PostgreSQL

PostgreSQL

Covering v7.0 for general release

目录

插图清单

表格清单

范例清单

摘要

Postgres 最初由 UC Berkeley 计算机科学系开发，它开创了如今在一些商业数据库中逐渐可用的许多对象-关系概念。它提供 SQL92/SQL3 语言支持、事务完整性和类型可扩展性。PostgreSQL 是这一原始 Berkeley 代码的开源后裔。

部分 I. 用户指南

用户信息。

目录

第 1 章 引言

本文档是最初在加州大学伯克利分校开发的 PostgreSQL[1] 数据库管理系统的用户手册。PostgreSQL基于 Postgres release 4.2[2]。Postgres项目由 Michael Stonebraker 教授领导，得到了国防高级研究计划局（DARPA）、陆军研究办公室（ARO）、国家科学基金会（NSF）以及 ESL 公司的赞助。

1.1. 什么是Postgres?

传统的关系数据库管理系统（DBMSs）支持的数据模型由一组命名的关系组成，
关系中包含特定类型的属性。在当前的商业系统中，可用的类型包括浮点数、整数、字符串、
货币和日期。人们普遍认识到，这个模型对于未来的数据处理应用是不够的。
关系模型之所以能够成功地取代以前的模型，部分原因在于它的"斯巴达式的简洁"。然而，
如前所述，这种简洁性常使某些应用的实现变得非常困难。
Postgres通过以下面这种方式纳入下列四个附加基本概念，提供了可观的额外能力，
使用户可以轻松地扩展系统：

类
继承
类型
函数

其他特性提供了额外的能力和灵活性：

约束
触发器
规则
事务完整性

这些特性使Postgres属于被称为对象-关系的数据库类别。注意，这不同于那些被称为
面向对象的数据库，后者通常不太适合支持传统的关系数据库语言。因此，尽管 Postgres有一些
面向对象的特性，它仍然坚定地站在关系数据库的世界中。事实上，一些商业数据库最近纳入了
Postgres开创的特性。

1.2. Postgres简史

如今名为 Postgres（并曾短暂称为 Postgres95）的对象关系数据库管理系统，源自伯克利编写的 Postgres 软件包。经过十多年的发展，Postgres 已成为当今任何地方都能获得的最先进的开源数据库，它提供多版本并发控制，支持几乎所有 SQL 结构（包括子查询、事务以及用户定义的类型和函数），并有广泛的语言绑定可用（包括 C、C++、Java、perl、tcl 和 python）。

1.2.1. 伯克利 Postgres 项目

Postgres DBMS 的实现始于 1986 年。该系统的最初概念见于 The Design of Postgres，而最初数据模型的定义见于 Rowe and Stonebraker, 1987。当时规则系统的设计见于 The Design of the Postgres Rules System。存储管理器的设计动机和体系结构详见 Stonebraker, 1987。

自那以后，Postgres 经历了几次重大版本发布。第一个"演示版"系统于 1987 年开始运行，并在 1988 年的 ACM-SIGMOD 大会上进行了展示。版本 1 在 The Implementation of Postgres 中有描述，并于 1989 年 6 月发布给少数外部用户。针对第一代规则系统所受到的批

[1] http://postgresql.org/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

评（A Commentary on the Postgres Rules System），规则系统被重新设计（On Rules, Procedures, Caching and Views in Database Systems），版本 2 连同新的规则系统于 1990 年 6 月发布。版本 3 于 1991 年问世，增加了对多个存储管理器的支持、改进后的查询执行器以及重写后的规则系统。此后直到 Postgres95（见下文）之前的后续版本，主要聚焦于可移植性和可靠性。

Postgres 被用于实现许多不同的研究和生产应用。这些应用包括金融数据分析系统、喷气发动机性能监测软件包、小行星跟踪数据库、医疗信息数据库以及若干地理信息系统。Postgres 还在多所大学中被用作教学工具。最后，Illustra Information Technologies[3]（后来并入 Informix[4]）接手了这套代码并将其商业化。在 1992 年末，Postgres 成为Sequoia 2000 科学计算项目[5]的主要数据管理器。

1993 年期间，外部用户社区的规模几乎翻了一番。人们越来越清楚地意识到，对原型代码的维护和支持工作占用了大量本应用于数据库研究的时间。为了减轻这一支持负担，该项目在版本 4.2 时正式结束。

1.2.2. Postgres95

1994 年，Andrew Yu[6] 和 Jolly Chen[7] 为 Postgres 添加了一个 SQL 语言解释器。随后，它以 Postgres95 这一新名称发布到网络上，作为最初的 Postgres 伯克利代码的开源后继者自行发展。

Postgres95的代码完全采用 ANSI C 编写，并将体积缩减了 25%。许多内部更改提升了性能和可维护性。Postgres95 v1.0.x 版本在 Wisconsin Benchmark 上快了大约 30-50%，比较对象是 Postgres v4.2。除了错误修复之外，这些就是主要的增强：

