数据库不一致性的风险：一个操作系统或数据库崩溃可能会造成一些最近据说已提交的事务丢失，但数据库状态是一致的，就像这些事务已经被干净地中止。因此，当性能比完全确保事务的持久性更重要时，关闭synchronous_commit可以作为一个有效的代替手段。更多讨论见第 30.3 节

如果synchronous_standby_names为非空，这个参数也控制事务提交是否将等待它们的 WAL 记录被复制到后备服务器上。当这个参数被设置为on时，直到来自于当前同步的后备服务器的回复指示它们已经收到了事务的提交记录并将其刷入了磁盘，主服务器上的事务才会提交。这保证事务将不会被丢失，除非主服务器和所有同步后备都遭受到了数据库存储损坏的问题。当被设置为remote_apply时，提交将会等待，直到来自当前的同步后备的回复指示它们已经收到了该事务的提交记录并且已经应用了该事务，这样该事务才变得对后备上的查询可见。当这个参数被设置为remote_write时，提交将等待，直到来自当前的同步后备的回复指示它们已经收到了该事务的提交记录并且已经把该记录写出到它们的操作系统，这种设置足以保证数据在后备服务器的PostgreSQL实例崩溃时得以保存，但是不能保证后备服务器遭受操作系统级别崩溃时数据能被保持，因为数据不一定必须要在后备机上达到稳定存储。最后，设置local会导致提交等待本地刷写到磁盘而不是复制完成。在使用同步复制时这通常不是我们想要的效果，但是为了完整性，还是提供了这样一个选项。

如果synchronous_standby_names为空，设置on、remote_apply、remote_write和local都提供了同样的同步级别：事务提交只等待本地刷写磁盘。

这个参数可以随时被修改；任何一个事务的行为由其提交时生效的设置决定。因此，可以同步提交一些事务，同时异步提交其他事务。例如，当默认是相反时，实现一个单一多语句事务的异步提交，在事务中发出SET LOCAL synchronous_commit TO OFF。

wal_sync_method (enum)

用来向强制 WAL 更新到磁盘的方法。如果fsync是关闭的，那么这个设置就不相关，因为 WAL 文件更新将根本不会被强制。可能的值是：

- open_datasync（用open()选项O_DSYNC写 WAL 文件）
- fdatasync（在每次提交时调用fdatasync()）
- fsync（在每次提交时调用fsync()）
- fsync_writethrough（在每次提交时调用fsync()，强制任何磁盘写高速缓存的直通写）
- open_sync（用open()选项O_SYNC写 WAL 文件）

open_* 选项也可以使用O_DIRECT（如果可用）。不是在所有平台上都能使用所有这些选择。默认值是列表中第一个被平台支持的那个， 不过fdatasync是 Linux 中的默认值。默认值不一定是最理想的；有可能需要修改这个设置或系统配置的其他方面来创建一个崩溃-安全的配置，或达到最佳性能。这些方面在第 30.1 节讨论。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

full_page_writes (boolean)

当这个参数为打开时，PostgreSQL服务器在一个检查点之后的页面的第一次修改期间将每个页面的全部内容写到 WAL 中。这么做是因为在操作系统崩溃期间正在处理的一次页写入可能只有部分完成，从而导致在一个磁盘页面中混合有新旧数据。在崩溃后的恢复期间，通常存储在 WAL 中的行级改变数据不足以完全恢复这样一个页面。存储完整的

页面映像可以保证页面被正确存储，但代价是增加了必须被写入 WAL 的数据量（因为 WAL 重放总是从一个检查点开始，所以在检查点后每个页面的第一次改变时这样做就够了。因此，一种减小全页面写开销的方法是增加检查点间隔参数值）。

把这个参数关闭会加快正常操作，但是在系统失败后可能导致不可恢复的数据损坏，或者静默的数据损坏。其风险类似于关闭fsync，但是风险较小。并且只有在可关闭fsync的情况下才应该关闭它。

关闭这个选项并不影响用于时间点恢复（PITR）的 WAL 归档使用（见第 25.3 节。

这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是on。

wal_log_hints (boolean)

当这个参数为on时，PostgreSQL服务器一个检查点之后页面被第一次修改期间把该磁盘页面的整个内容都写入 WAL，即使对所谓的提示位做非关键修改也会这样做。

如果启用了数据校验和，提示位更新总是会被 WAL 记录并且这个设置会被忽略。你可以使用这个 设置测试如果你的数据库启用了数据校验和，会有多少额外的 WAL 记录发生。

这个参数只能在服务器启动时设置。默认值是off。

wal_compression (boolean)

当这个参数为on时，如果full_page_writes 为打开或者处于基础备份期间，PostgreSQL 服务器 会压缩写入到 WAL 中的完整页面镜像。压缩页面镜像将在 WAL 重放时 被解压。默认值为off。只有超级用户可以更改这个设置。

打开这个参数可以减小 WAL 所占的空间且无需承受不可恢复的数据损坏风险， 但是代价是需要额外的 CPU 开销以便在 WAL 记录期间进行压缩以及在 WAL 重放时解压。

wal_buffers (integer)

用于还未写入磁盘的 WAL 数据的共享内存量。默认值 -1 选择等于shared_buffers的 1/32 的尺寸（大约3%），但是不小于64kB也不大于 WAL 段的尺寸（通常为）。如果自动的选择太大或太小可以手工设置该值，但是任何小于32kB的正值都将被当作32kB。这个参数只能在服务器启动时设置。

在每次事务提交时，WAL 缓冲区的内容被写出到磁盘，因此极大的值不可能提供显著的收益。不过，把这个值设置为几个兆字节可以在一个繁忙的服务器（其中很多客户端会在同一时间提交）上提高写性能。由默认设置 -1 选择的自动调节将在大部分情况下得到合理的结果。

wal_writer_delay (integer)

指定 WAL 写入器刷写 WAL 的频繁程度。在刷写 WAL 之后它会睡眠wal_writer_delay毫秒，除非被一个异步提交事务唤醒。假如上一次刷写发生在少于wal_writer_delay毫秒以前并且从上一次刷写发生以来产生了少于wal_writer_flush_after字节的 WAL，则WAL将只被写入到操作系统而不是被刷到磁盘。默认值是 200 毫秒（200ms）。注意在很多系统上，有效的睡眠延迟粒度是 10 毫秒，把wal_writer_delay设置为一个不是 10 的倍数的值，其效果和把它设置为大于该值的下一个 10 的倍数产生的效果相同。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

wal_writer_flush_after (integer)

指定 WAL 写入器刷写 WAL 的频繁程度。如果上一次刷写发生在少于wal_writer_delay毫秒以前并且从上一次刷写发生以来产生了少于wal_writer_flush_after字节的 WAL，则WAL将只被写入到操作系统而不是被刷到磁盘。如果wal_writer_flush_after被设置为0，则WAL数据会被立即刷写。默认是1MB。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

commit_delay (integer)

在一次 WAL 刷写被发起之前，commit_delay增加一个时间延迟，以微妙计。如果系统负载足够高，使得在一个给定间隔内有额外的事务准备好提交，那么通过允许更多事务通过一个单次 WAL 刷写来提交能够提高组提交的吞吐量。但是，它也把每次 WAL 刷写的潜伏期增加到了最多commit_delay微秒。因为如果没有其他事务准备好提交，就会浪费一次延迟，只有在当一次刷写将要被发起时有至少commit_siblings个其他活动事务时，才会执行一次延迟。另外，如果fsync被禁用，则将不会执行任何延迟。默认的commit_delay是零（无延迟）。只有超级用户才能修改这个设置。

在PostgreSQL的 9.3 发布之前，commit_delay的行为不同并且效果更差：它只影响提交，而不是所有 WAL 刷写，并且即使在 WAL 刷写马上就要完成时也会等待一整个配置的延迟。从PostgreSQL 9.3 中开始，第一个准备好刷写的进程会等待配置的间隔，而后续的进程只等到领先者完成刷写操作。

commit_siblings (integer)

在执行commit_delay延迟时，要求的并发活动事务的最小数目。大一些的值会导致在延迟间隔期间更可能有至少另外一个事务准备好提交。默认值是五个事务。

19.5.2. 检查点

checkpoint_timeout (integer)

自动 WAL 检查点之间的最长时间，以秒计。合理的范围在 30 秒到 1 天之间。默认是 5 分钟（5min）。增加这个参数的值会增加崩溃恢复所需的时间。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

checkpoint_completion_target (floating point)

指定检查点完成的目标，作为检查点之间总时间的一部分。默认是 0.5。 这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

checkpoint_flush_after (integer)

在执行检查点时，只要有checkpoint_flush_after字节被写入，就尝试强制 OS 把这些写发送到底层存储。这样做将会限制内核页面高速缓存中的脏数据数量，降低在检查点末尾发出fsync或者 OS 在后台大批量写回数据时被卡住的可能性。那常常会导致大幅度压缩的事务延迟，但是也有一些情况（特别是负载超过shared_buffers但小于 OS 页面高速缓存）的性能会降低。这种设置可能会在某些平台上没有效果。合法的范围在0（禁用强制写回）和2MB之间。Linux 上的默认值是256kB，其他平台上是0（如果BLCKSZ不是8kB，则默认值和最大值会按比例缩放到它）。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

checkpoint_warning (integer)

如果由于填充WAL段文件导致的检查点之间的间隔低于这个参数表示的秒数，那么就向服务器日志写一个消息（它建议增加max_wal_size的值）。默认值是 30 秒（30s）。零则关闭警告。如果checkpoint_timeout低于checkpoint_warning，则不会有警告产生。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

max_wal_size (integer)

在自动 WAL 检查点之间允许 WAL 增长到的最大尺寸。这是一个软限制， 在特殊的情况下 WAL 尺寸可能会超过max_wal_size， 例如在重度负荷下、archive_command失败或者高的 wal_keep_segments设置。默认为 1 GB。增加这个参数 可能导致崩溃恢复所需的时间。这个参数只能在postgresql.conf 或者服务器命令行中设置。

min_wal_size (integer)

只要 WAL 磁盘用量保持在这个设置之下，在检查点时旧的 WAL 文件总是 被回收以便未来使用，而不是直接被删除。这可以被用来确保有足够的 WAL 空间被保留来应

付 WAL 使用的高峰，例如运行大型的批处理任务。 默认是 80 MB。这个参数只能在postgresql.conf 或者服务器命令行中设置。

19.5.3. 归档

archive_mode (enum)

当启用archive_mode时，可以通过设置 archive_command命令将完成的 WAL 段发送到 归档存储。除用于禁用的off之外，还有两种模式： on和always。在普通操作期间，这两种模式之间 没有区别，但是当设置为always时，WAL 归档器在归档恢复 或者后备模式下也会被启用。在always模式下，所有从归档恢复 的或者用流复制传来的文件将被（再次）归档。详见 第 26.2.9 节

archive_mode和archive_command是独立的变量，这样可以在不影响归档模式的前提下修改archive_command。这个参数只能在服务器启动时设置。当wal_level被设置为minimal时，archive_mode不能被启用。

archive_command (string)

本地 shell 命令被执行来归档一个完成的 WAL 文件段。字符串中的任何%p被替换成要被归档的文件的路径名， 而%f只被文件名替换（路径名是相对于服务器的工作目录， 即集簇的数据目录）。如果要在命令里嵌入一个真正的%字符，可以使用%%。有一点很重要，该命令只在成功时返回一个零作为退出状态。更多信息请见第 25.3.1 节

这个参数只能在postgresql.conf文件中或在服务器命令行上设置。除非服务器启动时启用了archive_mode，否则它会被忽略。如果archive_mode被启用时，archive_command是一个空字符串（默认），WAL 归档会被临时禁用，但服务器仍会继续累计 WAL 段文件，期待着一个命令被提供。将archive_command设置为一个只返回真但不做任何事的命令（例如/bin/true或 Windows 上的REM）实际上会禁用归档，也会打破归档恢复所需的 WAL 文件链，因此只有在极少数情况下才能用。

archive_timeout (integer)

archive_command仅在已完成的 WAL 段上调用。因此，如果你的服务器只产生很少的 WAL 流量（或产生流量的周期很长），那么在事务完成和它被安全地记录到归档存储之间将有一个很长的延迟。为了限制未归档数据存在的时间，你可以设置archive_timeout来强制服务器来周期性地切换到一个新的 WAL 段文件。当这个参数被设置为大于零时，只要从上次段文件切换后过了参数所设置的那么多秒并且已经有过任何数据库活动（包括一个单一检查点），服务器将切换到一个新的段文件（如果没有数据库活动则会跳过检查点）。注意，由于强制切换而提早关闭的被归档文件仍然与完整的归档文件长度相同。因此，使用非常短的archive_timeout是不明智的 — 它将占用巨大的归档存储。一分钟左右的archive_timeout设置通常比较合理。如果你希望数据能被更快地从主服务器上复制下来，你应该考虑使用流复制而不是归档。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

19.6. 复制

这些设置控制内建流复制特性（见第 26.2.5 节的行为。服务器将可以是主控服务器或后备服务器。主控机能发送数据，而后备机总是被复制数据的接收者。当使用级联复制（见第 26.2.7 节时，后备服务器也可以是发送者，同时也是接收者。这些参数主要用于发送服务器和后备服务器，尽管某些只在主服务器上有意义。如果有必要，设置可以在集群中变化而不出问题。

19.6.1. 发送服务器

这些参数可以在任何发送复制数据给一个或多个后备服务器的服务器上设置。主控机总是一个发送服务器，因此这些参数总是要在主控机上设置。这些参数的角色和含义不会在一个后备机变成主控机后改变。

max_wal_senders (integer)

指定来自后备服务器或流式基础备份客户端的并发连接的最大数量（即同时运行 WAL 发送进程 的最大数）。默认值是10。值0意味着禁用复制。WAL 发送进程被计算在连接总数内，因此该参数的值必须小于max_connections减去superuser_reserved_connections的值。突然的流客户端断开 连接可能留下一个孤立连接槽（知道达到超时），因此这个参数应该设置得略高于最大客户端 连接数，这样断开连接的客户端可以立刻重新连接。这个参数只能在服务器启动时被设置。此外，wal_level必须设置为replica或更高级别以允许来自后备服 务器的连接。

max_replication_slots (integer)

指定服务器可以支持的复制槽（见第 26.2.6 节 最大数量。默认值为10。这个参数只能在服务器启动时设置。将它设置为一个比当前已有复制槽要少的值会阻碍服务器启动。此外，要允许使用复制槽， wal_level必须被设置为replica或 更高。

wal_keep_segments (integer)

指定在后备服务器需要为流复制获取日志段文件的情况下，pg_wal目录下所能保留的过去日志文件段的最小数目。每个段通常是 16 兆字节。如果一个连接到发送服务器的后备服务器落后了超过wal_keep_segments个段，发送服务器可以移除一个后备机仍然需要的 WAL 段，在这种情况下复制连接将被中断。最终结果是下行连接也将最终失败（不过，如果在使用 WAL 归档，后备服务器可以通过从归档获取段来恢复）。

只设置pg_wal中保留的文件段的最小数目；系统可能需要为 WAL 归档或从一个检查点恢复保留更多段。如果wal_keep_segments为零（默认值）， 更多的空间来 存放WAL归档或从一个检查点恢复。如果wal_keep_segments是零（缺省）， 系统不会为后备目的保留任何多余的段，因此后备服务器可用的旧 WAL 段的数量是一个上个检查点位置和 WAL 归档状态的函数。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

wal_sender_timeout (integer)

中断那些停止活动超过指定毫秒数的复制连接。这对发送服务器检测一个后备机崩溃或网络中断有用。零值将禁用该超时机制。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是 60 秒。

track_commit_timestamp (boolean)

记录事务的提交时间。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。默认值是off。

19.6.2. 主服务器

这些参数可以在发送复制数据给一个或多个后备服务器的主控/主要服务器上设置。注意除了这些参数之外，在主控服务器上必须设置合适的wal_level，并且也启用可选的 WAL 归档（见第 19.5.3 节。这些参数值与后备服务器无关，尽管你可能希望为了准备好一个后备机转变成主控机来设置这些参数。

synchronous_standby_names (string)

如第 26.2.8 所述，这个参数指定一个支持同步复制的后备服务器的列表。可能会有一个或者多个活动的同步后备服务器，在这些后备服务器确认收到它们的数据之后，等待提交的事务将被允许继续下去。同步后备服务器是那些名字出现在这个列表前面，并且当前已连接并且正在实时流传输数据（如pg_stat_replication视图中streaming的状态所示）的服务器。指定多于一台同步后备可以得到非常高的可用性并且能防止数据丢失。

用于这一目的的后备服务器的名称是其application_name设置，它在后备服务器的连接信息中设置。在物理复制后备的情况下，这应该被设置在recovery.conf文件

的primary_conninfo设置中，默认是walreceiver。对于逻辑复制，可以在订阅的连接信息中设置。对于其他复制流消费者，请参考其文档。

这个参数使用下面的语法之一来指定一个后备服务器列表：

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

其中num_sync是事务需要等待其回复的同步后备服务器的数量，standby_name是一个后备服务器的名称。FIRST以及ANY指定从所列服务器中选取同步后备的方法。

关键词FIRST加上num_sync指定一种基于优先的同步复制，并且会让事务提交等待，直到它们的WAL记录被复制到基于优先级选择的num_sync台同步后备上为止。例如，设置FIRST 3 (s1, s2, s3, s4)将导致每次提交都等待来自三台较高优先级的后备机的答复，这三台后备机将从后备服务器s1、s2、s3以及s4中选出。在该列表中出现较早的后备服务器将被给予较高的优先级，并且将被考虑为同步后备。列表中出现的其他后备服务器表示潜在的同步后备。如果当前的任何同步后备因为某种原因断开连接，它将立刻被下一个最高优先级的后备服务器替代。关键词FIRST是可选的。

关键词ANY加上num_sync指定一种基于规定数量的同步复制，并且会让事务提交等待，直到它们的WAL记录被复制到所列出后备服务器中的至少num_sync台上为止。例如，设置ANY 3 (s1, s2, s3, s4)将导致每次提交会在收到s1、s2、s3以及s4中任意三台后备服务器的回答后立刻继续下去。

FIRST和ANY是大小写不敏感的。如果这些关键词被用作后备服务器的名字，其standby_name必须被放在双引号内。

PostgreSQL版本 9.6 之前使用过第三种语法，目前也仍然支持。它和FIRST和num_sync等于1的第一种语法相同。例如，FIRST 1 (s1, s2)和s1, s2具有相同的含义：s1或者s2会被选中作为同步后备服务器。

特殊项*匹配任意后备名称。

没有机制强制后备服务器名称的唯一性。在出现重复的情况下，匹配的后备之一将被认为是较高优先级，不过无法弄清到底是哪一个。

> **注意**
>
> 每一个standby_name都应该具有合法 SQL 标识符的形式，除非它是*。如果必要你可以使用双引号。但是注意在比较standby_name和后备机应用程序名称时是大小写不敏感的（不管有没有双引号）。

如果这里没有指定同步后备机名称，那么同步复制不能被启用并且事务提交将不会等待复制。这是默认的配置。即便当同步复制被启用时，个体事务也可以被配置为不等待复制，做法是将synchronous_commit参数设置为local或off。

这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

vacuum_defer_cleanup_age (integer)

指定VACUUM和HOT更新在清除死亡行版本之前，应该推迟多久（以事务数量计）。默认值是零个事务，表示死亡行版本将被尽可能快地清除，即当它们不再对任何打开的事务可见时尽快清除。在一个支持热后备服务器的主服务器上，你可能希望把这个参数设置为一个非零值，如第 26.5 节所述。这允许后备机上的查询有更多时间来完成而不会由于先前的行清除产生冲突。但是，由于该值是用在主服务器上发生的写事务的数目衡量的，很难预测对后备机查询可用的附加时间到底是多少。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

你也可以考虑设置后备服务器上的hot_standby_feedback作为使用这个参数的一种替代方案。

这无法阻止已经达到old_snapshot_threshold所指定年龄的死亡行被清除。

19.6.3. 后备服务器

这些设置空值接收复制数据的一个后备服务器的行为。它们的值与主服务器无关。

hot_standby (boolean)

指定在恢复期间，你是否能够连接并运行查询，如第 26.5 节所述。默认值是on。这个参数只能在服务器启动时设置。它只在归档恢复期间或后备机模式下才有效。

max_standby_archive_delay (integer)

当热后备机处于活动状态时，这个参数决定取消那些与即将应用的 WAL 项冲突的后备机查询之前，后备服务器应该等待多久，如第 26.5.2 节所述。当 WAL 数据被从 WAL 归档（并且因此不是当前的 WAL）时，max_standby_archive_delay可以应用。默认值是 30 秒。如果没有指定，衡量单位是毫秒。值 -1 允许后备机一直等到冲突查询结束。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

注意，max_standby_archive_delay与取消之前一个查询能够运行的最长时间不同；它表示应用任何一个 WAL 段数据能够被允许的最长总时间。因此，如果一个查询早于 WAL 段导致了显著的延迟，后续冲突查询将只有更少的时间。

max_standby_streaming_delay (integer)

当热后备机处于活动状态时，这个参数决定取消那些与即将应用的 WAL 项冲突的后备机查询之前，后备服务器应该等待多久，如第 26.5.2 节所述。当 WAL 数据正在通过流复制被接收时，max_standby_streaming_delay可以应用。默认值是 30 秒。如果没有指定，衡量单位是毫秒。值 -1 允许后备机一直等到冲突查询结束。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

注意，max_standby_streaming_delay与取消之前一个查询能够运行的最长时间不同；它表示在从主服务器接收到 WAL 数据并立刻应用它能够被允许的最长总时间。因此，如果一个查询导致了显著的延迟，后续冲突查询将只有更少的时间，直到后备服务器再次赶上进度。

wal_receiver_status_interval (integer)

指定在后备机上的 WAL 接收者进程向主服务器或上游后备机发送有关复制进度的信息的最小频度，它可以使用pg_stat_replication视图看到。后备机将报告它已经写入的上一个预写式日志位置、它已经刷到磁盘的上一个位置以及它已经应用的最后一个位置。这个参数的值是报告之间的最大间隔，以秒计。每次写入或刷出位置改变时会发送状态更新，或者至少按这个参数的指定的频度发送。因此，应用位置可能比真实位置略微滞后。将这个参数设置为零将完全禁用状态更新。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是 10 秒。

hot_standby_feedback (boolean)

指定一个热后备机是否将会向主服务器或上游后备机发送有关于后备机上当前正被执行的查询的反馈。这个参数可以被用来排除由于记录清除导致的查询取消，但是可能导致在主服务器上用于某些负载的数据库膨胀。反馈消息的发送频度不会高于每个wal_receiver_status_interval周期发送一次。默认值是off。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

如果使用级联复制，反馈将被向上游传递直到它最后到达主服务器。后备机在接收到反馈之后除了传递给上游不会做任何其他操作。

这个设置不会覆盖主服务器上的old_snapshot_threshold的行为，后备服务器上一个超过了主服务器年龄阈值的快照可能会变得不可用，导致后备服务器上事务的取消。这是因为old_snapshot_threshold是为了对死亡行能够存在的时间给出一个绝对限制，不然就会因为一个后备服务器的配置而被违背。

wal_receiver_timeout (integer)

中止处于非活动状态超过指定毫秒数的复制链接。这对于正在接收的后备服务器检测主服务器崩溃或网络断开有用。值零会禁用超时机制。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是 60 秒。

wal_retrieve_retry_interval (integer)

指定当从任何来源（流复制、本地pg_wal或者 WAL 归档）都得不到 WAL 数据时，后备服务器应该等待多久才去重新尝试 获取 WAL 数据。这个参数只能在postgresql.conf文件 或者服务器命令行中设置。默认值是 5 秒。如果没有指定，则单位是毫秒。

这个参数对恢复中的节点需要为新 WAL 数据可用等待多少时间的配置有用。 例如，在归档恢复中，通过减小这个参数的值可以让恢复更积极地检测新的 WAL 日志文件。在一个 WAL 活动较低的系统上，增加这个参数的值可以减 少访问 WAL 归档所必需的请求数量，这对于例如云环境是有用的，在其中 对于基础设施的访问时间也是被考虑的。

19.6.4. 订阅者

这些设置控制逻辑复制订阅者的行为。它们在发布者上的值与此无关。

注意，配置参数wal_receiver_timeout、wal_receiver_status_interval以及wal_retrieve_retry_interval也影响逻辑复制工作者。

max_logical_replication_workers (int)

指定逻辑复制工作者的最大数目。这同时包括应用工作者和表同步工作者。

逻辑复制工作者是从max_worker_processes定义的池中取出的。

默认值是4。

max_sync_workers_per_subscription (integer)

每个订阅的同步工作者的最大数目。这个参数控制订阅初始化期间或者新表增加时的初始数据拷贝的并行度。

当前，每个表只能有一个同步工作者。

同步工作者是从max_logical_replication_workers定义的池中取出的。

默认值为2。

19.7. 查询规划

19.7.1. 规划器方法配制

这些配置参数提供了影响查询优化器选择查询规划的原始方法。如果优化器 为特定的查询选择的缺省规划并不是最优，那么我们就可以通过使用这些 配置参数强制优化器选择一个更好的规划来temporary解决这个 问题。不过，永久地关闭这些设置几乎从不是个好主意。更好的改善优化器 选择规划的方法包括调节Section 18.6.2、 更频繁运行ANALYZE、增大配置参数 default_statistics_target的值、使用 ALTER TABLE SET STATISTICS为某个字段增加收集的 统计信息。 这些配置参数影响查询优化器选择查询计划的暴力方法。如果优化器为一个特定查询选择的默认计划不是最优的，一种临时解决方案是使用这些配置参数之一来

强制优化器选择一个不同的计划。提高优化器选择的计划质量的更好的方式包括调整规划器的代价常数（见第 19.7.2 节、手工运行ANALYZE、增加default_statistics_target配置参数的值以及使用ALTER TABLE SET STATISTICS增加为特定列收集的统计信息量。

enable_bitmapscan (boolean)

允许或禁止查询规划器使用位图扫描计划类型。默认值是on。

enable_gathermerge (boolean)

启用或者禁用查询规划器对收集归并计划类型的使用。默认值是on。

enable_hashagg (boolean)

允许或禁用查询规划器使用哈希聚集计划类型。默认值是on。

enable_hashjoin (boolean)

允许或禁止查询规划器使用哈希连接计划类型。默认值是on。

enable_indexscan (boolean)

允许或禁止查询规划器使用索引扫描计划类型。默认值是on。

enable_indexonlyscan (boolean)

允许或禁止查询规划器使用只用索引扫描计划类型（见第 11.9 节。默认值是on。

enable_material (boolean)

允许或者禁止查询规划器使用物化。它不可能完全禁用物化，但是关闭这个变量将阻止规划器插入物化节点，除非为了保证正确性。默认值是on。

enable_mergejoin (boolean)

允许或禁止查询规划器使用归并连接计划类型。默认值是on。

enable_nestloop (boolean)

允许或禁止查询规划器使用嵌套循环连接计划。它不可能完全禁止嵌套循环连接，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_parallel_append (boolean)

允许或禁止查询规划器使用并行追加计划类型。默认值是on。

enable_parallel_hash (boolean)

允许或禁止查询规划器对并行哈希使用哈希连接计划类型。如果哈希连接计划也没有启用，这个参数没有效果。默认值是on。

enable_partition_pruning (boolean)

允许或者禁止查询规划器从查询计划中消除一个分区表的分区。这也控制着规划器产生允许执行器在查询执行期间移除（忽略）分区的查询计划的能力。默认值是on。详情请参考第 5.10.4 节

enable_partitionwise_join (boolean)

允许或者禁止查询规划器使用面向分区的连接，这使得分区表之间的连接以连接匹配的分区的方式来执行。面向分区的连接当前只适用于连接条件包括所有分区键的情况，连接条件必须是相同的数据类型并且子分区集合要完全匹配。由于面向分区的连接规划在规划期间会使用可观的CPU时间和内存，所以默认值为off。

enable_partitionwise_aggregate (boolean)

允许或者禁止查询规划器使用面向分区的分组或聚集，这使得在分区表上的分组或聚集可以在每个分区上分别执行。如果GROUP BY子句不包括分区键，只有部分聚集能够以基于每个分区的方式执行，并且finalization必须最后执行。由于面向分区的分组或聚集在规划期间会使用可观的CPU时间和内存，所以默认值为off。

enable_seqscan (boolean)

允许或禁止查询规划器使用顺序扫描计划类型。它不可能完全禁止顺序扫描，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_sort (boolean)

允许或禁止查询规划器使用显式排序步骤。它不可能完全禁止显式排序，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_tidscan (boolean)

允许或禁止查询规划器使用TID扫描计划类型。默认值是on。

19.7.2. 规划器代价常量

这一节中描述的代价变量可以按照任意尺度衡量。我们只关心它们的相对值，将它们以相同的因子缩放不会影响规划器的选择。默认情况下，这些代价变量是基于顺序页面获取的代价的，即seq_page_cost被设置为1.0并且其他代价变量都参考它来设置。不过你可以使用你喜欢的不同尺度，例如在一个特定机器上的真实执行时间。

> **注意**
>
> 不幸的是，没有一种良定义的方法来决定代价变量的理想值。它们最好被作为一个特定安装将接收到的查询的平均值来对待。这意味着基于少量的实验来改变它们是有风险的。

seq_page_cost (floating point)

设置规划器计算一次顺序磁盘页面抓取的开销。默认值是1.0。 通过设置同名的表空间参数，这个值可以重写为一个特定的表空间。 参阅ALTER TABLESPACE。 设置规划器对一系列顺序磁盘页面获取中的一次的代价估计。默认值是 1.0。通过把表和索引放在一个特殊的表空间（要设置该表空间的同名参数）中可以覆盖这个值（见ALTER TABLESPACE）。

random_page_cost (floating point)

设置规划器对一次非顺序获取磁盘页面的代价估计。默认值是 4.0。通过把表和索引放在一个特殊的表空间（要设置该表空间的同名参数）中可以覆盖这个值（见ALTER TABLESPACE）。

减少这个值（相对于seq_page_cost）将导致系统更倾向于索引扫描；提高它将让索引扫描看起来相对更昂贵。你可以一起提高或降低两个值来改变磁盘 I/O 代价相对于 CPU 代价的重要性，后者由下列参数描述。

对磁盘存储的随机访问通常比顺序访问要贵不止四倍。但是，由于对磁盘的大部分随机访问（例如被索引的读取）都被假定在高速缓冲中进行，所以使用了一个较低的默认值（4.0）。默认值可以被想成把随机访问建模为比顺序访问慢 40 倍，而期望 90% 的随机读取会被缓存。

如果你相信 90% 的缓冲率对你的负载是一个不正确的假设，你可以增加 random_page_cost 来更好的反映随机存储读取的真正代价。相应地，如果你的数据可以

完全放在高速缓存中（例如当数据库小于服务器总内存时），降低 random_page_cost 可能是合适的。为具有很低的随机读取代价的存储（例如固态驱动器）采用较低的 random_page_cost 值可能更好。

> **提示**
>
> 虽然允许你将random_page_cost设置的比 seq_page_cost小，但是物理上的实际情况并不受此影响。 然而当所有数据库都位于内存中时，两者设置为相等是非常合理的，因为 在此情况下，乱序抓取并不比顺序抓取开销更大。同样，在缓冲率很高的 数据库上，你应当相对于 CPU 开销同时降低这两个值，因为获取内存中 的页比通常情况下的开销小许多。 尽管系统可以是你把random_page_cost设置得小于seq_page_cost，但是实际上没有意义。不过，如果数据库被整个缓存在 RAM 中，将它们设置为相等是有意义的，因为在那种情况中不按顺序访问页面是没有惩罚值的。同样，在一个高度缓存化的数据库中，你应该相对于 CPU 参数降低这两个值，因为获取一个已经在 RAM 中的页面的代价要远小于通常情况下的代价。

cpu_tuple_cost (floating point)

设置规划器对一次查询中处理每一行的代价估计。默认值是 0.01。

cpu_index_tuple_cost (floating point)

设置规划器对一次索引扫描中处理每一个索引项的代价估计。默认值是 0.005。

cpu_operator_cost (floating point)

设置规划器对于一次查询中处理每个操作符或函数的代价估计。默认值是 0.0025。

parallel_setup_cost (floating point)

设置规划器对启动并行工作者进程的代价估计。默认是 1000。

parallel_tuple_cost (floating point)

设置规划器对于从一个并行工作者进程传递一个元组给另一个进程的代价估计。默认是 0.1。

min_parallel_table_scan_size (integer)

为必须扫描的表数据量设置一个最小值，扫描的表数据量超过这一个值才会考虑使用并行扫描。对于并行顺序扫描，被扫描的表数据量总是等于表的尺寸，但是在使用索引时，被扫描的表数据量通常会更小。默认值是8兆字节（8MB）。

min_parallel_index_scan_size (integer)

为必须扫描的索引数据量设置一个最小值，扫描的索引数据量超过这一个值时才会考虑使用并行扫描。注意并行索引扫描通常并不会触及整个索引，它是规划器认为该扫描会实际用到的相关页面的数量。默认值是512千字节（512kB）。

effective_cache_size (integer)

设置规划器对一个单一查询可用的有效磁盘缓冲区尺寸的假设。这个参数会被考虑在使用一个索引的代价估计中，更高的数值会使得索引扫描更可能被使用，更低的数值会使得顺序扫描更可能被使用。在设置这个参数时，你还应该考虑PostgreSQL的共享缓冲区以及将被用于PostgreSQL数据文件的内核磁盘缓冲区，尽管有些数据可能在两个地方都存在。另外，还要考虑预计在不同表上的并发查询数目，因为它们必须共享可用的空

间。这个参数对PostgreSQL分配的共享内存尺寸没有影响，它也不会保留内核磁盘缓冲，它只用于估计的目的。系统也不会假设在查询之间数据会保留在磁盘缓冲中。默认值是 4吉字节（4GB）。

jit_above_cost (floating point)

设置激活JIT编译的查询代价，如果查询代价超过这个值就会激活JIT编译（如果启用了JIT，见第 32 章。执行JIT会消耗一些规划时间，但是能够加速查询执行。将这个值设置为-1会禁用JIT编译。默认值是100000。

jit_inline_above_cost (floating point)

设置JIT编译尝试内联函数和操作符的查询代价阈值，如果查询代价超过这个值，JIT编译就会尝试内联。内联会增加规划时间，但是可以提高执行速度。将这个参数设置成小于jit_above_cost是没有意义的。将这个参数设置为-1会禁用内联。默认值是500000。

jit_optimize_above_cost (floating point)

设置JIT编译应用优化的查询代价阈值，如果查询代价超过这个值，JIT编译就会应用开销较大的优化。这类优化会增加规划时间，但是更能够改进执行速度。将这个参数设置成小于jit_above_cost是没有意义的，并且将它设置成大于jit_inline_above_cost也未必有益。将这个参数设置为-1会禁用开销较大的优化。默认值是500000。

19.7.3. 遗传查询优化

GEQO是一个使用探索式搜索来执行查询规划的算法。它可以降低负载查询的规划时间。 同时，GEQO的检索是随机的，因此它的规划可能会不可确定。 更多信息参阅Chapter 50。 遗传查询规划器（GEQO）是一种使用启发式搜索来进行查询规划的算法。它可以降低对于复杂查询（连接很多表的查询）的规划时间，但是代价是它产生的计划有时候要差于使用穷举搜索算法找到的计划。详见第 60 章

geqo (boolean)

允许或禁止遗传查询优化。默认是启用。在生产环境中通常最好不要关闭它。geqo_threshold变量提供了对 GEQO 更细粒度的空值。

geqo_threshold (integer)

只有当涉及的FROM项数量至少有这么多个的时候，才使用遗传查询优化（注意一个FULL OUTER JOIN只被计为一个FROM项）。默认值是 12。对于更简单的查询，通常会使用普通的穷举搜索规划器，但是对于有很多表的查询穷举搜索会花很长时间，通常比执行一个次优的计划带来的惩罚值还要长。因此，在查询尺寸上的一个阈值是管理 GEQO 使用的一种方便的方法。

geqo_effort (integer)

控制 GEQO 里规划时间和查询规划的有效性之间的平衡。这个变量必须是 一个范围从 1 到 10 的整数。缺省值是 5 。大的数值增加花在进行查询 规划上面的时间，但是也很可能会提高选中更有效的查询规划的几率。 控制 GEQO 中规划时间和查询计划质量之间的折中。这个变量必须是位于 1 到 10 之间的一个整数。默认值是 5。更大的值会增加花在查询规划上的时间，但是同时也增加了选择一个高效查询计划的可能性。

geqo_effort实际并不直接做任何事情；它只是被用来计算其他影响 GEQO 行为的变量（如下所述）的默认值。如果你愿意，你可以手工设置其他参数。

geqo_pool_size (integer)

控制 GEQO 使用的池尺寸，它就是遗传种群中的个体数目。它必须至少为 2，且有用的值通常在 100 到 1000 之间。如果它被设置为零（默认设置）则会基于geqo_effort和查询中表的数量选择一个合适的值。

geqo_generations (integer)

控制 GEQO 使用的子代数目。子代的意思是算法的迭代次数。它必须至少 是 1 ，有用的值范围和池大小相同。如果设置为零(缺省)，那么将基于 geqo_pool_size 选取合适的值。 控制 GEQO 使用的代数，也是算法的迭代次数。它必须至少为 1，并且有用值的范围和池尺寸相同。如果它被设置为零（默认设置）则会基于geqo_pool_size选择一个合适的值。

geqo_selection_bias (floating point)

控制 GEQO 使用的选择偏好。选择偏好是种群中的选择压力。值可以是 1.5 到 2.0 之间，后者是默认值。

geqo_seed (floating point)

控制 GEQO 使用的随机数生成器的初始值，随机数生成器用于在连接顺序搜索空间中选择随机路径。该值可以从 0 （默认值）到 1。变化该值会改变被探索的连接路径集合，并且可能导致找到一个更好或更差的路径。

19.7.4. 其他规划器选项

default_statistics_target (integer)

为没有通过ALTER TABLE SET STATISTICS设置列相关目标的表列设置默认统计目标。更大的值增加了需要做ANALYZE的时间，但是可能会改善规划器的估计质量。默认值是 100。有关PostgreSQL查询规划器使用的统计信息的更多内容， 请参考第 14.2 节

constraint_exclusion (enum)

constraint_exclusion的允许值是on（对所有表检查约束）、off（从不检查约束）和partition（只对继承的子表和UNION ALL子查询检查约束）。partition是默认设置。它通常被用于继承和分区表来提高性能。

当对一个特定表允许这个参数，规划器比较查询条件和表的CHECK约束，并且忽略那些条件违反约束的表扫描。例如：

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

在启用约束排除时，这个SELECT将完全不会扫描child1000，从而提高性能。

目前，约束排除只在通过继承表实现表分区的情况中被默认启用。为所有表启用它会增加额外的规划开销，特别是在简单查询上并且不会产生任何好处。如果没有继承分区表时，最好是完全关闭它。

更多关于使用约束排除和分区的信息请参阅第 5.10.5 节

cursor_tuple_fraction (floating point)

设置规划器对将被检索的一个游标的行的比例的估计。默认值是 0.1。更小的值使得规划器偏向为游标使用“快速开始”计划，它将很快地检索前几行但是可能需要很长时间来获取所有行。更大的值强调总的估计时间。最大设置为 1.0，游标将和普通查询完全一样地被规划，只考虑总估计时间并且不考虑前几行会被多快地返回。

from_collapse_limit (integer)

如果生成的FROM列表不超过这么多项，规划器将把子查询融合到上层查询。较小的值可以减少规划时间，但是可能 会生成较差的查询计划。默认值是 8。详见第 14.3 节

将这个值设置为geqo_threshold或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见第 19.7.3 节

jit (boolean)

决定如果可用（见第 32 章，PostgreSQL是否可以使用JIT编译。默认值是off。

join_collapse_limit (integer)

如果得出的列表中不超过这么多项，那么规划器将把显式JOIN（除了FULL JOIN）结构重写到 FROM项列表中。较小的值可减少规划时间，但是可能会生成差些的查询计划。

默认情况下，这个变量被设置成和from_collapse_limit相同， 这样适合大多数使用。把它设置为 1 可避免任何显式JOIN的重排序。因此查询中指定的显式连接顺序就是关系被连接的实际顺序。因为查询规划器并不是总能 选取最优的连接顺序，高级用户可以选择暂时把这个变量设置为 1，然后显式地指定他们想要的连接顺序。更多信息请见第 14.3 节

将这个值设置为geqo_threshold或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见第 19.7.3 节

parallel_leader_participation (boolean)

允许领导者进程执行Gather和Gather Merge节点之下的查询计划而不是等待工作者进程。默认值是on。将这个值设置为off会降低工作者由于领导者读取元组速度不够快而被阻塞的可能性，但是要求领导者在产生第一个元组之前等待工作者进程启动。领导者能够帮助或者阻碍性能的程度取决于计划的类型、工作者的数量以及查询时长。

force_parallel_mode (enum)

允许为测试目的使用并行查询，即便是并不期望在性能上得到效益。force_parallel_mode的允许值是off（只在期望改进性能时才使用并行模式）、on（只要查询被认为是安全的，就强制使用并行查询）以及regress（和on相似，但是有如下文所解释的额外行为改变）。

更具体地说，把这个值设置为on会在任何一个对于并行查询安全的查询计划顶端增加一个Gather节点，这样查询会在一个并行工作者中运行。即便当一个并行工作者不可用或者不能被使用时，诸如开始一个子事务等在并行查询环境中会被禁止的操作将会被禁止，除非规划器相信这样做会导致查询失败。当这个选项被设置时如果出现失败或者意料之外的结果，查询使用的某些函数可能需要被标记为PARALLEL UNSAFE（或者可能是PARALLEL RESTRICTED）。

把这个值设置为regress具有设置成on所有相同的效果，外加一些有助于自动回归测试的额外的效果。一般来说，来自于一个并行工作者的消息会包括一个上下文行指出这一点，但是设置为regress会消除这一行，这样输出就和非并行执行完全一样。同样，被这个设置加到计划上的Gather节点在EXPLAIN输出终会被隐藏起来，这样产生的输出匹配设置为off时产生的输出。

19.8. 错误报告和日志

19.8.1. 在哪里做日志

log_destination (string)

PostgreSQL支持多种方法来记录服务器消息，包括stderr、csvlog和syslog。在 Windows 上还支持eventlog。设置这个参数为一个由想要的日志目的地的列表，之间用逗号分隔。默认值是只记录到stderr。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

如果csvlog被包括在log_destination中，日志项会以“逗号分隔值” （CSV）格式被输出，这样可以很方便地把日志载入到程序中。详见第 19.8.4 节要产生 CSV 格式的日志输出，必须启用logging_collector。

当包括有stderr或csvlog时，会创建文件current_logfiles来记录当前正在被日志收集器使用的日志文件的位置以及相关的日志目的地。这提供了一种查找实例当前使用的日志的便利手段。这里是该文件内容的一个例子：

```
stderr log/postgresql.log
csvlog log/postgresql.csv
```

当由于轮转效应创建一个新的日志文件时以及log_destination被重载时，current_logfiles文件会被重建。当log_destination中不包括stderr和csvlog时以及当日志收集器被禁用时，这个文件会被删除。

> **注意**
>
> 在大多数 Unix 系统上，你将需要修改系统的syslog守护进程的配置来使用log_destination的syslog选项。PostgreSQL可以在syslog设备LOCAL0到LOCAL7中记录（见syslog_facility），但是大部分平台上的默认syslog配置会丢弃所有这种消息。你将需要增加这样的内容：
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> 到syslog守护进程的配置文件来让它工作。
>
> 在 Windows 上，当你使用log_destination的eventlog选项时，你应该在操作系统中注册一个事件源极其库，这样 Windows 事件查看器能够清楚地显示事件日志消息。详见第 18.11 节

logging_collector (boolean)

这个参数启用日志收集器，它是一个捕捉被发送到stderr的日志消息的后台进程，并且它会将这些消息重定向到日志文件中。这种方法比记录到syslog通常更有用，因为某些类型的消息不会在syslog输出中出现（一个常见的例子是动态链接器错误消息；另一个例子是由archive_command等脚本产生的错误消息）。这个参数只能在服务器启动时设置。

> **注意**
>
> 也可以不使用日志收集器而把日志记录到stderr，日志消息将只会去到服务器的stderr被定向到的位置。不过，那种方法只适合于低日志量，因为它没有提供方法来轮转日志文件。还有，在某些不使用日志收集器的平台上可能会导致丢失或者混淆日志输出，因为多个进程并发写入同一个日志文件时会覆盖彼此的输出。

> **注意**
>
> 日志收集器被设计成从来不会丢失消息。这意味着在极高的负载下，如果服务器进程试图在收集器已经落后时发送更多的日志消息，那么它会被阻塞。相反，syslog倾向于在无法写入消息时丢掉消息，这意味着在这样的情况下它可能会无法记录某些消息，但是它不会阻塞系统的其他部分。

log_directory (string)

当logging_collector被启用时，这个参数决定日志文件将被在哪个目录下创建。它可以被指定为一个绝对路径，也可以被指定为一个相对于集簇数据目录的相对路径。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。 默认是pg_log。

log_filename (string)

当logging_collector被启用时，这个参数设置被创建的日志文件的文件名。该值被视为一种strftime模式，因此%转义可以被用来指定根据时间变化的文件名（注意如果有任何时区独立的%转义，计算将在由log_timezone指定的时区中完成）。被支持的%转义和开放组织的strftime[1]说明中列举的类似。注意系统的strftime不会被直接使用，因此平台相关（非标准）的扩展无法工作。 默认是postgresql-%Y-%m-%d_%H%M%S.log。

如果你不使用转义来指定一个文件名，你应该计划使用一个日志轮转工具来避免最终填满整个磁盘。在 8.4 发行之前，如果不存在%转义，PostgreSQL将追加新日志文件创建时间的纪元，但是现在已经不再这样做了。

如果在log_destination中启用了 CSV 格式输出，.csv将会被追加到时间戳日志文件名中来创建 CSV 格式输出（如果log_filename以.log结尾，该后缀会被替换）。

这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

log_file_mode (integer)

在 Unix 系统上，当logging_collector被启用时，这个参数设置日志文件的权限（在微软 Windows 上这个参数将被忽略）。这个参数值应当是一个数字形式的模式，它可以被chmod和umask系统调用接受（要使用通常的十进制格式，该数字必须以一个0（零）开始）。

默认的权限是0600，表示只有服务器拥有者才能读取或写入日志文件。其他常用的设置是0640，它允许拥有者的组成员读取文件。不过要注意你需要修改log_directory为将文件存储在集簇数据目录之外的某个位置，才能利用这个设置。在任何情况下，让日志文件变成任何人都可读是不明智的，因为日志文件中可能包含敏感数据。

这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

log_rotation_age (integer)

当logging_collector被启用时，这个参数决定一个个体日志文件的最长生命期。当这些分钟过去后，一个新的日志文件将被创建。将这个参数设置为零将禁用基于时间的新日志文件创建。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

log_rotation_size (integer)

当logging_collector被启用时，这个参数决定一个个体日志文件的最大尺寸。当这么多千字节被发送到一个日志文件后，将创建一个新的日志文件。将这个参数设置为零将禁用基于尺寸的新日志文件创建。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

log_truncate_on_rotation (boolean)

当logging_collector被启用时，这个参数将导致PostgreSQL截断（覆盖而不是追加）任何已有的同名日志文件。不过，截断只在一个新文件由于基于时间的轮转被打开时发生，在服务器启动或基于尺寸的轮转时不会发生。如果被关闭，在所有情况下以前存在的文件将被追加。例如，使用这个设置和一个类似postgresql-%H.log的log_filename将导致产生 24 个每小时的日志文件，并且循环地覆盖它们。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

例子：要保留 7 天的日志，每天的一个日志文件被命令为server_log.Mon、server_log.Tue等等，并且自动用本周的日志覆盖上一周的日志。

[1] http://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

可以这样做：将log_filename设置为server_log.%a、将log_truncate_on_rotation设置为on并且将log_rotation_age设置为1440。

例子：要保留 24 小时的日志，每个小时一个日志文件，但是在日志文件尺寸超过 1GB 时轮转。可以这样做：将log_filename设置为server_log.%H%M、 将log_truncate_on_rotation设置为on、 将log_rotation_age设置为60并且将log_rotation_size设置为1000000。 在log_filename中包括%M允许发生任何尺寸驱动的轮转来选择一个不同于每个小时的初始文件名的新文件名。

syslog_facility (enum)

当启用了向syslog记录时，这个参数决定要使用的syslog“设备”。你可以在LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、 LOCAL5、LOCAL6、LOCAL7中选择，默认值是LOCAL0。还请参阅系统的syslog守护进程的文档。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

syslog_ident (string)

当启用了向syslog记录时，这个参数决定用来标识syslog中的PostgreSQL消息的程序名。默认值是postgres。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

syslog_sequence_numbers (boolean)

当日志被记录到syslog并且这个设置为 on （默认）时，每一个消息会被加上一个增长的序号作为前缀（例如[2]）。这种行为避开了很多 syslog 实现默认采用的“--- 上一个消息重复 N 次 ---”形式。在现代 syslog 实现中，抑制重复消息是可以配置的（例如rsyslog中的$RepeatedMsgReduction），因此这个参数可能不是必需的。此外，如果你真的想抑制重复消息，你可以把这个参数设置为 off。

这个参数只能在postgresql.conf文件或者服务器命令行上设置。

syslog_split_messages (boolean)

当启用把日志记录到syslog时，这个参数决定消息如何送达 syslog。当设置为 on（默认）时，消息会被分成行，并且长的行也会被划分以便能够放到 1024 字节中，这是传统 syslog 实现一种典型的尺寸限制。当设置为 off 时，PostgreSQL 服务器日志消息会被原样送达 syslog 服务，而处理可能的大体量消息的任务由 syslog 服务负责。

如果 syslog 最终被记录到一个文本文件中，那么两种设置的效果是一样的，但最好设置为 on，因为大部分 syslog 实现要么不能处理大型消息，要么需要做特殊的配置以处理大型消息。但是如果 syslog 最终写入到某种其他媒介，有必要让消息保持逻辑上的整体性（也更加有用）。

这个参数只能在postgresql.conf文件或者服务器命令行上设置。

event_source (string)

当启用了向事件日志记录时，这个参数决定用来标识日志中PostgreSQL消息的程序名。默认值是PostgreSQL。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

19.8.2. 什么时候记录日志

log_min_messages (enum)

控制哪些消息级别 被写入到服务器日志。有效值是DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、 INFO、NOTICE、WARNING、 ERROR、LOG、FATAL和 PANIC。每个级别都包括以后的所有级别。级别越靠后，被发送的消息越少。默认值是WARNING。注意LOG在这里有与client_min_messages中不同的排名。只有超级用户可以改变这个设置。

log_min_error_statement (enum)

控制哪些导致一个错误情况的 SQL 语句被记录在服务器日志中。任何指定 严重级别 或更高级别的消息的当前 SQL 语句将被包括在日志项中。有效值是DEBUG5、DEBUG4、DEBUG3、 DEBUG2、DEBUG1、 INFO、NOTICE、 WARNING、ERROR、LOG、 FATAL和PANIC。默认值是ERROR，它表示导致错误、日志消息、致命错误或恐慌错误的语句将被记录在日志中。要有效地关闭记录失败语句，将这个参数设置为PANIC。只有超级用户可以改变这个设置。

log_min_duration_statement (integer)

如果语句运行至少指定的毫秒数，将导致记录每一个这种完成的语句的持续时间。将这个参数设置为零将打印所有语句的执行时间。设置为 -1 （默认值）将停止记录语句持续时间。例如，如果你设置它为250ms，那么所有运行 250ms 或更久的 SQL 语句将被记录。启用这个参数可以有助于追踪应用中未优化的查询。只有超级用户可以改变这个设置。

对于使用扩展查询协议的客户端，解析、绑定和执行步骤的持续时间将被独立记录。

> **注意**
>
> 当把这个选项和log_statement一起使用时，已经被log_statement记录的语句文本不会在持续时间日志消息中重复。如果你没有使用syslog，我们推荐你使用log_line_prefix记录 PID 或会话 ID，这样你可以使用进程 ID 或会话 ID 把语句消息链接到后来的持续时间消息。

表 19.解释了PostgreSQL所使用的消息严重级别。如果日志输出被发送到syslog或 Windows的eventlog，严重级别会按照表中所示进行转换。

表 19.1. 消息严重级别

严重性	用法	syslog	eventlog
DEBUG1..DEBUG5	为开发者提供连续的更详细的信息。	DEBUG	INFORMATION
INFO	提供用户隐式要求的信息，例如来自VACUUM VERBOSE的输出。	INFO	INFORMATION
NOTICE	提供可能对用户有用的信息，例如长标识符截断提示。	NOTICE	INFORMATION
WARNING	提供可能出现的问题的警告，例如在一个事务块外COMMIT。	NOTICE	WARNING
ERROR	报告一个导致当前命令中断的错误。	WARNING	ERROR
LOG	报告管理员可能感兴趣的信息，例如检查点活动。	INFO	INFORMATION
FATAL	报告一个导致当前会话中断的错误。	ERR	ERROR
PANIC	报告一个导致所有数据库会话中断的错误。	CRIT	ERROR

19.8.3. 记录什么到日志

application_name (string)

application_name可以是任意小于NAMEDATALEN个字符（标准编译中是 64 个字符）的字符串。这通常由一个应用通过到服务器的连接设置。该名称将被显示在pg_stat_activity视图中并被包括在 CSV 日志项中。它也会被通过log_line_prefix包括在普通日志项中。只有可打印 ASCII 字符能被使用在application_name之中。其他字符将被替换为问号（?）。

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

这个参数启用发出各种调试输出。当设置时，会打印生成的解析树， 查询重写输出，或执行的每个查询的执行计划。这些信息是在LOG 信息级别发出，因此默认的，它们会出现在服务器日志中，但不会发送给客户端。 可以通过 client_min_messages和/或log_min_messages 来设置。这些参数缺省是off。 这些参数将会让多种调试输出被发出。当被设置时，它们为每一个被执行的查询打印结果分析树、查询重写器输出或执行计划。这些消息在LOG消息级别上被发出，因此默认情况下它们将出现在服务器日志中但不会被发送到客户端。你可以通过调整client_min_messages和/或log_min_messages来改变这种情况。这些参数默认是关闭的。

debug_pretty_print (boolean)

当被设置时，debug_pretty_print会缩进由debug_print_parse、debug_print_rewritten或 debug_print_plan产生的输出。这将导致比关闭参数时使用的“紧凑”模式可读性更强但是更长的输出。它默认是打开的。

log_checkpoints (boolean)

导致检查点和重启点被记录在服务器日志中。一些统计信息也被包括在日志消息中，包括写入缓冲区的数据和写它们所花的时间。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是关闭。

log_connections (boolean)

导致每一次尝试对服务器的连接被记录，客户端认证的成功完成也会被记录。 只有超级用户能在会话开始时更改这个参数，在会话中它不能被更改。默认 为off。

> **注意**
>
> 某些客户端程序（例如psql）在要求密码时会尝试连接两次，因此重复的“收到连接”消息并不一定表示一个错误。

log_disconnections (boolean)

导致会话终止被记录。日志输出提供的信息类似于 log_connections，不过还外加会话的持续时间。 只有超级用户能在会话开始时更改这个参数，在会话中它不能被更改。默认 为off。

log_duration (boolean)

导致每一个完成的语句的持续时间被记录。默认值是off。只有超级用户可以改变这个设置。

对于使用扩展查询协议的客户端，解析、绑定和执行步骤的持续时间将被独立记录。

> **注意**
>
> 设置这个选项和设置log_min_duration_statement为零之间的区别是，超过log_min_duration_statement强制查询的文本被记录，但这个选项不会。因此，如果log_duration为on并且log_min_duration_statement为正值，所有持续时间都将被记录，但是只有超过阈值的语句才会被记录查询文本。这种行为有助于在高负载安装中收集统计信息。

log_error_verbosity (enum)

控制为每一个被记录的消息要写入到服务器日志的细节量。有效值是TERSE、DEFAULT和VERBOSE，每一个都为显示的消息增加更多域。TERSE排除记录DETAIL、HINT、QUERY和CONTEXT错误信息。VERBOSE输出包括SQLSTATE错误码（见附录 A 以及产生错误的源代码文件名、函数名和行号。只有超级用户能够更改这个设置。

log_hostname (boolean)

默认情况下，连接日志消息只显示连接主机的 IP 地址。打开这个参数将导致也记录主机名。注意根据你的主机名解析设置，这可能会导致很微小的性能损失。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

log_line_prefix (string)

这是一个printf风格的字符串，它在每个日志行的开头输出。%字符开始“转义序列”，它将被按照下文描述的替换成状态信息。未识别的转义被忽略。其他字符被直接复制到日志行。某些转义只被会话进程识别并且被主服务器进程等后台进程当作空。通过指定一个在%之后和该选项之前的数字可以让状态信息左对齐或右对齐。 负值将导致在右边用空格填充状态信息已达到最小宽度，而正值则在左边填充。填充对于日志文 件的人类可读性大有帮助。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。默认值是'%m [%p] '，它记录时间戳和进程ID。

转义	效果	只限会话
%a	应用名	是
%u	用户名	是
%d	数据库名	是
%r	远程主机名或 IP 地址，以及远程端口	是
%h	远程主机名或 IP 地址	是
%p	进程 ID	否
%t	无毫秒的时间戳	否
%m	带毫秒的时间戳	否
%n	带毫秒的时间戳（作为 Unix 时间戳）	no
%i	命令标签：会话当前命令的类型	是
%e	SQLSTATE 错误代码	否
%c	会话 ID：见下文	否
%l	对每个会话或进程的日志行号，从 1 开始	否
%s	进程开始的时间戳	否

转义	效果	只限会话
%v	虚拟事务 ID (backendID/localXID)	否
%x	事务 ID （如果未分配则为0）	否
%q	不产生输出，但是告诉非会话进程在字符串的这一点停止；会话进程忽略	否
%%	纯文字 %	否

%c转义打印一个准唯一的会话标识符，它由两个 4 字节的十六进制数（不带先导零）组成，以点号分隔。这些数字是进程启动时间和进程 ID，因此%c也可以被用作保存打印这些项的方式的空间。例如，要从pg_stat_activity生成会话标识符，使用这个查询：

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

> **提示**
>
> 如果你为log_line_prefix设置了非空值，你通常应该让它的最后一个字符为空格，这样用以提供和日志行的剩余部分的视觉区别。也可以使用标点符号。

> **提示**
>
> Syslog产生自己的时间戳和进程 ID 信息，因此如果你记录到syslog你可能不希望包括哪些转义。

> **提示**
>
> 在包括仅在会话（后端）上下文中可用的信息（如用户名或者数据库名）时，%q转义很有用。例如：
>
> ```
> log_line_prefix = '%m [%p] %q%u@%d/%a '
> ```

log_lock_waits (boolean)

控制当一个会话为获得一个锁等到超过deadlock_timeout时，是否要产生一个日志消息。这有助于决定是否所等待造成了性能低下。默认值是off。只有超级用户可以更改这个设置。

log_statement (enum)

控制哪些 SQL 语句被记录。有效值是 none (off)、ddl、mod和 all（所有语句）。ddl记录所有数据定义语句，例如CREATE、ALTER和 DROP语句。mod记录所有ddl语句，外加数据修改语句例如INSERT, UPDATE、DELETE、TRUNCATE, 和COPY FROM。 如果PREPARE、EXECUTE和 EXPLAIN ANALYZE包含合适类型的命令，它们也会被记录。对于使用扩展查询协议的客户端，当收到一个执行消息时会产生日志并且会包括绑定参数的值（任何内嵌的单引号会被双写）。

默认值是none。只有超级用户可以改变这个设置。

> **注意**
>
> 即使使用log_statement = all设置，包含简单语法错误的语句也不会被记录。这是因为只有在完成基本语法解析并确定了语句类型之后才会发出日志消息。在扩展查询协议的情况下，在执行阶段之前（即在解析分析或规划期间）出错的语句也不会被记录。将log_min_error_statement设置为ERROR（或更低）来记录这种语句。

log_replication_commands (boolean)

导致每一个复制命令都被记录在服务器日志中。关于复制命令的详细信息请见第 53.4 节默认值是off。只有 超级用户可以更改这个设置。

log_temp_files (integer)

控制记录临时文件名和尺寸。临时文件可以被创建用来排序、哈希和存储临时查询结果。当每一个临时文件被删除时都会制作一个日志项。一个零值记录所有临时文件信息，而正值只记录尺寸大于或等于指定千字节数的文件。默认设置为 -1，它禁用这种记录。只有超级用户可以更改这个设置。

log_timezone (string)

设置在服务器日志中写入的时间戳的时区。和TimeZone不同，这个值是集簇范围的，因此所有会话将报告一致的时间戳。内建默认值是GMT，但是通常会被在postgresql.conf中覆盖。initdb将安装一个对应于其系统环境的设置。详见第 8.5.3 节这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

19.8.4. 使用 CSV 格式的日志输出

在log_destination列表中包括csvlog提供了一种便捷方式将日志文件导入到一个数据库表。这个选项发出逗号分隔值（CSV）格式的日志行，包括这些列： 带毫秒的时间戳、 用户名、 数据库名、 进程 ID、 客户端主机:端口号、 会话 ID、 每个会话的行号、 命令标签、 会话开始时间、 虚拟事务 ID、 普通事务 ID、 错误严重性、 SQLSTATE 代码、 错误消息、 错误消息详情、 提示、 导致错误的内部查询（如果有）、 错误位置所在的字符计数、 错误上下文、 导致错误的用户查询（如果有且被log_min_error_statement启用）、 错误位置所在的字符计数、 在 PostgreSQL 源代码中错误的位置（如果log_error_verbosity被设置为verbose）以及应用名。 下面是一个定义用来存储 CSV 格式日志输出的样表：

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
```

```
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

使用COPY FROM命令将一个日志文件导入到这个表中：

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

你可以做一些事情来简化导入 CSV 日志文件：

1. 设置log_filename和log_rotation_age为你的日志文件提供一种一致的、可预测的命名空间。这让你预测文件名会是怎样以及知道什么时候一个个体日志文件完成并且因此准备好被导入。

2. 将log_rotation_size设置为 0 来禁用基于尺寸的日志轮转，因为它使得日志文件名难以预测。

3. 将log_truncate_on_rotation设置为on，这样在同一个文件中旧日志数据不会与新数据混杂。

4. 上述表定义包括一个主键声明。这有助于避免意外地两次导入相同的信息。COPY命令一次提交所有它导入的数据，因此任何错误将导致整个导入失败。如果你导入一个部分完成的日志文件并且稍后当它完全完成后再次导入，主键违背将导致导入失败。请等到日志完成且被关闭之后再导入。这个过程也可以避免意外地导入部分完成的行，这种行也将导致COPY失败。

19.8.5. 进程标题

这些设置控制服务器进程的进程标题如何被修改。进程标题通常可以用ps或者 Windows 上的进程浏览器等程序来查看。详见第 28.1 节

cluster_name (string)

为这个集簇中所有的服务器进程设置出现在进程标题中的集簇名称。这个名称可以是任何长度不超过NAMEDATALEN个字符（在标准编译中是 64字符）的任何字符串。只有可打印的 ASCII 字符能被用在cluster_name值中。其他字符将被替换为问号（?）。如果这个参数被设置为空字符串''（也是默认值），将不会显示名称。这个参数只能在服务器启动时设置。

update_process_title (boolean)

启用进程标题更新，每次服务器接收到一个新的 SQL 命令时都更新进程标题。在大部分平台上这个设置默认为on，但是由于 Windows 上更新进程标题的开销更大，所以在 Windows 这个设置默认为off。只有超级用户能更改这个设置。

19.9. 运行时统计数据

19.9.1. 查询和索引统计收集器

这些参数控制服务器范围的统计数据收集特性。当统计收集被启用时，被产生的数据可以通过pg_stat和pg_statio系统视图族访问。详见第 28 章

track_activities (boolean)

启用对每个会话的当前执行命令的信息收集，还有命令开始执行的时间。这个参数默认为打开。注意即使被启用，这些信息也不是对所有用户可见，只有超级用户和拥有报告信息的会话的用户可见，因此它不会表现为一个安全风险。只有超级用户可以更改这个设置。

track_activity_query_size (integer)

指定跟踪每个活动会话当前执行命令所保留的字节数，它们被用于pg_stat_activity.query域。默认值是 1024。这个参数只能在服务器启动时被设置。

track_counts (boolean)

启用在数据库活动上的统计收集。这个参数默认为打开，因为自动清理守护进程需要被收集的信息。只有超级用户可以更改这个设置。

track_io_timing (boolean)

启用对系统 I/O 调用的计时。这个参数默认为关闭，因为它将重复地向操作系统查询当前时间，这会在某些平台上导致显著的负荷。你可以使用pg_test_timing工具来度量你的系统中计时的开销。I/O 计时信息被显示在pg_stat_database中、当BUFFERS选项被使用时的EXPLAIN输出中以及pg_stat_statements中。只有超级用户可以更改这个设置。

track_functions (enum)

启用跟踪函数调用计数和用时。指定pl只跟踪过程语言函数，指定all还会跟踪 SQL 和 C 语言函数。默认值是none，它禁用函数统计跟踪。只有超级用户可以更改这个设置。

> **注意**
>
> 简单到足以被“内联”到调用查询中的 SQL 语言函数不会被跟踪， 而不管这个设置。

stats_temp_directory (string)

设置存储临时统计数据的目录。这可以是一个相对于数据目录的路径或一个绝对路径。默认值是pg_stat_tmp。在一个基于 RAM 的文件系统上指明这个参数将降低物理 I/O 需求，并且提高性能。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

19.9.2. 统计监控

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)

对每个查询，向服务器日志里输出相应模块的性能统计。这是一种粗糙的分析工具。类似于 Unix 的getrusage()系统功能。log_statement_stats报告总的语句统计，而其它的报告针每个模块的统计。log_statement_stats不能和 其它任何针对每个模块统计的选项一起启用。所有这些选项都是默认禁用的。只有超级用户可以更改这个设置。

19.10. 自动清理

这些设置控制autovacuum特性的行为。详情请参考 第 24.1.6 节注意很多这些设置可以被针对每个表 的设置所覆盖，请见存储参数。

autovacuum (boolean)

控制服务器是否运行自动清理启动器后台进程。默认为开启， 不过要自动清理正常工作还需要启用track_counts。 该参数只能在postgresql.conf文件或服务器命令行中设置， 不过，通过更改表存储参数可以为表禁用自动清理。

注意即使该参数被禁用，系统也会在需要防止事务ID回卷时发起清理进程。详情请见第 24.1.5 节

log_autovacuum_min_duration (integer)

如果自动清理运行至少该值所指定的毫秒数，被自动清理执行的每一个动作都会被日志记录。 将该参数设置为0会记录所有的自动清理动作。-1（默认值）将禁用对自动清理动作的记录。 例如，如果你将它设置为250ms，则所有运行250ms或更长时间的 自动清理和分析将被记录。此外，当该参数被设置为除-1外的任何值时， 如果一个自动清理动作由于一个锁冲突或者被并发删除的关系而被跳过，将会为此记录一个消息。 开启这个参数对于追踪自动清理活动非常有用。这个参数只能在 postgresql.conf文件或者服务器命令行中设置。但是可以通过更改表的存储 参数为个别表覆盖这个设置。

autovacuum_max_workers (integer)

指定能同时运行的自动清理进程（除了自动清理启动器之外）的最大数量。默认值为3。该参数只能在服务器启动时设置。

autovacuum_naptime (integer)

指定自动清理在任意给定数据库上运行的最小延迟。在每一轮中后台进程检查数据库并根据需要为数据库中的表发出VACUUM和ANALYZE命令。延迟以秒计，且默认值为1分钟（1min）。该参数只能在postgresql.conf文件或在服务器命令行上设置。

autovacuum_vacuum_threshold (integer)

指定能在一个表上触发VACUUM的被插入、被更新或被删除元组的最小数量。默认值为50个元组。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_analyze_threshold (integer)

指定能在一个表上触发ANALYZE的被插入、被更新或被删除元组的最小数量。默认值为50个元组。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_vacuum_scale_factor (floating point)

指定一个表尺寸的分数，在决定是否触发VACUUM时将它加到autovacuum_vacuum_threshold上。默认值为0.2（表尺寸的20%）。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_analyze_scale_factor (floating point)

指定一个表尺寸的分数，在决定是否触发ANALYZE时将它加到autovacuum_analyze_threshold上。默认值为0.1（表尺寸的10%）。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_freeze_max_age (integer)

指定在一个VACUUM操作被强制执行来防止表中事务ID回卷之前，一个表的pg_class.relfrozenxid域能保持的最大年龄（事务的）。注意即便自动清理被禁用，系统也将发起自动清理进程来阻止回卷。

清理也允许从pg_xact子目录中移除旧文件，这也是为什么默认值被设置为较低的2亿事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见第 24.1.5 节

autovacuum_multixact_freeze_max_age (integer)

指定在一个VACUUM操作被强制执行来防止表中多事务ID回卷之前，一个表的pg_class.relminmxid域能保持的最大年龄（多事务的）。注意即便自动清理被禁用，系统也将发起自动清理进程来阻止回卷。

清理多事务也允许从pg_multixact/members和pg_multixact/offsets子目录中移除旧文件，这也是为什么默认值被设置为较低的400万多事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见第 24.1.5.1 节

autovacuum_vacuum_cost_delay (integer)

指定用于自动VACUUM操作中的代价延迟值。如果指定-1（默认值），则使用vacuum_cost_delay值。默认值为20毫秒。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_vacuum_cost_limit (integer)

指定用于自动VACUUM操作中的代价限制值。如果指定-1（默认值），则使用vacuum_cost_limit值。注意该值被按比例地分配到运行中的自动清理工作者上（如果有多个），因此每一个工作者的限制值之和不会超过这个变量中的值。该参数只能在postgresql.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

19.11. 客户端连接默认值

19.11.1. 语句行为

client_min_messages (enum)

控制被发送给客户端的消息级别。有效值是DEBUG5、 DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、 WARNING、ERROR。 每个级别都包括其后的所有级别。级别越靠后，被发送的消息越少。默认值是NOTICE。 注意LOG在这里有与log_min_messages中不同的排名。

INFO 级别的消息总是被发送到客户端。

search_path (string)

这个变量指定当一个对象（表、数据类型、函数等）被用一个无模式限定的简单名称引用时，用于进行搜索该对象的模式顺序。当在不同模式中有同名对象时，将使用第一个在搜索路径中被找到的对象。一个不属于搜索路径中任何一个模式的对象只能通过用限定名（带点号）指定包含它的模式来引用。

search_path的值必需是一个逗号分隔的模式名列表。任何不是一个已有模式的名称，或者是一个用户不具有USAGE权限的模式，将被安静地忽略。

如果列表项之一是特殊名$user，则具有SESSION_USER返回的名字的模式将取代它（如果有这样一个模式并且该用户有该模式的USAGE权限；如果没有，$user会被忽略）。

系统目录模式pg_catalog总是被搜索，不管它是否在搜索路径中被提及。如果它在路径中被提及，那么它将被按照路径指定的顺序搜索。如果pg_catalog不在路径中，则它将在任何路径项之前被搜索。

同样，当前会话的临时表模式pg_temp_nnn也总是被搜索（如果存在）。它可以在路径中通过使用别名pg_temp显式列出。如果在路径中没有列出，那么会首先对它进行搜索（甚

至是在pg_catalog之前）。然而，临时模式只被用来搜索关系（表、视图、序列等）和数据类型名。它从不用于搜索函数或操作符名。

当对象创建时没有指定一个特定目标模式，它们将被放置在search_path中第一个合法模式中。如果搜索路径为空将报告一个错误。

这个参数的缺省值是"$user", public。这种设置支持一个数据库（其中没有用户拥有私有模式，并且所有人共享使用public）、每个用户私有模式及其组合的共享使用。其它效果可以通过全局或者针对每个用户修改默认搜索路径设置获得。

更多有关方案处理的信息，请参考第 5.8 节特别地，只有当数据库只有一个用户或者有少数的相互信任的用户时，默认配置是合适的。

搜索路径的当前有效值可以通过SQL函数current_schemas检查（见第 9.25 节。它和检查search_path的值不太一样，因为current_schemas显示出现在search_path中的项如何被解析。

row_security (boolean)

这个变量控制是否以抛出一个错误来代替应用一条行安全性策略。在设置为on时，策略正常应用。在设置为off时，只要有至少一条策略被应用则查询就会失败。默认为on。受限的行可见性会导致不正确的结果时，可将其改成off。例如，pg_dump默认会做这种更改。这个变量对能绕过每一条行安全性策略的角色（即超级用户和具有BYPASSRLS）属性的角色没有效果。

更多关于行安全性策略的信息请见CREATE POLICY。

default_tablespace (string)

这个变量指定当一个CREATE命令没有显式指定一个表空间时，创建对象（表和索引）的默认表空间。

该值要么是一个表空间的名字，要么是一个指定使用当前数据库默认表空间的空字符串。如果该值和任何现有表空间的名字都不匹配，PostgreSQL将自动使用当前数据库的默认表空间。如果指定了一个非默认的表空间，用户必须对它有CREATE权限，否则创建企图将失败。

这个变量不被用于临时表，对临时表会使用temp_tablespaces。

当创建数据库时也会使用这个变量。默认情况下，一个新数据库会从它的模板数据库继承其表空间设置。

有关表空间的更多的信息，请见第 22.6 节

temp_tablespaces (string)

这个变量指定当一个CREATE命令没有显式指定一个表空间时，创建临时对象（临时表和临时表上的索引）的默认表空间。用于排序大型数据集的临时文件也被创建在这些表空间中。

该值是一个表空间名字的列表。当列表中有多于一个名称时，每次一个临时对象被创建时PostgreSQL随机选择列表中的一个成员。例外是在一个事务中，连续创建的临时对象被放置在里表中的连续表空间中。如果列表被选中元素是一个空字符串，PostgreSQL将自动使用当前数据库的默认表空间。

当temp_tablespaces被交互式地设置时，指定一个不存在的表空间是一种错误，类似于为用户指定一个不具有CREATE权限的表空间。不过，当使用一个之前设置的值时，不存在的表空间会被忽略，就像用户缺少CREATE权限的表空间一样。特殊地，使用一个在postgresql.conf中设置的值时，这条规则起效。

默认值是一个空字符串，它使得所有临时对象被创建在当前数据库的默认表空间中。

参阅default_tablespace。

check_function_bodies (boolean)

这个参数通常为打开 。当设置为off时，它禁用CREATE FUNCTION期间对函数体字符串的验证。禁用验证避免了验证处理的副作用并且避免了如向前引用导致的伪肯定。在代表其他用户载入函数之前设置这个参数为off；pg_dump会自动这样做。

default_transaction_isolation (enum)

每个 SQL 事务都有一个隔离级别，可以是“读未提交”、“读已提交”、“可重复读”或者“可序列化”。这个参数控制每个新事务的默认隔离级别。默认是“读已提交”。

更多信息请参阅第 13 章和SET TRANSACTION。

default_transaction_read_only (boolean)

一个只读的 SQL 事务不能修改非临时表。这个参数控制每个新事务的默认只读状态 。默认是off（读/写）。

更多信息请参考SET TRANSACTION。

default_transaction_deferrable (boolean)

当运行在可序列化隔离级别时，一个可延迟只读 SQL 事务可以在它被允许继续之前延迟一段时间。但是，一旦它开始执行就不会产生任何用来保证可序列化性的负荷；因此序列化代码将没有任何理由因为并发更新而强制它中断，使得这个选项适合于长时间运行的只读事务。

这个参数控制每个新事务的默认可延迟状态。目前它对读写事务或者那些操作在低于可序列化隔离级别上的事务无效。默认值是off。

详情请参阅SET TRANSACTION。

session_replication_role (enum)

为当前会话控制复制相关的触发器和规则的触发。需要超级用户权限才能设置这个变量，并且会导致丢弃任何之前缓存下来的查询计划。可能的值有origin（默认）、replica和local。

这个设置的预期用途是由逻辑复制系统在应用所复制的更改时将它设置为replica。其效果将是触发器和规则（没有对其默认配置做修改）在复制机上将不会被触发。更多信息请参考ALTER TABLE的子句ENABLE TRIGGER以及ENABLE RULE。

PostgreSQL在内部会把设置origin和local同样对待。第三方复制系统可能会把这两个值用于其内部目的，例如把local用来标出一个不应复制其更改的会话。

因为外键被实现为触发器，将这个参数设置为replica还会禁用所有的外键检查，如果使用不当可能会让数据处于一种不一致的状态。

statement_timeout (integer)

中止任何使用了超过指定毫秒数的语句，从命令到达服务器开始计时。如果log_min_error_statement被设置为ERROR或更低，语句如果超时也会被记录。一个零值（默认）将关闭这个参数。

我们不推荐在postgresql.conf中设置statement_timeout，因为它会影响所有会话。

lock_timeout (integer)

如果任何语句在试图获取表、索引、行或其他数据库对象上的锁时等到超过指定的毫秒数，该语句将被中止。该时间限制独立地应用于每一次锁获取尝试。该限制会应用到显

式锁定请求（如LOCK TABLE或不带NOWAIT的SELECT FOR UPDATE）和隐式获得的锁。如果log_min_error_statement被设置为ERROR或更低，超时的语句会被记录。一个零值（默认）将关闭这个参数。

与statement_timeout不同，这个超时只在等待锁时发生。注意如果statement_timeout为非零，设置lock_timeout为相同或更大的值没有意义，因为事务超时将总是第一个被触发。

我们不推荐在postgresql.conf中设置lock_timeout，因为它会影响所有会话。

idle_in_transaction_session_timeout (integer)

终止任何已经闲置超过这个参数所指定的时间（以毫秒计）的打开事务的会话。这使得该会话所持有的任何锁被释放，并且其所持有的连接槽可以被重用，它也允许只对这个事务可见的元组被清理。有关于此的详情请见第 24.1 节

默认值 0 会禁用这个特性。

vacuum_freeze_table_age (integer)

当表的pg_class.relfrozenxid域达到该设置指定的年龄时，VACUUM会执行一次激进的扫描。激进的扫描与常规VACUUM的不同在于它会访问每一个可能包含未冻结 XID 或者 MXID 的页面，而不只是那些可能包含死亡元组的页面。默认值是 1.5 亿个事务。尽管用户可以把这个值设置为从 0 到 20 亿，VACUUM会悄悄地将有效值设置为autovacuum_freeze_max_age值的95%，因此在表上启动一次反回卷自动清理之前有机会进行一次定期手动VACUUM。更多信息请见第 24.1.5 节

vacuum_freeze_min_age (integer)

指定VACUUM在扫描表时用来决定是否冻结行版本的切断年龄（以事务计）。默认值是 5 千万个事务。尽管用户可以将这个值设置为从 0 到 10 亿，VACUUM会悄悄地将有效值设置为autovacuum_freeze_max_age值的一半，这样在强制执行的自动清理之间不会有过短的时间间隔。更多信息请见第 24.1.5 节

vacuum_multixact_freeze_table_age (integer)

如果表的pg_class.relminmxid域超过了这个设置指定的年龄，VACUUM会执行一次激进的扫描。激进的扫描与常规VACUUM的区别在于它会访问每一个可能包含未冻结 XID 或者 MXID 的页面，而不是只扫描那些可能包含死亡元组的页面。默认值是 1.5 亿个组合事务。尽管用户可以把这个值设置为从 0 到 20 亿，VACUUM会悄悄地将有效值设置为autovacuum_multixact_freeze_max_age值的95%，因此在表上启动一次反回卷自动清理之前有机会进行一次定期手动VACUUM。更多信息请见第 24.1.5.1 节

vacuum_multixact_freeze_min_age (integer)

指定VACUUM在扫描表时用来决定是否把组合事务 ID 替换为一个更新的事务 ID 或组合事务 ID 的切断年龄（以组合事务计）。默认值是 5 千万个组合事务。尽管用户可以将这个值设置为从 0 到 10 亿，VACUUM会悄悄地将有效值设置为autovacuum_multixact_freeze_max_age值的一半，这样在强制执行的自动清理之间不会有过短的时间间隔。更多信息请见第 24.1.5.1 节

vacuum_cleanup_index_scale_factor (floating point)

指定在以前的统计信息收集过程中计数到的堆元组总数的一个分数，插入不超过这一数量所代表的元组不会导致VACUUM清理阶段的索引扫描。这个设置当前仅适用于B-树索引。

如果没有元组从堆中删除，则当至少满足下列条件之一时，在VACUUM清理阶段仍会扫描B-树索引：索引统计信息过时或者索引中包含在清理时可回收的已删除页。如果新近插

入的元组数占上次统计信息收集时检测到的堆元组总数的比例超过vacuum_cleanup_index_scale_factor，则认为索引信息已经过时。堆元组的总数被存放在索引的元页中。注意，直到VACUUM找不到死亡元组之前，元页中都不包括这个数据。因此只有在第二次以及之后的VACUUM周期检测不到死亡元组时，清理阶段的B-树索引扫描才能被跳过。

该值的取值范围可以是从0到10000000000。当vacuum_cleanup_index_scale_factor被设置为0时，在VACUUM清理期间不会跳过索引扫描。默认值是0.1。

bytea_output (enum)

设置bytea类型值的输出格式。有效值是hex（默认）和 escape（传统的 PostgreSQL 格式）。详见第 8.4 节不管这个设置的值如何，bytea类型总是接受这两种格式的输入。

xmlbinary (enum)

设置二进制值如何被编码为 XML。例如，这适用于通过xmlelement函数或xmlforest函数将bytea值转换到 XML 值。可能的值有base64和hex，它们都是用 XML 模式标准定义的。默认值是base64。更多关于 XML 相关函数的信息可参阅第 9.14 节

这里的实际选择都是根据爱好做出的，只受客户端应用中可能存在的限制的约束。两种方法都支持所有可能的值，尽管十六进制编码将比 base64 编码更大。

xmloption (enum)

当在XML和字符串值之间进行转换时，无论设置DOCUMENT或 CONTENT都是隐式的。可参阅 Section 8.13。 有效值是DOCUMENT和CONTENT。 缺省值是CONTENT。当在 XML 和字符串值之间进行转换时，设置DOCUMENT或CONTENT都是隐式的。详见第 8.13 节有效值是DOCUMENT和CONTENT。默认值是CONTENT。

根据 SQL 标准，设置这个选项的命令是：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

这种语法在 PostgreSQL 也可用。

gin_pending_list_limit (integer)

设置fastupdate被启用时可以使用的 GIN 待处理列表的最大尺寸。如果该列表增长到超过这个最大尺寸，会通过批量将其中的项移入主 GIN 数据结构来清理列表。默认值是四兆字节（4MB）。可以通过更改索引的存储参数来为个别 GIN 索引覆盖这个设置。更多信息请见第 66.4.1 和第 66.5 节

19.11.2. 区域和格式化

DateStyle (string)

设置日期和时间值的显示格式，以及解释有歧义的日期输入值的规则。由于历史原因，这个变量包含两个独立的部分：输出格式声明（ISO、Postgres、SQL或German）、 输入/输出的年/月/日顺序（DMY、MDY或YMD）。这些可以被独立设置或者一起设置。关键字Euro和European是DMY的同义词；关键字US、NonEuro和NonEuropean是MDY的同义词。详见第 8.5 节内建默认值是ISO, MDY，但是initdb将用对应于选中的lc_time区域行为的设置初始化配置文件。

IntervalStyle (enum)

设置间隔值的显示格式。值sql_standard将产生匹配 SQL 标准间隔文本的输出。当DateStyle参数被设置为ISO时，值postgres（默认）将产生匹配PostgreSQL发行 8.4

之前的输出。当DateStyle参数被设置为非ISO输出时，值postgres_verbose会产生匹配PostgreSQL发行 8.4 之前的输出。值iso_8601会产生匹配在 ISO 8601 的 4.4.3.2 节中定义的“带标志符格式”的时间间隔的输出。

IntervalStyle参数也可以影响对有歧义的间隔输入的解释。详见第 8.5.4 节

TimeZone (string)

设置用于显示和解释时间戳的时区。内建默认值是GMT，但是它通常会在postgresql.conf中被覆盖；initdb将安装一个对应于其系统环境的设置。详见第 8.5.3 节

timezone_abbreviations (string)

设置服务器接受的日期时间输入中使用的时区缩写集合。默认值为'Default'， 这个集合在全世界大多数地方都能工作。也还有'Australia'和'India'，以及可能为一种特定安装定义的其他集合。详见第 B.4 节

extra_float_digits (integer)

这个参数为浮点值调整显示的位数，包括float4、float8以及几何数据类型。参数值被加在标准的位数（FLT_DIG或DBL_DIG，视情况而定）上。该值最高可以被设置为 3 来包括部分有效位；这特别有助于转储需要被准确恢复的否点数据。或者它可以被设置为负值来消除不需要的位。另请参见第 8.1.3 节

client_encoding (string)

设置客户端编码（字符集）。默认使用数据库编码。PostgreSQL服务器所支持的字符集在第 23.3.1 节描述。

lc_messages (string)

设置消息显示的语言。可接受的值是系统相关的；详见第 23.1 节如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

在一些系统上，这个区域分类并不存在。仍然可以设置这个变量，只是不会有任何效果。同样，所期望语言的翻译消息也可能不存在。在这种情况下，你将仍然继续看到英文消息。

只有超级用户可以改变这个设置。因为它同时影响发送到服务器日志和客户端的消息。一个不正确的值可能会降低服务器日志的可读性。

lc_monetary (string)

设置用于格式化货币量的区域，例如用to_char函数族。可接受的值是系统相关的；详见第 23.1 节如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

lc_numeric (string)

设置用于格式化数字的区域，例如用to_char函数族。可接受的值是系统相关的；详见第 23.1 节如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

lc_time (string)

设置用于格式化日期和时间的区域，例如用to_char函数族。可接受的值是系统相关的；详见第 23.1 节如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

default_text_search_config (string)

选择被那些没有显式参数指定配置的文本搜索函数变体使用的文本搜索配置。详见第 12 章内建默认值是pg_catalog.simple，但是如果能够标识一个匹配区域的配置，initdb将用对应于选中的lc_ctype区域的设置初始化配置文件。

19.11.3. 共享库预载入

为了载入附加的功能或者达到提高性能的目的，可用多个设置来预先载入共享库到服务器中。 例如'$libdir/mylib'设置可能会导致mylib.so（或者某些平台上的mylib.sl）从安装的标准库目录被预装载。这些设置之间的区别在于生效的时间以及改变它们所需的特权。

可以用这个方法预装载PostgreSQL的过程语言库，通常是使用'$libdir/plXXX'语法，其中的XXX是pgsql、perl、tcl或python。

只有特别为与PostgreSQL一起使用设计的共享库才能以这种方式载入。每一个PostgreSQL支持 的库都有一个“魔法块”，它会被检查以保证兼容性。由于这个原因，非PostgreSQL无法 以这种方式被载入。你可能可以使用操作系统的工具（如LD_PRELOAD）载入它。

总之，请参考特定模块的文档来用推荐的方法载入它。

local_preload_libraries (string)

这个变量指定一个或者多个要在连接开始时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数 值只在连接开始时生效。后续的更改不会有任何效果。如果一个指定的 库没有找到，连接尝试将会失败。

任何用户都能设置这个选项。正因为如此，能被这样载入的库被严格限制为出现于安装的标准库 目录中plugins子目录下的共享库（保证只有“安全的”库被安装到 这里是数据库管理员的责任）。local_preload_libraries中的项可以显式 指定这个目录，例如$libdir/plugins/mylib，或者只是指定库的 名称 — mylib 和 $libdir/plugins/mylib的效果是相同的。

这个特性的目的是允许非特权用户在特定的会话中载入正在调试的或者性能度量库， 而无需一个显式的LOAD命令。为了这个目的，通常通过使用客 户端的PGOPTIONS环境变量或者 ALTER ROLE SET来设置这个参数。

不过，除非一个模块被特别设计成由非超级用户以这种方式使用，通常不推荐使用这个设置。应该看看 session_preload_libraries。

session_preload_libraries (string)

这个变量指定一个或者多个要在连接开始时预载入的共享库。只有超级用户更够更改这个设置。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数只在连接开始时起效。后续的改变没有效果。如果指定的库没有找到，连接尝试将会失败。只有超级用户能够更改这个设置。

这个特性的意图是允许在特定会话中载入调试用的或者测量性能的库，而不需要显式的给出一个 LOAD命令。例如，通过用ALTER ROLE SET设置这个参数可以 为一个给定用户名下的所有会话启用auto_explain。还有，无需重启 服务器就能更改这个参数（但是只有新会话启动时才会生效），这样可以以这种方式更容易地增 加新模块，即便它们会应用到所有会话。

和shared_preload_libraries不同，相对于在库被第一次使用 时载入它，在会话开始时载入库并没有什么性能优势。不过，当使用连接池时这样做还是有一些 优势。

shared_preload_libraries (string)

这个变量指定一个或者多个要在服务器启动时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果

需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数只能在服务器启动时设置。如果指定的库没有找到，服务器将无法启动。

有些库需要执行只能在postmaster启动时发生的特定操作，例如分配共享内存、保留轻量级锁 或者启动后台工作者。这些库必须通过这个参数在服务器启动时载入。每个库的详情请见文档。

其他库也能被预载入。通过预载入一个共享库，当该库被第一次使用时就可以避免库的启动时间。 不过，启动每个新服务器进程的时间可能会略有增加，即使该进程从不使用该库。因此，推荐只 把这个参数用于那些要在大多数会话中使用的库上。还有，改变这个参数要求重启服务器，因此 对于短期的调试任务来说这不是好的选择，应该转用session_preload_libraries。

> **注意**
>
> 在 Windows 主机上，在服务器启动时预载入一个库并不会减少启动每个新服务器进程所需的 时间；每一个服务器进程将会重新载入预载入的库。不过，对于那些要在postmaster启动时 执行操作的库来说，Windows 主机上的 shared_preload_libraries任然有用。

jit_provider (string)

这个变量是要被使用的JIT提供者库的名称（见第 32.4.2 节。默认是llvmjit。这个参数只能在服务器启动时设置。

如果这个变量被设置为一个不存在的库，JIT将不可用，但是也不会发生错误。这种特性允许在主PostgreSQL包之外单独安装JIT支持。

19.11.4. 其他默认值

dynamic_library_path (string)

如果需要打开一个可以动态装载的模块并且在CREATE FUNCTION或LOAD命令中指定的文件名没有目录部分（即名字中不包含斜线），那么系统将搜索这个路径以查找所需的文件。

dynamic_library_path的值必须是一个冒号分隔（或者在 Windows 上以分号分隔）的绝对目录路径的列表。如果一个列表元素以特殊字符串开始，$libdir会被替换为PostgreSQL包中已编译好的库目录。这里是PostgreSQL发布提供的模块被安装的位置（使用pg_config --pkglibdir来找到这个目录的名字）。例如：

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:
$libdir'
```

或者在 Windows 环境中：

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

这个参数的默认值是'$libdir'。如果该值被设置为一个空字符串，则关闭自动路径搜索。

这个参数可以在运行时由超级用户修改，但是这样修改的设置只能保持到这个客户端连接的结尾，因此这个方法应该保留给开发目的。 我们建议在postgresql.conf配置文件中设置这个参数。

gin_fuzzy_search_limit (integer)

GIN 索引返回的集合尺寸的软上限。详见第 66.5 节

19.12. 锁管理

deadlock_timeout (integer)

这是进行死锁检测之前在一个锁上等待的总时间（以毫秒计）。死锁检测相对昂贵，因此服务器不会在每次等待锁时都运行这个它。我们乐观地假设在生产应用中死锁是不常出现的，并且只在开始检测死锁之前等待一会儿。增加这个值就减少了浪费在无用的死锁检测上的时间，但是减慢了报告真正死锁错误的速度。默认是 1 秒（1s），这可能是实际中你想要的最小值。在一个高负载的服务器上，你可能需要增大它。这个值的理想设置应该超过你通常的事务时间，这样就可以减少在锁释放之前就开始死锁检查的机会。只有超级用户可以更改这个设置。

当log_lock_waits被设置时，这个参数还可以决定发出关于锁等待的日志之前等待的时长。如果你想调查锁延迟，你可能希望设置一个比正常的deadlock_timeout小的值。

max_locks_per_transaction (integer)

共享锁表跟踪在max_locks_per_transaction * (max_connections + max_prepared_transactions) 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。个体事务可以锁住更多对象，数量可以和锁表中能容纳的所有事务的锁一样多。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经被历史证明是足够的，但是如果你有需要在一个事务中使用很多不同表的查询（例如查询一个有很多子表的父表），你可能需要提高这个值。这个参数只能在服务器启动时设置。

当运行一个后备服务器时，你必须设置这个参数为大于等于主服务器上的值。否则，后备服务器上将不允许查询。

max_pred_locks_per_transaction (integer)

共享谓词锁表跟踪在max_pred_locks_per_transaction * (max_connections + max_prepared_transactions) 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。个体事务可以锁住更多对象，数量可以和锁表中能容纳的所有事务的锁一样多。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经在测试中被证明通常是足够的，但是如果你有需要在一个可序列化事务中使用很多不同表的查询（例如查询一个有很多子表的父表），你可能需要提高这个值。这个参数只能在服务器启动时设置。

max_pred_locks_per_relation (integer)

这个参数控制在谓词锁被提升为覆盖整个关系之前，该谓词锁能够在单个关系上锁住多少页面或元组。大于等于零的值表示一种绝对限制，而负值表示用max_pred_locks_per_transaction除以这个设置的绝对值。默认值为-2，它将保持以往版本的PostgreSQL中的行为。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

max_pred_locks_per_page (integer)

这个参数控制在谓词锁被提升为覆盖整个页面之前，该谓词锁能在单一页面上锁住多少行。默认值是2。这个参数只能在postgresql.conf文件中或者服务器命令行上设置。

19.13. 版本和平台兼容性

19.13.1. 以前的 PostgreSQL 版本

array_nulls (boolean)

这个参数控制数组输入解析器是否把未用引号的NULL识别为一个空数组元素。默认为on，允许输入包含空值的数组值。但是PostgreSQL 8.2 之前的版本不支持数组中的空

值，并且因此将把NULL当作指定一个值为字符串“NULL”的正常数组元素。对于那些要求旧行为的应用的向后兼容性，这个变量可以被设置为off。

注意即使这个变量为off也能够创建包含空值的数组值。

backslash_quote (enum)

这个参数控制字符串文本中的单引号是否能够用\'来表示。首选的 SQL 标准的方法是将其双写（''），但是PostgreSQL在历史上也接受\'。不过使用\'容易导致安全风险，因为在某些客户端字符集编码中，有多字节字符的最后一个字节在数值上等价于 ASCII 的\'。如果客户端代码没有做到正确转义，那么将会导致 SQL 注入攻击。如果服务器拒绝看起来带有被反斜线转义的单引号的查询，那么就可以避免这种风险。backslash_quote的可用值是on（总是允许\'）、off（总是拒绝）以及safe_encoding（只有客户端编码不允许在多字节字符中存在 ASCII \时允许）。safe_encoding是默认设置。

注意在符合标准的字符串文本中，\就表示\。这个参数只影响不符合标准的文本的处理，包括转义字符串语法（E'...'）。

default_with_oids (boolean)

这个参数控制CREATE TABLE和CREATE TABLE AS在既没有指定WITH OIDS也没有指定WITHOUT OIDS的情况下，是否在新创建的表中包含 OID 列。它还决定被SELECT INTO创建的表里面是否包含 OID 。这个参数在默认情况下为off。在PostgreSQL 8.0 及更早的版本中，它默认为on。

我们反对在用户表中使用 OID ，因此大多数安装应该禁用这个变量。需要为一个特殊表使用 OID 的应用应该在创建表的时候指定WITH OIDS。为了兼容不遵循这一行为的老旧应用，这个变量可以被启用。

escape_string_warning (boolean)

打开时，如果在普通字符串文本中（'...'语法）出现了 一个反斜线（\）并且standard_conforming_strings为关闭，那么就会发出一个警告。默认值是on。

希望使用反斜线作为转义符的应用应该被修改来使用转义字符串语法（E'...'），因为在 SQL 标准中普通字符串的默认行为是将反斜线视作一个普通字符。这个变量可以被启用来帮助定位需要被更改的代码。

lo_compat_privileges (boolean)

在PostgreSQL 9.0 之前，大对象不具有访问特权并且因此总是所有用户可读可写的。为了和以前的版本兼容，把这个变量设置为on可以禁用这种新的特权检查。默认是off。只有超级用户可以更改这个设置。

设置这个参数不会禁用所有与大对象相关的安全检查 — 除了那些在PostgreSQL 9.0中已经修改了的默认行为。

operator_precedence_warning (boolean)

当开启时，对于任何从PostgreSQL 9.4 以来由于操作符优先级 变化而导致含义改变的结构，解析器将发出一个警告。这有助于审计应用，已检查是否 优先级变化破坏了什么东西。但是它的本意并不是希望在生产环境中保持打开，因为它 会对某些完全合法、兼容标准的 SQL 代码发出警告。默认为off。

更多信息请见第 4.1.6 节

quote_all_identifiers (boolean)

当数据库产生 SQL 时，强制所有标识符被引号包围，即使它们（当前）不是关键字。这将影响EXPLAIN的输出以及pg_get_viewdef等函数的结果。另请参阅pg_dump和pg_dumpall的--quote-all-identifiers选项。

standard_conforming_strings (boolean)

控制普通字符串文本（'...'）是否按照 SQL 标准把反斜线当普通文本。从PostgreSQL 9.1 开始，默认值为on（之前的发行中默认值为off）。应用可以检查这个参数来判断字符串文本如何被处理。这个参数的存在也可以被当做转义字符串语法（E'...'）被支持的标志。如果一个应用希望反斜线被当做转义字符，应该使用转义字符串语法（第 4.1.2.2 节。

synchronize_seqscans (boolean)

它允许对大型表的顺序扫描与其他扫描同步，因此并发扫描可以在几乎相同的时刻读取相同的块，这样可以分担 I/O 负载。当启用这个参数时，一个扫描可能会从表的中间开始并且之后“绕回”到开头以覆盖所有的行，这样可以与已在进行中的扫描活动同步。对于没有ORDER BY子句的查询来，这样的扫描会在返回行的顺序中造成不可预料的改变。将这个参数设置为off以保证 8.3 之前的行为（顺序扫描总是从表的起始处开始）。默认值是on。

19.13.2. 平台和客户端兼容性

transform_null_equals (boolean)

当打开时，形为expr = NULL（或NULL = expr）的表达式将被当做expr IS NULL， 也就是说，如果expr得出空值值则返回真，否则返回假。正确的 SQL 标准兼容的expr = NULL行为总是返回空（未知）。因此这个参数默认为off。

不过，在Microsoft Access里的过滤表单生成的查询似乎使用expr = NULL来测试空值，因此，如果你使用这个接口访问数据库，你可能想把这个选项打开。因为expr = NULL形式的表达式总是返回空值（使用 SQL 标准解释）。它们不是非常有用并且在普通应用中也不常见，在应用中也不常见，因此这个选项实际上没有什么危害。但是新用户常常对涉及 空值的表达式的语义上感到困惑，因此这个选项默认为关闭。

请注意这个选项只影响= NULL形式，而不影响其它比较操作符或者其它与一些涉及等值操作符的表达式在计算上等效的其他表达式（例如IN）。因此，这个选项不是劣质程序的一般修复。

相关信息请见第 9.2 节

19.14. 错误处理

exit_on_error (boolean)

如果为真，任何错误将中止当前会话。默认情况下，这个值被设置为假，这样只有 FATAL 错误（致命）将中止会话。

restart_after_crash (boolean)

当被设置为真（默认值）时，PostgreSQL将在一次后端崩溃后自动重新初始化。让这个值设置为真通常是将数据库可用性最大化的最佳方法。但是在某些环境中，例如 PostgreSQL被集群软件调用时，禁用重启可能很有用，这样集群软件可以得到控制并且采取它认为适当的行动。

data_sync_retry (boolean)

如果设置为false（默认值），PostgreSQL在将修改的数据文件刷新到文件系统失败时，将引发PANIC级错误。这样会导致数据库服务器崩溃。

在某些操作系统上，回写失败后，内核页面缓存中的数据状态未知。 在某些情况下，它可能已被完全遗忘，因此重试不安全；第二次尝试可能报告为成功，而事实上数据已丢失。在此类情形下，避免数据丢失的唯一方法是在报告任何故障后从WAL中恢复，最好是在调查了故障的根本原因并更换了任何有故障的硬件之后。

如果设置为true，PostgreSQL将报告错误，但会继续运行，以便可以在以后的检查点中重试数据刷新操作。仅在调查操作系统假如回写失败时对缓冲数据的处理方式的情况下，才将其设置为true。

19.15. 预置选项

下列“参数”是只读的，它们是在编译或安装PostgreSQL时决定的。同样，它们被排除在postgresql.conf文件例子之外。这些选项报告特定应用可能感兴趣的多种PostgreSQL行为，特别是管理前端相关的行为。

block_size (integer)

报告一个磁盘块的大小。它由编译服务器时BLCKSZ的值确定。默认值是 8192 字节。有些配置变量的含义（例如shared_buffers）会被block_size影响。详见第 19.4 节

data_checksums (boolean)

报告对这个集簇是否启用了数据校验码。详见data checksums。

data_directory_mode (integer)

在Unix系统上，这个参数报告启动时data_directory所定义的数据目录的权限（在Microsoft Windows上这个参数将总是显示0700）。更多信息请参考group access。

debug_assertions (boolean)

报告编译PostgreSQL时是否启用了断言。 如果PostgreSQL被编译时定义了宏USE_ASSERT_CHECKING is defined when PostgreSQL（例如通过 configure选项--enable-cassert定义），那么会报告已启用。默认情况下 PostgreSQL编译时没有用断言。

integer_datetimes (boolean)

报告PostgreSQL是否在编译时打开了 64 位整数日期和时间。从PostgreSQL 10起，这个值总是on。

lc_collate (string)

报告文本数据排序使用的区域。详见第 23.1 节该值是在数据库被创建时确定的。

lc_ctype (string)

报告决定字符分类的区域。详见第 23.1 节该值是在数据库被创建时决定的。通常它和lc_collate一样，但是可以为特殊应用设置成不同的值。

max_function_args (integer)

报告函数参数的最大数量。它由编译服务器时的FUNC_MAX_ARGS值决定的。默认值是 100 个参数。

max_identifier_length (integer)

报告标识符的最大长度。它由编译服务器时的NAMEDATALEN值减一决定。NAMEDATALEN的默认值是 64 ；因此max_identifier_length的默认值是 63，但是在使用多字节编码时可以少于 63 个字符。

max_index_keys (integer)

报告索引键的最大数目。它由编译服务器时的INDEX_MAX_KEYS值决定。默认值是 32 个键。

segment_size (integer)

报告一个文件段中可以存储的块（页）的数量。由编译服务器时的RELSEG_SIZE值决定。一个段文件的最大尺寸（以字节计）等于segment_size乘以block_size，默认是 1GB。

server_encoding (string)

报告数据库的编码（字符集）。这是在数据库被创建时决定的。通常，客户端只需要关心client_encoding的值。

server_version (string)

报告服务器版本数值。它是由编译服务器时的PG_VERSION值决定的。

server_version_num (integer)

报告服务器版本数值的整数值。它是由编译服务器时的PG_VERSION_NUM值决定的。

wal_block_size (integer)

报告一个 WAL 磁盘块的尺寸。由编译服务器时的XLOG_BLCKSZ值决定。默认是 8192 字节。

wal_segment_size (integer)

报告 WAL 段文件的大小。默认是 16MB。详见第 30.4 节

19.16. 自定义选项

这个特性被设计用来由附加模块向PostgreSQL添加通常不为系统知道的参数（例如过程语言）。这允许使用标准方法配制扩展模块。

自定义选项有两部分名称：一个扩展名，然后是一个句点，再然后是正确的参数名，就像 SQL 中的合格名称。一个例子是plpgsql.variable_conflict。

因为自定义选项可能需要在没有载入相关扩展模块的进程中设置，PostgreSQL将接收任意两部分参数名的设置。这种变量被认为是占位符并且在定义它们的模块被载入之前不会有实际功能。当一个扩展模块被载入，它将加入它的变量定义、根据那些定义转换任何占位符值并且对以其扩展名开始的任意未识别占位符发出警告。

19.17. 开发者选项

下面的参数目的是用在PostgreSQL源代码上， 并且在某些情况下可以帮助恢复严重损坏了的数据库。在一个生产数据库中没有理由使用它们。同样，它们被从例子postgresql.conf文件中排除。请注意许多这些参数要求特殊的源代码编译标志才能工作。

allow_system_table_mods (boolean)

允许对系统表结构的修改。它可以被initdb使用。这个参数只能在服务器启动时设置。

ignore_system_indexes (boolean)

读取系统表时忽略系统索引（但是修改系统表时依然同时更新索引）。这在从被破坏的系统索引中恢复数据的时有用。这个参数在会话开始之后不能被更改。

post_auth_delay (integer)

如果为非零，那么在一个新的服务器进程派生出来之后并且在它开始认证过程之前，就会发生这么多秒的延迟。这是为了给开发者们一个机会在一个服务器进程上附加一个调试器。这个参数在会话开始之后不能被更改。

pre_auth_delay (integer)

如果为非零，那么在一个新的服务器进程派生出来之后并且在它开始认证过程之前，就会发生这么多秒的延迟。这是为了给开发者们一个机会在一个服务器进程上附加一个调试器来跟踪认证过程中的不当行为。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

trace_notify (boolean)

为LISTEN和NOTIFY命令生成大量调试输出。client_min_messages和log_min_messages必须是DEBUG1或者更低才能把这种输出分别发送到客户端或者服务器日志。

trace_recovery_messages (enum)

启用记录与恢复有关的调试输出，否则无法记录。这个参数允许用户覆盖log_min_messages的正常设置，但只用于指定的消息。这个参数的目的是用来调试热后备。有效值包括DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1和LOG。默认值LOG完全不会影响日志决定。其他值会记录相关级别或更高级别的与恢复相关的调试消息，就好像它们具有LOG优先级一样；对于log_min_messages的通用设置，这会无条件的将消息发送给服务器日志。这个参数只能在postgresql.conf文件中或在服务器命令行上设置。

trace_sort (boolean)

如果打开，发出在排序操作中的资源使用的相关信息。只有在编译PostgreSQL时定义了TRACE_SORT宏， 这个参数才可用（不过，当前在默认情况下就定义了TRACE_SORT）。

trace_locks (boolean)

如果开启，发出锁使用情况的信息。被转储信息中包括锁操作的类型、锁的类型和 被锁或被解锁对象的唯一标识符。同样包括的还有已经授予这个对象的锁类型的位掩码和 等待这个对象的锁类型的位掩码。对每一种锁类型，已授权锁和等待锁的计数也会被一起转储。一个日志文件输出的例子如下：

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

被转储结构的详细信息可以在src/include/storage/lock.h中找到。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

trace_lwlocks (boolean)

如果开启，发出轻量级锁的使用信息。轻量级锁主要是为了提供对共享内存数据结构的互斥访问。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

trace_userlocks (boolean)

如果开启，发出关于用户锁使用的信息。与trace_locks的输出一样，但只用于咨询锁。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

trace_lock_oidmin (integer)

如果设置，不会跟踪小于这个 OID 的锁（用于避免在系统表上的输出）。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

trace_lock_table (integer)

无条件地跟踪此表（OID）上的锁。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

debug_deadlocks (boolean)

如果设置，当死锁超时发生时，转储所有当前锁的信息。

只有在编译PostgreSQL时定义了LOCK_DEBUG宏， 这个参数才可用。

log_btree_build_stats (boolean)

如果设置，会记录 B 树操作上的系统资源使用情况统计（内存和 CPU）。

只有在编译PostgreSQL时定义了BTREE_BUILD_STATS宏， 这个参数才可用。

wal_consistency_checking (string)

这个参数被设计用来检查WAL重做例程中的缺陷。当这个参数被启用时，被修改的任何缓冲区的全页映像及其WAL记录都被加入到记录中。如果该记录后来被重放，系统将首先应用每个记录然后测试该记录修改的缓冲区是否符合存储的映像。在某些情况下（例如提示位），小的变动是可以接受的，并且会被忽略。任何预期之外的差别都将导致致命错误，最后中止恢复。

这个设置的默认值是空字符串，它将禁用这一特性。它可以被设置为all以检查所有记录，或者被设置为一个逗号分隔的资源管理器列表用以检查那些资源管理器产生的记录。当前，支持的资源管理器
是heap、heap2、btree、hash、gin、gist、sequence、spgist、brin以及generic。只有超级用户可以更改这一设置。

wal_debug (boolean)

如果被打开，WAL 相关的调试输出将被发出。只有在编译PostgreSQL时定义了WAL_DEBUG宏的情况下，这个参数才可用。

ignore_checksum_failure (boolean)

只有当data checksums被启用时才有效。

在读取过程中检测到一次校验码失败通常会导致PostgreSQL报告一个错误。设置ignore_checksum_failure为打开会导致系统忽略失败（但是仍然报告一个警告），并且继续执行。这种行为可能导致崩溃、传播或隐藏损坏或者其他严重的问题。但是，它允许你绕过错误并且在块头部仍然健全的情况下从表中检索未损坏的元组。如果头部被损坏，即便这个选项被启用系统也将报告一个错误。默认设置是off，并且只能被超级用户改变。

zero_damaged_pages (boolean)

检测到一个损坏的页面头部通常会导致PostgreSQL报告一个错误，并且中止当前事务。把zero_damaged_pages设置为打开会让系统报告一个警告、把损坏的页面填充零，然后继续处理。这种行为会毁掉数据，即被损坏页面上的所有行。但是它允许你绕开错误并且从可能存在表中的任何未损坏页面中检索行。如果由于一次硬件或软件错误而发生毁

坏，这种方法可用于恢复数据。通常你不应该把它设置为打开，除非你已经彻底放弃从表的损坏页面中恢复数据。被填充零的页面不会被强制到磁盘上，因此我们推荐在再次关闭这个参数之前先重建表或索引。默认的设置是off，并且只有超级用户可以改变它。

jit_debugging_support (boolean)

如果LLVM有所需要的功能，用GDB注册所生成的函数。这会让调试更加容易。默认设置是off。这个参数只能在服务器启动时设置。

jit_dump_bitcode (boolean)

把生成的LLVM IR写出到文件系统，写到data_directory中。只有在做JIT内部实现工作时，这个参数才能派上用场。默认设置是off。这个参数只能由超级用户修改。

jit_expressions (boolean)

当JIT编译被激活时（见第 32.2 节），确定表达式是否用JIT编译。默认值是on。

jit_profiling_support (boolean)

如果LLVM有所需的功能，发出需要的数据以允许perf对JIT生成的函数画像。这会写出文件到$HOME/.debug/jit/中，如果需要，由用户负责对其执行清除。默认设置是off。这个参数只能在服务器启动时设置。

jit_tuple_deforming (boolean)

当JIT编译被激活时（见第 32.2 节），确定元组拆解是否被JIT编译。默认值是on。

19.18. 短选项

为了方便起见，系统中还为一些参数提供了单字母的命令行选项开关。它们在表 19.2 中描述。其中一些选项是由于历史原因而存在，并且它们作为一个单字母选项存在并不表示它们会被大量使用。

表 19.2. 短选项键

短选项	等效于
-B x	shared_buffers = x
-d x	log_min_messages = DEBUGx
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h x	listen_addresses = x
-i	listen_addresses = '*'
-k x	unix_socket_directories = x
-l	ssl = on
-N x	max_connections = x
-O	allow_system_table_mods = on
-p x	port = x

短选项	等效于
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S x	work_mem = x
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W x	post_auth_delay = x

第 20 章 客户端认证

当一个客户端应用连接一个数据库服务器时，它将指定以哪个PostgreSQL 数据库用户名连接，就像我们以一个特定用户登录一台 Unix 计算机一样。在 SQL 环境中，活动的数据库用户名决定对数据库对象的访问权限 — 详见第 21 章因此，它本质上是哪些数据库用户可以连接。

> **注意**
>
> 如第 21 章所释，PostgreSQL实际上以“角色”来进行权限管理。在本章中，我们用数据库用户表示“拥有LOGIN权限的角色”。

认证是数据库服务器建立客户端身份的过程，并且服务器决定客户端应用（或者运行客户端应用的用户）是否被允许以请求的数据库用户名来连接。

PostgreSQL提供多种不同的客户端认证方式。被用来认证一个特定客户端连接的方法可以基于（客户端）主机地址、数据库和用户来选择。

PostgreSQL数据库用户名在逻辑上是和服务器运行的操作系统中的用户名相互独立的。如果一个特定服务器的所有用户在那台服务器的机器上也有帐号，那么分配与操作系统用户名匹配的数据库用户名是有意义的。不过，一个接受远程连接的服务器可能有许多没有本地操作系统帐号的用户，并且在这种情况下数据库用户和操作系统用户名之间不必有任何联系。

20.1. pg_hba.conf文件

客户端认证是由一个配置文件（通常名为pg_hba.conf并被存放在数据库集簇目录中）控制（HBA表示基于主机的认证）。在initdb初始化数据目录时，它会安装一个默认的pg_hba.conf文件。不过我们也可以把认证配置文件放在其它地方； 参阅hba_file配置参数。

pg_hba.conf文件的常用格式是一组记录，每行一条。空白行将被忽略， #注释字符后面的任何文本也被忽略。记录不能跨行。一条记录由若干用空格 和/或制表符分隔的域组成。如果域值用双引号包围，那么它可以包含空白。在数据库、用户或地址域中 引用一个关键字（例如，all或replication）将使该词失去其特殊 含义，并且只是匹配一个有该名字的数据库、用户或主机。

每条记录指定一种连接类型、一个客户端 IP 地址范围（如果和连接类型相关）、一个数据库名、一个用户名以及对匹配这些参数的连接使用的认证方法。第一条匹配连接类型、客户端地址、连接请求的数据库和用户名的记录将被用于执行认证。这个过程没有“落空”或者“后备”的说法：如果选择了一条记录而且认证失败，那么将不再考虑后面的记录。如果没有匹配的记录，那么访问将被拒绝。

记录可以是下面七种格式之一：

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

各个域的含义如下：

local

这条记录匹配企图使用 Unix 域套接字的连接。如果没有这种类型的记录，就不允许Unix 域套接字连接。

host

这条记录匹配企图使用 TCP/IP 建立的连接。host记录匹配SSL和非SSL的连接尝试。

> **注意**
>
> 除非服务器带着合适的listen_addresses配置参数值启动，否则将不可能进行远程的 TCP/IP 连接，因为默认的行为是只监听在本地环回地址localhost上的 TCP/IP 连接。

hostssl

这条记录匹配企图使用 TCP/IP 建立的连接，但必须是使用SSL加密的连接。

要使用这个选项，编译服务器的时候必须打开SSL支持。此外，在服务器启动的时候必须通过设置ssl配置参数（详见第 18.9 节打开SSL。否则，hostssl记录会被忽略，并且会记录一个警告说它无法匹配任何连接。

hostnossl

这条记录的行为与hostssl相反；它只匹配那些在 TCP/IP上不使用SSL的连接企图。

database

指定记录所匹配的数据库名称。值all指定该记录匹配所有数据库。值 sameuser指定如果被请求的数据库和请求的用户同名，则匹配。值samerole指定请求的用户必须是一个与数据库同名的角色中的成员（samegroup是一个已经废弃了，但目前仍然被接受的samerole同义词）。对于一个用于samerole目的的角色，超级用户不会被考虑为其中的成员，除非它们是该角色的显式成员（直接或间接），而不是由于超级用户的原因。值replication指定如果一个物理复制连接被请求则该记录匹配（注意复制连接不指定任何特定的数据库）。在其它情况里，这就是一个特定的PostgreSQL数据库名字。可以通过用逗号分隔的方法指定多个数据库，也可以通过在文件名前面放@来指定一个包含数据库名的文件。

user

指定这条记录匹配哪些数据库用户名。值all指定它匹配所有用户。否则，它要么是一个特定数据库用户的名字或者是一个有前导+的组名称（回想一下，在PostgreSQL里，用户和组没有真正的区别，+实际表示“匹配这个角色的任何直接或间接成员角色”，而没有+记号的名字只匹配指定的角色）。出于这个目的，如果超级用户显式的是一个角色的成员（直接或间接），那么超级用户将只被认为是该角色的一个成员而不是作为一个超级用户。多个用户名可以通过用逗号分隔的方法提供。一个包含用户名的文件可以通过在文件名前面加上@来指定。

address

指定这个记录匹配的客户端机器地址。这个域可以包含一个主机名、一个 IP 地址范围或下文提到的特殊关键字之一。

一个 IP 地址范围以该范围的开始地址的标准数字记号指定，然后是一个斜线（/） 和一个CIDR掩码长度。掩码长度表示客户端 IP 地址必须匹配的高序二进制位位数。在给出的 IP 地址中，这个长度的右边的二进制位必须为零。 在 IP 地址、/和 CIDR 掩码长度之间不能有空白。

这种方法指定一个 IPv4 地址范围的典型例子是： 172.20.143.89/32用于一个主机，172.20.143.0/24用于一个小型网络， 10.6.0.0/16用于一个大型网络。 一个单主机的IPv6 地址范围看起来像这样：::1/128（IPv6 回环地址）， 一个小型网络的 IPv6 地址范围则类似于：fe80::7a31:c1ff:0000:0000/96。 0.0.0.0/0表示所有 IPv4 地址，并且::0/0表示所有 IPv6 地址。要指定一个单一主机，IPv4 用一个长度为 32 的 CIDR掩码或者 IPv6 用 长度为 128 的 CIDR 掩码。在一个网络地址中，不要省略结尾的零。

一个以 IPv4 格式给出的项将只匹配 IPv4 连接并且一个以 IPv6 格式给出的项将只匹配 IPv6 连接，即使对应的地址在 IPv4-in-IPv6 范围内。请注意如果系统的 C 库不支持 IPv6 地址，那么 IPv6 格式中的项将被拒绝。

你也可以写all来匹配任何 IP 地址、写samehost来匹配任何本服务器自身的 IP 地址或者写samenet来匹配本服务器直接连接到的任意子网的任意地址。

若果指定了一个主机名（任何除 IP 地址单位或特殊关键字之外的都被作为主机名处理）， 该名称会与客户端的 IP 地址的反向名字解析（例如使用 DNS 时的反向DNS 查找）结果进行比较。主机名比较是大小写敏感的。如果匹配上，那么将在主机名上执行一次正向名字解析（例如正向 DNS 查找）来检查它解析到的任何地址是否等于客户端的 IP 地址。如果两个方向都匹配，则该项被认为匹配（pg_hba.conf中使用的主机名应该是客户端 IP 地址的地址到名字解析返回的结果，否则该行将不会匹配。某些主机名数据库允许将一个 IP 地址关联多个主机名，但是当被要求解析一个 IP 地址时，操作系统将只返回一个主机名）。

一个以点号（.）开始的主机名声明匹配实际主机名的后缀。因此.example.com将匹配foo.example.com（但不匹配example.com）。

当主机名在pg_hba.conf中被指定时，你应该保证名字解析很快。建立一个类似nscd的本地名字解析缓存是一种不错的选择。另外，你可能希望启用配置参数log_hostname来在日志中查看客户端的主机名而不是 IP 地址。

这个域只适用于host、 hostssl和hostnossl记录。

> **注意**
>
> 用户有时候会疑惑为什么这样处理的主机名看起来很复杂，因为需要两次名字解析（包括一次 客户端 IP 地址的反向查找）。在客户端的反向 DNS 项没有建立或者得到某些意料之外的主机 名的情况下，这种方式会让该特性的使用变得复杂。这样做主要是为了效率：通过这种方式，一次 连接尝试要求最多两次解析器查找，一次逆向以及一次正向。如果有一个解析器对于该地址有问 题，这仅仅是客户端的问题。一种假想的替代实现是只做前向查找，这种方法不得不在每一次连接 尝试期间解析pg_hba.conf中提到的每一个主机名。如果列出了很多 名称，这就会很慢。并且如果主机名之一有解析器问题，它会变成所有人的问题。
>
> 另外，一次反向查找也是实现后缀匹配特性所需的，因为需要知道实际的客户端主机名来与模式进行匹配。
>
> 注意这种行为与其他流行的基于主机名的访问控制实现相一致，例如Apache HTTP Server 和 TCP Wrappers。

IP-address
IP-mask

这两个域可以被用作IP-address/ mask-length记号法的替代方案。和指定掩码长度不同，实际的掩码被指 定在一个单独的列中。例如，255.0.0.0表示 IPv4 CIDR 掩码长度8，而255.255.255.255表示 CIDR 掩码长度 32。

这些域只适用于host、hostssl和hostnossl记录。

auth-method

指定当一个连接匹配这个记录时，要使用的认证方法。下面对可能的选择做了概述，详见第 20.3 节

trust

无条件地允许连接。这种方法允许任何可以与PostgreSQL数据库服务器连接的用户以他们期望的任意PostgreSQL数据库用户身份登入，而不需要口令或者其他任何认证。详见第 20.4 节

reject

无条件地拒绝连接。这有助于从一个组中“过滤出”特定主机，例如一个reject行可以阻塞一个特定的主机连接，而后面一行允许一个特定网络中的其余主机进行连接。

scram-sha-256

执行SCRAM-SHA-256认证来验证用户的口令。详见第 20.5 节

md5

执行SCRAM-SHA-256或MD5认证来验证用户的口令。详见第 20.5 节

password

要求客户端提供一个未加密的口令进行认证。因为口令是以明文形式在网络上发送的，所以我们不应该在不可信的网络上使用这种方式。详见第 20.5 节

gss

用 GSSAPI 认证用户。只对 TCP/IP 连接可用。详见第 20.6 节

sspi

用 SSPI 来认证用户。只在 Windows 上可用。详见第 20.7 节

ident

通过联系客户端的 ident 服务器获取客户端的操作系统名，并且检查它是否匹配被请求的数据库用户名。Ident 认证只能在 TCIP/IP 连接上使用。当为本地连接指定这种认证方式时，将用 peer 认证来替代。详见第 20.8 节

peer

从操作系统获得客户端的操作系统用户，并且检查它是否匹配被请求的数据库用户名。这只对本地连接可用。详见第 20.9 节

ldap

使用LDAP服务器认证。详见第 20.10 节

radius

用 RADIUS 服务器认证。详见第 20.11 节

cert

使用 SSL 客户端证书认证。详见第 20.12 节

pam

使用操作系统提供的可插入认证模块服务（PAM）认证。详见第 20.13 节

bsd

使用由操作系统提供的 BSD 认证服务进行认证。详见第 20.14 节

auth-options

在auth-method域的后面，可以是形如name=value的域，它们指定认证方法的选项。关于哪些认证方法可以用哪些选项的细节请见下文。

除了下文列出的与方法相关的选项之外，还有一个与方法无关的认证选项clientcert，它可以在任何hostssl记录中指定。当被设置为1时，这个选项要求客户端在认证方法的其他要求之外出示一个有效的（可信的）SSL 证书。

用@结构包括的文件被读作一个名字列表，它们可以用空白或者逗号分隔。注释用#引入，就像在pg_hba.conf中那样，并且允许嵌套@结构。除非跟在@后面的文件名是一个绝对路径，文件名都被认为是相对于包含引用文件的目录。

因为每一次连接尝试都会顺序地检查pg_hba.conf记录，所以这些记录的顺序是非常关键的。通常，靠前的记录有比较严的连接匹配参数和比较弱的认证方法，而靠后的记录有比较松的匹配参数和比较强的认证方法。 例如，我们希望对本地 TCP/IP 连接使用trust认证，而对远程 TCP/IP 连接要求口令。在这种情况下为来自于 127.0.0.1 的连接指定trust认证的记录将出现在为一个更宽范围的客户端 IP 地址指定口令认证的记录前面。

在启动以及主服务器进程收到SIGHUP信号时，pg_hba.conf文件会被读取。 如果你在活动的系统上编辑了该文件，你将需要通知 postmaster（使用pg_ctl reload或kill -HUP）重新读取该文件。

> **注意**
>
> 前面的说明在Microsoft Windows上不为真：在Windows上，pg_hba.conf文件中的任何更改会立即被应用到后续的新连接上。

系统视图pg_hba_file_rules有助于预先测试对pg_hba.conf文件的更改，该视图也可以在该文件的装载没有产生预期效果时用于诊断问题。该视图中带有非空error域的行就表示该文件对应行中存在问题。

> **提示**
>
> 要连接到一个特定数据库，一个用户必须不仅要通过pg_hba.conf检查，还必须要有该数据库上的CONNECT权限。如果你希望限制哪些用户能够连接到哪些数据库，授予/撤销CONNECT权限通常比在pg_hba.conf项中设置规则简单。

例 20.1中展示了pg_hba.conf项的一些例子。不同认证方法的详情请见下一节。

例 20.1. 示例 pg_hba.conf 项

```
# 允许本地系统上的任何用户
# 通过 Unix 域套接字以任意
# 数据库用户名连接到任意数据库
#（本地连接的默认值）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# 相同的规则，但是使用本地环回 TCP/IP 连接。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
```

```
host    all             all             127.0.0.1/32            trust

# 和前一行相同，但是使用了一个独立的掩码列
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK
 METHOD
host    all             all             127.0.0.1       255.255.255.255
 trust

# IPv6 上相同的规则
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# 使用主机名的相同规则（通常同时覆盖 IPv4 和 IPv6）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# 允许来自任意具有 IP 地址
# 192.168.93.x 的主机上任意
# 用户以 ident 为该连接所
# 报告的相同用户名连接到
# 数据库 "postgres"
#（通常是操作系统用户名）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident

# 如果用户的口令被正确提供，
# 允许来自主机 192.168.12.10
# 的任意用户连接到数据库 "postgres"。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256

# 如果用户的口令被正确提供，
# 允许 example.com 中主机上
# 的任意用户连接到任意数据库。
#
# 为大部分用户要求SCRAM认证，但是用户'mike'是个例外，
# 他使用的是不支持SCRAM认证的旧客户端。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256

# 如果没有前面的 "host" 行，这两
# 行将拒绝所有来自 192.168.54.1
# 的连接（因为那些项将首先被匹配），
# 但是允许来自互联网其他任何地方的
# GSSAPI 连接。零掩码导致主机
# IP 地址中的所有位都不会被考虑，
# 因此它匹配任意主机。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss
```

```
# 允许来自 192.168.x.x 主机的用户
# 连接到任意数据库，如果它们能够
# 通过 ident 检查。例如，假设 ident
# 说用户是 "bryanh" 并且他要求以
# PostgreSQL 用户 "guest1" 连接，
# 如果在 pg_ident.conf 有一个映射
# "omicron" 的选项说 "bryanh" 被
# 允许以 "guest1" 连接，则该连接将被允许。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident
 map=omicron

# 如果这些是本地连接的唯一三行，
# 它们将允许本地用户只连接到它们
# 自己的数据库（与其数据库用户名
# 同名的数据库），不过管理员和角
# 色 "support" 的成员除外（它们可
# 以连接到所有数据库）。文件
# $PGDATA/admins 包含一个管理员
# 名字的列表。在所有情况下都要求口令。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# 上面的最后两行可以被整合为一行：
local   all             @admins,+support                        md5

# 数据库列也可以用列表和文件名：
local   db1,db2,@demodbs  all                                   md5
```

20.2. 用户名映射

当使用像 Ident 或者 GSSAPI 之类的外部认证系统时，发起连接的操作系统用户名可能不同于要被使用的数据库用户（角色）。在这种情况下，一个用户名映射可被用来把操作系统用户名映射到数据库用户。要使用用户名映射，在pg_hba.conf的选项域指定map=map-name。此选项支持所有接收外部用户名的认证方法。由于不同的连接可能需要不同的映射，在pg_hba.conf中的map-name参数中指定要被使用的映射名，用以指示哪个映射用于每个个体连接。

用户名映射定义在 ident 映射文件中，默认情况下它被命名为pg_ident.conf并被存储在集簇的数据目录中（不过，可以把该映射文件放在其他地方，见ident_file配置参数）。ident 映射文件包含的行的一般格式：

```
map-name system-username database-username
```

以在pg_hba.conf中同样的方式处理注释和空白。map-name是一个任意名称，它将被用于在pg_hba.conf中引用该映射。其他两个域指定一个操作系统用户名和一个匹配的数据库用户名。相同的map-name可以被反复地用在同一个映射中指定多个用户映射。

对于一个给定操作系统用户可以对应多少个数据库用户没有限制，反之亦然。因此，一个映射中的项应该被看成意味着“这个操作系统用户被允许作为这个数据库用户连接”，而不是按时它们等价。如果有任何映射项把从外部认证系统获得的用户名和用户要求的数据库用户名配对，该连接将被允许。

如果system-username域以一个斜线（/）开始，域的剩余部分被当做一个正则表达式（PostgreSQL的正则表达式语法详见第 9.7.3.1 节。正则表达式可以包括一个单一的捕获，或圆括号子表达式，然后它可以在database-username域中以\1（反斜线一）被引用。这允许在单个行中多个用户名的映射，这特别有助于简单的语法替换。例如，这些项

```
mymap   /^(.*)@mydomain\.com$       \1
mymap   /^(.*)@otherdomain\.com$    guest
```

将为用户移除以@mydomain.com结束的系统用户名的域部分，以及允许系统名以@otherdomain.com结束的任意用户作为guest登入。

> **提示**
>
> 记住在默认情况下，一个正则表达式可以只匹配字符串的一部分。如上例所示，使用^和$来强制匹配整个系统用户名通常是明智的。

在启动以及主服务器进程收到SIGHUP信号时，pg_ident.conf文件会被读取。 如果你在活动的系统上编辑了该文件，你将需要通知 postmaster（使用pg_ctl reload或kill -HUP）重新读取该文件。

例 20.2中展示了一个可以联合pg_hba.conf文件（例 20.1） 使用的pg_ident.conf文件。在这个例子中，对于任何登入到 192.168 网络上的一台机器的用户， 如果该用户没有操作系统用户名bryanh、ann或robert，则他不会被授予访问权限。只有当 Unix 用户robert尝试作为PostgreSQL用户bob（而不是作为robert或其他人）连接时，他才被允许访问。ann只被允许作为ann连接。用户bryanh被允许以bryanh或者guest1连接。

例 20.2. 一个示例pg_ident.conf 文件

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# bob 在这些机器上有用户名 robert
omicron         robert                  bob
# bryanh 也可以作为 guest1 连接
omicron         bryanh                  guest1
```

20.3. 认证方法

下列小节更详细地描述认证方法。

20.4. 信任认证

当trust认证被指定时，PostgreSQL假设任何可以连接到服务器的人都被授权使用他们指定的任何数据库用户名（即使是超级用户）访问数据库。当然，在database和 user列中设置的限制仍然适用。只有当在操作系统层对进入服务器的连接有足够保护时，才应该使用这种方法。

trust认证对于单用户工作站的本地连接是非常合适和方便的。通常它本身不适用于一台多用户机器。不过，只要你利用文件系统权限限制了对服务器的 Unix 域套接字文件的访问，即使在多用户机器上，你也可以使用trust。 要做这些限制，你可以设置第 19.3 节中描述的unix_socket_permissions配置参数（可能还有unix_socket_group）。 或者你可以设置unix_socket_directories配置参数来把 Unix 域套接字文件放在一个经过恰当限制的目录中。

设置文件系统权限只能有助于 Unix 套接字连接。本地 TCP/IP 连接不会被文件系统权限限制。因此，如果你想利用文件系统权限来控制本地安全，那么从pg_hba.conf中移除host ... 127.0.0.1 ...行，或者把它改为一个非trust认证方法。

如果通过指定trust的pg_hba.conf行让你信任每一个被允许连接到服务器的机器上的用户，trust认证只适合 TCP/IP 连接。为任何不是来自localhost（127.0.0.1）的 TCP/IP 连接使用trust很少是合理的。

20.5. 口令认证

有几种基于口令的认证方法。这些方法的过程类似，但是区别在于用户口令如何被存放在服务器上以及客户端提供的口令如何被通过连接发送。

scram-sha-256

方法scram-sha-256按照RFC 7677[1]中的描述执行SCRAM-SHA-256认证。它使用的是一种挑战-响应的方案，可以防止在不可信连接上对口令的嗅探并且支持在服务器上以一种加密哈希的方式存放口令，因此被认为是安全的。

这是当前提供的方法中最安全的一种，但是旧的客户端库不支持这种方法。

md5

方法md5使用一种自定义的安全性较低的挑战-响应机制。它能防止口令嗅探并且防止口令在服务器上以明文存储，但是无法保护攻击者想办法从服务器上窃取了口令哈希的情况。此外，现在认为MD5哈希算法对于确定攻击已经不再安全。

md5方法不能与db_user_namespace特性一起使用。

为了简化从md5方法到较新的SCRAM方法的转变，如果在pg_hba.conf中指定了md5但是用户在服务器上的口令是为SCRAM（见下文）加密的，则将自动选择基于SCRAM的认证。

password

方法password以明文形式发送口令，因此它对于口令“嗅探”攻击很脆弱。如果可能应该尽量避免使用它。不过，如果连接被SSL加密保护着，那么可以安全地使用password（不过如果依靠SSL，SSL证书认证可能是更好的选择）。

PostgreSQL数据库口令独立于操作系统用户口令。每个数据库用户的口令被存储在pg_authid系统目录中。口令可以用 SQL 命令CREATE USER和ALTER ROLE管理，例如CREATE ROLE foo WITH LOGIN PASSWORD 'secret'或者psql的\password命令。如果没有为一个用户设置口令，那么存储的口令为空并且对该用户的口令认证总会失败。

不同的基于口令的认证方法的可用性取决于用户的口令在服务器上是如何被加密（或者更准确地说是哈希）的。这由设置口令时的配置参数password_encryption控制。如果口令使用scram-sha-256设置加密，那么它可以被用于认证方法scram-sha-256和password（但后一种情况中口令将以明文传输）。如上所释，在这种情况下，指定的认证方法md5将自动切换到使用scram-sha-256方法。如果口令使用md5设置加密，那么它仅能用于md5和password认证方法说明（同样，后一种情况中口令以明文传输）。（之前的PostgreSQL发行版支持在服务器上存储明文口令。现在已经不可能了）。要检查当前存储的口令哈希，可以参考系统目录pg_authid。

要把现有的安装从md5升级到scram-sha-256，可以在确保所有在用的客户端已经足以支持SCRAM之后，在postgresql.conf中设置password_encryption = 'scram-sha-256'，然后让所有用户设置新口令并且在pg_hba.conf中将认证方法说明改为scram-sha-256.

20.6. GSSAPI 认证

[1] https://tools.ietf.org/html/rfc7677

GSSAPI是用于 RFC 2743 中定义的安全认证的一个工业标准协议。PostgreSQL根据 RFC 1964 支持带Kerberos认证的GSSAPI。GSSAPI为支持它的系统提供自动认证（单点登录）。认证本身是安全的，但通过数据库连接发送的数据将不被加密，除非使用SSL。

当编译PostgreSQL时，GSSAPI 支持必须被启用，详见第 16 章

当GSSAPI使用Kerberos时， 它会使用格式为 servicename/hostname@realm的标准principal。 PostgreSQL服务器将接受该服务器所使用的 keytab 中包括的任何 principal，但是在从使用 krbsrvname连接参数的客户端建立连接时要注意指定正确的principal 细节（另见 第 34.1.2 节。安装的默认值postgres 可以在编译时使用 ./configure --with-krb-srvnam=其他值修改。 在大部分的环境中，这个参数从不需要被更改。某些 Kerberos 实现可能要求一个不同的服务名， 例如 Microsoft Active Directory 要求服务名是大写形式（POSTGRES）。

hostname是服务器机器的被完全限定的主机名。服务 principal 的 realm 是该服务器机器的首选 realm。

客户端 principal 可以被通过pg_ident.conf映射到不同的 PostgreSQL数据库用户名。例如， pgusername@realm可能会被映射到pgusername。 或者，你可以使用完整的username@realm当事人作为 PostgreSQL中的角色而无需任何映射。

PostgreSQL也支持一个参数把 realm 从 principal 中剥离。这种方法是为了向后兼容性，并且我们强烈反对使用它，因为这样就无法区分具有相同用户名却来自不同 realm 的不同用户了。要启用这种方法，可将include_realm设置为 0。对于简单的单 realm 安装，这样做并且设置krb_realm参数（这会检查 principal 的 realm 是否正好匹配krb_realm中的参数）仍然是安全的。但比起在pg_ident.conf中指定一个显式映射来说，这种方法的能力较低。

确认你的服务器的 keytab 文件是可以被PostgreSQL服务器帐 户读取的（最好是只读的）（又见第 18.1 节。密钥文件的位置由配置 参数krb_server_keyfile指定。默认是/usr/local/pgsql/etc/krb5.keytab（或者任何在编译的时候作为sysconfdir的目录）。 出于安全原因，推荐对PostgreSQL服务器使用一个独立 的 keytab而不是开放系统 keytab 文件的权限。

keytab 文件由 Kerberos 软件生成，详见 Kerberos 文档。下面是 MIT 兼容的 Kerberos 5 实现的例子：

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

当连接到数据库时，确保你有一个匹配被请求数据库用户名的 principal 的票据。例如，对于数据库用户名fred，principal fred@EXAMPLE.COM将能够连接。要也允许 principal fred/users.example.com@EXAMPLE.COM，可使用一个用户名映射，如第 20.2 节所述。

下列被支持的配置选项用于GSSAPI：

include_realm

如果设置为 0，在通过用户名映射之前（第 20.2 节，来自已认证用户 principal 的 realm 名称会被剥离掉。我们不鼓励这样做，这种方法主要是为了向后兼容性而存在的，因为它在多 realm 环境中是不安全的（除非也使用 krb_realm）。推荐用户让include_realm设置为默认值（1）并且在pg_ident.conf中提供一条显式的映射来把 principal 名称转换成PostgreSQL用户名。

map

允许在系统和数据库用户名之间的映射。详见第 20.2 节 对于一个 GSSAPI/Kerberos 原则，例如username@EXAMPLE.COM （或者更不常见的username/hostbased@EXAMPLE.COM）， 用于映射的用户名会是username@EXAMPLE.COM（或者username/hostbased@EXAMPLE.COM，相应地），除非 include_realm已经被设置为 0，

在那种情况下 username（或者username/hostbased）是 映射时被视作系统用户名的东西。

krb_realm

设置 realm 为对用户 principal 名进行匹配的范围。 如果这个参数被设置，只有那个 realm 的用户将被接受。 如果它没有被设置，任何 realm 的用户都能连接，服从任何已完成的用户名映射。

20.7. SSPI 认证

SSPI是一种用于带单点登录的安全认证的Windows技术。 PostgreSQL在negotiate模式中将使用 SSPI，它在可能的情况下使用Kerberos并在其他情况下自动降回到NTLM。只有在服务器和客户端都运行着Windows时，SSPI才能工作。或者在非 Windows 平台上GSSAPI可用时，SSPI也能工作。

当使用Kerberos认证时，SSPI和GSSAPI的工作方式相同，详见第 20.6 节

下列被支持的配置选项用于SSPI：

include_realm

如果设置为 0，在通过用户名映射之前（第 20.2 节，来自已认证用户 principal 的 realm 名称会被剥离掉。我们不鼓励这样做，这种方法主要是为了向后兼容性而存在的，因为它在多 realm 环境中是不安全的（除非也使用krb_realm）。推荐用户让 include_realm 设置为默认值（1）并且在pg_ident.conf中提供一条显式的映射来把 principal 名称转换成PostgreSQL用户名。

compat_realm

如果被设置为 1，该域的 SAM 兼容名称（也被称为 NetBIOS 名称）被用于include_realm选项。这是默认值。如果被设置为 0，会使用来自 Kerberos 用户主名的真实 realm 名称。

不要禁用这个选项，除非你的服务器运行在一个域账号（这包括一个域成员系统上的虚拟服务账号）下并且所有通过 SSPI 认证的所有客户端也在使用域账号，否则认证将会失败。

upn_username

如果这个选项和compat_realm一起被启用，来自 Kerberos UPN 的用户名会被用于认证。如果它被禁用（默认），会使用 SAM 兼容的用户名。默认情况下，对于新用户账号这两种名称是一样的。

注意如果没有显式指定用户名，libpq会使用 SAM 兼容的名称。如果你使用的是libpq或者基于它的驱动，你应该让这个选项保持禁用或者在连接字符串中显式指定用户名。

map

允许在系统和数据库用户名之间的映射。详见第 20.2 节 对于一个 GSSAPI/Kerberos 原则，例如username@EXAMPLE.COM （或者更不常见的username/hostbased@EXAMPLE.COM）， 用于映射的用户名会是username@EXAMPLE.COM（或者 username/hostbased@EXAMPLE.COM，相应地），除非 include_realm已经被设置为 0，在那种情况下 username（或者username/hostbased）是 映射时被视作系统用户名的东西。

krb_realm

设置领域为对用户 principal 名进行匹配的范围。如果这个参数被设置，只有那个领域的用户将被接受。如果它没有被设置，任何领域的用户都能连接，服从任何已完成的用户名映射。

20.8. Ident 认证

ident 认证方法通过从一个 ident 服务器获得客户端的操作系统用户名并且用它作为被允许的数据库用户名（和可选的用户名映射）来工作。它只在 TCP/IP 连接上支持。

> **注意**
>
> 当为一个本地（非 TCP/IP）连接指定 ident 时，将实际使用 peer 认证（见第 20.9 节。

下列被支持的配置选项用于ident：

map

允许系统和数据库用户名之间的映射。详见第 20.2 节

“Identification Protocol（标识协议）”在 RFC 1413 中描述。实际上每个类 Unix 操作系统都带着一个默认监听 TCP 113 端口的 ident 服务器。ident 服务器的基本功能是回答类似这样的问题：“哪个用户从你的端口X发起了连接并且连到了我的端口Y？” 。因为当一个物理连接被建立后，PostgreSQL既知道X也知道Y， 所以它可以询问尝试连接的客户端主机上的 ident 服务器并且在理论上可以判断任意给定连接的操作系统用户。

这个过程的缺点是它依赖于客户端的完整性：如果客户端机器不可信或者被攻破，攻击者可能在 113 端口上运行任何程序并且返回他们选择的任何用户。因此这种认证方法只适用于封闭的网络， 这样的网络中的每台客户端机器都处于严密的控制下并且数据库和操作系统管理员操作时可以方便地联系。换句话说，你必须信任运行 ident 服务器的机器。注意这样的警告：

> 标识协议的本意不是作为一种认证或访问控制协议。
>
> —RFC 1413

有些 ident 服务器有一个非标准的选项，它导致返回的用户名是被加密的，使用的是只有原机器管理员知道的一个密钥。当与PostgreSQL配合使用 ident 服务器时，一定不要使用这个选项，因为PostgreSQL没有任何方法对返回的字符串进行解密以获取实际的用户名。

20.9. Peer 认证

Peer 认证方法通过从内核获得客户端的操作系统用户名并把它用作被允许的数据库用户名（和可选的用户名映射）来工作。这种方法只在本地连接上支持。

下列被支持的配置选项用于peer：

map

允许在系统和数据库用户名之间的映射。详见第 20.2 节

Peer 认证只在提供getpeereid()函数、SO_PEERCRED套接字参数或相似机制的操作系统上可用。这些 OS 当前包括Linux、大部分的BSD包括OS X以及Solaris。

20.10. LDAP 认证

这种认证方法操作起来类似于password，只不过它使用 LDAP 作为密码验证方法。LDAP 只被用于验证用户名/口令对。因此，在使用 LDAP 进行认证之前，用户必须已经存在于数据库中。

LDAP 认证可以在两种模式下操作。在第一种模式中（我们将称之为简单绑定模式），服务器将绑定到构造成prefix username suffix的可区分名称。通常，prefix参数被用于指定 cn=或者一个活动目录环境中的DOMAIN\。suffix被用来指定非活动目录环境中的DN的剩余部分。

在第二种模式中（我们将称之为搜索与绑定模式），服务器首先用一个固定的用户名和密码（用ldapbinddn和ldapbindpasswd指定）绑定到 LDAP 目录 ，并为试图登入该数据库的用户执行一次搜索。如果没有配置用户名和密码， 将尝试一次匿名绑定到目录。搜索将在位于ldapbasedn的子树上被执行，并将尝试做一次ldapsearchattribute中指定属性的精确匹配。一旦在这次搜索中找到用户，服务器断开并且作为这个用户重新绑定到目录，使用由客户端指定的口令来验证登录是正确的。这种模式与在其他软件中的 LDAP 认证所使用的相同，例如 Apache mod_authnz_ldap 和 pam_ldap。这种方法允许位于目录中用户对象的更大灵活性，但是会导致建立两个到 LDAP 服务器的独立连接。

下列配置选项被用于两种模式：

ldapserver

要连接的LDAP服务器的名称或IP地址。可以指定多个服务器，用空格分隔。

ldapport

要连接的LDAP服务器的端口号。如果没有指定端口，LDAP库的默认端口设置将被使用。

ldapscheme

设置为ldaps可以使用LDAPS。这是一种非标准的在SSL之上使用LDAP的方法，在有一些LDAP服务器实现上可以支持。其他选择还可以参考ldaptls选项。

ldaptls

设置为1以使PostgreSQL和LDAP服务器之间的连接使用TLS加密。这会按照RFC 4513使用StartTLS操作。其他选择还可以参考ldapscheme选项。

注意使用ldapscheme或ldaptls仅会加密PostgreSQL服务器和LDAP服务器之间的通信。PostgreSQL服务器和PostgreSQL客户端之间的连接仍是未加密的，除非也在其上使用SSL。

下列选项只被用于简单绑定模式：

ldapprefix

当做简单绑定认证时，前置到用户名形成要用于绑定的DN的字符串。

ldapsuffix

当做简单绑定认证时，前置到用户名形成要用于绑定的DN的字符串。

下列选项只被用于搜索与绑定模式：

ldapbasedn

当做搜索与绑定认证时，开始搜索用户的根DN。

ldapbinddn

当做搜索与绑定认证时，用户要绑定到目录开始执行搜索的DN。

ldapbindpasswd

当做搜索与绑定认证时，用户用于绑定到目录开始执行搜索的口令。

ldapsearchattribute

当做搜索与绑定认证时，在搜索中用来与用户名匹配的属性。如果没有指定属性，将会使用uid属性。

ldapsearchfilter

在做search+bind认证时使用的搜索过滤器。$username的出现将被替换为用户名。这允许比ldapsearchattribute更加灵活的搜索过滤器。

ldapurl

一个RFC 4516 LDAP URL。这是一种用更紧凑和标准的形式书写某些其他LDAP选项的可选方法。格式是

```
ldap[s]://host[:port]/basedn[?[attribute][?[scope][?[filter]]]]
```

scope必须是base、one、sub之一，通常是最后一个（默认是base，但它在这个应用中通常没啥用）。attribute可以指定一个属性，在这种情况中它被用作ldapsearchattribute的一个值。如果attribute为空，那么filter可以被用作ldapsearchfilter的一个值。

URL模式ldaps选择LDAPS方法来在SSL上建立LDAP连接，等效于使用ldapscheme=ldaps。要使用StartTLS操作加密LDAP连接，可以用普通的URL模式ldap并且在ldapurl之外指定ldaptls选项。

对于非匿名绑定，ldapbinddn和ldapbindpasswd必须被指定为独立选项。

LDAP URL 当前只支持 OpenLDAP，而不支持 Windows。

将简单绑定的选项中混合用于搜索与绑定的选项是一种错误。

在使用search+bind模式时，可以用ldapsearchattribute指定的单个属性执行搜索，或者使用ldapsearchfilter指定的自定义搜索过滤器执行搜索。指定ldapsearchattribute=foo等效于指定ldapsearchfilter="(foo=$username)"。如果两个选项都没有被指定，则默认为ldapsearchattribute=uid。

这里是一个简单绑定 LDAP 配置的例子：

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=",
 dc=example, dc=net"
```

当请求一个作为数据库用户someuser到数据库服务器的连接时，PostgreSQL 将尝试使用cn=someuser, dc=example, dc=net和客户端提供的口令来绑定到 LDAP 服务器。如果那个连接成功，将被授予数据库访问。

这里是一个搜索与绑定配置的例子：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
 ldapsearchattribute=uid
```

当请求一个作为数据库用户someuser到数据库服务器的连接时，PostgreSQL 将尝试匿名绑定（因为没有指定ldapbinddn）到 LDAP 服务器，在指定的基础 DN 下执行一次对于(uid=someuser)的搜索。如果找到一个项，则它将尝试使用找到的信息和客户端提供的口令进行绑定。如果第二个连接成功，将被授予数据库访问。

这里是被写成一个 URL 的相同搜索与绑定配置：

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

一些支持根据 LDAP 认证的其他软件使用相同的 URL 格式，因此很容易共享该配置。

这里是一个search+bind配置的例子，它使用ldapsearchfilter而不是ldapsearchattribute来允许用用户ID或电子邮件地址进行认证：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
 ldapsearchfilter="(|(uid=$username)(mail=$username))"
```

> **提示**
>
> 如例子中所示，由于 LDAP 通常使用逗号和空格来分割一个 DN 的不同部分，在配置 LDAP 选项时通常有必要使用双引号包围的参数值。

20.11. RADIUS 认证

这种认证方法的操作类似于password，不过它使用 RADIUS 作为密码验证方式。RADIUS 只被用于验证 用户名/密码对。因此，在 RADIUS 能被用于认证之前，用户必须已经存在于数据库中。

当使用 RADIUS 认证时，一个访问请求消息将被发送到配置好的 RADIUS 服务器。这一请求将是Authenticate Only类型，并且包含参数user name、password（加密的）和NAS Identifier。该请求将使用一个与服务器共享的密钥加密。RADIUS 服务器将对这个服务器响应Access Accept或者Access Reject。不支持RADIUS accounting。

可以指定多个RADIUS服务器，这种情况下将会依次尝试它们。如果从一台服务器接收到否定响应，则认证失败。如果没有接收到响应，则将会尝试列表中的下一台服务器。要指定多台服务器，可将服务器名放在引号内并且用逗号分隔开。如果指定了多台服务器，所有其他RADIUS选项也可以以逗号分隔的列表给出，用来为每台服务器应用个别的值。也可以把选项指定为一个单一值，这样该值将被应用到所有的服务器。

下列被支持的配置选项用于 RADIUS：

radiusservers

连接到 RADIUS 服务器的名称或IP地址。此参数是必需的。

radiussecrets

和 RADIUS 服务器秘密交谈时会用到共享密钥。这在 PostgreSQL 和 RADIUS 服务器之间必须有完全相同的值。我们推荐用一个至少 16 个字符的字符串。这个参数是必需的。

> **注意**
>
> 如果PostgreSQL编译为支持OpenSSL，所用的加密向量将只是强密码。在其他情况下，到 RADIUS 服务器的传输应该被视为应该被视为被混淆的、不安全的。如有必要，应采用外部安全措施。

radiusports

用于连接到 RADIUS 服务器的端口号。如果没有指定端口，则使用默认端口1812。

radiusidentifiers

在 RADIUS 请求中字符串被用作NAS Identifier。 这个参数可以被用作第二个参数标识例如该用户试图以哪个数据库用户进行认证，它可以被用于 RADIUS 服务器上的策略匹配。如果没有指定标识符，默认使用postgresql。

20.12. 证书认证

这种认证方法使用 SSL 客户端证书执行认证。因此，它只适用于 SSL 连接。当使用这种认证方法时，服务器将要求客户端提供一个有效的、可信的证书。不会有密码提示将被发送到客户端。证书的cn（通用名）属性将与被请求的数据库用户名进行比较，并且如果匹配将允许登录。用户名映射可以被用来允许cn与数据库用户名不同。

下列被支持的配置选项用于 SSL 证书认证：

map

允许在系统和数据库用户名之间的映射。详见第 20.2 节

在一条指定证书认证的pg_hba.conf记录中，认证选项clientcert被假定为1，并且它不能被关掉，因为这种方法中一个客户端证书是必需的。cert方法对基本clientcert证书验证测试所增加的东西是检查cn属性是否匹配数据库用户名。

20.13. PAM 认证

这种认证方法操作起来类似password， 只不过它使用 PAM （插入式验证模块）作为认证机制。默认的 PAM 服务名是postgresql。PAM 只被用于验证用户名/口令对并且可以有选择地验证已连接的远程主机名或 IP 地址。因此，在使用 PAM 进行认证之前，用户必须已经存在于数据库中。有关 PAM 的更多信息，请阅读 Linux-PAM 页面[2].

下列被支持的配置选项用于 PAM：

pamservice

PAM服务名称。

pam_use_hostname

判断是否通过PAM_RHOST项把远程 IP 地址或者主机名提供给 PAM 模块。默认情况下会使用 IP 地址。把这个选项设置为 1 可以使用解析过的主机名。主机名解析可能导致登录延迟（大部分的 PAM 配置不使用这些信息，因此只有使用为利用这种信息而特别创建的 PAM 配置时才需要考虑这个设置）。

> **注意**
>
> 如果 PAM 被设置为读取/etc/shadow，认证将会失败，因为 PostgreSQL 服务器是由一个非 root 用户启动 。然而，当 PAM 被配置为使用 LDAP 或其他认证验证方法时这就不是一个问题。

20.14. BSD 认证

这种认证方法操作起来类似于password，不过它使用 BSD 认证来验证口令。BSD 认证只被用来验证用户名/口令对。因此，在 BSD 认证可以被用于认证之前，用户的角色必须已经存在于数据库中。BSD 认证框架当前只在 OpenBSD 上可用。

[2] https://www.kernel.org/pub/linux/libs/pam/

PostgreSQL中的 BSD 认证使用auth-postgresql登录类型，如果login.conf中定义了postgresql登录分类，就会用它来认证。默认情况下这种登录分类不存在，PostgreSQL将使用默认的登录分类。

> **注意**
>
> 要使用 BSD 认证，PostgreSQL 用户账号（也就是运行服务器的操作系统用户）必须首先被加入到auth组中。在 OpenBSD 系统上默认存在auth组。

20.15. 认证问题

认证失败以及相关的问题通常由类似下面的错误消息显示：

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database
 "testdb"
```

这条消息最可能出现的情况是你成功地联系了服务器，但它不愿意和你说话。就像消息本身所建议的，服务器拒绝了连接请求，因为它没有在其pg_hba.conf配置文件里找到匹配项。

```
FATAL:  password authentication failed for user "andym"
```

这样的消息表示你联系了服务器，并且它也愿意和你交谈，但是你必须通过pg_hba.conf文件中指定的认证方法。检查你提供的口令，或者如果错误消息提到了 Kerberos 或 ident 认证类型，检查那些软件。

```
FATAL:  user "andym" does not exist
```

指示的数据库用户没有被找到。

```
FATAL:  database "testdb" does not exist
```

你试图连接的数据库不存在。请注意如果你没有声明数据库名，默认会用数据库用户名作为数据库名，这可能正确也可能不正确。

> **提示**
>
> 服务器日志可能包含比报告给客户端的更多的有关认证失败的信息。如果你为失败的原因而困惑，那么请检查服务器日志。

第 21 章 数据库角色

PostgreSQL使用角色的概念管理数据库访问权限。一个角色可以被看成是一个数据库用户或者是一个数据库用户组，这取决于角色被怎样设置。角色可以拥有数据库对象（例如，表和函数）并且能够把那些对象上的权限赋予给其他角色来控制谁能访问哪些对象。此外，还可以把一个角色中的成员资格授予给另一个角色，这样允许成员角色使用被赋予给另一个角色的权限。

角色的概念把“用户”和“组”的概念都包括在内。在PostgreSQL版本 8.1 之前，用户和组是完全不同的两种实体，但是现在只有角色。任意角色都可以扮演用户、组或者两者。

本章描述如何创建和管理角色。更多角色权限在多个数据库对象上的效果可以在第 5.6 节找到。

21.1. 数据库角色

数据库角色在概念上已经完全与操作系统用户独立开来。事实上可能维护一个对应关系会比较方便，但是这并非必需。数据库角色在一个数据库集簇安装范围内是全局的（而不是独立数据库内）。要创建一个角色，可使用CREATE ROLE SQL 命令：

```
CREATE ROLE name;
```

name遵循 SQL 标识符的规则：或是未经装饰没有特殊字符，或是用双引号包围（实际上，你将总是给该命令要加上额外选项，例如LOGIN。更多细节可见下文）。要移除一个已有的角色，使用相似的DROP ROLE命令：

```
DROP ROLE name;
```

为了方便，createuser和dropuser程序被提供作为这些 SQL 命令的包装器，它们可以从 shell 命令行调用：

```
createuser name
dropuser name
```

要决定现有角色的集合，检查pg_roles系统目录，例如：

```
SELECT rolname FROM pg_roles;
```

psql程序的\du元命令也可以用来列出现有角色。

为了引导数据库系统，一个刚刚被初始化好的系统总是包含一个预定义角色。这个角色总是一个“superuser”，并且默认情况下（除非在运行initdb时修改）它的名字和初始化数据库集簇的操作系统用户相同。习惯上，这个角色将被命名为postgres。为了创建更多角色，你首先必须以初始角色的身份连接。

每一个到数据库服务器的连接都是使用某个特定角色名建立的，并且这个角色决定发起连接的命令的初始访问权限。要使用一个特定数据库连接的角色名由客户端指示，该客户端以一种应用相关的风格发起连接请求。例如，psql程序使用-U命令行选项来指定要以哪个角色连接。很多应用假定该名字默认是当前操作系统用户（包括createuser和psql）。因此在角色和操作系统用户之间维护一个名字对应关系通常是很方便的。

一个给定客户端连接能够用来连接的数据库角色的集合由该客户端的认证设置决定，这些在第 20 章有解释（因此，一个客户端不止限于以匹配其操作系统用户的角色连接，就像一个人的登录名不需要匹配她的真实名字一样）。因为角色身份决定一个已连接客户端可用的权限集合，在设置一个多用户环境时要小心地配置权限。

21.2. 角色属性

一个数据库角色可以有一些属性，它们定义角色的权限并且与客户端认证系统交互。

login privilege

只有具有LOGIN属性的角色才能被用于一个数据库连接的初始角色名称。一个带有LOGIN属性的角色可以被认为和一个“数据库用户”相同。要创建一个带有登录权限的角色，使用两者之一：

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

（CREATE USER和CREATE ROLE等效，除了CREATE USER默认假定有LOGIN，而CREATE ROLE不这样认为）。

superuser status

一个数据库超级用户会绕开所有权限检查，除了登入的权利。这是一个危险的权限并且应该小心使用，最好用一个不是超级用户的角色来完成你的大部分工作。要创建一个新数据库超级用户，使用CREATE ROLE name SUPERUSER。你必须作为一个超级用户来完成这些。

database creation

一个角色必须被显式给予权限才能创建数据库（除了超级用户，因为它们会绕开所有权限检查）。要创建这样一个角色，使用CREATE ROLE name CREATEDB。

role creation

一个角色必须被显式给予权限才能创建更多角色（除了超级用户，因为它们会绕开所有权限检查）。要创建这样一个角色，使用CREATE ROLE name CREATEROLE。一个带有CREATEROLE权限的角色也可以修改和删除其他角色，还可以授予或回收角色中的成员关系。然而，要创建、修改、删除或修改一个超级用户角色的成员关系，需要以超级用户的身份操作。CREATEROLE不足以完成这一切。

initiating replication

一个角色必须被显式给予权限才能发起流复制（除了超级用户，因为它们会绕开所有权限检查）。一个被用于流复制的角色必须也具有LOGIN权限。要创建这样一个角色，使用CREATE ROLE name REPLICATION LOGIN。

password

只有当客户端认证方法要求用户在连接数据库时提供一个口令时，一个口令才有意义。password和md5认证方法使用口令。数据库口令与操作系统命令独立。在角色创建时指定一个口令：CREATE ROLE name PASSWORD 'string'。

在创建后可以用ALTER ROLE修改一个角色属性。CREATE ROLE和ALTER ROLE命令的细节可见参考页。

> **提示**
>
> 一个好习惯是创建一个具有CREATEDB和CREATEROLE权限的角色，而不是创建一个超级用户，并且然后用这个角色来完成对数据库和角色的例行管理。这种方法避免了在非必要时作为超级用户操作任务的风险。

对于第 19 章描述的运行时配置设置，一个角色也可以有角色相关的默认值。例如，如果出于某些原因你希望在每次连接时禁用索引扫描（提示：不是好主意），你可以使用：

```
ALTER ROLE myname SET enable_indexscan TO off;
```

这将保存设置（但是不会立刻设置它）。在这个角色的后续连接中，它就表现得像在会话开始之前执行过SET enable_indexscan TO off。你也可以在会话期间改变该设置，它将只是作为默认值。要移除一个角色相关的默认设置，使用ALTER ROLE rolename RESET varname。注意附加到没有LOGIN权限的角色的角色相关默认值相当无用，因为它们从不会被调用。

21.3. 角色成员关系

把用户分组在一起来便于管理权限常常很方便：那样，权限可以被授予一整个组或从一整个组回收。在PostgreSQL中通过创建一个表示组的角色来实现，并且然后将在该组角色中的成员关系授予给单独的用户角色。

要建立一个组角色，首先创建该角色：

```
CREATE ROLE name;
```

通常被用作一个组的角色不需要有LOGIN属性，不过如果你希望你也可以设置它。

一旦组角色存在，你可以使用GRANT和REVOKE命令增加和移除成员：

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

你也可以为其他组角色授予成员关系（因为组角色和非组角色之间其实没有任何区别）。数据库将不会让你设置环状的成员关系。另外，不允许把一个角色中的成员关系授予给PUBLIC。

组角色的成员可以以两种方式使用角色的权限。第一，一个组的每一个成员可以显式地做SET ROLE来临时“成为”组角色。在这种状态中，数据库会话可以访问组角色而不是原始登录角色的权限，并且任何被创建的数据库对象被认为属于组角色而不是登录角色。第二，有INHERIT属性的成员角色自动地具有它们所属角色的权限，包括任何组角色继承得到的权限。作为一个例子，假设我们已经有：

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

在作为角色joe连接后，一个数据库会话将立即拥有直接授予给joe的权限，外加任何授予给admin的权限，因为joe“继承了” admin的权限。然而，授予给wheel的权限不可用，因为即使joe是wheel的一个间接成员，但是该成员关系是通过带NOINHERIT属性的admin得到的。在：

```
SET ROLE admin;
```

之后，该会话将只拥有授予给admin的权限，但是没有授予给joe的权限。在执行：

```
SET ROLE wheel;
```

之后，该会话将只拥有授予给wheel的权限，但是没有授予给joe或admin的权限。初始的权限状态可以使用下面命令之一恢复：

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> **注意**
>
> SET ROLE命令总是允许选择原始登录角色的直接或间接组角色。因此，在上面的例子中，在成为wheel之前不必先成为admin。

> **注意**
>
> 在 SQL 标准中，用户和角色之间的区别很清楚，并且用户不会自动继承权限而角色会继承。这种行为在PostgreSQL中也可以实现：为要用作 SQL 角色的角色给予INHERIT属性，而为要用作 SQL 用户的角色给予NOINHERIT属性。不过，为了向后兼容 8.1 以前的发布（在其中用户总是拥有它们所在组的权限），PostgreSQL默认给所有的角色INHERIT属性。

角色属性LOGIN、SUPERUSER、CREATEDB和CREATEROLE可以被认为是一种特殊权限，但是它们从来不会像数据库对象上的普通权限那样被继承。要使用这些属性，你必须实际SET ROLE到一个有这些属性之一的特定角色。继续上述例子，我们可以选择授予CREATEDB和CREATEROLE给admin角色。然后一个以joe角色连接的会话将不会立即有这些权限，只有在执行了SET ROLE admin之后才会拥有。

要销毁一个组角色，使用DROP ROLE：

```
DROP ROLE name;
```

任何在该组角色中的成员关系会被自动撤销（但是成员角色不会受到影响）。

21.4. 删除角色

由于角色可以拥有数据库对象并且能持有访问其他对象的特权，删除一个角色 常常并非一次DROP ROLE就能解决。任何被该用户所拥有 的对象必须首先被删除或者转移给其他拥有者，并且任何已被授予给该角色的 权限必须被收回。

对象的拥有关系可以使用ALTER命令一次转移出去，例如：

```
ALTER TABLE bobs_table OWNER TO alice;
```

此外，REASSIGN OWNED命令可以被用来把要被删除的 角色所拥有的所有对象的拥有关系转移给另一个角色。由于 REASSIGN OWNED不能访问其他数据库中的对象，有必要 在每一个包含该角色所拥有对象的数据库中运行该命令（注意第一个这样的 REASSIGN OWNED将更改任何在数据库间共享的该角色拥 有的对象的拥有关系，即数据库或者表空间）。

一旦任何有价值的对象已经被转移给新的拥有者，任何由被删除角色拥有的剩余对象 就可以用DROP OWNED命令删除。再次，由于这个命令不能 访问其他数据库中的对象， 有必要在每一个包含该角色所拥有对象的数据库中运行 该命令。还有，DROP OWNED将不会删除整个

数据库或者表空间， 因此如果该角色拥有任何还没有被转移给新拥有者的数据库或者表空间，有必要手工 删除它们。

DROP OWNED也会注意移除为不属于目标角色的对象授予给目标 角色的任何特权。因为REASSIGN OWNED不会触碰这类对象，通 常有必要运行REASSIGN OWNED和 DROP OWNED（按照这个顺序！）以完全地移除要被删除对象的 从属物。

总之，移除曾经拥有过对象的角色的方法是：

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- 在集簇中的每一个数据库中重复上述命令
DROP ROLE doomed_role;
```

如果不是所有的拥有对象都被转移给了同一个后继拥有者，最好手工处理异常 然后执行上述步骤直到结束。

如果在依赖对象还存在时尝试了DROP ROLE，它将发出 消息标识哪些对象需要被重新授予或者删除。

21.5. 默认角色

PostgreSQL提供了一组默认角色，它们提供对特定的、通常需要的、需要特权的功能和信息的访问。管理员可以把这些角色GRANT给其环境中的用户或者其他角色，让这些用户能够访问指定的功能和信息。

表 21.1中描述了默认的角色。注意由于额外功能的增加，每一种默认角色相关的权限可能会在未来被改变。管理员应该关注发行注记中提到的这方面的变化。

表 21.1. 默认角色

角色	允许的访问
pg_read_all_settings	读取所有配置变量，甚至是那些通常只对超级用户可见的变量。
pg_read_all_stats	读取所有的pg_stat_*视图并且使用与扩展相关的各种统计信息，甚至是那些通常只对超级用户可见的信息。
pg_stat_scan_tables	执行可能会在表上取得ACCESS SHARE锁的监控函数（可能会持锁很长时间）。
pg_signal_backend	向其他后端发送信号（例如：取消查询、中止）。
pg_read_server_files	允许使用COPY以及其他文件访问函数从服务器上该数据库可访问的任意位置读取文件。
pg_write_server_files	允许使用COPY以及其他文件访问函数在服务器上该数据库可访问的任意位置中写入文件。
pg_execute_server_program	允许用运行该数据库的用户执行数据库服务器上的程序来配合COPY和其他允许执行服务器端程序的函数。
pg_monitor	读取/执行各种监控视图和函数。这个角色是pg_read_all_settings、pg_read_all_stats以及pg_stat_scan_tables的成员。

pg_read_server_files、pg_write_server_files以及pg_execute_server_program角色的目的是允许管理员有一些可信但不是超级用户的角色来访问文件以及以运行数据库的用户在数

据库服务器上运行程序。由于这些角色能够防伪服务器文件系统上的任何文件，因此在直接访问文件时它们会绕过任何数据库级别的权限检查并且它们可以被用来得到超级用户级别的访问，因此在把这些角色授予给用户时应当非常小心。

pg_monitor、pg_read_all_settings、pg_read_all_stats和pg_stat_scan_tables角色的目的是允许管理员能为监控数据库服务器的目的很容易地配置角色。它们授予一组常用的特权，这些特权允许角色读取各种有用的配置设置、统计信息以及通常仅限于超级用户的其他系统信息。

在授予这些角色时应当非常小心，以确保它们只被用在需要的地方，并且要理解这些角色会授予对特权信息的访问。

管理员可以用GRANT命令把对这些角色的访问授予给用户：

```
GRANT pg_signal_backend TO admin_user;
```

21.6. 函数和触发器安全性

函数、触发器以及行级安全性策略允许用户在后端服务器中插入代码，其他用户不会注意到这些代码的执行。因此，这些机制允许用户相对容易地为其他人设置“特洛伊木马”。最强的保护是严格控制哪些人能定义对象。如果做不到，则编写查询时应该只引用具有可信任拥有者的对象。可以从search_path中去除public方案以及任何其他允许不可信用户创建对象的方案。

在后端服务器进程中运行的函数带有数据库服务器守护进程的操作系统权限。如果用于函数的编程语言允许非检查的内存访问，它就可能改变服务器的内部数据结构。因此，在很多其他事情中，这些函数可能绕开任何系统访问控制。允许这种访问的函数语言被认为是“不可信的”，并且PostgreSQL只允许超级用户创建用这些语言编写的函数。

第 22 章 管理数据库

每个正在运行的PostgreSQL服务器实例都管理着一个或多个数据库。因此，在组织SQL对象（“数据库对象”）的层次中，数据库位于最顶层。本章描述数据库的属性，以及如何创建、管理、删除它们。

22.1. 概述

一个数据库是一些SQL对象（“数据库对象”）的命名集合。通常每个数据库对象（表、函数等） 属于并且只属于一个数据库（不过有几个系统表如pg_database属于整个集簇并且对集簇中的每个数据库都是可访问的）。更准确地说，一个数据库是一个模式的集合， 而模式包含表、函数等等。因此完整的层次是这样的：服务器、数据库、模式、表（或者某些其他对象类型，如函数）。

当连接到数据库服务器时，客户端必须在它的连接请求中指定它要连接的数据库名。每次连接不能访问超过一个数据库。不过，一个应用能够在同一个或者其他数据库上打开的连接数并没有受到限制。数据库是物理上相互隔离的，并且访问控制是在连接层面进行管理的。如果一个PostgreSQL服务器实例用于承载那些应该分隔并且相互之间并不知晓的用户 和项目，那么我们建议把它们放在不同的数据库里。如果项目或者用户是相互关联的， 并且可以相互使用对方的资源，那么应该把它们放在同一个数据库里，但可能在不同的模式 中。模式只是一个纯粹的逻辑结构并且谁能访问某个模式由权限系统管理。有关管理模式的更多信息在第 5.8 节。

数据库是使用CREATE DATABASE（见第 22.2 节，并且用DROP DATABASE命令删除（见第 22.5 节。要确定现有数据库的集合，可以检查系统目录pg_database，例如

```
SELECT datname FROM pg_database;
```

psql程序的\l元命令和-l命令行选项也可以用来列出已有的数据库。

> **注意**
>
> SQL标准把数据库称作“目录”，不过实际上没有区别。

22.2. 创建一个数据库

为了创建一个数据库，PostgreSQL服务器必须启动并运行（见第 18.3 节。

数据库用 SQL 命令CREATE DATABASE创建：

```
CREATE DATABASE name;
```

其中name遵循SQL标识符的一般规则。当前角色自动成为该新数据库的拥有者。以后删除这个数据库也是该拥有者的特权（同时还会删除其中的所有对象，即使那些对象有不同的拥有者）。

创建数据库是一个受限的操作。如何授权请见第 21.2 节

因为你需要连接到数据库服务器来执行CREATE DATABASE命令， 那么还有一个问题是任意给定站点的第一个数据库是怎样创建的？第一个数据库总是由initdb命令在初始化数据存储区域时创建的（见第 18.2 节。这个数据库被称为postgres。因此要创建第一个“普通”数据库时，你可以连接到postgres。

在数据库集簇初始化期间也会创建第二个数据库template1,。当在集簇中创建一个新数据库时，实际上就是克隆了template1。这就意味着你对template1所做的任　何修改都会体现在所有随后创建的数据库中。因此应避免在template1中创建对象，除非你想把它们传播到每一个新创建的数据库中。详见第　22.3　节

为了方便，你还可以用一个程序来创建新数据库： createdb。

```
createdb dbname
```

createdb没什么神奇的。它连接到postgres数据库并且发出CREATE　DATABASE命令，和前面介绍的完全一样。createdb参考页包含了调用细节。注意不带任何参数的createdb将创建一个使用当前用户名的数据库。

> **注意**
>
> 第　20　章包含有关如何限制谁能连接到一个给定数据库的信息。

有时候你想为其他人创建一个数据库，并且使其成为新数据库的拥有者，　这样他们就可以自己配置和管理这个数据库。要实现这个目标，使用下列命令之一： 用于 SQL 环境的

```
CREATE DATABASE dbname OWNER rolename;
```

或者用于 shell 的

```
createdb -O rolename dbname
```

只有超级用户才被允许为其他人（即为一个你不是其成员的角色）创建一个数据库。

22.3.　模板数据库

CREATE　　DATABASE实际上通过拷贝一个已有数据库进行工作。默认情况下，它拷贝名为template1的标准系统数据库。所以该数据库是创建新数据库的“模板”。　　如果你为template1数据库增加对象，这些对象将被拷贝到后续创建的用户数据库中。　这种行为允许对数据库中标准对象集合的站点本地修改。例如，如果你把过程语言PL/Perl安装到template1中，那么你在创建用户数据库后不需要额外的操作就可以使用该语言。

系统里还有名为template0的第二个标准系统数据库。这个数据库包含和template1初始内容一样的数据，也就是说，只包含你的PostgreSQL版本预定义的标准对象。在数据库集簇被初始化之后，不应该对template0做任何修改。通过指示CREATE　DATABASE使用template0取代template1进行拷贝，你可以创建一个“纯净的”用户数据库，它不会包含任何template1中的站点本地附加物。　这一点在恢复一个pg_dump转储时非常方便：转储脚本应该在一个纯净的数据库　中恢复以确保我们重建被转储数据库的正确内容，而不和任何现在可能已经被加入到template1中的附加物相冲突。

另一个从template0而不是template1复制的常见原因是，　可以在复制template0时指定新的编码和区域设置，而一个template1的副本必须使用和它相同的设置。这是因为的template1可能包含编码相关或区域相关的数据，而template0中没有。

要通过拷贝template0来创建一个数据库，使用：SQL 环境中的

```
CREATE DATABASE dbname TEMPLATE template0;
```

或者 shell 中的

```
createdb -T template0 dbname
```

可以创建额外的模板数据库，并且实际上可以通过将集簇中任意数据库指定为CREATE DATABASE的模板来从该数据库拷贝。不过，我们必需明白，这个功能并不是设计作为一般性的“COPY DATABASE”功能。主要的限制是当源数据库被拷贝时，不能有其他会话连接到它。如果在CREATE DATABASE开始时存在任何其它连接，那么该命令将会失败。在拷贝操作期间，到源数据库的新连接将被阻止。

对于每一个数据库在pg_database中存在两个有用的标志：datistemplate和datallowconn列。datistemplate可以被设置来指示该数据库是不是要作为CREATE DATABASE的模板。如果设置了这个标志，那么该数据库可以被任何有 CREATEDB权限的用户克隆；如果没有被设置，那么只有超级用户和该数据库的拥有者可以克隆它。如果datallowconn为假，那么将不允许与该数据库建立任何新的连接（但已有的会话不会因为把该标志设置为假而被中止）。template0通常被标记为datallowconn = false来阻止对它的修改。template0和template1通常总是被标记为datistemplate = true。

> **注意**
>
> 除了template1是CREATE DATABASE的默认源数据库名之外，template1和template0没有任何特殊的状态。例如，我们可以删除template1然后从template0重新创建它而不会有任何不良效果。如果我们不小心在template1中增加了一堆垃圾，那么我们就会建议做这样的操作（要删除template1，它必须有pg_database.datistemplate = false）。
>
> 当数据库集簇被初始化时，也会创建postgres数据库。这个数据库用于做为用户和应用连接的默认数据库。它只是 template1的一个拷贝，需要时可以删除并重建。

22.4. 数据库配置

回顾一下第 19 章PostgreSQL服务器提供了大量的运行时配置变量。你可以为其中的许多设置数据库相关的默认值。

例如，如果由于某种原因，你想禁用指定数据库上的GEQO优化器，正常情况下你不得不对所有数据库禁用它，或者确保每个连接的客户端小心地发出了SET geqo TO off。要令这个设置在一个特定数据库中成为默认值，你可以执行下面的命令：

```
ALTER DATABASE mydb SET geqo TO off;
```

这样将保存该设置（但不是立即设置它）。在后续建立的到该数据库的连接中它将表现得像在会话开始后马上调用SET geqo TO off;。注意用户仍然可以在该会话中更改这个设置，它只是默认值。要撤消这样的设置，使用ALTER DATABASE dbname RESET varname。

22.5. 销毁一个数据库

数据库用DROP DATABASE命令删除：

```
DROP DATABASE name;
```

只有数据库的拥有者或者超级用户才可以删除数据库。删除数据库会移除其中包括的所有对象。数据库的删除不能被撤销。

你不能在与目标数据库连接时执行DROP DATABASE命令。不过，你可以连接到任何其它数据库，包括 template1数据库。template1也是你删除一个给定集簇中最后一个用户数据库的唯一选项。

为了方便，有一个在 shell 程序可以删除数据库，dropdb：

```
dropdb dbname
```

（和createdb不同，删除当前用户名的数据库不是默认动作）。

22.6. 表空间

PostgreSQL中的表空间允许数据库管理员在文件系统中定义用来存放表示数据库对象的文件的位置。一旦被创建，表空间就可以在创建数据库对象时通过名称引用。

通过使用表空间，管理员可以控制一个PostgreSQL安装的磁盘布局。 这么做至少有两个用处。首先，如果初始化集簇所在的分区或者卷用光了空间，而又不能在逻辑上扩展或者做别的什么操作，那么表空间可以被创建在一个不同的分区上，直到系统可以被重新配置。

其次，表空间允许管理员根据数据库对象的使用模式来优化性能。例如，一个很频繁使用的索引可以被放在非常快并且非常可靠的磁盘上，如一种非常贵的固态设备。同时，一个很少使用的或者对性能要求不高的存储归档数据的表可以存储在一个便宜但比较慢的磁盘系统上。

> **警告**
>
> 即便是位于主要的 PostgreSQL 数据目录之外，表空间也是数据库集簇的一部分 并且不能被视作数据文件的一个自治集合。 它们依赖于包含在主数据目录中的元数据，并且因此不能被附加到一个 不同的数据库集簇或者单独备份。类似地，如果丢失一个表空间（文件删除、磁盘失效等）， 数据库集簇可能会变成不可读或者无法启动。把一个表空间放在一个临时文件系统 （如一个内存虚拟盘）上会带来整个集簇的可靠性风险。

要定义一个表空间，使用CREATE TABLESPACE命令，例如：

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

这个位置必须是一个已有的空目录，并且属于PostgreSQL操作系统用户。 所有后续在该表空间中创建的对象都将被存放在这个目录下的文件中。该位置不能放在可移动 或者瞬时存储上，因为如果表空间丢失会导致集簇无法工作。

> **注意**
>
> 通常在每个逻辑文件系统上创建多于一个表空间没有什么意义，因为你无法控制在一个逻辑文件系统中特定文件的位置。不过，PostgreSQL不强制任何这样的限制，并且事实上它不会注意你的系统上的文件系统边界。它只是在你告诉它要使用的目录中存储文件。

表空间的创建本身必须作为一个数据库超级用户完成，但在创建完之后之后你可以允许普通数据库用户来使用它。要这样做，给数据库普通用户授予表空间上的CREATE权限。

表、索引和整个数据库都可以被分配到特定的表空间。想这么做，在给定表空间上有CREATE权限的用户必须把表空间的名字以一个参数的形式传递给相关的命令。例如，下面的命令在表空间space1中创建一个表：

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

另外，还可以使用default_tablespace参数：

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

当default_tablespace被设置为非空字符串，那么它就为没有显式TABLESPACE子句的CREATE TABLE和CREATE INDEX命令提供一个隐式TABLESPACE子句。

还有一个temp_tablespaces参数，它决定临时表和索引的位置，以及用于大数据集排序等目的的临时文件的位置。 这可以是一个表空间名的列表，而不是只有一个。因此，与临时对象有关的负载可以散布在多个表空间上。每次要创建一个临时对象时，将从列表中随机取一个成员来存放它。

与一个数据库相关联的表空间用来存储该数据库的系统目录。此外，如果没有给出TABLESPACE子句并且没有在default_tablespace或temp_tablespaces（如适用）中指定其他选择，它还是在该数据库中创建的表、索引和临时文件的默认表空间。如果一个数据库被创建时没有指定表空间，它会使用其模板数据库相同的表空间。

当初始化数据库集簇时，会自动创建两个表空间。pg_global表空间被用于共享系统目录。pg_default表空间是template1和template0数据库的默认表空间（并且，因此也将是所有其他数据库的默认表空间，除非被一个CREATE DATABASE中的TABLESPACE子句覆盖）。

表空间一旦被创建，就可以被任何数据库使用，前提是请求的用户具有足够的权限。这也意味着，一个表空间只有在所有使用它的数据库中所有对象都被删除掉之后才可以被删掉。

要删除一个空的表空间，使用DROP TABLESPACE命令。

要确定现有表空间的集合，可检查pg_tablespace 系统目录，例如

```
SELECT spcname FROM pg_tablespace;
```

psql程序的\db元命令也可以用来列出现有的表空间。

PostgreSQL使用符号连接来简化表空间的实现。这就意味着表空间只能在支持符号连接的系统上使用。

$PGDATA/pg_tblspc目录包含指向集簇中定义的每个非内建表空间的符号连接。 尽管我们不推荐，但还是可以通过手工重定义这些连接来调整表空间布局。在服务器运行时，绝不要这样做。注意在 PostgreSQL 9.1 及更早的版本中，你将还需要用新位置更新pg_tablespace目录（如果你不更新，pg_dump将继续输出旧的表空间位置）。

第 23 章 本地化

本章从管理员的角度描述可用的本地化特性。PostgreSQL支持两种本地化方法：

- 利用操作系统的区域（locale）特性，提供对区域相关的排序顺序、数字格式、 翻译过的信息和其它方面。这种方法在第 23.1 和第 23.2 节。
- 提供一些不同的字符集来支持存储所有种类语言的文本，并提供在客户端和服务器之间的字符集转换。这种方法在第 23.3 节。

23.1. 区域支持

区域支持指的是应用遵守文化偏好的问题，包括字母表、排序、数字格式等。PostgreSQL使用服务器操作系统提供的标准 ISO C 和POSIX的区域机制。更多的信息请参考你的系统的文档。

23.1.1. 概述

区域支持是在使用initdb创建一个数据库集簇时自动被初始化的。默认情况下，initdb将会按照它的执行环境的区域设置初始化数据库集簇； 因此如果你的系统已经设置为你的数据库集簇想要使用的区域， 那么你就没有什么可干的。如果你想使用其它的区域（或者你还不知道你的系统设置的区域是什么），那么你可以用--locale选项准确地告诉initdb你要用哪一个区域。 比如：

```
initdb --locale=sv_SE
```

这个Unix系统上的例子把区域设置为瑞典（SE）瑞典语（sv）。 其他的可能性包括 en_US（美国英语）和fr_CA（加拿大法语）。如果有多于一种字符集可以用于区域，那么声明可以采用如下的形式：language_territory.codeset。例如fr_BE.UTF-8表示在比利时（BE）讲的法语（fr），使用一个UTF-8字符集编码。

在你的系统上有哪些区域可用取决于操作系统提供商提供了什么以及安装了什么。在大部分Unix系统上，命令locale -a将会提供一个所有可用区域的列表。Windows使用一些更繁琐的区域名，例如German_Germany或者Swedish_Sweden.1252，但是其原则是相同的。

有时候，把几种区域规则混合起来也很有用，比如，使用英语排序规则而用西班牙语消息。为了支持这些，我们有一套区域子类用于控制本地化规则的某些方面：

LC_COLLATE	字符串排序顺序
LC_CTYPE	字符分类（什么是一个字符？它的大写形式是否等效？）
LC_MESSAGES	消息使用的语言Language of messages
LC_MONETARY	货币数量使用的格式
LC_NUMERIC	数字的格式
LC_TIME	日期和时间的格式

这些类名转换成initdb的选项名来覆盖某个特定分类的区域选择。比如，要把区域设置为加拿大法语，但使用 U.S. 规则格式化货币，可以使用initdb --locale=fr_CA --lc-monetary=en_US。

如果你想让系统表现得象没有区域支持，那么使用特殊的区域名C或者等效的POSIX。

一些区域分类的值必需在数据库被创建时的就被固定。你可以为不同的数据库使用不同的设置，但是一旦一个数据库被创建，你就不能在数据库上修改这些区域分类的值。LC_COLLATE和LC_CTYPE就是这样的分类。它们影响索引的排序顺序，因此它们必需保持固定， 否则在文本列上的索引将会崩溃（但是你可以使用排序规则放松这种限制，讨论

见第 23.2 节。这些分类的默认值在initdb运行时被确定，并且这些值在新数据库被创建时使用，除非在CREATE DATABASE命令中特别指定。

其它区域分类可以在任何时候被更改，更改的方式是设置与区域分类同名的服务器配置参数（详见第 19.11.2 节。被initdb选中的值实际上只是被写入到配置文件postgresql.conf中作为服务器启动时的默认值。如果你将这些赋值从postgresql.conf中除去，那么服务器将会从其执行环境中继承该设置。

请注意服务器的区域行为是由它看到的环境变量决定的，而不是由任何客户端的环境变量影响的。 因此，我们要在启动服务器之前认真地设置好这些变量。这样带来的一种后果是如果客户端和服务器设置成不同的区域， 那么消息可能以不同的语言呈现，实际情况取决于它们的起源地。

> **注意**
>
> 在我们谈到从执行环境继承区域的时候，我们的意思是在大多数操作系统上的下列动作： 对于一个给定的区域分类，比如排序规则，按照下面的顺序评估这些环境变量， 直到找到一个被设置了的：LC_ALL、 LC_COLLATE（或者对应于相应分类的变量）、LANG。如果这些环境变量一个都没有被设置，那么将区域缺省设置为C。
>
> 一些消息本地化库也查看环境变量LANGUAGE，它覆盖所有其它用于设置消息语言的区域设置。如果有疑问， 请参考你的操作系统的文档，特别是有关gettext的文档。

要允许消息被翻译成用户喜欢的语言，编译时必需打开NLS（configure --enable-nls）。所有其他区域支持都会被自动编译。

23.1.2. 行为

区域设置特别影响下面的 SQL 特性：

- 在文本数据上使用ORDER BY或标准比较操作符的查询中的排序顺序
- 函数upper、lower和initcap
- 模式匹配操作符（LIKE、SIMILAR TO和POSIX风格的正则表达式）；区域影响大小写不敏感匹配和通过字符类正则表达式的字符分类
- to_char函数家族
- 为LIKE子句使用索引的能力

PostgreSQL中使用非C或非POSIX区域的缺点是性能影响。它降低了字符处理的速度并且阻止了在LIKE中对普通索引的使用。因此，只能在真正需要的时候才使用它。

作为允许PostgreSQL在非 C 区域下为LIKE子句使用索引， 有好几种自定义操作符类可用。这些操作符类允许创建一个执行严格按字符比较的索引。详见第 11.10 节另一种方法是创建使用C排序规则的索引，如第 23.2 节讨论的。

23.1.3. 问题

如果根据上面解释区域支持仍然不能运转，检查一下操作系统的区域支持是否被正确配置。要检查系统中安装了哪些区域，你可以使用命令locale -a（如果你的操作系统提供了该命令）。

请检查PostgreSQL确实正在使用你认为它该用的区域设置。LC_COLLATE和LC_CTYPE设置都是在数据库创建时决定的，并且在除了创建数据库之外的操作中都不能被更改。其它的区域设置包括LC_MESSAGES和LC_MONETARY都是由服务器启动的环境决定的， 但是可以在运行时修改。你可以用SHOW命令检查活跃的区域设置。

源代码目录的src/test/locale中包含PostgreSQL的区域支持的测试套件。

那些通过分析错误消息来处理服务器端错误的客户端应用很明显会有问题，因为服务器来的消息可能会是以不同语言表示的。 我们建议这类应用的开发人员改用错误代码机制。

维护消息翻译目录需要许多志愿者的坚持不懈的努力， 他们希望PostgreSQL以他们的语言说话。 如果以你的语言表示的消息目前还不可用或者没有完全翻译完成，那么我们很感谢你的协助。如果你想帮忙，那么请参考第 55 章或者向开发者邮递列表发邮件。

23.2. 排序规则支持

排序规则特性允许指定每一列甚至每一个操作的数据的排序顺序和字符分类行为。这放松了数据库的LC_COLLATE和LC_CTYPE设置自创建以后就不能更改这一限制。

23.2.1. 概念

在概念上，一种可排序数据类型的每一种表达式都有一个排序规则（内建的可排序数据类型是text、varchar和char。用户定义的基础类型也可以被标记为可排序的，并且在一种可排序数据类型上的域也是可排序的）。如果该表达式是一个列引用，该表达式的排序规则就是列所定义的排序规则。如果该表达式是一个常量，排序规则就是该常量数据类型的默认排序规则。更复杂表达式的排序规则根据其输入的排序规则得来，如下所述：

一个表达式的排序规则可以是“默认”排序规则，它表示数据库的区域设置。一个表达式的排序规则也可能是不确定的。在这种情况下，排序操作和其他需要知道排序规则的操作会失败。

当数据库系统必须要执行一次排序或者字符分类时，它使用输入表达式的排序规则。这会在使用例如ORDER BY子句以及函数或操作符调用（如<）时发生。应用于ORDER BY子句的排序规则就是排序键的排序规则。应用于函数或操作符调用的排序规则从它们的参数得来，具体如下文所述。除比较操作符之外，在大小写字母之间转换的函数会考虑排序规则，例如lower、upper和initcap。模式匹配操作符和to_char及相关函数也会考虑排序规则。

对于一个函数或操作符调用，其排序规则通过检查在执行指定操作时参数的排序规则来获得。如果该函数或操作符调用的结果是一种可排序的数据类型，万一有外围表达式要求函数或操作符表达式的排序规则，在解析时结果的排序规则也会被用作函数或操作符表达式的排序规则。

一个表达式的排序规则派生可以是显式或隐式。该区别会影响多个不同的排序规则出现在同一个表达式中时如何组合它们。当使用一个COLLATE子句时，将发生显式排序规则派生。所有其他排序规则派生都是隐式的。当多个排序规则需要被组合时（例如在一个函数调用中），将使用下面的规则：

1. 如果任何一个输入表达式具有一个显式排序规则派生，则在输入表达式之间的所有显式派生的排序规则必须相同，否则将产生一个错误。如果任何一个显式派生的排序规则存在，它就是排序规则组合的结果。

2. 否则，所有输入表达式必须具有相同的隐式排序规则派生或默认排序规则。如果任何一个非默认排序规则存在，它就是排序规则组合的结果。否则，结果是默认排序规则。

3. 如果在输入表达式之间存在冲突的非默认隐式排序规则，则组合被认为是具有不确定排序规则。这并非一种错误情况，除非被调用的特定函数要求提供排序规则的知识。如果它确实这样做，运行时将发生一个错误。

例如，考虑这个表定义：

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

然后在

```
SELECT a < 'foo' FROM test1;
```

中，<比较被根据de_DE规则执行，因为表达式组合了一个隐式派生的排序规则和默认排序规则。但是在

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

中，比较被使用fr_FR规则执行，因为显式排序规则派生重载了隐式排序规则。更进一步，给定

```
SELECT a < b FROM test1;
```

解析器不能确定要应用哪个排序规则，因为a列和b列具有冲突的隐式排序规则。由于<操作符不需要知道到底使用哪一个排序规则，这将会导致一个错误。该错误可以通过在一个输入表达式上附加一个显式排序规则说明符来解决，因此：

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

或者等效的

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

在另一方面，结构相似的情况

```
SELECT a || b FROM test1;
```

不会导致一个错误，因为||操作符不关心排序规则：不管排序规则怎样它的结果都相同。

如果一个函数或操作符发送一个具有可排序数据类型的结果，分配给该函数或操作符的组合输入表达式的排序规则也被考虑应用在函数或操作符的结果。因此，在

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

中排序将根据de_DE规则完成。但这个查询：

```
SELECT * FROM test1 ORDER BY a || b;
```

会导致一个错误，因为即使||操作符不需要知道排序规则，但ORDER BY子句需要。按照以前，冲突可以通过使用一个显式排序规则说明符来解决：

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

23.2.2. 管理排序规则

排序规则是SQL模式对象，它将SQL名称映射到操作系统中安装的库提供的语言环境。 排序规则定义中有一个提供程序， 它指定哪个库提供语言环境数据。一个标准的提供者名称是libc， 它使用操作系统C库提供的语言环境。这些是操作系统提供的大多数工具使用的语言环境。 另一个提供者是icu，它使用外部ICU库。 只有在构建PostgreSQL时配置了对ICU的支持，才能使用ICU区域设置。

libc提供的一个排序规则对象映射到LC_COLLATE 和LC_CTYPE设置的组合， 如setlocale()系统库调用所接受的。 （正如其名字所说的，一个排序规则的主要目的是设置LC_COLLATE，它控制排序顺序。但是在实际中LC_CTYPE设置与LC_COLLATE 不同是很少有必要的，因此通

过一个概念来收集这些信息比为了设置每一个表达式的 LC_CTYPE而创建另一种架构要更加方便）。此外， 一个libc排序规则是和一个字符集编码（见第 23.3 节 绑定在一起的。相同的排序规则名字可能存在于不同的编码中。

由icu提供的排序规则对象映射到由ICU库提供的指定整理器。 ICU不支持单独的“collate”和“ctype”设置， 所以它们总是相同的。此外，ICU排序规则与编码无关，因此在数据库中总是只有一个给定名称的ICU排序规则。

23.2.2.1. 标准的排序规则

在所有的平台上，名为default、C和POSIX的排序规则都可用。附加的排序规则是否可用取决于操作系统的支持。default排序规则选择在数据库创建时指定的LC_COLLATE和LC_CTYPE值。C和POSIX排序规则都指定了“传统的C”行为，在其中只有ASCII字母“A”到“Z”被视为字母，并且排序严格地按照字符编码的字节值完成。

此外，SQL标准排序规则名称ucs_basic可用于编码UTF8。 它相当于C，并按Unicode代码点排序。

23.2.2.2. 预定义的排序规则

如果操作系统支持在一个程序中使用多个区域（newlocale和相关函数）， 或者配置了ICU支持，那么在一个数据集簇被初始化时，initdb 将以它在操作系统中能找到的所有区域为基础在系统目录pg_collation 中填充排序规则。

要检查当前可用的语言环境，请在psql中使用查询 SELECT * FROM pg_collation或命令\dOS+。

23.2.2.2.1. libc 排序规则

例如，操作系统可能会提供一个名为de_DE.utf8的区域。initdb则会创建一个用于编码UTF8的名为de_DE.utf8的排序规则，在其中LC_COLLATE和LC_CTYPE都被设置为de_DE.utf8。它也会创建一个具有去掉名称的.utf8标签的排序规则。这样你也可以使用名字de_DE来使用该排序规则，这写起来更简单并且使得名字更加独立于编码。不过要注意，最初的排序规则名称的集合是平台依赖的。

由libc提供的默认排序规则直接映射到操作系统中安装的语言环境， 可以使用命令locale -a列出。如果所需的libc 排序规则与LC_COLLATE和LC_CTYPE的值不同， 或者在数据库系统初始化之后， 操作系统中安装了新的语言环境，可以使用CREATE COLLATION 命令创建新的排序规则。新的操作系统语言环境也可以使用 pg_import_system_collations() 函数集中导入。

在任何特定的数据库中，只有使用数据库编码的排序规则是令人感兴趣的。其他pg_collation中的项会被忽略。因此，一个如de_DE的被剥离的排序规则名在一个给定数据库中可以被认为是唯一的，即使它在全局上并不唯一。我们推荐使用被剥离的排序规则名，因为在你决定要更改到另一个数据库编码时需要做的事情更少。但是要注意default、C和POSIX排序规则在使用时可以不考虑数据库编码。

PostgreSQL在碰到具有相同属性的不同排序规则对象时会认为它们是不兼容的。因此对于例子：

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

将会得到一个错误，即使C和POSIX排序规则具有相同的行为。因此，我们不推荐混合使用被剥离的和非被剥离的排序规则名。

23.2.2.2.2. ICU 排序规则

对于ICU，枚举所有可能的语言环境名称并不明智。 ICU为语言环境使用特定的命名系统，但命名语言环境的方法多于实际上不同的语言环境。 initdb使用ICU API提取一组不同的语言环境以填充初始排序规则集合。 由ICU提供的排序规则是在SQL环境中创建的，名称采用

BCP 47语言标记格式， 并附有一个“专用”扩展名-x-icu， 以将它们与libc语言环境区分开来。

以下是可能创建的一些排序规则的示例：

de-x-icu

德语排序规则，默认变体

de-AT-x-icu

奥地利的德语排序规则，默认变体

（也就是说de-DE-x-icu 或de-CH-x-icu，但是这种写法，相当于 de-x-icu。）

und-x-icu (for “undefined”)

ICU “root” 排序规则。 使用它获取合理的语言无关的排序顺序

一些（不常用的）编码不受ICU支持。当数据库编码是其中之一时， 忽略pg_collation中的ICU排序规则项。 试图使用其中一个将会抛出一个类似“collation "de-x-icu" for encoding "WIN874" does not exist”的错误。

23.2.2.3. 创建新的排序规则对象

如果标准和预定义的排序规则不够用，用户可以使用SQL命令 CREATE COLLATION创建自己的排序规则对象。

与所有预定于的对象一样，标准和预定义的排序规则在模式 pg_catalog中。用户定义的排序规则应该在用户模式中创建。 这也确保它们由pg_dump保存。

23.2.2.3.1. libc 排序规则

可以像这样创建新的libc排序规则：

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

该命令中locale子句可接受的确切值取决于操作系统。 在类Unix系统上，命令locale -a将显示一个列表。

由于预定义的libc排序规则已经包含了数据库实例初始化时在操作系统中定义的所有排序规则， 因此通常不需要手动创建新排序规则。如果需要不同的命名系统（在这种情况下， 另请参阅第 23.2.2.3.3 节， 或者操作系统已经升级以提供新的区域设置定义（在这种情况下， 另请参阅pg_import_system_collations()）， 可能需要手动创建。

23.2.2.3.2. ICU 排序规则

ICU允许自定义超出由initdb 预加载的基本语言+国家/地区集的排序规则。鼓励用户定义他们自己的排序规则对象， 利用这些条件来满足他们排序行为的需求。请参阅 http://userguide.icu-project.org/locale和 http://userguide.icu-project.org/collation/api 获取有关ICU区域设置命名的信息。可接受的名称和属性集取决于特定的ICU版本。

这里有些例子：

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de@collation=phonebook');
```

德语排序规则和电话簿排序规则类型

第一个例子使用“语言标签”根据 BCP 47选择了ICU区域设置。 第二个示例使用传统的ICU特定区域设置语法。第一种风格是首选， 但它不受旧版ICU支持。

请注意，您可以在SQL环境中任意指定排序规则对象的名称。 在这个例子中，我们遵循预定义排序规则使用的命名风格， 而这种风格又遵循BCP 47，但这对于用户定义的排序规则不是必需的。

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = 'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = '@collation=emoji');
```

根据Unicode技术标准＃51，使用表情符号排序规则类型的根排序规则

观察传统ICU区域命名系统中的方式，根区域设置由空字符串选择。

```
CREATE COLLATION digitslast (provider = icu, locale = 'en-u-kr-latn-digit');
CREATE COLLATION digitslast (provider = icu, locale = 'en@colReorder=latn-digit');
```

在拉丁字母后面排列数字。（默认是字母前的数字。）

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-upper');
CREATE COLLATION upperfirst (provider = icu, locale = 'en@colCaseFirst=upper');
```

在小写字母前面排列大写字母。（默认是小写字母在前面。）

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-kr-latn-digit');
CREATE COLLATION special (provider = icu, locale = 'en@colCaseFirst=upper;colReorder=latn-digit');
```

结合上述两个选项。

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE COLLATION numeric (provider = icu, locale = 'en@colNumeric=yes');
```

数字排序，按数字值排序数字序列，例如： A-21 < A-123（也称为自然排序）。

参阅Unicode 技术标准 #35[1] 和BCP 47[2]获取详细信息。 可能的排序规则类型（co子标签）列表可以在 CLDR 仓库[3]中找到。 区域设置浏览器[4] 可以用于检查一个特定区域设置定义的细节。使用k* 子标签的示例至少要求ICU版本54。

请注意，虽然此系统允许创建“忽略大小写”或“忽略重音符” 或类似（使用ks键）的排序规则， 但PostgreSQL目前不允许这样的排序规则以真正的不区分大小写或不区分重音的方式进行操作。 根据排序规则比较相等但按照字节不相等的任何字符串将根据其字节值进行排序。

> **注意**
>
> 根据设计，ICU几乎可以接受任何字符串作为区域名称， 并使用其文档中描述的后备程序将其与最接近的区域设置相匹配。因此， 如果使用给定ICU安装实际上不支持的功能组合排序规范，则不会有直接反馈。 因此建议创建应用程序级别的测试用例，以检查排序规则定义是否满足需求。

23.2.2.3.3. 复制排序规则

也可以使用命令CREATE COLLATION 从现有的排序规则创建新的排序规则， 这对于能够在应用程序中使用与操作系统无关的排序规则名称、 创建兼容性名称或以更易读的名称使用ICU提供的排序规则很有帮助。例如：

[1] http://unicode.org/reports/tr35/tr35-collation.html
[2] https://tools.ietf.org/html/bcp47
[3] http://www.unicode.org/repos/cldr/trunk/common/bcp47/collation.xml
[4] https://ssl.icu-project.org/icu-bin/locexp

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

23.3. 字符集支持

PostgreSQL里面的字符集支持你能够以各种字符集存储文本，包括单字节字符集，比如 ISO 8859 系列，以及多字节字符集 ，比如EUC（扩展 Unix 编码 Extended Unix Code）、UTF-8 和 Mule 内部编码。所有被支持的字符集都可以被客户端透明地使用，但少数只能在服务器上使用（即作为一种服务器方编码）。默认的字符集是在使用 initdb初始化你的PostgreSQL数据库集簇时选择的。在你创建一个数据库时可以重载它，因此你可能会有多个数据库并且每一个使用不同的字符集。

但是，一个重要的限制是每个数据库的字符集必须和数据库的LC_CTYPE （字符分类）和LC_COLLATE （字符串排序顺序）设置兼容。对于 C或POSIX环境，任何字符集都是允许的， 但是对于其他libc提供的环境只有一种字符集可以正确工作（不过， 在Windows上UTF-8编码可以和任何环境配合使用）。 如果您配置了ICU支持，则ICU提供的区域设置可用于大多数服务器端编码， 但不能用于所有服务器端编码。

23.3.1. 被支持的字符集

表 23.显示了PostgreSQL中可用的字符集。

表 23.1. PostgreSQL字符集

名称	描述	语言	是否服务器端?	ICU?	字节/字符	别名
BIG5	Big Five	繁体中文	否	否	1-2	WIN950, Windows950
EUC_CN	扩展UNIX编码-中国	简体中文	是	是	1-3	
EUC_JP	扩展UNIX编码-日本	日文	是	是	1-3	
EUC_JIS_2004	扩展UNIX编码-日本, JIS X 0213	日文	是	否	1-3	
EUC_KR	扩展UNIX编码-韩国	韩文	是	是	1-3	
EUC_TW	扩展UNIX编码-台湾	繁体中文，台湾话	是	是	1-3	
GB18030	国家标准	中文	否	否	1-4	
GBK	扩展国家标准	简体中文	否	否	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	拉丁语/西里尔语	是	是	1	
ISO_8859_6	ISO 8859-6, ECMA 114	拉丁语/阿拉伯语	是	是	1	
ISO_8859_7	ISO 8859-7, ECMA 118	拉丁语/希腊语	是	是	1	

名称	描述	语言	是否服务器端?	ICU?	字节/字符	别名
ISO_8859_8	ISO 8859-8, ECMA 121	拉丁语/希伯来语	是	是	1	
JOHAB	JOHAB	韩语	否	否	1-3	
KOI8R	KOI8-R	西里尔语（俄语）	是	是	1	KOI8
KOI8U	KOI8-U	西里尔语（乌克兰语）	是	是	1	
LATIN1	ISO 8859-1, ECMA 94	西欧	是	是	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	中欧	是	是	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	南欧	是	是	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	北欧	是	是	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	土耳其语	是	是	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	日耳曼语	是	是	1	ISO885910
LATIN7	ISO 8859-13	波罗的海	是	是	1	ISO885913
LATIN8	ISO 8859-14	凯尔特语	是	是	1	ISO885914
LATIN9	ISO 8859-15	带欧罗巴和口音的 LATIN1	是	是	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	罗马尼亚语	是	否	1	ISO885916
MULE_INTERNAL	Mule内部编码	多语种编辑器	是	否	1-4	
SJIS	Shift JIS	日语	否	否	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日语	否	否	1-2	
SQL_ASCII	未指定（见文本）	任意	是	否	1	

名称	描述	语言	是否服务器端?	ICU?	字节/字符	别名
UHC	统一韩语编码	韩语	否	否	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	所有	是	是	1-4	Unicode
WIN866	Windows CP866	西里尔语	是	是	1	ALT
WIN874	Windows CP874	泰语	是	否	1	
WIN1250	Windows CP1250	中欧	是	是	1	
WIN1251	Windows CP1251	西里尔语	是	是	1	WIN
WIN1252	Windows CP1252	西欧	是	是	1	
WIN1253	Windows CP1253	希腊语	是	是	1	
WIN1254	Windows CP1254	土耳其语	是	是	1	
WIN1255	Windows CP1255	希伯来语	是	是	1	
WIN1256	Windows CP1256	阿拉伯语	是	是	1	
WIN1257	Windows CP1257	波罗的海	是	是	1	
WIN1258	Windows CP1258	越南语	是	是	1	ABC, TCVN, TCVN5712, VSCII

并非所有的客户端API都支持上面列出的字符集。比如，PostgreSQL的JDBC 驱动就不支持MULE_INTERNAL、LATIN6、LATIN8和LATIN10。

SQL_ASCII设置与其他设置表现得相当不同。如果服务器字符集是SQL_ASCII，服务器把字节值0-127根据 ASCII标准解释，而字节值128-255则当作无法解析的字符。如果设置为SQL_ASCII，就不会有编码转换。因此，这个设置基本不是用来声明所使用的指定编码，因为这个声明会忽略编码。在大多数情况下，如果你使用了任何非ASCII数据，那么使用SQL_ASCII设置都是不明智的，因为PostgreSQL将无法帮助你转换或者校验非ASCII字符。

23.3.2. 设置字符集

initdb为一个PostgreSQL集簇定义缺省的字符集（编码）。比如：

```
initdb -E EUC_JP
```

把缺省字符集设置为EUC_JP（用于日文的扩展Unix 编码）。如果你喜欢用长选项字符串，你可以用--encoding代替-E。 如果没有给出-E或者--encoding选项，initdb会尝试基于指定的或者默认的区域判断要使用的合适编码。

你可以在数据库创建时指定一个非默认编码，提供的编码应和选择的区域兼容：

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr
 korean
```

将创建一个使用EUC_KR字符集和ko_KR区域的名为korean的数据库。 另外一种实现方法是使用 SQL 命令：

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
 LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

注意上述命令指定拷贝template0数据库。在拷贝任何其他数据库时，不能更改从源数据库得来的编码和区域设置，因为这可能会导致破坏数据。详见第 22.3 节

数据库的编码存储在系统目录pg_database中。你可以使用psql -l选项或者\l命令来查看。

```
$ psql -l
                                  List of databases
   Name    |  Owner   | Encoding |  Collation  |    Ctype    |          Access
 Privileges
-----------+----------+----------+-------------+-------------
+-----------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

> **重要**
>
> 在大部分现代操作系统上，PostgreSQL可以判断LC_CTYPE设置意味着哪一种字符集，并且它强制只有匹配的数据库编码被使用。在老的系统上你需要自己负责确保所使用的编码就是你所选择的区域所期望的。在这里的一个错误很可能导致区域依赖的操作产生奇怪的行为，例如排序。
>
> 即使LC_CTYPE不是C或POSIX时，PostgreSQL将允许超级用户使用SQL_ASCII编码创建数据库。正如前文所述，SQL_ASCII并不强制存储在数据库中的数据具有任何特定的编码，并且这样这种选择存在着区域依赖的不正当行为的风险。使用这种设置组合的做法已经被废弃，并且在某天将被完全禁止。

23.3.3. 服务器和客户端之间的自动字符集转换

PostgreSQL支持一些编码在服务器和前端之间的自动编码转换。转换信息在系统目录pg_conversion中存储。PostgreSQL带着一些预定义的转换，如表 23.2所示。你可以使用SQL命令CREATE CONVERSION创建一个新的转换。

表 23.2. 客户/服务器字符集转换

服务器字符集	可用的客户端字符集
BIG5	不支持作为一个服务器编码
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8

服务器字符集	可用的客户端字符集
EUC_JIS_2004	EUC_JIS_2004, SHIFT_JIS_2004, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	不支持作为一个服务器编码
GBK	不支持作为一个服务器编码
ISO_8859_5	ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	不支持作为服务端编码
KOI8R	KOI8R, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	KOI8U, UTF8
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	不支持作为一个服务器编码
SHIFT_JIS_2004	不支持作为一个服务器编码
SQL_ASCII	任意（不会执行任何转换）
UHC	不支持作为一个服务器编码
UTF8	所有支持的编码
WIN866	WIN866, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8

服务器字符集	可用的客户端字符集
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

要想启用自动字符集转换功能，你必须告诉PostgreSQL你想在客户端使用的字符集（编码）。你可以用好几种方法来完成：

- 用psql里的\encoding命令。\encoding允许你动态修改客户端编码。比如，把编码改变为SJIS，键入：

  ```
  \encoding SJIS
  ```

- libpq（见第 34.10 节中提供函数控制客户端编码。

- 使用SET client_encoding TO。 可以使用这个SQL命令设置客户端编码：

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  你还可以把标准SQL语法里的SET NAMES用于这个目的：

  ```
  SET NAMES 'value';
  ```

  要查询当前客户端编码：

  ```
  SHOW client_encoding;
  ```

  要返回到缺省编码：

  ```
  RESET client_encoding;
  ```

- 使用PGCLIENTENCODING。如果在客户端的环境里定义了PGCLIENTENCODING环境变量， 那么在与服务器进行了连接后将自动选择客户端编码（这个设置随后可以用上文提到的任何其他方法重载）。

- 使用client_encoding配置变量。如果client_encoding变量被设置， 那么在与服务器建立了连接之后，这个客户端编码将备自动选定（这个设置随后可以用上文提到的其他方法重载）。

假如无法进行一个特定字符的转换 — 假如你选的服务器编码是EUC_JP而 客户端是LATIN1，那么有些日文字符不能转换成LATIN1 — 将会报告一个错误。

如果客户端字符集定义成了SQL_ASCII，那么编码转换会被禁用， 不管服务器的字符集是什么都一样。和服务器一样，除非你的工作环境全部是 ASCII 数据， 否则使用SQL_ASCII是不明智的。

23.3.4. 进一步阅读

下面是学习各种类型的编码系统的好资源。

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

包含对EUC_JP、 EUC_CN、EUC_KR、 EUC_TW的详细解释。

http://www.unicode.org/

Unicode联盟的网站。

RFC 3629

UTF-8 （8-bit UCS/Unicode转换格式）在这里定义。

第 24 章 日常数据库维护工作

和任何数据库软件一样，PostgreSQL需要定期执行特定的任务来达到最优的性能。这里讨论的任务是必需的，但它们本质上是重复性的并且可以很容易使用cron脚本或Windows的任务计划程序等标准工具来自动进行。建立合适的脚本并检查它们是否成功运行是数据库管理员的职责。

一个显而易见的维护任务是定期创建数据的后备拷贝。如果没有一个最近的备份，你就不可能在灾难（磁盘失败、或在、错误地删除一个关键表等）后进行恢复。PostgreSQL中的备份和恢复机制在第 25 章有详细的介绍。

另一种主要类型的维护任务是周期性地“清理”数据库。该活动在第 24.1 节讨论。与之相关，更新将被查询规划器使用的统计信息的活动将在第 24.1.3 节讨论。

另一项需要周期性考虑的任务是日志文件管理。这在第 24.3 节讨论。

check_postgres[1]可用于检测数据库的健康并报告异常情况。check_postgres与Nagios和MRTG整合在一起，但也可以被单独运行。

相对于其他数据库管理系统，PostgreSQL的维护量较低。但是，适当对这些任务加以注意将大有助于愉快和高效地使用该系统。

24.1. 日常清理

PostgreSQL数据库要求周期性的清理维护。对于很多安装，让自动清理守护进程来执行清理已经足够，如第 24.1.6 节所述。你可能需要调整其中描述的自动清理参数来获得最佳结果。某些数据库管理员会希望使用手动管理的VACUUM命令来对后台进程的活动进行补充或者替换，这通常使用cron或任务计划程序脚本来执行。要正确地设置手动管理的清理，最重要的是理解接下来几小节中讨论的问题。依赖自动清理的管理员最好也能略读该内容以帮助他们理解和调整自动清理。

24.1.1. 清理的基础知识

PostgreSQL的VACUUM命令出于几个原因必须定期处理每一个表：

1. 恢复或重用被已更新或已删除行所占用的磁盘空间。

2. 更新被PostgreSQL查询规划器使用的数据统计信息。

3. 更新可见性映射，它可以加速只用索引的扫描。

4. 保护老旧数据不会由于事务ID回卷或多事务ID回卷而丢失。

正如后续小节中解释的，每一个原因都将指示以不同的频率和范围执行VACUUM操作。

有两种VACUUM的变体：标准VACUUM和VACUUM FULL。VACUUM FULL可以收回更多磁盘空间但是运行起来更慢。另外，标准形式的VACUUM可以和生产数据库操作并行运行（SELECT、INSERT、UPDATE和DELETE等命令将继续正常工作，但在清理期间你无法使用ALTER TABLE等命令来更新表的定义）。VACUUM FULL要求在其工作的表上得到一个排他锁，因此无法和对此表的其他使用并行。因此，通常管理员应该努力使用标准VACUUM并且避免VACUUM FULL。

VACUUM会产生大量I/O流量，这将导致其他活动会话性能变差。可以调整一些配置参数来后台清理活动造成的性能冲击 — 参阅第 19.4.4 节

24.1.2. 恢复磁盘空间

[1] https://bucardo.org/check_postgres/

在PostgreSQL中，一次行的UPDATE或DELETE不会立即移除该行的旧版本。这种方法对于从多版本并发控制（MVCC，见第 13 章获益是必需的：当旧版本仍可能对其他事务可见时，它不能被删除。但是最后，任何事务都不会再对一个过时的或者被删除的行版本感兴趣。它所占用的空间必须被回收来用于新行，这样可避免磁盘空间需求的无限制增长。这通过运行VACUUM完成。

VACUUM的标准形式移除表和索引中的死亡行版本并将该空间标记为可在未来重用。不过，它将不会把该空间交还给操作系统，除非在特殊的情况中表尾部的一个或多个页面变成完全空闲并且能够很容易地得到一个排他表锁。相反，VACUUM FULL通过把死亡空间之外的内容写成一个完整的新版本表文件来主动紧缩表。这将最小化表的尺寸，但是要花较长的时间。它也需要额外的磁盘空间用于表的新副本，直到操作完成。

例行清理的一般目标是多做标准的VACUUM来避免需要VACUUM FULL。自动清理守护进程尝试这样工作，并且实际上永远不会发出VACUUM FULL。在这种方法中，其思想不是让表保持它们的最小尺寸，而是保持磁盘空间使用的稳定状态：每个表占用的空间等于其最小尺寸外加清理之间被用完的空间。尽管VACUUM FULL可被用来把一个表收缩回它的最小尺寸并将该磁盘空间交还给操作系统，但是如果该表将在未来再次增长这样就没什么意义。因此，对于维护频繁被更新的表，适度运行标准VACUUM运行比少量运行VACUUM FULL要更好。

一些管理员更喜欢自己计划清理，例如在晚上负载低时做所有的工作。根据一个固定日程来做清理的难点在于，如果一个表有一次预期之外的更新活动尖峰，它可能膨胀得真正需要VACUUM FULL来回收空间。使用自动清理守护进程可以减轻这个问题，因为守护进程会根据更新活动动态规划清理操作。除非你的负载是完全可以预估的，完全禁用守护进程是不理智的。一种可能的折中方案是设置守护进程的参数，这样它将只对异常的大量更新活动做出反应，因而保证事情不会失控，而在负载正常时采用有计划的VACUUM来做批量工作。

对于那些不使用自动清理的用户，一种典型的方法是计划一个数据库范围的VACUUM，该操作每天在低使用量时段执行一次，并根据需要辅以在重度更新表上的更频繁的清理（一些有着极高更新率的安装会每几分钟清理一次它们的最繁忙的表）。如果你在一个集簇中有多个数据库，别忘记VACUUM每一个，你会用得上vacuumdb程序。

> **提示**
>
> 当一个表因为大量更新或删除活动而包含大量死亡行版本时，纯粹的VACUUM可能不能令人满意。如果你有这样一个表并且你需要回收它占用的过量磁盘空间，你将需要使用VACUUM FULL，或者CLUSTER，或者ALTER TABLE的表重写变体之一。这些命令重写该表的一整个新拷贝并且为它构建新索引。所有这些选项都要求排他锁。注意它们也临时使用大约等于该表尺寸的额外磁盘空间，因为直到新表和索引完成之前旧表和索引都不能被释放。

> **提示**
>
> 如果你有一个表，它的整个内容会被周期性删除，考虑用TRUNCATE而不是先用DELETE再用VACUUM。TRUNCATE会立刻移除该表的整个内容，而不需要一次后续的VACUUM或VACUUM FULL来回收现在未被使用的磁盘空间。其缺点是会违背严格的 MVCC 语义。

24.1.3. 更新规划器统计信息

PostgreSQL查询规划器依赖于有关表内容的统计信息来为查询产生好的计划。这些统计信息由ANALYZE命令收集，它除了直接被调用之外还可以作为VACUUM的一个可选步骤被调用。拥有适度准确的统计信息很重要，否则差的计划可能降低数据库性能。

自动清理守护进程如果被启用，当一个表的内容被改变得足够多时，它将自动发出ANALYZE命令。不过，管理员可能更喜欢依靠手动的ANALYZE操作，特别是如果知道一个表

上的更新活动将不会影响“感兴趣的”列的统计信息时。守护进程严格地按照一个被插入或更新行数的函数来计划ANALYZE，它不知道那是否将导致有意义的统计信息改变。

正如用于空间恢复的清理一样，频繁更新统计信息对重度更新的表更加有用。但即使对于一个重度更新的表，如果该数据的统计分布没有很大改变，也没有必要更新统计信息。一个简单的经验法则是考虑表中列的最大和最小值改变了多少。例如，一个包含行被更新时间的timestamp列将在行被增加和更新时有一直增加的最大值；这样一列将可能需要更频繁的统计更新，而一个包含一个网站上被访问页面 URL 的列则不需要。URL 列可以经常被更改，但是其值的统计分布的变化相对很慢。

可以在指定表上运行ANALYZE甚至在表的指定列上运行，因此如果你的应用需要，可以更加频繁地更新某些统计。但实际上，通常只分析整个数据库是最好的，因为它是一种很快的操作。ANALYZE对一个表的行使用一种统计的随机采样，而不是读取每一个单一行。

> **提示**
>
> 尽管对每列的ANALYZE频度调整可能不是非常富有成效，你可能会发现值得为每列调整被ANALYZE收集统计信息的详细程度。经常在WHERE中被用到的列以及数据分布非常不规则的列可能需要比其他列更细粒度的数据直方图。见ALTER TABLE SET STATISTICS，或者使用default_statistics_target配置参数改变数据库范围的默认值。
>
> 还有，默认情况下关于函数的选择度的可用信息是有限的。但是，如果你创建一个使用函数调用的表达式索引，关于该函数的有用的统计信息将被收集，这些信息能够大大提高使用该表达式索引的查询计划的质量。

> **提示**
>
> 自动清理守护进程不会为外部表发出ANALYZE命令，因为无法确定一个合适的频度。如果你的查询需要外部表的统计信息来正确地进行规划，比较好的方式是按照一个合适的时间表在那些表上手工运行ANALYZE命令。

24.1.4. 更新可见性映射

清理机制为每一个表维护着一个可见性映射，它被用来跟踪哪些页面只包含对所有活动事务（以及所有未来的事务，直到该页面被再次修改）可见的元组。这样做有两个目的。第一，清理本身可以在下一次运行时跳过这样的页面，因为其中没有什么需要被清除。

第二，这允许PostgreSQL回答一些只用索引的查询，而不需要引用底层表。因为PostgreSQL的索引不包含元组的可见性信息，一次普通的索引扫描会为每一个匹配的索引项获取堆元组，用来检查它是否能被当前事务所见。另一方面，一次只用索引的扫描会首先检查可见性映射。如果它了解到在该页面上的所有元组都是可见的，堆获取就可以被跳过。这对大数据集很有用，因为可见性映射可以防止磁盘访问。可见性映射比堆小很多，因此即使堆非常大，可见性映射也可以很容易地被缓存起来。

24.1.5. 防止事务 ID 回卷失败

PostgreSQL的 MVCC 事务语义依赖于能够比较事务 ID（XID）数字：如果一个行版本的插入 XID 大于当前事务的 XID，它就是“属于未来的”并且不应该对当前事务可见。但是因为事务 ID 的尺寸有限（32位），一个长时间（超过 40 亿个事务）运行的集簇会遭受到事务 ID 回卷问题：XID 计数器回卷到 0，并且本来属于过去的事务突然间就变成了属于未来 — 这意味着它们的输出变成不可见。简而言之，灾难性的数据丢失（实际上数据仍然在那里，但是如果你不能得到它也无济于事）。为了避免发生这种情况，有必要至少每 20 亿个事务就清理每个数据库中的每个表。

周期性的清理能够解决该问题的原因是，VACUUM会把行标记为 冻结，这表示它们是被一个在足够远的过去提交的事务所插入， 这样从 MVCC 的角度来看，效果就是该插入事务对所有当前和未来事务来说当然都 是可见的。PostgreSQL保留了一个特殊的 XID（FrozenTransactionId），这个 XID 并不遵循普通 XID 的比较规则 并且总是被认为比任何普通 XID 要老。普通 XID 使用模-2^{32}算 法来比较。这意味着对于每一个普通 XID都有 20 亿个 XID “更老”并且 有 20 亿个“更新”，另一种解释的方法是普通 XID 空间是没有端点的环。 因此，一旦一个行版本创建时被分配了一个特定的普通 XID，该行版本将成为接下 来 20 亿个事务的“过去”（与我们谈论的具体哪个普通 XID 无关）。如 果在 20 亿个事务之后该行版本仍然存在，它将突然变得好像在未来。要阻止这一切 发生，被冻结行版本会被看成其插入 XID 为FrozenTransactionId， 这样它们对所有普通事务来说都是“在过去”，而不管回卷问题。并且这样 的行版本将一直有效直到被删除，不管它有多旧。

> **注意**
>
> 在9.4之前的PostgreSQL版本中，实际上会通过将一行的插入 XID 替换为FrozenTransactionId来实现冻结，这种FrozenTransactionId在行的 xmin系统列中是可见的。较新的版本只是设置一个标志位， 保留行的原始xmin用于可能发生的鉴别用途。不过， 在9.4之前版本的数据库pg_upgrade中可能仍会找到 xmin等于FrozenTransactionId (2)的行。
>
> 此外，系统目录可能会包含xmin等于BootstrapTransactionId (1) 的行，这表示它们是在initdb的第一个阶段被插入的。 和FrozenTransactionId相似，这个特殊的 XID 被认为比所有正常 XID 的年龄都要老。

vacuum_freeze_min_age控制在其行版本被冻结前一个 XID 值应该有多老。如果被冻结的行将很快会被再次修改，增加这个设置可以避免不必要 的工作。但是减少这个设置会增加在表必须再次被清理之前能够流逝的事务数。

VACUUM通常会跳过不含有任何死亡行版本的页面，但是不会跳过那些含有带旧 XID 值的行版本的页面。要保证所有旧的行版本都已经被冻结，需要对整个表做一次扫描。vacuum_freeze_table_age控制VACUUM什么时候这样做：如果该表经过vacuum_freeze_table_age减去vacuum_freeze_min_age个事务还没有被完全扫描过，则会强制一次全表清扫。将这个参数设置为 0 将强制VACUUM总是扫描所有页面而实际上忽略可见性映射。

一个表能保持不被清理的最长时间是 20 亿个事务减去VACUUM上次扫描全表时的vacuum_freeze_min_age值。如果它超过该时间没有被清理，可能会导致数据丢失。要保证这不会发生，将在任何包含比autovacuum_freeze_max_age配置参数所指定的年龄更老的 XID 的未冻结行的表上调用自动清理（即使自动清理被禁用也会发生）。

这意味着如果一个表没有被清理，大约每autovacuum_freeze_max_age减去vacuum_freeze_min_age事务就会在该表上调用一次自动清理。对那些为了空间回收目的而被正常清理的表，这是无关紧要的。然而，对静态表（包括接收插入但没有更新或删除的表）就没有为空间回收而清理的需要，因此尝试在非常大的静态表上强制自动清理的间隔最大化会非常有用。显然我们可以通过增加autovacuum_freeze_max_age或减少vacuum_freeze_min_age来实现此目的。

vacuum_freeze_table_age的实际最大值是 0.95 * autovacuum_freeze_max_age，高于它的设置将被上限到最大值。一个高于autovacuum_freeze_max_age的值没有意义，因为不管怎样在那个点上都会触发一次防回卷自动清理，并且 0.95 的乘数为在防回卷自动清理发生之前运行一次手动VACUUM留出了一些空间。作为一种经验法则，vacuum_freeze_table_age应当被设置成一个低于autovacuum_freeze_max_age的值，留出一个足够的空间让一次被正常调度的VACUUM或一次被正常删除和更新活动触发的自动清理可以在这个窗口中被运行。将它设置得太接近可能导致防回卷自动清理，即使该表最近因为回收空间的目的被清理过，而较低的值将导致更频繁的全表扫描。

增加autovacuum_freeze_max_age（以及和它一起的vacuum_freeze_table_age）的唯一不足是数据库集簇的pg_xact和pg_commit_ts子目录将占据更多空间，因为它必须存储所有向

后autovacuum_freeze_max_age范围内的所有事务的提交状态和（如果启用了track_commit_timestamp）时间戳。提交状态为每个事务使用两个二进制位，因此如果autovacuum_freeze_max_age被设置为它的最大允许值 20 亿，pg_xact将会增长到大约0.5 吉字节，pg_commit_ts大约20GB。如果这对于你的总数据库尺寸是微小的，我们推荐设置autovacuum_freeze_max_age为它的最大允许值。否则，基于你想要允许pg_xact和pg_commit_ts使用的存储空间大小来设置它（默认情况下 2 亿个事务大约等于pg_xact的 50 MB存储空间，pg_commit_ts的2GB的存储空间）。

减小vacuum_freeze_min_age的一个不足之处是它可能导致VACUUM做无用的工作：如果该行在被替换成FrozenXID之后很快就被修改（导致该行获得一个新的 XID），那么冻结一个行版本就是浪费时间。因此该设置应该足够大，这样直到行不再可能被修改之前，它们都不会被冻结。

为了跟踪一个数据库中最老的未冻结 XID 的年龄，VACUUM在系统表pg_class和pg_database中存储 XID 的统计信息。特别地，一个表的pg_class行的relfrozenxid列包含被该表的上一次全表VACUUM所用的冻结截止 XID。该表中所有被有比这个截断 XID 老的普通 XID 的事务插入的行 都确保被冻结。相似地，一个数据库的pg_database行的datfrozenxid列是出现在该数据库中的未冻结 XID 的下界— 它只是数据库中每一个表的relfrozenxid值的最小值。一种检查这些信息的方便方法是执行这样的查询：

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

age列度量从该截断 XID 到当前事务 XID 的事务数。

VACUUM通常只扫描从上次清理后备修改过的页面，但是只有当全表被扫描时relfrozenxid才能被推进。当relfrozenxid比vacuum_freeze_table_age个事务还老时、当VACUUM的FREEZE选项被使用时或当所有页面正好要求清理来移除死亡行版本时，全表将被扫描。当VACUUM扫描全表时，在它被完成后，age(relfrozenxid)应该比被使用的vacuum_freeze_min_age设置略大（比在VACUUM开始后开始的事务数多）。如果在autovacuum_freeze_max_age被达到之前没有全表扫描VACUUM在该表上被发出，将很快为该表强制一次自动清理。

如果出于某种原因自动清理无法从一个表中清除旧的 XID，当数据库的最旧 XID 和回卷点之间达到 1 千万个事务时，系统将开始发出这样的警告消息：

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

（如该示意所建议的，一次手动的VACUUM应该会修复该问题；但是注意该次VACUUM必须由一个超级用户来执行，否则它将无法处理系统目录并且因而不能推进数据库的datfrozenxid）。如果这些警告被忽略，一旦距离回卷点只剩下 1 百万个事务时，该系统将会关闭并且拒绝开始任何新的事务：

```
ERROR:  database is not accepting commands to avoid wraparound data loss in
 database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

这一百万个事务的富余是为了让管理员能通过手动执行所要求的VACUUM命令进行恢复而不丢失数据。但是，由于一旦系统进入到安全关闭模式，它将不会执行命令。做这个操作的唯一

方法是停止服务器并且以单一用户启动服务器来执行VACUUM。单一用户模式中不会强制该关闭模式。关于使用单一用户模式的细节请见postgres参考页。

24.1.5.1. 多事务和回卷

Multixact ID被用来支持被多个事务锁定的行。由于在一个元组头部 只有有限的空间可以用来存储锁信息，所以只要有多于一个事务并发地锁住一个行， 锁信息将使用一个“多个事务 ID”（或简称多事务 ID）来编码。任何特定 多事务 ID 中包括的事务 ID 的信息被独立地存储在pg_multixact子目 录中，并且只有多事务 ID 出现在元组头部的xmax域中。和事务 ID 类似，多事务 ID 也是用一个 32 位计数器实现，并且也采用了相似的存储，这些都要 求仔细的年龄管理、存储清除和回卷处理。在每个多事务中都有一个独立的存储区域 保存成员列表，它也使用一个 32 位计数器并且也应被管理。

在一次VACUUM表扫描（部分或者全部）期间，任何比 vacuum_multixact_freeze_min_age 要老的多事务 ID 会被替换为一个不同的值，该值可以是零值、 一个单一事务 ID 或者一个更新的多事务 ID。 对于每一个表，pg_class.relminmxid 存储了在该表任意元组中仍然存在的最老可能多事务 ID。如果这个值比 vacuum_multixact_freeze_table_age老， 将强制一次全表扫描。可以在 pg_class.relminmxid 上使用mxid_age()来找到它的年龄。

全表VACUUM扫描（不管是什么导致它们）将为表推进该值。 最后，当所有数据库中的所有表被扫描并且它们的最老多事务值被推进， 较老的多事务的磁盘存储可以被移除。

作为一种安全设备，对任何多事务年龄超过 autovacuum_multixact_freeze_max_age的表，都将发生一次全表清理扫描。如果已用的成员存储空间超过总量的 50%，全表清理扫描 也将逐步在所有表上进行，这会从那些具有最老多事务年龄的表开始。即使自动清理被 在名义上被禁用，这两中类型的全表扫描都将会发生。

24.1.6. 自动清理后台进程

PostgreSQL有一个可选的但是被高度推荐的特性autovacuum，它的目的是自动执行VACUUM和ANALYZE 命令。当它被启用时，自动清理会检查被大量插入、更新或删除元组的表。这些检查会利用统计信息收集功能，因此除非track_counts被设置为true，自动清理不能被使用。在默认配置下，自动清理是被启用的并且相关配置参数已被正确配置。

“自动清理后台进程”实际上由多个进程组成。有一个称为 自动清理启动器的常驻后台进程， 它负责为所有数据库启动自动清理工作者进程。 启动器将把工作散布在一段时间上，它每隔 autovacuum_naptime秒尝试在每个数据库中启动一个工作者 （因此，如果安装中有N个数据库，则每 autovacuum_naptime/N秒将启动一个新的工作者）。 在同一时间只允许最多autovacuum_max_workers 个工作者进程运行。如果有超过autovacuum_max_workers个数据库需要被处理，下一个数据库将在第一个工作者结束后马上被处理。 每一个工作者进程将检查其数据库中的每一个表并且在需要时执行 VACUUM和/或ANALYZE。 可以设置log_autovacuum_min_duration 来监控自动清理工作者的活动。

如果在一小段时间内多个大型表都变得可以被清理，所有的自动清理工作者可能都会被占用来在一段长的时间内清理这些表。这将会造成其他的表和数据库无法被清理，直到一个工作者变得可用。对于一个数据库中的工作者数量并没有限制，但是工作者确实会试图避免重复已经被其他工作者完成的工作。注意运行着的工作者的数量不会被计入max_connections或superuser_reserved_connections限制。

relfrozenxid值比autovacuum_freeze_max_age事务年龄更大的表总是会被清理（这页表示这些表的冻结最大年龄被通过表的存储参数修改过，参见后文）。否则，如果从上次VACUUM以来失效的元组数超过“清理阈值”，表也会被清理。清理阈值定义为：

清理阈值 = 清理基本阈值 + 清理缩放系数 * 元组数

其中清理基本阈值为autovacuum_vacuum_threshold， 清理缩放系数为autovacuum_vacuum_scale_factor， 元组数为pg_class.reltuples。 失效元组的数量从统计信息收集器获得，它是一个由每个UPDATE和DELETE命令更新的半准确的计数（它只是半

准确，是因为在高负载的情况下某些信息可能会丢失）。如果表的relfrozenxid值比vacuum_freeze_table_age事务年龄更大，整个表将被扫描以冻结旧元组并增长relfrozenxid，否则只有从上次清理以来被修改的页面会被扫描。

对于分析，也使用了一个相似的阈值：

分析阈值 = 分析基本阈值 + 分析缩放系数 * 元组数

该阈值将与自从上次ANALYZE以来被插入、更新或删除的元组数进行比较。

临时表不能被自动清理访问。因此，临时表的清理和分析操作必须通过会话期间的SQL命令来执行。

默认的阈值和缩放系数都取自于postgresql.conf， 但是可以为每一个表重写它们(和许多其他自动清理控制参数)， 详情参见存储参数。 如果一个设置已经通过一个表的存储参数修改，那么在处理该表时使用该值， 否则使用全局设置。全局设置请参阅第 19.10 节

当多个工作者运行时，在所有运行着的工作者之间自动清理代价延迟参数 （参阅第 19.4.4 节）是 “平衡的”，这样不管实际运行的工作者数量是多少， 对于系统的总体 I/O 影响总是相同的。不过，任何正在处理已经设置了每表autovacuum_vacuum_cost_delay或 autovacuum_vacuum_cost_limit 存储参数的表的工作者不会被考虑在均衡算法中。

24.2. 日常重建索引

在某些情况下值得周期性地使用REINDEX命令或一系列独立重构步骤来重建索引。

已经完全变成空的B树索引页面被收回重用。但是，还是有一种低效的空间利用的可能性：如果一个页面上除少量索引键之外的全部键被删除，该页面仍然被分配。因此，在这种每个范围中大部分但不是全部键最终被删除的使用模式中，可以看到空间的使用是很差的。对于这样的使用模式，推荐使用定期重索引。

对于非B树索引可能的膨胀还没有很好地定量分析。在使用非B树索引时定期监控索引的物理尺寸是个好主意。

还有，对于B树索引，一个新建立的索引比更新了多次的索引访问起来要略快， 因为在新建立的索引上，逻辑上相邻的页面通常物理上也相邻（这样的考虑目前并不适用于非B树索引）。仅仅为了提高访问速度也值得定期重索引。

REINDEX在所有情况下都可以安全和容易地使用。但是由于该命令要求一个排他表锁，因此更好的方法是用一个由创建和替换步骤组成的序列来执行索引重建。支持带CONCURRENTLY选项的CREATE INDEX的索引类型可以用这种方式重建。如果创建成功并且得到的索引是可用的，则原来的索引可以使用ALTER INDEX和DROP INDEX的命令组合替换成新创建的索引。当一个索引被用于强制唯一性或者其他约束时，可能需要用ALTER TABLE将现有的约束换成由新索引所强制的约束。在使用这种多步重建方法之前应仔细地检查，因为对于哪些索引可以采用这种方法重索引是有限制的，并且出现的错误必须被处理。

24.3. 日志文件维护

把数据库服务器的日志输出保存在一个地方是个好主意， 而不是仅仅通过/dev/null丢弃它们。 在进行问题诊断的时候，日志输出是非常宝贵的。不过，日志输出可能很庞大（特别是在比较高的调试级别上）， 因此你不会希望无休止地保存它们。你需要轮转日志文件，这样在一段合理的时间后会开始新的日志文件并且移除旧的。

如果你简单地把postgres的stderr定向到一个文件中，你会得到日志输出， 但是截断该日志文件的唯一方法是停止并重起服务器。这样做对于开发环境中使用的PostgreSQL可能是可接受的，但是你肯定不想在生产环境上这么干。

一个更好的办法是把服务器的stderr输出发送到某种日志轮转程序里。 我们有一个内建的日志轮转程序，你可以通过在 postgresql.conf里设置配置参数logging_collector为true的办法启用它。该程序的控制参数在 第 19.8.1 节描述。你也可以使用这种方法把日志数据捕捉成机器可读的CSV（逗号分隔值）格式。

另外，如果在你已经使用的其他服务器软件中有一个外部日志轮转程序，你可能更喜欢使用它。 比如，包含在Apache发布里的rotatelogs工具就可以用于PostgreSQL。 要这么做，只需要把服务器的stderr用管道重定向到要用的程序。 如果你用pg_ctl启动服务器，那么stderr已经重定向到stdout， 因此你只需要一个管道命令，比如：

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

另外一种生产级的管理日志输出的方法就是把它们发送给syslog，让syslog处理文件轮转。要利用这个工具，我们需要设置postgresql.conf里的log_destination配置参数设置为syslog（记录syslog日志）。然后在你想强迫syslog守护进程开始写入一个新日志文件的时候， 你就可以发送一个 SIGHUP信号给它。 如果你想自动进行日志轮转，可以配置logrotate程序处理 来自syslog的日志文件。

不过，在很多系统上，syslog不是非常可靠，特别是在面对大量日志消息的情况下； 它可能在你最需要那些消息的时候截断或者丢弃它们。另外，在Linux，syslog会把每个消息刷写到磁盘上， 这将导致很差的性能（你可以在syslog配置文件里面的文件名开头使用一个“-”来禁用这种行为）。

请注意上面描述的所有解决方案关注的是在可配置的间隔上开始一个新的日志文件， 但它们并没有处理对旧的、不再需要的日志文件的删除。你可能还需要设置一个批处理任务来定期地删除旧日志文件。 另一种可能的方法是配置日志轮转程序，让它循环地覆盖旧的日志文件。

pgBadger[2]是一个外部项目，它可以进行日志文件的深度分析。 check_postgres[3]可在重要消息出现在日志文件中时向Nagios提供警告，也可以探测很多其他的特别情况。

[2] https://pgbadger.darold.net/
[3] https://bucardo.org/check_postgres/

第 25 章 备份和恢复

由于包含着有价值的数据，PostgreSQL数据库应当被定期地备份。虽然过程相当简单，但清晰地理解其底层技术和假设是非常重要的。

有三种不同的基本方法来备份PostgreSQL数据：

- SQL转储
- 文件系统级备份
- 连续归档

每一种都有其优缺点，在下面的小节中将分别讨论。

25.1. SQL转储

SQL 转储方法的思想是创建一个由SQL命令组成的文件，当把这个文件回馈给服务器时，服务器将利用其中的SQL命令重建与转储时状态一样的数据库。 PostgreSQL为此提供了工具pg_dump。这个工具的基本用法是：

```
pg_dump dbname > dumpfile
```

正如你所见，pg_dump把结果输出到标准输出。我们后面将看到这样做有什么用处。 尽管上述命令会创建一个文本文件，pg_dump可以用其他格式创建文件以支持并行 和细粒度的对象恢复控制。

pg_dump是一个普通的PostgreSQL客户端应用（尽管是个 相当聪明的东西）。这就意味着你可以在任何可以访问该数据库的远端主机上进行备份工作。但是请记住 pg_dump不会以任何特殊权限运行。具体说来，就是它必须要有你想备份的表的读 权限，因此为了备份整个数据库你几乎总是必须以一个数据库超级用户来运行它（如果你没有足够的特权 来备份整个数据库，你仍然可以使用诸如-n schema 或-t table选项来备份该数据库中你能够 访问的部分）。

要声明pg_dump连接哪个数据库服务器，使用命令行选项-hhost和 -p port。 默认主机是本地主机或你的PGHOST环境变量指定的主机。 类似地，默认端口是环境变量PGPORT或（如果PGPORT不存在）内建的默认值。 （服务器通常有相同的默认值，所以还算方便。）

和任何其他PostgreSQL客户端应用一样， pg_dump默认使用与当前操作系统用户名同名的数据库用户名进行连接。 要使用其他名字，要么声明-U选项，要么设置环境变量PGUSER。请注意pg_dump的连接也要通过客户认证机制（在第 20 章描述）。

pg_dump对于其他备份方法的一个重要优势是，pg_dump的输出可以很容易地在新版本的PostgreSQL中载入，而文件级备份和连续归档都是极度的服务器版本限定的。pg_dump也是唯一可以将一个数据库传送到一个不同机器架构上的方法，例如从一个32位服务器到一个64位服务器。

由pg_dump创建的备份在内部是一致的， 也就是说，转储表现了pg_dump开始运行时刻的数据库快照，且在pg_dump运行过程中发生的更新将不会被转储。pg_dump工作的时候并不阻塞其他的对数据库的操作。 （但是会阻塞那些需要排它锁的操作，比如大部分形式的ALTER TABLE）

25.1.1. 从转储中恢复

pg_dump生成的文本文件可以由psql程序读取。 从转储中恢复的常用命令是：

```
psql dbname < dumpfile
```

其中dumpfile就是pg_dump命令的输出文件。这条命令不会创建数据库dbname，你必须在执行psql前自己从template0创建（例如，用命令createdb -T template0 dbname）。psql支持类似pg_dump的选项用以指定要连接的数据库服务器和要使用的用户名。参阅psql的手册获取更多信息。 非文本文件转储可以使用pg_restore工具来恢复。

在开始恢复之前，转储库中对象的拥有者以及在其上被授予了权限的用户必须已经存在。如果它们不存在，那么恢复过程将无法将对象创建成具有原来的所属关系以及权限（有时候这就是你所需要的，但通常不是）。

默认情况下，psql脚本在遇到一个SQL错误后会继续执行。你也许希望在遇到一个SQL错误后让psql退出，那么可以设置ON_ERROR_STOP变量来运行psql，这将使psql在遇到SQL错误后退出并返回状态3：

```
psql --set ON_ERROR_STOP=on dbname < infile
```

不管怎样，你将只能得到一个部分恢复的数据库。作为另一种选择，你可以指定让整个恢复作为一个单独的事务运行，这样恢复要么完全完成要么完全回滚。这种模式可以通过向psql传递-1或--single-transaction命令行选项来指定。在使用这种模式时，注意即使是很小的一个错误也会导致运行了数小时的恢复被回滚。但是，这仍然比在一个部分恢复后手工清理复杂的数据库要更好。

pg_dump和psql读写管道的能力使得直接从一个服务器转储一个数据库到另一个服务器成为可能，例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

> **重要**
>
> pg_dump产生的转储是相对于template0。这意味着在template1中加入的任何语言、过程等都会被pg_dump转储。结果是，如果在恢复时使用的是一个自定义的template1，你必须从template0创建一个空的数据库，正如上面的例子所示。

一旦完成恢复，在每个数据库上运行ANALYZE是明智的举动，这样优化器就有有用的统计数据了，更多信息参见第 24.1.3 和第 24.1.6 节更多关于如何有效地向PostgreSQL里装载大量数据的建议， 请参考第 14.4 节

25.1.2. 使用pg_dumpall

pg_dump每次只转储一个数据库，而且它不会转储关于角色或表空间（因为它们是集簇范围的）的信息。为了支持方便地转储一个数据库集簇的全部内容，提供了pg_dumpall程序。pg_dumpall备份一个给定集簇中的每一个数据库，并且也保留了集簇范围的数据，如角色和表空间定义。该命令的基本用法是：

```
pg_dumpall > dumpfile
```

转储的结果可以使用psql恢复：

```
psql -f dumpfile postgres
```

（实际上，你可以指定恢复到任何已有数据库名，但是如果你正在将转储载入到一个空集簇中则通常要用（postgres）。在恢复一个pg_dumpall转储时常常需要具有数据库超级用户访问权限，因为它需要恢复角色和表空间信息。如果你在使用表空间，请确保转储中的表空间路径适合于新的安装。

pg_dumpall工作时会发出命令重新创建角色、表空间和空数据库，接着为每一个数据库pg_dump。这意味着每个数据库自身是一致的，但是不同数据库的快照并不同步。

集簇范围的数据可以使用pg_dumpall的--globals-only选项来单独转储。如果在单个数据库上运行pg_dump命令，上述做法对于完全备份整个集簇是必需的。

25.1.3. 处理大型数据库

在一些具有最大文件尺寸限制的操作系统上创建大型的pg_dump输出文件可能会出现问题。幸运地是，pg_dump可以写出到标准输出，因此你可以使用标准Unix工具来处理这种潜在的问题。有几种可能的方法：

使用压缩转储。. 你可以使用你喜欢的压缩程序，例如gzip：

```
pg_dump dbname | gzip > filename.gz
```

恢复：

```
gunzip -c filename.gz | psql dbname
```

或者：

```
cat filename.gz | gunzip | psql dbname
```

使用split。. split命令允许你将输出分割成较小的文件以便能够适应底层文件系统的尺寸要求。例如，让每一块的大小为1兆字节：

```
pg_dump dbname | split -b 1m - filename
```

恢复：

```
cat filename* | psql dbname
```

使用pg_dump的自定义转储格式。. 如果PostgreSQL所在的系统上安装了zlib压缩库，自定义转储格式将在写出数据到输出文件时对其压缩。这将产生和使用gzip时差不多大小的转储文件，但是这种方式的一个优势是其中的表可以被有选择地恢复。下面的命令使用自定义转储格式来转储一个数据库：

```
pg_dump -Fc dbname > filename
```

自定义格式的转储不是psql的脚本，只能通过pg_restore恢复，例如：

```
pg_restore -d dbname filename
```

详情请参阅pg_dump和pg_restore。

对于非常大型的数据库，你可能需要将split配合其他两种方法之一进行使用。

使用pg_dump的并行转储特性。. 为了加快转储一个大型数据库的速度，你可以使用pg_dump的并行模式。它将同时转储多个表。你可以使用-j参数控制并行度。并行转储只支持“目录”归档格式。

```
pg_dump -j num -F d -f out.dir dbname
```

你可以使用pg_restore -j来以并行方式恢复一个转储。它只能适合于“自定义”归档或者“目录”归档，但不管归档是否由pg_dump -j创建。

25.2. 文件系统级别备份

另外一种备份策略是直接复制PostgreSQL用于存储数据库中数据的文件，第 18.2 解释了这些文件的位置。你可以采用任何你喜欢的方式进行文件系统备份，例如：

```
tar -cf backup.tar /usr/local/pgsql/data
```

但是这种方法有两个限制，使得这种方法不实用，或者说至少比pg_dump方法差：

1. 为了得到一个可用的备份，数据库服务器必须被关闭。例如阻止所有连接的半路措施是不起作用的（部分原因是tar和类似工具无法得到文件系统状态的一个原子的快照，还有服务器内部缓冲的原因）。关于停止服务器的信息可以在第 18.5 节找到。不用说，在恢复数据之前你也需要关闭服务器。

2. 如果你已经深入地了解了数据库的文件系统布局的细节，你可能会有兴趣尝试通过相应的文件或目录来备份或恢复特定的表或数据库。这种方法也不会起作用，因为包含在这些文件中的信息只有配合提交日志文件（pg_xact/*）才有用，提交日志文件包含了所有事务的提交状态。一个表文件只有和这些信息一起才有用。当然也不可能只恢复一个表及相关的pg_xact数据，因为这会导致数据库集簇中所有其他表变得无用。因此文件系统备份值适合于完整地备份或恢复整个数据库集簇。

另一种文件系统备份方法是创建一个数据目录的“一致快照”，如果文件系统支持此功能（并且你相信它的实现正确）。典型的过程是创建一个包含数据库的卷的“冻结快照”，然后从该快照复制整个数据目录（如上，不能是部分复制）到备份设备，最后释放冻结快照。即使在数据库服务器运行时，这种方式也有效。但是，以这种方式创建的备份保存的文件看起来就像数据库没有被正确关闭时的状态。因此，当你从备份数据上启动数据库服务器时，它会认为上一次的服务器实例崩溃了并尝试重放WAL日志。这不是问题，只是需要注意（当然WAL文件必须要包括在备份中）。你可以在拍摄快照之前执行一次CHECKPOINT以便节省恢复时间。

如果你的数据库跨越多个文件系统，可能没有任何方式可以对所有卷获得完全同步的冻结快照。例如，如果你的数据文件和WAL日志放置在不同的磁盘上，或者表空间在不同的文件系统中，可能没有办法使用快照备份，因为快照必须是同步的。在这些情况下，一定要仔细阅读你的文件系统文档以了解其对一致快照技术的支持。

如果没有可能获得同步快照，一种选择是将数据库服务器关闭足够长的时间以建立所有的冻结快照。另一种选择是执行一次连续归档基础备份（第 25.3.2 节，因为这种备份对于备份期间发生的文件系统改变是免疫的。这要求在备份过程中允许连续归档，恢复时使用连续归档恢复（第 25.3.4 节。

还有一种选择是使用rsync来执行一次文件系统备份。其做法是先在数据库服务器运行时执行rsync，然后关闭数据库服务器足够长时间来做一次rsync --checksum （--checksum是必需的，因为rsync的文件修改 时间粒度只能精确到秒）。第二次rsync会比第一次快，因为它只需要传送相对很少的数据，由于服务器是停止的，所以最终结果将是一致的。这种方法允许在最小停机时间内执行一次文件系统备份。

注意一个文件系统备份通常会比一个SQL转储体积更大（例如pg_dump不需要转储索引的内容，而是转储用于重建索引的命令）。但是，做一次文件系统备份可能更快。

25.3. 连续归档和时间点恢复（PITR）

在任何时间，PostgreSQL在数据集簇目录的pg_wal/子目录下都保持有一个预写式日志（WAL）。这个日志存在的目的是为了保证崩溃后的安全：如果系统崩溃，可以“重放”从最后一次检查点以来的日志项来恢复数据库的一致性。该日志的存在也使得第三种备

份数据库的策略变得可能：我们可以把一个文件系统级别的备份和WAL文件的备份结合起来。当需要恢复时，我们先恢复文件系统备份，然后从备份的WAL文件中重放来把系统带到一个当前状态。这种方法比之前的方法管理起来要更复杂，但是有其显著的优点：

- 我们不需要一个完美的一致的文件系统备份作为开始点。备份中的任何内部不一致性将通过日志重放（这和崩溃恢复期间发生的并无显著不同）来修正。因此我们不需要文件系统快照功能，只需要tar或一个类似的归档工具。
- 由于我们可以结合一个无穷长的WAL文件序列用于重放，可以通过简单地归档WAL文件来达到连续备份。这对于大型数据库特别有用，因为在其中不方便频繁地进行完全备份。
- 并不需要一直重放WAL项一直到最后。我们可以在任何点停止重放，并得到一个数据库在当时的一致快照。这样，该技术支持时间点恢复：在得到你的基础备份以后，可以将数据库恢复到它在其后任何时间的状态。
- 如果我们连续地将一系列WAL文件输送给另一台已经载入了相同基础备份文件的机器，我们就得到了一个热后备系统：在任何时间点我们都能提出第二台机器，它差不多是数据库的当前副本。

> **注意**
>
> pg_dump和pg_dumpall不会产生文件系统级别的备份，并且不能用于连续归档方案。这类转储是逻辑的并且不包含足够的信息用于WAL重放。

就简单的文件系统备份技术来说，这种方法只能支持整个数据库集簇的恢复，却无法支持其中一个子集的恢复。另外，它需要大量的归档存储：一个基础备份的体积可能很庞大，并且一个繁忙的系统将会产生大量需要被归档的WAL流量。尽管如此，在很多需要高可靠性的情况下，它是首选的备份技术。

要使用连续归档（也被很多数据库厂商称为“在线备份”）成功地恢复，你需要一个从基础备份时间开始的连续的归档WAL文件序列。为了开始，在你建立第一个基础备份之前，你应该建立并测试用于归档WAL文件的过程。对应地，我们首先讨论归档WAL文件的机制。

25.3.1. 建立WAL归档

抽象地来说，一个运行中的PostgreSQL系统产生一个无穷长的WAL记录序列。系统从物理上将这个序列划分成WAL 段文件，通常是每个16MB（段尺寸在initdb期间可修改）。段文件会被分配一个数字名称以便反映它在整个抽象WAL序列中的位置。在没有使用WAL归档时，系统通常只创建少量段文件，并且通过重命名不再使用的段文件为更高的段编号来“回收”它们。系统假设内容位于最后一个检查点之前的段文件是无用的且可以被回收。

在归档WAL数据时，我们需要在每一段被填充满时捕捉其内容，并且在段文件被回收重用之前保存该数据。依靠应用和可用的硬件，有很多不同的方法来“保存数据”：我们可以将段文件拷贝到一个已挂载的位于另一台机器上的NFS目录，或者将它们写出到一个磁带驱动器（确保你有办法标识每个文件的原始文件名），或者将它们批量烧录到CD上，或者其他什么方法。为了向数据库管理员提供灵活性，PostgreSQL不对如何归档做任何假设。取而代之的是，PostgreSQL让管理员声明一个shell命令来拷贝一个完整的段文件到它需要去的地方。该命令可以简单得就是一个cp，或者它可以调用一个复杂的 shell 脚本 — 所有都由你决定。

要启用WAL归档，需设置wal_level配置参数为replica或更高，设置archive_mode为on，并且使用archive_command配置参数指定一个shell命令。实际上，这些设置总是被放置在postgresql.conf文件中。在archive_command中，%p会被将要归档的文件路径所替代，而%f只会被文件名所替代（路径名是相对于当前工作目录而言的，即集簇的数据目录）。如果你需要在命令中嵌入一个真正的%字符，可以使用%%。最简单的命令类似于：

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/
archivedir/%f'  # Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

它将把 WAL 段拷贝到目录/mnt/server/archivedir（这个只是一个例子，并非我们建议的方法，可能不能在所有系统上都正确运行）。在%p和%f参数被替换之后，实际被执行的命令看起来可能是：

```
test ! -f /mnt/server/archivedir/000000010000000A900000065 && cp
 pg_wal/000000010000000A900000065 /mnt/server/archivedir/000000010000000A900000065
```

对每一个将要被归档的新文件都会生成一个类似的命令。

归档命令将在运行PostgreSQL服务器的同一个用户的权限下执行。因为被归档的一系列WAL文件实际上包含你的数据库里的所有东西，所以你应该确保自己的归档数据不会被别人窥探； 比如，归档到一个没有组或者全局读权限的目录里。

有一点很重要：当且仅当归档命令成功时，它才返回零退出。在得到一个零值结果之后，PostgreSQL将假设该文件已经成功归档， 因此它稍后将被删除或者被新的数据覆盖。但是，一个非零值告诉PostgreSQL该文件没有被归档； 因此它会周期性的重试直到成功。

归档命令通常应该被设计成拒绝覆盖已经存在的归档文件。这是一个非常重要的安全特性，可以在管理员操作失误（比如把两个不同的服务器的输出发送到同一个归档目录）的时候保持你的归档的完整性。

我们建议你首先要测试你准备使用到归档命令，以保证它实际上不会覆盖现有的文件，并且在这种情况下它返回非零状态。以上Unix中的命令例子通过包含一个独立的test步骤来保证这一点。在某些Unix平台上，cp具有诸如-i的开关，可用来更简洁地完成这一切，但是在没有验证返回的退出状态正确之前你不能依赖它们（特别地，GNU的cp在使用-i时将对已存在的目标文件返回状态零，这并不是我们所期望的行为）。

在设计你的归档环境时，请考虑一下如果归档命令不停失败会发生什么情况， 因为有些情况要求操作者的干涉，或者是归档空间不够了。 例如，如果你往磁带机上写，但是没有自动换带机，那么就有可能发生这种情况； 如果磁带满了，除非换磁带，否则任何事也做不了。 你应该确保任何错误情况或者任何要求操作员干涉的情况都会被正确报告， 这样才能迅速解决这些问题。否则pg_wal/目录会不停地被WAL段文件填充，直到问题解决（如果包含pg_wal/的文件系统被填满，PostgreSQL将会做一次致命关闭。不会有未提交事务被丢失，但是数据库将会保持离线直到你释放一部分空间）。

归档命令的速度并不要紧，只要它能跟上你的服务器生成 WAL 数据的平均速度即可。即使归档进程稍微落后，正常的操作也会继续进行。 如果归档进程慢很多，就会增加灾难发生的时候丢失的数据量。这同时也意味着pg_wal/目录包含大量未归档的段文件， 并且可能最后超出了可用磁盘空间。我们建议你监控归档进程，确保它是按照你的期望运转的。

在写自己的归档命令的时候，你应该假设被归档的文件名最长为64个字符并且可以包含 ASCII 字母、数字以及点的任意组合。 我们不需要保持原始的相对路径（%p），但是有必要保持文件名（%f）。

请注意尽管 WAL 归档允许你恢复任何对你的PostgreSQL数据库中数据所做的修改， 但它不会恢复对配置文件的修改（即postgresql.conf、pg_hba.conf以及pg_ident.conf），因为这些文件都是手工编辑的，而不是通过 SQL 操作来编辑的。 所以你可能会需要把你的配置文件放在一个日常文件系统备份过程可处理的位置。如何重定位配置文件请参阅第 19.2 节

归档命令只会为完成的WAL段调用。因此如果你的服务器产生了一点点WAL流量（或者在产生时有宽松的周期），从一个事务完成到它被安全地记录在归档存储中之间将会有较长的延迟。要为未归档数据设置一个年龄限制，你可以设置archive_timeout来强制要求服务器按照其设定的频度切换到一个新的WAL段。注意由于强制切换而被归档的文件还是具有和完全归档的文件相同的长度。因此设置一个很短的archive_timeout是很不明智的 — 它会膨胀你的归档存储。将archive_timeout设置为1分钟左右通常是合理的。

同样，如果你希望确保一个刚刚完成的事务能被尽快归档，可以使用pg_switch_wal进行一次手动段切换。其他与WAL管理相关的使用函数在表 9.79中列出。

如第 14.4.7 节所述，当wal_level为minimal时，一些SQL命令被优化为避免记录WAL日志。在这些语句的其中之一的执行过程中如果打开了归档或流复制，WAL中将不会包含足够的信息用于归档恢（崩溃恢复不受影响）。出于这个原因，wal_level只能在服务器启动时修改。但是，archive_command可以通过重载配置文件来修改。如果你希望暂时停止归档，一种方式是将archive_command设置为空串（''）。这将导致WAL文件积累在pg_wal/中，直到一个可用的archive_command被重新建立。

25.3.2. 制作一个基础备份

执行一次基础备份最简单的方法是使用pg_basebackup工具。它将会以普通文件或一个tar归档的方式创建一个基础备份。如果需要比pg_basebackup更高的灵活性，你也可以使用低级API（见第 25.3.3 节来制作一个基础备份。

没有必要关心创建一个基础备份所需的时间。但是，如果你正常地运行停用了full_page_writes的服务器，你可能会注意到备份运行时的性能下降，因为full_page_writes在备份模式期间会被实际强制实施。

要使用备份，你将需要保留所有在文件系统备份期间及之后生成的WAL段文件。为了便于你做这些，基础备份过程会创建一个备份历史文件，它将被立刻存储到WAL归档区域。该文件以文件系统备份中你需要的第一个WAL段文件命名。例如，如果开始的WAL文件是000000010000123400000055CD，则备份历史文件将被命名为000000010000123400000055CD.007C9330.backup.（文件名的第二部分表明WAL文件中的一个准确位置，一般可以被忽略）。一旦你已经安全地归档了文件系统备份和在备份过程中被使用的WAL段文件（如备份历史文件中所指定的） ，所有名字在数字上低于备份历史文件中记录值的已归档WAL段对于恢复文件系统备份就不再需要了，并且可以被删除。但是你应该考虑保持多个备份集以绝对保证你能恢复你的数据。

备份历史文件是一个很小的文本文件。它包含你指定给pg_basebackup的标签字符串，以及备份的起止时间以及起止WAL段。如果你使用该标签来标识相关转储文件，则已归档的历史文件足以说明需要哪个转储文件进行恢复。

由于你不得不保存最后一次基础备份之后的所有归档WAL文件，基础备份之间的间隔通常应该根据你希望在归档WAL文件上花费的存储空间来设定。你也应该考虑你准备花多长时间来进行恢复，如果需要恢复 — 系统将不得不重放所有那些WAL段，如果这些WAL段覆盖了最后一次基础备份以后的很长时间，重放过程将会花费一些时间。

25.3.3. 使用低级API制作一个基础备份

使用低级API制作一个基础备份的过程比pg_basebackup方法要包含更多的步骤，但相对要更简单。很重要的一点是，这些步骤要按照顺序执行，并且在执行下一步之前要验证上一步是否成功。

可以用非排他或者排他的方法来制作低级基础备份。我们推荐非排他方法，而排他 的方法已经被废弃并且最终将被去除。

25.3.3.1. 制作一个非排他低级备份

非排他低级备份允许其他并发备份运行（既包括那些使用同样的 备份 API 开始的备份，也包括那些使用 pg_basebackup开始的备份）。

1. 确保WAL归档被启用且正在工作。

2. 作为一个具有运行 pg_start_backup 权利的用户（超级用户，或者被授予在该 函数上 EXECUTE 的用户）连接到服务器（不在乎是哪个数据库）并且发出命令：

```
SELECT pg_start_backup('label', false, false);
```

其中label是用来唯一标识这次备份操作的任意字符串。调用 pg_start_backup的连接必须被保持到备份结束，否则备份 将被自动中止。

默认情况下，pg_start_backup可能需要较长的时间完成。 这是因为它会执行一个检查点，并且该检查点所需要的 I/O 将会分散到一段 显著的时间上，默认情况下是你的检查点间隔（见配置参数 checkpoint_completion_target）的一半。这通常 是你所想要的，因为它可以最小化对查询处理的影响。如果你想要尽可能快地 开始备份，请把第二个参数改成true，这将会发出一个立即的检查点并且使用尽可能多的I/O。

第三个参数为false会告诉pg_start_backup 开始一次非排他基础备份。

3. 使用任何趁手的文件系统备份工具（例如tar或者 cpio，不是pg_dump 或者pg_dumpall）执行备份。当你做这些 时，不需要也不值得停止正常的数据库操作。在这类备份期间要考虑的事情 请见小节第 25.3.3.3 节

4. 在同一个连接中，发出命令：

```
SELECT * FROM pg_stop_backup(false);
```

这会终止备份模式。在主控机上，它还执行一次自动切换到下一个WAL段。在后备机上，它无法自动切换WAL段，因此用户可能希望在主控机上运行pg_switch_wal来执行一次手工切换。要做切换的原因是让在备份期间写入的最后一个WAL段文件能准备好被归档。

pg_stop_backup将返回一个具有三个值的行。这些域的 第二个应该被写入到该备份根目录中名为backup_label的 文件。第三个域应该被写入到一个名为tablespace_map 的文件，除非该域为空。这些文件对该备份正常工作来说是至关重要的， 不能被随意修改。

5. 一旦备份期间活动的WAL段文件被归档，备份就完成了。由 pg_stop_backup的第一个返回值标识的文件是构成一个 完整备份文件集合所需的最后一个段。在主控机上，如果archive_mode被启用并且wait_for_archive参数为true，在最后一个段被归档之前pg_stop_backup都不会返回。在后备机上，为了让pg_stop_backup等待，archive_mode必须为always。从你已经配置好archive_command之后这些文件的 归档就会自动发生。在大部分情况下，这些归档会很快发生，但是建议你监 控你的归档系统确保没有延迟。如果归档进程由于归档命令的失败而落后， 它将会持续重试知道归档成功并且备份完成。如果你希望对 pg_stop_backup的执行给出一个时间限制，可以设置一个 合适的statement_timeout值，但要注意如果 pg_stop_backup因此而中止会致使备份可能失效。

   如果备份进程监控并且确保备份所需的所有WAL段文件都被成功地归档，那么wait_for_archive参数（默认为true）可以被设置为false，以便pg_stop_backup在停止备份记录被写入到WAL后立即返回。默认情况下，pg_stop_backup将会等待，直至所有WAL都被归档，这种等待会花一段时间。这个选项必须被小心地使用：如果WAL归档没有被正确的监控，则备份可能没有包括所有的WAL文件并且因此将变得不完整和不可恢复。

25.3.3.2. 制作一个排他低级备份

一个排他备份的处理绝大部分都和非排他备份相同，但是在一些关键步骤上 不同。这种备份只能在主控机上制作，并且不允许并发备份。在PostgreSQL 9.6 之前，这是唯一可用的低级方法，但是现在 推荐所有用户在可能的情况下升级他们的脚本来使用非排他备份。

1. 确保 WAL 归档被启用且正常工作。

2. 作为一个具有运行 pg_start_backup 权利的用户（超级用户，或者被授予在该 函数上 EXECUTE 的用户）连接到服务器（不在乎是哪个数据库）并且发出命令：

```
SELECT pg_start_backup('label');
```

这里label是任何你希望用来唯一标识这个备份操作的字符串。 pg_start_backup在集簇目录中创建一个关于备份信息的 备份标签文件，也被称为backup_label， 其中包括了开始时间和标签字符串。该函数也会在集簇目录中创建一个 名为tablespace_map的表空间映射文件， 如果在pg_tblspc/中有一个或者多个表空间符号链接存在， 该文件会包含它们的信息。如果你需要从备份中恢复，这两个文件对于备份的 完整性都至关重要。

默认情况下，pg_start_backup会花费很长时间来完成。这是因为它会执行一个检查点，而检查点所需要的I/O在相当一段时间内将会被传播，默认情况下这段时间是内部检查点间隔的一半（参见配置参数checkpoint_completion_target）。这通常是你所希望的，因为它能将对查询处理的影响最小化。如果你要尽快开始备份，可使用：

```
SELECT pg_start_backup('label', true);
```

这会使检查点尽可能快地被完成。

3. 使用任何方便的文件系统备份工具执行备份，例如tar 或cpio（不是pg_dump 或pg_dumpall）。在此期间，不需要也 不值得停止正常的数据库操作。在备份期间要考虑的事情可见 第 25.3.3.3 节。

   注意，如果服务器在备份期间崩溃，它可能无法重启，直至从PGDATA目录中手工地移除backup_label文件。

4. 再次以具有运行 pg_stop_backup 权利的用户（超级用户，或者已经被授予 该函数上 EXECUTE 的用户）连接到数据库并且发出命令：

   ```
   SELECT pg_stop_backup();
   ```

   这个函数将终止备份模式，并且执行一个自动切换到下一个WAL段。进行切换的原因是将在备份期间生成的最新WAL段文件安排为可归档。

5. 一旦备份期间活动的WAL段文件被归档，你的工作就完成了。 pg_stop_backup的结果所标识的文件是构成一个完整备份 文件组所需的最新段。如果archive_mode被启用，直到最新段被归档pg_stop_backup都不会返回。由于你已经配置了 archive_command，这些文件的归档过程会自动发生。在 大部分情况下这会很快发生，但还是建议你监控你的归档系统来确保不会有 延迟。如果归档处理由于归档命令的错误而延迟，它会保持重试直到归档成 功和备份完成。如果你希望在pg_stop_backup的执行上 设置一个时间限制，可对statement_timeout设 置一个合适的值，但要注意如果pg_stop_backup因此而 中止会致使备份可能失效。

25.3.3.3. 备份数据目录

如果被拷贝的文件在拷贝过程中发生变化，某些文件系统备份工具会发出警告或错误。在建立一个活动数据库的基础备份时，这种情况是正常的，并非一个错误。然而，你需要确保你能够把它们和真正的错误区分开。例如，某些版本的rsync为“消失的源文件”返回一个独立的退出码，且你可以编写一个驱动脚本来将该退出码接受为一种非错误情况。同样，如果一个文件在被tar复制的过程中被截断，某些版本的GNU tar会返回一个与致命错误无法区分的错误代码。幸运的是，如果一个文件在备份期间被改变，版本为1.16及其后的GNU tar将会退出并返回1，而对于其他错误返回2。在版本1.23及其后的GNU tar中，你可以使用警告选项--warning=no-file-changed --warning=no-file-removed来隐藏相关的警告消息。

确认你的备份包含数据库集簇目录（例如/usr/local/pgsql/data）下的所有文件。如果你使用了不在此目录下的表空间，注意也把它们包括在内（并且确保你的备份将符号链接归档为链接，否则恢复过程将破坏你的表空间）。

不过，你应当从备份中忽略集簇的pg_wal/子目录中的文件。这种微小的调整是值得的，因为它降低了恢复时的错误风险。如果pg_wal/是一个指向位于集簇目录之外其他地方的符号链接就很容易安排了，这是一种出于性能原因的常见设置。你可能也希望排

除postmaster.pid和postmaster.opts，它们记录了关于postmaster运行的信息，但与最终使用这个备份的postmaster无关（这些文件可能会使pg_ctl搞混淆）。

从备份中忽略集簇的pg_replslot/子目录中的文件通常也是个好主意，这样 主控机上存在的复制槽不会成为备份的一部分。否则，后续用该备份创建一个后备机可能会导致该 后备机上的WAL文件被无限期保留，并且在启用了热后备反馈的情况下可能导致主控机膨胀，因为使用 那些复制槽的客户端将继续连接到主控机（而不是后备机）并且继续更新其上的槽。即使该备份是要被 用来创建一个新的主控机，拷贝复制槽也不是特别有用，因为这些槽的内容在新主控机上线时很可能已 经过时。

目录pg_dynshmem/、pg_notify/、pg_serial/、pg_snapshots/、pg_stat_tmp/和pg_subtrans/的内容（但不是这些目录本身）可以从备份中省略，因为它们在postmaster启动时会被初始化。如果stats_temp_directory被设置并且位于数据目录中，则该目录的内容也可以被省略。

任何以pgsql_tmp开始的文件或目录都可以从备份中省略。这些文件在postmaster启动时会被移除，而目录将被根据需要重建。

只要找到名为pg_internal.init的文件，它就可以从备份中省略。这些文件包含关系缓冲数据，它们在恢复时总是会被重建。

备份标签文件包含你指定给 pg_start_backup的标签字符串，以及 pg_start_backup被运行的时刻和起始WAL文件的名字。 在发生混乱的情况下就可以在备份文件中查看并准确地决定该转储文件来 自于哪个备份会话。表空间映射文件包括存在于目录pg_tblspc/ 中的符号链接名称以及每一个符号链接的完整路径。这些文件不仅是为了供参考， 它们的存在和内容对于系统恢复过程的正确操作是至关重要。

在服务器停止时也可以创建一个备份。在这种情况下，你显然不能使用pg_start_backup或pg_stop_backup， 并且因此你只能依靠你的自己的策略来跟踪哪个备份是哪个， 以及相关WAL文件应该走回到什么程度。通常最好遵循上面的连续归档过程。

25.3.4. 使用一个连续归档备份进行恢复

好，现在最坏的情况发生了，你需要从你的备份进行恢复。这里是其过程：

1. 如果服务器仍在运行，停止它。

2. 如果你具有足够的空间，将整个集簇数据目录和表空间复制到一个临时位置，稍后你将用到它们。注意这种预防措施将要求在你的系统上有足够的空闲空间来保留现有数据库的两个拷贝。如果你没有足够的空间，你至少要保存集簇的pg_wal子目录的内容，因为它可能包含在系统垮掉之前还未被归档的日志。

3. 移除所有位于集簇数据目录和正在使用的表空间根目录下的文件和子目录。

4. 从你的文件系统备份中恢复数据库文件。注意它们要使用正确的所有权恢复（数据库系统用户，不是root！）并且使用正确的权限。如果你在使用表空间，你应该验证pg_tblspc/中的符号链接被正确地恢复。

5. 移除pg_wal/中的任何文件，这些是来自于文件系统备份而不是当前日志，因此可以被忽略。如果你根本没有归档pg_wal/，那么以正确的权限重建它。注意如果以前它是一个符号链接，请确保你也以同样的方式重建它。

6. 如果你有在第2步中保存的未归档WAL段文件，把它们拷贝到pg_wal/（最好是拷贝而不是移动它们，这样如果在开始恢复后出现问题你任然有未修改的文件）。

7. 在集簇数据目录中创建一个恢复命令文件recovery.conf（见第 27 章。你可能还想临时修改pg_hba.conf来阻止普通用户在成功恢复之前连接。

8. 启动服务器。服务器将会进入到恢复模式并且进而根据需要读取归档WAL文件。恢复可能因为一个外部错误而被终止，可以简单地重新启动服务器，这样它将继续恢复。恢复过程

结束后，服务器将把recovery.conf重命名为recovery.done（为了阻止以后意外地重新进入恢复模式），并且开始正常数据库操作。

9. 检查数据库的内容来确保你已经恢复到了期望的状态。如果没有，返回到第1步。如果一切正常，通过恢复pg_hba.conf为正常来允许用户连接。

所有这些的关键部分是设置一个恢复配置文件，它描述你希望如何恢复以及恢复要运行到什么程度。你可以使用recovery.conf.sample（通常在安装的share/目录中）作为一个原型。你绝对必须在recovery.conf中指定的是restore_command，它告诉PostgreSQL如何获取归档WAL文件段。与archive_command相似，这也是一个shell命令字符串。它可以包含%f（将被期望的日志文件名替换）和%p（将被日志文件被拷贝的目标路径名替换）。（路径名是相对于当前工作目录的，即集簇的数据目录）。如果你需要在命令中嵌入一个真正的%字符，可以写成%%。最简单的命令类似于：

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

它将从目录/mnt/server/archivedir中拷贝之前归档的WAL段。当然，你可以使用更复杂的，甚至是一个要求操作者装载合适磁带的shell脚本。

重要的是命令在失败时返回非零退出状态。该命令将被调用来请求不在归档中的文件， 在这种情况下它应该返回非零值。这不是一种错误情况。一种例外是该命令被一个信号（除了被用作数 据库服务器关闭动作一部分的SIGTERM）终止或者被shell的错误 （例如命令未找到）终止，那样恢复将中止并且服务器将不会启动。

并非所有被请求的文件都是WAL段文件，你也许还会请求一些具有.history后缀的文件。还要注意的是，%p路径的基本名字将会和%f 不同，但不要期望它们可以互换。

归档中找不到的WAL段可以在pg_wal/中看到，这使得可以使用最近未归档的段。但是，在归档中可用的段将会被优先于pg_wal/中的文件被使用。

通常，恢复将会处理完所有可用的WAL段，从而将数据库恢复到当前时间点（或者尽可能接近给定的可 用WAL段）。因此，一个正常的恢复将会以一个“文件未找到”消息结束，错误消息的准确文 本取决于你选择的restore_command。你也可能在恢复的开始看到一个针对名称类 似于00000001.history文件的错误消息。这也是正常的并且不表示在简单恢复情 况中的问题，对此的讨论见第 25.3.5 节

如果你希望恢复到之前的某个时间点（例如，恢复到幼稚的DBA丢弃了你主要的交易表之前），只需要 在recovery.conf中指定要求的停止点。你可以使用日期/时间、命名恢复点或一个 指定事务ID的结束时间来定义停止点（也被称为“恢复目标”）。在这种写法中，只有日期/时 间和命名恢复点选项非常有用，因为没有工具可以帮助你准确地确定要用哪个事务ID。

> **注意**
>
> 停止点必须位于基础备份的完成时间之后，即pg_stop_backup的完成时间。在备份过程中你不能使用基础备份来恢复（要恢复到这个时间，你必须回到你之前的基础备份并且从这里开始前滚）。

如果恢复找到被破坏的WAL数据，恢复将会停止于该点并且服务器不会启动。在这种情况下，恢复进程需要从开头重新开始运行，并指定一个在损坏点之前的“恢复目标”以便恢复能够正常完成。如果恢复由于一个外部原因失败，例如一个系统崩溃或者WAL归档变为不可访问，则该次恢复可以被简单地重启并且它将会从几乎是上次失败的地方继续。恢复重启工作起来很像普通操作时的检查点：服务器周期性地强制把它的所有状态写到磁盘中，然后更新pg_control文件来说明已经处理过的WAL数据，这样它们就不会被再次扫描。

25.3.5. 时间线

将数据库恢复到一个之前的时间点的能力带来了一些复杂性，这和有关时间旅行和平行宇宙的科幻小说有些相似。例如，在数据库的最初历史中，假设你在周二晚上5：15时丢弃了一个关键表，但是一直到周三中午才意识到你的错误。不用苦恼，你取出你的备份，恢复到周二晚上5：14的时间点，并上线运行。在数据库宇宙的这个历史中，你从没有丢弃该表。但是假设你后来意识到这并非一个好主意，并且想回到最初历史中周三早上的某个时间。你没法这样做，在你的数据库在线运行期间，它重写了某些WAL段文件，而这些文件本来可以将你引向你希望回到的时间。因此，为了避免出现这种状况，你需要将完成时间点恢复后生成的WAL记录序列与初始数据库历史中产生的WAL记录序列区分开来。

要解决这个问题，PostgreSQL有一个时间线概念。无论何时当一次归档恢复完成，一个新的时间线被创建来标识恢复之后生成的WAL记录序列。时间线ID号是WAL段文件名的一部分，因此一个新的时间线不会重写由之前的时间线生成的WAL数据。实际上可以归档很多不同的时间线。虽然这可能看起来是一个无用的特性，但是它常常扮演救命稻草的角色。考虑到你不太确定需要恢复到哪个时间点的情况，你可能不得不做多次时间点恢复尝试和错误，直到最终找到从旧历史中分支出去的最佳位置。如果没有时间线，该处理将会很快生成一堆不可管理的混乱。而有了时间线，你可以恢复到任何之前的状态，包括早先被你放弃的时间线分支中的状态。

每次当一个新的时间线被创建，PostgreSQL会创建一个“时间线历史”文件，它显示了新时间线是什么时候从哪个时间线分支出来的。系统在从一个包含多个时间线的归档中恢复时，这些历史文件对于允许系统选取正确的WAL段文件非常必要。因此，和WAL段文件相似，它们也要被归档到WAL归档区域。历史文件是很小的文本文件，因此将它们无限期地保存起来的代价很小，而且也是很合适的（而段文件都很大）。如果你喜欢，你可以在一个历史文件中增加注释来记录如何和为什么要创建该时间线。当你由于试验的结果拥有了一大堆错综复杂的不同时间线时，这种注释将会特别有价值。

恢复的默认行为是沿着相同的时间线进行恢复，该时间线是基础备份创建时的当前时间线。如果你希望恢复到某个子女时间线（即，你希望回到在一次恢复尝试后产生的某个状态），你需要在recovery.conf中指定目标时间线ID。你不能恢复到早于该基础备份之前分支出去的时间线。

25.3.6. 建议和例子

这里将给出一些配置连续归档的建议。

25.3.6.1. 单机热备份

可以使用PostgreSQL的备份功能来产生单机热备份。这些备份不能被用于时间点恢复，然而备份和恢复时要比使用pg_dump转储更快（它们也比pg_dump转储更大，所以在某些情况下速度优势可能会被否定）。

在基础备份的帮助下，产生一个单机热备份最简单的方式是使用pg_basebackup工具。如果你在调用它时使用了-X参数，使用该备份所需的所有事务日志将会被自动包含在该备份中，并且恢复该备份也不需要特殊的动作。

如果在复制备份文件时需要更多灵活性，也可以使用一个较低层的处理来创建单机热备份。要为低层 单机热备份做准备，确保wal_level被设置为replica或更高， archive_mode设置为on，并且设置一个archive_command，该命令只当一个开关文件存在时执行归档。例如：

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /
var/lib/pgsql/archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

该命令在/var/lib/pgsql/backup_in_progress存在时执行归档，否则会安静地返回0值退出状态（让PostgreSQL能回收不需要的WAL文件）。

通过这样的准备，可以使用一个如下所示的脚本来建立备份：

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

开关文件/var/lib/pgsql/backup_in_progress首先被创建，这使对于未完成WAL文件的归档操作发生。备份完成之后开关文件会被删除。归档的WAL文件则被加入到备份中，这样基础备份和所有需要的WAL文件都是同一个tar文件的组成部分。请记住在你的备份脚本中加入错误处理。

25.3.6.2. 压缩的归档日志

如果担心归档存储的尺寸，你可以使用gzip来压缩归档文件：

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

那么在恢复时你将需要使用gunzip：

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

25.3.6.3. archive_command脚本

很多人选择使用脚本来定义他们的archive_command，这样他们的postgresql.conf项看起来非常简单：

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

任何时候如果你希望在归档处理中使用多个命令，明智的方法是使用一个独立的脚本文件。这可以使脚本更为复杂，它可以使用一种流行的脚本语言来编写，例如bash或perl。

需要在一个脚本内解决的需求例子包括：

- 将数据拷贝到安全的场外数据存储
- 批处理WAL文件，这样它们可以每三小时被传输一次，而不是一次一个
- 与其他备份和恢复软件交互
- 与监控软件交互以报告错误

> **提示**
>
> 在使用一个archive_command脚本时，最好启用logging_collector。任何从该脚本被写到stderr的消息将出现在数据库服务器日志中，这允许在复杂配置失败后能更容易被诊断。

25.3.7. 警告

在编写此文档时，连续归档技术存在一些限制。这可能会在未来的发布中被修复：

- 如果一个CREATE DATABASE命令在基础备份时被执行，然后在基础备份进行时CREATE DATABASE所复制的模板数据库被修改，恢复中可能会导致这些修改也被传播到已创建的数

据库中。这当然是我们不希望的。为了避免这种风险，最好不要在创建基础备份时修改任何模板数据库。

- CREATE TABLESPACE命令会WAL以其字面绝对路径记录，并且因此将在重放时以相同的绝对路径来创建表空间。当日志在一台不同的机器上被重放时，这可能也不是我们希望的。即使日志在同一台机器上被重放也是危险的，就算是恢复到一个新的数据目录重放过程也会覆盖原来表空间的内容。为了避免这种潜在的陷阱，最佳做法是在创建或丢弃表空间后创建一个新的基础备份。

还需要注意的是，默认的WAL格式相当庞大，因为它包括了很多磁盘页快照。这些页快照被设计用于支持崩溃恢复，因为我们可能需要修复断裂的磁盘页。依靠你的系统硬件和软件，页断裂的风险可能会小到可以忽略，在此种情况下你可以通过使用full_page_writes参数关闭页快照来显著降低归档日志的总容量（在这样做之前阅读第 30 章的注解和警告）。关闭页快照并不会阻止使用日志进行PITR操作。一个未来的开发点是通过移除不需要的页拷贝来压缩归档的WAL数据，即使full_page_writes为on。同时，管理员可能希望通过尽可能增大检查点间隔参数来减少WAL中包含的页快照数量。

第 26 章 高可用、负载均衡和复制

数据库服务器可以一起工作，这样如果主要的服务器失效则允许一个第二服务器快速接手它的任务（高可用性），或者可以允许多个计算机提供相同的数据（负载均衡）。理想情况下，数据库服务器能够无缝地一起工作。提供静态网页服务的网页服务器可以非常容易地通过把网页请求均衡到多个机器来组合。事实上，只读的数据库服务器也可以相对容易地组合起来。不幸的是，大部分数据库服务器收到的请求是读/写混合的，并且读/写服务器更难于组合。这是因为尽管只读数据只需要在每台服务器上放置一次，但对于任意服务器的一次写动作却必须被传播给所有的服务器，这样才能保证未来对于那些服务器的读请求能返回一致的结果。

这种同步问题是服务器一起工作的最根本的困难。因为没有单一解决方案能够消除该同步问题对所有用例的影响。有多种解决方案，每一种方案都以一种不同的方式提出了这个问题，并且对于一种特定的负载最小化了该问题所产生的影响。

某些方案通过只允许一台服务器修改数据来处理同步。能修改数据的服务器被称为读/写、主控或主要服务器。跟踪主控机中改变的服务器被称为后备或次级服务器。如果一台后备服务器只有被提升为一台主控服务器后才能被连接，它被称为一台温后备服务器，而一台总是能够接受连接并且提供只读查询的后备服务器被称为一台热后备服务器。

某些方案是同步的，即一个数据修改事务只有到所有服务器都提交了该事务之后才被认为是提交成功。这保证了一次故障转移不会丢失任何数据并且所有负载均衡的服务器将返回一致的结果（不管哪台服务器被查询）。相反，异步的方案允许在一次提交和它被传播到其他服务器之间有一些延迟，这产生了切换到一个备份服务器时丢失某些事务的可能性，并且负载均衡的服务器可能会返回略微陈旧的结果。当同步通信可能很慢时，可以使用异步通信。

方案也可以按照它们的粒度进行分类。某些方案只能处理一整个数据库服务器，而其他的允许在每个表或每个数据库的级别上进行控制。

在任何选择中，都必须考虑性能。通常在功能和性能之间都存在着权衡。例如，在一个低速网络上的一种完全同步的方案可能使性能减少超过一半，而一种异步的方案产生的性能影响可能是最小的。

本节的剩余部分勾勒了多种故障转移、复制和负载均衡方案。

26.1. 不同方案的比较

共享磁盘故障转移

> 共享磁盘故障转移避免了只使用一份数据库拷贝带来的同步开销。它使用一个由多个服务器共享的单一磁盘阵列。如果主数据库服务器失效，后备服务器则可以挂载并启动数据库，就好像它从一次数据库崩溃中恢复过来了。这是一种快速的故障转移，并且不存在数据丢失。
>
> 共享硬件功能在网络存储设备中很常见。也可以使用一个网络文件系统，但是要注意的是该文件系统应具有完全的POSIX行为（见第 18.2.2 节。这种方法的一个重大限制是如果共享磁盘阵列失效或损坏，主要和后备服务器都会变得无法工作。另一个问题是在主要服务器运行时，后备服务器永远不能访问共享存储。

文件系统（块设备）复制

> 共享硬件功能的一种修改版本是文件系统复制，在其中对一个文件系统的所有改变会被镜像到位于另一台计算机上的一个文件系统。唯一的限制是该镜像过程必须能保证后备服务器有一份该文件系统的一致的拷贝 — 特别是对后备服务器的写入必须按照主控机上相同的顺序进行。DRBD是用于 Linux 的一种流行的文件系统复制方案。

预写式日志传送

温备和热备服务器能够通过读取一个预写式日志（WAL）记录的流来保持为当前状态。如果主服务器失效，后备服务器拥有主服务器的几乎所有数据，并且能够快速地被变成新的主数据库服务器。这可以是同步的或异步的，并且只能用于整个数据库服务器。

可以使用基于文件的日志传送（第 26.2 节、流复制（见第 26.2.5 节或两者的组合来实现一个后备服务器。关于热备的信息可见第 26.5 节

逻辑复制

逻辑复制允许数据库服务器发送数据修改流给另一台服务器。PostgreSQL的逻辑复制从WAL中构建出一个逻辑数据修改流。逻辑复制允许复制个体表中的数据更改。逻辑复制不要求特定的服务器作为主服务器或者复制机，而是允许数据以多方向流动。更多有关逻辑复制的信息，请参考第 31 章通过逻辑解码接口（第 49 章，第三方扩展也能提供类似的功能。

基于触发器的主-备复制

一个主-备复制设置会把所有数据修改查询发送到主服务器。主服务器异步地将数据修改发送给后备服务器。当主服务器正在运行时，后备服务器可以回答只读查询。后备服务器对数据仓库查询是一种理想的选择。

Slony-I是这种复制类型的一个例子。它使用表粒度，并且支持多个后备服务器。因为它会异步更新后备服务器（批量），在故障转移时可能会有数据丢失。

基于语句的复制中间件

通过基于语句的复制中间件，一个程序拦截每一个 SQL 查询并把它发送给一个或所有服务器。每一个服务器独立地操作。读写查询必须被发送给所有服务器，这样每一个服务器都能接收到任何修改。但只读查询可以被只发送给一个服务器，这样允许读负载在服务器之间分布。

如果查询被简单地且未经修改地广播，random()、CURRENT_TIMESTAMP之类的函数以及序列在不同服务器上可能有不同的值。这是因为每一个服务器会独立地操作，并且 SQL 查询被广播（而不是真正被修改的行）。如果这不可接受，中间件或应用必须从一个单一服务器查询这样的值并且然后将那些值用在写查询中。另一个选项是将这个复制选项和一种传统主-备设置一起使用，即数据修改查询只被发送给主服务器并且通过主-备复制传播到后备服务器，而不是通过复制中间件。必须要注意的是，所有事务要么在所有服务器上都提交，要么在所有服务器上都中止，也许可以使用两阶段提交（PREPARE TRANSACTION和COMMIT PREPARED）。Pgpool-II和Continuent Tungsten是这种复制类型的例子。

异步多主控机复制

对于不会被定期连接的服务器（如笔记本或远程服务器），保持服务器间的数据一致是一个挑战。通过使用异步的多主控机复制，每一个服务器独立工作并且定期与其他服务器通信来确定冲突的事务。这些冲突可以由用户或冲突解决规则来解决。Bucardo 是这种复制类型的一个例子。

同步多主控机复制

在同步多主控机复制中，每一个服务器能够接受写请求，并且在每一个事务提交之前，被修改的数据会被从原始服务器传送给每一个其他服务器。繁重的写活动可能导致过多的锁定，进而导致很差的性能。事实上，写性能通常比一个单一服务器还要糟。读请求可以被发送给任意服务器。某些实现使用共享磁盘来减少通信负荷。同步多主控机复制主要对于读负载最好，尽管它的大优点是任意服务器都能接受写请求 — 没有必要在主服务器和后备服务器之间划分负载，并且因为数据修改被从一个服务器发送到另一个服务器，不会有非确定函数（如random()）的问题。

PostgreSQL不提供这种复制类型，尽管在应用代码或中间件中可以使用PostgreSQL的两阶段提交（PREPARE TRANSACTION和COMMIT PREPARED）来实现这种复制。

商业方案

Because 因为PostgreSQL是开源的并且很容易被扩展，一些公司已经使用PostgreSQL并且创建了带有唯一故障转移、复制和负载均衡能力的商业性的闭源方案。

表 26.1总结了上述多种方案的能力。

表 26.1. 高可用、负载均衡和复制特性矩阵

特性	共享磁盘故障转移	文件系统复制	预写式日志传送	逻辑复制	基于触发器的主-备复制	基于语句的复制中间件	异步多主控机复制	同步多主控机复制
最通用的实现	NAS	DRBD	内建流复制	内建Londiste，Slony逻辑复制，pglogical		pgpool-II	Bucardo	
通信方法	共享磁盘	磁盘块	WAL	逻辑解码	表行	SQL	表行	表行和行锁
不要求特殊硬件件		•	•	•	•	•	•	•
允许多个主控机服务器				•		•	•	•
无主服务器负载	•		•	•		•		
不等待多个服务器	•		with sync off	with sync off	•		•	
主控机失效将永不丢失数据	•	•	with sync on	with sync on		•		•
复制体接受只读查询			with hot	•	•	•	•	•
每个表粒度				•	•		•	•
不需要冲突解决	•	•	•		•			•

有一些方案不适合上述的类别：

数据分区

数据分区将表分开成数据集。每个集合只能被一个服务器修改。例如，数据可以根据办公室划分，如伦敦和巴黎，每一个办公室有一个服务器。如果查询有必要组合伦敦和巴黎的数据，一个应用可以查询两个服务器，或者可以使用主/备复制来在每一台服务器上保持其他办公室数据的一个只读拷贝。

多服务器并行查询执行

上述的很多方案允许多个服务器来处理多个查询，但是没有一个允许一个单一查询使用多个服务器来更快完成。这种方案允许多个服务器在一个单一查询上并发工作。这通常通过把数据在服务器之间划分并且让每一个服务器执行该查询中属于它的部分，然后将结果返回给一个中心服务器，由它整合结果并发回给用户。Pgpool-II具有这种能力。同样，也可以使用PL/Proxy工具集来实现这种方案。

26.2. 日志传送后备服务器

连续归档可以被用来创建一个高可用性（HA）集群配置，其中有一个或多个后备服务器随时准备在主服务器失效时接管操作。这种能力被广泛地称为温备或日志传送。

主服务器和后备服务器一起工作来提供这种能力，但这些服务器只是松散地组织在一起。主服务器在连续归档模式下操作，而每一个后备服务器在连续恢复模式下操作并且持续从主服务器读取 WAL 文件。要启用这种能力不需要对数据库表做任何改动，因此它相对于其他复制方案降低了管理开销。这种配置对主服务器的性能影响也相对较低。

直接从一个数据库服务器移动 WAL 记录到另一台服务器通常被描述为日志传送。PostgreSQL 通过一次一文件（WAL 段）的 WAL 记录传输实现了基于文件的日志传送。不管 WAL 文件（16 MB）要被送到一个临近的系统、同一站点的另一个系统或是在地球遥远的另一端的一个系统上，它都可以在任何距离上被简单和便宜地传送。这种技术所需的带宽取根据主服务器的事务率而变化。基于记录的日志传送具有更细的粒度并且能够在网络连接上以流的方式增量传递 WAL 的改变（见第 26.2.5 节。

需要注意的是日志传送是异步的，即 WAL 记录是在事务提交后才被传送。正因为如此，在一个窗口期内如果主服务器发生灾难性的失效则会导致数据丢失，还没有被传送的事务将会被丢失。基于文件的日志传送中这个数据丢失窗口的尺寸可以通过使用参数archive_timeout进行限制，它可以被设置成低至数秒。但是这样低的设置大体上会增加文件传送所需的带宽。流复制（见第 26.2.5 节允许更小的数据丢失窗口。

这种配置的恢复性能是足够好的，后备服务器在被激活后通常只有片刻就可以到达完全可用。因此，这被称为一种提供高可用性的温备配置。从一个已归档的基础备份恢复一个服务器并且前滚将需要较长时间，因此该技术只提供了灾难恢复的一种方案，而不适合于高可用性。一台后备服务器也可以被用于只读查询，在这种情况下它被称为一台热备服务器。更多信息请见第 26.5 节

26.2.1. 规划

创建主服务器和后备服务器通常是明智的，因此它们可以尽可能相似，至少从数据库服务器的角度来看是这样。特别地，与表空间相关的路径名将被未经修改地传递，因此如果该特性被使用，主、备服务器必须对表空间具有完全相同的挂载路径。记住如果CREATE TABLESPACE在主服务器上被执行，在命令被执行前，它所需要的任何新挂载点必须在主服务器和所有后备服务器上先创建好。硬件不需要完全相同，但是经验显示，在应用和系统的生命期内维护两个相同的系统比维护两个不相似的系统更容易。在任何情况下硬件架构必须相同 — 从一个 32 位系统传送到一个 64 位系统将不会工作。

通常，不能在两个运行着不同主版本PostgreSQL的服务器之间传送日志。PostgreSQL 全球开发组的策略是不在次版本升级中改变磁盘格式，因此在主服务器和后备服务器上运行不同次版本将会成功地工作。不过，在这方面并没有提供正式的支持，因此我们建议让主备服务器上运行的版本尽可能相同。当升级到一个新的次版本时，最安全的策略是先升级后备服务器 — 一个新的次版本发行更可能兼容从前一个次版本读取 WAL 文件。

26.2.2. 后备服务器操作

在后备模式中，服务器持续地应用从主控服务器接收到的 WAL。后备服务器可以从一个 WAL 归档（restore_command）或者通过一个 TCP 连接直接从主控机（流复制）读取 WAL。后备服务器将也尝试恢复在后备集簇的pg_wal目录中找到的 WAL。那通常在一次数据库重启后发生，那时后备机将在重启之前重播从主控机流过来的 WAL，但是你也可以在任何时候手动拷贝文件到pg_wal让它们被重播。

在启动时，后备机通过恢复归档位置所有可用的 WAL 来开始，这称为restore_command。一旦它到达那里可用的 WAL 的末尾并且restore_command失败，它会尝试恢复pg_wal目录中可用的任何 WAL。如果那也失败并且流复制已被配置，后备机会尝试连接到主服务器并且从在

归档或pg_wal中找到的最后一个可用记录开始流式传送 WAL。如果那失败并且没有配置流复制，或者该连接后来断开，后备机会返回到步骤 1 并且尝试再次从归档里的文件恢复。这种尝试归档、pg_wal和流复制的循环会一直重复知道服务器停止或者一个触发器文件触发了故障转移。

当pg_ctl promote被运行或一个触发器文件被找到（trigger_file），后备模式会退出并且服务器会切换到普通操作。在故障转移之前，在归档或pg_wal中立即可用的任何 WAL 将被恢复，但不会尝试连接到主控机。

26.2.3. 为后备服务器准备主控机

如第 25.3 节所述，在主服务器上设置连续归档到一个后备服务器可访问的归档目录。即使主服务器垮掉该归档位置也应当是后备服务器可访问的，即它应当位于后备服务器本身或者另一个可信赖的服务器，而不是位于主控服务器上。

如果你想要使用流复制，在主服务器上设置认证来允许来自后备服务器的复制连接。即创建一个角色并且在pg_hba.conf中提供一个或多个数据库域被设置为replication的项。还要保证在主服务器的配置文件中max_wal_senders被设置为足够大的值。如果要使用复制槽，请确保max_replication_slots也被设置得足够高。

按第 25.3.2 节所述取得一个基础备份来引导后备服务器。

26.2.4. 建立一个后备服务器

要建立后备服务器，恢复从主服务器取得的基础备份（第 25.3.4 节。在后备服务器的集簇数据目录中创建一个恢复命令文件recovery.conf，并且打开standby_mode。
将restore_command设置为一个从 WAL 归档中复制文件的简单命令。如果你计划为了高可用性目的建立多个后备服务器，将recovery_target_timeline设置为latest来使得该后备服务器遵循发生在故障转移到另一个后备服务器之后发生的时间线改变。

> **注意**
>
> 不要把 pg_standby 或相似的工具和这里描述的内建后备模式一起使用。如果文件不存在，restore_command应该立即返回，如果必要该服务器将再次尝试该命令。使用类似 pg_standby 的工具请见第 26.4 节

如果你想要使用流复制，在primary_conninfo中填入一个 libpq 连接字符串，其中包括主机名（或 IP 地址）和连接到主服务器所需的任何附加细节。如果主服务器需要一个口令用于认证，口令也应该被指定在primary_conninfo中。

如果你正在为高性能目的建立后备服务器，像主服务器一样建立 WAL 归档、连接和认证，因为在故障转移后该后备服务器将作为一个主服务器工作。

如果你正在使用一个 WAL 归档，可以使用archive_cleanup_command参数来移除后备服务器不再需要的文件，这样可以最小化 WAL 归档的尺寸。pg_archivecleanup工具被特别设计为在典型单一后备配置下与archive_cleanup_command共同使用，见pg_archivecleanup。不过要注意，如果你正在为备份目的使用归档，有一些文件即使后备服务器不再需要你也需要保留它们，它们被用来从至少最后一个基础备份恢复。

recovery.conf的一个简单例子是：

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

你可以有任意数量的后备服务器，但是如果你使用流复制，确保你在主服务器上将max_wal_senders设置得足够高，这样可以允许它们能同时连接。

26.2.5. 流复制

流复制允许一台后备服务器比使用基于文件的日志传送更能保持为最新的状态。后备服务器连接到主服务器，主服务器则在 WAL 记录产生时即将它们以流式传送给后备服务器而不必等到 WAL 文件被填充。

默认情况下流复制是异步的（见第 26.2.8 节，在这种情况下主服务器上提交一个事务与该变化在后备服务器上变得可见之间存在短暂的延迟。不过这种延迟比基于文件的日志传送方式中要小得多，在后备服务器的能力足以跟得上负载的前提下延迟通常低于一秒。在流复制中，不需要archive_timeout来缩减数据丢失窗口。

如果你使用的流复制没有基于文件的连续归档，该服务器可能在后备机收到 WAL 段之 前回收这些旧的 WAL 段。如果发生这种情况，后备机将需要重新从一个新的基础备 份初始化。通过设置wal_keep_segments为一个足够高的值来确保旧 的 WAL 段不会被太早重用或者为后备机配置一个复制槽，可以避免发生这种情况。如 果设置了一个后备机可以访问的 WAL 归档，就不需要这些解决方案，因为该归档可以 为后备机保留足够的段，后备机总是可以使用该归档来追赶主控机。

要使用流复制，按第 26.2 所述建立一个基于文件的日志传送后备服务器。将一个基于文件日志传送后备服务器转变成流复制后备服务器的步骤是把recovery.conf文件中的primary_conninfo设置指向主服务器。设置主服务器上的listen_addresses和认证选项（见pg_hba.conf），这样后备服务器可以连接到主服务器上的伪数据库replication（见第 26.2.5.1 节。

在支持 keepalive 套接字选项的系统上，设置tcp_keepalives_idle、tcp_keepalives_interval和tcp_keepalives_count有助于主服务器迅速地注意到一个断开的连接。

设置来自后备服务器的并发连接的最大数目（详见max_wal_senders）。

当后备服务器被启动并且primary_conninfo被正确设置，后备服务器将在重放完归档中所有可用的 WAL 文件之后连接到主服务器。如果连接被成功建立，你将在后备服务器中看到一个 walreceiver 进程，并且在主服务器中有一个相应的 walsender 进程。

26.2.5.1. 认证

设置好用于复制的访问权限非常重要，这样只有受信的用户可以读取 WAL 流，因为很容易从 WAL 流中抽取出需要特权才能访问的信息。后备服务器必须作为一个超级用户或一个具有REPLICATION特权的账户向主服务器认证。我们推荐为复制创建一个专用的具有REPLICATION和LOGIN特权的用户账户。虽然REPLICATION特权给出了非常高的权限，但它不允许用户修改主系统上的任何数据，而SUPERUSER特权则可以。

复制的客户端认证由一个在database域中指定replication的pg_hba.conf记录控制。例如，如果后备服务器运行在主机 IP 192.168.1.100并且用于复制的账户名为foo，管理员可以在主服务器上向pg_hba.conf文件增加下列行：

```
# 允许来自 192.168.1.100 的用户 "foo" 在提供了正确的口令时作为一个
# 复制后备机连接到主控机。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

主服务器的主机名和端口号、连接用户名和口令在recovery.conf文件中指定。在后备服务器上还可以在~/.pgpass文件中设置口令（在database域中指定replication）。例如，如果

主服务器运行在主机 IP 192.168.1.50、端口5432上，并且口令为foopass，管理员可以在后备服务器的recovery.conf文件中增加下列行：

```
# 后备机要连接到的主控机运行在主机 192.168.1.50 上，
# 端口号是 5432，连接所用的用户名是 "foo"，口令是 "foopass"。
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

26.2.5.2. 监控

流复制的一个重要健康指标是在主服务器上产生但还没有在后备服务器上应用的 WAL 记录数。你可以通过比较主服务器上的当前 WAL 写位置和后备服务器接收到的最后一个 WAL 位置来计算这个滞后量。这些位置分别可以用主服务器上的pg_current_wal_lsn和后备服务器上的pg_last_wal_receive_lsn来检索（详见表 9.79和表 9.80 。后备服务器的最后 WAL 接收位置也被显示在 WAL 接收者进程的进程状态中，即使用ps命令显示的状态（详见第 28.1 节。

你可以通过pg_stat_replication视图检索 WAL 发送者进程的列表。pg_current_wal_lsn与sent_lsn域之间的巨大差异表示主服务器承受着巨大的负载，而sent_lsn和后备服务器上pg_last_wal_receive_lsn之间的差异可能表示网络延迟或者后备服务器正承受着巨大的负载。

在一台热后备上，WAL接收者进程的状态可以通过pg_stat_wal_receiver视图检索到。pg_last_wal_replay_lsn和该视图的received_lsn的差别表示WAL的接收速度大于它被重放的速度。

26.2.6. 复制槽

复制槽提供了一种自动化的方法来确保主控机在所有的后备机收到 WAL 段 之前不会移除它们，并且主控机也不会移除可能导致 恢复冲突的行，即使后备机断开也是如此。

作为复制槽的替代，也可以使用wal_keep_segments 阻止移除旧的 WAL 段，或者使用archive_command 把段保存在一个归档中。不过，这些方法常常会导致保留的 WAL 段比需要的 更多，而复制槽只保留已知所需要的段。这些方法的一个优点是它们为 pg_wal的空间需求提供了界限，但目前使用复制槽无法做到。

类似地，hot_standby_feedback和 vacuum_defer_cleanup_age保护了相关行不被 vacuum 移除，但是前者在后备机断开期间无法提供保护，而后者则需要被设置为一个很高 的值已提供足够的保护。复制槽克服了这些缺点。

26.2.6.1. 查询和操纵复制槽

每个复制槽都有一个名字，名字可以包含小写字母、数字和下划线字符。

已有的复制槽和它们的状态可以在 pg_replication_slots 视图中看到。

槽可以通过流复制协议（见第 53.4 节 或者 SQL 函数（见第 9.26.6 节创建并且移除。

26.2.6.2. 配置实例

你可以这样创建一个复制槽：

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
```

```
  slot_name   | slot_type | active
--------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

要配置后备机使用这个槽，在后备机的recovery.conf中应该配置 primary_slot_name。这里是一个简单的例子：

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

26.2.7. 级联复制

级联复制特性允许一台后备服务器接收复制连接并且把 WAL 记录流式传送给其他后备服务器，就像一个转发器一样。这可以被用来减小对于主控机的直接连接数并且使得站点间的带宽开销最小化。

一台同时扮演着接收者和发送者角色的后备服务器被称为一台级联后备服务器。“更直接”（通过更少的级联后备服务器）连接到主控机的后备服务器被称为上游服务器，而那些离得更远的后备服务器被称为下游服务器。级联复制并没有对下游服务器的数量或布置设定限制。

一台级联后备服务器不仅仅发送从主控机接收到的 WAL 记录，还要发送那些从归档中恢复的记录。因此即使某些上游连接中的复制连接被中断，只要还有新的 WAL 记录可用，下游的流复制都会继续下去。

级联复制目前是异步的。同步复制（见第 26.2.8 节设置当前对级联复制无影响。

不管在什么样的级联布置中，热备反馈都会向上游传播。

如果一台上游后备服务器被提升为新的主控机，且下游服务器的recovery_target_timeline被设置成'latest'，下游服务器将继续从新的主控机得到流。

要使用级联复制，要建立级联后备服务器让它能够接受复制连接（即设置max_wal_senders和hot_standby，并且配置基于主机的认证）。你还将需要设置下游后备服务器中的primary_conninfo指向级联后备服务器。

26.2.8. 同步复制

PostgreSQL流复制默认是异步的。如果主服务器崩溃，则某些已被提交的事务可能还没有被复制到后备服务器，这会导致数据丢失。数据的丢失量与故障转移时的复制延迟成比例。

同步复制能够保证一个事务的所有修改都能被传送到一台或者多台同步后备服务器。这扩大了由一次事务提交所提供的标准持久化级别。在计算机科学理论中这种保护级别被称为 2-safe 复制。而当synchronous_commit被设置为remote_write时，则是 group-1-safe（group-safe 和 1-safe）。

在请求同步复制时，一个写事务的每次提交将一直等待，直到收到一个确认表明该提交在主服务器和后备服务器上都已经被写入到磁盘上的预写式日志中。数据会被丢失的唯一可能性是主服务器和后备服务器在同一时间都崩溃。这可以提供更高级别的持久性，尽管只有系统管理员要关系两台服务器的放置和管理。等待确认提高了用户对于修改不会丢失的信心，但是同时也不必要地增加了对请求事务的响应时间。最小等待时间是在主服务器和后备服务器之间的来回时间。

只读事务和事务回滚不需要等待后备服务器的回复。子事务提交也不需要等待后备服务器的响应，只有顶层提交才需要等待。长时间运行的动作（如数据载入或索引构建）不会等待最后的提交消息。所有两阶段提交动作要求提交等待，包括预备和提交。

同步后备可以是物理复制后备或者是逻辑复制订阅者。它还可以是任何其他物理或者逻辑WAL复制流的消费者，它懂得如何发送恰当的反馈消息。除内建的物理和逻辑复制系统之外，还包括pg_receivewal和pg_recvlogical之类的特殊程序，以及一些第三方复制系统和定制程序。同步复制支持的细节请查看相应的文档。

26.2.8.1. 基本配置

一旦流复制已经被配置，配置同步复制就只需要一个额外的配置步骤：synchronous_standby_names必须被设置为一个非空值。synchronous_commit也必须被设置为on，但由于这是默认值，通常不需要改变（见第 19.5.1 和第 19.6.2 节。这样的配置将导致每一次提交都等待确认消息，以保证后备服务器已经将提交记录写入到持久化存储中。synchronous_commit可以由个体用户设置，因此它可以在配置文件中配置、可以为特定用户或数据库配置或者由应用动态配置，这样可以在一种每事务基础上控制持久性保证。

在一个提交记录已经在主服务器上被写入到磁盘后，WAL 记录接着被发送到后备服务器。每次一批新的 WAL 数据被写入到磁盘后，后备服务器会发送回复消息，除非在后备服务器上wal_receiver_status_interval被设置为零。如果synchronous_commit被设置为remote_apply，当提交记录被重放时后备服务器会发送回应消息，这会让该事务变得可见。如果根据主服务器的synchronous_standby_names设置选中该后备服务器作为一个同步后备，将会根据来自该后备服务器和其他同步后备的回应消息来决定何时释放正在等待确认提交记录被收到的事务。这些参数允许管理员指定哪些后备服务器应该是同步后备。注意同步复制的配置主要在主控机上。命名的后备服务器必须直接连接到主控机，主控机对使用级联复制的下游后备服务器一无所知。

将synchronous_commit设置为remote_write将导致每次提交都等待后备服务器已经接收提交记录并将它写出到其自身所在的操作系统的确认，但并非等待数据都被刷出到后备服务器上的磁盘。这种设置提供了比on要弱一点的持久性保障：在一次操作系统崩溃事件中后备服务器可能丢失数据，尽管它不是一次PostgreSQL崩溃。不过，在实际中它是一种有用的设置，因为它可以减少事务的响应时间。只有当主服务器和后备服务器都崩溃并且主服务器的数据库同时被损坏的情况下，数据丢失才会发生。

把synchronous_commit设置为remote_apply将导致每一次提交都会等待，直到当前的同步后备服务器报告说它们已经重放了该事务，这样就会使该事务对用户查询可见。在简单的情况下，这为带有因果一致性的负载均衡留出了余地。

如果请求一次快速关闭，用户将停止等待。不过，在使用异步复制时，在所有未解决的 WAL 记录被传输到当前连接的后备服务器之前，服务器将不会完全关闭。

26.2.8.2. 多个同步后备

同步复制支持一个或者更多个同步后备服务器，事务将会等待，直到所有同步后备服务器都确认收到了它们的数据为止。事务必须等待其回复的同步后备的数量由synchronous_standby_names指定。这个参数还指定一个后备服务器名称及方法（FIRST和ANY）的列表来从列出的后备中选取同步后备。

方法FIRST指定一种基于优先的同步复制并且让事务提交等待，直到它们的WAL记录被复制到基于优先级选中的所要求数量的同步后备上为止。在列表中出现较早的后备被给予较高的优先级，并且将被考虑为同步后备。其他在这个列表中位置靠后的后备服务器表示可能的同步后备。如果任何当前的同步后备由于任何原因断开连接，它将立刻被下一个最高优先级的后备所替代。

基于优先的多同步后备的synchronous_standby_names示例为：

```
synchronous_standby_names = '2 (s1, s2, s3)'
```

在这个例子中，如果有四个后备服务器s1、s2、s3和s4在运行，两个后备服务器s1和s2将被选中为同步后备，因为它们出现在后备服务器名称列表的前部。s3是一个潜在的同步后备，

当s1或s2中的任何一个失效， 它就会取而代之。s4则是一个异步后备因为它的名字不在列表中。

方法ANY指定一种基于规定数量的同步复制并且让事务提交等待，直到它们的WAL记录至少被复制到列表中所要求数量的同步后备上为止。

synchronous_standby_names的基于规定数量的多同步后备的例子：

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

在这个例子中，如果有四台后备服务器s1、s2、s3以及s4正在运行，事务提交将会等待来自至少其中任意两台后备服务器的回复。s4是一台异步后备，因为它的名字不在该列表中。

后备服务器的同步状态可以使用pg_stat_replication视图查看。

26.2.8.3. 性能规划

同步复制通常要求仔细地规划和放置后备服务器来保证应用能令人满意地工作。等待并不利用系统资源，但是事务锁会持续保持直到传输被确认。其结果是，不小心使用同步复制将由于响应时间增加以及较高的争用率而降低数据库应用的性能。

PostgreSQL允许应用开发者通过复制来指定所要求的持久性级别。这可以为整个系统指定，不过它也能够为特定的用户或连接指定，甚至还可以为单个事务指定。

例如，一个应用的载荷的组成可能是这样：10% 的改变是重要的客户详情，而 90% 的改变是不太重要的数据，即使它们丢失业务也比较容易容忍（例如用户间的聊天消息）。

通过在应用级别（在主服务器上）指定的同步复制选项，我们可以为大部分重要的改变提供同步复制，并且不会拖慢整体的载荷。应用级别选项是使高性能应用享受同步复制的一种重要和实用的工具。

你应该认为网络带宽必须比 WAL 数据的产生率高。

26.2.8.4. 高可用性规划

当synchronous_commit被设置为on、remote_apply或者remote_write时，synchronous_standby_names指定产生的事务提交要等待其回应的同步后备的数量和名称。如果任一同步后备崩溃，这类事务提交可能无法完成。

高可用的最佳方案是确保有所要求数量的同步后备。这可以通过使用synchronous_standby_names指定多个潜在后备服务器来实现。

在基于优先的同步复制中，出现在该列表前部的后备服务器将被用作同步后备。后面的后备服务器将在当前同步后备服务器失效时取而代之。

在基于规定数量的同步复制中，所有出现在该列表中的后备服务器都将被用作同步后备的候选。即使其中的一个失效，其他后备仍将继续担任候选同步后备的角色。

当一台后备服务器第一次附加到主服务器时，它将处于一种还没有正确同步的状态。这被描述为追赶模式。一旦后备服务器和主服务器之间的迟滞第一次变成零，我们就来到了实时的流式状态。在后备服务器被创建之后的很长一段时间内可能都是追赶模式。如果后备服务器被关闭，则追赶周期将被增加，增加量由后备服务器被关闭的时间长度决定。只有当后备服务器到达流式状态后，它才能成为一台同步后备。这种状态可以使用pg_stat_replication视图查看。

如果在提交正在等待确认时主服务器重启，那些正在等待的事务将在主数据库恢复时被标记为完全提交。没有办法确认所有后备服务器已经收到了在主服务器崩溃时所有还未处理的 WAL 数据。某些事务可能不会在后备服务器上显示为已提交，即使它们在主服务器上显示为已提交。我们提供的保证是：在 WAL 数据已经被所有后备服务器安全地收到之前，应用将不会收到一个事务成功提交的显式确认。

如果实在无法保持所要求数量的同步后备，那么应该减少synchronous_standby_names中指定的事务提交应该等待其回应的同步后备的数量（或者禁用），并且在主服务器上重载配置文件。

如果主服务器与剩下的后备服务器是隔离的，你应当故障转移到那些其他剩余后备服务器中的最佳候选者上。

如果在事务等待时你需要重建一台后备服务器，确保命令 pg_start_backup() 和 pg_stop_backup() 被运行在一个synchronous_commit = off的会话中，否则那些请求将永远等待后备服务器出现。

26.2.9. 在后备机上连续归档

当在一个后备机上使用连续归档时，有两种不同的情景：WAL 归档在主服务器 和后备机之间共享，或者后备机有自己的 WAL 归档。当后备机拥有其自身的 WAL 归档时，将archive_mode设置为 always，后备机将在收到每个 WAL 段时调用归档命令， 不管它是从归档恢复还是使用流复制恢复。共享归档可以类似地处理，但是archive_command必须测试要被归档的文件是否 已经存在，以及现有的文件是否有相同的内容。这要求 archive_command中有更多处理，因为它必须当心 不要覆盖具有不同内容的已有文件，但是如果完全相同的文件被归档两次时 应返回成功。并且如果两个服务器尝试同时归档同一个文件，所有这些都必须 在没有竞争情况的前提下完成。

如果archive_mode被设置为on，归档器 在恢复或者后备模式中无法启用。如果后备服务器被提升，它将在被提升后开始 归档，但是它将不会归档任何不是它自身产生的 WAL。要在归档中得到完整的 一系列 WAL 文件，你必须确保所有 WAL 在到达后备机之前都被归档。对于基于 文件的日志传输来说天然就是这样，因为后备机只能恢复在归档中找到的文件， 而启用了流复制时则不是这样。当一台服务器不在恢复模式中时，在 on和always模式之间没有差别。

26.3. 故障转移

如果主服务器失效，则后备服务器应该开始故障转移过程。

如果后备服务器失效，则不会有故障转移发生。如果后备服务器可以被重启（即使晚一点），由于可重启恢复的优势，那么恢复处理也能被立即重启。如果后备服务器不能被重启，则一个全新的后备服务器实例应该被创建。

如果主服务器失效并且后备服务器成为了新的主服务器，那么接下来旧的主服务器重启后，你必须有一种机制来通知旧的主服务器不再成为主服务器。有些时候这被称为 STONITH（Shoot The Other Node In The Head，关闭其他节点），这对于避免出现两个系统都认为它们是主服务器的情况非常必要，那种情况将导致混乱并且最终导致数据丢失。

很多故障转移系统仅使用两个系统，主系统和后备系统，它们由某种心跳机制连接来持续验证两者之间的连接性和主系统的可用性。也可能会使用第三个系统（称为目击者服务器）来防止某些不当故障转移的情况，但是除非非常小心地建立它并且经过了严格地测试，额外的复杂度可能会使该工作得不偿失。

PostgreSQL并不提供在主服务器上标识失败并且通知后备数据库服务器所需的系统软件。现在已有很多这样的工具并且很好地与成功的故障转移所需的操作系统功能整合在一起，例如 IP 地址迁移。

一旦发生到后备服务器的故障转移，就只有单一的一台服务器在操作。这被称为一种退化状态。之前的后备服务器现在是主服务器，但之前的主服务器处于关闭并且可能一直保持关闭。要回到正常的操作，一个后备服务器必须被重建，要么在之前的主系统起来时使用它重建，要么使用第三台（可能是全新的）服务器来重建。在大型集簇上，pg_rewind功能可以被用来加速这个过程。一旦完成，主服务器和后备服务器可以被认为是互换了角色。某些人选择使用第三台服务器来为新的主服务器提供备份，直到新的后备服务器被重建，不过显然这会使得系统配置和操作处理更复杂。

因此，从主服务器切换到后备服务器可以很快，但是要求一些时间来重新准备故障转移集群。从主服务器到后备服务器的常规切换是有用的，因为它允许每个系统有常规的关闭时间来进行维护。这也可以作为一种对故障转移机制的测试，以保证在你需要它时它真地能够工作。我们推荐写一些管理过程来做这些事情。

要触发一台日志传送后备服务器的故障转移，运行pg_ctl promote或者创建一个触发器文件，其文件名和路径由recovery.conf中的trigger_file设置指定。如果你正在规划使用pg_ctl promote进行故障转移，trigger_file就不是必要的。如果你正在建立只用于从主服务器分流只读查询而不是高可用性目的的报告服务器，你不需要提升它。

26.4. 日志传送的替代方法

前一节描述的内建后备模式的一种替代方案是使用一个轮询归档位置的restore_command。这是版本 8.4 及以下版本中唯一可用的选项。在这种设置中，设置standby_mode为关闭，因为你要自行实现后备操作所需的轮询。关于这种实现的一个参考请见pg_standby模块。

注意在这种模式中，服务器将一次应用一整个文件的 WAL，因此如果你使用后备服务器来查询（见热备），那么主服务器上的一个动作和后备服务器上该动作变得可见之间会有一个延迟，该延迟对应着填满 WAL 文件的时间。archive_timeout可以被用来缩短该延迟。还要注意你不能把流复制和这种方法组合起来使用。

在主服务器和后备服务器上都会发生的操作是通常的连续归档和恢复任务。两个数据库服务器之间唯一的接触点是两者共享的 WAL 文件归档：主服务器写这个归档，后备服务器读取这个归档。必须要小心地保证来自独立主服务器的 WAL 归档不会混合在一起或者混淆。如果归档只被后备操作需要，它不必很大。

使得两台松耦合的服务器一起工作的诀窍是在后备服务器上使用的restore_command，当要求下一个 WAL 文件时，会等待它在主服务器上变得可用。restore_command在后备服务器的recovery.conf文件中指定。正常的恢复处理将从 WAL 归档请求一个文件，如果该文件不可用则会报告失败。对于后备处理来说下一个 WAL 文件不可用很正常，因此后备服务器必须等待它出现。对于以.backup或.history结尾的文件没有必要等待，并且必须返回一个非零的返回码。一个等待的restore_command可以用一种习惯的脚本编写，在其中轮询下一个 WAL 文件的存在之后进行循环。也必须有某种方式来触发故障转移，那将打断restore_command：打破循环并返回一个文件未找到错误给后备服务器。这会结束恢复并且后备服务器将接下来变成一个正常的服务器。

一个合适的restore_command的伪代码是：

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

在pg_standby模块中提供了一个等待的restore_command的工作例子。它也可被用作如何正确实现上述逻辑的参考。它也可以根据需要被扩展来支持指定的配置和环境。

触发故障转移的方法是规划和设计中的一个重要部分。一种潜在的选项是restore_command命令。它对每一个 WAL 文件被执行一次，但是运行restore_command的进程会为每一个文件创建和死亡，因此没有守护进程或服务器进程，并且也不能使用信号或信号句柄。因此，restore_command不适合于触发故障转移。可以使用一种简单的超时功能，特别是和主服务器上已知的archive_timeout设置一起。但是，由于一个网络问题或者繁忙的主服务器可能足以发起故障转移，这有点容易产生错误。如果可以安排，一种提醒机制（例如显式创建一个触发器文件）是最理想的。

26.4.1. 实现

使用这种替代方案配置一个后备服务器的简短过程如下所示。对于每一步的细节，可以参考之前的小节。

1. 尽可能将主系统和后背系统设置成近乎一致，包括在同一发行级别上的两个相同的PostgreSQL拷贝。

2. 在后备服务器上建立从主系统到一个 WAL 归档目录的连续归档。确保在主服务器上archive_mode、archive_command和archive_timeout被恰当地设置（见第 25.3.1 节。

3. 建立主服务器的一个基础备份（见第 25.3.2 节，并且把该数据载入到后备服务器。

4. 在后备服务器上开始从本地 WAL 归档的恢复，在recovery.conf中指定一个按之前所述进行等待的restore_command（见第 25.3.4 节。

恢复将 WAL 归档当作只读的来处理，因此一旦一个 WAL 文件已经被复制到后备系统，在它正在被后备数据库服务器读取时可以被同时复制到磁带。因此，可以在为了长期灾难恢复目的存储文件的同时运行一个用于高可用性的后备服务器。

为了测试的目的，可以在一个相同的系统上运行主服务器和后备服务器。这对于服务器鲁棒性并不会提供任何有意义的改进，对 HA 也一样。

26.4.2. 基于记录的日志传送

也可以使用这种替代方法来实现基于记录的日志传送，不过这需要定制开发，并且只有在一整个 WAL 文件被传送之后改变才会对热后备查询可见。

一个外部程序可以调用pg_walfile_name_offset()函数（见第 9.26 节来找出 WAL 的当前末端的文件名和其中准确的字节偏移。它接着可以直接访问 WAL 文件并且将从上一个已知的 WAL 末尾到当前末尾之间的数据拷贝到后备服务器。通过这种方法，数据丢失的窗口是复制程序的轮询周期时间，这可以为非常小，并且不会有强制部分使用的段文件被归档所浪费的带宽。注意后备服务器的restore_command脚本只能处理整个 WAL 文件，因此增量复制的数据通常不会对后备服务器可用。只有当主服务器死掉时它才有用 — 那时最后一个部分 WAL 文件会在允许它发生之前被喂给后备服务器。这个处理的正确实现要求restore_command脚本和数据复制程序的合作。

从PostgreSQL 版本 9.0 开始，你可以使用流复制（见第 26.2.5 节来实现事半功倍的效果。

26.5. 热备

术语热备用来描述处于归档恢复或后备模式中的服务器连接到服务器并运行只读查询的能力。这有助于复制目的以及以高精度恢复一个备份到一个期望的状态。术语热备也指服务器从恢复转移到正常操作而用户能继续运行查询并且保持其连接打开的能力。

在热备模式中运行查询与正常查询操作相似，尽管如下所述存在一些用法和管理上的区别。

26.5.1. 用户概览

当hot_standby参数在一台后备服务器上被设置为真时，一旦恢复将系统带到一个一致的状态它将开始接受连接。所有这些连接都被限制为只读，甚至临时表都不能被写入。

后备服务器上的数据需要一些时间从主服务器到达后备服务器，因此在主服务器和后备服务器之间将有一段可以度量的延迟。近乎同时在主服务器和后备服务器上运行相同的查询可能因此而返回不同的结果。我们说后备服务器上的数据与主服务器是最终一致的。一旦一个事务的提交记录在后备服务器上被重播，那个事务所作的修改将对后备服务器上所有新取得的

快照可见。快照可以在每个查询或每个事务的开始时取得，这取决于当前的事务隔离级别。详见第 13.2 节

在热备期间开始的事务可能发出下列命令：

- 查询访问 - SELECT、COPY TO
- 游标命令 - DECLARE、FETCH、CLOSE
- 参数 - SHOW、SET、RESET
- 事务管理命令
  - BEGIN、END、ABORT、START TRANSACTION
  - SAVEPOINT、RELEASE、ROLLBACK TO SAVEPOINT
  - EXCEPTION块或其他内部子事务
- LOCK TABLE，不过只在下列模式之一中明确发出： ACCESS SHARE、ROW SHARE 或 ROW EXCLUSIVE.
- 计划和资源 - PREPARE、EXECUTE、 DEALLOCATE、DISCARD
- 插件和扩展 - LOAD
- UNLISTEN

在热备期间开始的事务将不会被分配一个事务 ID 并且不能被写入到系统的预写式日志。因此，下列动作将产生错误消息：

- 数据操纵语言（DML） - INSERT、 UPDATE、DELETE、COPY FROM、 TRUNCATE。注意不允许在恢复期间导致一个触发器被执行的动作。这个限制甚至被应用到临时表，因为不分配事务 ID 表行就不能被读或写，而当前不可能在一个热备环境中分配事务 ID。
- 数据定义语言（DDL） - CREATE、 DROP、ALTER、COMMENT。这个限制甚至被应用到临时表，因为执行这些操作会要求更新系统目录表。
- SELECT ... FOR SHARE | UPDATE，因为不更新底层数据文件就无法取得行锁。
- SELECT语句上的能产生 DML 命令的规则。
- 显式请求一个高于ROW EXCLUSIVE MODE的模式的LOCK。
- 默认短形式的LOCK，因为它请求ACCESS EXCLUSIVE MODE。
- 显式设置非只读状态的事务管理命令：
  - BEGIN READ WRITE、 START TRANSACTION READ WRITE
  - SET TRANSACTION READ WRITE、 SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
  - SET transaction_read_only = off
- 两阶段提交命令 - PREPARE TRANSACTION、 COMMIT PREPARED、ROLLBACK PREPARED，因为即使只读事务也需要在准备阶段（两阶段提交中的第一个阶段）写 WAL。
- 序列更新 - nextval()、setval()
- LISTEN、NOTIFY

在正常操作中，“只读”事务被允许使用LISTEN和NOTIFY，因此热备会话在比普通只读会话更紧一点的限制下操作。这些限制中的某些可能会在一个未来的发行中被放松。

在热备期间，参数transaction_read_only总是为真并且不可以被改变。但是只要不尝试修改数据库，热备期间的连接工作起来更像其他数据库连接。如果发生故障转移或切换，该数据库将切换到正常处理模式。当服务器改变模式时会话将保持连接。一旦热备结束，它将可以发起读写事务（即使是一个在热备期间启动的会话）。

用户将可以通过发出SHOW transaction_read_only来了解他们的会话是不是只读的。此外，一组函数（表 9.80 允许用户访问关于后备服务器的信息。这些允许你编写关心数据库当前状态的程序。这些可以被用来监控恢复的进度，或者允许你编写恢复数据库到特定状态的复杂程序。

26.5.2. 处理查询冲突

主服务器和后备服务器在多方面都松散地连接在一起。主服务器上的动作将在后备服务器上产生效果。结果是在它们之间有潜在的负作用或冲突。最容易理解的冲突是性能：如果在主服务器上发生一次大数据量的载入，那么着将在后备服务器上产生一个相似的 WAL 记录流，因而后备服务器查询可能要竞争系统资源（例如 I/O）。

随着热备发生的还可能有其他类型的冲突。对于可能需要被取消的查询和（某些情况中）解决它们的已断开会话来说，这些冲突是硬冲突。用户可以用几种方式来处理这种冲突。冲突情况包括：

- 在主服务器上取得了访问排他锁（包括显式LOCK命令和多种DDL动作）与后备查询中的表访问冲突。
- 在主服务器上删除一个表空间与使用该表空间存储临时工作文件的后备查询冲突。
- 在主服务器上删除一个数据库与在后备服务器上连接到该数据库的会话冲突。
- 从 WAL 清除记录的应用与快照仍能“看见”任意要被移除的行的后备事务冲突。
- 从 WAL 清除记录的应用与在后备服务器上访问该目标页的查询冲突，不管要被移除的数据是否为可见。

在主服务器上，这些情况仅仅会导致等待；并且用户可以选择取消这些冲突动作中间的一个。但是，在后备服务器上则没有选择：已被 WAL 记录的动作已经在主服务器上发生，那么后备服务器不能在应用它时失败。此外，允许 WAL 应用无限等待是非常不可取的，因为后备服务器的状态将变得逐渐远远落后于主服务器的状态。因此，提供了一种机制来强制性地取消与要被应用的 WAL 记录冲突的后备查询。

该问题情形的一个例子是主服务器上的一位管理员在一个表上运行DROP TABLE，而该表正在后备服务器上被查询。如果DROP TABLE被应用在后备服务器上，很明显该后备查询不能继续。如果这种情况在主服务器上发生，DROP TABLE将等待直到其他查询结束。但是当DROP TABLE被运行在主服务器上，主服务器没有关于运行在后备服务器上查询的信息，因此它将不会等待任何这样的后备查询。WAL 改变记录在后备查询还在运行时来到后备服务器上，导致一个冲突。后备服务器必须要么延迟 WAL 记录的应用（还有它们之后的任何事情），要么取消冲突查询这样DROP TABLE可以被应用。

当一个冲突查询很短时，我们通常期望能延迟 WAL 应用一小会儿让它完成；但是在 WAL 应用中的一段长的延迟通常是不受欢迎的。因此取消机制有参数，max_standby_archive_delay和max_standby_streaming_delay，它们定义了在 WAL 应用中的最大允许延迟。当应用任何新收到的 WAL 数据花费了超过相关延迟设置值时，冲突查询将被取消。设立两个参数是为了对从一个归档读取 WAL 数据（即来自一个基础备份的初始恢复或者“追赶”一个已经落后很远的后备服务器）和通过流复制读取 WAL数据的两种情况指定不同的延迟值。

在一台后备服务器上这主要是为了该可用性而存在，最好把延迟参数设置得比较短，这样服务器不会由于后备查询导致的延迟落后主服务器太远。但是，如果该后备服务器是位了执行

长时间运行的查询，则一个较高甚至无限的延迟值更好。但是记住一个长时间运行的查询延迟了 WAL 记录的应用，它可能导致后备服务器上的其他会话无法看到主服务器上最近的改变。

一旦max_standby_archive_delay或max_standby_streaming_delay指定的延迟被超越，冲突查询将被取消。这通常仅导致一个取消错误，尽管在重放一个DROP DATABASE的情况下整个冲突会话都将被中断。另外，如果冲突发生在一个被空闲事务持有的锁上，该冲突会话会被中断（这种行为可能在未来被改变）。

被取消的查询可能会立即被重试（当然是在开始一个新的事务后）。因为查询取消依赖于 WAL 记录被重放的本质，如果一个被取消的查询被再次执行，它可能会很好地成功完成。

记住延迟参数是从 WAL 数据被后备服务器收到后流逝的时间。因此，留给后备服务器上任何一个查询的宽限期从不会超过延迟参数，并且如果后备服务器已经由于等待之前的查询完成而落后或者因为过重的更新负载而无法跟上主服务器，宽限期可能会更少。

在后备查询和 WAL 重播之间发生冲突的最常见原因是“过早清除”。正常地，PostgreSQL允许在没有事务需要看到旧行版本时对它们进行清除，这样可以保证根据 MVCC 规则的正确的数据可见性。不过，这个规则只能被应用于执行在主控机上的事务。因此有可能主控机上的清除会移除对一个后备服务器事务还可见的行版本。

有经验的用户应当注意行版本清除和行版本冻结都可能与后备查询冲突。即便在一个没有被更新或被删除行的表上运行一次手工VACUUM FREEZE也可能导致冲突。

用户应当清楚，主服务器上被正常和重度更新的表将快速地导致后备服务器上长时间运行的查询被取消。在这样的情况
下，max_standby_archive_delay或max_standby_streaming_delay的有限制设置可以被视作statement_timeout设置。

如果发现后备查询取消的数量不可接受，还是有补救的可能。第一种选项是设置参数hot_standby_feedback，它阻止VACUUM 移除最近死亡的元组并且因此清除冲突不会产生。如果你这样做，你应当 注意这将使主服务器上的死亡元组清除被延迟，这可能会导致不希望发生 的表膨胀。不过，清除的情况不会比在主服务器上直接运行后备查询时更糟， 并且你仍然能够享受将执行分流到后备服务器的好处。如果后备服务器频繁地连接和 断开，你可能想要做些调整来处理无法提供hot_standby_feedback 反馈的时期。例如，考虑增加max_standby_archive_delay，这样 在断开连接的期间查询就不会快速地被 WAL 归档文件中的冲突取消。你也应该考虑 增加max_standby_streaming_delay来避免重新连接后新到达的流 WAL 项导致的快速取消。

另一个选项是增加主服务器上的vacuum_defer_cleanup_age，这样死亡行不会像平常那么快地被清理。这将允许在后备服务器上的查询能在被取消前有更多时间执行，并且不需要设置一个很高的max_standby_streaming_delay。但是，这种方法很难保证任何指定的执行时间窗口，因为vacuum_defer_cleanup_age是用主服务器上被执行的事务数来衡量的。

查询取消的数量和原因可以使用后备服务器上的pg_stat_database_conflicts系统视图查看。pg_stat_database系统视图也包含汇总信息。

26.5.3. 管理员概览

如果hot_standby在postgresql.conf中被设置为on并且存在一个recovery.conf文件，服务器将运行在热备模式。但是，可能需要一些时间来允许热备连接，因为在服务器完成足够的恢复来为查询提供一个一致的状态之前，它将不会接受连接。在此期间，尝试连接的客户端将被一个错误消息拒绝。要确认服务器已经可连接，要么循环地从应用尝试连接，要么在服务器日志中查找这些消息：

```
LOG:  entering standby mode

... then some time later ...
```

```
LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

在主服务器上，一旦创建一个检查点，一致性信息就被记录下来。当读取在特定时段（当在主服务器上wal_level没有被设置为replica或者logical的期间）产生的 WAL 时无法启用热备。在同时存在这些条件时，到达一个一致状态也会被延迟：

- 一个写事务有超过 64 个子事务
- 生存时间非常长的写事务

如果你正在运行基于文件的日志传送（“温备”），你可能需要等到下一个 WAL 文件到达，这可能和主服务器上的archive_timeout设置一样长。

如果后备服务器上某些参数在主服务器上已经被改变，它们的设置将需要重新配置。对这些参数，在后备服务器上的值必须等于或者大于主服务器上的值。因此，如果想要增加这些值，就应该在把这些更改应用到主服务器之前，先在所有的后备服务器上做同样的事情。反过来，如果想要减小这些值，就应该在把这些更改应用到所有后备服务器之前，先在主服务器上做同样的事情。如果这些参数没有被设置得足够高，后备服务器将拒绝开始。较高的值被提供之后，服务器重新启动再次开始恢复。这些参数是：

- max_connections
- max_prepared_transactions
- max_locks_per_transaction
- max_worker_processes

管理员为max_standby_archive_delay和max_standby_streaming_delay选择适当的设置很重要。最好的选择取决于业务的优先级。例如如果服务器主要的任务是作为高可用服务器，那么你将想要低延迟设置，甚至是零（尽管它是一个非常激进的设置）。如果后备服务器的任务是作为一个用于决策支持查询的额外服务器，那么将其最大延迟值设置为很多小时甚至 -1 （表示无限等待）可能都是可以接受的。

在主服务器上写出的事务状态 "hint bits" 是不被 WAL 记录的，因此后备服务器上的数据将可能重新写出该提示。这样，即使所有用户都是只读的，后备服务器仍将执行磁盘写操作；但数据值本身并没有发生改变。用户将仍写出大的排序临时文件并且重新生成 relcache 信息文件，这样在热备模式中数据库没有哪个部分是真正只读的。还要注意使用dblink模块写到远程数据库以及其他使用 PL 函数位于数据库之外的操作仍将可用，即使该事务是本地只读的。

在恢复模式期间，下列类型的管理命令是不被接受的：

- 数据定义语言（DDL） - 如CREATE INDEX
- 特权和所有权 - GRANT、REVOKE、REASSIGN
- 维护命令 - ANALYZE、VACUUM、CLUSTER、REINDEX

注意这些命令中的某些实际上在主服务器上的“只读”模式事务期间是被允许的。

结果是，你无法创建只存在于后备服务器上的额外索引以及统计信息。如果需要这些管理命令，它们应该在主服务器上被执行，并且最后那些改变将被传播到后备服务器。

pg_cancel_backend()和pg_terminate_backend()将在用户后端上工作，而不是执行恢复的 Startup 进程。pg_stat_activity不会为 Startup 进程显示一个项，也不会把恢复事务显示为活动。结果是在恢复期间pg_prepared_xacts总是为空。如果你希望解决不能确定的预备事务，查看主服务器上的pg_prepared_xacts并且发出命令来解决那里的事务或者在恢复结束后来解决它们。

和平常一样，pg_locks将显示被后端持有的锁。pg_locks也会显示一个由 Startup 进程管理的虚拟事务，它拥有被恢复重播的事务所持有的所有AccessExclusiveLocks。注意 Startup 进程不请求锁来做数据库更改，并且因此对于 Startup 进程除AccessExclusiveLocks之外的锁不显示在pg_locks中，它们仅被假定存在。

Nagios的插件check_pgsql将可以工作，因为它检查的简单信息是存在的。check_postgres监控脚本也将能工作，尽管某些被报告的值可能给出不同或者混乱的结果。例如，上一次清理时间将不会被维护，因为在后备服务器上不会发生清理。在主服务器上运行的清理仍会把它们的改变发送给后备服务器。

WAL 文件控制命令在恢复期间将不会工作，如pg_start_backup、pg_switch_wal等。

可动态载入的模块可以工作，包括pg_stat_statements。

咨询锁在恢复期间工作正常，包括死锁检测。注意咨询锁从来都不会被 WAL 记录，因此在主服务器或后备服务器上一个咨询锁不可能会与 WAL 重播发生冲突。也不可能会在主服务器上获得一个咨询锁并且在后备服务器上开始一个相似的咨询锁。咨询锁只与它们被取得的那个服务器相关。

基于触发器的复制系统（如Slony、Londiste和Bucardo）将根本不会运行在后备服务器上，然而只要改变不被发送到要被应用的后备服务器，它们将在主服务器上运行得很好。WAL 重播不是基于触发器的，因此你不能用后备服务器接替任何需要额外数据库写操作或依赖触发器使用的系统。

新的 OID 不能被分配，然而某些UUID生成器仍然能工作，只要它们不依赖于向数据库写新的状态。

当前，在只读事务期间不允许创建临时表，因此在某些情况中现有的脚本将不会正确运行。这个限制可能会在稍后的发行中被放松。这既是一个 SQL 标准符合问题也是一个技术问题。

只有在表空间为空时DROP TABLESPACE才能成功。某些后备服务器用户可能正在通过他们的temp_tablespaces参数使用该表空间。如果在该表空间中有临时文件，所有活动查询将被取消来保证临时文件被移除，这样该表空间可以被移除并且 WAL 重播可以继续。

在主服务器上运行DROP DATABASE或ALTER DATABASE ... SET TABLESPACE将产生一个 WAL 项，它将导致所有连接到后备服务器上那个数据库的用户被强制地断开连接。这个动作会立即发生，不管max_standby_streaming_delay的设置是什么。注意ALTER DATABASE ... RENAME不会断开用户，这在大部分情况中不会有提示，然而如果它依赖某种基于数据库名的方法，在某些情况中会导致程序混乱。

在普通（非恢复）模式中，如果你为具有登录能力的角色发出DROP USER或DROP ROLE，而该用户仍然连接着，则对已连接用户不会发生任何事情 - 他们保持连接。但是用户不能重新连接。这种行为也适用于恢复，因此在主服务器上的一次DROP USER不会使后备服务器上的用户断开。

在恢复期间统计收集器是活动的。所有扫描、读、阻塞、索引使用等将在后备服务器上被正常的记录。被重播的动作将不会重复它们在主服务器上的效果，因此重播一个插入将不会导致pg_stat_user_tables的 Inserts 列上的递增。在恢复的开始 stats 文件会被删除，因此来自主服务器和后备服务器的 stats 将不同；这被认为是一种特性而不是缺陷。

在恢复期间自动清理不是活动的。它将在恢复末尾正常启动。

后台写入器在恢复期间是活动的并且将执行检查点（与主服务器上的检查点相似）以及正常的块清洁活动。这可以包括存储在后备服务器上的提示位信息的更新。在恢复期间，CHECKPOINT命令会被接受，然而它会执行一个重启点而不是一个新的检查点。

26.5.4. 热备参数参考

多个参数已经在第 26.5.2 和第 26.5.3 节提到过。

在主服务器上，可以使用参数wal_level和vacuum_defer_cleanup_age。在主服务器上设置max_standby_archive_delay和max_standby_streaming_delay不会产生效果。

在主服务器上，可以使用参数hot_standby、max_standby_archive_delay和max_standby_streaming_delay。只要服务器保持在后备模式vacuum_defer_cleanup_age就没有效果，然而当后备服务器变成主服务器时它将变得相关。

26.5.5. 警告

热备有一些限制。这些限制很可能在未来的发行中被修复：

- 在能够取得快照之前，需要正在运行的事务的完整知识。使用大量子事务（目前指超过 64 个）的事务将延迟只读连接的启动，直到最长的运行着的写事务完成。如果发生这种情况，说明消息将被发送到服务器日志。

- 主服务器上的每一个检查点将产生用于后备查询的可用启动点。如果后备服务器在主控机处于关闭状态时被关闭，就没有办法在主服务器启动之前重新进入热后备，因此它在 WAL 日志中产生一个进一步启动点。这种情况在它可能发生的大部分常见情况中不是一个问题。通常，如果主服务器被关闭并且不再可用，这可能是由于某种严重错误要求后备服务器被转变成为一个新的主服务器来操作。并且在主服务器被故意关闭的情况下，协调保证后备服务器平滑地过渡为新的主服务器也是一种标准过程。

- 在恢复尾声，由预备事务持有的AccessExclusiveLocks将要求两倍的正常锁表项。如果你计划运行大量并发的通常要求AccessExclusiveLocks的预备事务，或者你计划运行一个需要很多AccessExclusiveLocks的大型事务，我们建议你为max_locks_per_transaction选择一个更大的值，也许是主服务器上该参数值的两倍。如果你的max_prepared_transactions设置为 0，你根本不需要考虑这个问题。

- 可序列化事务隔离级别目前在热备中不可用（详见第 13.2.3 和第 13.4.1 节。尝试在热备模式中将一个事务设置为可序列化隔离级别将产生一个错误。

第 27 章 恢复配置

这一章描述recovery.conf文件中可用的设置。它们只应用于恢复期。对于你希望执行的任意后续恢复，它们必须被重置。一旦恢复已经开始，它们就不能被更改。

recovery.conf中的设置以name = 'value'形式指定。每一行指定一个参数。井号（#）表示行的剩余部分是一段注释。要在一个参数值中嵌入一个单引号，将其双写（''）。

作为一个例子文件，share/recovery.conf.sample被放置在安装的share/目录中。

27.1. 归档恢复设置

restore_command (string)

用于获取 WAL 文件系列的一个已归档段的本地 shell 命令。这个参数是归档恢复所必需的，但是对于流复制是可选的。在该字符串中的任何%f会被替换为从归档中获得的文件的名字，并且任何%p会被在服务器上的复制目标路径名替换（该路径名是相对于当前工作目录的，即集簇的数据目录）。任何%r会被包含上一个可用重启点的文件的名字所替换。在那些必须被保留用于使得一次恢复变成可重启的文件中，这个文件是其中最早的一个，因此这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。%r通常只被温备配置（见第 26.2 节所使用。要嵌入一个真正的%字符，需要写成%%。

很重要的一点是，该命令只有在成功时才返回一个为零的退出状态。该命令将会被询问不存在于归档中的文件名，当这样被询问时它必须返回非零。例子：

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

一个例外是如果该命令被一个信号（不是SIGTERM，它是数据库服务器关闭的一部分）或者一个 shell 错误（例如命令未找到）终止，则恢复将会中止并且服务器将不会启动。

archive_cleanup_command (string)

这个可选参数指定了一个 shell 命令，它将在每一个重启点被执行。archive_cleanup_command的目的是提供一种清除不再被后备服务器需要的旧的已归档 WAL 文件的机制。任何%r会被替换为包含最后一个可用重启点的文件的名称。那是使一次恢复变成可重启的所必须被保留的最早的文件，并且因此比%r更早的所有文件可以被安全地移除。这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。对于单一后备配置，pg_archivecleanup模块常常被用在archive_cleanup_command中，例如：

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

但是注意，如果多个后备服务器正在从同一个归档目录中恢复，你将需要保证只有当任意服务器都不再需要 WAL 文件时才会删除它们。archive_cleanup_command通常被用于一种温后备配置（见第 26.2 节中。要在该命令中嵌入一个真正的%字符，需要写成%%。

如果该命令返回一个非零退出状态，则将会写出一个警告日志消息。一个例外是如果该命令被一个信号或者一个 shell 错误（例如命令未找到）终止，则会抛出一个致命错误。

recovery_end_command (string)

这个参数指定了一个将只在恢复末尾被执行一次的 shell 命令。这个参数是可选的。recovery_end_command的目的是为复制或恢复之后的清除提供一种机制。
与archive_cleanup_command中相似，任何%r会被替换为包含最后一个可用重启点的文件的名称。

如果该命令返回一个非零退出状态，则一个警告日志消息将被写出并且不管怎样该数据库将继续启动。一个例外是如果该命令被一个信号或者 shell 错误（例如命令未找到）中止，该数据库将不会继续启动。

27.2. 恢复目标设置

默认情况下，恢复将会一直恢复到 WAL 日志的末尾。下面的参数可以被用来指定一个更早的停止点。
在recovery_target、recovery_target_lsn、recovery_target_name、recovery_target_time和recovery_ta
最多只能使用一个，如果在配置文件中使用了多个，将使用最后一个。

recovery_target = 'immediate'

这个参数指定恢复应该在达到一个一致状态后尽快结束，即尽早结束。在从一个在线备份中恢复时，这意味着备份结束的那个点。

在技术上，这是一个字符串参数，但是'immediate'是目前唯一允许的值。

recovery_target_name (string)

这个参数指定（pg_create_restore_point()所创建）的已命名的恢复点，恢复将进入该恢复点。

recovery_target_time (timestamp)

这个参数指定恢复将进入的时间戳。

recovery_target_xid (string)

这个参数指定恢复将进入的事务 ID。记住虽然事务 ID 是在事务开始时顺序分配的，但是事务可能以不同的数字顺序完成。那些在指定事务之前（也可以包括该事务）提交的事务将被恢复。精确的停止点也受到recovery_target_inclusive的影响。

recovery_target_lsn (pg_lsn)

此参数指定恢复将继续进行的预写日志位置的LSN。精确的停靠点也受recovery_target_inclusive的影响。 使用系统数据类型pg_lsn解析此参数。

下列选项进一步指定恢复目标，并且影响到达目标时会发生什么：

recovery_target_inclusive (boolean)

指定我们是否仅在指定的恢复目标之后停止（true），或者仅在恢复目标之前停止（false）。适用于recovery_target_lsn、recovery_target_time或者recovery_target_xid被指定的情况。 这个设置分别控制事务是否有准确的目标WAL位置(LAN)、提交时间或事务ID将被包括在该恢复中。 默认值为true。

recovery_target_timeline (string)

指定恢复到一个特定的时间线中。默认值是沿着基础备份建立时的当前时间线恢复。将这个参数设置为latest会恢复到该归档中能找到的最新的时间线，这在一个后备服务器中有用。除此之外，你只需要在复杂的重恢复情况下设置这个参数，在这种情况下你需要返回到一个状态，该状态本身是在一次时间点恢复之后到达的。相关讨论见第 25.3.5 节

recovery_target_action (enum)

指定在达到恢复目标时服务器应该立刻采取的动作。默认动作是pause，这表示恢复将会被暂停。promote表示恢复处理将会结束并且服务器将开始接受连接。最后，shutdown将在达到恢复目标之后停止服务器。

使用pause设置的目的是：如果这个恢复目标就是恢复最想要的位置，就允许对数据库执行查询。暂停的状态可以使用pg_wal_replay_resume()（见表 9.80）继续，这会让恢复终结。如果这个恢复目标不是想要的停止点，那么关闭服务器，将恢复目标设置改为一个稍后的目标并且重启以继续恢复。

要让实例在想要的重放点那里准备好，shutdown设置可以派上用场。该实例将仍能重放更多 WAL 记录（并且事实上将不得不重放从下一次它被启动后最后一个检查点以来的 WAL 记录）。

注意由于在recovery_target_action被设置为shutdown时，recovery.conf将不会被重命名，任何后续的启动都将会以立刻关闭为终结，除非该配置被改变或者recovery.conf文件被手工移除。

如果没有设置恢复目标，这个设置没有效果。如果没有启用hot_standby，pause设置的动作将和shutdown一样。

27.3. 后备服务器设置

standby_mode (boolean)

指定是否将PostgreSQL服务器作为一个后备服务器启动。如果这个参数为on，当到达已归档 WAL 末尾时该服务器将不会停止恢复，但是将通过使用restore_command获得新的 WAL 段以及/或者通过使用primary_conninfo设置连接到主服务器来尝试继续恢复。

primary_conninfo (string)

指定后备服务器用来连接主服务器的连接字符串。这个字符串的格式在第 34.1.1 节描述。如果在这个字符串中有任何选项未被指定，那么将检查相应的环境变量（见第 34.14 节。如果环境变量也没有被设置，则使用默认值。

连接字符串应当指定主服务器的主机名（或地址），以及端口号（如果它和后备服务器的默认端口不同）。还要指定对应于主服务器上合适权限角色的用户名（见第 26.2.5.1 节。如果主服务器要求口令认证，还需要提供一个口令。它可以在primary_conninfo字符串中提供，或者在后备服务器（使用replication作为数据库名）的一个单独~/.pgpass文件中提供。不要在primary_conninfo字符串中指定一个数据库名。

如果standby_mode为off，这个设置没有效果。

primary_slot_name (string)

有选择地指定通过流复制连接到主服务器时使用一个现有的复制槽来控制上游节点上的资源移除（见第 26.2.6 节。如果没有设置primary_conninfo则这个设置无效。

trigger_file (string)

指定一个触发器文件，该文件的存在会结束后备机中的恢复。即使这个值没有被设置，你也能够使用pg_ctl promote来提升后备机。如果standby_mode为off，这个设置没有效果。

recovery_min_apply_delay (integer)

某人情况下，一个后备服务器会尽快恢复来自于主服务器的 WAL 记录。有一份数据的延时拷贝是有用的，它能提供机会纠正数据丢失错误。这个参数允许你将恢复延迟一段固定的时间，如果没有指定单位则以毫秒为单位。例如，如果你设置这个参数为5min，对于一个事务提交，只有当后备机上的系统时钟超过主服务器报告的提交时间至少 5分钟时，后备机才会重放该事务。

有可能服务器之间的复制延迟会超过这个参数的值，在这种情况下则不会增加延迟。注意延迟是根据主服务器上写 WAL 的时间戳以及后备机上的当前时间来计算。由于网络延

迟或者级联复制配置导致的传输延迟可能会显著地减少实际等待时间。如果主服务器和后备机上的系统时钟不同步，这会导致恢复比预期的更早应用记录。但这不是一个主要问题，因为这个参数有用的设置比服务器之间的典型事件偏差要大得多。

只有在事务提交的 WAL 记录上才会发生延迟。其他记录还是会被尽可能快地重放，这不会成为问题，因为 MVCC 可见性规则确保了在对应的提交记录被应用之前它们的效果不会被看到。

一旦恢复中的数据库已经达到一致状态，延迟就会产生，直到后备机被提升或者触发。在那之后，后备机将会结束恢复并且不再等待。

这个参数的目的是和流复制部署一起使用，但是，如果指定了该参数，所有的情况下都会遵守它。使用这个特性也会让hot_standby_feedback被延迟，这可能导致主服务器的膨胀，两者一起使用时要小心。

> **警告**
>
> 当synchronous_commit被设置为remote_apply时，同步复制会受到这个设置的影响，每一个COMMIT都需要等待被应用。

第 28 章 监控数据库活动

一个数据库管理员常常会疑惑，“系统现在正在做什么？”这一章会讨论如何搞清楚这个问题。

一些工具可以用来监控数据库活动并且分析性能。这一章的大部分都致力于描述PostgreSQL的统计收集器，但是我们也不能忽视常规的 Unix 监控程序，如ps、top、iostat和vmstat。另外，一旦我们发现了一个性能差的查询，可能需要PostgreSQL的EXPLAIN命令来进行进一步的调查。第 14.1 章讨论EXPLAIN以及其他用来理解个体查询行为的方法。

28.1. 标准 Unix 工具

在大部分 Unix 平台上，PostgreSQL会修改由ps报告的命令标题，这样个体服务器进程可以被标识。一个显示样例是

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres:
 background writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
 checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
 walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres:
 autovacuum launcher
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres:
 stats collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe
 runbug 127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] idle in transaction
```

（ps的调用方式随不同的平台而变，但是显示的细节都差不多。这个例子来自于一个最近的Linux 系统）。列在这里的第一个进程是主服务器进程。为它显示的命令参数是当它被启动时使用的那些。接下来的五个进程是由主进程自动启动的后台工作者进程（如果你已经设置系统为不启动统计收集器，“统计收集器”进程将不会出现；同样“自动清理发动”进程也可以被禁用）。剩余的每一个进程都是一个处理一个客户端连接的服务器进程。每个这种进程都会把它的命令行显示设置为这种形式

```
postgres: user database host activity
```

在该客户端连接的生命期中，用户、数据库以及（客户端）主机项保持不变，但是活动指示器会改变。活动可以是闲置（即等待一个客户端命令）、在事务中闲置（在一个BEGIN块里等待客户端）或者一个命令类型名，例如SELECT。还有，如果服务器进程正在等待一个其它会话持有的锁， 等待中会被追加到上述信息中。在上面的例子中，我们可以推断：进程 15606 正在等待进程 15610 完成其事务并且因此释放一些锁（进程 15610 必定是阻塞者，因为没有其他活动会话。在更复杂的情况中，可能需要查看pg_locks系统视图来决定谁阻塞了谁）。

如果配置了cluster_name，则集簇的名字 也将会显示在ps的输出中：

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres:
 server1: background writer
...
```

如果你已经关闭了update_process_title，那么活动指示器将不会被更新，进程标题仅在新进程被启动的时候设置一次。 在某些平台上这样做可以为每个命令节省可观的开销，但在其它平台上却不明显。

> **提示**
>
> Solaris需要特别的处理。你必需使用/usr/ucb/ps而不是/bin/ps。 你还必需使用两个w标志，而不是一个。另外，你对postgres命令的最初调用必须用一个比服务器进程提供的短的ps状态显示。如果你没有满足全部三个要求，每个服务器进程的ps输出将是原始的postgres命令行。 command line.