- 查询语言 Postquel 被 SQL 所取代（在服务器中实现）。直到 PostgreSQL（见下文）才支持子查询，但在 Postgres95 中可以用用户定义的 SQL 函数来模拟。聚合函数也得到了重新实现，并增加了对 GROUP BY 查询子句的支持。`libpq` 接口仍然可用于 C 程序。
- 除了 monitor 程序之外，还提供了一个用于交互式 SQL 查询的新程序（psql），它使用了 GNU `readline`。
- 新增的前端库 `libpgtcl` 支持基于 Tcl 的客户端。示例 shell pgtclsh 提供了新的 Tcl 命令，用于让 tcl 程序与 Postgres95 服务器交互。
- 大对象接口经过了全面改造。反转大对象成为存储大对象的唯一机制。（反转文件系统已被移除。）
- 实例级规则系统被移除。规则仍然可以作为重写规则使用。
- 源代码中附带了一份简短教程，介绍常规 SQL 特性以及 Postgres95 的特性。
- 构建时使用的是 GNU make（而不是 BSD make）。此外，Postgres95 可以用未经修改的 gcc 编译（双精度浮点数的数据对齐问题已修复）。

1.2.3. PostgreSQL

到了 1996 年，很明显 "Postgres95" 这个名字经不起时间的考验。我们选择了一个新名字 PostgreSQL，以体现最初的 Postgres 与后来具备 SQL 能力的版本之间的关系。同时，我们将版本号从 6.0 开始，使编号重新回到最初由伯克利 Postgres 项目开启的序列中。

[3] http://www.illustra.com/
[4] http://www.informix.com/
[5] http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html
[6] mailto:ayu@informix.com
[7] http://http.cs.berkeley.edu/~jolly/

在开发 Postgres95 期间，重点是识别并理解服务器代码中已有的问题。到了 PostgreSQL 阶段，重点则转向增强特性和能力，不过各个方面的工作仍在继续。

PostgreSQL 中的主要增强包括：

- 表级锁已被多版本并发控制取代，它允许读取者在写入者活动期间继续读取一致的数据，并使得在数据库保持可查询的同时，可以用 pg_dump 进行热备份。
- 实现了重要的后端特性，包括子查询、默认值、约束和触发器。
- 增加了额外的符合 SQL92 的语言特性，包括主键、带引号的标识符、字面串类型强制转换、类型转换以及二进制和十六进制整数输入。
- 内置类型得到了改进，包括新的大范围日期/时间类型和额外的几何类型支持。
- 后端代码的整体速度提升了大约 20-40%，而且自 v6.0 发布以来，后端启动时间缩短了 80%。

1.3. 关于本发行版

PostgreSQL 是免费提供的。本手册描述的是 PostgreSQL 的 7.0 版。

我们把 Postgres 用作以 PostgreSQL 名称发行的版本的简称。

当前支持的机器列表请查阅管理员指南。一般来说，Postgres 可以移植到任何具有完整 libc 库支持的 Unix/Posix 兼容系统。

1.4. 资源

本手册集被组织为若干部分：

教程

面向新用户的介绍。不涉及高级特性。

用户指南

面向用户的一般信息，包括可用的命令和数据类型。

程序员指南

面向应用程序员的高级信息。主题包括类型和函数可扩展性、库接口以及应用设计问题。

管理员指南

安装和管理信息。受支持的机器列表。

开发者指南

面向 Postgres 开发者的信息。此书面向为 Postgres 项目做贡献的人；应用开发的信息见程序员指南。目前包含在程序员指南中。

参考手册

关于命令语法的详细参考信息。目前包含在用户指南中。

除本手册集之外，还有其他资源可以帮助你安装和使用 Postgres：

man 手册页

man 手册页包含关于命令语法的一般信息。

FAQ

常见问题（FAQ）文档既阐述一般性问题，也阐述一些平台特定的问题。

README

某些附带的软件包提供了 README 文件。

网站

Postgres[8] 网站上可能有发行包中未包含的信息。那里有一个 mhonarc 邮件列表流量存档，是许多主题的丰富资源。

邮件列表

pgsql-general[9]（archive[10]）邮件列表是一个让用户问题得到解答的好地方。
还有其他可用的邮件列表；详情请参阅 PostgreSQL 网站的 Info Central 栏目。

你自己！

Postgres 是一项开源产品。因此，它的持续支持依赖于用户社区。当你开始使用 Postgres 时，你会依赖他人的帮助——无论是通过文档还是通过邮件列表。请考虑回馈你的知识。如果你学到了文档中没有的东西，把它写出来并贡献回去。如果你给代码添加了特性，也把它们贡献出去。

即使经验不多的人也能为文档提供更正和小改动，而这是一个很好的起点。pgsql-docs[11]（archive[12]）邮件列表就是开始的地方。

1.5. 术语

在后面的文档中，站点（site）可以解释为安装 Postgres 的主机。由于在单台主机上可以安装多套 Postgres 数据库，这个术语更确切地指任何一套特定的已安装的 Postgres 二进制文件和数据库。