28.2. 统计收集器

PostgreSQL的统计收集器是一个支持收集和报告服务器活动信息的子系统。 目前，这个收集器可以对表和索引的访问计数，计数可以按磁盘块和个体行来进行。它还跟踪每个表中的总行数、每个表的清理和分析动作的信息。它也统计调用用户定义函数的次数以及在每次调用中花费的总时间。

PostgreSQL也支持报告有关系统正在干什么的 动态信息，例如当前正在被其他服务器进程执行的命令以及系统中存在哪些其他连接。 这个功能是独立于收集器进程存在的。

28.2.1. 统计收集配置

因为统计收集给查询执行增加了一些负荷，系统可以被配置为收集或不收集信息。这由配置参数控制，它们通常在postgresql.conf中设置（关于设置配置参数的细节请见第 19 章。

参数track_activities允许监控当前被任意服务器进程执行的命令。

参数track_counts控制是否收集关于表和索引访问的统计信息。

参数track_functions启用对用户定义函数使用的跟踪。

参数track_io_timing启用对块读写次数的监控。

通常这些参数被设置在postgresql.conf中，这样它们会应用于所有服务器进程，但是可以在单个会话中使用SET命令打开或关闭它们（为了阻止普通用户对管理员隐藏他们的活动，只有超级用户被允许使用SET来改变这些参数）。

统计收集器通过临时文件将收集到的信息传送给其他PostgreSQL进程。这些文件被存储在名字由stats_temp_directory参数指定的目录中，默认是pg_stat_tmp。为了得到更好的性能，stats_temp_directory可以被指向一个基于 RAM 的文件系统来降低物理 I/O 需求。当服务器被干净地关闭时，一份统计数据的永久拷贝被存储在pg_stat子目录中，这样在服务器重启后统计信息能被保持。当在服务器启动时执行恢复时（例如立即关闭、服务器崩溃以及时间点恢复之后），所有统计计数器会被重置。

28.2.2. 查看统计信息

表 28.1中列出了一些预定义视图 可以用来显示系统的当前状态。 表 28.2中列出了另一些视图可以 显示统计收集的结果。你也可以使用底层统计函数（在 第 28.2.3 节讨论）来建立自定义的视图。

在使用统计信息监控收集到的数据时，你必须了解这些信息并非是实时更新的。每个独立的服务器进程只在进入闲置状态之前才向收集器传送新的统计计数；因此正在进行的查询或事务并不影响显示出来的总数。同样，收集器本身也最多每PGSTAT_STAT_INTERVAL毫秒（缺省为 500ms，除非在编译服务器的时候修改过）发送一 次新的报告。因此显示的信息总是落后于实际活动。但是由track_activities收集的当前查询信息总是最新的。

另一个重点是当一个服务器进程被要求显示任何这些统计信息时，它首先取得收集器进程最近发出的报告并且接着为所有统计视图和函数使用这个快照，直到它的当前事务的结尾。因此只要你继续当前事务，统计数据将会一直显示静态信息。相似地，当任何关于所有会话的当前查询的信息在一个事务中第一次被请求时，这样的信息将被收集。并且在整个事务期间将显示相同的信息。这是一种特性而非缺陷，因为它允许你在该统计信息上执行多个查询并且关联结果而不用担心那些数字会在你不知情的情况下改变。但是如果你希望用每个查询都看到新结果，要确保在任何事务块之外做那些查询。或者，你可以调用pg_stat_clear_snapshot()，那将丢弃当前事务的统计快照（如果有）。下一次对统计性信息的使用将导致获取一个新的快照。

一个事务也可以在视图pg_stat_xact_all_tables、pg_stat_xact_sys_tables、pg_stat_xact_user_tables和pg_stat_xact_user_
看到它自己的统计信息（还没有被传送给收集器）。这些数字并不像上面所述的那样行动，相反它们在事务期间持续被更新。

表 28.1. 动态统计视图

视图名称	描述
pg_stat_activity	每个服务器进程一行，显示与那个进程的当前活动相关的信息，例如状态和当前查询。详见pg_stat_activity。
pg_stat_replication	每一个 WAL 发送进程一行，显示有关到该发送进程 连接的后备服务器的复制的统计信息。详见 pg_stat_replication。
pg_stat_wal_receiver	只有一行，显示来自 WAL 接收器所连接服务器的有关该接收器的统计信息。详见pg_stat_wal_receiver。
pg_stat_subscription	每个订阅至少一行，显示有关该订阅的工作者的信息。详见pg_stat_subscription。
pg_stat_ssl	每个连接（常规的或者复制）一行，显示在这个连接上使用的SSL的信息。详见pg_stat_ssl。
pg_stat_progress_vacuum	每个运行着VACUUM的后端（包括autovacuum工作者进程）一行，显示当前的进度。详见第 28.4.1 节

表 28.2. 已收集统计信息的视图

视图名称	描述
pg_stat_archiver	只有一行，显示有关 WAL 归档进程活动的统计信息。详见pg_stat_archiver。
pg_stat_bgwriter	只有一行，显示有关后台写进程的活动的统计信息。详见pg_stat_bgwriter。

视图名称	描述
pg_stat_database	每个数据库一行，显示数据库范围的统计信息。详见pg_stat_database。
pg_stat_database_conflicts	每个数据库一行，显示数据库范围的统计信息， 这些信息的内容是关于由于与后备服务器的恢复过程 发生冲突而被取消的查询。详见 pg_stat_database_conflicts。
pg_stat_all_tables	当前数据库中每个表一行，显示有关访问指定表的统计信息。详见pg_stat_all_tables。
pg_stat_sys_tables	和pg_stat_all_tables一样，但只显示系统表。
pg_stat_user_tables	和pg_stat_all_tables一样，但只显示用户表。
pg_stat_xact_all_tables	和pg_stat_all_tables相似，但计数动作只在当前事务内发生（还没有被包括在pg_stat_all_tables和相关视图中）。用于生存和死亡行数量的列以及清理和分析动作在此视图中不出现。
pg_stat_xact_sys_tables	和pg_stat_xact_all_tables一样，但只显示系统表。
pg_stat_xact_user_tables	和pg_stat_xact_all_tables一样，但只显示用户表。
pg_stat_all_indexes	当前数据库中的每个索引一行，显示：表OID、索引OID、模式名、表名、索引名、 使用了该索引的索引扫描总数、索引扫描返回的索引记录数、使用该索引的简 单索引扫描抓取的活表(livetable)中数据行数。 当前数据库中的每个索引一行，显示与访问指定索引有关的统计信息。详见pg_stat_all_indexes。
pg_stat_sys_indexes	和pg_stat_all_indexes一样，但只显示系统表上的索引。
pg_stat_user_indexes	和pg_stat_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库中每个表一行(包括TOAST表)，显示：表OID、模式名、表名、 从该表中读取的磁盘块总数、缓冲区命中次数、该表上所有索引的磁盘块读取总数、 该表上所有索引的缓冲区命中总数、在该表的辅助TOAST表(如果存在)上的磁盘块读取总数、 在该表的辅助TOAST表(如果存在)上的缓冲区命中总数、TOAST表的索引的磁盘块读 取总数、TOAST表的索引的缓冲区命中总数。 当前数据库中的每个表一行，显示有关在指定表上 I/O 的统计信息。详见pg_statio_all_tables。
pg_statio_sys_tables	和pg_statio_all_tables一样，但只显示系统表。
pg_statio_user_tables	和pg_statio_all_tables一样，但只显示用户表。

视图名称	描述
pg_statio_all_indexes	当前数据库中每个索引一行，显示：表OID、索引OID、模式名、 表名、索引名、该索引的磁盘块读取总数、该索引的缓冲区命中总数。 当前数据库中的每个索引一行，显示与指定索引上的 I/O 有关的统计信息。详见pg_statio_all_indexes。
pg_statio_sys_indexes	和pg_statio_all_indexes一样，但只显示系统表上的索引。
pg_statio_user_indexes	和pg_statio_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库中每个序列对象一行，显示：序列OID、模式名、序列名、序列的磁盘读取总数、序列的缓冲区命中总数。 当前数据库中的每个序列一行，显示与指定序列上的 I/O 有关的统计信息。详见pg_statio_all_sequences。
pg_statio_sys_sequences	和pg_statio_all_sequences一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
pg_statio_user_sequences	和pg_statio_all_sequences一样，但只显示用户序列。
pg_stat_user_functions	对于所有跟踪功能，函数的OID，模式，名称，数量 通话总时间，和自我的时间。自我时间是 在函数本身所花费的时间量，总时间包括 它调用函数所花费的时间。时间值以毫秒为单位。 每一个被跟踪的函数一行，显示与执行该函数有关的统计信息。详见pg_stat_user_functions。
pg_stat_xact_user_functions	和pg_stat_user_functions相似，但是只统计在当前事务期间的调用（还没有被包括在pg_stat_user_functions中）。

针对每个索引的统计信息对于判断哪个索引正被使用以及它们的效果特别有用。

pg_statio_系列视图主要用于判断缓冲区的效果。当实际磁盘读取数远小于缓冲区命中时，这个缓冲能满足大部分读请求而无需进行内核调用。但是，这些统计信息并没有给出所有的事情：由于PostgreSQL处理磁盘 I/O 的方式，不在PostgreSQL缓冲区中的数据库仍然驻留在内核的 I/O 缓存中，并且因此可以被再次读取而不需要物理磁盘读取。我们建议希望了解PostgreSQL I/O 行为更多细节的用户将PostgreSQL统计收集器和操作系统中允许观察内核处理 I/O 的工具一起使用。

表 28.3.pg_stat_activity 视图

列	类型	描述
datid	oid	这个后端连接到的数据库的 OID
datname	name	这个后端连接到的数据库的名称
pid	integer	这个后端的进程 ID
usesysid	oid	登录到这个后端的用户的 OID
usename	name	登录到这个后端的用户的名称

列	类型	描述
application_name	text	连接到这个后端的应用的名称
client_addr	inet	连接到这个后端的客户端的 IP 地址。如果这个域为空，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程（如自动清理）。
client_hostname	text	已连接的客户端的主机名，由client_addr的反向 DNS 查找报告。这个域将只对 IP 连接非空，并且只有log_hostname被启用时才会非空。
client_port	integer	客户端用以和这个后端通信的 TCP 端口号，如果使用 Unix 套接字则为-1
backend_start	timestamp with time zone	这个进程被启动的时间。对客户端后端来说就是客户端连接到服务器的时间。
xact_start	timestamp with time zone	这个进程的当前事务被启动的时间，如果没有活动事务则为空。如果当前查询是它的第一个事务，这一列等于query_start。
query_start	timestamp with time zone	当前活动查询被开始的时间，如果state不是active，这个域为上一个查询被开始的时间
state_change	timestamp with time zone	state上一次被改变的时间
wait_event_type	text	后端正在等待的事件类型，如果不存在则为 NULL。可能的值有： • LWLock：后端正在等待一个轻量级锁。每一个这样的锁保护着共享内存中的一个特殊数据结构。wait_event将含有一个标识该轻量级锁目的的名称（一些锁具有特定的名称，其他是一组具有类似目的的锁中的一部分）。 • Lock：后端正在等待一个重量级锁。重量级锁，也称为锁管理器锁或者简单锁，主要保护 SQL 可见的对象，例如表。不过，它们也被用于确保特定内部操作的互斥，例如关系扩

列	类型	描述
		展。wait_event将标识等待的锁的类型。 • BufferPin：服务器进程正在等待访问一个数据缓冲区，而此时没有其他进程正在检查该缓冲区。如果另一个进程持有一个最终从要访问的缓冲区中读取数据的打开的游标，缓冲区 pin 等待可能会被拖延。 • Activity：服务器进程处于闲置状态。这被用于在其主处理循环中等待活动的系统进程。wait_event将标识特定的等待点。 • Extension：服务器进程正在一个扩展模块中等待活动。这一个分类被用于要跟踪自定义等待点的模块。 • Client：服务器进程正在一个套接字上等待来自用户应用的某种活动，并且该服务器预期某种与其内部处理无关的事情发生。wait_event将标识特定的等待点。 • IPC：服务器进程正在等待来自服务器中另一个进程的某种活动。wait_event将标识特定的等待点。 • Timeout：服务器进程正在等待一次超时发生。wait_event将标识特定的等待点。 • IO：服务器进程正在等待一次IO完成。wait_event将标识特定的等待点。
wait_event	text	如果后端当前正在等待，则是等待事件的名称，否则为NULL。详见表 28.4
state	text	这个后端的当前总体状态。可能的值是： • active：后端正在执行一个查询。

列	类型	描述
		• idle：后端正在等待一个新的客户端命令。 • idle in transaction：后端在一个事务中，但是当前没有正在执行一个查询。 • idle in transaction (aborted)：这个状态与idle in transaction相似，不过在该事务中的一个语句导致了一个错误。 • fastpath function call：后端正在执行一个 fast-path 函数。 • disabled：如果在这个后端中track_activities被禁用，则报告这个状态。
backend_xid	xid	这个后端的顶层事务标识符（如果存在）。
backend_xmin	xid	当前后端的xmin范围。
query	text	这个后端最近查询的文本。如果state为active，这个域显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。默认情况下，查询文本会被截断至1024个字符，这个值可以通过参数track_activity_query_size更改。
backend_type	text	当前后端的类型。可能的类型是 autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpointer, startup, walreceiver, walsender 以及 walwriter。 除此以外，由扩展注册的后台Worker可能有额外的类型。

pg_stat_activity视图将为每一个服务器进程有一行，显示与该进程的当前活动相关的信息。

> **注意**
>
> wait_event和state列是独立的。如果一个后端处于active状态，它可能是也可能不是某个事件上的waiting。如果状态是active并且wait_event为非空，它意味着一个查询正在被执行，但是它被阻塞在系统中某处。

表 28.4.wait_event 描述

等待事件类型	等待事件名称	描述
LWLock	ShmemIndexLock	正等待在共享内存中查找或者分配空间。
	OidGenLock	正等待分配或者赋予一个 OID。
	XidGenLock	正等待分配或者赋予一个事务 ID。
	ProcArrayLock	正等待在事务结尾得到一个快照或者清除事务 ID。
	SInvalReadLock	正等待从共享无效消息队列中检索或者移除消息。
	SInvalWriteLock	正等待在共享无效消息队列中增加一个消息。
	WALBufMappingLock	正等待在 WAL 缓冲区中替换一个页面。
	WALWriteLock	正等待 WAL 缓冲区被写入到磁盘。
	ControlFileLock	正等待读取或者更新控制文件或创建一个新的 WAL 文件。
	CheckpointLock	正等待执行检查点。
	CLogControlLock	正等待读取或者更新事务状态。
	SubtransControlLock	正等待读取或者更新子事务信息。
	MultiXactGenLock	正等待读取或者更新共享多事务状态。
	MultiXactOffsetControlLock	正等待读取或者更新多事务偏移映射。
	MultiXactMemberControlLock	正等待读取或者更新多事务成员映射。
	RelCacheInitLock	正等待读取或者写入关系缓冲区初始化文件。
	CheckpointerCommLock	正等待管理 fsync 请求。
	TwoPhaseStateLock	正等待读取或者更新预备事务的状态。
	TablespaceCreateLock	正等待创建或者删除表空间。

等待事件类型	等待事件名称	描述
	BtreeVacuumLock	正等待读取或者更新一个 B-树索引的 vacuum 相关的信息。
	AddinShmemInitLock	正等待管理共享内存中的空间分配。
	AutovacuumLock	自动清理工作者或者启动器正等待更新或者读取自动清理工作者的当前状态。
	AutovacuumScheduleLock	正等待确认选中进行清理的表仍需要清理。
	SyncScanLock	正等待为同步扫描得到一个表上扫描的开始位置。
	RelationMappingLock	正等待更新用来存储目录到文件节点映射的关系映射文件。
	AsyncCtlLock	正等待读取或者更新共享通知状态。
	AsyncQueueLock	正等待读取或者更新通知消息。
	SerializableXactHashLock	正等待检索或者存储有关可序列化事务的信息。
	SerializableFinishedListLock	正等待访问已结束可序列化事务的列表。
	SerializablePredicateLockListLock	正等待在由可序列化事务持有的所列表上执行一个操作。
	OldSerXidLock	正等待读取或者记录冲突的可序列化事务。
	SyncRepLock	正等待读取或者更新有关同步复制的信息。
	BackgroundWorkerLock	正等待读取或者更新后台工作者状态。
	DynamicSharedMemoryControlLock	正等待读取或者更新动态共享内存状态。
	AutoFileLock	正等待更新postgresql.auto.conf文件。
	ReplicationSlotAllocationLock	正等待分配或者释放一个复制槽。
	ReplicationSlotControlLock	正等待读取或者更新复制槽状态。
	CommitTsControlLock	正等待读取或者更新事务提交时间戳。
	CommitTsLock	正等待读取或者更新事务时间戳的最新设置值。
	ReplicationOriginLock	正等待设置、删除或者使用复制源头。

等待事件类型	等待事件名称	描述
	MultiXactTruncationLock	正等待读取或者阶段多事务信息。
	OldSnapshotTimeMapLock	正等待读取或者更新旧的快照控制信息。
	BackendRandomLock	正等待产生一个随机数。
	LogicalRepWorkerLock	正等待逻辑复制工作者上的动作完成。
	CLogTruncationLock	正等待截断预写式日志或者等待预写式日志截断操作完成。
	clog	正在等地clog （事务状态)缓冲区上的I/O。
	commit_timestamp	正等待提交时间戳缓冲区上的 I/O。
	subtrans	正等待子事务缓冲区上的 I/O。
	multixact_offset	正等待多事务偏移缓冲区上的 I/O。
	multixact_member	正等待多事务成员缓冲区上的 I/O。
	async	正等待 async（通知）缓冲区上的 I/O。
	oldserxid	正等待 oldserxid 缓冲区上的 I/O。
	wal_insert	正等待把 WAL 插入到一个内存缓冲区。
	buffer_content	正等待读取或者写入内存中的一个数据页。
	buffer_io	正等待一个数据页面上的 I/O。
	replication_origin	正等待读取或者更新复制进度。
	replication_slot_io	正等待一个复制槽上的 I/O。
	proc	正等待读取或者更新 fast-path 锁信息。
	buffer_mapping	正等待把一个数据块与缓冲池中的一个缓冲区关联。
	lock_manager	正等待增加或者检查用于后端的锁，或者正等待加入或者退出一个锁定组（并行查询使用）。
	predicate_lock_manager	正等待增加或者检查谓词锁信息。
	parallel_query_dsa	正等待并行查询动态共享内存分配锁。
	tbm	正等待TBM共享迭代器锁。

等待事件类型	等待事件名称	描述
	parallel_append	在Parallel Append计划执行期间等待选择下一个子计划。
	parallel_hash_join	在Parallel Hash计划执行期间等待分配或交换一块内存或者更新计数器。
Lock	relation	正等待获得一个关系上的锁。
	extend	正等待扩展一个关系。
	page	正等待获得一个关系上的页面的锁。
	tuple	正等待获得一个元组上的锁。
	transactionid	正等待一个事务结束。
	virtualxid	正等待获得一个虚拟 xid 锁。
	speculative token	正等待获取一个 speculative insertion lock。
	object	正等待获得一个非关系数据库对象上的锁。
	userlock	正等待获得一个用户锁。
	advisory	正等待获得一个咨询用户锁。
BufferPin	BufferPin	正等待在一个缓冲区上加pin。
Activity	ArchiverMain	正在归档进程的主循环中等待。
	AutoVacuumMain	正在autovacuum启动器进程的主循环中等待。
	BgWriterHibernate	正在后台写入器进程中等待，休眠中。
	BgWriterMain	正在后台写入器进程的后台工作者的主循环中等待。
	CheckpointerMain	正在检查点进程的主循环中等待。
	LogicalApplyMain	正在逻辑应用进程的主循环中等待。
	LogicalLauncherMain	正在逻辑启动器进程的主循环中等待。
	PgStatMain	正在统计收集器进程的主循环中等待。
	RecoveryWalAll	在恢复时等待来自于任意类型来源（本地、归档或流）的WAL。
	RecoveryWalStream	在恢复时等待来自于一个流的WAL。

等待事件类型	等待事件名称	描述
	SysLoggerMain	正在系统日志进程的主循环中等待。
	WalReceiverMain	正在WAL接收器进程的主循环中等待。
	WalSenderMain	正在WAL发送器进程的主循环中等待。
	WalWriterMain	正在WAL写入器进程的主循环中等待。
Client	ClientRead	正等待从客户端读取数据。
	ClientWrite	正等待向客户端写入数据。
	LibPQWalReceiverConnect	正在WAL接收器中等待建立与远程服务器的连接。
	LibPQWalReceiverReceive	正在WAL接收器中等待从远程服务器接收数据。
	SSLOpenServer	正在尝试连接期间等待SSL。
	WalReceiverWaitStart	正等待startup进程发送流复制的初始数据。
	WalSenderWaitForWAL	正在WAL发送器进程中等待WAL被刷写。
	WalSenderWriteData	在WAL发送器进程中处理来自WAL接收器的回复时等待任意活动。
Extension	Extension	正在一个扩展中等待。
IPC	BgWorkerShutdown	正等待后台工作者关闭。
	BgWorkerStartup	正等待后台工作者启动。
	BtreePage	正等待继续并行B-树扫描所需的页号变得可用。
	ClogGroupUpdate	正等待组领袖在事务结束时更新事务状态。
	ExecuteGather	在执行Gather节点时等待来自子进程的活动。
	Hash/Batch/Allocating	正等待一个选出的Parallel Hash参与者分配哈希表。
	Hash/Batch/Electing	正在选出一个Parallel Hash参与者来分配一个哈希表。
	Hash/Batch/Loading	正等待其他Parallel Hash参与者完成装载哈希表。
	Hash/Build/Allocating	正等待一个选出的Parallel Hash参与者分配初始哈希表。
	Hash/Build/Electing	正在选出一个Parallel Hash参与者以分配初始哈希表。
	Hash/Build/HashingInner	正等待其他Parallel Hash参与者完成对内关系的哈希操作。

等待事件类型	等待事件名称	描述
	Hash/Build/HashingOuter	正等待其他Parallel Hash参与者完成对外关系的哈希操作。
	Hash/GrowBatches/Allocating	正等待一个选出的Parallel Hash参与者分配更多批次。
	Hash/GrowBatches/Deciding	正在选出一个Parallel Hash参与者决定未来的批次增长。
	Hash/GrowBatches/Electing	正在选出一个Parallel Hash参与者分配更多批次。
	Hash/GrowBatches/Finishing	正在等待一个选出的Parallel Hash参与者决定未来的批次增长。
	Hash/GrowBatches/Repartitioning	正等待其他Parallel Hash参与者完成重新分区。
	Hash/GrowBuckets/Allocating	正等待一个选出的Parallel Hash参与者完成更多桶的分配。
	Hash/GrowBuckets/Electing	正在选出一个Parallel Hash参与者分配更多桶。
	Hash/GrowBuckets/Reinserting	正等待其他Parallel Hash参与者完成将元组插入到新桶的操作。
	LogicalSyncData	正等待逻辑复制的远程服务器发送用于初始表同步的数据。
	LogicalSyncStateChange	正等待逻辑复制的远程服务器更改状态。
	MessageQueueInternal	正等待其他进程被挂接到共享消息队列。
	MessageQueuePutMessage	正等待把一个协议消息写到一个共享消息队列。
	MessageQueueReceive	正等待从一个共享消息队列接收字节。
	MessageQueueSend	正等待向一个共享消息队列中发送字节。
	ParallelBitmapScan	正等待并行位图扫描被初始化。
	ParallelCreateIndexScan	正等待并行CREATE INDEX工作者完成堆扫描。
	ParallelFinish	正等待并行工作者完成计算。
	ProcArrayGroupUpdate	正等待组领袖在事务结束时清除事务ID。
	ReplicationOriginDrop	正等待一个复制源头变得不活跃以便被删除。
	ReplicationSlotDrop	正等待一个复制槽变得不活跃以便被删除。

等待事件类型	等待事件名称	描述
	SafeSnapshot	正等待一个用于READ ONLY DEFERRABLE事务的快照。
	SyncRep	正在同步复制期间等待来自远程服务器的确认。
Timeout	BaseBackupThrottle	当有限流活动时在基础备份期间等待。
	PgSleep	正在调用pg_sleep的进程中等待。
	RecoveryApplyDelay	在恢复时等待应用WAL，因为它被延迟了。
IO	BufFileRead	正等待从一个缓存的文件中读取。
	BufFileWrite	正等待向一个缓存的文件中写入。
	ControlFileRead	正等待从控制文件中读取。
	ControlFileSync	正等待控制文件到达稳定存储。
	ControlFileSyncUpdate	正等待对控制文件的更新到达稳定存储。
	ControlFileWrite	正等待一个对控制文件的写入。
	ControlFileWriteUpdate	正等待一个写操作更新控制文件。
	CopyFileRead	正在文件拷贝操作期间等待一个读操作。
	CopyFileWrite	正在文件拷贝操作期间等待一个写操作。
	DataFileExtend	正等待一个关系数据文件被扩充。
	DataFileFlush	正等待一个关系数据文件到达稳定存储。
	DataFileImmediateSync	正等待一个关系数据文件的立即同步到达稳定存储。
	DataFilePrefetch	正等待从一个关系数据文件中的一次异步预取。
	DataFileRead	正等待一次对一个关系数据文件的读操作。
	DataFileSync	正等待对一个关系数据文件的更改到达稳定存储。
	DataFileTruncate	正等待一个关系数据文件被截断。
	DataFileWrite	正等待一次对一个关系数据文件的写操作。
	DSMFillZeroWrite	等待向一个动态共享内存备份文件中写零字节。
	LockFileAddToDataDirRead	在向数据目录锁文件中增加一行时等待一个读操作。

等待事件类型	等待事件名称	描述
	LockFileAddToDataDirSync	在向数据目录锁文件中增加一行时等待数据到达稳定存储。
	LockFileAddToDataDirWrite	在向数据目录锁文件中增加一行时等待一个写操作。
	LockFileCreateRead	在创建数据目录锁文件期间等待读取。
	LockFileCreateSync	在创建数据目录锁文件期间等待数据到达稳定存储。
	LockFileCreateWrite	在创建数据目录锁文件期间等待一个写操作。
	LockFileReCheckDataDirRead	在重新检查数据目录锁文件的过程中等待一个读操作。
	LogicalRewriteCheckpointSync	在一个检查点期间等待逻辑重写映射到达稳定存储。
	LogicalRewriteMappingSync	在一次逻辑重写期间等待映射数据到达稳定存储。
	LogicalRewriteMappingWrite	在一次逻辑重写期间等待对映射数据的写操作。
	LogicalRewriteSync	正等待逻辑重写映射到达稳定存储。
	LogicalRewriteWrite	正等待对逻辑重写映射的写操作。
	RelationMapRead	正等待对关系映射文件的读操作。
	RelationMapSync	正等待关系映射文件到达稳定存储。
	RelationMapWrite	正等待对关系映射文件的写操作。
	ReorderBufferRead	在重排序缓冲区管理期间等待一个读操作。
	ReorderBufferWrite	在重排序缓冲区管理期间等待一个写操作。
	ReorderLogicalMappingRead	在重排序缓冲区管理期间等待对一个逻辑映射的读操作。
	ReplicationSlotRead	正等待对一个复制槽控制文件的读操作。
	ReplicationSlotRestoreSync	在把一个复制槽控制文件恢复到内存的过程中等待它到达稳定存储。
	ReplicationSlotSync	正等待一个复制槽控制文件到达稳定存储。
	ReplicationSlotWrite	正等待对一个复制槽控制文件的写操作。
	SLRUFlushSync	在检查点或者数据库关闭期间等待SLRU数据到达稳定存储。

等待事件类型	等待事件名称	描述
	SLRURead	正等待对一个SLRU页面的读操作。
	SLRUSync	正等待SLRU数据在一个页面写之后到达稳定存储。
	SLRUWrite	正等待一个SLRU页面上的写操作。
	SnapbuildRead	正等待一个序列化历史目录快照的读操作。
	SnapbuildSync	正等待一个序列化历史目录快照到达稳定存储。
	SnapbuildWrite	正等待一个序列化历史目录快照的写操作。
	TimelineHistoryFileSync	正等待一个通过流复制接收到的时间线历史文件到达稳定存储。
	TimelineHistoryFileWrite	正等待一个通过流复制接收到的时间线历史文件的读操作。
	TimelineHistoryRead	正等待一个时间线历史文件上的读操作。
	TimelineHistorySync	正等待一个新创建的时间线历史文件达到稳定存储。
	TimelineHistoryWrite	正等待一个新创建的时间线历史文件上的写操作。
	TwophaseFileRead	正等待一个两阶段状态文件的读操作。
	TwophaseFileSync	正等待一个两阶段状态文件到达稳定存储。
	TwophaseFileWrite	正等待一个两阶段状态文件的写操作。
	WALBootstrapSync	在自举期间等待WAL到达稳定存储。
	WALBootstrapWrite	在自举期间等待一个WAL页面的写操作。
	WALCopyRead	在通过拷贝一个已有WAL段创建一个新的WAL段时等待一个读操作。
	WALCopySync	正等待一个通过拷贝已有WAL段创建的新WAL段到达稳定存储。
	WALCopyWrite	在通过拷贝一个已有WAL段创建一个新的WAL段时等待一个写操作。
	WALInitSync	正等待一个新初始化的WAL文件到达稳定存储。
	WALInitWrite	在初始化一个新的WAL文件期间等待一个写操作。

等待事件类型	等待事件名称	描述
	WALRead	正等待一次对一个WAL文件的读操作。
	WALSenderTimelineHistoryRead	在walsender的时间线命令期间等待对一个时间线历史文件的读操作。
	WALSyncMethodAssign	在指派WAL同步方法时等待数据到达稳定存储。
	WALWrite	正等待一次对一个WAL文件的写操作。

> **注意**
>
> 对于扩展安装的切片（tranche），这个名称由扩展指定并且会被wait_event显示出来。很有可能在其他后端不知道的情况下，用户在其中一个后端中注册了切片（通过在动态共享内存中分配），那么我们对这种情况会显示extension。

下面的例子展示了如何查看等待事件

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event
 is NOT NULL;
 pid  | wait_event_type |  wait_event
------+-----------------+--------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArrayLock
(2 rows)
```

表 28.5.pg_stat_replication 视图

列	类型	描述
pid	integer	一个 WAL 发送进程的进程 ID
usesysid	oid	登录到这个 WAL 发送进程的用户的 OID
usename	name	登录到这个 WAL 发送进程的用户的名称
application_name	text	连接到这个 WAL 发送进程的应用的名称
client_addr	inet	连接到这个 WAL 发送进程的客户端的 IP 地址。 如果这个域为空，它表示该客户端通过服务器机器上的一个 Unix 套接字连接。
client_hostname	text	连接上的客户端的主机名，由一次对client_addr 的逆向 DNS 查找报告。这个域将只对 IP 连接非空，并且只有在log_hostname被启用时非空
client_port	integer	客户端用来与这个 WAL 发送进程通讯的 TCP 端口号， 如果使用 Unix 套接字则为-1

列	类型	描述
backend_start	timestamp with time zone	这个进程开始的时间，即客户端是何时连接到这个 WAL 发送进程的
backend_xmin	xid	由hot_standby_feedback报告 的这个后备机的xmin水平线。
state	text	当前的 WAL 发送进程状态。可能的值是： • startup：这个WAL发送器正在启动。 • catchup：这个WAL发送器连接的后备机正在追赶主服务器。 • streaming：这个WAL发送器在它连接的后备服务器追上主服务器之后用流传送更改。 • backup：这个WAL发送器正在发送一个备份。 • stopping：这个WAL发送器正在停止。
sent_lsn	pg_lsn	在这个连接上发送的最后一个预写式日志的位置
write_lsn	pg_lsn	被这个后备服务器写入到磁盘的最后一个预写式日志的位置
flush_lsn	pg_lsn	被这个后备服务器刷入到磁盘的最后一个预写式日志的位置
replay_lsn	pg_lsn	被重放到这个后备服务器上的数据库中的最后一个预写式日志的位置
write_lag	interval	在本地刷写近期的WAL与接收到后备服务器已经写入它（但还没有刷写或者应用）的通知之间流逝的时间。如果这台服务器被配置为一个同步后备，这可以用来计量在提交时synchronous_commit的级别remote_write所导致的延迟。
flush_lag	interval	在本地刷写近期的WAL与接收到后备服务器已经写入并且刷写它（但还没有应用）的通知之间流逝的时间。如果这台服务器被配置为一个同步后备，这可以用来计量在提交

列	类型	描述
		时synchronous_commit的级别on所导致的延迟。
replay_lag	interval	在本地刷写近期的WAL与接收到后备服务器已经写入它、刷写它并且应用它的通知之间流逝的时间。如果这台服务器被配置为一个同步后备，这可以用来计量在提交时synchronous_commit的级别remote_apply所导致的延迟。
sync_priority	integer	在基于优先的同步复制中，这台后备服务器被选为同步后备的优先级。在基于规定数量的同步复制中，这个值没有效果。
sync_state	text	这一台后备服务器的同步状态。 可能的值是： • async：这台后备服务器是异步的。 • potential：这台后备服务器现在是异步的，但可能在当前的同步后备失效时变成同步的。 • sync：这台后备服务器是同步的。 • quorum：这台后备服务器被当做规定数量后备服务器的候选。

pg_stat_replication视图中将为每一个 WAL 发送进程包含一行，用来显示与该发送进程连接的后备服务器的复制统计信息。 这个视图中只会列出直接连接的后备机，下游后备服务器的信息不包含在此。

pg_stat_replication视图中报告的滞后时间近期的WAL被写入、刷写并且重放以及发送器知道这一切所花的时间的度量。如果远程服务器被配置为一台同步后备，这些时间表示由每一种同步提交级别所带来（或者是可能带来）的提交延迟。对于一台异步后备，replay_lag列是最近的事务变得对查询可见的延迟时间的近似值。如果后备服务器已经完全追上了发送服务器并且没有WAL活动，在短时间内将继续显示最近测到的滞后时间，再然后就会显示为NULL。

对于物理复制会自动测量滞后时间。逻辑解码插件可能会选择性地发出跟踪消息，如果它们没有这样做，跟踪机制将把滞后显示为NULL。

> **注意**
>
> 报告的滞后时间并非按照当前的重放速率该后备还有多久才能追上发送服务器的预测。在新的WAL被生成期间，这样一种系统将显示类似的时间，但是当发送器变为闲置时会显示不同的值。特别是当后备服务器完全追上时，pg_stat_replication显示的是写入、刷写及重放最近报告的WAL位置所花的时间而不是一些用户可能预期的零。这种做法与为近期的写事务测量同步提交和事务可见性延迟的目的一致。为了降低用户预期一种不同的滞后模型带来

的混淆，在一个完全重放完的闲置系统上，lag列会在一段比较短的时间后回复成NULL。监控系统应该选择将这种情况表示为缺失数据、零或者继续显示最近的已知值。

表 28.6.pg_stat_wal_receiver 视图

列	类型	描述
pid	integer	WAL 接收器进程的进程 ID
status	text	WAL 接收器进程的活动状态
receive_start_lsn	pg_lsn	WAL 接收器启动时使用的第一个预写式日志位置
receive_start_tli	integer	WAL 接收器启动时使用的第一个时间线编号
received_lsn	pg_lsn	已经接收到并且已经被杀入磁盘的最后一个预写式日志的位置，这个域的初始值是 WAL 接收器启动时使用的第一个日志位置
received_tli	integer	已经接收到并且已经被杀入磁盘的最后一个预写式日志的时间线编号，这个域的初始值是 WAL 接收器启动时使用的第一个日志所在的时间线编号
last_msg_send_time	timestamp with time zone	从源头 WAL 发送器接收到的最后一个消息的发送时间
last_msg_receipt_time	timestamp with time zone	从源头 WAL 发送器接收到的最后一个消息的接收时间
latest_end_lsn	pg_lsn	报告给源头 WAL 发送器的最后一个预写式日志位置
latest_end_time	timestamp with time zone	报告给源头 WAL 发送器最后一个事务日志位置的时间
slot_name	text	这个 WAL 接收器使用的复制槽的名称
sender_host	text	这个WAL接收器连接到的PostgreSQL实例的主机。这可以是一个主机名、一个IP地址，如果连接是通过Unix套接字则是一个目录路径（为目录的情况可以被辨别出来，因为路径将总是一个绝对路径并且以/开头）。
sender_port	integer	这个WAL接收器连接到的PostgreSQL实例的端口号。
conninfo	text	这个 WAL 接收器使用的连接串，安全相关的域会被隐去。

pg_stat_wal_receiver事务只包含一行，它显示了从 WAL 接收器所连接的服务器得到的有关该接收器的统计信息。

表 28.7.pg_stat_subscription视图

列	类型	介绍
subid	oid	订阅的OID
subname	text	订阅的名称
pid	integer	订阅工作者进程的进程ID
relid	Oid	工作者正在同步的关系的OID，对于主应用工作者为空
received_lsn	pg_lsn	接收到的最后一个预写式日志位置，这个字段的初始值是0
last_msg_send_time	timestamp with time zone	从源头WAL发送器接收到的最后一个消息的发送时间
last_msg_receipt_time	timestamp with time zone	从源头WAL发送器接收到的最后一个消息的接收时间
latest_end_lsn	pg_lsn	最后一个报告给源头WAL发送器的预写式日志位置
latest_end_time	timestamp with time zone	报告给源头WAL发送器的最后一个预写式日志位置的时间

每一个订阅的主工作者都在pg_stat_subscription视图中有一行（如果工作者没有运行则PID为空），处理被订阅表的初始数据拷贝操作的工作者还会有额外的行。

表 28.8.pg_stat_ssl视图

列	类型	描述
pid	integer	一个后端或者 WAL 发送进程的进程 ID
ssl	boolean	如果在这个连接上使用了 SSL 则为真
version	text	在用的 SSL 版本，如果这个连接上没有使用 SSL 则为 NULL
cipher	text	在用的 SSL 密码的名称，如果这个连接上没有使用 SSL 则为 NULL
bits	integer	使用的加密算法中的位数，如果这个连接上没有使用 SSL 则为 NULL
compression	boolean	如果使用了 SSL 压缩则为真，否则为假， 如果这个连接上没有使用 SSL 则为 NULL
clientdn	text	来自所使用的客户端证书的识别名（DN）域， 如果没有提供客户端证书或者这个连接上没有使用 SSL 则为 NULL。如果 DN 域长度超过 NAMEDATALEN（标准编译 中是 64 个字符），则它会被截断。

pg_stat_ssl视图将为每一个后端或者 WAL 发送进程 包含一行，用来显示这个连接上的 SSL 使用情况。可以把它与 pg_stat_activity或者 pg_stat_replication通过 pid列连接来得到更多有关该连接的细节。

表 28.9.pg_stat_archiver视图

列	类型	描述
archived_count	bigint	已被成功归档的 WAL 文件数量
last_archived_wal	text	最后一个被成功归档的 WAL 文件名称
last_archived_time	timestamp with time zone	最后一次成功归档操作的时间
failed_count	bigint	失败的归档 WAL 文件尝试的数量
last_failed_wal	text	最后一次失败的归档操作的 WAL 文件名称
last_failed_time	timestamp with time zone	最后一次失败的归档操作的时间
stats_reset	timestamp with time zone	这些统计信息最后一次被重置的时间

The pg_stat_archiver视图将总是一个单一的行， 该行包含着有关集簇的归档进程的数据。

表 28.10.pg_stat_bgwriter视图

列	类型	描述
checkpoints_timed	bigint	已经被执行的计划中检查点的数量
checkpoints_req	bigint	已经被执行的请求检查点的数量
checkpoint_write_time	double precision	在文件被写入磁盘的检查点处理部分花费的总时间，以毫秒计
checkpoint_sync_time	double precision	在文件被同步到磁盘中的检查点处理部分花费的总时间，以毫秒计
buffers_checkpoint	bigint	在检查点期间被写的缓冲区数目
buffers_clean	bigint	被后台写进程写的缓冲区数目
maxwritten_clean	bigint	后台写进程由于已经写了太多缓冲区而停止清洁扫描的次数
buffers_backend	bigint	被一个后端直接写的缓冲区数量
buffers_backend_fsync	bigint	一个后端不得不自己执行fsync调用的次数（通常即使后端自己进行写操作，后台写进程也会处理这些）
buffers_alloc	bigint	被分配的缓冲区数量

列	类型	描述
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间

pg_stat_bgwriter视图将总是只有单独的一行，它包含集簇的全局数据。

表 28.11.pg_stat_database视图

列	类型	描述
datid	oid	一个数据库的 OID
datname	name	这个数据库的名称
numbackends	integer	当前连接到这个数据库的后端数量。这是在这个视图中唯一一个返回反映当前状态值的列。所有其他列返回从上次重置以来积累的值。
xact_commit	bigint	在这个数据库中已经被提交的事务的数量
xact_rollback	bigint	在这个数据库中已经被回滚的事务的数量
blks_read	bigint	在这个数据库中被读取的磁盘块的数量
blks_hit	bigint	磁盘块被发现已经在缓冲区中的次数，这样不需要一次读取（这只包括 PostgreSQL 缓冲区中的命中，而不包括在操作系统文件系统缓冲区中的命中）
tup_returned	bigint	在这个数据库中被查询返回的行数
tup_fetched	bigint	在这个数据库中被查询取出的行数
tup_inserted	bigint	在这个数据库中被查询插入的行数
tup_updated	bigint	在这个数据库中被查询更新的行数
tup_deleted	bigint	在这个数据库中被查询删除的行数
conflicts	bigint	由于与恢复冲突而在这个数据库中被取消的查询的数目（冲突只发生在后备服务器上，详见pg_stat_database_conflicts）。
temp_files	bigint	在这个数据库中被查询创建的临时文件的数量。所有临时文件都被统计，不管为什么创建这些临时文件（如排序或哈希），并且不管log_temp_files设置。
temp_bytes	bigint	在这个数据库中被查询写到临时文件中的数据总量。所有临时文件都被统计，不管

列	类型	描述
		为什么创建这些临时文件（如排序或哈希），并且不管log_temp_files设置。
deadlocks	bigint	在这个数据库中被检测到的死锁数
blk_read_time	double precision	在这个数据库中后端花费在读取数据文件块的时间，以毫秒计
blk_write_time	double precision	在这个数据库中后端花费在写数据文件块的时间，以毫秒计
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间

pg_stat_database视图将为集簇中的每一个数据库包含有一行，每一行显示数据库范围的统计信息。

表 28.12.pg_stat_database_conflicts视图

列	类型	描述
datid	oid	一个数据库的 OID
datname	name	这个数据库的名称
confl_tablespace	bigint	这个数据库中由于表空间被删掉而取消的查询数量
confl_lock	bigint	这个数据库中由于锁超时而取消的查询数量
confl_snapshot	bigint	这个数据库中由于旧快照而取消的查询数量
confl_bufferpin	bigint	这个数据库中由于被占用的缓冲区而取消的查询数量
confl_deadlock	bigint	这个数据库中由于死锁而取消的查询数量

pg_stat_database_conflicts视图为每一个 数据库包含一行，用来显示数据库范围内由于与后备服务器上的恢复过程 冲突而被取消的查询的统计信息。 这个视图将只包含后备服务器上的信息， 因为冲突会不发生在主服务器上。

表 28.13.pg_stat_all_tables视图

列	类型	描述
relid	oid	一个表的 OID
schemaname	name	这个表所在的模式的名称
relname	name	这个表的名称
seq_scan	bigint	在这个表上发起的顺序扫描的次数
seq_tup_read	bigint	被顺序扫描取得的活着的行的数量
idx_scan	bigint	在这个表上发起的索引扫描的次数

列	类型	描述
idx_tup_fetch	bigint	被索引扫描取得的活着的行的数量
n_tup_ins	bigint	被插入的行数
n_tup_upd	bigint	被更新的行数（包括 HOT 更新的行）
n_tup_del	bigint	被删除的行数
n_tup_hot_upd	bigint	被更新的 HOT 行数（即不要求独立索引更新的行更新）
n_live_tup	bigint	活着的行的估计数量
n_dead_tup	bigint	死亡行的估计数量
n_mod_since_analyze	bigint	从这个表最后一次被分析后备修改的行的估计数量
last_vacuum	timestamp with time zone	上次这个表被手动清理的时间（不统计VACUUM FULL）
last_autovacuum	timestamp with time zone	上次这个表被自动清理守护进程清理的时间
last_analyze	timestamp with time zone	上次这个表被手动分析的时间
last_autoanalyze	timestamp with time zone	上次这个表被自动清理守护进程分析的时间
vacuum_count	bigint	这个表已被手工清理的次数（不统计VACUUM FULL）
autovacuum_count	bigint	这个表已被自动清理守护进程清理的次数
analyze_count	bigint	这个表已被手工分析的次数
autoanalyze_count	bigint	这个表已被自动清理守护进程分析的次数

pg_stat_all_tables视图将为当前数据库中的每一个表（包括 TOAST 表）包含一行，该行显示与对该表的访问相关的统计信息。pg_stat_user_tables和pg_stat_sys_tables视图包含相同的信息，但是被过滤得分别只显示用户和系统表。

表 28.14.pg_stat_all_indexes视图

列	类型	描述
relid	oid	这个索引的基表的 OID
indexrelid	oid	这个索引的 OID
schemaname	name	这个索引所在的模式的名称
relname	name	这个索引的基表的名称
indexrelname	name	这个索引的名称
idx_scan	bigint	在这个索引上发起的索引扫描次数
idx_tup_read	bigint	在这个索引上由扫描返回的索引项数量
idx_tup_fetch	bigint	被使用这个索引的简单索引扫描取得的活着的表行数量

pg_stat_all_indexes视图将为当前数据库中的每个索引包含一行，该行显示关于对该索引访问的统计信息。pg_stat_user_indexes和pg_stat_sys_indexes视图包含相同的信息，但是被过滤得只分别显示用户和系统索引。

索引可以被简单索引扫描、“位图”索引扫描以及优化器使用。在一次位图扫描中，多个索引的输出可以被通过 AND 或 OR 规则组合，因此当使用一次位图扫描时难以将取得的个体堆行与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引
的pg_stat_all_indexes.idx_tup_read计数，并且为每个表增
加pg_stat_all_tables.idx_tup_fetch计数，但是它不影
响pg_stat_all_indexes.idx_tup_fetch。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查，因为优化器统计信息可能已经“不新鲜”了。

> **注意**
>
> 即使不用位图扫描，idx_tup_read和idx_tup_fetch计数也可能不同，因为idx_tup_read统计从该索引取得的索引项而idx_tup_fetch统计从表取得的或者的行。如果使用该索引取得了任何死亡行或还未提交的行，或者如果通过一次只用索引扫描的方式避免了任何堆获取，后者将较小。

表 28.15.pg_statio_all_tables视图

列	类型	描述
relid	oid	一个表的 OID
schemaname	name	这个表所在的模式的名称
relname	name	这个表的名称
heap_blks_read	bigint	从这个表读取的磁盘块数量
heap_blks_hit	bigint	在这个表中的缓冲区命中数量
idx_blks_read	bigint	从这个表上所有索引中读取的磁盘块数
idx_blks_hit	bigint	在这个表上的所有索引中的缓冲区命中数量
toast_blks_read	bigint	从这个表的 TOAST 表（如果有）读取的磁盘块数
toast_blks_hit	bigint	在这个表的 TOAST 表（如果有）中的缓冲区命中数量
tidx_blks_read	bigint	从这个表的 TOAST 表索引（如果有）中读取的磁盘块数
tidx_blks_hit	bigint	在这个表的 TOAST 表索引（如果有）中的缓冲区命中数量

pg_statio_all_tables视图将为当前数据库中的每个表（包括 TOAST 表）包含一行，该行显示指定表上有关 I/O 的统计信息。pg_statio_user_tables和pg_statio_sys_tables视图包含相同的信息，但是被过滤得分别只显示用户表和系统表。

表 28.16.pg_statio_all_indexes视图

列	类型	描述
relid	oid	这个索引的基表的 OID
indexrelid	oid	这个索引的 OID

列	类型	描述
schemaname	name	这个索引所在的模式的名称
relname	name	这个索引的基表的名称
indexrelname	name	这个索引的名称
idx_blks_read	bigint	从这个索引读取的磁盘块数
idx_blks_hit	bigint	在这个索引中的缓冲区命中数量

pg_statio_all_indexes视图将为当前数据库中的每个索引包含一行，该行显示指定索引上有关 I/O 的统计信息。pg_statio_user_indexes和pg_statio_sys_indexes视图包含相同的信息，但是被过滤得分别只显示用户索引和系统索引。

表 28.17.pg_statio_all_sequences视图

列	类型	描述
relid	oid	一个序列的 OID
schemaname	name	这个序列所在的模式的名称
relname	name	这个序列的名称
blks_read	bigint	从这个序列中读取的磁盘块数
blks_hit	bigint	在这个序列中的缓冲区命中数量

pg_statio_all_sequences视图将为当前数据库中的每个序列包含一行，该行显示在指定序列上有关 I/O 的统计信息。

表 28.18.pg_stat_user_functions视图

列	类型	描述
funcid	oid	一个函数的 OID
schemaname	name	这个函数所在的模式的名称
funcname	name	这个函数的名称
calls	bigint	这个函数已经被调用的次数
total_time	double precision	在这个函数以及它所调用的其他函数中花费的总时间，以毫秒计
self_time	double precision	在这个函数本身花费的总时间，不包括被它调用的其他函数，以毫秒计

pg_stat_user_functions视图将为每一个被追踪的函数包含一行，该行显示有关该函数执行的统计信息。track_functions参数控制到底哪些函数被跟踪。

28.2.3. 统计函数

其他查看统计信息的方法是直接使用查询，这些查询使用上述标准视图用到的底层统计信息访问函数。如要了解如函数名等细节，可参考标准视图的定义（例如，在psql中你可以发出\d+ pg_stat_activity）。针对每一个数据库统计信息的访问函数把一个数据库 OID 作为参数来标识要报告哪个数据库。而针对每个表和每个索引的函数要求表或索引 OID。针对每个函数统计信息的函数用一个函数 OID。注意只有在当前数据库中的表、索引和函数才能被这些函数看到。

与统计收集相关的额外函数被列举在表 28.19中。

表 28.19. 额外统计函数

函数	返回类型	描述
pg_backend_pid()	integer	处理当前会话的服务器进程的进程 ID
pg_stat_get_activity(integer)	setof record	返回具有指定 PID 的后端相关的一个记录，或者在指定NULL的情况下为系统中每一个活动后端返回一个记录。被返回的域是pg_stat_activity视图中的那些域的一个子集。
pg_stat_get_snapshot_timestamp()	timestamp with time zone	返回当前统计信息快照的时间戳
pg_stat_clear_snapshot()	void	抛弃当前的统计快照
pg_stat_reset()	void	把用于当前数据库的所有统计计数器重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）。
pg_stat_reset_shared(text)	void	把某些集簇范围的统计计数器重置为零，具体哪些取决于参数（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）。 调用pg_stat_reset_shared('bgwriter')把pg_stat_bgwriter视图中显示的所有计数器清零。调用pg_stat_reset_shared('archiver')将会把pg_stat_archiver视图中展示的所有计数器清零。
pg_stat_reset_single_table_counters(oid)	void	把当前数据库中用于单个表或索引的统计数据重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）
pg_stat_reset_single_function_counters(oid)	void	把当前数据库中用于单个函数的统计信息重置为零（默认要求超级用户权限，但这个函数的 EXECUTE 可以被授予给其他人）

pg_stat_get_activity是pg_stat_activity视图的底层函数，它返回一个行集合，其中包含有关每个后端进程所有可用的信息。有时只获得该信息的一个子集可能会更方便。在那些情况中，可以使用一组更老的针对每个后端的统计访问函数，这些显示在表 28.20中。这些访问函数使用一个后端 ID 号，范围从 1 到当前活动后端数目。函数pg_stat_get_backend_idset提供了一种方便的方法为每个活动后端产生一行来调用这些函数。例如，要显示PID以及所有后端当前的查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
```

```
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

表 28.20. 针对每个后端的统计函数

函数	返回类型	描述
pg_stat_get_backend_idset()	setof integer	当前活动后端 ID 号的集合（从 1 到活动后端数目）
pg_stat_get_backend_activity(integer)	text	这个后端最近查询的文本
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	最近查询被开始的时间
pg_stat_get_backend_client_addr(integer)	inet	该客户端连接到这个后端的 IP 地址
pg_stat_get_backend_client_port(integer)	integer	该客户端用来通信的 TCP 端口号
pg_stat_get_backend_dbid(integer)	oid	这个后端连接到的数据库的 OID
pg_stat_get_backend_pid(integer)	integer	这个后端的进程 ID
pg_stat_get_backend_start(integer)	timestamp with time zone	这个进程被开始的时间
pg_stat_get_backend_userid(integer)	oid	登录到这个后端的用户的 OID
pg_stat_get_backend_wait_event_type(integer)	text	如果后端正在等待，则是等待事件类型的名称，否则为 NULL。详见表 28.4
pg_stat_get_backend_wait_event(integer)	text	如果后端正在等待，则是等待事件的名称，否则为 NULL。详见表 28.4
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	当前事务被开始的时间

28.3. 查看锁

监控数据库活动的另外一个有用的工具是pg_locks系统表。这样就允许数据库管理员查看在锁管理器里面未解决的锁的信息。例如，这个功能可以被用于：

- 查看当前所有未解决的锁、在一个特定数据库中的关系上所有的锁、在一个特定关系上所有的锁，或者由一个特定PostgreSQL会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系（它很可能是数据库客户端的竞争源）。
- 判断锁竞争给数据库总体性能带来的影响，以及锁竞争随着整个数据库流量的变化范围。

pg_locks视图的细节在第 52.73 节。更多有关PostgreSQL的锁和管理并发性的信息，请参考第 13 章。

28.4. 进度报告

PostgreSQL有能力在命令执行期间报告特定命令的进度。当前，唯一一种支持进度报告的命令是VACUUM。这在未来可能会被扩充。

28.4.1. VACUUM进度报告

只要VACUUM正在运行，每一个当前正在清理的后端（包括autovacuum工作者进程）在pg_stat_progress_vacuum视图中都会有一行。下面的表描述了将被报告的信息并且提供了如何解释它们的信息。进度报告当前不支持VACUUM FULL，运行着VACUUM FULL的后端将不会在这个视图中列出。

表 28.21.pg_stat_progress_vacuum视图

列	类型	描述
pid	integer	后端的进程ID。
datid	oid	这个后端连接的数据库的OID。
datname	name	这个后端连接的数据库的名称。
relid	oid	被vacuum的表的OID。
phase	text	vacuum的当前处理阶段。请参考表 28.22
heap_blks_total	bigint	该表中堆块的总数。这个数字在扫描开始时报告，之后增加的块将不会（并且不需要）被这个VACUUM访问。
heap_blks_scanned	bigint	被扫描的堆块数量。由于可见性映射被用来优化扫描，一些块将被跳过而不做检查，被跳过的块会被包括在这个总数中，因此当清理完成时这个数字最终将会等于heap_blks_total。仅当处于扫描堆阶段时这个计数器才会前进。
heap_blks_vacuumed	bigint	被清理的堆块数量。除非表没有索引，这个计数器仅在处于清理堆阶段时才会前进。不包含死亡元组的块会被跳过，因此这个计数器可能有时会向前跳跃一个比较大的增量。
index_vacuum_count	bigint	已完成的索引清理周期数。
max_dead_tuples	bigint	在需要执行一个索引清理周期之前我们可以存储的死亡元组数，取决于maintenance_work_mem。
num_dead_tuples	bigint	从上一个索引清理周期以来收集的死亡元组数。

表 28.22. VACUUM的阶段

阶段	描述
初始化	VACUUM正在准备开始扫描堆。这个阶段应该很简短。
扫描堆	VACUUM正在扫描堆。如果需要，它将会对每个页面进行修建以及碎片整理，并且可能会执行冻结动作。heap_blks_scanned列可以用来监控扫描的进度。
清理索引	VACUUM当前正在清理索引。如果一个表拥有索引，那么每次清理时这个阶段会在堆扫描完成后至少发生一次。如果maintenance_work_mem不足以存放找到的死亡元组，则每次清理时会多次清理索引。

阶段	描述
清理堆	VACUUM当前正在清理堆。清理堆与扫描堆不是同一个概念，清理堆发生在每一次清理索引的实例之后。如果heap_blks_scanned小于heap_blks_total，系统将在这个阶段完成之后回去扫描堆；否则，系统将在这个阶段完成后开始清理索引。
清除索引	VACUUM当前正在清除索引。这个阶段发生在堆被完全扫描并且对堆和索引的所有清理都已经完成以后。
截断堆	VACUUM正在截断堆，以便把关系尾部的空页面返还给操作系统。这个阶段发生在清除完索引之后。
执行最后的清除	VACUUM在执行最终的清除。在这个阶段中，VACUUM将清理空闲空间映射、更新pg_class中的统计信息并且将统计信息报告给统计收集器。当这个阶段完成时，VACUUM也就结束了。

28.5. 动态追踪

PostgreSQL提供了功能来支持数据库服务器的动态追踪。这样就允许在代码中的特 定点上调用外部工具来追踪执行过程。

一些探针或追踪点已经被插入在源代码中。这些探针的目的是被数据库开发者和管理员使用。默认情况下，探针不被编译到PostgreSQL中；用户需要显式地告诉配置脚本使得探针可用。

目前，在写本文当时DTrace[1]已被支持，它在 Solaris、macOS、FreeBSD、NetBSD 和 Oracle Linux 上可用。Linux 的SystemTap[2]项目提供了一种可用的 DTrace 等价物。支持其他动态追踪工具在理论上可以通过改变src/include/utils/probes.h中的宏定义实现。

28.5.1. 动态追踪的编译

默认情况下，探针是不可用的，因此你将需要显式地告诉配置脚本让探针在PostgreSQL中可用。要包括 DTrace 支持，在配置时指定--enable-dtrace。更多信息请见第 16.4 节。

28.5.2. 内建探针

如表 28.23所示，源代码中提供了一些标准探针。表 28.24显式了在探针中使用的类型。当然，可以增加更多探针来增强PostgreSQL的可观测性。

表 28.23. 内建 DTrace 探针

名称	参数	描述
transaction-start	(LocalTransactionId)	在一个新事务开始时触发的探针。arg0 是事务 ID。
transaction-commit	(LocalTransactionId)	在一个事务成功完成时触发的探针。arg0 是事务 ID。
transaction-abort	(LocalTransactionId)	当一个事务失败完成时触发的探针。arg0 是事务 ID。

[1] https://en.wikipedia.org/wiki/DTrace
[2] http://sourceware.org/systemtap/

名称	参数	描述
query-start	(const char *)	当一个查询的处理被开始时触发的探针。arg0 是查询字符串。
query-done	(const char *)	当一个查询的处理完成时触发的探针。arg0 是查询字符串。
query-parse-start	(const char *)	当一个查询的解析被开始时触发的探针。arg0 是查询字符串。
query-parse-done	(const char *)	当一个查询的解析完成时触发的探针。arg0 是查询字符串。
query-rewrite-start	(const char *)	当一个查询的重写被开始时触发的探针。arg0 是查询字符串。
query-rewrite-done	(const char *)	当一个查询的重写完成时触发的探针。arg0 是查询字符串。
query-plan-start	()	当一个查询的规划被开始时触发的探针。
query-plan-done	()	当一个查询的规划完成时触发的探针。
query-execute-start	()	当一个查询的执行被开始时触发的探针。
query-execute-done	()	当一个查询的执行完成时触发的探针。
statement-status	(const char *)	任何时候当服务器进程更新它的pg_stat_activity.status时触发的探针。arg0 是新的状态字符串。
checkpoint-start	(int)	当一个检查点被开始时触发的探针。arg0 保持逐位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
checkpoint-done	(int, int, int, int, int)	当一个检查点完成时触发的探针（检查点处理过程中序列中列出的下一个触发的探针）。arg0 是要写的缓冲区数量。arg1 是缓冲区的总数。arg2、arg3 和 arg4 分别包含了增加、删除和循环回收的 WAL 文件的数量。
clog-checkpoint-start	(bool)	当一个检查点的 CLOG 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。

名称	参数	描述
clog-checkpoint-done	(bool)	当一个检查点的 CLOG 部分完成时触发的探针。arg0 的含义与clog-checkpoint-start中相同。
subtrans-checkpoint-start	(bool)	当一个检查点的 SUBTRANS 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
subtrans-checkpoint-done	(bool)	当一个检查点的 SUBTRANS 部分完成时触发的探针。arg0 的含义与subtrans-checkpoint-start中相同。
multixact-checkpoint-start	(bool)	当一个检查点的 MultiXact 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
multixact-checkpoint-done	(bool)	当一个检查点的 MultiXact 部分完成时触发的探针。arg0 的含义与multixact-checkpoint-start中相同。
buffer-checkpoint-start	(int)	当一个检查点的写缓冲区部分被开始时触发的探针。arg0 保持逐位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
buffer-sync-start	(int, int)	当我们在检查点期间开始写脏缓冲区时（在标识哪些缓冲区必须被写之后）触发的探针。arg0 是缓冲区总数，arg1 是当前为脏并且需要被写的缓冲区数量。
buffer-sync-written	(int)	在检查点期间当每个缓冲区被写完之后触发的探针。arg0 是缓冲区的 ID。
buffer-sync-done	(int, int, int)	当所有脏缓冲区被写之后触发的探针。arg0 是缓冲区总数。arg1 是检查点进程实际写的缓冲区数量。arg2 是期望写的数目（buffer-sync-start的 arg1）；arg1 和 arg2 的任何的不同反映在该检查点期间有其他进程刷写了缓冲区。
buffer-checkpoint-sync-start	()	在脏缓冲区被写入到内核之后并且在开始发出 fsync 请求之前触发的探针。
buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探针。

名称	参数	描述
twophase-checkpoint-start	()	当一个检查点的两阶段部分被开始时触发的探针。
twophase-checkpoint-done	()	当一个检查点的两阶段部分完成时触发的探针。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当一次缓冲区读被开始时触发的探针。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 为 -1）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。表示真，对共享缓冲区表示假。 arg6 为真表示一次关系扩展请求，为假表示正常读。
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当一次缓冲区读完成时触发的探测器。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 现在包含新增加块的块号）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。表示真，对共享缓冲区表示假。 arg6 为真表示一次关系扩展请求，为假表示正常读。arg7 为真表示在池中找到该缓冲区，为假表示没有找到。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	在发出对一个共享缓冲区的任意写请求之前触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个写请求完成时触发的探针（注意这只反映传递数据给内核的时间，它通常并没有实际地被写入到磁盘）。参数和buffer-flush-start的相同。

名称	参数	描述
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个服务器进程开始写一个脏缓冲区时触发的探针（如果这经常发生，表示shared_buffers太小，或需要调整后台写入器的控制参数）。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一次脏缓冲区写完成时触发的探针。参数与buffer-write-dirty-start相同。
wal-buffer-write-dirty-start	()	当一个服务器进程因为没有可用 WAL 缓冲区空间开始写一个脏 WAL 缓冲区时触发的探针（如果这经常发生，表示wal_buffers太小）。
wal-buffer-write-dirty-done	()	当一次脏 WAL 缓冲区完成时触发的探针。
wal-insert	(unsigned char, unsigned char)	当一个 WAL 记录被插入时触发的探针。arg0 是该记录的资源管理者（rmid）。arg1 包含 info 标志。
wal-switch	()	当请求一次 WAL 段切换时触发的探针。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始从一个关系读取一块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一次块读取完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6 是实际读取的字节数，而 arg7 是请求读取的字节数（如果两者不同就意味着麻烦）。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始向一个关系中写入一个块时触发的探针。arg0 和 arg1 包含该页的分叉号

名称	参数	描述
		和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个块写操作完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3和arg4 包含表空间、数据库和关系 OID来标识该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6是实际写的字节数，而 arg7是要求写的字节数（如果这两者不同，则意味着麻烦）。
sort-start	(int, bool, int, int, bool, int)	当一次排序操作开始时触发的探针。arg0 指示是堆排序、索引排序或数据排序。arg1 为真表示唯一值强制。arg2 是键列的数目。arg3 是允许使用的工作内存数（以千字节计）。如果要求随机访问排序结果，那么 arg4 为真。arg5为0时表示串行，为1时表示并行工作者，为2时表示并行领袖。
sort-done	(bool, long)	当一次排序完成时触发的探针。arg0 为真表示外排序，为假表示内排序。arg1 是用于一次外排序的磁盘块的数目，或用于一次内排序的以千字节计的内存。
lwlock-acquire	(char *, LWLockMode)	当成功获得一个 LWLock 时触发的探针。 arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-release	(char *)	当一个 LWLock 被释放时（但是注意还没有唤醒任何一个被释放的等待者）触发的探针。 arg0 是该 LWLock 所在的切片（Tranche）。
lwlock-wait-start	(char *, LWLockMode)	当一个 LWLock不是当即可用并且一个服务器进程因此开始等待该锁变为可用时触发的探针。 arg0 是该 LWLock 所在的切片

名称	参数	描述
		（Tranche）。 arg1 请求的锁模式，是排他或共享。
lwlock-wait-done	(char *, LWLockMode)	当一个进程从对一个 LWLock 的等待中被释放时（它实际还没有得到该锁）时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire	(char *, LWLockMode)	当调用者指定无需等待而成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire-fail	(char *, LWLockMode)	当调用者指定无需等待而没有成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr锁）的请求由于锁不可用开始等待时触发的探针。arg0 到 arg3 是标识被锁定对象的标签域。arg4 指示被锁对象的类型。arg5 表示被请求的锁类型。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求结束等待时（即已经得到锁）触发的探针。参数与lock-wait-start一样。
deadlock-found	()	当死锁检测器发现死锁时触发的探针。

表 28.24. 定义用在探针参数中的类型

类型	定义
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
oid	unsigned int
ForkNumber	int
bool	char

28.5.3. 使用探针

下面的例子展示了一个分析系统中事务计数的 DTrace 脚本，可以用来代替一次性能测试之前和之后的pg_stat_database快照：

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

当被执行时，该例子 D 脚本给出这样的输出：

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

> **注意**
>
> SystemTap 为追踪脚本使用一个不同于 DTrace 的标记，但是底层的探针是兼容的。值得注意的是，在这样写的时候，SystemTap 脚本必须使用双下划线代替连字符来引用探针名。在未来的 SystemTap 发行中这很可能会被修复。

你应该记住，DTrace 脚本需要细心地编写和调试，否则被收集的追踪信息可能会毫无意义。在大部分发现问题的情况中，它就是发生问题的部件，而不是底层系统。当讨论使用动态追踪发现的信息时，一定要封闭使用的脚本来允许这些以便被检查和讨论。

28.5.4. 定义新探针

开发者可以在代码中任意位置定义新的探针，当然这要重新编译之后才能生效。下面是插入新探针的步骤：

1. 决定探针名称以及探针可用的数据
2. 把该探针定义加入到src/backend/utils/probes.d
3. 如果pg_trace.h还不存在于包含该探针点的模块中，包括它，并且在源代码中期望的位置插入TRACE_POSTGRESQL探针宏
4. 重新编译并验证新探针是可用的

例子：. 这里是一个如何增加一个探针来用事务 ID 追踪所有新事务的例子。

1. 决定探针将被命名为transaction-start并且需要一个LocalTransactionId类型的参数

2. 将该探针定义加入到src/backend/utils/probes.d：

   ```
   probe transaction__start(LocalTransactionId);
   ```

   注意探针名字中双下划线的使用。在一个使用探针的 DTrace 脚本中，双下划线需要被替换为一个连字符，因此 ，对用户而言transaction-start是文档名。

3. 在编译时，transaction__start被转换成一个宏调用TRACE_POSTGRESQL_TRANSACTION_START（注意这里是单下划线），可以通过包括头文件pg_trace.h获得。将宏调用加入到源代码中的合适位置。在这种情况下，看起来类似：

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```

4. 在重新编译和运行新的二进制文件之后，通过运行下面的 DTrace 命令来检查新增的探针是否可用。你应该看到类似下面的输出：

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand transaction-
   start
   18755 postgresql49877     postgres     StartTransactionCommand transaction-
   start
   18805 postgresql49876     postgres     StartTransactionCommand transaction-
   start
   18855 postgresql49875     postgres     StartTransactionCommand transaction-
   start
   18986 postgresql49873     postgres     StartTransactionCommand transaction-
   start
   ```

向C代码中添加追踪宏时，有一些事情需要注意：

- 需要小心的是，为探针参数指定的数据类型要匹配宏中使用的变量的数据类型，否则会发生编译错误。
- 在大多数平台上，如果用--enable-dtrace编译了PostgreSQL，无论何时当控制经过一个追踪宏时，都会评估该宏的参数，即使没有进行追踪也会这样做。通常不需要担心你是否只在报告一些局部变量的值。但要注意将开销大的函数调用放置在这些参数中。如果你需要这样做，考虑通过检查追踪是否真的被启用来保护该宏：

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  每个追踪宏有一个对应的ENABLED宏。

第 29 章 监控磁盘使用

本章讨论如何监控PostgreSQL数据库系统的磁盘使用情况。

29.1. 判断磁盘用量

每个表都有一个主要的堆磁盘文件，大多数数据都存储在其中。如果一个表有着可能会很宽（尺寸大）的列， 则另外还有一个TOAST文件与这个表相关联， 它用于存储因为太宽而不能存储在主表里面的值（参阅第 68.2 节。如果有这个附属文件，那么TOAST表上会有一个可用的索引。 当然，同时还可能有索引和基表关联。每个表和索引都存放在单独的磁盘文件里 — 如果文件超过 1G 字节，甚至可能多于一个文件。这些文件的命名原则在第 68.1 节描述。

你可以以三种方式监视磁盘空间：使用表 9.84中列出的SQL函数、使用oid2name模块或者人工观察系统目录。SQL函数是最容易使用的方法，同时也是我们通常推荐的方法。本节剩余的部分将展示如何通过观察系统目录来监视磁盘空间。

在一个最近清理过或者分析过的数据库上使用psql，你可以发出查询来查看任意表的磁盘用量：

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname =
 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

每个页通常都是 8K 字节（记住，relpages只会由VACUUM、ANALYZE和少数几个 DDL 命令如CREATE INDEX所更新）。如果你想直接检查表的磁盘文件，那么文件路径名应该有用。

要显示TOAST表使用的空间，我们可以使用一个类似下面这样的查询：

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

你也可以很容易地显示索引的尺寸：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
```

```
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname         | relpages
-----------------------+----------
 customer_id_indexdex |       26
```

我们很容易用下面的信息找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname         | relpages
-----------------------+----------
 bigtable              |     3290
 customer              |     3144
```

29.2. 磁盘满失败

一个数据库管理员最重要的磁盘监控任务就是确保磁盘不会写满。一个写满了的数据磁盘可能不会导致数据的崩溃，但它肯定会让系统变得不可用。如果保存 WAL 文件的磁盘变满，会发生数据库服务器致命错误并且可能发生关闭。

如果你不能通过删除一些其他的东西来释放一些磁盘空间，那么你可以通过使用表空间把一些数据库文件移动到其他文件系统上去。参阅第 22.6 获取更多信息。

> **提示**
>
> 有些文件系统在快满的时候性能会急剧恶化，因此不要等到磁盘完全满的时候才采取行动。

如果你的系统支持每用户的磁盘份额，那么数据库将自然地受制于用户所处的服务器给他的份额限制。超过份额的负面影响和完全用光磁盘是完全一样的。

第 30 章 可靠性和预写式日志

本章解释预写式日志如何用于获得有效的、可靠的操作。

30.1. 可靠性

可靠性是任何严肃的数据库系统的重要属性，PostgreSQL尽一切可能来保证可靠的操作。可靠的操作的一个方面是，被一个提交事务记录的所有数据应该被存储在一个非易失的区域，这样就不会因为失去电力、操作系统失败以及硬件失败（当然，除了非易失区域自身失效之外）等原因导致的数据丢失。 向计算机的永久存储（磁盘驱动器或者等效的设备）成功写入数据通常可以满足这个要求。 实际上，即使计算机受到致命损坏，只要磁盘驱动器幸存下来，那么它们就可以被移动到另外一台具有类似硬件的计算机上， 而所有已经提交的事务将保持原状。

周期地强制数据进入磁盘盘片看上去像一件简单的操作，但实际上并不是。 因为磁盘驱动器比内存和CPU要慢很多，在计算机的主存和磁盘盘片之间存在多层的高速缓存。 首先，有操作系统的高速缓存，它缓冲常用的磁盘块并且组合对磁盘的写入。 幸运的是，所有操作系统都给予应用一种强制从高速缓存写入磁盘的方法，PostgreSQL则使用了那个特性（参阅wal_sync_method参数调节如何完成之）。

然后，在磁盘驱动器的控制器上可能还有一个高速缓存；这在RAID控制卡上是特别常见的。有些高速缓存是直写式的，即写入动作在到达的时候就立刻写入到磁盘上。其它是回写式的， 即发送给驱动器的数据在稍后的某个时间写入驱动器。这样的高速缓存可能会称为可靠性灾难，因为磁盘控制器高速缓存的内存是易失性的，在发生电力失败的情况下会丢失其内容。 好一些的控制器卡有后备电池单元（BBU）， 即这种卡上面有电池可以在系统电力失败的情况下提供电力。 在电力恢复之后，这些数据将会被写入磁盘驱动器。

最后，大多数磁盘驱动器都有高速缓存。有些是直写的，有些是回写的， 和磁盘控制器一样，回写的磁盘高速缓存也存在数据丢失的问题。 消费级别的IDE和SATA驱动器尤其可能包含回写式高速缓存，在掉电的情况下很容易丢失数据。很多固态驱动器（SSD）也具有易失性回写式高速缓存。

这些高速缓存通常可以被禁用，但是不同的操作系统和驱动器类型有不同的做法：

- 在Linux上，可以使用hdparm -I查询IDE和SATA驱动器，如果在Write cache之后有一个*则表示写高速缓存被启用。可以用hdparm -W 0来关闭写高速缓存。可以使用sdparm[1]查询SCSI驱动器。使用sdparm --get=WCE来检查写高速缓存是否被启用，而sdparm --clear=WCE可以用来禁用它。

- 在FreeBSD上，IDE驱动器可以使用atacontrol查询，而写高速缓存可以用/boot/loader.conf中的hw.ata.wc=0关闭。SCSI驱动器可以使用camcontrol identify查询，而写高速缓存的查询和更改都可以使用sdparm。

- 在Solaris上，磁盘的写高速缓存被format -e控制（Solaris的ZFS文件系统对于开启的磁盘写高速缓存是安全的，因为它会发出它自己的磁盘高速缓存刷写命令）。

- 在Windows上，如果wal_sync_method是open_datasync（默认值），写高速缓存可以通过取消选中My Computer\Open\disk drive\Properties\Hardware\Properties\Policies\Enable write caching on the disk禁用。另一种方法可以通过设置wal_sync_method为fsync或fsync_writethrough来阻止写高速缓存。

- 在macOS上，通过设置wal_sync_method为fsync_writethrough可以阻止写高速缓存。

最近的SATA驱动器（遵循ATAPI-6及更新标准）提供了一个驱动器高速缓存刷写命令（FLUSH CACHE EXT），而SCSI驱动器有一个存在很长时间的类似命令SYNCHRONIZE CACHE。这些命令

[1] http://sg.danny.cz/sg/sdparm.html

对于PostgreSQL并不能直接访问，但某些文件系统（例如ZFS、ext4）可以使用它们将数据刷写到回写式驱动器的盘片上。不幸的是，这些文件系统在和后备电池单元（BBU）一起工作时的表现要略差。在这种设置下，同步命令强制所有来自控制器高速缓存的数据到磁盘，消除了BBU的很多好处。你可以运行pg_test_fsync程序来看你是否被影响。如果你被影响了，BBU带来的性能好处可以通过关闭文件系统的写障碍或者重新配置磁盘控制器来重新获得。如果写障碍被关闭，请确认电池是否保持有效，一个有问题的电池可能会导致数据丢失。但愿文件系统和磁盘控制器设计师们将最终解决这种次优行为。

在操作系统向存储硬件发出一个写请求的时候，它没有什么好办法来保证数据真正到达非易失的存储区域。 实际上，确保所有存储部件都保证数据和文件系统元数据的完整性是管理员的责任。 避免使用那些没有电池作为后备的写高速缓存的磁盘控制器。在驱动器级别，如果驱动器不能保证在关闭（掉电）之前写入数据， 那么关闭回写高速缓冲。如果你在使用SSD，注意很多SSD默认都没有兑现高速缓存刷写命令。你可以使用diskchecker.pl[2]来测试可靠的I/O子系统行为。

另外一个数据丢失的风险来自磁盘盘片写操作自身。磁盘盘片会被分割为扇区，通常每个扇区512字节。每次物理读写都对整个扇区进行操作。当一个写操作到达磁盘的时候，它可能是512 字节（PostgreSQL通常一次写8192字节或者16个扇区）的某个倍数，而写入处理可能因为电力失效在任何时候失败，这 意味着某些512字节的扇区写入了，而有些没有。为了避免这样的失效，PostgreSQL在修改磁盘上的实际页面之前， 周期地把整个页面的映像写入永久WAL存储。这么做之后，在崩溃恢复的时候，PostgreSQL可以从WAL恢复部分写入的页面。如果你的文件系统阻止部分页面写入（如ZFS），你可以通过关闭full_page_writes参数来关闭这种页映像。后备电池单元（BBU）磁盘控制器不阻止部分页面写入，除非它们保证数据都是以整页（8kB）写入到BBU。

PostgreSQL也能防止由于硬件错误或者介质失败超时在存储设备上造成的各种数据损坏，例如读/写垃圾数据。

- WAL文件中的每一个记录都被一个CRC-32（32位）校验码所保护，这让我们可以判断记录内容是否正确。CRC值在我们写入每一个WAL记录时设置，并且在崩溃恢复、归档恢复和复制时检查。

- 目前数据页并没有默认地被校验，但是WAL记录中记录的整页映像将被保护。关于启用数据页校验的内容详见initdb。

- 诸如pg_xact、pg_subtrans、pg_multixact、pg_serial、pg_notify、pg_stat、pg_snapshots等内部数据结构既没有被直接校验，其页面也没有被整页写保护。但是，这些数据结构是持久的话，WAL记录被写入，它允许最近的修改能在崩溃恢复时被准确重建且这些WAL记录被按照以上讨论的方式保护着。

- pg_twophase中的单个状态文件被CRC-32保护。

- 用在大型SQL查询中排序的临时数据库文件、物化和中间结果目前没有被校验，对于这些文件的改变也不会导致写入WAL记录。

PostgreSQL无法避免可更正内存错误，它假定你会操作由工业标准纠错码（ECC）或更好方案保护的RAM。

30.2. 预写式日志（WAL）

预写式日志（WAL）是保证数据完整性的一种标准方法。对其详尽的描述几乎可以在所有（如果不是全部）有关事务处理的书中找到。简单来说，WAL的中心概念是数据文件（存储着表和索引）的修改必须在这些动作被日志记录之后才被写入，即在描述这些改变的日志记录被刷到持久存储以后。如果我们遵循这种过程，我们不需要在每个事务提交时刷写数据页面到磁盘，因为我们知道在发生崩溃时可以使用日志来恢复数据库：任何还没有被应用到数据页面的改变可以根据其日志记录重做（这是前滚恢复，也被称为REDO）。

[2] https://brad.livejournal.com/2116715.html

> **提示**
>
> 因为WAL在崩溃后恢复数据库文件内容，不需要日志化文件系统作为数据文件或WAL文件的可靠存储。实际上，日志会降低性能，特别是如果日志导致文件系统数据被刷写到磁盘。幸运地是，日志期间的数据刷写常常可以在文件系统挂载选项中被禁用，例如在Linux ext3文件系统中可以使用data=writeback。在崩溃后日志化文件系统确实可以提高启动速度。

使用WAL可以显著降低磁盘的写次数，因为只有日志文件需要被刷出到磁盘以保证事务被提交，而被事务改变的每一个数据文件则不必被刷出。日志文件被按照顺序写入，因此同步日志的代价要远低于刷写数据页面的代价。在处理很多影响数据存储不同部分的小事务的服务器上这一点尤其明显。此外，当服务器在处理很多小的并行事务时，日志文件的一个fsync可以提交很多事务。

WAL也使得在线备份和时间点恢复能被支持，如第 25.3 所述。通过归档WAL数据，我们可以支持回转到被可用WAL数据覆盖的任何时间：我们简单地安装数据库的一个较早的物理备份，并且重放WAL日志一直到所期望的时间。另外，该物理备份不需要是数据库状态的一个一致的快照 — 如果它的制作经过了一段时间，则重放这一段时间的WAL日志将会修复任何内部不一致性。

30.3. 异步提交

异步提交是一个允许事务能更快完成的选项，代价是在数据库崩溃时最近的事务会丢失。在很多应用中这是一个可接受的交换。

如前一节所述，事务提交通常是同步的：服务器等到事务的WAL记录被刷写到持久存储之后才向客户端返回成功指示。因此客户端可以确保那些报告已被提交的事务确会被保存，即便随后马上发生了一次服务器崩溃。但是，对于短事务来说这种延迟是其总执行时间的主要部分。选择异步提交模式意味着服务器将在事务被逻辑上提交后立刻返回成功，而此时由它生成的WAL记录还没有被真正地写到磁盘上。这将为小型事务的生产力产生显著地提升。

异步提交会带来数据丢失的风险。在向客户端报告事务完成到事务真正被提交（即能保证服务器崩溃时它也不会被丢失）之间有一个短的时间窗口。因此如果客户端将会做一些要求其事务被记住的外部动作，就不应该用异步提交。例如，一个银行肯定不会使用异步提交事务来记录一台ATM的现金分发。但是在很多情境中不需要这种强的保证，例如事件日志。

使用异步提交带来的风险是数据丢失，而不是数据损坏。如果数据库可能崩溃，它会通过重放WAL到被刷写的最后一个记录来进行恢复。数据库将因此被恢复到一个自身一致状态，但是任何还没有被刷写到磁盘的事务将不会反映在该状态中。因此其影响就是丢失了最后的少量事务。由于事务按照提交顺序被重放，所以不会出现任何不一致性 — 例如一个事务B按照前面一个事务A的效果来进行修改，则不会出现A的效果丢失而B的效果被保留的情况。

用户可以选择每一个事务的提交模式，这样可以有同步提交和异步提交的事务并行运行。这允许我们灵活地在性能和事务持久性之间进行权衡。提交模式由用户可设置的参数synchronous_commit控制，它可以使用任何一种修改配置参数的方法进行设置。一个事务真正使用的提交模式取决于当事务提交开始时synchronous_commit的值。

特定的实用命令，如DROP TABLE，被强制按照同步提交而不考虑synchronous_commit的设定。这是为了确保服务器文件系统和数据库逻辑状态之间的一致性。支持两阶段提交的命令页总是同步提交的，如PREPARE TRANSACTION。

如果数据库在异步提交和事务WAL记录写入之间的风险窗口期间崩溃，在该事务期间所作的修改将丢失。风险窗口的持续时间是有限制的，因为一个后台进程（“WAL写进程”）每wal_writer_delay毫秒会把未写入的WAL记录刷写到磁盘。风险窗口实际的最大持续时间是wal_writer_delay的3倍，因为WAL写进程被设计成倾向于在忙时一次写入所有页面。

> **小心**
>
> 一个立刻关闭等同于一次服务器崩溃，因此也将会导致未刷写的异步提交丢失。

异步提交提供的行为与配置fsync = off不同。fsync是一个服务器范围的设置，它将会影响所有事务的行为。它禁用了PostgreSQL中所有尝试同步写入到数据库不同部分的逻辑，并且因此一次系统崩溃（即，一个硬件或操作系统崩溃，不是PostgreSQL本身的失败）可能造成数据库状态的任意损坏。在很多情境中，带来大部分性能提升的异步提交可以通过关闭fsync来获得，而且不会带来数据损坏的风险。

commit_delay也看起来很像异步提交，但它实际上是一种同步提交方法（事实上，commit_delay在异步提交时被忽略）。commit_delay会使事务在刷写WAL到磁盘之前有一个延迟，它期望由一个这样的事务所执行的刷写能够也服务于其他同时提交的事务。该设置可以被看成是一种时间窗口，在其期间事务可以参与到一次单一的刷写中，这种方式用于在多个事务之间摊销刷写的开销。

30.4. WAL配置

有几个WAL相关的配置参数会影响数据库性能。本节将解释它们的使用。关于服务器配置参数的设置的一般信息请参考第 19 章

检查点是在事务序列中的点，这种点保证被更新的堆和索引数据文件的所有信息在该检查点之前已被写入。在检查点时刻，所有脏数据页被刷写到磁盘，并且一个特殊的检查点记录将被写入到日志文件（修改记录之前已经被刷写到WAL文件）。在崩溃时，崩溃恢复过程检查最新的检查点记录用来决定从日志中的哪一点（称为重做记录）开始REDO操作。在这一点之前对数据文件所做的任何修改都已经被保证位于磁盘之上。因此，完成一个检查点后位于包含重做记录的日志段之前的日志段就不再需要了，可以将其回收或删除（当WAL归档工作时，日志段在被回收或删除之前必须被归档）。

检查点对于刷写所有脏数据页到磁盘的要求可能会导致可观的I/O负载。出于这一原因，检查点活动是被有所限制的，这样I/O在检查点开始时开始并且能在下一个检查点将要开始之间完成，这使得检查点期间的性能下降被最小化。

服务器的检查点进程常常自动地执行一个检查点。检查点在每checkpoint_timeout秒开始，或者在快要超过 max_wal_size时开始。 默认的设置分别是 5 分钟和 1 GB。如果从前一个检查点以来没有WAL被写入， 则即使过了checkpoint_timeout新的检查点也会被跳过（ 如果正在使用WAL归档并且你想对文件被归档频率设置一个较低的限制来约束 潜在的数据丢失，你应该调整archive_timeout 参数而不是检查点参数）。也可以使用SQL命令 CHECKPOINT来强制一个检查点。

降低checkpoint_timeout和/或max_wal_size会导致检查点更频繁地发生。这使得崩溃后恢复更快，因为需要重做的工作更少。但是，我们必须在这一点和增多的刷写脏数据页开销之间做出平衡。如果full_page_writes被设置（默认情况），则还有一个因素需要考虑。为了确保数据页一致性，在每个检查点之后对一个数据页的第一次修改将导致整个页面内容被日志记录。在这情况下，一个较小的检查点间隔会增加输出到WAL日志的容量，这让使用较小间隔的效果打了折扣并且将导致更多的磁盘I/O。

检查点的代价相对比较昂贵，首先是因为它们要求写出所有当前为脏的缓冲区，正如以上讨论的，第二个原因是它们会导致额外的WAL流量。因此比较明智的做法是将检查点参数设置得足够高，这样检查点就不会过于频繁地发生。你可以设置checkpoint_warning参数作为对于你的检查点参数的一种简单完整性检查。如果检查点的发生时间间隔比checkpoint_warning秒还要接近，一个消息将会被发送到服务器日志来推荐你增加max_wal_size。偶尔出现的这样的消息并不会导致警报，但是如果它出现得太频繁，那么就应该增加检查点控制参数。 如果你没有把max_wal_size设置得足够高， 那么在进行如大型COPY传输等批量操作的时候可能会导致出现大量类似的警告消息。

为了避免大批页面写入对I/O系统产生的冲击，一个检查点中对脏缓冲区的写出操作被散布到一段时间上。这个时间段由checkpoint_completion_target控制，它用检查点间隔的一个分数表示。I/O率将被调整，以便能按照要求完成检查点：当checkpoint_timeout给定的秒数已经过去，或者max_wal_size被超过之前会发生检查点，以先达到的为准。默认值为0.5，PostgreSQL被期望能够在下一个检查点启动之前的大约一半时间内完成每个检查点。在一个接近于正常操作期间最大I/O的系统上，你可能希望增加checkpoint_completion_target来降低检查点的I/O负载。但这种做法的缺点是被延长的检查点将会影响恢复时间，因为需要保留更多WAL段来用于可能的恢复操作。尽管checkpoint_completion_target可以被设置为高于1.0，但最好还是让它小于1.0（也许最多0.9），因为检查点还包含除了写出脏缓冲区之外的其他一些动作。1.0的设置极有可能导致检查点不能按时被完成，这可能由于所需的WAL段数量意外变化导致性能损失。

在 Linux 和 POSIX 平台上，checkpoint_flush_after允许强制 OS 超过一个可配置的字节数后将检查点写入的页面刷入磁盘。否则，这些页面可能会被保留在 OS 的页面缓存中，当检查点结束发出fsync时就会导致大量刷写形成延迟。这个设置通常有助于减小事务延迟，但是它也可能对性能带来负面影响，尤其是对于超过shared_buffers但小于 OS 页面缓存的负载来说更是如此。

pg_wal目录中的 WAL 段文件数量取决于min_wal_size、max_wal_size以及在之前的检查点周期中产生的 WAL 数量。当旧的日志段文件不再被需要时，它们将被移除或者被再利用（也就是被重命名变成数列中未来的段）。如果由于日志输出率的短期峰值导致超过max_wal_size，不需要的段文件将被移除直到系统回到这个限制以下。低于该限制时，系统会再利用足够的 WAL 文件来覆盖直到下一个检查点之前的需要。这种需要是基于之前的检查点周期中使用的 WAL 文件数量的移动平均数估算出来的。如果实际用量超过估计值，移动平均数会立即增加，因此它能在一定程度上适应峰值用量而不是平均用量。min_wal_size对回收给未来使用的 WAL 文件的量设置了一个最小值，这个参数指定数量的 WAL 将总是被回收给未来使用，即便系统很闲并且 WAL 用量估计建议只需要一点点 WAL 时也是如此。

独立于max_wal_size之外，wal_keep_segments + 1 个最近的 WAL 文件将总是被保留。还有，如果使用了 WAL 归档，旧的段在被归档之前不能被移除或者再利用。如果 WAL 归档无法跟上产生 WAL 的步伐，或者如果archive_command重复失败，旧的 WAL 文件将累积在pg_wal中，直到该情况被解决。一个使用了复制槽的较慢或者失败的后备服务器也会带来同样的效果（见第 26.2.6 节。

在归档恢复模式或后备模式，服务器周期性地执行重启点。和正常操作时的检查点相似：服务器强制它所有的状态到磁盘，更新pg_control来指示已被处理的WAL数据不需要被再次扫描，并且接着回收pg_wal中的任何旧日志段文件。重启点的执行频率不能高于主机中检查点的执行频率，因为重启点只有在检查点记录处才能被执行。 如果从最后一个重启点之后过去了至少checkpoint_timeout秒或者 WAL 尺寸快要达到max_wal_size，则会到达一个检查点，这时会触发一个重启点。不过，因为对于何时可以执行一个重启点有限制，在恢复期间max_wal_size常常被超过，最多会超过一个检查点周期间的 WAL（不管怎样，max_wal_size从来不是一个硬限制，因此你应该总是应该留出充足的净空来避免耗尽磁盘空间）。

有两个常用的内部WAL函数：XLogInsertRecord和XLogFlush。 XLogInsertRecord用于向共享内存中的WAL缓冲区里放置一个新记录。如果没有空间存放新记录， 那么XLogInsertRecord就不得不写出（向内核缓存里写）一些填满了的WAL缓冲区。 这并非我们所期望的，因为XLogInsertRecord用于每次数据库低层修改（比如，记录插入）时都要在受影响的数据页上持有一个排它锁，因为该操作需要越快越好。但糟糕的是， 写WAL缓冲可能还会强制创建新的日志段，这花的时间甚至更多。通常，WAL缓冲区应该由一个XLogFlush请求来写和刷出， 在大部分时候它都是发生在事务提交的时候以确保事务记录被刷写到永久存储。在那些日志输出量比较大的系统上，XLogFlush请求可能不够频繁，这样就不能避免XLogInsert进行写操作。在这样的系统上，我们应该通过修改配置参数wal_buffers的值来增加WAL缓冲区的数量。如果设置了 full_page_writes并且系统相当繁忙， 把wal_buffers设置得更高一些将有助于在紧随每个检查点之后的时间段里得到平滑的响应时间。

commit_delay定义了一个组提交领导者进程在XLogFlush中要求一个锁之后将会休眠的微秒数，而组提交追随者都排队等候在领导者之后。这样的延迟可以允许其它服务器进程把它们

提交的记录追加到WAL缓冲区中，这样所有的这些记录将会被领导者的最终同步操作刷出。如果fsync被禁用或者当前处于活跃事务中的会话数少于commit_siblings，休眠将不会发生；这样就避免了在其它事务不会很快提交的情况下进行休眠。 请注意在某些平台上，休眠要求的单位是十毫秒，所以任何介于 1 和 10000 微秒之间的非零commit_delay设置的作用都是一样的。 还要注意在某些平台上，休眠操作用的时间会比该参数所请求的要略长一点。

由于commit_delay的目的是允许每次刷写操作的开销能够在并发提交的事务之间进行分摊（可能会以事务延迟为代价），在能够明智地选择该设置之前有必要对代价进行量化。代价越高，在一定程度上commit_delay对于提高事务吞吐量的效果就越好。pg_test_fsync程序可以被用来衡量一次WAL刷写操作需要的平均微秒数。该程序报告的一次8kB写操作后的刷出所用的平均时间的一半常常是commit_delay最有效的设置，因此在优化一种特定工作负荷时，该值被推荐为起始点。当WAL日志被存储在高延迟的旋转磁盘上时，调节commit_delay特别有效，即使在具有非常快同步时间的存储介质上也能得到很显著的收益，例如固态驱动器或具有电池后备写高速缓存的RAID阵列。但是这应该在一个具有代表性的工作负荷下进行明确地测试。较高的commit_siblings值应该用在这种情况中，反之较小的commit_siblings值通常对高延迟介质有用。注意过高的commit_delay设置也很有可能增加事务延迟甚至于整个事务吞吐量都会受到影响。

当commit_delay被设置为0（默认值），仍然有可能出现组提交的形式，但是组中的成员只能是那些在前一个刷写操作发生过程窗口中需要刷写它们提交记录的会话。在较高的客户端数量时很可能发生“gangway effect”，因此即使commit_delay为0，组提交的效果也很显著，并且显式地设置commit_delay将会没有作用。设置commit_delay只有在两种情况下有帮助：（1）有一些并发提交的事务，以及（2）吞吐量在某种程度上被提交率限制。但是在高旋转延迟的设备上，即使少到只有两个客户端，该设置也能有效提高事务吞吐量。

wal_sync_method参数决定PostgreSQL如何请求内核强制将WAL更新到磁盘。只要满足可靠性，那么除了fsync_writethrough所有选项应该都是一样的，fsync_writethrough可以在某些时候强制磁盘高速缓存的刷写，而其他选项不能这样做。不过，哪种选项最快则可能和平台密切相关。 你可以使用pg_test_fsync程序来测试不同选项的速度。请注意如果你关闭了fsync，那么这个参数就无所谓了。

启用wal_debug配置参数（前提是PostgreSQL编译的时候打开了这个支持） 将导致每次XLogInsertRecord和XLogFlush WAL调用都被记录到服务器日志。这个选项以后可能会被更通用的机制取代。

30.5. WAL内部

WAL是自动被启用的。除了确保满足WAL日志存放所需要的磁盘空间以及一些必要的调优外（参阅第 30.4 节，管理员无需执行任何操作。

当每个新记录被写入时，WAL记录被追加到WAL日志中。 插入位置由日志序列号（LSN）描述，该日志序列号是日志中的字节偏移量， 随每个新记录单调递增。LSN值作为数据类型pg_lsn返回。 值可以进行比较以计算分离它们的WAL数据量，因此它们用于衡量复制和恢复的进度。

WAL日志被存放在数据目录的pg_wal目录里，它是作为一个文件段的集合存储的，通常每个段16MB大小（不过这个大小可以通过initdb配置选项--with-wal-segsize来修改）。每个段分割成多个页，通常每个页为8K（该尺寸可以通过--with-wal-blocksize配置选项来修改）。日志记录头部在access/xlogrecord.h里描述；日志内容取决于它记录的事件类型。段文件的名字是不断增长的数字，从000000010000000000000000开始。目前这些数字不能回卷，不过要把所有可用的数字都用光也需要非常非常长的时间。

日志被放置在和主数据库文件不同的另外一个磁盘上会比较好。你可以通过把pg_wal目录移动到另外一个位置（当然在此期间服务器应当被关闭），然后在原来的位置上创建一个指向新位置的符号链接来实现重定位日志。

WAL的目的是确保在数据库记录被修改之前先写了日志，但是这可能会被那些谎称向内核写成功的破坏， 这时候它们实际上只是缓冲了数据而并未把数据存储到磁盘上。 这种情况下

的电源失效仍然可能导致不可恢复的数据崩溃。 管理员应该确保保存PostgreSQL的WAL日志文件的磁盘不会做这种谎报（参见第 30.1 节。

在完成一个检查点并且刷写了日志文件之后，检查点的位置被保存在文件pg_control里。因此在恢复的开始， 服务器首先读取pg_control，然后读取检查点记录； 接着它通过从检查点记录里标识的日志位置开始向前扫描执行 REDO操作。 因为数据页的所有内容都保存在检查点之后的第一个页面修改的日志里（假设full_page_writes没有被禁用）， 所以自检查点以来的所有变化的页都将被恢复到一个一致的状态。

为了处理pg_control被损坏的情况， 我们应该支持对于现有日志段反向扫描的功能 — 从最新到最老 — 这样才能找到最后的检查点。但这些目前还没有被实现。pg_control很小（比一个磁盘页小），因此它不会出现页断裂问题， 并且到目前为止还没有发现仅仅由于无法读取pg_control本身导致数据库失败的报告。 因此，尽管这在理论上是一个薄弱环节，但是pg_control看起来似乎并不是实际会发生的问题。

第 31 章 逻辑复制

逻辑复制是一种基于数据对象的复制标识（通常是主键）复制数据对象及其更改的方法。我们使用术语“逻辑”来与物理复制加以区分，后者使用准确的块地址以及逐字节的复制方式。PostgreSQL两种机制都支持，请见第 26 章逻辑复制允许在数据复制和安全性上更细粒度的控制。

逻辑复制使用一种发布和订阅模型，其中有一个或者更多订阅者订阅一个发布者节点上的一个或者更多发布 。订阅者从它们所订阅的发布拉取数据并且可能后续重新发布这些数据以允许级联复制或者更复杂的配置。

一个表的逻辑复制通常开始于对发布者服务器上的数据取得一个快照并且将快照拷贝给订阅者。一旦这项工作完成，发布者上的更改会被实时发送给订阅者。订阅者以与发布者相同的顺序应用那些数据，这样在一个订阅中能够保证发布的事务一致性。这种数据复制的方法有时候也被称为事务性复制。

逻辑复制的典型用法是：

- 在一个数据库或者一个数据库的子集中发生更改时，把增量的改变发送给订阅者。
- 在更改到达订阅者时引发触发器。
- 把多个数据库联合到单一数据库中（例如用于分析目的）。
- 在PostgreSQL的不同主版本之间进行复制。
- 在不同平台上（例如Linux到Windows）的PostgreSQL实例之间进行复制。
- 将复制数据的访问给予不同的用户组。
- 在多个数据库间共享数据库的一个子集。

订阅者数据库的行为与任何其他PostgreSQL实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。当订阅者被应用当作只读时，单一的订阅中不会有冲突。在另一方面，如果应用或者对相同表集合的订阅者执行了其他的写动作，冲突可能会发生。

31.1. 发布

发布可以被定义在任何物理复制的主服务器上。定义有发布的节点被称为发布者。发布是从一个表或者一组表生成的改变的集合，也可以被描述为更改集合或者复制集合。每个发布都只存在于一个数据库中。

发布与模式不同，不会影响表的访问方式。如果需要，每个表都可以被加入到多个发布。当前，发布只能包含表。对象必须被明确地加入到发布，除非发布是用ALL TABLES创建的。

Publication可以选择把它们产生的更改限制为INSERT、UPDATE、DELETE以及TRUNCATE的任意组合，类似于触发器如何被特定事件类型触发的方式。默认情况下，所有操作类型都会被复制。

为了能够复制UPDATE和DELETE操作，被发布的表必须配置有一个“复制标识”，这样在订阅者那一端才能标识对于更新或删除合适的行。默认情况下，复制标识就是主键（如果有主键）。也可以在复制标识上设置另一个唯一索引（有特定的额外要求）。如果表没有合适的键，那么可以设置成复制标识“full”，它表示整个行都成为那个键。不过，这样做效率很低，只有在没有其他方案的情况下才应该使用。如果在发布者端设置了“full”之外的复制标识，在订阅者端也必须设置一个复制标识，它应该由相同的或者少一些的列组成。如何设置复制标识的细节请参考REPLICA IDENTITY。如果在复制UPDATE或DELETE操作的发布中加入了没有复制标识的表，那么订阅者上后续的UPDATE或DELETE操作将导致错误。不管有没有复制标识，INSERT操作都能继续下去。

每一个发布都可以由多个订阅者。

Publication通过使用CREATE PUBLICATION命令创建并且可以在之后使用相应的命令进行修改或者删除。

表可以使用ALTER PUBLICATION动态地增加或者移除。ADD TABLE以及DROP TABLE操作都是事务性的，因此一旦该事务提交，该表将以正确的快照开始或者停止复制。

31.2. 订阅

订阅是逻辑复制的下游端。订阅被定义在其中的节点被称为订阅者。一个订阅会定义到另一个数据库的连接以及它想要订阅的发布集合（一个或者多个）。

订阅者数据库的行为与任何其他PostgreSQL实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。

如果需要，一个订阅者节点可以有多个订阅。可以在一对发布者-订阅者之间定义多个订阅，在这种情况下要确保被订阅的发布对象不会重叠。

每一个订阅都将通过一个复制槽（见第 26.2.6 节）接收更改。预先存在的表数据的初始数据同步过程可能会要求额外的临时复制槽。

逻辑复制订阅可以是同步复制（见第 26.2.8 节）的后备服务器。后备名称默认是该订阅的名称。可以在订阅的连接信息中用application_name指定一个可供选择的名称。

如果当前用户是一个超级用户，则订阅会被pg_dump转储。否则订阅会被跳过并且写出一个警告，因为非超级用户不能从pg_subscription目录中读取所有的订阅信息。

可以使用CREATE SUBSCRIPTION增加订阅，并且使用ALTER SUBSCRIPTION在任何时刻停止/继续订阅，还可以使用DROP SUBSCRIPTION删除订阅。

在一个订阅被删除并且重建时，同步信息会丢失。这意味着数据必须被重新同步。

模式定义不会被复制，并且被发布的表必须在订阅者上存在。只有常规表可以成为复制的目标。例如，不能复制视图。

表在发布者和订阅者之间使用完全限定的表名进行匹配。不支持复制到订阅者上命名不同的表。

表的列也通过名称匹配。允许在目标表中的列序不同，但是列类型必须匹配。目标表可以有被发布表没有提供的额外列。额外列将用其默认值填充。

31.2.1. 复制槽管理

如早前所提到的，每一个（活跃的）订阅会从远（发布）端上的一个复制槽接收更改。通常，远程复制槽是在使用CREATE SUBSCRIPTION创建订阅是自动创建的，并且在使用DROP SUBSCRIPTION删除订阅时，复制槽也会自动被删除。不过，在一些情况下，有必要单独操纵订阅以及其底层的复制槽。下面是一些场景：

- 在创建一个订阅时，复制槽已经存在。在这种情况下，可以使用create_slot = false选项创建订阅并关联到现有的槽。
- 在创建一个订阅时，远程主机不可达或者处于一种不明状态。在这种情况下，可以使用connect = false选项创建订阅。那么远程主机将根本不会被联系。这是pg_dump所使用的方式。这样，在订阅可以被激活之前，必须手工创建远程复制槽。
- 在删除一个订阅时，复制槽应该被保留。当订阅者数据库正在被移动到一台不同的主机并且将从那里再被激活时，这种行为很有用。在这种情况下，可以在尝试删除该订阅之前，使用ALTER SUBSCRIPTION将复制槽解除关联。

- 在删除一个订阅是，远程主机不可达。在这种情况下，可以在尝试删除该订阅之前，使用ALTER SUBSCRIPTION将复制槽解除关联。如果远程数据库实例不再存在，那么不需要进一步的行动。不过，如果远程数据库实例只是不可达，那么复制槽应该被手动删除。否则它将会继续保留WAL并且最终可能会导致磁盘被填满。这种情况应该要仔细地研究。

31.3. 冲突

逻辑复制的行为类似于正常的DML操作，即便数据在订阅者节点本地被修改，逻辑复制也会根据收到的更改来更新数据。如果流入的数据违背了任何约束，复制将停止。这种情况被称为一个冲突。在复制UPDATE或DELETE操作时，缺失的数据将不会产生冲突并且这类操作将被简单地跳过。

冲突将会产生错误并且停止复制，它必须由用户手工解决。在订阅者的服务器日志中可以找到有关冲突的详细情况。

通过更改订阅者上的数据（这样它就不会与到来的数据发生冲突）或者跳过与已有数据冲突的事务可以解决这种冲突。通过调用pg_replication_origin_advance()函数可以跳过该事务，函数的参数是对应于该订阅名称的node_name以及一个位置。复制源头的当前位置可以在pg_replication_origin_status系统视图中看到。

31.4. 限制

逻辑复制当前有下列限制或者缺失的功能。这些可能在未来的发行中解决。

- 数据库模式和DDL命令不会被复制。初始模式可以手工使用pg_dump --schema-only进行拷贝。后续的模式改变需要手工保持同步（不过值得注意的是，模式其实不需要在两端保持绝对相同）。当一个活跃的数据库中模式定义改变时，逻辑复制是鲁棒的：当模式在发布者上发生改变并且被复制的数据开始到达订阅者但却不适合表模式时，复制将报错，直至模式被更新。在很多情况下，可以通过先对订阅者应用额外的模式更改来避免间歇性的错误。

- 序列数据不被复制。后台由序列支撑的serial或者标识列中的数据当然将被作为表的一部分复制，但是序列本身在订阅者上仍将显示开始值。如果订阅者被用作一个只读数据库，那么这通常不会是什么问题。不过，如果订阅者数据库预期有某种转换或者容错，那么序列需要被更新到最后的值，要么通过从发布者拷贝当前数据的防范（也许使用pg_dump），要么从表本身决定一个足够高的值。

- 支持TRUNCATE命令的复制，但是在截断由外键连接在一起的表群体时必须要小心。在复制截断动作时，订阅者将截断与发布者上被截断的相同的表群体，这些表或者被明确指定或者通过CASCADE隐含地收集而来，然后还要减去不属于该订阅的表。如果所有受影响的表都属于同一个订阅，这会正确地工作。但是如果订阅者上要被截断的某些表有外键链接到不属于同一订阅的表，那么在订阅者上该截断动作的应用将会失败。

- 大对象（见第 35 章）不会被复制。没有办法可以解决这个问题，除非把数据存储在普通表中。

- 复制只能从基表到基表。也就是说，发布端和订阅端上的表都必须是普通表，而不是视图、物化视图、分区根表或者外部表。如果是分区，可以一一对应地复制分区层次，但当前不能复制成一种不同的分区设置。尝试复制不是基表的表将会导致错误。

31.5. 架构

逻辑复制从拷贝发布者数据库上的数据库快照开始。拷贝一旦完成，发布者上的更改会在它们发生时实时传送给订阅者。订阅者按照数据在发布者上被提交的顺序应用数据，这样任意单一订阅中的发布的事务一致性才能得到保证。

逻辑复制被构建在一种类似于物理流复制（见第 26.2.5 节）的架构上。它由“walsender”和“apply”进程实现。walsender进程开始对WAL的逻辑解码

（在第 49 章描述）并且载入标准逻辑解码插件（pgoutput）。该插件把从WAL中读取的更改转换成逻辑复制协议（见第 53.5 节并且根据发布说明过滤数据。然后数据会被连续地使用流复制协议传输到应用工作者，应用工作者会把数据映射到本地表并且以正确的事务顺序应用它们接收到的更改。

订阅者数据库上的应用进程总是将session_replication_role设置为replica运行，这会产生触发器和约束上通常的效果。

逻辑复制应用进程当前仅会引发行触发器，而不会引发语句触发器。不过，初始的表同步是以类似一个COPY命令的方式实现的，因此会引发INSERT的行触发器和语句触发器。

31.5.1. 初始快照

已有的被订阅表中的初始数据会被快照并且以一种特殊类型的应用进程的并行实例进行拷贝。这种进程将创建自己的临时复制槽并且拷贝现有的数据。一旦现有的数据被拷贝完，工作者会进入到同步模式，主应用进程会流式传递在使用标准逻辑复制拷贝初始数据期间发生的任意改变，这会确保表被带到一种已同步的状态。一旦同步完成，该表的复制的控制权会被交回给主应用进程，其中复制会照常继续。

31.6. 监控

因为逻辑复制是基于与物理流复制相似的架构的，一个发布节点上的监控也类似于对物理复制主节点（见第 26.2.5.2 节的监控。

有关订阅的监控信息在pg_stat_subscription中可以看到。每一个订阅工作者在这个视图都有一行。一个订阅能有零个或者多个活跃订阅工作者取决于它的状态。

通常，对于一个已启用的订阅会有单一的应用进程运行。一个被禁用的订阅或者崩溃的订阅在这个视图中不会有行存在。如果有任何表的数据同步正在进行，对正在被同步的表会有额外的工作者。

31.7. 安全性

用于复制连接的角色必须有REPLICATION属性（或者是一个超级用户）。该角色的访问必须被配置在pg_hba.conf中，并且它必须有LOGIN属性。

为了能够拷贝初始表数据，用于复制连接的角色必须在被发布的表上具有SELECT特权（或者是一个超级用户）。

要创建发布，用户必须在数据库中有CREATE特权。

要把表加入到一个发布，用户必须在该表上有拥有权。要创建一个自动发布所有表的发布，用户必须是一个超级用户。

要创建订阅，用户必须是一个超级用户。

订阅的应用过程将在本地数据库上以超级用户的特权运行。

特权检查仅在复制连接开始时被执行一次。在从发布者读到每一个更改记录时不会重新检查特权，在每一个更改被应用时也不会重新检查特权。

31.8. 配置设置

逻辑复制要求设置一些配置选项。

在发布者端，wal_level必须被设置为logical，而max_replication_slots中设置的值必须至少是预期要连接的订阅数加上保留给表同步的连接数。max_wal_senders应该至少被设置为max_replication_slots加上同时连接的物理复制体的数量。

订阅者还要求max_replication_slots被设置。在这种情况下，它必须至少被设置为将被加入到该订阅者的订阅数。max_logical_replication_workers必须至少被设置为订阅数加上保留给表同步的连接数。此外，可能需要调整max_worker_processes以容纳复制工作者，至少为(max_logical_replication_workers + 1)。注意，一些扩展和并行查询也会从max_worker_processes中取得工作者槽。

31.9. 快速设置

首先在postgresql.conf中设置配置选项：

```
wal_level = logical
```

对于一个基础设置来说，其他所需的设置使用默认值就足够了。

需要调整pg_hba.conf以允许复制（这里的值取决于实际的网络配置以及用于连接的用户）：

```
host     all     repuser     0.0.0.0/0     md5
```

然后在发布者数据库上：

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

并且在订阅者数据库上：

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser'
 PUBLICATION mypub;
```

上面的语句将开始复制过程，它会同步表users以及departments的初始表内容，然后开始复制对那些表的增量更改。

第 32 章 即时编译（JIT）

这一章解释什么是即时编译以及如何在PostgreSQL中配置即时编译。

32.1. 什么是JIT编译？

即时（Just-In-Time，JIT）编译是将某种形式的解释程序计算转变成原生程序的过程，并且这一过程是在运行时完成的。例如，与使用能够计算任意SQL表达式的通用代码来计算一个特定的SQL谓词（如WHERE a.col = 3）不同，可以产生一个专门针对该表达式的函数并且可以由CPU原生执行，从而得到加速。

当使用--with-llvm编译PostgreSQL后，PostgreSQL内建支持用LLVM[1]执行JIT编译。

进一步的细节请参考src/backend/jit/README。

32.1.1. JIT加速的操作

当前，PostgreSQL的JIT实现支持对表达式计算以及元组拆解的加速。未来可能有更多其他操作采用这种技术加速。

表达式计算被用来计算WHERE子句、目标列表、聚集以及投影。通过为每一种情况生成专门的代码来实现加速。

元组拆解是把一个磁盘上的元组（见第 68.6.1 节转换成其在内存中表示的过程。通过创建一个专门针对该表布局和要被抽取的列数的函数来实现加速。

32.1.2. 内联

PostgreSQL有很好的扩展性并且允许定义新的数据类型、函数、操作符以及其他数据库对象，见第 38 章实际上，内建对象都使用近乎完全相同的机制来实现。这种可扩展性隐含了一些开销，例如函数调用带来的开销（见第 38.3 节。为了降低这类开销，JIT编译可以把小函数的函数体内联到使用它们的表达式中。这种方式可以优化掉可观的开销。

32.1.3. 优化

LLVM支持对生成的代码进行优化。一些优化代价很低，以至于可以在每次使用JIT时都执行，而另一些优化则只有在运行时间较长的查询中才能获益。更多有关优化的细节请参考https://llvm.org/docs/Passes.html#transform-passes。

32.2. 什么时候会用JIT？

JIT编译主要可以让长时间运行的CPU密集型的查询受益。对于短查询，执行JIT编译增加的开销常常比它节省的时间还要多。

为了判断是否应该使用JIT编译，会用到一个查询的总的估计代价（见第 70 章和第 19.7.2 节。查询的估计代价将与jit_above_cost的设置进行比较。如果代价更高，JIT编译将被执行。然后需要两个进一步的决定。首先，如果估计代价超过jit_inline_above_cost的设置，该查询中使用的短函数和操作符都将被内联。其次，如果估计代价超过jit_optimize_above_cost的设置，会应用昂贵的优化来改进产生的代码。这些选项中的每一种都会增加JIT编译的开销，但是可以可观地降低查询执行时间。

这些基于代价的决定将在规划时做出，而不是在执行时做出。这意味着，在使用预备语句并且使用了一个一般性的计划时（见PREPARE），配置参数的值实际上是在预备时控制这些决定，而不是由执行时的设置来决定。

[1] https://llvm.org/

> **注意**
>
> 如果jit被设置为off或者没有JIT实现可用（例如因为服务器没有用--with-llvm编译），即便基于上述原则能带来很大的好处，JIT也不会被执行。
> 把jit设置成off对规划时和执行时都有影响。

EXPLAIN可以被用来看看是否使用了JIT。例如，这是一个没有使用JIT的查询：

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=0.303..0.303 rows=1
 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.017..0.111 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

看看给出的计划代价，不使用JIT是非常合理的，JIT的代价会比可能得到的节省更高。调整代价限制会导致用到JIT：

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1
 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.019..0.052 rows=356 loops=1)
 Planning Time: 0.133 ms
 JIT:
   Functions: 3
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization 0.797 ms,
 Emission 5.048 ms, Total 7.104 ms
 Execution Time: 7.416 ms
```

如这里所看到的，JIT被用到了，但是内联和昂贵的优化没有被用到。如果jit_inline_above_cost或者jit_optimize_above_cost也被降低，这种情况会被改变。

32.3. 配置

配置变量jit决定启用或者禁用JIT编译。如果它被启用，配置变量jit_above_cost、jit_inline_above_cost以及jit_optimize_above_cost判断是否要为一个查询执行JIT编译以及在执行中花费多大的努力。

jit_provider决定使用哪一种JIT实现。很少需要改变这一设置。见第 32.4.2 节

如第 19.17 节所述，对于开发和调试目的，还有一些额外的配置参数存在。

32.4. 可扩展性

32.4.1. 对扩展的内联支持

PostgreSQL的JIT实现可以内联C以及internal类型的函数体，还有基于这类函数的操作符。为了能对扩展中的函数这样做，需要让那些函数的定义可用。在使用PGXS对一个已经编译有LLVM JIT支持的服务器构建一个扩展时，相关的文件将被自动构建并且安装。

相关的文件必须被安装在$pkglibdir/bitcode/$extension/中并且对它们的一份概要必须被安装在$pkglibdir/bitcode/$extension.index.bc中，其中$pkglibdir是pg_config --pkglibdir返回的目录里，而$extension是扩展的共享库的基础名称。

> **注意**
>
> 对于编译在PostgreSQL本身中的函数，其bitcode被安装在$pkglibdir/bitcode/postgres。

32.4.2. 可插拔的JIT提供者

PostgreSQL提供一种基于LLVM的JIT实现。JIT提供者的接口是可插拔的，可以无需重编译就能改变提供者（尽管当前构建过程仅提供了对LLVM的内联支持数据）。活跃的提供者通过设置jit_provider来选择。

32.4.2.1. JIT提供者接口

JIT提供者需要通过动态装载其共享库来载入。正常的搜索路径被用来定位该库。为了提供所要求的JIT提供者回调并且表示该库实际上是一个JIT提供者，它需要提供一个名为_PG_jit_provider_init的C函数。会有一个结构被传入这个函数，在函数中应该用回调函数指针来填充该结构：

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};

extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

第 33 章 回归测试

回归测试是PostgreSQL中对于 SQL 实现的一组综合测试集。它们测试标准 SQL 操作以及PostgreSQL的扩展能力。

33.1. 运行测试

回归测试可以在一个已经安装并运行的服务器上运行，或者在编译树中的一个临时安装上运行。此外，还有运行该测试的“并行”和“顺序”模式。顺序方法单独运行每一个测试脚本，而并行方法则开启多个服务器进程来并行地运行多组测试。并行测试能够发现进程间通信和锁定是否工作正确。

33.1.1. 在一个临时安装上运行测试

要在编译之后且在安装之前运行并行回归测试，可在顶层目录中键入：

```
make check
```

（或者你可以切换到src/test/regress并且在那里运行该命令）。最后你应该看到这样的信息：

```
=======================
 All 115 tests passed.
=======================
```

或者关于哪些测试失败的提示。见下面的第 33.2 节来确定一个“失败”是否表示一个严重的问题。

因为这种测试方法运行一个临时服务器，如果你作为根用户进行了编译，它将无法工作，因为服务器无法用 root 启动。我们推荐的过程是不要作为 root 编译，或者在完成安装后执行测试。

如果你已经配置PostgreSQL安装到一个已经存在有旧的PostgreSQL安装的位置，并且你在安装新版本前执行了make check，你可能会发现测试会因为新程序尝试使用已经安装的共享库而失败（典型特征是抱怨未定义的符号）。如果你希望在覆盖旧安装之前运行测试，你将需要使用configure --disable-rpath编译。但是我们不推荐为最终安装使用这个选项。

并行回归测试会在你的用户 ID 下启动相当多的进程。当前，最大并发量是二十个并行测试脚本，这意味着四十个进程：对每一个测试脚本有一个服务器进程和一个psql进程。因此如果你的系统对每个用户的进程数有强制限制，确保这个限制至少是五十，否则你将在并行测试中失败。如果你没有权利提升该限制，你可以通过设置MAX_CONNECTIONS参数来降低并发度。例如：

```
make MAX_CONNECTIONS=10 check
```

会并发运行不超过十个测试。

33.1.2. 在一个现有安装上运行测试

要在安装后运行测试（见第 16 章，初始化一个数据区域并且按照第 18 章解释的启动服务器，然后输入：

```
make installcheck
```

或者进行一次并行测试：

```
make installcheck-parallel
```

该测试将期望联系在本地主机和默认端口号上的服务器（除非通过PGHOST和PGPORT环境变量覆盖）。该测试将在一个名为regression的数据库中运行，任何以该名称存在的数据库将被删除。

该测试还将短暂地创建一些集簇范围内的对象，例如角色和表空间。这些对象的名称都会以regress_开始。在实际具有以这种方式命名对象的安装中使用installcheck模式时要格外小心。

33.1.3. 附加测试套件

make check和make installcheck命令只运行“核心的”回归测试，这只测试PostgreSQL服务器的内建功能。源代码发布也包含额外的测试套件，它们中的大部分用于测试附加功能，例如可选的过程语言。

要运行将被编译模块的所有测试套件（包括核心测试），在编译树的顶端输入这些命令之一：

```
make check-world
make installcheck-world
```

这些命令分别在临时服务器或已经安装好的服务器上运行测试（与之前介绍的make check和make installcheck类似）。其他的考虑与之前为每种方法解释的相同。注意make check-world为每一个受测模块建立一个独立的临时安装树，因此它比起make installcheck-world需要更多的时间和磁盘空间。

你也可以通过在编译树适当的子目录中输入make check或make installcheck来运行个体的测试套件。记住make installcheck假设你已经安装了相关模块，而不仅仅是核心服务器。

可以以这种方法调用的额外测试包括：

- 可选过程语言的回归测试（除PL/pgSQL之外，它将被核心测试测试）。这些位于src/pl之下。
- contrib模块的回归测试，位于contrib。不是所有的contrib模块都有测试。
- ECPG 接口库的回归测试，位于src/interfaces/ecpg/test。
- 并发会话行为的压力测试，位于src/test/isolation。
- 客户端的测试程序在src/bin下。另见 第 33.4 节

在使用installcheck模式时，这些测试将毁掉任何现有的名为pl_regression、contrib_regression、isolation_regression、ecpg1_regression或者ecpg2_regression的数据库，以及regression。

只有在PostgreSQL被使用选项--enable-tap-tests配置时，基于TAP的测试才能被运行。推荐在开发时使用这种方式，但如果没有合适的Perl安装可用也可以忽略。

一些测试套件默认不会被运行，因为它们在多用户系统上运行不安全或者它们需要特殊的软件。可以通过设置make或环境变量为空格分隔的列表来决定额外运行哪些测试套件，例如：

```
make check-world PG_TEST_EXTRA='kerberos ldap ssl'
```

当前支持下列值：

kerberos

运行src/test/kerberos下的测试套件。这要求一个MIT Kerberos安装并且打开TCP/IP监听端口。

ldap

运行src/test/ldap下的测试套件。这要求一个OpenLDAP安装并且打开TCP/IP监听端口。

ssl

运行src/test/ssl下的测试套件。这会打开TCP/IP监听端口。

即便在PG_TEST_EXTRA中提到了当前编译配置不支持的特性，针对它们的测试也不会被运行。

33.1.4. 区域和编码

默认情况下，测试使用的临时安装将使用在当前环境中定义的区域和由initdb决定的相应数据库编码。通过设置适当的环境变量来测试不同的区域是有用的，例如：

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

由于实现的原因，为此目的设置LC_ALL不能工作，所有其他区域相关的环境变量都可以工作。

在对一个现有安装测试时，区域由现有数据库集簇决定并且不能为测试而独立设置。

你也可以通过设置变量ENCODING来显式地选择数据库编码，例如：

```
make check LANG=C ENCODING=EUC_JP
```

这样设置数据库编码通常只对区域为 C 有意义；否则编码将自动从区域选择，并且指定一个不匹配区域的编码将会导致错误。

不管测试是针对临时安装还是已有安装，数据库编码都可以被设置，然而在后一种情况中它必须与安装的区域相兼容。

33.1.5. 额外测试

核心回归测试套件包含一些默认情况下不被运行的测试文件，因为它们可能平台相关的或者需要很长时间来运行。你可以通过设置变量EXTRA_TESTS来运行这些或者其他额外测试文件。例如，要运行numeric_big测试：

```
make check EXTRA_TESTS=numeric_big
```

要运行排序规则测试：

```
make check EXTRA_TESTS='collate.icu.utf8 collate.linux.utf8' LANG=en_US.utf8
```

collate.linux.utf8测试只在 Linux/glibc 平台上能够工作。只有在编译了ICU支持时，collate.icu.utf8测试才能工作。两种测试只有在使用 UTF-8 编码的数据库中才能成功运行。

33.1.6. 测试热备

源代码发布中还包含有用于热备的静态行为的回归测试。这些测试需要一个运行着的主服务器和一个运行着的后备服务器，并且后备服务器正从主服务器接受新的 WAL 改变（使用基于文件的日志传送或流复制）。那些服务器不是自动创建的，这里也没有关于建立复制的文档。请查阅本文档中的相关章节。

要运行热备测试，首先在主服务器上创建一个名为regression的数据库：

```
psql -h primary -c "CREATE DATABASE regression"
```

接下来，在主服务器上的 regression 数据库中运行准备脚本src/test/regress/sql/hs_primary_setup.sql，例如：

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

允许这些改变传播到后备服务器。

为受测后备服务器安排默认数据库连接（例如通过设置PGHOST和PGPORT环境变量）。最后，在 regression 目录中运行make standbycheck：

```
cd src/test/regress
make standbycheck
```

在主服务器上也可以使用src/test/regress/sql/hs_primary_extremes.sql脚本生成某些极限行为来允许测试后备服务器的行为。

33.2. 测试评估

一些正确安装的并且全功能的PostgreSQL安装可能会在这些回归测试中的某些上“失败”，其原因是平台相关的因素，例如可变浮点表示和 message wording。这些测试目前采用diff命令来比较测试输出和在参考系统上产生的输出，这样测试的结果对小的系统差异也很敏感。当一个测试被报告为“失败”时，请总是检查实际结果和期望结果之间的差异，你可能会发现该差异其实并不明显。不管怎样，我们将努力维护在所有被支持平台上的准确的参考文件，以期待所有的测试都能通过。

回归测试的实际输出在src/test/regress/results目录中的文件内。测试脚本会使用diff来把每一个输出文件与存储在src/test/regress/expected目录中的参考输出进行比较。任何差异都被保存在src/test/regress/regression.diffs中便于你的观察（当运行一个除核心测试之外的测试套件时，这些文件当然会出现在相关子目录中，而不是src/test/regress）。

如果你不喜欢被默认使用的diff选项，请设置环境变量PG_REGRESS_DIFF_OPTS，例如PG_REGRESS_DIFF_OPTS='-u'（或者如果你愿意，你可以自己运行diff）。

如果由于某种原因一个特定的平台对一个给定测试产生了“失败”，而对输出的检查却说明该结果是合法的，你可以增加一个新的比较文件来让失败报告在未来的测试运行中保持沉默。详见第 33.3 节

33.2.1. 错误消息差异

某些回归测试涉及到故意的非法输入值。错误消息可能来自PostgreSQL代码或主机平台系统例程。在后一种情况中，消息会随着平台而变化，但是会反映相似的信息。这些消息中的差异将导致一次“失败的”回归测试，这可以通过检查来确认。

33.2.2. 区域差异

如果你在一台使用除 C 之外的排序规则顺序区域初始化的服务器上运行测试，那么可能会出现由于排序顺序和后续失败产生的差异。回归测试套件被设置为可以处理这种问题，方法是提供替代的结果文件来处理大量的区域。

要在使用临时安装方法时在一种不同的区域中运行测试，可在make命令行上传递适当的区域相关的环境变量，例如：

```
make check LANG=de_DE.utf8
```

（回归测试驱动器会取消LC_ALL设置，因此使用这个变量选择区域是不起作用的）。要不使用区域，要么取消所有区域相关的环境变量设置（或把它们设置为C），要么使用下列特殊调用：

```
make check NO_LOCALE=1
```

当对一个现有安装运行测试时，区域设置由现有安装决定。要改变它，通过向initdb传递合适的选项来使用不同的区域初始化数据库集簇。

通常，我们建议对将要在生产环境中使用的区域设置运行回归测试，因为这样可以测试即将真正被用在生产环境中的与区域和编码相关的代码。根据　操作系统环境，你可能会得到失败，但是那样你将至少知道在真实应用运行时会得到什么样的与区域相关的行为。

33.2.3. 日期和时间差异

大部分的日期和时间结果依赖于时区环境。参考文件是用时区PST8PDT（伯克利，加利福利亚）生成的，并且如果测试不是运行在该时区设置中显然会出现失败。回归测试驱动器会设置环境变量PGTZ为 PST8PDT，这通常能保证正确的结果。

33.2.4. 浮点差异

某些测试涉及到从表列中计算 64 位浮点数（双精度）。我们已经发现了涉及到双精度列的数学函数的结果中的差异。float8和geometry测试容易在不同平台之间产生小的差异，甚至对不同的编译器优化设置也可能产生差异。这些差异通常位于小数点右边的 10 个位置，决定这些差异的实际意义需要人类眼球比较。

某些系统显示负零为-0，而其他的只显示0。

某些系统标志来自pow()和exp()的错误的机制不同于当前PostgreSQL代码所期望的机制。

33.2.5. 行序差异

你可能看到这样一些差异：一组相同的行在输出中的顺序与参考文件中的顺序不同。严格来说，在大部分情况下这不是缺陷。大部分回归测试脚本没有为每一个单独的SELECT使用一个ORDER BY，并且因此它们的结果行顺序根据 SQL 规范是非良定义的。实际上，因为我们考虑的是由相同的软件在相同的数据上执行相同的查询，我们通常会在所有平台上得到相同的结果顺序，所以缺少ORDER BY不是一个问题。但是，某些查询确实会在不同平台上产生不同的顺序。当对一个已经安装的服务器运行测试时，顺序差异可能由非 C 区域设置或非默认参数设置导致，例如work_mem的自定义值或规划器代价参数。

因此，如果你看到一个顺序差异，没有什么可担心的，除非结果被未被的查询确实有一个ORDER BY。但是，不管怎样请报告它，这样我们可以为特定的查询加上一个ORDER BY来在未来的发布中消除虚假的“失败”。

你可能好奇为什么我们不对所有回归测试查询进行显式排序来一次性解决这个问题。其原因是那可能会降低回归测试的有用性，因为它们已经倾向于测试产生有序结果的查询计划类型而排除了那些无法产生有序结果的计划类型。

33.2.6. 栈深度不足

如果错误测试导致了在select infinite_recurse()命令上的一次服务器崩溃，它意味着平台对进程栈尺寸的限制低于max_stack_depth参数所指定的值。这可以通过在一个更高的栈尺寸限制（对max_stack_depth的默认值，我们推荐 4 MB）下运行该服务器来修复。如果你不能这样做，一种可替代的方案是减小max_stack_depth的值。

在支持getrlimit()的平台上，服务器应该自动选择一个max_stack_depth的安全值。所以除非你已经手工覆盖了该设置，这类失败就是一个可报告的缺陷。

33.2.7. “随机”测试

随机测试脚本用来产生随机结果。在非常少见的情况下，这会导致回归测试失败。输入：

```
diff results/random.out expected/random.out
```

应当产生一行或少数几行差异。你不需要担心，除非随机测试重复地失败。

33.2.8. 配置参数

当对一个现有安装运行测试时，某些非默认参数设置可能导致测试失败。例如，改变enable_seqscan或enable_indexscan等参数可能导致计划改变，然后影响使用EXPLAIN的测试的结果。

33.3. 变体比较文件

因为某些测试生来就会产生依赖环境的结果，我们提供了方法来指定替代的“预期”结果文件。每一个回归测试可以有多个比较文件来展示在不同平台上的可能结果。有两种独立的机制来决定为每一个测试使用哪个比较文件。

第一种机制允许为指定平台选择比较文件。这是一个映射文件src/test/regress/resultmap，它定义了为每一个平台使用哪个比较文件。要为一个特定平台消除虚假的测试“失败”，你可以首先选择或创建一个变体结果文件，然后在resultmap文件中增加一行。

在该映射文件中的每一行的形式为：

```
testname:output:platformpattern=comparisonfilename
```

测试名只是该特定回归测试模块的名称。输出值指定要检查哪个输出文件。对于标准回归测试，这总是out。该值对应于输出文件的文件扩展。平台模式是一个 Unix 工具expr风格的模式（即在开头带有一个隐式^锚的正则表达式）。它被与config.guess打印出的平台名称进行匹配。匹配文件名称是替补的结果比较文件的基础名。

例如：某些系统会把非常小的浮点值解释为零，而不是报告一个下溢错误。这在float8回归测试中会导致一些差异。因此，我们提供一个变体比较文件float8-small-is-zero.out，其中包括了在这些系统上的期望结果。要在OpenBSD平台上屏蔽这种虚假的“失败”消息，resultmap包括：

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

这将在任何config.guess输出匹配i.86-.*-openbsd的机器上触发。resultmap中的其他行为其他平台选择变体比较文件。

第二种变体比较文件的选择机制更加自动：它简单地在多个提供的比较文件中采用“最佳匹配”。回归测试驱动器脚本对一个测试考虑两种标准比较文件，testname.out以及名

为testname_digit.out的变体文件（其中digit是任何单一数字0-9）。如果任一这种文件是一个完全匹配，测试被认为是通过的。否则，产生最短区别的文件被用来创建失败报告（如果resultmap包括特定测试的一个项，那么基础testname是resultmap中给定的替补名称）。

例如，对于char测试，比较文件char.out包含在C和POSIX区域中期望的结果，而文件char_1.out包含在其他很多区域中的排序结果。

最佳匹配机制被设计为与区域依赖的结果协同工作，但是它可以被用在任何测试结果无法只从平台名很容易地预测的情况中。这种机制的一个限制是测试驱动器不能说出哪个变体对当前环境是真正“正确的”，它将只是选择看起来工作得最好的变体。因此对你认为在所有上下文中具有同等合法性的变体结果使用这种机制才是最安全的。

33.4. TAP 测试

很多测试，特别是src/bin下面的客户端程序测试使用 Perl 的 TAP 工具并且用Perl测试程序prove运行。你可以通过 设置make变量PROVE_FLAGS 向prove传递命令行选项，例如：

```
make -C src/bin check PROVE_FLAGS='--timer'
```

详见prove的手册页。

make变量PROVE_TESTS可被用来定义一个空格分隔的列表，其中是调用prove来运行的指定测试子集的路径，这些测试子集将取代默认的t/*.pl，并且这些路径是相对于Makefile的。例如：

```
make check PROVE_TESTS='t/001_test1.pl t/003_test3.pl'
```

TAP测试需要 Perl 模块IPC::Run。 这个模块可以从 CPAN 或者一个操作系统包得到。

33.5. 测试覆盖检查

PostgreSQL 源代码可以使用覆盖测试指令编译，因此可以检查哪些部分的代码被回归测试或任何其他测试套件所覆盖。当前使用 GCC 编译时支持该特性，并且需要gcov和lcov程序。

一个典型的工作流程看起来是：

```
./configure --enable-coverage ... OTHER OPTIONS ...
make
make check # 或其他测试套件
make coverage-html
```

然后将你的 HTML 浏览器指向coverage/index.html。make命令在子目录中也能工作。

如果没有lcov或者更喜欢文本输出而不是HTML报告，还可以运行

```
make coverage
```

来取代make coverage-html，它将为每个与测试相关的源文件产生.gcov输出文件（make coverage和make coverage-html将覆盖彼此的文件，所以把它们混合在一起可能会导致混乱）。

要在多次测试运行之间重置执行计数，运行：

```
make coverage-clean
```

部分 IV. 客户端接口

这一部分描述和PostgreSQL一起发布的客户端编程接口。这些章中的每一个都能被独立阅读。注意，还有很多用于客户端程序的其他编程接口是被独立发布的并且包含它们自己的文档（附录 I列出了一些很流行的）。这部份的读者应该熟悉使用SQL命令来操纵和查询数据库（见第 II 部分，以及熟悉接口所使用的编程语言。

目录

第 34 章 libpq - C 库

libpq是应用程序员使用PostgreSQL的C接口。libpq是一个库函数的集合，它们允许客户端程序传递查询给PostgreSQL后端服务器并且接收这些查询的结果。

libpq也是很多其他PostgreSQL应用接口的底层引擎，包括为 C++、Perl、Python、Tcl 和 ECPG编写的接口。如果你使用那些包，某些方面的libpq行为将会对你很重要。特别是，第 34.14 节第 34.15 和第 34.18 描述了任何使用libpq的应用的用户可见的行为。

在本章的末尾（第 34.21 节）包括了一些短程序来展示如何编写使用libpq的应用。在源代码发布的src/test/examples目录中还有一些完整的libpq应用的例子。

使用libpq的客户端程序必须包括头文件libpq-fe.h并必须与libpq库链接在一起。

34.1. 数据库连接控制函数

下列函数会建立到一个PostgreSQL后端服务器的连接。一个应用程序可以在一个时刻打开多个后端连接（原因之一就是为了访问多个数据库）。每个连接用一个PGconn对象表示，它从函数PQconnectdb、PQconnectdbParams或PQsetdbLogin得到。注意这些函数将总是返回一个非空的对象指针，除非正好没有内存来分配PGconn对象。在通过该连接对象发送查询之前，应该调用PQstatus函数来检查返回值以确定是否得到了一个成功的连接。

> **警告**
>
> 如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，开始每个会话时应该从search_path中移除公共可写的方案。我们可以把参数关键词options设置为-csearch_path=。也可以在连接后发出PQexec(conn, "SELECT pg_catalog.set_config('search_path', '', false)")。这种考虑并非专门针对libpq，它适用于每一种用来执行任意SQL命令的接口。

> **警告**
>
> 在 Unix 上，复制一个拥有打开 libpq 连接的进程可能导致不可以预料的结果，因为父进程和子进程会共享相同的套接字和操作系统资源。出于这个原因，我们不推荐这样的用法，尽管从子进程执行一个exec来载入新的可执行代码是安全的。

> **注意**
>
> 在 Windows 上，如果一个单一数据库连接被反复地开启并且关闭，这是一种提升性能的方式。在内部，libpq 为开启和关闭分别调用WSAStartup()和WSACleanup()。WSAStartup()会增加一个内部 Windows 库引用计数而WSACleanup()则会减少之。当引用计数正好为一时，调用WSACleanup()会释放所有资源并且所有 DLL 会被卸载。这是一种昂贵的操作。为了避免它，一个应用可以手动调用WSAStartup()，这样当最后的数据库连接被关闭时资源不会被释放。

PQconnectdbParams

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

这个函数使用从两个以NULL结尾的数组中取得的参数打开一个新的数据库连接。第一个数组keywords被定义为一个字符串数组，每一个元素是一个关键词。第二个数组values给出了每个关键词的值。和下面的PQsetdbLogin不同，可以在不改变函数签名的情况下扩展参数集合，因此使用这个函数（或者与之相似的非阻塞的PQconnectStartParams和PQconnectPoll）对于新应用编程要更好。

当前能被识别的参数关键词被列举在第 34.1.2 节。

当expand_dbname为非零时，dbname关键词的值被允许识别为一个连接字符串。只有dbname的第一次出现会按这种方式扩展，任何后续dbname值会被当做一个普通数据库名处理。有关可能的连接字符串格式的详情可见第 34.1.1 节

被传递的数组可以为空，这样就会使用所有默认参数。也可以只包含一个或几个参数设置。两个参数数组应该在长度上匹配。对于参数数组的处理将会停止于keywords数组中第一个非-NULL元素。

如果任何一个参数是NULL或者一个空字符串，那么将会检查相应的环境变量（见第 34.14 节。如果该环境变量也没有被设置，那么将使用内建的默认值。

通常，关键词的处理是从这些数组的头部开始并且以索引顺序进行。这样做的效果就是，当关键词有重复时，只会保留最后一个被处理的值。因此，通过小心地放置关键词dbname，可以决定什么会被一个conninfo字符串所重载，以及什么不会被重载。

PQconnectdb

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdb(const char *conninfo);
```

这个函数使用从字符串conninfo中得到的参数开启一个新的数据库连接。

被传递的字符串可以为空，这样将会使用所有的默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个URI。详见第 34.1.1 节

PQsetdbLogin

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

这是PQconnectdb的带有固定参数集合的前辈。它具有相同的功能，不过其中缺失的参数将总是采用默认值。对任意一个固定参数写NULL或一个空字符串将会使它采用默认值。

如果dbName包含一个=符号或者具有一个合法的连接URI前缀，它会被当作一个conninfo字符串，就好像它已经被传递给了PQconnectdb，并且剩余的参数则被应用为指定给PQconnectdbParams。

PQsetdb

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

这是一个调用PQsetdbLogin的宏，其中为login和pwd参数使用空指针。提供它是为了向后兼容非常老的程序。

PQconnectStartParams
PQconnectStart
PQconnectPoll

以非阻塞的方式建立一个到数据库服务器的连接。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

这三个函数被用来开启一个到数据库服务器的连接，这样你的应用的执行线程不会因为远程的I/O而被阻塞。这种方法的要点在于等待 I/O 完成可能在应用的主循环中发生，而不是在PQconnectdbParams或PQconnectdb中，并且因此应用能够把这种操作和其他动作并行处理。

在PQconnectStartParams中，数据库连接使用从keywords和values数组中取得的参数创建，并且被expand_dbname控制，这和之前描述的PQconnectdbParams相同。

在PQconnectStart中，数据库连接使用从字符串conninfo中取得的参数创建，这和之前描述的PQconnectdb相同。

只要满足一些限制，PQconnectStartParams或PQconnectStart或PQconnectPoll都不会阻塞：

- hostaddr和host参数必须被合适地使用，以防止做DNS查询。详见第 34.1.2 节这些参数的文档。
- 如果你调用PQtrace，确保你追踪的该流对象不会阻塞。
- 如后文所述，你必须要确保在调用PQconnectPoll之前，套接字处于合适的状态。

要开始无阻塞的连接请求，可调用PQconnectStart或者PQconnectStartParams。如果结果为空，则libpq无法分配一个新的PGconn结构。否则，一个有效的PGconn指针会被返回（不过还没有表示一个到数据库的有效连接）。接下来调用PQstatus(conn)。如果结果是CONNECTION_BAD，则连接尝试已经失败，通常是因为有无效的连接参数。

如果PQconnectStart成功，下一个阶段是轮询libpq，这样它能够继续进行连接序列。使用PQsocket(conn)来获得该数据库连接底层的套接字描述符（警告：不要假定在PQconnectPoll调用之间套接字会保持相同）。这样循环：如果PQconnectPoll(conn)上一次返回PGRES_POLLING_READING，等到该套接字准备好读取（按照select()、poll()或类似的系统函数所指示的）。则再次调用PQconnectPoll(conn)。反之，如果PQconnectPoll(conn)上一次返回PGRES_POLLING_WRITING，等到该套接字准备好写入，则再次调用PQconnectPoll(conn)。在第一次迭代时，即如果你还没有调用PQconnectPoll，行为就像是它上次返回了PGRES_POLLING_WRITING。持续这个循环直

到PQconnectPoll(conn)返回PGRES_POLLING_FAILED指示连接过程已经失败，或者返回PGRES_POLLING_OK指示连接已经被成功地建立。

在连接期间的任意时刻，该连接的状态可以通过调用PQstatus来检查。如果这个调用返回CONNECTION_BAD，那么连接过程已经失败。如果该调用返回CONNECTION_OK，则该连接已经准备好。如前所述，这些状态同样都可以从PQconnectPoll的返回值检测。在一个异步连接过程中（也只有在这个过程中）也可能出现其他状态。这些状态指示该连接过程的当前阶段，并且可能有助于为用户提供反馈。这些状态是：

CONNECTION_STARTED

等待连接被建立。

CONNECTION_MADE

连接 OK，等待发送。

CONNECTION_AWAITING_RESPONSE

等待来自服务器的一个回应。

CONNECTION_AUTH_OK

收到认证，等待后端启动结束。

CONNECTION_SSL_STARTUP

协商 SSL 加密。

CONNECTION_SETENV

协商环境驱动的参数设置。

CONNECTION_CHECK_WRITABLE

检查连接是否能够处理写事务。

CONNECTION_CONSUME

消费连接上的任何剩下的响应消息。

注意，尽管这些常数将被保留（为了维护兼容性），一个应用永远不应该依赖这些状态按照特定顺序出现，或者根本就不依赖它们，或者不依赖状态总是这些文档中所说的值。一个应用可能做些这样的事情：

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Connecting...";
            break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
            feedback = "Connecting...";
}
```

在使用PQconnectPoll时，连接参数connect_timeout会被忽略：判断是否超时是应用的责任。否则，PQconnectStart后面跟着后面跟着PQconnectPoll循环等效于PQconnectdb。

注意当PQconnectStart或PQconnectStart返回一个非空的指针时，你必须在用完它之后调用PQfinish来处理那些结构和任何相关的内存块。即使连接尝试失败或被放弃时也必须完成这些工作。

PQconndefaults

返回默认连接选项。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* 该选项的关键词 */
    char   *envvar;    /* 依赖的环境变量名 */
    char   *compiled;  /* 依赖的内建默认值 */
    char   *val;       /* 选项的当前值，或者 NULL */
    char   *label;     /* 连接对话框中域的标签 */
    char   *dispchar;  /* 指示如何在一个连接对话框中显示这个域。值是：
                          ""        显示输入的值
                          "*"       口令域 - 隐藏值
                          "D"       调试选项 - 默认不显示 */
    int     dispsize;  /* 用于对话框的以字符计的域尺寸 */
} PQconninfoOption;
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的PQconnectdb选项和它们的当前缺省值。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。如果无法分配内存，则返回该空指针。注意当前缺省值（val域）将依赖于环境变量和其他上下文。一个缺失或者无效的服务文件将会被无声地忽略掉。调用者必须把连接选项当作只读对待。

在处理完选项数组后，把它交给PQconninfoFree释放。如果没有这么做，每次调用PQconndefaults都会导致一小部分内存泄漏。

PQconninfo

返回被一个活动连接使用的连接选项。

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的PQconnectdb选项和它们的当前缺省值。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。上述所有对于PQconndefaults的注解也适用于PQconninfo的结果。

PQconninfoParse

返回从提供的连接字符串中解析到的连接选项。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

解析一个连接字符串并且将结果选项作为一个数组返回，或者在连接字符串有问题时返回NULL。这个函数可以用来抽取所提供的连接字符串中的PQconnectdb选项。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。

所有合法选项将出现在结果数组中，但是任何在连接字符串中没有出现的选项的PQconninfoOption的val会被设置为NULL，默认值不会被插入。

如果errmsg不是NULL，那么成功时*errmsg会被设置为NULL，　否则设置为被malloc过的错误字符串以说明该问题（也可以将*errmsg设置为NULL并且函数返回NULL，这表示一种内存耗尽的情况）。

在处理完选项数组后，把它交给PQconninfoFree释放。如果没有这么做，　　每次调用PQconninfoParse都会导致一小部分内存泄漏。反过来，如果发生一个错误并且errmsg不是NULL，确保使用PQfreemem释放错误字符串。

PQfinish

关闭与服务器的连接。同时释放PGconn对象使用的内存。

```
void PQfinish(PGconn *conn);
```

注意，即使与服务器的连接尝试失败（由PQstatus指示），应用也应当调用PQfinish来释放PGconn对象使用的内存。不能在调用PQfinish之后再使用PGconn指针。

PQreset

重置与服务器的通讯通道。

```
void PQreset(PGconn *conn);
```

此函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。

PQresetStart
PQresetPoll

以非阻塞方式重置与服务器的通讯通道。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

这些函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。它们和上面的PQreset的不同在于它们工作在非阻塞方式。这些函数受到PQconnectStartParams、PQconnectStart和PQconnectPoll相同的限制。

要发起一次连接重置，调用PQresetStart。如果它返回　　0，那么重置失败。如果返回1，用与使用PQresetPoll建立连接的相同方法使用PQconnectPoll重置连接。

PQpingParams

PQpingParams报告服务器的状态。它接受与PQconnectdbParams相同的连接参数。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

该函数返回下列值之一：

PQPING_OK

服务器正在运行，并且看起来可以接受连接。

PQPING_REJECT

服务器正在运行，但是处于一种不允许连接的状态（启动、关闭或崩溃恢复）。

PQPING_NO_RESPONSE

无法联系到服务器。这可能表示服务器没有运行，或者给定的连接参数中有些错误（例如，错误的端口号），或者有一个网络连接问题（例如，一个防火墙阻断了连接请求）。

PQPING_NO_ATTEMPT

没有尝试联系服务器，因为提供的参数显然不正确，或者有一些客户端问题（例如，内存用完）。

PQping

PQping报告服务器的状态。它接受与PQconnectdb相同的连接参数。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQping(const char *conninfo);
```

返回值和PQpingParams相同。

34.1.1. 连接字符串

几个libpq函数会解析一个用户指定的字符串来获得连接参数。这些字符串有两种被接受的格式：纯关键词 = 值字符串以及URI。URI通常遵循RFC 3986[1]，除非像下文进一步描述的那样允许多主机连接字符串。

34.1.1.1. 关键词/值连接字符串

在第一种格式中，每一个参数设置的形式都是关键词 = 值。等号周围的空白是可选的。要写一个空值或一个包含空白的值，将它用单引号包围，例如关键词 = 'a value'。值里面的单引号和反斜线必须用一个反斜线转义，即\'和\\。

例子：

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

能被识别的参数关键词在第 34.1.2 节列出。

34.1.1.2. 连接 URI

一个连接URI的一般形式是：

```
postgresql://[user[:password]@][netloc][:port][,...][/dbname][?
param1=value1&...]
```

URI模式标志符可以是postgresql://或postgres://。每一个URI部分都是可选的。下列例子展示了合法的URI语法：

[1] https://tools.ietf.org/html/rfc3986

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?
target_session_attrs=any&application_name=myapp
```

URI的层次部分的组件可以作为参数给出。例如：

```
postgresql:///mydb?host=localhost&port=5433
```

在任意URI部分中可以使用百分号编码来包括有特殊含义的符号，例如用%3D替换=。

任何不对应第 34.1.2 节列出的关键词的连接参数将被忽略并且关于它们的警告消息会被发送到stderr。

为了提高和 JDBC 连接URI的兼容性，参数ssl=true的实例会被翻译成sslmode=require。

主机部分可能是主机名或一个 IP 地址。要指定一个 IPv6 主机地址，将它封闭在方括号中：

```
postgresql://[2001:db8::1234]/database
```

主机组件会被按照参数host对应的描述来解释。特别地，如果主机部分是空或开始于一个斜线，将使用一个 Unix 域套接字连接，否则将启动一个 TCP/IP 连接。不过要注意，斜线是 URI 层次部分中的一个保留字符。因此，要指定一个非标准的 Unix 域套接字目录，要么忽略 URI 中的主机说明并且指定该主机为一个参数，要么在 URI 的主机部分用百分号编码路径：

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

可以在一个URI中指定多个主机，每一个都有一个可选的端口。一个形式为postgresql://host1:port1,host2:port2,host3:port3/的URI等效于host=host1,host2,host3 port=port1,port2,port3形式的连接字符串。每一个主机都将被尝试，直到成功地建立一个连接。

34.1.1.3. 指定多个主机

可以指定多个要连接的主机，这样它们会按给定的顺序被尝试。在键/值格式中，host、hostaddr和port选项都接受逗号分隔的值列表。在指定的每一个选项中都必须给出相同数量的元素，这样第一个hostaddr对应于第一个主机名，第二个hostaddr对应于第二个主机名，以此类推。不过，如果仅指定一个port，它将被应用于所有的主机。

在连接URI格式中，在URI的host部分我们可以列出多个由逗号分隔的host:port对。

不管是哪一种格式，单一的主机名可以被翻译成多个网络地址。常见的例子是一个主机同时具有IPv4和IPv6地址。

当多个主机被指定时或者单个主机名被翻译成多个地址时，所有的主机和地址都将按照顺序被尝试，直至遇到一个成功的。如果没有主机可以到达，则连接失败。如果成功地建立一个连接但是认证失败，也不会尝试列表中剩下的主机。

如果使用了口令文件，可以为不同的主机使用不同的口令。所有其他连接选项对列表中的每一台主机都是相同的，例如不能为不同的主机指定不同的用户名。

34.1.2. 参数关键词

当前被识别的参数关键词是：

host

要连接的主机名。如果一个主机名以斜线开始，则表示一个 Unix 域通信而不是 TCP/IP 通信，其值是存储套接字文件的目录名。当host没有指定或者为空时的默认行为是连接到一个/tmp（或者PostgreSQL编译时指定的任何套接字目录）中的 Unix 域套接字。在没有 Unix 域套接字的机器上，默认是连接到localhost。

也接受由逗号分隔的主机名列表，这种情况下将依次尝试列表中的主机名，列表中的空项会选择上述的默认行为。详情请参考第 34.1.1.3 节

hostaddr

要连接的主机的数字 IP 地址。它应该是标准的 IPv4 地址格式，例如172.28.40.9。如果你的机器支持 IPv6，你也可以使用那些地址。当为这个参数指定一个非空字符串时，总是会使用 TCP/IP 通信。

使用hostaddr代替host允许应用能避免一次主机名查找，这对于具有时间约束的应用可能非常重要。不过，GSSAPI 或 SSPI 认证方法以及verify-full SSL 证书验证还是要求一个主机名。使用的是下列规则：

- 如果指定了host但没有hostaddr，则会发生主机名查找（在使用PQconnectPoll时，PQconnectPoll第一次考虑这个主机名时会发生查找，可能会导致PQconnectPoll阻塞可观的时间）。

- 如果hostaddr被指定且没有host，hostaddr的值给出了服务器的网络地址。如果认证方法要求一个主机名则连接尝试将会失败。

- 如果host和hostaddr都被指定，hostaddr的值给出服务器的网络地址。host的值将被忽略，除非认证方法要求它，在那种情况下它将被用作主机名。

注意如果host不是网络地址hostaddr上的服务器名，认证很可能会失败。此外，当host和hostaddr都被指定时，host被用来在口令文件中标识该连接（见第 34.15 节。

逗号分隔的hostaddr值列表也被接受，这种情况下将依次尝试列表中的主机。列表中的空项会导致对应的主机名被使用，或者如果对应的主机名也为空时使用默认主机名。详情请参考第 34.1.1.3 节

如果没有一个主机名或主机地址，libpq将尝试使用一个本地 Unix 域套接字连接，或者在没有 Unix 域套接字的机器上尝试连接到localhost。

port

在服务器主机上要连接的端口号，或者用于 Unix 域连接的套接字文件名扩展。如果在host或hostaddr参数中给出了多个主机，这个参数可以指定一个逗号分隔的端口列表，列表长度与主机列表相同，或者这个参数可以指定一个单一的端口号用于所有的主机。空字符串或者逗号分隔列表中的空项指定的是PostgreSQL编译时建立的默认端口号。

dbname

数据库名。默认和用户名相同。在一般的环境下，会为扩展格式检查该值，详见第 34.1.1 节

user

PostgreSQL 要作为哪个用户连接。默认与运行着该应用的用户的操作系统名相同。

password

服务器要求口令认证时要使用的口令。

passfile

指定用于存放口令的文件名（见第 34.15 节。默认是~/.pgpass，或者是Microsoft Windows上的%APPDATA%\postgresql\pgpass.conf（如果该文件不存在也不会报错）。

connect_timeout

连接的最大等待时间，以秒为单位（写作一个十进制整数，例如10）。零、负值或者不指定表示无限等待。允许的最小超时是2秒，因此值1会被解释为2。这个超时单独应用于每个主机名或者IP地址。例如，如果指定两个主机并且connect_timeout为5，每个主机都会在5秒内没有建立起连接的情况下超时，因此花费在等待连接上的总时间可能高达10秒。

client_encoding

为连接设置client_encoding配置参数。除了被相应服务器选项所接受的值，你还能使用auto从客户端的当前区域（Unix 系统上的LC_CTYPE环境变量）决定正确的编码。

options

指定在连接开始时发送给服务器的命令行选项。例如，设置这个参数为-c geqo=off会把会话的geqo参数值设置为off。这个字符串中的空格被认为是命令行参数的分隔符，除非用一个反斜线（\）对它转义，用\\可以表示一个字面意义上的反斜线。可用选项的详细讨论请参考第 19 章

application_name

为application_name配置参数指定一个值。

fallback_application_name

为application_name配置参数指定一个后补值。如果通过一个连接参数或PGAPPNAME环境变量没有为application_name给定一个值，将使用这个值。在希望设置一个默认应用名但不希望它被用户覆盖的一般工具程序中指定一个后补值很有用。

keepalives

控制是否使用客户端的 TCP 保持存活机制。默认值是 1，表示打开。但是如果不想要保持存活机制，你可以改成 0 表示关闭。对于通过一个 Unix 域套接字建立的连接会忽略这个参数。

keepalives_idle

控制非活动多少秒之后 TCP 应该向服务器发送一个存活消息。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被TCP_KEEPIDLE或等效的套接字选项可用的系统以及Windows支持，在其他系统上，它没有效果。

keepalives_interval

控制一个 TCP 存活消息没有被服务器认可多少秒之后应该被重传。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被TCP_KEEPINTVL或等效的套接字选项可用的系统以及Windows支持，在其他系统上，它没有效果。

keepalives_count

控制该客户端到服务器的连接被认为死亡之前可以丢失的 TCP 存活消息数量。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被TCP_KEEPCNT或等效的套接字选项可用的系统以及 Windows支持，在其他系统上，它没有效果。

tty

被忽略（之前，这指定向哪里发送服务器调试输出）。

replication

这个选项决定是否该连接应该使用复制协议而不是普通协议。这是PostgreSQL的复制连接以及pg_basebackup之类的工具在内部使用的协议，但也可以被第三方应用使用。关于复制协议的介绍可以参考第 53.4 节

支持下列值，大小写无关：

true、on、yes、1

连接进入到物理复制模式。

database

连接进入到逻辑复制模式，连接到dbname参数中指定的数据库。

false、off、no、0

该连接是一个常规连接，这是默认行为。

在物理或者逻辑复制模式中，仅能使用简单查询协议。

sslmode

这个选项决定一个SSL TCP/IP连接是否将与服务器协商，或者决定以何种优先级协商。有六种模式：

disable

只尝试非SSL连接

allow

首先尝试非SSL连接，如果失败再尝试SSL连接

prefer（默认）

首先尝试SSL连接，如果失败再尝试非SSL连接

require

只尝试SSL连接。如果存在一个根 CA 文件，以verify-ca被指定的相同方式验证该证书

verify-ca

只尝试SSL连接，并且验证服务器证书是由一个可信的证书机构颁发的（CA）

verify-full

只尝试SSL连接，验证服务器证书是由一个可信的CA颁发并且请求的服务器主机名匹配证书中的主机名

这些选项如何工作的详细描述见第 34.18 节

对于 Unix 域套接字通信，sslmode会被忽略。如果PostgreSQL被编译为不带 SSL 支持，使用选项require、verify-ca或verify-full将导致错误，而选项allow和prefer将会被接受但是libpq将不会真正尝试SSL连接。

requiressl

为了支持sslmode模式，这个选项已被废弃。

如果设置为 1，则要求一个到服务器的SSL连接（这等效于sslmode require）。如果服务器不接受SSL连接，libpq则将拒绝连接。如果设置为 0（默认），libpq将与该服务器协商连接类型（等效于sslmode prefer）。只有PostgreSQL被编译为带有 SSL 支持，这个选项才可用。

sslcompression

如果设置为1，通过SSL连接发送的数据将被压缩。如果设置为0，压缩将被禁用。默认值为0。如果建立的连接没有使用SSL，则这个参数会被忽略。

现如今SSL压缩被认为是不安全的，因此已经不再推荐使用。OpenSSL 1.1.0默认禁用压缩，并且很多操作系统发行版在前面的版本中也将其禁用，因此如果服务器不接受压缩，将这个参数设置为on不会有任何效果。另一方面，1.0.0之前的OpenSSL不支持禁用压缩，因此对那些版本来说会忽略这个参数，而是否使用压缩取决于服务器。

如果安全性不是主要考虑的问题，在网络是瓶颈的情况下，压缩能够改进吞吐量。如果CPU性能是受限的因素，禁用压缩能提高响应时间和吞吐量。

sslcert

这个参数指定客户端 SSL 证书的文件名，它替换默认的~/.postgresql/postgresql.crt。如果没有建立 SSL 连接，这个参数会被忽略。

sslkey

这个参数指定用于客户端证书的密钥位置。它能指定一个会被用来替代默认的~/.postgresql/postgresql.key的文件名，或者它能够指定一个从外部“引擎”（引擎是OpenSSL的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立 SSL 连接，这个参数会被忽略。

sslrootcert

这个参数指定一个包含 SSL 证书机构（CA）证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。默认值是~/.postgresql/root.crt。

sslcrl

这个参数指定 SSL 证书撤销列表（CRL）的文件名。列在这个文件中的证书如果存在，在尝试认证该服务器证书时会被拒绝。默认值是~/.postgresql/root.crl。

requirepeer

这个参数指定服务器的操作系统用户，例如requirepeer=postgres。当建立一个 Unix 域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与 TCP/IP 连接上 SSL 证书相似的服务器认证（注意，如果 Unix 域套接字在/tmp或另一个公共可写的位置，任何用户能开始一个在那里监听的服务器。使用这个参数来保证你连接的是一个由可信用户运行的服务器）。这个选项只在实现了peer认证方法的平台上受支持，见第 20.9 节

krbsrvname

当用 GSSAPI 认证时，要使用的 Kerberos 服务名。为了让 Kerberos 认证成功，这必须匹配在服务器配置中指定的服务名（另见第 20.6 节。

gsslib

用于 GSSAPI 认证的 GSS 库。只用在 Windows 上。设置为gssapi可强制 libpq 用 GSSAPI 库来代替默认的 SSPI 进行认证。

service

用于附加参数的服务名。它指定保持附加连接参数的pg_service.conf中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。见第 34.16 节

target_session_attrs

如果这个参数被设置为read-write，只有默认接受读写事务的连接才是可接受的。在任何成功的连接上将发出查询SHOW transaction_read_only，如果它返回on则连接将被关闭。如果在连接字符串中指定了多个主机，只要连接尝试失败就会尝试剩余的服务器。这个参数的默认值any认为所有连接都可接受。

34.2. 连接状态函数

这些函数可以被用来询问一个已有数据库连接对象的状态。

> **提示**
>
> libpq应用程序员应该小心地维护PGconn抽象。使用下面描述的访问函数来理解PGconn的内容。我们不推荐使用libpq-int.h引用内部的PGconn域，因为它们可能在未来改变。

下列函数返回一个连接所建立的参数值。这些值在连接的生命期中是固定的。如果使用的是多主机连接字符串，如果使用同一个PGconn对象建立新连接，PQhost、PQport以及PQpass可能会改变。其他值在PGconn对象的一生中都是固定的。

PQdb

返回该连接的数据库名。

```
char *PQdb(const PGconn *conn);
```

PQuser

返回该连接的用户名。

```
char *PQuser(const PGconn *conn);
```

PQpass

返回该连接的口令。

```
char *PQpass(const PGconn *conn);
```

PQpass将返回连接参数中指定的口令，如果连接参数中没有口令并且能从口令文件中得到口令，则它将返回得到的口令。在后一种情况中，如果连接参数中指定了多个主机，在连接被建立之前都不能依赖PQpass的结果。连接的状态可以用函数PQstatus检查。

PQhost

返回活跃连接的服务器主机名。可能是主机名、IP 地址或者一个目录路径（如果通过 Unix 套接字连接，路径的情况很容易区分，因为路径总是一个绝对路径，以/开始）。

```
char *PQhost(const PGconn *conn);
```

如果连接参数同时指定了host和hostaddr，则PQhost将返回host信息。如果仅指定了hostaddr，则返回它。如果在连接参数中指定了多个主机，PQhost返回实际连接到的主机。

如果conn参数是NULL，则PQhost返回NULL。否则，如果有一个错误产生主机信息（或许是连接没有被完全建立或者有什么错误），它会返回一个空字符串。

如果在连接参数中指定了多个主机，则在连接建立之前都不能依赖于PQhost的结果。连接的状态可以用函数PQstatus检查。

PQport

返回活跃连接的端口。

```
char *PQport(const PGconn *conn);
```

如果在连接参数中指定了多个端口，PQport返回实际连接到的端口。

如果conn参数是NULL，则PQport返回NULL。否则，如果有一个错误产生端口信息（或许是连接没有被完全建立或者有什么错误），它会返回一个空字符串。

如果在连接参数中指定了多个端口，则在连接建立之前都不能依赖于PQport的结果。连接的状态可以用函数PQstatus检查。

PQtty

返回该连接的调试TTY（这已被废弃，因为服务器不再关心TTY设置，但这个函数保持了向后兼容）。

```
char *PQtty(const PGconn *conn);
```

PQoptions

返回被传递给连接请求的命令行选项。

```
char *PQoptions(const PGconn *conn);
```

下列函数返回会随着在PGconn对象上执行的操作改变的状态数据。

PQstatus

返回该连接的状态。

```
ConnStatusType PQstatus(const PGconn *conn);
```

该状态可以是一系列值之一。不过，其中只有两个在一个异步连接过程之外可见：CONNECTION_OK和CONNECTION_BAD。一个到数据库的完好连接的状态为CONNECTION_OK。一个失败的连接尝试则由状态CONNECTION_BAD表示。通常，一个 OK状态将一直保持到PQfinish，但是一次通信失败可能导致该状态过早地改变为CONNECTION_BAD。在那种情况下，该应用可以通过调用PQreset尝试恢复。

关于其他可能会被返回的状态代码，请见PQconnectStartParams、PQconnectStart和PQconnectPoll的条目。

PQtransactionStatus

返回服务器的当前事务内状态。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

该状态可能是PQTRANS_IDLE（当前空闲）、PQTRANS_ACTIVE（一个命令运行中）、PQTRANS_INTRANS（空闲，处于一个合法的事务块中）或者PQTRANS_INERROR（空闲，处于一个失败的事务块中）。如果该连接损坏，将会报告PQTRANS_UNKNOWN。只有当一个查询已经被发送给服务器并且还没有完成时，才会报告PQTRANS_ACTIVE。

PQparameterStatus

查找服务器的一个当前参数设置。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

某一参数值会被服务器在连接开始或值改变时自动报告。PQparameterStatus可以被用来询问这些设置。它为已知的参数返回当前值，为未知的参数返回NULL。

自当前发布开始会被报告的参数包括 server_version、 server_encoding、client_encoding、 application_name、 is_superuser、 session_authorization、DateStyle、 IntervalStyle、 TimeZone、 integer_datetimes以及 standard_conforming_strings（ server_encoding、TimeZone以及integer_datetimes在 8.0 以前的发布中不被报告；standard_conforming_strings在 8.1 以前的发布中不被报告；IntervalStyle在 8.4 以前的发布中不被报告；application_name在 9.0 以前的发布中不被报告）。注意 server_version、server_encoding以及 integer_datetimes在启动之后无法改变。

3.0 之前协议的服务器不报告参数设置，但是libpq包括获得server_version以及client_encoding值的逻辑。我们鼓励应用使用PQparameterStatus而不是ad hoc代码来确定这些值（不过注意在一个 3.0 之前的连接上，连接开始后通过SET改变client_encoding不会被PQparameterStatus反映）。对于server_version（另见PQserverVersion），它以一种数字形式返回信息，这样更容易进行比较。

如果没有为standard_conforming_strings报告值，应用能假设它是off，也就是说反斜线会被视为字符串中的转义。还有，这个参数的存在可以被作为转义字符串语法（E'...'）被接受的指示。

尽管被返回的指针被声明成const，它事实上指向与PGconn结构相关的可变存储。假定该指针在存储之间保持有效是不明智的。

PQprotocolVersion

询问所使用的 前端/后端协议。

```
int PQprotocolVersion(const PGconn *conn);
```

应用可能希望用这个函数来确定某些特性是否被支持。当前，可能值是 2（2.0 协议）、3（3.0 协议）或零（连接损坏）。协议版本在连接启动完成后将不会改变，但是理论上在连接重置期间是可以改变的。当与PostgreSQL 7.4 或以后的服务器通信时通常使用 3.0 协议，7.4 之前的服务器只支持协议 2.0（协议 1.0 已被废弃并且不再被libpq支持）。

PQserverVersion

返回一个表示服务器版本的整数。

```
int PQserverVersion(const PGconn *conn);
```

应用可能会使用这个函数来判断它们连接到的数据库服务器的版本。结果通过将服务器的主版本号乘以10000再加上次版本号形成。例如，版本10.1将被返回为100001，而版本11.0将被返回为110000。如果连接无效则返回零。

在主版本10之前，PostgreSQL采用一种由三个部分组成的版本号，其中前两部分共同表示主版本。对于那些版本，PQserverVersion为每个部分使用两个数字，例如版本9.1.5将被返回为90105，而版本9.2.0将被返回为90200。

因此，出于判断特性兼容性的目的，应用应该将PQserverVersion的结果除以100而不是10000来判断逻辑的主版本号。在所有的发行序列中，只有最后两个数字在次发行（问题修正发行）之间不同。

PQerrorMessage

返回连接上的一个操作最近产生的错误消息。

```
char *PQerrorMessage(const PGconn *conn);
```

几乎所有的libpq在失败时都会为PQerrorMessage设置一个消息。注意按照libpq习惯，一个非空PQerrorMessage结果由多行构成，并且将包括一个尾部新行。调用者不应该直接释放结果。当相关的PGconn句柄被传递给PQfinish时，它将被释放。在PGconn结构上的多个操作之间，不能指望结果字符串会保持不变。

PQsocket

获得到服务器连接套接字的文件描述符号。一个合法的描述符将会大于等于零。结果为-1 表示当前没有打开服务器连接（在普通操作期间这将不会改变，但是在连接设置或重置期间可能改变）。

```
int PQsocket(const PGconn *conn);
```

PQbackendPID

返回处理这个连接的后端进程的进程ID（PID）。

```
int PQbackendPID(const PGconn *conn);
```

后端PID有助于调试目的并且可用于与NOTIFY消息（它包括发出提示的后端进程的PID）进行比较。注意PID属于一个在数据库服务器主机上执行的进程，而不是本地主机进程！

PQconnectionNeedsPassword

如果连接认证方法要求一个口令但没有可用的口令，返回真（1）。否则返回假（0）。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

这个函数可以在连接尝试失败后被应用于决定是否向用户提示要求一个口令。

PQconnectionUsedPassword

如果连接认证方法使用一个口令，返回真（1）。否则返回假（0）。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

这个函数能在一次连接尝试失败或成功后用于检测该服务器是否要求一个口令。

下面的函数返回与 SSL 有关的信息。这些信息在连接建立后通常不会改变。

PQsslInUse

如果该连接使用了 SSL 则返回真（1），否则返回假（0）。

```
int PQsslInUse(const PGconn *conn);
```

PQsslAttribute

返回关于该连接的有关 SSL 的信息。

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

可用的属性列表取决于使用的 SSL 库以及连接类型。如果一个属性不可用， 会返回 NULL。

下列属性通常是可用的：

library

正在使用的 SSL 实现的名称（当前只实现了 "OpenSSL"）

protocol

正在使用的 SSL/TLS 版本。常见的值有"TLSv1"、 "TLSv1.1"以及"TLSv1.2"，但是如果 一种实现使用了其他协议，可能返回其他字符串。

key_bits

加密算法所使用的密钥位数。

cipher

所使用的加密套件的简短名称，例如 "DHE-RSA-DES-CBC3-SHA"。这些名称与每一种 SSL 实现相关。

compression

如果正在使用 SSL 压缩，则返回压缩算法的名称。如果使用了压缩 但是算法未知则返回 "on" 。如果没有使用压缩，则返回 "off"。

PQsslAttributeNames

返回一个可用的 SSL 名称的数组。该数组以一个 NULL 指针终结。

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

PQsslStruct

返回一个描述该连接的 SSL 实现相关的对象指针。

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

可用的结构，这些结构取决于使用的 SSL 实现。对于 OpenSSL，有一个结构可用， 其名称为 "OpenSSL"。它返回一个指向该 OpenSSL SSL 结构 的指针。要使用这个函数，可以用下面的代码：

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* 使用 OpenSSL 函数来访问 ssl */
    }
```

这个结构可以被用来验证加密级别、检查服务器证书等等。有关这个结构 的信息请参考OpenSSL的文档。

PQgetssl

返回连接中使用的 SSL 结构，如果没有使用 SSL 则返回空。

```
void *PQgetssl(const PGconn *conn);
```

这个函数等效于PQsslStruct(conn, "OpenSSL")。在新的应用中不应该使用它，因为被返回的结构是 OpenSSL 专用的，如果使用了另一种 SSL 实现就不再可用。要检查一个连接是否使用 SSL，应该调用PQsslInUse，而要得到有关连接的更多细节应该使用PQsslAttribute。

34.3. 命令执行函数

一旦到一个数据库服务器的连接被成功建立，这里描述的函数可以被用来执行 SQL 查询和命令。

34.3.1. 主要函数

PQexec

提交一个命令给服务器并且等待结果。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

返回一个PGresult指针或者可能是一个空指针。除了内存不足的情况或者由于严重错误无法将命令发送给服务器之外，一般都会返回一个非空指针。PQresultStatus函数应当被调用来检查返回值是否代表错误（包括空指针的值，它会返回PGRES_FATAL_ERROR）。用PQerrorMessage可得到关于那些错误的详细信息。

命令字符串可以包括多个 SQL 命令（用分号分隔）。在一次PQexec调用中被发送的多个查询会在一个事务中处理，除非其中有显式的BEGIN/COMMIT命令将该查询字符串划分成多个事务（服务器如何处理多查询字符串的更多细节请参考第 53.2.2.1 节。但是注意，返回的PGresult结构只描述该字符串中被执行的最后一个命令的结果。如果一个命令失败，该字符串的处理会在它那里停止并且返回的PGresult会描述错误情况。

PQexecParams

提交一个命令给服务器并且等待结果，它可以在 SQL 命令文本之外独立地传递参数。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

PQexecParams与PQexec相似，但是提供了额外的功能：参数值可以与命令字符串分开指定，并且可以以文本或二进制格式请求查询结果。PQexecParams只在 3.0 协议及其后的连接中被支持，当使用 2.0 协议时它会失败。

该函数的参数是：

conn

要在其中发送命令的连接对象。

command

要执行的 SQL 命令字符串。如果使用了参数，它们在该命令字符串中被引用为$1、$2等。

nParams

提供的参数数量。它是数组paramTypes[]、paramValues[]、paramLengths[]和paramFormats[]的长度（当nParams为零时，数组指针可以是NULL）。

paramTypes[]

通过 OID 指定要赋予给参数符号的数据类型。如果paramTypes为NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。

paramValues[]

指定参数的实际值。这个数组中的一个空指针表示对应的参数为空，否则该指针指向一个以零终止的文本字符串（用于文本格式）或者以服务器所期待格式的二进制数据（用于二进制格式）。

paramLengths[]

指定二进制格式参数的实际数据长度。它对空参数和文本格式参数被忽略。当没有二进制参数时，该数组指针可以为空。

paramFormats[]

指定参数是否为文本（在参数相应的数组项中放一个零）或二进制（在参数相应的数组项中放一个一）。如果该数组指针为空，那么所有参数都会被假定为文本串。

以二进制格式传递的值要求后端所期待的内部表示形式的知识。例如，整数必须以网络字节序被传递。传递numeric值要求关于服务器存储格式的知识，正如src/backend/utils/adt/numeric.c::numeric_send()以及src/backend/utils/adt/numeric.c::numeric_recv()中所实现的。

resultFormat

指定零来得到文本格式的结果，或者指定一来得到二进制格式的结果（目前没有规定要求以不同格式得到不同的结果列，尽管在底层协议中这是可以实现的）。

PQexecParams相对于PQexec的主要优点是参数值可以从命令串中分离，因此避免了冗长的书写、容易发生错误的引用以及转义。

和PQexec不同，PQexecParams至多允许在给定串中出现一个 SQL 命令（其中可以有分号，但是不能有超过一个非空命令）。这是底层协议的一个限制，但是有助于抵抗 SQL 注入攻击。

> **提示**
>
> 通过 OID 指定参数类型很罗嗦，特别是如果你不愿意将特定的 OID 值硬编码到你的程序中时。不过，即使服务器本身也无法确定参数的类型，你可以避免这样做，或者选择一种与你想要的不同的类型。在 SQL 命令文本中，附加一个显式造型给参数符号来表示你将发送什么样的数据类型。例如：
>
> ```
> SELECT * FROM mytable WHERE x = $1::bigint;
> ```
>
> 这强制参数$1被当作bigint，而默认情况下它将被赋予与x相同的类型。当以二进制格式发送参数值时，我们强烈推荐以这种方式或通过指定一个数字类型的 OID 来强制参数类型决定。因为二进制格式比文本格式具有更少的冗余，并且因此服务器将不会有更多机会为你检测一个类型匹配错误。

PQprepare

提交一个请求用给定参数创建一个预备语句并且等待完成。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

PQprepare创建一个后面会由PQexecPrepared执行的预备语句。这个特性允许命令被反复执行而无需每次都进行解析和规划，详见PREPARE。PQprepare只在协议 3.0 及之后的连接中被支持，当使用协议 2.0 时它将失败。

该函数从query串创建一个名为stmtName的预备语句，该串必须包含一个单一 SQL 命令。stmtName可以是""来创建一个未命名语句，在这种情况下任何已存在未命名语句将被自动替换。如果语句名称已经在当前会话中被定义，则是一种错误。如果使用了任何参数，它们在查询中以$1、$2等引用。nParams是参数的个数，其类型在数组paramTypes[]中被预先指定（当nParams为零时，该数组指针可以是NULL）。paramTypes[]通过 OID 指定要赋予给参数符号的数据类型。如果paramTypes是NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。还有，查询能够使用编号高于nParams的参数符号，它们的数据类型也会被自动推测（找出推测出的数据类型的方法见PQdescribePrepared）。

正如PQexec一样，结果通常是一个PGresult对象，其内容代表服务器端成功或失败。一个空结果表示内存不足或者根本无法发送命令。关于错误的更多信息请见PQerrorMessage。

用于PQexecPrepared的预备语句也能通过执行 SQL PREPARE语句来创建。还有，尽管没有libpq函数来删除一个预备语句，SQL DEALLOCATE语句可被用于此目的。

PQexecPrepared

发送一个请求来用给定参数执行一个预备语句，并且等待结果。

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared像PQexecParams，但是要被执行的命令是用之前准备的语句的名字指定，而不是指定一个查询串。这个特性允许将被重复使用的命令只被解析和规划一次，而不是在每次被执行时都被解析和规划。这个语句必须之前在当前会话中已经被准备好。PQexecPrepared仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

参数和PQexecParams相同，除了给定的是一个预备语句的名称而不是一个查询语句，以及不存在paramTypes[]参数（因为预备语句的参数类型已经在它被创建时决定好了）。

PQdescribePrepared

提交一个请求来获得有关指定预备语句的信息，并且等待完成。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePrepared允许一个应用获得有关一个之前预备好的语句的信息。PQdescribePrepared仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

stmtName可以用""或者NULL来引用未命名语句，否则它必须是一个现有预备语句的名字。如果成功，一个PGresult以及状态PGRES_COMMAND_OK会被返回。函数PQnparams和PQparamtype可以被应用到这个PGresult来得到关于该预备语句参数的额信息，而函数PQnfields、PQfname、PQftype等提供该语句结果列（如果有）的信息。

PQdescribePortal

提交一个请求来得到有关指定入口的信息，并且等待完成。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal允许一个应用获得有关一个之前被创建的入口的信息（libpq不提供对入口任何直接的访问，但是你可以使用这个函数来观察一个通过DECLARE CURSOR SQL 命令创建的游标的属性）。PQdescribePortal仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

portalName可以用""或者NULL来引用未命名入口，否则它必须是一个现有入口的名字。如果陈功，一个PGresult和状态PGRES_COMMAND_OK会被返回。函数PQnfields、PQfname、PQftype等可以被应用到PGresult来获得有关该入口结果列（如果有）的信息。

PGresult结构封装了由服务器返回的结果。libpq应用程序员应该小心地维护PGresult的抽象。使用下面的存储器函数来得到PGresult的内容。避免直接引用PGresult结构的域，因为它们可能在未来更改。

PQresultStatus

返回该命令的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus能返回下列值之一：

PGRES_EMPTY_QUERY

发送给服务器的字符串为空。

PGRES_COMMAND_OK

一个不返回数据的命令成功完成。

PGRES_TUPLES_OK

一个返回数据的命令（例如SELECT或者SHOW）成功完成。

PGRES_COPY_OUT

从服务器复制出数据的传输开始。

PGRES_COPY_IN

复制数据到服务器的传输开始。

PGRES_BAD_RESPONSE

无法理解服务器的响应。

PGRES_NONFATAL_ERROR

发生了一次非致命错误（一个提示或警告）。

PGRES_FATAL_ERROR

发生了一次致命错误。

PGRES_COPY_BOTH

向服务器复制数据/从服务器复制数据的传输开始。这个特性当前只被用于流复制，因此这个状态应该不会在普通应用中出现。

PGRES_SINGLE_TUPLE

PGresult包含来自于当前命令的一个单一结果元组。这个状态只在查询选择了单一行模式时发生（见第 34.5 节。

如果结果状态是PGRES_TUPLES_OK或者PGRES_SINGLE_TUPLE，那么下面所描述的函数能被用来检索该查询所返回的行。注意，一个恰好检索零行的SELECT命令仍然会显示PGRES_TUPLES_OK。PGRES_COMMAND_OK用于从不返回行的命令（不带RETURNING子句的INSERT或者UPDATE等）。一个PGRES_EMPTY_QUERY可能表示客户端软件中的一个缺陷。

一个状态为PGRES_NONFATAL_ERROR的结果将不会被PQexec或者其他查询执行函数直接返回，这类结果将被传递给提示处理器（见 第 34.12 节。

PQresStatus

将PQresultStatus返回的枚举转换成描述状态编码的字符串常量。调用者不应该释放结果。

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

返回与该命令相关的错误消息，如果有错误则会返回一个空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

如果有一个错误，被返回的字符串将包含一个收尾的新行。调用者不应该直接释放结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

紧跟着一个PQexec或PQgetResult调用，PQerrorMessage（在连接上）将返回与PQresultErrorMessage相同的字符串（在结果上）。不过，一个PGresult将保持它的错误消息直到被销毁，而连接的错误消息将在后续操作被执行时被更改。当你想要知道与一个特定PGresult相关的状态，使用PQresultErrorMessage。而当你想要知道连接上最后一个操作的状态，使用PQerrorMessage。

PQresultVerboseErrorMessage

返回与PGresult对象相关的错误消息的重新格式化的版本。

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

在有些情况下，客户端可能希望得到之前报告过的错误的更加详尽的版本。如果在产生给定PGresult的连接上 verbosity 设置有效，PQresultVerboseErrorMessage会通过计算已经被PQresultErrorMessage产生过的消息来满足这种需求。如果PGresult不是一个错误结果，则会报告“PGresult is not an error result”。返回的字符串包括一个新行作为结尾。

和大部分从PGresult中提取数据的其他函数不同，这个函数的结果是一个全新分配的字符串。调用者在不需要这个字符串以后，必须使用PQfreemem()释放它。

如果内存不足，可能会返回 NULL。

PQresultErrorField

返回一个错误报告的一个域。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode是一个错误域标识符，见下列符号。如果PGresult不是一个错误或者警告结果或者不包括指定域，会返回NULL。域通常不包括一个收尾的新行。调用者不应该直接释放结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

下列域代码可用：

PG_DIAG_SEVERITY

严重性。域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、DEBUG、INFO或LOG（在一个提示消息中）。或者是其中之一的一个本地化翻译。总是存在。

PG_DIAG_SEVERITY_NONLOCALIZED

域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、DEBUG、INFO或LOG（在一个提示消息中）。这和PG_DIAG_SEVERITY域相同，不过内容不会被本地化。只存在于PostgreSQL 9.6 版本以后产生的报告中。

PG_DIAG_SQLSTATE

用于错误的 SQLSTATE 代码。SQLSTATE 代码标识了已经发生的错误的类型，它可以被前端应用用来执行特定操作（例如错误处理）来响应一个特定数据库错误。一个可能的 SQLSTATE 代码列表可见附录 A 这个域无法被本地化，并且总是存在。

PG_DIAG_MESSAGE_PRIMARY

主要的人类可读的错误消息（通常是一行）。总是存在。

PG_DIAG_MESSAGE_DETAIL

细节：一个可选的次级错误消息，它携带了关于问题的等多细节。可能有多行。

PG_DIAG_MESSAGE_HINT

提示：一个关于如何处理该问题的可选建议。它与细节的区别在于它提供了建议（可能不合适）而不是铁的事实。可能有多行。

PG_DIAG_STATEMENT_POSITION

包含一个十进制整数的字符串，它表示一个错误游标位置，该位置是原始语句字符串的索引。第一个字符的索引是 1，位置以字符计算而不是以及字节计算。

PG_DIAG_INTERNAL_POSITION

这被定义为与PG_DIAG_STATEMENT_POSITION域相同，但是它被用在游标位置引用一个内部产生的命令而不是客户端提交的命令时。当这个域出现
时，PG_DIAG_INTERNAL_QUERY域将总是出现。

PG_DIAG_INTERNAL_QUERY

一个失败的内部产生的命令的文本。例如，这可能是由一个 PL/pgSQL 函数发出的 SQL 查询。

PG_DIAG_CONTEXT

指示错误发生的环境。当前这包括活动过程语言函数的调用栈追踪以及内部生成的查询。追踪是每行一项，最近的排在最前面。

PG_DIAG_SCHEMA_NAME

如果错误与某个特定的数据库对象相关，这里是包含该对象的模式名（如果有）。

PG_DIAG_TABLE_NAME

如果错误与某个特定表相关，这里是该表的名字（该表的模式参考模式名域）。

PG_DIAG_COLUMN_NAME

如果错误与一个特定表列相关，这里是该表列的名字（参考模式和表名域来标识该表）。

PG_DIAG_DATATYPE_NAME

如果错误与一个特定数据类型相关，这里是该数据了行的名字（该数据类型的模式名参考模式名域）。

PG_DIAG_CONSTRAINT_NAME

如果错误与一个特定约束相关，这里是该约束的名字。相关的表或域参考上面列出的域（为了这个目的，索引也被视作约束，即使它们不是用约束语法创建的）。

PG_DIAG_SOURCE_FILE

报告错误的源代码所在的文件名。

PG_DIAG_SOURCE_LINE

报告错误的源代码行号。

PG_DIAG_SOURCE_FUNCTION

报告错误的源代码函数的名字。

> **注意**
>
> 用于模式名、表名、列名、数据类型名和约束名的域只提供给有限的错误类型，见附录 A 不要假定任何这些域的存在保证另一个域的存在。核心错误源会遵守上面提到的内在联系，但是用户定义的函数可能以其他方式使用这些域。同样地，不要假定这些域代表当前数据库中同类的对象。

客户端负责格式化显示信息来迎合它的需要，特别是根据需要打断长的行。出现在错误消息域中的新行字符应该被当作分段而不是换行。

libpq内部产生的错误将有严重和主要消息，但是通常没有其他域。3.0 协议之前的服务器返回的错误将包括严重和主要消息，并且有时候还有细节消息，但是没有其他域。

注意错误与只从PGresult对象中有效，对PGconn对象无效。没有PQerrorField函数。

PQclear

Frees the storage associated with a 释放与一个PGresult相关的存储。每一个命令结果不再需要时应该用PQclear释放。

```
void PQclear(PGresult *res);
```

你可以按照需要保留PGresult对象，当你发出一个新命令时它也不会消失，甚至关闭连接时也不会消失。要去掉它，你必须调用PQclear。没有这样做将会导致在应用中的内存泄露。

34.3.2. 检索查询结果信息

这些函数被用来从一个代表成功查询结果（也就是状态为PGRES_TUPLES_OK或者PGRES_SINGLE_TUPLE）的PGresult对象中抽取信息。它们也可以被用来从一个成功的 Describe 操作中抽取信息：一个 Describe 的结果具有和该查询被实际执行所提供的完全相同的列信息，但是它没有行。对于其他状态值的对象，这些函数会认为结果具有零行和零列。

PQntuples

返回查询结果中的行（元组）数（注意，PGresult对象被限制为不超过INT_MAX行，因此一个int结果就足够了）。

```
int PQntuples(const PGresult *res);
```

PQnfields

返回查询结果中每一行的列（域）数。

```
int PQnfields(const PGresult *res);
```

PQfname

返回与给定列号相关联的列名。列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

```
char *PQfname(const PGresult *res,
              int column_number);
```

如果列号超出范围，将返回NULL。

PQfnumber

返回与给定列名相关联的列号。

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

如果给定的名字不匹配任何列，将返回 -1。

给定的名称被视作一个 SQL 命令中的一个标识符，也就是说，除非被双引号引用，它是小写形式的。例如，给定一个 SQL 命令：

```
SELECT 1 AS FOO, 2 AS "BAR";
```

我们将得到结果：

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

PQftable

返回给定列从中取出的表的 OID。列号从 0 开始。

```
Oid PQftable(const PGresult *res,
             int column_number);
```

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回InvalidOid。你可以查询系统表pg_class来确定究竟是哪个表被引用。

当你包括libpq头文件，类型oid以及常数InvalidOid将被定义。它们将都是某种整数类型。

PQftablecol

返回构成指定查询结果列的列（在其表中）的列号。查询结果列号从 0 开始，但是表列具有非零编号。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回零。

PQfformat

返回指示给定列格式的格式编码。列号从 0 开始。

```
int PQfformat(const PGresult *res,
              int column_number);
```

格式代码零指示文本数据表示，而格式代码一表示二进制表示（其他代码被保留用于未来的定义）。

PQftype

返回与给定列号相关联的数据类型。被返回的整数是该类型的内部 OID 号。列号从 0 开始。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

你可以查询系统表pg_type来得到多个数据类型的名字和属性。内建数据类型的OID被定义在源代码树中的文件src/include/catalog/pg_type_d.h中。

PQfmod

返回与给定列号相关联的列的修饰符类型。列号从 0 开始。

```
int PQfmod(const PGresult *res,
           int column_number);
```

修饰符值的解释是与类型相关的，它们通常指示精度或尺寸限制。值 -1 被用来指示“没有信息可用”。大部分的数据类型不适用修饰符，在那种情况中值总是 -1。

PQfsize

返回与给定列号相关的列的尺寸（以字节计）。列号从 0 开始。

```
int PQfsize(const PGresult *res,
            int column_number);
```

PQfsize返回在一个数据库行中为这个列分配的空间，换句话说是服务器对该数据类型的内部表示的尺寸（因此，它对客户端并不是真地非常有用）。一个负值指示该数据类型是变长的。

PQbinaryTuples

如果PGresult包含二进制数据，返回 1。如果包含的是文本数据，返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

这个函数已经被废弃（除了与COPY一起使用），因为一个单一PGresult可以在某些列中包含文本数据而且在另一些列中包含二进制数据。PQfformat要更好。只有结果的所有列是二进制（格式 1）时PQbinaryTuples才返回 1。

PQgetvalue

返回一个PGresult的一行的单一域值。行和列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

对于文本格式的数据，PQgetvalue返回的值是该域值的一种空值结束的字符串表示。对于二进制格式的数据，该值是由该数据类型的typsend和typreceive函数决定的二进制表示（在这种情况下该值实际上也跟随着一个零字节，但是这通常没有用处，因为该值很可能包含嵌入的空）。

如果该域值为空，则返回一个空串。关于区分空值和空字符串值请见PQgetisnull。

PQgetvalue返回的指针指向作为PGresult结构一部分的存储。我们不应该修改它指向的数据，并且如果要在超过PGresult结构本身的生命期之外使用它，我们必须显式地把该数据拷贝到其他存储中。

PQgetisnull

测试一个域是否为空值。行号和列号从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

如果该域是空，这个函数返回 1。如果它包含一个非空值，则返回 0（注意PQgetvalue将为一个空域返回一个空串，不是一个空指针）。

PQgetlength

返回一个域值的真实长度，以字节计。行号和列号从 0 开始。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

这是特定数据值的真实数据长度，也就是PQgetvalue指向的对象的尺寸。对于文本数据格式，这和strlen()相同。对于二进制格式这是基本信息。注意我们不应该依赖于PQfsize来得到真值的数据长度。

PQnparams

返回一个预备语句的参数数量。

```
int PQnparams(const PGresult *res);
```

只有在查看PQdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

PQparamtype

返回所指示的语句参数的数据类型。参数号从 0 开始。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

只有在查看PQdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

PQprint

将所有的行打印到指定的输出流，以及有选择地将列名打印到指定的输出流。

```
void PQprint(FILE *fout,      /* 输出流 */
```

```
                const PGresult *res,
                const PQprintOpt *po);
    typedef struct
    {
        pqbool  header;      /* 打印输出域标题和行计数 */
        pqbool  align;       /* 填充对齐域 */
        pqbool  standard;    /* 旧的格式 */
        pqbool  html3;       /* 输出 HTML 表格 */
        pqbool  expanded;    /* 扩展表格 */
        pqbool  pager;       /* 如果必要为输出使用页 */
        char    *fieldSep;   /* 域分隔符 */
        char    *tableOpt;   /* 用于 HTML 表格元素的属性 */
        char    *caption;    /* HTML 表格标题 */
        char    **fieldName; /* 替换域名称的空终止数组 */
    } PQprintOpt;
```

这个函数以前被psql用来打印查询结果，但是现在不是这样了。注意它假定所有的数据都是文本格式。

34.3.3. 检索其他结果信息

这些函数被用来从PGresult对象中抽取其他信息。

PQcmdStatus

返回来自于产生PGresult的 SQL 命令的命令状态标签。

```
char *PQcmdStatus(PGresult *res);
```

通常这就是该命令的名称，但是它可能包括额外数据，例如已被处理的行数。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

PQcmdTuples

返回受该 SQL 命令影响的行数。

```
char *PQcmdTuples(PGresult *res);
```

这个函数返回一个字符串，其中包含着产生PGresult的SQL语句影响的行数。这个只能被用于下列情况：执行一个SELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH或COPY语句，或者一个包含INSERT、UPDATE或DELETE语句的预备查询的EXECUTE。如果产生PGresult的命令是其他什么东西，PQcmdTuples会返回一个空串。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

PQoidValue

如果该SQL命令是一个正好将一行插入到具有 OID 的表的INSERT，或者是一个包含合适INSERT语句的预备查询的EXECUTE，这个函数返回被插入行的 OID。否则，这个函数返回InvalidOid。如果被INSERT语句影响的表不包含 OID，这个函数也将返回InvalidOid。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

由于PQoidValue和不是线程安全的，这个函数已经被废弃。它返回包含被插入行的 OID 的一个字符串，而PQoidValue返回 OID 值。

```
char *PQoidStatus(const PGresult *res);
```

34.3.4. 用于包含在 SQL 命令中的转移串

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

为了让一个串能用在 SQL 命令中，PQescapeLiteral可对它进行转义。当在 SQL 命令中插入一个数据值作为文字常量时，这个函数很有用。一些字符（例如引号和反斜线）必须被转义以防止它们被 SQL 解析器解释成特殊的意思。PQescapeLiteral执行这种操作。

PQescapeLiteral返回一个str参数的已被转义版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeLiteral会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被PostgreSQL字符串解析器正确地处理。还会加上一个终止零字节。包括在结果串中的PostgreSQL字符串必须用单引号包围。

发生错误时，PQescapeLiteral返回NULL并且一个合适的消息会被存储在conn对象中。

> **提示**
>
> 在处理从一个非可信源接收到的串时，做正确的转义特别重要。否则就会有安全性风险：你容易受到“SQL 注入”攻击，其中可能会有预期之外的 SQL 语句会被喂给你的数据库。

注意，当一个数据值被作为PQexecParams或其兄弟例程中的一个独立参数传递时，没有必要做转义而且做转义也不正确。

PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifier转义一个要用作 SQL 标识符的字符串，例如表名、列名或函数名。当一个用户提供的标识符可能包含被 SQL 解析器解释为标识符一部分的特殊字符时，或者当该标识符可能包含大小写形式应该被保留的大写字符时，这个函数很有用。

PQescapeIdentifier返回一个str参数的已被转义为 SQL 标识符的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeIdentifier会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被作为一个 SQL 标识符正确地处理。还会加上一个终止零字节。返回串也将被双引号包围。

发生错误时，PQescapeIdentifier返回NULL并且一个合适的消息会被存储在conn对象中。

> **提示**
>
> 与字符串一样，要阻止 SQL 注入攻击，当从一个不可信的来源接收到 SQL 标识符时，它们必须被转义。

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConn转义字符串，它很像PQescapeLiteral。与PQescapeLiteral不一样的是，调用者负责提供一个合适尺寸的缓冲区。此外，PQescapeStringConn不产生必须包围PostgreSQL字符串的单引号。它们应该在结果要插入的 SQL 命令中提供。参数from指向要被转义的串的第一个字符，并且length参数给出了这个串中的字节数。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeStringConn会停止在零，该行为更像strncpy）。 to应当指向一个缓冲区，它能够保持至少比length值的两倍还要多至少一个字节，否则该行为是未被定义的。如果to和from串重叠，行为也是未被定义的。

如果error参数不是NULL，那么成功时*error被设置为零，错误时设置为非零。当前唯一可能的错误情况涉及源串中非法的多字节编码。错误时仍然会产生输出串，但是可以预期服务器将认为它是畸形的并且拒绝它。在发生错误时，一个合适的消息被存储在conn对象中，不管error是不是NULL。

PQescapeStringConn返回写到to的字节数，不包括终止的零字节。

PQescapeString

PQescapeString是一个更老的被废弃的PQescapeStringConn版本。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

PQescapeStringConn和PQescapeString之间的唯一区别是不需要PGconn或error参数。正因为如此，它不能基于连接属性（例如字符编码）调整它的行为并且因此它可能给出错误的结果。还有，它没有方法报告错误情况。

PQescapeString可以在一次只使用一个PostgreSQL连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。在其他环境中它是一个安全性灾难并且应该用PQescapeStringConn来避免。

PQescapeByteaConn

把要用于一个 SQL 命令的二进制数据用类型bytea转义。和PQescapeStringConn一样，只有在将数据直接插入到一个 SQL 命令串时才使用它。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

当某些字节值被用作一个SQL语句中的bytea文字的一部分时，它们必须被转义。PQescapeByteaConn转义使用十六进制编码或反斜线转义的字节。详见第 8.4 节

from参数指向要被转义的串的第一个字节，并且from_length参数给出这个二进制串中的字节数（一个终止的零字节是不需要的也是不被计算的）。to_length参数指向一个将保持生成的已转义串长度的变量。这个结果串长度包括结果的终止零字节。

PQescapeByteaConn返回一个from参数的已被转义为二进制串的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。返回串中的所有特殊字符都被替换掉，这样它们能被PostgreSQL的字符串解析器以及bytea输入函数正确地处理。还会加上一个终止零字节。不是结果串一部分的PostgreSQL字符串必须被单引号包围。

在发生错误时，将返回一个空指针，并且一个合适的错误消息被存储在conn对象中。当前，唯一可能的错误是没有足够的内存用于结果串。

PQescapeBytea

PQescapeBytea是一个更老的被废弃的PQescapeByteaConn版本。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

与PQescapeByteaConn的唯一区别是PQescapeBytea不用一个PGconn参数。正因为这样，PQescapeBytea只能在一次只使用一个PostgreSQL连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。如果在有多个数据库连接的程序中使用，它可能给出错误的结果（在那种情况下使用PQescapeByteaConn）。

PQunescapeBytea

将二进制数据的一个字符串表示转换成二进制数据 — 它是PQescapeBytea的逆向函数。当检索文本格式的bytea数据时，需要这个函数，但检索二进制个事时则不需要它。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

from参数指向一个字符串，例如PQgetvalue被应用到一个bytea列上所返回的。PQunescapeBytea把这个串表示转换成它的二进制表示。它返回一个指向用malloc()分配的缓冲区的指针，在错误时返回NULL，并且把缓冲区的尺寸放在to_length中。当结果不再需要时，它必须使用PQfreemem释放。

这种转换并不完全是PQescapeBytea的逆函数，因为当从PQgetvalue接收到字符串时，我们并不能期待它被“转义”。特别地这意味着不需要考虑字符串引用，并且因此也不需要一个参数。

34.4. 异步命令处理

PQexec函数对于在普通的同步应用中提交命令是足以胜任的。不过，它的一些缺点可能对某些用户很重要：

- PQexec会等待命令完成。该应用可能有其他的工作要做（例如维护用户界面），这时它将不希望阻塞等待回应。
- 因为客户端应用的执行在它等待结果时会被挂起，对于应用来说很难决定要不要尝试取消正在进行的命令（这可以在一个信号处理器中完成，但别无他法）。
- PQexec只能返回一个PGresult结构。如果提交的命令串包含多个SQL命令， 除了最后一个PGresult之外都会被PQexec丢弃。
- PQexec总是收集命令的整个结果，把它缓存在一个单一的PGresult中。虽然这简化了应用的错误处理逻辑，它对于包含很多行的结果并不现实。

不想受到这些限制的应用可以改用构建PQexec的底层函数：PQsendQuery以及PQgetResult。还有 PQsendQueryParams、 PQsendPrepare、 PQsendQueryPrepared、PQsendDescribePrepared以及 PQsendDescribePortal， 它们可以与PQgetResult一起使用来分别复制PQexecParams、 PQprepare、 PQexecPrepared、 PQdescribePrepared和PQdescribePortal的功能。

PQsendQuery

向服务器提交一个命令而不等待结果。如果该命令被成功发送则返回 1，否则返回0（此时，可以用PQerrorMessage获取关于失败的信息）。

```
int PQsendQuery(PGconn *conn, const char *command);
```

在成功调用PQsendQuery后，调用PQgetResult一次或者多次来获取结果。
在PQgetResult返回一个空指针之前，都不能再次调用PQsendQuery，返回的空指针指示该命令已经完成。

PQsendQueryParams

向服务器提交一个命令和单独的参数，而不等待结果。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

这个函数等效于PQsendQuery，不过查询参数可以独立于查询字符串分开指定。该函数的参数处理和PQexecParams一样。和PQexecParams类似，它不能在 2.0 协议的连接上工作，并且它只允许在查询字符串中有一条命令。

PQsendPrepare

发送一个请求用给定参数创建一个预备语句，而不等待完成。

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

这个函数是PQprepare的异步版本：如果它能发送这个请求，则返回 1；如果不能，则返回 0。在成功调用之后，调用PQgetResult判断服务器是否成功创建了预备语句。这个函数的参数的处理和PQprepare一样。和PQprepare类似，它不能在 2.0 协议的连接上工作。

PQsendQueryPrepared

发送一个请求用给定参数执行一个预备语句，而不等待结果。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

这个函数与PQsendQueryParams类似，但是要执行的命令是通过一个之前已经命名的预备语句指定， 而不是一个给出的查询字符串。该函数的参数处理和PQexecPrepared一样。和PQexecPrepared类似，它不能在 2.0 协议的连接上工作。

PQsendDescribePrepared

发送一个请求获得指定的预备语句的信息，但不等待完成。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

这个函数是PQdescribePrepared的一个异步版本：如果它能够发送请求，则返回 1；否则，返回 0。在一次成功的调用后，调用PQgetResult来得到结果。该函数的参数处理和PQdescribePrepared一样。和PQdescribePrepared类似，它不能在 2.0 协议的连接上工作。

PQsendDescribePortal

提交一个请求来获得关于指定入口的信息，但不等待完成。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

这个函数是PQdescribePortal的一个异步版本：如果它能够发送请求，则返回 1；否则，返回 0。在一次成功的调用后，调用PQgetResult来得到结果。该函数的参数处理和PQdescribePortal一样。和PQdescribePortal类似，它不能在 2.0 协议的连接上工作。

PQgetResult

等待来自于一个之前的 PQsendQuery、 PQsendQueryParams、 PQsendPrepare、 PQsendQueryPrepared、 PQsendDescribePrepared或 PQsendDescribePortal调用的结果，并且返回它。当该命令完成并且没有更多结果时，将返回一个空指针。

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult必须被反复调用直到它返回一个空指针，空指针表示该命令完成（如果在没有命令活动时被调用，PQgetResult将立即返回一个空指针）。每一个来自PQgetResult的非空结果应该使用之前描述的同一个PGresult访问器处理。不要忘记在处理完之后释放每一个结果对象。注意，只有一个命令是活动的并且PQconsumeInput还没有读取必要的响应数据时， PQgetResult将会阻塞。

> **注意**
>
> 即使当PQresultStatus指出一个致命错误时，PQgetResult也应当被调用直到它返回一个空指针，以允许libpq完全处理该错误信息。

使用PQsendQuery和PQgetResult解决了PQexec的一个问题：如果一个命令字符串包含多个SQL命令，这些命令的结果可以被个别地获得（顺便说一句：这样就允许一种简单的重叠处理形式， 客户端可以处理一个命令的结果，而同时服务器可以继续处理同一命令字符串中后面的查询）。

可以被PQsendQuery和PQgetResult获得的另一种常常想要的特性是一次从大型结果中检索一行。这会在第 34.5 节讨论。

如果只调用PQgetResult（不调用PQsendQuery等）将仍会导致客户端阻塞直到服务器完成下一个SQL命令。用两个函数的正确使用可以避免这种情况：

PQconsumeInput

如果有来自服务器的输入可用，则使用之。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput通常返回 1 表明“没有错误”，而返回 0 表明有某种麻烦发生（此时可以用PQerrorMessage）。注意该结果并不表明是否真正收集了任何输入数据。在调用PQconsumeInput之后，应用可以检查PQisBusy和/或PQnotifies来看看它们的状态是否改变。

即使应用还不准备处理一个结果或通知，PQconsumeInput也可以被调用。这个函数将读取可用的数 据并且把它保存在一个缓冲区中，从而导致一个select()的读准备好指示消失。因此应用可以使用PQconsumeInput立即清除select()条件，并且在空闲时再检查结果。

PQisBusy

如果一个命令繁忙则返回 1，也就是说PQgetResult会阻塞等待输入。返回 0 表示可以调用PQgetResult而不用担心阻塞。

```
int PQisBusy(PGconn *conn);
```

PQisBusy本身将不会尝试从服务器读取数据，因此必须先调用PQconsumeInput，否则繁忙状态将永远不会结束。

一个使用这些函数的典型应用将有一个主循环，在主循环中会使用select()或poll()等待所有它必须响应的情况。其中之一将是来自服务器的输入可用，对select()来说意味着PQsocket标识的文件描述符上有可读的数据。当主循环检测到输入准备好时，它将调用PQconsumeInput读取输入。然后它可以调用PQisBusy，如果PQisBusy返回假（0）则接着调用PQgetResult。它还可以调用PQnotifies检测NOTIFY消息（见第 34.8 节。

一个使用PQsendQuery/PQgetResult的客户端也可以尝试取消一个正在被服务器处理的命令，见第 34.6 节但是，不管PQcancel的返回值是什么，应用都必须继续使用PQgetResult进行正常的结果读取序列。一次成功的取消只会导致命令比不取消时更快终止。

通过使用上述函数，我们可以避免在等待来自数据库服务器的输入时被阻塞。不过，在应用发送输出给服务器时还是可能出现阻塞。这种情况比较少见，但是如果发送非常长的 SQL 命令或者数据值时确实可能发生（不过，最有可能是在应用通过COPY IN发送数据时）。为了避免这种可能性并且实现完全地非阻塞数据库操作，可以使用下列附加函数。

PQsetnonblocking

把连接的状态设置为非阻塞。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

如果arg为 1，把连接状态设置为非阻塞；如果arg为 0，把连接状态设置为阻塞。如果 OK 返回 0，如果错误返回 -1。

在非阻塞状态，调用 PQsendQuery、PQputline、 PQputnbytes、PQputCopyData和PQendcopy将不会阻塞， 但是如果它们需要被再次调用则会返回一个错误。

注意PQexec不会遵循任何非阻塞模式；如果调用PQexec，那么它的行为总是阻塞的。

PQisnonblocking

返回数据库连接的阻塞状态。

```
int PQisnonblocking(const PGconn *conn);
```

如果连接被设置为非阻塞状态，返回 1，如果是阻塞状态返回 0。

PQflush

尝试把任何正在排队的输出数据刷到服务器。如果成功（或者发送队列为空）返回 0，如果因某种原因失败则返回 -1，或者如果还无法把发送队列中的所有数据都发送出去，则返回 1（这种情况只在连接为非阻塞时候才会发生）。

```
int PQflush(PGconn *conn);
```

在一个非阻塞连接上发送任何命令或者数据之后，要调用PQflush。如果它返回 1，就要等待套接字变成读准备好或写准备好。如果它变为写准备好，应再次调用PQflush。如果它变为读准备好，则应先调用PQconsumeInput，然后再调用PQflush。一直重复直到PQflush返回0（有必要检查读准备好并且用PQconsumeInput耗尽输入，因为服务器可能阻塞给我们发送数据的尝试，例如 NOTICE 消息，并且在我们读它的数据之前它都不会读我们的数据）。一旦PQflush返回 0，应等待套接字变成读准备好并且接着按照上文所述读取响应。

34.5. 一行一行地检索查询结果

通常，libpq会收集一个 SQL 命令的整个结果并且把它作为单个PGresult返回给应用。这对于返回大量行的命令是行不通的。对于这类情况，应用可以使用PQsendQuery和PQgetResult的单行模式。在这种模式中，结果行以一次一行的方式被返回给应用。

要进入到单行模式，在一次成功的PQsendQuery（或者其他兄弟函数）调用后立即调用PQsetSingleRowMode。这种模式选择只对当前正在执行的查询有效。然后反复调用PQgetResult，直到它返回空，如第 34.4 节所示。如果该查询返回行，它们会作为单个的PGresult对象返回，它们看起来都像普通的查询结果，只不过其状态代码是PGRES_SINGLE_TUPLE而非PGRES_TUPLES_OK。在最后一行之后或者紧接着该查询返回零行之后，一个状态为PGRES_TUPLES_OK的零行对象会被返回，这就是代表不会有更多行的信号（但是注意仍然有必要继续调用PQgetResult直到它返回空）。所有这些PGresult对象将包含相同的行描述数据（列名、类型等等），这些数据和通常一个查询的PGresult对象的相同。每一个对象都应该按常规用PQclear释放。

PQsetSingleRowMode

为当前正在执行的查询选择单行模式。

```
int PQsetSingleRowMode(PGconn *conn);
```

这个函数只能在调用PQsendQuery或一个其兄弟函数之后立刻调用，并且要在任何连接上的其他操作之前调用，例如PQconsumeInput或PQgetResult。如果在正确的时间被调用，该函数会为当前查询激活单行模式并且返回 1。否则模式会保持不变并且该函数返回0。在任何情况下，当前查询结束之后模式都会恢复到正常。

> **小心**
>
> 在处理一个查询时，服务器可能返回一些行并且接着遇到一个错误导致查询被中断。通常，libpq会丢弃掉这样的行并且至报告错误。但是在单行模式中，那些行（错误之前返回的行）已经被返回给应用。因此，应用将看到一些PGRES_SINGLE_TUPLE PGresult对象并且然后看到一个PGRES_FATAL_ERROR对象。为了得到正确的事务行为，如果查询最终失败，应用必须被设计为丢弃或者撤销使用之前处理的行完成的事情。

34.6. 取消进行中的查询

一个客户端应用可以使用本节描述的函数请求取消一个仍在被服务器处理的命令。

PQgetCancel

创建一个数据结构，这个数据结构包含取消一个通过特定数据库连接发出的命令所需要的信息。

```
PGcancel *PQgetCancel(PGconn *conn);
```

给出一个PQgetCancel连接对象，PQgetCancel创建一个 PGcancel对象。如果给出 的conn是NULL或者是一个无效的连接，那么它将返回NULL。PGcancel对象是一个不透明的结构， 不应该为应用所直接访问；我们只能把它传递给PQcancel或者PQfreeCancel。 给定一个PGconn连接对象，PQgetCancel创建一个PGcancel对象。如果给定的conn为NULL或者一个不合法的连接，它将返回NULL。PGcancel对象是一个透明的结构，它不能直接被应用访问。它只能被传递给PQcancel或PQfreeCancel。

PQfreeCancel

释放一个由PQgetCancel创建的数据结构。

```
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancel释放一个由前面的PQgetCancel创建的数据对象。 PQfreeCancel释放一个之前由PQgetCancel创建的数据对象。

PQcancel

要求服务器放弃当前命令的处理。

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

如果取消请求成功发送，则返回值为 1，否则为 0。如果不成功，则errbuf会被填充一个解释性的错误消息。errbuf必须是一个尺寸为errbufsize的字符数组（推荐尺寸为 256 字节）。，解释为何不成功。errbuf必须是 一个大小为errbufsize的 char 数组（建议大小为 256 字节）。

不过，成功的发送并不保证请求会有任何效果。如果取消有效，那么当前的命令将提前终止并且返回一个错误结果。如果取消失败（也就是说， 因为服务器已经完成命令的处理），那么就根本不会有可见的结果。

如果PQcancel是信号句柄里的一个局部变量，那么PQcancel可以在一个信号 句柄里安全地调用。在PQcancel涉及的范围里，PQcancel对象都是只读的， 因此我们也可以从一个与处理PGconn对象的线程分离的线程里处理它。 如果errbuf是信号处理器中的一个局部变量，PQcancel可以从一个信号处理器中安全地调用。在PGcancel有关的范围内，PQcancel都是只读的，因此也可以在一个从操纵PGconn对象的线程中独立出来的线程中调用它。

PQrequestCancel

PQrequestCancel是PQcancel的一个被废弃的变体。

```
int PQrequestCancel(PGconn *conn);
```

要求服务器放弃当前命令的处理。它直接在PGconn对象上进行操作， 并且如果失败，就会在PGconn对象里存储错误消息（因此可以用PQerrorMessage检索出来）。 尽管功能相同，这个方法在多线程程序里和信号处理器里会带来危险，因为它可能 覆盖PGconn的错误消息，进而将当前连接上正在处理的操作搞乱。

34.7. 快速路径接口

PostgreSQL提供一种快速路径接口来向服务器发送简单的函数调用。

> **提示**
>
> 这个接口在某种程度上已被废弃，因为我们可以通过创建一个定义该函数调用的预备语句来达到类似或者更强大的功能。然后，用参数和结果的二进制传输执行该语句，从而取代快速函数调用。

函数PQfn请求通过快速路径接口执行服务器函数。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

fnid参数是要被执行的函数的 OID。args和nargs定义了要传递给函数的参数；它们必须匹配已声明的函数参数列表。当一个参数结构的isint域为真时，u.integer值被以指定长度（必须是 1、2 或者 4 字节）整数的形式发送给服务器；这时候会发生恰当的字节交换。当isint为假时，*u.ptr中指定数量的字节将不做任何处理被发送出去；这些数据必须是服务器 预期的用于该函数参数数据类型的二进制传输的格式（由于历史原因u.ptr被声明为类型int *，其实把它考虑成void *会更好）。result_buf是放置该函数返回值的缓冲区。调用者必须已经分配了足够的空间来存储返回值（这里没有检查！）。实际的结果长度将被放在result_len指向的整数中返回。如果预期结果是 2 或 4 字节整数，把result_is_int设为 1；否则设为 0。把result_is_int设为 1 导致libpq在必要时对值进行交换字节，这样它就作为对客户端机器正确的int值被传输，注意对任一种允许的结果大小都会传递一个 4 字节到*result_buf。当result_is_int是 0 时，服务器发送的二进制格式字节将不做修改直接返回（在这种情况下，把result_buf考虑为类型void *更好）。

PQfn总是返回一个有效的PGresult指针。在使用结果之前应该检查结果状态。当结果不再使用后，调用者有义务使用PQclear释放PGresult。

注意我们没办法处理空参数、空结果，也没办法在使用这个接口时处理集值结果。

34.8. 异步提示

PostgreSQL通过LISTEN和NOTIFY命令提供了异步通知。一个客户端会话用LISTEN命令在一个特定的通知频道中注册它感兴趣的通知（也可以用UNLISTEN命令停止监听）。当任何会话执行一个带有特定频道名的NOTIFY命令时，所有正在监听该频道的会话会被异步通知。可以传递一个“载荷”字符串来与监听者沟通附加的数据。

libpq应用把LISTEN、UNLISTEN和NOTIFY命令作为通常的 SQL 命令提交。 随后通过调用PQnotifies来检测NOTIFY消息的到达。

函数PQnotifies从来自服务器的未处理通知消息列表中返回下一个通知。如果没有待处理的信息则返回一个空指针。一旦PQnotifies返回一个通知，该通知会被认为已处理并且将被从通知列表中删除。

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* notification channel name */
    int  be_pid;                /* process ID of notifying server process */
    char *extra;                /* notification payload string */
} PGnotify;
```

在处理完PQnotifies返回的PGnotify对象后，别忘了用PQfreemem把它释放。释放PGnotify指针就足够了；relname和extra域并不代表独立分配的内存（这些域的名称是历史性的，尤其是频道名称与关系名称没有什么联系）。

例 34.2给出了一个例子程序展示异步通知的使用。

PQnotifies实际上并不从服务器读取数据；它只是返回被另一个libpq函数之前吸收的消息。在以前的libpq版本中，及时收到NOTIFY消息的唯一方法是持续地提交命令，即使是空命令也可以，并且在每次PQexec后检查PQnotifies。 虽然这个方法还能用，但是由于太过浪费处理能力已被废弃。

当你没有可用的命令提交时，一种更好的检查NOTIFY消息的方法是调用PQconsumeInput，然后检查PQnotifies。你可以使用select()来等待服务器数据到达，这样在无事可做时可以不浪费CPU能力（参考PQsocket来获得用于select()的文件描述符）。注意不管是用PQsendQuery/PQgetResult提交命令还是简单地使用PQexec，这种方法都能正常工作。不过，你应该记住在每次PQgetResult或PQexec之后检查PQnotifies，看看在命令的处理过程中是否有通知到达。

34.9. COPY命令相关的函数

PostgreSQL中的COPY命令有用于libpq的对网络连接读出或者写入的选项。这一节描述的函数允许应用通过提供或者消耗已拷贝的数据来充分利用这个功能。

整个处理是应用首先通过PQexec或者一个等效的函数发出 SQL COPY命令。对这个命令的响应（如果命令无误）将是一个状态代码是PGRES_COPY_OUT或 者PGRES_COPY_IN（取决于指定的拷贝方向）的PGresult对象。应用然后就应该使用这一节的函数接收或者传送数据行。在数据传输结束之后，另外一个PGresult对象会被返回以表明传输的成功或者失败。它的状态将是：PGRES_COMMAND_OK表示成功，PGRES_FATAL_ERROR表示发生了一些问题。此时我们可以通过PQexec发出进一步的 SQL 命令（在COPY操作的处理过程中，不能用同一个连接执行其它 SQL 命令）。

如果一个COPY命令是通过PQexec在一个可能包含额外命令的字符串中发出的，那么应用在完成COPY序列之后必须继续用PQgetResult取得结果。只有在PQgetResult返回NULL时，我们才能确信PQexec的命令字符串已经处理完毕， 并且可以安全地发出更多命令。

这一节的函数应该只在从PQexec或PQgetResult获得了PGRES_COPY_OUT或PGRES_COPY_IN结果状态的后执行。

一个承载了这些状态值之一的PGresult对象携带了正在开始的COPY操作的一些额外数据。这些额外的数据可以用于那些与带查询结果的连接一起使用的函数：

PQnfields

返回要拷贝的列（域）的个数。

PQbinaryTuples

0 表示整体拷贝格式都是文本（行用新行分隔，列用分隔字符分隔等等）。1 表示整体拷贝格式都是二进制。详见COPY。

PQfformat

返回与拷贝操的每列相关的格式代码（0 是文本，1 是二进制）。当整体拷贝格式是文本时，那么每列的格式代码将总是零，但是二进制格式可以同时支持文本和二进制列（不过，就目前的COPY实现而言，二进制拷贝中只会出现二进制列；所以目前每列的格式总是匹配总体格式）。

> **注意**
>
> 这些额外的数据值只在使用协议 3.0 时可用。在使用协议 2.0 时，所有这些函数都返回 0。

34.9.1. 用于发送COPY数据的函数

这些函数用于在COPY FROM STDIN期间发送数据。如果在连接不是COPY_IN状态，调用它们会失败。

PQputCopyData

在COPY_IN状态中向服务器发送数据。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

传输指定buffer中长度为nbytes的COPY数据到服务器。如果数据被放在队列中，结果是1；如果因为缓冲区满而无法被放在队列中（只可能发生在连接是非阻塞模式时），那么结果是零；如果发生错误，结果为 -1（如果返回值为 -1，那么使用PQerrorMessage检索细节。如果值是零，那么等待写准备好然后重试）。

应用可以把COPY数据流划分成任意方便的大小放到缓冲区中。在发送时，缓冲区载荷的边界没有什么语意。数据流的内容必须匹配COPY命令预期的数据格式；详见COPY。

PQputCopyEnd

在COPY_IN状态中向服务器发送数据结束的指示。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

如果errormsg是NULL，则成功结束COPY_IN操作。如果errormsg不是NULL则COPY被强制失败，errormsg指向的字符串是错误消息（不过，我们不应假定这个准确的错误信息将会从服务器传回，因为服务器可能已经因为其自身原因导致COPY失败。还要注意的是在使用 3.0 协议之前的连接时，强制失败的选项是不能用的）。

如果终止消息被发送，则结果为 1；在非阻塞模式中，结果为 1 也可能只表示终止消息被成功地放在了发送队列中（在非阻塞模式中，要确认数据确实被发送出去，你应该接着等待写准备好并且调用PQflush，重复这些直到返回零）。零表示该函数由于缓冲区满而无法将该终止消息放在队列中，这只会发生在非阻塞模式中（在这种情况下，等待写准备好并且再次尝试PQputCopyEnd调用）。如果发生系统错误，则返回 -1，可以使用PQerrorMessage检索详情。

在成功调用PQputCopyEnd之后，调用PQgetResult获取COPY命令的最终结果状态。我们可以用平常的方法来等待这个结果可用。然后返回到正常的操作。

34.9.2. 用于接收COPY数据的函数

这些函数用于在COPY TO STDOUT的过程中接收数据。如果连接不在COPY_OUT状态，那么调用它们将会失败。

PQgetCopyData

在COPY_OUT状态下从服务器接收数据。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

在一个COPY期间尝试从服务器获取另外一行数据。数据总是以每次一个数据行的方式被返回；如果只有一个部分行可用，那么它不会被返回。成功返回一个数据行涉及到分配一块内存来保存该数据。buffer参数必须为非NULL。*buffer被设置为指向分配到的内存的指针，或者是在没有返回缓冲区的情况下指向NULL。一个非NULL的结果缓冲区在不需要时必须用PQfreemem释放。

在成功返回一行之后，返回的值就是该数据行里数据的字节数（将是大于零）。被返回的字符串总是空终止的，虽然这可能只是对文本COPY有用。 一个零结果表示该COPY仍然在处理中，但是还没有可用的行（只在async为真时才可能）。一个 -1 结果表示COPY已经完成。-2 结果表示发生了错误（参考PQerrorMessage获取原因）。

当async为真时（非零），PQgetCopyData将不会阻塞等待输入；如果COPY仍在处理过程中并且没有可用的完整行，那么它将返回零（在这种情况下等待读准备好，然后在再次调用PQgetCopyData之前，调用PQconsumeInput）。当async为假（零）
时，PQgetCopyData将阻塞，直到数据可用或者操作完成。

在PQgetCopyData返回 -1 之后，调用PQgetResult获取COPY命令的最后结果状态。我们可以用平常的方法来等待这个结果可用。然后返回到正常的操作。

34.9.3. 用于COPY的废弃函数

这些函数代表了以前的处理COPY的方法。尽管它们还能用，但是现在已经被废弃，因为它们的错误处理很糟糕、检测结束数据的方法也不方便，并且缺少对二进制或非阻塞传输的支持。

PQgetline

读取一个以新行终止的字符行到（由服务器传输） 到一个长度为length的字符串缓冲区。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

这个函数拷贝最多length-1 个字符到该缓冲区中，并且把终止的新行转换成一个零字节。PQgetline在输入结束时返回EOF，如果整行都被读取则返回 0，如果缓冲区填满了而还没有遇到结束的新行则返回 1。

注意，应用必须检查是否一个新行包含两个字符\.，这表明服务器 已经完成了COPY命令的结果发送。如果应用可能收到超过length-1 字符长的行， 我们就应该确保正确识别\.行（例如，不要把一个长数据行的结束当作一个终止行）。

PQgetlineAsync

不阻塞地读取一行COPY数据（由服务器传输）到一个缓冲区中。

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

这个函数类似于PQgetline，但是可以被用于那些必须异步读取COPY数据的应用， 也就是不阻塞的应用。在发出了COPY命令并得到了PGRES_COPY_OUT响应之后，应用应该调用PQconsumeInput和PQgetlineAsync直到检测到结束数据的信号。

不像PQgetline，这个函数负责检测结束数据。

在每次调用时，如果libpq的输入缓冲区中有一个完整的数据行可用，PQgetlineAsync都将返回数据。否则，在剩余行到达之前不会返回数据。如果识别到拷贝数据结束的标志，此函数返回 -1；如果没有可用数据则返回 0； 或者是给出一个正数给出被返回的字节数。如果返回 -1，调用者下一步必须调用PQendcopy，然后回到正常处理。

返回的数据将不超过一个数据行的范围。如果可能，每次将返回一个完整行。但如果调用者提供的缓冲区太小不足以容下服务器发送的行，那么将返回部分行。对于文本数据，这可以通过测试返回的最后一个字节是否\n来检测（在二进制COPY中， 需要对COPY数据格式进行实际的分析，以便做相同的判断）。被返回的字符串不是空结尾的（如果你想增加一个终止空，确保传递一个比实际可用空间少一字节的bufsize）。

PQputline

向服务器发送一个空终止的字符串。如果 OK 则返回 0；如果不能发送字符串则返回EOF。

```
int PQputline(PGconn *conn,
              const char *string);
```

一系列PQputline调用发送的COPY数据流和PQgetlineAsync返回的数据具有相同的格式，只是应用不需要每次PQputline调用中发送刚好一个数据行；在每次调用中发送多行或者部分行都是可以的。

> **注意**
>
> 在PostgreSQL协议 3.0 之前，应用必须显式地发送两个字符\.作为最后一行来指示服务器已经完成发送COPY数据。虽然这么做仍然有效，但是它已经被废弃并且\.的特殊含义可能在将来的版本中删除。在发送完实际数据之后， 调用PQendcopy就足够了。

PQputnbytes

向服务器发送一个非空终止的字符串。如果 OK 则返回 0，如果不能发送字符串则返回EOF。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

这个函数类似PQputline，除了数据缓冲区不需要是空终止，因为要发送的字节数是直接指定的。在发送二进制数据时使用这个过程。

PQendcopy

与服务器同步。

```
int PQendcopy(PGconn *conn);
```

这个函数等待服务器完成拷贝。当最后一个字符串已经用PQputline发送给服务器时或者当最后一个字符串已经用PGgetline从服务器接收到时，就会发出这个函数。这个函数必须被发出，否则服务器将会和客户端“不同步”。从这个函数返回后，服务器就已经准备好接收下一个 SQL 命令了。函数成功完成时返回值为 0，否则返回非零值（如果返回值为非零值，用PQerrorMessage检索详情）。

在使用PQgetResult时，应用应该通过反复调用PQgetline并且在看到终止行后调用PQendcopy来响应PGRES_COPY_OUT结果。然后它应该返回到PQgetResult循环直到PQgetResult返回一个空指针。类似地，PGRES_COPY_IN结果会用一系列PQputline加上之后的PQendcopy来处理，然后返 回到PQgetResult循环。这样的安排将保证嵌入到一系列SQL命令中的COPY命令将被正确执行。

旧的应用很可能会通过PQexec提交一个COPY命令并且假定事务在PQendcopy之后完成。只有在COPY是命令字符串中唯一的SQL命令时才能正确工作。

34.10. 控制函数

这些函数控制libpq行为各种各样的细节。

PQclientEncoding

返回客户端编码。

```
int PQclientEncoding(const PGconn *conn);
```

请注意，它返回的是编码 ID，而不是一个符号串字符串，如EUC_JP。如果不成功，它会返回 -1。要把一个编码 ID 转换为为一个编码名称，可以用：

```
char *pg_encoding_to_char(int encoding_id);
```

PQsetClientEncoding

设置客户端编码。

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

conn是一个到服务器的连接，而encoding是你想使用的编码。如果函数成功地设置编码，则返回 0，否则返回 -1。这个连接的当前编码可以使用PQclientEncoding确定。

PQsetErrorVerbosity

决定PQerrorMessage和PQresultErrorMessage返回的消息的细节程度。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;
```

```
PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity设置细节模式，并返回该连接的前一个设置。在TERSE模式下，返回的消息只包括严重性、主要文本以及位置；这些东西通常放在一个单一行中。缺省模式生成的消息包括上面的信息加上任何细节、提示或者上下文域（这些可能跨越多行）。VERBOSE模式包括所有可以可用的域。修改细节模式不会影响来自已有PGresult对象中的可用消息。只有随后创建的PGresult对象才受到影响（如果想要用不同的详细程度打印之前的错误，请见PQresultVerboseErrorMessage）。

PQsetErrorContextVisibility

决定如何处理PQerrorMessage和PQresultErrorMessage返回的消息中的CONTEXT域。

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn,
 PGContextVisibility show_context);
```

PQsetErrorContextVisibility设置上下文显示模式，返回该连接上之前的设置。这个模式控制消息中是否包括CONTEXT域（除非 verbosity 设置是TERSE，那种情况下CONTEXT不会被显示）。NEVER模式不会包括CONTEXT，而ALWAYS则尽可能地包括这个域。在ERRORS模式（默认）中，只在错误消息中包括CONTEXT域，而在通知和警告消息中不会包括。更改这个模式不会影响从已经存在的PGresult对象项中得到的消息，只会影响后续创建的PGresult对象（如果想要用不同的详细程度打印之前的错误，请见PQresultVerboseErrorMessage）。

PQtrace

启用对客户端/服务器通讯的跟踪，把跟踪信息输出到一个调试文件流中。

```
void PQtrace(PGconn *conn, FILE *stream);
```

> **注意**
>
> 在 Windows上，如果libpq库和应用使用了不同的标志编译，那么这个函数调用会导致应用崩溃，因为FILE指针的内部表达是不一样的。特别是多线程/单线程、发布/调试 以及静态/动态标志应该是库和所有使用库的应用都一致。

PQuntrace

禁用PQtrace打开的跟踪。

```
void PQuntrace(PGconn *conn);
```

34.11. 杂项函数

一如往常，总有一些函数不适合放在任何其他地方。

PQfreemem

释放libpq分配的内存。

```
void PQfreemem(void *ptr);
```

释放libpq分配的内存，尤其是PQescapeByteaConn、PQescapeBytea、PQunescapeBytea和PQnotifies分配的内存。特别重要的是，在微软 Windows 上使用这个函数，而不是free()。这是因为只有 DLL 和应用的当多线程/单线程、发布/调试以及静态/动态标志相同时，才能在一个 DLL 中分配内存并且在应用中释放它。在非微软 Windows 平台上，这个函数与标准库函数free()相同。

PQconninfoFree

释放PQconndefaults或PQconninfoParse分配的数据结构。

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

一个简单的PQfreemem不会做这些，因为数组包含对子字符串的引用。

PQencryptPasswordConn

准备一个PostgreSQL口令的加密形式。

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char
 *user, const char *algorithm);
```

这个函数旨在用于那些希望发送类似于ALTER USER joe PASSWORD 'pwd'命令的客户端应用。不在这样一个命令中发送原始的明文密码是一个好习惯，因为它可能被暴露在命令日志、活动显示等等中。相反，在发送之前使用这个函数可以将口令转换为加密的形式。

passwd和user参数是明文口令以及用户的SQL名称。algorithm指定用来加密口令的加密算法。当前支持的算法是md5和scram-sha-256（on和off也被接受作为md5的别名，用于与较老的服务器版本兼容）。注意，对scram-sha-256支持是在PostgreSQL版本10中引入的，并且在老的服务器版本上无法工作。如果algorithm是NULL，这个函数将向服务器查询password_encryption设置的当前值。这种行为可能会阻塞当前事务，并且当前事务被中止或者连接正忙于执行另一个查询时会失败。如果希望为服务器使用默认的算法但避免阻塞，应在调用PQencryptPasswordConn之前查询你自己的password_encryption，并且将该值作为algorithm传入。

返回值是一个由malloc分配的字符串。调用者可以假设该字符串不含有需要转义的任何特殊字符。在处理完它之后，用PQfreemem释放结果。发生错误时，返回的是NULL，并且适当的消息会被存储在连接对象中。

PQencryptPassword

准备一个PostgreSQL口令的md5加密形式。

```
char *PQencryptPassword(const char *passwd, const char *user);
```

PQencryptPassword是PQencryptPasswodConn的一个较老的已经被废弃的版本。其差别是PQencryptPassword不要求一个连接对象，并且总是用md5作为加密算法。

PQmakeEmptyPGresult

用给定的状态，构造一个空PGresult对象。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是libpq内部用于分配并初始化一个空PGresult对象的函数。如果不能分配内存，那么这个函数返回NULL。它也是可以对外使用的，因为一些应用认为它可以用于产生结果对象（特别是带有错误状态的对象）本身。如果conn非空，并且status表示一个错误，那么指定连接的当前错误消息会被复制到PGresult中。如果conn非空，那么连接中的任何已注册事件过程也会被复制到PGresult中（它们不会获得PGEVT_RESULTCREATE调用，但会看到PQfireResultCreateEvents）。注意在该对象上最终应该调用PQclear，正如对libpq本身返回的PGresult对象所作的那样。

PQfireResultCreateEvents

为每一个在PGresult对象中注册的事件过程触发一个PGEVT_RESULTCREATE事件（见第 34.13 节。成功时返回非 0，如果任何事件过程失败则返回 0。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

conn参数被传送给事件过程，但不会被直接使用。如果事件过程不使用它，则会返回NULL。

已经接收到这个对象的PGEVT_RESULTCREATE或PGEVT_RESULTCOPY事件的事件过程不会被再次触发。

这个函数与PQmakeEmptyPGresult分开的主要原因是在调用事件过程之前创建一个PGresult并且填充它常常是合适的。

PQcopyResult

为一个PGresult对象创建一个拷贝。这个拷贝不会以任何方式链接到源结果，并且当该拷贝不再需要时，必须调用PQclear进行清理。如果函数失败，返回NULL。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

这个函数的意图并非是制作一个准确的拷贝。返回的结果总是会被放入PGRES_TUPLES_OK状态，并且不会拷贝来源中的任何错误消息（不过它确实会拷贝命令状态字符串）。flags参数决定还要拷贝些什么。它通常是几个标志的按位OR。PG_COPYRES_ATTRS指定复制源结果的属性（列定义）。PG_COPYRES_TUPLES指定复制源结果的元组（这也意味着复制属性）。PG_COPYRES_NOTICEHOOKS指定复制源结果的提醒钩子。PG_COPYRES_EVENTS指定复制源结果的事件（但是不会复制与源结果相关的实例数据）。

PQsetResultAttrs

设置PGresult对象的属性。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc
 *attDescs);
```

提供的attDescs被复制到结果中。如果attDescs指针为NULL或numAttributes小于1，那么请求将被忽略并且函数成功。如果res已经包含属性，那么函数会失败。如果函数失败，返回值是 0。如果函数成功，返回值是非 0。

PQsetvalue

设置一个PGresult对象的一个元组域值。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int
 len);
```

这个函数将自动按需增加结果的内置元组数组。但是，tup_num参数必须小于等于PQntuples，意味着这个函数对元组数组一次只能增加一个元组。但已存在的任意元组中的任意域可以以任意顺序进行调整。如果field_num的一个值已经存在，它会被覆盖。如果len是 -1，或value是NULL， 该域值会被设置为一个 SQL 空值。value会被复制到结果的私有存储中，因此函数返回后就不再需要了。如果函数失败，返回值是 0。如果函数成功，返回值会是非 0。

PQresultAlloc

为一个PGresult对象分配附属存储。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

当res被清除时，这个函数分配的内存也会被释放掉。如果函数失败，返回值是NULL。结果被保证为按照数据的任意类型充分地对齐，正如malloc所作的。

PQlibVersion

返回所使用的libpq版本。

```
int PQlibVersion(void);
```

在运行时，这个函数的结果可以被用来决定在当前已载入的 libpq 版本中特定的功能是否可用。例如，这个函数可以被用来决定哪些选项可以被用于PQconnectdb。

结果通过将库的主版本号乘以10000再加上次版本号形成。例如，版本10.1将被返回为100001，而版本11.0将被返回为110000。

在主版本10之前，PostgreSQL采用一种由三个部分组成的版本号，其中前两部分共同表示主版本。对于那些版本，PQlibVersion为每个部分使用两个数字，例如版本9.1.5将被返回为90105，而版本9.2.0将被返回为90200。

因此，出于判断特性兼容性的目的，应用应该将PQlibVersion的结果除以100而不是10000来判断逻辑的主版本号。在所有的发行序列中，只有最后两个数字在次发行（问题修正发行）之间不同。

> **注意**
>
> 这个函数出现于PostgreSQL版本 9.1，因此它不能被用来在早期的版本中检测所需的功能，因为调用它将会创建一个对版本9.1及其后版本的链接依赖。

34.12. 通知处理

服务器产生的通知和警告消息不会被查询执行函数返回，因为它们不代表查询失败。它们可以被传递给一个通知处理函数，并且在处理者返回后执行会继续正常进行。默认的处理函数会把消息打印在stderr上，但是应用可以通过提供它自己的处理函数来重载这种行为。

由于历史原因，通知处理有两个级别，称为通知接收器和通知处理器。通知接收器的默认行为是格式化通知并且将一个字符串传递给通知处理器来打印。不过，如果一个应用选择提供自己的通知接收器，它通常会忽略通知处理器层并且在通知接收器中完成所有工作。

函数PQsetNoticeReceiver 为一个连接对象设置或者检查当前的通知接收器。相似地，PQsetNoticeProcessor 设置或检查当前的通知处理器。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

这些函数中的每一个会返回之前的通知接收器或处理器函数指针，并且设置新值。如果你提供了一个空函数指针，将不会采取任何动作，只会返回当前指针。

当接收到一个服务器产生的或者libpq内部产生的通知或警告消息，通知接收器函数会被调用。它会以一种PGRES_NONFATAL_ERROR PGresult的形式传递该消息（这允许接收器使用PQresultErrorField抽取个别的域，或者使用PQresultErrorMessage或者PQresultVerboseErrorMessage得到一个完整的预格式化的消息）。被传递给PQsetNoticeReceiver的同一个空指针也被传递（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知接收器会简单地抽取消息（使用PQresultErrorMessage）并且将它传递给通知处理器。

通知处理器负责处理一个以文本形式给出的通知或警告消息。该消息的字符串文本（包括一个收尾的新行）被传递给通知处理器，外加一个同时被传递给PQsetNoticeProcessor的空指针（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知处理器很简单：

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

一旦你设定了一个通知接收器或处理器，你应该期待只要PGconn对象或者从它构造出的PGresult对象存在，该函数就应该能被调用。在一个PGresult创建时，PGconn的当前通知处理指针被复制到PGresult中，以备类似PQgetvalue的函数使用。

34.13. 事件系统

libpq的事件系统被设计为通知已注册的事件处理器它感兴趣的libpq事件，例如PGconn以及PGresult对象的创建和毁灭。一种主要的使用情况是这允许应用将自己的数据与一个PGconn或者PGresult关联在一起，并且确保那些数据在适当的时候被释放。

每一个已注册的事件处理器与两部分数据相关，对于libpq它们只是透明的void *指针。当事件处理器被注册到一个PGconn时，会有一个应用提供的转移指针。该转移指针在PGconn及其产生的所有PGresult的生命期内都不会改变。因此，如果使用它，它必须指向长期存在的数据。此外，还有一个instance data指针，它在每一个PGconn和PGresult中都开始于NULL。这个指针可以使用 PQinstanceData、 PQsetInstanceData、 PQresultInstanceData和 PQsetResultInstanceData函数操纵。注意和转移指针不同，一个PGconn的实例数据不会被从它创建的PGresult自动继承。libpq不知道转移和实例数据指针指向的是什么（如果有），并且将不会尝试释放它们 — 那是事件处理器的责任。

34.13.1. 事件类型

枚举PGEventId命名了事件系统处理的事件类型。它的所有值的名称都以PGEVT开始。对于每一种事件类型，都有一个相应的事件信息结构用来承载传递给事件处理器的参数。事件类型是：

PGEVT_REGISTER

当PQregisterEventProc被调用时，注册事件会发生。这是一个初始化每一个事件过程都可能需要的instanceData的最佳时机。每个连接的每个事件处理器只会触发一个注册事件。如果该事件过程失败，注册会被中止。

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

当收到一个PGEVT_REGISTER事件时，evtInfo指针应该被造型为PGEventRegister *。这个结构包含一个状态应该为CONNECTION_OK的PGconn，保证在得到一个良好的PGconn之后能马上调用PQregisterEventProc。当返回一个失败代码时，所有的清理都必须被执行而不会发送PGEVT_CONNDESTROY事件。

PGEVT_CONNRESET

连接重置事件在PQreset或PQresetPoll完成时被触发。在两种情况中，只有重置成功才会触发该事件。如果事件过程失败，整个连接重置将失败，PGconn会被置为CONNECTION_BAD状态并且PQresetPoll将返回PGRES_POLLING_FAILED。

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

当收到一个PGEVT_CONNRESET事件时，evtInfo指针应该被造型为PGEventConnReset *。尽管所包含的PGconn刚被重置，所有的事件数据还是保持不变。这个事件应该被用来重置/重载/重新查询任何相关的instanceData。注意即使事件过程无法处理PGEVT_CONNRESET，它仍将在连接被关闭时接收到一个PGEVT_CONNDESTROY事件。

PGEVT_CONNDESTROY

为了响应PQfinish，连接销毁事件会被触发。由于 libpq 没有能力管理事件数据，事件过程有责任正确地清理它的事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

当接收到一个PGEVT_CONNDESTROY事件时，evtInfo指针应该被造型为PGEventConnDestroy *。这个事件在PQfinish执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示一个来自PQfinish的失败。还有，一个事件过程失败不该中断对不需要的内存的清理。

PGEVT_RESULTCREATE

为了响应任何生成一个结果的查询执行函数，结果创建事件会被触发。这些函数包括PQgetResult。这个事件只有在结果被成功地创建之后才会被触发。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

当接收到一个PGEVT_RESULTCREATE事件时，evtInfo指针应该被造型为PGEventResultCreate *。conn是用来产生结果的连接。这是初始化任何需要与结果关联的instanceData的理想位置。如果该事件过程失败，结果将被清除并且失败将会被传播。该事件过程不能尝试自己PQclear结果对象。当返回一个失败代码时，所有清理必须被执行而不会发送PGEVT_RESULTDESTROY事件。

PGEVT_RESULTCOPY

为了响应PQcopyResult，结果复制事件会被触发。这个事件只会在复制完成后才被触发。只有成功地处理了PGEVT_RESULTCREATE和PGEVT_RESULTCOPY事件的事件过程才将会收到PGEVT_RESULTCOPY事件。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

当收到一个PGEVT_RESULTCOPY事件时，evtInfo指针应该被造型为PGEventResultCopy *。src结果是要被复制的，而dest结果则是复制的目的地。这个事件可以被用来提供instanceData的一份深度副本，因为PQcopyResult没法这样做。如果该事件过程失败，整个复制操作将失败并且dest结果将被清除。当返回一个失败代码时，所有清理必须被执行而不会为目标结果发送PGEVT_RESULTDESTROY事件。

PGEVT_RESULTDESTROY

为了响应PQclear，结果销毁事件会被触发。由于 libpq 没有能力管理事件数据，事件过程有责任正确地清理它的事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

当接收到一个PGEVT_RESULTDESTROY事件时，evtInfo指针应该被造型为PGEventResultDestroy *。这个事件在PQclear执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示来自PQclear的失败。还有，一个事件过程失败不该中断不需要的内存的清理过程。

34.13.2. 事件回调函数

PGEventProc

PGEventProc是到一个事件过程的指针的 typedef，也就是从 libpq 接收事件的用户回调函数。一个事件过程的原型必须是

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

evtId指示发生了哪一个PGEVT事件。evtInfo指针必须被造型为合适的结构类型才能获得关于事件的进一步信息。当事件过程已被注册时，passThrough参数是提供给PQregisterEventProc的指针。如果成功，该函数应该返回非零值，失败则返回零。

在任何一个PGconn中，一个特定事件过程只能被注册一次。这是因为该过程的地址被用作查找键来标识相关的实例数据。

> **小心**
>
> 在 Windows 上，函数能够有两个不同的地址：一个对 DLL 之外可见而另一个对 DLL 之内可见。我们应当小心只有其中之一会被用于libpq的事件过程函数，否则将会产生混淆。编写代码的最简单规则是将所有的事件过程声明为static。如果过程的地址必须对它自己的源代码文件之外可见，提供一个单独的函数来返回该地址。

34.13.3. 事件支持函数

PQregisterEventProc

为 libpq 注册一个事件回调过程。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

在每一个你想要接收事件的PGconn上必须注册一个事件过程。和内存不同，没有限制说一个连接上能注册多少个事件过程。如果该函数成功，它会返回一个非零值。如果它失败，则会返回零。

当一个 libpq 事件被触发时，proc参数将被调用。它的内存地址也被用来查找instanceData。name参数被用来在错误消息中引用该事件过程。这个值不能是NULL或一个零长度串。名字串被复制到PGconn中，因此传递进来的东西不需要长期存在。当一个事件发生时，passThrough指针被传递给proc。这个参数可以是NULL。

PQsetInstanceData

设置连接conn的用于过程proc的instanceData为data。它在成功时返回非零值，失败时返回零（只有proc没有被正确地注册在conn中，才可能会失败）。

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

PQinstanceData

返回连接conn的与过程proc相关的instanceData，如果没有则返回NULL。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

PQresultSetInstanceData

把结果的用于proc的instanceData设置为data。成功返回非零，失败返回零（只有proc没有被正确地注册在conn中，才可能会失败）。

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

PQresultInstanceData

返回结果的与过程proc相关的instanceData，如果没有则返回NULL。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

34.13.4. 事件实例

这里是一个管理与 libpq 连接和结果相关的私有数据的例子的框架。

```
/* 要求 libpq 事件的头文件（注意：包括 libpq-fe.h） */
#include <libpq-events.h>

/* instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PPGconn *conn =
        PQconnectdb("dbname=postgres options=--csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* 在任何应该接收事件的连接上调用一次。
     * 发送一个 PGEVT_REGISTER 给 myEventProc。
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* conn 的 instanceData 可用 */
    data = PQinstanceData(conn, myEventProc);

    /* 发送一个 PGEVT_RESULTCREATE 给 myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* 结果的 instanceData 可用 */
    data = PQresultInstanceData(res, myEventProc);

    /* 如果使用了 PG_COPYRES_EVENTS，发送一个 PGEVT_RESULTCOPY 给 myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);
```

```
    /* 如果在 PQcopyResult 调用时使用了 PG_COPYRES_EVENTS，结果的 instanceData
 可用。*/
    data = PQresultInstanceData(res_copy, myEventProc);

    /* 两个清除都发送一个 PGEVT_RESULTDESTROY 给 myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* 发送一个 PGEVT_CONNDESTROY 给 myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* 将应用相关的数据与连接关联起来 */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* 因为连接正在被销毁，释放示例数据 */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* 把应用相关的数据与结果（从 conn 复制过来）关联起来 */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
```

```
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* 把应用相关的数据与结果（从一个结果复制过来）关联起来 */
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            /* 因为结果正在被销毁，释放实例数据 */
            if (data)
              free_mydata(data);
            break;
        }

        /* 未知事件 ID，只返回true。 */
        default:
            break;
    }

    return true; /* 事件处理成功 */
}
```

34.14. 环境变量

下列环境变量能被用于选择默认的连接参数值，如果调用代码没有直接指定值，它们将被用于PQconnectdb、PQsetdbLogin和PQsetdb。例如，这些有助于防止数据库连接信息被硬编码到简单的客户端应用中。

- PGHOST的行为和host连接参数相同。
- PGHOSTADDR的行为和hostaddr连接参数相同。可以设置它来替代或者作为PGHOST的补充来防止 DNS 查找负担。
- PGPORT的行为和port连接参数相同。
- PGDATABASE的行为和dbname连接参数相同。
- PGUSER的行为和user连接参数相同。
- PGPASSWORD的行为和password连接参数相同。出于安全原因，我们不推荐使用这个环境变量，因为某些操作系统允许非根用户通过ps看到进程的环境变量。可以考虑使用一个口令文件（见第 34.15 节。
- PGPASSFILE的行为和passfile连接参数相同。
- PGSERVICE的行为和service连接参数相同。

- PGSERVICEFILE指定针对每个用户的连接服务文件名。如果没有设置，默认为~/.pg_service.conf（见第 34.16 节。
- PGOPTIONS的行为和options连接参数相同。
- PGAPPNAME的行为和application_name连接参数相同。
- PGSSLMODE的行为和sslmode连接参数相同。
- PGREQUIRESSL的行为和requiressl连接参数相同。为了支持PGSSLMODE变量，这个环境变量已被废弃。同时设置两个变量会抑制这一个的效果。
- PGSSLCOMPRESSION的行为和sslcompression连接参数相同。
- PGSSLCERT的行为和sslcert连接参数相同。
- PGSSLKEY的行为和sslkey连接参数相同。
- PGSSLROOTCERT的行为和sslrootcert连接参数相同。
- PGSSLCRL的行为和sslcrl连接参数相同。
- PGREQUIREPEER的行为和requirepeer连接参数相同。
- PGKRBSRVNAME的行为和krbsrvname连接参数相同。
- PGGSSLIB的行为和gsslib连接参数相同。
- PGCONNECT_TIMEOUT的行为和connect_timeout连接参数相同。
- PGCLIENTENCODING的行为和client_encoding连接参数相同。
- PGTARGETSESSIONATTRS的行为和target_session_attrs连接参数相同。

下面的环境变量可用来为每一个PostgreSQL会话指定默认行为（为每一个用户或每一个数据库设置默认行为的方法还可见ALTER ROLE和ALTER DATABASE命令）。

- PGDATESTYLE设置日期/时间表示的默认风格（等同于SET datestyle TO ...）。
- PGTZ设置默认的时区（等同于SET timezone TO ...）。
- PGGEQO为遗传查询优化器设置默认模式（等同于SET geqo TO ...）。

这些环境变量的正确值可参考SQL 命令 SET。

下面的环境变量决定libpq的内部行为，它们会覆盖编译在程序中的默认值。

- PGSYSCONFDIR设置包含pg_service.conf文件以及未来版本中可能出现的其他系统范围配置文件的目录。
- PGLOCALEDIR设置包含用于消息本地化的locale文件的目录。

34.15. 口令文件

一个用户主目录中的.pgpass文件能够包含在连接需要时使用的口令（并且其他情况不会指定口令）。在微软的 Windows 上该文件被命名为%APPDATA%\postgresql\pgpass.conf（其中%APPDATA%指的是用户配置中的应用数据子目录）。另外，可以使用连接参数passfile或者环境变量PGPASSFILE指定一个口令文件。

这个文件应该包含下列格式的行：

```
hostname:port:database:username:password
```

（你可以向该文件增加一个提醒：把上面的行复制到该文件并且在前面加上#）。前四个域的每一个都可以是文字值或者匹配任何东西的*。第一个匹配当前连接参数的行中的口令域

将被使用（因此，在使用通配符时把更特殊的项放在前面）。如果一个条目需要包含:或者\，用\对该字符转义。如果指定了host连接参数，主机名字段会被匹配到host，否则如果指定了hostaddr参数则匹配到hostaddr，如果两者都没有给出，则会搜索主机名localhost。当连接是一个Unix域套接字连接并且host参数匹配libpq的默认套接字目录路径时，也会搜索主机名localhost。在一台后备服务器上，值为replication的数据库字段匹配连接到主服务器的里复制连接。否则数据库字段的用途有限，因为用户对同一个集簇中的所有数据库都有相同的口令。

在 Unix 系统上，口令文件上的权限必须不允许所有人或组内访问，可以用chmod 0600 ~/.pgpass这样的命令实现。如果权限没有这么严格，该文件将被忽略。在微软 Windows 上，该文件被假定存储在一个安全的目录中，因此不会进行特别的权限检查。

34.16. 连接服务文件

连接服务文件允许 libpq 连接参数与一个单一服务名称关联。那个服务名称可以被一个 libpq 连接指定，与其相关的设置将被使用。这允许在不重新编译 libpq 应用的前提下修改连接参数。服务名称也可以被使用PGSERVICE环境变量来指定。

连接服务文件可以是每个用户都有一个的服务文件，它位于~/.pg_service.conf或者环境变量PGSERVICEFILE指定的位置。它也可以是一个系统范围的文件，位于`pg_config --sysconfdir`/pg_service.conf的或者环境变量PGSYSCONFDIR指定的目录。如果相同名称的服务定义存在于用户和系统文件中，用户文件将优先考虑。

该文件使用一种“INI 文件”格式，其中小节名是服务名并且参数是连接参数。列表见第 34.1.2 节例如：

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

share/pg_service.conf.sample中提供了一个例子文件。

34.17. 连接参数的 LDAP 查找

如果libpq已经在编译时打开了 LDAP 支持（configure的选项--with-ldap），就可以通过 LDAP 从一个中央服务器检索host或dbname之类的连接参数。这样做的好处是如果一个数据库的连接参数改变，不需要在所有的客户端机器上更新连接信息。

LDAP 连接参数查找使用连接服务文件pg_service.conf（见第 34.16 节。pg_service.conf中一个以ldap://开始的行将被识别为一个 LDAP URL 并且将执行一个 LDAP 查询。结果必须是一个keyword = value对列表，它将被用来设置连接选项。URL 必须遵循 RFC 1959 并且是形式

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

其中hostname默认为localhost并且port默认为 389。

一次成功的 LDAP 查找后，pg_service.conf的处理被终止。但是如果联系不上 LDAP 则会继续处理pg_service.conf。这就提供了后手，可以加入更多指向不同 LDAP 服务器的 LDAP URL 行、经典的keyword = value对或者默认连接选项。如果你宁愿在这种情况下得到一个错误消息，在该 LDAP URL 之后增加一个语法错误的行。

一个和 LDIF 文件一起创建的 LDAP 条目实例

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
```

```
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn:mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

可以用下面的 LDAP URL 查询：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)
```

你也可以将常规的服务文件条目和 LDAP 查找混合。pg_service.conf中一节的完整例子：

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

可以用下面的 LDAP URL 查询到：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

34.18. SSL 支持

PostgreSQL本地支持使用SSL连接加密客户端/服务器通信以提高安全性。关于服务器端的SSL功能详见第 18.9 节

libpq读取系统范围的OpenSSL配置文件。默认情况下，这个文件被命名为openssl.cnf并且位于openssl version -d所报告的目录中。可以通过设置环境变量OPENSSL_CONF把这个默认值覆盖为想要的配置文件的名称。

34.18.1. 服务器证书的客户端验证

默认情况下，PostgreSQL将不会执行服务器证书的任何验证。这意味着可以在不被客户端知晓的情况下伪造服务器身份（例如通过修改一个 DNS 记录或者接管服务器的 IP 地址）。为了阻止哄骗，客户端必须能够通过一条信任链验证服务器的身份。信任链可以这样建立：在一台计算机上放置一个根（自签名的）证书机构（CA）的证书并且在另一台计算机上放置一个由根证书签发的叶子证书。还可以使用一种“中间”证书，它由根证书签发并且可以签发叶子证书。

为了允许客户端验证服务器的身份，在客户端上放置一份根证书并且在服务器上放置由根证书签发的叶子证书。为了允许服务器验证客户端的身份，在服务器上放置一份根证书并且在客户端上放置由根证书签发的叶子证书。也可以使用一个或者更多个中间证书（通常与叶子证书存在一起）来将叶子证书链接到根证书。

一旦信任链被建立起来，客户端有两种方法验证服务器发过来的叶子证书。如果参数sslmode被设置为verify-ca，libpq将通过检查该证书是否链接到存储在客户端上的根证

书来验证服务器。如果sslmode被设置为verify-full，libpq还将验证服务器的主机名匹配存储在服务器证书中的名称。如果服务器证书无法被验证，则SSL连接将失败。在大部分对安全性很敏感的环境中，推荐使用verify-full。

在verify-full模式中，主机名被拿来与证书的主体别名属性 匹配，或者在不存在类型dNSName的主体别名时与通用名称属性匹配。 如果证书的名称属性以一个星号（*）开始，这个星号将被 视作一个通配符，它将匹配所有除了句点（.） 之外的字符。这意味着该证书将不会匹配子域。如果连接是使用一个 IP 地址而不是一个主机名创建的，该 IP 地址将被匹配（不做任何 DNS 查找）。

要允许服务器证书验证，必须将一个或者更多个根证书放置在用户主目录下的~/.postgresql/root.crt文件中（在Microsoft Windows上该文件名为%APPDATA%\postgresql\root.crt）。如果需要把服务器发来的证书链链接到存储在客户端的根证书，还应该将中间证书加到该文件中。

如果文件~/.postgresql/root.crl存在（微软 Windows 上的%APPDATA%\postgresql\root.crl），证书撤销列表（CRL）项也会被检查。

根证书文件和 CRL 的位置可以通过设置连接参数sslrootcert和sslcrl或环境变量PGSSLROOTCERT和PGSSLCRL改变。

> **注意**
>
> 为了与 PostgreSQL 的早期版本达到向后兼容，如果存在一个根 CA 文件，sslmode=require的行为将与verify-ca相同，即服务器证书根据 CA 验证。我们鼓励依赖这种行为，并且需要证书验证的应用应该总是使用verify-ca或者verify-full。

34.18.2. 客户端证书

如果服务器尝试通过请求客户端的叶子证书来验证客户端的身份，libpq将发送用户主目录下文件~/.postgresql/postgresql.crt中存储的证书。该证书必须链接到该服务器信任的根证书。也必须存在一个匹配的私钥文件~/.postgresql/postgresql.key。该私钥文件不能允许全部用户或者组用户的任何访问，可以通过命令chmod 0600 ~/.postgresql/postgresql.key实现。在微软 Windows 上这些文件被命名为%APPDATA%\postgresql\postgresql.crt和%APPDATA%\postgresql\postgresql.key，不会有特别的权限检查，因为该目录已经被假定为安全。证书和密钥文件的位置可以使用连接参数sslcert和sslkey或者环境变量PGSSLCERT和PGSSLKEY覆盖。

postgresql.crt中的第一个证书必须是客户端的证书，因为它必须匹配客户端的私钥。可以选择将“中间”证书追加到该文件 — 这样做避免了在服务器上存放中间证书的要求（ssl_ca_file）。

创建证书的指令请参考第 18.9.5 节

34.18.3. 不同模式中提供的保护

sslmode参数的不同值提供了不同级别的保护。SSL 能够针对三类攻击提供保护：

窃听

如果一个第三方能够检查客户端和服务器之间的网络流量，它能读取连接信息（包括用户名和口令）以及被传递的数据。SSL使用加密来阻止这种攻击。

中间人（MITM）

如果一个第三方能对客户端和服务器之间传送的数据进行修改，它就能假装是服务器并且因此能看见并且修改数据，即使这些数据已被加密。然后第三方可以将连接信息和数据转送给原来的服务器，使得它不可能检测到攻击。这样做的通常途径包括 DNS 污染和

地址劫持，借此客户端被重定向到一个不同的服务器。还有几种其他的攻击方式能够完成这种攻击。SSL使用证书验证让客户端认证服务器，就可以阻止这种攻击。

模仿

如果一个第三方能假装是一个授权的客户端，它能够简单地访问它本不能访问的数据。通常这可以由不安全的口令管理所致。SSL使用客户端证书来确保只有持有合法证书的客户端才能访问服务器，这样就能阻止这种攻击。

对于一个已知安全的连接，在连接被建立之前，SSL 使用必须被配置在客户端和服务器之上。如果只在服务器上配置，客户端在知道服务器要求高安全性之前可能会结束发送敏感信息（例如口令）。在 libpq 中，要确保连接安全，可以设置sslmode参数为verify-full或verify-ca并且为系统提供一个根证书用来验证。这类似于使用一个https URL进行加密网页浏览。

一旦服务器已经被认证，客户端可以传递敏感数据。这意味着直到这一点，客户端都不需要知道是否证书将被用于认证，这样只需要在服务器配置中指定就比较安全。

所有SSL选项都带来了加密和密钥交换的负荷，因此必须在性能和安全性之间做出平衡。表 34.不同sslmode值所保护的风险，以及它们是怎样看待安全性和负荷的。

表 34.1. SSL 模式描述

sslmode	窃听保护	MITM保护	声明
disable	No	No	我不关心安全性，并且我不想为加密增加负荷。
allow	可能	No	我不关心安全性，但如果服务器坚持，我将承担加密带来的负荷。
prefer	可能	No	我不关心安全性，但如果服务器支持，我希望承担加密带来的负荷。
require	Yes	No	我想要对数据加密，并且我接受因此带来的负荷。我信任该网络会保证我总是连接到想要连接的服务器。
verify-ca	Yes	取决于 CA-策略	我想要对数据加密，并且我接受因此带来的负荷。我想要确保我连接到的是我信任的服务器。
verify-full	Yes	Yes	我想要对数据加密，并且我接受因此带来的负荷。我想要确保我连接到的是我信任的服务器，并且就是我指定的那一个。

verify-ca和verify-full之间的区别取决于根CA的策略。如果使用了一个公共CA，verify-ca允许连接到那些可能已经被其他人注册到该CA的服务器。在这种情况下，总是应该使用verify-full。如果使用了一个本地CA或者甚至是一个自签名的证书，使用verify-ca常常就可以提供足够的保护。

sslmode的默认值是prefer。如表中所示，这在安全性的角度来说没有意义，并且它只承诺可能的性能负荷。提供它作为默认值只是为了向后兼容，并且我们不推荐在安全部署中使用它。

34.18.4. SSL 客户端文件使用

表 34.2总结了与客户端 SSL 设置相关的文件。

表 34.2. Libpq/客户端 SSL 文件用法

文件	内容	效果
~/.postgresql/postgresql.crt	客户端证书	由服务器要求
~/.postgresql/postgresql.key	客户端私钥	证明客户端证书是由拥有者发送；不代表证书拥有者可信
~/.postgresql/root.crt	可信的证书机构	检查服务器证书是由一个可信的证书机构签发
~/.postgresql/root.crl	被证书机构撤销的证书	服务器证书不能在这个列表上

34.18.5. SSL 库初始化

如果你的应用初始化libssl或libcrypto库以及带有SSL支持的libpq，你应该调用PQinitOpenSSL来告诉libpq：libssl或libcrypto库已经被你的应用初始化，这样libpq将不会也去初始化那些库。 关于 SSL API 详见http://h71000.www7.hp.com/doc/83final/ba554_90007/ch04.html。

PQinitOpenSSL

允许应用选择要初始化哪个安全性库。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

当do_ssl是非零时，libpq将在第一次打开数据库连接前初始化OpenSSL库。当do_crypto是非零时，libcrypto库将被初始化。默认情况下（如果没有调用PQinitOpenSSL），两个库都会被初始化。当 SSL 支持没有被编译时，这个函数也存在但是什么也不做。

如果你的应用使用并且初始化OpenSSL或者它的底层libcrypto库，你必须在第一次打开数据库连接前以合适的非零参数调用这个函数。同时要确保在打开一个数据库连接前已经完成了初始化。

PQinitSSL

允许应用选择要初始化哪个安全性库。

```
void PQinitSSL(int do_ssl);
```

这个函数等效于PQinitOpenSSL(do_ssl, do_ssl)。这对于要么初始化OpenSSL以及libcrypto要么都不初始化的应用足够用了。

PQinitSSL从PostgreSQL 8.0 就存在了，而PQinitOpenSSL直到PostgreSQL 8.4 才被加入，因此PQinitSSL可能对那些需要与旧版本libpq一起工作的应用来说更合适。

34.19. 在线程化程序中的行为

libpq默认是可再入的并且是线程安全的。你可能需要使用特殊的编译器命令行选项来编译你的应用代码。参考你的系统文档来了解如何编译启用线程的应用，或者在src/Makefile.global中查找PTHREAD_CFLAGS和PTHREAD_LIBS。这个函数允许查询libpq的线程安全状态：

PQisthreadsafe

返回libpq库的线程安全状态。

```
int PQisthreadsafe();
```

如果libpq是线程安全的则返回 1，否则返回 0。

一个线程限制是不允许两个线程同时尝试操纵同一个PGconn对象。特别是你不能从不同的线程通过同一个连接对象发出并发的命令（如果你需要运行并发命令，请使用多个连接）。

PGresult对象在创建后通常是只读的，并且因此可以在线程之间自由地被传递。但是，如果你使用任何第 34.11 或第 34.13 节描述的PGresult修改函数，你需要负责避免在同一个PGresult上的并发操作。

被废弃的函数PQrequestCancel以及PQoidStatus不时线程安全的并且不应当在多线程程序中使用。PQrequestCancel可以被替换为PQcancel。PQoidStatus可以被替换为PQoidValue。

如果你在应用中使用 Kerberos（除了在libpq中之外），你将需要对 Kerberos 调用加锁，因为 Kerberos 函数不是线程安全的。参考libpq源代码中的PQregisterThreadLock函数，那里有在libpq和应用之间做合作锁定的方法。

如果你在线程化应用中碰到问题，将该程序运行在src/tools/thread来查看是否你的平台有线程不安全的函数。这个程序会被configure运行，但是对于二进制发布，你的库可能不匹配用来编译二进制的库。

34.20. 编译 libpq 程序

要编译（即编译并且链接）一个使用libpq的程序，你需要做下列所有的事情：

- 包括libpq-fe.h头文件：

  ```
  #include <libpq-fe.h>
  ```

  如果你无法这样做，那么你通常会从你的编译器得到像这样的错误消息：

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- 通过为你的编译器提供-Idirectory选项，向你的编译器指出PostgreSQL头文件安装在哪里（在某些情况下编译器默认将查看该目录，因此你可以忽略这个选项）。例如你的编译命令行可能看起来像：

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  如果你在使用 makefile，那么把该选项加到CPPFLAGS变量中：

```
CPPFLAGS += -I/usr/local/pgsql/include
```

如果你的程序可能由其他用户编译，那么你不应该像那样硬编码目录位置。你可以运行工具pg_config在本地系统上找出头文件在哪里：

```
$ pg_config --includedir
/usr/local/include
```

如果你安装了pkg-config，你可以运行：

```
$ pkg-config --cflags libpq
-I/usr/local/include
```

注意这将在路径前面包括-I。

无法为编译器指定正确的选项将导致一个错误消息，例如：

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- 当链接最终的程序时，指定选项-lpq，这样libpq库会被编译进去，也可以用选项-Ldirectory向编译器指出libpq库所在的位置（再次，编译器将默认搜索某些目录）。为了最大的可移植性，将-L选项放在-lpq选项前面。例如：

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
```

你也可以使用pg_config找出库目录：

```
$ pg_config --libdir
/usr/local/pgsql/lib
```

或者再次使用pkg-config：

```
$ pkg-config --libs libpq
-L/usr/local/pgsql/lib -lpq
```

再次提示这会打印出全部的选项，而不仅仅是路径。

指出这一部分问题的错误消息可能看起来像：

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

This means you forgot -lpq.

```
/usr/bin/ld: cannot find -lpq
```

这意味着你忘记了-L选项或者没有指定正确的目录。

34.21. 例子程序

这些例子和其他例子可以在源代码发布的src/test/examples目录中找到。

例 34.1. libpq 例子程序 1

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      测试 libpq（PostgreSQL 前端库） 的 C 版本。
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立到数据库的一个连接 */
    conn = PQconnectdb(conninfo);

    /* 检查看后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
```

```
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 任何时候不再需要 PGresult 时，应该 PQclear 它来避免内存泄露
     */
    PQclear(res);

    /*
     * 我们的测试案例这里涉及使用一个游标，对它我们必须用在一个事务块内。
     * 我们可以在一个单一的 "select * from pg_database" 的 PQexec() 中做整个事
情，
     * 但是作为一个好的例子它太琐碎。
     */

    /* 开始一个事务块 */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    PQclear(res);

    /*
     * 从 pg_database 取得行，它是数据库的系统目录
     */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* 首先，打印出属性名 */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* 接下来，打印出行 */
    for (i = 0; i < PQntuples(res); i++)
    {
```

```
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* 关闭入口，我们不需要考虑检查错误 */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* 结束事务 */
    res = PQexec(conn, "END");
    PQclear(res);

    /* 关闭到数据库的连接并且清理 */
    PQfinish(conn);

    return 0;
}
```

例 34.2. libpq例子程序 2

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      测试异步通知接口
 *
 * 开始这个程序，然后在另一个窗口的 psql 中做
 *   NOTIFY TBL2;
 * 重复四次来让这个程序退出。
 *
 * 或者，如果你想要得到奇妙的事情，尝试：
 * 用下列命令填充一个数据库
 * （在 src/test/examples/testlibpq2.sql 中提供）
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * 开始这个程序，然后从psql做下面的操作四次：
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include "libpq-fe.h"
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * 用过用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立一个到数据库的连接 */
    conn = PQconnectdb(conninfo);

    /* 检查后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 任何时候不再需要 PGresult 时，应该 PQclear 它来避免内存泄露
     */
```

```
    PQclear(res);

    /*
     * 发出 LISTEN 命令启用来自规则的 NOTIFY 的通知。
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    PQclear(res);

    /* 在接收到四个通知后退出。 */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*
         * 休眠到在连接上发生某些事情。我们使用 select(2) 来等待输入，但是你也可
以使用 poll() 或相似的设施。
         */
        int         sock;
        fd_set      input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break;              /* 不应该发生 */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* 现在检查输入 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");

    /* 关闭到数据库的连接并且清理 */
    PQfinish(conn);
```

```
    return 0;
}
```

例 34.3. libpq例子程序 3

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      测试线外参数和二进制 I/O。
 *
 * 在运行之前，使用下列命令填充一个数据库（在 src/test/examples/testlibpq3.sql
中提供）
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * 期待的输出是：
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */
#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
/*
 * 这个函数打印一个查询结果，该结果以二进制格式从上面的注释中定义的表中取得。
 * 我们把它分离出来是因为 main() 函数需要使用它两次。
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* 使用 PQfnumber 来避免假定结果中域的顺序 */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* 得到域值（我们忽略它们为空值的可能性！） */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * INT4 的二进制表示是按照网络字节序的，我们最好强制为本地字节序。
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * TEXT 的二进制表示是文本，并且因为 libpq 会为它追加一个零字节，它将工
作得和 C 字符串一样好。
         *
         * BYTEA 的二进制表示是一堆字节，其中可能包含嵌入的空值，因此我们必须注
意域长度。
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}
```

```
int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立一个到数据库的连接 */
    conn = PQconnectdb(conninfo);

    /* 检查看后端连接是否成功被建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn, "SET search_path = testlibpq3");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /*
     * 这个程序的要点在于用线外参数展示 PQexecParams() 的使用，以及数据的二进制
传输。
     *
     * 第一个例子将参数作为文本传输，但是以二进制格式接收结果。
     * 通过使用线外参数，我们能够避免使用繁杂的引用和转义，即便数据是文本。
     * 注意我们怎么才能对参数值中的引号不做任何事情。
     */

    /* 这里是我们的线外参数值 */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1,       /* 一个参数 */
                       NULL,    /* 让后端推导参数类型 */
```

```
                        paramValues,
                        NULL,    /* 因为文本不需要参数长度 */
                        NULL,    /* 对所有文本参数的默认值 */
                        1);      /* 要求二进制结果 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /*
     * 在第二个例子中，我们以二进制形式传输一个整数参数，并且再次以二进制形式接收结果。
     *
     * 尽管我们告诉 PQexecParams 我们让后端推导参数类型，我们实际上通过在查询文本中造型参数符号来强制该决定。
     * 在发送二进制参数时，这是一种好的安全测度。
     */

    /* 将整数值 "2" 转换为网络字节序 */
    binaryIntVal = htonl((uint32_t) 2);

    /* 为 PQexecParams 设置参数数组 */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* binary */

    res = PQexecParams(conn,
                        "SELECT * FROM test1 WHERE i = $1::int4",
                        1,       /* 一个参数 */
                        NULL,    /* 让后端推导参数类型 */
                        paramValues,
                        paramLengths,
                        paramFormats,
                        1);      /* 要求二进制结果 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* 关闭到数据库的连接并清理 */
    PQfinish(conn);

    return 0;
}
```

第 35 章 大对象

PostgreSQL具有大对象的功能，它提供了对于存储在一个特殊大对象结构中的用户数据的流式访问。对于那些大得无法以一个整体处理的数据值 ，流式访问非常有用。

本章介绍了PostgreSQL中大对象数据的实现、编程和查询语言接口。在本章中我们使用名为libpq的C库来作为例子，但是绝大部分PostgreSQL支持的本地编程接口也具有相同的功能。其他接口可能会在内部使用大对象接口来提供对大值的一般支持，但这里就不再描述。

35.1. 简介

所有的大对象都存在一个名为pg_largeobject的系统表中。每一个大对象还在系统表pg_largeobject_metadata中有一个对应的项。大对象可以通过类似于标准文件操作的读/写API来进行创建、修改和删除。

PostgreSQL也支持一种称为“TOAST”的存储系统， 它自动把大于一个数据库页的值存储到一个二级存储区域，每一个表都有专属的二级存储区域。这使得大对象功能显得有些陈旧。但是大对象功能仍然有一个优势是能够支持高达4TB的值，而TOAST域只能支持最大1GB。此外，读写一个大对象的片段会很高效，但是大部分在TOAST域上的操作都将把整个值作为一个单元进行读或写。

35.2. 实现特性

大对象的实现将大对象分解成很多“数据块”并且将这些数据块存储在数据库的行中。一个B-tree索引用来保证在进行随机访问读写时能够根据数据块号快速地搜索到正确的数据块。

为一个大对象存储的数据块并不需要是连续的。例如，如果一个应用打开了一个新的大对象，移动到偏移量1000000并写了一些字节，这并不会导致分配1000000字节的存储，只有覆盖写入字节范围的数据块需要被分配。而一个读操作将会把现有最后的数据块之前还未分配的位置读出为0。这和Unix文件系统中“稀疏”文件的一般行为相对应。

自PostgreSQL 9.0起，大对象可以有一个拥有者和一组访问权限，它们可以用GRANT和REVOKE管理。读一个大对象需要SELECT权限，而写或者截断一个大对象则需要UPDATE权限。只有大对象的拥有者（或者一个数据库超级用户）可以创建大对象、注释大对象或修改大对象的拥有者。要调整这些行为以兼容以前的发行，请见lo_compat_privileges的运行时参数。

35.3. 客户端接口

本节描述PostgreSQL的libpq客户端接口为访问大对象所提供的功能。PostgreSQL的大对象接口按照Unix文件系统的接口建模，也有相似的open、read、write、lseek等。

所有使用这些函数对大对象的操作都必须发生在一个SQL事务块中，因为大对象文件描述符只在事务期间有效。

在执行任何一个这种函数期间如果发生一个错误，该函数将会返回一个其他的不可能值，典型的是0或-1。一个关于该错误的消息亦会被保存在连接对象中，可以通过PQerrorMessage检索到。

使用这些函数的客户端应用应该包括头文件libpq/libpq-fs.h并链接libpq库。

35.3.1. 创建一个大对象

函数

```
Oid lo_creat(PGconn *conn, int mode);
```

创建一个新的大对象。其返回值是分配给这个新大对象的OID或者InvalidOid（0）表示失败。 mode自PostgreSQL 8.1就不再使用且会被忽略。但是，为了和以前的发行兼容，它最好被设置为INV_READ、INV_WRITE或INV_READ | INV_WRITE（这些符号常量定义在头文件libpq/libpq-fs.h中）。

一个例子：

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

函数

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

也创建一个新的大对象。分配给该大对象的OID可以通过lobjId指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果lobjId是InvalidOid（0），则lo_create会分配一个未使用的OID（这时和lo_creat的行为相同）。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。

lo_create在从PostgreSQL 8.1开始的版本中是新的，如果该函数在旧服务器版本上运行，它将失败并返回InvalidOid。

一个例子：

```
inv_oid = lo_create(conn, desired_oid);
```

35.3.2. 导入一个大对象

要将一个操作系统文件导入成一个大对象，调用：

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename指定了要导入为大对象的操作系统文件名。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。注意该文件是被客户端接口库而不是服务器所读取，因此它必须存在于客户端文件系统中并且对于客户端应用是可读的。

函数

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

也可以导入一个新大对象。分配给新大对象的OID可以用lobjId指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果lobjId是InvalidOid（0），则lo_import_with_oid会分配一个未使用的OID（这和lo_import的行为相同）。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。

lo_import_with_oid在从PostgreSQL 8.1开始的版本中是新的并且在内部使用了lo_create（在8.1中也是新的），如果该函数在旧服务器版本上运行，它将失败并返回InvalidOid。

35.3.3. 导出一个大对象

要把一个大对象导出到一个操作系统文件，调用：

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

lobjId参数指定要导出的大对象的OID，filename参数指定操作系统文件名。注意该文件是被客户端接口库而不是服务器写入。成功返回1，错误返回-1。

35.3.4. 打开一个现有的大对象

要打开一个现有的大对象进行读写，调用：

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

lobjId参数指定要打开的大对象的OID。mode位控制着打开对象是为了只读（INV_READ）、只写（INV_WRITE）或者读写（这些符号常量定义在头文件libpq/libpq-fs.h中）。lo_open返回一个（非负）大对象描述符以便后面用于lo_read、lo_write、lo_lseek、lo_lseek64、lo_tell、lo_tell64、lo_truncate、lo_truncate64以及lo_close。该描述符只在当前事务期间有效。如果打开错误将会返回-1。

服务器目前并不区分模式INV_WRITE和INV_READ |INV_WRITE：在两种情况中都允许从描述符读取。但是在这些模式和单独的INV_READ之间有明显的区别：使用INV_READ我们不能向描述符写入，从中读取的数据则反映了该大对象在活动事务快照时刻的内容（该快照在lo_open被执行时创建），而不管之后被该事务或其他事务写入的内容。从一个以INV_WRITE模式打开的描述符读取的数据所有其他已提交事务以及当前事务所作的写入。这与普通SQL命令 SELECT的REPEATABLE READ和READ COMMITTED事务模式之间的区别相似。

如果大对象的SELECT特权不可用，或者如果在指定了INV_WRITE时UPDATE特权不可用，则lo_open将会失败（在PostgreSQL 11之前，这些特权的检查是在使用该描述符的第一次实际读取或写入时进行）。这些特权检查可以用lo_compat_privileges运行时参数禁用。

一个例子：

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

35.3.5. 向一个大对象写入数据

函数

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

从buf（大小必须是 len）中写出len字节到大对象描述符fd。参数fd必须是已经由前面的lo_open返回的大对象描述符。函数将返回实际写入的字节数（在当前的实现中，除非出错，返回的字节数总是等于len）。在出错时，返回值为-1。

尽管参数len被声明为类型size_t，该函数会拒绝超过INT_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

35.3.6. 从一个大对象读取数据

函数

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

从大对象描述符fd中读取最多len字节到buf （大小必须是len）中。参数fd必须是已经由前面的lo_open返回的大对象描述符。实际读出的字节数将被返回，如果先到达了大对象的末尾返回值可能会小于len。出错时返回值为-1。

尽管参数len被声明为类型size_t，该函数会拒绝超过INT_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

35.3.7. 在一个大对象中查找

要改变一个大对象描述符的当前读或写位置，调用：

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

该函数将大对象文件描述符fd的当前位置指针移动到由offset指定的新位置。whence的可用值是SEEK_SET（从对象开头定位）、SEEK_CUR （从当前位置定位）以及SEEK_END（从对象末尾定位）。返回值是新位置的指针，或者是-1表示出错。

在处理可能超过2GB大小的大对象时，换用

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

该函数的行为和lo_lseek相同，但是它能接受一个超过2GB的offset并/或传送一个超过2GB的结果。注意如果新位置的指针超过2GB，lo_lseek会失败。

lo_lseek64是从 PostgreSQL 9.3开始增加的新函数。如果该函数在一个旧服务器版本上执行，将会失败并返回-1。

35.3.8. 获取一个大对象的查找位置

要得到一个大对象描述符的当前读或写位置，调用：

```
int lo_tell(PGconn *conn, int fd);
```

如果出现错误，返回值是-1。

在处理可能超过2GB大小的大对象时，换用：

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

该函数和lo_tell的行为相同，但是它能传递超过2GB的结果。注意如果当前读/写位置超过2GB，lo_tell将会失败。

lo_tell64是从PostgreSQL 9.3开始新增的函数。如果该函数在旧服务器版本上运行，将会失败并返回-1。

35.3.9. 截断一个大对象

要将一个大对象截断成一个给定长度，调用：

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

该函数将大对象描述符fd截断为长度len。参数fd必须是已经由前面的lo_open返回的大对象描述符。如果len超过了大对象的当前长度，大对象将会被使用空字节（'\0'）扩展到指定长度。成功时lo_truncate返回0，失败时返回值为-1。

描述fd的读/写位置不变。

尽管参数len被声明为类型size_t，lo_truncate会拒绝超过INT_MAX的长度值。

在处理可能超过2GB大小的大对象时，换用：

```
int lo_truncate64(PGcon *conn, int fd, pg_int64 len);
```

该函数和lo_truncate的行为相同，但它能够接受超过2GB的len值。

lo_truncate是从PostgreSQL 8.3开始新的函数，如果该函数运行在一个旧服务器版本上，它将失败并返回-1。

lo_truncate64是从PostgreSQL 9.3开始新的函数，如果该函数运行在一个旧服务器版本上，它将失败并返回-1。

35.3.10. 关闭一个大对象描述符

要关闭一个大对象描述符，调用：

```
int lo_close(PGconn *conn, int fd);
```

其中fd是由lo_open返回的大对象描述符。成功时，lo_close返回0，失败时返回-1。

在事务末尾仍然保持打开的任何大对象描述符都会自动被关闭。

35.3.11. 移除一个大对象

要从数据库中移除一个大对象，调用：

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

lobjId参数指定要移除的大对象的OID。成功时返回1，失败时返回-1。

35.4. 服务器端函数

表 35.1中列出了为从 SQL 操纵大对象定制的服务器端函数。

表 35.1. 面向 SQL 的大对象函数

函数	返回类型	描述	实例	结果
lo_from_bytea(loid oid, string bytea)	oid	创建一个大对象并且在其中存储数据，返回它的OID。 传递0会让系统选择一个OID。	lo_from_bytea(0, '\xffffff00')	24528
lo_put(loid oid, offset bigint, str bytea)	void	在给定的偏移位置写入数据。	lo_put(24528, 1, '\xaa')	
lo_get(loid oid [, from bigint, for int])	bytea	在其中抽取内容或一个子串。	lo_get(24528, 0, 3)	\xffaaff

之前描述过的每个客户端函数都有一个相应的服务器端函数。实际上， 多半客户端函数都是等效的服务器端函数的简单接口。这些可以从 SQL 命令方便调用的函数是： lo_creat、 lo_create、 lo_unlink、 lo_import以及 lo_export。 下面是使用它们的例子：

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);        -- 返回新的空大对象的OID

SELECT lo_create(43213);    -- 尝试创建OID为43213的大对象
```

```
SELECT lo_unlink(173454);  -- 删除OID为173454的大对象

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- 和上面相同，但是指定了使用的OID
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

服务器端的lo_import和lo_export函数具有和它们的客户端同类大不相同的行为。这两个函数从服务器的文件系统中读和写文件，使用的是数据库所有者的权限。因此，默认情况下它们的使用被限制于超级用户。相反，客户端的导入和导出函数读写的是客户端的文件系统，使用的是客户端程序的权限。除了读取或写入所请求的大对象的特权之外，客户端函数不要求任何数据库特权。

> **小心**
>
> 可以把服务器端的lo_import和lo_export函数GRANT给非超级用户，但需要仔细地考虑安全因素。有这类特权的恶意用户可以很容易地利用它们成为超级用户（例如通过重写服务器配置文件），或者攻击该服务器文件系统的其他部分而无需获得数据库超级用户特权。因此对具有这类特权的角色访问必须受到和超级用户角色一样的仔细保护。尽管如此，如果某些例行任务需要使用服务器端的lo_import或者lo_export，使用具有这类特权的角色比使用具有完整超级用户特权的角色更加安全，因为那样会减小意外错误造成的损伤风险。

函数lo_read和 lo_write的功能也可以在服务器端调用，但是在服务器端的名称与客户端接口不同：它们的名称中不包含下划线。我们必须以loread和lowrite调用这些函数。

35.5. 例子程序

例 35.是一个展示libpq中大对象接口如何使用的例子程序。部分程序被注释但仍保留在代码中，用户可以利用之。该程序可以在源代码的src/test/examples/testlo.c中找到。

例 35.1. 用libpq操作大对象的例子程序

```
/*-------------------------------------------------------------------------
 *
 * testlo.c
 *    测试通过 libpq 使用大对象
 *
 * Portions Copyright (c) 1996-2018, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    把文件 "in_filename" 作为一个大对象 "lobjOid" 载入到数据库
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * 打开要读入的文件
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }

    /*
     * 创建大对象
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * 从该 Unix 文件读取并写入到大对象
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
```

```
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
```

```
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile -
 *    把大对象 "lobjOid" 导出成文件 "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * 打开大对象
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * 打开要写入的文件
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* 错误 */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * 从大对象读入并写出到 Unix 文件
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
```

```
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * 设置连接
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* 检查看看后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制权。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
```

```
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

第 36 章 ECPG - C 中的嵌入式 SQL

这一章描述了用于PostgreSQL的嵌入式SQL包。它由 Linus Tolke (<linus@epact.se>) 和 Michael Meskes (<meskes@postgresql.org>) 编写。最初它是为了与C一起工作而编写的。它也能与C++配合，但是它还不识别所有的C++结构。

这份文档还远没有完成。但是因为这个结构是标准化的，额外的信息可以在有关 SQL 的很多资源中找到。

36.1. 概念

一个嵌入式 SQL 程序由一种普通编程语言编写的代码（在这里是 C）和位于特殊标记的小节中的 SQL 命令混合组成。要构建该程序，源代码（*.pgc）首先会通过嵌入式 SQL 预处理器，它会将源代码转换成一个普通 C 程序（*.c），并且后来它能够被一个 C 编译器所处理（编译和链接详见第 36.10 节。转换过的 ECPG 应用会通过嵌入式 SQL 库（ecpglib）调用 libpq 库中的函数，并且与 PostgreSQL 服务器使用普通的前端/后端协议通信。

嵌入式SQL在为 C 代码处理SQL命令方面比起其他方法来具有优势。首先，它会搞定向你的C程序变量传递或者读取信息时的繁文缛节。其次，程序中的 SQL 代码在编译时就会被检查以保证语法正确性。第三，C 中的嵌入式SQL是在SQL标准中指定的并且受到很多其他SQL数据库系统的支持。PostgreSQL实现被设计为尽可能匹配这个标准，并且通常可以相对容易地把为其他 SQL 数据库编写的SQL程序移植到PostgreSQL。

正如已经支出的，为嵌入式SQL接口编写的程序是插入了用于执行数据库相关动作的特殊代码的普通的 C 程序。这种特殊代码总是具有这样的形式：

```
EXEC SQL ...;
```

这些语句在语法上取代了一个 C 语句。取决于特定的语句，它们可以出现在全局层面或者是一个函数中。嵌入式 SQL语句遵循普通SQL代码的大小写敏感性规则， 而不是 C 的大小写敏感性规则。它们也允许嵌套的 C 风格注释（SQL 标准的一部分）。 不过，程序的 C 部分遵循 C 的标准不接受嵌套注释。

下列小节解释了所有嵌入式 SQL 语句。

36.2. 管理数据库连接

这一节描述如何打开、关闭以及切换数据库连接。

36.2.1. 连接到数据库服务器

我们可以使用下列语句连接到一个数据库：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

target可以用下列方法指定：

- dbname[@hostname][:port]
- tcp:postgresql://hostname[:port][/dbname][?options]
- unix:postgresql://hostname[:port][/dbname][?options]
- 一个包含上述形式之一的 SQL 字符串

- 到一个包含上述形式之一（参见例子）的字符变量的引用
- DEFAULT

如果你用字面（也就是不通过一个变量引用）指定连接目标并且你没有引用该值，那么将会应用普通 SQL 的大小写不敏感性规则。在那种情况中，你也能够按照需要单独将个体参数放置在双引号中。实际上，使用一个（单引号引用）的字符串或一个变量引用出错的可能性更小。连接目标DEFAULT会以默认用户名发起一个到默认数据库的连接。在那种情况中不能指定单独的用户名或连接名。

也有不同的方法来指定用户名：

- username
- username/password
- username IDENTIFIED BY password
- username USING password

如上所述，参数username以及password可以是一个 SQL 标识符、一个 SQL 字符串或者一个对字符变量的引用。

connection-name被用来在一个程序中处理多个连接。如果一个程序只使用一个连接，它可以被忽略。最近被打开的连接将成为当前连接，当一个 SQL 语句要被执行时，将默认使用它（见这一章稍后的部分）。

如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，开始每个会话时应该从search_path中移除公共可写的方案。例如，把options=-csearch_path=加到options中或者在连接后发出EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);。这种考虑并非专门针对ECPG，它适用于每一种用来执行任意SQL命令的接口。

这里有一些CONNECT语句的例子：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER
 john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* 或者 EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最后一种形式利用被上文成为字符变量引用的变体。你将在后面的小节中看到当你把 C 变量前放上一个冒号时，它们是怎样被用于 SQL 语句的。

注意连接目标的格式没有在 SQL 标准中说明。因此如果你想要开发可移植的应用，你可能想要使用某种基于上述最后一个例子的方法来把连接目标字符串封装在某个地方。

36.2.2. 选择一个连接

嵌入式 SQL 程序中的 SQL 语句默认是在当前连接（也就是最近打开的那一个）上执行的。如果一个应用需要管理多个连接，那么有两种方法来处理这种需求。

第一个选项是显式地为每一个 SQL 语句选择一个连接，例如：

```
EXEC SQL AT connection-name SELECT ...;
```

如果应用需要以混合的顺序使用多个连接，这个选项特别合适。

如果你的应用使用多个线程执行，它们不能并发地共享一个连接。你必须显式地控制对连接的访问（使用互斥量）或者为每个线程使用一个连接。

第二个选项是执行一个语句来切换当前的连接。该语句是：

```
EXEC SQL SET CONNECTION connection-name;
```

如果很多语句要被在同一个连接上执行，这个选项特别方便。

这里有一个管理多个数据库连接的例子程序：

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    /* 这个查询将在最近打开的数据库 "testdb3" 中执行 */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* 使用 "AT" 在 "testdb2" 中运行一个查询 */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* 切换当前连接到 "testdb1" */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子将产生这样的输出：

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
```

```
current=testdb1 (should be testdb1)
```

36.2.3. 关闭一个连接

要关闭一个连接，使用下列语句：

```
EXEC SQL DISCONNECT [connection];
```

connection可以用下列方法指定：

- connection-name
- DEFAULT
- CURRENT
- ALL

如果没有指定连接名，当前连接将被关闭。

在一个应用中总是显式地从它打开的每一个连接断开是一种好的风格。

36.3. 运行 SQL 命令

任何 SQL 命令都可以在一个嵌入式 SQL 应用中被运行。下面是一些在嵌入式 SQL 应用中运行 SQL 命令的例子。

36.3.1. 执行 SQL 语句

创建一个表：

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

插入行：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除行：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

返回一个单一结果行的SELECT语句也可以直接使用EXEC SQL执行。要处理有多行的结果集，一个应用必须使用一个游标，可参考下面的第 36.3.2 节作为一种特殊情况，一个应用可以一次取出多行到一个数组主变量中，参考第 36.4.4.3.1 节。

单行选择：

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

还有，一个配置参数可以用SHOW命令检索：

```
EXEC SQL SHOW search_path INTO :var;
```

:something形式的记号是主变量，即它们指向 C 程序中的变量。它们在第 36.4 节解释。

36.3.2. 使用游标

要检索一个保持多行的结果集，一个应用必须声明一个游标并且从该游标中取得每一行。使用一个游标的步骤如下：声明一个游标、打开它、从该游标取得一行、重复并且最终关闭它。

使用游标选择：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

有关声明游标的更多细节，可参考DECLARE；FETCH命令的细节则可以参考FETCH。

> **注意**
>
> ECPG DECLARE命令实际上不会导致一个语句被发送到 PostgreSQL 后端。在OPEN命令被执行时，游标会在后端被打开（使用后端的DECLARE命令）。

36.3.3. 管理事务

在默认模式中，只有当EXEC SQL COMMIT被发出时才会提交命令。嵌入式 SQL 接口也通过ecpg的-t命令行选项（见ecpg）或者通过EXEC SQL SET AUTOCOMMIT TO ON语句支持事务的自动提交（类似于psql的默认行为）。在自动提交模式中，除非位于一个显式事务块内，每一个命令都会被自动提交。这种模式可以使用EXEC SQL SET AUTOCOMMIT TO OFF显式地关闭。

可以使用下列事务管理命令：

EXEC SQL COMMIT

提交一个进行中的事务。

EXEC SQL ROLLBACK

回滚一个进行中的事务。

EXEC SQL PREPARE TRANSACTION transaction_id

为两阶段提交准备当前事务。

EXEC SQL COMMIT PREPARED transaction_id

提交一个处于准备好状态的事务。

EXEC SQL ROLLBACK PREPARED transaction_id

回滚一个处于准备好状态的事务。

EXEC SQL SET AUTOCOMMIT TO ON

启用自动提交模式。

EXEC SQL SET AUTOCOMMIT TO OFF

禁用自动提交模式。这是默认值。

36.3.4. 预备语句

当传递给 SQL 语句的值在编译时未知或者同一个语句要被使用多次时，那么预备语句就有用武之地了。

语句使用命令PREPARE进行预备。对于还未知的值，使用占位符“?”：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid
 = ?";
```

如果一个语句返回一个单一行，应用可以在PREPARE之后调用EXECUTE来执行该语句，同时要用一个USING子句为占位符提供真实的值：

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

如果一个语句返回多行，应用可以使用一个基于该预备语句声明的游标。要绑定输入参数，该游标必须用一个USING子句打开：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* 当到达结果集末尾时，跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

当你不再需要该预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

更多有关PREPARE的细节，可参考PREPARE。关于使用占位符和输入参数的细节，可参考第 36.5 节

36.4. 使用主变量

在第 36.3 节，你了解了如何从一个嵌入式 SQL 程序执行 SQL 语句。某些那种语句只使用固定值并且没有提供方法来插入用户提供的值到语句中或者让程序处理查询返回的值。那种语句在实际应用中其实没有什么用处。这一节详细解释了如何使用一种简单的机制（主变量）在 C 程序和嵌入式 SQL 语句之间传递数据。在一个嵌入式 SQL 程序中，我们认为 SQL 语句是 C 程序代码中的客人，而 C 代码是主语言。因此 C 程序的变量被称为主变量。

另一种在 PostgreSQL 后端和 ECPG 应用之间交换值的方式是使用 SQL 描述符，它在第 36.7 节介绍。

36.4.1. 概述

在嵌入式 SQL 中进行 C 程序和 SQL 语句见的数据传递特别简单。我们不需要让程序把数据粘贴到语句（这会导致很多复杂性，例如正确地引用值），我们可以简单地在 SQL 语句中写 C 变量的名称，只要在它前面放上一个冒号。例如：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

这个语句引用了两个 C 变量（名为v1和v2）并且还使用了一个常规的 SQL 字符串来说明你没有被限制于使用某一种数据。

这种在 SQL 语句中插入 C 变量的风格可以用在 SQL 语句中每一个应该出现值表达式的地方。

36.4.2. 声明小节

要从程序传递数据给数据库（例如作为一个查询的参数）或者从数据库传数据回程序，用于包含这些数据的 C 变量必须在特别标记的节中被声明，这样嵌入式 SQL 预处理器才会注意它们。

这个节开始于：

```
EXEC SQL BEGIN DECLARE SECTION;
```

并且结束于：

```
EXEC SQL END DECLARE SECTION;
```

在这两行之间，必须是正常的 C 变量声明，例如：

```
int   x = 4;
char  foo[16], bar[16];
```

如你所见，你可以选择为变量赋一个初始值。变量的可见范围由定义它的节在程序中的位置决定。你也可以使用下面的语法声明变量，这种语法将会隐式地创建一个声明节：

```
EXEC SQL int i = 4;
```

你可以按照你的意愿在一个程序中放上多个声明节。

这些声明也会作为 C 变量被重复在输出文件中，因此无需再次声明它们。不准备在 SQL 命令中使用的变量可以正常地在这些特殊节之外声明。

一个结构或联合的定义也必须被列在一个DECLARE节中。否则预处理器无法处理这些类型，因为它不知道它们的定义。

36.4.3. 检索查询结果

现在你应该能够把程序产生的数据传递到一个 SQL 命令中了。但是怎么检索一个查询的结果呢？为此，嵌入式 SQL 提供了常规命令SELECT和FETCH的特殊变体。这些命令有一个特殊的INTO子句，它指定被检索到的值要被存储在哪些主变量中。SELECT被用于只返回单一行的查询，而FETCH被用于使用一个游标返回多行的查询。

这里是一个例子：

```
/*
 * 假定有这个表：
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

那么INTO子句出现在选择列表和FROM子句之间。选择列表中的元素数量必须和INTO后面列表（也被称为目标列表）的元素数量相等。

这里有一个使用命令FETCH的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

这里INTO子句出现在所有正常子句的后面。

36.4.4. 类型映射

当 ECPG 应用在 PostgreSQL 服务器和 C 应用之间交换值时（例如从服务器检索查询结果时或者用输入参数执行 SQL 语句时），值需要在 PostgreSQL 数据类型和主语言变量类型（具体来说是 C 语言数据类型）之间转换。ECPG 的要点之一就是它会在大多数情况下自动搞定这种转换。

在这方面有两类数据类型：一些简单 PostgreSQL 数据类型（例如integer和text）可以被应用直接读取和写入。其他 PostgreSQL 数据类型（例如timestamp和numeric）只能通过特殊库函数访问，见第 36.4.4.2 节

表 36.展示了哪种 PostgreSQL 数据类型对应于哪一种 C 数据类型。当你希望发送或接收一种给定 PostgreSQL 数据类型的值时，你应该在声明节中声明一个具有相应 C 数据类型的 C 变量。

表 36.1. 在 PostgreSQL 数据类型和 C 变量类型之间映射

PostgreSQL 数据类型	主变量类型
smallint	short
integer	int
bigint	long long int
decimal	decimal[a]
numeric	numeric[a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(n), varchar(n), text	char[n+1], VARCHAR[n+1][b]
name	char[NAMEDATALEN]
timestamp	timestamp[a]
interval	interval[a]
date	date[a]
boolean	bool[c]
bytea	char *

[a]这种类型只能通过特殊的库函数访问，见第 36.4.4.2 节

[b] 在ecpglib.h中声明

[c]如果不是本地化类型，则声明在ecpglib.h中

36.4.4.1. 处理字符串

要处理 SQL 字符串数据类型（例如varchar以及text），有两种可能的方式来声明主变量。

一种方式是使用char[]（一个char字符串），这是在 C 中处理字符数据最常见的方式。

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

注意你必须自己照看长度。如果你把这个主变量用作一个查询的目标变量并且该查询返回超过 49 个字符的字符串，那么将会发生缓冲区溢出。

另一种方式是使用VARCHAR类型，它是 ECPG 提供的一种特殊类型。在一个VARCHAR类型数组上的定义会被转变成一个命名的struct。这样一个声明：

```
VARCHAR var[180];
```

会被转变成：

```
struct varchar_var { int len; char arr[180]; } var;
```

成员arr容纳包含一个终止零字节的字符串。因此，要在一个VARCHAR主变量中存储一个字符串，该主变量必须被声明为具有包括零字节终止符的长度。成员len保存存储在arr中的字符串的长度，不包括终止零字节。当一个主变量被用做一个查询的输入时，如果strlen(arr)和len不同，将使用短的那一个。

VARCHAR可以被写成大写或小写形式，但是不能大小写混合。

char和VARCHAR主变量也可以保存其他 SQL 类型的值，它们将被存储为字符串形式。

36.4.4.2. 访问特殊数据类型

ECPG 包含一些特殊类型帮助你容易地与来自 PostgreSQL 服务器的一些特殊数据类型交互。特别地，它已经实现了对于numeric、decimal、date、timestamp以及interval类型的支持。这些数据类型无法有效地被映射到原始的主变量类型（例如int、long long int或者char[]），因为它们有一种复杂的内部结构。应用通过声明特殊类型的主变量以及使用 pgtypes 库中的函数来处理这些类型。pgtypes 库（在第 36.6 节详细描述）包含了处理这些类型的基本函数，这样你不需要仅仅为了给一个时间戳增加一个时段而发送一个查询给 SQL 服务器。

下面的小节描述了这些特殊数据类型。关于 pgtypes 库函数的更多细节，请参考第 36.6 节

36.4.4.2.1. timestamp, date

这里有一种在 ECPG 主应用中处理timestamp变量的模式。

首先，程序必须包括用于timestamp类型的头文件：

```
#include <pgtypes_timestamp.h>
```

接着，在声明节中声明一个主变量为类型timestamp：

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

并且在读入一个值到该主变量中之后，使用 pgtypes 库函数处理它。在下面的例子中，timestamp值被PGTYPEStimestamp_to_asc()函数转变成文本（ASCII）形式：

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

这个例子将展示像下面形式的一些结果：

```
ts = 2010-06-27 18:03:56.949343
```

另外，DATE 类型可以用相同的方式处理。程序必须包括pgtypes_date.h，声明一个主变量为日期类型并且将一个 DATE 值使用PGTYPESdate_to_asc()函数转变成一种文本形式。关于 pgtypes 库函数的更多细节，请参考第 36.6 节

36.4.4.2.2. interval

对interval类型的处理也类似于timestamp和date类型。不过，必须显式为一个interval类型分配内存。换句话说，该变量的内存空间必须在堆内存中分配，而不是在栈内存中分配。

这里是一个例子程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.2.3. numeric, decimal

numeric和decimal类型的处理类似于interval类型：需要定义一个指针、在堆上分配一些内存空间并且使用 pgtypes 库函数访问该变量。关于 pgtypes 库函数的更多细节，请参考第 36.6 节

pgtypes 库没有特别为decimal类型提供函数。一个应用必须使用一个 pgtypes 库函数把它转变成一个numeric变量以便进一步处理。

这里是一个处理numeric和decimal类型变量的例子程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;
```

```
    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

    /* 将一个decimal转变成numeric以显示一个decimal值。 */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.3. 非简单类型的主变量

你也可以把数组、typedefs、结构和指针用作主变量。

36.4.4.3.1. 数组

将数组用作主变量有两种情况。第一种如第 36.4.4.1 所述，是一种将一些文本字符串存储在char[]或VARCHAR[]中的方法。第二种是不用一个游标从一个查询结果中检索多行。如果没有一个数组，要处理由多个行组成的查询结果，我们需要使用一个游标以及FETCH命令。但是使用数组主变量，多个行可以被一次收取。该数组的长度必须被定义成足以容纳所有的行，否则很可能会发生一次缓冲区溢出。

下面的例子扫描pg_database系统表并且显示所有可用数据库的 OID 和名称：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
```

```
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    /* 一次检索多行到数组中。 */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子显示下面的结果（确切的值取决于本地环境）。

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

36.4.4.3.2. 结构

一个成员名称匹配查询结果列名的结构可以被用来一次检索多列。该结构使得我们能够在一个单一主变量中处理多列值。

下面的例子从pg_database系统表以及使用pg_database_size()函数检索可用数据库的OID、名称和尺寸。在这个例子中，一个成员名匹配SELECT结果的每一列的结构变量dbinfo_t被用来检索结果行，而不需要把多个主变量放在FETCH语句中。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid)
 AS size FROM pg_database;
    EXEC SQL OPEN cur1;

    /* 在达到结果集末尾时，跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* 将多列取到一个结构中。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval;
```

```
        /* 打印该结构的成员。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
 dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

这个例子会显示下列结果（确切的值取决于本地环境）。

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

结构主变量将列尽数“吸收”成结构的域。额外的列可以被分配给其他主变量。例如，上面的程序也可以使用结构外部的size变量重新构造：

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid)
 AS size FROM pg_database;
    EXEC SQL OPEN cur1;

    /* 在达到结果集末尾时，跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* 将多列取到一个结构中。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* 打印该结构的成员。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
 size);
    }

    EXEC SQL CLOSE cur1;
```

36.4.4.3.3. Typedefs

使用typedef关键词可以把新类型映射到已经存在的类型。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
```

```
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

注意你也可以使用：

```
EXEC SQL TYPE serial_t IS long;
```

这种声明不需要位于一个声明节之中。

36.4.4.3.4. 指针

你可以声明最常见类型的指针。不过注意，你不能使用指针作为不带自动分配内存的查询的目标变量。关于自动分配内存的详情请参考第 36.7 节

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

36.4.5. 处理非简单 SQL 数据类型

这一节包含关于如何处理 ECPG 应用中非标量以及用户定义的 SQL 级别数据类型。注意这和上一节中描述的简单类型主变量的处理有所不同。

36.4.5.1. 数组

ECPG 中不直接支持 SQL 级别的多维数组。一维 SQL 数组可以被映射到 C 数组主机变量，反之 亦然。不过，在创建一个语句时，ecpg并不知道列的类型，因此它无法检查一个 C 数组否是一个 SQL 数组的输入。在处理一个 SQL 语句的输出时，ecpg 有必需的信息并且进而检查是否两者都是 数组。

如果一个查询个别地访问一个数组的元素，那么这可以避免使用 ECPG 中的数组。然后，应该使用一个能被映射到该元素类型的类型的主变量。例如，如果一个列类型是integer数组，可以使用一个类型int的主变量。还有如果元素类型是varchar或text，可以使用一个类型char[]或VARCHAR[]的主变量。

这里是一个例子。假定有下面的表：

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

下面的例子程序检索数组的第四个元素并且把它存储到一个类型为int的主变量中：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
```

```
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

这个例子会显示下面的结果：

```
ii=4
```

要把多个数组元素映射到一个数组类型主变量中的多个元素，数组列的每一个元素以及主变量数组的每一个元素都必须被单独管理，例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

注意

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 错误 */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

在这种情况中不会正确工作，因为你无法把一个数组类型列直接映射到一个数组主变量。

另一种变通方案是在类型char[]或VARCHAR[]的主变量中存储数组的外部字符串表达。关于这种表达的详情请见第 8.15.2 节注意这意味着该数组无法作为一个主程序中的数组被自然地访问（没有解析文本表达的进一步处理）。

36.4.5.2. 组合类型

ECPG 中并不直接支持组合类型，但是有一种可能的简单变通方案。可用的变通方案和上述用于数组的方案相似：要么单独访问每一个属性或者使用外部字符串表达。

对于下列例子，假定有下面的类型和表：

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

最显而易见的解决方案是单独访问每一个属性。下面的程序通过单独选择类型comp_t的每一个属性从例子表中检索数据：

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* 将组合类型列的每一个元素放在 SELECT 列表中。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM
 t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将组合类型列的每一个元素取到主变量中。 */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

为了加强这个例子，在FETCH命令中存储值的主变量可以被集中在一个结构中。结构形式的主变量的详情可见第 36.4.4.3.2 节要切换到结构形式，该例子可以被改成下面的样子。两个主变量intval和textval变成comp_t结构的成员，并且该结构在FETCH命令中指定。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* 将组合类型列的每一个元素放在 SELECT 列表中。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM
 t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

尽管在FETCH命令中使用了一个结构，SELECT子句中的属性名还是要一个一个指定。可以通过使用一个*来要求该组合类型值的所有属性来改进。

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

通过这种方法，即便 ECPG 不理解组合类型本身，组合类型也能够几乎无缝地被映射到结构。

最后，也可以在类型char[]或VARCHAR[]的主变量中把组合类型值存储成它们的外部字符串表达。但是如果使用那种方法，就不太可能从主程序中访问该值的各个域了。

36.4.5.3. 用户定义的基础类型

ECPG 并不直接支持新的用户定义的基本类型。你可以使用外部字符串表达以及类型char[]或VARCHAR[]的主变量，并且这种方案事实上对很多类型都是合适和足够的。

这里有一个使用来自第 38.12 节例子里的数据类型complex的例子。该类型的外部字符串表达是(%f,%f)，它被定义在函数complex_in()以及第 38.12 节的complex_out()函数内。下面的例子把复杂类型值(1,1)和(3,3)插入到列a和b，并且之后把它们从表中选择出来。

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
```

```
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

这个例子会显示下列结果：

```
a=(1,1), b=(3,3)
```

另一种变通方案是避免在 ECPG 中直接使用用户定义的类型，而是创建一个在用户定义的类型和 ECPG 能处理的简单类型之间转换的函数或者造型。不过要注意，在类型系统中引入类型造型（特别是隐式造型）要非常小心。

例如，

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0)' + $2 * complex '(0,1)' $$;
```

在这个定义之后 ，下面的语句

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b),
 create_complex(:c, :d));
```

具有和

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

相同的效果。

36.4.6. 指示符

上述例子并没有处理空值。事实上，如果检索的例子从数据库取到了一个空值，它们将会产生一个错误。要能够向数据库传递空值或者从数据库检索空值，你需要对每一个包含数据的主变量追加一个次要主变量说明。这个次要主变量被称为指示符并且包含一个说明数据是否为空的标志，如果为空真正的主变量中的值就应该被忽略。这里有一个能正确处理检索空值的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:
```

```
 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

如果值不为空，指示符变量val_ind将为零；否则它将为负值。

指示符有另一种功能：如果指示符值为正，它表示值不为空，但是当它被存储在主变量中时已被截断。

如果参数-r no_indicator被传递给预处理器ecpg，它会工作在“无指示符”模式。在无指示符模式中，如果没有指定指示符变量，对于字符串类型空值被标志（在输入和输出上）为空串，对于整数类型空值被标志为类型的最低可能值（例如，int的是INT_MIN）。

36.5. 动态 SQL

在很多情况中，一个应用必须要执行的特定 SQL 语句在编写该应用时就已知。不过在某些情况中，SQL 语句在运行时构造或者由一个外部来源提供。这样你就不能直接把 SQL 语句嵌入到 C 源代码，不过有一种功能允许你调用在一个字符串变量中提供的任意 SQL 语句。

36.5.1. 执行没有结果集的语句

执行一个任意 SQL 语句的最简单方法是使用命令EXECUTE IMMEDIATE。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

EXECUTE IMMEDIATE可以被用于不返回结果集的 SQL 语句（例如 DDL、INSERT、UPDATE、DELETE）。你不能用这种方法执行检索数据的语句（例如SELECT）。下一节将描述如何执行这一种语句。

36.5.2. 执行一个有输入参数的语句

执行任意 SQL 语句的一种更强大的方法是准备它们一次并且在每次需要时执行该预备语句。也可以准备一个一般化的语句，然后通过替换参数执行它的特定版本。在准备语句时，在你想要稍后替换参数的地方写上问号。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

当你不再需要预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

36.5.3. 执行一个有结果集的语句

要执行一个只有单一结果行的 SQL 语句，可以使用EXECUTE。要保存结果，在其中增加一个INTO子句。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

一个EXECUTE命令可以有一个INTO子句、一个USING子句，可以同时有这两个子句，也可以不带这两个子句。

如果一个查询被期望返回多于一个结果行，应该如下列例子所示使用一个游标（关于游标详见第 36.3.2 节。

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

36.6. pgtypes 库

pgtypes 库将PostgreSQL数据库类型映射到 C 中等价的类型以便在 C 程序中使用。它还提供在 C 中对这些类型进行基本计算的函数，即不依赖PostgreSQL服务器进行计算。请看下面的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
```

```
    timestamp ts1, tsout;
    interval iv1;
    char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

36.6.1. 字符串

PGTYPESnumeric_to_asc之类的一些函数返回一个新分配的字符串的指针。这些结果应该用PGTYPESchar_free而不是free释放（这只在Windows上很重要，因为Windows上的内存分配和释放有时候需要由同一个库完成）。

36.6.2. numeric类型

numeric类型用来完成对任意精度的计算。PostgreSQL服务器中等效的类型请见第 8.1 节因为要用于任意精度，这种变量需要能够动态地扩展和收缩。这也是为什么你只能用PGTYPESnumeric_new和PGTYPESnumeric_free函数在堆上创建numeric变量。decimal类型与numeric类型相似但是在精度上有限制，decimal类型可以在堆上创建也可以在栈上创建。

下列函数可以用于numeric类型：

PGTYPESnumeric_new

请求一个指向新分配的numeric变量的指针。

```
numeric *PGTYPESnumeric_new(void);
```

PGTYPESnumeric_free

释放一个numeric类型，释放它所有的内存。

```
void PGTYPESnumeric_free(numeric *var);
```

PGTYPESnumeric_from_asc

从字符串记号中解析一个numeric类型。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

例如，可用的格式是： -2、 .794、 +3.44、 592.49E07或者 -32.84e-4。如果值能被成功地解析，将返回一个有效的指针，否则返回 NULL 指针。目前 ECPG 总是解析整个字符串并且因此当前不支持把第一个非法字符的地址存储在*endptr中。你可以安全地把endptr设置为 NULL。

PGTYPESnumeric_to_asc

返回由malloc分配的字符串的指针，它包含numeric类型num的字符串表达。

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

numeric值将被使用dscale小数位打印，必要时会圆整。结果必须用PGTYPESchar_free()释放。

PGTYPESnumeric_add

把两个numeric变量相加放到第三个numeric变量中。

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var1和var2相加放到结果变量result中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_sub

把两个numeric变量相减并且把结果返回到第三个numeric变量。

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var2从变量var1中减除。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_mul

把两个numeric变量相乘并且把结果返回到第三个numeric变量。

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var1和var2相乘。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_div

把两个numeric变量相除并且把结果返回到第三个numeric变量。

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

该函数用变量var2除变量var1。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_cmp

比较两个numeric变量。

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

这个函数比较两个numeric变量。错误时会返回INT_MAX。成功时，该函数返回三种可能结果之一：

- var1大于var2则返回 1
- 如果var1小于var2则返回 -1
- 如果var1和var2相等则返回 0

PGTYPESnumeric_from_int

把一个整数变量转换成一个numeric变量。

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

这个函数接受一个有符号整型变量并且把它存储在numeric变量var中。成功时返回 0，失败时返回 -1。

PGTYPESnumeric_from_long

把一个长整型变量转换成一个numeric变量。

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

这个函数接受一个有符号长整型变量并且把它存储在numeric变量var中。成功时返回 0，失败时返回 -1。

PGTYPESnumeric_copy

把一个numeric变量复制到另一个中。

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

这个函数把src指向的变量的值复制到dst指向的变量中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_from_double

把一个双精度类型的变量转换成一个numeric变量。

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

这个函数接受一个双精度类型的变量并且把结果存储在dst指向的变量中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_to_double

将一个numeric类型的变量转换成双精度。

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

这个函数将nv指向的变量中的numeric值转换成dp指向的双精度变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_int

将一个numeric类型的变量转换成整数。

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

该函数将nv指向的变量的numeric值转换成ip指向的整数变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_long

将一个numeric类型的变量转换成长整型。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

该函数将nv指向的变量的numeric值转换成ip指向的长整型变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_decimal

将一个numeric类型的变量转换成decimal。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

该函数将nv指向的变量的numeric值转换成ip指向的decimal变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_from_decimal

将一个decimal类型的变量转换成numeric。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

该函数将nv指向的变量的decimal值转换成ip指向的numeric变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。因为decimal类型被实现为numeric类型的一个有限的版本，在这个转换上不会发生溢出。

36.6.3. 日期类型

C 中的日期类型允许你的程序处理 SQL 日期类型的数据。PostgreSQL服务器的等效类型可见第 8.5 节

下列函数可以被用于日期类型：

PGTYPESdate_from_timestamp

从一个时间戳中抽取日期部分。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

该函数接收一个时间戳作为它的唯一参数并且从这个时间戳返回抽取的日期部分。

PGTYPESdate_from_asc

从日期的文本表达解析一个日期。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

该函数接收一个 C 的字符串str以及一个指向 C 字符串的指针endptr。当前 ECPG 总是解析完整的字符串并且因此当前不支持将第一个非法字符的地址存储在*endptr中。你可以安全地把endptr设置为 NULL。

注意该函数总是假定格式按照 MDY 格式化并且当前在 ECPG 中没有变体可以改变这种格式。

表 36.12 展示了所有允许的输入格式。

表 36.2.PGTYPESdate_from_asc的合法输入格式

输入	结果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	年以及积日
J2451187	儒略日
January 8, 99 BC	公元前 99 年

PGTYPESdate_to_asc

返回一个日期变量的文本表达。

```
char *PGTYPESdate_to_asc(date dDate);
```

该函数接收日期dDate作为它的唯一参数。它将以形式1999-01-18输出该日期，即以YYYY-MM-DD格式输出。结果必须用PGTYPESchar_free()释放。

PGTYPESdate_julmdy

从一个日期类型变量中抽取日、月和年的值。

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

该函数接收日期d以及一个指向有 3 个整数值的数组mdy的指针。变量名就表明了顺序：mdy[0]将被设置为包含月份，mdy[1]将被设置为日的值，而mdy[2]将包含年。

PGTYPESdate_mdyjul

从一个由 3 个整数构成的数组创建一个日期值，3 个整数分别指定日、月和年。

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

这个函数接收 3 个整数（mdy）组成的数组作为其第一个参数，其第二个参数是一个指向日期类型变量的指针，它被用来保存操作的结果。

PGTYPESdate_dayofweek

为一个日期值返回表示它是星期几的数字。

```
int PGTYPESdate_dayofweek(date d);
```

这个函数接收日期变量d作为它唯一的参数并且返回一个整数说明这个日期是星期几。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三
- 4 - 星期四
- 5 - 星期五
- 6 - 星期六

PGTYPESdate_today

得到当前日期。

```
void PGTYPESdate_today(date *d);
```

该函数接收一个指向一个日期变量（d）的指针并且把该参数设置为当前日期。

PGTYPESdate_fmt_asc

使用一个格式掩码将一个日期类型的变量转换成它的文本表达。

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

该函数接收要转换的日期（dDate）、格式掩码（fmtstring）以及将要保存日期的文本表达的字符串（outbuf）。

成功时，返回 0；如果发生错误，则返回一个负值。

下面是你可以使用的域指示符：

- dd - 一个月中的第几天。
- mm - 一年中的第几个月。
- yy - 两位数的年份。
- yyyy - 四位数的年份。
- ddd - 星期几的名称（简写）。
- mmm - 月份的名称（简写）。

所有其他字符会被原封不动地复制到输出字符串中。

表 36.3指出了一些可能的格式。这将给你一些线索如何使用这个函数。所有输出都是基于同一个日期：1959年11月23日。

表 36.3.PGTYPESdate_fmt_asc的合法输入格式

格式	结果
mmddyy	112359
ddmmyy	231159

格式	结果
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

PGTYPESdate_defmt_asc

使用一个格式掩码把一个 C 的 char*子返回串转换成一个日期类型的值。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

该函数接收一个用来保存操作结果的指向日期值的指针（d）、用于解析日期的格式掩码（fmt）以及包含日期文本表达的 C char* 串（str）。该函数期望文本表达匹配格式掩码。不过你不需要字符串和格式掩码的一一映射。该函数只分析相继顺序并且查找表示年份位置的文字yy或者yyyy、表示月份位置的mm以及表示日位置的dd。

表 36.给出了一些可能的格式。这将给你一些线索如何使用这个函数。

表 36.4.rdefmtdate的合法输入格式

格式	字符串	结果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	在 2525 年的七月二十八日，人类还将存在	2525-07-28
dd-mm-yy	也是 2525 年七月的二十八日	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

36.6.4. 时间戳类型

C 中的时间戳类型允许你的程序处理 SQL 时间戳类型的数据。PostgreSQL服务器的等效类型可见第 8.5 节

下列函数可用于时间戳类型：

PGTYPEStimestamp_from_asc

从文本表达解析一个时间戳并放到一个时间戳变量中。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

这个函数接收一个要解析的字符串（str）以及一个 C char* 的指针（endptr）。当前 ECPG 总是解析完整的字符串并且因此当前不支持将第一个非法字符的地址存储在*endptr中。你可以安全地把endptr设置为 NULL。

成功时该函数返回解析到的时间戳。错误时，会返回PGTYPESInvalidTimestamp并且errno会被设置为PGTYPES_TS_BAD_TIMESTAMP。关于这个值的重要提示请见PGTYPESInvalidTimestamp。

通常，该输入字符串能够包含一个允许的日期说明、一个空格字符和一个允许的时间说明的任意组合。注意 ECPG 不支持时区。它能够解析时区但是不会应用任何计算（例如PostgreSQL服务器所作的事情）。时区指示符会被无声无息地丢弃。

表 36.5包含输入字符串的一些例子。

表 36.5.PGTYPEStimestamp_from_asc的合法输入格式

输入	结果
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 （忽略了时区指示符）
J2451187 04:05-08:00	1999-01-08 04:05:00 （忽略了时区指示符）

PGTYPEStimestamp_to_asc

将一个日期转换成一个 C char* 字符串。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

该函数接收时间戳tstamp作为它的唯一参数并且返回一个分配好的包含该时间戳文本表达的字符串。结果必须用PGTYPESchar_free()释放。

PGTYPEStimestamp_current

检索当前的时间戳。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

该函数检索当前的时间戳并且将它保存在ts指向的时间戳变量。

PGTYPEStimestamp_fmt_asc

使用一个格式掩码将一个时间戳变量转换成一个 C char* 。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char
 *fmtstr);
```

该函数接收一个指向时间戳的指针作为它的第一个参数（ts）、一个指向输出缓冲区的指针（output）、为输出缓冲区分配的最大长度（str_len）以及用于转换的格式掩码（fmtstr）。

成功时，该函数返回 0；如果有错误发生，则返回一个负值。

你可以为格式掩码使用下列格式指示符。格式指示符就是用在libc的strftime函数中的那一些。任何非格式指示符将被复制到输出缓冲区。

- %A - 被完整的星期几名称的本国表达所替换。
- %a - 被简写星期几名称的本国表达所替换。
- %B - 被完整的月份名称的本国表达所替换。
- %b - 被简写月份名称的本国表达所替换。
- %C - 被十进制数（年份/100）所替换，单一数字会被前置一个零。
- %c - 被时间和日期的本国表达所替换。
- %D - 等效于%m/%d/%y。
- %d - 被十进制数（01-31）的日所替换。
- %E* %O* - POSIX 区域扩展。序列 %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy 被假定提供可供选择的表达。

  此外还实现了%OB来表达可供选择的月份名称（单独使用，不用提过的日）。
- %e - 被十进制数（01-31）的日所替换，单一数字被前置一个空格。
- %F - 等效于%Y-%m-%d。
- %G - 被替换为一个带有世纪的十进制数年份。这个年份是包含这一周大部分的年份（星期一作为这一周的第一天）。
- %g - 被替换为与%G中相同的年份，但是作为一个不带世纪的十进制数（00-99）。
- %H - 被替换为一个十进制数的小时（24 小时制，00-23）。
- %h - 和%b相同。
- %I - 被替换为一个十进制数的小时（12 小时制，01-12）。
- %j - 被替换为一个十进制数的积日（001-366）。
- %k - 被替换为一个十进制数的小时（24 小时制，00-23），单一数字被前置一个空白。
- %l - 被替换为一个十进制数的小时（12 小时制，01-12），单一数字被前置一个空白。
- %M - 被替换为一个十进制数的分钟（00-59）。
- %m - 被替换为一个十进制数的月份（01-12）。
- %n - 被替换为一个新行。
- %O* - 和%E*相同。
- %p - 根据情况被替换为“午前”或“午后”的本国表达。
- %R - 等效于%H:%M。

- %r - 等效于%I:%M:%S%p。
- %S - 被替换为十进制数的秒（00-60）。
- %s - 被替换为从 UTC 新纪元以来的秒数。
- %T - 等效于%H:%M:%S
- %t - 被替换为一个制表符。
- %U - 被替换为十进制数的周数（周日作为一周的第一天，00-53）。
- %u - 被替换为十进制数的星期几（周一作为一周的第一天，1-7）。
- %V - 被替换为十进制数的周数（周一作为一周的第一天，01-53）。如果包含 1 月 1 日的周在新年中有 4 天或更多天，那么它是第一周。否则它是前一年的最后一周，并且下一周是第一周。
- %v - 等效于%e-%b-%Y。
- %W - 被替换为十进制数的周数（周一作为一周的第一天，00-53）。
- %w - 被替换为十进制数的星期几（0-6，周日作为一周的第一天）。
- %X - 被替换为时间的本国表达。
- %x - 被替换为日期的本国表达。
- %Y - 被替换为十进制数的带世纪的年份。
- %y - 被替换为十进制数的不带世纪的年份（00-99）。
- %Z - 被替换为时区名称。
- %z - 被替换为相对于 UTC 的时区偏移；一个前导的加号表示 UTC 东部，一个负号表示 UTC 西部，接着是分别有两个数字的小时和分钟并且它们之间没有定界符（RFC 822 日期头部的一般形式）。
- %+ - 被替换为日期和时间的本国表达。
- %-* - GNU libc 扩展。在执行数值输出时不做任何填充。
- $_* - GNU libc 扩展。显式地指定用空格填充。
- %0* - GNU libc 扩展。显式地指定用零填充。
- %% - 被替换为%。

PGTYPEStimestamp_sub

从一个时间戳中减去另一个时间戳并且把结果保存在一个区间类型的变量中。

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数将从ts1指向的时间戳变量中减去ts2指向的时间戳变量，并且将把结果存储在iv指向的区间变量中。

成功时，该函数返回 0；发生错误时则返回一个负值。

PGTYPEStimestamp_defmt_asc

用一个格式掩码从时间戳的文本表达解析其值。

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

该函数接收一个放在变量str中的时间戳文本表达以及放在变量fmt中的要使用的格式掩码。结果将被存放在d指向的变量中。

如果格式掩码fmt是NULL，该函数将回退到使用默认的格式掩码%Y-%m-%d %H:%M:%S。

这是PGTYPEStimestamp_fmt_asc的逆函数。可能的格式掩码项可以参考那个函数的文档。

PGTYPEStimestamp_add_interval

把一个interval变量加到一个时间戳变量上。

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp
 *tout);
```

该函数接收一个指向时间戳变量的指针tin以及一个指向interval变量的指针span。它把interval加到时间戳上，然后将结果时间戳保存在tout指向的变量中。

成功时该函数返回0，如果发生错误则返回一个负值。

PGTYPEStimestamp_sub_interval

从一个时间戳变量中减去一个interval变量。

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp
 *tout);
```

该函数从tin指向的时间戳变量中减去span指向的interval变量，然后把结果保存在tout指向的变量中。

成功时该函数返回0，如果发生错误则返回一个负值。

36.6.5. 区间类型

C 中的区间类型允许你的程序处理 SQL 区间类型的数据。PostgreSQL服务器的等效类型可见第 8.5 节

下列函数可以用于区间类型：

PGTYPESinterval_new

返回一个指向新分配的区间变量的指针。

```
interval *PGTYPESinterval_new(void);
```

PGTYPESinterval_free

释放先前分配的区间变量的内存。

```
void PGTYPESinterval_new(interval *intvl);
```

PGTYPESinterval_from_asc

从文本表达解析一个区间。

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

该函数解析输入字符串str并且返回一个已分配的区间变量的指针。目前 ECPG 总是解析整个字符串并且因此当前不支持把第一个非法字符的地址存储在*endptr中。你可以安全地把endptr设置为 NULL。

PGTYPESinterval_to_asc

将一个区间类型的变量转换成它的文本表达。

```
char *PGTYPESinterval_to_asc(interval *span);
```

该函数将span指向的区间变量转换成一个 C char*。输出看起来像这个例子： @ 1 day 12 hours 59 mins 10 secs。结果必须用PGTYPESchar_free()释放。

PGTYPESinterval_copy

复制一个区间类型的变量。

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

该函数将intvlsrc指向的区间变量复制到intvldest指向的区间变量。注意你需要现为目标变量分配好内存。

36.6.6. decimal类型

decimal类型和numeric类型相似。不过，它被限制为最大精度是 30 个有效位。与numeric类型只能在堆上创建相反，decimal类型既可以在栈上也可以在堆上创建（使用函数PGTYPESdecimal_new 和PGTYPESdecimal_free）。在第 36.15 节描述的Informix兼容模式中有很多其它函数可以处理decimal类型。

下列函数可以被用于decimal类型并且不仅被包含于libcompat库中。

PGTYPESdecimal_new

要求一个指向新分配的decimal变量的指针。

```
decimal *PGTYPESdecimal_new(void);
```

PGTYPESdecimal_free

释放一个decimal类型，释放它的所有内存。

```
void PGTYPESdecimal_free(decimal *var);
```

36.6.7. pgtypeslib 的 errno 值

PGTYPES_NUM_BAD_NUMERIC

一个参数应该包含一个numeric变量（或者指向一个numeric变量），但是实际上它的内存表达非法。

PGTYPES_NUM_OVERFLOW

发生一次溢出。由于numeric类型可以处理几乎任何精度，将一个numeric变量转换成其他类型可能导致溢出。

PGTYPES_NUM_UNDERFLOW

发生一次下溢。由于numeric类型可以处理几乎任何精度，将一个numeric变量转换成其他类型可能导致下溢。

PGTYPES_NUM_DIVIDE_ZERO

尝试了一次除零。

PGTYPES_DATE_BAD_DATE

一个非法的日期字符串被传给了PGTYPESdate_from_asc函数。

PGTYPES_DATE_ERR_EARGS

非法参数被传给了PGTYPESdate_defmt_asc函数。

PGTYPES_DATE_ERR_ENOSHORTDATE

PGTYPESdate_defmt_asc函数在输入字符串中发现了一个非法记号。

PGTYPES_INTVL_BAD_INTERVAL

一个非法的区间字符串被传给了PGTYPESinterval_from_asc函数，或者一个非法的区间值被传给了PGTYPESinterval_to_asc函数。

PGTYPES_DATE_ERR_ENOTDMY

在PGTYPESdate_defmt_asc函数中有日/月/年不匹配的赋值。

PGTYPES_DATE_BAD_DAY

PGTYPESdate_defmt_asc函数发现了月中的一个非法日值。

PGTYPES_DATE_BAD_MONTH

PGTYPESdate_defmt_asc函数发现了一个非法的月值。

PGTYPES_TS_BAD_TIMESTAMP

一个非法的时间戳字符串被传给了PGTYPEStimestamp_from_asc函数，或者一个非法的时间戳值被传给了PGTYPEStimestamp_to_asc函数。

PGTYPES_TS_ERR_EINFTIME

在一个无法处理无限时间戳值的环境中遇到了这样一个值。

36.6.8. pgtypeslib 的特殊常量

PGTYPESInvalidTimestamp

表示一个非法时间戳的时间戳类型值。在解析错误时，函数PGTYPEStimestamp_from_asc会返回这个值。注意由于timestamp数据类型的内部表达，PGTYPESInvalidTimestamp在同时也是一个合法的时间戳。它被设置为1899-12-31 23:59:59。为了检测到错误，确认你的应用在每次调用PGTYPEStimestamp_from_asc后不仅仅测试PGTYPESInvalidTimestamp，还应该测试errno != 0。

36.7. 使用描述符区域

一个 SQL 描述符区域是一种处理SELECT、FETCH或者DESCRIBE语句结果的高级方法。一个 SQL 描述符区域把数据中一行的数据及元数据项组合到一个数据结构中。在执行动态 SQL 语句时（结果行的性质无法提前预知），元数据特别有用。PostgreSQL 提供两种方法来使用描述符区域：命名 SQL 描述符区域和 C 结构 SQLDA。

36.7.1. 命名 SQL 描述符区域

一个命名 SQL 描述符区域由一个头部以及一个或多个条目描述符区域构成，头部包含与整个描述符相关的信息，而条目描述符区域则描述结果行中的每一列。

在使用 SQL 描述符区域之前，需要先分配一个：

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

identifier 会作为该描述符区域的“变量名”。当不再需要该描述符时，应当释放它：

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

要使用一个描述符区域，把它指定为INTO子句的存储目标（而不是列出主变量）：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集为空，该描述符区域仍然会包含查询的元数据，即域的名称。

对于还没有执行的预备查询，DESCRIBE可以被用来得到其结果集的元数据：

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

在 PostgreSQL 9.0 之前，SQL关键词是可选的，因此使用DESCRIPTOR和SQL DESCRIPTOR都会产生命名 SQL 描述符区域。现在该关键词是强制性的，省略SQL关键词会产生 SQLDA 描述符区域（见第 36.7.2 节。

在DESCRIBE和FETCH语句中，INTO和USING关键词的使用相似：它们产生结果集以及一个描述符区域中的元数据。

现在我们如何从描述符区域得到数据呢？你可以把描述符区域看成是一个具有命名域的结构。要从头部检索一个域的值并且把它存储到一个主变量中，可使用下面的命令：

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

当前，只定义了一个头部域：COUNT，它告诉我们有多少个条目描述符区域（也就是，结果中包含多少列）。主变量需要是一个整数类型。要从条目描述符区域中得到一个域，可使用下面的命令：

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

num可以是一个字面整数或者包含一个整数的主变量。可能的域有：

CARDINALITY （整数）

结果集中的行数

DATA

实际的数据项（因此，这个域的数据类型取决于查询）

DATETIME_INTERVAL_CODE （整数）

当TYPE是9时， DATETIME_INTERVAL_CODE将具有以下值之一： 1 表示 DATE， 2 表示 TIME， 3 表示 TIMESTAMP， 4 表示 TIME WITH TIME ZONE， 5 表示 TIMESTAMP WITH TIME ZONE。

DATETIME_INTERVAL_PRECISION （整数）

没有实现

INDICATOR （整数）

指示符（表示一个空值或者一个值截断）

KEY_MEMBER （整数）

没有实现

LENGTH （整数）

以字符计的数据长度

NAME (string)

列名

NULLABLE （整数）

没有实现

OCTET_LENGTH （整数）

以字节计的数据字符表达的长度

PRECISION （整数）

精度（用于类型numeric）

RETURNED_LENGTH （整数）

以字符计的数据长度

RETURNED_OCTET_LENGTH （整数）

以字节计的数据字符表达的长度

SCALE （整数）

比例（用于类型numeric）

TYPE （整数）

列的数据类型的数字编码

在EXECUTE、DECLARE以及OPEN语句中，INTO和USING关键词的效果不同。也可以手工建立一个描述符区域来为一个查询或者游标提供输入参数，并且USING SQL DESCRIPTOR name是用来传递输入参数给参数化查询的方法。建立一个命名 SQL 描述符区域的语句如下：

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQL 支持在一个FETCH语句中检索多于一个记录并且在这种情况下把主变量假定为一个数组来存储数据。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

36.7.2. SQLDA 描述符区域

SQLDA 描述符区域是一个 C 语言结构，它也能被用来得到一个查询的结果集和元数据。一个结构存储一个来自结果集的记录。

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

注意SQL关键词被省略了。第 36.7.1 节关于INTO和USING关键词用例的段落在一定条件下也适用于这里。在一个DESCRIBE语句中，如果使用了INTO关键词，则DESCRIPTOR关键词可以完全被省略：

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

使用 SQLDA 的程序的一般流程是：

1. 准备一个查询，并且为它声明一个游标。
2. 为结果行声明一个 SQLDA 。
3. 为输入参数声明一个 SQLDA，并且初始化它们（内存分配、参数设置）。
4. 用输入 SQLDA 打开一个游标。
5. 从游标中取得行，并且把它们存储到一个输出 SQLDA。
6. 从输出 SQLDA 读取值到主变量中（必要时使用转换）。
7. 关闭游标。
8. 关闭为输入 SQLDA 分配的内存区域。

36.7.2.1. SQLDA 数据结构

SQLDA 使用三种数据结构类型：sqlda_t、sqlvar_t以及struct sqlname。

> **提示**
>
> PostgreSQL 的 SQLDA 与 IBM DB2 Unversal 数据库中相似的数据结构，因此一些 DB2 的 SQLDA 的技术信息有助于更好地理解 PostgreSQL 的 SQLDA。

36.7.2.1.1. sqlda_t 结构

结构类型sqlda_t是实际 SQLDA 的类型。它保存一个记录。并且两个或者更多个sqlda_t结构能够以desc_next域中的指针连接成一个链表，这样可以表示一个有序的行集合。因此，当两个或多个行被取得时，应用可以通过沿着每一个sqlda_t节点中的desc_next指针读取它们。

sqlda_t的定义是：

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

域的含义是：

sqldaid

它包含一个字符串"SQLDA "。

sqldabc

它包含已分配空间的尺寸（以字节计）。

sqln

当它被传递给使用USING关键词的OPEN、DECLARE或者EXECUTE语句时，它包含用于一个参数化查询实例的输入参数的数目。在它被用作SELECT、EXECUTE或FETCH语句的输出时，它的值和sqld一样

sqld

它包含一个结果集中的域的数量。

desc_next

如果查询返回不止一个记录，会返回多个链接在一起的 SQLDA 结构，并且desc_next保存一个指向下一个项的指针。

sqlvar

这是结果集中列的数组。

36.7.2.1.2. sqlvar_t 结构

结构类型sqlvar_t保存一个列值和元数据（例如类型和长度）。该类型的定义是：

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

各个域的含义是：

sqltype

包含该域的类型标识符。值可以参考ecpgtype.h中的enum ECPGttype。

sqllen

包含域的二进制长度，例如ECPGt_int是 4 字节。

sqldata

指向数据。数据的格式在第 36.4.4 节描述。

sqlind

指向空指示符。0 表示非空，-1 表示空。

sqlname

域的名称。

36.7.2.1.3. struct sqlname 结构

一个struct sqlname结构保持一个列名。它被用作sqlvar_t结构的一个成员。该结构的定义是：

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

各个域的含义是：

length

包含域名称的长度。

data

包含实际的域名称。

36.7.2.2. 使用一个 SQLDA 检索一个结果集

通过一个 SQLDA 检索一个查询结果集的一般步骤是：

1. 声明一个sqlda_t结构来接收结果集。
2. 执行 FETCH/EXECUTE/DESCRIBE 命令来处理一个指定已声明 SQLDA 的查询。
3. 通过查看sqlda_t结构的成员sqln来检查结果集中记录的数量。
4. 从sqlda_t结构的成员sqlvar[0]、sqlvar[1]等中得到每一列的值。
5. 沿着sqlda_t结构的成员desc_next指针到达下一行（sqlda_t）。
6. 根据你的需要重复上述步骤。

这里是一个通过 SQLDA 检索结果集的例子。

首先，声明一个sqlda_t结构来接收结果集。

```
sqlda_t *sqlda1;
```

接下来，指定一个命令中的 SQLDA。这是一个FETCH命令的例子。

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

运行一个循环顺着链表来检索行。

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

在循环内部，运行另一个循环来检索行中每一列的数据（sqlvar_t结构）。

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

要得到一列的值，应检查sqlvar_t结构的成员sqltype的值。然后，根据列类型切换到一种合适的方法从sqlvar域中复制数据到一个主变量。

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) -
 1 : sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

36.7.2.3. 使用一个 SQLDA 传递查询参数

使用一个 SQLDA 传递输入参数给一个预备查询的一般步骤是：

1. 创建一个预备查询（预备语句）。

2. 声明一个 sqlda_t 结构作为输入 SQLDA。

3. 为输入 SQLDA 分配内存区域（作为 sqlda_t 结构）。

4. 在分配好的内存中设置（复制）输入值。

5. 打开一个说明了输入 SQLDA 的游标。

这里是一个例子。

首先，创建一个预备语句。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE
 d.oid = s.datid AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

接下来为一个 SQLDA 分配内存，并且在sqlda_t结构的sqln成员变量中设置输入参数的数量。当预备查询要求两个或多个输入参数时，应用必须分配额外的内存空间，空间的大小为(参数数目 - 1) * sizeof(sqlvar_t)。这里的例子展示了为两个输入参数分配内存空间。

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* 输入变量的数目 */
```

内存分配之后，把参数值存储到sqlvar[]数组（当 SQLDA 在接收结果集时，这也是用来检索列值的数组）。在这个例子中，输入参数是"postgres"（字符串类型）和1（整数类型）。

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

通过打开一个游标并且说明之前已经建立好的 SQLDA，输入参数被传递给预备语句。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

最后，用完输入 SQLDA 后必须显式地释放已分配的内存空间，这与用于接收查询结果的 SQLDA 不同。

```
free(sqlda2);
```

36.7.2.4. 一个使用 SQLDA 的应用例子

这里是一个例子程序，它描述了如何按照输入参数的指定从系统目录中取得数据库的访问统计。

这个应用在数据库 OID 上连接两个系统表（pg_database 和 pg_stat_database），并且还取得和显示通过两个输入参数（一个数据库postgres和 OID 1）检索到的数据库统计。

首先，为输入和输出分别声明一个 SQLDA。

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* 一个输出描述符 */
sqlda_t *sqlda2; /* 一个输入描述符  */
```

接下来，连接到数据库，准备一个语句并且为预备语句声明一个游标。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s
 WHERE d.oid=s.datid AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

然后，为输入参数在输入 SQLDA 中放入一些值。为输入 SQLDA 分配内存，并且在sqln中设置输入参数的数目。在sqlvar结构的sqltype、sqldata和sqllen中存入类型、值和值长度。

```
    /* 为输入参数创建 SQLDA 结构。 */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* 输入变量的数量 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

设置完输入 SQLDA 之后，用输入 SQLDA 打开一个游标。

```
    /* 用输入参数打开一个游标。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

从打开的游标中取行到输出 SQLDA 中（通常，你不得不在循环中反复调用FETCH来取出结果集中的所有行）。

```
    while (1)
    {
        sqlda_t *cur_sqlda;

        /* 分配描述符给游标  */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

再后，沿着sqlda_t结构的链表从 SQLDA 中检索取得的记录。

```
    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

读取第一个记录中的每一列。列的数量被存储在sqld中，第一列的实际数据被存储在sqlvar[0]中，两者都是sqlda_t结构的成员。

```
        /* 打印一行中的每一列。 */
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

现在，列数据已经被存在了变量v中。把每个数据复制到主变量中，列的类型可以查看。

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ?
 sizeof(var_buf)-1 : sqllen));
                    break;

                case ECPGt_int: /* 整数 */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

处理所有记录后关闭游标，并且从数据库断开连接。

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

整个程序显示在例 36.中。

例 36.1. 示例 SQLDA 程序

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* 用于输出的描述符 */
sqlda_t *sqlda2; /* 用于输入的描述符 */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s
 WHERE d.oid=s.datid AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* 为一个输入参数创建一个 SQLDA 结构 */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* 输入变量的数量 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

    /* 用输入参数打开一个游标。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* 给游标分配描述符  */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

```
        for (cur_sqlda = sqlda1 ;
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* 打印一行中的每一列。 */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ?
sizeof(var_buf)-1 : sqllen) );
                        break;

                    case ECPGt_int: /* 整数 */
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* 大整数 */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned
char) sqldata[i]);
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }
```

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

这个例子的输出应该看起来类似下面的结果（一些数字会变化）。

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
```

```
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

36.8. 错误处理

这一节描述在一个嵌入式 SQL 程序中如何处理异常情况和警告。有两种非互斥的工具可以用于这个目的。

- 可以使用WHENEVER命令配置回调来处理警告和错误情况。
- 可以从sqlca变量中获得错误或警告的详细信息。

36.8.1. 设置回调

一种捕捉错误和警告的简单方法是设置一个特殊的动作，只要一个特定情况发生就执行该动作。通常是这样：

```
EXEC SQL WHENEVER condition action;
```

condition可以是下列之一：

SQLERROR

只要在 SQL 语句执行期间发生一个错误就调用指定的动作。

SQLWARNING

只要在 SQL 语句执行期间发生一个警告就调用指定的动作。

NOT FOUND

只要一个 SQL 语句检索或者影响零行就调用指定的动作（这种情况不是一个错误，但是你可能需要特别地处理它）。

action可以是下列之一：

CONTINUE

这实际上表示该情况被忽略。这是默认值。

GOTO label
GO TO label

调到指定的标签（使用一个 C goto语句）。

SQLPRINT

把一个消息打印到标准错误。对于简单程序或原型开发中这很有用。消息的细节无法配置。

STOP

调用exit(1)终止程序。

DO BREAK

执行 C 语句break。只应被用在循环或switch语句中。

DO CONTINUE

执行C语句continue。这应该只被用在循环语句中。如果被执行，将导致控制流返回到循环的顶层。

```
CALL name (args)
DO name (args)
```

用指定参数调用指定的C函数（这种用法不同于正常PostgreSQL语法中CALL和DO的含义）。

SQL 标准只提供动作CONTINUE和GOTO（以及GO TO）。

这里有一个可能会用在简单程序中的例子。当一个警告发生时它打印一个简单消息，而发生一个错误时它会中止程序：

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

语句EXEC SQL WHENEVER是 SQL 预处理器的一个指令，而不是一个 C 语句。不管 C 程序的控制流程如何，该语句设置的错误或警告动作适用于所有位于处理程序设置点之后的嵌入式 SQL 语句，除非在第一个EXEC SQL WHENEVER和导致情况的 SQL 语句之间为同一个情况设置了不同的动作。因此下面的两个 C 程序都不会得到预期的效果：

```
/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

36.8.2. sqlca

为了更强大的错误处理，嵌入式 SQL 接口提供了一个名为sqlca（SQL 通讯区域）的全局变量，它具有下面的结构：

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（在一个多线程程序中，每一个线程会自动得到它自己的sqlca副本。这和对于标准 C 全局变量errno的处理相似。）

sqlca覆盖了警告和错误。如果执行一个语句时发生了多个警告或错误，那么sqlca将只包含关于最后一个的信息。

如果在上一个SQL语句中没有产生错误，sqlca.sqlcode将为 0 并且sqlca.sqlstate将为"00000"。如果发生一个警告或错误，则sqlca.sqlcode将为负并且sqlca.sqlstate将不为"00000"。一个正的sqlca.sqlcode表示一种无害的情况，例如上一个查询返回零行。sqlcode和sqlstate是两种不同的错误代码模式，详见下文。

如果上一个 SQL 语句成功，那么sqlca.sqlerrd[1]包含被处理行的 OID （如果可用），并且sqlca.sqlerrd[2]包含被处理或被返回的行数（如果适用于该命令）。

在发生一个错误或警告的情况下，sqlca.sqlerrm.sqlerrmc将包含一个描述该错误的字符串。域sqlca.sqlerrm.sqlerrml包含存储在sqlca.sqlerrm.sqlerrmc中错误消息的长度（strlen()的结果，对于一个 C 程序员来说并不感兴趣）。注意一些消息可能太长不能适应定长的sqlerrmc数组，它们将被截断。

在发生一个警告的情况下，sqlca.sqlwarn[2]被设置为W（在所有其他情况中，它被设置为不同于W的东西）。如果sqlca.sqlwarn[1]被设置为W，那么一个值被存储在一个主变量中时会被截断。如果任意其他元素被设置为指示一个警告，sqlca.sqlwarn[0]会被设置为W。

域sqlcaid、 sqlcabc、 sqlerrp以及 sqlerrd的剩余元素还有 sqlwarn当前不包含有用的信息。

SQL 标准中没有定义sqlca结构，但是在一些其他的 SQL 数据系统中都有实现。在核心上这些定义都想死，但是如果你想要编写可移植的应用，那么你应该仔细研究不同的实现。

这里有一个整合使用WHENEVER和sqlca的例子，当一个错误发生时打印出sqlca的内容。在安装一个更“用户友好”的错误处理器之前，这可能对调试或开发原型应用有用。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
 sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
```

```
 sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0],
 sqlca.sqlwarn[1], sqlca.sqlwarn[2],
                                                          sqlca.sqlwarn[3],
 sqlca.sqlwarn[4], sqlca.sqlwarn[5],
                                                          sqlca.sqlwarn[6],
 sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

结果看起来像（这里的错误是一个拼写错误的表名）：

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

36.8.3. SQLSTATE 与 SQLCODE

域sqlca.sqlstate以及sqlca.sqlcode是提供错误代码的两种不同模式。两种都源自于 SQL 标准，但是在标准的 SQL-92 版本中SQLCODE已经被标记为弃用并且在后面的版本中被删除。因此，强烈建议新应用使用SQLSTATE。

SQLSTATE是一个五字符数组。这五个字符包含数字或大写字母，它表示多种错误或警告情况的代码。SQLSTATE具有一种层次模式：前两个字符表示情况的总体分类，后三个字符表示总体情况的子类。代码00000表示一种成功的状态。SQL 标准中的大部分都有对应的SQLSTATE代码。PostgreSQL服务器本地支持SQLSTATE错误代码，因此通过在所有应用中自始至终使用这种错误代码模式可以实现高度的一致性。进一步的信息请见附录 A。

被弃用的错误代码模式SQLCODE是一个简单的整数。值为 0 表示成功，一个正值表示带附加信息的成功，一个负值表示一个错误。SQL 标准只定义了正值 +100，它表示上一个命令返回或者影响了零行，并且没有特定的负值。因此，这种模式只能实现很可怜的可移植性并且不具有层次性的代码分配。历史上，PostgreSQL的嵌入式 SQL 处理器已经分配了一些特定的SQLCODE值供它使用，它们的数字值和符号名称被列在下文。记住这些对其他 SQL 实现不是可移植的。为了简化移植应用到SQLSTATE模式，对应的SQLSTATE也被列出。不过，在两种模式之间没有一对一或者一对多的映射（事实上是多对多），因此在每一种情况下你都应该参考附录 A中列出的全局SQLSTATE。

这些是已分配的SQLCODE值：

0 (ECPG_NO_ERROR)

表示没有错误（SQLSTATE 00000）。

100 (ECPG_NOT_FOUND)

这是一种无害情况，它表示上一个命令检索或者处理了零行，或者你到达了游标的末尾（SQLSTATE 02000）。

在一个循环中处理一个游标时，你可以使用这个代码作为一种方法来检测何时中止该循环，像这样：

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

但是WHENEVER NOT FOUND DO BREAK实际上会在内部这样做，因此显式地把它写出来通常没有什么好处。

-12 (ECPG_OUT_OF_MEMORY)

表示你的虚拟内存已被耗尽。数字值被定义为-ENOMEM（SQLSTATE YE001）。

-200 (ECPG_UNSUPPORTED)

表示预处理器已经产生了一些该库不知道的东西。也许你正在运行一个不兼容版本的预处理和库（SQLSTATE YE002）。

-201 (ECPG_TOO_MANY_ARGUMENTS)

这表示命令指定了超过该命令预期数量的主变量（SQLSTATE 07001 或 07002）。

-202 (ECPG_TOO_FEW_ARGUMENTS)

这表示命令指定的主变量数量低于该命令的预期（SQLSTATE 07001 或 07002）

-203 (ECPG_TOO_MANY_MATCHES)

这意味着一个查询已经返回了多个行，但是该语句只准备存储一个结果行（例如，因为指定的变量不是数组）（SQLSTATE 21000）。

-204 (ECPG_INT_FORMAT)

主变量是类型int而数据库中的数据是一种不同的类型并且含有一个不能被解释为int的值。该库使用strtol()进行这种转换（SQLSTATE 42804）。

-205 (ECPG_UINT_FORMAT)

主变量是类型unsigned int而数据库中的数据是一种不同的类型并且含有一个不能被解释为unsigned int的值。该库使用strtoul()进行这种转换（SQLSTATE 42804）。

-206 (ECPG_FLOAT_FORMAT)

主变量是类型float而数据库中的数据是另一种类型并且含有一个不能被解释为float的值。该库使用strtod()进行这种转换（SQLSTATE 42804）。

-207 (ECPG_NUMERIC_FORMAT)

主变量是类型numeric而数据库中的数据是另一种类型并且含有一个不能被解释为numeric的值（SQLSTATE 42804）。

-208 (ECPG_INTERVAL_FORMAT)

主变量是类型interval而数据库中的数据是另一种类型并且含有一个不能被解释为interval的值（SQLSTATE 42804）。

-209 (ECPG_DATE_FORMAT)

主变量是类型date而数据库中的数据是另一种类型并且含有一个不能被解释为date的值（SQLSTATE 42804）。

-210 (ECPG_TIMESTAMP_FORMAT)

主变量是类型timestamp而数据库中的数据是另一种类型并且含有一个不能被解释为timestamp的值（SQLSTATE 42804）。

-211 (ECPG_CONVERT_BOOL)

这表示主变量是类型bool而数据库中的数据既不是't'也不是'f'（SQLSTATE 42804）。

-212 (ECPG_EMPTY)

发送给PostgreSQL服务器的语句是空的（通常在一个嵌入式 SQL 程序中不会发生，因此它可能指向一个内部错误）（SQLSTATE YE002）。

-213 (ECPG_MISSING_INDICATOR)

返回了一个空值并且没有提供空值指示符（SQLSTATE 22002）。

-214 (ECPG_NO_ARRAY)

在要求一个数组的地方使用了一个普通变量（SQLSTATE 42804）。

-215 (ECPG_DATA_NOT_ARRAY)

在一个要求数组值的地方数据库返回了一个普通变量（SQLSTATE 42804）。

-216 (ECPG_ARRAY_INSERT)

该值不能被插入到数组（SQLSTATE 42804）。

-220 (ECPG_NO_CONN)

程序尝试访问一个不存在的连接（SQLSTATE 08003）。

-221 (ECPG_NOT_CONN)

程序尝试访问一个存在的连接但是它没有打开（这是一个内部错误）（SQLSTATE YE002）。

-230 (ECPG_INVALID_STMT)

你尝试使用的语句还没有被准备好（SQLSTATE 26000）。

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

重复键错误，违背唯一约束（Informix 兼容模式）（SQLSTATE 23505）。

-240 (ECPG_UNKNOWN_DESCRIPTOR)

没有找到指定的描述符。你尝试使用的语句还没有被准备好（SQLSTATE 33000）。

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

指定的描述符超出范围（SQLSTATE 07009）。

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

请求了一个非法的描述符（这是一个内部错误）（SQLSTATE YE002）。

-243 (ECPG_VAR_NOT_NUMERIC)

在执行一个动态语句期间，数据库返回了一个numeric值而主变量不是numeric的（SQLSTATE 07006）。

-244 (ECPG_VAR_NOT_CHAR)

在执行一个动态语句期间，数据库返回了一个非numeric值而主变量是numeric的（SQLSTATE 07006）。

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

子查询的结果不是单一行（Informix 兼容模式）（SQLSTATE 21000）。

-400 (ECPG_PGSQL)

PostgreSQL服务器导致了某个错误。该消息包含来自PostgreSQL服务器的错误消息。

-401 (ECPG_TRANS)

PostgreSQL服务器通知我们不能启动、提交或回滚事务（SQLSTATE 08007）。

-402 (ECPG_CONNECT)

到数据库的连接尝试没有成功（SQLSTATE 08001）。

-403 (ECPG_DUPLICATE_KEY)

重复键错误，违背唯一约束（SQLSTATE 23505）。

-404 (ECPG_SUBSELECT_NOT_ONE)

子查询的结果不是单一行（SQLSTATE 21000）。

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

指定了一个非法的游标名（SQLSTATE 34000）。

-603 (ECPG_WARNING_IN_TRANSACTION)

事务正在进行（SQLSTATE 25001）。

-604 (ECPG_WARNING_NO_TRANSACTION)

没有活动（正在进行）的事务（SQLSTATE 25P01）。

-605 (ECPG_WARNING_PORTAL_EXISTS)

指定了一个现有的游标名（SQLSTATE 42P03）。

36.9. 预处理器指令

一些预处理器指令可以用来改变ecpg预处理器解析和处理一个文件的方式。

36.9.1. 包括文件

要包括一个外部文件到你的嵌入式 SQL 程序中，可以用：

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

嵌入式 SQL 预处理器将查找一个名为filename.h的文件，处理它并且把它包括在结果 C 输出中。这样，被包括文件中的嵌入式 SQL 语句会被正确地处理。

ecpg预处理器将以下列顺序在几个目录中搜索一个文件：

- 当前目录
- /usr/local/include
- PostgreSQL 的包括目录，在编译时定义（例如/usr/local/pgsql/include）
- /usr/include

但是当使用EXEC SQL INCLUDE "filename"时，只有当前目录会被搜索。

在每一个目录中，预处理器将首先按给定的文件名搜索，如果没有找到将会追加.h到文件名并且重试（除非指定的文件名已经具有该后缀）。

注意EXEC SQL INCLUDE不同于：

```
#include <filename.h>
```

因为这个文件不服从 SQL 命令预处理。自然地，你可以继续使用 C 的#include指令来包括其他头文件。

> **注意**
>
> 包括文件名是大小写敏感的，即使EXEC SQL INCLUDE命令的剩余部分遵守通常的 SQL 大小写敏感规则。

36.9.2. define 和 undef 指令

与 C 中我们熟知的指令#define相似，嵌入式 SQL 具有类似的概念：

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

因此你可以定义一个名称：

```
EXEC SQL DEFINE HAVE_FEATURE;
```

并且你也可以定义常量：

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

使用undef来移除一个之前的定义：

```
EXEC SQL UNDEF MYNUMBER;
```

当然在你的嵌入式 SQL 程序中你可以继续使用 C 版本的#define和#undef。区别在于你定义的值会在哪里被计算。如果你使用EXEC SQL DEFINE，那么ecpg预处理器会计算这些定义并且替换值。例如，如果你写：

```
EXEC SQL DEFINE MYNUMBER 12;
```

```
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

那么ecpg将已经做过替换并且你的 C 编译器将永远不会看见名为MYNUMBER的任何名称或标识符。注意你不能把#define用于一个将要在一个嵌入式 SQL 查询中使用的常量，因为在这种情况下嵌入式 SQL 预编译器不能看到这个声明。

36.9.3. ifdef、ifndef、else、elif 以及 endif 指令

你可以使用下列指定来有条件地编译代码小节：

EXEC SQL ifdef name;

检查一个name，如果已经用EXEC SQL define name创建了name就处理接下来的行。

EXEC SQL ifndef name;

检查一个name，如果没有用EXEC SQL define name创建name就处理接下来的行。

EXEC SQL else;

为一个由EXEC SQL ifdef name或者EXEC SQL ifndef name引入的小节开始处理一个备选小节。

EXEC SQL elif name;

检查name，如果已经用EXEC SQL define name创建了name就开始处理一个备选小节。

EXEC SQL endif;

结束一个备选小节。

例子：

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

36.10. 处理嵌入式 SQL 程序

现在你已经对如何构造嵌入式 SQL C 程序有所了解了，你可能希望知道如何编译它们。在编译之前，你需要让该文件通过嵌入式SQL C预处理器，它会把你用到的SQL转换成特殊的函数调用。在编译之后，你必须链接一个包含所需函数的特殊库。这些函数从参数中取得信息、使用libpq执行SQL命令并且把结果放在指定的参数中用来输出。

该预处理器程序被称作ecpg并且被包括在一个正常的PostgreSQL安装中。嵌入式 SQL 程序通常带有扩展名.pgc。如果你有一个程序文件prog1.pgc，你可以调用下面的命令对它进行预处理：

```
ecpg prog1.pgc
```

这将创建一个文件prog1.c。如果你的输入文件不遵循建议的命名模式，你可以用-o选项显式地指定输出文件。

预处理过的文件可以被正常地编译，例如：

```
cc -c prog1.c
```

产生的 C 源文件从PostgreSQL安装中包括头文件，因此如果你把PostgreSQL安装在一个不被默认搜索的位置，你必须在编译命令行中增加一个选项（例如-I/usr/local/pgsql/include）。

要链接一个嵌入式 SQL 程序，你需要包括libecpg库，像这样：

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

再次，你可能不得不在命令行中增加类似-L/usr/local/pgsql/lib的选项。

你可以使用pg_config 或者pkg-config 加上包名libecpg来得到你的安装路径。

如果你使用make来管理一个大工程的构建过程，把下面的隐式规则包括在你的 makefile 中将会很方便：

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

ecpg命令的完整语法可见ecpg。

ecpg库默认是线程安全的。不过，你可能需要使用一些线程命令行选项来编译你的客户端代码。

36.11. 库函数

libecpg库主要包含用于实现嵌入式 SQL 命令所表达功能的“隐藏”函数。但是也有一些可以被直接调用的函数。但是注意这会让你的代码不可移植。

- 如果调用时第一个参数非零，ECPGdebug(int on, FILE *stream)会打开调试日志。调试日志在流上完成。该日志包含所有插入了输入变量的SQL语句，以及来自于PostgreSQL服务器的结果。在你的SQL语句中查找错误时这会非常有用。

> **注意**
>
> 在 Windows 上，如果ecpg库和应用使用不同标志编译的，这个函数调用将会是应用崩溃，因为FILE指针的内部表达不同。特别地，库和使用库的应用应该使用相同的多线程/单线程、发行/调试以及静态/动态标志。

- ECPGget_PGconn(const char *connection_name) 返回由给定名称标识的库数据库连接句柄。如果connection_name被设置为NULL，当前连接句柄将被返回。如果无法定位到连接句柄，该函数返回NULL。如果需要，返回的连接句柄可以被用来调用任何其他来自于libpq的函数。

> **注意**
>
> 直接使用libpq例程来操纵ecpg中建立的数据库连接句柄是一种糟糕的做法。

- ECPGtransactionStatus(const char *connection_name)返回由connection_name标识的给定连接的当前事务状态。关于返回的状态代码请参考第 34.2 和 libpq 的PQtransactionStatus()。

- 如果你连接到了一个数据库，ECPGstatus(int lineno, const char* connection_name)会返回真；否则返回假。 如果使用的是一个单一连接，connection_name可以为NULL。

36.12. 大对象

ECPG 并不直接支持大对象，在调用ECPGget_PGconn()函数获得所需的PGconn对象后，ECPG 应用能通过 libpq 大对象函数操纵大对象（不过，对ECPGget_PGconn()函数的使用以及直接接触PGconn对象都必须非常小心，并且最好不要与其他 ECPG 数据库访问调用混合在一起）。

更多关于ECPGget_PGconn()的细节可见第 36.11 节大对象函数接口的相关信息可见第 35 章

大对象函数必须在一个事务块中被调用，因此当自动提交关闭时，必须显式地发出BEGIN命令。

例 36.2给出了一个例子程序，它展示了在一个 ECPG 应用中如何创建、写入和读取一个大对象。

例 36.2. 访问大对象的 ECPG 程序

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* 创建 */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));
```

```
    printf("loid = %d\n", loid);

    /* 写入测试 */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* 读取测试 */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* 检查 */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* 清理 */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.13. C++ 应用

ECPG 对于 C++ 应用提供了有限的支持。这一节描述了一些忠告。

ecpg预处理器采用一个用 C（或者类似 C 的东西）和嵌入式 SQL 命令编写的输入文件，把嵌入式 SQL 命令转换成 C 语言块，并且最终产生一个.c文件。在 C++ 下使用时，因此它们应该能在 C++ 中无缝地使用。

不过，通常ecpg预处理器只理解 C，它无法处理 C++ 语言的特殊语法和保留词。因此，一些写在 C++ 应用代码中的使用了 C++ 特定复杂特性的嵌入式 SQL 代码可能无法被正确地预处理或者无法按预期工作。

使用 C++ 应用中嵌入式 SQL 代码的安全方法是把 ECPG 调用隐藏在一个 C 模块中，C++ 应用代码会调用它来访问数据库，还要把它和剩余的 C++ 代码链接起来。详见第 36.13.2 节

36.13.1. 主变量的可见范围

ecpg预处理器能理解 C 中变量的可见范围。在 C 语言中，这是相当简单的，因为变量的可见范围是基于它们的代码块的。不过在 C++ 中，引用类成员变量的代码块是不同于定义它的代码块的，因此ecpg预处理器将无法理解类成员变量的可见范围。

例如，在下面的情况中，ecpg预处理器无法为test方法中的变量dbname找到任何生命，因此将发生一个错误。

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

这段代码将导致一个这样的错误：

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

为了避免这种可见性问题，可以修改test方法来把一个本地变量用作中间存储。但是这种方法只是一种比较差的变通方案，因为它让代码变得丑陋并且降低了性能。

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;
```

```
    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

36.13.2. 使用外部 C 模块的 C++ 应用开发

如果你理解了 C++ 中ecpg预处理器的这些技术限制，你可能已经知道在链接阶段把 C 对象和 C++ 对象链接起来让 C++ 应用能使用 ECPG 特性比直接在 C++ 代码中写一些嵌入式 SQL 命令要更好。这一节用一个简单的例子描述了一种将嵌入式 SQL 命令从 C++ 应用代码中独立出去的方法。在这个例子中，应用由 C++ 实现，而 C 和 ECPG 被用来连接到 PostgreSQL 服务器。

需要创建三种文件：一个 C 文件（*.pgc）、一个头文件和一个 C++ 文件：

test_mod.pgc

一个执行嵌入在 C 中的 SQL 命令的子例程模块。它将被预处理器转换成test_mod.c。

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

test_mod.h

包含 C 模块（test_mod.pgc）中函数定义的头文件。它会被test_cpp.cpp包括。这个文件必须在声明周围有一个extern "C"块，因为它将被链接到 C++ 模块。

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
    void db_connect();
    void db_test();
    void db_disconnect();

    #ifdef __cplusplus
    }
    #endif
```

test_cpp.cpp

应用的主代码，包括main例程以及这个例子中的一个 C++ 类。

```
    #include "test_mod.h"

    class TestCpp
    {
      public:
        TestCpp();
        void test();
        ~TestCpp();
    };

    TestCpp::TestCpp()
    {
        db_connect();
    }

    void
    TestCpp::test()
    {
        db_test();
    }

    TestCpp::~TestCpp()
    {
        db_disconnect();
    }

    int
    main(void)
    {
        TestCpp *t = new TestCpp();

        t->test();
        return 0;
    }
```

要构建该应用，按以下步骤处理。通过运行ecpg将test_mod.pgc转换为test_mod.c，并且用 C 编译器将test_mod.c编译成test_mod.o：

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

接着，用 C++ 编译器把test_cpp.cpp编译成test_cpp.o：

```
c++ -c test_cpp.cpp -o test_cpp.o
```

最后，使用 C++ 编译器链接这些对象文件（test_cpp.o和test_mod.o）成为一个可执行文件：

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

36.14. 嵌入式 SQL 命令

这一节描述嵌入式 SQL 所有特定的 SQL 命令。SQL 命令中的 SQL 命令也能被用于嵌入式 SQL，如果有例外会特别说明。

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — 分配一个 SQL 描述符区域

大纲

```
ALLOCATE DESCRIPTOR name
```

描述

ALLOCATE DESCRIPTOR分配一个新的命名 SQL 描述符区域，它能被用来在 PostgreSQL 服务器和主程序之间交换数据。

以后可以使用DEALLOCATE DESCRIPTOR命令释放描述符区域。

参数

name

SQL 描述符的名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准中说明了ALLOCATE DESCRIPTOR。

参见

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — 建立一个数据库连接

大纲

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
CONNECT TO DEFAULT
CONNECT connection_user
DATABASE connection_target
```

描述

CONNECT命令在客户端和 PostgreSQL 服务器之间建立一个连接。

参数

connection_target

connection_target以下列形式之一指定连接的目标服务器。

[database_name] [@host] [:port]

通过 TCP/IP 连接

unix:postgresql://host [:port] / [database_name] [?connection_option]

通过 Unix 域套接字

tcp:postgresql://host [:port] / [database_name] [?connection_option]

通过 TCP/IP 连接

SQL string constant

包含上述形式之一的一个值

host variable

类型char[]或VARCHAR[]的主变量，它包含上述形式之一的一个值

connection_object

用于该连接的一个可选标识符，这样可以在其他命令中引用它。这可以是一个 SQL 标识符或者一个主变量。

connection_user

用于数据库连接的用户名。

使用 user_name/password、 user_name IDENTIFIED BY password或者 user_name USING password之一，这个参数也能指定用户名和口令.

用户名和口令可以是 SQL 标识符、字符串常量或者主变量。

DEFAULT

按 libpq 的定义使用所有默认连接参数。

例子

这里是一些指定连接参数的变体：

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER
 connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER
 connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser
 IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser
 IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER
 connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser
 IDENTIFIED BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING
 "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14
 USER connectuser;
```

这里是一个展示使用主变量指定连接参数的例子程序：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* 数据库名 */
    char *user       = "testuser";  /* 连接用户名 */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* 连接字符串 */
    char ver[256];                  /* 存储版本字符串的缓冲区 */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;
```

```
    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

兼容性

SQL 标准中说明了CONNECT，但是连接参数的格式是与实现相关的。

参见

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — 释放一个 SQL 描述符区域

大纲

```
DEALLOCATE DESCRIPTOR name
```

描述

DEALLOCATE DESCRIPTOR释放一个命名的 SQL 描述符区域。

参数

name

要被释放的描述符的名称。它是大小写敏感的。这可以是一个 SQL 标识符或者一个主变量。

例子

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准说明了DEALLOCATE DESCRIPTOR。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH
 | WITHOUT } HOLD ] FOR prepared_name
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH
 | WITHOUT } HOLD ] FOR query
```

描述

DECLARE声明一个游标用来在一个预备语句的结果集上迭代。这个命令与直接的 SQL 命令DECLARE在语义上有一点点区别：后者会执行一个查询并且准备结果集用于检索，而这个嵌入式 SQL 命令仅仅声明一个名称作为“循环变量”用于在一个查询的结果集上迭代，实际的执行在游标被OPEN命令打开时才发生。

参数

cursor_name

一个游标名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

prepared_name

一个预备查询的名称，可以是一个 SQL 标识符或者一个主变量。

query

一个提供游标要返回的行的SELECT或者VALUES命令。

游标选项的含义请见DECLARE。

例子

为一个查询声明一个游标的例子：

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

为一个预备语句声明一个游标的例子：

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

兼容性

SQL 标准中说明了DECLARE。

参见

OPEN, CLOSE, DECLARE

DESCRIBE

DESCRIBE — 得到有关一个预备语句或结果集的信息

大纲

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqlda_name
```

描述

DESCRIBE检索被一个预备语句所含的结果列的元信息，而不会实际取得一行。

参数

prepared_name

一个预备语句的名称。这可以是一个 SQL 标识符或者一个主变量。

descriptor_name

一个描述符名称。它是大小写敏感的。它可以是一个 SQL 标识符或者一个主变量。

sqlda_name

一个 SQLDA 变量的名称。

例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准中说明了DESCRIBE。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — 终止一个数据库连接

大纲

```
DISCONNECT connection_name
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

描述

DISCONNECT关闭一个（或者所有）到数据库的连接。

参数

connection_name

一个由CONNECT命令建立的数据库连接名称。

CURRENT

关闭“当前的”连接，它可以是最近打开的连接或者是由SET CONNECTION命令设置的连接。如果没有参数被传给DISCONNECT命令，这将是默认值。

DEFAULT

关闭默认连接。

ALL

关闭所有打开的连接。

例子

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* 关闭 con3          */
    EXEC SQL DISCONNECT DEFAULT;  /* 关闭 DEFAULT       */
    EXEC SQL DISCONNECT ALL;      /* 关闭 con2 以及 con1 */

    return 0;
}
```

兼容性

SQL 标准中说明了DISCONNECT。

参见

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — 动态地准备和执行一个语句

大纲

```
EXECUTE IMMEDIATE string
```

描述

EXECUTE IMMEDIATE立刻预备并且执行一个动态指定的 SQL 语句，不检索结果行。

参数

string

包含要被执行的 SQL 语句的一个 C 字符串或者是一个主变量。

例子

这里是一个用EXECUTE IMMEDIATE和一个名为command的主变量执行INSERT语句的例子：

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

兼容性

SQL 标准中说明了EXECUTE IMMEDIATE。

GET DESCRIPTOR

GET DESCRIPTOR — 从一个 SQL 描述符区域得到信息

大纲

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item
 [, ... ]
```

描述

GET DESCRIPTOR从一个 SQL 描述符区域检索关于一个查询结果集的信息并且把它存储在主变量中。在使用这个命令把信息传输到主语言变量之前，一个描述符区域通常是用FETCH或SELECT填充的。

这个命令有两种形式：第一种形式检索描述符的“头部”项，它适用于全面地查看结果集。一种例子是行计数。第二种形式要求列号作为附加参数，它检索有关一个特定列的信息。其例子是查看列名和实际列值。

参数

descriptor_name

一个描述符名称。

descriptor_header_item

一个标识要检索哪一个头部信息项的记号。当前只支持用于得到结果集中列数的COUNT。

column_number

要检索其信息的列号。计数从 1 开始。

descriptor_item

一个标识要检索哪一个有关一列信息的项的记号。被支持的项可见第 36.7.1 节

cvariable

接收从描述符区域检索到的数据的主变量。

例子

检索一个结果集中列数的例子：

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

检索第一列中数据长度的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
 RETURNED_OCTET_LENGTH;
```

把第二列的数据体检索成一个字符串的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

这里是执行SELECT current_database();并且显示列数、列数据长度和列数据的完整过程的例子：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* 描述、打开一个游标，并且分配一个描述符给该游标  */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* 得到全部列的数量 */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                  = %d\n", d_count);

    /* 得到一个返回列的长度 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
 RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* 将返回的列取出成一个字符串 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                   = %s\n", d_data);

    /* 关闭 */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

当该例子被执行时，结果看起来是：

```
d_count                  = 1
d_returned_octet_length = 6
d_data                   = testdb
```

兼容性

SQL 标准中说明了GET DESCRIPTOR。

参见

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — 打开一个动态游标

大纲

```
OPEN cursor_name
OPEN cursor_name USING value [, ... ]
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

描述

OPEN打开一个游标并且可选地绑定实际值到游标声明中的占位符。该游标必须之前用DECLARE命令声明。OPEN的执行会导致查询开始在服务器上执行。

参数

cursor_name

要被打开的游标的名称。这可以是一个 SQL 标识符或者一个主变量。

value

要被绑定到游标中一个占位符的值。这可以是一个 SQL 常量、一个主变量或者一个带有指示符的主变量。

descriptor_name

包含要绑定到游标中占位符的值的描述符的名称。这可以是一个 SQL 标识符或者一个主变量。

例子

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

兼容性

SQL 标准中说明了OPEN。

参见

DECLARE, CLOSE

PREPARE

PREPARE — 准备一个语句用于执行

大纲

```
PREPARE name FROM string
```

描述

PREPARE将一个作为字符串动态指定的语句准备好执行。这不同于直接的 SQL 语句PREPARE（也可以用于嵌入式程序）。EXECUTE命令被用来执行两种类型的预备语句。

参数

prepared_name

预备查询的一个标识符。

string

包含一个可预备语句的一个 C 字符串或一个主变量，可预备语句是 SELECT、INSERT、UPDATE 或者 DELETE 之一。

例子

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

兼容性

SQL 标准中说明了PREPARE。

参见

EXECUTE

SET AUTOCOMMIT

SET AUTOCOMMIT — 设置当前会话的自动提交行为

大纲

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

描述

SET AUTOCOMMIT设置当前数据库会话的自动提交行为。默认情况下，嵌入式 SQL 程序不在自动提交模式中，因此需要显式地发出COMMIT。这个命令可以把会话改成自动提交模式，这样每一个单独的语句都会被隐式提交。

兼容性

SET AUTOCOMMIT是 PostgreSQL ECPG 的扩展。

SET CONNECTION

SET CONNECTION — 选择一个数据库连接

大纲

```
SET CONNECTION [ TO | = ] connection_name
```

描述

SET CONNECTION设置“当前的”数据库连接，除非被覆盖，所有命令都会使用这个连接。

参数

connection_name

一个由CONNECT命令建立的数据库连接名。

DEFAULT

设置该连接为默认连接。

例子

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

兼容性

SQL 标准中说明了SET CONNECTION。

参见

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — 在一个 SQL 描述符区域中设置信息

大纲

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

描述

SET DESCRIPTOR用值填充一个 SQL 描述符区域。然后该描述符区域通常会被用来在一个预备查询执行中绑定参数。

这个命令由两种形式：第一种形式适用于描述符“头部”，它独立于特定的数据。第二种形式为由数字标识的特定数据赋值。

参数

descriptor_name

一个描述符名称。

descriptor_header_item

一个标识要设置哪个头部信息项的记号。当前只有设置描述符项数量的COUNT被支持。

number

要设置的描述符项的编号。计数从 1 开始。

descriptor_item

一个标识在描述符中要设置哪个信息项的记号。受支持的项的列表可见第 36.7.1 节

value

一个要存储在描述符项中的值。这可以是一个 SQL 标识符或者一个主变量。

例子

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

兼容性

SQL 标准中说明了SET DESCRIPTOR。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — 定义一种新数据类型

大纲

```
TYPE type_name IS ctype
```

描述

TYPE命令定义一个新的 C 类型。它等效于把一个typedef放在声明节中。

只有使用选项-c运行ecpg时才能识别这个命令。

参数

type_name

新类型的名称。这必须是一个合法的 C 类型名。

ctype

一个 C 类型说明。

例子

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

这里是一个使用EXEC SQL TYPE的例子程序：

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
```

```
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

这个程序的输出看起来像：

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

兼容性

TYPE命令是一种 PostgreSQL 扩展。

VAR

VAR — 定义一个变量

大纲

```
VAR varname IS ctype
```

描述

VAR命令分配一个新的 C 数据类型给一个主变量。主变量必须之前在一个声明节中声明过。

参数

varname

一个 C 变量名。

ctype

一个 C 类型说明。

例子

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

兼容性

VAR命令是一个 PostgreSQL 扩展。

WHENEVER

WHENEVER — 指定一个要在一个 SQL 语句导致发生一个特定类别的情况时要采取的动作

大纲

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

描述

定义一个行为，它会在 SQL 执行结果的特殊情况（行未找到、SQL 警告或错误）中被调用。

参数

参数描述见第 36.8.1 节

例子

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

一个典型的应用是使用WHENEVER NOT FOUND BREAK来处理通过结果集的循环：

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* 当到达结果集末尾时，跳出循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;
```

```
    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

兼容性

SQL 标准中说明了WHENEVER，但是大部分动作是 PostgreSQL 扩展。

36.15. Informix兼容模式

ecpg可以运行在一种所谓的Informix 兼容模式中。如果这种模式被激活，它的行为就好像它是一个用于Informix E/SQL 的Informix预编译器。一般而言，这将允许你使用美元符号替代EXEC SQL来引入嵌入式 SQL 命令：

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **注意**
>
> 在$之间不能有任何空白以及下列之一的预处理器指令：include、define、ifdef等。否则，预处理器将把记号解析成一个主变量。

有两种兼容性模式：INFORMIX、INFORMIX_SE

在链接使用这种兼容性模式的程序时，要记得链接上和 ECPG 一起发布的libcompat。

除了之前解释过的语法糖，Informix兼容性模式从 E/SQL 中移植了一些用于输入、输出和数据转换的函数以及嵌入式 SQL 语句到 ECPG 中。

Informix兼容性模式与 ECPG 的 pgtypeslib 库紧密连接。pgtypeslib 把 SQL 数据类型映射到 C 主程序中的数据类型并且大部分Informix兼容性模式的附加函数允许我们在那些 C 主程序类型上操作。不过注意兼容性的范围被有所限制。它并不是想尝试复制Informix的行为。它允许你做或多或少的相同操作并且给你具有相同名称和相同基本行为的函数，但是此刻如果你使用Informix，其中并没有唾手可得的替代品。此外，一些数据类型也不同。例如，PostgreSQL的日期时间和区间类型不理解范围（例如YEAR TO MINUTE），因此你也无法在 ECPG 中找到支持。

36.15.1. 附加类型

用于存储右切边字符串数据的 Informix-特殊的 "string" 伪类型现在在 Informix 模式中不用typedef就能支持。事实上，在 Informix 模式中，ECPG 拒绝处理包含typedef sometype string;的源文件。

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* 这个变量将包含切边过的数据 */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

36.15.2. 附加的/缺少的 嵌入式 SQL 语句

CLOSE DATABASE

这个语句关闭当前连接。事实上，这是 ECPG 的DISCONNECT CURRENT语句的同义词：

```
$CLOSE DATABASE;                /* 关闭当前连接 */
EXEC SQL CLOSE DATABASE;
```

FREE cursor_name

由于 ECPG 和 Informix ESQL/C 在工作方式上的区别（一个是纯语法转换而另一个依赖于底层的运行时库），在 ECPG 中没有FREE cursor_name语句。这是因为在 ECPG 中，DECLARE CURSOR不会翻译成一个运行时库中使用游标名的函数调用。这意味着在 ECPG 运行时库中不会有 SQL 游标的运行时登记，SQL 游标只登记在 PostgreSQL 服务器中。

FREE statement_name

FREE statement_name是DEALLOCATE PREPARE statement_name的同义词。

36.15.3. Informix-兼容的 SQLDA 描述符区域

Informix-兼容模式支持一种与第 36.7.2 节所述不同的结构。如下：

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
```

```
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

全局属性是：

sqld

SQLDA描述符中域的数量。

sqlvar

每一个域属性的指针。

desc_name

未使用，用零字节填充。

desc_occ

已分配结构的尺寸。

desc_next

如果结果集包含多于一个记录，这个域是下一个 SQLDA 结构的指针。

reserved

未使用的指针，包含 NULL。为 Informix-兼容性而保留。

对每一个域的属性如下，它们被存储在sqlvar数组中：

sqltype

域的类型。可以使用的常量定义在sqltypes.h中。

sqllen

域数据的长度。

sqldata

域数据的指针。该指针是char *类型，它所指向的数据是二进制个事。例子：

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

sqlind

NULL 指示符的指针。如果是由 DESCRIBE 或 FETCH 返回，那么它总是一个有效的指针。如果被用作EXECUTE ... USING sqlda;的输入，那么 NULL-指针值意味着这个域的值是非-NULL 的。否则必须正确地设置一个有效的指针和sqlitype。例子：

```
    if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
        printf("value is NULL\n");
```

sqlname

域的名称。以 0 终止的字符串。

sqlformat

在 Informix 中保留，是该域的PQfformat()的值。

sqlitype

NULL 指示符数据的类型。当从服务器返回数据时，它总是 SQLSMINT。当SQLDA被用于一个参数化查询时，数据要根据设置的类型对待。

sqlilen

NULL 指示符数据的长度。

sqlxid

该域的扩展类型，PQftype()的结果。

sqltypename
sqltypelen
sqlownerlen
sqlsourcetype
sqlownername
sqlsourceid
sqlflags
sqlreserved

未使用。

sqlilongdata

如果sqllen大于 32kB，它等于sqldata。

例子：

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* 这不需要在嵌入式 DECLARE SECTION 下 */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);
```

```
EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL OPEN mycursor;
EXEC SQL WHENEVER NOT FOUND GOTO out;

while (1)
{
  EXEC SQL FETCH mycursor USING sqlda;
}

EXEC SQL CLOSE mycursor;

free(sqlda); /* 主结构完全被 free()，sqlda 和 sqlda->sqlvar 在一个已分配区域中 */
```

更多信息可见sqlda.h头部和src/interfaces/ecpg/test/compat_informix/sqlda.pgc回归测试。

36.15.4. 附加函数

decadd

将两个decimal类型值相加。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

该函数接收第一个类型为 decimal 的操作数的指针（arg1）、第二个类型为 decimal 的操作数的指针（arg2）以及将包含和的 decimal 值的指针（sum）。成功时该函数返回 0。溢出时返回ECPG_INFORMIX_NUM_OVERFLOW，下溢时返回ECPG_INFORMIX_NUM_UNDERFLOW。其他失败会返回 -1 并且errno会被设置为相应的 pgtypeslib 中的errno编号。

deccmp

比较两个 decimal 变量。

```
int deccmp(decimal *arg1, decimal *arg2);
```

该函数接收第一个 decimal 值的指针（arg1）、第二个 decimal 值的指针（arg2）并且返回一个整数值说明哪一个值更大。

- 1，如果arg1指向的值大于var2指向的值
- -1，如果arg1指向的值小于var2指向的值
- 0，如果arg1指向的值与arg2指向的值相等

deccopy

拷贝一个 decimal 值。

```
void deccopy(decimal *src, decimal *target);
```

该函数接收要拷贝的 decimal 值的指针作为第一个参数（src）以及一个类型为 decimal 的目标结构的指针作为第二个参数（target）。

deccvasc

把一个值从 ASCII 表达转换成一个 decimal 类型。

```
int deccvasc(char *cp, int len, decimal *np);
```

该函数接收一个包含要转换的数字的字符串表达的字符串指针（cp）及其长度（len）。np是一个用来保存操作结果的 decimal 值的指针。

例如，可用的格式有： -2、 .794、 +3.44、 592.49E07或者 -32.84e-4。

成功时该函数返回 0。如果发生溢出或者下溢，分别返回ECPG_INFORMIX_NUM_OVERFLOW或者 ECPG_INFORMIX_NUM_UNDERFLOW。如果 ASCII 表达无法被解析，将返回ECPG_INFORMIX_BAD_NUMERIC。如果解析指数时发生问题则返回ECPG_INFORMIX_BAD_EXPONENT。

deccvdbl

将一个 double 值转换成一个 decimal 值。

```
int deccvdbl(double dbl, decimal *np);
```

该函数接收要被转换的 double 变量作为第一个参数（dbl）。该函数接收一个 decimal 变量的指针作为第二个参数（np），它被用来保存操作的结果。

该函数在成功时返回 0，在转换失败时返回一个负值。

deccvint

将一个 int 值转换成 decimal 值。

```
int deccvint(int in, decimal *np);
```

该函数接收要被转换的 int 变量作为第一个参数（in）。该函数接收一个 decimal 变量的指针作为第二个参数（np），它被用来保存操作的结果。

该函数在成功时返回 0，在转换失败时返回一个负值。

deccvlong

将一个 long 值转换成 decimal 值。

```
int deccvlong(long lng, decimal *np);
```

该函数接收要被转换的 long 变量作为第一个参数（lng）。该函数接收一个 decimal 变量的指针作为第二个参数（np），它被用来保存操作的结果。

该函数在成功时返回 0，在转换失败时返回一个负值。

decdiv

用两个 decimal 类型的变量做除法。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个（n1）和第二个（n2）操作数并且结算n1/n2。result是一个指向保存操作结果的变量的指针。

成功时返回 0，如果除法失败则返回一个负值。如果发生溢出或下溢，该函数分别返回ECPG_INFORMIX_NUM_OVERFLOW或者 ECPG_INFORMIX_NUM_UNDERFLOW。如果发现一次除零尝试，该函数返回 ECPG_INFORMIX_DIVIDE_ZERO。

decmul

将两个 decimal 值相乘。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个（n1）和第二个（n2）操作数并且结算n1*n2。result是一个指向保存操作结果的变量的指针。

成功时返回 0，如果乘法失败则返回一个负值。如果发生溢出或下溢，该函数分别返回ECPG_INFORMIX_NUM_OVERFLOW或者 ECPG_INFORMIX_NUM_UNDERFLOW。

decsub

从一个 decimal 值中减去另一个。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个（n1）和第二个（n2）操作数并且结算n1-n2。result是一个指向保存操作结果的变量的指针。

成功时返回 0，如果减法失败则返回一个负值。如果发生溢出或下溢，该函数分别返回ECPG_INFORMIX_NUM_OVERFLOW或者 ECPG_INFORMIX_NUM_UNDERFLOW。

dectoasc

将一个 decimal 变量转换成它的 ASCII 表达放在一个 C char* 字符串中。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

该函数接收一个要被转换成文本表达的 decimal 类型变量的指针（np）。cp是应保存操作结果的缓冲区。参数right指定小数点右边应该有多少位保留在输出中。结果将被圆整到所指定数量的十进制位。将right设置为 -1 表示输出中应该包括所有可用的十进制位。如果输出缓冲区的长度（由len指定）不足以保存包含拖尾零字节的文本表达，结果中将只保存一个单一的*字符并且返回 -1。

如果缓冲区cp太小该函数返回 -1；如果内存耗尽，则返回ECPG_INFORMIX_OUT_OF_MEMORY。

dectodbl

将一个 decimal 类型变量转换成一个 double 类型变量。

```
int dectodbl(decimal *np, double *dblp);
```

该函数接收一个要转换的 decimal 值的指针（np）以及一个保存操作结果的 double 变量的指针（dblp）。

该函数在成功时返回 0，在转换失败时返回一个负值。

dectoint

将一个 decimal 类型变量转换成一个整数类型变量。

```
int dectoint(decimal *np, int *ip);
```

该函数接收一个要转换的 decimal 值的指针（np）以及一个保存操作结果的整数变量的指针（ip）。

该函数在成功时返回 0，在转换失败时返回一个负值。如果发生溢出，会返回ECPG_INFORMIX_NUM_OVERFLOW。

注意 ECPG 实现与Informix实现不同。Informix限制一个整数的范围是从 -32767 到 32767，而 ECPG 实现中的限制取决于架构（-INT_MAX .. INT_MAX）。

dectolong

将一个 decimal 类型变量转换成一个长整型变量。

```
int dectolong(decimal *np, long *lngp);
```

该函数接收一个要转换的 decimal 值的指针（np）以及一个保存操作结果的长整型变量的指针（lngp）。

该函数在成功时返回 0，在转换失败时返回一个负值。如果发生溢出，会返回ECPG_INFORMIX_NUM_OVERFLOW。

注意 ECPG 实现与Informix实现不同。Informix限制一个整数的范围是从 -2,147,483,647 到 2,147,483,647，而 ECPG 实现中的限制取决于架构（-LONG_MAX .. LONG_MAX）。

rdatestr

将一个日期转换成一个 C char* 字符串。

```
int rdatestr(date d, char *str);
```

该函数接收两个参数，第一个是要转换的日期（d），第二个是目标字符串的指针。输出格式总是yyyy-mm-dd，因此你需要为该字符串分配至少 11 个字节（包括零字节终止符）。

成功时该函数返回 0，如果发生错误则返回一个负值。

注意 ECPG 实现与Informix实现不同。在Informix中，该格式可能受到环境变量设置的影响。而在 ECPG 中，你不能改变输出格式。

rstrdate

解析一个日期的文本表达。

```
int rstrdate(char *str, date *d);
```

该函数接收要转换的日期的文本表达（str）以及一个日期类型变量的指针（d）。这个函数不允许你指定一个格式掩码。它使用Informix的默认格式掩码mm/dd/yyyy。在内部，这个函数用rdefmtdate的方式实现。因此，rstrdate不会更快，并且如果可以选择，你应该选用允许你显式指定格式掩码的rdefmtdate。

该返回与rdefmtdate相同的值。

rtoday

得到当前日期。

```
void rtoday(date *d);
```

该函数接收一个日期变量的指针（d），它会把该变量设置为当前日期。

在内部这个函数使用PGTYPESdate_today函数。

rjulmdy

从一个日期类型变量中抽取日、月、年的值。

```
int rjulmdy(date d, short mdy[3]);
```

该函数接收日期d和由 3 个短整型值构成的数组的指针mdy。该变量名指定了顺序：mdy[0]将被设置为包含月的编号，mdy[1]将被设置为日的值，而mdy[2]将包含年。

当前该函数总是返回 0。

在内部该函数使用PGTYPESdate_julmdy函数。

rdefmtdate

使用一个格式掩码把一个字符串转换成一个日期类型的值。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

该函数接收一个用于保存操作结果的日期值的指针（d）、要用来解析日期的格式掩码（fmt）以及包含日期文本表达的 C char* 字符串（str）。该文本表达应该匹配格式掩码。不过，你不需要具有从该字符串到格式掩码的 一一映射。该函数将分析顺序并且寻找表示年的位置的文字yy或yyyy、表示月的位置的mm以及表示日的位置的dd。

该函数返回下列值：

- 0 - 该函数成功终止。
- ECPG_INFORMIX_ENOSHORTDATE - 该日期不包含日、月、年之间的定界符。在这种情况下，输入字符串必须是正好 6 个或 8 个字节，但实际上却不是。
- ECPG_INFORMIX_ENOTDMY - 格式字符串没有正确地指示年、月、日的顺序。
- ECPG_INFORMIX_BAD_DAY - 输入字符串不含一个合法的日。
- ECPG_INFORMIX_BAD_MONTH - 输入字符串不含一个合法的月。
- ECPG_INFORMIX_BAD_YEAR - 输入字符串不含一个合法的年。

在内部这个函数被实现为使用PGTYPESdate_defmt_asc函数。示例输入表可以在那里找到。

rfmtdate

使用一个格式掩码将一个日期类型变量转换成它的文本表达。

```
int rfmtdate(date d, char *fmt, char *str);
```

该函数接收要转换的日期（d）、格式掩码（fmt）以及将保存日期的文本表达的字符串（str）。

成功时该函数返回 0，如果发生错误则返回一个负值。

在内部这个函数使用PGTYPESdate_fmt_asc函数，例子请参考该函数。

rmdyjul

从由 3 个短整型组成的数组创建一个日期值，它指定了该日期的日、月、年。

```
int rmdyjul(short mdy[3], date *d);
```

该函数接收一个由 3 个短整型构成的数组（mdy）以及一个用来保存操作结构的日期类型变量的指针。

当前该函数总是返回 0。

在内部这个函数被实现为使用PGTYPESdate_mdyjul。

rdayofweek

为一个日期值返回一个表示它是星期几的数字。

```
int rdayofweek(date d);
```

该函数接收日期变量d作为它的唯一参数并且返回一个整数指示这一天是星期几。

- 0 - 周日
- 1 - 周一
- 2 - 周二
- 3 - 周三
- 4 - 周四
- 5 - 周五
- 6 - 周六

在内部这个函数被实现为使用函数PGTYPESdate_dayofweek。

dtcurrent

检索当前的时间戳。

```
void dtcurrent(timestamp *ts);
```

该函数检索当前时间戳并且把它保存在ts指向的时间戳变量中。

dtcvasc

把一个时间戳从它的文本表达解析到一个时间戳变量中。

```
int dtcvasc(char *str, timestamp *ts);
```

该函数接收要解析的字符串（str）以及一个指向保存操作结果的时间戳变量的指针（ts）。

成功时该函数返回 0，如果发生错误则返回一个负值。

在内部这个函数使用PGTYPEStimestamp_from_asc函数。一个输入示例的表格可以参考该函数的文档。

dtcvfmtasc

使用一个格式掩码把一个时间戳从它的文本表达解析到一个时间戳变量中。

dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)

该函数接收要解析的字符串（inbuf）、要使用的格式掩码（fmtstr）以及一个指向保存操作结果的时间戳变量的指针（dtvalue）。

这个函数通过PGTYPEStimestamp_defmt_asc函数实现。可以使用的格式说明符的列表可以参考该函数的文档。

成功时该函数返回 0，如果发生错误则返回一个负值。

dtsub

从一个时间戳中减去另一个并且返回一个区间类型变量。

int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);

该函数将从ts1指向的时间戳变量中减去ts2指向的时间戳变量，并且将把结果存储在iv指向的区间变量中。

成功时该函数返回 0，如果发生错误则返回一个负值。

dttoasc

将一个时间戳变量转换成一个 C char* 字符串。

int dttoasc(timestamp *ts, char *output);

该函数接收一个要转换的时间戳变量的指针（ts）以及用于保存操作结果的字符串（output）。它根据 SQL 标准把ts转换成它的文本表达，形式为YYYY-MM-DD HH:MM:SS。

成功时该函数返回 0，如果发生错误则返回一个负值。

dttofmtasc

使用一个格式掩码将一个时间戳变量转换成一个 C char*。

int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);

该函数接收一个要转换的时间戳的指针（ts）、一个输出缓冲区的指针（output）、已经为输出缓冲区分配的最大长度（str_len）以及用于转换的格式掩码（fmtstr）。

成功时该函数返回 0，如果发生错误则返回一个负值。

在内部，这个函数使用PGTYPEStimestamp_fmt_asc函数。可以使用的格式说明符的列表可以参考该函数的文档。

intoasc

将一个区间变量转换成一个 C char* 字符串。

int intoasc(interval *i, char *str);

该函数接收一个要转换的区间变量的指针（i）以及要保持该操作结果的字符串（str）。它根据 SQL 标准把i转换成它的文本表达，形式为YYYY-MM-DD HH:MM:SS。

成功时该函数返回 0，如果发生错误则返回一个负值。

rfmtlong

用一个格式掩码将一个长整型值转换成它的文本表达。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

该函数接收长整型值lng_val、格式掩码fmt以及输出缓冲区的指针outbuf。它根据格式掩码将长整型值转换成文本表达。

格式掩码可以由下列格式说明字符构成：

- * （星） - 如果这个位置可以为空白，否则用一个星号填充。
- & （花号） -如果这个位置可以为空白，否则用一个零填充。
- # - 把前导零转变成空白。
- < - 左对齐字符串中的数字。
- , （逗号） - 将有四个或者更多数位的数字份组成用逗号分隔的 3 数位组。
- . （点） - 这个字符分隔数字的整数部分和小数部分。
- - （负） - 如果该数字是一个负值则负号会出现。
- + （加） - 如果该数字是一个正值则加号会出现。
- (- 这会替换负数前面的负号。负号将不会出现。
-) - 这个字符替换负号并且被打印在负值的后面。
- $ - 货币符号。

rupshift

把一个字符串转换成大写形式。

```
void rupshift(char *str);
```

该函数接收一个字符串的指针并且把每一个小写形式的字符变成大写形式。

byleng

返回一个字符串的字符数，其中不含拖尾的空白。

```
int byleng(char *str, int len);
```

该函数期待一个定长字符串作为它的第一个参数（str）并且把它的长度作为第二个参数（len）。该函数会返回有效字符的数量，也就是字符串不含拖尾空白的长度。

ldchar

复制一个定长字符串到一个空终止的字符串。

```
void ldchar(char *src, int len, char *dest);
```

该函数接收要被复制的定长字符串（src）、它的长度（len）以及目标内存的指针（dest）。注意你需要为dest指向的字符串保留至少len+1个字节。该函数复制至多len个字节到新的位置（如果源字符串有拖尾的空格）并且增加空终止符。

rgetmsg

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

这个函数存在，但是目前还没有实现！

rtypalign

```
int rtypalign(int offset, int type);
```

这个函数存在，但是目前还没有实现！

rtypmsize

```
int rtypmsize(int type, int len);
```

这个函数存在，但是目前还没有实现！

rtypwidth

```
int rtypwidth(int sqltype, int sqllen);
```

这个函数存在，但是目前还没有实现！

rsetnull

设置一个变量为 NULL。

```
int rsetnull(int t, char *ptr);
```

该函数接收一个表示变量类型的整数以及一个被造型成 C char* 指针的变量本身的指针。

存在下列类型：

- CCHARTYPE - 用于类型char或者char*的一个变量
- CSHORTTYPE - 用于类型short int的一个变量
- CINTTYPE - 用于类型int的一个变量
- CBOOLTYPE - 用于类型boolean的一个变量
- CFLOATTYPE - 用于类型float的一个变量
- CLONGTYPE - 用于类型long的一个变量
- CDOUBLETYPE - 用于类型double的一个变量
- CDECIMALTYPE - 用于类型decimal的一个变量
- CDATETYPE - 用于类型date的一个变量
- CDTIMETYPE - 用于类型timestamp的一个变量

这里是一个调用这个函数的例子：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

risnull

测试一个变量是否为 NULL。

```
int risnull(int t, char *ptr);
```

该函数接收要测试的变量的类型（t）以及一个指向该变量的指针（ptr）。注意后者需要被造型为一个 char*。可能的变量类型请见函数rsetnull。

这里是一个如何使用这个函数的例子：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

36.15.5. 额外的常量

注意这里所有的常量都描述错误并且它们都被定义为表示负值。在每一种不同常量的描述中，你还可以找到在当前实现中该常量表示的值。不过你不应该依赖于这个数字。但是你可以相信所有的这些常量都是被定义为表示负值。

ECPG_INFORMIX_NUM_OVERFLOW

如果在一次计算中发生了溢出，函数会返回这个值。在内部它被定义为 -1200（Informix定义）。

ECPG_INFORMIX_NUM_UNDERFLOW

如果在一次计算中发生了下溢，函数会返回这个值。在内部它被定义为 -1201（Informix定义）。

ECPG_INFORMIX_DIVIDE_ZERO

如果发现尝试除零，函数会返回这个值。在内部它被定义为 -1202（Informix定义）。

ECPG_INFORMIX_BAD_YEAR

如果在解析一个日期时为年找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1204（Informix定义）。

ECPG_INFORMIX_BAD_MONTH

如果在解析一个日期时为月找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1205（Informix定义）。

ECPG_INFORMIX_BAD_DAY

如果在解析一个日期时为日找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1206（Informix定义）。

ECPG_INFORMIX_ENOSHORTDATE

如果一个解析例程需要一个短日期表示但是却没有得到正确长度的日期自如穿，函数会返回这个值。在内部它被定义为 -1209（Informix定义）。

ECPG_INFORMIX_DATE_CONVERT

如果在日期格式化时产生了一个错误，函数会返回这个值。在内部它被定义为 -1210（Informix定义）。

ECPG_INFORMIX_OUT_OF_MEMORY

如果在操作时内存被耗尽，函数会返回这个值。在内部它被定义为 -1211（Informix定义）。

ECPG_INFORMIX_ENOTDMY

如果一个解析例程被假定为得到一个格式掩码（如mmddyy）但是列出的域并不是全部正确，函数会返回这个值。在内部它被定义为 -1212（Informix定义）。

ECPG_INFORMIX_BAD_NUMERIC

如果一个解析例程因为一个numeric值的文本表达包含错误而不能解析它或者一个例程因为至少一个numeric变量非法而无法完成一次涉及numeric变量的计算，函数会返回这个值。在内部它被定义为 -1213（Informix定义）。

ECPG_INFORMIX_BAD_EXPONENT

如果一个解析例程不能解析一个指数，函数会返回这个值。在内部它被定义为 -1216（Informix定义）。

ECPG_INFORMIX_BAD_DATE

如果一个解析例程不能解析一个日期，函数会返回这个值。在内部它被定义为 -1218（Informix定义）。

ECPG_INFORMIX_EXTRA_CHARS

如果一个解析例程被传递了它不能解析的额外字符，函数会返回这个值。在内部它被定义为 -1264（Informix定义）。

36.16. 内部

这一节解释ECPG在内部如何工作。这些信息有时有助于用户理解如何使用ECPG。

ecpg写到输出的头四行是固定行。两行是注释，两行是与库接口必须的包括行。然后预处理器会从文件读取并且写输出。通常它会把所有东西回显在输出上。

当它看见一个EXEC SQL语句时，它会干预并且改变它。命令开始于EXEC SQL 并且结束于;。之间的任何东西都被视作一个SQL语句，并且会被解析进行变量替换。

当一个符号开始于一个冒号（:）时，变量替换会发生。有该名称的变量会被在之前声明于EXEC SQL DECLARE小节中的变量中搜索。

该库中最重要的函数是ECPGdo，它负责执行大部分命令。它采用可变数量的参数。可以很容易地增加到最多 50 个左右的参数，并且我们希望在任何平台上这都不会成为问题。