Postgres 超级用户是名为 *postgres* 的用户，他拥有 Postgres 的二进制文件和数据库文件。作为数据库超级用户，所有的保护机制都可以被绕过，任何数据都可以任意访问。此外，Postgres 超级用户还被允许执行一些通常不向所有用户开放的支持程序。注意，Postgres 超级用户不是 Unix 超级用户（后者将称为 root）。出于安全原因，超级用户应具有非零的用户标识符（UID）。

数据库管理员（DBA）负责安装 Postgres 并为站点建立实施安全策略的机制。DBA 可以按下述方法添加新用户，并维护一组供 createdb 使用的模板数据库。

postmaster 是为发往 Postgres 系统的请求充当中转站的进程。前端应用连接到 postmaster，由它跟踪系统错误和后端进程之间的通信。postmaster 可以接受若干命令行参数来调整其行为。不过，只有当你打算运行多个站点或非默认站点时才需要提供参数。

[8] http://www.postgresql.org
[9] mailto:pgsql-general@postgresql.org
[10] http://www.postgresql.org/mhonarc/pgsql-general/
[11] mailto:pgsql-docs@postgresql.org
[12] http://www.PostgreSQL.ORG/mhonarc/pgsql-docs/

Postgres 后端（实际的可执行程序 postgres）可以由 Postgres 超级用户直接在用户 shell 中执行（以数据库名作为参数）。但这样做会绕过与 postmaster/站点关联的共享缓冲池和锁表，因此在多用户站点中不建议这样做。

1.6. 记法

文件名前面的"..."或 /usr/local/pgsql/ 用来表示 Postgres 超级用户主目录的路径。

在命令概要中，方括号（"[和"]"）表示可选的短语或关键字。花括号（"{和"}"）中含有竖线（"|"）时表示你必须选择其中一个。

在示例中，圆括号（"(和")"）用于给布尔表达式分组。"|"是布尔运算符 OR。

示例将展示从不同账户和程序执行的命令。从 root 账户执行的命令前带有 ">"。从 Postgres 超级用户账户执行的命令前带有"%"，而从无特权用户账户执行的命令前带有 "$"。SQL 命令前带有"=>" 提示符，也可能没有前导提示符，视上下文而定。

> **注意**
>
> 在撰写本文时（Postgres v7.0），整个文档集中标记命令的记法并不完全一致。如发现问题请报告给文档邮件列表[13]。

1.7. 问题报告指南

当你在PostgreSQL中遇到问题时，我们希望得知此事。你的缺陷报告对于提高 PostgreSQL的可靠性十分重要，因为即使再怎么小心，也无法保证 PostgreSQL 的每一部分都能在每个平台、任何情况下正常工作。

以下建议旨在帮助你组织缺陷报告，以便能够高效处理。没有人必须遵循这些建议，但这样做通常对各方都有好处。

我们无法承诺立刻修复每一个缺陷。如果该缺陷明显、严重，或者影响大量用户，很可能会有人去调查它。也可能我们会让你升级到更新的版本，看看在新版本中是否也会出现同样的缺陷。或者，我们可能会认定，在某些计划中的重大重写完成之前，这个缺陷无法修复。再或者，它只是过于棘手，而当前还有更重要的事情要处理。如果你需要立即获得帮助，请考虑购买商业支持合同。

1.7.1. 识别缺陷

在问"这是不是缺陷？"之前，请反复阅读文档，以确认你确实可以完成正在尝试做的事情。如果从文档中无法明确看出某件事是否可行，也请报告；那就是文档中的缺陷。如果程序实际做的事情与文档描述不同，那就是缺陷。这可能包括但不限于以下情况：

- 程序因致命信号而终止，或者给出一条表明程序本身存在问题的操作系统错误消息（反例可能是 "磁盘满" 之类的消息，因为那必须在 Postgres 之外解决）。
- 程序对给定输入产生了错误的输出。

[13] mailto:docs@postgresql.org

- 程序拒绝接受有效输入。
- 程序接受了无效输入，却没有给出提示或错误消息。
- 在受支持的平台上，PostgreSQL无法按照说明完成编译、构建或安装。

这里的"程序"指的是任何可执行程序，不只是后端服务器。

运行缓慢或消耗大量资源，并不一定就是缺陷。请阅读文档，
或在某个邮件列表中寻求调优应用程序的帮助。不符合 SQL标准，也不一定是缺陷，除非明确声称该特定特性符合标准。

在继续之前，请查看 TODO 列表和 FAQ，确认你的缺陷是否已经是已知问题。如果你看不懂 TODO 列表中的信息，也请报告你的问题。至少我们可以把 TODO 列表写得更清楚。