参数是：

一个行号

这是原始行的行号，只用于错误消息。

一个字符串

这是要被发出的SQL命令。它会被输入变量修改，即在编译时不知道但是要在命令中被输入的变量。其中变量应该去到包含?的字符串中。

输入变量

每一个输入参数导致十个参数被创建（见下文）。

ECPGt_EOIT

一个说明没有更多输入变量的enum。

输出变量

每一个输出变量导致十个参数被创建（见下文）。这些变量由该函数填充。

ECPGt_EORT

一个说明没有更多变量的enum。

对于每一个作为SQL命令一部分的变量，该函数得到十个参数：

1. 作为一个特殊符号的类型。

2. 一个值的指针或者一个指针的指针。

3. 如果变量是一个char或者varchar，这是它的尺寸。

4. 数组中元素的数量（用于数组获取）。

5. 数组中下一个元素的偏移量（用于数组获取）。

6. 作为一个特别符号的指示符变量的类型。

7. 一个指示符变量的指针。

8. 0

9. 指示符数组中的元素数量（用于数组获取）。

10.到指示符数组中下一个元素的偏移量（用于数组取得）。

注意并非所有 SQL 命令都被以这种方式对待。例如，一个打开游标语句：

```
EXEC SQL OPEN cursor;
```

不会被复制到输出。反而，游标的DECLARE命令被用在OPEN命令的位置上，因为它事实上会打开该游标。

这里有一个完整的例子，它描述了一个文件foo.pgc的预处理器输出（对预处理器的每一个特定版本细节可能不同）：

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
```

```
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

会被翻译成：

```
/* 由 ecpg (2.6.0) 处理 */
/* 这两个头文件由预处理器增加 */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* 声明节开始 */

#line 1 "foo.pgc"

 int index;
 int result;
/* 声明节结束 */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

（这里的缩进是为了可读性而添加的，并非是预处理器做的处理）。

第 37 章 信息模式

信息模式由一组视图构成，它们包含定义在当前数据库中对象的信息。信息模式以 SQL 标准定义，因此能够被移植并且保持稳定 — 系统目录则不同，它们是与PostgreSQL相关的并且是为了实现的考虑而建模的。不过，信息模式视图不包含与PostgreSQL-相关特性有关的信息。要咨询那些信息你需要查询系统目录或其他PostgreSQL-相关视图。

> **注意**
>
> 当在数据库中查询约束信息时，一个期望返回一行的标准兼容的查询可能返回多行。这是因为 SQL 标准要求约束名在一个模式中唯一，但是PostgreSQL并不强制这种限制。PostgreSQL自动产生的约束名避免在相同的模式中重复，但是用户能够指定这种重复的名称。
>
> 这个问题可能在查询信息模式视图时出现，例如check_constraint_routine_usage、check_constraints、domain_constraints和 referential_constraints。一些其他视图也有相似的问题，但是它们包含了表名来帮助区分重复行，例如constraint_column_usage、constraint_table_usage、table_constraints。

37.1. 模式

信息模式本身是一个名为information_schema的模式。这个模式自动存在于所有数据库中。这个模式的拥有者是集簇中的初始数据库用户，并且该用户自然地拥有这个模式上的所有特权，包括删除它的能力（但是这样节省的空间是很小的）。

默认情况下，信息模式不在模式搜索路径中，因此你需要使用限定名访问其中的所有对象。因为信息模式中的某些对象的名称是可能出现在用户应用中的一般名称，如果你想把该信息模式放在路径中，你应该小心。

37.2. 数据类型

信息模式视图的列使用定义在信息模式中的特殊数据类型。它们被定义为普通内建类型之上的简单域。你不应在信息模式之外使用这些类型进行工作，但是如果你的应用从信息模式中进行选择，那你的应用就必须准备好面对它们。

这些类型是：

cardinal_number

一种非负整数。

character_data

一种字符串（没有指定最大长度）。

sql_identifier

一种字符串。这种类型被用于 SQL 标识符，类型character_data被用于任何其他类型的文本数据。

time_stamp

在类型timestamp with time zone之上的一个域。

yes_or_no

一种字符串域，它包含YES或NO。这被用来在信息模式中表示布尔（真/假）（信息模式是在类型boolean被加到 SQL 标准之前被发明的，因此这个惯例是用来使信息模式向后兼容）。

信息模式中的每一列都是这五种类型之一。

37.3. information_schema_catalog_name

information_schema_catalog_name是一个表，它总是包含一行和一列，其中包含了当前数据库（SQL 术语中的当前目录）的名字。

表 37.1.information_schema_catalog_name列

名称	数据类型	描述
catalog_name	sql_identifier	包含这个信息模式的数据库名

37.4. administrable_role_authorizations

视图administrable_role_authorizations标识当前用户对其有管理选项的所有角色。

表 37.2.administrable_role_authorizations列

名称	数据类型	描述
grantee	sql_identifier	被授予这个角色的成员关系的角色名（可以是当前用户，或者在嵌套角色成员关系情况下的一个不同角色）
role_name	sql_identifier	角色名
is_grantable	yes_or_no	总是 YES

37.5. applicable_roles

视图applicable_roles当前用户可以使用其特权的所有角色。这意味着有某种角色授权链从当前用户到讨论中的角色。当前用户本身也是一个可应用的角色。可应用的角色的集合通常被用于权限检查。

表 37.3.applicable_roles列

名称	数据类型	描述
grantee	sql_identifier	被授予这个角色的成员关系的角色名（可以是当前用户，或者在嵌套角色成员关系情况下的一个不同角色）
role_name	sql_identifier	一个角色的名字
is_grantable	yes_or_no	YES表示被授予者在该角色上有管理选项，NO表示没有管理选项

37.6. attributes

视图attributes包含数据库中定义的组合数据类型的属性的有关信息（注意该视图并不给出有关表列的信息，表列有时候在 PostgreSQL 上下文环境中也被称为属性）。只有当前用户能够访问（由于是拥有者获得的权限或是在类型上有某些特权）的那些属性会被显示。

表 37.4.attributes列

名称	数据类型	描述
udt_catalog	sql_identifier	包含该数据类型的数据库名（总是当前数据库）
udt_schema	sql_identifier	包含该数据类型的模式名
udt_name	sql_identifier	数据类型名
attribute_name	sql_identifier	属性名
ordinal_position	cardinal_number	属性在该数据类型内部的顺序位置（从 1 开始计算）
attribute_default	character_data	该属性的默认表达式
is_nullable	yes_or_no	如果该属性是可能为空的，值为YES，否则为NO
data_type	character_data	如果该属性是一个内建类型，此列值为该属性的数据类型；如果该属性是某种数组，此列值为ARRAY（在这种情况下，见视图element_types）；其他情况，此列值为USER-DEFINED（在这种情况下，该类型在attribute_udt_name和相关列中标识）。
character_maximum_length	cardinal_number	如果data_type标识一个字符或位串类型，这里是声明的最大长度；如果没有声明最大长度，则对于所有其他数据类型为空。
character_octet_length	cardinal_number	如果data_type标识一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	包含此属性排序规则的数据库名（总是当前数据库），如果默认或者该属性的数据类型不可排序则为空

名称	数据类型	描述
collation_schema	sql_identifier	包含此属性排序规则的模式名，如果默认或者该属性的数据类型不可排序则为空
collation_name	sql_identifier	该属性排序规则的名称，如果默认或者该属性的数据类型不可排序则为空
numeric_precision	cardinal_number	如果data_type标识一种数字类型，这列包含这个属性类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix	cardinal_number	如果data_type标识一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale	cardinal_number	如果data_type标识一种准确数字类型，这列包含这个属性类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision	cardinal_number	如果data_type标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个属性类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位数。对于所有其他数据类型，这一列为空。
interval_type	character_data	如果data_type标识一种时间间隔类型，这一列包含时间间隔为这个属性包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
interval_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性（关于时间间隔类型属性的分数秒精度可见datetime_precision）

名称	数据类型	描述
attribute_udt_catalog	sql_identifier	属性数据类型被定义的数据库名（总是当前数据库）
attribute_udt_schema	sql_identifier	属性数据类型被定义的模式名
attribute_udt_name	sql_identifier	属性数据类型的名称
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该列的数据类型描述符的一个标识符，在从属于该表的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
is_derived_reference_attribute	yes_or_no	应用于一个PostgreSQL中不可用的特性

关于某些列的详情，参见第 37.16 节下的一个相似结构的视图。

37.7. character_sets

视图character_sets标识当前数据库中可用的字符集。因为 PostgreSQL 不支持在同一个数据库中有多个字符集，这个视图只显示一个字符集，它就是数据库编码。

注意下列术语在 SQL 标准中是怎样使用的：

字元集（character repertoire）

字符的一个抽象集合，例如UNICODE、UCS或LATIN1。它不作为一个 SQL 对象显示，但是在这个视图中可见。

字符编码形式（character encoding form）

某种字元集的一种编码。大部分较老的字元集只使用一种编码形式，并且因此它们没有独立的名字（例如LATIN1就是一种适用于LATIN1字元集的编码形式）。但是 Unicode 就有几种编码形式如UTF8、UTF16等等（并非全部被 PostgreSQL 支持）。编码形式不作为一个 SQL 对象显示，但是在这个视图中可见。

字符集（character set）

一个标识一种字元集、一种字符编码以及一种默认排序规则的命名 SQL 对象。一个预定义的字符集通常具有和一种编码形式相同的名字，但是用户可以定义其他名字。例如，字符集UTF8通常标识字元集UCS、编码形式UTF8以及某种默认排序规则。

你可以把 PostgreSQL 中的一种“编码”想成一个字符集或是一种字符编码形式。它们将具有相同的名字，并且在一个数据库中只能用其中一个。

表 37.5.character_sets列

名称	数据类型	描述
character_set_catalog	sql_identifier	当前字符集并未被实现为模式对象，因此这一列为空。
character_set_schema	sql_identifier	当前字符集并未被实现为模式对象，因此这一列为空。
character_set_name	sql_identifier	该字符集的名字，当前实现为显示该数据库编码的名字
character_repertoire	sql_identifier	字元集，如果编码为UTF8则显示UCS，否则只显示编码名称
form_of_use	sql_identifier	字符编码形式，与数据库编码相同
default_collate_catalog	sql_identifier	包含该默认排序规则的数据库名（如果任意排序规则被标识，总是当前数据库）
default_collate_schema	sql_identifier	包含该默认排序规则的模式名
default_collate_name	sql_identifier	默认排序规则的名字。该默认排序规则被标识为匹配当前数据库 的COLLATE和CTYPE设置的排序规则。如果没有那种排序规则，那么这一列和相关模式以及目录列为空。

37.8. check_constraint_routine_usage

视图check_constraint_routine_usage标识被检查约束琐事使用的例程（函数和过程）。只有被一个当前启用的角色所拥有的例程才被显示。

表 37.6.check_constraint_routine_usage列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“指定名称”。详见第 37.40 节

37.9. check_constraints

视图check_constraints包含所有检查约束，不管是定义在一个表上的还是定义在一个域上的，它们被一个当前启用的角色所拥有（表或域的拥有者就是约束的拥有者）。

表 37.7.check_constraints列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
check_clause	character_data	该检查约束的检查表达式

37.10. collations

视图collations包含在当前数据库中可用的排序规则。

表 37.8.collations列

名称	数据类型	描述
collation_catalog	sql_identifier	包含该排序规则的数据库名（总是当前数据库）
collation_schema	sql_identifier	包含该排序规则的模式名
collation_name	sql_identifier	默认排序规则的名称
pad_attribute	character_data	总是NO PAD（另一种选择PAD SPACE没有被 PostgreSQL 支持）

37.11. collation_character_set_applicability

视图collation_character_set_applicability标识可用的排序规则适用于哪些字符集。在 PostgreSQL 中，每个数据库中只有一种字符集（解释见第 37.7 节，因此这个视图没有提供很有用的信息。

表 37.9.collation_character_set_applicability列

名称	数据类型	描述
collation_catalog	sql_identifier	包含该排序规则的数据库名（总是当前数据库）
collation_schema	sql_identifier	包含该排序规则的模式名
collation_name	sql_identifier	默认排序规则的名称
character_set_catalog	sql_identifier	当前字符集还未被实现为模式对象，所以这一列为空
character_set_schema	sql_identifier	当前字符集还未被实现为模式对象，所以这一列为空
character_set_name	sql_identifier	字符集名称

37.12. column_domain_usage

视图column_domain_usage标识所有使用定义在当前数据库中并且被一个当前启用的角色拥有的域的列（表列或视图列）。

表 37.10.column_domain_usage列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名
domain_name	sql_identifier	域名称
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
column_name	sql_identifier	列名称

37.13. column_options

视图column_options包含为当前数据库中外部表列定义的所有选项。只有当前用户能够访问（作为拥有者或具有某些特权）的那些外部表列才被显示。

表 37.11.column_options列

名称	数据类型	描述
table_catalog	sql_identifier	包含该外部表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该外部表的模式名
table_name	sql_identifier	外部表名
column_name	sql_identifier	列名称
option_name	sql_identifier	一个选项名
option_value	character_data	该选项的值

37.14. column_privileges

视图column_privileges标识所有授予给一个当前启用的角色或者被一个当前启用的角色授予的特权。对每一个列、授予者、被授予者的组合只有一行。

如果一个特权被授予在一整个表上，它在这个视图中被显示为在每一列上授予，但是只有可用于列粒度的特权类型才会这样： SELECT、INSERT、 UPDATE、REFERENCES。

表 37.12.column_privileges列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
table_catalog	sql_identifier	包含该列的表所在的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名
table_name	sql_identifier	包含该列的表名
column_name	sql_identifier	列名称
privilege_type	character_data	特权类型：SELECT、INSERT、UPDATE或REFERENCES

名称	数据类型	描述
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.15. column_udt_usage

视图column_udt_usage标识所有使用被一个当前启用的角色拥有的数据类型的列。注意在PostgreSQL中，内建数据类型的行为和用户定义的类型相似，因此它们也被包括在这里。详见第 37.16 节

表 37.13.column_udt_usage列

名称	数据类型	描述
udt_catalog	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的模式名
udt_name	sql_identifier	该列数据类型（如果适用，底层的域类型）的名称
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
column_name	sql_identifier	列名称

37.16. columns

视图columns包含数据库中有关所有表列（或视图列）的信息。系统列（oid等）不被包括在内。只有那些当前用户能够访问（作为拥有者或具有某些特权）的列才被显示。

表 37.14.columns列

名称	数据类型	描述
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
column_name	sql_identifier	列名称
ordinal_position	cardinal_number	该列在表内的顺序位置（从 1 开始计）
column_default	character_data	该列的默认表达式
is_nullable	yes_or_no	如果该列可以为空，则为YES，否则为NO。一个非空约束是让一列成为不能为空的方法，但还有其他方法。
data_type	character_data	如果该列的数据类型是一种内建类型，则为该列的数据类型；如果是某种数组（此种情况见视

名称	数据类型	描述
		图element_types），则为ARRAY；否则为USER-DEFINED（此种情况下该类型被标识在udt_name和相关列中）。如果该列基于一个域，这一列引用该域底层的类型（该列被标识在domain_name和相关列中）。
character_maximum_length	cardinal_number	如果data_type标识一种字符或位串类型，这里是声明的最大长度；如果没有声明最大长度或者所有其他数据类型，这里为空。
character_octet_length	cardinal_number	如果data_type标识一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
numeric_precision	cardinal_number	如果data_type标识一种数字类型，这列包含这个属性类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix	cardinal_number	如果data_type标识一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale	cardinal_number	如果data_type标识一种准确数字类型，这列包含这个属性类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision	cardinal_number	如果data_type标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个属性类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位

名称	数据类型	描述
		数。对于所有其他数据类型，这一列为空。
interval_type	character_data	如果data_type标识一种时间间隔类型，这一列包含时间间隔为这个属性包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
interval_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性（关于时间间隔类型属性的分数秒精度可见datetime_precision）
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	包含此属性排序规则的数据库名（总是当前数据库），如果默认或者该属性的数据类型不可排序则为空
collation_schema	sql_identifier	包含此属性排序规则的模式名，如果默认或者该属性的数据类型不可排序则为空
collation_name	sql_identifier	该属性排序规则的名称，如果默认或者该属性的数据类型不可排序则为空
domain_catalog	sql_identifier	如果该列有一个域类型，这里是该域所在的数据库名（总是当前数据库），否则为空。
domain_schema	sql_identifier	如果该列有一个域类型，这里是该域所在的模式名，否则为空。
domain_name	sql_identifier	如果该列有一个域类型，这里是该域的名称，否则为空。
udt_catalog	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的模式名
udt_name	sql_identifier	该列数据类型（如果适用，底层的域类型）的名称

名称	数据类型	描述
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该列的数据类型描述符的一个标识符，在从属于该表的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
is_self_referencing	yes_or_no	应用于一个PostgreSQL中不可用的特性
is_identity	yes_or_no	如果该列是一个标识列，则为YES，否则为NO。
identity_generation	character_data	如果该列是一个标识列，则为ALWAYS或者BY DEFAULT，它反映该列的定义。
identity_start	character_data	如果该列是一个标识列，则是内部序列的起始值，否则为空。
identity_increment	character_data	如果该列是一个标识列，则是内部序列的增量，否则为空。
identity_maximum	character_data	如果该列是一个标识列，则是内部序列的最大值，否则为空。
identity_minimum	character_data	如果该列是一个标识列，则是内部序列的最小值，否则为空。
identity_cycle	yes_or_no	如果该列是一个标识列，则YES和NO分别表示内部序列可循环和不可循环，否则为空。
is_generated	character_data	应用于一个PostgreSQL中不可用的特性
generation_expression	character_data	应用于一个PostgreSQL中不可用的特性
is_updatable	yes_or_no	如果该列是可更新的，则为YES，否则为NO（基表中的列总是可更新的，视图中的列则不一定）

因为在 SQL 中有多种方式定义数据类型，而PostgreSQL还包含额外的方式来定义数据类型，它们在信息模式中的表示可能有点困难。列data_type应该标识列的底层内建类型。在

PostgreSQL中，这表示定义在系统目录模式pg_catalog中的类型。如果应用能够特别地（例如以不同方式格式化数字类型或使用精度列中的数据）处理总所周知的内建类型，这列可能会有用。列udt_name、udt_schema和udt_catalog总是标识列的底层数据类型，即使该列是基于一个域的（因为PostgreSQL对待内建类型和用户定义类型的方式是一样的，内建类型也出现在这里。这是 SQL 标准的一种扩展）。如果一个应用想要根据该类型以不同的方式处理数据，就应该使用这些列，因为在那种情况下即使该列真地基于一个域也没有关系。如果该列是基于一个域，该域的标识被存储在列domain_name、domain_schema和domain_catalog。如果你想要把列和它们相关的数据类型配对并且把域视作单独的类型，你可以写coalesce(domain_name, udt_name)等等。

37.17. constraint_column_usage

视图constraint_column_usage标识在当前数据库中被某个约束使用的所有列。只有包含在被一个当前启用的角色拥有的表中的那些列才被显示。对于一个检查约束，这个视图标识被用在该检查约束中的列。对于一个外键约束，这个视图标识外键引用的列。对于一个唯一或主键约束，这个视图标识被约束的列。

表 37.15.constraint_column_usage列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的列的表所在的数据库名（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的列的表所在的模式名
table_name	sql_identifier	包含被某个约束使用的列的表名
column_name	sql_identifier	包含被某个约束使用的列名
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名

37.18. constraint_table_usage

视图constraint_table_usage标识在当前数据库中被某个约束使用的所有表（这与视图table_constraints不同，它标识哪些表约束定义在哪些表上）。对于一个外键约束，这个视图标识该外键引用的表。对于一个唯一或主键约束，这个视图仅标识该约束属于的表。检查约束和非空约束不被包括在这个视图中。

表 37.16.constraint_table_usage列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的表的模式名
table_name	sql_identifier	包含被某个约束使用的表名
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名

名称	数据类型	描述
constraint_name	sql_identifier	约束名

37.19. data_type_privileges

视图data_type_privileges标识当前用户能够访问（作为被描述对象的拥有者或者具有其上的某种特权）的所有数据类型描述符。只要一个数据类型被用在一个表列、一个域或一个函数（作为参数或返回类型）就会生成一个数据类型描述符并且在那个实例中存储一些有关该数据类型如何被使用的信息（例如，声明的最大长度，如果适用）。每一个数据类型描述符被赋予一个任意的标识符，它在被赋予给一个对象（表、域、函数）的数据类型描述符中唯一。这个视图对于应用可能没什么用，但是它被用于定义信息模式中的一些其他视图。

表 37.17.data_type_privileges列

名称	数据类型	描述
object_catalog	sql_identifier	包含该被描述对象的数据库名（总是当前数据库）
object_schema	sql_identifier	包含该被描述对象的模式名
object_name	sql_identifier	该描述对象的名字
object_type	character_data	被描述对象的类型：TABLE（从属于表的一列的数据类型描述符）、DOMAIN （从属于域的数据类型描述符）、ROUTINE（从属于函数的一个参数或返回数据类型的数据类型描述符）。
dtd_identifier	sql_identifier	数据类型描述符的标识符，它在同一对象的数据类型描述符之间唯一。

37.20. domain_constraints

视图domain_constraints包含所有属于当前数据库中定义的域的约束。只有当前用户能访问的那些域才被显示（作为拥有者或具有某些特权）。

表 37.18.domain_constraints列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
domain_catalog	sql_identifier	包含该域的数据库名（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名
domain_name	sql_identifier	该域的名称
is_deferrable	yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred	yes_or_no	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO

37.21. domain_udt_usage

视图domain_udt_usage标识所有基于被一个当前启用的角色拥有的数据类型的域。注意在PostgreSQL中，内建数据类型的行为相似于用户定义的类型，因此它们也被包括在这里。

表 37.19.domain_udt_usage列

名称	数据类型	描述
udt_catalog	sql_identifier	该域数据类型被定义的数据库名（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名
udt_name	sql_identifier	该域数据类型的名称
domain_catalog	sql_identifier	包含该域的数据库名（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名
domain_name	sql_identifier	该域的名称

37.22. domains

视图domains包含定义在当前数据库中的所有域。只有当前用户能够访问（作为拥有者或具有某些特权）的域才被显示。

表 37.20.domains列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名
domain_name	sql_identifier	该域的名称
data_type	character_data	该域的数据类型如果是一种内建类型，这里是该域的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length	cardinal_number	如果该域有一个字符或位串类型，这里是声明的最大长度；如果没有声明最大长度，则对于所有其他数据类型为空。
character_octet_length	cardinal_number	如果该域有一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。

名称	数据类型	描述
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	包含此域排序规则的数据库名（总是当前数据库），如果默认或者该域的数据类型不可排序则为空
collation_schema	sql_identifier	包含此域排序规则的模式名，如果默认或者该域的数据类型不可排序则为空
collation_name	sql_identifier	该域排序规则的名称，如果默认或者该域的数据类型不可排序则为空
numeric_precision	cardinal_number	如果该域有一种数字类型，这列包含这个域类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix	cardinal_number	如果该域有一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale	cardinal_number	如果该域有一种准确数字类型，这列包含这个域类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision	cardinal_number	如果data_type标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个域类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位数。对于所有其他数据类型，这一列为空。
interval_type	character_data	如果data_type标识一种时间间隔类型，这一列包含时间

名称	数据类型	描述
		间隔为这个域包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
interval_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性（关于时间间隔类型域的分数秒精度可见datetime_precision）
domain_default	character_data	该域的默认表达式
udt_catalog	sql_identifier	该域数据类型被定义的数据库名（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名
udt_name	sql_identifier	该域数据类型的名称
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该域的数据类型描述符的一个标识符，在从属于该域的数据类型标识符之中唯一（这不重要，因为一个域只包含一个数据类型描述符）。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。

37.23. element_types

视图element_types包含数组元素的数据类型描述符。当一个表列、组合类型属性、域、函数参数或函数返回值被定义为一种数组类型，相应的信息模式视图只在列data_type中包含ARRAY。要获得该数组元素类型的信息，你可以连接该相应的视图和这个视图。例如，要显示一个表的列及其数据类型和数组元素类型，你可以：

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE',
 c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
 e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
```

```
ORDER BY c.ordinal_position;
```

这个视图只包括当前用户能够访问（作为拥有者或具有某些特权）的对象。

表 37.21.element_types列

名称	数据类型	描述
object_catalog	sql_identifier	包含使用被描述的数组的对象的数据库名（总是当前数据库）
object_schema	sql_identifier	包含使用被描述的数组的对象的模式名
object_name	sql_identifier	使用被描述的模式的对象名
object_type	character_data	使用被描述的数组的对象的类型：TABLE（被一个表列使用的数组）、USER-DEFINED TYPE（被组合类型的一个属性使用的数组）、DOMAIN（被域使用的数组）、ROUTINE（被函数的一个参数或返回数据类型使用的数组）。
collection_type_identifier	sql_identifier	被描述的数组的数据类型描述符的标识符。使用这个去与其他信息模式视图的dtd_identifier列连接。
data_type	character_data	如果数组元素的数据类型是内建类型，这里是数组元素的数据类型，否则为USER-DEFINED（在那种情况下，该类型被标识在udt_name和相关列中）。
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	包含元素类型排序规则的数据库名（总是当前数据库），如果默认或该元素的数据类型是不可排序的则为空
collation_schema	sql_identifier	包含元素类型排序规则的模式名，如果默认或该元素的数据类型是不可排序的则为空

名称	数据类型	描述
collation_name	sql_identifier	元素类型的排序规则名，如果默认或该元素的数据类型是不可排序的则为空
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
interval_type	character_data	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
interval_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的数组元素数据类型
domain_default	character_data	还未被实现
udt_catalog	sql_identifier	元素的数据类型所在的数据库名（总是当前数据库）
udt_schema	sql_identifier	元素的数据类型所在的模式名
udt_name	sql_identifier	模式的数据类型名
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该元素的数据类型描述符的标识符。当前无用。

37.24. enabled_roles

视图enabled_roles标识当前“已被启用的角色”。已被启用的角色被递归地定义为：当前用户以及被授予给具有自动继承的已被启用角色的所有角色。换句话说，就是当前用户是其直接或间接成员的所有角色。

为了权限检查，“可应用角色”的集合被应用，它会比已被启用角色的集合包含的角色范围更宽。因此通常使用视图applicable_roles要更好，applicable_roles视图的详情请见第 37.5 节

表 37.22.enabled_roles列

名称	数据类型	描述
role_name	sql_identifier	角色名称

37.25. foreign_data_wrapper_options

视图foreign_data_wrapper_options包含为当前数据库中外部数据包装器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部数据包装器被显示。

表 37.23.foreign_data_wrapper_options列

名称	数据类型	描述
foreign_data_wrapper_catalog	sql_identifier	该外部数据包装器所在的数据库名（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	该外部数据包装器的名字
option_name	sql_identifier	一个选项名
option_value	character_data	该选项的值

37.26. foreign_data_wrappers

视图foreign_data_wrappers包含定义在当前数据库中的所有外部数据包装器。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部数据包装器才会被显示。

表 37.24.foreign_data_wrappers列

名称	数据类型	描述
foreign_data_wrapper_catalog	sql_identifier	包含该外部数据包装器的数据库名（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	外部数据包装器的名字
authorization_identifier	sql_identifier	外部服务器拥有者的名字
library_name	character_data	实现这个外部数据包装器的库文件名
foreign_data_wrapper_language	character_data	用于实现这个外部数据包装器的语言

37.27. foreign_server_options

视图foreign_server_options包含为当前数据库中外部服务器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部服务器才会被显示。

表 37.25.foreign_server_options列

名称	数据类型	描述
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名（总是当前数据库）
foreign_server_name	sql_identifier	该外部服务器的名字
option_name	sql_identifier	一个选项名
option_value	character_data	该选项的值

37.28. foreign_servers

视图foreign_servers包含当前数据库中定义的所有外部服务器。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部服务器才会被显示。

表 37.26.foreign_servers列

名称	数据类型	描述
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名（总是当前数据库）
foreign_server_name	sql_identifier	该外部服务器的名字
foreign_data_wrapper_catalog	sql_identifier	包含被该外部服务器使用的外部数据包装器的数据库名（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	被该外部服务器所使用的外部数据包装器的名字
foreign_server_type	character_data	外部服务器类型信息（如果在创建时指定过）
foreign_server_version	character_data	外部服务器版本信息（如果在创建时指定过）
authorization_identifier	sql_identifier	该外部服务器的拥有者名字

37.29. foreign_table_options

视图foreign_table_options包含为当前数据库中外部表定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部表才会被显示。

表 37.27.foreign_table_options列

名称	数据类型	描述
foreign_table_catalog	sql_identifier	包含该外部表的数据库名（总是当前数据库）
foreign_table_schema	sql_identifier	包含该外部表的模式名
foreign_table_name	sql_identifier	外部表的名字
option_name	sql_identifier	一个选项名
option_value	character_data	该选项的值

37.30. foreign_tables

视图foreign_tables包含定义在当前数据库中的所有外部表。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部表才会被显示。

表 37.28.foreign_tables列

名称	数据类型	描述
foreign_table_catalog	sql_identifier	该外部表所在的数据库名（总是当前数据库）
foreign_table_schema	sql_identifier	包含该外部表的模式名
foreign_table_name	sql_identifier	该外部表的名字
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名（总是当前数据库）

名称	数据类型	描述
foreign_server_name	sql_identifier	该外部服务器的名字

37.31. key_column_usage

视图key_column_usage标识当前数据库中所有被某种唯一、主键或外键约束限制的列。检查约束不被包括在这个视图中。只有那些当前用户能够访问的列才会被显示（作为拥有者或具有某些特权）。

表 37.29.key_column_usage列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
table_catalog	sql_identifier	包含被这个约束限制的列的表所在的数据库名（总是当前数据库）
table_schema	sql_identifier	包含被这个约束限制的列的表所在的模式名
table_name	sql_identifier	包含被这个约束限制的列的表的名字
column_name	sql_identifier	被这个约束限制的列名
ordinal_position	cardinal_number	该列在约束键中的顺序位置（从 1 开始计数）
position_in_unique_constraint	cardinal_number	对于一个外键约束，被引用行在其唯一约束中的顺序位置（从 1 开始计数）；对于其他约束为空

37.32. parameters

视图parameters包含当前数据库中所有函数的参数的有关信息。只有那些当前用户能够访问（作为拥有者或具有某些特权）的函数才会被显示。

表 37.30.parameters列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“专用名”。详见第 37.40 节
ordinal_position	cardinal_number	该参数在函数参数列表中的顺序位置（从 1 开始计数）
parameter_mode	character_data	IN表示输入参数， OUT表示输出参数， INOUT表示输入输出参数。
is_result	yes_or_no	应用于一个PostgreSQL中不可用的特性

名称	数据类型	描述
as_locator	yes_or_no	应用于一个PostgreSQL中不可用的特性
parameter_name	sql_identifier	参数名，如果参数没有名称则为空
data_type	character_data	该参数的数据类型如果是一种内建类型，这里是该参数的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_schema	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_name	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
interval_type	character_data	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型

名称	数据类型	描述
interval_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
udt_catalog	sql_identifier	该参数的数据类型所在的数据库名（总是当前数据库）
udt_schema	sql_identifier	该参数的数据类型所在的模式名
udt_name	sql_identifier	该参数的数据类型的名字
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该参数的数据类型描述符的一个标识符，在从属于该函数的数据类型标识符之中唯一（这不重要，因为一个域只包含一个数据类型描述符）。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
parameter_default	character_data	该参数的默认表达式，如果没有或者该函数不被一个当前启用的角色拥有则为空值。

37.33. referential_constraints

视图referential_constraints包含当前数据库中的所有引用（外键）约束。只有那些当前用户具有其引用表上写权限（作为拥有者或具有某些除SELECT之外的特权）的约束才会被显示。

表 37.31.referential_constraints列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
unique_constraint_catalog	sql_identifier	包含该外键约束所引用的唯一或主键约束的数据库名（总是当前数据库）
unique_constraint_schema	sql_identifier	包含该外键约束所引用的唯一或主键约束的模式名

名称	数据类型	描述
unique_constraint_name	sql_identifier	包含该外键约束所引用的唯一或主键约束的名字
match_option	character_data	外键约束的匹配选项：FULL、PARTIAL或NONE。
update_rule	character_data	外键约束的更新规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT或NO ACTION。
delete_rule	character_data	外键约束的删除规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT或NO ACTION。

37.34. role_column_grants

视图role_column_grants标识所有在列上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在column_privileges中找到。这个视图和column_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的列。

表 37.32.role_column_grants列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
table_catalog	sql_identifier	包含该列的表所在的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名
table_name	sql_identifier	包含该列的表名
column_name	sql_identifier	列名称
privilege_type	character_data	特权类型：SELECT、INSERT、UPDATE或REFERENCES
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.35. role_routine_grants

视图role_routine_grants标识所有在函数上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在routine_privileges中找到。这个视图和routine_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的函数。

表 37.33.role_routine_grants列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）

名称	数据类型	描述
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“专用名”。详见第 37.40 节
routine_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名
routine_name	sql_identifier	该函数的名字（在重载的情况下可能会重复）
privilege_type	character_data	总是为EXECUTE（函数唯一的特权类型）
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.36. role_table_grants

视图role_table_grants标识所有在表或视图上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在table_privileges中找到。这个视图和table_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的表。

表 37.34. role_table_grants列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
privilege_type	character_data	该特权的类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO
with_hierarchy	yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION是一个独立的（子）特权，它允许在表继承层级上的特定操作。在PostgreSQL 中，这被包括在SELECT特权中，因此这一列在特权为SELECT时显示YES，其他时候显示NO。

37.37. role_udt_grants

视图role_udt_grants标识所有在用户定义类型上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在udt_privileges中找到。这个视图和udt_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当

前用户获得其访问权限的对象。因为数据类型在 PostgreSQL 中并没有真正的特权，而是只有一个给PUBLIC的隐式授予，这个视图为空。

表 37.35.role_udt_grants列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
udt_catalog	sql_identifier	包含该类型的数据库名（总是当前数据库）
udt_schema	sql_identifier	包含该类型的模式名
udt_name	sql_identifier	该类型的名字
privilege_type	character_data	总是TYPE USAGE
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.38. role_usage_grants

视图role_usage_grants标识所有在多种对象上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在usage_privileges中找到。这个视图和usage_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的对象。

表 37.36.role_usage_grants列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
object_catalog	sql_identifier	包含该对象的数据库名（总是当前数据库）
object_schema	sql_identifier	如果适用，则为包含该对象的模式名，否则为一个空字符串
object_name	sql_identifier	对象的名字
object_type	character_data	COLLATION或DOMAIN或FOREIGN DATA WRAPPER或FOREIGN SERVER或SEQUENCE
privilege_type	character_data	总是USAGE
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.39. routine_privileges

视图routine_privileges标识所有在函数上授予的特权，其授予者或被授予者是一个当前已被启用的角色。对于每一种函数、授予者和被授予者的组合，这里都有一行。

表 37.37.routine_privileges列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“专用名”。详见第 37.40 节
routine_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名
routine_name	sql_identifier	该函数的名字（在重载的情况下可能重复）
privilege_type	character_data	总是EXECUTE（函数唯一的特权类型）
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.40. routines

视图routines包含当前数据库中所有的函数和过程。只有那些当前用户能够访问（作为拥有者或具有某些特权）的函数和过程才会被显示。

表 37.38.routines列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“专用名”。这是一个在模式中唯一标识该函数的名称，即使该函数真正的名称已经被重载。专用名的格式尚未被定义，它应当仅被用来与指定例程名称的其他实例进行比较。
routine_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名
routine_name	sql_identifier	该函数的名字（在重载的情况下可能重复）
routine_type	character_data	FUNCTION表示是一个函数，PROCEDURE表示是一个过程
module_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
module_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
module_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
udt_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性

名称	数据类型	描述
udt_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
udt_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
data_type	character_data	该函数的返回数据类型如果是一种内建类型，这里是该数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在type_udt_name和相关列中）。如果是过程则此项为空。
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_schema	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_name	sql_identifier	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
interval_type	character_data	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型

名称	数据类型	描述
interval_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL中的返回数据类型
type_udt_catalog	sql_identifier	该函数的返回数据类型所在的数据库名（总是当前数据库）。如果是过程则此项为空。
type_udt_schema	sql_identifier	该函数的返回数据类型所在的模式名。如果是过程则此项为空。
type_udt_name	sql_identifier	该函数的返回数据类型的名字。如果是过程则此项为空。
scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
maximum_cardinality	cardinal_number	总是空，因为数组在PostgreSQL中总是有无限制的最大势
dtd_identifier	sql_identifier	该函数返回数据类型的数据类型描述符的一个标识符，在从属于该函数的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
routine_body	character_data	如果该函数是一个 SQL 函数，则为SQL，否则为EXTERNAL。
routine_definition	character_data	该函数的源文本（如果该函数不属于一个当前已被启用的角色，则为空）。（根据SQL 标准，只有routine_body为SQL时这一列才适用。但是在PostgreSQL中，它将会包含该函数被创建时所指定的任何源文本。）
external_name	character_data	如果这个函数是一个 C 函数，则为该函数的外部名称（链接符号），否则为空（这会产生和显示在routine_definition中相同的值）。
external_language	character_data	该函数所用的语言

名称	数据类型	描述
parameter_style	character_data	总是GENERAL（SQL 标准定义了其他参数风格，但在PostgreSQL中不可用）
is_deterministic	yes_or_no	如果该函数被声明为不变（在 SQL 标准中被称为确定性的），则为YES，否则为NO（你不能通过该信息模式查询在PostgreSQL中可用的其他易变级别）。
sql_data_access	character_data	总是MODIFIES，表示该函数可能修改 SQL 数据。这种信息对PostgreSQL没有用处。
is_null_call	yes_or_no	如果该函数在任一参数为空时自动返回空值，则为YES，否则为NO。如果是过程则此项为空。
sql_path	character_data	应用于一个PostgreSQL中不可用的特性
schema_level_routine	yes_or_no	总是YES（反例是一个用户定义类型的方法，这是在PostgreSQL不可用的一种特性）。
max_dynamic_result_sets	cardinal_number	应用于一个PostgreSQL中不可用的特性
is_user_defined_cast	yes_or_no	应用于一个PostgreSQL中不可用的特性
is_implicitly_invocable	yes_or_no	应用于一个PostgreSQL中不可用的特性
security_type	character_data	如果该函数以当前用户的特权运行，则为INVOKER；如果该函数以定义它的用户的特权运行，则为DEFINER。
to_sql_specific_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
to_sql_specific_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
to_sql_specific_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
as_locator	yes_or_no	应用于一个PostgreSQL中不可用的特性
created	time_stamp	应用于一个PostgreSQL中不可用的特性
last_altered	time_stamp	应用于一个PostgreSQL中不可用的特性
new_savepoint_level	yes_or_no	应用于一个PostgreSQL中不可用的特性
is_udt_dependent	yes_or_no	当前总是NO。另一个选项YES应用于一个PostgreSQL中不可用的特性。

名称	数据类型	描述
result_cast_from_data_type	character_data	应用于一个PostgreSQL中不可用的特性
result_cast_as_locator	yes_or_no	应用于一个PostgreSQL中不可用的特性
result_cast_char_max_length	cardinal_number	应用于一个PostgreSQL中不可用的特性
result_cast_char_octet_length	character_data	应用于一个PostgreSQL中不可用的特性
result_cast_char_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_char_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_char_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_collation_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_collation_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_collation_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_numeric_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性
result_cast_numeric_precision_radix	cardinal_number	应用于一个PostgreSQL中不可用的特性
result_cast_numeric_scale	cardinal_number	应用于一个PostgreSQL中不可用的特性
result_cast_datetime_precision	character_data	应用于一个PostgreSQL中不可用的特性
result_cast_interval_type	character_data	应用于一个PostgreSQL中不可用的特性
result_cast_interval_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性
result_cast_type_udt_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_type_udt_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_type_udt_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_scope_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_scope_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_scope_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
result_cast_maximum_cardinality	cardinal_number	应用于一个PostgreSQL中不可用的特性

名称	数据类型	描述
result_cast_dtd_identifier	sql_identifier	应用于一个PostgreSQL中不可用的特性

37.41. schemata

视图schemata包含当前数据库中被当前用户（作为属主或具有某些特权）可访问的所有模式。

表 37.39.schemata列

名称	数据类型	描述
catalog_name	sql_identifier	该模式所在的数据库名（总是当前数据库）
schema_name	sql_identifier	该模式的名称
schema_owner	sql_identifier	该模式拥有者的名称
default_character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
default_character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
default_character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
sql_path	character_data	应用于一个PostgreSQL中不可用的特性

37.42. sequences

视图sequences包含所有定义在当前数据库中的序列。只有那些当前用户能够访问（作为拥有者或具有某些特权）的序列才会被显示。

表 37.40.sequences列

名称	数据类型	描述
sequence_catalog	sql_identifier	包含该序列的数据库名（总是当前数据库）
sequence_schema	sql_identifier	包含该序列的模式名
sequence_name	sql_identifier	该序列的名字
data_type	character_data	该序列的数据类型。
numeric_precision	cardinal_number	这列包含这个序列数据类型（见上文）的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。
numeric_precision_radix	cardinal_number	这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。

名称	数据类型	描述
numeric_scale	cardinal_number	这列包含这个序列数据类型（见上文）的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于2）。
start_value	character_data	该序列的开始值
minimum_value	character_data	该序列的最小值
maximum_value	character_data	该序列的最大值
increment	character_data	该序列的增量
cycle_option	yes_or_no	如果该序列会循环，则为YES，否则为NO

注意依照 SQL 标准，开始值、最小值、最大值和增量值被作为字符串返回。

37.43. sql_features

表sql_features包含的信息指示了哪些 SQL 标准中定义的正式特性被PostgreSQL所支持。这和附录 D中的信息一样。这里你也能找到一些额外的背景信息。

表 37.41.sql_features列

名称	数据类型	描述
feature_id	character_data	该特性的标识符字符串
feature_name	character_data	该特性的描述性名称
sub_feature_id	character_data	该子特性的标识符字符串，或者如果不是一个子特性则为一个长度为零的字符串
sub_feature_name	character_data	该子特性的描述性名称，或者如果不是一个子特性则为一个长度为零的字符串
is_supported	yes_or_no	如果该特性被当前版本的PostgreSQL完全支持，则为YES，否则为NO
is_verified_by	character_data	总是为空，因为PostgreSQL开发组没有对特性的一致性执行正式的测试
comments	character_data	可能会是关于该特性被支持状态的一段注释

37.44. sql_implementation_info

表sql_implementation_info包含的信息指示剩下的由 SQL 标准实现定义的多个方面。这类信息主要用来在 ODBC 接口的情境中使用；其它接口的用户可能将发现这类信息用处不大。由于这个原因，个体实现信息项没有在这里描述，你将会在 ODBC 接口的描述中找到它们。

表 37.42.sql_implementation_info列

名称	数据类型	描述
implementation_info_id	character_data	该实现信息项的标识符字符串
implementation_info_name	character_data	该实现信息项的描述性名称
integer_value	cardinal_number	该实现信息项的值，如果该值被包含在character_value列中则为空
character_value	character_data	该实现信息项的值，如果该值被包含在integer_value列中则为空
comments	character_data	可能是从属于该实现信息项的一段注释

37.45. sql_languages

表sql_languages为每一种被PostgreSQL支持的 SQL 语言绑定包含一行。PostgreSQL支持在 C 中的直接 SQL 和嵌入式 SQL，这是你从这张表中知道的所有东西。

这个表在 SQL:2008 中已被从 SQL 标准中移除，因此这里没有项引用 SQL:2003 之后的标准。

表 37.43.sql_languages列

名称	数据类型	描述
sql_language_source	character_data	该语言定义的源名称，总是ISO 9075，即 SQL 标准
sql_language_year	character_data	sql_language_source中引用的标准被通过的年份。
sql_language_conformance	character_data	该语言绑定的标准一致性级别。对于 ISO 9075:2003 总是CORE。
sql_language_integrity	character_data	总是为空（这个值与一个早期版本的 SQL 标准相关）。
sql_language_implementation	character_data	总是为空
sql_language_binding_style	character_data	语言绑定风格，为DIRECT或EMBEDDED
sql_language_programming_language	character_data	如果绑定风格为EMBEDDED，则为编程语言，否则为空。PostgreSQL仅支持 C 语言。

37.46. sql_packages

表sql_packages包含的信息指示哪些定义在 SQL 标准中的特性包被PostgreSQL支持。特性包上的背景信息可参考附录 D。

表 37.44.sql_packages列

名称	数据类型	描述
feature_id	character_data	该包的标识符字符串

名称	数据类型	描述
feature_name	character_data	该包的描述性名称
is_supported	yes_or_no	如果该包被当前版本的PostgreSQL支持，则为YES，否则为NO
is_verified_by	character_data	总是为空，因为PostgreSQL开发组没有对特性的一致性执行正式的测试
comments	character_data	可能会是关于该包被支持状态的一段注释

37.47. sql_parts

表sql_parts包含的信息指示哪些定义在 SQL 标准中的部分被PostgreSQL支持。

表 37.45.sql_parts列

名称	数据类型	描述
feature_id	character_data	包含该部分编号的一个标识符字符串
feature_name	character_data	该部分的描述性名称
is_supported	yes_or_no	如果当前版本的PostgreSQL完全支持该部分，则为YES，否则为NO
is_verified_by	character_data	总是为空，因为PostgreSQL开发组没有对特性的一致性执行正式的测试
comments	character_data	可能会是关于该部分被支持状态的一段注释

37.48. sql_sizing

表sql_sizing包含有关PostgreSQL中多种尺寸限制和最大值的信息。这类信息主要用来在 ODBC 接口的情境中使用；其它接口的用户可能将发现这类信息用处不大。由于这个原因，个体实现信息项没有在这里描述，你将会在 ODBC 接口的描述中找到它们。

表 37.46.sql_sizing列

名称	数据类型	描述
sizing_id	cardinal_number	该尺寸项的标识符
sizing_name	character_data	该尺寸项的描述性名称
supported_value	cardinal_number	尺寸项的值，如果尺寸是不受限制或不能确定的则为 0，如果尺寸项适用的特性不受支持则为空
comments	character_data	可能是从属于尺寸项的一段注释

37.49. sql_sizing_profiles

表sql_sizing_profiles包含有关 SQL 标准的多种 profile 所需的sql_sizing值的信息。PostgreSQL不追踪任何 SQL profile，因此这个表为空。

表 37.47.sql_sizing_profiles列

名称	数据类型	描述
sizing_id	cardinal_number	该尺寸项的标识符
sizing_name	character_data	该尺寸项的描述性名称
profile_id	character_data	一个 profile 的标识符字符串
required_value	cardinal_number	该 SQL profile 对尺寸项所要求的值，如果该 profile 对尺寸项没有限制则为 0，如果该 profile 不要求该尺寸项所适用的任何特性则为空
comments	character_data	可能是从属于该 profile 中尺寸项的一段注释

37.50. table_constraints

视图table_constraints包含属于特定表的所有约束，这些表要满足的条件是：当前用户拥有表或者是当前用户在表上具有某种除SELECT之外的特权。

表 37.48.table_constraints列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名
constraint_name	sql_identifier	约束名
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
constraint_type	character_data	该约束的类型：CHECK、FOREIGN KEY、PRIMARY KEY或UNIQUE
is_deferrable	yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred	yes_or_no	如果该约束是可延迟的并且是初始已被延迟，则为YES，否则为NO
enforced	yes_or_no	适用于一种PostgreSQL中不可用的特性（当前总是YES）

37.51. table_privileges

视图table_privileges标识在表或视图上所有被授予的特权，这些特权必须是被一个当前已被启用角色授出或者被授予给一个当前已被启用角色。对每一个表、授予者和被授予者的组合都有一行。

表 37.49.table_privileges列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
privilege_type	character_data	特权类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO
with_hierarchy	yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION是一个独立的（子）特权，它允许在表继承层级上的特定操作。在PostgreSQL 中，这被包括在SELECT特权中，因此这一列在特权为SELECT时显示YES，其他时候显示NO。

37.52. tables

视图tables包含定义在当前数据库中的所有表和视图。只有那些当前用户能够访问（作为拥有者或具有某些特权）的表和视图才会被显示。

表 37.50.tables列

名称	数据类型	描述
table_catalog	sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名
table_name	sql_identifier	表名称
table_type	character_data	该表的类型：BASE TABLE表示一个持久的基本表（常见表类型），VIEW表示一个视图，FOREIGN表示一个外部表，LOCAL TEMPORARY表示一个临时表
self_referencing_column_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
reference_generation	character_data	应用于一个PostgreSQL中不可用的特性
user_defined_type_catalog	sql_identifier	如果该表是一个有类型的表，则是包含其底层数据类型的数据库名（总是当前数据库），否则为空。

名称	数据类型	描述
user_defined_type_schema	sql_identifier	如果该表是一个有类型的表，则是包含其底层数据类型的模式名，否则为空。
user_defined_type_name	sql_identifier	如果该表是一个有类型的表，则是其底层数据类型的名称，否则为空。
is_insertable_into	yes_or_no	如果该表能够被插入，则为YES，否则为NO（基本表总是能被插入，而视图则不一定）。
is_typed	yes_or_no	如果该表是一个有类型的表，则为YES，否则为NO
commit_action	character_data	还未被实现

37.53. transforms

视图transforms包含定义在当前数据库中的转换的信息。更准确 来说， 包含在转换中的每一个函数（“FROM SQL”或者 “TO SQL”函数）在其中都有一行。

表 37.51.transforms 列

名称	数据类型	描述
udt_catalog	sql_identifier	包含该转换所适用类型的数据库的名称（总是当前数据库）
udt_schema	sql_identifier	包含该转换所适用类型的模式的名称
udt_name	sql_identifier	该转换所适用类型的名称
specific_catalog	sql_identifier	包含该函数的数据库的名称（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式的名称
specific_name	sql_identifier	该函数的“专用名”。更多信息请见第 37.40 节
group_name	sql_identifier	SQL 标准允许在“组”中定义转换，并且在运行时选择一个 组。PostgreSQL 不支持这种做法，转换是与一种语言相关的。作为一种折衷， 这个域包含该转换所适用的语言。
transform_type	character_data	FROM SQL或者TO SQL

37.54. triggered_update_columns

对于当前数据库中指定一个列列表（如UPDATE OF column1, column2）的触发器，视图triggered_update_columns标识这些列。没有指定一个列列表的触发器不被包括在这个视图中。只有那些当前用户拥有或具有某种除SELECT之外特权的列才会被显示。

表 37.52.triggered_update_columns列

名称	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名
trigger_name	sql_identifier	该触发器的名称
event_object_catalog	sql_identifier	包含触发器所在的表的数据库名（总是当前数据库）
event_object_schema	sql_identifier	包含触发器所在的表的模式名
event_object_table	sql_identifier	触发器所在的表的名称
event_object_column	sql_identifier	触发器所在的列的名称

37.55. triggers

视图triggers包含所有定义在当前数据库中表和视图上的触发器，并且只显示当前用户拥有的触发器或者是当前用户在其上具有某种除SELECT之外特权的触发器。

表 37.53.triggers列

名称	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名
trigger_name	sql_identifier	该触发器的名称
event_manipulation	character_data	触发该触发器的事件（INSERT、UPDATE或DELETE）
event_object_catalog	sql_identifier	包含触发器所在的表的数据库名（总是当前数据库）
event_object_schema	sql_identifier	包含该触发器所在的表的模式名
event_object_table	sql_identifier	该触发器所在的表的名称
action_order	cardinal_number	同一个表上具有相同event_manipulation、action_timing和acti触发器之间的触发顺序。在PostgreSQL中，触发器按照名称顺序被触发，因此这一列会反映这种规则。
action_condition	character_data	触发器的WHEN条件，如果没有则为空（如果该表不被一个当前已启用角色拥有也是为空）
action_statement	character_data	该触发器执行的语句（当前总是 EXECUTE FUNCTION function(...)）
action_orientation	character_data	标识触发器是对每个被处理的行触发一次还是为每个语句触发一次（ROW或STATEMENT）

名称	数据类型	描述
action_timing	character_data	触发器在什么时候触发（BEFORE、 AFTER或INSTEAD OF）
action_reference_old_table	sql_identifier	“旧”传递表的名称，如果没有则为空
action_reference_new_table	sql_identifier	“新”传递表的名称，如果没有则为空
action_reference_old_row	sql_identifier	应用于一个PostgreSQL中不可用的特性
action_reference_new_row	sql_identifier	应用于一个PostgreSQL中不可用的特性
created	time_stamp	应用于一个PostgreSQL中不可用的特性

PostgreSQL中的触发器有两点与 SQL 标准不兼容，这会影响在该信息模式中的表示。第一，在PostgreSQL中触发器的名字是局限于每个表的，而不是独立于模式对象。因此可能在一个模式中会有重复的触发器名称，只要它们属于不同的表（trigger_catalog和trigger_schema才真正标识了触发器被定义在哪个表上）。第二，在PostgreSQL中触发器可以被定义为在多个事件上触发（例如ON INSERT OR UPDATE），而在SQL 标准中只允许一个。如果一个触发器被定义为在多个事件上触发，它在信息模式中被表示为多行，每一行对应于一类事件。作为这两个问题的结果，视图triggers的主键实际上是(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)，而不是(trigger_catalog, trigger_schema, trigger_name)（这是SQL 标准指定的）。尽管如此，如果你以符合 SQL 标准（在模式中触发器名称唯一并且每个触发器只能有一种事件类型）的方式定义你的触发器，这将不会影响你。

> **注意**
>
> 在PostgreSQL 9.1 之前，这个视图的列 action_timing、action_reference_old_table、action_reference_new_table、action_reference_old_row和 action_reference_new_row 分别被命名为 condition_timing、condition_reference_old_table、condition_reference_new_table、condition_reference_old_row和condition_reference_new_row。 那也是它们在 SQL:1999 标准中的命名。新的命名遵循 SQL:2003 及其后的版本。

37.56. udt_privileges

视图udt_privileges标识所有在用户定义类型上授予的 USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。 对每一个类型、授予者和被授予者的组合都有一行。这个视图只显示组合类型（原因见下面的 第 37.58 节。 域特权见第 37.57 节

表 37.54.udt_privileges列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
udt_catalog	sql_identifier	包含该类型的数据库名（总是当前数据库）
udt_schema	sql_identifier	包含该类型的模式名
udt_name	sql_identifier	该类型的名字

名称	数据类型	描述
privilege_type	character_data	总是TYPE USAGE
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.57. usage_privileges

视图usage_privileges标识所有在多种对象上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。在PostgreSQL中，这当前适用于排序规则、域、外部数据包装器、外部服务器和序列。对每一个对象、授予者和被授予者都有一行。

由于在PostgreSQL中排序规则并没有真正的特权，这个视图对所有排序规则显示由拥有者授予给PUBLIC的隐式非可授予的USAGE特权。但是对其他对象类型则显示真实的特权。

在 PostgreSQL 中，序列也支持除USAGE之外的SELECT和UPDATE特权。这些是非标准的并且因此在该信息模式中不可见。

表 37.55.usage_privileges列

名称	数据类型	描述
grantor	sql_identifier	授予该特权的角色名
grantee	sql_identifier	被授予该特权的角色名
object_catalog	sql_identifier	包含该对象的数据库名（总是当前数据库）
object_schema	sql_identifier	如果适用，则是包含该对象的模式名，否则为一个空字符串
object_name	sql_identifier	该对象的名称
object_type	character_data	COLLATION或DOMAIN或FOREIGN DATA WRAPPER或FOREIGN SERVER或SEQUENCE
privilege_type	character_data	总是USAGE
is_grantable	yes_or_no	如果该特权是可授予的，则为YES，否则为NO

37.58. user_defined_types

视图user_defined_types目前包含定义在当前数据库中的所有组合类型。只有那些当前用户能够访问（作为拥有者或具有某些特权）的类型才会被显示。

SQL 知道两种用户定义类型：结构类型（在PostgreSQL中也被称为组合类型）以及独特类型（在PostgreSQL没有实现）。要经得起未来的考验，请使用列user_defined_type_category来区分它们。其他用户定义类型如基类型和枚举（都是PostgreSQL的扩展）不会被显示在这里。对于域，请见第 37.22 节

表 37.56.user_defined_types列

名称	数据类型	描述
user_defined_type_catalog	sql_identifier	包含该类型的数据库名（总是当前数据库）
user_defined_type_schema	sql_identifier	包含该类型的模式名
user_defined_type_name	sql_identifier	该类型的名字
user_defined_type_category	character_data	当前总是STRUCTURED

名称	数据类型	描述
is_instantiable	yes_or_no	应用于一个PostgreSQL中不可用的特性
is_final	yes_or_no	应用于一个PostgreSQL中不可用的特性
ordering_form	character_data	应用于一个PostgreSQL中不可用的特性
ordering_category	character_data	应用于一个PostgreSQL中不可用的特性
ordering_routine_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
ordering_routine_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
ordering_routine_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
reference_type	character_data	应用于一个PostgreSQL中不可用的特性
data_type	character_data	应用于一个PostgreSQL中不可用的特性
character_maximum_length	cardinal_number	应用于一个PostgreSQL中不可用的特性
character_octet_length	cardinal_number	应用于一个PostgreSQL中不可用的特性
character_set_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
character_set_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_catalog	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_schema	sql_identifier	应用于一个PostgreSQL中不可用的特性
collation_name	sql_identifier	应用于一个PostgreSQL中不可用的特性
numeric_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性
numeric_precision_radix	cardinal_number	应用于一个PostgreSQL中不可用的特性
numeric_scale	cardinal_number	应用于一个PostgreSQL中不可用的特性
datetime_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性
interval_type	character_data	应用于一个PostgreSQL中不可用的特性
interval_precision	cardinal_number	应用于一个PostgreSQL中不可用的特性

名称	数据类型	描述
source_dtd_identifier	sql_identifier	应用于一个PostgreSQL中不可用的特性
ref_dtd_identifier	sql_identifier	应用于一个PostgreSQL中不可用的特性

37.59. user_mapping_options

视图user_mapping_options包含在当前数据库中为用户映射定义的所有选项。只有那些当前用户能够访问其相应外部服务器（作为拥有者或具有某些特权）的用户映射才会被显示。

表 37.57.user_mapping_options列

名称	数据类型	描述
authorization_identifier	sql_identifier	被映射的用户名，如果映射是公共的则为PUBLIC
foreign_server_catalog	sql_identifier	这个映射所使用的外部服务器所在的数据库名（总是当前数据库）
foreign_server_name	sql_identifier	这个映射所使用的外部服务器的名称
option_name	sql_identifier	一个选项名
option_value	character_data	选项的值。除非当前用户是被映射的用户或者映射是PUBLIC的并且当前用户是服务器拥有者或者超级用户，这一列将显示为空。这样做的目的是保护作为用户映射选项存储的口令信息。

37.60. user_mappings

视图user_mappings包含定义在当前数据库中的所有用户映射。只有当前用户能够访问其对应外部服务器（作为拥有者或具有某些特权）的用户映射才会被显示。

表 37.58.user_mappings列

名称	数据类型	描述
authorization_identifier	sql_identifier	被映射的用户名，如果映射是公共的则为PUBLIC
foreign_server_catalog	sql_identifier	这个映射所使用的外部服务器所在的数据库名（总是当前数据库）
foreign_server_name	sql_identifier	这个映射所使用的外部服务器的名称

37.61. view_column_usage

视图view_column_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有列。只有当包含一列的表被一个当前已被启用角色拥有时，该列才会被包括在这个视图中。

> **注意**
>
> 系统表列不被包括。在某个时候这应该会被修复。

表 37.59. view_column_usage列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名
view_name	sql_identifier	该视图的名称
table_catalog	sql_identifier	被视图所使用的列所属表的数据库名（总是当前数据库）
table_schema	sql_identifier	被视图所使用的列所属表的模式名
table_name	sql_identifier	被视图所使用的列所属表的名称
column_name	sql_identifier	被该视图所使用的列名

37.62. view_routine_usage

视图view_routine_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有例程（函数和过程）。只有被一个当前已被启用角色拥有的例程才会被包括在这个视图中。

表 37.60. view_routine_usage列

名称	数据类型	描述
table_catalog	sql_identifier	包含该视图的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该视图的模式名
table_name	sql_identifier	该视图的名称
specific_catalog	sql_identifier	包含该函数的数据库名（总是当前数据库）
specific_schema	sql_identifier	包含该函数的模式名
specific_name	sql_identifier	该函数的“专用名”。详见第 37.40 节

37.63. view_table_usage

视图view_table_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有表。只有被一个当前已被启用角色拥有的表才会被包括在这个视图中。

> **注意**
>
> 系统表没有被包括。这应当会在某个时候被修复。

表 37.61.view_table_usage列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名
view_name	sql_identifier	该视图的名称
table_catalog	sql_identifier	包含被该视图所使用的表的数据库名（总是当前数据库）
table_schema	sql_identifier	包含被该视图所使用的表的模式名
table_name	sql_identifier	包含被该视图所使用的表的名称

37.64. views

视图views包含定义在当前数据库中的所有视图。只有当前用户能够访问（作为拥有者或具有某些特权）的视图才会被显示。

表 37.62.views列

名称	数据类型	描述
table_catalog	sql_identifier	包含该视图的数据库名（总是当前数据库）
table_schema	sql_identifier	包含该视图的模式名
table_name	sql_identifier	该视图的名称
view_definition	character_data	定义视图的查询表达式（如果该视图不被一个当前已被启用角色拥有则为空）
check_option	character_data	应用于一个PostgreSQL中不可用的特性
is_updatable	yes_or_no	如果该视图是可更新的（允许UPDATE和DELETE），则为YES，否则为NO
is_insertable_into	yes_or_no	如果该视图是可插入的（允许INSERT），则为YES，否则为NO
is_trigger_updatable	yes_or_no	如果该视图上有一个INSTEAD OF UPDATE触发器，则为YES，否则为NO
is_trigger_deletable	yes_or_no	如果该视图上有一个INSTEAD OF DELETE触发器，则为YES，否则为NO
is_trigger_insertable_into	yes_or_no	如果该视图上有一个INSTEAD OF INSERT触发器，则为YES，否则为NO

部分 V. 服务器编程

这部分关于使用用户定义的函数、数据类型、触发器等扩展服务器功能。这些是高级主题，读者应该在理解了有关PostgreSQL的所有其他用户文档之后才阅读这些主题。这一部分的后面一些章节描述PostgreSQL发布中可用的服务器端编程语言，以及与服务器端编程语言相关的一般性问题。在深入研究服务器端编程语言的材料之前，请至少阅读第 38 章前几节（覆盖函数）。

目录

第 38 章 扩展 SQL

在下面的小节中，我们将讨论如何通过增加各种元素来扩展PostgreSQL SQL 查询语言：

- 函数（从第 38.3 节开始）
- 聚集（从第 38.11 节开始）
- 数据类型（从第 38.12 节开始）
- 操作符（从第 38.13 节开始）
- 用于索引的操作符类（从第 38.15 节开始）
- 相关对象的包（从第 38.16 节开始）

38.1. 扩展性如何工作

PostgreSQL是可扩展的，因为它的操作是目录驱动的。如果你熟悉标准的关系型数据库系统，你会知道它们把有关数据库、表、列等的信息存储在总所周知的系统目录中（某些系统称之为数据目录）。目录对于用户来说好像其他的表一样，但是DBMS把自己的内部信息记录在其中。PostgreSQL和标准关系型数据库系统的一个关键不同是PostgreSQL在其目录中存储更多信息：不只是有关表和列的信息，还有关于数据类型、函数、访问方法等等的信息。这些表可以被用户修改，并且因为PostgreSQL的操作是基于这些表的，所以PostgreSQL可以被用户扩展。通过比较，传统数据库系统只能通过在源代码中改变硬编码的过程或者载入由DBMS提供者特殊编写的模块进行扩展。

此外，PostgreSQL服务器能够通过动态载入把用户编写的代码结合到它自身中。也就是，用户能够指定一个实现了一个新类型或函数的对象代码文件（例如一个共享库），并且PostgreSQL将按照要求载入它。把用SQL编写的代码加入到服务器会更繁琐。这种“即时”修改其操作的能力让PostgreSQL独特地适合新应用和存储结构的快速原型设计。

38.2. PostgreSQL类型系统

PostgreSQL数据类型被划分为基础类型、容器类型、域和伪类型。

38.2.1. 基础类型

基础类型是那些被实现在SQL语言层面之下的类型（通常用一种底层语言，如 C），例如integer。它们通常对应于常说的抽象数据类型。PostgreSQL只能通过由用户提供的函数在这类类型上操作，并且只能理解到用户描述这种类型行为的程度。第 8 章描述了内建的基础类型。

枚举（enum）类型可以被认为是基础类型的一个子类。主要区别是它们可以使用SQL命令创建，不需要用到底层的编程。更多信息请参考第 8.7 节

38.2.2. 容器类型

PostgreSQL有三种“容器”类型，它们是包含多个其他类型值的类型。它们是数组、组合以及范围。

数组可以保存全部是同种类型的多个值。为每一种基本类型、组合类型、范围类型以及域类型都会自动创建一个数组类型。但是没有数组的数组。就类型系统的认知而言，多维数组就和一维数组一样。更多信息请参考第 8.15 节

只要用户创建一个表，就会创建组合类型或者行类型。也可以使用CREATE TYPE来定义一个没有关联表的“stand-alone”组合类型。一个组合类型只是一个具有相关域名称的类型列表。一个组合类型的值是一个行或者域值记录。用户可以访问来自SQL查询的组成域。更多信息请参考第 8.16 节

范围类型可以保存同种类型的两个值，它们是该范围的上下界。范围类型是用户创建的，不过也存在一些内建的范围类型。更多信息请参考第 8.17 节

38.2.3. 域

一个域是基于一种特定底层类型的，并且出于很多目的它可以与其底层类型互换。不过，一个域能够具有约束来限制它的合法值于其底层基础类型允许值的一个子集。可以使用SQL命令CREATE DOMAIN创建域。更多信息请参考第 8.18 节

38.2.4. 伪类型

有一些用于特殊目的“伪类型”。伪类型不能作为表列或者容器类型的组件出现，但是它们能被用于声明函数的参数和结果类型。这在类型系统中提供了一种机制来标识函数的特殊分类。表 8.25列出了现有的伪类型。

38.2.5. 多态类型

特别让人感兴趣的五种伪类型是anyelement、anyarray、anynonarray、anyenum以及anyrange，它们被统称为多态类型。任何使用这些类型声明的函数被称作是一个多态函数。通过使用根据一次特定调用实际传递的数据类型所决定的相关数据类型，一个多态函数能够在多种不同数据类型上操作。

多态参数和结果是相互关联的，并且它们在解析调用多态函数的查询时被决定到一种特定的数据类型。每一个被声明为anyelement的位置（参数或返回值）被允许具有任意特定的实际数据类型，但是在任何给定的查询中它们必须全部是相同的实际类型。每一个被声明为anyarray的位置可以有任意数组数据类型，但是相似地，它们必须全部具有相同类型。并且类似地，被声明为anyrange的位置必须是全部是相同的范围类型。此外，如果有位置被声明为anyarray并且其他位置被声明为anyelement，anyarray位置中的实际数组类型必须是一个数组，该数组的元素都是出现在anyelement位置的同一种类型。相似地，如果有位置被声明为anyrange并且其他位置被声明为anyelement，anyrange位置的实际范围类型必须是一个范围，该范围的子类型是出现在anyelement位置的同一种类型。anynonarray被当做和anyelement相同，但是增加了额外的约束要求实际类型不能是一种数组类型。anyenum被当做和anyelement相同，但是增加了额外的约束要求实际类型不能是一种枚举类型。

因此，当使用一种多态类型声明了多于一个参数位置，有效效果是只有实际参数类型的某些组合才被允许。例如，一个被声明为equal(anyelement, anyelement)的函数将要求任意两个输入值，只要它们是同一种数据类型。

当一个函数的返回值被声明为多态类型时，必须至少有一个参数位置也是多态的，并且作为该参数提供的实际数据类型决定了该调用的实际结果类型。例如，如果还没有一种数组下标机制，我们可以定义一个函数来实现下标：subscript(anyarray, integer) returns anyelement。这个声明约束了实际的第一个参数是一种数组类型，并且允许解析器从实际的第一个参数类型推断正确的结果类型。另一个例子是一个被声明为f(anyarray) returns anyenum的函数将只接受枚举类型的数组。

注意anynonarray和anyenum并不表示独立的类型变量，它们是和anyelement相同的类型，只是有一个额外的约束。例如，将一个函数声明为f(anyelement, anyenum)等效于把它声明为f(anyenum, anyenum)：两种实际参数必须是相同的枚举类型。

一个可变函数（可以有可变数量的参数，如第 38.5.5 节所述）能够是多态的：这可以通过声明其最后一个参数为VARIADIC anyarray来实现。为了匹配和决定实际结果类型的参数，这样一种函数的行为和写了合适数量的anynonarray参数是一样的。

38.3. 用户定义的函数

PostgreSQL提供四种函数：

- 查询语言函数（用SQL编写的函数）（第 38.5 节
- 过程语言函数（例如，用PL/pgSQL或PL/Tcl编写的函数）（第 38.8 节

- 内部函数（第 38.9 节
- C 语言函数（第 38.10 节

每一类函数可以采用基本类型、组合类型或者它们的组合作为参数。此外，每一类函数可以返回一个基本类型或一个组合类型。函数也能被定义成返回基本类型或组合类型值的集合。

很多类函数可以接受或者返回特定的伪类型（例如，多态类型），但是可用的功能会变化。详情可以参考每一种函数的描述。

定义SQL函数最容易，因此我们将从讨论SQL函数开始。大部分SQL函数的概念也能用到其他类型的函数上。

在这一章中，查看CREATE FUNCTION命令的参考页有助于更好地理解例子。这章中的一些例子可以在PostgreSQL源代码发布的src/tutorial目录中的funcs.sql和funcs.c中找到。

38.4. 用户定义的过程

过程是一种类似于函数的数据库对象。两者的区别在于过程不返回值，因此没有返回类型声明。而函数可以作为一个查询或者DML命令的一部分被调用，过程则需要明确地用CALL语句调用。

本章剩余部分中对如何定义用户定义的函数的解释同样适用于过程，不同的地方有：需要使用CREATE PROCEDURE命令定义、没有返回类型、一些如严格性这样的其他特性不适用。

函数和过程一起构成了例程。有ALTER ROUTINE以及DROP ROUTINE这样的命令可以操作函数和过程而不需要知道它们是哪一种。不过，要注意没有CREATE ROUTINE命令。

38.5. 查询语言（SQL）函数

SQL 函数执行一个由任意 SQL 语句构成的列表，返回列表中最后一个查询的结果。在简单（非集合）的情况中，最后一个查询的结果的第一行将被返回（记住一个多行结果的“第一行”不是良定义的，除非你使用ORDER BY）。如果最后一个查询正好根本不返回行，将会返回空值。

或者，一个 SQL 函数可以通过指定函数的返回类型为SETOF sometype被声明为返回一个集合（也就是多个行），或者等效地声明它为RETURNS TABLE(columns)。在这种情况下，最后一个查询的结果的所有行会被返回。下文将给出进一步的细节。

一个 SQL 函数的主体必须是一个由分号分隔的 SQL 语句的列表。最后一个语句之后的分号是可选的。除非函数被声明为返回void，最后一个语句必须是一个SELECT或者一个带有RETURNING子句的INSERT、UPDATE或者DELETE。

SQL语言中的任何命令集合都能被打包在一起并且被定义成一个函数。除了SELECT查询，命令可以包括数据修改查询（INSERT、UPDATE以及DELETE）和其他 SQL 命令（你不能在SQL函数中使用事务控制命令，例如COMMIT、SAVEPOINT，以及一些工具命令，例如VACUUM）。不过，最后一个命令必须是一个SELECT或者带有一个RETURNING子句，该命令必须返回符合函数返回类型的数据。或者，如果你想要定义一个执行动作但是不返回有用的值的函数，你可以把它定义为返回void。例如，这个函数从emp表中移除具有负值薪水的行：

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------
```

(1 row)

> **注意**
>
> 在被执行前，SQL 函数的整个主体都要被解析。虽然 SQL 函数可以包含修改系统目录的命令（如CREATE TABLE），但这类命令的效果对于该函数中后续命令的解析分析不可见。例如，如果把CREATE TABLE foo (...); INSERT INTO foo VALUES(...);打包到一个 SQL 函数中是得不到预期效果的，因为在解析INSERT命令时foo还不存在。在这类情况下，推荐使用PL/pgSQL而不是 SQL 函数。

CREATE FUNCTION命令的语法要求函数体被写作一个字符串常量。使用用于字符串常量的美元引用通常最方便（见第 4.1.2.4 节。你过你选择使用常规的单引号引用的字符串常量语法，你必须在函数体中双写单引号（’）和反斜线（\）（假定转义字符串语法）（见第 4.1.2.1 节。

38.5.1. SQL函数的参数

一个 SQL 函数的参数可以在函数体中用名称或编号引用。下面会有两种方法的例子。

要使用一个名称，将函数参数声明为带有一个名称，然后在函数体中只写该名称。如果参数名称与函数内当前 SQL 命令中的任意列名相同，列名将优先。如果不想这样，可以用函数本身的名称来限定参数名，也就是function_name.argument_name（如果这会与一个被限定的列名冲突，照例还是列名赢得优先。你可以通过为 SQL 命令中的表选择一个不同的别名来避免这种混淆）。

在更旧的数字方法中，参数可以用语法$n引用：$1指的是第一个输入参数，$2指的是第二个，以此类推。不管特定的参数是否使用名称声明，这种方法都有效。

如果一个参数是一种组合类型，那么点号记法（如 argname.fieldname 或$1.fieldname）也可以被用来 访问该参数的属性。同样，你可能需要用函数的名称来限定参数的名称以避免歧义。

SQL 函数参数只能被用做数据值而不能作为标识符。例如这是合理的：

```
INSERT INTO mytable VALUES ($1);
```

但这样就不行：

```
INSERT INTO $1 VALUES (42);
```

> **注意**
>
> 使用名称来引用 SQL 函数参数的能力是在PostgreSQL 9.2 中加入的。要在老的服务器中使用的函数必须使用$n记法。

38.5.2. 基本类型上的SQL

最简单的SQL函数没有参数并且简单地返回一个基本类型，例如integer：

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
```

```
$$ LANGUAGE SQL;

-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

注意我们为该函数的结果在函数体内定义了一个列别名（名为result），但是这个列别名在函数以外是不可见的。因此，结果被标记为one而不是result。

定义用基本类型作为参数的SQL函数也很容易：

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

我们也能省掉参数的名称而使用数字：

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

这里是一个更有用的函数，它可以被用来借记一个银行账号：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

一个用户可以这样执行这个函数来从账户 17 中借记 $100.00：

```
SELECT tf1(17, 100.0);
```

在这个例子中，我们为第一个参数选择了名称accountno，但是这和表bank中的一个列名相同。 在UPDATE命令中， accountno引用列bank.accountno，因此 tf1.accountno必须被用来引用该参数。 我们当然可以通过为该参数使用一个不同的名称来避免这样的问题。

实际上我们可能喜欢从该函数得到一个更有用的结果而不是一个常数 1，因此一个更可能的定义是：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

它会调整余额并且返回新的余额。 同样的事情也可以用一个使用RETURNING的命令实现：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

SQL函数必须恰好返回其声明的结果类型。这可能会要求插入一个显式的造型。例如，假设我们想要之前的add_em函数返回类型float8。下面的做法是不行的：

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

即便在其他的环境中PostgreSQL也会想要插入一个隐式造型把integer转换成float8。我们需要把它写成

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT ($1 + $2)::float8;
$$ LANGUAGE SQL;
```

38.5.3. 组合类型上的SQL函数

在编写使用组合类型参数的函数时，我们必须不仅指定我们想要哪些参数，还要指定参数的期望属性（域）。例如，假定 emp是一个包含雇员数据的表，并且因此它也是该表每一行的组合类型的名称。 这里是一个函数double_salary，它计算某个人的双倍薪水：

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

```
 name | dream
------+-------
 Bill |  8400
```

注意语法$1.salary的使用是要选择参数行值的一个域。 还要注意调用的SELECT命令是如何使用table_name.*来选择一个表的整个当前行作为一个组合值的。该表行也可以只用表名来引用：

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

但这种用法已被废弃因为它很容易让人搞混（关于表行的组合值的这两种记法的详细情况请见第 8.16.5 节。

有时候实时构建一个组合参数很方便。这可以用ROW结构完成。 例如，我们可以调整被传递给函数的数据：

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

也可以构建一个返回组合类型的函数。这是一个返回单一emp行的函数例子：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

在这个例子中，我们为每一个属性指定了一个常量值，但是可以用任何计算来替换这些常量。

有关定义函数有两件重要的事情：

- 查询中的选择列表顺序必须和列在与组合类型相关的表中出现的顺序完全相同（如我们上面所作的，列的命名与系统无关）。
- 我们必须确保每个表达式的类型匹配该组合类型相应的列，必要时插入一个造型。否则我们将会得到像这样的错误：

```
ERROR:  function declared to return emp returns varchar instead of text at
 column 1
```

与基础类型的情况一样，该函数将不会自动插入任何造型。

定义同样的函数的一种不同的方法是：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

这里我们写了一个只返回正确组合类型的单一列的SELECT。 在这种情况下这种写法实际并非更好，但是它在一些情况下比较方便 — 例如，我们需要通过调用另一个返回所期望的组

合值的函数来计算结果。另一个例子是，如果我们尝试写一个函数返回组合上的一个域，而不是返回纯粹的组合类型，那么总是有必要把它写成返回单一列，因为没有其他办法能够产生一个正好是那种域类型的值。

我们可以直接调用这个函数或者在一个值表达式中使用它：

```
SELECT new_emp();

          new_emp
--------------------------
 (None,1000.0,25,"(2,2)")
```

或者把它当做一个表函数调用：

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

第二种方式在第 38.5.7 节有更完全的描述。

当你使用一个返回组合类型的函数时，你可能只想要其结果中的一个域（属性）。 你可以这样做：

```
SELECT (new_emp()).name;

 name
------
 None
```

额外的圆括号是必须的，它用于避免解析器被搞混。如果你不写这些括号，会这样：

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

另一个选项是使用函数记号来抽取一个属性：

```
SELECT name(new_emp());

 name
------
 None
```

如第 8.16.5 节所说，字段记法和函数记法是等效的。

另一种使用返回组合类型的函数的方法是把结果传递给另一个接收正确行类型作为输入的函数：

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;
```

```
SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

38.5.4. 带有输出参数的SQL函数

一种描述一个函数的结果的替代方法是定义它的输出参数，例如：

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

这和第 38.5.2 节展示的add_em版本没有本质上的不同。输出参数的真正价值是它们提供了一种方便的方法来定义返回多个列的函数。例如，

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

这里实际发生的是我们为该函数的结果创建了一个匿名的组合类型。上述例子具有与下面相同的最终结果

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

但是不必单独定义组合类型常常很方便。注意输出参数的名称并非只是装饰，而且决定了匿名组合类型的列名（如果你为一个输出参数忽略了名称，系统将自行选择一个名称）。

在从 SQL 调用这样一个函数时，输出参数不会被包括在调用参数列表中。这是因为PostgreSQL只考虑输入参数来定义函数的调用签名。这也意味着在为诸如删除函数等目的引用该函数时只有输入参数有关系。我们可以用下面的命令之一删除上述函数

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

参数可以被标记为IN（默认）、OUT、INOUT或者VARIADIC。一个INOUT参数既作为一个输入参数（调用参数列表的一部分）又作为一个输出参数（结果记录类型的一部分）。VARIADIC参数是输入参数，但被按照后文所述特殊对待。

38.5.5. 带有可变数量参数的SQL函数

只要“可选的”参数都是相同的数据类型，SQL函数可以被声明为接受可变数量的参数。可选的参数将被作为一个数组传递给该函数。声明该函数时要把最后一个参数标记为VARIADIC，这个参数必须被声明为一个数组类型，例如：

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

实际上，所有位于或者超过VARIADIC位置的实参会被收集成一个一位数组，就好像你写了：

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- 不起作用
```

但是你实际无法这样写 — 或者说至少它将无法匹配这个函数定义。一个被标记为VARIADIC的参数匹配其元素类型的一次或者多次出现，而不是它自身类型的出现。

有时候能够传递一个已经构造好的数组给 variadic 函数是有用的，特别是当 一个 variadic 函数想要把它的数组参数传递给另一个函数时这会特别方便。此外，这是在一个允许不可信用户创建对象的方案中调用一个variadic函数的唯一安全的方式，见第 10.3 节你可以通过在调用中指定VARIADIC来做到这一点：

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

这会阻止该函数的 variadic 参数扩展成它的元素结构，从而允许数组参 数值正常匹配。VARIADIC只能被附着在函数调用的最后一 个实参上。

在调用中指定VARIADIC也是将空数组传递给 variadic 函数 的唯一方式，例如：

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

简单地写成SELECT mleast()是没有作用的，因为一个 variadic 参数必须匹配至少一个实参（如果想允许这类调用，你可以定义第二个没有 参数且也叫mleast的函数）。

从一个 variadic 参数产生的数组元素参数会被当做自己不具有名称。这 意味着不能使用命名参数调用 variadic 函数（第 4.3 节，除非你指定了 VARIADIC。例如下面的调用是可以工作的：

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

但这些就不行：

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

38.5.6. 带有参数默认值的SQL函数

函数可以被声明为对一些或者所有输入参数具有默认值。只要调用函数时 没有给出足够多的实参，就会插入默认值来弥补缺失的实参。由于参数只 能从实参列表的尾部开始被省

略，在一个有默认值的参数之后的所有参数　都不得不也具有默认值（尽管使用命名参数记法可以允许放松这种限制，　这种限制仍然会被强制以便位置参数记法能工作）。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施，见第　10.3　节

例如：

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- 因为第一个参数没有默认值，所以会失败
ERROR:  function foo() does not exist
```

=符号也可以用来替代关键词 DEFAULT。

38.5.7.　SQL 函数作为表来源

所有的　SQL　函数都可以被用在查询的FROM子句中，但是　对于返回组合类型的函数特别有用。如果函数被定义为返回一种基本类型，　该表函数会产生一个单列表。如果该函数被定义为返回一种组合类型，该 表函数会为该组合类型的每一个属性产生一列。

这里是一个例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
```

```
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

正如例子所示，我们可以把函数结果的列当作常规表的列来使用。

注意我们只从函数得到了一行。这是因为我们没有使用SETOF。 这会在下一节中介绍。

38.5.8. 返回集合的SQL函数

当一个 SQL 函数被声明为返回SETOF sometype时，该函数的 最后一个查询会被执行完，并且它输出的每一行都会被 作为结果集的一个元素返回。

在FROM子句中调用函数时通常会使用这种特性。在这种 情况下，该函数返回的每一行都变成查询所见的表的一行。例如，假设 表foo具有和上文一样的内容，并且我们做了以下动作：

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

那么我们会得到：

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

也可以返回多个带有由输出参数定义的列的行，像这样：

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

这里的关键点是必须写上RETURNS SETOF record来指示 该函数返回多行而不是一行。如果只有一个输出参数，则写上该参数的 类型而不是record。

通过多次调用集合返回函数来构建查询的结果非常有用，每次调用的参数 来自于一个表或者子查询的连续行。做这种事情最好的方法是使用 第 7.2.1.5 节描述的LATERAL关键词。这里是一个使用集合返回函数枚举树结构中元素的例子：

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name   |   child
---------+-----------
 Top     | Child1
 Top     | Child2
 Top     | Child3
 Child1  | SubChild1
 Child1  | SubChild2
(5 rows)
```

这个例子和我们使用的简单连接的效果没什么不同，但是在更复杂的　计算中，把一些工作放在函数中会是一种很方便的选项。

返回集合的函数也能在查询的选择列表中调用。对于该查询本身产生的每一行都会调用集合返回函数，并且会从该函数的结果集中的每一个元素生成一个输出行。之前的例子也可以用这样的查询实现：

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name   | listchildren
---------+--------------
 Top     | Child1
 Top     | Child2
 Top     | Child3
 Child1  | SubChild1
 Child1  | SubChild2
(5 rows)
```

在最后一个SELECT中，注意对于Child2、 Child3等没有出现输出行。这是因为listchildren 对这些参数返回空集，因此没有产生结果行。这和使用LATERAL 语法时，我们从与该函数结果的内连接得到的行为是一样的。

PostgreSQL中，写在查询的选择列表中的集合返回函数的行为几乎和写在LATERAL FROM子句项中的集合返回函数完全一样。例如：

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

几乎等效于

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

这会是完全一样的，除了在这个特别的例子中，规划器会选择把g放在嵌套循环连接的外侧，因为g对tab没有实际的横向依赖。那会导致一种不同的输出行顺序。选择列表中的集合返回函数总是会被计算，就好像它们在FROM子句剩余部分的嵌套循环连接的内侧一样，因此在考虑来自FROM子句的下一行之前，这些函数会运行到完成。

如果在查询的选择列表中有不止一个集合返回函数，则行为类似于把那些函数放到一个单一的LATERAL ROWS FROM(...) FROM子句项中的行为。对于来自底层查询的每一行，都有一个用到每个函数首个结果的输出行，然后是一个使用每个函数第二个结果的输出行，以此类推。如果某些集合返回函数产生的输出比其他函数少，会用空值代替缺失的数据，因此为一个底层行形成的总行数等于产生最多输出的集合返回函数的输出行数。因此集合返回函数会“步调一致”地运行直到它们的输出被耗尽，然后用下一个底层行继续执行。

集合返回函数可以被嵌套在一个选择列表中，不过在FROM子句项中不允许这样做。在这种情况下，嵌套的每一层会被单独对待，就像它是一个单独的LATERAL ROWS FROM(...)项一样。例如，在

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

中，集合返回函数srf2、srf3和srf5将为tab的每一行步调一致地运行，然后会对较低层的函数产生的每一行以步调一致的形式应用srf1和srf4。

在CASE或COALESCE这样的条件计算结构中，不能使用集合返回函数。例如，考虑

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

看起来这个语句应该产生满足x > 0的输入行的五次重复，以及不满足的行的一次重复。但实际上，由于在CASE表达时被计算前，generate_series(1, 5)会被运行在一个隐式的LATERAL FROM项中，它会为每个输入行产生五次重复。为了减少混乱，这类情况会产生一个解析时错误。

注意

如果函数的最后一个命令是带有RETURNING的 INSERT、UPDATE或者 DELETE，该命令将总是会被执行完，即使函数没有用 SETOF定义或者调用查询不要求取出所有结果行也是如此。 RETURNING子句产生的多余的行会被悄无声息地丢掉，但是 在命令的目标表上的修改仍然会发生（而且在从该函数返回前就会全部完成）。

注意

在PostgreSQL 10之前，把多个集合返回函数放在同一个选择列表中的行为并不容易察觉，除非它们总是产生同等的行数。否则，你得到的输出行数将会是

各集合返回函数产生的行数的最小公倍数。此外，嵌套的集合返回函数不会按照上述的方式工作。相反，一个集合返回函数只能有最多一个集合返回参数，集合返回函数的每一次嵌套会被独立运行。此外，条件执行（CASE等中的集合返回函数）以前是被允许的，但是会让事情更加复杂。在编写需要在较老的PostgreSQL版本中工作的查询时，推荐使用LATERAL语法，因为这种语法能够在不同的版本间提供一致的结果。如果有一个依赖于集合返回函数的条件执行，那么可能可以通过把条件测试移到一个自定义集合返回函数中来修正该问题。例如，

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM
 tab;
```

可以变成

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int,
 els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

这种表达形式将在所有版本的PostgreSQL中以相同的方式工作。

38.5.9. 返回TABLE的SQL函数

还有另一种方法可以把函数声明为返回一个集合，即使用 RETURNS TABLE(columns)语法。这等效于使用一个或者多个OUT参数外加把函数标记为返回 SETOF record（或者是SETOF单个输出参数的 类型）。这种写法是在最近的 SQL 标准中指定的，因此可能比使用 SETOF的移植性更好。

例如，前面的求和并且相乘的例子也可以这样来做：

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

不允许把显式的OUT或者INOUT参数用于 RETURNS TABLE记法 — 必须把所有输出列放在TABLE列表中。

38.5.10. 多态SQL函数

SQL函数可以被声明为接受并且返回多态类型 anyelement、anyarray、anynonarray、anyenum以及 anyrange。更多关于多态函数的解释请见第 38.2.5 节这里是一个从两种任意数据 类型的元素构建一个数组的多态函数make_array：

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
```

```
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

注意类型造型'a'::text的使用是为了指定该参数的类型 是text。如果该参数只是一个字符串这就是必须的，因为 否则它会被当作unknown类型，并且 unknown的数组也不是一种合法的类型。如果没有改类型 造型，将得到这样的错误：

```
ERROR:  could not determine polymorphic type because input has type "unknown"
```

允许具有多态参数和固定的返回类型，但是反过来不行。例如：

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one
 polymorphic argument.
```

多态化可以用在具有输出参数的函数上。例如：

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

多态化也可以用在 variadic 函数上。例如：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
```

```
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

38.5.11. 带有排序规则的SQL函数

当一个 SQL 函数具有一个或者更多可排序数据类型的参数时，按照 第 23.2 节所述，对每一次函数调用都会根据分 配给实参的排序规则为其确定一个排序规则。如果成功地确定（即在 参数之间没有隐式排序规则的冲突），那么所有的可排序参数都被认 为隐式地具有该排序规则。这将会影响函数中对排序敏感的操作的行 为。例如，使用上述的anyleast函数时，

```
SELECT anyleast('abc'::text, 'ABC');
```

的结果将依赖于数据库的默认排序规则。在C区域中， 结果将是ABC，但是在很多其他区域中它将是 abc。可以在任意参数上增加一个COLLATE 子句来强制要使用的排序规则，例如：

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

此外，如果你希望一个函数用一个特定的排序规则工作而不管用什么排序规则 调用它，可根据需要在函数定义中插入COLLATE子句。 这种版本的anyleast将总是使用en_US区域来比较字符串：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

但是注意如果应用到不可排序数据类型上，这将会抛出一个错误。

如果在实参之间无法确定共同的排序规则，那么 SQL 函数会把它的参数 当作拥有其数据类型的默认排序规则（通常是数据库的默认排序规则， 但是域类型的参数可能会不同）。

可排序参数的行为可以被想成是多态的一种受限形式，只对于文本数据 类型有效。

38.6. 函数重载

可以用同样的 SQL 名称定义多于一个函数，只要它们的参数不同即可。 换句话说，函数名可以被重载。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施，见第 10.3 节当一个查询 被执行时，服务器将从数据类型和所提供的参数个数来决定要调用哪个 函数。重载也可用来模拟具有可变参数个数（最大个数有限）的函数。

在创建一个重载函数家族时，应该小心不要创建歧义。例如，给定函数：

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

对于test(1, 1.5)这样的输入就无法立刻清楚地知道 应该调用哪个函数。当前实现的解决规则在第 10 章中有描述，但是设计一个依赖于这种行为的系统是不明智的。

一个具有单个组合类型参数的函数通常不应与该类型的任何属性（域） 重名。回想一下，attribute(table)被认为等效于 table.attribute。在出现“一个组合类型 上的函数”与“组合类型的一个属性”的情况下，将总是使用属性。 可以通过用模式限定该函数名（即 schema.func(table) ）来覆盖这种选择，但是最好 不要选择有冲突的名称以避免此类问题。

另一种可能的冲突在于 variadic 和非 variadic 函数之间。例如，可以创建foo(numeric)和foo(VARIADIC numeric[])。 在这种情况下，对于提供了一个数字参数的调用（例如foo(10.1)） 就不清楚应该匹配哪一个函数。规则是使用在搜索路径中出现得较早的 函数，或者当两者都在同一个模式中时优先使用非 variadic 的那一个函数。

在重载 C 语言函数时有一个额外的约束：重载函数家族中的每一个 函数的 C 名称必须与其他所有函数的 C 名称不同，不管是内部的 还是动态载入的。如果这条规则被违背，该行为将不可移植。你可能 会得到一个运行时链接器错误，或者这些函数之一将被调用（通常 是内部的那一个）。SQL CREATE FUNCTION命令的AS子句的另一种形式 可以把 SQL 函数名和 C 源代码中的函数名分离。例如：

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

这里的 C 函数名称反映了很多种可能的习惯之一。

38.7. 函数易变性分类

每一个函数都有一个易变性分类，可能是 VOLATILE、STABLE或者IMMUTABLE。 如果CREATE FUNCTION命令没有指定一个分类，则默认是 VOLATILE。易变性分类是给优化器的关于该函数行为的一种承诺：

- 一个VOLATILE函数可以做任何事情，包括修改数据库。在 使用相同的参数连续调用时，它能返回不同的结果。优化器不会对这类函 数的行为做任何假定。在每一行需要 volatile 函数值时，一个使用 volatile 函数的查询都会重新计算该函数。

- 一个STABLE函数不能修改数据库并且被确保对一个语句中 的所有行用给定的相同参数返回相同的结果。这种分类允许优化器把该函 数的多个调用优化成一个调用。特别是，在一个索引扫描条件中使用包含 这样一个函数的表达式是安全的（因为一次索引扫描只会计算一次比较值， 而不是为每一行都计算一次，在一个索引扫描条件中不能使用VOLATILE函数）。

- 一个IMMUTABLE函数不能修改数据库并且被确保用相同的参数 永远返回相同的结果。这种分类允许优化器在一个查询用常量参数调用该函数 时提前计算该函数。例如，一个 SELECT ... WHERE x = 2 + 2这样的查询可以被简化为 SELECT ... WHERE x = 4，因为整数加法操作符底层的函数被 标记为IMMUTABLE。

为了最好的优化结果，你应该把函数标记为对它们合法的易变性分类中最严格 的那种。

任何带有副作用的函数必须被标记为VOLATILE， 这样对它的调用就不能被优化掉。甚至如果一个函数的值在一个查询中会 变化，即使它没有副作用也需要被标记为VOLATILE。这样的 例子有random()、currval()、 timeofday()等。

另一种重要的例子是current_timestamp家族的函数有资格 被标记为STABLE，因为它们的值在一个事务中不会改变。

在考虑先规划然后立即执行的简单交互式查询时，在STABLE和 IMMUTABLE分类间的区别相对较小：一个函数是在规划时只 执行一次还是在查询执行开始期间只执行一次没有太大关系。但是如果计划 被保存下来然后在后面被重用，区别就大了。如果在不允许过早把一个函数 变成规划期间的一个常数时把它标记为IMMUTABLE，会导致 在后续重用该计划时用到一个陈旧的值。当使用预备语句或者使用会缓存计 划的函数语言（PL/pgSQL）时，这就会是一种灾难。

对于用 SQL 或者其他任何标准过程语言编写的函数，还有第二种由易变性分类 决定的特性，即由调用该函数的 SQL 命令所作的数据修改的可见性。 VOLATILE函数将看到这些更改，STABLE 或者IMMUTABLE函数则看不到。这种行为使用 MVCC 的快照 行为（见第 13 章实现：STABLE和 IMMUTABLE函数使用一个在调用查询开始时建立的快照，而 VOLATILE函数在它们执行的每一个查询的开始都获得一个新鲜 的快照。

> **注意**
>
> 用 C 编写的函数按照它们自己需要的方式管理快照，但是通常最好 让 C 函数也按照上面的方式来。

由于这种快照行为，一个只包含SELECT命令的函数可以被 安全地标记为STABLE，即便它选择的表可能正在被并发查询所 修改。PostgreSQL将使用为调用查询所 建立的快照来执行STABLE函数中的所有命令，因此它将在整个 查询期间看到一种数据库的固定视图。

对IMMUTABLE函数中的SELECT使用了相同 的快照行为。通常在一个IMMUTABLE函数中从数据库表选择是 不明智的，因为如果表内容变化就会破坏不变性。不过， PostgreSQL不会强制不让你这样做。

一种常见的错误是当一个函数的结果依赖于一个配置参数时把它标记为 IMMUTABLE。例如，一个操纵时间戳的函数有可能结果 依赖于TimeZone设置。为了安全起见，这类 函数应该被标记为STABLE。

> **注意**
>
> PostgreSQL要求STABLE 和IMMUTABLE函数中不包含非SELECT 的 SQL 命令以阻止数据修改（这也不是完全万无一失，因为这类函数还可以 调用修改数据库的VOLATILE函数。如果那样做，你将发现 该STABLE或IMMUTABLE函数不会发现由被调 用函数所作的数据库改变，因为它们对它的快照不可见）。

38.8. 过程语言函数

PostgreSQL允许用除 SQL 和 C 之外 的语言编写用户定义的函数。这些语言通常被称为过程语言（PL）。 过程语言并不内建在PostgreSQL服务器中， 它们通过可装载模块提供。更多信息请见第 42 章以及接下来的 章节。

38.9. 内部函数

内部函数由 C 编写并且已经被静态链接到PostgreSQL 服务器中。该函数定义的“主体”指定该函数的 C 语言名称， 它必须和声明 SQL 函数所用的名称一样（为了向后兼容性的原因，也接受空 主体，那时会认为 C 语言函数名与 SQL 函数名相同）。

通常，所有存在于服务器中的内部函数都在数据库集簇的初始化（见 第 18.2 节期间被声明，但是用户可以使用 CREATE FUNCTION为一个内部函数创建 额外的别名。在CREATE FUNCTION中用 语言名internal来声明内部函数。例如，要为 sqrt函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

（大部分内部函数应该被声明为“严格”）。

> **注意**
>
> 上述场景中并非所有“预定义”的函数都是 “内部”函数。有些预定义的函数由 SQL 编写。

38.10. C 语言函数

用户定义的函数可以用 C 编写（或者可以与 C 兼容的语言，例如 C++）。 这类函数被编译成动态载入对象（也被称为共享库）并且由服务器在 需要时载入。动态载入是把“C语言”函数和 “内部”函数区分开的特性 — 两者真正的编码习惯 实际上是一样的（因此，标准的内部函数库是用户定义的 C 函数很好 的源代码实例）。

当前仅有一种调用约定被用于C函数（“版本1”）。如下文所示，为函数编写一个PG_FUNCTION_INFO_V1()宏就能指示对该调用约定的支持。

38.10.1. 动态载入

在一个会话中第一次调用一个特定可载入对象文件中的用户定义函数时， 动态载入器会把那个对象文件载入到内存以便该函数被调用。因此用户 定义的 C 函数的CREATE FUNCTION必须 为该函数指定两块信息：可载入对象文件的名称，以及要在该对象文件中 调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则 它被假定为和 SQL 函数名相同。

下面的算法被用来基于CREATE FUNCTION 命令中给定的名称来定位共享对象文件：

1. 如果名称是一个绝对路径，则载入给定的文件。

2. 如果该名称以字符串$libdir开始，那么这一部分会被 PostgreSQL包的库目录名（在编译时确定）替换。

3. 如果该名称不包含目录部分，会在配置变量 dynamic_library_path指定的路径中搜索该文件。

4. 否则（在该路径中没找到该文件，或者它包含一个非绝对目录）， 动态载入器将尝试接受给定的名称，这大部分会导致失败（依赖 当前工作目录是不可靠的）。

如果这个序列不起作用，会把平台相关的共享库文件名扩展（通常是 .so）追加到给定的名称并且再次尝试上述 的过程。如果还是失败，则载入失败。

我们推荐相对于$libdir或者通过动态库路径来 定位共享库。如果升级版本时新的安装在一个不同的位置，则可以 简化升级过程。$libdir实际表示的目录可以用 命令pg_config --pkglibdir来找到。

用于运行PostgreSQL服务器的 用户 ID 必须能够通过要载入文件的路径。常见的错误是把文件或 更高层的目录变得对postgres用户 不可读或者不可执行。

在任何情况下，CREATE FUNCTION命令 中给定的文件名会被原封不动地记录在系统目录中，这样如果需要再次 载入该文件则会应用同样的过程。

> **注意**
>
> PostgreSQL不会自动编译 C 函数。在 从CREATE FUNCTION命令中引用对象文件之前，它必须先被编译好。更多信息请见第 38.10.5 节

为了确保动态载入对象文件不会被载入到一个不兼容的服务器， PostgreSQL会检查该文件是否包含一个 带有合适内容的“magic block”。这允许服务器检测到明显的不兼 容，例如为不同PostgreSQL主版本编译 的代码。要包括一个 magic block，在写上包括 头文件fmgr.h的语句之后，在该模块的源文件之一（并且只 能在其中一个）中写上这些：

```
PG_MODULE_MAGIC;
```

在被第一次使用后，动态载入对象文件会留在内存中。在同一个会话中 对该函数未来的调用将只会消耗很小的负荷进行符号表查找。如果需要 重新载入一个对象文件（例如重新编译以后），需要开始一个新的会话。

可以选择让一个动态载入文件包含初始化和终止化函数。如果文件包含一个 名为_PG_init的函数，则文件被载入后会立刻调用该函数。 该函数不接受参数并且应该返回void。如果文件包括一个名为 _PG_fini的函数，则在卸载该文件之前会立即调用该函数。同样地，该函数不接受参数并且应该返回 void。注意将只在卸载文件的过程 中会调用_PG_fini，进程结束时不会调用它（当前，卸载被 禁用并且从不发生，但是未来可能会改变）。

38.10.2. C 语言函数中的基本类型

要了解如何编写 C 语言函数，你需要了解 PostgreSQL如何在内部表达基本数据类型 以及如何与函数传递它们。在内部， PostgreSQL把一个基本类型认为是 “一团内存”。在类型上定义的用户定义函数说明了 PostgreSQL在该类型上操作的方式。也就 是说，PostgreSQL将只负责把数据存在磁盘以 及从磁盘检索数据，而使用你的用户定义函数来输入、处理和输出该数据。

基本类型可以有三种内部格式之一：

- 传值，定长
- 传引用，定长
- 串引用，变长

传值类型在长度上只能是 1、2 或 4 字节（如果你的机器上 sizeof(Datum)是 8，则还有 8 字节）。你应当小心地 定义你的类型以便它们在所有的架构上都是相同的尺寸（字节）。例如， long类型很危险，因为它在某些机器上是 4 字节但在 另外一些机器上是 8 字节，而int类型在大部分 Unix 机器 上都是 4 字节。在 Unix 机器上int4类型一种合理的实现可能是：

```
/* 4 字节整数，传值 */
typedef int int4;
```

（实际的 PostgreSQL C 代码会把这种类型称为int32，因为 C 中的习惯是intXX 表示XX 位。注意 因此还有尺寸为 1 字节的 C 类型int8。SQL 类型 int8在 C 中被称为int64。另见 表 38.1 。

在另一方面，任何尺寸的定长类型可以用传引用的方法传递。例如，这里有一种PostgreSQL类型的实现示例：

```
/* 16 字节结构，传引用 */
typedef struct
{
    double  x, y;
} Point;
```

在PostgreSQL函数中传进或传出这种 类型时，只能使用指向这种类型的指针。要返回这样一种类型的值，用 palloc分配正确的内存量，然后填充分配好的内存， 并且返回一个指向该内存的指针（还有，如果只想返回与具有相同数据类型的 一个输入参数相同的值，可以跳过额外的palloc并且返回 指向该输入值的指针）。

最后，所有变长类型必须也以引用的方式传递。所有变长类型必须用一个 正好 4 字节的不透明长度域开始，该域会由SET_VARSIZE 设置，绝不要直接设置该域！所有要被存储在该类型中的数据必须在内存 中接着该长度域的后面存储。长度域包含该结构的总长度，也就是包括长 度域本身的尺寸。

另一个重点是要避免在数据类型值中留下未被初始化的位。例如，要注意 把可能存在于结构中的任何对齐填充字节置零。如果不这样做，你的数据 类型的逻辑等价常量可能会被规划器认为是不等的，进而导致低效的（不过 还是正确的）计划。

> **警告**
>
> 绝不要修改通过引用传递的输入值的内容。如果这样做 很可能会破坏磁盘上的数据，因为给出的指针可能直接指向一个磁盘缓冲 区。这条规则唯一的例外在第 38.11 节有解释。

例如，我们可以这样定义类型text：

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

[FLEXIBLE_ARRAY_MEMBER]记号表示数据部分的实际 长度不由该声明指定。

在操纵变长字节时，我们必须小心地分配正确数量的内存并且正确地 设置长度域。例如，如果我们想在一个text结构 中存储 40 字节，我们可以使用这样的代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ和sizeof(int32)一样， 但是用宏VARHDRSZ来引用变长类型的载荷的 尺寸被认为是比较好的风格。还有，必须 使用SET_VARSIZE宏来设置长度域，而不是用 简单的赋值来设置。

表 38.指定在编写使用一种 PostgreSQL内建类型的 C 语言函数时， 哪一种 C 类型对应于哪一种 SQL 类型。 “定义文件”列给出了要得到该类型定义需要 包括的头文件（实际的定义可能在一个由列举文件包括的不同 文件中。推荐用户坚持使用已定义的接口）。注意在任何源文 件中应该总是首先包括postgres.h， 因为它声明了很多你需要的东西。

表 38.1. 内建 SQL 类型等效的 C 类型

SQL 类型	C 类型	定义文件
abstime	AbsoluteTime	utils/nabstime.h
bigint (int8)	int64	postgres.h
boolean	bool	postgres.h（可能是编译器内建）
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	（编译器内建）
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

现在我们已经复习了基本类型所有可能的结构，现在可以展示一些 真实函数的例子了。

38.10.3. 版本 1 的调用约定

版本-1 的调用规范依赖于宏来降低传参数和结果的复杂度。版本-1 函数的 C 声明总是：

```
Datum funcname(PG_FUNCTION_ARGS)
```

此外，宏调用：

```
PG_FUNCTION_INFO_V1(funcname);
```

必须出现在同一个源文件中（按惯例会正好写在该函数本身之前）。 这种宏调用不是internal语言函数所需要的，因为 PostgreSQL会假定所有内部函数都使用 版本-1 规范。不过，对于动态载入函数是必需的。

在版本-1 函数中，每一个实参都使用对应于该参数数据类型的PG_GETARG_xxx()宏取得。在非严格的函数中，需要使用PG_ARGNULL_xxx()对参数是否为空提前做检查。结果要用对应于返回类型的PG_RETURN_xxx()宏返回。PG_GETARG_xxx()的参数是要取得的函数参数的编号，从零开始计。PG_RETURN_xxx()的参数是实际要返回的值。

这里是一些使用版本-1调用约定的例子：

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_MODULE_MAGIC;

/* 传值 */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* 传引用，定长 */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* FLOAT8 的宏隐藏了它的传引用本质。 */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* 这里，Point 的传引用本质没有被掩盖。 */
    Point     *pointx = PG_GETARG_POINT_P(0);
```

```
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* 传引用，变长 */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);

    /*
     * VARSIZE_ANY_EXHDR是该结构的尺寸（以字节为单位）减去其头部的
     * VARHDRSZ或VARHDRSZ_SHORT。用一个完整长度的头部构建该拷贝。
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA是指向新结构的数据区域的指针。来源可以是一个短数据，
     * 所以要通过VARDATA_ANY检索它的数据。
     */
    memcpy((void *) VARDATA(new_t), /* 目标 */
           (void *) VARDATA_ANY(t), /* 源头 */
           VARSIZE_ANY_EXHDR(t));   /* 多少字节 */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

假定上述代码已经准备在文件funcs.c中并且被编译成一个共享对象，我们可以用这样的命令在PostgreSQL中定义函数：

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- 注意SQL函数名“add_one”的重载
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

这里，DIRECTORY表示共享库文件的目录（例如PostgreSQL的教程目录，它包含这一节中用到的例子的代码）。（更好的风格是先把DIRECTORY放入搜索路径，在AS子句中只使用'funcs'。在任何情况下，我们可以为一个共享库省略系统相关的扩展名，通常是.so）。

注意我们已经把函数指定为“strict”，这意味着如果有任何输入值为空，系统应该自动假定得到空结果。通过这种做法，我们避免在函数代码中检查空值输入。如果不这样做，我们必须使用PG_ARGISNULL()明确地检查空值输入。

乍一看，版本-1编码习惯好像是在使用普通C调用约定之上的无意义的愚民政策。不过，它们确实允许处理可为NULL的参数/返回值以及被“TOAST”过（压缩或者线外）的值。

宏PG_ARGISNULL(n)允许一个函数测试是否每一个输入为空（当然，只需要在没有声明为“strict”的函数中这样做）。和PG_GETARG_xxx()宏一样，输入参数也是从零开始计数。注意应该在验证了一个参数不是空之后才执行PG_GETARG_xxx()。要返回一个空结果，应执行PG_RETURN_NULL()，它对严格的以及非严格的函数都有用。

在版本-1接口中提供的其他选项是PG_GETARG_xxx()宏的两个变种。其中的第一种是PG_GETARG_xxx_COPY()，它确保返回的指定参数的拷贝可以被安全地写入（通常的宏有时会返回一个指向表中物理存储的值，它不能被写入。使用PG_GETARG_xxx_COPY()宏可以保证得到一个可写的结果）。第二种变种PG_GETARG_xxx_SLICE()宏有三个参数。第一个是函数参数的编号（如上文）。第二个和第三个是要被返回的段的偏移量和长度。偏移量从零开始计算，而负值的长度则表示要求返回该值的剩余部分。当大型值的存储类型为“external”时，这些宏提供了访问这些大型值的更有效的方法（列的存储类型可以使用ALTER TABLE tablename ALTER COLUMN colname SET STORAGE storagetype来指定。storagetype取plain、external、extended或者main）。

最后，版本-1 的函数调用规范可以返回集合结果（第 38.10.8 节）、实现触发器函数（第 39 章）和过程语言调用处理器（第 56 章）。更多细节可见源代码发布中的src/backend/utils/fmgr/README。

38.10.4. 编写代码

在开始更高级的话题之前，我们应该讨论一下用于 PostgreSQL C 语言函数的编码规则。 虽然可以把不是 C 编写的函数载入到 PostgreSQL中，这通常是很困难的， 因为其他语言（例如 C++、FORTRAN 或者 Pascal）通常不会遵循和 C 相同的调用规范。也就是说，其他语言不会以同样的方式在函数之间传递 参数以及返回值。由于这个原因，我们会假定你的 C 语言函数确实是用 C 编写的。

编写和编译 C 函数的基本规则如下：

- 使用pg_config --includedir-server 找出PostgreSQL服务器头文件安装在 系统的哪个位置。
- 编译并且链接你的代码（这样它就能被动态载入到 PostgreSQL中）总是 要求特殊的标志。对特定的操作系统的做法详见 第 38.10.5 节
- 记住为你的共享库按第 38.10.1 节所述 定义一个“magic block”。
- 在分配内存时，使用 PostgreSQL函数 palloc和 pfree， 而不是使用对应的 C 库函数 malloc和free。 在每个事务结束时会自动释放通过palloc 分配的内存，以免内存泄露。
- 总是要使用memset把你的结构中的字节置零（或者 最开始就用palloc0分配它们）。即使你对结构中的 每个域都赋值，也可能有对齐填充（结构中的空洞）包含着垃圾值。 如果不这样做，很难支持哈希索引或哈希连接，因为你必须选出数据 结构中有意义的位进行哈希计算。规划器有时也依赖于用按位相等来 比较常量，因此如果逻辑等价的值不是按位相等的会导致出现不想要 的规划结果。
- 大部分的内部PostgreSQL类型 都声明在postgres.h中，不过函数管理器 接口（PG_FUNCTION_ARGS等）在 fmgr.h中，因此你将需要包括至少这两个 文件。为了移植性，最好在包括任何其他系统或者用户头文件之前， 先包括postgres.h。包括postgres.h也将会为你包括 elog.h和palloc.h。
- 对象文件中定义的符号名不能相互冲突或者与 PostgreSQL服务器可执行程序中 定义的符号冲突。如果出现有关于此的错误消息，你将必须重命名你的 函数或者变量。

38.10.5. 编译和链接动态载入的函数

在使用 C 编写的PostgreSQL扩展函数之前，必须以一种特殊的方式编译并且链接它们，以便产生一个能被服务器动态载入的文件。简而言之，需要创建一个共享库。

超出本节所含内容之外的信息请参考你的操作系统文档，特别是 C 编译器（cc）和链接编辑器（ld）的手册页。另外，PostgreSQL源代码的contrib目录中包含了一些可以工作的例子。但是，如果你依靠这些例子，也会使你的模块依赖于PostgreSQL源码的可用性。

创建共享库通常与链接可执行文件相似：首先源文件被编译成对象文件，然后对象文件被链接起来。对象文件需要被创建为独立位置代码（PIC），，这在概念上意味着当它们被可执行文件载入时，它们可以被放置在内存中的任意位置（用于可执行文件的对象文件通常不会以那种方式编译）。链接一个共享库的命令会包含特殊标志来把它与链接一个可执行文件区分开（至少在理论上 — 在某些系统上实际上很丑陋）。

在下列例子中，我们假定你的源代码在一个文件foo.c中，并且我们将创建一个共享库foo.so。除非特别注明，中间的对象文件将被称为foo.o。一个共享库能包含多于一个对象文件，但是我们在这里只使用一个。

FreeBSD

用来创建PIC的编译器标志是-fPIC。要创建共享库，编译器标志是-shared。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

这适用于FreeBSD从 3.0 开始的版本。

HP-UX

该系统编译器创建PIC的编译器标志是+z。当使用GCC自己的-fPIC时。用于共享库的链接器标志是-b。因此：

```
cc +z -c foo.c
```

或者：

```
gcc -fPIC -c foo.c
```

并且然后：

```
ld -b -o foo.sl foo.o
```

和大部分其他系统不同，HP-UX为共享库使用扩展.sl。

Linux

创建PIC的编译器标志是-fpic。创建一个共享库的编译器标志是-shared。一个完整的例子看起来像：

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

这里是一个例子。它假定安装了开发者工具。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

创建PIC的编译器标志是-fPIC。对于ELF系统，带着标志-shared的编译器被用来链接共享库。在旧的非 ELF 系统上，ld -Bshareable会被使用。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

创建PIC的编译器标志是-fPIC。ld -Bshareable被用来链接共享库。

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

创建PIC的编译器标志是-KPIC（用于 Sun 编译器）以及-fPIC（用于GCC）。要链接共享库，编译器选项对几种编译器都是-G或者是对GCC使用-shared。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

> **提示**
>
> 如果这对你来说太复杂，你应该考虑使用 GNU Libtool[1]，它会用一个统一的接口隐藏平台差异。

结果的共享库文件接着就可以被载入到PostgreSQL。当把文件名指定给CREATE FUNCTION命令时，必须把共享库文件的名字给它，而不是中间的对象文件。注意系统的标准共享库扩展（通常是.so或者.sl）在CREATE FUNCTION命令中可以被忽略，并且通常为了最好的可移植性应该被忽略。

服务器会期望在哪里寻找共享库文件请参考第 38.10.1 节

38.10.6. 组合类型参数

组合类型没有像 C 结构那样的固定布局。组合类型的实例可能包含 空值域。此外，继承层次中的组合类型可能具有和同一继承层次中 其他成员不同的域。因此， PostgreSQL提供了函数接口 来访问 C 的组合类型的域。

假设我们想要写一个函数来回答查询：

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

如果使用版本-1的调用规范，我们可以定义 c_overpaid为：

```
#include "postgres.h"
#include "executor/executor.h"  /* 用于 GetAttributeByName() */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* 另外，我们可能更想对空 salary 用 PG_RETURN_NULL() 。*/

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

[1] http://www.gnu.org/software/libtool/

GetAttributeByName是返回指定行的属性的 PostgreSQL系统函数。它有三个参数： 类型为HeapTupleHeader的传入参数、想要访问的函数名 以及一个说明该属性是否为空的返回参数。 GetAttributeByName返回一个Datum 值，可以把它用合适的DatumGetXXX() 宏转换成正确的数据类型。注意如果空值标志被设置，那么返回值是没有 意义的，所以在对结果做任何事情之前应该先检查空值标志。

也有GetAttributeByNum函数，它可以用目标属性 的属性号而不是属性名来选择目标属性。

下面的命令声明 SQL 中的c_overpaid：

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

注意我们用了STRICT，这样我们不需要检查输入参数是否 为 NULL。

38.10.7. 返回行（组合类型）

要从一个 C 语言函数中返回一个行或者组合类型值，你可以使用一种 特殊的 API，它提供的宏和函数隐藏了大部分的构建组合数据类型的 复杂性。要使用这种 API，源文件中必须包括：

```
#include "funcapi.h"
```

有两种方式可以构建一个组合数据值（以后就叫一个“元组”）： 可以从一个 Datum 值的数组构造，或者从一个 C 字符串（可被传递给该元组 各列的数据类型的输入转换函数）的数组构造。在两种情况下，都首先需要为 该元组的结构获得或者构造一个TupleDesc描述符。在处 理 Datum 时，需要把该TupleDesc传递给 BlessTupleDesc，接着为每一行调用 heap_form_tuple。在处理 C 字符串时，需要把该 TupleDesc传递给TupleDescGetAttInMetadata，接着为每一行调用 BuildTupleFromCStrings。对于返回一个元组集合的函数， 这些设置步骤可以在第一次调用该函数时一次性完成。

有一些助手函数可以用来设置所需的TupleDesc。在大部分 返回组合值的函数中推荐的方式是调用：

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

传递传给调用函数本身的同一个fcinfo结构（这当然要求使用的 是版本-1 的调用规范）。resultTypeId可以被指定为 NULL或者一个本地变量的地址以接收该函数的结果类型 OID。 resultTupleDesc应该是一个本地 TupleDesc变量的地址。检查结果是不是TYPEFUNC_COMPOSITE，如果是则 resultTupleDesc已经被用所需的 TupleDesc填充（如果不是，你可以报告一个错误，并且 返回“function returning record called in context that cannot accept type record”字样的消息）。

> **提示**
>
> get_call_result_type能够解析一个多态函数结果的实际类型， 因此不仅在返回组合类型的函数中，在返回标量多态结果的函数中它也是非常 有用的。resultTypeId输出主要用于返回多态标量的函数。

> **注意**
>
> get_call_result_type有一个兄弟 get_expr_result_type，它被用来解析被表示为一棵表达式 树的函数调用的输出类型。在尝试确定来自函数外部的结果类型时可以用它。 也有一个get_func_result_type，当只有函数的OID 可用时 可以用它。不过这些函数无法处理被声明为返回record的函数，并且get_func_result_type无法解析多态类型，因此你 应该优先使用get_call_result_type。

比较老的，现在已经被废弃的获得TupleDesc的函数是：

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

它可以为一个提到的关系的行类型得到TupleDesc， 还有：

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

可以基于一个类型 OID 得到TupleDesc。这可以被用来 为一种基础或者组合类型获得TupleDesc。不过，对于 返回record的函数它不起作用，并且它无法解析多态类型。

一旦有了一个TupleDesc，如果计划处理 Datum可以调用：

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

如果计划处理 C 字符串，可调用：

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

如果正在编写一个返回集合的函数，你可以把这些函数的结果保存在 FuncCallContext结构中 — 分别使用 tuple_desc或者attinmeta域。

在处理 Datum 时，使用

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

来用 Datum 形式的用户数据构建一个HeapTuple。

在处理 C 字符串时，使用

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

来用 C 字符串形式的用户数据构建一个HeapTuple。 values是一个 C 字符串数组，每一个元素是返回行 的一个属性。每一个 C 字符串应该是该属性数据类型的输入函数所期望 的格式。为了对一个属性返回空值，values数组中对 应的指针应该被设置为NULL。对于你返回的每一行都将 再次调用这个函数。

一旦已经构建了一个要从函数中返回的元组，它必须被转换成一个 Datum。使用

```
HeapTupleGetDatum(HeapTuple tuple)
```

可把一个HeapTuple转换成合法的 Datum。如果你 只想返回一行，那么这个Datum可以被直接返回，在一个 集合返回函数中它也可以被当做当前的返回值。

下一节中会有一个例子。

38.10.8. 返回集合

也提供了一种特殊的 API 来支持从 C 语言函数中返回集合（多个行）。 集合返回函数必须遵循版本-1 的调用规范。如上文所述，源文件中还 必须包括funcapi.h。

对返回的每一个项，一个集合返回函数（SRF）都会 被调用一次。因此，这个SRF必须保存足够的状态来 记住它正在做什么并且在每次调用时返回下一个项。结构 FuncCallContext被提供来帮助控制这个过程。在 一个函数中，fcinfo->flinfo->fn_extra被用来在 多次调用中保持指向FuncCallContext的指针。

```
typedef struct FuncCallContext
{
    /*
     * 本次调用以前已经被调用过多少次
     *
     * SRF_FIRSTCALL_INIT() 会为你把 call_cntr 初始化为 0，
     * 并且在每次调用 SRF_RETURN_NEXT() 时增加。
     */
    uint64 call_cntr;

    /*
     * 可选：最大调用次数
     *
     * 这里的 max_calls 只是为了方便，设置它是可选的。
     * 如果没有设置，你必须提供替代的方法来了解函数什么时候做完。
     */
    uint64 max_calls;

    /*
     * 可选：指向结果槽的指针
     *
     * 这已经被废弃并且只为向后兼容而存在，也就是那些使用被废弃的
     * TupleDescGetSlot() 的用户定义 SRF。.
     */
    TupleTableSlot *slot;

    /*
     * 可选：指向用户提供的上下文信息的指针
     *
     * user_fctx 是一个指向你自己的数据的指针，它可用来在函数的多次
     * 调用之间保存任意的上下文信息。
     */
    void *user_fctx;

    /*
     * 可选：指向包含属性类型输入元数据的结构的指针
     *
     * attinmeta 被用在返回元组（即组合数据类型）时，在返回基本数据类型
     * 时不会使用。只有想用BuildTupleFromCStrings()创建返回元组时才需要它。
     */
    AttInMetadata *attinmeta;

    /*
     * 用于保存必须在多次调用间都存在的结构的内存上下文
     *
     * SRF_FIRSTCALL_INIT() 会为你设置 multi_call_memory_ctx，并且由
     * SRF_RETURN_DONE() 来清理。对于任何需要在 SRF 的多次调用间都
```

```
     * 存在的内存来说，它是最合适的内存上下文。
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * 可选：指向包含元组描述的结构的指针
     *
     * tuple_desc 被用在返回元组（即组合数据类型）时，并且只有在用
     * heap_form_tuple() 而不是 BuildTupleFromCStrings() 构建元组时才需要它。
     * 注意这里存储的 TupleDesc 指针通常已经被先运行过 BlessTupleDesc()。
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

SRF使用一些自动操纵FuncCallContext （并且期望通过fn_extra找到它）的函数和宏。可使用：

SRF_IS_FIRSTCALL()

来判断你的函数是否是第一次被调用。在第一次调用时（只能在第一次调用时）使用：

SRF_FIRSTCALL_INIT()

可初始化FuncCallContext。在每一次函数调用时（包括第一次） 可使用：

SRF_PERCALL_SETUP()

为使用FuncCallContext做适当的设置并且清除上一次 留下来的任何已返回的数据。

如果你的函数有数据要返回，可使用：

SRF_RETURN_NEXT(funcctx, result)

把它返回给调用者（result必须是类型Datum， 可以是一个单一值或者按上文所述准备好的元组）。最后，当函数完成了 数据返回后，可使用：

SRF_RETURN_DONE(funcctx)

来清理并且结束SRF。

SRF被调用时的当前内存上下文被称作一个瞬时上下文， 在两次调用之间会清除它。这意味着你不必对用palloc 分配的所有东西调用pfree，它们将自动被释放。不过， 如果你想要分配任何需要在多次调用间都存在的数据结构，需要把它们 放在其他地方。对于任何需要在SRF结束运行之前都存 在的数据来说，multi_call_memory_ctx引用的内存 上下文是一个合适的位置。在大部分情况中，这意味着应该在做第一次 调用设置时就切换到multi_call_memory_ctx中。

> **警告**
>
> 虽然函数的实参在多次调用之间保持不变，但如果在瞬时上下文中 反 TOAST了参数（通常由 PG_GETARG_xxx 宏完成），那么被反 TOAST 的拷贝将在每次循环中被释放。相应地， 如果你把这些值的引用保存在user_fctx中，你也必须在反 TOAST 之后把它们拷贝到 multi_call_memory_ctx中，或者确保你只在那个 上下文中反 TOAST 这些值。

一个完整的伪代码例子：

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* 这里是一次性设置代码： */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* 这里是每一次都要做的设置代码： */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* 这里只是一种测试是否执行完的方法： */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* 这里返回另一个项： */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* 这里已经完成了项的返回并且需要进行清理： */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

一个返回组合类型的简单SRF的完整例子：

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;
```

```
/* 只在第一次函数调用时做的事情 */
if (SRF_IS_FIRSTCALL())
{
    MemoryContext   oldcontext;

    /* 创建一个函数上下文，让它在多次调用间都保持存在 */
    funcctx = SRF_FIRSTCALL_INIT();

    /* 切换到适合多次函数调用的内存上下文 */
    oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

    /* 要返回的元组总数 */
    funcctx->max_calls = PG_GETARG_UINT32(0);

    /* 为结果类型构造一个元组描述符 */
    if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("function returning record called in context "
                        "that cannot accept type record")));

    /*
     * 生成后面需要用来从原始 C 字符串产生元组的属性元数据
     */
    attinmeta = TupleDescGetAttInMetadata(tupdesc);
    funcctx->attinmeta = attinmeta;

    MemoryContextSwitchTo(oldcontext);
}

/* 在每一次函数调用都要完成的事情 */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* 如果还有要发送的 */
{
    char       **values;
    HeapTuple    tuple;
    Datum        result;

    /*
     * 为构建返回元组准备一个值数组。这应该是一个 C
     * 字符串数组，之后类型输入函数会处理它。
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* 构建一个元组 */
```

```
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* 把元组变成 datum */
        result = HeapTupleGetDatum(tuple);

        /* 清理（实际并不必要） */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* 如果没有要发送的 */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

在 SQL 中声明这个函数的一种方法是：

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

一种不同的方法是使用 OUT 参数：

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

注意在这种方法中，函数的输出类型在形式上是一种匿名的 record类型。

源码中的目录contrib/tablefunc 下的模块包含集合返回函数更加复杂的例子。

38.10.9. 多态参数和返回类型

C 语言函数可以被声明为接受和返回多态类型 anyelement、anyarray、 anynonarray、anyenum以及anyrange。 关于多态函数的详细介绍请见 第 38.2.5 节当函数参数或者返回 类型被定义为多态类型时，函数的编写者无法提前知道会用什么数据类型 调用该函数或者该函数需要返回什么数据类型。在fmgr.h 中提供了两种例程来允许版本-1 的 C 函数发现其参数的实际数据类型以及 它要返回的类型。这些例程被称为 get_fn_expr_rettype(FmgrInfo *flinfo)和 get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)。它们 返回结果或者参数的类型的 OID，或者当该信息不可用时返回 InvalidOid。结构flinfo通常被当做 fcinfo->flinfo访问。参数argnum则是从零 开始计。get_call_result_type也可被用作 get_fn_expr_rettype的一种替代品。还有get_fn_expr_variadic，它可以被用来找出 variadic 参数 是否已经被合并到了一个数组中。这主要用于 VARIADIC "any"函数，因为对于接收普通数组类型的 variadic 函数来说总是会发生这类合并。

例如，假设我们想要写一个接收一个任意类型元素并且返回一个该类型的一维 数组的函数：

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 得到提供的元素，小心它为 NULL 的情况 */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* 只有一个维度 */
    ndims = 1;
    /* 和一个元素 */
    dims[0] = 1;
    /* 且下界是 1 */
    lbs[0] = 1;

    /* 得到该元素类型所需的信息 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* 现在构建数组 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

下面的命令在 SQL 中声明函数make_array：

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

有一种只对 C 语言函数可用的多态变体：它们可以被声明为接受类型为 "any"的参数（注意这种类型名必须用双引号引用，因为它也 是一个 SQL 保留字）。这和anyelement相似，不过它不约束 不同的"any"参数为同一种类型，它们也不会帮助确定函数的 结果类型。C 语言函数也能声明它的第一个参数为 VARIADIC "any"。这可以匹配一个或者多个任意类型的实参（ 不需要是同一种类型）。这些参数不会像普通 variadic 函 数那样被收集到一个数组中，它们将被单独传递给该函数。使用这种特性时， 必须用PG_NARGS()宏以及上述方法来判断实参的个数和类 型。还有，这种函数的用户可能希望在他们的函数调用中使用 VARIADIC关键词，以期让该函数将数组元素作为单独的参数 对待。如果想要这样，在

使用get_fn_expr_variadic检测被 标记为VARIADIC的实参之后，函数本身必须实现这种行为。

38.10.10. 转换函数

一些函数调用可以在规划期间基于该函数的特定的属性被简化。例如， int4mul(n, 1)可以被简化为n。要定义这种 与函数相关的优化，可以写一个转换函数并且将其 OID 放在主函数的pg_proc项的 protransform域中。该转换函数必须具有 SQL 式样 protransform(internal) RETURNS internal。其参数（实 际是FuncExpr *）是一个表示对主函数调用的伪节点。如果 该转换函数对表达式树的研究证明一个简化的表达式树能够替代所有可能 的具体调用，则会构造并且返回简化的表达式。否则会返回一个 NULL指针（不是一个 SQL 空值）。

我们不保证PostgreSQL在转换函数可以进行简化的 情况下绝不会调用主函数。要保证简化后的表达式和对主函数的一次实际 调用是严格等价的。

当前，由于安全性的考虑，这种功能没有在 SQL 层面上显示给用户。因此， 这种功能实际只用于优化内建函数。

38.10.11. 共享内存和 LWLock

外接程序可以在服务器启动时保留 LWLock 和共享内存。 必须通过在shared_preload_libraries 中 指定外接程序的共享库来预先载入它。从_PG_init 函数中调用

```
void RequestAddinShmemSpace(int size)
```

可以保留共享内存。

通过从_PG_init中调用

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

可以保留 LWLock。这将确保一个名为tranche_name 的 LWLock 数组可用，该数组的长度为num_lwlocks。 使用GetNamedLWLockTranche可得到该数组的指针。

为了避免可能的竞争情况，在连接并且初始化共享内存时，每一个 后端应该使用 LWLock AddinShmemInitLock，如下所示：

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->locks = GetNamedLWLockTranche("my tranche name");
        }
        LWLockRelease(AddinShmemInitLock);
}
```

38.10.12. 把 C++ 用于可扩展性

尽管PostgreSQL后端是用 C 编写的， 只要遵循下面的指导方针也可以用 C++ 编写扩展：

- 所有被后端访问的函数必须对后端呈现一种 C 接口，然后这些 C 函数 调用 C++ 函数。例如，对后端访问的函数要求extern C 链接。对需要在后端和 C++ 代码之间作为指针传递的任何函数也要 这样做。
- 使用合适的释放方法释放内存。例如，大部分后端内存是通过 palloc()分配的，所以应使用pfree() 来释放。在这种情况中使用 C++ 的delete会失败。
- 防止异常传播到 C 代码中（在所有extern C函数的顶层 使用一个捕捉全部异常的块）。即使 C++ 代码不会显式地抛出任何 异常也需要这样做，因为类似内存不足等事件仍会抛出异常。任何异常 都必须被捕捉并且用适当的错误传回给 C 接口。如果可能，用 -fno-exceptions 来编译 C++ 以完全消灭异常。在这种 情况下，你必须在 C++ 代码中检查失败，例如检查new() 返回的 NULL。
- 如果从 C++ 代码调用后端函数，确定 C++ 调用栈值包含传统 C 风格 的数据结构（POD）。这是必要的，因为后端错误会 产生远距离的longjmp()，它无法正确的退回具有非 POD 对象的 C++ 调用栈。

总之，最好把 C++ 代码放在与后端交互的extern C函数之后， 并且避免异常、内存和调用栈泄露。

38.11. 用户定义的聚集

PostgreSQL中的聚集函数用状态值和状态转换函数定义。也就是，一个聚集操作使用一个状态值，它在每一个后续输入行被处理时被更新。要定义一个新的聚集函数，我们要为状态值选择一种数据类型、一个状态的初始值和一个状态转换函数。状态转换函数接收前一个状态值和该聚集当前行的输入值，并且返回一个新的状态值。万一该聚集的预期结果与需要保存在运行状态之中的数据不同，还能指定一个最终函数。最终函数接受结束状态值并且返回作为聚集结果的任何东西。原则上，转换函数和最终函数只是也可以在聚集环境之外使用的普通函数（实际上，通常出于性能的原因，会创建特殊的只能作为聚集的一部分工作的转换函数）。

因此，除了该聚集的用户所见的参数和结果数据类型之外，还有一种可能不同于参数和结果状态的内部状态值数据类型。

如果我们定义一个聚集但不使用一个最终函数，我们就得到了一个从每一行的列值计算一个运行函数的聚集。sum是这类聚集的一个例子。sum从零开始，并且总是把当前行的值加到它的运行总和上。例如，如果我们希望让一个sum聚集能工作在复数数据类型上，我们只需要该数据类型的加法函数。聚集定义是：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

我们可以这样使用：

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

（注意我们依赖于函数重载：有多于一个名为sum的聚集，但是PostgreSQL能够找出哪种 sum 适用于一个类型为complex的列）。

如果没有非空输入值，上述的sum定义将返回零（初始状态值）。也许我们想要在这种情况下返回空 — SQL 标准期望sum以这种方式行事。我们可以通过忽略initcond阶段简单地做到这一点，这样初始状态值就为空。通常这表示sfunc将需要检查一个空状态值输入。但是对于sum和一些其他简单聚集（如max和min），把第一个非空输入值插入到状态变量中并且接着在第二个非空输入值上开始应用转换函数就足够了。如果初始状态值为空并且转换函数被标记为“strict”（即不为空输入调用），PostgreSQL会自动这样做。

“strict”转换函数的另一点默认行为是只要碰到了一个空输入值，之前的状态值就保持不变。因此，空值会被忽略。如果你需要某些其他用于空输入的行为，不要把你的转换函数声明为 strict，而是把它编码为测试空输入并且做所需要的事情。

avg（均值）是一种更复杂的聚集例子。它要求两份运行状态：输入的总和以及输入的计数。最终结果通过将这些量相除而得到。均值是使用一个数组作为状态值的典型实现。例如，内建的avg(float8)实现看起来像：

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

> **注意**
>
> （float8_accum要求一个三元素的数组，而不只是两个元素，因为它累积平方和以及输入的总和以及计数。因此它也可以被用于其他聚集函数以及avg）。

SQL 中的聚集函数调用允许用DISTINCT和ORDER BY选项控制以什么顺序把行传递给该聚集的转换函数。这些选项的实现不需要该聚集的支持函数关心。

进一步的细节可见CREATE AGGREGATE命令。

38.11.1. 移动聚集模式

聚集函数可以选择性地支持移动聚集模式，这种模式允许很大程度上提高在具有移动帧起点的窗口中执行的聚集函数的速度（有关把聚集函数用作窗口函数请见第 3.5 和第 4.2.8 节。基本思想是在通常的“前向”转换函数之外，聚集提供一个逆向转换函数，该函数允许当行退出窗口帧时从聚集的运行状态值中移除它们的值。例如一个sum聚集使用加法作为前向转换函数，它可以使用减法作为逆向转换函数。如果没有一个逆向转换函数，每一次帧起点移动时，窗口函数机制必须重新从头计算该聚集，这会导致运行时间与输入行的数量乘以平均帧长度成比例。如果有一个逆向转换函数，运行时间只与输入行的数量成比例。

当前状态值和包含在当前状态中最早的行的聚集输入值被传递给逆向转换函数。它必须重新构建出如果给定的输入行不再被聚集（只聚集其后的行）时状态值会是什么样。这有时要求前向转换函数保存比普通聚集模式下更多的状态。因此，移动聚集模式使用一种完全不同于普通模式的实现：它有自己的状态数据类型、自己的前向转换函数以及自己的状态函数（如果需要）。如果不需要额外的状态，这些可以和普通模式的数据类型和函数相同。

作为一个例子，我们可以把上面给定的sum聚集扩展成支持移动聚集模式：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
```

```
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

名称以m开始的参数定义移动聚集实现。除了逆向转换函数minvfunc，它们都对应于没有m的普通聚集参数。

用于移动聚集模式的前向转换函数不允许返回空值作为新状态值。如果逆向转换函数返回空值，这被当作一种指示，它表明该逆向函数无法为这个特定输入逆转状态计算，因此该聚集计算将根据当前的帧开始位置重新从头计算。这种习惯允许移动聚集模式被用在一些不适合逆转运行状态值的少数情况下。逆向转换函数在这些情况下可以“撒手不管”，然后在它能够工作的大部分情况中再出来干活。例如，一个浮点数的聚集可能会在必须从运行状态值中移除一个NaN（不是一个数字）输入时撒手不管。

在编写移动聚集支持函数时，很重要的是确保逆向转换函数能够准确地重构正确的状态值。否则会导致用不用移动聚集模式时结果中产生用户可见的差别。为一个聚集增加一个逆向转换函数的例子最初看起来很简单，但是却无法满足float4或者float8输入上的sum的要求。sum(float8)的一种未经考虑的定义可以是

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

但是，这个聚集可能给出与没有逆向转换函数时很不同的结果。例如，考虑

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

这个查询返回0作为它的第二个结果，而不是我们期待的1。其原因是浮点值的有限精度：把1加到1e20还是会得到1e20，因此从中减去1e20会得到0而不是1。这是对于浮点计算的一种一般性限制，而不是PostgreSQL的限制。

38.11.2. 多态和可变聚集

聚集函数可以使用多态状态转换函数或最终函数，这样同样的函数能被用来实现多个聚集。关于多态函数的解释可参见第 38.2.5 节更进一步，聚集函数本身可以被指定为具有多态输入类型和状态类型，允许一个聚集函数服务于多种输入数据类型。这里是一个多态聚集的例子：

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

这里，每一次给定聚集调用的实际状态类型是把实际输入类型作为元素的数组类型。该聚集的行为是串接所有输入成一个该类型的数组（注意：内建的聚集array_agg提供了相似的功能，但是具有比上述定义更好的性能）。

这里是使用两种不同实际数据类型作为参数的输出：

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

如上述例子所示，通常一个具有多态结果类型的聚集函数有一个多态状态类型。这是必须的，因为否则就无法有意义地声明最终函数：它会需要有一个多态结果类型但是不能有多态参数类型，CREATE FUNCTION将当场拒绝那些无法从调用中推断结果类型的函数。但是使用一个多态状态类型有时并不方便。最常见的情况是，聚集支持函数使用 C 编写并且状态类型应该被声明为internal，因为在 SQL 层面上没有与它等效的类型。为了表述这种情况，可以声明最终函数为接受额外的匹配该聚集输入参数的“dummy”参数。这种假参数总是被传递为空值，因为当最终函数被调用时没有特定的值可用。它们的唯一用途是允许一个多态最终函数的结果类型被连接到该聚集的输入类型。例如，内建聚集array_agg的定义等效于：

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

这里，finalfunc_extra选项指定该最终函数接收除了状态值之外，还接收对应于该聚集输入参数的额外假参数。额外的anynonarray参数允许array_agg_finalfn的声明成为合法。

与常规函数的习惯大致相同，可以通过把一个聚集函数的最后一个参数声明为一个VARIADIC数组，这样可以让该函数接受可变数量的参数（见第 38.5.5 节。该聚集的转换函数也必须有相同的数组类型作为它们的最后一个参数。通常这类转换函数也会被标上VARIADIC，但这不被严格要求。

> **注意**
>
> 可变聚集最容易被误用的情况是与ORDER BY选项（见第 4.2.7 节一起使用，因为解析器无法在这样一种组合中是否给出了错误的实际参数数量。要记住

在ORDER BY右侧的任何东西都是一个排序键，而不是一个聚集的参数。例如，在

```
SELECT myaggregate(a ORDER BY a, b, c) FROM ...
```

中，解析器将认为看到的是一个聚集函数参数和三个排序键。但是，用户可能想要的是

```
SELECT myaggregate(a, b, c ORDER BY a) FROM ...
```

如果myaggregate是可变的，两种调用都是合法的。

出于相同的原因，在创建具有相同名称以及不同数量的常规参数的聚集函数时一定要三思而后行。

38.11.3. 有序集聚集

目前为止我们已经描述的聚集都是“普通”聚集。PostgreSQL还支持有序集聚集，它和普通聚集在两个关键点上相区别。首先，除了对每个输入行都要计算一次的普通聚集参数之外，一个有序集聚集可以有“直接”参数，这类参数针对每次聚集操作只计算一次。其次，用于普通聚集参数的语法需要显式地为它们指定一个排序顺序。一个有续集聚集通常被用来实现一种依赖于特定行序的计算（例如排名或者百分位数），因此排序是任何调用都要求的。例如，percentile_disc的内建定义等效于：

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

这个聚集接受一个float8直接参数（百分位数分数）以及一个可以是任意可排序数据类型的聚集输入。它可以用来得到一个家庭收入的中位数：

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

这里0.5是一个直接参数，它对于要作为一个在行之间变化的百分位数分数没有意义。

和普通聚集的情况不同，用于有序集聚集的输入行排序不是在幕后完成的，而是由该聚集的支持函数负责完成。典型的实现方法是在该聚集的状态值中保持对于一个“tuplesort”对象的引用，把到来的行输入给该对象，然后完成排序并且在最终函数中读出该数据。这种设计允许最终函数能够执行特殊操作，例如把附加的“假想”行注入到被排序的数据中。虽然用由PL/pgSQL或另一种 PL 语言编写的支持函数通常能够实现普通聚集，但是有序集聚集通常必须用 C 编写，因为它们的状态值无法用任何 SQL 数据类型来定义（在上面的例子中，注意状态值被声明为类型internal — 这很典型）。此外，由于最终函数会执行排序，后面就不能继续通过再次执行转移函数增加输入行。这意味着最终函数不是READ_ONLY的，它必

须在CREATE AGGREGATE中被声明为 READ_WRITE，或者在可以有额外的最终函数调用利用已经排序好的状态时声明为SHAREABLE。

用于一个有序集聚集的状态转移函数接收当前状态值外加对于每一行的聚集输入值，并且返回更新后的状态值。这和普通聚集的定义相同，但是注意没有提供直接参数（如果有）。最终函数接收最后的状态值、直接参数（如果有）的值以及对应于聚集输入的空值（如果指定了finalfunc_extra）。正如普通聚集，只有聚集是多态时finalfunc_extra才真正有用，那时就需要额外的假参数把最终函数的结果类型连接到该聚集的输入类型。

当前，有序集聚集不能被用做窗口函数，并且因此没有必要让它们支持移动聚集模式。

38.11.4. 部分聚集

可选地，一个聚集函数可以支持部分聚集。部分聚集的思想是在输入数据的不同子集上独立的运行该聚集的状态转移函数，然后把从这些子集得到的状态值组合起来产生最终的状态值，这样得到的状态值与在单次聚集操作中扫描所有输入得到的状态值相同。这种模式可以被用来进行并行聚集，用不同的工作者进程扫描表的不同部分。每一个工作者产生一个部分状态值，最后把这些部分状态值组合产生最终状态值（在未来，这种模式可能也会被用于组合在本地表和远程表上的聚集，但目前还未实现）。

为了支持部分聚集，聚集定义必须提供一个组合函数，这个函数接收两个该聚集的状态类型（表示在输入行的两个不同子集上得到的聚集结果）并且产生一个该状态类型的新值，该结果表示组合哪些聚集结果后的状态。至于来自两个集合的输入行的相对顺序则并没有指定。这意味着通常不可能为对输入行顺序敏感的聚集定义出可用的组合函数。

作为简单的例子，通过指定组合函数为与其转移函数中相同的“两者中较大者”和“两者中较小者”比较函数，MAX和MIN聚集可以支持部分聚集。SUM聚集则只需要一个额外的函数作为组合函数（同样，组合函数与其转移函数相同，除非状态值的宽度比输入数据类型更宽）。

组合函数很像一个把状态类型值而不是底层输入类型值作为其第二个参数的转移函数。尤其是处理空值和严格函数的规则是相似的。此外，如果聚集定义指定了非空的initcond，记住那不仅会被作为每一次部分聚集运行的初始状态，还会被作为组合函数的初始状态，对每一个部分结果都会调用组合函数将部分结果组合到该初始状态中。

如果聚集的状态类型被声明为internal，则组合函数应负责在用于聚集状态值的内存上下文中分配其结果。这意味着当第一个输入为NULL时，不能简单地返回第二个输入，因为那个值将会在错误的上下文中并且将不具有足够的寿命。

当聚集的状态类型被声明为internal时，通常聚集定义提供序列化函数和反序列化函数也是合适的，这两个函数允许这样一种状态值被从一个进程复制到另一个进程。如果没有这些函数就无法执行并行聚集，并且未来的本地/远程聚集之类的应用也可能无法工作。

一个序列化函数必须接收一个单一的internal类型参数并且返回一个bytea类型的结果，它表示把状态值打包成一个平面化的字节串。反过来，反序列化函数是上述转换的逆变换。反序列化函数必须接收两个类型为bytea和internal的参数，并且返回类型为internal的结果（第二个参数没有被使用并且总是为零，它的存在是由于类型安全性的原因）。反序列化函数的结果应该直接在当前内存上下文中分配，这与组合函数的结果不同，因为它不需要长期存在。

还有一点值得提示的是关于要被并行执行的聚集，聚集本身必须被标记上PARALLEL SAFE。其支持函数上的并行安全性标记不会被参考。

38.11.5. 聚集的支持函数

用 C 编写的函数能够通过调用AggCheckCallContext检测它是作为聚集支持函数调用的，例如：

```
if (AggCheckCallContext(fcinfo, NULL))
```

检查这个区别的原因是当它为真时，第一个输入必须是一个临时状态值并且可以因此安全地被就地修改而不是分配一个新的副本。例子可见int8inc()（虽然聚集的转移函数总是被允许就地修改转移值，但不鼓励聚集的最终函数这样做。如果最终函数要这样做，必须在创建聚集时声明这种行为。更多细节请参考CREATE AGGREGATE）。

AggCheckCallContext的第二个参数可以被用来检索保存有聚集状态值的内存上下文。这对希望把“扩展”对象（见第 38.12.1 节用作状态值的转移函数有用。在第一次调用时，转移函数应该返回一个扩展对象，其内存上下文是聚集状态上下文的一个子节点，然后在后续的调用中都保持返回同一个扩展对象。示例请见array_append()（array_append()不是任意内建聚集的转移函数，但其编写的目的就是在被用作一种自定义聚集的转移函数时表现得有效）。

另一种可用于由 C 编写的聚集函数的支持例程是AggGetAggref，它返回定义该聚集调用的Aggref解析节点。这主要对有序集聚集有用，它能检查Aggref的子结构来找出它们本应实现的排序顺序。在PostgreSQL源代码的orderedsetaggs.c中可以找到例子。

38.12. 用户定义的类型

如第 38.2 节所述， PostgreSQL能够被扩展成支持新的数据类型。这一节描述了如何定义新的基本类型，它们是被定义在SQL语言层面之下的数据类型。创建一种新的基本类型要求使用低层语言（通常是 C）实现在该类型上操作的函数。

这一节中的例子可以在源代码src/tutorial目录下的complex.sql和complex.c中找到。运行这些例子的指令可以在该目录的README文件中找到。

一种用户定义的类型必须总是具有输入和输出函数。这些函数决定该类型如何出现在字符串中（用于用户输入或者对用户的输出）以及如何在内存中组织该类型。输入函数采用一个空终止的字符串作为它的参数并且返回该类型的内部（内存）表达。输出函数采用该类型的内部表达作为参数并且返回一个空终止的字符串。如果我们想要对该类型做更多事情而不是只存储它，我们必须提供为我们想要的任何操作提供额外的实现函数。

假设我们想要定义一种类型complex，它表示复数。一种在内存中表达复数的自然的方法是下面的 C 结构：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

我们将需要让它成为一种传引用类型，因为它没办法放到一个单一的Datum值中。

至于该类型的外部字符串表达，我们选择了一种字符串形式的(x,y)。

输入和输出函数通常并不难编写，特别是输出函数。但是在定义类型的外部字符串表达时，记住你必须最终为该表达编写一个完整并且鲁棒的解析器作为你的输入函数。例如：

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
```

```
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

输出函数可以简单地写作：

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

你应当让输入和输出函数互为彼此的逆函数。如果不这样做，当你需要把数据转储到一个文件并且以后将它重新读入时会遇到很严重的问题。在涉及到浮点数时这是一个特别常见的问题。

可选地，一种用户定义的类型可以提供二进制输入和输出例程。二进制 I/O 通常比文本 I/O 更快但是可移植性更差。与文本 I/O 一样，定义准确的外部二进制表达是你需要负责的工作。大部分的内建数据类型都尝试提供一种不依赖机器的二进制表达。对于complex，我们的工作将建立在为类型float8提供的二进制 I/O 转换器上：

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;
```

```
    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

一旦我们编写了 I/O 函数并且把它们编译到了一个共享库中，我们就可以在 SQL 中定义complex类型。首先我们把它声明为一种 shell 类型：

```
CREATE TYPE complex;
```

这个语句的作用是为要定义的类型创建了一个占位符，这样允许我们在定义其 I/O 函数时引用该类型。现在我们可以定义 I/O 函数：

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

最后，我们可以提供该数据类型的完整定义：

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

在定义了一种新的基本类型后， PostgreSQL会自动提供对这种类型的数组支持。数组类型通常具有和基本类型相同的名称以及一个前置的下划线字符（_）。

一旦数据类型存在，我们就能够声明额外的函数来提供在该数据类型上有用的操作。然后可以在函数之上定义操作符，并且如果需要，可以创建操作符类来支持对该数据类型进行索引。这些额外的内容会在下面的小节中讨论。

如果数据类型的内部表达是可变长的，则内部表达必须遵循可变长数据的标准布局： 头四个字节必须是一个char[4]域，它从来不会被直接访问（通常 被称为vl_len_）。你必须使用SET_VARSIZE() 宏在这个域中存储整个数据的尺寸（包括长度域本身），并且使用VARSIZE()来检索它（这些宏之所以存在，是因为长度域 可能会根据平台来进行解码）。

更多细节请见CREATE TYPE命令的描述。

38.12.1. TOAST 考量

如果你的数据类型值的尺寸（内部形式）是可变的，更适合让该数据类型变成可 TOAST的（见第 68.2 节。即便值总是很小不会被压缩或者线外存储你也应该 这样做，因为TOAST也能通过减少头部负荷来为小数据减少空间。

为了支持TOAST存储，在该数据类型上操作的 C 函数必须总是要使用PG_DETOAST_DATUM解包任何交给它们的被 TOAST 过的值（习惯上这些细节都通过定义类型相关的GETARG_DATATYPE_P宏隐藏起来）。然后，在运行CREATE TYPE命令时，指定内部长度为variable并且选择某个不是plain的适当的存储选项。

如果数据对齐无关紧要（不管是为一个特定函数或者因为数据类型指定了字节对齐），那么有可能避免PG_DETOAST_DATUM的一些开销。你可以转而使用PG_DETOAST_DATUM_PACKED（习惯上通过定义一个GETARG_DATATYPE_PP宏隐藏）并且使用宏VARSIZE_ANY_EXHDR以及VARDATA_ANY来访问一个可能包装过的数据。此外，即使数据类型定义指定了一种对齐方式，这些宏返回的数据也不是对齐过的。如果对齐对你很重要，你必须使用常规的PG_DETOAST_DATUM接口。

> **注意**
>
> 老的代码经常声明vl_len_为一个int32域而不是char[4]。只要结构定义含有其他具有至少int32对齐的域，这就是 OK 的。但是在使用可能未对齐的数据时，使用这样一种结构定义就是危险的，编译器可能会把它当作一个授权来假定数据实际上已经被对齐，在对于对齐很严格的架构上会导致核心转储。

TOAST支持带来的另一个特性是能够拥有一种 扩展内存中数据表达，它比存储在磁盘上的格式使用起来更方便。 常规的或者“扁平的” varlena 存储格式最终只是一堆字节，它不能包含 指针，因为它可能会被复制到内存中的其他位置。对于复杂数据类型，扁平格式使 用起来可能代价更高，因此PostgreSQL提供了一种方式把 扁平格式“扩展”成更适合计算的一种表达，然后在该数据类型的函数之 间传递这种在内存中的格式。

要使用扩展存储，数据类型必须遵循src/include/utils/expandeddatum.h 中给定的规则定义一种扩展的格式，并且提供函数把扁平的 varlena 值“扩展” 到该格式以及从该格式“扁平化”回常规的 varlena 表达。然后确保所有该 数据类型的 C 函数都能接受这两种表达（可能通过一接收到其中一种就立刻 转换成另一种来做到）。这不要求一次性修改所有该数据类型的现有函数， 因为标准的PG_DETOAST_DATUM宏可以把扩展输入转换 成常规扁平格式。因此，现有的用于扁平 varlena 格式的函数仍然能够用于 扩展输入（虽然效率略低）。它不需要被转换，直到需要提高性能。

直到如何对付扩展表达的 C 函数通常分为两类：只能处理扩展格式的，以及 能同时处理扩展或扁平 varlena 输入的。前者更容易编写，但是可能总体效率 较低，因为由单个函数将一种扁平输入转换为扩展的形式的开销可能会超过在 扩展格式上操作所节省的开销。在只需要处理扩展格式时，可以把扁平输入到 扩展形式的转换隐藏在一个参数获取宏中，这样该函数就显得不比处理传统 varlena 输入的函数更复杂了。要处理两种类型的输入，需要编写一个参数获取 函数来反 TOAST 外部、短头部以及压缩的 varlena 输入，但不需要处理扩展 输入。这样一个函数可以被定义为返回一个指向由扁平 varlena 格式和扩展 格式组成的联合的指针。调用者可以使用 VARATT_IS_EXPANDED_HEADER()宏来判断它们接收到的 是哪种格式。

TOAST机制不仅允许把常规 varlena 值同扩展值区分开来， 还能区分指向扩展值的“read-write”和“read-only” 指针。只需要检查扩展值或者只会以安全的并且非语义可见的方式更改扩展 值的 C 函数不需要关心它们收到的是哪种类型的指针。如果收到一个读写 指针，要为输入值产生一个修改版本的 C 函数将被允许就地修改该扩展输入 值，但是如果它们收到的是一个只读指针则不能修改，在这种情况下它们不 得不先复制该值产生一个用于

修改的新值。构建了新扩展值的 C 函数应该总 是返回一个指向该值的读写指针。还有，如果一个就地修改读写扩展值的 C 函数中途失败，它应该负责让该值处于一种正常的状态。

有关使用扩展值的例子，请见标准数组这种基础结构，特别是 src/backend/utils/adt/array_expanded.c。

38.13. 用户定义的操作符

对于一个完成实际工作的底层函数的调用来说，每一个操作符都是“语法糖”，因此在创建操作符之前你必须先创建底层函数。不过，一个操作符不只是语法糖，因为它携带了额外的信息来帮助查询规划器优化使用该操作符的查询。下一节将致力于解释这些额外信息。

PostgreSQL支持左一元、右一元和二元操作符。操作符可以被重载，也就是说相同的操作符名称可以被用于具有不同操作数数量和类型的操作符。在执行一个查询时，系统会根据提供的操作数的数量和类型决定要调用的操作符。

这里有一个创建用于对两个复数做加法的操作符的例子。我们假设我们已经创建了类型complex（见第 38.12 节的定义。首先我们需要一个函数做这个加法，然后我们可以定义该操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

现在我们可以执行一个这样的查询：

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

这里我们已经展示了如何创建一个二元操作符。要创建一元操作符，只要忽略leftarg（左一元）和rightarg（右一元）之一即可。在CREATE OPERATOR中只要求procedure子句和参数子句。例子中展示的commutator子句是一个可选的子句，它被用作一个查询优化器使用的提示。有关commutator以及其他优化器提示的细节出现在下一小节中。

38.14. 操作符优化信息

一个PostgreSQL的操作符定义能够包括几种可选的子句，它们可以把有关操作符行为的有用的事情告诉系统。只要合适就应该提供这些子句，因为它们能够为使用该操作符的查询带来可观的速度提升。但是如果你提供了它们，你必须确保它们是正确的！不正确地使用一个优化子句可能导致很慢的查询、错误的输出或者其他不好的事情。如果你没有把握你可以总是省去优化子句，这样做的唯一后果是查询会比正常的速度慢。

在PostgreSQL的未来版本中可能会增加更多的优化子句。这里描述的优化子句都是版本 11.2 能理解的。

38.14.1. COMMUTATOR

如果提供了COMMUTATOR子句，它指定一个操作符作为被定义的操作符的交换子。如果对于所有可能输入的 x、y 值， (x A y) 等于 (y B x)，我们可以说操作符 A 是操作符 B 的交换子。注意，B 也是 A 的交换子。例如，用于一种特定数据类型的操作符 < 和 > 通常互为交换子，并且操作符 + 通常和它本身是交换的。但是操作符 - 通常不能与任何东西交换。

一个可交换操作符的左操作数类型与其交换子的右操作数类型相同，反之亦然。因此要查找交换子，只需要给PostgreSQL该交换子操作符的名称即可，并且在COMMUTATOR子句中也只需要提供它的名称。

为将要在索引和连接子句中使用的操作符提供交换子信息是很关键的，因为这允许查询优化器把这样一个子句“翻转”成不同计划类型所需的形式。例如，考虑一个这样的 WHERE 子句tab1.x = tab2.y，其中tab1.x和tab2.y是一种用户定义的类型，并且假设tab2.y被索引。除非优化器能决定如何把该子句翻转成tab2.y = tab1.x，否则它无法产生一个索引扫描，因为索引扫描机制期望看到被索引列出现在被给出的操作符的左边。PostgreSQL将无法简单地假定有一个可用的变换 — =操作符的创建者必须指定它是合法的（通过为该操作符标记交换子信息）。

在你定义一个子交换的操作符时，你这样做就行了。自拟定义一堆交换的操作符时，事情有一点棘手：如何在没有定义第二个操作符时完成第一个操作符的定义？因为第一个操作符需要第二个操作符作为其交换子。对这个问题有两种解决方案：

- 一种方法是忽略你定义的第一个操作符的COMMUTATOR子句，并且然后在第二个操作符的定义中提供第一个操作符作为交换子。由于PostgreSQL知道交换的操作符是成对出现的，当它看到第二个定义时它将自动回去并且填上第一个定义中缺失的COMMUTATOR子句。
- 另一种更直接的方法是就在两个定义中包括COMMUTATOR子句。当PostgreSQL处理第一个定义并且意识到COMMUTATOR引用了一个不存在的操作符时，系统将为那个操作符在系统目录中创造一个虚拟项。这个虚拟项只有操作符名称、左右操作数类型和结果类型的数据，因为这些是PostgreSQL在此时能够推断出来的所有东西。第一个操作符的目录项将会链接到这个虚拟项。稍后，当你定义第二个操作符时，系统用来自第二个定义的额外信息更新那个虚拟项。如果你尝试在虚拟操作符还未被填充之前使用它，你将只会得到一个错误消息。

38.14.2. NEGATOR

如果提供了NEGATOR子句，它指定一个操作符是正在被定义的操作符的求反器。如果操作符 A 和 B 都返回布尔结果并且对于所有可能的 x、y 输入都有 (x A y) 等于 NOT (x B y)，那么我们可以说 A 是 B 的求反器。注意 B 也是 A 的求反器。例如，< 和 >= 就是大部分数据类型的一对求反器。一个操作符不可能是它自身的求反器。

与交换子不同，一对一元操作符可以合法地被标记为对方的求反器。这意味着对于所有 x 有 (A x) 等于 NOT (B x)，或者对右一元操作符也相似。

一个操作符的求反器必须具有和被定义的操作符相同的左或右操作数类型，因此正如COMMUTATOR一样，NEGATOR子句中只需要给出操作符的名称即可。

提供一个求反器对查询优化器非常有帮助，因为它允许NOT (x = y)这样的表达式被简化为x <> y。这可能比你想象的更多地发生，因为NOT操作可能会被作为其他调整的结果被插入。

求反器对的定义可以使用与定义交换子对相同的方法来完成。

38.14.3. RESTRICT

如果提供了RESTRICT子句，它为该操作符指定一个限制选择度估计函数（注意这是一个函数名而不是一个操作符名）。RESTRICT子句只对返回boolean的二元操作符有意义。一个限制选择度估计器背后的思想是猜测一个表中有多大比例的行对于当前的操作符和一个特定的常数值将会满足一个

column OP constant

形式的WHERE子句条件。这能通过告知优化器具有这种形式的WHERE子句将会消除掉多少行来协助它的工作（你可能会好奇，如果常数位于左部会发生什么？好吧，COMMUTATOR就是干这个的）。

编写一个新的限制选择度估算函数已经超出了本章的范围，但是幸运地是你通常可以将系统的一个标准估算器用于很多你自己的操作符。标准的限制估算器有：

eqsel用于=
neqsel用于<>
scalarltsel用于<
scalarlesel用于<=
scalargtsel用于>
scalargesel用于>=

你能经常成功地为具有非常高或者非常低选择度的操作符使用eqsel或neqsel，即使它们实际上并非相等或不相等。例如，近似相等几何操作符使用eqsel的前提是假定它们通常只匹配表中的一小部分项。

你可以使用scalarltsel、scalarlesel、scalargtsel以及scalargesel来比较被转换为数字标量进行范围比较具有意义的数据类型。如果可能，增加一种能被src/backend/utils/adt/selfuncs.c中的函数convert_to_scalar()所理解的数据类型（最后，这个函数应该被通过pg_type系统目录的一列所标识的针对每个数据类型的函数所替换，但是那还没有发生）。如果你没有这样做，还是能工作，但是优化器的估计将不会达到最好的效果。

有一些额外的选择度估算函数是为src/backend/utils/adt/geo_selfuncs.c中的几何操作符设计的：areasel、positionsel和contsel。在写这份材料时，这些还只是存根，但是你可能想要使用它们（或者甚至改进它们）。

38.14.4. JOIN

如果提供了JOIN子句，表示用于该操作符的一个连接选择度估计函数（注意这是一个函数名而不是一个操作符名）。JOIN子句只对返回boolean的二元操作符有意义。一个连接选择度估算器背后的思想是猜测一对表中有多大比例的行对于当前的操作符将会满足一个

table1.column1 OP table2.column2

形式的WHERE子句条件。和RESTRICT子句一样，这通过让优化器知道哪种连接序列需要做的工作最少来极大地帮助优化器。

一如既往，这一章将不会尝试解释如何编写一个连接选择度估算函数，而只是建议你在适当的时候使用一种标准估算器：

eqjoinsel用于=
neqjoinsel用于<>
scalarltjoinsel用于<
scalarlejoinsel用于<=
scalargtjoinsel用于>
scalargejoinsel用于>=
areajoinsel用于基于 2D 区域比较
positionjoinsel用于基于 2D 位置比较
contjoinsel用于基于 2D 包含比较

38.14.5. HASHES

如果存在HASHES子句，它告诉系统它被许可为基于这个操作符的一个连接使用哈希连接方法。HASHES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

哈希连接之下的假设是连接操作符只能对哈希到相同哈希码的左右值返回真。如果两个值被放到不同的哈希桶中，连接将根本不会比较它们，这隐式地假定该连接操作符的结果必须是假。因此，为不表示某种形式相等的操作符指定HASHES是没有意义的。在大部分情况下，只有为在两端都是相同数据类型的操作符支持哈希才有意义。不过，有时可以为两种或更多数据类型设计兼容的哈希函数，也就是说，对于“相等”的值（即使具有不同的表达）会产生相同哈希码的函数。例如，在哈希不同宽度的证书时，安排这个属性相当简单。

要被标记为HASHES，连接操作符必须出现在一个哈希索引操作符族中。当你创建该操作符时这不会被强制，因为要引用的操作符族当然不可能已经存在。但是如果这样的操作符族不存在，尝试在哈希连接中使用该操作符将在运行时失败。系统需要用该操作符族来为操作符的输入数据类型寻找数据类型相关的哈希函数。当然，在创建操作符族之前，你还必须创建合适的哈希函数。

在准备一个哈希函数时应当慎重，因为有一些方法是依赖于机器的，这样它可能无法做正确的事情。例如，如果你的数据类型是一个结构，其中可能有无用的填充位，你不能简单地把整个结构传递给hash_any（除非你编写你自己的操作符和函数按照推荐的策略来保证未被使用的位总是为零）。另一个例子是在符合IEEE浮点标准的机器上，负数零和正数零是不同的值（不同的位模式），但是它们被定义为相等。如果一个浮点值可能包含负数零，那么需要额外的步骤来保证它产生的哈希值与正数零产生的相同。

一个可哈希连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的哈希操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

> **注意**
>
> 一个可哈希连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个哈希连接尝试使用该操作符。

> **注意**
>
> 如果一个可哈希连接的操作符有一个被标记为 strict 的底层函数，该函数必须也是 complete：也就是对于任意两个非空输入它应当返回真或假，从不会返回空。如果没有遵守这个规则，IN操作的哈希优化可能产生错误的结果（特别是，当依据标准的正确答案可能是空时，IN可能会返回假，或者它会产生一个错误来抱怨它没有准备会收到一个空结果）。

38.14.6. MERGES

如果存在MERGES子句，它告诉系统它被许可为基于这个操作符的一个连接使用归并连接方法。MERGES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

归并连接的思想是排序左右手表并且接着并行扫描它们。这样，两种数据类型必须能够被完全排序，并且该连接操作符必须只为落在排序顺序上“同一位置”的值对返回成功。实际上这意味着该连接操作符必须和相等的行为一样。但是只要两种不同的数据类型在逻辑上是兼容的，就能对它们使用归并连接。例如，smallint-versus-integer相等操作符就是可归并连接的。我们只需要将两种数据类型变成逻辑上兼容的序列的排序操作符。

要被标记为MERGES，该连接操作符必须作为一个btree索引操作符的相等成员出现。当你创建该操作符时，这不是强制的，因为要引用的操作符族当然可能还不存在。但是除非能找到一个匹配的操作符族，否则该操作符将不会被实际用于归并连接。MERGES标志因此扮演着一种对于规划器的提示，表示值得去寻找一个匹配的操作符族。

一个可归并连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的btree操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

> **注意**
>
> 一个可归并连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个归并连接尝试使用该操作符。

38.15. 索引的接口扩展

迄今为止已经描述的过程让我们能够定义新的类型、新的函数以及新的操作符。但是，我们还不能在一种新数据类型的列上定义索引。要做这件事情，我们必须为新数据类型定义一个操作符类。在这一小节稍后的部分，我们将用一个例子阐述这部份内容：一个用于 B-树索引方法的操作符类，它以绝对值的升序存储和排序复数。

操作符类可以被分组成操作符族来体现语义兼容的类之间的联系。当只涉及到一种单一数据类型时，一个操作符类就足矣。因此我们将先把重点放在这种情况上，然后再回到操作符族。

38.15.1. 索引方法和操作符类

pg_am表为每一种索引方法都包含一行（内部被称为访问方法）。PostgreSQL中内建了对表常规访问的支持，但是所有的索引方法则是在pg_am中描述。可以通过编写必要的代码并且在pg_am中创建一项来增加一种新的索引访问方法 — 但这超出了本章的范围（见第 61 章。

一个索引方法的例程并不直接了解它将要操作的数据类型。而是由一个操作符类标识索引方法用来操作一种特定数据类型的一组操作。之所以被称为操作符类是因为它们指定的一件事情就是可以被用于一个索引的WHERE子句操作符集合（即，能被转换成一个索引扫描条件）。一个操作符类也能指定一些索引方法内部操作所需的支持函数，这些过程不能直接对应于能用于索引的任何WHERE子句操作符。

可以为相同的数据类型和索引方法定义多个操作符类。通过这种方式，可以为一种数据类型定义多个索引语义集合。例如，一个B-树索引要求在它要操作的每一种数据类型上都定义一个排序顺序。对一种复数数据类型来说，拥有一个可以根据复数绝对值排序的 B-树操作符类和另一个可以根据实数部分排序的操作符类可能会有用。典型地，其中一个操作符类将被认为是最常用的并且将被标记为那种数据类型和索引方法的默认操作符类。

相同的操作符类名称可以被用于多个不同的索引方法（例如，B-树和哈希索引方法都有名为int4_ops的操作符类）。但是每一个这样的类都是一个独立实体并且必须被单独定义。

38.15.2. 索引方法策略

与一个操作符类关联的操作符通过“策略号”标识，它被用来标识每个操作符在其操作符类中的语义。例如，B-树在键上施行了一种严格的顺序（较小到较大），因此“小于”和“大于等于”这样的操作符就是 B-树所感兴趣的。因为PostgreSQL允许用户定义操作符，PostgreSQL不能看着一个操作符（如<和>=）的名字并且说出它是哪一种比较。取而代之的是，索引方法定义了一个“策略”集合，它们可以被看成是广义的操作符。每一个操作符类会说明对于一种特定的数据类型究竟是哪个实际的操作符对应于每一种策略以及该索引语义的解释。

B-树索引方法定义了五种策略，如表 38.20 所示。

表 38.2. B-树策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

哈希索引只支持等值比较，因此它们只使用一种策略，如表 38.3所示。

表 38.3. 哈希策略

操作	策略号
等于	1

GiST 索引更加灵活：它们根本没有一个固定的策略集合。取而代之的是，每一个特定 GiST 操作符类的“consistency”支持例程会负责解释策略号。例如，一些内建的 GiST 索引操作符类索引二维几何对象，它们提供表 38.4中所示的“R-树”策略。其中四个是真正的二维测试（重叠、相同、包含、被包含），其中四个只考虑 X 方向，其他四个提供 Y 方向上的相同测试。

表 38.4. GiST 二维“R-树” 策略

操作	策略号
左参数严格地位于右参数的左边	1
左参数不会延伸到右参数的右边	2
重叠	3
左参数不会延伸到右参数的左边	4
左参数严格地位于右参数的右边	5
相同	6
包含	7
被包含	8
不会延伸到高于	9
严格低于	10
严格高于	11
不会延伸到低于	12

SP-GiST 索引在灵活性上与索引相似：它们没有一个固定的策略集合。取而代之的是，每一个操作符类的支持例程负责根据该操作符类的定义解释策略号。例如，被内建操作符类用于点的策略号如表 38.5中所示。

表 38.5. SP-GiST 点策略

操作	策略号
左参数严格地位于右参数的左边	1
左参数严格地位于右参数的右边	5
相同	6
被包含	8
严格地低于	10

操作	策略号
严格地高于	11

GIN 索引与 GiST 和 SP-GiST 索引类似，它们也没有一个固定的策略集合。取而代之的是，每一个操作符类的支持例程负责根据该操作符类的定义解释策略号。例如，被内建操作符类用于数组的策略号如表 38.6所示。

表 38.6. GIN 数组策略

操作	策略号
重叠	1
包含	2
被包含	3
等于	4

在没有固定的策略集合这一点上，BRIN 索引和 GiST、SP-GiST 和 GIN 索引是类似的。每一个操作符类的支持函数会根据操作符类的定义解释策略编号。例如，表 38.7中展示了内建的Minmax操作符类所使用的策略编号。

表 38.7. BRIN 最小最大策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

注意上文列出的所有操作符都返回布尔值。实际上，所有作为索引方法搜索操作符定义的操作符必须返回类型boolean，因为它们必须出现在一个WHERE子句的顶层来与一个索引一起使用（某些索引访问方法还支持排序操作符，它们通常不返回布尔值，这种特性在第 38.15.7 节讨论）。

38.15.3. 索引方法支持例程

对于系统来说只有策略信息通常不足以断定如何使用一种索引。实际上，为了能工作，索引方法还要求额外的支持例程。例如，B-树索引方法必须能比较两个键并且决定其中一个是否大于、等于或小于另一个。类似地，哈希索引方法必须能够为键值计算哈希码。这些操作并不对应在 SQL 命令的条件中使用的操作符。它们是索引方法在内部使用的管理例程。

与策略一样，操作符类会标识哪些函数应该为一种给定的数据类型扮演这些角色以及相应的语义解释。索引方法定义它需要的函数集合，而操作符类则会通过为函数分配由索引方法说明的“支持函数号”来标识正确的函数。

如表 38.8所示， B-树要求一个比较支持函数，并且允许在操作符类作者的选项中提供两个额外的支持函数。这些支持函数的要求在第 63.3 节会进一步解释。

表 38.8. B-树支持函数

函数	支持号
比较两个键并且返回一个小于零、等于零或大于零的整数，它表示第一个键小于、等于或者大于第二个键。	1
返回C可调用的排序支持函数的地址（可选）。	2

函数	支持号
将一个测试值与一个基础值加上/减去一个偏移量的结果进行比较，根据比较的结果返回真或假（可选）	3

如表 38.9所示，哈希索引要求一个支持函数，并且允许在操作符类作者的选项中提供第二个支持函数。

表 38.9. 哈希支持函数

函数	支持号
为一个键计算32位哈希值	1
给定一个64-位salt，计算一个键的64位哈希值。如果salt为0，结果的低32位必须匹配会由函数1计算出来的值（可选）	2

如表 38.10所示，GiST 索引有九个支持函数，其中两个是可选的（详见第 64 章。

表 38.10. GiST 支持函数

函数	描述	支持号
consistent	判断键是否满足查询修饰语	1
union	计算一个键集合的联合	2
compress	计算一个要被索引的键或值的压缩表达	3
decompress	计算一个压缩键的解压表达	4
penalty	计算把新键插入到带有给定子树键的子树中带来的罚值	5
picksplit	判断一个页面中的哪些项要被移动到新页面中并且计算结果页面的联合键	6
equal	比较两个键并且在它们相等时返回真	7
distance	判断键到查询值的距离（可选）	8
fetch	为只用索引扫描计算一个压缩键的原始表达（可选）	9

如表 38.11所示，SP-GiST 索引要求五个支持函数（详见第 65 章。

表 38.11. SP-GiST 支持函数

函数	描述	支持号
config	提供有关该操作符类的基本信息	1
choose	判断如何把一个新值插入到一个内元组中	2
picksplit	判断如何划分一组值	3
inner_consistent	判断对于一个查询需要搜索哪一个子划分	4
leaf_consistent	判断键是否满足查询修饰语	5

如表 38.12所示，GIN 索引有六个支持函数，其中三个是可选的（详见第 66 章。

表 38.12. GIN 支持函数

函数	描述	支持号
compare	比较两个键并且返回一个小于零、等于零或大于零的整数，它表示第一个键小于、等于或者大于第二个键	1
extractValue	从一个要被索引的值中抽取键	2
extractQuery	从一个查询条件中抽取键	3
consistent	判断值是否匹配查询条件（布尔变体）（如果支持函数 6 存在则是可选的）	4
comparePartial	比较来自查询的部分键和来自索引的键，并且返回一个小于零、等于零或大于零的整数，表示 GIN 是否应该忽略该索引项、把该项当做一个匹配或者停止索引扫描（可选）	5
triConsistent	判断值是否匹配查询条件（三元变体）（如果支持函数 4 存在则是可选的）	6

如表 38.13中所示，BRIN 索引具有四个基本的支持函数。这些基本函数可能会要求提供额外的支持函数（更多信息请见第 67.3 节。

表 38.13. BRIN 支持函数

函数	描述	支持编号
opcInfo	返回描述被索引列的摘要数据的内部信息	1
add_value	向一个现有的摘要索引元组增加一个新值	2
consistent	判断值是否匹配查询条件	3
union	计算两个摘要元组的联合	4

和搜索操作符不同，支持函数返回特定索引方法所期望的数据类型，例如在 B 树的比较函数中是一个有符号整数。每个支持函数的参数数量和类型也取决于索引方法。对于 B 树和哈希，比较和哈希支持函数和包括在操作符类中的操作符接收一样的输入数据类型，但是大部分 GiST、SP-GiST、GIN 和 BRIN 支持函数则不是这样。

38.15.4. 一个例子

现在我们已经看过了基本思想，这里是创建一个新操作符类的例子（可以在源代码的src/tutorial/complex.c和src/tutorial/complex.sql中找到这个例子）。该操作符类封装了以绝对值顺序排序复数的操作符，因此我们为它取名为complex_abs_ops。首先，我们需要一个操作符集合。定义操作符的过程已经在第 38.13 节讨论过。对于一个 B-树上的操作符类，我们需要的操作符有：

- 绝对值小于（策略 1）
- 绝对值小于等于（策略 2）
- 绝对值等于（策略 3）

- 绝对值大于等于（策略 4）
- 绝对值大于（策略 5）

定义一个比较操作符的相关集合最不容易出错的方式是，先编写 B-树比较支持函数，然后编写该支持函数的包装器函数。这降低了极端情况下得到不一致结果的几率。遵照这种方法，我们首先编写：

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

现在小于函数看起来像这样：

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其他四个函数的区别只在于它们如何比较内部函数的结果与 0。

接下来我们基于这些函数声明 SQL 的函数和操作符：

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

指定正确的交换子和求反器操作符很重要，合适的限制和连接选择度函数也是一样，否则优化器将无法有效地利用索引。

其他值得注意的事情：

- 只能有一个操作符被命名为=且两个操作数都为类型complex。在这种要求下，我们对于complex没有任何其他操作符=。但是如果我们是在构建一种实际的数据类型，我们可能

想让=成为复数的普通等值操作（不是绝对值的相等）。这样，我们需要为complex_abs_eq使用某种其他的操作符名称。

- 尽管PostgreSQL能够处理具有相同 SQL 名称的函数（只要它们具有不同的参数数据类型），但 C 只能处理具有给定名称一个全局函数。因此，我们不能简单地把 C 函数命名为abs_eq之类的东西。通常，在 C 函数名中包括数据类型的名称是一种好习惯，这样就不会与其他数据类型的函数发生冲突。
- 我们可以让函数也具有abs_eq这样的 SQL 名称，而依靠PostgreSQL通过参数数据类型来区分它和其他同名 SQL 函数。为了保持例子的简洁，我们这里让 C 级别和 SQL 级别的函数具有相同的名称。

下一步是注册 B-树所要求的支持例程。实现支持例程的 C 代码例子在包含操作符函数的同一文件中。我们这样来声明该函数：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

现在我们已经有了所需的操作符和支持例程，就可以最终创建操作符类：

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

做好了！现在应该可以在complex列上创建并且使用 B-树索引了。

我们可以把操作符项写得更繁琐，像这样：

```
        OPERATOR        1       < (complex, complex) ,
```

但是当操作符操作的数据类型和正在定义的操作符类所服务的数据类型相同时可以不用这么做。

上述例子假定这个新操作符类是complex数据类型的默认 B-树操作符类。如果不是这样，只需要省去关键词DEFAULT。

38.15.5. 操作符类和操作符族

到目前为止，我们暗地里假设一个操作符类只处理一种数据类型。虽然在一个特定的索引列中必定只有一种数据类型，但是把被索引列与一种不同数据类型的值比较的索引操作通常也很有用。还有，如果与一种操作符类相关的扩数据类型操作符有用，通常情况是其他数据类型也有其自身相关的操作符类。在相关的类之间建立起明确的联系会很有用，因为这可以帮助规划器进行 SQL 查询优化（尤其是对于 B-树操作符类，因为规划器包含了大量有关如何使用它们的知识）。

为了处理这些需求，PostgreSQL使用了操作符族的概念。一个操作符族包含一个或者多个操作符类，并且也能包含属于该族整体而不属于该族中任何单一类的可索引操作符和相应的支持函数。我们说这样的操作符和函数是“松散地”存在于该族中，而不是被绑定在一个特定的类中。通常每个操作符类包含单一数据类型的操作符，而跨数据类型操作符则松散地存在于操作符族中。

一个操作符族中的所有操作符和函数必须具有兼容的语义，其中的兼容性要求由索引方法设定。你可能因此而奇怪为什么要这么麻烦地把族的特定子集单另出来成为操作符类，并且实际上（由于很多原因）这种划分与操作符之间没有什么直接的关联，只有操作符族才是实际的分组。定义操作符类的原因是，它们指定了特定索引对操作符族的依赖程度。如果一个索引使用着一个操作符类，那么不删除该索引是不能删除该操作符类的 — 但是操作符族的其他部分（即其他操作符类和松散的操作符）可以被删除。因此，一个操作符类应该包含一个索引在特定数据类型上正常工作所需要的最小操作符和函数集合，而相关但不关键的操作符可以作为操作符族的松散成员被加入。

例如，PostgreSQL有一个内建的 B-树操作符族integer_ops，它包括分别用于类型bigint(int8)、integer (int4)和smallint (int2)列上索引的操作符类int8_ops、int4_ops以及int2_ops。这个族也包含跨数据类型比较操作符，它们允许对这些类型中的任意两种进行比较，这样可以通过一种类型的比较值来搜索另一种类型之上的索引。这个族可以用这些定义来重现：

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- 标准 int8 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- 标准 int4 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- 标准 int2 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ,
  FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- 跨类型比较 int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
```

```
OPERATOR 3 = (int8, int2) ,
OPERATOR 4 >= (int8, int2) ,
OPERATOR 5 > (int8, int2) ,
FUNCTION 1 btint82cmp(int8, int2) ,

-- 跨类型比较 int8 vs int4
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- 跨类型比较 int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- 跨类型比较 int4 vs int8
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- 跨类型比较 int2 vs int8
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- 跨类型比较 int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ,

-- 跨类型的in_range函数
FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

注意这种定义“重载”了操作符策略和支持函数号：每一个编号在该族中出现多次。只要一个特定编号的每一个实例都有可区分的输入数据类型，就允许这样做。输入类型等于操作符类输入类型的实例是该操作符类的主要操作符和支持函数，并且在大部分情况下应该被声明为该操作符类的一部分而不是作为操作符族的松散成员存在。

如第 63.2 节的细节所述，在一个 B-树操作符族中，所有该族中的操作符必须以兼容的方式排序。对该族中的每一个操作符都必须有一个与该操作符具有相同的两个输入数据类型的支持函数。我们推荐让操作符族保持完整，即对每一种数据类型的组合都应该包括所有的操作符。每个操作符类只应该包括非跨类型操作符和用于其数据类型的支持函数。

为了构建一个多数据类型的哈希操作符族，必须为该族支持的每一种数据类型创建相兼容的哈希支持函数。这里的兼容性是指这些函数对于任意两个被该族中等值操作符认为相等的值会保证返回相同的哈希码，即便这些值具有不同的类型时也是如此。当这些类型具有不同的物理表示时，这通常难以实现，但是在某些情况下是可以做到的。此外，将该操作符族中一种数据类型的值通过隐式或者二进制强制造型转换成该族中另一种数据类型时，不应该改变所计算出的哈希值。注意每种数据类型只有一个支持函数，而不是每个等值操作符一个。我们推荐让操作符族保持完整，即对每一种数据类型的组合提供一个等值操作符。每个操作符类只应该包括非跨类型等值操作符和用于其数据类型的支持函数。

GiST、SP-GiST 和 GIN 索引没有任何明显的跨数据类型操作的概念。它们所支持的操作符集合就是一个给定操作符类能够处理的主要支持函数。

在 BRIN 中，需求取决于提供操作符类的框架。对于基于minmax的操作符类，必要的行为和 B-树操作符族相同：族中的所有操作符必须以兼容的方式排序，并且转换不能改变相关的排序顺序。

> **注意**
>
> 在PostgreSQL 8.3 之前，没有操作符族的概念，并且因此要在索引中使用的任何跨数据类型操作符必须被直接绑定到该索引的操作符类中。虽然这种方法仍然有效，但是已被废弃，因为它会让索引的依赖过于广泛，还因为当两种数据类型都在同一操作符族中有操作符时规划器可以更有效地处理跨数据类型比较。

38.15.6. 操作符类上的系统依赖

PostgreSQL使用操作符类来以更多方式推断操作符的属性，而不仅仅是它们是否能被用于索引。因此，即便不准备对你的数据类型的列建立索引，也可能想要创建操作符类。

特别地，ORDER BY和DISTINCT等 SQL 特性要求对值的比较和排序。为了在用户定义的数据类型上实现这些特性，PostgreSQL会为数据类型查找默认 B-树操作符类。这个操作符类的“equals”成员定义了用于GROUP BY和DISTINCT的值的等值概念，而该操作符类施加的排序顺序定义了默认的ORDER BY顺序。

如果一种数据类型没有默认的 B-树操作符类，系统将查找默认的哈希操作符类。但由于这类操作符类只提供等值，所以它只能支持分组而不能支持排序。

在一种数据类型没有默认操作符类时，如果尝试对该数据类型使用这些 SQL 特性，你将得到类似“could not identify an ordering operator”（无法标识排序操作符）的错误。

> **注意**
>
> 在版本 7.4 以前的PostgreSQL中，排序和分组操作将隐式地使用名为=、<以及>的操作符。新的依赖于默认操作符类的行为避免了对具有特定名字的操作符行为作出任何假设。

通过在一个USING选项中指定一个非默认B-树操作符类的小于操作符，可以使用该操作符进行排序，例如

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

或者，在USING中指定该操作符类的大于操作符可以选择升序的排序。

用户定义类型的数组的比较还依赖于该类型的默认B-树操作符类所定义的语义。如果没有默认的B-树操作符类，但有一个默认的哈希操作符类，则支持数组的相等比较，但不支持顺序的比较。

另一种要求更多数据类型相关知识的SQL特性是窗口函数（见第 4.2.8 节的RANGE offset PRECEDING/FOLLOWING帧选项。对于这样的一个查询

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
  FROM mytable;
```

不足以了解如何用x进行排序，数据库还必须理解如何对当前行的x值“减5”或者“加10”以标识当前窗口帧的边界。把得到的边界与其他行的x值用B-树操作符类提供的比较操作符（定义了ORDER BY顺序）进行比较是可能的 — 但是加和减操作符并不是该操作符类的一部分，因此应该用哪些操作符呢？硬编码的选择是不切实际的，因为不同的排序顺序（不同的B-树操作符）可能需要不同的行为。因此，一个B-树操作符类可以指定一个in_range支持函数，它封装有对排序顺序有意义的加和减行为。如果有多种数据类型可以用作RANGE子句中的偏移量，甚至可以提供多个in_range支持函数。如果与窗口的ORDER BY子句关联的B-树操作符类没有一个匹配的in_range支持函数，则不支持RANGE offset PRECEDING/FOLLOWING选项。

另一个要点是，出现在一个哈希操作符族中的操作符是哈希连接、哈希聚集和相关优化的候选。这些情况下哈希操作符族就是至关重要的，因为它标识了要使用的哈希函数。

38.15.7. 排序操作符

有些索引访问方法（当前只有 GiST）支持排序操作符的概念。到目前为止我们所讨论的都是搜索操作符。搜索索引时，会用搜索操作符来寻找所有满足 WHERE indexed_column operator constant 的行。注意被返回的匹配行的顺序是没有任何保证的。相反，一个排序操作符并不限制能被返回的行集合，而是决定它们的顺序。扫描索引时，会使用排序操作符来以 ORDER BY indexed_column operator constant 所表示的顺序返回行。这样定义排序操作符的原因是，如果该操作符能度量距离，它就能支持最近邻搜索。例如，这样的一个查询

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

寻找离一个给定目标点最近的十个位置。位置列上的 GiST 索引可以有效地完成这个查询，因为<->是一个排序操作符。

搜索操作符必须返回布尔结果，排序操作符通常返回某种其他类型，例如浮点、数字或者距离。这种类型通常不同于被索引的数据类型。为了避免硬编码有关不同数据类型行为的假设，需要定义一个排序操作符来提名一个 B-树操作符族指定结果数据类型的排序顺序。正如我们在前一节介绍的，B-树操作符族定义了PostgreSQL的顺序概念，因此这是一种自然的表达。由于点<->操作符返回float8，可以在一个操作符类创建命令中这样指定它：

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

其中float_ops是包括float8上操作的内建操作符族。这种声明说明该索引能够以<->操作符的递增值顺序返回行。

38.15.8. 操作符类的特性

有两个操作符类的特性我们还没有讨论，主要是因为它们对于最常用的索引方法不太有用。

通常，把一个操作符声明为一个操作符类（或操作符族）的成员意味着该索引方法能够使用该操作符准确地检索满足WHERE条件的行集。例如：

```
SELECT * FROM table WHERE integer_column < 4;
```

恰好可以被该整数列上一个 B-树索引满足。但是也有情况下索引只是作为匹配行的非精确向导。例如，如果一个 GiST 索引只存储几何对象的边界框，那么它无法精确地满足测试非矩形对象（如多边形）之间相交的WHERE条件。但是我们可以使用该索引来寻找边界框与目标对象的边界框相交的对象，并且只在通过该索引找到的对象上做精确的相交测试。如果适用于这种场景，该索引被称为对该操作符是“有损的”。有损索引搜索通过在一行可能满足或者不满足该查询条件时返回一个recheck标志来实现。核心系统将接着在检索到的行上测试原始查询条件来看它是否应该被作为一个合法匹配返回。如果索引被保证能返回所有所需的行外加一些额外的行，这种方法就能有效，因为那些额外的行可以通过执行原始的操作符调用来消除。支持有损搜索的索引方法（当前有 GiST、SP-GiST 和 GIN）允许个别操作符类的支持函数设置 recheck 标志，因此这也是一种操作符类的重要特性。

再次考虑在索引中只存储复杂对象（如多边形）的边界框的情况。在这种情况下，把整个多边形存储在索引项中没有很大价值 — 我们也可以只存储一个更简单的box类型对象。这种情况通过CREATE OPERATOR CLASS中的STORAGE选项表示：

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

当前，只有 GiST、GIN 和 BRIN 索引方法支持不同于列数据类型的STORAGE类型。在使用STORAGE时，GiST 的支持例程compress和decompress必须处理数据类型转换。在 GIN 中，STORAGE类型标识“key”值的类型，它通常不同于被索引列的类型 — 例如，一个用于整数数组列的操作符类可能具有整数键值。GIN 的支持例程extractValue和extractQuery负责从被索引值中抽取键。BRIN 类似于 GIN：STORAGE类型标识被存储的摘要值的类型，而操作符类的支持过程负责正确解释摘要值。

38.16. 打包相关对象到一个扩展中

一个对PostgreSQL有用的扩展通常包括多个 SQL 对象，例如，一种新的数据类型将需要新函数、新操作符以及可能的新索引操作符类。将所有这些对象收集到一个单一包中有助于简化数据库管理。PostgreSQL称这样一个包为一个扩展。要定义一个扩展，你至少需要一个包含创建该扩展的对象的SQL命令的脚本文件以及一个指定扩展本身的一些基本属性的控制文件。如果扩展包括 C 代码，通常还有一个 C 代码编译而成的共享库文件。一旦你有了这些文件，一个简单的CREATE EXTENSION命令可以把这些对象载入到你的数据库。

使用一个扩展而不是只运行SQL脚本载入一堆“松散”对象到数据库的主要优点是，PostgreSQL将能理解该扩展的对象是一起的。你可以用一个单一的DROP EXTENSION命令删除所有的对象（不用维护一个单独的“卸载”脚本）。 甚至更有用的一点是，pg_dump知道它不应该转储该扩展中的个体成员对象 — 它将只在转储中包括一个CREATE EXTENSION命令。这大大简化了迁移到一个包含不同于旧版扩展中对象的新版扩展的工作。不过，注意在把这样一个转储载入到一个新数据库时，该扩展的控制、脚本和其他文件必须可用。

PostgreSQL不会让你删除包含在一个扩展中的个体对象，除非删除整个扩展。还有，虽然你能够改变一个扩展的成员对象的定义（例如，通过CREATE OR REPLACE FUNCTION改变一个函数），记住被修改后的定义将不会被pg_dump转储。这种改变通常只有在你并发地在扩展脚本文件中做出相同更改时才有意义（但是对于包含配置数据的表有特殊的规定，见第 38.16.4 节。在生产环境中，通常更好的方式是创建一个扩展更新脚本来执行对扩展中成员对象的更改。

扩展脚本可能会通过GRANT和REVOKE语句设置扩展中所含对象的特权。每一个对象的最终特权集合（如果设置了任何特权）将被存储在pg_init_privs系统目录中。使用pg_dump时，CREATE EXTENSION命令将被包括在转储中，后面会跟着必要的GRANT和REVOKE语句集合来将对象的特权设置成取得该转储时的样子。

PostgreSQL当前不支持扩展脚本发出CREATE POLICY或者SECURITY LABEL语句。这些东西的设置应该在扩展被创建好之后来进行。所有在扩展对象上创建的 RLS 策略和安全标签都将被包括在pg_dump创建的转储中。

扩展机制也对打包调整一个扩展中所含 SQL 对象定义的修改脚本有规定。例如，如果一个扩展的 1.1 版本比 1.0 版本增加了一个函数并且更改了另一个函数的函数体，该扩展的作者可以提供一个更新脚本来做这两个更改。那么ALTER EXTENSION UPDATE命令可以被用来应用这些更改并且跟踪在给定数据库中实际安装的是该扩展的哪个版本。

能作为一个扩展的成员的 SQL 对象的种类如ALTER EXTENSION所示。尤其是数据库集簇范围的对象（例如数据库、角色和表空间）不能作为扩展成员，因为一个扩展只在一个数据库范围内可见（尽管一个扩展脚本并没有被禁止创建这些对象，但是这样做将无法把它们作为扩展的一部分来跟踪）。还要注意虽然一个表可以是一个扩展的成员，它的扶助对象（例如索引）不会被直接认为是该扩展的成员。另一个重点是模式可以属于扩展，但是反过来不行：一个扩展本身有一个不被限定的名称并且不存在于任何模式“中”。不过，扩展的成员对象只要对象类型合适就可以属于模式。一个扩展拥有包含其成员对象的模式可能合适也可能不合适。

如果一个扩展的脚本创建任何临时对象（例如临时表），在当前会话的剩余部分会把它们当作扩展的成员，但是在会话结束会自动删除它们，这和任何其他临时对象是一样的。对于不删除整个扩展就不能删除扩展的成员对象的规则来说，这是一种例外。

38.16.1. 定义扩展对象

广泛分发的扩展应该尽量少地假定它们所占据的数据库。特别是，除非你发出了SET search_path = pg_temp，应该假定每一个未限定的名称都可能解析成恶意用户定义的对象。要小心隐式依赖于search_path的结构：IN以及CASE expression WHEN总是使用搜索路径选择操作符。对于它们，可使用OPERATOR(schema.=) ANY和CASE WHEN expression。

38.16.2. 扩展文件

CREATE EXTENSION命令依赖每一个扩展都有的控制文件，控制文件必须被命名为扩展的名称加上一个后缀.control，并且必须被放在安装的SHAREDIR/extension目录中。还必须至少有一个SQL脚本文件，它遵循命名模式extension--version.sql（例如，foo--1.0.sql表示扩展foo的1.0版本）。默认情况下，脚本文件也被放置在SHAREDIR/extension目录中，但是控制文件中可以为脚本文件指定一个不同的目录。

一个扩展控制文件的格式与postgresql.conf文件相同，即是一个parameter_name = value赋值的列表，每行一个。允许空行和#引入的注释。注意对任何不是单一词或数字的值加上引号。

一个控制文件可以设置下列参数：

directory (string)

包含扩展的SQL脚本文件的目录。除非给出一个绝对路径，这个目录名是相对于安装的SHAREDIR目录。默认行为等效于指定directory = 'extension'。

default_version (string)

该扩展的默认版本（就是如果在CREATE EXTENSION中没有指定版本时将会被安装的那一个）。尽管可以忽略这个参数，但如果没有出现VERSION选项时那将会导致CREATE EXTENSION失败，因此你通常不会希望这样做。

comment (string)

一个关于该扩展的注释（任意字符串）。该注释会在初始创建扩展时应用，但是扩展更新时不会引用该注释（因为可能会覆盖用户增加的注释）。扩展的注释也可以通过在脚本文件中写上COMMENT命令来设置。

encoding (string)

该脚本文件使用的字符集编码。当脚本文件包含任何非 ASCII 字符时，可以指定这个参数。否则文件都会被假定为数据库编码。

module_pathname (string)

这个参数的值将被用于替换脚本文件中每一次出现的MODULE_PATHNAME。如果设置，将不会进行替换。通常，这会被设置为$libdir/shared_library_name并且接着MODULE_PATHNAME被用在CREATE FUNCTION命令中进行 C-语言函数的创建，因此该脚本文件不必把共享库的名称硬编码在其中。

requires (string)

这个扩展依赖的其他扩展名的一个列表，例如requires = 'foo, bar'。被依赖的扩展必须先于这个扩展安装。

superuser (boolean)

如果这个参数为true（默认情况），只有超级用户能够创建该扩展或者将它更新到一个新版本。如果被设置为false，只需要用来执行安装中命令或者更新脚本的特权。

relocatable (boolean)

如果一个扩展可能在初始创建之后将其所含的对象移动到一个不同的模式中，它就是relocatable。默认值是false，即该扩展是不可重定位的。详见第 38.16.3 节

schema (string)

这个参数只能为非可重定位扩展设置。它强制扩展被载入到给定的模式中而非其他模式中。只有在初始创建一个扩展时才会参考schema参数，扩展更新时则不会参考这个参数。详见第 38.16.3 节

除了主要控制文件extension.control，一个扩展还可以有二级控制文件，它们以extension--version.control的风格命名。如果提供了二级控制文件，它们必须被放置在脚本文件的目录中。二级控制文件遵循主要控制文件相同的格式。在安装或更新该扩展的版本时，一个二级控制文件中设置的任何参数将覆盖主要控制文件中的设置。不过，参数directory以及default_version不能在二级控制文件中设置。

一个扩展的SQL脚本文件能够包含任何 SQL 命令，除了事务控制命令（BEGIN、COMMIT等）以及不能在一个事务块中执行的命令（如VACUUM）。这是因为脚本文件会被隐式地在一个事务块中被执行。

一个扩展的SQL脚本文件也能包含以\echo开始的行，它将被扩展机制忽略（当作注释）。如果脚本文件被送给psql而不是由CREATE EXTENSION载入（见第 38.16.7 节的示例脚本），这种机制通常被用来抛出错误。如果没有这种功能，用户可能会意外地把该扩展的内容作为“松散的”对象而不是一整个扩展载入，这样的状态恢复起来比较麻烦。

尽管脚本文件可以包含指定编码允许的任何字符，但是控制文件应该只包含纯 ASCII 字符，因为PostgreSQL没有办法知道一个控制文件是什么编码。实际上，如果你想在扩展的注释中使用非 ASCII 字符只有一个问题。推荐的方法是不使用控制文件的comment参数，而是使用脚本文件中的COMMENT ON EXTENSION来设置注释。

38.16.3. 扩展可再定位性

用户常常希望把扩展中包含的对象载入到一个与扩展的作者所设想的不一样的模式中。对于这种可重定位性，有三种支持的级别：

- 一个完全可重定位的扩展能在任何时候被移动到另一个模式中，即使在它被载入到一个数据库中之后。这种移动通过ALTER EXTENSION SET SCHEMA命令完成，该命令会自动地把所有成员对象重命名到新的模式中。通常，只有扩展不包含任何对其所在模式的内部假设时

才可能这样做。还有，该扩展的对象必须全部在同一个模式中（忽略那些不属于任何模式的对象，例如过程语言）。要让一个扩展变成完全可定位，在它的控制文件中设置relocatable = true。

- 一个扩展可能在安装过程中是可重定位的，但是安装完后就不再可重定位。典型的情况是扩展的脚本文件需要显式地引用目标模式，例如为 SQL 函数设置search_path属性。对于这样一种扩展，在其控制文件中设置relocatable = false，并且使用@extschema@在脚本文件中引用目标模式。在脚本被执行前，所有这个字符串的出现都将被替换为实际的目标模式名。用户可以使用CREATE EXTENSION的SCHEMA选项设置目标模式名。

- 如果扩展根本就不支持重定位，在它的控制文件中设置relocatable = false，并且还设置schema为想要的目标模式的名称。这将阻止使用CREATE EXTENSION的SCHEMA选项修改目标模式，除非它指定的是和控制文件中相同的模式。如果该扩展包括关于模式名的内部假设且模式名不能使用@extschema@的方法替换，这种选择通常是必须的。@extschema@替换机制在这种情况中也是可用的，不过由于模式名已经被控制文件所决定，它的使用受到了很大的限制。

在所有情况下，脚本文件将被用search_path执行，它最初会被设置为指向目标模式，也就是说CREATE EXTENSION做的也是等效的工作：

```
SET LOCAL search_path TO @extschema@;
```

这允许由这个脚本文件创建的对象进入到目标模式中。如果脚本文件希望，它可以改变search_path，但这种用法通常是不受欢迎的。在CREATE EXTENSION结束后，search_path会被恢复到之前的设置。

如果控制文件中给出了schema参数，目标模式就由该参数决定，否则目标模式由CREATE EXTENSION的SCHEMA选项给出，如果以上两者都没有给出则会用当前默认的对象创建模式（在调用者search_path中的第一个）。当使用扩展文件的schema参数时，如果目标模式还不存在将创建它，但是在另外两种情况下它必须已经存在。

如果在控制文件中的requires中列举了任何先导扩展，它们的目标模式会被追加到search_path的初始设置中。这允许新扩展的脚本文件能够看到它们的对象。

尽管一个不可重定位的扩展能够包含散布在多个模式中的对象，通常还是值得将意图用于外部使用的所有对象放置在一个模式中，这被认为是该扩展的目标模式。这样一种安排可以在依赖的扩展创建过程中方便地与search_path的默认设置一起工作。

38.16.4. 扩展配置表

某些扩展包括配置表，其中包含可以由用户在扩展安装后增加或修改的数据。通常，如果一个表是一个扩展的一部分，该表的定义和内容都不会被pg_dump转储。但是对于一个配置表来说并不希望是这样的行为，任何用户做出的数据修改都需要被包括在转储中，否则该扩展在重载之后的行为将和转储之前不同。

要解决这个问题，一个扩展的脚本文件可以把一个它创建的表或者序列标记为配置关系，这将导致pg_dump把该表或者序列的内容（而不是它的定义）包括在转储中。要这样做，在创建表或序列之后调用函数pg_extension_config_dump(regclass, text)，例如

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

可以用这种方法标记任意数量的表或者序列。与serial或者bigserial列相关联的序列也可以被标记。

当pg_extension_config_dump的第二个参数是一个空字符串时，该表的全部内容都会被pg_dump转储。这通常只有在表被扩展脚本创建为初始为空时才正确。如果在表中混合有初始数据和用户提供的数据，pg_extension_config_dump的第二个参数提供了一种WHERE条件来选择要被转储的数据。例如，你可能会做

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT
 standard_entry');
```

并且确保只有扩展脚本创建的行中standard_entry才为真。

对于序列，pg_extension_config_dump的第二个参数没有影响。

更复杂的情况（例如用户可能会修改初始提供的数据）可以通过在配置表上创建触发器来处理，触发器将负责保证被修改的行会被正确地标记。

你可通过再次调用pg_extension_config_dump来修改与一个配置表相关的过滤条件（这通常对于一个扩展更新脚本有用）。将一个表标记为不再是一个配置表的方法是用ALTER EXTENSION ... DROP TABLE将它与扩展脱离开。

注意这些表之间的外键关系将会指导这些表被 pg_dump 转储的顺序。特别地，pg_dump 将尝试 先转储被引用的表再转储引用表。由于外键关系是在 CREATE EXTENSION 时间（先于数据被载入　到表中）建立的，环状依赖还没有建立。当环状依赖存在时，数据将仍然被转储，但是该转储无法被 直接恢复并且必须要用户的介入。

与serial或者bigserial列相关联的序列需要被直接标记以转储它们的状态。只标记它们的父关系不足以转储它们的状态。

38.16.5. 扩展更新

扩展机制的一个优点是它提供了方便的方法来管理那些定义扩展中对象的 SQL 命令的更新。这是通过为扩展的安装脚本的每一个发行版本关联一个版本名称或者版本号实现的。此外，如果你希望用户能够动态地把他们的数据库从一个版本更新到下一个版本，你应该提供更新脚本来做必要的更改。更新脚本的名称遵循extension--oldversion--newversion.sql模式（例如，foo--1.0--1.1.sql包含着把扩展foo的版本1.0修改成版本1.1的命令）。

假定有一个合适的更新脚本可用，命令ALTER EXTENSION UPDATE将把一个已安装的扩展更新到指定的新版本。更新脚本运行在与CREATE EXTENSION提供给安装脚本相同的环境中：特别是search_path会按照相同的方式设置，并且该脚本创建的任何新对象会被自动地加入到扩展中。此外，如果脚本选择删除扩展的成员对象，它们会自动与扩展解除关联。

如果一个扩展具有二级控制文件，用于更新脚本的控制参数是那些与新目标版本相关的参数。

更新机制可以被用来解决一种重要的特殊情况：将一个“松散的”对象集合转变成一个扩展。在扩展机制被加入到PostgreSQL（从 9.1 开始）之前，很多人编写的扩展模块简单地创建各种各样未打包的对象。给定一个包含这类对象的现有数据库，我们怎样才能将这些对象转变成一个正确打包的扩展？将它们全部删除然后做一次CREATE EXTENSION是一种方法，但是如果对象之间有依赖（例如，如果有一些表列使用了扩展创建的数据类型）这就行不通。修正这种情况的方法是创建一个空扩展，然后使用ALTER EXTENSION ADD把每一个以前就存在的对象附着到该扩展，最后创建在当前扩展版本中而不再未打包版本中的任何新对象。CREATE EXTENSION用它的FROM old_version选项支持这种情况，这会导致它不为目标版本运行正常的安装脚本，而是运行名为extension--old_version--target_version.sql的更新脚本。选择作为old_version使用的虚假版本名称是扩展作者的工作，不过unpackaged是一种习惯用法。如果你有多个早期版本需要更新到扩展风格，使用多个虚假版本名称来标识它们。

ALTER EXTENSION能够执行更新脚本的序列来实现一个要求的更新。例如，如果只有foo--1.0--1.1.sql和foo--1.1--2.0.sql可用，当前安装了1.0版本并且要求更新到版本2.0，ALTER EXTENSION将依次应用它们。

PostgreSQL并不假定任何有关版本名称的性质：例如，它不知道1.1是否跟在1.0后面。它只是匹配可用的版本名称并且遵照要求应用最少更新脚本的路径进行（一个版本名称实际上可以是不含--或者前导或后缀-的字符串）。

有时提供“降级”脚本也有用，例如foo--1.1--1.0.sql允许把版本1.1相关的改变恢复原状。如果你这样做，要当心降级脚本被意外应用的可能性，因为它会得到一个较短的路径。危险的情况是，有一个跳过几个版本的“快速路径”更新脚本还有一个降级到该快速路径开始点的降级脚本。先应用降级然后再应用快速路径可能比一次升级一个版本需要更少的步骤。如果降级版本删除了任何不可替代的对象，这将会得到意想不到的结果。

要检查意料之外的更新路径，可使用这个命令：

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

这会为指定的扩展显示已知的每一个可区分的版本名对，每一个版本名对还带有一个从源版本到目标版本的更新路径序列，如果没有可用的更新路径则这部份信息为NULL。该路径显示为用--分隔符的文本形式。如果你更喜欢数组格式，可以使用regexp_split_to_array(path,'--')。

38.16.6. 用更新脚本安装扩展

一个已经存在一段时间的扩展可能存在多个版本，作者将需要为它们编写更新脚本。例如，如果你已经发布了扩展foo的版本1.0、1.1和1.2，就应该有更新脚本foo--1.0--1.1.sql和foo--1.1--1.2.sql。在PostgreSQL 10之前，还有必要创建新的脚本文件foo--1.1.sql和foo--1.2.sql，它们直接构建比较新的扩展版本，或者新的版本无法被直接安装，而是通过先安装1.0然后更新。那种方式是无聊的重复性工作，但是现在它是不必要的了，因为CREATE EXTENSION能够自动遵循更新链。例如，如果只有脚本文件foo--1.0.sql、foo--1.0--1.1.sql和foo--1.1--1.2.sql可用，那么安装版本1.2的请求会通过按顺序运行上述三个脚本来实现。这种处理和先安装1.0然后更新到1.2是一样的（和ALTER EXTENSION UPDATE一样，如果有多条路径可用则优先选择最短的）。按这种风格安排扩展 的脚本文件可以减少生产小更新所需的维护工作量。

如果以这种风格维护的扩展中使用了二级（版本相关的）控制文件，记住每个版本都需要一个控制文件，即使它没有单独的安装脚本，因为该控制文件将决定如何执行到这个版本的隐式更新。例如，如果foo--1.0.control指定有requires = 'bar'，但foo的其他控制文件没有这样做，在从1.0更新到另一个版本时，该扩展对bar的依赖将被删除。

38.16.7. 扩展实例

这里是一个只用SQL的扩展的完整例子，一个两个元素的组合类型，它可以在它的槽（命名为“k”和“v”）中存储任何类型的值。非文本值会被自动强制为文本进行存储。

脚本文件pair--1.0.sql看起来像这样：

```
-- 如果脚本是由 psql 而不是 CREATE EXTENSION 执行，则报错
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE OR REPLACE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);
```

```
-- "SET search_path"容易操作，但限定名称更好。
CREATE OR REPLACE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE OR REPLACE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

控制文件pair.control看起来像这样：

```
# pair 扩展
comment = 'A key/value pair data type'
default_version = '1.0'
relocatable = false
```

虽然你几乎不会需要一个 makefile 来安装这两个文件到正确的目录，你还是可以使用一个Makefile：

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

这个 makefile 依赖于PGXS，它在第 38.17 节描述。命令make install将把控制和脚本文件安装到pg_config报告的正确的目录中。

一旦文件被安装，使用CREATE EXTENSION命令就可以把对象载入到任何特定的数据库中。

38.17. 扩展的构建基础设施

如果你正在考虑发布你的PostgreSQL扩展模块，为它们建立一个可移植的构建系统实在是相当困难。因此PostgreSQL安装为扩展提供了一种被称为PGXS构建基础设施，因此简单的扩展模块能够在一个已经安装的服务器上简单地编译。PGXS主要是为了包括 C 代码的扩展而设计，不过它也能用于纯 SQL 的扩展。注意PGXS并不想成为一种用于构建任何与PostgreSQL交互的软件的通用构建系统框架。它只是简单地把简单服务器扩展模块的公共构建规则自动化。对于更复杂的包，你可能需要编写你自己的构建系统。

要把PGXS基础设施用于你的扩展，你必须编写一个简单的 makefile。在这个 makefile 中，你需要设置一些变量并且把它们包括在全局的PGXS makefile 中。这里有一个例子，它构建一个名为isbn_issn的扩展模块，其中包括一个含有 C 代码的共享库、一个扩展控制文件、一个 SQL 脚本、一个包括文件（仅当其他模块可能需要通过调用而不是SQL访问这个扩展的函数时才需要）以及一个文档文件：

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h
```

```
PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

最后三行应该总是相同的。在这个文件的前面部分，你要对变量赋值或者增加自定义的make规则。

设置这三个变量之一来指定要构建什么：

MODULES

要从源文件构建的具有相同词干的共享库对象的列表（不要在这个列表中包括库后缀）

MODULE_big

一个要从多个源文件中构建的共享库（在OBJS中列出对象文件）

PROGRAM

一个要构建的可执行程序（在OBJS中列出对象文件）

还可以设置下列变量：

EXTENSION

扩展名称；你必须为每一个名称提供一个extension.control文件，它将被安装到prefix/share/extension中

MODULEDIR

subdirectory of prefix/share的子目录，DATA 和 DOCS 文件会被安装到其中（如果没有设置，设置了EXTENSION时默认为extension，没有设置EXTENSION时默认为contrib）

DATA

要安装到prefix/share/$MODULEDIR中的随机文件

DATA_built

要安装到prefix/share/$MODULEDIR中的随机文件，它们需要先被构建

DATA_TSEARCH

要安装到prefix/share/tsearch_data中的随机文件

DOCS

要安装到prefix/doc/$MODULEDIR中的随机文件

HEADERS
HEADERS_built

要（构建并且）安装在prefix/include/server/$MODULEDIR/$MODULE_big下面的文件。

和DATA_built不同，HEADERS_built中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA_CLEAN或者增加自己的规则来做这件事。

HEADERS_$MODULE
HEADERS_built_$MODULE

要安装（如果指定了构建则在构建之后安装）在prefix/include/server/$MODULEDIR/$MODULE之下的文件，这里$MODULE必须是一个在MODULES or MODULE_big中用到的模块名。

和DATA_built不同，HEADERS_built_$MODULE中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA_CLEAN或者增加自己的规则来做这件事。

可以为同一个模块同时使用这两个变量或者两者的任意组合，除非你在MODULES列表中有两个模块名称仅有前缀built_上的区别，因为那样会导致歧义。在那种情况下（还好不太可能），应该仅使用HEADERS_built_$MODULE变量。

SCRIPTS

要安装到prefix/bin中的脚本文件（非二进制）

SCRIPTS_built

要安装到prefix/bin中的脚本文件（非二进制），它们需要先被构建

REGRESS

回归测试案例（不带后缀）的列表，见下文

REGRESS_OPTS

要传递给pg_regress的附加开关

NO_INSTALLCHECK

不定义installcheck目标，如果测试要求特殊的配置就会很有用，或者不使用pg_regress

EXTRA_CLEAN

要在make clean中移除的额外文件

PG_CPPFLAGS

将被加到CPPFLAGS前面

PG_CFLAGS

将被加到CFLAGS后面

PG_CXXFLAGS

将被加到CXXFLAGS后面

PG_LDFLAGS

将被加到LDFLAGS前面

PG_LIBS

将被加到PROGRAM链接行

SHLIB_LINK

将被加到MODULE_big链接行

PG_CONFIG

要在其中构建的PostgreSQL安装的pg_config程序的路径（通常只用在你的PATH中的第一个pg_config）

把这个 makefile 作为Makefile放在保存你扩展的目录中。然后你可以执行make进行编译，并且接着make install来安装你的模块。默认情况下，该模块会为在你的PATH中找到的

第一个pg_config程序所对应的PostgreSQL安装编译和安装。你可以通过在 makefile 中或者make命令行中设置PG_CONFIG指向另一个pg_config程序来使用一个不同的安装。

如果你想保持编译目录独立，你也可以在你的扩展所属的源代码树之外的目录中运行make。 这个过程也被称为一个 VPATH 编译。下面是做法：

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

此外，你可以以对核心代码所作的方式一样为 VPATH 设置一个目录。一种方式是使用核心脚本 config/prep_buildtree。一旦这样做，你可以这样设置 make变量VPATH：

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

这个过程可以在很多种目录布局下工作。

列举在REGRESS变量中的脚本会被用来对你的扩展进行回归测试，回归测试可以在做完make install之后用make installcheck调用。要让这能够工作，你必须已经有一个运行着的PostgreSQL服务器。列举在REGRESS中的脚本文件必须在你扩展目录的名为sql/的子目录中出现。这些文件必须带有扩展.sql，但扩展不能被包括在 makefile 的REGRESS列表中。对每一个测试还应该在名为expected/的子目录中有一个包含预期输出的文件，它具有和脚本文件相同的词干并带有扩展.out。make installcheck会用psql执行每一个测试脚本，并且将得到结果输出与相应的预期输出比较。任何区别都将以diff -c格式写入到文件regression.diffs中。注意尝试运行一个不带预期文件的测试将被报告为“故障”，因此确保你拥有所有的预期文件。

> **提示**
>
> 创建预期文件最简单的方法是创建空文件，然后做一次测试运行（这当然将报告区别）。检查在results/目录中找到的实际结果文件，如果它们符合你的预期则把它们复制到expected/中。

第 39 章 触发器

本章提供了编写触发器函数的一般信息。触发器函数可以使用大部分可用过程性语言，包括PL/pgSQL（第 43 章、 PL/Tcl（第 44 章、 PL/Perl（第 45 章和 PL/Python（第 46 章 阅读本章后，你应该查阅你喜欢的过程性语言来找到用其编写触发器的相关细节。

也可以使用C编写一个触发器函数，尽管大部分人觉得使用一种过程性语言更简单。当前不能用纯SQL函数语言来编写触发器函数。

39.1. 触发器行为概述

一个触发器声明了当执行一种特定类型的操作时数据库应该自动执行一个特殊的函数。触发器可以被附加到表（分区的或者不分区的）、视图和外部表。

在表和外部表上，触发器可以被定义为在 INSERT、UPDATE或 DELETE操作之前或之后被执行， 可以为每个SQL语句被执行一次或者为每个修改的行 被执行一次。UPDATE 触发器可以进一步地设置为只针对UPDATE 语句的SET子句的特定列出发。触发器也可以被 TRUNCATE语句触发。如果一个触发器事件发生， 触发器函数会在适当的事件被调用来处理该事件。

在视图上，触发器可以被定义来取代INSERT、UPDATE或 DELETE操作的执行。这种INSTEAD OF触发器对视图中需要被修改的每一行触发一次。触发器函数的职责是对视图的底层基本表执行必要的修改，并且在合适的时候返回被修改的行以便显示在视图中。视图上的触发器也可以被定义为对每个SQL语句执行一次，在INSERT\UPDATE或DELETE操作之前或之后。不过，只有在该视图上还有一个INSTEAD OF触发器时，上述那些触发器才会被触发。否则，以该视图为目标的任何语句都必须被重写成一个影响其底层基表的语句，然后附着在那些基表上的触发器将会被引发。

触发器函数必须在触发器本身被创建之前被定义好。触发器函数必须被定义成一个没有参数的函数，并且返回类型为trigger（触发器函数通过一个特殊传递的TriggerData结构作为其输入，而不是以普通函数参数的形式）。

一旦一个合适的触发器函数被创建，就可以使用CREATE TRIGGER建立触发器。同一个触发器函数可以被用于多个触发器。

PostgreSQL同时提供每行的触发器和每语句的触发器。对于一个每行的触发器，对于触发触发器的语句所修改的每一行都会调用一次触发器函数。相反，一个每语句的触发器对于其触发语句只被调用一次，而不管该语句影响了多少行。特别地，一个不影响任何行的语句仍然会导致任何可用每语句的触发器的执行。这两类触发器有时也分别被称作行级触发器和语句级触发器。TRUNCATE上的触发器只能被定义在语句级。

触发器也可以根据它们是否在操作之前、之后触发，或者被触发来取代操作来分类。它们分别指BEFORE触发器、AFTER 触发器以及INSTEAD OF触发器。语句级BEFORE触发器在语句开始做任何事情之前被触发，而语句级AFTER触发器则在语句做完所有事情之后被触发。这些触发器类型可以被定义在表、视图或外部表上。行级BEFORE触发器在每一个行被操作之前被触发，而行级AFTER触发器在语句结束之后被触发（但在任何语句级AFTER触发器之前）。这些触发器类型只能被定义在非分区表和外部表上，但不能定义在视图上。INSTEAD OF触发器只能被定义在视图上，并且只能定义在行级，当视图中的每一行被标识为需要被操作时，它们会立即触发。

一个以继承或者分区层次中父表为目标的语句不会导致受影响的子表的语句级触发器被引发，只有父表的语句级触发器会被引发。不过，受影响的子表的行级触发器将被引发。

如果一个INSERT包含ON CONFLICT DO UPDATE子句并且引用了EXCLUDED列，有可能所有行级 BEFORE INSERT触发器和所有行级 BEFORE UPDATE触发器的效果可能会以一种对于 被更新行最终状态透明的方式被应用。不过，对于要执行的两种集合的行级 BEFORE触发器都不需要有EXCLUDED列引用。当同时有行级 BEFORE INSERT和 BEFORE UPDATE触发器影响被插

入/ 更新的行时（如果在两者不幂等时修改或多或少地等价，这仍可能是有问题的）， 应该考虑可能出现的意料之外的结果。注意在指定了 ON CONFLICT DO UPDATE时，不管有没有行被 UPDATE影响（并且不管是否采用了其他 UPDATE路径），语句级 UPDATE都将被执行。一个带有 ON CONFLICT DO UPDATE子句的INSERT 将首先执行语句级BEFORE INSERT， 然后执行语句级BEFORE UPDATE触发器， 接着是语句级AFTER UPDATE触发器， 最后是语句级AFTER INSERT触发器。

如果一个分区表上的UPDATE导致一行移动到另一个分区，它将被从原始分区DELETE掉然后再INSERT到新分区中。在这种情况下，原始分区上所有的行级BEFORE UPDATE触发器和所有行级BEFORE DELETE触发器会被引发。然后目标分区上所有的行级BEFORE INSERT触发器会被引发。当所有这些触发器都影响被移动的行时，应该对令人惊讶的结果有心理准备。至于AFTER ROW触发器，AFTER DELETE和AFTER INSERT触发器会被应用，但AFTER UPDATE触发器不会被应用，因为UPDATE已经被转换成了一个DELETE和一个INSERT。对于语句级触发器，即便发生行移动，DELETE和INSERT触发器也都不会被引发，只有UPDATE语句中用到的目标表上的UPDATE触发器将被引发。

被语句级触发器调用的触发器函数应该总是返回NULL。根据行级触发器的选择，被其调用的触发器函数可以返回一个表行（类型HeapTuple的一个值）给执行器。在一个操作前触发的行级触发器有下列选择：

- 它可以返回NULL来跳过对当前行的操作。这指示执行器不要执行调用触发器的行级操作（对一个特定表行的插入、修改或删除）。
- 仅对行级INSERT和UPDATE触发器来说，被返回的行称为将要被插入的行或者替代将被更新的行。这允许触发器函数修改将要被插入或更新的行。

一个无意导致任何这些行为的行级BEFORE触发器必须小心地它的结果，使之和被传入的行一样（即，INSERT和UPDATE触发器的NEW行，DELETE触发器的OLD行）。

一个行级INSTEAD OF触发器可以返回NULL来指示它没有修改任何来自于视图底层基表的数据，也可以返回被传入的视图行（INSERT和UPDATE操作的NEW行，或者DELETE操作的OLD行）。一个非空返回值被用于标志触发器在视图中执行了必须的数据修改。这将会导致被命令修改的行计数被增加。对于INSERT和UPDATE操作，触发器可能会在返回NEW行之前对其进行修改。这将会改变INSERT RETURNING或UPDATE RETURNING返回的数据，并在视图无法正确地显示提供给它的相同数据时有用。

对于在一个操作之后触发的行级触发器，返回值会被忽略，因此它们可以返回NULL。

如果为同一个关系上的同一事件定义了超过一个触发器，它们将按照其名称的字母表顺序被触发。在BEFORE和INSTEAD OF触发器的情况下，每一个触发器返回的可能被修改的行将成为下一个触发器的输入。如果任何一个BEFORE或INSTEAD OF触发器返回NULL，该操作将在该行上被禁用并且对于该行不会触发后续的触发器。

一个触发器定义也能指定一个布尔的WHEN条件，它将被测试来看该触发器是否应该被触发。在行级触发器中，WHEN条件可以检查该行的旧列值和/或新列值（语句级触发器也能有WHEN条件，但是该特性对它们不太有用）。在一个BEFORE触发器中，WHEN条件只是在该函数被或者将被执行前计算，因此使用WHEN条件与在该触发器函数的开始测试相同的条件没有本质区别。不过，在一个AFTER触发器中，WHEN条件只是在行更新发生之后被计算，并且它决定在语句的末尾一个事件是否被排队来触发该触发器。因此当一个AFTER触发器的WHEN不返回真时，在语句的末尾没有必要将一个事件进行排队，也没有必要重新取出该行。如果触发器只对少数行触发，这可以使得修改很多行的语句明显加快。INSTEAD OF触发器不支持WHEN条件。

通常，行级BEFORE被用来检查或修改即将被插入或更新的数据。例如，一个BEFORE触发器可以被用来把当前时间插入到一个timestamp列中，或者检查该行的两个元素之间是否一致。行级AFTER触发器大多数被用来将更新传播到其他表，或者针对其他表进行一致性检查。进行这种工作分工的原因是，一个AFTER触发器可以肯定它看到的是该行的最终值，而一个BEFORE触发器则不能，因为还可能有其他BEFORE触发器在它之后触发。如果你不知道让一个触发器是BEFORE或AFTER，则BEFORE形式更加有效，因为关于该操作的信息直到语句的末尾都不需要被保存。

如果一个触发器函数执行 SQL 命令，则这些命令可能会再次引发触发器。这就是所谓的级联触发器。对于级联的层数没有直接的限制。级联有可能会导致对同一个触发器的递归调用。例如，一个INSERT触发器可能执行一个向同一个表插入一个额外行的命令，这就导致该INSERT触发器被再次引发。所以在这种情形下，触发器程序员应该负责避免无限递归。

在定义一个触发器时，可以为它指定参数。在触发器定义中包括参数的目的是允许具有相似需求的不同触发器调用同一个函数。例如，可能有一个一般性的触发器函数，它需要两个列名作为参数，一个放当前用户而另一个放当前时间戳。在正确编写的情况下，这个触发器函数应该独立于它所触发的表。因此同一个函数可以被用于具有适当列的任意表上的INSERT事件，这样做的用途之一是可以自动追踪一个交易表中记录的创建。如果被定义成一个UPDATE触发器，它也可以被用来追踪最新的更新事件。

每一种支持触发器的编程语言都有自己的方法来让触发器输入数据对触发器函数可用。这种输入数据包括触发器事件的类型（如INSERT或UPDATE）以及被列在CREATE TRIGGER中的任何参数。对于一个行级触发器，输入数据还包括用于INSERT和UPDATE触发器的NEW行，和/或用于UPDATE和DELETE触发器的OLD行。语句级触发器当前没有任何方法检查被语句修改的单个行。

默认情况下，语句级触发器没有办法检查该语句修改的行。但是AFTER STATEMENT触发器可以请求创建传递表，这样可以让受影响的行集合对该触发器可用。AFTER ROW触发器也可以请求传递表，这样它们可以看到表中的整个变化，同时也能看到当前引发它们的个体行中的变化。检查传递表的方法仍是取决于使用的编程语言，但是通常的方法让传递表变得像触发器函数内部发出的SQL命令能够访问的只读临时表一样。

39.2. 数据改变的可见性

如果你在你的触发器函数中执行 SQL 命令，并且这些命令会访问触发器所在的表，那么你需要注意数据可见性规则。因为这些规则决定了这些 SQL 命令是否将能看见引发触发器的数据改变。简单地：

- 语句级触发器遵循简单的可见性规则：一个语句所作的改变对于语句级 BEFORE触发器都不可见，而所有修改对于语句级 AFTER触发器都是可见的。
- 导致触发器被引发的数据更改（插入、更新或删除）自然对于在一个行级BEFORE触发器中执行的 SQL 命令不可见，因为它还没有发生。
- 但是，在一个行级BEFORE触发器中执行的 SQL 命令将会看见之前在同一个外层命令中所作的数据更改的效果。这里需要小心，因为这些更改时间的顺序通常是不可预测的，一个影响多行的 SQL 命令可能以任何顺序访问这些行。
- 类似地，一个行级INSTEAD OF触发器将会看见之前在同一个外层命令中INSTEAD OF触发器引发所作的数据更改。
- 当一个行级AFTER触发器被引发时，所有由外层命令所作的数据更改已经完成，并且对于该被调用的触发器函数是可见的。

如果你的触发器函数使用任何一种标准过程语言编写的，那么只有在该函数被声明为VOLATILE时上述陈述才适用。被声明为STABLE或IMMUTABLE的函数在任何情况下将不能看到由调用命令所作出的更改。

有关数据可见性规则的更多信息可见第 47.5 节第 39.4 节的例子包含了对这些规则的示范。

39.3. 用 C 编写触发器函数

这一节描述了一个触发器函数的接口的低层细节。只有用 C 编写触发器函数时才需要这些信息。如果你使用一种更高层的语言，那么这些细节就不需要你来处理。在大部分情况下，你应该优先考虑使用一种过程语言。每一种过程语言的文档阐述了如何使用那种语言编写一个触发器。

触发器函数必须使用“版本 1”函数管理器接口。

当一个函数被触发器管理器调用时，不会给它传递任何常规的参数，但是会有一个“context”指针传递给它，该指针指向一个TriggerData结构。C 函数可以通过执行一个宏来检查它们是否是从触发器管理器被调用：

```
CALLED_AS_TRIGGER(fcinfo)
```

它会展开成为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

如果这返回真，那么将fcinfo->context造型成类型TriggerData *并且利用所指向的TriggerData结构就是安全的。该函数不能修改该TriggerData结构或者它指向的任何数据。

struct TriggerData被定义在commands/trigger.h中：

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    Buffer           tg_trigtuplebuf;
    Buffer           tg_newtuplebuf;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
} TriggerData;
```

其中的成员被定义如下：

type

总是T_TriggerData.

tg_event

描述该函数是为什么事件被调用的。你可以使用下列宏来检查tg_event：

TRIGGER_FIRED_BEFORE(tg_event)

如果该触发器在操作前被引发则返回真。

TRIGGER_FIRED_AFTER(tg_event)

如果该触发器在操作后被引发则返回真。

TRIGGER_FIRED_INSTEAD(tg_event)

如果该触发器被引发替代操作则返回真。

TRIGGER_FIRED_FOR_ROW(tg_event)

如果该触发器为一个行级事件而引发则返回真。

TRIGGER_FIRED_FOR_STATEMENT(tg_event)

如果该触发器为一个语句级事件而引发则返回真。

TRIGGER_FIRED_BY_INSERT(tg_event)

如果该触发器由一个INSERT命令引发则返回真。

TRIGGER_FIRED_BY_UPDATE(tg_event)

如果该触发器由一个UPDATE命令引发则返回真。

TRIGGER_FIRED_BY_DELETE(tg_event)

如果该触发器由一个DELETE命令引发则返回真。

TRIGGER_FIRED_BY_TRUNCATE(tg_event)

如果该触发器由一个TRUNCATE命令引发则返回真。

tg_relation

一个结构指针，该结构描述该触发器为其引发的关系。关于这个结构的细节请参考utils/rel.h。最有趣的东西是tg_relation->rd_att（该关系元组的描述符）和tg_relation->rd_rel->relname（关系名称，该类型不是char*而是NameData。如果你需要该名称的一个拷贝，可使用SPI_getrelname(tg_relation)来得到一个char*）。

tg_trigtuple

一个该触发器为其引发的行的指针。这是被插入、更新或删除的行。如果这个触发器是为一个INSERT或DELETE而引发，在你不想把该行替换成另一行（在INSERT的情况中）或不想跳过该操作时你应该从该函数中返回它。 对于外部表上的触发器，此中的系统列值未被指定。

tg_newtuple

如果该触发器为一个UPDATE而引发，则是一个指向该行新版本的指针。如果是为一个INSERT或DELETE而引发，则是NULL。如果事件是一个UPDATE并且你并不想用一个不同的行替换这个行或者不想跳过该操作时，你必须从函数中返回它。对于外部表上的触发器，此中的系统列值未被指定。

tg_trigger

一个指向类型为Trigger的结构的指针，定义在utils/reltrigger.h中：

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

其中tgname是该触发器的名称，tgnargs是tgargs中参数的数量，而tgargs是一个指向CREATE TRIGGER语句中指定的参数的指针数组。其他成员只用于内部用途。

tg_trigtuplebuf

包含tg_trigtuple的缓冲区。如果没有那个元组或者它没有被存储在一个磁盘缓冲区中，则为InvalidBuffer。

tg_newtuplebuf

包含tg_newtuple的缓冲区。如果没有那个元组或者它没有被存储在一个磁盘缓冲区中，则为InvalidBuffer。

tg_oldtable

一个指向Tuplestorestate类型的结构的指针，该结构包含格式由tg_relation指定的零行或者多行。如果没有OLD TABLE传递关系，则为NULL指针。

tg_newtable

一个指向Tuplestorestate类型的结构的指针，该结构包含格式由tg_relation指定的零行或者多行。如果没有NEW TABLE传递关系，则为NULL指针。

为了允许通过SPI发出的查询引用传递表，请参考SPI_register_trigger_data。

一个触发器函数必须返回一个HeapTuple指针或一个NULL指针（不是一个 SQL 空值，也就是不会设置isNull为真）。如果你不希望修改正在被操作的行，要小心地根据情况返回tg_trigtuple或tg_newtuple。

39.4. 一个完整的触发器实例

这里有一个用 C 编写的触发器函数的非常简单的例子（用过程语言编写的触发器的例子可以在过程语言的文档中找到）。

如果该命令试图向列x中插入一个空值，函数trigf报告表ttest中的行数并且跳过实际的操作（这样该触发器会作为一个非空约束但不会中止事务）。

首先，表定义：

```
CREATE TABLE ttest (
    x integer
);
```

这是该触发器函数的源代码：

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
```

```
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

在你编译了该源代码（见第 38.10.5 节之后，声明该函数和触发器：

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

现在你可以测试该触发器的操作：

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                        ^^^^^^^^

                             remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                        ^^^^^^

                             remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
```

```
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                        ^^^^^^

                             remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

在src/test/regress/regress.c和spi中有更多复杂的例子。

第 40 章 事件触发器

为了对第 39 章讨论的触发器机制加以补充， PostgreSQL也提供了事件触发器。和常规触发器（附着在 一个表上并且只捕捉 DML 事件）不同，事件触发器对一个特定数据库来说是全局 的，并且可以捕捉 DDL 事件。

和常规触发器相似，可以用任何包括了事件触发器支持的过程语言或者 C 编写 事件触发器，但是不能用纯 SQL 编写。

40.1. 事件触发器行为总览

只要与一个事件触发器相关的事件在事件触发器所在的数据库中发生， 该事件触发器就会被引发。当前支持的事件是 ddl_command_start、ddl_command_end、table_rewrite和sql_drop。未来的发行版 中可能会增加对更多事件的支持。

ddl_command_start事件就在CREATE、 ALTER、DROP、SECURITY LABEL、 COMMENT、GRANT或者REVOKE 命令的执行之前发生。在事件触发器引发前不会做受影响对象是否存在的检查。不过，一个例外是，这个事件不会为目标是共享对象 — 数据库、角色 以及表空间 — 的 DDL 命令发生，也不会为目标是事件触发器的 DDL 命令发生。事件触发器机制不支持这些对象类型。 ddl_command_start也会在SELECT INTO 命令的执行之前发生，因为这等价于 CREATE TABLE AS。

ddl_command_end事件就在同一组命令的执行之后发生。为了 得到发生的DDL操作的更多细节，可以从 ddl_command_end事件触发器代码中使用集合返回函数 pg_event_trigger_ddl_commands()（见 第 9.28 节。注意该触发器是在那些动作 已经发生之后（但是在事务提交前）引发，并且因此系统目录会被读作已更改。

sql_drop事件为任何删除数据库对象的操作在 ddl_command_end事件触发器之前发生。要列出已经被删除的 对象，可以从sql_drop事件触发器代码中使用集合返回函数 pg_event_trigger_dropped_objects()见 第 9.28 节。注意该触发器是在对象已经 从系统目录删除以后执行，因此不能再查看它们。

table_rewrite事件在表被命令ALTER TABLE和 ALTER TYPE的某些动作重写之前发生。虽然其他控制语句（例如 CLUSTER和VACUUM）也可以用来重 写表，但是它们不会触发table_rewrite事件。

不能在一个中止的事务中执行事件触发器（其他函数也一样）。因此，如果一个 DDL 命令出现错误失败，将不会执行任何相关的 ddl_command_end触发器。反过来，如果一个 ddl_command_start触发器出现错误失败，将不会引发进一步的 事件触发器，并且不会尝试执行该命令本身。类似地，如果一个 ddl_command_end触发器出现错误失败，DDL 命令的效果将被 回滚，就像其他包含事务中止的情况中那样。

第 40.2 节有事件触发器机制所支持的完整 命令列表。

事件触发器通过命令CREATE EVENT TRIGGER创建。为了 创建一个事件触发器，你必须首先创建一个有特殊返回类型 event_trigger的函数。这个函数不一定需要返回一个值， 该返回类型仅仅是作为一种信号表示该函数要被作为一个事件触发器调用。

如果对于一个特定的事件定义了多于一个事件触发器，它们将按照触发器名称 的字母表顺序被引发。

一个触发器定义也可以指定一个WHEN条件，这样事件触 发器（例如ddl_command_start触发器）就可以只对用户 希望介入的特定命令触发。这类触发器的通常用法是用于限制用户可能执行的 DDL 操作的范围。

40.2. 事件触发器触发矩阵

表 40.1列出了所有命令的 事件触发器支持情况。

表 40.1. 支持事件触发器的命令标签

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注解
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注解
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
COMMENT	X	X	-	-	只对本地对象
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注解
CREATE STATISTICS	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注解
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	只对本地对象
IMPORT FOREIGN SCHEMA	X	X	-	-	
REVOKE	X	X	-	-	只对本地对象
SECURITY LABEL	X	X	-	-	只对本地对象
SELECT INTO	X	X	-	-	

40.3. 用 C 编写事件触发器函数

这一节描述了事件触发器函数接口的低层细节。只有在用 C 编写事件 触发器函数时才需要用到这里的信息。如果使用更高层的语言，那么 这些细节已经被处理好了。在大部分情况下都应该优先考虑使用过程 语言来编写你的事件触发器。每一种过程语言的文档都解释了如何用 它编写事件触发器。

事件触发器函数必须使用“版本 1”的函数管理器接口。

当一个函数被事件触发器管理器调用时，向它传递的并不是普通参数，而是一个指向EventTriggerData结构的 “context”指针。C 函数可以通过执行以下宏来检查它是否 被事件触发器管理器调用：

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

这个宏会被扩展为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

如果这个宏返回真，那么就可以安全地把 fcinfo->context造型为类型EventTriggerData *并且使用所指向的EventTriggerData结构。 函数不能修改 EventTriggerData结构以及它指向的任何内容。

struct EventTriggerData在 commands/event_trigger.h中定义：

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* 事件名称 */
    Node       *parsetree;  /* 解析树 */
    const char *tag;        /* 命令标签 */
} EventTriggerData;
```

其中的成员定义如下：

type

总是T_EventTriggerData。

event

描述要为其调用这个函数的事件，可以是 "ddl_command_start"、"ddl_command_end"、"sql_drop"、"table_rewrite"之一。 这些事件的含义请见第 40.1 节

parsetree

该命令的解析树的指针。其细节可以参考 PostgreSQL 的源代码。 解析树结构可能会在未经通知的情况下改变。

tag

与事件触发器的事件相关联的命令标签，例如 "CREATE FUNCTION"。

一个事件触发器函数必须返回一个NULL指针（ 不是一个 SQL 空值，也就是不要把 isNull设置为真）。

40.4. 一个完整的事件触发器例子

这里是一个用 C 编写的事件触发器函数的简单例子（用过程语言编写的触发器 例子可以在过程语言的文档中找到）。

函数noddl在每一次被调用时抛出一个异常。事件触发器定义把该函数和ddl_command_start事件关联在了一起。其效果就是所有 DDL 命令（除第 40.1 节提到的例外）都 被阻止运行。

这是该触发器函数的源代码：

```
#include "postgres.h"
#include "commands/event_trigger.h"


PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
        (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

在编译了源代码（见第 38.10.5 节后， 声明函数和触发器：

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE FUNCTION noddl();
```

现在你可以测试该触发器的操作：

```
=# \dy
                        List of event triggers
 Name  |       Event       | Owner | Enabled | Function | Tags
-------+-------------------+-------+---------+----------+------
 noddl | ddl_command_start | dim   | enabled | noddl    |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

在这种情况下，为了在需要时能运行某些 DDL 命令，你必须删除该事件触发器 或者禁用它。只在一个事务期间禁用该触发器会比较方便：

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

（回忆一下，事件触发器本身上的 DDL 命令不受事件触发器影响）。

40.5. 一个表重写事件触发器例子

得益于table_rewrite事件的存在，我们可以实现一种只允许在 维护窗口中重写的表重写策略。

这里是实现这种策略的例子。

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- 实现本地表重写策略：
---   public.foo 不允许重写，其他表只允许在 1am 和 6am 之间重写，
---   且前提是它们拥有不超过 100 块
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than %
 pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE FUNCTION no_rewrite();
```

第 41 章 规则系统

本章讨论PostgreSQL中的规则系统。产生规则系统的概念很简单，但是在实际使用的时候会碰到很多细节问题。

某些其它数据库系统定义活动的数据库规则，通常是存储过程和触发器。在PostgreSQL中，这些东西可以通过函数和触发器来实现。

规则系统（更准确地说是查询重写规则系统）与存储过程和触发器完全不同。它把查询修改为需要考虑规则，并且然后把修改过的查询传递给查询规划器进行规划和执行。它非常强大，并且可以被用于许多东西如查询语言过程、视图和版本。这个规则系统的理论基础和能力也在[ston90b]和[ong90]中讨论。

41.1. 查询树

要了解规则系统是如何工作的，必须要知道它什么时候被调用以及它的输入和结果是什么。

规则系统位于解析器和规划器之间。它采用解析器的输出（即一个查询树）和用户定义的重写规则（也是查询树，不过带有一些额外信息），并且常见零个或者更多个查询树作为结果。因此它的输入和输出总是那些规划器自身就能产生的东西，并且因此它看到的任何东西都可以被表示成一个SQL语句。

那么什么是一个查询树？它是一个SQL语句的一种内部表示，其中用于创建它的每一个单独的部分都被独立存储。如果你设置了配置参数debug_print_parse、debug_print_rewritten或debug_print_plan，这些查询树可以被显示在服务器日志中。规则动作也被做为查询树存储在系统目录pg_rewrite中。它们没有被格式化为日志输出的形式，但是它们包含完全相同的信息。

阅读一棵未加工的查询树需要要一些经验。但是由于查询树的SQL表示形式足以用来理解规则系统，本章将不会教授如何阅读查询树。

在阅读本章中查询树的SQL表现形式时，读者需要能够知道语句被分解成了哪些部分并且能在查询树结构中标识它们。一棵查询树的部分有：

命令类型

这是一个简单的值来说明是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了该查询树。

范围表

范围表是被使用在该查询中的关系的列表。在一个SELECT语句中，范围表是在关键词FROM后面给出的关系。

每一个范围表项标识一个表或视图，并且说明在该查询的其他部分要以哪个名称调用它。在查询树中，范围表项被使用编号而不是名称来引用，因此在一个SQL语句中出现重复的名字也没有关系。在规则的范围表被合并以后可能会发生这种情况。本章中的例子将不会有这种情况。

结果关系

这是一个指向范围表的索引，它标识了该查询的结果应该去哪个关系。

SELECT查询没有结果关系（特殊情况SELECT INTO几乎等于CREATE TABLE后面跟上INSERT ... SELECT，并且不在这里单独讨论）。

对于INSERT、UPDATE和DELETE命令，结果关系是修改要进行的表（或视图！）。

目标列表

目标列表是一个表达式的列表，它定义了查询的结果。在一个SELECT的情况下，这些表达式会构建出该查询最终的输出。它们对应于关键字SELECT和FROM之间的表达式（*是一个关系所有列名的缩写。解析器会把它扩展成独立的列，因此规则系统永远见不到它）。

DELETE命令不需要一个目标列表，因为它们不产生任何结果。相反，规划器会向空的目标列表中加入一个特殊的CTID项来允许执行器找到要被删除的行（当结果关系是一个普通表时才加入CTID。如果结果关系是一个视图，则会被规则系统加入一个整行变量，如第 41.2.4 所述）。

对于INSERT命令，目标列表描述了将要进入到结果关系中的新行。它由VALUES子句中的表达式或来自INSERT ... SELECT中SELECT子句的表达式构成。重写处理的第一步会为那些没有被原始命令赋值但有默认值的列增加目标列项。任何剩余的列（既没有给定值也没有默认值）将被规划器用一个常量空值表达式填充。

对于UPDATE命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含来自命令的SET column = expression部分的表达式。规划器将处理缺失的列，做法是为它们插入表达式，这种表达式会把旧行的值复制到新行。正如DELETE一样，会增加一个CTID或整行变量，这样执行器能够标识要被更新的旧行。

目标列表中的每一个项所包含的表达式可以是一个常量值、一个指向范围表中关系的列的变量、一个参数或一个由函数调用、常量、变量、操作符等构成的表达式树。

条件

查询的条件是一个表达式，它很像包含在目标列表项中的表达式。这个表达式的结果值是一个布尔值，它说明对最终结果行的操作（INSERT、UPDATE、DELETE或SELECT）是否应该被执行。它对应于一个SQL语句的WHERE子句。

连接树

查询的连接树展示了FROM子句的结构。对于一个SELECT ... FROM a, b, c这样的简单查询，连接树就是FROM项的一个列表，因为我们被允许以任何顺序连接它们。但是当JOIN表达式（特别是外连接）被使用时，我们必须按照连接显示的顺序来连接。在这种情况下，连接树展示了JOIN表达式的结构。与特定JOIN子句（来自ON或USING）相关的限制被存储为附加到那些连接树节点的条件表达式。我们发现把顶层WHERE表达式存储为附加到顶层连接树项的一个条件也很方便。这样实际上连接树表达了一个SELECT的FROM和WHERE子句。

其他

查询树的其他部分（如ORDER BY子句）在这里并不受到关注。规则系统在应用规则时会替换这里的某些项，但是这些与规则系统的基础没有什么关系。

41.2. 视图和规则系统

PostgreSQL中的视图是通过规则系统来实现的。事实上，下面的命令

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面两个命令相比没有不同：

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

因为这就是CREATE VIEW命令在内部所作的。 这样做有一些副作用。其中之一就是在PostgreSQL系统目录中的视图信息与表的信息完全一样。所以对于解析器来说，表和视图之间完全没有区别。它们是同样的事物：关系。

41.2.1. SELECT规则如何工作

规则ON SELECT被应用于所有查询作为最后一步，即使给出的是一条INSERT、UPDATE或DELETE命令。而且它们与其他命令类型上的规则有着不同的语义，它们会就地修改查询树而不是创建一个新的查询树。因此我们首先描述SELECT规则。

目前，一个ON SELECT规则中只能有一个动作， 而且它必须是一个无条件的INSTEAD的SELECT动作。 这个限制是为了令规则足够安全，以便普通用户也可以打开它们，并且它限制ON SELECT规则使之行为类似视图。

本章的例子是两个连接视图，它们做一些运算并且某些更多视图会轮流使用它们。最前面的两个视图之一后面将利用对INSERT、UPDATE和DELETE操作增加规则的方法被自定义，这样最终结果将是一个视图，它表现得像一个具有魔力的真正的表。这个例子不适合于作为简单易懂的例子，它可能会让本章更难懂。但是用一个覆盖所有关键点的例子来一步一步讨论要比举很多例子搞乱思维好。

例如，我们需要一个小巧的min函数用于返回两个整数值中较小的那个。我们这样创建它：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

在前两个规则系统描述中我们需要真实表是：

```
CREATE TABLE shoe_data (
    shoename   text,          -- 主键
    sh_avail   integer,       -- 可用的双数
    slcolor    text,          -- 首选的鞋带颜色
    slminlen   real,          -- 最小鞋带长度
    slmaxlen   real,          -- 最大鞋带长度
    slunit     text           -- 长度单位
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- 主键
    sl_avail   integer,       -- 可用的双数
    sl_color   text,          -- 鞋带颜色
    sl_len     real,          -- 鞋带长度
    sl_unit    text           -- 长度单位
);

CREATE TABLE unit (
    un_name    text,          -- 主键
    un_fact    real           -- 转换到厘米的参数
);
```

如你所见，它们表示鞋店的数据。

视图被创建为：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
```

```
            sh.slcolor,
            sh.slminlen,
            sh.slminlen * un.un_fact AS slminlen_cm,
            sh.slmaxlen,
            sh.slmaxlen * un.un_fact AS slmaxlen_cm,
            sh.slunit
       FROM shoe_data sh, unit un
      WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
       FROM shoelace_data s, unit u
      WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
       FROM shoe rsh, shoelace rsl
      WHERE rsl.sl_color = rsh.slcolor
        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建shoelace视图的CREATE VIEW命令（也是最简单的一个）将创建一个shoelace关系和一个pg_rewrite项， 这个pg_rewrite项说明有一个重写规则，只要一个查询的范围表中引用了关系shoelace，就必须应用它。该规则没有规则条件（稍后和非SELECT规则一起讨论，因为目前的SELECT规则不能有规则条件）并且它是INSTEAD规则。要注意规则条件与查询条件不一样。我们的规则的动作有一个查询条件。该规则的动作是一个查询树，这个查询是视图创建命令中的SELECT语句的一个拷贝。

> **注意**
>
> 你在pg_rewrite项中看到的两个额外的用于NEW和OLD的范围表项不是SELECT规则感兴趣的东西。

现在我们填充unit、shoe_data和shoelace_data，并且在视图上运行一个简单的查询：

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
```

```
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

这是你可以在我们的视图上做的最简单的SELECT，所以我们用这次机会来解释视图规则的基本要素。SELECT * FROM shoelace会被解析器解释并生成下面的查询树：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

然后这将被交给规则系统。规则系统遍历范围表，检查有没有可用于任何关系的规则。在为shoelace（到目前为止的唯一一个）处理范围表时， 它会发现查询树里有_RETURN规则：

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

要扩展该视图，重写器简单地创建一个子查询范围表项，它包含规则的动作的查询树，然后用这个范围表记录取代原来引用视图的那个。作为结果的重写后的查询树几乎与你键入的那个一样：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

不过有一个区别：子查询的范围表有两个额外的项shoelace old和shoelace new。这些项并不直接参与到查询中，因为它们没有被子查询的连接树或者目标列表引用。重写器用它们存

储最初出现在引用视图的范围表项中表达的访问权限检查信息。以这种方式，执行器仍然会检查该用户是否有访问视图的正确权限，尽管在重写后的查询中没有对视图的直接使用。

这是被应用的第一个规则。规则系统将继续检查顶层查询里剩下的范围表项（本例中没有了），并且它将递归的检查增加的子查询中的范围表项，看看其中有没有引用视图的（不过这样不会扩展old或new — 否则我们会得到无限递归！）。在这个例子中，没有用于shoelace_data或unit的重写规则，所以重写结束并且上面得到的就是给规划器的最终结果。

现在我们想写一个查询，它找出目前在店里哪些鞋子有匹配的（颜色和长度）鞋带并且完全匹配的鞋带双数大于等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

这词解析器的输出是查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

第一个被应用的规则将是用于shoe_ready的规则并且它会导致查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

相似地，用于shoe和shoelace的规则被替换到子查询的范围表中，得到一个三层的最终查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
```

```
                    sh.sh_avail,
                    sh.slcolor,
                    sh.slminlen,
                    sh.slminlen * un.un_fact AS slminlen_cm,
                    sh.slmaxlen,
                    sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                    sh.slunit
                FROM shoe_data sh, unit un
               WHERE sh.slunit = un.un_name) rsh,
             (SELECT s.sl_name,
                    s.sl_avail,
                    s.sl_color,
                    s.sl_len,
                    s.sl_unit,
                    s.sl_len * u.un_fact AS sl_len_cm
                FROM shoelace_data s, unit u
               WHERE s.sl_unit = u.un_name) rsl
      WHERE rsl.sl_color = rsh.slcolor
        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail > 2;
```

最后规划器会把这个树折叠成一个两层查询树： 最下层的SELECT命令将被“提升”到中间的SELECT中，因为没有必要分别处理它们。但是中间的SELECT仍然和顶层的分开，因为它包含聚集函数。如果我们把它们也提升，它将改变顶层SELECT的行为，这不是我们想要的。不过，折叠查询树是一种优化，重写系统不需要关心它。

41.2.2. 非SELECT语句中的视图规则

有两个查询树的细节在上面的视图规则的描述中没有涉及。它们是命令类型和结果关系。实际上，视图规则不需要命令类型，但是结果关系可能会影响查询重写器工作的方式，因为如果结果关系是一个视图，我们需要采取特殊的措施。

一个SELECT的查询树和其它命令的查询树之间很少的几处不同。显然，它们有不同的命令类型并且对于SELECT之外的命令，结果关系指向结果将进入的范围表项。其它所有东西都完全相同。所以如果有两个表t1和t2分别有列a和b，下面两个语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表包含表t1和t2的项。
- 目标列表包含一个变量，该变量指向表t2的范围表项的列b。
- 条件表达式比较两个范围表项的列a以寻找相等。
- 连接树展示了t1和t2之间的一次简单连接。

结果是，两个查询树生成相似的执行计划：它们都是两个表的连接。 对于UPDATE语句，规划器把t1缺失的列加到目标列并且最终查询树读起来是：

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因此在连接上运行的执行器将产生完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是在UPDATE中有个小问题：执行器计划中执行连接的部分不关心连接的结果的含义。它只是产生一个行的结果集。一个是SELECT命令而另一个是由执行器中的更高层处理的UPDATE命令，在那里执行器知道这是一个UPDATE，并且它知道这个结果应该进入表t1。但是这里的哪些行必须被新行替换呢？

要解决这个问题，在UPDATE和DELETE语句的目标列表里面增加了另外一个项：当前元组 ID（CTID）。这是一个系统列，它包含行所在的文件块编号和在块中的位置。在已知表的情况下，CTID可以被用来检索要被更新的t1的原始行。在添加CTID到目标列之后，该查询实际看起来像：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在，另一个PostgreSQL的细节进入到这个阶段了。表中的旧行还没有被覆盖，这就是为什么ROLLBACK很快的原因。在一个UPDATE中，新的结果行被插入到表中（在剥除CTID之后），并且把CTID指向的旧行的行头部中的cmax和xmax项设置为当前命令计数器和当前事务 ID 。这样旧的行就被隐藏起来，并且在事务提交之后 vacuum 清理器就可以最终移除死亡的行。

知道了所有这些，我们就可以用完全相同的方式简单地把视图规则应用到任意命令中。没有任何区别。

41.2.3. PostgreSQL中视图的能力

上文演示了规则系统如何把视图定义整合到原始的查询树中。在第二个例子中，一个来自于一个视图的简单SELECT创建了一个四表连接（unit以不同的名字被用了两次）的最终查询树。

用规则系统实现视图的好处是，规划器拥有关于哪些表必须被扫描、这些表之间的联系、来自于视图的限制性条件、一个单一查询树中原始查询的条件等所有信息。当原始查询已经是一个视图上的连接时仍然是这样。规划器必须决定执行查询的最优路径，而且规划器拥有越多信息，该决定就越好。并且PostgreSQL中实现的规则系统保证这些信息是此时能获得的有关该查询的所有信息。

41.2.4. 更新一个视图

如果视图是INSERT、UPDATE或DELETE的目标关系会怎样？使用上文所述的替换将给出一个查询树，其中的结果关系指向一个子查询范围表项，这样无法工作。不过，PostgreSQL中有几种方法来支持更新视图。

如果子查询从一个单一基本关系选择并且该关系足够简单，重写器会自动地把该子查询替换成底层的基本关系，这样INSERT、UPDATE或DELETE会被以适当的方式应用到该基本关系。其中“足够简单”的视图被称为自动可更新。有关这种可以被自动更新的视图类别的详细信息，请见CREATE VIEW。

或者，该操作可以被定义在视图上的一个用户提供的INSTEAD OF触发器处理。在这种情况下重写工作有一点点不同。对于INSERT，重写器对视图什么也不做，让它作为查询的结果关系。对于UPDATE和DELETE，仍有必要扩展该视图查询来产生命令将尝试更新或删除的“旧”行。因此该视图被按照通常的方式扩展，但是另一个未被扩展的范围表项会被增加到查询来表示该视图会尽其所能作为结果关系。

现在出现的问题是如何标识在视图中要被更新的行。回忆一下，当结果关系是一个表时，一个特殊的CTID项会被加入到目标列表来标识要被更新的行的物理位置。如果结果关系是一个视图这就行不通，因为一个视图根本就没有CTID，它的行没有实际的物理位置。对于一个UPDATE或DELETE操作，一个特殊的wholerow项会被增加到目标列表中，它会扩展来包括来自该视图的所有列。执行器使用这个值来提供“旧”行给INSTEAD OF触发器。现在就轮到触发器来基于新旧行值来找出要更新什么了。

另外一种可能性是让用户定义INSTEAD规则，这种规则指定对视图上的INSERT\UPDATE和DELETE命令的替代动作。这些规则将重写该命令，通常是重写成一个更新一个或多个表（而不是视图）的命令。这是第 41.4 的主题。

注意规则会首先被计算，然后在原始查询被规划和执行之前重写它。因此，如果一个视图上同时有INSTEAD OF触发器和INSERT、UPDATE或DELETE规则，那么首先会计算规则，然后根据其结果决定是否执行触发器，触发器可能完全都不会被使用。

Automatic rewriting of an 在一个简单视图上的INSERT、UPDATE或DELETE查询的自动重写总是在最后尝试。因此，如果一个视图有规则或触发器，它们将重载自动可更新视图的默认行为。

如果对该视图没有INSTEAD规则或INSTEAD OF触发器，并且重写器不能自动地把该查询重写成一个底层基本关系上的更新，将会抛出一个错误，因为执行器不能更新一个这样的视图。

41.3. 物化视图

PostgreSQL中的物化视图像视图一样使用了规则系统，但是以一种类表的形式保留了结果。在物化视图：

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

和视图：

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

之间的主要区别是物化视图不能直接被更新，并且用于创建物化视图的查询的存储方式和视图查询的存储方式完全相同，因此要为物化视图生成新鲜的数据：

```
REFRESH MATERIALIZED VIEW mymatview;
```

The information about a materialized view in the 有关一个PostgreSQL系统目录中的物化视图的信息和一个表或视图的信息完全相同。因此对于解析器，一个物化视图就是一个关系，就像一个表或一个视图。当一个物化视图被一个查询引用时，数据直接从物化视图中返回，如同表一样；规则只被用来填充物化视图。

虽然对物化视图中存储的数据的访问常常要快于直接访问底层表或通过一个视图访问，但是数据并不总是最新的；但是某些时候并不需要当前数据。考虑一个记录销售情况的表：

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- 销售员的 ID
    invoice_date  date,          -- 销售日期
    invoice_amt   numeric(13,2)  -- 销售量
);
```

如果人们想快速绘制历史销售数据，他们可能希望汇总，并且他们可能并不关心当前日期的不完整数据：

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
```

```
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date
    ORDER BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

这个物化视图可能对在为销售员创建的控制面板上显示一个图表非常有用。可以用一个计划任务在每晚使用这个 SQL 语句更新该统计信息：

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一种使用是允许通过一个外部数据包装器对来　自一个远程系统的数据进行更快的访问。下面有一个使用　　file_fdw的简单例子，但是由于本地系　　统上可以使用高速缓存，因此比起访问一个远程系统的性　　能差异可能会比这里所展示的更大。注意鉴于file_fdw不支持索引，我们也使用这种　能力来在物化视图上放置索引。这种优势可能不适用于其 他种类的外部数据访问。

建立：

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

现在让我们对一个词进行拼写检查。直接使用file_fdw：

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

通过EXPLAIN ANALYZE，我们可以看到：

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual
 time=188.180..188.181 rows=1 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual
 time=188.177..188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
```

```
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

如果使用物化视图，该查询会快很多：

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1
 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0)
 (actual time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

不管哪种方式，单词都是被拼错的，因此让我们看看什么是我们可能想要的。再次使用file_fdw：

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
--------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual
 time=1431.591..1431.594 rows=10 loops=1)
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual
 time=1431.589..1431.591 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32)
 (actual time=0.057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

使用物化视图：

```
 Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10
 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829
 width=10) (actual time=187.219..188.252 rows=10 loops=1)
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

如果你能够忍受定期把远程数据更新到本地数据库，其性能收益可能是巨大的。

41.4. INSERT、UPDATE和DELETE上的规则

定义在INSERT、UPDATE和DELETE上的规则与前一节描述的视图规则有明显的不同。首先，它们的CREATE RULE命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是INSTEAD或ALSO（缺省）。
- 伪关系NEW和OLD变得有用了。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零个或多个新查询树并且可能把原始的那个查询树扔掉。

> **小心**
>
> 在很多情况下，由INSERT/UPDATE/DELETE上的规则执行的任务用触发器能做得更好。触发器在记法上要更复杂些，但是它们的语义理解起来更简单些。当原始查询包含不稳定函数时，规则容易产生令人惊讶的结果：在执行规则的过程中不稳定函数的执行次数可能比语气中的更多。
>
> 还有，有些情况根本无法用这些类型的规则支持，典型的是在原始查询中包括WITH子句以及在UPDATE查询的SET列表中包括多个赋值的子SELECT。这是因为把这些结构复制到一个规则查询中可能导致子查询的多次计算，这与查询作者表达的意图相悖。

41.4.1. 更新规则如何工作

记住以下语法：

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

在随后的内容中，更新规则表示定义在INSERT、UPDATE或DELETE上的规则。

如果查询树的结果关系和命令类型等于CREATE RULE命令中给出的对象和事件，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表。一开始该查询树列表是空的。更新规则中可以有零个（NOTHING关键字）、一个或多个动作。为简单起见，我们先看一个只有一个动作的规则。这个规则可以有条件或者没有条件，并且它可以是INSTEAD或ALSO（缺省）。

什么是规则条件？它是一个限制，告诉规则动作什么时候做、什么时候不做。这个条件只能引用NEW和/或OLD伪关系，它们基本上代表作为对象给定的关系（但是有着特殊含义）。

所以，对这个单动作的规则生成下面的查询树，我们有三种情况。

没有条件，有ALSO或INSTEAD

来自规则动作的查询树，在其上增加原始查询树的条件

给出了条件，有ALSO

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件

给出了条件，有INSTEAD

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件；以及带有反规则条件的原始查询树

最后，如果规则是ALSO，那么未修改的原始查询树也被加入到列表。因为只有合格的INSTEAD规则已经被加入到原始查询树中，对于单动作的规则，我们将结束于一个或两个输出查询树。

对于ON INSERT规则，原始查询（如果没有被INSTEAD取代）是在任何规则增加的动作之前完成的。这样就允许动作看到被插入的行。但是对ON UPDATE 和ON DELETE规则，原始查询是在规则增加的动作之后完成的。这样就确保动作可以看到将要更新或者将要删除的行；否则，动作可能什么也不做，因为它们无法发现符合它们要求的行。

从规则动作生成的查询树会被再次丢给重写系统，并且可能有更多规则被应用而得到更多或更少的查询树。所以一个规则的动作必须有一种不同的命令类型或者和规则所在的关系不同的另一个结果关系。 否则这样的递归处理就会没完没了（规则的递规展开会被检测到，并当作一个错误报告）。

在pg_rewrite系统目录中的动作中的查询树只是模板。因为它们可以引用NEW和OLD的范围表项，在使用它们之前必须做一些替换。对于任何NEW的引用，都要先在原始查询的目标列表中搜索对应的项。如果找到，该项的表达式将会替换该引用。否则NEW和OLD的含义一样（对于UPDATE）或者被替换成一个空值（对于INSERT）。 任何对OLD的引用都用结果关系的范围表项的引用替换。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

41.4.1.1. 第一个规则循序渐进

假设我们想要跟踪shoelace_data关系中的sl_avail列。所以我们建立一个日志表和一条规则，这条规则每次在shoelace_data上执行UPDATE时有条件地写入一个日志项。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- 改变的鞋带
    sl_avail   integer,       -- 新的可用值
    log_who    text,          -- 谁做的
    log_when   timestamp      -- 何时做的
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

现在有人做：

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

然后看看日志表：

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这就是我们所期望的。在后台发生的的事情如下。解析器创建查询树：

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

这是一个带有规则条件表达式的ON UPDATE规则log_shoelace ，条件是：

```
NEW.sl_avail <> OLD.sl_avail
```

它的动作是：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

（这看起来有点奇怪，因为你通常不能写INSERT ... VALUES ... FROM。这里的FROM子句只是表示查询树里有用于new和old的范围表项。这些东西是必需的，这样它们就可以被INSERT命令的查询树中的变量引用）。

该规则是一个有条件的ALSO规则，所以规则系统必须返回两个查询树：更改过的规则动作和原始查询树。在第 1 步里，原始查询的范围表被集成到规则动作的查询树中。得到：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

第 2 步把规则条件增加进去，所以结果集被限制为sl_avail改变了的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

（这看起来更奇怪，因为INSERT ... VALUES也没有WHERE子句，但是规划器和执行器处理它没有任何难度。不管怎样，它们需要为INSERT ... SELECT支持这种相同功能）。

第 3 步把原始查询树的条件加进去，把结果集进一步限制成只有被初始查询树改变的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
```

```
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 4 步把NEW引用替换为来自原始查询树的目标列表项或来自结果关系的相匹配的变量引用：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 5 步，用结果关系引用把OLD引用替换掉：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

这就完成了。因为规则是ALSO，我们还要输出原始查询树。简而言之，从规则系统输出的是一个包含两个查询树的列表，它们与下面语句相对应：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

这些会按照这个顺序被执行，并且这也正是规则要做的事情。

做的替换和追加的条件用于确保对于下面这样的原始查询不会有日志记录被写入：

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

在这种情况下，原始查询树不包含sl_avail的目标列表项，因此NEW.sl_avail将被shoelace_data.sl_avail代替。所以，规则生成的额外命令是：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

并且条件将永远不可能为真。

如果原始查询修改多个行，这也能争产工作。所以如果某人发出命令：

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上有四行（sl1、sl2、sl3和sl4）被更新。但sl3已经是sl_avail = 0。在这种情况下，原始查询树的条件不同并且导致规则产生额外的查询树：

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

这个查询树将肯定插入三个新的日志项。这也是完全正确的。

到这里我们就能明白为什么原始查询树最后执行非常重要。如果UPDATE先被执行，则所有的行都已经被设为零，所以记日志的INSERT将无法找到任何符合0 <> shoelace_data.sl_avail的行。

41.4.2. 与视图合作

要保护一个视图关系不被INSERT、UPDATE或DELETE，一种简单的方法是让那些查询树被丢掉。因此我们可以创建规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

如果现在某人尝试对视图关系shoe做任何这些操作，规则系统将应用这些规则。因为这些规则没有动作而且是INSTEAD，作为的查询树列表将是空的并且整个查询将变得什么也不做，因为经过规则系统处理后没有什么东西剩下来被优化或执行了。

一个更好的使用规则系统的方法是创建一些规则，这些规则把查询树重写成一个在真实表上进行正确的操作的查询树。 要在视图shoelace上做这件事，我们创建下列规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
```

```
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

如果你要在视图上支持RETURNING查询，你需要让规则包含RETURNING子句来计算视图行。这对于基于单个表的视图来说通常非常简单，但是对于连接视图（如shoelace）就有点冗长了。对于插入的一个例子：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

注意，这个规则同时支持该视图上的INSERT和INSERT RETURNING查询 — 对于INSERT会简单地忽略RETURNING子句。

现在假设有时一包鞋带抵达了商店，并且随着它有一个大的清单。但是你不想每次都手工更新shoelace视图。取而代之的是我们建立两个小表：一个用来从清单向其中插入东西，另一个则用了一个特殊的技巧。这些东西的创建命令如下：

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在你可以用来自清单的数据填充表shoelace_arrive：

```
SELECT * FROM shoelace_arrive;
```

```
 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

快速地看一看当前的数据：

```
SELECT * FROM shoelace;
```

```
 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

现在把到的货鞋带移到：

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

并检查结果：

```
SELECT * FROM shoelace ORDER BY sl_name;
```

```
 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+---------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一个INSERT ... SELECT到这些结果经过了很长的过程。并且该查询树转换的描述将出现在本章的最后。首先，这里是解析器的输出：

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则shoelace_ok_ins被应用并且把这个输出转换成：

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

并且丢掉shoelace_ok上的INSERT。这个被重写后的查询被再次传递给规则系统，并且第二个被应用的规则shoelace_upd会产生：

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

同样这是一个INSTEAD规则并且前一个查询树会被丢弃掉。注意这个查询仍然使用视图shoelace。但是规则系统还没有完成这一步，所以它会继续并在其上应用_RETURN规则，并且我们得到：

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最后，规则log_shoelace被应用，生成额外的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
```

```
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

完成这些之后，规则系统用完了所有的规则并且返回生成的查询树。

所以我们结束于两个最终查询树，它们等效于SQL语句：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是从一个关系来的数据插入了到另一个中，改变成第三个上的更新，改变成更新第四个外加做日志，在第五个中的最后更新缩减为两个查询。

有一个小细节有点丑陋。看看那两个查询，我们会发现shoelace_data关系在范围表中出现了两次而实际上绝对可以缩为出现一次。规划器不会处理它，因此INSERT的规则系统输出的执行规划会是

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

在省略额外的范围表项后会得到

```
Merge Join
  ->  Seq Scan
        ->  Sort
```

```
            ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

这在日志表中生成完全一样的项。因此，规则系统导致了shoelace_data表上的一次绝对不必要的扫描。并且同样的冗余扫描会在UPDATE中进行。但是要把这些全部实现实在是一项很困难的工作。

现在我们对PostgreSQL规则系统及其能力做最后一个演示。假设你向你的数据库中添加一些有特别颜色的鞋带：

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

我们想要建立一个视图来检查哪些shoelace项在颜色上不配任何鞋子。适用的视图是：

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

现在我们想建立它，这样没有库存的不匹配的鞋带都会被从数据库中删除。为了对PostgreSQL有点难度，我们不直接删除它们。而是我们再创建一个视图：

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后用下面方法：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

```
 sl5    |        4 | brown    |      1 | m      |       100
 sl6    |       20 | brown    |    0.9 | m      |        90
(9 rows)
```

对一个视图上的DELETE，这个命令带有一个总共使用了四个嵌套/连接视图的子查询条件，这四个视图之一本身有一个包含一个视图的子查询条件，该条件计算使用的视图列； 这个命令被重写成了一个查询树，该查询树从一个真正的表里面把需要删除的数据删除。

在现实世界里只有很少的情况需要上面的这样的构造。但这些东西能运转肯定让你感觉不错。

41.5. 规则和权限

由于PostgreSQL规则系统对查询的重写，会访问没有在原始查询中指定的表/视图。使用更新规则时，这可能包括对表的写权限。

重写规则并不拥有一个独立的所有者。关系（表或视图）的所有者自动成为为其所定义的重写规则的所有者。PostgreSQL规则系统改变了默认的访问控制系统的行为。由于规则被使用的关系会按照规则所有者的权限来检查，而不是调用规则的用户。这表示用户只需要在其查询中显式指定的表/视图上的所需权限。

例如：某用户有一个电话号码列表，其中一些是私人的，另外的一些是办公室助理需要的。该用户可以构建下面的东西：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

除了该用户以外（还有数据库超级用户）没有人可以访问phone_data表。但因为GRANT的原因，助理可以在phone_number视图上运行SELECT。规则系统将把phone_number上的SELECT重写为phone_data上的SELECT。因为该用户是phone_number的所有者，因此也是规则的所有者，对phone_data的读访问现在被根据该用户的权限检查，并且该查询被允许。同时也要检查访问phone_number的权限，但这是针对调用用户进行的，所以除了用户自己和助理外没有人可以使用它。

权限检查是按规则逐条进行的。所以此时助理是唯一的一个可以看到公共电话号码的人。但助理可以建立另一个视图并且赋予该视图公共权限。这样，任何人都可以通过助理的视图看到phone_number数据。 助理不能做的事情是创建一个直接访问phone_data的视图（实际上助理是可以的，但没有任何作用，因为每次访问都会因通不过权限检查而被否定）。而且该用户一旦注意到助理开放了他的phone_number视图，该用户还可以收回助理的访问权限。立刻，所有对助理视图的访问将会失败。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但事实上不是。 如果这样做不能奏效，助理将必须建立一个与phone_number有相同列的表并且每天拷贝一次数据进去。 那么这是助理自己的数据因而助理可以为每一个想要访问的人授权。一个GRANT意味着“我信任你”。 如果某个你信任的人做了上面的事情，那么是时候认为信任已经结束并且要使用REVOKE。

需要注意的是，虽然视图可以用前文展示的技术来隐藏特定列的内容，它们不能可靠地在不可见行上隐藏数据，除非标志被设置。 例如，下面的视图是不安全的：

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图看起来是安全的，因为规则系统会把任何phone_number上的SELECT重写成phone_data上的SELECT，并且增加限制使得只有phone 不以 412 开头的项才被处理。但是如

果用户可以创建自己的函数，那就不难让规划器在NOT LIKE表达式之前先执行用户自定义函数。例如：

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

phone_data表中的每一个人和电话号码会被打印成一个NOTICE，因为规划器会选择在执行NOT LIKE之前先执行tricky，因为前者的开销大。 即使禁止用户自定义一个新函数，内置函数也可以用在类似的攻击中（例如，大部分造型函数会在它们产生的错误信息中包含它们的输入值）。

类似的考虑应用于更新规则。在前一节的例子中，例子数据库中表的所有者可以把shoelace视图上的SELECT、INSERT、UPDATE和DELETE权限授予其他人，但对shoelace_log只有SELECT权限。写日志项的规则动作将仍然可以被成功地执行，并且其它用户可以看到日志项。但他们不能创建伪造的项，并且他们也不能操纵或移除现有的项。在这种情况下，不可能通过让规划器改变操作的顺序来推翻规则，因为引用shoelace_log的唯一规则是无限制的INSERT。在更复杂的情景中，这可能不正确。

当需要对一个视图提供行级安全时，security_barrier属性应该被应用到该视图。这会阻止恶意选择的函数和操作符通过行被传递，直到视图完成其工作。例如，如果前文所示的视图被创建成这样，它就是安全的：

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Views created with the 使用security_barrier创建的视图的性能会远差于没有使用该选项的视图。通常，没有办法来避免这种现状：如果最快的候选计划可能在安全性上折衷，它就必须被拒绝。出于该原因，这个选在在默认情况下是没有启用的。

当处理没有副作用的函数时，查询规划器有更多的灵活性。这类函数被称为LEAKPROOF，并且包括很多简单常用的操作符，例如很多等于操作符。查询规划器可以安全地允许这类函数在查询执行过程中的任何点被计算，因为在用户不可见的行上调用它们将不会泄露关于不可见行的任何信息。更进一步，不接收参数或者不从安全屏障视图得到任何参数的函数不必被标记为LEAKPROOF以便被下推，因为它们从来不会从该视图接收数据。相反，一个可能会基于接收到的参数值抛出错误的函数（例如在溢出或被零除事件中抛出错误的函数）不是防泄漏的，并且如果它被应用在安全性视图的行过滤器之前，它可能会提供有关不可见行的有效信息。

有一点很重要的是理解：即使一个视图使用security_barrier选项创建，它也只在不可见元组不会被传递给可能不安全的函数的前提下才是安全的。用户可能也有其他方式来推断不可见数据；例如，他们可以使用EXPLAIN看到查询计划，或者针对视图来测量查询的运行时间。一个恶意攻击者可能有能力推断有关不可见数据的总量，或者甚至得到有关数据分布的某些信息或最常用值（因为这些东西可以影响计划的运行时间；或者甚至计划的选择，因为它们也被反映在优化器的统计数据中）。如果这类“隐通道”攻击很重要，那么授予任何到该数据的访问都可能是不明智的。

41.6. 规则和命令状态

PostgreSQL服务器为它收到的每个命令返回一个命令状态字符串，例如INSERT 149592 1。没有涉及规则时这很简单，但是查询被规则重写时会发生什么呢？

规则对命令状态的影响如下：

- 如果没有查询的无条件INSTEAD规则，那么原始给出的查询将会被执行，并且它的命令状态将像平常一样被返回（但是请注意如果存在任何有条件INSTEAD规则，那么它们的反条件将被加到原始查询中。这样可能会减少它处理的行数，并且报告的状态将受影响）。
- 如果有查询的任何无条件INSTEAD规则，那么原始查询将完全不被执行。在这种情况下，服务器将返回由服务器将返回由INSTEAD规则（有条件的或无条件的）插入的最后一条和原始查询命令类型（INSERT、UPDATE或DELETE）相同的查询的命令状态。如果任何规则添加的查询都不符合这些要求，那么返回的命令状态显示原始查询类型并且行计数和 OID 域为零。

通过为任何想要的INSTEAD规则指定在活动规则中排名最后的规则名，程序员可以确保该规则都是在第二种情况里设置命令状态的规则，因为它会被最后一个应用。

41.7. 规则 vs 触发器

许多触发器可以干的事情同样也可以用PostgreSQL规则系统来实现。目前不能用规则来实现的东西之一是某些约束，特别是外键。 可以放置一个合格的规则在一列上，这个规则在列的值没有出现在另一个表中时把命令重写成NOTHING。但是这样做数据就会被不声不响地丢弃，因此也不是一个好主意。如果要求检查值的有效性，并且在出现无效值的情况下应该生成一个错误消息，这种需求就必须要用触发器来完成。

在本章中，我们关注于使用规则来更新视图。本章中所有的更新规则的例子都可以使用视图上的INSTEAD OF触发器来实现。编写这类触发器通常比编写规则要容易，特别是在要求使用复杂逻辑来执行更新的情况下。

对于两者都可实现的情况，哪个更好取决于对数据库的使用。触发器为每一个受影响的行都执行一次。规则修改查询树或生成一个额外的查询。所以如果在一个语句中影响到很多行，一个发出额外查询的规则通常可能会比一个触发器快，因为触发器对每一个行都要被调用，并且每次被调用时都需要重新判断要做什么样的操作。不过，触发器方法从概念上要远比规则方法简单，并且很容易让新人上手。

下面我们展示一个例子，该例子说明了在同种情况下两种选择的比较。这里有两个表：

```
CREATE TABLE computer (
    hostname        text,    -- 被索引
    manufacturer    text     -- 被索引
);

CREATE TABLE software (
    software        text,    -- 被索引
    hostname        text     -- 被索引
);
```

两个表都有数千行，并且在hostname上的索引是唯一的。规则或触发器应该实现一个约束，该约束从software中删除引用已删除计算机的行。 触发器可以用下面这条命令：

```
DELETE FROM software WHERE hostname = $1;
```

因为触发器会为每一个从computer中删除的独立行调用一次， 那么它可以准备并且保存这个命令的规划，把hostname作为参数传入。规则应该被写为：

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在看看不同类型的删除。在这种情况：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

表computer被使用索引（快速）扫描，并且由触发器发出的命令也将使用一个索引扫描（同样快速）。来自规则的额外查询应该是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建一个规划

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

所以在触发器和规则的实现之间没有太多的速度差别。

在接下来的删除中，我们想要去掉所有 2000 个hostname以old开头的计算机。有两个命令可以来做这件事。一个是：

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

被规则增加的命令将是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname <
 'ole'
                       AND software.hostname = computer.hostname;
```

计划是：

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一个可能的命令是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会为规划增加的命令产生下面的执行计划：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这表明，当有多个条件表达式被使用AND组合在一起时，规划器不能认识到表computer中hostname上的条件也可以被用于一个software上的索引扫描，　而在该命令的正则表达式版本中正是这样做的。触发器将为要被删除的　2000　个旧计算机中的每一个调用，并且会导致在computer上的一次索引扫描和software上的 2000 次索引扫描。采用规则的实现将会使用两个使用索引的命令来完成。　并且在顺序扫描情况下规则是否仍将更快是取决于software表的总体大小的。即使所有的索引块都将很快地进入高速缓存，通过 SPI 管理器执行来自触发器的 2000 个命令也要花不少时间。

我们要看的最后一个命令是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致很多行被从computer中删除。所以触发器同样会通过执行器运行很多命令。规则生成的命令将会是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了computer上的另一个索引：

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

在任何这些情况之一，来自规则系统的额外命令都或多或少与命令中影响的行数无关。

概括来说，规则只有在其动作导致了大而且糟糕的条件连接时才会明显地慢于触发器，这种情况下规划器将没有什么办法。

第 42 章 过程语言

PostgreSQL允许使用除了 SQL 和 C 之外的其他语言编写用户定义的函数。这些其他的语言通常被称作过程语言（PL）。对于一个用过程语言编写的函数，数据库服务器没有关于如何解释该函数的源文本的内建知识。因此，这个任务被交给一个了解语言细节的特殊处理器。该处理器能够自己处理所有的解析、语法分析、执行工作，或者它可以作为一种PostgreSQL和编程语言既有实现之间的“粘合剂”。就像任何其他 C 函数一样，处理器本身是一个编译到共享对象并且按需载入的 C 语言函数。

在PostgreSQL的标准发布中当前有四种过程语言可用： PL/pgSQL（第 43 章、 PL/Tcl（第 44 章、 PL/Perl（第 45 章以及 PL/Python（第 46 章。 还有其他过程语言可用，但是它们没有被包括在核心发布中。在附录 H中可以了解如何找到它们。另外，用户可以定义其他语言，第 56 章介绍了开发一种新过程语言的基础知识。

42.1. 安装过程语言

在每一个要使用过程语言的数据库中都必须“安装”相应的过程语言。不过安装在数据库template1中的过程语言会被后续创建的数据库自动继承，因为template1中与过程语言相关的项会被CREATE DATABASE复制。因此，数据库管理员可以决定在哪些数据库中可以使用哪些语言，并且按照选择让一些语言默认可用。

对于标准发布所提供的语言，只需要执行CREATE EXTENSION language_name来把该语言安装在当前数据库中。 下文所述的手工过程主要是为了安装没有被包装成扩展的语言。

手工安装过程语言

安装一个过程语言到一个数据库中包括五个步骤，且必须由一个数据库超级用户来执行。在大部分情况下，所需的 SQL 命令应该被打包成一个“扩展”的安装脚本，这样可以用CREATE EXTENSION来执行它们。

1. 用于语言处理器的共享对象必须被编译并安装到一个合适的库目录中。这和编译和安装常规的用户定义 C 函数一样，参见第 38.10.5 节通常，语言处理器将依赖于一个实际提供编程语言引擎的外部库，如果是这样，那些外部库也应该被安装。

2. 处理器必须用下面的命令声明

   ```
   CREATE FUNCTION handler_function_name()
       RETURNS language_handler
       AS 'path-to-shared-object'
       LANGUAGE C;
   ```

   特殊的返回类型language_handler告诉数据库系统，这个函数不返回已定义的SQL数据类型，并且不能直接在SQL语句中使用。

3. （可选的） 可选地，语言处理器能提供一个“内联”处理器函数来执行用这种语言编写的匿名代码块（DO命令）。如果该语言提供了一个内联函数，用类似下面的命令声明它

   ```
   CREATE FUNCTION inline_function_name(internal)
       RETURNS void
       AS 'path-to-shared-object'
       LANGUAGE C;
   ```

4. （可选的） 可选地，语言处理器能提供一个“验证器”函数用来检查一个函数定义的正确性而无需实际执行它。如果验证器函数存在，它将被CREATE FUNCTION调用。如果该语言提供了一个验证器函数，用类似下面的命令声明它

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. 最后，PL 必须用下面的命令声明

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

可选的关键词TRUSTED指定，如果用户不具有访问数据的权限，该语言不会授予对数据的访问。可信的语言是为普通数据库用户（没有超级用户特权）设计的并且允许他们安全地创建函数和过程。由于 PL 函数是在数据库内部执行的，TRUSTED标志只应被给予不允许访问数据库服务器内部或文件系统的语言。语言 PL/pgSQL、 PL/Tcl以及 PL/Perl被认为是可信的，语言 PL/TclU、 PL/PerlU以及 PL/PythonU是被设计用来提供无限制功能的并且不应该被标记为可信。

例 42.展示了手工安装过程如何安装语言PL/Perl。

例 42.1. PL/Perl的手工安装

下列命令告诉数据库服务器在哪里寻找用于PL/Perl语言调用处理器函数的共享对象：

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl有一个内联处理器函数和一个验证器函数，因此我们也要声明它们：

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

命令

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

则定义了前面声明的函数会为语言属性为plperl的函数及过程所调用。

在一个默认的PostgreSQL安装中，用于PL/pgSQL语言的处理器会被编译并且安装到“library”目录，此外PL/pgSQL语言本身会被安装在所有的数据库中。如果配置了Tcl支持，用于PL/Tcl和PL/TclU的处理器会被编译并且安装到库目录中，但语言本身默认不会被安装在任何数据库中。同样地，PL/Perl和PL/PerlU处理器在配置了 Perl 支持时被编译和安装，并且PL/PythonU处理器在配置了 Python 支持时被安装，但是这些语言默认都不会被安装。

第 43 章 PL/pgSQL - SQL过程语言

43.1. 综述

PL/pgSQL是一种用于PostgreSQL数据库系统的可载入的过程语言。PL/pgSQL的设计目标是创建一种这样的可载入过程语言

- 可以被用来创建函数和触发器过程，
- 对SQL语言增加控制结构，
- 可以执行复杂计算，
- 继承所有用户定义类型、函数和操作符，
- 可以被定义为受服务器信任，
- 便于使用。

用PL/pgSQL创建的函数可以被用在任何可以使用内建函数的地方。例如，可以创建复杂条件的计算函数并且后面用它们来定义操作符或把它们用于索引表达式。

在PostgreSQL 9.0 和以后的版本中，PL/pgSQL是默认被安装的。但是它仍然是一种可载入模块，因此特别关注安全性的管理员可以选择移除它。

43.1.1. 使用PL/pgSQL的优点

SQL被PostgreSQL和大多数其他关系数据库用作查询语言。它是可移植的并且容易学习。但是每一个SQL语句必须由数据库服务器单独执行。

这意味着你的客户端应用必须发送每一个查询到数据库服务器、等待它被处理、接收并处理结果、做一些计算，然后发送更多查询给服务器。如果你的客户端和数据库服务器不在同一台机器上，所有这些会引起进程间通信并且将带来网络负担。

通过PL/pgSQL，你可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力并且使 SQL 更易用，但是节省了相当多的客户端/服务器通信开销。

- 客户端和服务器之间的额外往返通信被消除
- 客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送
- 多轮的查询解析可以被避免

与不使用存储函数的应用相比，这能够导致可观的性能提升。

还有，通过PL/pgSQL你可以使用 SQL 中所有的数据类型、操作符和函数。

43.1.2. 支持的参数和结果数据类型

PL/pgSQL编写的函数可以接受服务器支持的任何标量或数组数据类型作为参数，并且它们能够返回任何这些类型的结果。它们也能接受或返回任何用名称指定的组合类型（行类型）。还可以声明一个PL/pgSQL函数为接受record，这表示任意组合类型都将作为输入，或者声明为返回record，表示结果是一种行类型，它的列由调用查询中的说明确定（如第 7.2.1.4 节所讨论）。

PL/pgSQL函数可以通过使用VARIADIC标记被声明为接受数量不定的参数。如第 38.5.5 节所讨论的，它的工作方式和 SQL 函数一样。

PL/pgSQL函数也可以被声明为接受并返回多态类型anyelement、anyarray、anynonarray、anyenum以及anyrange。如第 38.2.5 节所讨论的，由一个多态函数处理的实际数据类型会随着调用改变。在第 43.3.1 节展示了一个例子。

PL/pgSQL函数还能够被声明为返回一个任意（可作为一个单一实例返回的）数据类型的“集合”（或表）。这样的一个函数通过为结果集的每个期望元素执行RETURN NEXT来产生输出，或者通过使用RETURN QUERY来输出一个查询计算的结果。

最后，如果一个PL/pgSQL函数没有可用的返回值，它可以被声明为返回void（另外一种选择是，在那种情况下它可以被写作一个过程）。

PL/pgSQL函数也能够被声明为用输出参数代替返回类型的一个显式说明。这没有为该语言增加任何基础功能，但是它常常很方便，特别是对于要返回多个值的情况。RETURNS TABLE符号也可以被用来替代RETURNS SETOF。

在第 43.3.1 节和第 43.6.1 节有详细的例子。

43.2. PL/pgSQL的结构

通过执行CREATE FUNCTION命令，以PL/pgSQL写成的函数可以被定义到服务器中。这种命令通常看起来是这样：

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

就目前CREATE FUNCTION所关心的来说，函数体就是简单的一个字符串。通常在写函数体时，使用美元符号引用（见第 4.1.2.4 节）通常比使用普通单引号语法更有帮助。如果没有美元引用，函数体中的任何单引号或者反斜线必须通过双写来转义。这一章中几乎所有的例子都在其函数体中使用美元符号引用。

PL/pgSQL是一种块结构的语言。一个函数体的完整文本必须是一个块。一个块被定义为：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

在一个块中的每一个声明和每一个语句都由一个分号终止。如上所示，出现在另一个块中的块必须有一个分号在END之后。不过最后一个结束函数体的END不需要一个分号。

> **提示**
>
> 一种常见的错误是直接在BEGIN之后写一个分号。这是不正确的并且将会导致一个语法错误。

如果你想要标识一个块以便在一个EXIT语句中使用或者标识在该块中声明的变量名，那么label是你唯一需要的。如果一个标签在END之后被给定，它必须匹配在块开始处的标签。

所有的关键词都是大小写无关的。除非被双引号引用，标识符会被隐式地转换为小写形式，就像它们在普通 SQL 命令中。

PL/pgSQL代码中的注释和普通 SQL 中的一样。一个双连字符（--）开始一段注释，它延伸到该行的末尾。一个/*开始一段块注释，它会延伸到匹配*/出现的位置。块注释可以嵌套。

一个块的语句节中的任何语句可以是一个子块。子块可以被用来逻辑分组或者将变量局部化为语句的一个小组。在子块的持续期间，在一个子块中声明的变量会掩盖外层块中相同名称的变量。但是如果你用块的标签限定外层变量的名字，你仍然可以访问它们。例如：

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- 创建一个子块
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints
 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **注意**
>
> 在任何PL/pgSQL函数体的外部确实有一个隐藏的“外层块”包围着。这个块提供了该函数参数（如果有）的声明，以及某些诸如FOUND之类特殊变量（见第 43.5.5 节。外层块被标上函数的名称，这意味着参数和特殊变量可以用该函数的名称限定。

重要的是不要把PL/pgSQL中用来分组语句的BEGIN/END与用于事务控制的同名 SQL 命令弄混。PL/pgSQL的BEGIN/END只用于分组，它们不会开始或结束一个事务。有关PL/pgSQL中管理事务的信息，请参考第 43.8 节此外，一个包含EXCEPTION子句的块实际上会形成一个子事务，它可以被回滚而不影响外层事务。详见第 43.6.8 节

43.3. 声明

在一个块中使用的所有变量必须在该块的声明小节中声明（唯一的例外是在一个整数范围上迭代的FOR循环变量会被自动声明为一个整数变量，并且相似地在一个游标结果上迭代的FOR循环变量会被自动地声明为一个记录变量）。

PL/pgSQL变量可以是任意 SQL 数据类型，例如integer、varchar和char。

这里是变量声明的一些例子：

```
user_id integer;
```

```
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

一个变量声明的一般语法是：

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | :=
 | = } expression ];
```

如果给定DEFAULT子句，它会指定进入该块时分 配给该变量的初始值。如果没有给出DEFAULT子句， 则该变量被初始化为SQL空值。 CONSTANT选项阻止该变量在初始化之后被赋值， 这样它的值在块的持续期内保持不变。COLLATE 选项指定用于该变量的一个排序规则（见 第 43.3.6 节。如果指 定了NOT NULL，对该变量赋值为空值会导致一个 运行时错误。所有被声明为NOT NULL的变量必须 被指定一个非空默认值。 等号（=）可以被用来代替PL/SQL-兼容的 :=。

一个变量的默认值会在每次进入该块时被计算并且赋值给该变量（不是每次函数调用只计算一次）。因此，例如将now()赋值给类型为timestamp的一个变量将会导致该变量具有当前函数调用的时间，而不是该函数被预编译的时间。

例子：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

43.3.1. 声明函数参数

传递给函数的参数被命名为标识符$1、$2等等。可选地，能够为$n参数名声明别名来增加可读性。不管是别名还是数字标识符都能用来引用参数值。

有两种方式来创建一个别名。比较好的方式是在CREATE FUNCTION命令中为参数给定一个名称。例如：

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

另一种方式是显式地使用声明语法声明一个别名。

```
name ALIAS FOR $n;
```

使用这种风格的同一个例子看起来是：

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
```

```
$$ LANGUAGE plpgsql;
```

> **注意**
>
> 这两个例子并非完全等效。在第一种情况中，subtotal可以被引用为sales_tax.subtotal，但在第二种情况中它不能这样引用（如果我们为内层块附加了一个标签，subtotal则可以用那个标签限定）。

更多一些例子：

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- 这里是一些使用 v_string 和 index 的计算
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

当一个PL/pgSQL函数被声明为带有输出参数，输出参数可以用普通输入参数相同的方式被给定$n名称以及可选的别名。一个输出参数实际上是一个最初为 NULL 的变量，它应当在函数的执行期间被赋值。该参数的最终值就是要被返回的东西。例如，sales-tax 例子也可以用这种方式来做：

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

注意我们忽略了RETURNS real — 我们也可以包括它，但是那将是冗余。

当返回多个值时，输出参数最有用。一个小例子是：

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

如第 38.5.4 节所讨论的，这实际上为该函数的结果创建了一个匿名记录类型。如果给定了一个RETURNS子句，它必须RETURNS record。

声明一个PL/pgSQL函数的另一种方式是用RETURNS TABLE，例如：

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

这和声明一个或多个OUT参数并且指定RETURNS SETOF sometype完全等效。

当一个PL/pgSQL函数的返回类型被声明为一个多态类型（anyelement、anyarray、anynonarray、anyenum或anyrange），一个特殊参数$0会被创建。它的数据类型是该函数的实际返回类型，这是从实际输入类型（第 38.2.5 节推演得来。$0被初始化为空并且不能被该函数修改，因此它能够被用来保持可能需要的返回值，不过这不是必须的。$0也可以被给定一个别名。例如，这个函数工作在任何具有一个+操作符的数据类型上：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

通过声明一个或多个输出参数为多态类型可以得到同样的效果。在这种情况下，不使用特殊的$0参数，输出参数本身就用作相同的目的。例如：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

43.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

ALIAS语法比前一节中建议的更一般化：你可以为任意变量声明一个别名，而不只是函数参数。其主要实际用途是为预先决定了名称的变量分配一个不同的名称，例如在一个触发器过程中的NEW或OLD。

例子：

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

因为ALIAS创造了两种不同的方式来命名相同的对象，如果对其使用不加限制就会导致混淆。最好只把它用来覆盖预先决定的名称。

43.3.3. 复制类型

```
variable%TYPE
```

%TYPE提供了一个变量或表列的数据类型。你可以用它来声明将保持数据库值的变量。例如，如果你在users中有一个名为user_id的列。要定义一个与users.user_id具有相同数据类型的变量：

```
user_id users.user_id%TYPE;
```

通过使用%TYPE，你不需要知道你要引用的结构的实际数据类型，而且最重要地，如果被引用项的数据类型在未来被改变（例如你把user_id的类型从integer改为real），你不需要改变你的函数定义。

%TYPE在多态函数中特别有价值，因为内部变量所需的数据类型能在两次调用时改变。可以把%TYPE应用在函数的参数或结果占位符上来创建合适的变量。

43.3.4. 行类型

```
name table_name%ROWTYPE;
name composite_type_name;
```

一个组合类型的变量被称为一个行变量（或行类型变量）。这样一个变量可以保持一个SELECT或FOR查询结果的一整行，前提是查询的列集合匹配该变量被声明的类型。该行值的各个域可以使用通常的点号标记访问，例如rowvar.field。

通过使用table_name%ROWTYPE标记，一个行变量可以被声明为具有和一个现有表或视图的行相同的类型。它也可以通过给定一个组合类型名称来声明（因为每一个表都有一个相关联的具有相同名称的组合类型，所以在PostgreSQL中实际上写不写%ROWTYPE都没有关系。但是带有%ROWTYPE的形式可移植性更好）。

一个函数的参数可以是组合类型（完整的表行）。在这种情况下，相应的标识符$n将是一个行变量，并且可以从中选择域，例如$1.user_id。

这里是一个使用组合类型的例子。table1和table2是已有的表，它们至少有以下提到的域：

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

43.3.5. 记录类型

```
name RECORD;
```

记录变量和行类型变量类似，但是它们没有预定义的结构。它们采用在一个SELECT或FOR命令期间为其赋值的行的真实行结构。一个记录变量的子结构能在每次它被赋值时改变。这样

的结果是直到一个记录变量第一次被赋值之前，它都没有子结构，并且任何尝试访问其中一个域都会导致一个运行时错误。

注意RECORD并非一个真正的数据类型，只是一个占位符。我们也应该认识到当一个PL/pgSQL函数被声明为返回类型record，这与一个记录变量并不是完全相同的概念，即便这样一个函数可能会用一个记录变量来保持其结果。在两种情况下，编写函数时都不知道真实的行结构，但是对于一个返回record的函数，当调用查询被解析时就已经决定了真正的结构，而一个行变量能够随时改变它的行结构。

43.3.6. PL/pgSQL变量的排序规则

当一个PL/pgSQL函数有一个或多个可排序数据类型的参数时，为每一次函数调用都会基于赋值给实参的排序规则来确定出一个排序规则，如第 23.2 节所述。如果一个排序规则被成功地确定（即在参数之间隐式排序规则没有冲突），那么所有的可排序参数会被当做隐式具有那个排序规则。这将在函数中影响行为受到排序规则影响的操作。例如，考虑

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

less_than的第一次使用将会采用text_field_1和text_field_2共同的排序规则进行比较，而第二次使用将采用C排序规则。

此外，被确定的排序规则也被假定为任何可排序数据类型本地变量的排序规则。因此，当这个函数被写为以下形式时，它工作将不会有什么不同

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

如果没有可排序数据类型的参数，或者不能为它们确定共同的排序规则，那么参数和本地变量会使用它们数据类型的默认排序规则（通常是数据库的默认排序规则，但是可能不同于域类型的变量）。

通过在一个可排序数据类型的本地变量的声明中包括COLLATE选项，可以为它指定一个不同的排序规则，例如

```
DECLARE
    local_a text COLLATE "en_US";
```

这个选项会覆盖根据上述规则被给予该变量的排序规则。

还有，如果一个函数想要强制在一个特定操作中使用一个特定排序规则，当然可以在该函数内部写一个显式的COLLATE子句。例如：

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

这会覆盖表达式中使用的表列、参数或本地变量相关的排序规则，就像在纯 SQL 命令中发生的一样。

43.4. 表达式

PL/pgSQL语句中用到的所有表达式会被服务器的主SQL执行器处理。例如，当你写一个这样的PL/pgSQL语句时

```
IF expression THEN ...
```

PL/pgSQL将通过给主 SQL 引擎发送一个查询

```
SELECT expression
```

来计算该表达式。如第 43.11.1 节所详细讨论的，在构造该SELECT命令时，PL/pgSQL变量名的每一次出现会被参数所替换。这允许SELECT的查询计划仅被准备一次并且被重用于之后的对于该变量不同值的计算。因此，在一个表达式第一次被使用时实际发生的本质上是一个PREPARE命令。例如，如果我们已经声明了两个整数变量x和y，并且我们写了

```
IF x < y THEN ...
```

在现象之后发生的等效于

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

并且然后为每一次IF语句的执行，这个预备语句都会被EXECUTE，执行时使用变量的当前值作为参数值。通常这些细节对于一个PL/pgSQL用户并不重要，但是在尝试诊断一个问题时了解它们很有用。更多信息可见第 43.11.2 节

43.5. 基本语句

在这一节和接下来的小节中，我们会描述PL/pgSQL能明确理解的所有语句类型。任何不被识别为这些语句类型之一的被假定为是一个 SQL 命令，并且会被发送给主数据库引擎执行，具体如第 43.5.2 和第 43.5.3 节所述。

43.5.1. 赋值

为一个PL/pgSQL变量赋一个值可以被写为：

```
variable { := | = } expression;
```

正如以前所解释的，这样一个语句中的表达式被以一个 SQL SELECT命令被发送到主数据库引擎的方式计算。 该表达式必须得到一个单一值（如果该变量是一个行或记录变量， 它可能是一个行值）。该目标变量可以是一个简单变量（ 可以选择用一个块名限定）、一个行或记录变量的域或是一个简单 变量或域的数组元素。 等号（=）可以被用来代替 PL/SQL-兼容的 :=。

如果该表达式的结果数据类型不匹配变量的数据类型，该值将被强制为变量 的类型，就好像做了赋值造型一样（见第 10.4 节。 如果没有用于所涉及到的数据类型的赋值造型可用， PL/pgSQL解释器将尝试以文本的方式转换结果值，也就 是在应用结果类型的输出函数之后再应用变量类型的输入函数。注意如果结果 值的字符串形式无法被输入函数所接受，这可能会导致由输入函数产生的运行 时错误。

例子：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

43.5.2. 执行一个没有结果的命令

对于任何不返回行的 SQL 命令（例如没有一个RETURNING子句的INSERT），你可以通过把该命令直接写在一个 PL/pgSQL 函数中执行它。

任何出现在该命令文本中的PL/pgSQL变量名被当作一个参数，并且接着该变量的当前值被提供为运行时该参数的值。这与早前描述的对表达式的处理完全相似，详见第 43.11.1 节

当以这种方式执行一个 SQL 命令时，如第 43.11.2 节讨论的，PL/pgSQL会为该命令缓存并重用执行计划。

有时候计算一个表达式或SELECT查询但抛弃其结果是有用的，例如调用一个有副作用但是没有有用的结果值的函数。在PL/pgSQL中要这样做，可使用PERFORM语句：

```
PERFORM query;
```

这会执行query并且丢弃掉结果。以写一个SQL SELECT命令相同的方式写该query，并且将初始的关键词SELECT替换为PERFORM。对于WITH查询，使用PERFORM并且接着把该查询放在圆括号中（在这种情况中，该查询只能返回一行）。PL/pgSQL变量将被替换到该查询中，正像对不返回结果的命令所作的那样，并且计划被以相同的方式被缓存。还有，如果该查询产生至少一行，特殊变量FOUND会被设置为真，而如果它不产生行则设置为假（见第 43.5.5 节。

> **注意**
>
> 我们可能期望直接写SELECT能实现这个结果，但是当前唯一被接受的方式是PERFORM。一个能返回行的 SQL 命令（例如SELECT）将被当成一个错误拒绝，除非它像下一节中讨论的有一个INTO子句。

一个例子：

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

43.5.3. 执行一个有单一行结果的查询

一个产生单一行（可能有多个列）的 SQL 命令的结果可以被赋值给一个记录变量、行类型变量或标量变量列表。这通过书写基础 SQL 命令并增加一个INTO子句来达成。例如：

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
```

```
DELETE ... RETURNING expressions INTO [STRICT] target;
```

其中target可以是一个记录变量、一个行变量或一个有逗号分隔的简单变量和记录/行域列表。PL/pgSQL变量将被替换到该查询的剩余部分中，并且计划会被缓存，正如之前描述的对不返回行的命令所做的。这对SELECT、带有RETURNING的INSERT/UPDATE/DELETE以及返回行集结果的工具命令（例如EXPLAIN）。除了INTO子句，SQL 命令和它在PL/pgSQL之外的写法一样。

> **提示**
>
> 注意带INTO的SELECT的这种解释和PostgreSQL常规的SELECT INTO命令有很大的不同，后者的INTO目标是一个新创建的表。如果你想要在一个PL/pgSQL函数中从一个SELECT的结果创建一个表，请使用语法CREATE TABLE ... AS SELECT。

如果一行或一个变量列表被用作目标，该查询的结果列必须完全匹配该结果的结构，包括数量和数据类型，否则会发生一个运行时错误。当一个记录变量是目标时，它会自动地把自身配置成查询结果列组成的行类型。

INTO子句几乎可以出现在 SQL 命令中的任何位置。通常它被写成刚好在SELECT命令中的select_expressions列表之前或之后，或者在其他命令类型的命令最后。我们推荐你遵循这种惯例，以防PL/pgSQL的解析器在未来的版本中变得更严格。

如果STRICT没有在INTO子句中被指定，那么target将被设置为该查询返回的第一个行，或者在该查询不返回行时设置为空（注意除非使用了ORDER BY，否则“第一行”的界定并不清楚）。第一行之后的任何结果行都会被抛弃。你可以检查特殊的FOUND变量（见第 43.5.5 节来确定是否返回了一行：

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果指定了STRICT选项，该查询必须刚好返回一行或者将会报告一个运行时错误，该错误可能是NO_DATA_FOUND（没有行）或TOO_MANY_ROWS（多于一行）。如果你希望捕捉该错误，可以使用一个异常块，例如：

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

成功执行一个带STRICT的命令总是会将FOUND置为真。

对于带有RETURNING的INSERT/UPDATE/DELETE，即使没有指定STRICT，PL/pgSQL也会针对多于一个返回行的情况报告一个错误。这是因为没有类似于ORDER BY的选项可以用来决定应该返回哪个被影响的行。

如果为该函数启用了If print_strict_params，那么当因为 STRICT的要求没有被满足而抛出一个错误时，该错误消息 的DETAIL将包括传递给该查询的参数信息。可以通过设置

plpgsql.print_strict_params为所有函数更改 print_strict_params设置，但是只有修改后被编译的函数 才会生效。也可以使用一个编译器选项来为一个函数启用它，例如：

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END
$$ LANGUAGE plpgsql;
```

失败时，这个函数会产生一个这样的错误消息

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> **注意**
>
> STRICT选项匹配 Oracle PL/SQL 的SELECT INTO和相关语句的行为。

对于要处理来自于一个 SQL 查询的结果行的情况，请见第 43.6.6 节

43.5.4. 执行动态命令

很多时候你将想要在PL/pgSQL函数中产生动态命令，也就是每次执行中会涉及到不同表或不同数据类型的命令。PL/pgSQL通常对于命令所做的缓存计划尝试（如第 43.11.2 节讨论）在这种情境下无法工作。要处理这一类问题，需要提供EXECUTE语句：

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

其中command-string是一个能得到一个包含要被执行命令字符串（类型text）的表达式。可选的target是一个记录变量、一个行变量或者一个逗号分隔的简单变量以及记录/行域的列表，该命令的结果将存储在其中。可选的USING表达式提供要被插入到该命令中的值。

在计算得到的命令字符串中，不会做PL/pgSQL变量的替换。任何所需的变量值必须在命令字符串被构造时被插入其中，或者你可以使用下面描述的参数。

还有，对于通过EXECUTE执行的命令不会有计划被缓存。该命令反而在每次运行时都会被做计划。因此，该命令字符串可以在执行不同表和列上动作的函数中被动态创建。

INTO子句指定一个返回行的 SQL 命令的结果应该被赋值到哪里。如果提供了一个行或变量列表，它必须完全匹配查询结果的结构（当使用一个记录变量时，它会自动把它自己配置为匹配结果结构）。如果返回多个行，只有第一个行会被赋值给INTO变量。如果没有返回行，NULL 会被赋值给INTO变量。如果没有指定INTO变量，该查询结果会被抛弃。

如果给出了STRICT选项，除非该查询刚好产生一行，否则将会报告一个错误。

命令字符串可以使用参数值，它们在命令中用$1、$2等引用。这些符号引用在USING子句中提供的值。这种方法常常更适合于把数据值作为文本插入到命令字符串中：它避免了将该值

转换为文本以及转换回来的运行时负荷，并且它更不容易被 SQL 注入攻击，因为不需要引用或转义。一个例子是：

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

需要注意的是，参数符号只能用于数据值 — 如果想要使用动态决定的表名或列名，你必须将它们以文本形式插入到命令字符串中。例如，如果前面的那个查询需要在一个动态选择的表上执行，你可以这么做：

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

一种更干净的方法是为表名或者列名使用format()的 %I规范（被新行分隔的字符串会被串接起来）：

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

另一个关于参数符号的限制是，它们只能在SELECT、INSERT、UPDATE和DELETE命令中工作。在另一种语句类型（通常被称为实用语句）中，即使值是数据值，你也必须将它们以文本形式插入。

在上面第一个例子中，带有一个简单的常量命令字符串和一些USING参数的EXECUTE命令在功能上等效于直接用PL/pgSQL写的命令，并且允许自动发生PL/pgSQL变量替换。重要的不同之处在于，EXECUTE会在每一次执行时根据当前的参数值重新规划该命令，而PL/pgSQL则是创建一个通用计划并且将其缓存以便重用。在最佳计划强依赖于参数值的情况中，使用EXECUTE来明确地保证不会选择一个通用计划是很有帮助的。

EXECUTE目前不支持SELECT INTO。但是可以执行一个纯的SELECT命令并且指定INTO作为EXECUTE本身的一部分。

> **注意**
>
> PL/pgSQL中的EXECUTE语句与EXECUTE PostgreSQL服务器支持的 SQL 语句无关。服务器的EXECUTE语句不能直接在PL/pgSQL函数中使用（并且也没有必要）。

例 43.1. 在动态查询中引用值

在使用动态命令时经常不得不处理单引号的转义。我们推荐在函数体中使用美元符号引用来引用固定的文本（如果你有没有使用美元符界定的老代码，请参考第 43.12.1 节的概述，这样在把上述代码转换成更合理的模式时会省力些）。

动态值需要被小心地处理，因为它们可能包含引号字符。一个使用 format()的例子（这假设你用美元符号引用了函数 体，因此引号不需要被双写）：

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

还可以直接调用引用函数：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

这个例子展示了quote_ident和quote_literal函数的使用（见第 9.4 节。为了安全，在进行一个动态查询中的插入之前，包含列或表标识符的表达式应该通过quote_ident被传递。如果表达式包含在被构造出的命令中应该是字符串的值时，它应该通过quote_literal被传递。这些函数采取适当的步骤来分别返回被封闭在双引号或单引号中的文本，其中任何嵌入的特殊字符都会被正确地转义。

因为quote_literal被标记为STRICT，当用一个空参数调用时，它总是会返回空。在上面的例子中，如果newvalue或keyvalue为空，整个动态查询字符串会变成空，导致从EXECUTE得到一个错误。可以通过使用quote_nullable函数来避免这种问题，它工作起来和quote_literal相同，除了用空参数调用时会返回一个字符串NULL。例如：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

如果正在处理的参数值可能为空，那么通常应该用quote_nullable来代替quote_literal。

通常，必须小心地确保查询中的空值不会递送意料之外的结果。例如如果keyvalue为空，下面的WHERE子句

```
'WHERE key = ' || quote_nullable(keyvalue)
```

永远不会成功，因为在=操作符中使用空操作数得到的结果总是为空。如果想让空和一个普通键值一样工作，你应该将上面的命令重写成

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

（目前，IS NOT DISTINCT FROM的处理效率不如=，因此只有在非常必要时才这样做。关于空和IS DISTINCT的详细信息请见第 9.2 节。

请注意美元符号引用只对引用固定文本有用。尝试写出下面这个例子是一个非常糟糕的主意：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

因为如果newvalue的内容碰巧含有$$，那么这段代码就会出问题。同样的缺点可能适用于你选择的任何其他美元符号引用定界符。因此，要想安全地引用事先不知道的文本，必须恰当地使用quote_literal、quote_nullable或quote_ident。

动态 SQL 语句也可以使用format（见第 9.4.1 节）函数来安全地构造。例如：

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

%I等效于quote_ident并且 %L等效于quote_nullable。 format函数可以和 USING子句一起使用：

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

这种形式更好，因为变量被以它们天然的数据类型格式处理，而不是无条件地把它们转换成文本并且通过%L引用它们。这也效率更高。

动态命令和EXECUTE的一个更大的例子可以在例 43.10中找到，它会构建并且执行一个CREATE FUNCTION命令来定义一个新的函数。

43.5.5. 获得结果状态

有好几种方法可以判断一条命令的效果。第一种方法是使用GET DIAGNOSTICS命令，其形式如下：

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

这条命令允许检索系统状态指示符。CURRENT是一个噪声词（另见第 43.6.8.1 节的GET STACKED DIAGNOSTICS）。每个item是一个关键字， 它标识一个要被赋予给指定变量的状态值（变量应具有正确的数据类型来接收状态值）。表 43.1中展示了当前可用的状态项。冒号等号（:=）可以被用来取代 SQL 标准的=符号。例如：

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

表 43.1. 可用的诊断项

名称	类型	描述
ROW_COUNT	bigint	最近的SQL命令处理的行数
RESULT_OID	oid	最近的SQL命令插入的最后一行的 OID（只有在一条INSERT命令插入到一个具有 OID 的表后才有用）
PG_CONTEXT	text	描述当前调用栈的文本行（见第 43.6.9 节）

第二种判断命令效果的方法是检查一个名为FOUND的boolean类型的特殊变量。在每一次PL/pgSQL函数调用时，FOUND开始都为假。它的值会被下面的每一种类型的语句设置：

- 如果一个SELECT INTO语句赋值了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个PERFORM语句生成（并且抛弃）一行或多行，它将把FOUND设置为真，如果没有产生行则将之设置为假。

- 如果UPDATE、INSERT以及DELETE语句影响了至少一行，它们会把FOUND设置为真，如果没有影响行则将之设置为假。
- 如果一个FETCH语句返回了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个MOVE语句成功地重定位了游标，它将会把FOUND设置为真，否则设置为假。
- 如果一个FOR或FOREACH语句迭代了一次或多次，它将会把FOUND设置为真，否则设置为假。当循环退出时，FOUND用这种方式设置；在循环执行中，尽管FOUND可能被循环体中的其他语句的执行所改变，但它不会被循环语句修改。
- 如果查询返回至少一行，RETURN QUERY和RETURN QUERY EXECUTE语句会把FOUND设为真，如果没有返回行则设置为假。

其他的PL/pgSQL语句不会改变FOUND的状态。尤其需要注意的一点是：EXECUTE会修改GET DIAGNOSTICS的输出，但不会修改FOUND的输出。

FOUND是每个PL/pgSQL函数的局部变量；任何对它的修改只影响当前的函数。

43.5.6. 什么也不做

有时一个什么也不做的占位语句也很有用。例如，它能够指示 if/then/else 链中故意留出的空分支。可以使用NULL语句达到这个目的：

```
NULL;
```

例如，下面的两段代码是等价的：

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- 忽略错误
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- 忽略错误
END;
```

究竟使用哪一种取决于各人的喜好。

> **注意**
>
> 在 Oracle 的 PL/SQL 中，不允许出现空语句列表，并且因此在这种情况下必须使用NULL语句。而PL/pgSQL允许你什么也不写。

43.6. 控制结构

控制结构可能是PL/pgSQL中最有用的（以及最重要）的部分了。利用PL/pgSQL的控制结构，你可以以非常灵活而且强大的方法操纵PostgreSQL的数据。

43.6.1. 从一个函数返回

有两个命令让我们能够从函数中返回数据：RETURN和RETURN NEXT。

43.6.1.1. RETURN

```
RETURN expression;
```

带有一个表达式的RETURN用于终止函数并把expression的值返回给调用者。这种形式被用于不返回集合的PL/pgSQL函数。

如果一个函数返回一个标量类型，表达式的结果将被自动转换成函数的返回类型。但是要返回一个复合（行）值，你必须写一个正好产生所需列集合的表达式。这可能需要使用显式造型。

如果你声明带输出参数的函数，那么就只需要写不带表达式的RETURN。输出参数变量的当前值将被返回。

如果你声明函数返回void，一个RETURN语句可以被用来提前退出函数；但是不要在RETURN后面写一个表达式。

一个函数的返回值不能是未定义。如果控制到达了函数最顶层的块而没有碰到一个RETURN语句，那么会发生一个运行时错误。不过，这个限制不适用于带输出参数的函数以及返回void的函数。在这些情况中，如果顶层的块结束，将自动执行一个RETURN语句。

一些例子：

```
-- 返回一个标量类型的函数
RETURN 1 + 2;
RETURN scalar_var;

-- 返回一个组合类型的函数
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- 必须把列造型成正确的类型
```

43.6.1.2. RETURN NEXT以及RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

当一个PL/pgSQL函数被声明为返回SETOF sometype，那么遵循的过程则略有不同。在这种情况下，要返回的个体项被用一个RETURN NEXT或者RETURN QUERY命令的序列指定，并且接着会用一个不带参数的最终RETURN命令来指示这个函数已经完成执行。RETURN NEXT可以被用于标量和复合数据类型；对于复合类型，将返回一个完整的结果“表”。RETURN QUERY将执行一个查询的结果追加到一个函数的结果集中。在一个单一的返回集合的函数中，RETURN NEXT和RETURN QUERY可以被随意地混合，这样它们的结果将被串接起来。

RETURN NEXT和RETURN QUERY实际上不会从函数中返回 — 它们简单地向函数的结果集中追加零或多行。然后会继续执行PL/pgSQL函数中的下一条语句。随着后继的RETURN NEXT和RETURN QUERY命令的执行，结果集就建立起来了。最后一个RETURN（应该没有参数）会导致控制退出该函数（或者你可以让控制到达函数的结尾）。

RETURN QUERY有一种变体RETURN QUERY EXECUTE，它可以动态指定要被执行的查询。可以通过USING向计算出的查询字符串插入参数表达式，这和在EXECUTE命令中的方式相同。

如果你声明函数带有输出参数，只需要写不带表达式的RETURN NEXT。在每一次执行时，输出参数变量的当前值将被保存下来用于最终返回为结果的一行。注意为了创建一个带有输出参数的集合返回函数，在有多个输出参数时，你必须声明函数为返回SETOF record；或者如果只有一个类型为sometype的输出参数时，声明函数为SETOF sometype。

下面是一个使用RETURN NEXT的函数例子：

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- 这里可以做一些处理
        RETURN NEXT r; -- 返回 SELECT 的当前行
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

这里是一个使用RETURN QUERY的函数的例子：

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- 因为执行还未结束，我们可以检查是否有行被返回
    -- 如果没有就抛出异常。
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- 返回可用的航班或者在没有可用航班时抛出异常。
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> **注意**
>
> 如上所述，目前RETURN NEXT和RETURN QUERY的实现在从函数返回之前会把整个结果集都保存起来。这意味着如果一个PL/pgSQL函数生成一个非常大的结果

集，性能可能会很差：数据将被写到磁盘上以避免内存耗尽，但是函数本身在整个结果集都生成之前不会退出。将来的PL/pgSQL版本可能会允许用户定义没有这种限制的集合返回函数。目前，数据开始被写入到磁盘的时机由配置变量work_mem控制。拥有足够内存来存储大型结果集的管理员可以考虑增大这个参数。

43.6.2. 从过程中返回

过程没有返回值。因此，过程的结束可以不用RETURN语句。 如果想用一个RETURN语句提前退出代码，只需写一个没有表达式的RETURN。

如果过程有输出参数，那么输出参数最终的值会被返回给调用者。

43.6.3. 调用存储过程

PL/pgSQL函数，存储过程或DO块可以使用 CALL调存储用过程。 输出参数的处理方式与纯SQL中CALL的工作方式不同。 存储过程的每个INOUT参数必须和CALL语句中的变量对应， 并且无论存储过程返回什么，都会在返回后赋值给该变量。 例如：

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- prints 15
END
$$;
```

43.6.4. 条件

IF和CASE语句让你可以根据某种条件执行二选其一的命令。PL/pgSQL有三种形式的IF：

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

以及两种形式的CASE：

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

43.6.4.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN语句是IF的最简单形式。 如果条件为真，在THEN和END IF之间的语句将被执行。否则，将忽略它们。

例子：

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

43.6.4.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE语句对IF-THEN进行了增加，它让你能够指定一组在条件不为真时应该被执行的语句（注意这也包括条件为 NULL 的情况）。

例子：

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

43.6.4.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...
]
]
[ ELSE
    statements ]
```

```
END IF;
```

有时会有多于两种选择。IF-THEN-ELSIF则提供了一个简便的方法来检查多个条件。IF条件会被一个接一个测试，直到找到第一个为真的。然后执行相关语句，然后控制会被交给END IF之后的下一个语句（后续的任何IF条件不会被测试）。如果没有一个IF条件为真，那么ELSE块（如果有）将被执行。

这里有一个例子：

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- 嗯，唯一的其他可能性是数字为空
    result := 'NULL';
END IF;
```

关键词ELSIF也可以被拼写成ELSEIF。

另一个可以完成相同任务的方法是嵌套IF-THEN-ELSE语句，如下例：

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

不过，这种方法需要为每个IF都写一个匹配的END IF，因此当有很多选择时，这种方法比使用ELSIF要麻烦得多。

43.6.4.4. 简单CASE

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASE的简单形式提供了基于操作数等值判断的有条件执行。search-expression会被计算（一次）并且一个接一个地与WHEN子句中的每个expression比较。如果找到一个匹配，那么相应的statements会被执行，并且接着控制会被交给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到匹配，ELSE 语句会被执行。但是如果ELSE不存在，将会抛出一个CASE_NOT_FOUND异常。

这里是一个简单的例子：

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

43.6.4.5. 搜索ASE

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASE的搜索形式基于布尔表达式真假的有条件执行。每一个WHEN子句的boolean-expression会被依次计算，直到找到一个得到真的。然后相应的statements会被执行，并且接下来控制会被传递给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到为真的结果，ELSE statements会被执行。但是如果ELSE不存在，那么将会抛出一个CASE_NOT_FOUND异常。

这里是一个例子：

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

这种形式的CASE整体上等价于IF-THEN-ELSIF，不同之处在于CASE到达一个被忽略的ELSE子句时会导致一个错误而不是什么也不做。

43.6.5. 简单循环

使用LOOP、EXIT、CONTINUE、WHILE、FOR和FOREACH语句，你可以安排PL/pgSQL重复一系列命令。

43.6.5.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP定义一个无条件的循环，它会无限重复直到被EXIT或RETURN语句终止。可选的label可以被EXIT和CONTINUE语句用在嵌套循环中指定这些语句引用的是哪一层循环。

43.6.5.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

如果没有给出label，那么最内层的循环会被终止，然后跟在END LOOP后面的语句会被执行。如果给出了label，那么它必须是当前或者更高层的嵌套循环或者语句块的标签。然后该命名循环或块就会被终止，并且控制会转移到该循环/块相应的END之后的语句上。

如果指定了WHEN，只有boolean-expression为真时才会发生循环退出。否则，控制会转移到EXIT之后的语句。

EXIT可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

在和BEGIN块一起使用时，EXIT会把控制交给块结束后的下一个语句。需要注意的是，一个标签必须被用于这个目的；一个没有被标记的EXIT永远无法被认为与一个BEGIN块匹配（这种状况从PostgreSQL 8.4 之前的发布就已经开始改变。这可能允许一个未被标记的EXIT匹配一个BEGIN块）。

例子：

```
LOOP
    -- 一些计算
    IF count > 0 THEN
        EXIT;  -- 退出循环
    END IF;
END LOOP;

LOOP
    -- 一些计算
    EXIT WHEN count > 0;  -- 和前一个例子相同的结果
END LOOP;

<<ablock>>
BEGIN
    -- 一些计算
    IF stocks > 100000 THEN
        EXIT ablock;  -- 导致从 BEGIN 块中退出
    END IF;
    -- 当stocks > 100000时，这里的计算将被跳过
END;
```

43.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

如果没有给出label，最内层循环的下一次迭代会开始。也就是，循环体中剩余的所有语句将被跳过，并且控制会返回到循环控制表达式（如果有）来决定是否需要另一次循环迭代。如果label存在，它指定应该继续执行的循环的标签。

如果指定了WHEN，该循环的下一次迭代只有在boolean-expression为真时才会开始。否则，控制会传递给CONTINUE后面的语句。

CONTINUE可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

例子：

```
LOOP
    -- 一些计算
```

```
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- 一些用于 count IN [50 .. 100] 的计算
END LOOP;
```

43.6.5.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

只要boolean-expression被计算为真，WHILE语句就会重复一个语句序列。在每次进入到循环体之前都会检查该表达式。

例如：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- 这里是一些计算
END LOOP;

WHILE NOT done LOOP
    -- 这里是一些计算
END LOOP;
```

43.6.5.5. FOR（整型变体）

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

这种形式的FOR会创建一个在一个整数范围上迭代的循环。变量name会自动定义为类型integer并且只在循环内存在（任何该变量名的现有定义在此循环内都将被忽略）。给出范围上下界的两个表达式在进入循环的时候计算一次。如果没有指定BY子句，迭代步长为1，否则步长是BY中指定的值，该值也只在循环进入时计算一次。如果指定了REVERSE，那么在每次迭代后步长值会被减除而不是增加。

整数FOR循环的一些例子：

```
FOR i IN 1..10 LOOP
    -- 我在循环中将取值 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- 我在循环中将取值 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- 我在循环中将取值 10,8,6,4,2
END LOOP;
```

如果下界大于上界（或者在REVERSE情况下是小于），循环体根本不会被执行。而且不会抛出任何错误。

如果一个label被附加到FOR循环，那么整数循环变量可以用一个使用那个label的限定名引用。

43.6.6. 通过查询结果循环

使用一种不同类型的FOR循环，你可以通过一个查询的结果进行迭代并且操纵相应的数据。语法是：

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

target是一个记录变量、行变量或者逗号分隔的标量变量列表。target被连续不断被赋予来自query的每一行，并且循环体将为每一行执行一次。下面是一个例子：

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- 现在 "mviews" 有一个来自于 cs_materialized_views 的记录

        RAISE NOTICE 'Refreshing materialized view %s ...',
 quote_ident(mviews.mv_name);
        EXECUTE format('TRUNCATE TABLE %I', mviews.mv_name);
        EXECUTE format('INSERT INTO %I %s', mviews.mv_name, mviews.mv_query);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

如果循环被一个EXIT语句终止，那么在循环之后你仍然可以访问最后被赋予的行值。

在这类FOR语句中使用的query可以是任何返回行给调用者的 SQL 命令：最常见的是SELECT，但你也可以使用带有RETURNING子句的INSERT、UPDATE或DELETE。一些EXPLAIN之类的功能性命令也可以用在这里。

PL/pgSQL变量会被替换到查询文本中，并且如第 43.11.1 和第 43.11.2 节详细讨论的，查询计划会被缓存以用于可能的重用。

FOR-IN-EXECUTE语句是在行上迭代的另一种方式：

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

这个例子类似前面的形式，只不过源查询被指定为一个字符串表达式，在每次进入FOR循环时都会计算它并且重新规划。这允许程序员在一个预先规划好了的命令的速度和一个动态命令的灵活性之间进行选择，就像一个纯EXECUTE语句那样。在使用EXECUTE时，可以通过USING将参数值插入到动态命令中。

另一种指定要对其结果迭代的查询的方式是将它声明为一个游标。这会在第 43.7.4 节描述。

43.6.7. 通过数组循环

FOREACH循环很像一个FOR循环，但不是通过一个 SQL 查询返回的行进行迭代，它通过一个数组值的元素来迭代（通常，FOREACH意味着通过一个组合值表达式的部件迭代；用于通过除数组之外组合类型进行循环的变体可能会在未来被加入）。在一个数组上循环的FOREACH语句是：

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

如果没有SLICE，或者如果没有指定SLICE 0，循环会通过计算expression得到的数组的个体元素进行迭代。target变量被逐一赋予每一个元素值，并且循环体会为每一个元素执行。这里是一个通过整数数组的元素循环的例子：

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

元素会被按照存储顺序访问，而不管数组的维度数。尽管target通常只是一个单一变量，当通过一个组合值（记录）的数组循环时，它可以是一个变量列表。在那种情况下，对每一个数组元素，变量会被从组合值的连续列赋值。

通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代。SLICE值必须是一个不大于数组维度数的整数常量。target变量必须是一个数组，并且它接收数组值的连续切片，其中每一个切片都有SLICE指定的维度数。这里是一个通过一维切片迭代的例子：

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
```

```
NOTICE:  row = {10,11,12}
```

43.6.8. 俘获错误

默认情况下，任何在PL/pgSQL函数中发生的错误会中止该函数的执行，而且实际上会中止其周围的事务。你可以使用一个带有EXCEPTION子句的BEGIN块俘获错误并且从中恢复。其语法是BEGIN块通常的语法的一个扩展：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

如果没有发生错误，这种形式的块只是简单地执行所有statements，并且接着控制转到END之后的下一个语句。但是如果在statements内发生了一个错误，则会放弃对statements的进一步处理，然后控制会转到EXCEPTION列表。系统会在列表中寻找匹配所发生错误的第一个condition。如果找到一个匹配，则执行对应的handler_statements，并且接着把控制转到END之后的下一个语句。如果没有找到匹配，该错误就会传播出去，就好像根本没有EXCEPTION一样：错误可以被一个带有EXCEPTION的闭合块捕捉，如果没有EXCEPTION则中止该函数的处理。

condition的名字可以是附录 A中显示的任何名字。一个分类名匹配其中所有的错误。特殊的条件名OTHERS匹配除了QUERY_CANCELED和ASSERT_FAILURE之外的所有错误类型（虽然通常并不明智，还是可以用名字捕获这两种错误类型）。条件名是大小写无关的。一个错误条件也可以通过SQLSTATE代码指定，例如以下是等价的：

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

如果在选中的handler_statements内发生了新的错误，那么它不能被这个EXCEPTION子句捕获，而是被传播出去。一个外层的EXCEPTION子句可以捕获它。

当一个错误被EXCEPTION捕获时，PL/pgSQL函数的局部变量会保持错误发生时的值，但是该块中所有对持久数据库状态的改变都会被回滚。例如，考虑这个片段：

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

当控制到达对y赋值的地方时，它会带着一个division_by_zero错误失败。这个错误将被EXCEPTION子句捕获。而在RETURN语句中返回的值将是x增加过后的值。但是UPDATE命令的

效果将已经被回滚。不过，在该块之前的INSERT将不会被回滚，因此最终的结果是数据库包含Tom Jones但不包含Joe Jones。

> **提示**
>
> 进入和退出一个包含EXCEPTION子句的块要比不包含EXCEPTION的块开销大的多。因此，只在必要的时候使用EXCEPTION。

例 43.2.UPDATE/INSERT的异常

这个例子使用异常处理来酌情执行UPDATE或 INSERT。我们推荐应用使用带有 ON CONFLICT DO UPDATE的INSERT 而不是真正使用这种模式。下面的例子主要是为了展示 PL/pgSQL如何控制流程：

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- 首先尝试更新见
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- 不在这里，那么尝试插入该键
        -- 如果其他某人并发地插入同一个键，
        -- 我们可能得到一个唯一键失败
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- 什么也不做，并且循环再次尝试 UPDATE
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

这段代码假定unique_violation错误是INSERT造成，并且不是由该表上一个触发器函数中的INSERT导致。如果在该表上有多于一个唯一索引，也可能会发生不正确的行为，因为不管哪个索引导致该错误它都将重试该操作。通过接下来要讨论的特性来检查被捕获的错误是否为所预期的会更安全。

43.6.8.1. 得到有关一个错误的信息

异常处理器经常被用来标识发生的特定错误。有两种方法来得到PL/pgSQL中当前异常的信息：特殊变量和GET STACKED DIAGNOSTICS命令。

在一个异常处理器内，特殊变量SQLSTATE包含了对应于被抛出异常的错误代码（可能的错误代码列表见表 A.1 ）。特殊变量SQLERRM包含与该异常相关的错误消息。这些变量在异常处理器外是未定义的。

在一个异常处理器内，我们也可以用GET STACKED DIAGNOSTICS命令检索有关当前异常的信息，该命令的形式为：

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

每个item是一个关键词，它标识一个被赋予给指定变量（应该具有接收该值的正确数据类型）的状态值。表 43.2中显示了当前可用的状态项。

表 43.2. 错误诊断项

名称	类型	描述
RETURNED_SQLSTATE	text	该异常的 SQLSTATE 错误代码
COLUMN_NAME	text	与异常相关的列名
CONSTRAINT_NAME	text	与异常相关的约束名
PG_DATATYPE_NAME	text	与异常相关的数据类型名
MESSAGE_TEXT	text	该异常的主要消息的文本
TABLE_NAME	text	与异常相关的表名
SCHEMA_NAME	text	与异常相关的模式名
PG_EXCEPTION_DETAIL	text	该异常的详细消息文本（如果有）
PG_EXCEPTION_HINT	text	该异常的提示消息文本（如果有）
PG_EXCEPTION_CONTEXT	text	描述产生异常时调用栈的文本行（见第 43.6.9 节

如果异常没有为一个项设置值，将返回一个空字符串。

这里是一个例子：

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- 某些可能导致异常的处理
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

43.6.9. 获得执行位置信息

GET DIAGNOSTICS（之前在第 43.5.5 节描述）命令检索有关当前执行状态的信息（反之上文讨论的GET STACKED DIAGNOSTICS命令会把有关执行状态的信息报告成一个以前的错误）。它的PG_CONTEXT状态项可用于标识当前执行位置。状态项PG_CONTEXT将返回一个文本字符串，其中有描述该调用栈的多行文本。第一行会指向当前函数以及当前正在执行GET DIAGNOSTICS的命令。第二行及其后的行表示调用栈中更上层的调用函数。例如：

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
```

```
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT返回同类的栈跟踪，但是它描述检测到错误的位置而不是当前位置。

43.7. 游标

和一次执行整个查询不同，可以建立一个游标来封装该查询，并且接着一次读取该查询结果的一些行。这样做的原因之一是在结果中包含大量行时避免内存不足（不过，PL/pgSQL用户通常不需要担心这些，因为FOR循环在内部会自动使用一个游标来避免内存问题）。一种更有趣的用法是返回一个函数已经创建的游标的引用，允许调用者读取行。这提供了一种有效的方法从函数中返回大型行集。

43.7.1. 声明游标变量

所有在PL/pgSQL中对游标的访问都会通过游标变量，它总是特殊的数据类型refcursor。创建游标变量的一种方法是把它声明为一个类型为refcursor的变量。另外一种方法是使用游标声明语法，通常是：

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

（为了对Oracle的兼容性，可以用IS替代FOR）。如果指定了SCROLL，那么游标可以反向滚动；如果指定了NO SCROLL，那么反向取的动作会被拒绝；如果二者都没有被指定，那么能否进行反向取就取决于查询。如果指定了arguments， 那么它是一个逗号分隔的name datatype对的列表， 它们定义在给定查询中要被参数值替换的名称。实际用于替换这些名字的值将在游标被打开之后指定。

一些例子：

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
```

```
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

所有这三个变量都是refcursor类型，但是第一个可以用于任何查询，而第二个已经被绑定了一个完全指定的查询，并且最后一个被绑定了一个参数化查询。（游标被打开时，key将被一个整数参数值替换）。变量curs1被称为未绑定，因为它没有被绑定到任何特定查询。

43.7.2. 打开游标

在一个游标可以被用来检索行之前，它必需先被打开（这是和 SQL 命令DECLARE CURSOR等效的操作）。PL/pgSQL有三种形式的OPEN命令，其中两种用于未绑定游标变量，另外一种用于已绑定的游标变量。

> **注意**
>
> 可以通过第 43.7.4 节描述的FOR语句在不显式打开游标的情况下使用已绑定的游标变量。

43.7.2.1. OPEN FOR query

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

该游标变量被打开并且被给定要执行的查询。游标不能是已经打开的，并且它必需已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询必须是一条SELECT或者其它返回行的东西（例如EXPLAIN）。该查询会按照其它PL/pgSQL中的 SQL 命令同等的方式对待：先代换PL/pgSQL变量名，并且执行计划会被缓存用于可能的重用。当一个PL/pgSQL变量被替换到游标查询中时，替换的值是在OPEN时它所具有的值。对该变量后续的改变不会影响游标的行为。对于一个已经绑定的游标，SCROLL和NO SCROLL选项具有相同的含义。

一个例子：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

43.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                    [ USING expression [, ... ] ];
```

打开游标变量并且执行指定的查询。该游标不能是已打开的，并且必须已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询以和EXECUTE中相同的方式被指定为一个字符串表达式。照例，这提供了灵活性，因此查询计划可以在两次运行之间变化（见第 43.11.2 节），并且它也意味着在该命令字符串上还没有完成变量替换。正如EXECUTE，可以通过format()和USING将参数值插入到动态命令中。SCROLL和NO SCROLL选项具有和已绑定游标相同的含义。

一个例子：

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING
 keyvalue;
```

在这个例子中，表名被通过format()插入到查询中。 col1的比较值被通过一个USING参数插入， 所以它不需要引用。

43.7.2.3. 打开一个已绑定的游标

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

这种形式的OPEN被用于打开一个游标变量，它的查询是在声明时绑定的。该游标不能是已经打开的。当且仅当该游标被声明为接收参数时，才必需出现一个实际参数值表达式的列表。这些值将被替换到命令中。

一个已绑定游标的查询计划总是被认为是可缓存的，在这种情况中没有EXECUTE的等效形式。注意SCROLL和NO SCROLL不能在OPEN中指定，因为游标的滚动行为已经被确定。

使用位置或命名记号可以传递参数值。在位置记号中，所有参数都必须按照顺序指定。在命名记号中，每一个参数的名字被使用:=指定以将它和参数表达式分隔开。类似于第 4.3 节描述的调用函数，也允许混合位置和命名记号。

例子（这些例子使用上面例子中的游标声明）：

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

因为在一个已绑定游标的查询上已经完成了变量替换，实际有两种方式将值传到游标中：给OPEN一个显式参数，或者在查询中隐式引用一个PL/pgSQL变量。不过，只有在已绑定游标之前声明的变量才将会被替换到游标中。在两种情况下，要被传递的值都是在OPEN时确定的。例如，得到上例中curs3相同效果的另一种方式是

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

43.7.3. 使用游标

一旦一个游标已经被打开，那么就可以用这里描述的语句操作它。

这些操作不需要发生在打开该游标开始操作的同一个函数中。你可以从一个函数返回一个refcursor值，并且让调用者在该游标上操作（在内部，refcursor值只是一个包含该游标活动查询的所谓入口的字符串名称。这个名字可以被传递、赋予给其它refcursor变量等等，而不用担心扰乱入口）。

所有入口会在事务的结尾被隐式地关闭。因此一个refcursor值只能在该事务结束前用于引用一个打开的游标。

43.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

就像SELECT INTO一样，FETCH从游标中检索下一行到目标中，目标可以是一个行变量、记录变量或者逗号分隔的简单变量列表。如果没有下一行，目标会被设置为 NULL。与SELECT INTO一样，可以检查特殊变量FOUND来看一行是否被获得。

direction子句可以是 SQL FETCH命令中允许的任何变体，除了那些能够取得多于一行的。即它可以是 NEXT、 PRIOR、 FIRST、 LAST、

ABSOLUTE count、RELATIVE count、FORWARD或者BACKWARD。 省略direction和指定NEXT是一样的。在使用count的形式中，count可以是任意的整数值表达式（与SQL命令FETCH不一样，FETCH仅允许整数常量）。除非游标被使用SCROLL选项声明或打开，否则要求反向移动的direction值很可能会失败。

cursor必须是一个引用已打开游标入口的refcursor变量名。

例子：

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

43.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE重新定位一个游标而不检索任何数据。MOVE的工作方式与FETCH命令很相似，但是MOVE只是重新定位游标并且不返回至移动到的行。与SELECT INTO一样，可以检查特殊变量FOUND来看要移动到的行是否存在。

例子：

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

43.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

当一个游标被定位到一个表行上时，使用该游标标识该行就可以对它进行更新或删除。对于游标的查询可以是什么是有限制的（尤其是不能有分组），并且最好在游标中使用FOR UPDATE。详见DECLARE参考页。

一个例子：

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

43.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE关闭一个已打开游标的底层入口。这样就可以在事务结束之前释放资源，或者释放掉该游标变量以便再次打开。

一个例子：

```
CLOSE curs1;
```

43.7.3.5. 返回游标

PL/pgSQL函数可以向调用者返回游标。这对于返回多行或多列（特别是巨大的结果集）很有用。要想这么做，该函数打开游标并且把该游标的名字返回给调用者（或者简单的使用调用者指定的或已知的入口名打开游标）。调用者接着可以从游标中取得行。游标可以由调用者关闭，或者是在事务关闭时自行关闭。

用于一个游标的入口名可以由编程者指定或者自动生成。要指定一个入口名，只需要在打开refcursor变量之前简单地为它赋予一个字符串。OPEN将把refcursor变量的字符串值用作底层入口的名字。不过，如果refcursor变量为空，OPEN会自动生成一个与任何现有入口不冲突的名称，并且将它赋予给refcursor变量。

> **注意**
>
> 一个已绑定的游标变量被初始化为表示其名称的字符串值，因此入口的名字和游标变量名相同，除非程序员在打开游标之前通过赋值覆盖了这个名字。但是一个未绑定的游标变量最初默认为空值，因此它会收到一个自动生成的唯一名字，除非被覆盖。

下面的例子显示了一个调用者提供游标名字的方法：

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

下面的例子使用了自动游标名生成：

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)
```

```
FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

下面的例子展示了从一个函数中返回多个游标的一种方法：

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

43.7.4. 通过一个游标的结果循环

有一种FOR语句的变体，它允许通过游标返回的行进行迭代。语法是：

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value
 [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

该游标变量必须在声明时已经被绑定到某个查询，并且它不能已经被打开。FOR语句会自动打开游标，并且在退出循环时自动关闭游标。当且仅当游标被声明要使用参数时，才必须出现一个实际参数值表达式的列表。这些值会被替换到查询中，采用OPEN期间的方式（见第 43.7.2.3 节。

变量recordvar会被自动定义为record类型，并且只存在于循环内部（循环中该变量名任何已有定义都会被忽略）。每一个由游标返回的行都会被陆续地赋值给这个记录变量并且执行循环体。

43.8. 事务管理

在由CALL命令调用的过程中以及匿名代码块（DO命令）中，可以用命令COMMIT和ROLLBACK结束事务。在一个事务被使用这些命令结束后，一个新的事务会被自动开始，因此没有单独的START TRANSACTION命令（注意BEGIN和END在PL/pgSQL中有不同的含义）。

这里是一个简单的例子：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
```

```
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END
$$;

CALL transaction_test1();
```

只有在从顶层调用的CALL或DO中才能进行事务控制，在没有任何其他中间命令的嵌套CALL或DO调用中也能进行事务控制。例如，如果调用栈是CALL proc1() → CALL proc2() → CALL proc3()，那么第二个和第三个过程可以执行事务控制动作。但是如果调用栈是CALL proc1() → SELECT func2() → CALL proc3()，则最后一个过程不能做事务控制，因为中间有SELECT。

对于游标循环有特殊的考虑。看看这个例子：

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

通常，游标会在事务提交时被自动关闭。但是，一个作为循环的组成部分创建的游标会自动被第一个COMMIT或ROLLBACK转变成一个可保持游标。这意味着该游标在第一个COMMIT或ROLLBACK处会被完全计算出来，而不是逐行被计算。该游标在循环后仍会被自动删除，因此这通常对用户是不可见的。

有非只读命令（UPDATE ... RETURNING）驱动的游标循环中不允许有事务命令。

事务在一个具有异常处理部分的块中不能被结束。

43.9. 错误和消息

43.9.1. 报告错误和消息

使用RAISE语句报告消息以及抛出错误。

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression
 [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
```

RAISE ;

level选项指定了错误的严重性。允许的级别有DEBUG、LOG、INFO、NOTICE, WARNING以及EXCEPTION，默认级别是EXCEPTION。EXCEPTION会抛出一个错误（通常会中止当前事务）。其他级别仅仅是产生不同优先级的消息。不管一个特定优先级的消息是被报告给客户端、还是写到服务器日志、亦或是二者同时都做，这都由log_min_messages和client_min_messages配置变量控制。详见第 19 章

如果有level， 在它后面可以写一个format（ 它必须是一个简单字符串而不是表达式）。该格式字符串指定要被报告的 错误消息文本。在格式字符串后面可以跟上可选的要被插入到该消息的 参数表达式。在格式字符串中，%会被下一个可选参数 的值所替换。写%%可以发出一个字面的 %。参数的数量必须匹配格式字符串中% 占位符的数量，否则在函数编译期间就会发生错误。

在这个例子中，v_job_id的值将替换字符串中的%：

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

通过写一个后面跟着option = expression项的USING，可以为错误报告附加一些额外信息。每一个expression可以是任意字符串值的表达式。允许的option关键词是：

MESSAGE

设置错误消息文本。这个选项可以被用于在USING之前包括一个格式字符串的RAISE形式。

DETAIL

提供一个错误的细节消息。

HINT

提供一个提示消息。

ERRCODE

指定要报告的错误代码（SQLSTATE），可以用附录 A中所示的条件名，或者直接作为一个五字符的 SQLSTATE 代码。

COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA

提供一个相关对象的名称。

这个例子将用给定的错误消息和提示中止事务：

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

这两个例子展示了设置 SQLSTATE 的两种等价的方法：

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

还有第二种RAISE语法，在其中主要参数是要被报告的条件名或 SQLSTATE，例如：

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

在这种语法中，USING能被用来提供一个自定义的错误消息、细节或提示。另一种做前面的例子的方法是

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

仍有另一种变体是写RAISE USING或者RAISE level USING并且把所有其他东西都放在USING列表中。

RAISE的最后一种变体根本没有参数。这种形式只能被用在一个BEGIN块的EXCEPTION子句中，它导致当前正在被处理的错误被重新抛出。

> **注意**
>
> 在PostgreSQL 9.1 之前，没有参数的RAISE被解释为重新抛出来自包含活动异常处理器的块的错误。因此一个嵌套在那个处理器中的EXCEPTION子句无法捕捉它，即使RAISE位于嵌套EXCEPTION子句的块中也是这样。这种行为很奇怪，也并不兼容 Oracle 的 PL/SQL。

如果在一个RAISE EXCEPTION命令中没有指定条件名以及 SQLSTATE，默认是使用RAISE_EXCEPTION (P0001)。如果没有指定消息文本，默认是使用条件名或 SQLSTATE 作为消息文本。

> **注意**
>
> 当用 SQLSTATE 代码指定一个错误代码时，你不会受到预定义错误代码的限制，而是可以选择任何由五位以及大写 ASCII 字母构成的错误代码，只有00000不能使用。我们推荐尽量避免抛出以三个零结尾的错误代码，因为这些是分类代码并且只能用来捕获整个类别。

43.9.2. 检查断言

ASSERT语句是一种向 PL/pgSQL函数中插入调试检查的方便方法。

```
ASSERT condition [ , message ];
```

condition是一个布尔 表达式，它被期望总是计算为真。如果确实如此， ASSERT语句不会再做什么。但如果结果是假 或者空，那么将发生一个ASSERT_FAILURE异常（如果在计算condition时发生错误， 它会被报告为一个普通错误）。

如果提供了可选的message， 它是一个结果（如果非空）被用来替换默认错误消息文本“assertion failed”的表达式（如果 condition失败）。 message表达式在 断言成功的普通情况下不会被计算。

通过配置参数plpgsql.check_asserts可以启用或者禁用断言测试， 这个参数接受布尔值且默认为on。如果这个参数为off， 则ASSERT语句什么也不做。

注意ASSERT是为了检测程序的 bug，而不是 报告普通的错误情况。如果要报告普通错误，请使用前面介绍的 RAISE语句。

43.10. 触发器函数

PL/pgSQL可以被用来在数据更改或者数据库事件上定义触发器函数。触发器函数用CREATE FUNCTION命令创建，它被声明为一个没有参数并且返回类型为trigger（对于数据更改触发器）或者event_trigger（对于数据库事件触发器）的函数。名为PG_something的特殊局部变量将被自动创建用以描述触发该调用的条件。

43.10.1. 数据改变的触发器

一个数据更改触发器被声明为一个没有参数并且返回类型为trigger的函数。注意，如下所述，即便该函数准备接收一些在CREATE TRIGGER中指定的参数 — 这类参数通过TG_ARGV传递，也必须把它声明为没有参数。

当一个PL/pgSQL函数当做触发器调用时，在顶层块会自动创建一些特殊变量。它们是：

NEW

数据类型是RECORD；该变量为行级触发器中的INSERT/UPDATE操作保持新数据行。在语句级别的触发器以及DELETE操作，这个变量是null。

OLD

数据类型是RECORD；该变量为行级触发器中的UPDATE/DELETE操作保持新数据行。在语句级别的触发器以及INSERT操作，这个变量是null。

TG_NAME

数据类型是name；该变量包含实际触发的触发器名。

TG_WHEN

数据类型是text；是值为BEFORE、AFTER或INSTEAD OF的一个字符串，取决于触发器的定义。

TG_LEVEL

数据类型是text；是值为ROW或STATEMENT的一个字符串，取决于触发器的定义。

TG_OP

数据类型是text；是值为INSERT、UPDATE、DELETE或TRUNCATE的一个字符串，它说明触发器是为哪个操作引发。

TG_RELID

数据类型是oid；是导致触发器调用的表的对象 ID。

TG_RELNAME

数据类型是name；是导致触发器调用的表的名称。现在已经被废弃，并且可能在未来的一个发行中消失。使用TG_TABLE_NAME替代。

TG_TABLE_NAME

数据类型是name；是导致触发器调用的表的名称。

TG_TABLE_SCHEMA

数据类型是name；是导致触发器调用的表所在的模式名。

TG_NARGS

数据类型是integer；在CREATE TRIGGER语句中给触发器函数的参数数量。

TG_ARGV[]

数据类型是text数组；来自CREATE TRIGGER语句的参数。索引从 0 开始记数。非法索引（小于 0 或者大于等于tg_nargs）会导致返回一个空值。

一个触发器函数必须返回NULL或者是一个与触发器为之引发的表结构完全相同的记录/行值。

BEFORE引发的行级触发器可以返回一个空来告诉触发器管理器跳过对该行剩下的操作（即后续的触发器将不再被引发，并且不会对该行发生INSERT/UPDATE/DELETE)。如果返回了一个非空值，那么对该行值会继续操作。返回不同于原始NEW的行值将修改将要被插入或更新的行。因此，如果该触发器函数想要触发动作正常成功而不修改行值，NEW（或者另一个相等的值）必须被返回。要修改将被存储的行，可以直接在NEW中替换单一值并且返回修改后的NEW，或者构建一个全新的记录/行来返回。在一个DELETE上的前触发器情况下，返回值没有直接效果，但是它必须为非空以允许触发器动作继续下去。注意NEW在DELETE触发器中是空值，因此返回它通常没有意义。在DELETE中的常用方法是返回OLD.

INSTEAD OF触发器（总是行级触发器，并且可能只被用于视图）能够返回空来表示它们没有执行任何更新，并且对该行剩余的操作可以被跳过（即后续的触发器不会被引发，并且该行不会被计入外围INSERT/UPDATE/DELETE的行影响状态中）。否则一个非空值应该被返回用以表示该触发器执行了所请求的操作。对于INSERT 和UPDATE操作，返回值应该是NEW，触发器函数可能对它进行了修改来支持INSERT RETURNING和UPDATE RETURNING（这也将影响被传递给任何后续触发器的行值，或者被传递给带有ON CONFLICT DO UPDATE的INSERT语句中一个特殊的EXCLUDED别名引用）。对于DELETE操作，返回值应该是OLD。

一个AFTER行级触发器或一个BEFORE或AFTER语句级触发器的返回值总是会被忽略，它可能也是空。不过，任何这些类型的触发器可能仍会通过抛出一个错误来中止整个操作。

例 43.3展示了PL/pgSQL中一个触发器函数的例子。

例 43.3. 一个 PL/pgSQL 触发器函数

这个例子触发器保证：任何时候一个行在表中被插入或更新时，当前用户名和时间也会被标记在该行中。并且它会检查给出了一个雇员的姓名以及薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- 检查给出了 empname 以及 salary
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- 谁会倒贴钱为我们工作?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;
```

```
        -- 记住谁在什么时候改变了工资单
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

另一种记录对表的改变的方法涉及到创建一个新表来为每一个发生的插入、更新或删除保持一行。这种方法可以被认为是对一个表的改变的审计。例 43.4展示了PL/pgSQL中一个审计触发器函数的例子。

例 43.4. 一个用于审计的 PL/pgSQL 触发器函数

这个例子触发器保证了在emp表上的任何插入、更新或删除一行的动作都被记录（即审计）在emp_audit表中。当前时间和用户名会被记录到行中，还有在其上执行的操作类型。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- 在 emp_audit 中创建一行来反映 emp 上执行的动作，
        -- 使用特殊变量 TG_OP 来得到操作。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        END IF;
        RETURN NULL; -- 因为这是一个 AFTER 触发器，结果被忽略
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

前一个例子的一种变体使用一个视图将主表连接到审计表来展示每一项最后被修改是什么时间。这种方法还是记录了对于表修改的完整审查跟踪，但是也提供了审查跟踪的一个简化视图，只为每一个项显示从审查跟踪生成的最后修改时间戳。例 43.5展示了在PL/pgSQL中一个视图上审计触发器的例子。

例 43.5. 一个用于审计的 PL/pgSQL 视图触发器函数

这个例子在视图上使用了一个触发器让它变得可更新，并且确保视图中一行的任何插入、更新或删除被记录（即审计）在emp_audit表中。当前时间和用户名会被与执行的操作类型一起记录，并且该视图会显示每一行的最后修改时间。

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- 执行 emp 上所要求的操作，并且在 emp_audit 中创建一行来反映对 emp 的改
变。
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

触发器的一种用法是维护一个表的另一个汇总表。作为结果的汇总表可以用来在特定查询中替代原始表 — 通常会大量减少运行时间。这种技术常用于数据仓库中，在其中被度量或被观察数据的表（称为事实表）可能会极度大。例 43.6展示了PL/pgSQL中一个为数据仓库事实表维护汇总表的触发器函数的例子。

例 43.6. 一个 PL/pgSQL 用于维护汇总表的触发器函数

这里详述的模式有一部分是基于 Ralph Kimball 所作的The Data Warehouse Toolkit中的Grocery Store例子。

```
--
-- 主表 - 时间维度和销售事实。
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- 汇总表 - 按时间汇总销售
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- 在 UPDATE、INSERT、DELETE 时修改汇总列的函数和触发器。
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
```

```
    BEGIN

        -- 算出增量/减量数。
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- 禁止更改 the time_key 的更新-
            -- （可能不会太麻烦，因为大部分的更改是用 DELETE + INSERT 完成的）。
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                    OLD.time_key,
NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- 插入或更新带有新值的汇总行。
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );
```

```
                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- 什么也不做
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

AFTER也可以利用传递表来观察被触发语句更改的整个行集合。CREATE TRIGGER命令会为一个或者两个传递表分配名字，然后函数可以引用那些名字，就好像它们是只读的临时表一样。例 43.展示了一个例子。

例 43.7. 用传递表进行审计

这个例子产生和例 43.相同的结果，但并未使用一个为每一行都触发的触发器，而是在把相关信息收集到一个传递表中之后用了一个只为每个语句引发一次的触发器。当调用语句修改了很多行时，这种方法明显比行触发器方法快。注意我们必须为每一种事件建立一个单独的触发器声明，因为每种情况的REFERENCING子句必须不同。但是这并不能阻止我们使用单一的触发器函数（实际上，使用三个单独的函数会更好，因为可以避免在TG_OP上的运行时测试）。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
```

```
        -- 在emp_audit中创建行来反映在emp上执行的操作，
        -- 利用特殊变量TG_OP来区分操作。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), user, o.* FROM old_table o;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), user, n.* FROM new_table n;
        END IF;
        RETURN NULL; -- 由于这是一个AFTER触发器，所以结果被忽略
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

43.10.2. 事件触发器

PL/pgSQL可以被用来定义事件触发器。PostgreSQL要求一个可以作为事件触发器调用的函数必须被声明为没有参数并且返回类型为event_trigger。

当一个PL/pgSQL函数被作为一个事件触发器调用，在顶层块中会自动创建一些特殊变量。它们是：

TG_EVENT

数据类型是text；它是一个表示引发触发器的事件的字符串。

TG_TAG

数据类型是text；它是一个变量，包含了该触发器为之引发的命令标签。

例 43.8展示了PL/pgSQL中一个事件触发器函数的例子。

例 43.8. 一个 PL/pgSQL 事件触发器函数

这个例子触发器在受支持命令每一次被执行时会简单地抛出一个NOTICE消息。

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

43.11. PL/pgSQL的内部

这一节讨论了一些PL/pgSQL用户应该知道的一些重要的实现细节。

43.11.1. 变量替换

一个PL/pgSQL函数中的 SQL 语句和表达式能够引用该函数的变量和参数。在现象背后，PL/pgSQL会为这些引用替换查询参数。只有在语法上允许一个参数或列引用的地方才会替换参数。作为一种极端情况，考虑这个编程风格糟糕的例子：

```
INSERT INTO foo (foo) VALUES (foo);
```

foo的第一次出现在语法上必须是一个表名， 因此它将不会被替换，即使该函数有一个名为foo的变量。第二次出现必须是该表的一列的名称，因此它也将不会被替换。只有第三次出现是对该函数变量引用的候选。

> **注意**
>
> PostgreSQL 9.0 之前的版本将尝试替换所有三种情况的变量，这会导致语法错误。

因为变量名在语法上与表列的名字没什么区别，在也引用表的语句中会有歧义：一个给定的名字意味着一个表列或一个变量？让我们把前一个例子改成：

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

这里，dest和src必须是表名，并且col必须是dest的一列，但是foo和bar可能该函数的变量或者src的列。

默认情况下，如果一个 SQL 语句中的名称可能引用一个变量或者一个表列，PL/pgSQL将报告一个错误。修复这种问题的方法很多：你可以重命名变量或列来，或者对有歧义的引用加以限定，或者告诉PL/pgSQL要引用哪种解释。

最简单的解决方案是重命名变量或列。一种常用的编码规则是为PL/pgSQL变量使用一种不同于列名的命名习惯。例如，如果你将函数变量统一地命名为v_something，而你的列名不会开始于v_，就不会发生冲突。

另外你可以限定有歧义的引用让它们变清晰。在上面的例子中，src.foo将是对表列的一种无歧义的引用。要创建对一个变量的无歧义引用，在一个被标记的块中声明它并且使用块的标签（见第 43.2 节。例如

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

这里block.foo表示变量，即使在src中有一个列foo。函数参数以及诸如FOUND的特殊变量，都能通过函数的名称被限定，因为它们被隐式地声明在一个带有该函数名称的外层块中。

有时候在一个大型的PL/pgSQL代码体中修复所有的有歧义引用是不现实的。在这种情况下，你可以指定PL/pgSQL应该将有歧义的引用作为变量（这与PL/pgSQL在 PostgreSQL 9.0 之前的行为兼容）或表列（这与某些其他系统兼容，例如Oracle）解决。

要在系统范围内改变这种行为，将配置参数plpgsql.variable_conflict设置为error、use_variable或者use_column（这里error是出厂设置）之一。这个参数会影响PL/pgSQL函数中语句的后续编译，但是不会影响在当前会话中已经编译过的语句。因为改变这个设置能够导致PL/pgSQL函数中行为的意想不到的改变，所以只能由一个超级用户来更改它。

你也可以对逐个函数设置该行为，做法是在函数文本的开始插入这些特殊命令之一：

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

这些命令只影响它们所属的函数，并且会覆盖plpgsql.variable_conflict的设置。一个例子是：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

在UPDATE命令中，curtime、comment以及id将引用该函数的变量和参数，不管users有没有这些名称的列。注意，我们不得不在WHERE子句中对users.id的引用加以限定，以便让它引用表列。但是我们不需要在UPDATE列表中把对comment的引用限定为一个目标，因为语法上那必须是users的一列。我们可以用下面的方式写一个相同的不依赖于variable_conflict设置的函数：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment =
 stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

被交给EXECUTE或其变体的命令字符串中不会发生变量替换。如果你需要插入一个变化值到这样一个命令中，在构建该字符串值时就这样做，或者使用USING，如第 43.5.4 节所阐明的。

当前变量替换只能在SELECT、INSERT、UPDATE和DELETE命令中工作，因为主 SQL 引擎只允许查询参数在这些命令中。要在其他语句类型（通常被称为实用语句）中使用一个非常量名称或值，你必须将实用语句构建为一个字符串并且EXECUTE它。

43.11.2. 计划缓存

在函数被第一次调用时（在每个会话中），PL/pgSQL解释器解析函数的源文本并且产生一个内部的二进制指令树。该指令树完全翻译了PL/pgSQL语句结构，但是该函数中使用的SQL表达式以及SQL命令并没有被立即翻译。

作为该函数中每一个表达式和第一次被执行的SQL命令，PL/pgSQL解释器使用SPI管理器的SPI_prepare函数解析并且分析该命令来创建一个预备语句。对于那个表达式或命令的后续访问将会重用该预备语句。因此，一个带有很少被访问的条件性代码路径的函数将永远不会发生分析那些在当前会话中永远不被执行的命令的开销。一个缺点是在一个特定表达式或命令中的错误将不能被检测到，直到函数的该部分在执行时被到达（不重要的语法错误在初始的解析中就会被检测到，但是任何更深层次的东西将只有在执行时才能检测到）。

PL/pgSQL（或者更准确地说是 SPI 管理器）能进一步尝试缓冲与任何特定预备语句相关的执行计划。如果没有使用一个已缓存的计划，那么每次访问该语句时都会生成一个新的执行计划，并且当前的参数值（也就是PL/pgSQL的变量值）可以被用来优化被选中的计划。如果该语句没有参数，或者要被执行很多次，SPI 管理器将考虑创建一个不依赖特定参数值的一般计划并且将其缓存用于重用。通常只有在执行计划对其中引用的PL/pgSQL变量值不那么敏感时，才会这样做。如果这样做，每一次生成的计划就是纯利。关于预备语句的行为请详见PREPARE。

由于PL/pgSQL保存预备语句并且有时候以这种方式保存执行计划，直接出现在一个PL/pgSQL函数中的 SQL 命令必须在每次执行时引用相同的表和列。也就是说，你不能在一个 SQL 命令中把一个参数用作表或列的名字。要绕过这种限制，你可以构建PL/pgSQL EXECUTE使用的动态命令，但是会付出在每次执行时需要执行新解析分析以及构建新执行计划的代价。

记录变量的易变天性在这个关系中带来了另一个问题。当一个记录变量的域被用在表达式或语句中时，域的数据类型不能在该函数的调用之间改变，因为每一个表达式被分析时都将使用第一次到达该表达式时存在的数据类型。必要时，可以用EXECUTE来绕过这个问题。

如果同一个函数被用作一个服务于多个表的触发器，PL/pgSQL会为每一个这样的表独立地准备并缓存语句 — 也就是对每一种触发器函数和表的组合都会有一个缓存，而不是每个函数一个缓存。这减轻了数据类型变化带来的问题。例如，一个触发器函数将能够成功地使用一个名为key的列工作，即使该列正好在不同的表中有不同的类型。

同样，具有多态参数类型的函数也会为它们已经被调用的每一种实参类型组合都保留一个独立的缓存，这样数据类型差异不会导致意想不到的失败。

语句缓存有时可能在解释时间敏感的值时产生令人惊讶的效果。例如这两个函数做的事情就有区别：

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

以及：

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

在logfunc1中，PostgreSQL的主解析器在分析INSERT时就知道字符串'now'应该被解释为timestamp，因为logtable的目标列是这种类型。因此，在INSERT被分析时'now'将被转换为一个timestamp常量，并且在该会话的生命周期内被用于所有对logfunc1的调用。不用说，这不是程序员想要的。一个更好的主意是使用now()或current_timestamp函数。

在logfunc2中，PostgreSQL的主解析器不知道'now'应该变成什么类型并且因此返回一个text类型的包含字符串now的数据值。在确定对本地变量curtime的赋值期间，PL/pgSQL解

释器通过调用用于转换的text_out以及timestamp_in函数将这个字符串造型为timestamp类型。因此，计算出来的时间戳会按照程序员的期待在每次执行时更新。虽然这正好符合预期，但是它的效率很糟糕，因此使用now()函数仍然是一种更好的方案。

43.12. PL/pgSQL开发提示

在PL/pgSQL中进行开发的一种好方法是使用你自己选择的文本编辑器来创建函数，并且在另一个窗口中使用psql来载入并且测试那些函数。如果你正在这样做，使用CREATE OR REPLACE FUNCTION来编写函数是一个好主意。用那种方式你只需要重载该文件来更新函数的定义。例如：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
        ....
$$ LANGUAGE plpgsql;
```

在运行psql期间，你可以用下面的命令载入或者重载这样一个函数定义文件：

```
\i filename.sql
```

并且接着立即发出 SQL 命令来测试该函数。

另一种在PL/pgSQL中开发的方式是用一个 GUI 数据库访问工具，它能方便对过程语言的开发。这种工具的一个例子是pgAdmin。这些工具通常提供方便的特性，例如转义单引号以及便于重新创建和调试函数。

43.12.1. 处理引号

一个PL/pgSQL函数的代码在一个CREATE FUNCTION中被指定为一个字符串。如果你用通常的方式把该字符串写在单引号中间，那么该函数体中的任何单引号都必须被双写；同样任何反斜线也必须被双写（假定使用了转义字符串语法）。双写引号最多有点冗长，并且在更复杂的情况中代码会变得完全无法理解，因为你很容易发现你需要半打或者更多相邻的引号。我们推荐你转而把函数体写成一个“美元引用”的字符串（见第 4.1.2.4 节。在美元引用方法中，你从不需要双写任何引号。但是要注意为你需要的每一层嵌套选择一个不同的美元引用定界符。例如，你可能把CREATE FUNCTION命令写成：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE plpgsql;
```

在这里面，你可以在 SQL 命令中为简单字符串使用引号并且用$$来界定被你组装成字符串的 SQL 命令片段。如果你需要引用包括$$的文本，你可以使用Q等等。

下列图表展示了在写没有美元引用的引号时需要做什么。在将之前用美元引用的代码翻译成更容易理解的代码时，它们会有所帮助。

1 个引号

用来开始和结束函数体，例如：

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

在一个单引号引用的函数体中的任何位置，引号必须成对出现。

2 个引号

用于函数体内的字符串，例如：

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

在美元引用方法中，你只需要写：

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

这恰好就是PL/pgSQL在两种情况中会看到的。

4 个引号

当你在函数内的一个字符串常量中需要一个单引号时，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

实际会被追加到a_output的值将是： AND name LIKE 'foobar' AND xyz。

在美元引用方法中，你可以写：

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

要小心在这周围的任何美元引用定界符不只是$$。

6 个引号

当在函数体内的一个字符串中的一个单引号与该字符串常量末尾相邻，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

被追加到a_output的值则将是： AND name LIKE 'foobar'。

在美元引用方法中，这会变成：

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 个引号

当你想在一个字符串常量（占 8 个引号）中有两个单引号时并且这会挨着该字符串常量的末尾（另外 2 个）。如果你正在写一个产生其他函数的函数（如例 43.10中），你将很可能只需要这种。例如：

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

a_output的值将是：

```
if v_... like ''...'' then return ''...''; end if;
```

在美元引用方法中，这会变成：

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
```

```
|| referrer_keys.key_string || $$'
then return '$$  || referrer_keys.referrer_type
|| $$'; end if;$$;
```

这里我们假定我们只需要把单引号放在a_output中，因为在使用前它将被再引用。

43.12.2. 额外的编译时检查

为了辅助用户在一些简单但常见的问题产生危害之前找到它们， PL/pgSQL提供了额外的检查。当被启用时， 根据配置，它们可以在一个函数的编译期间被用来发出 WARNING或者ERROR。一个已经收到了 WARNING的函数可以被继续执行而不会产生进一步的消息， 因此建议你在一个单独的开发环境中进行测试。

这些额外的检查通过配置变量plpgsql.extra_warnings 和plpgsql.extra_errors启用，其中前者用于警告而后者用 于错误。两者都可以被设置为一个用逗号分隔的检查列表、"none"或者"all"。默认值是"none"。 当前列表中可用的检查只有一种：

shadowed_variables

检查一个声明是否遮盖了另一个之前定义的变量。

下面的例子展示了将plpgsql.extra_warnings 设置为shadowed_variables的效果：

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

43.13. 从Oracle PL/SQL 移植

这一节解释了PostgreSQL的PL/pgSQL语言和 Oracle 的PL/SQL语言之间的差别，用以帮助那些从Oracle®向PostgreSQL移植应用的人。

PL/pgSQL与 PL/SQL 在许多方面都非常类似。它是一种块结构的、命令式的语言并且所有变量必须先被声明。赋值、循环和条件则很类似。在从PL/SQL向PL/pgSQL移植时必须记住一些事情：

- 如果一个 SQL 命令中使用的名字可能是一个表的列名或者是对一个函数中变量的引用，那么PL/SQL会将它当作一个列名。如第 43.11.1 节所述，这对应的是PL/pgSQL的plpgsql.variable_conflict = use_column行为（不是默认行为）。通常最好是首先避免这种歧义，但如果不得不移植依赖于该行为的大量代码，那么设置variable_conflict将是最好的方案。

- 在PostgreSQL中，函数体必须写成字符串文本。因此你需要使用美元符引用或者转义函数体中的单引号（见第 43.12.1 节。

- 数据类型名称常常需要翻译。例如，在 Oracle 中字符串值通常被声明为类型varchar2，这并非 SQL 标准类型。在PostgreSQL中则要使用类型varchar或者text来替代。类似地，要把类型number替换成numeric，或者在适当的时候使用某种其他数字数据类型。

- 应该用模式把函数组织成不同的分组，而不是用包。
- 因为没有包，所以也没有包级别的变量。这一点有时候挺讨厌。你可以在临时表里保存会话级别的状态。
- 带有REVERSE的整数FOR循环的工作方式不同：PL/SQL中是从第二个数向第一个数倒数，而PL/pgSQL是从第一个数向第二个数倒数，因此在移植时需要交换循环边界。不幸的是这种不兼容性是不太可能改变的（见第 43.6.5.5 节。
- 查询上的FOR循环（不是游标）的工作方式同样不同：目标变量必须已经被声明，而PL/SQL总是会隐式地声明它们。但是这样做的优点是在退出循环后，变量值仍然可以访问。
- 在使用游标变量方面，存在一些记法差异。

43.13.1. 移植示例

例 43.9展示了如何从PL/SQL移植一个简单的函数到PL/pgSQL中。

例 43.9. 从PL/SQL移植一个简单的函数到PL/pgSQL

这里有一个Oracle PL/SQL函数：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

让我们过一遍这个函数并且看看与PL/pgSQL相比有什么样的不同：

- 类型名称varchar2被改成了varchar或者text。在这一节的例子中，我们将使用varchar，但如果不需要特定的字符串长度限制，text常常是更好的选择。
- 在函数原型中（不是函数体中）的RETURN关键字在PostgreSQL中变成了RETURNS。还有，IS变成了AS，并且你还需要增加一个LANGUAGE子句，因为PL/pgSQL并非唯一可用的函数语言。
- 在PostgreSQL中，函数体被认为是一个字符串，所以你需要使用引号或者美元符号包围它。这代替了Oracle 方法中的用于终止的/。
- 在PostgreSQL中没有show errors命令，并且也不需要这个命令，因为错误是自动报告的。

这个函数被移植到PostgreSQL后看起来会是这样：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
```

```
END;
$$ LANGUAGE plpgsql;
```

例 43.10展示了如何移植一个会创建另一个函数的函数，以及如何处理引号问题。

例 43.10. 从PL/SQL移植一个创建另一个函数的函数到PL/pgSQL

下面的过程从一个SELECT语句抓取行，并且为了效率而构建一个带有IF语句中结果的大型函数。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN
 VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS
 BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

这里是PostgreSQL的版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() RETURNS void AS $func
$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
```

```
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

请注意函数体是如何被单独构建并且通过quote_literal被传递以双写其中的任何引号。需要这个技术是因为无法安全地使用美元引用定义新函数：我们不确定从referrer_key.key_string域中来的什么字符串会被插入（我们这里假定referrer_key.kind可以确信总是为host、domain或者url，但是referrer_key.key_string可能是任何东西，特别是它可能包含美元符号）。这个函数实际上是在 Oracle 的原版上的改进，因为当referrer_key.key_string或者referrer_key.referrer_type包含引号时，它将不会生成坏掉的代码。

例 43.1展示了如何移植一个带有OUT参数和字符串处理的函数。PostgreSQL没有内建的instr函数，但是你可以用其它函数的组合来创建一个。在第 43.13.3 节有一个instr的PL/pgSQL实现，你可以用它让你的移植变得更容易。

例 43.11. 从PL/SQL移植一个带有字符串操作以及OUT参数的过程到PL/pgSQL

下面的Oracle PL/SQL 过程被用来解析一个 URL 并且返回一些元素（主机、路径和查询）。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- 这将被传回去
    v_path OUT VARCHAR2,  -- 这个也是
    v_query OUT VARCHAR2) -- 还有这个
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;
```

```
    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

这里是一种到PL/pgSQL的可能翻译：

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- 这将被传递回去
    v_path OUT VARCHAR,  -- 这个也是
    v_query OUT VARCHAR) -- 以及这个
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

这个函数可以这样使用：

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

例 43.12展示了如何移植一个使用了多种 Oracle 特性的过程。

例 43.12. 从PL/SQL移植一个过程到PL/pgSQL

Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
 NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        raise_application_error(-20000,
                'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- 如果已经存在也不用担心
    END;
    COMMIT;
END;
/
show errors
```

这是我们如何将这个过程移植到PL/pgSQL：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
 NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        RAISE EXCEPTION 'Unable to create a new job: a job is currently
 running'; -- 1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN -- 2
```

```
            -- 如果已经存在不要担心
    END;
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

1. RAISE的语法与 Oracle 的语句相当不同，尽管基本的形式RAISE exception_name工作起来是相似的。
2. PL/pgSQL所支持的异常名称不同于 Oracle。内建的异常名称集合要更大（见附录 A 。目前没有办法声明用户定义的异常名称，尽管你能够抛出用户选择的SQLSTATE 值。

43.13.2. 其他要关注的事项

这一节解释了在移植 Oracle PL/SQL函数到PostgreSQL中时要关注的一些其他问题。

43.13.2.1. 异常后隐式回滚

在PL/pgSQL，当一个异常被EXCEPTION子句捕获之后，从该块的BEGIN以来的所有数据库改变都会被自动回滚。也就是，该行为等效于你在 Oracle 中用下面的代码得到的效果：

```
BEGIN
    SAVEPOINT s1;
    ... 代码 ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... 代码 ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... 代码 ...
END;
```

如果你正在翻译一个使用这种风格的SAVEPOINT以及ROLLBACK TO的 Oracle 过程，你的工作比较简单：只要忽略掉SAVEPOINT以及ROLLBACK TO。如果你的 Oracle 过程是以不同的方法使用SAVEPOINT以及ROLLBACK TO，那么就要真正地动一番脑筋了。

43.13.2.2. EXECUTE

PL/pgSQL的EXECUTE与PL/SQL中的工作相似，但是必须要记住按照第 43.5.4 节所述地使用quote_literal以及quote_ident。EXECUTE 'SELECT * FROM $1';类型的结构将无法可靠地工作除非你使用这些函数。

43.13.2.3. 优化 PL/pgSQL 函数

PostgreSQL提供了两种函数创建修饰符来优化执行：“volatility”（对于给定的相同参数，函数是否总是返回相同的结果）以及“strictness” （如果任何参数为空，函数是否返回空）。详见CREATE FUNCTION参考页。

在利用这些优化属性时，你的CREATE FUNCTION语句应该看起来像这样：

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

43.13.3. 附录

这一节包含了一组 Oracle 兼容的instr函数代码，你可以用它来简化你的移植工作。

```
--
-- instr 函数模仿 Oracle 的对应函数
-- 语法: instr(string1, string2 [, n [, m]])
-- 其中 [] 表示可选参数。
--
-- 从第n个字符开始搜索string1，要求找到string2的第m次出现。
-- 如果n为负，则从后向前搜索，从string1的末尾开始的第abs(n)个字符开始。
-- 如果没有传n，假定它为1（从第1个字符开始搜索）。
-- 如果没有传m，假定它为1（找第1次出现）。
-- 在string1中返回string2的开始索引，如果没有找到string2则为0。
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
```

```
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

第 44 章 PL/Tcl - Tcl 过程语言

PL/Tcl 是一种用于PostgreSQL数据库系统的可载入过程语言， 它可以让Tcl 语言[1]被用来编写PostgreSQL函数。

44.1. 概述

PL/Tcl 提供了大部分函数编写者在 C 语言中能够获得的能力，虽然有一些限制，但是却额外提供了 Tcl 中强大的字符串处理库。

一种强制性的好限制是所有被执行的东西都处于 Tcl 解释器的安全上下文中。除了安全 Tcl 的有限的命令集合之外，只有几个通过 SPI 访问数据库的命令以及通过elog()产生消息的命令。PL/Tcl 没有提供访问数据库服务器内部或者在PostgreSQL服务器进程权限之下得到 OS-级访问的方法，而 C 函数是可以那样做的。因此，非特权数据库用户可以使用这种语言，它不会给予他们无限制的权利。

其他值得注意的实现限制是 Tcl 函数不能被用来创建新数据类型的输入/输出函数。

有时候我们想要编写不受安全 Tcl 限制的 Tcl 函数。例如，我们可能想要一个能发送电子邮件的 Tcl 函数。要处理这些情况，可以使用一种PL/Tcl的变体，它被称为PL/TclU（用于非可信 Tcl）。它其实是完全相同的一种语言，不过它使用了一个完整的 Tcl 解释器。如果使用了PL/TclU，它必须被安装为一种非可信的过程语言，这样只有数据库超级用户可以用它来创建函数。PL/TclU函数的编写者必须注意该函数不能被用来做其设计目的之外的事情，因为该函数能做一个作为数据库管理员登录的用户可以做的任何事情。

如果在安装过程的配置步骤中指定了 Tcl 支持，PL/Tcl以及PL/TclU调用处理器的共享对象代码会被自动编译并且被安装在PostgreSQL的库目录中。要在一个特定数据库中安装PL/Tcl或者PL/TclU，请使用CREATE EXTENSION命令，例如CREATE EXTENSION pltcl或者CREATE EXTENSION pltclu。

44.2. PL/Tcl 函数和参数

要用PL/Tcl语言创建函数，可使用标准的CREATE FUNCTION语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU的函数是一样的语法，只是语言被指定为pltclu。

函数的主体就是一个 Tcl 脚本。当函数被调用时，参数值会被以变量名$1 ... $n传递给该Tcl脚本。结果会以常见的方式通过一个return语句从 Tcl 脚本中返回。在一个过程中，Tcl代码的返回值会被忽略。

例如，一个返回两个整数值中较大值的函数可以定义为：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

注意子句STRICT，它让我们不用去操心空输入值：如果空值被传入，函数根本就不会被调用，而是自动地返回一个空结果。

[1] http://www.tcl.tk/

在非严格函数中，如果一个参数的实际值为空，对应的$n变量将被设置为一个空串。为了检测一个特定参数是否为空，可使用函数argisnull。例如，假设我们想要带有一个空参数和一个非空参数并且返回非空参数的tcl_max：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

如上所述，要从一个 PL/Tcl 函数返回空值，可执行return_null。不管函数是严格还是非严格都可以这样做。

组合类型参数会被作为 Tcl 数组传递给函数。该数组的元素名就是组合类型的属性值。如果被传入行的一个属性为空值，它不会出现在数组中。这里是一个例子：

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

PL/Tcl函数也能返回组合类型的结果。要返回组合类型结果，Tcl代码必须返回匹配预期结果类型的“列名/值”对的列表。任何从该列表中省略的列名将被返回为空，如果有预期之外的列名则会报出错误。这里是一个例子：

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

过程的输出参数以相同的方式返回，例如：

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
    return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

> **提示**
>
> 结果列表可以用Tcl的array get命令从想得到的元组的数组表示中造出。例如：

```
CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS $
$
    set 1(salary) [expr {$1(salary) + $2}]
    return [array get 1]
$$ LANGUAGE pltcl;
```

PL/Tcl函数能够返回集合。要返回集合，Tcl代码应该对每一个要返回的行调用一次return_next，在返回标量类型时传入合适的值或者在返回组合类型时传入“列名/值”堆的列表。这里是一个返回标量类型的例子：

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

这里是一个返回组合类型的例子：

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

44.3. PL/Tcl 中的数据值

提供给 PL/Tcl 函数代码的参数值是输入参数简单转换而成的文本形式（就像被SELECT语句显示的那样）。反过来，return和return_next命令将接受任何字符串，只要它是该函数声明的返回类型的可接受的输入格式，或者是组合结果类型的指定列的可接受输入格式。

44.4. PL/Tcl 中的全局数据

有时候需要在同一个函数的两次调用间保持某些全局数据或者在不同的函数之间共享全局数据。在 PL/Tcl 中这很容易做到，但是必须了解一些限制。

由于安全性原因，PL/Tcl 会为一个 SQL 角色建立一个单独的 Tcl 解释器来执行该角色调用的函数。这可以避免一个用户无意或者恶意地干涉另一个用户的 PL/Tcl 函数的行为。对任何“全局” Tcl 变量，每一个这样的解释器都有其自身的值。因此，当且仅当两个 PL/Tcl 函数由用一个 SQL 角色执行时，它们才能共享相同的全局变量。在使用单个会话执行多个 SQL 角色的代码（通过SECURITY DEFINER函数、使用SET ROLE等）的应用中，需要采取显式的步骤以保证 PL/Tcl 函数能共享数据。要这样做，需要确保要通信的函数都属于同一个用户，并且把它们标记为SECURITY DEFINER。当然，要小心这样的函数被滥用。

在一个会话中使用的所有 PL/TclU 函数都在同一个 Tcl 解释器中执行，这当然与用于 PL/Tcl 函数的解释器不同。因此，在 PL/TclU 函数之间会自动地共享全局数据。这并不是一种安全性风险，因为所有的 PL/TclU 函数都在同样的信任级别上执行，即都以数据库超级用户的级别执行。

为了保护 PL/Tcl 函数不会无意间彼此干扰，通过upvar命令可以建立一个对每个函数可用的全局数组。这个变量的全局名称是该函数的内部名称，并且本地名称为GD。推荐使用GD来保持一个函数的持久私有数据。只对你特别希望在多个函数之间共享的值使用常规的 Tcl 全局变量（注意GD数组只在一个特定的解释器中是全局的，因此它们不会绕过上文提到的安全性限制）。

下文的spi_execp例子中有一个使用GD的例子。

44.5. 从 PL/Tcl 访问数据库

下面的命令可以用来从 PL/Tcl 函数体中访问数据库：

spi_exec ?-count n? ?-array name? command ?loop-body?

执行一个以字符串给出的 SQL 命令。命令中错误将会导致错误发生。否则，spi_exec的返回值是被命令处理的行数（选择、插入、更新或者删除），如果命令是一条功能性语句则返回零。此外，如果命令是一条SELECT语句，被选中的列的值会被放在上文所述的 Tcl 变量中。

可选的-count值告诉spi_exec命令中要处理的最大行数。这种效果类似于用游标建立一个查询然后使用FETCH n。

如果命令是一条SELECT语句，结果列的值会被放在以列名命名的 Tcl 变量中。如果给出了-array选项，列值会被存储在所提及的关联数组的元素中，而列名则被用作数组索引。此外，结果中的当前行号（从零开始记）被存储在数组元素“.tupno”中，除非这个名字在结果中已经被用作一个列名。

如果命令是一条SELECT语句并且没有给出loop-body脚本，则只有结果的第一行被存储在 Tcl 变量或数组元素中。如果结果中有剩余的行，它们会被忽略。如果查询不返回行则不存储任何东西（这种情况可以通过spi_exec的结果检测到）。例如：

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

将把 Tcl 变量$cnt设置为pg_proc系统目录中的行数。

如果给出了可选的loop-body参数，它会是一个 Tcl 脚本，对查询结果中的每一行都要执行这个脚本（如果给出的查询不是SELECT则忽略loop-body）。在每次迭代前当前行的列值会被存储在 Tcl 变量或数组元素中。例如：

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

会对pg_class的每一行打印一段日志消息。这种特性工作起来类似于其他的 Tcl 循环结构。特别是continue和break的动作方式与在循环体中的通常方式相同。

如果一个查询结果的一列为空，为它准备的目标变量不会被建立，而是会被“unset”。

spi_prepare query typelist

为后面的执行准备并且保存一个查询计划。保存下来的计划将在当前会话的生命期内保持存在。

查询可以使用参数，也就是占位符。在计划真正被执行时将会为占位符提供值。在查询字符串中，可以用符号$1 ... $n引用参数。如果查询使用了参数，参数类型的名称必须以一个 Tcl 列表的形式给出（如果不使用参数，可以为typelist写一个空列表）。

从spi_prepare返回的值是一个查询 ID，在后续的spi_execp调用中需要用到这个 ID。例子可见spi_execp。

spi_execp ?-count n? ?-array name? ?-nulls string? queryid ?value-list? ?loop-body?

执行一个之前用spi_prepare准备的查询。queryid是spi_prepare返回的 ID。如果查询引用参数，则必须提供一个value-list。这是一个参数实际值的 Tcl 列表。这个列表必

须和之前传给spi_prepare的参数类型列表具有相同的长度。如果查询没有参数则可省略value-list。

-nulls的值可选，它是一个空格和'n'字符构成的串，它告诉spi_execp哪些参数是空值。如果给出这个值，它必须正好和value-list长度相等。如果没有给出这个值，所有的参数值都是空。

除了指定查询及其参数的方法，spi_execp和spi_exec很像。-count、-array以及loop-body选项是相同的，并且结果值也一样。

这里是一个使用预备计划的 PL/Tcl 函数的例子：

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # 第一次调用时准备保存的计划
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \
$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

我们需要在给spi_prepare的查询字符串里放上反斜线来确保$n标记会被原样传递给spi_prepare，并且不会被 Tcl 变量替换。

spi_lastoid

返回最后一次spi_exec或者spi_execp插入的行的 OID（如果该命令是一个单行INSERT且被修改表包含 OID。否则将会得到零）。

subtransaction command

command中包含的Tcl脚本会被在一个SQL子事务中执行。如果该脚本返回一个错误，那么整个子事务会在把错误返回到外围的Tcl代码之前就回滚。更多细节和例子请参考第 44.9 节

quote string

在给定的字符串中双写所有单引号和反斜线字符。这可以被用来引用字符串，以便它们能被安全地插入到传给spi_exec或者spi_prepare的 SQL 命令字符串中。例如，考虑这样的 SQL 命令字符串：

```
"SELECT '$val' AS ret"
```

这里的 Tcl 变量val实际上包含doesn't。这将会导致最终的命令串：

```
SELECT 'doesn't' AS ret
```

这种命令串会导致spi_exec或spi_prepare期间的解析错误。要正确地工作，提交的命令应该包含：

```
SELECT 'doesn''t' AS ret
```

在 PL/Tcl 中可以这样做：

"SELECT '[quote $val]' AS ret"

spi_execp的一个好处是你不必这样引用参数值，因为参数值不会被作为 SQL 命令串的一部分被解析。

elog level msg

发出一段日志或者错误消息。可能的级别是DEBUG、LOG、INFO、NOTICE、WARNING、ERROR以及FATAL。ERROR产生一个错误情况。如果周围的 Tcl 代码没有捕捉它，错误会传播到调用查询，导致当前事务或者子事务被中止。这实际上与 Tcl 的error命令相同。FATAL中止事务并且导致当前会话关闭（可能在 PL/Tcl 函数中没有很好的理由来使用这种错误级别，但是为了完整性还是提供了这种级别）。其他级别只产生不同优先级的消息。一个特定级别的消息是被报告给客户端、写入到服务器日志或者两者都做，是由配置变量log_min_messages和client_min_messages所控制。详见第 19 章和第 44.8 节

44.6. PL/Tcl 中的触发器函数

触发器函数也可以用 PL/Tcl 编写。PostgreSQL要求能作为触发器被调用的函数必须被声明为没有参数并且返回类型为trigger。

来自于触发器管理器的信息通过下列变量被传递给函数体：

$TG_name

CREATE TRIGGER语句中触发器的名字。

$TG_relid

导致触发器函数被调用的表的对象 ID。

$TG_table_name

导致触发器函数被调用的表的名字。

$TG_table_schema

导致触发器函数被调用的表所在的模式。

$TG_relatts

表列名的 Tcl 列表，前面放上一个空列表元素。因此用Tcl的lsearch命令在该列表中查找一个列名返回的元素编号会从 1 开始（对于第一列），这和PostgreSQL中的自定义编号是同样的方式（空列表元数也出现在被删除的列的位置上，这样其右边的列的属性编号才是正确的）。

$TG_when

可以为BEFORE、AFTER或者INSTEAD OF，具体的选择取决于触发器事件的类型。

$TG_level

可以为ROW或者STATEMENT，取决于触发器事件的类型。

$TG_op

可以为INSERT、UPDATE、DELETE或者TRUNCATE，取决于触发器事件的类型。

$NEW

对于INSERT或者UPDATE动作是一个包含着新表行值的关联数组，对于DELETE为空。该数组以列名为索引。为空的列不会出现在数组中。对于语句级触发器这个变量不会被设置。

$OLD

对于UPDATE或者DELETE动作是一个包含着新表行值的关联数组，对于INSERT为空。该数组以列名为索引。为空的列不会出现在数组中。对于语句级触发器这个变量不会被设置。

$args

在CREATE TRIGGER语句中对过程给出的参数的 Tcl 列表。在过程体中也可以用$1 ... $n来访问这些参数。

一个触发器函数的返回值可以是字符串OK或者SKIP，或者是一个“列名/值”对的列表。如果返回值是OK，引发该触发器的操作（INSERT/UPDATE/DELETE）将正常继续下去。SKIP告诉触发器管理器悄悄地抑制对这一行的该操作。如果返回一个列表，它告诉PL/Tcl返回一个被修改的行给触发器管理器，这个被修改的行的内容由列表中的列名和值指定。该列表中没有提到的任何列会被设置为空。返回被修改行只对行级BEFORE INSERT或UPDATE触发器有意义，对它们来说这个被修改的行将被插入而不是插入$NEW中给出的行。返回被修改行还对行级INSTEAD OF INSERT或UPDATE触发器有意义，其中被返回的行被用作INSERT RETURNING或UPDATE RETURNING子句的源数据。在行级BEFORE DELETE或INSTEAD OF DELETE触发器中，返回一个被修改行的效果和返回OK的效果相同，即操作继续。对所有其他类型的触发器来说，触发器返回值会被忽略。

> **提示**
>
> 结果列表可以用Tcl的array get命令从被修改的元组的数组表示中造出。

这里有一个触发器函数的例子，它用一个表中的整数值来跟踪在行上被执行的更新数。对于被插入的新行，该值被初始化为 0 并且之后在每一次更新操作时被加一。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

注意触发器函数本身不知道列名，列名由触发器参数提供。这让触发器函数可以被重用于不同的表。

44.7. PL/Tcl 中的事件触发器函数

事件触发器函数也可以用 PL/Tcl 编写。PostgreSQL要求能作为事件触发器被调用的函数必须被声明为没有参数并且返回类型为event_trigger。

来自于触发器管理器的信息通过下列变量被传递给函数体：

$TG_event

触发器为其引发的事件名。

$TG_tag

触发器为其引发的命令标签。

触发器函数的返回值被忽略。

这里是一个事件触发器函数的小例子，它在所支持的命令每次执行时简单地产生一个NOTICE消息：

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE FUNCTION
 tclsnitch();
```

44.8. PL/Tcl 中的错误处理

PL/Tcl 函数中的 Tcl 代码或者从 PL/Tcl 函数中调用的代码可以抛出一个错误，错误可以由执行某些非法操作产生或者通过使用 Tcl error命令或者 PL/Tcl 的elog命令产生。Tcl中可以使用 Tcl catch命令捕获这类错误。如果一个错误没有被捕捉但是被允许传播到该PL/Tcl函数执行的顶层，它会在该函数的调用查询中被报告为一个SQL错误。

相反，在 PL/Tcl 的spi_exec、spi_prepare以及spi_execp命令中发生的SQL错误会被报告为 Tcl 错误，因此它们也可以被 Tcl 的catch命令捕获（这些PL/Tcl命令中的每一个都在一个子事务中运行它的SQL操作，该子事务在错误时会被回滚，这样任何部分完成的操作也会被自动清除）。同样地，如果一个错误被传播到顶层而没有被捕获，它会转变成SQL错误。

Tcl 提供了一个errorCode变量，它表示有关于一个错误的附加信息，它的格式易于 Tcl 程序解释。该变量的内容符合 Tcl 列表格式，第一个词标识报告该错误的子系统或者库，之后的内容则留给子系统或者库来填充。对于 PL/Tcl 命令报告的数据库错误，第一个词是POSTGRES，第二个词是 PostgreSQL 的版本号，剩下的部分是域名称/域值构成的对，它们提供有关该错误的详细信息。域SQLSTATE、condition以及message总是会被提供（前两个表示附录 A中所示的错误代码和情况名称）。可能出现的域包括 detail、hint、context、schema、table、column、 datatype、constraint、 statement、cursor_position、filename、lineno以及 funcname。

使用 PL/Tcl 的errorCode信息的一种便捷方式是把它载入到一个数组中，这样域名称就变成了数组下标。这样做的代码看起来像这样

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # deal with missing table
        } else {
            # deal with some other type of SQL error
        }
    }
```

```
}
```

（双冒号显式地指定errorCode是一个全局变量）。

44.9. PL/Tcl中的显式子事务

从第 44.8 节介绍的数据库访问导致的错误中恢复可能导致一种不可取的情况，其中一些操作在它们中的一个失败前成功完成，并且在从错误中恢复过来后数据还处于一种不一致的状态。PL/Tcl以显式子事务的形式为这类问题提供了一个解决方案。

考虑一个在两个账户间实现转账的函数：

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name
 = 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name
 = 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

如果第二个UPDATE语句导致一个异常，这个函数将记下该失败，但是第一个UPDATE的结果将被提交。换句话说，资金将从Joe的账户中被取走，但不会被转到Mary的账户。这是因为每个spi_exec都是一个单独的子事务，并且那些子事务中只有一个被回滚。

为了处理这类情况，你可以把多个数据库操作包裹在一个显式子事务中，它将作为一个整体成功完成或者回滚。PL/Tcl提供了一个subtransaction命令来做这件事情。我们可以把我们的函数写成：

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
            spi_exec "UPDATE accounts SET balance = balance - 100 WHERE
 account_name = 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100 WHERE
 account_name = 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

注意，为了实现这个目的仍要求使用catch。否则错误将传播到该函数的顶层，导致想要对operations表的插入被阻止。subtransaction命令不会捕捉错误，它仅确保报告错误时在其范围内执行的所有数据库操作将被一起回滚。

一个显式子事务的回滚发生在包含它的Tcl代码报告任何错误时，而不仅仅是数据库访问导致的错误。因此一个subtransaction命令中发生的常规Tcl异常也将导致该子事务被回滚。不过，无错误退出到包含子事务的Tcl代码外面（例如，由于return）不会导致回滚。

44.10. 事务管理

在从顶层调用的过程中或者从顶层调用的匿名代码块（DO命令）中，可以控制事务。要提交当前的事务，可调用commit。要回滚当前事务，可调用rollback（注意不能通过spi_exec或类似的函数运行SQL命令COMMIT或者ROLLBACK。这类工作必须用这些函数完成）。在事务结束以后，一个新的事务会自动开始，因此没有独立的函数用来开始新事务。

这里是一个例子：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
for {set i 0} {$i < 10} {incr i} {
    spi_exec "INSERT INTO test1 (a) VALUES ($i)"
    if {$i % 2 == 0} {
        commit
    } else {
        rollback
    }
}
$$;

CALL transaction_test1();
```

当一个显式的子事务处于活跃状态时，事务不能被结束。

44.11. PL/Tcl配置

这一节列举影响PL/Tcl的配置参数。

pltcl.start_proc (string)

如果被设置为一个非空字符串，这个参数指定一个无参数PL/Tcl函数的名称（可能是方案限定的），只要为PL/Tcl创建一个新的Tcl解释器，就会执行这个函数。这样一个函数可以执行针对会话的初始化，例如载入额外的Tcl代码。当一个PL/Tcl在一个数据库会话中被第一次执行时，或者由于一个PL/Tcl函数被一个新的SQL角色调用而必须创建一个额外的解释器时，一个新的Tcl解释器会被创建。

被引用的函数必须用pltcl语言编写，并且不能被标记为SECURITY DEFINER（这些限制确保它运行在它应该要初始化的解释器中）。当前用户也必须有权限调用它。

如果该函数带着一个错误失败，它将中止导致新解释器创建的函数并且把错误传播到调用查询，进而导致当前事务或子事务被中止。在Tcl中已完成的任何动作将不会被撤销，不过，那个解释器将不会被再次使用。如果该语言被再次使用，则初始化将在一个全新的Tcl解释器中被再次尝试。

只有超级用户能够更改这个设置。尽管这个设置能在会话中更改，但这种更改将不会影响已经被创建的Tcl解释器。

pltclu.start_proc (string)

这个参数与pltcl.start_proc几乎一模一样，只不过它适用于PL/TclU。被引用的函数必须用pltclu语言编写。

44.12. Tcl 过程名

在PostgreSQL，同一个函数名可以被用于不同的函数定义，只要它们的参数个数或者类型不同。不过，Tcl 要求所有过程名必须能区分。PL/Tcl 通过让内部 Tcl 过程名称包含该函数

在系统表pg_proc中的对象 ID 作为名称的一部分来解决这样的限制。因此，具有相同名称和不同参数类型的PostgreSQL函数也将是不同的 Tcl 过程。这对 PL/Tcl 程序员来说通常不需要关心，但是在调试时可见。

第 45 章 PL/Perl - Perl 过程语言

PL/Perl 是一种可载入过程语言，它允许我们用 Perl 编程语言[1]编写 PostgreSQL函数。

使用 PL/Perl 的主要优势它允许在存储函数中使用大量 Perl 的 “串整理”操作符和函数。使用 Perl 解析复杂 串比使用 PL/pgSQL 中提供的串函数和控制结构要更容易。

要在一个特定数据库中安装 PL/Perl，使用 CREATE EXTENSION plperl。

> **提示**
>
> 如果把语言安装在template1中，所有后续创建的数据库 都将自动地安装有该语言。

> **注意**
>
> 使用源码包安装的用户必须在安装过程中开启对 PL/Perl 的编译（ 更多信息请参考第 16 章。使用二进制包 安装的用户可能会在独立的子包中找到 PL/Perl。

45.1. PL/Perl 函数和参数

要用 PL/Perl 语言创建一个函数，可使用标准的 CREATE FUNCTION语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Perl 函数体
$$ LANGUAGE plperl;
```

函数的主体就是普通的 Perl 代码。事实上，PL/Perl 的粘合代码会把它 包裹在一个 Perl 子程序中。一个 PL/Perl 函数会在一种标量上下文中 被调用，因此它无法返回列表。如下文所述，可以通过返回引用来返回 非标量值（数组、记录和集合）。

在一个PL/Perl过程中，任何从Perl代码返回的值都会被忽略。

PL/Perl 也支持用DO语句调用的匿名代码块：

```
DO $$
    # PL/Perl 代码
$$ LANGUAGE plperl;
```

一个匿名代码块没有参数，并且它返回的任何值都会被抛弃。否则 其行为就像一个函数。

> **注意**
>
> 在 Perl 中使用命名嵌套子程序是有危险的，特别是当它们在作用域内 引用局部变量时。因为 PL/Perl 函数被包装成一个子程序，任何放在 其中的命名子程序都会被嵌套。总之，创建通过 coderef 调用的匿名 子程序要安全得多。更多信息可见 perldiag手册页 中的Variable "%s" will not stay shared以

[1] http://www.perl.org

及 Variable "%s" is not available，或者在互联网上 搜索“perl nested named subroutine”。

CREATE FUNCTION命令的语法要求函数 体被写作一个字符串常量。通常对字符串常量使用美元引用（见 第 4.1.2.4 节最方便。如果选择使用 转义字符串语法E''，必须双写任何在函数体中使用的单引号 （''）和反斜线（\\）（见 第 4.1.2.1 节。

参数和结果的处理和在任何其他 Perl 子程序中一样：参数被传递到 @_中，并且结果值用return 返回或者把函数中计算的最后一个表达式作为结果值。

例如，一个返回两个整数值中较大值的函数可以定义为：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

注意

参数将被从数据库的编码转换到 PL/Perl 中使用的 UTF-8，返回时再从 UTF-8 转回到数据库编码。

如果一个 SQL 空值被传给一个函数，在 Perl 中该参数值将呈现为“undefined”。上述函数定义对于 空输入的行为不太好（实际上，它会把它们当作零）。我们可以为函数 定义增加STRICT让PostgreSQL 干得更合理：如果空值被传入，函数将根本不会被调用，而只是自动 返回一个空结果。另外一种方式，我们可以在函数体中检查未定义的 输入。例如，假设我们想让带有一个空参数或者一个非空参数的 perl_max返回非空参数而不是空值：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

如上所述，要从一个 PL/Perl 函数返回一个 SQL 空值，就返回一个未定义值。 不管函数是严格的还是非严格的都可以这样做。

一个非引用的函数参数中的任何东西都是一个串，是相关数据类型的标准 PostgreSQL外部文本表达。在普通 数字或文本类型的情况下，Perl 将会做正确的事情并且程序员通常不需要 操心。不过，在其他情况下将需要被转换成在 Perl 中更可用的形式。例如，decode_bytea函数可以被用来把类型 bytea的参数转换成未转义的二进制形式。

类似地，回传给PostgreSQL的值必须 是外部文本表达格式。例如，encode_bytea 函数可以被用来转义二进制数据得到类型bytea的返回值。

Perl 可以把PostgreSQL数组返回为对 Perl 数组的引用。这里有一个例子：

```
CREATE OR REPLACE function returns_array()
```

```
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl 把PostgreSQL数组作为被 bless 过的 PostgreSQL::InServer::ARRAY对象传递。这个对象可以被当作 一个数组引用或者一个串，允许为了向后兼容性与为 9.1 以下版本的PostgreSQL编写的 Perl 代码一起运行。 例如：

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # as an array reference
    for (@$arg) {
        $result .= $_;
    }

    # also works as a string
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **注意**
>
> 多维数组被以一种对每一个 Perl 程序员都公认的方法表示为对较低维引用数组 的引用。

组合类型参数被作为哈希的引用传递给函数。哈希的键是组合类型的 属性名。这里是一个例子：

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

PL/Perl 函数可以使用相同的方法返回组合类型：返回具有所要求属性的 哈希的引用。例如：

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);
```

```
CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

任何所要求结果数据类型中不存在于哈希中的列将被返回为空值。

类似的，过程的输出参数也可以被返回为哈希引用：

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

PL/Perl 函数也能返回标量或者组合类型集合。为了加速启动并且避免在 内存中让整个结果集排队等候，我们通常希望能一次返回一行。可以按 下文所说的用return_next来这样做。注意在 最后一次return_next后，必须放上 return或者return undef（后者更好）。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

对于小结果集，可以返回到一个数组的引用，该数组分别包含用于 简单类型、数组类型和组合类型的标量、数组引用或者哈希引用。 这里有一些简单的例子把整个结果集作为数组引用返回：

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;
```

```
SELECT * FROM perl_set();
```

如果你想要对你的代码使用strict编译指示，有几种选项可用。 对于临时的全局使用，你可以SET plperl.use_strict为真。这将影响后续 PL/Perl函数的编译，但是对当前会话中已经编译过的 函数没有影响。对于持久的全局使用，可以在 postgresql.conf文件中设置plperl.use_strict为真。

对于在特定函数中的持久使用，可以简单地把

```
use strict;
```

放在函数体的顶层。

如果 Perl 版本是 5.10.0 或者更高，也可以使用 feature编译指示。

45.2. PL/Perl 中的数据值

提供给 PL/Perl 函数代码的参数值是被转换成文本形式的输入参数（就像它们 被SELECT语句显示的那样）。反过来， return和return_next命令 将接受任何该函数返回类型可接受的输入格式的串。

45.3. 内建函数

45.3.1. 从 PL/Perl 访问数据库

可以通过下列函数从 Perl 函数中访问数据库本身：

spi_exec_query(query [, max-rows])

spi_exec_query执行一个 SQL 命令并且以哈希引用数组 的引用的形式返回整个行集。只有在知道结果集相对较小时才 应该使用这个命令。这里是一个带有可选最大行数的查询（ SELECT命令）的例子：

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

这会从表my_table.返回最多 5 行。如果 my_table有一个列my_column， 可以从结果的$i行得到值：

```
$foo = $rv->{rows}[$i]->{my_column};
```

可以这样访问从一个SELECT查询返回 的总行数：

```
$nrows = $rv->{processed}
```

这里是使用不同命令类型的一个例子：

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

你可以这样访问命令状态（例如SPI_OK_INSERT）：

```
$res = $rv->{status};
```

要得到受影响的行数：

```
$nrows = $rv->{processed};
```

这里是一个完整的例子：

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

```
spi_query(command)
spi_fetchrow(cursor)
spi_cursor_close(cursor)
```

spi_query和spi_fetchrow 结对用于可能比较大的行集合，或者用于希望在行到达时返回的情况。 spi_fetchrow只和 spi_query一起工作。下面的例子展示了如何使用 它们：

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
```

```
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

通常，spi_fetchrow应该重复执行直到它返回 undef（表示没有更多行要读取）。当spi_fetchrow返回undef时， spi_query返回的游标会自动被释放。如果不 想读取所有的行，可以调用spi_cursor_close来 释放游标。如果没有这样做会导致内存泄露。

```
spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)
```

spi_prepare、spi_query_prepared、 spi_exec_prepared和spi_freeplan 为预备查询实现了相同的功能。spi_prepare接受一个查询 字符串，其中包括编好号的参数占位符（$1、$2 等）以及参数类型的字符串 列表：

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

一旦通过调用spi_prepare准备好一个查询计划，就可以在 spi_exec_prepared（返回的结果和 spi_exec_query相同）或者spi_query_prepared （返回的结果和spi_query一样，后面会被传给 spi_fetchrow）中用该计划来取代字符串查询。spi_exec_prepared可选的第二个参数是属性的哈希引用， 当前唯一支持的属性是limit，它限定了一个查询返回的最大 行数。

预备查询的有点是可以把一个准备好的计划用于多次查询执行。不再需要该计划后， 可以用spi_freeplan释放它：

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
```

```
 2005-12-10 | 2005-12-11 | 2005-12-12
```

注意spi_prepare中的参数下标通过 $1、$2、$3 等定义， 这样避免了用双引号来声明查询串（容易导致难以捕捉的缺陷）。

另一个展示spi_exec_prepared中可选参数用法的例子：

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

spi_commit()
spi_rollback()

提交或者回滚当前事务。只能在从顶层调用的过程或者匿名代码块（DO命令）中调用这个函数（注意不能通过spi_exec_query或者类似的函数运行SQL命令COMMIT或者ROLLBACK。这样的工作只能使用这些函数完成）。在一个事务结束后，一个新的事务会自动开始，因此没有单独的函数来开始新事务。

这里是一个例子：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
```

```
    }
}
$$;

CALL transaction_test1();
```

45.3.2. PL/Perl 中的工具函数

elog(level, msg)

发出一个日志或者错误消息。可用的级别有 DEBUG、LOG、INFO、 NOTICE、WARNING以及ERROR。 ERROR产生一种错误情况，如果它没有被周围的 Perl 代码 捕获，错误会传播到调用查询中，导致当前事务或者子事务被中止。这实际 上和 Perl 的die 命令相同。其他级别只产生不同优先级的消息。 特定优先级的消息是被报告给客户端、写到服务器日志或者两者都做由 配置变量log_min_messages和 client_min_messages控制。详见 第 19 章

quote_literal(string)

返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串中的字符串。 嵌入的引号和反斜线会被正确地双写。注意对 undef 输入quote_literal 会返回undef。如果参数可能是 undef，quote_nullable通常更合适。

quote_nullable(string)

返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串中的字符串。 或者在参数为 undef 时，返回未引用的串 "NULL"。 嵌入的引号和反斜线会被正确地双写。

quote_ident(string)

返回给定字符串的被适当引用后的形式，这种形式能被用作 SQL 语句字符串 中的标识符。只有在必要时才增加引号（即，如果串包含非标识符字符或者是 大小写折叠的）。嵌入的引号会被正确地双写。

decode_bytea(string)

返回由给定串的内容（应该用bytea编码）表示的未转义二进制数据。

encode_bytea(string)

返回给定串的二进制数据内容的bytea编码形式。

encode_array_literal(array)
encode_array_literal(array, delimiter)

把被引用的数组的内容返回成数组文字格式（见第 8.15.2 节 的一个串。如果它不是一个数组的引用，则不加修改地返回参数值。如果没有指定 定界符或者定界符为undef，则默认把", "用作数组文字的元素 之间的定界符。

encode_typed_literal(value, typename)

把一个 Perl 变量转换为由第二个参数传入的数据类型的值，并且返回该值 的字符串表达。它能正确地处理嵌套数组和组合类型的值。

encode_array_constructor(array)

把被引用数组的内容返回为数组构造器格式（ 第 4.2.12 节的一个串。其中的个体值用 quote_nullable引用。如果参数不是一个数组引用，则 返回用quote_nullable引用的该参数值。

looks_like_number(string)

如果给定串的内容对于 Perl 看起来像是数字则返回真，否则返回假。如果 参数是 undef 则返回 undef。前导和结尾的空格会被忽略。 Inf和Infinity被视作数字。

is_array_ref(argument)

如果给定参数可以被当作一个数组引用对待则返回真值，即该参数的定义为 ARRAY或者PostgreSQL::InServer::ARRAY时返回 真。否则返回假。

45.4. PL/Perl 中的全局值

可以在函数调用之间或者当前会话的生命期中用全局哈希 %_SHARED来存储数据，包括代码引用。

这是共享数据的一个简单例子：

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

这是一个使用代码引用的稍微复杂一点的例子：

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* 初始化函数 */

/* 设置一个使用引用函数的函数 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

（你可以把上面的代码用一行 return $_SHARED{myquote}->($_[0]);替换， 代价是牺牲了可读性）。

处于安全原因，PL/Perl 一个 SQL 角色独立的 Perl 解释器中执行该角色调用 的任何一个函数。这可以避免一个用户无意或者恶意地干涉另一个用户的 PL/Perl 函数的行为。每一个这

样的解释器都具有其自身的 %_SHARED变量值和其他全局状态。因此，只有当 两个 PL/Perl 函数是由同一个 SQL 角色执行时，它们才能共享同一个 %_SHARED值。在使用单个会话执行多个 SQL 角色 的代码（通过SECURITY DEFINER函数、使用 SET ROLE等）的应用中，需要采取显式的步骤以保证 PL/Perl 函数能够通过%_SHARED共享数据。要这样做，需要 确保要通信的函数都属于同一个用户，并且把它们标记为 SECURITY DEFINER。当然，要小心这样的函数被滥用。

45.5. 可信的和不可信的 PL/Perl

通常，PL/Perl 被作为一种“可信的”编程语言安装，其名称 为plperl。在这种设置下，为了保持安全性禁用了某些 Perl 操作。一般来说，被限制的操作是那些与环境交互的操作。它们 包括文件处理操作、require以及 use（外部模块）。没有办法像 C 函数那样访问 数据库服务器进程的内部或者用服务器进程的权限得到 OS 级别的访问。 因此，任何没有特权的数据库用户也被允许使用这种语言。

下面例子中的函数将无法工作，因为出于安全原因不允许它做文件操作：

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

这个函数的创建会失败，因为验证器会捕捉到它使用了禁用的操作。

有些时候需要编写不受限制的 Perl 函数。例如，我们可能想要一个能发送 电子邮件的 Perl 函数。要处理这些情况，可以把 PL/Perl 安装成一种 “不可信的”语言（通常被称作 PL/PerlU）。 在这种情况下整个 Perl 语言的特性都可以使用。在安装语言时，用语言 名称plperlu将会选择不可信的 PL/Perl 变体。

PL/PerlU函数的编写者必须注意该函数不能被用来做 其设计目的之外的事情，因为该函数能做一个作为数据库管理员登录的用户 可以做的任何事情。注意数据库系统只允许数据库超级用户用不可信语言 创建函数。

如果上述函数是一个超级用户用语言plperlu创建的，则可以 执行成功。

以和plperl语言同样的方式，可以用plperlu 编写 Perl 中的匿名代码块，这样的代码块能够使用受限的操作，不过调用 者必须是超级用户。

> **注意**
>
> 虽然对每个 SQL 角色会在一个独立的 Perl 解释器中运行 PL/Perl函数，但是在一个给定会话中执行的所有 PL/PerlU函数都运行在一个 Perl 解释器中（与用于 任何PL/Perl函数的解释器不同）。这允许 PL/PerlU函数自由地共享数据，但是 PL/Perl和PL/PerlU函数之间不会 发生任何交流。

> **注意**
>
> Perl 不支持一个进程中的多个解释器，除非编译它时使用了合适的标志，即usemultiplicity或者useithreads（ usemultiplicity会更好，除非你确实需要使用线程。更多细节， 请见perlembed手册页）。 如果PL/Perl用的是一份没有这样编译的 Perl 拷贝，那么 在每个会话中只能有一个 Perl 解释器，

> 并且因此任一会话只能要么执行 PL/PerlU函数，要么执行同一个 SQL 角色调用的 PL/Perl函数。

45.6. PL/Perl 触发器

PL/Perl 可以被用来编写触发器函数。在触发器函数中，哈希引用 $_TD包含有关当前触发器事件的信息。 $_TD是一个全局变量，对触发器的每一次调用它都会 得到一个独立的本地值。$_TD哈希引用的域有：

$_TD->{new}{foo}

列foo的NEW值

$_TD->{old}{foo}

列foo的OLD值

$_TD->{name}

要被调用的触发器的名称

$_TD->{event}

触发器事件：INSERT、UPDATE、 DELETE、TRUNCATE或者UNKNOWN

$_TD->{when}

什么时候调用触发器：BEFORE、 AFTER、INSTEAD OF或者 UNKNOWN

$_TD->{level}

触发器级别：ROW、STATEMENT或者UNKNOWN

$_TD->{relid}

触发器定义在其上的表的 OID

$_TD->{table_name}

触发器定义在其上的表的名称

$_TD->{relname}

触发器定义在其上的表的名称。这已经被废弃，并且可能会在 未来的发布中被移除。请使用 $_TD->{table_name}。

$_TD->{table_schema}

触发器定义在其上的表所在的模式的名称

$_TD->{argc}

触发器函数的参数数目

@{$_TD->{args}}

触发器函数的参数。如果$_TD->{argc}为 0 则不存在

行级触发器可以返回下列之一：

return;

执行操作

"SKIP"

不执行操作

"MODIFY"

指示触发器函数修改了NEW行

这里是一个触发器函数的例子，展示上文所说的一些东西：

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # skip INSERT/UPDATE command
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # 修改行并且执行 INSERT/UPDATE 命令
    } else {
        return;           # 执行 INSERT/UPDATE 命令
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE FUNCTION valid_id();
```

45.7. PL/Perl 事件触发器

PL/Perl 可以被用来编写事件触发器函数。在事件触发器函数中，哈希引用 $_TD包含有关当前触发器事件的信息。 $_TD是一个全局变量，对触发器的每一次调用它都会 得到一个独立的本地值。$_TD哈希引用的域有：

$_TD->{event}

触发器为其触发的事件名称。

$_TD->{tag}

触发器为其触发的命令标签。

触发器函数的返回值会被忽略。

这里是一个事件触发器函数的例子，展示了上文所说的一些东西：

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
```

```
ON ddl_command_start
EXECUTE FUNCTION perlsnitch();
```

45.8. PL/Perl 下面的东西

45.8.1. 配置

这一节列出了影响PL/Perl的配置参数。

plperl.on_init (string)

指定当第一次初始化一个 Perl 解释器时要执行的 Perl 代码，这会在 具体用于plperl或plperlu之前做完。当 这段代码被执行时 SPI 函数不可用。如果该代码由于错误失败，它 将中止解释器的初始化并且把错误传播到调用查询，最终导致当前 事务或者子事务被中止。

该 Perl 代码被限制为一个单一的字符串。更长的代码可以放在一个 模块中，然后由on_init字符串载入。例子：

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

任何被plperl.on_init载入的模块（不管是直接还是间接）都 可以被plperl使用。这可能会导致安全性风险。要看哪些模块 已经被载入，可以使用：

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

如果 plperl 库被包括在shared_preload_libraries 中，那么初始化将发生在 postmaster 中，在这种情况下要特别地考虑对 postmaster 带来的不稳定风险。使用这种特性的主要原因是， plperl.on_init载入的 Perl 模块只需要在 postmaster 开始时被载入，并且在数据库会话中不需要任何工作就立刻可用。不过，要记住这 只免除了一个数据库会话中使用的第一个 Perl 解释器的负载 — 不管 是 PL/PerlU 还是用于第一个 SQL 角色调用 PL/Perl 函数的 PL/Perl。在一个 数据库会话中创建的任何额外的 Perl 解释器将不得不重新执行 plperl.on_init。还有，在 Windows 上无论从什么里面进行 预先载入，都不会有这种节约，因为在 postmaster 进程中创建的 Perl 解释器 不会传播到子进程中。

这个参数只能在postgresql.conf文件或者服务器命令中设置。

plperl.on_plperl_init (string)
plperl.on_plperlu_init (string)

这些参数分别指定当为plperl或plperlu专门准备好 一个 Perl 解释器时要执行的 Perl 代码。当一个 PL/Perl 或者 PL/PerlU 函数 第一次在一个数据库会话中被执行时会发生这种动作，或者由于调用其他语言 或者新的 SQL 角色调用 PL/Perl 函数导致创建额外的解释器时也会发生这种 动作。这些初始化跟随着plperl.on_init所作的初始化。当这段 代码被执行时，SPI 函数不可用。plperl.on_plperl_init中的 Perl 代码在“锁闭”解释器之后被执行，因此它只能执行可信的操作。

如果该代码由于错误失败，它将中止初始化并且把错误传播到调用查询， 最终导致当前事务或者子事务被中止。在 Perl 中已经完成的任何动作都 不会被撤销。不过，该解释器将不能被再次使用。如果再次使用该语言， 将在一个新鲜的 Perl 解释器中再次尝试初始化。

只有超级用户能够更改这些设置。尽管这些设置可以在会话中被修改， 但是这类更改将不会影响已经被用来执行函数的 Perl 解释器。

plperl.use_strict (boolean)

如果被设置为真，则后续的 PL/Perl 函数编译将会启用 strict编译指示。这个参数不影响当前会话中已编译的函数。

45.8.2. 限制和缺失的特性

PL/Perl 中目前缺少下列特性，但是欢迎大家对此作出贡献。

- PL/Perl 函数不能直接调用彼此。
- SPI 还没有被完全实现。
- 如果正在使用spi_exec_query取一个非常大的数据集， 你应该注意它们都会进入到内存中。可以按先前所述，通过使用 spi_query/spi_fetchrow来避免发生 这类情况。

  如果一个集合返回函数通过return把一个大型的行集合 返回给 PostgreSQL，同样会发生这种情况。同样如前所述，可以为每一个 要返回的行使用return_next来避免这种问题。
- 当会话正常结束（而不是由于致命错误结束）时，任何已经定义的 END块将被执行。当前不会执行其他动作。特别地， 此时文件句柄不会被自动刷写并且对象不会被自动销毁。

第 46 章 PL/Python - Python 过程语言

PL/Python过程语言允许用Python 语言[1]编写PostgreSQL函数。

要在一个特定数据库中安装 PL/Python，请使用CREATE EXTENSION plpythonu（另见第 46.1 节。

> **提示**
>
> 如果把一种语言安装在template1中，所有后续创建的数据库都会自动安装该语言。

PL/Python 只是一种“不可信”语言，这意味着它没有提供任何方法来限制用户在其中的所作所为，并且因此被命名为plpythonu。如果未来开发出在 Python 中的安全执行机制，可能会出现一种可信的变体plpython。不可信 PL/Python 中函数的编写者必须注意该函数不能被用来做任何它不应该做的事情，因为它将能做以数据库管理员用户登录能做的事情。只有超级用户能够创建plpythonu等不可信语言中的函数。

> **注意**
>
> 源码包的用户必须在安装过程中特别地启用 PL/Python 的编译（更多信息请参考安装指导）。二进制包的用户可以在一个单独的子包中找到 PL/Python。

46.1. Python 2 vs. Python 3

PL/Python 同时支持 Python 2 和 Python 3 两种语言变体（PostgreSQL 安装指导可能包含了所支持的 Python 次版本的更精确的信息）。因为 Python 2 和 Python 3 语言变体在某些重要的方面并不兼容，PL/Python 使用了下列命名和转换模式以避免混淆它们：

- 名为plpython2u的 PostgreSQL 语言实现了基于 Python 2 语言变体的 PL/Python。
- 名为plpython3u的 PostgreSQL 语言实现了基于 Python 3 语言变体的 PL/Python。
- 名为plpythonu的语言实现了基于默认 Python语言变体（当前是 Python 2）的 PL/Python（这种默认与任何本地 Python 安装所认为的“默认”无关，例如/usr/bin/python）。在遥远的未来，PostgreSQL 的发布中可能会把这种默认语言变体改成 Python 3，这取决于 Python 社区迁移到 Python 3 的进度。

这种模式类似于PEP 394[2]中关于python命令的命名和转换的推荐。

到底是 Python 2 还是 Python 3 的 PL/Python 可用或是两者都可用，取决于编译配置或者被安装的包。

> **提示**
>
> 如果是编译安装，则取决于在安装期间找到的 Python 版本或者用PYTHON环境变量显式设置的版本，见第 16.4 节要在一个安装中让两种变体的 PL/Python 都可用，源代码树必须被配置和编译两次。

[1] http://www.python.org
[2] https://www.python.org/dev/peps/pep-0394/

这产生了下列的使用和迁移策略：

- 现有用户以及对 Python 3 不感兴趣的用户使用plpythonu语言并且在可预见的未来不必做出任何改变。我们推荐通过迁移到 Python 2.6/2.7 逐步地让代码“经得起未来的考验”以简化最终迁移到 Python 3 的工作。

  实际上，很多 PL/Python 函数可以用很少或者不做修改就迁移到 Python 3。

- 对于代码严重依赖于 Python 2 并且不打算做改变的用户可以使用plpython2u语言。这将在很长时间内都有效，直到 PostgreSQL 完全删除掉对 Python 2 的支持。

- 想投入 Python 3 的怀抱的用户可以使用plpython3u语言，在当前的标准下这将一直有效。在遥远的未来，当 Python 3 成为默认以后，出于审美的原因，“3”可能会被移除。

- 想要构建一个只有 Python 3 的操作系统环境的冒险者们，可以更改pg_pltemplate的内容让plpythonu等价于plpython3u，记住这将会让他们的安装与世界的其他大部分东西都不兼容。

有关移植到 Python 3 的更多信息还可见文档What's New In Python 3.0[3]。

不允许在同一个会话中使用基于 Python 2 的 PL/Python 以及基于 Python 3 的 PL/Python，因为动态模块中的符号会冲突，这会导致 PostgreSQL 服务器进程的崩溃。在一个会话中有一个检查来阻止混淆 Python 的主版本，如果检测到不匹配会中断会话。不过，可以在同一个数据库中对不同的会话使用两种 PL/Python 变体。

46.2. PL/Python 函数

PL/Python 中的函数通过标准的CREATE FUNCTION语法声明：

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python 函数体
$$ LANGUAGE plpythonu;
```

函数体就是一个 Python 脚本。当函数被调用时，它的参数被当做列表args的元素传递，命名参数也被作为普通变量传递给 Python 脚本。使用命名参数通常可读性更好。Python 代码会以通常的方式返回结果，即使用return或者yield（在结果集合语句的情况中）。如果没有提供一个返回值，Python 会返回默认的None。PL/Python会把 Python 的None翻译成 SQL 空值。在一个过程中，Python代码的结果必须是None（通常实现为结束过程时不写return语句或者使用不带参数的return），否则将会发生错误。

例如，一个返回两个整数中较大的整数的函数可以定义为：

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

作为该函数定义给出的 Python 代码会被转换成一个 Python 函数。例如上面的代码会得到：

[3] https://docs.python.org/3/whatsnew/3.0.html

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

假定 23456 是PostgreSQL分配给这个函数的 OID。

参数被设置为全局变量。由于 Python 的可见范围规则，这会导致一种后果：在函数内不能把一个参数变量重新赋予给一个涉及该变量名称本身的表达式的值，除非在该代码块中重新声明该变量为全局的。例如，下面的代码无法工作：

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # 错误
  return x
$$ LANGUAGE plpythonu;
```

因为对x的赋值让x成为了整个代码块的一个局部变量，并且因此该赋值操作右边的x引用的是一个还未赋值的局部变量x，而不是 PL/Python 函数的参数。通过使用global语句，可以让上面的代码正常工作：

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # 现在好了
  return x
$$ LANGUAGE plpythonu;
```

但是不建议依赖于这类 PL/Python 的实现细节。最好把函数参数当作是只读。

46.3. 数据值

一般来讲，PL/Python 的目标是提供在 PostgreSQL 和 Python 世界之间的一种“自然的”映射。这包括下面介绍的数据映射规则。

46.3.1. 数据类型映射

在调用一个 PL/Python 函数时，它的参数会被从 PostgreSQL 的数据类型转换成相应的 Python 类型：

- PostgreSQL 的boolean被转换成 Python 的bool。
- PostgreSQL 的smallint和int被转换成 Python 的int。 PostgreSQL 的bigint和oid被转换成 Python 2 的long或者 Python 3 的int。
- PostgreSQL 的real和double被转换成 Python 的float。
- PostgreSQL 的numeric被转换成 Python 的Decimal。如果存在cdecimal包，则会从其中导入该类型。否则将使用来自标准库的decimal.Decimal。cdecimal比decimal要更快。不过，在 Python 3.3 以及更高的版本中，cdecimal已经被整合到了标准库中（也是用decimal这个名字），因此也就不再有什么区别。
- PostgreSQL 的bytea被转换成 Python 的str（Python 2）和bytes（Python 3）。在 Python 2 中，串应该被当做没有任何字符编码的字节序列对待。

- 包括 PostgreSQL 字符串类型在内的所有其他数据类型会被转换成一个 Python 的str。在 Python 2 中，这个串将用 PostgreSQL 服务器编码；在 Python 3 中，它将和所有串一样使用 Unicode。
- 对于非标量数据类型，请见下文。

当一个 PL/Python 函数返回时，会按照下列规则把它的返回值转换成该函数声明的 PostgreSQL 返回数据类型：

- 当 PostgreSQL 返回类型是boolean时，返回值会被根据Python规则计算真假。也就是说，0 和空串是假，但是要特别注意'f'是真。
- 当 PostgreSQL 返回类型是bytea时，返回值会被使用相应的 Python 内建机制转换成串（Python 2）或者字节（Python 3），结果将被转换成bytea。
- 对于所有其他 PostgreSQL 返回类型，返回值被使用 Python 内建的str转换成一个串，并且结果会被传递给 PostgreSQL 数据类型的输入函数（如果该 Python 值是一个float，它会被用内建的repr而不是str转换，这是为了避免精度损失）。

  当 Python 2 的串被传递给 PostgreSQL 时，它们被要求是 PostgreSQL 服务器编码。在当前服务器编码中不可用的串将会产生错误，但是并非所有的编码失配都能被检测到，因此当没有正确地将串编码时，垃圾数据仍然会产生。Unicode 串会被自动地转换为正确的编码，因此使用 Unicode 串更加安全并且更加方便。在 Python 3 中，所有串都是 Unicode 串。
- 对于非标量数据类型，请见下文。

注意所声明的 PostgreSQL 返回类型和实际返回对象的 Python 数据类型之间的逻辑失配不会被标志，无论怎样该值都会被转换。

46.3.2. Null, None

如果一个 SQL 空值被传递给一个函数，该参数值将作为 Python 中的None出现。例如，第 46.2 节展示的pymax的函数定义对于空值输入将会返回错误的回答。我们可以为函数定义增加STRICT让PostgreSQL做得更加合理：如果一个空值被传入，该函数将根本不会被调用，而只是自动地返回一个空结果。此外，我们可以在函数体中检查空输入：

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

如前所示，要从一个 PL/Python 函数返回一个 SQL 空值，可返回值None。不管该函数严格与否都可以这样做。

46.3.3. 数组、列表

SQL 数组会被作为一个 Python 列表传递到 PL/Python 中。要从一个 PL/Python 函数中返回出一个 SQL 数组值，可返回一个Python列表：

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
```

```
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

多维数组被当做嵌套的Python列表传入PL/Python。例如，一个2维数组是一个列表的列表。在把一个多维SQL数组从PL/Python函数返回出去时，每一层的内层列表都必须是相同的尺寸。例如：

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpythonu;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

为了与PostgreSQL的9.6以及更低版本的向后兼容性，当不支持多维数组时，也接受元组之类的其他Python序列。不过，它们总是被当做一维数组，因为它们会和组合类型混淆。出于同样的原因，当一个组合类型被用在多维数组中时，它必须被表示为一个元组而不是一个列表。

注意在 Python 中，串是序列，这可能产生与 Python 程序员所熟悉的不同的效果：

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
-----------------
 {h,e,l,l,o}
(1 row)
```

46.3.4. 组合类型

组合类型参数被作为 Python 映射传递给函数。映射的元素名称就是组合类型的属性名。如果被传递的行中有一个属性是空值，在映射中它的值是None。这里是一个例子：

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
```

```
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

有多种方式从一个 Python 函数返回行或者组合类型。下面的例子假设我们有：

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

一个组合结果可以被返回为：

序列类型（一个元组或者列表，但不是集合，因为 集合不能被索引）

被返回的序列对象必须具有和组合结果类型的域个数相同的项。索引号为 0 的项被分配给组合类型的第一个域，为 1 的项给第二个域，以此类推。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )
  # or alternatively, as tuple: return [ name, value ]
$$ LANGUAGE plpythonu;
```

要为任意列返回一个 SQL 空，应在对应的位置插入None。

当一个组合类型的数组被返回时，它不能被返回为列表，因为会弄不清该Python列表究竟是表示一个组合类型还是另一个数组维度。

映射（字典）

用列名作为键从映射中检索每一个结果类型列的值。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

任何额外的字典键/值对都会被忽略。丢失的键会被当做错误。要为任意列返回一个 SQL 空，应用相应的列名作为键插入None。

对象（任何提供方法__getattr__的对象）

这和映射的运作方式相同。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
```

```
        self.name = n
        self.value = v
    return named_value(name, value)

    # 或简单地
    class nv: pass
    nv.name = name
    nv.value = value
    return nv
$$ LANGUAGE plpythonu;
```

也支持具有OUT参数的函数。例如：

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

过程的输出参数会以同样的方式传回。例如：

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS $$
return (a * 3, b * 3)
$$ LANGUAGE plpythonu;

CALL python_triple(5, 10);
```

46.3.5. 集合返回函数

PL/Python函数也能返回标量类型或者组合类型的集合。有多种方法可以做到这一点，因为被返回的对象在内部会被转变成一个迭代器。下面的例子假设我们有组合类型：

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

可从以下类型返回集合结果：

序列类型（元组、列表、集合）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # 把包含列表的元组返回为组合类型
  # 所有其他组合也能行
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

迭代器（任何提供__iter__以及 next方法的对象）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
```

```
        def __init__ (self, how, who):
          self.how = how
          self.who = who
          self.ndx = -1

        def __iter__ (self):
          return self

        def next (self):
          self.ndx += 1
          if self.ndx == len(self.who):
            raise StopIteration
          return ( self.how, self.who[self.ndx] )

      return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
    $$ LANGUAGE plpythonu;
```

发生器（yield）

```
    CREATE FUNCTION greet (how text)
      RETURNS SETOF greeting
    AS $$
      for who in [ "World", "PostgreSQL", "PL/Python" ]:
        yield ( how, who )
    $$ LANGUAGE plpythonu;
```

也支持有OUT参数的集合返回函数（使用RETURNS SETOF record）。例如：

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer)
 RETURNS SETOF record AS $$
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

46.4. 共享数据

在对同一个函数的重复调用之间可用全局字典SD来存储私有数据。全局字典GD是公共数据，它对一个会话中的所有 Python 函数都可用，使用起来要小心。

在 Python 解释器中每一个函数都会得到自己的执行环境，因此来自myfunc的全局数据和函数参数对myfunc2不可用。例外是GD字典中的数据。

46.5. 匿名代码块

PL/Python 也支持用DO语句调用的匿名代码块：

```
DO $$
    # PL/Python 代码
$$ LANGUAGE plpythonu;
```

匿名代码块没有参数，并且任何可能返回的值都会被丢弃。否则，其行为就像一个函数。

46.6. 触发器函数

当函数被用作触发器时，字典TD包含触发器相关的值：

TD["event"]

包含字符串型的事件：INSERT、UPDATE、DELETE或者TRUNCATE。

TD["when"]

包含BEFORE、AFTER或者INSTEAD OF之一。

TD["level"]

包含ROW或者STATEMENT。

TD["new"]
TD["old"]

对于行级触发器，这些域的一个或者两个包含相应的触发器行，这取决于触发器事件是什么。

TD["name"]

包含触发器的名称。

TD["table_name"]

包含该触发器发生其上的表名。

TD["table_schema"]

包含该触发器发生其上的表所属的模式名。

TD["relid"]

包含该触发器发生其上的表的 OID。

TD["args"]

如果CREATE TRIGGER命令包括参数，它们可以通过TD["args"][0]至TD["args"][n-1]使用。

如果TD["when"]是BEFORE或者INSTEAD OF并且TD["level"]是ROW，可以从 Python 函数返回None或者"OK"来表示行没有被修改。返回"SKIP"可以中止事件，或者在TD["event"]为INSERT或UPDATE时可以返回"MODIFY"以表示已经修改了新行。否则返回值会被忽略。

46.7. 数据库访问

PL/Python 语言模块会自动导入一个被称为plpy的 Python 模块。这个模块中的函数和常量在 Python 代码中可以用plpy.foo这样的方式访问。

46.7.1. 数据库访问函数

plpy模块提供了几个函数来执行数据库命令：

plpy.execute(query [, max-rows])

用一个查询字符串和一个可选的行限制参数调用plpy.execute会让该查询运行并且其结果会被以一个结果对象返回。

结果对象模拟一个列表或者字典对象。可以用行号和列名来访问结果对象。例如：

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

会从my_table中返回 5 行。如果my_table有一列是my_column，可以这样来访问它：

```
foo = rv[i]["my_column"]
```

可以用内建的len函数获得返回的行数。

结果对象有这些额外的方法：

nrows()

返回被该命令处理的行数。注意这不一定与返回的行数相同。例如，UPDATE命令将会设置这个值但是不返回任何行（除非使用RETURNING）。

status()

SPI_execute()的返回值。

colnames()
coltypes()
coltypmods()

分别返回一个列名列表、列类型 OID 列表以及列的类型相关的类型修饰符列表。

在来自于不产生结果集合的命令的结果对象上调用这些方法会产生异常，例如不带RETURNING的UPDATE或者DROP TABLE。但是在包含的行数为零的结果集合上使用这些方法是 OK 的。

__str__()

也定义了标准的__str__方法，例如可以使用plpy.debug(rv)来调试查询执行结果。

结果对象可以被修改。

注意调用plpy.execute将会导致整个结果集合被读入到内存中。只有当确信结果集相对比较小时才应使用这个函数。在取得大型结果时，如果不想冒着耗尽内存的风险，应使用plpy.cursor而不是plpy.execute。

```
plpy.prepare(query [, argtypes])
plpy.execute(plan [, arguments [, max-rows]])
```

plpy.prepare为一个查询准备执行计划。它的参数是一个查询串和一个参数类型列表（如果查询中有参数引用）。例如：

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1",
 ["text"])
```

text是要为$1传递的变量的类型。如果不想给查询传递任何参数，第二个参数就是可选的。

在准备好一个语句后，可以使用函数plpy.execute的一种变体来运行它：

```
rv = plpy.execute(plan, ["name"], 5)
```

把计划作为第一个参数传递（而不是查询字符串），并且把要替换到查询中的值列表作为第二个参数传递。如果查询不需要任何参数，则第二个参数是可选的。和前面一样，第三个参数是可选的，它用来指定行数限制。

另外，你可以在计划对象上调用execute方法：

```
rv = plan.execute(["name"], 5)
```

查询参数以及结果行域会按照第 46.3 节所述在 PostgreSQL 和 Python 数据类型之间转换。

在使用 PL/Python 模块准备一个计划时，它会被自动保存。其含义可以阅读 SPI 文档（第 47 章。为了有效在函数调用之间利用这种特性，需要使用一种持久化存储字典SD或者GD（见第 46.4 节。例如：

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # 函数的剩余部分
$$ LANGUAGE plpythonu;
```

```
plpy.cursor(query)
plpy.cursor(plan [, arguments])
```

plpy.cursor函数接受和plpy.execute相同的参数（行数限制除外）并且返回一个游标对象，它允许以较小的块来处理大型的结果集合。和plpy.execute一样（行数限制除外），既可以使用一个查询字符串，也可以使用带有参数列表的计划对象，或者cursor函数可以作为计划对象的一个方法来调用。

游标对象提供了一种fetch方法，它接受一个整数参数并返回一个结果对象。每次调用fetch，返回的对象将包含下一批行，行数不会超过参数值。一旦所有的行都被消耗掉，fetch会开始返回一个空的结果对象。游标对象也提供一种迭代器接口[4]，它一次得到一行直到所有行被耗尽。用这种方法取得的数据不会被作为结果对象返回，而是以字典的形式返回，每一个字典对应于一个结果行。

从一个大型表中以上述两种方式处理数据的例子：

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
```

[4] https://docs.python.org/library/stdtypes.html#iterator-types

```
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0",
 ["integer"])
rows = list(plpy.cursor(plan, [2]))  # or: = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

游标会被自动丢弃掉。但是如果想要显式地释放游标所持有的所有资源，可使用close方法。一旦被关闭，就再也不能从游标中取得数据。

> **提示**
>
> 不要把plpy.cursor创建的游标对象与Python Database API specification[5]定义的 DB-API 游标弄混。除了名字之外，它们之间没有任何共同点。

46.7.2. 捕捉错误

访问数据库的函数可能会碰到错误，这将导致函数中止并且产生异常。plpy.execute和plpy.prepare都能产生plpy.SPIError的一个子类的实例，这默认将终止该函数。通过使用try/except结构，这种错误可以像其他 Python 异常一样被处理。例如：

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

产生的异常的实际类对应于特定的导致该错误的情况。可能的情况列表请参考 A.1 模块plpy.spiexceptions为每一种PostgreSQL情况定义了一个异常类，并且根据情况的名称命名。例如：division_by_zero变成DivisionByZero，unique_violation变成UniqueViolation，fdw_error变成FdwError，等等等等。这些异常类的每一种都是从SPIError继承而来。这种分离让处理特定错误更加容易，例如：

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS
 $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int",
 "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
```

[5] https://www.python.org/dev/peps/pep-0249/

```
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

注意因为所有来自于plpy.spiexceptions模块的异常都继承自SPIError，一个处理它的except子句将捕捉任何数据库访问错误。

作为另一种处理不同错误情况的方法，可以捕捉SPIError异常并且在except块中通过查看异常对象的sqlstate属性来判断错误情况。这种属性是包含着“SQLSTATE”错误代码的一个字符串值。这种方法提供了近乎相同的功能

46.8. 显式子事务

按第 46.7.2 节所述的从数据库访问导致的错误中恢复可能导致不好的情况：某些操作在其中一个操作失败之前已经成功，并且在从错误中恢复后这些操作的数据形成了一种不一致的状态。PL/Python 通过显式子事务的形式为这种问题提供了一套解决方案。

46.8.1. 子事务上下文管理器

考虑一个实现在两个账户间进行转账的函数：

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name
 = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name
 = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

如果第二个UPDATE语句导致产生一个异常，这个函数将会报告该错误，但是第一个UPDATE的结果却不会被提交。换句话说，资金将从 Joe 的账户中收回，而不会转移到 Mary 的账户中。

为了避免这类问题，可以把plpy.execute包裹在显式子事务中。plpy模块提供了一种助手对象来管理用plpy.subtransaction()函数创建的显式子事务。这个函数创建的对象实现了上下文管理器接口[6]。通过使用显式子事务，我们可以把函数写成：

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE
 account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE
 account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
```

[6] https://docs.python.org/library/stdtypes.html#context-manager-types

```
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

注意仍需使用try/catch。否则异常会传播到 Python 栈的顶层并且将导致整个函数以一个PostgreSQL错误中止，这样不会有任何行被插入到operations表。子事务上下文管理器不会捕捉错误，它只确保在其范围内执行的所有数据库操作将被原子性地提交或者回滚。在任何类型的异常（并非只是数据库访问产生的错误）退出时，会发生子事务块回滚。在显式子事务块内部产生的常规 Python 异常也会导致子事务被回滚。

46.8.2. 更旧的 Python 版本

Python 2.6 中默认可用的是使用with关键词的上下文管理器语法。如果 PL/Python 用的是一种较老的 Python 版本，仍然可以使用显式子事务，尽管不是那么透明。你可以使用别名enter和exit调用子事务管理器的__enter__和__exit__函数。转移资金的例子函数可以写成：

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE
 account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE
 account_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

> **注意**
>
> 尽管 Python 2.5 中实现了上下文管理器，要在那个版本中使用with语法，需要使用一个future 语句[7]。不过，由于实现细节的原因，不能在 PL/Python 函数中使用 future 语句。

46.9. 事务管理

在从顶层调用的过程中或者从顶层调用的匿名代码块（DO命令）中，可以控制事务。要提交当前的事务，可调用plpy.commit()。要回滚当前事务，可调用plpy.rollback()（注意不能

[7] https://docs.python.org/release/2.5/ref/future.html

通过plpy.execute或类似的函数运行SQL命令COMMIT或者ROLLBACK。这类工作必须用这些函数完成）。在事务结束以后，一个新的事务会自动开始，因此没有独立的函数用来开始新事务。

这里是一个例子：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpythonu
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

当一个显式的子事务处于活跃状态时，事务不能被结束。

46.10. 实用函数

plpy模块也提供了函数

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
plpy.notice(msg, **kwargs)
plpy.warning(msg, **kwargs)
plpy.error(msg, **kwargs)
plpy.fatal(msg, **kwargs)
```

plpy.error和plpy.fatal实际上会产生一个 Python 异常（如果没被捕捉），它会被传播到调用查询中导致当前事务或者子事务被中止。raise plpy.Error(msg)和raise plpy.Fatal(msg)分别等效于调用plpy.error(msg)和plpy.fatal(msg)，不过raise形式不允许传递关键词参数。其他函数只生成不同优先级的消息。一个特定优先级的消息是被报告给客户端、写入服务器日志还是两者都做，由log_min_messages和client_min_messages配置变量控制。详见第 19 章

msg参数被给定位一个位置参数。为了向后兼容，可以给出多于一个位置参数。在那种情况下， 位置参数形成的元组的字符串表达将会变成报告给客户端的消息。

下列 keyword-only 参数会被接受：

```
detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
constraint_name
```

作为 keyword-only 参数传递的对象的字符串表达可以用来丰富报告给客户端的消息。例如：

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

另一组工具函数是plpy.quote_literal(string)、plpy.quote_nullable(string)以及plpy.quote_ident(string)。它们等效于第 9.4 节描述的内建引用函数。在构建临时查询时它们能派上用场。例 43.1中动态 SQL 的 PL/Python 等效体是：

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

46.11. 环境变量

某些 Python 解释器接受的环境变量也能被用来影响 PL/Python 行为。它们需要在主 PostgreSQL 服务器进程的环境中设置，例如在一个启动脚本中设置。可用的环境变量取决于 Python 的版本，细节可见 Python 文档。在编写这份文档时，下面的环境变量可以对 PL/Python 产生影响（假定有一个合适的 Python 版本）：

- PYTHONHOME
- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

（Python 的实现细节似乎超出了 PL/Python 的控制范围，某些列在python手册页上的环境变量只在命令行解释器中有效，但在嵌入式 Python 解释器中无效）。

第 47 章 服务器编程接口

服务器编程接口（SPI）给予用户定义C函数编写者在其函数内运行SQL命令的能力。SPI是一组接口函数，它们可以简化对解析器、规划器和执行器的访问。SPI也做一些内存管理。

> **注意**
>
> 可用的过程语言提供了多种方法从函数中执行 SQL 命令。大部分这些设施都是基于 SPI 的，因此这个文档也对那些语言的用户有用。

注意如果一个通过 SPI 调用的命令失败，那么控制将会返回到你的C函数中。当然啦，你的C函数所在的事务或者子事务将被回滚（这可能看起来令人惊讶，因为据文档所说 SPI 函数大多数都有错误返回约定。但是那些约定只适用于在 SPI 函数本身内部检测到的错误）。通过在可能失败的 SPI 调用周围建立自己的子事务可以在错误之后恢复控制。

SPI成功时返回一个非负结果（要么通过一个返回的整数值，要么如下所述放在全局变量SPI_result中）。错误时，将会返回一个负结果或者NULL。

使用 SPI 的源代码文件必须包括头文件executor/spi.h。

47.1. 接口函数

SPI_connect

SPI_connect, SPI_connect_ext — 连接一个C函数到 SPI 管理器

大纲

```
int SPI_connect(void)
```

```
int SPI_connect_ext(int options)
```

描述

SPI_connect从一个C函数调用中打开一个到 SPI 管理器的连接。如果你想要通过 SPI 执行命令，你必须调用这个函数。有一些功能性 SPI 函数可以从未连接的C函数中调用。

SPI_connect_ext会做同样的事情，但是有一个允许传递选项标志的参数。当前有下列选项值可用：

SPI_OPT_NONATOMIC

设置SPI连接为nonatomic，这表示允许事务控制调用SPI_commit、SPI_rollback以及SPI_start_transaction。否则，调用这些函数将立即导致错误。

SPI_connect()等效于SPI_connect_ext(0)。

返回值

SPI_OK_CONNECT

成功时

SPI_ERROR_CONNECT

错误时

SPI_finish

SPI_finish — 将一个C函数从 SPI 管理器断开

大纲

```
int SPI_finish(void)
```

描述

SPI_finish关闭一个到 SPI 管理器的现有连接。你必须在完成你的C函数的当前调用中所需的 SPI 操作之后必须调用这个函数。不过，如果你通过elog(ERROR)中断了事务，你无须担心这个函数的调用。在那种情况下，SPI 将自己自动进行清理。

返回值

SPI_OK_FINISH

如果正确地断开连接

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_execute

SPI_execute — 执行一个命令

大纲

```
int SPI_execute(const char * command, bool read_only, long count)
```

描述

SPI_execute执行指定的 SQL 命令以获得count行。如果read_only为true，该命令必须是只读的，并且执行开销也会有所降低。

只能从一个已连接的C函数中调用这个函数。

如果count为零，那么该命令会为其所适用的所有行执行。如果count大于零，那么会检索不超过count行，当到达该计数时执行会停止，这很像为查询增加一个LIMIT子句。例如：

```
SPI_execute("SELECT * FROM foo", true, 5);
```

会从表中检索至多 5 行。注意这样一个限制只有当命令真正返回行时才有效。例如：

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

插入所有来自于bar的行，而忽略count参数。不过，通过

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

将插入至多 5 行，因为在第五个RETURNING结果行被检索到后执行就会停止。

你可以在一个字符串中传递多个命令，SPI_execute会返回最后一个被执行的命令的结果。count限制单独适用于每一个命令（即便只有最后一个结果会被实际返回）。该限制 不适用于由规则产生的任何隐藏命令。

当read_only是false时， SPI_execute增加命令计数器并且在执行字符串中每一个命令之前计算一个新的snapshot。如果当前事务隔离级别是SERIALIZABLE或REPEATABLE READ， 该快照并不会实际改变。但是在READ COMMITTED模式中，快照更新允许每个命令看到来自其他会话中新近已提交事务 的结果。当命令正在修改数据库时，这对一致性行为非常重要。

当read_only是true时， SPI_execute不更新快照或者命令计数器，并且它只允许纯SELECT命令出现在命令字符串中。这些命令被使用之前为周围查询 建立的快照来执行。这种执行模式要比读/写模式更快，因为消除了每个命令跟新快照的开销。 它也允许建立真正stable的函数：因为连续执行将会使用同一个快照，因此结果不会有改变。

在一个使用 SPI 的单一函数中混合只读和读写命令通常是不明智的， 这样可能会导致非常令人困惑的行为，因为只读查询将看不到任何 由读写查询完成的数据库更新结果。

被执行的（最后一个）命令的实际行数使用全局变量SPI_processed返回。 如果该函数的返回值是SPI_OK_SELECT、 SPI_OK_INSERT_RETURNING、 SPI_OK_DELETE_RETURNING或者SPI_OK_UPDATE_RETURNING， 那么你可以使用全局指针SPITupleTable *SPI_tuptable来访问结果行。 某些工具命令（例如EXPLAIN）也返回行集合，并且在这些情况中SPI_tuptable也会包含该结果。某些工具命令（COPY、CREATE TABLE AS）不返回一个行集合， 因此SPI_tuptable为 NULL，但是它们仍然会在SPI_processed中返回被处理的行数。

结构SPITupleTable被定义为：

```
typedef struct
{
    MemoryContext tuptabcxt;    /* 结果表的内存上下文 */
    uint64      alloced;        /* 已分配值的数量 */
    uint64      free;           /* 空限值的数量 */
    TupleDesc   tupdesc;        /* 行描述符 */
    HeapTuple  *vals;           /* 行 */
} SPITupleTable;
```

vals是一个行指针的数组（可用项的数量由SPI_processed给出）。 tupdesc是一个行描述符，你可以把它传递给 SPI 函数来处理行。tuptabcxt、 alloced和free是不准备给 SPI 调用者使用的内部域。

SPI_finish释放在当前的C函数中已分配的所有SPITupleTable。 如果你已经用完了一个结果表，你可以通过调用SPI_freetuptable提早释放它。

参数

const char * command

包含要执行命令的字符串

bool read_only

对只读执行为true

long count

要返回的最大行数，或者用0表示没有限制

返回值

如果命令的执行成功，那么将会返回下列（非负）值之一：

SPI_OK_SELECT

如果执行了一个SELECT（但不是SELECT INTO）

SPI_OK_SELINTO

如果执行了一个SELECT INTO

SPI_OK_INSERT

如果执行了一个INSERT

SPI_OK_DELETE

如果执行了一个DELETE

SPI_OK_UPDATE

如果执行了一个UPDATE

SPI_OK_INSERT_RETURNING

如果执行了一个INSERT RETURNING

SPI_OK_DELETE_RETURNING

如果执行了一个DELETE RETURNING

SPI_OK_UPDATE_RETURNING

如果执行了一个UPDATE RETURNING

SPI_OK_UTILITY

如果执行了一个工具命令（例如CREATE TABLE）

SPI_OK_REWRITTEN

如果该命令被一个规则重写成了另一类命令（例如UPDATE变成了一个INSERT）

发生错误时，将会返回下列负值之一：

SPI_ERROR_ARGUMENT

如果command为NULL或者count小于 0

SPI_ERROR_COPY

如果尝试COPY TO stdout或者COPY FROM stdin

SPI_ERROR_TRANSACTION

如果尝试了一个事务操纵命令（ BEGIN、 COMMIT、 ROLLBACK、 SAVEPOINT、 PREPARE TRANSACTION、 COMMIT PREPARED、 ROLLBACK PREPARED或者其他变体）

SPI_ERROR_OPUNKNOWN

如果命令类型位置（不应该会发生）

SPI_ERROR_UNCONNECTED

如果从未连接的C函数中调用

注解

所有 SPI 查询执行函数都会设置SPI_processed和SPI_tuptable（只是指针， 而不是结构的内容）。如果你需要在以后访问SPI_execute或另一个查询执行函数的结果表， 请将这两个全局变量保存到本地的C函数变量中。

SPI_exec

SPI_exec — 执行一个读/写命令

大纲

```
int SPI_exec(const char * command, long count)
```

描述

SPI_exec和 SPI_execute相同，但后者的 read_only参数的值总是取 false。

参数

const char * command

包含要执行的命令的字符串

long count

要返回的最大行数，0表示没有限制

返回值

见SPI_execute。

SPI_execute_with_args

SPI_execute_with_args — 用线外参数执行一个命令

大纲

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

描述

SPI_execute_with_args执行一个可能包括 对外部提供的参数引用的命令。命令文本用 $n引用一个参数，并且调用 会为每一个这种符号指定数据类型和值。 read_only和 count的解释与 SPI_execute中相同。

相对于SPI_execute，这个例程的主要优　　点是数据值可以被插入到命令中而无需冗长的引用/转义，并且因此 减少了 SQL 注入攻击的风险。

可以通过在SPI_prepare后面跟上　SPI_execute_plan达到相似的结果。但是，　使用这个函数时查询计划总是被定制成提供的指定参数值。对于一次　性的查询执行，这个函数应该更好。如果同样的命令需要用很多不同　的参数执行，两种方法都可能会更快，这取决于重新做规划的代价与 定制计划带来的好处之间的对比。

参数

const char * command

　命令字符串

int nargs

　输入参数的数量（$1、$2等等）。

Oid * argtypes

　一个长度为nargs的数组，　包含参数的数据类型的OID

Datum * values

　一个长度为nargs的数组，　包含实际的参数值

const char * nulls

　一个长度为nargs的数组，　描述哪些参数为空值

　如果nulls为NULL，　那么SPI_execute_with_args会假设没有参数　为空值。否则，如果对应的参数值为非空，　nulls　数组的每一个项都应该是’　’；如果对应参数值为空，nulls数组的项应为’n’（在后　面的情况中，对应的values项中的值没有　关系）。注意nulls不是一个文本字符串，　它只是一个数组：它不需要一个’\0’终止符。

bool read_only

　对只读执行是true

long count

　要返回的最大行数，0表示没有限制

返回值

该返回值和SPI_execute一样。

如果成功SPI_execute会设置 SPI_processed和 SPI_tuptable。

SPI_prepare

SPI_prepare — 准备一个语句，但不执行它

大纲

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

描述

SPI_prepare为指定的命令创建并且返回一个 预备语句，但是并不执行该命令。该预备语句会在稍后使用 SPI_execute_plan重复执行。

当相同的或者相似的命令要被重复执行时，通常来说只执行一次解析分 析是有利的，并且更有利的是重用该命令的执行计划。 SPI_prepare把一个命令字符串转换成一个预 备语句，它包装了解析分析的结果。如果发现为每一次执行都生成一个 定制计划没有帮助，该预备语句也提供了一个地方缓存执行计划。

一个预备命令可以被一般化为在一个普通命令中应该出现常量的地方写 上参数（$1、$2等等）。参数的实际值在 SPI_execute_plan被调用时指定。这让该预备 语句可以比没有参数的形式用户与更广泛的情况。

SPI_prepare返回的语句只能在当前的C过程调用 中使用，因为SPI_finish会释放为这样一个语句 分配的内存。但是可以使用函数SPI_keepplan 或SPI_saveplan把该语句保存更久。

参数

`const char * command`

命令字符串

`int nargs`

输入参数（$1、$2等等）的数量

`Oid * argtypes`

一个数组指针，它指向的数组包含参数的数据类型的 OID

返回值

SPI_prepare返回一个指向SPIPlan 的非空指针，它是一个表示一个预备语句的不透明结构。发生错误时， 将会返回NULL，并且 SPI_result将被设置为一个也被 SPI_execute使用的错误码，不过当 command为NULL、 或者nargs小于零、或者nargs大于 零但是argtypes为NULL 时它会被设置为SPI_ERROR_ARGUMENT。

注解

如果没有定义参数，在第一次使用SPI_execute_plan 时将会创建一个一般的计划，并且把它用于所有的后续执行。如果有参数， SPI_execute_plan的前几次使用将根据提供的参数值 产生定制计划。在使用同一个预备语句足够多次后， SPI_execute_plan将构建一个一般计划，并且如果它 并不比定制计划昂贵太多， SPI_execute_plan将开始使用一般计划来取代每次都进行重新规划。如果这种默认的行为不合适，你可以 通过传递CURSOR_OPT_GENERIC_PLAN或 CURSOR_OPT_CUSTOM_PLAN标志给 SPI_prepare_cursor，以分别强制使用一般或者定制 计划。

尽管一个预备语句的要点是避免对语句的重复解析分析以及规划，只要语句中 用到的数据库对象从上一次使用该预备语句以来经历过定义性（DDL）改变， PostgreSQL将会强制重新分析和重新规划该语句。还有， 如果search_path的值从一个改变成下一个，该语句将 会使用新的search_path进行重新解析（后一种行为是从 PostgreSQL 9.3 开始的新行为）。更多有关预备语句行为的信息请见PREPARE。

这个函数只能从一个已连接的C函数调用。

SPIPlanPtr被声明为spi.h中的一种不透明结构类型 的指针。尝试直接访问其内容是不明智的，因为那会让你的代码更有可能会在未 来版本的PostgreSQL中崩溃。

SPIPlanPtr这个名字多少有点历史原因，因为该数据结构不再需要包 含一个执行计划。

SPI_prepare_cursor

SPI_prepare_cursor — 预备一个语句，但是不执行它

大纲

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

描述

SPI_prepare_cursor和 SPI_prepare一样，不过它也允许说明规划器的 “游标选项”参数。这是一个位掩码，它的值如 nodes/parsenodes.h中 DeclareCursorStmt的options域所示。SPI_prepare总是把该游标选项取做零。

参数

`const char * command`

命令字符串

`int nargs`

输入参数（$1、$2等等）的数量

`Oid * argtypes`

一个数组指针，它指向的数组包含参数的数据类型的 OID

`int cursorOptions`

整数形式的游标选项位掩码，零会导致默认行为

返回值

SPI_prepare_cursor具有和 SPI_prepare一样的返回习惯。

注解

在cursorOptions设置的有用的位包括 CURSOR_OPT_SCROLL、 CURSOR_OPT_NO_SCROLL、CURSOR_OPT_FAST_PLAN、 CURSOR_OPT_GENERIC_PLAN以及 CURSOR_OPT_CUSTOM_PLAN。 注意CURSOR_OPT_HOLD被特别地忽略。

SPI_prepare_params

SPI_prepare_params — 预备一个语句，但是不执行它

大纲

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

描述

SPI_prepare_params为指定的命令创建并　返回一个预备语句，但是不执行该命令。这个函数等效于 SPI_prepare_cursor，此外调用者可以指定 解析器钩子函数来控制外部参数引用的解析。

参数

const char * command

　命令字符串

ParserSetupHook parserSetup

　解析器钩子设置函数

void * parserSetupArg

　用于parserSetup的转嫁参数

int cursorOptions

　整数形式的游标选项位掩码，零会导致默认行为

返回值

SPI_prepare_params具有和 SPI_prepare相同的返回习惯。

SPI_getargcount

SPI_getargcount — 返回一个由SPI_prepare 准备好的语句所需的参数数量

大纲

```
int SPI_getargcount(SPIPlanPtr plan)
```

描述

SPI_getargcount返回执行一个由 SPI_prepare准备好的语句所需的参数数量。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

返回值

plan所期望的参数计数。如果该 plan为NULL或者无效， SPI_result会被设置为SPI_ERROR_ARGUMENT 并且返回 -1。

SPI_getargtypeid

SPI_getargtypeid — 为由SPI_prepare 准备好的一个语句的一个参数返回其数据类型 OID

大纲

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

描述

SPI_getargtypeid返回由 SPI_prepare准备好的一个语句的 第argIndex个参数的类型的OID。 第一个参数的索引为零。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

int argIndex

参数的索引，从零开始

返回值

给定索引处的参数的类型 OID。如果该 plan为NULL或者无效， 或者argIndex小于零或者小于为 plan声明的参数数量， SPI_result会被设置为 SPI_ERROR_ARGUMENT并且将会返回InvalidOid。

SPI_is_cursor_plan

SPI_is_cursor_plan — 如果一个由SPI_prepare预备好 的语句可以用于SPI_cursor_open则返回 true

大纲

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

描述

如果一个由SPI_prepare预备好 的语句可以被作为一个参数传递给SPI_cursor_open，SPI_is_cursor_plan会返回true。 否则返回false。原则是该 plan表示一个单一命令并且这个命令向其调用者 返回元组。例如，只要不含INTO子句，SELECT 就被允许，而只有包含一个RETURNING子句时才允许 UPDATE。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

返回值

如果该plan能产生一个游标则返回 true，否则返回false 并且把SPI_result设置为零。如果不可能决定答案 （例如，如果plan为 NULL或无效，或者在没有连接到 SPI 时调用）， 那么SPI_result会被设置为一个合适的错误码 并且返回false。

SPI_execute_plan

SPI_execute_plan — 执行一个由SPI_prepare预备好的语句

大纲

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

描述

SPI_execute_plan执行一个由 SPI_prepare或其同类方法准备好的语句。 read_only和 and count的解释和 SPI_execute中相同。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

Datum * values

一个实际参数值的数组。必须和语句的参数数量等长。

const char * nulls

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果nulls为NULL， 那么SPI_execute_plan会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool read_only

true表示只读执行

long count

要返回的行的最大数量，或者用0表示没有限制

返回值

返回值和SPI_execute相同， 还有下列额外可能的错误（负值）结果：

SPI_ERROR_ARGUMENT

如果plan为NULL 或者非法，或者count小于 0

SPI_ERROR_PARAM

如果values为NULL但是 plan被准备时用了一些参数

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — 执行一个由SPI_prepare预备好的语句

大纲

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

描述

SPI_execute_plan_with_paramlist执行一个由 SPI_prepare准备好的语句。这个函数与 SPI_execute_plan等效，不过被传递给该查询 的参数值的信息以不同的方式呈现。ParamListInfo表现形 式更方便于把这种格式的值向下传递。它也支持通过ParamListInfo中指定的钩子函数动态设置参数。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

ParamListInfo params

包含参数类型和值的数据结构，如果没有则为 NULL

bool read_only

true表示只读执行

long count

要返回的行的最大数量，或者用0表示没有限制

返回值

返回值和SPI_execute_plan相同。

成功时，在SPI_execute_plan中会设置 SPI_processed和 SPI_tuptable。

SPI_execp

SPI_execp — 以读/写模式执行一个语句

大纲

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

描述

SPI_execp与 SPI_execute_plan相同，不过后者的 read_only参数总是取false。

参数

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

Datum * values

实际参数值的数组。长度必须等于该语句的参数数量。

const char * nulls

描述哪些参数为空值的数据。长度必须等于该语句的参数数量。

如果nulls为NULL， 那么SPI_execp会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

long count

要返回的行的最大数量，或者用0表示没有限制

返回值

见SPI_execute_plan。

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

SPI_cursor_open

SPI_cursor_open — 使用由SPI_prepare创建的 语句建立一个游标

大纲

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

描述

SPI_cursor_open建立一个游标（在内部是一个 portal），该游标将执行由SPI_prepare准备好 的一个语句。参数具有和SPI_execute_plan的 相应参数相同的含义。

使用一个游标而不是直接执行该语句有两个好处。首先，可以一次只取出 一些结果行，避免为返回很多行的查询过度使用内存。其次，一个 portal 可以比当前的C函数生存更长时间（事实上，它可以生存到当前事务结束）。 把 portal 的名称返回给该C函数的调用者提供了一种将一个行集合返回为结 果的方法。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

参数

const char * name

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

Datum * values

实际参数值的数组。长度必须等于该语句的参数数量。

const char * nulls

描述哪些参数是空值的数据。长度必须等于该语句的参数数量。

如果nulls为NULL， 那么SPI_cursor_open会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是’ ’；如果对应参数值为空， nulls数组的项应为’n’（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个’\0’终止符。

bool read_only

true表示只读执行

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定， 任何错误都将通过elog报告。

SPI_cursor_open_with_args

SPI_cursor_open_with_args — 使用一个查询和参数建立一个游标

大纲

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

描述

SPI_cursor_open_with_args建立一个将 执行指定查询的游标（在内部是一个 portal）。大部分参数具有和 SPI_prepare_cursor 和SPI_cursor_open中相应参数相同的含 义。

对于一次性的查询执行，这个函数应该比 SPI_prepare_cursor加上其后的SPI_cursor_open更好。如果相同的命令 要被用很多不同的参数执行，哪种方法更快就要取决于重做计划的 代价与定制计划带来的好处之间谁更有利。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

参数

const char * name

portal 的名字，或者设置成NULL 让系统选择一个名称

const char * command

命令字符串

int nargs

输入参数的数量（$1、$2等等）

Oid * argtypes

一个长度为nargs的数组，它包含参数的 数据类型的OID

Datum * values

一个长度为nargs的数组，它包含实际的参数值

const char * nulls

一个长度为nargs的数组，它描述哪些参数为空值

如果nulls为NULL， 那么SPI_cursor_open_with_args会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '; 如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool read_only

true表示只读执行

int cursorOptions

游标选项的整数型位掩码，为零会产生默认行为

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定， 任何错误都将通过elog报告。

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — 使用参数建立一个游标

大纲

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

描述

SPI_cursor_open_with_paramlist建立一个 游标（在内部是一个 portal），它将执行一个由 SPI_prepare准备好的语句。这个函数等效于 SPI_cursor_open，不过被传递给该查询 的参数值的信息以不同的方式呈现。ParamListInfo表现形 式更方便于把这种格式的值向下传递。它也支持通过 ParamListInfo中指定的钩子函数动态设置参数。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

参数

const char * name

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr plan

预备语句（由SPI_prepare返回）

ParamListInfo params

包含参数类型和值的数据结构，如果没有就为 NULL

bool read_only

true表示只读执行

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定， 任何错误都将通过elog报告。

SPI_cursor_find

SPI_cursor_find — 用名称查找一个现有的游标

大纲

```
Portal SPI_cursor_find(const char * name)
```

描述

SPI_cursor_find用名称查找一个现有的 portal。 这主要被用于解析由其他某个函数返回的一个油表名称。

参数

const char * name

该 portal 的名称

返回值

带有指定名称的 portal 的指针，如果没有找到就是 NULL

SPI_cursor_fetch

SPI_cursor_fetch — 从一个游标取出一些行

大纲

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

描述

SPI_cursor_fetch从一个游标取得一些行。 这等效于 SQL 命令FETCH的一个子集（更多功能见SPI_scroll_cursor_fetch）。

参数

Portal portal

包含该游标的 portal

bool forward

为真表示向前获取，为假表示向后获取

long count

要取得的最大行数

返回值

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

注解

如果该游标的计划不是用CURSOR_OPT_SCROLL 选项创建的，向后获取会失败。

SPI_cursor_move

SPI_cursor_move — 移动一个游标

大纲

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

描述

SPI_cursor_move跳过一个游标中的一些行。 这等效于 SQL 命令MOVE的一个子集（更多的功能 请见SPI_scroll_cursor_move）。

参数

Portal portal

包含该游标的 portal

bool forward

为真表示前移，为假表示后移

long count

要移动的最大行数

注解

如果该游标的计划不是用CURSOR_OPT_SCROLL 选项创建的，向后移动会失败。

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — 从一个游标取出一些行

大纲

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

描述

SPI_scroll_cursor_fetch从一个游标中取出一些行。 这等效于 SQL 命令FETCH。

参数

Portal portal

包含该游标的 portal

FetchDirection direction

FETCH_FORWARD、 FETCH_BACKWARD、 FETCH_ABSOLUTE或者 FETCH_RELATIVE之一

long count

FETCH_FORWARD或 FETCH_BACKWARD方式中要取出的行数； FETCH_ABSOLUTE方式中要取出的绝对行号； FETCH_RELATIVE方式中要取出的相对行号

返回值

成功时，就像在SPI_execute中会设置 SPI_processed和 SPI_tuptable。

注解

参数direction和 count的详细解释请见 SQL FETCH命令。

如果该游标的计划不是用CURSOR_OPT_SCROLL 选项创建的，除FETCH_FORWARD之外的方向值会 失败。

SPI_scroll_cursor_move

SPI_scroll_cursor_move — 移动一个游标

大纲

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

描述

SPI_scroll_cursor_move在一个游标中跳过 一定数量的行。这等效于 SQL 命令MOVE。

参数

Portal portal

包含该游标的 portal

FetchDirection direction

FETCH_FORWARD、 FETCH_BACKWARD、 FETCH_ABSOLUTE或者 FETCH_RELATIVE之一

long count

FETCH_FORWARD或者 FETCH_BACKWARD方式中要移动的行数； FETCH_ABSOLUTE方式中要移动到的绝对行号； FETCH_RELATIVE方式中要移动到的相对行号

返回值

成功时，就像在SPI_execute中会设置 SPI_processed。 SPI_tuptable被设置为NULL， 因为这个函数不需要返回行。

注解

参数direction和 count的详细解释请见 SQL FETCH命令。

如果该游标的计划不是用CURSOR_OPT_SCROLL 选项创建的，除FETCH_FORWARD之外的方向值会 失败。

SPI_cursor_close

SPI_cursor_close — 关闭一个游标

大纲

```
void SPI_cursor_close(Portal portal)
```

描述

SPI_cursor_close关闭一个之前创建的游标 并且释放它的 portal 存储。

所有打开的游标会在事务结束时自动被关闭。 只有在希望尽快释放资源时，才需要调用SPI_cursor_close。

参数

Portal portal

包含该游标的 portal

SPI_keepplan

SPI_keepplan — 保存一个预备语句

大纲

```
int SPI_keepplan(SPIPlanPtr plan)
```

描述

SPI_keepplan保存一个被传入的语句（由 SPI_prepare准备好），这样它将不会被SPI_finish或者事务管理器释放。这让你能够在当前会话的后续C函数调用中重用预备语句。

参数

SPIPlanPtr plan

要保存的预备语句

返回值

成功返回 0；如果plan为NULL 或者无效则返回SPI_ERROR_ARGUMENT

注解

这个函数通过指针调整的方法（不需要数据复制）将被传入的语句重定位到永久存储中。如果你后来需要删除它，可以对它使用 SPI_freeplan。

SPI_saveplan

SPI_saveplan — 保存一个预备语句

大纲

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

描述

SPI_saveplan把一个被传入的语句（由 SPI_prepare准备好）复制到不会被 SPI_finish或者事务管理器释放的内存中。 这让你能够在当前会话的后续C函数调用中重用预备语句。

参数

SPIPlanPtr plan

要保存的预备语句

返回值

要被复制的语句的指针；如果没有成功则返回NULL。 错误时，SPI_result会被这样设置：

SPI_ERROR_ARGUMENT

如果plan为NULL或无效

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数调用

注解

原始的被传入的语句不会被释放，因此你可能希望在其上执行 SPI_freeplan以避免在SPI_finish之前发生内存泄露。

在大部分情况下，SPI_keepplan更适合于 执行这种功能，因为它极大程度上达到了同样的结果而无需物理地 复制该预备语句的数据结构。

SPI_register_relation

SPI_register_relation — make an ephemeral named relation available by name in SPI queries

大纲

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

简介

SPI_register_relation用相关信息建立一个短暂的命名关系，它对通过当前SPI连接规划和执行的查询可用。

参数

EphemeralNamedRelation enr

短暂的命名关系的注册项

返回值

如果该命令的执行成功，则将返回下列（非负）值：

SPI_OK_REL_REGISTER

如果该关系已经成功地用名称注册

在出错时，会返回下列负值之一：

SPI_ERROR_ARGUMENT

如果enr是NULL或者其name字段是NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_DUPLICATE

如果enr的name字段中指定的名称已经为这个连接注册

SPI_unregister_relation

SPI_unregister_relation — remove an ephemeral named relation from the registry

大纲

```
int SPI_unregister_relation(const char * name)
```

简介

SPI_unregister_relation从当前连接的注册项中移除一个短暂存在的关系。

参数

const char * name

关系的注册项名称

返回值

如果该命令的执行成功，则会返回下列（非负）值：

SPI_OK_REL_UNREGISTER

如果tuplestore已经被成功地从注册项中移除

出现错误时，会返回下列负值之一：

SPI_ERROR_ARGUMENT

如果name为NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_NOT_FOUND

如果没有在当前连接的注册项中找到name

SPI_register_trigger_data

SPI_register_trigger_data — make ephemeral trigger data available in SPI queries

大纲

```
int SPI_register_trigger_data(TriggerData *tdata)
```

简介

SPI_register_trigger_data会造出被触发器捕获的任何短暂存在的关系，它们对通过当前SPI连接规划和执行的查询可用。当前，这表示用REFERENCING OLD/NEW TABLE AS ... 子句定义的被AFTER触发器捕获的传递表。这个函数应该被一个PL触发器的处理器函数在连接之后调用。

参数

TriggerData *tdata

以fcinfo->context传递给触发器处理器函数的TriggerData对象

返回值

如果命令的执行成功，则会返回下列（非负）值：

SPI_OK_TD_REGISTER

如果被捕获的触发器数据（如果有）已经被成功地注册

出现错误时，会返回下列负值之一：

SPI_ERROR_ARGUMENT

如果tdata为NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的C函数中调用

SPI_ERROR_REL_DUPLICATE

如果任何触发器数据瞬时关系的名字已经为这个连接注册过

47.2. 接口支持函数

这里描述的函数提供了一个接口从SPI_execute 及其他 SPI 函数返回的结果集中抽取信息。

这一小节中描述的所有函数都可以被用在已连接和未连接的C函数中。

SPI_fname

SPI_fname — 为指定的列号确定列名

大纲

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

描述

SPI_fname返回一个指定列的列名的拷贝（ 当你不再需要该列名拷贝后，可以使用pfree 释放它）。

参数

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

返回值

列名；如果colnumber超出范围则返回 NULL。出错时 SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

SPI_fnumber

SPI_fnumber — 为一个指定的列名确定列号

大纲

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

描述

SPI_fnumber返回指定列名的列号。

如果colname引用的是一个系统列（例如， oid），那么将返回对应的负值列号。调用者应该小心地测试 返回值是不是正好为SPI_ERROR_NOATTRIBUTE 来检测错误；除非系统列应该被拒绝，测试结果是否小于或者等于零这种 方式是不正确的。

参数

TupleDesc rowdesc

输入行描述

const char * colname

列名

返回值

列号（用户定义的列从1开始计），如果没有找到所提到的列名则返回SPI_ERROR_NOATTRIBUTE。

SPI_getvalue

SPI_getvalue — 返回指定列的字符串值

大纲

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

描述

SPI_getvalue返回指定列的值的字符串表示。

结果在使用palloc分配的内存中返回（当你不再 需要该结果时，你可以使用pfree释放该内存）。

参数

HeapTuple row

要检查的输入行

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

返回值

列值，如果列为空值、colnumber超出范围（SPI_result被设置为SPI_ERROR_NOATTRIBUTE）或者没有输出函数可用（SPI_result被设置为SPI_ERROR_NOOUTFUNC）则返回 NULL。

SPI_getbinval

SPI_getbinval — 返回指定列的二进制值

大纲

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

描述

SPI_getbinval以内部格 式（以Datum类型）返回指定列的值。

这个函数不会为该 datum 分配新空间。在传引用数据类型的情况下， 返回值将是一个被传递行的指针。

参数

HeapTuple row

要检查的输入行

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

bool * isnull

列中是否为空值的标志

返回值

该列的二进制值会被返回。如果该列为空值，由isnull 指向的变量将被设置为真，否则会被设置为假。

错误时SPI_result会被设置成 SPI_ERROR_NOATTRIBUTE。

SPI_gettype

SPI_gettype — 返回指定列的数据类型名称

大纲

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

描述

SPI_gettype返回该指定列的数据类型名称的拷贝（当你不再需要该拷贝后，可以使用pfree 释放它）。

参数

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

返回值

指定列的数据类型名称，或者在错误时返回NULL。错误时SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

SPI_gettypeid

SPI_gettypeid — 返回指定列的数据类型的OID

大纲

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

描述

SPI_gettypeid返回该指定列的数据类型的 OID。

参数

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

返回值

指定列的数据类型的OID，或者出错时返回 InvalidOid。出错时， SPI_result会被设置成SPI_ERROR_NOATTRIBUTE。

SPI_getrelname

SPI_getrelname — 返回指定关系的名称

大纲

```
char * SPI_getrelname(Relation rel)
```

描述

SPI_getrelname返回该指定关系的名称 的拷贝（当你不再需要该拷贝后，可以使用pfree 释放它）。

参数

Relation rel

输入关系

返回值

指定关系的名称。

SPI_getnspname

SPI_getnspname — 返回指定关系的名字空间

大纲

```
char * SPI_getnspname(Relation rel)
```

描述

SPI_getnspname返回指定 关系所属的名字空间的名称拷贝。这等效 于该关系的模式。当你用完这个函数的返回值后，应该调用 pfree释放它。

参数

Relation rel

输入关系

返回值

指定关系的名字空间的名称。

SPI_result_code_string

SPI_result_code_string — return error code as string

大纲

```
const char * SPI_result_code_string(int code);
```

简介

SPI_result_code_string返回之前各种SPI函数返回的或者存储在SPI_result中的结果代码的字符串表示。

参数

int code

结果代码

返回值

结果代码的字符串表示。

47.3. 内存管理

PostgreSQL 在内存上下文中分配内存，内存上下文为管理 在多个不同位置、具有不同生存时间需要的分配提供了一种便捷的方法。 销毁一个上下文会释放所有在其中分配的内存。因此不必跟踪单个对象来 避免内存泄露，而是只需要管理数量相对较少的上下文即可。 palloc和相关的函数可以从“当前” 上下文中分配内存。

SPI_connect创建一个新的内存上下文并且让它 成为当前上下文。SPI_finish恢复之前的当前上下 文并且销毁由SPI_connect创建的内存上下文。 这些动作确保在你的C函数中分配的内存在C函数退出时被回收，从而避免内存 泄露。

不过，如果你的C函数需要返回一个在已分配内存中的对象（例如一个 传引用数据类型的值），你不能使用palloc 分配内存，或者说至少不能在连接到 SPI 时这样做。如果你试着这样 做，该对象会被SPI_finish接触分配，那么 你的C函数将无法可靠地工作。要解决这个问题，应使用 SPI_palloc来为要返回的对象分配内存。 SPI_palloc会在 “上层执行器上下文”中分配内存，也就是当 SPI_connect被调用时的当前内存上下文， 它才是从你的C函数中返回的值最适合的上下文。这一节中描述的几个其他实用函数也会返回在上层执行器上下文中创建的对象。

当SPI_connect被调用时，这个C函数的私有 上下文（由SPI_connect）会被作为当前上 下文。所有用palloc、 repalloc或者 SPI 功能函数（除了这个小节中描述的例外）分配的内存都在这个上下文中。 当一个C函数从SPI管理器断开连接时（通过 SPI_finish），当前上下文被恢复到上层的 执行器上下文，并且在该过程的内存上下文中分配的内存都会被释放， 之后再不能被使用。

SPI_palloc

SPI_palloc — 在上层执行器上下文中分配内存

大纲

```
void * SPI_palloc(Size size)
```

描述

SPI_palloc在上层的执行器上下文中分配内存。

这个函数只能在连接到SPI时使用。否则，它会报出错误。

参数

Size size

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针

SPI_repalloc

SPI_repalloc — 在上层执行器上下文中重分配内存

大纲

```
void * SPI_repalloc(void * pointer, Size size)
```

描述

SPI_repalloc改变之前用SPI_palloc 分配的内存段的大小。

这个函数不再和普通的repalloc相区别。 保留它只是为了对现有代码保持向后兼容。

参数

void * pointer

指向要改变的现有存储空间的指针

Size size

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针，现有区域的内容会被复制到其中

SPI_pfree

SPI_pfree — 在上层执行器上下文中释放内存

大纲

```
void SPI_pfree(void * pointer)
```

描述

SPI_pfree释放之前使用 SPI_palloc或者 SPI_repalloc分配的内存。

这个函数不再和普通的pfree相区别。 保留它只是为了对现有代码保持向后兼容。

参数

void * pointer

指向要释放的现有存储空间的指针

SPI_copytuple

SPI_copytuple — 在上层执行器上下文中创建一行的拷贝

大纲

```
HeapTuple SPI_copytuple(HeapTuple row)
```

描述

SPI_copytuple在上层执行器上下文中为一行创建 一份拷贝。这通常被用来从一个触发器中返回一个被修改的行。在一个被声 明为返回组合类型的函数中，应使用 SPI_returntuple。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

参数

HeapTuple row

要拷贝的行

返回值

被拷贝的行，或者在出错时返回NULL（错误的内容请参考SPI_result）

SPI_returntuple

SPI_returntuple — 准备把一个元组返回为一个 Datum

大纲

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

描述

SPI_returntuple为一个行在上层执行器上下文中 创建一个拷贝，把它以一种行类型Datum的形式返回。被 返回的指针只需要在返回前通过PointerGetDatum 被转换成Datum。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

注意这应该被用于声明为要返回组合类型的函数。它不能用于触发器，在 触发器中应使用SPI_copytuple来返回一个被修改的行。

参数

HeapTuple row

要被拷贝的行

TupleDesc rowdesc

行的描述符（对大部分有效的缓存，每次都传递相同的描述符）

返回值

指向被拷贝行的HeapTupleHeader，或者在出错时返回NULL（错误的内容请参考SPI_result）

SPI_modifytuple

SPI_modifytuple — 通过替换一个给定行的选定域来创建一行

大纲

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

描述

SPI_modifytuple创建一个新行，其中选定的列 用新值替代，其他列则从输入行中拷贝。输入行本身不被修改。新行被返回在上层的执行器上下文中。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI_result设置为SPI_ERROR_UNCONNECTED。

参数

Relation rel

只被用作该行的行描述符的来源（传递一个关系而不是 一个行描述符是一种令人头痛的设计）。

HeapTuple row

要被修改的行

int ncols

要被修改的列数

int * colnum

一个长度为ncols的数组，包含了要被修改的列号 （列号从 1 开始）

Datum * values

一个长度为ncols的数组，包含了指定列的新值

const char * nulls

一个长度为ncols的数组，描述哪些新值为空值

如果nulls为NULL，那么 SPI_modifytuple假定没有新值为空值。否则， 如果对应的新值为非空，nulls数组的每一项都应 该是' '，而如果对应的新值为空值则为'n'（在 后一种情况中，对应的values项中的新值无关紧 要）。注意nulls不是一个文本字符串，只是一个 数组：它不需要一个'\0'终止符。

返回值

修改过的新行，它被分配在上层的执行器上下文中，或者在出错时返回NULL（错误的内容请参考SPI_result）

出错时，SPI_result被设置如下：

SPI_ERROR_ARGUMENT

如果rel为NULL，或者 row为NULL，或者ncols 小于等于 0，或者colnum为NULL， 或者values为NULL。

SPI_ERROR_NOATTRIBUTE

如果colnum包含一个无效的列号（小于等于 0 或者大于 row中的列数）。

SPI_ERROR_UNCONNECTED

如果SPI不是活跃状态

SPI_freetuple

SPI_freetuple — 释放一个在上层执行器上下文中分配的行

大纲

```
void SPI_freetuple(HeapTuple row)
```

描述

SPI_freetuple释放之前在上层执行器上下文中 分配的一个行。

这个函数不再和普通的heap_freetuple相区别。 保留它只是为了对现有代码保持向后兼容。

参数

HeapTuple row

要释放的行

SPI_freetuptable

SPI_freetuptable — 释放一个由SPI_execute 或者类似函数创建的行集合

大纲

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

描述

SPI_freetuptable释放一个由之前的 SPI 命令 执行函数（例如SPI_execute）创建的行集合。因此， 调用这个函数时，常常使用SPI_tuptable作为 参数。

如果一个使用SPI的C函数需要执行多个命令并且不想保留早期命令的结果，这个 函数就有用了。注意，SPI_finish会释放任何还未释放的 行集合。还有，如果在一个使用SPI的C函数的执行中开始了一个子事务并且后来 被中止，SPI 会自动释放该子事务运行期间创建的任何行集合。

从PostgreSQL 9.3 开始， SPI_freetuptable包含了保护逻辑以避免对于同 一行集的重复删除请求。在以前的发布中，重复的删除将会导致崩溃。

参数

SPITupleTable * tuptable

要释放的行集的指针，NULL 表示什么也不做

SPI_freeplan

SPI_freeplan — 释放一个之前保存的预备语句

大纲

```
int SPI_freeplan(SPIPlanPtr plan)
```

描述

SPI_freeplan释放一个之前由 SPI_prepare返回的或者由 SPI_keepplan、SPI_saveplan 保存的预备语句。

参数

SPIPlanPtr plan

要释放的语句的指针

返回值

成功返回 0；如果plan为 NULL或无效则返回 SPI_ERROR_ARGUMENT

47.4. 事务管理

不能通过SPI_execute这样的SPI函数运行COMMIT和ROLLBACK之类的事务控制命令。不过，也有单独的接口函数允许通过SPI进行事务控制。

如果不考虑被调用的上下文，在任意的用户定义的可从SQL调用的函数中开始以及结束事务通常并不是安全和明智的。例如，一个事务位于一个函数内，而该函数是某个SQL命令中的一个复杂SQL表达式的一部分，这样的事务有可能会导致隐蔽的内部错误或者崩溃。这里介绍的接口函数的主要目的是被过程语言的实现用于支持在CALL命令调用的SQL层过程中进行事务管理，同时把CALL调用的上下文也加以考虑。用C实现的使用SPI的过程可以实现同样的逻辑，但是其细节超出了这份文档的范围。

SPI_commit

SPI_commit — commit the current transaction

大纲

```
void SPI_commit(void)
```

简介

SPI_commit提交当前事务。它近似等效于运行SQL命令COMMIT。在一个事务被提交后，在进一步的数据库动作被执行前必须使用SPI_start_transaction开始一个新的事务。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这个函数。

SPI_rollback

SPI_rollback — abort the current transaction

大纲

```
void SPI_rollback(void)
```

简介

SPI_rollback回滚当前事务。它近似等效于运行SQL命令ROLLBACK。在一个事务被回滚后，在进一步的数据库动作被执行前必须使用SPI_start_transaction开始一个新的事务。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这个函数。

SPI_start_transaction

SPI_start_transaction — start a new transaction

大纲

```
void SPI_start_transaction(void)
```

简介

SPI_start_transaction开始一个新事务。它只能在SPI_commit或SPI_rollback之后被调用，因为在那时没有事务在活动。通常，当一个使用SPI的过程被调用时，会有一个事务已处于活跃状态，因此在关闭当前事务之前尝试启动另一个事务将会导致错误。

只有当SPI连接已经在对SPI_connect_ext的调用中被设置为非原子的情况下才能执行这个函数。

47.5. 数据改变的可见性

下列规则主导了使用 SPI 的函数（或者任何其他 C 函数）中数据改变 的可见性：

- 在一个 SQL 命令的执行期间，该命令所作的任何数据更改对该命令本身 是不可见的。例如，在

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  中，被插入的行对SELECT部分不可见。

- 一个命令 C 所作的更改对所有在 C 之后开始的命令可见，不管它们是否 在 C 之中（在 C 的执行期间）开始还是在 C 完成之后开始。

- 在一个 SQL 命令（或者一个普通函数或者触发器）调用的函数内通过 SPI 执行的命令遵循以上哪条规则取决于传递给 SPI 的读/写标志。以 只读模式执行的命令遵循第一条规则：它们不能看到调用它们的命令的 改变。在读写模式中执行的命令遵循第二条规则：它们能看见目前为止 所有的改变。

- 所有的标准过程语言会基于函数的易变性属性设置 SPI 读写模式。 STABLE和IMMUTABLE函数的命令会以 只读模式完成，而VOLATILE函数的命令会以读写模式 完成。虽然 C 函数的作者可以违反这种习惯，但是最好不要那样做。

下一节包含一个关于这些规则应用的例子：

47.6. 例子

这一节包含了 SPI 用法的一个非常简单的例子。C函数 execq用一个 SQL 命令作为其第一个参数 并且用一个行计数作为第二个参数，使用 SPI_exec执行该命令并且返回被该该命令 处理过的行的数量。你可以在源代码树的 src/test/regress/regress.c和 spi模块中找到 SPI 的更复杂的例子。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;
```

```
PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* 把给定的文本对象转换成一个 C 字符串 */
    command = text_to_cstring(PG_GETARG_TEXT_PP(1));
    cnt = PG_GETARG_INT32(2);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * 如果取出了一些行，通过 elog(INFO) 打印它们。
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        uint64 j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}
```

在把该函数编译到一个共享库中（详见第 38.10.5 节之 后，这样声明该函数：

```
CREATE FUNCTION execq(text, integer) RETURNS int8
    AS 'filename'
    LANGUAGE C STRICT;
```

下面是一个会话实例：

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inserted by execq
INFO:  EXECQ:  1    -- returned by execq and inserted by upper INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, 按照所指定的, 只有一行被插入

 execq
-------
     3              -- 10 只是最大值, 3 是实际的行数
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- 没有行在 a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- 有一行在 in a + 1
(2 rows)

-- 这证明了数据改变可见性规则:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
```

```
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 行 * 1 （第一行中的 x）
 6                  -- 3 rows (2 + 1 被插入) * 2 （第二行中的 x）
(4 rows)                ^^^^^^
                     不同调用中 execq() 的行可见性
```

第 48 章 后台工作者进程

PostgreSQL可以被扩展来在独立进程中运行用户提供的代码。这种进程被postgres启动、停止和监控，这使它们的生命期与服务器的状态紧密联系。这些进程具有选项可以挂接上PostgreSQL的共享内存区域，并且可以从内部连接到数据库。它们也可以连续地运行多个事务，就像一个正常的被客户端连接的服务器进程。同样，通过链接到libpq，它们可以连接到服务器并像一个正常客户端应用工作。

> **警告**
>
> 在使用后台工作者进程时具有相当大的鲁棒性和安全性风险，因为它们由C语言编写，对数据具有无限制的访问权。希望使用包括后台工作者进程在内的模块的管理员必须要极度小心。只有仔细审计过的模块才会被允许运行后台工作者进程。

通过将模块名放在shared_preload_libraries中，可以在PostgreSQL被启动时初始化后台工作者。一个希望运行后台工作者的模块需要通过在其_PG_init()中调用RegisterBackgroundWorker(BackgroundWorker *worker)来注册它。也可以在系统启动后通过调用函数RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)来启动后台工作者。与只能在postmaster内调用的RegisterBackgroundWorker不同，必须从一个常规后端调用RegisterDynamicBackgroundWorker。

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

bgw_name和bgw_type是要被用在日志消息、进程列表和类似环境中的字符串。对于同种类型的所有后台工作者，bgw_type应该相同，例如这样才可能将进程列表中的这些工作组分组。另一方面，bgw_name可以包含有关特定进程的额外信息（通常，bgw_name中的字符串在某种程度上也会包含类型，但是并没有严格的要求）。

bgw_flags是一个按位与的位掩码，它用于指示模块想要的能力。可能的值是：

BGWORKER_SHMEM_ACCESS

请求共享内存访问。没有共享内存使用权的工作者不能访问任何的PostgreSQL共享数据结构，例如重量级或者轻量级锁、共享缓冲区以及该工作者本身想要创建和使用的任何自定义数据结构。

BGWORKER_BACKEND_DATABASE_CONNECTION

请求建立数据库连接的能力，这样它后面可以通过建立起的连接运行事务和查询。一个使用BGWORKER_BACKEND_DATABASE_CONNECTION来连接一个数据库的后台工作者也必须使用BGWORKER_SHMEM_ACCESS挂接到共享内存，否则工作者启动将会失败。

bgw_start_time是服务器状态，在该状态中postgres会启动该进程，它可以是BgWorkerStart_PostmasterStart（在postgres本身完成初始化之后立即启动，这种进程不能使用数据库连接）、BgWorkerStart_ConsistentState（当一个热后备中达到一个一致性状态之后立即启动，允许进程连接到数据库并运行只读查询）和BgWorkerStart_RecoveryFinished（在系统进入到正常读写状态后立即启动）之一。注意后两种值在服务器不是一个热后备的情况下是等同的。注意这种设置仅仅表示何时启动进程，当一个不同状态到达时它们不会停止。

bgw_restart_time是在崩溃情况下postgres启动进程之前等待的时间间隔，以秒计。它可以是任何正值，或者BGW_NEVER_RESTART，表示在出现崩溃后不重启进程。

bgw_library_name是应该在其中定位后台工作者初始入口点的库名称。所指的库将被工作者进程动态载入并且bgw_function_name将被用来标识要调用的函数。如果从核心代码载入一个函数，这必须被设置为"postgres"。

bgw_function_name是一个动态载入库中的一个函数名，该函数将被用作一个新后台工作者的初始入口点。

bgw_main_arg是后台工作者主函数的Datum参数。这个主函数应该有一个单一的Datum类型的参数，并且返回void。bgw_main_arg将被作为参数传递。此外，全局变量MyBgworkerEntry指向注册时传入的BackgroundWorker结构的一份拷贝，工作者会发现检查这个结构会很有用。

在 Windows （以及任何定义了EXEC_BACKEND的地方）上或者动态后台工作者中，用引用的方式传递Datum是不安全的，只有传值才安全。如果要求一个参数，最安全的方式是传递一个int32 或者其他的小型值，并且把它当做共享内存中分配的一个数组的索引来使用。如果被传递的是一个cstring或者text这样的值，那么在新的后台工作者进程中该指针将不会有效。

bgw_extra可以包含要传递给后台工作者的额外数据。与bgw_main_arg不同，这个数据不会被作为一个参数传递给工作者的主函数，而是按照上面所述通过MyBgworkerEntry来访问。

bgw_notify_pid是一个PostgreSQL后端进程的PID，当后台工作者进程启动或者退出时，postmaster会向这个PID所指的进程发送SIGUSR1。 对于在postmaster启动时注册的工作者，它应该为0；或者注册该工作者的后端不希望等待该工作者启动时，它也应该为0。否则，它应该被初始化为MyProcPid。

一旦运行起来，进程可以通过调用BackgroundWorkerInitializeConnection(char *dbname, char *username)或者BackgroundWorkerInitializeConnectionByOid(Oid dboid, Oid useroid)来连接到一个数据库。这使得该进程可以使用SPI接口运行事务和查询。 如果dbname为NULL或者dboid为InvalidOid，该会话没有连接到任何特定数据库，但共享的目录可以被访问。 如果username为NULL或者useroid为InvalidOid，该进程将以在initdb阶段创建的超级用户身份运行。如果BGWORKER_BYPASS_ALLOWCONN被指定为flags，就可以绕过该限制连接不允许用户连接的数据库。在每一个后台进程中，只能调用两者之一，并且只能调用一次，所以不可能切换数据库。

当控制到达后台工作者的主函数时，信号初始会被阻塞，并且必须被它解除阻塞。这是为了允许进程自定义它的信号处理器。在新进程中可以通过调用BackgroundWorkerUnblockSignals来解除对信号的阻塞，还可以通过调用BackgroundWorkerBlockSignals来阻塞信号。

如果一个后台工作者的bgw_restart_time被配置为 BGW_NEVER_RESTART，或者它退出时的退出码为0，又或者它是被 TerminateBackgroundWorker所终止，它将会被postmaster在退出时自动解除 注册。否则，它将在等待通过bgw_restart_time配置的时间段之后被重新启动， 或者在postmaster因为一次后端失败重新初始化集簇时立刻被重启。需要临时禁止执行的后端应该使用 可中断的休眠而不是退出，这可以通过调用WaitLatch()实现。 调用该函数时要确保WL_POSTMASTER_DEATH标志被设置，并且验证在 postgres本身被终止的紧急情况下产生的快速退出返回码。

当一个后台工作者是通过RegisterDynamicBackgroundWorker函数 注册时，后端可以执行该注册以获得有关该工作者的状态信息。希望这样做的后端应该把一个

BackgroundWorkerHandle *的地址作为第二个参数传递给RegisterDynamicBackgroundWorker。如果工作者被成功地注册， 这个指针将被用一个非透明句柄初始化，它之后会被传递给 GetBackgroundWorkerPid(BackgroundWorkerHandle *, pid_t *)或者 TerminateBackgroundWorker(BackgroundWorkerHandle *)。GetBackgroundWorkerPid可以被用来测试工作者的状态：返回值为BGWH_NOT_YET_STARTED表示该工作者还未被postmaster启动； BGWH_STOPPED表示它已经被启动但是不再运行； 而BGWH_STARTED表示它正在运行。在最后一种情况下，PID也将被通过第二个参数返回。 TerminateBackgroundWorker导致postmaster发送SIGTERM 给工作者（如果它在运行），并且在它不再运行时尽快解除注册。

在某些情况下，一个注册后台工作者的进程可能希望等待该工作者启动起来。其实现方式是：把 bgw_notify_pid初始化成MyProcPid并且接着 把注册时得到的BackgroundWorkerHandle *传递给WaitForBackgroundWorkerStartup(BackgroundWorkerHandle *handle, pid_t *)函数。 这个函数将阻塞直到postmaster已经尝试启动该后台工作者，或者直到postmaster死亡。如果后台 工作者正在运行，返回值将是BGWH_STARTED，并且其PID将被写入到所提供的地址。 否则，返回值将是BGWH_STOPPED或者 BGWH_POSTMASTER_DIED。

进程也可以等待一个后台工作者关闭，方法是使用WaitForBackgroundWorkerShutdown(BackgroundWorkerHandle *handle)函数并且传入注册时得到的BackgroundWorkerHandle *。这个函数将阻塞直至后台工作者退出或者postmaster死掉。当后台工作者退出时，返回值是BGWH_STOPPED，如果postmaster死掉则会返回BGWH_POSTMASTER_DIED。

如果一个后台工作者通过服务器编程接口（SPI）用NOTIFY命令发送异步通知，在提交外层事务之后它应该显式地调用ProcessCompletedNotifies，这样通知才能被发送出去。如果一个后台工作者通过SPI使用LISTEN注册为接收异步通知，它将记录那些通知，但是对于工作者来说没有程序化的方式可以拦截以及响应那些通知。

src/test/modules/worker_spi模块包含了一个实例，它展示了一些有用的技巧。

注册的后台工作者的最大数量由max_worker_processes限制。

第 49 章 逻辑解码

PostgreSQL 提供了方法将所执行的修改通过 SQL 以流的方式传送给外部消费者。 这种功能可以被用于多种目的，包括复制方案以及审计。

在流中被送出的更改通过逻辑复制槽标识。

流式传输这些更改的格式由使用的输出插件决定。PostgreSQL 发布中包括了 一个例子插件。可以编写额外的插件来扩展可用的格式选择，而无需修改任何 核心代码。每一个输出插件都能访问每一个由 INSERT产生的新行以及每一个由 UPDATE创建的新行版本。UPDATE和DELETE 的旧行版本的可用性取决于配置的复制标识（见 REPLICA IDENTITY）。

可以通过流复制协议（见第 53.4 和 第 49.3 节或者通过 SQL 调用函数（第 49.4 节来接收流式传送的更改。也可以编写额外的 接收复制槽输出的模块而无需修改核心代码（ 第 49.7 节。

49.1. 逻辑解码的例子

下面的例子演示了使用 SQL 接口控制逻辑解码。

在你能使用逻辑解码之前，你必须设置wal_level为 logical，并且max_replication_slots必须至少被设置为 1。然后，你应该作为一个超级用户连接到目标数据库（在下面 的例子中是postgres）。

```
postgres=# -- 使用输出插件'test_decoding'创建一个名为'regression_slot'的槽
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot',
 'test_decoding');
    slot_name    |    lsn
-----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn,
 confirmed_flush_lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active | restart_lsn |
 confirmed_flush_lsn
-----------------+---------------+-----------+----------+--------+-------------
+-----------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   |
 0/16A4440
(1 row)

postgres=# -- 目前还看不到更改
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL,
 NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL 没有被复制，因此你将看到的东西只有事务
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL,
 NULL);
    lsn    |  xid  |     data
-----------+-------+--------------
```

```
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- 一单读到更改，它们会被消费掉并且不会在一个后续调用中被发出：
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL,
 NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# BEGIN;
postgres=# INSERT INTO data(data) VALUES('1');
postgres=# INSERT INTO data(data) VALUES('2');
postgres=# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL,
 NULL);
    lsn     |  xid  |                          data
------------+-------+--------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- 你也可以不消费更改而在更改流中先看一看
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL,
 NULL);
    lsn     |  xid  |                          data
------------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- 接下来对 pg_logical_slot_peek_changes() 的调用再次返回相同的更改
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL,
 NULL);
    lsn     |  xid  |                          data
------------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- 可以向输出插件传递选项来影响格式化
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL,
 NULL, 'include-timestamp', 'on');
    lsn     |  xid  |                          data
------------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)
```

```
postgres=# -- 当不再需要一个槽后记住销毁它以停止消耗服务器资源：
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
--------------------------

(1 row)
```

下面的例子展示了如何在流复制协议上使用 PostgreSQL 发布所包括的程序pg_recvlogical来控制逻辑解码。这要求设置客户端认证以允许复制连接（见第 26.2.5.1 节，并且把max_wal_senders设置成足够高以允许一个额外的连接。

```
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

49.2. 逻辑解码概念

49.2.1. 逻辑解码

逻辑解码是一种将对数据库表的所有持久更改抽取到一种清晰、易于理解的格式 的处理，这种技术允许在不了解数据库内部状态的详细知识的前提下解释该格式。

在PostgreSQL中，逻辑解码通过解码 预写式日志的内容来实现，预写式日志描述了存储 层面上的更改，而逻辑解码则会把更改解码成一种应用相关的形式，例如一个元组 流或者 SQL 语句流。

49.2.2. 复制槽

在逻辑复制的环境下，一个槽表示一个更改流，这些更改可以在客户机上以它们在原服务器上产生的顺序被重播。每一个流从一个单一数据库中流式传送更改序列。

> **注意**
>
> PostgreSQL也有流复制槽（见第 26.2.5 节，但是它们的使用有所不同。

一个复制槽在一个PostgreSQL集簇的所 有数据库之间具有一个唯一的标识符。槽在使用它们的连接之间保持独立并且 对于崩溃是安全的。

在常规操作中，一个逻辑槽只会把每次更改发出一次。只有在检查点时才会持久化每一个槽的当前位置，因此如果发生崩溃，槽可能会回到一个较早的 LSN，这会导致服务器重启时重新发送最近的更改。逻辑解码客户端负责避免多次处理同一消息导致的副作用。客户端可能会希望在解码时记录它们看到的最新的 LSN，并且跳过任何从该 LSN 解码得到的重复数据或者（使用复制协议时的）请求，而不是让服务器来决定开始点。复制进度跟踪特性就是为此服务的，请参考复制源头。

对于同一个数据库可能会存在多个独立的槽。每一个槽有自己的状态，允许不 同的消费者从该数据库的更改流中的不同点开始接收更改。对于大多数应用， 每一个消费者都将要求一个单独的槽。

逻辑复制槽完全不知道接收者的状态。甚至可能会有多个不同的接收者在不同 时间使用同一个槽，它们将只是从上一个接收者停止消费更改的地方开始得到 更改。但在任一给定时刻，只有一个接收者可以从一个槽中消费更改。

> **小心**
>
> 复制槽可以在崩溃时保持，并且不知道其消费者的状态。即便没有连接使用它们， 它们也将阻止移除所需的资源。这会消耗存储，因为只要还有一个复制槽需要， WAL 和来自于系统目录的行就不能被VACUUM移除。在极端情况下这会导致数据库关闭以防止事务ID回卷（见第 24.1.5 节。因此如果不再需要一个槽，那就应该删除它。

49.2.3. 输出插件

输出插件将数据从预写式日志的内部表示转换成复制槽的消费者所需的格式。

49.2.4. 导出快照

当使用流复制接口创建一个新的复制槽时（见CREATE_REPLICATION_SLOT），一个快照将被导出（见 第 9.26.5 节，在它所显示的 数据库状态之后所有的更改都将被包括在更改流中。通过使用 SET TRANSACTION SNAPSHOT读取槽被创建时的数据库状态，这可以用来创建 一个新的复制。然后这个事务可以被用来及时转储那一点的数据库状态，它后来 可以被槽的内容更新而不丢失任何更改。

并非总能够创建快照。特别是在连接到热备时，快照创建将会失败。不要求快照导出的应用可以用NOEXPORT_SNAPSHOT选项来抑制它。 option.

49.3. 流复制协议接口

命令

- CREATE_REPLICATION_SLOT slot_name LOGICAL output_plugin
- DROP_REPLICATION_SLOT slot_name [WAIT]
- START_REPLICATION SLOT slot_name LOGICAL ...

被用来创建、删除以及流式传送一个复制槽。这些命令只能在一个复制连接上使用。 它们不同通过 SQL 使用。这些命令的详情请见 第 53.4 节

命令pg_recvlogical可以被用来控制一个流复制连接上的逻辑 解码（它在内部使用上述命令）。

49.4. 逻辑解码的 SQL 接口

与逻辑解码互动的 SQL 层 API 详见第 9.26.6 节

同步复制（见第 26.2.8 节只在使用流复制接口的复制槽上 支持。函数接口以及额外的、非核心的接口不支持同步复制。

49.5. 与逻辑解码相关的系统目录

pg_replication_slots 视图和 pg_stat_replication 视图分别提供了有关复制槽和流复制连接的当前状态的信息。这些视图适用于物理和逻辑复制。

49.6. 逻辑解码输出插件

可以在 PostgreSQL 源码树的 contrib/test_decoding 子目录中找到一个输出插件的例子。

49.6.1. 初始化函数

一个输出插件是通过动态载入一个以输出插件名称作为基础名称的共享库来载入的。 将使用普通的库搜索路径来定位该库。为了提供所要求的输出插件回调并且指示该 库确实是一个输出插件，需要提供一个名为 _PG_output_plugin_init的函数。这个函数会被传入一个结构，其中被填充了各个动作的回调函数指针。

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

回调函数begin_cb、change_cb 以及commit_cb是必需的，而startup_cb、filter_by_origin_cb、truncate_cb 和shutdown_cb是可选的。如果没有设置truncate_cb但是要对一个TRUNCATE进行编码，则该动作将被忽略。

49.6.2. 能力

要解码、格式化并且输出更改，输出插件可以使用大部分后端的标准功能，包括调用 输出函数。只要访问的关系是initdb在 pg_catalog模式中创建的或者被使用

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

标记为用户提供的系统表，就允许对关系的只读访问。任何导致事务 ID 分配的动作 都被禁止。其中包括写表、执行 DDL 更改以及调用txid_current()。

49.6.3. 输出模式

输出插件回调可以以近乎任意格式向消费者传递数据。对于某些用例，例如通过 SQL 查看更改，以可能包含任何数据的数据类型（例如bytea）返回数据 可能会很麻烦。如果输出插件只输出服务器编码的文本数据，它可以在 启动回调中通过把OutputPluginOptions.output_type设 置为OUTPUT_PLUGIN_TEXTUAL_OUTPUT替代OUTPUT_PLUGIN_BINARY_OUTPUT来声明这一点。在这种情况下， 所有的数据必须是属于服务器的编码，这样一个text数据就能包含它。在 启用了断言的编译中会检查这一点。

49.6.4. 输出插件回调

一个输出插件需要提供一些回调，它通过它们得到有关更改发生的通知。

并发事务以提交顺序被解码，并且只有属于特定事务的更改会在 begin和commit回调之间被解码。被显式 或隐式回滚的事务不会被解码。成功的检查点被折叠到包含它们的事务中，并且 保持它们在该事务中被执行的顺序。

> **注意**
>
> 只有已经被安全地刷入磁盘的事务将会被解码。当 synchronous_commit被设置为off 时，这会导致一个COMMIT在随后的pg_logical_slot_get_changes()中不会立即被解码。

49.6.4.1. 启动回调

只要一个复制槽被创建或者被要求流式传送更改，可选的 startup_cb回调就会被调用，不管有多少更改准备输出。

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

当复制槽被创建时，is_init参数将为真，否则为假。 options指向一个输出插件可以设置的选项 的结构：

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool        receive_rewrites;
} OutputPluginOptions;
```

output_type必须被设置为 OUTPUT_PLUGIN_TEXTUAL_OUTPUT 或者OUTPUT_PLUGIN_BINARY_OUTPUT。另见 第 49.6.3 节如果receive_rewrites为真，还将为在某些DDL操作期间的堆重写造成的更改调用输出插件。这些是处理DDL复制的插件感兴趣的事情，但是它们要求特殊的处理。

启动回调应该验证出现在 ctx->output_plugin_options中的选项。如果输出插件 需要有一个状态，它可以使用 ctx->output_plugin_private来存储之。

49.6.4.2. 关闭回调

只要一个之前活跃的复制槽不再使用，就会调用可选的 shutdown_cb回调，它可以被用来释放输出插件 私有的资源。该槽并不一定需要被删除，只要其中的流被停止即可。

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

49.6.4.3. 事务开始回调

只要一个已提交事务的开始动作被解码，就会调用必须提供的 begin_cb回调。被中止的事务及其内容不会被解码。

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

txn参数包含有关该事务的元信息，例如该 事务被提交的时间戳以及该事务的 XID。

49.6.4.4. 事务结束回调

只要一个已提交事务的提交动作被解码，就会调用必须提供的 commit_cb回调。在此之前，如果有任何被修改 的行，将为所有被修改的行调用change_cb回调。

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

49.6.4.5. 更改回调

对于一个事务中的每一个行修改，都将调用必须提供的 change_cb回调，这种修改可能是一个 INSERT、UPDATE或者 DELETE。即使原始命令一次修改了多行，该回调也会 为其中的每一行调用一次。

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

ctx和txn参数与 begin_cb和commit_cb 回调具有相同的内容，但是额外多出一个关系描述符relation指向该行所属的关系以及一个结构 change描述被传入的行修改。

> **注意**
>
> 只有没有被标记为“不做日志”（见 UNLOGGED）并且非临时（见 TEMPORARY or TEMP）的用户定义表中的 更改才能用逻辑解码抽取。

49.6.4.6. 截断回调

truncate_cb回调会为一个TRUNCATE命令被调用。

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         int nrelations,
                                         Relation relations[],
                                         ReorderBufferChange *change);
```

参数类似于change_cb回调。不过，由于通过外键连接起来的表上的TRUNCATE动作需要一起被执行，这个回调会接收到一个关系的数组而不是单个关系。详情请见对TRUNCATE语句的介绍。

49.6.4.7. 源过滤器回调

可选的filter_by_origin_cb回调被用来 决定从origin_id重放的数据是否是 输出插件感兴趣的数据。

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext
 *ctx,
                                               RepOriginId origin_id);
```

ctx参数具有和其他回调相同的内容。 对这个回调只有复制源的信息可用。要标志传进来的节点上发生的 更改是无关的，返回真，这会导致这些更改被过滤掉，否则返回假。 对于被过滤掉的事务和更改将不会调用其他回调。

在实现级联或者多向复制方案时，这个回调可以派上用场。用源头 过滤允许阻止在这样的设置下来回地复制同样的更改。虽然事务和 更改也携带了有关源头的信息，通过这个回调过滤明显更有效些。

49.6.4.8. 通用消息回调

只要一个逻辑解码消息被解码出来，可选的message_cb回调就会被调用。

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

txn参数包含关于该事务的元信息，如被提交的时间戳和 XID。不过要注意，当消息是非事务性的并且记录该消息的事务中还没有被分配 XID 时，这个参数可以为 NULL。lsn是该消息的 WAL 位置。transactional说明该消息是否为事务性的。prefix是一个任意的空终结的前缀，它当前插件被用来标识感兴趣的消息。最后的message参数保存着大小为message_size的消息。

应该格外小心确保输出插件用于标识感兴趣消息的前缀是唯一的。建议使用扩展或者输出插件本身的名称。

49.6.5. 用于产生输出的函数

在begin_cb、commit_cb或者 change_cb回调中，为了实际产生输出， 输出插件可以把数据写入到ctx->out中的 StringInfo输出缓冲区中。在写出到输出缓冲区之前，必须先 调用OutputPluginPrepareWrite(ctx, last_write)，在完 成写入到缓冲区后，必须调用OutputPluginWrite(ctx, last_write)来执行写出。 last_write指出一次特定的写出是否为该回调的最后 一次写出。

下面的例子展示了如何把数据输出给一个输出插件的消费者：

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

49.7. 逻辑解码输出写入器

可以为逻辑解码增加更多输出方法。详情可见 src/backend/replication/logical/logicalfuncs.c。 本质上，需要提供三个函数：一个读取 WAL，一个准备写输出，另一个写输出（ 见第 49.6.5 节。

49.8. 逻辑解码的同步复制支持

逻辑解码可以被用来构建 同步复制方案，该方案 具有和流复制的同步复制 相同的用户接口。要这样做，流复制接口（见 第 49.3 节必须被用来流式传出数据。 正如流复制客户端所作的一样，逻辑解码的客户端必须发出 后备机状态更新 (F) （见第 53.4 节消息。

> **注意**
>
> 一个通过逻辑解码接收更改的同步复制机将工作在一个单一数据库的范围内。因为与之相反，synchronous_standby_names 当前是服务器范围的，这意味着如果有多于一个数据库被活跃地使用，这种技术将 无法正常工作。

第 50 章 复制进度追踪

复制源是为了更容易地在逻辑解码 上实现逻辑复制解决方案而设计。它们提供了对两种常见问题的解决方案：

- 如何安全地跟踪复制进度？
- 如何基于一行的来源更改复制行为？例如，阻止双向复制 设置中的循环

复制源只有两个属性，名称和 OID。名称应该可以被用来在系统间引用该源， 它是一种自由形式的文本。为了避免复制源之间的冲突，可以 在复制源的名称前加上复制解决方案的名称。在空间效率很重要的情况下，OID 被用来避免不得不存储长版本。OID 不能在系统间被共享。

可以使用函数 pg_replication_origin_create()创建复制源，使用函数 pg_replication_origin_drop()删除复制源，并且在系统目录 pg_replication_origin中查看复制源。

构建一套复制解决方案的一个重要部分是以一种安全的方式跟踪重放进度。当应用 过程或者整个集簇死掉时，需要能够找出数据被成功地复制到了什么地方。对此的 简单解决方案（例如为每一个被重放的事务更新一个表行）有运行时负荷和数据库 膨胀的问题。

通过使用复制源，一个会话可以被标记为从一个远程节点重放（使用 pg_replication_origin_session_setup()函数）。此外， 可以使用 pg_replication_origin_xact_setup()以每一个事务为基础配置每一个源事务的 LSN和提交时间戳。如果完成这样的配置， 复制过程将保持在一种对崩溃安全的方式中。所有复制源的重放进度可以在 pg_replication_origin_status 视图中看到。一个源的进度（例如在继续复制时）可以使用 pg_replication_origin_progress()（用于任何源）或者 pg_replication_origin_session_progress() （用于在当前会话中配置的源）获得。

在比从一个系统复制到另一个系统更复杂的复制拓扑中，另一个问题是很难避免再次 复制已经被重放的行。这可能导致复制中的循环和低效。复制源提供了一种可选的机制 来识别和阻止这种问题。在使用前一段提到的函数配置时，每一个被传递 给输出插件回调（见第 49.6 节的由该会话 生成的改变和事务会被标记上该会话的复制源。这使得可以在输出插件中以不同的方式 对待它们，例如忽略除本地生成的行之外的所有行。此外，filter_by_origin_cb回调可以被用来基于来源过滤逻辑 解码改变流。虽然灵活性较低，通过这种回调进行过滤比用输出插件过滤效率更高。

部分 VI. 参考

这份参考中的条目意欲提供关于相应主题的权威、完整和正式的总结。关于使用PostgreSQL的更多信息（以叙述、教程或例子的形式）可以在本书的其他部分找到。见每个参考页面上列出的交叉引用。

这些参考条目也在传统“man”页面中可用。

目录

SQL 命令

这部分包含PostgreSQL支持的SQL命令的参考信息。每条命令的标准符合和兼容的信息可以在相关的参考页中找到。

目录

ABORT

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ]
```

描述

ABORT回滚当前事务并且导致由该事务所作的所有更新被丢弃。这个命令的行为与标准SQL命令ROLLBACK的行为一样，并且只是为了历史原因存在。

参数

WORK
TRANSACTION

可选关键词。它们没有效果。

注解

使用COMMIT成功地终止一个事务。

在一个事务块之外发出ABORT会发出一个警告消息并且不会产生效果。

例子

中止所有更改：

```
ABORT;
```

兼容性

这个命令是一个因为历史原因而存在的PostgreSQL扩展。ROLLBACK是等效的标准 SQL 命令。

参见

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — 更改一个聚集函数的定义

大纲

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
                OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

其中 aggregate_signature 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

描述

ALTER AGGREGATE更改一个聚集函数的定义。

要使用ALTER AGGREGATE，你必须拥有该聚集函数。 要更改一个聚集函数的模式，你还必须具有新模式上的 CREATE特权。要修改拥有者，你还必须是新拥有角色 的一个直接或者间接成员，并且那个角色必须在聚集函数的模式上拥有 CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该聚集函数来做任何你不能做的事情。不过，一个超级用户可以更改任何聚集函数的所有权）。

参数

name

一个现有聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或VARIADIC。 如果省略，默认为IN。

argname

一个参数的名称。注意ALTER AGGREGATE 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数的数据类型。

argtype

聚集函数要在其上操作的输入数据类型。要引用一个零参数聚集函数，在参数 说明列表的位置写上*。要引用一个有序集聚集函数，在直接参数 说明和聚集参数说明之间写上ORDER BY。

new_name

聚集函数的新名称。

new_owner

聚集函数的新拥有者。

new_schema

聚集函数的新模式。

注解

引用有序集聚集的推荐语法是在直接参数说明和聚集参数说明之间写上 ORDER BY，这和CREATE AGGREGATE 中的风格相同。不过，省略ORDER BY并且只把直接和 聚集参数说明放到一个单一列表中也是可以的。在这种简写形式中，如果 在直接和聚集参数列表中都使用了VARIADIC "any"，只用 写一次VARIADIC "any"。

示例

要把用于类型integer的聚集函数 myavg重命名为my_average：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

要把用于类型integer的聚集函数 myavg的拥有者改为joe：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把带有float8类型直接参数和integer 类型聚集参数的有序集聚集mypercentile 移动到 模式myschema中：

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

这也能行：

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

兼容性

在 SQL 标准中没有ALTER AGGREGATE语句。

另见

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — 更改一个排序规则的定义

大纲

```
ALTER COLLATION name REFRESH VERSION

ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER COLLATION name SET SCHEMA new_schema
```

描述

ALTER COLLATION更改一个排序规则的定义。

你必须拥有要对其使用ALTER COLLATION的排序规则。要更改拥有者，你必须是新的拥有角色的直接或者间接成员，并且该角色必须在排序规则的模式上具有CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该排序规则来做任何你不能做的事情。不过，一个超级用户可以更改任何排序规则的所有权）。

参数

name

一个现有排序规则的名称（可以是模式限定的）。

new_name

排序规则的新名称。

new_owner

排序规则的新拥有者。

new_schema

排序规则的新模式。

REFRESH VERSION

更新排序规则的版本。 参阅下面的注意。

注意

使用ICU库提供的排序规则时，创建排序规则对象时，系统目录中会记录排序规则的特定ICU版本。 使用排序规则时，将根据记录的版本检查当前版本，并在发生不匹配时发出警告，例如：

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but
 the operating system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION
 pg_catalog."xx-x-icu" REFRESH VERSION, or build PostgreSQL with the right
 library version.
```

排序规则定义的更改会导致索引损坏和其他问题，因为数据库系统依赖于具有特定排序顺序的存储对象。 通常，应该避免这种情况，但它可以在合法的情况下发生，例如使用pg_upgrade 升级到与更新版本的ICU链接的服务器二进制文件。发生这种情况时， 应该重建所有依赖于该排序规则的对象，例如，使用REINDEX。 完成后，使用命令ALTER COLLATION ... REFRESH VERSION可以刷新排序规则版本。 这将更新系统目录以记录当前的排序规则版本，并会使警告消失。请注意， 这实际上并不检查是否所有受影响的对象都已正确重建。

以下查询可用于识别当前数据库中需要刷新的所有排序规则以及依赖它们的对象：

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

例子

要把排序规则de_DE重命名为german：

```
ALTER COLLATION "de_DE" RENAME TO german;
```

要把排序规则en_US的拥有者改成joe：

```
ALTER COLLATION "en_US" OWNER TO joe;
```

兼容性

在 SQL 标准中没有ALTER COLLATION语句。

参见

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — 改变一个转换的定义

大纲

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

描述

ALTER CONVERSION改变一个转换的定义。

你必须拥有要对其使用ALTER CONVERSION的转换。要更改拥有者，你必须是新的拥有角色的直接或者间接成员，并且该角色必须在转换的模式上具有CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该转换来做任何你不能做的事情。不过，一个超级用户可以更改任何转换的所有权）。

参数

name

一个现有转换的名称（可以是模式限定的）。

new_name

转换的新名称。

new_owner

转换的新拥有者。

new_schema

转换的新模式。

例子

要把转换iso_8859_1_to_utf8重命名为latin1_to_unicode：

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

要把转换iso_8859_1_to_utf8的拥有者改成joe：

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

兼容性

在 SQL 标准中没有ALTER CONVERSION语句。

参见

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — 更改一个数据库

大纲

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]

这里 option 可以是：

    ALLOW_CONNECTIONS allowconn
    CONNECTION LIMIT connlimit
    IS_TEMPLATE istemplate

ALTER DATABASE name RENAME TO new_name

ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL
```

描述

ALTER DATABASE更改一个数据库的属性。

第一种形式更改某些针对每个数据库的设置（详见下文）。只有数据库拥有者或者超级用户可以更改这些设置。

第二种形式更改数据库的名称。只有数据库拥有者或者超级用户可以重命名一个数据库，非超级用户拥有者还必须拥有CREATEDB特权。当前数据库不能被重命名（如果你需要这样做请连接到一个不同的数据库）。

第三种形式更改数据库的拥有者。要修改拥有者，你必须拥有该数据库并且也是新拥有角色的一个直接或间接成员，并且你必须具有CREATEDB特权（注意超级用户自动拥有所有这些特权）。

第四种形式更改数据库的默认表空间。只有数据库拥有者或超级用户能够这样做，你还必须对新表空间具有创建特权。这个命令会在物理上移动位于该数据库旧的默认表空间中的任何表或索引到新的表空间中。新的默认表空间对于这个数据库必须是空的，并且不能有人可以连接到该数据库。在非默认表空间中的表和索引不受影响。

剩下的形式更改用于一个PostgreSQL数据库的运行时配置变量的会话默认值。接下来只要一个新的会话在该数据库中开始，指定的值就会成为该会话的默认值。数据库相关的默认值会覆盖出现在postgresql.conf中或者从postgres命令行接收到的设置。只有数据库拥有者或超级用户可以更改一个数据库的会话默认值。一些变量不能用这种方式设置或者只能由超级用户更改。

参数

name

要被修改属性的数据库名称。

allowconn

如果为假，则没有人能连接到这个数据库。

connlimit

与这个数据库可以建立多少个并发连接。-1 表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从这个数据库进行克隆。如果为假，则只有超级用户或者这个数据库的拥有者可以克隆它。

new_name

数据库的新名称。

new_owner

数据库的新拥有者。

new_tablespace

数据库的新默认表空间。

这种形式的命令不能在事务块内执行。

configuration_parameter
value

将这个数据库的指定配置参数的会话默认值设置为给定值。如果value是DEFAULT，或者等效地使用了RESET，数据库相关的设置会被移除，因此系统范围的默认设置将会在新会话中继承。使用RESET ALL可清除所有数据库相关的设置。SET FROM CURRENT会保存该会话的当前参数值作为数据库相关的值。

更多关于允许的参数名称和值的信息可参考SET和第 19 章

注解

也可以把一个会话的默认值绑定到一个特定角色而不是一个数据库，见ALTER ROLE。如果有冲突，角色相关的设置会覆盖数据库相关的值。

例子

要在数据库test中默认禁用索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

兼容性

ALTER DATABASE语句是一个PostgreSQL扩展。

参见

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — 定义默认访问特权

大纲

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke
```

其中abbreviated_grant_or_revoke是下列之一：

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

描述

ALTER DEFAULT PRIVILEGES允许你设置将被应用于未来要创建的对象的特权（它不会影响分配给已经存在的对象的特权）。当前，只能修改用于模式、表（包括视图和外部表）、序列、函数和类型（包括域）的特权。其中，可设置权限的函数包括聚集函数和过程函数。当这个命令应用于函数时，单词FUNCTIONS和ROUTINES是等效的。（推荐使用ROUTINES，因为它是用来囊括函数和过程的一个标准术语。在较早的PostgreSQL发行版中，只允许单词FUNCTIONS。无法为函数或过程单独设置默认特权。）

你只能改变你自己或者你属于其中的角色所创建的对象的默认特权。这些特权可以对全局范围设置（即对当前数据库中创建的所有对象），或者只对在指定模式中创建的对象设置。被指定的针对模式的默认特权会被增加到用于特定数据类型的全局默认特权中。

如GRANT中所述，用于任何对象类型的默认特权通常会把所有可授予的权限授予给对象拥有者，并且也可能授予一些特权给PUBLIC。不过，这种行为可以通过使用ALTER DEFAULT PRIVILEGES修改全局默认特权来改变。

参数

target_role

一个现有角色的名称，当前角色是它的一个成员。如果FOR ROLE被忽略，将假定为当前角色。

schema_name

一个现有模式的名称。如果被指定，以后在那个模式中创建的对象的默认特权会被修改。如果IN SCHEMA被忽略，全局默认特权会被修改。 在使用ON SCHEMAS时，不能使用IN SCHEMA， 因为模式不能嵌套。

role_name

要为其授予或者收回特权的一个现有角色的名称。这个参数以及所有abbreviated_grant_or_revoke中的其他参数会按照GRANT或者REVOKE中描述的方式运作，不过这里是为一整类的对象而不是特别指定的对象设置权限。

注解

使用psql的\ddp命令可以获得关于默认特权的现有分配信息。特权值的含义和GRANT下为\dp命令描述的相同。

如果你希望删除一个默认特权被修改的角色，有必要撤销其默认特权上的改变或者使用DROP OWNED BY来为该角色去除默认特权项。

例子

为你后续在模式myschema中创建的所有表（和视图）授予 SELECT 特权，并且也允许角色webuser向它们之中 INSERT 数据：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

撤销上面的动作，因此后续创建的表不会有任何不寻常的权限：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM
 webuser;
```

为后续由角色admin创建的所有函数移除通常在函数上会授予的公共 EXECUTE 权限：

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

兼容性

在 SQL 标准中没有ALTER DEFAULT PRIVILEGES语句。

参见

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN — 更改一个域的定义

大纲

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
     RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

描述

ALTER DOMAIN更改一个现有域的定义。有几种形式：

SET/DROP DEFAULT

这些形式设置或者移除一个域的默认值。注意默认值只会应用到后续的 INSERT命令，它们不影响使用该域的已经 存在于表中的行。

SET/DROP NOT NULL

这些形式更改一个域是被标记为允许 NULL 值还是拒绝 NULL 值。只有当使用该 域的列不包含空值时才能SET NOT NULL。

ADD domain_constraint [NOT VALID]

这种形式使用和CREATE DOMAIN相同的语法为一个 域增加一个新的约束。当一个新的约束被增加到一个域时，所有使用该域的列 都会被根据新加的约束进行检查。可以通过增加使用 NOT VALID选项的新约束来抑制这类检查，而该约束则可 以在以后使用 ALTER DOMAIN ... VALIDATE CONSTRAINT 变为可用。新插入和更新的行总是会被根据所有约束进行检查（包括被标记为 NOT VALID的约束）。只有CHECK约束接受 NOT VALID。

DROP CONSTRAINT [IF EXISTS]

这种形式删除一个域上的约束。如果指定了IF EXISTS并且 约束不存在，不会抛出错误。在这种情况下会转而发出一个提示。

RENAME CONSTRAINT

这种形式更改一个域上的一个约束的名称。

VALIDATE CONSTRAINT

这种形式验证一个之前作为NOT VALID增加的约束，也就是说 验证使用该域的列中所有数据满足指定的约束。

OWNER

这种形式更改域的拥有者为指定用户。

RENAME

这种形式更改域的名称。

SET SCHEMA

这种形式更改域的模式。任何与该域关联的约束也被会移动到新的模式中。

要使用ALTER DOMAIN，你必须拥有该域。要更改一个域的模式， 你还必须具有新模式上的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建域做不到的事情。不过，一个超级用户怎么都能更改任何域的所有权。）。

参数

name

要修改的一个现有域的名称（可能是模式限定的）。

domain_constraint

用于该域的新域约束。

constraint_name

要删除或重命名的一个现有约束的名称。

NOT VALID

不为约束的合法性验证现有的列数据。

CASCADE

自动删除依赖于该约束的对象，并且接着删除依赖于那些对象的 所有对象（见第 5.13 节。

RESTRICT

如果有任何依赖对象则拒绝删除该约束。这是默认行为。

new_name

域的新名称。

new_constraint_name

约束的新名称。

new_owner

域的新拥有者的用户名。

new_schema

域的新模式。

注解

当前，如果域或者任何衍生域被数据库中的任意表的一个容器类型　列（组合、数组、范围类型的列）使用，ALTER DOMAIN ADD CONSTRAINT、ALTER DOMAIN VALIDATE CONSTRAINT和ALTER DOMAIN SET NOT NULL将会失败。这些命令最终将 会被改进成能够对这类嵌套值进行约束验证。

示例

要把一个NOT NULL约束加到一个域：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

要从一个域中移除一个NOT NULL约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

要把一个检查约束增加到一个域：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

要从一个域移除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

要重命名一个域上的一个检查约束：

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

要把域移动到一个不同的模式：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

兼容性

ALTER DOMAIN conforms to the SQL standard, except for the 除OWNER、RENAME、SET SCHEMA 以及VALIDATE CONSTRAINT变体之外（它们是 PostgreSQL的扩展）， ALTER DOMAIN符合SQL标准。 ADD CONSTRAINT变体的NOT VALID子句也是一个 PostgreSQL扩展。

另见

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — 更改一个事件触发器的定义

大纲

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

描述

ALTER EVENT TRIGGER更改一个现有事件触发器 的属性。

必须作为一个超级用户才能修改一个时间触发器。

参数

name

要修改的现有触发器的名称。

new_owner

该事件触发器的新拥有者的用户名。

new_name

该事件触发器的新名称。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置事件触发器的触发。一个被禁用的触发器对系统来说仍然是可知的， 但是当期触发事件发生时却不会执行它。另见 session_replication_role。

兼容性

在 SQL 标准中没有ALTER EVENT TRIGGER语句。

另见

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — 更改一个扩展的定义

大纲

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object
```

其中 member_object 是：

```
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name
```

并且 aggregate_signature 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

描述

ALTER EXTENSION更改一个已安装扩展的定义。 有几种子形式：

UPDATE

这种形式把该扩展更新到一个新版本。该扩展必须提供一个适当的更新 脚本（或者一系列脚本）来把当前已安装的版本修改成所要求的版本。

SET SCHEMA

这种形式把该扩展的对象移动到另一个模式中。要使这个命令成功，该扩展 必须是可重定位的。

ADD member_object

这种形式把一个现有的对象加入到该扩展中。这主要对扩展更新脚本有用。 该对象后续将被当作该扩展的一个成员。尤其是该对象只有通过删除扩展 才能删除。

DROP member_object

这种形式从该扩展移除一个成员对象。这主要对扩展更新脚本有用。只有 撤销该对象与其扩展之间的关联后才能删除该对象。

关于这些操作详见第 38.16 节

要使用ALTER EXTENSION，你必须拥有该扩展。 ADD/DROP形式还要求被增加/删除对象的所有权。

参数

name

一个已安装扩展的名称。

new_version

想要得到的该扩展的新版本。这可以写成一个标识符或者一个字符串。如果没有指定，ALTER EXTENSION UPDATE会尝试更新到该扩展的控制文件中的 默认版本。

new_schema

该扩展的新模式。

object_name
aggregate_name
function_name
operator_name
procedure_name
routine_name

要从该扩展增加或者移除的对象的名称。表、聚集、域、外部表、函数、 操作符、操作符类、操作符族、过程、例程、序列、文本搜索对象、类型和视图的名称 可以被模式限定。

source_type

该转换的源数据类型的名称。

target_type

该转换的目标数据类型的名称。

argmode

一个函数、过程或者聚集参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认值是 IN。注意，ALTER EXTENSION 并不真正关心OUT参数，因为决定该函数的身份时只需要输入 参数。因此列出IN、INOUT和 VARIADIC参数足矣。

argname

一个函数、过程或者聚集参数的名称。注意， ALTER EXTENSION并不真正关心参数名称，因为 决定该函数的身份时只需要参数的数据类型。

argtype

一个函数、过程或者或聚集参数的数据类型。

left_type
right_type

该操作符参数的数据类型（可以用模式限定）。对一个前缀或后缀操作符的缺失的 参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

示例

把hstore扩展更新到版本 2.0：

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

把hstore扩展的模式更改到utils：

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

要向hstore扩展增加一个现有函数：

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

兼容性

ALTER EXTENSION是一个PostgreSQL 扩展。

另见

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — 更改一个外部数据包装器的定义

大纲

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

描述

ALTER FOREIGN DATA WRAPPER更改一个 外部数据包装器的定义。该命令的第一种形式用于更改外部数据包装器的 支持函数或者一般选项（至少要求一个子句）。第二种形式更改外部数据包 装器的拥有者。

只有超级用户能修改外部数据包装器。此外，只有超级用户能够拥有外部数 据包装器。

参数

name

一个已有的外部数据包装器的名称。

HANDLER handler_function

为外部数据包装器指定一个新的处理器函数。

NO HANDLER

用于指定该外部数据包装器不再具有一个处理器函数。

注意使用没有处理器的外部数据包装器的外部表不能被访问。

VALIDATOR validator_function

为外部数据包装器指定一个新的验证器函数。

注意，新的验证器可能会认为该外部数据包装器或者依赖于它的独立服务器 的已有选项、用户映射、外部表无效。PostgreSQL 不会做这种检查。在使用修改过的外部数据包装器之前确认这些选项正确是 用户的责任。不过，在这个ALTER FOREIGN DATA WRAPPER命令中指定的选项将会被使用新的验证器检查。

NO VALIDATOR

用于指定该外部数据包装器不再拥有一个验证器函数。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

为该外部数据包装器更改选项。ADD、SET 以及DROP指定要被执行的动作。如果没有显式地指定操作， 将假定为ADD。选项名称必须唯一，选项名称和值（如果有） 也会使用该外部数据包装器的验证器函数来验证。

new_owner

该外部数据包装器的新拥有者的用户名。

new_name

该外部数据包装器的新名称。

示例

更改一个外部数据包装器dbi，增加选项 foo并删除bar：

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

把外部数据包装器dbi的验证器改为 bob.myvalidator：

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

兼容性

ALTER FOREIGN DATA WRAPPER符合 ISO/IEC 9075-9 (SQL/MED)。不过HANDLER、VALIDATOR、OWNER TO 以及RENAME子句是扩展。

另见

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — 更改一个外部表的定义

大纲

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

其中 action 是以下之一：

```
    ADD [ COLUMN ] column_name data_type [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation
 ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value']
 [, ... ])
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITH OIDS
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
```

描述

ALTER FOREIGN TABLE更改一个现有外部表的定义。 有几种子形式：

ADD COLUMN

这种形式使用与CREATE FOREIGN TABLE相同的语法把 一个新的列增加到该外部表。和为常规表增加一列不同，这种形式并不影响底层 的存储：这个动作只是简单地声明通过该外部表可以访问某个新的列而已。

DROP COLUMN [IF EXISTS]

这种形式从一个外部表删掉一列。如果在该表外部有任何东西依赖于该列， 你将需要写上CASCADE，典型的例子就是视图。如果指定了 IF EXISTS并且该列不存在，将不会抛出错误。在这种 情况下会转而发出一个提示。

SET DATA TYPE

这种形式更改一个外部表的一列的类型。同样，这种形式并不影响底层 的存储：这个动作只是简单地更改PostgreSQL 相信该列所具有的类型。

SET/DROP DEFAULT

这些形式设置或者移除一列的默认值。默认值只会应用于后续的 INSERT或UPDATE命令，它们 不会导致已经在表中的行被更改。

SET/DROP NOT NULL

把一列标记为允许或者不允许空值。

SET STATISTICS

这种形式为后续的ANALYZE操作设置针对每列 的统计收集目标。详见ALTER TABLE的类似形式。

SET (attribute_option = value [, ...])
RESET (attribute_option [, ...])

这种形式设置或重置针对每个属性的选项。详见ALTER TABLE 的类似形式。

SET STORAGE

这种形式设置一个列的存储模式。详见 ALTER TABLE中类似的模式。注意存储模式 不会产生效果，除非该表的外部数据包装器选择处理它。

ADD table_constraint [NOT VALID]

这种形式为外部表增加一个新的约束，使用的语法和 CREATE FOREIGN TABLE中相同。当前只 支持CHECK约束。

和向常规表增加约束的情况不同，为外部表增加约束时不会做任何事情来 验证该约束是否正确。这个动作只是简单地声明了该外部表中所有的行都 应该满足的某种新的条件（见 CREATE FOREIGN TABLE中的讨论）。如果该 约束被标记为NOT VALID，那么它不被假设为有效，而只是 被记录下来以备未来使用。

VALIDATE CONSTRAINT

这种形式把一个之前被标记为NOT VALID的约束标 记为有效。不会做任何动作来验证该约束，但是未来的查询将会假定该 约束是保持的。

DROP CONSTRAINT [IF EXISTS]

这种形式删掉在一个外部表上的指定约束。如果指定了 IF EXISTS但约束并不存在，则不会抛出错误。 在这种情况下会发出一个提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于该外部表的触发器的触发情况。详见 ALTER TABLE的类似形式。

SET WITH OIDS

这种形式为表增加一个oid系统列（见 第 5.4 节。如果该表已经有 OID，则 这种形式不会做任何事情。除非该表的外部数据包装器支持 OID，这个列 将被简单地读作零。

注意这和ADD COLUMN oid oid并不等效，后者将增加 一个刚好名字为oid的普通列而不是一个系统列。

SET WITHOUT OIDS

这种形式从表移除oid系统列。这正好 等效于DROP COLUMN oid RESTRICT， 不过如果表上已经没有oid时它不会做出 提示或者报错。

INHERIT parent_table

这种形式把目标外部表作为指定的父表的新后代。详见 ALTER TABLE的类似的形式。

NO INHERIT parent_table

这种形式把目标外部表从指定的父表的子女列表中移除。

OWNER

这种形式把该外部表的拥有者改成指定的用户。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

更改该外部表或者其中一个列的选项。ADD、SET 以及DROP指定要执行的动作。如果没有显式地指定操作，将假定 为ADD。不允许重复的名称（不过一个表选项和一个列选项可以重 名）。选项名称和值也会用外部数据包装器库来验证。

RENAME

RENAME形式更改外部表的名称或者外 部表中一个列的名称。

SET SCHEMA

这种形式把外部表移动到另一个模式中。

所有除了RENAME和SET SCHEMA的 动作都能被整合到一个多修改列表以便能被并行应用。例如，可以在一个 命令中增加几个列并且/或者修改几个列的类型。

如果该命令被写作ALTER FOREIGN TABLE IF EXISTS ...并且 该外部表不存在，则不会抛出错误。这种情况下会发出一个提示。

你必须拥有该表以使用ALTER FOREIGN TABLE。要更改一个 外部表的模式，你必须还拥有新模式上的CREATE特权。要 更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须 具有在该表的模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建该表做不到的事情。不过，一个超级用户怎么都能 更改任何表的所有权）。要增加一列或者修改一个列的类型，你还必须具有该数 据类型上的USAGE特权。

参数

name

一个要修改的现有外部表的名称（可以被模式限定）。如果在表名前指定了 ONLY，则只有该表被修改。如果没有指定ONLY， 该表和它所有的后代表（如果有）都会被修改。可选地，在表名后面指定 *可以显式地表示将后代表包括在内。

column_name

一个新的或者现有列的名称。

new_column_name

一个现有列的新名称。

new_name

该表的新名称。

data_type

新列的数据类型或者一个现有列的新数据类型。

table_constraint

New table constraint for the foreign table.

constraint_name

Name of an existing constraint to drop.

CASCADE

自动删除依赖于被删除列或约束的对象（例如，引用该列的视图）， 并且接着删除依赖于那些对象的 所有对象（见第 5.13 节。

RESTRICT

如果有任何依赖对象就拒绝删除该列或约束。这是默认行为。

trigger_name

要禁用或启用的一个触发器的名称。

ALL

禁用或者启用所有属于该外部表的触发器（如果任何触发器是内部生成 的触发器，这都要求超级用户特权。核心系统不会向外部表增加这类触发 器，但是附加代码会这样做。）。

USER

禁用或者启用属于该外部表的除了内部生成的触发器之外的所有触发器。

parent_table

要与这个外部表关联或者解除关联的父表。

new_owner

该表的新拥有者的用户名。

new_schema

该表要被移动到其中的模式的名称。

注解

关键词COLUMN是噪声词并且可以被忽略。

当使用ADD COLUMN或 DROP COLUMN增加或移除一列、增加一个NOT NULL 或者CHECK 约束或者用SET DATA TYPE更改一个列类型时，不会检查与外部服务器的一 致性。确保该表定义匹配远端是用户的责任。

关于有效参数的进一步描述可参考CREATE FOREIGN TABLE。

示例

要把一列标记为非空：

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

要更改一个外部表的选项：

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2
 'value2', DROP opt3 'value3');
```

兼容性

形式ADD、DROP以及 SET DATA TYPE符合 SQL 标准。其他形式是 SQL 标准的 PostgreSQL扩展。在一个 ALTER FOREIGN TABLE命令中指定多于一个操作也是一种扩展。

ALTER FOREIGN TABLE DROP COLUMN可以被用来删除 一个外部表的唯一一列，从而留下一个没有列的表。这是一种 SQL 的扩展，它 允许没有列的外部表。

另见

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — 更改一个函数的定义

大纲

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中 action 是以下之一：

```
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

ALTER FUNCTION更改一个函数的定义。

你必须拥有该函数以使用ALTER FUNCTION。要更改一个函数 的模式，你还必须具有新模式上的CREATE特权。要 更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须 具有在该函数的模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建该函数做不到的事情。不过，一个超级用户怎么都能 更改任何函数的所有权）。

参数

name

一个现有函数的名称（可以被模式限定）。如果没有指定参数列表， 则该名称必须在它的模式中唯一。

argmode

一个参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认为 IN。注意ALTER FUNCTION 并不真正关心OUT参数，因为在决定函数的身份时只需要输 入参数。因此列出IN、INOUT以及 VARIADIC参数即可。

argname

一个参数的名称。注意ALTER FUNCTION 并不真正参数名称，因为在确定函数的身份时只需要参数的数据类型即可。

argtype

该函数的参数（如果有）的数据类型（可以被模式限定）。

new_name

该函数的新名称。

new_owner

该函数的新拥有者。注意如果该函数被标记为 SECURITY DEFINER，它的后续执行将会使用新拥有者。

new_schema

该函数的新模式。

extension_name

该函数所以来的扩展名。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT将该函数改为在某些 或者全部参数为空值时可以被调用。RETURNS NULL ON NULL INPUT或者 STRICT将该函数改为只要任一参数为空值就不被调用而 是自动假定一个空值结果。详见CREATE FUNCTION。

IMMUTABLE
STABLE
VOLATILE

把该函数的稳定性更改为指定的设置。详见 CREATE FUNCTION。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

更改该函数是否为一个安全性定义者。关键词EXTERNAL 是为了符合 SQL，它会被忽略。关于这项能力的详情请见 CREATE FUNCTION。

PARALLEL

决定该函数对于并行是否安全。详见 CREATE FUNCTION。

LEAKPROOF

更改该函数是否被认为是防泄漏的。关于这项能力的详情请见 CREATE FUNCTION。

COST execution_cost

更改该函数的估计执行代价。详见CREATE FUNCTION。

ROWS result_rows

更改一个集合返回函数的估计行数。详见 CREATE FUNCTION。

configuration_parameter
value

当该函数被调用时，要对一个配置参数做出增加或者更改的赋值。如果 value是DEFAULT或者使用等价的RESET，该函数本地的设置将会被 移除，这样该函数会使用其环境中存在的值执行。使用RESET ALL可以清除所有函数本地的设置。 SET FROM CURRENT把ALTER FUNCTION 执行时该参数的当前值保存为进入 该函数时要应用的值。

有关允许的参数名称和值可详见SET以及 第 19 章

RESTRICT

为了符合 SQL 标准存在，被忽略。

示例

要把用于类型integer的函数sqrt 重命名为square_root：

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

要把用于类型integer的函数sqrt 的拥有者改为joe：

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

要把用于类型integer的函数sqrt 的模式改为maths：

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

要把类型integer的函数sqrt 标记为依赖于扩展mathlib：

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

要调整一个函数的自动搜索路径：

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

要禁止一个函数的search_path的自动设置：

```
ALTER FUNCTION check_password(text) RESET search_path;
```

这个函数将用其调用者使用的搜索路径来执行。

兼容性

这个语句部分兼容 SQL 标准中的ALTER FUNCTION语句。该标准允许修改一个函数的更多属性，但是不提供 重命名一个函数、标记一个函数为安全性定义者、为一个函数附加配置参 数值或者更改一个函数的拥有者、模式或者稳定性等功能。该标准还要求 RESTRICT关键字，它在PostgreSQL 中是可选的。

另见

CREATE FUNCTION, DROP FUNCTION, ALTER PROCEDURE, ALTER ROUTINE

ALTER GROUP

ALTER GROUP — 更改角色名称或者成员关系

大纲

```
ALTER GROUP role_specification ADD USER user_name [, ... ]
ALTER GROUP role_specification DROP USER user_name [, ... ]
```

其中 role_specification 可以是：

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

```
ALTER GROUP group_name RENAME TO new_name
```

描述

ALTER GROUP更改一个用户组的属性。 这是一个被废弃的命令，不过为了向后兼容还是会被接受，因为组（以及用户） 已经被更一般的角色概念替代了。

前两个变体向一个组增加用户或者从一个组中移除用户（为了这个目的， 任何角色都可以扮演“用户”或者“组”）。这些变体 实际上等效于在被称为“组”的角色中授予或者回收成员关系， 因此最好的方法是使用GRANT或者 REVOKE。

第三种变体会更改该组的名称。这恰好等效于用ALTER ROLE 重命名该角色。

参数

group_name

要修改的组（角色）的名称。

user_name

要被加入到该组或者从该组移除的用户（角色）。这些用户必须已经存在， ALTER GROUP不会创建或者删除用户。

new_name

该组的新名称。

示例

向一个组增加用户：

```
ALTER GROUP staff ADD USER karl, john;
```

从一个组移除一个用户：

```
ALTER GROUP workers DROP USER beth;
```

兼容性

在 SQL 标准中没有ALTER GROUP语句。

另见

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — 更改一个索引的定义

大纲

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX name ATTACH PARTITION index_name
ALTER INDEX name DEPENDS ON EXTENSION extension_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX [ IF EXISTS ] name ALTER [ COLUMN ] column_number
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

描述

ALTER INDEX更改一个现有索引的定义。 它有几种形式：

RENAME

RENAME形式更改该索引的名称。如果索引与一个表约束（UNIQUE、PRIMARY KEY或者EXCLUDE）关联，该约束也会被重命名。这对已存储的数据没有影响。

SET TABLESPACE

这种形式更改该索引的表空间为指定的表空间，并且把与该索引相关联的数据文件 移动到新的表空间中。要更改一个索引的表空间，你必须拥有该索引并且具有新表 空间上的CREATE特权。可以使用 ALL IN TABLESPACE形式把当前数据库中在一个表空间内的所有索引全部移动到另一个表空间中，这将会锁定所有要被移动的索引然后挨个移 动它们。这种形式也支持OWNED BY，即只移动属于指定角色 的索引。如果指定了NOWAIT选项，那么当该命令无法立刻获 得所有锁时将会失败。注意这个命令不会移动系统目录，如果想要移动系统目录， 应使用ALTER DATABASE或者显式的 ALTER INDEX调用。另见CREATE TABLESPACE。

ATTACH PARTITION

导致提到的索引变成附着于被修改的索引。提及的索引必须在包含被修改索引的表的一个分区上，并且具有一种等效的定义。一个附着索引不能被单独删除，它会在其父索引被删除时自动连带删除。

DEPENDS ON EXTENSION

这种形式把该索引标记为依赖于扩展，这样如果该扩展被删除，该索引也将被 自动删除。

SET (storage_parameter = value [, ...])

这种形式为该索引更改一个或者多个索引方法相关的存储参数。可用的参数详见 CREATE INDEX。注意这个命令不会立刻修改索引内容， 根据参数你可能需要用REINDEX重建索引来得到想要的 效果。

RESET (storage_parameter [, ...])

这种形式把一个或者多个索引方法相关的存储参数重置为其默认值。正如 SET一样，可能需要一次REINDEX来完全更新 该索引。

ALTER [COLUMN] column_number SET STATISTICS integer

这种形式为后续的ANALYZE操作设置针对每个列的统计信息收集目标，不过只能用在被定义为表达式的索引列上。由于表达式缺少唯一的名称，我们通过该索引列的顺序号来引用它们。收集目标可以被设置为范围0到10000之间的值。另外，把它设置为-1会恢复到使用系统的默认统计信息目标（default_statistics_target）。更多有关PostgreSQL查询规划器使用统计信息的内容，请参考第 14.2 节

参数

IF EXISTS

如果该索引不存在不要抛出错误。这种情况下将发出一个提示。

column_number

引用该索引列的顺序（从左往右）位置的顺序号。

name

要更改的一个现有索引的名称（可能被模式限定）。

new_name

该索引的新名称。

tablespace_name

该索引将被移动到的表空间。

extension_name

该索引所依赖的扩展的名称。

storage_parameter

一个索引方法相关的存储参数的名称。

value

一个索引方法相关的存储参数的新值。根据该参数，这可能是一个数字或者一个 词。

注解

也可以用ALTER TABLE来做这些操作。实际上， ALTER INDEX只是ALTER TABLE应用在索引 上的形式的别名而已。

以前有一种ALTER INDEX OWNER变体，但现在已被忽略（会出现 一个警告）。一个索引的拥有者不能与其基表的拥有者不同。更改基表的拥有者 会自动地更改索引的拥有者。

不允许更改系统目录索引的任何部分。

示例

要重命名一个现有索引：

```
ALTER INDEX distributors RENAME TO suppliers;
```

把一个索引移动到一个不同的表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

更改一个索引的填充因子（假设该索引方法支持填充因子）：

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

为一个表达式索引设置统计信息收集目标：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

兼容性

ALTER INDEX是一种 PostgreSQL扩展。

另见

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — 更改一种过程语言的定义

大纲

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
```

描述

ALTER LANGUAGE更改一种过程语言的定义。唯一 的功能是重命名该语言或者为它赋予一个新的拥有者。要使用 ALTER LANGUAGE，你必须是一个超级用户或者该 语言的拥有者。

参数

name

语言的名称

new_name

该语言的新名称

new_owner

该语言的新拥有者

兼容性

在 SQL 标准中没有ALTER LANGUAGE语句。

另见

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — 更改一个大对象的定义

大纲

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
```

描述

ALTER LARGE OBJECT更改一个大对象的定义。

您必须拥有大对象才能使用ALTER LARGE OBJECT。要更改所有者，您还必须是新所有者的直接或间接成员。（不过，超级用户仍然可以更改任何大对象。）当前，唯一的功能是分配新所有者，因此两者的约束都始终适用。

参数

large_object_oid

要被修改的大对象的 OID

new_owner

该大对象的新拥有者

兼容性

在 SQL 标准中没有ALTER LARGE OBJECT 语句。

另见

第 35 章

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — 更改一个物化视图的定义

大纲

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW name
    DEPENDS ON EXTENSION extension_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

其中 action是下列之一：

```
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

ALTER MATERIALIZED VIEW更改一个现有物化视图的 多个辅助属性。

要使用ALTER MATERIALIZED VIEW，你必须拥有该物化视图。要 更改一个物化视图的模式，你还必须拥有新模式上的CREATE特权。要更　改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须拥有该　物化视图所在模式上的CREATE特权（这些限制强制修改拥有者不　能做一些通过删除和重建该物化视图做不到的事情。不过，一个超级用户怎么都能更改 任何视图的所有权。）。

DEPENDS ON EXTENSION形式把该物化视图标记为依赖于一个 扩展，这样该扩展被删除时会自动地删除掉这个物化视图。

可用于ALTER MATERIALIZED VIEW的语句形式和动作是 ALTER TABLE的一个子集，并且在用于物化视图时具有相 同的含义。详见ALTER TABLE的描述。

参数

name

一个现有物化视图的名称（可以是模式限定的）。

column_name

一个新的或者现有的列的名称。

extension_name

该物化视图所依赖的扩展的名称。

new_column_name

一个现有列的新名称。

new_owner

该物化视图的新拥有者的用户名。

new_name

该物化视图的新名称。

new_schema

该物化视图的新模式。

示例

把物化视图foo重命名为 bar：

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

兼容性

ALTER MATERIALIZED VIEW是一种 PostgreSQL扩展。

另见

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — 更改一个操作符的定义

大纲

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET SCHEMA new_schema

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET ( {  RESTRICT = { res_proc | NONE }
           | JOIN = { join_proc | NONE }
         } [, ... ] )
```

描述

ALTER OPERATOR更改一个操作符的定义。

要使用ALTER OPERATOR，你必须拥有该操作符。要更改拥有者， 你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该操作符所在　模式上的CREATE特权（这些限制强制修改拥有者不能做一些通过　删除和重建操作符做不到的事情。不过，一个超级用户怎么都能更改任何操作符的所 有权。）。

参数

name

　一个现有操作符的名称（可以是模式限定的）。

left_type

　该操作符左操作数的数据类型，如果该操作符没有左操作数可以写成 NONE。

right_type

　该操作符右操作数的数据类型，如果该操作符没有右操作数可以写成 NONE。

new_owner

　该操作符的新拥有者。

new_schema

　该操作符的新模式。

res_proc

　这个操作符的约束选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。

join_proc

　这个操作符的连接选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。

示例

更改类型text的一个自定义操作符a @@ b 的拥有者：

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

更改类型int[]的自定义操作符a && b的 约束和连接选择度估算器函数：

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN =
 _int_contjoinsel);
```

兼容性

在 SQL 标准中没有ALTER OPERATOR语句。

另见

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — 更改一个操作符类的定义

大纲

```
ALTER OPERATOR CLASS name USING index_method
    RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS name USING index_method
    SET SCHEMA new_schema
```

描述

ALTER OPERATOR CLASS更改一个操作符类的定义。

要使用ALTER OPERATOR CLASS，你必须拥有该操作符类。要 修改拥有者，你还必须是新拥有角色的一个直接或间接成员，并且该角色必须具有 该操作符类所在模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建操作符类做不到的事情。不过，一个超级用户怎么都 能更改任何操作符类的所有权。）。

参数

name

一个现有操作符类的名称（可以是模式限定的）。

index_method

这个操作符类所服务的索引方法的名称。

new_name

该操作符类的新名称。

new_owner

该操作符类的新拥有者。

new_schema

该操作符类的新模式。

兼容性

在 SQL 标准中没有ALTER OPERATOR CLASS语句。

另见

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — 更改一个操作符族的定义

大纲

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type )
             [ FOR SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
             function_name [ ( argument_type [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type [ , op_type ] )
   | FUNCTION support_number ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method
    RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY name USING index_method
    SET SCHEMA new_schema
```

描述

ALTER OPERATOR FAMILY更改一个操作符族 的定义。你能增加操作符以及支持函数到该家族、从该族中移除它们或者更改 该族的名称或者拥有者。

在用ALTER OPERATOR FAMILY增加操作符和 支持函数到一个族中时，它们不是族内任何特定操作符类的组成部分，而只是 “松散”地存在于该族中。这表示这些操作符和函数与该族的语义兼 容，但是没有被任何特定索引的正确功能所要求（所要求的操作符和函数应该 被作为一个操作符类的一部分声明，见CREATE OPERATOR CLASS）。 PostgreSQL将允许一个族的松散成员在 任何时候被从该族中删除，但是在删除一个操作符类的成员之前，必须已经删 除整个类以及依赖于该成员的索引。具有代表性的是，单一数据类型操作符和 函数是操作符类的一部分，因为在特定数据类型上的索引需要它们的支持。而 多数据类型操作符和函数则被作为该族的松散成员。

要使用ALTER OPERATOR FAMILY，你必须是超级用户（ 做这样的限制是因为一个错误的操作符族定义可能会迷惑服务器甚至让它崩溃）。

ALTER OPERATOR FAMILY目前不检测操作符族 定义是否包括该索引方法所要求的所有操作符和函数，也不检查操作符和函数是 否形成了一个有理的集合。定义一个合法的操作符族是用户的责任。

进一步的信息请参考第 38.15 节

参数

name

一个现有操作符族的名称（可以是模式限定的）。

index_method

这个操作符族所应用的索引方法的名称。

strategy_number

与该操作符族相关的一个操作符的索引方法策略号。

operator_name

与该操作符族相关的一个操作符的名称（可以是模式限定的）。

op_type

在一个OPERATOR子句中指定该操作符的操作数数据类型， 或者用NONE来表示一个左一元或者右一元操作符。不同于 CREATE OPERATOR CLASS中类似的语法，操作数数据 类型总是必须被指定。

在一个ADD FUNCTION子句中指定该函数意图支持的操作数 数据类型（如果不同于该函数的输入数据类型）。对于 B-树比较函数和哈希 函数，有必要指定op_type，因为该函数的输入数据类型 总是正确的。对于 B-树排序支持函数和 GiST、SP-GiST 和 GIN 操作符类中 的所有函数，有必要指定该函数要使用的操作数数据类型。

在一个DROP FUNCTION子句中，必须指定该函数要支持的操 作数数据类型。

sort_family_name

一个现有btree操作符族的名称（可能是模式限定的）， 它描述与一个排序操作符相关的排序顺序。

如果既没有指定FOR SEARCH也没有指定FOR ORDER BY， 默认值是FOR SEARCH。

support_number

一个与该操作符族相关的函数的索引方法支持过程编号。

function_name

作为该操作符族的一种索引方法支持函数的函数名称（可以是模式限定的）。 如果没有指定参数列表，则该名称必须在其模式中唯一。

argument_type

该函数的参数数据类型。

new_name

该操作符族的新名称。

new_owner

该操作符族的新拥有者。

new_schema

该操作符族的新模式。

OPERATOR和FUNCTION子句可以以任何顺序出现。

注解

注意DROP语法只通过策略或者支持号以及输入数据类型指定该 操作符族中的“slot”。占用这个槽的操作符或函数的名称不会被提及。 还有，对于DROP FUNCTION，要指定的类型是

该函数意图支持 的输入数据类型。对于 GiST、SP-GiST 以及 GIN 索引，可能无需对该函数的 实际输入参数类型做任何事情。

因为索引机制在使用函数之前不会检查其上的访问权限，包括一个操作符族中的 函数或操作符都等同于授予了其上的公共执行权限。这对于操作符族中很有用的 这类函数来说，这通常不成问题。

操作符应该由 SQL 函数定义。一个 SQL 函数很可能被内联到调用查询中，这将 阻止优化器识别出该查询匹配一个索引。

在PostgreSQL 8.4 之前， OPERATOR子句可以包括一个RECHECK选项。这不再 被支持，因为一个索引操作符是否为“lossy”现在会在运行时即时决定。 这允许高效地处理一个操作符可能或者不可能为有损的情况。

示例

下列示例命令为一个操作符族增加跨数据类型的操作符和支持函数，该操 作符族已经包含用于数据类型int4以及int2的 B-树 操作符类。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

再次移除这些项：

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

兼容性

在 SQL 标准中没有 ALTER OPERATOR FAMILY语句。

另见

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — 更改一条行级安全性策略的定义

大纲

```
ALTER POLICY name ON table_name RENAME TO new_name

ALTER POLICY name ON table_name
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

描述

ALTER POLICY更改一条现有行级安全性策略的定义。 请注意，ALTER POLICY只允许修改策略所应用的角色集合， 和要修改的USING和WITH CHECK表达式。 要更改策略的其他属性，例如其应用的命令，或者是允许还是限制， 则必须删除并重新创建策略。

要使用ALTER POLICY，你必须拥有该策略所适用的 表。

在ALTER POLICY的第二种形式中，如果指定了角色列表、 using_expression以及check_expression， 它们会被独立地替换。当这些子句之一被省略时，策略的对应部分不会被更改。

参数

name

要更改的现有策略的名称。

table_name

该策略所在的表的名称（可以被模式限定）。

new_name

该策略的新名称。

role_name

该策略适用的角色。可以一次指定多个角色。要把该策略 应用于所有角色，可使用PUBLIC。

using_expression

该策略的USING表达式。详见 CREATE POLICY。

check_expression

该策略的WITH CHECK表达式。详见 CREATE POLICY。

兼容性

ALTER POLICY是一种PostgreSQL扩展。

另见

CREATE POLICY, DROP POLICY

ALTER PROCEDURE

ALTER PROCEDURE — change the definition of a procedure

大纲

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中action是下列之一：

```
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

简介

ALTER PROCEDURE更改一个过程的定义。

要使用ALTER PROCEDURE，你必须拥有该过程。要更改一个过程的方案，你还必须有新方案上的CREATE特权。要更改拥有者，你还必须是新拥有角色的直接或间接成员，并且那个角色在该过程的方案上拥有CREATE特权（这些限制强制更新拥有者无法做到通过删除和重建该过程无法做到的事情。不过，超级用户总是能够更改任何过程的拥有关系）。

参数

name

一个现有的过程的名字（可以被方案限定）。如果没有指定参数列表，这个名字必须在其方案中唯一。

argmode

参数的模式：IN或VARIADIC。如果被省略，默认是IN。

argname

参数的名字。注意ALTER PROCEDURE实际上并不关心参数名，因为只需要参数的数据类型来确定过程的身份。

argtype

如果该过程有参数，这是参数的数据类型（可以被方案限定）。

new_name

该过程的新名字。

new_owner

该过程的新拥有者。注意，如果这个过程被标记为SECURITY DEFINER，接下来它将被作为新拥有者执行。

new_schema

该过程的新方案。

extension_name

该过程所依赖的扩展的名称。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

更改该过程是否为一个安全性定义器。关键词EXTERNAL由于SQL符合性的原因被忽略。更多有关这个能力的信息请见CREATE PROCEDURE。

configuration_parameter
value

增加或者更改在调用该过程时，要对一个配置参数做的赋值。如果value是DEFAULT或者等效的值，则会使用RESET，过程本地的设置会被移除，这样该过程的执行就会使用其所处环境中的值。使用RESET ALL可以清除所有的过程本地设置。SET FROM CURRENT会把ALTER PROCEDURE执行时该参数的当前值保存为进入该过程时要被应用的值。

关于允许的参数名和参数值的更多信息请见SET和第 19 章

RESTRICT

为了符合SQL标准会被忽略。

示例

要重命名具有两个integer类型参数的过程insert_data为insert_record：

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

要把具有两个integer类型参数的过程insert_data的拥有者改为joe：

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

要重把具有两个integer类型参数的过程insert_data的方案改为accounting：

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

把过程insert_data(integer, integer)标记为依赖于扩展myext：

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;
```

要调整一个过程自动设置的搜索路径：

```
ALTER PROCEDURE check_password(text) SET search_path = admin, pg_temp;
```

要为一个过程禁用search_path的自动设置：

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

现在这个过程将用其调用者所使用的任何搜索路径执行。

兼容性

这个语句与SQL标准中的ALTER PROCEDURE语句部分兼容。标注你允许修改一个过程的更多性质，但是不提供重命名过程、让过程成为安全性定义器、为过程附加配置参数值或者更改过程的拥有者、方案或者可变性的能力。标准还要求RESTRICT关键字，而它在PostgreSQL中是可选的。

另见

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

ALTER PUBLICATION

ALTER PUBLICATION — 修改发布的定义

大纲

```
ALTER PUBLICATION name ADD TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name DROP TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
ALTER PUBLICATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION name RENAME TO new_name
```

描述

命令ALTER PUBLICATION可以更改发布的属性。

前三个语句更改哪些表是该发布的一部分。SET TABLE 子句用指定的表替换发布中的表的列表。ADD TABLE和 DROP TABLE子句将从发布中添加和删除一个或多个表。 请注意，将表添加到已订阅的发布中将需要在订阅端执行ALTER SUBSCRIPTION ... REFRESH PUBLICATION操作才能生效。

第四条语句可以改变在CREATE PUBLICATION中指定的所有发布属性。 该命令中未提及的属性保留其先前的设置。

其余语句更改所有者和发布的名称。

你必须拥有该发布才能使用ALTER PUBLICATION。要改变所有者， 你也必须是新所有者角色的直接或间接成员。新的所有者必须在数据库上拥有 CREATE权限。此外，FOR ALL TABLES发布的新所有者必须是超级用户。但是， 超级用户可以在避开这些限制的情况下更改发布的所有权。

参数

name

要修改定义的现有发布的名称。

table_name

现有表的名称。如果在表名之前指定了ONLY，则只有该表受到影响。 如果没有指定ONLY，则该表及其所有后代表（如果有的话）都会受到影响。 可选地，可以在表名之后指定*以明确指示包含后代表。

SET (publication_parameter [= value] [, ...])

该子句修改最初由CREATE PUBLICATION设置的发布参数。

new_owner

发布的新所有者的用户名。

new_name

发布的新名称。

示例

将发布修改为只发布删除和更新： Change the publication to publish only deletes and updates:

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

给发布添加一些表：

```
ALTER PUBLICATION mypublication ADD TABLE users, departments;
```

兼容性

ALTER PUBLICATION是PostgreSQL的一个扩展。

又见

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — 更改一个数据库角色

大纲

```
ALTER ROLE role_specification [ WITH ] option [ ... ]
```

其中option可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中role_specification可以是：

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

描述

ALTER ROLE更改一个 PostgreSQL角色的属性。

前面列出的这个命令的第一种变体能够更改CREATE ROLE中 指定的很多角色属性（覆盖了所有可能的属性，不过没有增加和移除成员关系的选项， 如果要增加和移除成员关系可使用GRANT和 REVOKE）。该命令中没有提到的属性保持它们之前的设置。 数据库超级用户能够更改任何角色的任何这些设置。具有CREATEROLE 特权的角色能够更改任何这些设置，但是只能为非超级用户和非复制角色修改。普通 角色只能更改它们自己的口令。

第二种变体更改该角色的名称。数据库超级用户能重命名任何角色。具有 CREATEROLE特权的角色能够重命名任何非超级用户角色。当前的会话 用户不能被重命名（如果需要这样做，请以一个不同的用户连接）。由于 MD5加密的口令使用角色名作为 salt，因此如果一个角色的口令是 MD5加密的，重命名该角色会清空其口令。

其余的变体用于更改一个角色的配置变量的会话默认值，可以为所有数据库设置，或者 只为IN DATABASE中指定的数据库设置。如果指定的是 ALL而不是一个角色名，将会为所有角

色更改该设置。把 ALL和IN DATABASE一起使用实际上和使用命 令ALTER DATABASE ... SET ...相同。

只要改角色后续开始一个新会话，指定的值将会成为该会话的默认值，并且会覆盖 postgresql.conf中存在的值或者从 postgres命令行收到的值。这只在登录时发生，执行 SET ROLE或者 SET SESSION AUTHORIZATION不会导致新的配置值被设置。 对于所有数据库设置的值会被附加到一个角色的数据库相关的设置所覆盖。特定数 据库或角色的设置会覆盖为所有角色所作的设置。

超级用户能够更改任何人的会话默认值。具有CREATEROLE特权的角色 能够更改非超级用户的默认值。普通角色只能为它们自己设置默认值。某些配置变量 不能以这种方式设置，或者只能由一个超级用户发出的命令设置。只有超级用户能够 更改所有角色在所有数据库中的设置。

参数

name

要对其属性进行修改的角色的名称。

CURRENT_USER

修改当前用户而不是一个显式标识的角色。

SESSION_USER

修改当前会话用户而不是一个显式标识的角色。

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT connlimit
[ENCRYPTED] PASSWORD 'password'
PASSWORD NULL
VALID UNTIL 'timestamp'

这些子句修改原来有CREATE ROLE 设置的属性。更多信息请见 CREATE ROLE参考页。

new_name

该角色的新名称。

database_name

要在其中设置该配置变量的数据库名称。

configuration_parameter
value

把这个角色的指定配置参数的会话默认值设置为给定值。如果 value为DEFAULT 或者等效地使用了RESET，角色相关的变量 设置会被移除，这样该角色将会在新会话中继承系

统范围的默认 设置。使用RESET ALL可清除所有角色相关的 设置。SET FROM CURRENT可以把会话中该参数的 当前值保存为角色相关的值。如果指定了 IN DATABASE，只会为给定的角色和数据库 设置或者移除该配置参数。

角色相关的变量设置只在登录时生效， SET ROLE以及 SET SESSION AUTHORIZATION不会处理角色 相关的变量设置。

关于允许的参数名称和值详见SET和 第 19 章

注解

使用CREATE ROLE增加新角色，使用 DROP ROLE移除一个角色。

ALTER ROLE无法更改一个角色成员关系。 可以使用GRANT和 REVOKE来实现。

在使用这个命令指定一个未加密口令时要多加小心。该口令将会以明文 传送到服务器，并且它还可能会被记录在客户端的命令历史或者服务器 日志中。psql包含了一个命令\password，它可以被用来更改一个角色 的口令而不暴露明文口令。

也可以把一个会话默认值绑定到一个指定的数据库而不是一个角色，详见 ALTER DATABASE。如果出现冲突，数据库角色相关 的设置会覆盖角色相关的设置，角色相关的又会覆盖数据库相关的设置。

示例

更改一个角色的口令：

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

移除一个角色的口令：

```
ALTER ROLE davide WITH PASSWORD NULL;
```

更改一个口令的失效日期，指定该口令应该在 2015 年 5 月 4 日中午 （在一个比UTC快 1 小时的时区）过期：

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

让一个口令永远有效：

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

让一个角色能够创建其他角色和新的数据库：

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

为一个角色指定 maintenance_work_mem参数的非默认设置：

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

为一个角色指定 client_min_messages参数的数据库相关的非 默认设置：

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

兼容性

ALTER ROLE语句是一个 PostgreSQL扩展。

另见

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER ROUTINE

ALTER ROUTINE — 更改一个例程的定义

大纲

```
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中action是下列之一：

```
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

ALTER ROUTINE更改一个例程的定义，它可以是聚集函数、普通函数或者过程。参数的描述、更多的例子以及进一步的细节请参考ALTER AGGREGATE、ALTER FUNCTION以及ALTER PROCEDURE。

示例

将类型integer的例程foo重命名为foobar：

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

不管foo是聚集、函数还是过程，这个命令都能使用。

兼容性

这个语句与SQL标准中的ALTER ROUTINE语句部分兼容。更多细节请参考ALTER FUNCTION和ALTER PROCEDURE。允许例程名称引用聚集函数是一种PostgreSQL的扩展。

另见

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

注意没有CREATE ROUTINE命令。

ALTER RULE

ALTER RULE — 更改一个规则定义

大纲

```
ALTER RULE name ON table_name RENAME TO new_name
```

描述

ALTER RULE更改一条现有规则的定义。当前，唯一可用的 动作是更改规则的名称。

要使用ALTER RULE，你必须拥有该规则适用的表或者视图。

参数

name

要修改的一条现有规则的名称。

table_name

该规则适用的表或视图的名称（可以是模式限定的）。

new_name

该规则的新名称。

示例

要重命名一条现有的规则：

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

兼容性

ALTER RULE是一种 PostgreSQL的语言扩展，整个查询重写系统也是。

另见

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — 更改一个模式的定义

大纲

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

ALTER SCHEMA更改一个模式的定义。

要使用ALTER SCHEMA，你必须拥有该模式。要重命名一个模式， 你还必须拥有该数据库的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该数据库上的 CREATE特权（注意超级用户自动拥有所有这些特权）。

参数

name

一个现有模式的名称。

new_name

该模式的新名称。新名称不能以pg_开始，因为这些名称被 保留用于系统模式。

new_owner

该模式的新拥有者。

兼容性

在 SQL 标准中没有ALTER SCHEMA语句。

另见

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — 更改一个序列发生器的定义

大纲

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

描述

ALTER SEQUENCE更改一个现有序列发生器的参数。 任何没有在ALTER SEQUENCE命令中明确设置的参数 保持它们之前的设置。

要使用ALTER SEQUENCE，你必须拥有该序列。要更改一个序列 的模式，你还必须拥有新模式上的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建该序列做不到的事情。不过，一个超级用户怎么都能更改任何序列的所有权。）。

参数

name

要修改的序列的名称（可以是模式限定的）。

IF EXISTS

在序列不存在时不要抛出一个错误。这种情况下会发出一个提示。

data_type

可选子句AS data_type 改变序列的数据类型。有效类型是smallint、integer和bigint。

当且仅当先前的最小值和最大值是旧数据类型的最小值或最大值时（换句话说， 如果序列是使用NO MINVALUE或NO MAXVALUE， 隐式或显式创建的），则更改数据类型会自动更改序列的最小值和最大值。 否则，将保留最小值和最大值，除非将新值作为同一命令的一部分给出。 如果最小值和最大值不符合新的数据类型，则会生成错误。

increment

子句INCREMENT BY increment是可选的。 一个正值将产生一个上升序列，一个负值会产生一个下降序列。如果 没有指定，旧的增量值将被保持。

minvalue
NO MINVALUE

可选的子句MINVALUE minvalue决定一个序列 能产生的最小值。如果指定了NO MINVALUE，上升序列和下降序列的默认值分别是 1 和 数据类型的最小值。如果这些选项都没有被指定，将保持当前的 最小值。

maxvalue
NO MAXVALUE

可选的子句MAXVALUE maxvalue决定一个序列 能产生的最大值。如果指定了NO MAXVALUE，上升序列和下降序列的默认值分别是 数据类型的最大值和 -1。如果这些选项都没有被指定，将保持当前的 最大值。

start

可选的子句START WITH start更改该序列被记录的开始值。 这对于当前序列值没有影响，它会简单地设置 未来ALTER SEQUENCE RESTART命令将会使用的值。

restart

可选的子句RESTART [WITH restart]更改该序列的 当前值。这类似于用is_called = false 调用setval函数：被指定的值将会被 下一次nextval调用返回。写上没有restart值的 RESTART等效于提供被 CREATE SEQUENCE记录的或者上一次被 ALTER SEQUENCE START WITH设置的开始值。

与setval调用相比，序列上的RESTART 操作是事务性的并阻止并发事务从同一序列中获取数字。 如果这不是所需的操作模式，则应使用setval。

cache

子句CACHE cache使得序列数字被预先 分配并且保存在内存中以便更快的访问。最小值是 1（每次只产生一个值，即 无缓存）。如果没有指定，旧的缓冲值将被保持。

CYCLE

可选的CYCLE关键词可以被用来允许该序列在达到 maxvalue（上升序列）或 minvalue（下降序列）时 回卷。如果到达该限制，下一个被产生的数字将分别是 minvalue或者 maxvalue。

NO CYCLE

如果指定了可选的NO CYCLE关键词，任何在该 序列到达其最大值后的nextval调用将会返回 一个错误。如果既没有指定CYCLE也没有指定 NO CYCLE，旧的循环行为将被保持。

OWNED BY table_name.column_name
OWNED BY NONE

OWNED BY选项导致该序列与一个特定的表列相关联， 这样如果该列（或者整个表）被删除，该序列也会被自动删除。如果指定， 这种关联会替代之前为该序列指定的任何关联。被指定的表必须具有相同的 拥有者并且与该序列在同一个模式中。指定 OWNED BY NONE可以移除任何现有的关联，让该序列 “自立”。

new_owner

该序列的新拥有者的用户名。

new_name

该序列的新名称。

new_schema

该序列的新模式。

注解

ALTER SEQUENCE将不会立即影响除当前后端外 其他后端中的nextval结果，因为它们有预分配（缓存）的序列 值。在注意到序列生成参数被更改之前它们将用尽所有缓存的值。当前后端将被 立刻影响。

ALTER SEQUENCE不会影响该序列的 currval状态（在 PostgreSQL 8.3 之前有时会影响）。

ALTER SEQUENCE阻塞并发nextval、 currval、lastval和 setval调用。

由于历史原因，ALTER TABLE也可以被用于序列， 但是只有等效于上述形式的ALTER TABLE变体才被 允许用于序列。

示例

在 105 重启一个被称为serial的序列：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

ALTER SEQUENCE符合SQL 标准，不过AS、START WITH、 OWNED BY、OWNER TO、RENAME TO 以及SET SCHEMA子句是 PostgreSQL扩展。

另见

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — 更改一个外部服务器的定义

大纲

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

描述

ALTER SERVER更改一个外部服务器的定义。 第一种形式更改该服务器的版本字符串或者该服务器的一般选项（至少要求 一个子句）。第二种形式更改该服务器的拥有者。

要修改该服务器，你必须是它的拥有者。此外为了修改拥有者，你必须拥有 该服务器并且是新拥有角色的一个直接或者间接成员，并且你必须具有该服 务器的外部数据包装器上的USAGE特权（注意超级用户自动 满足所有这些政策）。

参数

name

一个现有服务器的名称。

new_version

新的服务器版本。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

更改该服务器的选项。ADD、SET和 DROP指定要执行的动作。如果没有显式地指定操作，将会假定为ADD。选项名称必须唯一，名称和值也会 使用该服务器的外部数据包装器库进行验证。

new_owner

该外部服务器的新拥有者的用户名。

new_name

该外部服务器的新名称。

示例

修改服务器foo，增加连接选项：

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

修改服务器foo，更改版本、更改host选项：

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

兼容性

ALTER SERVER符合 ISO/IEC 9075-9 (SQL/MED)。 OWNER TO和RENAME形式是 PostgreSQL 扩展。

另见

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — 更改扩展统计对象的定义

大纲

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
ALTER STATISTICS name SET SCHEMA new_schema
```

描述

ALTER STATISTICS更改现有扩展统计对象的参数。 任何在ALTER STATISTICS命令中没有明确设定的参数保持它们之前的设置。

您必须拥有统计对象才能使用ALTER STATISTICS。 要更改统计对象的模式，还必须在新模式上具有CREATE权限。 要更改所有者，还必须是新所有者角色的直接或间接成员， 且该角色在统计对象的模式上必须具有CREATE权限。 （这些限制强制了通过删除和重新创建统计对象来改变所有者不会做任何你不能做的事情， 但是超级用户可以改变任何统计对象的所有权。）

参数

name

要修改的统计对象的名称（可能有模式修饰）。

new_owner

统计对象的新所有者的用户名。

new_name

统计对象的新名称。

new_schema

统计对象的新模式。

兼容性

SQL标准中没有ALTER STATISTICS命令。

又见

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — 修改订阅的定义

大纲

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...] [ WITH
 ( set_publication_option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name REFRESH PUBLICATION [ WITH ( refresh_option [= value]
 [, ... ] ) ]
ALTER SUBSCRIPTION name ENABLE
ALTER SUBSCRIPTION name DISABLE
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )
ALTER SUBSCRIPTION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION name RENAME TO new_name
```

描述

ALTER SUBSCRIPTION可以修改大部分可以在 CREATE SUBSCRIPTION中指定的订阅属性。

要使用ALTER SUBSCRIPTION，你必须拥有该订阅。要改变所有者， 你也必须是新所有者的直接或间接成员。新所有者必须是超级用户。（目前， 所有的订阅所有者必须是超级用户，所以所有者的检查将在实践中被绕过， 但这可能在未来发生变化。）

参数

name

要修改属性的订阅的名称。

CONNECTION 'conninfo'

该子句修改最初由CREATE SUBSCRIPTION设置的连接属性。

SET PUBLICATION publication_name

更改订阅发布的列表。参阅CREATE SUBSCRIPTION 获取更多信息。默认情况下，此命令也将像REFRESH PUBLICATION一样工作。

set_publication_option指定了这个操作的附加选项。 支持的选项是：

refresh (boolean)

如果为false，则该命令将不会尝试刷新表信息。然后应单独执行 REFRESH PUBLICATION。默认值是true。

此外，可以指定REFRESH PUBLICATION下描述的刷新选项。

REFRESH PUBLICATION

从发布者获取缺少的表信息。这将开始复制自上次调用REFRESH PUBLICATION 或从CREATE SUBSCRIPTION以来添加到订阅发布中的表。

refresh_option指定了刷新操作的附加选项。 支持的选项有：

copy_data (boolean)

指定在复制启动后是否应复制正在订阅的发布中的现有数据。 默认值是true。

ENABLE

启用先前禁用的订阅，在事务结束时启动逻辑复制工作。

DISABLE

禁用正在运行的订阅，在事务结束时停止逻辑复制工作。

SET (subscription_parameter [= value] [, ...])

该子句修改原先由CREATE SUBSCRIPTION设置的参数。 允许的选项是slot_name和synchronous_commit。

new_owner

订阅的新所有者的用户名。

new_name

订阅的新名称。

示例

将订阅的发布更改为insert_only：

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

禁用（停止）订阅：

```
ALTER SUBSCRIPTION mysub DISABLE;
```

兼容性

ALTER SUBSCRIPTION是PostgreSQL 的一个扩展。

又见

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — 更改一个服务器配置参数

大纲

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' |
 DEFAULT }

ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

描述

ALTER SYSTEM被用来在整个数据库集簇范围内更改 服务器配置参数。它比传统的手动编辑postgresql.conf 文件的方法更方便。ALTER SYSTEM会把给出的参数 设置写入到postgresql.auto.conf文件中，该文件会随着 postgresql.conf一起被读入。把一个参数设置为 DEFAULT或者使用RESET变体可以 把该配置项从postgresql.auto.conf文件中移除。使用 RESET ALL可以移除所有这类配置项。

用ALTER SYSTEM设置的值将在下一次重载服务器 配置后生效，那些只能在服务器启动时更改的参数则会在下一次服务器重启后生效。 重载服务器配置可以通过以下做法实现：调用 SQL函数pg_reload_conf()， 运行pg_ctl reload或者向主服务器进程发送一个SIGHUP信号。

只有超级用户能够使用ALTER SYSTEM。还有，由于 这个命令直接作用于文件系统并且不能被回滚，不允许在一个事务块或者函数中使用它。

参数

configuration_parameter

一个可设置配置参数的名称。可用的参数可见第 19 章

value

该参数的新值。值可以被指定为字符串常量、标识符、数字或者以上这些构成的 逗号分隔的列表，值的具体形式取决于特定的参数。写上 DEFAULT可以用来把该参数及其值从postgresql.auto.conf中移除。

注解

不能用这个命令来设置data_directory以及 postgresql.conf中不被允许的参数（例如preset options）。

其他设置参数的方法见第 19.1 节

示例

设置wal_level：

```
ALTER SYSTEM SET wal_level = replica;
```

撤销以上的设置，恢复postgresql.conf中有效的设置：

```
ALTER SYSTEM RESET wal_level;
```

兼容性

ALTER SYSTEM语句是一种 PostgreSQL扩展。

另见

SET, SHOW

ALTER TABLE

ALTER TABLE — 更改一个表的定义

大纲

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
    ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec |
 DEFAULT }
ALTER TABLE [ IF EXISTS ] name
    DETACH PARTITION partition_name
```

其中action 是以下之一：

```
    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation
 ] [ USING expression ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS
 IDENTITY [ ( sequence_options ) ]
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } |
 SET sequence_option | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
 DEFERRED | INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
```

```
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }

and partition_bound_spec is:

IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE }
 [, ...] )
  TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE }
 [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )

and column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
 SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

而table_constraint是：

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
 [, ... ] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
 [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON
 UPDATE action ] }
```

```
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

并且 table_constraint_using_index 是：

```
    [ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的index_parameters是：

```
[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

exclude_element in an EXCLUDE constraint is:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ]
```

描述

ALTER TABLE更改一个现有表的定义。下文描述了 几种形式。注意每一种形式所要求的锁级别可能不同。如果没有明确说明，将会 持有一个ACCESS EXCLUSIVE所。当列出多个子命令时，所 持有的锁将是子命令所要求的最严格的那一个。

ADD COLUMN [IF NOT EXISTS]

这种形式向该表增加一个新列，使用与 CREATE TABLE相同的语法。如果指定了 IF NOT EXISTS并且使用这个名字的列已经存在，则 不会抛出错误。

DROP COLUMN [IF EXISTS]

这种形式从表删除一列。涉及到该列的索引和表约束也将会被自动 删除。如果该列的移除会导致引用它的多元统计信息仅包含单一列的数据，则该多元统计信息也将被移除。如果在该表之外有任何东西（例如外键引用或者视图）依赖 于该列，你将需要用到CASCADE。如果指定了 IF EXISTS但该列不存在，则不会抛出错误。 这种情况中会发出一个提示。

SET DATA TYPE

这种形式更改表中一列的类型。涉及到该列的索引和简单表约束将通过 重新解析最初提供的表达式被自动转换为使用新的列类型。可选的 COLLATE子句为新列指定一种排序规则，如果被省略， 排序规则会是新列类型的默认排序规则。可选的USING 子句指定如何从旧的列值计算新列值，如果被省略，默认的转换和从旧类型 到新类型的赋值造型一样。如果没有从旧类型到新类型的隐式或者赋值造型， 则必须提供一个USING子句。

SET/DROP DEFAULT

这些形式为一列设置或者移除默认值。默认值只在后续的 INSERT或UPDATE命令中生效，它们不会导致已经在表中的行改变。

SET/DROP NOT NULL

这些形式更改一列是否被标记为允许空值或者拒绝空值。只有当该列 不包含空值时，你才能使用SET NOT NULL。

如果这个表是一个分区，对于在父表中被标记为NOT NULL的列，不能在其上执行DROP NOT NULL。要从所有的分区中删除NOT NULL约束，可以在父表上执行DROP NOT NULL。即使在父表上没有NOT NULL约束，这样的约束还是能被增加到分区上。也就是说，即便父表允许空值，子表也可以不允许空值，但反过来不行。

ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [IF EXISTS]

这些形式更改一列是否是一个标识列，或者是更改一个已有的标识列的产生属性。详情请参考CREATE TABLE。

如果DROP IDENTITY IF EXISTS被指定并且该列不是一个标识列，则不会有错误被抛出。在这种情况下会发出一个提示。

SET sequence_option
RESTART

这些形式修改位于一个现有标识列之下的序列。sequence_option是一个ALTER SEQUENCE所支持的选项，例如INCREMENT BY。

SET STATISTICS

这种形式为后续的ANALYZE操作设置针对每列 的统计收集目标。目标可以被设置在范围 0 到 10000 之间，还可以 把它设置为 -1 来恢复到使用系统默认的统计目标（ default_statistics_target）。更多有关 PostgreSQL查询规划器使用统计 信息的内容可见第 14.2 节

SET STATISTICS要求一个SHARE UPDATE EXCLUSIVE锁。

SET (attribute_option = value [, ...])
RESET (attribute_option [, ...])

这种形式设置或者重置每个属性的选项。当前，已定义的针对每个属性的 选项只有n_distinct和n_distinct_inherited， 它们会覆盖后续ANALYZE操作所得到的可区分值数量 估计。n_distinct影响该表本身的统计信息，而 n_distinct_inherited影响为该表外加其继承子女收集的统计信息。 当被设置为一个正值时，ANALYZE将假定该列刚好包含指定 数量的可区分非空值。当被设置为一个负值（必须大于等于 -1）时，ANALYZE将假定可区分非空值的数量与表的尺寸成线性比例， 确切的计数由估计的表尺寸乘以给定数字的绝对值计算得到。例如，值 -1 表示 该列中所有的值都是可区分的，而值 -0.5 则表示每一个值平均出现两次。当表 的尺寸随时间变化时，这会有所帮助，因为这种计算只有在查询规划时才会被 执行。指定值为 0 将回到正常的估计可区分值数量的做法。更多有关 PostgreSQL查询规划器使用统计 信息的内容可见第 14.2 节

更改针对每个属性的选项要求一个 SHARE UPDATE EXCLUSIVE锁。

SET STORAGE

这种形式为一列设置存储模式。这会控制这列是会被保持在线内还是放在一个 二级TOAST表中，以及数据是否应被压缩。对于 integer之类的定长、线内、未压缩值必须使用 PLAIN。MAIN用于线内、可压缩的 数据。EXTERNAL用于外部的、未压缩数据。而EXTENDED用于外部的、压缩数据。对于大部分支持 非-PLAIN存储的数据类型，EXTENDED是默认值。使用EXTERNAL将会让很大的 text和bytea之上的子串操作运行得更快， 但是代价是存储空间会增加。注意SET STORAGE本身并不改变 表中的任何东西，它只是设置在未来的表更新时要追求的策略。详见 第 68.2 节

ADD table_constraint [NOT VALID]

这种形式使用和CREATE TABLE相同的语法外加 NOT VALID选项为一个表增加一个新的约束，该选项 当前只被允许用于外键和 CHECK 约束。如果约束被标记为 NOT VALID，将会跳过验证表中所有行满足该约束的 初检，这种检查可能会很漫长。该约束仍将被强制到后续的插入和删除上 （也就是说，在外键的情况下如果在被引用表中没有一个匹配的行，操作 会失败；如果新行不匹配指定的检查约束，操作也会失败）。但是数据库 不会假定约束对该表中的所有行都成立，直到通过使用VALIDATE CONSTRAINT选项对它进行验证。当前，分区表上的外键约束不可以被声明为NOT VALID。

外键约束的增加要求在被引用表上的一个SHARE ROW EXCLUSIVE锁。

当唯一或者主键约束被添加到分区表时，会有额外的限制，请参考CREATE TABLE。

ADD table_constraint_using_index

这种形式基于一个已有的唯一索引为一个表增加新的 PRIMARY KEY或UNIQUE约束。该索引中的 所有列将被包括在约束中。

该索引不能有表达式列或者是一个部分索引。还有，它必须是一个带有 默认排序顺序的B-树索引。这些限制确保该索引等效于使用常规 ADD PRIMARY KEY或者ADD UNIQUE命令时创建的索引。

如果PRIMARY KEY被指定，并且该索引的列没有被标记 NOT NULL，那么这个命令将尝试对每一个这样的列做 ALTER COLUMN SET NOT NULL。这需要一次全表扫描 来验证这些列不包含空值。在所有其他情况中，这都是一种很快的操作。

如果提供了一个约束名，那么该索引将被重命名以匹配该约束名。否则 该约束将被命名成索引的名称。

这个命令被执行后，该索引被增加的约束“拥有”，这和用常规 ADD PRIMARY KEY或ADD UNIQUE命令 创建的索引一样。特别地，删掉该约束将会导致该索引也消失。

当前在分区表上不支持这种形式。

> **注意**
>
> 如果需要增加一个新的约束但是不希望长时间阻塞表更新，那么使用现有索引增加约束会有所帮助。要这样做，用 CREATE INDEX CONCURRENTLY创建该索引，并且 接着使用这种语法把它安装为一个正式的约束。例子见下文。

ALTER CONSTRAINT

这种形式修改之前创建的一个约束的属性。当前只能修改外键约束。

VALIDATE CONSTRAINT

这种形式验证之前创建为NOT VALID的外键或检查约束， 它会扫描表来确保对于该约束没有行不满足约束。如果约束已经被标记为合法， 则什么也不会发生。

在大型表上的验证可能是一个长时间的处理。把约束的验证和创建分离开来让 我们可以把验证过程推迟到系统闲时进行，或者可以得到额外的时间来更正已 经存在的错误从而避免新的错误。还要注意验证本身并不会在运行时阻止对表 的写命令。

验证只要求被修改表上的一个SHARE UPDATE EXCLUSIVE 锁。如果该约束是一个外键，则还会在被该约束引用的表上要求一个 ROW SHARE锁。

DROP CONSTRAINT [IF EXISTS]

这种形式在一个表上删除指定的约束，还有位于该约束之下的任何索引。如果IF EXISTS被指定并且该约束不存在，不会抛出错误。在这种情况下会发出一个提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于该表的触发器的触发设置。系统仍然知道被禁用触发器 的存在，但是即使它的触发事件发生也不会执行它。对于一个延迟触发器， 会在事件发生时而不是触发器函数真正被执行时检查其启用状态。可以禁 用或者启用用名称指定的单个触发器、

表上的所有触发器、用户拥有的触 发器（这个选项会排除内部生成的约束触发器，例如用来实现外键约束或 可延迟唯一和排除约束）。禁用或者启用内部生成的约束触发器要求超级 用户特权，这样做要小心因为如果这类触发器不被执行，约束的完整性当 然无法保证。

触发器引发机制也受到配置变量 session_replication_role的影响。当复制角色是 “origin”（默认）或者“local”时，被简单启用的 触发器将被触发。被配置为ENABLE REPLICA的触发 器只有在会话处于“replica”模式时才将被触发。被配置为ENABLE ALWAYS的触发器的触发不会考虑当前复制 角色。

这种机制的效果就是，在默认配置中，触发器不会在复制体上引发。这种效果很有用，因为如果一个触发器在源头上被用来在表之间传播数据，那么复制系统也将复制被传播的数据，并且触发器不应该在复制体上引发第二次，因为那会导致重复。不过，如果一个触发器被用于另一种目的（例如创建外部告警），那么将它设置为ENABLE ALWAYS可能更加合适，这样它在复制体上也会被引发。

这个命令要求一个SHARE ROW EXCLUSIVE锁。

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

这些形式配置属于表的重写规则的触发设置。系统仍然知道一个被禁用规则的 存在，但在查询重写时不会应用它。其语义与禁用的/启用的触发器的一样。 对于ON SELECT规则会忽略这个配置，即使当前会话处于 一种非默认的复制角色，这类规则总是会被应用以保持视图工作正常。

规则引发机制也受到配置变量session_replication_role的影响，这和上述的触发器类似。

DISABLE/ENABLE ROW LEVEL SECURITY

这些形式控制属于该表的行安全性策略的应用。如果被启用并且该表上 不存在策略，则将应用一个默认否定的策略。注意即使行级安全性被禁 用，在表上还是可以存在策略。在这种情况下，这些策略将 不 会被应用 并且会被忽略。另见CREATE POLICY。

NO FORCE/FORCE ROW LEVEL SECURITY

这些形式控制当用户是表拥有者时表上的行安全性策略的应用。如果被启用， 当用户是表拥有者时，行级安全性策略将被应用。如果被禁用（默认），则 当用户是表拥有者时，行级安全性将不会被应用。另见 CREATE POLICY。

CLUSTER ON

这种形式为未来的CLUSTER操作选择默认的索引。 它不会真正地对表进行聚簇。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

SET WITHOUT CLUSTER

这种形式从表中移除最近使用的 CLUSTER索引说明。这会影响未来的不指定索引 的聚簇操作。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

SET WITH OIDS

这种形式为表增加一个oid系统列（见 第 5.4 节。如果该表已经有 OID，则它 什么也不会做。

注意这不等效于ADD COLUMN oid oid，后者只是会增加一个 恰好名为oid的普通列而不是系统列。

SET WITHOUT OIDS

这种形式从该表移除oid系统列。这完全等效于 DROP COLUMN oid RESTRICT，不过如果没有 oid列它不会抱怨。

SET TABLESPACE

这种形式把该表的表空间更改为指定的表空间并且把该表相关联的数据文件 移动到新的表空间中。表上的索引（如果有）不会被移动，但是它们可以用 额外的SET TABLESPACE命令单独移动。当前数据库在 一个表空间中的所有表可以用ALL IN TABLESPACE形式 移动，这将会首先锁住所有将被移动的表然后逐个移动。这种形式也支持 OWNED BY，它将只移动指定角色所拥有的表。如果指 定了NOWAIT选项，则命令将在无法立刻获得所有所需 要的锁时失败。注意这个命令不移动系统目录，如果想要移动系统目录，应 该用ALTER DATABASE或者显式的 ALTER TABLE调用。对于这种形式来说， information_schema关系不被认为是系统目录的一部分， 因此它们将会被移动。另见CREATE TABLESPACE。

SET { LOGGED | UNLOGGED }

This form changes the table from unlogged to logged or vice-versa (see UNLOGGED). It cannot be applied to a temporary table.

SET (storage_parameter = value [, ...])

这种形式为该表更改一个或者更多存储参数。可用的参数请见 存储参数。注意这个 命令将不会立刻修改表内容，这取决于重写表以得到想要的结果可能需要的 参数。可以用VACUUM FULL、CLUSTER或者 ALTER TABLE的一种形式来强制一次表重写。对于规划器相关的参数，更改将从该表下一次被锁定开始生效，因此当前执行的查询不会受到影响。

对fillfactor、toast以及autovacuum存储参数，还有下面的规划器相关参数，将会拿取SHARE UPDATE EXCLUSIVE锁：effective_io_concurrency、parallel_workers、seq_page_cost、random_page_cost、n_distinct以及n_distinct_inherited。

> **注意**
>
> 虽然CREATE TABLE允许在WITH (storage_parameter)语法中指定 OIDS，但是 ALTER TABLE没有把OIDS当作一个存储 参数，而是使用SET WITH OIDS和SET WITHOUT OIDS形式来更改 OID 状态。

RESET (storage_parameter [, ...])

这种形式把一个或者更多存储参数重置到它们的默认值。和 SET一样，可能需要一次表重写来更新整个表。

INHERIT parent_table

这种形式把目标表增加为指定父表的一个新子女。随后，针对父亲的查询将 包括目标表中的记录。要被增加为一个子女，目标表必须已经包含和父表完 全相同的列（也可以有额外的列）。这些列必须具有匹配的数据类型，并且 如果它们在父表中具有NOT NULL约束，它们在子表中 也必须有NOT NULL约束。

也必须把子表约束与所有父表的CHECK约束进行匹配， 不过父表中那些被标记为非可继承（也就是用ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT 创建的）除外，它们会被忽略。所有匹配得上的子表约束不能被标记为不可 继承。当前，UNIQUE、PRIMARY KEY 以及FOREIGN KEY约束没有被考虑，但是这种情况可能 会在未来发生变化。

NO INHERIT parent_table

这种形式把目标表从指定父表的子女列表中移除。针对父表的查询将不再包括 来自目标表的记录。

OF type_name

这种形式把该表链接到一种组合类型，就好像CREATE TABLE OF所做的那样。该表的列名和类型列表必须精确地匹配 该组合类型。oid系统列的存在情况可以不同。该表必须 不从任何其他表继承。这些限制确保 CREATE TABLE OF能允许一个等价的表定义。

NOT OF

这种形式解除一个有类型的表和其类型之间的关联。

OWNER TO

这种形式把表、序列、视图、物化视图或外部表的拥有者改为指定用户。

REPLICA IDENTITY

这种形式更改被写入到预写式日志来标识被更新或删除行的信息。除非使用逻辑复制，这个选项将不会产生效果。DEFAULT（非系统表的默认值）记录主键列 （如果有）的旧值。USING INDEX记录被所提到的索引所覆盖的列的 旧值，该索引必须是唯一索引、不是部分索引、不是可延迟索引并且只包括被标记成 NOT NULL的列。FULL记录行中所有列的旧值。 NOTHING不记录有关旧行的任何信息（这是系统表的默认值）。在所 有情况下，除非至少有一个要被记录的列在新旧行版本之间发生变化，将不记录旧值。

RENAME

RENAME形式更改一个表（或者一个索引、序列、视图、物化视图 或者外部表）的名称、表中一个列的名称或者表的一个约束的名称。在重命名一个具有底层索引的约束时，该索引也会被重命名。它对已存储的数据 没有影响。

SET SCHEMA

这种形式把该表移动到另一个模式中。相关的该表列拥有的索引、约束和序列也会被 移动。

ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }

这种形式把一个已有表（自身也可能被分区）作为一个分区挂接到目标表。该表可以为特定的值使用FOR VALUES挂接为分区，或者用DEFAULT挂接为一个默认分区。对于目标表中的每一个索引，在被挂接的表上都将创建一个响应的索引，如果已经存在等效的索引，该索引将被挂接到目标表的索引，就像执行了ALTER INDEX ATTACH PARTITION一样。

一个使用FOR VALUES的分区使用与CREATE TABLE中partition_bound_spec相同的语法。分区边界说明必须对应于目标表的分区策略以及分区键。要被挂接的表必须具有和目标表完全相同的所有列，并且不能有多出来的列，而且列的类型也必须匹配。此外，它必须有目标表上所有的NOT NULL以及CHECK约束。当前不考虑FOREIGN KEY约束。来自于父表的UNIQUE和PRIMARY KEY约束将被创建在分区上（如果它们还不存在）。如果被挂接的表上的任何CHECK约束被标记为NO INHERIT，则命令将失败，这类约束必须被重建且重建时不能有NO INHERIT子句。

如果新分区是一个常规表，会执行一次全表扫描来检查表中没有现有行违背分区约束。可以通过对表增加一个有效的CHECK约束来避免这种扫描，该约束可以在运行这个命令之前仅允许满足所需分区约束的行。 使用这样一个约束，就可以让表无需被扫描就能验证分区约束。但是，如果任一分区键是一个表达式并且该分区不接受NULL值，这种方式就无效了。如果挂接一个不接受NULL值的列表分区，还应该为分区键列增加NOT NULL约束，除非它是一个表达式。

如果新分区是一个外部表，则不需要验证该外部表中的所有行遵守分区约束（有关外部表上的约束请参考CREATE FOREIGN TABLE中的讨论）。

当一个表有默认分区时，定义新分区会更改默认分区的分区约束。默认分区不能包含任何需要被移动到新分区中的行，并且将被扫描以验证不存在那样的行。如果一个合适

的CHECK约束存在，这种扫描（和新分区的扫描一样）可以被避免。还是和新分区的扫描一样，当默认分区是外部表时这种扫描总是会被跳过。

DETACH PARTITION partition_name

这种形式会分离目标表的指定分区。被分离的分区继续作为独立的表存在，但是与它之前挂接的表不再有任何联系。任何被挂接到目标表索引的索引也会被分离。

除了RENAME、SET SCHEMA、ATTACH PARTITION和DETACH PARTITION之外，所有形式的ALTER TABLE都作用在单个表上，前面这些形式可以被组合成一个多修改的列表被一起应用。例如，可以在一个命令中增加多个列并且/或者修改多个列的类型。对于大型表来说这会特别有用，因为只需要对表做一趟操作。

要使用ALTER TABLE，你必须拥有该表。要更改一个表的 模式或者表空间，你还必须拥有新模式或表空间上的 CREATE特权。要把一个表作为一个父表的新子表加入， 你必须也拥有该父表。此外，要把一个表挂接为另一个表的新分区，你必须拥有被挂接的表。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该表的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建表做不到的事情。不过，一个超级用户怎么都能更改任何表的所有权。）。 要增加一个列、修改一列的类型或者使用OF子句，你还必 须具有该数据类型上的USAGE特权。

参数

IF EXISTS

如果表不存在则不要抛出一个错误。这种情况下会发出一个提示。

name

要修改的一个现有表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，则只会修改该表。如果没有指定ONLY， 该表及其所有后代表（如果有）都会被修改。可选地，在表名后面可以指定 *用来显式地指示包括后代表。

column_name

一个新列或者现有列的名称。

new_column_name

一个现有列的新名称。

new_name

该表的新名称。

data_type

一个新列的数据类型或者一个现有列的新数据类型。

table_constraint

该表的新的表约束。

constraint_name

一个新约束或者现有约束的名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图）， 并且接着删除依赖于那些对象的 所有对象（见第 5.13 节。

RESTRICT

如果有任何依赖对象时拒绝删除列或者约束。这是默认行为。

trigger_name

一个要禁用或启用的触发器的名称。

ALL

禁用或者启用属于该表的所有触发器（如果有任何触发器是内部产生的约 束触发器则需要超级用户特权，例如那些被用来实现外键约束或者可延迟 一致性和排他约束的触发器）。

USER

禁用或者启用属于该表的所有触发器，内部产生的约束触发器（例如那些 被用来实现外键约束或者可延迟一致性和排他约束的触发器）除外。

index_name

一个现有索引的名称。

storage_parameter

一个表存储参数的名称。

value

一个表存储参数的新值。根据该参数，该值可能是一个数字或者一个词。

parent_table

要与这个表关联或者解除关联的父表。

new_owner

该表的新拥有者的用户名。

new_tablespace

要把该表移入其中的表空间的名称。

new_schema

要把该表移入其中的模式的名称。

partition_name

要被作为新分区附着到这个表或者从这个表上分离的表的名称。

partition_bound_spec

新分区的分区边界说明。更多细节请参考CREATE TABLE中相同的语法。

注解

关键词COLUMN是噪声，可以被省略。

在使用ADD COLUMN增加一列并且指定了一个非易失性DEFAULT时，默认值会在该语句执行时计算并且结果会被保存在表的元数据中。这个值将被用于所有现有行的该列。如果没有指定DEFAULT，则使用NULL。在两种情况下都不需要重写表。

增加一个带有非易失性DEFAULT子句的列或者更改一个现有列的类型将 要求重写整个表及其索引。在更改一个现有列的类型时有一种例外：如果 USING子句不更改列的内容并且旧类型在二进制上与新类型可 强制转换或者是新类型上的一个未约束域，则不需要重写表。但是受影响列上 的任何索引仍必须被重建。增加或者移除一个系统oid列也要求 重写整个表。对于一个大型表，表和/或索引重建可能会消耗相当多的时间， 并且会临时要求差不多两倍的磁盘空间。

增加一个CHECK或者NOT NULL约束要求扫描 表以验证现有行符合该约束，但是不要求一次表重写。

类似地，在挂接一个新分区时，它需要被扫描以验证现有行满足该分区约束。

提供在一个ALTER TABLE中指定多个更改的选项的主要 原因就是多次表扫描或者重写可以因此被整合成一次。

DROP COLUMN形式不会在物理上移除列，而只是简 单地让它对 SQL 操作不可见。后续该表中的插入和更新操作将为该列存储 一个空值。因此，删除一个列很快，但是它不会立刻减少表所占的磁盘空间， 因为被删除列所占用的空间还没有被回收。随着现有列被更新，空间将被逐渐 回收（这些说法不适用于删除系统oid列的情况，那时会立刻 使用重写来完成）。

要强制立即回收被已删除列占据的空间，你可以执行一种能导致全表重写的 ALTER TABLE形式。这种形式会导致重新构造每一个把被 删除列替换为空值的行。

ALTER TABLE的重写形式对于 MVCC 是不安全的。 在一次表重写之后，如果并发事务使用的是一个在重写发生前取得的 快照，该表将对这些并发事务呈现出空表的形态。详见第 13.5 节

SET DATA TYPE的USING选项能实际指定 涉及该列旧值的任何表达式。也就是说，它可以不但可以引用要被转换的列， 还可以引用其他列。这允许使用SET DATA TYPE语法完成十分 普遍的转换。由于这种灵活性，USING表达式不适合于列 的默认值（如果有），结果可能不是一个默认值所需的常量表达式。这意味着 在没有从旧类型到新类型的隐式或者赋值造型时，即便提供了一个 USING子句，SET DATA TYPE还是可能无法 转换默认值。在这种情况下，用DROP DEFAULT删除该默认值， 执行ALTER TYPE并且接着使用SET DEFAULT增加 一个合适的新默认值。类似的考虑也适用于涉及该列的索引和约束。

如果一个表有任何后代表，在不对后代表做相同操作的情况下，不允许在父表中增加列、重命名列或者更改列的类型。这确保了后代总是具有和父表匹配的列。类似地，如果不对所有后代上的CHECK约束进行重命名，就不能在父表中重命名该CHECK约束，这样CHECK约束也能在父表及其后代之间保持匹配（不过，这个限制不适用于基于索引的约束）。此外，因为从父表中选择也会从其后代中选择，父表上的约束不能被标记为有效，除非它在那些后代上也被标记为有效。在所有这些情况下，ALTER TABLE ONLY都将被拒绝。

只有当一个后代表的列不是从任何其他父表继承而来并且没有该列的独立定义时， 一次递归的DROP COLUMN操作才会移除该列。一次非递归 的DROP COLUMN（即 ALTER TABLE ONLY ... DROP COLUMN）不会移除 任何后代列，而是会把它们标记成独立定义的列。对于一个分区表，一个非递归的DROP COLUMN命令将会失败，因为一个表的所有分区都必须有和分区根节点相同的列。

标识列的动作（ADD GENERATED、SET等、DROP IDENTITY）以及动作TRIGGER、CLUSTER、OWNER和TABLESPACE不会递归到后代表上，也就是说它们执行时总是好像指定了ONLY一样。增加约束的动作仅对没有标记为NO INHERIT的CHECK约束递归。

不允许更改一个系统目录表的任何部分。

可用参数的进一步描述请见CREATE TABLE。 第 5 章有关于继承的进一步信息。

示例

要向一个表增加一个类型为varchar的列：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

要从表中删除一列：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

要在一个操作中更改两个现有列的类型：

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

通过一个USING子句更改一个包含 Unix 时间戳的整数列为 timestamp with time zone：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

同样的，当该列具有一个不能自动造型成新数据类型的默认值表达式时：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

重命名一个现有的约束：

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

为一列增加一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

从一列移除一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

向一个表及其所有子女增加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

只向一个表增加一个检查约束（不为其子女增加）：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5)
 NO INHERIT;
```

（该检查约束也不会被未来的子女继承）。

从一个表及其子女移除一个检查约束：

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

只从一个表移除一个检查约束：

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（该检查约束仍为子女表保留在某个地方）。

为一个表增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address);
```

为一个表增加一个外键约束，并且尽量不要影响其他工作：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

为一个表增加一个（多列）唯一约束：

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id,
 zipcode);
```

为一个表增加一个自动命名的主键约束，注意一个表只能拥有一个主键：

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

把一个表移动到一个不同的表空间：

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

把一个表移动到一个不同的模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

重建一个主键约束，并且在重建索引期间不阻塞更新：

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

要把一个分区挂接到一个范围分区表上：

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO
 ('2016-08-01');
```

要把一个分区挂接到一个列表分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

要把一个分区挂接到一个哈希分区表上：

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

要把一个默认分区挂接到一个分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

从一个分区表分离一个分区：

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

兼容性

形式ADD（没有USING INDEX）、 DROP [COLUMN]、DROP IDENTITY、RESTART、 SET DEFAULT、SET DATA TYPE（没有USING)、 SET GENERATED以及SET sequence_option服从SQL标准。其他形式都是PostgreSQL对SQL标准的扩展。此外，在单个ALTER TABLE命令中指定多个操作的能力是一种扩展。

ALTER TABLE DROP COLUMN可以被用来删除一个表的唯一的 列，从而留下一个零列的表。这是一种 SQL 的扩展，SQL 中不允许零列的表。

另见

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — 更改一个表空间的定义

大纲

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

描述

ALTER TABLESPACE可以被用于更改一个 表空间的定义。

要更改一个表空间的定义，你必须拥有它。要修改拥有者，你还必须是 新拥有角色的一个直接或间接成员（注意超级用户自动拥有这些特权）。

参数

name

一个现有表空间的名称。

new_name

该表空间的新名称。新名称不能以pg_开始，因为这类名称被 保留用于系统表空间。

new_owner

该表空间的新拥有者。

tablespace_option

要设置或者重置的一个表空间参数。当前，唯一可用的参数是 seq_page_cost、random_page_cost和effective_io_concurrency。 为一个特定表空间设定这两个参数值将覆盖规划器对从该表空间中的表读取 页面代价的估计值，这些估计值由具有相同名称配置参数建立（见 seq_page_cost、random_page_cost、effective_io_concurrency）。如果一个表空间位于一个比 其余I/O 子系统更快或者更慢的磁盘上时，这些参数就能派上用场。

示例

将表空间index_space重命名为fast_raid：

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

更改表空间index_space的拥有者：

```
ALTER TABLESPACE index_space OWNER TO mary;
```

兼容性

在 SQL 标准中没有 ALTER TABLESPACE语句。

另见

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — 更改一个文本搜索配置的定义

大纲

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary
 WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH CONFIGURATION 更改一个文本搜索配置的定义。你可以修改其从记号类型到词典的映射 或者更改该配置的名称或者拥有者。

要使用ALTER TEXT SEARCH CONFIGURATION， 你必须是该配置的拥有者。

参数

name

一个现有文本搜索配置的名称（可以是模式限定的）。

token_type

由该配置的解析器发出的记号类型的名称。

dictionary_name

在其中查阅指定记号类型的文本搜索字典的名称。如果列出了 多个字典，会按照指定的顺序查阅它们。

old_dictionary

在映射中要替换的文本搜索字典的名称。

new_dictionary

被用来替代old_dictionary 的文本搜索字典的名称。

new_name

该文本搜索配置的新名称。

new_owner

该文本搜索配置的新拥有者。

new_schema

该文本搜索配置的新模式。

ADD MAPPING FOR形式会安装一些词典（用列表列出）用于在其中 查阅指定的记号类型。如果对任一记号类型已经有一个映射，则会发生错误。 ALTER MAPPING FOR形式做同样的事情，但是首先会移除这些记号 类型的任何现有映射。ALTER MAPPING REPLACE形式用 new_dictionary来替换任何位 置上的old_dictionary。当出 现FOR时，只会为指定的记号类型做这样的事情。如果不出现 FOR，则会为该配置中所有的映射都这样做。 DROP MAPPING形式会移除指定记号类型的所有字典，导致该文本 搜索配置忽略这些类型。除非出现IF EXISTS，在那些记号类型没有 任何映射时会发生错误。

示例

下面的例子把my_config中任何位置上的english字典 替换为swedish字典。

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH CONFIGURATION 语句。

另见

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — 更改一个文本搜索字典的定义

大纲

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH DICTIONARY更改一个 文本搜索字典的定义。你可以更改该字典的与模板相关的选项，或者更改该 字典的名称或者拥有者。

要使用 ALTER TEXT SEARCH DICTIONARY，你必须是超级用户。

参数

name

一个现有的文本搜索字典的名称（可以是模式限定的）。

option

要为这个字典设置的与模板相关的选项的名称。

value

用于一个模板相关选项的新值。如果等号和值被忽略，则会从该字典 中移除该选项之前的设置而允许使用默认值。

new_name

该文本搜索字典的新名称。

new_owner

该文本搜索字典的新拥有者。

new_schema

该文本搜索字典的新模式。

模板相关的选项可以以任何顺序出现。

示例

下面的命令更改一个基于 Snowball 的字典的停用词列表。其他参数保持不变。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

下面的命令更改语言选项为dutch，并且完全移除停用词选项。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

下面的命令“更新”该字典的定义，但是实际没有做任何更改。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

（之所以能这样做是因为选项移除代码在选项不存在时也不会抱怨）。 这种技巧在为该字典更改配置文件时有用：ALTER 将强制现有的数据库会话重读配置文件，否则如果会话之前已经读取过 就不会再次读取。

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH DICTIONARY语句。

另见

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — 更改一个文本搜索解析器的定义

大纲

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH PARSER更改一个文本 搜索解析器的定义。当前，唯一支持的功能是更改该解析器的名称。

要使用ALTER TEXT SEARCH PARSER，你必须是超级用户。

参数

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

new_name

该文本搜索解析器的新名称。

new_schema

该文本搜索解析器的新模式。

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH PARSER语句。

另见

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — 更改一个文本搜索模板的定义

大纲

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH TEMPLATE更改一个 文本搜索模板的定义。当前唯一支持的功能是更改该模板的名称。

要使用ALTER TEXT SEARCH TEMPLATE，你必须是超级用户。

参数

name

一个现有文本搜索模板的名称（可以是模式限定的）。

new_name

该文本搜索模板的新名称。

new_schema

该文本搜索模板的新模式。

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH TEMPLATE语句。

另见

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — 更改一个触发器的定义

大纲

```
ALTER TRIGGER name ON table_name RENAME TO new_name
ALTER TRIGGER name ON table_name DEPENDS ON EXTENSION extension_name
```

描述

ALTER TRIGGER更改一个现有触发器的属性。 RENAME子句更改给定触发器的名称而不更改其定义。 DEPENDS ON EXTENSION子句把该触发器标记为依赖于 一个扩展，这样如果扩展被删除，该触发器也会被自动删除。

要更改一个触发器的属性，你必须拥有该触发器所作用的表。

参数

name

要修改的一个现有触发器的名称。

table_name

这个触发器所作用的表的名称。

new_name

该触发器的新名称。

extension_name

该触发器所依赖的扩展的名称。

注解

临时启用或者禁用一个触发器的功能由ALTER TABLE而不是 ALTER TRIGGER提供，因为ALTER TRIGGER 无法表示一次性启用或者禁用一个表上所有触发器的选项。

示例

要重命名一个现有的触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

要把一个触发器标记为依赖于一个扩展：

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

兼容性

ALTER TRIGGER是一种 PostgreSQL的 SQL 标准扩展。

另见

ALTER TABLE

ALTER TYPE

ALTER TYPE — 更改一个类型的定义

大纲

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE
 | RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
 AFTER } neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
```

这里action 是以下之一：

```
    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
 RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type
 [ COLLATE collation ] [ CASCADE | RESTRICT ]
```

描述

ALTER TYPE更改一种现有类型的定义。 它有几种形式：

ADD ATTRIBUTE

这种形式为一种组合类型增加一个新属性，使用的语法和 CREATE TYPE相同。

DROP ATTRIBUTE [IF EXISTS]

这种形式从一种组合类型删除一个属性。如果指定了 IF EXISTS并且该属性不存在，则不会抛出错误。 这种情况下会发出一个提示。

SET DATA TYPE

这种形式更改一种组合类型的一个属性类型。

OWNER

这种形式更改该类型的拥有者。

RENAME

这种形式更改该类型的名称或者一种组合类型的一个属性的名称。

SET SCHEMA

这种形式把该类型移动到另一个模式中。

ADD VALUE [IF NOT EXISTS] [BEFORE | AFTER]

这种形式为一种枚举类型增加一个新值。可以用BEFORE或者 AFTER一个现有值来指定新值在枚举顺序中的位置。 否则，新项会被增加在值列表的最后。

如果指定了IF NOT EXISTS，该类型已经包含新值时不会发生 错误：会发出一个提示但是不采取其他行动。否则，如果新值已经存在会发生错误。

RENAME VALUE

该形式重命名枚举类型的值。该值在枚举排序中的位置不受影响。 如果指定的值不存在或新名称已存在，则会发生错误。

ADD ATTRIBUTE、DROP ATTRIBUTE和ALTER ATTRIBUTE动作 可以被整合到一个多个修改组成的列表中，以便被平行应用。例如， 可以在一个命令中增加多个属性并且/或者修改多个属性的类型。

要使用ALTER TYPE，你必须拥有该类型。要更改 一个类型的模式，你还必须拥有新模式上的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该类型的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建该类型做不到的事情。不过，一个超级用户怎么都能更改任何类型的所有权。）。要增加一个属性或者修改一个属性类型，你还必须具有该数据类型上的 USAGE特权。

参数

name

要修改的一个现有类型的名称（可能被模式限定）。

new_name

该类型的新名称。

new_owner

该类型新拥有者的用户名。

new_schema

该类型的新模式。

attribute_name

要增加、修改或者删除的属性名称。

new_attribute_name

要被重命名的属性的新名称。

data_type

要增加的属性的数据类型，或者是要修改的属性的新类型。

new_enum_value

要被增加到一个枚举类型的值列表的新值，或将赋予现有值的新名称。 和所有枚举文本一样，它需要被引号引用。

neighbor_enum_value

一个现有枚举值，新值应该被增加在紧接着该枚举值之前或者 之后的位置上。和所有枚举文本一样，它需要被引号引用。

existing_enum_value

现有的应该重命名的枚举值。和所有的枚举文本一样，它需要被引号引用。

CASCADE

自动将操作传播到被更改类型的类型表及其后代。

RESTRICT

如果被更改的类型是类型表的类型，则拒绝该操作。这是默认设置。

注解

ALTER TYPE ... ADD VALUE（增加一个新值到枚举类型的形式） 不能在一个事务块中执行。

涉及到一个新增加枚举值的比较有时会被只涉及原始枚举值的比较更慢。这通常 只会在利用BEFORE或者AFTER来把新值的 排序位置设置为非列表结尾的地方时发生。不过，有时候即使把新值增加到最后时 也会发生这种情况（如果创建了该枚举类型之后出现过 OID 计数器 “回卷”，就会发生这种情况）。这种减速通常不明显，但是如果它确实 带来了麻烦，通过删除并且重建该枚举类型或者转储并且重载整个数据库可以重新 得到最优性能。

示例

要重命名一个数据类型：

```
ALTER TYPE electronic_mail RENAME TO email;
```

把类型email的拥有者改为 joe：

```
ALTER TYPE email OWNER TO joe;
```

把类型email的模式改为 customers：

```
ALTER TYPE email SET SCHEMA customers;
```

增加一个新属性到一个类型：

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

在一个特定的排序位置上为一个枚举类型增加一个新值：

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

重命名一个枚举值：

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

兼容性

增加和删除属性的变体是 SQL 标准的一部分，而其他变体是 PostgreSQL 扩展。

另见

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — 更改一个数据库角色

大纲

```
ALTER USER role_specification [ WITH ] option [ ... ]
```

其中 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
```

```
ALTER USER name RENAME TO new_name
```

```
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中 role_specification 可以是：

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

描述

ALTER USER现在是 ALTER ROLE的一种别名。

兼容性

ALTER USER语句是一种 PostgreSQL扩展。SQL 标准把用户的定义留给 实现来处理。

另见

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — 更改一个用户映射的定义

大纲

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER |
 PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

描述

ALTER USER MAPPING更改一个用户映射的定义。

一个外部服务器的拥有者可以为任何用户修改用于该服务器的用户映射。还有， 如果一个用户被授予了外部服务器上的USAGE特权，它就能为它们自 己的用户名修改一个用户映射。

参数

user_name

该映射的用户名。CURRENT_USER和USER匹配当前 用户的名称。PUBLIC被用来匹配系统中所有当前以及未来的用户名。

server_name

该用户映射的服务器名。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

为该用户映射更改选项。新选项会覆盖任何之前指定的选项。ADD、 SET和DROP指定要被执行的动作。如果没有显式地指定 操作，将假定为ADD。选项名称必须为唯一，该服务器的外部数据包装 器也会验证选项。

示例

为服务器 foo的用户映射bob更改口令：

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

兼容性

ALTER USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。有一点细微的语法问题：该标准会忽略FOR关键字。 由于CREATE USER MAPPING以及 DROP USER MAPPING都在类似的位置上使用FOR，并且 IBM DB2 （作为其他主要的 SQL/MED 实现）也在 ALTER USER MAPPING中要求该关键字，因此为了一致性和互 操作性，PostgreSQL 在这里的实现与标准不同。

另见

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — 更改一个视图的定义

大纲

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET
 DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value]
 [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

描述

ALTER VIEW更改一个视图的多种辅助属性（如果想要 修改视图的查询定义，应使用CREATE OR REPLACE VIEW）。

要使用ALTER VIEW，你必须拥有该视图。要更改一个视图的模式， 你还必须具有新模式上的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该视图的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建视图做不到的事情。不过，一个超级用户怎么都能更改任何视图的所有权。）。

参数

name

一个现有视图的名称（可以是模式限定的）。

IF EXISTS

该视图不存在时不要抛出一个错误。这种情况下会发出一个提示。

SET/DROP DEFAULT

这些形式为一个列设置或者移除默认值。对于任何在该视图上的 INSERT或者UPDATE命令，一个视图列的默认值 会在引用该视图的任何规则或触发器之前被替换进来。因此，该视图的默认值将会 优先于来自底层关系的任何默认值。

new_owner

该视图的新拥有者的用户名。

new_name

该视图的新名称。

new_schema

该视图的新模式。

```
SET ( view_option_name [= view_option_value] [, ... ] )
RESET ( view_option_name [, ... ] )
```

设置或者重置一个视图选项。当前支持的选项有：

check_option (string)

更改该视图的检查选项。值必须是local 或者cascaded。

security_barrier (boolean)

更改该视图的安全屏障属性。值必须是一个布尔值，如 true或者false。

注解

由于历史原因，ALTER TABLE也可以用于视图，但是 只允许等效于以上形式的ALTER TABLE变体用于视图。

示例

把视图foo重命名为 bar：

```
ALTER VIEW foo RENAME TO bar;
```

要为一个可更新视图附加一个默认列值：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

兼容性

ALTER VIEW是一种PostgreSQL 的 SQL 标准扩展。

另见

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — 收集有关一个数据库的统计信息

大纲

```
ANALYZE [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
ANALYZE [ VERBOSE ] [ table_and_columns [, ...] ]
```

其中option可以是：

```
    VERBOSE
```

table_and_columns是：

```
    table_name [ ( column_name [, ...] ) ]
```

描述

ANALYZE收集一个数据库中的表的内容的统计信息，并且将结果存储在pg_statistic系统目录中。接下来，查询规划器会使用这些统计信息来帮助确定查询最有效的执行计划。

如果没有table_and_columns列表，则ANALYZE处理当前用户有权分析的当前数据库中的每个表和物化视图。使用列表，ANALYZE仅处理那些表。还可以给出表的列名列表，在这种情况下，仅收集这些列的统计信息。

当选项列表用括号括起来时，选项可以按任何顺序来写。 带括号的语法是在PostgreSQL 11中添加的；不带括号的语法已弃用。

参数

VERBOSE

允许显示进度消息。

table_name

要分析的一个指定表的名称（可以是模式限定的）。如果省略，则分析当前数据库中的所有常规表、分区表和物化视图（但不包含外部表）。 如果指定的表是分区表，则整个分区表的继承统计信息和各个分区的统计信息都将更新。

column_name

要分析的一个指定列的名称。默认是所有列。

输出

当指定了VERBOSE时，ANALYZE会发出进度消息来指示当前正在处理哪个表。还会打印有关那些表的多种统计信息。

注解

只有被显式选中时才会分析外部表。并非所有外部数据包装器都支持ANALYZE。如果表的包装器不支持ANALYZE，该命令会打印一个警告并且什么也不做。

在默认的PostgreSQL配置中，自动清理守护进程（见第 24.1.6 节）会在表第一次载入数据或者用常规操作改变时负责表的自动分析。当启用自动清理时，定期运行ANALYZE是个好主意，或者可以在表内容做了大的修改后运行ANALYZE。准确的统计信息将帮助规划器选择最合适的查询计划，从而提升查询处理的速度。主读数据库的一般策略是在一天中使用量最低时运行一次VACUUM和ANALYZE（如果有大量的更新动作则是不够的）。

ANALYZE只要求目标表上的一个读锁，因此它可以和表上的其他动作并行。

ANALYZE收集的统计信息通畅包括每列中最常见值的列表以及展示每列中近似数据分布的一个直方图。如果ANALYZE认为这些东西无趣（例如在一个唯一键列中，没有共同值）或者该列的数据类型不支持合适的操作符，以上工作都会被省略。在第 24 章有与统计信息相关的更多信息。

对于大型的表，ANALYZE会对表内容做随机采样而不是检查每一行。这允许在很少的时间内完成对大型表的分析。不过要注意，这些统计信息只是近似值，并且即使实际表内容没有改变，每次运行ANALYZE时统计信息都会有微小地改变。这可能会导致EXPLAIN显示的规划器估算代价有小的改变。在很少的情况下，这会非决定性地导致规划器的查询计划选择在ANALYZE运行后改变。为了避免这种情况，可以按照下文所述提高ANALYZE所收集的统计信息量。

通过调整default_statistics_target配置变量可以控制分析量，对每个列可以用ALTER TABLE ... ALTER COLUMN ... SET STATISTICS设置每列的统计信息目标（见ALTER TABLE）。目标值会设置最常用值列表中的最大项数以及直方图中的最大容器数。默认目标值是 100，可以把它调大或者调小在规划器估计值精度和ANALYZE花费的时间以及pg_statistic所占空间之间做出平衡。特别地，将统计信息目标设置为零会禁用该列的统计信息收集。在查询的WHERE、GROUP BY或者ORDER BY子句中从不出现的列上这样做会有所帮助，因为规划器用不上这些列上的统计信息。

被分析的列中最大的统计信息目标决定了为准备统计信息要采样的表行数。增加该目标会导致做ANALYZE所需的时间和空间成比例增加。

ANALYZE所估算的值之一是出现在每个列中的可区分值。因为只会检查行的一个子集，即便使用最大的统计信息目标，这种估计有时也可能很不精确。如果这种不精确导致不好的查询计划，可以手工确定一个更精确的值并且用ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)设置该值（见ALTER TABLE）。

如果被分析的表有一个或者更多子女，ANALYZE将会收集两次统计信息：一次只对父表的行收集，第二次则在父表及其所有子女表的行上收集。在规划需要遍历整个继承树的查询时需要第二个统计信息集。不过，在决定是否触发表上的自动分析时，自动清理后台进程将只考虑父表本身上的插入和更新。如果该表很少被插入或者更新，只有手工运行ANALYZE时才会把继承统计信息更新到最新。

如果任何子表是外部表并且其外部数据包装器不支持ANALYZE，在收集继承统计信息时会忽略那些子表。

如果被分析的表不完全为空，ANALYZE将不会为该表记录新统计信息。任何现有统计信息将会被保留。

兼容性

SQL 标准中没有ANALYZE语句。

另见

VACUUM, vacuumdb, 第 19.4.4 节,第 24.1.6 节

BEGIN

BEGIN — 开始一个事务块

大纲

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

其中 transaction_mode 是以下之一：

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

BEGIN开始一个事务块，也就是说所有 BEGIN命令之后的所有语句将被在一个 事务中执行，直到给出一个显式的COMMIT或 者ROLLBACK。默认情况下（没有 BEGIN）， PostgreSQL在 “自动提交”模式中执行事务，也就是说每个语句都 在自己的事务中执行并且在语句结束时隐式地执行一次提交（如果执 行成功，否则会完成一次回滚）。

在一个事务块内的语句会执行得更快，因为事务的开始/提交也要求可观 的 CPU 和磁盘活动。在进行多个相关更改时，在一个事务内执行多个语 句也有助于保证一致性：在所有相关更新还没有完成之前，其他会话将不 能看到中间状态。

如果指定了隔离级别、读/写模式或者延迟模式，新事务也会有那些特性， 就像执行了SET TRANSACTION一样。

参数

WORK
TRANSACTION

可选的关键词。它们没有效果。

这个语句其他参数的含义请参考 SET TRANSACTION。

注解

START TRANSACTION具有和BEGIN 相同的功能。

使用COMMIT或者 ROLLBACK来终止一个事务块。

在已经在一个事务块中时发出BEGIN将惹出一个警告 消息。事务状态不会被影响。要在一个事务块中嵌套事务，可以使用保 存点（见SAVEPOINT）。

由于向后兼容的原因，连续的 transaction_modes 之间的逗号可以被省略。

示例

开始一个事务块：

```
BEGIN;
```

兼容性

BEGIN是一种 PostgreSQL语言扩展。它等效于 SQL 标准的命令START TRANSACTION，它的参考页 包含额外的兼容性信息。

DEFERRABLE transaction_mode 是一种PostgreSQL语言扩展。

附带地，BEGIN关键词被用于嵌入式 SQL 中的一种 不同目的。在移植数据库应用时，我们建议小心对待事务语义。

另见

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CALL

CALL — 调用一个过程

大纲

```
CALL name ( [ argument ] [, ...] )
```

简介

CALL执行一个过程。

如果过程有任何输出参数，则会返回一个结果行，返回这些参数的值。

参数

name

过程的名称（可以被方案限定）。

argument

过程调用的一个输入参数。函数和过程调用语法的完整细节（包括参数的使用）请参考第 4.3 节

注解

用户必须有过程上的EXECUTE特权才能调用它。

要调用一个函数（不是过程），应使用SELECT。

如果在事务块中执行CALL，那么被调用的过程不能执行事务控制语句。只有当CALL在其自身的事务中执行时，才允许过程执行事务控制语句。

PL/pgSQL 以不同的方式处理CALL中的输出参数。 详见第 43.6.3 节

示例

```
CALL do_db_maintenance();
```

兼容性

CALL符合SQL标准。

另见

CREATE PROCEDURE

CHECKPOINT

CHECKPOINT — 强制一个事务日志检查点

大纲

```
CHECKPOINT
```

描述

一个检查点是事务日志序列中的一个点，在该点上所有数据文件 都已经被更新为反映日志中的信息。所有数据文件将被刷写到磁盘。 检查点期间发生的细节可见第 30.4 节

CHECKPOINT命令在发出时强制一个 立即的检查点，而不用等待由系统规划的常规检查点（由 第 19.5.2 节的设置控制）。 CHECKPOINT不是用来在普通操作中 使用的命令。

如果在恢复期间执行，CHECKPOINT 命令将强制一个重启点（见第 30.4 节 而不是写一个新检查点。

只有超级用户能够调用CHECKPOINT。

兼容性

CHECKPOINT命令是一种 PostgreSQL语言扩展。

CLOSE

CLOSE — 关闭一个游标

大纲

```
CLOSE { name | ALL }
```

描述

CLOSE释放与一个已打开游标相关 的资源。在游标被关闭后，不允许在其上做后续的操作。当不再 需要使用一个游标时应该关闭它。

当一个事务被COMMIT或者 ROLLBACK终止时，每一个非可保持 的已打开游标会被隐式地关闭。当创建一个可保持游标的事务通过 ROLLBACK中止时，该可保持游标会 被隐式地关闭。如果该创建事务成功地提交，可保持游标会保持打 开，直至执行一个显式的CLOSE或者 客户端连接断开。

参数

name

要关闭的已打开游标的名称。

ALL

关闭所有 已打开的游标。

注解

PostgreSQL没有一个显式的 OPEN游标语句，一个游标在被声明时 就被认为是打开的。使用DECLARE语句可以声明游标。

通过查询pg_cursors 系统视图可以看到所有可用的游标。

如果一个游标在一个保存点之后关闭，并且后来回滚到了这个保存点， 那么CLOSE不会被回滚，也就是说回滚 后游标仍然保持关闭。

示例

关闭游标liahona：

```
CLOSE liahona;
```

兼容性

CLOSE完全服从 SQL 标准。 CLOSE ALL是一种PostgreSQL 扩展。

另见

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — 根据一个索引聚簇一个表

大纲

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

描述

CLUSTER指示PostgreSQL 基于index_name 所指定的索引来聚簇 table_name 所指定的表。该索引必须已经定义在 table_name上。

当一个表被聚簇时，会基于索引信息对它进行物理上的排序。聚簇是一种　一次性的操作：当表后续被更新时，更改没有被聚簇。也就是说，不会尝　试根据新行或者被更新行的索引顺序来存储它们（如果想这样做，可以周　期性地通过发出该命令重新聚簇。还有，把表的fillfactor存储参数设置为小于 100% 有助于在更 新期间保持聚簇顺序，因为如果空间足够会把被更新行保留在同一个页面 中）。

当一个表被更新时，PostgreSQL　　会记住它是按照哪个索引聚簇的。形式　　CLUSTER table_name 会使用前面所用的同一个索引对表重新聚簇。你也可以使用 CLUSTER或者ALTER TABLE 的SET WITHOUT CLUSTER形式把索引设置为可用于 未来的聚簇操作，或者清除任何之前的设置。

不带任何参数的CLUSTER会重新聚簇调用用　户所拥有的当前数据库中已经被聚簇过的表（如果是超级用户调用，则是 所有已被聚簇过的表）。这种形式的 CLUSTER不能在一个事务块内执行。

当一个表被聚簇时，会在其上要求一个ACCESS　EXCLUSIVE锁。这会阻止任何其他数据库操作（包括读和写） 在CLUSTER结束前在该表上操作。

参数

table_name

　一个表的名称（可能是模式限定的）。

index_name

　一个索引的名称。

VERBOSE

　在每一个表被聚簇时打印一个进度报告。

注解

在随机访问一个表中的行时，表中数据的实际顺序是无关紧要的。　不过，如果你想要更多地访问其中一些数据，并且有一个索引把它　们分组在一起，使用CLUSTER就会带　来好处。如果你从一个表中要求一个范围的被索引值或者多行都匹　配的一个单一值，CLUSTER就会有所　帮助，因为一旦该索引标识出了第一个匹配行所在的表页，所有其　他匹配行很可能就在同一个表页中，并且因此节省了磁盘访问并且 提高了查询速度。

CLUSTER可以使用指定索引上的一次索引扫描　或者遵循排序的一次顺序扫描（如果索引是 B树）对表重新排序。 它将会基于规划器代价参数以及可用的统计信息来选择较快的方法。

在使用索引扫描时，会创建该表的一份临时拷贝，其中包含按索引顺序　排列的表数据。该表上每一个索引的临时拷贝也会被创建。因此，在磁　盘上需要至少等于表尺寸加上索引尺寸的综合的空闲空间。

在使用顺序扫描以及排序时，也会创建一个临时排序文件，因此临时空　间需求的峰值也就是表尺寸的两倍外加索引尺寸。这种方法通常比索引　扫描方法更快，但是如果磁盘空间需求是不能接受的，你可以通过临时 地把enable_sort设置为off来禁 用这种选择。

建议在聚簇前把maintenance_work_mem设　置为一个合理地比较大的值（但是不能超过你可以用于 CLUSTER操作的 RAM 容量）。

因为规划器会记录有关表顺序的统计信息，建议在新近被聚簇的表上　运行ANALYZE。否则，规划器可能会产生很差 的查询计划。

因为CLUSTER会记住哪些索引被聚簇，　我们可以第一次手动聚簇想要聚簇的表，然后设置一个定期运行的维护 脚本，其中执行不带任何参数的CLUSTER，这样那些 表就会被周期性地重新聚簇。

示例

基于索引employees_ind聚簇表 employees：

```
CLUSTER employees USING employees_ind;
```

使用之前用过的同一个索引聚簇employees表：

```
CLUSTER employees;
```

对数据库中以前被聚簇过的所有表进行聚簇：

```
CLUSTER;
```

兼容性

在 SQL 标准中没有CLUSTER语句。

为了兼容 8.3 之前的PostgreSQL版本，

```
CLUSTER index_name ON table_name
```

语法也被支持。

另见

clusterdb

COMMENT

COMMENT — 定义或者更改一个对象的注释

大纲

```
COMMENT ON
{
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN relation_name.column_name |
  CONSTRAINT constraint_name ON table_name |
  CONSTRAINT constraint_name ON DOMAIN domain_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  POLICY policy_name ON table_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  STATISTICS object_name |
  SUBSCRIPTION object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

其中 aggregate_signature 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

描述

COMMENT存储关于一个数据库对象的注释。

对每一个对象只保存一个注释字符串，因此为了修改一段注释，对同一个对象 发出一个新的COMMENT命令。要移除一段注释，可在文 本字符串的位置上写上NULL。当对象被删除时，其注释 也会被自动删除。

对大部分类型的对象，只有对象的拥有者可以设置注释。角色没有拥有者，因此 COMMENT ON ROLE的规则是你必须作为一个超级用户来对一个 超级用户角色设置注释，或者具有CREATEROLE特权来对非超级用 户角色设置注释。同样的，访问方法也没有拥有者，你必须作为一个超级用户来 对一个访问方法设置注释。当然，一个超级用户可以对任何东西设置注释。

使用psql的\d 命令家族可以查看注释。其他检索注释的用户接口可以构建在 psql使用的内建函数之上，即 obj_description、col_description 以及shobj_description （见表 9.68 。

参数

object_name
relation_name.column_name
aggregate_name
constraint_name
function_name
operator_name
policy_name
procedure_name
routine_name
rule_name
trigger_name

要被注释的对象的名称。表、聚集、排序方式、转换、域、外部表、函数、 索引、操作符、操作符类、操作符族、存储过程、例程、序列、统计信息、文本搜索对象、类型和视图 的名称可以被模式限定。在注释一列时， relation_name必须 引用一个表、视图、组合类型或者外部表。

table_name
domain_name

当在一个约束、触发器、规则或者策略上创建一段注释时，这些参数指定在其上定义 该对象的表或域的名称。

source_type

造型的源数据类型的名称。

target_type

造型的目标数据类型的名称。

argmode

一个函数，存储过程或者聚集函数的参数的模式：IN、 OUT、INOUT或者VARIADIC。 如果被省略，默认值是IN。注意 COMMENT并不真正关心 OUT参数，因为决定函数的身份只需要输入参数。因此 列出IN、INOUT和VARIADIC 参数就足够了。

argname

一个函数，存储过程或者聚集函数参数的名称。注意 COMMENT并不真正关心参数名称，因为决定函数的身份只需要参数数据类型。

argtype

一个函数，存储过程或者聚集函数参数的数据类型。

large_object_oid

大对象的 OID。

left_type
right_type

操作符的参数的数据类型（可以是模式限定的）。对一个前缀后后缀操作符 的缺失参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

text

写成一个字符串的新注释。如果要删除注释，写成NULL。

注解

当前对查看注释没有安全机制：任何连接到一个数据库的用户能够看到 该数据库中所有对象的注释。对于数据库、角色、表空间这类共享对象， 注释被全局存储，因此连接到集簇中任何数据库的任何用户可以看到共 享对象的所有注释。因此，不要在注释中放置有安全性风险的信息。

示例

为表mytable附加一段注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

移除它：

```
COMMENT ON TABLE mytable IS NULL;
```

更多的一些例子：

```
COMMENT ON ACCESS METHOD rtree IS 'R-Tree access method';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample
 variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
```

```
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of
 domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other
 database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for
 btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for
 btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Runs a report';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON STATISTICS my_statistics IS 'Improves planner row estimations';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish
 language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore
 and Python dict';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

兼容性

SQL 标准中没有COMMENT命令。

COMMIT

COMMIT — 提交当前事务

大纲

```
COMMIT [ WORK | TRANSACTION ]
```

描述

COMMIT提交当前事务。所有由该　事务所作的更改会变得对他人可见并且被保证在崩溃发生时仍能 持久。

参数

WORK
TRANSACTION

　可选的关键词。它们没有效果。

注解

使用ROLLBACK中止一个事务。

当不在一个事务内时发出COMMIT不会 产生危害，但是它会产生一个警告消息。

示例

要提交当前事务并且让所有更改持久化：

```
COMMIT;
```

兼容性

SQL 标准仅指定了两种形式： COMMIT和COMMIT WORK。除此之外，这个命令完全符合。

另见

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — 提交一个早前为两阶段提交预备的事务

大纲

```
COMMIT PREPARED transaction_id
```

描述

COMMIT PREPARED提交一个处于预备状态的事务。

参数

transaction_id

要被提交的事务的事务标识符。

注解

要提交一个预备的事务，你必须是原先执行该事务的同一用户或者超级用户。 但是不需要处于执行该事务的同一会话中。

这个命令不能在一个事务块中执行。该预备事务将被立刻提交。

pg_prepared_xacts 系统视图中列出了所有当前可用的预备事务。

例子

提交由事务标识符foobar标识的事务：

```
COMMIT PREPARED 'foobar';
```

兼容性

COMMIT PREPARED是一种 PostgreSQL扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — 在一个文件和一个表之间复制数据

大纲

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

其中 option 可以是下列之一：

```
    FORMAT format_name
    OIDS [ boolean ]
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

描述

COPY在 PostgreSQL表和标准文件系统文件之间 移动数据。COPY TO把一个表的内容复制 到一个文件，而COPY FROM 则从一个文件复制数据到一个表（把数据追加到表中原有数据）。COPY TO也能复制一个 SELECT查询的结果。

如果指定了一个列列表，COPY将只把指定列的 数据复制到文件或者从文件复制数据到指定列。如果表中有列不在列列表中， COPY FROM将会为那些列插入默认值。

带一个文件名的COPY指示 PostgreSQL服务器直接从一个文件读取 或者写入到一个文件。该文件必须是 PostgreSQL用户（运行服务器的用户 ID） 可访问的并且应该以服务器的视角来指定其名称。当指定了 PROGRAM时，服务器执行给定的命令并且从该程序的标准 输出读取或者写入到该程序的标准输入。该程序必须以服务器的视角指定，并且 必须是PostgreSQL用户可执行的。在指定 STDIN或者STDOUT时，数据会通过客 户端和服务器之间的连接传输。

参数

table_name

一个现有表的名称（可以是模式限定的）。

column_name

可选的要被复制的列列表。如果没有指定列列表，则该表的所有列都会被复制。

query

其结果要被复制的SELECT、 VALUES、 INSERT、UPDATE或者 DELETE命令。注意查询周围的圆括号是必要的。

对于INSERT、UPDATE以及 DELETE查询，必须提供一个 RETURNING 子句并且 目标关系不能具有会扩展成多条语句的条件规则、 ALSO规则或者INSTEAD规则。

filename

输入或者输出文件的路径名。一个输入文件的名称可以是一个绝对或相对路径， 但一个输出文件的名称必须是绝对路径。Windows 用户可能需要使用一个 E''字符串并且双写路径名称中使用的任何反斜线。

PROGRAM

一个要执行的命令。在COPY FROM中，输入 将从该命令的标准输出读取，而在COPY TO中，输出会 写入到该命令的标准输入。

注意该命令是由 shell 调用，因此如果你需要传递任何来自不可信来源的 参数给 shell 命令，你必须小心地剥离那些可能对 shell 有特殊意义的特殊 字符。出于安全原因，最好使用一个固定的命令字符串，或者至少避免传递 任何用户输入到其中。

STDIN

指定输入来自客户端应用。

STDOUT

指定输出会去到客户端应用。

boolean

指定选中的选项是应该被关闭还是打开。可以写TRUE、 ON或1来启用选项，写FALSE、OFF或0禁用它。 boolean值也可以被省略， 那样会假定为TRUE。

FORMAT

选择要读取或者写入的数据格式： text、 csv（逗号分隔值）或者binary。 默认是text。

OIDS

指定为每一行复制 OID（如果为一个没有 OID 的表指定了 OIDS或者正在复制的是一个查询，则会出现错误）。

FREEZE

请求复制已经完成了行冻结的数据，就好像在运行 VACUUM FREEZE命令之后复制。这是为了初始 数据载入的性能而设计的。只有被载入表已经在当前子事务中被创建 或截断、该事务中没有游标打开并且该事务没有持有更旧的快照时， 行才会被冻结。目前无法在分区表上执行COPY FREEZE。

注意一旦成功地载入，所有其他会话将能立即看到该数据。这违背了 普通的 MVCC 可见性规则，指定该选项的用户应该注意这可能会导致 的潜在问题。

DELIMITER

指定分隔文件每行中各列的字符。文本格式中默认是一个制表符， 而CSV格式中默认是一个逗号。这必须是一个单一 的单字节字符。使用binary格式时不允许这个选项。

NULL

指定表示一个空值的字符串。文本格式中默认是 \N（反斜线-N），CSV格式中默认 是一个未加引用的空串。在你不想区分空值和空串的情况下，即使在文本 格式中你也可能更喜欢空串。使用binary格式时不允许这 个选项。

> **注意**
>
> 在使用COPY FROM时，任何匹配 这个串的数据项将被存储为空值，因此你应该确定你使用的是和 COPY TO时相同的串。

HEADER

指定文件包含标题行，其中有每一列的名称。在输出时，第一行包含 来自表的列名。在输入时，第一行会被忽略。只有使用 CSV格式时才允许这个选项。

QUOTE

指定一个数据值被引用时使用的引用字符。默认是双引号。 这必须是一个单一的单字节字符。只有使用 CSV格式时才允许这个选项。

ESCAPE

指定应该出现在一个匹配QUOTE值的数据字符之前 的字符。默认和QUOTE值一样（这样如果引用字符 出现在数据中，它会被双写）。这必须是一个单一的单字节字符。 只有使用CSV格式时才允许这个选项。

FORCE_QUOTE

强制必须对每个指定列中的所有非NULL值使用引用。 NULL输出不会被引用。如果指定了*， 所有列的非NULL值都将被引用。只有在 COPY TO中使用CSV格式时才允许 这个选项。

FORCE_NOT_NULL

不要把指定列的值与空值串匹配。在空值串就是空串的默认情况下， 这意味着空串将被读作长度为零的字符串而不是空值（即使它们没有 被引用）。只有在COPY FROM中使用CSV格式时才允许这个选项。

FORCE_NULL

将指定列的值与空值串匹配（即使它已经被加上引号），并且在找到 匹配时将该值设置为NULL。在空值串就是空串的默认 情况下，这会把一个被引用的空串转换为 NULL。 只有在COPY FROM中使用 CSV格式时才允许这个选项。

ENCODING

指定文件被以encoding_name编码。如果省略 这个选项，将使用当前的客户端编码。详见下文的注解。

输出

在成功完成时，一个COPY命令会返回一个形为

COPY count

的命令标签。 count是被复制 的行数。

> **注意**
>
> 如果命令不是COPY ... TO STDOUT或者等效的 psql元命令\copy ... to stdout， psql将只打印这个命令标签。这是为了防止弄混 命令标签和刚刚打印的数据。

注解

COPY TO只能被用于纯粹的表，不能用于视图。 不过你可以写COPY (SELECT * FROM viewname) TO ...来拷贝一个视图的当前内容。

COPY FROM可以被用于普通表、外部表、分区表或者具有INSTEAD OF INSERT触发器的视图。

COPY只处理提到的表，它不会从子表复制 数据或者复制数据到子表中。例如 COPY table TO会显示与SELECT * FROM ONLY table相同的数据。而COPY (SELECT * FROM table) TO ... 可以被用来转储一个继承层次中的所有数据。

你必须拥有被COPY TO读取的表上的选择特权， 以及被COPY FROM插入的表上的插入特权。拥有在命令中列出的列上的特权就足够了。

如果对表启用了行级安全性，相关的SELECT策略将应用于COPY table TO语句。当前，有行级安全性的表不支持COPY FROM。不过可以使用等效的INSERT语句。

COPY命令中提到的文件会被服务器（而不是 客户端应用）直接读取或写入。因此它们必须位于数据库服务器（不是客户 端）的机器上或者是数据库服务器可以访问的。它们必须是 PostgreSQL用户（运行服务器的用户 ID）可访问的并且是可读或者可写的。类似地，用PROGRAM 指定的命令也会由服务器（不是客户端应用）直接执行，它也必须是 PostgreSQL用户可以执行的。 只允许数据库超级用户或者授予了默认角色pg_read_server_files、pg_write_server_files及pg_execute_server_program之一的用户COPY一个文件或者命令， 因为它允许读取或者写入服务器有特权访问的任何文件或者运行服务器有特权访问的程序。

不要把COPY和 psql指令 \copy 弄混。\copy会调用 COPY FROM STDIN或者COPY TO STDOUT，然后读取/存储一个 psql客户端可访问的文件中的数据。 因此，在使用\copy时，文件的可访 问性和访问权利取决于客户端而不是服务器。

我们推荐在COPY中使用的文件名总是 指定为一个绝对路径。在COPY TO的 情况下服务器会强制这一点，但是对于 COPY FROM你可以选择从一个用相对 路径指定的文件中读取。该路径将根据服务器进程（而不是客户端） 的工作目录（通常是集簇的数据目录）解释。

用PROGRAM执行一个命令可能会受到操作系统 的访问控制机制（如 SELinux）的限制。

COPY FROM将调用目标表上的任何触发器 和检查约束。但是它不会调用规则。

对于标识列，COPY FROM命令将总是写上输入数据中提供的列值，这和INSERT的选项OVERRIDING SYSTEM VALUE的行为一样。

COPY输入和输出受到 DateStyle的影响。为了确保到其他 可能使用非默认DateStyle设置的PostgreSQL安装的可移植性，在使用 COPY TO前应该把 DateStyle设置为ISO。避免转储把IntervalStyle设置为 sql_standard的数据也是一个好主意，因为负的区间值可能会 被具有不同IntervalStyle设置的服务器解释错误。

即使数据会被服务器直接从一个文件读取或者写入一个文件而不通过 客户端，输入数据也会被根据ENCODING选项或者当前 客户端编码解释，并且输出数据会被根据ENCODING或 者当前客户端编码进行编码。

COPY会在第一个错误处停止操作。这在 COPY TO的情况下不会导致问题，但是 在COPY FROM中目标表将已经收到了一 些行。这些行将不会变得可见或者可访问，但是它们仍然占

据磁盘空间。 如果在一次大型的复制操作中出现错误，这可能浪费相当可观的磁盘空间。你可能希望调用VACUUM来恢复被浪费的 空间。

FORCE_NULL和FORCE_NOT_NULL可以被同时 用在同一列上。这会导致把已被引用的空值串转换为空值并且把未引用的空值 串转换为空串。

文件格式

文本格式

在使用text格式时，读取或写入的是一个文本文件， 其中每一行就是表中的一行。一行中的列被定界字符分隔。列值 本身是由输出函数产生的或者是可被输入函数接受的属于每个属性 数据类型的字符串。在为空值的列的位置使用指定的空值串。如果 输入文件的任何行包含比预期更多或者更少的列， COPY FROM将会抛出一个错误。如 果指定了OIDS，在处理用户数据列志气啊， 会从第一列读取 OID 或者把 OID 写入第一列。

数据的结束可以表示为一个只包含反斜线-点号（\.）的 单一行。从一个文件读取时，数据结束标记并不是必要的，因为文件 结束符就已经足够用了。只有使用 3.0 客户端协议之前的客户端应用 复制数据时才需要它。

反斜线字符（\）可以被用在 COPY数据中来引用被用作行或者列定界符的 字符。特别地，如果下列字符作为一个列值的一部分出现，它们 必须被前置一个反斜线：反斜线本身、新行、回车以及 当前的定界符字符。

COPY TO会不加任何反斜线返回指定的空值串。 相反，COPY FROM会在移除反斜线之前把输入 与空值串相匹配。因此，一个空值串（例如\N）不会与实 际的数据值\N（它会被表示为\\N）搞混。

COPY FROM识别下列特殊的反斜线序列：

序列	表示
\b	退格 (ASCII 8)
\f	换页 (ASCII 12)
\n	新行 (ASCII 10)
\r	回车 (ASCII 13)
\t	制表 (ASCII 9)
\v	纵向制表 (ASCII 11)
\digits	反斜线后跟一到三个十进制位表示该数字代码对应的字符
\xdigits	反斜线加x后跟一到三个十六进制位表示该数字代码对应的字符

当前，COPY TO不会发出一个十进制或十六进制位 反斜线序列，但是它确实把上面列出的其他序列用于那些控制字符。

任何上述表格中没有提到的其他反斜线字符将被当作表示其本身。不过，要注意 增加不必要的反斜线，因为那可能意外地产生一个匹配数据结束标记（ \.）或者空值串（默认是\N）的字符串。这些字符串 将在完成任何其他反斜线处理之前被识别。

强烈建议产生COPY数据的应用把数据新行和回车分别 转换为\n和\r序列。当前可以把一个数据回车表示为 一个反斜线和回车，把一个数据新行表示为一个反斜线和新行。不过，未来的发行 可能不会接受这些表示。如果在不同的机器之间（例如从 Unix 到 Windows） 传输COPY文件，它们也很容易受到破坏。

COPY TO将用一个 Unix 风格的新行（ “\n”）终止每一行。运行在 Microsoft Windows 上的服务器则会输出回车/新行（“\r\n”），不过只对 COPY到一个服务器文件这样做。为了

做到跨平台一致， COPY TO STDOUT总是发送“\n”而 不管服务器平台是什么。COPY FROM能够处理以 新行、回车或者回车/新行结尾的行。为了减少由作为数据的未加反斜线的新行 或者回车带来的风险，如果输出中的行结束并不完全相似， COPY FROM将会抱怨。

CSV 格式

这种格式选项被用于导入和导出很多其他程序（例如电子表格）使用的逗号 分隔值（CSV）文件格式。不同于 PostgreSQL标准文本格式使用的转义 规则，它产生并且识别一般的 CSV转义机制。

每个记录中的值用DELIMITER字符分隔。如果值包含 定界符字符、QUOTE字符、NULL字符串、 一个回车或者换行字符，那么整个值会被加上QUOTE字符 作为前缀或者后缀，并且在该值内QUOTE字符或者 ESCAPE字符的任何一次出现之前放上转义字符。在输出 指定列中非NULL值时，还可以使用 FORCE_QUOTE来强制加上引用。

CSV格式没有标准方式来区分NULL值和空字符串。 PostgreSQL的COPY用引用来处理 这种区分工作。NULL被按照NULL参数字符串输出 并且不会被引用，而匹配NULL参数字符串的非NULL 值会被加上引用。例如，使用默认设置时，NULL被写作一个未 被引用的空字符串，而一个空字符串数据值会被写成带双引号（""）。 值的读取遵循类似的规则。你可以用FORCE_NOT_NULL来防止 对指定列的NULL输入比较。你还可以使用 FORCE_NULL把带引用的空值字符串数据值转换成NULL。

因为反斜线在CSV格式中不是一种特殊字符，数据结束标记 \.也可以作为一个数据值出现。为了避免任何解释误会，在 一行上作为孤项出现的\.数据值输出时会自动被引用，并且 输入时如果被引用，则不会被解释为数据结束标记。如果正在载入一个由 另一个应用创建的文件并且其中具有一个未被引用的列且可能具有 \.值，你可能需要在输入文件中引用该值。

> **注意**
>
> CSV格式中，所有字符都是有意义的。一个被空白或者其他 非 DELIMITER字符围绕的引用值将包括那些字符。在导入 来自用空白填充CSV行到固定长度的系统的数据时，这可能 会导致错误。如果出现这种情况，在导入数据到PostgreSQL之前，你可能需要预处理该 CSV文件以移除拖尾的空白。

> **注意**
>
> CSV 格式将识别并且产生带有包含嵌入的回车和换行的引用值的 CSV 文件。因此文件并不限于文本格式文件的每个表行一行的形式。

> **注意**
>
> 很多程序会产生奇怪的甚至偶尔是不合常理的 CSV 文件，因此该文件 格式更像是一种习惯而不是标准。因此你可能会碰到一些无法使用这种 机制导入的文件，并且COPY也可能产生其他程序无 法处理的文件。

二进制格式

binary格式选项导致所有数据被以二进制格式 而不是文本格式存储/读取。它比文本和CSV格式要 快一些，但是二进制格式文件在不同的机器架构和 PostgreSQL版本之间的可移 植性要差些。还有，二进制格式与数据格式非常相关。例如不能从 一个smallint列中输出二进制数据并且把它读入到一个 integer列中，虽然这样做在文本格式中是可行的。

binary人间格式由一个文件头、零个或者更多个包含 行数据的元组以及一个文件尾构成。头部和数据都以网络字节序表示。

> **注意**
>
> 7.4 之前的PostgreSQL发行 使用一种不同的二进制文件格式。

文件头

文件头由 15 字节的固定域构成，后面跟着一个变长的头部扩展区。 固定域有：

签名

11-字节的序列PGCOPY\n\377\r\n\0 — 注意 零字节是签名的一个必要的部分（该签名是为了能容易地发现文件被 无法正确处理 8 位字符编码的传输所破坏。这个签名将被行尾翻译过 滤器、删除零字节、删除高位或者奇偶修改等改变）。

标志域

32-位整数位掩码，用以表示该文件格式的重要方面。位被编号为 从 0 （LSB）到 31（MSB）。 注意这个域以网络字节序存放（最高有效位在前），所有该文件格式 中使用的整数域都是这样。16-31 位被保留用来表示严重的文件格式 问题， 读取者如果在这个范围内发现预期之外的被设置位，它应该 中止。0-15 位被保留用来表示向后兼容的格式问题，读取者应该简单 地略过这个范围内任何预期之外的被设置位。当前只定义了一个标志 位，其他位必须为零：

位 16

如果为 1，表示数据中包含 OID；如果为 0，则不包含

头部扩展区长度

32-为整数，表示头部剩余部分的以字节计的长度，不包括其本身。 当前，这个长度为零，并且其后就紧跟着第一个元组。未来对该 格式的更改可能会允许在头部中表示额外的数据。如果读取者不知 道要对头部扩展区数据做什么，可以安静地跳过它。

头部扩展区域被预期包含一个能自我解释的块的序列。 该标志域并不想告诉读取者扩展数据是什么。详细的 头部扩展内容的设计留给后来的发行去做。

这种设计允许向后兼容的头部增加（增加头部扩展块或者设置低位标志位）以及 非向后兼容的更改（设置高位标志位来表示这类更改并且在需要时向扩展区域 中增加支持数据）。

元组

每一个元组由一个表示元组中域数量的 16 位整数计数开始（当前，一个表中 的所有元组都应该具有相同的计数，但是这可能不会总是为真）。然后是元组 中的每一个域，它是一个 32 位的长度字，后面则跟随着这么多个字节的域数 据（长度字不包括其本身，并且可以是零）。作为一种特殊情况，-1 表示一个 NULL 域值。在 NULL 情况下，后面不会跟随值字节。

在域之间没有对齐填充或者任何其他额外的数据。

当前，一个二进制格式文件中的所有数据值都被假设为二进制格式（格式代码一）。 可以预见未来的扩展可能会增加一个允许独立指定各列的格式代码的头部域。

要为实际的元组数据决定合适的二进制格式，你应该参考 PostgreSQL源码，特别是用于各列 数据类型的*send和*recv函数（通常可 以在源码的src/backend/utils/adt/目录中找到这些函数）。

如果文件中包含 OID，OID 域会紧跟在域计数字之后。它是一个普通域， 不过它没有被包含在域计数中。特别地，它有一个长度字 — 这将 允许容易地处理 4 字节和 8 字节 OID 的选择，并且将允许在需要时把 OID 显示为空值。

文件尾

文件位由一个包含 -1 的 16 位整数字组成。这很容易与一个 元组的域计数字区分开。

如果一个域计数字不是 -1 也不是期望的列数，读取者应该报告错误。 这提供了一种针对某种数据不同步的额外检查。

示例

下面的例子使用竖线（|）作为域定界符把一个表复制到客户端：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中复制数据到country表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

只把名称以 'A' 开头的国家复制到一个文件中：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/
sql/a_list_countries.copy';
```

要复制到一个压缩文件中，你可以用管道把输出导到一个外部压缩程序：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

这里是一个适合于从STDIN复制到表中的数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意每一行上的空白实际是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 od -c过滤后显示的。该表具有三列， 第一列类型是char(2)，第二列类型是text， 第三列类型是integer。所有行在第三列都是空值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

兼容性

SQL 标准中没有COPY语句。

下列语法用于PostgreSQL 9.0 之前的版本， 并且仍然被支持：

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE NOT NULL column_name [, ...] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE QUOTE { column_name [, ...] | * } ] ] ]
```

注意在这种语法中，BINARY和CSV被视作独立的关键词， 而不是FORMAT选项的参数。

下列语法用于PostgreSQL 7.3 之前的版本， 并且仍然被支持：

```
COPY [ BINARY ] table_name [ WITH OIDS ]
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]

COPY [ BINARY ] table_name [ WITH OIDS ]
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

CREATE ACCESS METHOD

CREATE ACCESS METHOD — 定义一种新的访问方法

大纲

```
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
```

简介

CREATE ACCESS METHOD创建一种新的访问方法。

访问方法名称在数据库中必须唯一。

只有超级用户可以定义新的访问方法。

参数

name

要创建的访问方法的名称。

access_method_type

这个子句指定要定义的访问方法的类型。当前只支持INDEX。

handler_function

handler_function是一个之前已注册的函数的名称（可能被模式限定），该函数表示要创建的访问方法。处理器函数必须被声明为接收一个单一的internal类型的参数，并且它的返回类型取决于访问方法的类型：对于INDEX访问方法，它必须是index_am_handler。处理器函数必须实现的 C 级别 API 取决于访问方法的类型。第 61 章描述了索引访问方法的 API。

示例

用处理器函数heptree_handler创建一种索引访问方法heptree：

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

兼容性

CREATE ACCESS METHOD是一种PostgreSQL扩展。

另见

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — 定义一个新的聚集函数

大纲

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
                        ORDER BY [ argmode ] [ argname ] arg_data_type
 [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = initial_condition ]
    [ , HYPOTHETICAL ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)
```

或者旧的语法

```
CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
```

```
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)
```

描述

CREATE AGGREGATE定义一个新的聚集函数。 在发布中已经包括了一些基本的和常用的聚集函数，它们的文档请见 第 9.20 节如果要定义一个新类型或者需要 一个还没有被提供的聚集函数，那么 CREATE AGGREGATE就可以被用来提供想要 的特性。

如果给定了一个模式名（例如CREATE AGGREGATE myschema.myagg ...），那么该聚集会被创建在指定的模式中。否则它 会被创建在当前模式中。

一个聚集函数需要用它的名称和输入数据类型标识。同一个模式中的两个聚集 可以具有相同的名称，只要它们在不同的输入类型上操作即可。一个聚集的名称 和输入数据类型必须与同一模式中的每一个普通函数区分开。这种行为与普通函 数名的重载完全一样（见CREATE FUNCTION）。

一个简单的聚集函数由一个或者多个普通函数组成： 一个状态转移函数 sfunc和一个可选的最终 计算函数 ffunc。 它们被这样使用：

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL创建一个数据类型 stype的临时变量来 保持聚集的当前内部状态。对每一个输入行，聚集参数值会被计算并且状态 转移函数会被调用，它用当前状态值和新参数值计算一个新的内部状态值。 等所有行都被处理完后，最终函数会被调用一次来计算该聚集的返回值。如果 没有最终函数，则最终的状态值会被返回。

一个聚集函数可以提供一个初始条件，即一个用于内部状态值的初始值。它被 作为一个类型text的值指定并且存储在数据库中，但是它必须 是状态值数据类型的一个常量的合法外部表示。如果没有提供，则状态值从空值 开始。

如果状态转移函数被声明为“strict”，那么不能用空值输入来 调用它。如果有这种转移函数，聚集将按照下面的行为执行。带有任何空值的 行会被忽略（函数不被调用并且之前的状态值被保持）。如果初始状态值就是 空值，那么碰到第一个没有空值的行时，状态值会被替换成第一个参数值，并且 对于每一个后续的没有空值的行都会调用该转移函数。这对实现 max这样的聚集很方便。注意只有当 state_data_type 和第一个 arg_data_type相同时， 这种行为才可用。当这些类型不同时，你必须提供一个非空初始条件或者使用 一个非严格转移函数。

如果状态转移函数不是严格的，那么在碰到每个输入行时都将会调用它，并且 它必须自行处理空值输入和空状态值。这允许聚集的作者完全控制该聚集如何 处理空值。

如果最终函数被声明为“strict”，那么当最终状态值为空时将 不会调用它，而是自动地返回一个空结果（当然，这只是严格函数的普通行为）。 在任何情况下最终函数都可以返回一个空值。例如，avg的最终函数会在看到零个 输入行时返回空。

有时候把最终函数声明成不仅采用状态值还采用对应于聚集输入值的额外参数 是有用的。这样做的主要原因是，如果最终函数是多态的，那么状态值的数据 类型将不适合于用来确

定结果类型。这些额外的参数总是以 NULL 形式传递 （因此使用FINALFUNC_EXTRA选项时，最终函数不能是严格的）， 但尽管如此它们都是合法参数。例如，最终函数可以利用get_fn_expr_argtype来标识当前调用中的实际参数类型。

如第 38.11.1 节所述，一个聚集可以 选择支持移动聚集模式。这要求指定MSFUNC、MINVFUNC以及 MSTYPE参数，并且参数MSPACE、 MFINALFUNC、MFINALFUNC_EXTRA和MINITCOND是可选的。除了MINVFUNC， 这些参数的工作都和对应的不带M的简单聚集参数相似，它们 定义了包括一个逆向转移函数的聚集的一种独立实现。

在参数列表中带有ORDER BY的语法会创建一种被称为 有序集聚集的特殊聚集类型。如果指定了 HYPOTHETICAL，则会创建一个 假想集聚集。这些聚集以依赖排序的方法在排好序 的值上操作，因此指定一个输入排序顺序是调用过程的重要一环。还有，它们 可以有直接参数，这类参数只对每次聚集计算一次，而不是对 每一个输入行计算一次。假想集聚集是有序集聚集的一个子类，其中一些直接 参数要求在数量和类型上都匹配被聚集的参数列。这允许这些直接参数的值被 当做一个附加的“假想”行被加入到聚集输入行的集合中。

如第 38.11.4 节所示，一个聚集可以支持 部分聚集。这要求指定COMBINEFUNC参数。 如果state_data_type 为internal，通常也可以提供SERIALFUNC和 DESERIALFUNC参数，这样可以让并行聚集成为可能。注意， 该聚集还必须被标记为PARALLEL SAFE以启用并行聚集。

行为与MIN或MAX相似的聚集有时可以通过 直接查看一个索引而不是扫描每一个输入行来优化。如果这个聚集可以被这样 优化，请通过指定一个排序操作符来指出。基本要求是，该聚集必须得出由该操作符产生的排序顺序中的第一个元素，换句话说：

```
SELECT agg(col) FROM tab;
```

必须等价于：

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

进一步的假定是该聚集忽略空输入，并且当且仅当没有非空输入时它才会返回 一个空结果。通常，一种数据类型的<操作符是 MIN的合适的排序操作符，而>是 MAX的合适的排序操作符。注意，除非指定的操作符是一个 B-树索引操作符类的“小于”或者“大于” 策略成员，优化将永远不会产生实际效果。

要能够创建一个聚集函数，你必须具有参数类型、状态类型和返回类型上的 USAGE特权，还有在支持函数上的 EXECUTE特权。

参数

name

要创建的聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或者VARIADIC（聚集函数 不支持OUT参数）。如果忽略，默认值是IN。 只有最后一个参数能被标记为VARIADIC。

argname

一个参数的名称。当前这只用于文档的目的。如果被忽略，该参数就没有名称。

arg_data_type

这个聚集函数操作的一个输入数据类型。要创建一个零参数的聚集函数，可以 写一个*来替代参数说明的列表（这类聚集的一个例子是 count(*)）。

base_type

在CREATE AGGREGATE的旧语法中，输入数据类型是由 一个basetype参数指定而不是写在聚集名之后。注意这种语法 只允许一个输入参数。要用这种语法定义一个零参数的聚集函数，把 basetype指定为"ANY"（不是*）。 有序集聚集不能用旧语法定义。

sfunc

要为每一个输入行调用的状态转移函数名。对于一个正常的 N-参数的聚集函数，sfunc必须接收 N+1 个参数， 第一个参数的类型是state_data_type而其余的参数匹配该聚集被声明的输入数据类型。该函数必须返回一个类型为 state_data_type 的值。这个函数会采用当前的状态值以及当前的输入数据值，并且返回下一个 状态值。

对于有序集（包括假想集）聚集，状态转移函数只接收当前的状态值和聚集参数， 但无需直接参数。否则它就和其他转移函数一样了。

state_data_type

聚集的状态值的数据类型。

state_data_size

聚集的状态值的近似平均尺寸（以字节计）。如果这个参数被忽略或者为零， 将使用一个基于state_data_type的默认估计值。规划器 使用这个值来估计一个分组聚集查询所需的内存。只有估计哈希表能够放在 work_mem大小的内存中时，规划器才会对这类查询使用哈希聚集。因此，对这个参数设置大的值会阻止使用哈希聚集。

ffunc

最终函数的名称，该函数在所有输入行都被遍历之后被调用来计算聚集的结果。 对于一个常规聚集，这个函数必须只接受一个类型为state_data_type的单一参数。该聚集 的返回数据类型被定义为这个函数的返回类型。如果没有指定 ffunc，则结束状态值 被用作聚集的结果，并且返回类型为state_data_type。

对于有序集（包括假想集）聚集，最终函数不仅接收最终状态值，还会接收所 有直接参数的值。

如果指定了FINALFUNC_EXTRA，则除了最终状态值和任何直接 参数之外，最终函数还接收额外的对应于该聚集的常规（聚集）参数的 NULL 值。 这主要用于在定义了一个多态聚集时允许正确地决定聚集的结果类型。

FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项指定最终函数是否为不会修改参数的纯函数。READ_ONLY表示它不会修改；其他两个值表示它可能会更改迁移状态值。请参见注解 以获取更多详细信息。除了有序集合的聚合使用默认值READ_WRITE，其他默认值均为READ_ONLY。

combinefunc

combinefunc函数可以被 有选择地指定以允许聚集函数支持部分聚集。如果提供这个函数， combinefunc必须组合两个 state_data_type值，每一个 都包含在输入值某个子集上的聚集结果，它会产生一个新的 state_data_type来表示在 两个输入集上的聚集结果。这个函数可以被看做是一个 sfunc，和后者在一个个体 输入行上操作并且把它加到运行聚集状态上不同，这个函数是把另一个聚集状态加 到运行状态上。

combinefunc必须被声明为 有两个state_data_type参数 并且返回一个state_data_type值。这个函数可以有选择性地被标记为“strict”。在被标记的情况下， 当任何一个输入状态为空时，将不会调用该函数，而是把另一个状态当作正确的结果。

对于state_data_type为 internal的聚集函数， combinefunc不能为 strict。这种情况下， combinefunc必须确保 正确地处理空状态并且被返回的状态能被恰当地存储在聚集内存上下文中。

serialfunc

state_data_type为 internal的一个聚集函数可以参与到并行聚集中，当且仅当它具有一个 serialfunc函数，该函数 必须把聚集状态序列化成一个bytea值以传送给另一个进程。这个函数 必须有一个单一的internal类型参数并且返回类型bytea。 相应地也需要一个deserialfunc。

deserialfunc

把一个之前序列化后的聚集状态反序列化为 state_data_type。这个函数 必须有两个类型分别为bytea和internal的参数，并且产生 一个类型internal的结果（注意：第二个类型为internal的 参数是无用的，但是为了类型安全的原因还是要求有该参数）。

initial_condition

状态值的初始设置。这必须是以数据类型state_data_type能够接受的形式出现 的一个字符串常量。如果没有指定，状态值会从空值开始。

msfunc

前向状态转移函数的名称，在移动聚集模式中会为每个输入行调用这个函数。它 非常像常规的转移函数，不过它的第一个参数和结果类型是 mstate_data_type，这可能与state_data_type不同。

minvfunc

在移动聚集模式中用到的逆向状态转移函数的名称。这个函数与 msfunc具有相同的参数和结果类型，但是它被用于从当前聚集 状态中移除一个值，而不是向其中增加一个值。逆向转移函数必须具有和前向状态 转移函数相同的严格性属性。

mstate_data_type

使用移动聚集模式时，用于聚集状态值的数据类型。

mstate_data_size

使用移动聚集模式时，聚集状态值的近似平均尺寸（以字节计）。它的作用和state_data_size相同。

mffunc

使用移动聚集模式时用到的最终函数名称，当所有输入行都被遍历后会调用它来 计算聚集的结果。它的工作和ffunc一样，但是它的第一个参 数类型是mstate_data_type并且额外的空参数要通过书写 MFINALFUNC_EXTRA来指定。mffunc 或者mstate_data_type决定的聚集结果类型必须匹配由聚集 的常规实现所确定的类型。

MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项类似于FINALFUNC_MODIFY，只是它描述了移动聚集最终函数的行为。

minitial_condition

使用移动聚集模式时，状态值的初始设置。它的作用和 initial_condition一样。

sort_operator

一个MIN- 类或者MAX-类聚集的相关 排序操作符。这只是一个操作符名称（可能被模式限定）。这个操作符被 假定为具有和该聚集（必须是一个单一参数的常规聚集）相同的输入数据 类型。

PARALLEL = { SAFE | RESTRICTED | UNSAFE }

PARALLEL SAFE、PARALLEL RESTRICTED和PARALLEL UNSAFE的含义和 CREATE FUNCTION中的相同。如果一个聚集被标记为 PARALLEL UNSAFE（默认）或者 PARALLEL RESTRICTED，

将不会考虑将它并行化。注意 规划器不会参考聚集的支持函数的并行安全性标记，它只会考虑聚集本身 的这类标记。

HYPOTHETICAL

只用于有序集聚集，这个标志指定聚集参数会被根据假想集聚集的要求进行处理： 即后面的直接参数必须匹配聚集（WITHIN GROUP）参数的数据 类型。HYPOTHETICAL标志在运行时没有任何效果，它只在 命令解析期间对确定数据类型和聚集参数的排序规则有用。

CREATE AGGREGATE的参数可以用任意顺序书写， 而无需遵照以上说明的顺序。

注解

在指定支持函数名的参数中，如果需要你可以写一个模式名，例如 SFUNC = public.sum。在这里不能写参数类型 — 支持函数 的参数类型是根据其他参数决定的。

通常，PostgreSQL函数是不要修改输入值的真函数。然而，一个聚合迁移函数在聚合上下文中使用时被允许诈欺并修改已在迁移状态的参数。因为与每次创建一个迁移状态的新的拷贝相比，这样可以提供实质的性能提升。

同样，虽然人们一般不期望聚合最终函数修改它的输入值，但有时回避修改迁移态参数是不切实际的。这种行为必须使用FINALFUNC_MODIFY参数声明。READ_WRITE值表示最终函数以某种未指定的方式修改了迁移状态值。这个值防止将聚合用作窗口函数，并且还可以防止因共用相同的输入值和迁移函数的聚合调用而合并迁移状态。SHAREABLE值表示过渡函数不能在最终功能之后使用用，但多重最终函数调用可以对最终的迁移状态值执行调用。这个值阻止将聚合用作窗口函数，但允许合并过渡状态。（也就是说，此处所关注的优化不是重复地使用相同的最终函数，而是把不同的最终函数应用到相同的最终迁移状态值。只要所有最终功能都没有标记为READ_WRITE就被允许。）

如果一个聚集支持移动聚集模式，当该聚集被用于一个具有移动帧起点（即帧起点 模式不是UNBOUNDED PRECEDING）的窗口函数时，它将提升计 算效率。在概念上，当从底部进入窗口帧时前向转移函数会把输入值加到聚集的状 态上，而逆向转移函数会在从顶部离开帧时再次移除输入值。因此，当值被移除时， 它们总是按照被加入的相同顺序被移除。无论何时调用逆向转移函数，它都将因此 接收最近增加但是还未被移除的参数值。逆向转移函数可以假定在它移除最旧的行 之后至少有一行保留在当前状态中（当情况不是这样时，窗口函数机制会简单地开 始一次新的聚集，而不是使用逆向转移函数）。

用于移动聚集模式的前向转移函数不允许返回 NULL 作为新的状态值。如果逆向 转移函数返回 NULL，这表明逆向函数无法为这个特定的输入逆转状态计算，并且 该聚集计算因此必须从当前帧的开始位置“从零开始”重新被计算。在一些少见的情 况中，逆转正在计算中的状态值是不切实际的，这种习惯可以允许在此类情形中使用 移动聚集模式。

如果没有提供移动聚集实现，聚集仍然可以被用于移动帧，但是 只要帧起点移动，PostgreSQL都将会重新计算 整个聚集。注意不管聚集有没有支持移动聚集模式，PostgreSQL都能处理一个移动帧结束而无需重 新计算，这可以通过增加新值到聚集状态完成。这就是为什么使用聚合作为窗口函数需要最终函数只读的原因。人们认为最终函数不能破坏聚集的状 态值，这样即使已经为一组帧边界得到了一个聚集结果值，该聚集也能继续下去。

有序集聚集的语法允许为最后一个直接参数以及最后一个聚集（ WITHIN GROUP）参数指定VARIADIC。但是，当前的 实现限制只能以两种方式使用VARIADIC。第一种，有续集聚集只能 使用VARIADIC "any"，不能使用其他可变数组类型。第二种，如果最 后一个直接参数是VARIADIC "any"，那么只能有一个聚集参数并且它 也必须是VARIADIC "any"（在系统目录中使用的表示中，这两个参数 会被合并为一个单一的VARIADIC "any"项，因为 pg_proc无法表示具有超过一个VARIADIC参数的 函数）。如果该聚集是一个假想集聚集，匹配VARIADIC "any"参数的 直接参数就是假想参数。任何在前面的参数表示额外的直接参数，它们不被约束为需 要匹配聚集参数。

当前，有序集聚集无须支持移动聚集模式，因为它们不能被用作窗口函数。

部分（包括并行）聚集当前不被有续集聚集支持。还有，包括 DISTINCT或者ORDER BY子句的聚集调用将不会使用 部分聚集，因为在部分聚集时无法支持那些语义。

示例

见第 38.11 节

兼容性

CREATE AGGREGATE是一种 PostgreSQL的语言扩展。SQL 标准没有提供 用户定义的聚集函数。

另见

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — 定义一种新的造型

大纲

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

描述

CREATE CAST定义一种新的造型。 一种造型指定如何在两种数据类型之间执行转换。例如，

```
SELECT CAST(42 AS float8);
```

通过调用一个之前指定的函数（这种情况中是 float8(int4)）把整型常量 42 转换成类型float8（如果没有定义合适的造型， 该转换会失败）。

两种类型可以是二进制可强制，这表示该转换 可以被“免费”执行而不用调用任何函数。这要求相应的 值使用同样的内部表示。例如，类型text和 varchar在双向都是二进制可强制的。二进制可强制性并 不必是一种对称关系。例如，在当前实现中从xml到 text的造型可以被免费执行，但是反向则需要一个函数来 执行至少一次语法检查（两种在双向都二进制值兼容的类型也被称作二进 制兼容）。

通过使用WITH INOUT语法，你可以把一种造型定义 成I/O 转换造型。一种 I/O 转换造型执行时，会调用源 数据类型的输出函数，并且把结果字符串传递给目标数据类型的输入函数。在很多常见情况中，这种特性避免了为转换单独定义一个造型函数。一种 I/O 转换造型表现得和一个常规的基于函数的造型相同，只是实现不同而已。

默认情况下，只有一次显式造型请求才会调用造型， 形式是CAST(x AS typename) or x::typename。

如果造型被标记为AS ASSIGNMENT，那么在为一个目标数据 类型的列赋值时会隐式地调用它。例如，假设foo.f1是 一个类型text的列，那么如果从类型integer 到类型text的造型被标记为AS ASSIGNMENT， 则：

```
INSERT INTO foo (f1) VALUES (42);
```

将被允许，否则不会允许（我们通常使用赋值造型 来描述此类造型）。

如果造型被标记为AS IMPLICIT，那么可以在任何上下文 中隐式地调用它，无论是赋值还是在一个表达式内部（我们通常用术语 隐式造型来描述这类造型）。例如，考虑这个 查询：

```
SELECT 2 + 4.0;
```

解析器初始会把常量分别标记为类型integer和 numeric。在系统目录中没有integer + numeric操作符，但是有一个 numeric + numeric操作符。 因此，如果有一种可用的从integer到 numeric的造型且被标记为AS IMPLICIT — 实际上确实有 — 该查询将会成功。解析器将应用该隐式造型 并且解决该查询，就好像它被写成：

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

现在，系统目录也提供一种从numeric到integer 的造型。如果这种造型被标记为AS IMPLICIT — 实际上并没有 — 那么解析器将面临选择：是用前面介绍的过程， 还是把numeric常量造型成integer并且应用 integer + integer操作符。由于 缺少哪种选择更好的知识，解析器会放弃并且说明查询有歧义。我们能 告诉解析器把一个混合了numeric和integer的 表达式解析成numeric更好的方法就是只让这两种造型中的 一个是隐式的，没有有关于此的内建知识。

对标记造型为隐式持保守态度是明智的。过多的隐式造型路径可能导致 PostgreSQL以令人吃惊的方式解 释命令，或者由于有多种可能解释而根本无法解析命令。一种好的经验 是让一种造型只对于同一种一般类型分类中的类型间的信息保持转换隐式 可调用。例如，从int2到int4的造型 可以被合理地标记为隐式，但是从float8到 int4的造型可能应该只能在赋值时使用。跨类型分类 的造型（如text到int4）最好只被用于显式使用。

> **注意**
>
> 有时为了可用性或者标准兼容的原因，有必要提供在一个类型集合之间 的多种隐式造型，这会导致上述不可避免的歧义。解析器还有一招基于 类型分类和优先类型的后手，它能帮助 提供这类情况下预期的行为。详见 CREATE TYPE。

要创建一种造型，你必须拥有源数据类型和目标数据类型并且具有在其他类型上 的USAGE特权。要创建一种二进制可强制造型，你必须是一 个超级用户（这种限制是因为错误的二进制可强制造型转换很容易让服务器崩溃）。

参数

source_type

该造型的源数据类型的名称。

target_type

该造型的目标数据类型的名称。

function_name[(argument_type [, ...])]

被用于执行该造型的函数。函数名称可以用模式限定。如果没有被限定， 将在模式搜索路径中查找该函数。函数的结果数据类型必须是该造型的 目标数据类型。它的参数讨论如下。 如果没有指定参数列表，则该函数名称在其模式中必须是唯一的。

WITHOUT FUNCTION

指示源类型可以二进制强制到目标类型，因此执行该造型不需要函数。

WITH INOUT

指示该造型是一种 I/O 转换造型，执行需要调用源数据类型的输出函数， 并且把结果字符串传递给目标数据类型的输入函数。

AS ASSIGNMENT

指示该造型可以在赋值的情况下被隐式调用。

AS IMPLICIT

指示该造型可以在任何上下文中被隐式调用。

造型实现函数可以具有 1 到 3 个参数。第一个参数类型必须等于源类型或者 能从源类型二进制强制得到。第二个参数（如果存在）必须是类型 integer，它接收与目标类型相关联的类型修饰符，如果没有类型 修饰符，它会收到-1。第三个参数（如果存在）必须是类型boolean，如果该造型是一种显式造型，它会收到 true，否则会收到false（奇怪地是，SQL标准在 某些情况中对显式和隐式造型要求不同的行为。这个参数被提供给必须实现这 类造型的函数。不推荐在设计自己的数据类型时用它）。

一个造型函数的返回类型必须等于目标类型或者能二进制强制到目标类型。

通常，强制转换必须具有不同的源和目标数据类型。但是，如果它有一个带有多个参数的强制转换实现函数，则可以声明具有相同源类型和目标类型的造型。它用于表示系统目录中特定类型的长度强制函数。命名函数用于将类型的值强制转为其第二个参数提供的类型修饰符的值。

当强制转换具有不同的源类型和目标类型，并且一个函数使用多个参数时，它支持从一种类型转换为另一种类型，并在单个步骤中应用长度强制。如果没有这样的条目，强制转换为使用类型修饰符的类型将涉及两个强制转换步骤，一个是在数据类型之间进行转换，另一个是应用修饰符。

向域类型强制转换或从域类型强制转换当前无效。向域或从域强制转换使用与其基础类型关联的造型。

注解

使用DROP CAST移除用户定义的造型。

记住如果你想要能够双向转换类型，你需要在两个方向上都 显式声明造型。

通常没有必要创建用户定义类型和标准字符串类型（text、 varchar和char(n)， 以及被定义在字符串分类中的用户定义类型）之间的造型。 PostgreSQL会为它们提供自动的 I/O 转换造型。 到字符串类型的自动造型被当做赋值造型，而字符串类型作为源的自动 造型只能是显式的。通过声明你自己的造型来替换自动造型可以覆盖这 种行为，但是这样做的唯一原因是你想让该转换比标准的设置更容易被 调用。另一种可能的原因是你想让该转换的行为与该类型的 I/O 函数不 同，但这种原因足够令人感到意外，你应该考虑再三它是不是个好主意 （确实有少量内建类型对转换具有不同的行为，绝大部分是因为 SQL 标准的要求）。

虽然不必要，推荐你继续遵循这种在目标数据类型后面命名造型 实现函数的习惯。很多用户习惯于能够使用一种函数风格的记法来造型 数据类型，即typename(x)。 这种记法正好是对造型实现函数的调用，这里它没有被作为造型特殊对待。 如果你的转换函数没有被指定支持这种习惯，那么你的用户会觉得意外。 由于PostgreSQL允许用不同的参数类型重载同一个 函数名，因此存在多个从不同类型到同一目标类型的同名转换函数并不困难。

> **注意**
>
> 实际上前一段过于简化了：有两种情况中一个函数调用结构在没有被匹配到一个实际函数时将被当作一次造型请求。如果函数调用 name(x)没有正好匹配任何现有函数， 但name是一种数据类型的名称并且 pg_cast提供了一种从x的类型到这种 类型的二进制可强制造型，那么该调用将被翻译为一次二进制可强制造型。 通过这种例外，二进制可强制造型能够以函数语法调用，即便没

有该函数。 同样的，如果没有pg_cast项，但是该造型是要造型到一种 字符串类型或者是要从一种字符串类型造型，调用将被翻译成一次 I/O 转换 造型。这种例外允许以函数语法调用 I/O 转换造型。

注意

还有一种例外中的例外：从组合类型到字符串类型的 I/O 转换造型不能使用函数语法调用，而必须被写成显式造型语法（CAST或者 ::记号）。增加这种例外是因为在引入了自动提供的 I/O 转换 造型之后，在想要引用一个函数或者列时太容易意外地调用这种造型。

示例

要使用函数int4(bigint)创建一种从类型 bigint到类型int4的赋值造型：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（在系统中这种造型已经被预定义）。

兼容性

CREATE CAST命令符合 SQL标准，不过 SQL 没有为二进制可强制 类型或者实现函数的额外参数做好准备。 AS IMPLICIT也是一种 PostgreSQL扩展。

另见

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — 定义一种新排序规则

大纲

```
CREATE COLLATION [ IF NOT EXISTS ] name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype, ]
    [ PROVIDER = provider, ]
    [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

描述

CREATE COLLATION使用指定的操作系统区域 设置或者复制一个现有的排序规则来定义新的排序规则。

要创建一种排序规则，你必须拥有目标模式上的 CREATE特权。

参数

IF NOT EXISTS

如果已经存在了同名的排序规则，则不要抛出错误。在这种情况下发出一个通知。 请注意，不保证已经存在的排序规则与要创建的这个类似。

name

排序规则的名字，可以被模式限定。如果没有用模式限定，该排序规则 会被定义在当前模式中。排序规则名称在其所处的模式中必须唯一（系统 目录可以为其他编码包含具有相同名称的排序规则，但数据库编码不匹配 时它们会被忽略）。

locale

这是一种一次设置LC_COLLATE 和LC_CTYPE的快捷方式。如果你指定它，你就 不能指定那两个参数。

lc_collate

为LC_COLLATE区域分类使用指定的操作系统 区域。

lc_ctype

为LC_CTYPE区域分类使用指定的操作系统 区域。

provider

指定用于与此排序规则相关的区域服务的提供程序。可能的值是： icu、 libc。 默认是libc。 可用的选择取决于操作系统和构建选项。

version

指定使用该排序规则存储的版本字符串。通常忽略该选项， 这会导致版本从操作系统提供的排序规则实际版本中计算出来。 此选项旨在供pg_upgrade用于复制现有安装中的版本。

又见ALTER COLLATION获取如何处理排序规则版本错误匹配。

existing_collation

要复制的一种现有的排序规则的名称。新的排序规则将和现有的具有 同样的属性，但是它是一个独立的对象。

注解

使用DROP COLLATION可移除用户定义的排序规则。

关于如何创建排序规则的更多信息可见第 23.2.2.3 节

使用libc排序规则提供程序时，语言环境必须适用于当前的数据库编码。 有关精确的规则，请参见CREATE DATABASE。

示例

从操作系统区域fr_FR.utf8创建一种排序规则（假定 当前数据库编码是UTF8）：

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

使用German phone book排序顺序使用ICU提供程序创建排序规则：

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');
```

从一个现有的排序规则创建一个新的排序规则：

```
CREATE COLLATION german FROM "de_DE";
```

能在应用中使用与操作系统无关的排序规则名称就很方便了。

兼容性

在 SQL 标准中有一个CREATE COLLATION 语句，但是它被限制为只能复制一个现有的排序规则。创建新排序规则的 语法是一种PostgreSQL扩展。

另见

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — 定义一种新的编码转换

大纲

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM function_name
```

描述

CREATE CONVERSION定义一种字符集编码间 新的转换。还有，被标记为DEFAULT的转换将被自动地用于客户端和服务器之间的编码转换。为了这个目的，必须定义两个 转换（从编码 A 到 B 以及从编码 B 到 A）。

要创建一个转换，你必须拥有该函数上的EXECUTE特权 以及目标模式上的CREATE特权。

参数

DEFAULT

DEFAULT子句表示这个转换是从源编码到目标编码的默认 转换。在一个模式中对于每一个编码对，只应该有一个默认转换。

name

转换的名称，可以被模式限定。如果没有被模式限定，该转换被定义在 当前模式中。在一个模式中，转换名称必须唯一。

source_encoding

源编码名称。

dest_encoding

目标编码名称。

function_name

被用来执行转换的函数。函数名可以被模式限定。如果没有，将在路径 中查找该函数。

该函数必须具有一下的特征：

```
conv_proc(
    integer,  -- 源编码 ID
    integer,  -- 目标编码 ID
    cstring,  -- 源字符串（空值终止的 C 字符串）
    internal, -- 目标（用一个空值终止的 C 字符串填充）
    integer   -- 源字符串长度
) RETURNS void;
```

注解

使用DROP CONVERSION可以移除用户定义的转换。

创建转换所要求的特权可能在未来的发行中被更改。

示例

使用myfunc创建一个从编码UTF8到 LATIN1的转换：

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

兼容性

CREATE CONVERSION是一种 PostgreSQL扩展。在 SQL 标准中 没有CREATE CONVERSION语句，但是有 一个目的和语法都类似的 CREATE TRANSLATION语句。

另见

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] user_name ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ] ]
```

描述

CREATE DATABASE创建一个新的PostgreSQL数据库。

要创建一个数据库，你必须是一个超级用户或者具有特殊的CREATEDB特权。见CREATE ROLE。

默认情况下，新数据库将通过克隆标准系统数据库template1被创建。可以通过写TEMPLATE name指定一个不同的模板。特别地，通过写TEMPLATE template0你可以创建一个干净的数据库，它将只包含你的PostgreSQL版本所预定义的标准对象。如果你希望避免拷贝任何可能被加入到template1中的本地安装对象，这将有所帮助。

参数

name

要创建的数据库名。

user_name

将拥有新数据库的用户的角色名，或者用DEFAULT来使用默认值（即，执行该命令的用户）。要创建一个被另一个角色拥有的数据库，你必须是该角色的一个直接或间接成员，或者是一个超级用户。

template

要从其创建新数据库的模板名称，或者用DEFAULT来使用默认模板（template1）。

encoding

要在新数据库中使用的字符集编码。指定一个字符串常量（例如'SQL_ASCII'），或者一个整数编码编号，或者DEFAULT来使用默认的编码（即，模板数据库的编码）。PostgreSQL服务器所支持的字符集在第 23.3.1 节描述。附加限制见下文。

lc_collate

要在新数据库中使用的排序规则顺序（LC_COLLATE）。这会影响应用到字符串的排序顺序，例如在带 ORDER BY 的查询中，以及文本列上索引所使用的顺序。默认是使用模板数据库的排序规则顺序。附加限制见下文。

lc_ctype

要在新数据库中使用的字符分类（LC_CTYPE）。这会影响字符的类别，如小写、大写和数字。默认是使用模板数据库的字符分类。附加限制见下文。

tablespace_name

将与新数据库相关联的表空间名称，或者DEFAULT来使用模板数据库的表空间。这个表空间将是在这个数据库中创建的对象的表空间。详见CREATE TABLESPACE。

allowconn

如果为假，则没有人能连接到这个数据库。默认为真，表示允许连接（除了 被其他机制约束以外，例如GRANT/REVOKE CONNECT）。

connlimit

这个数据库允许多少并发连接。-1 （默认值）表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从 这个数据库克隆。如果为假（默认），则只有超级用户或者该数据库的拥有者 可以克隆它。

可选的参数可以被写成任何顺序，不用按照上面说明的顺序。

注解

CREATE DATABASE不能在一个事务块内被执行。

带有一行“不能初始化数据库目录”的错误大部分与在数据目录上权限不足、磁盘满或其他文件系统问题有关。

使用DROP DATABASE移除一个数据库。

程序createdb是这个命令的一个包装器程序，为了使用方便而提供。

不会从模板数据库中复制数据库层面的配置参数（通过ALTER DATABASE设置）。

尽管可以通过指定一个数据库作为模板来从其中而不是template1复制，这（还）不是“COPY DATABASE”功能的一般目的。主要的限制是在模板数据库被拷贝期间其他会话不能连接到它。如果CREATE DATABASE启动时还存在任何其他连接，它将失败。否则，到模板数据库的新连接将被挡在外面直到CREATE DATABASE完成。详见第 22.3 节

为新数据库指定的字符集编码必须与选定的区域设置（LC_COLLATE和LC_CTYPE）相兼容。如果区域是C（或者等效的POSIX），那么所有编码都被允许，但是对于其他区域设置只有一种编码能正确工作（不过，在 Windows 上 UTF-8 编码能够与任何区域一起使用）。CREATE DATABASE将允许超级用户指定SQL_ASCII编码而不管区域设置，但是这种选择已被废弃并且可能在数据与数据库中存储的区域不是编码兼容时让字符串函数行为失当。

编码和区域设置必须匹配模板数据的编码和区域，除非template0被用作模板。这是因为其他数据库可能包含不匹配指定编码的数据，或者可能包含排序顺序受LC_COLLATE和LC_CTYPE影响的索引。拷贝这种数据将导致一个由于该新设置损坏的数据库。不过，template0是不会含有任何可能被影响的数据或索引的。

CONNECTION LIMIT选项大概是唯一会被强制的，如果两个新会话在大约同一时间开始并且那时该数据库只剩有一个连接“槽”，可能两者都会失败。还有，该限制对超级用户或后台工作进程无效。

例子

要创建一个新数据库：

```
CREATE DATABASE lusiadas;
```

要在一个默认表空间salesspace中创建一个被用户salesapp拥有的新数据库sales：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

要用不同的语言环境创建数据库music：

```
CREATE DATABASE music
    LC_COLLATE 'sv_SE.utf8' LC_CTYPE 'sv_SE.utf8'
    TEMPLATE template0;
```

在这个例子中，如果指定的语言环境与template1中的语言环境不同， 则需要TEMPLATE template0子句。（如果不是，则明确指定区域设置是多余的。）

要用不同的语言环境和不同的字符集编码创建数据库music2：

```
CREATE DATABASE music2
    LC_COLLATE 'sv_SE.iso885915' LC_CTYPE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

指定的区域设置和编码设置必须匹配，否则会报告错误。

请注意，区域名称是特定于操作系统的， 因此上述命令可能无法在任何地方以相同的方式工作。

兼容性

在 SQL 标准中没有CREATE DATABASE语句。数据库等效于目录，而目录的创建由实现定义。

参见

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — 定义一个新的域

大纲

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

其中 constraint 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

描述

CREATE DOMAIN创建一个新的域。域 本质上是一种带有可选约束（在允许的值集合上的限制）的数据类型。 定义一个域的用户将成为它的拥有者。

如果给定一个模式名（例如CREATE DOMAIN myschema.mydomain ...），那么域将被创建在该指定的模式中。 否则它会被创建在当前模式中。域的名称在其模式中的类型和域之间 必须保持唯一。

域主要被用于把字段上的常用约束抽象到一个单一的位置以便维护。例如， 几个表可能都包含电子邮件地址列，而且都要求相同的 CHECK 约束来验证 地址的语法。可以为此定义一个域，而不是在每个表上都单独设置一个约束。

要创建一个域，你必须在其底层类型上拥有USAGE特权。

参数

name

要被创建的域的名称（可以被模式限定）。

data_type

域的底层数据类型。可以包括数组指示符。

collation

用于该域的可选的排序规则。如果没有指定排序规则，将使用底层 数据类型的默认排序规则。如果指定了COLLATE， 底层类型必须是可排序的。

DEFAULT expression

DEFAULT子句为该域数据类型的列指定一个默认值。 该值是任何没有变量的表达式（但不允许子查询）。默认值表达式 的数据类型必须匹配域的数据类型。如果没有指定默认值，那么 默认值就是空值。

默认值表达式将被用在任何没有指定列值的插入操作中。如果为一个 特定列定义了默认值，它会覆盖与域相关的默认值。继而，域默认值 会覆盖任何与底层数据类型相关的默认值。

CONSTRAINT constraint_name

一个约束的名称（可选）。如果没有指定，系统会生成一个名称。

NOT NULL

这个域的值通常不能为空值（但是看看下面的注释）。

NULL

这个域的值允许为空值。这是默认值。

这个子句只是为了与非标准 SQL 数据库相兼容而设计。在新的 应用中不鼓励使用它。

CHECK (expression)

CHECK子句指定该域的值必须满足的完整性 约束或者测试。每一个约束必须是一个产生布尔结果的表达式。 它应该使用关键词VALUE来引用要被测试的值。计算为 TRUE 或者 UNKNOWN 的表达式成功。如果该表达式产生一个 FALSE 结果，会报告一个错误并且该值不允许被转换成该域类型。

当前，CHECK表达式不能包含子查询，也不能 引用除VALUE之外的其他变量。

当一个域有多个CHECK约束，将按照其名字的 字母顺序测试它们（版本 9.5 之前的 PostgreSQL 不遵循任何用于CHECK约束的特定触发顺序）。

注解

在把一个值转换成域类型时会检查域约束（特别是NOT NULL）。 即使有一个这样的约束，有可能一个名义上属于该域类型的列也会被读成空值。 例如，如果在一次外连接查询中，属于该域的列出现在外连接的空值端。下面 是一个更精细的例子：

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

空的标量子-SELECT 将产生一个空值，它被认为是该域类型的值，因此不会 在其上应用任何进一步的约束检查，并且插入将会成功。

要避免这类问题很难，因为 SQL 的一般假设是空值也是每一种数据类型的合法值。 因此，最好的方法是设计一个允许空值的域约束，然后根据需要在该域类型的列上 应用列的NOT NULL约束。

示例

这个例子创建us_postal_code数据类型并且把它用在 一个表定义中。一个正则表达式测试被用来验证值是否看起来像一个 合法的 US 邮政编码：

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
```

```
  postal us_postal_code NOT NULL
);
```

兼容性

命令CREATE DOMAIN符合 SQL 标准。

另见

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — 定义一个新的事件触发器

大纲

```
CREATE EVENT TRIGGER name
    ON event
    [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
    EXECUTE { FUNCTION | PROCEDURE } function_name()
```

描述

CREATE EVENT TRIGGER创建一个新的事件触发器。 只要指定的事件发生并且与该触发器相关的WHEN条件（如果有）被 满足，该触发器的函数将被执行。关于事件触发器的一般性介绍可见 第 40 章创建事件触发器的用户会成为它的拥有者。

参数

name

给新触发器的名称。在该数据库中这个名称必须唯一。

event

会触发对给定函数调用的事件名称。更多事件名称的信息请见 第 40.1 节

filter_variable

用来过滤事件的变量名称。这可以用来限制触发器只为它支持的那一部分 情况引发。当前唯一支持的 filter_variable 是TAG。

filter_value

与该触发器要为其引发的 filter_variable相关联 的一个值列表。对于TAG，这表示一个命令标签列表（例如 'DROP FUNCTION'）。

function_name

一个用户提供的函数，它被声明为没有参数并且返回类型 event_trigger。

在CREATE EVENT TRIGGER的语法中，关键字CREATE EVENT TRIGGER和PROCEDURE是等效的，但是被引用的函数在任何情况下都必须是函数，而不是过程。 此处关键字PROCEDURE的使用是历史性的，已弃用。

注解

只有超级用户能创建事件触发器。

在单用户模式（见postgres）中事件触发器被禁用。 如果一个错误的事件触发器禁用了数据库让你甚至无法删除它，可以重启到 单用户模式，这样你就能删除它。

示例

禁止执行任何DDL命令：

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

兼容性

在 SQL 标准中没有 CREATE EVENT TRIGGER语句。

另见

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — 安装一个扩展

大纲

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ FROM old_version ]
             [ CASCADE ]
```

描述

CREATE EXTENSION把一个新的扩展载入到 当前数据库中。不能有同名扩展已经被载入。

载入一个扩展本质上是运行该扩展的脚本文件。该脚本通常将创建新的 SQL对象，例如函数、数据类型、操作符以及索引支持 方法。CREATE EXTENSION会额外地记录 所有被创建对象的标识，这样发出 DROP EXTENSION时可以删除它们。

载入一个扩展要求创建其组件对象所要求的特权。对于大部分扩展这意味 这需要超级用户或者数据库拥有者的特权。为了以后特权检察的目的，运行 CREATE EXTENSION的用户会成为该扩展的 拥有者以及由该扩展的脚本创建的任何对象的拥有者。

参数

IF NOT EXISTS

已有同名扩展存在时不要抛出错误。这种情况下会发出一个提示。 注意，不保证现有的扩展与将要从当前可用的脚本文件创建的脚本 有任何相似。

extension_name

要安装的扩展的名称。PostgreSQL 将使用文件 SHAREDIR/extension/extension_name.control 中的指令来创建该扩展。

schema_name

假定该扩展允许其内容被重定位，这是要在其中安装该扩展的对象的 模式名称。被提到的模式必须已经存在。如果没有指定并且该扩展的 控制文件也没有指定一个模式，将使用当前的默认对象创建模式。

如果该扩展在其控制文件中指定了一个schema参数， 那么不能用SCHEMA子句覆盖该模式。通常，如果 给出了一个SCHEMA子句并且它与扩展的 schema参数冲突，则会发生错误。不过，如果也给 出了CASCADE子句，则schema冲突时会忽略 schema_name。 给定的schema_name 将被用来安装任何需要的并且没有在其控制文件中指定 schema的扩展。

记住扩展本身被认为不在任何模式中：扩展具有无限定的名称，并且 要在整个数据库范围内唯一。但是属于扩展的对象可以在模式中。

version

要安装的扩展的版本。这可以写成一个标识符或者一个字符串。 默认版本在该扩展的控制文件中指定。

old_version

当且仅当尝试要安装一个“老式” 的模块（它只是一组没有被打包成扩展的对象的集合）时， 才必须指定 FROM old_version。 这个选项导致CREATE EXTENSION运行 另一个安装脚本把现有的对象吸收到该扩展中，而不是创建 新对象。当心SCHEMA指定的是包含已经存在 对象的模式。

用于old_version的值由扩展的作者决定， 且如果有多于一种版本的老式模块可以被升级到扩展，该值还可能变化。 对于 9.1 之前的PostgreSQL提供的 标准附加模块，在升级模块到扩展风格时要为 old_version使用 unpackaged。

CASCADE

自动安装这个扩展所依赖的任何还未安装的扩展。它们的依赖也会同样 被自动安装。如果给出SCHEMA子句，它会应用于这种方式 下安装的所有扩展。这个语句的其他选项不会被应用于自动安装的扩展。 特别地，这些自动安装的扩展的默认版本将被选中。

注解

在使用CREATE EXTENSION载入扩展到数据库中之前， 必须先安装好该扩展的支持文件。关于安装 PostgreSQL提供的扩展的信息可以在 额外提供的模块中找到。

当前可以用于载入的扩展可以在系统视图 pg_available_extensions 或者 pg_available_extension_versions 中看到。

更多有关编写新扩展的内容请见第 38.16 节

示例

安装hstore扩展到当前数据库中：

```
CREATE EXTENSION hstore;
```

升级一个 9.1 之前的hstore安装成为扩展：

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

要小心地指定安装现有hstore对象的模式。

兼容性

CREATE EXTENSION是一种 PostgreSQL扩展。

另见

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — 定义一个新的外部数据包装器

大纲

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

CREATE FOREIGN DATA WRAPPER创建一个 新的外部数据包装器。定义外部数据包装器的用户将成为它的拥有者。

在数据库内外部数据包装器名称必须唯一。

只有超级用户能够创建外部数据包装器。

参数

name

要创建的外部数据包装器的名称。

HANDLER handler_function

handler_function是一个以前注册 好的函数的名称，它将被调用来为外部表检索执行函数。处理器函数必 须不能有参数，并且它的返回类型必须是fdw_handler。

可以创建一个没有处理器函数的外部数据包装器，但是使用这个包装 器的外部表只能被声明，但不能被访问。

VALIDATOR validator_function

validator_function 是一个之前已注册的函数的名称，它将被调用来检查给予该外部数据包装器 的选项，还有用于外部服务器、用户映射以及使用 该外部数据包装器的外部表的选项。如果没有验证器函数或者指定了 NO VALIDATOR，那么在创建时不会检查选项（ 外部数据包装器可能会在运行时忽略或者拒绝无效的选项说明，这取决于 实现）。验证器函数必须接受两个参数：一个类型是text[]， 它将包含存储在系统目录中的选项数组，另一个是类型 oid，它将是包含该选项的系统目录的 OID。返回类型 会被忽略，该函数应该使用ereport(ERROR) 函数报告无效选项。

OPTIONS (option 'value' [, ...])

这个子句为新的外部数据包装器指定选项。允许的选项名称和值与每一个 外部数据包装器有关，并且它们会被该外部数据包装器的验证器函数验证。 选项名称必须唯一。

注解

PostgreSQL的外部数据功能仍在积极的开发中。 查询的优化还很原始（也是剩下工作最多的部分）。因此，未来还有很 可观的性能提升空间。

示例

创建一个无用的外部数据包装器dummy：

```
CREATE FOREIGN DATA WRAPPER dummy;
```

用处理器函数file_fdw_handler创建一个外部数据包装器 file：

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

用一些选项创建一个外部数据包装器mywrapper：

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

兼容性

CREATE FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)，不过HANDLER和VALIDATOR子句是扩展，并且标准子句 LIBRARY和LANGUAGE还没有 在PostgreSQL中被实现。

不过要注意，整体上来说 SQL/MED 功能还没有符合。

另见

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — 定义一个新的外部表

大纲

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ]
 [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
  PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] partition_bound_spec
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]
```

其中 column_constraint 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr }
```

而 table_constraint 是：

```
[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

描述

CREATE FOREIGN TABLE在当前数据库中创建 一个新的外部表。该表将由发出这个命令的用户所拥有。

如果给定了一个模式名称（例如CREATE FOREIGN TABLE myschema.mytable ...），那么该表会被创建在指定的模式中。 否则它会被创建在当前模式中。该外部表的名称必须与同一个模式中 的任何其他外部表、表、序列、索引、视图或者物化视图区分开来。

CREATE FOREIGN TABLE还将自动创建 一个数据类型来表示该外部表行相应的组合类型。因此，外部表不能和 同一个模式中任何现有的数据类型同名。

如果指定了PARTITION OF子句， 则该表被创建为具有指定边界的parent_table的分区。

要创建一个外部表，你必须具有该外部服务器上的USAGE 特权，以及该表中用到的所有列类型上的USAGE特权。

参数

IF NOT EXISTS

已经存在同名关系时不要抛出错误。这种情况下会发出一个提示。注意， 并不保证已经存在的关系与将要创建的那一个相似。

table_name

要创建的表的名称（可以被模式限定）。

column_name

要在新表中创建的列名。

data_type

该列的数据类型。可以包括数组指示符。更多 PostgreSQL支持的数据类型可见第 8 章

COLLATE collation

COLLATE子句为该列（必须是一个可排序的数据类型） 赋予一个排序规则。如果没有指定，则会使用该列的数据类型的默认 排序规则。

INHERITS (parent_table [, ...])

可选的INHERITS子句指定了一个表的列表，新的外部表 会自动从中继承所有列。父表可以是普通表或者外部表。详见 CREATE TABLE的类似形式。

CONSTRAINT constraint_name

一个可选的用于列或者表约束的名字。如果该约束被违背，这个约束名字会 出现在错误消息中，这样col must be positive这种约束名就 可以被用来与客户端应用交流有用的约束信息（指定包含空格的约束名需要 使用双引号）。如果没有指定约束名，系统会自动生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列可以包含空值，这是默认值。

提供这个子句只是为了兼容非标准的 SQL 数据库。在新的应用中 不鼓励使用它。

CHECK (expression) [NO INHERIT]

CHECK子句指定一个产生布尔结果的表达式，该外部表 中的每一行都应该满足该表达式。也就是说，对于该外部表中所有的行， 这个表达式应该产生 TRUE 或者 UNKNOWN 而不能产生 FALSE。被 作为列约束指定的检查约束应该只引用该列的值，而出现在表约束中的 表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能 引用除当前行的列之外的其他变量。可以引用系统列 tableoid，但是不能引用其他系统列。

被标记为NO INHERIT的约束将不会传播到子表上。

DEFAULT default_expr

DEFAULT子句为包含它的列定义赋予一个默认数据值。该 值是任意不包含变量的表达式（不允许子查询和对当前表中其他列的交叉 引用）。默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有指定列值的插入操作中。如果一列没有 默认值，则默认值为空值。

server_name

要用于该外部表的一个现有外部服务器的名称。有关定义一个服务器 的细节可以参考CREATE SERVER。

OPTIONS (option 'value' [, ...])

要与新外部表或者它的一个列相关联的选项。被允许的选项名称和值是与 每一个外部数据包装器相关的，并且它们会被该外部数据包装器的验证器 函数验证。不允许重复的选项名称（不过一个表选项和一个列选项重名是 可以的）。

注解

PostgreSQL核心系统不会强制外部表上的约束（ 例如CHECK或NOT NULL子句），大部分外部 数据包装器也不会尝试强制它们。也就是说，这类约束会被简单地认为保持 为真。这种强制其实没什么意义，因为它只适用于通过该外部表插入或者更 新的行，而对通过其他方式修改的行（例如直接在远程服务器上修改）没有 作用。附着在外部表上的约束应该表示由外部服务器强制的一个约束。

有些特殊目的的外部数据包装器可能是它们所访问的数据的唯一一种访问 机制，在那种情况下让外部数据包装器自己来执行约束强制可能是合适的。 但是不应该假设包装器会这样做，除非它的文档说它会。

尽管PostgreSQL不会尝试强制外部表上的约束， 但它确实假定它们对于查询优化的目的是正确的。如果在外部表中有不 满足约束的行可见，在该表上的查询可能会产生不正确的回答。确保 约束定义符合实际是用户的责任。

示例

创建外部表films，通过服务器film_server访问它：

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

创建外部表measurement_y2016m07，通过服务器 server_07访问它，作为范围分区表measurement 的分区：

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

兼容性

CREATE FOREIGN TABLE命令大部分符合 SQL标准。不过，与 CREATE TABLE 很相似，它允许NULL约束以及零列外部表。能够指定列默认值 也是一种PostgreSQL扩展。 PostgreSQL定义的表继承形式是非标准的。

另见

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

描述

CREATE FUNCTION定义一个新函数。CREATE OR REPLACE FUNCTION将创建一个新函数或者替换一个现有的函数。要定义一个函数，用户必须具有该语言上的USAGE特权。

如果包括了一个模式名，那么该函数会被创建在指定的模式中。否则，它会被创建在当前模式中。新函数的名称不能匹配同一个模式中具有相同输入参数类型的任何现有函数或过程。不过，不同参数类型的函数和过程能够共享一个名字（这被称作重载）。

要替换一个现有函数的当前定义，可以使用CREATE OR REPLACE FUNCTION。但不能用这种方式更改函数的名称或者参数类型（如果尝试这样做，实际上就会创建一个新的不同的函数）。还有，CREATE OR REPLACE FUNCTION将不会让你更改一个现有函数的返回类型。要这样做，你必须先删除再重建该函数（在使用OUT参数时，这意味着除了删除函数之外无法更改任何OUT参数的类型）。

当CREATE OR REPLACE FUNCTION被用来替换一个现有的函数，该函数的拥有权和权限不会改变。所有其他的函数属性会按照该命令中所指定的或者隐含的来赋值。必须拥有（包括成为拥有角色的成员）该函数才能替换它。

如果你删除并且重建一个函数，新函数将和旧的不一样，你将必须删掉引用旧函数的现有规则、视图、触发器等。使用CREATE OR REPLACE FUNCTION更改一个函数定义不会破坏引用该函数的对象。还有，ALTER FUNCTION可以被用来更改一个现有函数的大部分辅助属性。

创建该函数的用户将成为该函数的拥有者。

要创建一个函数，你必须拥有参数类型和返回类型上的USAGE特权。

参数

name

要创建的函数的名称（可以被模式限定）。

argmode

一个参数的模式：IN、OUT、INOUT或者VARIADIC。如果省略，默认为IN。只有OUT参数能跟在一个VARIADIC参数后面。还有，OUT和INOUT参数不能和RETURNS TABLE符号一起使用。

argname

一个参数的名称。一些语言（包括 SQL 和 PL/pgSQL）让你在函数体中使用该名称。对于其他语言，一个输入参数的名字只是额外的文字（就该函数本身所关心的来说）。但是你可以在调用一个函数时使用输入参数名来提高可读性（见第 4.3 节。在任何情况下，输出参数的名称是有意义的，因为它定义了结果行类型中的列名（如果忽略一个输出参数的名称，系统将选择一个默认的列名）。

argtype

该函数参数（如果有）的数据类型（可以是模式限定的）。参数类型可以是基本类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据实现语言，也可以允许指定cstring之类的“伪类型”。伪类型表示实际参数类型没有被完整指定或者不属于普通 SQL 数据类型集合。

可以写table_name.column_name%TYPE来引用一列的类型。使用这种特性有时可以帮助创建一个不受表定义更改影响的函数。

default_expr

如果参数没有被指定值时要用作默认值的表达式。该表达式必须能被强制为该参数的参数类型。只有输入（包括INOUT）参数可以具有默认值。所有跟随在一个具有默认值的参数之后的输入参数也必须有默认值。

rettype

返回数据类型（可能被模式限定）。返回类型可以是一种基本类型、组合类型或者域类型，也可以引用一个表列的类型。根据实现语言，也可以允许指定cstring之类的“伪类型”。如果该函数不会返回一个值，可以指定返回类型为void。

当有OUT或者INOUT参数时，可以省略RETURNS子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

SETOF修饰符表示该函数将返回一个项的集合而不是一个单一项。

可以写table_name.column_name%TYPE来引用一列的类型。

column_name

RETURNS TABLE语法中一个输出列的名称。这实际上是另一种声明OUT参数的方法，不过RETURNS TABLE也隐含了RETURNS SETOF。

column_type

RETURNS TABLE语法中的输出列的数据类型。

lang_name

用以实现该函数的语言的名称。可以是sql、c、internal或者一个用户定义的过程语言的名称，例如plpgsql。不推荐用单引号包围该名称，并且要求区分大小写。

TRANSFORM { FOR TYPE type_name } [, ...] }

一个由转换构成的列表，对该函数的调用适用于它们。转换在 SQL 类型和语言相关的数据类型之间进行变换，详见CREATE TRANSFORM。过程语言实现通常把有关内建类型的知

识硬编码在代码中，因此那些不需要列举在这里。如果一种过程语言实现不知道如何处理一种类型并且没有转换被提供，它将回退到一种默认的行为来转换数据类型，但是这取决于具体实现。

WINDOW

WINDOW表示该函数是一个窗口函数而不是一个普通函数。当前只用于用 C 编写的函数。在替换一个现有函数定义时，不能更改WINDOW属性。

IMMUTABLE
STABLE
VOLATILE

这些属性告知查询优化器该函数的行为。最多只能指定其中一个。如果这些都不出现，则会默认为VOLATILE。

IMMUTABLE表示该函数不能修改数据库并且对于给定的参数值总是会返回相同的值。也就是说，它不会做数据库查找或者使用没有在其参数列表中直接出现的信息。如果给定合格选项，任何用全常量参数对该函数的额调用可以立刻用该函数值替换。

STABLE表示该函数不能修改数据库，并且对于相同的参数值，它在一次表扫描中将返回相同的结果。但是这种结果在不同的 SQL 语句执行期间可能会变化。对于那些结果依赖于数据库查找、参数变量（例如当前时区）等的函数来说，这是合适的（对希望查询被当前命令修改的行的AFTER触发器不适合）。还要注意current_timestamp函数族适合被标记为稳定，因为它们的值在一个事务内不会改变。

VOLATILE表示该函数的值在一次表扫描中都有可能改变，因此不能做优化。在这种意义上，相对较少的数据库函数是不稳定的，一些例子
是random()、currval()、timeofday()。但是注意任何有副作用的函数都必须被分类为不稳定的，即便其结果是可以预测的，这是为了调用被优化掉。一个例子是setval()。

更多细节可见第 38.7 节

LEAKPROOF

LEAKPROOF表示该函数没有副作用。它不会泄露有关其参数的信息（除了通过返回值）。例如，一个只对某些参数值抛出错误消息而对另外一些却不抛出错误的函数不是防泄漏的，一个把参数值包括在任何错误消息中的函数也不是防泄漏的。这会影响系统如何执行在使用security_barrier选项创建的视图或者开启了行级安全性的表上执行查询。对于包含有非防泄漏函数的查询，系统将在任何来自查询本身的用户提供条件之前强制来自安全策略或者安全屏障的条件，防止无意中的数据暴露。被标记为防泄漏的函数和操作符被假定是可信的，并且可以在安全性策略和安全性屏障视图的条件之前被执行。此外，没有参数的函数或者不从安全屏障视图或表传递任何参数的函数不一定要被标记为防泄漏的。详见CREATE VIEW和第 41.5 节这个选项只能由超级用户设置。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT（默认）表示在某些参数为空值时应正常调用该函数。如果有必要，函数的作者应该负责检查空值并且做出适当的相应。

RETURNS NULL ON NULL INPUT或STRICT表示只要其任意参数为空值，该函数就会返回空值。如果指定了这个参数，当有空值参数时该函数不会被执行，而是自动返回一个空值结果。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER表示要用调用该函数的用户的特权来执行它。这是默认值。SECURITY DEFINER指定要用拥有该函数的用户的特权来执行该函数。

为了符合 SQL，允许使用关键词EXTERNAL。但是它是可选的，因为与 SQL 中不同，这个特性适用于所有函数而不仅是那些外部函数。

PARALLEL

PARALLEL UNSAFE表示该函数不能在并行模式中运行并且 SQL 语句中存在一个这样的函数会强制使用顺序执行计划。这是默认选项。PARALLEL RESTRICTED表示该函数能在并行模式中运行，但是其执行被限制在并行组的领导者中。PARALLEL SAFE表示该函数对于在并行模式中运行是安全的并且不受限制。

如果函数修改任何数据库状态、会使用子事务之类的方式改变事务、访问序列或者对设置（如setval）做出持久性的更改，它们就应该被标记为并行不安全。如果它们访问临时表、客户端连接状态、游标、预备语句或者系统无法在并行模式中同步的本地后端状态（例如setseed只能在组领导者中执行，因为另一个进程所作的更改不会在领导者中被反映出来），它们应该被标为并行受限。通常，如果一个函数是受限的或者不安全的却被标成了安全，或者它本来是不安全的却被标成了受限，在并行查询中执行时它可能会抛出错误或者产生错误的答案。如果被错误的标记， C 语言函数理论上可能展现出完全无法定义的行为，因为系统没有办法保护自己不受任意的 C 代码影响，但是在大部分情况下其结果也不会比任何其他函数差到哪里去。如果有疑问，函数应该被标为UNSAFE，这也是默认值。

COST execution_cost

一个给出该函数的估计执行代价的正数，单位是cpu_operator_cost。如果该函数返回一个集合，这就是每个被返回行的代价。如果没有指定代价，对 C 语言和内部函数会指定为 1 个单位，对其他语言的函数则会指定为 100 单位。更大的值会导致规划器尝试避免对该函数的不必要的过多计算。

ROWS result_rows

一个正数，它给出规划器期望该函数返回的行数估计。只有当该函数被声明为返回一个集合时才允许这个参数。默认假设为 1000 行。

configuration_parameter
value

SET子句导致进入该函数时指定配置参数将被设置为指定值。并且在该函数退出时恢复到该参数之前的值。SET FROM CURRENT会把CREATE FUNCTION被执行时该参数的当前值保存为进入该函数时将被应用的值。

如果一个SET子句被附加到一个函数，那么在该函数内为同一个变量执行的SET LOCAL命令会被限制于该函数：在函数退出时该配置参数之前的值仍会被恢复。不过，一个普通的SET命令（没有LOCAL）会覆盖SET子句，更像一个之前的SET LOCAL命令所做的那样：这种命令的效果在函数退出后将会持续，除非当前事务被回滚。

更多有关允许的参数名和参数值的信息请见SET和第 19 章

definition

一个定义该函数的字符串常量，其含义取决于语言。它可以是一个内部函数名、一个对象文件的路径、一个 SQL 命令或者用一种过程语言编写的文本。

美元引用第 4.1.2.4 通常对书写函数定义字符串有所帮助，而普通单引号语法则不会有用。如果没有美元引用，函数定义中的任何单引号或者反斜线必须用双写来转义。

obj_file, link_symbol

当 C 语言源代码中该函数的名称与 SQL 函数的名称不同时，这种形式的AS子句被用于动态可载入 C 语言函数。字符串obj_file是包含编译好的C函数的动态库文件的名称，它会由LOAD命令解析。字符串link_symbol是该函数的链接符号，也就是该函数在 C 语言源代码中的名称。如果省略链接符号，它将被假定为要定义的 SQL 函数的名称。所有函

数的C名称都必须不同，因此必须为重载的C函数给出不同的C名称（例如把参数类型作为C名称的一部分）。

在重复调用引用同一对象文件的CREATE FUNCTION时，对每个会话该文件只会被载入一次。要卸载并且重新装载该文件（可能是在开发期间），需要开始一个新会话。

编写函数的进一步信息可以参考第 38.3 节

重载

PostgreSQL允许函数重载，也就是说同一个名称可以被用于多个不同的函数，只要它们具有可区分的输入参数类型。不管是否使用它，在有些用户不信任另一些用户的数据库中调用函数时，这种兼容性需要安全性的预防措施，请参考第 10.3 节

如果两个函数具有相同的名称和输入参数类型，它们被认为相同（不考虑任何OUT参数）。因此这些声明会冲突：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

具有不同参数类型列表的函数在创建时将不会被认为是冲突的，但是如果默认值被提供，在使用时它们有可能会冲突。例如，考虑

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

调用foo(10)将会失败，因为在要决定应该调用哪个函数时会有歧义。

注解

允许把完整的SQL类型语法用于声明一个函数的参数和返回值。不过，CREATE FUNCTION会抛弃带圆括号的类型修饰符（例如类型numeric的精度域）。例如CREATE FUNCTION foo (varchar(10)) ...和CREATE FUNCTION foo (varchar) ...完全一样。

在用CREATE OR REPLACE FUNCTION替换一个现有函数时，对于更改参数名是有限制的。不能更改已经分配给任何输入参数的名称（不过可以给之前没有名称的参数增加名称）。如果有多于一个输出参数，不能更改输出参数的名称，因为可能会改变描述函数结果的匿名组合类型的列名。这些限制是为了确保函数被替换时，已有的对该函数的调用不会停止工作。

如果一个被声明为STRICT的函数带有一个VARIADIC参数，会严格检查该可变数组作为一个整体是否为非空。如果该数组有空值元素，该函数仍将被调用。

示例

这里是一些小例子，它们可以帮你了解函数创建。更多信息和例子可见 第 38.3 节

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

在PL/pgSQL中，使用一个参数名称增加一个整数：

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
```

```
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

返回一个包含多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

你可以用更复杂的方式（用一个显式命名的组合类型）来做同样的事情：

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

另一种返回多列的方法是使用一个TABLE函数：

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

不过，TABLE函数与之前的例子不同，因为它实际返回了一个记录集合而不只是一个记录。

安全地编写 SECURITY DEFINER函数

因为一个SECURITY DEFINER函数会被以创建它的用户的特权来执行，需要小心地确保该函数不会被误用。为了安全，search_path应该被设置为排除任何不可信用户可写的模式。这可以阻止恶意用户创建对象（例如表、函数以及操作符）来掩饰该函数所要用到的对象。在这方面特别重要的是临时表模式，默认情况下它会第一个被搜索并且通常对任何用户都是可写的。可以通过强制最后搜索临时模式来得到一种安全的布局。要这样做，把pg_temp写成search_path中的最后一项。这个函数展示了安全的用法：

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- 设置一个安全的 search_path: 受信的模式，然后是 'pg_temp'。
```

```
    SET search_path = admin, pg_temp;
```

这个函数的目的是为了访问表admin.pwds。但是如果没有SET子句或者带有SET子句却只提到admin，该函数会变成创建一个名为pwds的临时表。

在PostgreSQL 版本 8.3 之前，SET子句不可用，因而较老的函数可能包含相当复杂的逻辑来保存、设置以及恢复search_path。对于这种目的，SET子句更容易。

另一点要记住的是默认情况下，会为新创建的函数给PUBLIC授予执行特权（详见GRANT）。你常常会希望把安全定义器函数的使用限制在某些用户中。要这样做，你必须收回默认的PUBLIC特权，然后选择性地授予执行特权。为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建它并且设置特权。例如：

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

兼容性

SQL标准中定义了CREATE FUNCTION命令。PostgreSQL的版本与之类似但不完全兼容。属性是不可移植的，不同的可用语言也是不能移植的。

对于和一些其他数据库系统的兼容性，argmode可以被写在argname之前或者之后。但只有第一种方式是兼容标准的。

对于参数默认值，SQL 标准只指定带有DEFAULT关键词的语法。带有=的语法被用在 T-SQL 和 Firebird 中。

另见

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — 定义一个新的数据库角色

大纲

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

其中 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

CREATE GROUP现在是 CREATE ROLE的一种别名。

兼容性

在 SQL 标准中没有CREATE GROUP语句。

另见

CREATE ROLE

CREATE INDEX

CREATE INDEX — 定义一个新索引

大纲

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
 [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC |
 DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

描述

CREATE INDEX在指定关系的指定列上构建 一个索引，该关系可以是一个表或者一个物化视图。索引主要被用来提升 数据库性能（不过不当的使用会导致性能变差）。

索引的键域被指定为列名或者写在圆括号中的表达式。如果索引方法支持 多列索引，可以指定多个域。

一个索引域可以是一个从表行的一列或者更多列值进行计算的表达式。 这种特性可以被用来获得对基于基本数据某种变换的数据的快速访问。 例如，一个在upper(col)上计算的索引可以允许子句 WHERE upper(col) = 'JIM'使用索引。

PostgreSQL提供了索引方法 B-树、哈希、GiST、SP-GiST、GIN 以及 BRIN。用户也可以定义自己的索引 方法，但是相对较复杂。

当WHERE子句存在时，会创建一个 部分索引。部分索引只包含表中一部分行的项， 通常索引这一部分会比表的其他部分更有用。例如，如果有一个表包含了 已付和未付订单，其中未付订单占了整个表的一小部分并且是经常被使用 的部分，可以通过只在这一部分上创建一个索引来改进性能。另一种可能 的应用是使用带有UNIQUE的 WHERE在表的一个子集上强制唯一性。更多的讨论 请见第 11.8 节

WHERE子句中使用的表达式只能引用底层表的列，但 它可以引用所有列而不仅仅是被索引的列。当前， WHERE中也禁止使用子查询和聚集表达式。同样的 限制也适用于表达式索引中的表达式域。

所有在索引定义中使用的函数和操作符必须是“不可变的”， 就是说它们的结果必须仅依赖于它们的参数而不受外在因素（例如另 一个表的内容和当前的时间）的影响。这种限制确保了索引的行为是 良定的。要在一个索引表达式或者WHERE子句中 使用用户定义的函数，记住在创建函数时把它标记为不可变。

参数

UNIQUE

导致系统在索引被创建时（如果数据已经存在）或者加入数据时 检查重复值。会导致重复项的数据插入或者更新尝试将会产生一 个错误。

当唯一索引被应用在分区边上时会有额外的限制，请参考CREATE TABLE。

CONCURRENTLY

当使用了这个选项时，PostgreSQL在构建索引时 不会取得任何会阻止该表上并发插入、更新或者删除的锁。而标准的索引 构建将会把表锁住以阻止对表的写（但不阻塞读），这种锁定会持续到索 引创建完毕。在使用这个选项时有多个需要注意的地方 — 请见 并发构建索引。

IF NOT EXISTS

如果一个同名关系已经存在则不要抛出错误。这种情况下会发出一个提示。 注意着并不保证现有的索引与将要创建的索引有任何相似。当 IF NOT EXISTS被指定时，需要指定索引名。

INCLUDE

可选的INCLUDE子句指定一个列的列表，其中的列将被包括在索引中作为非键列。非键列不能作为索引扫描的条件，并且该索引所强制的任何唯一性或者排除约束都不会考虑它们。不过，只用索引的扫描可以返回非键列的内容而无需访问该索引的基表，因为在索引项中就能直接拿到它们。因此，非键列的增加允许查询使用只用索引的扫描，否则就无法使用。

保守地向索引中增加非键列是明智的，特别是很宽的列。如果一个索引元组超过索引类型允许的最大尺寸，数据插入将会失败。在任何情况下，非键列都会重复来自索引基表的数据并且让索引的尺寸膨胀，因此可能会拖慢搜索。

INCLUDE子句中列出的列不需要合适的操作符类，甚至数据类型没有为给定的访问方法定义操作符类的列都可以包括在这个子句中。

不支持把表达式作为被包括列，因为它们不能被用在只用索引的扫描中。

当前，仅有B-树索引访问方法支持这一特性。在B-树索引中，INCLUDE子句中列出的列的值被包括在对应于堆元组的叶子元组中，但是不包括在用于树导航的上层索引项中。

name

要创建的索引名称。这里不能包括模式名，因为索引总是被创建在其基表所在 的模式中。如果索引名称被省略，PostgreSQL将基于基 表名称和被索引列名称选择一个合适的名称。

ONLY

如果该表是分区表，指示不要在分区上递归创建索引。默认会递归创建索引。

table_name

要被索引的表的名称（可以被模式限定）。

method

要使用的索引方法的名称。可以选择 btree、hash、 gist、spgist、 gin以及brin。 默认方法是btree。

column_name

一个表列的名称。

expression

一个基于一个或者更多个表列的表达式。如语法中所示，表达式通常必须 被写在圆括号中。不过，如果该表达式是一个函数调用的形式，圆括号可 以被省略。

collation

要用于该索引的排序规则的名称。默认情况下，该索引使用被索引列 的排序规则或者被索引表达式的结果排序规则。当查询涉及到使用非 默认排序规则的表达式时，使用非默认排序规则的索引就能排上用场。

opclass

一个操作符类的名称。详见下文。

ASC

指定上升排序（默认）。

DESC

指定下降排序。

NULLS FIRST

指定把空值排序在非空值前面。在指定DESC时， 这是默认行为。

NULLS LAST

指定把空值排序在非空值后面。在没有指定DESC时， 这是默认行为。

storage_parameter

索引方法相关的存储参数的名称。详见 索引存储参数。

tablespace_name

在其中创建索引的表空间。如果没有指定，将会使用 default_tablespace。或者对临时表上的索引使用 temp_tablespaces。

predicate

部分索引的约束表达式。

索引存储参数

可选的WITH子句为索引指定存储参数。每一种 索引方法都有自己的存储参数集合。B-树、哈希、GiST以及SP-GiST索引方法都接受这个参数：

fillfactor

索引的填充因子是一个百分数，它决定索引方法将尝试填充索引页面的充满程度。对于B-树，在初始的索引构建过程中，叶子页面会被填充至该百分数，当在索引右端扩展索引（增加新的最大键值）时也会这样处理。如果页面后来被完全填满，它们就会被分裂，导致索引的效率逐渐退化。B-树使用了默认的填充因子 90，但是也可以选择为 10 到 100 的任何整数值。如果表是静态的，那么填充因子 100 是最好的，因为它可以让索引的物理尺寸最小化。但是对于更新负荷很重的表，较小的填充因子有利于最小化对页面分裂的需求。其他索引方法以不同但是大致类似的方式使用填充因子，不同方法的默认填充因子也不相同。

B-树索引还额外接受这个参数：

vacuum_cleanup_index_scale_factor

vacuum_cleanup_index_scale_factor针对每个索引的值。

GiST还额外接受这个参数：

buffering

决定是否用第 64.4.1 节描述的缓冲构建技术来构建索引。OFF会禁用它，ON则启用该特性，如果设置为AUTO则初始会禁用它，但是一旦索引尺寸到达effective_cache_size就会随时打开。默认值是AUTO。

GIN索引接受不同的参数：

fastupdate

这个设置控制第 66.4.1 节描述的快速更新 技术的使用。它是一个布尔参数：ON启用快速更新， OFF禁用之（ON和OFF的其他 写法在第 19.1 节有介绍）。默认是 ON。

> **注意**
>
> 通过ALTER INDEX关闭fastupdate 会阻止未来的更新进入到待处理索引项列表中，但它不会自己处理之前的 待处理项。可以使用VACUUM或者调用gin_clean_pending_list确保处理完待处理列表的项。

gin_pending_list_limit

自定义gin_pending_list_limit参数。这个值 要以千字节来指定。

BRIN索引接受不同的参数：

pages_per_range

定义用于每一个BRIN索引项的块范围由多少个表块组成（详见 第 67.1 节。默认是128。

autosummarize

定义是否只要在下一个页面上检测到插入就为前面的页面范围运行概要操作。

并发构建索引

创建索引可能会干扰数据库的常规操作。通常 PostgreSQL会锁住要被索引的表，让它不能被写入， 并且用该表上的一次扫描来执行整个索引的构建。其他事务仍然可以读取表 ， 但是如果它们尝试在该表上进行插入、更新或者删除，它们会被阻塞直到索引 构建完成。如果系统是一个生产数据库，这可能会导致严重的后果。索引非常 大的表可能会需要很多个小时，而且即使是较小的表，在构建索引过程中阻塞 写入者一段时间在生产系统中也是不能接受的。

PostgreSQL支持构建索引时不阻塞写入。这种方法通过 指定CREATE INDEX的CONCURRENTLY选项 实现。当使用这个选项时，PostgreSQL必须执行该表的 两次扫描，此外它必须等待所有现有可能会修改或者使用该索引的事务终止。因此这种 方法比起标准索引构建过程来说要做更多工作并且需要更多时间。不过，由于它 允许在构建索引时继续普通操作，这种方式对于在生产环境中增加新索引很有用。 当然，由索引创建带来的额外 CPU 和 I/O 开销可能会拖慢其他操作。

在并发索引构建中，索引实际上在一个事务中被录入到系统目录，然后在两个 事务中发生两次表扫描。在每一次表扫描之前，索引构建必须等待已经修改了 表的现有事务终止。在第二次扫描之后，索引构建必须等待任何持有早于第二 次扫描的快照（见第 13 章的事务终止。然后该索引最终 能被标记为准备好使用，并且CREATE INDEX命令终止。 不过即便那样，该索引也不是立刻可以用于查询：在最坏的情况下，只要早于 索引构建开始时存在的事务存在，该索引就无法使用。

如果在扫描表示出现问题，例如死锁或者唯一索引中的唯一性被违背， CREATE INDEX将会失败，但留下一个“不可用” 的索引。这个索引会被查询所忽略，因为它可能不完整。不过它仍将消耗更新 开销。psql的\d命令将把这类索引报告为 INVALID：

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
    "idx" btree (col) INVALID
```

这种情况下推荐的恢复方法是删除该索引并且尝试再次执行 CREATE INDEX CONCURRENTLY（另一种可能性是用 REINDEX重建该索引。不过，由于 REINDEX不支持并发构建，这种选择不那么有吸引力）。

并发构建一个唯一索引时需要注意的另一点是，当第二次表扫描开始时，唯一约束 已经被强制在其他事务上。这意味着在该索引变得可用之前，其他查询中可能就会 报告该约束被违背，或者甚至在索引构建最终失败的情况中也是这样。还有，如果在 第二次扫描时发生失败，“无效的”索引也会继续强制它的唯一性约束。

表达式索引和部分索引的并发构建也被支持。在这些表达式计算过程中发生的 错误可能导致和上述唯一约束违背类似的行为。

常规索引构建允许在同一个表上同时构建其他常规索引，但是在一个表上同时 只能有一个并发索引构建发生。在两种情况下，在索引被构建时不允许表的模式修改。另一个不同是，一个常规CREATE INDEX 命令可以在一个事务块中执行，但是 CREATE INDEX CONCURRENTLY不行。

注解

关于索引何时能被使用、何时不被使用以及什么情况下它们有用的信息请 见第 11 章

当前，只有 B-树、GiST、GIN 和 BRIN 索引方法支持多列索引。默认最多可以 索引 32 个域（可以在构建 PostgreSQL修改这种限制）。当前 只有 B-树支持唯一索引。

为索引的每一列可以指定一个操作符类。该操作符 类标识要被该索引用于该列的操作符。例如，一个四字节整数上的 B-树索引 会使用int4_ops类。这个操作符类包括了用于四字节整数 的比较函数。实际上，通常列数据类型的默认操作符类就足够了。对某些数据 类型指定操作符类的主要原因是，可能会有多于一种有意义的顺序。例如， 我们可能想用绝对值或者实数部分对复数类型排序。我们可以通过为该数据 类型定义两个操作符类来做到，并且在创建索引时选择其中合适的类。更多 有关操作符类的信息请见第 11.10 节以及第 38.15 节

当在一个分区表上调用CREATE INDEX时，默认的行为是递归到所有的分区上以确保它们都具有匹配的索引。每一个分区首先会被检查是否有一个等效的索引存在，如果有则该索引将被挂接为被创建索引的一个分区索引，而被创建的索引将成为其父索引。如果不存在匹配的索引，则会创建一个新的索引并且自动进行挂接。如果命令中没有指定索引名称，每个分区中的新索引的名称将被自动决定。如果指定了ONLY选项，则不会进行递归，并且该索引会被标记为无效（一旦所有的分区都得到该索引，ALTER INDEX ... ATTACH PARTITION可以把该索引标记为有效）。不过，要注意不管是否指定这一选项，未来使用CREATE TABLE ... PARTITION OF创建的任何分区将自动有一个匹配的索引，不管有没有指定ONLY。

对于支持有序扫描的索引方法（当前只有 B-树），可以指定可选子句ASC、 DESC、NULLS FIRST以及NULLS LAST 来修改索引的排序顺序。由于一个有序索引能前向或者反向扫描，通常创建一个 单列DESC索引没什么用处 — 一个常规索引已经提供了排序 顺序。这些选项的价值是可以创建多列索引，让它的排序顺序匹配有混合排序要求 的查询，例如SELECT ... ORDER BY x ASC, y DESC。如果你想要在依靠索引避免排序步骤的查询中支持 “空值排序低”这种行为，NULLS选项就能派上用场，默认 的行为是“空值排序高”。

对于大多数索引方法，索引的创建速度取决于 maintenance_work_mem的设置。较大的值将会减少 索引创建所需的时间，当然不要把它设置得超过实际可用的内存量（那会迫使 机器进行交换）。

PostgreSQL可以在构建索引时利用多个CPU以更快地处理表行。这种特性被称为并行索引构建。对于支持并行构建索引的索引方法（当前只有B-树），maintenance_work_mem指定每次索引构建操作整体可用的最大内存量，而不管启动了多少工作者进程。一般来说，一个代价模型（如果有）自动判断应该请求多少工作者进程。

增加maintenance_work_mem可以让并行索引构建受益，而等效的串行索引构建将无法受益或者得到很小的益处。注意maintenance_work_mem可能会影响请求的工作者进程的数量，因为并行工作者必须在总的maintenance_work_mem预算中占有至少32MB的份额。还必须有32MB的份额留给领袖进程。增加max_parallel_maintenance_workers可以允许使用更多的工作者，这将降低索引创建所需的时间，只要索引构建不是I/O密集型的。当然，还需要有足够的CPU计算能力，否则工作者们会闲置。

通过ALTER TABLE为parallel_workers设置一个值直接控制着CREATE INDEX会对表请求多少并行工作者进程。这会完全绕过代价模型，并且防止maintenance_work_mem对请求多少并行工作者产生影响。通过ALTER TABLE将parallel_workers设置为0将禁用所有情况下的并行索引构建。

> **提示**
>
> 在把parallel_workers用于调优一次索引构建之后，你可能想要重置parallel_workers。这可以避免对查询计划的无意更改，因为parallel_workers影响所有的并行表扫描。

虽然带有CONCURRENTLY选项的CREATE INDEX支持并行构建并且没有特殊的限制，但只有第一次表扫描会实际以并行方式执行。

使用DROP INDEX可以移除一个索引。

以前的PostgreSQL发行也有一种 R-树 索引方法。这种方法已经被移除，因为它比起 GiST 方法来说没有什么明显的 优势。如果指定了USING rtree，CREATE INDEX 将会把它解释为USING gist，以便把旧的数据库转换成 GiST。

示例

在表films中的列title上创建一个 B-树索引：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

要在表films的列title上创建一个唯一的B-树索引并且包括列director和rating：

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

在表达式lower(title)上创建一个索引来允许高效的大小写 无关搜索：

```
CREATE INDEX ON films ((lower(title)));
```

（在这个例子中我们选择省略索引名称，这样系统会选择一个名字，通常是films_lower_idx）。

创建一个具有非默认排序规则的索引：

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

创建一个具有非默认空值排序顺序的索引：

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

创建一个具有非默认填充因子的索引：

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

创建一个禁用快速更新的GIN索引：

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate =
 off);
```

在表films中的列code上创建一个而索引并且 把索引放在表空间indexspace中：

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

在一个点属性上创建一个 GiST 索引，这样我们可以在转换函数的结果 上有效地使用 box 操作符：

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

创建一个表而不排斥对表的写操作：

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

兼容性

CREATE INDEX是一种 PostgreSQL的语言扩展。在 SQL 标准中 没有对于索引的规定。

另见

ALTER INDEX, DROP INDEX

CREATE LANGUAGE

CREATE LANGUAGE — 定义一种新的过程语言

大纲

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

描述

CREATE LANGUAGE为一个 PostgreSQL数据库注册一种新的 过程语言。接着，可以用这种新语言定义函数和存储过程。

> **注意**
>
> 从PostgreSQL 9.1 开始，大多数 过程语言已经被做成了“扩展”，并且应该用 CREATE EXTENSION而不是 CREATE LANGUAGE来安装。 CREATE LANGUAGE的直接使用现在应该 被限制在扩展安装脚本中。如果在数据库中有一种“裸”语言（ 可能是一次升级的结果），可以用 CREATE EXTENSION langname FROM unpackaged把它转换成一个扩展。

CREATE LANGUAGE实际上把该语言名称与 负责执行用该语言编写的函数的处理器函数关联在一起。有关语言处理器的 更多信息可以参考第 56 章

有两种形式的CREATE LANGUAGE命令。在 第一种形式中，用户只提供想要的语言的名称。 PostgreSQL服务器会查询 pg_pltemplate系统目录来决定正确的参数。在第二种形式中，用户 要提供语言参数和语言名称。第二种形式可以被用来创建一种没有定义在pg_pltemplate中的语言，但是这种方法被认为即将 废弃。

当服务器在pg_pltemplate目录中为给定的语言名称 找到一个项时，即使命令中已经包括了语言参数，它也将使用目录中的 数据。这种行为简化了旧转储文件的载入，旧转储文件很可能包含过时的 信息。

通常，用户必须拥有 PostgreSQL超级用户特权来注册 一种新的语言。不过，如果该语言被列举在 pg_pltemplate目录中并且被标记为允许 由数据库拥有者创建（tmpldbacreate为真），则数据 库的拥有者可以把新语言注册在数据库中。默认是可信的语言能够由数据 库拥有者创建，但是超级用户可以通过修改 pg_pltemplate的内容来调整这种行为。 语言的创建者会成为它的拥有者，并且以后可以删除它、对它重命名或者 把它赋予给一个新的拥有者。

CREATE OR REPLACE LANGUAGE将创建 或者替换一种现有的定义。如果该语言已经存在，其参数会被根据指定的 值或者来自pg_pltemplate的值更新。但 该语言的拥有关系和权限设置不会更改，并且任何已有的用该语言编写的 函数仍然被假定有效。除了创建一种语言的普通特权需求，用户还必须是 超级用户或者已有语言的拥有者。REPLACE情况主要被用来 确保该语言存在。如果该语言有一个 pg_pltemplate项，那么 REPLACE将不会实际更改现有定义的任何东西，除非从该语 言被创建以来pg_pltemplate已经被修改 过（很少见的情况）。

参数

TRUSTED

TRUSTED指定该语言不会授予用户不该具有的 数据访问。如果在注册语言时这个关键词被省略，只有具有 PostgreSQL超级用户特权的用户才能 使用该语言创建新函数。

PROCEDURAL

这是一个噪声词。

name

新过程语言的名称。该名称必须在该数据库的语言中唯一。

为了向后兼容，名称可以用单引号围绕。

HANDLER call_handler

call_handler 是一个之前注册的函数的名称，它将被调用来执行该过程语言的函数。一种过程语言的调用处理器必须以一种编译型语言（如 C）编写并且 具有版本 1 的调用约定，它必须在 PostgreSQL内注册为一个没有 参数并且返回language_handler类型的函数。 language_handler是一种占位符类型， 它被用来 标识该函数为一个调用处理器。

INLINE inline_handler

inline_handler 是一个之前注册的函数的名称，它将被调用来执行一个该语言的匿名代码块（ DO命令）。如果没有指定 inline_handler函数，则 该语言不支持匿名代码块。该处理器函数必须接受一个internal 类型的参数，该参数将是DO命令的内部表示，而且它通常 将返回void。该处理器的返回值会被忽略。

VALIDATOR valfunction

valfunction is the 是一个之前注册的函数的名称，当一个该语言的新函数被创建时会调用该函数来 验证新函数。如果没有指定验证器函数，那么一个新函数被创建时不会被检查。 验证器函数必须接受一个oid类型的参数，它将是要被创建的 函数的 OID，而且它通常将返回void。

一个验证器函数通常会检查函数体中的语法正确性，但是它也能查看函数的其他 属性，例如该语言能否处理特定的参数类型。为了发出一个错误，验证器函数应该 使用ereport()函数。验证器函数的返回值会被忽略。

如果指定的语言名称在pg_pltemplate中有一项，服务器会忽略 TRUSTED选项和支持函数的名称。

注解

使用DROP LANGUAGE删除过程语言。

系统目录pg_language（见第 52.29 节记录着有关当前已安装的语言的信息。 还有，psql命令\dL列出已安装的语言。

要以一种过程语言创建函数，用户必须具有对于该语言的 USAGE特权。默认情况下，对于可信语言， USAGE被授予给PUBLIC（即所有人）。 如果需要可以将它收回。

过程语言对于单个数据库来说是本地的。但是，一种语言可以被安装在 template1数据库中，这会导致它在所有后续创建的 数据库中自动变得可用。

如果对语言在服务器的pg_pltemplate中没有一项， 调用处理器函数、内联处理器函数（如果有）以及验证器函数（如果有） 必须已经存在。但是当有一个那样的项时，这些函数不必已经存在。如果 它们在数据库中不存在，将自动定义它们（如果安装中实现该语言的共享 库不可用可能会导致CREATE LANGUAGE失败）。

在PostgreSQL 版本 7.3 之前，需要 将处理器函数声明为返回占位符类型opaque而不是 language_handler。为了支持载入旧的转储文件， CREATE LANGUAGE将接受被声明为返回 opaque的函数，但是它将发出一个提示并且把该函数的声明 返回类型改为language_handler。

示例

创建任何标准过程语言的最好的方式是：

```
CREATE LANGUAGE plperl;
```

对于pg_pltemplate目录不知道的一种语言，需要这样的命令序列：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

CREATE LANGUAGE是一种 PostgreSQL扩展。

另见

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — 定义一个新的物化视图

大纲

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

描述

CREATE MATERIALIZED VIEW定义一个查询的物化视图。 在该命令被发出时，查询会被执行并且被用来填充该视图（除非使用了 WITH NO DATA），并且后来可能会用 REFRESH MATERIALIZED VIEW进行刷新。

CREATE MATERIALIZED VIEW类似于 CREATE TABLE AS，不过它还会记住被用来初始化该视图的查询， 这样它可以在后来被命令刷新。一个物化视图有很多和表相同的属性，但是不支持临时物化视图以及自动生成 OID。

参数

IF NOT EXISTS

如果已经存在一个同名的物化视图时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的物化视图与即将创建的物化视图相似。

table_name

要创建的物化视图的名称（可以被模式限定）。

column_name

新物化视图中的一个列名。如果没有提供列名，会从查询的输出列名来得到。

WITH (storage_parameter [= value] [, ...])