1.7.2. 报告哪些内容

关于缺陷报告，最重要的一点是陈述全部事实，而且只陈述事实。
不要猜测你觉得哪里出了问题、"它看起来像是在做什么"，或者程序的哪一部分有错。
如果你不熟悉实现，十有八九会猜错，对我们毫无帮助。即使你熟悉实现，
有根据的解释也只能是事实的有益补充，而不能替代事实。如果我们要修复这个缺陷，
仍然必须先亲自重现它。陈述这些基本事实并不难做（你大概可以直接从屏幕上复制粘贴），
但太多时候，重要细节会被遗漏，因为有人觉得它们不重要，或者认为即使不说清楚，
报告也一样能被理解。

以下各项应包含在每一个缺陷报告中：

- 重现问题所需的精确步骤序列，而且必须是从程序启动开始。这些步骤应当是自包含的；
  如果输出依赖表中的数据，那么仅仅发来一条 select 语句，而不附带前面的 create table 和 insert 语句，是不够的。我们没有时间去逆向推导你的数据库模式；如果要靠我们自己编造数据，多半会错过问题。对于 SQL 相关问题，测试用例的最佳形式是一个可由psql前端运行、并能展示问题的文件。（请务必确保你的`~/.psqlrc`启动文件中没有任何内容。）
  我们鼓励你尽量缩小示例规模，但这并非绝对必要。如果缺陷可重现，
  我们无论如何都能找出来。

  如果你的应用使用的是别的客户端接口，例如 PHP，那么请尽量隔离出触发问题的查询。我们大概不会为了重现你的问题去搭建一个 Web 服务器。无论如何，都请记得提供精确的输入文件；不要笼统地猜测问题发生在 "大文件"或"中等大小的数据库"等情况下，因为这些信息不够精确，派不上用场。

- 你实际得到的输出。请不要只说它"没起作用"或"失败了"。如果有错误消息，请把它贴出来，
  即使你看不懂。如果程序因操作系统错误而终止，请说明是哪一种错误。如果什么都没发生，
  也请说清楚。即使你的测试用例导致程序崩溃，或出现其他显而易见的问题，
  这种情况在我们的平台上也未必会发生。如果可以，最简单的做法就是直接从终端复制输出。

  > **注意**
  >
  > 在致命错误的情况下，客户端报告的错误消息可能不包含全部可用信息。
  > 也请查看数据库服务器的日志输出。如果你平时没有保留服务器日志，
  > 现在正是开始这样做的好时机。

- 你期望得到的输出，也非常重要。如果你只是写 "这个命令给了我那个输出。"或"这不是我期望的。"，我们可能会自己运行一遍，扫一眼输出，然后觉得它看起来没问题，

也正是我们所期望的。我们不应该花时间去揣摩你的命令背后的确切语义。尤其不要只是说 "这不是 SQL 规定的/Oracle 所做的。"从SQL标准中找出正确行为并不是一件轻松的事，而且我们也不都了解其他所有关系数据库的行为。（如果你的问题是程序崩溃，
显然可以省略这一项。）

- 任何命令行选项和其他启动选项，包括所有被你改动过、
  且与问题相关的环境变量或配置文件。同样，请提供精确的信息。
  如果你使用的是一种预打包发行版，并且它会在系统启动时自动启动数据库服务器，
  那么你应当尽量弄清楚它是怎么做到的。

- 任何偏离安装说明的做法。

- PostgreSQL的版本。你可以运行命令 `SELECT version();` 来查明你当前运行的版本。如果这个函数不存在，请说明这一点，这样我们就知道你的版本已经够老了。
  如果你无法启动服务器或客户端，请查看源代码目录中的 README 文件，或者查看你的发行文件或软件包的名称。如果你的版本早于 7.0，我们几乎肯定会建议你升级。每个新版本都包含大量的缺陷修复，这正是我们编写新版本的原因。

- 平台信息。这包括内核名称和版本、C　库、处理器、内存信息等。在大多数情况下，报告供应商和版本就足够了，但不要想当然地认为所有人都知道 "Debian"具体包含什么，或者所有人都运行在 i386 上。如果你遇到的是安装问题，那么你机器上的工具链信息（编译器、make 等）也是必需的。

不要担心你的缺陷报告会变得很长；这很正常。第一次就把所有内容都报告出来，
总比让我们反复追问细节要好。另一方面，如果你的输入文件非常大，
先问问是否有人愿意看一看，也是合理的。

不要把全部时间都花在找出输入中的哪些变化会让问题消失上。这多半无助于解决问题。
如果最后发现这个缺陷无法立刻修复，你仍然有时间去寻找并分享变通办法。还有，
再强调一次，不要浪费时间去猜测这个缺陷为什么存在。我们迟早会查明原因。

在编写缺陷报告时，请选择不会引起混淆的术语。整个软件包本身叫作 "PostgreSQL"，有时简称 "Postgres"。（有时也会用缩写 "Pgsql"，但请不要那样做。）如果你特指后端服务器，请明确说出来，不要只是说 "Postgres 崩溃了"。交互式前端叫作 "psql"，它实际上与后端完全是分开的。