这个子句为新的物化视图指定可选的存储参数，详见 存储参数。所有CREATE TABLE支持的参数CREATE MATERIALIZED VIEW也支持，不过OIDS除外。 详见CREATE TABLE。

TABLESPACE tablespace_name

tablespace_name是 要把新物化视图创建在其中的表空间的名称。如果没有指定， 将查阅default_tablespace。

query

一个SELECT、TABLE 或者VALUES命令。这个查询将在一个安全受限的操作中运行。 特别地，对本身会创建临时表的函数的调用将会失败。

WITH [NO] DATA

这个子句指定物化视图是否在创建时被填充。如果不是，该物化视图将被标记为 不可扫描并且在REFRESH MATERIALIZED VIEW被使用前不能被查询。

兼容性

CREATE MATERIALIZED VIEW是一种 PostgreSQL扩展。

另见

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — 定义一个新的操作符

大纲

```
CREATE OPERATOR name (
    {FUNCTION|PROCEDURE} = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

描述

CREATE OPERATOR定义一个新的操作符 name。定义操作符 的用户会成为该操作符的拥有者。如果给出一个模式名，该操作符将被创建 在指定的模式中。否则它会被创建在当前模式中。

操作符名称是最多NAMEDATALEN-1（默认为 63） 个字符的序列，这些字符可以是：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

对名称的选择有一些限制：

- -- and /*不能出现在 操作符名称中，因为它们会被当做一段注释的开始。
- 多字符操作符名称不能以+或者- 结束，除非该名称也包含至少一个下列字符：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，@-是一个被允许的操作符名称，而 *-不是。这种限制允许 PostgreSQL解析 SQL 兼容的命令 而无需记号之间的空格。

- 将=>用作一个操作符名称已经不被推荐。在未来的发行中 可能会被禁用。

在输入时!=会被映射为<>， 因此这两个名字总是等效的。

必须至少定义LEFTARG和RIGHTARG中的一个。 对于二元操作符，两者都必须被定义。对于右一元操作符，只应该定义 LEFTARG，而对于左一元操作符只应该定义 RIGHTARG。

function_name函数 必须在之前已经用CREATE FUNCTION定义好， 并且必须被定义为接受正确数量的指定类型的参数。

在CREATE OPERATOR的语法中，关键词FUNCTION和PROCEDURE是等效的，但不管哪种情况下被引用的函数都必须是一个函数而不是过程。这里对关键词PROCEDURE的是用是有历史原因的，现在已经被废弃。

其他子句指定可选的操作符优化子句。它们的含义在 第 38.14 节有详细描述。

要创建一个操作符，必须具有参数类型和返回类型上的USAGE 特权，以及底层函数上的EXECUTE特权。如果指定了一个 交换子或者求反器操作符，必须拥有这些操作符。

参数

name

要定义的操作符的名称。允许使用的字符请见上文。名称可以被模式 限定，例如CREATE OPERATOR myschema.+ (...)。如果 没有被模式限定，该操作符将被创建在当前模式中。如果两个同一模式 中的操作符在不同的数据类型上操作，它们可以具有相同的名称。这被称为重载。

function_name

用来实现这个操作符的函数。

left_type

这个操作符的左操作数（如果有）的数据类型。忽略这个选项 可以表示一个左一元操作符。

right_type

这个操作符的右操作数（如果有）的数据类型。忽略这个选项 可以表示一个右一元操作符。

com_op

这个操作符的交换子。

neg_op

这个操作符的求反器。

res_proc

用于这个操作符的限制选择度估计函数。

join_proc

用于这个操作符的连接选择度估算函数。

HASHES

表示这个操作符可以支持哈希连接。

MERGES

表示这个操作符可以支持归并连接。

要在com_op 或者其他可选参数中给出一个模式限定的操作符名称， 请使用OPERATOR()语法，例如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注解

进一步的信息可参考第 38.13 节

无法在CREATE OPERATOR中指定一个操作符的 词法优先级，因为解析器的优先级行为是硬写在代码中的。详见 第 4.1.6 节

废弃的选项SORT1、SORT2、 LTCMP以及GTCMP以前被用来指定与支持 归并连接的操作符相关的排序操作符的名称。现在不再需要它们了，因为 相关操作符的信息可以在 B-树的操作符族中找到。如果给出了这些选项 之一，它会被忽略（除非是为了隐式设置MERGES为真）。

使用DROP OPERATOR从数据库中删除用户定义的操作符。 使用ALTER OPERATOR修改数据库中的操作符。

示例

下面的命令为数据类型box定义了一种新的操作符--面积相等：

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    FUNCTION = area_equal_function,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_function,
    JOIN = area_join_function,
    HASHES, MERGES
);
```

兼容性

CREATE OPERATOR是一种 PostgreSQL扩展。在 SQL 标准中没有用户定义操作符的规定。

另见

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — 定义一个新的操作符类

大纲

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR
 SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name
 ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

描述

CREATE OPERATOR CLASS创建新的操作符类。 一个操作符类定义一种特殊的数据类型如何被用于一个索引。操作符类指定为 该数据类型和索引方法扮演特殊角色或者“策略”的操作符。操作符 类还指定当该操作符类被选择用于一个索引列时，索引方法要使用的支持函数。 操作符类所使用的所有操作符和函数必须在操作符类被创建之前被定义好。

如果给出了一个模式名称，那么该操作符类会被创建在指定模式中。否则，它 会被创建在当前模式中。同一模式中的两个操作符类只有在被用于不同的索引 方法时才可以具有相同的名称。

定义操作符类的用户将成为其拥有者。当前，创建用户必须是超级用户（做出 这种限制是因为错误的操作符类定义会让服务器混淆甚至崩溃）。

CREATE OPERATOR CLASS当前不会检查操作符 类定义是否包括该索引方法所要求的所有操作符和函数，也不会检查这些操作符 和函数是否构成一个一致的集合。定义一个合法的操作符类是用户的责任。

相关的操作符类可以被组成操作符族。要把一个新的操作符类 加入到一个现有的族中，可以在CREATE OPERATOR CLASS中指定FAMILY选项。如果没有这个选项， 新的类会被放到一个同名的族中（如果族不存在会创建之）。

进一步的信息可参考第 38.15 节

参数

name

要创建的操作符类的名称。该名称可以被模式限定。

DEFAULT

如果存在，该操作符类将成为其数据类型的默认操作符类。对一种 特定的数据类型和索引方法至多有一个默认操作符类。

data_type

这个操作符类所用于的列数据类型。

index_method

这个操作符类所用于的索引方法的名称。

family_name

要把这个操作符类加入其中的已有操作符族的名称。如果没有指定， 将使用一个同名操作符族（如果还不存在则创建之）。

strategy_number

用于一个与该操作符类相关联的操作符的索引方法策略号。

operator_name

一个与该操作符类相关联的操作符的名称（可以被模式限定）。

op_type

在一个OPERATOR子句中，这表示该操作符的操作数数据 类型，或者用NONE来表示一个左一元或者右一元操作符。 在操作数数据类型与该操作符的数据类型相同的一般情况下，操作数的 数据类型可以被省略。

在一个FUNCTION子句中，这表示该函数要支持的操作数 数据类型，如果它与该函数的输入数据类型（对于 B-树比较函数和哈希 函数）或者操作符类的数据类型（对于 B-树排序支持函数和所有GiST、 SP-GiST、GIN 和 BRIN 操作符类中的函数）不同。这些默认值是正确的，并且 op_type因此不必 在FUNCTION子句中被指定，对于 B-树排序支持函数的情 况来说，这表示跨数据类型比较。

sort_family_name

一个现有btree操作符族的名称（可以是模式限定的）， 它描述与一种排序操作符相关联的排序顺序。

如果FOR SEARCH和FOR ORDER BY都没有被 指定，那么FOR SEARCH是默认值。

support_number

用于一个与该操作符类相关联的函数的索引方法支持函数编号。

function_name

一个用于该操作符类的索引方法支持函数的函数名称（可以是 模式限定的）。

argument_type

该函数的参数数据类型。

storage_type

实际存储在索引中的数据类型。通常这和列数据类型相同，但是有些 索引方法（当前有GiST、GIN 和 BRIN）允许它们不同。 除非索引方法允许使用不同的类型，STORAGE子句必须 被省略。 如果data_type列被指定为anyarray， 那么storage_type可以被声明为anyelement 以指示索引条目是属于为每个特定索引创建的实际数组类型的元素类型的成员。

OPERATOR、FUNCTION和STORAGE 子句可以以任何顺序出现。

注解

因为索引机制在使用函数之前不检查它们的权限，将一个函数或者操作符包括在 一个操作符类中相当于在其上授予公共执行权限。这对操作符类中很有用的函数 来说通常不成问题。

操作符不应该用 SQL 函数定义。SQL 函数很有可能会被内联到调用查询中，这 会妨碍优化器识别该查询匹配一个索引。

在PostgreSQL 8.4 之前， OPERATOR子句可以包括一个RECHECK选项。现在 已经不再支持，因为一个索引操作符是否为“有损的”现在是在运行 时实时决定的。这允许在一个操作符可能是或者可能不是有损的情况下有效地处理。

示例

下面的例子为数据类型_int4（int4数组） 定义了一个 GiST 索引操作符。完整的例子请见intarray模块。

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, smallint,
 oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

兼容性

CREATE OPERATOR CLASS是一种 PostgreSQL扩展。在 SQL 标准中没有 CREATE OPERATOR CLASS语句。

另见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — 定义一个新的操作符族

大纲

```
CREATE OPERATOR FAMILY name USING index_method
```

描述

CREATE OPERATOR FAMILY创建一个新的 操作符族。一个操作符族定义一个相关操作符类组成的集合，并且可能还　包含一些额外的、与这些操作符类兼容但对于任何个体索引的功能不是至关 重要的操作符和支持函数（对索引至关重要的操作符和函数应该被分组在 相关的操作符类中，而不是“松散地”在操作符中。通常，单一　数据类型操作符被限制在操作符类中，而跨数据类型操作符可以松散地存在　于一个包含用于两种数据类型的操作符类的操作符族中）。

新的操作符族初始时为空。应该通过发出后续的 CREATE OPERATOR CLASS命令来增加包含 在其中的操作符类，还可以用可选的 ALTER OPERATOR FAMILY命令来增加 “松散的”操作符和它们对应的支持函数。

如果给出一个模式名称，该操作符族会被创建在指定的模式中。否则，　它会被创建在当前模式中。只有当同一个模式中的两个操作符族是用于　不同的索引方法时，它们才能拥有相同的名字。

定义一个操作符族的用户将成为它的拥有者。当前，创建用户必须是超级用户（　做出这样的限制是因为错误的操作符族会让服务器混淆甚至崩溃）。

进一步的信息可以参考第　38.15　节

参数

name

要创建的操作符族的名称。该名称可以被模式限定。

index_method

这个操作符族要用于的索引方法的名称。

兼容性

CREATE OPERATOR FAMILY是一种 PostgreSQL扩展。在 SQL 标准中没有 CREATE OPERATOR FAMILY语句。

另见

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — 为一个表定义一条新的行级安全性策略

大纲

```
CREATE POLICY name ON table_name
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

描述

CREATE POLICY为一个表定义一条行级 安全性策略。注意为了应用已被创建的策略，在表上必须启用行级安全 性（使用ALTER TABLE ... ENABLE ROW LEVEL SECURITY）。

一条策略授予权限以选择、插入、更新或者删除匹配相关策略表达式的行。 现有的表行会按照USING中指定的表达式进行检查， 而将要通过INSERT或UPDATE创建 的新行会按照WITH CHECK中指定的表达式进行检查。 当USING表达式对于一个给定行返回真时，该行对用户 可见，而返回假或空时该行不可见。当一个WITH CHECK 表达式对一行返回真时，该行会被插入或更新，而如果返回假或空时会发生 一个错误。

对于INSERT和 UPDATE语句，在BEFORE 触发器被引发后并且在任何数据修改真正发生之前，WITH CHECK表达式会被强制。因此， 一个BEFORE ROW触发器可以修改要被插入的数据， 从而影响安全性策略检查的结果。WITH CHECK表达式 在任何其他约束之前被强制。

策略名称是针对每个表的。因此，一个策略名称可以被用于很多个不同的表 并且对于不同的表呈现适合于该表的定义。

策略可以被应用于特定的命令或者特定的角色。除非特别指定，新创建的策略 的默认行为是适用于所有命令和角色。多个策略可以应用于单个命令，更多细节请见下文。表 2405总结了不同类型的策略如何应用于特定的命令。

对同时具有USING和WITH CHECK 表达式（ALL和UPDATE）的策略， 如果没有定义WITH CHECK表达式，那么 USING表达式将被用于决定哪些行可见（普通 USING情况）以及允许哪些新行被增加（ WITH CHECK情况）。

如果为一个表启用了行级安全性但是没有适用的策略存在，将假定为一种 “默认否定”策略，这样任何行都不可见也不可更新。

参数

name

要创建的策略的名称。这必须和该表上已有的任何其他策略名称相区分。

table_name

该策略适用的表的名称（可以被模式限定）。

PERMISSIVE

指定策略被创建为宽容性策略。适用于一个给定查询的所有宽容性策略将被使用布尔“OR”操作符组合在一起。通过创建宽容性策略，管理员可以在能被访问的记录集合中进行增加。策略默认是宽容性的。

RESTRICTIVE

指定策略被创建为限制性策略。适用于一个给定查询的所有限制性策略将被使用布尔“AND”操作符组合在一起。通过创建限制性策略，管理员可以减少能被访问的记录集合，因为每一条记录都必须通过所有的限制性策略。

注意在限制性策略真正能发挥作用减少访问之前，需要至少一条宽容性策略来授予对记录的访问。如果只有限制性策略存在，则没有记录能被访问。当宽容性和限制性策略混合存在时，只有当一个记录能通过至少一条宽容性策略以及所有的限制性策略时，该记录才是可访问的。

command

该策略适用的命令。合法的选项是 ALL、SELECT、 INSERT、UPDATE 以及DELETE。 ALL为默认。有关这些策略如何被应用的 细节见下文。

role_name

该策略适用的角色。默认是PUBLIC，它将把策略应用 到所有的角色。

using_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被增加到引用该表的查询。 让这个表达式返回真的行将可见。让这个表达式返回假或者空的任何行 将对用户不可见（在SELECT中）并且将对修改不可用（ 在UPDATE或DELETE中）。这类行 会被悄悄地禁止而不会报告错误。

check_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被用在该表上的 INSERT以及 UPDATE查询中。只有让该表达式计算为真 的行才被允许。如果任何被插入的记录或者跟新后的记录导致该表达式计 算为假或者空，则会抛出一个错误。注意 check_expression 是根据行的新内容而不是原始内容计算的。

针对每种命令的策略

ALL

为一条策略使用ALL表示它将适用于所有命令， 不管命令的类型如何。如果存在一条ALL策略 以及更多特定的策略，则ALL策略和那些策略 会被应用。此外， ALL策略将同时适用于一个查询的选择端和修 改端，如果只定义了一个USING表达式则将 该USING表达式用于两种情况。

例如，如果发出一个UPDATE，那么 ALL策略将同时影响UPDATE 能更新哪些行（应用USING表达式）以及更新后 的行是否被允许加入到表中（如果定义了WITH CHECK 表达式，则应用之；否则使用USING表达式）。 如果一条INSERT 或者UPDATE命令尝试增加行到表中， 但行没有通过ALL策略的 WITH CHECK表达式，则整个语句将会中断。

SELECT

对一条策略使用SELECT表示它将适用于 SELECT查询，并且无论何时都要求该约束所在的关系上 的SELECT权限。其结果是在一次 SELECT查询期间，只有该关系中那些通过了SELECT策略的记录才将被返回，并且查询要求 SELECT权限，例如 UPDATE也将只能看到那些 SELECT策略允许的行。一条 SELECT策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中检索记录的情况。

INSERT

为一条策略使用INSERT表示它适用于 INSERT命令。没有通过这种策略的正在被插入的行会导致策略违背错误，并且整个INSERT命令将会中止。 一条INSERT策略不能具有USING表达式，因为它只适用于正在向关系增加记录的情况。

注意在带有ON CONFLICT DO UPDATE的INSERT中，只有对通过 INSERT路径追加到关系的行才会检查 INSERT策略的WITH CHECK 表达式。

UPDATE

对策略使用UPDATE 意味着它将应用于UPDATE、SELECT FOR UPDATE和SELECT FOR SHARE命令，还有INSERT 命令的辅助性的ON CONFLICT DO UPDATE 子句。由于UPDATE 需要提取现有的记录并且用新修改的记录代替，故UPDATE 策略接受USING 表达式和WITH CHECK表达式。USING 表达式决定UPDATE 命令将能看到哪些要对其操作的记录，而WITH CHECK表达式定义哪些被修改的行允许存回到关系中。

任何更新后的值无法通过WITH CHECK表达式的行 将会导致错误，并且整个命令将被中止。如果只指定了一个 USING子句，那么该子句将被用于 USING和WITH CHECK两种情况。

典型地，UPDATE命令也需要从待更新关系中的列读数据（例如在WHERE子句、RETURNING子句或在SET子句右侧的表达式中）。这种情况下，正被更新的关系上也需要SELECT权限，并且除了UPDATE策略外，也要应用适当的SELECT或者ALL策略。这样，除由UPDATE或ALL策略授权更新行之外，通过SELECT或ALL策略用也必须能访问正被更新的行。

当INSERT命令附加了ON CONFLICT DO UPDATE子句时，如果采用UPDATE路径，先以任何UPDATE策略的USING表达式检查待更新的行，然后以WITH CHECK表达式检查新修改的行。但要注意的是，不同于单独的UPDATE命令，如果现有的行不能通过USING表达式检查，则抛出错误（UPDATE路径永不会静默地避免）。

DELETE

为一条策略使用DELETE表示它适用于 DELETE命令。只有通过这条策略的行才将能被DELETE命令所看到。如果有的行不能通过该 DELETE策略的USING表达式，则 它们可以通过SELECT看到但不能被删除。

大多数情况下，DELETE命令也需要从其所删除的关系中的列读取数据（例如在WHERE子句或RETURNING子句中）。这种情况下，在该关系上也需要SELECT权限，并且除了DELETE策略，也要应用适当的SELECT或ALL策略。这样，除由DELETE或ALL策略授权删除行之外，通过SELECT或ALL策略，用户也必须能访问正被删除的行。

DELETE策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中删除记录的情况， 所以没有新行需要检查。

表 240. 按命令类型应用的策略

命令	SELECT/ALL策略	INSERT/ALL策略	UPDATE/ALL策略		DELETE/ALL策略
	USING表达式	WITH CHECK表达式	USING表达式	WITH CHECK表达式	USING表达式
SELECT	现有行	—	—	—	—
SELECT FOR UPDATE/SHARE	现有行	—	现有行	—	—
INSERT	—	新行	—	—	—
INSERT ... RETURNING	新行[a]	新行	—	—	—
UPDATE	现有 & 新行 [a]	—	现有行	新行	—
DELETE	现有行[a]	—	—	—	现有行
ON CONFLICT DO UPDATE	现有 & 新行	—	现有行	新行	—

[a] 对于现有行或新行，如果需要读访问的话（例如涉及到关系内列的WHERE或RETURNING子句）。

多重策略的应用

当多种不同命令类型的策略应用于相同命令（例如SELECT和UPDATE策略应用于UPDATE命令）时，用户就必须同时具有这两种类型的权限（例如从关系中选取行和更新的权限）。这样一种策略类型的表达式就与另一种策略类型的表达式通过使用AND操作符组合在一起。

当相同命令类型的多种策略应用于同一命令时，则必须至少有一个PERMISSIVE策略授权对该关系的访问，所有的RESTRICTIVE策略必须通过。这样，所有的PERMISSIVE策略表达式都用OR来组合，所有的RESTRICTIVE策略表达式都用AND来组合，而结果用AND来组合。如果没有PERMISSIVE策略，则拒绝访问。

要注意的是，出于组合多种策略的目的，将ALL策略视为与所应用的任何其他类型的策略具有相同的类型。

例如，在UPDATE命令中，SELECT和UPDATE两种权限都需要，如果每种类型都有多个适用的策略，则将之以下面的方式组合：

```
expression from RESTRICTIVE SELECT/ALL policy 1
AND
expression from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE SELECT/ALL policy 1
  OR
  expression from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
expression from RESTRICTIVE UPDATE/ALL policy 1
AND
expression from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE UPDATE/ALL policy 1
  OR
  expression from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)
```

注解

要为一个表创建或者修改策略，你必须是该表的拥有者。

虽然策略将被应用于针对数据库中表的显式查询上，但当系统正在执行　内部引用完整性检查或者验证约束时不会应用它们。这意味着有间接的　方法来决定一个给定的值是否存在。一个例子是向一个作为主键或者拥　有唯一约束的列中尝试插入重复值。如果插入失败则用户可以推导出该 值已经存在（这个例子假设用户被策略允许插入他们看不到的记录）。 另一个例子是一个用户被允许向一个引用了其他表的表中插入，然而另　一个表是隐藏表。通过用户向引用表中插入值可以判断存在性，　成功表示该值存在于被引用表中。为了解决这些问题，应该仔细地制作　策略以完全阻止用户插入、删除或者更新那些可能指示他们不能看到的 值的记录，或者使用生成的值（例如代理键）来代替具有外部含义的键。

通常，系统将在应用用户查询中出现的条件之前先强制由安全性策略施　加的过滤条件，这是为了防止无意中把受保护的数据暴露给可能不可信　的用户定义函数。不过，被系统（或者系统管理员）标记为 LEAKPROOF的函数和操作符可以在策略表达式之前 被计算，因为它们已经被假定为可信。

因为策略表达式会被直接加到用户查询上，它们将使用运行整个查询的用户的　权限运行。因此，使用一条给定策略的用户必须能够访问表达式中引用的任何　表或函数，否则在尝试查询启用了行级安全性的表时，他们将简单地收到一条　没有权限的错误。不过，这不会改变视图的工作方式。就普通查询和视图来说，　权限检查和视图所引用的表的策略将使用视图拥有者的权限以及任何适用于视 图拥有者的策略。

在第　5.7　节可以找到额外的讨论和实际的例子。

兼容性

CREATE POLICY是一种PostgreSQL扩展。

另见

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PROCEDURE

CREATE PROCEDURE — 定义一个新的过程

大纲

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

简介

CREATE PROCEDURE定义一个新的过程。CREATE OR REPLACE PROCEDURE将会创建一个新过程或者替换一个已有的定义。为了能够定义过程，用户必须具有所使用的语言上的USAGE特权。

如果这个命令中包括了一个方案名称，则该过程将被创建在该方案中。否则过程将被创建在当前的方案中。新过程的名称不能匹配同一方案中具有相同输入参数类型的任何现有过程或函数。不过，具有不同参数类型的过程和函数可以共享同一个名称（这被称为重载）。

要替换一个已有过程的当前定义，请使用CREATE OR REPLACE PROCEDURE。不能用这种方式更改过程的名称或者参数类型（如果尝试这样做，实际上会创建一个新的、不同的过程）。

当CREATE OR REPLACE PROCEDURE被用来替换一个现有的过程时，该过程的拥有关系和权限保持不变。所有其他的过程属性会被赋予这个命令中指定的或者暗示的值。必须拥有（包括成为拥有角色的成员）该过程才能替换它。

创建过程的用户将成为该过程的拥有者。

为了能够创建一个过程，用户必须具有参数类型上的USAGE特权。

Parameters

name

要创建的过程的名称（可以是被方案限定的）。

argmode

参数的模式可以是：IN、INOUT或者VARIADIC。如果省略，则默认为IN（当前对过程不支持OUT参数，可使用INOUT）。

argname

参数的名称。

argtype

过程的参数（如果有）的数据类型（可以是被方案限定的）。参数类型可以是基础类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据具体的实现语言，还可能可以指定“伪类型”，例如cstring。伪类型表示实际的参数类型没有完全确定，或者是位于普通SQL数据类型的集合之外。

写上table_name.column_name%TYPE可以引用某个列的类型。使用这种特性有时可以让过程不受表定义改变的影响。

default_expr

没有指定参数时要被用作默认值的表达式。这个表达式必须符合该参数的参数类型。跟在有默认值的参数后面的输入参数也都必须有默认值。

lang_name

用于实现该过程的语言名称。它可以是sql、c、internal或者一种用户定义的过程语言的名称，例如plpgsql。将名称包裹在单引号内的方式已经被废弃，并且要求大小写匹配。

TRANSFORM { FOR TYPE type_name } [, ...] }

列出对过程的调用应该应用哪些Transform。Transform负责在SQL类型和语言相关的数据类型之间进行转换，请参考CREATE TRANSFORM。过程语言实现通常采用硬编码的方式保存内建类型的知识，因此它们无需在这里列出。但如果一种过程语言实现不知道如何处理一种类型并且没有提供Transform，它将回退到默认的行为来转换数据类型，但是这依赖于其实现。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER指示过程以调用它的用户的特权来执行。这是默认方式。SECURITY DEFINER指定过程以拥有它的用户的特权来执行。

为了符合SQL标注，允许使用EXTERNAL关键词，但它是可选的，因为和SQL中不同，这个特性适用于所有的过程而不仅仅是外部过程。

SECURITY DEFINER过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

configuration_parameter
value

SET子句导致在进入该过程时指定的配置参数被设置为指定的值，并且在过程退出时恢复到之前的值。SET FROM CURRENT把CREATE PROCEDURE执行时该参数的当前值保存为在进入该过程时要应用的值。

如果对过程附加一个SET子句，那么在该过程中为同一个变量执行的SET LOCAL命令的效果就被限制于该过程：在过程退出时还是会恢复到该配置参数的以前的值。不过，一个普通的SET命令（没有LOCAL）会重载这个SET子句，很像它对一个之前的SET LOCAL命令所做的事情：这样一个命令的效果将持续到过程退出之后，除非当前事务被回滚。

如果对过程附加一个SET子句，则该过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

有关允许的参数名和值的更多信息请参考SET和第 19 章

definition

一个定义该过程的字符串常量，其含义取决于语言。它可以是一个内部的过程名、一个对象文件的路径、一个SQL命令或者以一种过程语言编写的文本。

在编写过程的定义字符串时，使用美元引用（见第 4.1.2.4 节）而不是普通的单引号语法常常会很有帮助。如果没有美元引用，过程定义中的任何单引号或者反斜线必须以双写的方式进行转义。

obj_file, link_symbol

当C语言源码中的过程名与SQL过程的名称不同时，这种形式的AS子句被用于动态可装载的C语言过程。字符串obj_file是包含已编译好的C过程的共享库文件名，并且被按照LOAD命令的方式解析。字符串link_symbol是该过程的链接符号，也就是该过程在C语言源代码中的名称。如果链接符号被省略，则会被假定为与正在被定义的SQL过程的名称相同。

当重复的CREATE PROCEDURE调用引用同一个对象文件时，只会对每一个会话装载该文件一次。要卸载或者重新载入该文件（可能是在开发期间），应该开始一个新的会话。

注解

函数创建也适用于过程，更多细节请参考CREATE FUNCTION。

使用CALL来执行过程。

示例

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

兼容性

SQL标准中定义有一个CREATE PROCEDURE命令。PostgreSQL的版本类似但是并不完全兼容。详情请见CREATE FUNCTION。

另见

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

CREATE PUBLICATION

CREATE PUBLICATION — 定义一个新的发布

大纲

```
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

描述

CREATE PUBLICATION向当前数据库添加一个新的发布。 发布的名称必须与当前数据库中任何现有发布的名称不同。

发布本质上是一组表，其数据更改旨在通过逻辑复制进行复制。 有关发布如何适应逻辑复制设置的详细信息， 请参见第 31.1 节

参数

name

新发布的名称。

FOR TABLE

指定要添加到发布的表的列表。如果在表名之前指定了ONLY， 那么只有该表被添加到发布中。如果没有指定ONLY， 则添加表及其所有后代表（如果有的话）。可选地， 可以在表名之后指定*以明确指示包含后代表。

只有持久基表才能成为出版物的一部分。临时表、非日志表、外表、物化视图、 常规视图和分区表不能成为发布的一部分。要复制分区表， 请将各个分区添加到发布中。

FOR ALL TABLES

将发布标记为复制数据库中所有表的更改，包括在将来创建的表。

WITH (publication_parameter [= value] [, ...])

该子句指定发布的可选参数。支持下列参数：

publish (string)

这个参数决定了哪些DML操作将由新的发布发布给订阅者。 该值是用逗号分隔的操作列表。允许的操作是insert， update，delete和truncate。 默认是发布所有的动作，所以这个选项的默认值是 'insert, update, delete, truncate'。

注意

如果既没有指定FOR TABLE，也没有指定FOR ALL TABLES， 那么这个发布就是以一组空表开始的。这在稍后添加表格的情况下是有用的。

创建发布不会开始复制。它只为未来的订阅者定义一个分组和过滤逻辑。

要创建一个发布，调用者必须拥有当前数据库的CREATE权限。 （当然，超级用户不需要这个检查。）

要将表添加到发布中，调用者必须拥有该表的所有权。FOR ALL TABLES 子句要求调用者是超级用户。

添加到发布UPDATE和/或DELETE 操作的发布的表必须已经定义了REPLICA IDENTITY。 否则将在这些表上禁止这些操作。

对于INSERT ... ON CONFLICT命令， 发布将公布从命令实际产生的操作。因此，根据结果，它可以作为 INSERT或UPDATE发布，也可以根本不发布。

COPY ... FROM命令是作为INSERT操作发布的。

不发布DDL操作。

示例

创建一个发布，发布两个表中所有更改布：

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

创建一个发布，发布所有表中的所有更改：

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

创建一个发布，只发布一个表中的INSERT操作：

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

兼容性

CREATE PUBLICATION是一个PostgreSQL 扩展。

又见

ALTER PUBLICATION, DROP PUBLICATION

CREATE ROLE

CREATE ROLE — 定义一个新的数据库角色

大纲

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option可以是：

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

CREATE ROLE向PostgreSQL数据库集簇增加一个新的角色。一个角色是一个实体，它可以拥有数据库对象并且拥有数据库特权。根据一个角色如何被使用，它可以被考虑成一个“用户”、一个“组”或者两者。有关管理用户和认证的信息可以参考第 21 章和第 20 章要使用这个命令，你必须具有CREATEROLE特权或者成为一个数据库超级用户。

注意角色是定义在数据库集簇层面上的，并且因此在集簇中的所有数据库中都可用。

参数

name

新角色的名称。

SUPERUSER
NOSUPERUSER

这些子句决定新角色是否是一个“超级用户”，它可以越过数据库内的所有访问限制。超级用户状态很危险并且只应该在确实需要时才用。要创建一个新超级用户，你必须自己是一个超级用户。如果没有指定，默认值是NOSUPERUSER。

CREATEDB
NOCREATEDB

这些子句定义一个角色创建数据库的能力。如果指定了CREATEDB，被定义的角色将被允许创建新的数据库。指定NOCREATEDB将否定一个角色创建数据库的能力。如果没有指定，默认值是NOCREATEDB。

CREATEROLE
NOCREATEROLE

这些子句决定一个角色是否被允许创建新的角色（也就是执行CREATE ROLE）。一个带有CREATEROLE特权的角色也能修改和删除其他角色。如果没有指定，默认值是NOCREATEROLE。

INHERIT
NOINHERIT

如果新的角色是其他角色的成员，这些子句决定新角色是否从那些角色中“继承”特权，把新角色作为成员的角色称为新角色的父角色。一个带有INHERIT属性的角色能够自动使用已经被授予给其直接或间接父角色的任何数据库特权。如果没有INHERIT，在另一个角色中的成员关系只会把SET ROLE的能力授予给那个其他角色，只有在这样做后那个其他角色的特权才可用。如果没有指定，默认值是INHERIT。

LOGIN
NOLOGIN

这些子句决定一个角色是否被允许登录，也就是在客户端连接期间该角色是否能被给定为初始会话认证名称。一个具有LOGIN属性的角色可以被考虑为一个用户。没有这个属性的角色对于管理数据库特权很有用，但是却不是用户这个词的通常意义。如果没有指定，默认值是NOLOGIN，不过当CREATE ROLE被通过CREATE USER调用时默认值会是LOGIN。

REPLICATION
NOREPLICATION

这些子句决定一个角色是否为复制角色。角色必须具有这个属性（或者成为一个超级用户）才能以复制模式（物理复制或者逻辑复制）连接到服务器以及创建或者删除复制槽。一个具有REPLICATION属性的角色是一个具有非常高特权的角色，并且只应被用于确实需要复制的角色上。如果没有指定，默认值是NOREPLICATION。

BYPASSRLS
NOBYPASSRLS

这些子句决定是否一个角色可以绕过每一条行级安全性（RLS）策略。 默认是NOBYPASSRLS。注意 pg_dump 将默认 把row_security设置为OFF， 以确保一个表的所有内容被转储出来。如果运行 pg_dump 的用户不 具有适当的权限，将会返回一个错误。超级用户和被转储表的拥有者 总是可以绕过 RLS。

CONNECTION LIMIT connlimit

如果角色能登录，这指定该角色能建立多少并发连接。-1（默认值）表示无限制。注意这个限制仅针对于普通连接。预备事务和后台工作者连接都不受这一限制管辖。

[ENCRYPTED] PASSWORD 'password'
PASSWORD NULL

设置角色的口令（口令只对具有LOGIN属性的角色有用，但是不管怎样你还是可以为没有该属性的角色定义一个口令）。如果你没有计划使用口令认证，你可以忽略这个选项。如果没有指定口令，口令将被设置为空并且该用户的口令认证总是会失败。也可以用PASSWORD NULL显式地写出一个空口令。

> **注意**
>
> 指定一个空字符串也将把口令设置为空，但是在PostgreSQL版本10之前是不能这样做的。在早期的版本中，是否可以使用空字符串取决于认证方法和确切的版本，而libpq在任何情况下都将拒绝使用空字符串。为了避免混淆，应该避免指定空字符串。

口令总是以加密的方式存放在系统目录中。ENCRYPTED关键词没有实际效果，它只是为了向后兼容性而存在。加密的方法由配置参数password_encryption决定。如果当前的口令字符串已经是MD5加密或者SCRAM加密的格式，那么不管password_encryption的值是什么，口令字符串还是原样存储（因为系统无法解密以不同格式加密的口令字符串）。这种方式允许在转储/恢复时重载加密的口令。

VALID UNTIL 'timestamp'

VALID UNTIL机制设置一个日期和时间，在该时间点之后角色的口令将会失效。如果这个子句被忽略，那么口令将总是有效。

IN ROLE role_name

IN ROLE子句列出一个或多个现有的角色，新角色将被立即作为新成员加入到这些角色中（注意没有选项可以把新角色作为一个管理员加入，需要用一个单独的GRANT命令来完成）。

IN GROUP role_name

IN GROUP是IN ROLE的一种已废弃的拼写方式。

ROLE role_name

ROLE子句列出一个或者多个现有角色，它们会被自动作为成员加入到新角色中（这实际上新角色变成了一个“组”）。

ADMIN role_name

ADMIN子句与ROLE相似，但是被提及的角色被使用WITH ADMIN OPTION加入到新角色中，让它们能够把这个角色中的成员关系授予给其他人。

USER role_name

USER子句是ROLE子句的一个已废弃的拼写方式。

SYSID uid

SYSID子句会被忽略，但是会为了向后兼容，还是会接受它。

注解

使用ALTER ROLE来更改一个角色的属性，以及使用DROP ROLE来移除一个角色。所有用CREATE ROLE指定的属性可以被后来的ALTER ROLE命令所修改。

增加和移除组角色成员的最佳方式是使用GRANT和REVOKE。

VALID UNTIL子句只为一个口令而不是为一个角色本身定义了一个过期时间。特别地，当使用一个非基于口令认证的方法登录时，过期时间是不会被强制的。

INHERIT属性管理可授予特权（也就是对数据库对象和角色成员关系的访问特权）的继承性。它并不适用于由CREATE ROLE和ALTER ROLE设置的特殊角色属性。例如，作为具有CREATEDB特权的角色的一个成员，并不会立刻授予创建数据库的角色，即便INHERIT被设置也是如此，在创建一个数据库之前必须通过SET ROLE成为该角色。

INHERIT属性是用于向后兼容原因的默认值：在早前的PostgreSQL发布中，用户总是能够访问其所属组的所有特权。不过，NOINHERIT更加接近于 SQL 标准中指定的语义。

要小心CREATEROLE特权。对于一个CREATEROLE角色的特权没有继承的概念。那意味着即使一个角色没有特定的特权但被允许创建其他角色，它可以轻易地创建与自身特权不同的另一个角色（除了创建具有超级用户特权的角色）。例如，如果角色“user”具有CREATEROLE特权

但是没有CREATEDB特权，但是它能够创建一个带有CREATEDB特权的新角色。因此，可以把具有CREATEROLE特权的角色看成是准超级用户角色。

PostgreSQL包括一个程序createuser，它具有和CREATE ROLE相同的功能（事实上，它会调用这个命令），但是它可以从命令 shell 中运行。

CONNECTION LIMIT只被近似地强制，如果两个新会话在几乎相同时间被开始，而此时该角色只剩下刚好一个连接“槽”，两者可能都将失败。还有，该限制从不对超级用户强制。

用这个命令指定一个非加密口令时必须加以注意。该命令将被以明文的形式传输到服务器，并且它也可能被记录在客户端命令历史或者服务器日志中。不过，命令createuser会传输加密的口令。还有，psql包含一个命令\password，它可以被用来安全地改变该口令。

例子

创建一个能登录但是没有口令的角色：

```
CREATE ROLE jonathan LOGIN;
```

创建一个有口令的角色：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

（CREATE USER和CREATE ROLE完全相同，除了它带有LOGIN）。

创建一个角色，它的口令有效期截止到 2004 年底。在进入 2005 年第一秒时，该口令会失效。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

创建一个能够创建数据库并且管理角色的角色：

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

兼容性

SQL 标准中有CREATE ROLE语句，但是标准只要求语法

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

多个初始管理员以及CREATE ROLE的所有其他选项都是PostgreSQL扩展。

SQL 标准定义了用户和角色的概念，但是它把它们看成是可区分的概念并且将定义用户的所有命令留给每一种数据库实现来指定。在PostgreSQL中，我们已经选择把用户和角色统一成一种单一实体类型。因此角色比起标准中拥有更多可选的属性。

SQL 标准指定的行为可以通过给用户NOINHERIT属性来得到最大的金丝，而角色会被给予INHERIT属性。

参见

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

CREATE RULE — 定义一条新的重写规则

大纲

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

其中 event 可以是以下之一：

```
    SELECT | INSERT | UPDATE | DELETE
```

描述

CREATE RULE定义一条应用于指定表或视图的 新规则。CREATE OR REPLACE RULE将创建一条新规则或者替换同一个表上具有同一名称的现有规则。

PostgreSQL规则系统允许我们定义 针对数据库表中插入、更新或者删除动作上的替代动作。大约来说，当在 一个给定表上执行给定命令时，一条规则会导致执行额外的命令。或者， INSTEAD规则可以用另一个命令替换给定的命令，或 者导致一个命令根本不被执行。规则也被用来实现 SQL 视图。规则实际上 是一种命令转换机制或者命令宏。这种转换会在命令的执行开始之前进行。 如果你实际上想要为每一个物理行独立地触发一个操作，你可能更需要一个 触发器而不是规则。更多有关规则系统的信息请见第 41 章

当前，ON SELECT规则必须是无条件 INSTEAD规则并且其动作必须由一个单一 SELECT命令构成。因此，一条 ON SELECT规则实际上把表变成了一个视图，它的可见 内容是由该规则的SELECT命令返回，而不是直 接存在该表中的内容（如果有）。不过，使用 CREATE VIEW命令还是要比创建一个真实表并且 在其上定义一条ON SELECT规则更好。

可以通过定义ON INSERT、ON UPDATE 以及ON DELETE规则（或者这些规则的任意子集）来创建 可更新的视图，这些规则可以把视图上的更新动作替换为其他表上适当的更新 动作。如果想要支持INSERT RETURNING等等，那么一定要 在每一个这类规则中放上一个合适的RETURNING子句。

如果你尝试为复杂视图更新使用有条件的规则，有一点是很重要的：对于 你希望在该视图上允许的每一个动作，必须有一条 INSTEAD规则。如果该规则是有条件的，或者不是INSTEAD，那么系统仍将拒绝尝试执行该更新动作， 因为它会认为在某些情况下它应该停止尝试在该视图的傀儡表上执行动作。 如果你想处理有条件规则中的所有有用的情况，可以增加一条无条件的 DO INSTEAD NOTHING规则来确保系统理解它将 永远不会被调用来更新傀儡表。然后让有条件规则变成 非-INSTEAD。在它们适用的情况下，它们会加到 默认的INSTEAD NOTHING动作（不过，当前这种方法不 支持RETURNING查询）。

> **注意**
>
> 足够简单的视图自动就是可更新的（见CREATE VIEW），它们不需要依靠用户创建的规则来变成可 更新的。不过还是可以创建一条显式规则，自动更新转换通常比显式规则效 率高。
>
> 另一种值得考虑的办法是使用INSTEAD OF触发器（见 CREATE TRIGGER）代替规则。

参数

name

要创建的规则的名称。它必须与同一个表上任何其他规则的名称相区分。 同一个表上同一种事件类型的多条规则会按照其名称的字符顺序被应用。

event

时间是SELECT、 INSERT、UPDATE或者 DELETE之一。 注意包含ON CONFLICT子句的INSERT不能被用在具有INSERT或者 UPDATE规则的表上。那种情况下请考虑使用 可更新的视图。

table_name

规则适用的表或者视图的名称（可以是模式限定的）。

condition

任意的SQL条件表达式（返回 boolean）。该条件表达式不能引用除NEW以及 OLD之外的任何表，并且不能包含聚集函数。

INSTEAD

INSTEAD指示该命令应该取代 原始命令被执行。

ALSO

ALSO指示应该在原始命令 之外执行这些命令。

如果ALSO和INSTEAD都没有被指定， 默认是ALSO。

command

组成规则动作的命令。可用的命令有SELECT、 INSERT、UPDATE、 DELETE或者NOTIFY。

在condition和 command中，名为 NEW和OLD的表可以被用来引用被 引用表中的值。在ON INSERT和 ON UPDATE规则中，NEW被用来 引用被插入或者更新的新行。在ON UPDATE和 ON DELETE规则中，OLD被用来引 用被更新或者删除的现有行。

注解

要在表上创建或者修改规则，必须是表的拥有者。

在一条用于视图上INSERT、UPDATE 或者DELETE的规则中， 可以增加一个RETURNING 子句来发出视图的列。如果该规则被一个INSERT RETURNING、 UPDATE RETURNING或者DELETE RETURNING命令触发，这个子句将被用来计算输出。 当规则被一个没有RETURNING的命令触发时，该规则的 RETURNING子句将被忽略。当前的实现只允许无条件 INSTEAD规则包含RETURNING。此外，用于同一事件 的所有规则中至多只能有一个RETURNING子句（这确保了只有一个 候选RETURNING子句被用来计算结果）。如果在任何可用规则中都 没有RETURNING子句，视图上的RETURNING查询将 被拒绝。

避免循环规则非常重要。例如，尽管下面的两条规则定义都被 PostgreSQL所接受，SELECT命令将导致 PostgreSQL报告一个错误，因为会产生一条 规则的递归扩展：

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;
```

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

当前，如果一个规则动作包含一个NOTIFY命令，　　该NOTIFY命令将被无条件执行，也就是说，即使 该规则不被应用到任何行上，也会发出NOTIFY。 例如，在

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

中，UPDATE期间将发出一个 NOTIFY事件，不管是否有行匹配条件 id = 42。这是一种实现限制，它可能会在未来的发行中被修复。

兼容性

CREATE RULE是一种 PostgreSQL语言扩展， 整个查询重写系统也是这样。

另见

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — 定义一个新模式

大纲

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element
 [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification
```

其中 role_specification 可以是：

```
    user_name
  | CURRENT_USER
  | SESSION_USER
```

描述

CREATE SCHEMA输入一个新模式到当前数据库中。 该模式名必须与当前数据库中任何现有模式的名称不同。

一个模式本质上是一个名字空间：它包含命令对象（表、数据类型、函数以及操作符），对象可以与在其他模式中存在的对象重名。可以通过用模式名作为一个前缀 “限定”命名对象的名称来访问它们，或者通过把要求的模式包括 在搜索路径中来访问命名对象。一个指定非限定对象名的 CREATE命令在当前模式（搜索路径中的第一个模式，由函数current_schema决定）中创建对象。

CREATE SCHEMA中可以选择包括子命令用以在 新模式中创建对象。这些子命令实际被当做独立的在创建该模式后被发出的命令 一样，除非使用AUTHORIZATION子句，所有被创建的对象都会 由该用户拥有。

参数

schema_name

要创建的一个模式名。如果省略， user_name将被用作模式名。 该名称不能以pg_开始，因为这样的名称是用作系统模式的。

user_name

将拥有新模式的用户的角色名。如果省略，默认为执行该命令的用户。要 创建由另一个角色拥有的角色，你必须是那个角色的一个直接或者间接成员， 或者是一个超级用户。

schema_element

要在该模式中创建的对象的定义 SQL 语句。当前，只有CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER以及GRANT被接受为 CREATE SCHEMA中的子句。其他类型的对象可以在模式被 创建之后用单独的命令创建。

IF NOT EXISTS

如果一个具有同名的模式已经存在，则什么也不做（不过发出一个提示）。 使用这个选项时不能包括 schema_element子命令。

注解

要创建一个模式，调用用户必须拥有当前数据库的CREATE 特权（当然，超级用户可以绕过这种检查）。

示例

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户joe创建一个模式，该模式也将被命名为 joe：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个被用户joe拥有的名为test的模式， 除非已经有一个名为test的模式（不管joe 是否拥有该已经存在的模式）。

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

创建一个模式并且在其中创建一个表和视图：

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

注意子命令不以分号结束。

下面是达到相同结果的等效的方法：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

兼容性

SQL 标准允许在CREATE SCHEMA中有一个 DEFAULT CHARACTER SET子句，以及当前 PostgreSQL接受的更多子命令类型。

SQL 标准制定CREATE SCHEMA中的子命令 可以以任何顺序出现。当前的 PostgreSQL实现不能处理子命令中 所有情况的向前引用。有时候可能有必要对子命令进行重排序以避免向前 引用。

根据 SQL 标准，模式的拥有者总是拥有其中的所有对象。 PostgreSQL允许模式包含非模式 拥有者所拥有的对象。只有模式拥有者把其模式上的CREATE 特权授予给了其他人或者一个超级用户选择在该模式中创建对象时才会 发生这种事情。

IF NOT EXISTS选项是一种 PostgreSQL扩展。

另见

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — 定义一个新的序列发生器

大纲

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name [ INCREMENT
 [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

描述

CREATE SEQUENCE创建一个新的序列数 发生器。这涉及到用名称name创建并且初始化 一个新的特殊的单行表。该发生器将由发出该命令的用户所拥有。

如果给出一个模式名称，则该序列将将被创建在指定的模式中。否则它会被 创建在当前模式中。临时序列存在于一个特殊的模式中，因此在创建临时序列 时不能给出模式名。序列名称必须与同一模式中任何其他序列、表、索引、 视图或者外部表的名称不同。

在序列被创建后，可以使用函数 nextval、 currval以及 setval来操作该序列。这些函数在第 9.16 节有介绍。

尽管无法直接更新一个序列，可以使用这样的查询：

```
SELECT * FROM name;
```

来检查一个序列的参数以及当前状态。特别地，序列的 last_value域显示被任意会话最后一次取得的值（当然， 在被打印时该值可能已经过时了，因为可能有其他会话正在执行nextval调用）。

参数

TEMPORARY or TEMP

如果被指定，只会为这个会话创建序列对象，并且在会话退出时自动 删除它。当临时序列存在时，已有的同名永久序列（在这个会话中） 会变得不可见，不过可以用模式限定的名称来引用同名永久序列。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的关系与即将创建的序列相似 -- 它甚至可能 都不是一个序列。

name

要创建的序列的名称（可以是模式限定的）。

data_type

可选的子句AS data_type 制定序列的数据类型。有效类型是 smallint、integer、和bigint。默认是bigint。 数据类型决定了序列的默认最小和最大值。

increment

可选的子句INCREMENT BY increment指定为了 创建新值会把哪个值加到当前序列值上。一个正值将会创造一个上升 序列，负值会创造一个下降序列。默认值是 1。

minvalue
NO MINVALUE

可选的子句MINVALUE minvalue决定一个序列 能产生的最小值。如果没有提供这个子句或者指定了 NO MINVALUE，那么会使用默认值。 升序序列的默认值为1。降序序列的默认值为数据类型的最小值。

maxvalue
NO MAXVALUE

可选的子句MAXVALUE maxvalue决定该序列 的最大值。如果没有提供这个子句或者指定了 NO MAXVALUE，那么将会使用默认值。 升序序列的默认值是数据类型的最大值。降序序列的默认值是-1。

start

可选的子句START WITH start 允许序列从任何 地方开始。对于上升序列和下降序列来说，默认的开始值分别是 minvalue和 maxvalue。

cache

可选的子句CACHE cache指定要预分配多少 个序列数并且把它们放在内存中以便快速访问。最小值为 1 （一次只生成 一个值，即没有缓存），默认值也是 1。

CYCLE
NO CYCLE

对于上升序列和下降序列，CYCLE选项允许序列 在分别达到maxvalue和minvalue时回卷。如果达到 该限制，下一个产生的数字将分别是minvalue和maxvalue。

如果指定了NO CYCLE，当序列到达其最大值 后任何nextval调用将返回一个错误。如果CYCLE和NO CYCLE都没有 被指定，则默认为NO CYCLE。

OWNED BY table_name.column_name
OWNED BY NONE

OWNED BY选项导致序列被与一个特定的表列关联 在一起，这样如果该列（或者整个表）被删除，该序列也将被自动删除。 指定的表必须和序列具有相同的拥有者并且在同一个模式中。默认选项 OWNED BY NONE指定该序列不与某个列关联。

注解

使用DROP SEQUENCE移除一个序列。

序列是基于bigint算法的，因此范围是不能超过一个八字节 整数的范围（-9223372036854775808 到 9223372036854775807）。

由于nextval和setval调用绝不会回滚， 如果需要序数的“无间隙”分配，则不能使用序列对象。可以 通过在一个只包含一个计数器的表上使用排他锁来构建无间隙的分配， 但是这种方案比序列对象开销更大，特别是当有很多事务并发请求序数 时。

如果对一个将由多个会话并发使用的序列对象使用了大于 1 的cache设置，可能会得到意想不到的结果。 每个会话会在访问该序列对象时分配并且缓存后续的序列值，并且相应地增加 该序列对象的last_value。然后，在该会话中下一次 nextval会做 cache-1，并且简单地 返回预分配的值而不修改序列对象。因此，任何已分配但没有在会话中使用的 数字将会在该会话结束时丢失，导致该序列中的“空洞”。

进一步，尽管多个会话能分配到不同的序列值，这些值可能会在所有会话都被 考虑时生成出来。例如， cache的设置为 10，会话 A 可能储存值 1..10 并且返回nextval=1，然后会话 B 可能储存值 11..20 并且在 A 生成nextval=2 之前返回 nextval=11。因此，如果cache设置为 1，可以 安全地假设nextval值被顺序地生成。如果cache设置大于 1，就只能假定 nextval值都是可区分的，但不能保证它们被完全地顺序生成。 还有，last_value将反映服务于任意会话的最后一个值，不管它 是否已经被nextval返回过。

另一个考虑是，在这样一个序列上执行的setval将不会通知 其他会话，直到它们用尽了任何已缓存的预分配值。

示例

创建一个称作serial的上升序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从这个序列中选取下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     101
```

再从这个序列中选取下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     102
```

在一个INSERT命令中使用这个序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一次COPY FROM后更新新列值：

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

兼容性

CREATE SEQUENCE符合SQL 标准，不过下列除外：

- 使用nextval()而不是标准的NEXT VALUE FOR 表达式获取下一个值。
- OWNED BY子句是一种PostgreSQL扩展。

另见

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — 定义一个新的外部服务器

大纲

```
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION
 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

CREATE SERVER定义一个新的外部服务器。 定义该服务器的用户会成为拥有者。

外部服务器通常包装了外部数据包装器用来访问一个外部数据源所需的 连接信息。额外的用户相关的连接信息可以通过用户映射的方式来指定。

服务器名称在数据库中必须唯一。

创建服务器要求所使用的外部数据包装器上的USAGE特权。

参数

IF NOT EXISTS

如果已经存在同名的服务器，不要抛出错误。在这种情况下发出一个通知。 请注意，不能保证现有服务器与要创建的服务器类似。

server_name

要创建的外部服务器的名称。

server_type

可选的服务器类型，可能对外部数据包装器有用。

server_version

可选的服务器版本，可能对外部数据包装器有用。

fdw_name

管理该服务器的外部数据包装器的名称。

OPTIONS (option 'value' [, ...])

这个子句为服务器指定选项。这些选项通常定义该服务器的连接细节， 但是实际的名称和值取决于该服务器的外部数据包装器。

注解

在使用dblink模块时，一个外部服务器的名称可以被 用作dblink_connect函数的一个参数来指示 连接参数。以这种方式使用外部服务器，需要在其上具有 USAGE特权。

示例

创建使用外部数据包装器postgres_fdw 的服务器myserver：

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo',
 dbname 'foodb', port '5432');
```

详见postgres_fdw。

兼容性

CREATE SERVER符合 ISO/IEC 9075-9 (SQL/MED)。

另见

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — 定义扩展统计

大纲

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON column_name, column_name [, ...]
    FROM table_name
```

描述

CREATE STATISTICS将创建一个新的扩展统计对象， 追踪指定表、外部表或物化视图的数据。该统计对象将在当前数据库中创建， 被发出该命令的用户所有。

如果给定了模式名（比如，CREATE STATISTICS myschema.mystat ...）， 那么在给定的模式中创建统计对象。否则在当前模式中创建。 统计对象的名称必须与相同模式中的任何其他统计对象不同。

参数

IF NOT EXISTS

如果具有相同名称的统计对象已经存在，不会抛出一个错误，只会发出一个提示。 请注意，这里只考虑统计对象的名称，不考虑其定义细节。

statistics_name

要创建的统计对象的名称（可以有模式限定）。

statistics_kind

在此统计对象中计算的统计种类。目前支持的种类是启用n-distinct统计的ndistinct，以及启用功能依赖性统计的dependencies。 如果省略该子句，则统计对象中将包含所有支持的统计类型。 有关更多信息，请参阅第 14.2.2 和 第 70.2 节

column_name

被计算的统计信息包含的表格列的名称。至少必须给出两个列名。

table_name

包含计算统计信息的列的表的名称（可以是模式限定的）。

注意

你必须是表的所有者才能创建读取它的统计对象。不过，一旦创建， 统计对象的所有权与基础表无关。

示例

用两个功能相关的列创建表t1， 即第一列中的值的信息足以确定另一列中的值。然后， 在这些列上构建函数依赖关系统计信息：

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- 匹配行的数量将被大大低估：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- 现在行计数估计会更准确：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

如果没有函数依赖性统计，规划器会认为两个WHERE条件是独立的，　　并且会将它们的选择性乘以一起，以致得到太小的行数估计。　通过这样的统计，规划器认识到WHERE条件是多余的，并且不会低估行数。

兼容性

SQL标准中没有CREATE STATISTICS命令。

又见

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — 定义一个新的订阅

大纲

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

描述

CREATE SUBSCRIPTION为当前数据库添加一个新的订阅。 订阅名称必须与数据库中任何现有的订阅不同。

订阅表示到发布者的复制连接。因此，此命令不仅在本地目录中添加定义， 还会在发布者上创建复制插槽。

在运行此命令的事务提交时，将启动逻辑复制工作器以复制新订阅的数据。

关于订阅和逻辑复制的更多信息，请参见 第 31.2 和 第 31 章

参数

subscription_name

新订阅的名称。

CONNECTION 'conninfo'

连接发布者的字符串。详细信息请见第 34.1.1 节

PUBLICATION publication_name

要订阅的发布者上的发布名称。

WITH (subscription_parameter [= value] [, ...])

该子句指定订阅的可选参数。支持的参数有：

copy_data (boolean)

指定在复制启动后是否应复制正在订阅的发布中的现有数据。 默认值是true。

create_slot (boolean)

指定该命令是否要在发布者上创建复制槽。默认值是true。

enabled (boolean)

指定订阅是否应该主动复制，或者是否应该只是设置，但尚未启动。 默认值是true。

slot_name (string)

要使用的复制插槽的名称。默认行为是使用订阅名称作为插槽的名称。

当slot_name设置为NONE时， 将不会有复制槽与订阅关联。这在需要稍后手动设置复制槽的情况下会使用。 这样的订阅必须同时enabled并且 create_slot设置为false。

synchronous_commit (enum)

该参数的值会覆盖synchronous_commit设置。 默认值是off。

对于逻辑复制使用off是安全的： 如果订阅者由于缺少同步而丢失事务，数据将从发布者重新发送。

进行同步逻辑复制时，不同的设置可能是合适的。 逻辑复制工作者向发布者报告写入和刷新的位置，当使用同步复制时， 发布者将等待实际刷新。这意味着，当订阅用于同步复制时， 将订阅者的synchronous_commit设置为off 可能会增加发布服务器上COMMIT的延迟。 在这种情况下，将synchronous_commit设置为 local或更高是有利的。

connect (boolean)

指定CREATE SUBSCRIPTION是否应该连接到发布者。 将其设置为false将会改变默认值enabled、 create_slot和copy_data为false。

不允许将connect设置为false的同时将 enabled、create_slot或copy_data 设置为true。

因为该选项设置为false时不会建立连接， 因此表没有被订阅，所以当启用订阅后，不会复制任何内容。 需要运行ALTER SUBSCRIPTION ... REFRESH PUBLICATION才能订阅表。

注意

有关如何在订阅和发布实例之间配置访问控制的详细信息，请参见 第 31.7 节

创建复制槽时(默认行为), CREATE SUBSCRIPTION不能在事务块内部执行。

如果复制插槽不是作为同一命令的一部分创建的，则创建连接到相同数据库集群的订阅（例如，在同一集群中的数据库之间进行复制或在同一个数据库中进行复制）只能成功 否则，CREATE SUBSCRIPTION调用将挂起。要做到这一点， 单独创建复制插槽（使用函数pg_create_logical_replication_slot 和插件名称pgoutput），并使用参数create_slot = false 创建订阅。这是一个实施限制，可能会在未来的版本中解除。

示例

创建一个到远程服务器的订阅，复制发布mypublication和 insert_only中的表，并在提交时立即开始复制：

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

创建一个到远程服务器的订阅，复制insert_only发布中的表， 并且不开始复制直到稍后启用复制。

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

兼容性

CREATE SUBSCRIPTION是一个PostgreSQL 扩展。

又见

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — 定义一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
 [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    OF type_name [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
 [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
 [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

其中 column_constraint 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
```

```
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
 SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

table_constraint 是:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
 [, ... ] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
 [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON
 UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

like_option 是:

{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | IDENTITY |
 INDEXES | STATISTICS | STORAGE | ALL }

partition_bound_spec 是:

IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE }
 [, ...] )
  TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE }
 [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的index_parameters是:

[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]

一个EXCLUDE约束中的exclude_element是:

{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ]
```

描述

CREATE TABLE将在当前数据库中创建一个新的、初始为空的表。该表将由发出该命令的用户所拥有。

如果给定了一个模式名（例如CREATE TABLE myschema.mytable ...），那么该表被创建在指定的模式中。否则它被创建在当前模式中。临时表存在于一个特殊的模式中，因此在创建一个临时表时不能给定一个模式名。该表的名称必须与同一个模式中的任何其他表、序列、索引、视图或外部表的名称区分开。

CREATE TABLE也会自动地创建一个数据类型来表示对应于该表一行的组合类型。因此，表不能用同一个模式中任何已有数据类型的名称。

可选的约束子句指定一个插入或更新操作要成功，新的或更新过的行必须满足的约束（测试）。一个约束是一个 SQL 对象，它帮助以多种方式定义表中的合法值集合。

有两种方式来定义约束：表约束和列约束。一个列约束会作为列定义的一部分定义。一个表约束定义不与一个特定列绑定，并且它可以包含多于一个列。每一个列约束也可以被写作一个表约束，列约束只是一种当约束只影响一列时方便书写的记号习惯。

要能创建一个表，你必须分别具有所有列类型或OF子句中类型的USAGE特权。

参数

TEMPORARY or TEMP

如果指定，该表被创建为一个临时表。临时表会被在会话结束时自动被删除，或者也可以选择在当前事务结束时删除（见下文的ON COMMIT）。当临时表存在时，已有的同名持久表将对于当前会话不可见，不过可以使用模式限定的名称进行引用。在一个临时表上创建的任何索引也自动地变为临时的。

自动清理守护进程不能访问并且因此也不能清理或分析临时表。由于这个原因，应该通过会话的 SQL 命令执行合适的清理和分析操作。例如，如果一个临时表将要被用于复杂的查询，最好在把它填充完毕后在其上运行ANALYZE。

可以选择将GLOBAL或LOCAL写在TEMPORARY或TEMP的前面。这当前在PostgreSQL中没有区别并且已被废弃，见Compatibility。

UNLOGGED

如果指定，该表被创建为一个不受日志记录的表。被写入到不做日志的表中的数据不会被写到预写式日志中（见第 30 章，这让它们比普通表快非常多。不过，它们在崩溃时是不安全的：一个不做日志的表在一次崩溃或非干净关闭之后会被自动地截断。一个不做日志的表中的内容也不会被复制到后备服务器中。在一个不做日志的表上创建的任何索引也会自动地不被日志记录。

IF NOT EXISTS

如果一个同名关系已经存在，不要抛出一个错误。在这种情况下会发出一个提示。注意这不保证现有的关系是和将要被创建的表相似的东西。

table_name

要被创建的表名（可以选择用模式限定）。

OF type_name

创建一个类型化的表，它的结构取自于指定的组合类型（名字可以选择用模式限定）。一个类型化的表和它的类型绑定在一起，例如如果类型被删除，该表也将被删除（用DROP TYPE ... CASCADE）。

当一个类型化的表被创建时，列的数据类型由底层的组合类型决定而没有在CREATE TABLE命令中直接指定。但是CREATE TABLE命令可以对表增加默认值和约束，并且可以指定存储参数。

column_name

列的名称会在新表中被建立.

data_type

列的数据类型. 这可以包括数组 规格. 有关PostgreSQL支持数据类型的详细信息, 请参考第 8 章

COLLATE collation

COLLATE子句为该列（必须是一种可排序数据类型）赋予一个排序规则。 如果没有指定，将使用该列数据类型的默认排序规则。

INHERITS (parent_table [, ...])

可选的INHERITS子句指定一个表的列表， 新表将从其中自动地继承所有列。 父表可以是普通表或者外部表。

INHERITS的使用在新的子表和它的父表之间创建一种持久的关系。 对于父表的模式修改通常也会传播到子表， 并且默认情况下子表的数据会被包括在对父表的扫描中。

如果在多个父表中存在同名的列，除非父表中每一个这种列的数据类型都能匹配， 否则会报告一个错误。如果没有冲突，那么重复列会被融合来形成新表中的一个单一列。 如果新表中的列名列表包含一个也是继承而来的列名，该数据类型必须也匹配继承的列，并且列定义会被融合成一个。如果新表显式地为列指定了任何默认值， 这个默认值将覆盖来自该列继承声明中的默认值。 否则，任何父表都必须为该列指定相同的默认值，或者会报告一个错误。

CHECK约束本质上也采用和列相同的方式被融合： 如果多个父表或者新表定义中包含相同的命名CHECK约束， 这些约束必须全部具有相同的检查表达式，否则将报告一个错误。 具有相同名称和表达式的约束将被融合成一份拷贝。 一个父表中的被标记为NO INHERIT的约束将不会被考虑。 注意新表中一个未命名的CHECK约束将永远不会被融合，因为那样总是会为它选择一个唯一的名字。

列的STORAGE设置也会从父表复制过来。

如果父表中的列是标识列，那么该属性不会被继承。 如果需要，可以将子表中的列声明为标识列。

PARTITION BY { RANGE | LIST | HASH } ({ column_name | (expression) } [opclass] [, ...])

可选的PARTITION BY子句指定了对表进行分区的策略。 这样创建的表称为分区表。 带括号的列或表达式的列表构成表的分区键。 使用范围或哈希分区时，分区键可以包含多个列或表达式（最多32个，但在构建 PostgreSQL时可以更改此限制）， 但对于列表分区，分区键必须由单个列或表达式组成。

范围和列表分区需要 btree 运算符类，而哈希分区需要哈希运算符类。 如果没有运算符类被显式指定，将使用相应类型的默认运算符类； 如果不存在默认运算符类，则将引发错误。 使用哈希分区时，所使用的运算符类必须实现支持功能 2（详情请参阅第 38.15.3 节。

分区表被分成多个子表（称为分区），它们是使用单独的CREATE TABLE命令创建的。 分区表本身是空的。插入到表中的数据行将根据分区键中的列或表达式的值路由到分区。 如果没有现有的分区与新行中的值匹配，则会报告错误。

Partitioned tables do not support EXCLUDE constraints; however, you can define these constraints on individual partitions. Also, while it's possible to define PRIMARY KEY constraints on partitioned tables, creating foreign keys that reference a partitioned table is not yet supported. 分区表不支持EXCLUDE约束； 但是，你可以在各个分区上定义这些约束。 此外，虽然可以在分区表上定义PRIMARY KEY约束， 但引用分区表来创建外键还不能支持。

有关表分区的更多讨论，请参阅第 5.10 节

PARTITION OF parent_table { FOR VALUES partition_bound_spec | DEFAULT }

将表创建为指定父表的分区。 该表建立时，可以使用FOR VALUES创建为特定值的分区，也可以使用DEFAULT创建默认分区。 此选项不适用于哈希分区表。

partition_bound_spec 必须对应于父表的分区方法和分区键，并且必须不能与该父表的任何现有分区重叠。 具有IN的形式用于列表分区， 具有FROM和TO的形式用于范围分区， 具有WITH的形式用于哈希分区。

在partition_bound_spec 中指定的每个值都是一个文字、NULL、MINVALUE或MAXVALUE。每个文字值必须是可对相应的分区键列类型强制的数字常量， 或者是该类型的有效输入的字符串文字。

在创建列表分区时，可以指定NULL来表示分区允许分区键列为空。 但是，给定父表不能有多于一个这样的列表分区。无法为范围分区指定 NULL。

创建范围分区时，由FROM指定的下限是一个包含范围， 而用TO指定的上限是排除范围。也就是说， 在FROM列表中指定的值是该分区的相应分区键列的有效值， 而TO列表中的值不是。请注意， 必须根据按行比较的规则来理解此语句（第 9.23.5 节。 例如，给定PARTITION BY RANGE (x,y)，分区范围 FROM (1, 2) TO (3, 4)允许x=1与任何y>=2，x=2与任何非空y，和x=3与任何y<4。

在创建范围分区以指示列值没有下限或上限时，可以使用特殊值MINVALUE 和MAXVALUE。例如，使用FROM (MINVALUE) TO (10) 定义的分区允许任何小于10的值，并且使用FROM (10) TO (MAXVALUE) 定义的分区允许任何大于或等于10的值。

创建涉及多个列的范围分区时，将MAXVALUE作为下限的一部分并将 MINVALUE作为上限的一部分也是有意义的。例如，使用 FROM (0, MAXVALUE) TO (10, MAXVALUE) 定义的分区允许第一个分区键列大于0且小于或等于10的任何行。类似地， 使用FROM ('a', MINVALUE) TO ('b', MINVALUE)定义的分区 允许第一个分区键列以"a"开头的任何行。

请注意，如果MINVALUE或MAXVALUE用于分区边界的一列， 则必须为所有后续列使用相同的值。例如，(10, MINVALUE, 0) 不是有效的边界；你应该写(10, MINVALUE, MINVALUE)。

还要注意，某些元素类型，如timestamp，具有“无穷”的概念， 这只是另一个可以存储的值。这与MINVALUE和MAXVALUE不同， 它们不是可以存储的实际值，而是它们表示值无界的方式。MAXVALUE 可以被认为比任何其他值（包括“无穷”）都大的值，MINVALUE 可以被认为是比任何其他值（包括“负无穷”）都小的值。因此， 范围FROM ('infinity') TO (MAXVALUE)不是空的范围； 它只允许存储一个值—"infinity"。

如果指定了DEFAULT，则表将创建为父表的默认分区。 父级表可以是列表或范围分区表。 不适合给定父级表的任何其他分区的分区键值将路由到默认分区。 给定父级表只能有一个默认分区。

当一个表已有DEFAULT 分区并且要对它添加新分区时， 必须扫描现有的默认分区以验证它不包含可能属于新分区的任何行。 如果默认分区包含大量行，则速度可能会很慢。 如果默认分区是外表或者它具有可证明的不可能包含能放置在新分区中的行的约束，则将略过扫描

当创建哈希分区时，必须指定模数和余数。 模数必须是正整数，余数必须是小于模数的非负整数。 通常情况下，当初始设置哈希分区表时，应选择一个与分区数相等的模数，并为每个表分配相同的模数和不同的余数（请参阅下方示例）。 不过，并不要求每个分区都具有相同的模数，只要求哈希分区表里面的分区中出现的每个模数都是下一个较大模数的因数。 这允许以增量的方式增加分区数量而不需要一次移动所有数据。 例如，假设你有一个包含 8 个分区的哈希分区表，每个分区有模数8，但发现有必要将分区数增加到 16 个。 您可以拆分其中一个模数-8分区，然后创建两个新的模数-16分区来覆盖键空间的相同部分（一个的余数等于被拆分的分区的余数，另一个的余数等于该值加 8），而后用数据重新填充他们。 然后，你可以对每一个余数-8分区重复此操作过程，直到没有剩余。 虽然这其中的每个步骤都可能会导致大量的数据移动操作，它仍然要好于建一个全新的表并一次移动全部数据。

分区必须与其所属的分区表的字段名和类型相同。如果父表声明为WITH OIDS那么所有的分区必须具有OID；父表的OID字段和其他字段一样被所有分区继承。 对分区表字段名或

类型的修改，或者OID字段的添加或删除，将自动传播到所有分区。 CHECK约束将自动被每一个分区继承，但是单独的分区可以指定额外的 CHECK约束；与父表相同名称和条件的额外约束将被父表约束合并。 可以为每个分区分别指定默认值。

插入分区表中的行将自动路由到正确的分区。如果不存在合适的分区，则会发生错误。

像TRUNCATE这样的操作通常会影响一个表及其所有继承子级，这些操作将级联到所有分区， 但也可能在单个分区上执行。请注意，使用DROP TABLE 删除分区需要在父表上采用ACCESS EXCLUSIVE锁。

LIKE source_table [like_option ...]

LIKE指定新表将从哪一个表自动地复制所有的列名、数据类型以及它们的非空约束。

和INHERITS不同，新表和原始表在创建完成之后是完全分离的。对原始表的更改将不会被应用到新表，并且不可能在原始表的扫描中包括新表的数据。

只有INCLUDING DEFAULTS被指定时，被拷贝的列定义的默认表达式才会被拷贝。默认的行为是排除默认表达式，导致新表中被拷贝过来的列的默认值为空值。注意，如果拷贝的默认值调用了数据库修改函数（如nextval），则可能在原始表和新表之间创建功能联系。

仅在声明了INCLUDING IDENTITY的情况下拷贝复制字段定义的标识声明。 为新表的每个标识列创建一个新的序列，与旧表相关的序列区分开。

非空约束总是会被复制到新表。CHECK约束只有在INCLUDING CONSTRAINTS被指定时才会被复制。列约束和表约束之间没有区别对待。

Extended statistics are copied to the new table if INCLUDING STATISTICS is specified. 如果指定了INCLUDING STATISTICS，那么扩展统计会被复制到新表。

只有INCLUDING INDEXES被指定时，原始表上的索引、PRIMARY KEY、UNIQUE以及EXCLUDE约束将被创建在新表上。新索引和约束的名称将根据默认规则选定，而不管原始的名称如何（这种行为可以避免新索引重名导致的失败）。

只有INCLUDING STORAGE被指定时，复制而来的列定义的STORAGE设置才会被复制。默认行为会排除STORAGE设置，导致新表中复制而来的列具有与类型相关的默认设置。更多关于STORAGE设置的信息，请见第 68.2 节

只有INCLUDING COMMENTS被指定时，复制而来的列、约束和索引的注释才会被拷贝。默认行为是排除注释，这导致新表中复制而来的列和约束没有注释。

INCLUDING ALL是 INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS INCLUDING IDENTITY INCLUDING INDEXES INCLUDING STATISTICS INCLUDING STORAGE. 的简写形式。

注意和INHERITS不同，用LIKE拷贝的列和约束不会和相似的命名列及约束融合。如果显式指定了相同的名称或者在另一个LIKE子句中指定了相同的名称，将会发出一个错误。

LIKE子句也能被用来从视图、外部表或组合类型拷贝列定义。不适合的选项（例如来自视图的INCLUDING INDEXES）会被忽略。

CONSTRAINT constraint_name

一个列约束或表约束的可选名称。如果该约束被违背，约束名将会出现在错误消息中，这样类似列必须为正的约束名可以用来与客户端应用沟通有用的约束信息（指定包含空格的约束名时需要用到双引号）。如果没有指定约束名，系统将生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列允许包含空值。这是默认情况。

这个子句只是提供与非标准 SQL 数据库的兼容。在新的应用中不推荐使用。

CHECK (expression) [NO INHERIT]

CHECK指定一个产生布尔结果的表达式，一个插入或更新操作要想成功，其中新的或被更新的行必须满足该表达式。计算出 TRUE 或 UNKNOWN 的表达式就会成功。只要任何一个插入或更新操作的行产生了 FALSE 结果，将报告一个错误异常并且插入或更新不会修改数据库。一个被作为列约束指定的检查约束只应该引用该列的值，而一个出现在表约束中的表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能引用当前行的列之外的变量。可以引用系统列tableoid，但不能引用其他系统列。

一个被标记为NO INHERIT的约束将不会传播到子表。

当一个表有多个CHECK约束时，检查完NOT NULL约束后，对于每一行会以它们名称的字母表顺序来进行检查（版本 9.5 之前的PostgreSQL对于CHECK约束不遵从任何特定的引发顺序）。

DEFAULT default_expr

DEFAULT子句为出现在其定义中的列赋予一个默认数据。该值是可以使用变量的表达式（不允许用子查询和对其他列的交叉引用）。默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有为该列指定值的插入操作中。如果一列没有默认值，那么默认值为空值。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(sequence_options)]

该子句将列创建为标识列。 它将拥有一个隐式序列附加到它，并且新行中的列将自动从分配给它的序列中获取值。

子句ALWAYS和BY DEFAULT确定在 INSERT语句中，序列值如何优先于用户指定的值。 如果指定了ALWAYS，则只有在INSERT 语句指定OVERRIDING SYSTEM VALUE时才接受用户指定的值。 如果指定了BY DEFAULT，则用户指定的值优先。 有关详细信息，请参见INSERT。（在COPY命令中，无论此设置如何，都始终使用用户指定的值。）

可选的sequence_options子句可用于覆盖序列的选项。 有关详细信息，请参见CREATE SEQUENCE。

UNIQUE （列约束）
UNIQUE (column_name [, ...]) [INCLUDE (column_name [, ...])] (表约束)

UNIQUE约束指定一个表中的一列或多列组成的组包含唯一的值。唯一表约束的行为与列约束的行为相同，只是表约束能够跨越多列。

对于一个唯一约束的目的来说，空值不被认为是相等的。

每一个唯一表约束必须命名一个列的集合，并且它与该表上任何其他唯一或主键约束所命名的列集合都不相同（否则它将是一个被列举了两次的约束）。

When establishing a unique constraint for a multi-level partition hierarchy,all the columns in the partition key of the target partitioned table, as well as those of all its descendant partitioned tables, must be included in the constraint definition. 在为多级分区层次结构建立唯一约束时， 目标分区表的分区键中的所有列，以及那些由它派生的所有分区表， 必须被包含在约束定义中。

添加唯一约束将自动在使用于约束的列或列组上创建唯一的 btree索引。 可选子句INCLUDE在不强制唯一性的情况下向该索引添加一个或多个列。 请注意虽然约束在包含的列上是非强制的，但是它仍然依赖于它们。 因此，这些列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

PRIMARY KEY （列约束）
PRIMARY KEY (column_name [, ...]) [INCLUDE (column_name [, ...])] (表约束)

PRIMARY KEY约束指定表的一个或者多个列只能包含唯一（不重复）、非空的值。一个表上只能指定一个主键，可以作为列约束或表约束。

主键约束所涉及的列集合应该不同于同一个表上定义的任何唯一约束的列集合（否则，该唯一约束是多余的并且会被丢弃）。

PRIMARY KEY强制的数据约束可以看成是UNIQUE和NOT NULL的组合， 不过把一组列标识为主键也为模式设计提供了元数据，因为主键标识其他表可以依赖这一个列集合作为行的唯一标识符。

PRIMARY KEY 约束共享UNIQUE 约束放到分区表上时限制。

添加PRIMARY KEY约束将自动在用于约束的列或列组上创建唯一的 btree 索引。 可选的INCLUDE 子句允许指定将被包含在索引的非-键部分中的列的列表。 虽然包含的列的唯一性是非强制的，但约束仍依赖于它们。 因此，这些包含的列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

EXCLUDE [USING index_method] (exclude_element WITH operator [, ...]) index_parameters [WHERE (predicate)]

EXCLUDE子句定一个排除约束，它保证如果任意两行在指定列或表达式上使用指定操作符进行比较，不是所有的比较都将会返回TRUE。如果所有指定的操作符都测试相等，这就等价于一个UNIQUE约束，尽管一个普通的唯一约束将更快。不过，排除约束能够指定比简单相等更通用的约束。例如，你可以使用&&操作符指定一个约束，要求表中没有两行包含相互覆盖的圆（见 第 8.8 节。

排除约束使用一个索引实现，这样每一个指定的操作符必须与用于索引访问方法index_method的一个适当的操作符类（见第 11.10 节相关联。操作符被要求是交换的。每一个exclude_element可以选择性地指定一个操作符类或者顺序选项，这些在CREATE INDEX中有完整描述。

访问方法必须支持amgettuple（见第 61 章，目前这意味着GIN无法使用。尽管允许，但是在一个排除约束中使用 B-树或哈希索引没有意义，因为它无法做得比一个普通唯一索引更出色。因此在实践中访问方法将总是GiST或SP-GiST。

predicate允许你在该表的一个子集上指定一个排除约束。在内部这会创建一个部分索引。注意在为此周围的圆括号是必须的。

REFERENCES reftable [(refcolumn)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] （列约束）
FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] （表约束）

这些子句指定一个外键约束，它要求新表的一列或一个列的组必须只包含能匹配被引用表的某个行在被引用列上的值。 如果refcolumn列表被忽略，将使用reftable的主键。被引用列必须是被引用表中一个非可延迟唯一约束或主键约束的列。 用户必须在被引用的表（或整个表,或特定的引用列）上拥有REFERENCES权限。 增加的外键约束需要SHARE ROW EXCLUSIVE 锁定引用的表。 注意外键约束不能在临时表和永久表之间定义。 此外请注意，虽然可以在分区表上定义外键，但不能够声明引用分区表的外键。

被插入到引用列的一个值会使用给定的匹配类型与被引用表的值进行匹配。 有三种匹配类型：MATCH FULL、MATCH PARTIAL以及MATCH SIMPLE（这是默认值）。

MATCH FULL将不允许一个多列外键中的一列为空，除非所有外键列都是空；如果它们都是空，则不要求该行在被引用表中有一个匹配。 MATCH SIMPLE允许任意外键列为空，如果任一为空，则不要求该行在被引用表中有一个匹配。 MATCH PARTIAL现在还没有被实现（当然，NOT NULL约束能被应用在引用列上来组织这些情况发生）。

另外，当被引用列中的数据被改变时，在这个表的列中的数据上可以执行特定的动作。ON DELETE指定当被引用表中一个被引用行被删除时要执行的动作。同样，ON UPDATE指定当被引用表中一个被引用列被更新为新值时要执行的动作。如果该行被更新，但是被引用列并没有被实际改变，不会做任何动作。除了NO ACTION检查之外的引用动作不能被延迟，即便该约束被声明为可延迟的。对每一个子句可能有以下动作：

NO ACTION

产生一个错误指示删除或更新将会导致一个外键约束违背。如果该约束被延迟，并且仍存在引用行，这个错误将在约束检查时被产生。这是默认动作。

RESTRICT

产生一个错误指示删除或更新将会导致一个外键约束违背。这个动作与NO ACTION形同，不过该检查不是可延迟的。

CASCADE

删除任何引用被删除行的行，或者把引用列的值更新为被引用列的新值。

SET NULL

将引用列设置为空。

SET DEFAULT

设置引用列为它们的默认值（如果该默认值非空，在被引用表中必须有一行匹配该默认值，否则该操作将会失败）。

如果被引用列被频繁地更改，最好在引用列上加上一个索引，这样与外键约束相关的引用动作能够更高效地被执行。

DEFERRABLE
NOT DEFERRABLE

这个子句控制该约束是否能被延迟。一个不可延迟的约束将在每一次命令后立刻被检查。可延迟约束的检查将被推迟到事务结束时进行（使用SET CONSTRAINTS命令）。NOT DEFERRABLE是默认值。当前，只有UNIQUE、PRIMARY KEY、EXCLUDE以及REFERENCES（外键）约束接受这个子句。NOT NULL以及CHECK约束是不可延迟的。注意在包括ON CONFLICT DO UPDATE子句的INSERT语句中，可延迟约束不能被用作冲突裁判者。

INITIALLY IMMEDIATE
INITIALLY DEFERRED

如果一个约束是可延迟的，这个子句指定检查该约束的默认时间。如果该约束是INITIALLY IMMEDIATE，它会在每一个语句之后被检查。这是默认值。如果该约束是INITIALLY DEFERRED，它只会在事务结束时被检查。约束检查时间可以用SET CONSTRAINTS命令修改。

WITH (storage_parameter [= value] [, ...])

这个子句为一个表或索引指定可选的存储参数，详见存储参数。一个表的WITH子句还可以包括OIDS=TRUE（或者只包括OIDS）来指定新表的行应该具有被分配的 OID（对象标识符），或者包括OIDS=FALSE来指定新表的行不具有 OID。如果没有指定OIDS，默认设置取决于default_with_oids配置参数（如果新表是从任何具有 OID 的表继承而来，那么即使该命令要求OIDS=FALSE也会强制使用OIDS=TRUE）。

如果指定或者蕴含了OIDS=FALSE，新表就不会存储 OID 并且对插入其中的一个新行不会分配 OID。这通常值得考虑，因为它将减少 OID 消耗并且因而推迟 32 为 OID 计数器的回卷。一旦计数器回卷，OID 就不再能被假定为唯一，这就使它们不那么有用了。另外，从一个表中排除 OID 可以减少存储该表所需的磁盘空间，减少的量是每行减少 4 字节（在大部分机器上），这也略微提高了性能。

要在表被创建后从中移除 OID，使用ALTER TABLE.

WITH OIDS
WITHOUT OIDS

这些语法已被荒废，它们分别等效于WITH (OIDS)和WITH (OIDS=FALSE)。如果你希望同时给出一个OIDS设置和存储参数，你必须使用WITH (...)语法，见上文。

ON COMMIT

临时表在一个事务块结束时的行为由ON COMMIT控制。三种选项是：

PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

DELETE ROWS

在每一个事务块结束时将删除临时表中的所有行。 实质上，在每一次提交时会完成一次自动的TRUNCATE。 当应用于分区表上时，这不会级联到它的分区。

DROP

在当前事务块结束时将删除临时表。 当在分区表上使用时，这个操作会删除他的分区，而在具有继承子级的表上使用时，它将删除依赖的子级。

TABLESPACE tablespace_name

tablespace_name是新表要创建于其中的表空间名称。如果没有指定，将参考default_tablespace，或者如果表是临时的则参考temp_tablespaces。

USING INDEX TABLESPACE tablespace_name

这个子句允许选择与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引将被创建在哪个表空间中。如果没有指定，将参考default_tablespace，或者如果表是临时的则参考temp_tablespaces。

存储参数

WITH子句能够为表或与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引指定存储参数。 用于索引的存储参数已经在CREATE INDEX中介绍过。 当前可用于表的存储参数在下文中列出。 如下文所示，对于很多这类参数，都有一个名字带有toast.前缀的附加参数，它能被用来控制该表的二级TOAST表（如果存在）的行为（关于 TOAST 详见第 68.2 节。 如果一个表的参数值被设置但是相应的toast.参数没有被设置，那么 TOAST 表将使用该表的参数值。 不支持为分区表指定这些参数，但可以为单个叶子分区指定它们。

fillfactor (integer)

一个表的填充因子是一个 10 到 100 之间的百分数。100（完全填满）是默认值。当一个较小的填充因子被指定时，INSERT操作会把表页面只填满到指定的百分比，每个页面上剩余的空间被保留给该页上行的更新。这就让UPDATE有机会把一行的已更新版本放在与其原始版本相同的页面上，这比把它放在一个不同的页面上效率更高。对于一个项从来不会被更新的表来说，完全填满是最好的选择，但是在更新繁重的表上则较小的填充因子更合适。这个参数不能对 TOAST 表设置。

toast_tuple_target (integer)

在我们尝试将长列值移动到TOAST表中之前，toast_tuple_target指定需要的最小元组长度， 也是在toasting开始时尝试减少长度的目标长度。 这仅影响标记为"外部"或"扩展"的列，并且仅适用于新元数 - 对现有行没有影响。 默认情况下此参数设置为允许每个块至少 4 个元组，默认块大小为 2040 字节。 有效值介于 128 字节和(块大小-标头)之间，默认大小为 8160 字节。 更改此值对于非常短或非常长的行可能没有用处。 请注意默认设置通常接近最佳状态，在某些情况下设置此参数可能会产生负面影响。 不能对TOAST表设置此参数。

parallel_workers (integer)

这个参数设置用于辅助并行扫描这个表的工作者数量。 如果没有设置这个参数，系统将基于关系的尺寸来决定一个值。 规划者或使用并行扫描的实用程序选择的工作者数量可能会比较少，例如max_worker_processes的设置较小就是一种可能的原因。

autovacuum_enabled, toast.autovacuum_enabled (boolean)

为一个特定的表启用或者禁用自动清理守护进程。如果为真，自动清理守护进程将遵照第 24.1.6 节讨论的规则在这个表上执行自动的VACUUM或者ANALYZE操作。如果为假，这个表不会被自动清理，不过为了阻止事务 ID 回卷时还是会对它进行自动的清理。有关回卷阻止请见第 24.1.5 节如果autovacuum参数为假，自动清理守护进程根本就不会运行（除非为了阻止事务 ID 回卷），设置独立的表存储参数也不会覆盖这个设置。因此显式地将这个存储参数设置为true很少有大的意义，只有设置为false才更有用。

autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold (integer)

autovacuum_vacuum_threshold参数对于每个表的值。

autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor (float4)

autovacuum_vacuum_scale_factor参数对于每个表的值。

autovacuum_analyze_threshold (integer)

autovacuum_analyze_threshold参数对于每个表的值。

autovacuum_analyze_scale_factor (float4)

autovacuum_analyze_scale_factor参数对于每个表的值。

autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay (integer)

autovacuum_vacuum_cost_delay参数对于每个表的值。

autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit (integer)

autovacuum_vacuum_cost_limit参数对于每个表的值。

autovacuum_freeze_min_age, toast.autovacuum_freeze_min_age (integer)

vacuum_freeze_min_age参数对于每个表的值。注意自动清理将忽略超过系统范围autovacuum_freeze_max_age参数一半的针对每个表的autovacuum_freeze_min_age参数。

autovacuum_freeze_max_age, toast.autovacuum_freeze_max_age (integer)

autovacuum_freeze_max_age参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的autovacuum_freeze_max_age参数。

autovacuum_freeze_table_age, toast.autovacuum_freeze_table_age (integer)

vacuum_freeze_table_age参数对于每个表的值。

autovacuum_multixact_freeze_min_age, toast.autovacuum_multixact_freeze_min_age (integer)

vacuum_multixact_freeze_min_age参数对于每个表的值。注意自动清理将忽略超过系统范围autovacuum_multixact_freeze_max_age参数一半的针对每个表的autovacuum_multixact_freeze_min_age参数。

autovacuum_multixact_freeze_max_age, toast.autovacuum_multixact_freeze_max_age (integer)

autovacuum_multixact_freeze_max_age参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的autovacuum_multixact_freeze_max_age参数。

autovacuum_multixact_freeze_table_age, toast.autovacuum_multixact_freeze_table_age (integer)

vacuum_multixact_freeze_table_age参数对于每个表的值。

log_autovacuum_min_duration, toast.log_autovacuum_min_duration (integer)

log_autovacuum_min_duration参数对于每个表的值。

user_catalog_table (boolean)

声明该表是一个用于逻辑复制目的的额外的目录表。详见第 49.6.2 节不能对 TOAST 表设置这个参数。

注解

我们不推荐在新应用中使用 OID：在可能要用到的地方，使用一个标识列或者其他序列生成器作为表的主键会更好。不过，如果你的应用确实需要用到 OID 来标识一个表的特定行，我们推荐在表的oid列上创建一个唯一约束，来确保表中的 OID 在计数器回卷后能唯一标识行。如果你需要一个数据库范围的唯一标识符，要避免假定 OID 在表之间也是唯一的，应该用tableoid和行 OID 的组合来实现该目的。

> **提示**
>
> 我们不推荐对没有主键的表使用OIDS=FALSE，因为没有 OID 或 唯一数据键，就很难标识特定的行。

PostgreSQL为每一个唯一约束和主键约束创建一个索引来强制唯一性。因此，没有必要显式地为主键列创建一个索引（详见CREATE INDEX）。

在当前的实现中，唯一约束和主键不会被继承。这使得继承和唯一约束的组合相当不正常。

一个表不能有超过 1600 列（实际上，由于元组长度限制，有效的限制通常更低）。

例子

创建表films和表distributors：

```
CREATE TABLE films (
```

```
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建有一个二维数组的表：

```
CREATE TABLE array_int (
    vector  int[][]
);
```

为表films定义一个唯一表约束。唯一表约束能够被定义在表的一列或多列上：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个列检查约束：

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

定义一个表检查约束：

```
CREATE TABLE distributors (
    did     integer,
 name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

为表films定义一个主键表约束：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
```

```
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

为表distributors定义一个主键约束。下面的两个例子是等价的，第一个使用表约束语法，第二个使用列约束语法：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

为列name赋予一个文字常量默认值，安排列did的默认值是从一个序列对象中选择下一个值产生，并且让modtime的默认值是该行被插入的时间：

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表distributors上定义两个NOT NULL列约束，其中之一被显式给定了一个名称：

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

为name列定义一个唯一约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

创建相同的表，指定表和它的唯一索引指定 70% 的填充因子：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
```

```
)
WITH (fillfactor=70);
```

创建表circles，带有一个排除约束阻止任意两个圆重叠：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

在表空间diskvol1中创建表cinemas：

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

创建一个组合类型以及一个类型化的表：

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

创建一个范围分区表：

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

创建在分区键中具有多个列的范围分区表：

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));
```

创建列表分区表：

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

建立哈希分区表:

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

创建范围分区表的分区：

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

使用分区键中的多个列创建范围分区表的几个分区：

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

创建列表分区表的分区：

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

创建本身是分区的列表分区表的分区，然后向其添加分区：

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

建立哈希分区表的分区：

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
```

```
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

建立默认分区：

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

Compatibility

CREATE TABLE命令遵从SQL标准，除了以下例外。

临时表

尽管CREATE TEMPORARY TABLE的语法很像 SQL 标准的语法，但事实是并不相同。在标准中，临时表只需要被定义一次并且会自动地存在（从空内容开始）于需要它们的每一个会话中。PostgreSQL则要求每一个会话为每一个要用的临时表发出它自己的CREATE TEMPORARY TABLE命令。这允许不同的会话为不同的目的使用相同的临时表名，而标准的方法约束一个给定临时表名的所有实例都必须具有相同的表结构。

标准中对于临时表行为的定义被广泛地忽略了。PostgreSQL在这一点上的行为和多种其他 SQL 数据库是相似的。

SQL 标准也区分全局和局部临时表，其中一个局部临时表为每一个会话中的每一个 SQL 模块具有一个独立的内容集合，但是它的定义仍然是多个会话共享的。因为PostgreSQL不支持 SQL 模块，这种区别与PostgreSQL无关。

为了兼容性目的，PostgreSQL将在临时表声明中接受GLOBAL和LOCAL关键词，但是它们当前没有效果。我们不鼓励使用这些关键词，因为未来版本的PostgreSQL可能采用一种更兼容标准的（对它们含义的）解释。

临时表的ON COMMIT子句也和 SQL 标准相似，但是有一些不同。如果忽略ON COMMIT子句，SQL 指定默认行为是ON COMMIT DELETE ROWS。但是，PostgreSQL中的默认行为是ON COMMIT PRESERVE ROWS。SQL 中不存在ON COMMIT DROP选项。

非延迟唯一性约束

但一个UNIQUE或PRIMARY KEY约束是非可延迟的，只要一个行被插入或修改，PostgreSQL就会立即检查唯一性。SQL 标准指出只有在语句结束时才应该强制唯一性。当一个单一命令更新多个键值时，这两者是不同的。要得到兼容标准的行为，将该约束声明为DEFERRABLE但是不延迟（即INITIALLY IMMEDIATE）。注意这可能要显著地慢于立即唯一性检查。

列检查约束

SQL 标准指出CHECK列约束只能引用它们应用到的列，只有CHECK表约束能够引用多列。PostgreSQL并没有强制这个限制，它同样处理列检查约束和表检查约束。

EXCLUDE 约束

EXCLUDE约束类型是一种PostgreSQL扩展。

NULL “约束”

NULL “约束”（实际上是一个非约束）是一个PostgreSQL对 SQL 标准的扩展，它也被包括（以及对称的NOT NULL约束）在一些其他的数据库系统中以实现兼容性。因为它是任意列的默认值，它的存在就像噪声一样。

Constraint Naming

SQL标准规定在包含表或域的模式范围内表和域的约束必须具有唯一的名称。 PostgreSQL是比较宽松的：它只需要约束名称在附加到特定表或域的约束之间是唯一的。 但是，对于基于索引的约束(UNIQUE,PRIMARY KEY, and EXCLUDEconstraints)， 这个特别的自由度并不存在， 因为关联的索引被命名为与约束相同的名称，并且索引名称在相同模式的所有关系中必须是唯一的。

Currently, PostgreSQL does not record names for NOT NULL constraints at all, so they are not subject to the uniqueness restriction. This might change in a future release. 当前，PostgreSQL没有记录NOT NULL约束的名称， 因此它们不受唯一性限制的影响。这在将来的版本中可能会改变。

继承

通过INHERITS子句的多继承是一种PostgreSQL的语言扩展。SQL:1999 以及之后的标准使用一种不同的语法和不同的语义定义了单继承。SQL:1999-风格的继承还没有被PostgreSQL。

零列表

PostgreSQL允许创建一个没有列的表（例如CREATE TABLE foo();）。这是一个对于 SQL 标准的扩展，它不允许零列表。零列表本身并不是很有用，但是不允许它们会为ALTER TABLE DROP COLUMN带来奇怪的特殊情况，因此忽略这种规则限制看起来更加整洁。

多个标识列

PostgreSQL允许一个表拥有多个标识列。 该标准指定一个表最多只能有一个标识列。这主要是为了给模式更改或迁移提供更大的灵活性。 请注意，INSERT命令仅支持一个适用于整个语句的覆盖子句， 因此不支持具有不同行为的多个标识列。

LIKE 子句

虽然 SQL 标准中有一个LIKE子句，但是PostgreSQL接受的很多LIKE子句选项却不在标准中，并且有些标准中的选项也没有被PostgreSQL实现。

WITH子句

WITH子句是一个PostgreSQL扩展，存储参数和 OID 都不在标准中。

表空间

PostgreSQL的表空间概念不是标准的一部分。因此，子句TABLESPACE和USING INDEX TABLESPACE是扩展。

类型化的表

类型化的表实现了 SQL 标准的一个子集。根据标准，一个类型化的表具有与底层组合类型相对应的列，以及其他的“自引用列”。PostgreSQL 不显式支持这些自引用列，但是可以使用 OID 特性获得相同的效果。

PARTITION BY 子句

PARTITION BY子句是PostgreSQL的一个扩展。

PARTITION OF 子句

PARTITION OF子句PostgreSQL的一个扩展。

参见

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

CREATE TABLE AS — 从一个查询的结果创建一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

描述

CREATE TABLE AS创建一个表，并且用 由一个SELECT命令计算出来的数据填充 该表。该表的列具有和SELECT的输出列　相关的名称和数据类型（不过可以通过给出一个显式的新列名列表来覆 盖这些列名）。

CREATE　TABLE　AS和创建一个视图有些　相似，但是实际上非常不同：它会创建一个新表并且只计算该查询一次　用来初始填充新表。这个新表将不会跟踪该查询源表的后续变化。相反，　一个视图只要被查询，它的定义SELECT 语句就会被重新计算。

参数

GLOBAL或者LOCAL

为兼容性而忽略。不推荐使用这些关键词，详见 CREATE TABLE。

TEMPORARY或者TEMP

如果被指定，该表会被创建为一个临时表。详见 CREATE TABLE。

UNLOGGED

如果被指定，该表会被创建为一个不做日志的表。详见 CREATE TABLE。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个　　提示。详见CREATE TABLE。

table_name

要创建的表的名称（可以被模式限定）。

column_name

新表中一列的名称。如果没有提供列名，会从查询的输出列名中得到。

WITH (storage_parameter [= value] [, ...])

这个子句为新表指定可选的存储参数，详见　　存储参数。　　WITH子句也能包括OIDS=TRUE（或者只是 OIDS）来指定新表的行应该被分配 OID （对象标识符）， 或者包括OIDS=FALSE来指定行没有 OID。详见 CREATE TABLE。

WITH OIDS
WITHOUT OIDS

这些是分别等效于WITH (OIDS)和 WITH (OIDS=FALSE)的即将过时的语法。如果你希望同时 给出OIDS设置和存储参数，你必须使用 WITH (...)语法，见上文。

ON COMMIT

临时表在事务块结束时的行为可以用ON COMMIT 控制。三个选项是：

PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

DELETE ROWS

在每一个事务块结束时临时表中的所有行都将被删除。本质上， 在每次提交时会完成一次自动的TRUNCATE。

DROP

在当前事务块结束时将删掉临时表。

TABLESPACE tablespace_name

tablespace_name 是要在其中创建新表的表空间名称。如果没有指定，将会查询default_tablespace，临时表会查询 temp_tablespaces。

query

一个SELECT、TABLE或者VALUES 命令，或者是一个运行准备好的SELECT、 TABLE或者VALUES查询的EXECUTE命令。

WITH [NO] DATA

这个子句指定查询产生的数据是否应该被复制到新表中。如果不是，只有 表结构会被复制。默认是复制数据。

注解

这个命令在功能上类似于SELECT INTO，但是它更好， 因为不太可能被SELECT INTO语法的其他使用混淆。更 进一步，CREATE TABLE AS提供了 SELECT INTO的功能的一个超集。

CREATE TABLE AS命令允许用户显式地指定 是否应该包括 OID。如果没有显式地指定 OID 的存在，将使用 default_with_oids配置变量来判断。

示例

创建一个新表films_recent，它只由表 films中最近的项组成：

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

要完全地复制一个表，也可以使用TABLE命令的 简短形式：

```
CREATE TABLE films2 AS
  TABLE films;
```

用一个预备语句创建一个新的临时表films_recent， 它仅由表films中最近的项组成。新表有 OID 并且将在提交时被删除：

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

兼容性

CREATE TABLE AS符合 SQL标准。下面的是非标准扩展：

- 标准要求在子查询子句周围有圆括号，在 PostgreSQL中这些圆括号是可选的。
- 在标准中，WITH [NO] DATA子句是必要的，而 PostgreSQL 中是可选的。
- PostgreSQL处理临时表的方式和标准不同。 详见CREATE TABLE。
- WITH子句是一种 PostgreSQL扩展， 标准中既没有存储参数也没有 OID。
- PostgreSQL的表空间概念是标准的 一部分。因此，子句TABLESPACE是一种扩展。

另见

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — 定义一个新的表空间

大纲

```
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

描述

CREATE TABLESPACE注册一个新的集簇范围 的表空间。表空间的名称必须与数据库集簇中现有的任何表空间不同。

表空间允许超级用户在文件系统上定义另一个位置，可以把包含数据库对象 （例如表和索引）的数据文件放在那里。

一个具有适当特权的用户可以把 tablespace_name传递给 CREATE DATABASE、CREATE TABLE、 CREATE INDEX或者ADD CONSTRAINT 来让这些对象的数据文件存储在指定的表空间中。

> **警告**
>
> 表空间不能独立于定义它的集簇使用，见 第 22.6 节

参数

tablespace_name

The name of a tablespace to be created. The name cannot begin with pg_, as such names are reserved for system tablespaces.

user_name

将拥有该表空间的用户名。如果省略，默认为执行该命令的用户。只有 超级用户能创建表空间，但是它们能把表空间的拥有权赋予给非超级 用户。

directory

要被用于表空间的目录。该目录应该为空并且必须由 PostgreSQL系统用户拥有。该目录必须用一个绝对 路径指定。

tablespace_option

要设置或者重置的表空间参数。当前，唯一可用的参数是 seq_page_cost、random_page_cost 以及effective_io_concurrency。 为一个特定表空间设定其中一个值将覆盖规划器对该表空间中表页读取的 常规代价估计，常规代价估计是由同名的配置参数所建立（见 seq_page_cost、 random_page_cost、effective_io_concurrency）。如果一个表空间位于一个 比其他 I/O 子系统更慢或者更快的磁盘上，这些参数就能发挥作用。

注解

只有在支持符号链接的系统上才支持表空间。

CREATE TABLESPACE不能在一个事务块内被执行。

示例

在/data/dbs创建一个表空间dbspace：

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

在/data/indexes创建一个genevieve 用户拥有的表空间indexspace：

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

兼容性

CREATE TABLESPACE是一种 PostgreSQL扩展。

另见

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — 定义一个新的文本搜索配置

大纲

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

描述

CREATE TEXT SEARCH CONFIGURATION 创建一个新的文本搜索配置。一个文本搜索配置指定一个文本搜索解析器（ 它能把字符串解析成记号），外加一些词典（可被用来决定哪些记号是搜索 感兴趣的）。

如果只指定了解析器，那么新文本搜索配置最初没有从记号类型到词典的映射， 并且因此将忽略所有词。后续的ALTER TEXT SEARCH CONFIGURATION命令必须被用来创建映射以让该配置变得可用。 另一种方式是复制一个现有的文本搜索配置。

如果给定一个模式名称，则文本搜索配置会被创建在指定的模式中。否则它将会 被创建在当前模式中。

定义一个文本搜索配置的用户会成为其拥有者。

进一步的信息请参考第 12 章

参数

name

要创建的文本搜索配置的名称。该名称可以被模式限定。

parser_name

这个配置要使用的文本搜索解析器的名称。

source_config

要复制的已有文本搜索配置的名称。

注解

PARSER和COPY选项是互斥的，因为当 一个已有的配置被复制时，它的解析器选择也会被复制。

兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH CONFIGURATION语句。

另见

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — 定义一个新的文本搜索字典

大纲

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

描述

CREATE TEXT SEARCH DICTIONARY创建一个 新的文本搜索字典。一个文本搜索字典指定一种识别搜索感兴趣或者不感兴趣 的单词的方法。 一个字典依赖于一个文本搜索模板，后者指定了实际执行该工 作的函数。通常该字典提供一些控制该模板函数细节行为的选项。

如果给出了一个模式名称，那么该文本搜索字典会被创建在指定的模式中。 否则它会被创建在当前模式中。

定义文本搜索字典的用户将成为其拥有者。

进一步的信息可参考第 12 章

参数

name

要创建的文本搜索字典的名称。该名称可以是模式限定的。

template

将定义这个字典基本行为的文本搜索模板的名称。

option

要为此字典设置的模板相关选项的名称。

value

用于模板相关选项的值。如果该值不是一个简单标识符或者数字，它必须 被加引用（你可以按照你所希望的总是对它加上引用）。

选项可以以任意顺序出现。

示例

下面的例子命令创建了一个基于 Snowball 的字典，它使用了非标准的 停用词列表。

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH DICTIONARY语句。

另见

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — 定义一个新的文本搜索解析器

大纲

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

描述

CREATE TEXT SEARCH PARSER创建一个 新的文本搜索解析器。一个文本搜索解析器定义把文本字符串分解成记号 并且为记号分配类型（分类）的方法。一个解析器本身并不特别有用，但 是必须与一些要用于搜索的文本搜索字典一起被绑定到一个文本搜索配置 中。

如果给出了一个模式名称，那么文本搜索解析器将被创建在指定的模式中。 否则它会被创建在当前模式中。

只有超级用户才能使用 CREATE TEXT SEARCH PARSER（这是因为 错误的文本搜索定义可能会让服务器混淆甚至崩溃）。

更多信息可以参考第 12 章

参数

name

要创建的文本搜索解析器的名称。该名称可以是模式限定的。

start_function

用于该解析器的开始函数的名称。

gettoken_function

用于该解析器的取下一个记号的函数名称。

end_function

用于该解析器的结束函数的名称。

lextypes_function

用于该解析器的词法分析器函数（一个返回其产生的记号类型集合信息的函数）的名称。

headline_function

用于该解析器的标题函数（一个对记号集合进行综述的函数）的名称。

如有必要，函数的名称可以被模式限定。参数类型没有给出， 因为函数的每个类型的参数列表无法被预先决定。除了标题函数之外， 所有函数都是必要的。

参数可以以任何顺序出现，而不是必须按照上面所展示的顺序。

兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH PARSER语句。

另见

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — 定义一种新的文本搜索模板

大纲

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

描述

CREATE TEXT SEARCH TEMPLATE创建一个 新的文本搜索模板。文本搜索模板定义实现文本搜索字典的函数。一个模板 本身没什么用处，但是必须被实例化为一个字典来使用。字典通常指定要给 予模板函数的参数。

如果给出了一个模式名，文本搜索模板会被创建在指定模式中。否则它会被 创建在当前模式中。

必须成为超级用户以使用 CREATE TEXT SEARCH TEMPLATE。做出 这种限制是因为错误的模板定义会让服务器混淆甚至崩溃。将模板与字典 分隔开来的原因是模板中封装了定义字典的“不安全”方面。在 定义字典时可以被设置的参数对非特权用户是可以安全设置的，因此创建 字典不需要拥有特权来操作。

进一步的信息可以参考第 12 章

参数

name

要创建的额文本搜索模板的名称。该名称可以被模式限定。

init_function

用于模板的初始化函数的名称。

lexize_function

用于模板的分词函数的名称。

如有必要，函数名称可以被模式限定。参数类型没有给出，因为每一类 函数的参数列表是预先定义好的。分词函数是必需的，但是初始化函数 是可选的。

参数可以以任何顺序出现，而不是只能按照上文的顺序。

兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH TEMPLATE语句。

另见

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — 定义一个新的转换

大纲

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (
    FROM SQL WITH FUNCTION from_sql_function_name [ (argument_type [, ...]) ],
    TO SQL WITH FUNCTION to_sql_function_name [ (argument_type [, ...]) ]
);
```

简介

CREATE TRANSFORM定义一种新的转换。 CREATE OR REPLACE TRANSFORM将 创建一种新的转换或者替换现有的定义。

一种转换指定了如何把一种数据类型适配到一种过程语言。例如，在用 PL/Python 编写一个使用hstore类型的函数时，PL/Python 没有关于如何在 Python 环境中表示hstore值的先验知识。 语言的实现通常默认会使用文本表示，但是在一些时候这很不方便，例如 有时可能用一个联合数组或者列表更合适。

一种转换指定了两个函数：

- 一个“from SQL”函数负责将类型从 SQL 环境转换到语言。 这个函数将在该语言编写的一个函数的参数上调用。
- 一个“to SQL”函数负责将类型从语言转换到 SQL 环境。这 个函数将在该语言编写的一个函数的返回值上调用。

没有必要同时提供这些函数。如果有一种没有被指定，将在必要时使用与语言相 关的默认行为（为了完全阻止在一个方向上发生转换，你也可以写一个总是报错 的转换函数）。

要创建一种转换，你必须拥有该类型并且具有该类型上的 USAGE特权，拥有该语言上的USAGE特权，并且拥有 from-SQL 和 to-SQL 函数（如果 指定了）及其上的EXECUTE特权。

参数

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

from_sql_function_name[(argument_type [, ...])]

将该类型从 SQL 环境转换到该语言的函数名。它必须接受一个 internal类型的参数并且返回类型internal。 实参将是该转换所适用的类型，并且该函数也应该被写成认为它是那种类型（ 但是不允许声明一个返回internal但没有至少一个 internal类型参数的 SQL层函数）。实际的返回值将与 语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

to_sql_function_name[(argument_type [, ...])]

将该类型从语言转换到 SQL 环境的函数名。它必须接受一个 internal类型的参数并且返回该转换所适用的类型。实参值 将与语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

注解

使用DROP TRANSFORM移除转换。

示例

要为类型hstore和语言 plpythonu创建一种转换，先搞定该类型和语言：

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpythonu;
```

然后创建需要的函数：

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

最后创建转换把它们连接起来：

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

实际上，这些命令将被包裹在扩展中。

contrib小节包含了一些提供转换的扩展， 它们可以作为实际的例子。

兼容性

这种形式的CREATE TRANSFORM是一种 PostgreSQL扩展。在 SQL标准中有一个CREATE TRANSFORM命令，但是它是用于把数据类型适配到 客户端语言。该用法不受 PostgreSQL支持。

另见

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — 定义一个新触发器

大纲

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event
 [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY
 DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name }
 [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

这里的event可以是下列之一：

```
    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

描述

CREATE TRIGGER创建一个新触发器。该触发器将被关联到指定的表、视图或者外部表并且在表上发生特定操作时将执行指定的函数function_name。

该触发器可以被指定为在一行上尝试该操作之前触发（在约束被检查并且INSERT、UPDATE或者DELETE被尝试之前）；也可以在该操作完成之后触发（在约束被检查并且INSERT、UPDATE或者DELETE完成之后）；或者取代该操作（在对一个视图插入、更新或删除的情况中）。如果该触发器在事件之前触发或者取代事件，该触发器可以跳过对当前行的操作或者改变正在被插入的行（只对INSERT以及UPDATE操作）。如果该触发器在事件之后触发，所有更改（包括其他触发器的效果）对该触发器“可见”。

一个被标记为FOR EACH ROW的触发器会对该操作修改的每一行都调用一次。例如，一个影响 10 行的DELETE将导致在目标关系上的任何ON DELETE触发器被独立调用 10 次，也就是为每一个被删除的行调用一次。与此相反，一个被标记为FOR EACH STATEMENT的触发器只会为任何给定的操作执行一次，不管该操作修改多少行（特别地，一个修改零行的操作将仍会导致任何可用的FOR EACH STATEMENT触发器被执行）。

被指定为要触发INSTEAD OF触发器事件的触发器必须被标记为FOR EACH ROW，并且只能被定义在视图上。一个视图上的BEFORE和AFTER触发器必须被标记为FOR EACH STATEMENT。

此外，触发器可以被定义成为TRUNCATE触发，但只能是FOR EACH STATEMENT。

下面的表格总结了哪些触发器类型可以被用在表、视图和外部表上：

何时	事件	行级	语句级
BEFORE	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表
	TRUNCATE	—	表

何时	事件	行级	语句级
AFTER	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表
	TRUNCATE	—	表
INSTEAD OF	INSERT/ UPDATE/DELETE	视图	—
	TRUNCATE	—	—

还有，一个触发器定义可以指定一个布尔的WHEN条件，它将被测试来看看该触发器是否应该被触发。在行级触发器中，WHEN条件可以检查该行的列的新旧值。语句级触发器也可以有WHEN条件，尽管该特性对于它们不是很有用（因为条件不能引用表中的任何值）。

如果有多个同种触发器被定义为相同事件触发，它们将按照名称的字母表顺序被触发。

当CONSTRAINT选项被指定，这个命令会创建一个约束触发器。这和一个常规触发器相同，不过触发该触发器的时机可以使用SET CONSTRAINTS调整。约束触发器必须是表上的AFTER ROW触发器。它们可以在导致触发器事件的语句末尾被引发或者在包含该语句的事务末尾被引发。在后一种情况中，它们被称作是被延迟。一个待处理的延迟触发器的引发也可以使用SET CONSTRAINTS立即强制发生。当约束触发器实现的约束被违背时，约束触发器应该抛出一个异常。

REFERENCING选项启用对传递关系的收集，传递关系是包括被当前SQL语句插入、删除或者修改的行的行集合。这个特性让触发器能看到该语句做的事情的全局视图，而不是一次只看到一行。仅对非约束触发器的AFTER触发器允许这个选项。此外，如果触发器是一个UPDATE触发器，则它不能指定column_name列表。OLD TABLE仅可以被指定一次，并且只能为在UPDATE或DELETE事件上引发的触发器指定，它创建的传递关系包含有该语句更新或删除的所有行的前映像。类似地，NEW TABLE仅可以被指定一次，并且只能为在UPDATE或INSERT事件上引发的触发器指定，它创建的传递关系包含有该语句更新或插入的所有行的后映像。

SELECT不修改任何行，因此你无法创建SELECT触发器。规则和视图可以为需要SELECT触发器的问题提供可行的解决方案。

关于触发器的更多信息请见第 39 章

参数

name

给新触发器的名称。这必须与同一个表上的任何其他触发器相区别。名称不能是模式限定的 — 该触发器会继承它所在表的模式。对于一个约束触发器，这也是使用SET CONSTRAINTS修改触发器行为时要用到的名字。

BEFORE
AFTER
INSTEAD OF

决定该函数是要在事件之前、之后被调用还是会取代该事件。一个约束触发器也能被指定为AFTER。

event

INSERT、UPDATE、DELETE或者TRUNCATE之一，这指定了将要引发该触发器的事件。多个事件可以用OR指定，要求传递关系的时候除外。

对于UPDATE事件，可以使用下面的语法指定一个列的列表：

```
UPDATE OF column_name1 [, column_name2 ... ]
```

只有当至少一个被列出的列出现在UPDATE命令的更新目标中时，该触发器才会触发。

INSTEAD OF UPDATE事件不允许列的列表。在请求传递关系时，也不能指定列的列表。

table_name

要使用该触发器的表、视图或外部表的名称（可能是模式限定的）。

referenced_table_name

约束引用的另一个表的名称（可能是模式限定的）。这个选项被用于外键约束并且不推荐用于一般的目的。这只能为约束触发器指定。

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

该触发器的默认时机。这些约束选项的细节可参考CREATE TABLE文档。这只能为约束触发器指定。

REFERENCING

这个关键词紧接在一个或者两个关系名的声明之前，这些关系提供对触发语句的传递关系的访问。

OLD TABLE
NEW TABLE

这个子句指示接下来的关系名是用于前映像传递关系还是后映像传递关系。

transition_relation_name

在该触发器中这个传递关系要使用的（未限定）名称。

FOR EACH ROW
FOR EACH STATEMENT

这指定该触发器函数是应该为该触发器事件影响的每一行被引发一次，还是只为每个SQL 语句被引发一次。如果都没有被指定，FOR EACH STATEMENT会是默认值。约束触发器只能被指定为FOR EACH ROW。

condition

一个决定该触发器函数是否将被实际执行的布尔表达式。如果指定了WHEN，只有condition返回true时才会调用该函数。在FOR EACH ROW触发器中，WHEN条件可以分别写OLD.column_name或者NEW.column_name来引用列的新旧行值。当然，INSERT触发器不能引用OLD并且DELETE触发器不能引用NEW。

INSTEAD OF触发器不支持WHEN条件。

当前，WHEN表达式不能包含子查询。

注意对于约束触发器，对于WHEN条件的计算不会被延迟，而是直接在行更新操作被执行之后立刻发生。如果该条件计算得不到真，那么该触发器就不会被放在延迟执行的队列中。

function_name

一个用户提供的函数，它被声明为不用参数并且返回类型trigger，当触发器引发时会执行该函数。

在CREATE TRIGGER的语法中，关键词FUNCTION和PROCEDURE是等效的，但是任何情况下被引用的函数必须是一个函数而不是过程。这里，关键词PROCEDURE的使用是有历史原因的并且已经被废弃。

arguments

一个可选的逗号分隔的参数列表，它在该触发器被执行时会被提供给该函数。参数是字符串常量。简单的名称和数字常量也可以被写在这里，但是它们将全部被转换成字符串。请检查该触发器函数的实现语言的描述来找出在函数内部如何访问这些参数，这可能与普通函数参数不同。

注解

要在一个表上创建一个触发器，用户必须具有该表上的TRIGGER特权。用户还必须具有在触发器函数上的EXECUTE特权。

使用DROP TRIGGER移除一个触发器。

当一个列相关的触发器（使用UPDATE OF column_name语法定义的触发器）的列被列为UPDATE命令的SET列表目标时，它会被触发。即便该触发器没有被引发，一个列的值也可能改变，因为BEFORE UPDATE触发器对行内容所作的改变不会被考虑。相反，一个诸如UPDATE ... SET x = x ...的命令将引发一个位于列x上的触发器，即便该列的值没有改变。

在一个BEFORE触发器中，WHEN条件正好在函数被或者将被执行之前被计算，因此使用WHEN与在触发器函数的开始测试同一个条件没有实质上的区别。特别注意该条件看到的NEW行是当前值，虽然可能已被早前的触发器所修改。还有，一个BEFORE触发器的WHEN条件不允许检查NEW行的系统列（例如oid），因为那些列还没有被设置。

在一个AFTER触发器中，WHEN条件正好在行更新发生之后被计算，并且它决定一个事件是否要被放入队列以便在语句的末尾引发该触发器。因此当一个AFTER触发器的WHEN条件不返回真时，没有必要把一个事件放入队列或者在语句末尾重新取得该行。如果触发器只需要为一些行被引发，就能够显著地加快修改很多行的语句的速度。

在一些情况下，单一的SQL命令可能会引发多种触发器。例如，一个带有ON CONFLICT DO UPDATE子句的INSERT可能同时导致插入和更新操作，因此它将根据需要引发这两种触发器。提供给触发器的传递关系与它们的事件类型有关，因此INSERT触发器将只看到被插入的行，而UPDATE触发器将只看到被更新的行。

由外键强制动作导致的行更新或删除（例如ON UPDATE CASCADE或ON DELETE SET NULL）被当做导致它们的SQL命令的一部分。受影响的表上的相关触发器将被引发，这样就提供了另一种方法让SQL命令引发不直接匹配其类型的触发器。在简单的情况中，请求传递关系的触发器将在一个传递关系中看到由原始SQL命令在其表中做出的所有改变。不过，有些情况中一个请求传递关系的AFTER ROW触发器的存在将导致由单个SQL命令触发的外键强制动作被分成多步，每一步都有其自己的传递关系。在这种情况下，没创建一个传递关系集合都会引发存在的所有语句级触发器，确保那些触发器能够在一个传递关系中看到每个受影响的行一次，并且只看到一次。

只有当视图上的动作被一个行级INSTEAD OF触发器处理时才会引发视图上的语句级触发器。如果动作被一个INSTEAD规则处理，那么该语句发出的任何语句都会代替提及该视图的原始语句执行，这样将被引发的触发器是替换语句中提及的表上的那些触发器。类似地，如果视图是自动可更新的，则该动作将被处理为把该语句自动重写成在视图基表上的一个动作，这样基表的语句级触发器就是要被引发的。

在分区表上创建一个行级触发器将导致在它所有的现有分区上创建相同的触发器，并且以后创建或者挂接的任何分区也将包含一个相同的触发器。分区表上的触发器只能是AFTER。

修改分区表或者带有继承子表的表会引发挂接到显式提及表的语句级触发器，但不会引发其分区或子表的语句级触发器。相反，行级触发器会在受影响的分区或子表上引发，即便它们在查询中没有被明确提及。如果一个语句级触发器用REFERENCING子句定义有传递关系，则

来自所有受影响分区或子表中的行的前后映像都是可见的。在继承子表的情况中，行映像仅包括该触发器所附属的表中存在的列。当前，不能在分区或继承子表上定义带有传递关系的行级触发器。

在PostgreSQL 7.3 以前的版本中，必须要声明触发器函数为返回占位符类型opaque而不是trigger。要支持载入旧的转储文件，CREATE TRIGGER将接受一个被声明为返回opaque的函数，但是它会发出一个通知并且会把该函数的声明返回类型改为trigger。

例子

只要表accounts的一行即将要被更新时会执行函数check_account_update：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

下面的例子与上面一个例子相同，但是只在UPDATE命令指定要更新balance列时才执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

这种形式只有列balance具有真正被改变的值时才执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

调用一个函数来记录accounts的更新，但是只在有东西被改变时才调用：

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

为每一个要插入到视图底层表中的行执行函数view_insert_row：

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE FUNCTION view_insert_row();
```

为每个语句执行函数check_transfer_balances_to_zero以确认transfer的行不会有净值增加：

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
```

```
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

为每一行执行函数check_matching_pairs以确认（同一个语句）同时对匹配对做了更改 ：

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE FUNCTION check_matching_pairs();
```

第 39.4 节包含一个用 C 编写的触发器函数的完整例子。

兼容性

PostgreSQL中的CREATE TRIGGER语句实现了SQL标准的一个子集。目前缺少下列功能：

- 虽然AFTER触发器的传递表名是以标准的方式用REFERENCING子句指定，但REFERENCING子句中不能指定FOR EACH ROW触发器中用到的行变量。它们以依赖于编写该触发器函数的语言的方式可用，但是对任意一种语言来说是固定的。一些语言实际上的行为就像有包含OLD ROW AS OLD NEW ROW AS NEW的REFERENCING子句存在一样。

- 标准允许把传递表与和列相关的UPDATE触发器一起使用，那么应该在传递表中可见的行集合取决于该触发器的列列表。当前PostgreSQL没有实现这一点。

- PostgreSQL只允许为被触发动作执行一个用户定义的函数。标准允许执行许多其他的 SQL 命令作为被触发的动作，例如CREATE TABLE。这种限制可以很容易地通过创建一个执行想要的命令的用户定义函数来绕过。

SQL 指定多个触发器应该以被创建时间的顺序触发。PostgreSQL则使用名称顺序，这被认为更加方便。

SQL 指定级联删除上的BEFORE DELETE触发器在级联的DELETE完成之后引发。PostgreSQL的行为则是BEFORE DELETE总是在删除动作之前引发，即使是一个级联删除。这被认为更加一致。 如果BEFORE触发器修改行或者在引用动作引起的更新期间阻止更新，这也是非标准行为。这能导致约束违背或者被存储的数据不遵从引用约束。

使用OR为一个单一触发器指定多个动作的能力是 SQL 标准的一个PostgreSQL扩展。

为TRUNCATE引发触发器的能力是 SQL 标准的一个PostgreSQL扩展，在视图上定义语句级触发器的能力也是一样。

CREATE CONSTRAINT TRIGGER是SQL标准的一个PostgreSQL扩展。

参见

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

CREATE TYPE — 定义一种新的数据类型

大纲

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

描述

CREATE TYPE在当前数据库中注册一种新的 数据类型。定义数据类型的用户将成为它的拥有者。

如果给定一个模式名，那么该类型将被创建在指定的模式中。否则它会被 创建在当前模式中。类型名称必须与同一个模式中任何现有的类型或者域 相区别（因为表具有相关的数据类型，类型名称也必须与同一个模式中任 何现有表的名字不同）。

如上面的语法所示，有五种形式的CREATE TYPE。 它们分别创建组合类型、枚举类型、 范围类型、基础类型或者 shell 类型。下文将依次讨论前四种形式。shell 类型仅仅 是一种用于后面要定义的类型的占位符，通过发出一个不带除类型名之外其 他参数的CREATE TYPE命

令可以创建这种类型。 在创建范围类型和基础类型时，需要 shell 类型作为一种向前引用。

组合类型

第一种形式的CREATE TYPE创建组合类型。 组合类型由一个属性名和数据类型的列表指定。如果属性的数据类型是可 排序的，也可以指定该属性的排序规则。组合类型本质上和表的行类型相 同，但是如果只想定义一种类型，使用 CREATE TYPE避免了创建一个实际的表。单独的组合类型也是很有用的，例如可以作为函数的参数或者返回类型。

为了能够创建组合类型，必须拥有在其所有属性类型上的 USAGE特权。

枚举类型

如第 8.7 节所述，第二种形式的 CREATE TYPE创建枚举类型。枚举类型需要 一个带引号的标签构成的列表，每一个标签长度必须不超过 NAMEDATALEN字节（在标准的 PostgreSQL编译中是 64 字节）。 可以创建具有零个标签的枚举类型，但是在使用ALTER TYPE 添加至少一个标签之前，不能使用这种类型来保存值。

范围类型

如第 8.17 节所述，第三种形式的 CREATE TYPE创建范围类型。

范围类型的subtype可以 是任何带有一个相关的 B 树操作符类（用来决定该范围类型值的顺序）的类型。 通常，子类型的默认 B 树操作符类被用来决定顺序。要使用一种非默认操作符 类，可以用 subtype_opclass指定它 的名字。如果子类型是可排序的并且希望在该范围的顺序中使用一种非默认的 排序规则，可以用collation选项来指定。

可选的canonical函数 必须接受一个所定义的范围类型的参数，并且返回同样类型的一个值。在适 用时，它被用来把范围值转换成一种规范的形式。更多信息请见第 8.17.8 节 创建一个 canonical函数有点棘 手，因为必须在声明范围类型之前定义它。要这样做，必须首先创建一种 shell 类型，它是一种没有属性只有名称和拥有者的占位符类型。这可以通过 发出不带额外参数的命令CREATE TYPE name来完成。然后可以使用该 shell 类型作为 参数和结果来声明该函数，并且最终用同样的名称来声明范围类型。这会自动 用一种合法的范围类型替换 shell 类型项。

可选的subtype_diff函数 必须接受两个subtype类型 的值作为参数，并且返回一个double precision值表示两个给定 值之间的差别。虽然这是可选的，但是提供这个函数会让该范围类型列上 GiST 索 引效率更高。详见第 8.17.8 节

基础类型

第四种形式的CREATE TYPE创建一种新的 基本类型（标量类型）。为了创建一种新的基本类型，你必须是一个超级 用户（做这种限制的原因是一种错误的类型定义可能让服务器混淆甚至 崩溃）。

参数可以以任意顺序出现（而不仅是按照上面所示的顺序），并且大部分 是可选的。在定义类型前，必须注册两个或者更多函数（使用 CREATE FUNCTION）。支持函数 input_function以及 output_function 是必需的，而函数 receive_function、send_function、 type_modifier_input_function、 type_modifier_output_function和analyze_function 是可选的。通常来说这些函数必须是用 C 或者另外一种低层语言编写的。

input_function将 类型的外部文本表达转换成为该类型定义的操作符和函数所使用的内部表达。 output_function 执行反向的转换。输入函数可以被声明为有一个cstring 类型的参数，或者有三个类型分别为cstring、 oid、integer的参数。第一个参数是 以 C 字符串存在的输入文本，第二个参数是该类型自身的 OID（对于 数组类型则是其元素类型的 OID），第三个参数是目标列的 typmod（如果知道，不知道则将传递 -1）。输入函数必须 返回一个该数据类型本身的值。通常，一个输入函数应该被声明为 STRICT。 如果不是这样，在读到一个 NULL 输入值时，调用它时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回

NULL，除非它发生了错误（ 这种情况主要是想支持域输入函数，它们可能需要拒绝 NULL 输入）。 输出函数必须被声明为有一个新数据类型的参数。输出函数必须返回类型 cstring。对于 NULL 值不会调用输出函数。

T可选的receive_function 会把类型的外部二进制表达转换成内部表达。如果没有提供这个函数， 该类型不能参与到二进制输入中。二进制表达转换成内部形式代价更低， 然而却更容易移植（例如，标准的整数数据类型使用网络字节序作为外 部二进制表达，而内部表达是机器本地的字节序）。接收函数应该执行 足够的检查以确保该值是有效的。接收函数可以被声明为有一个 internal类型的参数，或者有三个类型分别为 internal、oid、integer的参数。第一个参数是一个指向StringInfo缓冲区的 指针，其中保存着接收到的字节串。其余可选的参数和文本输入函数的 相同。接收函数必须返回一个该数据类型本身的值。通常，一个接收函 数应该被声明为 STRICT。如果不是这样，在读到一个 NULL 输入值时，调用它时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非它发生了错误（这种情况主要是想支持域接收函数，它们 可能需要拒绝 NULL 输入）。类似地，可选的 send_function将 内部表达转换成外部二进制表达。如果没有提供这个函数，该类型将不 能参与到二进制输出中。发送函数必须被声明为有一个新数据类型的参 数。发送函数必须返回类型bytea。对于 NULL 值不 会调用发送函数。

到这里你应该在疑惑输入和输出函数是如何能被声明为具有新类型的 结果或参数的？因为必须在创建新类型之前创建这两个函数。这个问题 的答案是，新类型应该首先被定义为一种shell type， 它是一种占位符类型，除了名称和拥有者之外它没有其他属性。这可以 通过不带额外参数的命令CREATE TYPE name做到。然后用 C 写的 I/O 函数可以 被定义为引用这种 shell type。最后，用带有完整定义的 CREATE TYPE把该 shell type 替换为一个完全的、合 法的类型定义，之后新类型就可以正常使用了。

如果该类型支持修饰符（附加在类型声明上的可选约束，例如 char(5)或者numeric(30,2)），则需要可选的 type_modifier_input_function 以及type_modifier_output_function。 PostgreSQL允许用户定义的类型有一个或者 多个简单常量或者标识符作为修饰符。不过，为了存储在系统目录中，该信息必须 能被打包到一个非负整数值中。所声明的修饰符会被以cstring数组的形式 传递给type_modifier_input_function。 它必须检查该值的合法性（如果值错误就抛出一个错误），如果值正确，要返回 一个非负integer值，它将被存储在“typmod”列中。如果 类型没有 type_modifier_input_function 则类型修饰符将被拒绝。type_modifier_output_function 把内部的整数 typmod 值转换回正确的形式用于用户显示。它必须返回一个 cstring值，该值就是追加到类型名称后的字符串。例如 numeric的函数可能会返回(30,2)。如果默认的显示格式 就是只把存储的 typmod 整数值放在圆括号内，则允许省略 type_modifier_output_function。

可选的analyze_function 为该数据类型的列执行与类型相关的统计信息收集。默认情况下，如果 该类型有一个默认的 B-树操作符类，ANALYZE将尝试用 类型的“equals”和“less-than”操作符来收集统计信息。 这种行为对于非标量类型并不合适，因此可以通过指定一个自定义分析函数来 覆盖这种行为。分析函数必须被声明为有一个类型为internal的参 数，并且返回一个boolean结果。分析函数的详细 API 请见 src/include/commands/vacuum.h。

虽然只有 I/O 函数和其他为该类型创建的函数才知道新类型的内部表达的细节， 但是内部表达的一些属性必须被向 PostgreSQL声明。其中最重要的是 internallength。基本数据 类型可以是定长的（这种情况下 internallength是一个正 整数）或者是变长的（把internallength设置为 VARIABLE，在内部通过把typlen设置为 -1 表示）。 所有变长类型的内部表达都必须以一个 4 字节整数开始，它给出了这个值的总 长度（注意如第 68.2 节所述，长度域常常是被编码 过的，直接接受它是不明智的）。

可选的标志PASSEDBYVALUE表示这种数据类型的值需要 被传值而不是传引用。传值的类型必须是定长的，并且它们的内部表达不能超 过Datum类型（某些机器上是 4 字节，其他机器上是 8 字节）的 尺寸。

alignment参数指定数据 类型的存储对齐要求。允许的值等同于以 1、2、4 或 8 字节边界对齐。注意 变长类型的 alignment 参数必须至少为 4，因为它们需要包含一个 int4作为它们的第一个组成部分。

storage参数允许 为变长数据类型选择存储策略（对定长类型只允许 plain）。plain指定该类型的数 据将总是被存储在线内并且不会被压缩。extended 指定系统将首先尝试压缩一个长的数据值，并且将在数据仍然太长的情 况下把值移出主表行。external允许值被移出主表， 但是系统将不会尝试对它进行压缩。main允许压缩， 但是不鼓励把值移出主表（如果没有其他办法让行的大小变得合适，具有 这种存储策略的数据项仍将被移出主表，但比起extended以及external项来， 这种存储策略的数据项会被优先考虑保留在主表中）。

如第 68.2 和第 38.12.1 节所述，除plain之外所有的 storage值都暗示 该数据类型的函数能处理被TOAST 过的值。指定的值 仅仅是决定一种可 TOAST 数据类型的列的默认 TOAST 存储策略，用户 可以使用ALTER TABLE SET STORAGE为列选取其他策略。

like_type参数提供 了另一种方法来指定一种数据类型的基本表达属性：从某种现有的类型中 拷贝。internallength、 passedbyvalue、 alignment和 storage的值会从指 定的类型中复制而来（也可以通过在LIKE子句中指定这些属 性的值来覆盖复制过来的值，不过通常并不这么做）。当新类型的低层 实现是以一种现有的类型为“载体”时，用这种方式指定表达特别有用。

category和 preferred参数可以被用来 帮助控制在混淆的情况下应用哪一种隐式造型。每一种数据类型都属于一个用 单个 ASCII 字符命名的分类，并且每一种类型可以是其所属分类中的 “首选”。当有助于解决重载函数或操作符时，解析器将优先 造型到首选类型（但是只能从同类的其他类型造型）。更多细节请见 第 10 章对于没有隐式造型到任意其他类型或者 从任意其他类型造型的类型，让这些设置保持默认即可。不过，对于一组 具有隐式造型的相关类型，把它们都标记为属于同一个类别并且选择一种 或两种“最常用”的类型作为该类别的首选通常是很有用的。在 把一种用户定义的类型增加到一个现有的内建类别（例如数字或者字符串 类型）中时， category参数特别 有用。不过，也可以创建新的全部是用户定义类型的类别。对这样的类别， 可选择除大写字母之外的任何 ASCII 字符。

如果用户希望该数据类型的列被默认为某种非空值，可以指定一个默认值。 默认值可以用DEFAULT关键词指定（这样一个默认值 可以被附加到一个特定列的显式DEFAULT子句覆盖）。

要指定一种类型是数组，用ELEMENT关键词指定该数组元素 的类型。例如，要定义一个 4 字节整数的数组（int4）， 应指定ELEMENT = int4。更多有关数组类型的细节请 见下文。

要指定在这种类型数组的外部表达中分隔值的定界符，可以把delimiter设置为一个特定字符。默认 的定界符是逗号（,）。注意定界符是与数组元素类型相 关的，而不是数组类型本身相关。

如果可选的布尔参数 collatable为真，这种 类型的列定义和表达式可能通过使用COLLATE子句携带 有排序规则信息。在该类型上操作的函数的实现负责真正利用这些信息，仅 把类型标记为可排序的并不会让它们自动地去使用这类信息。

数组类型

只要一种用户定义的类型被创建， PostgreSQL会自动地创建一种相关的 数组类型，其名称由元素类型的名称前面加上一个下划线组成，并且如果长 度超过NAMEDATALEN字节会自动地被截断（如果 这样生成的名称与一种现有类型的名称冲突，该过程将会重复直到找到一个 不冲突的名字）。这种隐式创建的数组类型是变长的并且使用内建的输入和 输出函数（array_in以及array_out）。该数组 类型会追随其元素类型的拥有者或所在模式的任何更改，并且在元素类型被 删除时也被删除。

如果系统会自动地创建正确的数组类型，你可能会很合情合理地问为什么 会有一个ELEMENT选项。使用ELEMENT 唯一有用的情况是：当你在创建一种定长类型，它正好在内部是一个多个 相同东西的数组，并且除了计划给该类型提供的整体操作之外，你想要 允许用下标来直接访问这些东西。例如，类型point被表示 为两个浮点数，可以使用point[0]以及point[1] 来访问它们。注意，这种功能只适用于内部形式正好是一个相同定长域 序列的定长类型。可用下标访问的变长类型必须具有 array_in以及array_out使用的一般化的内部 表达。由于历史原因（即很明显是错的，但现在改已经太晚了），定长 数组类型的下标是从零开始的，而不是像变长数组那样。

参数

name

要创建的类型的名称（可以被模式限定）。

attribute_name

组合类型的一个属性（列）的名称。

data_type

要成为组合类型的一个列的现有数据类型的名称。

collation

要关联到组合类型的一列或者范围类型的现有排序规则的名称。

label

一个字符串，它表达与枚举类型的一个值相关的文本标签。

subtype

范围类型的元素类型的名称，范围类型表示的范围属于该类型。

subtype_operator_class

用于 subtype 的 B 树操作符类的名称。

canonical_function

范围类型的规范化函数的名称。

subtype_diff_function

用于 subtype 的差函数的名称。

input_function

将数据从类型的外部文本形式转换为内部形式的函数名。

output_function

将数据从类型的内部形式转换为外部文本形式的函数名。

receive_function

将数据从类型的外部二进制形式转换成内部形式的函数名。

send_function

将数据从类型的内部形式转换为外部二进制形式的函数名。

type_modifier_input_function

将类型的修饰符数组转换为内部形式的函数名。

type_modifier_output_function

将类型的修饰符的内部形式转换为外部文本形式的函数名。

analyze_function

为该数据类型执行统计分析的函数名。

internallength

一个数字常量，它指定新类型的内部表达的字节长度。默认的假设是 它是变长的。

alignment

该数据类型的存储对齐需求。如果被指定，它必须是 char、int2、 int4或者double。默认是 int4。

storage

该数据类型的存储策略。如果被指定，必须是 plain、external、 extended或者main。默认是plain。

like_type

与新类型具有相同表达的现有数据类型的名称。会从这个类型中复制 internallength、passedbyvalue、 alignment以及 storage的值（ 除非在这个CREATE TYPE命令的其他地方用显式说 明覆盖）。

category

这种类型的分类码（一个 ASCII 字符）。 默认是 “用户定义类型”的'U'。其他的标准分类码可见 表 52.63 为了创建自定义分类， 你也可以选择其他 ASCII 字符。

preferred

如果这种类型是其类型分类中的优先类型则为真，否则为假。默认 为假。在一个现有类型分类中创建一种新的优先类型要非常小心， 因为这可能会导致行为上令人惊奇的改变。

default

数据类型的默认值。如果被省略，默认值是空。

element

被创建的类型是一个数组，这指定了数组元素的类型。

delimiter

在由这种类型组成的数组中值之间的定界符。

collatable

如果这个类型的操作可以使用排序规则信息，则为真。默认为假。

注解

由于一旦数据类型被创建，对该数据类型的使用就没有限制，创建一种基本类型 或者范围类型就等同于在类型定义中提到的函数上授予公共执行权限。对于在类 型定义中有用的函数来说这通常不是问题。但是如果设计一种类型时要求在转换 到外部形式或者从外部形式转换时使用“秘密”信息，你就应该三思而 后行。

在PostgreSQL版本 8.3 之前，自动生成的 数组类型的名称总是正好为元素类型的名称外加一个前置的下划线字符（ _）。因此类型名称的长度限制比其他名称还要少一个字符。 虽然现在这仍然是通常情况，但如果名称达到最大长度或者与其他下划线开头 的用户类型名称

冲突，数组类型的名称也可以不同于这种规则。因此依靠这种　习惯编写代码现在已经不适用了。现在，可以使用 pg_type.typarray来定位与给定类型相关 的数组类型。

建议避免使用以下划线开始的类型名和表名。虽然服务器会改变生成的数组　类型名称以避免与用户给定的名称冲突，仍然有混淆的风险，特别是对旧的　客户端软件来说，它们可能会假定以下划线开始的类型名总是表示数组。

在PostgreSQL 版本 8.2 之前， shell-type 的创建语法 CREATE TYPE name不存在。创建 一种新基本类型的方法是先创建它的输入函数。在这种方法中， PostgreSQL 将首先把新数据类型的名 称看做是输入函数的返回类型。在这种情况下 shell type 会被隐式地创建， 并且能在剩余的 I/O 函数的定义中引用。这种方法现在仍然有效，但是已经 被弃用并且可能会在未来的某个发行中被禁止。还有，为了避免由于函数定 义中的打字错误导致 shell type 弄乱系统目录，当输入函数用 C 编写时， 将只能用这种方法创建一种 shell type。

在PostgreSQL 7.3 以前的版本中， 常常为了完全避免创建 shell type 而把函数对该类型名的向前引用用占位符 伪类型opaque替换。在 7.3 以前，cstring参数和 结果也必须被声明为opaque。为了支持载入旧的转储文件， CREATE TYPE将接受使用opaque声明的 I/O 函数，但是它将发出一个提示并且把函数的声明改成使用正确的类型。

示例

这个例子创建了一种组合类型并且将其用在了一个函数定义中：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

这个例子创建了一个枚举类型并且将其用在一个表定义中：

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

这个例子创建了一个范围类型：

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

这个例子创建了基本数据类型box然后将它用在一个表定义中：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);
```

```
CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果box的内部结构是四个 float4元素的一个数组，我们可能会使用：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这将允许用下标来访问一个 box 值的组件编号。否则该类型的行为和 前面的一样。

这个例子创建了一个大对象类型并且将它用在了一个表定义中：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

更多例子（包括配套的输入和输出函数）请见第 38.12 节

兼容性

创建组合类型的第一种形式的CREATE TYPE命令 符合SQL标准。其他的形式都是 PostgreSQL扩展。SQL 标准中的CREATE TYPE语句也定义了其他 PostgreSQL中没有实现的形式。

创建一种具有零个属性的组合类型的能力是一种 PostgreSQL对标准的背离（类似于 CREATE TABLE中相同的情况）。

另见

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — 定义一个新的数据库角色

大纲

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

这里 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

CREATE USER现在是 CREATE ROLE的一个别名。唯一的区别是 CREATE USER中LOGIN 被作为默认值，而NOLOGIN是 CREATE ROLE的默认值。

兼容性

CREATE USER语句是一种 PostgreSQL扩展。SQL 标准 把用户的定义留给实现来解释。

另见

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — 定义一个用户到一个外部服务器的新映射

大纲

```
CREATE USER MAPPING [IF NOT EXISTS] FOR { user_name | USER | CURRENT_USER |
 PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

描述

CREATE USER MAPPING定义一个用户 到一个外部服务器的新映射。一个用户映射通常会包含连接信息，外部数据包装器 会使用连接信息和外部服务器中包含的信息一起来访问一个外部数据源。

一个外部服务器的拥有者可以为任何服务器任何用户创建用户映射。还有， 如果一个用户被授予了服务器上的USAGE特权，该用户可以 为他们自己的用户名创建用户映射。

参数

IF NOT EXISTS

如果给定用户到给定外部服务器的映射已经存在，则不要抛出错误。 在这种情况下发出通知。请注意，不能保证现有的用户映射与要创建的映射完全相同。

user_name

要映射到外部服务器的一个现有用户的名称。 CURRENT_USER和USER匹配当前用户的名称。 当PUBLIC被指定时，一个所谓的公共映射会被创建，当没有 特定用户的映射可用时将会使用它。

server_name

将为其创建用户映射的现有服务器的名称。

OPTIONS (option 'value' [, ...])

这个子句指定用户映射的选项。这些选项通常定义该映射实际的用户名和 口令。选项名必须唯一。允许的选项名和值与该服务器的外部数据包装器 有关。

示例

为用户bob、服务器foo创建一个用户映射：

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

兼容性

CREATE USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。

另见

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — 定义一个新视图

大纲

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name
 [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

描述

CREATE VIEW定义一个查询的视图。该视图不会被 物理上物质化。相反，在每一次有查询引用该视图时，视图的查询都会被运行。

CREATE OR REPLACE VIEW与之相似，但是如果 已经存在一个同名视图，该视图会被替换。新查询必须产生和现有试图查询相同 的列（也就是相同的列序、相同的列名、相同的数据类型），但是它可以在列表 的末尾加上额外的列。产生输出列的计算可以完全不同。

如果给定了一个模式名（例如CREATE VIEW myschema.myview ...），那么该视图会被创建在指定的模式中。否则，它会 被创建在当前模式中。临时视图存在于一个特殊模式中，因此创建临时视图时不能 给定一个模式名。视图的名称不能与同一模式中任何其他视图、表、序列、索引或 外部表同名。

参数

TEMPORARY或者TEMP

如果被指定，视图被创建为一个临时视图。在当前会话结束时会自动 删掉临时视图。当临时视图存在时，具有相同名称的已有永久视图对 当前会话不可见，除非用模式限定的名称引用它们。

如果视图引用的任何表是临时的，视图将被创建为临时视图（不管有 没有指定TEMPORARY）。

RECURSIVE

创建一个递归视图。语法

```
CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;
```

等效于

```
CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names)
 AS (SELECT ...) SELECT column_names FROM view_name;
```

对于一个递归视图必须指定一个视图列名列表。

name

要创建的视图的名字（可以是模式限定的）。

column_name

要用于视图列的名称列表，可选。如果没有给出，列名会根据查询 推导。

WITH (view_option_name [= view_option_value] [, ...])

这个子句为视图指定一些可选的参数，支持下列参数：

check_option (string)

这个参数可以是local或者cascaded，并且它 等效于指定 WITH [CASCADED | LOCAL] CHECK OPTION（见下文）。 可以使用ALTER VIEW在一个现有视图上修改这个选项。

security_barrier (boolean)

如果希望视图提供行级安全性，应该使用这个参数。详见 第 41.5 节

query

提供视图的行和列的一个SELECT或者 VALUES命令。

WITH [CASCADED | LOCAL] CHECK OPTION

这个选项控制自动可更新视图的行为。这个选项被指定时，将检查该视图上的 INSERT和UPDATE命令以确保新行满足 视图的定义条件（也就是，将检查新行来确保通过视图能看到它们）。如果新行 不满足条件，更新将被拒绝。如果没有指定CHECK OPTION， 会允许该视图上的INSERT和UPDATE命令 创建通过该视图不可见的行。支持下列检查选项：

LOCAL

只根据直接定义在该视图本身的条件检查新行。任何定义在底层基视图上的 条件都不会被检查（除非它们也指定了CHECK OPTION）。

CASCADED

会根据该视图和所有底层基视图上的条件检查新行。如果 CHECK OPTION被指定，并且没有指定 LOCAL和CASCADED，则会假定为 CASCADED。

CHECK OPTION不应该和RECURSIVE视图一起使用。

注意，只有在自动可更新的、没有INSTEAD OF触发器或者 INSTEAD规则的视图上才支持CHECK OPTION。 如果一个自动可更新的视图被定义在一个具有INSTEAD OF 触发器的基视图之上，那么LOCAL CHECK OPTION可以被 用来检查该自动可更新的视图之上的条件，但具有INSTEAD OF 触发器的基视图上的条件不会被检查（一个级联检查选项将不会级联到一个 触发器可更新的视图，并且任何直接定义在一个触发器可更新视图上的检查 选项将被忽略）。如果该视图或者任何基础关系具有导致 INSERT或UPDATE命令被重写的 INSTEAD规则，那么在被重写的查询中将忽略所有检查选项， 包括任何来自于定义在带有INSTEAD规则的关系之上的自动 可更新视图的检查。

注解

使用DROP VIEW语句删除视图。

要小心视图列的名称和类型将会按照你想要的方式指定。例如：

```
CREATE VIEW vista AS SELECT 'Hello World';
```

是不好的形式，因为列名默认为?column?，而且列的数据类型默认为text，这可能不是用户想要的。视图结果中一个字符串更好的风格类似于这样：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

对视图中引用的表的访问由视图拥有者的权限决定。在某些情况下，这可以 被用来提供安全但是受限的底层表访问。不过，并非所有视图都对篡改是安 全的，详见第 41.5 节在视图中调用的函数会被 同样对待，就好像是直接在使用该视图的查询中调用它们一样。因此，一个 视图的用户必须具有调用视图所使用的全部函数的权限。

当CREATE OR REPLACE VIEW被用在一个现有视图上时， 只有该视图的定义 SELECT 规则被改变。其他包括拥有关系、权限和非 SELECT 规则在内的视图属性不会被更改。要替换视图，你必须拥有它（包括 作为拥有角色的一个成员）。

可更新视图

简单视图是自动可更新的：系统将允许在这类视图上以在常规表上相同的方式 使用INSERT、UPDATE以及 DELETE语句。如果一个视图满足以下条件，它就是自动 可更新的：

- 在该视图的FROM列表中刚好只有一项，并且它必须是一个 表或者另一个可更新视图。
- 视图定义的顶层不能包含WITH、DISTINCT、 GROUP BY、HAVING、 LIMIT或者OFFSET子句。
- 视图定义的顶层不能包含集合操作（UNION、 INTERSECT或者EXCEPT）。
- 视图的选择列表不能包含任何聚集、窗口函数或者集合返回函数。

一个自动可更新的视图可以混合可更新列以及不可更新列。如果一个列是对底层 基本关系中一个可更新列的简单引用，则它是可更新的。否则该列是只读的，并 且在一个INSERT或者UPDATE语句尝试对 它赋值时会报出一个错误。

如果视图是自动可更新的，系统将把视图上的任何INSERT、 UPDATE或者DELETE语句转换成在底层 基本关系上的对应语句。带有ON CONFLICT UPDATE子句的 INSERT语句已经被完全支持。

如果一个自动可更新视图包含一个WHERE条件，该条件会限制 基本关系的哪些行可以被该视图上的UPDATE以及 DELETE语句修改。不过，一个允许被UPDATE 修改的行可能让该行不再满足WHERE条件，并且因此也不再能 从视图中可见。类似地，一个INSERT命令可能插入不满足WHERE条件的基本关系行，并且因此从视图中也看不到这些行 （ON CONFLICT UPDATE可能会类似地影响无法通过该视图见 到的现有行）。 CHECK OPTION可以被用来阻止INSERT和UPDATE命令创建这类从视图中无法看到的行。

如果一个自动可更新视图被标记了security_barrier属性，那么 所有该属性的WHERE条件（以及任何使用标记为 LEAKPROOF的操作符的条件）将在该视图使用者的任何条件 之前计算。详见第 41.5 节注意正因为这样，不会 被最终返回的行（因为它们不会通过用户的WHERE条件）可能 仍会结束被锁定的状态。可以用EXPLAIN来查看 哪些条件被应用在关系层面（并且因此不锁定行）以及哪些不会被应用在关系 层面。

一个更加复杂的不满足所有这些条件的视图默认是只读的：系统将不允许在 该视图上的插入、更新或者删除。可以通过在该视图上创建一个 INSTEAD OF触发器来获得可更新视图的效果，该触发器必须 把该视图上的尝试的插入等转换成其他表上合适的动作。更多信息请见CREATE TRIGGER。另一种可能性是创建规则（见 CREATE RULE），不过实际中触发器更容易理解和正确使用。

注意在视图上执行插入、更新或删除的用户必须具有该视图上相应的插入、 更新或删除特权。此外，视图的拥有者必须拥有底层基本关系上的相关特权， 但是执行更新的用户并不需要底层基本关系上的任何权限（见 第 41.5 节。

示例

创建一个由所有喜剧电影组成的视图：

```
CREATE VIEW comedies AS
```

```
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

创建的视图包含创建时film表中的列。尽管* 被用来创建该视图，后来被加入到该表中的列不会成为该视图的组成部分。

创建带有LOCAL CHECK OPTION的视图：

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

这将创建一个基于comedies视图的视图，只显示 kind = 'Comedy'和classification = 'U'的电影。 如果新行没有classification = 'U'，在该视图中的任何 INSERT或UPDATE尝试将被拒绝， 但是电影的kind将不会被检查。

用CASCADED CHECK OPTION创建一个视图：

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

这将创建一个检查新行的kind和classification 的视图。

创建一个由可更新列和不可更新列混合而成的视图：

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

这个视图将支持INSERT、UPDATE 以及DELETE。所有来自于films表的列都 将是可更新的，而计算列country和avg_rating 将是只读的。

创建一个由数字 1 到 100 组成的递归视图：

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

注意在这个CREATE中尽管递归的视图名称是方案限定的，但它内部的自引用不是方案限定的。这是因为隐式创建的CTE的名称不能是方案限定的。

兼容性

CREATE OR REPLACE VIEW是一种 PostgreSQL的语言扩展。临时 视图的概念也是这样。WITH (...)子句也是一种扩展。

另见

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — 释放一个预备语句

大纲

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

描述

DEALLOCATE被用来释放一个之前 准备好的 SQL 语句。如果不显式地释放一个预备语句，会话结束 时会释放它。

更多关于预备语句的信息请见PREPARE。

参数

PREPARE

这个关键词会被忽略。

name

要释放的预备语句的名称。

ALL

释放所有预备语句。

兼容性

SQL 标准包括一个DEALLOCATE语句， 但是只用于嵌入式 SQL。

另见

EXECUTE, PREPARE

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

描述

DECLARE允许用户创建游标， 游标可以被用来在大型查询暂停时检索少量的行。游标被创建后， 可以用FETCH从中取得行。

> **注意**
>
> 这个页面描述在 SQL 命令层面上游标的用法。如果想要在 PL/pgSQL函数中使用游标，其规则是不同的 — 详见第 43.7 节

参数

name

要创建的游标的名称。

BINARY

让游标返回二进制数据而不是返回文本格式数据。

INSENSITIVE

指示从游标中检索数据的过程不受游标创建之后在其底层表上发生的更新的 影响。在PostgreSQL中，这是默认的 行为。因此这个关键词没有实际效果，仅仅被用于兼容 SQL标准。

SCROLL
NO SCROLL

SCROLL指定游标可以用非顺序（例如，反向） 的方式从中检索行。根据查询的执行计划的复杂度，指定 SCROLL可能导致查询执行时间上的性能损失。 NO SCROLL指定游标不能以非顺序的方式从中检索 行。默认是允许在某些情况下滚动，但这和指定 SCROLL不完全相同。详情请见 注解。

WITH HOLD
WITHOUT HOLD

WITH HOLD指定该游标在创建它的事务提交 之后还能被继续使用。WITHOUT HOLD指定该游标 不能在创建它的事务之外使用。如果两者都没有指定，则默认为 WITHOUT HOLD。

query

用于提供该游标返回的行的SELECT或者 VALUES命令。

关键词BINARY、 INSENSITIVE和SCROLL 可以以任意顺序出现。

注解

普通游标以文本格式返回数据，这和SELECT产生的数据一样。 BINARY选项指定游标应该以二进制格式返回数据。这减少了服务 器和客户端的转换负担，但程序员需要付出更多工作来处理与平台相关的二进制 数据格式。例如，如果一个查询从一个整数列中返回一个值一，用一个默认游标 将得到一个字符串1，而使用一个二进制游标将得到该值的四字节 内部表示（big-endian 大端字节顺序）。

使用二进制游标时应该小心。很多应用（包括 psql）还没有准备好处理二进制游标， 它们仍然期待数据以文本格式到来。

> **注意**
>
> 当客户端应用使用“扩展查询”协议发出一个 FETCH命令，绑定协议消息会指定使用文本还是 二进制格式检索数据。这种选择会覆盖定义游标时指定的方式。因此 在使用扩展查询协议时，这样一个二进制游标的概念实际是被废弃的 — 任何游标都可以被视作文本或者二进制。

除非指定了WITH HOLD，这个命令创建的游标 只能在当前事务中使用。因此，没有WITH HOLD的DECLARE在事务块外是没有用的：游标只会生存 到该语句结束。因此如果这种命令在事务块之外被使用， PostgreSQL会报告一个错误。 定义事务块需要使用BEGIN和 COMMIT（或者ROLLBACK）。

如果指定了WITH HOLD并且创建游标的事务 成功提交，在同一个会话中的后续事务中还能够继续访问该游标（ 但是如果创建事务被中止，游标会被移除）。一个用 WITH HOLD创建的游标可以用一个显式的 CLOSE命令关闭，或者会话结束时它 也会被关闭。在当前的实现中，由一个被保持游标表示的行会被复 制到一个临时文件或者内存区域中，这样它们才会在后续事务中保 持可用。

当查询包括FOR UPDATE或FOR SHARE时， 不能指定WITH HOLD。

在定义一个将被反向取元组的游标时，应该指定SCROLL 选项。这是 SQL 标准所要求的。不过，为了和早期版本兼容， 如果游标的查询计划足够简单到支持它不需要额外的开销，PostgreSQL会允许在没有 SCROLL的情况下反向取元组。不过，建议应用开发者 不要依赖于从没有用SCROLL创建的游标中反向取 元组。如果指定了NO SCROLL，那么任何情况下都不 允许反向取元组。

当查询包括FOR UPDATE或FOR SHARE时， 也不允许反向取元组。因此在这种情况下不能指定SCROLL。

> **小心**
>
> 如果可滚动和WITH HOLD游标调用了任何不稳定的 函数（见第 38.7 节，它们可能给出预期之外 的结果。当重新取得一个之前取得过的行时，那些函数会被重新执行，这 可能导致得到与第一次不同的结果。对这类情况的一种变通方法是，声明 游标为WITH HOLD并且在从其中读取任何行之前提交 事务。这将强制该游标的整个输出被物化在临时存储中，这样针对每一行 只会执行一次不稳定函数。

如果游标的查询包括FOR UPDATE或者FOR SHARE，那么被返回的行会在它们第一次被取得时被锁定，这和带有 这些选项的常规SELECT命令一样。此外，被返回的 行将是最新的版本，因此这些选项提供了被 SQL 标准称为 “敏感游标”的等效体（把INSENSITIVE与 FOR UPDATE或者FOR SHARE一起指定是错误）。

> **小心**
>
> 如果游标要和UPDATE ... WHERE CURRENT OF或者 DELETE ... WHERE CURRENT OF一起使用，通常推荐 使用FOR UPDATE。使用FOR UPDATE可以 阻止其他会话在行被取得和被更新之间修改行。如果没有 FOR UPDATE，当行在游标创建后被更改后，一个后续的 WHERE CURRENT OF命令将不会产生效果。
>
> 另一个使用FOR UPDATE的原因是，如果没有它，当游标查询不符合 SQL 标准的“简单可更新”规则时，后续的 WHERE CURRENT OF可能会失败（特别地，该游标必须只引用一个 表并且没有使用分组或者ORDER BY）。不是简单可更新的游标可能 成功也可能不成功，这取决于计划选择的细节。因此在最坏的情况下，应用可能会 在测试时成功但是在生产中失败。如果指定了FOR UPDATE， 则保证游标是可更新的。
>
> 不把FOR UPDATE和WHERE CURRENT OF一起用的 主要原因是，需要游标时可滚动的或者对于后续更新不敏感（也就是说，继续显示 旧的数据）。如果这是你的需求，应密切关注安上述警示。

SQL 标准只对嵌入式SQL中的游标做出了规定。 PostgreSQL服务器没有为游标实现 OPEN语句。当游标被声明时就被认为已经 被打开。不过，ECPG（ PostgreSQL的嵌入式 SQL 预处理器） 支持标准 SQL 游标习惯，包括那些DECLARE 和OPEN语句。

你可以通过查询pg_cursors 系统视图可以看到所有可用的游标。

示例

声明一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

更多游标的例子请见FETCH。

兼容性

SQL 标准认为游标是否默认对底层数据的并发更新敏感是与实现相关的。在 PostgreSQL中，默认游标对此是不敏 感的，并且可以通过指定FOR UPDATE让它变得对此敏感。其他 产品的行为可能有所不同。

SQL 标准只允许在嵌入式SQL和模块中使用游标。 PostgreSQL允许以交互的方式使用游标。

二进制游标是一种PostgreSQL 扩展。

另见

CLOSE, FETCH, MOVE

DELETE

DELETE — 删除一个表的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

描述

DELETE从指定表中删除满足 WHERE子句的行。如果WHERE 子句没有出现，效果将会是删除表中的所有行。结果是一个合法的空表。

> **提示**
>
> TRUNCATE提供移除表中所有行的快速机制。

有两种方式可以使用数据库中其他表中包含的信息来删除一个表的行： 使用子选择或者在USING子句中指定额外的表。 哪种技术更合适取决于特定的环境。

可选的RETURNING子句导致DELETE 基于实际被删除的每一行计算并且返回值。任何使用被删除表列或者 USING中提到的其他表的列的表达式都可以被计算。 RETURNING列表的语法和SELECT的 输出列表语法相同。

要从表中删除行，你必须具有其上的DELETE特权， 以及USING子句中任何表以及其值在condition中被读取的表上的 SELECT特权。

参数

with_query

WITH子句允许你指定一个或者多个子查询，在 DELETE查询中可以用子查询的名字来引用它们。 详见第 7.8 和SELECT。

table_name

要从其中删除行的表名（可以是模式限定的）。如果在表名前指定 ONLY，只会从提到的表中删除匹配的行。如果没有指定 ONLY，还会删除该表的任何继承表中的匹配行。可选地， 可以在表名后面指定*来显式指定要包括继承表。

alias

目标表的一个别名。提供别名时，它会完全隐藏该表的真实名称。例如， 对于DELETE FROM foo AS f， DELETE语句的剩余部分都会用 f而不是 foo来引用该表。

using_list

一个表表达式的列表，它允许在WHERE条件中出现 来自其他表的列。这和SELECT语句的FROM 子句中指定 的表列表相似。例如，可以指定表的别名。除非想要进行自连接，否则不要在using_list 再写上目标表。

condition

一个返回boolean类型值的表达式。只有让这个 表达式返回true的行才将被删除。

cursor_name

要在WHERE CURRENT OF情况中使用的游标 的名称。最近一次从这个游标中取出的行将被删除。该游标 必须是DELETE的目标表上的非分组查询。 注意不能在使用WHERE CURRENT OF的同时 指定一个布尔条件。有关将游标用于 WHERE CURRENT OF的更多信息请见DECLARE。

output_expression

在每一行被删除后，会被DELETE计算并且返回的表达式。 该表达式可以使用table_name以及USING中的表的任何列。写成*可以返回所有列。

output_name

被返回列的名称。

输出

在成功完成时，一个DELETE命令会返回以下形式 的命令标签：

```
DELETE count
```

count是被删除行的数目。 注意如果有一个BEFORE DELETE触发器抑制删除，那么该数目 可能小于匹配condition 的行数。如果count为 0， 表示查询没有删除行（这并非一种错误）。

如果DELETE命令包含RETURNING子句， 则结果会与包含有RETURNING列表中定义的列和值的SELECT语句结果相似，这些结果是在被该命令删除的 行上计算得来。

注解

通过在USING子句中指定其他的表， PostgreSQL允许在 WHERE条件中引用其他表的列。例如，要 删除有一个给定制片人制作的所有电影，可以这样做：

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

这里实际发生的事情是在films和 producers之间进行连接，然后删除 所有成功连接的films行。这种语法不 属于标准。更标准的方式是：

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

在一些情况下，连接形式比子查询形式更容易书写或者执行更快。

示例

删除所有电影，但音乐剧除外：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清空表films：

```
DELETE FROM films;
```

删除已完成的任务，返回被删除行的明细：

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

删除tasks中游标c_tasks 当前位于其上的行：

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

兼容性

这个命令符合SQL标准，不过 USING和RETURNING子句是 PostgreSQL扩展，在 DELETE中使用WITH也是扩展。

又见

TRUNCATE

DISCARD

DISCARD — 抛弃会话状态

大纲

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

描述

DISCARD释放与一个数据库会话相关的内部资源。 这个命令有助于部分或者完全重置该会话的状态。有几个子命令来 释放不同类型的资源。DISCARD ALL变体把所有 其他形式都包含在内，并且还会重置额外的状态。

参数

PLANS

释放所有已缓存的查询计划，强制在下一次使用相关预备语句时 重新做计划。

SEQUENCES

丢弃所有已缓存的序列相关的状态，包括 currval()/lastval()信息 以及任何还未被nextval()返回的预分配的 序列值（预分配序列值的描述请见 CREATE SEQUENCE）；

TEMPORARY or TEMP

删除当前会话中创建的所有临时表。

ALL

释放与当前会话相关的所有临时资源并且把会话重置为初始状态。 当前这和执行以下语句序列的效果相同：

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD SEQUENCES;
DISCARD TEMP;
```

注解

DISCARD ALL不能在事务块内执行。

兼容性

DISCARD是一种 PostgreSQL扩展。

DO

DO — 执行一个匿名代码块

大纲

```
DO [ LANGUAGE lang_name ] code
```

描述

DO执行一个匿名代码块，或者换句话说 执行一个以一种过程语言编写的瞬时匿名函数。

代码块就好像是一个没有参数并且返回void的函数的函数体。 它会被在一次时间内解析并且执行。

可选的LANGUAGE子句可以卸载代码块之前或者之后。

参数

code

要被执行的过程语言代码。就像在 CREATE FUNCTION中一样，必须把它指定为一个 字符串。推荐使用一个美元引用的文本。

lang_name

编写该代码的过程语言的名称。如果省略，默认为plpgsql。

注解

要使用的过程语言必须已经用CREATE EXTENSION安装在 当前数据库中。默认已经安装了plpgsql，但是其他语言没有被 安装。

用户必须拥有该过程语言的USAGE特权，如果该语言 是不可信的则必须是一个超级用户。这和创建一个该语言的函数对 特权的要求相同。

如果在事务块中执行DO，过程代码则无法执行事务控制语句。只有在自己的事务中执行DO时，才允许使用事务控制语句。

例子

把模式public中所有视图上的所有特权授予 给角色webuser：

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
 quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

兼容性

SQL 标准中没有DO语句。

另见

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — 移除一种访问方法

大纲

```
DROP ACCESS METHOD [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

简介

DROP ACCESS METHOD移除一种现有的访问方法。只有超级用户能够删除访问方法。

参数

IF EXISTS

如果该访问方法不存在，则不会抛出错误。这种情况下会发出一个提示。

name

一种现有的访问方法的名称。

CASCADE

自动删除依赖于该访问方法的对象（例如操作符类、操作符族以及索引），并且接着删除所有依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该访问方法，则拒绝删除它。这是默认设置。

示例

删除访问方法heptree：

```
DROP ACCESS METHOD heptree;
```

兼容性

DROP ACCESS METHOD是一种PostgreSQL扩展。

另见

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — 移除一个聚集函数

大纲

```
DROP AGGREGATE [ IF EXISTS ] name ( aggregate_signature ) [, ...] [ CASCADE |
 RESTRICT ]
```

这里aggregate_signature是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

描述

DROP AGGREGATE移除一个现有的 聚集函数。要执行这个命令，当前用户必须是该聚集函数的拥有者。

参数

IF EXISTS

如果该聚集不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或VARIADIC。 如果被忽略，默认值是IN。

argname

一个参数的名称。注意DROP AGGREGATE 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数数据类型。

argtype

该聚集函数所操作的一个输入数据类型。要引用一个零参数的聚集函数，写 *来替代参数说明列表。要引用一个有序集聚集函数，在直接和 聚集参数说明之间写上ORDER BY。

CASCADE

自动删除依赖于该聚集函数的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该聚集函数，则拒绝删除它。这是默认值。

注解

ALTER AGGREGATE下描述了另一种引用有序集聚集的语法。

示例

要为类型integer移除聚集函数myavg：

```
DROP AGGREGATE myavg(integer);
```

要移除假想集聚集函数myrank，该函数接收一个排序列的　任意列表和直接参数的一个匹配的列表：

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

要在一个命令中删除多个聚合函数：

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

兼容性

在 SQL 标准中没有DROP AGGREGATE语句。

另见

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — 移除一个造型

大纲

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

描述

DROP CAST移除一个之前定义好的造型。

要能删除一个造型，你必须拥有源数据类型或目标数据类型。这也是　创建一个造型所要求的特权。

参数

IF EXISTS

　如果该造型不存在则不要抛出一个错误，而是发出一个提示。

source_type

　该造型的源数据类型的名称。

target_type

　该造型的目标数据类型的名称。

CASCADE
RESTRICT

　这些关键词没有任何效果，因为在造型上没有依赖性。

示例

要移除从类型text到类型int的造型：

```
DROP CAST (text AS int);
```

兼容性

DROP CAST命令符合 SQL 标准。

另见

CREATE CAST

DROP COLLATION

DROP COLLATION — 移除一个排序规则

大纲

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP COLLATION移除一个之前 定义好的排序规则。要能删除一个排序规则，你必须拥有它。

参数

IF EXISTS

如果该排序规则不存在则不要抛出一个错误，而是发出一个提示。

name

排序规则的名称。排序规则名称可以是模式限定的。

CASCADE

自动删除依赖于该排序规则的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该排序规则，则拒绝删除它。这是默认值。

示例

要删除名为german的排序规则：

```
DROP COLLATION german;
```

兼容性

除了IF EXISTS选项之外， DROP COLLATION命令符合 SQL标准。该选项是一个 PostgreSQL扩展。

另见

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — 移除一个转换

大纲

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP CONVERSION移除一个之前定义好的 转换。要能删除一个转换，你必须拥有该转换。

参数

IF EXISTS

如果该转换不存在则不要抛出一个错误，而是发出一个提示。

name

转换的名称。转换名称可以是模式限定的。

CASCADE
RESTRICT

这些关键词没有任何效果，因为在转换上没有依赖性。

示例

要删除名为myname的转换：

```
DROP CONVERSION myname;
```

兼容性

在 SQL 标准中没有DROP CONVERSION语句， 但是有一个DROP TRANSLATION语句。还有 对应的CREATE TRANSLATION语句，它与 PostgreSQL 中的CREATE CONVERSION 语句相似。

另见

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — 移除一个数据库

大纲

```
DROP DATABASE [ IF EXISTS ] name
```

描述

DROP DATABASE移除一个数据库。它会 移除该数据库的系统目录项并且删除包含数据的文件目录。它只能由数据库 拥有者执行。还有，当你或者任何其他人已经连接到目标数据库时，它不能 被执行（连接到postgres或者任何其他数据库来发出这 个命令）。

DROP DATABASE不能被撤销。请小心使用！

参数

IF EXISTS

如果该数据库不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据库的名称。

注解

DROP DATABASE不能在一个事务块内执行。

在连接到目标数据库时，这个命令不能被执行。因此，使用程序 dropdb会更方便，它是这个命令的一个包装器。

兼容性

SQL 标准中没有DROP DATABASE语句。

另见

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN — 移除一个域

大纲

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP DOMAIN移除一个域。只有域的拥有者 才能移除它。

参数

IF EXISTS

如果该域不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有域的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该域的对象（例如表列），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该域，则拒绝删除它。这是默认值。

示例

要移除域box：

```
DROP DOMAIN box;
```

兼容性

除了IF EXISTS选项，这个命令符合 SQL 标准。该选项 是一个PostgreSQL扩展。

另见

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — 移除一个事件触发器

大纲

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP EVENT TRIGGER移除一个已有的 事件触发器。要执行这个命令，当前用户必须是事件触发器的拥有者。

参数

IF EXISTS

如果该事件触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的事件触发器的名称。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

销毁触发器snitch：

```
DROP EVENT TRIGGER snitch;
```

兼容性

在 SQL 标准中没有DROP EVENT TRIGGER语句。

另见

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — 移除一个扩展

大纲

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP EXTENSION从数据库移除扩展。 删除一个扩展会导致其组成对象也被删除。

你必须拥有该扩展才能使用DROP EXTENSION。

参数

IF EXISTS

如果该扩展不存在则不要抛出一个错误，而是发出一个提示。

name

一个已安装扩展的名称。

CASCADE

自动删除依赖于该扩展的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该扩展（而不是它自己拥有的成员 对象和其他被列在同一个DROP命令中 的扩展），则拒绝删除它。这是默认值。

示例

要从当前数据库移除扩展hstore：

```
DROP EXTENSION hstore;
```

如果hstore的任何对象在该数据库库中 正在使用，例如有一个表的列是hstore类型，这个命令都将会失败。加上CASCADE选项可以强制 把这些依赖对象也移除。

兼容性

DROP EXTENSION是一个 PostgreSQL扩展。

另见

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — 移除一个外部数据包装器

大纲

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP FOREIGN DATA WRAPPER 移除一个已有的外部数据包装器。要执行这个命令，当前用户必须是该外部数据包装器的拥有者。

参数

IF EXISTS

如果该外部数据包装器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有外部数据包装器的名称。

CASCADE

自动删除依赖于该外部数据包装器的对象（例如服务器），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该外部数据包装器，则拒绝删除它。这是默认值。

示例

删除外部数据包装器dbi：

```
DROP FOREIGN DATA WRAPPER dbi;
```

兼容性

DROP FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句 是一个PostgreSQL扩展。

另见

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — 移除一个外部表

大纲

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP FOREIGN TABLE移除一个 外部表。只有一个外部表的拥有者才能移除它。

参数

IF EXISTS

如果该外部表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的外部表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该外部表的对象（例如视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该外部表，则拒绝删除它。这是默认值。

示例

要销毁两个外部表films 和distributors：

```
DROP FOREIGN TABLE films, distributors;
```

兼容性

这个命令符合 ISO/IEC 9075-9 (SQL/MED)，不过该标准只允许每个命令 中删除一个外部表并且没有IF EXISTS选项。该选项是一个 PostgreSQL扩展。

另见

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — 移除一个函数

大纲

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype
 [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

描述

DROP FUNCTION移除一个已有函数 的定义。要执行这个命令用户必须是该函数的拥有者。该函数的参数 类型必须被指定，因为多个不同的函数可能会具有相同的函数名和不 同的参数列表。

参数

IF EXISTS

如果该函数不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有函数的名称（可以是模式限定的）。 如果未指定参数列表，则该名称在其模式中必须是唯一的。

argmode

一个参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略， 则默认为IN。注意 DROP FUNCTION并不真正关心 OUT参数，因为决定函数的身份时只需要输入参数。 因此列出IN、INOUT和 VARIADIC参数足以。

argname

一个参数的名称。注意 DROP FUNCTION并不真正关心 参数名称，因为决定函数的身份时只需要参数的数据类型。

argtype

如果函数有参数，这是函数参数的数据类型（可以是模式限定的）。

CASCADE

自动删除依赖于该函数的对象（例如操作符和触发器），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该函数，则拒绝删除它。这是默认值。

示例

这个命令移除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

在一个命令中删除多个函数：

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

如果函数名称在其模式中是唯一的，则可以在不带参数列表的情况下引用它：

```
DROP FUNCTION update_employee_salaries;
```

请注意，这与

```
DROP FUNCTION update_employee_salaries();
```

不同，后者引用一个零个参数的函数，而第一个变体才可以引用具有任意数量参数的函数，包括零，只要该名称是唯一的。

兼容性

该命令符合SQL标准，使用这些PostgreSQL扩展：

- 该标准只允许每个命令删除一个函数。
- IF EXISTS选项
- 能够指定参数模式和名称

另见

CREATE FUNCTION, ALTER FUNCTION, DROP PROCEDURE, DROP ROUTINE

DROP GROUP

DROP GROUP — 移除一个数据库角色

大纲

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

描述

DROP GROUP现在是 DROP ROLE的一个别名。

兼容性

在 SQL 标准中没有DROP GROUP语句。

另见

DROP ROLE

DROP INDEX

DROP INDEX — 移除一个索引

大纲

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP INDEX从数据库系统中 移除一个已有的索引。要执行这个命令你必须是该索引的拥 有者。

参数

CONCURRENTLY

删除索引并且不阻塞在索引基表上的并发选择、插入、更新和删除操作。一个 普通的DROP INDEX会要求该表上的排他锁，这样会阻塞 其他访问直至索引删除完成。通过这个选项，该命令会等待直至冲突事务完成。

在使用这个选项时有一些需要注意的事情。只能指定一个索引名称，并且不支 持CASCADE选项（因此，一个支持UNIQUE或者 PRIMARY KEY约束的索引不能以这种方式删除）。还有，常规 的DROP INDEX命令可以在一个事务块内执行，而 DROP INDEX CONCURRENTLY不能。

IF EXISTS

如果该索引不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的索引的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该索引的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该索引，则拒绝删除它。这是默认值。

示例

这个命令将移除索引title_idx：

```
DROP INDEX title_idx;
```

兼容性

DROP INDEX是一个 PostgreSQL语言扩展。在 SQL 标准中没有提供索引。

另见

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — 移除一个过程语言

大纲

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP LANGUAGE移除一个之前注册的过程语言 的定义。你必须是一个超级用户或者该语言的拥有者才能使用 DROP LANGUAGE。

> **注意**
>
> 自PostgreSQL 9.1 起，大部分过程语言 已经被做成了“扩展”，因此应该用DROP EXTENSION而不是 DROP LANGUAGE删除。

参数

IF EXISTS

如果该语言不存在则不要抛出一个错误，而是发出一个提示。

name

一个已有过程语言的名称。为了向前兼容，这个名称可以用单引号包围。

CASCADE

自动删除依赖于该语言的对象（例如该语言中的函数），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该语言，则拒绝删除它。这是默认值。

示例

这个命令移除过程语言plsample：

```
DROP LANGUAGE plsample;
```

兼容性

在 SQL 标准中没有DROP LANGUAGE语句。

另见

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — 移除一个物化视图

大纲

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP MATERIALIZED VIEW删除一个 现有的物化视图。要执行这个命令你必须是该物化视图的拥有者。

参数

IF EXISTS

如果该物化视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的物化视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该物化视图的对象（例如其他物化视图或常规视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该物化视图，则拒绝删除它。这是默认值。

示例

这个命令将移除名为order_summary的物化视图：

```
DROP MATERIALIZED VIEW order_summary;
```

兼容性

DROP MATERIALIZED VIEW是一个 PostgreSQL扩展。

另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — 移除一个操作符

大纲

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type |
 NONE } ) [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP OPERATOR从数据库系统中 删除一个现有的操作符。要执行这个命令，你必须是该操作符的拥有者。

参数

IF EXISTS

如果该操作符不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有的操作符的名称（可以是模式限定的）。

left_type

该操作符左操作数的数据类型，如果没有左操作数就写 NONE。

right_type

该操作符右操作数的数据类型，如果没有右操作数就写 NONE。

CASCADE

自动删除依赖于该操作符的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该操作符，则拒绝删除它。这是默认值。

示例

为类型integer移除幂操作符 a^b：

```
DROP OPERATOR ^ (integer, integer);
```

为类型bit移除左一元按位补操作符 ~b：

```
DROP OPERATOR ~ (none, bit);
```

为类型bigint移除右一元阶乘操作符 x!：

```
DROP OPERATOR ! (bigint, none);
```

在一条命令中删除多个操作符：

```
DROP OPERATOR ~ (none, bit), ! (bigint, none);
```

兼容性

SQL 标准中没有DROP OPERATOR语句。

另见

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — 移除一个操作符类

大纲

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

描述

DROP OPERATOR CLASS删除一个现有的 操作符类。要执行这个命令，你必须是该操作符类的拥有者。

DROP OPERATOR CLASS不会删除任何被 该类引用的操作符或者函数。如果有索引依赖于该操作符类，你将需要指 定CASCADE来完成删除。

参数

IF EXISTS

如果该操作符类不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有的操作符类的名称（可以是模式限定的）。

index_method

该操作符类适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符类的对象（例如索引），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该操作符类，则拒绝删除它。这是默认值。

注解

DROP OPERATOR CLASS将不会删除包含该类的 操作符族，即使该族中已经没有任何成员（特别是由 CREATE OPERATOR CLASS隐式创建的族）。一个 空操作符族是无害的，但是为了整洁你可能希望用 DROP OPERATOR FAMILY移除该操作符族，或者 一开始就使用DROP OPERATOR FAMILY会更好。

示例

移除 B-树操作符类widget_ops：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果有任何使用该操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符类一起删除。

兼容性

SQL 标准中没有DROP OPERATOR CLASS语句。

另见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — 移除一个操作符族

大纲

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE |
 RESTRICT ]
```

描述

DROP OPERATOR FAMILY删除一个 现有的操作符族。要执行这个命令，你必须是该操作符族的拥有者。

DROP OPERATOR FAMILY包括删除 该族所包含的任何操作符类，但是它不会删除该族所引用的任何操作 符或函数。如果有任何依赖于该族中操作符类的索引存在，你将需要 指定CASCADE来完成删除。

参数

IF EXISTS

如果该操作符族不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有操作符族的名称（可以是模式限定的）。

index_method

该操作符族适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符族的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该操作符族，则拒绝删除它。这是默认值。

示例

移除 B-树操作符族float_ops：

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果有任何使用该族中操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符族一起删除。

兼容性

SQL 标准中没有DROP OPERATOR FAMILY 语句。

另见

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — 移除一个数据库角色拥有的数据库对象

大纲

```
DROP OWNED BY { name | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE |
 RESTRICT ]
```

描述

DROP OWNED删除当前数据库中被指定 角色之一拥有的所有对象。任何已被授予给给定角色在当前数据库中对 象上以及在共享对象（数据库、表空间）上的特权也将会被收回。

参数

name

其对象将被删除并且其特权将被收回的角色的名称。

CASCADE

自动删除依赖于受影响对象的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何其他数据库对象依赖于一个受影响的对象， 则拒绝删除一个角色所拥有的对象。这是默认值。

注解

DROP OWNED常常被用来为移除一个 或者多个角色做准备。因为DROP OWNED 只影响当前数据库中的对象，通常需要在包含将被移除角色所拥有的对象 的每一个数据库中都执行这个命令。

使用CASCADE选项可能导致这个命令递归去删除由其他 用户所拥有的对象。

REASSIGN OWNED命令是另一种选择，它可以把一个 或多个角色所拥有的所有数据库对象重新授予给其他角色。不过， REASSIGN OWNED不处理其他对象的特权。

角色所拥有的数据库、表空间将不会被移除。

更多讨论请见第 21.4 节

兼容性

DROP OWNED命令是一个 PostgreSQL扩展。

另见

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — 从一个表移除一条行级安全性策略

大纲

```
DROP POLICY [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

DROP POLICY从该表移除指定的策略。注意如果从 一个移除了最后一条策略并且该表的行级安全性仍被 ALTER TABLE启用，则默认的否定策略将被使用。 不管该表的策略存在与否，ALTER TABLE ... DISABLE ROW LEVEL SECURITY都可以被用来禁用一个表的行级安全性。

参数

IF EXISTS

如果该策略不存在也不抛出一个错误。这种情况下会发出一个提示。

name

要删除的策略的名称。

table_name

该策略所在的表的名称（可以被模式限定）。

CASCADE
RESTRICT

这些关键词不会产生效果，因为它们不依赖于策略。

例子

要在名为my_table上删除策略p1：

```
DROP POLICY p1 ON my_table;
```

兼容性

DROP POLICY是一种PostgreSQL扩展。

另见

CREATE POLICY, ALTER POLICY

DROP PROCEDURE

DROP PROCEDURE — 移除一个过程

大纲

```
DROP PROCEDURE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype
 [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

简介

DROP PROCEDURE移除一个现有过程的定义。为了执行这个命令，用户必须是该过程的拥有者。该过程的参数类型必须指定，因为可能存在多个不同的过程具有相同名称和不同参数列表。

参数

IF EXISTS

如果该过程不存在也不抛出一个错误。这种情况下会发出一个提示。

name

现有过程的名称（可以是被方案限定的）。如果没有指定参数列表，则该名称在其所属的方案中必须是唯一的。

argmode

参数的模式：IN或者VARIADIC。如果省略，默认为IN。

argname

参数的名称。注意，其实DROP PROCEDURE并不在意参数名称，因为只需要参数的数据类型来确定过程的身份。

argtype

该过程如果有参数，参数的数据类型（可以是被方案限定的）。

CASCADE

自动删除依赖于该过程的对象，然后接着删除依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该过程，则拒绝删除它。这是默认选项。

示例

```
DROP PROCEDURE do_db_maintenance();
```

兼容性

这个命令符合SQL标准，不过PostgreSQL做了这些扩展：

- 标准仅允许每个命令删除一个过程。
- IF EXISTS选项
- 指定参数模式和名称的能力

另见

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

DROP PUBLICATION

DROP PUBLICATION — 删除一个发布

大纲

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP PUBLICATION从数据库中删除一个现有的发布。

发布只能被它自己的所有者或超级用户删除。

参数

IF EXISTS

如果发布不存在，不要抛出一个错误。在这种情况下发出一个提示。

name

现有发布的名称。

CASCADE
RESTRICT

这些关键词没有任何作用，因为发布没有依赖关系。

示例

删除一个发布：

```
DROP PUBLICATION mypublication;
```

兼容性

DROP PUBLICATION是一个PostgreSQL 扩展。

又见

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — 移除一个数据库角色

大纲

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

描述

DROP ROLE移除指定的角色。要删除一个 超级用户角色，你必须自己就是一个超级用户。要删除一个非超级用户角 色，你必须具有CREATEROLE特权。

如果一个角色仍然被集簇中任何数据库中引用，它就不能被移除。如果尝试 移除将会抛出一个错误。在删除该角色前，你必须删除（或者重新授予所有 权）它所拥有的所有对象并且收回该已经授予给该角色的在其他对象上的特 权。REASSIGN OWNED和DROP OWNED 命令可以用于这个目的。更多讨论请见第 21.4 节

不过，没有必要移除涉及该角色的角色成员关系。 DROP ROLE会自动收回目标角色在其他角色中的成员 关系，以及其他角色在目标角色中的成员关系。其他角色不会被删除也不 会被影响。

参数

IF EXISTS

如果该角色不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的角色的名称。

注解

PostgreSQL包括一个程序dropuser具有和这个命令完全相同的功能（事实 上它会调用这个命令），但是该程序可以从 shell 运行。

示例

要删除一个角色：

```
DROP ROLE jonathan;
```

兼容性

SQL 标准定义了DROP ROLE， 但是它只允许一次删除一个角色并且它指定了和 PostgreSQL不同的特权需求。

另见

CREATE ROLE, ALTER ROLE, SET ROLE

DROP ROUTINE

DROP ROUTINE — 删除一个例程

大纲

```
DROP ROUTINE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype
 [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

简介

DROP ROUTINE删除一个现有例程的定义，它可以是一个聚集函数、一个普通函数或者过程。有关参数的描述、更多的示例以及进一步的细节请参考DROP AGGREGATE、DROP FUNCTION以及DROP PROCEDURE。

示例

删除类型integer的例程foo：

```
DROP ROUTINE foo(integer);
```

不管foo是一个聚集、函数或是一个过程，这个命令都能起作用。

兼容性

这个命令符合SQL标准，不过PostgreSQL做了下面这些扩展：

- 标准仅允许每个命令删除一个例程。
- IF EXISTS选项
- 指定参数模式和名称的能力
- 聚集函数是一种扩展。

另见

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

注意CREATE ROUTINE命令不存在。

DROP RULE

DROP RULE — 移除一个重写规则

大纲

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

DROP RULE删除一个重写规则。

参数

IF EXISTS

如果该规则不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的规则的名称。

table_name

该规则适用的表或视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该规则的对象，然后删除所有依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该规则，则拒绝删除它。这是默认值。

示例

要删除重写规则newrule：

```
DROP RULE newrule ON mytable;
```

兼容性

DROP RULE是一个 PostgreSQL语言扩展，整个 查询重写系统也是这样。

另见

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — 移除一个模式

大纲

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP SCHEMA从数据库中移除模式。

一个模式只能由其拥有者或一个超级用户删除。注意即使拥有者不拥有 该模式中的某些对象，也能删除该模式（以及所有含有的对象）。

参数

IF EXISTS

如果该模式不存在则不要抛出一个错误，而是发出一个提示。

name

一个模式的名称。

CASCADE

自动删除包含在该模式中的对象（表、函数等），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果该模式含有任何对象，则拒绝删除它。这是默认值。

注解

使用CASCADE选项可能会使这条命令移除除 指定模式之外其他模式中的对象。

示例

要从数据库中移除模式mystuff及其中 所包含的对象：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

DROP SCHEMA完全符合 SQL 标准， 不过该标准只允许在每个命令中删除一个模式并且没有 IF EXISTS选项。该选项是一个 PostgreSQL扩展。

另见

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — 移除一个序列

大纲

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP SEQUENCE移除序数生成器。 一个序列只能被其拥有者或超级用户删除。

参数

IF EXISTS

如果该序列不存在则不要抛出一个错误，而是发出一个提示。

name

一个序列的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该序列的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该序列，则拒绝删除它。这是默认值。

示例

要移除序列serial：

```
DROP SEQUENCE serial;
```

兼容性

DROP SEQUENCE符合 SQL标准，不过该标准只允许每个命令中删除一个 序列并且没有IF EXISTS选项。该选项是一个 PostgreSQL扩展。

另见

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — 移除一个外部服务器描述符

大纲

```
DROP SERVER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP SERVER移除一个现有的外部服务器 描述符。要执行这个命令，当前用户必须是该服务器的拥有者。

参数

IF EXISTS

如果该服务器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有服务器的名称。

CASCADE

自动删除依赖于该服务器的对象（例如用户映射），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该服务器，则拒绝删除它。这是默认值。

示例

如果一个服务器foo存在则删除它：

```
DROP SERVER IF EXISTS foo;
```

兼容性

DROP SERVER符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个 PostgreSQL扩展。

另见

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — 删除扩展统计

大纲

```
DROP STATISTICS [ IF EXISTS ] name [, ...]
```

描述

DROP STATISTICS删除数据库中的统计对象。 只有统计对象的所有者、模式的所有者或超级用户可以删除统计对象。

参数

IF EXISTS

name

要删除的统计对象的名称（可以有模式修饰）。

示例

删除不同模式中的两个统计对象，如果不存在时不会失败：

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

兼容性

SQL标准中没有DROP STATISTICS命令。

又见

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — 删除一个订阅

大纲

```
DROP SUBSCRIPTION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP SUBSCRIPTION删除数据库集群中的一个订阅。

只有超级用户才可以删除订阅。

如果订阅与复制槽相关联，就不能在事务块内部执行DROP SUBSCRIPTION。 （可以使用ALTER SUBSCRIPTION取消关联复制槽。）

参数

name

要删除的订阅的名称。

CASCADE
RESTRICT

这些关键词没有任何作用，因为订阅没有依赖关系。

注意

当删除与远程主机（正常状态）上的复制槽相关联的订阅时，DROP SUBSCRIPTION 将连接到远程主机，并尝试删除该复制槽，作为其操作的一部分。这是必要的， 以便释放远程主机上为订阅分配的资源。如果失败，因为远程主机不可访问， 或者因为远程复制槽不能被删除，或者复制槽不存在或从不存在， 则DROP SUBSCRIPTION命令将失败。要在这种情况下继续， 请执行ALTER SUBSCRIPTION ... SET (slot_name = NONE) 来解除复制槽与订阅的关联。之后，DROP SUBSCRIPTION 将不再尝试对远程主机执行任何操作。请注意，如果远程复制槽仍然存在， 则应手动删除该插槽；否则将继续保留WAL，最终可能导致磁盘空间不足。 另见第 31.2.1 节

如果订阅与复制槽相关联，那么不能在事务块内部执行DROP SUBSCRIPTION。

示例

删除一个订阅：

```
DROP SUBSCRIPTION mysub;
```

兼容性

DROP SUBSCRIPTION是一个PostgreSQL 扩展。

又见

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — 移除一个表

大纲

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP TABLE从数据库移除表。只有表拥有者、 模式拥有者和超级用户能删除一个表。要清空一个表中的行但是不销毁该表， 可以使用DELETE或者TRUNCATE。

DROP TABLE总是移除目标表的任何索引、规则、 触发器和约束。不过，要删除一个被视图或者另一个表的外键约束所引用的表， 必须指定CASCADE（CASCADE将会把依赖的视图 也完全移除，但是对于外键它将只移除外键约束，而完全不会移除其他表）。

参数

IF EXISTS

如果该表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该表的对象（例如视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

示例

要销毁两个表films和 distributors：

```
DROP TABLE films, distributors;
```

兼容性

这个命令符合 SQL 标准，不过该标准只允许每个命令删除一个表并且没有 IF EXISTS选项。该选项是一个 PostgreSQL扩展。

另见

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — 移除一个表空间

大纲

```
DROP TABLESPACE [ IF EXISTS ] name
```

描述

DROP TABLESPACE从系统中移除一个表空间。

一个表空间只能被其拥有者或超级用户删除。在一个表空间能被删除前，其中 必须没有任何数据库对象。即使当前数据库中没有对象正在使用该表空间，也 可能有其他数据库的对象存在于其中。还有，如果该表空间被列在任何活动会 话的temp_tablespaces设置中，DROP也可能会失败，因为可能有临时文件存在其中。

参数

IF EXISTS

如果该表空间不存在则不要抛出一个错误，而是发出一个提示。

name

一个表空间的名称。

注解

DROP TABLESPACE不能在一个事务块内执行。

示例

要从系统移除表空间mystuff：

```
DROP TABLESPACE mystuff;
```

兼容性

DROP TABLESPACE是一个 PostgreSQL扩展。

另见

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — 移除一个文本搜索配置

大纲

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP TEXT SEARCH CONFIGURATION 删除一个现有的文本搜索配置。要执行这个命令，你必须是该配置的拥有者。

参数

IF EXISTS

如果该文本搜索配置不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索配置的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索配置的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该文本搜索配置，则拒绝删除它。这是默认值。

示例

移除文本搜索配置my_english：

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

如果有任何在to_tsvector调用中引用该配置的 索引存在，这个命令都不会成功。增加CASCADE 可以把这类索引与该文本搜索配置一起删除。

兼容性

SQL 标准中没有DROP TEXT SEARCH CONFIGURATION 语句。

另见

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — 移除一个文本搜索字典

大纲

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP TEXT SEARCH DICTIONARY删除一个 现有的文本搜索字典。要执行这个命令，你必须是该字典的拥有者。

参数

IF EXISTS

如果该文本搜索字典不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索字典的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索字典的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该文本搜索字典，则拒绝删除它。这是默认值。

示例

移除文本搜索字典english：

```
DROP TEXT SEARCH DICTIONARY english;
```

如果有任何使用该字典的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与字典一起删除。

兼容性

SQL 标准中没有DROP TEXT SEARCH DICTIONARY 语句。

另见

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — 移除一个文本搜索解析器

大纲

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP TEXT SEARCH PARSER删除一个 现有的文本搜索解析器。你必须作为一个超级用户来使用这个命令。

参数

IF EXISTS

如果该文本搜索解析器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索解析器的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该文本搜索解析器，则拒绝删除它。这是默认值。

示例

移除文本搜索解析器my_parser：

```
DROP TEXT SEARCH PARSER my_parser;
```

如果有任何使用该解析器的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与解析器一起删除。

兼容性

SQL 标准中没有DROP TEXT SEARCH PARSER 语句。

另见

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — 移除一个文本搜索模板

大纲

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP TEXT SEARCH TEMPLATE删除一个 现有的文本搜索模板。你必须作为一个超级用户来使用这个命令。

参数

IF EXISTS

如果该文本搜索模板不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索模板的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索模板的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该文本搜索模板，则拒绝删除它。这是默认值。

示例

移除文本搜索模板thesaurus：

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

如果有任何使用该模板的文本搜索字典存在，这个命令都不会成功。增加 CASCADE可以把这类字典与模板一起删除。

兼容性

SQL 标准中没有DROP TEXT SEARCH TEMPLATE 语句。

另见

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — 移除转换

大纲

```
DROP TRANSFORM [ IF EXISTS ] FOR type_name LANGUAGE lang_name [ CASCADE |
 RESTRICT ]
```

简介

DROP TRANSFORM移除一个之前定义的转换。

为了删除一种转换，你必须拥有该类型和语言。这些同样也是创建转换所需要的 特权。

参数

IF EXISTS

如果该转换不存在也不要抛出一个错误。这种情况下会发出一个提示。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

CASCADE

自动删除依赖于该转换的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该转换，则拒绝删除它。这是默认行为。

示例

要删除用于类型hstore和语言plpythonu 的转换：

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

兼容性

这种形式的DROP TRANSFORM是一种 PostgreSQL扩展。详见 CREATE TRANSFORM。

另见

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER — 移除一个触发器

大纲

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

DROP TRIGGER移除一个现有的触发器定义。 要执行这个命令，当前用户必须是触发器基表的拥有者。

参数

IF EXISTS

如果该触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的触发器的名称。

table_name

定义了该触发器的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

销毁表films上的触发器 if_dist_exists：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

PostgreSQL中的 DROP TRIGGER语句与 SQL 标准不 兼容。在 SQL 标准中，不同表上也不能有同名的触发器，因此其 命令是简单的DROP TRIGGER name.

另见

CREATE TRIGGER

DROP TYPE

DROP TYPE — 移除一个数据类型

大纲

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP TYPE移除一种用户定义的数据类型。 只有一个类型的拥有者才能移除它。

参数

IF EXISTS

如果该类型不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据类型的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该类型的对象（例如表列、函数、操作符），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

示例

要移除数据类型box：

```
DROP TYPE box;
```

兼容性

这个命令类似于 SQL 标准中的对应命令，但IF EXISTS子句 是一个PostgreSQL扩展。但要注意 PostgreSQL中 CREATE TYPE命令的很大部分以及数据类 型扩展机制都与 SQL 标准不同。

另见

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — 移除一个数据库角色

大纲

```
DROP USER [ IF EXISTS ] name [, ...]
```

描述

DROP USER现在是 DROP ROLE另一种写法。

兼容性

DROP USER语句是一个 PostgreSQL扩展。SQL 标准把 用户的定义留给具体实现自行解释。

另见

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — 移除一个用于外部服务器的用户映射

大纲

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
 SERVER server_name
```

描述

DROP USER MAPPING从外部服务器移除 一个已有的用户映射。

一个外部服务器的拥有者可以为该服务器的任何用户删除用户映射。如果 该服务器上的USAGE特权被授予了一个用户，它也能删除 用于它们自己的用户名的用户映射。

参数

IF EXISTS

如果该用户映射不存在则不要抛出一个错误，而是发出一个提示。

user_name

该映射的用户名。CURRENT_USER 和USER匹配当前用户的名称。PUBLIC 被用来匹配系统中所有现存和未来的用户名。

server_name

用户映射的服务器名。

示例

删除一个用户映射bob（服务器foo），如果它存在：

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

兼容性

DROP USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个 PostgreSQL扩展。

另见

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — 移除一个视图

大纲

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP VIEW删除一个现有的视图。要执行 这个命令你必须是该视图的拥有者。

参数

IF EXISTS

如果该视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该视图的对象（例如其他视图），然后删除所有 依赖于那些对象的对象（见第 5.13 节。

RESTRICT

如果有任何对象依赖于该视图，则拒绝删除它。这是默认值。

示例

这个命令将移除名为kinds的视图：

```
DROP VIEW kinds;
```

兼容性

这个命令符合 SQL 标准，不过该标准只允许在每个命令中删除一个视图 并且没有IF EXISTS选项。该选项是一个 PostgreSQL扩展。

另见

ALTER VIEW, CREATE VIEW

END

END — 提交当前事务

大纲

```
END [ WORK | TRANSACTION ]
```

描述

END提交当前事务。所有该事务做的更改便得 对他人可见并且被保证发生崩溃时仍然是持久的。这个命令是一种 PostgreSQL扩展，它等效于 COMMIT。

参数

WORK
TRANSACTION

可选关键词，它们没有效果。

注解

使用ROLLBACK可以中止一个事务。

当不在一个事务中时发出END没有危害，但是会 产生一个警告消息。

示例

要提交当前事务并且让所有更改持久化：

```
END;
```

兼容性

END是一种 PostgreSQL扩展，它提供和 COMMIT等效的功能，后者在 SQL 标准中指定。

另见

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — 执行一个预备语句

大纲

```
EXECUTE name [ ( parameter [, ...] ) ]
```

描述

EXECUTE被用来执行一个之前准备好的语句。 由于预备语句只在会话期间存在，该预备语句必须在当前会话中由一个更早 执行的PREPARE语句所创建。

如果创建预备语句的PREPARE语句指定了一些参数， 必须向EXECUTE语句传递一组兼容的参数，否则会 发生错误。注意（与函数不同）预备语句无法基于其参数的类型或者数量重载。 在一个数据库会话中，预备语句的名称必须唯一。

更多创建和使用预备语句的信息请见PREPARE。

参数

name

要执行的预备语句的名称。

parameter

给预备语句的参数的实际值。这必须是一个能得到与该参数数据类型（ 在预备语句创建时决定）兼容的值的表达式。

输出

EXECUTE返回的命令标签是预备语句的命令标签而不是 EXECUTE。

例子

在PREPARE文档的例子小节给出了例子。

兼容性

SQL 标准包括了一个EXECUTE语句， 但是只被用于嵌入式 SQL。这个版本的 EXECUTE语句也用了一种有点不同的语法。

另见

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — 显示一个语句的执行计划

大纲

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

这里 option可以是：

```
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

描述

这个命令显示PostgreSQL计划器为提供的语句所生成的执行计划。该执行计划会显示将怎样扫描语句中引用的表 — 普通的顺序扫描、索引扫描等等 — 以及在引用多个表时使用何种连接算法来把来自每个输入表的行连接在一起。

显示中最重要的部分是估计出的语句执行代价，它是计划器对于该语句要运行多久的猜测（以任意的代价单位度量，但是习惯上表示取磁盘页面的次数）。事实上会显示两个数字：在第一行能被返回前的启动代价，以及返回所有行的总代价。对于大部分查询来说总代价是最重要的，但是在一些情景中（如EXISTS中的子查询），计划器将选择更小的启动代价来代替最小的总代价（因为因为执行器将在得到一行后停止）。此外，如果你用一个LIMIT子句限制返回行的数量，计划器会在终端代价之间做出适当的插值来估计到底哪个计划是真正代价最低的。

ANALYZE选项导致该语句被实际执行，而不仅仅是被计划。那么实际的运行时间统计会被显示出来，包括在每个计划结点上花费的总时间（以毫秒计）以及它实际返回的行数。这对观察计划器的估计是否与实际相近很有用。

> **重要**
>
> 记住当使用了ANALYZE选项时语句会被实际执行。尽管EXPLAIN将丢弃SELECT所返回的任何输出，照例该语句的其他副作用还是会发生。如果你希望在INSERT、UPDATE、DELETE、CREATE TABLE AS或者EXECUTE上使用EXPLAIN ANALYZE而不希望它们影响你的数据，可以使用下面的方法：
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

只有ANALYZE和VERBOSE选项能被指定，并且必须按照上述的顺序，不要把选项列表放在圆括号内。在PostgreSQL 9.0 之前，只支持没有圆括号的语法。我们期望所有新的选项将只在圆括号语法中支持。

参数

ANALYZE

执行命令并且显示实际的运行时间和其他统计信息。这个参数默认被设置为FALSE。

VERBOSE

显示关于计划的额外信息。特别是：计划树中每个结点的输出列列表、模式限定的表和函数名、总是把表达式中的变量标上它们的范围表别名，以及总是打印统计信息被显示的每个触发器的名称。这个参数默认被设置为FALSE。

COSTS

包括每一个计划结点的估计启动和总代价，以及估计的行数和每行的宽度。这个参数默认被设置为TRUE。

BUFFERS

包括缓冲区使用的信息。特别是：共享块命中、读取、标记为脏和写入的次数、本地块命中、读取、标记为脏和写入的次数、以及临时块读取和写入的次数。一次命中表示避免了一次读取，因为需要的块已经在缓存中找到了。共享块包含着来自于常规表和索引的数据，本地块包含着来自于临时表和索引的数据，而临时块包含着在排序、哈希、物化计划结点和类似情况中使用的短期工作数据。脏块的数量表示被这个查询改变的之前未被修改块的数量，而写入块的数量表示这个后台在查询处理期间从缓存中替换出去的脏块的数量。为一个较高层结点显示的块数包括它的所有子结点所用到的块数。在文本格式中，只会打印非零值。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为FALSE。

TIMING

在输出中包括实际启动时间以及在每个结点中花掉的时间。反复读取系统时钟的负荷在某些系统上会显著地拖慢查询，因此在只需要实际的行计数而不是实际时间时，把这个参数设置为FALSE可能会有用。即便用这个选项关闭结点层的计时，整个语句的运行时间也总是会被度量。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为TRUE。

SUMMARY

在查询计划之后包含摘要信息（例如，总计的时间信息）。当使用ANALYZE 时默认包含摘要信息，但默认情况下不包含摘要信息，但可以使用此选项启用摘要信息。 使用EXPLAIN EXECUTE中的计划时间包括从缓存中获取计划所需的时间 以及重新计划所需的时间（如有必要）。

FORMAT

指定输出格式，可以是 TEXT、XML、JSON 或者 YAML。非文本输出包含和文本输出格式相同的信息，但是更容易被程序解析。这个参数默认被设置为TEXT。

boolean

指定被选中的选项是否应该被打开或关闭。可以写TRUE、ON或1来启用选项，写FALSE、OFF或0禁用它。boolean值也能被忽略，在这种情况下会假定值为TRUE。

statement

你想查看其执行计划的任何SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE、CREATE TABLE AS或者CREATE MATERIALIZED VIEW AS语句。

输出

这个命令的结果是为statement选中的计划的文本描述，可能还标注了执行统计信息。第 14.1 描述了所提供的信息。

注解

为了允许PostgreSQL查询计划器在优化查询时能做出合理的知情决策，查询中用到的所有表的pg_statistic数据应该能保持为最新。通常这个工作会由autovacuum daemon负责自动完成。但是如果一个表最近在内容上有大量的改变，我们可能需要做一次手动的ANALYZE而不是等待 autovacuum 捕捉这些改变。

为了执行计划中每个结点的运行时间开销，当前的In order to measure the run-time cost of each node in the execution plan, the current implementation of EXPLAIN ANALYZE实现为查询执行增加了 profiling overhead 。这样，在一个查询上运行EXPLAIN ANALYZE有时候比正常执行该查询要慢很多。开销的量取决于该查询的性质，以及使用的平台。最坏的情况会发生在那些自身执行时间很短的结点上，以及在那些具有相对较慢的有关时间的操作系统调用的机器上。

例子

有一个具有单个integer列和 10000 行的表，要显示在其上的一个简单查询的计划：

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

这里有同样一个查询的 JSON 输出格式：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

如果有一个索引，并且我们使用了一个带有可索引WHERE条件的查询，EXPLAIN可能会显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

这里是同一查询的 YAML 格式：

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

XML 格式我们留给读者做练习。

这里是去掉了代价估计的同样一个计划：

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

这里是一个使用聚集函数的查询的查询计划例子：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

这里是一个使用EXPLAIN EXECUTE显示预备查询的执行计划的例子：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161
 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8)
 (actual time=0.039..0.091 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

当然，这里显示的有关数字取决于表涉及到的实际内容。还要注意这些数字甚至选中的查询策略，可能在PostgreSQL的不同版本之间变化，因为计划器可能被改进。此外，ANALYZE命令使用随机采样来估计数据统计。因此，在一次新的ANALYZE运行之后，代价估计可能会改变，即便是表中数据的实际分布没有改变也是如此。

兼容性

在 SQL 标准中没有定义EXPLAIN语句。

参见

ANALYZE

FETCH

FETCH — 使用游标从查询中检索行

大纲

```
FETCH [ direction [ FROM | IN ] ] cursor_name
```

其中 direction 可以为空或者以下之一：

```
    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

描述

FETCH从之前创建的一个游标中检索行。

游标具有一个相关联的位置，FETCH会用到该位置。 游标位置可能会位于查询结果的第一行之前、结果中任意行之上或者 结果的最后一行之后。在被创建时，游标被定位在第一行之前。在取出 一些行后，该游标被定位在最近被取出的行上。如果 FETCH运行超过了可用行的末尾，则该游标会被定位 在最后一行之后（如果向后取，则是第一行之前）。 FETCH ALL或者FETCH BACKWARD ALL将总是让游标被定位于最后一行之后或者第一行之前。

NEXT、PRIOR、FIRST、 LAST、ABSOLUTE、RELATIVE 形式会在适当移动游标后取出一行。如果没有这样一行，将返回一个空 结果，并且视情况将游标定位在第一行之前或者最后一行之后。

使用FORWARD和BACKWARD的形式会在 向前移动或者向后移动的方向上检索指定数量的行，然后将游标定位在 最后返回的行上（如果count超过可用的行数，则定位 在所有行之后或者之前）。

RELATIVE 0、FORWARD 0以及 BACKWARD 0都会请求检索当前行但不移动游标，也就是 重新取最近被取出的行。 只要游标没有被定位在第一行之前或者最后一行 之后，这种操作都会成功，否则不会返回任何行。

> **注意**
>
> 这个页面描述在 SQL 命令层面上对游标的使用。如果想要在 PL/pgSQL函数中使用游标，规则会有所不同 — 请见第 43.7.3 节

参数

direction

direction 定义获取方向以及要取得的行数。它可以是下列之一：

NEXT

取出下一行。如果省略direction，这将是默认值。

PRIOR

取出当前位置之前的一行。

FIRST

取出该查询的第一行（和ABSOLUTE 1相同）。

LAST

取出该查询的最后一行（和ABSOLUTE -1相同）。

ABSOLUTE count

取出该查询的第count个行，如果count为负则是从尾部开始取出 第abs(count)个行。如果 count超出范围，将定位在第一行 之前或者最后一行之后。特别地，ABSOLUTE 0 会定位在第一行之前。

RELATIVE count

取出第count个后继行，如果 count为负 则是取出前面的第abs(count)个行。RELATIVE 0重新取出当前行（如果有）。

count

取出接下来的count行（和 FORWARD count相同）。

ALL

取出所有剩余的行（和FORWARD ALL相同）。

FORWARD

取出下一行（和NEXT相同）。

FORWARD count

取出接下来的count行。 FORWARD 0重新取出当前行。

FORWARD ALL

取出所有剩下的行。

BACKWARD

取出当前行前面的一行（和PRIOR相同）。

BACKWARD count

取出前面的count行（反向扫描）。 BACKWARD 0会重新取出当前行。

BACKWARD ALL

取出所有当前位置之前的行（反向扫描）。

count

count 是一个可能带有符号的整数常量，它决定要取得的位置或者行数。对于FORWARD和BACKWARD情况，指定一个负的 count等效于改变 FORWARDhe BACKWARD的意义。

cursor_name

一个已打开游标的名称。

输出

如果成功完成，FETCH命令返回一个下面形式的命令标签：

```
FETCH count
```

count是取得的行数（可能 为零）。注意在psql中，命令标签将不会实际显示， 因为psql会显示被取得的行。

注解

如果想要使用FETCH的任意变体而不使用带有正计数的 FETCH NEXT或者FETCH FORWARD， 应该用SCROLL声明游标。对于简单查询， PostgreSQL将允许从不带 SCROLL的游标中反向取得行，但最好不要依赖这种行为。 如果游标被声明为带有SCROLL，则不允许反向取得。

用ABSOLUTE取行并不比用相对移动快多少：不管则样， 底层实现都必须遍历所有的中间行。用负绝对值获取的情况更糟：必须读到 查询尾部来找到最后一行，并且接着从那里反向开始遍历。不过，回卷到查询 的开始（正如FETCH ABSOLUTE 0）是很快的。

DECLARE被用来定义游标。使用 MOVE可改变游标位置而不检索数据。

示例

下面的例子用一个游标遍历一个表：

```
BEGIN WORK;

-- 建立一个游标：
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- 在游标 liahona 中取出前 5 行：
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- 取出前面一行：
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08
```

```
-- 关闭游标并且结束事务：
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准只定义FETCH在嵌入式 SQL 中使用。 这里描述的FETCH变体返回数据时就好像数据是 一个SELECT结果，而不是被放在主变量中。 除这一点之外，FETCH完全向上兼容于 SQL 标准。

涉及FORWARD和BACKWARD的 FETCH形式，以及形式FETCH count和FETCH ALL（其中FORWARD是隐式的）都是 PostgreSQL扩展。

SQL 标准只允许FROM在游标名之前。使用 IN的选项或者完全省去它们是一种扩展。

另见

CLOSE, DECLARE, MOVE

GRANT

GRANT — 定义访问特权

大纲

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [ [ argmode ]
 [ arg_name ] arg_type [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name
 [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

其中role_specification可以是：

```
    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

```
GRANT role_name [, ...] TO role_name [, ...] [ WITH ADMIN OPTION ]
```

描述

GRANT命令由两种基本的变体：一种授予在一个数据库对象（表、列、视图、外部表、序列、数据库、外部数据包装器、外部服务器、函数、过程、过程语言、模式或表空间）上的特权，另一个授予一个角色中的成员关系。这些变体在很多方面都相似，但是也有很多不同，所以还是得分别描述它们。

在数据库对象上 GRANT

这种GRANT命令的变体将一个数据库对象上的指定特权交给一个或多个角色。如果有一些已经被授予，这些特权会被加入到它们之中。

还有一个选项可以授予一个或多个模式中同种类型的所有对象上的特权。这种功能当前只支持表、序列、函数和过程。ALL TABLES还影响视图和外部表，就像特定对象GRANT命令。ALL FUNCTIONS还影响聚集函数，但不影响过程，就像特定对象的GRANT命令。

关键词PUBLIC指示特权要被授予给所有角色，包括那些可能稍后会被创建的角色。PUBLIC可以被认为是一个被隐式定义的总是包含所有角色的组。任何特定角色都将具有直接授予给它的特权、授予给它作为成员所在的任何角色的特权以及被授予给PUBLIC的特权。

如果指定了WITH GRANT OPTION，特权的接收者可以接着把它授予给其他人。如果没有授权选项，接收者就不能这样做。授权选项不能被授予给PUBLIC。

没有必要把权限授予给一个对象的拥有者（通常就是创建该对象的用户），　因为拥有者默认具有所有的特权（不过拥有者可能为了安全选择撤回一些 它们自己的特权）。

删除一个对象或者以任何方式修改其定义的权力是不被当作一个可授予特权的，它被固化在拥有者中，并且不能被授予和撤回（不过，相似地效果可以通过授予或者撤回在拥有该对象的角色中的成员关系来实现，见下文）。拥有者也隐式地拥有该对象的所有授权选项。

PostgreSQL　会把某些类型的对象上的默认特权授予给PUBLIC。默认在表、表列、序列、外部数据包装器、外部服务器、大对象、方案或表空间上不会有特权会被授予给PUBLIC。对于其他类型的对象，被授予给PUBLIC的默认特权是下面这些：数据库上的CONNECT以及TEMPORARY（创建临时表）；函数和过程上的EXECUTE特权；语言和数据类型（包括域）的USAGE特权。当然，对象拥有者可以REVOKE默认和专门授予的特权（为了最好的安全性，应该在创建该对象的同一个事务中发出REVOKE，这样其他用户就没有时间窗口使用该对象）。还有，这些初始的默认特权设置可以使用ALTER DEFAULT PRIVILEGES命令修改。

可能的特权是：

SELECT

允许从指定表、视图或序列的任何列或者列出的特定列进行SELECT。还允许使用COPY TO。在UPDATE或DELETE中引用已有列值时也需要这个特权。对于序列，这个特权也允许使用currval函数。对于大对象，这个特权允许读取对象。

INSERT

允许INSERT一个新行到指定表中。如果列出了特定的列，只有这些列能在INSERT命令中被赋值（其他列将因此收到默认值）。还允许COPY FROM。

UPDATE

允许对指定表、视图或序列的任何列或者列出的特定列进行UPDATE（实际上，任何非平凡的UPDATE命令也会要求SELECT特权，因为它必须引用表列来判断哪些行要被更新或者为列计算新值）。除SELECT特权之外，SELECT ... FOR UPDATE以及SELECT ... FOR SHARE也要求至少一列上的这个特权。对于序列，这个特权允许使用nextval和setval函数。对于大对象，这个特权允许写入或者截断对象。

DELETE

允许从指定的表中DELETE一行（实际上，任何非平凡的DELETE命令也将要求SELECT特权，因为它必须引用表列来判断要删除哪些行）。

TRUNCATE

允许在指定的表上TRUNCATE。

REFERENCES

允许创建引用指定表或者表的指定列的外键约束（见CREATE TABLE语句）。

TRIGGER

允许在指定的表上创建触发器（见CREATE TRIGGER语句）。

CREATE

对于数据库，允许在其中创建新方案和订阅。

对于方案，允许在其中创建新的对象。要重命名一个已有对象，你必须拥有该对象并且具有所在方案的这个特权。

对于表空间，允许在其中创建表、索引和临时文件，并且允许创建使用该表空间作为默认表空间的数据库（注意撤回这个特权将不会更改现有对象的放置位置）。

CONNECT

允许用户连接到指定数据库。在连接开始时会检查这个特权（除了检查由pg_hba.conf施加的任何限制之外）。

TEMPORARY
TEMP

允许在使用指定数据库时创建临时表。

EXECUTE

允许使用指定的函数或者过程以及使用实现在函数之上的任何操作符。这是唯一一种适用于函数和过程的特权。FUNCTION语法对聚集函数也有效。另外，可使用ROUTINE引用函数、聚集函数或者过程而无需操心它究竟是什么。

USAGE

对于过程语言，允许使用指定的语言创建函数。这是适用于过程语言的唯一一种特权类型。

对于模式，允许访问包含在指定模式中的对象（假定这些对象的拥有特权要求也满足）。本质上这允许被授权者在模式中“查阅”对象。如果没有这个权限，还是有可能看到对象名称，例如通过查询系统表。还有，在撤回这个权限之后，现有后端可能有语句之前已经执行过这种查阅，因此这不是一种阻止对象访问的完全安全的方法。

对于序列，这种特权允许使用currval和nextval函数。

对于类型和域，这种特权允许用该类型或域来创建表、函数和其他模式对象（注意这不能控制类型的一般“用法”，例如出现在查询中的该类型的值。它只阻止基于该类型创建对象。该特权的主要目的是控制哪些用户在一个类型上创建了依赖，这能够阻止拥有者以后更改该类型）。

对于外部数据包装器，这个特权让被授权者能够创建新的使用该外部数据包装器的服务器。

对于服务器，这个特权让被授权者使用该服务器创建外部表。被授权者还可以创建、修改或删除与该服务器相关的属于该用户的用户映射。

ALL PRIVILEGES

一次授予所有的可用特权。在PostgreSQL中，PRIVILEGES关键词是可选的，但是在严格的SQL中是要求它的。

其他命令所要求的特权会被列在相应命令的参考页中。

角色上的 GRANT

GRANT命令的这种变体把一个角色中的成员关系授予一个或者多个其他角色。一个角色中的成员关系是有意义的，因为它会把授予给一个角色的特权带给该角色的每一个成员。

如果指定了WITH ADMIN OPTION，成员接着可以把该角色中的成员关系授予给其他用户，也可以撤回该角色中的成员关系。如果没有管理选项，普通用户就不能做这些工作。一个角色不被认为持有自身的WITH ADMIN OPTION，但是它可以从一个会话用户匹配该角色的数据库会话中授予或撤回自身中的成员关系。数据库超级用户能够授予或撤回任何角色中任何人的成员关系。具有CREATEROLE特权的角色能够授予或者撤回任何非超级用户角色中的成员关系。

和特权的情况不同，一个角色中的成员关系不能被授予PUBLIC。还要注意这种形式的命令不允许噪声词GROUP。

注解

REVOKE命令被用来撤回访问特权。

从PostgreSQL 8.1 开始，用户和组的概念已经被统一到一种单一类型的实体（被称为一个角色）。因此不再需要使用关键词GROUP来标识一个被授权者是一个用户或者一个组。在该命令中仍然允许GROUP，但是它只是一个噪音词而已。

如果一个用户持有特定列或者其所在的整个表的特权， 该用户可以在该列上执行SELECT、INSERT等命令。在表层面上授予特权 然后对一列撤回该特权将不会按照你希望的运作： 表级别的授权不会受到列级别操作的影响。

当一个对象的非拥有者尝试GRANT该对象上的特权，如果该用户在该对象上什么特权都不拥有，该命令将立刻失败。只要有一些特权可用，该命令将继续，但是它将只授予那些用户具有授权选项的特权。如果不持有授权选项，GRANT ALL PRIVILEGES形式将发出一个警告消

息。而如果不持有命令中特别提到的任何特权的授权选项，其他形式将会发出一个警告（原则上这些语句也适用于对象拥有者，但是由于拥有者总是被视为持有所有授权选项，因此这种情况不会发生）。

需要注意的是，数据库超级用户可以访问所有对象而不管对象特权的设置。这可与 Unix 系统中的root权力相提并论。对于root来说，除非绝对必要，使用一个超级用户来操作是不明智的。

如果一个超级用户选择发出一个GRANT或者REVOKE命令，该命令将被执行，好像它是由被影响对象的拥有者发出的一样。特别地，通过这样一个命令授予的特权将好像是由对象拥有者授予的一样（对于角色成员关系，该成员关系好像是由该角色本身授予的一样）。

GRANT以及REVOKE也可以由一个不是受影响对象拥有者的角色完成，不过该角色是拥有该对象的角色的一个成员，或者是在该对象上持有特权的WITH GRANT OPTION的角色的一个成员。在这种情况下，特权将被记录为由实际拥有该对象的角色授予或者是由持有特权的WITH GRANT OPTION的角色授予。例如，如果表t1被角色g1拥有，u1是它的一个成员，那么u1可以把t1上的特权授予给u2，但是那些特权将好像是直接由g1授予的。角色g1的任何其他成员可以稍后撤回它们。

如果执行GRANT的角色间接地通过多于一条角色成员关系路径持有所需的特权，将不会指定哪一个包含它的角色将被记录为完成了该授权。在这样的情况中，最好使用SET ROLE来成为你想用其做GRANT的特定角色。

授予一个表上的权限不会自动地扩展权限给该表使用的任何序列，包括绑定在SERIAL列上的序列。序列上的权限必须被独立设置。

使用psql的\dp命令可获得表和列上现有的特权的信息。例如：

```
=> \dp mytable
                              Access privileges
 Schema |  Name   | Type  |   Access privileges   | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

\dp显示的项解释如下：

```
角色名=xxxx -- 被授予给一个角色的特权
       =xxxx -- 被授予给 PUBLIC 的特权

           r -- SELECT ("读")
           w -- UPDATE ("写")
           a -- INSERT ("追加")
           d -- DELETE
           D -- TRUNCATE
           x -- REFERENCES
           t -- TRIGGER
           X -- EXECUTE
           U -- USAGE
           C -- CREATE
           c -- CONNECT
           T -- TEMPORARY
     arwdDxt -- ALL PRIVILEGES （对于表，对其他对象会变化）
           * -- 用于前述特权的授权选项

       /yyyy -- 授予该特权的角色
```

用户miriam在创建了表mytable并且执行了下面的操作后会看到上述例子的显示：

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

对于非表对象，有其他的\d命令可显示它们的特权。

如果一个给定对象的“Access privileges”列为空，表示该对象具有默认的特权（也就是，它的特权列为空）。默认特权总是包括拥有者的所有特权，并且如前所述根据对象类型可以包括一些PUBLIC的特权。一个对象上的第一个GRANT或者REVOKE将实例化默认特权（例如，产生{miriam=arwdDxt/miriam}）并且接着为每一个指定请求修改它们。类似地，显示在“Column access privileges”列中的项只用于带有非默认特权的列（注意，为了这个目的“default privileges”总是表示该对象类型的内建默认特权）。一个特权已经被一个ALTER DEFAULT PRIVILEGES命令影响的对象将与一个显式特权项一起显示，该项包括ALTER的效果）。

注意拥有者的隐式授权选项没有在访问特权显示中被标出。当授权选项被显式地授予给某人时，只会出现一个*。

例子

把表films上的插入特权授予给所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

把视图kinds上的所有可用特权授予给用户manuel：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

注意虽然上述语句被一个超级用户或者kinds的拥有者执行时确实会授予所有特权，但是当由其他人执行时将只会授予那些执行者拥有授权选项的权限。

把角色admins中的成员关系授予给用户joe：

```
GRANT admins TO joe;
```

兼容性

根据 SQL 标准，ALL PRIVILEGES中的PRIVILEGES关键词是必须的。SQL 标准不支持在每个命令中设置超过一个对象上的特权。

PostgreSQL允许一个对象拥有者 撤回它们拥有的普通特权：例如，一个表拥有者可以通过撤回其自身拥有 的INSERT、UPDATE、DELETE 和TRUNCATE特权让该表对它们自己只读。根据 SQL 标准 这是不可能发生的。原因在于PostgreSQL 认为拥有者的特权是由拥有者授予给它们自己的，因此它们也能够撤回它们。 在 SQL 标准中，拥有者的特权是有一个假设的实体“_SYSTEM”所授予。 由于不是“_SYSTEM”，拥有者就不能撤回这些权力。

根据 SQL 标准，授权选项可以被授予给PUBLIC， PostgreSQL 只支持把授权选项授予给角色。

SQL 标准提供了其他对象类型上的USAGE特权：字符集、排序规则、翻译。

在 SQL 标准中，序列只有一个USAGE特权，它控制NEXT VALUE FOR表达式的使用，该表达式等效于 PostgreSQL 中的函数nextval。序列的特权SELECT和UPDATE是 PostgreSQL 扩展。应用序列的USAGE特权到currval函数也是一个 PostgreSQL 扩展（该函数本身也是）。

数据库、表空间、模式和语言上的特权都是PostgreSQL扩展。

参见

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — 从一个外部服务器导入表定义

大纲

```
IMPORT FOREIGN SCHEMA remote_schema
    [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
    FROM SERVER server_name
    INTO local_schema
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

简介

IMPORT FOREIGN SCHEMA创建表示存在于 外部服务器上的表的外部表。新外部表将由发出该命令的用户所拥有并且用 匹配远程表的正确的列定义和选项创建。

默认情况下，存在于外部服务器上一个特定模式中的所有表和视图都会被导入。 根据需要，表的列表可以被限制到一个指定的子集，或者可以排除特定的表。 新外部表都被创建在一个必须已经存在的目标模式中。

要使用IMPORT FOREIGN SCHEMA，用户必 须具有外部服务器上的USAGE特权以及在目标模式上的 CREATE特权。

参数

remote_schema

要从哪个远程模式导入。一个远程模式的特定含义依赖于所使用的外部数据包装器。

LIMIT TO (table_name [, ...])

只导入匹配给定表名之一的外部表。外部模式中其他的表将被忽略。

EXCEPT (table_name [, ...])

把指定的外部表排除在导入之外。除了列在这里的表之外，外部模式 中存在的所有表都将被导入。

server_name

要从哪个外部服务器导入。

local_schema

被导入的外部表将创建在其中的模式。

OPTIONS (option 'value' [, ...])

要在导入期间使用的选项。允许使用的选项名称和值与每一个外部数据包装器 有关。

示例

从服务器film_server上的远程模式foreign_films 中导入表定义，把外部表创建在本地模式films中：

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

同上，但是只导入两个表actors和 directors（如果存在）：

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

兼容性

IMPORT FOREIGN SCHEMA命令符合 SQL标准，不过OPTIONS 子句是一种PostgreSQL扩展。

另见

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — 在一个表中创建新行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER} VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
 | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

其中 conflict_target 可以是以下之一：

```
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ]
 [ opclass ] [, ...] ) [ WHERE index_predicate ]
    ON CONSTRAINT constraint_name
```

并且 conflict_action 是以下之一：

```
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT }
 [, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

描述

INSERT将新行插入到一个表中。我们可以 插入一个或者更多由值表达式指定的行，或者插入来自一个查询的零行 或者更多行。

目标列的名称可以以任意顺序列出。如果没有给出列名列表，则有两种确定 目标列的可能性。第一种是以被声明的顺序列出该表的所有列。另一种可能 性是，如果VALUES 子句或者query只提 供N个列，则以被声明的顺序列出该表的前 N列。VALUES 子句或者 query提供的值会被从左至右关联到这些显式或者隐式 给出的目标列。

每一个没有出现在显式或者隐式列列表中的列都将被默认填充，如果为该列 声明过默认值则用默认值填充，否则用空值填充。

如果任意列的表达式不是正确的数据类型，将会尝试自动类型转换。

ON CONFLICT可以用来指定发生唯一约束或者排除约束 违背错误时的替换动作（见下文的ON CONFLICT 子句）。

可选的RETURNING子句让INSERT根据 实际被插入（如果使用了ON CONFLICT DO UPDATE子句， 可能是被更新）的每一行来计算和返回值。这主要用来获取由默认值提供 的值，例如一个序列号。不过，允许在其中包括使用该表列的任何表达式。 RETURNING列表的语法与SELECT的输出 列表的相同。只有被成功地插入或者更新的行才将被返回。例如，如果一行被锁定但由于不满足ON CONFLICT DO UPDATE ... WHERE clause condition没有被更新，该行将 不被返回。

为了向表中插入，你必须具有其上的INSERT特权。 如果存在ON CONFLICT DO UPDATE子句，还要求该表上 的UPDATE特权。

如果一个列列表被指定，你只需要其中的列上的INSERT 特权。类似地，在指定了ON CONFLICT DO UPDATE时，你只 需要被列出要更新的列上的UPDATE特权。不过， ON CONFLICT DO UPDATE还要求其值被 ON CONFLICT DO UPDATE表达式或者 condition使用的列上的SELECT特权。

使用RETURNING子句需要RETURNING中提到的所有列的 SELECT权限。 如果使用query子句从查询中插入行， 则当然需要对查询中使用的任何表或列具有SELECT权限。

参数

插入

这个小节介绍了在只插入新行时可以使用的参数。 专门用于ON CONFLICT子句的 参数会单独介绍。

with_query

WITH子句允许指定一个或者更多子查询，在 INSERT查询中可以用名称引用这些子查询。详见 第 7.8 节以及SELECT。

query （SELECT语句）也可以包含一个 WITH子句。在这种情况下 query中可以引用 两组with_query，但是第二个优先级更 高（因为它被嵌套更近）。

table_name

一个已有表的名称（可以被模式限定）。

alias

table_name 的替补名称。当提供了一个别名时，它会完全隐藏掉表的实际名称。 当ON CONFLICT DO UPDATE的目标是一个被排除的表时这特别有用，因为那将被当作表示要被插入行的特殊表的名称。

column_name

名为table_name的表中的一个列 的名称。如有必要，列名可以用一个子域名或者数组下标限定（指向 一个组合列的某些列中插入会让其他域为空）。当用 ON CONFLICT DO UPDATE引用一列时，不要在一个 目标列的说明中国包括表名。例如， INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1是非法的（这遵循UPDATE 的一般行为）。

OVERRIDING SYSTEM VALUE

如果没有这个子句，为定义为GENERATED ALWAYS的标识列指定一个明确的值（不是DEFAULT）就是一种错误。这个子句推翻了这种限制。

OVERRIDING USER VALUE

如果指定这个子句，则会忽略提供给定义为GENERATED BY DEFAULT的标识列的值，并且应用默认的由序列生成的值。

例如，当在表之间拷贝值时，这个子句有能派上用场。INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1将从tbl1中拷贝所有在tbl2中不是标识列的列，而tbl2中标识列的值将由与tbl2关联的序列产生。

DEFAULT VALUES

所有列都将被其默认值填充（例如这种形式下不允许OVERRIDING子句）。

expression

要赋予给相应列的表达式或者值。

DEFAULT

相应的列将被其默认值填充。

query

提供要被插入行的查询（SELECT语句）。 其语法描述请参考SELECT语句。

output_expression

在每一行被插入或更新后由INSERT命令计算并且返回的 表达式。该表达式可以使用table_name 指定的表中的任何列。写成*可返回被插入或更新行的所有列。

output_name

要用于被返回列的名称。

ON CONFLICT 子句

可选的ON CONFLICT子句为出现唯一性违背或排除 约束违背错误时提供另一种可供选择的动作。对于每一个要插入的行， 不管是插入进行下去还是由conflict_target 指定的一个仲裁者约束或者索引被违背，都会 采取可供选择的conflict_action。 ON CONFLICT DO NOTHING简单地把避免插入行。 ON CONFLICT DO UPDATE则会 更新与要插入的行冲突的已有行。

conflict_target可以执行 唯一索引推断。在执行推断时，它由一个或者多个 index_column_name 列或者 index_expression 表达式以及一个可选的 index_predicate构成。所有刚好包含 conflict_target指定的列/表达式的table_name唯一索引（不管顺序）都 会被推断为（选择为）仲裁者索引。如果指定了 index_predicate，它 必须满足仲裁者索引（也是推断过程的一个进一步的要求）。注意这意味着如果 有一个满足其他条件的非部分唯一索引（没有谓词的唯一索引）可用，它将被 推断为仲裁者（并且会被ON CONFLICT使用）。如果推断 尝试不成功，则会发生一个错误。

ON CONFLICT DO UPDATE保证一个原子的 INSERT或者 UPDATE结果。在没有无关错误的前提下，这两种 结果之一可以得到保证，即使在很高的并发度也能保证。这也可以被称作 UPSERT — “UPDATE 或 INSERT”。

conflict_target

通过选择仲裁者索引来指定哪些行与 ON CONFLICT在其上采取可替代动作的行相冲突。要么执行唯一索引推断，要么显式命名一个 约束。对于ON CONFLICT DO NOTHING来说， 它对于指定一个conflict_target是可选的。 在被省略时，与所有有效约束（以及唯一索引）的冲突都会被处理。对于 ON CONFLICT DO UPDATE，必须 提供一个conflict_target。

conflict_action

conflict_action指定一个可替换的 ON CONFLICT动作。它可以是 DO NOTHING，也可以是一个指定在冲突情况下 要被执行的UPDATE动作细节的DO UPDATE子句。ON CONFLICT DO UPDATE中的SET和 WHERE子句能够使用该表的名称（或者别名） 访问现有的行，并且可以用特殊的被排除 表访问要插入的行。这个动作要求被排除 列所在目标表的任何列上的SELECT特权。

注意所有行级BEFORE INSERT触发器的效果都会 反映在被排除值中，因为那些效果可能会 让该行避免被插入。

index_column_name

一个table_name列 的名称。它被用来推断仲裁者索引。它遵循CREATE INDEX格式。这要求 index_column_name 上的SELECT特权。

index_expression

和index_column_name类似，但是 被用来推断出现在索引定义中的table_name列（非简单列）上的 表达式。遵循CREATE INDEX格式。这要求 任何出现在index_expression中的列上的 SELECT特权。

collation

指定时，强制相应的index_column_name或 index_expression 使用一种特定的排序规则以便在推断期间能被匹配上。通常 会被省略，因为排序规则通常不会影响约束违背的发生。遵循 CREATE INDEX格式。

opclass

指定时，强制相应的index_column_name或 index_expression 使用特定的操作符类以便在推断期间能被匹配上。通常会被省略， 因为相等语义在一种类型的操作符类 之间都是等价的，或者因为足以信任已定义的唯一索引具有适当的 相等定义。遵循CREATE INDEX格式。

index_predicate

用于允许推断部分唯一索引。任何满足该谓词（不一定需要真的是 部分索引）的索引都能被推断。遵循CREATE INDEX格式。这要求任何出现在index_predicate中的列上的SELECT特权。

constraint_name

用名称显式地指定一个仲裁者约束， 而不是推断一个约束或者索引。

condition

一个能返回boolean值的表达式。只有让这个表达式返回 true的行才将被更新，不过在采用 ON CONFLICT DO UPDATE动作时所有的行都会被锁定。 注意condition会被最后计算，即一个冲突 被标识为要更新的候选对象之后。

注意不支持把排除约束作为ON CONFLICT DO UPDATE的 仲裁者。在所有的情况中，只支持NOT DEFERRABLE约束和 唯一索引作为仲裁者。

带有ON CONFLICT DO UPDATE子句的 INSERT是一种“确定性的” 语句。这表明不允许该命令影响任何单个现有行超过一次，如果发生则会 发生一个基数违背错误。要插入的行不应该在仲裁者索引或约束所限制的 属性上相重复。

注意，当前不支持用分区表上的INSERT的ON CONFLICT DO UPDATE子句更新冲突行的分区键，因为那样会让行移动到新的分区中。

> **提示**
>
> 使用唯一索引推断通常比使用ON CONFLICT ON CONSTRAINT constraint_name直接提名一个约束更好。当底层索引被以重叠方式替换成另一个或多或少等效的索引时，推断将能继续正确地工作，例如在删除要被替换的索引之前使用CREATE UNIQUE INDEX ... CONCURRENTLY。

输出

成功完成时，INSERT命令会返回以下形式的命令标签：

INSERT oid count

count是被插入或更新的行数。 如果count正好为 1 并且 目标表具有 OID，那么 oid就是分配给被插入行的 OID。这个单一行必须已经被插入而不是被更新。 否则oid为零。

如果INSERT命令包含RETURNING子句， 其结果会类似于包含RETURNING列表中定义的列和值的 SELECT语句，这些结果是由该命令在被插入或更新行上 计算得到。

注解

如果指定的表是一个分区表，每一行都会被路由到合适的分区并且插入其中。如果指定的表是一个分区，如果输入行之一违背该分区的约束则将发生错误。

示例

向films中插入一行：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在这个例子中，len列被省略并且因此会具有默认值：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

这个例子为日期列使用DEFAULT子句而不是指定一个值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

插入一个完全由默认值构成的行：

```
INSERT INTO films DEFAULT VALUES;
```

用多行VALUES语法插入多个行：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

这个例子从表tmp_films中获得一些行插入到表 films中，两个表具有相同的列布局：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

这个例子插入数组列：

```
-- 为 noughts-and-crosses 游戏创建一个空的 3x3 棋盘
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- 实际上可以不用上面例子中的下标
```

```
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

向表distributors中插入一行，返回由 DEFAULT子句生成的序号：

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

增加为 Acme Corporation 管理账户的销售人员的销量，并且把整个被 更新的行以及当前时间记录到一个日志表中：

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

酌情插入或者更新新的 distributor。假设已经定义了一个唯一索引来约束 出现在did列中的值。注意，特殊的 excluded表被用来引用原来要插入的值：

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

插入一个 distributor，或者在一个被排除的行（具有一个匹配约束的列或者 会让行级前（或者后）插入触发器引发的列的行）存在时不处理要插入的行。 例子假设已经定义了一个唯一触发器来约束出现在did列 中的值：

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

酌情插入或者更新新的 distributor。例子假设已经定义了一个唯一触发器来 约束出现在did列中的值。WHERE子句被用 来限制实际被更新的行（不过，任何没有被更新的已有行仍将被锁定）：

```
-- 根据一个特定的 ZIP 编码更新 distributors
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- 直接在语句中命名一个约束（使用相关的索引来判断是否做
-- DO NOTHING 动作）
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

如果可能就插入新的 distributor，否则DO NOTHING。 例子假设已经定义了一个唯一索引，它约束让is_active 布尔列为true的行子集上did列中的值：

```
-- 这个语句可能推断出一个在 "did" 上带有谓词 "WHERE is_active"
-- 的部分唯一索引，但是它可能也只是使用了 "did" 上的一个常规唯一约束
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

兼容性

INSERT符合 SQL 标准，不过 RETURNING子句是一种 PostgreSQL扩展， 在 INSERT中使用WITH也是， 用ON CONFLICT指定一个替代动作也是扩展。 还有，标准不允许省略列名列表但不通过 VALUES子句或者query填充 所有列的情况。

SQL标准指定只有存在一个总是会生成值的标识列时才能指定OVERRIDING SYSTEM VALUE。而PostgreSQL在任何情况下都允许这个子句，并且在不适用时会忽略它。

query子句可能的限制在 SELECT有介绍。

LISTEN

LISTEN — 监听一个通知

大纲

```
LISTEN channel
```

描述

LISTEN在名为channel的通知频道上将当前会话注册为一个监听者。如果当前会话已经被注册为这个通知频道的一个监听者，则什么也不会发生。

只要命令NOTIFY channel被调用（不管是在这个会话还是在另一个连接到同一数据库的会话中），所有当前正在该通知频道上监听的会话都会被通知，并且每一个会话将会接着通知连接到它的客户端应用。

可以使用UNLISTEN命令在一个给定通知频道上反注册一个会话。当会话结束时，它的监听注册会被自动清除。

一个客户端应用检测通知事件的必用方法取决于它使用的PostgreSQL应用编程接口。如果使用libpq库，应用会将LISTEN作为一个普通 SQL 命令发出，并且接着必须周期性地调用函数PQnotifies来查看是否接收到通知事件。其他诸如libpgtcl的接口提供了更高层次上的处理通知事件的方法。事实上，通过使用libpgtcl应用程序员甚至不必直接发出LISTEN或UNLISTEN。更多细节可参阅所使用的接口的文档。

NOTIFY包含了使用LISTEN和NOTIFY的更广泛的讨论。

参数

channel

一个通知频道的名称（任意标识符）。

注解

LISTEN在事务提交时生效。如果在一个后来被回滚的事务中执行了LISTEN或UNLISTEN，被监听的通知频道集合不会变化。

一个已经执行了LISTEN的事务不能为两阶段提交做准备。

例子

从psql中配置并执行一个监听/通知序列

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

兼容性

在 SQL 标准中没有LISTEN语句。

参见

NOTIFY, UNLISTEN

LOAD

LOAD — 载入一个共享库文件

大纲

LOAD 'filename'

描述

这个命令把一个共享库文件载入到PostgreSQL服务器 的地址空间中。如果该文件已经被载入，这个命令什么也不会做。只要调用 包含 C 函数的共享库文件中的一个函数，这些共享库文件就会被自动载入。 因此，一次显式的LOAD通常只在载入一个通过 “钩子”修改服务器行为而不是提供一组函数的库时需要。

库文件名通常只是一个裸文件名，在服务器的库搜索路径（由 dynamic_library_path设置）中寻找。或者， 它可以作为完整的路径名称给出。无论哪种情况，平台的标准共享库文件扩展名都可以省略。 有关于此话题的更多信息可见第 38.10.1 节

非超级用户只能把LOAD应用在位于 $libdir/plugins/中的库文件 — 指定的 filename 必须正好以该字符串开始（确保在那里只安装了 “安全的”库是数据库管理员的责任）。

兼容性

LOAD是一种 PostgreSQL扩展。

另见

CREATE FUNCTION

LOCK

LOCK — 锁定一个表

大纲

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

其中 lockmode 可以是以下之一：

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

描述

LOCK TABLE获得一个表级锁，必要时会 等待任何冲突锁被释放。如果指定了NOWAIT， LOCK TABLE不会等待以获得想要的锁： 如果它不能立刻得到，该命令会被中止并且发出一个错误。一旦获取到， 该锁会被在当前事务中一直持有（没有UNLOCK TABLE命令，锁总是在事务结束时被释放）。

当一个视图被锁定时，出现在该视图定义查询中的所有关系也将被使用同样的锁模式递归地锁住。

在为引用表的命令自动获取锁时， PostgreSQL总是尽可能使用最不严格的 锁模式。提供LOCK TABLE是用于想要更严格 的锁定的情况。例如，假设一个应用运行一个READ COMMITTED 隔离级别的事务， 并且需要确保一个表中的数据在该事务的期间保持稳定。要实现这个目的， 必须在查询之前在表上获得SHARE锁模式。这将阻止并发的 数据更改并且确保该表的后续读操作会看到已提交数据的一个稳定视图， 因为SHARE锁模式与写入者所要求的ROW EXCLUSIVE锁有冲突，并且你的 LOCK TABLE name IN SHARE MODE 语句将等待，直到任何并发持有ROW EXCLUSIVE模式锁的持有者提交或者回滚。因此，一旦得到锁， 就不会有未提交的写入还没有解决。更进一步，在释放该锁之前，任何 人都不能开始。

要在运行在REPEATABLE READ或SERIALIZABLE 隔离级别的事务中得到类似的效果，你必须在执行任何 SELECT或者数据修改语句之前执行 LOCK TABLE语句。一个 REPEATABLE READ或者SERIALIZABLE事务的 数据视图将在它的第一个SELECT或者数据修改语句开始 时被冻结。在该事务中稍后的一个LOCK TABLE仍将阻止并发写 — 但它不会确保该事务读到的东西对应于最新的已提交值。

如果一个此类事务正要修改表中的数据，那么它应该使用 SHARE ROW EXCLUSIVE锁模式来取代 SHARE模式。这会保证一次只有一个此类事务运行。如果 不用这种模式，死锁就可能出现：两个事务可能都要求 SHARE模式，并且都不能获得 ROW EXCLUSIVE模式来真正地执行它们的更新（注意一个 事务所拥有的锁不会冲突，因此一个事务可以在它持有SHARE 模式时获得ROW EXCLUSIVE模式 — 但是如果有其他 人持有SHARE模式时则不能）。为了避免死锁，确保所有的 事务在同样的对象上以相同的顺序获得锁，并且如果在一个对象上涉及多 种锁模式，事务应该总是首先获得最严格的那种模式。

更多关于锁模式和锁策略的信息可见第 13.3 节

参数

name

要锁定的一个现有表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，只有该表会被锁定。如果没有指定了 ONLY，该表和它所有的后代表（如果有）都会被锁定。可选 地，在表名后指定*来显式地表示把后代表包括在内。

命令LOCK TABLE a, b;等效于 LOCK TABLE a; LOCK TABLE b;。这些表会被按照在 LOCK TABLE中指定的顺序一个一个 被锁定。

lockmode

锁模式指定这个锁和哪些锁冲突。锁模式在第 13.3 节描述。

如果没有指定锁模式，那儿将使用最严格的模式ACCESS EXCLUSIVE。

NOWAIT

指定LOCK TABLE不等待任何冲突锁被释放： 如果所指定的锁不能立即获得，那么事务就会中止。

注解

LOCK TABLE ... IN ACCESS SHARE MODE要求目标表上的SELECT特权。LOCK TABLE ... IN ROW EXCLUSIVE MODE要求目标表上的INSERT、UPDATE、DELETE或TRUNCATE特权。所有其他形式的LOCK要求表级UPDATE、DELETE或TRUNCATE特权。

在该视图上执行锁定的用户必须具有该视图上相应的特权。此外视图的拥有者必须拥有底层基关系上的相关特权，但是执行锁定的用户不需要底层基关系上的任何权限。

LOCK TABLE在一个事务块外部没有用处：锁将只保持到语句完成。因此如果在一个事务块外部使用了LOCK，PostgreSQL会报告一个错误。使用BEGIN和COMMIT（或者ROLLBACK）定义一个事务块。

LOCK TABLE只处理表级锁，因此涉及到 ROW的模式名称在这里都是不当的。这些模式名称应该通常 被解读为用户在被锁定表中获取行级锁的意向。还有， ROW EXCLUSIVE模式是一个可共享的表锁。记住就 LOCK TABLE而言，所有的锁模式都具有相同的语义， 只有模式的冲突规则有所不同。关于如何获取一个真正的行级锁的信息， 请见SELECT参考文档中的第 13.3.2 和锁定子句。

示例

在将要向一个外键表中执行插入时在主键表上获得一个 SHARE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果记录没有被返回就做 ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在将要执行一次删除操作前在主键表上取一个 SHARE ROW EXCLUSIVE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

在 SQL 标准中没有LOCK TABLE，SQL 标准中使用 SET TRANSACTION指定事务上的并发层次。PostgreSQL也支持这样做，详见 SET TRANSACTION。

除ACCESS SHARE、ACCESS EXCLUSIVE和 SHARE UPDATE EXCLUSIVE锁模式之外， PostgreSQL锁模式和 LOCK TABLE语法与 Oracle中的兼容。

MOVE

MOVE — 定位一个游标

大纲

```
MOVE [ direction [ FROM | IN ] ] cursor_name
```

其中direction可以为空或者以下之一：

```
    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

描述

MOVE重新定位一个游标而不检索任何数据。 MOVE的工作完全像 FETCH命令，但是它只定位游标并且不返回行。

用于MOVE命令的参数和 FETCH命令的一样，可参考 FETCH。

输出

成功完成时，MOVE命令返回的命令标签形式是

```
MOVE count
```

count是一个 具有同样参数的FETCH命令会返回的 行数（可能为零）。

示例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- 跳过前 5 行：
MOVE FORWARD 5 IN liahona;
MOVE 5

-- 从游标 liahona 中取第 6 行：
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
```

```
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- 关闭游标 liahona 并且结束事务：
CLOSE liahona;
COMMIT WORK;
```

兼容性

在 SQL 标准中没有MOVE语句。

另见

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — 生成一个通知

大纲

```
NOTIFY channel [ , payload ]
```

描述

NOTIFY命令发送一个通知事件以及一个可选的“载荷”字符串给每个正在监听的客户端应用，这些应用之前都在当前数据库中为指定的频道名执行过LISTEN channel。通知对所有用户都可见。

NOTIFY为访问同一个PostgreSQL数据库的进程集合提供了一种简单的进程间通讯机制。伴随着通知可以发送一个载荷字符串，通过使用数据库中的表从通知者向监听者传递额外的数据，也可以构建用于传输结构化数据的高层机制。

由一个通知事件传递给客户端的信息包括通知频道名称、发出通知的会话的服务器进程PID以及载荷字符串，如果载荷字符串没有被指定则它为空字符串。

将在一个给定数据库以及其他数据库中使用的频道名称由数据库设计者定义。通常，频道名称与数据库中某个表的名称相同，并且通知事件其实就意味着：“我改变了这个表，来看看改了什么吧”。但是NOTIFY和LISTEN命令并未强制这样的关联。例如，一个数据库设计者可以使用几个不同的频道名称来标志一个表上的不同种类的改变。另外，载荷字符串可以被用于多种不同的情况。

当NOTIFY被用来标志对一个特定表的改变时， 一种有用的编程技巧是把NOTIFY 放在一个由表更新触发的语句触发器中。在这种方式中，每当表被改变时 都将自动发生通知，并且应用程序员不可能会忘记发出通知。

NOTIFY以一些重要的方式与 SQL 事务互动。首先，如果一个NOTIFY在一个事务内执行，在事务被提交之前，该通知事件都不会被递送。这是合适的，因为如果该事务被中止，所有其中的命令都将不会产生效果，包括NOTIFY在内。但如果期望通知事件被立即递送，那这种行为就会令人不安。其次，如果一个监听会话收到了一个通知信号而它正在一个事务中，在该事务完成（提交或者中止）之前，该通知事件将不会被递送给它连接的客户端。同样，原因在于如果一个通知在一个事务内被递送且该事务后来被中止，我们会希望该通知能以某种方式被撤销 — 但是服务器一旦把通知发送给客户端就无法“收回它”。因此通知事件只能在事务之间递送。其中的要点是把NOTIFY用作实时信号的应用应该让它们事务尽可能短小。

如果从同一个事务多次用相同的载荷字符串对同一个频道名称发送通知，数据库服务器能决定只递送一个单一的通知。另一方面，带有不同载荷字符串的通知将总是作为不同的通知被递送。类似地，来自不同事务的通知将不会被折叠成一个通知。除了丢弃后来生成的重复通知实例之外，NOTIFY保证来自同一个事务的通知按照它们被发送的顺序被递送。还可以保证的是，来自不同事务的消息会按照其事务被提交的顺序递送。

一个执行NOTIFY的客户端自己也同时在监听同一个通知频道是很常见的事。在这种情况下，和所有其他监听会话一样，它会取回一个通知事件。根据应用的逻辑，这可能导致无用的工作，例如从自己刚刚写入的一个表中读出相同的更新。可以通过关注发出通知的服务器进程PID（在通知事件消息中提供）与自己的会话PID（可以从libpq得到）是否相同来避免这种额外的工作。当两者相同时，该通知事件就是当前会话自己发出的，所以可以忽略。

参数

channel

要对其发信号的通知频道的名称（任意标识符）。

payload

要通过通知进行沟通的“载荷”字符串。这必须是一个简单的字符串。在默认配置下，该字符串不能超过 8000 字节（如果需要发送二进制数据或者更多信息，最好是把它放在一个数据库表中并且发送该记录的键）。

注解

有一个队列保持着已经发送但是还没有被所有监听会话处理的通知。如果该队列被占满，调用NOTIFY的事务将在提交时失败。该队列非常大（标准安装中是 8GB）并且应该足以应付几乎每一种用例。不过，如果一个会话执行了NOTIFY并且接着长时间进入一个事务，不会发生清理操作。一旦该队列使用过半，你将在日志文件中看到警告，它指出哪个会话阻止了清理。在这种情况中，应该确保这个会话结束它的当前事务，这样清理才能够进行下去。

函数pg_notification_queue_usage返回队列中当前被 待处理通知所占据的比例。详见第 9.25 节

一个已经执行了NOTIFY的事务不能为两阶段提交做准备。

pg_notify

要发送一个通知，你也能使用函数pg_notify(text, text)。该函数采用频道名称作为第一个参数，而载荷则作为第二个参数。如果你需要使用非常量的频道名称和载荷，这个函数比NOTIFY命令更容易使用。

例子

从psql配置和执行一个监听/通知序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received
 from server process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server
 process with PID 14728.
```

兼容性

在 SQL 标准中没有NOTIFY语句。

参见

LISTEN, UNLISTEN

PREPARE

PREPARE — 为执行准备一个语句

大纲

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

描述

PREPARE创建一个预备语句。预备语句是一种服务器端对象，它可以被用来优化性能。当PREPARE语句被执行时，指定的语句会被解析、分析并且重写。当后续发出一个EXECUTE命令时，该预备语句会被规划并且执行。这种工作的划分避免了重复性的解析分析工作，不过允许执行计划依赖所提供的特定参数值。

预备语句可以接受参数：在执行时会被替换到语句中的值。在创建预备语句时，可以用位置引用参数，如$1、$2等。也可以选择性地指定参数数据类型的一个列表。当一个参数的数据类型没有被指定或者被声明为unknown时，其类型会从该参数第一次被引用的环境中推知（如果可能）。在执行该语句时，在EXECUTE语句中为这些参数指定实际值。更多有关于此的信息可参考EXECUTE。

预备语句只在当前数据库会话期间存在。当会话结束时，预备语句会消失，因此在重新使用之前必须重新建立它。这也意味着一个预备语句不能被多个数据库客户端同时使用。不过，每一个客户端可以创建它们自己的预备语句来使用。预备语句可以用DEALLOCATE命令手工清除。

当一个会话要执行大量类似语句时，预备语句可能会有最大性能优势。如果该语句很复杂（难于规划或重写），例如，如果查询涉及很多表的连接或者要求应用多个规则，性能差异将会特别明显。如果语句相对比较容易规划和重写，但是执行起来开销相对较大，那么预备语句的性能优势就不那么显著了。

参数

name

给这个特定预备语句的任意名称。它在一个会话中必须唯一并且后续将被用来执行或者清除一个之前准备好的语句。

data_type

预备语句一个参数的数据类型。如果一个特定参数的数据类型没有被指定或者被指定为unknown，将从该参数第一次被引用的环境中推得。要在预备语句本身中引用参数，可以使用 $1、$2等。

statement

任何SELECT、INSERT、UPDATE、DELETE或者VALUES语句。

注解

对每一组提供的EXECUTE值，预备语句可以使用通用计划而不是重新做计划。对于没有参数的预备语句马上就会这样做，否则只有五次或者更多次执行产生的计划的估计代价平均值（包括规划开销）比通用计划的代价估计更昂贵时才会这样做。一旦选中一个通用计划，在该预备语句剩余的生存时间内都将使用它。使用在重复值很多的列中很少出现的EXECUTE值

可以产生比通用计划更加廉价的定制计划（即使加上规划开销），这样通用计划将不会被使用。

通用计划假定每一个提供给EXECUTE的值都是该列的可区分值并且列值是均匀分布的。例如，如果统计信息记录了三个可区分的列值，通用计划会假定一个列等值比较将匹配被处理行中的 33%。列统计值也允许通用计划准确地计算唯一列的选择度。如果一个通用计划被选中时，在非均匀分布列上的比较以及制定不存在的值都会影响平均计划代价。

要检查PostgreSQL为一个预备语句使用的查询计划，可以使用EXPLAIN，例如EXPLAIN EXECUTE。如果使用的是一个通用计划，它将包含参数符号$n，而一个定制计划则会把提供的参数值替换进去。通用计划中的行估计值反映了这些参数计算出来的选择度。

更多关于查询规划以及PostgreSQL为此所收集的统计信息的内容，请见ANALYZE文档。

尽管预备语句主要是为了避免重复对语句进行解析分析以及规划，但是只要上一次使用该预备语句后该语句中用到的数据库对象发生了定义性（DDL）改变，PostgreSQL将会对该语句强制进行重新分析和重新规划。还有，如果search_path的值发生变化，也将使用新的search_path重新解析该语句（后一种行为是从PostgreSQL 9.3 开始的新行为）。这些规则让预备语句的使用在语义上几乎等效于反复提交相同的查询文本，但是能在性能上获利（如果没有对象定义被改变，特别是如果最优计划保持不变时）。该语义等价性不完美的一个例子是：如果语句用一个未限定的名称引用表，并且之后在search_path中更靠前的模式中创建了一个新的同名表，则不会发生自动的重解析，因为该语句使用的对象没有被改变。不过，如果某些其他更改造成了重解析，后续使用中都会引用新表。

可以通过查询pg_prepared_statements系统视图来看到会话中所有可用的预备语句。

例子

为一个INSERT语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个SELECT语句创建一个预备语句，然后执行它：

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

注意第二个参数的数据类型没有被指定，因此会从使用$2的环境中推知。

兼容性

SQL 标准包括一个PREPARE语句，但是它只用于嵌入式 SQL。这个版本的PREPARE语句也使用了一种有些不同的语法。

另见

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — 为两阶段提交准备当前事务

大纲

```
PREPARE TRANSACTION transaction_id
```

描述

PREPARE TRANSACTION为两阶段提交准备 当前事务。在这个命令之后，该事务不再与当前会话关联。相反，它的状态 被完全存储在磁盘上，并且有很高的可能性它会被提交成功（即便在请求提 交前发生数据库崩溃）。

一旦被准备好，事务稍后就可以分别用 COMMIT PREPARED 或者ROLLBACK PREPARED提交或者回滚。可以从任何 会话而不仅仅是执行原始事务的会话中发出这些命令。

从发出命令的会话的角度来看，PREPARE TRANSACTION不像ROLLBACK命令： 在执行它之后，就没有活跃的当前事务，并且该预备事务的效果也不再可见（ 如果该事务被提交，效果将重新变得可见）。

如果由于任何原因PREPARE TRANSACTION 命令失败，它会变成一个ROLLBACK：当前事务会被取消。

参数

transaction_id

一个任意的事务标识符， COMMIT PREPARED或者ROLLBACK PREPARED 以后将用这个标识符来标识这个事务。该标识符必须写成一个字符串， 并且长度必须小于 200 字节。它也不能与任何当前已经准备好的事务的标识符相同。

注解

PREPARE TRANSACTION并不是设计为在应用或者交互式 会话中使用。它的目的是允许一个外部事务管理器在多个数据库或者其他事务性 来源之间执行原子的全局事务。除非你在编写一个事务管理器，否则你可能不会 用到PREPARE TRANSACTION。

这个命令必须在一个事务块中使用。事务块用BEGIN开始。

当前在已经执行过任何涉及到临时表或者会话的临时命名空间、创建带 WITH HOLD的游标或者执行 LISTEN、UNLISTEN或NOTIFY的 事务中，不允许PREPARE该事务。这些特性与当前会话绑定得太过紧密，所以对一个要被准备的事务来说没有什么用处。

如果用SET（不带LOCAL选项）修改过事务的 任何运行时参数，这些效果会持续到 PREPARE TRANSACTION之后，并且将不会被后续的任何 COMMIT PREPARED或 ROLLBACK PREPARED所影响。因此，在这一 方面PREPARE TRANSACTION的行为更像 COMMIT而不是ROLLBACK。

所有当前可用的准备好事务被列在pg_prepared_xacts系统视图中。

> **小心**
>
> 让一个事务处于准备好状态太久是不明智的。这将会干扰 VACUUM回收存储的能力，并且在极限情况下可能导致 数据库关闭以阻止事务 ID 回卷

（见第 24.1.5 节。还要记住，该事务会继续持有 它已经持有的锁。该特性的设计用法是，只要一个外部事务管理器已经验证 其他数据库也准备好了要提交，一个准备好的事务将被正常地提交或者回滚。

如果没有建立一个外部事务管理器来跟踪准备好的事务并且确保它们被迅速地结束，最好禁用准备好事务特性（设置 max_prepared_transactions为零）。这将防止意外 地创建准备好事务，不然该事务有可能被忘记并且最终导致问题。

例子

为两阶段提交准备当前事务，使用foobar作为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

兼容性

PREPARE TRANSACTION是一种 PostgreSQL扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — 更改一个数据库角色拥有的数据库对象的拥有关系

大纲

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
               TO { new_role | CURRENT_USER | SESSION_USER }
```

描述

REASSIGN OWNED指示系统把 old_role们拥有 的任何数据库对象的拥有关系更改为new_role。

参数

old_role

一个角色的名称。这个角色在当前数据库中所拥有的 所有对象以及所有共享对象（数据库、表空间）的 所有权都将被重新赋予给 new_role。

new_role

将作为受影响对象的新拥有者的角色名称。

注解

REASSIGN OWNED经常被用来为移除一个 或者多个角色做准备。因为REASSIGN OWNED不影响其他数据库中的对象，通常需要在包含有 被删除的角色所拥有的对象的每一个数据库中都执行这个命令。

REASSIGN OWNED同时要求源角色和目标 角色上的成员资格。

DROP OWNED命令可以简单地删掉一个或者多个角色所拥有 的所有数据库对象。

REASSIGN OWNED命令不会影响授予给 old_role们 的在它们不拥有的对象上的任何特权。DROP OWNED 可以回收那些特权。

更多讨论请见第 21.4 节

兼容性

REASSIGN OWNED命令是一种 PostgreSQL扩展。

另见

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — 替换一个物化视图的内容

大纲

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

描述

REFRESH MATERIALIZED VIEW完全替换一个 物化视图的内容。你必须是该物化视图的属主才能执行这个命令.旧的内容会被抛弃。如果指定了 WITH DATA（或者作为默认值），支持查询将被执行以 提供新的数据，并且会让物化视图将处于可扫描的状态。如果指定了 WITH NO DATA，则不会生成新数据并且会让物化视图 处于一种不可扫描的状态。

CONCURRENTLY和WITH NO DATA 不能被一起指定。

参数

CONCURRENTLY

对物化视图的刷新不阻塞在该物化视图上的并发选择。如果没有这个选项， 一次影响很多行的刷新将使用更少的资源并且更快结束，但是可能会阻塞 其他尝试从物化视图中读取的连接。这个选项在只有少量行被影响的情况 下可能会更快。

只有当物化视图上有至少一个UNIQUE索引（只用列名 并且包括所有行）时，才允许这个选项。也就是说，该索引不能建立在任何 表达式上或者包括WHERE子句。

当物化视图还未被填充时，这个选项不能被使用。

即使带有这个选项，对于任意一个物化视图一次也只能运行一个 REFRESH。

name

要刷新的物化视图的名称（可以被模式限定）。

注解

虽然用于未来的CLUSTER操作的默认索引会被保持， REFRESH MATERIALIZED VIEW不会基于这个属性排序产生 的行。如果希望数据在产生时排序，必须在支持查询中使用 ORDER BY子句。

示例

这个命令将使用物化视图order_summary定义中的查询 来替换该物化视图的内容，并且让它处于一种可扫描的状态：

```
REFRESH MATERIALIZED VIEW order_summary;
```

这个命令将释放与物化视图annual_statistics_basis相关 的存储并且让它变成一种不可扫描的状态：

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

兼容性

REFRESH MATERIALIZED VIEW是一种 PostgreSQL扩展。

另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — 重建索引

大纲

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name
```

描述

REINDEX使用索引的表里存储的数据重建一个索引， 并且替换该索引的旧拷贝。有一些场景需要使用REINDEX：

- 一个索引已经损坏，并且不再包含合法数据。尽管理论上这不会发生， 实际上索引会因为软件缺陷或硬件失效损坏。 REINDEX提供了一种恢复方法。
- 一个索引变得“臃肿”，其中包含很多空的或者近乎为空的页面。 PostgreSQL中的 B-树索引在特定的非 常规访问模式下可能会发生这种情况。REINDEX 提供了一种方法来减少索引的空间消耗，即制造一个新版本的索引，其中没有 死亡页面。详见第 24.2 节
- 修改了一个索引的存储参数（例如填充因子），并且希望确保这种修改完全 生效。
- 用CONCURRENTLY选项进行的一次索引创建失败，留下了 一个“无效的”索引。这类索引是没有用处的，但是可以用 REINDEX来重建它们。注意， REINDEX将不会执行一次并发构建。要构建该索引并且 不干扰生产，应该先删除索引并且重新发出 CREATE INDEX CONCURRENTLY命令。

参数

INDEX

重新创建指定的索引。

TABLE

重新创建指定表的所有索引。如果该表有一个二级 “TOAST”表，它也会被重索引。

SCHEMA

重建指定方案的所有索引。如果这个方案中的一个表有次级的“TOAST”表，它也会被重建索引。共享系统目录上的索引也会被处理。这种形式的REINDEX不能在事务块内执行。

DATABASE

重新创建当前数据库内的所有索引。共享的系统目录上的索引也会被 处理。这种形式的REINDEX不能在一个 事务块内执行。

SYSTEM

重新创建当前数据库中在系统目录上的所有索引。共享系统目录上的 索引也被包括在内。用户表上的索引则不会被处理。这种形式的 REINDEX不能在一个事务块内执行。

name

要被重索引的特定索引、表或者数据库的名字。索引和表名可以被 模式限定。当前，REINDEX DATABASE和 REINDEX SYSTEM只能重索引当前数据库，因此 它们的参数必须匹配当前数据库的名称。

VERBOSE

在每个索引被重建时打印进度报告。

注解

如果怀疑一个用户表上的索引损坏，可以使用 REINDEX INDEX或者 REINDEX TABLE简单地重建该索引 或者表上的所有索引。

如果你需要从一个系统表上的索引损坏中恢复，就更困难一些。在 这种情况下，对系统来说重要的是没有使用过任何可疑的索引本身（ 实际上，这种场景中，你可能会发现服务器进程会在启动时立刻崩溃， 这是因为对于损坏的索引的依赖）。要安全地恢复，服务器必须用 -P选项启动，这将阻止它使用索引来进行系统 目录查找。

这样做的一种方法是关闭服务器，并且启动一个单用户的 PostgreSQL服务器，在其命令行中包括-P选项。然后，可以发出 REINDEX DATABASE、REINDEX SYSTEM、 REINDEX TABLE或者REINDEX INDEX， 具体使用哪个命令取决于你想要重构多少东西。如果有疑问，可以使用REINDEX SYSTEM来选择重建数据库中的所有系统索引。 然后退出单用户服务器会话并且重启常规的服务器。更多关于如何与 单用户服务器接口交互的内容请见postgres参考页。

在另一种方法中，可以开始一个常规的服务器会话，在其命令行选项 中包括-P。这样做的方法与客户端有关，但是在 所有基于libpq的客户端中都可以在开始客户端 之前设置PGOPTIONS环境变量为-P。 注意虽然这种方法不要求用锁排斥其他客户端，在修复完成之前避免 其他用户连接到受损的数据库才是更加明智的。

REINDEX类似于删除索引并且重建索引，在其中 索引内容会被从头开始建立。不过，锁定方面的考虑却相当不同。 REINDEX会用锁排斥写，但不会排斥在索引的父表上的读。 它也会在被处理的索引上取得一个排他锁，该锁将会阻塞对该索引的使用尝试。 相反，DROP INDEX会暂时在附表上取得一个排他锁，阻塞 写和读。后续的CREATE INDEX会排斥写但不排斥读，由于 该索引不存在，所以不会有读取它的尝试，这意味着不会有阻塞但是读操作可能 被强制成昂贵的顺序扫描。

重索引单独一个索引或者表要求用户是该索引或表的拥有者。对方案或数据库重建索引要求是该方案或者数据库的拥有者。注意因此非超级用户有时无法重建其他用户拥有的表上的索引。不过，作为一种特例，当一个非超级用户发出REINDEX DATABASE、REINDEX SCHEMA或者REINDEX SYSTEM时，共享目录上的索引将被跳过，除非该用户拥有该目录（通常不会是这样）。当然，超级用户总是可以重建所有的索引。

不支持重建分区表的索引或者分区索引。不过可以单独为每个分区重建索引。

示例

重建单个索引：

```
REINDEX INDEX my_index;
```

重建表my_table上的所有索引：

```
REINDEX TABLE my_table;
```

重建一个特定数据库中的所有索引，且不假设系统索引已经可用：

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
```

```
broken_db=> \q
```

兼容性

在 SQL 标准中没有REINDEX命令。

RELEASE SAVEPOINT

RELEASE SAVEPOINT — 销毁一个之前定义的保存点

大纲

```
RELEASE [ SAVEPOINT ] savepoint_name
```

描述

RELEASE SAVEPOINT销毁在当前事务 中之前定义的一个保存点。

销毁一个保存点会使得它不能再作为一个回滚点，但是它没有其他用户 可见的行为。它不会撤销在该保存点被建立之后执行的命令的效果（要 这样做，可见ROLLBACK TO SAVEPOINT）。当不再需要一个 保存点时销毁它允许系统在事务结束之前回收一些资源。

RELEASE SAVEPOINT也会销毁所有 在该保存点建立之后建立的保存点。

参数

savepoint_name

要销毁的保存点的名称。

注解

指定一个不是之前定义的保存点名称是错误。

当事务处于中止状态时不能释放保存点。

如果多个保存点具有相同的名称，只有最近被定义的那个会被释放。

示例

建立并且销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上述事务将插入 3 和 4。

兼容性

这个命令符合SQL标准。该标准指定关键词 SAVEPOINT是强制需要的，但 PostgreSQL允许省略。

另见

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — 把一个运行时参数的值恢复到默认值

大纲

```
RESET configuration_parameter
RESET ALL
```

描述

RESET把运行时参数恢复到它们的默认值。 RESET是

```
SET configuration_parameter TO DEFAULT
```

的另一种写法。 详见SET。

默认值被定义为如果在当前会话中没有发出过SET， 参数必须具有的值。这个值的实际来源可能是一个编译在内部的默认值、 配置文件、命令行选项、或者针对每个数据库或者每个用户的默认设置。 这和把它定义成“在会话开始时该参数得到的值”有细微的差 别，因为如果该值来自于配置文件，它将被重置为现在配置文件所指定 的任何东西。详见第 19 章

RESET的事务行为和SET相同： 它的效果会被事务回滚撤销。

参数

configuration_parameter

一个可设置的运行时参数名称。可用的参数记录在 第 19 章以及 SET参考页中。

ALL

把所有可设置的运行时参数重置为默认值。

示例

把timezone配置变量设置为默认值：

```
RESET timezone;
```

兼容性

RESET是一种 PostgreSQL扩展。

另见

SET, SHOW

REVOKE

REVOKE — 移除访问特权

大纲

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
```

```
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION function_name [ ( [ [ argmode ] [ arg_name ] arg_type
 [, ...] ] ) ] [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_name [, ...]
    [ CASCADE | RESTRICT ]
```

描述

REVOKE命令收回之前从一个或者更多角色　授予的特权。关键词PUBLIC隐式定义的全部角色的组。

特权类型的含义见GRANT命令的描述。

注意任何特定角色拥有的特权包括直接授予给它的特权、从它作为其成员的　角色中得到的特权以及授予给PUBLIC的特权。因此，　从PUBLIC收回SELECT特权并不一定会意味　着所有角色都会失去在该对象上的SELECT特权：那些直接被授　予的或者通过另一个角色被授予的角色仍然会拥有它。类似地，从一个用户　收回SELECT后，如果PUBLIC或者另一个　成员关系角色仍有SELECT权利，该用户还是可以使用 SELECT。

如果指定了GRANT OPTION FOR，只会回收该特权 的授予选项，特权本身不被回收。否则，特权及其授予选项都会被回收。

如果一个用户持有一个带有授予选项的特权并且把它授予给了其他用户，　那么被那些其他用户持有的该特权被称为依赖特权。如果第一个用户持有　的该特权或者授予选项正在被收

回且存在依赖特权，指定 CASCADE可以连带回收那些依赖特权，不指定则会 导致回收动作失败。这种递归回收只影响通过可追溯到该 REVOKE命令的主体的用户链授予的特权。因此，如果该特权经由其他用户授予给受影响用户，受影响用户可能实际上还 保留有该特权。

在回收一个表上的特权时，也会在该表的每一个列上自动回收对应的列 特权（如果有）。在另一方面，如果一个角色已经被授予一个表上的 特权，那么从个别的列上回收同一个特权将不会生效。

在回收一个角色中的成员关系时，GRANT OPTION被改 称为ADMIN OPTION，但行为是类似的。也要注意这种 形式的命令不允许噪声词GROUP。

注解

使用psql的\dp 命令能够显示在现有表和列上已被授予的特权。其格式可见GRANT。对于不是表的对象有其他\d 命令可以显示它们的特权。

用户只能回收由它直接授出的特权。例如，如果用户 A 已经把一个带有 授予选项的特权授予给了用户 B，并且用户 B 接着把它授予给了用户 C， 那么用户 A 无法直接从 C 收回该特权。反而，用户 A 可以从用户 B 收回 该授予选项并且使用CASCADE选项，这样该特权会被 依次从用户 C 回收。对于另一个例子，如果 A 和 B 都把同一个特权授予 给了 C，A 能够收回它们自己的授权但不能收回 B 的授权，因此 C 实际上 仍将拥有该特权。

当一个对象的非拥有者尝试REVOKE该对象上的特权时， 如果该用户在该对象上什么特权都不拥有，该命令会立刻失败。只要有某个 特权可用，该命令将继续，但是它只会收回那些它具有授予选项的特权。 如果没有持有授予选项，REVOKE ALL PRIVILEGES形式 将发出一个警告，而其他形式在没有持有该命令中特别提到的任何特权的 授予选项时就会发出警告（原则上，这些语句也适用于对象拥有者，但是 由于拥有者总是被认为持有所有授予选项，这些情况永远不会发生）。

如果一个超级用户选择发出一个GRANT或者 REVOKE命令，该命令就好像被受影响对象的拥有者发出 的一样被执行。因为所有特权最终来自于对象拥有者（可能是间接地通过 授予选项链），可以由超级用户收回所有特权，但是这可能需要前述的 CASCADE。

REVOKE也可以由一个并非受影响对象的拥有者的角色 完成，但是该角色应该是一个拥有该对象的角色的成员或者是一个在该对象 上拥有特权的WITH GRANT OPTION的角色的成员。 在这种情况中，该命令就好像被实际拥有该对象或者特权的 WITH GRANT OPTION的包含角色发出的一样被执行。 例如，如果表t1被角色g1拥有，而u1 是g1的一个成员，那么u1能收回t1上被记录为由g1授出的特权。这会包括由u1 以及由角色g1的其他成员完成的授予。

如果执行REVOKE的角色持有通过多于一条角色成员 关系路径间接得到的特权，其中哪一条包含将被用于执行该命令的 角色是没有被指明的。在这种情况中，最好使用 SET ROLE成为你想作为其身份执行 REVOKE的特定角色。如果无法做到这一点 可能会导致回收超过你预期的特权，或者根本回收不了任何东西。

示例

从 public 收回表films上的插入特权：

```
REVOKE INSERT ON films FROM PUBLIC;
```

从用户manuel收回视图 kinds上的所有特权：

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

注意着实际意味着“收回所有我授出的特权”。

从用户joe收回角色admins中的成员关系：

```
REVOKE admins FROM joe;
```

兼容性

GRANT命令的兼容性注解同样适用于 REVOKE。根据标准，关键词 RESTRICT或CASCADE 是必要的，但是PostgreSQL默认假定为 RESTRICT。

另见

GRANT

ROLLBACK

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ]
```

描述

ROLLBACK回滚当前事务并且导致 该事务所作的所有更新都被抛弃。

参数

WORK
TRANSACTION

可选关键词，没有效果。

注解

使用COMMIT可成功地终止一个事务。

在一个事务块之外发出ROLLBACK会发出一个 警告并且不会有效果。

示例

要中止所有更改：

```
ROLLBACK;
```

兼容性

SQL 标准只指定了 ROLLBACK和ROLLBACK WORK两种形式。除此之外，这个命令完全符合 标准。

另见

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — 取消一个之前为两阶段提交准备好的事务

大纲

```
ROLLBACK PREPARED transaction_id
```

描述

ROLLBACK PREPARED回滚 一个处于准备好状态的事务。

参数

transaction_id

要被回滚的事务的事务标识符。

注解

要回滚一个准备好的事务，你必须是原先执行该事务的同一个用户或者 是一个超级用户。但是你必须处在执行该事务的同一个会话中。

这个命令不能在一个事务块内被执行。准备好的事务会被立刻回滚。

pg_prepared_xacts 系统视图中列出了当前可用的所有准备好的事务。

例子

用事务标识符foobar回滚对应的事务：

```
ROLLBACK PREPARED 'foobar';
```

兼容性

ROLLBACK PREPARED是一种 PostgreSQL扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — 回滚到一个保存点

大纲

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

描述

回滚在该保存点被建立之后执行的所有命令。该保存点保持有效并且可以在 以后再次回滚到它（如果需要）。

ROLLBACK TO SAVEPOINT隐式地销毁在所提及的保存点 之后建立的所有保存点。

参数

savepoint_name

要回滚到的保存点。

注解

使用RELEASE SAVEPOINT销毁一个保存点而 不抛弃在它建立之后被执行的命令的效果。

指定一个没有被建立的保存点是一种错误。

相对于保存点，游标有一点非事务的行为。在保存点被回滚时，任何在该保存点 内被打开的游标将会被关闭。如果一个先前打开的游标在一个保存点内被 FETCH或MOVE命令所影响，而该该保存点 后来又被回滚，那么该游标将保持FETCH使它指向的位置（也 就是说由FETCH导致的游标动作不会被回滚）。回滚也不能 撤销关闭一个游标。不过，其他由游标查询导致的副作用（例如 被该查询所调用的易变函数的副作用） 可以被回滚，只要它们发生在一个后来被回滚的保存点期间。 如果一个游标的执行导致事务中止，它会被置于一种不能被执行的状态，这样当 事务被用ROLLBACK TO SAVEPOINT恢复后，该游标也不再能 被使用。

示例

要撤销在my_savepoint建立后执行的命令的效果：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不会受保存点回滚的影响：

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
```

```
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

兼容性

SQL标准指定关键词 SAVEPOINT是强制的，但是PostgreSQL 和Oracle允许省略它。SQL 只允许WORK而 不是TRANSACTION作为ROLLBACK之后的噪声词。 还有，SQL 有一个可选的子句 AND [NO] CHAIN，当前 PostgreSQL并不支持。在其他方面，这个命令符合 SQL 标准。

另见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — 在当前事务中定义一个新的保存点

大纲

```
SAVEPOINT savepoint_name
```

描述

SAVEPOINT在当前事务中建立一个新保存点。

保存点是事务内的一种特殊标记，它允许所有在它被建立之后执行的命令被　回滚，把该事务的状态恢复到它处于保存点时的样子。

参数

savepoint_name

　给新保存点的名字。

注解

使用ROLLBACK TO SAVEPOINT回滚到一个保存点。 使用RELEASE SAVEPOINT销毁一个保存点，但保持在它被建立之后执行的命令的效果。

保存点只能在一个事务块内建立。可以在一个事务内定义多个保存点。

示例

要建立一个保存点并且后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值 1 和 3，但不会插入 2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4。

兼容性

当建立另一个同名保存点时，SQL 要求之前的那个保存点自动被销毁。在 PostgreSQL中，旧的保存点会被保留，不过在进行 回滚或释放时只能使用最近的那一个（用 RELEASE SAVEPOINT释放较新的保存点将会 导致较旧的保存点再次变得可以被 ROLLBACK TO SAVEPOINT和 RELEASE SAVEPOINT访问）。在其他方面， SAVEPOINT完全符合 SQL。

另见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — 定义或更改应用到一个对象的安全标签

大纲

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SUBSCRIPTION object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'

其中 aggregate_signature 是:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

描述

SECURITY LABEL对一个数据库对象应用一个安全 标签。可以把任意数量的安全标签（每个标签提供者对应一个）关联到一个给定 的数据库对象。标签提供者是使用函数register_label_provider 注册自己的可装载模块。

> **注意**
>
> register_label_provider不是一个 SQL 函数，它只能在被载入 到后端的 C 代码中调用。

标签提供者决定一个给定标签是否合法并且它是否可以被分配该标签给一个给定 对象。一个给定标签的含义也同样由标签提供者判断。 PostgreSQL没有对一个标签提供者是否必须或者如何解释 安全标签做出限定，它仅仅只是提供了一种机制来存储它们。实际上，这个

功能是 为了允许与基于标签的强制访问控制（MAC）系统集成（例如 SE-Linux）。这类系统会基于对象标签而不是传统的自主 访问控制（DAC）概念（例如用户和组）做出所有访问控制决定。

参数

object_name
table_name.column_name
aggregate_name
function_name
procedure_name
routine_name

要被贴上标签的对象的名称。表、聚集、域、外部表、函数、存储过程、例程、序列、类型和视图 的名称可以是模式限定的。

provider

这个标签相关联的提供者的名称。所提到的提供者必须已被载入并且必须赞同所提出 的标签操作。如果正好只载入了一个提供者，可以出于简洁的需要忽略提供者的名称。

argmode

一个函数，存储过程或者聚集函数参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认值会是 IN。注意SECURITY LABEL并不真正 关心OUT参数，因为判断函数的身份时只需要输入参数。因此列出 IN、INOUT和VARIADIC参数足矣。

argname

一个函数，存储过程或者聚集函数参数的名称。注意SECURITY LABEL 并不真正关心参数的名称，因为判断函数的身份时只需要参数的数据类型。

argtype

一个函数，存储过程或聚集函数参数的数据类型。

large_object_oid

大对象的 OID。

PROCEDURAL

这是一个噪声词。

label

写成一个字符串文本的新安全标签。如果写成NULL表示删除 原有的安全标签。

示例

下面的例子展示了如何更改一个表的安全标签。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS
 'system_u:object_r:sepgsql_table_t:s0';
```

兼容性

在 SQL 标准中没有SECURITY LABEL命令。

另见

sepgsql, src/test/modules/dummy_seclabel

SELECT

SELECT, TABLE, WITH — 从一个表或视图检索行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
 LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name
 [, ...] ] [ NOWAIT ] [...] ]
```

其中 from_item 可以是以下之一：

```
    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE
 ( seed ) ] ]

    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
 [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias
 ( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition
 [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS
 ( column_definition [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
 [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING
 ( join_column [, ...] ) ]
```

并且 grouping_element 可以是以下之一：

```
    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )
```

并且 with_query 是：

```
    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert
 | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

描述

SELECT从零或更多表中检索行。 SELECT的通常处理如下：

1. WITH列表中的所有查询都会被计算。这些查询实际 充当了在FROM列表中可以引用的临时表。在 FROM中被引用多次的WITH查 询只会被计算一次（见下文的 WITH 子句）。
2. FROM列表中的所有元素都会被计算（ FROM中的每一个元素都是一个真实表或者虚拟表）。 如果在FROM列表中指定了多于一个元素，它们会被 交叉连接在一起（见下文的 FROM 子句）。
3. 如果指定了WHERE子句，所有不满足该条件的行都会 被从输出中消除（见下文的WHERE 子句）。
4. 如果指定了GROUP BY子句或者如果有聚集函数，输出 会被组合成由在一个或者多个值上匹配的行构成的分组，并且在其上计算聚 集函数的结果。如果出现了HAVING子句，它会消除不 满足给定条件的分组（见下文的 GROUP BY 子句以及 HAVING 子句）。
5. 对于每一个被选中的行或者行组，会使用SELECT 输出表达式计算实际的输出行（见下文的 SELECT 列表）。
6. SELECT DISTINCT从结果中消除重复的行。 SELECT DISTINCT ON消除在所有指定表达式上匹 配的行。SELECT ALL（默认）将返回所有候选行， 包括重复的行（见下文的DISTINCT子句）。
7. 通过使用操作符UNION、 INTERSECT和EXCEPT，多于 一个SELECT语句的输出可以被整合形成 一个结果集。UNION操作符返回位于一个或者两 个结果集中的全部行。INTERSECT操作符返回同时 位于两个结果集中的所有行。EXCEPT操作符返回 位于第一个结果集但不在第二个结果集中的行。在所有三种情况下， 重复行都会被消除（除非指定ALL）。可以增加噪 声词DISTINCT来显式地消除重复行。注意虽然 ALL是SELECT自身的默认行为， 但这里DISTINCT是默认行为（见下文的 UNION 子句、INTERSECT 子句以及 EXCEPT 子句）。
8. 如果指定了ORDER BY子句，被返回的行会以指定的 顺序排序。如果没有给定ORDER BY，系统会以能最 快产生行的顺序返回它们（见下文的 ORDER BY 子句）。
9. 如果指定了LIMIT（或FETCH FIRST） 或者OFFSET子句，SELECT 语句只返回结果行的一个子集（见下文的LIMIT 子句）。
10. 如果指定了FOR UPDATE、 FOR NO KEY UPDATE、 FOR SHARE 或者FOR KEY SHARE， SELECT语句会把被选中的行锁定而不让并发 更新访问它们（见下文的锁定子句）。

你必须拥有在一个SELECT命令中使用的每一列上的 SELECT特权。FOR NO KEY UPDATE、 FOR UPDATE、 FOR SHARE或者FOR KEY SHARE 还要求（对这样选中的每一个表至少一列的）UPDATE特权。

参数

WITH 子句

WITH子句允许你指定一个或者多个在主查询中可以 其名称引用的子查询。在主查询期间子查询实际扮演了临时表或者视图 的角色。每一个子查询都可以是一个SELECT、TABLE、VALUES、 INSERT、 UPDATE或者 DELETE语句。在WITH中书写 一个数据修改语句（INSERT、 UPDATE或者 DELETE）时，通常要包括一个 RETURNING子句。构成被主查询读取

的临时表的是 RETURNING的输出，而不是该语句修改的 底层表。如果省略RETURNING，该语句仍会被执行，但是它 不会产生输出，因此它不能作为一个表从主查询引用。

对于每一个WITH查询，都必须指定一个名称（无需模 式限定）。可选地，可以指定一个列名列表。如果省略该列表，会从该子查 询中推导列名。

如果指定了RECURSIVE，则允许一个 SELECT子查询使用名称引用自身。 这样一个子查询的形式必须是

```
non_recursive_term UNION [ ALL | DISTINCT ] recursive_term
```

其中递归自引用必须出现在UNION的右手边。每个 查询中只允许一个递归自引用。不支持递归数据修改语句，但是 可以在一个数据查询语句中使用一个递归 SELECT查询的结果。例子可见 第 7.8 节

RECURSIVE的另一个效果是 WITH查询不需要被排序：一个查询可以引用另一个 在列表中比它靠后的查询（不过，循环引用或者互递归没有实现）。 如果没有RECURSIVE，WITH 查询只能引用在WITH列表中位置更前面的兄弟 WITH查询。

WITH查询的一个关键特性是，对主查询的每次 查询它们都只计算一次，即使该主查询引用它们多次也是如此。特别 是，不管主查询读取数据修改语句的多少输出，数据修改语句都被保 证仅执行一次。

主查询以及WITH查询全部（理论上）在同一时间 被执行。这意味着从该查询的任何部分都无法看到 WITH中的一个数据修改语句的效果，不过可以读 取其RETURNING输出。如果两个这样的数据修改语句 尝试修改相同的行，结果将无法确定。

更多信息请见第 7.8 节

FROM 子句

FROM子句为SELECT 指定一个或者更多源表。如果指定了多个源表，结果将是所有源表的 笛卡尔积（交叉连接）。但是通常会增加限定条件（通过 WHERE）来把返回的行限制为该笛卡尔积的一个小子集。

FROM子句可以包含下列元素：

table_name

一个现有表或视图的名称（可以是模式限定的）。如果在表名前指定了 ONLY，则只会扫描该表。如果没有指定 ONLY，该表及其所有后代表（如果有）都会被扫描。可 选地，可以在表名后指定*来显式地指示包括后代表。

alias

一个包含别名的FROM项的替代名称。别名被用于 让书写简洁或者消除自连接中的混淆（其中同一个表会被扫描多 次）。当提供一个别名时，表或者函数的实际名称会被隐藏。例 如，给定FROM foo AS f， SELECT的剩余部分就必须以 f而不是foo来引用这个FROM项。如果写了一个别名，还可以写一个列别 名列表来为该表的一个或者多个列提供替代名称。

TABLESAMPLE sampling_method (argument [, ...]) [REPEATABLE (seed)]

table_name之后的 TABLESAMPLE子句表示应该用指定的 sampling_method 来检索表中行的子集。这种采样优先于任何其他过滤器（例如 WHERE子句）。标准 PostgreSQL发布包括两种采样 方法：BERNOULLI和SYSTEM， 其他采样方法可以通过扩展安装在数据库中。

BERNOULLI以及SYSTEM采样方法都接受 一个参数，它表示要采样的表 的分数，表示为一个 0 到 100 之间的百分数。这个参数可以是任意的 实数值表达式（其他的采样方法可能接受更多或者不同的 参数）。这两种方法都返回一个随机选取的该表采样，其中

包含了指定 百分数的表行。BERNOULLI方法扫描整个表并且 用指定的几率选择或者忽略行。SYSTEM方法会做 块层的采样，每个块都有指定的机会能被选中，被选中块中的所有行都 会被返回。在指定较小的采样百分数时，SYSTEM 方法要比BERNOULLI方法快很多，但是前者可能 由于聚簇效应返回随机性较差的表采样。

可选的REPEATABLE子句指定一个用于产生采样方法中随机数的种子数或表达式。种子值可以是任何非空浮点值。如果查询时表没有被更改，指定相同种子和argument值的两个查询将会选择该表相同的采样。但是不同的种子值通常将会产生不同的采样。如果没有给出REPEATABLE，则会基于一个系统产生的种子为每一个查询选择一个新的随机采样。注意有些扩展采样方法不接受REPEATABLE，并且将总是为每一次使用产生新的采样。

select

一个子-SELECT可以出现在 FROM子句中。这就好像把它的输出创建为一个 存在于该SELECT命令期间的临时表。注意 子-SELECT必须用圆括号包围，并且 必须为它提供一个别名。也可以在这里 使用一个VALUES命令。

with_query_name

可以通过写一个WITH查询的名称来引用它，就好像 该查询的名称是一个表名（实际上，该WITH查询会 为主查询隐藏任何具有相同名称的真实表。如果必要，你可以使用 带模式限定的方式以相同的名称来引用真实表）。可以像表一样， 以同样的方式提供一个别名。

function_name

函数调用可以出现在FROM子句中（对于返回结果 集合的函数特别有用，但是可以使用任何函数）。这就好像把该函数的 输出创建为一个存在于该SELECT命令期 间的临时表。当为该函数调用增加可选的 WITH ORDINALITY子句时，会在该函数 的输出列之后追加一个新的列来为每一行编号。

可以用和表一样的方式提供一个别名。如果写了一个别名，还可以写一个 列别名列表来为该函数的组合返回类型的一个或者多个属性提供替代名称， 包括由ORDINALITY（如果有）增加的新列。

通过把多个函数调用包围在ROWS FROM(...)中可以把它们 整合在单个FROM-子句项中。这样一个项的输出是把每一个 函数的第一行串接起来，然后是每个函数的第二行，以此类推。如果有些 函数产生的行比其他函数少，则在缺失数据的地方放上空值，这样被返回 的总行数总是和产生最多行的函数一样。

如果函数被定义为返回record数据类型，那么必须出现一个 别名或者关键词AS，后面跟上形为 (column_name data_type [, ...])的列定义列表。列定义列表必须匹配该函数返回的列的实际 数量和类型。

在使用ROWS FROM(...)语法时，如果函数之一要求一个列 定义列表，最好把该列定义列表放在ROWS FROM(...)中该 函数的调用之后。当且仅当正好只有一个函数并且没有WITH ORDINALITY子句时，才能把列定义列表放在 ROWS FROM(...)结构后面。

要把ORDINALITY和列定义列表一起使用，你必须使用 ROWS FROM(...)语法，并且把列定义列表放在 ROWS FROM(...)里面。

join_type

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

之一。 对于INNER和OUTER连接类型，必须指定 一个连接条件，即 NATURAL、ON join_condition或者 USING (join_column [, ...]) 之一（只能有一种）。其含义见下文。对于 CROSS JOIN，上述子句不能出现。

一个JOIN子句联合两个FROM项（ 为了方便我们称之为“表”，尽管实际上它们可以是任何类型 的FROM项）。如有必要可以使用圆括号确定嵌套的顺序。 在没有圆括号时，JOIN会从左至右嵌套。在任何情 况下，JOIN的联合比分隔FROM-列表 项的逗号更强。

CROSS JOIN和INNER JOIN 会产生简单的笛卡尔积，也就是与在FROM的顶层列出两个 表得到的结果相同，但是要用连接条件（如果有）约束该结果。 CROSS JOIN与INNER JOIN ON (TRUE)等效，也就是说条件不会移除任何行。这些连接类型只是一种 记号上的方便，因为没有什么是你用纯粹的FROM和 WHERE能做而它们不能做的。

LEFT OUTER JOIN返回被限制过的笛卡尔积 中的所有行（即所有通过了其连接条件的组合行），外加左手表中 没有相应的通过了连接条件的右手行的每一行的拷贝。通过在右手列中插入空值，这种左手行会被扩展为连接表的完整行。注意在决 定哪些行匹配时，只考虑JOIN子句自身的条件。之后 才应用外条件。

相反，RIGHT OUTER JOIN返回所有连接行，外加每 一个没有匹配上的右手行（在左端用空值扩展）。这只是为了记号 上的方便，因为你可以通过交换左右表把它转换成一个LEFT OUTER JOIN。

FULL OUTER JOIN返回所有连接行，外加每 一个没有匹配上的左手行（在右端用空值扩展），再外加每一个没有 匹配上的右手行（在左端用空值扩展）。

ON join_condition

join_condition 是一个会得到boolean类型值的表达式（类似于一个 WHERE子句），它说明一次连接中哪些行被认为 相匹配。

USING (join_column [, ...])

形式USING (a, b, ...)的子句是 ON left_table.a = right_table.a AND left_table.b = right_table.b ...的简写。还有， USING表示每一对相等列中只有一个会被 包括在连接输出中。

NATURAL

NATURAL是一个USING列表的速记，该列表中提到两个表中具有匹配名称的所有的列。如果没有公共列名，则NATURAL等效于ON TRUE。

LATERAL

LATERAL关键词可以放在一个 子-SELECT FROM项前面。这允许该 子-SELECT引用FROM列表中在它之 前的FROM项的列（如果没有LATERAL，每一 个子-SELECT会被独立计算并且因此不能交叉引用 任何其他的FROM项）。

LATERAL也可以放在一个函数调用 FROM项前面，但是在这种情况下它只是一个噪声 词，因为在任何情况下函数表达式都可以引用在它之前的 FROM项。

LATERAL项可以出现在FROM列表 顶层，或者一个JOIN中。在后一种情况中，它也可以引用其作为右手端的JOIN左手端上的任何项。

当一个FROM项包含LATERAL交叉引用 时，计算会如此进行：对提供被交叉引用列的FROM项的每 一行或者提供那些列的多个FROM项的每一个行集，使用该 行或者行集的那些列值计算LATERAL项。结果行会与 计算得到它们的行进行通常的连接。对来自哪些列的源表的每一行或者行 集都会重复这样的步骤。

列的源表必须以INNER或者LEFT的方式连接到 LATERAL项，否则就没有用于为 LATERAL项计算每一个行集的良定行集。尽管 X RIGHT JOIN LATERAL Y这样的结构在语法上是合法的， 但实际上不允许用于在Y中引用 X。

WHERE 子句

可选的WHERE子句的形式

```
WHERE condition
```

其中condition 是任一计算得到布尔类型结果的表达式。任何不满足 这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的 变量引用后，该表达式返回真，则该行符合条件。

GROUP BY 子句

可选的GROUP BY子句的形式

```
GROUP BY grouping_element [, ...]
```

GROUP BY将会把所有被选择的行中共享相同分组表达式 值的那些行压缩成一个行。一个被用在 grouping_element中的 expression可以是输入列名、输出列 （SELECT列表项）的名称或序号或者由输入列 值构成的任意表达式。在出现歧义时，GROUP BY名称 将被解释为输入列名而不是输出列名。

如果任何GROUPING SETS、ROLLUP或者 CUBE作为分组元素存在，则GROUP BY子句 整体上定义了数个独立的分组集。其效果等效于在子 查询间构建一个UNION ALL，子查询带有分组集作为它们 的GROUP BY子句。处理分组集的进一步细节请见 第 7.2.4 节

聚集函数（如果使用）会在组成每一个分组的所有行上进行计算，从而为每 一个分组产生一个单独的值（如果有聚集函数但是没有 GROUP BY子句，则查询会被当成是由所有选中行构成 的一个单一分组）。传递给每一个聚集函数的行集合可以通过在聚集函数调 用附加一个FILTER子句来进一步过滤，详见 第 4.2.7 节当存在一个 FILTER子句时，只有那些匹配它的行才会被包括在该聚 集函数的输入中。

当存在GROUP BY子句或者任何聚集函数时， SELECT列表表达式不能引用非分组列（除非它出现在聚集函数中或者它函数依赖于分组列），因为这样做会导致返回 非分组列的值时会有多种可能的值。如果分组列是包含非分组列的表的主键（ 或者主键的子集），则存在函数依赖。

记住所有的聚集函数都是在HAVING子句或者 SELECT列表中的任何“标量”表达式之前被计算。 这意味着一个CASE表达式不能被用来跳过一个聚集表达式的 计算，见第 4.2.14 节

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和 GROUP BY一起指定。

HAVING 子句

可选的HAVING子句的形式

```
HAVING condition
```

其中condition与 WHERE子句中指定的条件相同。

HAVING消除不满足该条件的分组行。 HAVING与WHERE不同： WHERE会在应用GROUP BY之前过滤个体行，而HAVING过滤由 GROUP BY创建的分组行。 condition中引用 的每一个列必须无歧义地引用一个分组列（除非该引用出现在一个聚集 函数中或者该非分组列函数依赖于分组列。

即使没有GROUP BY子句，HAVING 的存在也会把一个查询转变成一个分组查询。这和查询中包含聚集函数但没有 GROUP BY子句时的情况相同。所有被选择的行都被认为是一个 单一分

组，并且SELECT列表和 HAVING子句只能引用聚集函数中的表列。如果该 HAVING条件为真，这样一个查询将会发出一个单一行； 否则不返回行。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能与 HAVING一起指定。

WINDOW 子句

可选的WINDOW子句的形式

```
WINDOW window_name AS ( window_definition ) [, ...]
```

其中window_name 是一个可以从OVER子句或者后续窗口定义中引用的名称。window_definition是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
[ frame_clause ]
```

如果指定了一个existing_window_name， 它必须引用WINDOW列表中一个更早出现的项。新窗口将从 该项中复制它的划分子句以及排序子句（如果有）。在这种情况下，新窗口 不能指定它自己的PARTITION BY子句，并且它只能在被复制 窗口没有ORDER BY的情况下指定该子句。新窗口总是使用它 自己的帧子句，被复制的窗口不必指定一个帧子句。

PARTITION BY列表元素的解释以 GROUP BY 子句元素的方式 进行，不过它们总是简单表达式并且绝不能是输出列的名称或编号。另一个区 别是这些表达式可以包含聚集函数调用，而这在常规GROUP BY 子句中是不被允许的。它们被允许的原因是窗口是出现在分组和聚集之后的。

类似地，ORDER BY列表元素的解释也以 ORDER BY 子句元素的方式进行， 不过该表达式总是被当做简单表达式并且绝不会是输出列的名称或编号。

可选的frame_clause为依赖帧的窗口函数 定义窗口帧（并非所有窗口函数都依赖于帧）。窗口帧是查询中 每一样（称为当前行）的相关行的集合。 frame_clause可以是

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

之一，其中frame_start和frame_end可以是

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

之一，并且frame_exclusion可以是

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

之一。如果省略frame_end，它会被默认为CURRENT ROW。限制是： frame_start不能是UNBOUNDED FOLLOWING， frame_end不能是UNBOUNDED PRECEDING， 并且frame_end的选择在上面of frame_start以及frame_end 选项的列表中不能早于 frame_start的选择 — 例如RANGE BETWEEN CURRENT ROW AND offset PRECEDING是不被允许的。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。它把帧设置为从分区开始直到当前行的最后一个平级行（被该窗口的ORDER BY子句认为等价于当前行的行，如果没有ORDER BY则所有的行都是平级的）。通常， UNBOUNDED PRECEDING表示从分区第一行开始的帧，类似地 UNBOUNDED FOLLOWING表示以分区最后一行结束的帧，不论是处于RANGE、ROWS或者GROUPS模式中。在ROWS模式中， CURRENT ROW表示以当前行开始或者结束的帧。而在 RANGE或者GROUPS模式中它表示当前行在ORDER BY排序中的第一个 或者最后一个平级行开始或者结束的帧。 offset PRECEDING和offset FOLLOWING选项的含义会随着帧模式而变化。在ROWS模式中，offset是一个整数，表示帧开始或者结束于当前行之前或者之后的那么多行处。在GROUPS模式中，offset是一个整数，表示真开始或者结束于当前行的平级组之前或者之后那么多个平级组处，其中平级组是一组根据窗口的ORDER BY子句等效的行。在RANGE模式中，offset选项的使用要求在窗口定义中正好有一个ORDER BY列。那么该帧包含的行的排序列值不超过offset且小于（对于PRECEDING）或者大于（对于FOLLOWING）当前行的排序列值。在这些情况中，offset表达式的数据类型取决于排序列的数据类型。对于数字排序列，它通常与排序列是相同类型，但对于datetime类型的排序列它是interval。在所有这些情况中，offset的值必须是非空和非负。此外，虽然offset并非必须是简单常量，但它不能包含变量、聚集函数或者窗口函数。

frame_exclusion选项允许从帧中排除当前行周围的行，即便根据帧的起始选项来说它们应该被包含在帧中。EXCLUDE CURRENT ROW把当前行从帧中排除。EXCLUDE GROUP把当前行和它在排序上的平级行从帧中排除。EXCLUDE TIES从帧中排除当前行的任何平级行，但是不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的默认行为。

注意，如果ORDER BY排序无法把行唯一地排序，则ROWS模式可能产生不可预测的结果。RANGE以及GROUPS模式的目的是确保在ORDER BY顺序中平等的行被同样对待：一个给定平级组中的所有行将在一个帧中或者被从帧中排除。

WINDOW子句的目的是指定出现在查询的 SELECT 列表或 ORDER BY 子句中的 窗口函数的行为。这些函数可以在它们的 OVER子句中用名称引用WINDOW 子句项。不过，WINDOW子句项不是必须被引用。 如果在查询中没有用到它，它会被简单地忽略。可以使用根本没有任何 WINDOW子句的窗口函数，因为窗口函数调用可 以直接在其OVER子句中指定它的窗口定义。不过，当多 个窗口函数都需要相同的窗口定义时， WINDOW子句能够减少输入。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和 WINDOW一起被指定。

窗口函数的详细描述在 第 3.5 节 第 4.2.8 节及 第 7.2.5 节。

SELECT 列表

SELECT列表（位于关键词 SELECT和FROM之间）指定构成 SELECT语句输出行的表达式。这些表达式 可以（并且通常确实会）引用FROM子句中计算得到的列。

正如在表中一样，SELECT的每一个输出列都有一个名称。 在一个简单的SELECT中，这个名称只是被用来标记要显 示的列，但是当SELECT是一个大型查询的一个子查询时，大型查询会把该名称看做子查询产生的虚表的列名。要指定用于输出列的名称，在该列的表达式 后面写上 AS output_name（ 你可以省略AS，但只能在期望的输出名称不匹配任何 PostgreSQL关键词（见附录 C 时省略。为了避免和未来增加的关键词冲突， 推荐总是写上AS或者用双引号引用输出名称）。如果你不指定列名， PostgreSQL会自动选择一个名称。如果列的表达式 是一个简单的列引用，那么被选择的名称就和该列的名称相同。在使用函数或者类型名称 的更复杂的情况中，系统可能会生成诸如 ?column?之类的名称。

一个输出列的名称可以被用来在ORDER BY以及 GROUP BY子句中引用该列的值，但是不能用于WHERE和HAVING子句（在其中 必须写出表达式）。

可以在输出列表中写*来取代表达式，它是被选中 行的所有列的一种简写方式。还可以写table_name.*，它 是只来自那个表的所有列的简写形式。在这些情况中无法用 AS指定新的名称，输出行的名称将和表列的名称相同。

根据 SQL 标准，输出列表中的表达式应该在应用DISTINCT、ORDER BY或者LIMIT之前计算。在使用DISTINCT时显然必须这样做，否则就无法搞清到底在区分什么值。不过，在很多情况下如果先计算ORDER BY和LIMIT再计算输出表达式会很方便，特别是如果输出列表中包含任何 volatile 函数或者代价昂贵的函数时尤其如此。通过这种行为，函数计算的顺序更加直观并且对于从未出现在输出中的行将不会进行计算。只要输出表达式没有被DISTINCT、ORDER BY或者GROUP BY引用，PostgreSQL实际将在排序和限制行数之后计算输出表达式（作为一个反例，SELECT f(x) FROM tab ORDER BY 1显然必须在排序之前计算f(x)）。包含有集合返回函数的输出表达式实际是在排序之后和限制行数之前被计算，这样LIMIT才能切断来自集合返回函数的输出。

> **注意**
>
> 9.6 版本之前的PostgreSQL不对执行输出表达式、排序、限制行数的时间顺序做任何保证，那将取决于被选中的查询计划的形式。

DISTINCT 子句

如果指定了SELECT DISTINCT，所有重复的行会被从结果 集中移除（为每一组重复的行保留一行）。SELECT ALL则 指定相反的行为：所有行都会被保留，这也是默认情况。

SELECT DISTINCT ON (expression [, ...]) 只保留在给定表达式上计算相等的行集合中的第一行。 DISTINCT ON表达式使用和 ORDER BY相同的规则（见上文）解释。注意，除非用ORDER BY来确保所期望的行出现在第一位，每一个集 合的“第一行”是不可预测的。例如：

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果我们不使用 ORDER BY来强制对每个地点的时间值进行降序排序， 我们为每个地点得到的报告的时间可能是无法预测的。

DISTINCT ON表达式必须匹配最左边的 ORDER BY表达式。ORDER BY子句通常 将包含额外的表达式，这些额外的表达式用于决定在每一个 DISTINCT ON分组内行的优先级。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和 DISTINCT一起使用。

UNION 子句

UNION子句具有下面的形式：

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

select_statement 是任何没有ORDER BY、LIMIT、 FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和FOR KEY SHARE子句的 SELECT语句（如果子表达式被包围在圆括号内， ORDER BY和LIMIT可以被附着到其上。如果没有 圆括号，这些子句将被应用到UNION的结果而不是右手边 的表达式上）。

UNION操作符计算所涉及的 SELECT语句所返回的行的并集。如果一行 至少出现在两个结果集中的一个内，它就会在并集中。作为 UNION两个操作数的 SELECT语句必须产生相同数量的列并且 对应位置上的列必须具有兼容的数据类型。

UNION的结果不会包含重复行，除非指定了 ALL选项。ALL会阻止消除重复（因此， UNION ALL通常显著地快于UNION， 尽量使用ALL）。可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 UNION操作符会从左至右计算。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于UNION结果或者 UNION的任何输入。

INTERSECT 子句

INTERSECT子句具有下面的形式：

```
select_statement INTERSECT [ ALL | DISTINCT ] select_statement
```

select_statement 是任何没有ORDER BY, LIMIT、FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

INTERSECT操作符计算所涉及的 SELECT语句返回的行的交集。如果 一行同时出现在两个结果集中，它就在交集中。

INTERSECT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有 m次重复并且在右表中有n 次重复的行将会在结果中出现 min(m,n) 次。 DISTINCT可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 INTERSECT操作符会从左至右计算。 INTERSECT的优先级比 UNION更高。也就是说， A UNION B INTERSECT C将被读成A UNION (B INTERSECT C)。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于INTERSECT结果或者 INTERSECT的任何输入。

EXCEPT 子句

EXCEPT子句具有下面的形式：

```
select_statement EXCEPT [ ALL | DISTINCT ] select_statement
```

select_statement 是任何没有ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

EXCEPT操作符计算位于左 SELECT语句的结果中但不在右 SELECT语句结果中的行集合。

EXCEPT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有 m次重复并且在右表中有 n次重复的行将会在结果集中出现 max(m-n,0) 次。 DISTINCT可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 EXCEPT操作符会从左至右计算。 EXCEPT的优先级与 UNION相同。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于EXCEPT结果或者 EXCEPT的任何输入。

ORDER BY 子句

可选的ORDER BY子句的形式如下：

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
 [, ...]
```

ORDER BY子句导致结果行被按照指定的表达式排序。 如果两行按照最左边的表达式是相等的，则会根据下一个表达式比较它们， 依次类推。如果按照所有指定的表达式它们都是相等的，则它们被返回的 顺序取决于实现。

每一个expression 可以是输出列（SELECT列表项）的名称或 者序号，它也可以是由输入列值构成的任意表达式。

序号指的是输出列的顺序（从左至右）位置。这种特性可以为不具有唯一 名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS子句为输出列赋予一个名称。

也可以在ORDER BY子句中使用任意表达式，包括没 有出现在SELECT输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在UNION、 INTERSECT或EXCEPT子句结果上的 ORDER BY只能指定输出列名称或序号，但不能指定表达式。

如果一个ORDER BY表达式是一个既匹配输出列名称又匹配 输入列名称的简单名称，ORDER BY将把它解读成输出列名 称。这与在同样情况下GROUP BY会做出的选择相反。这种 不一致是为了与 SQL 标准兼容。

可以为ORDER BY子句中的任何表达式之后增加关键词 ASC（上升）DESC（下降）。如果没有指定， ASC被假定为默认值。或者，可以在USING 子句中指定一个特定的排序操作符名称。一个排序操作符必须是某个 B-树操作符族的小于或者大于成员。ASC通常等价于 USING <而DESC通常等价于 USING >（但是一种用户定义数据类型的创建者可以 准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

如果指定NULLS LAST，空值会排在非空值之后；如果指定 NULLS FIRST，空值会排在非空值之前。如果都没有指定， 在指定或者隐含ASC时的默认行为是NULLS LAST， 而指定或者隐含DESC时的默认行为是 NULLS FIRST（因此，默认行为是空值大于非空值）。 当指定USING时，默认的空值顺序取决于该操作符是否为 小于或者大于操作符。

注意顺序选项只应用到它们所跟随的表达式上。例如 ORDER BY x, y DESC和 ORDER BY x DESC, y DESC是不同的。

字符串数据会被根据引用到被排序列上的排序规则排序。根据需要可以通过在expression中包括一个 COLLATE子句来覆盖，例如 ORDER BY mycolumn COLLATE "en_US"。更多信息请见 第 4.2.10 和 第 23.2 节

LIMIT 子句

LIMIT子句由两个独立的子句构成：

```
LIMIT { count | ALL }
OFFSET start
```

count指定要返回 的最大行数，而start 指定在返回行之前要跳过的行数。在两者都被指定时，在开始计算要返回的 count行之前会跳过 start行。

如果count表达式计算 为 NULL，它会被当成LIMIT ALL，即没有限制。如果 start计算为NULL，它会被当作OFFSET 0。

SQL:2008 引入了一种不同的语法来达到相同的结果， PostgreSQL也支持它：

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY
```

在这种语法中，标准要求start或count是一个文本常量、一个参数或者一个变量名。而作为一种 PostgreSQL的扩展，还允许其他的表达式，但通常需要被封闭在圆括号中以避免歧义。如果在一个 FETCH子句中省略 count，它的默认值为 1。 ROW和ROWS以及FIRST和NEXT是噪声，它们不影响 这些子句的效果。根据标准，如果都存在，OFFSET子句 必须出现在FETCH子句之前。但是 PostgreSQL更宽松，它允许两种顺序。

在使用LIMIT时，用一个ORDER BY子句把 结果行约束到一个唯一顺序是个好办法。否则你讲得到该查询结果行的 一个不可预测的子集 — 你可能要求从第 10 到第 20 行，但是在 什么顺序下的第 10 到第 20 呢？除非指定ORDER BY，你 是不知道顺序的。

查询规划器在生成一个查询计划时会考虑LIMIT，因此 根据你使用的LIMIT和OFFSET，你很可能 得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET值来选择一个查询结果的 不同子集将会给出不一致的结果，除非你 用ORDER BY强制一种可预测的结果顺序。这不是一个 缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY来约束顺序）给出一个查询结果这一事实造 成的必然后果。

如果没有一个ORDER BY来强制选择一个确定的子集， 重复执行同样的LIMIT查询甚至可能会返回一个表中行 的不同子集。同样，这也不是一种缺陷，再这样一种情况下也无法 保证结果的确定性。

锁定子句

FOR UPDATE、FOR NO KEY UPDATE、 FOR SHARE和FOR KEY SHARE 是锁定子句，它们影响SELECT把行从表中取得时如何对它们加锁。

锁定子句的一般形式：

```
FOR lock_strength [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

其中lock_strength可以是

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

之一。

更多关于每一种行级锁模式的信息可见 第 13.3.2 节

为了防止该操作等待其他事务提交，可使用NOWAIT或者 SKIP LOCKED选项。使用NOWAIT时，如果选中的行不能被立即锁定，该语句会报告错误而不是等待。使用 SKIP LOCKED时，无法被立即锁定的任何选中行都 会被跳过。跳过已锁定行会提供数据的一个不一致的视图，因此这不适合 于一般目的的工作，但是可以被用来避免多个用户访问一个类似队列的表 时出现锁竞争。注意NOWAIT和 SKIP LOCKED只适合行级锁 — 所要求的 ROW SHARE表级锁仍然会以常规的方式（见 第 13 章取得。如果想要不等待的表级锁，你可以先 使用带NOWAIT的LOCK。

如果在一个锁定子句中提到了特定的表，则只有来自于那些表的 行会被锁定，任何SELECT中用到的 其他表还是被简单地照常读取。一个没有表列表的锁定子句会影响 该语句中用到的所有表。如果一个锁定子句被应用到一个视图或者 子查询，它会影响在该视图或子查询中用到的所有表。不过，这些 子句不适用于主查询引用的WITH查询。如果你希望在一个WITH查询中发生行锁定，应该在该 WITH查询内指定一个锁定子句。

如果有必要对不同的表指定不同的锁定行为，可以写多个锁定子句。 如果同一个表在多于一个锁定子句中被提到（或者被隐式的影响到）， 那么会按照所指定的最强的锁定行为来

处理它。类似地，如果在任何 影响一个表的子句中指定了NOWAIT，就会按照 NOWAIT的行为来处理该表。否则如果 SKIP LOCKED在任何影响该表的子句中被指定， 该表就会被按照SKIP LOCKED来处理。

如果被返回的行无法清晰地与表中的行保持一致，则不能使用锁定子句。 例如锁定子句不能与聚集一起使用。

当一个锁定子句出现在一个SELECT查询的顶层时， 被锁定的行正好就是该查询返回的行。在连接查询的情况下，被锁定 的行是那些对返回的连接行有贡献的行。此外，自该查询的快照起满足 查询条件的行将被锁定，如果它们在该快照后被更新并且不再满足 查询条件，它们将不会被返回。如果使用了LIMIT，只要 已经返回的行数满足了限制，锁定就会停止（但注意被 OFFSET跳过的行将被锁定）。类似地，如果在一个游标 的查询中使用锁定子句，只有被该游标实际取出或者跳过的行才将被 锁定。

当一个锁定子句出现在一个子-SELECT中时，被锁定 行是那些该子查询返回给外层查询的行。这些被锁定的行的数量可能比 从子查询自身的角度看到的要少，因为来自外层查询的条件可能会被用 来优化子查询的执行。例如：

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

将只锁定具有col1 = 5的行（虽然在子查询中并没有写上 该条件）。

早前的发行无法维持一个被之后的保存点升级的锁。例如，这段代码：

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

在ROLLBACK TO之后将无法维持 FOR UPDATE锁。在 9.3 中已经修复这个问题。

> **小心**
>
> 一个运行在READ COMMITTED事务隔离级别并且使用ORDER BY和锁定子句的SELECT命令有可能返回无序的行。 这是因为ORDER BY会被首先应用。该命令对结果排序，但是可能 接着在尝试获得一个或者多个行上的锁时阻塞。一旦SELECT解除 阻塞，某些排序列值可能已经被修改，从而导致那些行变成无序的（尽管它们根 据原始列值是有序的）。根据需要，可以通过在子查询中放置 FOR UPDATE/SHARE来解决之一问题，例如
>
> ```
> SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY
>  column1;
> ```
>
> 注意这将导致锁定mytable的所有行，而顶层的 FOR UPDATE只会锁定实际被返回的行。这可能会导致显著的 性能差异，特别是把ORDER BY与LIMIT或者其他 限制组合使用时。因此只有在并发更新排序列并且要求严格的排序结果时才推 荐使用这种技术。
>
> 在REPEATABLE READ或者SERIALIZABLE 事务隔离级别上这可能导致一个序列化失败（SQLSTATE 是'40001'），因此在这些隔离级别下不可能收到无序行。

TABLE 命令

命令

TABLE name

等价于

```
SELECT * FROM name
```

它可以被用作一个顶层命令，或者用在复杂查询中以节省空间。只有 WITH、UNION、INTERSECT、EXCEPT、 ORDER BY、LIMIT、OFFSET、 FETCH以及FOR锁定子句可以用于TABLE。不能使用WHERE子句和任何形式 的聚集。

示例

把表films与表 distributors连接：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did
```

```
       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

要对所有电影的len列求和并且用 kind对结果分组：

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

```
   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

要对所有电影的len列求和、对结果按照 kind分组并且显示总长小于 5 小时的分组：

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';
```

```
   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

下面两个例子都是根据第二列（name）的内容来排序结果：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
```

```
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

接下来的例子展示了如何得到表distributors和 actors的并集，把结果限制为那些在每个表中以 字母 W 开始的行。只想要可区分的行，因此省略了关键词 ALL。

```
distributors:               actors:
 did |     name              id |     name
-----+--------------       ----+----------------
 108 | Westward              1 | Woody Allen
 111 | Walt Disney           2 | Warren Beatty
 112 | Warner Bros.          3 | Walter Matthau
 ...                         ...
```

```
SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

这个例子展示了如何在FROM子句中使用函数， 分别使用和不使用列定义列表：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

这里是带有增加的序数列的函数的例子：

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |          1
 b      |          2
 c      |          3
 d      |          4
 e      |          5
 f      |          6
(6 rows)
```

这个例子展示了如何使用简单的WITH子句：

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

注意该WITH查询只被计算一次，这样我们得到的两个 集合具有相同的三个随机值。

这个例子使用WITH RECURSIVE从一个只显示 直接下属的表中寻找雇员 Mary 的所有下属（直接的或者间接的）以及他们的间接层数：

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

注意这种递归查询的典型形式：一个初始条件，后面跟着 UNION，然后是查询的递归部分。要确保 查询的递归部分最终将不返回任何行，否则该查询将无限循环（ 更多例子见第 7.8 节。

这个例子使用LATERAL为manufacturers 表的每一行应用一个集合返回函数get_product_names()：

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

当前没有任何产品的制造商不会出现在结果中，因为这是一个内连接。 如果我们希望把这类制造商的名称包括在结果中，我们可以：

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

兼容性

当然，SELECT语句是兼容 SQL 标准的。 但是也有一些扩展和缺失的特性。

省略的FROM子句

PostgreSQL允许我们省略 FROM子句。一种简单的使用是计算简单表达式 的结果：

```
SELECT 2+2;

 ?column?
----------
        4
```

某些其他SQL数据库需要引入一个假的 单行表放在该SELECT的 FROM子句中才能做到这一点。

注意，如果没有指定一个FROM子句，该查询 就不能引用任何数据库表。例如，下面的查询是非法的：

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL在 8.1 之前的发行 会接受这种形式的查询，并且为该查询引用的每一个表在FROM子句中隐式增加一个项。现在已经不再允许 这样做。

空SELECT列表

SELECT之后的输出表达式列表可以为空， 这会产生一个零列的结果表。对 SQL 标准来说这不是合法的 语法。PostgreSQL允许 它是为了与允许零列表保持一致。不过在使用DISTINCT时不允许空列表。

省略AS关键词

在 SQL 标准中，只要新列名是一个合法的列名（就是说与任何保留关键词不同）， 就可以省略输出列名之前的可选关键词AS。 PostgreSQL要稍微严格些：只要新列名匹配 任何关键词（保留或者非保留）就需要AS。推荐的习惯是使用 AS或者带双引号的输出列名来防止与未来增加的关键词可能的冲突。

在FROM项中，标准和 PostgreSQL都允许省略非保留 关键词别名之前的AS。但是由于语法的歧义，这无法 用于输出列名。

ONLY和继承

在书写ONLY时，SQL 标准要求在表名周围加上圆括号，例如 SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...。PostgreSQL 认为这些圆括号是可选的。

PostgreSQL允许写一个拖尾的*来 显式指定包括子表的非-ONLY行为。而标准则不允许 这样。

（这些点同等地适用于所有支持ONLY选项的 SQL 命令）。

TABLESAMPLE子句限制

当前只在常规表和物化视图上接受TABLESAMPLE子句。 根据 SQL 标准，应该可以把它应用于任何FROM项。

FROM中的函数调用

PostgreSQL允许一个函数调用被直接写作 FROM列表的一个成员。在 SQL 标准中，有必要把这样一个函数 调用包裹在一个子-SELECT中。也就是说，语法 FROM func(...) alias 近似等价于 FROM LATERAL (SELECT func(...)) alias。 注意该LATERAL被认为是隐式的，这是因为标准对于 FROM中的一个UNNEST()项要求 LATERAL语义。PostgreSQL会把 UNNEST()和其他集合返回函数同样对待。

GROUP BY和ORDER BY可用的名字空间

在 SQL-92 标准中，一个ORDER BY子句只能使用输出 列名或者序号，而一个GROUP BY子句只能使用基于输 入列名的表达式。PostgreSQL扩展了 这两种子句以允许它们使用其他的选择（但如果有歧义时还是使用标准的 解释）。PostgreSQL也允许两种子句 指定任意表达式。注意出现在一个表达式中的名称将总是被当做输入列名而 不是输出列名。

SQL:1999 及其后的标准使用了一种略微不同的定义，它并不完全向后兼容 SQL-92。不过，在大部分的情况下， PostgreSQL会以与 SQL:1999 相同的 方式解释ORDER BY或GROUP BY表达式。

函数依赖

只有当一个表的主键被包括在GROUP BY列表中时， PostgreSQL才识别函数依赖（允许从GROUP BY中省略列）。SQL 标准指定了应该要识别 的额外情况。

LIMIT和OFFSET

LIMIT和OFFSET子句是 PostgreSQL-特有的语法，在 MySQL也被使用。SQL:2008 标准已经 引入了具有相同功能的子句OFFSET ... FETCH {FIRST|NEXT} ...（如上文 LIMIT 子句中所示）。这种语法 也被IBM DB2使用（ Oracle编写的应用常常使用自动生成的 rownum列来实现这些子句的效果，这在 PostgreSQL 中是没有的）。

FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE

尽管 SQL 标准中出现了FOR UPDATE，但标准只允许它作为 DECLARE CURSOR的一个选项。PostgreSQL允许它出现在任何 SELECT查询以及子-SELECT中，但这是 一种扩展。FOR NO KEY UPDATE、FOR SHARE 以及FOR KEY SHARE变体以及NOWAIT 和SKIP LOCKED选项没有在标准中出现。

WITH中的数据修改语句

PostgreSQL允许把INSERT、 UPDATE以及DELETE用作WITH 查询。这在 SQL 标准中是找不到的。

非标准子句

DISTINCT ON (...)是 SQL 标准的扩展。

ROWS FROM(...)是 SQL 标准的扩展。

SELECT INTO

SELECT INTO — 从一个查询的结果定义一个新表

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
 LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

描述

SELECT INTO创建一个新表并且用一个查询 计算得到的数据填充它。这些数据不会像普通的SELECT那样被返回给客户端。新表的列具有 和SELECT的输出列相关的名称和数据类型。

参数

TEMPORARY or TEMP

如果被指定，该表被创建为一个临时表。详见 CREATE TABLE。

UNLOGGED

如果被指定，该表被创建为一个不做日志的表。详见 CREATE TABLE。

new_table

要创建的表的名字（可以是模式限定的）。

所有其他参数在SELECT中有详细描述。

注解

CREATE TABLE AS在功能上与 SELECT INTO相似。CREATE TABLE AS 是被推荐的语法，因为这种形式的SELECT INTO在ECPG 或PL/pgSQL中不可用，因为它们对 INTO子句的解释不同。此外， CREATE TABLE AS提供的功能是 SELECT INTO的超集。

要为SELECT INTO创建的表增加 OID， 启用default_with_oids配置变量。 CREATE TABLE AS可以使用 WITH OIDS子句。

示例

创建一个只由来自films的最近项构成的 新表films_recent：

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

兼容性

SQL 标准使用SELECT INTO表示把值选择 到一个宿主程序的标量变量中，而不是创建一个新表。这实际上就是 ECPG（见第 36 章和 PL/pgSQL（见第 43 章 中的用法。 PostgreSQL 使用 SELECT INTO的来表示表创建是有历史原因的。 最好在新代码中使用CREATE TABLE AS。

另见

CREATE TABLE AS

SET

SET — 更改一个运行时参数

大纲

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' |
 DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

描述

SET命令更改运行时配置参数。很多 第 19 章列出的参数可以用 SET即时更改（但是有些需要超级用户 特权才能更改，并且还有一些在服务器或者会话启动之后不能被更改）。SET只影响当前会话所使用的值。

如果在一个事务内发出SET （或者等效的SET SESSION）而该事务后来 中止，在该事务被回滚时SET命令的效果会 消失。一旦所在的事务被提交，这些效果将会持续到会话结束（除非被另 一个SET所覆盖）。

SET LOCAL的效果只持续到当前事务结束， 不管事务是否被提交。一种特殊情况是在一个事务内 SET后面跟着 SET LOCAL： SET LOCAL值将会在该事务结束前一直可见， 但是之后（如果该事务被提交）SET值将会生效。

SET或SET LOCAL 的效果也会因为回滚到早于它们的保存点而消失。

如果在一个函数内使用SET LOCAL并且该函数 还有对同一变量的SET选项（见 CREATE FUNCTION），在函数退出时 SET LOCAL命令的效果会消失。也就是说，该 函数被调用时的值会被恢复。这允许用 SET LOCAL在函数内动态地或者重复地更改 一个参数，同时仍然能便利地使用SET选项来保存以及恢复调用 者的值。不过，一个常规的SET命令会覆盖它所在的任何函 数的SET选项，除非回滚，它的效果将一直保持。

> **注意**
>
> 在PostgreSQL 版本 8.0 到 8.2 中， 一个SET LOCAL的效果会因为释放较早的 保存点或者成功地从一个PL/pgSQL异常块 中退出而被取消。这种行为已经被更改，因为它被认为不直观。

参数

SESSION

指定该命令对当前会话有效（这是默认值）。

LOCAL

指定该命令只对当前事务有效。在COMMIT或者 ROLLBACK之后，会话级别的设置会再次生效。 在事务块外部发出这个参数会发出一个警告并且不会有效果。

configuration_parameter

一个可设置运行时参数的名称。可用的参数被记录在 第 19 章中。

value

参数的新值。根据特定的参数，值可以被指定为字符串常量、标识符、 数字或者以上构成的逗号分隔列表。写DEFAULT 可以指定把该参数重置成它的默认值（也就是说在当前会话中还没有 执行SET命令时它具有的值）。

除了在第 19 章记录的配置参数， 还有一些参数只能用SET命令设置 或者具有特殊的语法：

SCHEMA

SET SCHEMA 'value'是 SET search_path TO value的一个别名。 使用这种语法只能指定一个模式。

NAMES

SET NAMES value是 SET client_encoding TO value的一个别名。

SEED

为随机数生成器（函数random）设置 一个内部种子。允许的值是 -1 和 1 之间的浮点数，它会被乘上 $2^{31}-1$。

也可以通过调用函数setseed来设置种子：

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE value是 SET timezone TO value的一个别 名。语法SET TIME ZONE允许用于时区指定的特 殊语法。这里是合法值的例子：

'PST8PDT'

加州伯克利的时区。

'Europe/Rome'

意大利的时区。

-7

UTC 以西 7 小时的时区（等效于 PDT）。正值则是 UTC 以东。

INTERVAL '-08:00' HOUR TO MINUTE

UTC 以西 8 小时的时区（等效于 PST）。

LOCAL
DEFAULT

把时区设置为你的本地时区（也就是说服务器的timezone默认值）。

以数字或区间给出的时区设置在内部被翻译成 POSIX 时区语法。 例如，在SET TIME ZONE -7之后， SHOW TIME ZONE将会报告 <-07>+07。

有关时区的更多信息可见第 8.5.3 节

注解

函数set_config提供了等效的功能，见 第 9.26 节此外，可以更新 pg_settings 系统视图来执行与SET等效的工作。

示例

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

把日期风格设置为传统 POSTGRES的 “日在月之前”的输入习惯：

```
SET datestyle TO postgres, dmy;
```

设置时区为加州伯克利：

```
SET TIME ZONE 'PST8PDT';
```

设置时区为意大利：

```
SET TIME ZONE 'Europe/Rome';
```

兼容性

SET TIME ZONE扩展了 SQL 标准定义的语法。标准 只允许数字的时区偏移量而 PostgreSQL允许更灵活的时区说明。 所有其他SET特性都是 PostgreSQL扩展。

另见

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — 为当前事务设置约束检查时机

大纲

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

描述

SET CONSTRAINTS设置当前事务内约束检查 的行为。IMMEDIATE约束在每个语句结束时被检查。 DEFERRED约束直到事务提交时才被检查。每个约束都有 自己的IMMEDIATE或DEFERRED模式。

在创建时，一个约束会被给定三种特性之一： DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE或者 NOT DEFERRABLE。第三类总是 IMMEDIATE并且不会受到 SET CONSTRAINTS命令的影响。前两类在每个 事务开始时都处于指定的模式，但是它们的行为可以在一个事务内用 SET CONSTRAINTS更改。

带有一个约束名称列表的SET CONSTRAINTS 只更改那些约束（都必须是可延迟的）的模式。每一个约束名称都可以是 模式限定的。如果没有指定模式名称，则当前的模式搜索路径将被用来寻找 第一个匹配的名称。SET CONSTRAINTS ALL 更改所有可延迟约束的模式。

当SET CONSTRAINTS把一个约束的模式从 DEFERRED改成IMMEDIATE时， 新模式会有追溯效果：任何还没有解决的数据修改（本来会在事务结束时 被检查）会转而在SET CONSTRAINTS命令 的执行期间被检查。如果任何这种约束被违背， SET CONSTRAINTS将会失败（并且不会改 变该约束模式）。这样，SET CONSTRAINTS可以被用来在一个事务中的特定点强制进 行约束检查。

当前，只有UNIQUE、PRIMARY KEY、 REFERENCES（外键）以及EXCLUDE 约束受到这个设置的影响。 NOT NULL以及CHECK约束总是在一行 被插入或修改时立即检查（不是在语句结束时）。 没有被声明为DEFERRABLE的唯一和排除约束也会被 立刻检查。

被声明为“约束触发器”的触发器的引发也受到这个设置 的控制 — 它们会在相关约束被检查的同时被引发。

注解

因为PostgreSQL并不要求约束名称在模式内 唯一（但是在表内要求唯一），可能有多于一个约束匹配指定的约束名称。在这种 情况下SET CONSTRAINTS将会在所有的匹配上操作。 对于一个非模式限定的名称，一旦在搜索路径中的某个模式中发现一个或者多个匹 配，路径中后面的模式将不会被搜索。

这个命令只修改当前事务内约束的行为。在事务块外部发出这个命令会产生一个 警告并且也不会有任何效果。

兼容性

这个命令符合 SQL 标准中定义的行为，但有一点限制：在 PostgreSQL中，它不会应用在NOT NULL和CHECK约束上。还有， PostgreSQL会立刻检查非可延迟的 唯一约束，而不是按照标准建议的在语句结束时检查。

SET ROLE

SET ROLE — 设置当前会话的当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

描述

这个命令把当前 SQL 会话的当前用户标识符设置为 role_name。 角色名可以写成一个标识符或者一个字符串。在 SET ROLE之后，对 SQL 命令的权限检查时就 好像该角色就是原先登录的角色一样。

当前会话用户必须是指定的 角色role_name 的一个成员（如果会话用户是一个超级用户，则可以选择任何角色）。

SESSION和LOCAL修饰符发挥的作用和 常规的SET命令一样。

NONE和RESET形式把当前用户标识符 重置为当前会话用户标识符。这些形式可以由任何用户执行。

注解

使用这个命令可以增加特权或者限制特权。如果会话用户角色具有 INHERITS属性，则它会自动具有它能 SET ROLE到的所有角色的全部特权。在这种情况下 SET ROLE实际会删除所有直接分配给会话用户的特 权以及分配给会话用户作为其成员的其他角色的特权，只留下所提及 角色可用的特权。换句话说，如果会话用户没有 NOINHERITS属性，SET ROLE会删除 直接分配给会话用户的特权而得到所提及角色可用的特权。

特别地，当一个超级用户选择SET ROLE到一个非 超级用户角色时，它们会丢失其超级用户特权。

SET ROLE的效果堪比 SET SESSION AUTHORIZATION，但是涉及的特权检查 完全不同。还有，SET SESSION AUTHORIZATION决定 后来的SET ROLE命令可以使用哪些角色， 不过用 SET ROLE更改角色并不会改变后续 SET ROLE能够使用的角色集。

SET ROLE不会处理角色的ALTER ROLE 设置指定的会话变量。这只在登录期间发生。

SET ROLE不能在一个 SECURITY DEFINER函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

兼容性

PostgreSQL允许标识符 语法（"rolename"），而 SQL 标准要求 角色名被写成字符串。SQL 不允许在事务中使用这个命令，而 PostgreSQL并不做此限 制，因为并没有原因需要这样做。和RESET语法 一样，SESSION和 LOCAL修饰符是一种 PostgreSQL扩展。

另见

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — 设置当前会话的会话用户标识符和当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

描述

这个命令把当前 SQL 会话的会话用户标识符和当前用户标识符设置为 user_name。 用户名可以被写成一个标识符或者一个字符串。例如，可以使用这个 命令临时成为一个无特权用户并且稍后切换回来成为一个超级用户。

会话用户标识符初始时被设置为客户端提供的（可能已认证的）用户名。 当前用户标识符通常等于会话用户标识符，但是可能在 SECURITY DEFINER函数和类似机制的环境中临时更改。 也可以用SET ROLE更改它。当前用户标识符与 权限检查相关。

会话用户标识符只能在初始会话用户 已认证用户具有超级用户特权时被更改。 否则，只有该命令指定已认证用户名时才会被接受。

SESSION和LOCAL修饰符发挥的作用和常规 SET命令一样。

DEFAULT和RESET形式把会话用户标识符和 当前用户标识符重置为初始的已认证用户名。这些形式可以由任何用户执行。

注解

SET SESSION AUTHORIZATION不能在一个 SECURITY DEFINER函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许一些其他表达式出现在文本 user_name的位置上，但是实际上这些 选项并不重要。PostgreSQL允许标 识符语法（"username"），而 SQL 标准不允许。 SQL 不允许在事务

中使用这个命令，而 PostgreSQL并不做此限 制，因为并没有原因需要这样做。和RESET语法一样，SESSION和 LOCAL修饰符是一种 PostgreSQL扩展。

标准把执行这个命令所需的特权留给实现定义。

另见

SET ROLE

SET TRANSACTION

SET TRANSACTION — 设置当前事务的特性

大纲

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

其中 transaction_mode 是下列之一：

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

SET TRANSACTION命令设置当前 会话的特性。SET SESSION CHARACTERISTICS设置一个会话后续事务的默认 事务特性。在个体事务中可以用 SET TRANSACTION覆盖这些默认值。

可用的事务特性是事务隔离级别、事务访问模式（读/写或只读）以及 可延迟模式。此外，可以选择一个快照，不过只能用于当前事务而不能 作为会话默认值。

一个事务的隔离级别决定当其他事务并行运行时该事务能看见什么数据：

READ COMMITTED

一个语句只能看到在它开始前提交的行。这是默认值。

REPEATABLE READ

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句之前提交的行。

SERIALIZABLE

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句之前提交的行。如果并发的可序列化事务间的读写 模式可能导致一种那些事务串行（一次一个）执行时不可能出现 的情况，其中之一将会被回滚并且得到一个serialization_failure错误。

SQL 标准定义了一种额外的级别：READ UNCOMMITTED。在 PostgreSQL中READ UNCOMMITTED被视作 READ COMMITTED。

一个事务执行了第一个查询或者数据修改语句（ SELECT、 INSERT、DELETE、UPDATE、FETCH或 COPY）之后就无法更改事务隔离级别。 更多有关事务隔离级别和并发控制的信息可见第 13 章

事务的访问模式决定该事务是否为读/写或者只读。读/写是默认值。 当一个事务为只读时，如果 SQL 命令 INSERT、UPDATE、 DELETE和COPY FROM 要写的表不是一个临时表，则它们不被允许。不允许 CREATE、ALTER以及 DROP命令。不允许COMMENT、 GRANT、REVOKE、TRUNCATE。如果EXPLAIN ANALYZE 和EXECUTE要执行的命令是上述命令之一， 则它们也不被允许。这是一种高层的只读概念，它不能阻止所有对 磁盘的写入。

只有事务也是SERIALIZABLE以及 READ ONLY时，DEFERRABLE 事务属性才会有效。当一个事务的所有这三个属性都被选择时，该事务在 第一次获取其快照时可能会阻塞，在那之后它运行时就不会有 SERIALIZABLE事务的开销并且不会有任何牺牲或者 被一次序列化失败取消的风险。这种模式很适合于长时间运行的报表或者 备份。

SET TRANSACTION SNAPSHOT命令允许新的事务 使用与一个现有事务相同的快照运行。已经存在的事务 必须已经把它的快照用pg_export_snapshot函数（ 见第 9.26.5 节导出。 该函数会返回一个快照标识符，SET TRANSACTION SNAPSHOT需要被给定一个快照标识符来指定要导入的快照。 在这个命令中该标识符必须被写成一个字符串，例如 '000003A1-1'。 SET TRANSACTION SNAPSHOT只能在一个事务的 开始执行，并且要在该事务的第一个查询或者数据修改语句（ SELECT、 INSERT、DELETE、 UPDATE、FETCH或 COPY）之前执行。此外，该事务必须已经被设置 为SERIALIZABLE或者 REPEATABLE READ隔离级别（否则，该快照将被立刻抛弃， 因为READ COMMITTED模式会为每一个命令取一个新快照）。 如果导入事务使用了SERIALIZABLE隔离级别，那么导入快照 的事务必须也使用该隔离级别。还有，一个非只读可序列化事务不能导入来自只读 事务的快照。

注解

如果执行SET TRANSACTION之前没有 START TRANSACTION或者 BEGIN，它会发出一个警告并且不会有任何效果。

可以通过在BEGIN或者 START TRANSACTION中指定想要的transaction_modes来省掉 SET TRANSACTION。但是在 SET TRANSACTION SNAPSHOT中该选项不可用。

会话默认的事务模式也可以通过设置配置参数 default_transaction_isolation、default_transaction_read_only和 default_transaction_deferrable来设置（实际上 SET SESSION CHARACTERISTICS只是用 SET设置这些变量的等效体）。这意味着可以通过配置文件、 ALTER DATABASE等方式设置默认值。详见 第 19 章

示例

要用一个已经存在的事务的同一快照开始一个新事务，首先要从该现有 事务导出快照。这将会返回快照标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

然后在一个新开始的事务的开头把该快照标识符用在一个 SET TRANSACTION SNAPSHOT命令中：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

兼容性

SQL标准中定义了这些命令，不过 DEFERRABLE事务模式和 SET TRANSACTION SNAPSHOT形式除外，这两者是 PostgreSQL扩展。

SERIALIZABLE是标准中默认的事务隔离级别。在 PostgreSQL中默认值是普通的 READ COMMITTED，但是你可以按上述的方式更改。

在 SQL 标准中，可以用这些命令设置一个其他的事务特性：诊断区域 的尺寸。这个概念与嵌入式 SQL 有关，并且因此没有在 PostgreSQL服务器中实现。

SQL 标准要求连续的transaction_modes之间有逗号， 但是出于历史原因 PostgreSQL允许省略逗号。

SHOW

SHOW — 显示一个运行时参数的值

大纲

```
SHOW name
SHOW ALL
```

描述

SHOW将显示运行时参数的当前设置。 这些变量可以使用SET语句、编辑 postgresql.conf配置参数、通过 PGOPTIONS环境变量（使用 libpq或者基于libpq的应用时） 或者启动postgres服务器时通过命令行 标志设置。详见第 19 章

参数

name

一个运行时参数的名称。可用的参数记录在 第 19 章和SET参考页。此外，有一些可以显示但不能设置的参数：

SERVER_VERSION

显示服务器的版本号。

SERVER_ENCODING

显示服务器端的字符集编码。当前，这个参数可以被显示 但不能被设置，因为该设置是在数据库创建时决定的。

LC_COLLATE

显示数据库的排序规则（文本序）的区域设置。当前， 这个参数可以被显示但不能被设置，因为该设置是在 数据库创建时决定的。

LC_CTYPE

显示数据库的字符分类的区域设置。当前， 这个参数可以被显示但不能被设置，因为该设置 是在数据库创建时决定的。

IS_SUPERUSER

如果当前角色具有超级用户特权则为真。

ALL

显示所有配置参数的值，并带有描述。

注解

函数current_setting产生等效的输出，见 第 9.26 节还有， pg_settings 系统事务产生同样的信息。

示例

显示参数DateStyle的当前设置：

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

显示参数geqo的当前设置：

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

显示所有设置：

```
SHOW ALL;
            name          | setting |                 description

-------------------------+---------
+-----------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure
 of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit
 parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page
 headers.
(196 rows)
```

兼容性

SHOW命令是一种 PostgreSQL扩展。

另见

SET, RESET

START TRANSACTION

START TRANSACTION — 开始一个事务块

大纲

```
START TRANSACTION [ transaction_mode [, ...] ]
```

其中 transaction_mode 是下列之一：

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

这个命令开始一个新的事务块。如果指定了隔离级别、读写模式 或者可延迟模式，新的事务将会具有这些特性，就像执行了 SET TRANSACTION一样。这和 BEGIN命令一样。

参数

这些参数对于这个语句的含义可参考 SET TRANSACTION。

兼容性

在标准中，没有必要发出START TRANSACTION 来开始一个事务块：任何 SQL 命令会隐式地开始一个块。 PostgreSQL的行为可以被视作 在每个命令之后隐式地发出一个没有跟随在START TRANSACTION（ 或者BEGIN）之后的 COMMIT并且因此通常被称作 “自动提交”。为了方便，其他关系型数据库系统也可能会 提供自动提交特性。

DEFERRABLE transaction_mode 是一种PostgreSQL语言扩展。

SQL 标准要求在连续的transaction_modes之间有逗号， 但是由于历史原因PostgreSQL允许省略逗号。

另见SET TRANSACTION的兼容性小节。

另见

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — 清空一个表或者一组表

大纲

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

描述

TRUNCATE可以从一组表中快速地移除所有行。 它具有和在每个表上执行无条件DELETE相同的 效果，不过它会更快，因为它没有实际扫描表。此外，它会立刻回收磁盘空间， 而不是要求一个后续的VACUUM操作。在大表上 它最有用。

参数

name

要截断的表的名字（可以是模式限定的）。如果在表名前指定了 ONLY，则只会截断该表。如果没有指定ONLY， 该表及其所有后代表（如果有）都会被截断。可选地，可以在表名后指定 *来显式地包括后代表。

RESTART IDENTITY

自动重新开始被截断表的列所拥有的序列。

CONTINUE IDENTITY

不更改序列值。这是默认值。

CASCADE

自动截断所有对任一所提及表有外键引用的表以及任何由于 CASCADE被加入到组中的表。

RESTRICT

如果任一表上具有来自命令中没有列出的表的外键引用，则拒绝截断。这是默认值。

注解

要截断一个表，你必须具有其上的TRUNCATE特权。

TRUNCATE在要操作的表上要求一个 ACCESS EXCLUSIVE锁，这会阻塞所有其他在该表上的 并发操作。当指定RESTART IDENTITY时，任何需要被 重新开始的序列也会被排他地锁住。如果要求表上的并发访问，那么 应该使用DELETE命令。

TRUNCATE不能被用在被其他表外键引用的表上， 除非那些表也在同一个命令中被阶段。这些情况中的可行性检查将会 要求表扫描，并且重点不是为了做扫描。CASCADE 选项可以被用来自动地包括所有依赖表 — 但使用它时要非常 小心，否则你可能丢失数据！

TRUNCATE将不会引发表上可能存在的任何 ON DELETE触发器。但是它将会引发 ON TRUNCATE触发器。如果在这些表的任意一个 上定义了ON TRUNCATE触发器，那么所有的BEFORE TRUNCATE触发器将在任何截断发生之前 被引发，而所有AFTER TRUNCATE触发器将在

最后 一次截断完成并且所有序列被重置之后引发。触发器将以表被处理的顺 序被引发（首先是那些被列在命令中的，然后是由于级联被加入的）。

TRUNCATE不是 MVCC 安全的。截断之后， 如果并发事务使用的是一个在截断发生前取得的快照， 表将对这些并发事务呈现为空。详见第 13.5 节

从表中数据的角度来说，TRUNCATE是事务安全的： 如果所在的事务没有提交，阶段将会被安全地回滚。

在指定了RESTART IDENTITY时，隐含的 ALTER SEQUENCE RESTART操作也会被事务性地完成。也就是说，如果所在事务没有提交，它们也将被回滚。这和 ALTER SEQUENCE RESTART的通常行为不同。注意如果 事务回滚前在被重启序列上还做了额外的序列操作，这些操作在序列上的效果 也将被回滚，但是它们在currval()上的效果不会被回滚。也就 是说，在事务之后，currval()将继续反映在失败事务内得到的 最后一个序列值，即使序列本身可能已经不再与此一致。这和失败事务之后 currval()的通常行为类似。

TRUNCATE当前不支持外部表。 这表示如果一个指定的表具有任何外部的后代表，这个命令将会失败。

示例

截断表bigtable和 fattable：

```
TRUNCATE bigtable, fattable;
```

做同样的事情，并且还重置任何相关联的序列发生器：

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

截断表othertable，并且级联地截断任何通过 外键约束引用othertable的表：

```
TRUNCATE othertable CASCADE;
```

兼容性

SQL:2008 标准包括了一个TRUNCATE命令， 语法是TRUNCATE TABLE tablename。子句 CONTINUE IDENTITY/RESTART IDENTITY 也出现在了该标准中，但是含义有些不同。这个命令的一些并发行为被标准 留给实现来定义，因此如果必要应该考虑上述注解并且与其他实现进行比较。

参见

DELETE

UNLISTEN

UNLISTEN — 停止监听一个通知

大纲

```
UNLISTEN { channel | * }
```

描述

UNLISTEN被用来移除一个已经存在的对 NOTIFY事件的注册。 UNLISTEN取消任何已经存在的把当前 PostgreSQL会话作为名为 channel的通知 频道的监听者的注册。特殊的通配符*取消当前会话 的所有监听者注册。

NOTIFY包含有关LISTEN 和NOTIFY使用的更深入讨论。

参数

channel

一个通知频道的名称（任何标识符）。

*

所有用于这个会话的当前监听注册都会被清除。

注解

你可以 unlisten 你没有监听的东西，不会出现警告或者错误。

在每一个会话末尾，会自动执行UNLISTEN *。

一个已经执行了UNLISTEN的事务不能为 两阶段提交做准备。

示例

做一次注册：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

一旦执行了UNLISTEN，进一步的NOTIFY 消息将被忽略：

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

兼容性

SQL 标准中没有UNLISTEN命令。

另见

LISTEN, NOTIFY

UPDATE

UPDATE — 更新一个表的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] )
 |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

描述

UPDATE更改满足条件的所有行中指定列 的值。只有要被修改的列需要在SET子句中提及， 没有被显式修改的列保持它们之前的值。

有两种方法使用包含在数据库其他表中的信息来修改一个表：使用子选择 或者在FROM子句中指定额外的表。这种技术只适合 特定的环境。

可选的RETURNING子句导致UPDATE 基于实际被更新的每一行计算并且返回值。任何使用该表的列以及 FROM中提到的其他表的列的表达式都能被计算。 计算时会使用该表的列的新（更新后）值。RETURNING 列表的语法和SELECT的输出列表相同。

你必须拥有该表上的UPDATE特权，或者至少拥有 要被更新的列上的该特权。如果任何一列的值需要被 expressions或者 condition读取， 你还必须拥有该列上的SELECT特权。

参数

with_query

WITH子句允许你指定一个或者更多个在 UPDATE中可用其名称引用的子查询。详见第 7.8 和SELECT。

table_name

要更新的表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，只会更新所提及表中的匹配行。如果没有指定 ONLY，任何从所提及表继承得到的表中的匹配行也会 被更新。可选地，在表名之后指定*可以显式地指示要 把后代表也包括在内。

alias

目标表的一个替代名称。在提供了一个别名时，它会完全隐藏表的真实 名称。例如，给定UPDATE foo AS f， UPDATE语句的剩余部分必须用 f而不是foo来引用该表。

column_name

table_name 所指定的表的一列的名称。如果需要，该列名可以用一个子域名称或者 数组下标限定。不要在目标列的说明中包括表的名称 — 例如 UPDATE table_name SET table_name.col = 1是非法的。

expression

要被赋值给该列的一个表达式。该表达式可以使用该表中这一列或者 其他列的旧值。

DEFAULT

将该列设置为它的默认值（如果没有为它指定默认值表达式，默认值 将会为 NULL）。

sub-SELECT

一个SELECT子查询，它产生和在它之前的圆括号中列列表中 一样多的输出列。被执行时，该子查询必须得到不超过一行。如果它得到 一行，其列值会被赋予给目标列。如果它得不到行，NULL 值将被赋予给 目标列。该子查询可以引用被更新表中当前行的旧值。

from_list

表表达式的列表，允许来自其他表的列出现在WHERE 条件和更新表达式中。这类似于可以在 SELECT语句的FROM 子句中指定的表列表。注意目标表不能出现在 from_list中，除非你想做自连接（这种情况下它必须 以别名出现在from_list中）。

condition

一个返回boolean类型值的表达式。让这个 表达式返回true的行将会被更新。

cursor_name

要在WHERE CURRENT OF条件中使用的游标名。 要被更新的是从这个游标中最近取出的行。该游标必须是一个 在UPDATE目标表上的非分组查询。注意 WHERE CURRENT OF不能和一个布尔条件一起 指定。有关对游标使用WHERE CURRENT OF的 更多信息请见DECLARE。

output_expression

在每一行被更新后，要被UPDATE命令计算并且返回 的表达式。该表达式可以使用table_name指定 的表或者FROM列出的表中的任何列名。写* 可以返回所有列。

output_name

用于一个被返回列的名称。

输出

成功完成时，一个UPDATE命令返回形如

UPDATE count

的命令标签。 count是被更新的行数， 包括值没有更改的匹配行。注意，当更新被一个BEFORE UPDATE 触发器抑制时，这个数量可能比匹配 condition的行数少。如果 count为零，没有行被该查 询更新（这不是一个错误）。

如果UPDATE命令包含一个RETURNING 子句，其结果将类似于一个包含RETURNING列表中定义的 列和值的SELECT语句（在被该命令更新的行上计算） 的结果。

注解

当存在FROM子句时，实际发生的是：目标表被连接到 from_list中的表，并且该连接的每一个输出行表示对目标表的一个更新操作。在使用FROM 时，你应该确保该连接对每一个要修改的行产生至多一个输出行。换 句话说，一个目标行不应该连接到来自其他表的多于一行上。如果发 生这种情况，则只有一个连接行将被用于更新目标行，但是将使用哪 一行是很难预测的。

由于这种不确定性，只在一个子选择中引用其他表更安全，不过这种 语句通常很难写并且也比使用连接慢。

在分区表的情况下，更新一行有可能导致它不再满足其所在分区的分区约束。此时，如果这个行满足分区树中某个其他分区的分区约束，那么这个行会被移动到那个分区。如果没有这样的分区，则会发生错误。在后台，行的移动实际上是一次DELETE操作和一次INSERT操作。不过，有可能同一行上的并发UPDATE或DELETE会错过这一行。详情请见第 5.10.2.3 节。当前，行不能从外部表分区移动到其他分区，但是如果外部数据包装器支持，它们可以被移入到外部表。

示例

把表films的列kind 中的单词Drama改成Dramatic：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表weather的一行中调整温度项并且 把沉淀物重置为它的默认值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

执行相同的操作并且返回更新后的项：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

使用另一种列列表语法来做同样的更新：

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

为管理Acme Corporation账户的销售人员增加销售量，使用 FROM子句语法：

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

执行相同的操作，在 WHERE子句中使用子选择：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

更新 accounts 表中的联系人姓名以匹配当前被分配的销售员：

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

可以用连接完成类似的结果：

```
UPDATE accounts SET contact_first_name = first_name,
```

```
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

不过，如果salesmen.id不是一个唯一键，　　第二个查询可能会给出令人意外的结果，然而如果有多个id匹配，　　第一个查询保证会发生错误。还有，如果对于一个特定的accounts.sales_id项没有匹配，第一个查询将 把相应的姓名域设置为 NULL，而第二个查询完全不会更新该行。

更新一个统计表中的统计数据以匹配当前数据：

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

尝试插入一个新库存项及其库存量。如果该项已经存在，则转而更新　已有项的库存量。要这样做并且不让整个事务失败，可以使用保存点：

```
BEGIN;
-- 其他操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假定上述语句由于未被唯一键失败，
-- 那么现在我们发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 继续其他操作，并且最终
COMMIT;
```

更改表films中由游标c_films 定位的行的kind列：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

兼容性

这个命令符合SQL标准，不过　FROM和RETURNING子句是　PostgreSQL扩展，把　WITH用于UPDATE也是扩展。

有些其他数据库系统提供了一个FROM选项，在其中在其中目标表　　可以在FROM中被再次列出。但 PostgreSQL不是这样解释 FROM的。在移植使用这种扩展的应用时要小心。

根据标准，一个目标列名的圆括号子列表的来源值可以是任意得到正确列数的行值　　表达式。PostgreSQL只允许来源值是一个 行构造器或者一个子-SELECT。一个列的 被更新值可以在行构造器的情况中被指定为DEFAULT，但在 子-SELECT的情况中不能这样做。

VACUUM

VACUUM — 垃圾收集并根据需要分析一个数据库

大纲

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]
```

其中option可以是下列之一：

```
    FULL
    FREEZE
    VERBOSE
    ANALYZE
    DISABLE_PAGE_SKIPPING
```

而table_and_columns是：

```
    table_name [ ( column_name [, ...] ) ]
```

描述

VACUUM收回由死亡元组占用的存储空间。在通常的PostgreSQL操作中，被删除或者被更新废弃的元组并没有在物理上从它们的表中移除，它们将一直存在直到一次VACUUM被执行。因此有必要周期性地做VACUUM，特别是在频繁被更新的表上。

在没有table_and_columns列表的情况下，VACUUM会处理当前用户具有清理权限的当前数据库中的每一个表和物化视图。如果给出一个列表，VACUUM可以只处理列表中的那些表。

VACUUM　ANALYZE对每一个选定的表ANALYZE。这是两种命令的一种方便的组合形式，可以用于例行的维护脚本。其处理细节可参考ANALYZE。

简单的　VACUUM（不带FULL）简单地收回空间并使其可以被重用。这种形式的命令可以和表的普通读写操作并行，因为它不会获得一个排他锁。但是，这种形式中额外的空间并没有被还给操作系统（在大多数情况下），它仅仅被保留在同一个表中以备重用。VACUUM　FULL将表的整个内容重写到一个新的磁盘文件中，并且不包含额外的空间，这使得没有被使用的空间被还给操作系统。这种形式的命令更慢并且在其被处理时要求在每个表上保持一个排他锁。

当选项列表被包围在圆括号中时，选项可以被写成任何顺序。如果没有圆括号，选项必须严格按照上面所展示的顺序指定。有圆括号的语法在PostgreSQL　9.0时被加入，无圆括号的语法则被废弃。

参数

FULL

选择“完全”清理，它可以收回更多空间，并且需要更长时间和表上的排他锁。这种方法还需要额外的磁盘空间，因为它会创建该表的一个新拷贝，并且在操作完成之前都不会释放旧的拷贝。通常这种方法只用于需要从表中收回数量庞大的空间时。

FREEZE

选择激进的元组“冻结”。指定FREEZE　　等价于参数vacuum_freeze_min_age和vacuum_freeze_table_age设置为0的　VACUUM。当表被重写时总是会执行激进的冻结，因此指定FULL时这个选项是多余的。

VERBOSE

为每个表打印一份详细的清理活动报告。

ANALYZE

更新优化器用以决定最有效执行一个查询的方法的统计信息。

DISABLE_PAGE_SKIPPING

通常，VACUUM将基于可见性映射跳过页面。已知所有元组都被冻结的页面总是会被跳过，而那些所有元组对所有事务都可见的页面则可能会被跳过（除非执行的是激进的清理）。此外，除非在执行激进的清理时，一些页面也可能会被跳过，这样可避免等待其他页面完成对其使用。这个选项禁用所有的跳过页面的行为，其意图是只在可见性映射内容被怀疑时使用，这种情况只有在硬件或者软件问题导致数据库损坏时才会发生。

table_name

要清理的表或物化视图的名称（可以有模式修饰）。如果指定的表示一个分区表，则它所有的叶子分区也会被清理。

column_name

要分析的指定列的名称。缺省是所有列。如果指定了一个列的列表，则ANALYZE也必须被指定。

输出

如果声明了VERBOSE，VACUUM会发出进度消息来表明当前正在处理哪个表。各种有关这些表的统计信息也会打印出来。

注意

要清理一个表，操作者通常必须是表的拥有者或者超级用户。但是，数据库拥有者被允许清理他们的数据库中除了共享目录之外的所有表（对于共享目录的限制意味着一个真正的数据库范围的VACUUM只能被超级用户执行）。VACUUM将会跳过执行者不具备清理权限的表。

VACUUM不能在一个事务块内被执行。

对具有GIN索引的表，VACUUM（任何形式）也会通过将待处理索引项移动到主要GIN索引结构中的合适位置来完成任何待处理的索引插入。详见第 66.4.1 节

我们建议经常清理活动的生产数据库（至少每晚一次），以保证移除失效的行。在增加或删除了大量行之后， 对受影响的表执行VACUUM ANALYZE命令是一个很好的做法。这样做将把最近的更改更新到系统目录，并且允许 PostgreSQL查询规划器在规划用户查询时做出更好的选择。

日常使用时，不推荐FULL选项，但在特殊情况时它会有用。一个例子是当你删除或者更新了一个表中的绝大部分行时，如果你希望在物理上收缩表以减少磁盘空间占用并且允许更快的表扫描，则该选项是比较合适的。VACUUM FULL通常会比简单VACUUM更多地收缩表。

VACUUM会导致I/O流量的大幅度增加，这可能导致其他活动会话性能变差。因此，有时建议使用基于代价的清理延迟特性。详情请参阅第 19.4.4 节

PostgreSQL包括了一个“autovacuum”工具，它可以自动进行例行的清理维护。关于自动和手动清理的更多信息请见第 24.1 节

例子

清理单一表onek，为优化器分析它并且打印出详细的清理活动报告：

```
VACUUM (VERBOSE, ANALYZE) onek;
```

兼容性

在SQL标准中没有VACUUM语句。

参见

vacuumdb, 第 19.4.4 节第 24.1.6 节

VALUES

VALUES — 计算一个行集合

大纲

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

描述

VALUES计算由值表达式指定的一个行值或者 一组行值。更常见的是把它用来生成一个大型命令内的“常量表”， 但是它也可以被独自使用。

当多于一行被指定时，所有行都必须具有相同数量的元素。结果表的列数据类型 由出现在该列的表达式的显式或者推导类型组合决定，决定的规则与 UNION相同（见第 10.5 节。

在大型的命令中，在语法上允许VALUES出现在 SELECT出现的任何地方。因为语法把它当做一个 SELECT，可以为一个VALUES 命令使用ORDER BY、 LIMIT（或者等效的FETCH FIRST）以及OFFSET子句。

参数

expression

要在结果表（行集合）中指定位置计算并且插入的一个常量或者表达式。 在一个出现于INSERT顶层的 VALUES列表中， expression可以 被DEFAULT替代以表示应该插入目标列的默认值。 当VALUES出现在其他环境中时，不能使用 DEFAULT。

sort_expression

一个指示如何排序结果行的表达式或者整型常量。这个表达式 可以用column1、column2等来 引用该VALUES结果的列。详见 ORDER BY 子句。

operator

一个排序操作符。详见 ORDER BY 子句。

count

要返回的最大行数。详见 LIMIT 子句。

start

开始返回行之前要跳过的行数。详见 LIMIT 子句。

注解

应该避免具有大量行的VALUES列表，否则可能会 碰到内存不足失败或者很差的性能。出现在INSERT 中的VALUES是一种特殊情况（因为想要的列类型 可以从INSERT的目标表得知，并且不需要通过扫描 该VALUES列表来推导），因此它可以处理比其他 环境中更大的列表。

示例

一个纯粹的VALUES命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个具有两列、三行的表。它实际等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

更常用地，VALUES可以被用在一个大型 SQL 命令中。 在INSERT中最常用：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在INSERT的环境中，一个VALUES列表 的项可以是DEFAULT来指示应该使用该列的默认值而不是 指定一个值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES也可以被用在可以写子-SELECT 的地方，例如在一个FROM子句中：

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

注意当VALUES被用在一个FROM子句中时， 需要提供一个AS子句，与SELECT相同。 不需要为所有的列用AS子句指定名称，但是那样做是一种好 习惯（在PostgreSQL中， VALUES的默认列名是column1、 column2等，但在其他数据库系统中可能会不同）。

当在INSERT中使用VALUES时，值都会 被自动地强制为相应目标列的数据类型。当在其他环境中使用时，有必要指定 正确的数据类型。如果项都是带引号的字符串常量，强制第一个就足以为所有 项假设数据类型：

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'),
 ('192.168.1.43'));
```

> **提示**
>
> 对于简单的IN测试，最好使用IN的 list-of-scalars形式 而不是写一个上面那样的VALUES查询。标量列表方法的 书写更少并且常常更加高效。

兼容性

VALUES符合 SQL 标准。 LIMIT和OFFSET是 PostgreSQL扩展，另见 SELECT。

另见

INSERT, SELECT

PostgreSQL 客户端应用

这部份包含PostgreSQL客户端应用和工具的参考信息。不是所有这些命令都是通用工具，某些需要特殊权限。这些应用的共同特征是它们可以被运行在任何主机上，而不管数据库服务器在哪里。

当在命令行上指定用户和数据库名时，它们的大小写会被保留 — 空格或特殊字符的出现可能需要使用引号。表名和其他标识符的大小写不会被保留并且可能需要使用引号。

目录

clusterdb

clusterdb — 聚簇一个PostgreSQL数据库

大纲

```
clusterdb [connection-option...] [ --verbose | -v ] [ --table | -t table ] ... [dbname]

clusterdb [connection-option...] [ --verbose | -v ] --all | -a
```

描述

clusterdb是一个工具，它用来对一个PostgreSQL数据库中的表进行重新聚簇。它会寻找之前已经被聚簇过的表，并且再次在最后使用过的同一个索引上对它们重新聚簇。没有被聚簇过的表将不会被影响。

clusterdb是 SQL 命令CLUSTER的一个包装器。在通过这个工具和其他方法访问服务器来聚簇数据库之间没有实质性的区别。

选项

clusterdb接受下列命令行参数：

-a
--all

聚簇所有数据库。

[-d] dbname
[--dbname=]dbname

指定要被聚簇的数据库名称。如果这个参数没有被指定并且-a（或--all）没有被使用，数据库名将从环境变量PGDATABASE中读出。如果该环境变量也没有被设置，指定给该连接的用户名将被用作数据库名。

-e
--echo

回显clusterdb生成并发送给服务器的命令。

-q
--quiet

不显示进度消息。

-t table
--table=table

只聚簇table。可以通过写多个-t开关来聚簇多个表。

-v
--verbose

在处理期间打印详细信息。

-V
--version

打印clusterdb版本并退出。

-?
--help

显示关于clusterdb命令行参数的帮助并退出。

clusterdb也接受下列命令行参数用于连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制clusterdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，clusterdb将自动提示要求一个口令。但是，clusterdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

指定要连接到来发现哪些其他数据库应该被聚簇的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在，将使用template1。

环境

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在CLUSTER和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要聚簇数据库test：

```
$ clusterdb test
```

要聚簇在数据库xyzzy中的一个表foo：

```
$ clusterdb --table foo xyzzy
```

参见

CLUSTER

createdb

createdb — 创建一个新的PostgreSQL数据库

大纲

```
createdb [connection-option...] [option...] [dbname [description]]
```

描述

createdb创建一个新的PostgreSQL数据库。

通常，执行这个命令的数据库用户将成为新数据库的所有者。但是，如果执行用户具有合适的权限，可以通过-O选项指定一个不同的所有者。

createdb是SQL命令CREATE DATABASE的一个包装器。在通过这个工具和其他方法访问服务器来创建数据库之间没有实质性的区别。

选项

createdb接受下列命令行参数：

dbname

指定要被创建的数据库名。该名称必须在这个集簇中所有PostgreSQL数据库中唯一。默认是创建一个与当前系统用户同名的数据库。

description

指定与新创建的数据库相关联的一段注释。

-D tablespace
--tablespace=tablespace

指定该数据库的默认表空间（这个名称被当做一个双引号引用的标识符处理）。

-e
--echo

回显createdb生成并发送到服务器的命令。

-E encoding
--encoding=encoding

指定要在这个数据库中使用的字符编码模式。PostgreSQL服务器支持的字符集在第 23.3.1 节描述。

-l locale
--locale=locale

指定要在这个数据库中使用的区域。这等效于同时指定--lc-collate和--lc-ctype。

--lc-collate=locale

指定要在这个数据库中使用的 LC_COLLATE 设置。

--lc-ctype=locale

指定要在这个数据库中使用的 LC_CTYPE 设置。

-O owner
--owner=owner

指定拥有这个新数据库的数据库用户（这个名称被当做一个双引号引用的标识符处理）。

-T template
--template=template

指定用于创建这个数据库的模板数据库（这个名称被当做一个双引号引用的标识符处理）。

-V
--version

打印createdb版本并退出。

-?
--help

显示关于createdb命令行参数的帮助并退出。

选项-D、-l、-E、 -O和 -T对应于底层 SQL 命令CREATE DATABASE的选项，关于这些选项的信息可见该命令的内容。

createdb也接受下列命令行参数用于连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制createdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，createdb将自动提示要求一个口令。但是，createdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

指定要连接到来发现哪些其他数据库应该被聚簇的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在（或者如果它就是要创建新数据库的名称），将使用template1。

环境

PGDATABASE

如果被设置，就是要创建的数据库名，除非在命令行中覆盖。

PGHOST
PGPORT
PGUSER

默认连接参数。如果没有在命令行或PGDATABASE指定要创建的数据库名，PGUSER也决定要创建的数据库名。

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在CREATE DATABASE和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要使用默认数据库服务器创建数据库demo：

```
$ createdb demo
```

要在主机eden、端口 5000 上使用template0 模板数据库创建数据库demo，这里是命令行命令和底层SQL命令：

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

参见

dropdb, CREATE DATABASE

createuser

createuser — 定义一个新的PostgreSQL用户账户

大纲

```
createuser [connection-option...] [option...] [username]
```

描述

createuser创建一个新的PostgreSQL用户（或者更准确些，是一个角色）。只有超级用户和具有CREATEROLE特权的用户才能创建新用户，因此createuser必须被以上两种用户调用。

如果你希望创建一个新的超级用户，你必须作为一个超级用户连接，而不仅仅是具有CREATEROLE特权。作为一个超级用户意味着绕过数据库中所有访问权限检查的能力，因此超级用户地位不能轻易被授予。

createuser是SQL命令CREATE ROLE的一个包装器。在通过这个工具和其他方法访问服务器来创建用户之间没有实质性的区别。

选项

createuser接受下列命令行参数：

username

指定要被创建的PostgreSQL用户的名称。这个名称必须与这个PostgreSQL安装中所有现存角色不同。

-c number
--connection-limit=number

为该新用户设置一个最大连接数。默认值为不设任何限制。

-d
--createdb

新用户将被允许创建数据库。

-D
--no-createdb

新用户将不被允许创建数据库。这是默认值。

-e
--echo

回显createuser生成并发送给服务器的命令。

-E
--encrypted

此选项已过时，但为了实现向后兼容仍然接受。

-g role
--role=role

指定一个角色，这个角色将立即加入其中成为其成员。 如果要把这个角色加入到多个角色中作为成员， 可以写多个-g开关。

-i
--inherit

新角色将自动继承把它作为成员的角色的特权。这是默认值。

-I
--no-inherit

新角色将不会自动继承把它作为成员的角色的特权。

--interactive

如果在命令行没有指定用户名，提示要求用户名，并且在命令行没有指定选项 -d/-D、-r/-R、 -s/-S时也提示（一直到 PostgreSQL 9.1 这都是默认行为）。

-l
--login

新用户将被允许登入（即，该用户名能被用作初始会话用户标识符）。这是默认值。

-L
--no-login

新用户将不被允许登入（一个没有登录特权的角色仍然可以作为管理数据库权限的方式而存在）。

-P
--pwprompt

如果给定，createuser将发出一个提示要求新用户的口令。如果你没有计划使用口令认证，这就不是必须的。

-r
--createrole

新用户将被允许创建新的角色（即，这个用户将具有CREATEROLE特权）。

-R
--no-createrole

新用户将不被允许创建新角色。这是默认值。

-s
--superuser

新用户将成为一个超级用户。

-S
--no-superuser

新用户将不会成为一个超级用户。这是默认值。

-V
--version

打印createuser版本并退出。

--replication

新用户将具有REPLICATION特权，这在CREATE ROLE的文档中有更完整的描述。

--no-replication

新用户将不具有REPLICATION特权，这在CREATE ROLE的文档中有更完整的描述。

-?
--help

显示有关createuser命令行参数的帮助并退出。

createuser也接受下列命令行参数作为连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接（不是要创建的用户名）。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制createuser在连接到一个数据库之前提示要求一个口令（用来连接到服务器，而不是新用户的口令）。

这个选项不是必不可少的，因为如果服务器要求口令认证，createuser将自动提示要求一个口令。但是，createuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

环境

PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在CREATE ROLE和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要在默认数据库服务器上创建一个用户joe：

```
$ createuser joe
```

要在默认数据库服务器上创建一个用户joe并提示要求一些额外属性：

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

要使用在主机eden、端口 5000 上的服务器创建同一个用户joe，并带有显式指定的属性，看看下面的命令：

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

要创建用户joe为一个超级用户并且立刻分配一个口令：

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER
 CREATEDB CREATEROLE INHERIT LOGIN;
```

在上面的例子中，在录入新口令时新口令并没有真正地被回显，但是为了清晰，我们特意把它列了出来。如你所见，该口令在被发送给客户端之前会被加密。

参见

dropuser, CREATE ROLE

dropdb

dropdb — 移除一个PostgreSQL数据库

大纲

```
dropdb [connection-option...] [option...] dbname
```

描述

dropdb毁掉一个现有的PostgreSQL数据库。执行这个命令的用户必须是一个数据库超级用户或该数据库的拥有者。

dropdb是SQL命令DROP DATABASE的一个包装器。在通过这个工具和其他方法访问服务器来删除数据库之间没有实质性的区别。

选项

dropdb接受下列命令行参数：

dbname

指定要被移除的数据库的名字。

-e
--echo

回显dropdb生成并发送给服务器的命令。

-i
--interactive

在做任何破坏性的工作之前发出一个验证提示。

-V
--version

打印dropdb版本并退出。

--if-exists

如果数据库不存在也不抛出一个错误。在这种情况下会发出一个提醒。

-?
--help

显示有关dropdb命令行参数的帮助并退出。

dropdb也接受下列命令行参数作为连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制dropdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，dropdb将自动提示要求一个口令。但是，dropdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

指定要连接到来发现哪些其他数据库应该被删除的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在，将使用template1。

环境

PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在DROP DATABASE和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要在默认数据库服务器上毁掉数据库demo：

```
$ dropdb demo
```

要使用在主机eden、端口 5000 上的服务器中毁掉数据库demo，并带有验证和回显，看看下面的命令：

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

参见

createdb, DROP DATABASE

dropuser

dropuser — 移除一个PostgreSQL用户账户

大纲

```
dropuser [connection-option...] [option...] [username]
```

描述

dropuser移除一个已有的PostgreSQL用户。只有超级用户以及具有CREATEROLE特权的用户能够移除PostgreSQL用户（要移除一个超级用户，你必须自己是一个超级用户）。

dropuser是SQL命令DROP ROLE的一个包装器。在通过这个工具和其他方法访问服务器来删除用户之间没有实质性的区别。

选项

dropuser接受下列命令行参数：

username

指定要移除的PostgreSQL用户的名字。如果没有在命令行指定并且使用了-i/--interactive选项，你将被提醒要求一个用户名。

-e
--echo

回显dropuser生成并发送给服务器的命令。

-i
--interactive

在实际移除该用户之前提示要求确认，并且在没有在命令行指定用户名提示要求一个用户名。

-V
--version

打印dropuser版本并退出。

--if-exists

如果用户不存在时不要抛出一个错误。在这种情况下将发出一个提示。

-?
--help

显示有关dropuser命令行参数的帮助并退出。

dropuser也接受下列命令行参数作为连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接（不是要移除的用户名）。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制dropuser在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，dropuser将自动提示要求一个口令。但是，dropuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

环境

PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在DROP ROLE和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要从默认数据库服务器移除用户joe：

```
$ dropuser joe
```

要使用在主机eden、端口 5000 上的服务器移除用户joe，并带有验证和回显，可使用下面的命令：

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

参见

createuser, DROP ROLE

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

```
ecpg [option...] file...
```

描述

ecpg是用于 C 程序的嵌入式 SQL 预处理器。它通过将 SQL 调用替换为特殊函数调用把带有嵌入式 SQL 语句的 C 程序转换为普通 C 代码。输出文件可以被任何 C 编译器工具链处理。

ecpg将把命令行中给出的每一个输入文件转换为相应的 C 输出文件。 输入文件更适宜于使用扩展名.pgc。 该扩展名将被替换为.c来决定输出文件名。 输出文件名也可以使用-o选项覆盖。

这个参考页没有描述嵌入式 SQL 语言。关于该主题请参考第 36 章

选项

ecpg接受下列命令行参数：

-c

自动从 SQL 代码生成确定的 C 代码。当前，这对EXEC SQL TYPE起效。

-C mode

设置一个兼容性模式。mode可以是INFORMIX，INFORMIX_SE或ORACLE。

-D symbol

定义一个 C 预处理器符号。

-i

分析系统也包括文件。

-I directory

指定一个额外的包括路径，用来寻找通过EXEC SQL INCLUDE包括的文件。默认值是.（当前目录）、/usr/local/include、在编译时定义的PostgreSQL包括目录（默认：/usr/local/pgsql/include）以及/usr/include。

-o filename

指定ecpg应该将它的所有输出写到给定的filename。

-r option

选择运行时行为。option可以是下列之一：

no_indicator

不使用指示器而使用特殊值来表示空值。历史上曾有数据库使用这种方法。

prepare

在使用所有语句之前准备它们。libecpg 将保持一个预备语句的缓冲并当语句再被执行时重用该语句。如果缓冲满了，libecpg 将释放最少使用的语句。

questionmarks

为兼容性原因允许使用问号作为占位符。在很久以前这被用作默认值。

-t

打开事务的自动提交。在这种模式下，每一个 SQL 命令会被自动提交，除非它位于一个显式事务块中。在默认模式中，命令只有当EXEC SQL COMMIT被发出时才被提交。

-v

打印额外信息，包括版本和“包括”路径。

--version

打印ecpg版本并退出。

-?
--help

显示关于ecpg命令行参数的帮助并退出。

注解

在编译预处理好的 C 代码文件时，编译器需要能够找到PostgreSQL包括目录中的ECPG头文件。因此，在调用编译器时，你可能必须使用-I选项（例如，-I/usr/local/pgsql/include）。

使用带有嵌入式 SQL 的 C 代码的程序必须被链接到libecpg库，例如使用链接器选项-L/usr/local/pgsql/lib -lecpg。

适合于安装的这些目录的值可以使用pg_config找到。

例子

如果你有一个名为prog1.pgc的嵌入式 SQL C 源文件，你可以使用下列命令序列创建一个可执行程序：

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

pg_basebackup — 获得一个PostgreSQL集簇的一个基础备份

大纲

```
pg_basebackup [option...]
```

描述

pg_basebackup被用于获得一个正在运行的PostgreSQL数据库集簇的基础备份。获得这些备份不会影响连接到该数据库的其他客户端，并且可以被用于时间点恢复（见第 25.3 节以及用作一个日志传送或流复制后备服务器的开始点（见第 26.2 节。

pg_basebackup建立数据库集簇文件的一份二进制副本，同时保证系统进入和退出备份模式。备份总是从整个数据库集簇获得，不可能备份单个数据库或数据库对象。关于个体数据库备份，必须使用一个像pg_dump的工具。

备份通过一个常规PostgreSQL连接制作，并且使用复制协议。该连接必须由一个超级用户或者一个具有REPLICATION权限（第 21.2 节的用户建立，并且pg_hba.conf必须显式地允许该复制连接。该服务器还必须被配置，使max_wal_senders设置得足够高以留出至少一个会话用于备份以及一个用于WAL流（如果使用流）。

在同一时间可以有多个pg_basebackup运行，但是从性能的角度来说最好只做一个备份并且复制结果。

pg_basebackup不仅能从主控机也能从后备机创建一个基础备份。要从后备机获得一个备份，设置后备机让它能接受复制连接（也就是，设置max_wal_senders和hot_standby，并且配置基于主机的认证）。你将也需要在主控机上启用full_page_writes。

注意在来自后备机的在线备份中有一些限制：

- 不会在被备份的数据库集簇中创建备份历史文件。
- 如果正在使用-X none，不保证备份所需的所有 WAL 文件在备份结束时被归档。
- 如果在在线备份期间后备机被提升为主控机，备份会失败。
- 备份所需的所有 WAL 记录必须包含足够的全页写，这要求你在主控机上启用full_page_writes并且不使用一个类似pg_compresslog的工具以archive_command从 WAL 文件中移除全页写。

选项

下列命令行选项控制输出的位置和格式。

-D directory
--pgdata=directory

将输出写到哪个目录。如果必要，pg_basebackup将创建该目录及任何父目录。该目录可能已经存在，但是如果该目录已经存在并且非空就是一个错误。

当备份处于 tar 模式中并且目录被指定为-（破折号）时，tar 文件将被写到stdout。

这个选项是必需的。

-F format
--format=format

为输出选择格式。format可以是下列之一：

p
plain

把输出写成平面文件，使用和当前数据目录和表空间相同的布局。当集簇没有额外表空间时，整个数据库将被放在目标目录中。如果集簇包含额外的表空间，主数据目录将被放置在目标目录中，但是所有其他表空间将被放在它们位于服务器上的相同的绝对路径中。

这是默认格式。

t
tar

将输出写成目标目录中的 tar 文件。主数据目录将被写入到一个名为base.tar的文件中，并且其他表空间将被以其 OID 命名。

如果值-（破折号）被指定为目标目录，tar 内容将被写到标准输出，适合于管道输出到其他程序，例如gzip。只有当集簇没有额外表空间并且没有使用WAL流时这才是可能的。

-r rate
--max-rate=rate

从该服务器传输数据的最大传输率。值的单位是千字节每秒。加上一个后缀M表示兆字节每秒。也接受后缀k，但是没有效果。合法的值在 32 千字节每秒到 1024 兆字节每秒之间。

其目标是限制在运行服务器上的pg_basebackup产生的影响。

这个选项总是会影响数据目录的传输。如果收集方法是fetch时，只有 WAL 文件受到影响。

-R
--write-recovery-conf

在输出目录中（或者当使用 tar 格式时再基础归档文件中）写一个最小的recovery.conf来简化设置一个后备服务器。recovery.conf文件将记录连接设置（如果有）以及pg_basebackup所使用的复制槽，这样流复制后面就会使用相同的设置。

-T olddir=newdir
--tablespace-mapping=olddir=newdir

在备份期间将目录olddir中的表空间重定位到newdir中。为使之有效，olddir必须正好匹配表空间所在的路径（但如果备份中没有包含olddir中的表空间也不是错误）。olddir和newdir必须是绝对路径。如果一个路径凑巧包含了一个=符号，可用反斜线对它转义。对于多个表空间可以多次使用这个选项。例子见下文。

如果以这种方法重定位一个表空间，主数据目录中的符号链接会被更新成指向新位置。因此新数据目录已经可以被一个所有表空间位于更新后位置的新服务器实例使用。

--waldir=waldir

指定用于预写式日志目录的位置。waldir必须是绝对路径。只有当备份是平面文件模式时才能指定事务日志目录。

-X method
--wal-method=method

在备份中包括所需的预写式日志文件（WAL文件）。这包括所有在备份期间产生的预写式日志。除非指定了方法none，可以直接在提取出的目录中启动postmaster而无需参考日志归档，所以这样得到的是一种完整的独立备份。

支持下列收集预写式日志的方法：

n
none

不要在备份中包括预写式日志。

f
fetch

在备份末尾收集预写式日志文件。因此，有必要把wal_keep_segments参数设置得足够高，这样在备份末尾之前日志不会被移除。如果在要传输日志时它已经被轮转，备份将失败并且是不可用的。

如果使用tar格式，预写式日志文件将被写入到base.tar文件。

s
stream

在备份被创建时流传送预写式日志。这将开启一个到服务器的第二连接并且在运行备份时并行开始流传输预写式日志。因此，它将使用最多两个由max_wal_senders参数配置的连接。只要客户端能保持接收预写式日志，使用这种模式不需要在主控机上保存额外的预写式日志。

如果使用tar格式，预写式日志文件被写入到一个单独的名为pg_wal.tar的文件（如果服务器的版本超过10，该文件将被命名为pg_wal.tar）。

这个值是默认值。

-z
--gzip

启用对 tar 文件输出的 gzip 压缩，使用默认的压缩级别。只有使用 tar 格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀.gz。

-Z level
--compress=level

启用对 tar 文件输出的 gzip 压缩，并且制定压缩机别（0 到 9，0 是不压缩，9 是最佳压缩）。只有使用 tar 格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀.gz。

下列命令行选项控制备份的生成和程序的运行。

-c fast|spread
--checkpoint=fast|spread

将检查点模式设置为 fast（立刻）或 spread（默认）（见第 25.3.3 节。

-C
--create-slot

这个选项会导致在开始备份前创建一个由--slot选项指定名称的复制槽。如果槽已经存在则会发生错误。

-l label
--label=label

为备份设置标签。如果没有指定，将使用一个默认值“pg_basebackup base backup”。

-n
--no-clean

默认情况下，当pg_basebackup因为一个错误而中止时，它会把它意识到无法完成该工作之前已经创建的目录（例如数据目录和预写式日志目录）都移除。这个选项可以禁止这种清洗，因此可以用于调试。

注意不管哪一种方式都不会清除表空间目录。

-N
--no-sync

默认情况下，pg_basebackup将等待所有文件被安全地写到磁盘上。这个选项导致pg_basebackup不做这种等待就返回，这样会更快一些，但是也意味着后续发生的操作系统崩溃可能会使得这个基础备份损坏。通常这个选项对测试比较有用，在创建生产安装时不应该使用。

-P
--progress

启用进度报告。启用这个选项将在备份期间发表一个大致的进度报告。由于数据库可能在备份期间改变，这仅仅是一种近似并且可能不会刚好在100%结束。特别地，当 WAL 日志被包括在备份中时，总数据量无法预先估计，并且在这种情况中估计的目标尺寸会在它经过不带 WAL 的总估计后增加。

当这个选项被启用时，备份开始时会列举整个数据库的尺寸，并且接着回头开始发送实际的内容。这可能使备份需要多花一点点时间，并且它在发送第一个数据之前花费的时间更长。

-S slotname
--slot=slotname

这个选项仅能与-X stream一起使用。它导致WAL流使用指定的复制槽。如果该基础备份的目的是被用作一台使用复制槽的流复制后备，则它应该使用与recovery.conf中相同的复制槽名称。通过这种方式，可以确保服务器不会移除位于该基础备份结束与流复制开始之间产生的任何所需的WAL数据。

指定的复制槽必须已经存在，除非同时使用了选项-C。

如果这个选项没有被指定并且服务器支持临时复制槽（版本10以后），则会自动使用一个临时复制槽来进行WAL流。

-v
--verbose

启用冗长模式。将在启动和关闭期间输出一些额外步骤，并且如果进度报告也被启用，还会显示当前正在被处理的确切文件名。

--no-slot

如果服务器支持临时复制槽，这个选项防止备份期间创建临时复制槽。

在使用日志流时，如果没有用选项-S指定槽名称，则默认会创建临时复制槽。

这个选项的主要目的是允许在服务器没有空闲复制槽可用时制作基础备份。使用复制槽几乎总是最好的方式，因为它能防止备份期间所需的WAL被删除。

--no-verify-checksums

如果在取基础备份的服务器上启用了校验码验证，则禁用校验码验证。

默认情况下，校验码会被验证并且校验码失败将会导致一种非零的退出状态。不过，基础备份在这种情况下将不会被移除，就好像使用了--no-clean选项一样。

下列命令行选项控制数据库连接参数。

-d connstr
--dbname=connstr

以一个连接字符串的形式指定用于连接到服务器的参数。详见第 34.1.1 节

为了和其他客户端应用一致，该选项被称为--dbname。但是因为pg_basebackup并不连接到集簇中的任何特定数据库，连接字符串中的数据库名将被忽略。

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自PGHOST环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用PGPORT环境变量中的值（如果设置），或者一个编译在程序中的默认值。

-s interval
--status-interval=interval

指定发送回服务器的状态包之间的秒数。这允许我们更容易地监控服务器的进度。一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新会被发送来避免超时导致的断开连接。默认值是 10 秒。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制pg_basebackup在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，pg_basebackup将自动提示要求一个口令。但是，pg_basebackup将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

其他选项也可用：

-V
--version

打印pg_basebackup版本并退出。

-?
--help

显示有关pg_basebackup命令行参数的帮助并退出。

环境

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

注解

在备份的开始时，需要向从中拿去备份的服务器写一个检查点。尤其在没有使用选项--checkpoint=fast时，这可能需要一点时间，在其间pg_basebackup看起来处于闲置状态。

备份将包括数据目录和表空间中的所有文件，包括配置文件以及由第三方放在该目录中的任何额外文件，不过由PostgreSQL管理的特定临时文件除外。但只有常规文件和目录会被拷贝，但用于表空间的符号链接会被保留。指向PostgreSQL已知的特定目录的符号链接被拷贝为空目录。其他符号链接和特殊设备文件会被跳过。准确的细节请参考第 53.4 节

表空间默认将以普通格式备份到与它们在服务器上相同的路径中，除非使用了--tablespace-mapping选项。如果没有这个选项并且表空间正在使用，在同一台服务器上进行普通格式的基础备份将无法工作，因为备份必须要写入到与原始表空间相同的目录位置。

在使用 tar 格式模式时，用户应负责在启动 PostgreSQL 服务器前解压每一个 tar 文件。如果有额外的表空间，用于它们的 tar 文件需要被解压到正确的位置。在这种情况下，服务器将根据包含在base.tar文件中的tablespace_map文件的内容为那些表空间创建符号链接。

pg_basebackup可以和具有相同或较低主版本的服务器一起工作，最低是 9.1。但是，WAL 流模式（-X 流）只能和版本为 9.3 及以上版本的服务器一起工作。当前版本的 tar 格式模式（--format=tar）只能用于版本 9.5 及以上的服务器。

如果在源集簇上启用了组权限，在plain以及tar模式中pg_basebackup将保留组权限。

例子

要创建服务器mydbserver的一个基础备份并将它存储在本地目录/usr/local/pgsql/data中：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

要创建本地服务器的一个备份，为其中每一个表空间产生一个压缩过的 tar 文件，并且将它存储在目录backup中，在运行期间显示一个进度报告：

```
$ pg_basebackup -D backup -Ft -z -P
```

要创建一个单一表空间本地数据库的备份并且使用bzip2压缩它：

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

（如果在该数据库中有多个表空间，这个命令将失败）。

要创建一个本地数据库的备份，其中/opt/ts中的表空间被重定位到./backup/ts：

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

参见

pg_dump

pgbench

pgbench — 在PostgreSQL上运行一个基准测试

大纲

pgbench -i [option...] [dbname]

pgbench [option...] [dbname]

描述

pgbench是一种在PostgreSQL上运行基准测试的简单程序。它可能在并发的数据库会话中一遍一遍地运行相同序列的 SQL 命令，并且计算平均事务率（每秒的事务数）。默认情况下，pgbench会测试一种基于 TPC-B 但是要更宽松的场景，其中在每个事务中涉及五个SELECT、UPDATE以及INSERT命令。但是，通过编写自己的事务脚本文件很容易用来测试其他情况。

pgbench的典型输出像这样：

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

前六行报告一些最重要的参数设置。接下来的行报告完成的事务数以及预期的事务数（后者就是客户端数量与每个客户端事务数的乘积），除非运行在完成之前失败，这些值应该是相等的（在-T模式中，只有实际的事务数会被打印出来）。最后两行报告每秒的事务数，分别代表包括和不包括开始数据库会话所花时间的情况。

默认的类 TPC-B 事务测试要求预先设置好特定的表。可以使用-i（初始化）选项调用pgbench来创建并且填充这些表（当你在测试一个自定义脚本时，你不需要这一步，但是需要按你自己的测试需要做一些设置工作）。初始化类似这样：

```
pgbench -i [ other-options ] dbname
```

其中dbname是要在其中进行测试的预先创建好的数据库的名称（你可能还需要-h、-p或-U选项来指定如何连接到数据库服务器）。

> **小心**
>
> pgbench -i会创建四个表pgbench_accounts、pgbench_branches、pgbench_history以及pgbench_tellers，如果同名表已经存在会被先删除。如果你已经有同名表，一定注意要使用另一个数据库！

在默认的情况下“比例因子”为 1，这些表初始包含的行数为：

```
table                   # of rows
```

```
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

你可以使用-s（比例因子）选项增加行的数量。-F（填充因子）选项也可以在这里使用。

一旦你完成了必要的设置，你就可以用不包括-i的命令运行基准，也就是：

```
pgbench [ options ] dbname
```

在近乎所有的情况中，你将需要一些选项来做一次有用的测试。最重要的选项是-c（客户端数量）、 -t（事务数量）、-T（时间限制）以及-f（指定一个自定义脚本文件）。完整的列表见下文。

选项

下面分成三个部分。数据库初始化期间使用的选项和运行基准时会使用不同的选项，但也有一些选项在两种情况下都使用。

初始化选项

pgbench接受下列命令行初始化参数：

-i
--initialize

要求调用初始化模式。

-I init_steps
--init-steps=init_steps

只执行选出的一组普通初始化步骤。init_steps指定要被执行的初始化步骤，每一个步骤使用一个字符代表。每一个步骤都以指定的顺序被调用。默认是dtgvp。可用的步骤是：

d（删除）

删除任何已有的pgbench表。

t（创建表）

创建标准pgbench场景使用的表，即pgbench_accounts、pgbench_branches、pgbench_history以及pgbench_tellers。

g（生成数据）

生成数据并且装入到标准的表中，替换掉已经存在的任何数据。

v（清理）

在标准的表上调用VACUUM。

p（创建主键）

在标准的表上创建主键索引。

f（创建外键）

在标准的表之间创建外键约束（注意这一步默认不会被执行）。

-F fillfactor
--fillfactor=fillfactor

用给定的填充因子创建表pgbench_accounts、pgbench_tellers以及pgbench_branches。默认是100。

-n
--no-vacuum

在初始化期间不执行清理（这个选项会抑制v初始化步骤，即便在-I中指定了该步骤）。

-q
--quiet

把记录切换到安静模式，只是每 5 秒产生一个进度消息。默认的记录会每 100000 行打印一个消息，这经常会在每秒钟输出很多行（特别是在好的硬件上）。

-s scale_factor
--scale=scale_factor

将生成的行数乘以比例因子。例如，-s 100将在pgbench_accounts表中创建 10,000,000 行。默认为 1。当比例为 20,000 或更高时，用来保存账号标识符的列（aid列）将切换到使用更大的整数（bigint），这样才能足以保存账号标识符。

--foreign-keys

在标准的表之间创建外键约束（如果f在初始化步骤序列中不存在，这个选项会把它加入）。

--index-tablespace=index_tablespace

在指定的表空间而不是默认表空间中创建索引。

--tablespace=tablespace

在指定的表空间而不是默认表空间中创建表。

--unlogged-tables

把所有的表创建为非日志记录表而不是永久表。

基准选项

pgbench接受下列命令行基准参数：

-b scriptname[@weight]
--builtin=scriptname[@weight]

把指定的内建脚本加入到要执行的脚本列表中。@之后是一个可选的整数权重，它允许调节抽取该脚本的可能性。如果没有指定，它会被设置为 1。可用的内建脚本有：tpcb-like、simple-update和select-only。这里也接受内建名称无歧义的前缀缩写。如果用上特殊的名字list，将会显示内建脚本的列表并且立刻退出。

-c clients
--client=clients

模拟的客户端数量，也就是并发数据库会话数量。默认为 1。

-C
--connect

为每一个事务建立一个新连接，而不是只为每个客户端会话建立一个连接。这对于度量连接开销有用。

-d
--debug

打印调试输出。

-D varname=value
--define=varname=value

定义一个由自定义脚本（见下文）使用的变量。允许多个-D选项。

-f filename[@weight]
--file=filename[@weight]

把一个从filename读到的事务脚本加入到被执行的脚本列表中。@后面是一个可选的整数权重，它允许调节抽取该测试的可能性。详见下文。

-j threads
--jobs=threads

pgbench中的工作者线程数量。在多 CPU 机器上使用多于一个线程会有用。客户端会尽可能均匀地分布到可用的线程上。默认为 1。

-l
--log

把与每一个事务相关的信息写到一个日志文件中。详见下文。

-L limit
--latency-limit=limit

对持续超过limit毫秒的事务进行独立的计数和报告， 这些事务被认为是迟到（late）了的事务。

在使用限流措施时（--rate=...），滞后于计划超过 limit毫秒并且因此没有希望满足延迟限制的事务根本 不会被发送给服务器。这些事务被认为是被跳过（skipped） 的事务，它们会被单独计数并且报告。

-M querymode
--protocol=querymode

要用来提交查询到服务器的协议：

- simple：使用简单查询协议。
- extended使用扩展查询协议。
- prepared：使用带预备语句的扩展查询语句。

默认是简单查询协议（详见第 53 章。

-n
--no-vacuum

在运行测试前不进行清理。如果你在运行一个不包括标准的表pgbench_accounts、pgbench_branches、pgbench_history和 pgbench_tellers的自定义测试场景时，这个选项是必需的。

-N
--skip-some-updates

运行内建的简单更新脚本。这是-b simple-update的简写。

-P sec
--progress=sec

每sec秒显示进度报告。该报告包括运行了多长时间、从上次报告以来的 tps 以及从上次报告以来事务延迟的平均值和标准偏差。如果低于限流值（-R），延迟会相对于事务预定的开始时间（而不是实际的事务开始时间）计算，因此其中也包括了平均调度延迟时间。

-r
--report-latencies

在基准结束后，报告平均的每个命令的每语句等待时间（从客户端的角度来说是执行时间）。详见下文。

-R rate
--rate=rate

按照指定的速率执行事务而不是尽可能快地执行（默认行为）。该速率 以 tps（每秒事务数）形式给定。如果目标速率高于最大可能速率，则 该速率限制不会影响结果。

该速率的目标是按照一条泊松分布的调度时间线开始事务。期望的开始 时间表会基于客户端第一次开始的时间（而不是上一个事务结束的时 间）前移。这种方法意味着当事务超过它们的原定结束时间时，更迟的 那些有机会再次追赶上来。

当限流措施被激活时，运行结束时报告的事务延迟是从预订的开始时间计 算而来的，因此它包括每一个事务不得不等待前一个事务结束所花的时 间。该等待时间被称作调度延迟时间，并且它的平均值和最大值也会被 单独报告。关于实际事务开始时间的事务延迟（即在数据库中执行事务 所花的时间）可以用报告的延迟减去调度延迟时间计算得到。

如果把--latency-limit和--rate一起使用， 当一个事务在前一个事务结束时已经超过了延迟限制时，它可能会滞后 非常多，因为延迟是从计划的开始时间计算得来。这类事务不会被发送 给服务器，而是一起被跳过并且被单独计数。

一个高的调度延迟时间表示系统无法用选定的客户端和线程数按照指定 的速率处理事务。当平均的事务执行时间超过每个事务之间的调度间隔 时，每一个后续事务将会落后更多，并且随着测试运行时间越长，调度 延迟时间将持续增加。发生这种情况时，你将不得不降低指定的事务速率。

-s scale_factor
--scale=scale_factor

在pgbench的输出中报告指定的比例因子。对于内建测试，这并非必需；正确的比例因子将通过对pgbench_branches表中的行计数来检测。不过，当只测试自定义基准（-f选项）时，比例因子将被报告为 1（除非使用了这个选项）。

-S
--select-only

执行内建的只有选择的脚本。是-b select-only简写形式。

-t transactions
--transactions=transactions

每个客户端运行的事务数量。默认为 10。

-T seconds
--time=seconds

运行测试这么多秒，而不是为每个客户端运行固定数量的事务。-t和-T是互斥的。

-v
--vacuum-all

在运行测试前清理所有四个标准的表。在没有用-n以及-v时，pgbench将清理pgbench_tellers 和pgbench_branches表，并且截断pgbench_history。

--aggregate-interval=seconds

聚集区间的长度（单位是秒）。仅可以与-l选项一起使用。通过这个选项，日志会包含针对每个区间的概要数据，如下文所述。

--log-prefix=prefix

设置--log创建的日志文件的文件名前缀。默认是pgbench_log。

--progress-timestamp

当显示进度（选项-P）时，使用一个时间戳（Unix 时间）取代从运行开始的秒数。单位是秒，在小数点后是毫秒精度。这可以有助于比较多种工具生成的日志。

--random-seed=SEED

设置随机数生成器种子。为系统的随机数生成器提供种子，然后随机数生成器会产生一个初始生成器状态序列，每一个线程一个状态。SEED的值可以是：time（默认值，种子基于当前时间）、rand（使用一种强随机源，如果没有可用的源则失败）或者一个无符号十进制整数值。一个pgbench脚本中会显式（random...函数）地或者隐式地（如--rate使用随机数生成器调度事务）调用随机数生成器。在被明确设置时，用作种子的值会显示在终端上。 还可以通过环境变量PGBENCH_RANDOM_SEED提供用于SEED的值。为了确保所提供的种子影响所有可能的使用，把这个选项放在第一位或者使用环境变量。

明确地设置种子允许准确地再生一个pgbench运行，对随机数而言。因为随机状态是针对每个线程管理，这意味着如果每一个线程有一个客户端并且没有外部或者数据依赖，则对于一个相同的调用就会有完全相同的pgbench运行。从一种统计的角度来看，再生运行不是什么好主意，因为它能隐藏性能可变性或者不正当地改进性能，即通过命中前一次运行的相同页面来改进性能。不过，它也可以对调试起到很大帮助作用，例如重新运行一种导致错误的棘手用例。请善用。

--sampling-rate=rate

采样率，在写入数据到日志时被用来减少日志产生的数量。如果给出这个选项，只有指定比例的事务被记录。1.0 表示所有事务都将被记录，0.05 表示只有 5% 的事务会被记录。

在处理日志文件时，记得要考虑这个采样率。例如，当计算TPS值时，你需要相应地乘以这个数字（例如，采样率是 0.01，你将只能得到实际TPS的 1/100）。

普通选项

pgbench接受下列命令行普通参数：

-h hostname
--host=hostname

数据库服务器的主机名

-p port
--port=port

数据库服务器的端口号

-U login
--username=login

要作为哪个用户连接

-V
--version

打印pgbench版本并退出。

-?
--help

显示有关pgbench命令行参数的信息，并且退出。

注解

在pgbench中实际执行的“事务”是什么？

pgbench执行从指定列表中随机选中的测试脚本。它们包括带有-b的内建脚本和带有-f的用户提供的自定义脚本。每一个脚本可以在其后用@指定一个相对权重，这样可以更改该脚本的抽取概率。默认权重是1。权重为0的脚本会被忽略。

默认的内建事务脚本（也会被-b tpcb-like调用）会在每个事务上发出七个从aid、tid、bid和balance中随机选择的命令。该场景来自于 TPC-B 基准，但并不是真正的 TPC-B，只是名字像而已。

1. BEGIN;
2. UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
4. UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
7. END;

如果选择simple-update内建脚本（还有-N），第 4 和 5 步不会被包括在事务中。这将避免更新那些表中的内容，但是它会让该测试用例更不像 TPC-B。

如果选择select-only内建脚本（还有-S），只会发出SELECT。

自定义脚本

pgbench支持通过从一个文件中（-f选项）读取事务脚本替换默认的事务脚本（如上文所述）来运行自定义的基准场景。在这种情况中，一个“事务”就是一个脚本文件的一次执行。

脚本文件包含一个或者多个被分号终结的 SQL 命令。空行以及以--开始的行会被忽略。脚本文件也可以包含“元命令”，它会由pgbench自身解释，详见下文。

注意

在PostgreSQL 9.6 之前，脚本文件中的 SQL 命令被新行终结，因此命令不能跨行。现在需要分号来分隔连续的 SQL 命令（如果 SQL 命令后面跟着一个元命令则不需要一个分号）。如果需要创建一个能在新

> 旧版本pgbench下工作的脚本文件，要确保把每个 SQL 命令写在一个由分号终结的行中。

对脚本文件有一种简单的变量替换功能。变量名必须由字母（包括非拉丁字母）、数字以及下划线构成。如上所述，变量可以用命令行的 -D选项设置，或者按下文所说的使用元命令设置。 除了用-D命令行选项预先设置的任何变量之外， 还有一些被自动预先设置的变量，它们被列在 表 241中。一个用-D 为这些变量值指定的值会优先于自动的预设值。 一旦被设置，可以在 SQL 命令中写:variablename 来插入一个变量的值。当运行多于一个客户端会话时， 每一个会话拥有它自己的变量集合。

表 241. 自动变量

变量	简介
client_id	标识客户端会话的唯一编号（从零开始）
default_seed	默认在哈希函数中使用的种子
random_seed	随机数生成器种子（除非用-D重载）
scale	当前的缩放因子

脚本文件元命令以反斜线（\）开始并且通常延伸到行的末尾，不过它们也能够通过写一个反斜线回车继续到额外行。一个元命令和它的参数用空白分隔。支持的元命令是：

```
\if expression
\elif expression
\else
\endif
```

这一组命令实现了可嵌套的条件块，类似于psql的\if expression。条件表达式与\set的相同，非零值会被解释为真。

```
\set varname expression
```

设置变量varname为一个从expression计算出的值。该表达式可以包含NULL常量、布尔常量TRUE和FALSE、5432这样的整数常量、3.14159这样的double常量、对变量的引用:variablename、 操作符（保持它们通常的SQL优先级和结合性）、函数调用、 SQL CASE一般条件表达式以及括号。

函数和大部分操作符在NULL输入上会返回NULL。

对于条件目的，非零数字值是TRUE，数字零值以及NULL是FALSE。

在没有为CASE提供最终的ELSE子句时，默认值是NULL。

示例：

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

```
\sleep number [ us | ms | s ]
```

导致脚本执行休眠指定的时间，时间的单位可以是微妙（us）、毫秒（ms）或者秒（s）。如果单位被忽略，则秒是默认值。number要么是一个整数常量，要么是一个引用了具有整数值的变量的:variablename。

例子：

```
\sleep 10 ms
```

\setshell varname command [argument ...]

用给定的argument设置变量varname为 shell 命令command的结果。该命令必须通过它的标准输出返回一个整数值。

command和每个argument要么是一个文本常量，要么是一个引用了一个变量的:variablename。如果你想要使用以冒号开始的argument，在argument的开头写一个额外的冒号。

例子：

```
\setshell variable_to_be_assigned command
 literal_argument :variable ::literal_starting_with_colon
```

\shell command [argument ...]

与\setshell相同，但是结果被抛弃。

例子：

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

内建操作符

表 242中列举的算数、按位、比较以及逻辑操作符都被编译到了pgbench中并且可以被用于\set中出现的表达式中。

表 242. 按优先级升序排列的pgbench操作符

操作符	简介	示例	结果
OR	逻辑或	5 or 0	TRUE
AND	逻辑与	3 and 0	FALSE
NOT	逻辑非	not false	TRUE
IS [NOT] (NULL|TRUE|FALSE)	值测试	1 is null	FALSE
ISNULL|NOTNULL	空测试	1 notnull	TRUE
=	等于	5 = 4	FALSE
<>	不等于	5 <> 4	TRUE
!=	不等于	5 != 5	FALSE
<	小于	5 < 4	FALSE
<=	小于等于	5 <= 4	FALSE
>	大于	5 > 4	TRUE
>=	大于等于	5 >= 4	TRUE
|	整数按位OR	1 | 2	3
#	整数按位XOR	1 # 3	2
&	整数按位AND	1 & 3	1
~	整数按位NOT	~ 1	-2
<<	整数按位左移	1 << 2	4
>>	整数按位右移	8 >> 2	2

操作符	简介	示例	结果
+	加	5 + 4	9
-	减	3 - 2.0	1.0
*	乘	5 * 4	20
/	除（整数会截断结果）	5 / 3	1
%	取模	3 % 2	1
-	取负	- 2.0	-2.0

内建函数

表 243中列出的函数被编译在pgbench中，并且可能被用在出现于\set的表达式中。

表 243. pgbench 函数

函数	返回类型	简介	例子	结果
abs(a)	和a相同	绝对值	abs(-17)	17
debug(a)	和a相同	把a打印到stderr，并且返回a	debug(5432.1)	5432.1
double(i)	double	造型成double	double(5432)	5432.0
exp(x)	double	指数	exp(1.0)	2.718281828459045
greatest(a [, ...])	如果任何a是double则为double，否则为integer	参数中的最大值	greatest(5, 4, 3, 2)	5
hash(a [, seed])	integer	hash_murmur2()的别名	hash(10, 5432)	-5817877081768721676
hash_fnv1a(a [, seed])	integer	FNV-1a hash[1]	hash_fnv1a(10, 5432)	-7793829335365542153
hash_murmur2(a [, seed])	integer	MurmurHash2 hash[2]	hash_murmur2(10, 5432)	-5817877081768721676
int(x)	integer	造型成int	int(5.4 + 3.8)	9
least(a [, ...])	如果任何a是double则为double，否则为integer	参数中的最小值	least(5, 4, 3, 2.1)	2.1
ln(x)	double	自然对数	ln(2.718281828459045)	1.0
mod(i, j)	integer	取模	mod(54, 32)	22
pi()	double	常量PI的值	pi()	3.14159265358979323846
pow(x, y), power(x, y)	double	求幂	pow(2.0, 10), power(2.0, 10)	1024.0
random(lb, ub)	integer	[lb, ub]中均匀分布的随机整数	random(1, 10)	1和10之间的一个整数
random_exponential(lb, ub, parameter)	integer	[lb, ub]中指数分布的随机整数，见下文	random_exponential(1, 10, 3.0)	1和10之间的一个整数

[1] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
[2] https://en.wikipedia.org/wiki/MurmurHash

函数	返回类型	简介	例子	结果
random_gaussian(lb, ub, parameter)	integer	[lb, ub]中高斯分布的随机整数，见下文	random_gaussian(1, 10, 2.5)	1和10之间的一个整数
random_zipfian(lb, ub, parameter)	integer	[lb, ub]中Zipfian分布的随机整数，见下文	random_zipfian(1, 10, 1.5)	1和10之间的一个整数
sqrt(x)	double	平方根	sqrt(2.0)	1.414213562

random函数使用均匀分布生成值，即所有的值都以相等的概率从指定的范围中抽出。random_exponential、random_gaussian以及random_zipfian函数要求一个额外的 double 参数，它决定分布的精确形状。

- 对于指数分布，parameter通过在parameter处截断一个快速下降的指数分布来控制分布，然后投影到边界之间的整数上。确切地来说，

  ```
  f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1 - exp(-parameter))
  ```

  然后min和max之间（包括两者）的值i会被以概率f(i) - f(i + 1)抽出。

  直观上，parameter越大，接近min的值会被越频繁地访问，并且接近max的值会被越少访问。parameter越接近 0，访问分布会越平坦（更均匀）。该分布的粗近似值是范围中当时被抽取 parameter% 次接近min的最频繁的 1% 值。parameter值必须严格为正。

- 对于高斯分布，区间被映射到一个在左边-parameter和右边+parameter截断的标准正态分布（经典钟型高斯曲线）。区间中间的值更可能被抽到。准确地说，如果PHI(x)是标准正态分布的累计分布函数，均值mu定义为(max + min) / 2.0，有

  ```
  f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /
         (2.0 * PHI(parameter) - 1)
  ```

  则min和max（包括两者）之间的值i被抽出的概率是：f(i + 0.5) - f(i - 0.5)。直观上，parameter越大，靠近区间终端的值会被越频繁地抽出，并且靠近上下界两端的值会被更少抽出。大约 67% 的值会被从中间1.0 / parameter的地方抽出，即均值周围0.5 / parameter的地方。并且 95% 的值会被从中间2.0 / parameter的地方抽出，即均值周围1.0 / parameter的地方。例如，如果parameter是 4.0，67% 的值会被从该区间的中间四分之一（1.0 / 4.0）抽出（即从3.0 / 8.0到5.0 / 8.0）。并且 95% 的值会从该区间的中间一半（2.0 / 4.0）抽出（第二和第三四分位）。为了 Box-Muller 变换的性能，parameter最小为 2.0。

- random_zipfian生成一个近似有界的Zipfian分布。对于(0, 1)中的parameter，近似算法采用"Quickly Generating Billion-Record Synthetic Databases", Jim Gray et al, SIGMOD 1994。对于(1, 1000)中的parameter，会使用一种拒绝方法，该方法基于"Non-Uniform Random Variate Generation", Luc Devroye, p. 550-551, Springer 1986。当参数的值为1.0时，该分布没有被定义。对于参数值接近并且大于1.0且在一个小范围上的情况，这个函数性能很差。

  parameter定义该分布有多么倾斜。parameter越大，绘制越接近间隔开头的值越频繁。parameter约接近于0，访问分区就越平坦（更均匀）。 分布是这样的，假设范围从1开始，绘制k与绘制k+1的概率之比为 ((k+1)/k)**parameter。 例如，random_zipfian(1, ..., 2.5)生成值1大约(2/1)**2.5 = 5.66次，比 2 更频繁，它本身被产生(3/2)*2.5 = 2.76次， 比3更频繁，依此类推。

哈希函数hash、hash_murmur2以及hash_fnv1a接受一个输入值和一个可选的种子参数。在没有提供种子的情况下，会使用:default_seed的值，该变量会被随机地初始化，除非用命令行的-D 选项重载。哈希函数可以被用于分散random_zipfian或random_exponential这样的

随机函数的分布。例如，下列pgbench脚本模拟了社交媒体和博客平台上很常见的真实负载，其中少数账号产生了过量的负载：

```
\set r random_zipfian(0, 100000000, 1.07)
\set k abs(hash(:r)) % 1000000
```

在一些情况中需要几个不同的分布，它们彼此之间不相关并且隐式的随机数参数在此时就能派上用场：

```
\set k1 abs(hash(:r, :default_seed + 123)) % 1000000
\set k2 abs(hash(:r, :default_seed + 321)) % 1000000
```

作为一个例子，内建的类 TPC-B 事务的全部定义是：

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

这个脚本允许该事务的每一次迭代能够引用不同的、被随机选择的行（这个例子也展示了为什么让每一个客户端会话有其自己的变量很重要 — 否则它们不会独立地接触不同的行）。

对每个事务做日志

通过-l选项（但是没有--aggregate-interval选项），pgbench把关于每个事务的信息写入到一个日志文件。该日志文件将被命名为prefix.nnn，其中prefix默认为pgbench_log，而nnn是pgbench进程的PID。前缀可以用--log-prefix选项更改。如果-j选项是2或者更高（有多个工作者线程），那么每一个工作者线程将会有它自己的日志文件。第一个工作者的日志文件的命名将和标准的单工作者情况相同。其他工作者的额外日志文件将被命名为prefix.nnn.mmm，其中mmm是每个工作者的一个序列号，这种序列号从1开始编。

日志的格式是：

```
client_id transaction_no time script_no time_epoch time_us [schedule_lag]
```

其中client_id表示哪个客户端会话运行该事务，transaction_no是那个会话已经运行了多少个事务的计数，time是以微秒计的总共用掉的事务时间，script_no标识了要使用哪个脚本文件（当用-f或者-b指定多个脚本时有用），而time_epoch/time_us是一个 Unix 纪元格式的时间戳以及一个显示事务完成时间的以微秒计的偏移量（适合于创建一个带有分数秒的 ISO 8601 时间戳）。 域schedule_lag是事务的预定开始时间和实际开始时间之间的差别，以微秒计。只有使用--rate选项时它才存在。当--rate和--latency-limit同时被使用时，一个被跳过的事务的time会被报告为skipped。

这里是在单个客户端运行中生成的一个日志文件的片段：

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
```

```
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

另一个例子使用的是--rate=100以及--latency-limit=5（注意额外的 schedule_lag列）：

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

在这个例子中，事务 82 迟到了，因为它的延迟（6.173 ms）超过了 5 ms 限制。接下来的两个事务被跳过，因为它们在开始之前就已经迟到了。

在能够处理大量事务的硬件上运行一次长时间的测试时，日志文件可能变得非常大。--sampling-rate选项能被用来只记录事务的一个随机采样。

聚合的日志记录

通过--aggregate-interval选项，日志文件会使用一种不同的格式：

```
interval_start num_of_transactions latency_sum latency_2_sum min_latency max_latency
 [lag_sum lag_2_sum min_lag max_lag]
```

其中interval_start是区间的开始（作为一个Unix纪元的时间戳）、num_transactions是该区间中的事务数、sum_latency是该区间中事务时延的总量、sum_latency_2是该区间中事务时延的平方和、min_latency是该区间中的最小时延、max_latency是该区间中的最大时延。接下来的字段sum_lag、sum_lag_2、min_lag以及max_lag只有在使用--rate选项时才存在。它们提供每个事务要等待前一个事务完成所花的时间的统计信息，即每个事务的计划启动时间与实际启动时间之间的差值。最后一个字段skipped只有在使用--latency-limit选项时才存在。它对因为启动过完被跳过的事务进行计数。每一个事务被计入在其提交时的区间中。

这里是一些输出示例：

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

注意虽然纯（非聚合）日志文件显示为每个事务使用了哪个脚本，但聚合日志却不包含索引。因此如果你需要针对每个脚本的数据，你需要自行聚合数据。

每语句延迟

通过-r选项，pgbench收集每一个客户端执行的每一个语句花费的事务时间。然后在基准完成后，它会报告这些值的平均值，作为每个语句的延迟。

对于默认脚本，输出看起来会像这样：

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
```

```
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
        0.002  \set aid random(1, 100000 * :scale)
        0.005  \set bid random(1, 1 * :scale)
        0.002  \set tid random(1, 10 * :scale)
        0.001  \set delta random(-5000, 5000)
        0.326  BEGIN;
        0.603  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE
 aid = :aid;
        0.454  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        5.528  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid
 = :tid;
        7.335  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE
 bid = :bid;
        0.371  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
        1.212  END;
```

如果指定了多个脚本文件，会为每一个脚本文件单独报告平均值。

注意为每个语句的延迟计算收集额外的时间信息会增加一些负荷。这将拖慢平均执行速度并且降低计算出的 TPS。降低的总量会很显著地依赖于平台和硬件。对比使用和不适用延迟报告时的平均 TPS 值是评估时间开销是否明显的好方法。

良好的做法

很容易使用pgbench产生完全没有意义的数字。这里有一些指导可以帮你得到有用的结果。

排在第一位的是，永远不要相信任何只运行了几秒的测试。使用-t或-T选项让运行持续至少几分钟，这样可以用平均值去掉噪声。在一些情况中，你可能需要数小时来得到能重现的数字。多运行几次测试是一个好主意，这样可以看看你的数字是不是可以重现。

对于默认的类 TPC-B 测试场景，初始化的比例因子（-s）应该至少和你想要测试的最大客户端数量一样大（-c），否则你将主要在度量更新争夺。在pgbench_branches表中只有-s行，并且每个事务都想更新其中之一，因此-c值超过-s将毫无疑问地导致大量事务被阻塞来等待其他事务。

默认的测试场景也对表被初始化了多久非常敏感：表中死亡行和死亡空间的累积会改变结果。要理解结果，你必须跟踪更新的总数以及何时发生清理。如果开启了自动清理，它可能会在度量的性能上产生不可预估的改变。

pgbench的一个限制是在尝试测试大量客户端会话时，它自身可能成为瓶颈。这可以通过在数据库服务器之外的一台机器上运行pgbench来缓解，不过必须是具有低网络延迟的机器。甚至可以在多个客户端机器上针对同一个数据库服务器并发地运行多个pgbench实例。

安全性

如果不可信用户能够访问没有采用安全方案使用模式的数据库，不要在那个数据库中运行pgbench。pgbench使用非限定名称并且不会操纵搜索路径。

pg_config

pg_config — 获取已安装的PostgreSQL的信息

大纲

```
pg_config [option...]
```

描述

pg_config工具打印当前安装版本的PostgreSQL的配置参数。它的设计目的之一是便于想与PostgreSQL交互的软件包能够找到所需的头文件和库。

选项

要使用pg_config，提供一个或多个下列选项：

--bindir

打印用户可执行文件的位置。例如使用这个选项来寻找psql程序。这通常也是pg_config程序所在的位置。

--docdir

打印文档文件的位置。

--htmldir

打印 HTML 文档文件的位置。

--includedir

打印客户端接口的 C 头文件的位置。

--pkgincludedir

打印其它 C 头文件的位置。

--includedir-server

打印用于服务器编程的 C 头文件的位置。

--libdir

打印对象代码库的位置。

--pkglibdir

打印动态可载入模块的位置，或者服务器可能搜索它们的位置（其它架构独立数据文件可能也被安装在这个目录）。

--localedir

打印区域支持文件的位置（如果在PostgreSQL被编译时没有配置区域支持，这将是一个空字符串）。

--mandir

打印手册页的位置。

--sharedir

打印架构独立支持文件的位置。

--sysconfdir

打印系统范围配置文件的位置。

--pgxs

打印扩展 makefile 的位置。

--configure

打印当PostgreSQL被配置编译时给予configure脚本的选项。这可以被用来重新得到相同的配置，或者找出是哪个选项编译了一个二进制包（不过注意二进制包通常包含厂商相关的自定补丁）。参见下面的例子。

--cc

打印用来编译PostgreSQL的CC变量值。这显示被使用的 C 编译器。

--cppflags

打印用来编译PostgreSQL的CPPFLAGS变量值。这显示在预处理时需要的 C 编译器开关（典型的是-I开关）。

--cflags

打印用来编译PostgreSQL的CFLAGS变量值。这显示被使用的 C 编译器开关。

--cflags_sl

打印用来编译PostgreSQL的CFLAGS_SL变量值。这显示被用来编译共享库的额外 C 编译器开关。

--ldflags

打印用来编译PostgreSQL的LDFLAGS变量值。这显示链接器开关。

--ldflags_ex

打印用来编译PostgreSQL的LDFLAGS_EX变量值。这只显示被用来编译可执行程序的链接器开关。

--ldflags_sl

打印用来编译PostgreSQL的LDFLAGS_SL变量值。这只显示被用来编译共享库的链接器开关。

--libs

打印用来编译PostgreSQL的LIBS变量值。这通常包含用于链接到PostgreSQL中的外部库的-l开关。

--version

打印PostgreSQL的版本。

-?
--help

显示有关pg_config命令行参数的帮助并退出。

如果给定多于一个选项，将按照相同的顺序打印信息，每行一项。如果没有给定选项，将打印所有可用信息，并带有标签。

注解

选项--docdir、--pkgincludedir、 --localedir、--mandir、 --sharedir、--sysconfdir、 --cc、--cppflags、 --cflags、--cflags_sl、 --ldflags、--ldflags_sl和--libs在PostgreSQL 8.1 被加入。选项--htmldir在PostgreSQL 8.4 被加入。选项--ldflags_ex在PostgreSQL 9.0 被加入。

例子

要重建当前 PostgreSQL 安装的编译配置，可运行下列命令：

```
eval ./configure `pg_config --configure`
```

pg_config --configure的输出包含 shell 引号，这样带空格的参数可以被正确地表示。因此，为了得到正确的结果需要使用eval。

pg_dump

pg_dump — 把PostgreSQL数据库抽取为一个脚本文件或其他归档文件

大纲

pg_dump [connection-option...] [option...] [dbname]

描述

pg_dump是用于备份一种PostgreSQL数据库的工具。即使数据库正在被并发使用，它也能创建一致的备份。pg_dump不阻塞其他用户访问数据库（读取或写入）。

pg_dump只转储单个数据库。要备份一个集簇或者集簇中　　对于所有数据库公共的全局对象（例如角色和表空间），应使用 pg_dumpall。

转储可以被输出到脚本或归档文件格式。脚本转储是包含 SQL 命令的纯文本文件，它们可以用来重构数据库到它被转储时的状态。要从这样一个脚本恢复，将它喂给psql。脚本文件甚至可以被用来在其他机器和其他架构上重构数据库。在经过一些修改后，甚至可以在其他 SQL 数据库产品上重构数据库。

另一种可选的归档文件格式必须与pg_restore配合使用来重建数据库。它们允许pg_restore能选择恢复什么，或者甚至在恢复之前对条目重排序。归档文件格式被设计为在架构之间可移植。

当使用归档文件格式之一并与pg_restore组合时，pg_dump提供了一种灵活的归档和传输机制。pg_dump可以被用来备份整个数据库，然后pg_restore可以被用来检查归档并/或选择数据库的哪些部分要被恢复。最灵活的输出文件格式是“自定义”格式（-Fc）和“目录”格式（-Fd）。它们允许选择和重排序所有已归档项、支持并行恢复并且默认是压缩的。“目录”格式是唯一一种支持并行转储的格式。

当运行pg_dump时，我们应该检查输出中有没有任何警告（打印在标准错误上），特别是考虑到下面列出的限制。

选项

下列命令选项控制输出的内容和格式。

dbname

指定要被转储的数据库名。如果没有指定，将使用环境变量PGDATABASE。如果环境变量也没有设置，则使用指定给该连接的用户名。

-a
--data-only

只转储数据，而不转储模式（数据定义）。表数据、大对象和序列值都会被转储。

这个选项类似于指定--section=data，但是由于历史原因又不完全相同。

-b
--blobs

在转储中包括大对象。这是当--schema、--table或--schema-only被指定时的默认行为。因此，只有在请求转储一个特定方案或者表的情况中，-b开关才对向转储中加入大对象有用。注意blobs是被考虑的数据，因此在使用--data-only时将被包括在内，但在使用--schema-only时则不会包括。

-B
--no-blobs

在转储中排除大对象。

当同时给定-b和-B时，行为是在数据被转储时输出大对象，请参考-b文档。

-c
--clean

在输出创建数据库对象的命令之前输出清除（删除）它们的命令 （除非也指定了--if-exists，如果任何对象不存在于 目的数据库中，恢复可能会产生一些伤害性的错误消息）。

这个选项只对纯文本格式有意义。对于归档格式，你可以在调用pg_restore时指定该选项。

-C
--create

使得在输出的开始是一个创建数据库本身并且重新连接到被创建的数据库的命令（通过这种形式的一个脚本，在运行脚本之前你连接的是目标安装中的哪个数据库都没有关系）。如果也指定了--clean，脚本会在重新连接到目标数据库之前先删除它然后再重建。

通过--create，输出还会包括数据库的注释（如果有）以及与这个数据库相关的任何配置变量设置，也就是任何提到了这个数据库的ALTER DATABASE ... SET ...命令和ALTER ROLE ... IN DATABASE ... SET ...命令。该数据库本身的访问特权也会被转储，除非指定有--no-acl。

这个选项只对纯文本格式有意义。对于归档格式，你可以在你调用pg_restore时指定这个选项。

-E encoding
--encoding=encoding

以指定的字符集编码创建转储。在默认情况下，该转储会以该数据库的编码创建（另一种得到相同结果的方式是将PGCLIENTENCODING环境变量设置成想要的转储编码）。

-f file
--file=file

将输出发送到指定文件。对于基于输出格式的文件这个参数可以被忽略，在那种情况下将使用标准输出。不过对于目录输出格式必须给定这个参数，在目录输出格式中指定的是一个目录而不是一个文件。在这种情况中，该目录会由pg_dump创建并且不需要以前就存在。

-F format
--format=format

选择输出的格式。format可以是下列之一：

p
plain

输出一个纯文本形式的SQL脚本文件（默认值）。

c
custom

输出一个适合于作为pg_restore输入的自定义格式归档。和目录输出格式一起，这是最灵活的输出格式，它允许在恢复时手动选择和排序已归档的项。这种格式在默认情况还会被压缩。

d
directory

输出一个适合作为pg_restore输入的目录格式归档。这将创建一个目录，其中每个被转储的表和大对象都有一个文件，外加一个所谓的目录文件，该文件以一种pg_restore能读取的机器可读格式描述被转储的对象。一个目录格式归档能用标准 Unix 工具操纵，例如一个未压缩归档中的文件可以使用gzip工具压缩。这种格式默认情况下是被压缩的并且也支持并行转储。

t
tar

输出一个适合于输入到pg_restore中的tar-格式归档。tar 格式可以兼容目录格式，抽取一个 tar 格式的归档会产生一个合法的目录格式归档。不过，tar 格式不支持压缩。还有，在使用 tar 格式时，表数据项的相对顺序不能在恢复过程中被更改。

-j njobs
--jobs=njobs

通过同时归档njobs个表来运行并行转储。这个选项缩减了转储的时间，但是它也增加了数据库服务器上的负载。你只能和目录输出格式一起使用这个选项，因为这是唯一一种让多个进程能在同一时间写其数据的输出格式。

pg_dump将打开njobs + 1 个到该数据库的连接，因此确保你的max_connections设置足够高以容纳所有的连接。

在运行一次并行转储时请求数据库对象上的排他锁可能导致转储失败。其原因是，pg_dump主控进程会在工作者进程将要稍后转储的对象上请求共享锁，以便确保在转储运行时不会有人删除它们并让它们出错。如果另一个客户端接着请求一个表上的排他锁，那个锁将不会被授予但是会被排入队列等待主控进程的共享锁被释放。因此，任何其他对该表的访问将不会被授予或者将排在排他锁请求之后。这包括尝试转储该表的工作者进程。如果没有任何防范措施，这可能会是一种经典的死锁情况。要检测这种冲突，pg_dump工作者进程使用NOWAIT选项请求另一个共享锁。 如果该工作者进程没有被授予这个共享锁，其他某人必定已经在同时请求了一个排他锁并且没有办法继续转储，因此pg_dump除了中止转储之外别无选择。

对于一个一致的备份，数据库服务器需要支持同步的快照，在PostgreSQL 9.2的主服务器和10的后备服务器中引入了一种特性。有了这种特性，即便数据库客户端使用不同的连接，也可以保证他们看到相同的数据集。pg_dump -j使用多个数据库连接，它用主控进程连接到数据一次，并且为每一个工作者任务再一次连接数据库。如果没有同步快照特征，在每一个连接中不同的工作者任务将不能被保证看到相同的数据，这可能导致一个不一致的备份。

如果你希望运行一个 9.2 之前服务器的并行转储，你需要确保数据库内容从主控进程连接到数据库一直到最后一个工作者任务连接到数据库之间不会改变。做这些最简单的方法是在开始备份之前停止任何访问数据库的数据修改进程（DDL 以及 DML）。当对一个 9.2 之前的PostgreSQL服务器运行pg_dump -j时，你还需要指定--no-synchronized-snapshots参数。

-n schema
--schema=schema

只转储匹配schema的模式，这会选择模式本身以及它所包含的所有对象。当没有指定这个选项时，目标数据库中所有非系统模式都将被转储。多个模式可以通过书写多个-n开关来选择。另外，schema参数可以被解释为一种根据psql's \d命令所用的相同规则（见模式（Pattern））编写的模式，这样多个模式也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，见实例。

> **注意**
>
> 当-n被指定时，pg_dump不会尝试转储所选模式可能依赖的任何其他数据库对象。因此，无法保证一次指定模式转储的结果能够仅凭其本身被成功地恢复到一个干净的数据库中。

> **注意**
>
> 当-n被指定时，非模式对象（如二进制大对象）不会被转储。你可以使用--blobs开关将二进制大对象加回到该转储中。

-N schema
--exclude-schema=schema

不转储匹配schema模式的任何模式。该模式被根据-n所用的相同规则被解释。-N可以被给定多次来排除匹配几个模式中任意一个的模式。

当-n和-N都被给定时，该行为是只转储匹配至少一个-n开关但是不匹配-N开关的模式。如果只有-N而没有-n，那么匹配-N的模式会被从一个正常转储中排除。

-o
--oids

转储对象标识符（OID）作为每个表数据的一部分。如果你的应用以某种方式引用OID列（例如在一个外键约束中），应使用这个选项。否则，这个选项不应该被使用。

-O
--no-owner

不输出设置对象拥有关系来匹配原始数据库的命令。默认情况下，pg_dump会发出ALTER OWNER或SET SESSION AUTHORIZATION语句来设置被创建的数据库对象的拥有关系。除非该脚本被一个超级用户（或是拥有脚本中所有对象的同一个用户）启动，这些语句都将会失败。要使一个脚本能够被任意用户恢复，但把所有对象的拥有关系都给这个用户，可指定-O。

这个选项只对纯文本格式有意义。对于归档格式，你可以在调用pg_restore时指定该选项。

-R
--no-reconnect

这个选项已经废弃，但是为了向后兼容仍然能被接受。

-s
--schema-only

只转储对象定义（模式），而非数据。

这个选项是--data-only的逆选项。它和指定--section=pre-data --section=post-data相似，但是由于历史原因又不完全相同。

（不要把这个选项和--schema选项混淆，后者在“schema”的使用上有不同的含义）。

要为数据库中表的一个子集排除表数据，见--exclude-table-data。

-S username
--superuser=username

指定要在禁用触发器时使用的超级用户的用户名。只有使用--disable-triggers时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

-t table
--table=table

只转储名字匹配table的表，“table”还可以包括视图、物化视图、序列和外部表。通过写多个-t开关可以选择多个表。另外，table参数可以被解释为一种根据psql's \d命令所用的相同规则（见模式（Pattern））编写的模式，这样多个表也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，见实例。

当-t被使用时，-n和-N开关不会有效果，因为被-t选择的表将被转储而无视那些开关，并且非表对象将不会被转储。

> **注意**
>
> 当-t被指定时，pg_dump不会尝试转储所选表可能依赖的任何其他数据库对象。因此，无法保证一次指定表转储的结果能够仅凭其本身被成功地恢复到一个干净的数据库中。

> **注意**
>
> -t开关的行为不完全向前兼容 8.2 之前的PostgreSQL版本。以前，写-t tab将转储所有命名为tab的表，但现在它仅仅转储在你默认搜索路径中可见的那一个。要得到旧的行为，你可以写成-t '*.tab'。还有，你必须写类似-t sch.tab的东西来选择一个特定模式中的一个表，而不是用老的惯用语-n sch -t tab。

-T table
--exclude-table=table

不转储匹配table模式的任何表。该模式被根据-t所用的相同规则被解释。-T可以被给定多次来排除匹配几个模式中任意一个的模式。

当-t和-T都被给定时，该行为是只转储匹配至少一个-t开关但是不匹配-T开关的表。如果只有-T而没有-t，那么匹配-T的表会被从一个正常转储中排除。

-v
--verbose

指定冗长模式。这将导致pg_dump向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。

-V
--version

pg_dump版本并退出。

-x
--no-privileges
--no-acl

防止转储访问特权（授予/收回命令）。

-Z 0..9
--compress=0..9

指定要使用的压缩级别。零意味着不压缩。对于自定义归档格式，这会指定个体表数据段的压缩，并且默认是进行中等级别的压缩。对于纯文本输出，设置一个非零压缩级别会导致整个输出文件被压缩，就好像它被gzip处理过一样，但是默认是不压缩。tar 归档格式当前完全不支持压缩。

--binary-upgrade

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

--column-inserts
--attribute-inserts

将数据转储为带有显式列名的INSERT命令（INSERT INTO table (column, ...) VALUES ...）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非PostgreSQL数据库中。不过，由于这个选项为每一行都产生一个单独的命令，重载一行时的一个错误只会导致那一行被丢失而不是整个表内容丢失。

--disable-dollar-quoting

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

--disable-triggers

只有在创建一个只转储数据的转储时，这个选项才相关。它指示pg_dump包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。

当前，为--disable-triggers发出的命令必须作为超级用户来执行。因此，你还应当使用-S指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。

这个选项只对纯文本格式有意义。对于归档格式，你可以在调用pg_restore时指定这个选项。

--enable-row-security

只有在转储具有行安全性的表的内容时，这个选项才相关。默认情况下， pg_dump将把row_security设置为 off 来确保从该表中转储 出所有的数据。如果用户不具有足够能绕过行安全性的特权，那么会抛出 一个错误这个参数指示pg_dump将 row_security设置为 on，允许用户只转储该表中 它们能够访问到的部分内容。

注意如果当前你使用了这个选项，你可能还想得到INSERT格式的转储，因为恢复期间的COPY FROM不支持行安全性。

--exclude-table-data=table

不转储匹配table模式的任何表中的数据。该模式根据-t的相同规则被解释。--exclude-table-data可以被给定多次来排除匹配多个模式的表。当你需要一个特定表的定义但不想要其中的数据时，这个选项就有用了。

要排除数据库中所有表的数据，见--schema-only。

--if-exists

时间条件性命令（即增加一个IF EXISTS子句）来清除数据库和其他对象。 只有同时指定了--clean时，这个选项才可用。

--inserts

将数据转储为INSERT命令（而不是COPY）。这将使得恢复非常慢，这主要用于使转储能够被载入到非PostgreSQL数据库中。不过，由于这个选项为每一行都产生一个单独的命令，重载一行时的一个错误只会导致那一行被丢失而不是整个表内容丢失。注意如果你已经重新安排了列序，该恢复可能会一起失败。--column-inserts选项对于列序改变是安全的，但是会更慢。

--load-via-partition-root

在为一个分区表转储数据时，让COPY语句或者INSERT语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。

在从用这个选项制作的归档恢复时，最好不要使用并行，因为pg_restore将不能准确地知道一个给定的归档数据项将把数据装载到哪个分区中。这会导致效率不高，因为在并行任务见会有锁冲突，或者甚至可能由于在所有的相关数据被装载前建立了外键约束而导致重新装载失败。

--lock-wait-timeout=timeout

在转储的开始从不等待共享表锁的获得。而是在指定的timeout内不能锁定一个表时失败。超时时长可以用SET statement_timeout接受的任何格式指定（允许的值根据你从其转出的服务器版本变化，但是从 7.3 以来的所有版本都接受一个整数表示的毫秒数。如果从 7.3 以前的服务器转出，这个选项会被忽略。）。

--no-comments

不转储注释。

--no-publications

不转储publication。

--no-security-labels

不转储安全标签。

--no-subscriptions

不转储订阅。

--no-sync

默认情况下，pg_dump将等待所有文件被安全地写入磁盘。这个选项会让pg_dump不等待直接返回，这样会更快，但是也意味着后续的一次操作系统崩溃会让该转储损坏。通常这个选项对测试有用，但是不应该在从生产安装中转储数据时使用。

--no-synchronized-snapshots

这个选项允许对 9.2 以前的服务器运行pg_dump -j，详见-j参数的文档。

--no-tablespaces

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。

这个选项只对纯文本格式有意义。对于归档格式，你可以在调用pg_restore时指定该选项。

--no-unlogged-table-data

不转储非日志记录表的内容。这个选项对于表定义（模式）是否被转储没有影响，它只会限制转储表数据。当从一个后备服务器转储时，在非日志记录表中的数据总是会被排除。

--quote-all-identifiers

强制引用所有标识符。当从PostgreSQL主版本与pg_dump不同的服务器上转储一个数据库时或者当输出准备载入到一个具有不同主版本的服务器时，推荐使用这个选项。默认情况下，pg_dump只对在其主版本中是被保留词的标识符加上引号。在转储其他版本服务器时，这种默认行为有时会导致兼容性问题，因为那些版本可能具有些许不同的被保留词集合。使用--quote-all-identifiers能阻止这种问题，但代价是转储脚本更难阅读。

--section=sectionname

只转储命名节。节的名称可以是pre-data、data或post-data。这个选项可以被指定多次来选择多个节。默认是转储所有节。

数据节包含真正的表数据、大对象内容和序列值。数据后项包括索引、触发器、规则和除了已验证检查约束之外的约束的定义。数据前项包括所有其他数据定义项。

--serializable-deferrable

为转储使用一个可序列化事务，以保证所使用的快照与后来的数据库状态是一致的。但是这样做是在事务流中等待一个点，在该点上不能存在异常，这样就不会有转储失败或者导致其他事务带着serialization_failure回滚的风险。关于事务隔离和并发控制详见第 13 章

对于一个只为灾难恢复存在的转储，这个选项没什么益处。如果一个转储被用来在原始数据库持续被更新期间载入一份用于报表或其他只读负载的数据库拷贝时，这个选项就有所帮助。如果没有这个选项，转储可能会反映一个与最终提交事务的任何执行序列都不一致的状态。例如，如果使用了批处理技术，一个批处理在转储中可以显示为关闭，而其中的所有项都不出现。

如果 pg_dump 被启动时没有读写事务在活动，则这个选项没有什么不同。如果有读写事务在活动，该转储的启动可能会被延迟一段不确定的时间。一旦开始运行，有没有这个开关的表现是相同的。

--snapshot=snapshotname

在做一个数据库的转储时指定一个同步的快照（详见 表 9.82 ）。

在需要把转储和一个逻辑复制槽（见第 49 章 或者一个并发会话同步时可以用上这个选项。

在并行转储的情况下，将使用这个选项指定的快照名而不是取一个新快照。

--strict-names

要求每一个模式（-n/--schema）和表（-t/--table）限定符匹配要转储的数据库中至少一个模式/表。注意，如果没有找到有这样的模式/表限定符匹配，即便没有--strict-names，pg_dump也将生成一个错误。

这个选项对-N/--exclude-schema、-T/--exclude-table或者--exclude-table-data没有效果。无法匹配任何对象的排除模式不会被当作错误。

--use-set-session-authorization

输出 SQL-标准的SET SESSION AUTHORIZATION命令取代ALTER OWNER命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法

正常恢复。而且，一个使用SET SESSION AUTHORIZATION的转储将一定会要求超级用户特权来正确地恢复，而ALTER OWNER要求更少的特权。

-?
--help

显示有关pg_dump命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

-d dbname
--dbname=dbname

指定要连接到的数据库名。这等效于指定dbname为命令行上的第一个非选项参数。

如果这个参数包含一个=符号或者以一个合法的URI前缀（postgresql://或postgres://）开始，它将被视作一个conninfo字符串。详见第 34.1 节

-h host
--host=host

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从PGHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在PGPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制pg_dump在连接到一个数据库之前提示要求一个口令。

这个选项从来不是必须的，因为如果服务器要求口令认证，pg_dump将自动提示要求一个口令。但是，pg_dump将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入-W来避免额外的连接尝试。

--role=rolename

指定一个用来创建该转储的角色名。这个选项导致pg_dump在连接到数据库后发出一个SET ROLE rolename命令。当已认证用户（由-U指定）缺少pg_dump所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

PGDATABASE
PGHOST
PGOPTIONS
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

pg_dump在内部执行SELECT语句。如果你运行pg_dump时出现问题，确定你能够从正在使用的数据库中选择信息，例如psql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

pg_dump的数据库活动会被统计收集器正常地收集。如果不想这样，你可以通过PGOPTIONS或ALTER USER命令设置参数track_counts为假。

注解

如果你的数据库集簇对于template1数据库有任何本地添加，要注意将pg_dump的输出恢复到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从template0而不是template1复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

当一个只含数据的转储被选中并且使用了选项--disable-triggers时，pg_dump在开始插入数据之前会发出命令禁用用户表上的触发器，并且接着在数据被插入之后发出命令重新启用它们。如果恢复中途被停止，系统目录可能会停留在一种错误状态。

pg_dump产生的转储文件不包含优化器用来做出查询计划决定的统计信息。因此，在从一个转储文件恢复后运行ANALYZE来确保最优性能是明智的，详见第 24.1.3 和第 24.1.6 节。

因为pg_dump被用来传输数据到更新版本的PostgreSQL，pg_dump的输出被认为可以载入到比pg_dump版本更新的PostgreSQL服务器中。pg_dump也能够从比其版本更旧的PostgreSQL服务器中转储（当前支持回退到版本 7.0）。不过，pg_dump无法从比起主版本号更新的PostgreSQL服务器中转储，它甚至将拒绝冒着创建一个非法转储的风险尝试。还有，不保证pg_dump的输出能被载入到一个更旧主版本的服务器 — 即使该转储是从该版本的服务器中被取得也不行。将一个转储文件载入到一个更旧的服务器可能需要手工编辑该转储文件来移除旧服务器无法理解的语法。在跨版本的情况下，推荐使用--quote-all-identifiers选项，因为它可以避免因为不同PostgreSQL版本间的保留词列表变化而发生问题。

在转储逻辑复制订阅时，pg_dump将生成使用connect = false选项的CREATE SUBSCRIPTION命令，这样恢复订阅时不会建立远程连接来创建复制槽或者进行初始的表拷贝。通过这种方式，可以无需到远程服务器的网络访问就能恢复该转储。然后就需要用户以一种合适的方式重新激活订阅。如果涉及到的主机已经改变，连接信息可能也必须被改变。在开启一次新的全表拷贝之前，截断目标表也可能是合适的。

实例

要把一个数据库mydb转储到一个 SQL 脚本文件：

```
$ pg_dump mydb > db.sql
```

要把这样一个脚本重新载入到一个（新创建的）名为newdb的数据库中：

```
$ psql -d newdb -f db.sql
```

要转储一个数据库到一个自定义格式归档文件：

```
$ pg_dump -Fc mydb > db.dump
```

要转储一个数据库到一个目录格式的归档：

```
$ pg_dump -Fd mydb -f dumpdir
```

要用 5 个并行的工作者任务转储一个数据库到一个目录格式的归档：

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

要把一个归档文件重新载入到一个（新创建的）名为newdb的数据库：

```
$ pg_restore -d newdb db.dump
```

把一个归档文件重新装载到同一个数据库（该归档正是从这个数据库中转储得来）中，丢掉那个数据库中的当前内容：

```
$ pg_restore -d postgres --clean --create db.dump
```

要转储一个名为mytab的表：

```
$ pg_dump -t mytab mydb > db.sql
```

要转储detroit模式中名称以emp开始的所有表，排除名为employee_log的表：

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

要转储名称以east或者west开始并且以gsm结束的所有模式，排除名称包含词test的任何模式：

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

同样，用正则表达式记号法来合并开关：

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

要转储除了名称以ts_开头的表之外的所有数据库对象：

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

要在-t和相关开关中指定一个大写形式或混合大小写形式的名称，你需要双引用该名称，否则它会被折叠到小写形式（见模式（Pattern））。但是双引号对于 shell 是特殊的，所以

反过来它们必须被引用。因此，要转储一个有混合大小写名称的表，你需要类似这样的东西：

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

参见

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — 将一个PostgreSQL数据库集簇抽取到一个脚本文件中

大纲

pg_dumpall [connection-option...] [option...]

描述

pg_dumpall工具可以一个集簇中所有的PostgreSQL数据库写出到（“转储”）一个脚本文件。该脚本文件包含可以用作psql的输入SQL命令来恢复数据库。它会对集簇中的每个数据库调用pg_dump来完成该工作。pg_dumpall还转储对所有数据库公用的全局对象（pg_dump不保存这些对象），也就是说数据库角色和表空间都会被转储。　目前这包括适数据库用户和组、表空间以及适合所有数据库的访问权限等属性。

因为pg_dumpall从所有数据库中读取表，所以你很可能需要以一个数据库超级用户的身份连接以便生成完整的转储。同样，你也需要超级用户特权执行保存下来的脚本，这样才能增加角色和组以及创建数据库。

SQL 脚本将被写出到标准输出。使用 [-f|file] 选项或者 shell 操作符可以把它重定向到一个文件。

pg_dumpall需要多次连接到PostgreSQL服务器（每个数据库一次）。如果你使用口令认证，可能每次都会要求口令。这种情况下使用一个~/.pgpass会比较方便。详见第 34.15 节

选项

下列命令行选项用于控制输出的内容和格式。

-a
--data-only

只转储数据，不转储模式（数据定义）。

-c
--clean

包括在重建数据库之前清除（移除）它们的 SQL 命令。角色和表空间的DROP命令也会被加入进来。

-E encoding
--encoding=encoding

用指定的字符集编码创建转储。默认情况下，转储使用数据库的编码创建（另一种得到相同结果的方法是设置PGCLIENTENCODING环境变量为想要的转储编码）。

-f filename
--file=filename

将输出发送到指定的文件中。如果省略，将使用标准输出。

-g
--globals-only

只转储全局对象（角色和表空间），而不转储数据库。

-o
--oids

将对象标识符（OID）转储为数据的一部分。如果你的应用以某种方式引用OID列（例如在外键约束中），请使用这个选项。否则不应该使用这个选项。

-O
--no-owner

不输出用于设置对象所有权以符合原始数据库的命令。默认情况下，pg_dumpall发出ALTER OWNER或SET SESSION AUTHORIZATION语句来设置被创建的模式元素的所有权。除非脚本是由一个超级用户（或者是拥有脚本中所有对象的同一个用户）所运行，这些语句在脚本运行时会失败。要使得一个脚本能被任意用户恢复，但又不想给予该用户所有对象的所有权，可以指定-O。

-r
--roles-only

只转储角色，不转储数据库和表空间。

-s
--schema-only

只转储对象定义（模式），不转储数据。

-S username
--superuser=username

指定要在禁用触发器时使用的超级用户的用户名。只有使用--disable-triggers时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

-t
--tablespaces-only

只转储表空间，不转储数据库和角色。

-v
--verbose

指定细节模式。这将导致pg_dumpall向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。它也会启用pg_dump中的细节输出。

-V
--version

打印pg_dumpall版本并退出。

-x
--no-privileges
--no-acl

防止转储访问特权（授予/收回命令）。

--binary-upgrade

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

--column-inserts
--attribute-inserts

将数据转储为带有显式列名的INSERT命令（INSERT INTO table(column, ...) VALUES ...）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非PostgreSQL数据库中。

--disable-dollar-quoting

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

--disable-triggers

只有在创建一个只转储数据的转储时，这个选项才相关。它指示pg_dumpall包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。

当前，为--disable-triggers发出的命令必须作为超级用户来执行。因此，你还应当使用-S指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。

--if-exists

时间条件性命令（即增加一个IF EXISTS子句）来清除数据库和其他对象。 只有同时指定了--clean时，这个选项才可用。

--inserts

将数据转储为INSERT命令（而不是COPY）。这将使得恢复非常慢，这主要用于使转储能够被载入到非PostgreSQL数据库中。注意如果你已经重新安排了列序，该恢复可能会一起失败。--column-inserts选项对于列序改变是安全的，但是会更慢。

--load-via-partition-root

在为一个分区表转储数据时，让COPY语句或者INSERT语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。

--lock-wait-timeout=timeout

在转储的开始从不等待共享表锁的获得。而是在指定的timeout内不能锁定一个表时失败。超时时长可以用SET statement_timeout接受的任何格式指定（允许的值根据你从其转出的服务器版本变化，但是从 7.3 以来的所有版本都接受一个整数表示的毫秒数。如果从 7.3 以前的服务器转出，这个选项会被忽略。）。

--no-comments

不转储注释。

--no-publications

不转储publication。

--no-role-passwords

不为角色转储口令。在恢复完后，角色的口令将是空口令，并且在设置口令之前口令认证都不会成功。由于指定这个选项时并不需要口令值，角色信息将从目录视图pg_roles而不是pg_authid中读出。因此，如果对pg_authid的访问被某条安全性策略所限制，那么这个选项也会有所帮助。

--no-security-labels

不转储安全标签。

--no-subscriptions

不转储subscription。

--no-sync

默认情况下，pg_dumpall将等待所有文件被安全地写入到磁盘。这个选项会让pg_dumpall不做这种等待而返回，这样会更快，但是意味着后续的操作系统崩溃可能留下被损坏的转储。通常来说，这个选项对测试有用，但不应该在从生产安装中转储数据时使用。

--no-tablespaces

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。

--no-unlogged-table-data

不转储非日志记录表的内容。这个选项对于表定义（模式）是否被转储没有影响，它只会限制转储表数据。

--quote-all-identifiers

强制引用所有标识符。在从一个与pg_dumpall主版本不同的PostgreSQL服务器转储数据库时或者要将输出载入到一个不同主版本的服务器时，推荐使用这个选项。默认情况下，pg_dumpall只会对为其主版本中保留词的标识符加上引号。在与其他版本的具有不同保留词集合的服务器交互时，这有时会导致兼容性问题。使用--quote-all-identifiers可以阻止这类问题，但是代价是转储脚本会更加难读。

--use-set-session-authorization

输出 SQL-标准的SET SESSION AUTHORIZATION命令取代ALTER OWNER命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法正常恢复。

-?
--help

显示有关pg_dumpall命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

-d connstr
--dbname=connstr

以一个连接字符串的形式，指定用来连接到服务器的参数。详见第 34.1.1 节

这个选项被称为--dbname是为了和其他客户端应用一致，但是因为pg_dumpall需要连接多个数据库，连接字符串中的数据库名将被忽略。使用-l选项指定一个数据库，该数据库被用于初始连接，这将转储全局对象并且发现需要转储哪些其他数据库。

-h host
--host=host

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从PGHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

-l dbname
--database=dbname

指定要连接到哪个数据库转储全局对象以及发现要转储哪些其他数据库。如果没有指定，将会使用postgres数据库，如果postgres不存在，就使用 template1。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在PGPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制pg_dumpall在连接到一个数据库之前提示要求一个口令。

这个选项从来不是必须的，因为如果服务器要求口令认证，pg_dumpall将自动提示要求一个口令。但是，pg_dumpall将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入-W来避免额外的连接尝试。

注意对每个要被转储的数据库，口令提示都会再次出现。通常，最好设置一个~/.pgpass文件来减少手工口令输入。

--role=rolename

指定一个用来创建该转储的角色名。这个选项导致pg_dump在连接到数据库后发出一个SET ROLE rolename命令。当已认证用户（由-U指定）缺少pg_dump所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

PGHOST
PGOPTIONS
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

注解

因为pg_dumpall在内部调用pg_dump，所以， 一些诊断消息可以参考pg_dump。

即使当用户的目的是把转储脚本恢复到一个空的集簇中，--clean选项也有用武之地。--clean的使用让该脚本删除并且重建内建的postgres和template1数据库，确保这两个数据库保持与源集簇中相同的属性（例如locale和编码）。如果不用这个选项，这两个数据库将保持它们现有的数据库级属性以及任何已有的内容。

一旦恢复，建议在每个数据库上运行ANALYZE，这样优化器就可以得到有用的统计信息。你也可以运行vacuumdb -a -z来分析所有数据库。

不应该预期转储脚本运行到结束都不出错。特别是由于脚本将为源集簇中已有的每一个角色发出CREATE ROLE语句，对于bootstrap超级用户当然会得到一个“role already exists”错误，除非目标集簇用一个不同的bootstrap超级用户名完成的初始化。这种错误是无害的并且应该被忽略。--clean选项的使用很可能会产生额外的有关于不存在对象的无害错误消息，不过可以通过加上--if-exists减少这类错误消息。

pg_dumpall要求所有需要的表空间目录在进行恢复之前就必须存在；否则，数据库创建就会由于在非默认位置创建数据库而失败。

例子

要转储所有数据库：

```
$ pg_dumpall > db.out
```

要从这个文件重新载入数据库，你可以使用：

```
$ psql -f db.out postgres
```

这里你连接哪一个数据库并不重要，因为由pg_dumpall创建的脚本将包含合适的命令来创建和连接到被保存的数据库。一个例外是，如果指定了--clean，则开始时必须连接到postgres数据库，该脚本将立即尝试删除其他数据库，并且这种动作对于已连接上的这个数据库将会失败。

参见

可能的错误情况请查看pg_dump。

pg_isready

pg_isready — 检查一个PostgreSQL服务器的连接状态

大纲

```
pg_isready [connection-option...] [option...]
```

描述

pg_isready是一个用来检查一个PostgreSQL数据库服务器的连接状态的工具。其退出状态指定了连接检查的结果。

选项

-d dbname
--dbname=dbname

指定要连接的数据库名。

如果这个参数包含一个=记号或者以一个合法的URI前缀（postgresql://或postgres://）开始，它会被当作一个conninfo字符串。详见第 34.1.1 节

-h hostname
--host=hostname

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认值取自PGPORT环境变量。如果环境变量没有设置，则默认值使用编译时指定的端口（通常是5432）。

-q
--quiet

不显示状态消息。当脚本编程时有用。

-t seconds
--timeout=seconds

尝试连接时，在返回服务器不响应之前等待的最大秒数。设置为 0 则禁用。默认值是 3 秒。

-U username
--username=username

作为用户username连接数据库，而不是用默认用户。

-V
--version

打印pg_isready版本并退出。

-?
--help

显示有关pg_isready命令行参数的帮助并退出。

退出状态

如果服务器正常接受连接，pg_isready返回0给 shell；如果服务器拒绝连接（例如处于启动阶段）则返回1；如果连接尝试没有被相应则返回2；如果没有尝试（例如由于非法参数）则返回3。

环境

和大部分其他PostgreSQL工具相似，pg_isready也使用libpq（见第 34.14 节支持的环境变量。

注解

要获得服务器状态，不一定需要提供正确的用户名、口令或数据库名。 不过，如果提供了不正确的值，服务器将会记录一次失败的连接尝试。

例子

标准用法：

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

使用连接参数运行连接到处于启动中的PostgreSQL集簇：

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

使用连接参数运行连接到无响应的PostgreSQL集簇：

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — 以流的方式从一个PostgreSQL服务器得到预写式日志

大纲

```
pg_receivewal [option...]
```

描述

pg_receivewal被用来从一个运行着的PostgreSQL集簇以流的方式得到预写式日志。预写式日志会被使用流复制协议以流的方式传送，并且被写入到文件的一个本地目录。这个目录可以被用作归档位置来做一次使用时间点恢复的恢复（见第 25.3 节。

当预写式日志在服务器上被产生时，pg_receivewal实时以流的方式传输预写式日志，并且不像archive_command那样等待段完成。由于这个原因，在使用pg_receivewal时不必设置archive_timeout。

与 PostgreSQL 后备服务器上的 WAL 接收进程不同， pg_receivewal默认只在一个 WAL 文件被关闭时 才刷入 WAL 数据。要实时刷入 WAL 数据，必须指定选项 --synchronous。

预写式日志在一个常规PostgreSQL连接上被以流式传送，并且使用复制协议。连接必须由一个超级用户或一个具有REPLICATION权限（见第 21.2 节的用户建立，并且pg_hba.conf必须允许复制连接。服务器也必须被配置一个足够高的max_wal_senders来至少留出一个可用会话给流。

如果该连接丢失，或者它一开始就由于一个非致命错误而没有被建立，pg_receivewal将无限期地重试连接并且尽可能重新建立流。为了避免这种行为，使用-n参数。

如果不出现致命错误，pg_receivewal将一直运行直至被SIGINT信号（Control+C）终止。

选项

```
-D directory
--directory=directory
```

要把输出写到哪个目录。

这个参数是必需的。

```
-E lsn
--endpos=lsn
```

当接收到达指定的LSN时，自动停止复制并且以正常退出状态0退出。

如果有一个记录的LSN正好等于lsn，则该记录将会被处理。

```
--if-not-exists
```

当指定--create-slot并且具有指定名称 的槽已经存在时不要抛出错误。

```
-n
--no-loop
```

不要在连接错误上循环。相反，碰到一个错误时立刻退出。

--no-sync

这个选项导致pg_receivewal不强制WAL数据被刷回磁盘。这样会更快，但是也意味着接下来的操作系统崩溃会让WAL段损坏。通常，这个选项对于测试有用，但不应该在对生产部署进行WAL归档时使用。

这个选项与--synchronous不兼容。

-s interval
--status-interval=interval

指定发送回服务器的状态包之间的秒数。这允许我们更容易地监控服务器的进度。 一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新 会被发送来避免超时导致的断开连接。默认值是 10 秒。

-S slotname
--slot=slotname

要求pg_receivewal使用一个已有的复制槽（见 第 26.2.6 节。在使用这个选项时， pg_receivewal将会报告给服务器一个刷写位置，指示每一个 段是何时被同步到磁盘的，这样服务器可以在不需要该段时移除它。

当pg_receivewal的复制客户端在服务器 上被配置为一个同步后备时，那么使用一个复制槽将会向服务器报告刷写 位置，但只在一个 WAL 文件被关闭时报告。因此，该配置将导致主服务 器上的事务等待很长的时间并且无法令人满意地工作。要让这种配置工作正确，还必须制定选项--synchronous（见下文）。

--synchronous

在 WAL 数据被收到后立即刷入到磁盘。还要在刷写后立即向服务器回送 一个状态包（不考虑--status-interval）。

如果pg_receivewal的复制客户端在服务器 上被配置为一个同步后备，应该指定这个选项来确保向服务器发送及时的反馈。

-v
--verbose

启用冗长模式。

-Z level
--compress=level

启用预写式日志上的gzip压缩，并且指定压缩级别（0到9，0是不压缩而9是最大压缩）。所有的文件名后都将被追加后缀.gz。

下列命令行选项控制数据库连接参数。

-d connstr
--dbname=connstr

指定用于连接到服务器的参数为一个连接字符串。详见第 34.1.1 节

为了和其他客户端应用一致，该选项被称为--dbname。但是因为pg_receivewal并不连接到集簇中的任何特定数据库，连接字符串中的数据库名将被忽略。

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自PGHOST环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用PGPORT环境变量中的值（如果设置），或者一个编译在程序中的默认值。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制pg_receivewal在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，pg_receivewal将自动提示要求一个口令。但是，pg_receivewal将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

为了控制物理复制槽，pg_receivewal 可以执行下列两种动作之一：

--create-slot

用--slot中指定的名称创建一个新的物理复制槽， 然后退出。

--drop-slot

删除--slot中指定的复制槽，然后退出。

其他选项也可用：

-V
--version

打印pg_receivewal版本并退出。

-?
--help

显示有关pg_receivewal命令行参数的帮助并退出。

退出状态

在被SIGINT信号终止（没有正常的方式结束它。因此这不是一种错误）时，pg_receivewal将以状态0退出。 对于致命错误或者其他信号，退出状态将不是零。

环境

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

注解

在使用pg_receivewal替代 archive_command作为主要的 WAL 备份方法时， 强烈建议使用复制槽。否则，服务器可能会在预写式日志文件被备份好之前重用 或者移除它们，因为没有任何信息（不管是来自 archive_command或是复制槽）能够指示 WAL 流已 经被归档到什么程度。不过要注意，如果接收者没有持续地取走 WAL 数据， 一个复制槽将会填满服务器的磁盘空间。

如果在源集簇上启用了组权限，pg_receivewal将保留接收到的WAL文件上的组权限。

例子

要从位于mydbserver的服务器流式传送预写式日志并且将它存储在本地目录/usr/local/pgsql/archive：

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

参见

pg_basebackup

pg_recvlogical

pg_recvlogical — 控制 PostgreSQL 逻辑解码流

大纲

pg_recvlogical [option...]

描述

pg_recvlogical控制逻辑解码复制槽以及来自这种复制槽的流数据。

它会创建一个复制模式的连接，因此它受到和pg_receivewal 相同的约束，还有逻辑复制（第 49 章的约束。

pg_recvlogical与逻辑解码SQL接口的peek和get模式没有等效性。它咋接收到数据以及干净地退出时，它会惰性地发送数据的确认。为了检查一个槽上还未消费的待处理数据，可以使用pg_logical_slot_peek_changes。

选项

必须至少要指定下列选项之一来选择一个动作：

--create-slot

为--dbname指定的数据库用--slot 指定的名称创建一个新的逻辑复制槽，使用 --plugin指定的输出插件。

--drop-slot

删除名称由--slot指定的复制槽，然后退出。

--start

从--slot指定的逻辑复制槽开始进行流式传送更改，一直继续 到被一个信号终止。如果服务器端关机或者断开连接导致更改流结束，会进入一个 循环一直重试，通过指定--no-loop可以防止这种情况下进入 循环重试。

流格式由槽创建时指定的输出插件决定。

连接必须是连接到用于创建该槽的同一个数据库上。

--create-slot和--start可以被一起指定。 --drop-slot不能和另一个动作组合在一起。

下面的命令行选项控制输出的位置和格式以及其他复制行为：

-E lsn
--endpos=lsn

在--start模式中，当接收过程到达指定的LSN时会自动地停止复制并且以正常的退出状态0退出。如果不处于--start模式时指定这个选项，则会发生错误。

如果有一个记录的LSN正好等于lsn，则该记录将被输出。

--endpos不会察觉到事务边界并且可能会在一个事务中间截断输出。任何部分输出的事务都将不会被消费，并且在下一次从该槽中读取时将会重放该事务。单个的消息不会被截断。

-f filename
--file=filename

把接收到并且解码好的事务数据写入到一个文件。使用-可以写到stdout。

-F interval_seconds
--fsync-interval=interval_seconds

指定pg_recvlogical发出 fsync()调用确保输出文件被安全地刷到磁盘的频度。

服务器将会偶尔要求客户端执行一次刷写并且把刷写位置报告给服务器。 这个设置可以在此之外更加频繁地执行刷写。

指定间隔为0会完全禁止发出fsync() 调用，但是仍会报告进度给服务器。在这种情况下，发生崩溃会导致数据丢失。

-I lsn
--startpos=lsn

在--start模式中，从给定的 LSN 开始复制。这个参数的 效果请见第 49 章和第 53.4 节的文档。在其他模式中会忽略这个参数。

--if-not-exists

当指定--create-slot并且具有指定名称 的槽已经存在时不要抛出错误。

-n
--no-loop

当服务器连接丢失时，不要在循环中重试，直接退出。

-o name[=value]
--option=name[=value]

如果指定了输出插件，把选项值value 传递给选项name。存在哪些选项以及它们的效果取决于使用的输出插件。

-P plugin
--plugin=plugin

在创建一个槽时使用指定的逻辑解码输出插件。 见第 49 章如果该槽已经存在，这个选项没有效果。

-s interval_seconds
--status-interval=interval_seconds

这个选项和pg_receivewal中的同名选项具有 相同的效果。请参考那里的描述。

-S slot_name
--slot=slot_name

在--start模式中，使用名为slot_name 的已有逻辑复制槽。在--create-slot模式中，使用这个名称 创建该槽。在--drop-slot模式中，删除这个名称指定的槽。

-v
--verbose

开启详细输出模式。

下列命令行选项控制数据库连接参数。

-d database
--dbname=database

要连接的数据库。这个选项的详细含义请见动作的描述。它可以是一个 libpq连接 字符串，详见第 34.1.1 节默认为用户名。

-h hostname-or-ip
--host=hostname-or-ip

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线， 它被用作一个 Unix 域套接字的目录。默认是从 PGHOST环境变量中取得（如果被设置）， 否则将尝试一次 Unix 域套接字连接。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。 默认是放在PGPORT环境变量中（如果被设置）， 否则使用编译在程序中的默认值。

-U user
--username=user

要作为哪个用户连接。默认是用当前操作系统用户名。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有 其他方式提供口令（例如一个.pgpass文件）， 那么连接尝试将失败。这个选项对于批处理任务和脚本有用， 因为在其中没有一个用户来输入口令。

-W
--password

强制pg_dump在连接到一个数据库之前提示要求一个口令。

这个选项不是必须的，因为如果服务器要求口令认证， pg_dump将自动提示要求一个口令。 但是，pg_dump将浪费一次连接尝试 来发现服务器想要一个口令。在某些情况下，值得键入 -W来避免额外的连接尝试。

还有下列附加选项可用：

-V
--version

打印pg_recvlogical的版本并且退出。

-?
--help

显示关于pg_recvlogical命令行参数的帮助，并且退出。

环境

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

注解

如果在源服务器上启用了组权限，pg_recvlogical将会在接收到的WAL文件上保留组权限。

例子

一个例子请见第 49.1 节

另见

pg_receivewal

pg_restore

pg_restore — 从一个由pg_dump创建的归档文件恢复一个PostgreSQL数据库

大纲

pg_restore [connection-option...] [option...] [filename]

描述

pg_restore是一个用来从pg_dump创建的非文本格式归档恢复PostgreSQL数据库的工具。它将发出必要的命令把该数据库重建成它被保存时的状态。这些归档文件还允许pg_restore选择恢复哪些内容或者在恢复前对恢复项重排序。这些归档文件被设计为可以在不同的架构之间迁移。

pg_restore可以在两种模式下操作。如果指定了一个数据库名称，pg_restore会连接那个数据库并且把归档内容直接恢复到该数据库中。否则，会创建一个脚本，其中包含着重建该数据库所必要的 SQL 命令，它会被写入到一个文件或者标准输出。这个脚本输出等效于pg_dump的纯文本输出格式。因此，一些控制输出的选项与pg_dump的选项类似。

显然，pg_restore无法恢复不在归档文件中的信息。例如，如果归档使用“以INSERT命令转储数据”选项创建， pg_restore将无法使用COPY语句装载数据。

选项

pg_restore接受下列命令行参数。

filename

指定要被恢复的归档文件（对于一个目录格式的归档则是目录）的位置。如果没有指定，则使用标准输入。

-a
--data-only

只恢复数据，不恢复模式（数据定义）。如果在归档中存在，表数据、大对象和序列值会被恢复。

这个选项类似于指定--section=data，但是由于历史原因两者不完全相同。

-c
--clean

在重新创建数据库对象之前清除（丢弃）它们（除非使用了--if-exists，如果有对象在目标数据库中不存在，这可能会生成一些无害的错误消息）。

-C
--create

在恢复一个数据库之前县创建它。如果还指定了--clean，在连接到目标数据库之前丢弃并且重建它。

如果使用--create，pg_restore还会恢复数据库的注释（如果有）以及与其相关的配置变量设置，也就是任何提到过这个数据库的ALTER DATABASE ... SET ...和ALTER ROLE ... IN DATABASE ... SET ...命令。不管是否指定--no-acl，数据库本身的访问特权都会被恢复。

在使用这个选项时，-d提到的数据库只被用于发出初始的DROP DATABASE和CREATE DATABASE命令。所有要恢复到该数据库名中的数据都出现在归档中。

-d dbname
--dbname=dbname

连接到数据库dbname并且直接恢复到该数据库中。

-e
--exit-on-error

在发送 SQL 命令到该数据库期间如果碰到一个错误就退出。默认行为是继续并且在恢复结束时显示一个错误计数。

-f filename
--file=filename

为生成的脚本或列表（当使用-l时）指定输出文件。默认是标准输出。

-F format
--format=format

指定归档的格式。并不一定要指定该格式，因为pg_restore将会自动决定格式。如果指定，可以是下列之一：

c
custom

归档是pg_dump的自定义格式。

d
directory

归档是一个目录归档。

t
tar

归档是一个tar归档。

-I index
--index=index

只恢复提及的索引的定义。可以通过写多个-I开关指定多个索引。

-j number-of-jobs
--jobs=number-of-jobs

使用并发任务运行pg_restore中最耗时的部分 — 载入数据、创建索引或者创建约束。对于一个运行在多处理器机器上的服务器，这个选项能够大幅度减少恢复一个大型数据库的时间。

每一个任务是一个进程或者一个线程，这取决于操作系统，它们都使用一个独立的服务器连接。

这个选项的最佳值取决于服务器、客户端以及网络的硬件设置。因素包括 CPU 的核心数和磁盘设置。一个好的建议是服务器上 CPU 的核心数，但是更大的值在很多情况下也能导致更快的恢复时间。当然，过高的值会由于超负荷反而导致性能降低。

这个选项只支持自定义和目录归档格式。输入必须是一个常规文件或目录（例如，不能是一个管道）。当发出一个脚本而不是直接连接到数据库服务器时会忽略这个选项。还有，多任务不能和选项--single-transaction一起用。

-l
--list

列出归档的内容的表格。这个操作的输出能被用作-L选项的输入。注意如果把-n或-t这样的过滤开关与-l一起使用，它们将会限制列出的项。

-L list-file
--use-list=list-file

只恢复在list-file中列出的归档元素，并且按照它们出现在该文件中的顺序进行恢复。注意如果把-n或-t这样的过滤开关与-L一起使用，它们将会进一步限制要恢复的项。

list-file通常是编辑一个-l操作的输出来创建。行可以被移动或者移除，并且也可以通过在行首放一个（;）将其注释掉。例子见下文。

-n shcema
--schema=schema

只恢复在被提及的模式中的对象。可以用多个-n开关来指定多个模式。这可以与-t选项组合在一起只恢复一个指定的表。

-N schema
--exclude-schema=schema

不恢复所提及方案中的对象。可以用多个-N开关指定多个要被排除的方案。

如果对同一个方案名称同时给出了-n和-N，则-N会胜出并且该方案会被排除。

-O
--no-owner

不要输出将对象的所有权设置为与原始数据库匹配的命令。默认情况下，pg_restore会发出ALTER OWNER或者SET SESSION AUTHORIZATION语句来设置已创建的模式对象的所有权。除非到该数据库的初始连接是一个超级用户（或者拥有脚本中所有对象的同一个用户）建立的，这些语句将会失败。通过-O，任何用户名都可以被用于初始连接，并且这个用户将会拥有所有被创建的对象。

-P function-name(argtype [, ...])
--function=function-name(argtype [, ...])

只恢复被提及的函数。要小心地拼写函数的名称和参数使它们正好就是出现在转储文件的内容表中的名称和参数。可以使用多个-P开关指定多个函数。

-R
--no-reconnect

这个选项已被废弃，但是出于向后兼容性的目的，系统仍然还接受它。

-s
--schema-only

只恢复归档中的模式（数据定义）不恢复数据。

这个选项是--data-only的逆选项。它与指定--section=pre-data --section=post-data相似，但是由于历史原因并不完全相同。

（不要把这个选项和--schema选项弄混，后者把词“schema”用于一种不同的含义）。

-S username
--superuser=username

指定在禁用触发器时要用的超级用户名。只有使用--disable-triggers时这个选项才相关。

-t table
--table=table

只恢复所提及的表的定义和数据。出于这个目的，“table”包括视图、物化视图、序列和外部表。可以写上多个-t开关可以选择多个表。这个选项可以和-n选项结合在一起指定一个特定模式中的表。

> **注意**
>
> 在指定-t时，pg_restore不会尝试恢复所选表可能依赖的任何其他数据库对象。因此，无法确保能成功地把一个特定表恢复到一个干净的数据库中。

> **注意**
>
> 这个标志的行为和pg_dump的-t标志不一样。在pg_restore中当前没有任何通配符匹配的规定，也不能在其-t选项中包括模式的名称。而且，虽然pg_dump的-t标志也会转储选中表的附属对象（例如索引），但是pg_restore的-t标志不包括这些附属对象。

> **注意**
>
> 在 9.6 版本之前的PostgreSQL 9.6 中，这个标志只匹配表，而并不匹配其他类型的关系。

-T trigger
--trigger=trigger

只恢复所提及的触发器。可以用多个-T开关指定多个触发器。

-v
--verbose

指定冗长模式。

-V
--version

打印该pg_restore的版本并退出。

-x
--no-privileges
--no-acl

阻止恢复访问特权（授予/收回命令）。

-1
--single-transaction

将恢复作为单一事务执行（即把发出的命令包裹在BEGIN/COMMIT中）。这可以确保要么所有命令完全成功，要么任何改变都不被应用。这个选项隐含了--exit-on-error。

--disable-triggers

只有在执行一个只恢复数据的恢复时，这个选项才相关。它指示pg_restore在装载数据时执行命令临时禁用目标表上的触发器。如果你在表上有参照完整性检查或者其他触发器并且你不希望在数据载入期间调用它们时，请使用这个选项。

目前，为--disable-triggers发出的命令必须以超级用户身份完成。因此你还应该用-S指定一个超级用户名，或者更好的方法是以一个PostgreSQL超级用户运行pg_restore。

--enable-row-security

只有在恢复具有行安全性的表的内容时，这个选项才相关。默认情况下，pg_restore将把row_security设置为 off 来确保所有数据都被恢复到表中。如果用户不拥有足够绕过行安全性的特权，那么会抛出一个错误。这个参数指示pg_restore把row_security设置为on允许用户尝试恢复启用了行安全性的表的内容。如果用户没有从转储向表中插入行的权限，这仍将失败。

注意当前这个选项还要求转储处于INSERT格式，因为COPY FROM不支持行安全性。

--if-exists

使用条件命令（即增加一个IF EXISTS子句）删除数据库对象。只有指定了--clean时，这个选项才有效。

--no-comments

即便归档中包含注释也不输出恢复注释的命令。

--no-data-for-failed-tables

默认情况下，即便表的创建命令失败（例如因为表已经存在），表数据也会被恢复。通过这个选项，对这类表的数据会被跳过。如果目标数据库已经包含了想要的表内容，这种行为又很有有用。例如，PostgreSQL扩展（如PostGIS）的辅助表可能已经被载入到目标数据库中，指定这个选项就能阻止把重复的或者废弃的数据载入到这些表中。

只有当直接恢复到一个数据库中时这个选项才有效，在产生 SQL脚本输出时这个选项不会产生效果。

--no-publications

即便归档中包含publication也不输出恢复publication的命令。

--no-security-labels

不要输出恢复安全标签的命令，即使归档中包含安全标签。

--no-subscriptions

即便归档中包含subscription也不输出恢复subscription的命令。

--no-tablespaces

不输出命令选择表空间。通过这个选项，所有的对象都会被创建在恢复时的默认表空间中。

--section=sectionname

只恢复提及的小节。小节的名称可以是pre-data、data或者post-data。可以把这个选项指定多次来选择多个小节。默认值是恢复所有小节。

数据小节包含实际的表数据以及大对象定义。Post-data 项由索引定义、触发器、规则和除已验证的检查约束之外的约束构成。Pre-data 项由所有其他数据定义项构成。

--strict-names

要求每一个模式（-n/--schema）以及表（-t/--table）限定词匹配备份文件中至少一个模式/表。

--use-set-session-authorization

输出 SQL 标准的SET SESSION AUTHORIZATION命令取代ALTER OWNER命令来决定对象拥有权。这会让转储更加兼容标准，但是依赖于转储中对象的历史，可能无法正确恢复。

-?
--help

显示有关pg_restore命令行参数的帮助，并且退出。

pg_restore也接受下列用于连接参数的命令行参数：

-h host
--host=host

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从PGHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在PGPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制pg_restore在连接到一个数据库之前提示要求一个口令。

这个选项不是必须的，因为如果服务器要求口令认证，pg_restore将自动提示要求一个口令。但是，pg_restore将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入-W来避免额外的连接尝试。

--role=rolename

指定一个用来创建该转储的角色名。这个选项导致pg_restore在连接到数据库后发出一个SET ROLE rolename命令。当已认证用户（由-U指定）缺少pg_restore所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

PGHOST
PGOPTIONS
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

当使用-d选项指定一个直接数据库连接时，pg_restore在内部执行SELECT语句。如果你运行pg_restore时出现问题，确定你能够从正在使用的数据库中选择信息，例如psql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

注解

如果你的数据库集簇对于template1数据库有任何本地添加，要注意将pg_restore的输出载入到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从template0而不是template1复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

下面将详细介绍pg_restore的局限性。

- 在恢复数据到一个已经存在的表中并且使用了选项--disable-triggers时，pg_restore会在插入数据之前发出命令禁用用户表上的触发器，然后在完成数据插入后重新启用它们。如果恢复在中途停止，可能会让系统目录处于错误的状态。
- pg_restore不能有选择地恢复大对象，例如只恢复特定表的大对象。如果一个归档包含大对象，那么所有的大对象都会被恢复，如果通过-L、-t或者其他选项进行了排除，它们一个也不会被恢复。

pg_dump的局限性详见pg_dump文档。

一旦完成恢复，应该在每一个被恢复的表上运行ANALYZE，这样优化器能得到有用的统计信息。更多信息请见第 24.1.3 和第 24.1.6 节

示例

假设我们已经以自定义格式转储了一个叫做mydb的数据库：

```
$ pg_dump -Fc mydb > db.dump
```

要删除该数据库并且从转储中重新创建它：

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

-d开关中提到的数据库可以是任何已经存在于集簇中的数据库，pg_restore只会用它来为mydb发出CREATE DATABASE命令。通过-C，数据总是会被恢复到出现在归档文件的数据库名中。

要把转储重新载入到一个名为newdb的新数据库中：

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

注意我们不使用-C，而是直接连接到要恢复到其中的数据库。还要注意我们是从template0而不是template1创建了该数据库，以保证它最初是空的。

要对数据库项重排序，首先需要转储归档的表内容：

```
$ pg_restore -l db.dump > db.list
```

列表文件由一个头部和一些行组成，这些行每一个都用于一个项，例如：

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

分号表示开始一段注释，行首的数字表明了分配给每个项的内部归档 ID。

文件中的行可以被注释掉、删除以及重排序。例如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

把这样一个文件作为pg_restore的输入将会只恢复项 10 和 6，并且先恢复 10 再恢复 6。

```
$ pg_restore -L db.list db.dump
```

另见

pg_dump, pg_dumpall, psql

psql

psql — PostgreSQL的交互式终端

大纲

```
psql [option...] [dbname [username]]
```

描述

psql是一个PostgreSQL的基于终端的前端。它让你能交互式地键入查询，把它们发送给PostgreSQL，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，psql还提供一些元命令和多种类似 shell 的特性来为编写脚本和自动化多种任务提供便利。

选项

-a
--echo-all

把所有非空输入行按照它们被读入的形式打印到标准输出（不适用于交互式行读取）。这等效于把变量ECHO设置为 all。

-A
--no-align

切换到非对齐输出模式（默认输出模式是对齐的）。这等效于\pset format unaligned。

-b
--echo-errors

把失败的 SQL 命令打印到标准错误输出。这等效于把变量ECHO设置为errors。

-c command
--command=command

指定psql执行一个给定的命令字符串command。这个选项可以重复多次并且以任何顺序与-f选项组合在一起。当-c或者-f被指定时，psql不会从标准输入读取命令，直到它处理完序列中所有的-c和-f选项之后终止。

command必须是一个服务器完全可解析的命令字符串（即不包含psql相关的特性）或者单个反斜线命令。因此不能在一个-c选项中混合SQL和psql元命令。要那样做，可以使用多个-c选项或者把字符串用管道输送到psql中，例如：

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

或者

```
echo '\x \\ SELECT * FROM foo;' | psql
```

（\\是分隔符元命令）。

每一个被传递给-c的SQL命令字符串会被当做一个单独的请求发送给服务器。因此，即便该字符串包括多个SQL命令，服务器也会把它当做一个事务来执行，除非在该字符串中有显式的BEGIN/COMMIT命令把它划分成多个事务（服务器如何处理多查询字符串的更多细

节请参考第 53.2.2.1 节。此外，psql只会打印出该字符串中最后一个SQL命令的结果。这和从文件中读取同一字符串或者把同一字符串传给psql的标准输出时的行为不同，因为那两种情况下psql会独立地发送每一个SQL命令。

由于这种行为，把多于一个SQL命令放在-c字符串中通常会得到意料之外的结果。最好使用多个-c命令或者把多个命令输送给psql的标准输入，按照上文所说的使用echo或者通过一个 shell，例如：

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

-d dbname
--dbname=dbname

指定要连接的数据库的名称。这等效于指定dbname为命令行上的第一个非选项参数。

如果这个参数包含一个=符号或者以一个合法的URI前缀（postgresql://或者postgres://）开始，它会被当作一个conninfo字符串。详见第 34.1.1 节

-e
--echo-queries

也把发送到服务器的所有 SQL 命令复制到标准输出。这等效于把变量ECHO设置为queries。

-E
--echo-hidden

回显\d以及其他反斜线命令生成的实际查询。可以用它来学习psql的内部操作。这等效于把变量ECHO_HIDDEN设置为on。

-f filename
--file=filename

从文件filename而不是标准输入中读取命令。这个选项可以被重复多次，也可以以任意顺序与-c选项组合。当-c或者-f被指定时，psql不会从标准输入读取命令，直到它处理完序列中所有的-c和-f选项之后终止。除此以外，这个选项很大程度上等价于元命令\i。

如果filename是-（连字符），那么会读取标准输入直到遇见一个 EOF 指示或者\q元命令。这种方式可以用把自多个文件的输入组合成一种交互式输入。不过注意在这种情况下不会使用 Readline（很像指定了-n的情况）。

使用这个选项与psql < filename有细微的不同。通常，两种形式都可以做到我们所期望的，但是使用-f启用了一些好的特性，例如带有行号的错误消息。使用这个选项还有一丝机会可以降低启动开销。在另一方面，使用 shell输入重定向的变体（理论上）保证会得到与手工输入时相同的输出。

-F separator
--field-separator=separator

使用separator作为非对齐输出的域分隔符。这等效于\pset fieldsep或者\f。

-h hostname
--host=hostname

指定运行服务器的机器的主机名。如果这个值由一个斜线开始，它会被用作 Unix 域套接字的目录。

-H
--html

打开HTML表格输出。这等效于\pset format html或者\H命令。

-l
--list

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这与元命令\list类似。

在使用这个选项时，psql将连接到数据库postgres，除非在命令行上提及一个不同的数据（选项-d或非选项参数，可能是通过一个服务项，但不能通过一个环境变量）。

-L filename
--log-file=filename

除了把所有查询输出写到普通输出目标之外，还写到文件filename中。

-n
--no-readline

不使用Readline做行编辑并且不使用命令历史。在剪切和粘贴时，关掉 Tab 展开会有所帮助。

-o filename
--output=filename

把所有查询输出放到文件filename中。这等效于命令\o。

-p port
--port=port

指定服务器用于监听连接的 TCP 端口或者本地 Unix 域套接字文件扩展。默认是PGPORT环境变量的值，如果没有设置，则默认为编译时指定的端口号（通常是5432）。

-P assignment
--pset=assignment

以\pset的形式指定打印选项。注意，这里你必须用一个等号而不是空格来分隔名称和值。例如，要设置输出格式为LaTeX，应该写成-P format=latex。

-q
--quiet

指定psql应该安静地工作。默认情况下，它会打印出欢迎消息以及多种输出。如果使用了这个选项，以上那些就都不会输出。在使用-c选项时，配合这个选项很有用。这等效于设置变量QUIET为on。

-R separator
--record-separator=separator

把separator用作非对齐输出的记录分隔符。这等效于\pset recordsep命令。

-s
--single-step

运行在单步模式中。这意味着在每个命令被发送给服务器之前都会提示用户一个可以取消执行的选项。使用这个选项可以调试脚本。

-S
--single-line

运行在单行模式中，其中新行会终止一个 SQL 命令，就像分号的作用一样。

> **注意**
>
> 这种模式被提供给那些坚持使用它的用户，但是并不一定要使用它。特别地，如果在一行中混合了SQL和元命令，那对于没有经的用户来说，它们的执行顺序可能不总是那么清晰。

-t
--tuples-only

关闭打印列名和结果行计数页脚等。这等效于\t或者\pset tuples_only命令。

-T table_options
--table-attr=table_options

指定要替换HTML table标签的选项。详见\pset tableattr。

-U username
--username=username

作为用户username而不是默认用户连接到数据库（当然，你必须具有这样做的权限）。

-v assignment
--set=assignment
--variable=assignment

执行一次变量赋值，和\set元命令相似。注意你必须在命令行上用等号分隔名字和值（如果有）。要重置一个变量，去掉等号就行。要把一个变量置为空值，使用等号但是去掉值。这些赋值在命令行处理期间被完成，因此反映连接状态的变量将在稍后被覆盖。

-V
--version

打印psql版本并且退出。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

注意这个选项将对整个会话保持设置，并且因此它会影响元命令\connect的使用，就像初始的连接尝试那样。

-W
--password

强制psql在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，psql将自动提示要求一个口令。但是，psql将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

注意这个选项将对整个会话保持设置，并且因此它会影响元命令\connect的使用，就像初始的连接尝试那样。

-x
--expanded

打开扩展表格式模式。这等效于\x或者\pset expanded命令。

-X,
--no-psqlrc

不读取启动文件（要么是系统范围的psqlrc文件，要么是用户的~/.psqlrc文件）。

-z
--field-separator-zero

设置非对齐输出的域分隔符为零字节。这等效于\pset fieldsep_zero。

-0
--record-separator-zero

设置非对齐输出的记录分隔符为零字节。例如，这对与xargs -0配合有关。这等效于\pset recordsep_zero。

-1
--single-transaction

这个选项只能被用于与一个或者多个-c以及/或者-f选项组合。它会让psql在第一个上述选项之前发出一条BEGIN命令并且在最后一个上述选项之后发出一条COMMIT命令，这样就把所有的命令都包裹在一个事务中。这个选项可以保证要么所有的命令都成功地完成，要么不应用任何更改。

如果命令本身包含BEGIN、COMMIT或者ROLLBACK，这个选项将不会得到想要的效果。还有，如果当个命令不能在一个事务块中执行，指定这个选项将导致整个事务失败。

-?
--help[=topic]

显示有关psql的帮助并且退出。可选的topic参数（默认为options）选择要解释哪一部分的psql：commands描述psql的反斜线命令；options描述可以被传递给psql的命令行选项；而variables则显示有关psql配置变量的帮助。

退出状态

如果psql正常完成，它会向 shell 返回 0。如果它自身发生一个致命错误（例如内存用完、找不到文件），它会返回 1。如果到服务器的连接出问题并且事务不是交互式的，它会返回2。如果在脚本中发生错误，它会返回 3 并且变量ON_ERROR_STOP会被设置。

用法

连接到数据库

psql是一个常规PostgreSQL客户端应用。为了连接到数据库，你需要知道你的目标数据库的名称、主机名和该服务器的端口号，还有要作为哪个用户名连接。可以通过命令行选项告知psql这些参数，分别是-d、-h、-p以及-U。如果发现一个参数不属于任何选项，它将被解释为数据库名称（如果已经给出数据库名称，就解释为用户名）。并非所有这些选项都是必需的，它们都有可用的默认值。如果省略主机名，psql将通过一个 Unix 域套接字连接到本地主机上的服务器，或者通过 TCP/IP 连接到没有 Unix 域套接字的主机上的localhost。默认端口号则在编译时决定。由于数据库服务器使用相同的默认值，大多数情况下你将不必指定端口。默认的用户名是你的操作系统用户名，它也会是默认的数据库名。注意你不一定能连接到任意用户名下的任何数据库。你的数据库管理员应该已经告知过你有关你的访问权限。

当默认值不是很符合实际时，可以把环境变量PGDATABASE、PGHOST、PGPORT以及PGUSER设置为适当的值，这样也能节省一些敲打键盘的工作（额外的环境变量可见第 34.14 节。用一个~/.pgpass文件来避免定期输入密码也很方便。详见第 34.15 节

另一种指定连接参数的方法是用一个conninfo字符串或者一个URI，它可以被用来替代数据库名。这种机制可以让我们对连接具有很广的控制权。例如：

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

用这种方式，你也可以把LDAP用于第 34.17 节描述的连接参数查找。可用连接选项的更多信息请见第 34.1.2 节

如果由于任何原因（例如权限不足、服务器没有在目标主机上运行等）导致连接无法建立，psql将返回一个错误并且终止。

如果标准输入和标准输出都是一个终端，那么psql会把客户端编码设置成“auto”，这会使psql从区域设置（Unix 系统上的LC_CTYPE环境变量）中检测合适的客户端编码。如果这样不起作用，可以使用环境变量PGCLIENTENCODING覆盖客户端编码。

输入 SQL 命令

在正常操作时，psql会提供一个提示符，该提示符是psql当前连接到的数据库名称后面跟上字符串=>。例如：

```
$ psql testdb
psql (11.2)
Type "help" for help.

testdb=>
```

在提示符下，用户可以键入SQL命令。正常情况下，当碰到一个表示命令终结的分号时，输入的行会被发送给服务器。一行的结束并不表示命令的完结。因此，为了清晰，可以把命令散布在多个行上。如果命令被发送并且执行而不产生错误，该命令的结果将会显示在屏幕上。

如果不可信用户对还没有采用安全方案使用模式的一个而数据库拥有访问，通过从search_path移除公共可写的方案来开始你的会话。人们可以在连接字符串中加入options=-csearch_path=或者在其他SQL命令之前发出SELECT pg_catalog.set_config('search_path', '', false)。这种考虑并非专门针对psql，它适用于每一种执行任意SQL命令的接口。

只要执行命令，psql还会测试LISTEN和NOTIFY产生的异步通知。

虽然 C 风格的注释块会被传给服务器处理并且移除，psql会自己移除掉 SQL 标准的注释。

元命令

你输入到psql中的任何以未加引用的反斜线开始的东西都是一个psql元命令，它们由psql自行处理。这些命令让psql对管理和编写脚本更有用。元命令常常被称作斜线或者反斜线命令。

psql命令的格式是用反斜线后面直接跟上一个命令动词，然后是一些参数。参数与命令动词和其他参数之间用任意多个空白字符分隔开。

要在一个参数中包括空白，可以将它加上单引号。要在一个参数中包括一个单引号，则需要在文本中写上两个单引号。任何包含在单引号中的东西都服从与 C 语言中\n（新行）、\t（制表符）、\b（退格）、\r（回车）、\f（换页）、\digits（10 进制）以及\xdigits（16 进制）类似的替换规则。单引号内文本中的其他任何字符（不管它是什么）前面的反斜线都没有实际意义（会被忽略）。

如果在一个参数中出现一个未加引号的冒号（:）后面跟着一个psql变量名，它会被该变量的值替换，如SQL 中插入变量中所述。在其中描述的形式:'variable_name'和:"variable_name"也有同样的效果。:{?variable_name}语法允许测

试一个变量是否被定义。它会被TRUE或FALSE替换。用一个反斜线转义该冒号可以防止它被替换。

在一个参数中，封闭在反引号（`）中的文本会被当做一个传递给shell的命令行。该命令的输出（移除任何拖尾的新行）会替换反引号文本。在封闭在反引号的文本中，不会有特别的引号或者其他处理发生，:variable_name的出现除外，其中variable_name是一个会被其值替换的psql变量名。此外，Also, appearances of :'variable_name'的出现会被替换为该变量的值，而值会被适当地加以引用以变成一个单一shell命令参数（后一种形式几乎总是优先，除非你非常确定变量中有什么）。因为回车和换行字符在所有的平台上都不能被安全地引用，:'variable_name'形式会打印一个错误消息并且在这类字符出现在值中时不替换该变量值。

有些命令把SQL标识符（例如一个表名）当作参数。这些参数遵循SQL的语法规则：无引号的字母被强制变为小写，而双引号（"）可以保护字母避免大小写转换并且允许在标识符中包含空白。 在双引号内，成对的双引号会被缩减为结果名称中的单个双引号。例如，FOO"BAR"BAZ会被解释成fooBARbaz，而"A weird"" name"会变成A weird" name。

对参数的解析会在行尾或者碰到另一个未加引号的反斜线时停止。一个未加引号的反斜线被当做新元命令的开始。特殊的序列\\（两个反斜线）表示参数结束并且应继续解析SQL命令（如果还有）。使用这种方法，SQL命令和psql命令可以被自由地混合在一行中。但是无论在何种情况中，元命令的参数都无法跨越一行。

很多元命令作用在当前查询缓冲区上。这就是一个缓冲区而已，它保存任何已经被键入但是还没有发送到服务器执行的SQL命令文本。这将包括之前输入的行以及在该元命令同一行上出现在前面的任何文本。

可以使用下列元命令：

\a

如果当前的表输出格式是非对齐的，则切换成对齐格式。如果不是非对齐格式，则设置成非对齐格式。保留这个命令是为了向后兼容性。更一般的方案请见\pset。

\c or \connect [-reuse-previous=on|off] [dbname [username] [host] [port] | conninfo]

与一台PostgreSQL服务器建立一个新连接。可以使用位置语法指定要使用的连接参数，或者使用第 34.1.1 节详细介绍的conninfo连接串。

在省略了数据库名、用户、主机或者端口的命令中，新的连接将会重用之前一个连接的值。默认情况下，前一个连接的值将会被重用，除非给出了一个conninfo串。给出第一个参数-reuse-previous=on或者-reuse-previous=off可以覆盖默认行为。当这个命令既没有指定一个参数也没有重用它时，将使用libpq的默认值。
把dbname、username、host或者port中的任何一个指定为-等价于省略该参数。

如果新连接成功地被建立，之前的连接会被关闭。如果连接尝试失败（错误的用户名、访问被拒绝等），只有在psql处于交互模式的情况下才会保留之前的连接。当执行一个非交互式脚本时出现连接尝试失败，处理将被立即停止，并且报出一个错误。这种区别一方面可以帮助用户发现打字错误，另一方面也可以作为一种安全机制防止脚本在错误的数据库上执行动作。

例子：

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10
 sslmode=disable"
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

\C [title]

设置查询结果的任何表的标题，或者重置这类标题。这个命令等效于\pset title title（这个命令的名称来自于“caption”，因为它之前只被用来在HTML表格中设置标题）。

\cd [directory]

把当前工作目录改为directory。如果不带参数，则切换到当前用户的主目录。

> **提示**
>
> 要打印当前的工作目录，可以使用\! pwd。

\conninfo

输出有关当前数据库连接的信息。

\copy { table [(column_list)] | (query) } { from | to } { 'filename' | program 'command' | stdin | stdout | pstdin | pstdout } [[with] (option [, ...])]

执行一次前端拷贝。这个操作会运行一个SQL COPY命令，不过不是服务器读取或者写入指定的文件，而是由psql读写文件并且把数据从本地文件系统导向服务器。这意味着文件的可访问性和权限是本地用户的而非服务器上的，并且不需要 SQL 超级用户特权。

当program被指定时，command被psql执行并且传给command的数据或者从command传出的数据会在服务器和客户端之间流动。同样地，执行特权是本地用户的而非服务器上的，并且不需要 SQL 超级用户特权。

对于\copy ... from stdin，数据行从发出该命令的同一来源读取，一直到读到\.或者数据流到达EOF。这个选项可以用来填充内嵌在一个 SQL 脚本文件中的表。对于\copy ... to stdout，输出被发送到与psql命令输出相同的位置，并且COPY count命令的状态不会被打印（因为它会被一个数据行搞乱）。要读/写psql的标准输入或者输出而不管当前命令的来源或者\o选项，可以写from pstdin或者to pstdout。

这个命令的语法和SQL COPY命令类似。所有除开数据来源/目的地的选项都和COPY指定的一样。因此，\copy元命令由特殊的解析规则。与大部分其他元命令不同，该行的所有剩余部分总是会被当做\copy的参数，并且在参数中不会执行变量篡改以及反引号展开。

> **提示**
>
> 获得与\copy ... to相同结果的另一种方法是使用SQL COPY ... TO STDOUT命令并使用\g filename或\ g | program 终止它。 \copy不与同，此方法允许命令跨越多行; 此外，可以使用变量插值和反引号扩展。

> **提示**
>
> 这些操作不如带有文件或程序数据源或目标的SQL COPY命令有效， 因为所有数据都必须通过客户端/服务器连接。 对于大量数据，SQL命令可能更可取。

\copyright

显示PostgreSQL的版权以及发布条款。

\crosstabview [colV [colH [colD [sortcolH]]]]

执行当前的查询缓冲区（像\g那样）并且在一个交叉表格子中显示结果。该查询必须返回至少三列。由colV标识的输出列会成为垂直页眉并且colH所标识的输出列会成为水平页眉。colD标识显示在格子中的输出列。sortcolH标识用于水平页眉的可选的排序列。

每一个列说明可以是一个列编号（从 1 开始）或者一个列名。常用的 SQL 大小写折叠和引用规则适用于列名。如果省略，colV被当做列 1 并且colH被当做列 2。colH必须和colV不同。如果没有指定colD，那么在查询结果中必须正好有三列，并且colV和colH之外的那一列会被当做colD。

垂直页眉显示为最左边的列，它包含列colV中找到的值，值的顺序和查询结果中的顺序相同，但是重复值会被移除。

水平页眉显示为第一行，它包含列colH中找到的值，其中的重复值被移除。默认情况下，这些值会以查询结果中相同的顺序出现。但是如果给出了可选的sortcolH参数，它标识一个值必须为整数编号的列，并且来自colH的值将会根据相应的sortcolH值排序后出现在水平页眉中。

在交叉表格子中，对于colH的每一个可区分的值x以及colV的每一个可区分的值y，位于交叉点(x,y)的单元包含colH值为x且colV值为y的查询结果行中colD列的值。如果没有这样的行，则该单元为空。如果有多个这样的行，则会报告一个错误。

\d[S+] [pattern]

对于每一个匹配pattern的关系（表、视图、物化视图、索引、序列或者外部表）或者组合类型，显示所有的列、它们的类型、表空间（如果非默认表空间）以及任何诸如NOT NULL或者默认值的特殊属性。相关的索引、约束、规则以及触发器也会被显示。对于外部表，还会显示相关的外部服务器（下文的模式（Pattern）中定义了“匹配模式”）。

对于某些类型的关系，\d会为每一列显示额外的信息：对于序列会显示列值，对于索引显示被索引的表达式，对于外部表显示外部数据包装器选项。

命令形式\d+是一样的，不过会显示更多信息：与该表的列相关的任何注释，表中是否存在 OID，如果关系是视图则显示视图定义，非默认的replica identity设置。

默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

> **注意**
>
> 如果使用\d但不带有pattern参数，它等价于\dtvmsE，后者将显示所有可见的表、视图、物化视图、序列和外部表的列表。这纯粹是一种便利措施。

\da[S] [pattern]

列出聚集函数，以及它们的返回类型和它们所操作的数据类型。如果指定了pattern，只显示名称匹配该模式的聚集。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

\dA[+] [pattern]

列出访问方法。如果指定了pattern，只显示名称匹配该模式的访问方法。如果在命令名称后面追加+，则与访问方法相关的处理器函数和描述也会和访问方法本身一起被列出。

\db[+] [pattern]

列出表空间。如果指定了pattern，只显示名称匹配该模式的表空间。如果在命令名称后面追加+，则与表空间相关的选项、磁盘上的尺寸、权限以及描述也会和表空间本身一起被列出。

\dc[S+] [pattern]

列出字符集编码之间的转换。如果指定了pattern，只列出名称匹配该模式的转换。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果在命令名称后面追加+，则每一个对象相关的描述也会被列出。

\dC[+] [pattern]

列出类型转换。如果指定了pattern，只列出源类型和目标类型匹配该模式的转换。如果在命令名称后面追加+，则每一个对象相关的描述也会被列出。

\dd[S] [pattern]

显示约束、操作符类、操作符族、规则以及触发器类型对象的描述。所有其他注释可以通过那些对象类型相应的反斜线命令查看。

\dd显示匹配pattern的对象的描述，如果没有给出参数则显示合适类型的可见对象的描述。但是在任一种情况下都只列出具有描述的对象。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

对象的描述可以用SQL命令COMMENT创建。

\dD[S+] [pattern]

列出域。如果指定了pattern，只有名称匹配该模式的域会被显示。默认情况下，只有用户创建的对象会被显示，可以提供一个模式或者S修饰符以包括系统对象。如果+被追加到命令名称上，每一个被列出的对象会带有其相关的权限和描述。

\ddp [pattern]

列出默认的访问特权设置。对那些默认特权设置已经被改变得与内建默认值不同的角色（以及模式，如果适用），为每一个角色（以及模式）显示一项。如果指定了pattern，只列出角色名称或者模式名称匹配该模式的项。

ALTER DEFAULT PRIVILEGES命令被用来设置默认访问特权。在GRANT中解释了显示的特权的含义。

\dE[S+] [pattern]
\di[S+] [pattern]
\dm[S+] [pattern]
\ds[S+] [pattern]
\dt[S+] [pattern]
\dv[S+] [pattern]

在这一组命令中，字母E、i、m、s、t和v分别对应着外部表、索引、物化视图、序列、表和视图。你可以以任何顺序指定这些字母中的任意一个或者多个，这样可以得到这些类型的对象的列表。例如，\dit会列出索引和表。如果在命令名称后面追加+，则每一个对象的物理尺寸以及相关的描述也会被列出。如果指定了pattern，只列出名称匹配该模式的对象。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

\des[+] [pattern]

列出外部服务器（助记：“外部服务器”）。如果指定了pattern，只列出名称匹配该模式的那些服务器。如果使用了\des+形式，将显示每个服务器的完整描述，包括该服务器的 ACL、类型、版本、选项和描述。

\det[+] [pattern]

列出外部表（助记：“外部表”）。如果指定了pattern，只列出表名称或者模式名称匹配该模式的项。如果使用了\det+选项，一般选项和外部表描述也会被显示。

\deu[+] [pattern]

列出用户映射（助记：“外部用户”）。如果指定了pattern，只列出用户名匹配该模式的那些映射。如果使用了\deu+形式，有关每个映射的额外信息也会被显示。

> **小心**
>
> \deu+可能也会显示远程用户的用户名和口令，所以要小心不要把它们泄露出去。

\dew[+] [pattern]

列出外部数据包装器（助记：“外部包装器”）。如果指定了pattern，只列出名称匹配该模式的那些外部数据包装器。如果使用了\dew+形式，外部数据包装器的 ACL、选项和描述也会被显示。

\df[anptwS+] [pattern]

列出函数，以及它们的结果数据类型、参数数据类型和函数类型，函数类型被分为“agg”（聚集）、“normal”、“procedure”、“trigger”以及“window”。如果要只显示指定类型的函数，可以在该命令上增加相应的字母a、n、p、t或者w。如果指定了pattern，只显示名称匹配该模式的函数。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果使用了\df+形式，则有关每个函数的额外信息也会被显示，包括易失性、并行安全性、拥有者、安全性分类、访问特权、语言、源代码和描述。

> **提示**
>
> 如果要查找接收指定数据类型参数或者返回指定类型值的函数，可以使用分页器的搜索能力来滚动显示\df输出。

\dF[+] [pattern]

列出文本搜索配置。如果指定了pattern，只显示名称匹配该模式的配置。如果使用了\dF+形式，每种配置的完整描述也会被显示，包括底层的文本搜索解析器和用于每一种解析器记号类型的字典列表。

\dFd[+] [pattern]

列出文本搜索字典。如果指定了pattern，只显示名称匹配该模式的字典。如果使用了\dFd+形式，有关每一种选中的字典的额外信息也会被显示，包括底层的文本搜索模板和选项值。

\dFp[+] [pattern]

列出文本搜索解析器。如果指定了pattern，只显示名称匹配该模式的解析器。如果使用了\dFp+形式，每一种解析器的完整描述也会被显示，包括底层的函数和可识别的记号类型列表。

\dFt[+] [pattern]

列出文本搜索模板。如果指定了pattern，只显示名称匹配该模式的模板。如果使用了\dFt+形式，每一种模板有关的额外信息也会被显示，包括底层的函数名称。

\dg[S+] [pattern]

列出数据库角色（因为“用户”和“组”的概念已经被统一成“角色”，这个命令现在等价于\du）。默认情况下只会显示用户创建的角色，提供一个模式或者S修饰符可以把

系统角色包括在内。如果指定了pattern，只列出名称匹配该模式的那些角色。如果使用了\dg+形式，有关每种角色的额外信息也将被显示，当前这种形式会为角色增加显示注释。

\dl

这是\lo_list的一个别名，它显示大对象的列表。

\dL[S+] [pattern]

列出过程语言。如果指定了pattern，只列出名称匹配该模式的语言。默认情况下只会显示用户创建的语言，提供一个模式或者S修饰符可以把系统对象包括在内。如果向命令名称追加+，则每一种语言会和它的调用处理器、验证器、访问特权以及它是否为系统对象一起列出。

\dn[S+] [pattern]

列出模式（名字空间）。如果指定了pattern，只列出名称匹配该模式的模式。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果向命令名称追加+，每个对象会与它相关的权限及描述（如果有）一起被列出。

\do[S+] [pattern]

列出操作符及其操作数和结果类型。如果指定了pattern，只列出名称匹配该模式的操作符。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果向命令名称追加+，有关每个操作符的额外信息也将被显示，当前只包括底层函数的名称。

\dO[S+] [pattern]

列出排序规则。如果指定了pattern，只列出名称匹配该模式的排序规则。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果向命令名称追加+，每个排序规则将和它相关的描述（如果有）一起被列出。注意只有可用于当前数据库编码的排序规则会被显示，因此在同一个安装下的不同数据库中执行此命令可能会得到不同的结果。

\dp [pattern]

列出表、视图和序列，包括与它们相关的访问特权。如果指定了pattern，只列出名称匹配该模式的表、视图以及序列。

GRANT和REVOKE命令被用来设置访问特权。所显示的特权的含义在GRANT中有介绍。

\drds [role-pattern [database-pattern]]

列出已定义的配置设置。这些设置可以是针对角色的、针对数据库的或者同时针对两者的。role-pattern和database-pattern分别被用来选择要列出的角色和数据库。如果省略它们或者指定了*，则会列出所有设置，分别会包括针对角色和针对数据库的设置。

ALTER ROLE以及ALTER DATABASE命令可以用来定义一个角色以及一个数据库的配置设置。

\dRp[+] [pattern]

列出复制的publication。如果指定有pattern，只有那些名称匹配该模式的publication会被列出。如果+被追加到命令的名称上，与每个publication相关的表也会被显示。

\dRs[+] [pattern]

列出复制的订阅。如果指定有pattern，只有那些名字匹配该模式的订阅才会被列出。如果+被追加到命令的名称上，订阅的额外属性会被显示。

\dT[S+] [pattern]

列出数据类型。如果指定了pattern，只列出名称匹配该模式的类型。如果向命令名称追加+，每一种类型、其内部名称和尺寸、允许的值（如果是一种enum类型）以及相关权限会被一同列出。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

\du[S+] [pattern]

列出数据库角色（因为“用户”和“组”的概念已经被统一成“角色”，这个命令现在等价于\dg）。默认情况下只会显示用户创建的角色，提供一个模式或者S修饰符可以把系统角色包括在内。如果指定了pattern，只列出名称匹配该模式的那些角色。如果使用了\du+形式，有关每一种角色的额外信息也会被显示，当前只会多显示角色的注释。

\dx[+] [pattern]

列出已安装的扩展。如果指定了pattern，只列出名称匹配该模式的那些扩展。如果使用了\dx+形式，所有属于每个匹配扩展的对象会被列出。

\dy[+] [pattern]

列出事件触发器。如果指定了pattern，只列出名称匹配该模式的事件触发器。如果在命令名称后面加上+，还会为每个列出的对象显示其相关的描述。

\e或\edit [filename] [line_number]

如果指定了filename，则它是被编辑的文件，在编辑器退出后，该文件的内容会被拷贝到当前查询缓冲区中。如果没有给定filename，当前查询缓冲区会被拷贝到一个临时文件中，并且接着以相同的方式编辑。或者，如果当前查询缓冲区为空，则最近被执行的查询会被拷贝到一个临时文件并且以同样的方式编辑。

然后会根据psql的一般规则重新解析查询缓冲区的新内容，把整个缓冲区当作一个单一行来处理。任何完整的查询都会被立即执行，也就是说，如果查询缓冲区包含一个分号或者以一个分号结尾，则到分号处为止的所有东西都会被执行。剩下的东西会在查询缓冲区中等待，输入分号或者\g会把它发送出去，输入\r会通过清除查询缓冲区来取消它。把缓冲区当作单一行主要会影响元命令：缓冲区中在一个元命令之后的任何东西都将被当作该元命令的参数，即便元命令之后的内容跨越多行也是如此。（因此不能以这种方式来制作使用元命令的脚本。应该使用\i。）

如果指定了一个行号，psql将会把游标（注意不是服务器端的游标）定位到文件或者查询缓冲区的指定行上。注意如果给出了一个全是数字的参数，psql就会假定它是行号而不是文件名。

> **提示**
>
> 关于如何配置以及自定义编辑器，请见环境。

\echo text [...]

把参数打印到标准输出，参数之间用一个空格分隔，最后加上一个新行。这可以用来在脚本的输出中间混入信息，例如：

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

如果第一个参数是一个没有加引号的-n，则不会加上最后的新行。

> **提示**
>
> 如果使用\o命令来重定向查询的输出，你可能希望使用\qecho来取代这个命令。

\ef [function_description [line_number]]

这个命令会以一个CREATE OR REPLACE FUNCTION或CREATE OR REPLACE PROCEDURE命令的形式取出并且编辑指定函数或过程的定义。编辑的方式与\edit完全相同。在编辑器退出后，更新过的命令将在查询缓冲区中等待，可以键入分号或者\g把它发出，也可以用\r取消之。

目标函数可以单独用名称或者用名称和参数（例如foo(integer, text)）来指定。如果有多于一个函数具有同样的名称，则必须给出参数的类型。

如果没有指定函数，将会给出一个空白的CREATE FUNCTION模板来编辑。

如果指定了一个行号，psql将把游标定位在该函数体的指定行上（注意函数体通常不是开始于文件的第一行）。

和大部分其他元命令不同，该行的整个剩余部分总是会被当作\ef的参数，并且在参数中不会执行变量篡改以及反引号展开。

> **提示**
>
> 有关如何配置和自定义编辑器可见环境。

\encoding [encoding]

设置客户端字符集编码。如果没有参数，这个命令会显示当前的编码。

\errverbose

以最详细的程度重复最近的服务器错误消息，就好像VERBOSITY被设置为verbose且SHOW_CONTEXT被设置为always。

\ev [view_name [line_number]]

这个命令会以一个CREATE OR REPLACE VIEW的形式取出并且编辑指定函数的定义。编辑的方式与\edit完全相同。在编辑器退出后，更新过的命令将在查询缓冲区中等待，可以键入分号或者\g把它发出，也可以用\r取消之。

如果没有指定函数，将会给出一个空白的CREATE VIEW模板来编辑。

如果指定了一个行号，psql将把游标定位在该视图定义的指定行上。

和大部分其他元命令不同，该行的整个剩余部分总是会被当作\ev的参数，并且在参数中不会执行变量篡改以及反引号展开。

\f [string]

设置用于非对齐查询输出的域分隔符。默认值是竖线（|）。它等效于\pset fieldsep。

\g [filename]
\g [|command]

把当前查询缓冲区发送给服务器执行。如果给出一个参数，查询的输出将被写到提到的文件或者用管道导向给定的shell命令，而不是按照惯常显示出来。只有该查询成功地返

回零或更多个元组时才会写文件或命令，如果查询失败或者不是一个数据返回SQL命令，则不会写文件或者导向shell命令。

如果当前查询缓冲区为空，则重新执行最近一次被发送的查询。除了这种行为之外，没有参数的\g实际上等效于一个分号。一个带有参数的\g是\o命令的一种“一次性”选择。

如果该参数以|开始，则该行的所有剩余部分总是会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传给shell。

\gdesc

显示当前查询缓冲区的结果的描述（即列名和数据类型）。查询不会被实际执行，不过，如果它含有某种类型的语法错误，该错误将被以通常的方式报出。

如果当前查询缓冲区为空，则会描述最近被发送的查询。

\gexec

把当前查询缓冲区发送到服务器，然后该查询输出（如果有）中的每一行的每一列都当作一个要被执行的 SQL 语句。例如，要在my_table的每一列上都创建一个索引：

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

产生的查询会按照其所在行被返回的顺序执行，如果有多个列，则同一行中按照从左至右的顺序执行。NULL 域会被忽略。产生的查询会被原样发送给服务器处理，因此它们即不能是psql元命令，也不能包含psql变量引用。如果其中任何一个查询失败，剩余查询的执行将会继续，除非设置了ON_ERROR_STOP。每个查询的执行都遵照ECHO的处理（在使用\gexec时，通常建议设置ECHO为all或者queries）。查询日志、单步模式、计时以及其他查询执行特性也适用于每一个生成的查询。

如果当前查询缓冲区为空，则会重新执行最近被发送的查询。

\gset [prefix]

把当前查询输入缓冲区发送给服务器并且将查询的输出存储在psql变量中（见变量）。被执行的查询必须只返回一行。该行的每一列会被存储到一个单独的变量中，变量和该列的名字一样。例如：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

如果指定了一个prefix，那么该字符串会被追加在该查询的输出列名称之前用来创建要使用的变量名：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
```

```
hello 10
```

如果一个列的结果为 NULL，那么对应的变量会被重置而不是被设置。

如果查询失败或者没有返回一行，则不会有任何变量被更改。

如果当前查询缓冲区为空，则重新执行最近被发送的查询。

```
\gx [ filename ]
\gx [ |command ]
```

\gx等效于\g，但会为这个查询强制扩展的输出模式。请参考\x。

```
\h or \help [ command ]
```

给出指定SQL命令的语法帮助。如果没有指定command，则psql会列出可以显示语法帮助的所有命令。如果command是一个星号（*），则会显示所有SQL命令的语法帮助。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\help的参数，并且在参数中不会执行变量篡改以及反引号展开。

> **注意**
>
> 为了简化输入，由几个词构成的命令不需要被加上引号。因此，键入\help alter table是可以的。

```
\H or \html
```

开启HTML查询输出格式。如果HTML格式已经开启，这会把它切换回默认的对齐文本格式。这个命令是为了兼容性和方便，有关设置其他输出选项请见\pset。

```
\i or \include filename
```

从文件filename读取输入并且把它当作从键盘输入的命令来执行。

如果filename是-（连字符），那么会一直读取标准输入直到碰到一个 EOF 指示符或者\q元命令。这可以用来把交互式输入与文件输入混杂。注意只有在最外层激活了 readline 行为的情况下才将会使用 readline 行为。

> **注意**
>
> 如果想在屏幕上看到被读入的行，必须把变量ECHO设置成all。

```
\if expression
\elif expression
\else
\endif
```

这一组命令实现可嵌套的条件块。条件块必须以一个\if开始并且以一个\endif结束。两者之间可能有任意数量的\elif子句，后面也可能有选择地跟着一个单一的\else子句。一般查询以及其他类型的反斜线命令可以出现在这些命令之间构成条件块。

\if和\elif命令读取它们的参数并且将它们作为布尔表达式进行计算。如果表达式得到真则处理正常继续下去，否则会跳过下面的行直到到达一个匹配的\elif、\else或者\endif。一旦一个\if或者\elif测试成功，同一个块中后面的\elif命令的参数将不会被计算但会被当作为假。跟在一个\else后面的行只有在先前的匹配的\if或\elif成功时才被处理。

就像任何其他反斜线命令参数一样，\if或者\elif命令的expression参数服从变量篡改以及反引号展开。然后会像一个on/off选项变量的值一样来计算它。因此，对下列项无歧义、大小写无关的匹配都是有效的值：true、false、1、0、on、off、yes、no。例如，t、T以及tR都将被认为是真。

无法被正确计算为真或假的表达式将产生一个警告并且被当做假。

正在被跳过的行还是会被正常地解析以标识查询和反斜线命令，但是查询不会被发送到服务器，并且非条件（\if、\elif、\else、\endif）反斜线命令会被忽略。条件命令会被检查以判断嵌套是否有效。被跳过的行中的变量引用不会被展开，并且也不会执行反引号展开。

一个给定条件块中的所有反斜线命令必须出现在相同的源文件中。如果在所有的本地\if块被关闭之前，主输入文件或者一个\include进来的文件上就达到了EOF，则psql将产生一个错误。

这里是一个例子：

```
-- 检查数据库中两个单独记录的存在性并且把结果存在单独的psql变量中
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

\ir or \include_relative filename

\ir命令类似于\i，但是以不同的方式处理相对路径文件名。在交互模式中执行时，这两个命令的行为相同。不过，当被从脚本中调用时，\ir相对于脚本所在的目录而不是根据当前工作目录来解释文件名。

\l[+] or \list[+] [pattern]

列出服务器中的数据库并且显示它们的名称、拥有者、字符集编码以及访问特权。如果指定了pattern，则只列出名称匹配该模式的数据库。如果向命令名称追加+，则还会显示数据库的尺寸、默认表空间以及描述（尺寸信息只对当前用户能连接的数据库可用）。

\lo_export loid filename

从数据库中读取具有OID　loid的大对象并且将它写入到filename。注意这和服务器函数lo_export有微妙的不同，后者会以运行数据库服务器的用户权限来执行并且运行在服务器的文件系统上。

> **提示**
>
> 使用\lo_list可以找出大对象的OID。

\lo_import filename [comment]

把该文件存储到PostgreSQL大对象。可选地，它可以把给定的注释关联到该对象。例如：

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

该响应表示该大对象得到的对象 ID 是 152801，未来可以用这个 ID 来访问这个新创建的大对象。为了便于阅读，推荐总是给每一个对象都关联人类可读的注释。OID 和注释都可以用\lo_list命令查看。

注意这个命令和服务器端的lo_import有微妙的不同，因为它以本地文件系统上的本地用户的身份运行，而不是服务器用户和文件系统。

\lo_list

显示当前存储在数据库中的所有PostgreSQL大对象，同时显示它们的任何注释。

\lo_unlink loid

从数据库中删除OID为loid的大对象。

> **提示**
>
> 使用\lo_list可以找出该大对象的OID。

\o or \out [filename]
\o or \out [|command]

安排把未来的查询结果保存到文件filename中或者用管道导向到 shell 命令command。如果没有指定参数，查询输出会被重置到标准输出。

如果该参数以|开始，则该行的所有剩余部分总是会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传给shell。

“查询结果”包括从数据库服务器得到的所有表、命令响应和提示，还有查询数据库的各种反斜线命令（如\d）的输出，但不包括错误消息。

> **提示**
>
> 要在查询结果之间混入文本输出，可以使用\qecho。

\p or \print

把当前查询缓冲区打印到标准输出。如果当前查询缓冲区为空，会打印最近被执行的查询。

\password [username]

更改指定用户（默认情况下是当前用户）的口令。这个命令会提示要求输入新口令、对口令加密然后把加密后的口令作为一个ALTER ROLE命令发送到服务器。这确保新口令不会以明文的形式出现在命令历史、服务器日志或者其他地方。

\prompt [text] name

提示用户提供一个文本用于分配给变量name。可以指定一个可选的提示字符串text（对于多个词组成的提示，把文本包裹在单引号中）。

默认情况下，\prompt使用终端进行输入和输出。不过，如果使用了-f命令行开关，\prompt会使用标准输入和标准输出。

\pset [option [value]]

这个命令设置影响查询结果表输出的选项。option表示要设置哪个选项。value的语义取决于选中的选项。对于某些选项，如果省略value会导致该选项值被切换或者被重置，具体是哪些选项可见特定选项的描述。如果没有上面提到的那种行为，那么省略value只会导致当前设置被显示。

不带任何参数的\pset显示所有打印选项的当前状态。

可调整的打印选项有：

border

value必须是一个数字。通常，数字越大，表格就会有更多的边框和线条，但具体要看是哪一种格式。在HTML格式中，这会直接被转换成border=...属性。在大部分其他格式中，只有值 0（没有边框）、1（内部分隔线）和 2（表格边框）有意义，并且 2 以上的值会被视为与border = 2相同。latex和latex-longtable格式会额外地允许一个值 3 表示在数据行之间增加分隔线。

columns

为wrapped格式设置目标宽度，还有扩展自动模式中决定输出是否足够多到需要分页器或者切换到垂直显示的宽度限制。零（默认）导致目标宽度由环境变量COLUMNS所控制，如果没有设置COLUMNS则使用检测到的屏幕宽度。此外，如果columns为零则wrapped格式只影响屏幕输出。如果columns为非零则文件和管道输出也会被包裹成该宽度。

expanded (or x)

如果value被指定，它必须是on或者off，它们分别会启用或者禁用扩展模式，也可以是auto。如果value被省略，则该命令会在开启和关闭设置之间切换。当扩展模式被启用时，查询结果被显示在两列中，第一列是列名而第二列是列值。如果在通常的“水平”模式中数据不适合屏幕，则可以用这种模式。在自动设置中，只要查询输出有多于一列并且比屏幕宽，就会使用扩展模式。否则，将使用常规模式。只有在对齐格式和 wrapped 格式中自动设置才有效。在其他格式中，它的行为总是像扩展模式被关闭一样。

fieldsep

指定在非对齐输出格式中使用的域分隔符。用那种方式，用户可以创建 tab 或者逗号分隔的输出，这种形式其他程序可能更喜欢。要设置 tab 为域分隔符，可以键入\pset fieldsep '\t'。默认的域分隔符是'|'（一个竖线）。

fieldsep_zero

把用在非对齐输出格式中的域分隔符设置为一个零字节。

footer

如果value被指定，它必须是on或者off，它们分别会启用或者禁用表格页脚（(n rows)计数）的显示。如果value被省略，则该命令会切换页脚显示为打开或者关闭。

format

设置输出格式为unaligned、aligned、wrapped、html、asciidoc、latex（使用tabular）、latex-longtable或者troff-ms之一。也可以使用不造成歧义的缩写（这意味着一个字母就够了）。

unaligned格式把一个数据行的所有列都写在一行上，之间用当前活动的域分隔符分隔。这可用于生成意图由其他程序读取的输出（例如，tab 分隔或者逗号分隔格式）。

aligned格式是标准的、人类可读的、格式化好的文本输出，这是默认格式。

wrapped格式和aligned相似，但是前者会把过宽的数据值分成多个行以便输出能够适合目标行的宽度。目标行的宽度由columns选项决定。注意psql将不会尝试对列头部标题进行换行，因此如果列头部需要的总宽度超过目标宽度，wrapped格式的行为就变得和aligned一样了。

html、asciidoc、latex、latex-longtable和troff-ms格式分别用相应的标记语言把要输出的表格放在文档中，不过它们的输出并不是完整的文档。在HTML中这可能并不重要，但是在LaTeX中必须有完整的文档。latex-longtable还要求有LaTeX的longtable以及booktabs包。

linestyle

设置边框线的绘制样式为ascii、old-ascii或者unicode之一。允许不产生歧义的缩写（这意味着一个字母就足够了）。默认的设置是ascii。这个选项只影响aligned以及wrapped输出格式。

ascii样式使用纯ASCII字符。数据中的新行使用一个+符号在右手边的空白处显示。当在wrapped格式中包裹两行中间没有新行字符的数据时，会在第一行右手边空白处显示一个点号（.），并且在下一行的左手边空白处也显示一个点号（.）。

old-ascii样式使用纯ASCII字符，使用PostgreSQL 8.4 及更早版本中用过的格式化样式。数据中的新行使用:符号来代替左手边的列分隔符显示。在包裹两行中间没有新行字符的数据时，会用一个;符号取代左手边的列分隔符。

unicode样式使用 Unicode 的方框绘制字符。数据中的新行会使用一个回车符号显示在右手边的空白处。在包裹两行中间没有新行字符的数据时，会在第一行的右手边空白处显示一个省略号，并且在下一行的左手边空白处也显示一个省略号。

当border设置大于零时，linestyle选项也决定边框线用什么字符绘制。纯ASCII字符到处都可以使用，但是在识别 Unicode 字符的显示上使用 Unicode 字符会更好看。

null

设置要用来替代空值被打印的字符串。默认是什么也不打印，对于一个空字符串这很容易弄错。例如，有人可能更想用\pset null '(null)'。

numericlocale

如果value被指定，它必须是on或者off，它们将分别启用或者禁用一个与区域相关的字符来分隔数字和左边的十进制标记。如果value被省略，该命令会在常规输出和区域相关的数字输出之间切换。

pager

控制对查询和psql的帮助输出使用分页器程序。如果环境变量PSQL_PAGER或PAGER被设置，输出会被用管道输送到指定的程序。否则将使用与平台相关的默认分页器程序（例如more）。

如果pager选项被设为off，则不会使用分页器程序。如果pager选项被设为on，则会在适当的时候使用分页器，即当输出到终端并且无法适合屏幕时就会使用分页器。pager选项也可以被设置为always，这会导致对所有的终端输出都是用分页器而不管输出是否适合屏幕。不带value的\pset pager会切换分页器开、关状态。

pager_min_lines

如果pager_min_lines被设置为一个大于页面高度的数字，在至少这么多输出行被显示之前都不会调用分页器程序。默认设置为 0。

recordsep

指定用在非对齐输出格式中的记录（行）分隔符。

recordsep_zero

把用在非对齐输出格式中的记录分隔符设置为一个零字节。

tableattr (or T)

在HTML格式中，这会指定要放在table标记内的属性。例如，这可能是cellpadding或者bgcolor。注意你可能不想在这里指定border，因为那由\pset border负责。如果没有给出value，则表属性会被重置。

在latex-longtable格式中，这个选项控制每个包含左对齐数据类型的列的宽度比例。这个选项的值是一个由空格分隔的值列表，例如'0.2 0.2 0.6'。没有指定的输出列会使用最后一个指定的值。

title (or C)

设置用于任何后续被打印表的表标题。这可以用来给输出加上描述性的标签。如果没有给出value，这个标题会被复原。

tuples_only (or t)

如果value被指定，它必须是on或者off，这个选项将启用或者禁用只显示元组的模式。如果value被省略，则该命令会在常规输出和只显示元组输出之间切换。常规输出包括列头、标题以及多种页脚之类的额外信息。在只显示元组的模式中，只会显示实际的表数据。

unicode_border_linestyle

设置unicode线型的边框绘制风格为single或者double之一。

unicode_column_linestyle

设置unicode线型的列绘制风格为single或者double之一。

unicode_header_linestyle

设置unicode线型的页眉绘制风格为single或者double之一。

这些不同格式的外观可以在示例小节的图示中看到。

> **提示**
>
> \pset有多种快捷命令。请参见\a、\C、\f、\H、\t、\T以及\x。

\q or \quit

退出psql程序。在一个脚本文件中，只有该脚本的执行会被终止。

\qecho text [...]

这个命令和\echo一样，不过输出将被写到\o所设置的查询输出通道。

\r or \reset

重置（清除）查询缓冲区。

\s [filename]

打印psql的命令行历史到filename。如果省略filename，该历史会被写入到标准输出（如果适用则使用分页器）。如果编译psql时没有加上Readline支持，则这个命令不可用。

\set [name [value [...]]]

设置psql变量name为value，如果给出了多于一个值，则把该变量的值设置为所有给出的值的串接。如果只给了一个参数，该变量会被设置为空字符串值。要重置一个变量，可以使用\unset 命令。

不带任何参数的\set显示所有当前设置的psql变量的名称和值。

合法的变量名可以包含字母、数字和下划线。详见下文的变量。变量名是大小写敏感的。

某些变量是特殊的，它们控制psql的行为或者会被自动设置以反映连接状态。这些变量在下文的变量中记录。

> **注意**
>
> 这个命令和SQL命令SET无关。

\setenv name [value]

把环境变量name设置为value，如果没有提供value，则会重置该环境变量。例如：

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

\sf[+] function_description

这个命令以一个CREATE OR REPLACE FUNCTION命令或者CREATE OR REPLACE PROCEDURE命令取出并且显示指定函数或者过程的定义。定义会被打印到当前的查询输出渠道，就像\o所作的那样。

目标函数可以单独用名称指定，也可以用名称和参数指定，例如foo(integer, text)。如果有多于一个函数具有相同的名字，则必须给出参数的类型。

如果向命令名称追加+，那么输出行会被编号，函数体的第一行会被编为 1。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\sf的参数，并且在参数中不会执行变量篡改以及反引号展开。

\sv[+] view_name

这个命令以一个CREATE OR REPLACE VIEW命令取出并且显示指定视图的定义。定义会被打印到当前的查询输出渠道，就像\o所作的那样。

如果在命令名称上追加+，那么输出行会从 1 开始编号。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\sv的参数，并且在参数中不会执行变量篡改以及反引号展开。

\t

切换输出列名标题和行计数页脚的显示。这个命令等效于\pset tuples_only，提供它只是为了使用方便而已。

\T table_options

指定在HTML输出格式中，要放在table标签内的属性。这个命令等效于\pset tableattr table_options。

\timing [on | off]

如果给出一个参数，这个参数用来打开或者关闭对每个SQL语句执行时长的显示。如果没有参数，则在打开和关闭之间切换。显示的数据以毫秒为单位，超过1秒的区间还会被显示为“分钟:秒”的格式，如果必要还会加上小时和日的字段。

\unset name

重置（删除）psql变量name。

大部分控制psql行为的变量不能被重置，相反，\unset命令会被解释为把它们设置为其默认值。请参考下文的变量。

\w or \write filename
\w or \write |command

把当前查询缓冲区写到文件filename或者用管道导出到 shell 命令command。如果当前查询缓冲区为空，则写最近被执行的查询。

如果参数以|开始，则该行的整个剩余部分会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传递给shell。

\watch [seconds]

反复执行当前的查询缓冲区（就像\g那样）直到被中止或者查询失败。两次执行之间等待指定的秒数（默认是 2 秒）。显示每个查询结果时带上一个由\pset title字符串（如果有）、从查询开始起的时间以及延时间隔组成的页眉。

如果当前查询缓冲区为空，则会重新执行最近被发送的查询。

\x [on | off | auto]

设置或者切换扩展表格格式化模式。究其本身而言，这个命令等效于\pset expanded。

\z [pattern]

列出表、视图和序列，以及它们相关的访问特权。如果指定了pattern，则只会列出名称匹配该模式的表、视图和序列。

这是\dp（“display privileges”）的一个别名。

\! [command]

如果没有参数，就跳出到一个子shell，当子shell退出时psql会继续。如果有一个参数，则执行shell命令command。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\!的参数，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传递给shell。

\? [topic]

显示帮助信息。可选的topic参数（默认是commands）选择解释psql的哪一部分：commands表示psql的反斜线命令；options表示可以传递给psql的命令行选项；而variables显示有关psql配置变量的帮助。

\;

反斜线分号并非和前述命令相同的元命令，它只是会把一个分号加入到查询缓冲区且不会进一步执行。

通常，只要psql达到了命令结束的分号，它就将分发一个SQL命令给服务器，即使在当前行上还留有更多输入。因此，例如输入

```
select 1; select 2; select 3;
```

将导致三个SQL命令被逐个发送给服务器，在继续到下一个命令前会显示每一个命令的结果。不过，被输入为\;的分号将不会触发命令处理，这样在它之前的命令以及其后的命令实际上会被组合在一个请求中发送给服务器。例如

```
select 1\; select 2\; select 3;
```

会导致在到达非反斜线分号时用一个单一的请求把三个SQL命令发送给服务器。服务器会把这样一个请求当作单一的事务执行，除非该字符串中有显式的BEGIN/COMMIT命令把它划分成多个事务（服务器如何处理多查询字符串的更多细节请参考第 53.2.2.1 节。psql对每个请求仅打印出它接收到的最后一个查询结果。在这个例子中，尽管所有三个SELECT确实都被执行了，但psql只会打印出3。

模式（Pattern）

很多\d命令都可以用一个pattern参数来指定要被显示的对象名称。在最简单的情况下，模式正好就是该对象的准确名称。在模式中的字符通常会被变成小写形式（就像在 SQL 名称中那样），例如\dt FOO将会显示名为foo的表。就像在 SQL 名称中那样，把模式放在双引号中可以阻止它被转换成小写形式。如果需要在一个模式中包括一个真正的双引号字符，则需要把它写成两个相邻的双引号，这同样是符合 SQL 引用标识符的规则。例如，\dt "FOO""BAR"将显示名为FOO"BAR（不是foo"bar）的表。和普通的 SQL 名称规则不同，你不能只在模式的一部分周围放上双引号，例如\dt FOO"FOO"BAR将会显示名为fooFOObar的表。

只要pattern参数被完全省略，\d命令会显示在当前 schema 搜索路径中可见的全部对象 — 这等价于用*作为模式（如果一个对象所在的 schema 位于搜索路径中并且没有同类且同名的对象出现在搜索路径中该 schema 之前的 schema 中，则说该对象是可见的。这表示可以直接用名称引用该对象，而不需要用 schema 来进行限定）。要查看数据库中所有的对象而不管它们的可见性，可以把*.*用作模式。

如果放在一个模式中，*将匹配任意字符序列（包括空序列），而?会匹配任意的单个字符（这种记号方法就像 Unix shell 的文件名模式一样）。例如，\dt int*会显示名称以int开始的表。但是如果被放在双引号内，*和?就会失去这些特殊含义而变成普通的字符。

包含一个点号（.）的模式被解释为一个 schema 名称模式后面跟上一个对象名称模式。例如，\dt foo*.*bar*会显示名称以foo开始的 schema 中所有名称包括bar的表。如果没有出现点号，那么模式将只匹配当前 schema 搜索路径中可见的对象。同样，双引号内的点号会失去其特殊含义并且变成普通的字符。

高级用户可以使用字符类等正则表达式记法，如[0-9]可以匹配任意数字。所有的正则表达式特殊字符都按照第 9.7.3 节所说的工作，以下字符除外：.会按照上面所说的作为一种分

隔符，*会被翻译成正则表达式记号.*，?会被翻译成.，而$则按字面意思匹配。根据需要，可以通过书写?、(R+|)、(R|)和R?来分别模拟模式字符.、R*和R?。$不需要作为一个正则表达式字符，因为模式必须匹配整个名称，而不是像正则表达式的常规用法那样解释（换句话说，$会被自动地追加到模式上）。如果不希望该模式的匹配位置被固定，可以在开头或者结尾写上*。注意在双引号内，所有的正则表达式特殊字符会失去其特殊含义并且按照其字面意思进行匹配。还有，在操作符名称模式中（即作为\do的参数），正则表达式特殊字符也按照字面意思进行匹配。

高级特性

变量

psql提供了和普通 Unix 命令 shell 相似的变量替换特性。变量简单来说就是一对名称/值，其中值可以是任意长度的任意字符串。名称必须由字母（包括非拉丁字母）、数字和下划线构成。

要设置一个变量，可以使用psql的元命令\set。例如，

```
testdb=> \set foo bar
```

会设置foo为值bar。要检索该变量的内容，可以在名称前放一个分号，例如：

```
testdb=> \echo :foo
bar
```

这在常规 SQL 命令和元命令中均有效，下文的SQL 中插入变量中有更多细节。

如果调用\set时没有第二个参数，该变量会被设置为一个空字符串值。要重置（即删除）一个变量，可以使用命令\unset。要显示所有变量的值，在调用\set时不带任何参数即可。

> **注意**
>
> \set的参数服从与其他命令相同的替换规则。因此可以构造有趣的引用，例如\set :foo 'something'以及分别得到Perl或者PHP的“软链接”或者“可变变量”。不幸的是（或者幸运的是？），这些构造出来的东西并没有什么用处。在另一方面，\set bar :foo是一种很好的拷贝变量的方法。

有一些变量会被psql特殊对待。它们表示特定的选项设置，运行时这类选项设置可以通过修改该变量的值来改变，或者在某些情况下它们表示psql的可更改的状态。按照惯例，所有被特殊对待的变量的名称由全部大写形式的 ASCII 字母（还有可能是数字和下划线）组成。为了确保未来最大的兼容性，最好避免把这类变量名用于自己的目的。

控制psql行为的变量通常不能被重置或者设置为无效值。允许\unset命令，但它会被解释为将变量设置为它的默认值。没有第二参数的\set命令会被解释为将变量设置为on（对于接受该值的控制变量），对不接受该值的变量则会拒绝这个命令。此外，接受值on和off的控制变量也能接受其他常见的布尔值拼写方式，例如true和false。

被特殊对待的变量是：

AUTOCOMMIT

在被设置为on（默认）时，每一个 SQL 命令在成功完成时会被自动提交。在这种模式中要推迟提交，必须输入一个BEGIN或者START TRANSACTION SQL 命令。当被设置为off或者被重置时，在显式发出COMMIT或者END之前，SQL 命令不会被提交。自动提交打开模式会为你发出一个隐式的BEGIN，这会发生在任何不在一个事务块中且本身即不是BEGIN及其他事务控制命令且不是无法在事务块中执行的命令（例如VACUUM）之前。

> **注意**
>
> 在自动提交关闭模式中，必须通过ABORT或者ROLLBACK显式地放弃任何失败的事务。还要记住，如果退出会话时没有提交，则所有的工作都会丢失。

> **注意**
>
> 自动提交打开模式是PostgreSQL的传统行为，但是自动提交关闭模式更接近于 SQL 的规范。如果更喜欢自动提交关闭模式，可以在系统级的psqlrc文件或者个人的~/.psqlrc文件中设置它。

COMP_KEYWORD_CASE

确定在补全一个 SQL 关键词时要使用的大小写形式。如果被设置为lower或者upper，补全后的词将分别是小写或者大写形式。如果被设置为preserve-lower或者preserve-upper（默认），补全后的词将会保持该词已输入部分的大小写形式，但是如果被补全的词还没有被输入，则它会被分别补全成小写或者大写形式。

DBNAME

当前已连接的数据库名称。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

ECHO

如果被设置为all，所有非空输入行会被按照读入它们的样子打印到标准输出（不适用于交互式读取的行）。要在程序开始时选择这种行为，可以使用开关-a。如果被设置为queries，psql会在发送每个查询给服务器时将它们打印到标准输出。选择这种行为的开关是-e。如果被设置为errors，那么只有失败的查询会被显示在标准错误输出上。这种行为的开关是-b。如果被重置或者设置为none（默认值）则不会显示任何查询。

ECHO_HIDDEN

当这个变量被设置为on且一个反斜线命令查询数据库时，相应的查询会被先显示。这种特性可以帮助我们学习PostgreSQL的内部并且在自己的程序中提供类似的功能（要在程序开始时选择这种行为，可以使用开关-E）。如果把这个变量设置为值noexec，则对应的查询只会被显示而并不真正被发送给服务器执行。默认值是off。

ENCODING

当前的客户端字符集编码。每一次你连接到一个数据库（包括程序启动）时以及当你用\encoding更改编码时，这个变量都会被设置，但它可以被更改或者重置。

ERROR

如果上一个SQL查询失败则为true，如果成功则是false。另见SQLSTATE。

FETCH_COUNT

如果这个变量被设置为一个大于零的整数值，SELECT查询的结果会以一组一组的方式取出并且显示（而不是像默认的那样把整个结果集拿到以后再显示），每一组就会包括这么多个行。因此，这种方式只会使用有限的内存量，而不管整个结果集的大小。在启用这个特性时，通常会使用 100 到 1000 的设置。记住在使用这种特性时，一个查询可能会在已经显示了一些行之后失败。

> **提示**
>
> 尽管可以把这种特性用于任何的输出格式，但是默认的aligned格式看起来会比较糟糕，因为每一组的FETCH_COUNT个行将被单独格式化，这就会导致不同的行组的列宽不同。其他的输出格式会更好。

HISTCONTROL

如果这个变量被设置为ignorespace，则以一个空格开始的行不会被放入到历史列表中。如果被设置为值ignoredups，则匹配之前的历史行的行不会被放入。值ignoreboth组合了上述两种值。如果被重置或者被设置为none（默认值），所有在交互模式中被读入的行都会保存在历史列表中。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

HISTFILE

该文件名将被用于存储历史列表。如果被重设，文件名将从PSQL_HISTORY环境变量中取得。如果该环境变量也没有被设置，则默认值是~/.psql_history，在Windows上是%APPDATA%\postgresql\psql_history。例如，

```
\set HISTFILE ~/.psql_history- :DBNAME
```

放在~/.psqlrc中将会导致psql为每一个数据库维护一个单独的历史。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

HISTSIZE

存储在命令历史中的最大命令数（默认值是500）。如果被设置为一个负值，则不会应用限制。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

HOST

当前连接到的数据库服务器端口。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

IGNOREEOF

如果被设置为1或者更小，向一个psql的交互式会话发送一个EOF字符（通常是Control+D）将会终止应用。如果设置为一个较大的数字值，则必须键入多个连续的EOF字符才能让交互式会话终止。如果该变量被设置为一个非数字值，则它会被解释为10。默认值为0。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

LASTOID

最后被影响的 OID 的值，这可能会由INSERT或者\lo_import命令返回。这个变量只保证在下一个SQL命令被显示完之前有效。

LAST_ERROR_MESSAGE
LAST_ERROR_SQLSTATE

当前psql会话中最近一个失败查询的主错误消息和相关的SQLSTATE代码，如果在当前会话中没有发生错误，则是一个空字符串和00000。

ON_ERROR_ROLLBACK

当被设置为on时，如果事务块中的一个语句产生一个错误，该错误会被忽略并且该事务会继续。当被设置为interactive时，只在交互式会话中忽略这类错误，而读取脚本文件时则不会忽略错误。当被重置或者设置为off（默认值）时，事务块中产生错误的一个语句会中止整个事务。错误回滚模式的工作原理是在事务块的每个命令之前都为你发出一个隐式的SAVEPOINT，然后在该命令失败时回滚到该保存点。

ON_ERROR_STOP

默认情况下，出现一个错误后命令处理会继续下去。当这个变量被设置为on后，出现错误后命令处理会立即停止。在交互模式下，psql将会返回到命令提示符；否则，psql将会退出并且返回错误代码 3 来把这种情况与致命错误区分开来，致命错误会被报告为错误代码 1。在两种情况下，任何当前正在运行的脚本（顶层脚本以及任何它已经调用的其他脚本）将被立即中止。如果顶层命名字符串包含多个 SQL 命令，将在当前命令处停止处理。

PORT

当前连接到的数据库服务器端口。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

PROMPT1
PROMPT2
PROMPT3

这些变量指定psql发出的提示符的模样。见下文的提示符。

QUIET

把这个变量设置为on等效于命令行选项-q。在交互模式下可能用处不大。

ROW_COUNT

上一个SQL查询返回的行数或者受影响的行数，如果该查询失败或者没有报告行计数则为0。

SERVER_VERSION_NAME
SERVER_VERSION_NUM

字符串形式的服务器版本号，例如9.6.2、10.1或者11beta1，以及数字形式的服务器版本号，例如90602或者100001。每次你连接到一个数据库（包括程序启动）时，这些都会被设置，但可以被改变或者重设。

SHOW_CONTEXT

这个变量可以被设置为值never、errors或者always来控制是否在来自服务器的消息中显示CONTEXT域。默认是errors（表示在错误消息中显示上下文，但在通知和警告消息中不显示）。当VERBOSITY被设置为terse时，这个设置无效（另见\errverbose，它可以用来得到刚遇到的错误的详细信息）。

SINGLELINE

设置这个变量为on等效于命令行选项-S。

SINGLESTEP

设置这个变量为on等效于命令选项-s。

SQLSTATE

与上一个SQL查询的失败相关的错误代码（见附录 A ，如果上一个查询成功则为00000。

USER

当前连接的数据库用户。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

VERBOSITY

这个变量可以被设置为值default、verbose或者terse来控制错误报告的详细程度（另见\errverbose，在想得到之前的错误的详细版本时使用）。

VERSION
VERSION_NAME
VERSION_NUM

这些变量在程序启动时被设置以反映psql的版本，分别是一个详细的字符串、一个短字符串（例如9.6.2、10.1或者11beta1）以及一个数字（例如90602或者100001）。它们可以被更改或重设。

SQL 中插入变量

psql变量的一个关键特性是可以把它们替换（“插入”）到常规SQL语句中，也可以把它们作为元命令的参数。此外，psql还提供了功能来确保被用作 SQL 文字和标识符的变量值会被正确地引用。插入一个值而不需要加引用的语法是在变量名前面加上一个冒号（:）。例如，

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

将查询表my_table。注意这可能会不安全：该变量的值会被按字面拷贝，因此它可能包含不平衡的引号甚至反斜线命令。必须确保把它放在那里是有意义的。

当一个值被用作 SQL 文本或者标识符时，最安全的是把它加上引用。要引用一个变量的值作为 SQL 文本，可以把变量名称放在单引号中并且在引号前面写一个冒号。要引用作为 SQL 标识符，则可以把变量名称放在双引号中并且在引号前面写一个冒号。这种结构可以正确地处理变量值中嵌入的引号和其他特殊字符。之前的例子用这种方法写会更安全：

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

在被引用的SQL文本和标识符中将不会执行变量插入。因此，一个诸如':foo'的结构不会从一个变量的值产生一个被引用的文本（即便能够也会不安全，因为无法正确地处理嵌入在值中的引号）。

使用这种机制的一个例子是把一个文件的内容拷贝到一个表列中。首先把该文件载入到一个变量，然后把该变量的值作为一个被引用的字符串插入：

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

（注意如果my_file.txt包含 NUL 字节，这样也不行。psql不支持在变量值中嵌入 NUL 字节）。

因为冒号可以合法地出现在 SQL 命令中，一次明显的插入尝试（即:name、:'name'或者:"name"）不会被替换，除非所提及的变量就是当前被设置的。在任何情况下，可以用一个反斜线对冒号进行转义以避免它被替换。

:{?name}特殊语法根据该变量存在与否返回TRUE或者FALSE，并且因此总是会被替换，除非分号被反斜线转义。

变量的冒号语法对嵌入式查询语言（例如ECPG）来说是标准的SQL。用于数组切片和类型造型的冒号语法是PostgreSQL扩展，它有时可能会与标准用法冲突。把一个变量值转义成 SQL 文本或者标识符的冒号引用语法是一种psql扩展。

提示符

psql发出的提示符可以根据用户的喜好自定义。PROMPT1、PROMPT2和PROMPT3这三个变量包含了描述提示符外观的字符串和特殊转义序列。Prompt 1 是当psql等待新命令时发出的常规提示符。Prompt 2 是在命令输入时需要更多输入时发出的提示符，例如因为当命令没有被分号终止或者引用没有被关闭时就会发出这个提示符。在运行一个SQL COPY FROM STDIN命令并且需要在终端上输入一个行值时，会发出 Prompt 3。

被选中的提示符变量会被原样打印，除非碰到一个百分号（%）。百分号的下一个字符会被特定的其他文本替换。预定义好的替换有：

%M

数据库服务器的完整主机名（带有域名），或者当该连接是建立在一个 Unix 域套接字上时则是[local]，或者当 Unix 域套接字不在编译在系统内的默认位置上时则是[local:/dir/name]。

%m

数据库服务器的主机名称（在第一个点处截断），或者当连接建立在一个 Unix 域套接字上时是[local]。

%>

数据库服务器正在监听的端口号。

%n

数据库会话的用户名（在数据库会话期间，这个值可能会因为命令SET SESSION AUTHORIZATION的结果而改变）。

%/

当前数据库的名称。

%~

和%/类似，但是如果数据库是默认数据库时输出是~（波浪线）。

%#

如果会话用户时一个数据库超级用户，则是#，否则是一个>（在数据库会话期间，这个值可能会因为命令SET SESSION AUTHORIZATION的结果而改变）。

%p

当前连接到的后端的进程 ID。

%R

在提示符1下通常是=，但如果会话位于一个条件块的一个非活动分支中则是@，如果会话处于单行模式中则是^，如果会话从数据库断开连接（\connect失败时会发生这种情况）则是!。在提示符 2 中，根据为什么psql期待更多的输入，%R会被一个相应的字符替换：如果命令还没有被终止是-，如果有一个未完的/* ... */注释则是*，如果有一个未完的被引用字符串则是一个单引号，如果有一个未完的被引用标识符则是一个双引号，如果有一个未完的美元引用字符串则是一个美元符号，如果有一个还没有被配对的左圆括号则是(。在提示符 3 中%R不会产生任何东西。

%x

事务状态：当不在事务块中时是一个空字符串，在一个事务块中时是*，在一个失败的事务块中时是!，当事务状态是未判定时（例如因为没有连接）为?。

%l

当前语句中的行号，从1开始。

%digits

带有指定的八进制码的字符会被替换。

%:name:

psql变量name的值。详见变量。

%`command`

command的输出，类似于平常的“反引号”替换。

%[... %]

提示符可以包含终端控制字符，例如改变提示符文本的颜色、背景或者风格以及更改终端窗口标题的控制字符。为了让Readline的行编辑特性正确工作，这些不可打印的控制字符必须被包裹在%[和%]之间以指定它们是不可见的。在提示附中可以出现多个这样的标识对。例如：

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

会导致一个在兼容 VT100 的彩色终端上的粗体（1;）的、黑底黄字（33;40）的提示符。

要在你的提示符中插入一个百分号，可以写成%%。提示符 1 和 2 的默认提示是'%/%R%# '，提示符 3 的提示是'>> '。

> **注意**
>
> 这个特性是可耻地从tcsh抄袭过来的。

命令行编辑

为了方便的行编辑和检索，psql支持Readline库。psql退出时命令历史会被自动保存，而当psql启动时命令历史会被重新载入。psql也支持 tab 补全，不过补全逻辑绝不是一个SQL解析器。tab 补全产生的查询也可能会受其他 SQL 命令干扰，例如SET TRANSACTION ISOLATION LEVEL。如果出于某种原因不想用 tab 键补全，可以把下面的代码放在主目录下的名为.inputrc文件中关闭该特性：

```
$if psql
set disable-completion on
$endif
```

（这不是psql特性而是Readline的特性。进一步的细节请阅读它的文档。）

环境

COLUMNS

如果\pset columns为零，这个环境变量控制用于wrapped格式的宽度以及用来确定是否输出需要用到分页器或者切换到扩展自动模式中的垂直格式的宽度。

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数（见第 34.14 节。

PSQL_EDITOR
EDITOR
VISUAL

\e、\ef以及\ev命令所使用的编辑器。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，默认是使用Unix系统上的vi或者Windows系统上的notepad.exe。

PSQL_EDITOR_LINENUMBER_ARG

当\e、\ef或者\ev带有一个行号参数时，这个变量指定用于传递起始行号给用户编辑器的命令行参数。对于Emacs或者vi之类的编辑器，这个变量是一个加号。如果需要在选项名称和行号之间有空格，可以在该变量的值中包括一个结尾的空格。例如：

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

在 Unix 系统上默认是+（对应于默认编辑器vi，且对很多其他常见编辑器可用）。在 Windows 系统上没有默认值。

PSQL_HISTORY

命令历史文件的替代位置。波浪线（~）扩展会被执行。

PSQL_PAGER
PAGER

如果一个查询的结果在屏幕上放不下，它们会通过这个命令分页显示。典型的值是more或less。通过把PSQL_PAGER或PAGER设置为空字符串可以禁用分页器的使用，调整\pset命令与分页器相关的选项也能达到同样的效果。会按照列出的顺序检查这些变

量，第一个被设置的将被使用。如果都没有被设置，则大部分平台上默认使用more，但在Cygwin上使用less。

PSQLRC

用户的.psqlrc文件的替代位置。波浪线（~）扩展会被执行。

SHELL

被\!命令执行的命令。

TMPDIR

存储临时文件的目录。默认是/tmp。

和大部分其他PostgreSQL工具一样，这个工具也使用libpq所支持的环境变量（见第 34.14 节。

文件

psqlrc and ~/.psqlrc

如果没有-X选项，在连接到数据库后但在接收正常的命令之前，psql会尝试依次从系统级的启动文件（psqlrc）和用户的个人启动文件（~/.psqlrc）中读取并且执行命令。这些文件可以被用来设置客户端或者服务器，通常是一些\set和SET命令。

系统级的启动文件是psqlrc，它应该在安装好的PostgreSQL的“系统配置”目录中，最可靠的定位方法是运行pg_config --sysconfdir。默认情况下，这个目录将是../etc/（相对于包含PostgreSQL可执行文件的目录）。可以通过PGSYSCONFDIR环境变量显式地设置这个目录的名称。

用户个人的启动文件是.psqlrc，它应该在调用用户的主目录中。在 Windows 上，由于没有用户主目录的概念，个人的启动文件是%APPDATA%\postgresql\psqlrc.conf。用户启动文件的位置可以通过PSQLRC环境变量设置。

系统级和用户个人的启动文件都可以弄成是针对特定psql版本的，方法是在文件名后面加上一个横线以及PostgreSQL的主、次版本号，例如~/.psqlrc-9.2或者~/.psqlrc-9.2.5。版本最为匹配的文件会优先于不那么匹配的文件读入。

.psql_history

命令行历史被存储在文件~/.psql_history中，或者是 Windows 的文件%APPDATA%\postgresql\psql_history中。

历史文件的位置可以通过HISTFILE psql变量或者PSQL_HISTORY环境变量明确的设置。

注解

- psql和具有相同主版本或者更老的主版本服务器最为匹配。如果服务器的版本比psql本身要高，则反斜线命令尤其容易失败。不过，\d家族的反斜线命令应该可以和版本 7.4 之后的服务器一起使用，但服务器的版本不必比psql本身新。运行 SQL 命令并且显示查询结果的一般功能应该也能和具有更新主版本的服务器一起使用，但是并非在所有的情况下都能保证如此。

  如果你想用psql连接到多个具有不同主版本的服务器，推荐使用最新版本的psql。或者，你可以为每一个主版本保留一份psql拷贝，并且针对相应的服务器使用匹配的版本。但实际上，这种额外的麻烦是不必要的。

- 在PostgreSQL 9.6 之前，-c选项表示-X（--no-psqlrc），但现在不是这样了。

- 在PostgreSQL 8.4 之前，psql允许一个单字母反斜线命令的第一个参数直接写在该命令后面，中间不需要空格。现在则要求一些空格。

给 Windows 用户的注解

psql是一个“控制台应用”。由于 Windows 的控制台窗口使用的是一种和系统中其他应用不同的编码，在psql中使用 8 位字符时要特别注意。如果psql检测到一个有问题的控制台代码页，它将会在启动时警告你。要更改控制台代码页，有两件事是必要的：

- 输入cmd.exe /c chcp 1252可以设置代码页（1252 是适用于德语的一个代码页，请在这里替换成你的值）。如果正在使用 Cygwin，可以把这个命令放在/etc/profile中。
- 把控制台字体设置为Lucida Console，因为栅格字体无法与 ANSI 代码页一起使用。

示例

第一个例子展示了如何如何跨越多行输入一个命令。注意提示符的改变：

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

现在再看看表定义：

```
testdb=> \d my_table
             Table "public.my_table"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 first  | integer |           | not null | 0
 second | text    |           |          |
```

现在我们把提示符改一改：

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

假定已经用数据填充了这个表并且想看看其中的数据：

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

你可以用\pset命令以不同的方式显示表：

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
```

```
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

或者使用短命令：

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

如果需要，可以用\crosstabview命令以交叉表的形式显示查询结果：

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
     4 |     |     |       | t
(4 rows)
```

这第二个例子展示了表的“乘法”（连接），行按照序号降序排序且列按照独立的、升序的方式排序。

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as
 "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — 重索引一个PostgreSQL数据库

大纲

```
reindexdb [connection-option...] [option...] [ --schema | -S schema ] ... [ --table | -t table ] ... [ --index | -i index ] ... [dbname]

reindexdb [connection-option...] [option...] --all | -a

reindexdb [connection-option...] [option...] --system | -s [dbname]
```

描述

reindexdb是用于重建一个PostgreSQL数据库中索引的工具。

reindexdb是 SQL 命令REINDEX的一个包装器。在通过这个工具和其他方法访问服务器来重索引数据库之间没有实质性的区别。

选项

reindexdb接受下列命令行参数：

-a
--all

重索引所有数据库。

[-d] dbname
[--dbname=]dbname

指定要被重索引的数据库名。如果这没有被指定并且没有使用-a（或--all），数据库名可以从环境变量PGDATABASE中被读出。如果环境变量也没被设置，为该连接指定的用户名将被用作数据库名。

-e
--echo

回显reindexdb生成并发送到服务器的命令。

-i index
--index=index

只是重建index。可以通过写多个-i开关来重建多个索引。

-q
--quiet

不显示进度消息。

-s
--system

索引数据库的系统目录。

-S schema
--schema=schema

只对schema重建索引。 通过写多个-S开关可以指定多个要重建索引的模式。

-t table
--table=table

只索引table。可以通过写多个-t开关来重索引多个表。

-v
--verbose

在处理时打印详细信息。

-V
--version

打印reindexdb版本并退出。

-?
--help

显示有关reindexdb命令行参数的帮助并退出。

reindexdb也接受下列命令行参数用于连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制reindexdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，reindexdb将自动提示要求一个口令。但是，reindexdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

指定要连接到来发现哪些其他数据库应该被重索引的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在，将使用template1。

环境

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在REINDEX和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

注解

reindexdb可能需要多次连接到PostgreSQL服务器，每一次都会询问一个口令。在这种情况下使用一个~/.pgpass文件会更方便。详见第 34.15 节

例子

要重索引数据库test：

```
$ reindexdb test
```

要重索引名为abcd的数据库中的表foo和索引bar：

```
$ reindexdb --table foo --index bar abcd
```

参见

REINDEX

vacuumdb

vacuumdb — 对一个PostgreSQL数据库进行垃圾收集和分析

大纲

```
vacuumdb [connection-option...] [option...] [ --table | -t table [( column [,...] )] ] ... [dbname]

vacuumdb [connection-option...] [option...] --all | -a
```

描述

vacuumdb是用于清理一个PostgreSQL数据库的工具。vacuumdb也将产生由PostgreSQL查询优化器所使用的内部统计信息。

vacuumdb是 SQL 命令VACUUM的一个包装器。在通过这个工具和其他方法访问服务器来清理和分析数据库之间没有实质性的区别。

选项

vacuumdb接受下列命令行参数：

-a
--all

清理所有数据库。

[-d] dbname
[--dbname=]dbname

指定要被清理或分析的数据库名。如果没有被指定并且没有使用-a（或--all），数据库名将从环境变量PGDATABASE中读出。如果环境变量也没有设置，指定给该连接的用户名将用作数据库名。

-e
--echo

回显vacuumdb生成并发送给服务器的命令。

-f
--full

执行“完全”清理。

-F
--freeze

强有力地“冻结”元组。

-j njobs
--jobs=njobs

通过同时运行njobs 个命令来并行执行清理或者分析命令。这个选项会减少处理的时间， 但是它也会增加数据库服务器的负载。

vacuumdb将开启 njobs个到数据 库的连接，因此请确认你的max_connections 设置足够高以容纳所有的连接。

注意如果某些系统目录被并行处理，使用这种模式加上 -f（FULL）选项可能会导致 死锁失败。

-q
--quiet

不显示进度消息。

-t table [(column [,...])]
--table=table [(column [,...])]

只清理或分析table。列名只能和--analyze或--analyze-only选项一起被指定。通过写多个-t开关可以清理多个表。

> **提示**
>
> 如果你指定列，你可能必须转义来自 shell 的括号（见下面的例子）。

-v
--verbose

在处理期间打印详细信息。

-V
--version

打印vacuumdb版本并退出。

-z
--analyze

也计算优化器使用的统计信息。

-Z
--analyze-only

只计算优化器使用的统计信息（不清理）。

--analyze-in-stages

与--analyze-only相似，只计算优化器使用的统计信息（不做清理）。 使用不同的配置设置运行分析的几个（目前是 3个）阶段以更快地产生可用的统计信息。

这个选项对分析一个刚从转储恢复或者通过pg_upgrade得到 的数据库有用。这个选项将尝试尽可能快地创建一些统计信息来让该数据库可用，然后在后 续的阶段中产生完整的统计信息。

-?
--help

显示有关vacuumdb命令行参数的帮助并退出。

vacuumdb也接受下列命令行参数用于连接参数：

-h host
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username
--username=username

要作为哪个用户连接。

-w
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W
--password

强制vacuumdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，vacuumdb将自动提示要求一个口令。但是，vacuumdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

指定要连接到来发现哪些其他数据库应该被清理的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在，将使用template1。

环境

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

诊断

在有困难时，可以在VACUUM和psql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

注解

vacuumdb可能需要多次连接到PostgreSQL服务器，每次都询问一个口令。在这种情况下有一个~/.pgpass文件会很方便。详见第 34.15 节

例子

要清理数据库test：

```
$ vacuumdb test
```

要清理和为优化器分析一个名为bigdb的数据库：

```
$ vacuumdb --analyze bigdb
```

要清理在名为xyzzy的数据库中的一个表foo，并且为优化器分析该表的bar列：

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

参见

VACUUM

PostgreSQL 服务器应用

这一部分包含PostgreSQL服务器应用和支持工具的参考信息。这些命令只在数据库服务器所在的主机上运行才有用。其他工具程序在PostgreSQL 客户端应用中列出。

目录

initdb

initdb — 创建一个新的PostgreSQL数据库集簇

大纲

initdb [option...] [--pgdata | -D] directory

描述

initdb创建一个新的PostgreSQL数据库集簇。一个数据库集簇是由一个单一服务器实例管理的数据库的集合。

一个数据库集簇的创建包括创建存放数据库数据的目录、生成共享目录表（属于整个集簇而不是任何特定数据库的表）并且创建template1和postgres数据库。当你后来创建一个新的数据库时，任何在template1数据库中的东西都会被复制（因此，任何已安装在template1中的东西都会被自动地复制到后来创建的每一个数据库中）。postgres数据库是便于用户、工具和第三方应用使用的默认数据库。

尽管initdb将尝试创建指定的数据目录，它可能没有权限（如果想要的数据目录的父目录被根用户拥有）。要在这样一种设置中初始化，作为 root 创建一个空数据目录，然后使用chown将该目录赋予给数据库用户账户，再然后su成为该数据库用户并运行initdb。

initdb必须以将拥有该服务器进程的用户运行，因为该服务器需要访问initdb创建的文件和目录。因为该服务器不能作为 root 运行，你不能以 root 运行initdb（事实上它会拒绝这样做）。

由于安全原因，由initdb创建的新集簇默认将只能由集簇拥有者访问。--allow-group-access选项允许与集簇拥有者同组的任何用户读取集簇中的文件。这对非特权用户执行备份很有用。

initdb初始化该数据库集簇的默认区域和字符集编码。当一个数据库被创建时，其字符集编码、排序顺序（LC_COLLATE）和字符集类（LC_CTYPE，例如大写、小写、数字）可以被独立设置。initdb为template1数据库确定那些设置，它们将作为所有其他数据库的默认值。

要修改默认排序顺序或字符集类，使用--lc-collate和--lc-ctype选项。除C或POSIX之外的排序顺序也有性能罚值。由于这些原因，在运行initdb时选择正确的区域很重要。

余下的区域分类可以在服务器启动之后改变。你也可以使用--locale来为所有区域分类设置默认值，包括排序顺序和字符集类。所有服务器区域值（lc_*）可以通过SHOW ALL显示。详见第 23.1 节

要修改默认编码，使用--encoding。详见第 23.3 节

选项

-A authmethod
--auth=authmethod

这个选项为本地用户指定在pg_hba.conf中使用的默认认证方法（host和local行）。initdb将使用指定的认证方法为非复制连接以及复制连接填充pg_hba.conf项。

除非你信任你系统上的所有本地用户，不要使用trust。为了安装方便，trust是默认值。

--auth-host=authmethod

这个选项为通过 TCP/IP 连接的本地用户指定在pg_hba.conf中使用的认证方法（host行）。

--auth-local=authmethod

这个选项为通过 Unix 域套接字连接的本地用户指定在pg_hba.conf中使用的认证方法（local行）。

-D directory
--pgdata=directory

这个选项指定数据库集簇应该存放的目录。这是initdb要求的唯一信息，但是你可以通过设定PGDATA环境变量来避免书写它，这很方便因为之后数据库服务器（postgres）可以使用同一个变量来找到数据库目录。

-E encoding
--encoding=encoding

选择模板数据库的编码。这也将是后来创建的任何数据库的默认编码，除非你覆盖它。默认值来自于区域，或者如果该值不起作用则为SQL_ASCII。PostgreSQL服务器所支持的字符集在第 23.3.1 节描述。

-g
--allow-group-access

允许与集簇拥有者同组的用户读取initdb创建的所有集簇文件。 Windows会忽略此选项，因为它不支持POSIX样式的组权限。

-k
--data-checksums

在数据页面上使用校验码来帮助检测 I/O 系统造成的损坏。启用校验码将会引起显著的性能惩罚。这个选项只能在初始化期间被设置，并且以后不能修改。如果被设置，在所有数据库中会为所有对象计算校验码。

--locale=locale

为数据库集簇设置默认区域。如果这个选项没有被指定，该区域将从initdb所运行的环境中继承。区域支持在第 23.1 节描述。

--lc-collate=locale
--lc-ctype=locale
--lc-messages=locale
--lc-monetary=locale
--lc-numeric=locale
--lc-time=locale

和--locale相似，但是只在指定的分类中设置区域。

--no-locale

等效于--locale=C。

-N
--no-sync

默认情况下，initdb将等待所有文件被安全地写到磁盘。这个选项会导致initdb不等待就返回，这当然更快，但是也意味着一次后续的操作系统崩溃可能让数据目录损坏。通常，这个选项对测试有用，但是不应该在创建生产安装时使用。

--pwfile=filename

让initdb从一个文件读取数据库超级用户的口令。该文件的第一行被当作口令。

-S
--sync-only

安全地把所有数据库文件写入到磁盘并退出。这不会执行任何正常的initdb操作。

-T config
--text-search-config=config

设置默认的文本搜索配置。详见default_text_search_config。

-U username
--username=username

选择数据库超级用户的用户名。这个的默认值是实际运行initdb的用户的名称。超级用户的名字是什么真的不重要，但是你可以选择保留常用的名字postgres，即使操作系统的用户名不同。

-W
--pwprompt

让initdb提示要求为数据库超级用户给予一个口令。如果你没有计划使用口令认证，这就不重要。否则在你设置一个口令之前你就无法使用口令认证。

-X directory
--waldir=directory

这个选项指定预写式日志会被存储在哪个目录中。

--wal-segsize=size

设置WAL段尺寸，以兆字节为单位。这是WAL日志中每个文件的尺寸。默认的尺寸为16兆字节。该值必须位于2的1次幂和1024次幂（兆字节）之间。这个选项只能在初始化期间设置，并且之后不能更改。

调整这个值来控制WAL日志传送或者归档可能会有用。此外，在有大量WAL的数据库中，每个目录中数量巨大的WAL文件可能会成为性能和管理问题。增加WAL文件尺寸将会降低WAL文件的数量。

其他较少使用的选项：

-d
--debug

打印来自引导后端的调试输出以及普通大众不那么感兴趣的一些消息。引导后端被程序initdb用来创建目录表。这个选项会生成大量极端无聊的输出。

-L directory

指定initdb应从哪里寻找它的输入文件来初始化数据库集簇。这通常没有必要。如果你需要显式指定它们的位置，你应该被告知。

-n
--no-clean

默认情况下，当initdb确定有一个错误阻止它完整地创建数据库集簇，它会移除在它发现无法完成任务之前创建的任何文件。这个选项会抑制这种整理并且对调试有用。

其他选项：

-V
--version

打印initdb版本并退出。

-?
--help

显示有关initdb命令行参数的帮助并退出。

环境

PGDATA

指定数据库集簇应该被存放的目录，可以使用-D选项覆盖。

TZ

指定创建的数据集簇的默认时区。值应该是一个完整的时区名称（见第 8.5.3 节。

和大部分其他PostgreSQL工具相似，这个工具也使用libpq（见第 34.14 节支持的环境变量。

注解

initdb可以通过pg_ctl initdb被调用。

参见

pg_ctl, postgres

pg_archivecleanup

pg_archivecleanup — 清理PostgreSQL WAL 归档文件

大纲

```
pg_archivecleanup [option...] archivelocation oldestkeptwalfile
```

简介

pg_archivecleanup被设计用作 archive_cleanup_command在作为后备服务器运行（第 26.2 节时来清理 WAL 文件归档。 pg_archivecleanup也可以被用作一个单独的程序来清理 WAL 文件归档。

要配置一个后备服务器以使用pg_archivecleanup，把下面 的内容放在recovery.conf配置文件中：

```
archive_cleanup_command = 'pg_archivecleanup archivelocation %r'
```

其中archivelocation是要从中移除 WAL 段文件的目录。

当被用在archive_cleanup_command中时，所有逻辑上在 %r参数的值之前的 WAL 文件都将被从 archivelocation移除。这能最小化需要被保留的文件数量， 同时能保留崩溃后重启的能力。如果对于这台特定的后备服务器， archivelocation是一个短暂需要的区域，使用这个参数就是 合适的，但是当archivelocation要用作一个长期的 WAL 归档 区域或者当多个后备服务器正在从这个归档位置恢复时，使用这个参数就 不合适。

当被用作一个单独的程序时，所有逻辑上在oldestkeptwalfile 之前的 WAL 文件将被从archivelocation中移除。在这种模式 中，如果指定了.partial或者.backup文件名，则只有该文件前缀将被用作oldestkeptwalfile。这种对 .backup文件名的处理允许你移除所有在一个特定基础备份之前归 档的 WAL 文件而不出错。例如，下面的例子将移除所有比 WAL 文件名 000000010000003700000010老的文件：

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup假定 archivelocation是一个可读的目录并且对于服务器拥有者是可写的。

选项

pg_archivecleanup接受下列命令行参数：

-d

在stderr上打印很多调试日志输出。

-n

在stdout上打印将被移除的文件的名字（执行一次演习）。

-V
--version

打印pg_archivecleanup版本并退出。

-x extension

提供一个扩展名，在决定所有的文件 是否应该被删除之前，将从文件名中剥离这个扩展名。这通常有助于清理已经 存储期间被压缩过并且被压缩程序增加了一个扩展名的归档。例如： -x .gz。

-?
--help

显示pg_archivecleanup命令行参数的帮助并退出。

注解

在作为一个单独的工具时，pg_archivecleanup 被设计为与PostgreSQL 8.0 及其后的版本一起工作。如果 作为一个归档清理命令使用，则需要和PostgreSQL 9.0 及 其后的版本一起工作。

pg_archivecleanup以 C 写成并且具有很容易修改的 源代码，其中有特别指定的区域用于修改以符合你的需要

示例

在 Linux 或者 Unix 系统上，你可能会用：

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r
 2>>cleanup.log'
```

其中归档目录位于后备服务器上，这样archive_command通过 NFS 来访问它，但是文件对于后备服务器来说是本地的。这将会

- 在cleanup.log中产生调试输出
- 从归档目录中移除不再需要的文件

另见

pg_standby

pg_controldata

pg_controldata — 显示一个PostgreSQL数据库集簇的控制信息

大纲

pg_controldata [option] [[--pgdata | -D] datadir]

描述

pg_controldata打印在initdb期间初始化的信息，例如目录版本。它也显示关于预写式日志和检查点处理的信息。这种信息是集簇范围的，并且不针对任何一个数据库。

这个工具只能由初始化集簇的用户运行，因为它要求对数据目录的读访问。你可以在命令行中指定数据目录，或者使用环境变量PGDATA。这个工具支持选项-V和--version，它们打印pg_controldata版本并退出。它也支持选项-?和--help，它们输出支持的参数。

环境

PGDATA

默认的数据目录位置。

pg_ctl

pg_ctl — 初始化、启动、停止或控制一个PostgreSQL服务器

大纲

```
pg_ctl init[db] [-D datadir] [-s] [-o initdb-options]

pg_ctl start [-D datadir] [-l filename] [-W] [-t seconds] [-s] [-o options] [-p path] [-c]

pg_ctl stop [-D datadir] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t seconds] [-s]

pg_ctl restart [-D datadir] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t seconds] [-s] [-o options] [-c]

pg_ctl reload [-D datadir] [-s]

pg_ctl status [-D datadir]

pg_ctl promote [-D datadir] [-W] [-t seconds] [-s]

pg_ctl kill signal_name process_id
```

在Microsoft Windows上，还有：

```
pg_ctl register [-D datadir] [-N servicename] [-U username] [-P password] [-S a[uto] | d[emand] ] [-e source] [-W] [-t seconds] [-s] [-o options]

pg_ctl unregister [-N servicename]
```

描述

pg_ctl是一个用于初始化PostgreSQL数据库集簇，启动、停止或重启PostgreSQL数据库服务器（postgres），或者显示一个正在运行服务器的状态的工具。尽管服务器可以被手工启动，pg_ctl包装了重定向日志输出以及正确地从终端和进程组脱离等任务。它也提供了方便的选项用来控制关闭。

init或initdb模式会创建一个新的PostgreSQL数据库集簇，也就是将由一个单一服务器实例管理的数据库集合。这个模式调用initdb命令。详见initdb。

start模式启动一个新的服务器。该服务器被启动在后台，并且它的标准输出被附加到/dev/null（或 Windows 上的nul）。在 Unix 类系统上，默认情况下服务器的标准输出和标准错误被发送到pg_ctl的标准输出（不是标准错误）。pg_ctl的标准输出应该接着被重定向到一个文件或用管道导向另一个进程（例如日志轮转程序rotatelogs）。否则postgres将把它的输出写到控制终端（从后台）并且将不会离开 shell 的进程组。在 Windows 上，默认情况下服务器的标准输出和标准错误被发送到终端。这些默认行为可以使用-l追加服务器的输出到一个日志文件来改变。我们推荐使用-l或输出重定向。

stop模式关闭运行在指定数据目录中的服务器。对-m选项可以选择三种不同的关闭方法。“Smart”模式等待所有客户端断开连接以及任何在线备份结束。如果该服务器是热备，一旦所有的客户端已经断开连接，恢复和流复制将被终止。“Fast”模式（默认）不会等待客户端断开连接并且将终止进行中的在线备份。所有活动事务都被回滚并且客户端被强制断开连接，然后服务器被关闭。“Immediate”模式将立刻中止所有服务器进程，而不是做一次干净的关闭。这中选择将导致下一次重启时进行一次崩溃恢复。

restart模式实际上会先执行一个停止操作然后紧接着执行一个启动操作。这使得我们能够更改postgres的命令行选项，或者更改不通过重启服务器无法更改的配置文件选项。如果在

服务器启动期间在命令行上使用了相对路径，则restart可能会失败，除非pg_ctl被运行在与上次启动服务器相同的目录中。

reload模式简单地向postgres服务器进程发送一个SIGHUP信号，导致它重新读取它的配置文件（postgresql.conf、pg_hba.conf等）。这允许改变配置文件选项而无需一次完整的服务器重启来让改变生效。

status模式检查一个服务器是否运行在指定的数据目录中。如果有一个服务器正在运行，其PID和用来调用它的命令行选项将被显示。如果服务器没有在运行，pg_ctl将返回退出状态3。如果没有指定一个可以访问的数据目录，pg_ctl将返回退出状态 4。

promote模式命令运行在指定数据目录中的后备服务器结束后备模式并且开始读写操作。

kill模式向一个指定进程发送一个消息。这主要用于没有kill命令的Microsoft Windows。使用--help来查看受支持的信号名称列表。

register模式把PostgreSQL服务器注册为Microsoft Windows上的一个系统服务。-S选项允许选择服务启动类型，可以是“auto”（随系统自动启动）或“demand”（按需启动）。

unregister模式在Microsoft Windows上移除一个系统服务的注册。这会撤销register命令的效果。

选项

-c
--core-files

在可行的平台上尝试允许服务器崩溃产生核心文件，方法是提升在核心文件上的任何软性资源限制。这通过允许从一个失败的服务器进程中获得一个栈跟踪而有助于调试或诊断问题。

-D datadir
--pgdata=datadir

指定数据库配置文件的文件系统位置。如果这个选项被忽略，将使用环境变量PGDATA。

-l filename
--log=filename

追加服务器日志输出到filename。如果该文件不存在，它会被创建。umask被设置成077，这样默认情况下不允许其他用户访问该日志文件。

-m mode
--mode=mode

指定关闭模式。mode可以是smart、fast或immediate，或者这三者之一的第一个字母。如果这个选项被忽略，则fast是默认值。

-o options
--options=options

指定被直接传递给postgres命令的选项。-o可以被指定多次，所有给定的选项都会被传过去。

这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

-o initdb-options
--options=initdb-options

指定要直接传递给initdb命令的选项。-o可以被指定多次，所有给定的选项都会被传过去。

这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

-p path

指定postgres可执行程序的位置。默认情况下，postgres可执行程序可以从pg_ctl相同的目录得到，或者如果没有在那里找到，则在硬写的安装目录中获得。除非你正在做一些不同寻常的事并且得到错误说没有找到postgres可执行程序，这个选项不是必需的。

在init模式中，这个选项类似于指定了initdb可执行程序的位置。

-s
--silent

只打印错误，不打印信息性的消息。

-t seconds
--timeout=seconds

指定等待一个操作完成时要等待的最大秒数（见选项-w）。默认为PGCTLTIMEOUT环境变量的值，如果该环境变量没有设置则默认为60秒。

-V
--version

打印pg_ctl版本并退出。

-w
--wait

等待操作完成。模式start、stop、restart、promote以及register支持这个选项，并且对那些模式是默认的。

在等待时，pg_ctl会一遍又一遍地检查服务器的PID文件，在两次检查之间会休眠一小段时间。当PID文件指示该服务器已经做好准备接受连接时，启动操作被认为完成。当服务器移除PID文件时，关闭操作被认为完成。pg_ctl会基于启动或关闭的成功与否返回一个退出代码。

如果操作在超时时间（见选项-t）内未能完成，则pg_ctl会以一个非零退出状态退出。但是注意该操作可能会在后台继续进行并且最终取得成功。

-W
--no-wait

不等待操作完成。这是选项-w的对立面。

如果禁用等待，所请求的动作会被触发，但是不会有关于其成功与否的反馈。在这种情况下，可能必须用服务器日志文件或外部监控系统来检查该操作的进度以及成功与否。

在以前版本的PostgreSQL中，这是除stop模式之外的模式的默认选项。

-?
--help

显示有关pg_ctl命令行参数的帮助并退出。

如果一个指定的选项有效，但与选中的操作模式无关，则pg_ctl会忽略它。

用于 Windows 的选项

-e source

作为一个 Windows 服务运行时，pg_ctl用来 在事件日志中记录日志的事件源的名称。默认是PostgreSQL。 注意这只控制由pg_ctl本身发送的消息，一旦开

始， 服务器将使用event_source参数中指定的事件源。如果服务器在启动时很早（在该参数被设置前）就失败，它可能也会使用默认的事件源名称 PostgreSQL来记录。

-N servicename

要注册的系统服务的名称。这个名称将被用于服务名和显示名。默认是PostgreSQL。

-P password

用于运行该服务的用户的口令。

-S start-type

要注册的系统服务的启动类型。启动类型可以是auto、demand或者两者之一的第一个字母。如果这个选项被忽略，则auto是默认值。

-U username

用于运行该服务的用户的用户名。对于域用户，使用格式DOMAIN\username。

环境

PGCTLTIMEOUT

等待启动或者关闭完成时要等待的默认秒数限制。如果没有设置， 默认值是 60 秒。

PGDATA

默认的数据目录位置。

大部分的pg_ctl模式都要求知道数据目录的位置，因此-D选项是必需的，除非PGDATA被设置。

和大部分其他PostgreSQL工具相似，pg_ctl也使用libpq（见第 34.14 节支持的环境变量。

更多影响服务器的变量请见postgres。

文件

postmaster.pid

pg_ctl在数据目录中检查这个文件来判断服务器当前是否正在运行。

postmaster.opts

如果这个文件存在于数据目录中，pg_ctl（处于restart模式中）将把该文件的内容作为选项传递给postgres，除非通过-o选项进行了覆盖。这个文件的内容也会被显示在status模式中。

例子

启动服务器

要启动服务器并且等到服务器接受连接：

```
$ pg_ctl start
```

要使用端口 5433 启动服务器并且运行时不使用fsync：

```
$ pg_ctl -o "-F -p 5433" start
```

停止服务器

要停止服务器，使用：

```
$ pg_ctl stop
```

-m选项允许控制服务器如何关闭：

```
$ pg_ctl stop -m smart
```

重启服务器

重启服务器几乎等价于停止服务器并且再次启动它，不过pg_ctl默认会保存并重用被传递给之前的运行实例的命令行选项。要以和之前相同的选项重启服务器，使用：

```
$ pg_ctl restart
```

但是如果指定了-o，则会替换任何之前的选项。要使用端口 5433 重启并在重启时禁用fsync：

```
$ pg_ctl -o "-F -p 5433" restart
```

显示服务器状态

这里是pg_ctl状态输出的例子：

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B"
 "128"
```

第二行是在重启模式可能被调用的命令行。

参见

initdb, postgres

pg_resetwal

pg_resetwal — 重置一个PostgreSQL数据库集簇的预写式日志以及其他控制信息

大纲

pg_resetwal [--force | -f] [--dry-run | -n] [option...] [--pgdata | -D] datadir

描述

pg_resetwal会清除预写式日志（WAL）并且有选择地重置存储在pg_control文件中的一些其他控制信息。如果这些文件已经被损坏，某些时候就需要这个功能。当服务器由于这样的损坏而无法启动时，这只应该被用作最后的手段。

在运行这个命令之后，就可能可以启动服务器，但是记住数据库可能包含由于部分提交事务产生的不一致数据。你应当立刻转储你的数据、运行initdb并且重新载入。重新载入后，检查不一致并且根据需要修复之。

这个工具只能被安装服务器的用户运行，因为它要求对数据目录的读写访问。出于安全原因，你必须在命令行中指定数据目录。pg_resetwal不使用环境变量PGDATA。

如果pg_resetwal抱怨它无法为pg_control决定合法数据，你可以通过指定-f（强制）选项强制它继续。在这种情况下，丢失的数据将被替换为看似合理的值。可以期望大部分域是匹配的，但是下一个 OID、下一个事务 ID 和纪元、下一个多事务 ID 和偏移以及 WAL 开始位置域可能还是需要人工协助。这些域可以使用下面讨论的选项设置。如果你不能为所有这些域决定正确的值，-f还是可以被使用，但是恢复的数据库还是值得怀疑：一次立即的转储和重新载入是势在必行的。在你转储之前不要在该数据库中执行任何数据修改操作，因为任何这样的动作都可能使破坏更严重。

选项

-f
--force

即使pg_resetwal无法从pg_control中确定有效的数据（如前面所解释的），也强迫pg_resetwal继续运行。

-n
--dry-run

-n/--dry-run选项指示pg_resetwal打印从pg_control重构出来的值以及要被改变的值，然后不修改任何东西退出。这主要是一个调试工具，但是可以用来在允许pg_resetwal真正执行下去之前进行完整性检查。

-V
--version

显示版本信息然后退出。

-?
--help

显示帮助然后退出。

只有当pg_resetwal无法通过读取pg_control确定合适的值时，才需要下列选项。安全值可以按下文所述来确定。对于接收数字参数的值，可以使用前缀0x指定 16 进制值。

-c xid,xid
--commit-timestamp-ids=xid,xid

手工设置提交时间可以检索到的最老的和最新的事务 ID。

能检索到提交时间的最老事务 ID 的安全值（第一部分）可以通过在数据目录下pg_commit_ts目录中数字上最小的文件名来决定。反过来，能检索到提交时间的最新事务 ID 的安全值（第二部分）可以通过同一个目录中数字上最大的文件名来决定。文件名都是十六进制的。

-e xid_epoch
--epoch=xid_epoch

手工设置下一个事务 ID 的 epoch。

事务 ID 的 epoch 实际上并没有存储在数据库中的任何地方，除了被pg_resetwal设置在这个域中，所以只要关心的是数据库本身，任何值都可以用。你可能需要调整这个值来确保诸如Slony-I和Skytools之类的复制系统正确地工作 — 如果确实需要调整，应该可以从下游的复制数据库的状态中获得一个合适的值。

-l walfile
--next-wal-file=walfile

通过指定下一个WAL段文件名称来手工设置WAL开始位置。

下一个WAL段文件的名称应该比当前存在于数据目录下pg_wal目录中的任意 WAL 段文件名更大。这些名称也是十六进制的并且有三个部分。第一部分是“时间线 ID”并且通常应该被保持相同。例如，如果000000010000000320000004A是pg_wal中最大的项，则使用-l 000000010000000320000004B或更高的值。

注意在使用非默认WAL段尺寸时，WAL文件名中的数字与系统函数和系统视图报告的LSN不同。这个选项要的是WAL文件名而不是LSN。

> **注意**
>
> pg_resetwal本身查看pg_wal中的文件并选择一个超出最新现存文件名的默认-l设置。因此，只有当你知道 WAL 段文件当前不在pg_wal中时，或者当pg_wal的内容完全丢失时，才需要对-l的手工调整，例如一个离线归档中的项。

-m mxid,mxid
--multixact-ids=mxid,mxid

手工设置下一个和最老的多事务 ID。

确定下一个多事务 ID（第一部分）的安全值的方法：在数据目录下的pg_multixact/offsets目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 65536 (0x10000)。反过来，确定最老的多事务 ID（-m的第二部分）的方法：在同一个目录中查找最小的数字文件名并且乘以 65536。文件名是十六进制的数字，因此实现上述方法最简单的方式是以十六进制指定选项值并且追加四个零。

-o oid
--next-oid=oid

手工设置下一个 OID。

没有相对容易的方法来决定超过数据库中最大 OID 的下一个 OID。但幸运的是正确地得到下一个 OID 设置并不是决定性的。

-O mxoff
--multixact-offset=mxoff

手工设置下一个多事务偏移量。

确定安全值的方法：查找数据目录下pg_multixact/members目录中最大的数字文件名，然后在它的基础上加一并且乘以 52352 (0xCC80)。文件名是十六进制数字。没有像其他选项那样追加零的简单方法。

--wal-segsize=wal_segment_size

设置新的WAL段尺寸，以兆字节为单位。这个值必须被设为2的1次幂和1024次幂（兆字节）之间。更多信息请参考initdb的相同选项。

> **注意**
>
> 虽然pg_resetwal将把WAL起始地址设置成超过最新的现有WAL段文件，但一些段尺寸的改变可能导致之前的WAL文件名被重用。如果WAL文件名重叠会导致归档策略出现问题，推荐把-l和这个选项一起使用来手动设置WAL起始地址。

-x xid
--next-transaction-id=xid

手工设置下一个事务 ID。

确定安全值的方法：在数据目录下的pg_xact目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 1048576 (0x100000)。注意文件名是十六进制的数字。通常以十六进制的形式指定该选项值也是最容易的。例如，如果0011是pg_xact中的最大项，-x 0x1200000就可以（五个尾部的零就表示了前面说的乘数）。

注解

这个命令不能在服务器正在运行时被使用。如果在数据目录中发现一个服务器锁文件，pg_resetwal将拒绝启动。如果服务器崩溃那么一个锁文件可能会被留下，在那种情况下你能移除该锁文件来让pg_resetwal运行。但是在你那样做之前，再次确认没有服务器进程仍然存活。

pg_resetwal仅能在具有相同主版本的服务器上使用。

另见

pg_controldata

pg_rewind

pg_rewind — 把一个PostgreSQL数据目录与另一个从该目录中复制出来的数据目录同步

大纲

```
pg_rewind [option...] { -D | --target-pgdata } directory { --source-pgdata=directory | --source-server=connstr }
```

简介

pg_rewind是用于在集簇的时间线分叉以后，同步一个 PostgreSQL 集簇和同一集簇的另一份拷贝的工具。一种典型的场景是在失效后让一个旧的主服务器重新上线，同时有一个后备机跟随着新的主机。

其结果等效于把目标数据目录替换成源数据目录。关系文件中只有更改过的块才会被拷贝，所有其他的文件会被整个拷贝，包括配置文件。pg_rewind比起做一个新的基础备份或者rsync等工具的优势在于，pg_rewind不要求通读集簇中未更改的块。这使得它在数据库很大并且在集簇间只有小部分块不同时速度很快。

pg_rewind检查源集簇和目标集簇的时间线历史来判断它们在哪一点分叉，并且期望在目标集簇的pg_wal目录中找到 WAL 来返回到分叉点。分叉点可能会在目标时间线、源时间线或者它们的共同祖先上找到。在典型的失效场景中，目标集簇在分叉后很快就被关闭，这不是问题，但是如果目标集簇在分叉后已经运行了很长时间，旧的 WAL 文件可能已经不存在了。在这样的情况下，它们可以被手工从 WAL 归档复制到pg_wal目录，或者通过配置recovery.conf在启动时取得。pg_rewind的使用并不限于失效的场景，例如一个后备服务器可能被提升、运行一些写事务，然后被倒回再次成为一个后备。

当目标服务器在运行了pg_rewind之后第一次启动时，它将进入到恢复模式并且重放源服务器在分叉点之后产生的所有 WAL。当pg_rewind被运行时有某些 WAL 在源服务器上不可用，并且因此无法被pg_rewind会话所复制，则在目标服务器被启动时必须让这些 WAL 可用。这可以通过在目标数据目录中创建一个recovery.conf文件并且在其中使用一个适当的restore_command来实现。

pg_rewind要求目标服务器在postgresql.conf中启用了wal_log_hints选项，或者在用initdb初始化集簇时启用了数据校验。目前默认情况下这两者都没有被打开。full_page_writes也必须被设置为on，这是默认的。

> **警告**
>
> 如果在处理时pg_rewind失败，则目标的数据目录很可能不在可恢复的状态。在这种情况下，推荐创建一个新的备份。
>
> 如果pg_rewind发现它无法直接写入的文件，它将立刻失败。例如当源服务器和目标服务器为只读的SSL密钥及证书使用相同的文件映射，就会发生这种情况。如果在目标服务器上存在这样的文件，推荐在运行pg_rewind之前移除它们。在做了rewind之后，一些那样的文件可能已经被从源服务器拷贝，这样就有必要移除已经拷贝的数据并且恢复到rewind之前使用的链接集合。

选项

pg_rewind接受下列命令行参数：

-D directory
--target-pgdata=directory

这个选项指定要与源数据目录同步的目标数据目录。在运行pg_rewind之前目标服务器必须被干净地关闭。

--source-pgdata=directory

指定要和目标服务器同步的源服务器的数据目录的文件系统路径。这个选项要求源服务器必须被干净地关闭。

--source-server=connstr

指定一个 libpq 连接串用于连接要与目标服务器同步的源PostgreSQL服务器。该连接必须是一个具有超级用户访问权限的普通（非复制）连接。这个选项要求源服务器正在运行且不处于恢复模式。

-n
--dry-run

做除了实际修改目标目录之外的其他所有事情。

-P
--progress

启用进度报告。在从源集簇拷贝数据时，打开这个选项将会发送一个近似的进度报告。

--debug

打印冗长的调试输出，这主要对于调试pg_rewind的开发者有用。

-V
--version

显示版本信息然后退出。

-?
--help

显示帮助然后退出。

环境

在使用--source-server选项时，pg_rewind也使用libpq支持的环境变量（见第 34.14 节。

注解

如何工作

其基本思想是从源集簇拷贝所有文件系统级别的改变到目标集簇：

1. 以源集簇的时间线历史从目标集簇分叉出来的点之前的最后一个检查点为起点，扫描目标集簇的 WAL 日志。对于每一个 WAL 记录，读取每一个被动过的数据块。这会得到在目标集簇中从源集簇被分支出去以后所有被更改过的数据块列表。

2. 使用直接的文件系统访问（--source-pgdata）或者 SQL （--source-server），把所有那些更改过的块从源集簇拷贝到目标集簇。

3. 把所有其他诸如pg_xact和配置文件（除了关系文件之外所有的东西）从源集簇拷贝到目标集簇。与基础备份类似，在从源集簇拷贝的数据中，目

录pg_dynshmem/、pg_notify/、pg_replslot/、pg_serial/、pg_snapshots/
、pg_stat_tmp/以及pg_subtrans/的内容会被忽略。任何以pgsql_tmp开始的文件或目录都会被忽
略，backup_label、tablespace_map、pg_internal.init、postmaster.opts以及postmaster.pid也是这样。

4. 从源集簇应用 WAL，从失效处创建的检查点开始（严格来说，pg_rewind并不应用 WAL，它只是创建一个备份标签文件，该文件让PostgreSQL从那个检查点开始向前重放所有 WAL）。

pg_test_fsync

pg_test_fsync — 为PostgreSQL判断最快的 wal_sync_method

大纲

pg_test_fsync [option...]

简介

pg_test_fsync是想告诉你在特定的系统上，哪一种 wal_sync_method最快，还可以在发生认定的 I/O 问题时提供诊断信息。不过，pg_test_fsync 显示的区别可能不会在真实的数据库吞吐量上产生显著的区别，特别是由于 很多数据库服务器被它们的预写日志限制了速度。 pg_test_fsync为 wal_sync_method报告以微秒计的平均文件同步操作时间， 也能被用来提示用于优化commit_delay值的方法。

选项

pg_test_fsync接受下列命令行选项：

-f
--filename

指定要写入测试数据到其中的文件名。这个文件必须位于和 pg_wal目录所在或者将被放置的同一个文件系统中（ pg_wal包含WAL文件）。默认是当前 目录中的pg_test_fsync.out。

-s
--secs-per-test

指定每次测试的秒数。每个测试的时间越长，测试的精度就越高，但是 它需要更多时间来运行。默认是 5 秒，这允许程序在 2 分钟以内完成。

-V
--version

打印pg_test_fsync版本并且退出。

-?
--help

显示有关pg_test_fsync命令行参数的帮助并且退出。

另见

postgres

pg_test_timing

pg_test_timing — 度量计时开销

大纲

```
pg_test_timing [option...]
```

描述

pg_test_timing是一种度量在你的系统上计时开销以及确认系统时间绝不会回退的工具。收集计时数据很慢的系统会给出不太准确的EXPLAIN ANALYZE结果。

选项

pg_test_timing接受下列命令行选项：

-d duration
--duration=duration

指定测试的持续时间，以秒计。更长的持续时间会给出更好一些的精确度，并且更可能发现系统时钟回退的问题。默认的测试持续时间是 3 秒。

-V
--version

打印pg_test_timing版本并退出。

-?
--help

显示有关pg_test_timing的命令行参数，然后退出。

用法

结果解读

好的结果将显示大部分（>90%）的单个计时调用用时都小于 1 微秒。每次循环的平均开销将会更低，低于 100 纳秒。下面的例子来自于一台使用了一份 TSC 时钟源码的 Intel i7-860 系统，它展示了非常好的性能：

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  &lt; us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

注意每次循环时间和柱状图用的单位是不同的。循环的解析度可以在几个纳秒（ns），而单个计时调用只能解析到一个微秒（us）。

度量执行器计时开销

当查询执行器使用EXPLAIN ANALYZE运行一个语句时，单个操作会被计时，总结也会被显示。你的系统的负荷可以通过使用psql程序计数行来检查：

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

i7-860 系统测到运行该计数查询用了 9.8 ms 而EXPLAIN ANALYZE版本则需要 16.6 ms，每次处理都在 100,000 行上进行。6.8 ms 的差别意味着在每行上的计时负荷是 68 ns，大概是 pg_test_timing 估计的两倍。即使这样相对少量的负荷也造成了带有计时的计数语句耗时多出了 70%。在更大量的查询上，计时开销带来的问题不会有这么明显。

改变时间来源Changing time sources

在一些较新的 Linux 系统上，可以在任何时候更改用来收集计时数据的时钟来源。第二个例子显示了在上述快速结果的相同系统上切换到较慢的 acpi_pm 时间源可能带来的降速：

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/
current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  &lt; us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

在这种配置中，上面的例子EXPLAIN ANALYZE用了 115.9 ms。其中有 1061 ns 的计时开销，还是用这个工具直接度量结果的一个小倍数。这么多的计时开销意味着实际的查询本身只占了时间的一个很小的分数，大部分的时间都耗在了计时所需的管理开销上。在这种配置中，任何涉及到很多计时操作的EXPLAIN ANALYZE都会受到计时开销的显著影响。

FreeBSD 也允许即时更改时间源，并且它会记录在启动期间有关计时器选择的信息：

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

其他系统可能只允许在启动时设定时间源。在旧的 Linux 系统上，“clock”内核设置是做这类更改的唯一方法。并且即使在一些更近的系统上，对于一个时钟源你将只能看到唯一的选项 "jiffies"。Jiffies 是老的 Linux 软件时钟实现，当有足够快的计时硬件支持时，它能够具有很好的解析度，就像在这个例子中：

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  &lt; us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

时钟硬件和计时准确性

收集准确的计时信息在计算机上通常是使用具有不同精度的时钟硬件完成的。使用一些硬件，操作系统能几乎直接把系统时钟时间传递给程序。一个系统时钟也可以得自于一块简单地提供计时中断、在某个已知时间区间内的周期性滴答的芯片。在两种情况中，操作系统内核提供一个隐藏这些细节的时钟源。但是时钟源的精确度以及能多快返回结果会根据底层硬件而变化。

不精确的计时能够导致系统不稳定性。对任何时钟源的更改都要仔细地测试。操作系统默认是有时会更倾向于可靠性而不是最好的精确性。并且如果你在使用一个虚拟机器，应查看与之兼容的推荐时间源。在模拟计时器时虚拟硬件面临着额外的困难，并且提供商常常会建议每个操作系统的设置。

时间戳计数器（TSC）时钟源是当前一代 CPU 上最精确的一种。当操作系统支持 TSC 并且 TSC 可靠时，它是跟踪系统时间更好的方式。有多种方式会使 TSC 无法提供准确的计时源，这会让它不可靠。旧的系统能有一种基于 CPU 温度变化的 TSC 时钟，这让它不能用于计时。尝试在一些就的多核 CPU 上使用 TSC 可能在多个核心之间给出不一致的时间报告。这可能导致时间倒退，这个程序会检查这种问题。并且即使最新的系统，在非常激进的节能配置下也可能无法提供准确的 TSC 计时。

更新的操作系统可能检查已知的 TSC 问题并且当它们被发现时切换到一种更慢、更稳定的时钟源。如果你的系统支持 TSC 时间但是并不默认使用它，很可能是由于某种充分的理由才禁用它。某些操作系统可能无法正确地检测所有可能的问题，或者即便在知道 TSC 不精确的情况下也允许使用 TSC。

如果系统上有高精度事件计时器（HPET）并且 TSC 不准确，该系统将会更喜欢 HPET 计时器。计时器芯片本身是可编程的，最高允许 100 纳米的解析度，但是在你的系统时钟中可能见不到那么高的准确度。

高级配置和电源接口（ACPI）提供了一种电源管理（PM）计时器，Linux 把它称之为 acpi_pm。得自于 acpi_pm 的时钟最好时将能提供 300 纳秒的解析度。

在旧的 PC 硬件上使用的计时器包括 8254 可编程区间计时器（PIT）、实时时钟（RTC）、高级可编程中断控制器（APIC）计时器以及 Cyclone 计时器。这些计时器是以毫秒解析度为目标的。

参见

EXPLAIN

pg_upgrade

pg_upgrade — 升级PostgreSQL服务器实例

大纲

pg_upgrade -b oldbindir -B newbindir -d oldconfigdir -D newconfigdir [option...]

描述

pg_upgrade（之前被称为pg_migrator） 允许存储在PostgreSQL数据文件中的数据被升级到一个较晚 的PostgreSQL主版本而无需进行主版本升级（例如从9.5.8到9.6.4或者从10.7到11.2）通常所需的数据转储/重载。 对于次版本升级（例如从9.6.2到9.6.3或者从10.1到10.2）则不需要这个程序。

主 PostgreSQL 发行通常会加入新的特性，这些新特性常常会更改系统表的 布局，但是内部数据存储格式很少会改变。pg_upgrade 使用这一事实来通过创建新系统表并且重用旧的用户数据文件来执行快速升级。 如果一个未来的主发行没有把数据存储格式改得让旧数据格式不可读取，这类 升级就用不上pg_upgrade（社区将尝试避免这类情况）。

pg_upgrade会尽力（例如通过检查兼容的编译时设 置）确保新旧集簇在二进制上也是兼容的，包括 32/64 位二进制。保持 外部模块也是二进制兼容的也很重要，不过 pg_upgrade无法检查这一点。

pg_upgrade 支持从 8.4.X 及其后版本升级到当前的 PostgreSQL主发布，包括快照和 beta 发布。

选项

pg_upgrade接受下列命令行参数：

-b bindir
--old-bindir=bindir

旧的 PostgreSQL 可执行文件目录； 环境变量PGBINOLD

-B bindir
--new-bindir=bindir

新的 PostgreSQL 可执行文件目录； 环境变量PGBINNEW

-c
--check

只检查集簇，不更改任何数据

-d datadir
--old-datadir=datadir

旧的集簇数据目录；环境变量 PGDATAOLD

-D datadir
--new-datadir=datadir

新的集簇数据目录；环境变量 PGDATANEW

-j
--jobs

要同时使用的进程或线程数

-k
--link

使用硬链接来代替将文件拷贝到新集簇

-o options
--old-options options

直接传送给旧 postgres命令的选项，多个选项可以追加在后面

-O options
--new-options options

直接传送给新 postgres命令的选项，多个选项可以追加在后面

-p port
--old-port=port

旧的集簇端口号；环境变量 PGPORTOLD

-P port
--new-port=port

新的集簇端口号；环境变量 PGPORTNEW

-r
--retain

即使在成功完成后也保留 SQL 和日志文件

-U username
--username=username

集簇的安装用户名；环境变量 PGUSER

-v
--verbose

启用详细的内部日志

-V
--version

显示版本信息，然后退出

-?
--help

显示帮助，然后退出

使用

下面是用pg_upgrade执行一次升级的步骤：

1. 移动旧集簇（可选）

   如果你在使用一个与版本相关的安装目录（例如 /opt/PostgreSQL/11），你就不需要移动旧的集簇。 图形化的安装程序会使用版本相关的安装目录。

   如果你的安装目录不是版本相关的（例如/usr/local/pgsql）， 就有必要移动当前的 PostgreSQL 安装目录，以免它干扰新的 PostgreSQL安装。一旦当前的 PostgreSQL服务器被关闭，就可以安全地重命名 PostgreSQL 安装目录。假设旧目录是 /usr/local/pgsql，你可以这样：

mv /usr/local/pgsql /usr/local/pgsql.old

来重命名该目录。

2. 对于源码安装，编译新版本

用兼容旧集簇的configure标记编译新的 PostgreSQL 源码。在开始升级之前，pg_upgrade 将检查pg_controldata来确保所有设置都是兼容的。

3. 安装新的 PostgreSQL 二进制文件

安装新服务器的二进制文件和支持文件。pg_upgrade 会被包含在默认的安装中。

对于源码安装，如果你希望把新服务器安装在一个自定义的位置， 可以使用prefix变量：

```
make prefix=/usr/local/pgsql.new install
```

4. 初始化新的 PostgreSQL 集簇

使用initdb初始化新集簇。这里也要使用与 旧集簇相兼容的initdb标志。许多预编译的 安装程序会自动做这个步骤。这里没有必要启动新集簇。

5. 安装自定义的共享对象文件

把旧集簇使用的所有自定义共享对象文件（或者 DLL）安装到新集簇中， 例如pgcrypto.so，不管它们是来自于 contrib还是某些其他源码。不要安装模式定义（例如CREATE EXTENSION pgcrypto），因为这些将会从旧集簇升级得到。 还有，任何自定义的全文搜索文件（词典、同义词、辞典、停用词）也必须 被复制到新集簇中。

6. 调整认证

pg_upgrade将会多次连接到旧服务器和新服务器，因此 你可能想要在pg_hba.conf中把认证设置成 peer或者使用一个~/.pgpass文件（见 第 34.15 节。

7. 停止两个服务器

确认两个数据库服务器都被停止使用，例如在 Unix 上可以：

```
pg_ctl -D /opt/PostgreSQL/9.6 stop
pg_ctl -D /opt/PostgreSQL/11 stop
```

或者在 Windows 上使用正确的服务名：

```
NET STOP postgresql-9.6
NET STOP postgresql-11
```

直到后面的步骤之前，流复制和日志传送后备服务器可以保持运行。

8. 为后备服务器升级做准备

如果正在使用小节步骤 10中给出的方法升级后备服务器，请对旧的主集簇和后备集簇运行pg_controldata以验证旧的后备服务器已经完全追上。验证“Latest checkpoint location”值在所有集簇中都匹配（如果旧后备服务器在旧的主服务器之前被关闭或者如果旧的后备服务器仍在运行，则将会出现失配）。此外，在新的主集簇上的postgresql.conf文件中把wal_level改为replica。

9. 运行 pg_upgrade

   总是应该运行新服务器而不是旧服务器的pg_upgrade二进制文件。 pg_upgrade要求制定新旧集簇的数据和可执行文件（ bin）目录。你也可以指定用户和端口值，以及你是否想要用链接来 取代默认的复制行为对数据文件进行处理。

   如果你使用链接模式，升级将会快很多（不需要文件拷贝）并且将使用 更少的磁盘空间，但是在升级后一旦启动新集簇，旧集簇就无法被访问。 链接模式也要求新旧集簇数据目录位于同一个文件系统中（表空间和 pg_wal可以在不同的文件系统中）。完整的选项列表 可见pg_upgrade --help。

   --jobs选项允许多个 CPU 核心被用来复制/链接文件以及 并行地转储和重载数据库模式。这个选项一个比较好的值是 CPU 核心数 和表空间数的最大值。这个选项可以显著地减少升级运行在一台多处理 器机器上的多数据库服务器的时间。

   对于 Windows 用户，你必须以一个超级账号登录，并且以 postgres用户启动一个 shell 并且设置正确的路径：

   ```
   RUNAS /USER:postgres "CMD.EXE"
   SET PATH=%PATH%;C:\Program Files\PostgreSQL\11\bin;
   ```

   并且用带引号的目录运行pg_upgrade，例如：

   ```
   pg_upgrade.exe
           --old-datadir "C:/Program Files/PostgreSQL/9.6/data"
           --new-datadir "C:/Program Files/PostgreSQL/11/data"
           --old-bindir "C:/Program Files/PostgreSQL/9.6/bin"
           --new-bindir "C:/Program Files/PostgreSQL/11/bin"
   ```

   一旦启动，pg_upgrade将验证两个集簇是否兼容并且 执行升级。你可以使用pg_upgrade --check来只执行检查， 这种模式即使在旧服务器还在运行时也能使用。pg_upgrade --check也将列出任何在更新后需要做的手工 调整。如果你将要使用链接模式，你应该使用--link选项和 --check一起来启用链接模式相关的检查。 pg_upgrade要求在当前目录中的写权限。

   显然，没有人可以在升级期间访问这些集簇。pg_upgrade 默认会在端口 50432 上运行服务器来避免意外的客户端连接。在做升级时， 可以对两个集簇使用相同的端口号，因为新旧集簇不会在同时被运行。不过， 在检查一个旧的运行中服务器时，新旧端口号必须不同。

   如果在恢复数据库模式时发生错误，pg_upgrade将会退出 并且你必须按照下文步骤16中所说的恢复 旧集簇。要再次尝试pg_upgrade，你将需要修改 旧集簇，这样pg_upgrade 模式会成功恢复。如果问题是一个 contrib模块， 你可能需要从旧集簇中卸载该模块并且在升级后重新把它安装在新集簇中，不过 这样做的前提是该模块没有被用来存储用户数据。

10. 升级流复制和日志传送后备服务器

    如果使用链接模式并且有流复制（见第 26.2.5 节或者日志 传送（见第 26.2 节后备服务器，你可以遵照下面的 步骤对它们进行快速的升级。你将不用在这些后备服务器上运行 pg_upgrade，而是在主服务器上运行rsync。 到这里还不要启动任何服务器。

    如果你没有使用链接模式、没有或不想使用rsync或者想用一种更容易的解决方案，请跳过这一节中的过程并且在pg_upgrade完成并且新的主集簇开始运行后重建后备服务器。

a. 在后备服务器上安装新的 PostgreSQL 二进制文件

确保新的二进制和支持文件被安装在所有后备服务器上。

b. 确保不存在新的后备机数据目录

确保新的后备机数据目录不存在或者为空。如果 运行过initdb，请删除后备服务器的新数据目录。

c. 安装自定义共享对象文件

在新的后备机上安装和新的主集簇中相同的自定义共享对象文件。

d. 停止后备服务器

如果后备服务器仍在运行，现在使用上述的指令停止它们。

e. 保存配置文件

从旧后备机的配置目录保存任何需要保留的配置文件，例如 postgresql.conf、recovery.conf， 因为这些文件在下一步中会被重写或者移除。

f. 运行rsync

在使用链接模式时，后备服务器可以使用rsync快速升级。为了实现这一点，在主服务器上一个高于新旧数据库集簇目录的目录中为每个后备服务器运行这个命令：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive
 old_cluster new_cluster remote_dir
```

其中old_cluster和new_cluster是相对于主服务器上的当前目录的，而remote_dir是后备服务器上高于新旧集簇目录的一个目录。在主服务器和后备服务器上指定目录之下的目录结构必须匹配。指定远程目录的详细情况请参考rsync的手册，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /
opt/PostgreSQL/9.5 \
      /opt/PostgreSQL/9.6 standby.example.com:/opt/PostgreSQL
```

可以使用rsync的--dry-run选项验证该命令将做的事情。虽然在主服务器上必须为至少一台后备运行rsync，可以在一台已经升级过的后备服务器上运行rsync来升级其他的后备服务器，只要已升级的后备服务器还没有被启动。

这个命令所做的事情是记录由pg_upgrade的链接模式创建的链接，它们连接主服务器上新旧集簇中的文件。该命令接下来在后备服务器的旧集簇中寻找匹配的文件并且为它们在该后备的新集簇中创建链接。主服务器上没有被链接的文件会被从主服务器拷贝到后备服务器（通常都很小）。这提供了快速的后备服务器升级。不幸地是，rsync会不必要地拷贝与临时表和不做日志表相关的文件，因为通常在后备服务器上不存在这些文件。

如果有表空间，你将需要为每个表空间目录运行一个类似的rsync命令，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /
vol1/pg_tblsp/PG_9.5_201510051 \
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/vol1/pg_tblsp
```

如果你已经把pg_wal放在数据目录外面，也必须在那些目录上运行rsync。

g. 配置流复制和日志传送后备服务器

为日志传送配置服务器（不需要运行pg_start_backup() 以及pg_stop_backup()或者做文件系统备份，因为从属机 仍在与主机同步）。

11. 恢复 pg_hba.conf

如果你修改了pg_hba.conf，则要将其恢复到原始的设置。 也可能需要调整新集簇中的其他配置文件（例如 postgresql.conf）来匹配旧集簇。

12. 启动新服务器

现在可以安全地启动新的服务器，并且可以接着启动任何 rsync过的后备服务器。

13. 升级后处理

如果需要做任何升级后处理，pg_upgrade 将在完成后发出警告。它也将 生成必须由管理员运行的脚本文件。这些脚本文件将连接到每一个需要做 升级后处理的数据库。每一个脚本应该这样运行：

```
psql --username=postgres --file=script.sql postgres
```

这些脚本可以以任何顺序运行并且在运行之后立即删除。

> **小心**
>
> 通常在重建脚本运行完成之前访问重建脚本中引用的表是不安全的，这样做 可能会得到不正确的结果或者很差的性能。没有在重建脚本中引用的表可以 随时被访问。

14. 统计信息

由于pg_upgrade并未传输优化器统计信息，在升级的尾声 你将被指示运行一个命令来生成这些信息。你可能需要设置连接参数来匹配你 的新集簇。

15. 删除旧集簇

一旦你对升级表示满意，你就可以通过运行 pg_upgrade完成时提到的脚本来删除旧集簇的 数据目录（如果在旧数据目录中有用户定义的表空间就不可能实现自动删除）。你也可以删除旧安装目录（例如bin、share）。

16. 恢复到旧集簇

在运行pg_upgrade之后，如果你希望恢复到 旧集簇，有几个选项：

- 如果你运行了带有--check的 pg_upgrade，则没有对旧集簇做修改并且 可以在任何时候重新使用它。
- 如果你运行了带有--link的 pg_upgrade，数据文件在新旧集簇之间 共享。如果你开启了新集簇，并且新服务器已经对这些共享文件做了写 入，那么使用旧集簇就不安全。
- 如果你运行了不带 --link的 pg_upgrade或者没有启动新服务器， 旧集簇还没有被修改，如果已经执行了链接，会在 $PGDATA/global/pg_control后追加一个 .old后缀。要重用旧集簇，可以从 $PGDATA/global/pg_control移除 .old后缀，然后你就能重启旧集簇了。

注解

pg_upgrade不支持对某些数据库的升级，此类 数据库包含以下reg*开头的 OID 引用的系统数据类型： regproc、regprocedure、 regoper、regoperator、 regconfig以及regdictionary（ regtype可以被升级）。

如果失败、重建和重索引会影响你的安装，pg_upgrade 将会报告这些情况。用来重建表和索引的升级后脚本将会自动被建立。 如果你正在尝试自动升级很多集簇，你应该发现具有相同数据库模式的集簇 对所有集簇升级都要求同样的升级后步骤，这是因为升级后步骤是基于数据 库模式而不是用户数据。

对于部署测试，创建一个只有模式的旧集簇副本，在其中插入假数据并且升级。

如果你在升级一个PostgreSQL 9.2 之前的集簇，并且 它使用一个只有配置文件的目录，你必须向pg_upgrade 传递真正的数据目录位置，并且把配置目录位置传递给服务器，例如 -d /real-data-directory -o '-D /configuration-directory'。

如果正在使用的一个 9.1 之前的旧服务器用的是一个非默认Unix 域套接字目录 或者使用的默认值不同于新集簇的默认值，请把PGHOST设置为 指向旧服务器的套接字位置（这与Windows 无关）。

如果你想要使用链接模式并且你不想让你的旧集簇在新集簇启动时被修改， 可以复制一份旧集簇并且在副本上以链接模式进行升级。要创建旧集簇的一 份合法拷贝，可以在服务器运行时使用rsync创建旧集簇的 一份脏拷贝，然后关闭旧服务器并且再次运行rsync --checksum 把更改更新到该拷贝以让其一致（--checksum是必要的，因为 rsync在判断文件修改时间的更改时的精度只能到秒级）。如 第 25.3.3 节所述，你可能想要排除 一些文件，例如postmaster.pid。如果你的文件系统支持文 件系统快照或者 copy-on-write 文件副本，你可以使用它们来创建旧集簇和 表空间的一个备份，不过快照和副本必须被同时创建或者在数据库服务器关闭 期间被创建。

另见

initdb, pg_ctl, pg_dump, postgres

pg_verify_checksums

pg_verify_checksums — verify data checksums in a PostgreSQL database cluster

大纲

pg_verify_checksums [option...] [[-D | --pgdata] datadir]

简介

pg_verify_checksums在一个PostgreSQL集簇中验证数据的校验和。在运行pg_verify_checksums之前，服务器必须被干净地关闭。如果没有校验和错误，则这个程序的退出状态为零，否则为非零。

选项

有下列命令行选项可用：

-D directory
--pgdata=directory

指定存放数据库集簇的目录。

-v
--verbose

启用详细输出。列出所有被检查的文件。

-r relfilenode

仅验证指定relfilenode对应的关系中的校验和。

-V
--version

打印pg_verify_checksums版本并且退出。

-?
--help

显示有关Show help about pg_verify_checksums命令行参数的帮助，并且退出。

环境

PGDATA

指定存放该数据库集簇的目录，可以用-D选项重载。

pg_waldump

pg_waldump — 以人类可读的形式显示一个PostgreSQL 数据库集簇的预写式日志

大纲

pg_waldump [option...] [startseg [endseg]]

简介

pg_waldump显示预写式日志（WAL），它主要 用于调试或者教育目的。

这个工具只能由安装该服务器的用户运行，因为它要求对数据目录的只读访问。

选项

下列命令行选项控制输出的位置和格式：

startseg

从指定的日志段文件开始读取。这也隐含地决定了要搜索文件的路径以及 要使用的时间线。

endseg

在读取指定的日志段文件后停止。

-b
--bkp-details

输出有关备份块的细节。

-e end
--end=end

在指定的WAL位置停止读取，而不是一直读取到日志流的末尾。

-f
--follow

在到达可用 WAL 的末尾之后，保持每秒轮询一次是否有新的 WAL 出现。

-n limit
--limit=limit

显示指定数量的记录，然后停止。

-p path
--path=path

指定搜索日志段文件的目录或包含这些文件的包含pg_wal子目录的目录。 缺省值是在当前目录中搜索，当前目录的pg_wal子目录和 PGDATA的pg_wal子目录。

-r rmgr
--rmgr=rmgr

只显示由指定资源管理器生成的记录。如果把list作为资源管理器名称 传递给这个选项，则打印出可用资源管理器名称的列表然后退出。

-s start
--start=start

要从哪个WAL位置开始读取。默认是从找到的最早的文件的第一个可用日志记录开始。

-t timeline
--timeline=timeline

要从哪个时间线读取日志记录。默认是使用startseg（如果指定） 中的值，否则默认为1。

-V
--version

打印pg_waldump版本并且退出。

-x xid
--xid=xid

只显示用给定事务 ID 标记的记录。

-z
--stats[=record]

显示概括统计信息（记录的数量和尺寸以及全页镜像）而不是显示 每个记录。可以选择针对每个记录生成统计信息，而不是针对每个 资源管理器生成。

-?
--help

显示有关pg_waldump命令行参数的帮助并且退出。

注解

当服务器正在运行时可能会给出错误的结果。

只有指定的时间线 会被显示（如果没有指定，则显示默认时间线）。 其他时间线上的记录会被忽略。

pg_waldump不能读取具有后缀.partial 的 WAL 文件。如果需要读取那些文件，需要从文件名中移除 .partial后缀。

另见

第 30.5 节

postgres

postgres — PostgreSQL数据库服务器

大纲

postgres [选项...]

描述

postgres是PostgreSQL数据库服务器。一个客户端应用为了能访问一个数据库，它会（通过一个网络或者本地）连接到一个运行着的postgres实例。该postgres实例接着会开始一个独立的服务器进程来处理该连接。

一个postgres实例总是管理正好一个数据库集簇的数据。一个数据库集簇是一个数据库的集合，它们被存储在一个共同的文件系统位置（“数据区”）上。 一个系统上可以同时运行多个postgres实例，只要它们使用不同的数据区和不同的通信端口（见下文）。postgres启动时需要知道数据区的位置，该位置必须通过-D选项或PGDATA环境变量指定，对此是没有默认值的。通常，-D或PGDATA会直接指向由initdb创建的数据区目录。其他可能的文件布局在第 19.2 节讨论。

默认情况下，postgres会在前台启动并将日志消息打印到标准错误流。但在实际应用中，postgres应当作为一个后台进程启动，而且多数是在系统启动时自动启动。

postgres还能在单用户模式中被调用。这种模式的主要用途是在启动过程中由initdb使用。有时候它也被用于调试或者灾难性恢复。注意，运行一个单用户模式服务器并不真地适合调试服务器，因为不会发生实际的进程间通信和锁定。当从 shell 中调用单用户模式时，用户可以输入查询并且结果会被以一种更适合开发者阅读（不适合普通用户）的形式打印在屏幕上。在单用户模式中，会话用户将被设置为 ID 为 1 的用户，并且这个用户会被隐式地赋予超级用户权限。该用户不必实际存在，因此单用户模式运行可以被用来对某些意外损坏的系统目录进行手工恢复。

选项

postgres接受下列命令行参数。关于这些选项的详细讨论请参考第 19 章你也可以通过设置一个配置文件来减少键入大部分这些选项。有些（安全）选项还可以从连接的客户端以一种与应用相关只应用于会话的方法设置。例如，如果设置了PGOPTIONS环境变量，那么基于libpq的客户端将都把那个字符串传递给服务器，它将被服务器解释成postgres命令行选项。

通用选项

-B nbuffers

设置被服务器进程使用的共享内存缓冲区数量。这个参数的默认值是initdb自动选择的。指定这个选项等效于设置shared_buffers配置参数。

-c name=value

设置一个命名的运行时参数。PostgreSQL支持的配置参数在第 19 章描述。大多数其它命令行选项实际上都是这种参数赋值的短形式。-c可以出现多次用于设置多个参数。

-C name

打印命名运行时参数的值，并且退出（详见上面的-c选项）。这可以被用在一个运行服务器上，并且从postgresql.conf中返回值，这些值可能被在这次调用中的任何参数修改过。它并不反映集簇启动时提供的参数。

这个选项用于与一个服务器实例交互的其他程序来查询配置参数值，例如pg_ctl。面向用户的应用应该使用SHOW或者pg_settings视图。

-d debug-level

设置调试级别。数值设置得越高，写到服务器日志的调试输出就越多。取值范围是从 1 到 5。还可以针对某个特定会话使用-d 0来阻止父postgres进程的服务器日志级别被传播到这个会话。

-D datadir

指定数据库配置文件的文件系统位置。详见第 19.2 节

-e

把默认日期风格设置为“European”，也就是输入日期域的顺序是DMY。这也导致在一些日期输出格式中把日打印在月之前。详见第 8.5 节

-F

禁用fsync调用以提高性能，但是要冒系统崩溃时数据损坏的风险。指定这个选项等效于禁用fsync配置参数。在使用之前阅读详细文档!

-h hostname

指定postgres监听来自客户端应用 TCP/IP 连接的 IP 主机名或地址。该值也可以是一个用逗号分隔的地址列表，或者*表示监听所有可用的地址。一个空值表示不监听任何 IP 地址，在这种情况下可以使用 Unix 域套接字连接到服务器。缺省只监听localhost。声明这个选项等效于设置listen_addresses配置参数。默认只监听localhost。指定这个选项等效于设置listen_addresses配置参数。

-i

允许远程客户端使用 TCP/IP （互联网域）连接。没有这个选项，将只接受本地连接。这个选项等效于在postgresql.conf中或者通过-h选项将listen_addresses设为*。

这个选项已经被废弃，因为它不允许访问listen_addresses的完整功能。所以最好直接设置listen_addresses。

-k directory

指定postgres用来监听来自客户端应用连接的 Unix 域套接字的目录。这个值也可以是一个逗号分隔的目录列表。一个空值指定不监听任何 Unix 域套接字，在这种情况下只能用 TCP/IP 套接字来连接到服务器。默认值通常是/tmp，但是可以在编译的时候修改。指定这个选项等效于设置unix_socket_directories配置参数。

-l

启用使用SSL的安全连接。要使这个选项可用，编译PostgreSQL时必须打开SSL支持。有关使用SSL的更多信息，请参考第 18.9 节

-N max-connections

设置该服务器将接受的最大客户端连接数。该参数的默认值由initdb自动选择。指定这个选项等效于设置max_connections配置参数。

-o extra-options

在extra-options中指定的命令行风格的参数会被传递给所有由这个postgres进程派生的服务进程。

extra-options中的空格被视作 参数分隔符，除非用反斜线（\）转义。要表示一个字面意 义上的反斜线，可以写成\\。通过多次使用-o 也可以指定多个参数。

这个选项的使用已经被废弃。用于服务器进程的所有命令行选项可以在postgres命令行上直接指定。

-p port

指定postgres用于监听客户端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展。默认为PGPORT环境变量的值。如果PGPORT没有设置，那么默认值是编译期间设立的值（通常是 5432）。如果你指定了一个非默认端口，那么所有客户端应用都必须用命令行选项或者PGPORT指定同一个端口。

-s

在每条命令结束时打印时间信息和其它统计信息。这个选项对测试基准和调节缓冲区数量有用处。

-S work-mem

指定内部排序和散列在使用临时磁盘文件之前能使用的内存数量。见第 19.4.1 节对work_mem配置参数的描述。

-V
--version

打印postgres版本并退出。

--name=value

设置一个命名的运行时参数；其缩写形式是-c。

--describe-config

这个选项会用制表符分隔的COPY格式导出服务器的内部配置变量、描述以及默认值。设计它的目的是用于管理工具。

-?
--help

显示有关postgres的命令行参数，并且退出。

半内部选项

这里描述的选项主要被用于调试目的，并且在某些情况下可以协助恢复严重受损的数据库。在生产数据库环境中应该不会去使用它们。在这里列举它们只是为了让PostgreSQL系统开发者使用。此外，这些选项可能在将来的版本中更改或删除而不另行通知。

-f { s | i | o | b | t | n | m | h }

禁止某种扫描和连接方法的使用：s和i分别禁用顺序和索引扫描， o、b和t分别禁用只用索引扫描、位图索引扫描以及 TID 扫描，而n、m和h则分别禁用嵌套循环、归并和哈希连接。

顺序扫描和嵌套循环连接都不可能完全被禁用。-fs和-fn选项仅仅是在有其他选择时不鼓励优化器使用这些计划类型。

-n

该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定postgres将不会重新初始化共享数据结构。一个有经验的系统程序员这时就可以使用调试器检查共享内存和信号量状态。

-O

允许修改系统表的结构。这个选项用于initdb。

-P

读取系统表时忽略系统索引（但在更改系统表时仍然更新索引）。这在从损坏的系统索引中恢复时有用。

-t pa[rser] | pl[anner] | e[xecutor]

打印与每个主要系统模块相关的查询的时间统计。这个选项不能和-s选项一起使用。

-T

该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定postgres将通过发送SIGSTOP信号停止其他所有服务器进程，但是并不让它们终止。这样就允许系统程序员手动从所有服务器进程收集内核转储。

-v protocol

声明这次会话使用的前/后服务器协议的版本数。该选项仅在内部使用。

-W seconds

在一个新服务器进程被启动时，它实施认证过程之后会延迟这个选项所设置的秒数。这就留出了机会来用一个调试器附着在服务器进程上。

用于单用户模式的选项

下面的选项仅适用于单用户模式（见单用户模式）。

--single

选择单用户模式。这必须是命令行中的第一个选项。

database

指定要访问的数据库的名称。这必须是命令行中的最后一个参数。如果省略它，则默认为用户名。

-E

在执行命令之前回显所有命令到标准输出。

-j

使用跟着两个新行的分号而不是仅用新行作为命令终止符。

-r filename

将所有服务器日志输出发送到filename中。 只有在作为一个命令行选项提供时，这个选项才会兑现。

环境

PGCLIENTENCODING

客户端使用的默认字符编码（客户端可以独立地覆盖它）。这个值也可以在配置文件中设置。

PGDATA

默认的数据目录位置。

PGDATESTYLE

DateStyle运行时参数的默认值（这个环境变量的使用已被废弃）。

PGPORT

默认端口号（在配置文件中设置更好）

诊断

一个提到了semget或shmget的错误消息可能意味着你需要配置内核来提供足够的共享内存和信号量。更多讨论请见第 18.4 节你也可以通过降低shared_buffers值减少PostgreSQL的共享内存消耗， 或者降低max_connections值减少信号量消耗，这样可以推迟对内核的重新配置。

如果一个消息说另外一个服务器已经在运行，应该仔细地检查，例如根据你的系统可以用命令

```
$ ps ax | grep postgres
```

或

```
$ ps -ef | grep postgres
```

如果你确信没有冲突的服务器正在运行，那么你可以删除消息中提到的锁文件然后再次尝试。

如果一个失败消息指示它无法绑定到一个端口，可能意味着该端口已经被某些非PostgreSQL进程所使用。如果你终止postgres并且立即使用相同的端口重启它，你也可能会得到这种错误。在这种情况系，你必须等待几秒直到操作系统关闭该端口，然后再重试。最后，如果你指定了一个操作系统认为需要保留的端口号，你可能也会得到这个错误。例如，很多版本的 Unix 认为低于 1024 的端口号是“可信的”并且只允许 Unix 超级用户访问它们。

注解

实用命令pg_ctl可以用来安全方便地启动和关闭postgres服务器。

只要有可能，就不要使用SIGKILL杀死主postgres服务器。这样会阻止postgres在终止前释放它持有的系统资源（例如共享内存和信号量）。这样可能会导致启动新的postgres进程时出现问题。

要正常地终止postgres服务器，可以使用SIGTERM、SIGINT或者SIGQUIT信号。第一个在退出前将等待所有客户端终止，第二个将强行断开所有客户端的连接，第三个会不做正确的关闭立即退出并且会导致重启时的恢复。

SIGHUP信号会重新加载服务器配置文件。也可以向一个单独的服务器进程发送SIGHUP信号，但是这样做通常没什么意义。

要取消一个正在运行的查询，可以向运行该查询的进程发送SIGINT信号。要干净地终止一个后端进程，可向它发送SIGTERM。在 SQL 中可调用的与这两种动作等效的命令可参考第 9.26.2 节的pg_cancel_backend和pg_terminate_backend。

postgres服务器使用SIGQUIT来告诉子服务器进程终止但不做正常的清理。该信号不应该被用户使用。向一个服务器进程发送SIGKILL也是不明智的 — 主postgres进程将把这解释为一次崩溃，并且作为其标准崩溃恢复过程的一部分，它将强制所有的后代进程退出。

缺陷

--选项在FreeBSD或OpenBSD上无法运行，应该使用-c。这在受影响的系统里是个缺陷；　如果这个缺陷没有被修复，将来的PostgreSQL版本将提供一种解决方案。

单用户模式

要启动一个单用户模式的服务器，使用这样的命令

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

用-D给服务器提供正确的数据库目录的路径，或者确保环境变量PGDATA被设置。同时还要指定你想在其中工作的特定数据库的名字。

通常，单用户模式的服务器会把换行符当做命令输入的终止符。它不明白分号的作用，因为那属于psql。要想把一个命令分成多行，必须在最后一个换行符以外的每个换行符前面敲一个反斜线。这个反斜线和旁边的新行都会被从输入命令中去掉。注意即使在字符串或者注释中也会这样做。

但是如果使用了-j命令行选项，那么单个新行将不会终止命令输入。相反，分号-新行-新行的序列才会终止命令输入。也就是说，输入一个紧跟着空行的分号。在这种模式下，反斜线-新行不会被特殊对待。此外，在字符串或者注释内的这类序列也不会被特殊对待。

不管在哪一种输入模式中，如果输入的一个分号不是正好在命令终止符之前或者不是命令终止符的一部分，它会被认为是一个命令分隔符。当真正输入一个命令终止符时，已经输入的多个语句将被作为一个单个事务执行。

要退出会话，输入EOF（通常是Control+D）。如果从上一个命令终止符以来已经输入了任何文本，那么EOF将被当作命令终止符，并且如果要退出则需要另一个EOF。

请注意单用户模式的服务器不会提供复杂的行编辑特性（例如没有命令历史）。但用户模式也不会做任何后台处理，例如自动检查点或者复制。

例子

要用默认值在后台启动postgres：

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

要用指定端口启动postgres，例如 1234：

```
$ postgres -p 1234
```

要使用psql连接到这个服务器，用 -p 选项指定这个端口：

```
$ psql -p 1234
```

或者设置环境变量PGPORT：

```
$ export PGPORT=1234
$ psql
```

命名运行时参数可以用这些形式之一设置：

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

两种形式都覆盖postgresql.conf中可能存在的work_mem设置。请注意在参数名中的下划线在命令行可以写成下划线或连字符。除了用于短期的实验外，更好的习惯是编辑postgresql.conf中的设置， 而不是倚赖命令行开关来设置参数。

参见

initdb, pg_ctl

postmaster

postmaster — PostgreSQL数据库服务器

大纲

postmaster [option...]

描述

postmaster是postgres的一个废弃的别名。

参见

postgres

部分 VII. 内部

这一部分包含PostgreSQL开发者可能用到的各类信息。

目录

第 51 章 PostgreSQL内部概述

> 作者
>
> 这一章的内容来自于[sim98]的一部分，它是Stefan Simkovics在维也纳技术大学的硕士学位论文，指导人是 Georg Gottlob教授和Mag. Katrin Seyr。

本章给出了PostgreSQL后台内部结构的一个总览。在阅读了下面的小节后，你将对一个查询是如何被执行的有一定了解。本章并非要对PostgreSQL的内部操作给出一个详细的介绍，否则这个文档将会变得非常长。本章希望帮助读者理解当后台收到一个查询后，在结果被返回给客户之前通常会发生怎样的操作。

51.1. 一个查询的路径

这里我们将介绍为了获取结果，一个查询将会经历那些阶段。

1. 一个由应用程序到PostgreSQL服务器的连接必须被建立。应用程序传递一个查询给服务器并等待接收由服务器传回的结果。

2. 分析阶段对由应用程序传递的查询进行语法检查，并创建一个查询树。

3. 重写系统得到分析阶段创建的查询树，并查找可以应用到该查询树的任何规则（存储在系统目录中）。对找到的规则，它会执行规则体中给定的转换。

   重写系统的一个应用是实现视图。不论何时发生一个视图（即一个虚拟表）上的查询，重写系统将用户查询重写为一个访问视图定义中给定的基本表的查询来替代。

4. 规划器/优化器接手（重写过的）查询树并创建一个将被作为执行器输入的查询计划。

   它会先创建所有可能导向相同结果的路径。例如，如果在一个被扫描的关系上有一个索引，则有两条可供扫描的路径。其中之一是一个简单的顺序扫描，而另一个则是使用索引。接下来执行每条路径所需的代价被估算出来并且代价最低的路径将被选中。代价最低的路径将被扩展成一个完整的计划可供执行器使用。

5. 执行器递归地逐步通过计划树并按照计划表述的方式获取行。执行器在扫描关系时会使用存储系统、执行排序和连接、估算条件并最后归还得到的行。

在下面的小节中我们将覆盖上述列出的每一项的详细内容，以便更好地理解PostgreSQL的内部控制和数据结构。

51.2. 连接如何建立

PostgreSQL以一种简单的“一用户一进程”的客户端/服务器模型实现。在该模型中，一个客户端进程仅连接到一个服务器进程。由于我们无法预先知道会有多少连接被建立，我们必须使用一个主进程在每次连接请求时生产一个新的服务器进程。该主进程被称为postgres，它在一个特定的TCP/IP端口监听进入的连接。当一个连接请求被监测到时，postgres会产生一个新的服务器进程。服务器作业之间通过信号和共享内存通信，以保证并发数据访问时的数据完整性。

客户端进程可以是任何符合PostgreSQL协议（见第 53 章的程序。很多客户端基于C语言库libpq，但也有一些该协议的独立实现存在，例如Java的JDBC驱动。

一旦一个连接被建立，客户端进程就能发送一个查询给后端（服务器）。查询被以纯文本传送，即在前端（客户端）不做任何分析。服务器会分析查询，创建一个执行计划，然后执行之并通过已建立的连接向客户端返回检索到的行。

51.3. 分析器阶段

分析器阶段由两部分组成：

- 分析器定义在gram.y和scan.l中，它使用Unix工具bison和flex构建。
- 转换处理将对分析器返回的数据结构进行修改和增加。

51.3.1. 分析器

分析器必须检查查询字符串（以纯文本形式到达）是否为合法语法。如果语法正确将建立一个分析树并返回之，否则将返回一个错误。 语法分析器和词法分析器使用著名的Unix工具bison和flex实现。

词法分析器定义在文件scan.l中，并负责识别标识符、SQL关键词等。对于找到的每一个关键词或标识符将生成一个记号并返回给语法分析器。

语法分析器定义在gram.y文件中，它由一组语法规则 和动作，动作将在规则被触发时被执行。动作的代码（实际上是C代码）将被用于构建分析树。

程序flex把文件scan.l转换成C源文件scan.c， 程序bison把文件gram.y转换为gram.c。在这些转换结束后，一个正规的C编译器就可以用于创建分析器。绝不要对生成的C文件做任何修改，因为每次flex或bison被调用都会重写它们。

> **注意**
>
> 前面提到的转换和编译通常是由随PostgreSQL源代码发布的makefiles自动完成。

对于bison的详细介绍或者gram.y中的语法规则超出了本文的范围。有很多书籍和文档介绍flex和bison。在学习gram.y中的语法之前你应该先熟悉bison，否则你将无法理解发生了什么。

51.3.2. 转换处理

分析阶段根据SQL的语法结构的固定规则创建一个分析树。它不会在系统目录做任何查找，这样它不可能了解所请求的操作的详细语义。在分析器完成之后，转换处理接手分析器返还的树，并进行语义解释来理解该查询引用了哪些表、函数和操作符。用于表示该信息的数据结构被称为查询树。

将原始分析从语义分析中分离出来的原因是系统目录的查找只能在一个事务中完成，而我们不希望在收到一个查询字符串时立即开始一个事务。原始分析阶段足以识别事务控制命令（BEGIN、ROLLBACK等），并且这些可以在没有任何进一步分析之前正确地被执行。一旦我们知道我们正在处理一个确切的查询（例如SELECT或UPDATE），就可以开始一个事务（如果我们还不在其中）。只有到这时转换处理才能被调用。

由转换处理创建的查询树在结构上和原始分析树有很多地方相似，但是在细节上有很多不同之处。例如，分析树中的一个FuncCall节点表示某些在语法上看起来像一个函数调用的东西。它可能被转换成一个FuncExpr或Aggref节点，取决于被引用的名字是一个普通函数或是一个聚集函数。此外，关于列和表达式结果的实际数据类型的信息被加入到了查询树中。

51.4. PostgreSQL规则系统

PostgreSQL支持一个强大的 规则系统用于支持视图说明和模糊不清的视图更新。本来PostgreSQL规则系统由两种实现组成：

- 第一种利用行级处理工作，并且被实现于执行器深层。当单独一行被访问时规则系统被调用。这种实现在1995年被移除，那时最后一个Berkeley Postgres项目的官方发布被转换为Postgres95。
- 第二种规则系统的实现是一种称为查询重写的技术。重写系统是一个存在于分析器阶段和规划器/优化器之间的模块。这种技术也被实现了。

更多关于查询重写器的讨论可见第 41 章因此没有必要在这里涉及这些内容。我们将仅指出重写器的输入和输出都是查询树，即在树的表现形式或语义细节层次上都没有改变。重写可以被看成是某种形式的宏扩展。

51.5. 规划器/优化器

规划器/优化器的任务是创建一个最佳的执行计划。一个给定的SQL查询（今后将是一个查询树）实际上可以以很多种不同的方式被执行，其中的每一种都会产生相同的结果集。如果在计算上可行，查询优化器将检查这些可能的执行计划中的每一个，最后选择其中被期望“跑得最快的”那一个。

> **注意**
>
> 在某些情况下，检查一个查询的每一种可能的执行方式会耗费非常多的时间和内存空间。特别是当查询涉及到大量连接操作时。为了能在合理的时间内决定一个合理的（不一定是最佳的）查询计划，当连接数量超过一个阈值（见geqo_threshold）时PostgreSQL使用了一种遗传查询优化器（见第 60 章。

规划器搜索过程实际上依靠称为路径的数据结构工作，它是一种缩短版的计划，其中只包含规划器做决定所需要的信息。当最低代价的路径被确定后，一个全功能的计划树将被建立并传递给执行器。这表示所期望的执行计划已经拥有足够的细节以供执行器执行它。在本节剩下的部分，我们将忽略路径和计划之间的区别。

51.5.1. 生成可能的计划

规划器/优化器从扫描查询中用到的每一个单独的关系（表）开始生成计划。可能的计划根据每一个关系上可用的索引决定。在一个关系上总是有执行一个顺序扫描的可能，因此一个顺序扫描计划总是会被创建。假设在一个关系上定义有一个索引（例如一个B-tree索引）并且查询包含限制relation.attribute OPR constant。如果relation.attribute正好匹配该B-tree索引的键并且OPR是该索引的操作符类之一，另一个使用B-tree索引扫描该索引的计划将被创建。如果还有索引存在且查询中的限制正好匹配一个索引的键，其他计划也会被考虑。如果有索引的顺序能匹配ORDER BY子句（如果有）或者对于归并连接有用（见下文），也会为该索引创建索引扫描计划。

如果查询需要连接两个或更多关系，在所有扫描单个关系的可能计划都被找到后，连接计划将会被考虑。三种可用的连接策略是：

- 嵌套循环连接: 对左关系找到的每一行都要扫描右关系一次。这种策略最容易实现但是可能非常耗时（但是，如果右关系可以通过索引扫描，这将是一个不错的策略。因为可以用左关系当前行的值来作为右关系上索引扫描的键）。
- 归并连接：在连接开始之前，每一个关系都按照连接属性排好序。然后两个关系会被并行扫描，匹配的行被整合成连接行。由于这种连接中每个关系只被扫描一次，因此它很具有吸引力。它所要求的排序可以通过一个显式的排序步骤得到，或使用一个连接键上的索引按适当顺序扫描关系得到。
- 哈希连接：右关系先被扫描并且被载入到一个哈希表，使用连接属性作为哈希键。接下来左关系被扫描，扫描中找到的每一行的连接属性值被用作哈希键在哈希表中查找匹配的行。

当查询涉及两个以上的关系时，最终结果必须由一个连接步骤树构成，每个连接步骤有两个输入。规划器会检查不同可能的连接序列来找到代价最小的那一个。

如果查询是用的关系数少于geqo_threshold，将使用一次接近穷举的搜索来查找最好的连接顺序。如果任何两个关系在WHERE条件中存在一个相应的连接子句（即存在类似于where rel1.attr1=rel2.attr2的限制），规划器会有限考虑它们之间的连接。没有任何连接子句的连接对只有在别无选择时才会被考虑，即一个关系没有任何可用的对于其他关系的连接子句。对规划器所考虑的每一个连接对会生成所有可能的计划，其中代价（被估计为）最低的一个将被选择。

当连接关系数超过geqo_threshold时，连接序列将考虑通过启发式方法来确定，详见第 60 章否则处理将和前面相同。

成品计划树包含基本关系的顺序或索引扫描，外加所需的嵌套循环、归并或哈希连接节点，以及任何所需的辅助步骤，例如排序节点或聚集函数计算节点。这些节点中的大部分具有执行选择（丢弃不符合指定布尔条件的行）和投影（根据指定列值计算派生列，即标量表达式的计算）的能力。规划器的职责之一就是在计划树最合适的节点上附加来自于子句的选择条件和需要的输出表达式。

51.6. 执行器

执行器接手规划器/优化器创建的计划，并递归地处理之以抽取所需的行集。这本质上是一种需求拉动的管道机制。每次一个计划节点被调用时，它必须交付一个或多个行，或者报告已经完成了行的交付。

为了提供一个具体例子，假设顶层节点是一个MergeJoin节点。在归并完成之前，两个行必须先被获取（每一个来自于一个子计划）。因此执行器递归地调用它自己去处理子计划（从附加在lefttree的子计划开始）。新的顶层节点（左子计划的顶层节点），我们说是一个Sort节点，并且又要递归来获取一个输入行。Sort的子节点可以是一个SeqScan节点，表示真正地读取一个表。该节点的执行将会使执行器从表中获取一行并将它返回给调用节点。Sort节点将反复调用它的子节点来获得所有需要排序的行。当输入耗尽后（子节点将返回一个NULL来标识），Sort节点执行排序，并且最后能够返回它的第一个输出行，及排序后的第一个。它会把剩下的行保存下来，这样它可以根据后续的要求按照排好的顺序返回这些行。

MergeJoin节点也会相似地从其右子计划要求第一个行。然后它会比较两个子节点提供的行看它们是否能被连接，如果可以它会返回一个连接行给调用者。在下一次调用时，或者它无法连接当前的输入对时，它会前进到一个表或另一个表的下一行（取决于比较的结果），并再次检查匹配。最后，某个子计划耗尽，MergeJoin节点返回NULL表示它没有更多连接行可以提供。

复杂的查询可能涉及多层计划节点，但是一般的方法是相同的：每个节点在被调用时计算并返回它的下一个输出行。每个节点同时也负责应用由规划器分配给它的选择或投影表达式。

执行器机制被用于四种基本SQL查询类型：SELECT、INSERT、 UPDATE以及DELETE。对于SELECT，顶层执行器代码只需要发送查询计划树返回的每个行给客户端。对于INSERT，每一个被返回的行被插入到INSERT中指定的目标表中。这通过一个被称为ModifyTable的特殊顶层计划节点完成（一个简单的INSERT ... VALUES命令会创建一个由一个Result节点组成的简单计划树，该节点只计算一个结果行，在它之上的ModifyTable节点会执行插入。但是INSERT ... SELECT可以用到执行器机制的全部功能）。对于UPDATE，规划器会安排每一个计算行包含所有被更新的列值加上原始目标行的TID（元组ID或行ID），这些数据也会被输入到一个ModifyTable节点，该节点将利用这些信息创建一个新的被更新行并标记旧行为被删除。对于DELETE，唯一被计划返回的列是TID，ModifyTable节点简单地使用TID访问每一个目标行并将其标记为被删除。

第 52 章 系统目录

系统目录是关系型数据库存放模式元数据的地方，比如表和列的信息，以及内部统计信息等。PostgreSQL的系统目录就是普通表。你可以删除并重建这些表、增加列、插入和更新数值， 然后彻底把你的系统搞垮。 通常情况下，我们不应该手工修改系统目录，通常有SQL命令可以做这些事情。（例如，CREATE DATABASE向 pg_database表插入一行 — 并且实际上在磁盘上创建该数据库。）。 有几种特别深奥的操作例外，但是随着时间的流逝其中的很多也可以用 SQL 命令来完成，因此对系统目录直接修改的需求也越来越小。

52.1. 概述

表 52.列出了系统目录。 每个目录更详细的文档见后文。

大多数系统目录都是在数据库创建的过程中从模版数据库中拷贝过来的， 因此都是数据库相关的。少数目录在物理上是在一个集簇的所有数据库间中共享的，这些将在每一个目录单独的描述中介绍。

表 52.1. 系统目录

目录名	用途
pg_aggregate	聚集函数
pg_am	索引访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_attrdef	列默认值
pg_attribute	表列(“属性”)
pg_authid	认证标识符（角色）
pg_auth_members	认证标识符成员关系
pg_cast	转换（数据类型转换）
pg_class	表、索引、序列、视图 （“关系”）
pg_collation	排序规则（locale信息）
pg_constraint	检查约束、唯一约束、主键约束、外键约束
pg_conversion	编码转换信息
pg_database	本数据库集簇中的数据库
pg_db_role_setting	每角色和每数据库的设置
pg_default_acl	对象类型的默认权限
pg_depend	数据库对象间的依赖
pg_description	数据库对象上的描述或注释
pg_enum	枚举标签和值定义
pg_event_trigger	事件触发器
pg_extension	已安装扩展
pg_foreign_data_wrapper	外部数据包装器定义
pg_foreign_server	外部服务器定义
pg_foreign_table	外部表信息
pg_index	索引信息
pg_inherits	表继承层次

目录名	用途
pg_init_privs	对象初始特权
pg_language	编写函数的语言
pg_largeobject	大对象的数据页
pg_largeobject_metadata	大对象的元数据
pg_namespace	模式
pg_opclass	访问方法操作符类
pg_operator	操作符
pg_opfamily	访问方法操作符族
pg_partitioned_table	表的分区键的信息
pg_pltemplate	过程语言的模板数据
pg_policy	行安全策略
pg_proc	函数和过程
pg_publication	用于逻辑复制的发布
pg_publication_rel	发布映射的关系
pg_range	范围类型的信息
pg_replication_origin	已注册的复制源
pg_rewrite	查询重写规则
pg_seclabel	数据库对象上的安全标签
pg_sequence	有关序列的信息
pg_shdepend	共享对象上的依赖
pg_shdescription	共享对象上的注释
pg_shseclabel	共享数据库对象上的安全标签
pg_statistic	规划器统计
pg_statistic_ext	扩展的规划器统计信息
pg_subscription	逻辑复制订阅
pg_subscription_rel	订阅的关系状态
pg_tablespace	本数据库集簇内的表空间
pg_transform	转换（将数据类型转换为过程语言需要的形式）
pg_trigger	触发器
pg_ts_config	文本搜索配置
pg_ts_config_map	文本搜索配置的记号映射
pg_ts_dict	文本搜索字典
pg_ts_parser	文本搜索分析器
pg_ts_template	文本搜索模板
pg_type	数据类型
pg_user_mapping	将用户映射到外部服务器

52.2. pg_aggregate

目录pg_aggregate存储关于聚集函数的信息。聚集函数是对一个数值集合（典型的是每个匹配查询条件的行中的同一个列的值）　进行操作的函数，它返回从这些值中计算出的一个数

值。 典型的聚集函数是 sum、count和max。pg_aggregate里的每个项都是一个pg_proc项的扩展。pg_proc项记载该聚集的名字、输入和输出数据类型，以及其他一些和普通函数类似的信息。

表 52.2.pg_aggregate的列

名称	类型	引用	描述
aggfnoid	regproc	pg_proc.oid	聚集函数在pg_proc中的OID
aggkind	char		聚集类型： n表示“普通”聚集， o表示“有序集”聚集，或者 h表示“假想集”聚集
aggnumdirectargs	int2		一个有序集或者假想集聚集的直接（非聚集）参数的数量，一个可变数组算作一个参数。 如果等于pronargs，该聚集必定是可变的并且该可变数组描述聚集参数以 及最终直接参数。对于普通聚集总是为零。
aggtransfn	regproc	pg_proc.oid	转移函数
aggfinalfn	regproc	pg_proc.oid	最终函数（如果没有就为零）
aggcombinefn	regproc	pg_proc.oid	结合函数（如果没有就为零）
aggserialfn	regproc	pg_proc.oid	序列化函数（如果没有就为零）
aggdeserialfn	regproc	pg_proc.oid	反序列化函数（如果没有就为零）
aggmtransfn	regproc	pg_proc.oid	用于移动聚集模式的向前转移函数（如果没有就为零）
aggminvtransfn	regproc	pg_proc.oid	用于移动聚集模式的反向转移函数（如果没有就为零）
aggmfinalfn	regproc	pg_proc.oid	用于移动聚集模式的最终函数（如果没有就为零）
aggfinalextra	bool		为真则向aggfinalfn传递额外的哑参数
aggmfinalextra	bool		为真则向aggmfinalfn传递额外的哑参数
aggfinalmodify	char		aggfinalfn是否修改传递状态值： 如果是只读则为r， 如果不能在aggfinalfn之后

名称	类型	引用	描述
			应用aggtransfn则为s， 如果它修改该值则为w
aggmfinalmodify	char		和aggfinalmodify类似，但是用于aggmfinalfn
aggsortop	oid	pg_operator.oid	相关联的排序操作符（如果没有则为0）
aggtranstype	oid	pg_type.oid	聚集函数的内部转移（状态）数据的数据类型
aggtransspace	int4		转移状态数据的近似平均尺寸（字节），或者为零表示使用一个默认估算值
aggmtranstype	oid	pg_type.oid	聚集函数用于移动聚集欧氏的内部转移（状态）数据的数据类型（如果没有则为零）
aggmtransspace	int4		转移状态数据的近似平均尺寸（字节），或者为零表示使用一个默认估算值
agginitval	text		转移状态的初始值。这是一个文本域，它包含初始值的外部字符串表现形式。如果这个域为空，则转移状态值从空值开始。
aggminitval	text		用于移动聚集模式的转移状态初值。这是一个文本域，它包含了以其文本字符串形式表达的初值。 如果这个域为空，则转移状态值从空值开始。

新的聚集函数可通过CREATE AGGREGATE命令注册。更多关于编写聚集函数以及转移函数的含义等信息请参见第 38.11 节

52.3. pg_am

目录pg_am存储有关系引访问方法的信息。系统支持的每种访问方法在这个目录中都有一行。当前，只有索引拥有访问方法。索引访问方法的需求在第 61 章详细讨论。

表 52.3.pg_am的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）

名称	类型	引用	描述
amname	name		访问方法的名字
amhandler	regproc	pg_proc.oid	负责提供有关该访问方法信息的处理器函数的 OID
amtype	char		当前总是i，表示是一个索引访问方法。未来可能会允许其他值

> **注意**
>
> 在PostgreSQL 9.6 之前，pg_am包含很多额外的列以表示索引访问方法的性质。那些数据现在只有在 C 代码级别才是直接可见的。不过，系统中增加了pg_index_column_has_property()和一些相关函数来允许 SQL 查询检查索引访问方法的性质，请见表 9.63

52.4. pg_amop

目录pg_amop存储关于与访问方法操作符族相关的操作符信息。对于一个操作符族中的每一个成员即操作符都在这个目录中有一行。一个成员可以是一个搜索操作符或者一个排序操作符。一个操作符可以出现在多个族中，但在同一个组中既不能出现在多个搜索位置也不能出现在多个排序位置（虽然不太可能出现，但是允许一个操作符同时用于搜索和排序目的）。

表 52.4.pg_amop的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
amopfamily	oid	pg_opfamily.oid	这个项所在的操作符族
amoplefttype	oid	pg_type.oid	操作符的左手输入数据类型
amoprighttype	oid	pg_type.oid	操作符的右手输入数据类型
amopstrategy	int2		操作符策略号
amoppurpose	char		操作符目的，s表示搜索，o表示排序
amopopr	oid	pg_operator.oid	操作符的OID
amopmethod	oid	pg_am.oid	使用此操作符族的索引访问方法
amopsortfamily	oid	pg_opfamily.oid	如果是一个排序操作符，该项会根据这个B树操作符族排序，如果是一个搜索操作符则为0

一个“搜索”操作符项意味着该操作符族的一个索引可以被搜索来查找所有满足如下条件的行： WHERE indexed_column operator constant。 显然，这样的一个操作符必须返回boolean，且它的左手输入类型必须匹配索引列的数据类型。

一个“排序”操作符项意味着该操作符族的一个索引可以被扫描来返回以如下顺序排列的行： ORDER BY indexed_column operator constant。 这样一个操作符能够返回任何可排序数据类型，尽管它的左手输入类型必须匹配索引列的数据类型。 ORDER BY的准确语义由amopsortfamily列指定，它必须引用一个适合于操作符结果类型的B树操作符族。

> **注意**
>
> 目前，一个排序操作符的排序顺序被假设为其引用的操作符族的默认值，即ASC NULLS LAST。未来可能会通过增加额外的列来显式地指定排序选项。

一个项的amopmethod必须和它所包含的操作符族的opfmethod相匹配（这里包括amopmethod是一个为了性能原因而故意对目录结构做的反规范化）。此外，amoplefttype和amoprighttype也必须匹配被引用的pg_operator项的oprleft和oprright域。

52.5. pg_amproc

目录pg_amproc存储关于访问方法操作符族相关的支持函数。属于一个操作符族的每一个支持函数在这个目录中都有一行。

表 52.5.pg_amproc的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
amprocfamily	oid	pg_opfamily.oid	使用这个项的操作符族
amproclefttype	oid	pg_type.oid	相关操作符的左手输入数据类型
amprocrighttype	oid	pg_type.oid	相关操作符的右手输入数据类型
amprocnum	int2		支持过程编号
amproc	regproc	pg_proc.oid	过程的OID

amproclefttype和amprocrighttype列的通常解释是它们标识了一个特定支持过程所支持的操作符的左右输入类型。对于某些访问方法它们和支持过程本身的输入数据类型相匹配，而对其他的则不会匹配。对于一个索引有一个“默认”支持过程的概念，这些支持过程的amproclefttype和amprocrighttype都等于索引操作符类的opcintype。

52.6. pg_attrdef

pg_attrdef存储列的默认值。列的主要信息存储在pg_attribute（见下文）。只有那些显式指定了一个默认值的列（在表创建时或列增加时）才会在这个目录中有一个项。

表 52.6.pg_attrdef的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
adrelid	oid	pg_class.oid	该列所属的表

名称	类型	引用	描述
adnum	int2	pg_attribute.attnum	列号
adbin	pg_node_tree		列默认值的内部表示
adsrc	text		默认值的人类可读的表示

adsrc列的存在是有历史原因的，并且最好不要被使用，因为它不能跟踪可能导致默认值表现形式的外部变化。如果要显示默认值，反编译adbin列（例如使用pg_get_expr）是一种更好的方法。

52.7. pg_attribute

目录pg_attribute存储有关表列的信息。数据库中的每一个表的每一个列都恰好在pg_attribute中有一行。（这其中也会有索引的属性项，并且事实上所有具有pg_class项的对象在这里都有属性项） entries.）

术语属性等同于列，这里使用它只是出于历史原因。

表 52.7.pg_attribute的列

名称	类型	引用	描述
attrelid	oid	pg_class.oid	列所属的表
attname	name		列名
atttypid	oid	pg_type.oid	列的数据类型
attstattarget	int4		attstattarget控制由ANALYZE对此列收集的统计信息的细节层次。0值表示不会收集任何统计信息。一个负值则说明直接使用系统默认的目标。正值的确切含义取决于数据类型。对于标量数据类型，attstattarget既是要收集的“最常见值”的目标号，也是要创建的柱状图容器的目标号。
attlen	int2		本列类型的pg_type.typlen一个拷贝
attnum	int2		列的编号。一般列从1开始向上编号。系统列（如oid）则拥有（任意）负值编号。
attndims	int4		如果该列是一个数组类型，这里就是其维度数；否则为0。（在目前一个数组的维度数并不被强制，因此任何非零值都能有效地表明“这是一个数组”。）

名称	类型	引用	描述
attcacheoff	int4		在存储中总是为-1，但是当被载入到一个内存中的行描述符后，这里可能会被更新为属性在行内的偏移
atttypmod	int4		atttypmod记录了在表创建时提供的类型相关数据（例如一个varchar列的最大长度）。它会被传递给类型相关的输入函数和长度强制函数。对于那些不需要atttypmod的类型，这个值通常总是为-1。
attbyval	bool		该列类型的pg_type.typbyval的一个拷贝
attstorage	char		通常是该列类型的pg_type.typstorage的一个拷贝。对于可TOAST的数据类型，这可以在列创建后被修改以控制存储策略。
attalign	char		该列类型的pg_type.typalign的一个拷贝
attnotnull	bool		这表示一个非空约束。
atthasdef	bool		该列有一个默认值，在此情况下在pg_attrdef目录中会有一个对应项来真正记录默认值。
atthasmissing	bool		该列在行中完全缺失时会用到这个列的值，如果在行创建之后增加一个有非易失DEFAULT值的列，就会发生这种情况。实际使用的值被存放在attmissingval列中。
attidentity	char		如果是一个零字节（''），则不是一个标识列。否则，a = 总是生成，d = 默认生成。
attisdropped	bool		该列被删除且不再有效。一个删除的列仍然物理存在于表中，

名称	类型	引用	描述
			但是会被分析器忽略并因此无法通过SQL访问。
attislocal	bool		该列是由关系本地定义的。注意一个列可以同时是本地定义和继承的。
attinhcount	int4		该列的直接祖先的编号。一个具有非零编号祖先的列不能被删除或者重命名。
attcollation	oid	pg_collation.oid	该列被定义的排序规则，如果该列不是一个可排序数据类型则为0。
attacl	aclitem[]		列级访问权限
attoptions	text[]		属性级选项，以“keyword=value”形式的字符串
attfdwoptions	text[]		属性级的外部数据包装器选项，以“keyword=value”形式的字符串
attmissingval	anyarray		这个列中是一个含有一个元素的数组，其中的值被用于该列在行中完全缺失时，如果在行创建之后增加一个有非易失DEFAULT值的列，就会发生这种情况。只有当atthasmissing为真时才使用这个值。如果没有值则该列为空。

在一个被删除的列的pg_attribute的项中，atttypid被重置为0，但attlen以及其他从pg_type拷贝的域仍然有效。这种安排用于处理一种情况，即被删除列的数据类型后来被删除，并且因此不再有相应的pg_type行。attlen和其他域可以被用来解释表的一行的内容。

52.8. pg_authid

目录pg_authid包含关于数据库授权标识符（角色）的信息。角色把“用户”和“组”的概念包含在内。一个用户实际上就是一个rolcanlogin标志被设置的角色。任何角色（不管rolcanlogin设置与否）都能够把其他角色作为成员，参见pg_auth_members。

由于这个目录包含口令，它不能是公共可读的。pg_roles是在pg_authid上的一个公共可读视图，它隐去了口令域。

第 21 章包含关于用户和权限管理的详细信息。

由于用户标识符是集簇范围的，pg_authid在一个集簇的所有数据库之间共享：在一个集簇中只有一份pg_authid拷贝，而不是每个数据库一份。

表 52.8.pg_authid的列

名字	类型	描述
oid	oid	行标识符（隐藏属性，必须被显式选择才会显示）
rolname	name	角色名
rolsuper	bool	角色是否拥有超级用户权限
rolinherit	bool	如果本角色是另一个角色的成员，本角色是否自动另一个角色的权限
rolcreaterole	bool	角色是否能创建更多角色
rolcreatedb	bool	角色是否能创建数据库
rolcanlogin	bool	角色是否能登录。即该角色是否能够作为初始会话授权标识符
rolreplication	bool	角色是一个复制角色。复制角色可以启动复制连接并且创建和删除复制槽。
rolbypassrls	bool	角色是否可以绕过所有的行级安全性策略，详见 第 5.7 节
rolconnlimit	int4	对于可以登录的角色，本列设置该角色可以同时发起最大连接数。-1表示无限制。
rolpassword	text	口令（可能被加密过），如果没有口令则为空。格式取决于使用的加密方法的形式。
rolvaliduntil	timestamptz	口令过期时间（只用于口令鉴定），如果永不过期则为空

对于一个MD5加密的口令，rolpassword列将由字符串md5后面跟上一个32字符的十六进制MD5哈希值构成。MD5哈希值将是该用户的口令串接上它们的用户名。例如，如果用户joe的口令是xyzzy，则PostgreSQL将存储xyzzyjoe的md5哈希。

如果口令采用SCRAM-SHA-256加密，它的格式是：

SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:<ServerKey>

其中salt、StoredKey和ServerKey是Base64编码格式。这种格式与RFC 5803说明的格式相同。

不遵守上述格式的口令被假定为未加密。

52.9. pg_auth_members

目录pg_auth_members展示了角色之间的成员关系。允许任何无环的关系集合。

由于用户标识符是集簇范围的，pg_auth_members在一个集簇的所有数据库之间共享：在一个集簇中只有一份pg_auth_members拷贝，而不是每个数据库一份。

表 52.9.pg_auth_members的列

名称	类型	引用	描述
roleid	oid	pg_authid.oid	拥有成员的角色的ID
member	oid	pg_authid.oid	roleid的成员角色的ID
grantor	oid	pg_authid.oid	授权此成员关系的角色的ID
admin_option	bool		如果member能把roleid的成员关系授予他人，则为真

52.10. pg_cast

目录pg_cast存储数据类型转换路径，包括内建的和用户定义的类型。

需要注意的是，pg_cast并不表示系统知道如何执行的所有类型转换，它只包括哪些不能从某些普通规则推导出的转换。例如，一个域及其基类型之间的转换并未显式地在pg_cast中展示。另一个重要的例外是“自动 I/O转换造型”，它们通过数据类型自己的I/O函数来转换成（或者转换自）text或其他字符串类型，这些转换也没有显式地在pg_cast中表示。

表 52.10.pg_cast的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
castsource	oid	pg_type.oid	源数据类型的OID
casttarget	oid	pg_type.oid	目标数据类型的OID
castfunc	oid	pg_proc.oid	执行该转换的函数的OID。如果该转换方法不需要一个函数则存储0。
castcontext	char		指示该转换能被调用的环境。 e表示仅能作为一个显式转换（使用CAST或::语法）。 a表示在赋值给目标列时隐式调用， 和显式调用一样。 i表示在表达式中隐式调用，和其他转换一样。
castmethod	char		指示转换如何被执行。 f表明使用castfunc中指定的函数。 i表明使用输入/输出函数。 b表明该类型是二进制可转换的，因此不需要转换。

在pg_cast里列出的类型转换函数必须总是以转换的源类型作为它的第一个参数类型， 并且返回转换的目标类型作为它的结果类型。一个类型转换函数最多有三个参数。 如果出现了

第二个参数，必须是integer类型；它接受与目标类型关联的修饰词， 如果没有，就是 -1。如果出现了第三个参数，那么必须是boolean类型； 如果该类型转换是一种明确的转换，那么它接受true，否则接受false。

在pg_cast里创建一条源类型和目标类型相同的记录是合理的， 只要相关联的函数接受多过一个参数。这样的记录代表“长度转换函数”， 它们把该类型的值转换为对特定的类型合法的值。

如果一个pg_cast的项有着不同的原类型和目标类型， 并且有一个接收多于一个参数的函数，那么它会在一个步骤中完成从一种类型到另外一种类型的转换并应用一个长度转换。如果没有这样的项，使用一个类型修改器的转换涉及两个步骤， 一个是在数据类型之间转换，另外一个是应用修改器。

52.11. pg_class

目录pg_class记录表和几乎所有具有列或者像表的东西。这包括索引（但还要参见pg_index）、序列（但还要参见pg_sequence）、视图、物化视图、组合类型和TOAST表，参见relkind。下面，当我们提及所有这些类型的对象时我们使用“关系”。并非所有列对于所有关系类型都有意义。

表 52.11.pg_class的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
relname	name		表、索引、视图等的名字
relnamespace	oid	pg_namespace.oid	包含该关系的名字空间的OID
reltype	oid	pg_type.oid	可能存在的表行类型所对应数据类型的OID（对索引为0，索引没有pg_type项）
reloftype	oid	pg_type.oid	对于有类型的表，为底层组合类型的OID，对于其他所有关系为0
relowner	oid	pg_authid.oid	关系的拥有者
relam	oid	pg_am.oid	如果这是一个索引，表示索引使用的访问方法（B树、哈希等）
relfilenode	oid		该关系的磁盘文件的名字，0表示这是一个“映射”关系，其磁盘文件名取决于低层状态
reltablespace	oid	pg_tablespace.oid	该关系所存储的表空间。如果为0，使用数据库的默认表空间。（如果关系无磁盘文件时无意义）
relpages	int4		该表磁盘表示的尺寸，以页面计（页面尺寸为BLCKSZ）。这只是一个由规划器使

名称	类型	引用	描述
			用的估计值。它被VACUUM、ANALYZE以及一些DDL命令（如CREATE INDEX）所更新。
reltuples	float4		表中的存活行数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE以及一些DDL命令（如CREATE INDEX）所更新。
relallvisible	int4		在表的可见性映射表中被标记为全可见的页数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE以及一些DDL命令（如CREATE INDEX）所更新。
reltoastrelid	oid	pg_class.oid	与该表相关联的TOAST表的OID，如果没有则为0。TOAST表将大属性“线外”存储在一个二级表中。
relhasindex	bool		如果这是一个表并且其上建有（或最近建有）索引则为真
relisshared	bool		如果该表在集簇中的所有数据库间共享则为真。只有某些系统目录（如pg_database）是共享的。
relpersistence	char		p = 永久表，u = 无日志表， t = 临时表
relkind	char		r = 普通表， i = 索引， S = 序列， t = TOAST表， v = 视图，m = 物化视图， c = 组合类型， f = 外部表， p = 分区表， I = 分区索引
relnatts	int2		关系中用户列的数目（系统列不计算在内）。在pg_attribute中必须有这么多对应的项。另请参阅pg_attribute.attnum。

名称	类型	引用	描述
relchecks	int2		表上CHECK约束的数目，参见pg_constraint目录
relhasoids	bool		如果为关系的每一行生成一个OID则为真
relhasrules	bool		如果表有（或曾有）规则则为真，参见pg_rewrite目录
relhastriggers	bool		如果表有（或曾有）触发器则为真，参见pg_trigger目录
relhassubclass	bool		如果表有（或曾有）任何继承子女则为真
relrowsecurity	bool		如果表上启用了行级安全性则为真，参见pg_policy目录
relforcerowsecurity	bool		如果行级安全性（启用时）也适用于表拥有者则为真，参见pg_policy目录
relispopulated	bool		如果表已被填充则为真（对于所有关系该列都为真，但对于某些物化视图却不是）
relreplident	char		用来为行形成“replica identity”的列： d = 默认 （主键，如果存在)， n = 无， f = 所有列 i = 索引的indisreplident被设置或者为默认
relispartition	bool		如果表或索引是一个分区，则为真
relrewrite	oid	pg_class.oid	对于在要求表重写的DDL操作期间被写入的新关系，这个域包含原始关系的OID，否则为0。那种状态仅在内部可见，对于一个用户可见的关系这个域应该从不包含不是0的值。
relfrozenxid	xid		在此之前的所有事务ID在表中已经被替换为一个永久的（“冻结的”） 事务ID。这用于跟踪表是否需要被清理，以便阻止事务ID回卷或者允许pg_xact被收缩。如果该关系不是一个表

名称	类型	引用	描述
			则为0（InvalidTransactionId）。
relminmxid	xid		在此之前的多事务ID在表中已经被替换为一个事务ID。这被用于跟踪表是否需要被清理，以阻止 多事务ID回卷或者允许pg_multixact被收缩。如果关系不是一个表则 为0（InvalidMultiXactId）。
relacl	aclitem[]		访问权限，更多信息参见 GRANT和 REVOKE
reloptions	text[]		访问方法相关的选项，以“keyword=value”字符串形式
relpartbound	pg_node_tree		如果表示一个分区（见relispartition），分区边界的内部表达

pg_class中的一些逻辑标志被以一种懒惰的方式维护：在正确状态时它们被保证为真，但是当条件不再为真时它们并不会被立刻重置为假。例如，relhasindex由CREATE INDEX设置，但它从不会被DROP INDEX清除。作为替代，VACUUM会在找到无索引表后清除其relhasindex。这种安排避免了竞争条件并且提高了并发性。

52.12. pg_collation

目录pg_collation描述了可用的排序规则，其本质是从一个SQL名字到操作系统locale分类的映射。更多信息参见第 23.2 节

表 52.12.pg_collation的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
collname	name		排序规则名字（在每一个名字空间和编码中唯一）
collnamespace	oid	pg_namespace.oid	包含该排序规则的名字空间的OID
collowner	oid	pg_authid.oid	排序规则的拥有者
collprovider	char		排序规则的提供者：d = 数据库默认，c = libc，i = icu
collencoding	int4		该排序规则可应用的编码，-1表示它可用于任何编码
collcollate	name		该排序规则对象的LC_COLLATE

名称	类型	引用	描述
collctype	name		该排序规则对象的LC_CTYPE
collversion	text		排序规则的提供者相关的版本。这是在排序规则创建时记录下来的，并且在使用排序规则时会被检查以检测可能导致数据损坏的排序规则定义的改变。

注意在这个目录中的唯一键是（collname、 collencoding、 collencoding）， 不仅仅是（collname，collencoding）。 所有collencoding不等于当前数据库编码或-1的编码规则通常都会被PostgreSQL忽略，且禁止创建和collencoding = -1的项重名的项。因此使用一个受限的SQL名字（schema.name）来标识一个排序规则是足够的，即使这根据目录定义是不唯一的。以这种方式定义这个目录的原因是initdb会在集簇初始化时使用系统上所有可用的locale填充这个目录，所以它必须能够为所有可能在集簇中使用的编码保持项。

在template0数据库中，创建与数据库编码不匹配的编码是有用的，因为它们可以匹配后面从template0克隆的数据库的编码。这在目前必须手动完成。

52.13. pg_constraint

目录pg_constraint存储表上的检查、主键、唯一、外键和排他约束（列约束也不会被特殊对待。每一个列约束都等同于某种表约束。）。非空约束不在这里，而是在pg_attribute目录中表示。

用户定义的约束触发器（使用CREATE CONSTRAINT TRIGGER创建）也会在这个表中产生一项。

域上的检查约束也存储在这里。

表 52.13.pg_constraint的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
conname	name		约束名字（不需要唯一！）
connamespace	oid	pg_namespace.oid	包含此约束的名字空间的OID
contype	char		c = 检查约束， f = 外键约束， p = 主键约束， u = 唯一约束， t = 约束触发器， x = 排他约束
condeferrable	bool		该约束是否能被延迟？
condeferred	bool		该约束是否默认被延迟？
convalidated	bool		此约束是否被验证过？当前对于外键和检查约束只能是假

名称	类型	引用	描述
conrelid	oid	pg_class.oid	该约束所在的表，如果不是表约束则为0
contypid	oid	pg_type.oid	该约束所在的域，如果不是域约束则为0
conindid	oid	pg_class.oid	如果该约束是唯一、主键、外键或排他约束，此列表示支持此约束的索引，否则为0
conparentid	oid	pg_constraint.oid	如果这是一个分区中的约束，则是父分区表中对应的约束；否则为0
confrelid	oid	pg_class.oid	如果此约束是一个外键约束，此列为被引用的表，否则为0
confupdtype	char		外键更新动作代码：a = 无动作， r = 限制， c = 级联， n = 置空， d = 置为默认值
confdeltype	char		外键删除动作代码：a = 无动作， r = 限制， c = 级联， n = 置空， d = 置为默认值
confmatchtype	char		外键匹配类型： f = 完全， p = 部分， s = 简单
conislocal	bool		此约束是定义在关系本地。注意一个约束可以同时是本地定义和继承。
coninhcount	int4		此约束的直接继承祖先数目。一个此列非零的约束不能被删除或重命名。
connoinherit	bool		为真表示此约束被定义在关系本地。它是一个不可继承约束。
conkey	int2[]	pg_attribute.attnum	如果是一个表约束（包括外键但不包括约束触发器），此列是被约束列的列表
confkey	int2[]	pg_attribute.attnum	如果是一个外键，此列是被引用列的列表
conpfeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于PK = FK比较的等值操作符的列表
conppeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于PK = PK比较的等值操作符的列表

名称	类型	引用	描述
conffeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于FK = FK比较的等值操作符的列表
conexclop	oid[]	pg_operator.oid	如果是一个排他约束，此列是没列排他操作符的列表
conbin	pg_node_tree		如果是一个检查约束，此列是表达式的一个内部表示
consrc	text		如果是一个检查约束，此列是表达式的一个人类可读的表示

在一个排他约束的情况中，conkey只对约束元素是单一列引用时有用。对于其他情况，conkey为0且必须查阅相关索引来发现被约束的表达式（conkey因此和pg_index.indkey具有相同的内容）。

> **注意**
>
> 当被引用对象改变时，consrc不能被更新。例如，它不跟踪列的重命名。最好使用pg_get_constraintdef()来抽取一个检查约束的定义，而不是依赖这个域。

> **注意**
>
> pg_class.relchecks需要和每个关系在此目录中的检查约束数量保持一致。

52.14. pg_conversion

目录pg_conversion描述编码转换函数。更多信息参见CREATE CONVERSION。

表 52.14.pg_conversion的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
conname	name		转换的名字（在一个名字空间内唯一）
connamespace	oid	pg_namespace.oid	包含此转换的名字空间的OID
conowner	oid	pg_authid.oid	转换的拥有者
conforencoding	int4		源编码ID
contoencoding	int4		目标编码ID
conproc	regproc	pg_proc.oid	转换函数
condefault	bool		如果这是默认转换则为真

52.15. pg_database

目录pg_database存储有关可用数据库的信息。数据库通过CREATE DATABASE命令创建。 更多关于其参数的信息请查阅第 22 章

和大部分系统目录不同，pg_database是在集簇的所有数据库之间共享的：在一个集簇中只有一份pg_database拷贝，而不是每个数据库一份。

表 52.15.pg_database的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
datname	name		数据库名字
datdba	oid	pg_authid.oid	数据库的拥有者，通常是创建它的用户
encoding	int4		此数据库的字符编码的编号（pg_encoding_to_char()可将此编号转换成编码的名字）
datcollate	name		此数据库的LC_COLLATE
datctype	name		此数据库的LC_CTYPE
datistemplate	bool		如果为真，则此数据库可被任何具有CREATEDB特权的用户克隆；如果为假，则只有 超级用户或者该数据库的属主能够克隆它。
datallowconn	bool		如果为假则没有人能连接到这个数据库。这可以用来保护template0数据库不被修改。
datconnlimit	int4		设置能够连接到这个数据库的最大并发连接数。-1表示没有限制。
datlastsysoid	oid		数据库中最后一个系统OID，对pg_dump特别有用
datfrozenxid	xid		在此之前的所有事务ID在数据库中已经被替换为一个永久的（“冻结的”） 事务ID。这用于跟踪数据库是否需要被清理，以便组织事务ID回环或者允许pg_xact被收缩。它是此数据库中

名称	类型	引用	描述
			所有表的pg_class.relfrozenxid值的最小值。
datminmxid	xid		在此之前的所有多事务ID在数据库中已经被替换为一个事务ID。这用于跟踪数据库是否需要被 清理，以便组织事务ID回环或者允许pg_multixact被收缩。它是此数据库中所有表的pg_class.relminmxid值的最小值。
dattablespace	oid	pg_tablespace.oid	此数据库的默认表空间。在此数据库中，所有pg_class.reltablespace为0的表都将被存储在这个表空间中，尤其是非共享系统目录都会在其中。
datacl	aclitem[]		访问权限，更多信息参见 GRANT和 REVOKE

52.16. pg_db_role_setting

目录 pg_db_role_setting为每一个角色和数据库组合记录被设置到运行时配置变量的默认值。

和大部分系统目录不同，pg_db_role_setting是在集簇的所有数据库之间共享的：在一个集簇中只有一份pg_db_role_setting拷贝，而不是每个数据库一份。

表 52.16.pg_db_role_setting的列

名称	类型	引用	描述
setdatabase	oid	pg_database.oid	此设置可用的数据库OID，如果不与具体数据库相关则为0
setrole	oid	pg_authid.oid	此设置可用的角色OID，如果不与具体角色相关则为0
setconfig	text[]		运行时配置变量的默认值

52.17. pg_default_acl

目录pg_default_acl存储要被分配给新创建对象的初始权限。

表 52.17.pg_default_acl的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
defaclrole	oid	pg_authid.oid	与此项相关的角色的OID
defaclnamespace	oid	pg_namespace.oid	与此项相关的名字空间的OID，如果没有则为0
defaclobjtype	char		此项适合的对象类型： r = 关系（表、视图）， S = 序列， f = 函数， T = 类型， n = 方案
defaclacl	aclitem[]		此类对象在创建时应用有的访问权限

一个pg_default_acl项展示了要分配给属于一个指定用户的对象的初始权限。当前有两类项：defaclnamespace = 0的“全局”项和引用一个特殊模式的“每方案”项。如果一个全局项存在，则它重载该对象类型的普通hard-wired默认权限。一个每模式项如果存在，表示权限将被加入到全局或hard-wired默认权限中。

注意当在另一个表中的一个ACL项为空时，它用来表示其对象的hard-wired默认权限，而不是当时可能在pg_default_acl中的任何权限。只有在对象创建期间才会查阅pg_default_acl。

52.18. pg_depend

目录pg_depend记录数据库对象之间的依赖关系。这些信息允许DROP命令查找必须被DROP CASCADE删除的其他对象，或者在DROP RESTRICT情况下阻止删除。

另请参阅pg_shdepend，它对在一个数据库集簇中共享的对象之间的依赖提供了相似的功能。

表 52.18.pg_depend的列

名称	类型	引用	描述
classid	oid	pg_class.oid	依赖对象所在的系统目录OID
objid	oid	任意OID列	指定依赖对象的OID
objsubid	int4		对于一个表列，这里是列号（objid和classid指表本身）。对于所有其他对象类型，此列为0。
refclassid	oid	pg_class.oid	被引用对象所在的系统目录的OID
refobjid	oid	任意OID列	指定被引用对象的OID
refobjsubid	int4		对于一个表列，这里是列号

名称	类型	引用	描述
			（refobjid和refclassid指表本身）。对于所有其他对象类型，此列为0。
deptype	char		定义此依赖关系语义的一个代码，见文本

在所有情况下，一个pg_depend项表明被引用对象不能在没有删除其依赖对象的情况下被删除。但是，其中也有多种依赖类型，由deptype标识：

DEPENDENCY_NORMAL (n)

在独立创建的对象之间的一个普通关系。依赖对象可以在不影响被依赖对象的情况下被删除。被引用对象只能通过指定CASCADE被删除，在这种情况下依赖对象也会被删除。例子：一个表列对于其数据类型有一个普通依赖。

DEPENDENCY_AUTO (a)

依赖对象可以被独立于被依赖对象删除，且应该在被引用对象被删除时自动被删除（不管在RESTRICT或CASCADE模式）。例子：一个表上的一个命名约束应该被设置为自动依赖于表，这样在表被删除后它也会消失。

DEPENDENCY_INTERNAL (i)

依赖对象作为被引用对象创建过程的一部分被创建，并且只是其内部实现的一部分。一个依赖对象的DROP会被直接拒绝（作为替代，我们将告诉用户发出一个针对被引用对象的DROP）。不管是否指定CASCADE，一个被引用对象的DROP将被传播来删除其依赖对象。例子：一个用于强制外键约束的触发器将被设置为内部依赖于其约束的pg_constraint项。

DEPENDENCY_INTERNAL_AUTO (I)

依赖对象被作为被引用对象创建过程的一部分创建，并且确实是其内部实现的一部分。依赖对象的DROP将彻底被禁止（我们将告诉用户应该在被引用对象上发出DROP）。虽然一个常规的内部依赖将阻止依赖对象在任何这类依赖还存在的情况下被删除，当DEPENDENCY_INTERNAL_AUTO会允许这类删除，只要能顺着任一这类依赖找到该对象就行。例子：一个分区上的索引对分区本身和父分区表上的所有都有内部自动依赖，那么该分区索引会被和它索引的分区一起删除，或者和它所附着的父索引一起被删除。

DEPENDENCY_EXTENSION (e)

依赖对象是作为扩展的被引用对象的一个成员（参见pg_extension）。依赖对象可以通过被引用对象上的DROP EXTENSION来删除。在功能上，这种依赖类型和一个内部依赖的作用相同，其存在只是为了清晰和简化pg_dump。

DEPENDENCY_AUTO_EXTENSION (x)

依赖对象不是作为被引用对象的扩展的成员（因此不应该被 pg_dump 忽略），但是没有被引用对象又无法工作，所以当扩展本身被删除时也应该把该依赖对象删除。该依赖对象也可以自己被删除。

DEPENDENCY_PIN (p)

没有依赖对象，这种类型的项是一个信号，用于说明系统本身依赖于被引用对象，并且该对象永远不能被删除。这种类型的项只能被initdb创建。而此种项的依赖对象的列都为0。

在未来可能会需要其他依赖类型。

52.19. pg_description

目录pg_description存储对每一个数据库对象可选的描述（注释）。描述可以通过COMMENT操作，并可使用psql的\d命令查看。在pg_description的初始内容中提供了很多内建系统对象的描述。

参见pg_shdescription，它对在一个数据库集簇中共享的对象的描述提供了相似的功能。

表 52.19.pg_description的列

名称	类型	引用	描述
objoid	oid	任意OID列	描述所属对象的OID
classoid	oid	pg_class.oid	对象所述的系统目录的OID
objsubid	int4		对于一个表列上的一个注释，这里是列号（objoid和classoid指表本身）。对所有其他对象类型，此列为0。
description	text		作为该对象描述的任意文本

52.20. pg_enum

pg_enum目录包含每一个枚举类型的项，其中包括了值和标签。一个给定枚举值的内部表示实际上是它在pg_enum中的相关行的OID。

表 52.20.pg_enum的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
enumtypid	oid	pg_type.oid	包含此枚举值的pg_type项的OID
enumsortorder	float4		此枚举值在其枚举类型中的排序位置
enumlabel	name		此枚举值的文本标签

pg_enum行的OID值遵循一种特殊的规则：即OID的数值被保证按照其枚举类型的排序顺序进行排序。即如果两个偶数OID属于同一枚举类型，较小的OID必然具有较小的enumsortorder值。奇数OID值不需要遵循排序顺序。这种规则使得枚举比较例程在很多常见情况下可以避免系统目录查找。创建和修改枚举类型的例程将尝试尽可能地为枚举值分配偶数OID。

当一个枚举类型被创建后，其成员会被分配排序位置1..n。但后面增加的成员可能会被分配负值或者分数值的enumsortorder。对于这些值的唯一要求是它们必须被正确地排序且和保持唯一。

52.21. pg_event_trigger

目录pg_event_trigger存储事件触发器。更多信息参见第 40 章

表 52.21.pg_event_trigger的列

名称	类型	引用	描述
evtname	name		触发器名（必须唯一）
evtevent	name		此触发器触发的事件的标识符
evtowner	oid	pg_authid.oid	事件触发器的拥有者
evtfoid	oid	pg_proc.oid	将被调用的函数
evtenabled	char		控制事件触发器触发的session_replication_role模式。 O = 触发器在“origin”和“local”模式触发， D = 触发器被禁用， R = 触发器在“replica”模式触发， A = 触发器总是触发。
evttags	text[]		此触发器将触发的命令标签。如果为空，此触发器的触发不受命令标签的限制。

52.22. pg_extension

目录pg_extension存储有关已安装扩展的信息。有关扩展的细节请参见第 38.16 节

表 52.22.pg_extension的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
extname	name		扩展的名字
extowner	oid	pg_authid.oid	扩展的拥有者
extnamespace	oid	pg_namespace.oid	包含此扩展的导出对象的模式
extrelocatable	bool		如果扩展可被重定位到另一个模式则为真
extversion	text		扩展的版本名字
extconfig	oid[]	pg_class.oid	扩展的配置表的regclass项的OID数组，如果没有配置表则为NULL
extcondition	text[]		扩展的配置表的WHERE子句过滤条件的数组，如果没有则为NULL

注意和大部分具有一个“namespace”列的模式不同，extnamespace不是用来表示扩展属于该模式。扩展的名字从不用模式进行限定。extnamespace表明该模式包含了该扩展的大部分或全部对象。如果extrelocatable为真，则该模式事实上必须包含属于此扩展的全部模式限定的对象。

52.23. pg_foreign_data_wrapper

目录pg_foreign_data_wrapper存储外部数据包装器定义。外部数据包装器是一种访问位于外部服务器上数据的机制。

表 52.23.pg_foreign_data_wrapper的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
fdwname	name		外部数据包装器的名字
fdwowner	oid	pg_authid.oid	外部数据包装器的拥有者
fdwhandler	oid	pg_proc.oid	指一个负责为外部数据包装器提供执行例程的处理函数。如果没有提供处理函数则为0
fdwvalidator	oid	pg_proc.oid	指一个负责检查传给外部数据包装器的选项的有效性的验证函数，包括外部服务器选项以及使用外部数据包装器的用户映射。如果没有提供验证函数则为0
fdwacl	aclitem[]		访问权限，详见GRANT和REVOKE
fdwoptions	text[]		外部数据包装器特定选项，以“keyword=value”字符串形式

52.24. pg_foreign_server

目录pg_foreign_server存储外部服务器定义。外部服务器定义了外部数据的来源，例如一个远程服务器。外部服务器通过外部数据包装器来访问。

表 52.24.pg_foreign_server的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
srvname	name		外部服务器的名字
srvowner	oid	pg_authid.oid	外部服务器的拥有者
srvfdw	oid	pg_foreign_data_wrapper.oid	此外部服务器的外部数据包装器的OID
srvtype	text		服务器的类型（可选）

名称	类型	引用	描述
srvversion	text		服务器的版本（可选）
srvacl	aclitem[]		访问权限，详见GRANT和REVOKE
srvoptions	text[]		外部服务器特定选项，以“keyword=value”字符串形式

52.25. pg_foreign_table

目录pg_foreign_table包含关于外部表的辅助信息。一个外部表和普通表一样，主要由一个pg_class项表示。它的pg_foreign_table项包含外部表所特有的信息。

表 52.25.pg_foreign_table的列

名称	类型	引用	描述
ftrelid	oid	pg_class.oid	外部表的pg_class项的OID
ftserver	oid	pg_foreign_server.oid	外部表所在的外部服务器的OID
ftoptions	text[]		外部表选项，以“keyword=value”字符串形式

52.26. pg_index

目录pg_index包含关于索引的部分信息。其他信息大部分在pg_class中。

表 52.26.pg_index的列

名称	类型	引用	描述
indexrelid	oid	pg_class.oid	此索引的pg_class项的OID
indrelid	oid	pg_class.oid	此索引的基表的pg_class项的OID
indnatts	int2		索引中的总列数（与pg_class.relnatts重复），这个数目包括键和被包括的属性
indnkeyatts	int2		索引中键列的编号，不计入任何的内含列，它们只是被存储但不参与索引的语义
indisunique	bool		表示是否为唯一索引
indisprimary	bool		表示索引是否表示表的主键（如果此列为真，indisunique也总是为真）
indisexclusion	bool		表示索引是否支持一个排他约束

名称	类型	引用	描述
indimmediate	bool		表示唯一性检查是否在插入时立即被执行（如果indisunique为假，此列无关）
indisclustered	bool		如果为真，表示表最后以此索引进行了聚簇
indisvalid	bool		如果为真，此索引当前可以用于查询。为假表示此索引可能不完整：它肯定还在被INSERT/UPDATE操作所修改，但它不能安全地被用于查询。如果索引是唯一索引，唯一性属性也不能被保证。
indcheckxmin	bool		如果为真，直到此pg_index行的xmin低于查询的TransactionXmin视界之前，查询都不能使用此索引，因为表可能包含具有它们可见的不相容行的损坏HOT链
indisready	bool		如果为真，表示此索引当前可以用于插入。为假表示索引必须被INSERT/UPDATE操作忽略。
indislive	bool		如果为假，索引正处于被删除过程中，并且必须被所有处理忽略（包括HOT安全的决策）
indisreplident	bool		如果为真，这个索引被选择为使用ALTER TABLE ... REPLICA IDENTITY USING INDEX...的“replica identity”
indkey	int2vector	pg_attribute.attnum	这是一个indnatts值的数组，它表示了此索引索引的表列。例如一个1 3值可能表示表的第一和第三列组成了索引项。键列出现在非键（内含）列前面。数组中的一个0表示对应的索引属性是一个在表列上的表

名称	类型	引用	描述
			达式，而不是一个简单的列引用。
indcollation	oidvector	pg_collation.oid	对于索引键（indnkeyatts值）中的每一列，这包含要用于该索引的排序规则的OID，如果该列不是一种可排序数据类型则为零。
indclass	oidvector	pg_opclass.oid	对于索引键中的每一列（indnkeyatts值），这里包含了要使用的操作符类的OID。详见pg_opclass。
indoption	int2vector		这是一个indnkeyatts值的数组，用于存储每列的标志位。位的意义由索引的访问方法定义。
indexprs	pg_node_tree		非简单列引用索引属性的表达式树（以nodeToString()形式）。对于indkey中每一个为0的项，这个列表中都有一个元素。如果所有的索引属性都是简单引用，此列为空。
indpred	pg_node_tree		部分索引谓词的表达式树（以nodeToString()形式）。如果不是部分索引，此列为空。

52.27. pg_inherits

目录pg_inherits记录有关表继承层次的信息。数据库中每一个直接父子关系在这里都有一项（非直接继承可以通过顺着项构成的链来决定）。

表 52.27.pg_inherits的列

名称	类型	引用	描述
inhrelid	oid	pg_class.oid	孩子表的OID
inhparent	oid	pg_class.oid	父表的OID
inhseqno	int4		如果一个孩子表有多于一个直接父表（多继承），这个数字说明了继承列被排列的顺序。计数从1开始。

52.28. pg_init_privs

目录pg_init_privs记录系统中对象的初始特权。数据库中每一个具有非默认（非-NULL）初始特权集合的对象都有一个条目在其中。

对象可以在系统初始化（initdb）时获得其初始特权，也可以在CREATE EXTENSION期间创建该对象并且在扩展脚本中用GRANT来设置对象的初始特权。注意系统将自动处理扩展脚本执行期间对特权的记录，扩展的作者们只需要在他们的脚本中使用GRANT以及REVOKE语句以便特权被记录下来。privtype列表示初始特权是被initdb设置还是在一次CREATE EXTENSION命令期间被设置。

具有被initdb设置的初始特权的对象的条目中privtype是'i'，而具有被CREATE EXTENSION设置的初始特权的对象的条目中privtype为'e'。

表 52.28.pg_init_privs列

名称	类型	引用	描述
objoid	oid	任何 OID 列	指定对象的 OID
classoid	oid	pg_class.oid	对象所在的系统目录的 OID
objsubid	int4		对于一个表列，这里是列编号（objoid和classoid指向表本身）。对于所有其他对象类型，这列为零。
privtype	char		定义这个对象初始特权类型的代码，见文字说明
initprivs	aclitem[]		初始的访问特权，详见GRANT和REVOKE

52.29. pg_language

目录pg_language注册了可用于编写函数或存储过程的语言。 更多关于语言处理器的信息请参阅CREATE LANGUAGE和第 42 章

表 52.29.pg_language的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
lanname	name		语言的名字
lanowner	oid	pg_authid.oid	语言的拥有者
lanispl	bool		内部语言为假（如SQL），用户定义语言为真。当前，pg_dump仍然使用这个列来决定要转储哪些语言，但在未来这可能会被一种不同的机制所取代。

名称	类型	引用	描述
lanpltrusted	bool		为真表示这是一种可信的语言，即它被相信不会向普通SQL执行环境之外的任何东西授予权限。只有超级用户可以在非可信语言中创建函数。
lanplcallfoid	oid	pg_proc.oid	对于非内部语言，此列引用语言处理器，它是一个特殊函数负责执行所有用这种语言编写的函数
laninline	oid	pg_proc.oid	此列引用一个负责执行“内联”匿名代码块的函数（DO 块）。如果不支持内联块则为0。
lanvalidator	oid	pg_proc.oid	此列引用一个负责在函数创建时对其进行语法和可用性检查的语言验证函数。如果没有提供验证器则为0。
lanacl	aclitem[]		访问权限，详情参见GRANT和REVOKE

52.30. pg_largeobject

目录pg_largeobject保存构成“大对象”的数据。一个大对象在被创建时会被分配一个OID。每个大对象被分解成段或“页”，以便小到可以被方便地作为行存储在pg_largeobject中。每页中的数据量被定义为LOBLKSIZE（目前是BLCKSZ/4或是2 kB）。

在PostgreSQL 9.0之前，大对象没有相关的权限结构。作为结果，pg_largeobject是公共可读的并且可以用来获得系统中所有大对象的OID（和内容）。但现在不是这样了，可使用pg_largeobject_metadata来获得大对象OID的列表。

表 52.30.pg_largeobject的列

名称	类型	引用	描述
loid	oid	pg_largeobject_metadata.oid	包含此页的大对象的标识符
pageno	int4		此页在它所属大对象中的页号（从0开始计）
data	bytea		实际存储在大对象中的数据。它从不会超过LOBLKSIZE字节，也可能更少。

pg_largeobject的每一行保存一个大对象的一个页的数据，从对象内部的字节偏移量（pageno * LOBLKSIZE）开始。现在的实现允许稀疏存储：页面可能丢失，并且可能比LOBLKSIZE字节短（即便不是最后一页）。一个大对象中丢失的区域会被读出为0。

52.31. pg_largeobject_metadata

目录pg_largeobject_metadata保持着与大对象有关的元数据。真正的大对象数据被存储在pg_largeobject中。

表 52.31.pg_largeobject_metadata的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
lomowner	oid	pg_authid.oid	大对象的拥有者
lomacl	aclitem[]		访问权限，详见GRANT和REVOKE

52.32. pg_namespace

目录pg_namespace存储名字空间。名字空间是SQL模式之下的结构：每个名字空间拥有一个独立的表、类型等的集合，且其中没有名字冲突。

表 52.32.pg_namespace的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
nspname	name		名字空间的名字
nspowner	oid	pg_authid.oid	名字空间的拥有者
nspacl	aclitem[]		访问权限，详见GRANT和REVOKE

52.33. pg_opclass

目录pg_opclass定义索引访问方法的操作符类。每一个操作符类定义了一种特定数据类型和一种特定索引访问方法的索引列的语义。一个操作符类实际上指定了一个特定的操作符族可以用于一个特定可索引列数据类型。该族中可用于索引列的操作符能够接受该列的数据类型作为它们的左输入。

操作符类详见第 38.15 节

表 52.33.pg_opclass的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
opcmethod	oid	pg_am.oid	操作符类所属的索引访问方法
opcname	name		操作符类的名称
opcnamespace	oid	pg_namespace.oid	操作符类所属的名字空间
opcowner	oid	pg_authid.oid	操作符类的拥有者

名称	类型	引用	描述
opcfamily	oid	pg_opfamily.oid	包含此操作符类的操作符族
opcintype	oid	pg_type.oid	操作符类索引的数据类型
opcdefault	bool		如果此操作符类为opcintype的默认值则为真
opckeytype	oid	pg_type.oid	存储在索引中的数据的类型，如果值为0表示与opcintype相同

一个操作符类的opcmethod必须匹配包含它的操作符族的opfmethod。而且，对于任何给定的opcmethod和opcintype组合，只有不超过一个pg_opclass行的opcdefault值为真。

52.34. pg_operator

目录pg_operator存储关于操作符的信息。详见CREATE OPERATOR和第 38.13 节

表 52.34.pg_operator的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
oprname	name		操作符的名称
oprnamespace	oid	pg_namespace.oid	操作符所属的名字空间的OID
oprowner	oid	pg_authid.oid	操作符的拥有者
oprkind	char		b = 中缀（“并”），l = 前缀（“左”），r = 后缀（“右”）
oprcanmerge	bool		该操作符是否支持归并连接
oprcanhash	bool		该操作符是否支持哈希连接
oprleft	oid	pg_type.oid	左操作数类型
oprright	oid	pg_type.oid	右操作数类型
oprresult	oid	pg_type.oid	结果类型
oprcom	oid	pg_operator.oid	该操作符的交换子（如果存在）
oprnegate	oid	pg_operator.oid	该操作符的否定（如果存在）
oprcode	regproc	pg_proc.oid	实现该操作符的函数
oprrest	regproc	pg_proc.oid	该操作符的限制选择度估算函数
oprjoin	regproc	pg_proc.oid	该操作符的连接选择度估算函数

未用的列包含零值。例如，一个前缀操作符的oprleft为0。

52.35. pg_opfamily

目录pg_opfamily定义了操作符族。每一个操作符族是操作符和相关支持例程的集合，支持例程用于实现一个特定索引访问方法的语义。此外，按照访问方法指定的某种方式，一个族内的操作符都是“兼容的”。操作符族概念允许在索引中使用跨数据类型操作符，并可以使用访问方法语义的知识推导出。

操作符族最终在第 38.15 节描述。

表 52.35.pg_opfamily的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
opfmethod	oid	pg_am.oid	操作符族适用的索引访问方法
opfname	name		操作符族的名字
opfnamespace	oid	pg_namespace.oid	操作符族所属的名字空间
opfowner	oid	pg_authid.oid	操作符族的拥有者

定义操作符族的主要信息不在它的pg_opfamily行，而是在相关的pg_amop、pg_amproc和pg_opclass行中。

52.36. pg_partitioned_table

目录pg_partitioned_table存放有关表如何被分区的信息。

表 52.36.pg_partitioned_table列

名称	类型	引用	描述
partrelid	oid	pg_class.oid	这个分区表的pg_class项的OID
partstrat	char		分区策略；h = 哈希分区表，l = 列表分区表，r = 范围分区表
partnatts	int2		分区键中的列数
partdefid	oid	pg_class.oid	这个分区表的默认分区的pg_class项的OID，如果这个分区表没有默认分区则为零。
partattrs	int2vector	pg_attribute.attnum	这是一个长度为partnatts值的数组，它指示哪些表列是分区键的组成部分。例如，值1 3表示第一个和第三个表列组成了分区键。这个数组中的零表示对应

名称	类型	引用	描述
			的分区键列是一个表达式而不是简单的列引用。
partclass	oidvector	pg_opclass.oid	对于分区键中的每一个列，这个域包含要使用的操作符类的OID。详见pg_opclass。
partcollation	oidvector	pg_opclass.oid	对于分区键中的每一个列，这个域包含要用于分区的排序规则的OID，如果该列不是一种可排序数据类型则为零。
partexprs	pg_node_tree		非简单列引用的分区键列的表达式树（以nodeToString()的表达方式）。这是一个列表，partattrs中每一个零项都有一个元素。如果所有分区键列都是简单列引用，则这个域为空。

52.37. pg_pltemplate

目录pg_pltemplate存储了过程语言的“模板”信息。一个语言的模板允许我们在一个特定数据库中以一个简单的CREATE LANGUAGE命令创建语言，而不需要指定实现细节。

和大部分系统目录不同，pg_pltemplate是在集簇的所有数据库之间共享的：在一个集簇中只有一份pg_pltemplate拷贝，而不是每个数据库一份。这使得在每个需要的数据库中都可以访问该信息。

表 52.37.pg_pltemplate的列

名称	类型	描述
tmplname	name	该模板适用的语言名字
tmpltrusted	boolean	如果语言被认为是可信的则为真
tmpldbacreate	boolean	如果语言可以被一个数据库拥有者创建则为真
tmplhandler	text	调用处理函数的名字
tmplinline	text	匿名阻塞处理函数的名字，如果没有则为空
tmplvalidator	text	验证函数的名字，如果没有则为空
tmpllibrary	text	实现语言的共享库的路径
tmplacl	aclitem[]	模板的访问权限（并未真正使用）

目前任何命令都不能操纵过程语言模板。要改变内建信息，超级用户必须使用普通的INSERT、DELETE或UPDATE命令修改该表。

> **注意**
>
> 在PostgreSQL的某个未来版本的发布中，pg_pltemplate很有可能会被移除，而这些关于过程语言的知识可能会保持在它们相应的扩展安装脚本中。

52.38. pg_policy

目录pg_policy存储着表的行级安全性策略。 一个策略包括它适用于的命令种类（可能适用于所有命令）、它适用于的角色、 被作为安全屏障条件增加到包括该表的查询的表达式以及被作为 WITH CHECK选项增加到尝试向表增加新纪录的查询的表达式。

表 52.38.pg_policy列

名称	类型	引用	描述
polname	name		策略的名称
polrelid	oid	pg_class.oid	策略适用的表
polcmd	char		策略适用的命令类型： r表示SELECT，a表示INSERT， w表示UPDATE， d表示DELETE， *表示所有命令类型
polpermissive	boolean		策略是宽容性的还是限制性的?
polroles	oid[]	pg_authid.oid	策略适用的角色
polqual	pg_node_tree		被作为安全屏障条件增加到使用该表的查询的表达式树
polwithcheck	pg_node_tree		被作为WITH CHECK 条件增加到尝试向表增加行的查询的表达式树

> **注意**
>
> 存储在pg_policy中的策略只有在它们所适用的表的pg_class.relrowsecurity被设置时才其作用。

52.39. pg_proc

目录pg_proc存放有关函数、过程、聚集函数以及窗口函数（共称为例程）的信息。更多信息请参考CREATE FUNCTION、CREATE PROCEDURE和第 38.3 节

如果proisagg为真，则该项是一个聚集函数，在pg_aggregate中应该有一个相匹配的行。

表 52.39.pg_proc的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）

名称	类型	引用	描述
proname	name		函数的名字
pronamespace	oid	pg_namespace.oid	函数所属的名字空间的OID
proowner	oid	pg_authid.oid	函数的拥有者
prolang	oid	pg_language.oid	实现语言或该函数的调用接口
procost	float4		估计的执行代价（以cpu_operator_cost为单位），如果proretset为真，这是每行返回的代价
prorows	float4		估计的结果行数量（如果proretset为假，该值为0）
provariadic	oid	pg_type.oid	可变数组参数的元素的数据类型，如果函数没有可变参数则为0
protransform	regproc	pg_proc.oid	调用该函数时可以通过此列指定的函数来简化（见第 38.10.10 节
prokind	char		f表示普通函数，p表示过程，a表示聚集函数，w表示窗口函数
prosecdef	bool		函数是一个安全性定义者（即，一个“setuid”函数）
proleakproof	bool		该函数没有副作用。除了通过返回值，没有关于参数的信息被传播。任何会抛出基于其参数值的错误信息的函数都不是泄露验证的。
proisstrict	bool		当任意调用函数为空时，函数是否会返回空值。在那种情况下函数实际上根本不会被调用。非“strict”函数必须准备好处理空值输入。
proretset	bool		函数是否返回一个集合（即，指定数据类型的多个值）
provolatile	char		provolatile说明函数是仅仅只依赖于它的输入参数，还是会被外部因素影响。值i表示“不变的”函数，它对于相同的输入总

名称	类型	引用	描述
			是输出相同的结果。值s表示“稳定的”函数，它的结果（对于固定输入）在一次扫描内不会变化。值v表示“不稳定的”函数，它的结果在任何时候都可能变化（使用v页表示函数具有副作用，所以对它们的调用无法得到优化）
proparallel	char		proparallel说明该函数在并行模式下是否能安全地运行。对于能在并行模式下不受限制安全运行的函数，这列是s。对于可以在并行模式下运行但是只限于由并行分组的领导者执行的函数，这列是r。对于在并行模式中不安全的函数，这列是u，这种函数的存在会强制一个顺序执行计划。
pronargs	int2		输入参数的个数
pronargdefaults	int2		具有默认值的参数个数
prorettype	oid	pg_type.oid	返回值的数据类型
proargtypes	oidvector	pg_type.oid	一个函数参数的数据类型的数组。这只包括输入参数（含INOUT和VARIADIC参数），因此也表现了函数的调用特征。
proallargtypes	oid[]	pg_type.oid	一个函数参数的数据类型的数组。这包括所有参数（含OUT和INOUT参数）。但是，如果所有参数都是IN参数，这个域将为空。注意下标是从1开始 ，然而由于历史原因proargtypes的下标是从0开始。
proargmodes	char[]		一个函数参数的模式的数组。编码为：i表示IN参数 ， o表示OUT参数， b表示INOUT参数， v表示VARIADIC参数，t表示TABLE参数。 如果所有的参数都

名称	类型	引用	描述
			是IN参数，这个域为空。注意这里的下标对应着proallargtypes而不是proargtypes中的位置。
proargnames	text[]		一个函数参数的名字的数组。没有名字的参数在数组中设置为空字符串。如果没有一个参数有名字，这个域为空。注意这里的下标对应着proallargtypes而不是proargtypes中的位置。
proargdefaults	pg_node_tree		默认值的表达式树（按照nodeToString()的表现方式）。这是一个pronargdefaults元素的列表，对应于最后N个input参数（即最后N个proargtypes位置）。如果没有一个参数具有默认值，这个域为空。
protrftypes	oid[]		要在其上应用转换的数据类型的 OID。
prosrc	text		这个域告诉函数处理者如何调用该函数。它可能是针对解释型语言的真实源码、一个符号链接、一个文件名或任何其他东西，这取决于实现语言/调用规范。
probin	text		关于如何调用函数的附加信息。其解释是与语言相关的。
proconfig	text[]		函数对于运行时配置变量的本地设置值
proacl	aclitem[]		访问权限，详见GRANT和REVOKE

对于编译好的函数，包括内建的和动态载入的，prosrc包含了函数的C语言名字（链接符号）。对于所有其他已知的语言类型，prosrc包含函数的源码文本。除了对于动态载入的C函数之外，probin是不被使用的。对于动态载入的C函数，它给定了包含该函数的共享库文件的名称。

52.40. pg_publication

目录pg_publication包含数据库中创建的所有publication。更多关于publication的内容请见第 31.1 节

表 52.40.pg_publication的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择）
pubname	name		publication的名称
pubowner	oid	pg_authid.oid	publication的拥有者
puballtables	bool		如果为真，这个publication自动包括数据库中的所有表，包括未来将会创建的任何表。
pubinsert	bool		如果为真，为publication中的表复制INSERT操作。
pubupdate	bool		如果为真，为publication中的表复制UPDATE操作。
pubdelete	bool		如果为真，为publication中的表复制DELETE操作。
pubtruncate	bool		如果为真，为publication中的表复制TRUNCATE操作。

52.41. pg_publication_rel

目录pg_publication_rel包含数据库中关系和publication之间的映射。这是一种多对多映射。这些信息对用户更加友好的视图请参考第 52.78 节

表 52.41.pg_publication_rel列

名称	类型	引用	描述
prpubid	oid	pg_publication.oid	对publication的引用
prrelid	oid	pg_class.oid	对关系的引用

52.42. pg_range

目录pg_range存储关于范围类型的信息。它是类型在pg_type中项的补充。

表 52.42.pg_range的列

名称	类型	引用	描述
rngtypid	oid	pg_type.oid	范围类型的OID
rngsubtype	oid	pg_type.oid	该范围类型的元素类型（子类型）的OID

名称	类型	引用	描述
rngcollation	oid	pg_collation.oid	用于范围比较的排序规则的OID，如果没有则为0
rngsubopc	oid	pg_opclass.oid	用于范围比较的子类型的操作符类的OID
rngcanonical	regproc	pg_proc.oid	将一个范围值转换为规范形式的函数的OID，如果没有则为0
rngsubdiff	regproc	pg_proc.oid	以双精度返回两个元素值不同的函数的OID，如果没有则为0

rngsubopc （加上rngcollation，如果元素类型是可排序的）决定了被该范围类型所使用的排序顺序。rngcanonical用于离散类型的元素类型。rngsubdiff是可选的，但是提供它可以提高范围类型上的GiST索引性能。

52.43. pg_replication_origin

pg_replication_origin目录包含所有已创建的复制源。更多复制源的信息请见第 50 章

和大部分系统目录不同，pg_replication_origin在一个集簇的所有数据库之间共享：每个集簇只有一份pg_replication_origin拷贝，而不是每个数据库一份。

表 52.43.pg_replication_origin的列

名称	类型	引用	描述
roident	oid		一个集簇范围内唯一的复制源标识符。应该绝不会脱离系统。
roname	text		外部的由用户定义的复制源名称。

52.44. pg_rewrite

目录pg_rewrite存储对于表和视图的重写规则。

表 52.44.pg_rewrite的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
rulename	name		规则名称
ev_class	oid	pg_class.oid	使用该规则的表
ev_type	char		使用该规则的事件类型：1 = SELECT，2 = UPDATE，3 = INSERT，4 = DELETE
ev_enabled	char		控制在哪种session_replication_role模式中触发该规则。 O = 规则

名称	类型	引用	描述
			在“origin”和“local”模式触发， D = 规则被禁用， R = 规则在“replica”模式触发， A = 规则总是被触发。
is_instead	bool		为真表示是一个INSTEAD规则
ev_qual	pg_node_tree		规则条件的表达式树（按照nodeToString()的表现形式）
ev_action	pg_node_tree		规则动作的查询树（按照nodeToString()的表现形式）

> **注意**
>
> 如果一个表在这个目录中有任何规则，pg_class.relhasrules必须为真。

52.45. pg_seclabel

目录pg_seclabel存储数据库对象上的安全标签。安全标签可以通过SECURITY LABEL命令操纵。简单的查看安全标签方法请见第 52.83 节

同时请见pg_shseclabel，它对集簇共享的数据库对象的安全标签执行相似的功能。

表 52.45.pg_seclabel的列

名称	类型	引用	描述
objoid	oid	任意OID列	该安全标签依附的对象的OID
classoid	oid	pg_class.oid	该对象所出现的系统目录的OID
objsubid	int4		对于一个在表列上的安全标签，这将是列号（objoid和classoid指表本身）。对于所有其他对象类型，本列为0。
provider	text		与该标签相关的标签提供者。
label	text		应用于该对象的安全标签。

52.46. pg_sequence

目录pg_sequence包含有关序列的信息。一些序列的信息（例如名称和方案）放在pg_class中。

表 52.46.pg_sequence的列

名称	类型	引用	描述
seqrelid	oid	pg_class.oid	这个序列的pg_class项的OID
seqtypid	oid	pg_type.oid	序列的数据类型
seqstart	int8		序列的起始值
seqincrement	int8		序列的增量值
seqmax	int8		序列的最大值
seqmin	int8		序列的最小值
seqcache	int8		序列的缓冲尺寸
seqcycle	bool		序列是否循环

52.47. pg_shdepend

目录pg_shdepend记录数据库对象和共享对象之间的依赖关系，例如角色。这些信息使得PostgreSQL可以确保对象在被删除时没有被其他对象引用。

另请参阅pg_depend，它对单个数据库中对象之间的依赖提供了相似的功能。

与大部分其他系统目录不同，pg_shdepend在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg_shdepend的拷贝，而不是每个数据库一份。

表 52.47.pg_shdepend的列

名称	类型	引用	描述
dbid	oid	pg_database.oid	依赖者对象所在的数据库OID，如果是一个共享对象则值为0
classid	oid	pg_class.oid	依赖者对象所在的系统目录的OID
objid	oid	任意OID列	依赖者对象的OID
objsubid	int4		对于一个表列，这将是列号（objid和classid指向表本身）。对于所有其他对象类型，该列值为0。
refclassid	oid	pg_class.oid	被引用对象所在的系统目录的OID（必须是一个共享的目录）
refobjid	oid	任意OID列	被引用对象的OID
deptype	char		定义该依赖关系的特定语义的代码，见表后的说明

在所有情况下，一个pg_shdepend项表明被引用对象不能在没有删除其依赖对象的情况下被删除。但是，其中也有多种依赖类型，由deptype标识：

SHARED_DEPENDENCY_OWNER (o)

被引用对象（必须是一个角色）是依赖对象的拥有者。

SHARED_DEPENDENCY_ACL (a)

被引用对象（必须是一个角色）在依赖对象的ACL（访问控制列表，即权限列表）中被提到。 （不会为对象的拥有者创建一个SHARED_DEPENDENCY_ACL项，因为拥有者将会有一个SHARED_DEPENDENCY_OWNER项。）

SHARED_DEPENDENCY_POLICY (r)

作为一个依赖策略对象的目标被引用的对象（必须是一个角色）。

SHARED_DEPENDENCY_PIN (p)

没有依赖对象，这种类型的项是一个信号，用来指示系统本身依赖于被引用对象，并且因此该对象必须永远不能被删除。这种类型的项只能被initdb创建。这种项中关于依赖对象的列值都为0。

未来可能会需要其他的依赖类型。特别要注意的是在当前定义中只支持角色作为被引用对象。

52.48. pg_shdescription

目录pg_shdescription存储共享数据库对象的可选描述（注释）。描述可以通过COMMENT命令操作，并且可以使用psql的\d命令来查看。

另请参阅pg_description，它对单个数据库中对象之间的依赖提供了相似的功能。

与大部分其他系统目录不同，pg_shdescription在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg_shdescription的拷贝，而不是每个数据库一份。

表 52.48.pg_shdescription的列

名称	类型	引用	描述
objoid	oid	任意OID列	该描述所属的对象的OID
classoid	oid	pg_class.oid	该对象所在系统目录的OID
description	text		作为该对象描述的任意文本

52.49. pg_shseclabel

目录pg_shseclabel存储共享数据库对象上的安全标签。安全标签可以通过SECURITY LABEL命令操纵。更简单的查看安全标签的方式请见第 52.83 节

另请参阅pg_seclabel，它对单个数据库中对象的安全标签提供了相似的功能。

与大部分其他系统目录不同，pg_shseclabel在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg_shseclabel的拷贝，而不是每个数据库一份。

表 52.49.pg_shseclabel的列

名称	类型	引用	描述
objoid	oid	任意OID列	该安全标签所属对象的OID
classoid	oid	pg_class.oid	对象所属系统目录的OID
provider	text		与此标签关联的标签提供者

名称	类型	引用	描述
label	text		应用到该对象的安全标签

52.50. pg_statistic

目录pg_statistic存储有关数据库内容的统计数据。其中的项由ANALYZE创建，查询规划器会使用这些数据来进行查询规划。注意所有的统计数据天然就是近似的，即使它刚刚被更新。

通常对于数据表中一个已经被 ANALYZE 过的列，在本目录中会存在一个stainherit = false的项。如果该列所在的表具有后代（即有其他表继承该表），对于该列还会创建第二个stainherit = true的项。stainherit = true的项表示列在整个继承树上的统计数据，即通过SELECT column FROM table*看到的数据的统计，而stainherit = false的项表示对SELECT column FROM ONLY table的结果的统计。

pg_statistic也存储关于索引表达式值的统计数据，就好像它们是真正的数据列，但在这种情况中starelid指索引。对一个普通非表达式索引列不会创建项，因为它将是底层表列的项的冗余。当前，索引表达式的项都具有stainherit = false。

因为不同类型的统计信息适用于不同类型的数据， pg_statistic 被设计成不太在意自己存储的是什么类型的统计。 只有极为常用的统计信息（比如NULL的含量）才在pg_statistic里给予专用的字段。 其它所有东西都存储在“槽位”中，而槽位是一组相关的列， 它们的内容用槽位中的一个列里的代码表示。 更详细的信息请参阅 src/include/catalog/pg_statistic.h。

pg_statistic不应该是公共可读的，因为即使是一个表内容的统计性信息也可能被认为是敏感的（例子：一个薪水列的最大和最小值可能是非常有趣的）。pg_stats是pg_statistic上的一个公共可读的视图，它只会显示出当前用户可读的表的信息。

表 52.50.pg_statistic的列

名称	类型	引用	描述
starelid	oid	pg_class.oid	被描述列所属的表或索引
staattnum	int2	pg_attribute.attnum	被描述列的编号
stainherit	bool		如果为真，统计包含了继承后代的列而不仅仅是指定关系的列
stanullfrac	float4		列的项为空的比例
stawidth	int4		非空项的平均存储宽度，以字节计
stadistinct	float4		列中非空唯一值的数目。一个大于零的值是唯一值的真正数目。一个小于零的值是表中行数的乘数的负值；例如，对于一个 80% 的值为非空且每个非空值平均出现两次的列，可以表示为stadistinct = -0.4。一个0值表示唯一值的数目未知。
stakindN	int2		一个代码，它表示存储在

名称	类型	引用	描述
			该pg_statistic行中第N个“槽位”的统计类型。
staopN	oid	pg_operator.oid	一个用于生成这些存储在第N个“槽位”的统计信息的操作符。比如，一个柱面图槽位会用<操作符，该操作符定义了该数据的排序顺序。
stanumbersN	float4[]		第N个“槽位”的类型的数值类型统计， 如果该槽位不涉及数值类型则为NULL
stavaluesN	anyarray		第N个“槽位”的类型的列值，如果该槽位类型不存储任何数据值则为 NULL。 每个数组的元素值实际上都是指定列的数据类型或者是一个相关类型（如数组元素类型）， 因此，除了把这些列的类型定义成anyarray之外别无他法。

52.51. pg_statistic_ext

目录pg_statistic_ext保持有扩展的规划器统计信息。这个目录中的每一行对应于一个用CREATE STATISTICS创建的统计信息对象。

表 52.51.pg_statistic_ext的列

名称	类型	引用	描述
stxrelid	oid	pg_class.oid	包含这个对象所描述的列的表
stxname	name		统计信息对象的名称
stxnamespace	oid	pg_namespace.oid	包含这个统计信息对象的名字空间的OID
stxowner	oid	pg_authid.oid	统计信息对象的拥有者
stxkeys	int2vector	pg_attribute.attnum	一个属性编号的数组，表示哪些表列被这个统计信息对象覆盖。例如值1 3表示第一个和第三个表列被覆盖
stxkind	char[]		包含被启用统计类型代码的数组，可用的值有： d表示n-distinct统计信息，

名称	类型	引用	描述
			f表示函数依赖统计信息
stxndistinct	pg_ndistinct		N-distinct计数，序列化为pg_ndistinct类型
stxdependencies	pg_dependencies		函数依赖统计信息，序列化为pg_dependencies类型

在统计信息对象被创建时会填充stxkind域，表示想要哪些统计类型。其后的域初始为NULL，只有当ANALYZE计算出相应的统计信息后才会填充它们。

52.52. pg_subscription

目录pg_subscription包含所有现有的逻辑复制订阅。更多有关逻辑复制的信息请见第 31 章

和大部分系统目录不同，pg_subscription在集簇的所有数据库之间共享：每个集簇只有一份pg_subscription拷贝，而不是每个数据库一份。

对列subconninfo的访问被从普通用户那里收回，因为该列可能含有明文口令。

表 52.52.pg_subscription的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择）
subdbid	oid	pg_database.oid	订阅所在的数据库的OID
subname	name		订阅的名称
subowner	oid	pg_authid.oid	订阅的拥有者
subenabled	bool		如果为真，订阅被启用并且应该被复制。
subsynccommit	text		包含订阅工作者的synchronous_commit设置的值。
subconninfo	text		到上游数据库的连接字符串
subslotname	name		上游数据库中的复制槽的名称。也被用于本地复制源名称。
subpublications	text[]		被订阅的publication名称的数组。这些引用的是发布者服务器上的publication。更多有关publication的内容请见第 31.1 节

52.53. pg_subscription_rel

目录pg_subscription_rel包含每个订阅中每个被复制关系的状态。这是一种多对多映射。

这个目录仅包含运行CREATE SUBSCRIPTION或ALTER SUBSCRIPTION ... REFRESH PUBLICATION以后对订阅已知的表。

表 52.53.pg_subscription_rel的列

名称	类型	引用	描述
srsubid	oid	pg_subscription.oid	对订阅的引用
srrelid	oid	pg_class.oid	对关系的引用
srsubstate	char		状态代码： i = 初始化， d = 数据正在被拷贝， s = 已同步， r = 准备好（普通复制）
srsublsn	pg_lsn		s和r状态的结束LSN。

52.54. pg_tablespace

目录pg_tablespace存储关于可用表空间的信息。表可以被放置在特定表空间中以实现磁盘布局的管理。

与大部分其他系统目录不同，pg_tablespace在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg_tablespace的拷贝，而不是每个数据库一份。

表 52.54.pg_tablespace的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
spcname	name		表空间名
spcowner	oid	pg_authid.oid	表空间的拥有者，通常是创建它的用户
spcacl	aclitem[]		访问权限，详见GRANT和REVOKE
spcoptions	text[]		表空间级别的选项，形 如“keyword=value”的字符串

52.55. pg_transform

目录pg_transform存储有关转换的信息，转换是 一种让数据类型适应过程语言的机制。详见CREATE TRANSFORM。

表 52.55.pg_transform的列

名称	类型	引用	描述
trftype	oid	pg_type.oid	这个转换所针对的数据类型的 OID
trflang	oid	pg_language.oid	这个转换所针对的语言的 OID

名称	类型	引用	描述
trffromsql	regproc	pg_proc.oid	一个函数的 OID，该函数用来将数据类型转换为过程语言的输入（例如 函数参数）。如果不支持这种操作，这里存储零。
trftosql	regproc	pg_proc.oid	一个函数的 OID，该函数被用来转换过程语言的输出（例如返回值）为 该数据类型。如果不支持这种操作，这里存储零。

52.56. pg_trigger

目录pg_trigger存储表和视图上的触发器。详见CREATE TRIGGER。

表 52.56.pg_trigger的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
tgrelid	oid	pg_class.oid	触发器所在的表
tgname	name		触发器名（在同一个表的触发器中必须唯一）
tgfoid	oid	pg_proc.oid	要被触发器调用的函数
tgtype	int2		触发器触发条件的位掩码
tgenabled	char		控制触发器在session_replication_role模式中的触发。 O = 触发器在“origin”和“local”模式触发， D = 触发器被禁用， R = 触发器在“replica”模式触发， A = 触发器总是触发。
tgisinternal	bool		为真表示触发器是内部生成的（通常是为了强制由tgconstraint指定的约束）
tgconstrrelid	oid	pg_class.oid	被一个引用完整性约束引用的表
tgconstrindid	oid	pg_class.oid	支持一个唯一、主键、引用完整性约束或者排除约束的索引

名称	类型	引用	描述
tgconstraint	oid	pg_constraint.oid	可能存在的与触发器相关的pg_constraint项
tgdeferrable	bool		如果约束触发器可推迟则为真
tginitdeferred	bool		如果约束触发器初始可推迟则为真
tgnargs	int2		传递给触发器函数的参数字符串个数
tgattr	int2vector	pg_attribute.attnum	如果触发器是列限定的，这里存放列号；否则这是一个空数组
tgargs	bytea		传递给触发器的参数字符串，每一个都以NULL结尾
tgqual	pg_node_tree		触发器WHEN条件的表达式树（以nodeToString()的表现形式），如果没有则为空
tgoldtable	name		OLD TABLE的REFERENCING子句名称，如果没有则为空
tgnewtable	name		NEW TABLE的REFERENCING子句名称，如果没有则为空

当前，列限定触发器只被UPDATE事件支持，因此tgattr只用于这种事件类型。tgtype页可以包含用于其他事件类型的位，但其他事件类型是表范围的触发器且会忽略tgattr。

> **注意**
>
> 当tgconstraint非零时，tgconstrrelid、tgconstrindid、tgdeferrable和tginitdeferred与被引用的pg_constraint项有很大的冗余。但是，存在将一个不可延迟触发器关联到一个可延迟约束的可能性：外键约束可以有一些可延迟和一些不可延迟触发器。

> **注意**
>
> 如果一个关系在本目录中拥有任何触发器，其pg_class.relhastriggers必须为真。

52.57. pg_ts_config

pg_ts_config目录包含表示文本搜索配置的选项。一个配置指定了一个特定的文本搜索分析器和一个用于分析器输出记号的字典列表。分析器由pg_ts_config项展现，而记号到字典的映射则由pg_ts_config_map中的辅助项定义。

PostgreSQL的文本搜索特性在第 12 章有更详尽的描述。

表 52.57.pg_ts_config的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
cfgname	name		文本搜索配置名
cfgnamespace	oid	pg_namespace.oid	包含该配置的名字空间的OID
cfgowner	oid	pg_authid.oid	配置的拥有者
cfgparser	oid	pg_ts_parser.oid	该配置的文本搜索分析器的OID

52.58. pg_ts_config_map

pg_ts_config_map目录包含的项展示了对于每一个文本搜索配置的每一种输出记号类型，有哪些文本搜索字典可供查询以及以何种顺序。

PostgreSQL的文本搜索特性在第 12 章有更详尽的描述。

表 52.58.pg_ts_config_map的列

名称	类型	引用	描述
mapcfg	oid	pg_ts_config.oid	拥有该映射项的pg_ts_config项的OID
maptokentype	integer		一种由配置的分析器送出的记号类型
mapseqno	integer		查询该项的顺序（mapseqno值小的优先）
mapdict	oid	pg_ts_dict.oid	查询的文本搜索字典的OID

52.59. pg_ts_dict

pg_ts_dict目录包含定义文本搜索字典的项。一个字典依赖于一个文本搜索模板，它指定了所有需要的函数实现，字典本身则为模板支持的用户可设置参数提供值。这种分工允许无权限的用户创建字典。参数由一个文本串dictinitoption定义，其格式和意义随着模板而变化。

PostgreSQL的文本搜索特性在第 12 章有更详尽的描述。

表 52.59.pg_ts_dict的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
dictname	name		文本搜索字典名

名称	类型	引用	描述
dictnamespace	oid	pg_namespace.oid	包含该字典的名字空间OID
dictowner	oid	pg_authid.oid	字典的拥有者
dicttemplate	oid	pg_ts_template.oid	该字典的文本搜索模板的OID
dictinitoption	text		模板的初始化选项串

52.60. pg_ts_parser

pg_ts_parser目录包含定义文本搜索分析器的项。一个分析器负责将输入文本分割成词位并为每一个词位分配一个记号类型。由于一个分析器必须用C语言级别的函数实现，创建新分析器的工作只限于数据库的超级用户。

PostgreSQL的文本搜索特性在第 12 章有更详尽的描述。

表 52.60.pg_ts_parser的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
prsname	name		文本搜索分析器的名字
prsnamespace	oid	pg_namespace.oid	包含此分析器的名字空间的OID
prsstart	regproc	pg_proc.oid	分析器启动函数的OID
prstoken	regproc	pg_proc.oid	分析器的下一记号函数的OID
prsend	regproc	pg_proc.oid	分析器的关闭函数的OID
prsheadline	regproc	pg_proc.oid	分析器的大标题函数的OID
prslextype	regproc	pg_proc.oid	分析器的词汇类型函数的OID

52.61. pg_ts_template

pg_ts_template目录包含定义文本搜索模板的项。一个模板是一类文本搜索字典的实现骨架。由于一个模板必须用C语言级别的函数实现，新模板的创建只限于数据库超级用户。

PostgreSQL的文本搜索特性在第 12 章有更详尽的描述。

表 52.61.pg_ts_template的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
tmplname	name		文本搜索模板的名字
tmplnamespace	oid	pg_namespace.oid	包含此模板的名字空间的OID

名称	类型	引用	描述
tmplinit	regproc	pg_proc.oid	模板的初始化函数的OID
tmpllexize	regproc	pg_proc.oid	模板的词汇化函数的OID

52.62. pg_type

目录pg_type存储有关数据类型的信息。基类和枚举类型（标度类型）使用CREATE TYPE创建，而域使用CREATE DOMAIN创建。数据库中的每一个表都会有一个自动创建的组合类型，用于表示表的行结构。也可以使用CREATE TYPE AS创建组合类型。

表 52.62.pg_type的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
typname	name		数据类型的名字
typnamespace	oid	pg_namespace.oid	包含此类型的名字空间的OID
typowner	oid	pg_authid.oid	类型的拥有者
typlen	int2		对于一个固定尺寸的类型，typlen是该类型内部表示的字节数。对于一个变长类型，typlen为负值。-1表示一个“varlena”类型（具有长度字），-2表示一个以空值结尾的C字符串。
typbyval	bool		typbyval判断内部例程传递这个类型的数值时是通过传值还是传引用。如果typlen不是1、2或4（或者在Datum为8字节的机器上为8），因此typbyval最好是假。变长类型总是传引用。注意即使长度允许传值，typbyval也可以为假。
typtype	char		typtype可以是： b表示一个基类， c表示一个组合类型（例如一个表的行类型），d表示一个域， e表示一个枚举类型， p表示一个伪类型，或r表示一个范围类型。

名称	类型	引用	描述
			另请参阅typrelid和typbasetype.
typcategory	char		typcategory是一种任意的数据类型分类，它被分析器用来决定哪种隐式转换“更好”。参见表 52.63
typispreferred	bool		如果此类型在它的typcategory中是一个更好的转换目标，此列为真
typisdefined	bool		如果此类型已被定义则为真，如果此类型只是一个表示还未定义类型的占位符则为假。当typisdefined为假，除了类型名字、名字空间和OID之外什么都不能被依赖。
typdelim	char		在分析数组输入时，分隔两个此类型值的字符。注意该分隔符是与数组元素数据类型相关联的， 而不是和数组的数据类型关联。
typrelid	oid	pg_class.oid	如果这是一个复合类型（见typtype），那么这个列指向pg_class中定义对应表的项（对于自由存在的复合类型，pg_class项并不表示一个表，但不管怎样该类型的pg_attribute项需要链接到它）。对非复合类型此列为零。
typelem	oid	pg_type.oid	如果typelem不为0，则它标识pg_type里面的另外一行。 当前类型可以被加上下标得到一个值为类型typelem的数组来描述。 一个“真的”数组类型是变长的（typlen = -1），但是一些定长的（typlen > 0）类型也拥有非零的typelem，比如name和point。 如

名称	类型	引用	描述
			果一个定长类型拥有一个typelem， 则它的内部形式必须是某个typelem数据类型的值，不能有其它数据。变长数组类型有一个由该数组子例程定义的头。
typarray	oid	pg_type.oid	如果typarray不是0，则它标识pg_type中的另一行，这一行是一个将此类型作为元素的“真的”数组类型
typinput	regproc	pg_proc.oid	输入转换函数（文本格式）
typoutput	regproc	pg_proc.oid	输出转换函数（文本格式）
typreceive	regproc	pg_proc.oid	输入转换函数（二进制格式），如果没有则为0
typsend	regproc	pg_proc.oid	输出转换函数（二进制格式），如果没有则为0
typmodin	regproc	pg_proc.oid	类型修改器输入函数，如果类型没有提供修改器则为0
typmodout	regproc	pg_proc.oid	类型修改器输出函数，如果类型没有提供修改器则为0
typanalyze	regproc	pg_proc.oid	自定义ANALYZE函数，0表示使用标准函数
typalign	char		typalign是当存储此类型值时要求的对齐性质。它应用于磁盘存储以及该值在PostgreSQL内部的大多数表现形式。 如果数值是连续存放的，比如在磁盘上的一个完整行，在这种类型的数据前会插入填充，这样它就可以按照指定边界存储。 对齐引用是该序列中第一个数据的开头。对齐引用是序列中第一个数据的开始。 可能的值有： • c = char对齐，即不需要对齐。

名称	类型	引用	描述
			• s = short对齐（在大部分机器上为2字节）。 • i = int对齐（在大部分机器上为4字节）。 • d = double对齐（在很多机器上为8字节，但绝不是全部）。 **注意** 对于系统表中使用的类型，很关键的是，pg_type中定义的尺寸和对齐方式要和编译器在表示表行的结构中布局列的方式保持一致。
typstorage	char		如果一个变长类型（typlen = -1）可被TOAST，typstorage说明这种类型的列应采取的默认策略。可能的值是： • p：值必须平面存储。 • e：值可以被存储在一个“二级”关系（如果有，见pg_class.reltoastrelid）。 • m：值可以被压缩线内存储。 • x：值可以被压缩线内存储或存储在“二级”存储。

名称	类型	引用	描述
			注意m列也可以被移动到二级存储，但只能是作为最后一种方案（e和x列会先被移动）。
typnotnull	bool		typnotnull表示类型上的一个非空约束。只用于域。
typbasetype	oid	pg_type.oid	如果这是一个域（见typtype），则typbasetype标识这个域基于的类。如果此类不是一个域则为0。
typtypmod	int4		域使用typtypmod来记录被应用于它们基类型的typmod（如果基类型不使用typmod，则为-1）。如果此类型不是一个域则为-1。
typndims	int4		对于一个数组上的域，typndims是数组维度数（即，typbasetype是一个数组类型）。除数组类型上的域之外的类型的此列为0。
typcollation	oid	pg_collation.oid	typcollation指定此类型的排序规则。如果类型不支持排序规则，此列为0。一个支持排序规则的基类型此列值为DEFAULT_COLLATION_OID。如果一个可排序类型上的域被指定了一个排序规则，该域可能使用某些其他排序规则OID。
typdefaultbin	pg_node_tree		如果typdefaultbin为非空，那么它是 该类型默认表达式的nodeToString()表现形式。这个列只用于域。
typdefault	text		如果某类型没有相关默认值，那么typdefault为空。如果typdefaultbin不为空，那么typdefault必须包含一个typdefaultbin所指

名称	类型	引用	描述
			的默认表达式的人类可读的版本。 如果typdefaultbin为空但typdefault不为空，则typdefault是该类型默认值的外部表现形式， 它可以被交给该类型的输入转换器来产生一个常量。
typacl	aclitem[]		访问权限，另请参阅GRANT和REVOKE

表 52.63列出了typcategory的系统定义值。任何未来对此列表的增加都将是大写ASCII字母。所有其他ASCII字符都保留给用户定义的类别。

表 52.63. typcategory编码

编码	类别
A	数组类型
B	布尔类型
C	组合类型
D	日期/时间类型
E	枚举类型
G	几何类型
I	网络地址类型
N	数字类型
P	伪类型
R	范围类型
S	字符串类型
T	时间间隔类型
U	用户定义类型
V	位串类型
X	未知类型

52.63. pg_user_mapping

目录pg_user_mapping存储从本地用户到远程的映射。对这个目录的访问对普通用户有限制，可使用视图pg_user_mappings替代。

表 52.64. pg_user_mapping的列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须被显式选择才会显示）
umuser	oid	pg_authid.oid	将要被映射的本地角色的OID，如果用户映射是公共的则为0

名称	类型	引用	描述
umserver	oid	pg_foreign_server.oid	包含此映射的外部服务器的OID
umoptions	text[]		用户映射相关选项，以“keyword=value”字符串形式

52.64. 系统视图

除系统目录外，PostgreSQL提供了一些内建视图。一些系统视图为系统目录上一些常用查询提供了便利的访问。其他视图提供了对内部服务器状态的访问。

信息模式（第 37 章）提供了一组可供选择的视图，它和系统视图在功能上有所重叠。由于信息模式是SQL标准，而这里描述的视图是PostgreSQL特有的，如果信息模式能提供你所需要的信息，通常最好使用它。

表 52.65 列出了这里描述的系统视图。每一个视图的详细文档都在后文中。还有一些附加视图用于提供对于统计收集器结果的访问，它们在表 28.2 中描述。

除了特别注明的，所有这里描述的视图都是只读的。

表 52.65. 系统视图

视图名字	用途
pg_available_extensions	可用的扩展
pg_available_extension_versions	所有版本的扩展
pg_config	编译时配置参数
pg_cursors	打开的游标
pg_file_settings	配置文件内容摘要
pg_group	数据库用户组
pg_hba_file_rules	客户端认证配置文件内容的摘要
pg_indexes	索引
pg_locks	当前保持或者等待的锁
pg_matviews	物化视图
pg_policies	策略
pg_prepared_statements	预备好的语句
pg_prepared_xacts	预备好的事务
pg_publication_tables	publication和它们相关的表
pg_replication_origin_status	有关复制源的信息，包括复制进度
pg_replication_slots	复制槽信息
pg_roles	数据库角色
pg_rules	规则
pg_seclabels	安全标签
pg_sequences	序列
pg_settings	参数设置
pg_shadow	数据库用户
pg_stats	规划器统计信息
pg_tables	表

视图名字	用途
pg_timezone_abbrevs	时区简写
pg_timezone_names	时区名字
pg_user	数据库用户
pg_user_mappings	用户映射
pg_views	视图

52.65. pg_available_extensions

pg_available_extensions视图列出了可用于安装的扩展。参见pg_extension目录，它显示当前已安装的扩展。

表 52.66.pg_available_extensions的列

名字	类型	描述
name	name	扩展名
default_version	text	默认版本的名字，如果没有指定则为NULL
installed_version	text	当前已安装的扩展版本，如果没有安装则为NULL
comment	text	来自于扩展的控制文件的注释字符串

pg_available_extensions视图是只读的。

52.66. pg_available_extension_versions

pg_available_extension_versions视图列出了可用于安装的指定扩展版本。参见pg_extension目录，它显示当前已安装的扩展。

表 52.67.pg_available_extension_versions的列

名字	类型	描述
name	name	扩展名
version	text	版本名
installed	bool	如果此版本的扩展当前已安装则为真
superuser	bool	如果只有超级用户被允许安装此扩展则为真
relocatable	bool	如果扩展能被重定位到另一个模式则为真
schema	name	此扩展必须被安装到的模式名，如果此扩展是部分或者全部可以重定位的，此列为NULL
requires	name[]	先决条件扩展的名字，如果没有则为NULL
comment	text	来自于扩展的控制文件的注释字符串

pg_available_extension_versions视图是只读的。

52.67. pg_config

视图pg_config描述了当前安装的PostgreSQL版本中的编译时配置参数。它存在的本意是用于那些要和PostgreSQL交互的软件包，让它们能找到所需要的头文件和库。它提供了和pg_config PostgreSQL客户端应用相同的基本信息。

默认情况下，pg_config视图只能由超级用户读取。

表 52.68.pg_config列

名称	类型	描述
name	text	参数名
setting	text	参数值

52.68. pg_cursors

pg_cursors视图列出了当前可用的游标。游标可以以几种方式定义：

- 通过SQL中的DECLARE语句
- 通过前端/后端协议中的绑定消息，如第 53.2.3 节所描述的
- 通过服务器编程接口（SPI），如第 47.1 节所描述的

pg_cursors视图显示由任何这些方式创建的游标。视图只存在于定义它们的事务期间，除非声明了WITH HOLD。因此非保持游标只在它们的创建事务结束前存在于这个视图中。

> **注意**
>
> 视图用于在内部实现PostgreSQL的某些部件，例如过程语言。因此，pg_cursors视图可能包括那些不是由用户显式创建的游标。

表 52.69.pg_cursors的列

名字	类型	描述
name	text	游标名
statement	text	提交用于定义此游标的查询语句
is_holdable	boolean	如果游标是可保持的（即，它可以在其定义事务提交后被访问）则为true，否则为否
is_binary	boolean	如果游标被声明为BINARY则为true，否则为false
is_scrollable	boolean	如果游标是可滚动的（即，允许以一种非顺序的方式检索行）则为true，否则为false
creation_time	timestamptz	游标被声明的时间

pg_cursors视图是只读的。

52.69. pg_file_settings

视图pg_file_settings提供了服务器配置文件 内容的概要。这个视图中的每一行表示配置文件中出现的一个 “name = value”项，还带有注解指示该值是否被成功地应用。在 配置文件有问题时，有可能出现额外的行，它们没有相关的 “name = value”项，一个例子是配置文件中有语法错误。

这个视图有助于检查在配置文件中打算做的修改是否能工作，或者用来诊断 之前的失败。注意这个视图报告的是配置文件的当前内容， 而不是服务器最后应用的值（这些值通常查看 pg_settings 视图就够了）。

默认情况下，pg_file_settings视图只有超级用户可读。

表 52.70.pg_file_settings的列

名称	类型	描述
sourcefile	text	配置文件的完整路径名
sourceline	integer	该项在配置文件中出现的行号
seqno	integer	项被处理的顺序（1..n）
name	text	配置参数名
setting	text	被赋予给参数的值
applied	boolean	为真表示值已被成功应用
error	text	如果非空，表示一个错误消息，它说明为什么这个项不能被应用

如果配置文件包含语法错误或者非法参数名，服务器将不会尝试从其中应用 任何设置，并且因此所有的applied域都为假。在这种情 况下，将会有一个或者多个行的error域为非空， 它们说明了为什么出问题。否则，将尽可能应用每个设置。如果一个设置不能 被应用（例如非法值或者该设置不能在服务器开始后改变），会有一个合适的 消息存储在它的error域中。一个项的applied 域为假的另一种情况是它被后面一个具有相同参数名的项所覆盖，这种情况不 会被认为是一种错误，因此在error域中不会有 错误消息。

关于更改运行时参数的各种方法请见第 19.1 节

52.70. pg_group

视图pg_group为向后兼容而存在：它模拟了存在于PostgreSQL 8.1之前版本中的一个目录。它显式所有角色的名称和未被标记rolcanlogin的成员，它是被用做组的角色集合的近似。

表 52.71.pg_group的列

名称	类型	引用	描述
groname	name	pg_authid.rolname	组名
grosysid	oid	pg_authid.oid	组ID
grolist	oid[]	pg_authid.oid	包含此组中角色ID的一个数组

52.71. pg_hba_file_rules

视图pg_hba_file_rules提供客户端认证配置文件pg_hba.conf内容的摘要。该文件中每个非空、非注释行都会在这个视图中出现一行，行中还有标记表示该规则是否被成功地应用。

这个视图可用来检查认证配置文件中按计划的更改是否起作用，或者诊断之前的失败。注意这个视图报告的是该文件的当前内容，而不是服务器最后一次载入的内容。

默认情况下，只有超级用户可以读取pg_hba_file_rules视图。

表 52.72.pg_hba_file_rules的列

名称	列	描述
line_number	integer	这条规则在pg_hba.conf中的行号
type	text	连接类型
database	text[]	这条规则应用的数据库名列表
user_name	text[]	这条规则应用的用户及组名列表
address	text	主机名或IP地址，或者all、samehost、samenet之一，对于本地连接为空
netmask	text	IP地址掩码，如果不适用则为空
auth_method	text	认证方法
options	text[]	为认证方法指定的选项（如果有）
error	text	如果非空，则是一个错误消息，它表示为什么这一行无法被处理

通常，反映一个不正确项的行只有line_number和error域中有值。

更多有关客户端认证配置的信息请参考第 20 章

52.72. pg_indexes

视图pg_indexes提供对于数据库中每一个索引信息的访问。

表 52.73.pg_indexes的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表和索引的模式名
tablename	name	pg_class.relname	此索引的基表的名字
indexname	name	pg_class.relname	索引名
tablespace	name	pg_tablespace.spcname	包含索引的表空间名（如果是数据库的默认值则为空）
indexdef	text		索引定义（CREATE INDEX命令的重构）

52.73. pg_locks

视图pg_locks提供了数据库服务器上活动进程中保持的锁的信息。更多锁的讨论参见第 13 章

pg_locks中对每一个活动可锁对象、请求锁模式和相关进程的组合都有一行。因此，如果多个进程持有或者正在等待一个可锁对象上的锁，同一个可锁对象可能出现很多次。但是，一个当前没有被锁的对象根本不会出现。

有多种不同类型的可锁对象：整个关系（如表）、关系的单个页、关系的单个元组、事务ID（包括虚拟和永久ID）和普通数据库对象（由类OID和对象OID标识，和pg_description或pg_depend中的相同方式）。扩展一个关系的权力也被表示为一个独立的可锁对象。“咨询”锁可以具有用户定义的意义。

表 52.74.pg_locks的列

名称	类型	引用	描述
locktype	text		可锁对象的类型：relation， extend，page， tuple，transactionid，virtualxid，object， userlock或advisory
database	oid	pg_database.oid	锁目标存在的数据库的OID，如果目标是一个共享对象则为0，如果目标是一个事务ID则为空
relation	oid	pg_class.oid	作为锁目标的关系的OID，如果目标不是一个关系或者只是关系的一部分则此列为空
page	integer		作为锁目标的页在关系中的页号，如果目标不是一个关系页或元组则此列为空
tuple	smallint		作为锁目标的元组在页中的元组号，如果目标不是一个元组则此列为空
virtualxid	text		作为锁目标的事务虚拟ID，如果目标不是一个虚拟事务ID则此列为空
transactionid	xid		作为锁目标的事务ID，如果目标不是一个事务ID则此列为空ID
classid	oid	pg_class.oid	包含锁目标的系统目录的OID，如果目标不是一个普通数据库对象则此列为空
objid	oid	任意OID列	锁目标在它的系统目录中的OID，如果目标不是一个普通数据库对象则为空
objsubid	smallint		锁的目标列号（classid和objid指表本身），如果目标

名称	类型	引用	描述
			是某种其他普通数据库对象则此列为0，如果目标不是一个普通数据库对象则此列为空
virtualtransaction	text		保持这个锁或者正在等待这个锁的事务的虚拟ID
pid	integer		保持这个锁或者正在等待这个锁的服务器进程的PID，如果此锁被一个预备事务所持有则此列为空
mode	text		此进程已持有或者希望持有的锁模式（参见第 13.3.1 和第 13.2.3 节
granted	boolean		如果锁已授予则为真，如果锁被等待则为假
fastpath	boolean		如果锁通过快速路径获得则为真，通过主锁表获得则为假

一个行的granted为真表示一个被指定进程持有的锁。为假表示该进程当前正在等待获取这个锁，这意味着至少一个其他进程正持有或等待同一个可锁对象上的一个冲突锁。该等待进程将一直休眠直到其他锁被释放（或者一个死锁状态被检测到）。单个进程在同一时间只能等待最多一个锁。

贯穿一个事务的运行，一个服务器进程在其生存周期内都持有一个在其虚拟事务ID上的排他锁。如果一个永久ID被分配给事务（通常发生在事务改变数据库状态时），它也会持有一个在其永久事务ID上的排他锁直到它结束。当一个事务发现它需要等待另一个事务，它也会尝试获取其他事务ID上的共享锁（不管是虚拟还是永久ID，视情况而定）。这只有当其他进程终止并释放其锁后才会成功。

尽管元组是一种可锁对象，关于行级锁的信息被存储在磁盘而不是内存中，因此行级锁通常不在这个视图中出现。如果一个进程正在等待一个行级锁，它通常在这个视图中出现，并且表示形式为正在等待已持有该行级锁的永久事务ID上的锁。

咨询锁可以在由一个单一bigint值或两个整形值构成的键上获取。一个bigint键被显示为其高位部分在classid列中，低位部分在objid列中，并且objsubid等于1。原来的bigint值可以使用表达式(classid::bigint << 32) | objid::bigint重组。整形键被显示为第一个键在classid列中，第二个键在objid列中，并且objsubid等于2。键的实际意义由用户决定。咨询锁是每一个数据库的本地锁，所以database列对于一个咨询锁没有意义。

pg_locks提供了一个对于整个数据集簇中所有锁的全局视图，而不仅仅是与当前数据库相关的锁。尽管它的relation列可以被连接到pg_class.oid来标识被锁关系，但这种方法只有在关系属于当前数据库（database列是当前数据库OID或者0的锁对应的关系）的情况下才会得到正确的结果。

pid列可以被连接到 pg_stat_activity视图的pid列来得到持有或等待持有每一个锁的会话的信息。 例如

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

另外，如果正在使用预备事务，virtualtransaction列可以被连接到pg_prepared_xacts视图的transaction列来得到持有该锁的预备事务的信息（一个预备事务不可能正在等待一个锁，但它在运行中会一直持有已获得的锁）。 例如：

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

虽然通过自连接pg_locks可以获得哪些进程阻塞了其他哪些进程的信息，但是很难得到其中的细节。这样一个查询隐藏了关于哪些锁模式与其他哪些锁模式冲突的知识。更糟糕的是，pg_locks视图无法给出所等待队列中进程的等待顺序，也无法显示哪些进程是代表其他客户端会话运行的并行工作者。更好的方法是使用pg_blocking_pids()函数（见表 9.60来标识一个等待进程是被哪些进程阻塞的。

pg_locks视图显示来自于普通锁管理器和谓词锁管理器的数据，它们是独立的系统。此外，普通锁管理器把它的锁分为普通锁和快速路径锁。这些数据并不被保证是完全一致的。当视图被查询时，快速路径锁上的数据（fastpath = true）会被一次性从每一个后端收集起来，且并不冻结整个锁管理器的状态。因此有可能某些锁在上述信息被收集的过程中被获得或者释放。注意，不管怎样这些锁是已知不会和任何当前正在发生的锁冲突。在所有后端已经查询了快速路径锁后，普通锁管理器的剩余部分被作为一个单元锁住，并且所有剩余锁的一个一致快照被作为一个原子动作收集。在解锁普通锁管理器后，谓词锁管理器也被类似地锁住并且所有谓词锁被作为一个原子动作收集。因此，在快速路径锁这种特殊情况下，每一个锁管理器会传递一个一致的结果组。但由于我们并不会同时锁上两个锁管理器， 在我们询问完普通锁管理器后或者询问谓词锁管理器之前，锁可以被获得或者释放。

如果对此视图频繁地访问，对普通或者谓词锁管理器加锁可能会对数据库性能产生一定影响。虽然这些锁只会在最少的时间内被保持（足以从锁管理器获得数据），但这无法完全消除可能产生的性能影响。

52.74. pg_matviews

视图pg_matviews提供了关于数据库中每一个物化视图的信息。

表 52.75.pg_matviews的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含物化视图的模式的名字
matviewname	name	pg_class.relname	物化视图的名字
matviewowner	name	pg_authid.rolname	物化视图拥有者的名字
tablespace	name	pg_tablespace.spcname	包含物化视图的表空间名（如使用数据库默认表空间则为空）
hasindexes	boolean		如果物化视图有（或者最近有过）任何索引，则此列为真
ispopulated	boolean		如果物化视图当前已被填充，则此列为真
definition	text		物化视图的定义（一个重构的SELECT查询）

52.75. pg_policies

视图pg_policies提供了有关数据库中行级 安全性策略的信息。

表 52.76.pg_policies的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含策略所在表的模式的名称
tablename	name	pg_class.relname	策略所在表的名称
policyname	name	pg_policy.polname	策略名称
polpermissive	text		策略是宽容性的还是限制性的?
roles	name[]		这个策略适用的角色
cmd	text		这个策略适用的命令类型
qual	text		作为这个策略适用的查询的安全屏障条件增加的表达式
with_check	text		作为尝试向该表增加行的查询的 WITH CHECK 条件增加的表达式

52.76. pg_prepared_statements

pg_prepared_statements视图显示在当前会话中可用的所有预备语句。关于预备语句详见PREPARE。

pg_prepared_statements为每一个预备语句包含一行。当一个新的预备语句被创建时在此视图中会增加一行，反之当一个预备语句被释放时在此视图中会删除一行（例如，通过DEALLOCATE命令）。

表 52.77.pg_prepared_statements的列

名字	类型	描述
name	text	预备语句的标识符
statement	text	客户端提交用于创建此预备语句的查询语句。对于通过SQL创建的预备语句，这里是由客户端提交的PREPARE语句。对于通过前端/后端协议创建的预备语句，这里是预备语句本身的文本。
prepare_time	timestamptz	预备语句被创建的时间
parameter_types	regtype[]	预备语句期望的参数类型，以一个regtype数组的形式。这个数组中一个元素所对应的OID可通过将regtype值转换为oid获得。
from_sql	boolean	如果预备语句通过SQL命令PREPARE创建，则为true；如果预备语句通过前端/后端协议创建，则为false

pg_prepared_statements视图为只读。

52.77. pg_prepared_xacts

视图pg_prepared_xacts显示关于两阶段提交（详见PREPARE TRANSACTION）的当前准备好事务的信息。

pg_prepared_xacts为每一个预备事务包含一行。当事务被提交或回滚时，相应的项将被移除。

表 52.78.pg_prepared_xacts的列

名称	类型	引用	描述
transaction	xid		预备事务的数字事务标识符
gid	text		分配给事务的全局标识符
prepared	timestamp with time zone		此事务为提交准备好的时间
owner	name	pg_authid.rolname	执行此事务的用户名
database	name	pg_database.datname	执行此事务所在数据库的名字

当pg_prepared_xacts视图被访问时，内部事务管理器数据结构被暂时地锁住，并为视图的显示产生一个副本。这确保了视图中是一组一致的结果，并且不会阻塞普通操作。不管怎样，当此视图被频繁访问时，会对数据库性能有所影响。

52.78. pg_publication_tables

视图pg_publication_tables提供publication与其所包含的表之间的映射信息。和底层的目录pg_publication_rel不同，这个视图展开了定义为FOR ALL TABLES的publication，这样对这类publication来说，每一个合格的表都有一行。

表 52.79.pg_publication_tables的列

名称	类型	引用	描述
pubname	name	pg_publication.pubname	publication名称
schemaname	name	pg_namespace.nspname	包含表的方案名称
tablename	name	pg_class.relname	表名

52.79. pg_replication_origin_status

pg_replication_origin_status视图包含有关一个特定源已经重放了多少的信息。更多有关复制源的内容请见第 50 章

表 52.80.pg_replication_origin_status的列

名称	类型	引用	描述
local_id	oid	pg_replication_origin.roident	内部的节点标识符
external_id	text	pg_replication_origin.roname	外部的节点标识符
remote_lsn	pg_lsn		源节点的 LSN，到这个位置的数据都已经被复制。
local_lsn	pg_lsn		这个节点的 LSN，remote_lsn已经

名称	类型	引用	描述
			被复制到这里。使用异步提交时，在将数据持久化到磁盘前用它来刷入提交记录。

52.80. pg_replication_slots

pg_replication_slots视图提供了当前存在于数据库集簇上的所有复制槽的列表，其中也包括复制槽的当前状态。

更多关于复制槽的信息，请见第 26.2.6 和第 49 章

表 52.81.pg_replication_slots的列

名称	类型	引用	描述
slot_name	name		一个唯一的、集簇范围内的复制槽标识符
plugin	name		包含这个逻辑槽正在使用的输出插件的共享对象基础名称，这个列对于物理槽为空值。
slot_type	text		槽类型 - physical（物理）或者logical（逻辑）
datoid	oid	pg_database.oid	与这个槽相关的数据库的OID，或者为空值。只有逻辑槽具有相关的数据库。
database	text	pg_database.datname	与这个槽相关的数据库的名称，或者为空值。只有逻辑槽具有相关的数据库。
temporary	boolean		如果这是一个临时复制槽则为真。临时槽不会被保存在磁盘上并且会在出错或会话结束时自动被删除掉。
active	boolean		如果这个槽当前正在被使用则为真
active_pid	integer		如果槽当前正在被使用，则记录使用这个槽的会话的进程 ID。如果槽没有被使用则为NULL。
xmin	xid		这个槽要需要数据库保留的最旧事务。VACUUM不能移除被其后续事务删除的元组。
catalog_xmin	xid		这个槽要需要数据库保留的影响系统目录

名称	类型	引用	描述
			的最旧事务。VACUUM不能移除被其后续事务删除的目录元组。
restart_lsn	pg_lsn		可能仍被这个槽的消费者要求的最旧WAL地址（LSN），并且因此不会在检查点期间自动被移除。如果这个槽的LSN从未被保留过，则为NULL。
confirmed_flush_lsn	pg_lsn		代表逻辑槽的消费者已经确认接收数据到什么位置的地址（LSN）。比这个地址更旧的数据已经不再可用。对于物理槽这里是NULL。

52.81. pg_roles

视图pg_roles提供了关于数据库角色的信息。这是pg_authid的一个公共可读视图，它隐去了口令域。

此视图显示了低层表的OID列，因为需要它来和其他目录做连接。

表 52.82.pg_roles的列

名称	类型	引用	描述
rolname	name		角色名
rolsuper	bool		角色是否具有超级用户权限？
rolinherit	bool		如果此角色是另一个角色的成员，角色是否能自动继承另一个角色的权限？
rolcreaterole	bool		角色能否创建更多角色？
rolcreatedb	bool		角色能否创建数据库？
rolcanlogin	bool		角色是否能登录？即此角色能否被作为初始会话授权标识符？
rolreplication	bool		角色是一个复制角色。复制角色可以开启复制连接并且创建和删除复制槽。
rolconnlimit	int4		对于一个可登录的角色，这里设置角色可以发起的最大并发连接数。-1表示无限制。

名称	类型	引用	描述
rolpassword	text		不是口令（看起来是********）
rolvaliduntil	timestamptz		口令失效时间（只用于口令认证），如果永不失效则为空
rolbypassrls	bool		绕过每一条行级安全性策略的角色，详见第 5.7 节
rolconfig	text[]		运行时配置变量的角色特定默认值
oid	oid	pg_authid.oid	角色的ID

52.82. pg_rules

视图pg_rules提供对查询重写规则的信息访问。

表 52.83.pg_rules的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	规则适用的表名
rulename	name	pg_rewrite.rulename	规则名
definition	text		规则定义（创建命令的重构）

pg_rules视图排除了视图和物化视图的ON SELECT规则，它们可以在pg_views和pg_matviews中找到。

52.83. pg_seclabels

视图pg_seclabels提供对安全标签的信息访问。它是pg_seclabel目录的一个便于查询的版本。

表 52.84.pg_seclabels的列

名称	类型	引用	描述
objoid	oid	任意OID列	安全标签所属对象的OID
classoid	oid	pg_class.oid	对象出现的系统目录的OID
objsubid	int4		对于一个表列上的安全标签，这里是列号（objoid和classoid指表本身）。对于所有其他对象类型，此列为0。
objtype	text		此标签应用的对象类型，以文本方式。
objnamespace	oid	pg_namespace.oid	如果适用，为此对象的名字空间的OID；否则为空。

名称	类型	引用	描述
objname	text		此标签应用的对象名，以文本形式。
provider	text	pg_seclabel.provider	与此标签相关的标签提供者。
label	text	pg_seclabel.label	应用于此对象的安全标签。

52.84. pg_sequences

视图pg_sequences提供对数据库中每个序列的信息的访问。

表 52.85.pg_sequences的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含序列的方案名
sequencename	name	pg_class.relname	序列的名称
sequenceowner	name	pg_authid.rolname	序列的拥有者的名称
data_type	regtype	pg_type.oid	序列的数据类型
start_value	bigint		序列的起始值
min_value	bigint		序列的最小值
max_value	bigint		序列的最大值
increment_by	bigint		序列的增量值
cycle	boolean		序列是否循环
cache_size	bigint		序列的缓冲尺寸
last_value	bigint		最后一个被写入到磁盘的序列值。如果使用了缓冲，这个值可能比从序列中取出的最后一个值大。如果还没有从该序列读取过，则为空。此外，如果当前用户没有该序列上的USAGE或SELECT特权，则这个值为空。

52.85. pg_settings

视图pg_settings提供了对服务器上运行时参数的访问。它本质上是SHOW和SET命令的可替换接口。它还提供了SHOW不能提供的关于每一个参数的一些现实，例如最大值和最小值。

表 52.86.pg_settings的列

名字	类型	描述
name	text	运行时配置参数名
setting	text	参数的当前值
unit	text	参数的隐式单元
category	text	参数的逻辑组

名字	类型	描述
short_desc	text	参数的简短描述
extra_desc	text	附加的参数的详细描述
context	text	要求设置此参数值的上下文
vartype	text	参数类型（bool、enum、integer、real或string）
source	text	当前参数值的来源
min_val	text	参数的最小允许值（对非数字值为空）
max_val	text	参数的最大允许值（对非数字值为空）
enumvals	text[]	一个枚举参数的允许值（对非数字值为空）
boot_val	text	如果参数没有被别的其他设置，此列为在服务器启动时设定的参数值
reset_val	text	在当前会话中，RESET将会设置的参数值
sourcefile	text	当前值被设置的配置文件（空值表示从非配置文件的其他来源设置，由不是超级用户也不是pg_read_all_settings成员的用户检查时也为空值），在配置文件中使用include指令时有用
sourceline	integer	当前值被设置的配置文件中的行号（空值表示从非配置文件的其他来源设置，由不是超级用户也不是pg_read_all_settings成员的用户检查时也为空值）。
pending_restart	boolean	如果配置文件中修改了该值但需要重启，则为true， 否则为false。

对于context有多种可能的取值。为了降低改变设置的难度，它们是：

internal

这些设置不能被直接修改，它们反映了内部决定的值。某些可能在使用不同配置选项重建系统时或者改变initdb的选项时可以调整。

postmaster

这些设置只能在服务器启动时应用，因此任何修改都需要重启服务器。这些设置的值通常都存储在postgresql.conf文件中，或者在启动服务器时通过命令行传递。当然，具有更低context类型的设置也可以在服务器启动时间被设置。

sighup

对于这些设置的修改可以在postgresql.conf中完成并且不需要重启服务器。发送一个SIGHUP信号给postmaster会导致它重新读取postgresql.conf并应用修改。Postmaster将会把SIGHUP信号传递给它的孩子进程，这样它们也会获得新的值。

superuser-backend

对于这些设置的更改可以在postgresql.conf中进 行而无需重启服务器。也可以在连接请求包（例如通过libpq 的PGOPTIONS环境变量）中为一个特定的会话设定它们，但是只有在连接用户是超级用户时才能这样做。如果，在会话启动后这些设置就 不会改变。如果你在postgresql.conf改变了它们， 向 postmaster 发送一个SIGHUP信号让 postmaster 重新读取postgresql.conf。新的值将 只会影响后续启动的会话。

backend

对于这些设置的修改可以在postgresql.conf中完成并且不需要重启服务器。它们也可以在一个连接请求包（例如，通过libpq的PGOPTIONS环境变量）中为一个特定会话设置 ，任何用户都可以为这个会话做这种修改。然而，这些设置在会话启动后永不变化。如果你在postgresql.conf中修改它们，可以向postmaster发送一个SIGHUP信号让它重读postgresql.conf。新值只会影响后续启动的会话。

superuser

这些设置可以从postgresql.conf设置，或者在会话中用SET命令设置。仅当没有通过SET设置会话本地值时，postgresql.conf中的改变才会影响现有的会话。

user

这些设置可以从postgresql.conf设置，或者在会话中用SET命令设置。任何用户都被允许修改它们的会话本地值。仅当没有通过SET设置会话本地值时，postgresql.conf中的改变才会影响现有的会话。

更多关于修改这些参数的方法的信息请见第 19.1 节

pg_settings视图不能被插入或者从中删除，但是它可以被更新。在pg_settings的一行上的一个UPDATE等价于在该参数上执行一个SET命令。修改将只会影响当前会话使用的值。如果一个UPDATE在一个后来中断的事务中被发出，UPDATE命令的效果也会随着事务的回滚而消失。一旦所在的事务被提交，效果将一直保持到会话结束，除非有其他UPDATE或 SET重新修改它。

52.86. pg_shadow

视图pg_shadow的存在是为了向后兼容：它模拟了在PostgreSQL版本8.1之前的一个系统目录。它显示pg_authid中所有被标记为rolcanlogin的角色的属性。

由于这个表包含口令，所以不能是公众可读的，这也是采用pg_shadow这个名字的原因。而pg_user是pg_shadow上的一个公共可读视图，它屏蔽了口令域。

表 52.87.pg_shadow的列

名称	类型	引用	描述
usename	name	pg_authid.rolname	用户名
usesysid	oid	pg_authid.oid	用户的ID
usecreatedb	bool		用户能否创建数据库
usesuper	bool		用户是否为一个超级用户
userepl	bool		用户能否开启流复制并将系统设置或者取消备份模式。
usebypassrls	bool		用户能否绕过所有的行级安全性策略，详见 第 5.7 节

名称	类型	引用	描述
passwd	text		口令（可能被加密），如果没有则为空。关于加密口令如何存储请参见pg_authid。
valuntil	abstime		口令过期时间（仅用于口令认证）
useconfig	text[]		运行时配置变量的会话默认值

52.87. pg_stats

视图pg_stats提供对存储在pg_statistic目录中信息的访问。此视图能访问pg_statistic行是有限制的，可访问行所对应的表必须是用户有读权限的。因此让所有用户都可以读此视图是安全的。

pg_stats也被设计为能以更适合阅读的格式显示底层目录的信息—　　但代价是只要为pg_statistic定义了新的槽类型，就必须要扩展此视图的模式。

表　52.88.pg_stats的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	表名
attname	name	pg_attribute.attname	被此行描述的列名
inherited	bool		如果为真，表示此行包括继承子列，不仅仅是指定表中的值
null_frac	real		列项中为空的比例
avg_width	integer		列项的平均字节宽度
n_distinct	real		如果大于零，表示列中可区分值的估计个数。如果小于零，是可区分值个数除以行数的负值（当ANALYZE认为可区分值的数量会随着表增长而增加时采用负值的形式，而如果认为列具有固定数量的可选值时采用正值的形式）。例如，-1表示一个唯一列，即其中可区分值的个数等于行数。
most_common_vals	anyarray		列中最常用值的一个列表（如果没有任何一个值看起来比其他值更常用，此列为空）
most_common_freqs	real[]		最常用值的频率列表，即每一个常用值

名称	类型	引用	描述
			的出现次数除以总行数（如果most_common_vals为空，则此列为空）
histogram_bounds	anyarray		将列值划分成大小接近的组的值列表。如果存在most_common_vals，其中的值会被直方图计算所忽略（如果列类型没有一个<操作符或者most_common_vals等于整个值集合，则此列为空）
correlation	real		物理行顺序和列值逻辑顺序之间的统计关联。其范围从-1到+1。当值接近-1或+1时，在列上的一个索引扫描被认为比值接近0时的代价更低，因为这种情况减少了对磁盘的随机访问（如果列数据类型不具有一个<操作符，则此列为空）
most_common_elems	anyarray		在列值中，最经常出现的非空元素列表（对标度类型为空）
most_common_elem_freqs	real[]		最常用元素值的频度列表，即含有至少一个给定值实例的行的分数。在每个元素的频度之后有二至三个附加值，它们是每个元素频度的最小和最大值，以及可选的空元素的频度（如果most_common_elems为空，则此列为空）
elem_count_histogram	real[]		在列值中可区分非空元素值计数的一个直方图，后面跟随可区分非空元素的平均数（对于标度类型为空）

在数组域中项的最大数目可以使用ALTER TABLE SET STATISTICS命令控制，或者设置default_statistics_target运行时参数从全局上进行控制

52.88. pg_tables

视图pg_tables提供对数据库中每个表的信息的访问。

表 52.89.pg_tables的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	表名
tableowner	name	pg_authid.rolname	表拥有者的名字
tablespace	name	pg_tablespace.spcname	包含表的表空间的名字（如果使用数据库的默认表空间，此列为空）
hasindexes	boolean	pg_class.relhasindex	如果表有（或最近有过）任何索引，此列为真
hasrules	boolean	pg_class.relhasrules	如果表有（或曾经有过）规则，此列为真
hastriggers	boolean	pg_class.relhastriggers	如果表有（或者曾经有过）触发器，此列为真
rowsecurity	boolean	pg_class.relrowsecurity	如果表上启用了行安全性则为真

52.89. pg_timezone_abbrevs

视图pg_timezone_abbrevs提供了对当前被时间输入例程识别的时区缩写的列表。当timezone_abbreviations运行时参数被修改，此视图的内容会发生变化。

表 52.90.pg_timezone_abbrevs的列

名字	类型	描述
abbrev	text	时区缩写
utc_offset	interval	相对于UTC的偏移（正值表示格林威治东部）
is_dst	boolean	如果这是一个夏令时缩写，则为真

虽然大部分时区缩写表示从 UTC 开始的固定偏移，但是有一些在历史上有值的变化（详见第 B.4 节。在这种情况下，这个视图表示它们现在的含义。

52.90. pg_timezone_names

视图pg_timezone_names提供了一个被SET TIMEZONE识别的时区名字的列表，以及它们的相关缩写、UTC偏移和夏令时状态（从技术上来说，PostgreSQL不使用UTC是因为闰秒没有被处理）。和pg_timezone_abbrevs中展示的缩写不同，这里很多名字隐含了一组夏令时转换日期规则。因此，相关信息在本地DST边界间变化。所显示的信息基于CURRENT_TIMESTAMP的当前值计算得来。

表 52.91.pg_timezone_names的列

名字	类型	描述
name	text	时区名
abbrev	text	时区缩写

名字	类型	描述
utc_offset	interval	相对于UTC的偏移（正值表示格林威治东部）
is_dst	boolean	如果当前保持为夏令时则为真

52.91. pg_user

视图pg_user提供关于数据库用户的信息。这是pg_shadow的一个公共可读的视图，它消除了口令域。

表 52.92.pg_user的列

名字	类型	描述
usename	name	用户名
usesysid	oid	用户的ID
usecreatedb	bool	用户是否能创建数据库
usesuper	bool	用户是否为超级用户
userepl	bool	用户能否开启流复制以及将系统转入/转出备份模式。
usebypassrls	bool	用户能否绕过所有的行级安全性策略，详见 第 5.7 节
passwd	text	不是口令（总是显示为********）
valuntil	abstime	口令过期时间（只用于口令认证）
useconfig	text[]	运行时配置变量的会话默认值

52.92. pg_user_mappings

视图pg_user_mappings提供有关用户映射的信息。这是pg_user_mapping的一个公共可读视图，它对无权使用的用户省去了选项域。

表 52.93.pg_user_mappings的列

名称	类型	引用	描述
umid	oid	pg_user_mapping.oid	用户映射的OID
srvid	oid	pg_foreign_server.oid	包含该映射的外部服务器的OID
srvname	name	pg_foreign_server.srvname	外部服务器名
umuser	oid	pg_authid.oid	将被映射的本地角色的OID，如果用户映射是公共的则为0
usename	name		将被映射的本地用户名
umoptions	text[]		用户映射指定选项，以“keyword=value”字符串的形式

为了保护存储为用户映射选项的口令信息，umoptions列将被读作空，除非满足下列情况之一：

- 当前用户就是被映射的用户，并且拥有该服务器或者持有其上的USAGE特权
- 当前用户是服务器的拥有者并且映射是用于PUBLIC
- 当前用户是一个超级用户

52.93. pg_views

视图pg_views提供了数据库中每个视图的信息。

表 52.94.pg_views的列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含视图的模式名
viewname	name	pg_class.relname	视图名称
viewowner	name	pg_authid.rolname	视图拥有者的名字
definition	text		视图定义（一个重构的SELECT查询）

第 53 章 前端/后端协议

PostgreSQL使用一种基于消息的协议用于前端和后端（服务器和客户机）之间通讯。该协议是在TCP/IP和Unix 域套接字上实现的。端口号 5432 已经在IANA 注册为支持这种协议的服务器的常用端口，但实际上任何非特权端口号都可以使用。

这份文档描述了版本3.0的协议，它在PostgreSQL版本 7.4 和以后的版本中实现。对于以前版本协议的描述，请参考以前版本的PostgreSQL文档。一台服务器能够支持多种协议版本。初始的启动请求消息告诉服务器客户端尝试使用哪个协议版本。如果客户端请求的主版本不被服务器支持，连接将被拒绝（例如，如果客户端请求的协议版本是4.0就会发生这种情况，因为在写作这份文档时，4.0根本还不存在）。如果客户端请求的次版本不被服务器支持（例如客户端请求版本3.1，但服务器仅支持3.0），服务器可能会拒绝该连接或者用一个包含它支持的最高次协议版本的NegotiateProtocolVersion消息进行响应。然后客户端可以选择使用指定的协议版本继续连接或者中止连接。

为了可以有效地为多个客户端提供服务，服务器为每个客户端派生一个新的“后端”进程。在目前的实现里，在检测到新来的连接请求后，马上创建一个新的子进程。 不过，这些是对协议透明的。对于协议而言，术语“后端”和“服务器”是可以互换的； 类似的还有“前端”和“客户端”也是可以互换的。

53.1. 概述

协议在启动和正常操作过程中有不同的阶段。在启动阶段里，前端打开一个到服务器的连接并且认证自身以满足服务器（这可能涉及到一条或多条消息，取决于使用的认证方法）。如果一切正常，服务器就发送状态信息给前端，并最后进入正常操作。除了最初的启动请求消息之外，协议的这个部分是服务器驱动的。

在正常操作中，前端发送查询和其它命令到后端，然后后端返回查询结果和其它响应。在少数几种情况（比如NOTIFY）中，后端会发送未被请求的消息，但这个会话中的绝大多部分都是由前端请求驱动的。

会话的终止通常是由前端来选择的，但是也可以在某些情况下由后端强制执行。不管在那种情况下，如果后端关闭连接，那么它将在退出之前回滚所有打开的（未完成的）事务。

在正常操作中，SQL命令可以通过两个子协议中的任何一个执行。 在“简单查询”协议中，前端只是发送一个文本查询串， 然后后端马上分析并执行它。在“扩展查询”协议中， 查询的处理被分割为多个步骤：分析、参数值绑定和执行。这样就可以提供灵活性和性能的改进，但代价是额外的复杂性。

正常操作还有用于类似COPY这样的额外的子协议。

53.1.1. 消息概貌

所有通讯都是通过一个消息流进行的。消息的第一个字节标识消息类型， 然后后面跟着的四个字节声明消息剩下部分的长度（这个长度包括长度域自身，但是不包括消息类型字节）。 剩下的消息内容由消息类型决定。由于历史原因，客户端发送的最初的消息（启动消息）不包含消息类型字节。

为了避免失去与消息流的同步，服务器和客户端通常都是把整个消息读取到一个缓冲区里（使用字节计数）， 然后才试图处理其内容。这样在处理内容的过程时如果发现错误，就比较容易恢复。 在非常极端的情况下（比如说没有足够的内存缓冲消息），接收端可以使用字节计数来判断它在继续读取消息之前需要跳过多少输入。

反之，服务器和客户端都需要注意决不能发送一条不完整的消息。保证这一点的方法通常是在发送整条信息之前先在一个缓冲区里整理整条消息。 如果在发送或者接受一条消息的中间发生了通讯错误，那么唯一合理的响应是放弃连接，因为恢复消息边界同步的希望很小。

53.1.2. 扩展查询概述

在扩展查询协议中，SQL命令的执行是分割成多个步骤的。步骤与步骤之间保存的状态是由两类的对象代表的：预备语句（prepared statements）和入口（portals）。 一个预备语句代表一个文本查询字符串的经过分析、语意解析以及规划之后的结果。一个预备语句不代表它已经可以被执行，因为它可能还缺乏 参数的值。 一个入口代表一个已经可以执行的或者已经被部分执行过的语句，所有缺失的参数值都已经填充到位了（对于SELECT语句，入口等效于一个打开的游标， 但我们使用不同的术语是因为游标不能处理非SELECT语句）。

完整的执行周期包括一个分析步骤， 它从一个文本的查询字符串里创建一个预备语句； 一个绑定步骤， 它用一个预备语句和任何所需的参数值创建一个入口；以及一个执行步骤，它运行一个入口中的查询。如果查询会返回数据行（SELECT、SHOW等）， 执行步骤会被告知只抓取有限的一些行，这样就可能需要多个执行步骤来完成操作。

后端可以跟踪多个预备语句和入口（但是要注意，这些只存在于一个会话内部，不能在会话之间共享）。现有的预备语句和入口都是用创建它们的时候赋予的名字引用的。 另外，还存在一个“未命名” 的预备语句和入口。 尽管它们的行为和命名对象大部分相同，但是它们是针对只执行一次然后就抛弃的查询而优化的， 而在命名对象上的操作是针对多次使用而优化的。

53.1.3. 格式和格式代码

特定数据类型的数据可以用几种不同的格式中的任意一种来传递。 从PostgreSQL 7.4开始，只支持“文本”和“二进制”两种格式， 但是协议为未来的扩展提供了的手段。任意值要求的格式用一个格式代码声明。 客户端可以为每个传输的参数值和查询结果的每个列指定一个格式代码。 文本的格式代码是零，二进制的格式代码是一，所有其它的格式代码都保留给将来定义。

文本形式的数值是特定数据类型的输入/输出转换函数生成或接受的任何字符串。在传输形式上，字符串没有末尾空字符；如果前端要想把收到的值当作C字符串处理，那么必须自己加上一个（顺便说一下，文本格式不允许嵌入空字符）。

整数的二进制形式采用网络字节序（高位在前）。对于其它数据类型，请参考文档或者源代码获取其二进制形式的信息。请注意，复杂数据类型的二进制形式可能在不同服务器版本之间变化； 文本格式通常是最具有移植性的选择。

53.2. 消息流

本节描述消息流以及每种消息类型的语意（每种信息的准确形式在第 53.7 节）。根据连接的状态不同，存在几种不同的子协议： 启动、查询、函数调用、COPY和终止。还有特殊的规定用于一步操作（包括通知响应和命令取消），这些可能在启动阶段过后的任何时间产生。

53.2.1. 启动

要开始一个会话，前端打开一个与服务器的连接并且发送一个启动消息。这个消息包括用户名以及用户希望连接的数据库名； 它还标识要使用的特定的协议版本（启动信息可以有选择地包括运行时参数的额外设置）。服务器然后就使用这些信息及服务器配置文件的内容（比如 pg_hba.conf）来判断这个连接是否可以接受以及需要什么样的额外的认证（如果需要）。

然后服务器就发送合适的认证请求信息，前端必须用合适的认证响应信息来响应（比如一个口令）。对于除了GSSAPI、SSPI和SASL之外的所有认证方式都至少有一个请求和一个响应。在有些方法中前端不需要发出任何响应，并且因此就不会由任何认证请求发生。对于GSSAPI、SSPI和SASL，可能需要多个包的交换才能完成认证。

认证周期要么以服务器的拒绝连接（ErrorResponse）结束， 要么以AuthenticationOK 结束。

这个阶段来自服务器可能消息是：

ErrorResponse

连接请求被拒绝。然后服务器马上关闭连接。

AuthenticationOk

认证交换成功完成。

AuthenticationKerberosV5

现在前端必须与服务器进行一次Kerberos V5认证对话（在这里没有描述，Kerberos规范的一部分）。 如果对话成功，服务器响应一个AuthenticationOk，否则它响应一个ErrorResponse。这已经不再被支持。

AuthenticationCleartextPassword

现在前端必须以明文形式发送一个包含口令的PasswordMessage。如果这是正确的口令，服务器用一个 AuthenticationOk，否则它响应一个ErrorResponse。

AuthenticationMD5Password

现在前端必须发送一个PasswordMessage，其中包含口令，且口令先用用户名做MD5加密，然后使用在AuthenticationMD5Password消息里指定的4字节盐粒加密。 如果这是正确口令，服务器用一个AuthenticationOk 响应，否则它用一个ErrorResponse 响应。实际的PasswordMessage可以用SQL来计算：concat('md5', md5(concat(md5(concat(password, username)), random-salt)))（记住md5()函数返回的结果是一个十六进制串）。

AuthenticationSCMCredential

这个响应只用于在支持SCM信任消息的平台上的本地Unix域连接。前端必须发出一条SCM信任消息然后发送一个数据字节（数据字节的内容并没有意义； 它只被用于确保服务器等待足够长的时间来接受信任信息）。如果信任是可以接受的， 那么服务器用AuthenticationOk响应，否则用ErrorResponse响应（该消息只在9.1之前的服务器中发出。它可能最终会从协议规范中被删除）。

AuthenticationGSS

前端必须现在开始一次GSSAPI谈判。前端将发送一个带有GSSAPI数据流第一部分的GSSResponse消息来响应。如果需要进一步的消息，服务器将会响应AuthenticationGSSContinue。

AuthenticationSSPI

前端必须现在开始一次SSPI谈判。前端将发送一个带有SSPI数据流第一部分的GSSResponse来响应。如果需要进一步的消息，服务器将会响应AuthenticationGSSContinue。

AuthenticationGSSContinue

这个消息包含对于前一步的GSSAPI或SSPI谈判（AuthenticationGSS、AuthenticationSSPI或者前一个AuthenticationGSSContinue）的响应数据。如果这个消息中的GSSAPI或SSPI数据指示需要更多数据来完成认证，前端必须将所需的数据作为另一个GSSResponse发送。如果这个消息就能完成GSSAPI或SSPI认证，服务器将接着发送AuthenticationOk来指示成功认证，或者发送ErrorResponse来指示失败。

AuthenticationSASL

前端现在必须发起一次SASL协商，使用该消息中列出的一种SASL机制。前端将用选中机制的名字发送一个SASLInitialResponse，并且SASL数据流的第一部分对应于此。如果需要进一步的消息，服务器将用AuthenticationSASLContinue回复。详见第 53.3 节

AuthenticationSASLContinue

这个消息包含从SASL协商（AuthenticationSASL或者一个之前的AuthenticationSASLContinue）的前一步中得到的挑战数据。前端必须用一个SASLResponse消息响应。

AuthenticationSASLFinal

已经用额外的与机制相关的数据为该客户端完成SASL认证。服务器接下来将发送AuthenticationOk来表示认证成功，或者发送ErrorResponse表示失败。只有在SASL机制指定要在完成时从服务器发送额外数据到客户端的情况下才会发送这个消息。

NegotiateProtocolVersion

服务器不支持客户端请求的次协议版本，但是支持该协议更早的一个版本，这个消息会指出受支持的最高次版本。如果客户端在启动包中请求了不受支持的协议选项（例如以_pq_.开始），则这个消息也会被发出。这个消息后面将会跟着一个ErrorResponse或者一个指示认证成功或失败的消息。

如果前端不支持服务器要求的认证方式，那么它应该马上关闭连接。

在收到AuthenticationOk包之后，前端必须等待来自服务器的进一步消息。在这个阶段会启动一个后端进程，而前端只是一个感兴趣的旁观者。启动尝试仍有可能失败（ErrorResponse），服务器也有可能拒绝支持所请求的次协议版本（NegotiateProtocolVersion），但是通常情况下，后端将发送一些ParameterStatus消息、BackendKeyData以及最后的ReadyForQuery。

在这个阶段，后端将尝试应用任何在启动消息里给出的额外的运行时参数设置。如果成功，这些值将成为会话的缺省值。错误将导致ErrorResponse并退出。

这个阶段来自后端的可能消息是：

BackendKeyData

这个消息提供了密钥数据，前端如果想要在稍后发出取消请求，则必须保存这个数据。前端不应该响应这个信息，但是应该继续侦听等待ReadyForQuery消息。

ParameterStatus

这个消息告诉前端有关后端参数的当前（初始）设置，比如client_encoding或者DateStyle等。前端可以忽略这些信息，或者记录其设置用于将来使用；参阅第 53.2.6 获取更多细节。前端不应该响应这些信息，而是应该继续侦听ReadyForQuery消息。

ReadyForQuery

启动成功，前端现在可以发出命令。

ErrorResponse

启动失败，在发送完这个消息之后连接被关闭。

NoticeResponse

发出了一个警告信息。前端应该显示这个信息，但是要继续监听ReadyForQuery或ErrorResponse。

后端在每个命令周期后都会发出一个相同的ReadyForQuery消息。出于前端的编码需要，前端可以合理地认为ReadyForQuery是一个命令周期的开始，或者认为ReadyForQuery是启动阶段和每个随后命令周期的结束，具体是那种情况取决于前端的编码需要。

53.2.2. 简单查询

一个简单查询周期是由前端发送一条Query消息给后端进行初始化的。这条消息包含一个用文本字符串表达的 SQL 命令（或者一些命令）。 后端根据查询命令串的内容发送一条或者更多条响应消息给前端，并且最后是一条ReadyForQuery响应消息。ReadyForQuery通知前端它可以安全地发送新命令了 （实际上前端不必在发送其它命令之前等待ReadyForQuery，但是这样一来，前端必须能发现较早发出的命令失败而稍后发出的命令成功的情况）。

来自后端的可能的消息是：

CommandComplete

一个SQL命令正常结束。

CopyInResponse

后端已经准备好从前端拷贝数据到一个表里面去，参见第 53.2.5 节

CopyOutResponse

后端已经准备好从一个表里拷贝数据到前端，参见第 53.2.5 节

RowDescription

表示为了响应一个SELECT、FETCH等查询， 将要返回行。这条消息的内容描述了行的列布局。这条消息后面将跟着DataRow消息，每个DataRow消息包含一个要被返回的行。

DataRow

SELECT、FETCH等查询返回的结果集中的一行。

EmptyQueryResponse

识别了一个空的查询字符串。

ErrorResponse

出错了。

ReadyForQuery

查询字符串的处理完成。发送一个独立的消息来标识这种情况是因为查询字符串可能包含多个SQL命令（CommandComplete只是标记一条SQL命令处理完毕，而不是整个查询字符串处理完毕）。不管是处理成功结束还是产生错误，ReadyForQuery总会被发送。

NoticeResponse

发送了一个与查询有关的警告信息。提示信息是附加在其他响应上的，也就是说，后端将继续处理该命令。

对SELECT查询（或其它返回行集的查询，比如EXPLAIN或SHOW）的响应通常包含RowDescription、零个或者多个 DataRow 消息以及最后的 CommandComplete。 COPY到前端或者从前端COPY会调用第 53.2.5 节描述的特殊协议。所有其他查询类型通常只产生一个CommandComplete消息。

因为查询字符串可能包含若干个查询（用分号分隔），所以在后端完成查询字符串的处理之前可能有好几个这样的响应序列。如果整个字符串已经处理完，后端已经准备好接受新查询字符串的时候则发出 ReadyForQuery消息。

如果收到一个完全空（除了空白之外没有内容）的查询字符串，那么响应是一条EmptyQueryResponse后面跟着ReadyForQuery。

在出现错误的时候，发出一个ErrorResponse消息，后面跟着ReadyForQuery。查询字符串的所有进一步的处理都被ErrorResponse中止（即使里面还有查询）。请注意这些事情可能在处理一个查询产生的消息序列的中途发生。

在简单查询模式中，检索出来的值的格式总是文本，除非给出的命令是在一个使用BINARY选项声明的游标上FETCH。 在这种情况下，检索出来的值是二进制格式的。在 RowDescription 消息里给出的格式代码将告诉我们用了那种格式。

前端在等待其他类型的消息时必须准备接收ErrorResponse和NoticeResponse消息。 参阅第 53.2.6 节来了解后端因为外部事件可能生成的消息。

我们建议的方法是把前端代码写成状态机的风格，它可以在任何时刻接受任何有意义的消息类型，而不是假设消息的序列总是准确。

53.2.2.1. 一个简单查询中的多条语句

当一个简单查询消息中包含多于一条SQL语句（被分号分隔）时，那些语句会被当做一个事务中执行，除非其中包括显式事务控制命令来强制不同的行为。例如，如果消息包括

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

则SELECT中的除零失败将强制回滚第一个INSERT。进而，因为该消息的执行在第一个错误时就被放弃，第二个INSERT根本都不会被尝试。

如果该消息包含的是

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

那么第一个INSERT会被这个显式的COMMIT命令提交。第二个INSERT以及SELECT仍会被当作一个单一事务，这样除零失败将回滚第二个INSERT，但不会回滚第一个。

这种行为通过在一个隐式事务块中的一个多语句Query消息中运行那些语句来实现，除非它们运行在某个显式事务块中。隐式事务块与常规事务块之间的区别在于隐式块会在Query消息结束时自动被关闭，或者是在没有错误的情况下由一个隐式提交关闭，或者是在有错误时由一个隐式的回滚关闭。这类似于一个语句自己执行（当不在事务块中时）时发生的隐式提交或回滚。

如果会话已经在一个事务块中，作为前面某个消息中BEGIN的结果，那么Query消息会简单地继续那个事务块，不管该消息包含一个语句还是多个语句。不过，如果该Query消息包含一个关闭现有事务块的COMMIT或者ROLLBACK，那么任何接下来的语句都会在一个隐式事务块中被执行。反过来，如果在多语句Query消息中出现一个BEGIN，那么它会开始一个常规事务块，这个常规事务块将只能被一个显式的COMMIT或者ROLLBACK终止，不管这两种命令是出现在这个Query消息还是后面的一个Query消息中。如果BEGIN跟在一些作为隐式事务块执行的语句后面，那些语句不会被立刻提交。实际上，它们会被包括到新的常规事务块中。

出现在一个隐式事务块中的COMMIT或者ROLLBACK会被正常执行并且关闭该隐式块。不过，由于没有先前的BEGIN配对的COMMIT或者ROLLBACK表示一种错误，所以将会发出一个警告。如果后面还有更多语句，将会为它们开始一个新的隐式事务块。

在隐式事务块中不允许保存点，因为它们会与发生错误时自动关闭块的行为发生冲突。

记住，不管任何事务控制命令存不存在，Query消息的执行会在第一个错误时停止。因此，对于下面的在一个Query消息中的例子

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

会话中将留下一个失败的常规事务块，因为在出现除零错误后不会到达ROLLBACK。将需要另一个ROLLBACK把会话恢复到一种可用的状态。

另一种要注意的行为是，最初的词法和语法分析是在整个查询字符串被执行之前进行的。因此后面的语句中的简单错误（例如拼写错误的关键词）可能会阻止任何语句的执行。这通常对用户是不可见的，因为在当作一个隐式事务块执行时，这些语句不管怎样都会全部被回滚。不过，在尝试于一个多语句Query中执行多个事务时，这种现象可能是可见的。例如，如果一个拼写错误把我们之前的例子变成

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

那么这些语句都不会被运行，导致可见的差别，即第一个INSERT没有被提交。在语义分析及其后阶段检测到的错误（例如拼错的表名或者列名）不会有这种效果。

53.2.3. 扩展查询

扩展查询协议把上面描述的简单协议分裂成若干个步骤。准备步骤的结果可以被多次复用以提高效率。另外，还可以获得额外的特性， 比如可以把数据值作为独立的参数提供而不是必须把它们直接插入一个查询字符串。

在扩展协议里，前端首先发送一个Parse消息，它包含一个文本查询字符串， 另外还有一些可选的有关参数占位符的数据类型的信息，以及一个最终的预备语句对象的名字（一个空字符串选择未命名的预备语句）。响应是一个ParseComplete或者ErrorResponse。 参数的数据类型可以用OID来指定；如果没有给出，那么分析器将试图用应付无类型文字符串常量的方法来推导其数据类型。

> **注意**
>
> 一个参数的数据类型可以通过设置为零， 或者让参数类型OID的数目比查询字符串里的参数符号（$n）的数目少的方式不予指定。另外一个特例是参数的类型可以声明为void（也就是说，伪类型void的OID）。这是为了允许用于某些函数参数的参数符号实际上是OUT参数。通常情况下，没有什么环境会用到void参数， 但是如果在函数的参数列表里出现了这么一个参数符号，那么它实际上会被忽略。比如，一个像这样的函数调用：foo($1,$2,$3,$4)，如果$3和$4被指定为具有类型是void，那么这个函数调用会匹配一个带有两个IN和两个OUT参数的函数。

> **注意**
>
> 在一个Parse消息里包含的查询字符串不能包含超过一个SQL语句；否则就会报告一个语法错误。这个限制在简单查询协议中并不存在， 是它存在于扩展协议中，因为允许预备语句或者入口包含多个命令将导致协议过度地复杂。

如果成功创建了一个命名的预备语句对象，那么它将持续到当前会话结束， 除非被明确地删除。一个未命名的预备语句只持续到下一个声明未命名语句的Parse语句发出为止（请注

意一个简单的查询消息也会销毁未命名语句）。命名预备语句必须被明确地关闭，然后才能用一个Parse消息重新定义，但是未命名语句并不要求这个动作。命名预备语句也可以在SQL命令级别创建和访问，方法是使用PREPARE和EXECUTE。

一旦一个预备语句存在，就可以很使用Bind消息使之进入执行状态。Bind消息给出源预备语句的名字（空字符串表示未命名预备语句）、目标入口的名字（空字符串表示未命名的入口）及用于那些在预备语句中出现的所有参数占位符的值。提供的参数集必须匹配那些预备语句所需要的参数（如果你在Parse消息里声明任何void参数，那么在Bind消息里给它们传递NULL值）。Bind还指定被查询返回的数据的格式；格式可以在总体上声明，也可以对每个列进行声明。响应是BindComplete或ErrorResponse。

> **注意**
>
> 输出的格式是文本还是二进制是由Bind里给出的格式代码决定的，而不管涉及的是什么SQL命令。在使用扩展查询协议的时候，游标声明里的BINARY属性是无关的。

当Bind消息被处理时通常会进行查询规划。如果预备语句没有参数或者是被反复执行，服务器可能会保存创建好的计划并在后续对同一个预备语句的Bind消息中重用之。不过，当它发现可以创建一个效率比依赖指定参数值的计划不低很多的一般性计划时，它仍然会进行查询规划。但是这对于协议所关注的来说是透明的。

如果成功创建了一个命名入口对象，那么它将持续到当前事务的结尾，除非被明确地删除。一个未命名入口在事务的结尾删除，或者是在发出的下一个Bind语句声明了一个未命名入口为止（请注意一个简单查询消息也会删除这个未命名入口）。命名入口在可以用一个Bind消息重新定义之前必须明确地关闭，但是未命名入口不要求这个动作。命名入口也可以在SQL命令的级别创建和访问，方法是使用DECLARE CURSOR和FETCH。

一旦一个入口存在，那么就可以用一个Execute消息执行它。Execute消息指定入口的名字（空字符串表示未命名入口）和一个最大的结果行计数（零表示“取出所有行”）。 结果行计数只对包含返回行集的入口有意义；在其它情况下，该命令总是被执行到结束，而行计数会被忽略。Execute消息的可能响应和那些通过简单查询协议发出的查询一样，只不过执行不会导致后端发出ReadyForQuery或者RowDescription。

如果Execute在入口的执行完成之前终止（因为达到了一个非零的结果行计数），它将发送一个PortalSuspended消息；这个消息的出现告诉前端应该在同一个入口上发出另外一个Execute消息以完成操作。在入口的执行完成之前，不会发出表示源SQL命令结束的CommandComplete消息。因此执行阶段总是由下列消息之一出现标志着结束：CommandComplete、EmptyQueryResponse（如果入口是从一个空字符串创建出来的）、ErrorResponse或者PortalSuspended。

每个扩展查询消息序列完成后，前端都应该发出一条Sync消息。这个无参数的消息导致后端关闭当前事务——如果当前事务不是在一个BEGIN/COMMIT事务块中（“关闭”的意思就是在没有错误的情况下提交， 或者是有错误的情况下回滚）。然后响应一条ReadyForQuery消息。Sync的目的是提供一个错误恢复的重新同步的点。 如果在处理任何扩展查询消息的时候侦测到任何错误，那么后端会发出ErrorResponse，然后读取并抛弃消息直到一个Sync到来，然后发出ReadyForQuery并且返回到正常的消息处理中（但是要注意如果正在处理Sync的时候发生了错误，那么不会忽略任何东西 — 这样就保证了为每个Sync发出一个并且只是一个ReadyForQuery）。

> **注意**
>
> Sync并不导致一个用BEGIN打开的事务块关闭。我们可以侦测到这种情况，因为ReadyForQuery消息包含事务状态信息。

除了这些基本的、必须的操作之外，在扩展查询协议里还有几种可选的操作可以使用。

Describe消息（入口变体）指定一个现有的入口的名字（或者一个空字符串表示未命名入口）。响应是一个RowDescription消息，它描述了执行该入口将要返回的行；或者是一个NoData消息——果入口并不包含会返回行的查询；或者是一个ErrorResponse——如果入口不存在。

Describe消息（语句变体）指定一个现有的预备语句的名字（或者一个空字符串表示未命名预备语句）。 响应是一个描述该语句需要的参数的ParameterDescription消息，后面跟着一个描述该语句最终执行后返回的行的RowDescription消息（或者是 NoData 消息，如果该语句不返回行）。如果没有这样的预备语句，则返回ErrorResponse。请注意因为还没有发出Bind，所以后端还不知道用于返回列的格式；在这种情况下，RowDescription消息里面的格式代码域将是零。

> **提示**
>
> 在大多数情况下，前端在发出Execute之前应该发出某种Describe的变体，以保证它知道如何解析它将得到的结果。

Close消息关闭一个现有的预备语句或者入口，并且释放资源。对一个不存在的语句或者入口发出Close不是一个错误。响应通常是CloseComplete，但如果在释放资源的时候遇到了一些困难也可以是ErrorResponse。请注意关闭一个预备语句会隐含地关闭任何从该语句构造出来的打开的入口。

Flush消息并产生任何特定的输出，但是强制后端发送任何还在它的输出缓冲区中待处理的数据。Flush必须在除Sync外的任何扩展查询命令后面发出——如果前端希望在发出更多的命令之前检查该命令的结果的话。如果没有Flush，后端返回的消息将组合成尽可能少的数据包，以减少网络负荷。

> **注意**
>
> 简单查询消息大概等于一系列使用未命名预备语句和无参数入口对象的Parse、Bind、入口Describe、Execute、Close、Sync。一个区别是它会在查询字符串中接受多个SQL语句，并连续地为每个语句自动执行绑定/描述/执行序列。另外一个区别是它不会返回ParseComplete、Bindcomplete、CloseComplete或者NoData消息。

53.2.4. 函数调用

函数调用子协议允许客户端请求一个对存在于数据库pg_proc系统表中的任意函数的直接调用。客户端必须在该函数上有执行的权限。

> **注意**
>
> 函数调用子协议是一个遗留的特性，在新代码里可能最好避免用它。类似的结果可以通过设置一个执行SELECT function($1, ...)的预备语句得到。这样函数调用周期就可以用 Bind/Execute 代替。

一个函数调用周期是由前端向后端发送一条FunctionCall消息初始化的。然后后端根据函数调用的结果发送一条或者更多响应消息，并且最后是一条ReadyForQuery响应消息。ReadyForQuery通知前端它可以安全地发送一个新的查询或者函数调用了。

来自后端的可能的响应消息是：

ErrorResponse

发生了一个错误。

FunctionCallResponse

函数调用完成并且在消息中返回一个结果（请注意函数调用协议只能处理单个标量结果，不能处理行类型或者集合类型的结果）。

ReadyForQuery

函数调用处理完成。ReadyForQuery将总是被发送，不管是成功完成处理还是发生一个错误。

NoticeResponse

发出了一条有关该函数调用的警告信息。通知是附加在其他响应上的，也就是说，后端将继续处理该命令。

53.2.5. COPY操作

COPY命令允许在服务器和客户端之间进行高速大批量数据传输。拷贝入和拷贝出操作每个都把连接切换到一个独立的子协议中，并且持续到操作结束。

拷贝入模式（数据传输到服务器）是在后端执行一个COPY FROM STDIN语句的时候初始化的。后端发送一个CopyInResponse消息给前端。前端应该发送零条或者更多CopyData消息，形成一个输出数据的流（消息的边界和行的边界没有任何相关性要求，尽管通常那是合理的选择）。前端可以通过发送一个CopyDone消息来终止拷贝入操作（允许成功终止），也可以发出一个CopyFail消息（它将导致COPY语句带着错误失败）。 然后后端就恢复回它在COPY开始之前的命令处理模式，可能是简单查询协议，也可能是扩展查询协议。然后它会发送CommandComplete（如果成功）或者ErrorResponse（如果失败）。

如果在拷贝入模式下后端检测到了错误（包括接受接收到CopyFiail消息， 或者是任何除了CopyData或者CopyDone之外的前端消息），那么后端将发出一个ErrorResponse消息。如果COPY命令是通过一个扩展查询消息发出的， 那么后端从现在开始将抛弃前端消息，直到一个Sync消息到达，然后它将发出ReadyForQuery并且返回到正常的处理中。如果COPY命令是在一个简单查询消息里发出的，那么该消息剩余部分被丢弃然后发出ReadyForQuery消息。不管是哪种情况，任何前端发出的CopyData、CopyDone或者CopyFail消息都将被简单地抛弃。

在拷贝入模式下，后端将忽略所收到的Flush和Sync消息。收到任何其他非拷贝消息类型都会造成一个错误，它将导致上面所描述的拷贝入状态中断（Flush和Sync的例外是为了方便客户端库，它们总是在一个Execute消息之后发送Flush和Sync，而不检查被执行的命令是否为一个COPY FROM STDIN）。

拷贝出模式（数据从服务器发出）是在后端执行一个COPY TO STDOUT语句的时候初始化的。后端发出一个CopyOutResponse消息给前端，后面跟着零或者多个CopyData消息（总是每行一个），然后跟着CopyDone。然后后端回退到它在COPY开始之前的命令处理模式，然后发送CommandComplete。前端不能退出传输（除非是关闭连接或者发出一个Cancel请求），但是它可以抛弃不需要的CopyData和CopyDone消息。

在拷贝出模式中，如果后端检测到错误，那么它将发出一个ErrorResponse消息并且回到正常的处理。前端应该把收到ErrorResponse当作终止拷贝出模式的标志。

在CopyData消息中间可能会散布有NoticeResponse和ParameterStatus消息。前端必须处理这些情况，并且应该也为异步消息类型（参见第 53.2.6 节准备好。否则任何除CopyData或CopyDone之外的消息类型都会被认为是要中止拷贝出模式。

还有另外一种被称为双向拷贝的与拷贝相关的模式，它允许“向”和“从”服务器高速传输大批量数据。当后端处于walsender模式中执行一个START_REPLICATION语句时，它会启动双向拷贝模式。后端会发送一个CopyBothResponse消息给前端。然后前端和后端都会发送CopyData消息，然后直到最后发送一个CopyDone消息。在客户端发送一个CopyDone消息后，连接将从双向拷贝模式转换到拷贝出模式，并且客户端将不能发送更多CopyData消息。类似的，当服务器发送了一个CopyDone消息，连接进入到拷贝入模式，并且服务器将不能发送更多CopyData消息。在双方发送完一个CopyDone消息后，拷贝模式被中断，而后端将回到之前的命令处理模式。如果在双向拷贝模式中出现一个后端检测到的错误，后端将发出一个ErrorResponse消息，然后将发出ReadyForQuery并返回到普通处理。前端将把收到ErrorResponse作为在双向上中断拷贝的信号，在这种情况下不会有CopyDone被发出。关于在双向拷贝模式下传输的子协议请参见第 53.4 节

CopyInResponse、CopyOutResponse和CopyBothResponse消息包括域和格式代码，域告诉前端每行的列数，而格式代码则用于具体每个列（就目前的实现而言，一个给定COPY操作中的所有列都将使用同样的格式，但是消息设计并不做这个假设）。

53.2.6. 异步操作

有几种情况下后端会发送一些并非由特定前端命令流传达的消息。在任何时候前端都必须准备处理这些消息，即使它是并未参与一个查询。在最低限度下，我们应该在开始读取查询响应之前检查这些情况。

NoticeResponse消息有可能是因为外部的活动而产生的；比如，如果数据库管理员进行一次“快速”数据库关闭，那么后端将在关闭连接之前发送一个NoticeResponse来表明这些。相应地，前端应该总是准备接受和显示NoticeResponse消息，即使连接事实上是空闲的。

如果任何时候有任何参数值的活跃值改变且后端认为前端应该知道这些，那么都会产生ParameterStatus消息。这种情况最常见发生的情形是对前端执行的一个SET命令进行响应，并且这种情况实际上是同步的 — 但是也有可能是数据库管理员改变了配置文件然后项服务器发出SIGHUP信号导致了参数状态的变化。同样，如果一个SET命令回滚，那么也会生成一个合适的ParameterStatus 消息以报告当前有效值。

目前，系统内有一套会生成ParameterStatus消息的写成硬代码的参数，它们是：（server_encoding，TimeZone 和 integer_datetimes 在 8.0 版本之前没有报告。standard_conforming_strings 在版本 8.1 之前没有报告。） 请注意server_version， server_encoding 和 integer_datetimes 是伪参数，启动后不能修改。 这些可能在将来改变，或者甚至是变成可以配置的。 因此，前端应该简单地忽略那些它不懂或者不关心的 ParameterStatus。 server_version、 server_encoding、client_encoding、 application_name、 is_superuser、 session_authorization、DateStyle、 IntervalStyle、 TimeZone、 integer_datetimes以及standard_conforming_strings （server_encoding、TimeZone以及integer_datetimes在版本8.0之前不会被报告； standard_conforming_strings在版本8.1之前不会被报告；IntervalStyle在版本8.4之前不会被报告； application_name在版本9.0之前不会被报告）。 注意server_version、server_encoding以及integer_datetimes是伪参数，它们不能在启动之后被改变。这种设置可能在未来改变，甚至变成可配置的。相应地，一个前端应该简单地忽略那些与它不懂或者不关心的参数相关的ParameterStatus。

如果前端发出一个LISTEN命令， 那么无论何时在为同一个通道名NOTIFY时，后端将发送一个NotificationResponse消息（不要和NoticeResponse搞混！）。

> **注意**
>
> 目前，NotificationResponse只能在一个事务外面发送，因此它将不会在一个命令响应序列中间出现，但是它可能正好在ReadyForQuery之前出现。不过，在前端逻辑中做上述假设是不明智的。好的做法是在协议的任何点上都可以接受NotificationResponse。

53.2.7. 取消正在处理的请求

在一条查询正在处理的时候，前端可以请求取消该查询。这种取消请求不是直接通过打开的连接发送给后端的，这么做是因为实现的效率：我们不希望后端在处理查询的过程中不停地检查前端来的输入。 取消请求应该相对而言比较少见，所以我们把取消做得稍微笨拙一些，以便不影响正常状况的性能。

要发出一条取消请求，前端打开一个与服务器的新连接并且发送一条CancelRequest消息，而不是通常在新连接中经常发送的StartupMessage消息。服务器将处理这个请求然后关闭连接。 出于安全原因，对取消请求消息不做直接的响应。

除非CancelRequest消息包含在连接启动过程中传递给前端的相同的关键数据（PID和密钥），否则它将被忽略。如果该请求匹配当前运行着的后端的PID和密钥， 则退出当前查询的处理（目前的实现里采用的方法是向正在处理该查询的后端进程发送一个特殊的信号）。

取消信号可能产生或者不产生效果 — 例如，如果它在后端完成查询的处理后到达，那么它就没有做用。如果取消起作用了，会导致当前命令伴随着一个错误消息提前退出。

这么做是对安全性和有效性通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。它必须继续等待后端对查询响应。发出一个取消仅仅是增加了当前查询快些结束的可能性， 同时也增加了当前查询会伴随着一条错误消息失败而不是成功执行的可能性。

因为取消请求是通过新的联接发送给服务器而不是通过平常的前端/后端通讯链接，所以取消请求可能被任意进程发出的，而不仅仅是要取消查询的前端。 这样可能对创建多进程应用提供了更多的灵活性。同时这样也带来了安全风险，因为任何一个非授权用户都可能试图取消查询。这个安全风险通过要求在取消请求中提供一个动态生成的密钥来解决。

53.2.8. 终止

通常优雅的终止过程是前端发送一条Terminate消息并且立刻关闭连接。一旦收到消息，后端马上关闭连接并且终止。

在少数情况下（比如一个管理员命令数据库关闭），后端可能在没有任何前端请求的情况下断开连接。在这种情况下，后端将在它断开连接之前尝试发送一个错误或者通知消息给出断开的原因。

其它终止的情况发生在各种失败的场合，比如某一方的内核转储、失去通讯链路、丢失了消息边界同步等。不管是前端还是后端看到了一个意外的连接关闭，那么它应该清理现场并且终止。 如果前端不想终止自己，那么它有一个选项是重连服务器的方法启动一个新的后端。如果收到了一个无法识别的消息类型，那么我们也建议关闭连接，因为出现这种情况可能意味着是丢失了消息边界的同步。

不管是正常还是不正常的终止，任何打开的事务都会回滚而不是提交。不过，我们应该注意的是如果一个前端在一个非SELECT查询正在处理的时候断开， 那么后端很可能在发现断开之前先完成查询的处理。如果查询处于任何事务块之外（BEGIN ... COMMIT序列），那么其结果很可能在得知断开之前被提交。

53.2.9. SSL会话加密

如果编译PostgreSQL的时候打开了SSL支持，那么前后端通讯就可以用SSL加密。 这样就提供了一种在攻击者可能捕获会话通讯数据包的环境下保证通讯安全的方法。有关使用SSL加密PostgreSQL会话的更多信息， 请参阅第 18.9 节

要开始一次SSL加密连接，前端先是发送一个SSLRequest消息，而不是StartupMessage。然后服务器以一个包含S或N的字节响应，分别表示它愿意还是不愿意进行SSL。如果此时前端对响应不满意， 那么它可以关闭连接。要在S之后继续，那么先进行与服务器的SSL启动握手（没有在这里描述，是SSL规范的一部分）。 如果这些成功了，那么继续发送普通的StartupMessage。这种情况下，StartupMessage和所有随后的数据都将由SSL加密。要在N之后继续，则发送普通的StartupMessage并不适用加密继续处理。

前端应该也准备处理一个来自服务器的给SSLRequest的ErrorMessage响应。这种情况只在服务器早于PostgreSQL的SSL支持的情况下才会出现（这种服务器现在非常古老，并且可能不再存在了）。在这种情况下，连接必需关闭，但是前端可以选择打开一个新的连接然后不使用SSL进行连接。

一个初始化的 SSLRequest 也可以用于打开来用于发送一条 CancelRequest 消息的联接中。

如果协议本身并未提供某种方法强制SSL加密，那么管理员可以把服务器配置为拒绝未加密的会话，这是认证检查的一个副产品。

53.3. SASL认证

SASL是一种针对面向连接协议的认证框架。当前，PostgreSQL实现了两种SASL认证机制，即SCRAM-SHA-256和SCRAM-SHA-256-PLUS。未来可能会增加更多。下面的步骤展示了通常SASL认证是如何执行的，而接下来的小节会给出更多有关SCRAM-SHA-256和SCRAM-SHA-256-PLUS的细节。

SASL认证消息流

1. 为了开始一次SASL认证交换，服务器发送一个AuthenticationSASL消息。它包括一个服务器能够接受的SASL认证机制的列表，这个列表按照服务器喜欢的顺序组织。

2. 客户端从该列表中选取一种受支持的机制，并且发送一个SASLInitialResponse消息给服务器。该消息包括选中的机制名称以及一个可选的初始客户端响应（如果选中的机制要用到）。

3. 接下来将是一个或者多个服务器挑战和客户端响应消息。每一次的服务器挑战都通过一个AuthenticationSASLContinue消息发送，后面会跟着客户端通过SASLResponse消息发送的响应。这些消息的细节与机制有关。

4. 最后，当认证交换成功完成，服务器发送一个AuthenticationSASLFinal消息，后面马上跟上一个AuthenticationOk消息。AuthenticationSASLFinal包含服务器发给客户端的额外数据，其内容与选中的认证机制有关。如果认证机制不使用在完成时发送的额外数据，则不发送AuthenticationSASLFinal消息。

在出错时，服务器可以在任何阶段中止认证，并且发送一个ErrorMessage。

53.3.1. SCRAM-SHA-256认证

目前实现的SASL机制是SCRAM-SHA-256及其带有通道绑定的变体SCRAM-SHA-256-PLUS。它们在RFC 7677和RFC 5802中有详细描述。

当PostgreSQL使用SCRAM-SHA-256时，服务器将忽略客户端在client-first-message中发送的用户名，而是使用在启动消息中已经发过来的用户名。PostgreSQL支持多字符编码，然而SCRAM要求用户名使用UTF-8，因此可能无法表示PostgreSQL中的UTF-8用户名。

SCRAM规范要求口令也是UTF-8，并且要用SASLprep算法对口令进行处理。但是，PostgreSQL不要求对口令使用UTF-8。在设置一个用户的口令时，会把口令当作是UTF-8一样用SASLprep处理，而不管实际使用的编码是什么。不过，如果口令不是一种合法的UTF-8字节序列或者含有SASLprep算法禁止的UTF-8字节序列，将直接使用未被SASLprep处理过的原始口令，而不是抛出错误。这使得不是UTF-8的口令能够被规范化，但是也允许使用非UTF-8的口令，并且不要求系统了解口令到底是什么编码。

在编译了SSL支持的PostgreSQL中支持通道绑定。带有通道绑定的SCRAM的SASL机制名是SCRAM-SHA-256-PLUS。被PostgreSQL使用的通道绑定类型是tls-server-end-point。

在没有通道绑定的SCRAM中，服务器选择一个随机数，它被传送给客户端，客户端会把这个随机数与用户提供的口令混合在被传输的口令哈希中。虽然这能够防止在后面的会话中成功地重新传送这个口令哈希，但是却无法防止位于真实服务器和客户端之间的伪服务器经手服务器的随机值并且认证成功。

带有通道绑定的SCRAM通过将服务器证书的签名混入被传输的口令哈希来防止这类中间人攻击。虽然伪服务器能够重新传输真实服务器的证书，但是它无法访问匹配该证书的私钥，因此无法证明它是其拥有者，进而导致SSL连接失败。

示例

1. 服务器发送一个AuthenticationSASL消息。它包括一个服务器能够接受的SASL认证机制列表。如果服务器编译有SSL支持，这个列表将会是SCRAM-SHA-256-PLUS和SCRAM-SHA-256，否则列表中只会有后者。

2. 客户端通过发送一个SASLInitialResponse消息进行响应，该消息指出选中的机制，即SCRAM-SHA-256或者SCRAM-SHA-256-PLUS（客户端可以自由地选择机制，但是为了更安全在可以支持通道绑定变体时应该选择它）。在Initial Client响应字段中，该消息包含SCRAM的client-first-message。client-first-message还包含客户端选中的通道绑定类型。

3. 服务器发送一个AuthenticationSASLContinue消息，其内容是一个SCRAM的server-first message。

4. 客户端发送一个SASLResponse消息，其内容是SCRAM的client-final-message。

5. 服务器发送一个AuthenticationSASLFinal消息，其中是SCRAM的server-final-message，后面紧接着一个AuthenticationOk消息。

53.4. 流复制协议

要启动流复制，前端在启动消息中发送replication参数。布尔值true（或者on、yes、1）告诉后端进入到物理复制的walsender模式，在其中可以发出一个小型的复制命令集合（见下文）而不是普通的SQL命令。

将database作为replication参数的值传递会指示后端进入到逻辑复制的walsender模式，连接到dbname参数指定的数据库。在逻辑复制的walsender模式中，可以发出下文所示的复制命令以及普通的SQL命令。

在物理复制或者逻辑复制的walsender模式中，只能使用简单查询协议。

出于测试复制命令的目的，你可以通过psql或者任何使用libpq的工具使用包含replication选项的连接字符串建立一个复制连接，例如：

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

不过更常用的是：pg_receivewal（对于物理复制）或者pg_recvlogical（对于逻辑复制）。

当log_replication_commands被启用时，复制命令会被记录在服务器日志中。

在复制模式中可以接受的命令有：

IDENTIFY_SYSTEM

请求服务器标识它自己。服务器以一个行构成的结果集作为答复，其中包含四个域：

systemid (text)

标识该集簇的唯一的系统标识符。这可以被用来检查用于初始化后备机的基础备份是否来自于同一个集簇。

timeline (int4)

当前的时间线 ID。也对于检查后备机是否与主控机一致有用。

xlogpos (text)

当前的WAL刷写位置。用于得到一个在预写式日志中的已知位置作为流的开始位置。

dbname (text)

要连接到的数据库或者空。

SHOW name

请求服务器发送一个运行时参数的当前设置。这类似于SQL命令SHOW。

name

运行时参数的名称。可用的参数在第 19 章。

TIMELINE_HISTORY tli

请求服务器将时间线tli的历史文件发送过来。服务器将以一行组成的结果集作为答复，其中包含两个域：

filename (text)

时间线历史文件的文件名，例如00000002.history。

content (bytea)

时间线历史文件的内容。

CREATE_REPLICATION_SLOT slot_name [TEMPORARY] { PHYSICAL [RESERVE_WAL] | LOGICAL output_plugin [EXPORT_SNAPSHOT | NOEXPORT_SNAPSHOT | USE_SNAPSHOT] }

创建一个物理的或者逻辑的复制槽。更多关于复制槽的内容请见 第 26.2.6 节

slot_name

要创建的槽的名称。必须是一个合法的复制槽名称（见 第 26.2.6.1 节。

output_plugin

被用于逻辑解码的输出插件的名称（见 第 49.6 节。

TEMPORARY

指定这个复制槽是一个临时复制槽。临时复制槽不会被保存在磁盘上并且在发生错误或者会话结束时会被自动删除。

RESERVE_WAL

指定这个物理复制槽立即保留WAL。否则，只有来自流复制客户端的连接上才会保留WAL。

EXPORT_SNAPSHOT
NOEXPORT_SNAPSHOT
USE_SNAPSHOT

决定如何处理在逻辑槽初始化期间取到的快照。默认值EXPORT_SNAPSHOT将把该快照导出以备在其他会话中使用。这个选项不能在事务外面使用。USE_SNAPSHOT将把该快照用于正在执行命令的当前事务。这个选项必须被用在一个事务中，并且CREATE_REPLICATION_SLOT必须是该事务中运行的第一个命令。最后，NOEXPORT_SNAPSHOT将按照正常把该快照用于逻辑解码，但是不会对它做其他事情。

作为对这个命令的响应，服务器将发送一个一行的结果集，其中包含下列字段：

slot_name (text)

新创建的复制槽的名称。

consistent_point (text)

该槽达到一致的WAL位置。这是在该复制槽上可以开始流的最早位置。

snapshot_name (text)

被这个命令导出的快照的标识符。该快照一直是有效的，直到在这个连接中执行了新的命令或者该复制连接被关闭。如果被创建的槽是物理槽，则为空。

output_plugin (text)

新创建的复制槽使用的输出插件的名称。如果被创建的槽是物理槽，则为空。

START_REPLICATION [SLOT slot_name] [PHYSICAL] XXX/XXX [TIMELINE tli]

指示服务器开始启动流WAL，从 WAL 位置XXX/XXX开始。如果TIMELINE选项被指定，流传送会在时间线tli上开始，否则会选择服务器的当前时间线。服务器可以回复一个错误，例如如果被请求的WAL节已经被回收了。如果成功，服务器将会响应一个CopyBothResponse消息，并且然后开始以流的方式把WAL传送给前端。

如果通过slot_name提供了一个槽的名称，它将被更新复制进度，这样该服务器知道哪些WAL 段以及哪些事务（如果hot_standby_feedback为打开）仍然被后备机所需要。

如果客户端请求一个并非最新的时间线，但是属于服务器历史的一部分，服务器将会把该时间线上从请求点开始的所有WAL以流式传送，一直到服务器切换到另外一个时间线的点。如果客户端请求在一个老的时间线末尾进行流传送，服务器将在不进入COPY模式的情况下立即响应CommandComplete。

在流传送完一个非最新时间线上所有的WAL之后，服务器将会通过退出COPY模式来结束流。当客户端认识到这一点并也退出COPY模式时，服务器会发送一个包含一行两列的结果集，以指示在该服务器历史中的下一个时间线。第一列是下一个时间线的 ID（类型int8），而第二列是发生切换的 WAL 位置（类型text）。通常，切换位置是被流传送的WAL的末尾，但是在很少的情况下服务器会从旧的时间线中发送一些WAL，而该时间线是服务器本身在提示之前还没有重放的。最后，服务器发送CommandComplete消息，并且做好准备接受一个新的命令。

WAL数据以一系列CopyData消息的形式被发送（这允许其他信息穿插其中，特别是服务器可以在开始流传送后遇到失败时发送一个ErrorResponse消息）。每个从服务器到客户端的CopyData消息承载了一个下列格式之一的消息：

XLogData (B)

Byte1('w')

标识该消息是WAL数据。

Int64

在消息中WAL数据的起始点。

Int64

服务器上WAL的当前终点。

Int64

在传送时服务器的系统时钟，以从2000-01-01午夜开始的微妙计。

Byten

WAL数据流的一节。

一个WAL记录绝不会被分割到两个XLogData消息。如果一个WAL记录跨越了一个WAL页面的边界，并且因此已经被使用连续的记录分割，它可以在页面边界被分割。换句话说，第一个主要WAL记录和它的后续记录可以在不同的XLogData消息中被发送。

主要存活消息 (B)

Byte1('k')

标识该消息是一个发送者存活消息。

Int64

服务器上WAL的当前终点。

Int64

在传送时服务器的系统时钟，以从2000-01-01午夜开始的微妙计。

Byte1

1表示客户端应该尽快回复该消息，以避免连接超时。否则为0。

接收进程可以在任何时候给发送者发送回复，回复可以使用下列消息格式之一（也在CopData消息中使用）：

后备机状态更新 (F)

Byte1('r')

标识该消息是一个接收者状态更新。

Int64

接收到并且写入到后备机磁盘的最后一个WAL比特的位置+1。

Int64

被刷入到后备机磁盘的最后一个WAL比特的位置+1。

Int64

被应用在后备机上的最后一个WAL比特的位置+1。

Int64

在传送时客户端的系统时钟，以从2000-01-01午夜开始的微妙计。

Byte1

如果为1，客户端要求服务器马上回复这个消息。这可以被用来ping服务器以测试连接是否仍然完好。

热备机反馈消息 (F)

Byte1('h')

标识该消息是一个热备机反馈消息。

Int64

在传送时客户端的系统时钟，以从2000-01-01午夜开始的微妙计。

Int32

后备机的当前全局xmin，排除来自任何复制槽的catalog_xmin。如果这个值和接下来的catalog_xmin都为0，这会被当作一个通知表示在这个连接上将不会再发送热备反馈。后来的非0消息将重新启动反馈机制。

Int32

后备机上全局xmin事务ID的世代。

Int32

后备机上任意复制槽中的最小catalog_xmin。如果在后备机上不存在catalog_xmin或者热备反馈被禁用，这个值被设置为0。

Int32

后备机上全局catalog_xmin事务ID的世代。

START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [(option_name [option_value] [, ...])]

指示服务器为逻辑复制开始流式传送 WAL，从 WAL 位置XXX/XXX开始。服务器可以回复一个错误，例如如果请求的 WAL 小节已经回环。如果成功，服务器会响应一个 CopyBothResponse 消息，并且接着开始流失传送 WAL 给前端。

消息内部的消息与START_REPLICATION ... PHYSICAL中记录的格式相同。

与选中槽关联的输出插件被用来处理流的输出。

SLOT slot_name

要从哪个槽流式传送改变。这个参数是必须的，并且必须对应于一个现有的用LOGICAL模式的CREATE_REPLICATION_SLOT创建的逻辑复制槽。

XXX/XXX

要开始流传送的 WAL 位置。

option_name

一个传递给该槽的逻辑解码插件的选项的名称。

option_value

字符串常量形式的选项值，与前面指定的选项关联。

DROP_REPLICATION_SLOT slot_name [WAIT]

删除一个复制槽，释放任何保留的服务器端资源。如果该槽是一个在不同于walsender所连接的数据库中创建的逻辑槽，这个命令会失败。

slot_name

要删除的槽的名称。

WAIT

如果槽正处于活跃状态，这个选项会导致命令等待，直到槽变得不活跃，而不是像默认行为那样直接报错。

BASE_BACKUP [LABEL 'label'] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE rate] [TABLESPACE_MAP]

指示服务器开始流传送一个基础备份。在备份开始之前系统将自动被置于备份模式，而在备份结束时会自动被退出备份模式。可以接受下列选项：

LABEL 'label'

设置备份的标签。如果没有指定，将会使用base backup作为标签。标签的引号规则和standard_conforming_strings开启时标准SQL字符串的一样。

PROGRESS

请求用以生成一个进度报告的信息。这将送回位于每个表空间头部的一个近似大小，它可以被用于计算流还有多久才能被完成。它通过在传输开始之前枚举所有文件大小来计算，并且可能会对性能产生一种负面影响 -- 特别情况下它可能会在流传送第一个数据之前就耗费很长时间。因为数据库文件可能在备份期间改变，这个大小只是近似的并且可能在近似计算和发送真正的文件之间增长或者收缩。

FAST

请求一个快速检查点。

WAL

在备份中包含必需的WAL段。这将把开始和停止备份之间的所有文件包括在base目录tar文件中的pg_wal目录中。

NOWAIT

默认情况下，备份会等待直到最后一个要求的 WAL 段被归档，或者当日至归档被禁用时发出一个警告。指定NOWAIT会禁用等待和警告，而让客户端负责确保所要求的日志是可用的。

MAX_RATE rate

单位时间内从服务器传输到客户端的最大数据量限制。期望的单位是千字节每秒。如果指定了这个选项，值必须等于零或者位于 32 kB到 1 GB（包括）范围之间。如果 0 被传入或者没有指定该选项，对于传输将没有限制。

TABLESPACE_MAP

在名为tablespace_map的文件中包括有关pg_tblspc目录中存在的符号链接的信息。这个表空间映射文件包括了在目录pg_tblspc中存在的每一个符号链接的名字以及它的完整路径。

NOVERIFY_CHECKSUMS

默认情况下，如果启用了校验码，在基础备份期间会验证校验码。指定NOVERIFY_CHECKSUMS可以禁用这种验证。

当备份被启动，服务器将首先发送两个普通结果集，后面会跟着一个或多个CopyResponse结果。

第一个普通结果集在一行两列中包含了备份的起始位置。第一列包含使用XLogRecPtr格式给出的开始位置，第二列包含相应的时间线ID。

第二个普通结果集中为每一个表空间都有一行。行中的域有：

spcoid (oid)

表空间的 OID，如果是base目录则为空。

spclocation（text）

表空间目录的完整路径，如果是base目录则为空。

size（int8）

如果进度报告被请求，这里是表空间的近似大小，否则为空。

在第二个普通结果集之后，一个或多个CopyResponse结果将被发送，一个用于主数据目录而对每一个除pg_default和pg_global之外的额外表空间也会有一个。CopyResponse结果中的数据将会使一个tar格式（遵循POSIX 1003.1-2008标准中指定的“ustar交换格式”）的表空间内容转储，不过标准中定义的两个拖尾全0块将被忽略。在tar数据完成后，一个最终普通结果集将被发送，包含了备份的WAL结束位置，格式与起始位置相同。

用于数据目录和每个表空间的tar归档将包含目录中的所有文件，不管它们是否为PostgreSQL文件或者是被加入的其他文件。唯一被排除的文件是：
- postmaster.pid
- postmaster.opts
- pg_internal.init（在多个目录中都有）
- 在PostgreSQL服务器操作过程中创建的各种临时文件和目录，例如任何以pgsql_tmp开头的文件或目录以及临时关系。
- 不做日志的关系，init分支除外，恢复时重建（空的）不做日志关系时需要它。
- pg_wal及其子目录。如果备份运行时要求包括WAL文件，一个pg_wal的合成版本将被包括进来，但是只会包含那些备份工作必需的文件，而不是包含剩下的内容。
- pg_dynshmem、pg_notify、pg_replslot、pg_serial、pg_snapshots、pg_stat_tmp以及 pg_subtrans会被拷贝为空目录（即使它们是符号链接）。
- 跳过除常规文件和目录之外的其他文件，例如符号链接（不同于上面所列出的目录）和特殊设备文件。（pg_tblspc中的符号链接会被保留）。

如果服务器上的底层文件系统支持，所有者、组合文件模式都会被设置。

53.5. 逻辑流复制协议

这一节介绍逻辑复制协议，它是一种以复制命令START_REPLICATION SLOT slot_name LOGICAL开始的消息流。

逻辑流复制协议建立在物理流复制协议的原始积累之上。

53.5.1. 逻辑流复制参数

逻辑复制命令START_REPLICATION接受下列参数：

proto_version

协议版本。当前支持版本1。

publication_names

要订阅（接收更改）的publication名称列表，用逗号分隔。每一个publication名称个体都被当作一个标准的对象名称，并且可以根据需要加上引号。

53.5.2. 逻辑复制协议消息

协议消息的个体在接下来的小节中讨论。个体的消息在第 53.9 节介绍。

所有的顶层协议消息都以一个消息类型字节开头。虽然被表示为字符代码，但这是一个没有相关编码的有符号字节。

由于流复制协议提供了一个消息长度，因此不需要顶层协议消息在其头部嵌入长度。

53.5.3. 逻辑复制协议的消息流

除START_REPLICATION命令和重放进度消息之外，所有信息流的方向都是从后端到前端。

逻辑复制协议会逐个发送事务个体。这意味着在一对Begin消息和Commit消息之间的所有消息都属于同一个事务。

每一个被发送的事务都包含零个或者多个DML消息（插入、更新、删除）。在级联设置的情况下，它还包括Origin消息。Origin消息表示该事务是在不同的复制节点上产生的。由于逻辑复制协议范围内的复制节点可以是任何东西，所以唯一的标识符是源头的名称。其下游的责任是根据需要处理这一信息（如果需要处理）。Origin消息总是在事务中任何DML消息之前被发送。

每个DML消息中都包含一个任意的关系ID，它可以被映射到Relation消息中的一个ID。Relation消息描述给定关系的模式。为一个给定的关系发送Relation消息的时机是：在当前会话中第一次为该关系发送DML消息，或者从上一次该关系的Relation消息以后该关系的定义发生改变。协议假定客户端有能力缓存够用的关系元数据。

53.6. 消息数据类型

本节描述消息里用到的基本数据类型。

Intn(i)

一个网络字节序（高位在前）的n位整数。 如果指定了i， 它就是将出现的准确值，否则该值就是一个变量。如 Int16、Int32(42)。

Intn[k]

一个k个n位整数的数组， 每个都是以网络字节序表示的。数组长度k总是由消息前面的域来判断的。比如，Int16[M]。

String(s)

一个（C风格的）空值结束的字符串。对字符串没有特别的长度限制。如果指定了s，那么它是将出现的确切的值， 否则这个值就是一个变量。比如，String, String("user")。

> **注意**
>
> 后端能返回的字符串的长度没有预定义的限制。所以对前端比较好的编码策略是使用某种可扩展的缓冲区，以便能接受任何能放进内存里的东西。如果那样做不可行，则读取完整的字符串然后抛弃不能放进固定大小缓冲区的尾部字符。

Byten(c)

精确的n字节。 如果域宽度n不是一个常量， 那么我们总是可以从消息中更早的域中判断它。如果指定了c，那么它就是确切数值。 例如，Byte2, Byte1('\n')。

53.7. 消息格式

本节描述各种消息的详细格式。每种消息都标记来指示它是由前端（F）、后端（B）或者两者（F & B）发送的。 请注意，尽管每条消息在开头都包含一个字节计数，但是消息格式也被定义为无需参考字节计数就可以找到消息的结尾。 这样是为了有效性检查（CopyData消

息是一个例外，因为它形成一个数据流的一部分；任意独立的CopyData消息可能是无法自解释的）。

AuthenticationOk (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括这个长度本身。

Int32(0)

指定该认证是成功的。

AuthenticationKerberosV5 (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括长度自身。

Int32(2)

指定要求Kerberos V5认证。

AuthenticationCleartextPassword (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括长度自身。

Int32(3)

指定要求一个明文的口令。

AuthenticationMD5Password (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(12)

以字节计的消息内容的长度，包括长度本身。

Int32(5)

指定要求一个MD5加密的口令。

Byte4

加密口令的时候使用的盐粒。

AuthenticationSCMCredential (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(8)

以字节计的消息内容长度，包括长度本身。

Int32(6)

指定请求一个SCM信任消息。

AuthenticationGSS (B)

Byte1('R')

标识该消息是一个认证请求。

Int32(8)

Int32(7)

指定被请求的是GSSAPI认证。

AuthenticationSSPI (B)

Byte1('R')

标识该消息是一个认证请求。

Int32(8)

Int32(9)

指定被请求的是SSPI认证。

AuthenticationGSSContinue (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

Int32(8)

指定该消息包含GSSAPI或SSPI数据。

Byten

GSSAPI或SSPI认证数据。

AuthenticationSASL (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(10)

指定SASL认证被要求。

消息体是一个SASL认证机制的列表，该列表按照服务器偏爱的顺序组织。在最后一个认证机制名称的后面需要一个零字节作为终结符。对于每一种机制，有下列信息：

String

一种SASL认证机制的名称。

AuthenticationSASLContinue (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(11)

指定这个消息包含一个SASL挑战。

Byten

SASL数据，与使用的SASL机制有关。

AuthenticationSASLFinal (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(12)

指定SASL认证已经完成。

Byten

SASL产出的“额外数据”，与使用的SASL机制有关。

BackendKeyData (B)

Byte1('K')

标识该消息是一个取消键数据。如果前端希望能够在稍后发出CancelRequest消息，那么它必须保存这个值。

Int32(12)

以字节计的消息内容的长度，包括长度本身。

Int32

后端的进程号（PID）。

Int32

此后端的密钥。

Bind (F)

Byte1('B')

标识该信息是一个绑定命令。

Int32

以字节计的消息内容的长度，包括长度本身。

String

目标入口的名字（空字符串则选取未命名的入口）。

String

源预备语句的名字（空字符串则选取未命名的预备语句）。

Int16

后面跟着的参数格式代码的数目（由下文的C说明）。 这个数值可以是零，表示没有参数，或者是参数都使用缺省格式（文本）； 也可以是一，这种情况下指定的格式代码被应用于所有参数；或者它可以等于实际参数的数目。

Int16[C]

参数格式代码。目前每个都必须是零（文本）或者一（二进制）。

Int16

后面跟着的参数值的数目（可能为零）。这些必须和查询需要的参数个数匹配。

然后，每个参数都会出现下面的域对：

Int32

参数值的长度，以字节计（这个长度并不包含长度本身）。可以为零。 一个特殊的情况是，-1 表示一个NULL参数值。在NULL 的情况下， 后面不会跟着字节值。

Byten

参数值，使用关联的格式代码表示的格式。n是上文的长度。

在最后一个参数之后，出现下面的域：

Int16

后面跟着的结果列格式代码数目（下文的R描述）。 这个数目可以是零表示没有结果列或者结果列都使用缺省格式（文本）； 也可以是一，这种情况下指定的格式代码被应用于所有结果列（如果有的话）；或者它可以等于查询的结果列的实际数目。

Int16[R]

结果列格式代码。目前每个必须是零（文本）或者一（二进制）。

BindComplete (B)

Byte1('2')

标识该消息为一个绑定结束标识符。

Int32(4)

以字节计的消息长度，包括长度本身。

CancelRequest (F)

Int32(16)

以字节计的消息长度。包括长度本身。

Int32(80877102)

取消请求代码。该值被选中在高16位包含1234，并且在低16位包含 5678（为避免混淆，这个代码不能和任何协议版本号相同）。

Int32

目标后端的进程号（PID）。

Int32

目标后端的密钥。

Close (F)

Byte1('C')

标识这条消息是一个Close命令。

Int32

以字节计的消息内容长度，包括长度本身。

Byte1

'S'关闭一个准备的语句；或者'P'关闭一个入口。

String

一个要关闭的预备语句或者入口的名字（一个空字符串选择未命名的预备语句或者入口）。

CloseComplete (B)

Byte1('3')

标识消息是一个Close完成指示器。

Int32(4)

以字节计的消息内容的长度，包括长度本身。

CommandComplete (B)

Byte1('C')

标识此消息是一个命令结束响应。

Int32

以字节计的消息内容的长度，包括长度本身。

String

命令标记。它通常是一个单字，标识被完成的SQL命令。

对于INSERT命令，该标记是INSERT oid rows， 其中rows是已被插入的行数。oid是在rows为 1并且目标表有OID时已插入行的对象ID；否则oid就是 0。

对于DELETE命令，该标记是DELETE rows， 其中rows是已被删除的行数。

对于UPDATE命令，该标记是UPDATE rows，其中rows是已被更新的行数。

对于SELECT或CREATE TABLE AS命令，该标记是SELECT rows，其中rows是被检索的行数。

对于MOVE命令，该标记是MOVE rows，其中rows是游标位置被移动的行数。

对于FETCH命令，该标记是FETCH rows，其中rows是已从游标中检索出来的行数。

对于COPY命令，该标记是COPY rows，其中rows是已拷贝的行数（注意，行计数只在PostgreSQL 8.2及其后的版本中出现）。

CopyData (F & B)

Byte1('d')

标识这条消息是一个COPY数据。

Int32

以字节计的消息内容的长度，包括长度本身。

Byten

构成COPY数据流的一部分的数据。从后端发出的消息总是对应单一的数据行，但是前端发出的消息可能会任意分割数据流。

CopyDone (F & B)

Byte1('c')

标识这条信息是一个COPY结束指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

CopyFail (F)

Byte1('f')

标识这条消息是一个COPY失败指示器。

Int32

以字节计的消息内容的长度，包括长度本身。

String

一个报告失败原因的错误消息。

CopyInResponse (B)

Byte1('G')

标识这条消息是一条Start Copy In（开始拷贝入）响应消息。前端现在必须发送拷贝入数据（如果还没准备好做这些事情，那么发送一条CopyFail消息）。

Int32

以字节计的消息内容的长度，包括长度本身。

Int8

0表示全体拷贝格式都是文本（数据行由新符分隔， 列由分隔字符分隔等等）。1 表示全体拷贝格式都是二进制的（类似于DataRow 格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据中的列数（由下文的N解释）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。 如果全体拷贝格式都是文本，那么所有的都必须是零。

CopyOutResponse (B)

Byte1('H')

标识这条消息是一条Start Copy Out（开始拷贝出）响应消息。这条消息后面将跟着拷贝出数据。

Int32

以字节计的消息内容的长度，包括它自己。

Int8

0表示全体拷贝格式都是文本（数据行由新符分隔， 列由分隔字符分隔等等）。1 表示全体拷贝格式都是二进制的（类似于DataRow 格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据的列数（在下文的N说明）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。 如果全体拷贝格式都是文本，那么所有的都必须是零。

CopyBothResponse (B)

Byte1('W')

标识这个消息是一个Start Copy Both（启动双向复制）响应。这个消息只用于流复制。

Int32

以字节计的消息内容的长度，包括长度自身。

Int8

0表示全体COPY格式都是文本（数据行由新符分隔，列由分隔字符分隔等等）。1 表示全体拷贝格式都是二进制的（类似于DataRow格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据中的列数目（在下文的N说明）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。 如果全体拷贝格式都是文本，那么所有的代码都必须是零。

DataRow (B)

Byte1('D')

标识这个消息是一个数据行。

Int32

以字节计的消息内容的长度，包括长度自身。

Int16

后面跟着的列值的个数（可能是零）。

然后，为每个列都会出现下面的域对：

Int32

列值的长度，以字节计（这个长度不包括它自己）。可以为零。一个特殊的情况是，-1表示一个NULL的域值。 如果是NULL的情况则后面不会跟着值字节。

Byten

一个列的数值，以相关的格式代码指示的格式展现。n是上文的长度。

Describe (F)

Byte1('D')

标识该消息是一个Describe（描述）命令。

Int32

以字节计的消息内容的长度，包括字节本身。

Byte1

'S'描述一个预备语句；或者 'P' 描述一个入口。

String

要描述的预备语句或者入口的名字（或者一个空字符串，就会选取未命名的预备语句或者入口）。

EmptyQueryResponse (B)

Byte1('I')

标识这条消息是对一个空查询字符串的响应（这个消息替换了CommandComplete）。

Int32(4)

以字节计的消息内容长度，包括它自己。

ErrorResponse (B)

Byte1('E')

标识该消息是一条错误。

Int32

以字节计的消息内容的长度，包括长度本身。

消息体由一个或多个标识域组成，后面跟着一个零字节作为终止符。域可以以任何顺序出现。对于每个域都有下面的东西：

Byte1

一个标识域类型的代码；如果为零，这就是消息终止符并且不会跟着有字符串。目前定义的域类型在第 53.8 节列出。由于将来可能增加更多的域类型，所以前端应该默默地忽略未识别类型的域。

String

域值。

Execute (F)

Byte1('E')

标识该消息是一个Execute命令。

Int32

以字节计的消息内容的长度，包括长度自身。

String

要执行的入口的名字（空字符串选择未命名的入口）。

Int32

要返回的最大行数，如果入口包含返回行的查询（否则忽略）。零表示“无限制”。

Flush (F)

Byte1('H')

标识该消息是一条Flush命令。

Int32(4)

以字节计的消息内容的长度，包括长度本身。

FunctionCall (F)

Byte1('F')

标识该消息是一个函数调用。

Int32

以字节计的消息内容的长度，包括长度本身。

Int32

指定要调用的函数的对象ID（OID）。

Int16

后面跟着的参数格式代码的数目（用下文的C表示）。 它可以是零，表示没有参数，或者是所有参数都使用缺省格式（文本）； 也可以是一，这种情况下声明的格式代码被应用于所有参数；或者它可以等于参数的实际个数。

Int16[C]

参数格式代码。目前每个必须是零（文本）或者一（二进制）

Int16

指定提供给函数的参数个数。

然后，对每个参数都出现下面域对：

Int32

以字节计的参数值的长度（不包括长度本身）。可以为零。一个特殊的例子是，-1表示一个NULL参数值。如果是NULL，则没有参数字节跟在后面。

Byten

参数的值，格式由相关的格式代码指示。n是上文的长度。

在最后一个参数之后，出现下面的域：

Int16

函数结果的格式代码。目前必须是零（文本）或者一（二进制）。

FunctionCallResponse (B)

Byte1('V')

标识这条消息是一个函数调用结果。

Int32

以字节计的消息内容长度，包括长度本身。

Int32

以字节计的函数结果值的长度（不包括长度本身）。可以为零。一个特殊的情况是，-1表示NULL函数结果。如果是NULL则后面没有值字节跟随。

Byten

函数结果的值，格式由相关联的格式代码指示。n是上文的长度。

GSSResponse (F)

Byte1('p')

标识该消息是一个GSSAPI或SSPI响应。注意这也被用于SASL和口令响应消息。准确的消息类型可以从上下文中得出。

Int32

消息内容的长度，以字节为单位，包括其自身。

Byten

GSSAPI/SSPI相关的消息数据。

NegotiateProtocolVersion (B)

Byte1('v')

标识该消息是一个协议版本协商消息。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32

对于客户端请求的主协议版本，服务器能支持的最新的次协议版本。

Int32

服务器无法识别的协议选项数目。

然后，对于服务器无法识别的协议选项，还有下列信息：

String

选项名称。

NoData (B)

Byte1('n')

标识这条消息是一个无数据指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

NoticeResponse (B)

Byte1('N')

标识这条消息是一个通知。

Int32

以字节计的消息内容长度，包括长度本身。

消息体由一个或多个标识域组成，后面跟着零字节作为中止符。域可以以任何顺序出现。对于每个域，都有下面的东西：

Byte1

一个标识域类型的代码；如果为零，那么它就是消息终止符，并且后面不会跟着字符串。目前定义的域类型在第 53.8 节列出。由于将来可能会增加更多域类型，所以前端应该将不能识别的域安静地忽略掉。

String

域值。

NotificationResponse (B)

Byte1('A')

标识这条消息是一个通知响应。

Int32

以字节计地消息内容地长度，包括长度本身。

Int32

通知后端进程的进程ID。

String

通知被抛出的通道的名字。

String

从通知进程传递过来的“载荷”字符串。

ParameterDescription (B)

Byte1('t')

标识该消息是一个参数描述。

Int32

以字节计的消息内容长度，包括长度本身。

Int16

语句所使用的参数的个数（可以为零）。

然后，对每个参数，有下面的东西：

Int32

指定参数数据类型的对象ID。

ParameterStatus (B)

Byte1('S')

标识这条消息是一个运行时参数的状态报告。

Int32

以字节计的消息内容的长度，包括长度自身。

String

被报告的运行时参数的名字。

String

参数的当前值。

Parse (F)

Byte1('P')

标识该消息是一条Parse命令。

Int32

以字节计的消息内容的长度，包括长度自身。

String

目的预备语句的名字（空字符串选取未命名的预备语句）。

String

要分析的查询字符串。

Int16

指定的参数数据类型的数目（可以为零）。请注意这个参数并不表示可能在查询字符串里出现的参数个数， 只是前端希望预先为其指定类型的参数数目。

然后，对每个参数，有下面的东西：

Int32

指定参数数据类型的对象ID。这里为零等效于不指定该类型。

ParseComplete (B)

Byte1('1')

标识该消息是一个Parse完成指示器。

Int32(4)

以字节计的消息内容长度，包括长度自身。

PasswordMessage (F)

Byte1('p')

标识该消息是一个口令响应。注意这也被用于GSSAPI、SSPI以及SASL响应消息。确切的消息类型可以从上下文推出。

Int32

以字节计的消息内容的长度，包括长度本身。

String

口令（如果要求了，就是加密后的）。

PortalSuspended (B)

Byte1('s')

标识这条消息是一个入口暂停指示器。请注意这个消息只出现在达到一条Execute消息的行计数限制的时候。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

Query (F)

Byte1('Q')

标识该消息是一个简单查询。

Int32

以字节计的消息内容的长度，包括长度自身。

String

查询字符串自身。

ReadyForQuery (B)

Byte1('Z')

标识消息的类型。在后端为新的查询周期准备好的时候，总会发送ReadyForQuery。

Int32(5)

以字节计的消息内容的长度，包括长度本身。

Byte1

当前后端事务状态指示器。可能的值是空闲状况下的'I'（不在事务块里）；在事务块里是'T'； 或者在一个失败的事务块里是'E'（在事务块结束之前，任何查询都将被拒绝）。

RowDescription (B)

Byte1('T')

标识该消息是一个行描述。

Int32

以字节计的消息内容的长度，包括长度自身。

Int16

指定在一个行里面的域的数目（可以为零）。

然后对于每个字段，有下面的东西：

String

字段名字。

Int32

如果域可以被标识为一个指定表的列，这里就是表的对象ID；否则就是零。

Int16

如果该域可以被标识为一个指定表的列，这里就是该列的属性号；否则就是零。

Int32

域数据类型的对象ID。

Int16

数据类型尺寸（参阅pg_type.typlen）。请注意负值表示变宽类型。

Int32

类型修饰词（参阅pg_attribute.atttypmod）。 修饰词的含义是类型相关的。

Int16

用于该域的格式码。目前会是零（文本）或者一（二进制）。 在Describe语句的变体返回的RowDescription里，格式码还是未知的，因此总是零。

SASLInitialResponse (F)

Byte1('p')

标识该消息是一个初始SASL响应。注意这也被用于GSSAPI、SSPI以及口令响应消息。确切的消息类型可以从上下文推知。

Int32

消息内容的长度，以字节为单位，包括其自身。

String

客户端选择的SASL认证机制的名称。

Int32

后面跟着的SASL机制相关的"Initial Client Response"的长度，如果没有Initial Response则为-1。

Byten

SASL机制相关的"Initial Response"。

SASLResponse (F)

Byte1('p')

标识该消息是一个初始SASL响应。注意这也被用于GSSAPI、SSPI以及口令响应消息。确切的消息类型可以从上下文推知。

Int32

消息内容的长度，以字节为单位，包括其自身。

Byten

SASL机制相关的消息数据。

SSLRequest (F)

Int32(8)

以字节计的消息内容的长度，包括长度本身。

Int32(80877103)

SSL请求码。选取的值在高16位里包含1234，在低16位里包含5679 （为了避免混淆，这个编码必须和任何协议版本号不同）。

StartupMessage (F)

Int32

以字节计的消息内容长度，包括长度自身。

Int32(196608)

协议版本号。高16位是主版本号（对这里描述的协议而言是 3）。低16位是次版本号（对于这里描述的协议而言是 0）。

协议版本号后面跟着一个或多个参数名和值字符串的对。要求在最后一个名字/数值对后面有个零字节作为终止符。 参数可以以任意顺序出现。user是必须的，其它都是可选的。每个参数是这样指定的：

String

参数名。目前可以识别的名字是：

user

用于连接的数据库用户名。必须；无缺省。

database

要连接的数据库。缺省是用户名。

options

给后端的命令行参数（这个特性已经废弃，更好的方法是设置单独的运行时参数）。 这个字符串中的空格会被当做参数的分隔符，除非用一个反斜线（\）对它转义。写\\可表示一个而字面意义上的反斜线。

replication

用于连入流复制模式，其中可以发出复制命令的一个小型集合而不是SQL语句。值可以是true、false或者database，默认值是false。详情请参考第 53.4 节

除了上述参数之外，还可以列出其他参数。以_pq_.开头的参数名被保留给协议扩展之用，而其他的参数名被当做在后端开始时要设置的运行时参数。这类设置将在后端启动期间被应用（如果有命令行参数，则在解析完命令行参数之后）并且将作为会话的默认值。

String

参数值。

Sync (F)

Byte1('S')

表示该消息为一条 Sync 命令。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

Terminate (F)

Byte1('X')

标识本消息是一个终止。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

53.8. 错误和通知消息域

本节描述那些可能出现在ErrorResponse和NoticeResponse消息里的域。 每个域类型有一个单字节标识记号。请注意，任意给定的域类型在每条消息里应该最多出现一次。

S

严重性：该域的内容是ERROR、FATAL或者PANIC（在一个错误消息里），或者WARNING、NOTICE、DEBUG、INFO或者LOG（在一条通知消息里），或者是这些形式的某种本地化翻译。总是会出现。

V

严重性：该域的内容是ERROR、FATAL或者PANIC（在一个错误消息里），或者WARNING、NOTICE、DEBUG、INFO或者LOG（在一条通知消息里）。这和S域相同，不过其内容没有被本地化。这只存在于PostgreSQL版本 9.6 及其后版本产生的消息中。

C

代码：错误的SQLSTATE代码（参阅附录 A 。非本地化。总是出现。

M

消息：人类可读的错误消息的主体。这些信息应该准确并且简洁（通常是一行）。总是出现。

D

细节：一个可选的二级错误消息，携带了有关问题的更多错误消息。可以是多行。

H

提示：一个可选的有关如何处理问题的建议。它和细节不同的地方是它提出了建议（可能并不合适）而不仅仅是事实。可以是多行。

P

位置：这个域值是一个十进制ASCII整数，表示一个错误游标的位置，它是一个指向原始查询字符串的索引。第一个字符的索引是 1，位置是以字符计算而非字节计算的。

p

内部位置：这个域和P域定义相同，但是它被用于当游标位置指向一个内部生成的命令的情况， 而不是用于客户端提交的命令。这个域出现的时候，总是会出现q域。

q

内部查询：失败的内部生成的命令的文本。比如，它可能是一个PL/pgSQL函数发出的 SQL 查询。

W

哪里：一个指示错误发生的环境的指示器。目前，它包含一个活跃的过程语言函数的调用堆栈的路径和内部生成的查询。 这个路径每个项记录一行，最新的在最前面。

s

模式名：如果错误与一个指定数据库对象相关，这里是包含该对象的模式名（如果有）。

t

表名：如果错误与一个指定表相关，这里是表的名字（引用该表模式的名字的模式名域）。

c

列名：如果错误与一个指定表列相关，这里是该列的名字（引用该模式和表的名字来标识该表）。

d

数据类型名：如果错误与一个指定数据类型相关，这里是该数据类型的名字（引用该数据类型模式的名字的模式名域）。

n

约束名：如果错误是和一个指定约束相关，这里是该约束的名字。引用至上面列出的相关表或域的域（为了这个目的，索引被视作约束，即使它们并不是按照约束语法创建的）。

F

文件：报告的错误在源代码中的文件名。

L

行：报告的错误所在的源代码的位置的行号。

R

例程：报告错误的例程在源代码中的名字。

> **注意**
>
> 用于模式名、表名、列名、数据类型名和约束名只提供给有限的几种错误类型；请参考附录 A 前端不应当假设任何一个这些域的出现会保证其他域的出现。核心错误资源会看到上面提示的相互关系，但是用户定义的函数可能会以其他方式使用这些域。同样的道理，客户端不应该假设这些域表示当前数据库中同一时期的对象。

客户端负责格式化要显示的信息以符合需要；特别是它应该根据需要断开长的行。在错误消息域里出现的新行字符应该被当作一个分段的符号，而不是换行。

53.9. 逻辑复制消息格式

这一节介绍每一种逻辑复制消息的详细格式。这些消息会通过复制槽的SQL接口返回或者由walsender发送。在由walsender发送的情况下，它们被封装在第 53.4 节所述的复制协议WAL消息中，并且通常服从和物理复制相同的消息流。

Begin

Byte1('B')

标识该消息是一个开始消息。

Int64

该事务的最终LSN。

Int64

该事务的提交时间戳。该值是自PostgreSQL纪元（2000-01-01）以来的微秒数。

Int32

事务的XID。

Commit

Byte1('C')

标识该消息是一个提交消息。

Int8

标志，当前未使用（必须为0）。

Int64

提交的LSN。

Int64

该事务的结束LSN。

Int64

该事务的提交时间戳。该值是自PostgreSQL纪元（2000-01-01）以来的微秒数。

Origin

Byte1('O')

标识该消息是一个源头消息。

Int64

源头服务器上的提交LSN。

String

源头的名称。

注意在一个单一事务中可能有多个Origin消息。

Relation

Byte1('R')

标识该消息是一个关系消息。

Int32

关系的ID。

String

名字空间（pg_catalog是空字符串）。

String

关系名。

Int8

该关系的副本标识设置（和pg_class中的relreplident相同）。

Int16

列数。

接下来，会为每一列出现下面的消息部分：

Int8

该列的标志。当前可以是0（无标志）或者1（标记该列为键的一部分）。

String

列的名称。

Int32

列的数据类型的ID。

Int32

列的类型修饰符（atttypmod）。

Type

Byte1('Y')

标识该消息是一个类型消息。

Int32

数据类型的ID。

String

名字空间（pg_catalog的是空字符串）。

String

数据类型的名称。

Insert

Byte1('I')

标识该消息是一个插入消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('N')

标识接下来的TupleData消息是一个新元组。

TupleData

TupleData消息部分表示新元组的内容。

Update

Byte1('U')

标识该消息是一个更新消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('K')

标识接下来的TupleData子消息为一个键。这个字段是可选的并且仅当任意列中被更新更改的数据是REPLICA IDENTITY索引的一部分时才存在。

Byte1('O')

标识接下来的TupleData子消息为一个旧元组。这个字段是可选的并且仅当发生更新的表的REPLICA IDENTITY被设置为FULL时才存在。

TupleData

TupleData消息部分表示旧元组或主键的内容。仅当之前的'O'或'K'部分存在时才存在。

Byte1('N')

标识接下来的TupleData消息是一个新元组。

TupleData

TupleData消息部分表示一个新元组的内容。

Update消息可能包含一个'K'消息部分或者一个'O'消息部分，或者两者都不包含，但是绝不会同时包含两者。

Delete

Byte1('D')

标识该消息是一个删除消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('K')

标识接下来的TupleData子消息是一个键。如果发生删除的表正好用一个索引作为REPLICA IDENTITY，那么就会存在这个字段。

Byte1('O')

标识接下来的TupleData子消息为一个旧元组。这个字段是可选的并且仅当发生删除的表的REPLICA IDENTITY被设置为FULL时才存在。

TupleData

TupleData消息部分表示旧元组或主键的内容，取决于前面的字段。

Delete消息可能包含一个'K'消息部分或者一个'O'消息部分，或者两者都不包含，但是绝不会同时包含两者。

Truncate

Byte1('T')

标识该消息是一个截断消息。

Int32

关系的数目。

Int8

TRUNCATE的选项位：1表示CASCADE，2表示RESTART IDENTITY

Int32

对应于关系消息中的ID的关系的ID。这个字段会为每个关系重复出现。

下面的消息部分由上面的消息共享。

TupleData

Int16

列数。

接下来，为每一列会有下列子消息之一出现：

Byte1('n')

标识该数据为NULL值。

或者

Byte1('u')

标识未更改的被TOAST过的值（实际值没有被发送）。

或者

Byte1('t')

标识数据为文本格式化的值。

Int32

列值的长度。

Byten

该列的值的文本格式（未来的发行可能会支持额外的格式）。n是上面的长度。

53.10. 自协议2.0以来的变化总结

本节提供一个快速的改变检查列表，以便于那些试图将现有的客户端库更新到3.0协议的开发人员。

初始化的启动包用了一个灵活的字符串列表格式取代了固定的格式。请注意，运行时参数的会话缺省值现在可以直接在启动包中指定（实际上，你可以在使用options域之前干这件事情，但是因为options的宽度限制以及缺乏引用值中空白的方法，这并不是很安全的技巧）。

现在所有的消息在消息类型字节后面都有一个长度计数（除了启动包之外，它没有类型字节）。同时还要注意现在PasswordMessage有一个类型字节。

ErrorResponse和NoticeResponse（'E' 和 'N'）消息现在包含多个域， 从这些域里客户端代码可以组合出自己所希望的详细程度的错误消息。请注意个体的域通常不是用新行终止的，虽然在老的协议里发送的单个字符串总是会用新行终止。

ReadyForQuery（'Z'）消息包括一个事务状态指示符。

BinaryRow和DataRow消息类型之间的区别不再存在了；单个DataRow消息类型用于返回所有格式的数据。请注意DataRow的布局已经被变得更容易分析。同样，二进制值的表现形式已经改变了：它不再直接和服务器的内部表现形式绑定。

有了一种新的“扩展查询”的子协议，它增加了前端消息类型 Parse、Bind、Execute、Describe、Close、Flush和Sync，以及后端消息类型 ParseComplete、 BindComplete、PortalSuspended、ParameterDescription、NoData和CloseComplete。现有的客户端不用关心这个子协议，但是利用这个子协议将令我们可能提升性能或者功能。

COPY数据现在封装到了CopyData和CopyDone消息里。现在有种很好的方法从正在进行的COPY动作中的错误恢复。 最后一行的特殊的“\.”不再需要了，并且在COPY OUT的过程中不再发送（在COPY IN的时候它仍然被认为是一个终止符，但是它的使用已经废弃了并且最终将被删除）。 现在支持二进制COPY。CopyInResponse和CopyOutResponse消息包括指示列数目和每个列格式的域。

FunctionCall和FunctionCallResponse消息的布局变化了。FunctionCall现在支持给函数传递NULL参数。它同样可以处理以文本或者二进制格式传递参数和检索结果。我们不用再认为FunctionCall有潜在的安全性漏洞，因为它并不提供对内部服务器数据表现形式的直接访问。

在连接启动期间，后端会为它认为客户端库感兴趣的所有参数发送ParameterStatus（'S'）消息。随后，如果这些参数的任何活跃值发生变化，那么将发送一条ParameterStatus消息。

RowDescription（'T'）消息为所描述的行的每个列运载新表的OID和列编号域。它同样还表示每个列的格式代码。

后端不再生成 CursorResponse（'P'）消息。

NotificationResponse（'A'）消息有一个附加的字符串域，它能携带来自NOTIFY事件发送者的一个“载荷”字符串。

EmptyQueryResponse（'I'）以前包含一个空字符串参数；现在已经被删除。

第 54 章 PostgreSQL编码习惯

54.1. 格式化

源代码格式化使用 4 列制表间隔，并且保留制表符（即制表符不会被扩展成空格）。每一个逻辑缩进层就是一个额外的制表位。

布局规则（括号定位等）遵循 BSD 习惯。特别地，if、while、switch等受控块的花括号要独自占据一行。

限制行长度，这样在 80 列窗口中代码也是可读的（这并不意味着你不能超过 80 列。例如，为了保持代码在 80 列以内而在任意位置打断一段长的错误消息字符串可能不会给可读性带来什么好处）。

不使用 C++ 风格的注释（//注释）。严格的 ANSI C 编译器不接受这样的注释。出于相同的原因，不使用 C++ 扩展，例如在块中声明新变量。

多行注释块更好的风格是