1.7.3. 到哪里报告缺陷

一般来说，请将缺陷报告发送到缺陷报告邮件列表[14]。请你为电子邮件找一个描述性的主题，
必要时可以摘取部分错误消息。

不要将缺陷报告发送到任何用户邮件列表，例如 SQL 语言邮件列表[15]或一般主题邮件列表[16]。这些邮件列表用于回答用户问题，其订阅者通常并不希望收到缺陷报告。更重要的是，
他们也不太可能去修复这些缺陷。

此外，请不要把报告发送到开发者邮件列表[17]。这个列表是用来讨论 PostgreSQL开发的，最好把缺陷报告和这类讨论分开。如果这个缺陷需要更多的审查，
我们可能会选择在它上面继续讨论你的缺陷报告。

如果你遇到的是文档问题，请发送邮件到文档邮件列表[18]。请在问题报告中提及文档、
章节和小节。

[14] mailto:pgsql-bugs@postgresql.org
[15] mailto:pgsql-sql@postgresql.org
[16] mailto:pgsql-general@postgresql.org
[17] mailto:pgsql-hackers@postgresql.org
[18] mailto:pgsql-docs@postgresql.org

如果你的缺陷是在某个不受支持平台上出现的可移植性问题，请发送邮件到
移植问题邮件列表[19]，这样我们（以及你）就可以共同推进 PostgreSQL在该平台上的移植工作。

注意

由于垃圾邮件泛滥，上述所有电子邮件地址都是封闭邮件列表。也就是说，
你必须订阅它们才能发帖。如果你只是希望发邮件但不想接收列表邮件，可以订阅特殊的 pgsql-loophole 邮件列表，它允许你向所有 PostgreSQL 邮件列表发帖而不接收任何消息。要订阅请发送电子邮件到 pgsql-loophole-request@postgresql.org[20]。

1.8. Y2K 声明

作者

由 Thomas Lockhart[21] 写于 1998-10-22。更新于 2000-03-31。

PostgreSQL 全球开发队伍将 Postgres软件代码树作为公共服务提供，不对它的行为或性能作任何保证，也不承担任何责任。不过，在撰写本文时：

- 本声明的作者自 1996 年 11 月起就是 Postgres 支持团队的一名志愿者，他不知道 Postgres代码库中存在任何与 2000 年 1 月 1 日前后的时间过渡（Y2K）相关的问题。
- 本声明的作者没有听说在回归测试中或在 Postgres近期或当前版本的其他实际使用中发现过任何 Y2K 问题的报告。考虑到已安装的用户基础以及用户在支持邮件列表上的积极参与，如果存在问题，我们本应有所耳闻。
- 就作者所知，Postgres 对用两位数字年份指定的日期所做的假定记录在当前用户指南[22]关于数据类型的章节中。对于两位年份，关键的过渡年份是 1970 而不是 2000；例如"`70-01-01`"被解释为 1970-01-01，而"`69-01-01`"被解释为 2069-01-01。
- 底层操作系统中与获取"当前时间"相关的任何 Y2K 问题都可能传播为 Postgres中表面的 Y2K 问题。

关于 Y2K 问题的进一步讨论，特别是与开源、免费软件相关的讨论，请参阅 The Gnu Project[23] 和 The Perl Institute[24]。

1.9. 版权与商标

[19] mailto:pgsql-ports@postgresql.org
[20] mailto:pgsql-loophole-request@postgresql.org
[21] mailto:lockhart@alumni.caltech.edu
[22] http://www.postgresql.org/docs/user/datatype.htm
[23] http://www.gnu.org/software/year2000.html
[24] http://language.perl.com/news/y2k.html

在任何情况下，加利福尼亚大学都不对任何一方承担因使用本软件及其文档而产生的任何直接、间接、特殊、附带或后果性损害赔偿责任，包括利润损失，
即使加利福尼亚大学已被告知此类损害发生的可能性。

加利福尼亚大学明确不作任何担保，包括但不限于适销性和特定用途适用性的默示担保。
本协议项下提供的软件按"原样"提供，加利福尼亚大学没有义务提供维护、支持、更新、
增强或修改。

所有商标均为其各自所有者的财产。

第 2 章 SQL 语法

本章给出 SQL 的一般语法描述。

SQL 操作数据的集合。该语言由各种关键词组成。其中允许算术和过程表达式。我们将在本章中讨论这些主题；后续章节将包括数据类型、函数和操作符的细节。

2.1. 关键词

SQL92 为语言定义了具有特定含义的关键词。某些关键词是保留的，这表示它们被限制只能出现在特定的上下文中。另一些关键词不受限制，这表示它们在某些上下文中具有特定含义，但在其他方面不受约束。

Postgres 实现了 SQL92 和 SQL3 语言的一个扩展子集。由于 Postgres 的可扩展性特性，本实现中的某些语言元素没有语言标准所要求的那样受限。

关于 SQL92 和 SQL3 关键词的信息摘译自 Date and Darwen, 1997。

2.1.1. 保留关键词

SQL92 和 SQL3 有一些保留关键词，它们不允许用作标识符，并且除了作为 SQL 语句中的基本词元之外不允许任何其他用法。Postgres 还有一些具有类似限制的附加关键词。特别是，这些关键词不允许用作列名或表名，尽管在某些情况下它们允许用作列标签（即在 AS 子句中）。

> **提示**
>
> 任意字符串只要用双引号括起来（“像这样！”），就可以被指定为标识符。
> 需要小心的是，这样的标识符将区分大小写，
> 并且会保留嵌入的空白和大多数其他特殊字符。

下列是 Postgres 的保留词，它们既不是 SQL92 也不是 SQL3 的保留词。它们允许作为列标签出现，但不能用作标识符：

```
ABORT ANALYZE
BINARY
CLUSTER CONSTRAINT COPY
DO
EXPLAIN EXTEND
LISTEN LOAD LOCK
MOVE
NEW NONE NOTIFY
OFFSET
RESET
SETOF SHOW
UNLISTEN UNTIL
VACUUM VERBOSE
```

下列是 Postgres 的保留词，它们同时也是 SQL92 或 SQL3 的保留词，并且允许作为列标签出现，但不能用作标识符：

```
ALL ANY ASC BETWEEN BIT BOTH
CASE CAST CHAR CHARACTER CHECK COALESCE COLLATE COLUMN
 CONSTRAINT CROSS CURRENT CURRENT_DATE CURRENT_TIME
 CURRENT_TIMESTAMP CURRENT_USER
DEC DECIMAL DEFAULT DESC DISTINCT
ELSE END EXCEPT EXISTS EXTRACT
FALSE FLOAT FOR FOREIGN FROM FULL
GLOBAL GROUP
HAVING
IN INNER INTERSECT INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NATURAL NCHAR NOT NULL NULLIF NUMERIC
ON OR ORDER OUTER OVERLAPS
POSITION PRECISION PRIMARY PUBLIC
REFERENCES RIGHT
SELECT SESSION_USER SOME SUBSTRING
TABLE THEN TO TRANSACTION TRIM TRUE
UNION UNIQUE USER
VARCHAR
WHEN WHERE
```

下列是 Postgres 的保留词，它们同时也是 SQL92 或 SQL3 的保留词：

```
ADD ALTER AND AS
BEGIN BY
CASCADE CLOSE COMMIT CREATE CURSOR
DECLARE DEFAULT DELETE DESC DISTINCT DROP
EXECUTE EXISTS EXTRACT
FETCH FLOAT FOR FROM FULL
GRANT
HAVING
IN INNER INSERT INTERVAL INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NAMES NATIONAL NATURAL NCHAR NO NOT NULL
ON OR OUTER
PARTIAL PRIMARY PRIVILEGES PROCEDURE PUBLIC
REFERENCES REVOKE RIGHT ROLLBACK
SELECT SET SUBSTRING
TO TRAILING TRIM
UNION UNIQUE UPDATE USING
VALUES VARCHAR VARYING VIEW
WHERE WITH WORK
```

下列是 SQL92 的保留关键词，它们不是 Postgres 的保留关键词，但如果用作函数名，总是会被翻译成函数 `CHAR_LENGTH`：

```
CHARACTER_LENGTH
```

下列是 SQL92 或 SQL3 的保留关键词，它们不是 Postgres 的保留关键词，但如果用作类型名，总是会被翻译成一个替代的原生类型：

```
BOOLEAN DOUBLE FLOAT INT INTEGER INTERVAL REAL SMALLINT
```

下列不是任何种类的关键词，但当用在类型名的上下文中时会被翻译成一个原生的 Postgres 类型，而当用在函数名的上下文中时会被翻译成一个原生函数：

```
DATETIME TIMESPAN
```

（分别翻译为 `TIMESTAMP` 和 `INTERVAL`。）这一特性旨在帮助向 v7.0 过渡，并将在下一个完整版本（可能是 v7.1）中被移除。

下列是 SQL92 或 SQL3 的保留关键词，它们不是 Postgres 中的关键词。在撰写本文时（v7.0），它们在 Postgres 中没有被禁止的用法，但将来可能成为保留关键词：

> **注意**
>
> 其中某些关键词在 SQL92 中表示函数。这些函数在 Postgres 中有定义，但解析器不把这些名称当作关键词，它们被允许用在其他上下文中。

```
ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG
BIT_LENGTH
CASCADED CATALOG CHAR_LENGTH CHARACTER_LENGTH COLLATION
 CONNECT CONNECTION CONTINUE CONVERT CORRESPONDING COUNT
 CURRENT_SESSION
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR
 DIAGNOSTICS DISCONNECT DOMAIN
ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL
FIRST FOUND
GET GO GOTO
IDENTITY INDICATOR INPUT INTERSECT
LAST LOWER
MAX MIN MODULE
OCTET_LENGTH OPEN OUTPUT OVERLAPS
PREPARE PRESERVE
ROWS
SCHEMA SECTION SESSION SIZE SOME
 SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM_USER
TEMPORARY TRANSLATE TRANSLATION
UNKNOWN UPPER USAGE
VALUE
WHENEVER WRITE
```