```
/*
 * 注释文本从这里开始
 * 延续到这里
 */
```

注意从列 1 开始的注释块将被pgindent原样保留，但是它将重新对缩进的注释块断行，好像它们是纯文本一样。如果你想要保留一个缩进块中的换行，可以像这样增加破折号：

```
    /*----------
     * 注释文本从这里开始
     * 延续到这里
     *----------
     */
```

虽然在提交补丁时并不是一定要遵守这些格式化规则，但是最好能遵守。在下一次发行前你提交的代码将会通过pgindent，因此使用某种其他格式化习惯无法使代码被编排得很好。对于补丁，一种比较好的经验规则是“让新代码看起来像它周围的原有代码”。

src/tools目录包含有可以用于emacs、xemacs或者vim编辑器的设置文件，它们可以帮助确保这些编辑器会根据上述习惯格式化代码。

可以这样调用文本浏览工具more和less：

```
more -x4
less -x4
```

来让它们以合适的方式显示制表符。

54.2. 在服务器中报告错误

服务器代码内产生的错误、警告和日志消息应该使用ereport或者更老的elog生成。这个函数的使用有些复杂，因此有必要做一些解释。

对于每一个消息都有两个必要的元素：一个严重性级别（从DEBUG到PANIC）和一个主消息文本。此外还有一些可选元素，其中最常见的是一个遵守 SQL 规范中 SQLSTATE 习惯的错误标

识符代码。ereport本身只是一个 shell 函数，它的存在主要是为了在语法习惯上让消息的产生更像 C 源代码中的函数调用而已。ereport直接接受的唯一参数是严重性级别。主消息文本和任何其他可选消息元素通过在ereport调用中使用辅助函数产生，例如errmsg。

对于ereport的一次典型调用可能看起来像：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

这会指定错误严重性级别为ERROR（一个普通错误）。errcode调用使用src/include/utils/errcodes.h中定义的一个宏指定 SQLSTATE 错误代码。errmsg调用提供主消息文本。注意辅助函数调用周围的额外圆括号 — 它们虽然很烦人但是在语法上是必需的。

这里有一个更复杂的例子：

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.")));
```

这展示了使用格式代码把运行时值嵌入到一个消息文本中的方法，其中还提供了一个可选的“hint”消息。

如果严重级别是ERROR或更高，ereport 会中止用户定义函数的执行并且不会返回到调用者。如果严重级别低于 ERROR，ereport会正常返回。

ereport可用的辅助例程是：

- errcode(sqlerrcode)指定对于该情况的 SQLSTATE 错误标识符代码。如果没有调用这个例程，错误严重性级别是ERROR或更高时错误标识符会默认成ERRCODE_INTERNAL_ERROR，错误级别是WARNING时标识符为ERRCODE_WARNING，否则（对于NOTICE及以下的级别）标识符会被设置为ERRCODE_SUCCESSFUL_COMPLETION。虽然这些默认值常常很方便，在忽略errcode()调用之前请总是思考一下它们是否合适。

- errmsg(const char *msg, ...)指定主错误消息文本，以及需要插入其中的运行时值。这种插入以sprintf-风格的格式代码指定。除了sprintf接受的标准格式代码，格式代码%m可以用来插入由strerror为errno的当前值返回的错误消息。 [1] %m不要求errmsg参数列表中的任何对应项。注意在格式代码被处理之前，消息字符串将通过gettext来进行可能的本地化。

- errmsg_internal(const char *msg, ...)与errmsg相同，不过消息串将不会被翻译，也不会被包括在国际化的消息字典中。这不应该被用于“不能发生”的情况中，因为那些情况下不值得花费精力去做翻译。

- errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)很像errmsg，但是支持消息的多种复数形式。fmt_singular是英语单数格式，fmt_plural是英语复数格式，n是决定需要何种复数形式的整数值，剩下的参数会被根据选中的格式字符串进行格式化。详见第 55.2.2 节

- errdetail(const char *msg, ...)提供了一个可选的“详情”消息，如果有额外的信息但不适合放在主消息中时就可以使用这种方式。消息字符串的处理与errmsg相同。

[1] 也就是说，该值时到达ereport调用时的当前值，在辅助报告例程内errno的改变不会影响它。但如果你在errmsg的参数列表中显式地写了strerror(errno)则不是这样，因此不要那样做。

- errdetail_internal(const char *msg, ...)与errdetail相同，不过消息串将不会被翻译，也不会被包括在国际化的消息字典中。这应该被用于不值得花费精力翻译的详情消息上，例如它们对大部分用户太过技术化而没什么用处。
- errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)与errdetail相似，但是支持消息的多种复数形式。详见第 55.2.2 节
- errdetail_log(const char *msg, ...)与errdetail相同，除了这个字符串只会进入服务器的日志而不会发往客户端。如果同时使用了errdetail（或者上述的一种等效函数）以及errdetail_log，那么一个字符串会被发往客户端而另一个会被发往日志。如果错误细节的安全性过于敏感或者体积过于庞大而不适合于发往客户端，这个函数就非常有用。
- errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)与errdetail_log 相似，但是支持多种复数形式的消息。详见第 55.2.2 节
- errhint(const char *msg, ...)提供一个可选的“hint”消息，它被用来提供关于如何修复该问题的建议。该消息字符串以和errmsg相同的方式处理。
- errcontext(const char *msg, ...)通常不会被直接从一个ereport消息站点调用，它被用在error_context_stack回调函数中来提供错误发生的上下文，例如一个 PL 函数中的当前位置。该消息字符串以和errmsg相同的方式处理。不同于其他辅助函数，在每次ereport调用中可以多次调用这个函数，这样提供的连续的字符串将被用单独的新行串接在一起。
- errposition(int cursorpos)指定一个查询字符串中错误的文本位置。当前，它只对查询处理的词法和语法分析阶段中检测到的错误有用。
- errtable(Relation rel)指定一个关系，它的名称和模式名称应该被包括在错误报告中作为辅助域。
- errtablecol(Relation rel, int attnum)指定一个列，它的名称、表名和模式名称应该被包括在错误报告中作为辅助域。
- errtableconstraint(Relation rel, const char *conname)指定一个约束，它的名称、表名和模式名称应该被包括在错误报告中作为辅助域。索引应当为考虑成用于这种目的的约束，不管它们有没有一个相关联的pg_constraint项。要小心地以rel传递底层堆关系而不是索引本身。
- errdatatype(Oid datatypeOid)指定一个数据类型，它的名称和模式名称应该被包括在错误报告中作为辅助域。
- errdomainconstraint(Oid datatypeOid, const char *conname)指定一个域约束，它的名称、域名和模式名称应该被包括在错误报告中作为辅助域。
- errcode_for_file_access()是一个便捷函数，它可以在一个文件访问相关的系统调用中为一个失败选择一个合适的 SQLSTATE 错误标识符。它使用保存下来的errno来决定要差生哪种错误代码。通常，应该把它和主错误消息文本中的%m联合使用。
- errcode_for_socket_access()是一个便捷函数，它可以在一个套接字相关的系统调用中为一个失败选择一个合适的 SQLSTATE 错误标识符。
- errhidestmt(bool hide_stmt)可以被调用来指定 postmaster 日志中一个消息的STATEMENT:部分的禁止。通常如果该消息文本已经包括当前语句这就是合适的。
- 可以调用errhidecontext(bool hide_ctx)来指示抑制 postmaster 日志中消息里的CONTEXT:部分。这只应该被用于 verbose 模式的调试消息，这类消息中被重复包含的上下文信息会让日志容量膨胀得非常厉害。

> **注意**
>
> 在一个ereport调用中，最多可以使用一个errtable、errtablecol、errtableconstraint、errdatatype或

> 者errdomainconstraint函数。这些函数存在是为了允许应用抽取与错误情况相关的数据库对象名，而不需要检查可能已被本地化的错误消息文本。这些函数应该被用在应用需要对其进行自动错误处理的错误报告中。从PostgreSQL 9.3 开始，完整的覆盖只为 SQLSTATE 类别 23 中的错误存在（完整性约束违背），但是在未来这些很可能会被扩展。

有一个还在大量使用的旧的函数elog。一个elog调用：

```
elog(level, "format string", ...);
```

完全等效于：

```
ereport(level, (errmsg_internal("format string", ...)));
```

注意 SQLSTATE 错误代码总是被给予默认值，并且消息字符串不受翻译限制。因此，elog应该只被用于内部错误和低层次的调试日志。任何普通用户感兴趣的消息应该通过ereport。不管怎样，在仍广泛使用elog的系统中，有足够多的内部“不可能发生”的错误检查，出于记号简洁的目的这更适合于那些消息。

有关编写好的错误消息的建议可见第 54.3 节

54.3. 错误消息风格指导

提供这个风格指导是希望在所有由PostgreSQL产生的消息间维护一种一致的、用户友好的风格。

哪儿该放什么

主要的消息应该简短、实事求是并且避免引用诸如特定函数名这样的实现细节。“简短”意味着“在正常情况下应该能放在一行内”。如果需要保持主要消息简短，或者如果你觉得有必要提到诸如特定系统调用失败这样的实现细节，可以使用一个详细消息。主要消息和详细消息都应该实事求是。对于有关如何修复问题的建议可以使用一个提示消息，特别是该建议可能不总是合适时。

例如，与其用：

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

不如写：

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

原理：保持主要消息简短有助于扣住主题并且有助于客户端基于错误消息放在一行足以的假设来安排屏幕空间。详细消息和提示消息可以被归入一种详情模式，或者可能是一种弹出式错误细节窗口。还有，详细消息和提示消息通常会被排除在服务器日志之外以节省空间。最好避免对实现细节的引用，因为用户根本就不想知道细节。

格式化

不要对消息文本的格式化做任何特定假设。可以期望客户端和服务器日志换行来适应其自身的需要。在长消息中，新行字符（\n）可以被用来指示建议的分段。不要用一个新行结束一

个消息。不要使用制表符或其他格式化字符（在错误上下文显示中，新行会自动被增加来分割上下文的层次，例如函数调用）。

原理：消息不一定非得显示在终端上。在 GUI 显示或浏览器中，这些格式化指令最多会被忽略。

引号

在引用时，英语文本应该使用双引号。其他语言中的文本应该一致地使用一种符合出版习惯和其他程序计算机输出的引号。

原理：选择双引号而不是单引号其实有点武断，但是更倾向于是首选用法。有人建议过按照 SQL 习惯根据对象类型来选择引号的种类（即，字符串单引号，标识符双引号）。但是这是一个很多用户根本不熟悉的语言内部的技术问题，它不会扩散到其他种类的被引用术语，也不会翻译成其他语言，并且也相当没有意义。

引号的使用

总是用引号界定文件名、用户提供的标识符以及其他可能包含词的变量。不要用它们来标记不会包含词的变量（例如，操作符名称）。

在后端有函数会根据需要对其自身的输出加上双引号（例如format_type_be()）。不要在这种函数的输出周围再加上额外的引号。

原理：在被嵌入到一个消息时，对象可能具有产生歧义的名称。对于标记一个插入名称的开始和结束要始终如一。但不要在消息中混入不必要的或者重复的引号。

语法和标点

主要错误消息的规则与详细/提示消息的规则不同：

主要错误消息：不要大写第一个字母。不要用一个句点结束一个消息。甚至不要考虑用一个感叹号结束一个消息。

详细和提示消息：使用完整的句子，并且每一个都用句点结束。对句子的第一个词进行首字母大写。如果后面跟着另一个句子，在据点后面放两个空格（对于英语文本有效，在其他语言中可能不合适）。

错误上下文字符串：不要大写第一个字母并且不要用一个句点结束字符串。上下文字符串通常不应该是完整的句子。

原理：避免标点让客户端应用能更容易地把消息嵌入到各种各样的语法上下文中。主要消息场上不是语法上完整的句子（并且如果它们比一个句子还长，它们应该被分成主要和详细部分）。不过，详细和提示消息会更长并且需要包括多个句子。为了一致，即便它们只是一个句子，它们也应该遵循完整句子的风格。

大写 vs. 小写

对消息用于使用小写形式，包括一个主要错误消息的第一个字母。如果 SQL 命令或者关键词出现在消息中，为它们使用大写形式。

原理：用这种方式能更容易使所有的东西看起来更加一致，因为一些消息是完整的句子而另一些不是。

避免被动态

使用主动语态。在有主语时使用完整句子（“A could not do B”）。如果主语是程序本身，使用没有主语的电报风格，但不要为程序使用“I”。

原理：程序不是人类。所以不要假装。

现在式 vs. 过去式

如果一次尝试做某事失败但是可能在下一次成功（也许在修复某个问题之后），则使用过去式。如果失败必定是持久的，使用现在式。

在以下形式的句子之间存在着非平凡的语义区别：

```
could not open file "%s": %m
```

和

```
cannot open file "%s"
```

第一个表示打开文件的尝试失败。该消息应该给出一个原因，例如“磁盘满”或者“文件不存在”。过去式更合适，因为下一次磁盘可能不再满或者请求的文件可能就存在了。

第二种形式指示打开所提及文件的功能在程序中根本就不存在，或者概念上是不可能的。现在式更合适，因为该情况将无限期保持。

原理：诚然，普通用户没有能力从消息的时态上得出重要的结论，但是由于语言为我们提供了语法，我们就应该正确地使用它。

对象类型

在引用一个对象的名称时，说明该对象的种类。

原理：否则没有人会了解什么“foo.bar.baz”。 refers to.

括号

方括号只被用于： (1) 在命令对照表中标记可选参数，或者 (2) 用于标记一个数组下标。

原理：任何其他用法都无法符合总所周知的习惯用法并且将使人们混乱。

组装错误消息

当一个消息包括在别处产生的文本时，这样将它嵌入：

```
could not open file %s: %m
```

原理：将这种文本粘贴到一个单一的语句中很难解释所有可能的错误代码，因此需要某种标点。也有人建议把被嵌入的文本放在圆括号中，但是如果被嵌入的文本可能是消息中最重要的一部分（这是常有的事）时就显得不自然。

错误原因

消息应该总是说明为什么错误会发生的原因。例如：

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

如果没有已知原因，你最好修复代码。

函数名

不要在错误文本中包括报告例程的名称。需要时，我们有其他机制能够知道这些，并且对于大部分用户来说这种信息没有用处。如果没有函数名错误文本就没有意义，那么请重写它。

```
BAD:    pg_atoi: error in "z": cannot parse "z"
BETTER: invalid input syntax for integer: "z"
```

也要避免提及被调用的函数，而不是说代码尝试做过什么：

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

如果它真地看起来必要，在详细消息中提及系统调用（在某些情况中，可以为详细消息提供传递给系统调用的实际值）。

原理：用户不知道所有那些函数做了什么。

应该避免的捣蛋词

Unable.　　“Unable”接近于被动态。酌情使用“cannot”或者“could not”更好。

Bad.　　“bad result”之类的错误消息实在很难被解释。最好写出为什么结果“不好”，例如“无效格式”。

Illegal.　　“Illegal”表示未被了法律，剩下的才是“invalid”。同样，说明为什么无效。

Unknown.　　尝试避免“unknown”。考虑“error: unknown response”。如果你不知道响应是什么，你怎么知道它是错误的？“Unrecognized”常常是一个更好的选择。还有，确定不要包括被抱怨的值。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

找到 vs. 存在.　　如果程序使用了一个非平凡的算法来定位一个资源（例如一个路径搜索）并且该算法失败了，说该程序无法“找到”该资源比较公平。换句话说，如果该资源应该在的位置是已知的，但是程序无法在那里访问它，那么才说该资源不“存在”。在这种情况中使用“找到”听起来很弱并且会使问题混淆。

May vs. Can vs. Might.　　“May”表示权限（例如，"You may borrow my rake."），并且在文档或错误消息中用处有限。 “Can”表示能力（例如，"I can lift that log."），而“might”表示可能性（例如，"It might rain today."）。请使用合适的词来使含义清晰并且便于翻译。

缩略形式.　　避免缩略，如“can't”，请使用“cannot”。

适当的拼写

完整地拼出单词。例如，避免：

- spec
- stats
- parens

- auth
- xact

基本原理：这将增强一致性。

本地化

记住错误消息文本需要被翻译成其他语言。请遵循第 55.2.2 节的方针以避免让翻译者为难。

54.4. 其他编码习惯

C 标准

PostgreSQL中的代码应该只依赖于 C89 标准中的语言特性。这意味着遵循 C89 的编译器肯定能编译 postgres，至少除开少数平台依赖问题之外。 如果提供了回退机制，也可以使用来自后续 C 标准版本的特性或者编译器相关的特性。

例如虽然static inline和_StaticAssert()来自于新版的 C 标准，但是PostgreSQL中仍然用到了它们。如果它们不可用，我们分别会回退到定义没有内联的函数以及使用兼容 C89 的替代品来执行相同的检查（但是会发出晦涩的消息）。

类函数的宏以及内联函数

带有参数的宏以及static inline函数都可能被使用。当类似如下情况写作宏时会有多次计算风险或者宏可能非常长时，使用后者会更好。

```
#define Max(x, y)       ((x) > (y) ? (x) : (y))
```

在其他情况下只能使用宏，或者说使用宏至少更容易。例如，由于多种类型的表达式需要被传递给宏。

当一个内联函数的定义引用只在后端中可用的符号（即变量、函数）时，从前端代码引用该函数时该函数可能不可见。

```
#ifndef FRONTEND
static inline MemoryContext
MemoryContextSwitchTo(MemoryContext context)
{
    MemoryContext old = CurrentMemoryContext;

    CurrentMemoryContext = context;
    return old;
}
#endif   /* FRONTEND */
```

在这个例子中，CurrentMemoryContext只在后端中可用，但该函数引用了它并且该函数因此被#ifndef FRONTEND隐藏。之所以存在这条规则，是因为即使内联函数中包含的符号没有被使用，有些编译器也会发出对它们的引用。

编写信号处理器

为了能适合在信号处理器中运行，代码必须被非常仔细地编写。根本问题是，除非被阻塞，信号处理器能在任何时候打断代码。如果信号处理器内部的代码使用和外面代码相同的状

态，很可能会出现混乱。例如，可以想想如果一个信号处理器试图取得已经在被打断代码中持有的锁时会发生什么。

除特殊安排的代码之外，在信号处理器中只应该调用对异步信号安全的函数（如 POSIX 中定义的那样）并且只访问volatile sig_atomic_t类型的变量。一些postgres中的函数也被视作是信号安全的，其中很重要的一个是SetLatch()。

在大部分情况下，信号处理器应该只提示一个信号已经到达，并且使用一个 latch 唤醒运行在处理器之外的代码。这样一个处理器的例子如下：

```
static void
handle_sighup(SIGNAL_ARGS)
{
    int         save_errno = errno;

    got_SIGHUP = true;
    SetLatch(MyLatch);

    errno = save_errno;
}
```

errno会被保存并且恢复，因为SetLatch()可能会更改它。如果不这样做，当前正在观测errno的被中断代码可能会看到错误的值。

调用函数指针

为了清晰，如果函数指针是一个简单的变量，在调用指向的函数时显式地对其解除引用会更好。例如：

```
(*emit_log_hook) (edata);
```

（虽然emit_log_hook(edata)还会有效）。当函数指针是一个结构的组成部分时，则额外的标点能够被省略并且通常也应该被省略，例如：

```
paramInfo->paramFetch(paramInfo, paramId);
```

第 55 章 本国语言支持

55.1. 给翻译者

PostgreSQL程序（服务器和客户端）可以用你最喜欢的语言发出它们的消息 — 如果消息已经被翻译过。常见和维护已翻译的消息集需要以该语言为母语并且愿意为PostgreSQL做贡献的人们的帮助。要做这些你根本不需要成为一个程序员。本节将解释如何来帮助我们。

55.1.1. 需求

我们将不会判断你的语言技巧 — 这一节是关于软件工具。理论上，你只需要一个文本编辑器。但是这仅仅是最少见的情况：你不想试验你的已翻译好的消息。当你配置你的源代码树时，一定要使用--enable-nls选项。这将也会检查libintl库和msgfmt程序，这些是所有终端用户将会需要的。要试验你的工作，请按安装指导的相应部分进行。

如果你想开始一个新的翻译工作或想做一次消息目录合并（在后面描述），你将分别需要程序xgettext和msgmerge的 GNU 兼容的实现。稍后，我们将试着安排它这样如果你使用一个打包好的源代码发布，你将不需要xgettext（如果使用来自 Git 的发布，你仍将需要它）。我们目前推荐GNU Gettext 0.10.36或更高版本。

你的本地 gettext 实现应该有其自身的文档。其中的某些可能与接下来的内容重复，但是你应该看看来了解更多细节。

55.1.2. 概念

原始（英语）消息和它们（可能的）已翻译等价物组成的对被保持在消息目录中，每个程序（尽管相关程序可以共享一个消息目录）以及每个目标语言一个目录。消息目录有两种文件格式：第一种是“PO”文件（用于可移植对象），它是一个翻译者编辑的具有特殊语法的普通文本文件。第二种是“MO”文件（用于机器对象），它是从相应的 PO 文件生成的一个二进制文件并且在国际化程序运行时使用。翻译者不需要处理 MO 文件；事实上几乎没有人这样做。

消息目录文件的扩展名不出意料地是.po或.mo。基础名要么是与之合作的程序名，要么是该文件的语言名称，取决于具体情况。这有一点让人困惑。例子是psql.po（用于 psql 的 PO 文件）或fr.mo（法语的 MO 文件）。

PO 文件的文件格式示例如下：

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

msgid 被从程序源代码中抽取（不是必须这样做，但是这是最常用的方法）。msgstr 行最初是空的并且被翻译者用以填充有用的字符串。字符串可以包含 C 风格的转义字符并且可以跨越行（下一行必须开始于行首）。

字符引入一个注释。如果空白紧跟着 # 字符，那么这是一个由翻译者维护的注释。也可能有自动注释，它在 # 后紧跟一个非空白字符。这些由多种操作 PO 文件并且意图辅助翻译者的多种工具维护。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

#. 风格的注释是从使用消息的源文件中抽取而来。有可能程序员已经为翻译者插入了信息，例如关于期望的对齐。#: 注释指示该消息在源代码中被使用的确切位置。翻译者不需要查看程序源代码，但是如果有关于正确翻译的疑问，他也能够去查看。#, 注释包含以某种方法描述该消息的标志。当前有两种标志：如果该消息可能由于程序源码中的改变而过时，则fuzzy被设置。翻译者则可以验证这个标识并且可能移除 fuzzy 标志。注意 fuzzy 消息不会对终端用户可用。另一种标志是c-格式，它指示该消息是一个printf-风格的格式模板。这意味着翻译也应当是一个格式字符串，其中有相同个数和相同类型的占位符。有工具能验证这点，它会切断该 c-格式标志。

55.1.3. 创建并维护消息目录

好，那么我们如何创建一个“空的”消息目录呢？首先，进入到包含你希望翻译其消息的程序的目录中。如果那里有一个文件nls.mk，那么这个程序就已经准备好被翻译了。

如果已经有某些.po文件，那么有些人已经完成了一些翻译工作。这些文件被命名为语言.po，其中语言是 ISO 639-1 双字符语言代码（小写形式）[1]，例如法语是fr.po。如果真的对于一种语言有需要多个翻译任务那么这些文件也可以被命名为语言_区域.po，其中region是 ISO 3166-1 双字符国家代码（大写形式）[2]，例如巴西的葡萄牙语是pt_BR.po。如果你找到了你想要的语言，你就可以开始在那个文件上工作了。

如果你需要开始一项新的翻译工作，那么首先运行命令：

```
make init-po
```

这将创建一个文件 程序名.pot（.pot用来把它与“在生产中的” PO 文件区分开。T表示“template”）。将这个文件拷贝到语言.po并编辑它。要让让人知道新语言可用，还要编辑文件nls.mk并且增加该语言（或语言与国家）代码到如下的行中：

```
AVAIL_LANGUAGES := de fr
```

（当然可以出现其他的语言）。

如果底层程序或库改变，消息可能会被程序员改变或增加。在这种情况中，你不需要从头开始。相反，运行命令：

```
make update-po
```

它将创建一个新的空白消息目录文件（你从其开始的 pot 文件）并且将把它合并到现有的 PO 文件中。如果合并算法不确定一个特定的消息，它会把该消息标记为如上所释的“fuzzy”。新的 PO 文件会被保存为一个.po.new扩展名。

55.1.4. 编辑 PO 文件

PO 文件可以用一个常规文本编辑器编辑。翻译者应该只改变 msgstr 指令之后引号之间的区域、增加注释并且修改 fuzzy 标志。Emacs 有一个 PO 模式（不出意料），我们觉得它非常有用。

PO 文件不需要被完全填充。如果无翻译（或一个空翻译）可用，该软件将自动回退到原始的字符串。提交一个未完成的翻译包含到源码树中不存在问题，那样可以让其他人有机会继

[1] http://www.loc.gov/standards/iso639-2/php/English_list.php
[2] https://www.iso.org/iso-3166-country-codes.html

续你的工作。不过，我们鼓励你在做完合并之后优先移除模糊项。记住模糊项将不会被安装，它们只作为什么可能是正确翻译的参考。

这里有一些在编辑翻译时要记住的事情：

- 确定如果原始消息以一个新行结束，那么翻译后的消息也要保持一样。制表符等也类似。
- 如果原始消息是一个printf格式字符串，翻译消息也需要是那样。翻译也需要保留相同顺序的格式指示符。有时语言的自然规则让这样做不可能或者至少很尴尬。在那样的情况下，你可以这样修改格式指示符：

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  那么第一个占位符将实际上使用列表中的第二个参数。digits$需要在任何其他格式操纵符之前紧跟 %（这个特性真实地存在于printf函数族中。你可能之前没有听说过它，因为除消息国际化之外很少会用到它）。
- 如果原始字符串包含一个语法错误，报告它（或者在程序源码中自己修复它）并且正常进行翻译。更正过的字符串可以在程序源码已被更新时合并。如果原始字符串包含一个事实错误，报告它（或自己修复它）并且不要翻译它。相反，你可以在 PO 文件中对该字符串标注一个注释。
- 保持原始字符串的风格和语气。特别地，不是句子的消息（cannot open file %s）可能不以一个大写字母开始（如果你的语言区分字母大小写）或者一个句号结尾（如果你的语言使用标点符号）。阅读第 54.3 节能有所帮助。
- 如果你不知道一个消息的含义，或者如果它是模棱两可的，在开发者邮件列表中询问其正确含义。如果说英语的终端用户可能也无法理解它或者发现它是模棱两可的，那么最好改进该消息。

55.2. 给编程者

55.2.1. 技术

这一节描述如何在PostgreSQL发布中的一个程序或库中实现本地语言支持。当前，这些知识只适用于 C 程序。

为一个程序增加 NLS 支持

1. 将这里的代码插入到该程序的启动序列中：

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("progname", LOCALEDIR);
   textdomain("progname");
   #endif
   ```

   （程序名实际上可以自由选择）。

2. 不管在哪里找到一个可被翻译的消息，需要插入一个gettext()调用，例如：

```
fprintf(stderr, "panic level %d\n", lvl);
```

将被改成：

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

（如果 NLS 支持没有被配置，gettext被定义为一个空操作）。

这容易增加很多混乱。一种常用的捷径是：

```
#define _(x) gettext(x)
```

如果该程序通过一个或几个函数（例如后端中的ereport()）完成他的大部分通信，则有另一种可行的解决方案。那么你可以在所有输入字符串上都内部调用这个函数gettext。

3. 在程序源码的目录中增加一个文件nls.mk。这个文件将被读作一个 makefile。其中要创建下列变量赋值：

   CATALOG_NAME

   程序名，如textdomain()调用中所提供的。

   AVAIL_LANGUAGES

   提供的翻译列表 — 初始为空。

   GETTEXT_FILES

   包含可翻译字符串的文件列表，即那些被用gettext或另一种替代方案标记的文件。最终，这将包括该程序近乎所有的源文件。如果这个列表太长你可以让第一个“file”变成一个+并且第二个词变成一个包含那些文件名的文件，第二个词指向的文件中每个文件名一行。

   GETTEXT_TRIGGERS

   为翻译者产生在其上工作的消息目录的工具需要知道，哪些函数调用包含可翻译的字符串。默认情况下，只有gettext()调用是已知的。如果你使用_或其他标识符，你需要在这里列出它们。如果可翻译的字符串不是第一个参数，该条目需要是形式func:2（用于第二个参数）。如果你有一个支持复数消息的函数，该条目应该看起来像func:1,2（标识单数和复数消息参数）。

构建系统将自动处理消息目录的编译和安装。

55.2.2. 消息书写指南

下面是一些书写已于翻译的消息的指南。

- 不要在运行时构建句子，如：

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  该句子中的词序可能在其他语言中完全不同。同样，即使你记得在每一个片段上调用gettext()，片段也可能不会被独立翻译得很好。更好的方式是复制一点代码，这样每个消息将被以一个整体被翻译。只有数字、文件名和这样的运行时变量才应该被在运行时插入到一个消息文本中。

- 由于类似的原因，下面的例子也不会工作：

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  因为它假定了复数形式。如果你发现了这个问题，你可以这样解决它：

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

  然后会失望。某些语言有多于两种形式，使用某些古怪的规则。通常最好设计消息来避免一次性避免该问题，例如像这样：

  ```
  printf("number of copied files: %d", n);
  ```

  如果你真的想要构建一个正确的复数消息，有对此的支持，但是有点笨拙。当在ereport()中产生一个主要或细节错误消息时，你可以这样写一些东西：

  ```
  errmsg_plural("copied %d file",
                "copied %d files",
                n,
                n)
  ```

  第一个参数是适用于英语单数形式的格式字符串，第二个参数是适用于英语复数形式的格式字符串，并且第三个参数是控制使用哪种复数形式的整数。后续参数针对每个格式字符串按照常规被格式化（通常，复数控制值也将是要被格式化的值之一，因此它必须被写两次）。在英语中只有n是否唯一有关系，但是在其他语言中可以有多种不同的复数形式。翻译者将两种英语形式看成一组并且有机会提供多个替补字符串，基于n的运行时值会选择其中合适的那一个。

  如果你需要复数化一个不直接进入到errmsg或errdetail报告的消息，你必须使用底层函数ngettext。见 gettext 文档。

- 如果你想要与翻译者沟通（例如关于一个消息如何在其他输出上对齐），在该字符串出现的地方之前放上一个以translator开始的注释，例如：

  ```
  /* translator: This message is not what it seems to be. */
  ```

  这些注释被复制到消息目录文件中，这样翻译者可以看到它们。

第 56 章 编写一个过程语言处理器

所有对用不是当前“版本 1”接口（用于编译型语言）语言编写的函数（这包括用户定义的过程语言中的函数、SQL 编写的函数）的调用都会流经一个用于指定语言的调用处理器。调用处理器负责以一种有意义的方式执行该函数，例如通过解释所提供的源文本。本章勾勒了如何编写一个新的过程语言调用处理器的轮廓。

一个过程语言的调用处理器是一个“正常的”函数，它必须以一种编译型语言（如 C）编写、使用版本-1接口并且在PostgreSQL中注册为无参数且返回类型language_handler。这种特殊的伪类型标识该函数是一个调用处理器并且阻止它在 SQL 命令中被直接调用。关于 C 语言调用惯例和动态载入的更多细节，请见第 38.10 节

调用处理器的调用方式和其他任何函数相同：它接收一个包含参数值和有关被调用函数信息的FunctionCallInfoData 结构，并且它被期望返回一个Datum结果（并且如果它希望返回一个 SQL 空值结果，它可能设置FunctionCallInfoData结构的isnull域）。一个调用处理器和一个普通被调用函数之间的区别是FunctionCallInfoData结构的flinfo->fn_oid域将包含要被调用的实际函数的 OID，而不是调用处理器本身。调用处理器必须使用这个域来决定要执行哪个函数。同样，被传递的参数列表已经被根据目标函数而不是调用处理器的声明被设置好。

调用处理器负责从pg_proc系统目录中取得该函数的项并且分析被调用函数的参数和返回类型。该函数的CREATE FUNCTION命令的AS子句可以在pg_proc行的prosrc列中被找到。通常这是过程语言中的源文本，但是在理论上它可以是其他某种东西，例如一个文件的路径名或其他任何详细告诉调用处理器做什么的东西。

同一个函数在每个 SQL 命令中常常被调用多次。一个调用处理器可以通过使用flinfo->fn_extra域来避免重复查找关于被调用函数的信息。这个域最初将为NULL，但是可以被调用处理器设置为指向关于被调用函数的信息。在后续调用中，如果flinfo->fn_extra已经为非-NULL，则它可以被使用并且信息查找步骤将被跳过。调用处理器必须确保flinfo->fn_extra被指向直到当前查询的末尾都存活的内存，因为一个FmgrInfo数据接口可以被保持那么久。一种方式是在flinfo->fn_mcxt指定的内存上下文中分配额外的数据；这样的数据通常必须与FmgrInfo本身具有相同的生命期。但是处理器也可以选择使用一个生存时间更长的内存上下文，这样它能够在查询之间缓存函数定义信息。

当一个过程语言函数被作为一个触发器调用时，不会有参数通过常用方式被传递，但是FunctionCallInfoData的context域指向一个TriggerData结构而不是为NULL（就像它在一个普通函数调用中那样）。一个语言处理器应该为过程语言函数提供机制来得到触发器信息。

这是一个用 C 编写的过程语言处理器的模板：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum       retval;
```

```
    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Called as a trigger function
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Called as a function
         */

        retval = ...
    }

    return retval;
}
```

要完成该调用处理器，只需要加入几千行代码来替代点号即可。

在将处理器函数编译成一个可载入模块（第 38.10.5 节后，接着用下列命令注册例子过程语言：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

尽管提供一个调用处理器对于创建一个最小过程语言已经足够，还可以提供其他两个可选的函数来让该函数更易用。它们是验证器和内联处理器。一个验证器可以被提供来允许在CREATE FUNCTION期间完成语言相关的检查。一个内联处理器可以被提供来允许语言支持通过DO命令执行匿名代码块。

如果一个验证器被一个过程语言提供，它必须被声明为一个采用一个单一oid类型参数的函数。该验证器的结果被忽略，因为它通常被声明为返回void。验证器将在一个已经创建了或更新了一个以该过程语言编写的函数的CREATE FUNCTION命令之后被调用。被传入的 OID 是函数的pg_proc行的 OID。验证器必须用通常方式取得这个行，并且做任何合适的检查。首先，调用CheckFunctionValidatorAccess()来诊断对用户通过CREATE FUNCTION无法达到的验证器的显式调用。典型的检查包括验证函数的参数和结果类型是否被该语言支持，以及该函数体在该语言中语法是否正确。如果验证器发现该函数是好的，它应该只是返回。如果它发现一个错误，它应该通过通常的ereport()错误报告机制报告该错误。抛出一个错误将强制一次事务回滚并且因此阻止不正确的函数定义被提交。

验证器函数通常应该尊重check_function_bodies参数：如果它被关闭那么任何代价大的或上下文敏感的检查应该被跳过。如果该语言提供了编译时代码执行，验证器必须抑制可能引起这种执行的检查。特别地，这个参数会被pg_dump关闭，这样它能载入过程语言函数而不用担心副作用或那些函数体对其他数据库对象的依赖（因为这种要求，调用处理器应该避免假设验证器已经完整地检查过该函数。拥有一个验证器的要点不是让调用处理器忽略检查，而是如果在一个CREATE FUNCTION命令中发现明显错误时立即提示用户）。然而究竟检查什么的选择大部分都留给了验证器函数，注意当check_function_bodies为打开时，核心CREATE FUNCTION代码只执行附加到一个函数的SET子句。 因此，为了避免在重新载入一个转储时的伪失败，当check_function_bodies为关闭时，结果可能会被 GUC 参数影响的检查绝对应当被跳过。

如果一个过程语言提供了一个内联处理器，它必须被声明为一个采用一个单一internal类型参数的函数。内联处理器的结果会被忽略，因此它通常被声明为返回void。当一个DO语句被调用执行指定过程语言时，内联处理器将被调用。实际被传递的参数是一个指向一个InlineCodeBlock结构的指针，它包含有关DO语句参数的而信息，特别是将被执行的匿名代码块的文本。内联处理器应该执行该代码并返回。

我们推荐你包装所有这些函数声明，以及CREATE LANGUAGE命令本身到一个extension中，这样一个简单的CREATE EXTENSION命令就足以安装该语言。关于编写扩展的信息请见第 38.16 节

在尝试编写你自己的语言处理器时，包括在标准发布中的过程语言是很好的参考。看看源码树中的src/pl子目录。CREATE LANGUAGE参考页也有一些有用的细节。

第 57 章 编写一个外部数据包装器

所有在一个外部表上的操作都通过它的外部数据包装器来处理，外部数据包装器由一组被核心服务器调用的函数组成。外部数据包装器负责从远程数据源取得数据并把它返回给PostgreSQL执行器。如果要支持更新外部表，包装器也需要处理更新。本章将介绍如何编写一个新的外部数据包装器。

在你试图编写你自己的外部数据包装器时，包含在标准发布中的外部数据包装器会是很好的参考。请看看源代码树的contrib子目录。CREATE FOREIGN DATA WRAPPER参考页也会有很多有用的细节。

> **注意**
>
> SQL 标准声明了一个接口用来编写外部数据包装器。但是，PostgreSQL 没有实现该 API，因为将其纳入到 PostgreSQL 中的工作量将会很大，并且标准的 API 并没有得到广泛地采用。

57.1. 外部数据包装器函数

FDW的作者需要实现一个处理器函数，并且可以有选择地实现一个验证器函数。两个函数都必须被用一种编译语言（如 C）来编写，并使用版本-1接口。关于 C 语言调用规范和动态载入的细节，请见第 38.10 节

处理器函数简单地返回一个函数指针结构给回调函数，回调函数将被规划器、执行器和多种维护命令调用。编写一个FDW的大部分工作量都在实现这些回调函数上。处理器函数必须被注册在PostgreSQL中，并且注册为不需要参数并且返回特殊的伪类型fdw_handler。回调函数则是普通的 C 函数并且对于 SQL 层是不可见的或者不可调用的。回调函数在第 57.2 节描述。

验证器函数负责验证CREATE和ALTER命令中对它的外部数据包装器给出的选项，以及使用该包装器的外部服务器、用户映射和外部表。验证器函数必须被注册为要求两个参数：一个包含需要被验证的选项的文本数组，以及一个表示与这些选项相关联的对象类型的OID（以该对象可能被存储的系统目录的OID的形式，可以是ForeignDataWrapperRelationId、ForeignServerRelationId、UserMappingRelationId或ForeignTableRel
如果没有提供验证器函数，在对象创建或修改时选项不会被检查。

57.2. 外部数据包装器回调例程

FDW处理器函数返回一个palloc过的FdwRoutine结构，它包含下文描述的回调函数的指针。扫描相关的函数是必需的，剩下的是可选的。

FdwRoutine结构类型被声明在src/include/foreign/fdwapi.h中，可以查看它来获得额外的信息。

57.2.1. 用于扫描外部表的FDW例程

```
void
GetForeignRelSize(PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

获取一个外部表的关系尺寸估计。在对一个扫描外部表的查询进行规划的开头将调用该函数。root是规划器的关于该查询的全局信息；baserel是规划器的关于该表的信

息；foreigntableid是外部表在pg_class中的 OID （foreigntableid可以从规划器的数据结构中获得，但是为了减少工作量，这里直接显式地将它传递给函数）。

这个函数应该更新baserel->rows为表扫描根据限制条件完成了过滤后将返回的预期行数。baserel->rows的初始值只是一个常数的默认估计值，应该尽可能把它替换掉。如果该函数能够计算出一个平均结果行宽度的更好的估计值，该函数也可能选择更新baserel->width。

更多信息请见第 57.4 节

```
void
GetForeignPaths(PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

为一个外部表上的扫描创建可能的访问路径。这个函数在查询规划过程中被调用。参数和GetForeignRelSize相同，后者已经被调用过了。

这个函数必须为外部表上的扫描生成至少一个访问路径（ForeignPath节点），并且必须调用add_path把每一个这样的路径加入到baserel->pathlist中。我们推荐使用create_foreignscan_path来建立ForeignPath节点。该函数可以生成多个访问路径，例如一个具有合法pathkeys的路径表示一个预排序好的结果。每一个反问路径必须包含代价估计，并且能包含任何FDW的私有信息，这种信息被用来标识想要使用的指定扫描方法。

更多信息请见第 57.4 节

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid,
                ForeignPath *best_path,
                List *tlist,
                List *scan_clauses,
                Plan *outer_plan);
```

从选择的外部访问路径创建一个ForeignScan计划节点。这个函数在查询规划的末尾被调用。参数和GetForeignRelSize的一样，外加选中的ForeignPath（在前面由GetForeignPaths、GetForeignJoinPaths或者GetForeignUpperPaths产生）、被计划节点发出的目标列表以及计划节点强制的限制子句以及被RecheckForeignScan执行的复查所使用的ForeignScan的外子计划（如果该路径是用于一个连接而非基本关系，则foreigntableid是InvalidOid）。

这个函数必须创建并返回一个ForeignScan计划节点，我们对剑使用make_foreignscan来建立ForeignScan节点。

更多信息见第 57.4 节

```
void
BeginForeignScan(ForeignScanState *node,
                 int eflags);
```

开始执行一个外部扫描。这个函数在执行器启动阶段被调用。它应该执行任何在扫描能够开始之前需要完成的初始化工作，但是并不开始执行真正的扫描（会在第一次调用IterateForeignScan时完成）。ForeignScanState节点已经被创建好了，但是它的fdw_state域仍然为 NULL。关于要被扫描的表的信息可以通过ForeignScanState节点访问（特殊地，从底层的ForeignScan计划节点，它包含任何由GetForeignPlan提供的FDW私有信息）。eflags包含描述执行器对该计划节点操作模式的标志位。

注意当(eflags & EXEC_FLAG_EXPLAIN_ONLY)为真时，这个函数不应该执行任何外部可见的动作；它应当只做最少的事情来创建对ExplainForeignScan和EndForeignScan有效的节点状态。

```
TupleTableSlot *
IterateForeignScan(ForeignScanState *node);
```

从外部源获得一行，将它放在一个元组表槽中返回（节点的ScanTupleSlot应当被用于此目的）。如果没有更多的行可用则返回 NULL。元组表槽设施允许一个物理的或者虚拟的元组被返回；在大部分情况下出于性能的考虑会倾向于选择后者。注意这是在一个短期存在的内存上下文中被调用的，该内存上下文会在调用之间被重置。如果你需要长期存在的存储，请在BeginForeignScan中创建内存上下文，或者使用节点的EState中的es_query_cxt。

如果提供了fdw_scan_tlist目标列表，被返回的行必须匹配它，如果没有提供则它们必须匹配被扫描的外部表的行类型。如果选择优化掉不需要的列，你应该在那些列的位置上插入控制或者生成一个忽略了那些列的fdw_scan_tlist列表。

注意PostgreSQL的执行器并不在乎被返回的行是否违背了定义在该外部表上的任何约束 — 但是规划器会在乎这一点，并且如果在外部表中有可见行不满足一个约束，规划器可能会错误地优化查询。如果当用户已经声明一个约束应该为真时它却被违背，最合适的处理可能是产生一个错误（就像在数据类型失配的情况下所作的那样）。

```
void
ReScanForeignScan(ForeignScanState *node);
```

从头开始重启一个扫描。注意扫描所依赖的任何参数可能已经改变了值，因此新扫描不一定会返回完全相同的行。

```
void
EndForeignScan(ForeignScanState *node);
```

结束扫描并释放资源。通常释放palloc过的内存并不重要，但是打开的文件和到远程服务器的连接等应该被清理。

57.2.2. 用于扫描外部连接的 FDW 例程

如果一个 FDW 支持远程执行外部连接（而不是先把两个表的数据取到本地然后做本地连接），它应该提供这个回调函数：

```
void
GetForeignJoinPaths(PlannerInfo *root,
                    RelOptInfo *joinrel,
                    RelOptInfo *outerrel,
                    RelOptInfo *innerrel,
                    JoinType jointype,
                    JoinPathExtraData *extra);
```

它为两个（或更多）同属于一台外部服务器的外部表的连接创建可能的访问路径。这个可选的函数会在查询规划过程中被调用。和GetForeignPaths一样，这个函数应该为提供的joinrel生成ForeignPath路径，并且调用add_path把这些路径加入到该连接应该考虑的路径集合中。但是和GetForeignPaths不一样的是，不需要这个函数产生最少一个路径，因为涉及本地连接的路径总是可用的。

注意为相同的连接关系将会重复地调用这个函数用来生成内外关系的不同组合。FDW 需要负责最小化其中重复的工作。

如果一个ForeignPath路径被选中用于该连接，它将在整个连接处理中存在，为其中的成分表和子连接产生的路径将不会被使用。后续对该连接路径的处理大部分和扫描单个外部表的路径一样。一点不同是ForeignScan计划节点的scanrelid应该被设置为零，因为它表示的不是单个关系，而是用ForeignScan节点的fs_relids域来表示被连接的关系集合（后一个域会被核心规划器代码自动设置，不需要由 FDW 填充）。另一点不同是，由于一个远程连接的列列表无法在系统目录中找到，FDW 必须用一个合适的TargetEntry节点列表来填充fdw_scan_tlist，表示运行时它返回的元组中提供的列的集合。

更多信息请见第 57.4 节

57.2.3. 用于规划扫描/连接后处理的 FDW 例程

如果一个 FDW 支持执行远程的扫描/连接后处理，例如远程聚集，那么它应该提供这个回调函数：

```
void
GetForeignUpperPaths(PlannerInfo *root,
                     UpperRelationKind stage,
                     RelOptInfo *input_rel,
                     RelOptInfo *output_rel,
                     void *extra);
```

为上层关系处理创建可能的访问路径，这是规划器针对所有扫描/连接后查询处理的术语，例如聚集、窗口函数、排序和表更新。在查询规划期间会调用这个可选的函数。当前，只有当该查询中涉及的所有基本关系都属于同一个 FDW 时才会调用这个函数。这个函数应该为FDW 知道如何远程执行的任何扫描/连接后处理生成ForeignPath路径，并且调用add_path把这些路径加入到上层关系中。就GetForeignJoinPaths来说，并不要求这个函数在创建任何路径时都能成功，因为路径总是有可能涉及到本地处理。

stage参数表示当前正在考虑的是哪一个扫描/连接后处理步骤。output_rel是接收表示这一个步骤的路径的上层关系，而input_rel是表示这个步骤输入的关系。extra参数提供额外的细节，当前只会为UPPERREL_PARTIAL_GROUP_AGG或者UPPERREL_GROUP_AGG设置它，这种情况下它会指向一个GroupPathExtraData结构（注意被加入到output_rel中的ForeignPath路径通常对input_rel的路径没有直接的依赖，因为它们的处理被认为是在外部处理的。不过，检查为前一个处理步骤生成的路径有助于避免冗余的规划工作）。

更多信息请见第 57.4 节

57.2.4. 更新外部表的FDW例程

如果一个FDW支持可写的外部表，根据FDW的需要和功能它应该提供某些或者全部下列回调函数：

```
void
AddForeignUpdateTargets(Query *parsetree,
                        RangeTblEntry *target_rte,
                        Relation target_relation);
```

UPDATE和DELETE操作是在之前由表扫描函数取出的行上被执行的。FDW可能需要额外的信息（例如一个行ID或主键列的值）来保证它能够找到要更新或删除的准确行。要支持这些要求，这个函数可以项列列表中增加额外的隐藏或“junk”的目标列，它们在一个UPDATE或DELETE期间会被从外部表中获取。

要做到这一点，向parsetree->targetList中增加TargetEntry项，它们包含要被获取的额外值的表达式。每一个这样的项必须被标记为resjunk = true，并且必须有一个可区分的resname用于在执行期间标识它。请避免使用匹配ctidN、wholerow或wholerowN的名字，

因为核心系统可能会生成使用这些名字的junk列。如果额外的表达式比简单的Var更加复杂，在把它们加入到目标列表之前必须把它们用eval_const_expressions进行处理。

尽管这个函数在规划过程中被调用，但所提供的信息与其他规划例程可用的信息有点区别。parsetree是UPDATE或DELETE命令的分析树，而target_rte和target_relation描述目标外部表。

如果AddForeignUpdateTargets指针被设置为NULL，则不会有额外的目标表达式被加入（这将使得我们不可能实现DELETE操作，而UPDATE则还有可能是可行的，前提是FDW依赖一个未改变的主键来标识行）。

```
List *
PlanForeignModify(PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

执行外部表上插入、更新或删除所需的任何附加规划动作。这个函数生成FDW私有信息，该信息将被附加到执行该更新动作的ModifyTable计划节点。这个私有信息的形式必须是一个List，并将会在执行阶段被传递给BeginForeignModify。

root是规划器关于该查询的全局信息。plan是ModifyTable计划节点，它除了fdwPrivLists域之外是完整的。resultRelation通过目标外部表的范围表索引来标识它。subplan_index标识这是ModifyTable计划节点的哪个目标，从零开始计数；如果你希望索引到plan->plans或其他plan节点的子结构中，请使用它。

更多信息见第 57.4 节

如果PlanForeignModify指针被设置为NULL，则不会有额外的计划时动作被执行，并且传递给BeginForeignModify的fdw_private列表也将为 NIL。

```
void
BeginForeignModify(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo,
                   List *fdw_private,
                   int subplan_index,
                   int eflags);
```

开始执行一个外部表修改操作。这个例程在执行器启动期间被调用。它应该执行任何先于实际表修改的初始化工作。随
后，ExecForeignInsert、ExecForeignUpdate或ExecForeignDelete将被为每一个将被插入、更新或删除的元组调用。

mtstate是要被执行的ModifyTable计划节点的状态信息；通过这个结构可以得到关于规划和执行阶段的全局数据。rinfo是描述目标外部表的ResultRelInfo结构
（ResultRelInfo的ri_FdwState域用于FDW来存储它在此操作中需要的任何私有状态）。fdw_private包含PlanForeignModify生成的私有数据。subplan_index标识这是ModifyTable计划节点的哪个目标。eflags包含描述执行器对该计划节点操作模式的标志位。

注意当(eflags & EXEC_FLAG_EXPLAIN_ONLY)为真，这个函数不应执行任何外部可见的动作；它只应该做最少的工作来创建ExplainForeignModify和EndForeignModify可用的节点状态。

如果BeginForeignModify指针被设置为NULL，在执行器启动期间将不会采取任何动作。

```
TupleTableSlot *
ExecForeignInsert(EState *estate,
                  ResultRelInfo *rinfo,
```

```
                TupleTableSlot *slot,
                TupleTableSlot *planSlot);
```

插入一个元组到外部表。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot包含要被插入的元组；它将匹配外部表的行类型定义。planSlot包含由ModifyTable计划节点的子计划生成的元组；它与slot不同，它可能包含额外的“junk”列（INSERT情况通常不关心planSlot，但是为了完整性还是在这里提供它）。

返回值可以是一个包含实际被插入的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有插入行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在INSERT查询具有一个RETURNING子句或者外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是 FDW 应该选择优化成根据RETURNING子句的内容返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行计数将会是错误的。

如果ExecForeignInsert指针被设置为NULL，尝试向外部表插入将会失败并报告一个错误消息。

```
TupleTableSlot *
ExecForeignUpdate(EState *estate,
                ResultRelInfo *rinfo,
                TupleTableSlot *slot,
                TupleTableSlot *planSlot);
```

更新外部表中的一个元组。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot包含元组的新数据；它将匹配外部表的行类型定义。planSlot包含由ModifyTable计划节点的子计划生成的元组；它与slot不同，它可能包含额外的“junk”列（INSERT情况通常不关心planSlot，但是为了完整性还是在这里提供它）。特殊地，任何AddForeignUpdateTargets所要求的junk列在这个槽中都是有效的。

返回值可以是一个包含实际被更新的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有更新行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在UPDATE查询具有一个RETURNING子句或者外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是 FDW 应该选择优化成根据RETURNING子句的内容返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行计数将会是错误的。

如果ExecForeignUpdate指针被设置为NULL，尝试更新外部表将会失败并报告一个错误消息。

```
TupleTableSlot *
ExecForeignDelete(EState *estate,
                ResultRelInfo *rinfo,
                TupleTableSlot *slot,
                TupleTableSlot *planSlot);
```

从外部表删除一个元组。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot在调用时不包含任何有用的东西，但是可以被用于保持被返回的元组。planSlot包含由ModifyTable计划节点的子计划生成的元组；特殊地，它将携带AddForeignUpdateTargets所要求的任意垃圾列。垃圾列被用来标识要被删除的元组。

返回值可以是一个包含实际被删除的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有删除行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在DELETE查询具有一个RETURNING子句或者外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是 FDW 应该选择优化成根据RETURNING子句的内容返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行计数将会是错误的。

如果ExecForeignDelete指针被设置为NULL，尝试从外部表中删除将会失败并报告一个错误消息。

```
void
EndForeignModify(EState *estate,
                 ResultRelInfo *rinfo);
```

结束表更新并释放资源。通常释放palloc的内存并不重要，但是打开的文件和到远程服务器的连接等应当被清除。

如果EndForeignModify指针被设置为NULL，在执行器关闭期间不会采取任何动作。

被INSERT或者COPY FROM插入到分区表中的元组会被路由到分区。如果一个FDW支持可路由的外部表分区，它还应该提供下面的回调函数。当在外部表上执行COPY FROM时，也会调用这些函数。

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

开始在外部表上执行插入操作。当外部表被选中作为元组路由的分区以及COPY FROM命令中指定的目标时，在第一个元组被插入到该外部表之前会调用这个例程。它应该执行实际插入之前所需的任何初始化工作。随后，为每一个被插入到该外部表的元组都将调用ExecForeignInsert。

mtstate是正在被执行的ModifyTable计划节点的总体状态，通过这个结构可以得到有关计划和执行的全局数据。rinfo是描述目标外部表的ResultRelInfo结构（对于FDW，ResultRelInfo的ri_FdwState字段用来存放这个操作所需要的私有状态）。

当这个例程被一个COPY FROM命令调用时，不会提供mtstate中与计划相关的全局数据，并且后续为每个插入元组调用的ExecForeignInsert的planSlot参数为NULL，不管该外部表是为元组路由选中的分区还是命令中指定的目标。

如果BeginForeignInsert指针被设置为NULL，则不会采取初始化动作。

```
void
EndForeignInsert(EState *estate,
                 ResultRelInfo *rinfo);
```

结束插入操作并且释放资源。通常释放palloc的内存并不重要，但是打开的文件和与远程服务器的连接应该被清除。

如果EndForeignInsert指针被设置为NULL，则不会采取终止动作。

```
int
IsForeignRelUpdatable(Relation rel);
```

报告指定的外部表支持哪些更新操作。返回值应该是一个规则事件编号的位掩码，它指示了哪些操作被外部表支持，它使用CmdType枚举，即： (1 << CMD_UPDATE) = 4表示UPDATE、(1 << CMD_INSERT) = 8表示INSERT以及 (1 << CMD_DELETE) = 16表示DELETE。

如果IsForeignRelUpdatable指针被设置为NULL，而FDW提供了ExecForeignInsert、ExecForeignUpdate或ExecForeignDelete，则外部表分别被假定为

可插入、可更新或可删除。只有在FDW支持某些表是可更新的而某些不是可更新的时候，才需要这个函数（即便如此，也允许在执行例程中抛出一个错误而不是在这个函数中检查。但是，这个函数被用来决定显示在information_schema视图中的可更新性）。

一些对于外部表的插入、更新和删除可以通过实现另一组接口来优化。普通的插入、更新和删除接口会从远程服务器取得行，然后一次修改其中一行。在某些情况下，这种逐行的方式是必要的，但是可能效率不高。如果有可能让外部服务器判断哪些行可以直接修改而无需先检索它们并且没有本地触发器会影响该操作，那么可以让整个操作在远程服务器上执行。下面介绍的接口能让这种做法变成可能。

```
bool
PlanDirectModify(PlannerInfo *root,
                 ModifyTable *plan,
                 Index resultRelation,
                 int subplan_index);
```

决定在远程服务器上执行直接修改是否安全。如果安全，执行所需的规划动作然后返回true。否则返回false。这个可选的函数在查询规划期间被调用。如果这个函数成功，在执行阶段将会调用BeginDirectModify、IterateDirectModify和EndDirectModify。否则，对表的修改将采用上文描述的表更新函数来执行。参数和PlanForeignModify的相同。

要在远程服务器上执行直接修改，这个函数必须用一个ForeignScan计划节点（它在远程服务器上执行直接修改）重写目标子计划。ForeignScan的operation域必须被合适地设置为CmdType枚举值，即CMD_UPDATE表示UPDATE、CMD_INSERT表示INSERT而CMD_DELETE表示DELETE。

更多信息请见第 57.4 节

如果PlanDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```
void
BeginDirectModify(ForeignScanState *node,
                  int eflags);
```

准备在远程服务器上执行一次直接修改。这个函数会在执行器启动时被调用。它应该执行直接修改所需的任何初始化工作（应该在第一次IterateDirectModify调用之前完成）。ForeignScanState节点已经被创建，但是它的fdw_state域仍然为 NULL。有关要被修改的表的信息可以通过ForeignScanState节点（具体地，从底层的ForeignScan计划节点，它包含了PlanDirectModify提供的 FDW-私有信息）访问。eflags包含描述执行器对于这个计划节点操作模式的标志位。

注意当(eflags & EXEC_FLAG_EXPLAIN_ONLY)为真时，这个函数不应该执行任何外部可见的动作。它应当只做最少的工作让该节点状态对ExplainDirectModify和EndDirectModify有效。

如果BeginDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```
TupleTableSlot *
IterateDirectModify(ForeignScanState *node);
```

当INSERT、UPDATE或者DELETE查询没有RETURNING子句时，完成远程服务器上的直接修改后返回 NULL。当查询有该子句时，取出一个包含RETURNING计算所需数据的结果，用一个元组表槽返回它（节点的ScanTupleSlot应被用于这一目的）。实际被插入、更新或者删除的数据必须被存储在该节点的EState的es_result_relation_info->ri_projectReturning->pi_exprContext->ecxt_scantuple中。如果没有更多行可用，则返回 NULL。注意这个函数会在一个短期生存的内存上下文中被调用，该上下文会在两次调用之间被重置。如果需要一个长期存在的存储，可以在BeginDirectModify中创建一个内存上下文，或者使用该节点的EState中的es_query_cxt。

如果提供了fdw_scan_tlist目标列表，则被返回的行必须匹配它。否则，被返回的行必须匹配被更新的外部表的行类型。如果选择优化掉RETURNING计算不需要的列，应该在这些列的位置上插入空值，或者生成一个忽略这些列的fdw_scan_tlist列表。

不管该查询是否具有RETURNING子句，查询所报告的行计数必须由 FDW 本身增加。当查询没有该子句时，FDW 还必须为EXPLAIN ANALYZE情况下的ForeignScanState节点增加行计数。

如果IterateDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```
void
EndDirectModify(ForeignScanState *node);
```

在远程服务器上的直接修改后进行清理。通常释放用 palloc 分配的内存并不重要，但是诸如打开的文件和到远程服务器的连接应该被清除。

如果EndDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

57.2.5. 用于行锁定的 FDW 例程

如果一个 FDW 希望支持后期行锁定（如第 57.5 节所述），它必须提供下列回调函数：

```
RowMarkType
GetForeignRowMarkType(RangeTblEntry *rte,
                      LockClauseStrength strength);
```

报告要对一个外部表使用哪个行标记选项。rte是该表的RangeTblEntry节点，而strength描述FOR UPDATE/SHARE子句（如果有）所要求的锁长度。结果必须是RowMarkType枚举类型的一个成员。

这个函数在查询规划期间会为每一个出现在UPDATE、DELETE或者SELECT FOR UPDATE/SHARE查询中的外部表调用，并且该外部表不是UPDATE和DELETE的目标。

如果GetForeignRowMarkType指针被设置为NULL，将总是使用ROW_MARK_COPY选项（这意味着将不会调用RefetchForeignRow，因此也不必提供它）。

更多信息请见第 57.5 节

```
HeapTuple
RefetchForeignRow(EState *estate,
                  ExecRowMark *erm,
                  Datum rowid,
                  bool *updated);
```

从外部表中重新取得一个元组，如有必要先锁定它。estate是该查询的全局执行状态。erm是描述目标外部表以及要获取的行锁类型（如果有）的ExecRowMark结构。rowid标识要取得的元组。updated是一个输出参数。

这个函数应该返回被取得的元组的一个已经分配内存的拷贝，如果无法得到行锁则返回NULL。要获得的行锁由erm->markType定义，它是之前由GetForeignRowMarkType返回的值（ROW_MARK_REFERENCE标识只重新取得元组但不获得任何锁，这个例程将不会看到ROW_MARK_COPY）。

此外，如果取得的是一个更新过的版本而不是之前获得的同一版本，*updated应被设置为true（如果 FDW 无法确定这一点，推荐总是返回true）。

注意在默认情况下，获取行锁失败应该导致产生错误。如果erm->waitPolicy指定了SKIP LOCKED，只有返回NULL才是合适的。

rowid是要被重新取得的行之前读到的ctid值。尽管rowid值被作为Datum传递，但是目前它只能被读作tid。选择该函数 API 是希望未来能允许其他的行 ID 数据类型。

如果RefetchForeignRow指针被设置为NULL，重新取得行的尝试将会失败并伴随有一个错误消息。

更多信息请见第 57.5 节

```
bool
RecheckForeignScan(ForeignScanState *node, TupleTableSlot *slot);
```

重新检查之前返回的元组是否仍然匹配相关的扫描和连接条件，并且可能提供该元组的一个修改版本。对于不执行连接下推的外部数据包装器，通常把这设置为NULL并且恰当地设置fdw_recheck_quals会更方便。不过当外部连接被下推时，把与所有基表相关的检查重新应用在结果元组上是不够的，即便所有需要的属性都存在也是如此，因为匹配某个条件失败可能会导致某些属性变成 NULL，而不是没有元组被返回。RecheckForeignScan能够重新检查条件，并且在它们仍然满足时返回真，否则返回假，但是它也能够在提供的槽中存储一个替换元组。

要实现连接下推，外部数据包装器通常将构造一个可替代的本地连接计划，它只被用来做重新检查。这将变成ForeignScan的外子计划。在需要一次重新检查时，这个子计划可以被执行并且结果元组可以被存储在槽中。这个计划不需要效率很高，因为不会有基表返回超过一行。例如，它可以把所有的连接实现为嵌套循环。函数GetExistingLocalJoinPath可以被用来在已有的路径中搜索合适的本地连接路径，它可以被用作替换的本地连接计划。GetExistingLocalJoinPath会在指定连接关系的路径列表中搜索一个非参数化路径（如果没有找到这样的路径，它会返回 NULL，这种情况下外部数据包装器可以自行构造本地路径或者可以选择不为这个连接创建访问路径）。

57.2.6. EXPLAIN的FDW例程

```
void
ExplainForeignScan(ForeignScanState *node,
                   ExplainState *es);
```

为一个外部表扫描打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来向EXPLAIN输出中增加域。es中的标志域可以被用来决定什么将被打印，并且ForeignScanState节点的状态可以被检查来为EXPLAIN ANALYZE提供运行时统计数据。

如果ExplainForeignScan指针被设置为NULL，在EXPLAIN期间不会打印任何额外的信息。

```
void
ExplainForeignModify(ModifyTableState *mtstate,
                     ResultRelInfo *rinfo,
                     List *fdw_private,
                     int subplan_index,
                     struct ExplainState *es);
```

为一个外部表更新打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来向EXPLAIN输出中增加域。es中的标志域可以被用来决定什么将被打印，并且ModifyTableState节点的状态可以被检查来为EXPLAIN ANALYZE提供运行时统计数据。前四个参数和BeginForeignModify相同。

如果ExplainForeignModify指针被设置为NULL，在EXPLAIN期间不会打印任何额外的信息。

```
void
```

```
ExplainDirectModify(ForeignScanState *node,
                    ExplainState *es);
```

为远程服务器上的直接修改打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来为EXPLAIN输出增加域。es中的标志域可以被用来判断要打印什么，并且在EXPLAIN ANALYZE情况中可以观察ForeignScanState节点的状态来提供运行时统计信息。

如果ExplainDirectModify指针被设置为NULL，EXPLAIN期间不会打印出额外的信息。

57.2.7. ANALYZE的FDW例程

```
bool
AnalyzeForeignTable(Relation relation,
                    AcquireSampleRowsFunc *func,
                    BlockNumber *totalpages);
```

当ANALYZE被执行在一个外部表上时会调用这个函数。如果FDW可以为这个外部表收集统计信息，它会返回true并提供一个函数指针，该函数将将从func中的表上收集采样行，外加totalpages中页面中的表尺寸估计值。否则，返回false。

如果FDW不支持为任何表收集统计信息，AnalyzeForeignTable指针可以被设置为NULL。

如果提供，采样收集函数必须具有签名

```
int
AcquireSampleRowsFunc(Relation relation, int elevel,
                      HeapTuple *rows, int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

应该从该表上收集最多targrows行的一个随机采样并将它存放到调用者提供的rows数组中。实际被收集的行的数量必须被返回。另外，将表中有效行和死亡行的总数存储到输出参数totalrows和totaldeadrows中（如果FDW没有死亡行的概念，将totaldeadrows设置为0）。

57.2.8. IMPORT FOREIGN SCHEMA的 FDW 例程

```
List *
ImportForeignSchema(ImportForeignSchemaStmt *stmt, Oid serverOid);
```

取得一个外部表创建命令的列表。在执行IMPORT FOREIGN SCHEMA时会调用这个函数，并且会把该语句的解析树以及要使用的外部服务器的 OID 传递给它。它应该返回一个 C 字符串的列表，每一个必须包含一个CREATE FOREIGN TABLE命令。这些命令将被核心服务器所解析和执行。

在ImportForeignSchemaStmt结构中，remote_schema是要从其中导入这些表的远程模式的名称。list_type标识如何过滤表名：FDW_IMPORT_SCHEMA_ALL表示该远程模式中的所有表都应该被导入（这种情况下table_list为空），FDW_IMPORT_SCHEMA_LIMIT_TO表示只包括table_list中列出的表，而FDW_IMPORT_SCHEMA_EXCEPT则表示排除table_list中列出的表。options是一个用于该导入处理的选项列表。选项的含义由 FDW 决定。例如，一个 FDW 可以用一个选项来定义是否应该导入列的NOT NULL属性。这些选项不需要与那些 FDW 支持的数据库对象选项有什么关系。

FDW 可能会忽略ImportForeignSchemaStmt的local_schema域，因为核心服务器会自动地向解析好的CREATE FOREIGN TABLE命令中插入本地模式的名称。

FDW 也不必担心实现list_type以及table_list所指定的过滤，因为核心服务器将自动根据那些选项跳过为被排除的表所返回的命令。不过，起初就避免为被排除的表创建命令当然更好。函数IsImportableForeignTable()可以用来测试一个给定的外部表名是否能通过该过滤器。

如果 FDW 不支持导入表定义，ImportForeignSchema指针可以被设置为NULL。

57.2.9. 并行执行的 FDW 例程

ForeignScan节点可以选择支持并行执行。一个并行的ForeignScan将在多个进程中被执行并且在相互合作的进程中每一个元组必须只被返回一次。要做到这样，进程可以通过动态共享内存的固定尺寸块来协作。并不保证在每一个进程中这部份共享内存都被映射到相同的地址，因此不能包含指针。下面的函数通常都是可选的，但是如果要支持并行执行就必须提供其中的大部分。

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

测试一个扫描是否可以在一个并行工作者中被执行。只有当规划器相信可以使用并行计划时才会调用这个函数，如果该扫描在并行工作者中可以安全运行这个函数应该返回真。如果远程数据源具有事务语义，情况通常都不是这样，除非工作者到数据的连接能够以某种方式共享与领导者相同的事务环境。

如果没有定义这个函数，则假定该扫描必须被放置在并行领导者中。注意返回真并不意味着该扫描本身可以被并行完成，只是说明该扫描可以在一个并行工作者中执行。因此，即便当不支持并行执行时，定义这个方法也是有用的。

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

估算并行操作所需的动态共享内存的数量。这可能比实际要用的数量更大，但是绝不能更小。返回值的单位是字节。这个函数是可选的，并且在不需要时可以省略。但是如果它被省略，接下来的三个函数也必须被省略，因为不会为FDW分配共享内存。

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                         void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate指向一块共享内存区域，其尺寸等于EstimateDSMForeignScan的返回值。这个函数是可选的，并且在不需要时可以省略。

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                           void *coordinate);
```

当外部扫描计划将要被重新扫描时，重新初始化并行操作所要求的动态共享内存。这个函数是可选的，并且在不需要时可以省略。推荐的措施是这个函数只重置共享状态，而ReScanForeignScan函数仅重置本地状态。当前，这个函数将在ReScanForeignScan之前被调用，但是最好不要依赖于这种顺序。

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

基于领导者在InitializeDSMForeignScan期间建立的共享状态初始化并行工作者的本地状态。这个函数是可选的，并且在不需要时可以省略。

```
void
ShutdownForeignScan(ForeignScanState *node);
```

在预见到节点将不会被执行完时释放资源。这个函数不会在所有的情况中执行，有时会在没有先调用这个函数之前调用EndForeignScan。由于在这个回调被调用之后并行查询使用的DSM段将被销毁，希望在DMS段消失前采取某种行动的外部数据包装器应该实现这个方法。

57.2.10. 用于路径重新参数化的FDW例程

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List *fdw_private,
                                 RelOptInfo *child_rel);
```

在把一个由给定子关系child_rel的最顶层父关系参数化的路径转换成由该子关系参数化的路径时会调用这个函数。该函数被用于重新参数化任意路径或者转化一个ForeignPath的给定fdw_private成员中保存的任意表达式节点。该回调可能会根据需要使用reparameterize_path_by_child、adjust_appendrel_attrs或者adjust_appendrel_attrs_multilevel。

57.3. 外部数据包装器助手函数

多个助手函数被从核心服务器输出，这样外部数据包装器的作者们可以很容易访问到FDW相关对象的属性，例如FDW选项。要使用任何其中一个函数，你需要在你的源文件中包括头文件foreign/foreign.h。这个头也定义了被这些函数返回的结构类型。

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

这个函数为具有给定 OID 的外部数据包装器返回一个ForeignDataWrapper对象。一个ForeignDataWrapper对象包含该FDW的特性（详见foreign/foreign.h）。

```
ForeignServer *
GetForeignServer(Oid serverid);
```

这个函数为一个具有给定 OID 的外部服务器返回ForeignServer对象。一个ForeignServer对象包含该服务器的特性（详见foreign/foreign.h）。

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

这个函数为在给定服务器上的给定角色的用户映射返回UserMapping对象（如果指定用户没有映射，它将返回PUBLIC的映射，如果也没有则抛出错误）。一个UserMapping对象包含该用户映射的特性（详见foreign/foreign.h）。

```
ForeignTable *
GetForeignTable(Oid relid);
```

该函数为一个具有给定 OID 的外部表返回ForeignTable对象。一个ForeignTable对象包含该外部表的特性（详见foreign/foreign.h）。

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

这个函数为一个具有给定外部表 OID 和属性号的列返回针对每一列的FDW选项，形式为一个DefElem列表。如果该列没有选项则返回 NIL。

某些对象类型除了基于OID的查找函数之外，还具有基于名称的查找函数：

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部数据包装器返回ForeignDataWrapper对象。如果包装器没有找到，在missing_ok为真时返回 NULL，否则抛出一个错误。

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部服务器返回ForeignServer对象。如果该服务器没有被找到，在missing_ok为真时返回 NULL，否则抛出一个错误。

57.4. 外部数据包装器查询规划

FDW回调函数GetForeignRelSize、GetForeignPaths、GetForeignPlan、PlanForeignModify、GetForeignJoinPaths、GetForeignUpperPaths及PlanDirectModify必须适合PostgreSQL规划器的工作。这里有一些关于它们必须做什么的注记。

root和baserel中的信息可以被用来减少必须从外部表获得的信息量（并且因此降低代价）。baserel->baserestrictinfo是特别有趣的，因为它包含限制条件（WHERE）子句，它应该被用来过滤要被获取的行（FDW本身并不要求强制这些条件，因为核心执行器可以检查它们）。baserel->reltarget->exprs可以被用来决定哪些类需要被获取；但是注意它仅列出了ForeignScan计划节点所发出的列，不包含在条件计算中使用但并不被查询输出的列。

有多个私有域可以给FDW规划函数来保存信息。通常，不管你存储什么在FDW私有域中，它们都应该被palloc，这样它会在规划结束时被回收。

baserel->fdw_private是一个void指针，它可以被FDW规划函数用来存储与特定外部表相关的信息。核心规划器不会碰它除非当RelOptInfo节点被创建时把它初始化为NULL。它对从GetForeignRelSize传递信息给GetForeignPaths和/或从GetForeignPaths传递信息给GetForeignPlan非常有用，这样避免了重新计算。

GetForeignPaths可以通过在ForeignPath节点的fdw_private域中存储私有信息来标识不同的访问路径。fdw_private被声明为一个List指针，但是可能实际上包含任何东西，因为规划器不会触碰它。但是，最好是使用一种nodeToString可导出的形式，这样在后端可以用于调试支持。

GetForeignPlan可以检查选中的ForeignPath节点的fdw_private域，并且可以生成被放置于ForeignPath计划节点中的fdw_exprs和fdw_private列表。这两个列表必须被表示为一种copyObject可复制的形式。fdw_private列表没有任何其他限制并且不会被核心后端以任何形式解释。非 NIL 的fdw_exprs应该包含表达式树，该树会在运行时被执行。这些树将由规划器在后期处理，以便让它们变成完全可执行的。

在GetForeignPlan中，通常被传入的目标列表可以被照样复制到计划节点中。被传入的scan_clauses 列表包含和baserel->baserestrictinfo相同的子句，但是可能为了更好的执行效率会被重新排序。在简单情况下，FDW可以只把RestrictInfo节点从scan_clauses 列表剥离（使用extract_actual_clauses）并且把所有子句放到计划节点的条件列表中，这意味着所有子句将在运行时由执行器检查。更复杂的FDW可能可以在内部检查某些子句，着这

种情况下哪些子句可以从计划节点的条件列表中删除，这样执行器就不用浪费时间去检查它们。

作为一个例子，FDW可以标识某些foreign_variable = sub_expression形式的限制子句，它决定哪些可以使用由sub_expression给出的本地计算值在远程服务器上被执行。这样一个子句的实际标识应该在GetForeignPaths期间发生，因为它可能会影响路径的代价估计。路径的fdw_private域可能包括一个已标识的子句的RestrictInfo节点。然后GetForeignPlan将从scan_clauses 中移除该子句，但是将sub_expression加到fdw_exprs来保证它被揉成可执行的形式。它可能还将把控制信息放入到计划节点的fdw_private域来告诉执行函数在运行时要做什么。传递给远程服务器的查询将涉及类似WHERE foreign_variable = $1的东西，使用在运行时从fdw_exprs表达式树获得的参数值。

任何从该计划节点的条件列表移除的子句必须被加入到fdw_recheck_quals或者由RecheckForeignScan重新检查以便确保在READ COMMITTED隔离级别的正确行为。当查询中涉及的某个其他表上发生并发更新时，执行器可能需要验证原来的所有条件仍然对该元组满足（可能用一组不同的参数值）。使用fdw_recheck_quals通常比在RecheckForeignScan中实现检查要更容易，但是这种方法不足以应付外连接被下推的情况，因为那种情况下的连接元组可能会有一些域具有 NULL 但是不会导致整个元组被拒绝。

另一个可以由 FDW 填充的ForeignScan域是fdw_scan_tlist，它描述 FDW 为这个计划节点返回的元组。对于简单的外部表扫描这可以设置为NIL，表示返回的元组具有为外部表声明的行类型。非-NIL值必须是一个包含表示返回列的 Var 或表达式的目标列表（TargetEntry的列表）。例如，这可以被用来显示 FDW 省略了某些查询不需要的列。还有，如果 FDW 计算表达式比在本地计算代价更低，可以把那些表达式加入到fdw_scan_tlist。注意连接计划（从GetForeignJoinPaths创建的路径得到）必须总是提供fdw_scan_tlist来描述它们将返回的列集合。

FDW应该总是只依靠表的限制子句构建至少一个路径。在连接查询中，它可能还会选择依靠连接子句构建路径，例如foreign_variable = local_variable。这样的子句将不会在baserel->baserestrictinfo中找到，但是必须出现在关系的连接列表中。使用这样一个子句的路径被称为一个“参数化路径”。它必须用一个合适的param_info值来标识其他被使用在选中的连接子句中的关系；使用get_baserel_parampathinfo来计算该值。在GetForeignPlan中，连接子句的local_variable部分将被加到fdw_exprs中，并且接着在运行时和一个普通限制子句一样工作。

如果一个 FDW 支持远程连接，GetForeignJoinPaths应该和GetForeignPaths对基本表所作的那样为潜在的远程连接产生ForeignPath。有关想要进行的连接的信息可以以上述相同的方式传递给GetForeignPlan。不过，baserestrictinfo与连接关系无关，一个特定连接的相关连接子句将被作为一个独立的参数（extra->restrictlist）被传递给GetForeignJoinPaths。

FDW 可能会额外地支持直接执行某些在扫描和连接层次之上的计划动作，例如分组或者聚集。为了提供这类选项，FDW 应该生成路径并且把它们插入到合适的上层关系中。例如，一条表示远程聚集的路径应该被使用add_path插入到UPPERREL_GROUP_AGG关系中。这条路径的代价将会与通过读取外部关系的简单扫描路径的本地聚集（注意这样一条路径也必须被提供，否则规划时会有错误）进行比较。如果远程聚集路径胜出（通常是这样），它会被以通常的方式（调用GetForeignPlan）转化成计划。如果该查询的所有基本关系都来自于同一个FDW，推荐在GetForeignUpperPaths回调函数中生成这种路径，该函数会为每一个上层关系被调用（即每一次扫描/连接后处理步骤）。

第 57.2.4 节描述的PlanForeignModify以及其他回调的设计是建立在这样一个假设之上：外部表将以通常的方式被扫描并且行更新将被一个本地ModifyTable计划节点所驱动。这种方法对于更新需要读取本地表以及外部表的一般情况下是必要的。不过，如果操作可以完全由外部服务器执行，FDW 可以产生一个表示这种操作的计划并且把它插入到UPPERREL_FINAL上层关系中，在其中它会与ModifyTable方法竞争。这种方法还可以被用来实现远程SELECT FOR UPDATE，而不使用第 57.2.5 节描述的行锁定回调。记住插入到UPPERREL_FINAL中的路径负责实现查询的所有行为。

在规划一个UPDATE或DELETE时，PlanForeignModify和PlanDirectModify能为外部表查找RelOptInfo结构，并利用之前由扫描规划函数创建的baserel->fdw_private数据。但是，

在INSERT中目标表不会被扫描，因此不会有它的RelOptInfo。由PlanForeignModify返回的List具有和ForeignScan计划节点的fdw_private列表相同的限制，即它必须只包含copyObject知道怎么拷贝的结构。

带有一个ON CONFLICT子句的INSERT不支持指定冲突目标，因为本地不知道远程表上的唯一约束和排除约束的情况。然后这也意味着ON CONFLICT DO UPDATE不被支持，因为该说明是强制性的。

57.5. 外部数据包装器中的行锁定

如果一个 FDW 的底层存储机制具有锁定行的概念来阻止对行的并发更新，通常值得 FDW 去执行行级锁定以尽可能接近在普通PostgreSQL表中所实际使用的语义。涉及这个问题有多种考虑。

要做出的一个关键决定是执行早期锁定还是晚期锁定。在早期锁定中，当一行被第一次从底层存储中检索到时，它会被锁定；而在晚期锁定中，只有当行需要被锁定时才锁定它（由于某些行可能被本地检查的限制或者连接条件抛弃，所以会出现不同）。早期锁定更加简单并且能避免额外地与远程存储交互，但是可能会导致一些不需要锁定的行也被锁定，最终造成并发性下降甚至意外的死锁。还有，只有在要被锁定的行可以在后期唯一地重新标识时才可以用晚期锁定。较好的行标识符应该能标识行的特定版本，就像PostgreSQL TID 那样。

默认情况下，PostgreSQL在与 FDW 交互时会忽略锁定考虑，但是 FDW 可以在没有核心代码显式支持的情况下执行早期锁定。第 57.2.5 节描述的 API 函数（在PostgreSQL 9.5 中加入）允许 FDW 按照意愿使用晚期锁定。

一个额外的考虑是在READ COMMITTED隔离模式中，PostgreSQL可能需要对某个目标元组的更新版本进行限制以及连接条件的重新检查。重新检查连接条件要求重新获得之前连接成目标元组的非目标行拷贝。在标准PostgreSQL表的情况下，这可以通过在连接投影出的列列表中包括非目标表的 TID 并且在需要时重新取得非目标行来做到。这种方法可以让连接数据集保持紧凑，但是它要求代价较低的重新取得元组的功能，还有 TID 要能够唯一地标识要被重新取得的行版本。因此，默认情况下用于外部表的方法是将整个外部表元组的拷贝包括在从连接投影出的列列表中。这不会对 FDW 有特殊的要求，但是会导致归并和哈希连接性能下降。要满足重新取得元组需求的 FDW 可以选择第一种方式。

对于在外部表上的UPDATE或者DELETE，推荐目标表上的ForeignScan操作在它取得的行上执行早期锁定（可能通过SELECT FOR UPDATE的等效体）。通过在规划时比较一个表的 relid 和root->parse->resultRelation或在执行时使用ExecRelationIsTargetRelation()，一个 FDW 可以检测该表是否为UPDATE/DELETE的目标。另一种可能性是在ExecForeignUpdate或者ExecForeignDelete回调中执行晚期锁定，但是对此没有特别的支持。

对于通过SELECT FOR UPDATE/SHARE命令指定要被锁定的外部表，ForeignScan操作同样可以通过用SELECT FOR UPDATE/SHARE的等效体取元组来执行早期锁定。要执行晚期锁定，请提供第 57.2.5 节定义的回调函数。在GetForeignRowMarkType中，根据请求的锁长度来选择行标记选项ROW_MARK_EXCLUSIVE、ROW_MARK_NOKEYEXCLUSIVE、ROW_MARK_SHARE或者ROW_MARK_KEYSHARE（不管选择哪一种选项，核心代码都会做同样的事情）。在别的地方，可以在规划时用get_plan_rowmark或者在执行时用ExecFindRowMark来检测一个外部表是否被指定由这种类型的命令锁定。你必须不仅仅检测是否返回了一个非空的行标记结构，还要检测它的strength域不是LCS_NONE。

最后，对于在UPDATE、DELETE或者SELECT FOR UPDATE/SHARE命令中使用但是没有被指定要行锁定的外部表，你可以在看到锁长度LCS_NONE时通过使用GetForeignRowMarkType选择选项ROW_MARK_REFERENCE来把默认选择覆盖为拷贝整个行。 这将导致用那个值作为markType来调用RefetchForeignRow。它应该接着重新取得该行而不获取任何新锁（如果你有一个GetForeignRowMarkType函数，但是不想重新取未锁定的行，可为LCS_NONE选择选项ROW_MARK_COPY）。

更多信息可见src/include/nodes/lockoptions.h，以及src/include/nodes/plannodes.h中RowMarkType和PlanRowMark的注释，还有src/include/nodes/execnodes.h中ExecRowMark的注释。

第 58 章 编写一种表采样方法

PostgreSQL的TABLESAMPLE子句实现支持 在 SQL 标准要求的BERNOULLI和SYSTEM方法之 外自定义表采样方法。采样方法决定了使用TABLESAMPLE子句时 表的哪些行会被选择。

在 SQL 层上，一种表采样方法被表达为一个 SQL 函数（通常用 C 实现）， 其签名是：

```
method_name(internal) RETURNS tsm_handler
```

函数的名称是出现在TABLESAMPLE子句中的同一个方法名称。 internal参数是不起作用的（总是值为零），它仅仅是为了阻止 直接从 SQL 命令中调用该函数。函数的结果必须是一个 palloc 好的 TsmRoutine结构，它包含了该采样方法的支持函数的指针。 这些支持函数是纯 C 函数并且对于 SQL 层面不可见也不可调用。支持函数见 第 58.1 节

除了函数指针之外，TsmRoutine结构必须提供这些额外的域：

List *parameterTypes

这是一个 OID 列表，它包含了使用这种采样方法时TABLESAMPLE 子句接受的参数的数据类型 OID。例如，对于内建方法，这个列表只包含一个值 为FLOAT4OID的项，它表示采样的百分数。自定义采样方法可以有 更多或者不同的参数。

bool repeatable_across_queries

如果为true，当每次查询时给出相同的参数和 REPEATABLE种子值且表内容没有改变时，采样方法可以在 连续的查询中给出相同的采样。当这个域为false时，不能把REPEATABLE子句用于这种采样方法。

bool repeatable_across_scans

如果为true，这种采样方法在同一个查询的连续扫描中给出 相同的采样（假定参数、种子值和快照都不变）。当这个域为 false时，规划器将不会选择要求扫描被采样表多于一次的计划， 因为那会导致不一致的查询输出。

TsmRoutine结构类型被声明在 src/include/access/tsmapi.h中，需要更多细节可以参考该文件。

在尝试编写自己的采样方法时，包括在标准发布中的表采样方法是很好的参考。内建 采样方法的源代码树可见src/backend/access/tablesample子目录， 在contrib子目录中可以找到额外的方法。

58.1. 采样方法支持函数

TSM 处理器函数返回一个 palloc 好的TsmRoutine结构，其中包含 了下文所述的支持函数的指针。大部分函数是必须的，但是有些是可选的（它们 的指针可以为 NULL）。

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

这个函数在规划期间被掉欧勇。它必须估计在一次采样扫描中会被读到的关系 页面数，以及将被该扫描所选择的元组数（例如，可以先估计采样分数， 乘上baserel->pages和baserel->tuples 数，并且把结果圆整）。paramexprs列表保存作为

TABLESAMPLE子句的参数的表达式。如果出于优化的目的需要 这些表达式的值，我们推荐使用estimate_expression_value() 来尝试将这些表达式变成常量。但是即便这些表达不能被简化，该函数必须提供 估计的尺寸，并且即使出现不合法的值它也不应该失败（记住它们只是运行时值 的估计）。pages和tuples参数是输出。

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

为 SampleScan 计划节点的执行进行初始化。这会在执行器启动时被调用。 它应该执行执行处理启动所需的任何初始化工作。 SampleScanState节点已经被创建，但是它的tsm_state域为 NULL。 InitSampleScan函数可以 palloc 任何采样方法需要的内部状态数据，并且把它的一个指针存储在node->tsm_state 中。有关要扫描的表的信息可以通过SampleScanState 节点的其他域访问（但是要注意 node->ss.ss_currentScanDesc扫描描述符还没有被设置）。 eflags包含描述这个计划节点的执行器操作模式的标志位。

当(eflags & EXEC_FLAG_EXPLAIN_ONLY)为真时，该 扫描将不会被真正执行，因此这个函数应该只做最少的事情，让该节点状态对 EXPLAIN和EndSampleScan可用。

这个函数可以被省略（把指针设置为 NULL），在那种情况下 BeginSampleScan必须执行采样方法所需的所有初始化工作。

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

开始执行一次采样扫描。就在第一次尝试取得一个元组时就会调用这个函数， 如果该扫描需要被重启可能还要再次调用它。有关要扫描的表的信息可以通过 SampleScanState节点的其他域访问（但是要注意 node->ss.ss_currentScanDesc扫描描述符还没有被设置）。 长度为nparams的params数组包含在 TABLESAMPLE子句中提供的参数的值。这些参数的编号和类型 在采样方法的parameterTypes列表中指定，并且已经被 检查过不为空。seed包含用于在采样方法中生成任何随机数的 种子。如果给定了REPEATABLE值，种子将是该值的哈希。如果 没有指定REPEATABLE值，种子将是random()的 结果。

这个函数可能会调整域node->use_bulkread 以及node->use_pagemode。 如果node->use_bulkread为true（默认）， 该扫描将使用一种鼓励重用缓冲区的缓冲区访问策略。如果该扫描只访问 该表的页面的一小部分，将这个域设置为false比较合理。 如果node->use_pagemode为true（默认）， 那么对于每一个被访问的页面上的所有元组，该扫描将会在一趟中执行它们 的可见性检查。如果该扫描只选择每个被访问页面上的一小部分，将这个域 设置为false比较合理。这将导致执行较少次的元组可见性检查， 但是每一次都会代价更大，因为需要更多锁定。

如果采样方法被标记为repeatable_across_scans，在重 新扫描时，它必须能够选择和第一次扫描相同的元组集合，也就是说对 BeginSampleScan的一次新调用必须选择和之前相同的元组 （如果TABLESAMPLE参数和种子没有变化）。

```
BlockNumber
NextSampleBlock (SampleScanState *node);
```

返回下一个要扫描的页面的块号，如果没有剩余的页面需要扫描则返回 InvalidBlockNumber。

这个函数可以被省略（设置指针为 NULL），在那种情况下核心代码将 执行整个关系的一次顺序扫描。这样一个扫描可以使用同步扫描，这样 采样方法不能假定每一次扫描都用同样的顺序访问关系页面。

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

返回指定页面上下一个要被采样的元组的偏移量，如果没有剩余元组需要被采样，则返回InvalidOffsetNumber。maxoffset是页面上 使用的最大偏移量。

> **注意**
>
> NextSampleTuple没有被显式地告知范围 1 .. maxoffset中的哪些偏移量真正包含了合法的元组。这通常不 成问题，因为核心代码会忽略采样丢失或者不可见元组的请求。这不会导致采样 中的任何偏差。不过，如果必要，该函数可以检查 node->ss.ss_currentScanDesc->rs_vistuples[]来判断哪些 元组合法并且可见（这要求node->use_pagemode为 true）。

> **注意**
>
> NextSampleTuple不能假定 blockno是最近一次NextSampleBlock调用返回的同一个页面号。它由之前某次NextSampleBlock调用所返回， 但是核心代码被允许在真正扫描页面之前调用NextSampleBlock， 以便支持预取。假定一旦一个给定页面的采样开始，连续的 NextSampleTuple调用都将引用同一个页面（直到返回 InvalidOffsetNumber）。

```
void
EndSampleScan (SampleScanState *node);
```

结束扫描并且释放资源。释放 palloc 过的内存通常并不重要，但是任何外部 可见的资源应该被清除。在没有这类资源存在的通常情况下，这个函数可以被 省略（设置指针为NULL）。

第 59 章 编写一个自定义扫描提供者

PostgreSQL支持一组实验性的功能，它们的目的是允许扩展模块向系统中增加新的扫描类型。与外部数据包装器不同（只负责给出如何扫描其自身的外部表的知识），自定义扫描提供者可以提供另一种扫描系统中任一关系的方法。通常，编写一个自定义扫描提供者的动机是允许使用某种核心系统不支持的优化，例如缓冲或者某种形式的硬件加速。这一章简要介绍了如何编写一个新的自定义扫描提供者。

实现一个新类别的自定义扫描分成三步。首先，在规划期间需要生成表达使用所提出策略的扫描的访问路径。然后，如果规划器选择这些访问路径之一作为最优策略来扫描一个特定关系，该访问路径必须被转换成计划。最后，必须能执行该计划并且产生和其他以同一关系为目标的访问路径相同的结果。

59.1. 创建自定义扫描路径

一个自定义扫描提供者将通过设置下面的钩子函数来为基本关系增加路径， 在核心代码已经为该关系产生了所有访问路径集后（除了在此调用之后生成的Gather路径，以便它们可以使用被钩子添加的部分路径），这个钩子函数将被调用。

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

尽管这个钩子函数可被用来检查、修改或者移除核心系统产生的路径，自定义扫描提供程序通常还是局限于产生CustomPath对象并且使用add_path把它们加入到rel中。自定义扫描提供者负责初始化CustomPath对象，它被声明为这样：

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

path必须像任何其他路径一样被初始化，包括行计数估计、启动和总代价以及这条路径提供的排序顺序。flags是一个位掩码，如果该自定义路径能够支持反向扫描则它应该包括CUSTOMPATH_SUPPORT_BACKWARD_SCAN，如果支持标记和恢复则应该包括CUSTOMPATH_SUPPORT_MARK_RESTORE。这两种能力都是可选的。可选的custom_paths域是由这个自定义路径节点使用的Path节点的列表，这些将被规划器转换成Plan节点。custom_private可被用来存储该自定义路径的私有数据。私有数据应该被存储为能被nodeToString处理的形式，这样尝试打印该自定义路径的调试例程才能正常地工作。methods必须指向一个实现了所需自定义路径方法的对象（通常是静态分配的），当前只有一种方法。LibraryName和SymbolName域必须也被初始化，这样动态载入器才能解析它们来定位方法表。

一个自定义扫描提供者还能提供连接路径。就和基本关系一样，这样一条路径也应该产生和它将要替换的连接所产生的相同的输出。要做到这一点，连接提供程序应该设置下面的钩子函数，并且在该钩子函数里为连接关系创建CustomPath路径。

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
```

```
                                                RelOptInfo *joinrel,
                                                RelOptInfo *outerrel,
                                                RelOptInfo *innerrel,
                                                JoinType jointype,
                                                JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

对于同一个连接关系，这个钩子将被反复调用，因为要对不同的内外关系组合生成路径，所以如何最小化可能的重复工作是钩子函数的责任。

59.1.1. 自定义扫描路径回调

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

将一条自定义路径转换为一个完成的计划。返回值通常将是一个CustomScan对象，回调函数必须负责分配并且初始化这个对象。详见第 59.2 节

59.2. 创建自定义扫描计划

以一棵已完成的计划树表示的自定义扫描使用下面的结构：

```
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

scan必须和任何其他扫描一样被初始化，包括估计代价、目标列表、条件等等。flags是一个位掩码，它的含义和CustomPath中的一样。custom_plans可以用来存储子Plan节点。custom_exprs应该被用来存储需要由setrefs.c和subselect.c修整的表达式树，而custom_private应该被用来存储其他只由自定义扫描提供者本身使用的私有数据。在扫描一个基本关系时，custom_scan_tlist可以为 NIL，表示该自定义扫描返回符合该基本关系行类型的扫描元组。否则，它是一个描述实际扫描元组的目标列表。对于连接必须提供custom_scan_tlist。如果自定义扫描提供者能够计算某些非-Var 表达式，也应该提供这个域的值。custom_relids会被核心代码设置成这个扫描节点要处理的关系的集合（范围表索引）。当这个扫描被放在一个链接上时是一种例外，那时其中只有一个成员。methods必须指向一个实现了所需自定义扫描方法的对象（通常是静态分配的），将进一步在下文详细介绍。

当一个CustomScan扫描单个关系时，scan.scanrelid必须是被扫描的表的范围表索引。当它替代的是一个连接时，scan.scanrelid应该为零。

计划树必须能够被使用copyObject复制，因此所有存储在“custom”域中的数据必须由该函数能处理的节点构成。更进一步，自定义扫描提供者不能把CustomScan结构本身替换成包含CustomScan的更大的结构（就好像CustomPath或者CustomScanState）。

59.2.1. 自定义扫描计划回调

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

为这个CustomScan分配一个CustomScanState。实际的分配常常会比一个普通CustomScanState所要求的空间要大，因为很多提供者希望把它嵌入在一个更大的结构中作为第一个域。返回的值必须有节点标签并且设置好了合适的methods，不过在这个阶段其他域应该被设置为零。在ExecInitCustomScan执行基本的初始化之后，将调用BeginCustomScan回调函数来让自定义扫描提供者有机会做其他需要干的事情。

59.3. 执行自定义扫描

在执行一个CustomScan时，它的执行状态由一个CustomScanState表示，其定义如下：

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

ss和任何其他扫描状态一样被初始化，不过如果该扫描是用于连接而不是基本关系，则ss.ss_currentRelation会被留成 NULL。flags是一个位掩码，它的含义与CustomPath和CustomScan中的一样。methods必须指向一个实现了所需自定义扫描状态方法的对象（通常是静态分配的），将进一步在下文详细介绍。通常一个CustomScanState（不需要支持copyObject）实际将是一个较大的结构，上面的结构将嵌入在其中作为第一个成员。

59.3.1. 自定义扫描执行回调

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

完成所提供的CustomScanState的初始化。标准的域已经被ExecInitCustomScan初始化，但是任何私有的域应该在这里被初始化。

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

取下一个扫描元组。如果还有任何元组剩余，它应该用当前扫描方向的下一个元组填充ps_ResultTupleSlot，并且接着返回该元组槽。如果没有，则用NULL填充或者返回一个空槽。

```
void (*EndCustomScan) (CustomScanState *node);
```

清除任何与CustomScanState相关的私有数据。这个方法是必需的，但是如果没有相关的数据或者相关数据将被自动清除，则它不需要做任何事情。

```
void (*ReScanCustomScan) (CustomScanState *node);
```

把当前扫描倒回到开始处，并且准备重新扫描该关系。

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

保存当前的扫描位置，这样可以在以后由RestrPosCustomScan回调函数恢复。这个回调函数是可选的，只有在CUSTOMPATH_SUPPORT_MARK_RESTORE标志被设置时才需要提供。

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

恢复由MarkPosCustomScan回调函数保存的扫描位置。这个回调函数是可选的，只有在CUSTOMPATH_SUPPORT_MARK_RESTORE标志被设置时才需要提供。

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

估计并行操作所需要的动态共享内存的数量。这可能会比实际使用的量更大，但是绝不能更低。返回值的单位是字节。这个回调是可选的，只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                 ParallelContext *pcxt,
                                 void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate 指向一块大小等于EstimateDSMCustomScan 返回值的共享内存区域。这个回调是可选的， 只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

当自定义扫描计划节点即将被重新扫描时， 重新初始化并行操作所需的动态共享内存。这个回调是可选的， 只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。 推荐的做法是，此回调仅重置共享状态，而ReScanCustomScan 回调仅重置本地状态。目前，该回调将在ReScanCustomScan 之前调用，但最好不要依赖该顺序。

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
                                    void *coordinate);
```

基于InitializeDSMCustomScan期间通过领导者 设置的共享状态初始化并行工作者的本地状态。这个回调是可选的， 只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

预计节点将不会执行完成时释放资源。并不是在所有情况下都调用；有时，EndCustomScan可能会在调用此函数之前调用。 由于并行查询使用的DSM段在调用此回调后即被销毁， 因此希望在DSM段消失之前采取某些操作的自定义扫描提供程序应实现此方法。

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

为一个自定义扫描计划节点的EXPLAIN输出额外的信息。这个回调函数是可选的。即使没有这个回调函数，被存储在ScanState中的公共的数据（例如目标列表和扫描关系）也将被显示，但是该回调函数允许显示额外的信息（例如私有状态）。

第 60 章 遗传查询优化器

作者

Written by Martin Utesch (<utesch@aut.tu-freiberg.de>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

60.1. 将查询处理看成是一个复杂的优化问题

在所有关系操作符中，最难于处理和优化的是连接。可能的查询计划数目以查询中连接数量的指数增长。对各种各样处理独立连接的连接方法（如PostgreSQL中的嵌套循环、哈希连接、归并连接）和多种关系访问方法的indexes（如PostgreSQL中的 B 树、哈希、GiST 和 GIN）的支持也进一步加重了优化的负担。

通常的PostgreSQL查询优化器会执行一次在可选策略空间上的近似穷举搜索。这个算法最早由 IBM 的系统 R 数据库引入，它能产生接近最优的连接顺序，但是当查询中的连接数增长到很大时，该算法需要大量的时间和内存空间。这使得普通的PostgreSQL查询优化器不适合需要连接大量表的查询。

德国弗莱堡的矿业大学自动控制学院在使用PostgreSQL作为电力网格维护决策支持知识系统的后端时遇到了一些问题。DBMS 需要为知识系统中的推理机器处理大量连接查询。这些查询中的连接数不可能用普通的查询优化器来处理。

接下来我们将介绍一种遗传算法的实现，它被用来以一种更有效率的方式为涉及大量连接的查询解决连接顺序问题。

60.2. 遗传算法

遗传算法（GA）是一种通过随机搜索操作的启发式优化方法。优化问题的可能的解决方案集合被看成是个体的种群。一个个体对于它的环境的适应程度由其适应度指定。

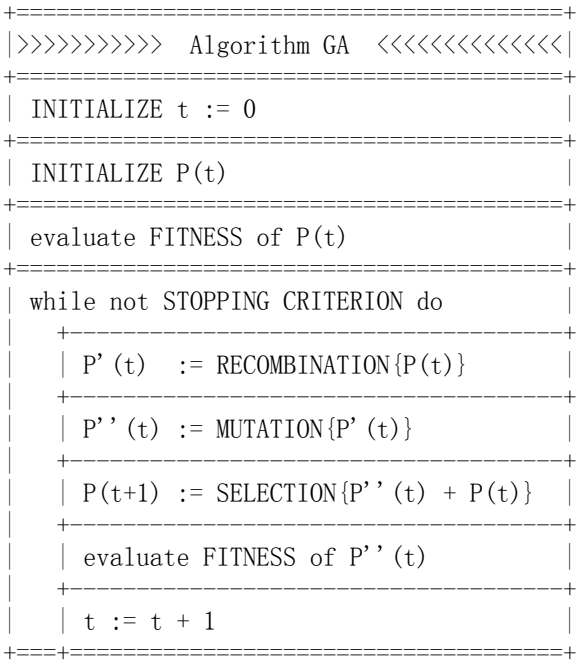
搜索空间中一个个体的座标被表示为染色体，实质上是一个字符串集合。一个基因是一个染色体的一个片段，它编码了一个要被优化参数的值。一个基因的典型编码包括二进制或整数。

通过对重组、变异和选择的模拟，比父辈平均适应度更好的新一代搜索点将被找到。

根据comp.ai.genetic FAQ，GA并非是一个纯粹的随机搜索。GA会使用随机处理，但是结果的确是非随机的（比随机好）。

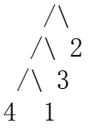
图 60.1. 一种遗传算法的结构图

P(t)	在时刻 t 的祖先代
P''(t)	在时刻 t 的后代



60.3. PostgreSQL 中的遗传查询优化（GEQO）

GEQO模块把查询优化问题当做著名的货郎担问题（TSP）来处理。可能的查询计划被编码为整数的串。每一个串表示从查询中一个关系到下一个关系的连接顺序。例如，连接树



被编码为整数串 '4-1-3-2'，它表示首先连接关系 '4' 和 '1'，然后连接 '3'，最后连接 '2'。这里 1、2、3、4 是PostgreSQL优化器中的关系 ID。

PostgreSQL中GEQO实现的特点有：

- 一种稳态 GA（在种群中替换适应度最差的个体，而不是整代替换）的使用允许对改进的查询计划快速收敛。这对在合理时间内处理查询是最重要的；
- 边重组杂交的使用特别适合于通过GA为TSP的解决方案保持低丢边率；
- 遗传操作符变异被废弃，这样不需要修补机制来产生合法的TSP旅行。

GEQO模块的一部分是从 D. Whitley 的遗传算法中改编而来。

GEQO模块允许PostgreSQL查询优化器支持通过非穷举搜索高效地处理大量连接的查询。

60.3.1. 用GEQO产生可能的计划

GEQO规划处理使用标准的规划器代码来产生用于扫描个体关系的计划。然后使用遗传方法发展连接计划。如上所示，每一个候选连接计划被表示为一个连接基本关系的序列。在初始阶

段，GEQO代码简单地随机产生某些可能的连接序列。对于被考虑的每一个连接序列，标准规划器代码被调用来估算使用该序列执行查询的代价（对于连接序列的每一步，所有三种连接策略都被考虑；并且所有初始决定的关系扫描计划都可用。估计的代价是这些可能性中最低的那个。）。具有较低估计代价的连接序列被认为比具有较高代价的“更适合”。遗传算法会丢弃最不适应的候选。然后通过组合更适合的候选的基因来产生新的候选 — 即使从已知代价低的连接序列随机选择片段来创建用于考虑的新序列。这个处理将被重复，直到已经考虑的连接序列的数量达到一个预设值。然后在搜索中任何时候找到的最好的一个将被用来产生最终的计划。

由于在初始种群选择和后续最佳候选的“变异”过程中都采用了随机选择，所以这种处理天生就是非确定性的。要避免被选中计划发生出乎意料的改变，每次 GEQO 算法的运行都会使用当前geqo_seed参数设置来重启它的随机数生成器。只要geqo_seed以及其他 GEQO 参数保持固定（以及其他规划器输入，如统计信息），对一个给定的查询将产生相同的计划。要试验不同的搜索路径，可以尝试改变geqo_seed。

60.3.2. PostgreSQL GEQO的未来实现任务

仍需对改进遗传算法的参数设置做一些工作。在文件src/backend/optimizer/geqo/geqo_main.c、例程gimme_pool_size和gimme_number_generations中，我们必须为参数设置找到一种折中来满足两个互相竞争的需求：

- 查询计划的最优性
- 计算时间

在当前的实现中，每一个候选连接序列的适应度通过运行标准规划器的连接选择和代价估计代码从零估算而来。即使不同的候选中使用了相似的连接子序列，还是需要重复大量的工作。通过保留子连接的代价估计可以在这种情况下节省很多时间。但问题在于要避免为了保留那样的状态而不合理地占用过多内存。

在更基础的层面上，用一个为 TSP 设计的 GA 算法来解决查询优化问题是否合适也还需要探讨。在 TSP 情况中，与任何子串（部分旅程）相关的代价独立于剩余的旅程，但是对于查询优化肯定不是这样。因此边重组杂交是否比变异过程更有效还存有疑问。

60.4. 进一步阅读

下列资源包含关于遗传算法更多的信息：

- The Hitch-Hiker's Guide to Evolutionary Computation[1], (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design[2], by Craig Reynolds
- [elma04]
- [fong]

[1] http://www.aip.de/~ast/EvolCompFAQ/
[2] http://www.red3d.com/cwr/evolve.html

第 61 章 索引访问方法接口定义

本章定义核心PostgreSQL系统和管理个别索引类型的索引访问方法之间的接口。除 了在这里指定的内容之外，核心系统对索引一无所知，因此可以通过编写附加代码来开发全新的索引类型。

PostgreSQL中所有的索引在技术上都叫做二级索引。也就是说，索引在物理上与它描述的表文件分离。每个索引被存储为它自己的物理关系并且被pg_class目录中的一个项所描述。一个索引的内容完全受到其索引访问方法控制。实际上，所有索引访问方法都把索引划分成标准大小的页面， 这样它们就可以使用常规的存储管理器和缓冲区管理器来访问索引内容（所有现有的索引访问方法还使用第 68.6 节描述的标准页面布局，并且大部分都使用相同的索引元组头部格式；但是这些决定都不是强制在访问方法上的）。

索引实际上是一些数据键值与索引父表中行版本（元组）的元组标识符或TIDs之间的映射。一个 TID 由一个块号和一个块内的项编号组成（见第 68.6 节。 这对于从表中取一个特定行就足够了。索引并不直接知道在 MVCC下，同一个逻辑行可能有多个现存的版本；对于索引而言，每个 行都是一个独立的对象，都需要自己的索引项。因此，对一行的更新总是为该行创建全新的索引项，即使键值没有改变（HOT 元组对这段陈述来说是个异常，但是索引也不会处理这些）。死亡元组的索引项将在随着死亡元组的回收而被回收（通过清理）。

61.1. 索引的基本 API 结构

每一个索引访问方法都由pg_am系统目录中的一行所描述。pg_am项为该访问方法指定了名称和一个处理器函数。这些项可以用CREATE ACCESS METHOD和DROP ACCESS METHOD SQL 命令创建和删除。

一个索引访问方法的处理器函数必须被声明为接受单一的类型为internal类型的参数并且返回伪类型index_am_handler。该参数是一个无用值，它只是被用来防止从 SQL 命令直接调用处理器函数。该函数的结果必须是一个已经 palloc 过的IndexAmRoutine类型结构，它包含核心代码使用该索引访问方法所需的所有信息。IndexAmRoutine结构（也被称为访问方法的API 结构）中的域指定了该访问方法的各种固定性质，例如它是否支持多列索引。更重要的是，它包含用于该访问方法的支持函数的指针，这些函数会完成真正访问索引的工作。这些支持函数是纯 C 函数，并且在 SQL 层面不可见也不可调用。支持函数在第 61.2 节介绍。

IndexAmRoutine结构定义如下：

```
typedef struct IndexAmRoutine
{
    NodeTag     type;

    /*
     * Total number of strategies (operators) by which we can traverse/search
     * this AM.  Zero if AM does not have a fixed set of strategy assignments.
     */
    uint16      amstrategies;
    /* total number of support functions that this AM uses */
    uint16      amsupport;
    /* does AM support ORDER BY indexed column's value? */
    bool        amcanorder;
    /* does AM support ORDER BY result of an operator on indexed column? */
    bool        amcanorderbyop;
    /* does AM support backward scanning? */
    bool        amcanbackward;
    /* does AM support UNIQUE indexes? */
    bool        amcanunique;
    /* does AM support multi-column indexes? */
```

```
    bool        amcanmulticol;
    /* does AM require scans to have a constraint on the first index column? */
    bool        amoptionalkey;
    /* does AM handle ScalarArrayOpExpr quals? */
    bool        amsearcharray;
    /* does AM handle IS NULL/IS NOT NULL quals? */
    bool        amsearchnulls;
    /* can index storage data type differ from column data type? */
    bool        amstorage;
    /* can an index of this type be clustered on? */
    bool        amclusterable;
    /* does AM handle predicate locks? */
    bool        ampredlocks;
    /* does AM support parallel scan? */
    bool        amcanparallel;
    /* does AM support columns included with clause INCLUDE? */
    bool        amcaninclude;
    /* type of data stored in index, or InvalidOid if variable */
    Oid         amkeytype;

    /* interface functions */
    ambuild_function ambuild;
    ambuildempty_function ambuildempty;
    aminsert_function aminsert;
    ambulkdelete_function ambulkdelete;
    amvacuumcleanup_function amvacuumcleanup;
    amcanreturn_function amcanreturn;   /* can be NULL */
    amcostestimate_function amcostestimate;
    amoptions_function amoptions;
    amproperty_function amproperty;     /* can be NULL */
    amvalidate_function amvalidate;
    ambeginscan_function ambeginscan;
    amrescan_function amrescan;
    amgettuple_function amgettuple;     /* can be NULL */
    amgetbitmap_function amgetbitmap;   /* can be NULL */
    amendscan_function amendscan;
    ammarkpos_function ammarkpos;       /* can be NULL */
    amrestrpos_function amrestrpos;     /* can be NULL */

    /* interface functions to support parallel index scans */
    amestimateparallelscan_function amestimateparallelscan;    /* can be NULL */
    aminitparallelscan_function aminitparallelscan;    /* can be NULL */
    amparallelrescan_function amparallelrescan;    /* can be NULL */
} IndexAmRoutine;
```

要想真正有用，一个索引访问方法还必须有一个或多个定义在pg_opfamily、 pg_opclass、pg_amop和 pg_amproc中的操作符族和操作符类。这些项允许规划器判断哪种查询条件适用于这个索引访问方法的索引。操作符族和类在第 38.15 节描述，它是阅读本章所需的前导材料。

一个独立的索引是由一个pg_class项定义的，该项描述索引为一个物理关系。还要加上一个pg_index项来显示索引的逻辑内容 — 也就是说，它所拥有的索引列集以及这些列的语义是被相关操作符类刻画的。索引列（键值）可以是底层表的 简单列，也可以是该表行上的表达式。索引访问方法通常不关心索引的键值来自那里（它总是操作预计算过的键值），但是它会对pg_index中的操作符类信息很感兴趣。所有这些目录项都可以被当作关系数据结构的一部分访问，这个数据结构会被传递给索引上的所有操作。

IndexAmRoutine中的有些标志域的含义并不那么直观。amcanunique的要求在第 61.5 节讨论。amcanmulticol标志断言该索引访问方法支持多列索引， amoptionalkey断言它允许对

那种在第一个索引列上没有给出可索引限制子句的扫描。如果amcanmulticol为假，那么amoptionalkey实际上说的是该访问方法是否允许不带限制子句的全索引扫描。 那些支持多索引列的访问方法必须支持那些在省略了除第一个列之外的任何或所有其它列上约束的扫描；不过，它们被允许去要求在第一个列上出现一些限制，并且这一点是以把amoptionalkey设置为假作为标志的。一个索引 AM 可能将amoptionalkey设置为假的一种原因是，如果它不索引空值。因为大多数可索引的操作符都是严格的并且因此不能对空输入返回真，所以不为空值存储索引项咋看上去很吸引人：因为它们不 可能被一个索引扫描返回。不过，当一个索引扫描对于一个给定索引列上没有约束子句时，这种讨论就不成立了。实际上，这意 味着设置了amoptionalkey为真的索引必须索引空值，因为规划器可能会决定在根本没有扫描键的时候使用这样的索引。一个相关的限制是一个支持 多索引列的索引访问方法必须支持索引第一列之后的列中的空值，因 为规划器会认为这个索引可以用于在那些列上没有限制的查询。例如，考虑一个在(a,b)上的索引和一个有WHERE a = 4的查询。系统会认为该索引可以用于扫描 a = 4的行， 如果索引忽略了 b 为空的行，那么就是错误的。不过，忽略那些在第一个索引列上值为空的行是 OK 的。一个索引空的索引访问方法可能也会设置amsearchnulls，表明它支持将IS NULL和IS NOT NULL子句作为搜索条件。

61.2. 索引访问方法函数

索引访问方法必须在IndexAmRoutine中提供的索引构造和维护函数有：

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

创建一个新索引。索引关系已经被物理创建，但是是空的。必须用索引访问方法要求的固定数据填充它，外加所有已经在表里的行的项。通常，ambuild函数会调用IndexBuildHeapScan()来扫描表以获取现有元组并计算需要被插入到索引的键。该函数必须返回一个已分配内存的结构，其中包含关于新索引的统计信息。

```
void
ambuildempty (Relation indexRelation);
```

构建一个空索引，并且把它写入到给定关系的初始化分叉中（ INIT_FORKNUM）。只会为不做日志的索引调用这个方法，被写入到 初始化分叉的空索引在每次服务器启动时将被复制到主关系分叉中。

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          IndexInfo *indexInfo);
```

向现有索引插入一个新元组。values和isnull数组给出需要被索引的键值，而heap_tid是要被索引的 TID。 如果该访问方法支持唯一索引（它的amcanunique标志为真），那么checkUnique指示要执行的唯一性检查类型。这根据唯一约束是否为可推迟的而变化，详见第 61.5 节通常在执行唯一性检查时访问方法仅需要heapRelation参数（因为那时它将不得不到堆中验证元组的存活性）。

该函数的布尔结果值仅仅在checkUnique为UNIQUE_CHECK_PARTIAL时才有意义。这种情况下一个“真”结果意味着这个新项是已知唯一的，反之“假”结果意味着它可能不 是唯一的（并且一个延迟的唯一性校验必须是预定的）。对于其他情况，建议使用一个常量“假”结果。

有些索引可能不会索引所有元组。如果元组不被索引，aminsert应该仅返回而什么都不做。

如果索引AM希望在SQL语句中连续的索引插入之间缓冲数据，它可以在indexInfo->ii_Context中分配空间并且在indexInfo->ii_AmCache（初始为NULL）中存放一个指向该数据的指针。

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

从索引中删除元组。这是一个“批量删除”操作，它的意图是通过扫描整个索引并检查每个项看它是否需要被删除。被传递进来的callback函数必须被调用（调用风格是：callback(TID, callback_state) returns bool）来判断任何其引用的 TID 标识的索引项是否需要删除。必须返回 NULL 或者是一个 palloc 过的、 包含删除操作效果的统计信息的结构。如果不需要向amvacuumcleanup传递信息，返回 NULL 也是 OK 的。

由于maintenance_work_mem被限制，在删除多行的时候ambulkdelete可能需要被调用多次。stats参数是对这个索引上一次调用的结果（在一个VACUUM操作中第一次调用时是NULL）。这将允许 AM 在整个操作过程中积累统计信息。典型的，如果被传递的stats非空，ambulkdelete将会修改并返回相同的结构。

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

在一个VACUUM操作（零个或更多次ambulkdelete调用）后清空。虽然不必做任何返回索引统计信息之外的事情，但是它可能执行批量清理，例如回收空索引页面。stats是最后一次ambulkdelete 调用返回的东西或者 NULL（如果没有元组需要删除而未调用ambulkdelete）。如果结果不是 NULL，那么它必须是一个已经被 palloc 的结构。它包含的统计信息将用于更新pg_class并且由VACUUM报 告（如果给出了VERBOSE）。如果索引在VACUUM操作期间根本没有改变，那么返回 NULL 也是可以的，否则必须返回正确的统计信息。

从PostgreSQL 8.4 开始，amvacuumcleanup将也会在一个ANALYZE操作结束时被调用。这种情况中stats总是 NULL 并且任何返回值都将会被忽略。这种情况可以通过检测info->analyze_only来区分。我们建议，在这样的调用中访问方法除了做插入后的清理之外什么也不做，并且那是仅仅是在一个自动清理工作者进程中。

```
bool
amcanreturn (Relation indexRelation, int attno);
```

通过返回型为一个IndexTuple的索引项的被索引列值，检查索引是否能在给定列上支持只用索引的扫描。属性编号从 1 开始编号，即第一列的 attno 是 1。如果支持返回 TRUE，否则返回 FALSE。如果访问方法 完全不支持只用索引的扫描，其IndexAmRoutine结构中的amcanreturn域可以被设置为 NULL。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
```

```
                double *indexPages);
```

估计一次索引扫描的开销。这个函数在下面的第 61.6 节有完整的讨论。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

分析和验证一个索引的 reloptions 数组。仅当一个索引存在非空 reloptions 数组时才会被调用。reloptions是一个text数组，包含name=value形式的项。 该函数应当构建一个bytea值，该值将被拷贝进索引的 relcache 项的rd_options域。bytea值的数据内容是开放由访问方法定义的， 大部分的标准访问方法都使用StdRdOptions结构。当validate为真时，如果任何一个选项都不可识别或者含有非法值，该函数都应当报告一个适当的错误消息；当validate为假时，非法 项应该被安静地忽略（当正在载入的选项已经在pg_catalog中时， validate为假；仅在访问方法已经改变了选项的规则时才可能找 到非法项，并且在此情况下忽略废弃的项是合适的）。如果想要默认行为，那么返回 NULL 也OK。

```
bool
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
            bool *res, bool *isnull);
```

amproperty方法允许索引方法覆盖pg_index_column_has_property和相关函数的默认行为。如果访问方法对于索引性质查询没有指定特殊的行为，其IndexAmRoutine结构的amproperty域可以被设置为 NULL。否则，对于pg_indexam_has_property调用会使用均为 0 的index_oid和attno参数来调用amproperty方法；对于pg_index_has_property调用会使用有效的index_oid和为 0 的attno参数来调用amproperty方法；对于pg_index_column_has_property调用会使用有效的index_oid以及大于零的attno参数来调用amproperty方法。prop是用于标识被测试性质的枚举值，而propname是原始的性质名称字符串。如果核心代码不能识别该性质名称，则prop为AMPROP_UNKNOWN。访问方法可以通过检查propname是否匹配（为与核心代码一致，使用pg_strcasecmp来匹配）来定义自定义性质名称；对于核心代码已知的名称，最好检查prop。 如果amproperty方法返回true则表示它已经确定了性质测试的结果：它必定会设置*res为要返回的布尔值，如果要返回 NULL 则设置*isnull为true（两个被引用的变量在调用之前要被初始化为false）。如果amproperty方法返回false则核心代码将会用其通常的逻辑来确定性质测试的结果。

支持排序操作符的访问方法应该实现AMPROP_DISTANCE_ORDERABLE性质测试，因为核心代码不知道如何做该测试并且会返回 NULL。如果有比打开索引并调用amcanreturn（这是核心代码的默认行为）更廉价的方法来做AMPROP_RETURNABLE测试，最好也实现它。默认行为应该对所有其他标准性质是符合要求的。

```
bool
amvalidate (Oid opclassoid);
```

只要访问方法能够，为指定的操作符类验证系统目录项。例如，这可能包括所有所需支持函数所提供的测试。如果该 opclass 不合法，amvalidate函数必须返回假。所存在的问题应由ereport消息报告。

当然，索引的目的是支持扫描那些匹配一个可索引WHERE情况的元组，常常也被称为限定词或扫描键。索引扫描的语义在下面的第 61.3 节有完整的描述。一个索引访问方法可以支持“普通”索引扫描、“位图”索引扫描或者两者。一个索引访问方法必须或可能提供的与扫描相关的函数是：

```
IndexScanDesc
ambeginscan (Relation indexRelation,
```

```
            int nkeys,
            int norderbys);
```

为一个索引扫描做准备。nkeys和norderbys参数说明要被用在扫描中的条件和排序操作符的数目，它们可以用于空间分配目的。注意扫描键的实际值还没有被提供。结果必须是一个palloc过的结构。由于实现的原因，索引访问方法必须通过调用RelationGetIndexScan()来创建这个结构。在大多数情况中，ambeginscan除了做这个调用和获取锁之外不会做很多工作，索引扫描启动中有趣的部分在amrescan中。

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

开始或者重新开始一个索引扫描，可能使用的是一个新的扫描键（要想使用之前传递的键重新开始，给keys 和/或orderbys传递 NULL）。请注意，使用的键或排序操作符的个数不能大于传递给ambeginscan的个数。实际上这个重新开始特性的使用场景是：在一个嵌套循环连接选取了一个新的 outer 元组时，因此需要一个新的键比较值，但扫描键结构仍然保持相同。

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

在给定扫描中取下一个元组，向给定方向移动（在索引中向前或者向后）。如果取到了元组，则返回 TRUE，如果取到匹配的元组，返回 FALSE。在 TRUE 的情况中，该元组的TID 被存储在scan结构中。请注意“成功”只 意味着索引包含一个匹配扫描键的项，并不意味该元组仍然在堆中存在， 或者是能够通过调用者的快照测试。在成功时，amgettuple也必须把scan->xs_recheck设 置成 TRUE 或者 FALSE。FALSE 意味着它确定索引项匹配扫描键。TRUE 意味着它并不确定，而且必须在取得堆元组之后对它重新检查扫描键表示的条件。 这条规定支持“有损的”索引操作符。注意重新检查仅仅对扫描条件扩展；一个部分索引谓语（如果有）从不被amgettuple调用者重新检查。

如果索引支持只用索引扫描（即amcanreturn对它返回 TRUE），则在成功时 AM 也必须检查scan->xs_want_itup，并且如果检查为真它必须返回索引项的原始被索引数据。该数据的返回形式可以是一个存储在scan->xs_itup中的IndexTuple指针外加元组描述符scan->xs_itupdesc，或者是一个存储在scan->xs_hitup中的HeapTuple指针外加元组描述符scan->xs_hitupdesc（在重构可能无法放在一个IndexTuple中的数据时，应该使用后一种格式）。不管是哪种形式，访问方法应该负责管理好指针引用的数据。至少在为扫描下一次调用amgettuple、amrescan或amendscan之前，该数据必须是完好的。

如果访问方法支持“普通”索引扫描，只需要提供amgettuple函数。如果不支持，它的IndexAmRoutine结构的amgettuple域必须被设置为 NULL。

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

在给定扫描中取所有元组并且把它们添加到调用者提供的TIDBitmap中（即，把元组 ID 的集合 OR 到已经存在于位图中的东西里面）。返回被取得的元组的数量（这可能仅仅是一个近似计数，例如一些 AM 不会去重）。在把元组 ID 插入到位图时，amgetbitmap可以指明对指定元组 ID 要求重新检查扫描条件。这与amgettuple的 xs_recheck输出参数类似。注意：在当前的实现中，这个特性的支持是和对位图本身有损存储的支持合并在一起的，并且调用者会对可重新检查的元组检查扫描条件和部分索引谓词（如果有）。但是，那不会总是真的。amgetbitmap和amgettuple不能被用

于同一个索引扫描；正如第 61.3 节所解释的，在使用amgetbitmap时也有其他的限制条件。

如果访问方法支持“bitmap”索引扫描，则仅需要提供amgetbitmap函数。如果不支持，它的IndexAmRoutine结构中的amgetbitmap域必须被设置为 NULL。

```
void
amendscan (IndexScanDesc scan);
```

结束扫描并释放资源。不应该释放scan结构本身，但访问方法内部使用的任何锁或者 pin 都应该被释放， 以及ambeginscan和其他扫描相关函数分配的任何其他内存。

```
void
ammarkpos (IndexScanDesc scan);
```

标记当前扫描位置。访问方法只需要支持每个扫描里面有一个被标记的扫描位置。

ammarkpos函数只有在访问方法支持有序扫描时才需要提供。如果不支持，则访问方法的IndexAmRoutine结构的ammarkpos域可以设置为 NULL。

```
void
amrestrpos (IndexScanDesc scan);
```

把扫描恢复到最近标记的位置。

amrestrpos函数只有在访问方法支持有序扫描时才需要提供。如果不支持，则访问方法的IndexAmRoutine结构的amrestrpos域可以设置为 NULL。

除了支持普通的索引扫描之外，某些类型的索引可能希望支持并行索引扫描，这种方式允许多个后端合作来执行一次索引扫描。索引访问方法应该安排好各种事情，这样每个参与合作的进程才能返回原本会由普通非并行索引扫描执行得到的元组的一个子集，但是得到的那些子集的并集应该等于普通非并行索引扫描得到的元组集合。此外，虽然不需要并行扫描返回的元组有任何全局顺序，但每个参与合作的后端中返回的元组子集的顺序必须匹配所要求的顺序。必须实现下列函数才能支持并行索引扫描：

```
Size
amestimateparallelscan (void);
```

估算并且返回访问方法执行一次并行扫描所需要的动态共享内存的字节数（这个数字是对ParallelIndexScanDescData中访问方法无关的数据所需空间量的补充而不是替代）。

对于不支持并行扫描或者额外存储需求的的字节数为零的访问方法，无需实现这个函数。

```
void
aminitparallelscan (void *target);
```

在一次并行扫描的开头将调用这个函数来初始化动态共享内存。target将指向一段动态共享内存空间，其大小至少为之前amestimateparallelscan返回的字节数，并且这个函数可以使用这部分空间来存放它希望存放的任何数据。

对于不支持并行扫描或者不要求初始化共享内存空间的情况，无需实现这个函数。

```
void
amparallelrescan (IndexScanDesc scan);
```

如果实现了这个函数，当并行索引扫描必须被重启时，将会调用这个函数。它应该重置由aminitparallelscan建立的任何共享状态，这样扫描将会被重头重新开始。

61.3. 索引扫描

在一个索引扫描中，索引访问方法负责提供它拿到的匹配扫描键的所有元组的TID。访问方法不会涉及从索引的父表中实际取得那些元组，也不会涉及判断它们是否通过了扫描的时间条件测试或者是其它条件。

一个扫描键是一个WHERE子句的内部表示，WHERE子句的形式是index_key operator constant，其中索引键字索引中的一个列，而操作符是和该索引列相关联的操作符族的一个成员。一个索引扫描拥有零个或者多个扫描键，它们是隐式 AND 关系 — 返回的元组被认为满足所有列出的条件。

对于一个特定查询，访问方法可能报告索引是有损的或者要求重新检查。这就暗示着该索引扫 描会返回所有通过扫描键的项，外加上一些可能没通过扫描键的项。核心系统的索引扫描机制然后就会再次在堆元组上应用索引条件来验证它是否真地应该被选择。如果没有指定重新检查选项，索引扫描必须返回准确的匹配项集合。

请注意，确保找到所有（只有）通过所有给定扫描键的条目的工作完全由访问方法负责。还有，核心系统将只是简单地 放过所有匹配扫描键和操作符族的WHERE子句，而不会做任何语义分析来判断它们是否冗余或者矛盾。例如，给定WHERE x > 4 AND x > 14（其中x是一个 B-树 索引列，它被留给 B-树 amrescan函数来发现第一个扫描键是冗余并且可以被丢弃。amrescan期间需要的预处理的范围将取决于索引访问方法需要什么来把扫描键缩减为一种“正规化的”形式。

一些访问方法按照一个良定义的顺序来返回索引项，其他的则不会。实际上一个访问方法可以有两种不同的方式支持排序输出：

- 总是按数据的自然序返回项的访问方法应该设置amcanorder为真。当前，这样的访问方法必须对它们的等值和排序操作符使用b-tree兼容的策略号。
- 支持排序操作符的访问方法应该设置amcanorderbyop为真。这表示索引有能力按照满足ORDER BY index_key operator constant的一种顺序返回项。如前所述，这种形式的扫描修饰符可以被传递给amrescan。

amgettuple函数有一个direction参数，它可以是 ForwardScanDirection（正常情况）或者BackwardScanDirection。如果amrescan之后的第一次调用指定了BackwardScanDirection，那么匹配条件的索引项集合是从后向前扫描的，而 不是通常的从前向后扫描，因此amgettuple必须返回索引中最后一个匹配元组，而不是通常情况下的第一个（这只对设置了amcanorder为真访问方法发生）。在第一次调用后，amgettuple必须被准备好从最近被返回项的位置按照任何一种方向推进扫描（但是如果amcanbackward为假，所有后续调用将使用第一次相同的方向）。

支持排序扫描的访问方法必须支持在扫描里“标记”一个位置并且随后返回到这个标记过的位置。同一个位置可能会被重复多次还原。但是，每个扫描中只有一个位置需要被记住；一个新的ammarkpos调用将重写之前标记的位置。一个不支持排序扫描的访问方法无需在IndexAmRoutine中提供ammarkpos和amrestrpos函数，把这些指针设置为 NULL 即可。

扫描位置和标记位置（如果存在)）都必须在面对索引中的并发插入和删除时保持一致性。如果一个新插入的项并未被一个扫描返回（如果该扫描开始的时候该项已经存在，该扫描将已经找到该项），或者说扫描通过重新扫描或者反向扫描返回这样一个项（即使它第一次没有返回这样一个项），这些情况都是可以接受的。类似的还有，一个并发的删除可能或不可能被反映在一个扫描的结果中。重要的是，插入或者删除不会导致扫描错过或者多次返回本身不是被插入或者删除的项。

如果索引存储原始被索引的数据值（并且不是它们的某种有损表示），它可用来支持只用索引的扫描，着这种扫描中索引返回的就是实际的数据而不只是堆元组的 TID。这只有在可见性映射显示该 TID 位于一个全部可见的页面时才能避免 I/O；否则必须访问堆元组来检查 MVCC 可见性。但是这就不用访问方法操心了。

除了使用amgettuple，一个索引扫描可以通过amgetbitmap在一次调用中取得所有元组来完成。这样做可能会比amgettuple有显著的效率提升，因为它可以避免在访问方法内的加锁/解锁循环。原则上amgetbitmap应该和重复调用amgettuple的效果相同，不过我们强加了一些限制来简化这件事。首先，amgetbitmap一次返回所有元组并且标记并且不支持标记或恢复扫描位置。第二，在一个位图中返回的元组没有任何指定的顺序，这也是为什么amgetbitmap没有一个direction参数的原因（排序操作符也将永远不会提供给这种扫描）。还有，对于使用amgetbitmap的只用索引扫描没有规定，因为没有办法返回索引元组的内容。最后，如第 61.4 节所说的，amgetbitmap不保证被返回元组上的任何锁。

注意如果访问方法的内部实现不适合一个 API 或其他 API，允许一个访问方法只实现amgetbitmap而不实现amgettuple，或者反过来。

61.4. 索引锁定考虑

索引访问方法必须支持多个进程对索引的并发更新。在索引扫描期间，核心PostgreSQL系统在索引上获取 AccessShareLock，并且在更新索引时（包括普通VACUUM）获取RowExclusiveLock。因为这些锁类型不会冲突，所以访问方法负责处理它可能需要的任何细粒度锁。把索引作为一个整体的排他锁只会在索引创建、删除或REINDEX时被使用。

创建一个支持并发更新的索引类型通常要求对所需的行为进行广泛并且细致的分析。对于b-tree 和哈希索引类型，你可以阅读src/backend/access/nbtree/README和src/backend/access/hash/README中的设计决策。

除了索引自己内部的一致性要求之外，并发更新带来了一些父表（堆）和索引之 间的一致性问题。因为 PostgreSQL是把堆的访问和 更新与索引的访问和更新分开的，所以存在一些窗口期，在其间索引可能会与堆不一致。我们用下面的规则处理这样的问题：

- 一个新堆项在其索引项之前被制作（因此并发的索引扫描很可能看不到堆项。这么做应该是可以的，因为索引的读取者对未提交的行不感兴趣。见第 61.5 节。
- 当一个堆项要被删除（通过VACUUM）时，它的所有索引项都必须首先删除。
- 一次索引扫描必须在保存有amgettuple最后返回项的索引页面上维护一个 pin， 并且ambulkdelete不能从页面中删除被其他后端加 pin 的项。下面会解释需要这条规则的原因。

没有第三条规则，那么一个索引读取者是可以在一条索引项被VACUUM删除之前看到它的，并且然后在VACUUM删除它之后找到其对应的堆项。如果读取者到达该项时，该项编号仍然没有被使用，那么这种 情况不会导致严重的问题，因为空的项槽位会被heap_fetch()忽略。 但是如果第三个后端已经为其它什么东西重用了这个项槽位又会怎样？在使用MVCC 兼容的快照时，那么就不会有问题，因为槽位的新占据者太新了以至于无法通过快照测试。但是，对于非 MVCC 兼容的快照（例如 SnapshotAny），那么就有可能接受并返回一个实际上并不匹配扫描键的行。可以通过要求扫描键在所有情况下都在堆行上重新检查来避免这种情况，但是这种方法开销太大了。取而 代之的是，通过在索引页面上使用一个pin 作为一个代理来表示，读取者可能还处于从索引项到匹配的堆项的“过程中”。用ambulkdelete来操作这样一个 pin 上的块确保VACUUM无法在读取者完成之前删除堆项。这种解决方案在运行时只有一点开销，而只是在真有一个冲突的非常罕见情况下才导致阻塞开销。

这个解决方法要求索引扫描是“同步的”：我们不得不在扫描完对应的索引项之后马上去取每个堆元组。这样的方案开销比较大，原因有多个。而一个“异步的”扫描可以先从索引里收集很多 TID ，并且在稍后的某个时间只访问堆元组，这样要求更少的索引锁定负荷并且能够允许一种更高效的堆访问模式。但是按照上面的分析，在非 MVCC 兼容的快照上我们必须使用同步方法，而异步扫描则适合于使用 MVCC 快照的查询。

在一个amgetbitmap索引扫描中，访问方法不会在任何被返回的元组上保持一个索引 pin。因此只有把这种扫描与 MVCC 兼容的快照一起使用才是安全的。

当ampredlocks标志没有被设置时，在一个可序列化事务中使用该索引访问方法的任何扫描将在整个索引上获取一个非阻塞的谓词锁。这将和一个并发可序列化事务中项索引中插入任

何元组发生读-写冲突。如果在一组并发可序列化事务之间检测到特定模式的读-写冲突，其中一个事务可能会被取消来保护数据完整性。当该标志被设置，它表示该索引访问方法实现了细粒度的谓词锁，这将有望缩减这种事务取消的频率。

61.5. 索引唯一性检查

PostgreSQL使用唯一索引来强制 SQL 唯一性约束，唯一 索引实际上是不允许多个项有相同键的索引。一个支持这个特性的访问方法要 设置amcanunique为真（目前，只有 b-tree 支持它）。在强制唯一性时不会考虑INCLUDE子句中列出的列。

因为 MVCC，必须允许重复的项在物理上存在于索引之中：这些项可能指向某个单一逻辑行的后继版本。实际想强制的行为是，任何 MVCC 快照都不能包含两个具有相同索引键的行。在向一个唯一索引中插入一个新行时需要被检查的情况可分解成：

- 如果一个有冲突的合法行已被当前事务删除，这是可以的（特别是因为一个 UPDATE 总是在插入新版本之前删除旧版本，这样就允许一个行上的UPDATE 不改变键）。
- 如果一个有冲突的行已经被还未提交的事务插入，那么准备插入的事务必须等待看看前面那个事务是否提交。如果它回滚额不会有冲突。如果它提交并且没有删除存在冲突的行，则有一个唯一性违背（实际上我们只是等待那个其他事务结束，然后在全部事务里重做可见性检查）。
- 类似的，如果一个有冲突的有效行被一个准备提交的事务删除，那么另外一个准备插入的事务必须等待该事务提交或者退出，然后重做测试。

此外，在根据上述规则报告唯一性违背之前，访问方法必须重新检查刚被插入的行的存活性。如果已经因为事务的提交而死亡，那么不应当报告任何违背（这种情况不可能出现在插入在同一事务中创建的行的普通场景中。但是在CREATE UNIQUE INDEX CONCURRENTLY的过程中是可能发生的）。

要求索引访问方法自己应用这些测试，这就意味着它必须到达堆来查看那些根据 索引内容有重复键的任意行的提交状态。这无疑是丑陋并且非模块化的，但是这样可以节约重复的工作：如果我们进行一次独立的探测，那么查找一个冲突行的索引查找本质上将在查找插入新行索引项位置时被重复。此外，没有很明显的方法来避免竞争情况，除非冲突检查是插入新索引项动作的一部分。

如果唯一约束是可延迟的，就存在额外的复杂性：我们需要能够为一个新行插入一个索引项，但是推迟任何唯一性违背错误直到语句结束或者更晚。为了避免对索引不必要的重复搜索，索引访问方法应该在初始插入过程中做一次初步的唯一性检查。如果显示绝对不会有冲突的活元组，就可以完成。否则，我们计划一次重新检查，它将在强制约束的时候发生。在重新检查时，如果具有相同键的被插入元组和某个其他元组都活着，则必须报告错误（注意为了这个目的，“活着”实际意味着“在索引项的 HOT 链上的任何元组都活着”）。要实现这一点，需要给 aminsert 传递一个 checkUnique 参数，其中包含下列值之一：

- UNIQUE_CHECK_NO表明不需要做唯一性检测（这不是一个唯一索引）。
- 如上所述，UNIQUE_CHECK_YES表明有一个不可延迟的唯 一索引，并且必须立即做唯一性检测。
- UNIQUE_CHECK_PARTIAL表明唯一性约束是可延迟的。PostgreSQL将会用这个模式来插入每一行的索引项。访问方法必须允许重复的项进入索引，并且通过从 aminsert返回 FALSE 来报告任何可能的重复。对于返回 FALSE 的每一行，将计划一个延迟的重新检查。

  访问方法必须能够标识任何可能违反唯一约束的行，但是对它来说假阳性报告不是错误。这样就允许检查不用等到其他事务都结束；这里报告的冲突不会被当做错误来看待，并且随后将会被重新检查，而到那时它们可能不再是冲突了。
- UNIQUE_CHECK_EXISTING表明这是一个行的延迟重新检查，该行被报告为一个潜在的唯一性违背。尽管这会通过调用aminsert来实现，在这种情况下这个访问方法不能插入一个新

索引项。该索引项已经存在。 当然，访问方法必须检查是否有另一个活着的索引项。如果有，并且如果目标行也仍然存活，那么报告错误。

我们推荐，在一个UNIQUE_CHECK_EXISTING调用中，访问方法进一步验证目标行真的在索引中有一个现有的项，并且如果不是这样就报错。这是个好主意，因为被传到aminsert的索引元组值将已经被重新计算过。如果索引定义涉及不是真正不变的函数，我们可能正在检查索引的错误区域。对重新检查中找到的目标行的检查会验证我们正在扫描之前被用于原始插入的同一元组值。

61.6. 索引开销估计函数

amcostestimate函数被给定描述一个可能的索引扫描的信息，包括决定在索引中使用的 WHERE 和 ORDER BY 子句的列表。它必须返回访问该索引的开销估计以及 WHERE 子句的选择度（也就是说，在索引扫描期间将检索的行在父表中所占据的比例）。对于简单情况，几乎开销估计器的所有工作都可以通过调用优化器中的标准过程完成；有amcostestimate函数的目的是允许索引访问方法提供和索引类型相关的知识，这种情况下可以改进标准的估计。

每个amcostestimate函数的签名必须是：

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

前三个参数是输入参数：

root

规划器的有关正在被处理的查询的信息。

path

被考虑的索引访问路径。其中除了开销和选择度值之外的域都有效。

loop_count

应该被开销估计所考虑的索引扫描重复次数。当考虑用在一个嵌套循环连接中的参数化扫描时，这个参数通常会大于 1 。注意代价估计应该仍然是对于一次扫描的，一个更大的loop_count意味着可能在多次扫描间允许一些缓冲效果比较合适。

后四个参数是传引用的输出参数：

*indexStartupCost

设置为索引启动处理的开销。

*indexTotalCost

设置为索引处理的总开销。

*indexSelectivity

设置为索引的选择度。

*indexCorrelation

设置为索引扫描顺序和下层的表的顺序之间的相关性。

*indexPages

设置为索引叶子页的数量

请注意开销估计函数必须用 C 编写，而不能用 SQL 或者任何可用的过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问开销应该采用被src/backend/optimizer/path/costsize.c使用的参数进行计算：一次顺序磁盘块获取的开销是seq_page_cost、一次非顺序获取的开销是random_page_cost并且处理一个索引行的开销通常应该是cpu_index_tuple_cost。另外，在索引处理期间（尤其是索引条件本身的计算）调用的任何比较操作符都会耗费cpu_operator_cost倍数的开销。

访问开销应该包括所有与扫描索引本身相关的磁盘和 CPU 开销，但是不包括检索或者处理被索引标识出来的父表行的开销。

“启动开销”是整个扫描开销中的一部分：在能够开始取第一行之前必须花掉的开销。对于大多数索引这个开销是零，但 是那些启动开销很大的索引类型不会把它设置为零。

indexSelectivity应该设置成在索引扫描期间，父表行被检索的估计比例。在一个有损查询的情况下，这个值通常高于实际通过 给定查询条件的行的比例。

indexCorrelation应该被设置成索引顺序和表顺序之间的相关性（范围从 -1.0 到 1.0）。这个数值被用于调整从父表中取出行的开销估计。

indexPages应该被设置为叶子页面的数量。它会被用来估算并行索引扫描用到的工作者的数目。

当loop_count大于一时，返回的数应该是该索引任何一次扫描的平均期望值。

开销估计

一个典型的开销估计器会像下面这样进行处理：

1. 基于给出的条件情况，估计并返回父表行将被访问的比例。如果缺乏索引类型相关的知识，那么使用标准的优化器函数clauselist_selectivity()：

```
*indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                           path->indexinfo->rel->relid,
                                           JOIN_INNER, NULL);
```

2. 估计在扫描过程中将被访问的索引行数。对于许多索引类型，这个等于indexSelectivity乘以索引中的行数，但是可能更多（请注意，页面和行中的索引尺寸从path->indexinfo结构中获得）。

3. 估计在扫描中将检索的索引页面数量。这个可能就是indexSelectivity乘以索引的总页面数。

4. 计算索引访问开销。一个通用的估计器可能会：

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they cost seq_page_cost each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
```

```
 */
cost_qual_eval(&index_qual_cost, path->indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

不过，上面没有考虑重复索引扫描间的索引读分期补偿（amortization）。

5. 估计索引的相关性。对于一个简单的在单列有序索引，这个值可以从 pg_statistic 中检索。如果相关性是未知，那么保守的估计是零（没有相关性）。

开销估计器函数的例子可以在src/backend/utils/adt/selfuncs.c中找到。

第 62 章 通用WAL 记录

虽然所有内建的被 WAL 记录的模块都有它们自己的 WAL 记录类型，系统中也还是有一种通用 WAL 记录类型，它以一种通用的方式描述了对页面的改变。这对于提供自定义访问方法的扩展有用，因为这类扩展无法注册自己的 WAL 重做例程。

构建通用 WAL 记录的 API 定义在access/generic_xlog.h中，实现在access/transam/generic_xlog.c中。

要使用通用 WAL 记录工具执行一次被 WAL 记录的数据更新，要遵循这些步骤：

1. state = GenericXLogStart(relation) — 为给定的关系构建一个通用 WAL 记录。

2. page = GenericXLogRegisterBuffer(state, buffer, flags) — 注册一个要在当前的通用 WAL 记录中修改的缓冲区。这个函数会返回一个指针指向该缓冲区页面的一份临时拷贝，修改将会在该拷贝上进行（不要直接修改该缓冲区的内容）。第三个参数是适用于该操作的标志的位掩码。当前这类标志只有GENERIC_XLOG_FULL_IMAGE，它表示在 WAL 记录中应该包括一个完整页面镜像而不是增量更新。如果是新页面或者页面已经被完全重写，通常会设置这个标志。如果被 WAL 记录的动作需要修改多个页面，可以反复调用GenericXLogRegisterBuffer。

3. 对包含在上一步中的页面镜像应用修改。

4. GenericXLogFinish(state) — 将更改应用到缓冲区并且发出通用 WAL 记录。

在上述步骤之间都可以调用GenericXLogAbort(state)取消 WAL 记录构造。这会丢弃所有对于页面镜像拷贝的更改。

在使用通用 WAL 记录功能时请注意以下几点：

- 不允许直接修改缓冲区！所有的修改必须在GenericXLogRegisterBuffer()取得的拷贝上完成。换句话说，制造通用 WAL 记录的代码不能为自己调用BufferGetPage()。不过，在合适的时间对缓冲区进行 pin/unpin 以及加锁/解锁仍然是调用者的责任。从GenericXLogRegisterBuffer()之前直到GenericXLogFinish()之后，每个目标上必须保持排他锁。

- 可以自由地混合注册缓冲区（步骤 2）和页面镜像修改（步骤 3），即两个步骤可以以任何顺序重复。记住注册缓冲区的顺序应该和重放时对它们加锁的顺序相同。

- 一个通用 WAL 记录能注册的缓冲区最大数量是MAX_GENERIC_XLOG_PAGES。如果超出这个限制将会抛出一个错误。

- 通用 WAL 假定要被修改的页面具有标准布局，特别是在pd_lower和pd_upper之间没有有用的数据。

- 由于正在修改缓冲区页面的拷贝，GenericXLogStart()不会开始临界区。因此可以在GenericXLogStart()之间GenericXLogFinish()安全地进行内存分配、抛出错误等。唯一真正的临界区存在于GenericXLogFinish()内。还有，不需要担心在错误退出期间对GenericXLogAbort()的调用。

- GenericXLogFinish()会负责标记缓冲区为脏并且设置它们的 LSN。你不需要显式地做这些工作。

- 对于不做日志的关系，所有的事情都一样，不过不会发出实际的 WAL 记录。因此，对于不做日志的关系你通常不需要做任何显式的检查。

- 通用 WAL 重做函数将按照注册缓冲区的顺序对它们获得排他锁。在重做所有更改后，这些锁将按照同样的顺序被释放。

- 如果对一个已注册的缓冲区没有指定GENERIC_XLOG_FULL_IMAGE，通用 WAL 记录包含了新旧页面镜像之间的不同。这个不同是以逐字节比较的方式形成的。对于在页面内移动数据的情况来说这种方式不是很紧凑，未来可能会有改进。

第 63 章 B-树索引

63.1. 简介

PostgreSQL包括了对标准btree（多路平衡树）索引数据结构的一个实现。任何能够被排序为良定义线性顺序的数据结构都可以用一个btree来索引。唯一的限制是一个索引项不能超过大约三分之一个页面（如果适用，可以是TOAST压缩后的大小）。

因为每一种btree操作符类都会在其数据类型上施加一种排序顺序，btree的操作符类（或者实际上是操作符族）已经被用作PostgreSQL对排序语义的一般表达和理解。因此，它们需要一些支持btree索引之外的特性，并且这个系统的一些部分与利用它们的btree访问方法有较大的不同。

63.2. B-树操作符类的行为

如表 38.2中所示，一个btree操作符类必须提供五种比较操作符：<、<=、=、>=以及>。有人可能会想<>应该也是操作符类的一部分，但不是这样，因为几乎从不会在索引搜索中使用有<>的WHERE子句（出于某种原因，规划器会认为<>与一个btree操作符类相关，但它是通过=操作符的逆操作符链接来找到这个操作符，而不是从pg_amop中查找）。

当一些数据类型共享近乎相同的排序语义时，它们的操作符类可以被组合成一个操作符族。这样做是有好处的，因为这样就允许规划器对跨类型比较进行推演。在操作符族中的每一种操作符类对其输入数据类型应该包含单一类型的操作符（及其相关的支持函数），而跨类型比较操作符及其支持函数则“松散”地放在操作符族中。推荐在操作符族中包括一套完整的跨类型操作符，这样能确保规划器可以表达它通过传递性推演出的任何比较条件。

这里有一些btree操作符类必须满足的基本假设：

- 一个=操作符必须是一种等值关系。也就是说，对于该数据类型的所有非空值A、B、C：
  - A = A为真（自反律）
  - 如果A = B，则有B = A（对称律）
  - 如果A = B并且B = C，则有A = C（传递律）
- 一个<操作符必须是一种强排序关系。也就是说，对于所有的非空值A、B、C：
  - A < A为假（非自反律）
  - 如果A < B以及B < C，则有A < C（传递律）
- 此外，该排序是完全的。也就是说，对于所有非空值A、B、C：
  - A < B、A = B和B < A之中恰好有一个为真（三分律）
    （三分律无疑证明了比较支持函数定义的正确性。）

其他三种操作符可以以显而易见的方式用=和<来定义，并且必须和它们的行为保持一致。

对于一个支持多种数据类型的操作符族来说，当A、B、C取自该族中任意数据类型时，上述定律都必须保持。传递律是最难以保证的，因为在跨类型的情况中，传递律说明两种或者三种不同的操作符的行为是一致的。举个例子，把float8和numeric放在同一个操作符族中是行不通的，至少在当前的语义（为了和一个float8比较，numeric值会被转换成float8）下不行。因为float8有限的精度，这意味着不同的numeric值将被认为等于同一个float8值，因此传递律将被破坏。

对于多数据类型操作符族的另一个要求是，其中包括的定义在数据类型之间的任何隐式或者二进制强制造型不能改变相关的排序顺序。

为何一个btree索引要求这些定律在单一数据类型中必须保持的原因应该相对比较清楚：没有这些定律就不存在用于安排键的顺序。此外，使用不同数据类型键的比较的索引搜索也要求比较操作在两种数据类型之间表现得稳定。btree索引机制本身并不严格要求在一个操作符族中扩展到三种或者更多种数据类型，但是规划器依赖于这种扩展来实现其优化的目的。

63.3. B-树支持函数

如表 38.8中所示，btree定义了一种必需的和两种可选的支持函数。

对于btree操作符族为其提供了比较操作符的每一种数据类型组合，操作符族必须提供一个比较支持函数，在pg_amproc中注册：支持函数编号为1，amproclefttype/amprocrighttype等于比较的左右数据类型（即匹配的操作符注册在pg_amop中的数据类型）。比较函数必须接收两个非空值A和B并且返回一个int32值，返回值在A < B、A = B以及A > B时分别为< 0、0和> 0。不允许空值结果：该数据类型的所有值必须是可比较的。例子请见src/backend/access/nbtree/nbtcompare.c。

如果被比较的值是一种可排序的数据类型，合适的排序规则OID将使用标准的PG_GET_COLLATION()机制被传递给比较支持函数。

可选地，btree操作符族可以提供排序支持函数，它们以支持函数编号2注册。这些函数允许以一种比单纯调用比较支持函数更加高效的方式实现排序比较。涉及的API在src/include/utils/sortsupport.h中定义。

可选地，btree操作符族可以提供in_range支持函数，它们以支持函数编号3注册。在btree索引操作期间不会用到这些函数，它们扩展了操作符族的语义，这样就能支持包含RANGE offset PRECEDING以及RANGE offset FOLLOWING窗口帧界类型（见第 4.2.8 节的窗口子句。归根到底，这些函数所提供的额外信息是如何以一种与该操作符族的数据排序相兼容的方式加上或者减去一个offset值。

一个in_range函数必须具有这样的签名

```
in_range(val type1, base type1, offset type2, sub bool, less bool)
returns bool
```

val和base必须是同一种类型，该类型也是操作符族所支持的类型之一（即它提供排序的一种类型）。不过，offset可以是一种不同的类型，该类型有可能不被该操作符族所支持。例如内建的time_ops族提供了一个in_range函数，其offset是类型interval。一个操作符族可以为其所支持的任意类型提供in_range函数以及一个或者更多种offset类型。每一个in_range函数在进入到pg_amproc时，需要有amproclefttype等于type1以及amprocrighttype等于type2。

in_range函数的本质语义取决于两个布尔标志参数。它应该将base和offset相加或者相减，然后用val与其结果比较：

- 如果!sub并且!less，则返回val >= (base + offset)
- 如果!sub并且less，则返回val <= (base + offset)
- 如果sub并且!less，则返回val >= (base - offset)
- 如果sub并且less，则返回val <= (base - offset)

在这样做之前，该函数应该检查offset的符号：如果它小于零，则抛出错误ERRCODE_INVALID_PRECEDING_OR_FOLLOWING_SIZE (22013)外加“invalid preceding or following size in window function”这样的错误文本（这是SQL标准所要求的，不过非标准操作符族可能会选择忽视这一限制，因为似乎其语义必要性很小）。这种要求被委托给了in_range函数，这样核心代码不需要理解对一种特定数据类型“less than zero”表示什么。

一个额外的期望是，如果可行，in_range函数应当在base + offset或者base - offset溢出时避免抛出错误。即便值超过了该数据类型的范围，也可以确定正确的比较结果。注意，如果数据类型包括诸如“infinity”或者“NaN”之类的概念，就需要额外的注意确保in_range的结果符合该操作符族的正常排序顺序。

in_range函数的结果必须与操作符族施加的排序顺序保持一致。准确的来说，给定任意固定的offset值以及sub值，那么：

- 如果带有less = true的in_range对某个val1和base为真，则它必须对每一个有相同base的val2 <= val1为真。
- 如果带有less = true的in_range对某个val1和base为假，则它必须对每一个有相同base的val2 >= val1为假。
- 如果带有less = true的in_range对于某些val和base1为真,那么它对于每一个有相同val的base2 >= base1也必须为真。
- 如果带有less = true的in_range对于某些val和base1为假, name它对于每一个有相同val的base2 <= base1也必须为假。

当less = false时，类似的具有相逆条件的语句成立。

如果被排序的类型（type1）是可排序的，合适的排序规则OID将使用标准的PG_GET_COLLATION()机制被传递给in_range函数。

in_range函数不需要处理NULL输入，并且通常将被标记为strict。

63.4. 实现

在src/backend/access/nbtree/README中可以找到对于btree索引实现的介绍。

第 64 章 GiST 索引

64.1. 简介

GiST表示通用搜索树。它是一种平衡的树结构的访问方法，它作为一种模板可用来实现任意索引模式。B 树、R 树和很多其他索引模式都可以在GiST中实现。

GiST的一个优势是它允许自定义数据类型的领域专家使用合适的访问方法开发自定义数据类型，而不是让数据库专家来开发。

这里的一些信息是来自加州大学伯克利分校的 GiST 索引项目网站[1]和 Marcel Kornacker 的学位论文Access Methods for Next-Generation Database Systems[2]。PostgreSQL中的GiST实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，在他们的网站[3]上有更多信息。

64.2. 内建操作符类

PostgreSQL核心发布中包括如表 64.1中所示的GiST操作符类（附录 F中描述的一些可选模块提供了额外的GiST操作符类）。

表 64.1. 内建GiST操作符类

名称	索引数据类型	可索引操作符	排序操作符
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
range_ops	任何范围类型	&& &> &< >> << <@ -|- = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

由于历史原因，inet_ops操作符类不是类型inet和cidr的默认操作符类。要使用它，需要在CREATE INDEX中指明操作符类的名称，例如

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

[1] http://gist.cs.berkeley.edu/
[2] http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz
[3] http://www.sai.msu.su/~megera/postgres/gist/

64.3. 可扩展性

在传统上，实现一种新的索引访问方法意味着很多困难的工作。开发者必须要理解数据库的内部工作，例如锁管理器和预写式日志。GiST接口有一个高层的抽象，要求访问方法实现者只实现要被访问的数据类型的语义。GiST层本身会处理并发、日志和对树结构的搜索。

这种可扩展性不应该与其他标准搜索树对于它们所处理的数据上的可扩展性混淆。例如，PostgreSQL支持可扩展的 B 树和哈希索引。也就是说你可以用PostgreSQL在任何你想要的数据类型上构建一个 B 树或哈希。但是 B 树只支持范围谓词（<、=、>），而哈希索引支持等值查询。

这样如果你用一个PostgreSQL的 B 树索引一个图像集合，你只能发出例如“imagex 等于 imagey 吗”、“imagex 小于 imagey 吗”以及“imagex 大于 imagey 吗”的查询。取决于你如何在这种上下文中定义“等于”、“小于”和“大于”，这可能会有用。但是，通过使用一个基于GiST的索引，你可以创建提问领域相关问题的方法，可能是“找所有马的图片”或者“找所有曝光过度的图片”。

建立一个GiST访问方法并让其运行的所有工作是实现几个用户定义的方法，它们定义了树中键的行为。当然这些方法必须相当特别来支持特别的查询，但是对于所有标准查询（B 树、R 树等）它们相对直接。简而言之，GiST在可扩展性之上结合了通用型、代码重用和一个干净的接口。

一个用于GiST的索引操作符类必须提供五种方法，并且还有四种可选的方法。索引的正确性由正确实现的same、consistent和union方法保证，而索引的效率（尺寸和速度）将依赖于penalty和picksplit方法。两种可选的方法是compress和decompress，它们允许一个索引能对内部数据使用一种不同于被其索引的数据的类型。叶子是被索引的数据类型，而其他树结点可以是任何 C 结构（但是你仍必须遵循PostgreSQL的数据类型规则，见用于可变尺寸数据的varlena）。如果树的内部数据类型在 SQL 层上存在，可以使用CREATE OPERATOR CLASS命令的STORAGE选项。可选的第八个方法是distance，如果操作符类希望支持有序扫描（最近邻搜索）就需要它。如果该操作符希望支持只用索引的扫描，则需要可选的第九个方法fetch，但compress方法被省略时不需要。

consistent

给定一个索引项p和一个查询值q，这个函数决定该索引项是否与该查询“一致”，就是说：是否该索引项表示的行使得谓词“indexed_columnindexable_operator q”为真？对于一个叶子索引项，这等效于测试索引条件；而对于一个内部树结点，这会决定是否需要扫描由该树结点表示的索引子树。当结果为true时，还必须返回一个recheck标志。这指示该谓词一定为真或者只是可能为真。如果recheck = false那么该索引已经完全测试过该谓词条件，而如果recheck = true则该行只是一个候选匹配。在那种情况下，系统将根据实际的行值自动评估indexable_operator来看它是否真的是一个匹配。这允许GiST同时支持有损和无损的索引结构。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid,
 internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
```

```
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * 根据策略、键和查询确定返回值。
     *
     * 使用 GIST_LEAF(entry) 可以了解当前函数是在索引树的哪里被调用，
     * 这在支持例如 = 操作符时很方便（可以在非叶子节点中检查非空 union()
     * 以及在叶子节点中检查等值）。
     */

    *recheck = true;        /* 如果检查是准确的则返回 false */

    PG_RETURN_BOOL(retval);
}
```

这里，key是该索引中的一个元素而query是在该索引中查找的值。StrategyNumber参数指示在你的操作符类中哪个操作符被应用 — 它匹配CREATE OPERATOR CLASS命令中的操作符编号之一。

取决于在操作符类中包含着哪些操作符，query的数据类型可能随着操作符而变化，因为它可能是该操作符右手边的任何类型，而这种类型可能和出现在其左手边的被索引数据类型不同（上面的代码框架假定只有一种类型；如果不是这样，取query参数值的方式可能必须取决于操作符）。我们推荐让consistent函数的 SQL 声明对query参数使用操作符类的被索引数据类型，即便实际类型可能是其他依赖于操作符的类型也是如此。

union

这个方法联合树中的信息。给定一组项，这个函数产生一个新的索引项，它表示所有给定的项。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;
```

```
        numranges = entryvec->n;
        tmp = DatumGetDataType(ent[0].key);
        out = tmp;

        if (numranges == 1)
        {
            out = data_type_deep_copy(tmp);

            PG_RETURN_DATA_TYPE_P(out);
        }

        for (i = 1; i < numranges; i++)
        {
            old = out;
            tmp = DatumGetDataType(ent[i].key);
            out = my_union_implementation(out, tmp);
        }

        PG_RETURN_DATA_TYPE_P(out);
    }
```

如你所见，在这个框架中我们处理一种数据类型union(X, Y, Z) = union(union(X, Y), Z)。通过在这个GiST支持方法中实现正确的联合算法，支持不是这种情况的数据类型足够简单。

union函数的结果必须是该索引的存储类型的一个值，它可能与被索引列的类型不同，也可能相同。union函数应该返回一个指针指向新palloc()的内存。不能照原样返回输入值，即使没有类型改变也不能。

如上所示，union函数的第一个internal参数实际上是一个GistEntryVector指针。第二个参数是一个指向整数变量的指针，它可以被忽略（过去要求union函数将其结果值的尺寸存储在这个变量中，但现在这已不再必要）。

compress

把数据项转换成适合于一个索引页面中物理存储的格式。如果compress方法被省略，数据项会被不加修改地存储在索引中。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* 用一个压缩版本替换 entry->key */
```

```
            compressed_data_type *compressed_data =
     palloc(sizeof(compressed_data_type));

            /* 从 entry->key ... 填充 *compressed_data */

            retval = palloc(sizeof(GISTENTRY));
            gistentryinit(*retval, PointerGetDatum(compressed_data),
                          entry->rel, entry->page, entry->offset, FALSE);
        }
        else
        {
            /* 通常我们不需要对非叶子项做任何事情 */
            retval = entry;
        }

        PG_RETURN_POINTER(retval);
    }
```

当然，为了压缩你的叶结点，你必须把compressed_data_type改编成你正在转换到的指定类型。

decompress

将一个数据项的存储表达转换成该操作符类中其他GiST方法能够操纵的格式。如果decompress方法被省略，则假设其他GiST方法能够直接在存储的数据格式上工作（decompress不一定要是compress方法的逆操作，特别是如果compress是有损的，那么decompress是不可能准确地重构原始数据的。decompress也不一定与fetch等效，因为其他GiST方法可能不需要数据的完整重构）。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

上述框架适合于不需要解压的情况（但是，将该方法一并省去当然更加容易，并且在这种情况下推荐这样做）。

penalty

返回一个值，它指示在树的一个特定分支插入新项的“代价”。项将被插入到树中具有最小penalty的路径中。penalty返回的值应该为非负。如果一个赋值被返回，它将被当作零来处理。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
```

```
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- in some cases penalty functions need not be strict
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

由于历史原因，penalty函数不只是返回一个float结果，而是必须把该值存储在由第三个参数指定的位置。虽然传回该参数的地址符合惯例，但返回值本身可以被忽略。

penalty函数对于索引的好性能是至关重要的。在插入时，当要选择在树中的哪个位置加入新项时，这个函数有助于决定应该顺着哪个分支进行。在查询时，索引越平衡，查找越快速。

picksplit

当需要一次索引页面分裂时，这个函数决定在该页面上哪些项会留在旧页面上，以及哪些项会移动到新页面上。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    int         i,
                nbytes;
    OffsetNumber *left,
               *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;
```

```
    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* 初始化裸的项向量。 */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * 选择在哪里放置索引项并且相应地更新 unionL 和 unionR。
         * 把项追加到 v_spl_left 或者 v_spl_right，并且设置好计数器。
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*
             * 和在右边的过程相同
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

注意picksplit函数的结果通过修改传入的v结构来传递。尽管传回v的地址符合惯例，但返回值本身可以被忽略。

和penalty一样，picksplit函数对于索引的好性能至关重要。设计合适的penalty和picksplit是实现一个好的GiST索引中最大的挑战。

same

如果两个索引项相同则返回真，否则返回假（一个“索引项”是该索引的存储类型的一个值，而不一定是原始被索引列类型的值）。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

由于历史原因，same函数不只返回一个布尔结果。相反它必须把该标志存储在第三个参数指示的位置。尽管传回该参数的地址符合惯例，但返回值本身可以被忽略。

distance

给定一个索引项p和一个查询值q，这个函数决定两者之间的“距离”。如果操作符类包含任何排序操作符，就必须提供这个函数。一个使用排序操作符的查询将首先返回具有最小“距离”值的索引项，因此结果必须与操作符的语义一致。对于一个页索引项，结果只表示到索引项的距离；对于一个内部树结点，结果必须是到任何子项的最小距离。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid,
 internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
```

```
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;

    /*
     * determine return value as a function of strategy, key and query.
     */

    PG_RETURN_FLOAT8(retval);
}
```

distance函数的参数和consistent函数的相同。

在决定距离时允许有某种近似，只要结果不要超过该项的实际距离即可。因此，例如在几何应用中到一个外包盒的距离就足够了。对于一个内部树节点，返回的距离不能超过到其任意一个子节点的距离。如果返回的距离不准确，该函数必须设置*recheck为真（这对于内部树节点是不必要的，对于它们，计算总是被假定为不准确）。在这种情况下，执行器将在从堆中取出元组后计算精确的距离，并且在必要时记录这些元组。

如果距离函数对任意叶子节点都返回*recheck = true，初始的排序操作符的返回类型必须是float8或者float4，并且距离函数的结果值必须能和初始排序操作符的结果进行比较，因为执行器将使用距离函数结果和重新计算的排序操作符结果进行排序。否则，该距离函数的结果值可以是任意有限的float8值，只要这些结果值的相对顺序匹配该排序操作符返回的顺序（在内部会使用无穷以及负无穷来处理空值等情况，因此我们不推荐distance函数返回这些值）。

fetch

为只用索引的扫描将一个数据项压缩过的索引表达转换成原始的数据类型。被返回的数据必须是原始被索引值的一份准确的、非有损的拷贝。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

参数是一个指向GISTENTRY结构的指针。在项上，它的key域包含一个压缩形式的非-NULL叶子数据。返回值是另一个GISTENTRY结构，其key域包含同一数据的原始的未压缩形式。如果操作符类的压缩函数不对叶子项做任何事情，fetch方法可以原样返回参数。或者，如果该opclass没有一个压缩函数，则fetch方法也可以被省略，因为它必须是一个空操作。

C 模块中相应的代码可能会遵循下面的框架：

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetP(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;
```

```
    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * 将 'fetched_data' 转换成原始数据类型的一个 Datum。
     */

    /* 从 fetch_data 填充 *retval。 */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

如果该压缩方法对于叶子项是有损的，操作符类就不能支持只用索引的扫描，并且不能定义fetch函数。

所有的 GiST 支持方法通常都在一个短暂存在的内存上下文中被调用；就是说，每个元组被处理之后CurrentMemoryContext将被重置。因此没有必要操心释放你 palloc 的所有东西。但是，在某些情况下，一个支持方法在重复调用之间缓存数据是有用的。要这样做，将这些长期生存的数据分配在fcinfo->flinfo->fn_mcxt中，并且在fcinfo->flinfo->fn_extra中保持一个到它的指针。这种数据将在索引操作期间都存在（例如一次 GiST 索引扫描、索引构建或索引元组插入）。注意当替换一个fn_extra值时要释放之前的值，否则在操作期间该泄露会累积。

64.4. 实现

64.4.1. GiST 缓冲构建

通过简单地插入所有元组来构建大型 GiST 索引很容易变得很慢，因为如果索引元组分散在索引中并且索引大到不足以放入在缓存中，插入操作需要执行很多随机 I/O。从版本 9.2 开始，PostgreSQL 支持一种更有效率的方法来基于缓冲构建 GiST 索引，这能显著地减少用于非排序数据集所需的随机 I/O 数量。对于排序好的数据集这种收益很小甚至不存在，因为在那时只有少数页面会接收新元组，并且那些页面能放在缓存中（即便整个索引不能放在缓存中）。

但是，缓冲索引构建需要更频繁地调用penalty函数，这会消耗更多额外的 CPU 资源。还有，在缓冲构建中使用的缓冲区需要临时磁盘空间，最多为结果索引的尺寸。缓冲也可能影响结果索引的质量，不管是正向还是负向。这种影响取决于多种因素，如输入数据的分布和操作符类的实现。

默认情况下，当索引尺寸达到effective_cache_size时，一个 GiST 索引构建会切换到缓冲方法。可以通过 CREATE INDEX 命令的buffering参数手工打开或关闭这个特性。默认行为对大部分情况是好的，但是如果输入数据是排序好的，关闭缓冲特性可能会加速构建过程。

64.5. 示例

PostgreSQL源码包包括了多个用GiST实现的索引方法的例子。核心系统当前提供文本搜索支持（用于tsvector和tsquery的索引）以及用于一些内建几何数据类型（src/backend/access/gist/gistproc.c）的 R 树等效功能。下列contrib模块也包含GiST操作符类：

btree_gist

多种数据类型的 B 树等效功能

cube

多维立方体的索引

hstore

存储键值对的模块

intarray

一维 int4 值数组的 RD 树

ltree

树状结构的索引

pg_trgm

使用 trigram 匹配的文本相似性

seg

“float ranges”的索引

第 65 章 SP-GiST索引

65.1. 简介

SP-GiST是空间划分GiST（Space-partitioned GiST）的简称。SP-GiST支持划分搜索树，它们可用于开发许多各种不同的非平衡数据结构，例如四叉树、k-d树和单词查找树。这些结构的共同特征是它们反复地将搜索空间划分成大小不需要相等的分区。匹配这些划分规则的搜索将会很快。

这些常用的数据结构最初是为在内存中使用而设计的。在主存中，它们通常被设计为一组由指针链接的动态分配的结点。这对直接在磁盘上存储并不合适，因为这些指针链可能很长并且需要太多次的磁盘访问。相反，基于磁盘的数据结构应该具有高扇出来最小化 I/O。SP-GiST所提出的挑战是将搜索树结点映射到磁盘页面，这样即使是一次搜索会穿过很多结点，它也只需要访问很少的几个磁盘页面。

和GiST一样，SP-GiST也打算允许带有合适访问方法的自定义数据类型的开发，这种开发只需由该数据类型的领域专家参与，而不需要数据库专家的参与。

这里的一些信息是来自于普渡大学的 SP-GiST 索引项目网站[1]。PostgreSQL中的SP-GiST实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，在他们的 网站[2]上有更多信息。

65.2. 内建操作符类

表 65.1中展示了PostgreSQL 核心发布所包括的SP-GiST操作符类。

表 65.1. 内建 SP-GiST 操作符类

名称	索引数据类型	可索引操作符
kd_point_ops	point	<< <@ <^ >> >^ ~=
quad_point_ops	point	<< <@ <^ >> >^ ~=
range_ops	任何范围类型	&& &< &> -|- << <@ = >> @>
box_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
poly_ops	polygon	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~ ^@
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =

在用于类型point的两种操作符类中，quad_point_ops是默认值。kd_point_ops支持相同的操作符，但是使用一种不同的索引数据结构，在某些应用中它可以提供更好的性能。

65.3. 可扩展性

SP-GiST提供了一个高抽象层次的接口，要求访问方法开发者实现与一个给定数据类型相关的几种方法。SP-GiST核心负责高效的磁盘映射和搜索树结构。它也会处理并发和日志。

SP-GiST树的叶子元组包含与被索引列数据类型相同的值。在根层的叶子元组总是包含原始的被索引数据值，但是在较下层的叶子元组可能只含有一个压缩后的表示，例如一个后缀。

[1] https://www.cs.purdue.edu/spgist/
[2] http://www.sai.msu.su/~megera/wiki/spgist_dev

在这种情况下，操作符类支持函数必须能够使用从内部元组计算出来的信息重构出原始的值，这些内部元组指的是在到达叶子层的过程中穿过的元组。

内部元组更加复杂，因为它们是搜索树的分支点。每一个内部元组包含一个或者更多个结点，结点表示一个具有相似叶子值的组。一个结点包含一个向下的链接，这个链接可以导向另一个较下层的内部元组，或者是由位于同一索引页面的叶子元组组成的一个短列表。每一个结点通常还有一个标签来描述它，例如，在一个 radix 树中结点标签可以是串值的下一个字符（或者，如果一种操作符类对于所有内部元组使用一个固定的节点集合，则它可以省略节点标签，见第 65.4.2 节。可选地，一个内部元组可以有一个前缀值来描述它所有的成员。在一个 radix 树中前缀可以是所表示的串的公共前缀。前缀值并不一定非要是一个真正的前缀，它可以是操作符类需要的任何数据。例如，在一个四叉树中它可以存储用于划分四个象限的中心点。一个四叉树的内部元组则可以包含对应于围绕该中心点的象限的四个结点。

某些树算法要求当前元组所在层（或深度）的知识，因此SP-GiST核心为操作符类提供了机会以便在沿着树下降时管理层计数。当需要重组被表示的值时，这也可以为增量地重构过程提供支持，这还可以为沿着树下降时向下层传递附加数据（称为贯穿值）提供支持。

> **注意**
>
> SP-GiST核心代码会关注空项。尽管SP-GiST索引确实可以存储被索引列中的空值，但这对索引操作符类代码是隐藏的：不会有空索引项或搜索条件会被传递给操作符类方法（我们假定SP-GiST操作符是严格的并且因此无法成功处理空值）。因此这里不会进一步讨论空值。

一个SP-GiST的索引操作符类必须提供五个用户定义的方法，并且还有一个可选的方法。所有五个强制的方法都接受两个internal参数，其中第一个是一个指针，它指向一个包含用于支持方法的值的 C 结构。而第二个参数也是一个指针，它指向将放置输出值的 C 结构。其中四个强制的函数只返回void，因为它们的所有结果都出现在输出结构中。但是leaf_consistent会额外返回一个boolean结果。这些方法不能修改它们的输入结构的任何域。在所有情况下，调用用户定义的方法之前输出结构都被初始化为零。可选的第六个方法compress接受要被索引的数据作为唯一的参数并且返回适合于在叶子元组中物理存储的值。

五个强制的用户定义的方法是：

config

返回关于索引实现的静态信息，包括前缀的数据类型的OID以及结点标签数据类型。

这个函数的SQL声明必须看起来像这样：

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

第一个参数是一个指向spgConfigIn C 结构的指针，包含该函数的输入数据。第二个参数是一个指向spgConfigOut C 结构的指针，函数必须将结果数据填充在其中。

```
typedef struct spgConfigIn
{
    Oid         attType;        /* 要被索引的数据类型 */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* 内部元组前缀的数据类型 */
    Oid         labelType;      /* 内部元组结点标签的数据类型 */
```

```
    Oid         leafType;       /* 叶子元组值的数据类型 */
    bool        canReturnData;  /* 操作符类能重构原始数据 */
    bool        longValuesOK;   /* 操作符类能处理值 > 1 页 */
} spgConfigOut;
```

为了支持多态的索引操作符类，attType要被传入；对于普通固定数据类型的操作符类，它将总是取相同的值，因此可以被忽略。

对于不使用前缀的操作符类，prefixType可以被设置为VOIDOID。同样，对于不使用结点标签的操作符类，labelType可以被设置为VOIDOID。如果操作符类能够重构出原来提供的被索引值，则canReturnData应该被设置为真。只有当attType是变长的并且操作符类能够将长值通过反复的添加后缀分段时，longValuesOK才应当被设置为真（参见第 65.4.1 节。

leafType通常和attType相同。为了向后兼容性的原因，方法config可以将leafType保留成未初始化的形态，那样会得到与把leafType设置为等于attType时一样的效果。当attType与leafType不同时，可选的方法compress必须被提供。方法compress负责把要被索引的数据从attType转换为leafType。注意：两种一致函数都会得到未更改的scankeys，也不会使用compress转换。

choose

为将一个新值插入到一个内部元组选择一种方法。

该函数的SQL声明必须看起来像这样：

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

第一个参数是一个指向spgChooseIn C 结构的指针，包含该函数的输入数据。第二个参数是一个指向spgChooseOut C 结构的指针，函数必须将结果数据填充在其中。

```
typedef struct spgChooseIn
{
    Datum       datum;          /* 要被索引的原始数据 */
    Datum       leafDatum;      /* 要被存储在叶子中的当前数据 */
    int         level;          /* 当前层（从零计数） */

    /* 来自当前内部元组的数据 */
    bool        allTheSame;     /* tuple is marked all-the-same? */
    bool        hasPrefix;      /* 元组有一个前缀？ */
    Datum       prefixDatum;    /* 如果有，前缀值 */
    int         nNodes;         /* 内部元组中的结点数目 */
    Datum      *nodeLabels;     /* 结点标签值（如果没有为 NULL） */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* 下降到现有结点 */
    spgAddNode,                 /* 向内部元组增加一个结点 */
    spgSplitTuple               /* 划分内部元组（修改它的前缀） */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* 动作代码，见上文 */
    union
    {
        struct                  /* 用于spgMatchNode的结果 */
```

```
        {
            int         nodeN;      /* 下降到这个结点（索引从 0 开始） */
            int         levelAdd;   /* 这次匹配增加的层 */
            Datum       restDatum;  /* 新叶数据 */
        }           matchNode;
        struct                  /* 用于spgAddNode的结果 */
        {
            Datum       nodeLabel;  /* 新结点的标签 */
            int         nodeN;      /* 在哪里插入它（索引从 0 开始） */
        }           addNode;
        struct                  /* 用于spgSplitTuple的结果 */
        {
            /* 来自有一个子元组的新上层内部元组的信息 */
            bool        prefixHasPrefix;    /* 元组能有前缀吗？ */
            Datum       prefixPrefixDatum;  /* 如果有，前缀值 */
            int         prefixNNodes;       /* 节点的数目 */
            Datum      *prefixNodeLabels;   /* 它们的标签（NULL表示无标签）*/
            int         childNodeN;         /* 哪个节点有子元组 */

            /* 来自放有所有旧结点的新下层内部元组的信息 */
            bool        postfixHasPrefix;   /* 元组能有前缀吗？ */
            Datum       postfixPrefixDatum; /* 如果有，前缀值 */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

datum是要被插入到该索引中的spgConfigIn.attType类型的原始数据。leafDatum是一个spgConfigOut.leafType类型的值，它最初是方法compress应用到datum上的结果（如果提供了方法compress）或者是和datum相同的值（如果没有提供compress方法）。但是如果choose或picksplit改变了它，那么位于树的较低层的leafDatum值可能会改变。当插入搜索到达一个叶子页，leafDatum的当前值就会被存储在新创建的叶子元组中。level是当前内部元组的层次，根层是 0 。如果当前内部元组被标记为包含多个等价节点（见第 65.4.3 节，allTheSame为真。如果当前内部元组有一个前缀，hasPrefix为真，如果这样，prefixDatum为前缀值。nNodes是包含在内部元组中子结点的数量，并且nodeLabels是这些子结点的标签值的数组，如果没有标签则为 NULL。

choose函数能决定新值是匹配一个现有子结点，或是必须增加一个新的子节点，亦或是新值和元组的前缀不一致并且因此该内部元组必须被分裂来创建限制性更低的前缀。

如果新值匹配一个现有的子结点，将resultType设置为spgMatchNode。将nodeN设置为该结点在结点数组中的索引（从零开始）。将levelAdd设置为传到该结点导致的level增加，或者在操作符类不使用层数时将它置为零。如果操作符类没有把数据从一层修改到下一层，将restDatum设置为等于datum，否则将它设置为在下一层用作leafDatum的被修改后的值。

如果必须增加一个新的子结点，将resultType设置为spgAddNode。将nodeLabel设置为在新结点中使用的标签，并将nodeN设置为插入该结点的位置在结点数组中的索引（从零开始）。在结点被增加之后，choose函数将被再次调用并使用修改后的内部元组，那时将会导致一个spgMatchNode结果。

如果新值和元组的前缀不一致，将resultType设置为spgSplitTuple。这个动作将所有现有的结点移动到一个新的下层内部元组，并且将现有的内部元组用一个新元组替换，该元组只有一个到那个新的下层内部元组的向下链接。将prefixHasPrefix设置为指示新的上层元组是否具有一个前缀，并且在如果有前缀时设置prefixPrefixDatum为前缀值。这个新的前缀值必须比原来的值要足够宽松以便能够接受将被索引的新值。
将prefixNNodes设置为新元组中所需的节点数，并且将prefixNodeLabels设置为一个已分配的保存它们的标签的数组，或者在不要求节点标签时设置为NULL。注意新上层元组的总尺寸必须不超过它所替换的元组的总尺寸，这限制了新前缀和新标签的长度。
将childNodeN设置为将下链到新的下层内元组的节点的索引（从零开始）。设

置postfixHasPrefix表示新的下层内元组是否应该有一个前缀，并且在应该有前缀的情况下设置postfixPrefixDatum为前缀值。这两种前缀以及下链节点的标签（如果有）的组合必须具有与原始前缀相同的含义，因为没有机会修改被移动到新下层元组的节点标签，也不能更改任何子索引项。在该节点被分裂后，将再次用替换的内元组调用choose函数。如果spgSplitTuple动作没有创建出合适的节点，该调用可以返回一个spgAddNode结果。最终choose必须返回spgMatchNode以允许插入下降到下一层次中。

picksplit

决定如何在一组叶子元组上创建一个新的内部元组。

该函数的SQL声明必须看起来像这样：

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

第一个参数是一个指向spgPickSplitIn C 结构的指针，包含该函数的输入数据。第二个参数是一个指向spgPickSplitOut C 结构的指针，函数必须将结果数据填充在其中。

```
typedef struct spgPickSplitIn
{
    int         nTuples;        /* 叶子元组的数量 */
    Datum      *datums;         /* 它们的数据（长度为 nTuples 的数组） */
    int         level;          /* 当前层次（从零开始计） */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool        hasPrefix;      /* 新内部元组应该有一个前缀吗？ */
    Datum       prefixDatum;    /* 如果有，前缀值 */

    int         nNodes;         /* 新内部元组的结点数 */
    Datum      *nodeLabels;     /* 它们的标签（没有标签则为NULL） */

    int        *mapTuplesToNodes;   /* 每一个叶子元组的结点索引 */
    Datum      *leafTupleDatums;    /* 存储在每一个新叶子元组中的数据 */
} spgPickSplitOut;
```

nTuples是提供的叶子元组的数目。datums是它们的spgConfigOut.leafType类型的数据值的数组。level是所有这些叶子元组共享的当前层，它将成为新内部元组所在的层次。

将hasPrefix设置为指示新内部元组是否应该有前缀，并且如果有前缀则将prefixDatum设置成前缀值。将nNodes设置为新内部元组将包含的结点数目，并且将nodeLabels设置为它们的标签值的数组或者 NULL（如果结点不要求标签）。将mapTuplesToNodes设置为一个数组，该数组告诉每一个叶子元组应该被赋予的结点的索引（从零开始）。将leafTupleDatums设置为由将要被存储在新叶子元组中的值构成的一个数组（如果操作符类不将数据从一层修改到下一层，这些值将和输入的datums相同）。注意picksplit函数负责为nodeLabels、mapTuplesToNodes和leafTupleDatums数组进行 palloc。

如果提供了多于一个叶子元组，picksplit被寄望于将它们分类到多余一个结点中；否则不可能将叶子元组划分到多个页面，这也是这个操作的终极目的。因此，如果picksplit函数结束时把所有叶子元组放在同一个结点中，核心SP-GiST代码将覆盖该决定，并且生成一个内部元组，将叶子元组随机分配到多个不同标签的结点。这样一个元组被标记为allTheSame来表示发生了这种情况。choose和inner_consistent函数必须对这样的内部元组采取合适的处理。详见第 65.4.3 节

picksplit只能在一种情况下被应用在单独一个叶子元组上，这种情况是config函数将longValuesOK设置为真并且提供了一个长于一页的输入。在这种情况中，该操作的要

点是剥离一个前缀并且产生一个新的、较短的叶子数据值。这种调用将被重复直到产生一个足够短能够放入到一页的叶子数据。详见第 65.4.1 节

inner_consistent

在树搜索过程中返回一组要追求的结点（分支）。

该函数的SQL声明必须看起来像这样：

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

第一个参数是一个指向spgInnerConsistentIn C 结构的指针，包含该函数的输入数据。第二个参数是一个指向spgInnerConsistentOut C 结构的指针，函数必须将结果数据填充在其中。

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* 操作符和比较值的数组 */
    int         nkeys;          /* 数组的长度 */

    Datum       reconstructedValue;     /* 在父结点中的重构值 */
    void       *traversalValue; /* 操作符类相关的贯穿值 */
    MemoryContext traversalMemoryContext;   /* 把新的贯穿值放在这里 */
    int         level;          /* 当前层次（从零开始计） */
    bool        returnData;     /* 是否必须返回原始数据？ */

    /* 来自当前内元组的数据 */
    bool        allTheSame;     /* 元组被标记为完全相同？ */
    bool        hasPrefix;      /* 元组有前缀？ */
    Datum       prefixDatum;    /* 如果有，前缀值 */
    int         nNodes;         /* 内元组中的结点数 */
    Datum      *nodeLabels;     /* 结点标签值（没有就是 NULL） */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* 要被访问的子结点数 */
    int        *nodeNumbers;    /* 它们在结点数组中的索引 */
    int        *levelAdds;      /* 为每个子结点要增加的层数 */
    Datum      *reconstructedValues;    /* 相关的重构值 */
    void      **traversalValues;        /* 操作符类相关的贯穿值 */
} spgInnerConsistentOut;
```

长度为nkeys的数组scankeys描述了索引搜索条件。这些条件用 AND 组合 — 只对满足所有条件的索引项感兴趣（注意，nkeys = 0 表示所有索引项满足该查询）。通常一致函数只关心每个数组项的sk_strategy和sk_argument，它们分别给出了可索引操作符和比较值。特别要说明的是，没有必要去检查sk_flags来看比较值是否为 NULL，因为 SP-GiST 的核心代码会过滤这样的条件。reconstructedValue是用于父元组的重构值，在根层时或者如果inner_consistent函数没有在父层提供一个值时，它为(Datum) 0。reconstructedValue总是spgConfigOut.leafType类型。traversalValue是任意贯穿数据的指针，该数据由父索引元组上的上一次inner_consistent调用传递下来，在根层上这个指针为 NULL。traversalMemoryContext是用于存放输出的贯穿值（见下文）的内存上下文。level是当前内元组层次，根层是 0。如果这个查询要求重构的数据，returnData是true。如果config断言canReturnData，returnData只会是true。如果当前的内元组被标记为“完全一样”，那么allTheSame为真。在这种情况下，所有的结点都具有相同的标签（如果有），而且它们要么全部匹配该查询，要么一个都不匹配查

询（见第 65.4.3 节。如果当前内元组包含一个前缀，则hasPrefix为真。如果这样，prefixDatum就是该前缀的值。nNodes是包含在内元组中的子结点的数量，nodeLabels是它们的标签值的数组。当然如果结点没有标签，这个数组就为 NULL。

nNodes必须被设置为搜索需要访问的子结点数，并且nodeNumbers必须被设置为子结点索引的数组。如果操作符类跟踪层次，把levelAdds设置成一个数组，其中说明了在下降到要被访问的每一个结点时需要增加的层数（通常这些增量对于所有结点都是相同的，但是并不一定如此，所以需要使用一个数组）。如果需要值重构，将reconstructedValues设置为一个spgConfigOut.leafType类型值的数组，这些值是为要被访问的每一个子节点构造的。否则，把reconstructedValues留为NULL。如果想要把额外的带外信息（“贯穿值”）向下传递给树搜索的较低层，可以把traversalValues设置成合适的贯穿值的数组，其中每一个元素用于一个要被访问的子节点。如果不需要传递额外的带外信息，则把traversalValues设置为 NULL。注意，inner_consistent函数负责在当前内存上下文中分配nodeNumbers、levelAdds、reconstructedValues和traversalValues数组。不过，任何由traversalValues数组指向的输出贯穿值应该在traversalMemoryContext中分配。每一个贯穿值必须是一个单独分配的块（chunk）。

leaf_consistent

如果一个叶子元组满足一个查询则返回真。

该函数的SQL声明必须看起来像这样：

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

第一个参数是一个指向spgLeafConsistentIn C 结构的指针，包含该函数的输入数据。第二个参数是一个指向spgLeafConsistentOut C 结构的指针，函数必须将结果数据填充在其中。

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* 操作符和比较值的数组 */
    int         nkeys;          /* 数组的长度 */

    Datum       reconstructedValue;     /* 在父节点重构的值 */
    void       *traversalValue; /* 操作符类相关的贯穿值 */
    int         level;          /* 当前层次（从零开始计） */
    bool        returnData;     /* 是否必须返回原始数据？ */

    Datum       leafDatum;      /* 叶子元组中的数据 */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* 重构的原始数据，如果有 */
    bool        recheck;        /* 如果操作符必须被重新检查则设为真 */
} spgLeafConsistentOut;
```

长度为nkeys的数组scankeys描述了索引搜索条件。这些条件用 AND 组合在一起 — 只有满足所有条件的索引项才满足该查询（注意nkeys = 0 表示所有的索引项都满足查询）。通常 consistent 函数值关注每一个数组项的sk_strategy和sk_argument域，它们分别给出了可索引操作符和比较值。特别是它无需检查sk_flags来检查比较值是否为 NULL，因为 SP-GiST 核心代码将过滤掉这类条件。reconstructedValue是为父元组重构的值，在根层或者当inner_consistent没有提供父层上的值时，它是(Datum) 0。reconstructedValue总是spgConfigOut.leafType类型。traversalValue是任意贯穿数据的指针，该数据由父索引元组上的上一次inner_consistent调用传递下来，在根层上这个指针为 NULL。level是当前的叶子元组所在的层次，根层为零。如果这个查询要

求重构的数据，则returnData为true。只有在config函数主张了canReturnData时才会如此。leafDatum是存储在当前叶子元组中的spgConfigOut.leafType的键值。

如果叶子元组匹配查询，则该函数必须返回true，否则返回false。在返回true的情况中，如果returnData为true，则leafValue必须被设置为最初为构建这个叶子元组提供的spgConfigIn.attType类型值。还有，如果匹配是不确定的并且操作符必须被重新应用在实际堆元组上验证匹配，则recheck会被设置为true。

可选的用户定义的方法是：

Datum compress(Datum in)

将数据项转换成一种适合索引页面的叶子元组中物理存储方式的格式。它接受spgConfigIn.attType值并且返回spgConfigOut.leafType值。输出值不应该被TOAST。value. Output value should not be toasted.

所有的 SP-GiST 支持方法通常都在一个短期存在的内存上下文中被调用，即在处理完每一个元组后CurrentMemoryContext将被重置。因此并不需要操心 pfree 你palloc 的任何东西（config方法是一个特例：它应该避免泄漏内存。但是通常config方法只需要为传出的参数结构赋常数值）。

如果被索引的列是一种可排序的数据类型，索引的排序规则将被使用标准的PG_GET_COLLATION()机制传递给所有的支持方法。

65.4. 实现

这一节覆盖了实现细节以及SP-GiST操作符类的实现者需要知道的有用的技巧。

65.4.1. SP-GiST 限制

单独的叶子节点和内部节点必须能适合一个单一的索引页面（默认为 8kB）。因此，当索引值是一种变长数据类型时（长值只能由如 radix 树的方法所支持），树的每一层包含的前缀都足够短以适合一个页面，并且最终的叶子层包括的后缀也足够短以适合一个页面。如果操作符类准备好做这种事情，它应该将longValuesOK设置为true。否则，SP-GiST核心将拒绝任何要索引超过一个所以页面长度的值的请求。

同样，操作符类应该负责不要让内部元组增长到无法放在一个索引页面中。这限制了能在一个内部元组中使用的子节点的数目，以及一个前缀值的最大尺寸。

另一个限制是，当一个内部元组的节点指向一组叶子元组时，这些元组必须都在同一个索引页面中（这种设计是为了减少在这类元组构成链中进行定位的时间并且节省空间）。如果叶子元组集合增长到无法放在一个页面中，将执行一次分裂并且插入一个中间的内部元组。为此，新的内部元组必须把叶子值的集合划分成多于一个节点分组。如果操作符类的picksplit函数无法做到这一点，SP-GiST核心只能求助于第 65.4.3 节所介绍的额外措施。

65.4.2. 无节点标签的 SP-GiST

某些树算法对每个内部元组都使用一种固定的节点集合。例如，在一个四叉树中总是正好有四个节点对应于围绕内部节点中心点的四个象限。在这种情况下，代码总是通过编号来处理节点，而不需要显式的节点标签。要抑制节点标签（因而节省一些空间），picksplit函数可以为nodeLabels数组返回NULL，同样choose函数可以在一个spgSplitTuple动作期间为prefixNodeLabels数组返回NULL。这将会导致后续对choose和inner_consistent调用时nodeLabels也为 NULL。原则上，可以为同一个索引中的某些内部元组使用节点标签而对其他内部节点省略节点标签。

在处理具有无标签节点的内部元组时，让choose返回spgAddNode是一种错误，因为该节点集合在这种情况下被假定为固定的集合。

65.4.3. “All-the-same”内部元组

当picksplit无法把提供的叶子值划分成至少两个节点分类，SP-GiST核心能推翻操作符类的picksplit函数的结果。在发生这种情况时，会创建一个新的内部元组，其中有多个节点，每一个节点都有相同的标签（如果有标签），标签是由picksplit之前给一个节点用的，并且叶子值会被随机地划分给这些等效的节点中。该内部元组上会设置allTheSame标志以警告choose和inner_consistent函数该元组不具有它们所期望的节点集合。

在处理allTheSame元组时，choose函数的结果spgMatchNode会被解释为新值可以被赋值给任一等价的节点。核心代码将忽略提供的nodeN值并且随机地下降到其中一个节点中（以便保持树平衡）。对choose来说，返回spgAddNode是一种错误，因为那会让节点不全部等效。如果要被插入的值不匹配现有的节点，则必须使用spgSplitTuple动作。

在处理allTheSame元组时，为了继续索引搜索，inner_consistent函数应该返回全部节点或者不返回节点作为目标，因为这些节点都是等效的。根据inner_consistent函数对这些节点含义的假定程度，这可能会也可能不会要求任何处理特殊情况的代码。

65.5. 例子

如表 65.1中所述，PostgreSQL源代码发布包括多个用于SP-GiST的索引操作符类的例子。其代码可以看看src/backend/access/spgist/和src/backend/utils/adt/中的文件。

第 66 章 GIN 索引

66.1. 简介

GIN意思是通用倒排索引。GIN被设计为处理被索引项为组合值的情况，并且这种索引所处理的查询需要搜索出现在组合项中的元素值。例如，项可以是文档，并且查询可以是搜索包含指定词的文档。

我们使用词项来表示要被索引的一个组合值，并且用词键来表示一个元素值。GIN总是存储和搜索键，而不是项值本身。

一个GIN存储一个（键，位置列表）对的集合，这里一个posting list是键在其中出现的一个行 ID 的集合。相同的行 ID 可以出现在多个位置列表中，因为一个项可以包含多于一个键。每个键值只被存储一次，因此对于同一个键出现多次的情况，一个GIN索引是非常紧凑的。

GIN访问方法代码不需要知道它所加速的是什么操作，从这个意义上来说，GIN是通用的。相反，它使用为特定数据类型定义的自定义策略。策略定义如何从被索引项和查询条件中抽取键，并且如何决定一个包含查询中某些键值的行是否真正满足查询。

GIN的一个优点是它允许由数据类型的领域专家开发有合适访问方法的自定义数据类型，而不是让一个数据库专家来做这件事。在这一点上很像GiST。

PostgreSQL中的GIN实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护。在他们的网站[1]上有更多关于GIN的信息。

66.2. 内建操作符类

PostgreSQL的核心发布包括表 66.1中所示的GIN操作符类（附录 F中描述的一些 可选模块提供了额外的GIN操作符类）。

表 66.1. 内建GIN操作符类

名称	索引数据类型	可索引操作符
array_ops	anyarray	&& <@ = @>
jsonb_ops	jsonb	? ?& ?| @>
jsonb_path_ops	jsonb	@>
tsvector_ops	tsvector	@@ @@@

在两种用于类型jsonb的操作符类中，jsonb_ops是默认项。 jsonb_path_ops支持较少的操作符但是为那些操作符提供了更好的性能。 详见第 8.14.4 节

66.3. 可扩展性

GIN接口有一个高层次的抽象，要求访问方法实现者只需要实现数据类型被访问的语义。GIN层本身会操心并发、日志和搜索树结构的事情。

要让一个GIN访问方法工作起来所要做的全部事情就是实现一些用户定义的方法，它们定义了树中键的行为以及键、被索引项以及可索引查询之间的关系。简而言之，GIN的可扩展性结合了通用性、代码重用和一个干净的接口。

一个用于GIN的操作符类必须提供的两种方法是：

[1] http://www.sai.msu.su/~megera/wiki/Gin

Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)

给定一个要被索引的项，返回一个 palloc 过的键的数组。被返回的键的数量必须被存储在*nkeys中。如果键中的任意一个可能为空，还要 palloc 一个*nkeys 个bool域的数组，将它的地址存储在*nullFlags中，并且根据需要设置这些空值标志。如果所有的键都非空，*nullFlags可以被留成NULL（其初始值）。如果该项不包含键，返回值可以为NULL。

Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)

给定一个要被查询的值，返回一个 palloc 过的键的数组。即query是一个可索引操作符（左手边是被索引列）的右手边的值。n是操作符类中操作符的策略号（见第 38.15.2 节。通常，extractQuery将需要参考n来判断query的数据类型以及它应该用什么方法来抽取键值。被返回的键的数量必须被存储在*nkeys中。如果键中的任意一个可能为空，还要 palloc 一个*nkeys 个bool域的数组，将它的地址存储在*nullFlags中，并且根据需要设置这些空值标志。如果所有的键都非空，*nullFlags可以被留成NULL（其初始值）。如果该项不包含键，返回值可以为NULL。

searchMode是一个输出参数，它允许extractQuery指定有关搜索如何被完成的细节。如果*searchMode被设置为GIN_SEARCH_MODE_DEFAULT（这是在被调用之前它被初始化的值），只有那些匹配至少一个被返回键的项才会被考虑作为候选匹配。如果*searchMode被设置为GIN_SEARCH_MODE_INCLUDE_EMPTY，那么除了至少包含一个匹配键的项之外，根本不包含键的项也被考虑作为候选匹配（例如，这种模式对于实现“是...的子集”操作符有用）。如果*searchMode被设置为GIN_SEARCH_MODE_ALL，那么索引中所有非空项都被考虑作为候选匹配，不管它们是否匹配被返回的键（这种模式比其他两种选择要慢很多，但是它对于正确实现极端情况可能是必要的。需要这种模式的操作符在大部分情况下可能并不是一个 GIN 操作符类的好选择）。用于设置这个模式的符号被定义在access/gin.h中。

pmatch是一个输出参数，它用于在部分匹配匹配被支持时使用。要用它，extractQuery必须分配一个*nkeys个布尔值的数组，并且把它的地址存储在*pmatch中。如果一个键要求部分匹配，该数组的对应元素应该被设置为 TRUE，否则设置为 FALSE。如果*pmatch被设置为NULL，则 GIN 假定不需要部分匹配。在调用前，该变量被初始化为NULL，这样这个参数可以简单地被不支持部分匹配的操作符类忽略。

extra_data是一个输出参数，它允许extractQuery传递额外数据给consistent和comparePartial方法。要用它，extractQuery必须分配一个*nkeys个指针的数组，并且把它的地址存储在*extra_data中，然后把任何它想存储的东西存到单个指针中。在调用前该变量被初始化为NULL，这样这个参数可以简单地被不需要额外数据的操作符类忽略。如果*extra_data被设置，整个数组被传递给consistent方法，并且适当的元素会被传递给comparePartial方法。

一个操作符类必须提供一个函数检查一个被索引的项是否匹配查询。有两种形式， 一个布尔函数consistent，以及一个三元函数triConsistent。 triConsistent覆盖了两者的功能，因此提供triConsistent一个足矣。但是， 如果布尔变体的计算代价要更低，两者都提供就会有好处。如果只提供布尔变体， 一些基于在取得所有键之前拒绝索引项的优化将会被禁用。

bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])

如果一个被索引项满足（如果重新检查指示被返回，则表示可能满足）有策略号n的查询操作符，则返回 TRUE。这个函数并没有直接访问被索引项的值，因为GIN没有显式存储项。可用的是关于哪些从查询抽取出的键值出现在一个给定被索引项中的知识。check数组的长度是nkeys，它和前面由extractQuery为这个查询数据返回的键的数目相同。 如果被索引项包含一个查询键，那么check数组的对应元素为 TRUE，即如果 (check[i] == TRUE) ，则extractQuery结果数组的第 i 个键存在于被索引项中。在consistent方法需

要参考原始query数据的情况中，它会被传递进来，前面由extractQuery返回的queryKeys[]和nullFlags[]数组也一样。extra_data是由extractQuery返回的额外数据数组，如果没有额外数据则为NULL。

当extractQuery在queryKeys[]中返回一个空值键时，如果被索引项包含一个空值键则对应的check[]元素为 TRUE。即，check[]的语义类似IS NOT DISTINCT FROM。如果consistent函数需要说出一个常规值匹配和一个空值匹配之间的区别，它可以检查对应的nullFlags[]元素。

在成功时，如果堆元组需要根据查询操作符被重新检查，则*recheck应该被设置为 TRUE，或者如果索引测试是准确的则设置为 FALSE。即，一个 FALSE 返回值保证堆元组不匹配查询；一个 TRUE 返回值以及设置为 FALSE 的*recheck保证堆元组匹配查询；并且一个 TRUE 返回值和设置为 TRUE 的*recheck表示堆元组可能匹配查询，因此它需要被取出并且通过在原始的被索引项上计算查询操作符来重新检查。

GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool nullFlags[])

triConsistent类似于consistent， 但和check[]中的布尔值不同，对每个键有三种可能值： GIN_TRUE、GIN_FALSE和GIN_MAYBE。 GIN_FALSE和GIN_TRUE具有和常规布尔值相同的含义， 而GIN_MAYBE意味着键的存在未知。当GIN_MAYBE值出现时， 如果项必定匹配（不管该索引项是否包含对应的查询键），该函数应该只返回GIN_TRUE。 同样地，如果项必定不匹配（不管它是否包含GIN_MAYBE）， 该函数必须只返回GIN_FALSE。如果结果依赖于GIN_MAYBE项，即无法根据已知查询键确认或拒绝匹配， 该函数必须返回GIN_MAYBE。

当在check向量中没有GIN_MAYBE值时， GIN_MAYBE返回值等效于在布尔函数consistent中设置 recheck标志等效。

此外，GIN必须有方法能排序存储在索引中的键值。操作符类可以通过指定一种比较方法来定义排序顺序：

int compare(Datum a, Datum b)

比较两个键（不是被索引项！）并且返回一个整数，该整数为小于零、等于零或者大于零分别表示第一个键小于、等于或者大于第二个键。空值键绝不会被传递给这个函数。

或者，如果操作符类不提供compare方法，GIN将查看索引键数据类型的默认btree操作符类，并且使用它的比较函数。推荐为只用于一种数据类型的GIN操作符类指定这个比较函数，因为查找btree操作符类需要消耗一些周期。不过，多态的GIN操作符类（例如array_ops）通常无法指定单一的比较函数。

可选的，一个用于GIN的操作符类可以提供下列方法：

int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)

比较一个部分匹配键和一个索引键。返回一个整数，其符号指示结果：小于零表示索引键不匹配查询，但是索引扫描应该继续；零表示索引键匹配查询；大于零表示索引扫描应该停止，因为没有更多可能的匹配。产生该部分匹配查询的操作符的策略号n将被提供，可以通过其语义决定什么时候结束扫描。还有，extra_data是由extractQuery产生的额外数据数组中的对应元素，如果没有则为NULL。空值不会被传递给这个函数。

要支持“部分匹配”查询，一个操作符类必须提供comparePartial方法，并且它的extractQuery方法必须在遇到一个部分匹配查询时设置pmatch参数。详见第 66.4.2 节

上面提到的多个Datum值的实际数据类型随着操作符类而变化。 被传递给extractValue的项值总是操作符类的输入类型， 并且所有的键值必须是类的STORAGE类型。被传递给extractQuery、 consistent和triConsistent的query 参数是由该策略号标识的类成员操作符的右手边输入类型。 这不需要和被索引类型相同，只要正确类型的键值能从其中被抽

取出来。不过， 推荐这三个支持函数的 SQL 声明对query参数使用操作符类的被 索引数据类型，即便实际类型可能是某种其他依赖于操作符的东西时也应如此。

66.4. 实现

在内部，一个GIN索引包含一个在键上构建的 B 树索引，其中每一个键是一个或者多个被索引项的一个元素（例如，数组的一个成员），并且叶子页中的每一个元组包含一个指向堆指针 B 树的指针（一个“位置树”）或者一个堆指针的简单列表（“位置列表”），只有位置列表小到能够和键值一起放入索引时才使用后一种形式。

自PostgreSQL 9.1 起，空键值可以被包括在索引中。同样，用于为空或者根据extractValue不包含键的被索引项的占位符空值也被包括在索引中。这允许实现应该找到空项的搜索。

多列GIN索引可以通过在组合值（列号，键值）上建立一个单一 B 树实现。不同列的键值可以是不同类型。

66.4.1. GIN 快速更新技术

更新一个GIN索引可能会比较慢，这是因为倒排索引的天然特性造成的： 对一个堆行的插入或更新可能导致对索引的很多次插入（每一次插入用于从被索引项中抽取的一个键）。 从PostgreSQL 8.4 开始，GIN可以通过将新元组插入到一个临时的未排序的待处理条目列表中来推迟很多这种工作。 当表被清理、自动分析、gin_clean_pending_list函数被调用 或者待处理列表变得大于gin_pending_list_limit时， 这些条目被使用初始索引创建时使用的批量插入技术移动到主GIN数据结构中。 这大幅度提高了GIN索引的更新速度，虽然带了一些额外的清理负荷。 此外，这些开销可以通过用一个后台进程来取代一个前台进程执行。

这种方式的主要缺点是搜索必须在搜索普通索引之外扫描待处理条目的列表，并且因此一个大型的待处理条目列表会显著地拖慢搜索。另一个缺点是，虽然大部分更新变快了，一次导致待处理列表变得“太大”的更新将导致一次立即清理循环并且因此会比其他更新慢很多。正确使用自动清理可以把这些问题的影响变得最小。

如果一致的响应时间比更新速度更重要，可以通过为一个GIN关闭fastupdate存储参数来禁用对待处理条目的使用。详见CREATE INDEX。

66.4.2. 部分匹配算法

GIN 可以支持“部分匹配”查询，在其中查询不能判断一个或多个键的精确匹配，但是可以确定落在键值（在compare支持方法决定的键排序顺序中）的一个合理的狭窄范围内的可能匹配。extractQuery方法，不会返回一个要被精确匹配的键值，而是返回一个作为要被搜索范围下界的键值，并且将pmatch标志设置为真。然后键范围将被使用comparePartial方法扫描。comparePartial必须对于一个匹配的索引键返回零，对一个不匹配但仍在要被搜索的范围内的返回小于零，对于超过被搜索范围的索引键返回大于零。

66.5. GIN 提示和技巧

创建 vs. 插入

插入到一个GIN索引可能会很慢，因为为一个项可能需要插入很多歌键。因此，对于一个表的批量插入，我们建议删除 GIN 索引，然后在完成批量插入后重建它。

从PostgreSQL 8.4 开始，这个建议已经不再需要了，因为可以使用延迟索引（详见第 66.4.1 节。但是对于非常大量的更新，还是最好先删除再重建索引。

maintenance_work_mem

一个GIN索引的建立时间对maintenance_work_mem设置很敏感；它不考虑在索引创建期间在工作内存上的节省。

gin_pending_list_limit

在向一个已有的开启了fastupdate的GIN 中进行插入时，系统将在待处理项列表增长到超过gin_pending_list_limit 时清理该列表。为了避免在观测到的响应时间上出现波动，让待处理列表清理操作 在后台进行（即通过 autovacuum）比较好。可以通过增加gin_pending_list_limit 或者让 autovacuum 更激进来避免前台的清理操作。不过，增大清理操作的 阈值意味着如果一次前台清理发生，它将需要更长的时间。

可以通过改变存储参数为每个 GIN 索引覆盖它所用的gin_pending_list_limit， 这样允许每个 GIN 索引拥有自己的清理阈值。例如，可以只为被大量更新的 GIN 索引增加该阈值，而对其他的索引减小该阈值。

gin_fuzzy_search_limit

开发GIN索引的主要目的是在PostgreSQL中创建对高可扩展全文搜索的支持，并且一次全文搜索返回一个非常大的结果集也是很常见的情况。此外，当查询包含非常频繁的词时情况也是如此，因此大结果集不是非常有用。因为从磁盘读取很多元组并且对它们排序可能会花费很多时间，这对于产品来说是不可接受的（注意索引搜索本身是很快的）。

为了能够控制这类查询的执行，GIN对于返回的行数有一个可配置的软上限：gin_fuzzy_search_limit配置参数。默认它被设置为 0 （意为无限制）。如果设置了一个非零限制，那么被返回的集合是整个结果集的一个子集，并且是随机选择的。

“软”意味着被返回结果的实际数量可以与指定的限制不同，这取决于查询和系统随机数生成器的质量。

根据经验，在数千级别的值（如 5000 — 20000）比较好。

66.6. 限制

GIN假定可索引操作符是严格的。这意味着对于一个空项值，extractValue将根本不会被调用（相反，一个占位符索引项将被自动创建），并且在一个空查询值上也不会调用extractQuery（相反，该查询被假定为不可满足的）。不过注意，在一个非空组合项或查询值中的空键值是被支持的。

66.7. 例子

PostgreSQL的核心发布中包括之前表 66.展示的GIN操作符类。tsvector中的前缀搜索就是使用GIN的部分匹配特性实现的。下列contrib模块也包含GIN操作符类：

btree_gin

多种数据类型的 B 树等效功能

hstore

存储键值对的模块

intarray

int[]的增强支持

pg_trgm

使用 trigram 匹配的文本相似性

第 67 章 BRIN 索引

67.1. 简介

BRIN表示块范围索引。 BRIN是为处理这样的表而设计的：表的规模非常大，并且其中某些列与它们在表中的物理位置存在某种自然关联。一个块范围是一组在表中物理上相邻的页面，对于每一个块范围在 索引中存储了一些摘要信息。例如，一个存储商店销售订单的表可能有一个日期 列记录每个订单产生的时间，并且很多时候较早的订单项也将出现在表中较早的 地方。一个存储 ZIP 代码列的表中一个城市的所有代码可能自然地聚在一起。

如果索引中存储的摘要信息与查询条件一致，BRIN 索引可以通过常规的位图索引扫描满足查询，并且将会返回每个范围中所有页面 中的所有元组。查询执行器负责再次检查这些元组并且抛弃掉那些不匹配查询条 件的元组 — 换句话说，这些索引是有损的。由于一个 BRIN 索引很小，扫描这种索引虽然比使用顺序扫描多出了一点点开销，但是可能会避 免扫描表中很多已知不包含匹配元组的部分。

一个BRIN索引将存储的特定数据以及该索引将能 满足的特定查询，都依赖于为该索引的每一列所选择的操作符类。具有一种 线性排序顺序的数据类型的操作符类可以存储在每个块范围内的最小和最大 值，例如几何类型可能会存储在块范围内的所有对象的外包盒。

块范围的尺寸在索引创建时由pages_per_range存储参数决定。 索引项的数量将等于该关系的尺寸（以页面计）除以为 pages_per_range选择的值。因此，该值越小，索引会变得越大（因为需要存储更多索引项），但是与此同时存储的摘要数据可以更加精确并 且在索引扫描期间可以跳过更多数据块。

67.1.1. 索引维护

在创建时，所有已有的堆页面将被扫描并且会为每一个范围创建一个摘要 索引元组，对于末尾的可能不完整的范围也是这样做。随着新页面被数据填 充，已经被创建摘要的页面范围的摘要信息会被来自新元组的数据所更新。 当一个被创建的新页面没有落在最后一个被摘要的范围内时，该范围不会自 动获得一个摘要元组，那些元组将保持未被摘要的状态，直到后面调用一次 摘要操作来创建初始的摘要。可以使用brin_summarize_range(regclass, bigint)或brin_summarize_new_values(regclass)函数手动调用这种处理，而当VACUUM处理表时或者插入发生时由autovacuum执行的自动摘要过程都会自动调用这种处理（最后这一个触发器默认是被禁用的，可以用autosummarize参数启用。相对地，可以用brin_desummarize_range(regclass, bigint)函数解除一个范围的摘要，在现有值发生变化导致索引元组不再是一个很好的表达式，这样做是很有用的。

当自动摘要被启用时，每次一个页面范围会被装进一个请求中发送给autovacuum，以便autovacuum为那个范围执行定向的摘要，这个请求会在运行在同一个数据库的下一个工作者的末尾被满足。如果请求队列满了，该请求不会被记录，并且在服务器日志中会有一条消息：

```
LOG:  request for BRIN range summarization for index "brin_wi_idx" page 128 was
 not recorded
```

如果这种情况发生，在该表的下一次常规vacuum时将会正常对这个范围做摘要。

67.2. 内建操作符类

核心PostgreSQL发布包括了 表 67.1中所示的 BRIN操作符类。

minmax操作符类存储范围内被索引列中出现的最小和最大值。 inclusion操作符类存储包括了范围内被索引列中值的一个值。

表 67.1. 内建 BRIN 操作符类

名称	被索引数据类型	可索引操作符
abstime_minmax_ops	abstime	< <= = >= >
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
macaddr8_minmax_ops	macaddr8	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >
range_inclusion_ops	any range type	<< &< && &> >> @> <@ -|- = < <= = > >=
float4_minmax_ops	real	< <= = >= >
reltime_minmax_ops	reltime	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >
timestamptz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >
timetz_minmax_ops	time with time zone	< <= = >= >
uuid_minmax_ops	uuid	< <= = >= >

67.3. 可扩展性

BRIN接口具有高层的抽象，要求访问方法实现者只需 实现被访问的数据类型的语义。BRIN层本身会负责并 发、日志以及对索引结构的搜索。

让一种BRIN访问方法能够工作要做的全部事情是实现 几个用户定义的方法， 它们定义存储在索引中的摘要值的行为以及它们和扫描键 的交互。简而言之，BRIN很好地把可扩展性和通用性、 代码重用以及干净的接口结合在了一起。

BRIN的一个操作符类必须提供四种方法：

BrinOpcInfo *opcInfo(Oid type_oid)

返回有关被索引列的摘要数据的内部信息。返回值必须指向一个已经 palloc 的BrinOpcInfo，该结构的定义是：

```
typedef struct BrinOpcInfo
{
    /* 这个 opclass 的一个索引列中存储的列数 */
    uint16      oi_nstored;

    /* 该 opclass 私有用途的不透明指针 */
    void       *oi_opaque;

    /* 被存储列的类型缓冲项 */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

BrinOpcInfo.oi_opaque可以被操作符类 例程用来在索引扫描期间在支持函数之间传递信息。

bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)

返回 ScanKey 是否和一个范围的被索引值一致。要使用的索引号作为 扫描键的一部分传递。

bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)

给定一个索引元组和一个被索引值，修改该元组的指示属性让该元组能额外地表 示新的值。如果对该元组做出了任何修改，就返回true。

bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)

联合两个索引元组。给定两个索引元组，修改第一个的指示属性让它能表示 两个元组。第二个元组不会被修改。

核心发布中包括了对两种类型的操作符类的支持：minmax 和 inclusion。 发布中也酌情为核心中的数据类型提供了使用它们的操作符类定义。用户可以用等效 的定义为其他数据类型定义额外的操作符类，而不需要编写任何源代码，只需要声明 一些适当的目录项就足够了。注意有关操作符策略的语义的假设是嵌在支持函数的源 代码中的。

实现完全不同的语义的操作符类也是可能的，只要提供上述的四个主要支持过程的实 现即可。注意在主要发行版之间的向后兼容性是不被保证的：例如，在以后的发行中 可能要求额外的支持过程。

要为一种实现了线性有序集的数据类型编写一个操作符类，可以使用 minmax 支持函数配上对应的操作符（如表 67.2所示）。 所有的操作符类成员（函数和操作符）都是强制性的。

表 67.2. Minmax 操作符类的函数和支持编号

操作符类成员	对象
支持函数 1	内部函数brin_minmax_opcinfo()
支持函数 2	内部函数brin_minmax_add_value()
支持函数 3	内部函数brin_minmax_consistent()
支持函数 4	内部函数brin_minmax_union()
操作符策略 1	小于操作符

操作符类成员	对象
操作符策略 2	小于等于操作符
操作符策略 3	等于操作符
操作符策略 4	大于等于操作符
操作符策略 5	大于操作符

要为值被包括在另一种类型的复杂数据类型编写操作符类，可以使用 inclusion 支持函数配上相应的操作符（如 表 67.3所示）。它只要求一个 可用任何语言编写的附加函数。可以定义更多函数来提供额外的功能。所有的 操作符都是可选的。如该表中的依赖性所示，某些操作符需要其他操作符。

表 67.3. Inclusion 操作符类的函数和支持编号

操作符类成员	对象	依赖性
支持函数 1	内部函数brin_inclusion_opcinfo()	
支持函数 2	内部函数brin_inclusion_add_value()	
支持函数 3	内部函数brin_inclusion_consistent()	
支持函数 4	内部函数brin_inclusion_union()	
支持函数 11	合并两个元素的函数	
支持函数 12	可选函数，检查两个元素是否可以合并	
支持函数 13	可选函数，检查一个元素是否被包含在另一个中	
支持函数 14	optional function to check whether an element is empty	
操作符策略 1	位于左边操作符 left-of	操作符策略 4
操作符策略 2	不超过左边操作符 does-not-extend-to-the-right-of	操作符策略 5
操作符策略 3	重叠操作符	
操作符策略 4	不超过左边操作符 does-not-extend-to-the-left-of	操作符策略 1
操作符策略 5	位于右边操作符 right-of	操作符策略 2
操作符策略 6, 18	相同或者等于操作符	操作符策略 7
操作符策略 7, 13, 16, 24, 25	包含或等于操作符	
操作符策略 8, 14, 26, 27	被包含或等于操作符	操作符策略 3
操作符策略 9	不超过上边操作符 does-not-extend-above	操作符策略 11
操作符策略 10	操作符 is-below	操作符策略 12
Operator Strategy 11	在上面操作符 is-above	操作符策略 9
操作符策略 12	不超过下面操作符 does-not-extend-below	操作符策略 10
操作符策略 20	小于操作符	操作符策略 4
操作符策略 21	小于等于操作符	操作符策略 4

操作符类成员	对象	依赖性
操作符策略 22	大于操作符	操作符策略 1
操作符策略 23	大于等于操作符	操作符策略 1

支持过程编号 1-10 被保留给 BRIN 的内部函数，因此 SQL 层面的函数从 编号 11 开始。支持函数编号 11 是用于构建该索引的主要函数。它应该接 受两个具有和操作符类相同数据类型的参数并且返回它们的并集。如果 inclusion 操作符类定义时用了STORAGE参数，则它可以存储具有 不同数据类型的合并值。该并集函数的返回值应该匹配 STORAGE的数据类型。

支持函数编号 12 和 14 被提供用来支持内建数据类型的不规则性。函数编号 12 被用来支持来自不同地址族的不能合并的网络地址。函数编号 14 被用来 支持空范围。函数编号 13 是可选的，但是我们推荐提供它。它允许在新值 被传递给并集函数前对其进行检查。因为 BRIN 框架在并集没有改变时可以 越过某些操作，所以使用这个函数可以提升索引性能。

minmax 和 inclusion 操作符类都支持跨数据类型操作符，不过如果要支持 会让依赖性变得更加复杂。minmax 操作符类要求用具有同样数据类型的 参数来定义一个完全的操作符集合。它允许通过定义额外的操作符集合来 支持附加的数据类型。如 表 67.9中所示，inclusion 操作符类的操作符策略是依赖于另一种操作符策略的（或者和它们 自身相同的操作符策略）。它们要求定义依赖性操作符时，把 STORAGE数据类型作为左手边参数并且让其他支持的数据类型 作为右手边的参数。minmax 的例子可见 float4_minmax_ops，inclusion 的例子是 box_inclusion_ops。

第 68 章 数据库物理存储

本章对PostgreSQL数据库使用的物理存储格式进行一个概述。

68.1. 数据库文件布局

本节在文件和目录的层次上描述存储格式。

在传统上，数据库集簇所使用的配置和数据文件都被一起存储在集簇的数据目录里， 通常用PGDATA来引用（用的是可以定义它的环境变量的名字）。PGDATA的一个常见位置是/var/lib/pgsql/data。由不同数据库实例所管理的多个集簇可以在同一台机器上共存。

PGDATA目录包含几个子目录以及一些控制文件， 如表 68.1所示。除了这些必要的东西之外，集簇的配置文件postgresql.conf、pg_hba.conf和pg_ident.conf通常都存储在PGDATA中，不过可以把它们放在别的地方。

表 68.1.PGDATA的内容

项	描述
PG_VERSION	一个包含PostgreSQL主版本号的文件
base	包含每个数据库对应的子目录的子目录
current_logfiles	记录当前被日志收集器写入的日志文件的文件
global	包含集簇范围的表的子目录，比如pg_database
pg_commit_ts	包含事务提交时间戳数据的子目录
pg_dynshmem	包含被动态共享内存子系统所使用的文件的子目录
pg_logical	包含用于逻辑复制的状态数据的子目录
pg_multixact	包含多事务（multi-transaction）状态数据的子目录（用于共享的行锁）
pg_notify	包含LISTEN/NOTIFY状态数据的子目录
pg_replslot	包含复制槽数据的子目录
pg_serial	包含已提交的可序列化事务信息的子目录
pg_snapshots	包含导出的快照的子目录
pg_stat	包含用于统计子系统的永久文件的子目录
pg_stat_tmp	包含用于统计信息子系统的临时文件的子目录
pg_subtrans	包含子事务状态数据的子目录
pg_tblspc	包含指向表空间的符号链接的子目录
pg_twophase	包含用于预备事务状态文件的子目录
pg_wal	包含 WAL （预写日志）文件的子目录
pg_xact	包含事务提交状态数据的子目录
postgresql.auto.conf	一个用于存储由ALTER SYSTEM 设置的配置参数的文件
postmaster.opts	一个记录服务器最后一次启动时使用的命令行参数的文件
postmaster.pid	一个锁文件，记录着当前的 postmaster 进程ID（PID）、集簇数据目录路径、postmaster

项	描述
	启动时间戳、端口号、Unix域套接字目录路径（Windows上为空）、第一个可用的listen_address（IP地址或者*，或者为空表示不在TCP上监听）以及共享内存段ID（服务器关闭后该文件不存在）

对于集簇里的每个数据库，在PGDATA/base里都有一个子目录对应， 子目录的名字为该数据库在 pg_database里的 OID。这个子目录是该数据库文件的缺省位置；特别值得一提的是，该数据库的系统目录存储在此。

每个表和索引都存储在独立的文件里。对于普通关系，这些文件以表或索引的filenode号命名，它可以在pg_class.relfilenode中找到。但是对于临时关系，文件名的形式为tBBB_FFF，其中BBB是创建该文件的后台的后台ID，FFF是文件节点号。在每种情况下，在主文件（a/k/a 主分支）之外，每个表和索引有一个空闲空间映射（见第 68.3 节），它存储关系中可用空闲空间的信息。空闲空间映射存储在一个文件中，该文件以节点号加上后缀_fsm命名。表还有一个可见性映射，存储在一个后缀为_vm的分支中，它用于跟踪哪些页面已知含有非死亡元组。可见性映射将在第 68.4 节进一步描述。不被日志记录的表和索引还有第三个分支，即初始化分支，它存储在后缀为_init的分支中（见第 68.5 节。

> **小心**
>
> 请注意，虽然一个表的文件节点通常和它的 OID 相匹配，但实际上并不必须如此； 有些操作，比如 TRUNCATE、REINDEX、CLUSTER以及某些形式的ALTER TABLE，都可以改变文件节点而同时保留 OID。我们不应该假设文件节点和表 OID 相同。此外，对于包含pg_class本身在内的特定系统目录，其pg_class.relfilenode包含0。这些目录的实际文件节点号被存储在一个低层数据结构中，并且可以使用pg_relation_filenode()函数获取。

在表或者索引超过 1GB 之后，它就被划分成1G大小的段。 第一个段的文件名和文件节点相同；随后的段被命名为 filenode.1、filenode.2等等。这样的安排避免了在某些有文件大小限制的平台上的问题（实际上，1GB只是默认的段尺寸。段尺寸可以在编译PostgreSQL时使用配置选项--with-segsize进行调整）。原则上，空闲空间映射和可见性映射分支也可以要求多个段，但实际上这很少发生。

如果一个表的列中可能存储相当大的项，那么该表就会有个与之相关联的TOAST表， 它用于存储无法保留在在表行中的域值的线外存储。如果表有TOAST表，该表的pg_class.reltoastrelid链接到它的TOAST表。 参阅第 68.2 节获取更多信息。

表和索引的内容在第 68.6 节进一步讨论。

表空间的情况更复杂些。每个用户定义的表空间都在PGDATA/pg_tblspc目录里面有一个符号连接，它指向物理的表空间目录（就是在CREATE TABLESPACE命令里指定的那个目录）。这个符号连接是用表空间的 OID 命名的。在物理表空间目录中有一个名称取决于PostgreSQL服务器版本的子目录，例如PG_9.0_201008051（使用该子目录的原因是后续版本的数据库可以使用CREATE TABLESPACE指定相同的目录位置而不会造成冲突）。在这个版本相关的子目录中，为每个在这个表空间里有元素的数据库都有一个子目录， 以数据库的OID命名。该目录里的表和索引遵循文件节点命名模式。pg_default不需要通过pg_tblspc来访问，而是对应于PGDATA/base。类似地，pg_global表空间也不通过pg_tblspc访问，而是对应于PGDATA/global。

pg_relation_filepath()函数显示任何关系的完整路径（相对于PGDATA）。它可以作为记住上面这么多规则的替代方法。但是记住该函数只给出关系的主分支的第一个段的名称 — 你也许需要追加一个段号和/或_fsm、_vm或者_init来找到与该关系相关的所有文件。

临时文件（用于如排序不能放在内存中的数据等操作）被创建在PGDATA/base/pgsql_tmp中，如果临时文件被指定在一个非pg_default表空间中则它们会被创建在该表空

间的pgsql_tmp子目录中。临时文件的名称的形式为pgsql_tmpPPP.NNN，其中PPP是其所属后端的PID，而NNN用于区别该后端的不同临时文件。

68.2. TOAST

本节提供TOAST的概述（超尺寸属性存储技术－The Oversized-Attribute Storage Technique）。

PostgreSQL使用固定的页面尺寸（通常是8kB)，并且不允许元组跨越多个额页面。因此不可能直接存储非常大的域值。为了克服这个限制，大的域值会被压缩并/或分解成多个物理行。这些处理对用户都是透明的，只是在大部分的后端代码上有一些小的影响。这个技术的昵称是TOAST（或者“切片面包之后的最好的东西”）。TOAST 机制也被用来提升内存中大型数据值的处理。

只有特定的数据类型支持TOAST — 我们没必要在那些不可能生成大域值的数据类型上强加这种负担。要支持TOAST，数据类型必须有变长（varlena）的表现形式， 通常在存储的值中，头四个字节表示值的总长度（包括长度本身，以字节计）。TOAST并不约束该数据类型的表达的剩余部分。这种特殊的表达被统称为已TOAST值， 对它们的操作都必须通过修改或者重新解释这个初始长度字来进行。因此，支持一种可TOAST数据类型的 C 函数必须要小心它们可能会处理被TOAST过的输入值： 一个输入值可能并不真正由一个四字节长度和内容构成，直到它被反TOAST（通常是在对一个输入值做任何事情之前，先调用PG_DETOAST_DATUM； 但是在某些情况下也存在更高效的方法，详见第 38.12.1 节。

TOAST占用使用变长类型的长度字的最高两个二进制位（ 大端法机器上的高位，小端法机器上的低位）， 这样就把任何可TOAST值的 逻辑长度限制在1GB（2^{30} - 1字节）。如果两个位都是零， 那么数值是该数据类型一个普通的未TOAST的值，并且长度字的剩余位给出整 个数据以字节计的大小（包括长度字）。当最高位或者最低位被设置时，该值 只是有一个单字节头部而不是通常的四字节头部，并且该字节的剩余位数给出 了以字节计的总数据尺寸（包括长度字节）。这种节省空间的方案支持对低于 127 字节的值的存储，不过需要时仍然允许数据类型增长到 1GB。带有单字节 头部的值不会按照任何特别的边界对齐，反之带有四字节头部的值会按照至少 一个四字节边界对齐。这种对齐填充的省略额外地节省了空间，这种节省比起 短值来说更加显著。作为一种特殊情况，如果一个单字节头部的剩余位全是零 （对于一个自包含的长度来说是不可能的），该值就是一个线外数据的指针， 这就可能有下文所述的几种可能的情况。这样一个TOAST指针 的类型和尺寸由该数据的第二个字节中存储的一个代码决定。最后，如果最高 位或最低位被清除而另一位被设置，则表示该数据的内容被压缩过并且在使用 前必须先解压。在这种情况中四字节长度字的剩余位指出了压缩过的数据的大 小，而不是原始数据的大小。注意对于线外数据也可能存在压缩，但是变长数 据的头部不会告诉我们压缩是否发生 — TOAST指针 的内容将说明这个问题。

如前所述，有多种类型的TOAST指针数据。最古老且最常见的类型是 指向存储在一个TOAST表中的线外数据 的指针，TOAST 表与包含该 TOAST指针数据本身的表是相关的，但两者又是被分离存储的。当 一个要被存储在磁盘上的元组过大时，这些磁盘上的指针数据由 TOAST管理代码（在access/heap/tuptoaster.c 中）所创建。第 68.2.1 节给出了更多的细节。或者，一个TOAST指针数据能够包含一个出现在内存中某处的线外 数据的指针。这种数据必定是短命的并且将不会出现在磁盘上，但是它们对于避免 大型数据值的复制和冗余处理非常有用。详见 第 68.2.2 节

线内或者线外压缩数据所使用的压缩技术是 LZ 压缩技术家族中一种 相对简单且非常快速的成员。详见 src/common/pg_lzcompress.c。

68.2.1. 线外磁盘上 TOAST 存储

如果一个表中有任何一个列是可以TOAST的， 那么该表将有一个与之关联的TOAST表，其 OID 存储在表的pg_class.reltoastrelid项中。磁盘上的被TOAST过的值保存在TOAST表里，下文有更详细的描述。

线外值被分裂成（如果压缩过，在压缩之后分裂）最大为TOAST_MAX_CHUNK_SIZE（默认情况下该值应选为使得四个块（chunk）行能放在一个页面中，这个数值大约为2000 字节）字节

的块。每个块都作为独立的行存储在从属于所属表的TOAST表中。每个TOAST表都有列chunk_id（一个表示特定的被TOAST过的数据的OID）、chunk_seq（一个序列号，存储该块在值中的位置）和一个chunk_data（该块的实际数据）。在chunk_id和chunk_seq上有一个唯一索引，提供对值的快速检索。因此，一个表示线外磁盘上TOAST过的值的指针数据应存储要查看的TOAST表的OID以及指定值的OID（它的chunk_id）。为了方便，指针数据还存储逻辑数据的尺寸（原始的未压缩的数据长度）以及物理存储的尺寸（如果应用了压缩，则两者不同）。 加上变长数据头部的字节，一个磁盘上TOAST指针数据的总尺寸是18字节，不管它代表的值的实际长度是多大。

TOAST管理代码只有在准备向一个表中存储超过TOAST_TUPLE_THRESHOLD字节（通常是2kB）的行值的时候才会触发。TOAST代码将压缩和/或线外存储域值，直到行值比TOAST_TUPLE_TARGET字节（通常也是2kB）短，或者无法得到更好的结果的时候才停止。在一个 UPDATE 操作过程中，未改变的域的值通常原样保存； 所以，如果 UPDATE 一个带有线外值的行时，假如线外值没有变化，那么将不会产生TOAST开销。

TOAST代码代码识别四种不同的在磁盘上存储可TOAST列的策略：

- PLAIN避免压缩或者线外存储；而且它禁用变长类型的单字节头部。这是不可TOAST数据类型列的唯一可能的策略。只是对那些不能TOAST的数据类型才有可能。

- EXTENDED允许压缩和线外存储。这是大多数可TOAST数据类型的默认策略。 首先将尝试进行压缩，如果行仍然太大，那么则进行线外存储。

- EXTERNAL允许线外存储，但是不许压缩。使用EXTERNAL将令那些在宽text和 bytea列上的子串操作更快（代价是增加了存储空间），因此这些操作被优化为只抓取未压缩线外数据中需要的部分。

- MAIN允许压缩，但不允许线外存储（实际上，在这样的列上仍然会进行线外存储，但只是作为没有办法把行变得足以放入一页的情况下的最后手段）。

每个可TOAST的数据类型都为该数据类型的列指定了一个缺省策略， 但是一个给定表的列的存储策略可以用ALTER TABLE ... SET STORAGE修改。

可以使用ALTER TABLE ... SET (toast_tuple_target = N)为每个表调整TOAST_TUPLE_TARGET

这个方法比那些更直接的方法（比如允许行值跨越多个页面）有更多优点。 假设查询通常是用相对比较短的键值进行匹配的，那么执行器的大多数工作都将使用主行项完成。TOAST过的属性的大值只是在把结果集发送给客户端的时候才被抽出来（如果它被选中）。 因此，主表要小得多，并且它的能放入到共享缓冲区中的行要比没有任何线外存储的方案更多。 排序集也缩小了，并且排序将更多地在内存里完成。一个小测试表明，一个典型的保存 HTML 页面以及它们的 URL 的表占用的存储（包括TOAST表在内）大约只有裸数据的一半，而主表只包含全部数据的 10%（URL和一些小的 HTML 页面）。与在一个非TOAST的对照表里面存储（把全部 HTML 页面裁剪成 7Kb 以匹配页面大小）同样的数据相比，运行时没有任何区别。

68.2.2. 线外内存中 TOAST 存储

TOAST指针可以指向不在磁盘上但在当前服务器进程内存中 的数据。这样的指针显然不是长期存在的，但是它们是有用的。当前有两种 子情况：指向间接数据的指针以及指向 扩展数据的指针。

间接TOAST指针指向存储在内存中某个地方的非间接 varlena 值。这种情况仅仅作为一种概念验证而创建，但是当前它被用来在逻辑解码期间 避免创建超过 1GB 的物理元组（把所有线外域值都拉入元组就会这样）。这种 情况用处有限，因为该指针数据的创建者需要负责确保只要指针存在，被引用数 据就应该存在，并且没有其他设施来帮助它。

扩展的TOAST指针对于复杂数据类型有用，这些数据类型的磁盘上 表示形式不是特别适合计算性的目的。例如，一个PostgreSQL 数组的标准 varlena 表达包括了维度信息、一个空值位图（如果有任何空值元素），然后按顺序是所有元素的值。当元素类型本身是变长时，

找到第N 个元素的唯一方式是扫描所有在它前面的元素。这种表达适合于磁盘上的存储，因为它 很紧凑。但是为了对该数组进行计算，则“扩展”或者“结构”表 达会更好，这些表达中所有元素的开始位置都会被标记出来。为了支持这种需要， TOAST指针机制通过允许一个传引用的数据指向一个标准 varlena 值（磁盘上的表达）或者一个TOAST指针指向内存中某处的一个扩展 表达。这种扩展表达的细节取决于数据类型，不过它必须具有一个标准的头部并且符合 src/include/utils/expandeddatum.h中给定的其他 API 要求。该数据 类型的 C-级别函数可以选择处理任一表达。不了解扩展表达但简单地在其输入上应用PG_DETOAST_DATUM的函数将自动地接收到传统的 varlena 表达。 因此对于一种扩展表达的支持可以被增量式地引入，一次一个函数。

扩展值的TOAST指针会被进一步分解成 read-write和read-only指针。两种方式下被 指向的表达是相同的，但是收到一个读写指针的函数被允许就地修改被引用值， 而接收到只读指针的函数则不能。如果后者想要做一个该值的被修改的版本， 它必须先创建一个副本。这种区分和一些相关的惯例使得可以在查询执行期间 避免不必要的扩展值副本。

对于所有类型的内存中TOAST指针，TOAST管理 代码会确保这类指针数据不会意外地被存储在磁盘上。在存储之前内存中 TOAST指针会被自动地扩展成通常的线内 varlena 值 — 然后可能会被转换成磁盘上的TOAST指针（如果包含的元组不是太大）。

68.3. 空闲空间映射

每一个堆和索引关系（除了哈希索引）都有一个空闲空间映射（FSM）来保持对关系中可用空间的跟踪。它伴随着主关系数据被存储在一个独立的关系分支中，以关系的文件节点号加上一个_fsm后缀命名。例如，如果一个关系的文件节点是12345，那么FSM被存储在一个名为12345_fsm的文件中，该文件与主关系文件在同一个目录中。

空闲空间映射被组织成一棵FSM页面的树。底层FSM页面存储了在每一个堆（或索引）页面中可用的空闲空间，对于每一个这样的页面使用一个字节来表示。上层FSM页面则聚集来自于下层页面的信息。

在每一个FSM页面中是一个二叉树，存储在一个数组中，每一个节点一个字节。每个叶节点表示一个堆页面或者一个下层FSM页面。在每一个非叶节点中存储了它孩子节点中的最大值。因此叶节点中的最大值被存储在根中。

关于如何构建、更新和搜索FSM的更多信息请参考src/backend/storage/freespace/README。pg_freespacemap模块可以用来检查存储在空闲空间映射中的信息。

68.4. 可见性映射

每一个堆关系都有一个可见性映射（VM）用来跟踪哪些页面 只包含已知对所有活动事务可见的元组，它也跟踪哪些页面只包含 未被冻结的元组。它伴随着主关系数据被存储在一个独立的关系分支中， 以该关系的文件节点号加上一个_vm后缀来命名。例如， 如果一个关系的文件节点为12345，其VM被存储在名为12345_vm的文件中， 该文件域主关系文件在同一个目录中。注意索引没有VM。

可见性映射仅为每个堆页面存储两个位。第一位如果被设置， 表示该页面上的元组都是可见的，或者换句话说该页面不含有 任何需要被清理的元组。这些信息也可以被index-only scans用来只依靠索引元组回答查询。第二位如果 被设置，表示该页面上的元组都已经被冻结。这也意味着防回卷 清理操作也不需要重新访问该页面。

该映射是保守的，我们可以确定不论何时一个位被设置，那就说明条件为真，但是如果一个位没有被设置，它可能为真也可能不为真。可见性映射的位只会被清理操作设置，但是可以被任何在页面上进行的数据修改操作清除。

pg_visibility模块可以被用来检查存储在可见性 映射中的信息。

68.5. 初始化分支

每一个不被日志记录的表以及在这类表上的每一个索引，都有一个初始化分支。初始化分支是一个适当类型的空表或空索引。当一个不被日志记录的表由于崩溃必须被重置为空时，初始化分支被随着主分支复制，而任何其他分支则被擦除（它们会在需要时自动被重建）。

68.6. 数据库页面布局

本章提供一个PostgreSQL的表和索引所使用的页面格式的概述。[1] 序列和TOAST表的格式就像一个普通表一样。

在下面解释中，一个字节被假定包含 8 个位。 另外，术语item指的是存储在一个页面里的独立数据值。 在一个表里，一个项是一个行；在一个索引里，一个项是 一条索引记录。

每个表和索引都以一个固定尺寸（通常是 8kB，不过在编译服务器时可以选择其他不同的尺寸）的页面数组存储。 在表中，所有页面在逻辑上都相同，所以一个特定的项（行）可以被存储在任何页面里。 在索引里，第一个页面通常保留为元页来保存控制信息， 并且依索引访问方法的不同，在索引里可能有不同类型的页面。

表 68.2显示一个页面的总体布局。每个页面有五个部分。

表 68.2. 总体页面布局

项	描述
PageHeaderData	24字节长。包含关于页面的一般信息，包括空闲空间指针。
ItemIdData	一个记录（偏移量，长度）对的数组，指向实际项。每个项 4 字节。
Free space	未分配的空间（空闲空间）。新项指针从这个区域的开头开始分配，新项从其结尾开始分配。
Items	实际的项本身。
Special space	索引访问模式相关的数据。不同的索引访问方式存放不同的数据。在普通表中为空。

每个页面的头24个字节组成页头（PageHeaderData）。它的格式在表 68.3里详细介绍。 第一个域跟踪与此页面相关的最近的 WAL 项。第二个域包含该页面的校验码（如果data checksums被启用）。接下来一个2字节的域包含标志位。此后跟随着三个 2 字节的整数域（pd_lower、pd_upper和pd_special）。 这些域包含从页面开始位置到未分配空间开头的字节偏移、到未分配空间结尾的字节偏移以及到特殊空间开头的字节偏移。页面头中再接下来的 2 字节（pd_pagesize_version）存储页面尺寸和一个版本指示器。从PostgreSQL 8.3开始的版本号为4；PostgreSQL 8.1和8.2使用版本号3；PostgreSQL 8.0 使用版本号 2；PostgreSQL 7.3 和 7.4 使用版本号 1； 更早的版本使用版本号 0（基本页面布局和头格式在大部分这些版本里都没有改变，但是堆的行头部布局有所变化）。 页面大小主要用于交叉检查；目前在一次安装里，还不能支持多于一种页面大小。最后的域是一个提示，它显示删除该页是否可能获益：它跟踪在页面上最老的未删除的XMAX。

表 68.3. PageHeaderData布局

域	类型	长度	描述
pd_lsn	PageXLogRecPtr	8 bytes	LSN: 最后修改这个页面的WAL记录最后一个字节后面的第一个字节

[1] 实际上，索引访问模式并不需要使用这种页面格式。目前所有的索引方法的确都使用这个基本格式， 但索引元页里的数据通常并不遵循项布局规则）。

域	类型	长度	描述
pd_checksum	uint16	2 bytes	页面校验码
pd_flags	uint16	2 bytes	标志位
pd_lower	LocationIndex	2 bytes	到空闲空间开头的偏移量
pd_upper	LocationIndex	2 bytes	到空闲空间结尾的偏移量
pd_special	LocationIndex	2 bytes	到特殊空间开头的偏移量
pd_pagesize_version	uint16	2 bytes	页面大小和布局版本号信息
pd_prune_xid	TransactionId	4 bytes	页面上最老未删除XMAX，如果没有则为0

所有细节都可以在src/include/storage/bufpage.h中找到。

在页头后面是项标识符（ItemIdData），每个占用四个字节。 一个项标识符包含一个到项开头的字节偏移量（它的长度以字节计）， 以及一些属性位，这些属性位影响对它的解释。 新的项标识符根据需要从未分配空间的开头分配。项标识符的数目可以通过查看pd_lower来判断，在分配新标识符的时候pd_lower会增长。因为一个项标识符在被释放前绝对不会移动，所以它的索引可以用于长期地引用一个项， 即使该项本身因为压缩空闲空间在页面内部进行了移动。实际上，PostgreSQL创建的每个指向项的指针（ItemPointer，也叫做CTID）都由一个页号和一个项标识符的索引组成。

项本身存储在从未分配空间末尾开始从后向前分配的空间里。它们的实际结构取决于表包含的内容。表和序列都使用一种叫做 HeapTupleHeaderData的结构，如下文所述。

最后一部分是“特殊部分”，它可以包含访问方法想存放的任何东西。比如，b-tree 索引用它存储指向页面的左右兄妹的链接，以及其他一些和索引结构相关的数据。普通表并不使用这个部分（通过设置pd_special等于页面大小来表示）。

68.6.1. 表行布局

所有表行都用同样方法构造。它们有一个定长的头部（在大多数机器上占据 23 个字节），后面跟着一个可选的空值位图、一个可选的对象 ID 域以及用户数据。 该头部在表 68.里详细描述。实际的用户数据（行的列）从t_hoff指示的偏移位置开始， 它必须总是该平台的 MAXALIGN 距离的倍数。空值位图只有在t_infomask中的HEAP_HASNULL位被设置时存在。 如果存在，那么它紧跟在定长的头部后面，并占据足够的字节来容纳每个数据列对应的一个位（也就是说，总共t_natts位）。 在这个位的列表中，为 1 的位表示非空，而为 0 的位表示空。如果这个位图不存在，那么所有列都被假设为非空的。对象 ID 只有在设置了 t_infomask里面的HEAP_HASOID位时才存在。 如果存在，它正好出现在t_hoff边界之前。如果需要对齐t_hoff使之成为 MAXALIGN 的倍数，那么填充将出现在空值位图和对象 ID 之间（这样也保证了对象 ID 得到恰当的对齐）。

表 68.4. HeapTupleHeaderData布局

域	类型	长度	描述
t_xmin	TransactionId	4 bytes	插入XID标志
t_xmax	TransactionId	4 bytes	删除XID标志
t_cid	CommandId	4 bytes	插入和/或删除CID标志（覆盖t_xvac）
t_xvac	TransactionId	4 bytes	VACUUM操作移动一个行版本的XID

域	类型	长度	描述
t_ctid	ItemPointerData	6 bytes	当前版本的TID或者指向更新的行版本
t_infomask2	uint16	2 bytes	一些属性，加上多个标志位
t_infomask	uint16	2 bytes	多个标志位
t_hoff	uint8	1 byte	到用户数据的偏移量

所有细节都可以在src/include/access/htup_details.h中找到。

只有从其他表里获取了信息之后才能对确切数据进行，这些信息大多数在pg_attribute中。 标识域位置的关键值是attlen和attalign。 我们没有办法直接获取某个特定属性，除非它们是定宽并且没有空值。所有这些复杂的操作都封装在函数heap_getattr、fastgetattr和heap_getsysattr中。

要读取数据的话，你需要轮流检查每个属性。首先根据空值位图检查该域是否为NULL。 如果是，那么跳到下一个。然后保证你的对齐是正确的。如果域是一个定宽域，那么所有字节都简单地放在其中。 如果它是一个变长域（attlen = -1），那么它就会有点复杂。所有变长数据类型都使用一个通用的头部结构struct varlena， 它包含所存储的值的总长度以及一些标志位。根据标志的不同，数据可能在线内或者是在一个TOAST中，还可能是压缩的（参阅第 68.2 节。

第 69 章 系统目录声明和初始内容

PostgreSQL使用很多不同的系统目录来跟踪数据库对象（例如表和函数）的存在以及属性。系统目录和普通用户表之间在物理上没有什么不同，但是后端的C代码知道每一个目录的结构和属性，并且能够在较低的层次上直接操纵它们。因此，不建议尝试在运行中修改目录的结构，那样做会破坏内建在C代码中对目录行如何放置的设想。但是目录的结构可能会在主版本之间发生变化。

目录的结构声明在特殊格式的C头文件中，它们位于源码树的src/include/catalog/目录中。特别地，对每一个目录都有一个以其名称命名的头文件（例如，pg_class的头文件是pg_class.h），头文件定义了目录具有的列集合，以及一些其他的基本属性（例如OID）。其他定义目录结构的重要文件包括indexing.h和toasting.h，前者定义所有系统目录上的索引，而后者为需要TOAST表的目录定义TOAST表。

很多目录都有初始数据，这些数据必须在initdb的“bootstrap”阶段装载到对应的目录中，这样才能让系统达到能够执行SQL命令的状态点（例如，pg_class.h必须包含表示其自身的一个项，还要为每个系统目录和索引都分别包含一项）。这些初始数据以可编辑的形式保存在src/include/catalog/目录下的数据文件中。例如，pg_proc.dat描述了所有必须被插入到pg_proc目录的初始行。

为了创建目录文件并且将这些初始数据载入其中，一个以bootstrap模式运行的后端会读取包含着命令和初始数据的BKI（后端接口）文件。这种模式中用到的postgres.bki文件正是在编译PostgreSQL时从前述的头文件和数据文件准备而来，这一过程由名为genbki.pl的Perl脚本负责。尽管postgres.bki与特定PostgreSQL发行版相关，但它是平台无关的并且被安装在安装树的share子目录中。

genbki.pl还会为每个目录产生一个头文件，例如为pg_class生成的头文件是pg_class_d.h。这个文件含有自动生成的宏定义，并且可能包含其他的宏、枚举声明，因此对于读取特定目录的客户端C代码很有用。

大部分Postgres的开发者不需要直接与BKI文件打交道，但是几乎在后端中增加任何非平凡的特性都需要修改目录头文件或者初始数据文件。本章的剩余部分会给出一些相关的信息，并且将会完整地描述BKI文件格式。

69.1. 系统目录声明规则

一个目录头文件的关键部分是一个C的结构定义，它描述该目录中每一行的布局。这个结构开始于一个CATALOG宏，它对于C编译器而言只不过是typedef struct FormData_catalogname的一个简写。该结构中的每一个域会导致出现一个目录列。域可以用genbki.h中描述的BKI属性宏进行标注，例如可以为域定义默认值或者把域标记为可以为空或者不能为空。CATALOG行也可以用genbki.h中描述的一些其他BKI属性宏标注，用于定义该目录整体的其他属性，例如是否有OID（默认是有的）。

系统目录缓冲代码（以及大部分目录功能代码）假定所有的系统目录元组的定长部分是实际的存在形式，因为它会把这个C结构声明映射到定长部分之上。因此，所有变长域和可以为空的域必须被放置在最后，并且不能够以结构的域的方式访问。例如，如果尝试设置pg_type.typrelid为NULL，当某段代码尝试引用typetup->typrelid（或者更糟糕的是引用typetup->typelem，因为它跟随在typrelid之后）时将会出现失败。这会导致随机错误乃至段错误。

作为对这类错误的一种部分保护，变长或可以为空的域不应该对C编译器可见。通过将它们包裹在#ifdef CATALOG_VARLEN ... #endif（其中CATALOG_VARLEN是一个永不被定义的符号）中可以实现这一点。这能防止C代码不小心尝试访问可能不在那里或者可能在其他某个偏移位置的域。作为一种防止创建不正确行的措施，我们要求所有应该为非空的列在pg_attribute中也被标记为非空。如果目录列是定长的并且前面没有任何可以为空的列，bootstrap代码将自动把它标记为NOT NULL。在这一规则不适用的地方，可以根据需要使用BKI_FORCE_NOT_NULL和BKI_FORCE_NULL标注强制正确的标记。但是要注意的是，NOT

NULL约束仅在执行器中被强制，而不会针对随机C代码产生的元组进行强制，因此在手工创建或者更新目录行时仍需谨慎。

前端代码不应该包括任何pg_xxx.h目录头文件，因为这些文件可能包含在后端之外无法编译的C代码（通常，这是因为这些文件还包含src/backend/catalog/文件中函数的声明）。不过，前端代码可以包括相应的pg_xxx_d.h头文件，它将包含OID #define以及任何其他可能要在客户端使用的数据。如果希望前端代码能看到目录头文件中的宏或者其他代码，可以在相应部分的周围写上#ifdef EXPOSE_TO_CLIENT_CODE ... #endif，这样会指示genbki.pl把相应的部分拷贝到pg_xxx_d.h头文件中。

少数目录是非常基础的，以至于它们无法用大部分目录采用的BKI create命令来创建，因为那个命令需要在这些目录中写入信息来描述新的目录。这些目录被称为bootstrap目录，定义一个这样的目录需要一些额外的工作：开发者必须为它在pg_class和pg_type的预装载内容中手工准备合适的项，并且后续对该目录结构的更改将会更新那些项（bootstrap目录还需要pg_attribute中的预装载项，但是幸运地是现如今的genbki.pl会处理这些杂务）。如果可能，一定避免将新目录创建为bootstrap目录。

69.2. 系统目录初始数据

每个有手工创建的初始数据（有些没有）的目录都有一个相应的.dat文件，其中以可编辑的格式包含着该目录的初始数据。

69.2.1. 数据文件格式

每个.dat文件含有Perl数据结构文本，它可以简单地通过eval产生由一个哈希引用数组构成的内存数据结构，每个目录行一个。从pg_database.dat摘出的经过略微修改的一小部分可以展示关键特性：

```
[

# A comment could appear here.
{ oid => '1', oid_symbol => 'TemplateDbOid',
  descr => 'database\'s default template',
  datname => 'template1', datdba => 'PGUID', encoding => 'ENCODING',
  datcollate => 'LC_COLLATE', datctype => 'LC_CTYPE', datistemplate => 't',
  datallowconn => 't', datconnlimit => '-1', datlastsysoid => '0',
  datfrozenxid => '0', datminmxid => '1', dattablespace => '1663',
  datacl => '_null_' },

]
```

需要注意的点：

- 总体的文件布局是：开方括号，一个或者多个花括号集合（每一个表示一个目录行），闭方括号。在每一个闭花括号之后写一个逗号。

- 在每个目录行内，写成逗号分隔的key => value对。允许的key是该目录的列名，外加上元数据键oid、oid_symbol以及descr（oid和oid_symbol的使用在下文的第 69.2.2 节描述。descr为该对象提供一个描述字符串，它将被插入到pg_description或pg_shdescription中）。虽然元数据键是可选的，但是目录中定义的列必须全部被提供，除非目录的.h文件为该列指定了默认值。

- 所有的值都必须被放在单引号中。用反斜线可以转义值中用到的单引号。作为数据的反斜线可以（但是不必）被双写，这遵循的是Perl对简单引用文本的规则。注意，作为数据出现的反斜线将被bootstrap扫描器根据转义字符串常量的相同规则（见第 4.1.2.2 节）当作转义处理。例如\t转换为一个制表符。如果在最终值中确实想要一个反斜线，则需要写成四个：Perl会剥离掉两个，留下\\给bootstrap扫描器。

- 空值被表示为_null_（注意没有办法创建就是该字符串的值）。
- 注释以#开头，并且必须位于它们自己的行上。
- 为了帮助可读性，是其他目录项OID的域值可以用名称而不是数字的OID表示。这会在下文的第 69.2.3 节描述。
- 因为哈希是无序的数据结构，域顺序和行布局并不重要。不过，为了维持一种一致的外貌，我们设定了一些规则，它们由格式化脚本reformat_dat_file.pl实施：
  - 在每一对花括号内，元数据域oid、oid_symbol和descr（如果存在）按照这个顺序放在最前面，然后以定义时的顺序放上该目录自己的域。
  - 如果可能，根据需要在域之间插入新行以限制行的长度低于80字符。在元数据域和普通域之间也插入一个新行。
  - 如果目录的.h文件为一个列指定了默认值并且一个数据项具有相同的值，reformat_dat_file.pl将从数据文件中省去它。这能使得数据表达紧凑。
  - reformat_dat_file.pl原样保留空行和注释行。

  推荐在提交目录数据补丁前运行reformat_dat_file.pl。为了方便起见，可以简单地更改src/include/catalog/并且运行make reformat-dat-files。
- 如果想要增加一种新方法让数据表达更小，必须在reformat_dat_file.pl中实现该方法并且还要教会Catalog::ParseData()如何将数据展开回完整的表达。

69.2.2. OID分配

通过写一个oid => nnnn元数据域，出现在初始数据中的目录行可以被给予一个手工分配的OID。此外，如果分配一个OID，可以通过书写一个oid_symbol => name元数据域为该OID创建一个C宏。

如果预装载的目录行被其他预装载行用OID引用，则必须给它们预先分配OID。如果行的OID必须被C代码引用，也需要预分配的OID。如果两种情况都不符合，则oid元数据域可以被省略，在这种情况下bootstrap代码会自动分配OID，如果是一个没有OID的目录则将OID留为零。实际上对于一个给定的目录，即便其中某些行实际并没有被交叉引用，我们也通常会为其中预装载的行全部预分配OID或者全部不分配OID。

在C代码中写出任何OID的实际数字值是一种非常糟糕的形式，通常应该使用宏。对pg_proc OID的直接引用太常见了，因此有一种特别的机制自动创建必需的宏，见src/backend/utils/Gen_fmgrtab.pl。类似地 — 但是由于历史原因，实现的方式不同 — 也有一种自动的为pg_type OID创建宏的方法。因此在这两个目录中，oid_symbol项不是必需的。同样，系统目录和索引的pg_class OID的宏是自动设置的。对于所有其他系统目录，开发者必需通过oid_symbol项手动指定所需的宏。

要为一个新的预装载行找到一个可用的OID，可以运行脚本src/include/catalog/unused_oids。它能打印出未被使用的OID的闭区间范围（例如，输出行“45-900”表示OID 45到900都还没有被分配出去）。当前，OID 1-9999被保留给手工分配，unused_oids脚本会简单地查看目录头部以及.dat文件来看看哪些OID没有出现。也可以使用duplicate_oids脚本来检查错误（genbki.pl还会在编译时检测重复的OID）。

在bootstrap运行开始时，OID计数器从10000开始。如果表中的一个目录行要求OID但没有通过oid域预分配OID，那么它将得到一个大于等于10000的OID。

69.2.3. OID引用查找

从一个初始目录行到另一个初始目录行的交叉引用只需要在引用行中写上被引用行的预分配OID就可以实现。但这种方式容易出错并且难于理解，因此对于频繁引用的目录，genbki.pl提供了机制支持符号化的引用。当前这些机制可以用来引用访问方法、函数、操作符、操作符类、操作符族以及类型。规则如下：

- 通过对特定的目录列定义附加BKI_LOOKUP(lookuprule)来开启对符号化引用的使用，其中lookuprule是pg_am、pg_proc、pg_operator、pg_opclass、pg_opfamily或者pg_type。BKI_LOOKUP可以被附加到类型为oid、regproc、oidvector或者Oid[]的列上，在后两种情况中它意味着在数组的每个元素上执行查找。
- 在这样的一个列中，所有的项必须使用符号化格式，不过写成0（表示InvalidOid）除外（如果列被声明为regproc，可以选择用-代替0）。对于无法识别的名称，genbki.pl将会告警。
- 访问方法就用它们的名称表示，这和类型一样。类型的名称必须匹配被引用的pg_type项的typname，不能使用任何别名，例如用integer来替代int4。
- 函数可以用其proname来表示，前提是它在pg_proc.dat项中是唯一的（这和regproc输入类似）。否则，要将函数写成proname(argtypename,argtypename,...)，就像regprocedure那样。参数的类型名称必须被拼写准确，和它们在pg_proc.dat项的proargtypes域中的值一致。不要插入任何空白。
- 操作符的名称由oprname(lefttype,righttype)表示，类型的名称要写得准确，与它们出现在pg_operator.dat项的oprleft和oprright域中的值一样（对于一元操作符省略的操作数，可以写成0）。
- 操作符类和操作符族的名称仅在一个访问方法中唯一，因此它们用access_method_name/object_name表示。
- 在这些情况中都不能有方案限定，所有在bootstrap期间创建的对象都应该出现在pg_catalog方案中。

genbki.pl在运行时会解决所有符号化引用并且把简单的数字OID放到输出的BKI文件中。因此不需要bootstrap后端处理符号化引用。

69.2.4. 编辑数据文件的方法

在更新目录数据文件时，对于执行常用任务的简便方法，这里有一些建议。

向一个目录增加一个带有默认值的新列：. 用BKI_DEFAULT(value)标注将列增加到头文件中。数据文件的调整仅需要在要求非默认值的现有行中增加该域即可。

为没有默认值的现有列增加默认值：. 在头文件中增加一个BKI_DEFAULT标注，然后运行make reformat-dat-files以移除现在变得冗余的域项。

移除一列（不管有默认值还是没有）：. 从头文件中移除该列，然后运行make reformat-dat-files以移除现在无用的域项。

更改或者移除现有的默认值：. 不能简单地更改头文件，因为这将会导致当前的数据被不正确地解读。首先运行make expand-dat-files用显式插入所有默认值的形式重写数据文件，然后更改或者移除BKI_DEFAULT标注，然后运行make reformat-dat-files移除多余的域。

临时批量编辑：. 可以修改reformat_dat_file.pl执行很多种批量更改。寻找其中展示可以插入一次性代码的注释块。在下面的例子中，我们将把pg_proc中的两个boolean域联合成一个char域：

1. 在pg_proc.h中增加一个带有默认值的新列：

```
+    /* see PROKIND_ categories below */
+    char        prokind BKI_DEFAULT(f);
```

2. 基于reformat_dat_file.pl创建一个新脚本以插入合适的值：

```
-           # At this point we have the full row in memory as a hash
```

```
-           # and can do any operations we want. As written, it only
-           # removes default values, but this script can be adapted to
-           # do one-off bulk-editing.
+           # One-off change to migrate to prokind
+           # Default has already been filled in by now, so change to other
+           # values as appropriate
+           if ($values{proisagg} eq 't')
+           {
+               $values{prokind} = 'a';
+           }
+           elsif ($values{proiswindow} eq 't')
+           {
+               $values{prokind} = 'w';
+           }
```

3. 运行新的脚本：

```
$ cd src/include/catalog
$ perl  rewrite_dat_with_prokind.pl  pg_proc.dat
```

到这里pg_proc.dat拥有所有三个列prokind、proisagg以及proiswindow，不过它们将只出现在它们有非默认值的行中。

4. 从pg_proc.h移除旧的列：

```
-    /* is it an aggregate? */
-    bool        proisagg BKI_DEFAULT(f);
-
-    /* is it a window function? */
-    bool        proiswindow BKI_DEFAULT(f);
```

5. 最后，运行make reformat-dat-files从pg_proc.dat中移除无用的旧项。

用于批量编辑的脚本的更多例子，请参考这个消息https://www.postgresql.org/message-id/CAJVSVGVX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5DOO3ShfnM6jehA@mail.gmail.com的附件convert_oid2name.pl和remove_pg_type_oid_symbols.pl。

69.3. BKI文件格式

本节描述PostgreSQL后端如何解释BKI文件。结合一份实际的postgres.bki文件，本节的内容将会更容易理解。

BKI输入由一个命令序列组成。根据命令的语法，命令由一系列记号构成。记号之间通常由空白分隔，但是在没有歧义时也可不用。没有什么特殊的命令分隔符；语法上无法属于前一命令的记号将开始新的一条命令（通常你会把一个新命令放在一个新行上以保持清晰）。记号可以是某些关键字、特殊字符（圆括弧，逗号等）、数字或者双引号字串。 所有东西都是大小写敏感的。

以#开头的行会被忽略。

69.4. BKI命令

create tablename tableoid [bootstrap] [shared_relation] [without_oids] [rowtype_oid oid] (name1 = type1 [FORCE NOT NULL | FORCE NULL] [, name2 = type2 [FORCE NOT NULL | FORCE NULL], ...])

创建一个叫做tablename，OID为tableoid的表，它的列在圆括弧中给出。

bootstrap.c直接支持下列列类型：bool、bytea、char（1 字节）、name、int2、int4、regproc、regclass、regtype、text、oid、tid、xid、cid、int2vector、oi组）、_text（数组）、_oid（数组）、_char（数组）、_aclitem（数组）。尽管我们可以创建包含其它类型列的表， 但是我们只有在创建完pg_type并且填充了合适的记录之后才行（这实际上就意味着在自举目录中，只能使用这些列类型，而非自举目录可以使用任意内置类型）。

如果声明了bootstrap，那么该表将只在磁盘上创建；不会向pg_class、pg_attribute等表里面输入任何与该表相关的东西。因此这样的表将无法被普通的SQL操作访问，直到那些记录被用硬办法（用insert命令）建立。 这个选项用于创建pg_class等表本身。

如果声明了shared_relation，那么表就作为共享表创建。除非声明了without_oids，否则表将会有OID。表的行类型OID（pg_type的OID）可以有选择性地通过rowtype_oid子句指定。如果没有指定，会为之自产生一个OID（如果bootstrap被指定，则rowtype_oid是无效的，但不管怎样它还是被写在了文档中）。

open tablename

打开名为tablename的表进行数据插入。任何当前打开的表将被关闭。

close [tablename]

关闭打开着的表。给出的表名用于交叉检查，但并不是必须的。

insert [OID = oid_value] (value1 value2 ...)

用value1、value2 等作为列值以及oid_value作为其 OID向打开的表插入一条新记录。如果oid_value为零（0）或者该子句被忽略而表可以具有OID，则会为之赋予下一个可用的OID。

NULL 可以用特殊的关键字_null_指定。看起来不像标识符或者数字字符串的值必须被加上双引号。

declare [unique] index indexname indexoid on tablename using amname (opclass1 name1 [, ...])