2.1.2. 非保留关键词

SQL92 和 SQL3 有一些非保留关键词，它们在语言中有规定的含义，但也允许用作标识符。Postgres 还有一些允许类似不受限用法的附加关键词。特别是，这些关键词允许用作列名或表名。

下列是 Postgres 的非保留关键词，它们既不是 SQL92 也不是 SQL3 的非保留关键词：

```
ACCESS AFTER AGGREGATE
```

```
BACKWARD BEFORE
CACHE COMMENT CREATEDB CREATEUSER CYCLE
DATABASE DELIMITERS
EACH ENCODING EXCLUSIVE
FORCE FORWARD FUNCTION
HANDLER
INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL
LANCOMPILER LOCATION
MAXVALUE MINVALUE MODE
NOCREATEDB NOCREATEUSER NOTHING NOTIFY NOTNULL
OIDS OPERATOR
PASSWORD PROCEDURAL
RECIPE REINDEX RENAME RETURNS ROW RULE
SEQUENCE SERIAL SHARE START STATEMENT STDIN STDOUT
TEMP TRUSTED
UNLISTEN UNTIL
VALID VERSION
```

下列是 Postgres 的非保留关键词，它们是 SQL92 或 SQL3 的保留关键词：

```
ABSOLUTE ACTION
CONSTRAINTS
DAY DEFERRABLE DEFERRED
HOUR
IMMEDIATE INITIALLY INSENSITIVE ISOLATION
KEY
LANGUAGE LEVEL
MATCH MINUTE MONTH
NEXT
OF ONLY OPTION
PENDANT PRIOR PRIVILEGES
READ RELATIVE RESTRICT
SCROLL SECOND
TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TRIGGER
YEAR
ZONE
```

下列是 Postgres 的非保留关键词，它们同时也是 SQL92 或 SQL3 的非保留关键词：

```
COMMITTED SERIALIZABLE TYPE
```

下列是 SQL92 或 SQL3 的非保留关键词，它们不是 Postgres 中任何种类的关键词：

```
ADA
C CATALOG_NAME CHARACTER_SET_CATALOG CHARACTER_SET_NAME
 CHARACTER_SET_SCHEMA CLASS_ORIGIN COBOL COLLATION_CATALOG
 COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME
 COMMAND_FUNCTION CONDITION_NUMBER
 CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME
 CONSTRAINT_SCHEMA CURSOR_NAME
DATA DATE_TIME_INTERVAL_CODE DATE_TIME_INTERVAL_PRECISION
 DYNAMIC_FUNCTION
```

```
FORTRAN
LENGTH
MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MORE MUMPS
NAME NULLABLE NUMBER
PAD PASCAL PLI
REPEATABLE RETURNED_LENGTH RETURNED_OCTET_LENGTH
 RETURNED_SQLSTATE ROW_COUNT
SCALE SCHEMA_NAME SERVER_NAME SPACE SUBCLASS_ORIGIN
TABLE_NAME
UNCOMMITTED UNNAMED
```

2.2. 注释

注释是以双连字符开始并延伸到行尾的任意字符序列，例如：

```
-- 这是一条标准 SQL 注释
```

我们也支持 C 风格的块注释，例如：

```
/* 多行
   注释
 */
```

以 "/*" 开始的注释延伸到第一次出现 "*/" 之处。

2.3. 名称

SQL 中的名称必须以字母（a-z）或下划线（_）开头。名称中的后续字符可以是字母、数字（0-9）或下划线。系统对名称最多只使用 NAMEDATALEN-1 个字符；在查询中可以写更长的名称，但它们会被截断。默认情况下，NAMEDATALEN 为 32，因此名称的最大长度是 31（但在编译系统时，可以修改 src/include/postgres_ext.h 中的 NAMEDATALEN）。

包含其他字符的名称可以通过用双引号（"）把它们括起来构成。例如，表名或列名如果加引号，就可以包含原本不允许的字符，如空格、和号等。给名称加引号还会使其区分大小写，而不加引号的名称总是被折叠为小写。例如，名称 FOO、foo 和 "foo" 在 Postgres 中被视为相同，但 "Foo" 是一个不同的名称。

双引号还可以用来保护一个否则会被当作 SQL 关键词的名称。例如，IN 是关键词，而 "IN" 是名称。

2.4. 常量

Postgres 中有三种隐式类型常量：字符串、整数和浮点数。常量也可以用显式类型指定，这可以使后端更准确地表示并更高效地处理它们。隐式常量将在下面介绍；显式常型随后讨论。