在名为tablename的表上用amname访问方法创建一个OID为indexoid的名为indexname的索引。 索引的域被称为name1、name2等，而使用的操作符类分别是opclass1, opclass2等。 该命令将会创建索引文件和适当的系统目录项，但是索引内容不会被此命令初始化。

declare toast toasttableoid toastindexoid on tablename

为名为tablename的表创建一个TOAST表。该TOAST表将被赋予由toasttableoid表示的OID，且它的索引将被赋予由toastindexoid表示的OID。和declare index一样，索引的填充将被推迟。

build indices

填充之前声明的索引。

69.5. 自举BKI文件的结构

在open命令打开某个表时，它需要系统中已经存在一些表并且其中要具有与被打开表相关的项，在这些先决条件满足之前，open命令不能被使用（这些至少应该存在的表是pg_class、pg_attribute、pg_proc和pg_type）。 为了允许这些表本身被填充，带着bootstrap选项的create将会隐式打开所创建的表用于插入数据。

同样，declare index和declare toast命令也必须在相关系统目录被创建和填充之后才能被使用。

因此，postgres.bki文件的结构必须是：

1. create bootstrap其中一个关键表

2. insert数据，这些数据至少要能描述这些关键表

3. close

4. 重复创建其他关键表。

5. create（不带bootstrap）一个非关键表

6. open

7. insert需要的数据

8. close

9. 重复创建其他非关键表。

10.定义索引和TOAST表。

11.build indices

无疑还有其它未被文档记录的顺序依赖关系。

69.6. BKI例子

下面的命令集将创建名为test_table的表，它有两个列cola和colb，类型分别为int4和text的表，且表的OID为420， 然后向该表插入两行：

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

第 70 章 规划器如何使用统计信息

本章建立在第 14.1 和第 14.2 节讨论的材料之上，展示了关于规划器如何使用系统统计信息来估计一个查询的各个部分可能返回的行数。这是规划处理中的一个重要的部分，它提供了代价计算中的很多原始材料。

本章的目的不是详细地解释代码，而是给出一个规划器如何使用统计信息的概述。这样可能可以降低那些想以后阅读这部份代码的人的学习曲线。

70.1. 行估计例子

下面展示的例子使用PostgreSQL回归测试数据库中的表。输出结果是从版本 8.3 获得。之前（或之后）版本的动作可能会有所变化。同时还要注意的是，由于ANALYZE使用随机采样来产生统计信息，在任何新的ANALYZE之后结果将有轻微改变。

让我们从一个很简单的查询开始：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

规划器如何判断tenk1的势在第 14.2 节介绍，但为了完整还会在这里重复介绍。行数或页数是从pg_class中查出来的：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

这些数字是在表上的最后一次VACUUM或ANALYZE以来的当前值。 之后，规划器取出该表中实际的当前页数（这个操作的开销很小，不需要扫描全表）。 如果与relpages不同，则对reltuples 进行相应的缩放以得到一个当前的行数估计。在上面的例子中， relpages的值是最新的， 因此行估计与reltuples相同。

换一个在WHERE子句中带有范围条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器检查WHERE子句条件，并在pg_operator中查找<操作符的选择度函数。这被保持在oprrest列中， 并且在这个例子中的项是scalarltsel。 scalarltsel函数从pg_statistic为unique1检索直方图。 对于手工查询来说，查看更简单的pg_stats视图会更方便：

```
SELECT histogram_bounds FROM pg_stats
```

```
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

然后，把直方图里面被“< 1000”占据的部分找出来。这就是选择度。直方图把范围分隔成等频的桶， 所以我们要做的只是把我们的值所在的桶找出来，然后计数其中的部分以及所有该值之前的部分。值 1000 很明显在第二个桶（970-1943）中。假设每个桶中的值是线性分布，那么就可以计算出选择度：

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/
num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

也就是一整个桶加上第二个桶的线性部分，除以桶数。那么估计的行数现在可以用选择度乘以tenk1的势来计算：

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

然后让我们考虑一个在WHERE子句有等于条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

规划器还是检查WHERE子句条件，并为=查找选择度函数（这次是eqsel）。对于等值估计而言，直方图是没用的；相反，最常见值（MCV）列表可以用来决定选择度。让我们来看一下MCV，以及一些额外的后面用得上的列：

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  |
 {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs |
 {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

因为CRAAAA出现在 MCV 列表中，那么选择度只是最常见频度（MCF）列表中的一个对应项：

```
selectivity = mcf[3]
            = 0.003
```

像之前一样，行数的估计只是和前面一样用tenk1的势乘以选择度：

```
rows = 10000 * 0.003
```

```
     = 30
```

现在看看同样的查询，但是常量不在MCV列表中：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                           QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

这是完全不同的一个问题：当值不在MCV列表中时， 如何估计选择度。解决方法是利用该值不在列表中的事实，结合所有MCV出现的频率的知识：

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

也就是，把所有MCV的频度加起来并从 1 减去，然后除以其他可区分值的个数。这相当于假设不是 MCV 中的列的部分均匀分布在所有其他可区分值上。 需要注意的是，这里没有空值，因此我们不需要担心这些（否则需要从分子中减去空值的部分）。估计的行数然后按照常规计算：

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

之前带有unique1 < 1000的例子是scalarltsel实际工作的过度简化。 现在我们已经看过了使用 MCV 的例子，可以增加一些具体细节了。 这个例子到目前为止是正确的，因为unique1是一个唯一列，它没有 MCV（显然， 没有一个值能比其他值更通用）。对一个非唯一列而言，通常会有直方图和 MCV 列表， 并且直方图不包括由 MCV 表示的那部分列。之所以这样做是因为可以得到更精确的估计。在这种情况下，scalarltsel直接应用条件（如“< 1000”）到 MCV 列表中的每个值，并且把那些条件判断为真的 MCV 的频度加起来。这对表中是 MCV 的那一部分给出了准确的选择度估计。然后以上述同样的方式使用直方图估计表中不是 MCV 的那部分的选择度，并且组合这两个数字来估计总的选择读。例如，考虑：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                           QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

我们已看到stringu1的 MCV 信息，这里是它的直方图：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
-------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

检查 MCV 列表，我们发现前 6 项满足条件stringu1 < 'IAAAAA'，而最后 4 项不满足， 所以 MCV 部分的选择度是：

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

累加所有的 MCF 也告诉我们由 MCV 表示的群体中的比例是 0.03033333，并且因此由直方图表示的 比例是 0.96966667（同样，没有空值，否则我们在这里必须排除它们）。我们可以看到值IAAAAA差不多落在第三个直方图桶的结尾。通过使用一些关于不同字符频率的相当漂亮的假设，规划器对小于IAAAAA的直方图群体部分得到估计值 0.298387。我们然后组合 MCV 和非 MCV 群体的估计：

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

在这个特别的例子中，来自 MCV 列表的纠正相当小，因为列分布实际上很平坦（统计显示这些特殊值比其它值更常见的原因大部分是由于抽样误差）。 在更典型的情况下某些值显著地比其它的更常见，这种复杂的处理过程有助于提高准确度，因为那些最常见值的选择度可以被准确地找到。

现在考虑一个WHERE子句中带有多个条件的情况：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                      QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器假定这两个条件是独立的，因此子句各自的选择度可以被乘在一起：

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

需要注意的是，从位图索引扫描中返回的估计行数只反映和索引一起使用的条件； 这一点很重要，因为它会影响后续取堆元组的代价估计。

最后我们将检查一个涉及连接的查询：

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
```

```
        Recheck Cond: (unique1 < 50)
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50
 width=0)
              Index Cond: (unique1 < 50)
  ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1
 width=244)
        Index Cond: (unique2 = t1.unique2)
```

在tenk1上的限制unique1 < 50在嵌套循环连接之前被计算。它的处理类似之前的那个范围查询例子。但是这次值 50 落在unique1直方图的第一个桶内：

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/
num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

连接的限制是t2.unique2 = t1.unique2。操作符是我们熟悉的=，然而选择度函数是从pg_operator的oprjoin列获得的，并且是eqjoinsel。eqjoinsel为tenk2和tenk1查找统计信息：

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

在这种情况中，没有unique2的MCV信息，因为所有值看上去都是唯一的，因此我们可以为关系和它们的空值部分使用一个只依赖可区分值数目的算法：

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/
num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

也就是说，从 1 中减去每个表的空值部分，并且除以可区分值的最大数目。连接可能发出的行数的计算是：嵌套循环里的两个输入值的笛卡尔积的势乘以选择度：

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

这里有两列的 MCV 列表，eqjoinsel将使用 MCV 列表的直接比较来决定在由 MCV 表示的列群体部分中的连接选择度。群体剩下部分的估计遵循这里展示的相同方法。

需要注意的是，我们把inner_cardinality显示为 10000，也就是未修改的tenk2尺寸。它可能出现于EXPLAIN输出检查，连接行的估计来自 50 * 1，即由 outer 行数乘以由tenk2上每个 inner 索引扫描的估计行数。但是这不是那种情况： 连接关系尺寸的估计在任何特定的连接计划被考虑之前进行。如果一切顺利，那么两种方式估计的连接尺寸将产生 大概同样的答案，但是由于舍入误差和其它因素它们有时差别显著。

如果对更进一步的细节感兴趣，一个表的尺寸（在任何WHERE子句之前）的估计在src/backend/optimizer/util/plancat.c中完成。子句选择度的一般逻辑在src/backend/

optimizer/path/clausesel.c中。操作符相关的选择度函数大部分可以在src/backend/utils/adt/selfuncs.c中找到。

70.2. 多变量统计例子

70.2.1. 函数依赖

多元相关性可以用一个非常简单的数据集来演示 — 一个有两列的表，它们都包含相同的值：

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

如第 14.2 所述，规划人员可以使用从 pg_class获取的页面和行数来确定 t的基数：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

他的数据分布非常简单；每列中只有100个不同的值，均匀分布。

以下示例显示了在a列上估计WHERE条件的结果：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

规划器检查条件并确定该子句的选择性为1%。通过比较这个估计值和实际的行数， 我们可以看到估计值非常准确（事实上，是因为表格非常小）。 更改WHERE条件以使用b列，将生成一个完全相同的计划。 但是观察一下，如果我们在两列上应用相同的条件，将它们用AND结合起来会发生什么：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

规划器分别估算每个条件的选择性，达到与上述相同的1%估计值。 然后它假定条件是独立的，因此它乘以它们的选择性，产生最终选择性估计值仅为0.01%。 这是一个明显的低估，因为符合条件（100）的实际行数要高于两个数量级。

通过创建一个指示ANALYZE 计算两列上的函数依赖性多变量统计信息的统计对象，可以解决此问题。

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
```

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

70.2.2. N 个不同变量的计数

估计多列集合的基数时会出现类似的问题，例如由GROUP BY 子句生成的组的数量。当GROUP BY列出单个列时， n个不同估计值（作为HashAggregate节点返回的估计行数可见）非常准确：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual rows=100
 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000
 loops=1)
```

但是，如果没有多变量统计信息，那么在GROUP BY 中包含两列的查询中的组数量估计值将在下面的示例中偏离一个数量级：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual rows=100
 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

通过重新定义统计对象以包括两列的n个不同值的计数，估计得到了很大改进：

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual rows=100
 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

70.3. 规划器统计和安全

对表pg_statistic的访问仅限于超级用户， 以便普通用户无法从中了解其他用户的表的内容。 一些选择性估算函数将使用用户提供的操作符（出现在查询中的操作符或相关操作

符） 来分析所存储的统计。例如，为了确定存储的最常用值是否适用， 选择性估计器将必须运行适当的= 运算符来将查询中的常量与存储的值进行比较。因此， pg_statistic中的数据可能会传递给用户定义的运算符。 适当制作的操作符可以故意泄漏传递的操作数（例如，通过记录它们或将它们写入不同的表）， 或者通过在错误消息中显示它们的值来意外泄漏它们，无论哪种情况都可能暴露 pg_statistic中的数据给一个不应该能够看到它的用户。

为了防止这种情况，以下内容适用于所有内置的选择性估计函数。在规划查询时， 为了能够使用存储的统计信息，当前用户必须在表或相关列上具有 SELECT权限，或者使用的操作符必须是 LEAKPROOF（更准确地说，是操作符所基于的函数）。 如果不是，那么选择性估计器将表现得好像没有统计数据可用， 并且规划器将继续进行默认或回退假设。

如果用户对表或列没有所需的权限，那么在很多情况下，查询最终会收到权限被拒绝的错误， 在这种情况下，这种机制在实践中是不可见的。但是，如果用户正在从安全屏障视图读取数据， 则规划器可能希望检查用户无法访问的基础表的统计数据。在这种情况下， 操作符应该是防漏的，否则统计数据将不会被使用。没有直接的反馈意见， 除非计划可能是次优的。如果有人怀疑是这种情况， 可以尝试将查询作为更有特权的用户来运行，以查看是否产生了不同的计划。

此限制仅适用于规划器需要对pg_statistic 中的一个或多个值执行用户定义的运算符的情况。因此，无论访问权限如何， 规划器都可以使用通用的统计信息，例如列中空值的比例或列中不同值的数量。

可能对用户定义的操作符进行统计操作的第三方扩展中包含的选择性估计函数， 应遵循相同的安全规则。请参考PostgreSQL源代码获取指导。

部分 VIII. 附录

目录

附录 A. PostgreSQL错误代码

PostgreSQL服务器发出的所有消息都被赋予了五个字符错误代码，这遵循 SQL 标准对“SQLSTATE”代码的习惯。需要知道发生了什么错误条件的应用通常应该测试错误代码，而不是查看文本形式的错误消息。这些错误代码轻易不会在PostgreSQL的版本之间改变，并且一般也不会随着错误消息的本地化而改变。请注意有些（但不是全部）PostgreSQL生成的错误代码是由 SQL 标准定义的；有些标准没有定义的额外错误代码是PostgreSQL发明的或者是从其它数据库借用的。

根据标准，错误代码的前两个字符表示错误类别，而后三个字符表示在该类别内的一种特定情况。因此， 那些不能识别特定错误代码的应用仍然可以从错误类别中推断要做什么。

表 A.1中列出了PostgreSQL 11.2定义的所有错误代码（有些实际上目前并没有使用，但是 SQL 标准中有定义）。错误类别也在其中列出。对于每个错误类别都有一个 “标准”错误代码，它的最后三个字符是000。这个代码只用于那些属于该类别但是没有被赋予更特定代码的错误情况。

“情况名称”列中显示的符号是在PL/pgSQL中使用的情况名称。情况名称可以被写成大写或小写形式（注意PL/pgSQL不识别警告（与错误不同）情况名称，它们是类别 00、01 和 02）。

对于某些类型的错误，服务器会报告与错误相关的数据库对象（一个表、表列、数据类型或约束）的名称。例如，导致一个unique_violation错误的唯一约束的名称。这些名字被包含在错误报告消息的独立域中，这样应用就不必从可能是本地化的人类可读的消息文本中抽取它们。到PostgreSQL 9.3 位置，对于这个特性的完全覆盖只在 SQLSTATE 类别 23 （完整性约束未被）的错误中存在，但是很可能会在未来进行扩展。

表 A.1. PostgreSQL错误代码

错误代码	情况名称
Class 00 — Successful Completion	
00000	successful_completion
Class 01 — Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure

错误代码	情况名称
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast

错误代码	情况名称
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation

错误代码	情况名称
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Class 26 — Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted

错误代码	情况名称
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always

错误代码	情况名称
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 — Insufficient Resources	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 — Object Not In Prerequisite State	

错误代码	情况名称
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 — System Error (errors external to PostgreSQL itself)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class 72 — Snapshot Failure	
72000	snapshot_too_old
Class F0 — Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV — Foreign Data Wrapper Error (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory

错误代码	情况名称
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

附录 B. 日期/时间支持

PostgreSQL使用一个内部的启发式解析器来进行所有日期/时间输入的支持。日期和时间被作为字符串输入，并且被分解为不同的域，这些域都已经预定义好了要存放哪一类信息。每个域会被解释并且被分配一个数字值或者被忽略、拒绝。该解析器为所有的文本形式的域都包含了内部查找表，包括月份、一周中的日和时区。

这个附录包括这些查找表内容的信息并且描述了解析器解码日期和时间所用的步骤。

B.1. 日期/时间输入解释

日期/时间类型输入使用下面的过程解码。

1. 将输入字符串打断成记号并且把每一个记号分类为一个字符串、时间、时区或数字。

   a. 如果数字记号包含一个分号（:），那么这是一个时间字符串。包括所有后续数位和分号。

   b. 如果数字记号包含一个连字符（-）、斜线（/）或两个以上的句点（.），那么这是一个日期字符串，它可能含有一个文本形式的月份。如果一个日期记号已经被看到，它会转而被解释为一个时区名（例如America/New_York）。

   c. 如果记号只是数字，那么它要么是一个单一域，要么是一个 ISO 8601 串连日期（例如 1999年1月13日是19990113）或时间（例如 14:15:16 是141516）。

   d. 如果记号以一个加号（+）或减号（-）开始，那么它要么是一个数字的时区，要么是一个特殊域。

2. 如果记号是一个字母字符串，使之匹配可能的串：

   a. 查看记号是否与任何已知的时区缩写匹配。 这些缩写由在第 B.4 节描述的配置文件提供。

   b. 如果没有找到，搜索一个内部表来查找将该记号是否匹配为一个特殊串（例如today）、日（例如Thursday）、月（例如January）或噪音词（例如at、on）。

   c. 如果仍然没有找到，则抛出一个错误。

3. 当记号是一个数字或数字域时：

   a. 如果有 8 位或 6 位，并且之前没有读到其他日期域，那么解释为一个“串连日期”（例如19990118或990118）。翻译是YYYYMMDD或YYMMDD。

   b. 如果记号是 3 位并且已经读到了一个年域，那么解释为一年中的第几日。

   c. 如果是 4 位或 6 位并且已经读到了一个年域，那么解释为一个时间域（HHMM或HHMMSS）。

   d. 如果是 3 位或更多位并且还没有读到日期域，解释为一个年域（这会强制剩余日期域的 yy-mm-dd 顺序）。

   e. 否则日期域顺序被假定为遵循DateStyle设置：mm-dd-yy、dd-mm-yy 或 yy-mm-dd。如果一个月或日域被发现超过范围，则抛出一个错误。

4. 如果已经指定了 BC ，对年求反并且加一用于内部存储（在格里高利历中没有 0 年，因此数字的 1 BC 就是 0 年）。

5. 如果没有指定 BC ，并且如果该年域长度为两位，那么把该年域调整为四位。如果该域小于 70，则增加 2000，否则增加 1900。

> **提示**
>
> 格里高利年 AD 1-99 可以使用带有前导零的 4 位形式录入（例如0099是AD 99）。

B.2. 处理无效或不明确的时间戳

通常，如果日期/时间字符串在语法上有效但包含 超出范围的字段值，将引发错误。 例如，输入2月31日将被拒绝。

在夏令时转换期间，看似有效的时间戳字符串可能表示不存在或不明确的时间戳。 这样的输入不会被拒绝;不确定性可以通过要应用哪个UTC偏移来解决。 例如，假设TimeZone参数设置为America/New_York，请考虑

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

因为那天是那个时区的春天过渡日期，所以没有民用时间凌晨2:30; 时钟从2AM EST跳转到3AM EDT。 PostgreSQL将给定时间解释为标准时间（UTC-5），然后呈现为3:30AM EDT（UTC-4）。

相反，请考虑后向过渡期间的行为:

```
=> SELECT '2018-11-04 02:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 02:30:00-05
(1 row)
```

在那一天，上午2:30有两种可能的解释; 2:30AM EDT，以及一小时以后，如果转换到标准时间，即2:30AM EST。 同样，PostgreSQL将给定时间解释为标准时间（UTC-5）。 我们可以通过指定夏令时来强行控制:

```
=> SELECT '2018-11-04 02:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

此时间戳可以有效地呈现为2:30 UTC-4或1:30 UTC-5; 时间戳输出代码选择后者。

在这些情况下应用的精确规则是，出现在前向跳转的夏令时转换中的无效时间戳被分配了转换之前的时区对应的UTC偏移; 可能落在后向跳转的两边的不确定的时间戳被分配了转换之后的时区对应的UTC偏移。 在大多数时区，这相当于说“在有疑问时，标准时间解释是首选”。

在所有情况下，可以显式使用数字UTC偏移或对应于固定UTC偏移的时区缩写明确指定与时间戳关联的UTC偏移。 刚刚给出的规则仅在需要推断偏移量变化的时区的UTC偏移时才适用。

B.3. 日期/时间关键词

表 B.展示了被识别为月份名称的记号。

表 B.1. 月份名称

月份	简写
一月	Jan
二月	Feb
三月	Mar
四月	Apr
五月	
六月	Jun
七月	Jul
八月	Aug
九月	Sep, Sept
十月	Oct
十一月	Nov
十二月	Dec

表 B.展示了被识别为一周内每一天的名称的记号。

表 B.2. 一周内每一天的名称

天	简写
周日	Sun
周一	Mon
周二	Tue, Tues
周三	Wed, Weds
周四	Thu, Thur, Thurs
周五	Fri
周六	Sat

表 B.展示了服务于多种修饰目的的记号。

表 B.3. 日期/时间域修饰语

标识符	描述
AM	12:00 之前的时间
AT	被忽略
JULIAN, JD, J	下一个域是儒略日期
ON	被忽略
PM	12:00 之后的时间
T	下一个域是时间

B.4. 日期/时间配置文件

因为时区缩写并未被很好地标准化，PostgreSQL提供了一种方法来自定义服务器所接受的缩写集合。timezone_abbreviations运行时参数决定活动的缩写集合。虽然这个参数可以被任何数据库用户修改，但它的可能值是受到数据库管理员的控制的 — 它们实际上是存储在安装目录的.../share/timezonesets/子目录中的一些配置文件。通过在那个目录中增加或修改文件，管理员可以为时区缩写设定本地策略。

timezone_abbreviations可以被设置为任何在.../share/timezonesets/中可以找到的文件名，前提该文件的名字完全是字母的（timezone_abbreviations中禁止非字母字符防止从预期目录的外面读取文件以及读取编辑器的备份文件和其他外部文件）。

一个时区缩写文件可以包含空行和以#开始的注释。非注释行必须具有下列格式之一：

```
zone_abbreviation offset
zone_abbreviation offset D
zone_abbreviation time_zone_name
@INCLUDE file_name
@OVERRIDE
```

一个zone_abbreviation就是被定义的缩写。offset是一个整数，它给出以秒计的到 UTC 的等效偏移量，为正表示东起格林威治，为负表示西起格林威治。例如， -18000 表示格林威治西边的五个小时，或者北美东海岸标准时间。D指示该区域名表示本地夏令时而非标准时间。另外，还可以给出一个time_zone_name，在这种情况下会查阅该时区定义，并且会使用该时区中的缩写含义。这种替代方案只用于那些含义在历史上有变化的缩写，因为比起使用一个固定的整数值，查找该含义开销要大得多。

另外，还可以给出一个time_zone_name，它引用 IANA 时区数据库中定义的时区名。这时会参考该时区的定义来判断在时区中是否有或者使用了该缩写。如果是，会使用适当的含义 — 也就是正在判断其值的时间戳中当前使用的含义，或者之前刚刚使用的含义（如果当时不是当前），或者最老的含义（如果只在那时之后用过）。这种行为对于处理其含义在历史上变化过的缩写是至关重要的。也允许按照缩写没有出现在其中的时区名来定义缩写，这样使用该缩写就等效于直接写出该时区名。

> **提示**
>
> 在定义其 UTC 偏移没有改变过的缩写时，使用简单的整数offset更好，这样的缩写在处理时代价比那些需要查阅时区定义的缩写更低。

@INCLUDE语法允许包括.../share/timezonesets/目录中 的其它文件。允许进行嵌套包括，但是嵌套深度有限制。

@OVERRIDE语法表示文件中后续项可以覆盖前面的项（典型的：从被包括的文件中 得到的项）。如果没有它，同一个时区缩写的相互冲突的定义会被认为是一种错误。

在一个未被修改的安装中，文件Default包含用于世界大部分地区的非冲突时区缩 写。附加文件Australia和India被提供给那些地区：这些文件 首先会包括Default文件，并且接着根据需要增加或修改缩写。

为了便于参考，标准安装也包含了Africa.txt、America.txt 等文件，它们包含了所有根据IANA 时区数据库中已知正在使用的时区缩写信息。 如果需要，这些文件中的时区名定义可以复制并粘贴到自定义的配置文件中。注意这些文件名 不能直接被timezone_abbreviations设置引用，因为它们的名称中嵌有句点。

> **注意**
>
> 如果在读取时区缩写集时发生错误，将不会应用任何新值并且保留旧的集合。如果该错误是在数据库 启动时发生，那么启动将失败。

> **小心**
>
> 配置文件中定义的时区缩写将会覆盖PostgreSQL中内建的非 时区含义。例如Australia配置文件定义了SAT（南澳洲标准时 间）。当该文件为活动时，SAT将不会被识别为周六的缩写。

> **小心**
>
> 如果你修改.../share/timezonesets/中的文件， 那么你必须自己创建备份 — 因为通常的数据库转储不会包括这个目录。

B.5. 单位的历史

SQL 标准说到“在一个‘日期时间文字’的定义中，‘日期时间值’根据格里高利历被日期和时间的自然规则所约束”。PostgreSQL遵循 SQL 标准，导致只在格里高利历内计算日期，即使对于该历法开始使用之前的日期也是如此。这个规则被称作外推格里高利历。

儒略日期是由 Julius Caesar 在公元前 45 年引入的。直到 1582 年开始转为使用公历之前，西方世界一直使用儒略日期。 在儒略日期中，一回归年近似等于 365 又 1/4 天 = 365.25 天。它大约在128年中会出现 1 天的误差。

不断积累的历法错误促使教皇格里高利十三世按照特伦托会议的指示改革了历法。在格里高利历中，一回归年近似为 365 + 97/400 天 = 365.2425 天。因此对于格里高利历，大约要 3300 年一回归年才会积累一天的误差。

近似值 365+97/400 是通过利用下面的规则，并规定每 400 年有 97 个闰年实现的：

每个可被 4 整除的年是一个闰年。
不过，可被 100 整除的年不是闰年。
但是，可以被 400 整除的年还是闰年。

因此，1700、1800、1900、2100 和 2200 都不是闰年。而 1600、2000、2400 是闰年。相比而言，旧式的儒略历法里面只有能被4整除的年是闰年。 相反，在旧的儒略历法中所有能被 4 整除的年都是闰年。

罗马教皇在 1582 年 2 月宣布从 1582 年的 10 月中减除 10 天，这样 10 月 15 日就紧跟在 10 月 4 日的后面。意大利、波兰、葡萄牙和西班牙遵守了这个要求。其他天主教的国家也紧跟它们的步伐。但新教国家拒绝改变，而希腊东正教国家却一直拖延到 20 世纪开始时才逐渐遵守这个规定。 大英帝国及其殖民地（包含今天的美国）在 1752 年开始遵守这项改革。因此 1752 年 9 月 2 日之后紧跟着 14 日。 这就是为什么 Unix 系统上的cal程序 会产生如下输出的原因：

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

不过，这种历法只对大英帝国及其殖民地有效，对其他地方无效。因为尝试在多个地方多个时刻使用的实际历法很难并且也很让人困惑，PostgreSQL并没有做这种尝试，而是对所有日期遵循格里高利历规则，即使该方法在历史上是不精确的。

在世界的多个地方，发展了不同的历法，很多比格里高利系统还早。 例如，中国历法的开端可以追溯到公元前 14 世纪。传说黄帝在公元前 2637 年就发明了这个历法。 中华人民共和国把格里高利历作为民用。中国历法则被用于决定节日/节气。

儒略日期不是另一种类型的历法，虽然在命名上很相似，但是它和儒略历法无关。儒略日期系统是法国学者 Joseph Justus Scaliger（1540-1609）（可能是取自其父亲的名字，即意大利学者 Julius Caesar Scaliger（1484-1558））发明的。 在儒略日期系统中，每天都有一个序数，从 JD 0 开始（有时被叫做那个儒略日期）。 JD 0 在儒略历法中对应公元前 4713 年 1 月 1 日，或者在格里高利历中对应公元前 4714 年 12 月 24 日。儒略日期计数经常被天文学家用来标注夜间观测，并且因此一个日期就是从一个正午 UTC 到下一个正午 UTC，而不是从午夜到另一个午夜：JD 0 设计的 24 小时是从公元前 4714 年 12 月 24 日的正午 UTC 到公元前 4714 年 12 月 25 日的正午 UTC。

尽管PostgreSQL在输入输出日期时支持儒略日期符号（并且也用在一些内部的日期时间计算上使用儒略日期），它不遵守从正午到正午。PostgreSQL把儒略日期当作是从午夜到午夜。

附录 C. SQL关键词

表 C.1列出了在SQL标准以及PostgreSQL 11.2中作为关键词的所有记号。背景资料可以在第 4.1.1 节中找到（由于篇幅的缘故，只包括了SQL标准的最近两个版本以及用于与历史比较的SQL-92。这些版本以及其他中间标准的版本之间的差别很小）。

SQL区分保留关键词和非保留关键词。根据标准，保留关键词才是真正的关键词，它们绝不会被允许作为标识符。非关键词仅仅是在特定上下文中具有特殊的含义并且可以在其他上下文中被用作标识符。大部分非保留关键词实际上是SQL指定的内建表和内建函数的名字。非保留关键词的概念存在的意义上实际上是声明某些上下文中的一个词被附加了某种预定义的含义。

在PostgreSQL的解析器中情况更加复杂。其中有多种不同的记号分类，从那些决不能被用作标识符的加号到那些在解析器中与普通标识符比起来绝对没有特殊状态的记号（后者通常是SQL中指定的函数）。在PostgreSQL中甚至保留关键词也不是完全被保留的，而是可以被用作列标签（例如可以写SELECT 55 AS CHECK，虽然CHECK是一个保留关键词）。

在表 C.1的PostgreSQL列中，我们把解析器明确知道但允许作为列名或者表名的那些关键词分类为“非保留”。有一些关键词是非保留的，但是不能被用作函数或数据类型名称，因此它们会被标记（大部分这些词表示有特殊语法的内建函数或数据类型。这种函数或类型仍然可用，但是不能被用户重新定义）。不允许作为列名或表名的记号被打上“保留”的标签。某些保留关键词被允许作为函数或数据类型的名字，这也显示在该表中。如果没有被那样标记，保留关键词仅被允许作为“AS”列的标签名。

作为一条一般性的规则，如果对包含所列出关键词作为标识符的命令得到了站不住脚的解析器错误，应该尝试将该标识符加上引号来看看是否能解决问题。

在学习表 C.1之前有一件重要的事情是理解一个在PostgreSQL中不被保留的关键字并不意味着与该词相关的特性没有被实现。反过来，一个关键词的存在也不表示相应特性的存在。

表 C.1. SQL关键词

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
A		非保留	非保留	
ABORT	非保留			
ABS		保留	保留	
ABSENT		非保留	非保留	
ABSOLUTE	非保留	非保留	非保留	保留
ACCESS	非保留			
ACCORDING		非保留	非保留	
ACTION	非保留	非保留	非保留	保留
ADA		非保留	非保留	非保留
ADD	非保留	非保留	非保留	保留
ADMIN	非保留	非保留	非保留	
AFTER	非保留	非保留	非保留	
AGGREGATE	非保留			
ALL	保留	保留	保留	保留
ALLOCATE		保留	保留	保留
ALSO	非保留			
ALTER	非保留	保留	保留	保留
ALWAYS	非保留	非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ANALYSE	保留			
ANALYZE	保留			
AND	保留	保留	保留	保留
ANY	保留	保留	保留	保留
ARE		保留	保留	保留
ARRAY	保留	保留	保留	
ARRAY_AGG		保留	保留	
ARRAY_MAX_CARDINALITY		保留		
AS	保留	保留	保留	保留
ASC	保留	非保留	非保留	保留
ASENSITIVE		保留	保留	
ASSERTION	非保留	非保留	非保留	保留
ASSIGNMENT	非保留	非保留	非保留	
ASYMMETRIC	保留	保留	保留	
AT	非保留	保留	保留	保留
ATOMIC		保留	保留	
ATTACH	非保留			
ATTRIBUTE	非保留	非保留	非保留	
ATTRIBUTES		非保留	非保留	
AUTHORIZATION	保留（可以是函数或类型）	保留	保留	保留
AVG		保留	保留	保留
BACKWARD	非保留			
BASE64		非保留	非保留	
BEFORE	非保留	非保留	非保留	
BEGIN	非保留	保留	保留	保留
BEGIN_FRAME		保留		
BEGIN_PARTITION		保留		
BERNOULLI		非保留	非保留	
BETWEEN	非保留（不能是函数或类型）	保留	保留	保留
BIGINT	非保留（不能是函数或类型）	保留	保留	
BINARY	保留（可以是函数或类型）	保留	保留	
BIT	非保留（不能是函数或类型）			保留
BIT_LENGTH				保留
BLOB		保留	保留	
BLOCKED		非保留	非保留	
BOM		非保留	非保留	
BOOLEAN	非保留（不能是函数或类型）	保留	保留	
BOTH	保留	保留	保留	保留
BREADTH		非保留	非保留	
BY	非保留	保留	保留	保留
C		非保留	非保留	非保留

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CACHE	非保留			
CALL	非保留	保留	保留	
CALLED	非保留	保留	保留	
CARDINALITY		保留	保留	
CASCADE	非保留	非保留	非保留	保留
CASCADED	非保留	保留	保留	保留
CASE	保留	保留	保留	保留
CAST	保留	保留	保留	保留
CATALOG	非保留	非保留	非保留	保留
CATALOG_NAME		非保留	非保留	非保留
CEIL		保留	保留	
CEILING		保留	保留	
CHAIN	非保留	非保留	非保留	
CHAR	非保留（不能是函数或类型）	保留	保留	保留
CHARACTER	非保留（不能是函数或类型）	保留	保留	保留
CHARACTERISTICS	非保留	非保留	非保留	
CHARACTERS		非保留	非保留	
CHARACTER_LENGTH		保留	保留	保留
CHARACTER_SET_CATALOG		非保留	非保留	非保留
CHARACTER_SET_NAME		非保留	非保留	非保留
CHARACTER_SET_SCHEMA		非保留	非保留	非保留
CHAR_LENGTH		保留	保留	保留
CHECK	保留	保留	保留	保留
CHECKPOINT	非保留			
CLASS	非保留			
CLASS_ORIGIN		非保留	非保留	非保留
CLOB		保留	保留	
CLOSE	非保留	保留	保留	保留
CLUSTER	非保留			
COALESCE	非保留（不能是函数或类型）	保留	保留	保留
COBOL		非保留	非保留	非保留
COLLATE	保留	保留	保留	保留
COLLATION	保留（可以是函数或类型）	非保留	非保留	保留
COLLATION_CATALOG		非保留	非保留	非保留
COLLATION_NAME		非保留	非保留	非保留
COLLATION_SCHEMA		非保留	非保留	非保留
COLLECT		保留	保留	
COLUMN	保留	保留	保留	保留
COLUMNS	非保留	非保留	非保留	
COLUMN_NAME		非保留	非保留	非保留
COMMAND_FUNCTION		非保留	非保留	非保留

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
COMMAND_FUNCTION_CODE		非保留	非保留	
COMMENT	非保留			
COMMENTS	非保留			
COMMIT	非保留	保留	保留	保留
COMMITTED	非保留	非保留	非保留	非保留
CONCURRENTLY	保留（可以是函数或类型）			
CONDITION		保留	保留	
CONDITION_NUMBER		非保留	非保留	非保留
CONFIGURATION	非保留			
CONFLICT	非保留			
CONNECT		保留	保留	保留
CONNECTION	非保留	非保留	非保留	保留
CONNECTION_NAME		非保留	非保留	非保留
CONSTRAINT	保留	保留	保留	保留
CONSTRAINTS	非保留	非保留	非保留	保留
CONSTRAINT_CATALOG		非保留	非保留	非保留
CONSTRAINT_NAME		非保留	非保留	非保留
CONSTRAINT_SCHEMA		非保留	非保留	非保留
CONSTRUCTOR		非保留	非保留	
CONTAINS		保留	非保留	
CONTENT	非保留	非保留	非保留	
CONTINUE	非保留	非保留	非保留	保留
CONTROL		非保留	非保留	
CONVERSION	非保留			
CONVERT		保留	保留	保留
COPY	非保留			
CORR		保留	保留	
CORRESPONDING		保留	保留	保留
COST	非保留			
COUNT		保留	保留	保留
COVAR_POP		保留	保留	
COVAR_SAMP		保留	保留	
CREATE	保留	保留	保留	保留
CROSS	保留（可以是函数或类型）	保留	保留	保留
CSV	非保留			
CUBE	非保留	保留	保留	
CUME_DIST		保留	保留	
CURRENT	非保留	保留	保留	保留
CURRENT_CATALOG	保留	保留	保留	
CURRENT_DATE	保留	保留	保留	保留
CURRENT_DEFAULT_TRANSFORM_GROUP		保留	保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CURRENT_PATH		保留	保留	
CURRENT_ROLE	保留	保留	保留	
CURRENT_ROW		保留		
CURRENT_SCHEMA	保留（可以是函数或类型）	保留	保留	
CURRENT_TIME	保留	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留	保留
CURRENT_TRANSFORM_GROUP_FOR_TYPE		保留	保留	
CURRENT_USER	保留	保留	保留	保留
CURSOR	非保留	保留	保留	保留
CURSOR_NAME		非保留	非保留	非保留
CYCLE	非保留	保留	保留	
DATA	非保留	非保留	非保留	非保留
DATABASE	非保留			
DATALINK		保留	保留	
DATE		保留	保留	保留
DATETIME_INTERVAL_CODE		非保留	非保留	非保留
DATETIME_INTERVAL_PRECISION		非保留	非保留	非保留
DAY	非保留	保留	保留	保留
DB		非保留	非保留	
DEALLOCATE	非保留	保留	保留	保留
DEC	非保留（不能是函数或类型）	保留	保留	保留
DECIMAL	非保留（不能是函数或类型）	保留	保留	保留
DECLARE	非保留	保留	保留	保留
DEFAULT	保留	保留	保留	保留
DEFAULTS	非保留	非保留	非保留	
DEFERRABLE	保留	非保留	非保留	保留
DEFERRED	非保留	非保留	非保留	保留
DEFINED		非保留	非保留	
DEFINER	非保留	非保留	非保留	
DEGREE		非保留	非保留	
DELETE	非保留	保留	保留	保留
DELIMITER	非保留			
DELIMITERS	非保留			
DENSE_RANK		保留	保留	
DEPENDS	非保留			
DEPTH		非保留	非保留	
DEREF		保留	保留	
DERIVED		非保留	非保留	
DESC	保留	非保留	非保留	保留
DESCRIBE		保留	保留	保留
DESCRIPTOR		非保留	非保留	保留

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
DETACH	非保留			
DETERMINISTIC		保留	保留	
DIAGNOSTICS		非保留	非保留	保留
DICTIONARY	非保留			
DISABLE	非保留			
DISCARD	非保留			
DISCONNECT		保留	保留	保留
DISPATCH		非保留	非保留	
DISTINCT	保留	保留	保留	保留
DLNEWCOPY		保留	保留	
DLPREVIOUSCOPY		保留	保留	
DLURLCOMPLETE		保留	保留	
DLURLCOMPLETEONLY		保留	保留	
DLURLCOMPLETEWRITE		保留	保留	
DLURLPATH		保留	保留	
DLURLPATHONLY		保留	保留	
DLURLPATHWRITE		保留	保留	
DLURLSCHEME		保留	保留	
DLURLSERVER		保留	保留	
DLVALUE		保留	保留	
DO	保留			
DOCUMENT	非保留	非保留	非保留	
DOMAIN	非保留	非保留	非保留	保留
DOUBLE	非保留	保留	保留	保留
DROP	非保留	保留	保留	保留
DYNAMIC		保留	保留	
DYNAMIC_FUNCTION		非保留	非保留	非保留
DYNAMIC_FUNCTION_CODE		非保留	非保留	
EACH	非保留	保留	保留	
ELEMENT		保留	保留	
ELSE	保留	保留	保留	保留
EMPTY		非保留	非保留	
ENABLE	非保留			
ENCODING	非保留	非保留	非保留	
ENCRYPTED	非保留			
END	保留	保留	保留	保留
END-EXEC		保留	保留	保留
END_FRAME		保留		
END_PARTITION		保留		
ENFORCED		非保留		
ENUM	非保留			

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
EQUALS		保留	非保留	
ESCAPE	非保留	保留	保留	保留
EVENT	非保留			
EVERY		保留	保留	
EXCEPT	保留	保留	保留	保留
EXCEPTION				保留
EXCLUDE	非保留	非保留	非保留	
EXCLUDING	非保留	非保留	非保留	
EXCLUSIVE	非保留			
EXEC		保留	保留	保留
EXECUTE	非保留	保留	保留	保留
EXISTS	非保留（不能是函数或类型）	保留	保留	保留
EXP		保留	保留	
EXPLAIN	非保留			
EXPRESSION		非保留		
EXTENSION	非保留			
EXTERNAL	非保留	保留	保留	保留
EXTRACT	非保留（不能是函数或类型）	保留	保留	保留
FALSE	保留	保留	保留	保留
FAMILY	非保留			
FETCH	保留	保留	保留	保留
FILE		非保留	非保留	
FILTER	非保留	保留	保留	
FINAL		非保留	非保留	
FIRST	非保留	非保留	非保留	保留
FIRST_VALUE		保留	保留	
FLAG		非保留	非保留	
FLOAT	非保留（不能是函数或类型）	保留	保留	保留
FLOOR		保留	保留	
FOLLOWING	非保留	非保留	非保留	
FOR	保留	保留	保留	保留
FORCE	非保留			
FOREIGN	保留	保留	保留	保留
FORTRAN		非保留	非保留	非保留
FORWARD	非保留			
FOUND		非保留	非保留	保留
FRAME_ROW		保留		
FREE		保留	保留	
FREEZE	保留（可以是函数或类型）			
FROM	保留	保留	保留	保留
FS		非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
FULL	保留（可以是函数或类型）	保留	保留	保留
FUNCTION	非保留	保留	保留	
FUNCTIONS	非保留			
FUSION		保留	保留	
G		非保留	非保留	
GENERAL		非保留	非保留	
GENERATED	非保留	非保留	非保留	
GET		保留	保留	保留
GLOBAL	非保留	保留	保留	保留
GO		非保留	非保留	保留
GOTO		非保留	非保留	保留
GRANT	保留	保留	保留	保留
GRANTED	非保留	非保留	非保留	
GREATEST	非保留（不能是函数或类型）			
GROUP	保留	保留	保留	保留
GROUPING	非保留（不能是函数或类型）	保留	保留	
GROUPS	非保留	保留		
HANDLER	非保留			
HAVING	保留	保留	保留	保留
HEADER	非保留			
HEX		非保留	非保留	
HIERARCHY		非保留	非保留	
HOLD	非保留	保留	保留	
HOUR	非保留	保留	保留	保留
ID		非保留	非保留	
IDENTITY	非保留	保留	保留	保留
IF	非保留			
IGNORE		非保留	非保留	
ILIKE	保留（可以是函数或类型）			
IMMEDIATE	非保留	非保留	非保留	保留
IMMEDIATELY		非保留		
IMMUTABLE	非保留			
IMPLEMENTATION		非保留	非保留	
IMPLICIT	非保留			
IMPORT	非保留	保留	保留	
IN	保留	保留	保留	保留
INCLUDE	非保留			
INCLUDING	非保留	非保留	非保留	
INCREMENT	非保留	非保留	非保留	
INDENT		非保留	非保留	
INDEX	非保留			

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
INDEXES	非保留			
INDICATOR		保留	保留	保留
INHERIT	非保留			
INHERITS	非保留			
INITIALLY	保留	非保留	非保留	保留
INLINE	非保留			
INNER	保留（可以是函数或类型）	保留	保留	保留
INOUT	非保留（不能是函数或类型）	保留	保留	
INPUT	非保留	非保留	非保留	保留
INSENSITIVE	非保留	保留	保留	保留
INSERT	非保留	保留	保留	保留
INSTANCE		非保留	非保留	
INSTANTIABLE		非保留	非保留	
INSTEAD	非保留	非保留	非保留	
INT	非保留（不能是函数或类型）	保留	保留	保留
INTEGER	非保留（不能是函数或类型）	保留	保留	保留
INTEGRITY		非保留	非保留	
INTERSECT	保留	保留	保留	保留
INTERSECTION		保留	保留	
INTERVAL	非保留（不能是函数或类型）	保留	保留	保留
INTO	保留	保留	保留	保留
INVOKER	非保留	非保留	非保留	
IS	保留（可以是函数或类型）	保留	保留	保留
ISNULL	保留（可以是函数或类型）			
ISOLATION	非保留	非保留	非保留	保留
JOIN	保留（可以是函数或类型）	保留	保留	保留
K		非保留	非保留	
KEY	非保留	非保留	非保留	保留
KEY_MEMBER		非保留	非保留	
KEY_TYPE		非保留	非保留	
LABEL	非保留			
LAG		保留	保留	
LANGUAGE	非保留	保留	保留	保留
LARGE	非保留	保留	保留	
LAST	非保留	非保留	非保留	保留
LAST_VALUE		保留	保留	
LATERAL	保留	保留	保留	
LEAD		保留	保留	
LEADING	保留	保留	保留	保留
LEAKPROOF	非保留			
LEAST	非保留（不能是函数或类型）			

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
LEFT	保留（可以是函数或类型）	保留	保留	保留
LENGTH		非保留	非保留	非保留
LEVEL	非保留	非保留	非保留	保留
LIBRARY		非保留	非保留	
LIKE	保留（可以是函数或类型）	保留	保留	保留
LIKE_REGEX		保留	保留	
LIMIT	保留	非保留	非保留	
LINK		非保留	非保留	
LISTEN	非保留			
LN		保留	保留	
LOAD	非保留			
LOCAL	非保留	保留	保留	保留
LOCALTIME	保留	保留	保留	
LOCALTIMESTAMP	保留	保留	保留	
LOCATION	非保留	非保留	非保留	
LOCATOR		非保留	非保留	
LOCK	非保留			
LOCKED	非保留			
LOGGED	非保留			
LOWER		保留	保留	保留
M		非保留	非保留	
MAP		非保留	非保留	
MAPPING	非保留	非保留	非保留	
MATCH	非保留	保留	保留	保留
MATCHED		非保留	非保留	
MATERIALIZED	非保留			
MAX		保留	保留	保留
MAXVALUE	非保留	非保留	非保留	
MAX_CARDINALITY			保留	
MEMBER		保留	保留	
MERGE		保留	保留	
MESSAGE_LENGTH		非保留	非保留	非保留
MESSAGE_OCTET_LENGTH		非保留	非保留	非保留
MESSAGE_TEXT		非保留	非保留	非保留
METHOD	非保留	保留	保留	
MIN		保留	保留	保留
MINUTE	非保留	保留	保留	保留
MINVALUE	非保留	非保留	非保留	
MOD		保留	保留	
MODE	非保留			
MODIFIES		保留	保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
MODULE		保留	保留	保留
MONTH	非保留	保留	保留	保留
MORE		非保留	非保留	非保留
MOVE	非保留			
MULTISET		保留	保留	
MUMPS		非保留	非保留	非保留
NAME	非保留	非保留	非保留	非保留
NAMES	非保留	非保留	非保留	保留
NAMESPACE		非保留	非保留	
NATIONAL	非保留（不能是函数或类型）	保留	保留	保留
NATURAL	保留（可以是函数或类型）	保留	保留	保留
NCHAR	非保留（不能是函数或类型）	保留	保留	保留
NCLOB		保留	保留	
NESTING		非保留	非保留	
NEW	非保留	保留	保留	
NEXT	非保留	非保留	非保留	保留
NFC		非保留	非保留	
NFD		非保留	非保留	
NFKC		非保留	非保留	
NFKD		非保留	非保留	
NIL		非保留	非保留	
NO	非保留	保留	保留	保留
NONE	非保留（不能是函数或类型）	保留	保留	
NORMALIZE		保留	保留	
NORMALIZED		非保留	非保留	
NOT	保留	保留	保留	保留
NOTHING	非保留			
NOTIFY	非保留			
NOTNULL	保留（可以是函数或类型）			
NOWAIT	非保留			
NTH_VALUE		保留	保留	
NTILE		保留	保留	
NULL	保留	保留	保留	保留
NULLABLE		非保留	非保留	非保留
NULLIF	非保留（不能是函数或类型）	保留	保留	保留
NULLS	非保留	非保留	非保留	
NUMBER		非保留	非保留	非保留
NUMERIC	非保留（不能是函数或类型）	保留	保留	保留
OBJECT	非保留	非保留	非保留	
OCCURRENCES_REGEX		保留	保留	
OCTETS		非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
OCTET_LENGTH		保留	保留	保留
OF	非保留	保留	保留	保留
OFF	非保留	非保留	非保留	
OFFSET	保留	保留	保留	
OIDS	非保留			
OLD	非保留	保留	保留	
ON	保留	保留	保留	保留
ONLY	保留	保留	保留	保留
OPEN		保留	保留	保留
OPERATOR	非保留			
OPTION	非保留	非保留	非保留	保留
OPTIONS	非保留	非保留	非保留	
OR	保留	保留	保留	保留
ORDER	保留	保留	保留	保留
ORDERING		非保留	非保留	
ORDINALITY	非保留	非保留	非保留	
OTHERS	非保留	非保留	非保留	
OUT	非保留（不能是函数或类型）	保留	保留	
OUTER	保留（可以是函数或类型）	保留	保留	保留
OUTPUT		非保留	非保留	保留
OVER	非保留	保留	保留	
OVERLAPS	保留（可以是函数或类型）	保留	保留	保留
OVERLAY	非保留（不能是函数或类型）	保留	保留	
OVERRIDING	非保留	非保留	非保留	
OWNED	非保留			
OWNER	非保留			
P		非保留	非保留	
PAD		非保留	非保留	保留
PARALLEL	非保留			
PARAMETER		保留	保留	
PARAMETER_MODE		非保留	非保留	
PARAMETER_NAME		非保留	非保留	
PARAMETER_ORDINAL_POSITION		非保留	非保留	
PARAMETER_SPECIFIC_CATALOG		非保留	非保留	
PARAMETER_SPECIFIC_NAME		非保留	非保留	
PARAMETER_SPECIFIC_SCHEMA		非保留	非保留	
PARSER	非保留			
PARTIAL	非保留	非保留	非保留	保留
PARTITION	非保留	保留	保留	
PASCAL		非保留	非保留	非保留
PASSING	非保留	非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
PASSTHROUGH		非保留	非保留	
PASSWORD	非保留			
PATH		非保留	非保留	
PERCENT		保留		
PERCENTILE_CONT		保留	保留	
PERCENTILE_DISC		保留	保留	
PERCENT_RANK		保留	保留	
PERIOD		保留		
PERMISSION		非保留	非保留	
PLACING	保留	非保留	非保留	
PLANS	非保留			
PLI		非保留	非保留	非保留
POLICY	非保留			
PORTION		保留		
POSITION	非保留（不能是函数或类型）	保留	保留	保留
POSITION_REGEX		保留	保留	
POWER		保留	保留	
PRECEDES		保留		
PRECEDING	非保留	非保留	非保留	
PRECISION	非保留（不能是函数或类型）	保留	保留	保留
PREPARE	非保留	保留	保留	保留
PREPARED	非保留			
PRESERVE	非保留	非保留	非保留	保留
PRIMARY	保留	保留	保留	保留
PRIOR	非保留	非保留	非保留	保留
PRIVILEGES	非保留	非保留	非保留	保留
PROCEDURAL	非保留			
PROCEDURE	非保留	保留	保留	保留
PROCEDURES	非保留			
PROGRAM	非保留			
PUBLIC		非保留	非保留	保留
PUBLICATION	非保留			
QUOTE	非保留			
RANGE	非保留	保留	保留	
RANK		保留	保留	
READ	非保留	非保留	非保留	保留
READS		保留	保留	
REAL	非保留（不能是函数或类型）	保留	保留	保留
REASSIGN	非保留			
RECHECK	非保留			
RECOVERY		非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
RECURSIVE	非保留	保留	保留	
REF	非保留	保留	保留	
REFERENCES	保留	保留	保留	保留
REFERENCING	非保留	保留	保留	
REFRESH	非保留			
REGR_AVGX		保留	保留	
REGR_AVGY		保留	保留	
REGR_COUNT		保留	保留	
REGR_INTERCEPT		保留	保留	
REGR_R2		保留	保留	
REGR_SLOPE		保留	保留	
REGR_SXX		保留	保留	
REGR_SXY		保留	保留	
REGR_SYY		保留	保留	
REINDEX	非保留			
RELATIVE	非保留	非保留	非保留	保留
RELEASE	非保留	保留	保留	
RENAME	非保留			
REPEATABLE	非保留	非保留	非保留	非保留
REPLACE	非保留			
REPLICA	非保留			
REQUIRING		非保留	非保留	
RESET	非保留			
RESPECT		非保留	非保留	
RESTART	非保留	非保留	非保留	
RESTORE		非保留	非保留	
RESTRICT	非保留	非保留	非保留	保留
RESULT		保留	保留	
RETURN		保留	保留	
RETURNED_CARDINALITY		非保留	非保留	
RETURNED_LENGTH		非保留	非保留	非保留
RETURNED_OCTET_LENGTH		非保留	非保留	非保留
RETURNED_SQLSTATE		非保留	非保留	非保留
RETURNING	保留	非保留	非保留	
RETURNS	非保留	保留	保留	
REVOKE	非保留	保留	保留	保留
RIGHT	保留（可以是函数或类型）	保留	保留	保留
ROLE	非保留	非保留	非保留	
ROLLBACK	非保留	保留	保留	保留
ROLLUP	非保留	保留	保留	
ROUTINE	非保留	非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ROUTINES	非保留			
ROUTINE_CATALOG		非保留	非保留	
ROUTINE_NAME		非保留	非保留	
ROUTINE_SCHEMA		非保留	非保留	
ROW	非保留（不能是函数或类型）	保留	保留	
ROWS	非保留	保留	保留	保留
ROW_COUNT		非保留	非保留	非保留
ROW_NUMBER		保留	保留	
RULE	非保留			
SAVEPOINT	非保留	保留	保留	
SCALE		非保留	非保留	非保留
SCHEMA	非保留	非保留	非保留	保留
SCHEMAS	非保留			
SCHEMA_NAME		非保留	非保留	非保留
SCOPE		保留	保留	
SCOPE_CATALOG		非保留	非保留	
SCOPE_NAME		非保留	非保留	
SCOPE_SCHEMA		非保留	非保留	
SCROLL	非保留	保留	保留	保留
SEARCH	非保留	保留	保留	
SECOND	非保留	保留	保留	保留
SECTION		非保留	非保留	保留
SECURITY	非保留	非保留	非保留	
SELECT	保留	保留	保留	保留
SELECTIVE		非保留	非保留	
SELF		非保留	非保留	
SENSITIVE		保留	保留	
SEQUENCE	非保留	非保留	非保留	
SEQUENCES	非保留			
SERIALIZABLE	非保留	非保留	非保留	非保留
SERVER	非保留	非保留	非保留	
SERVER_NAME		非保留	非保留	非保留
SESSION	非保留	非保留	非保留	保留
SESSION_USER	保留	保留	保留	保留
SET	非保留	保留	保留	保留
SETOF	非保留（不能是函数或类型）			
SETS	非保留	非保留	非保留	
SHARE	非保留			
SHOW	非保留			
SIMILAR	保留（可以是函数或类型）	保留	保留	
SIMPLE	非保留	非保留	非保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SIZE		非保留	非保留	保留
SKIP	非保留			
SMALLINT	非保留（不能是函数或类型）	保留	保留	保留
SNAPSHOT	非保留			
SOME	保留	保留	保留	保留
SOURCE		非保留	非保留	
SPACE		非保留	非保留	保留
SPECIFIC		保留	保留	
SPECIFICTYPE		保留	保留	
SPECIFIC_NAME		非保留	非保留	
SQL	非保留	保留	保留	保留
SQLCODE				保留
SQLERROR				保留
SQLEXCEPTION		保留	保留	
SQLSTATE		保留	保留	保留
SQLWARNING		保留	保留	
SQRT		保留	保留	
STABLE	非保留			
STANDALONE	非保留	非保留	非保留	
START	非保留	保留	保留	
STATE		非保留	非保留	
STATEMENT	非保留	非保留	非保留	
STATIC		保留	保留	
STATISTICS	非保留			
STDDEV_POP		保留	保留	
STDDEV_SAMP		保留	保留	
STDIN	非保留			
STDOUT	非保留			
STORAGE	非保留			
STRICT	非保留			
STRIP	非保留	非保留	非保留	
STRUCTURE		非保留	非保留	
STYLE		非保留	非保留	
SUBCLASS_ORIGIN		非保留	非保留	非保留
SUBMULTISET		保留	保留	
SUBSCRIPTION	非保留			
SUBSTRING	非保留（不能是函数或类型）	保留	保留	保留
SUBSTRING_REGEX		保留	保留	
SUCCEEDS		保留		
SUM		保留	保留	保留
SYMMETRIC	保留	保留	保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SYSID	非保留			
SYSTEM	非保留	保留	保留	
SYSTEM_TIME		保留		
SYSTEM_USER		保留	保留	保留
T		非保留	非保留	
TABLE	保留	保留	保留	保留
TABLES	非保留			
TABLESAMPLE	保留（可以是函数或类型）	保留	保留	
TABLESPACE	非保留			
TABLE_NAME		非保留	非保留	非保留
TEMP	非保留			
TEMPLATE	非保留			
TEMPORARY	非保留	非保留	非保留	保留
TEXT	非保留			
THEN	保留	保留	保留	保留
TIES	非保留	非保留	非保留	
TIME	非保留（不能是函数或类型）	保留	保留	保留
TIMESTAMP	非保留（不能是函数或类型）	保留	保留	保留
TIMEZONE_HOUR		保留	保留	保留
TIMEZONE_MINUTE		保留	保留	保留
TO	保留	保留	保留	保留
TOKEN		非保留	非保留	
TOP_LEVEL_COUNT		非保留	非保留	
TRAILING	保留	保留	保留	保留
TRANSACTION	非保留	非保留	非保留	保留
TRANSACTIONS_COMMITTED		非保留	非保留	
TRANSACTIONS_ROLLED_BACK		非保留	非保留	
TRANSACTION_ACTIVE		非保留	非保留	
TRANSFORM	非保留	非保留	非保留	
TRANSFORMS		非保留	非保留	
TRANSLATE		保留	保留	保留
TRANSLATE_REGEX		保留	保留	
TRANSLATION		保留	保留	保留
TREAT	非保留（不能是函数或类型）	保留	保留	
TRIGGER	非保留	保留	保留	
TRIGGER_CATALOG		非保留	非保留	
TRIGGER_NAME		非保留	非保留	
TRIGGER_SCHEMA		非保留	非保留	
TRIM	非保留（不能是函数或类型）	保留	保留	保留
TRIM_ARRAY		保留	保留	
TRUE	保留	保留	保留	保留

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
TRUNCATE	非保留	保留	保留	
TRUSTED	非保留			
TYPE	非保留	非保留	非保留	非保留
TYPES	非保留			
UESCAPE		保留	保留	
UNBOUNDED	非保留	非保留	非保留	
UNCOMMITTED	非保留	非保留	非保留	非保留
UNDER		非保留	非保留	
UNENCRYPTED	非保留			
UNION	保留	保留	保留	保留
UNIQUE	保留	保留	保留	保留
UNKNOWN	非保留	保留	保留	保留
UNLINK		非保留	非保留	
UNLISTEN	非保留			
UNLOGGED	非保留			
UNNAMED		非保留	非保留	非保留
UNNEST		保留	保留	
UNTIL	非保留			
UNTYPED		非保留	非保留	
UPDATE	非保留	保留	保留	保留
UPPER		保留	保留	保留
URI		非保留	非保留	
USAGE		非保留	非保留	保留
USER	保留	保留	保留	保留
USER_DEFINED_TYPE_CATALOG		非保留	非保留	
USER_DEFINED_TYPE_CODE		非保留	非保留	
USER_DEFINED_TYPE_NAME		非保留	非保留	
USER_DEFINED_TYPE_SCHEMA		非保留	非保留	
USING	保留	保留	保留	保留
VACUUM	非保留			
VALID	非保留	非保留	非保留	
VALIDATE	非保留			
VALIDATOR	非保留			
VALUE	非保留	保留	保留	保留
VALUES	非保留（不能是函数或类型）	保留	保留	保留
VALUE_OF		保留		
VARBINARY		保留	保留	
VARCHAR	非保留（不能是函数或类型）	保留	保留	保留
VARIADIC	保留			
VARYING	非保留	保留	保留	保留
VAR_POP		保留	保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
VAR_SAMP		保留	保留	
VERBOSE	保留（可以是函数或类型）			
VERSION	非保留	非保留	非保留	
VERSIONING		保留		
VIEW	非保留	非保留	非保留	保留
VIEWS	非保留			
VOLATILE	非保留			
WHEN	保留	保留	保留	保留
WHENEVER		保留	保留	保留
WHERE	保留	保留	保留	保留
WHITESPACE	非保留	非保留	非保留	
WIDTH_BUCKET		保留	保留	
WINDOW	保留	保留	保留	
WITH	保留	保留	保留	保留
WITHIN	非保留	保留	保留	
WITHOUT	非保留	保留	保留	
WORK	非保留	非保留	非保留	保留
WRAPPER	非保留	非保留	非保留	
WRITE	非保留	非保留	非保留	保留
XML	非保留	保留	保留	
XMLAGG		保留	保留	
XMLATTRIBUTES	非保留（不能是函数或类型）	保留	保留	
XMLBINARY		保留	保留	
XMLCAST		保留	保留	
XMLCOMMENT		保留	保留	
XMLCONCAT	非保留（不能是函数或类型）	保留	保留	
XMLDECLARATION		非保留	非保留	
XMLDOCUMENT		保留	保留	
XMLELEMENT	非保留（不能是函数或类型）	保留	保留	
XMLEXISTS	非保留（不能是函数或类型）	保留	保留	
XMLFOREST	非保留（不能是函数或类型）	保留	保留	
XMLITERATE		保留	保留	
XMLNAMESPACES	非保留（不能是函数或类型）	保留	保留	
XMLPARSE	非保留（不能是函数或类型）	保留	保留	
XMLPI	非保留（不能是函数或类型）	保留	保留	
XMLQUERY		保留	保留	
XMLROOT	非保留（不能是函数或类型）			
XMLSCHEMA		非保留	非保留	
XMLSERIALIZE	非保留（不能是函数或类型）	保留	保留	
XMLTABLE	非保留（不能是函数或类型）	保留	保留	
XMLTEXT		保留	保留	

关键词	PostgreSQL	SQL:2011	SQL:2008	SQL-92
XMLVALIDATE		保留	保留	
YEAR	非保留	保留	保留	保留
YES	非保留	非保留	非保留	
ZONE	非保留	非保留	非保留	保留

附录 D. SQL 符合性

这一节尝试勾勒出PostgreSQL与当前 SQL 标准相符合的范围。下面的信息并不是符合性的完整说明，但是它尽可能详细地表达了对用户合理且有用的主要主题。

SQL 标准的正式名称是 ISO/IEC 9075 “数据库语言 SQL”。该标准会不时地发布一个修改的版本，最近一次更新出现在 2011 年。2011 年的版本被称为 ISO/IEC 9075:2011，或者简单地为 SQL:2011。之前的版本是 SQL:2008、SQL:2003、SQL:1999 和 SQL-92。每一个版本会替换之前的一个版本，因此声称与早前的版本相符合没有意义。PostgreSQL的开发希望与该标准的最新官方版本相符，并且这种符合不会与额外特性或尝试相冲突。很多 SQL 标准所要求的特性都被支持，不过有时在语法或函数上有所不同。随着时间的推移，符合性会得到进一步的提高。

SQL-92为符合性定义了三个特性集：入口、中间和完整。大部分号称SQL标准符合的数据库管理系统只是在入口级别上的符合，因为中间和完整级别的整个特性集合要么太多要么与遗留行为冲突。

从SQL:1999开始，SQL 标准定义了一个大型的个体特性集合，而没有无用地拓宽SQL-92中的三个级别。这些特性中的一个大型子集代表“核心”特性，每一个符合 SQL 的实现都必须提供。剩下的特性纯粹是可选的。一些可选的特性被分组到一起形成“包”，SQL 实现可以声称符合它们，因此符合特定的特性组。

从SQL:2003开始的标准版本也被划分成数个部分。每一个部分有一个速记名。注意这些部分不是连续编号。

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

PostgreSQL核心覆盖了部分 1、2、9、11 和 14。部分 3 被 ODBC 驱动所覆盖，并且部分 13 被 PL/Java 插件所覆盖，但是目前准确的符合性还没有在这些组件上被验证。目前对于PostgreSQL没有部分 4 和 10 的实现。

PostgreSQL 支持 SQL:2011 的大部分主要特性。在 179 个完整核心符合所要求的强制特性中，PostgreSQL 至少符合 160 个。另外，还有一个受支持的可选特性的长长的列表。值得注意的是，在编写此文档时，还没有任何数据库管理系统的当前版本声称完全符合核心的 SQL:2011。

在下面的两节中，我们提供了一个PostgreSQL所支持特性的列表，以及一个在SQL:2011中定义却还未被PostgreSQL支持的特性的列表。这两个列表都是大概的：对于被列为支持的一个

特性可能会有少量的细节不符合，而且大部分未被支持的特性可能事实上已经被实现。本文档的主体部分包含了哪些能用哪些不能用的准确信息。

> **注意**
>
> 包含一个连字符的特性编码是子特性。因此，如果一个特定的子特性没有被支持，其主特性被列为未支持，即使其他的子特性都已被支持。

D.1. 已支持特性

标识符	包	描述	注释
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	

标识符	包	描述	注释
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	

标识符	包	描述	注释
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081	Core	Basic Privileges	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-05	Core	UPDATE privilege at the column level	
E081-06	Core	REFERENCES privilege at the table level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	
E081-09	Core	USAGE privilege	
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	

标识符	包	描述	注释
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121	Core	Basic cursor support	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	

标识符	包	描述	注释
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E153	Core	Updatable queries with subqueries	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	

标识符	包	描述	注释
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	

标识符	包	描述	注释
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in	

标识符	包	描述	注释
		queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F200		TRUNCATE TABLE statement	
F201	Core	CAST function	
F202		TRUNCATE TABLE: identity column restart option	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F262		Extended CASE expression	
F271		Compound character literals	
F281		LIKE enhancements	

标识符	包	描述	注释
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F382		Alter column data type	
F383		Set column not null clause	
F384		Drop identity property clause	
F386		Set identity column generation clause	
F391		Long identifiers	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	

标识符	包	描述	注释
F402		Named column joins for LOBs, arrays, and multisets	
F411	Enhanced datetime facilities	Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F442		Mixed column references in set functions	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	

标识符	包	描述	注释
F591		Derived tables	
F611		Indicator data types	
F641		Row and table constructors	
F651		Catalog name qualifiers	
F661		Simple tables	
F672		Retrospective check constraints	
F690		Collation support	but no character set support
F692		Extended collation support	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F731		INSERT column privileges	
F751		View CHECK enhancements	
F761		Session management	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Connection management	
F781		Self-referencing operations	
F791		Insensitive cursors	
F801		Full set function	
F850		Top-level <order by clause> in <query expression>	
F851		<order by clause> in subqueries	
F852		Top-level <order by clause> in views	
F855		Nested <order by clause> in <query expression>	
F856		Nested <fetch first clause> in <query expression>	
F857		Top-level <fetch first clause> in <query expression>	

标识符	包	描述	注释
F858		<fetch first clause> in subqueries	
F859		Top-level <fetch first clause> in views	
F860		<fetch first row count> in <fetch first clause>	
F861		Top-level <result offset clause> in <query expression>	
F862		<result offset clause> in subqueries	
F863		Nested <result offset clause> in <query expression>	
F864		Top-level <result offset clause> in views	
F865		<offset row count> in <result offset clause>	
S071	Enhanced object support	SQL paths in function and type name resolution	
S092		Arrays of user-defined types	
S095		Array constructors by query	
S096		Optional array bounds	
S098		ARRAY_AGG	
S111	Enhanced object support	ONLY in query expressions	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S211	Enhanced object support	User-defined cast functions	
S301		Enhanced UNNEST	
T031		BOOLEAN data type	
T071		BIGINT data type	

标识符	包	描述	注释
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T152		DISTINCT predicate with negation	
T171		LIKE clause in table definition	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T174		Identity columns	
T177		Sequence generator support: simple restart option	
T178		Identity columns: simple restart option	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Active database, Enhanced integrity management	BEFORE triggers	
T211-03	Active database, Enhanced integrity management	AFTER triggers	
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers	
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true	

标识符	包	描述	注释
		before the trigger is invoked	
T211-07	Active database, Enhanced integrity management	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T213		INSTEAD OF triggers	
T231		Sensitive cursors	
T241		START TRANSACTION statement	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T285		Enhanced derived column names	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-02	Core	User-defined stored procedures with no overloading	
T321-03	Core	Function invocation	
T321-04	Core	CALL statement	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T323		Explicit security for external routines	
T325		Qualified SQL parameter references	
T331		Basic roles	
T341		Overloading of SQL-invoked functions and procedures	
T351		Bracketed SQL comments (/*...*/ comments)	
T431	OLAP	Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	

标识符	包	描述	注释
T433		Multiargument GROUPING function	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T491		LATERAL derived table	
T501		Enhanced EXISTS predicate	
T521		Named arguments in CALL statement	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	
T611	OLAP	Elementary OLAP operations	
T613		Sampling	
T614		NTILE function	
T615		LEAD and LAG functions	
T617		FIRST_VALUE and LAST_VALUE function	
T620		WINDOW clause: GROUPS option	
T621		Enhanced numeric functions	
T631	Core	IN predicate with one list element	
T651		SQL-schema statements in SQL routines	
T655		Cyclically dependent routines	
X010		XML type	
X011		Arrays of XML type	
X014		Attributes of XML type	
X016		Persistent XML values	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	

标识符	包	描述	注释
X034		XMLAgg	
X035		XMLAgg: ORDER BY option	
X036		XMLComment	
X037		XMLPI	
X040		Basic table mapping	
X041		Basic table mapping: nulls absent	
X042		Basic table mapping: null as nil	
X043		Basic table mapping: table as forest	
X044		Basic table mapping: table as element	
X045		Basic table mapping: with target namespace	
X046		Basic table mapping: data mapping	
X047		Basic table mapping: metadata mapping	
X048		Basic table mapping: base64 encoding of binary strings	
X049		Basic table mapping: hex encoding of binary strings	
X050		Advanced table mapping	
X051		Advanced table mapping: nulls absent	
X052		Advanced table mapping: null as nil	
X053		Advanced table mapping: table as forest	
X054		Advanced table mapping: table as element	

标识符	包	描述	注释
X055		Advanced table mapping: with target namespace	
X056		Advanced table mapping: data mapping	
X057		Advanced table mapping: metadata mapping	
X058		Advanced table mapping: base64 encoding of binary strings	
X059		Advanced table mapping: hex encoding of binary strings	
X060		XMLParse: character string input and CONTENT option	
X061		XMLParse: character string input and DOCUMENT option	
X070		XMLSerialize: character string serialization and CONTENT option	
X071		XMLSerialize: character string serialization and DOCUMENT option	
X072		XMLSerialize: character string serialization	
X090		XML document predicate	
X120		XML parameters in SQL routines	
X121		XML parameters in external routines	
X222		XML passing mechanism BY REF	
X301		XMLTable: derived column list option	
X302		XMLTable: ordinality column option	
X303		XMLTable: column default option	

标识符	包	描述	注释
X304		XMLTable: passing a context item	
X400		Name and identifier mapping	
X410		Alter column data type: XML type	

D.2. 未支持特性

下列定义在SQL:2011中的特性还没有在这个PostgreSQL发行中被实现。在一些情况中，有等效的功能可用。

标识符	包	描述	注释
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B035		Non-extended descriptor names	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	

标识符	包	描述	注释
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
B211		Module language Ada: VARCHAR and NUMERIC support	
B221		Routine language Ada: VARCHAR and NUMERIC support	
E182	Core	Module language	
F054		TIMESTAMP in DATE type precedence list	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122		Enhanced diagnostics management	
F123		All diagnostics	
F181	Core	Multiple module support	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	

标识符	包	描述	注释
F311	Core	Schema definition statement	
F312		MERGE statement	consider INSERT ... ON CONFLICT DO UPDATE
F313		Enhanced MERGE statement	
F314		MERGE statement with DELETE branch	
F341		Usage tables	no ROUTINE_*_USAGE tables
F385		Drop column generation expression clause	
F394		Optional normal form specification	
F403		Partitioned joined tables	
F451		Character set definition	
F461		Named character sets	
F492		Optional table constraint enforcement	
F521	Enhanced integrity management	Assertions	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign and unique keys only
F741		Referential MATCH types	no partial match yet
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	

标识符	包	描述	注释
F831-02		Updatable ordered cursors	
F841		LIKE_REGEX predicate	
F842		OCCURRENCES_REGEX function	
F843		POSITION_REGEX function	
F844		SUBSTRING_REGEX function	
F845		TRANSLATE_REGEX function	
F846		Octet support in regular expression operators	
F847		Nonconstant regular expressions	
F866		FETCH FIRST clause: PERCENT option	
F867		FETCH FIRST clause: WITH TIES option	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	partially supported
S081	Enhanced object support	Subtables	
S091		Basic array support	partially supported

标识符	包	描述	注释
S091-01		Arrays of built-in data types	
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S094		Arrays of reference types	
S097		Array element assignment	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S162		Subtype treatment for references	
S202		SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
S401		Distinct types based on array types	
S402		Distinct types based on distinct types	
S403		ARRAY_MAX_CARDINALITY	

标识符	包	描述	注释
S404		TRIM_ARRAY	
T011		Timestamp in Information Schema	
T021		BINARY and VARBINARY data types	
T022		Advanced support for BINARY and VARBINARY data types	
T023		Compound binary literal	
T024		Spaces in binary literals	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T043		Multiplier T	
T044		Multiplier P	
T051		Row types	
T052		MAX and MIN for row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T101		Enhanced nullability determination	
T111		Updatable joins, unions, and columns	
T175		Generated columns	
T176		Sequence generator support	

标识符	包	描述	注释
T180		System-versioned tables	
T181		Application-time period tables	
T211	Active database, Enhanced integrity management	Basic trigger capability	
T211-06	Active database, Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T272		Enhanced savepoint management	
T301		Functional dependencies	partially supported
T321	Core	Basic SQL-invoked routines	
T321-05	Core	RETURN statement	
T322	PSM	Declared data type attributes	
T324		Explicit security for SQL routines	
T326		Table functions	
T332		Extended roles	mostly supported
T434		GROUP BY DISTINCT	
T471		Result sets return value	
T472		DESCRIBE CURSOR	
T495		Combined data change and retrieval	different syntax
T502		Period predicates	
T511		Transaction counts	
T522		Default values for IN parameters	supported except DEFAULT key word in invocation

标识符	包	描述	注释
		of SQL-invoked procedures	
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T612		Advanced OLAP operations	some forms supported
T616		Null treatment option for LEAD and LAG functions	
T618		NTH_VALUE function	function exists, but some options missing
T619		Nested window functions	
T641		Multiple column assignment	only some syntax variants supported
T652		SQL-dynamic statements in SQL routines	
T653		SQL-schema statements in external routines	
T654		SQL-dynamic statements in external routines	
M001		Datalinks	
M002		Datalinks via SQL/CLI	
M003		Datalinks via Embedded SQL	
M004		Foreign data support	partially supported
M005		Foreign schema support	
M006		GetSQLString routine	
M007		TransmitRequest	
M009		GetOpts and GetStatistics routines	

标识符	包	描述	注释
M010		Foreign data wrapper support	different API
M011		Datalinks via Ada	
M012		Datalinks via C	
M013		Datalinks via COBOL	
M014		Datalinks via Fortran	
M015		Datalinks via M	
M016		Datalinks via Pascal	
M017		Datalinks via PL/I	
M018		Foreign data wrapper interface routines in Ada	
M019		Foreign data wrapper interface routines in C	different API
M020		Foreign data wrapper interface routines in COBOL	
M021		Foreign data wrapper interface routines in Fortran	
M022		Foreign data wrapper interface routines in MUMPS	
M023		Foreign data wrapper interface routines in Pascal	
M024		Foreign data wrapper interface routines in PL/I	
M030		SQL-server foreign data support	
M031		Foreign data wrapper general routines	
X012		Multisets of XML type	
X013		Distinct types of XML type	
X015		Fields of XML type	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	

标识符	包	描述	注释
X065		XMLParse: BLOB input and CONTENT option	
X066		XMLParse: BLOB input and DOCUMENT option	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOB serialization and CONTENT option	
X074		XMLSerialize: BLOB serialization and DOCUMENT option	
X075		XMLSerialize: BLOB serialization	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: explicit ENCODING option	
X078		XMLSerialize: explicit XML declaration	
X080		Namespaces in XML publishing	
X081		Query-level XML namespace declarations	
X082		XML namespace declarations in DML	
X083		XML namespace declarations in DDL	
X084		XML namespace declarations in compound statements	
X085		Predefined namespace prefixes	
X086		XML namespace declarations in XMLTable	
X091		XML content predicate	
X096		XMLExists	XPath only
X100		Host language support for XML: CONTENT option	

标识符	包	描述	注释
X101		Host language support for XML: DOCUMENT option	
X110		Host language support for XML: VARCHAR mapping	
X111		Host language support for XML: CLOB mapping	
X112		Host language support for XML: BLOB mapping	
X113		Host language support for XML: STRIP WHITESPACE option	
X114		Host language support for XML: PRESERVE WHITESPACE option	
X131		Query-level XMLBINARY clause	
X132		XMLBINARY clause in DML	
X133		XMLBINARY clause in DDL	
X134		XMLBINARY clause in compound statements	
X135		XMLBINARY clause in subqueries	
X141		IS VALID predicate: data-driven case	
X142		IS VALID predicate: ACCORDING TO clause	
X143		IS VALID predicate: ELEMENT clause	
X144		IS VALID predicate: schema location	
X145		IS VALID predicate outside check constraints	
X151		IS VALID predicate with DOCUMENT option	
X152		IS VALID predicate with CONTENT option	
X153		IS VALID predicate with SEQUENCE option	

标识符	包	描述	注释
X155		IS VALID predicate: NAMESPACE without ELEMENT clause	
X157		IS VALID predicate: NO NAMESPACE with ELEMENT clause	
X160		Basic Information Schema for registered XML Schemas	
X161		Advanced Information Schema for registered XML Schemas	
X170		XML null handling options	
X171		NIL ON NO CONTENT option	
X181		XML(DOCUMENT(UNTYPED)) type	
X182		XML(DOCUMENT(ANY)) type	
X190		XML(SEQUENCE) type	
X191		XML(DOCUMENT(XMLSCHEMA)) type	
X192		XML(CONTENT(XMLSCHEMA)) type	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: passing a context item	
X204		XMLQuery: initializing an XQuery variable	
X205		XMLQuery: EMPTY ON EMPTY option	
X206		XMLQuery: NULL ON EMPTY option	
X211		XML 1.1 support	
X221		XML passing mechanism BY VALUE	
X231		XML(CONTENT(UNTYPED)) type	
X232		XML(CONTENT(ANY)) type	

标识符	包	描述	注释
X241		RETURNING CONTENT in XML publishing	
X242		RETURNING SEQUENCE in XML publishing	
X251		Persistent XML values of XML(DOCUMENT(UNTYPED)) type	
X252		Persistent XML values of XML(DOCUMENT(ANY)) type	
X253		Persistent XML values of XML(CONTENT(UNTYPED)) type	
X254		Persistent XML values of XML(CONTENT(ANY)) type	
X255		Persistent XML values of XML(SEQUENCE) type	
X256		Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	
X257		Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	
X260		XML type: ELEMENT clause	
X261		XML type: NAMESPACE without ELEMENT clause	
X263		XML type: NO NAMESPACE with ELEMENT clause	
X264		XML type: schema location	
X271		XMLValidate: data-driven case	
X272		XMLValidate: ACCORDING TO clause	
X273		XMLValidate: ELEMENT clause	
X274		XMLValidate: schema location	

标识符	包	描述	注释
X281		XMLValidate with DOCUMENT option	
X282		XMLValidate with CONTENT option	
X283		XMLValidate with SEQUENCE option	
X284		XMLValidate: NAMESPACE without ELEMENT clause	
X286		XMLValidate: NO NAMESPACE with ELEMENT clause	
X300		XMLTable	XPath only
X305		XMLTable: initializing an XQuery variable	

附录 E. 版本说明

版本说明包含每个PostgreSQL版本中重大变化， 在上面列出的主要特性和迁移问题。 版本说明不包含只影响少数用户或者改变是内部的不是用户可见的那些变化。 比如，在几乎每一个版本中都会改进优化器。不过这种改进往往要通过查询速度的加快才能被用户所觉察。

每个版本变化的完整列表可以通过查看每个版本的Git日志 获得。pgsql-committers email list[1]也记录所有源代码的变化。 还有一个WEB 接口[2]显示了特定文件的变化。

每个项目旁边显示的人名代表了该项目的主要开发人员。当然 所有的更改包含了社区讨论和补丁检查，因此每个项目都真的是社区努力的成果。

E.1. 版本11.2

发布日期: 2019-02-14

该版本包含一些自版本11.1以来的修补。关于主版本11的新特性的信息，见第 E.3 节

E.1.1. 迁移到版本11.2

对于运行11.X的数据库，不需要转储/恢复。

E.1.2. 修改列表

- 缺省地，在fsync()失败后用panic代替重试，以避免可能的数据损坏。(Craig Ringer, Thomas Munro)

  当不能将内核数据缓冲区写出来时，一些流行的操作系统会丢弃缓冲区的内容，将这报告为fsync()失败。如重新发出fsync()请求，会成功，但实际上数据已丢失，这样继续下去就会冒数据损坏的风险。相反通过出一次panic状况，我们可以从WAL重放，这其中可能包含该情形下仅余下的数据拷贝。这当然令人讨厌且效率低下，但几乎没有其他选择。所幸的是这种情况很少发生。

  新增加的服务器参数data_sync_retry用来控制此项设置；如果确定此情形下内核不丢弃脏数据缓冲区，可以设置data_sync_retry为on，以恢复过去的行为。

- 在文档中只包含每个主要发行分支的发行注释，而不包括该分支和所有后续分支的发行注释 (Tom Lane)

  以前的策略导致的重复近乎失控。我们的计划是在项目的网站上提供发行注释的全部归档，但不在每次发布中重复。

- 修复分区表上具有INCLUDE列的唯一索引的处理 (Álvaro Herrera)

  该情况没有适当地检查唯一性条件。

- 确保分区表的NOT NULL约束在其分区内得到遵守 (Álvaro Herrera, Amit Langote)

- 分离分区表时正确更新分区的表约束的目录状态 (Amit Langote, Álvaro Herrera)

  以前，此类约束的pg_constraint.conislocal字段可能不正确地保留为false，从而使其不可删除。转储/恢复或pg_upgrade可以解决此问题，但如有需要，可以手动调整目录字段。

[1] https://www.postgresql.org/list/pgsql-committers/
[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary

- 在具有外键约束的分区表中附加或分离分区时，正确地创建或删除强制触发器 (Amit Langote, Álvaro Herrera)
- 避免在分区表中无用地创建重复的外键约束 (Álvaro Herrera)
- 当在分区表上使用ONLY创建的索引还没有分区时，立即将其标记为有效 (Álvaro Herrera)

  否则，无法让其生效。
- 分离分区时，使用安全的表锁级别 (Álvaro Herrera)

  以前的表锁级别太弱，可能允许表上并发执行DDL，产生坏结果。
- 修复对分区表和具有继承孩子的表应用ON COMMIT DROP和ON COMMIT DELETE ROWS的问题 (Michael Paquier)
- 分区表上不允许COPY FREEZE (David Rowley)

  这最终应该能起作用，但需要一个非常复杂的补丁，可能有回补的风险。
- 修复索引列具有“快速默认值”(也就是说，表中已有一些行之后，又通过ALTER TABLE ADD COLUMN，以指定缺省值来增加行)时可能出现的索引损坏 Andres Freund)
- 在ALTER TABLE ... ALTER COLUMN TYPE时，正确调整“快速默认值” (Andrew Dunstan)
- 获取多个缓冲区锁时避免可能的死锁 (Nishant Fnu)
- 避免GIN清理和并发索引插入之间死锁 (Alexander Korotkov, Andrey Borodin, Peter Geoghegan)

  此更改部分恢复了版本10.0中引入的性能改进，该改进尝试减少删除GIN posting树页面期间锁定的索引页数。现在已经发现这会导致死锁，所以我们在进一步分析之前将之移除。
- 避免热备份查询与GIN索引页删除重放间的死锁 (Alexander Korotkov)
- 修复索引表达式或谓词在使用时逻辑复制可能的崩溃 (Peter Eisentraut)
- 避免表重写期间对TOAST数据进行无用的、高代价的逻辑解码 (Tomas Vondra)
- 修复同步复制启用时停止WAL发送器子集的逻辑 (Paul Guo, Michael Paquier)
- 避免在删除元组的WAL记录中写入不正确的副本标识字段 (Stas Kelvich)
- 防止在COPY到视图或外部表期间错误地使用WAL跳过优化 (Amit Langote, Michael Paquier)
- 在选择哪些文件要归档时，使归档器优先归档WAL历史文件，而后才是WAL数据文件 (David Steele)
- 修复使用sub-SELECT作为源、有多个SET子句的UPDATE可能出现的崩溃 (Tom Lane)
- 修复只有零行输入到json[b]_populate_recordset()或json[b]_to_recordset()时的崩溃 (Tom Lane)
- 避免libxml2返回空错误信息时的崩溃 (Sergio Conde Gómez)
- 修复具有许多列（大约超过800）的表的不正确JIT元组变形代码 (Andres Freund)
- 修复基于哈希的分组中性能和内存泄漏问题 (Andres Freund)

- 修复由于排序规则分配的处理不一致而导致的与分组相关的错误 (Andrew Gierth)

  在某些情况下，如果表达式包括可排序数据类型上的操作，认为应当匹配的表达式不见得就匹配。

- 修复CALL语句参数中排序规则敏感的表达式的分析处理 (Peter Eisentraut)

- 检测到CALL语句的参数列表中有错误后，确保正确清除 (Tom Lane)

- 检查LEAST()或GREATEST()下面的比较函数是否防泄漏，而不是假设它就是 (Tom Lane)

  由btree比较函数引起的实际信息泄漏通常很难发生，但原则上它们可能发生。

- 修复在Gather计划节点上方和下方涉及嵌套循环的查询的不正确规划 (Tom Lane)

  如果嵌套循环的两层都要将同样的变量传到右侧，就会产生错误的计划。

- 修复外部表扫描时必须评估横向引用的查询的错误计划 (Tom Lane)

- 修复行比较的第一列匹配索引列，但后面并不匹配且索引包含（非键）列时的查询器故障 (Tom Lane)

- 修复极端情况下归并连接成本的低估 (Tom Lane)

  当外部键范围比内部键范围小得多时，计划器可能更偏向归并连接，即使内部有太多重复键，这是一种糟糕的选择。

- 当查询包含数千条索引子句时，避免O(N^2)的计划时间增长 (Tom Lane)

- 提高大型继承或分区表组的规划速度 (Amit Langote, Etsuro Fujita)

- 改进并发更新行的ANALYZE处理 (Jeff Janes, Tom Lane)

  以前，由正在进行的事务删除的行在ANALYZE样本中是被忽略的，但这会导致比包含它们有更大的不一致性。实际上，现在采样对应于ANALYZE开始时的MVCC快照。

- 使TRUNCATE忽略作为其他会话的临时表的继承子表 (Amit Langote, Michael Paquier)

  这使TRUNCATE与其他命令的行为保持一致。以前这种情况通常是最终失败。

- 修复TRUNCATE以更新适当表的统计计数器 (Tom Lane)

  如果被截断的表有TOAST表，则该表计数器将被重置。

- 正确处理ALTER TABLE ONLY ADD COLUMN IF NOT EXISTS (Greg Stark)

- 允许在热备模式下UNLISTEN (Shay Rojansky)

  这必然是一个no-op，因为热备模式下不允许LISTEN；但允许虚操作简化客户端中的会话状态重置逻辑。

- 修复一些模式和数据类型权限列表中缺少的角色依赖 (Tom Lane)

  有些情况下可以删除已授予权限的角色。这不会立刻有问题，但是后来的转储/重载或升级失败，会出现尝试给全数字角色名授予权限的症状。

- 防止在两阶段事务中使用会话的临时模式 (Michael Paquier)

  访问此类事务中的临时表已被禁止了很长时间，但仍有可能导致在临时对象上其他操作的问题。

- 确保添加或删除外键约束后关系缓存适当地更新 (Álvaro Herrera)

  该疏忽会导致已有会话执行新建约束失败或继续执行已删除的约束。

- 确保重命名约束后关系缓存适当地更新 (Amit Langote)

- 修复GiST索引微清理操作的重放，这样并发热备查询不会看到不一致的状态 (Alexander Korotkov)

- 防止空GIN索引页回收得太快，导致并发搜索失败 (Andrey Borodin, Alexander Korotkov)

- 修复极端情况下浮点数到整数强制转换的失败 (Andrew Gierth, Tom Lane)

  比最大有效整数值大一点点的值可能不会被拒绝，然后会溢出，反而产生最小的有效整数。此外，应舍入最小或最大整数值的值可能被错误地拒绝。

- 修复pg_hba.conf中LDAP认证入口的ldapserver参数内由空格分隔的主机名列表的处理 (Thomas Munro)

- 进行PAM认证请求时，如果是通过Unix套接字连接，则不要设置PAM_RHOST变量 (Thomas Munro)

  以前，该变量设置为[local]，这其实没什么用，因为它应为主机名。

- 不允许设置client_min_messages的值高于ERROR (Jonah Harris, Tom Lane)

  以前，可以将此变量设置为FATAL或PANIC，这会抑制向客户端传输一般错误消息。但是，这与PostgreSQL有线协议规范中给出的保证相违背，导致一些客户端变得非常混乱。在已发布的分支中，通过将这些设置默认为有意义的ERROR来解决此问题。版本12和以后的版本完全不会考虑这些替代方案。

- 修复ecpglib，优先使用uselocale()或_configthreadlocale()，而后才是setlocale() (Michael Meskes, Tom Lane)

  由于setlocale()不是线程局部的，甚至可能不是线程安全的，所以导致以前的编码在多线程ecpg应用程序中出了问题。

- 修复通过ecpg SQLDA (SQL描述符区域)传数值数据的错误结果 (Daisuke Higuchi)

  带前导零的值未正确复制。

- 修复psql的\g target元命令以使用COPY TO STDOUT (Daniel Vérité)

  以前忽略了target选项，因此拷贝的数据总是跑到当前查询输出目标那里。

- 使psql的LaTeX输出格式正确地呈现特殊字符 (Tom Lane)

  反斜杠和其他一些ASCII标点符号未能正确呈现，导致文档语法错误或输出中出现错误字符。

- 当指定--random-seed=N时，使pgbench随机数生成完全确定且平台无关 (Fabien Coelho, Tom Lane)

  在任何特定平台上，用指定的N值获得的序列可能与打上补丁之前的序列不同。

- 修复pg_basebackup和pg_verify_checksums以适当地忽略临时文件 (Michael Banck, Michael Paquier)

- 修复pg_dump对间接依赖主键的物化视图的处理 (Tom Lane)

这导致对此类视图的转储归档条目的错误标记，从而导致有关“归档项的片段顺序不对”的无害警告；害处不大的是，依赖于这些标签的选择性恢复选项（例如--section）可能会出现错误行为。

- 使pg_dump包含ALTER INDEX SET STATISTICS命令 (Michael Paquier)

  当添加了将统计目标附加到索引表达式的功能时，我们忘记教pg_dump处理相关内容，因此在转储/重载中丢掉了这些设置。

- 修复pg_dump对具有OID的表的转储 (Peter Eisentraut)

  如果需要将WITH OIDS子句应用于要转储的第一个表，则省略该子句。

- 避免在某些平台上，当pg_dump或pg_restore尝试报告错误时发生空指针解引用崩溃 (Tom Lane)

- 防止由于内联压缩数据导致的contrib/amcheck错误索引损坏报告 (Peter Geoghegan)

- 如果COPY FROM PROGRAM早先停止读取程序输出，则适当地忽略SIGPIPE错误 (Tom Lane)

  这种情况实际上不能直接通过COPY达到，但是在使用contrib/file_fdw时可能会发生。

- 修复contrib/hstore以计算在版本8.4或之前版本创建的空hstore值的正确的哈希值 (Andrew Gierth)

  之前的编码并未给出与新版本创建的空hstore值相同的结果，因此可能导致哈希连接或哈希聚集中出现错误结果。如果表中可能包含早在8.4之前就存储的数据，并且从那时起从未转储/重新加载过这些数据，则建议对基于hstore列构建的任何哈希索引重新做索引。

- 避免向contrib/intarray的gist__int_ops索引支持提供大量输入时崩溃和过多的运行时间 (Andrew Gierth)

- 在configure时，如果没有找到python，则接着找python3，然后是python2 (Peter Eisentraut)

  这允许在已不提供无版本号的python可执行文件的平台上配置PL/Python，而无需显式指定PYTHON。

- 在已安装的头文件集中包含与JIT相关的头文件 (Donald Dong)

- 在pgxs构建中，支持新的Makefile变量PG_CFLAGS, PG_CXXFLAGS和PG_LDFLAGS (Christoph Berg)

  这简化了扩展构建过程的定制。

- 修复用Perl编写的构建脚本，不在假定“.”在搜索路径中，因为最近的Perl版本并不包含该功能 (Andrew Dunstan)

- 修复OpenBSD上服务器命令行选项分析的问题 (Tom Lane)

- 重新定位set_rel_pathlist_hook的调用，以便扩展程序可以用它来为并行查询提供部分路径 (KaiGai Kohei)

  这不会影响现有的用例。

- 将时区文件更新为tzdata 2018i版本，以应对哈萨克斯坦、梅特拉卡特拉、圣多美和普林西比的DST法律变更。哈萨克斯坦的Qyzylorda区一分为二，从而创建了一个新的Asia/Qostanay区，因为某些地区没有改变UTC偏移量。对香港（中国）和众多太平洋岛屿的历史性更正。

E.2. 版本11.1

发布日期: 2018-11-08

本次发布包含了对11.0的一些修正。主版本11的新特性的有关信息，见第 E.3 节

E.2.1. 迁移到版本11.1

对于运行11.X的，不需要转储/恢复。

但如果使用pg_stat_statements扩展，参见下面修改日志入口的有关内容。

E.2.2. 修改列表

- 当pg_dump发出CREATE TRIGGER ... REFERENCING命令时，要确保对转换表名加引号 (Tom Lane)

  此疏忽可被非特权用户利用，在下次转储/重载或pg_upgrade运行时获取到超级用户特权。(CVE-2018-16850)

- 创建孩子索引时，应用为分区索引指定的表空间 (Álvaro Herrera)

  以前，一直是在缺省表空间中创建孩子索引。

- 修复在做并行哈希的多批左连接中NULL处理的问题 (Andrew Gierth, Thomas Munro)

  连接结果中忽略哈希键为空值的外关系行。

- 修复出现在有常量测试表达式的CASE子句中对数组类型强制型转表达式的不正确处理 (Tom Lane)

- 修复缺少最近添加列的元组的不正确的扩张 (Andrew Dunstan, Amit Langote)

  众所周知，这会导致在有最近添加列的表上的触发器中崩溃，也还会有其他问题。

- 修复CALL参数列表中命名和缺省参数的BUG (Tom Lane, Pavel Stehule)

- 修复对有ORDER BY列的严格聚集的严格性检查 (Andrew Gierth, Andres Freund)

  严格性逻辑错误地忽略了ORDER BY值为空值的行。

- 禁用recheck_on_update优化 (Tom Lane)

  版本11中的这项新特性有点“生不逢时”。先禁用它，直到可以做点什么关于它的。

- 防止在附加到其父表的触发器中创建分区 (Amit Langote)

  理想情况下，允许这么做，但现在为了避免崩溃阻止这么做。

- 修复将ON COMMIT DELETE ROWS应用于已分区的临时表的问题 (Amit Langote)

- 修复字符类的检查，这样在Windows上，对于U+FFFF以上的Unicode字符不会失败 (Tom Lane, Kenji Uno)

  该BUG既影响全文搜索的操作，也影响contrib/ltree和contrib/pg_trgm。

- 确保服务器在等待客户端输入之前处理已接收的NOTIFY和SIGTERM中断 (Jeff Janes, Tom Lane)

- 修复反复进行SP-GiST索引扫描时的内存泄露 (Tom Lane)

这只有在单一命令中使用了SP-GiST的排除约束收到很多的索引项时才比较明显。

- 防止服务器启动时的wal_level值设得太低以至于无法支持一个已有的复制槽 (Andres Freund)
- 修复psql和文档的例子，在每次调用PQnotifies()之前调用PQconsumeInput() (Tom Lane)

  这修复了直到下一条命令之后psql才能报告NOTIFY消息的接收情况。
- 修复pg_verify_checksums，确定哪些文件要做校验和 (Michael Paquier)

  用户抱怨有些情况下一些文件无需校验和。
- 在contrib/pg_stat_statements，不允许pg_read_all_stats角色执行pg_stat_statements_reset() (Haribabu Kommi)

  pg_read_all_stats仅是为读统计授权，而不是改变它们，故该授权方式不正确。

  要使此修改生效，需要在每个安装了pg_stat_statements的数据库上运行ALTER EXTENSION pg_stat_statements UPDATE（新建的11.0版数据库不需要做这些，但从以前版本升级的数据库很可能还包含老版本的pg_stat_statements。即使该模块已经升级，UPDATE命令也是没有坏处的。）
- 重命名红黑树支持函数，使用rbt前缀，而不是rb前缀 (Tom Lane)

  这将避免与Ruby函数的名字冲突。此类冲突会破坏PL/Ruby。希望其他扩展不受影响。
- 修复macOS 10.14(Mojave)上构建的问题 (Tom Lane)

  调整configure，将-isysroot开关增加到CPPFLAGS；无此开关的话，在macOS 10.14上，PL/Perl和PL/Tcl配置或构建就会失败。通过在configure或make中设置PG_SYSROOT变量，所用的特定sysroot在配置或构建时会被覆盖。

  对于Perl有关的扩展，现在推荐在其编译标记处写$(perl_includespec)，而不是-I $(perl_archlibexp)/CORE。后者仍能工作，但最近的macOS不行。

  现在要在最近的macOS版本上构建PL/Tcl也不需要人工指定--with-tclconfig。
- 修复MSVC的构建和回归测试脚本，以使其能在最近的Perl版本上工作 (Andrew Dunstan)

  缺省情况下，Perl的搜索路径中不再包含当前目录；变通以解决之。
- Windows上，允许由Administrator账户运行回归测试 (Andrew Dunstan)

  为保险起见，pg_regress现在启动时会放弃任何此类特权。
- 将时区数据文件更新为tzdata 2018g版，以适应智利、斐济、摩洛哥和俄罗斯（伏尔加格勒）的DST法律更改，以及中国、夏威夷、日本、澳门（中国）和朝鲜的历史更正。

E.3. 版本11

发布日期: 2018-10-18

E.3.1. 概述

PostgreSQL11的主要改进有：

- 分区功能的改进，包括：

- 增加用哈希键分区的支持
- 增加分区表上PRIMARY KEY, FOREIGN KEY,索引和触发器的支持
- 允许创建“default”分区，以存储与其余分区都不匹配的数据
- 改变分区键列的UPDATE语句现在会导致受到影响的行移动到适当的分区
- 在查询计划和执行时通过增强分区排除策略来提高SELECT性能

- 并行性改进，包括：
  - 现在构建B-树索引时CREATE INDEX可使用并行处理
  - 现在CREATE TABLE ... AS,CREATE MATERIALIZED VIEW和某些使用UNION的查询中并行是可能的
  - 并行化的哈希连接和并行化顺序扫描现在表现更好
- 支持嵌入式事务的SQL存储过程
- 对于一些SQL代码，可用即时(JIT)编译加速表达式评估
- 窗口函数现在支持SQL:2011标准中的所有帧选项，包括RANGE distance PRECEDING/FOLLOWING, GROUPS模式和帧排除选项
- 现在使用 CREATE INDEX的INCLUDE子句可创建覆盖索引
- 其它很多有用的性能改进，包括在ALTER TABLE ... ADD COLUMN以非空列作为缺省时避免对表重写的能力

以上的项在下面的章节中详细说明。

E.3.2. 迁移到版本11

要从任何以前的版本迁移数据，需要使用pg_dumpall或使用pg_upgrade来转储/恢复。

版本11包含一些影响到与以前的版本的兼容性的修改。注意以下不兼容的方面：

- 使pg_dump转储数据库属性，而不仅仅是其内容 (Haribabu Kommi)

  以前，数据库本身的属性，例如数据库级的GRANT/REVOKE权限和ALTER DATABASE SET变量设置，仅由pg_dumpall来转储。现在pg_dump --create 和pg_restore --create将恢复数据库中的属性和对象。pg_dumpall -g现在仅转储角色和表空间相关属性。pg_dumpall的全部输出(无-g)未改变。

  pg_dump和pg_restore，没有--create选项时，不在转储/恢复数据库级注释和安全标记，这些内容现在被当作数据库属性。

  pg_dumpall的输出脚本现在创建数据库时总与其原来的区域设置和编码一起，所以如果目的系统的区域设置和编码名称未知的话，就会失败。以前，如果数据库的区域设置和编码与旧集群的缺省设置匹配的话，则CREATE DATABASE命令无此要求即被发出。

  pg_dumpall --clean现在可恢复postgres和template1数据库原来的区域设置和编码设置，这与用户创建的数据库是一样的。

- 消除函数与列引用的歧义时考虑句法形式 (Tom Lane)

  当x是表名或复合列时，PostgreSQL通常认为f(x)和x.f的句法形式是等价的。这就允许类似于写一个函数，然后使用时它象是一按需计算的列这样的小把戏。但是，如果两种解释

都可行，一般总选择列解释，就会导致用户需要函数解释时出现奇怪的结果。现在，如果有歧义，将选择匹配句法形式的解释。

- 完全强制表和域约束名的唯一性 (Tom Lane)

  PostgreSQL要求表名不同，约束名也不同。但是，对此没有严格的强制措施，以前出现过相同名可以创建的特殊案例。

- 使power(numeric, numeric)和power(float8, float8)按POSIX标准处理NaN输入 (Tom Lane, Dang Minh Huong)

  POSIX要求NaN ^ 0 = 1，1 ^ NaN = 1，所有其他有NaN输入的情况都应返回NaN。在这些所有的情况下，power(numeric, numeric)正好返回NaN；现在接受两种异常。power(float8, float8)遵循标准只要C库支持；但在一些老的Unix平台上C库并未遵循标准，在Windows的一些版本上也有问题。

- 防止to_number()在模板分隔符不匹配时消耗掉字符 (Oliver Ford)

  特别地，SELECT to_number('1234', '9,999')过去返回134，现在返回1234。L和TH现在仅消掉非数字、正/负符号、小数点和逗号的字符。

- 修复to_date(),to_number()和to_timestamp()，对于每个模板字符跳过一个字符 (Tom Lane)

  以前，对于模板字符的每一字节，跳过一个byte。如果字符串包含多字节字符，则导致奇怪的行为。

- 调整to_char(),to_number()和to_timestamp()模板字符串中双引号内的反斜杠的处理。

  现在反斜杠将对其后面的字符转义，特别是双引号或其它反斜杠。

- 正确处理xmltable(), xpath()和其他XML处理函数中的相对路径表达式 (Markus Winand)

  根据SQL标准，相对路径是从XML输入文档的文档节点开始，而不是这些函数以前所用的根节点。

- 在扩展查询协议中，使statement_timeout独立地应用于每个Execute消息，而不是Sync前的所有命令 (Tatsuo Ishii, Andres Freund)

- 从系统目录pg_class删除relhaspkey列(Peter Eisentraut)

  需要检查主键的应用程序应查询pg_index。

- 系统目录pg_proc的proisagg和proiswindow列被替换为prokind (Peter Eisentraut)

  新的列可更清晰地区分函数、过程、聚集和窗口函数。

- 纠正信息模式列tables.table_type，返回FOREIGN而不是FOREIGN TABLE (Peter Eisentraut)

  新输出符合SQL标准。

- 修改后台工作者的ps进程显示标记以匹配pg_stat_activity.backend_type标记 (Peter Eisentraut)

- 使大对象的权限检查在大对象打开即lo_open()时进行，而不是在尝试读和写时进行 (Tom Lane, Michael Paquier)

  如果写访问已请求且未可用，现在会抛出一条错误，即使该大对象从未被写过。

- 防止非超级用户重建共享的目录索引 (Michael Paquier, Robert Haas)

以前，允许数据库的拥有者这么做，但现在被看作是在其权限边界之外。

- 移除过时的adminpack函数pg_file_read(),pg_file_length()和pg_logfile_rotate() (Stephen Frost)

  相同的功能现在在核心后端中。现有的adminpack安装可继续访问这些函数，直到用ALTER EXTENSION ... UPDATE更新为止。

- 遵守双引号命令选项的大写（规则）(Daniel Gustafsson)

  以前，特定SQL命令中的选型名称是强制小写，即使有双引号；比如"FillFactor"将接受为索引存储选项，尽管其名字应为小写。这种情况现在会产生错误。

- 移除服务器参数replacement_sort_tuples (Peter Geoghegan)

  经确认，不再使用替换排序。

- 移除CREATE FUNCTION中的WITH子句 (Michael Paquier)

  关于这种功能的一种更符合标准的语法，PostgreSQL已支持了很久。

- PL/pgSQL触发器函数中，没有赋值时，变量OLD和NEW现在读出来是NULL (Tom Lane)

  以前，对这些变量的引用可被解析但不执行。

E.3.3. 修改列表

下面是PostgreSQL11和以前主版本间的变更的详细说明。

E.3.3.1. 服务器

E.3.3.1.1. 分区

- 允许创建基于键列哈希的的分区 (Amul Sul)

- 支持分区表上的索引 (Álvaro Herrera, Amit Langote)

  分区表上的“索引”不是一种跨整个分区表的物理索引，而是一种自动创建表的每个分区上类似索引的模板。

  如果分区键是索引列集的一部分，分区索引可声明为UNIQUE。它将代表跨整个分区表的有效唯一性约束，即使每个物理索引仅强制其自己分区内的唯一性。

  新命令ALTER INDEX ATTACH PARTITION使得分区表分区上已有索引与相配的索引模板关联。这样为已存在的分区表设置新的分区索引提供了灵活性。

- 允许分区表上的外键 (Álvaro Herrera)

- 允许分区表上的FOR EACH ROW触发器 (Álvaro Herrera)

  创建分区表上的触发器，会自动创建所有已存在的和未来的分区上的触发器。这也允许分区表上可延迟的唯一约束。

- 允许分区表有缺省分区 (Jeevan Ladhe, Beena Emerson, Ashutosh Bapat, Rahila Syed, Robert Haas)

  缺省分区将存储与任何其他已定义分区不匹配的行，作相应的搜索。

- 改变分区键列的UPDATE语句现在会使得受影响的行移动到适当的分区 (Amit Khandekar)

- 允许分区表上的INSERT, UPDATE和COPY，将行正确地路由到外部分区 (Etsuro Fujita, Amit Langote)

  该功能由postgres_fdw外部表支持。

- 允许查询处理时更快的分区排除 (Amit Langote, David Rowley, Dilip Kumar)

  这将加速对有许多分区的分区表的访问。

- 允许查询执行期间分区消除 (David Rowley, Beena Emerson)

  以前，分区消除仅在计划期间发生，意味着许多连接和准备的查询不能使用分区消除。

- 分区表之间等值连接时，允许匹配的分区直接连接 (Ashutosh Bapat)

  该特性缺省是禁用的，可修改enable_partitionwise_join来启用。

- 允许分区表上的聚集函数对每个分区独立评估而后归并结果 (Jeevan Chalke, Ashutosh Bapat, Robert Haas)

  该特性缺省是禁用的，可修改enable_partitionwise_aggregate来启用。

- 允许postgres_fdw将聚集下推到那些作为分区的外部表 (Jeevan Chalke)

E.3.3.1.2. 并行查询

- 允许并行构建btree索引 (Peter Geoghegan, Rushabh Lathia, Heikki Linnakangas)
- 允许使用共享哈希表并行执行哈希连接 (Thomas Munro)
- 允许UNION并行运行每个SELECT，如果单个SELECT无法并行化的话 (Amit Khandekar, Robert Haas, Amul Sul)
- 允许使用并行工作者进行更有效的分区扫描 (Amit Khandekar, Robert Haas, Amul Sul)
- 允许将LIMIT传给并行工作者 (Robert Haas, Tom Lane)

  这样允许工作者减少返回结果，使用有针对性的索引扫描。

- 允许单次评估查询，例如WHERE子句聚集查询和目标列表中的函数并行化 (Amit Kapila, Robert Haas)
- 增加服务器参数parallel_leader_participation用来控制领导者是否执行子计划 (Thomas Munro)

  缺省启用，意味着领导者要执行子计划。

- 允许命令CREATE TABLE ... AS, SELECT INTO和CREATE MATERIALIZED VIEW并行化 (Haribabu Kommi)
- 改进有许多并行工作者时顺序扫描的性能 (David Rowley)
- 在EXPLAIN中增加并行工作者排序活动的报告 (Robert Haas, Tom Lane)

E.3.3.1.3. 索引

- 允许B-树索引包含不是搜索键或唯一约束的，但可用于仅索引扫描读的列 (Anastasia Lubennikova, Alexander Korotkov, Teodor Sigaev)

  该功能可用CREATE INDEX的新的INCLUDE子句来启用。它利用构造“覆盖索引”来优化特定类型的查询。可包含一些列，即使其数据类型B-树不支持。

- 改进索引不断增长时的性能 (Pavan Deolasee, Peter Geoghegan)
- 改进哈希索引扫描的性能 (Ashutosh Sharma)
- 为哈希、GiST和GIN索引添加谓词锁 (Shubham Barai)

  这样减少了序列化模式事务中序列化冲突的可能性。

E.3.3.1.3.1. SP-Gist

- 增加前缀匹配操作符text ^@ text，由SP-GiST支持此项功能 (Ildus Kurbangaliev)

  这类似于btree索引时使用var LIKE 'word%'，但它更有效。
- 允许使用SP-GiST索引多边形。(Nikita Glukhov, Alexander Korotkov)
- 允许SP-GiST使用叶子键的有损表示 (Teodor Sigaev, Heikki Linnakangas, Alexander Korotkov, Nikita Glukhov)

E.3.3.1.4. 优化器

- 改进统计信息最频值的选择 (Jeff Janes, Dean Rasheed)

  以前，最频值(MCV)的标识是基于他们与所有列值比较的频度。现在MCV的选择是基于他们与非MCV值比较的频度。这样就改进了均匀和不均匀分布下的算法的健壮性。
- 改进>=和<=的选择性评估 (Tom Lane)

  以前，此类案例如>和<分别使用同样的选择性评估，除非比较常数是MCV。该变化对使用BETWEEN在小区间查询特别有用。
- 在等效的地方将var =var简化为var IS NOT NULL (Tom Lane)

  这会带来更好的选择性评估。
- 改进优化器对EXISTS和NOT EXISTS查询的行计数估计 (Tom Lane)
- 使优化器负责评估代价和HAVING子句的选择性 (Tom Lane)

E.3.3.1.5. 一般性能

- 增加对查询计划的某些部分的即时(JIT)编译，以改进执行性能 (Andres Freund)

  该特性需LLVM可用。当前不是默认启用，即使在构建中支持也不是。
- 允许在可能时位图扫描执行仅索引扫描 (Alexander Kuzmenkov)
- 更新VACUUM时空闲空间映射 (Claudio Freire)

  这样允许空闲空间能更快地重用。
- 允许VACUUM避免不必要的索引扫描 (Masahiko Sawada, Alexander Korotkov)
- 改进多并发事务的提交性能 (Amit Kapila)
- 减少那些在目标列表中使用集返回函数的查询的内存使用 (Andres Freund)
- 改进聚集计算的速度 (Andres Freund)
- 允许postgres_fdw将使用连接的UPDATE和DELETE命令推送到外部服务器 (Etsuro Fujita)

  以前，仅推送非连接的UPDATE和DELETE命令。

- 增加Windows上的大页支持 (Takayuki Tsunakawa, Thomas Munro)

  该功能由huge_pages配置参数来控制。

E.3.3.1.6. 监控

- 在log_statement_stats,log_parser_stats,log_planner_stats和log_executor_stats的输出中显示内存使用情况 (Justin Pryzby, Peter Eisentraut)

- 增加pg_stat_activity.backend_type列以显示后台工作者类型 (Peter Eisentraut)

  该类型在ps输出中也可见。

- 使log_autovacuum_min_duration记录那些被同时删除而忽略掉的表 (Nathan Bossart)

E.3.3.1.6.1. 信息模式

- 增加与表约束和触发器相关的information_schema列 (Peter Eisentraut)

  特别是现在triggers.action_order,triggers.action_reference_old_table和triggers.action_reference_new_ta填有数据，而以前则总为空。此外，现在table_constraints.enforced虽然存在，但仍未填入有用数据。

E.3.3.1.7. 认证

- 允许服务器在搜索+绑定模式中指定更为复杂的LDAP规格要求 (Thomas Munro)

  特别是ldapsearchfilter允许使用LDAP属性组合进行模式匹配。

- 允许LDAP认证使用加密的LDAP (Thomas Munro)

  我们已经支持TLS上的LDAP，这通过用ldaptls=1来实现。新的TLS LDAP是加密LDAP，用ldapscheme=ldaps或ldapurl=ldaps://来启用。

- 改进LDAP错误日志(Thomas Munro)

E.3.3.1.8. 权限

- 增加启用文件系统访问的默认角色 (Stephen Frost)

  特别地，这些新角色是：pg_read_server_files,pg_write_server_files和pg_execute_server_program。这些角色现在还控制着谁可以使用服务器端COPY和file_fdw扩展。以前，仅超级用户才能使用这些功能，这仍然是默认行为。

- 允许用GRANT/REVOKE权限许可来控制文件系统函数的访问，而不是由超级用户检查来控制 (Stephen Frost)

  特别地，修改了这些函数：pg_ls_dir(),pg_read_file(),pg_read_binary_file(),pg_stat_file()。

- 使用GRANT/REVOKE控制对lo_import()和lo_export()的访问 (Michael Paquier, Tom Lane)

  以前，仅超级用户才有权访问这些函数。

  编译时选项ALLOW_DANGEROUS_LO_FUNCTIONS已被移除。

- 防止对postgres_fdw表进行无密码访问时，使用视图拥有者而非会话拥有者 (Robert Haas)

PostgreSQL仅允许超级用户对postgres_fdw表进行无密码的访问，例如通过peer访问。以前，会话拥有者必须为超级用户才允许访问；现在以检查视图拥有者代之。

- 修复视图上SELECT FOR UPDATE的无效的锁权限检查问题 (Tom Lane)

E.3.3.1.9. 服务器配置

- 增加服务器参数ssl_passphrase_command以允许提供SSL密钥文件的密语 (Peter Eisentraut)

  此外增加ssl_passphrase_command_supports_reload以详细说明SSL配置是否应重新载入，且服务器配置重新载入时是否调用ssl_passphrase_command。

- 增加存储参数toast_tuple_target，以在考虑用TOAST存储前控制最小的元组长度 (Simon Riggs)

  缺省的TOAST阈值不变。

- 允许以字节为单位制定内存和文件大小相关的服务器选项 (Beena Emerson)

  新单位后缀为“B”。另外已有的单位是“kB”，“MB”，“GB”和“TB”。

E.3.3.1.10. 预写日志 (WAL)

- 允许initdb时设置WAL文件大小 (Beena Emerson)

  以前，缺省值16MB只能在编译时改变。

- 仅为单个检查点保留WAL数据 (Simon Riggs)

  以前，为两个检查点保留WAL。

- 为提高压缩率，以零填充强制切换的WAL段文件的未使用部分 (Chapman Flack)

E.3.3.2. 库备份和流复制

- 使用流复制时，复制TRUNCATE活动 (Simon Riggs, Marco Nenciarini, Peter Eisentraut)
- 将准备事务信息传给逻辑复制订阅者 (Nikhil Sontakke, Stas Kelvich)
- 将非日志表、临时表和pg_internal.init文件从流式库备份中排除 (David Steele)

  没有必要拷贝这些文件。

- 允许在流式库备份时验证堆页面的校验和 (Michael Banck)
- 允许复制槽以编程方式前进，而不是被订阅者消费 (Petr Jelinek)

  这样允许在没必要消费内容时，复制槽有效前进。此功能通过pg_replication_slot_advance()实现。

- 将时间线信息加到backup_label文件中 (Michael Paquier)

  另增WAL时间线与backup_label匹配的检查。

- 将主机和端口连接信息加到pg_stat_wal_receiver系统视图中 (Haribabu Kommi)

E.3.3.3. 实用工具命令

- 允许ALTER TABLE用非空缺省值添加列，而不需重写表 (Andrew Dunstan, Serge Rielau)

  当缺省值为常数时启用该项功能。

- 允许用视图所基于的表来锁定视图 (Yugo Nagata)
- 允许ALTER INDEX为表达式索引设置统计信息收集目标 (Alexander Korotkov, Adrien Nayrat)

  在psql中，\d+现在显示索引的统计信息目标。
- 允许一条VACUUM或ANALYZE命令中指定多个表 (Nathan Bossart)

  此外，VACUUM中提及的任何表若使用列（字段）列表，则必须提供ANALYZE关键词；以前此类情况下ANALYZE是隐含在内的。
- 把括号括起来的选项语法增加到ANALYZE (Nathan Bossart)

  这类似于VACUUM支持的语法。
- 增加CREATE AGGREGATE选项以指定聚集的终结函数的行为 (Tom Lane)

  这有助于允许用户定义聚集函数的优化、用作窗口函数。

E.3.3.4. 数据类型

- 允许创建域数组 (Tom Lane)

  这也允许array_agg()用在域上。
- 支持符合类型上的域 (Tom Lane)

  也允许PL/Perl, PL/Python和PL/Tcl处理复合域函数参数和结果。亦改进了PL/Python的域处理。
- 增加从JSONB标量到数字和布尔数据类型的类型转换 (Anastasia Lubennikova)

E.3.3.5. 函数

- 增加SQL:2011指定的所有窗口函数帧选项 (Oliver Ford, Tom Lane)

  特别地，允许RANGE模式使用PRECEDING和FOLLOWING选择那些在指定正负偏移内有分组的行。增加GROUPS模式以包含加减该数量的对等组。帧排除语法也加进来了。
- 增加SHA-2哈希函数家族 (Peter Eisentraut)

  特别地，增加sha224(),sha256(), sha384(),sha512()。
- 增加对64位非加密哈希函数的支持 (Robert Haas, Amul Sul)
- 允许to_char()和to_timestamp()指定自UTC的时区偏移的小时和分钟 (Nikita Glukhov, Andrew Dunstan)

  这以TZH和TZM格式规范来实施。
- 增加搜索函数websearch_to_tsquery()，支持一种类似于Web搜索引擎使用的查询语法 (Victor Drobny, Dmitry Ivanov)
- 增加json(b)_to_tsvector()函数以创建用于匹配JSON/JSONB值的文本搜索查询 (Dmitry Dolgov)

E.3.3.6. 服务器端语言

- 增加SQL级过程，在过程中能够开始和提交其自身的事务 (Peter Eisentraut)

用新命令CREATE PROCEDURE创建过程，用CALL来调用。

新命令ALTER/DROP ROUTINE可以改变/删除所有类似程序的对象，包括过程、函数和聚集。

而且在CREATE OPERATOR和CREATE TRIGGER时，现在写FUNCTION比写PROCEDURE好，因为引用的对象必须是函数而非过程。但是，为了兼容性，仍然接受过去的语法。

- 将事务控制增加到PL/pgSQL, PL/Perl, PL/Python, PL/Tcl和SPI服务器端语言中 (Peter Eisentraut)

  事务控制仅在顶部事务层过程和仅包含其它DO和CALL块的嵌套DO和CALL块中可用。

- 增加将PL/pgSQL复合类型变量定义为非空、常数或带初始值的功能 (Tom Lane)

- 允许PL/pgSQL处理同一会话中发生在第一次和后来函数执行之间对复合类型的修改 (Tom Lane)

  以前这些情况产生错误。

- 增加jsonb_plpython扩展以转换JSONB到/自PL/Python类型 (Anthony Bykov)

- 增加jsonb_plperl扩展以转换JSONB到/自PL/Perl类型 (Anthony Bykov)

E.3.3.7. 客户端接口

- 修改libpq，以禁用默认的压缩 (Peter Eisentraut)

  现代的OpenSSL版本中，压缩已被禁用，故与这些库一起的libpq设置无效。

- 将DO CONTINUE选项加到ecpg的WHENEVER语句中 (Vinayak Pokale)

  这会产生C的continue语句，特定条件发生时将导致返回含有它的循环的顶部。

- 增加ecpg模式以启用Oracle Pro*C风格的字符数组处理方式。

  该模式用-C来启用。

E.3.3.8. 客户端应用程序

E.3.3.8.1. psql

- 增加psql命令\gdesc以显示查询结果中的列的名字和类型 (Pavel Stehule)

- 增加psql变量以报告查询活动或错误 (Fabien Coelho)

  特别地，这些新变量是ERROR,SQLSTATE,
  ROW_COUNT,LAST_ERROR_MESSAGE和LAST_ERROR_SQLSTATE。

- 允许psql测试一个变量是否存在 (Fabien Coelho)

  特别地，语法:{?variable_name}允许在\if语句中测试变量是否存在。

- 允许环境变量PSQL_PAGER控制psql的分页 (Pavel Stehule)

  该项功能允许psql的缺省分页器可指定为单独环境变量，与其它应用程序分页器分开。如果PSQL_PAGER未设置，PAGER仍然起作用。

- 使psql的\d+命令总显示表分区信息 (Amit Langote, Ashutosh Bapat)

以前如果无分区，分区表不会显示分区信息。现在也要显示哪些分区是如何划分的。

- 确保psql提示输入密码时报告确切的用户名 (Tom Lane)

  以前嵌入在URI中的-U和用户名的组合会导致错误的报告。当指定--password时，也会在密码提示之前禁止（显示）用户名。

- 允许没有先前输入时，使用quit和exit退出psql (Bruce Momjian)

  此外输入缓冲区非空时，若在一行中单独使用quit和exit，则打印如何退出的提示。对于help也添加了类似的提示。

- 在一行中单独输入了\q但被忽略时，让psql提示使用CTRL+D (Bruce Momjian)

  例如，当\q是在字符串中给出时，该命令并不退出。

- 改进ALTER INDEX RESET/SET的制表符 (Masahiko Sawada)

- 增加允许psql基于服务器版本适配其制表符自动补全查询的基础架构 (Tom Lane)

  以前对老版本服务器的制表符自动补全查询会失败。

E.3.3.8.2. pgbench

- 在pgbench中增加对NULL、布尔数和一些函数与操作符的表达式支持 (Fabien Coelho)
- 在pgbench中增加\if条件支持 (Fabien Coelho)
- 允许在pgbench变量名中使用非ASCII字符 (Fabien Coelho)
- 增加pgbench选项--init-steps以控制初始化步骤的执行 (Masahiko Sawada)
- 将一种近似于Zipfian分布的随机数发生器添加到pgbench (Alik Khilazhev)
- 允许在pgbench中设置随机数种子 (Fabien Coelho)
- 允许pgbench用pow()和power()来做幂运算 (Raúl Marín Rodríguez)
- 将哈希函数添加到pgbench (Ildar Musin)
- 当使用--latency-limit和--rate时，使pgbench统计更准确 (Fabien Coelho)

E.3.3.9. 服务器应用程序

- 在pg_basebackup增加一创建命名复制槽的选项 (Michael Banck)

  使用WAL流式方法时，选项--create-slot创建命名复制槽(--slot)。

- 允许initdb设置对数据目录的组读取访问 (David Steele)

  该功能用新的initdb选项--allow-group-access来实现。在运行initdb之前，管理员也可以对空数据目录设置组权限。服务器变量data_directory_mode允许读取数据目录组权限。

- 增加pg_verify_checksums工具以验证离线时数据库校验和 (Magnus Hagander)
- 允许pg_resetwal通过--wal-segsize修改WAL大小 (Nathan Bossart)
- 在pg_resetwal和pg_controldata中增加长选项 (Nathan Bossart, Peter Eisentraut)

- 为了测试，将--no-sync选项添加到pg_receivewal以防止同步WAL写 (Michael Paquier)
- 将--endpos选项添加到pg_receivewal以指定WAL接收何时停止 (Michael Paquier)
- 允许pg_ctl向进程发送SIGKILL信号 (Andres Freund)

  以前该功能是不支持的，原因是担心可能被误用。
- 减少pg_rewind复制的文件数量 (Michael Paquier)
- 防止pg_rewind从root用户运行 (Michael Paquier)

E.3.3.9.1. pg_dump, pg_dumpall, pg_restore

- 增加pg_dumpall选项--encoding用来控制输出的编码 (Michael Paquier)

  pg_dump已有此选项。
- 增加pg_dump选项--load-via-partition-root，强制将数据加载到分区的根表中，而不是原来的分区 (Rushabh Lathia)

  如果系统要加载到不同的排序规则定义或字节序的库，相比以前而言可能需要将行存储在不同的分区。这种情况下就很有用。
- 增加禁止转储和恢复数据库对象注释的选项 (Robins Tharakan)

  新的pg_dumppg_dumpall和pg_restore选项是--no-comments。

E.3.3.10. 源代码

- 增加PGXS支持以安装头文件 (Andrew Gierth)

  这样可支持创建依赖其他模块的扩展模块。以前没有容易的方法让依赖模块找到所引用的模块的头文件。现有的几个定义数据类型的contrib模块已经调整，安装了相关文件。PL/Perl和PL/Python现在也安装了它们的头文件，以支持语言转换模块的创建。
- 安装errcodes.txt，以允许扩展访问PostgreSQL所知的错误代码列表 (Thomas Munro)
- 将文档转成了DocBook XML格式 (Peter Eisentraut, Alexander Lakhin, Jürgen Purtz)

  为了与以前分支兼容，文件名仍使用sgml扩展名。
- 在合适的平台上（大多数是这样），使用stdbool.h定义bool类型 (Peter Eisentraut)

  这样就消除了需包含stdbool.h的扩展模块的编码危险。
- 彻底修改初始系统目录内容定义的方法 (John Naylor)

  初始数据现在用Perl数据结构表示，使之更易进行机械式处理。
- 防止扩展创建带引号的值列表自定义服务器参数 (Tom Lane)

  该功能目前不支持，因为即使在扩展加载之前也需要参数属性的知识。
- 使用SCRAM认证时，增加使用通道绑定的功能 (Michael Paquier)

  通道绑定的目的是防止中间人攻击，但SCRAM无法阻止，除非它能被强制激活。不幸的是，在libpq中无法做到。期望在未来的libpq版本和不是由libpql构建的接口如JDBC中支持它。
- 允许后台工作者连接到通常不允许连接的数据库 (Magnus Hagander)

- ARMv8上，增加硬件CRC计算的支持 (Yuqi Gu, Heikki Linnakangas, Thomas Munro)
- 加速用OID对内置函数的查找 (Andres Freund)

  以前的二分法查找被替换为查找数组。
- 加速查询结果的构造 (Andres Freund)
- 改进系统缓存的访问速度 (Andres Freund)
- 增加代内存分配器以对顺序的分配/释放进行优化 (Tomas Vondra)

  这样会减少逻辑解码的内存使用。
- 使VACUUM对 pg_class.reltuples的计算与ANALYZE对其的计算一致 (Tomas Vondra)
- 升级以使用perltidy的版本20170521 (Tom Lane, Peter Eisentraut)

E.3.3.11. 附加模块

- 允许pg_prewarm启动时恢复以前的共享缓冲区内容 (Mithun Cy, Robert Haas)

  该功能实现的是让pg_prewarm在服务器运行期间和关闭时偶尔将共享缓冲区的关系和块数量的数据保存到磁盘。
- 增加pg_trgm函数strict_word_similarity()以计算整个单词的相似性 (Alexander Korotkov)

  为此，已有函数word_similarity()，但其设计是为了找到单词相似的部分，而strict_word_similarity()则是计算整个单词的相似性。
- 允许创建能用于在citext列上进行LIKE的索引 (Alexey Chernyshov)

  这样做的话，索引就必须用citext_pattern_ops操作符类创建。
- 允许btree_gin对bool, bpchar, name和uuid数据类型进行索引 (Matheus Oliveira)
- 允许cube和seg扩展使用GiST索引执行仅索引扫描 (Andrey Borodin)
- 允许用~>操作符接收负的立方体坐标 (Alexander Korotkov)

  这对在查找降序的坐标时做KNN-GiST搜索很有用。
- 将越南文处理加到unaccent扩展 (Dang Minh Huong, Michael Paquier)
- 改进amcheck，检查每个堆元组有无索引入口 (Peter Geoghegan)
- 使adminpack使用新的缺省文件系统访问角色 (Stephen Frost)

  以前，只有超级用户才能调用adminpack函数；现在检查角色的权限。
- 将pg_stat_statement的查询ID增宽至64位 (Robert Haas)

  这将大大减少查询ID出现哈希冲突的可能性。现在查询ID可能显示为负值。
- 移除contrib/start-scripts/osx脚本，因为其不再建议使用（contrib/start-scripts/macos代替） (Tom Lane)
- 移除chkpass扩展 (Peter Eisentraut)

  该扩展不再视为是一种有用的安全工具或如何编写扩展的示例。

E.3.4. 致谢

作为补丁的作者、提交者、审阅者、测试者或问题报告者，以下人员（按字母顺序排列）对该发行版做出了贡献。

Abhijit Menon-Sen
Adam Bielanski
Adam Brightwell
Adam Brusselback
Aditya Toshniwal
Adrián Escoms
Adrien Nayrat
Akos Vandra
Aleksander Alekseev
Aleksandr Parfenov
Alexander Korotkov
Alexander Kukushkin
Alexander Kuzmenkov
Alexander Lakhin
Alexandre Garcia
Alexey Bashtanov
Alexey Chernyshov
Alexey Kryuchkov
Alik Khilazhev
Álvaro Herrera
Amit Kapila
Amit Khandekar
Amit Langote
Amul Sul
Anastasia Lubennikova
Andreas Joseph Krogh
Andreas Karlsson
Andreas Seltenreich
André Hänsel
Andrei Gorita
Andres Freund
Andrew Dunstan
Andrew Fletcher
Andrew Gierth
Andrew Grossman
Andrew Krasichkov
Andrey Borodin
Andrey Lizenko
Andy Abelisto
Anthony Bykov
Antoine Scemama
Anton Dignös
Antonin Houska
Arseniy Sharoglazov
Arseny Sher
Arthur Zakirov
Ashutosh Bapat
Ashutosh Sharma
Ashwin Agrawal
Asim Praveen
Atsushi Torikoshi
Badrul Chowdhury
Balazs Szilfai

Basil Bourque
Beena Emerson
Ben Chobot
Benjamin Coutu
Bernd Helmle
Blaz Merela
Brad DeJong
Brent Dearth
Brian Cloutier
Bruce Momjian
Catalin Iacob
Chad Trabant
Chapman Flack
Christian Duta
Christian Ullrich
Christoph Berg
Christoph Dreis
Christophe Courtois
Christopher Jones
Claudio Freire
Clayton Salem
Craig Ringer
Dagfinn Ilmari Mannsåker
Dan Vianello
Dan Watson
Dang Minh Huong
Daniel Gustafsson
Daniel Vérité
Daniel Westermann
Daniel Wood
Darafei Praliaskouski
Dave Cramer
Dave Page
David Binderman
David Carlier
David Fetter
David G. Johnston
David Gould
David Hinkle
David Pereiro Lagares
David Rader
David Rowley
David Steele
Davy Machado
Dean Rasheed
Dian Fay
Dilip Kumar
Dmitriy Sarafannikov
Dmitry Dolgov
Dmitry Ivanov
Dmitry Shalashov
Don Seiler
Doug Doole
Doug Rady
Edmund Horner
Eiji Seki
Elvis Pranskevichus
Emre Hasegeli

Erik Rijkers
Erwin Brandstetter
Etsuro Fujita
Euler Taveira
Everaldo Canuto
Fabien Coelho
Fabrízio de Royes Mello
Feike Steenbergen
Frits Jalvingh
Fujii Masao
Gao Zengqi
Gianni Ciolli
Greg Stark
Gunnlaugur Thor Briem
Guo Xiang Tan
Hadi Moshayedi
Hailong Li
Haribabu Kommi
Heath Lord
Heikki Linnakangas
Hugo Mercier
Igor Korot
Igor Neyman
Ildar Musin
Ildus Kurbangaliev
Ioseph Kim
Jacob Champion
Jaime Casanova
Jakob Egger
Jean-Pierre Pelletier
Jeevan Chalke
Jeevan Ladhe
Jeff Davis
Jeff Janes
Jeremy Evans
Jeremy Finzel
Jeremy Schneider
Jesper Pedersen
Jim Nasby
Jimmy Yih
Jing Wang
Jobin Augustine
Joe Conway
John Gorman
John Naylor
Jon Nelson
Jon Wolski
Jonathan Allen
Jonathan S. Katz
Julien Rouhaud
Jürgen Purtz
Justin Pryzby
KaiGai Kohei
Kaiting Chen
Karl Lehenbauer
Keith Fiske
Kevin Bloch
Kha Nguyen

Kim Rose Carlsen
Konstantin Knizhnik
Kuntal Ghosh
Kyle Samson
Kyotaro Horiguchi
Lætitia Avrot
Lars Kanis
Laurenz Albe
Leonardo Cecchi
Liudmila Mantrova
Lixian Zou
Lloyd Albin
Luca Ferrari
Lucas Fairchild
Lukas Eder
Lukas Fittl
Magnus Hagander
Mai Peng
Maksim Milyutin
Maksym Boguk
Mansur Galiev
Marc Dilger
Marco Nenciarini
Marina Polyakova
Mario de Frutos Dieguez
Mark Cave-Ayland
Mark Dilger
Mark Wood
Marko Tiikkaja
Markus Winand
Martín Marqués
Masahiko Sawada
Matheus Oliveira
Matthew Stickney
Metin Doslu
Michael Banck
Michael Meskes
Michael Paquier
Michail Nikolaev
Mike Blackwell
Minh-Quan Tran
Mithun Cy
Morgan Owens
Nathan Bossart
Nathan Wagner
Neil Conway
Nick Barnes
Nicolas Thauvin
Nikhil Sontakke
Nikita Glukhov
Nikolay Shaplov
Noah Misch
Noriyoshi Shinoda
Oleg Bartunov
Oleg Samoilov
Oliver Ford
Pan Bian
Pascal Legrand

Patrick Hemmer
Patrick Krecker
Paul Bonaud
Paul Guo
Paul Ramsey
Pavan Deolasee
Pavan Maddamsetti
Pavel Golub
Pavel Stehule
Peter Eisentraut
Peter Geoghegan
Petr Jelínek
Petru-Florin Mihancea
Phil Florent
Philippe Beaudoin
Pierre Ducroquet
Piotr Stefaniak
Prabhat Sahu
Pu Qun
QL Zhuo
Rafia Sabih
Rahila Syed
Rainer Orth
Rajkumar Raghuwanshi
Raúl Marín Rodríguez
Regina Obe
Richard Yen
Robert Haas
Robins Tharakan
Rod Taylor
Rushabh Lathia
Ryan Murphy
Sahap Asci
Samuel Horwitz
Scott Ure
Sean Johnston
Shao Bret
Shay Rojansky
Shubham Barai
Simon Riggs
Simone Gotti
Sivasubramanian Ramasubramanian
Stas Kelvich
Stefan Kaltenbrunner
Stephen Froehlich
Stephen Frost
Steve Singer
Steven Winfield
Sven Kunze
Taiki Kondo
Takayuki Tsunakawa
Takeshi Ideriha
Tatsuo Ishii
Tatsuro Yamada
Teodor Sigaev
Thom Brown
Thomas Kellerer
Thomas Munro

Thomas Reiss
Tobias Bussmann
Todd A. Cook
Tom Kazimiers
Tom Lane
Tomas Vondra
Tomonari Katsumata
Torsten Grust
Tushar Ahuja
Vaishnavi Prabakaran
Vasundhar Boddapati
Victor Drobny
Victor Wagner
Victor Yegorov
Vik Fearing
Vinayak Pokale
Vincent Lachenal
Vitaliy Garnashevich
Vitaly Burovoy
Vladimir Baranoff
Xin Zhang
Yi Wen Wong
Yorick Peterse
Yugo Nagata
Yuqi Gu
Yura Sokolov
Yves Goergen
Zhou Digoal

E.4. 先前的版本

有关先前发行版分支的发行说明，请参阅 PostgreSQL[3] WEB站点。在版本11发布时,支持的先前版本分支如下:

- PostgreSQL 10: https://www.postgresql.org/docs/10/release.html [4]
- PostgreSQL 9.6: https://www.postgresql.org/docs/9.6/release.html [5]
- PostgreSQL 9.5: https://www.postgresql.org/docs/9.5/release.html [6]
- PostgreSQL 9.4: https://www.postgresql.org/docs/9.4/release.html [7]
- PostgreSQL 9.3: https://www.postgresql.org/docs/9.3/release.html [8]

旧版本分支的发布说明可以从 https://www.postgresql.org/docs/manuals/archive/ [9]找到

[3] https://www.postgresql.org/
[4] https://www.postgresql.org/docs/10/release.html
[5] https://www.postgresql.org/docs/9.6/release.html
[6] https://www.postgresql.org/docs/9.5/release.html
[7] https://www.postgresql.org/docs/9.4/release.html
[8] https://www.postgresql.org/docs/9.3/release.html
[9] https://www.postgresql.org/docs/manuals/archive/

附录 F. 额外提供的模块

这个附录和下一个附录介绍在PostgreSQL发布的contrib目录中能找到的模块。包括移植工具、分析工具和插件特性，它们不是 PostgreSQL 核心系统的一部分，主要因为只有很少的用户会用到或者是还处于实验阶段。但这不会影响它们的使用。

这个附录覆盖了在contrib中能找到的扩展和其他服务器插件模块。附录 覆盖了工具程序。

当从源码发布包编译时，这些组件不会自动被编译，除非编译了“world”目标（见步骤 2）。 可以在已配置的源代码树中的contrib路径下，通过下面的命令编译安装：

```
make
make install
```

或者只在选中模块的子目录下编译和安装。许多模块都有回归测试，可以通过下面的命令在安装之前运行测试：

```
make check
```

或者在一个PostgreSQL服务器正在运行时，运行

```
make installcheck
```

如果是用的是预打包版的PostgreSQL，这些模块通常可以作为一个单独的子包来获得，如postgresql-contrib。

许多模块提供新的用户自定义函数、操作符或数据类型。在已经安装了代码之后，为了使用这些模块，需要在数据库系统中注册新的 SQL 对象。在PostgreSQL 9.1 及之后的版本中，可以通过执行一个CREATE EXTENSION命令来完成。在一个新的数据库中，你可以简单地

```
CREATE EXTENSION module_name;
```

这个命令必须由一个数据库超级用户运行。这只会把新的 SQL 对象注册在当前数据库中，因此你需要在每一个你希望使用该模块功能的数据库中执行这个命令。另外，可以在template1数据库中运行这个命令以便该扩展能被默认地复制到后续创建的数据库中。

很多模块允许你将它们的对象安装在你选择的一个模式中。要这样做，需要将SCHEMA schema_name加入到CREATE EXTENSION命令中。默认情况下，这些对象将被放置在你的当前创建目标模式中，通常是public。

如果你的数据库是从一个PostgreSQL 9.1 之前版本的转储载入而来，并且你已经在其中使用了一个 9.1 之前版本的该模块，你应该使用

```
CREATE EXTENSION module_name FROM unpackaged;
```

这会把该 9.1 之前的模块对象更新到一个正确的扩展对象。未来对该模块的更新将由ALTER EXTENSION管理。更多关于扩展更新的信息，请见第 38.16 节

不过注意，有一些这样的模块不是这种意义上的“扩展”，而是以某种其他方式被载入到服务器，例如使用shared_preload_libraries方式。每个模块详见其文档。

F.1. adminpack

adminpack 提供了一些支持函数，pgAdmin 和其他管理工具会用这些函数来提供额外的功能，例如服务器日志文件的远程管理。所有这些函数默认只允许由超级用户使用，但是可以通过使用 GRANT 命令允许其他用户使用。

表 F.1中展示的函数提供了向运行着服务器的机器上的文件进行写入的途径（另见表 9.88中的函数，它们提供了 只读的访问途径）。对于这些函数，只有位于数据库集簇目录中的文件才能被访问（除非用户是一个超级用户或者被指定了 pg_read_server_files 或者 pg_write_server_files 角色之一），不过允许使用相对路径和绝对路径。

表 F.1. adminpack 函数

名称	返回类型	描述
pg_catalog.pg_file_write(filename text, data text, append boolean)	bigint	向一个文本文件写入或者追加
pg_catalog.pg_file_rename(oldname text, newname text [, archivename text])	boolean	重命名一个文件
pg_catalog.pg_file_unlink(filename text)	boolean	移除一个文件
pg_catalog.pg_logdir_ls()	setof record	列出在 log_directory 目录中的日志文件

pg_file_write 把指定的 data 写入到由 filename 命名的文件中。如果 append 为假，文件不能已经存在。如果 append 为真，该文件 可能已经存在，并且如果存在就会被追加。这个函数返回写入的字节数。

pg_file_rename重命名一个文件。如果 archivename被省略或者为 NULL，它简单地把 oldname重命名为newname（不能 已经存在）。如果提供了archivename，该函数首先把newname重命名为archivename（不 能已经存在），然后把oldname重命名为 newname。当第二次重命名失败时，这个函数会在报告 错误之前尝试把archivename重命名成 newname。成功时函数返回真，如果源文件不存在或者 不可写则返回假，其他情况下会抛出错误。

pg_file_unlink移除指定的文件。成功时函数返回真， 如果指定的文件不存在或者unlink()调用失败则返回 假，其他情况下会抛出错误。

pg_logdir_ls返回log_directory 目录中所有日志文件的开始时间戳以及路径名。要使用这个函数， log_filename参数必须具有其默认设置（ postgresql-%Y-%m-%d_%H%M%S.log）。

F.2. amcheck

amcheck 模块提供的函数让用户能验证关系结构的逻辑一致性。如果结构有效，则不会发生错误。

这些函数验证特定关系的结构表达中的各种不变条件。索引扫描以及其他重要操作背后的访问方法的正确性都要依仗这些不变条件的成立。例如，在这些函数中，有一些负责验证所有B树页面中的项都按照“逻辑”顺序（比如，对于text上的B树索引，索引元组应该按照词典顺序排列）摆放。如果特定的不变条件由于某种原因无法成立，则我们可以预料受影响页面上的二分搜索将无法正确地引导索引扫描，最终导致SQL查询得到错误的答案。

验证过程采用索引扫描自身使用的同种过程来执行，这些过程可能是用户定义的操作符类代码。例如，B树索引验证依赖于由一个或者多个B树支持函数1例程构成的比较。操作符类支持函数的详情请见第 38.15.3 节。

amcheck函数只能由超级用户使用。

F.2.1. 函数

bt_index_check(index regclass, heapallindexed boolean) returns void

bt_index_check测试一个B树索引，检查各种不变条件。用法实例：

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed =>
 i.indisunique),
               c.relname,
               c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Don't check temp tables, which may be from another session:
AND c.relpersistence != 't'
-- Function may throw an error when this is omitted:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
 bt_index_check |             relname              | relpages
----------------+----------------------------------+----------
                | pg_depend_reference_index        |       43
                | pg_depend_depender_index         |       40
                | pg_proc_proname_args_nsp_index   |       31
                | pg_description_o_c_o_index       |       21
                | pg_attribute_relid_attnam_index  |       14
                | pg_proc_oid_index                |       10
                | pg_attribute_relid_attnum_index  |        9
                | pg_amproc_fam_proc_index         |        5
                | pg_amop_opr_fam_index            |        5
                | pg_amop_fam_strat_index          |        5
(10 rows)
```

这个例子中的会话执行对数据库“test”中10个最大目录索引的验证。对于唯一索引会要求验证堆元组是否有对应的索引元组存在。由于没有错误报出，所有的被测索引都处于逻辑一致的状态。自然地，很容易将这个查询改为对支持验证的数据库中的每一个索引调用bt_index_check。

bt_index_check要求目标索引及其所属的堆关系上的AccessShareLock。这种锁模式与简单SELECT语句在关系上所要求的锁模式相同。bt_index_check不验证跨越父子关系的不变条件，但是在heapallindexed为true时将验证所有堆元组是否作为索引中的索引元组存在。当在生产环境中要求一个使用bt_index_check的例程进行轻量化损坏测试时，它常常需要在验证彻底性和减小对应用性能及可用性的影响之间做出权衡。

bt_index_parent_check(index regclass, heapallindexed boolean) returns void

bt_index_parent_check测试一个B树索引，检查多种不变条件。可选地，当heapallindexed参数为true时，该函数验证所有应该在索引中找到的堆元组的存在，同时验证在索引结构中没有缺失的下链。bt_index_parent_check能够执行的检查是bt_index_check能执行的检查的超集。bt_index_parent_check可以被想成是bt_index_check的一种更全面的变体：和bt_index_check不同，bt_index_parent_check还检查跨越父/子关系的不变条件。如果找到逻辑不一致或者其他问题，bt_index_parent_check遵循通常的报错习惯。

bt_index_parent_check要求目标索引上的一个ShareLock（还要求对关系上的一个ShareLock）。这些锁阻止来自INSERT、UPDATE以及DELETE命令的并发数据修改。这些

锁同时防止底层关系被并发的VACUUM以及其他工具命令处理。注意该函数只在其运行期间而不是整个事务期间持有锁。

bt_index_parent_check的额外验证更有可能检测到多种病态的情况。这些情况可能涉及到被查索引使用的一种不正确实现的B-树操作符类，或者说不定是底层B-树索引访问方法代码中未被发现的缺陷。注意与bt_index_check不同，当热备模式被启用时（即在只读的物理复制机上）不能使用bt_index_parent_check。

F.2.2. 可选的heapallindexed验证

当验证函数的heapallindexed参数为true时，会针对与目标索引关系关联的表执行一个额外的验证过程。这种验证由一个“假的”CREATE INDEX操作组成，它针对一个临时的、内存中的汇总结构（根据需要在基础的第一阶段验证过程中建立）检查所有假想的新索引元组的存在。这个汇总结构对目标索引中的每一个元组“采集指纹”。heapallindexed验证背后的高层原则是：等效于现有目标索引的新索引必须仅拥有能在现有结构中找得到的项。

额外的heapallindexed阶段会增加明显的开销：验证的时间通常将会延长几倍。不过，在执行heapallindexed验证时，所要求的关系级锁没有变化。

这一汇总结构的尺寸以maintenance_work_mem为界。为了确保对于每个堆元组应该存在于索引中这一检测有不超过2%的失效概率能检测到不一致，每个元组需要大约2个字节的内存。因为每个元组可用的内存变少，错失一处不一致的概率就会慢慢增加。这种方法显著地限制了验证的开销，但仅仅略微降低了检测到问题的概率，对于将验证当作例行维护任务的安装来说更是如此。对于每一次新的验证尝试，任何单一的缺失或者畸形元组都有新的机会被检测到。

F.2.3. 有效地使用amcheck

amcheck对于检测多种数据页面校验和无法捕捉到的失效模式非常有效。包括：

- 由不正确的操作符类实现导致的结构性不一致。

  这包括操作系统排序规则的比较规则变化导致的问题。text之类的可排序类型数据的比较必须是不变的（正如用于B-树索引扫描的所有比较必须不变一样），这意味着操作系统排序规则必须保持不变。但是在很少的情况下，操作系统排序规则的更新会导致这些问题。更常见的，主服务器和后备服务器之间排序顺序的不一致会相互牵连，这可能是因为使用的主操作系统版本不一致。这类不一致通常仅出现在后备服务器上，因此通常也仅能在后备服务器上检测到。

  如果这类问题出现，则它可能不会影响使用受影响排序规则排序的每一个索引，其原因是被索引值可能正好具有与行为不一致无关的相同的绝对顺序。关于PostgreSQL如何使用操作系统locale和排序规则的进一步细节请参考第 23.1 和第 23.2 节

- 索引和被索引的对关系之间的结构不一致（在执行heapallindexed验证时）。

  在普通操作时没有将索引针对其对关系进行交叉检查。堆损坏的症状可能是很微妙的。

- 由于底层PostgreSQL访问方法代码、排序代码或者事务管理代码中（假想的）未发现的缺陷导致的损坏。

  在测试可能引入逻辑不一致的PostgreSQL新特性或者被提议的特性时，索引的结构完整性自动验证扮演了重要角色。表结构、相关的可见性和事务状态信息的验证扮演了类似的角色。一种显而易见的测试策略是在运行标准回归测试时持续地调用amcheck函数。运行这些测试的详情请参考第 33.1 节

- 正巧没有开启校验和的文件系统或者存储子系统故障。

  注意，如果在访问块时仅有一次共享缓存命中，验证时amcheck会在检查表示在某个共享内存缓冲区中的页面。因此，amcheck没有必要在验证时检查从文件系统读出的数据。注

意当校验和被启用时，如果一个损坏的块被读取到缓冲区中，amcheck可能会由于校验和失效而产生错误。

- 有缺陷的RAM或者内存子系统导致的损坏。

  PostgreSQL无法提供针对可更正内存错误的保护并且它假定用户使用的是具有工业标准纠错码（ECC）或更好保护技术的RAM。不过，ECC内存通常只能免疫单个位错误，并且不应该假定它能提供对导致内存损坏失效的绝对保护。

  在执行heapallindexed验证时，通常有大幅增加的机会可以检测单个位错误，因为会测试严格的二元等值并且会在堆中测试被索引属性。

通常，amcheck仅能证明损坏的存在，但它无法证明损坏不存在。

F.2.4. 修复损坏

amcheck没有产生与损坏相关的错误绝不应该被当做假阳性。amcheck会在（定义上）应该绝不会发生的情况中抛出错误，因此常常需要对amcheck错误进行仔细地分析。

对于amcheck检测到的问题没有一般性的修复方法。应该寻找产生不变条件违背的根本原因。在诊断amcheck检测到的损坏时，pageinspect可能会扮演一个非常有用的角色。REINDEX在修复损坏过程中可能无法起到效果。

F.3. auth_delay

auth_delay使得服务器在报告鉴定失败之前短暂地停顿一会儿，这使得对数据库口令的蛮力攻击更加困难。注意它无助于组织拒绝服务攻击，甚至可能会加剧它们，因为在报告鉴定失败之前等待的进程仍然要消耗连接。

要使之工作，该模块必须通过postgresql.conf中的shared_preload_libraries载入。

F.3.1. 配置参数

auth_delay.milliseconds (int)

在报告鉴定失败之前等待的毫秒数。默认值为0。

这些参数必须在postgresql.conf中设置。典型的使用是：

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.4. auto_explain

auto_explain模块提供了一种方式来自动记录慢速语句的执行计划，而不需 要手工运行EXPLAIN。这在大型应用中追踪未被优化的查询时有用。

该模块没有提供 SQL 可访问的函数。要使用它，简单地将它载入服务器。你可以把它载入到一个单独的 会话：

```
LOAD 'auto_explain';
```

（你必须作为超级用户来这样做）。更典型的用法是通过在postgresql.conf的session_preload_libraries或shared_preload_libraries参数中包括auto_explain将它预先 载入到某些或者所有会话中。然后你就可以追踪那些出乎意料地慢的查询，而不管它们何时发生。当然为 此会付出一些额外的负荷作为代价。

F.4.1. 配置参数

有几个配置参数用来控制auto_explain的行为。注意默认行为是什么也不做，因此如果你想要任何结果就必须至少设置auto_explain.log_min_duration。

auto_explain.log_min_duration (integer)

auto_explain.log_min_duration是最小语句执行时间（以毫秒计），这将导致语句的计划被记录。设置这个参数为零将记录所有计划。负一（默认值）禁用记录计划。例如，如果你将它设置为250ms，则所有运行时间等于或超过 250ms 的语句将被记录。只有超级用户能够改变这个设置。

auto_explain.log_analyze (boolean)

当一个执行计划被记录时，auto_explain.log_analyze导致EXPLAIN ANALYZE输出（而不仅仅是EXPLAIN输出）被打印。默认情况下这个参数是关闭的。只有超级用户能够改变这个设置。

> **注意**
>
> 当这个参数为打开时，对所有被执行的语句将引起对每个计划节点的计时，不管它们是否运行得足够长以至于被记录。这可能对性能有极度负面的影响。

auto_explain.log_buffers (boolean)

当一个执行计划被记录时，auto_explain.log_buffers控制是否打印 缓冲区使用统计信息；它等效于EXPLAIN的BUFFERS选项。除非 auto_explain.log_analyze参数被设置，否则这个参数没有效果。这个参数默 认情况下是关闭的。只有超级用户能够改变这个设置。

auto_explain.log_timing (boolean)

当一个执行计划被记录时，auto_explain.log_timing控制是否打印每 个结点上的计时信息；它等效于EXPLAIN的TIMING选项。重复读取 系统锁的开销在某些系统上可能会显著地拖慢查询，因此当只需要实际行计数而非确切时间时，关闭 这个参数将会很有帮助。只有当auto_explain.log_analyze也被启用 时这个参数才有效。这个参数默认情况下是打开的。只有超级用户能够改变这个设置。

auto_explain.log_triggers (boolean)

当一个执行计划被记录时，auto_explain.log_triggers会导致触发 器执行统计信息被包括在内。只有当auto_explain.log_analyze也被 启用时这个参数才有效。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

auto_explain.log_verbose (boolean)

当一个执行计划被记录时，auto_explain.log_verbose控制是否打印很长的详细信息；它等效于EXPLAIN的VERBOSE选项。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

auto_explain.log_format (enum)

auto_explain.log_format选择要使用的EXPLAIN输出格式。允许的值是text、xml、json和yaml。默认是文本形式。只有超级用户能够改变这个设置。

auto_explain.log_nested_statements (boolean)

auto_explain.log_nested_statements导致嵌套语句（在一个函数内执行的语句）会被考虑在记录范围之内。当它被关闭时，只有顶层查询计划被记录。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

auto_explain.sample_rate (real)

auto_explain.sample_rate会让 auto_explain 只解释每个会话中的一部分语句。默认值为 1，表示解释所有的查询。在嵌套 语句的情况下，要么所有语句都被解释，要么一个也不被解释。只有超级用户 能够更改这个设置。

在普通用法中，这些参数都在postgresql.conf中设置，不过超级用户可以在他们自己的会话中随时修改这些参数。典型的用法可能是：

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. 例子

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

这可能会产生这样的日志输出：

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1
 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594
 rows=92 loops=1)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual
 time=0.016..0.140 rows=255 loops=1)
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238
 rows=92 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4)
 (actual time=0.008..3.187 rows=92 loops=1)
                      Filter: indisunique
```

F.4.3. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>

F.5. bloom

bloom提供了一种基于布鲁姆过滤器[1]的索引访问方法。

布鲁姆过滤器是一种空间高效的数据结构，它被用来测试一个元素是否为一个集合的成员。在索引访问方法的情况下，它可以通过尺寸在索引创建时决定的签名来快速地排除不匹配的元组。

签名是被索引属性的一种有损表达，并且因此容易报告伪肯定，也就是说对于一个不在集合中的元素有可能报告该元素在集合中。因此索引搜索结果必须使用来自堆项的实际属性值进行再次检查。较大的签名可以降低伪肯定的几率并且减少无用的堆访问的次数，但是这显然会让索引更大且扫描起来更慢。

当表具有很多属性并且查询可能会测试其中任意组合时，这种类型的索引最有用。传统的 btree 索引比布鲁姆索引更快，但是需要很多 btree 索引来支持所有可能的查询，而对于布鲁姆索引来说只需要一个即可。不过要注意 bloom 索引只支持等值查询，而 btree 索引还能执行不等和范围搜索。

F.5.1. 参数

bloom索引在其WITH子句中接受下列参数：

length

每个签名（索引项）的长度位数，它会被圆整成为最近的16的倍数。默认是80位，最长是4096位。

col1 — col32

从每一个索引列产生的位数。每个参数的名字表示它所控制的索引列的编号。默认是2位，最大是4095位。没有实际使用的索引列的参数会被忽略。

F.5.2. 例子

这是一个创建布鲁姆索引的例子：

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

该索引是用长度为 80 位的签名所创建，其中属性 i1 和 i2 被映射为 2 位，属性 i3 被映射为 4 位。我们可以省略length、col1和col2说明，因为它们都有默认值。

这里是布鲁姆索引定义和使用的更完整的例子，其中还与等效的 btree 做了对比。布鲁姆索引比 btree 索引更小，并且效率更高。

```
=# CREATE TABLE tbloom AS
   SELECT
     (random() * 1000000)::int as i1,
     (random() * 1000000)::int as i2,
     (random() * 1000000)::int as i3,
     (random() * 1000000)::int as i4,
     (random() * 1000000)::int as i5,
     (random() * 1000000)::int as i6
   FROM
  generate_series(1,10000000);
SELECT 10000000
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
```

[1] https://en.wikipedia.org/wiki/Bloom_filter

```
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 153 MB
(1 row)
=# CREATE index btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 387 MB
(1 row)
```

在这个大表上的顺序扫描需要很长时间：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on tbloom  (cost=0.00..213694.08 rows=1 width=24) (actual
 time=1445.438..1445.438 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 10000000
 Planning time: 0.177 ms
 Execution time: 1445.473 ms
(5 rows)
```

因此规划器通常将尽可能选择索引扫描。使用 btree 索引，我们可以得到这样的结果：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Only Scan using btreeidx on tbloom  (cost=0.56..298311.96 rows=1
 width=24) (actual time=445.709..445.709 rows=0 loops=1)
   Index Cond: ((i2 = 898732) AND (i5 = 123451))
   Heap Fetches: 0
 Planning time: 0.193 ms
 Execution time: 445.770 ms
(5 rows)
```

在处理这类搜索时，bloom 比 btree 表现得更好：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=178435.39..178439.41 rows=1 width=24) (actual
 time=76.698..76.698 rows=0 loops=1)
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 2439
   Heap Blocks: exact=2408
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..178435.39 rows=1 width=0)
 (actual time=72.455..72.455 rows=2439 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
 Planning time: 0.475 ms
 Execution time: 76.778 ms
(8 rows)
```

注意其中相对较大的伪肯定数：有 2439 行被选中进行堆访问但实际却不匹配查询。我们可以通过指定更大的签名长度来减少这种情况。在这个例子中，用length=200创建索引可以把伪肯定数减小到 55，但是这同时会使索引尺寸翻倍（306MB）并且最终使这个查询变慢（总体 125 ms）。

现在，btree 搜索的主要问题是，当搜索条件不约束前几个索引列时，btree 的效率不好。对于 btree 更好的策略是在每一列上创建一个独立的索引。那么规划器将选择这样的计划：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=9.29..13.30 rows=1 width=24) (actual
 time=0.148..0.148 rows=0 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
   ->  BitmapAnd  (cost=9.29..9.29 rows=1 width=0) (actual time=0.145..0.145
 rows=0 loops=1)
         ->  Bitmap Index Scan on tbloom_i5_idx  (cost=0.00..4.52 rows=11
 width=0) (actual time=0.089..0.089 rows=10 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on tbloom_i2_idx  (cost=0.00..4.52 rows=11
 width=0) (actual time=0.048..0.048 rows=8 loops=1)
               Index Cond: (i2 = 898732)
 Planning time: 2.049 ms
 Execution time: 0.280 ms
(9 rows)
```

尽管这个查询运行起来比在其中任一单个索引上都快，但是我们在索引尺寸上付出了很大的代价。每一个单列 btree 索引占用 214 MB，因此总的空间会超过 1.2GB，这是布鲁姆索引所使用的空间的 8 倍。

F.5.3. 操作符类接口

用于布鲁姆索引的操作符类只要一个用于被索引数据类型的哈希函数以及一个用于搜索的等值操作符。这个例子展示了用于text数据类型的操作符类定义：

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.5.4. 限制

- 在模块中只包括了用于int4以及text的操作符类。
- 搜索只支持=操作符。但是未来可以为带合并和交集操作的数组增加支持。
- bloom访问方法不支持UNIQUE索引。
- bloom访问方法不支持对NULL值的搜索。

F.5.5. 作者

Teodor Sigaev <teodor@postgrespro.ru>, Postgres Professional, Moscow, Russia

Alexander Korotkov <a.korotkov@postgrespro.ru>, Postgres Professional, Moscow, Russia

Oleg Bartunov <obartunov@postgrespro.ru>, Postgres Professional, Moscow, Russia

F.6. btree_gin

btree_gin提供了一个为数据类型 int2、int4、int8、float4、 float8、timestamp with time zone、 timestamp without time zone、time with time zone、 time without time zone、date、interval、 oid、money、"char"、 varchar、text、bytea、bit、varbit、macaddr、macaddr8、inet、 cidr、uuid、name、bool、 bpchar和所有enum类型实现B树等价行为的GIN操作符类例子。

通常，这些操作符类不会比等效的标准B树索引方法更好，并且它们缺少标准B树代码的一个主要特性：强制一致性的能力。但是，它们有助于GIN测试并且有助于作为开发其他GIN操作符类的基础。另外，对于测试一个GIN可索引的列和一个B树可索引的列的查询，创建一个使用这些操作符类之一的多列GIN索引要比创建必须通过位图AND组合在一起的两个独立索引要更有效。

F.6.1. 用法示例

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIN (a);
-- query
SELECT * FROM test WHERE a < 10;
```

F.6.2. 作者

Teodor Sigaev (<teodor@stack.net>) 和 Oleg Bartunov (<oleg@sai.msu.su>)。更多信息请见http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin。

F.7. btree_gist

btree_gist为数据类型 int2、int4、int8、float4、 float8、numeric、timestamp with time zone、 timestamp without time zone、time with time zone、 time without time zone、date、interval、 oid、money、char、 varchar、text、bytea、bit、varbit、macaddr、macaddr8、 inet、cidr、uuid和所有enum 类型提供了实现 B 树等效行为的 GiST 索引操作符类。

通常，这些操作符类不会比等效的标准 B 树索引方法更好，并且它们缺少标准 B 树代码的一个主要特性：强制一致性的能力。但是，如下文所述，它们提供了在一个 B 树索引中没有的一些其他特性。另外，当需要一个多列 GiST 索引，并且其某些列的数据类型只在 GiST 中是可索引的而其他列是简单数据类型时，这些操作符类就有用了。最后，这些操作符可以用于 GiST 测试以及作为开发其他 GiST 操作符类的基础。

除了典型的 B 树搜索操作符之外，btree_gist也为<>（“不等于”）提供了索引支持。这可能与下文描述的排他约束组合在一起产生作用。

另外，对于那些具有自然距离度量的数据类型，btree_gist定义了一个距离操作符<->，并且为使用这个操作符的最近邻搜索提供了 GiST 索引支持。距离操作符还提供给了：int2、int4、 int8、float4、 float8、timestamp with time zone、 timestamp without time zone、 time without time zone、date、interval、 oid和money。

F.7.1. 用法示例

使用btree_gist代替btree的简单例子：

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIST (a);
-- query
SELECT * FROM test WHERE a < 10;
-- nearest-neighbor search: find the ten entries closest to "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

使用一个排他约束来强制规则：一个动物园里的一个笼子只能装一种动物：

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint
 "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage,
 animal)=(123, zebra).
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.7.2. 作者

Teodor Sigaev (<teodor@stack.net>)、Oleg Bartunov (<oleg@sai.msu.su>)、Janko Richter (<jankorichter@yahoo.de>)和Paul Jungwirth (<pj@illuminatedcomputing.com>)。参阅 http://www.sai.msu.su/~megera/postgres/gist/。

F.8. citext

citext模块提供了一种大小写不敏感的字符串类型：citext。特别地，它在比较值时内部调用的是lower。除此之外，它的行为几乎与text完全相同。

F.8.1. 基本原理

在PostgreSQL中做大小写不敏感匹配的标准方法曾经是在比较值时使用lower函数，例如：

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

这工作得比较好，但是有一些缺点：

- 它让你的 SQL 语句冗长，并且你必须总是要记住在列和查询值上使用lower。
- 它不会使用一个索引，除非你使用lower创建一个函数索引。
- 如果你声明一个列为UNIQUE或PRIMARY KEY，隐式生成的索引是大小写敏感的。因此，它对于大小写不敏感的搜索是没有用处的，并且它不会强制大小写不敏感的唯一性。

citext数据类型允许你在 SQL 查询中消除对lower的调用，并且允许一个主键是大小写无关的。就和text一样，citext是区域相关的，这意味着大写和小写字符的匹配依赖于数据库LC_CTYPE设置的规则。此外，这种行为和在查询中使用lower是一样的。但是因为它是由数据类型以透明的方式完成的，你不需要记住在你的查询中做任何特殊的事情。

F.8.2. 如何使用它

这里是一个简单的用法示例：

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Tom',    sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Damian', sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'NEAL',   sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Bjørn',  sha256(random()::text::bytea) );

SELECT * FROM users WHERE nick = 'Larry';
```

即使nick列被设置为larry而查询是Larry，SELECT语句也将只返回一个元组。

F.8.3. 串比较行为

citext执行比较时先将每一个串转换成小写形式（调用lower）然后正常地比较结果。因此，如果两个串通过lower产生相同的结果，它们就被认为是相等。

为了尽可能接近地模拟一种大小写不敏感的排序规则，一些串处理操作符和函数有citext相关的版本。例如，当应用到citext时，正则表达式操作符~和~*会展现出相同的行为：它们都以大小写不敏感的方式匹配。!~和!~*也是一样，以及LIKE操作符~~和~~*，以及!~~和!~~*。如果你想以大小写敏感的方式匹配，你可以把该操作符的参数造型成text。

相似地，如果下列函数的参数是citext，它们会以大小写不敏感的方式执行匹配：

- regexp_match()
- regexp_matches()
- regexp_replace()
- regexp_split_to_array()
- regexp_split_to_table()
- replace()
- split_part()
- strpos()
- translate()

对于 regexp 函数，如果你想要以大小写敏感的方式匹配，你可以指定“c”标志来强制大小写敏感的匹配。否则，如果你想要大小写敏感的行为，你必须在使用这些函数之一之前造型到text。

F.8.4. 限制

- citext的大小写折叠行为取决于你的数据库的LC_CTYPE设置。因此它如何比较值是在数据库被创建时决定的。在 Unicode 标准定义的术语中没有真正的大小写不敏感。实际上，它的含义是，只要你对你的排序规则满意，你就应该对citext的比较满意。但是如果在你的数据库中存储有不同语言的数据，当排序规则是用于一种语言时，另一种语言的用户可能会发现他们的查询结果并不是所期待的。

- 自PostgreSQL 9.1 其，你可以为citext列或数据值附加一个COLLATE说明。当前，在比较大小写折叠过的字符串时，citext操作符将尊重一种非默认的COLLATE说明，但是最初到小写形式的折叠是根据数据库的 LC_CTYPE设置完成的（就是说，尽管给出了COLLATE "default"）。这可能在未来的发行中被改变，这样两步都能遵循输入的COLLATE说明。

- citext的效率不如text，因为操作符函数和 B 树比较函数必须创建数据的拷贝并且将它转换为小写形式来进行比较。不过，它比使用lower进行大小写不敏感的匹配的效率要略高。

- 如果你在某些环境下需要以大小写敏感的方式比较数据并且在另一些环境下需要以大小写不敏感的方式比较数据，citext就帮不上什么忙。标准的答案是使用text类型并且在你需要以大小写不敏感的方式比较时手工使用lower函数。如果大小写不敏感的比较需求不频繁，这会工作得不错。如果你大部分时间需要大小写不敏感的行为，考虑将数据存储为citext并且在进行大小写敏感比较时显式地将列造型为text。不管在那种情况下，你都需要两个索引来让两种类型的搜索更快。

- 包含citext操作符的模式必须在当前的search_path（通常是public）中。如果它不在搜索路径中，普通的大小写敏感的text操作符将会取而代之。

F.8.5. 作者

David E. Wheeler <david@kineticode.com>

受 Donald Fraser 的citext模块启发。

F.9. cube

这个模块实现了一种数据类型cube来表示多维立方体。

F.9.1. 语法

表 F.2展示了cube类型有效的外部表示。x、y等表示浮点数。

表 F.2. 立方体外部表示

外部语法	含义
x	一个一维点（或者长度为零的一维区间）
(x)	同上
x1,x2,...,xn	n-维空间中的一个点，内部表示为一个零容积立方体
(x1,x2,...,xn)	同上
(x),(y)	开始于x并且结束于y的一个一维区间，反之亦然。顺序并不重要
[(x),(y)]	同上
(x1,...,xn),(y1,...,yn)	一个 n-维立方体，用它的对角顶点对表示
[(x1,...,xn),(y1,...,yn)]	同上

一个立方体的对角录入的顺序无关紧要。如果需要创建一种统一的“左下 — 右上”的内部表示，cube函数会自动地交换值。当角重合时，cube只存储一个角和一个“is point”标志，这样避免浪费空间。

输入中的空白空间会被忽略，因此[(x),(y)]与[(x), (y)]相同。

F.9.2. 精度

值在内部被存储为 64 位浮点数。这意味着超过 16 位有效位的数字将被截断。

F.9.3. 用法

表 F.3展示了为类型cube提供的操作符。

表 F.3. 立方体操作符

操作符	结果	描述
a = b	boolean	立方体 a 和 b 相同。
a && b	boolean	立方体 a 和 b 重叠。
a @> b	boolean	立方体 a 包含 立方体 b。
a <@ b	boolean	立方体 a 被包含在立方体 b 中。
a < b	boolean	立方体 a 小于立方体 b。
a <= b	boolean	立方体 a 小于或者等于立方体 b。
a > b	boolean	立方体 a 大于立方体 b。
a >= b	boolean	立方体 a 大于或者等于立方体 b。
a <> b	boolean	立方体 a 不等于立方体 b。
a -> n	float8	得到立方体的第n个坐标（从 1 开始数）。
a ~> n	float8	以下列方式获取多维数据集的第n个坐标：n = 2 * k - 1表示第k维度的下限，n = 2 * k表示第k维度的上限。负的n表示相应正坐标的倒数值。此运算符专为KNN-GiST支持而设计。
a <-> b	float8	a 和 b 之间的欧氏距离。
a <#> b	float8	a 和 b 之间的直线距离（taxicab 距离，L1 度量）。
a <=> b	float8	a 和 b 之间的切比雪夫（L-inf 度量）距离。

（在 PostgreSQL 8.2 之前，包含操作符@>和<@分别被称为@和~。这些名称仍然可用，但是已经被废弃并且最终将会退休。注意旧的名字与之前核心几何数据类型遵循的习惯相反！）

标量排序操作符（<、>=等）除了用来排序之外没有什么实际用途。这些操作符首先比较第一个坐标，如果它们相等再比较第二个坐标等等。它们主要为支持cube的 b-树索引操作符类而存在，这类操作符对支持cube列上的 UNIQUE 约束等很有用。

cube模块也为cube值提供了一个 GiST 索引操作符类。cube GiST 索引可以被用于在WHERE子句中通过=、&&、@>以及<@操作符来搜索值。

此外，cube GiST 索引可以被用在ORDER BY子句中通过度量操作符<->、<#>和<=>来查找最近邻。例如， 3-D 点(0.5, 0.5, 0.5)的最近邻可以用下面的查询很快地找到：

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

也可以用这种方式使用~>操作符来高效地检索通过选定坐标排序后的前几个值。例如，可以用下面的查询得到通过第一个坐标（左下角）升序排列后的前几个立方体：

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

以及得到通过右上角第一个坐标降序排列后的 2-D 立方体：

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

表 F.展示了可用的函数。

表 F.4. 立方体函数

函数	结果	描述	例子
cube(float8)	cube	制造一个一维立方体，坐标都是相同的。	cube(1) == '(1)'
cube(float8, float8)	cube	制造一个一维立方体。	cube(1,2) == '(1), (2)'
cube(float8[])	cube	使用数组定义的坐标制造一个零容积的立方体。	cube(ARRAY[1,2]) == '(1,2)'
cube(float8[], float8[])	cube	用由两个数组定义的右上和左下坐标制造一个立方体，两个数组必须等长。	cube(ARRAY[1,2], ARRAY[3,4]) == '(1,2),(3,4)'
cube(cube, float8)	cube	在一个现有的立方体上增加一维来制造一个新立方体，对新坐标的各个端点都采用相同的值。这可以用于从计算得到的值逐渐地构建立方体。	cube('(1,2),(3,4)'::cube, 5) == '(1,2,5),(3,4,5)'
cube(cube, float8, float8)	cube	在一个现有的立方体上增加一维来制造一个新立方体。这可以用于从计算得到的值逐渐地构建立方体。	cube('(1,2),(3,4)'::cube, 5, 6) == '(1,2,5),(3,4,6)'
cube_dim(cube)	integer	返回该立方体的维数	cube_dim('(1,2),(3,4)') == '2'
cube_ll_coord(cube, integer)	float8	返回一个立方体的左下角的第 n个坐标值	cube_ll_coord('(1,2),(3,4)', 2) == '2'
cube_ur_coord(cube, integer)	float8	返回一个立方体的右上角的第n个坐标值	cube_ur_coord('(1,2),(3,4)', 2) == '4'

函数	结果	描述	例子
cube_is_point(cube)	boolean	如果一个立方体是一个点则返回真，也就是两个定义点相同。	
cube_distance(cube, cube)	float8	返回两个立方体之间的距离。如果两个都是点，这就是普通距离函数。	
cube_subset(cube, integer[])	cube	从一个现有的立方体制造一个新立方体，使用来自于一个数组的维索引列表。它可以被用来抽取一个单一维度的端点，或者它可以被用来去除维度，或者按照需要对它们重新排序。	cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) == '(3),(7)' cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) == '(5,3,1,1),(8,7,6,6)'
cube_union(cube, cube)	cube	产生两个立方体的并	
cube_inter(cube, cube)	cube	产生两个立方体的交	
cube_enlarge(c cube, r double, n integer)	cube	用一个指定的半径r在至少n个维度上增加立方体的尺寸。如果该半径是负值，则该立方体会收缩。这有助于围绕一个点创建一个外包盒来搜索附近点。所有已定义的维度都会按照半径r被改变。左下坐标按照r被减小并且右上坐标按照r被增加。如果一个左下坐标被增加得超过对应的右上坐标（这只会发生在r< 0时），则两个坐标会被设置为它们的均值。如果n大于已定义的维度数并且该立方体被增加（r >= 0），则额外的维度会被加入以让维度数达到n，对于额外的坐标将使用 0 作为初始值。这个函数可用来创建围绕一个点的外包盒以搜索临近点。	cube_enlarge('(1,2),(3,4)', 0.5, 3) == '(0.5,1.5,-0.5),(3.5,4.5,0.5)'

F.9.4. 默认值

我相信这个并：

```
select cube_union('(0,5,2),(2,3,1)', '0');
```

```
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

不会与常识矛盾，下面的交也不会

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

在所有不同维度立方体的二元操作中，我假定低纬度的那一个要做笛卡尔投影，即为字符串表示中被省略的坐标取零。上面的例子等同于：

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

下列包含谓词使用点语法，不过实际上第二个参数在内部被表示为一个盒子。这种语法让我们不必定义一种单独的点类型以及用于（盒子, 点）谓词的函数。

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.9.5. 注解

用法的例子可见回归测试sql/cube.sql。

为了不容易出问题，对于立方体的维度数有 100 的限制。如果你想要更大的立方体，可以在cubedata.h中修改。

F.9.6. 工作人员

原作者：Gene Selkov, Jr. <selkovjr@mcs.anl.gov>，数学与计算机科学部，阿尔贡国家实验室。

我的感谢主要要献给 Joe Hellerstein 教授（http://db.cs.berkeley.edu/jmh/），他阐明了 GiST （http://gist.cs.berkeley.edu/），还要送给他以前的学生 Andy Dong，他为 Illustra 编写了例子。我也对所有的 Postgres 开发者（现在的和以前的）心存感激，他们让我能够创造自己的世界并且宁静地生活在其中。我也要感谢阿尔贡实验室和美国能源局对我多年数据库研究的支持。

这个包的小更新由 Bruno Wolff III <bruno@wolff.to>于 2002 年 8/9 月完成。这些修改包括将精度从单精度改为双精度以及增加了一些新的函数。

额外的更新由 Joshua Reich <josh@root.net> 在 2006 年 7 月做出。其中包括cube(float8[], float8[])并且将代码从废弃的 V0 协议改到 V1 调用协议。

F.10. dblink

dblink是一个支持在一个数据库会话中连接到其他PostgreSQL数据库的模块。

还可以看看postgres_fdw，它以一种更现代和更加兼容标准的架构提供了相同的功能。

dblink_connect

dblink_connect — 打开一个到远程数据库的持久连接

大纲

```
dblink_connect(text connstr) 返回 text
dblink_connect(text connname, text connstr) 返回 text
```

描述

dblink_connect()建立一个到远程PostgreSQL数据库的连接。要联系的服务器和数据库通过一个标准的libpq连接串来标识。可以选择将一个名字赋予给该连接。多个命名的连接可以被一次打开，但是一次只允许一个未命名连接。连接将会持续直到被关闭或者数据库会话结束。

连接串也可以是一个现存外部服务器的名字。在使用外部服务器时，我们推荐使用外部数据包装器dblink_fdw。见下面的例子，以及CREATE SERVER和CREATE USER MAPPING。

参数

connname

要用于这个连接的名字。如果被忽略，将打开一个未命名连接并且替换掉任何现有的未命名连接。

connstr

libpq-风格的连接信息串，例如 hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd。详见第 34.1.1 节此外，还可以是一个外部服务器的名字。

返回值

返回状态，它总是OK（因为任何错误会导致该函数抛出一个错误而不是返回）。

注解

如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，应该在开始每个会话时从search_path中移除公共可写的方案。例如，可以把options=-csearch_path=增加到connstr。这种考虑不是特别针对dblink，它适用于每一种执行任意SQL命令的接口。

只有超级用户能够使用dblink_connect来创建无口令认证连接。如果非超级用户需要这种能力，使用dblink_connect_u。

选择包含等号的连接名是不明智的，因为这会产生与在其他dblink函数中的连接信息串混淆的风险。

例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
```

```
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

-- FOREIGN DATA WRAPPER functionality
-- Note: local connection must require password authentication for this to work
 properly
--       Otherwise, you will receive the following error from dblink_connect():
--       ------------------------------------------------------------------------
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a
 password.
--       HINT:  Target server's authentication method must be changed.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr
 '127.0.0.1', dbname 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user
 'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn','SELECT * FROM foo') AS t(a int, b text, c
 text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — 不安全地打开一个到远程数据库的持久连接

大纲

```
dblink_connect_u(text connstr) 返回 text
dblink_connect_u(text connname, text connstr) 返回 text
```

描述

dblink_connect_u()和dblink_connect()一样，不过它将允许非超级用户使用任意认证方式来连接。

如果远程服务器选择了一种不涉及口令的认证方式，那么可能发生模仿以及后续的扩大权限，因为该会话看起来像由运行PostgreSQL的用户发起的。此外，即使远程服务器不要求一个口令，也可能从服务器环境提供该口令，例如一个属于服务器用户的~/.pgpass文件。这带来的不只是模仿的风险，而且还有将口令暴露给不可信的远程服务器的风险。因此，dblink_connect_u()最初是用所有从PUBLIC撤销的特权安装的，这让它只能被超级用户调用。在某些情况中，为dblink_connect_u()授予EXECUTE权限给可信的指定用户是合适的，但是必须小心。我们也推荐任何属于服务器用户的~/.pgpass文件不能包含任何指定了一个通配符主机名的记录。

详见dblink_connect()。

dblink_disconnect

dblink_disconnect — 关闭一个到远程数据库的持久连接

大纲

```
dblink_disconnect() 返回 text
dblink_disconnect(text connname) 返回 text
```

描述

dblink_disconnect()关闭一个之前被dblink_connect()打开的连接。不带参数的形式关闭一个未命名连接。

参数

connname

要被关闭的命名连接的名字。

返回值

它总是OK（因为任何错误会导致该函数抛出一个错误而不是返回）。

例子

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

dblink — 在一个远程数据库中执行一个查询

大纲

```
dblink(text connname, text sql [, bool fail_on_error]) 返回记录集
dblink(text connstr, text sql [, bool fail_on_error]) 返回记录集
dblink(text sql [, bool fail_on_error]) 返回记录集
```

描述

dblink在一个远程数据库中执行一个查询（通常是一个SELECT，但是也可以是任意返回行的SQL 语句）。

当给定两个text参数时，第一个被首先作为一个持久连接的名称进行查找；如果找到，该命令会在该连接上被执行。如果没有找到，第一个参数被视作一个用于dblink_connect的连接信息字符串，并且被指出的连接只是在这个命令的持续期间被建立。

参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

connstr

如之前为dblink_connect所描述的一个连接信息字符串。

sql

你希望在远程数据库中执行的 SQL 查询，例如select * from foo。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不反回行。

返回值

该函数返回查询产生的行。因为dblink能与任何查询一起使用，它被声明为返回record，而不是指定任意特定的列集合。这意味着你必须指定在调用的查询中所期待的列集合 — 否则PostgreSQL将不知道会得到什么。这里是一个例子：

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

FROM子句的“alias”部分必须指定函数将返回的列名及类型（在一个别名中指定列名实际上是标准 SQL 语法，但是指定列类型是一种PostgreSQL扩展）。这允许系统在尝试执行该函数之前就理解*将展开成什么，以及WHERE子句中的proname指的什么。在运行时，如果来自远程数据库的实际查询结果和FROM子句中显示的列数不同，将会抛出一个错误。不过，列名不需要匹配，并且dblink并不坚持精确地匹配类型。只要被返回的数据字符串是FROM子句中声明的列类型的合法输入，它就将会成功。

注解

一种将预定义查询用于dblink的方便方法是创建一个视图。这允许列类型信息被埋藏在该视图中，而不是在每一个查询中都拼写出来。例如：

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

例子

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
```

```
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — 在一个远程数据库中执行一个命令

大纲

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

描述

dblink_exec在一个远程数据库中执行一个命令（也就是，任何不返回行的 SQL 语句）。

当给定两个text参数时，第一个被首先作为一个持久连接的名称进行查找；如果找到，该命令会在该连接上被执行。如果没有找到，第一个参数被视作一个用于dblink_connect的连接信息字符串，并且被指出的连接只是在这个命令的持续期间被建立。

参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

connstr

如之前为dblink_connect所描述的一个连接信息字符串。

sql

你希望在远程数据库中执行的 SQL 命令，例如insert into foo values(0,'a','{"a0","b0","c0"}')。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

返回值

返回状态，可能是命令的状态字符串或ERROR。

例子

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
```

```
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo
 values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null
 constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — 在一个远程数据库中打开一个游标

大纲

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) 返回 text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) 返回 text
```

描述

dblink_open()在一个远程数据库中打开一个游标。该游标能够随后使用dblink_fetch()和dblink_close()进行操纵。

参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要赋予给这个游标的名称。

sql

你希望在远程数据库中执行的SELECT语句，例如select * from pg_class。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

返回值

返回状态，OK或者ERROR。

注解

因为一个游标只能在一个事务中持续，如果远端还没有在一个事务中，dblink_open会在远端开始一个显式事务块（BEGIN）。当匹配的dblink_close被执行时，这个事务将再次被关闭。注意如果你使用dblink_exec在dblink_open和dblink_close之间改变数据，并且接着发生了一个错误或者你在dblink_close之前使用了dblink_disconnect，你的更改将被丢失，因为事务将被中止。

例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)
```

```
SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

dblink_fetch — 从一个远程数据库中的打开的游标返回行

大纲

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) 返回 record 集合
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) 返回 record 集合
```

描述

dblink_fetch从一个之前由dblink_open建立的游标中取得行。

参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要从中取数据的游标名。

howmany

要检索的最大行数。从当前游标位置向前的接下来howmany个行会被取出。一旦该游标已经到达了它的末端，将不会产生更多行。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不反回行。

返回值

该函数返回从游标中取出的行。要使用这个函数，你将需要指定想要的列集合，如前面dblink中所讨论的。

注解

当FROM子句中指定的返回列的数量和远程游标返回的实际列数不匹配时，将抛出一个错误。在这个事件中，远程游标仍会被前进错误没发生时应该前进的行数。对于远程FETCH完成之后在本地查询中发生的任何其他错误，情况也是一样。

例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)
```

```
SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname
 like ''bytea%''');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — 关闭一个远程数据库中的游标

大纲

```
dblink_close(text cursorname [, bool fail_on_error]) 返回 text
dblink_close(text connname, text cursorname [, bool fail_on_error]) 返回 text
```

描述

dblink_close关闭一个之前由dblink_open打开的游标。

参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要关闭的游标名。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

返回值

返回状态，OK或者ERROR。

注解

如果dblink_open开始了一个显式事务块，并且这是这个连接中最后一个保持打开的游标，dblink_close将发出匹配的COMMIT。

例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
```

```
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — 返回所有打开的命名 dblink 连接的名称

大纲

dblink_get_connections() 返回 text[]

描述

dblink_get_connections返回一个数组，其中是所有打开的命名dblink连接的名称。

返回值

返回一个连接名称的文本数组，如果没有则为 NULL。

例子

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — 得到在命名连接上的最后一个错误消息

大纲

```
dblink_error_message(text connname) 返回 text
```

描述

dblink_error_message为一个给定连接取得最近的远程错误消息。

参数

connname

要使用的连接名。

返回值

返回最后一个错误消息，如果在这个连接上没有错误则返回一个空字符串。

例子

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — 发送一个异步查询到远程数据库

大纲

```
dblink_send_query(text connname, text sql) 返回 int
```

描述

dblink_send_query发送一个要被异步执行的查询，也就是不需要立即等待结果。在该连接上不能有还在处理中的异步查询。

在成功地派送一个异步查询后，可以用dblink_is_busy检查完成状态，并且结果最终由dblink_get_result收集。也可以使用dblink_cancel_query尝试取消一个活动中的异步查询。

参数

connname

要使用的连接名。

sql

你希望在远程数据库中执行的 SQL 语句，例如select * from pg_class。

返回值

如果查询被成功地派送返回 1，否则返回 0。

例子

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

dblink_is_busy — 检查连接是否正在忙于一个异步查询

大纲

```
dblink_is_busy(text connname) 返回 int
```

描述

dblink_is_busy测试是否一个异步查询正在进行中。

参数

connname

要检查的连接名。

返回值

如果连接正忙则返回 1，如果不忙则返回 0。如果这个函数返回 0，dblink_get_result将被保证不会阻塞。

例子

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — 在一个连接上检索异步通知

大纲

```
dblink_get_notify() 返回 (notify_name text, be_pid int, extra text) 集合
dblink_get_notify(text connname) 返回 (notify_name text, be_pid int, extra text) 集合
```

描述

dblink_get_notify在一个未命名连接或者一个指定的命名连接上检索通知。要通过 dblink 接收通知，首先必须使用dblink_exec发出LISTEN。详见LISTEN和NOTIFY。

参数

connname

要在其上得到通知的命名连接的名称。

返回值

返回 (notify_name text, be_pid int, extra text) 集合，或者一个空集。

例子

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — 得到一个异步查询结果

大纲

dblink_get_result(text connname [, bool fail_on_error]) 返回 record 集合

描述

dblink_get_result收集之前dblink_send_query发送的一个异步查询的结果。如果该查询还没有完成，dblink_get_result将等待直到它完成。

参数

connname

要使用的连接名。

fail_on_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不返回行。

返回值

对于一个异步查询（也就是一个返回行的 SQL 语句），该函数返回查询产生的行。要使用这个函数，你将需要指定所期待的列集合，如前面为dblink所讨论的那样。

对于一个异步命令（也就是一个不返回行的 SQL 语句），该函数返回一个只有单个文本列的单行，其中包含了该命令的状态字符串。仍必须在调用的FROM子句中指定结果将具有一个单一文本行。

注解

如果dblink_send_query返回　1，这个函数就必须被调用。对每一个已发送的查询都必须调用一次这个函数，并且在连接再次可用之前还要多调用一次来得到一个空结果集。

当使用dblink_send_query和dblink_get_result时，在将结果集中的任何一行返回给本地查询处理器之前，dblink将取得整个远程查询结果。如果该查询返回大量的行，这可能会导致本地会话中短暂的内存膨胀。最好将这样的一个查询用dblink_open打开成一个游标并且接着每次取得数量可管理的行。也可以使用简单的dblink()，它会避免缓冲大型结果集到磁盘上导致的内存膨胀。

例子

```
contrib_regression=# SELECT dblink_connect('dtest1',
 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
```

```
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 <
 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 <
 3; select * from foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
 text, f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — 在命名连接上取消任何活动查询

大纲

```
dblink_cancel_query(text connname) 返回 text
```

描述

dblink_cancel_query尝试命名连接上正在进行的任何查询。注意这不一定会成功（例如，远程查询可能已经结束）。一个取消请求仅仅提高了该查询将很快失败的几率。你仍必须完成通常的查询协议，例如通过调用dblink_get_result。

参数

connname

要使用的连接名。

返回值

如果取消请求已经被发送，则返回OK；如果失败，则返回一个错误消息的文本。

例子

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — 返回一个关系的主键域的位置和域名称

大纲

dblink_get_pkey(text relname) 返回 dblink_pkey_results 集合

描述

dblink_get_pkey提供有关于本地数据库中一个关系的主键的信息。这有时候有助于生成要被发送到远程数据库的查询。

参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

返回值

为每一个主键域返回一行，如果该关系没有主键则不返回行。结果行类型被定义为：

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

position列值可以从 1 到 N，它是该域在主键中的编号，而不是在表列中的编号。

例子

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — 使用一个本地元组构建一个 INSERT 语句，将主键域值替换为提供的值

大纲

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) 返回 text
```

描述

dblink_build_sql_insert在选择性地将一个本地表复制到一个远程数据库时很有用。它基于主键从本地表选择一行，并且接着构建一个复制该行的INSERT命令，但是其中主键值被替换为最后一个参数中的值（要创建该行的一个准确拷贝，只要为最后两个参数指定相同的值）。

参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

primary_key_attnums

主键域的属性号（从 1 开始），例如1 2。

num_primary_key_atts

主键域的数量。

src_pk_att_vals_array

要被用来查找本地元组的主键域值。每一个域都被表示为文本形式。如果没有行具有这些主键值，则抛出一个错误。

tgt_pk_att_vals_array

要被替换到结果INSERT命令中的主键域值。每一个域被表示为文本形式。

返回值

将要求的 SQL 语句返回为文本。

注解

自PostgreSQL 9.0 开始，primary_key_attnums中的属性号被解释为逻辑列号，对应于列在SELECT * FROM relname中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。

例子

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
             dblink_build_sql_insert
-----------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — 使用所提供的主键域值构建一个 DELETE 语句

大纲

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) 返回 text
```

描述

dblink_build_sql_delete在选择性地将一个本地表复制到一个远程数据库时很有用。它构建一个 SQL DELETE命令用来删除具有给定主键值的行。

参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

primary_key_attnums

主键域的属性号（从 1 开始），例如1 2。

num_primary_key_atts

主键域的数量。

tgt_pk_att_vals_array

要用在结果DELETE命令中的主键域值。每一个域都被表示为文本形式。

返回值

将要求的 SQL 语句返回为文本。

注解

自PostgreSQL 9.0 开始，primary_key_attnums中的属性号被解释为逻辑列号，对应于列在SELECT * FROM relname中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。

例子

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — 使用一个本地元组构建一个 UPDATE 语句，将主键域值替换为提供的值

大纲

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) 返回 text
```

描述

dblink_build_sql_update在选择性地将一个本地表复制到一个远程数据库时很有用。它从本地表基于主键选择一行，并且接着构建一个 SQL UPDATE命令来复制该行，但是其中的主键值被替换为最后一个参数中的值（要创建该行的一个准确拷贝，只要为最后两个参数指定相同的值）。UPDATE命令总是为该行的所有域赋值 — 这个函数与dblink_build_sql_insert之间的主要区别是它假定目标行已经存在于远程表中。

参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

primary_key_attnums

主键域的属性号（从 1 开始），例如1 2。

num_primary_key_atts

主键域的数量。

src_pk_att_vals_array

要被用来查找本地元组的主键域值。每一个域都被表示为文本形式。如果没有行具有这些主键值，则抛出一个错误。

tgt_pk_att_vals_array

要用在结果UPDATE命令中的主键域值。每一个域都被表示为文本形式。

返回值

将要求的 SQL 语句返回为文本。

注解

自PostgreSQL 9.0 开始，primary_key_attnums中的属性号被解释为逻辑列号，对应于列在SELECT * FROM relname中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。

例子

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                  dblink_build_sql_update
------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.11. dict_int

dict_int是一个附加全文搜索词典模板的例子。这个例子词典的动机是控制整数（有符号和无符号）的索引，允许在阻止唯一词数量的过度增长（会严重影响搜索性能）时也能索引这些数字。

F.11.1. 配置

该词典接受两个选项：

- maxlen参数指定在一个整数词中允许的最大位数。默认值为 6。
- rejectlong参数指定一个超长整数是否应该被截断或忽略。如果rejectlong为false（默认），该词典返回该整数的第一个数字。如果rejectlong为true，该词典将一个超长整数作为一个停用词对待，因此它将不会被索引。注意这也意味着这样一个整数不能被搜索。

F.11.2. 用法

安装dict_int扩展会使用默认参数创建一个文本搜索模板intdict_template和一个基于它的词典intdict。你可以修改参数，例如

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

或者创建基于该模板的新词典。

要测试该词典，可以尝试

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

但是现实世界的用法将涉及将它包括在一个第 12 章描述的文本搜索配置中。看起来像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.12. dict_xsyn

dict_xsyn（扩展同义词字典）是一个附加全文搜索字典模板的例子。这种字典类型将词替换为它们的同义词分组，并且让使用其任一同义词进行搜索变得可能。

F.12.1. 配置

一个dict_xsyn词典接受以下选项：

- matchorig控制该词典是否接受原生词。默认为true。
- matchsynonyms控制该词典是否接受同义词。默认为false。
- keeporig控制原生词是否被包括在词典的输出中。默认为true。
- keepsynonyms控制同义词是否被包括在词典的输出中。默认为true。
- rules是包含同义词列表的文件的基本名。这个文件必须被存储在$SHAREDIR/tsearch_data/（其中$SHAREDIR表示PostgreSQL安装的共享数据目录）中。它的名称必须以.rules结束（这不包括在rules参数中）。

规则文件具有下面的格式：

- 每一行表示一个单一词的同义词分组，它在该行中首先被给出。同义词被空白分隔，这样：

```
word syn1 syn2 syn3
```

- 井号（#）是注释定界符。它可以出现在一行中的任何位置。该行的剩余部分将被跳过。

例如，可以看看安装在$SHAREDIR/tsearch_data/中的xsyn_sample.rules。

F.12.2. 用法

安装dict_xsyn扩展会用默认参数创建一个文本搜索模板xsyn_template以及一个基于它的词典xsyn。你可以修改参数，例如

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

或者基于该模板创建新的词典。

要测试该词典，你可以尝试

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false,
 MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
```

```
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true,
 MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

现实世界的用法将涉及将它包括在一个第 12 章描述的文本搜索配置中。看起来像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.13. earthdistance

earthdistance模块提供两种不同的方法来计算地球表面的大圆距离。第一种要介绍的依赖于cube模块（必须在earthdistance之前安装）。第二种基于内建的point数据类型，为座标使用精度和纬度。

在这个模块中，地球被假定为完美的球型（如果这对你不够精确，你可能希望去看看PostGIS[1]项目）。

F.13.1. 基于立方体的地球距离

数据被存储在立方体中，立方体的点（所有的角都一样）使用 3 个座标表示到地球中心的 x、y 和 z 距离。提供了一个cube之上的域earth，这包括检查值符合这些限制并且合理地接近于地球的真实表面的约束。

地球的半径获得自earth()函数。其单位是米。但是通过改变这一个函数你能够把该模块改为使用某些其他单位，或者使用一种你认为更合适的不同半径值。

这个包也有在天文数据库中的应用。天文学家可能想要改变earth()来返回一个180/pi()的半径，这样距离就会是度数。

函数也被提供来支持经纬度输入（以度数）、经纬度输出、计算两点间的大圆距离以及容易地指定一个可用于索引搜索的边界框。

所提供的函数在表 F.5中描述。

表 F.5. 基于立方体的地球距离函数

函数	返回	描述
earth()	float8	返回地球的假定半径。
sec_to_gc(float8)	float8	将地球表面两点间的普通直线（切线）距离转换为它们之间的大圆距离。
gc_to_sec(float8)	float8	将地球表面两点间的大圆距离转换为它们之间的普通直线（切线）距离。

[1] http://postgis.net/

函数	返回	描述
ll_to_earth(float8, float8)	earth	给定一个地球表面点的维度（参数 1）和精度（参数 2）度数，返回它的位置。
latitude(earth)	float8	返回一个地球表面点的以度数表示的维度。
longitude(earth)	float8	返回一个地球表面点的以度数表示的经度。
earth_distance(earth, earth)	float8	返回地球表面上两点间的大圆距离。
earth_box(earth, float8)	cube	为一个位置的给定大圆距离范围内的点使用立方体@>操作符返回一个适合于索引搜索的框。这个框中的某些点到该位置的大圆距离会超过指定的大圆距离，因此使用earth_distance的第二次检查应该被包括在查询中。

F.13.2. 基于点的地球距离

这个模块的第二部分依赖于将地球位置表示为类型point的值，其中第一部分被用来表示经度数，第二部分被用来表示纬度数。点被取做 (longitude, latitude) 并且不能反过来，因为经度更接近直观上的 x 轴，而纬度则接近 y 轴。

如表 F.6所示，这一部分只提供了一个单一操作符。

表 F.6. 基于点的地球距离操作符

操作符	返回	描述
point <@> point	float8	给定地球表面两点之间的法定英里距离。

注意和这个模块的基于cube的部分不同，这里的单位是被硬编码的：改变earth()函数将不会影响这个操作符的结果。

经度/纬度表示的一个缺点是你需要小心靠近两极和靠近经度正负 180 度处的边界情况。基于cube的表示可以避免这些不连续性。

F.14. file_fdw

file_fdw模块提供外部数据包装器file_fdw， 它能被用来访问服务器的文件系统中的数据文件，或者在服务器上执行程序并读取它们的输出。 数据文件或程序输出必须是能够被COPY FROM读取的格式， 详见COPY。当前只能读取数据文件。

用这个包装器创建的一个外部表可以有下列选项：

filename

指定要被读取的文件。必须是一个绝对路径名。 必须指定filename或program， 但不能同时指定两个。

program

指定要执行的命令。该命令的标准输出将被读取， 就像使用COPY FROM PROGRAM一样。必须指定program 或filename，但不能同时指定两个。

format

指定数据的格式，和COPY的FORMAT选项相同。

header

指定数据是否具有一个头部行，和COPY的HEADER选项相同。

delimiter

指定数据的定界符字符，和COPY的DELIMITER选项相同。

quote

指定数据的引用字符，和COPY的QUOTE选项相同。

escape

指定数据的转义字符，和COPY的ESCAPE选项相同。

null

指定数据的空字符串，和COPY的NULL选项相同。

encoding

指定数据的编码，和COPY的ENCODING选项相同。

注意虽然COPY允许诸如HEADER的选项不用一个相应的值指定， 但是外部表选项语法要求在所有情况下都出现一个值。要激活通常写入没有值的 COPY选项，你可以传递值 TRUE，因为所有这些选项都是布尔值。

使用这个包装器创建的表的一列可以具有下列选项：

force_not_null

这是一个布尔选项。如果为真，它指定该列的值不应该与空字符串匹配（也就是表级别的null选项）。这和把该列放在COPY的FORCE_NOT_NULL选项中具有相同的效果。

force_null

这是一个布尔选项。如果为真，它指定匹配空值字符串的列值会被返回为NULL， 即使该值被引号引用。如果没有这个选项，只有匹配空值字符串的未被引用的值会被返回为NULL。这和在COPY的FORCE_NULL 选项中列出该列有同样的效果。

COPY的OIDS和FORCE_QUOTE选项当前不被file_fdw支持。

这些选项只能为一个外部表及其列指定，而不能在file_fdw外部数据包装器的选项中指定，也不能在使用该包装器的服务器或者用户映射的选项中指定。

出于安全原因，改变表级别的选项要求超级用户特权或 具有默认角色pg_read_server_files（使用文件名）或 默认角色pg_execute_server_program （使用程序）的权限： 只有特定用户能够控制读取哪个文件或者运行哪个程序。 原则上普通用户可以被允许改变其它选项，但是当前还不支持这样做。

当指定program选项时，请记住，选项字符串是通过shell执行的。 如果想传递任何参数到来自不受信任的源的命令， 必须小心去掉或转义任何对shell来说可能有特殊含义的字符。安全起见，最好使用固定的命令字符串， 或者至少避免传递任何用户输入。

对于一个使用file_fdw的外部表，EXPLAIN显示要读取的文件名或要运行的程序。对于文件来说，除非指定COSTS OFF，否则文件尺寸（以字节计）也会被显示。

例 F.1. 为 PostgreSQL CSV 日志创建一个外部表

一种file_fdw的用法是把可用的 PostgreSQL 活动日志变成一个表用于查询。要这样做，首先你必须正在将日志记录到一个 CSV 文件，这里我们称其为pglog.csv。首先，将file_fdw安装为一个扩展：

```
CREATE EXTENSION file_fdw;
```

然后创建一个外部服务器：

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

现在你已经准备好创建外部数据表。使用CREATE FOREIGN TABLE命令，你将需要为该表定义列、CSV 文件名以及格式：

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/data/log/pglog.csv', format 'csv' );
```

就是这样了 — 现在你能够直接查询你的日志了。当然，在生产中你会需要定义一些方法来处理日志轮转。

F.15. fuzzystrmatch

fuzzystrmatch模块提供多个函数来判断字符串之间的相似性和距离。

> **小心**
>
> 当前，soundex、metaphone、dmetaphone和dmetaphone_alt函数使用多字节编码（例如 UTF-8）下工作得不好。

F.15.1. Soundex

语音表示法系统是一种将相似发音的名字转换成相同的代码来匹配它们的方法。这最初由美国国家统计局在 1880 年、1900 年和 1910 年使用。注意语音表示法对于非英语名称不是很有用。

fuzzystrmatch模块提供了两个函数用于语音表示法代码：

```
soundex(text) 返回 text
difference(text, text) 返回 int
```

soundex函数将一个字符串转换成它的语音表示法代码。difference函数将两个字符串转换成它们的语音表示法代码并且接着报告能匹配代码位置的数量。由于语音表示法代码具有四个字符，结果可以从零到四，零表示没有匹配而四表示完全匹配（因此这个函数的命名并不适当 — similarity才是更合适的名称）。

这里有一些例子：

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.15.2. Levenshtein

这个函数计算两个字符串之间的编辑距离。

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost)
 返回 int
levenshtein(text source, text target) 返回 int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int
 sub_cost, int max_d) 返回 int
levenshtein_less_equal(text source, text target, int max_d) 返回 int
```

source以及target都可以是任何非空字符串， 最长为 255 个字符。代价参数分别指定一个字符插入、删除或替换的开销。 你可以像这个函数的第二种版本那样忽略代价参数，那样它们都会默认为 1。

levenshtein_less_equal是 Levenshtein 函数的速度更快 的版本，它被用于只对小距离感兴趣的情况。如果实际距离小于等于max_d， 那么levenshtein_less_equal返回正确的距离。否则它返回某个 大于max_d的值。如果max_d是负值，那么其行为等同于 levenshtein。

例子：

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.15.3. Metaphone

和 Soundex 相似，Metaphone 的思想是构建一个输入字符串的一种代码。如果两个字符串具有相同的代码则认为它们相似。

这个函数计算一个输入字符串的变音位代码：

```
metaphone(text source, int max_output_length) 返回 text
```

source必须是一个非空字符串，最大长度为 255 个字符。max_output_length设置输出的变音位代码的最大长度，如果超长，输出会被截断到这个长度。

例子：

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.15.4. 双 Metaphone

双变音位系统为一个给定输入字符串计算两个“听起来像的”字符串 — 一个“主要”代码和一个“次要”代码。在大部分情况下它们是相同的，但是对于非英语名称它们可能有一点不同，这取决于发音。这些函数计算主要和次要代码：

```
dmetaphone(text source) 返回 text
dmetaphone_alt(text source) 返回 text
```

对输入字符串没有长度限制。

例子：

```
test=# SELECT dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.16. hstore

这个模块实现了hstore数据类型用来在一个单一PostgreSQL值中存储键值对。这在很多情景下都有用，例如带有很多很少被检查的属性的行或者半结构化数据。键和值都是简单的文本字符串。

F.16.1. hstore 外部表示

一个hstore的文本表示用于输入和输出，包括零个或者多个由逗号分隔的key => value对。一些例子：

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

键值对的顺序没有意义（并且在输出时也不会重现）。键值对之间或者=>号周围的空白会被忽略。双引号内的键和值可以包括空白、逗号、=或>。要在一个键或值中包括一个双引号或一个反斜线，用一个反斜线对它转义。

一个hstore中的每一个键是唯一的。如果你声明了一个有重复键的hstore，只有一个会被存储在hstore中并且无法保证哪一个将被保留：

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

一个值（但不是一个键）能够是一个 SQL NULL。例如：

```
key => NULL
```

NULL关键词是大小写不敏感的。将NULL放在双引号中可以将它当作一个普通的字符串“NULL”。

> **注意**
>
> 记住当hstore文本格式当被用于输入时，它应用在任何必须的引用或转义之前。如果你通过一个参数传递一个hstore文字，那么不需要额外的处理。但是如果你将它作为一个引用的文字常数，那么任何单引号字符以及（取决于standard_conforming_strings配置参数的设置）反斜线字符需要被正确地转义。更多关于处理字符串常量的处理可见第 4.1.2.1 节

在输出时，双引号总是围绕着键和值，即使这样做不是绝对必要。

F.16.2. hstore 操作符和函数

hstore模块所提供的操作符显示在表 F.7中，函数在表 F.8中。

表 F.7.hstore 操作符

操作符	描述	例子	结果
hstore -> text	为键得到值（不存在则是NULL）	'a=>x, b=>y'::hstore -> 'a'	x
hstore -> text[]	为多个键得到值（不存在则是NULL）	'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']	{"z","x"}
hstore || hstore	串接hstore	'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore	"a"=>"b", "c"=>"x", "d"=>"q"
hstore ? text	hstore是否包含键?	'a=>1'::hstore ? 'a'	t
hstore ?& text[]	hstore是否包含所有指定的键?	'a=>1,b=>2'::hstore ?& ARRAY['a','b']	t
hstore ?| text[]	hstore是否包含任何指定的键?	'a=>1,b=>2'::hstore ?| ARRAY['b','c']	t
hstore @> hstore	左操作数是否包含右操作数?	'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'	t
hstore <@ hstore	左操作数是否被包含在右操作数中?	'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'	f
hstore - text	从左操作数中删除键	'a=>1, b=>2, c=>3'::hstore - 'b'::text	"a"=>"1", "c"=>"3"
hstore - text[]	从左操作数中删除多个键	'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']	"c"=>"3"
hstore - hstore	从左操作数中删除匹配的对	'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore	"a"=>"1", "c"=>"3"
record #= hstore	用来自hstore的匹配值替换record中的域	见示例小节	
%% hstore	把hstore转换成键和值交替出现的数组	%% 'a=>foo, b=>bar'::hstore	{a,foo,b,bar}
%# hstore	把hstore转换成二维的键值数组	%# 'a=>foo, b=>bar'::hstore	{{a,foo},{b,bar}}

> **注意**
>
> 在 PostgreSQL 8.2 之前，包含操作符@>和<@分别被称为@和~。这些名称仍然可用，但是已经被弃用并且最终将被移除。注意，旧名称和原来核心几何数据类型所遵循的习惯是相反的！

表 F.8.hstore 函数

函数	返回类型	描述	例子	结果
hstore(record)	hstore	从一个记录或行构造一个hstore	hstore(ROW(1,2))	f1=>1,f2=>2
hstore(text[])	hstore	从一个数组构造一个hstore，数组可以是一个键值数组或者一个二维数组	hstore(ARRAY['a','1','b','2']) || hstore(ARRAY[['c','3'],['d','4']])	a=>1, b=>2, c=>3, d=>4
hstore(text[], text[])	hstore	从独立的键和值数组构建一个hstore	hstore(ARRAY['a','b'], ARRAY['1','2'])	"a"=>"1","b"=>"2"
hstore(text, text)	hstore	构造单一项的hstore	hstore('a', 'b')	"a"=>"b"
akeys(hstore)	text[]	取得hstore的键作为一个数组	akeys('a=>1,b=>2')	{a,b}
skeys(hstore)	setof text	取得hstore的键作为一个集合	skeys('a=>1,b=>2')	a b
avals(hstore)	text[]	取得hstore的值作为一个数组	avals('a=>1,b=>2')	{1,2}
svals(hstore)	setof text	取得hstore的值作为一个集合	svals('a=>1,b=>2')	1 2
hstore_to_array(hstore)	text[]	取得hstore的键和值作为一个键和值交替出现的数组	hstore_to_array('a=>1,b=>2')	{a,1,b,2}
hstore_to_matrix(hstore)	text[]	取得hstore的键和值作为一个二维的数组	hstore_to_matrix('a=>1,b=>2')	{{a,1},{b,2}}
hstore_to_json(hstore)	json	取得hstore作为一个json值，把所有非空值转换为 JSON 字符串	hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}
hstore_to_jsonb(hstore)	jsonb	取得hstore作为一个jsonb值，把把所有非空值转换为 JSON 字符串	hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}
hstore_to_json_loose(hstore)	json	取得hstore作为一个json值，但是尝试区分数字值和布尔值这样它们在 JSON 中无需引用	hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}

函数	返回类型	描述	例子	结果
hstore_to_jsonb_loose(hstore)	jsonb	取得hstore作为一个jsonb值，但是尝试区分数字值和布尔值这样它们在 JSON 中无需引用	hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}
slice(hstore, text[])	hstore	从一个hstore抽取一个子集	slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x'])	"b"=>"2", "c"=>"3"
each(hstore)	setof(key text, value text)	取得hstore的键和值作为一个集合	select * from each('a=>1,b=>2')	key | value -----+------- a | 1 b | 2
exist(hstore,text)	boolean	hstore是否包含键?	exist('a=>1','a')	t
defined(hstore,text)	boolean	hstore是否为键包含非NULL值?	defined('a=>NULL','a')	f
delete(hstore,text)	hstore	删除匹配键的对	delete('a=>1,b=>2','b')	"a"=>"1"
delete(hstore,text[])	hstore	删除匹配多个键的多个对	delete('a=>1,b=>2,c=>3','ARRAY['a','b'])	"c"=>"3"
delete(hstore,hstore)	hstore	删除匹配第二个参数的对	delete('a=>1,b=>2','a=>4,b=>2'::hstore)	"a"=>"1"
populate_record(**record**,hstore)	**record**	用来自hstore的匹配值替换record中的域	见示例小节	

> **注意**
>
> 当一个hstore值被造型成json时，将使用函数hstore_to_json。同样地，当一个hstore值被造型成jsonb时，将使用函数hstore_to_jsonb。

> **注意**
>
> 函数populate_record实际上被声明为第一个参数为anyelement而非record，但是它将会用一个运行时错误拒绝非记录类型。

F.16.3. 索引

hstore有对@>、?、?&和?|操作符的 GiST 和 GIN 索引支持。例如：

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

hstore也为=操作符支持btree或hash索引。这允许hstore列被声明为UNIQUE或者被使用在GROUP BY、ORDER BY或DISTINCT表达式中。hstore值的排序顺序不是特别有用，但是这些索引可能对等值查找有用。为=比较创建以下索引：

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.16.4. 例子

增加一个键，或者用一个新值更新一个现有的键：

```
UPDATE tab SET h = h || hstore('c', '3');
```

删除一个键：

```
UPDATE tab SET h = delete(h, 'k1');
```

将一个record转换成一个hstore：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                  hstore
-------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

将一个hstore转换成一个预定义的record类型：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

用来自于一个hstore的值修改一个现有的记录：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.16.5. 统计

由于hstore类型本质的宽大性，它能够包含一些不同的键。检查合法键是应用的任务。下列例子验证了用于检查键以及获得统计的一些技术。

简单例子：

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

使用一个表：

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

在线统计：

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.16.6. 兼容性

从 PostgreSQL 9.0 开始，hstore使用了与之前版本不同的内部表示。这不会为转储/恢复升级造成障碍，因为文本表示（用于转储）没有改变。

在一次二进制升级中，通过让新代码识别旧格式数据来维持向上兼容。当处理还没有被新代码修改过的数据时，这会带来一定的性能惩罚。可以通过执行一个下面的UPDATE语句来强制升级表中的所有值：

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

另一种方法：

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

ALTER TABLE方法要求表上的一个排他锁，但是不会导致表因为旧行版本而膨胀。

F.16.7. 转换

有一些额外的扩展为语言 PL/Perl 和 PL/Python 实现了hstore 类型的转换。用于 PL/Perl 的扩展叫做hstore_plperl和 hstore_plperlu，分别用于可信的和不可信的 PL/Perl。如果安装这些转换并且在创建函数时指定它们，hstore值会被 映射成 Perl 哈希。用于 PL/Python 的扩展是hstore_plpythonu、 hstore_plpython2u和hstore_plpython3u（ PL/Python 命名习惯见第 46.1 节。如果使用它们， hstore值会被映射成 Python 字典。

F.16.8. 作者

Oleg Bartunov <oleg@sai.msu.su>，俄罗斯莫斯科大学

Teodor Sigaev <teodor@sigaev.ru>，俄罗斯德尔塔软件有限公司

额外的提升由英国的 Andrew Gierth <andrew@tao11.riddles.org.uk> 提供

F.17. intagg

intagg模块提供了一个整数聚集器以及一个枚举器。intagg现在已被弃用，因为有内建的函数能提供其功能的超集。不过，该模块仍然作为内建函数的一个兼容性包装器被提供。

F.17.1. 函数

聚集器是一个聚集函数int_array_aggregate(integer)，它产生一个完全包含输入整数的整数数组。这是一个array_agg的包装器，它对任何数组类型做同样的事情。

枚举器是一个函数int_array_enum(integer[])，它返回setof integer。它本质上是聚集器的一个逆操作：给定一个整数数组，将它展开成行的集合。这是unnest的一个包装器，它对任何数组类型做相同的事情。

F.17.2. 使用示例

很多数据库系统都有一对多表的概念这样一个表通常位于两个被索引表之间，例如：

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

通常这样用它：

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = item;
```

这将为左手表中的一个项返回右手表中的所有项。这在 SQL 中是很常见的结构。

现在，这种方法对于具有非常多项的one_to_many表会有麻烦。通常，这样的一次连接将为一个特定的左手项导致一次索引扫描以及为每一个右手项导致一次取出。如果你有一个非常动态的系统，你没什么可做的。但是，如果你的数据相对比较静态，你可以使用聚集器创建一个汇总表。

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

这将创建一个表，其中对每一个左项都有一行，并且有一个右项的数组。现在如果没有某种方法来使用该数组，这样做是没有任何用处的。这时数组枚举器就派上用场了，你可以

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

上述使用int_array_enum的查询产生与以下语句相同的结果

```
SELECT left, right FROM one_to_many WHERE left = item;
```

区别是针对汇总表的查询必须只从表中得到一行，而针对one_to_many的直接查询必须索引扫描并且为每项都获取一行。

在一个系统上，一个EXPLAIN显示一个查询的代价从 8488 降低到了 329。原始查询是一个涉及到one_to_many表的连接，它被替换为：

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.18. intarray

intarray模块提供了一些有用的函数和操作符来操纵不含空值的整数数组。也提供了对使用某些操作符的索引搜索的支持。

如果一个提供的数组中包含任何 NULL 元素，所有这些操作都将抛出一个错误。

很多这些操作只对一维数组有意义。尽管它们将接受更多维数的数组输入，数据将被当作一个按照存储顺序排列的线性数组对待。

F.18.1. intarray 函数和操作符

intarray模块提供的函数被列在表 F.9中，操作符被列在表 F.10中。

表 F.9. intarray 函数

函数	返回类型	描述	例子	结果
icount(int[])	int	数组中元素的数量	icount('{1,2,3}'::int[])	3
sort(int[], text dir)	int[]	排序数组——dir必须是asc或desc	sort('{1,2,3}'::int[], 'desc')	{3,2,1}
sort(int[])	int[]	以升序排序	sort(array[11,77,44])	{11,44,77}
sort_asc(int[])	int[]	以升序排序		
sort_desc(int[])	int[]	以降序排序		
uniq(int[])	int[]	移除临近的重复	uniq(sort('{1,2,3,2,1}'::int[]))	{1,2,3}
idx(int[], int item)	int	匹配item的第一个元素的索引（如果没有为0）	idx(array[11,22,33,22,11], 22)	2
subarray(int[], int start, int len)	int[]	从位置start开始的由len个元素组成的元组部分	subarray('{1,2,3,2,1}'::int[], 2, 3)	{2,3,2}
subarray(int[], int start)	int[]	从位置start开始的元组部分	subarray('{1,2,3,2,1}'::int[], 2)	{2,3,2,1}
intset(int)	int[]	创建单一元素数组	intset(42)	{42}

表 F.10. intarray 操作符

操作符	返回	描述
int[] && int[]	boolean	重叠 — 如果数组有至少一个公共元素，则为true

操作符	返回	描述
int[] @> int[]	boolean	包含 — 如果左数组包含右数组，则为true
int[] <@ int[]	boolean	被包含 — 如果左数组被右数组包含，则为true
# int[]	int	数组中元素的数目
int[] # int	int	索引（与idx函数相同）
int[] + int	int[]	把元素推到数组中（增加到数组末尾）
int[] + int[]	int[]	数组串接（把右数组增加到左数组的末尾）
int[] - int	int[]	从数组中移除匹配右参数的项
int[] - int[]	int[]	从左数组中移除右数组的元素
int[] | int	int[]	参数的联合
int[] | int[]	int[]	数组的联合
int[] & int[]	int[]	数组的交
int[] @@ query_int	boolean	如果数组满足查询（见下文），则为true
query_int ~~ int[]	boolean	如果数组满足查询（@@交换子），则为true

（在 PostgreSQL 8.2 之前，包含操作符@>和<@分别被称为@和~。这些名称仍然有效，但是已被弃用并且将最终被移除。注意旧名称与核心几何数据类型之前所遵循的习惯相反！）

操作符&&、@>和<@等效于PostgreSQL的内建同名操作符，不过它们只能在不含空值的整数数组上工作，而内建的操作符可以对任何数组类型工作。这种限制使它们在很多情况下比内建操作符更快。

@@和~~操作符测试一个数组是否满足一个query，它被表示成一种特殊数据类型query_int的一个值。一个由整数值组成的查询会被针对数组的元素检查，可能会组合使用操作符&（AND）、|（OR）以及!（NOT）。根据需要可以使用圆括号。例如，查询1&(2|3)匹配包含 1 并且还包括 2 或 3 的数组。

F.18.2. 索引支持

intarray提供对于&&、@>、<@和@@操作符以及常规数组相等的索引支持。

提供了两种 GiST 索引操作符类：gist__int_ops（被默认使用）适合于中小尺寸的数据集，而gist__intbig_ops使用一种更大的签名并且更适合于索引大型数据集（即，包含大量可区分数组值的列）。该实现使用了一种带有内建有损压缩的 RD 树结构。

也有一种非默认的 GIN 操作符类gin__int_ops支持相同的操作符。

在 GiST 和 GIN 索引之间的选择取决于 GiST 和 GIN 的相对性能特点， 这将在其他地方讨论。

F.18.3. 例子

-- 一个消息可以在一个或多个“小节”中

```
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- 创建专门的索引
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- 选择小节 1 或 2 中的消息 - OVERLAP 操作符
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- 选择小节 1 和 2 中的消息 - CONTAINS 操作符
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- 相同，使用 QUERY 操作符
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.18.4. 测试基准

源代码目录contrib/intarray/bench包含有一个基准测试套件，可以针对 一个安装好的PostgreSQL服务器运行这个套件（还要求安装 DBD::Pg）。要运行测试基准：

```
cd .../contrib/intarray/bench
createdb TEST
psql -c "CREATE EXTENSION intarray" TEST
./create_test.pl | psql TEST
./bench.pl
```

bench.pl脚本有多个选项，当它不使用任何参数运行时会显示这些选项。

F.18.5. 作者

所有工作由 Teodor Sigaev (<teodor@sigaev.ru>) 和 Oleg Bartunov (<oleg@sai.msu.su>) 完成。更多信息请见http://www.sai.msu.su/~megera/postgres/gist/。Andrey Oktyabrski 完成了一项伟大的工作，他增加了新的函数和操作。

F.19. isn

isn模块为下列国际产品编号标准提供数据类型：EAN13、UPC、ISBN（图书）、ISMN（音乐）以及 ISSN（期刊）。在输入时会按照一个硬编码的前缀列表对输入进行验证，这个前缀的列表也被用来在输出时连接号码。因为新的前缀总是不时地出现，这个前缀列表可能会过时。这个模块的一个未来版本有希望得到一个来自于一个或多个表的前缀列表，这样用户可以根据需要来方便地更新前缀列表。不过，在当前该列表只能通过修改源代码并且重新编译来更新。另外一种方案是，在这个模块的未来版本中可能会直接移除掉前缀验证和连接支持。

F.19.1. 数据类型

表 F.1展示了isn模块提供的数据类型。

表 F.11.isn 数据类型

数据类型	描述
EAN13	欧洲文章号，总是以 EAN13 格式显示
ISBN13	国际标准图书号，以新的 EAN13 格式显示
ISMN13	国际标准音乐号，以新的 EAN13 格式显示

数据类型	描述
ISSN13	国际标准期刊号，以新的 EAN13 格式显示
ISBN	国际标准图书号，以旧的短格式显示
ISMN	国际标准音乐号，以旧的短格式显示
ISSN	国际标准期刊号，以旧的短格式显示
UPC	通用产品代码

一些注记：

1. ISBN13、ISMN13、ISSN13 号码都是 EAN13 号码。

2. EAN13 号码不总是 ISBN13、ISMN13 或 ISSN13（有些是）。

3. 一些 ISBN13 号码能够作为 ISBN 显示。

4. 一些 ISMN13 号码能够作为 ISMN 显示。

5. 一些 ISSN13 号码能够作为 ISSN 显示。

6. UPC 号码是 EAN13 号码的一个子集（它们基本上是去掉了第一个0位的 EAN13）。

7. 所有 UPC、ISBN、ISMN 以及 ISSN 号码可以被表示为 EAN13 号码。

在内部，所有这些类型使用同一种表达（一个 64 位整数）并且所有内部表达是可以互换的。多种类型被提供来控制显示格式化并且对假定为表示一种特定类型号码的输入进行更严格的合法性检查。

在可能时，ISBN、ISMN和ISSN类型将显示号码的短版本（ISxN 10），并且在无法适应短版本时显示号码的 ISxN 13 格式。EAN13、ISBN13、ISMN13和ISSN13类型总是显示长版本的ISxN（EAN13）。

F.19.2. 造型

isn模块提供了下列类型之间的造型：

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

当从EAN13造型为另一种类型时， 会有对该值是否在另一种类型的域中的运行时检查，如果不在则抛出一个错误。其他的造型则是简单地重新贴个标签，因而总是会成功。

F.19.3. 函数和操作符

isn模块提供了标准的比较操作符，外加对所有这些数据类型的 B 树和哈希索引支持。此外还有一些特殊的函数，它们展示在表 F.12中。在这个表中，isn意味着该模块的数据类型中的任何一种。

表 F.12.isn 函数

函数	返回	描述
isn_weak(boolean)	boolean	设置弱输入模式（返回新设置）
isn_weak()	boolean	得到弱模式的当前状态
make_valid(isn)	isn	验证一个非法号码（清除非法标志）
is_valid(isn)	boolean	检查非法标志的存在

弱模式被用来允许插入非法数据到一个表中。非法意味着校验位错误，而不是有丢失号码。

为什么你会想要使用弱模式？你可能有一个巨大的 ISBN 号码集合并且出于某种奇怪的原因其中具有错误的校验位（可能这些号码是从印刷稿中扫描并且 OCR 而来，也可能是手工输入的.....谁知道呢）。不管怎样，重点是你可能希望清理这些混乱，但是你仍然想要能够把这些号码放在你的数据库中并且可能会使用一个外部工具在数据库中定位非法号码，这样你能够更容易地验证信息。因此你可能会想要在表中选择所有非法的号码。

当你使用弱模式在一个表中插入非法号码时，被插入的号码将会被加上修正过的校验位，但是它的最后将会有一个感叹号（!），例如0-11-000322-5!。这种非法标志符可以用is_valid函数检查并且可以用make_valid函数清除。

即使不在弱模式中，你也能通过在号码某位追加!字符来强制非法号码的插入。

另一个特殊特性是在输入过程中，你可以写一个?代替校验位，然后正确的校验位将被自动插入。

F.19.4. 例子

```
--直接使用类型：
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--转换类型：
-- 注意只有在号码处于另一种类型的合法值之中时，才能从 EAN13 转换成另一种类型
-- 因此下面的用法将不会工作： select isbn(ean13('0220356483481'));
-- 但是下面的可以：
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--创建一个表，它有一个单一列来保存 ISBN 号码：
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--自动计算校验位（观察 '?'）：
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');
```

```
SELECT issn('3251231?');
SELECT ismn('979047213542?');

--使用弱模式：
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.19.5. 参考文献

实现这个模块的信息可以从下列网站收集到：

- https://www.isbn-international.org/
- http://www.issn.org/
- https://www.ismn-international.org/
- https://www.wikipedia.org/

用于连接的前缀：

- https://www.gs1.org/standards/id-keys
- https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups
- https://www.isbn-international.org/content/isbn-users-manual
- https://en.wikipedia.org/wiki/International_Standard_Music_Number
- https://www.ismn-international.org/ranges.html

创建算法时已经注意严谨地使用 ISBN、ISMN、ISSN 官方用户手册中的推荐算法进行了验证。

F.19.6. 作者

Germán Méndez Bravo (Kronuz), 2004 - 2006

这个模块受到了 Garrett A. Wollman 的isbn_issn代码的启发。

F.20. lo

lo模块提供管理大对象（也被称为 LO 或 BLOB）的支持。这包括一种数据类型lo以及一个触发器lo_manage。

F.20.1. 原理

JDBC 驱动的问题之一（并且这也影响 ODBC 驱动）是其说明书假定对 BLOB（二进制大对象）的引用被存储在一个表中，并且如果该项被改变相关的 BLOB 会被从数据库删除。

但对于PostgreSQL来说这并不会发生。大对象被当做自主的对象，一个表项可以通过 OID 引用一个大对象，但是可以有多个表项引用同一个大对象 OID，因此系统不会因为你改变或者删除这种项而删除大对象。

现在这对PostgreSQL-相关的应用挺好的，但是使用 JDBC 或 ODBC 的标准代码不会删除那些对象，从而导致孤立对象 — 不被任何东西引用的对象，而且会占据磁盘空间。

lo允许通过附加一个触发器到包含 LO 引用列的表来修复这种问题。该触发器本质上只是在你删除或修改一个引用大对象的值时做lo_unlink。当你使用这个触发器时，你必须假定在一个触发器控制的列中只有一个对任意大对象的数据库引用！

这个模块也提供了一种数据类型lo，它实际上只是oid类型的一个域。这有助于区分保存大对象引用的数据库列和保存其他东西 OID 的列。你并不是必须使用lo类型来使用该触发器，但是用它来追踪数据库中哪些列表示你要用触发器管理的大对象非常方便。也有传言说如果你不为 BLOB 列使用lo，ODBC 驱动会感到困惑。

F.20.2. 如何使用它

这里是一个简单的用法示例：

```
CREATE TABLE image (title text, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

对每一个将包含到大对象唯一引用的列，创建一个BEFORE UPDATE OR DELETE触发器，并且将该列名作为唯一的触发器参数。你也可以用BEFORE UPDATE OF column_name来限制该触发器只对该列上的更新事件执行。如果你需要在同一个表中有多个lo列，为每一个创建一个独立的触发器，记住为同一个表上的每个触发器指定一个不同的名称。

F.20.3. 限制

- 删除一个表仍将让它包含的任何对象变成孤立的，因为触发器在这种情况下不会被执行。你可以在DROP TABLE之前放上DELETE FROM table来避免这种问题。

  TRUNCATE有同样的危害。

  如果你已经有或者怀疑有孤立的大对象，参考vacuumlo模块可以帮助你清理它们。偶尔运行vacuumlo作为lo_manage触发器的后备是个好主意。

- 有些前端可能会创建它们自己的表，并且将不会创建相关的触发器。另外，用户可能不会记得（或知道）要创建触发器。

F.20.4. 作者

Peter Mount <peter@retep.org.uk>

F.21. ltree

这个模块实现了一种数据类型ltree用于表示存储在一个层次树状结构中的数据的标签。还提供了在标签树中搜索的扩展功能。

F.21.1. 定义

一个标签是一个字母数字字符和下划线的序列（例如，在 C 区域中允许字符A-Za-z0-9_）。标签长度必须少于 256 字节。

例子： 42, Personal_Services

一个标签路径是由点号分隔的零个或者更多个标签的序列，例如L1.L2.L3，它表示一个从层次树的根到一个特定节点的路径。一个标签路径的长度必须小于 65kB，但是最好将它保持在 2kB 以下。

例子：Top.Countries.Europe.Russia

ltree模块提供多种数据类型：

- ltree存储一个标签路径。
- lquery表示一个用于匹配ltree值的类正则表达式的模式。一个简单词匹配一个路径中的那个标签。一个星号（*）匹配零个或更多个标签。例如：

  ```
  foo         正好匹配标签路径foo
  *.foo.*     匹配任何包含标签foo的标签路径
  *.foo       匹配任何最后一个标签是foo的标签路径
  ```

  星号也可以被限定来限制它能匹配多少标签：

  ```
  *{n}        匹配正好n个标签
  *{n,}       匹配至少n个标签
  *{n,m}      匹配至少n个但是最多m个标签
  *{,m}       匹配最多m个标签 — 与*{0,m}相同
  ```

  在lquery中，有多种修饰符可以被放在一个非星号标签的末尾来让它不仅仅能准确匹配：

  ```
  @           不区分大小写匹配，例如a@匹配A
  *           匹配带此前缀的任何标签，例如foo*匹配foobar
  %           匹配开头以下划线分隔的词
  ```

  %的行为有点复杂。它尝试匹配词而不是整个标签。例如，foo_bar%匹配foo_bar_baz但是不匹配foo_barbaz。如果和*组合，前缀匹配可以单独应用于每一个词，例如foo_bar%*匹配foo1_bar2_baz但不匹配foo1_br2_baz。

  此外，你可以写多个带有|（OR）的可能改过的标签来匹配那些标签中的任何一个（或几个），并且你可以在最前面放上!（NOT）来匹配任何不匹配那些分支的标签。

  这里是一个lquery的例子：

  ```
  Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
  a.   b.      c.       d.                 e.
  ```

  这个查询将匹配任何这样的标签路径：

  a. 开始于标签Top

  b. 并且接着具有 0 到 2 个标签

  c. 之后是一个开始于大小写无关的前缀sport的标签

  d. 再后是一个不匹配football和tennis的标签

  e. 并且结尾是一个开始于Russ的标签，或者完全匹配Spain的标签。

- ltxtquery表示一种用于匹配ltree值的类全文搜索的模式。一个ltxtquery值包含词，也可能在末尾带有修饰符@、*、%，修饰符具有和lquery中相同的含义。词可以

用&（AND）、|（OR）、!（NOT）以及圆括号组合。lquery和ltxtquery的关键区别是前者匹配词时不考虑它们在标签路径中的位置。

这是一个ltxtquery的例子：

```
Europe & Russia*@ & !Transportation
```

这将匹配包含标签Europe以及任何以Russia开始（大小写不敏感）的标签的路径，但是不匹配包含标签Transportation的路径。这些词在路径中的位置并不重要。还有，当使用%时，该次可以与一个标签中任何下划线分隔的词匹配，而不管它们的位置如何。

注意：ltxtquery允许符号之间的空白，但是ltree和lquery不允许。

F.21.2. 操作符和函数

类型ltree有普通比较操作符 =、<>、 <、>、<=、>=。 比较会按照树遍历的顺序排序，一个节点的子女按照标签文本排序。另外，还有表 F.13中显示的特殊操作符。

表 F.13. ltree 操作符

操作符	返回值	描述
ltree @> ltree	boolean	左参数是不是右参数的一个祖先（或者相等）？
ltree <@ ltree	boolean	左参数是不是右参数的一个后代（或者相等）？
ltree ~ lquery	boolean	ltree匹配lquery吗？
lquery ~ ltree	boolean	ltree匹配lquery吗？
ltree ? lquery[]	boolean	ltree匹配数组中的任意lquery吗？
lquery[] ? ltree	boolean	ltree匹配数组中的任意lquery吗？
ltree @ ltxtquery	boolean	ltree匹配ltxtquery吗？
ltxtquery @ ltree	boolean	ltree匹配ltxtquery吗？
ltree || ltree	ltree	串接ltree路径
ltree || text	ltree	把文本转换成ltree并且串接
text || ltree	ltree	把文本转换成ltree并且串接
ltree[] @> ltree	boolean	数组是否包含ltree的一个祖先？
ltree <@ ltree[]	boolean	数组是否包含ltree的一个祖先？
ltree[] <@ ltree	boolean	数组是否包含ltree的一个后代？
ltree @> ltree[]	boolean	数组是否包含ltree的一个后代？
ltree[] ~ lquery	boolean	数组是否包含匹配lquery的路径？
lquery ~ ltree[]	boolean	数组是否包含匹配lquery的路径？
ltree[] ? lquery[]	boolean	ltree数组是否包含匹配任意lquery的路径？

操作符	返回值	描述
lquery[] ? ltree[]	boolean	ltree数组是否包含匹配任意lquery的路径？
ltree[] @ ltxtquery	boolean	数组是否包含匹配ltxtquery的路径？
ltxtquery @ ltree[]	boolean	数组是否包含匹配ltxtquery的路径？
ltree[] ?@> ltree	ltree	是ltree祖先的第一个数组项；如果没有则是 NULL
ltree[] ?<@ ltree	ltree	是ltree祖先的第一个数组项；如果没有则是 NULL
ltree[] ?~ lquery	ltree	匹配lquery的第一个数组项；如果没有则是 NULL
ltree[] ?@ ltxtquery	ltree	匹配lquery的第一个数组项；如果没有则是 NULL

操作符<@、@>、 @以及~有类似的、 ^<@、^@>、^@、 ^~，只是它们不适用索引。它们只对测试目的有用。

可用的函数在表 F.14中。

表 F.14. ltree 函数

函数	返回类型	描述	例子	结果
subltree(ltree, int start, int end)	ltree	ltree的从位置start到位置end-1（从 0 开始计）的子路径	subltree('Top.Child1.Child2',1,2)	Child1.Child2
subpath(ltree, int offset, int len)	ltree	ltree从位置offset开始长度为len的子路径。如果offset为负，则子路径开始于距离路径尾部那么远的位置。如果len为负，则从路径的尾部开始丢掉那么多个标签。	subpath('Top.Child1.Child2',0,2)	Top.Child1
subpath(ltree, int offset)	ltree	ltree从位置offset开始一直延伸到路径末尾的子路径。如果offset为负，则子路径开始于距离路径尾部那么远的位置。	subpath('Top.Child1.Child2',1)	Child1.Child2
nlevel(ltree)	integer	路径中标签的数量	nlevel('Top.Child1.Child2')	3
index(ltree a, ltree b)	integer	a中第一次出现b的位置，如果没有找到则为 -1	index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')	4

函数	返回类型	描述	例子	结果
index(ltree a, ltree b, int offset)	integer	a中第一次出现b的位置，搜索从offset开始。负的offset表示从距路径尾部-offset个标签的位置开始	index('0.1.2.3.5'94.5.6.8.5.6.8','5.6',-4)	
text2ltree(text)	ltree	把text转换成ltree		
ltree2text(ltree)	text	把ltree转换成text		
lca(ltree, ltree, ...)	ltree	路径的最长公共祖先（最多支持 8 个参数）	lca('1.2.3','1.2.1324.5.6')	
lca(ltree[])	ltree	数组中路径的最长公共祖先	lca(array['1.2.3'1:2ltree,'1.2.3.4'])	

F.21.3. 索引

ltree支持一些能加速上述操作符的索引类型：

- ltree上的 B-树索引： <、<=、=、 >=、>
- ltree上的 GiST 索引： <、<=、=、 >=、>、 @>、<@、 @、~、?

  创建这样一个索引的例子：

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

- ltree[]上的 GiST 索引： ltree[] <@ ltree、ltree @> ltree[]、 @、~、?

  创建这样一个索引的例子：

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  注意：这种索引类型是有损的。

F.21.4. 例子

这个例子使用下列数据（在源代码发布的contrib/ltree/ltreetest.sql文件中也有）：

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
```

```
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

现在，我们有一个表test，它被填充了描述下列层次的数据：

```
                      Top
                   /   |  \
             Science Hobbies Collections
                /      |              \
        Astronomy   Amateurs_Astronomy Pictures
           /  \                          |
Astrophysics  Cosmology                Astronomy
                                        /  |    \
                                  Galaxies Stars Astronauts
```

我们可以做继承：

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
-----------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

这里是一些路径匹配的例子：

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

这里是一些全文搜索的例子：

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
```

```
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

使用函数的路径构建：

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE
 path <@ 'Top.Science.Astronomy';
                 ?column?
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

我们可以通过常见一个在路径中指定位置插入标签的 SQL 函数来简化：

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@
 'Top.Science.Astronomy';
                ins_label
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.21.5. 转换

有一些额外的扩展为 PL/Python 实现了ltree类型的转换。 这些扩展是ltree_plpythonu、ltree_plpython2u 以及ltree_plpython3u（PL/Python 命名习惯请见第 46.1 节。如果安装了这些转换并且在 创建函数时指定了它们，ltree值会被映射为 Python 列表（ 不过，当前并不支持逆向的转换）。

F.21.6. 作者

所有工作都是 Teodor Sigaev（<teodor@stack.net>）和 Oleg Bartunov（<oleg@sai.msu.su>）完成的。额外信息可见 http://www.sai.msu.su/~megera/postgres/gist/。作者还要感谢 Eugeny Rodichev 参与讨论。欢迎评论和缺陷报告。

F.22. pageinspect

pageinspect模块提供函数让你从低层次观察数据库页面的内容，这对于调试目的很有用。所有这些函数只能被超级用户使用。

F.22.1. 通用函数

get_raw_page(relname text, fork text, blkno int) 返回 bytea

get_raw_page读取提及的关系中的指定块并且以一个bytea值的形式返回一个拷贝。这允许得到该块的一个单一的时间一致的拷贝。对于主数据分叉，fork应该是'main'，对于空闲空间映射应该是'fsm'，对于可见性映射应该是'vm'，对于初始化分叉应该是'init'。

get_raw_page(relname text, blkno int) 返回 bytea

一个简写版的get_raw_page，用于读取主分叉。等效于get_raw_page(relname, 'main', blkno)

page_header(page bytea) 返回 record

page_header显示所有PostgreSQL堆和索引页面的公共域。

用get_raw_page获得的一个页面映像应该作为参数传递。例如：

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version
 | prune_xid
-----------+----------+--------+-------+-------+---------+----------
+---------+-----------
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |     8192 |       4
 |         0
```

返回的列对应于PageHeaderData结构中的域。详见src/include/storage/bufpage.h。

checksum域是存放在页面中的校验和，如果页面被损坏它可能是不正确的。如果对这个实例没有启用数据校验和，则存储的这个值没有意义。

page_checksum(page bytea, blkno int4) returns smallint

page_checksum为页面计算校验和，就像它被放置在给定块上一样。

应该将get_raw_page得到的页面映像作为参数传入。例如：

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
---------------
         13443
```

注意校验和取决于块号，因此应该将匹配的块号传入（除非在做调试）。

用这个函数计算的校验和可以拿来和函数page_header的结果域checksum进行比较。如果为这个实例启用了数据校验和，则两个值应该相等。

heap_page_items(page bytea) 返回 setof record

heap_page_items显示一个堆页面上所有的行指针。 对那些使用中的行指针，元组头部和元组原始数据也会被显示。 不管元组对于拷贝原始页面时的 MVCC 快照是否可见，它们都会被显示。

用get_raw_page获得的一个堆页面映像应该作为参数传递。例如：

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

返回的域的解释可见src/include/storage/itemid.h和src/include/access/htup_details.h。

tuple_data_split(rel_oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]

tuple_data_split以后端内部的相同方式将元组数据拆解成属性。

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask,
 t_infomask2, t_bits) FROM heap_page_items(get_raw_page('pg_class', 0));
```

应该用与heap_page_items的返回属性相同的参数来调用这个函数。

如果do_detoast是true，则根据需要将把属性解除TOAST。默认值为false。

heap_page_item_attrs(rel_oid, t_data bytea, [, do_detoast bool]) returns bytea[]

heap_page_item_attrs等效于 heap_page_items，不过它会把元组原始数据 返回为属性的数组，如果do_detoast为真（ 默认为false），这些属性会被反 TOAST。

应该把用get_raw_page得到的一个堆页面映像 作为参数传入。例如：

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0),
 'pg_class'::regclass);
```

fsm_page_contents(page bytea) returns text

fsm_page_contents展示一个FSM页面的内部节点结构。输出是一个多行字符串，每一行对应于页面中二叉树的每一个节点。只有非零节点才会被打印。所谓的“next”指针（指向页面中下一个要返回的槽）也会被打印。

更多有关FSM页面结构的信息请见src/backend/storage/freespace/README。

F.22.2. B树函数

bt_metap(relname text) 返回 record

bt_metap返回关于一个B树索引元页的信息。例如：

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----------+-------
magic                   | 340322
version                 | 3
root                    | 1
level                   | 0
fastroot                | 1
fastlevel               | 0
oldest_xact             | 582
last_cleanup_num_tuples | 1000
```

bt_page_stats(relname text, blkno int) 返回 record

bt_page_stats返回有关 B-树索引单一页面的总计信息。例如：

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 256
```

```
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

bt_page_items(relname text, blkno int) 返回 setof record

bt_page_items返回一个 B-树索引页面上项的所有细节信息。例如：

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

在一个 B 树叶子页面中，ctid指向一个堆元组。在一个 内部页面中， ctid的块号部分指向索引本身中的另一个 页面，而偏移量部分（第二个数字）会被忽略并且通常为 1。

注意在任何非最右页面（页面的btpo_next域中有非零 值）上的第一个项是该页的“high key”，表示它的 data是作为该页面上所有项的一个上界存在，而它的 ctid域没有意义。还有，在非叶子页面上，第一个真正 的数据项（第一个不是 high key 的项）是一个“负无穷” 项，它的data域中没有实际值。不过，这样一个项确实 在其有ctid域中有向下的链接。

bt_page_items(page bytea) returns setof record

还可以把一个页面以bytea值的形式传递给bt_page_items。应该将get_raw_page得到的页面映像作为参数传入。因此，上一个例子可以被重写成这样：

```
test=# SELECT * FROM bt_page_items(get_raw_page('pg_cast_oid_index', 1));
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

所有其他细节和前一项中的解释相同。

F.22.3. BRIN函数

brin_page_type(page bytea) returns text

brin_page_type返回一个给定的 BRIN索引页面的页面类型，如果该页面不是 一个合法的BRIN页面则抛出错误。例如：

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

brin_metapage_info(page bytea) returns record

brin_metapage_info返回有关一个 BRIN索引元页的各类信息。例如：

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |             4 |              2
```

brin_revmap_data(page bytea) returns setof tid

brin_revmap_data返回一个 BRIN索引范围映射页面中元组标识符的列表。 例如：

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) LIMIT 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

brin_page_items(page bytea, index oid) returns setof record

brin_page_items返回存储在 BRIN数据页面中存储的数据。例如：

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value

------------+--------+--------+----------+----------+-------------
+--------------
        137 |      0 |      1 | t        | f        | f           |
        137 |      0 |      2 | f        | f        | f           | {1 .. 88}
        138 |      4 |      1 | t        | f        | f           |
        138 |      4 |      2 | f        | f        | f           | {89 ..
 176}
        139 |      8 |      1 | t        | f        | f           |
        139 |      8 |      2 | f        | f        | f           | {177 ..
 264}
```

返回的列对应于BrinMemTuple和 BrinValues结构中的域。详见 src/include/access/brin_tuple.h。

F.22.4. GIN函数

gin_metapage_info(page bytea) returns record

gin_metapage_info返回有关一个 GIN索引元页的信息。例如：

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

gin_page_opaque_info(page bytea) returns record

gin_page_opaque_info返回有关一个 GIN索引不透明区域的信息，如页面类型等。例如：

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+-----------------------
         5 |      0 | {data,leaf,compressed}
(1 row)
```

gin_leafpage_items(page bytea) returns setof record

gin_leafpage_items返回有关存储在一个 GIN叶子页面中的数据的信息。例如：

```
test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                          some_tids
-----------+--------
+-------------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 |
 {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 |
 {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 |
 {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

F.22.5. Hash函数

hash_page_type(page bytea) returns text

hash_page_type返回给定的HASH索引页面的页面类型。例如：

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

hash_page_stats(page bytea) returns setof record

hash_page_stats返回有关一个HASH索引的桶页或者溢出页的信息。例如：

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

hash_page_items(page bytea) returns setof record

hash_page_items返回有关一个HASH索引的桶页或者溢出页中存储的数据的信息。例如：

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT
 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
          5 | (890,111) | 1053474816
```

hash_bitmap_info(index oid, blkno int) returns record

hash_bitmap_info返回HASH索引一个特定溢出页在位图页中的位的状态。例如：

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

hash_metapage_info(page bytea) returns record

hash_metapage_info返回一个HASH索引的元页中存放的信息。例如：

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#     maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#     regexp_replace(spares::text, '(,0)*}', '}') as spares,
test-#     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD
 1 ]----------------------------------------------------------------------------
magic     | 105121344
version   | 4
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
```

```
bmshift    | 15
maxbucket  | 12512
highmask   | 16383
lowmask    | 8191
ovflpoint  | 28
firstfree  | 1204
nmaps      | 1
procid     | 450
spares     |
 {0,0,0,0,0,0,1,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,508,567,628,704,1193,1202,1204}
mapp       | {65}
```

F.23. passwordcheck

只要通过CREATE ROLE或ALTER ROLE设置用户， passwordcheck模块会检查用户的口令。如果一个口令被认为太弱，它将被拒绝并且该命令将带着一个错误终止。

要启用这个模块，把'$libdir/passwordcheck'加入到postgresql.conf中的shared_preload_libraries，然后重启服务器。

你可以通过修改源代码来按你的需要修改这个模块。例如，你可以使用CrackLib[2]来检查口令 — 这只需要在Makefile中取消两行的注释并且重新编译该模块（由于授权原因，我们不能默认包括CrackLib）。如果没有CrackLib，该模块会对口令强度强制一些简单的规则，你可以自行修改和扩充。

> **小心**
>
> 要阻止未加密的口令被通过网络传送、写入到服务器日志或者被一个数据库管理员窃取，PostgreSQL允许用户提供预加密的口令。很多客户端程序利用这种功能并且在把口令发送给服务器之前加密它。
>
> 这限制了passwordcheck模块的有用性，因为这种情况下它只能尝试猜测口令。由于这个原因，如果你的安全性需求很高，我们不推荐passwordcheck。使用一个诸如 GSSAPI （见第 20 章的外部认证方法比依赖数据库内的口令更加安全。
>
> 此外，你可以修改passwordcheck来拒绝预加密的口令，但是强制用户将口令设置为明文带来了它的安全风险。

F.24. pg_buffercache

pg_buffercache模块提供了一种方法实时检查共享缓冲区。

该模块提供了一个 C 函数pg_buffercache_pages，它返回一个记录的集合，外加一个包装了该函数以便于使用的视图pg_buffercache。

默认情况下，使用仅限于超级用户和pg_read_all_stats 角色的成员。可以使用GRANT给其他人授予访问权限。

F.24.1. pg_buffercache视图

视图显示的列的定义如表 F.1所示。

[2] https://sourceforge.net/projects/cracklib/

表 F.15.pg_buffercache 列

名称	类型	引用	描述
bufferid	integer		ID，在范围 1..shared_buffers中
relfilenode	oid	pg_class.relfilenode	关系的文件结点编号
reltablespace	oid	pg_tablespace.oid	关系的表空间 OID
reldatabase	oid	pg_database.oid	关系的数据库 OID
relforknumber	smallint		关系内的分叉数，见include/common/relpath.h
relblocknumber	bigint		关系内的页面数
isdirty	boolean		页面是否为脏？
usagecount	smallint		Clock-sweep 访问计数
pinning_backends	integer		对这个缓冲区加 pin 的后端数量

共享缓存中的每一个缓冲区都有一行。没有使用的缓冲区的行中只有bufferid为非空。共享的系统目录被显示为属于数据库零。

因为缓冲是所有数据库共享的，通常会有不属于当前数据库的关系的页面。这意味着对于一些行在pg_class中可能不会有匹配的连接行，或者甚至有错误的连接。如果你试图与pg_class连接，将连接限制于reldatabase等于当前数据库 OID 或零的行是一个好主意。

当访问pg_buffercache视图时，内部缓冲区管理器会被锁住足够长时间来拷贝视图将显示的所有缓冲区状态数据。这确保了该视图会产生一个一致的结果集合，而不会不必要地长时间阻塞普通的缓冲区活动。尽管如此，如果经常读取这个视图还是会对数据库性能产生一些影响。

F.24.2. 样例输出

```
regression=# SELECT c.relname, count(*) AS buffers
             FROM pg_buffercache b INNER JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             GROUP BY c.relname
             ORDER BY 2 DESC
             LIMIT 10;

              relname             | buffers
----------------------------------+---------
 tenk2                            |     345
 tenk1                            |     141
 pg_proc                          |      46
 pg_class                         |      45
 pg_attribute                     |      43
 pg_class_relname_nsp_index       |      30
 pg_proc_proname_args_nsp_index   |      28
 pg_attribute_relid_attnam_index  |      26
 pg_depend                        |      22
 pg_depend_reference_index        |      20
(10 rows)
```

F.24.3. 作者

Mark Kirkwood <markir@paradise.net.nz>

设计建议： Neil Conway <neilc@samurai.com>

调试建议： Tom Lane <tgl@sss.pgh.pa.us>

F.25. pgcrypto

pgcrypto模块为PostgreSQL提供了密码函数。

F.25.1. 普通哈希函数

F.25.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

计算一个给定data的一个二进制哈希值。type是要使用的算法。标准算法是md5、sha1、sha224、sha256、sha384和sha512。如果使用 OpenSSL 编译了pgcrypto，如表 F.19中所述，有更多算法可用。

如果你想摘要成为一个十六进制字符串，可以在结果上使用encode()。例如：

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

为带有密钥key的data计算哈希过的 MAC。type与digest()中相同。

这与digest()相似，但是该哈希只能在知晓密钥的情况下被重新计算出来。这阻止了某人修改数据且还想更改哈希以匹配之的企图。

如果该密钥大于哈希块的尺寸，它将先被哈希然后把结果用作密钥。

F.25.2. 口令哈希函数

函数crypt()和gen_salt()是特别设计用来做口令哈希的。crypt()完成哈希，而gen_salt()负责为前者准备算法参数。

crypt()中的算法在以下方面不同于通常的 MD5 或 SHA1 哈希算法：

1. 它们很慢。由于数据量很小，这是增加蛮力口令破解难度的唯一方法。
2. 它们使用一个随机值（称为salt），这样具有相同口令的用户将得到不同的密文口令。这也是针对逆转算法的一种额外保护。
3. 它们会在结果中包括算法类型，这样用不同算法哈希的口令能共存。
4. 其中一些是自适应的 — 这意味着当计算机变得更快时，你可以调整该算法变得更慢，而不会产生与现有口令的不兼容。

表 F.16列出了crypt()函数所支持的算法。

表 F.16.crypt()支持的算法

算法	最大口令长度	自适应?	Salt 位数	输出长度	描述
bf	72	yes	128	60	基于 Blowfish，变体 2a
md5	unlimited	no	48	34	基于 MD5 的加密
xdes	8	yes	24	20	扩展的 DES
des	8	no	12	13	原生 UNIX 加密

F.25.2.1. crypt()

crypt(password text, salt text) 返回 text

计算password的一个 crypt(3) 风格的哈希。在存储一个新口令时，你需要使用gen_salt()产生一个新的salt值。要检查一个口令，把存储的哈希值作为salt，并且测试结果是否匹配存储的值。

设置一个新口令的例子：

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

认证的例子：

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

如果输入的口令正确，这会返回true。

F.25.2.2. gen_salt()

gen_salt(type text [, iter_count integer]) 返回 text

产生一个在crypt()中使用的新随机 salt 字符串。该 salt 字符串也告诉了crypt()要使用哪种算法。

type参数指定哈希算法。可接受的类型是：des、xdes、md5以及bf。

iter_count参数让用户可以为使用迭代计数的算法指定迭代计数。计数越高，哈希口令花的时间更长并且因而需要更多时间去攻破它。不过使用太高的计数会导致计算一个哈希的时间高达数年 — 这并不使用。如果iter_count参数被忽略，将使用默认的迭代计数。允许的iter_count值与算法相关，如表 F.17所示。

表 F.17.crypt()的迭代计数

算法	默认值	最小值	最大值
xdes	725	1	16777215
bf	6	4	31

对xdes算法还有额外的限制：迭代计数必须是一个奇数。

要选取一个合适的迭代计数，考虑最初的 DES 加密被设计成在当时的硬件上每秒钟完成 4 次哈希。低于每秒 4 次哈希的速度很可能会损害可用性。而超过每秒 100 次哈希又可能太快了。

表 F.18给出了不同哈希算法的相对慢度的综述。该表展示了在假设口令只含有小写字母或者大小写字母及数字的情况下，在一个 8 字符口令中尝试所有字符组合所需要的时间。在crypt-bf项中，在一个斜线之后的数字是gen_salt的iter_count参数

表 F.18. 哈希算法速度

算法	次哈希/秒	对于[a-z]	对于[A-Za-z0-9]	相对于md5 hash的持续时间
crypt-bf/8	1792	4 年	3927 年	100k
crypt-bf/7	3648	2 年	1929 年	50k
crypt-bf/6	7168	1 年	982 年	25k
crypt-bf/5	13504	188 天	521 年	12.5k
crypt-md5	171584	15 天	41 年	1k
crypt-des	23221568	157.5 分	108 天	7
sha1	37774272	90 分	68 天	4
md5(hash)	150085504	22.5 分	17 天	1

注意：

- 使用的机器是一台 Intel Mobile Core i3。
- crypt-des和crypt-md5算法的数字是取自 John the Ripper v1.6.38 -test输出。
- md5 hash的数字来自于 mdcrack 1.2。
- sha1的数字来自于 lcrack-20031130-beta.
- crypt-bf的数字是采用一个在 1000 个 8 字符口令上循环的简单程序采集到的。用那种方法我能展示不同迭代次数的速度。供参考：john-test对于crypt-bf/5显示 13506 次循环/秒（结果中的微小差异符合pgcrypto中的crypt-bf实现与 John the Ripper 中的一致这一情况）。

注意“尝试所有组合”并非是现实中会采用的方式。通常口令破解都是在词典的帮助下完成的，词典中会包含常用词以及它们的多种变化。因此，甚至有些像词的口令被破解的时间可能会大大小于上面建议的数字，而一个 6 字符的不像词的口令可能会逃过破解，也可能不能逃脱。

F.25.3. PGP 加密函数

这里的函数实现了OpenPGP (RFC 4880) 标准的加密部分。对称密钥和公钥加密都被支持。

一个加密的 PGP 消息由两个部分或者包组成：

- 包含一个会话密钥的包 — 加密过的对称密钥或者公钥。
- 包含用会话密钥加密过的数据的包。

当用一个对称密钥（即一个口令）加密时：

1. 给定的口令被使用一个 String2Key (S2K) 算法哈希。这更像crypt()算法 — 有目的地慢并且使用随机 salt — 但是它会产生一个全长度的二进制密钥。
2. 如果要求一个独立的会话密钥，将会生成一个新的随机密钥。否则该 S2K 密钥将被直接用作会话密钥。

3. 如果直接使用 S2K 密钥，那么只有 S2K 设置将被放入会话密钥包中。否则会话密钥会用 S2K 密钥加密并且放入会话密钥包中。

当使用一个公共密钥加密时：

1. 一个新的随机会话密钥会被生成。

2. 它被用公共密钥加密并且放入到会话密钥包中。

在两种情况下，要被加密的数据按下列步骤被处理：

1. 可选的数据操纵：压缩、转换成 UTF-8 或者行末转换。

2. 数据会被加上一个随机字节的块作为前缀。这等效于使用一个随机 IV。

3. 追加一个随机前缀和数据的 SHA1 哈希。

4. 所有这些都用会话密钥加密并且放在数据包中。

F.25.3.1. pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

使用一个对称 PGP 密钥 psw加密data。options参数可以包含下文所述的选项设置。

F.25.3.2. pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) 返回 text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

解密一个用对称密钥加密过的 PGP 消息。

不允许使用pgp_sym_decrypt解密bytea数据。这是为了避免输出非法的字符数据。使用pgp_sym_decrypt_bytea解密原始文本数据是好的。

options参数可以包含下文所述的选项设置。

F.25.3.3. pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

用一个公共 PGP 密钥 key加密data。给这个函数一个私钥会产生一个错误。

options参数可以包含下文所述的选项设置。

F.25.3.4. pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) 返回 text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]])
 returns bytea
```

解密一个公共密钥加密的消息。key必须是对应于用来加密的公钥的私钥。如果私钥是用口令保护的，你必须在psw中给出该口令。如果没有口令，但你想要指定选项，你需要给出一个空口令。

不允许使用pgp_pub_decrypt解密bytea数据。这是为了避免输出非法的字符数据。使用pgp_pub_decrypt_bytea解密原始文本数据是好的。

options参数可以包含下文所述的选项设置。

F.25.3.5. pgp_key_id()

pgp_key_id(bytea) 返回 text

pgp_key_id抽取一个 PGP 公钥或私钥的密钥 ID。或者如果给定了一个加密过的消息，它给出一个用来加密数据的密钥 ID。

它能够返回 2 个特殊密钥 ID：

- SYMKEY

  该消息是用一个对称密钥加密的。

- ANYKEY

  该消息是用公钥加密的，但是密钥 ID 已经被移除。这意味着你将需要尝试你所有的密钥来看看哪个能解密该消息。pgcrypto本身不产生这样的消息。

注意不同的密钥可能具有相同的 ID。这很少见但是是一种正常事件。客户端应用则应该尝试用每一个去解密，看看哪个合适 — 像处理ANYKEY一样。

F.25.3.6. armor(), dearmor()

armor(data bytea [, keys text[], values text[]]) 返回 text
dearmor(data text) returns bytea

这些函数把二进制数据包装/解包成 PGP ASCII-armored 格式，其基本上是带有 CRC 和额外格式化的 Base64。

如果指定了keys和values数组，每一个 键/值对的 armored 格式上会增加一个armor header。两个 数组都必须是单一维度的，并且它们的长度必须相同。键和值不能包含任何非 ASCII 字符。

F.25.3.7. pgp_armor_headers

pgp_armor_headers(data text, key out text, value out text) returns setof record

pgp_armor_headers()从data中抽取 armor header。返回值是一个有两列的行集合，包括键和值。如果键或值 包含任何非-ASCII 字符，它们会被视作 UTF-8。

F.25.3.8. PGP 函数的选项

选项被命名为与 GnuPG 类似的形式。一个选项的值应该在一个等号后给出，各个选项之间用逗号分隔。例如：

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

除了convert-crlf之外所有这些选项只适用于加密函数。解密函数会从 PGP 数据中得到这些参数。

最有趣的选项可能是compress-algo和unicode-mode。其余的应该可以使用合理的默认值。

F.25.3.8.1. cipher-algo

要用哪个密码算法。

值：bf, aes128, aes192, aes256 (只用于 OpenSSL：3des, cast5)
默认：aes128
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.2. compress-algo

要使用哪种压缩算法。只有PostgreSQL编译时使用了 zlib 时才可用。

值：
  0 - 不压缩
  1 - ZIP 压缩
  2 - ZLIB 压缩 （= ZIP 外加元数据和块 CRC）
默认：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.3. compress-level

压缩多少。级别越高压缩得越小但是速度也越慢。0 表示禁用压缩。

值：0, 1-9
默认：6
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.4. convert-crlf

加密时是否把\n转换成\r\n以及解密时是否把\r\n转换成\n。RFC 4880 指定文本数据存储时应该使用\r\n换行。使用这个选项能够得到完全 RFC 兼容的行为。

值：0, 1
默认：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.25.3.8.5. disable-mdc

不用 SHA-1 保护数据。使用这个选项的唯一好的理由是实现与古董级别 PGP 产品的兼容，这些产品在受 SHA-1 保护的包被加入到 RFC 4880 之前就已经存在了。最近的 gnupg.org 和 pgp.com 软件能很好地支持它。

值：0，1
默认：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.6. sess-key

使用单独的会话密钥。公钥加密总是使用一个单独的会话密钥。这个选项是用于对称密钥加密的，对称密钥加密默认直接使用 S2K 密钥。

值：0，1
默认：0
适用于：pgp_sym_encrypt

F.25.3.8.7. s2k-mode

要使用哪一种 S2K 算法。

值：
  0 - 不用 salt。危险！
  1 - 用 salt 但是使用固定的迭代计数。
  3 - 可变的迭代计数。
默认：3
适用于：pgp_sym_encrypt

F.25.3.8.8. s2k-count

S2K 算法要使用的迭代次数。它必须是一个位于 1024 和 65011712 之间的值， 首尾两个值包括在内。

默认：65536 和 253952 之间的一个随机值
适用于：pgp_sym_encrypt，只能用于 s2k-mode=3

F.25.3.8.9. s2k-digest-algo

要在 S2K 计算中使用哪种摘要算法。

值：md5, sha1
默认：sha1
适用于：pgp_sym_encrypt

F.25.3.8.10. s2k-cipher-algo

要用哪种密码来加密独立的会话密钥。

值：bf, aes, aes128, aes192, aes256
默认：use cipher-algo
适用于：pgp_sym_encrypt

F.25.3.8.11. unicode-mode

是否把文本数据在数据库内部编码和 UTF-8 之间来回转换。如果你的数据库已经是 UTF-8，将不会转换，但是消息将被标记为 UTF-8。没有这个选项它将不会被标记。

值：0, 1
默认：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.9. 用 GnuPG 生成 PGP 密钥

要生成一个新密钥：

```
gpg --gen-key
```

更好的密钥类型是“DSA 和 Elgamal”。

对于 RSA 密钥，你必须创建仅用于签名的 DSA 或 RSA 密钥作为主控密钥，然后用gpg --edit-key增加一个 RSA 加密子密钥。

要列举密钥：

```
gpg --list-secret-keys
```

要以 ASCII-保护格式导出一个公钥：

```
gpg -a --export KEYID > public.key
```

要以 ASCII-保护格式导出一个私钥：

```
gpg -a --export-secret-keys KEYID > secret.key
```

在把这些密钥交给 PGP 函数之前，你需要对它们使用dearmor()。或者如果你能处理二进制数据，你可以从命令中去掉-a。

更多细节请参考man gpg、 The GNU Privacy Handbook[3]以及 https://www.gnupg.org/上的其他文档。

F.25.3.10. PGP 代码的限制

- 不支持签名。这也意味着它不检查加密子密钥是否属于主控密钥。
- 不支持加密密钥作为主控密钥。由于通常并不鼓励那种用法，这应该不是问题。
- 不支持多个子密钥。这可能看起来像一个问题，因为在实践中普遍需要多个子密钥。在另一方面，你不能把你的常规 GPG/PGP 密钥用于pgcrypto，而是创建一些新的密钥，因为使用场景相当不同。

F.25.4. 原始的加密函数

这些函数只在数据上运行一次加密，它们不具有 PGP 加密的任何先进特性。因此它们有一些主要的问题：

1. 它们直接把用户密钥用作加密密钥。

2. 它们不提供任何完整性检查来查看被加密数据是否被修改。

3. 它们希望用户自己管理所有加密参数，甚至是 IV。

4. 它们无法处理文本。

因此，在介绍了 PGP 加密后，我们不鼓励使用原始的加密函数。

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

使用type指定的密码方法加密/解密数据。type字符串的语法是：

```
algorithm [ - mode ] [ /pad: padding ]
```

其中algorithm是下列之一：

- bf — Blowfish
- aes — AES (Rijndael-128, -192 或 -256)

并且mode是下列之一：

[3] https://www.gnupg.org/gph/en/manual.html

- cbc — 下一个块依赖前一个（默认）
- ecb — 每一个块被独立加密（只用于测试）

并且padding是下列之一：

- pkcs — 数据可以是任意长度（默认）
- none — 数据必须是密码块尺寸的倍数

因此，例如这些是等效的：

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

在encrypt_iv和decrypt_iv中，iv参数是 CBC 模式的初始值，ECB 会忽略它。如果不是准确的块尺寸，它会被修剪或用零填充。在没有这个参数的函数中，它的值都被默认为零。

F.25.5. 随机数据函数

```
gen_random_bytes(count integer) returns bytea
```

返回count个密码上强壮的随机字节。一次最多可以抽取 1024 个字节。这是为了避免耗尽随机数发生池。

```
gen_random_uuid() 返回 uuid
```

返回一个版本 4 的（随机的）UUID。

F.25.6. 注解

F.25.6.1. 配置

pgcrypto会根据查找主 PostgreSQL configure脚本配置它自身。影响它的选项是--with-zlib以及--with-openssl。

在编译了 zlib 时，PGP 加密函数能够在加密前压缩数据。

在编译了 OpenSSL 时，会有更多可用算法。公钥加密函数也会更快，因为 OpenSSL 有优化得更好的 BIGNUM 函数。

表 F.19. 使用和不用 OpenSSL 的功能总结

功能	内建	使用 OpenSSL
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes
其他摘要算法	no	yes (注意 1)
Blowfish	yes	yes
AES	yes	yes
DES/3DES/CAST5	no	yes
原始加密	yes	yes
PGP 对称加密	yes	yes

功能	内建	使用 OpenSSL
PGP 公钥加密	yes	yes

注意：

1. OpenSSL 支持的任何摘要算法都是自动选取的。这对于使用密码来说是不可能的，因为需要被显式地支持。

F.25.6.2. NULL 处理

按照 SQL 中的标准，只要任何参数是 NULL， 所有的函数都会返回 NULL。在不当使用时这可能会导致安全风险。

F.25.6.3. 安全性限制

所有pgcrypto函数都在数据库服务器内部运行。这意味着在pgcrypto和客户端应用之间移动的所有数据和口令都是明文。因此，你必须：

1. 本地连接或者使用 SSL 连接。

2. 信任系统管理员和数据库管理员。

如果你不能这样做，那么最好在客户端应用中进行加密。

该实现无法抵抗 侧信道攻击[4]。例如，一个pgcrypto解密函 数完成所需的时间是随着密文尺寸变化的。

F.25.6.4. 有益的读物

- https://www.gnupg.org/gph/en/manual.html

  The GNU Privacy Handbook.

- http://www.openwall.com/crypt/

  描述 crypt-blowfish 算法。

- http://www.iusmentis.com/security/passphrasefaq/

  如何选取一个好的口令。

- http://world.std.com/~reinhold/diceware.html

  选择口令的有趣的想法。

- http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

  描述好的和不好的加密。

F.25.6.5. 技术性参考

- https://tools.ietf.org/html/rfc4880

  OpenPGP 消息格式。

- https://tools.ietf.org/html/rfc1321

  MD5 消息摘要算法。

- https://tools.ietf.org/html/rfc2104

[4] https://en.wikipedia.org/wiki/Side-channel_attack

HMAC：用于消息认证的钥控哈希。

- https://www.usenix.org/legacy/events/usenix99/provos.html

  crypt-des、crypt-md5 以及 bcrypt 算法的比较。

- https://en.wikipedia.org/wiki/Fortuna_(PRNG)

  Fortuna CSPRNG 的描述。

- http://jlcooke.ca/random/

  Linux 的 Jean-Luc Cooke Fortuna-based /dev/random驱动。

F.25.7. 作者

Marko Kreen <markokr@gmail.com>

pgcrypto使用了来自下列源码的代码：

算法	作者	源码起源
DES crypt	David Burren 等	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish cipher	Simon Tatham	PuTTY
Rijndael cipher	Brian Gladman	OpenBSD sys/crypto
MD5 hash and SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

pg_freespacemap模块提供了一种方法来检查空闲空间映射（FSM）。它提供了一个称为pg_freespace的函数，或者准确地说是两个重载的函数。这些函数显示空闲空间映射中为一个给定页面所记录的值，或者显示关系中所有页面的记录值。

默认情况下，使用仅限于超级用户和pg_stat_scan_tables 角色的成员。可以使用GRANT给其他人授予访问权限。

F.26.1. 函数

pg_freespace(rel regclass IN, blkno bigint IN) 返回 int2

根据 FSM，返回由blkno指定的关系页面上的空闲空间总量。

pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)

根据 FSM，显示关系的每个页面上的空闲空间总量。一个(blkno bigint, avail int2)元组的集合会被返回，每一个元组对应关系中的一个页面。

存储在空闲空间映射中的值不准确。它们被圆整到BLCKSZ的 1/256（对于默认的BLCKSZ是 32 字节），并且在元组被插入和更新时它们不会被实时更新。

对于索引，被跟踪的是整个没有使用的页面，而不是页面中的空闲空间。因此，这些值可能没有意义，只是表示一个页面是满的还是空的。

> **注意**
>
> 在版本 8.4 中接口已被更改，以反映在同一个版本中新引入的 FSM 实现。

F.26.2. 样例输出

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.26.3. 作者

最初的版本由 Mark Kirkwood <markir@paradise.net.nz>完成。在版本 8.4 中由 Heikki Linnakangas <heikki@enterprisedb.com>重写以适应新的 FSM 实现。

F.27. pg_prewarm

pg_prewarm模块提供一种方便的方法把关系 数据载入到操作系统缓冲区或者 PostgreSQL缓冲区。可以使用pg_prewarm函数手工执行预热，或者通过在shared_preload_libraries中包括pg_prewarm来自动执行预热。在后一种情况中，系统将运行一个后台工作者，它会周期性地把共享内存中的内容记录在一个名为autoprewarm.blocks的文件中，并且在重新启动后用两个后台工作者重新载入那些块。

F.27.1. 函数

pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8

第一个参数是要预热的关系。第二个参数是要使用的预热方法，下文将会 进一步讨论。第三个参数是要被预热的关系分叉，通常是main。 第四个参数是要预热的第一个块号（NULL也被接受，它等同于 零）。第五个参数是要预热的最后一个块号（NULL表示一直 预热到关系的最后一个块）。返回值是被预热的块数。

有三种可用的预热方法。prefetch会向操作系统发出异步 预取请求（如果支持异步预取），不支持异步预取则抛出一个错误。 read会读取要求范围的块。与prefetch 不同，它是同步的并且在所有平台上都被支持，但是可能较慢。 buffer会把要求范围的块读入道数据库的缓冲区。

注意使用任意一种方法尝试预热比能缓存的数量更多的块 — 使用 prefetch或者read（由OS）或者使用 buffer（由PostgreSQL ） — 将很可能导致高编号块被读入时把低编号的块从缓冲区中逐出的情况。 被预热的数据也不享受对缓冲区替换的特别保护，因此其他系统活动可能会在刚刚 被预热的块被读入后很快就将它们逐出。反过来，预热也可能把其他数据逐出缓存。 由于这些原因，预热通常在启动时最有用，那时缓冲大部分都为空。

autoprewarm_start_worker() RETURNS void

启动主要的autoprewarm工作者。这通常将会自动发生，但是如果没有在服务器启动时配置自动预热并且用户希望在稍晚的时候启动该工作者，这个函数就能发挥作用。

autoprewarm_dump_now() RETURNS int8

立即更新autoprewarm.blocks。如果autoprewarm工作者没有运行但用户希望它在下一次重启后运行，则这个函数会很有用。返回值是写入到autoprewarm.blocks中的记录数。

F.27.2. 配置参数

pg_prewarm.autoprewarm (boolean)

控制服务器是否应该运行autoprewarm工作者。默认这个参数为on。这个参数只能在服务器启动时设置。

pg_prewarm.autoprewarm_interval (int)

这是更新autoprewarm.blocks的间隔。默认是300秒。如果被设置为0，该文件将不会以常规的间隔方式转储，而是只在服务器关闭时转储。

F.27.3. 作者

Robert Haas <rhaas@postgresql.org>

F.28. pgrowlocks

pgrowlocks模块提供了一个函数来显示一个指定表的行锁定信息。

默认情况下，使用仅限于超级用户、pg_stat_scan_tables 角色的成员和在该表上拥有SELECT权限的用户。

F.28.1. 概述

pgrowlocks(text) 返回 setof record

参数是一个表的名称。结果是一个记录集合，其中每一行对应表中一个被锁定的行。输出列如表 F.20所示。

表 F.20.pgrowlocks 输出列

名称	类型	描述
locked_row	tid	被锁定行的元组 ID（TID）
locker	xid	持锁者的事务 ID，如果是多事务则为多事务 ID
multi	boolean	如果持锁者是一个多事务，则为真
xids	xid[]	持锁者的事务 ID（如果是多事务则多于一个）
modes	text[]	持锁者的锁模式（如果是多事务则多于一个），是一个Key Share、Share、For No Key Update、No Key Update、For Update、Update组成的数组。
pids	integer[]	锁定后端的进程 ID（如果是多事务则多于一个）

pgrowlocks会为目标表加AccessShareLock并且一个一个读取每一行来收集行的锁定信息。这对于一个大表不是很快。注意：

1. 如果表被其他人整体加上了排他锁，pgrowlocks将被阻塞。

2. pgrowlocks不保证能产生一个自我一致的快照。在它执行期间，有可能加上一个新行锁，也有可能有旧行锁被释放。

pgrowlocks不显示被锁定行的内容。如果你想同时查看行内容，你可以这样做：

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

不过要注意，这样一个查询将非常低效。

F.28.2. 样例输出

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.28.3. 作者

Tatsuo Ishii

F.29. pg_stat_statements

pg_stat_statements模块提供一种方法追踪一个服务器所执行的所有 SQL 语句的执行统计信息。

该模块必须通过在postgresql.conf的shared_preload_libraries中增加pg_stat_statements来载入，因为它需要额外的共享内存。这意味着增加或移除该模块需要一次服务器重启。

当pg_stat_statements被载入时，它会跟踪该服务器 的所有数据库的统计信息。该模块提供了一个视图 pg_stat_statements以及函数pg_stat_statements_reset和pg_stat_statements用于访问和操纵这些统计信息。这些视图 和函数不是全局可用的，但是可以用CREATE EXTENSION pg_stat_statements 为特定数据库启用它们。

F.29.1. pg_stat_statements视图

由该模块收集的统计信息可以通过一个名为 pg_stat_statements的视图使用。这个视图为每 一个可区分的数据库 ID、用户 ID 和查询 ID（最多到该模块可以追 踪的可区分语句的数量）的组合都包含一行。该视图的列如 表 F.2中所示。

表 F.21.pg_stat_statements列

名称	类型	引用	描述
userid	oid	pg_authid.oid	执行该语句的用户的 OID
dbid	oid	pg_database.oid	在其中执行该语句的数据库的 OID
queryid	bigint		内部哈希码，从语句的解析树计算得来
query	text		语句的文本形式
calls	bigint		被执行的次数
total_time	double precision		在该语句中花费的总时间，以毫秒计
min_time	double precision		在该语句中花费的最小时间，以毫秒计
max_time	double precision		在该语句中花费的最大时间，以毫秒计
mean_time	double precision		在该语句中花费的平均时间，以毫秒计
stddev_time	double precision		在该语句中花费时间的总体标准偏差，以毫秒计
rows	bigint		该语句检索或影响的行总数
shared_blks_hit	bigint		该语句造成的共享块缓冲命中总数
shared_blks_read	bigint		该语句读取的共享块的总数
shared_blks_dirtied	bigint		该语句弄脏的共享块的总数
shared_blks_written	bigint		该语句写入的共享块的总数
local_blks_hit	bigint		该语句造成的本地块缓冲命中总数

名称	类型	引用	描述
local_blks_read	bigint		该语句读取的本地块的总数
local_blks_dirtied	bigint		该语句弄脏的本地块的总数
local_blks_written	bigint		该语句写入的本地块的总数
temp_blks_read	bigint		该语句读取的临时块的总数
temp_blks_written	bigint		该语句写入的临时块的总数
blk_read_time	double precision		该语句花在读取块上的总时间，以毫秒计（如果track_io_timing被启用，否则为零）
blk_write_time	double precision		该语句花在写入块上的总时间，以毫秒计（如果track_io_timing被启用，否则为零）

由于安全性原因，只有超级用户和pg_read_all_stats 角色的成员被允许看到其他用户执行的查询 的 SQL 文本或者queryid。 不过，如果该视图被安装在其他用户的数据库中，那么他们就能够看见统 计信息。

只要可规划的查询（即SELECT、INSERT、UPDATE以及DELETE）根据一种内部哈希计算具有相同的查询结构，它们就会被组合到一个单一的pg_stat_statements项。通常，对于这里的目的，如果两个查询除了查询中的文本常量值之外在语义上等效，它们将会被认为是相同的。不过，功能性命令（即所有其他命令）会严格地以它们的文本查询字符串为基础进行比较。

当为了把一个查询与其他查询匹配，常数值会被忽略， 在pg_stat_statements显示中它会被一个参数符号， 比如$1所替换。查询文本的剩余部分就是具有与该pg_stat_statements项相关的特定queryid哈希值的第一个查询的文本。

在某些情况中，具有明显不同文本的查询可能会被融合到一个单一的pg_stat_statements项。通常这只会发生在语义等价的查询身上，但是也有很小的机会因为哈希碰撞的原因导致无关的查询被融合到一个项中（不过，对于属于不同用户或数据库的查询来说不会发生这种情况）。

由于queryid哈希值是根据查询被解析和分析后的表达计算的，对立的情况也可能存在：如果具有相同文本的查询由于参数（如不同的search_path设置）的原因而具有不同的含义，它们就可能作为不同的项存在。

pg_stat_statements的使用者可能希望使用 queryid（也许会与dbid和userid组合）作为一个项比查询文本更稳定和可靠的标识符。但是，有一点很重要的是，对于queryid哈希值稳定性只有有限的保障。因为该标识符是从解析分析后的树得来的，它的值是以这种形式出现的内部对象标识符的函数。这有一些违背直觉的含义。例如，如果有两个查询引用了同一个表，但是该表在两次查询之间被删除并且重建，显然这两个查询是完全一致的，但是pg_stat_statements将把它们认为是不同的。哈希处理也对机器架构以及平台的其他方面的差别很敏感。更进一步，认为PostgreSQL的不同主版本之间queryid将会保持稳定是不安全的。

根据经验，只有在底层服务器版本以及目录元数据细节保持完全相同时，queryid值才能被假定为稳定并且可比。两台参与到基于物理 WAL 重放的复制中的服务器会对相同的查询给出一样的queryid值。但是，逻辑复制模式并不保证在所有相关细节上都保持完全一样的复

制，因此在逻辑复制机之间计算代价时，queryid并非是一个有用的标识符。如果有疑问，推荐直接进行测试。

代表性查询文本中用于替换常量的参数符号从原始查询文本中最高的 $n参数之后的下一个数字开始， 如果没有则为$1。值得注意的是，在某些情况下， 可能存在影响编号的隐藏参数符号。例如，PL/pgSQL 使用隐藏参数符号将函数局部变量的值插入到查询中，以便像SELECT i + 1 INTO j的PL/pgSQL 语句将具有像SELECT i + $2这样的代表性文本。

有代表性的查询文本被保存在一个外部磁盘文件中，并且不会消耗共享内存。 因此，即便是很长的查询文本也能被成功的存储下来。不过，如果累积了很多 长的查询文本，该外部文件也会增长到很大。作为一种恢复方法，如果这样的 情况发生，pg_stat_statements可能会选择丢弃这些查询文本， 于是pg_stat_statements视图中的所有现有项将会显示空的query域，不过与每个queryid相关联的 统计信息会被保留下来。如果发生这种情况，可以考虑减小 pg_stat_statements.max来防止复发。

F.29.2. 函数

pg_stat_statements_reset() 返回 void

pg_stat_statements_reset抛弃目前由pg_stat_statements收集的所有统计信息。默认情况下，这个函数只能被超级用户执行。

pg_stat_statements(showtext boolean) returns setof record

pg_stat_statements视图按照一个也叫 pg_stat_statements的函数来定义。客户端可以直接调用 pg_stat_statements函数，并且通过指定 showtext := false来忽略查询文本（即，对应于视图的 query列的OUT参数将返回空值）。 这个特性是为了支持不想重复接收长度不定的查询文本的外部工具而设计的。 这类工具可以转而自行缓存第一个观察到的查询文本，因为这就是 pg_stat_statements自己所做的全部工作，并且只在需要的时候检索查询文本。因为服务器会把查询文本存储在一个文件中，这种方法可 以降低重复检查pg_stat_statements数据的 物理 I/O。

F.29.3. 配置参数

pg_stat_statements.max (integer)

pg_stat_statements.max是由该模块跟踪的语句的最大数目（即pg_stat_statements视图中行的最大数量）。如果观测到的可区分的语句超过这个数量，最少被执行的语句的信息将会被丢弃。默认值为 5000。这个参数只能在服务器启动时设置。

pg_stat_statements.track (enum)

pg_stat_statements.track控制哪些语句会被该模块计数。指定top可以跟踪顶层语句（那些直接由客户端发出的语句），指定all还可以跟踪嵌套的语句（例如在函数中调用的语句），指定none可以禁用语句统计信息收集。默认值是top。 只有超级用户能够改变这个设置。

pg_stat_statements.track_utility (boolean)

pg_stat_statements.track_utility控制该模块是否会跟踪工具命令。工具命令是除了SELECT、INSERT、 UPDATE和DELETE之外所有的其他命令。默认值是on。 只有超级用户能够改变这个设置。

pg_stat_statements.save (boolean)

pg_stat_statements.save指定是否在服务器关闭之后还保存语句统计信息。如果被设置为off，那么关闭后不保存统计信息并且在服务器启动时也不会重新载入统计信息。默认值为on。这个参数只能在postgresql.conf文件中或者在服务器命令行上设置。

该模块要求与pg_stat_statements.max成比例的额外共享内存。注意只要该模块被载入就会消耗这么多的内存，即便pg_stat_statements.track被设置为none。

这些参数必须在postgresql.conf中设置。典型的用法可能是：

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.29.4. 示例输出

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD
 1 ]-----------------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 $2;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD
 2 ]-----------------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
 $2;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD
 3 ]-----------------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 310.624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD
 4 ]-----------------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
 $2;
calls       | 3000
total_time  | 271.741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD
 5 ]-----------------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
```

```
calls       | 1
total_time  | 81.42
rows        | 0
hit_percent | 34.4947735191637631
```

F.29.5. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>。Peter Geoghegan <peter@2ndquadrant.com>为它加入了查询正规化的功能。

F.30. pgstattuple

pgstattuple模块提供多种函数来获得元组层的统计信息。

由于这些函数返回详细的页面级信息，因此默认访问权限是受限制的。 默认情况下，只有角色pg_stat_scan_tables具有EXECUTE特权。 超级用户当然可以绕过这个限制。扩展程序安装后,用户可以发送GRANT命令来更改函数的权限以允许其他用户来执行它们。但是，最好将这些用户添加到pg_stat_scan_tables角色。

F.30.1. 函数

pgstattuple(regclass) 返回 record

pgstattuple返回一个关系的物理长度、“死亡”元组的百分比以及其他信息。这可以帮助用户决定是否需要清理。参数是目标关系的名称（可以有选择地用模式限定）或者OID。例如：

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

表 F.22中描述了输出列。

表 F.22.pgstattuple 输出列

列	类型	描述
table_len	bigint	物理关系长度，以字节计
tuple_count	bigint	存活元组的数量
tuple_len	bigint	存活元组的总长度，以字节计
tuple_percent	float8	存活元组的百分比
dead_tuple_count	bigint	死亡元组的数量
dead_tuple_len	bigint	死亡元组的总长度，以字节计
dead_tuple_percent	float8	死亡元组的百分比

列	类型	描述
free_space	bigint	空闲空间总量，以字节计
free_percent	float8	空闲空间的百分比

> **注意**
>
> table_len将总是比tuple_len、 dead_tuple_len和free_space的和要大。 不同之处在于固定的页面开销，每页指向元组的指针表和填充以确保元组正确对齐。

pgstattuple只要求在关系上的一个读锁。因此结果不能反映一个即时快照，并发更新将影响结果。

如果HeapTupleSatisfiesDirty返回假，pgstattuple就判定一个元组是“死亡的”。

pgstattuple(text) 返回 record

与pgstattuple(regclass)相同，只不过通过 TEXT 指定目标关系。这个函数只是为了向后兼容而保留，在未来的发布中将会被废除。

pgstatindex(regclass) 返回 record

pgstatindex返回一个记录显示有关一个 B-树索引的信息。例如：

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

输出列是：

列	类型	描述
version	integer	B-树 版本号
tree_level	integer	根页的树层次
index_size	bigint	以字节计的索引总尺寸
root_block_no	bigint	根页的位置（如果没有则为零）
internal_pages	bigint	“内部”（上层）页面的数量
leaf_pages	bigint	叶子页的数量
empty_pages	bigint	空页的数量
deleted_pages	bigint	删除页的数量
avg_leaf_density	float8	叶子页的平均密度
leaf_fragmentation	float8	叶子页碎片

报告的index_size通常对应于internal_pages + leaf_pages + empty_pages + deleted_pages加一，因为它还包括索引的元页。

对于pgstattuple，结果是一页一页累计的并且不要期望结果会表示整个索引的一个即时快照。

pgstatindex(text) returns record

与pgstatindex(regclass)相同，只不过通过 TEXT 指定目标索引。这个函数只是为了向后兼容而保留，在未来的某个发布中将会被废除。

pgstatginindex(regclass) 返回 record

pgstatginindex返回一个记录显示有关一个 GIN 索引的信息。例如：

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

输出列是：

列	类型	描述
version	integer	GIN 版本号
pending_pages	integer	待处理列表中的页面数
pending_tuples	bigint	待处理列表中的元组数

pgstathashindex(regclass) returns record

pgstathashindex返回一个显示HASH索引信息的记录。例如：

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
version        | 4
bucket_pages   | 33081
overflow_pages | 0
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

输出字段是：

字段	类型	描述
version	integer	HASH版本号
bucket_pages	bigint	存储桶页面的数量
overflow_pages	bigint	溢出页面的数量
bitmap_pages	bigint	位图页数
unused_pages	bigint	未使用页面的数量
live_items	bigint	活元组的数量
dead_tuples	bigint	死元组的数量
free_percent	float	自由空间的百分比

pg_relpages(regclass) 返回 bigint

pg_relpages返回关系中的页面数。

pg_relpages(text) returns bigint

与pg_relpages(regclass)相同，只不过用 TEXT 来 指定目标关系。这个函数只是为了向后兼容而保留，在未来的某个发布中将会被废除。

pgstattuple_approx(regclass) returns record

pgstattuple_approx是pgstattuple的一个更加快速的替代品，它返回近似的结果。参数是目标关系的 OID 或者名称。例如：

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

输出列在表 F.28中描述。

鉴于pgstattuple总是执行全表扫描并且返回存活和死亡元组的准确计数、尺寸和空闲空间，pgstattuple_approx尝试避免全表扫描并且返回死亡元组的准确统计信息，以及存活元组和空闲空间的近似数量及尺寸。

这个函数通过根据可见性映射跳过只包含可见元组的页面来实现这一目的（如果一个页面对应的 VM 位被设置，那么就说明它不含有死亡元组）。对于这样的额页面，它会从空闲空间映射中得到空闲空间值，并且假定该页面上的剩余空间由存活元组占据。

对于不能被跳过的页面，它会扫描每个元组，在合适的计数器中记录它的存在以及尺寸，并且统计该页面上的空闲空间。最后，它会基于已扫描的页面和元组数量来估计存活元组的总数（采用与 VACUUM 估计 pg_class.reltuples 时相同的方法）。

表 F.23.pgstattuple_approx输出列

列	类型	描述
table_len	bigint	以字节计的物理关系长度（准确）
scanned_percent	float8	已扫描表的百分比
approx_tuple_count	bigint	存活元组的数量（估计）
approx_tuple_len	bigint	以字节计的存活元组总长度（估计）
approx_tuple_percent	float8	存活元组的百分比
dead_tuple_count	bigint	死亡元组的数量（准确）
dead_tuple_len	bigint	以字节计的死亡元组总长度（准确）
dead_tuple_percent	float8	死亡元组的百分比

列	类型	描述
approx_free_space	bigint	以字节计的总空闲空间（估计）
approx_free_percent	float8	空闲空间的百分比

在上述的输出中，空闲空间数字可能不完全匹配pgstattuple的输出，这是因为空闲空间映射会给出一个准确的数字，但是这个数字不能保证是一个准确的字节数。

F.30.2. 作者

Tatsuo Ishii, Satoshi Nagayasu 和 Abhijit Menon-Sen

F.31. pg_trgm

pg_trgm模块提供用于决定基于 trigram 匹配的字母数字文本相似度的函数和操作符，以及支持快速搜索相似字符串的索引操作符类。

F.31.1. Trigram（或者 Trigraph）概念

一个 trigram 是从一个字符串中取出的由三个连续字符组成的组。我们可以通过对两个字符串之间共享的 trigram 计数来度量它们的相似度。这种简单的思想已经成为在很多自然语言中度量词相似度的有效方法。

> **注意**
>
> 在从一个字符串中提取 trigram 时，pg_trgm会忽略非词字符（非字母数字）。在决定字符串中所含的 trigram 集合时，每一个词被认为具有两个空格前缀和一个空格后缀。例如，字符串“cat”中的 trigram 集合是： “ c”、 “ ca”、 “cat”以及 “at ”。 字符串“foo|bar”中的 trigram 集合是： “ f”、 “ fo”、 “foo”、 “oo ”、 “ b”、 “ ba”、 “bar”以及 “ar ”。

F.31.2. 函数和操作符

pg_trgm模块所提供的函数如表 F.24中所示，操作符则显示在表 F.25中。

表 F.24.pg_trgm函数

函数	返回值	描述
similarity(text, text)	real	返回一个数字指示两个参数有多相似。该结果的范围是0（指示两个字符串完全不相似）到 1（指示两个字符串完全一样）。
show_trgm(text)	text[]	返回一个给定字符串中所有的 trigram 组成的一个数组（实际上除了调试很少有用）。
word_similarity(text, text)	real	返回一个数字表示第一个字符串中的trigram集合与第二个字符串中trigram的有序集中任何连续部分的最大相似度。详情请见下文的解释。

函数	返回值	描述
strict_word_similarity(text, text)	real	与word_similarity(text, text)相同，但是强制连续部分的边界与词边界相匹配。由于我们没有跨词的trigram，这个函数实际上返回第一个字符串和第二个字符串任意连续部分的相似度。
show_limit()	real	返回%操作符使用的当前相似度阈值。例如，这设定两个词被认为足够相似时，它们之间应满足的最小相似度（已废弃）。
set_limit(real)	real	设定%操作符使用的当前相似度阈值。该阈值必须介于 0 和 1 之间（默认为 0.3）。返回传递进来的同一个值（已废弃）。

考虑下面的例子：

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

在第一个字符串中，trigram集合是{" w"," wo","wor","ord","rd "}。在第二个字符串中，trigram的有序集是{" t"," tw","two","wo "," w"," wo","wor","ord","rds","ds "}。在第二个字符串中最相似的trigram有序集的部分是{" w"," wo","wor","ord"}，并且相似度是0.8。

这个函数返回的值可以大概地理解为第一个字符串和第二个字符串任意子串的最大相似度。不过，这个函数不会对该部分的边界加入填充。因此，除了失配的词边界之外，第二个字符串中存在的额外字符的数目没有被考虑。

同时，strict_word_similarity(text, text)在第二个字符串中选择一个由词构成的部分。在上面的例子中，strict_word_similarity(text, text)会选择单个词'words'形成的部分，其trigram集合为{" w"," wo","wor","ord","rds","ds "}。

```
# SELECT strict_word_similarity('word', 'two words'), similarity('word',
 'words');
 strict_word_similarity | similarity
------------------------+------------
                0.571429 |   0.571429
(1 row)
```

因此，strict_word_similarity(text, text)函数对于计算整个词的相似度有用，而word_similarity(text, text)更适合于计算词的部分相似度。

表 F.25.pg_trgm操作符

操作符	返回值	描述
text % text	boolean	如果参数具有超过pg_trgm.similarity_threshold设

操作符	返回值	描述
		置的当前相似度阈值的相似度，则返回true。
text <% text	boolean	如果第一个参数中的trigram集合与第二个参数中有序trigram集合的一个连续部分之间的相似度超过pg_trgm.word_similarity_threshold参数设置的当前词相似度阈值，则返回true。
text %> text	boolean	<%操作符的交换子。
text <<% text	boolean	如果第二个参数有有序trigram集合的一个连续部分匹配词边界，并且其与第一个参数的trigram集合的相似度超过pg_trgm.strict_word_similarity_threshol数设置的当前严格词相似度阈值，则返回true。
text %>> text	boolean	<<%操作符的交换子。
text <-> text	real	返回参数之间的“距离”，即 1 减去similarity()值。
text <<-> text	real	返回参数之间的“距离”，它是 1 减去word_similarity()的值。
text <->> text	real	<<->操作符的交换子。
text <<<-> text	real	返回参数之间的“距离”，也就是1减去strict_word_similarity()的值。
text <->>> text	real	<<<->操作符的交换子。

F.31.3. GUC 参数

pg_trgm.similarity_threshold (real)

设置%操作符使用的当前相似度阈值。该阈值必须位于 0 和 1 之间（默认是 0.3）。

pg_trgm.word_similarity_threshold (real)

设置<%和%>操作符使用的当前词相似度阈值。该阈值必须位于 0 和 1 之间（默认是 0.6）。

F.31.4. 索引支持

pg_trgm模块提供了 GiST 和 GIN 索引操作符类，这允许你在一个文本列上创建索引用于快速相似度搜索的目的。这些索引类型支持上述的相似度操作符，并且额外支持基于 trigram 的索引搜索用于LIKE、ILIKE、~和~*查询（这些索引不支持等值或简单比较操作符，因此你可能还需要一个常规的 B-树索引）。

例子：

```
CREATE TABLE test_trgm (t text);
```

```
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

或

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

此时，你将有一个t列上的索引，你可以用它进行相似度搜索。一个典型的查询是

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

这将返回在文本列中与word足够相似的所有值，按最佳匹配到最差匹配的方式排序。索引将被用来让这种搜索变快，即使在一个非常大的数据集上。

上述查询的一种变体是

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

这能够用 GiST 索引有效地实现，但是用 GIN 索引无法做到。当只想要少数最接近的匹配时，这通常会比第一种形式更好。

也可以把一个t列上的索引用于词相似度或者严格词相似度。典型的查询是：

```
SELECT t, word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <% t
  ORDER BY sml DESC, t;
```

和

```
SELECT t, strict_word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <<% t
  ORDER BY sml DESC, t;
```

这将返回文本列中符合条件的所有值：这些值在其对应的有序trigram集中有一个连续部分与word的trigram集合足够相似，这些值会按照最好匹配到最差匹配的顺序排列。即便在非常大的数据集上，索引也将使得这一操作的速度够快。

上述查询可能的变体有：

```
SELECT t, 'word' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

和

```
SELECT t, 'word' <<<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

这可以用 GiST 索引很高效地实现，但是用 GIN 索引不行。

从PostgreSQL 9.1 中开始，这些索引类型也支持用于LIKE和ILIKE的索引搜索，例如

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

该索引搜索通过从搜索字符串中抽取 trigram 并且在索引中查找它们来工作。搜索字符串中有更多 trigram，索引搜索的效率更高。不像基于 B-树的搜索，搜索字符串不需要是左锚定的。

从PostgreSQL 9.3 中开始，这些索引类型也支持用于正则表达式匹配（~和~*操作符）的索引搜索，例如

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

该索引搜索通过从正则表达式中抽取 trigram 并且在索引中查找它们来工作。正则表达式中能抽取出更多 trigram，索引搜索的效率更高。不像基于 B-树的搜索，搜索字符串不需要是左锚定的。

对于LIKE和正则表达式搜索，记住没有可抽取 trigram 的模式将退化成一个全索引扫描。

GiST 和 GIN 索引之间的选择依赖于 GiST 和 GIN 的相对性能特性，这在其他地方讨论。

F.31.5. 文本搜索集成

在与一个全文索引联合使用时，trigram 匹配是一种非常有用的工具。特别是它能有助于识别拼写错误的输入词，这些词直接用全文搜索机制是不会被匹配的。

第一步是生成一个包含文档中所有唯一词的辅助表：

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

其中documents是一个具有我们希望搜索的文本域bodytext的表。对to_tsvector函数使用simple配置而不是使用语言相关的配置的原因是，我们想要一个原始（没有去掉词根的）词的列表。

接下来，在词列上创建一个 trigram 索引：

```
CREATE INDEX words_idx ON words USING GIN(word gin_trgm_ops);
```

现在，类似于前面例子的一个SELECT查询可以被用来为用户搜索术语中的拼写不当的词建议拼写。要求被选择的词也与拼写不当的词具有相似的长度是一种有用的额外测试。

> **注意**
>
> 由于words表已经被生成为一个单独的、静态的表，它将需要被定期地重新生成，这样它能合理地与文档集合保持一致。但是要求它完全与文档集合同步通常是不必要的。

F.31.6. 参考

GiST 开发站点 http://www.sai.msu.su/~megera/postgres/gist/

Tsearch2 开发站点 http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.31.7. 作者

Oleg Bartunov <oleg@sai.msu.su>，俄罗斯莫斯科大学

Teodor Sigaev <teodor@sigaev.ru>，俄罗斯莫斯科 Delta-Soft 有限责任公司

Alexander Korotkov <a.korotkov@postgrespro.ru>，俄罗斯莫斯科，Postgres Professional

文档：Christopher Kings-Lynne

这个模块由俄罗斯莫斯科 Delta-Soft 有限责任公司赞助。

F.32. pg_visibility

pg_visibility模块提供了一种方式来检查一个表的可见性映射（VM）以及页级别的可见性信息。它还提供了函数来检查可见性映射的完整性以及强制重建可见性映射。

有三个不同的位被用来存储有关页级别可见性的信息。 可见性映射中的“全部可见”位表示一个关系的对应页面上的所有元组对每一个当前和未来事务都可见。 可见性映射中的“全部冻结”位表示该页上的每一个元组都被冻结， 也就是说直到在那个页面上对一个元组进行插入、更新、 删除或者锁定之前都不需要用 vacuum 对该页面进行修改。 页面头部的PD_ALL_VISIBLE位具有和可见性映射中“全部可见”位相同的含义， 但是它存储在数据页面本身中而不是存储在单独的数据结构中。这两个位通常是相互一致的， 但是有时在崩溃恢复后会出现页面的“全部可见”位被设置而可见性映射位被清除的情况， 由于在pg_visibility检查了可见性映射之后， 且在它检查数据页面之前发生了修改，报告的值也会不一致。 任何导致数据损坏的事件也可能导致这些位不一致。

显示有关PD_ALL_VISIBLE的信息的函数代价比那些查看可见性映射的函数要高很多，因为它们必须读取关系的数据块而不是只读取（小很多的）可见性映射。类似地，检查关系的数据块的函数也很昂贵。

F.32.1. 函数

pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean) returns record

为给定关系的给定块返回其在可见性映射中的“全部可见”和“全部冻结”位。

pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns record

为给定关系的给定块返回其在可见性映射中的“全部可见”和“全部冻结”位，外加块的PD_ALL_VISIBLE位。

pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean) returns setof record

为给定关系的每一块返回其在可见性映射中的“全部可见”和“全部冻结”位。

pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record

为给定关系的每一块返回其在可见性映射中的“全部可见”和“全部冻结”位，外加每一块的PD_ALL_VISIBLE位。

pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen OUT bigint) returns record

根据可见性映射返回关系中“全部可见”页面和“全部冻结”页面的数量。

pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid

返回存储在可见性映射中被标为“全部冻结”的页面中的非冻结元组的 TID。 如果这个函数返回一个非空的 TID 集合，则可见性映射已经损坏。

pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid

返回存储在可见性映射中标记为全部可见的页面中的非全部可见元组的TID。 如果此函数返回非空的一组TID，则可见性映射已损坏。

pg_truncate_visibility_map(relation regclass) returns void

为给定关系截断可见性映射。如果您认为该关系的可见性映射已损坏并希望强制重建它， 则该函数非常有用。在这个函数被执行后，在给定关系上进行的第一次VACUUM 将会扫描关系中的每一个页面并且重建可见性映射。（在完成之前， 查询会将可见性映射看做包含的全是零。）

默认情况下，这些函数只有超级用户和pg_stat_scan_tables角色的成员可以执行， 除了pg_truncate_visibility_map(relation regclass)只能由超级用户执行之外。

F.32.2. 作者

Robert Haas <rhaas@postgresql.org>

F.33. postgres_fdw

postgres_fdw模块提供了外部数据包装器postgres_fdw，它可以被用来访问存储在外部PostgreSQL服务器中的数据。

这个模块提供的功能大体上覆盖了较老的dblink模块的功能。但是postgres_fdw提供了更透明且更兼容标准的语法来访问远程表，并且可以在很多情况下给出更好的性能。

要使用postgres_fdw来为远程访问做准备：

1. 使用CREATE EXTENSION来安装postgres_fdw扩展。
2. 使用CREATE SERVER创建一个外部服务器对象，它用来表示你想连接的每一个远程数据库。指定除了user和password之外的连接信息作为该服务器对象的选项。
3. 使用CREATE USER MAPPING创建一个用户映射，每一个用户映射都代表你想允许一个数据库用户访问一个外部服务器。指定远程用户名和口令作为用户映射的user和password选项。
4. 为每一个你想访问的远程表使用CREATE FOREIGN TABLE或者IMPORT FOREIGN SCHEMA创建一个外部表。外部表的列必须匹配被引用的远程表。但是，如果你在外部表对象的选项中指定了正确的远程名称，你可以使用不同于远程表的表名和/或列名。

现在你只需要从一个外部表SELECT来访问存储在它的底层的远程表中的数据。你也可以使用INSERT、UPDATE或DELETE修改远程表（当然，你在你的用户映射中已经指定的远程用户必须具有做这些事情的权限）。

注意当前postgres_fdw缺少对于带ON CONFLICT DO UPDATE子句的INSERT语句的支持。不过，它支持ON CONFLICT DO NOTHING子句，已提供的唯一索引推断说明会被省略。

我们通常推荐一个外部表的列被声明为与被引用的远程表列完全相同的数据类型和排序规则（如果可用）。尽管postgres_fdw目前已经能够容忍在需要时执行数据类型转换，但是当类型或排序规则不匹配时可能会发生奇怪的语义异常，因为远程服务器解释WHERE子句时可能会与本地服务器有所不同。

注意一个外部表可以被声明比底层的远程表较少的列，或者使用一种不同的列序。与远程表的列匹配是通过名字而不是位置进行的。

F.33.1. postgres_fdw 的 FDW 选项

F.33.1.1. 连接选项

一个使用postgres_fdw外部数据包装器的外部服务器可以使用和libpq在连接字符串中能接受的选项，如第 34.1.2 所述，不过不允许这些选项：

- user和password（应该在用户映射中指定这些）
- client_encoding（这是自动从本地服务器编码设置）
- fallback_application_name（总是设置为postgres_fdw）

只有超级用户可以在不经过口令认证的情况下连接到外部服务器，因此应总是为属于非超级用户的用户映射指定password选项。

F.33.1.2. 对象名称选项

这些选项可以被用来控制使用在被发送到远程PostgreSQL服务器的 SQL 语句中使用的名称。当一个外部表被使用不同于底层远程表的名称创建时，就需要这些选项。

schema_name

这个选项给出用在远程服务器之上的外部表的模式名称，它可以为一个外部表指定。如果这个选项被忽略，该外部表的模式名称将被使用。

table_name

这个选项给出用在远程服务器上的外部表给出表名，它可以为一个外部表指定。如果这个选项被忽略，该外部表的名字将被使用。

column_name

这个选项给出用在远程服务器上列的列名，它可以为一个外部表的一个列指定。如果这个选项被忽略，该列的名字将被使用。

F.33.1.3. 代价估计选项

postgres_fdw通过在远程服务器上执行查询来检索远程数据，因此理想的扫描一个外部表的估计代价应该是在远程服务器上完成它的花销，外加一些通信开销。得到这样一个估计的最可靠的方法是询问远程服务器并加上一些通信开销 — 但是对于简单查询，不值得为获得一个代价估计而额外使用一次远程查询。因此postgres_fdw提供了下列选项来控制如何完成代价估计：

use_remote_estimate

这个选项控制postgres_fdw是否发出EXPLAIN命令来获得代价估计，它可以为一个外部表或一个外部服务器指定。一个外部表的设置会覆盖它的服务器的任何设置，但是只用于这个表。默认值是false。

fdw_startup_cost

这个选项是一个要被加到那个服务器上所有外部表扫描的估计启动代价的数字值。这表示建立一个连接、在远端解析和规查询的额外负荷等。默认值是100。

fdw_tuple_cost

这个选项是一个数字值，它被用作那个服务器上外部表扫描的每元组额外代价，它可以为一个外部服务器指定。这表示在服务器之间数据传输的额外负荷。你可以增加或减少这个数来反映到远程服务器更高或更低的网络延迟。默认值是0.01。

当use_remote_estimate为真时，postgres_fdw从远程服务器获得行计数和代价估计，然后在代价估计上加上fdw_startup_cost和fdw_tuple_cost。当use_remote_estimate为假时，postgres_fdw执行本地行计数和代价估计，并且接着在代价估计上加上fdw_startup_cost 和fdw_tuple_cost。这种本地估计不会很准确，除非有远程表统计数据的本地拷贝可用。在外部表上运行ANALYZE是更新本地统计数据的方法，这将执行远程表的一次扫描并接着计算和存储统计数据，就好像表在本地一样。保留本地统计数据可能是一种有用的方法来减少一个远程表的预查询规划负荷 — 但是如果远程表被频繁更新，本地统计数据将很快就被废弃。

F.33.1.4. 远程执行选项

默认情况下，只有使用了内建操作符和函数的WHERE子句才会被考虑在远程服务器上执行。涉及非内建函数的子句将会在取完行后在本地进行检查。如果这类函数在远程服务器上可用并且可以用来产生和本地执行时一样的结果，则可以通过将这种WHERE子句发送到远程执行来提高性能。可以用下面的选项控制这种行为：

extensions

这个选项是一个用逗号分隔的已安装的PostgreSQL扩展名称列表，这些扩展在本地和远程服务器上具有兼容的版本。属于一个该列表中扩展的 immutable 函数和操作符将被考虑转移到远程服务器上执行。这个选项只能为外部服务器指定，无法逐个表指定。

在使用extensions选项时，用户应该负责确保列出的扩展在本地和远程服务器上都存在且保持一致。否则，远程查询可能失败或者行为异常。

fetch_size

这个选项指定在每次获取行的操作中postgres_fdw应该得到的行数。可以为一个外部表或者外部服务器指定这个选项。在表上指定的选项将会覆盖在服务器级别上指定的选项。默认值为100。

F.33.1.5. 可更新性选项

默认情况下，所有使用postgres_fdw的外部表都被假定是可更新的。这可以使用下列选项覆盖：

updatable

这个选项控制postgres_fdw是否允许外部表被使用INSERT、UPDATE和DELETE命令更新。它可以为一个外部表或一个外部服务器指定。一个表级选项会覆盖一个服务器级选项。默认值是true。

当然，如果远程表实际上并非可更新的，将产生一个错误。这个选项的使用主要是允许在不查询远程服务器的情况下在本地抛出错误。但是要注意information_schema视图会根据这个选项的设置报告一个postgres_fdw外部表是可更新的（或者不可更新），而不需要远程服务器的任何检查。

F.33.1.6. 导入选项

postgres_fdw能使用IMPORT FOREIGN SCHEMA导入外部表定义。这个命令会在本地服务器上创建外部表定义，这个定义能匹配存在于远程服务器上的表或者视图。如果要被导入的远程表有用户自定义数据类型的列，本地服务器上也必须具有相同名称的兼容类型。

导入行为可以用下列选项自定义（在IMPORT FOREIGN SCHEMA命令中给出）：

import_collate

这个选项控制是否在从外部服务器导入的外部表定义中包括列的COLLATE选项。默认是true。如果远程服务器具有和本地服务器不同的排序规则名集合，可能需要关闭这个选项，在不同的操作系统上运行时很可能就是这样。

import_default

这个选项控制是否在从外部服务器导入的外部表定义中包括列的DEFAULT表达式。默认是false。如果启用这个选项，要当心在远程服务器和本地服务器上计算表达式的方式不同，nextval()常会导致这类问题。如果导入的默认值表达式使用了一个本地不存在的函数或者操作符，IMPORT将整个失败。

import_not_null

这个选项控制是否在从外部服务器导入的外部表定义中包括列的NOT NULL约束。默认是true。

注意除NOT NULL之外的约束将不会从远程表中导入。虽然PostgreSQL确实支持外部表上的CHECK约束，但不会自动导入它们，因为存在本地和远程服务器计算约束表达式方式不同的风险。CHECK约束中的任何这类不一致都可能导致查询优化中很难检测的错误。因此，如果你希望导入CHECK约束，你必须手工来做，并且你应该仔细地验证每一个这种约束的语义。有关处理外部表上CHECK约束的更多细节，请见CREATE FOREIGN TABLE。

自动排除作为其他表的分区的表或外部表。分区表被导入，除非它们是其他表的分区。 由于所有数据都可以通过作为分区层次根的分区表来访问， 所以这种方法应该允许访问所有数据而不创建额外的对象。

F.33.2. 连接管理

postgres_fdw在第一个使用关联到外部服务器的外部表的查询期间建立一个到外部服务器的连接。这个连接会被保持，并被重用于同一个会话中的后续查询。但是，如果使用了多个用户实体（用户映射）来访问外部服务器，会为每一个用户映射建立一个连接。

F.33.3. 事务管理

在一个引用外部服务器上任何远程表的查询期间，如果还没有根据当前的本地事务打开一个远程事务，postgres_fdw将在远程服务器上打开一个事务。当本地事务提交或中止时，远程事务也被提交或中止。保存点也相似地采用创建相应的远程保存点来管理。

当本地事务为SERIALIZABLE隔离级别时，远程事务使用SERIALIZABLE隔离级别；否则它使用REPEATABLE READ隔离级别。如果一个查询在远程服务器上执行多个表查询，这种选择保证它将为所有扫描得到快照一致的结果。一种后果是在单一事务中的后继查询将会看到来自远程服务器的相同数据，即便由于其他活动在远程服务器上发生了其他并发更新。如果本地事务使用SERIALIZABLE或REPEATABLE READ隔离级别，这种行为也是可以预期的，但是对于一个READ COMMITTED本地事务它是奇怪的。一个未来的PostgreSQL发布可能会修改这些规则。

F.33.4. 远程查询优化

postgres_fdw尝试优化远程查询来减少从外部服务器传来的数据量。这可以通过把查询的WHERE子句发送给远程服务器执行来完成，并且还可以不检索当前查询不需要的表列。为了降低查询被误执行的风险，除非WHERE子句使用的数据类型、操作符和函数都是内建的或者属于列在该外部服务器的extensions选项中的一个扩展，将不会把WHERE子句发送到远程服务器。这些子句中的操作符合函数也必须是IMMUTABLE。对于UPDATE或者DELETE查询， 如果没有不能发送给远程服务器的WHERE子句、 没有查询的本地连接、目标表上没有本地的行级BEFORE或AFTER触发器， 并且没有来自父视图的CHECK OPTION约束，postgres_fdw会尝试通过将整个查询发送给远程服务器来优化查询的执行。在UPDATE中，赋值给目标列的表达式只能使用内建数据类型、IMMUTABLE操作符或者IMMUTABLE操作符，这样能降低查询被误执行的风险。

当postgres_fdw碰到同一个外部服务器上的外部表之间的连接时，它会把整个连接发送给外部服务器，除非由于某些原因它认为逐个从每一个表取得行的效率更高或者涉及的表引用属于不同的用户映射。在发送JOIN子句时，它也会采取和上述WHERE子句相同的预防措施。

实际被发送到远程服务器执行的查询可以使用EXPLAIN VERBOSE来检查。

F.33.5. 远程查询执行环境

在postgres_fdw开启的远程会话中，search_path参数只被设置为pg_catalog，因此只有内建对象可以在无模式限定时可见。这对于postgres_fdw本身产生的查询来说不是问题，因为它总是会提供这样的限定。不过，这可能会对在远程服务器上通过触发器或者远程表上的规则执行的函数带来灾难。例如，如果一个远程表实际是一个视图，任何在该视图中使用的函数都将被在这个受限的搜索路径中执行。我们推荐在这类函数中用模式限定所有名称，或者为这类函数附着SET search_path选项（见CREATE FUNCTION）来建立它们所期望的搜索路径环境。

postgres_fdw同样为各种参数建立远程会话设置：

- TimeZone设置为UTC
- DateStyle设置为ISO
- IntervalStyle设置为postgres
- 对于远程服务器9.0和更新版本，extra_float_digits 设置为3，并且针对更老版本设置为2

这些不如search_path有那么多问题，但是如果需要也可以使用函数 SET选项来处理。

我们不推荐通过更改这些参数的会话级设置来推翻这种行为，这很可能会导致postgres_fdw故障。

F.33.6. 跨版本兼容性

postgres_fdw能够与最老是PostgreSQL 8.3 的远程服务器一起使用。只读能力则最低可以在8.1 中使用。但是一个限制是postgres_fdw通常假定不变的内建函数和操作符是安全的，如果它们出现在一个外部表的WHERE子句中，它们可以发送给远程服务器执行。因此，由于一个由于远程服务器的发布可能被发送给它来执行而被增加的内建函数，会导致“function does not exist”或一个类似的错误。这类错误可以通过重写查询来解决，例如通过嵌入在一个带OFFSET 0的子SELECT中引用的外部表作为一种优化墙，并且把出问题的函数或操作符放在子SELECT的外部。

F.33.7. 例子

这里是一个用postgres_fdw创建外部表的例子。首先安装该扩展：

```
CREATE EXTENSION postgres_fdw;
```

然后使用CREATE SERVER创建一个外部服务器。在这个例子中我们希望连接到一个位于主机192.83.123.89上并且监听5432端口的PostgreSQL服务器。在该远程服务器上要连接的数据库名为foreign_db：

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

需要用CREATE USER MAPPING定义一个用户映射来标识在远程服务器上使用哪个角色：

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

现在就可以使用CREATE FOREIGN TABLE创建外部表了。在这个例子中我们希望访问远程服务器上名为some_schema.some_table的表。它的本地名称是foreign_table：

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

CREATE FOREIGN TABLE中声明的列数据类型和其他性质必须要匹配实际的远程表。列名也必须匹配，不过也可以为个别列附上column_name选项以表示它们在远程服务器上对应哪个列。在很多情况中，要手工构造外部表定义，使用IMPORT FOREIGN SCHEMA会更好。

F.33.8. 作者

Shigeru Hanada <shigeru.hanada@gmail.com>

F.34. seg

这个模块为表示线段或者浮点区间实现了一种数据类型seg。 seg可以表示区间端点中的不确定性，这使得它对表达实验室测量特别有用。

F.34.1. 原理

度量的几何结构通常比一个数字连续区中的一个点更复杂。 一个度量通常是具有一些模糊限制的连续区的一个分段。 由于不确定性和随机性，也因为被度量的值可能天然地就是 一个指示某种情况的区间（例如一种蛋白质的稳态的温度范围）， 度量呈现为区间的形式。

只用常识，我们就知道将这类数据存储为区间比存储为数字对更加方便。 实际上，这样做在大部分应用中也更有效。

还是根据常识，限度的模糊性意味着使用传统的数字数据类型会导致信息丢失。 试想：你的仪器读到 6.50，并且你把这个读数输入到数据库。在你取出它时会 得到什么？看看：

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

在度量世界里，6.50 和 6.5 并不相同。有时候它们可能很不同。实验者们通常会 写下（并且发表）他们信任的数字。6.50 实际上是一个模糊的区间，它被包含于 一个更大的而且更模糊的区间 6.5 中，它们的中心点（可能）是它们唯一共享的 公共特征。我们绝对不希望这类不同的数据项表现得相同。

结论？一种能够记录具有任意可变精度的区间的限度的特殊数据类型是很好的。 这种意义下，每一个数据元素都记录其自身的精度。

来看看这个：

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
```

```
6.25 .. 6.50
(1 row)
```

F.34.2. 语法

一个区间的外部表达由通过范围操作符（..或者...）连接的一个或者两个浮点数构成。 或者，它也可以被指定为一个中心点加上或者减去一个偏差值。 也能够存储可选的确定性指示符（<、 >或者~）。不过，所有内建操作符会忽略确定性指示符。 表 F.26给出了所有允许的表达形式，表 F.27展示了一些例子。

在表 F.26中，x、y和delta表示浮点数。 x和y可以前置一个确定性指示符，但是delta不行。

表 F.26. seg外部表达

x	单一值（零长度区间）
x .. y	从x到y的区间
x (+-) delta	从x - delta到 x + delta的区间
x ..	下界为x的开区间
.. x	上界为x的开区间

表 F.27. 合法seg输入的例子

5.0	创建一个零长度的段（一个点）
~5.0	创建一个零长度的段并且在数据中记录~。~会被seg操作忽略，但是会被保留为一个注释。
<5.0	在 5.0 创建一个点。<会被忽略，但是被保留为一个注释。
>5.0	在 5.0 创建一个点。>会被忽略，但是被保留为一个注释。
5(+-)0.3	创建一个区间4.7 .. 5.3。注意(+-)标记不会被保留。
50 ..	大于或等于 50 的所有东西
.. 0	小于或等于 0 的所有东西
1.5e-2 .. 2E-2	创建一个区间0.015 .. 0.02
1 ... 2	与1...2、1 .. 2或者1..2相同 （范围操作符周围的空格会被忽略）

由于...被广泛地用在数据源中，它被允许作为..的一种替代。 不幸地是，这会带来解析歧义：分不清0...23的上界是23或者0.23。 通过要求seg输入中所有数字的小数点前至少有一位可以解决这个问题。

作为一种完整性检查，seg会拒绝下界大于上界的区间，例如5 .. 2。

F.34.3. 精度

seg值在内部被存储为一对 32 位浮点数。这意味着具有超过 7 个有效位的数字会被截断。

具有 7 个或者更少有效位的数字会保留它们的原有精度。即，如果你的查询返回0.00， 你可以确信拖尾的零不是人工造成的，它们反映了原始数据的精度。前导零的数量不影响精度： 值 0.0067 被认为只有 2 个有效位。

F.34.4. 用法

seg模块包括了用于seg值的一个 GiST 索引操作符类。 该 GiST 操作符类所支持的操作符在表 F.28中展示。

表 F.28. Seg GiST 操作符

操作符	描述
[a, b] << [c, d]	[a, b] 完全位于 [c, d] 左边。即如果b < c 则 [a, b] << [c, d] 为真，否则为假。
[a, b] >> [c, d]	[a, b] 完全位于 [c, d] 右边。即如果b > c 则 [a, b] >> [c, d] 为真，否则为假。
[a, b] &< [c, d]	重叠或者是在左边 — 这最好读作 “不超过右边”。当 b <= d 时为真。
[a, b] &> [c, d]	重叠或者是在右边 — 这最好读作 “不超过左边”。当 a >= c 时为真。
[a, b] = [c, d]	相等 — 段 [a, b] 和 [c, d] 是一样的，也就是 a = c 并且 b = d。
[a, b] && [c, d]	段 [a, b] 和 [c, d] 重叠。
[a, b] @> [c, d]	段 [a, b] 包含段 [c, d]，也就是 a <= c 并且 b >= d。
[a, b] <@ [c, d]	段 [a, b] 被包含在 [c, d] 中，也就是 >= c 并且 b <= d。

（在 PostgreSQL 8.2 之前，包含操作符 @> 和 <@ 分别被 称为@ 和 ~。这些名称仍然可用，但是已被废弃并且最终会消失。 注意旧的名称与核心几何数据类型之前遵循的习惯相反！）

也提供了标准的 B-树操作符，例如

操作符	描述
[a, b] < [c, d]	小于
[a, b] > [c, d]	大于

这些操作符对除了排序之外的任何实际目的都没有什么意义。 这些操作符首先比较 a 和 c，并且如果它们相等则比较 b 和 d。 在大部分情况下这会得到相当好的排序，如果你想对这种类型使用 ORDER BY， 这会很有用。

F.34.5. 注解

使用的例子，请见回归测试sql/seg.sql。

将(+-)转换成常规范围的机制在确定边界的有效位数时并不完全准确。 例如，如果结果区间包括一个 10 的幂时，它会加上一个额外的位：

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- should be: 9 .. 11
```

一个 R-树索引的性能很大程度上依赖于输入值的初始顺序。 将输入表以seg列进行排序将会非常有帮助， 例子可见脚本sort-segments.pl。

F.34.6. 开发人员

原始作者： Gene Selkov, Jr. <selkovjr@mcs.anl.gov>， 阿尔贡国家实验室数学和计算机科学部。

我要感谢 Joe Hellerstein 教授 (http://db.cs.berkeley.edu/jmh/) 解释了 GiST (http://gist.cs.berkeley.edu/)。 我也要向现在和过去的所有 Postgres 开发者致敬，让我能创造自己的世界并且不受打扰地生活在其中。我也要感谢阿尔贡实验室以及美国能源部多年来对我数据库研究的支持。

F.35. sepgsql

sepgsql是一个基于SELinux安全策略的 支持基于标签的强制访问控制（MAC）模块。

> **警告**
>
> 当前的实现具有明显的限制，并且不支持对所有动作的强制访问控制。详见 第 F.35.7 节

F.35.1. 概述

这个模块和SELinux集成在一起在 PostgreSQL提供的安全检查之上提供了一个 额外的安全检查层。从SELinux的角度来看，这个模块允许 PostgreSQL作为一个用户空间对象管理器。对每一次由 DML 查询发起的表或者函数访问将根据系统安全策略进行检查。这种 检查是在PostgreSQL执行的常规 SQL 权限 检查之外进行的。

SELinux访问控制决定是通过使用安全标签 来做出的，安全标签使用system_u:object_r:sepgsql_table_t:s0 这样的字符串表示。每个访问控制决定涉及两个标签：尝试执行该动作的主体的 标签以及要在其上执行该动作的客体的标签。由于这些标签可以被应用于任何种 类的对象，对于存储在数据库中的对象的（用这个模块做出的）访问控制决定服 从于用于任意其他类型对象（例如文件）的同一种一般准则。这种设计是为了允 许一种中央安全策略来保护信息资产，而不依赖于这些资产是如何存储的。

SECURITY LABEL语句允许为一个数据库对象分配安全标签。

F.35.2. 安装

sepgsql只能在启用了 SELinux的 Linux 2.6.28 或者更高版本上使用。在任何 其他平台上都无法使用这个模块。你将还需要 libselinux 2.1.10 或者更高版本以及 selinux-policy 3.9.13 或者更高版本（尽管某些发行中可能 把必要的规则逆向移植到较老的策略版本中）。

你可以使用sestatus命令检查 SELinux的状态。一种典型的显示是：

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

如果没有安装或者启用SELinux，你就必须在安装这个模块 之前先安装或者启用它。

要编译这个模块，应该在你的 PostgreSQL configure 命令中包括 选项--with-selinux。还要确定编译时安装了 libselinux-devel RPM 包。

要使用这个模块，你必须在postgresql.conf文件中的 shared_preload_libraries参数里包括 sepgsql。如果以其他任何方式载入该模块，它将无法正确地工作。 一旦该模块被载入，你应该在每一个数据库中执行 sepgsql.sql。这将会安装安全标签管理所需的函数 并且分配初始的安全标签。

这里有一个展示如何用sepgsql函数和安全标签初始化一个新 数据库集簇的例子（根据你的安装调整其中的路径）：

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  change
    #shared_preload_libraries = ''                # (change requires restart)
  to
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

请注意，如果你具有特定版本的libselinux和 selinux-policy，你可能会看到下列提示中的一些或者全部：

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object
 type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object
 type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object
 type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object
 type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object
 type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object
 type db_language
```

这些消息是无害的并且应该被忽略。

如果该安装过程完成时没有出现错误，就可以正常启动服务器了。

F.35.3. 回归测试

由于SELinux的本质，为sepgsql 运行回归测试要求一些额外的配置步骤，某些步骤还需要由 root 来完成。该回归测试 无法通过普通的make check或者make installcheck命令运行， 你必须建立配置并且接着手工调用测试脚本。这些测试必须在一个已配置 PostgreSQL 编译树的contrib/sepgsql目录中运行。尽管它们要求一个编译树，但是 这些测试被设计成在一个已安装服务器上执行，也就是说它们可以比得上 make installcheck（而不是make check）。

首先，根据第 F.35.2 节的指导在一个工作数据库中设置 sepgsql。注意当前操作系统用户必须能够不使用口令认证作 为超级用户连接到该数据库。

第二，为该回归测试编译和安装策略包。sepgsql-regtest策略是一个 特殊的策略包，它提供一组在回归测试浅见要被允许的规则。它应该从策略源文件 sepgsql-regtest.te编译，

这需要通过使用 make和一个 SELinux 提供的 Makefile 完成。你将需要 在你自己的系统上找到合适的 Makefile，下面展示的路径只是一个例子。一旦编译好， 使用semodule命令安装这个策略包，它会把所提供的策略包载入到 内核中。如果该包被正确地安装，semodule -l应该把 sepgsql-regtest列成一个可用的策略包：

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

第三，打开sepgsql_regression_test_mode。由于安全性的原因， sepgsql-regtest中的规则默认没有被启用。 the sepgsql_regression_test_mode参数会启用启动该回归 测试所需的规则。它可以使用setsebool命令来启用：

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

第四，验证你的 shell 在unconfined_t域中操作：

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

如果有必要，可以参考第 F.35.8 节调整你的工作域。

最后，运行该回归测试脚本：

```
$ ./test_sepgsql
```

这个脚本将尝试验证你已经正确地完成了所有的配置步骤，接下来它将运行 sepgsql模块的回归测试。

完成测试后，推荐你禁用 sepgsql_regression_test_mode参数：

```
$ sudo setsebool sepgsql_regression_test_mode off
```

你可能想要完全移除sepgsql-regtest策略：

```
$ sudo semodule -r sepgsql-regtest
```

F.35.4. GUC 参数

sepgsql.permissive (boolean)

不管系统设置如何，这个参数让sepgsql在自由模式中运行。 默认值为关闭。这个参数只能在postgresql.conf文件中或者 服务器命令行上被设置。

当这个参数为打开时，sepgsql在自由模式中运行，即便 SELinux 运行在强制模式中也是如此。这个参数主要用于测试目的。

sepgsql.debug_audit (boolean)

不管系统策略设置如何，这个参数启用打印审计消息。默认值为关闭，表示将 根据系统设置打印消息。

SELinux的安全性策略也具有控制是否记录特定访问的 规则。默认情况下，违法访问将会被记录，但是被允许的访问则不会被记录。

这个参数强制打开所有可能的记录而不管该系统策略。

F.35.5. 特性

F.35.5.1. 控制对象类

SELinux的安全模型把所有访问控制规则描述为一个 主体（典型的是一个数据库客户端）和一个客体（例如一个数据库对象）之 间的关系， 每一个这样的关系被一个安全标签标识。如果尝试访问一个未加 标签的客体，会认为该客体被分配了标签unlabeled_t。

当前，sepgsql允许把安全标签分配给模式、表、列、 序列、视图和函数。在使用sepgsql时，安全标签会 在所支持的数据库对象创建时自动分配给它们。这种标签被称为默认安全标签 并且根据系统安全性策略决定，默认安全标签被用来输入创建者标签、分配给 新对象父对象的标签以及所构造对象的可选名称。

一个新数据库对象基本上会继承父对象的安全标签，不过当安全策略具有特殊的 类型转换规则时，将会应用一个不同的标签。对于模式，其父对象是当前数据库。 对于表、序列、视图和函数，父对象是包含它的模式。对于列，其父对象是包含 它的表。

F.35.5.2. DML 权限

对于表，根据语句的种类会对所有被引用的目标表检查db_table:select、db_table:insert、 db_table:update或者db_table:delete。此外，对于 所有其列被WHERE或RETURNING子句引用、作为 UPDATE的数据源（以及其他情况）的表，都要检查db_table:select。

对每一个被引用的列也将检查列级权限。不仅在使用SELECT读取列 时会检查db_column:select，在其他 DML 语句中引用列时也要检查。 对于被UPDATE或者INSERT修改的列也将检查 db_column:update或者db_column:insert。

例如，考虑：

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

这里，将对t1.x检查db_column:update，因为它 被更新。对t1.y将检查db_column:{select update}， 因为它既被更新也被引用。并且会对t1.z检查 db_column:select，因为它只被更新。还将在表层面上检查 db_table:{select update}。

对于序列，当我们使用SELECT引用一个序列对象时会检查 db_sequence:get_value。不过，我们当前不会检查执行相应 函数（例如lastval()）的权限。

对于视图，将检查db_view:expand，然后对从视图展开来的任何 对象都会分别检查所需的权限。

对于函数，当用户尝试在一个查询中或者使用快路径调用执行一个函数时会检查db_procedure:{execute}。如果该函数是一个可信过程，也会检查 db_procedure:{entrypoint}权限来看看它能否作为一个可信程序 的入口点来执行。

为了访问任何模式对象，在其所在的模式上需要db_schema:search 权限。当不用模式限定引用一个对象时，其上没有该权限的模式不会被搜索（就好 像该用户在该模式上没有USAGE特权）。如果出现一个显式的模式 限定，当该用户在提及的模式上没有要求的权限时将会发生一个错误。

客户端必须被允许访问所有引用到的表和列，即便它们是由视图扩展得来的。这样 我们可以应用一致的访问控制规则而不管表内容被引用的方式。

默认的数据库特权系统允许数据库超级用户使用 DML 命令修改系统目录并且引用 或者修改 TOAST 表。当sepgsql被启用时，这些操作会被禁止。

F.35.5.3. DDL 权限

SELinux为每一种对象类型定义了数个权限来控制 常用操作，例如创建、修改、删除以及重新标记安全标签。此外，数种 对象类型具有特殊的权限来控制它们的特性化操作，例如在一个特定模式 中增加或者删除名字项。

创建一个新的数据库对象要求create权限。 SELinux将基于客户端的安全标签来授予或者否决 这个权限并且为新对象提出安全标签。在某些情况下，还需要额外的特权：

- CREATE DATABASE额外要求源数据库或者模板数 据库的getattr权限。
- 创建一个模式对象额外地要求父模式上的add_name权限。
- 创建一个表额外要求创建单个表列的权限，就好像每一个表列都是一个 单独的顶层对象。
- 创建一个被标记为LEAKPROOF的函数额外要求 install权限（当为一个现有函数设置 LEAKPROOF时也要检查这个权限）。

当执行DROP命令时，在要移除的对象上会检查drop。 对于通过CASCADE间接被删除的对象也会检查权限。删除包含在 一个特定模式内的对象（表、视图、序列以及过程）额外地要求该模式上的 remove_name。

在执行ALTER命令时，会在被修改的对象上为每一种对象类型检查 setattr。附属对象（例如一个表的索引或者触发器）除外， 这种 情况下权限是在父对象上检查的。在某些情况下，还需要额外的权限：

- 将一个对象移动到一个新的模式要求旧模式上的remove_name 权限以及新模式上的add_name权限。
- 设置一个函数上的LEAKPROOF属性要求install权限。
- 在一个对象上使用SECURITY LABEL会额外对该对象要求 relabelfrom权限连同它的旧安全标签以及relabelto 权限连同它的新安全标签（在安装了多个标签提供者并且用户尝试设置一个不由 SELinux管理的安全标签的情况中，这里只应该检查 setattr。当前由于实现限制没有这样做。）。

F.35.5.4. 可信过程

可信过程类似于 SECURITY DEFINER 函数或者 setuid 命令。 SELinux提供了一个特性来允许可信代码使用一个不同 于客户端的安全标签运行，通常这是为了提供对敏感数据的高度控制的访问（ 例如行可能会被忽略或者存储值的精度可能会被降低）。一个函数是否可以 作为可信过程受到其安全标签和操作系统安全性策略的控制。例如：

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                        FROM customer WHERE cid = $1'
           LANGUAGE sql;
```

```
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

上述的操作应该由一个管理员用户执行。

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

在这种情况下，一个常规用户无法直接引用customer.credit，但是一个可信过程show_credit允许用户在打印客户的信用卡号时 把一些数字掩盖掉。

F.35.5.5. 动态域转换

如果安全性策略允许，可以使用 SELinux 的动态域转换特性来切换客户端 进程（客户端域）的安全性标签到一个新的上下文。该客户端域需要 setcurrent权限还有从旧的域到新的域的 dyntransition权限。

动态域转换需要被仔细考虑，因为在用户看来，它们允许用户切换其标签， 并且因而切换特权，而不是（像可信过程的情况那样）受系统的强制性管理。 因此，只有当被用来切换到一个比原来的域具有更少特权的域时， dyntransition才被认为是安全的。例如：

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-------------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

在上面的这个例子中，我们被允许从较大范围的c1.c1023切换到 较小范围的c1.c4，但却禁止切换回去。

动态域转换和可信过程的组合开启了一种有趣的使用案例，它适合典型的 连接池软件的处理生命周期。即便你的连接池软件不被允许运行大部分的 SQL 命令，你可以从一个可信过程中使用 sepgsql_setcon()函数允许它切换该客户端的安全标签， 这个过程应该采用一些证据来授权该请求切换该客户端标签。之后，这个 会话将会具有目标用户而不是连接池的特权。该连接池之后可以用 NULL参数再次调用sepgsql_setcon() 逆转这次安全标签改变，当然再次的调用也要在一个可信过程中配合适当 的权限检查进行。这里的要点是只有可信过程实际具有权限来更改有效的安全 标签，并且只有在得到适当的证据后才这样做。当然，对于安全操作，必须 保护证据存储（表、过程定义或者其他什么）不会受到未经授权的访问。

F.35.5.6. 杂项

我们全面拒绝LOAD命令，因为任何模块的装载都 可能很轻易地绕过安全策略的强制保护。

F.35.6. Sepgsql 函数

表 F.29展示了可用的函数。

表 F.29. Sepgsql 函数

sepgsql_getcon() returns text	返回该客户端域，也就是该客户端当前的安全标签。
sepgsql_setcon(text) returns bool	如果安全性策略允许，把当前会话的客户端域切换到一个新的域。它也接受 NULL输入，并且把它当做是切换到该客户端原始域的请求。
sepgsql_mcstrans_in(text) returns text	如果 mcstrans 守护进程在运行中，把给定的限定 MLS/MCS 范围 翻译成原始格式。
sepgsql_mcstrans_out(text) returns text	如果 mcstrans 守护进程在运行中，把给定的原始 MLS/MCS 范围 翻译成限定格式。
sepgsql_restorecon(text) returns bool	在当前数据库中为所有对象设置初始安全标签。参数可能是 NULL，或者是 一个被用作系统默认 specfile 替代品的 specfile 名称。

F.35.7. 限制

数据定义语言（DDL）权限

收到实现的限制，一些 DDL 操作无法检查权限。

数据控制语言（DCL）权限

由于实现限制，DCL 操作不检查权限。

行级访问控制

PostgreSQL支持行级访问，但是 sepgsql不支持行级访问。

隐蔽通道

sepgsql不会尝试隐藏一个特定对象的存在，即便是 用户不被允许引用该对象。例如，即便我们无法得到一个不可见对象 的内容，我们也可以通过主键冲突、外键违背等等结果来推知该对象 的存在。一个绝密表的存在无法被隐藏，我们只希望能够隐藏其内容。

F.35.8. 外部资源

SE-PostgreSQL 介绍[5]

这个 wiki 页面提供了一个简单的综述、安全性设计、架构、 管理和即将到来的特性。

SELinux用户和管理员指南[6]

这个文档提供了广泛的知识来管理系统上的 SELinux。它主要关注 Red Hat 操作系统，但是并不仅限于此。

[5] https://wiki.postgresql.org/wiki/SEPostgreSQL

[6] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/

Fedora SELinux FAQ[7]

这个文档回答了很多SELinux 的常见问题。它主要关注 Fedora，但是并不仅限于此。

F.35.9. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.36. spi

spi模块提供了多个可工作的使用服务器编程接口 (SPI) 和触发器的例子。尽管这些例子的价值只对它们自己合适，它们甚至更有助于作为例子来修改达到你自己的目的。这些函数足够普通，可以与任何表一起使用，但是在创建一个触发器时你必须指定表名和域名（如下所述） 。

下面描述的函数组中的每一个都作为一个独立可安装的扩展被提供。

F.36.1. refint — 用于实现参照完整性的函数

check_primary_key()和check_foreign_key()被用来检查外键约束（当然，这个功能很早以前被内建的外建机制取代了，但是该模块还是可以用作一个例子）。

check_primary_key()检查引用表。用法是使用这个函数在一个引用其他表的表上创建一个BEFORE INSERT OR UPDATE触发器。指定该触发器的参数为：引用表中构成外键的列名、被引用表名称以及在被引用表中构成主键/唯一键的列名。要处理多个外键，请为每一个引用创建一个触发器。

check_foreign_key()检查被引用表。用法是使用这个函数在一个被其他表引用的表上创建一个BEFORE DELETE OR UPDATE触发器。指定该触发器的参数为：该函数必须对其执行检查的引用表数量、找到一个引用键后的动作（cascade — 删除引用行，restrict — 如果引用键存在则中断事务，setnull — 设置引用键域为空）、触发器所在表中构成主键/唯一键的列名、引用表名称和列名称（第一个参数指定多少个引用表就重复多少次）。注意主键/唯一键列应该被标记为 NOT NULL 并且应该有一个唯一索引。

refint.example中有一些例子。

F.36.2. timetravel — 实现时间旅行的函数

很久以前，PostgreSQL有一个内建的时间旅行特性，它为每一个元组保留插入和删除时间。这能够用这些函数模拟。要使用这些函数，你必须为一个表增加两个abstime类型的列来存储元组被插入的日期（start_date）以及被改变/删除的日期（stop_date）：

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

这些列可以以你喜欢的方式命名，但是在这次讨论中我们称它们为 start_date 和 stop_date。

当一个新行被插入时，start_date 通常应该被设置为当前时间，并且 stop_date 应当被设置为infinity。如果被插入的数据在这些列上包含空，触发器将自动替换这些值。通常，在这些列中插入显式非空数据的情况只有重新载入已转储数据。

[7] https://fedoraproject.org/wiki/SELinux_FAQ

等于infinity的元组“现在是合法的”，并且能够被修改。具有有限 stop_date 的元组不能再被修改 — 触发器将阻止修改（如果你需要这样做，你可以按照下面展示的关闭时间旅行）。

对于一个可修改的行，更新时只有被更新元组的 stop_date 将被改变（为当前时间）并且将会插入一个带有被修改数据的新元组。在这个新元组中的 start_date 将被设置为当前时间而 stop_date 被设置为infinity。

一次删除并非真正地移除元组，而只是将它的 stop_date 设置为当前时间。

要查询“当前有效”的元组，在查询的 WHERE 条件中包括stop_date = 'infinity'（你可能希望在一个视图中使用它）。类似地，你可以用 start_date 和 stop_date 上合适的条件来查询在任何过去时刻有效的元组。

timetravel()是支持这种行为的通用触发器函数。使用这个函数在每一个需要时间旅行的表上创建一个BEFORE INSERT OR UPDATE OR DELETE触发器。指定两个触发器参数：start_date 和 stop_date 列的真实名称。可选地，你可以指定 1-3 个更多的参数，它们必须表示text类型的列。该触发器将会在 INSERT、UPDATE、DELETE 期间将当前用户名分别存储到第 1、2、3 列中。

set_timetravel()允许你在一个表上打开或关闭时间旅行。set_timetravel('mytab', 1)将为表mytab打开时间旅行。set_timetravel('mytab', 0)将为表mytab关闭时间旅行。在两种情况中都会报告旧的状态。当时间旅行被关闭时，你可以自由地修改 start_date 和 stop_date 列。注意开/关状态是对于当前数据库会话局部可见的 — 新会话开始时所有表上的时间旅行总是被打开的。

get_timetravel()返回一个表的时间旅行状态，但不会改变它。

在timetravel.example中有一个例子。

F.36.3. autoinc — 用于自增域的函数

autoinc()是一个将序列的下一个值存储到一个整数域的触发器。这和内建的“序数列”特性有些重叠，但是它并不完全一样：autoinc()在插入时会覆盖掉给出的不同域值，并且它可被选择用来在更新时增加域。

用法是使用这个函数创建一个BEFORE INSERT（或者BEFORE INSERT OR UPDATE）触发器。指定两个触发器参数：要被修改的整数列名和将提供值的序列对象名（事实上，如果你想要更新多于一个自增列，你可以指定任意数量的这种名称对）。

在autoinc.example中有一个例子。

F.36.4. insert_username — 用于跟踪谁修改了一个表的函数

insert_username()是存储当前用户名到一个文本域的触发器。这有助于跟踪是谁最后在一个表中修改了一个特定行。

用法是使用这个函数创建一个BEFORE INSERT以及/或者UPDATE触发器。指定一个触发器参数：要被修改的文本列名。

在insert_username.example中有一个例子。

F.36.5. moddatetime — 用于跟踪上一次修改时间的函数

moddatetime()是一个存储当前时间到一个timestamp域的触发器。它有助于跟踪一个表中特定行最后一次的修改时间。

用法是使用这个函数创建一个BEFORE UPDATE触发器。指定一个触发器参数：要被修改的列名。该列必须是类型timestamp或者timestamp with time zone。

在moddatetime.example中有一个例子。

F.37. sslinfo

在连接到PostgreSQL时，sslinfo模块提供当前客户端提供的 SSL 证书的有关信息。如果当前连接不使用 SSL，这个模块就没有用处（大部分函数将返回 NULL）。

除非安装时用--with-openssl进行了配置，这个扩展压根就不会被编译。

F.37.1. 提供的函数

ssl_is_used() 返回 boolean

如果当前到服务器的连接使用 SSL 则返回 true，否则返回 false。

ssl_version() 返回 text

返回 SSL 连接使用的协议名称（如 TLSv1.0、TLSv1.1 或者 TLSv1.2）。

ssl_cipher() 返回 text

返回 SSL 连接所用的加密方法名称（如 DHE-RSA-AES256-SHA）。

ssl_client_cert_present() 返回 boolean

如果当前客户端已经向服务器出示了一个合法的 SSL 客户端证书则返回 true，否则返回 false（服务器可能被配置要求一个客户端配置，也可能没有被配置成这样）。

ssl_client_serial() 返回 numeric

返回当前客户端证书的序列号。证书序列号和证书发行人的组合被确保可以 唯一标识一个证书（但是不能唯一标识其拥有者 — 拥有者应该定期地更换其密钥， 并且从发行人那里得到新的证书）。

因此，如果你运行自己的 CA 并且只允许服务器接收来自于这个 CA 的证书，序列号就是最可靠的（虽然并非很好记忆）标识一个用户的方法。

ssl_client_dn() 返回 text

返回当前客户端证书的完整主题，并且将字符数据转换成当前数据库的编码。我们假定如果你在证书名中使用非 ASCII 字符，你的数据库也有能力展示这些字符。如果你的数据库使用 SQL_ASCII 编码，名称中的非 ASCII 字符将被表示为 UTF-8 序列。

结果看起来像/CN=某人 /C=某个国家 /O=某个组织。

ssl_issuer_dn() 返回 text

返回当前客户端证书的完整的发行人名称，并把字符数据转换成当前数据库的编码。编码转换以与ssl_client_dn相同的方式处理。

这个函数的返回值与证书序列号的组合唯一地标识证书。

如果在服务器的证书授权中心文件中有多于一个的可信 CA 证书，或者如果 CAI 已经发行了某些中间认证授权证书，这个函数就真的很有用。

ssl_client_dn_field(fieldname text) 返回 text

这个函数返回证书主题中指定域的值，如果域不存在则返回 NULL。域的名称是字符串常量，它们被使用 OpenSSL 对象数据库转换成 ASN1 对象标识符。下列值是可接受的：

commonName (alias CN)

surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress

这些域中除了commonName都是可选的。它们之中哪些会被包括或者不会被包括完全取决于你的 CA 策略。不过，这些域的含义由 X.500 和 X.509 标准严格地定义，因此你不能只是为它们分配任意含义。

ssl_issuer_field(fieldname text) 返回 text

和ssl_client_dn_field一样，但是用于证书发行人而不是证书主题。

ssl_extension_info() 返回 setof record

提供有关客户端证书扩展的信息：扩展名、扩展值以及是否为 决定性的扩展。

F.37.2. 作者

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Dmitry Voronin <carriingfate92@yandex.ru>

Cryptocom OpenSSL 开发组的 E-Mail： <openssl@cryptocom.ru>

F.38. tablefunc

tablefunc模块包括多个返回表（也就是多行）的函数。这些函数都很有用，并且也可以作为如何编写返回多行的 C 函数的例子。

F.38.1. 所提供的函数

表 F.30显示了tablefunc模块提供的函数。

表 F.30. tablefunc函数

函数	返回	描述
normal_rand(int numvals, float8 mean, float8 stddev)	setof float8	产生一个正态分布的随机值集合
crosstab(text sql)	setof record	产生一个包含行名称外加N个值列的“数据透视表”，其

函数	返回	描述
		中N由调用查询中指定的行类型决定
crosstabN(text sql)	setof table_crosstab_N	产生一个包含行名称外加N个值列的“数据透视表”。crosstab2、crosstab3和crosstab4是被预定义的，但你可以按照下文所述创建额外的crosstabN函数
crosstab(text source_sql, text category_sql)	setof record	产生一个“数据透视表”，其值列由第二个查询指定
crosstab(text sql, int N)	setof record	crosstab(text)的废弃版本。参数N现在被忽略，因为值列的数量总是由调用查询所决定
connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld], text start_with, int max_depth [, text branch_delim])	setof record	产生一个层次树结构的表达

F.38.1.1. normal_rand

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

normal_rand产生一个正态分布随机值（高斯分布）的集合。

numvals是从该函数返回的值的数量。mean是值的正态分布的均值而stddev是值的正态分布的标准偏差。

例如，这个调用请求 1000 个值，它们具有均值 5 和标准偏差 3：

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
    -0.369151492880995
     0.283600703686639
        .
        .
        .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.38.1.2. crosstab(text)

```
crosstab(text sql)
crosstab(text sql, int N)
```

crosstab函数被用来产生“pivot”显示，在其中数据被横布在页面上而不是直接向下列举。例如，我们可能有这样的数据

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

而我们希望显示成这样

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

crosstab函数会采用一个文本参数，该文本是一个 SQL 查询，它产生按照第一种方式格式化的原始数据，并且产生以第二种方式格式化的一个表。

sql参数是一个产生数据的源集合的 SQL 语句。这个语句必须返回一个row_name列、一个category列和一个value列。N是一个废弃参数，即使提供也会被忽略（之前这必须匹配输出值列的数目，但是现在这由调用查询决定了）。

例如，所提供的查询可能会产生这样的一个集合：

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

crosstab函数被声明为返回setof record， 因此输出列的实际名称和类型必须定义在调用的SELECT语句的FROM子句中，例如：

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2
 text);
```

这个例子产生这样一个集合：

```
          <== value  columns  ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

FROM子句必须把输出定义为一个row_name列 （具有 SQL 查询的第一个结果列的相同数据类型），其后跟随着 N 个value列 （都具有 SQL 查询的第三个结果列的相同数据类型）。你可以按照你的意愿设置任意多的输出值列。 而输出列的名称取决于你。

crosstab函数为具有相同row_name值的 输入行的每一个连续分组产生一个输出行。它使用来自这些行的值域 从左至右填充输出的值列。如果一个分组中的行比输出值列少， 多余的输出列将被用空值填充。如果行更多，则多余的输入行会被跳过。

事实上，SQL 查询应该总是指定ORDER BY 1,2来保证输入行被正确地排序， 也就是说具有相同row_name的值会被放在一起并且在行内 被正确地排序。注意crosstab本身并不关注查询结果的第二列，它放在那里 只是为了被排序，以便控制出现在页面上的第三列值的顺序。

这是一个完整的例子：

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

你可以避免总是要写出一个FROM子句来定义输出列， 方法是设置一个在其定义中硬编码所期望的输出行类型的自定义 crosstab 函数。 这会在下一节中描述。另一种可能性是在一个视图定义中嵌入所需的FROM子句。

> **注意**
>
> 另见psql中的 \crosstabview 命令，它提供了和crosstab()类似的功能。

F.38.1.3. crosstabN(text)

```
crosstabN(text sql)
```

crosstabN系列函数是如何为普通crosstab 函数设置自定义包装器的例子，这样你不需要在调用的SELECT查询中 写出列名和类型。tablefunc模块包括crosstab2、 crosstab3以及crosstab4，它们的输入行类型被定义为：

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
```

```
        .
        .
        .
    category_N TEXT
);
```

因此，当输入查询产生类型为text的列row_name和value 并且想要 2、3 或 4 个输出值列时，这些函数可以被直接使用。在所有其他方法中，它们的行为都和上面的 一般crosstab函数完全相同。

例如，前一节给出的例子也可以这样来做

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

这些函数主要是出于举例的目的而提供。你可以基于底层的crosstab()函数 创建你自己的返回类型和函数。有两种方法来做：

- 与contrib/tablefunc/tablefunc--1.0.sql中相似，创建一个组合类型来描述所期望的输出列。 然后定义一个唯一的函数名，它接受一个text参数并且返回setof your_type_name，但是链接到同样的 底层crosstab C 函数。例如，如果你的源数据产生为text类型的行名称，并且值是float8， 并且你想要 5 个值列：

```
  CREATE TYPE my_crosstab_float8_5_cols AS (
      my_row_name text,
      my_category_1 float8,
      my_category_2 float8,
      my_category_3 float8,
      my_category_4 float8,
      my_category_5 float8
  );

  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
      RETURNS setof my_crosstab_float8_5_cols
      AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

- 使用OUT参数来隐式定义返回类型。同样的例子也可以这样来做：

```
  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
      IN text,
      OUT my_row_name text,
      OUT my_category_1 float8,
      OUT my_category_2 float8,
      OUT my_category_3 float8,
      OUT my_category_4 float8,
      OUT my_category_5 float8)
    RETURNS setof record
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.38.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

crosstab的单一参数形式的主要限制是它把一个组中的所有值都视作相似，并且把每一个值插入到第一个可用的列中。如果你想要值列对应于特定的数据分类，并且某些分组可能没有关于某些分类的数据，这样的形式就无法工作。crosstab的双参数形式通过提供一个对应于输出列的显式分类列表来处理这种情况。

source_sql是一个产生源数据集的SQL语句。这个语句必须返回一个row_name列、一个category列以及一个value列。也可以有一个或者多个“extra”列。row_name列必须是第一个。category和value列必须是按照这个顺序的最后两个列。row_name和category之间的任何列都被视作“extra”。对于具有相同row_name值的所有行，其“extra”列都应该相同。

例如，source_sql可能产生一组这样的东西：

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

category_sql是一个产生分类集合的SQL语句。这个语句必须只返回一列。它必须产生至少一行，否则会生成一个错误。还有，它不能产生重复值，否则会生成一个错误。category_sql可能是这样的：

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

crosstab函数被声明为返回setof record，这样输出列的实际名称和类型就必须在调用的SELECT语句的FROM子句中被定义，例如：

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4
 text);
```

这将产生这样的结果：

```
                <==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

FROM子句必须定义正确数量的输出列以及正确的数据类型。如果在source_sql查询的结果中有N列，其中的前N-2 列必须匹配前N-2 个输出列。剩余的输出列必须具有source_sql查询

结果的最后一列的类型，并且并且它们的数量　必须正好和source_sql查询结果中的行数相同。

crosstab函数为具有相同row_name值的输入行形成的每一个连续分组　　产生一个输出行。输出的row_name列外加任意一个“extra”列都是从分组的　　第一行复制而来。输出的value列被使用具有匹配的category值的行中的　value域填充。如果一个行的category不匹配category_sql　查询的任何输出，它的value会被忽略。匹配的分类不出现于分组中任何输出行中的的 输出列会被用空值填充。

事实上，source_sql查询应该总是指定ORDER BY 1来保证 具有相同row_name的值会被放在一起。但是，一个分组内分类的顺序并不重要。　还有，确保category_sql查询的输出的顺序与指定的输出列顺序匹配是非常重要的。

这里有两个完整的例子：

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------
+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 |
 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)


CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March
 2003');
```

```
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |
 test_startdate      | volts
-------+--------------------------+-------------+-------------
+--------------------------+--------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
         | 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01
 00:00:00 2003 | 3.1234
(2 rows)
```

你可以创建预定义的函数来避免在每个查询中都必须写出结果列的名称和类型。请参考前一节中的例子。 用于这种形式的crosstab的底层 C 函数被命名为crosstab_hash。

F.38.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

connectby函数产生存储在一个表中的层次数据的显示。该表必须具有一个用以 唯一标识行的键域，以及一个父亲键域用来引用其父亲（如果有）。connectby能 显示从任意行开始向下的子树。

表 F.3解释了参数。

表 F.31.connectby 参数

参数	描述
relname	源关系的名称
keyid_fld	键域的名称
parent_keyid_fld	父亲键域的名称
orderby_fld	用于排序兄弟的域的名称（可选）
start_with	起始行的键值
max_depth	要向下的最大深度，零表示无限深度
branch_delim	在分支输出中用于分隔键值的字符串（可选）

键域和父亲键域可以是任意数据类型，但是它们必须是同一类型。 注意start_with值必须作为一个文本串被输入，而不管键域的类型如何。

connectby函数被声明为返回setof record，因此输出列的实际名称和类型 就必须在调用的SELECT语句的FROM子句中被定义，例如：

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos',
 'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

前两个输出列被用于当前行的键和其父亲行的键，它们必须匹配该表的键域的类型。第三个输出行是该树中的深度， 并且必须是类型integer。如果给定了一个branch_delim参数，下一个输出列 就是分支显示并且必须是类型text。最后，如果给出了一个orderby_fld参数，最后一个输出列是一个序号，并且必须是类型integer。

“branch”输出列显示了用于到达当前行的由键构成的路径。其中的键用指定的branch_delim 字符串分隔开。如果不需要分支显示，可以在输出列列表中忽略branch_delim参数和分支列。

如果同一父亲的子女之间的顺序很重要，可以包括orderby_fld参数以指定用哪个域对兄弟排序。 这个域可以是任何可排序数据类型。当且仅当orderby_fld被指定时，输出列列表必须包括一个 最终的整数序号列。

表示表和列名的参数会被原样复制到connectby内部生成的 SQL 查询中。 因此，如果名称是大小写混合或者包含特殊字符，应包括双引号。你也可能需要用模式限定表名。

在大型的表中，除非在父亲键域上有索引，否则性能会很差。

branch_delim字符串不出现在任何键值中是很重要的，否则connectby可能会错误地 报告一个无限递归错误。注意如果没有提供branch_delim，将用一个默认值~来进行递归检测。

这里是一个例子：

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- 带有分支，但没有 orderby_fld （不保证结果的顺序）
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0,
 '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+---------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- 没有分支，也没有 orderby_fld （不保证结果的顺序）
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
```

```
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- 有分支，有 orderby_fld （注意 row5 在 row4 前面）
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos',
 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- 没有分支，有 orderby_fld （注意 row5 在 row4 前面）
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos',
 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.38.2. 作者

Joe Conway

F.39. tcn

tcn模块提供一个触发器函数，它通知监听者有关它所附着的任意表上的改变。它必须被用作一个行级AFTER触发器。

在一个CREATE TRIGGER语句中只可以为该函数提供一个参数，并且是可选的。如果提供该参数，它将被作为用于通知的频道名。如果忽略它，频道名将使用tcn。

通知的负载由表名、一个指示所执行操作类型的字母以及用于主键列的列名/值对构成。每一部分都用逗号与下一部分隔开。为了便于解析对正则表达式的使用，表和列名总是被包裹在双引号内，并且数据值总是被包裹在单引号内。嵌入的引号都被双写。

下面是使用该扩展的简单例子。

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute function triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='1',"b"='2012-12-22'" received from server process with PID
 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='1',"b"='2012-12-23'" received from server process with PID
 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='2',"b"='2012-12-23'" received from server process with PID
 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload
 ""tcndata",U,"a"='1',"b"='2012-12-22'" received from server process with PID
 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",U,"a"='1',"b"='2012-12-23'" received from server process with PID
 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload
 ""tcndata",D,"a"='1',"b"='2012-12-22'" received from server process with PID
 22770.
```

F.40. test_decoding

test_decoding是一个逻辑解码输出插件的例子。它其实不做 任何特别有用的事情，但是可以作为开发你自己的输出插件的起点。

test_decoding通过逻辑解码机制接收 WAL 并且把它解码成 被执行的操作的文本表达形式。

这个插件的典型输出（在 SQL 逻辑解码接口上使用）可能是：

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL,
 'include-xids', '0');
 location  | xid |                       data
-----------+-----+--------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
```

```
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

F.41. tsm_system_rows

tsm_system_rows模块提供了表采样方法 SYSTEM_ROWS，它可以用在SELECT 命令的TABLESAMPLE子句中。

这种表采样方法接受一个整数参数，它是要读取的最大行数。得到的采样将总是包 含正好这么多行，除非该表中没有足够的行，在那种情况下整个表都会被选择出来。

和内建的SYSTEM采样方法一样， SYSTEM_ROWS执行块级别的采样，所以采样不是完全随机的， 而是服从于聚簇效果，特别是只要求少量行时。

SYSTEM_ROWS不支持 REPEATABLE子句。

F.41.1. 示例

这里是一个用SYSTEM_ROWS选择一个表采样的例子。 首先安装扩展：

```
CREATE EXTENSION tsm_system_rows;
```

然后就可以在SELECT命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

这个命令从表my_table中返回一个 100 行的采样（除非 该表没有 100 个可见行，那时将会返回其中所有的行）。

F.42. tsm_system_time

tsm_system_time模块提供了表采样方法 SYSTEM_TIME，它可以用在SELECT 命令的TABLESAMPLE子句中。

这种表采样方法接受一个浮点类型的参数，它是花费在读表上的最大毫秒数。 这可以让你直接控制查询进行多久，但付出的代价是很难预测采样的尺寸。 得到的采样将包含在指定时间内能读入的那么多行，除非首先已经读入了整个表。

和内建的SYSTEM采样方法一样， SYSTEM_TIME执行块级别的采样，所以采样不是完全随机的， 而是服从于聚簇效果，特别是只选择少量行时。

SYSTEM_TIME不支持 REPEATABLE子句。

F.42.1. 示例

这里是一个用SYSTEM_TIME选择一个表采样的例子。 首先安装扩展：

```
CREATE EXTENSION tsm_system_time;
```

然后就可以在SELECT命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

这个命令将返回在 1 秒（1000 毫秒）能读到的my_table 采样。当然，如果 1 秒内就能读完整个表，所有的行都将被返回。

F.43. unaccent

unaccent是一个文本搜索字典，它能从词位中移除重音（附加符号）。它是一个过滤词典，这表示它的输出总是会被传递给下一个字典（如果有），这和字典的通常行为不同。这允许为全文搜索做与重音无关的处理。

unaccent的当前实现不能被用作thesaurus字典的正规化字典。

F.43.1. 配置

unaccent字典接受下列选项：

- RULES是包含翻译规则列表的文件的基本名。这个文件必须被存储在$SHAREDIR/tsearch_data/（这里$SHAREDIR表示PostgreSQL安装的共享数据目录）中。它的名称必须以.rules（不包含在RULES参数中）结束。

规则文件具有下面的格式：

- 每一行表示一个由带有重音的字符和不带重音的字符构成的对。第一个字符将被翻译成第二个。例如：

```
À       A
Á       A
Â       A
Ã       A
Ä       A
Å       A
Æ       AE
```

  两个字符必须由空格分隔，并且一行上的任何前导或尾随空白都将被忽略。

- 或者，如果一行只给出一个字符，则删除该字符的实例； 这在用单独的字符表示重音的语言中是有用的。

- 实际上，每个“字符”可以是不包含空格的任何字符串，因此， 除了去除变音符之外，unaccent字典也可以用于其他类型的字符串替换。

- 与其他PostgreSQL文本搜索配置文件一样， 规则文件必须以UTF-8编码方式存储。加载时，数据将自动转换为当前数据库的编码。 任何含有不可翻译字符的行都将被忽略，因此规则文件可以包含当前编码中不适用的规则。

在unaccent.rules中可以找到一个更完整的例子，它可以直接用于大部分欧洲语言，当unaccent模块被安装时，它被安装在$SHAREDIR/tsearch_data/中。

F.43.2. 用法

安装unaccent扩展会创建一个文本搜索模板unaccent和一个基于前者的字典unaccent。unaccent字典有默认的参数设置RULES='unaccent'，这会让该字典使用标准的unaccent.rules文件。如果希望修改该参数，可以

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

或者基于该模板创建新的字典。

要测试该字典，你可以尝试：

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

这里是一个展示把unaccent字典插入到一个文本搜索配置的例子：

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
(1 row)
```

F.43.3. 函数

unaccent()函数从一个给定的字符串中移除重音（附加符号）。基本上，它是unaccent字典的一个包装器，但是它能在普通的文本搜索环境之外使用。

```
unaccent([dictionary regdictionary, ] string text) returns text
```

如果省略dictionary参数，则使用名为unaccent并且与unaccent()函数有相同模式的文本搜索词典。

例如：

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.44. uuid-ossp

uuid-ossp模块提供函数使用几种标准算法之一产生通用唯一标识符（UUID）。还提供产生某些特殊 UUID 常量的函数。

F.44.1. uuid-ossp 函数

表 F.32展示了可用来产生 UUID 的函数。相关标准 ITU-T Rec. X.667、ISO/IEC 9834-8:2005 以及 RFC 4122 指定了四种用于产生 UUID 的算法，分别用版本号 1、3、4、5 标识（没有版本 2 的算法）。这些算法中的每一个都适合于不同的应用集合。

表 F.32. 用于 UUID 产生的函数

函数	描述
uuid_generate_v1()	这个函数产生一个版本 1 的 UUID。这涉及到计算机的 MAC 地址和一个时间戳。注意这种 UUID 会泄露产生该标识符的计算机标识以及产生的时间，因此它不适合某些对安全性很敏感的应用。
uuid_generate_v1mc()	这个函数产生一个版本 1 的 UUID，但是使用一个随机广播 MAC 地址而不是该计算机真实的 MAC 地址。
uuid_generate_v3(namespace uuid, name text)	这个函数使用指定的输入名称在给定的名字空间中产生一个版本 3 的 UUID。该名字空间应该是由uuid_ns_*()函数（如表 F.33所示）产生的特殊常量之一（理论上它可以是任意 UUID）。名称是选择的名字空间中的一个标识符。 例如： SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org'); 名称参数将使用 MD5 进行哈希，因此从产生的 UUID 中得不到明文。采用这种方法的 UUID 生成没有随机性并且不涉及依赖于环境的元素，因此是可以重现的。
uuid_generate_v4()	这个函数产生一个版本 4 的 UUID，它完全从随机数产生。
uuid_generate_v5(namespace uuid, name text)	这个函数产生一个版本 5 的 UUID，它和版本 3 的 UUID 相似，但是采用的是 SHA-1 作为哈希方法。版本 5 比版本 3 更好，因为 SHA-1 被认为比 MD5 更安全。

表 F.33. 返回 UUID 常量的函数

uuid_nil()	一个“nil” UUID 常量，它不作为一个真正的 UUID 发生。
uuid_ns_dns()	为 UUID 指定 DNS 名字空间的常量。
uuid_ns_url()	为 UUID 指定 URL 名字空间的常量。
uuid_ns_oid()	为 UUID 指定 ISO 对象标识符（OID）名字空间的常量（这属于 ASN.1 OID，它与PostgreSQL使用的 OID 无关）。
uuid_ns_x500()	为 UUID 指定 X.500 可识别名（DN）名字空间的常量。Constant designating the X.500 distinguished name (DN) namespace for UUIDs.

F.44.2. 编译uid-ossp

在历史上这个模块依赖于 OSSP UUID 库，这也是这个模块名称的由来。虽然 现在还能在http://www.ossp.org/pkg/lib/uuid/上 找到 OSSP UUID 库，但是它已经不再被维护并且越来越难以被一直到新的平台。 uuid-ossp现在在一些平台上可以脱离 OSSP 库被编译。在 FreeBSD、NetBSD 和一些其他源自 BSD 的平台上，在核心的libc 库中已经包括了合适的 UUID 创建函数。在 Linux、macOS和一些其他平台上， libuuid库中提供了合适的函数，它最初是来自于 e2fsprogs项目（不过在现代 Linux 上它被认为是 util-linux-ng的一部分）。在调用configure时， 指定--with-uuid=bsd可使用 BSD 的函数，指定 --with-uuid=e2fs会使用e2fsprogs的 libuuid，指定--with-uuid=ossp则会 使用 OSSP UUID 库。在一台特定的机器上可能会存在多种上述的库，因此 configure不会自动选择其中一个。

> **注意**
>
> 如果你只需要随机生成（版本4）的 UUID，可以考虑使用pgcrypto 模块中的gen_random_uuid()函数。

F.44.3. 作者

Peter Eisentraut <peter_e@gmx.net>

F.45. xml2

xml2模块提供 XPath 查询和 XSLT 功能。

F.45.1. 废弃公告

从PostgreSQL 8.3 开始，在核心服务器中就已经有基于 SQL/XML 标准的 XML 相关功能。那些功能覆盖了 XML 语法检查和 XPath 查询，这些本模块也能做，但是其 API 并不是完全兼容。这个模块已经有计划将从 PostgreSQL 的一个未来版本中移除，因此我们鼓励你尝试转换你的应用。如果你发现这个模块的某些功能在更新的 API 中没有合适的形式相对应，请向<pgsql-hackers@lists.postgresql.org>表达你的问题，这样该缺点会被进行改进。

F.45.2. 函数的描述

表 F.3展示了这个模块提供的函数。这些函数提供了直接的 XML 解析和 XPath 查询。所有参数都是text类型，因此为了简洁都没有被显示。

表 F.34. 函数

函数	返回	描述
xml_is_well_formed(document)	bool	这个函数解析其参数中的文档文本并且在该文档是一个结构良好的 XML 时返回真 （注意，这是标准PostgreSQL函数xml_is_well_formed()的别名。 名称xml_valid()在技术上是不正确的，因为在 XML 中有效性和结构良好性具有不同的含义。）
xpath_string(document, query)	text	这些函数在提供的文档上计算 XPath 查询，并且将结果造型为指定的类型。

函数	返回	描述
xpath_number(document, query)	float4	
xpath_bool(document, query)	bool	
xpath_nodeset(document, query, toptag, itemtag)	text	这个函数在文档上计算查询并且把结果包装在 XML 标签中。如果结果是多值的，输出看起来是这样： <toptag> <itemtag>Value 1 which could be an XML fragment</itemtag> <itemtag>Value 2....</itemtag> </toptag> 如果toptag或者itemtag是一个空字符串，相关的标签会被忽略。
xpath_nodeset(document, query)	text	与xpath_nodeset(document, query, toptag, itemtag)相似但是结果忽略两种标签。
xpath_nodeset(document, query, itemtag)	text	与xpath_nodeset(document, query, toptag, itemtag)相似但是结果忽略toptag。
xpath_list(document, query, separator)	text	这个函数返回多个值，并且用指定的分隔符分隔，例如分隔符是,，结果就是Value 1,Value 2,Value 3。
xpath_list(document, query)	text	这是上面函数的一个包装器，它用,作为分隔符。

F.45.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria)
 returns setof record
```

xpath_table是一个表函数，它在一组文档中的每一个上计算一组 XPath 查询，并且将结果作为一个表返回。来自原始文档表的主键域被返回为结果的第一列，这样结果集可以被用于连接。其参数在表 F.35中描述。

表 F.35.xpath_table 参数

参数	描述
key	“key”域的名称 — 这只是被用作输出表中第一列的域，即它标识每一个输出行是来自于哪个记录（见下面有关多个值的注解）
document	包含 XML 文档的域的名称
relation	包含文档的表或视图的名称
xpaths	一个或多个 XPath 表达式，用|分隔

参数	描述
criteria	WHERE 子句的内容。这不能被忽略，因此如果你想要处理关系中的所有行，可以使用true或1=1

这些参数（除了 XPath 字符串）只是会被替换到一个纯粹的 SQL SELECT 语句中，因此你有一些灵活性 — 该语句是

SELECT <key>, <document> FROM <relation> WHERE <criteria>

因此那些参数可以是那些特定位置上合法的任何东西。来自于这个 SELECT 的结果需要返回正好两列（它确实会这样，除非你尝试为键或文档列出多个域）。注意这种简单方法要求你验证任何用户提供的值来避免 SQL 注入攻击。

该函数必须被使用在一个FROM表达式中，并带有一个AS子句来指定输出列，例如

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

AS子句定义了输出表中列的名称和类型。第一个是“key”域并且剩下的对应于 XPath 查询。如果 XPath 查询比结果列多，额外的查询将被忽略。如果结果列比 XPath 查询多，额外的列将是 NULL。

注意这个例子定义page_count结果列为一个整数。该函数在内部处理字符串表达，因此当你在输出中想要一个整数时，它将采用 XPath 结果的字符串表达并且使用 PostgreSQL 输入函数来把它转换成一个整数（或者AS子句要求的任何类型）。如果它无法做到这一点将会导致一个错误 — 例如结果是空 — 因此如果你认为你的数据有任何问题，你可能希望坚持用text作为列类型。

调用的SELECT语句不必只是 SELECT * — 它可以用名称引用输出列或者将它们连接到其他表。该函数会产生一个虚拟表，你可以在其上执行任何所需的操作（例如聚集、连接、排序等）。因此我们也可以有：

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20''
 ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

作为一个更复杂的例子。当然，为了便利你也可以把所有这些包装在一个视图中。

F.45.3.1. 多值结果

xpath_table函数假定每一个 XPath 查询的结果可能是多值的，因此该函数返回的行数可能与输入文档的数目不同。被返回的第一行包含来自每一个查询的第一个结果，第二行则是来自每一个查询的第二个结果。如果其中一个查询的值比其他查询少，则会为它返回空值。

在某些情况下，一个用户将知道一个给定的 XPath 查询将只返回一个单一结果（可能是一个唯一文档标识符） — 如果和一个返回多值的 XPath 查询一起使用，单值结果将只出现在结

果的第一行中。对于这种情况的解决方案是使用键域作为针对一个更简单 XPath 查询的连接的一部分。一个例子：

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int,
 val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

要在每一行上得到doc_num，解决方案是使用xpath_table的两个调用并且连接结果：

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.45.4. XSLT 函数

如果安装了 libxslt，那么可以使用下列函数：

F.45.4.1. xslt_process

xslt_process(text document, text stylesheet, text paramlist) returns text

这个函数将 XSL 样式表应用于文档并且返回转换过的结果。paramlist是一个被用在转换中的参数赋值列表，以a=1,b=2的形式指定。注意参数解析是非常天真的：参数值不能包含逗号！

还有一个双参数版本的xslt_process，它不会向转换传递任何参数。

F.45.5. 作者

John Gray <jgray@azuli.co.uk>

这个模块的开发由 Torchbox Ltd. (www.torchbox.com) 赞助。它使用了和 PostgreSQL 相同的 BSD 许可证。

附录 G. 额外提供的程序

这个附录及之前的一个包含了在PostgreSQL发布的contrib目录中可以找到的模块的信息。在附录 I中有更多关于contrib小节的总体信息，以及contrib中的服务器扩展和插件的详细信息。

这个附录覆盖了contrib中能找到的功能程序。一旦被安装（不管是从源代码安装还是从包系统安装），它们就可以在PostgreSQL安装的bin目录中找到，并且能够和任何其他程序一样使用。

G.1. 客户端应用

这一节覆盖了contrib中的PostgreSQL客户端应用。它们可以从任何位置运行，不管数据库服务器运行在哪里。核心PostgreSQL发布中的客户端应用也可见PostgreSQL 客户端应用。

oid2name

oid2name — 解析一个PostgreSQL数据目录中的 OID 和文件结点。

大纲

```
oid2name [option...]
```

描述

oid2name是一个帮助管理员检查被 PostgreSQL 使用的文件结构的工具程序。要使用它，你需要熟悉数据库文件结构（见第 68 章。

> **注意**
>
> 名称“oid2name”是有历史原因的，它确实有些误导性，因为在你使用它的大部分时间里，你实际关心的是表的文件结点编号（在数据目录中是可见的文件名）。请确定你理解表 OID 和表文件结点之间的区别！

oid2name连接到一个目标数据库并且抽取 OID、文件节点或者表名信息。你也可以让它显示数据库 OID 或表空间 OID。

选项

oid2name接受下列命令行参数：

-f filenode

显示具有文件结点filenode的表的信息

-i

在列举中包括索引和序列

-o oid

显示具有 OID oid的表的信息

-q

忽略头部（用于脚本）

-s

显示表空间 OID

-S

包括系统对象（位于 information_schema、pg_toast和pg_catalog模式中）

-t tablename_pattern

显示匹配tablename_pattern的表的信息

-V
--version

打印oid2name版本并退出。

-x

为要显示的每个对象显示更多信息：表空间名、模式名以及 OID

-?
--help

显示有关oid2name命令行参数的帮助并退出。

oid2name也接受下列用于连接参数的命令行参数：

-d database

要连接的数据库

-H host

数据库服务器的主机

-p port

数据库服务器的端口

-U username

用于连接的用户名

-P password

口令（弃用 — 把口令放在命令行上是安全灾难）

要显示特定表，通过使用-o、-f和-t选择要显示哪个表。 -o采用一个 OID， -f采用一个文件节点， 而-t采用一个表名（实际上，它是一个LIKE模式，因此你可以用诸如foo%之类的东西）。 这些选项你想用多少就用多少，最后的列举将包括所有匹配任意一个这些选项的对象。但是注意这些选项只能显示由-d给定的数据库中的对象。

如果你没有给出任何-o、-f或者-t，但是给出了-d，它将列出由-d指定的数据库中的所有表。在这种模式下，-S和-i选项控制什么会被列出。

如果你也没有给出-d，它将显示一个数据库 OID 的列表。你也可以给出-s来得到一个表空间列表。

注解

oid2name要求一个运行着的数据库服务器并且其系统目录没有损坏。因此它对于数据库损坏的情况用处有限。

例子

```
$ # 在这个数据库服务器中到底有什么？
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
```

```
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # OK，让我们看看数据库 alvherre 里面
$ cd $PGDATA/base/17228

$ # 得到默认表空间中前十个数据库对象，按尺寸排序
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # 我好奇文件 155173 是 ...
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # 你可以请求多于一个对象
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
--------------------------
    155173       accounts
   1155291  accounts_pkey

$ # 你可以混合选项，并且用 -x 得到更多细节
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # 为每个数据库对象显示磁盘空间
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # 相同，但是按尺寸排序
$ du [0-9]* | sort -rn | while read SIZE FN
> do
```

```
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561             1155291  accounts_pkey
1177               16717  pg_proc_proname_args_nsp_index
...

$ # 如果你想看看表空间里有什么，使用 pg_tblspc 目录
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # 哪些数据库在表空间 "fastdisk" 中有对象?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # 噢，什么是数据库 17228 ?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # 让我们看这个数据库在该表空间中有哪些对象。
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # OK，这是个很小的表，但是哪一个是它?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

作者

B. Palmer <bpalmer@crimelabs.net>

vacuumlo

vacuumlo — 从PostgreSQL数据库中移除孤立的大对象

大纲

```
vacuumlo [option...] dbname...
```

描述

vacuumlo是一个从PostgreSQL数据库中移除“孤立”大对象的简单使用程序。一个孤立的大对象（LO）是指其OID不出现在数据中任何oid或lo数据列中的LO。

如果你使用该程序，你也许还会对lo模块中的lo_manage触发器感兴趣。lo_manage对于避免创建孤立LO有用处。

在命令行中提到的所有数据库都将被处理。

选项

vacuumlo接受下列命令行参数：

-l limit

在每一个事务中移除不超过limit个大对象（默认值为1000）。因为移除每一个LO时服务器都将要求一个锁，所以在一个事务中移除过多的LO会有超过max_locks_per_transaction的风险。如果你想在一个事务中就完成所有的移除工作，请将这个限制设置为0。

-n

不移除任何东西，只是显示将会做什么。

-v

写一些进度消息。

-V
--version

打印vacuumlo的版本并退出。

-?
--help

显示关于vacuumlo的命令行参数，并且退出。

vacuumlo也接受下列命令行参数用于连接：

-h hostname

数据库服务器的主机名。

-p port

数据库服务器的端口。

-U username

用于连接的用户名。

-w
--no-password

不要发出一个口令提示。如果服务器要求口令认证并且没有其他方式可以提供一个口令（例如一个.pgpass文件），连接尝试将会失败。这个选项可用于批处理任务以及脚本，因为在这些情况下不会有用户输入口令。

-W

强制vacuumlo在连接到数据库之前提示要求一个口令。

这个选项不是不可缺少的，因为如果服务器要求口令认证，vacuumlo会自动提示要求一个口令。但是，vacuumlo将会浪费一次连接尝试来了解到服务器需要口令。在某些情况，值得用-W来避免这种额外的连接尝试。

说明

vacuumlo按照下列方法工作：首先vacuumlo建立一个临时表，其中包含所选择数据库中所有大对象的OID。然后它会扫描数据库中所有类型为oid或lo的列，并且从临时表中移除在这些列中出现过的OID（注意：只有类型为这些名字的才被考虑，特别的，在这些类型之上的域是不被考虑的）。临时表中剩下的项就标识了鼓励LO。它们将被移除。

作者

Peter Mount <peter@retep.org.uk>

G.2. 服务器应用

这一节覆盖了contrib中与PostgreSQL服务器相关的应用。它们通常运行在数据库服务器所在的主机上。核心PostgreSQL发布中的服务器端应用也可见PostgreSQL 服务器应用。

pg_standby

pg_standby — 对创建一个PostgreSQL温备服务器提供支持

大纲

pg_standby [option...] archivelocation nextwalfile walfilepath [restartwalfile]

描述

pg_standby用于支持创建一个“温备”数据库服务器。它被设计为一个可以随时投入生产的程序，以及一个可定制的模板供你进行特定的修改。

pg_standby被设计成一个等待着的restore_command，它被用来把一次标准的归档恢复变成一次温备操作。还需要一些其他的配置，所有这些配置都在主服务器手册中有相应的描述（见第 26.2 节。

要配置一台后备服务器去使用pg_standby，可以把下面的内容放在recovery.conf配置文件中：

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

其中archiveDir是 WAL 段文件应该被存储的目录。

如果指定了restartwalfile（通常用%r宏指定），那么所有在逻辑上位于这个文件之前的 WAL 文件都将被从archivelocation中移除。这使需要被保留的文件数最小化，与此同时能够保持崩溃重启的能力。如果archivelocation对于这个特定后备服务器是一个瞬态区域，使用这个参数是合适的，但当archivelocation是一个长期 WAL 归档区域时则不是合适的。

pg_standby假定archivelocation是一个拥有服务器的用户可读的目录。如果指定了restartwalfile（或者-k），archivelocation目录必须也是可写的。

当主服务器失效时，有两种方式转移到一个“温备”数据库服务器：

智能失效备援

在智能失效备援中，服务器在应用归档中可用的所有 WAL 文件之后被提升。即便后备服务器落后于主服务器，这也会导致零数据丢失，但是如果有很多未应用的 WAL，在后备服务器准备好之前就需要比较长的时间。要触发一次智能失效备援，创建一个包含单词smart的触发文件，或者只创建一个空文件。

快速失效备援

在快速失效备援中，服务器被立即提升。归档中任何还未被应用的 WAL 文件将被忽略，并且这些文件中的所有事务都会丢失。要触发一次快速失效备援，创建一个触发文件并且把单词fast写在其中。pg_standby也能被配置为在一段定义好的区间内没有新 WAL 文件出现时自动执行一次快速失效备援。

选项

pg_standby接受下列命令行参数：

-c

使用cp或者copy命令来存储来自归档的 WAL 文件。这是唯一支持的行为，因此这个选项是无用的。

-d

在stderr上打印大量调试日志输出。

-k

从archivelocation移除文件，这样当前 WAL 文件之前不超过这么多个 WAL 文件会被保留在归档中。零（默认值）意味着不从archivelocation移除任何文件。如果指定了restartwalfile，这个参数将被安静地忽略，因为那种说明方法对于决定正确的归档切断点更为精确。从PostgreSQL 8.3 开始，这个参数的使用已经被废弃，指定一个restartwalfile参数更加安全和有效。一个太小的值可能导致后备服务器的重启仍需要已经被移除的文件，而一个太大的值则浪费归档空间。

-r maxretries

设定拷贝命令失败时重试的最大次数（默认为 3）。在每一次失败后，我们等待sleeptime * num_retries，这样等待时间会逐步增加。因此默认情况下，我们将等待 5 秒、10 秒、15秒，然后向后备服务器报告失败。这将被解释为恢复的终点并且该后备服务器将会完全被提升。

-s sleeptime

设置在检查要被恢复的 WAL 文件在归档中是否可用的测试之间休眠的秒数（最高 60，默认为 5）。默认设置并不一定值得推荐，相关的讨论可参考第 26.2 节

-t triggerfile

指定一个触发文件，它的出现将会导致失效备援。我们推荐使用一个有结构的文件名，这样可以避免在同一个系统上有多个服务器存在时无法确定是要触发哪个服务器。例如可以用/tmp/pgsql.trigger.5432。

-V
--version

打印pg_standby版本并退出。

-w maxwaittime

设置等待下一个 WAL 文件的最大秒数，之后将会执行一次快速失效备援。设置为 0 （默认值）表示永远等待。默认设置并不一定值得推荐，相关讨论可以参考第 26.2 节

-?
--help

显示有关pg_standby的命令行参数，然后退出。

注解

pg_standby被设计为配合PostgreSQL 8.2 及后续版本工作。

PostgreSQL 8.3 提供%r宏，它被设计用来让pg_standby知道它需要保留的最后一个文件。在 PostgreSQL 8.2 中，如果要求归档清理则必须使用-k选项。在 8.3 中该选项仍然存在，但是其使用已被废弃。

PostgreSQL 8.4 提供recovery_end_command选项。没有这个选项一个残留的触发文件能够导致灾难。

pg_standby由 C 写成并且其代码易于修改，其中的小节经过特别设计以便按你的需要修改。

例子

在 Linux 或 Unix 系统上，你可能会使用：

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f
 %p %r 2>>standby.log'

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

其中归档目录被物理防止在后备服务器上，因此archive_command通过 NFS 访问它，但是那些文件对于后备服务器是本地的（启用ln）。这将：

- 在standby.log中产生调试输出
- 在检查下一个 WAL 文件的可用性之间睡眠 2 秒
- 只有当一个名为/tmp/pgsql.trigger.5442的触发文件出现时停止等待并且根据其内容执行失效备援
- 在恢复结束时移除触发文件
- 从归档目录中移除不再需要的文件

在 Windows 上，你可能会用：

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p
 %r 2>>standby.log'

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

注意archive_command中的反斜线需要被双写，但是在restore_command或者recovery_end_command中不需要。这将：

- 使用copy命令恢复来自归档的 WAL 文件
- 在standby.log中产生调试输出
- 在检查下一个 WAL 文件的可用性之间睡眠 5 秒
- 只有当一个名为C:\pgsql.trigger.5442的触发文件出现时停止等待并且根据其内容执行失效备援
- 在恢复结束时移除触发文件
- 从归档目录中移除不再需要的文件

Windows 上的copy命令在文件被完全拷贝之前就会设置最终的文件尺寸，这通常会迷惑pg_standby。因此一旦看到正确的文件尺寸，pg_standby会等待sleeptime秒。GNUWin32的cp只会在文件拷贝完成后设置文件尺寸。

由于 Windows 的例子在两端都使用copy，一个或者两个服务器可能通过网络访问归档目录。

作者

Simon Riggs <simon@2ndquadrant.com>

参见

pg_archivecleanup

附录 H. 外部项目

PostgreSQL是一项复杂的软件项目，管理它是一项困难的工作。我们发现许多PostgreSQL的提高可以通过独立于核心项目的方式更有效地开发。

H.1. 客户端接口

在基本的PostgreSQL发布中仅包含两种客户端接口：

- libpq被包括的原因是它是主要的 C 语言接口，并且许多其它客户端接口都是在它的基础上构建的。
- ECPG被包括的原因是它依赖于服务器端 SQL 语法，并且因此对PostgreSQL自身的变化非常敏感。

除此以外的所有其它语言接口都是外部项目并独立发布。表 H.1包括了其中一些项目的列表。 需要注意的是其中一些包的发布许可证与PostgreSQL不同。要了解更多关于每种语言的接口细节（包括许可证条款），请参考它的网站和文档。

表 H.1. 外部维护的客户端接口

名字	语言	注释	网站
DBD::Pg	Perl	Perl DBI 驱动	https://metacpan.org/release/DBD-Pg
JDBC	Java	Type 4 JDBC 驱动	https://jdbc.postgresql.org/
libpqxx	C++	C++ 接口	http://pqxx.org/
node-postgres	JavaScript	Node.js 驱动器	https://node-postgres.com/
Npgsql	.NET	.NET 数据提供器	http://www.npgsql.org/
pgtcl	Tcl		https://github.com/flightaware/Pgtcl
pgtclng	Tcl		https://sourceforge.net/projects/pgtclng/
pq	Go	Go的数据库/sql的Pure Go驱动程序	https://github.com/lib/pq
psqlODBC	ODBC	ODBC 驱动	https://odbc.postgresql.org/
psycopg	Python	DB API 2.0兼容	http://initd.org/psycopg/

H.2. 管理工具

有多个管理工具可用于PostgreSQL。最流行的是pgAdmin[1]，并且也有一些商业版的可用。

H.3. 过程语言

[1] https://www.pgadmin.org/

PostgreSQL在基本发布中包括了多种过程语言：PL/pgSQL、PL/Tcl、PL/Perl和PL/Python。

此外，还有一些过程语言是在核心PostgreSQL发布之外被开发和维护的。表 H.2列出了其中的一些包。注意其中某些项目的发行许可证与PostgreSQL不同。要了解每种过程语言的更多信息（包括许可证信息），请参考它的网站和文档。

表 H.2. 外部维护的过程语言

名称	语言	网站
PL/Java	Java	https://tada.github.io/pljava/
PL/Lua	Lua	https://github.com/pllua/pllua
PL/R	R	http://www.joeconway.com/plr.html
PL/sh	Unix shell	https://github.com/petere/plsh
PL/v8	JavaScript	https://github.com/plv8/plv8

H.4. 扩展

PostgreSQL被设计成很容易扩展。由于这个原因，被载入到数据库中的扩展可以像内建特性那样运行。contrib/目录中包含了多个扩展的源码，在附录 F中有相关介绍。其他扩展的开发是独立进行的，例如PostGIS[2]。甚至PostgreSQL的复制解决方案也能在外部开发。例如，Slony-I[3]是一个在核心项目之外独立开发的流行的主/备复制解决方案。

[2] http://postgis.net/
[3] http://www.slony.info

附录 I. 源代码仓库

PostgreSQL源代码使用Git版本控制系统存储和管理。有一个主仓库的公共镜像可用，它每分钟都会根据主仓库的任何改变而更新。

我们的维基（https://wiki.postgresql.org/wiki/Working_with_Git）上有使用 Git 工作的一些讨论。

注意从源代码仓库编译PostgreSQL要求合理的最新版本的 bison、flex和Perl。 在从一个发布的 tarball 中编译时不需要这些工具，因为它们需要去编译的文件已经被包括在 tarball 中。 其他工具要求和第 16 章显示的一样。

I.1. 通过Git得到源码

使用Git你将可以在你的本地机器上创建一份整个代码库的拷贝，这样你将能够离线访问所有历史和分支。这是开发或测试补丁最快也是最灵活的方式。

Git

1. 你将需要一个已安装的Git版本，你可以从https://git-scm.com得到。很多系统已经默认安装了一个最近版本的Git，或者可以从它们的包发布系统中找到。

2. 要开始使用 Git 仓库，先做一个官方镜像的克隆：

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

   这将复制整个仓库到你的本地机器上，这样它会花上一会儿来完成（特别是如果你使用的互联网连接速度较慢）。文件将被放在你当前目录下的一个新子目录postgresql中。

   Git 镜像也可以通过 Git 协议访问。只需要将 URL 前缀改成git：

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

3. 不管什么时候你希望得到系统的最新更新，cd进入仓库并且运行：

   ```
   git fetch
   ```

除了取得源码，Git可以做更多事情。要了解更多，请参考Git手册页，或者访问网站https://git-scm.com。

附录 J. 文档

PostgreSQL有四种主要的文档格式：

- 纯文本，用于安装前信息
- HTML，用于在线浏览和参考
- PDF，用于打印
- 手册页，用于快速参考

另外，在PostgreSQL源码树里面还有许多纯文本风格的README文件，它们记录各种实现问题。

HTML文档和手册页是标准发布的一部分并且被默认安装。PDF格式的文档可以独立地下载。

J.1. DocBook

文档源码是用DocBook编写的，它是一种用XML定义的标记语言。在下文中，虽然术语 DocBook 和XML都被使用，但在技术上它们是不能互换的。

DocBook允许作者指定一份技术文档的结构和内容，而不需要关心表现的细节。一份文档风格定义其内容如何呈现为几种最终形式之一。DocBook 由OASIS group[1]工作组维护。官方 DocBook 站点[2]有很好的介绍和参考文档以及一整本 O'Reilly 的书可供你在线阅读。NewbieDoc Docbook Guide[3]非常适合初学者。 FreeBSD 文档项目[4]也使用 DocBook 并且有一些很好 的信息，包括一些很值得考虑的风格参考。

J.2. 工具集

下面的工具用于处理此文档。如标注所示，有些工具可能是可选的。

DocBook DTD[5]

这是 DocBook 本身的定义。我们目前使用版本 4.2；你不能使用更新或者更早的版本。你需要 DocBook DTD 的XML变体，而不是SGML变体。

DocBook XSL Stylesheets[6]

这些包含了将 DocBook 源码转换到其他格式（如HTML）所要用的处理指令。

最低要求的版本当前是 1.77.0，但推荐使用最新可用的版本来得到最好的结果。

xmllint的Libxml2[7]

这个库和它所包含的xmllint工具被用来处理 XML。 很多开发者将已经安装了Libxml2，因为在编译 PostgreSQL 代码时就已经用到它了。不过要注意，可能需要从一个独立的包中 安装xmllint。

xsltproc的Libxslt[8]

xsltproc是一个XSLT处理器，也就是说它是一个使用XSLT将XML转换成其他格式的程序。

[1] https://www.oasis-open.org
[2] https://www.oasis-open.org/docbook/
[3] http://newbiedoc.sourceforge.net/metadoc/docbook-guide.html
[4] https://www.freebsd.org/docproj/docproj.html
[5] https://www.oasis-open.org/docbook/
[6] https://github.com/docbook/wiki/wiki/DocBookXslStylesheets
[7] http://xmlsoft.org/
[8] http://xmlsoft.org/XSLT/

FOP[9]

这是一个在其他东西中将XML转换为PDF的程序。

我们已经在文档中记录了几种安装处理此文档所需的各种工具的方法。它们将在下文中描述。也可能有这些工具的其他打包发布。请向文档邮件列表报告那些包的状态，我们就会在这里包括它们的信息。

你可以在本地不安装DocBook XML和DocBook XSLT样式表的情况下开始工作，因为所需的文件将从Internet下载下来并且缓存在本地。如果你的操作系统包仅提供了旧版本的样式表或者根本就没有所需的包，那么这种方式实际上是最好的解决方案。更多信息请见xmllint和xsltproc的--nonet选项。

J.2.1. 在 Fedora、RHEL 和衍生品上安装

要安装所需的包，可使用：

```
yum install docbook-dtds docbook-style-xsl fop libxslt
```

J.2.2. 在 FreeBSD 上安装

要用pkg安装所需的包，可以使用：

```
pkg install docbook-xml docbook-xsl fop libxslt
```

在从doc目录构建文档时，你会需要用到gmake，因为所提供的makefile不适合于FreeBSD的make。

J.2.3. Debian 包

Debian GNU/Linux也有一整套可以用的文档工具包。要安装，只需简单地使用：

```
apt-get install docbook-xml docbook-xsl fop libxml2-utils xsltproc
```

J.2.4. macOS

在macOS上，你可以不安装任何额外的东西就编译HTML和man文档。如果你想要编译PDF或者想要安装DocBook的本地拷贝，可以采用你喜欢的包管理器来得到它们。

如果使用MacPorts，下面的命令会帮你准备好一切：

```
sudo port install docbook-xml-4.2 docbook-xsl fop
```

如果使用的是Homebrew，就用这个：

```
brew install docbook docbook-xsl fop
```

J.2.5. 用configure检测

在你能编译文档之前，你需要运行configure脚本，就像你在编译PostgreSQL程序本身时所作的那样。检查运行末尾附近的输出，应该看起来像这样：

[9] https://xmlgraphics.apache.org/fop/

```
checking for xmllint... xmllint
checking for DocBook XML V4.2... yes
checking for dbtoepub... dbtoepub
checking for xsltproc... xsltproc
checking for fop... fop
```

如果没有找到xmllint，那么一些后续测试将被跳过。

J.3. 编译文档

一旦你把所有的东西都设置好以后，切换到doc/src/sgml目录，并且运行下面小节中介绍的命令之一就可以编译文档（记住使用 GNU make）。

J.3.1. HTML

要编译文档的HTML版本：

```
doc/src/sgml$ make html
```

这也是默认的目标。这个命令的输出将出现在子目录html中。

要用postgresql.org[10]所使用的样式表 而不是默认的简单样式生成 HTML 文档：

```
doc/src/sgml$ make STYLE=website html
```

J.3.2. 手册页

我们使用 DocBook XSL 样式表来把DocBook refentry页转换成适合于手册页的 *roff 输出。要创建手册页，使用命令：

```
doc/src/sgml$ make man
```

J.3.3. PDF

要使用FOP产生文档的PDF版本，可以使用下列命令之一，取决于你喜欢的纸张格式：

- A4格式：

  ```
  doc/src/sgml$ make postgres-A4.pdf
  ```

- U.S.信纸格式：

  ```
  doc/src/sgml$ make postgres-US.pdf
  ```

因为PostgreSQL文档很大，FOP会要求可观的内存量。因此，在一些系统上，构建过程将会由于内存相关的错误而失败。通常可以通过在配置文件~/.foprc中配置Java的堆设置来解决这类问题，例如：

```
# FOP binary distribution
FOP_OPTS='-Xmx1500m'
```

[10] https://www.postgresql.org/docs/current

```
# Debian
JAVA_ARGS='-Xmx1500m'
# Red Hat
ADDITIONAL_FLAGS='-Xmx1500m'
```

这是所要求的最小内存量，当然更多的内存会让编译过程更快一些。在内存非常小（小于1GB）的系统上，编译过程会因为磁盘交换而非常慢或者根本就不工作。

也可以手工使用其他XSL-FO处理器，但是自动编译过程仅支持FOP。

J.3.4. 纯文本文件

安装指导也被发布为纯文本，它们被用于那些没有好的阅读工具的情况。INSTALL文件对应于第 16 章但针对不同的环境做了小幅修改。要重建该文件，切换到目录doc/src/sgml并输入make INSTALL。

在过去，发行注记和回归测试指导也被作为纯文本发布，但是事实上已经没有这样做了。

J.3.5. 语法检查

编译文档可能会花很长时间。但是有办法只检查文档中的语法，这个过程只需要数秒：

```
doc/src/sgml$ make check
```

J.4. 文档创作

文档的源码可以很方便地用具有XML编辑模式的编辑器来修改，而如果它们懂一些XML模式语言那就更好了，那样它们就能理解专门的DocBook语法。

注意由于历史原因，文档源文件被命名为以扩展名.sgml结尾，尽管它们现在都是XML文件。因此你可能需要调整你的编辑器配置来设置正确的模式。

J.4.1. Emacs

Emacs中自带的nXML Mode是用Emacs编辑XML文档最常用的模式。它允许用户使用Emacs插入标签并且检查标记的一致性，它还自带对DocBook的支持。详细的文档请参考nXML manual[11]。

src/tools/editors/emacs.samples包含了这种模式的推荐设置。

J.5. 样式指导

J.5.1. 参考页

参考页应该遵循一种标准布局。这允许用户更快找到想要的信息，并且它也鼓励作者记录一个命令的所有相关方面。不止在PostgreSQL参考页中需要一致性，操作系统或其他包提供的参考页也需要。因此人们开发了下列方针。它们的大部分与由多个操作系统建立的类似方针保持一致。

描述可执行命令的参考页应该包含以下的小节，并且是按照介绍的顺序。不适用的小节可以被忽略。额外的顶层小节应该只被用在特殊的环境下；通常那些信息属于“Usage”小节。

名称

这个小节是自动生成的。它包含命令名称和对其功能的一个半句话摘要。

[11] https://www.gnu.org/software/emacs/manual/html_mono/nxml-mode.html

概要

这个小节包含该命令的语法表。概要通常应该不列出每个命令行选项；那些东西由后续小节介绍。概要应该列出命令行的主要部分，比如输入和输出文件会到哪里去。

描述

解释命令干什么的几个段落。

选项

一个描述每一个命令行选项的列表。如果有很多选项，可以使用子节。

退出状态

如果程序用 0 表示成功，非零表示失败，那么你不需要为此写文档。如果在每个非零退出码有不同的含义，那么在这里列出它们。

用法

描述程序的任意子语言或者运行时接口。如果程序不是交互式的，那么本节通常可以省略。否则，本节是全方位描述运行时特性的地方。如果需要可以使用子小节。

环境

列出所有程序可能使用的环境变量。尽量完整；即使是那些看起来很琐碎的变量，比如SHELL都可能让读者感兴趣。

文件

列出程序可能隐式访问的任何文件。也就是说，不要列出在命令行上指定的输入和输出文件，但是要列出配置文件等。

诊断

解释程序可能生成的任何不正常的输出。避免列出所有可能的错误消息。 这样做工作量很大但没有太多实际用处。但是如果错误消息有一种用户可以解析的标准格式，那么它应当在这里解释。

注意

任何放在其他地方都不合适的东西可以放在这里，但是特别是缺陷、实现瑕疵、安全考虑、兼容性问题。

例子

例子

历史

如果在程序的历史上有一些主要的里程碑，那么可以在这里列出。通常，这个小节可以被省略。

作者

作者（只在 contrib 节中使用）

另见

交叉引用，按照下面的顺序列出：其它PostgreSQL命令参考页、PostgreSQL SQL命令参考页、引用PostgreSQL手册、 其它参考页（如操作系统、其它包）、其它文档。在同一组中的项按照字母表顺序列出。

描述 SQL 命令的参考页应该包含下列小节：名称、概要、描述、参数、输出、注意、例子、兼容性、历史、另见。　用法、诊断、注意、例子、兼容性、历史、又见。参数小节类似选项小节，　但是在选择哪些子句是可列出的方面有更多自由。输出小节只有在命令返回非命令结束标签的东西时才需要。兼容性小节应该解释此命令遵循 SQL 标准的程度，或者它兼容哪种其它数据库系统。SQL 命令的另见小节应该在交叉引用其它程序之前列出 SQL 命令。

附录 K. 首字母缩写词

在PostgreSQL的文档和讨论中有很多常用的首字母缩写词。

ANSI

American National Standards Institute[1]

API

Application Programming Interface[2]

ASCII

American Standard Code for Information Interchange[3]

BKI

Backend Interface

CA

Certificate Authority[4]

CIDR

Classless Inter-Domain Routing[5]

CPAN

Comprehensive Perl Archive Network[6]

CRL

Certificate Revocation List[7]

CSV

Comma Separated Values[8]

CTE

Common Table Expression

CVE

Common Vulnerabilities and Exposures[9]

DBA

Database Administrator[10]

[1] https://en.wikipedia.org/wiki/American_National_Standards_Institute
[2] https://en.wikipedia.org/wiki/API
[3] https://en.wikipedia.org/wiki/Ascii
[4] https://en.wikipedia.org/wiki/Certificate_authority
[5] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[6] https://www.cpan.org/
[7] https://en.wikipedia.org/wiki/Certificate_revocation_list
[8] https://en.wikipedia.org/wiki/Comma-separated_values
[9] http://cve.mitre.org/
[10] https://en.wikipedia.org/wiki/Database_administrator

DBI

Database Interface (Perl)[11]

DBMS

Database Management System[12]

DDL

Data Definition Language[13], SQL commands such as CREATE TABLE, ALTER USER

DML

Data Manipulation Language[14], SQL commands such as INSERT, UPDATE, DELETE

DST

Daylight Saving Time[15]

ECPG

Embedded C for PostgreSQL

ESQL

Embedded SQL[16]

FAQ

Frequently Asked Questions[17]

FSM

Free Space Map

GEQO

Genetic Query Optimizer

GIN

Generalized Inverted Index

GiST

Generalized Search Tree

Git

Git[18]

GMT

Greenwich Mean Time[19]

[11] https://dbi.perl.org/
[12] https://en.wikipedia.org/wiki/Dbms
[13] https://en.wikipedia.org/wiki/Data_Definition_Language
[14] https://en.wikipedia.org/wiki/Data_Manipulation_Language
[15] https://en.wikipedia.org/wiki/Daylight_saving_time
[16] https://en.wikipedia.org/wiki/Embedded_SQL
[17] https://en.wikipedia.org/wiki/FAQ
[18] https://en.wikipedia.org/wiki/Git_(software)
[19] https://en.wikipedia.org/wiki/GMT

GSSAPI

Generic Security Services Application Programming Interface[20]

GUC

Grand Unified Configuration, the PostgreSQL subsystem that handles server configuration

HBA

Host-Based Authentication

HOT

Heap-Only Tuples[21]

IEC

International Electrotechnical Commission[22]

IEEE

Institute of Electrical and Electronics Engineers[23]

IPC

Inter-Process Communication[24]

ISO

International Organization for Standardization[25]

ISSN

International Standard Serial Number[26]

JDBC

Java Database Connectivity[27]

JIT

Just-in-Time compilation[28]

JSON

JavaScript Object Notation[29]

LDAP

Lightweight Directory Access Protocol[30]

[20] https://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface
[21] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/access/heap/README.HOT;hb=HEAD
[22] https://en.wikipedia.org/wiki/International_Electrotechnical_Commission
[23] http://standards.ieee.org/
[24] https://en.wikipedia.org/wiki/Inter-process_communication
[25] https://www.iso.org/home.html
[26] https://en.wikipedia.org/wiki/Issn
[27] https://en.wikipedia.org/wiki/Java_Database_Connectivity
[28] https://en.wikipedia.org/wiki/Just-in-time_compilation
[29] http://json.org
[30] https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

LSN

Log Sequence Number, see pg_lsn and WAL Internals.

MSVC

Microsoft Visual C[31]

MVCC

Multi-Version Concurrency Control

NLS

National Language Support[32]

ODBC

Open Database Connectivity[33]

OID

Object Identifier

OLAP

Online Analytical Processing[34]

OLTP

Online Transaction Processing[35]

ORDBMS

Object-Relational Database Management System[36]

PAM

Pluggable Authentication Modules[37]

PGSQL

PostgreSQL

PGXS

PostgreSQL Extension System

PID

Process Identifier[38]

PITR

Point-In-Time Recovery (Continuous Archiving)

[31] https://en.wikipedia.org/wiki/Visual_C++
[32] https://en.wikipedia.org/wiki/Internationalization_and_localization
[33] https://en.wikipedia.org/wiki/Open_Database_Connectivity
[34] https://en.wikipedia.org/wiki/Olap
[35] https://en.wikipedia.org/wiki/OLTP
[36] https://en.wikipedia.org/wiki/ORDBMS
[37] https://en.wikipedia.org/wiki/Pluggable_Authentication_Modules
[38] https://en.wikipedia.org/wiki/Process_identifier

PL

Procedural Languages (server-side)

POSIX

Portable Operating System Interface[39]

RDBMS

Relational Database Management System[40]

RFC

Request For Comments[41]

SGML

Standard Generalized Markup Language[42]

SPI

Server Programming Interface

SP-GiST

Space-Partitioned Generalized Search Tree

SQL

Structured Query Language[43]

SRF

Set-Returning Function

SSH

Secure Shell[44]

SSL

Secure Sockets Layer[45]

SSPI

Security Support Provider Interface[46]

SYSV

Unix System V[47]

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)[48]

[39] https://en.wikipedia.org/wiki/POSIX
[40] https://en.wikipedia.org/wiki/Relational_database_management_system
[41] https://en.wikipedia.org/wiki/Request_for_Comments
[42] https://en.wikipedia.org/wiki/SGML
[43] https://en.wikipedia.org/wiki/SQL
[44] https://en.wikipedia.org/wiki/Secure_Shell
[45] https://en.wikipedia.org/wiki/Secure_Sockets_Layer
[46] https://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx
[47] https://en.wikipedia.org/wiki/System_V
[48] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

TID

Tuple Identifier

TOAST

The Oversized-Attribute Storage Technique

TPC

Transaction Processing Performance Council[49]

URL

Uniform Resource Locator[50]

UTC

Coordinated Universal Time[51]

UTF

Unicode Transformation Format[52]

UTF8

Eight-Bit Unicode Transformation Format[53]

UUID

Universally Unique Identifier

WAL

Write-Ahead Log

XID

Transaction Identifier

XML

Extensible Markup Language[54]

[49] http://www.tpc.org/
[50] https://en.wikipedia.org/wiki/URL
[51] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[52] http://www.unicode.org/
[53] https://en.wikipedia.org/wiki/Utf8
[54] https://en.wikipedia.org/wiki/XML

参考书目

一些有关SQL和PostgreSQL的参考和读物。

一些来自于最初的POSTGRES开发团队的白皮书和技术报告可以在加州大学伯克利分校计算机系的网站[1]上找到。

SQL参考书

[bowman01] The Practical SQL Handbook. Using SQL Variants. Fourth Edition. Judith Bowman、Sandra Emerson和Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] A Guide to the SQL Standard. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date和Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] An Introduction to Database Systems. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] Fundamentals of Database Systems. Fourth Edition. Ramez Elmasri和Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] Understanding the New SQL. A complete guide. Jim Melton和Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] Principles of Database and Knowledge. Base Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

PostgreSQL相关文档

[sim98] Enhancement of the ANSI SQL Implementation of PostgreSQL. Stefan Simkovics. Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.

[yu95] The Postgres95. User Manual. A. Yu和J. Chen. University of California. Berkeley, California. Sept. 5, 1995.

[fong] The design and implementation of the POSTGRES query optimizer[2]. Zelaine Fong. University of California, Berkeley, Computer Science Department.

会议记录和期刊论文

[olson93] Partial indexing in POSTGRES: research project. Nels Olson. UCB Engin T7.49.1993 O676. University of California. Berkeley, California. 1993.

[ong90] “A Unified Framework for Version Modeling Using Production Rules in a Database System”. L. Ong和J. Goh. ERL Technical Memorandum M90/33. University of California. Berkeley, California. April, 1990.

[rowe87] “The POSTGRES data model[3]”. L. Rowe和M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] “Generalized Partial Indexes[4]”. P. Seshadri和A. Swami. Eleventh International Conference on Data Engineering, 6-10 March 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420-7.

[1] http://db.cs.berkeley.edu/papers/
[2] http://db.cs.berkeley.edu/papers/UCB-MS-zfong.pdf
[3] http://db.cs.berkeley.edu/papers/ERL-M87-13.pdf
[4] http://citeseer.ist.psu.edu/seshadri95generalized.html

[ston86] “The design of POSTGRES[5]”. M. Stonebraker和L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] “The design of the POSTGRES. rules system”. M. Stonebraker、E. Hanson和C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] “The design of the POSTGRES storage system[6]”. M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] “A commentary on the POSTGRES rules system[7]”. M. Stonebraker、M. Hearst和S. Potamianos. SIGMOD Record 18(3). Sept. 1989.

[ston89b] “The case for partial indexes[8]”. M. Stonebraker. SIGMOD Record 18(4). Dec. 1989. 4-11.

[ston90a] “The implementation of POSTGRES[9]”. M. Stonebraker、L. A. Rowe和M. Hirohama. Transactions on Knowledge and Data Engineering 2(1). IEEE. March 1990.

[ston90b] “On Rules, Procedures, Caching and Views in Database Systems[10]”. M. Stonebraker、A. Jhingran、J. Goh和S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

[5] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[6] http://db.cs.berkeley.edu/papers/ERL-M87-06.pdf
[7] http://db.cs.berkeley.edu/papers/ERL-M89-82.pdf
[8] http://db.cs.berkeley.edu/papers/ERL-M89-17.pdf
[9] http://db.cs.berkeley.edu/papers/ERL-M90-34.pdf
[10] http://db.cs.berkeley.edu/papers/ERL-M90-36.pdf

索引

符号

A

B

C

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z