2.4.1. 字符串常量

SQL 中的字符串是由单引号括起的任意 ASCII 字符序列（""，例如 'This is a string'）。SQL92 允许在字符串中嵌入单引号，方法是写两个相邻的单引号（例如 'Dianne''s horse'）。在 Postgres 中，也可以用反斜线（"\"，例如 'Dianne

\'s horse'）对单引号转义。要在字符串常量中包含一个反斜线，请写两个反斜线。不可打印字符也可以通过在前面加一个反斜线（例如 '*tab*'）嵌入到字符串中。

2.4.2. 整数常量

SQL 中的整数常量是不带小数点的 ASCII 数字序列。合法值的范围是从 -2147483648 到 +2147483647。这个范围会随操作系统和主机机器而变化。

注意，更大的整数可以通过 SQL92 字符串记法或 Postgres 类型记法为 int8 指定：

```
int8 '4000000000'  -- string style
'4000000000'::int8 -- Postgres (historical) style
```

2.4.3. 浮点常量

浮点常量由一个整数部分、一个小数点和一个小数部分组成，或者是下列格式的科学记法：

```
{dig}.{dig} [e [+-] {dig}]
```

其中 *dig* 是一个或多个数字。如果使用那些选项，则必须在句点之后以及 [+-] 之后至少包含一个 *dig*。缺少尾数的指数会被插入尾数 1。字符串中不能嵌入多余的字符。

浮点常量的类型是 float8。float4 可以用 SQL92 字符串记法或 Postgres 类型记法显式指定：

```
float4 '1.23'  -- string style
'1.23'::float4 -- Postgres (historical) style
```

2.4.4. Postgres 用户定义类型的常量

一个任意类型的常量可以用下列记法之一输入：

```
type 'string'
'string'::type
CAST 'string' AS type
```

字符串之内的值被传递给名为 *type* 的类型的输入转换例程。结果就是所指示类型的一个常量。如果常量必须是哪个类型没有歧义，显式类型转换可以省略，此时它会被自动强制转换。

2.4.5. 数组常量

数组常量是任何 Postgres 类型的数组，包括其他数组、字符串常量等。数组常量的一般格式如下：

```
{val1delimval2delim}
```

其中 *delim* 是存储在 pg_type 类中该类型的分隔符。（对于内建类型，这是逗号字符（",")。）一个数组常量的示例是

```
{{1,2,3},{4,5,6},{7,8,9}}
```

这个常量是一个二维的 3×3 数组，由三个整数子数组组成。

单个数组元素只要有可能就应当放在引号之间，以避免与前导空白相关的歧义问题。

2.5. 字段和列

2.5.1. 字段

一个字段既可以是给定类的一个属性，也可以是下列之一：

oid

表示一个实例的唯一标识符，它由 Postgres 自动添加到所有实例上。Oid 不会被重用，并且是 32 位的量。

xmin

插入事务的标识。

xmax

删除事务的标识。

cmin

事务内部的命令标识符。

cmax

删除命令的标识。

关于这些字段的更多信息请参阅 Stonebraker, Hanson, Hong, 1987。时间在内部表示为 `abstime` 数据类型的实例。事务和命令标识符都是 32 位的量。事务从 512 开始顺序分配。

2.5.2. 列

一个列是下列形式的构造：

```
instance{.composite_field}.field `['number`]'
```

instance 标识一个特定的类，并且可以看作代表该类的实例。实例变量要么是一个类名，要么是一个由 FROM 子句定义的类的代理，要么是关键词 NEW 或 CURRENT。NEW 和 CURRENT 只能出现在规则的动作部分，而其他实例变量可以用在任何 SQL 语句中。*composite_field* 是某个 Postgres 复合类型的一个字段，而后续的复合字段寻址该复合字段所求值到的（那些）类中的属性。最后，*field* 是最后被寻址的（那些）类中的一个普通（基本类型）字段。如果 *field* 的类型是 `array`，那么可选的 *number* 指示符指定该数组中的一个特定元素。如果没有指示编号，则返回所有数组元素。

2.6. 操作符

SQL 中可以使用任何内建系统操作符或用户定义操作符。关于内建和系统操作符的列表请参阅操作符。关于用户定义操作符的列表，请咨询你的系统管理员或在 `pg_operator` 类上运行一个查询。圆括号可用于在表达式中对操作符进行任意分组。

2.7. 表达式

SQL92 允许用表达式变换表中的数据。表达式可以包含操作符（详见操作符）和函数（函数有更多信息）。

一个表达式是下列之一：

(a_expr)
常量
属性
a_exprbinary_operatora_expr
a_exprright_unary_operator
left_unary_operatora_expr
参数
函数表达式
聚合表达式

我们已经讨论过常量和属性。三种操作符表达式分别表示二元（中缀）、右一元（后缀）和左一元（前缀）操作符。下面的小节讨论剩下的选项。

2.7.1. 参数

参数用来表示 SQL 函数中的一个参数。这典型地用在 SQL 函数定义语句中。参数的形式是：

$number

例如，考虑一个函数 `dept` 的定义：

```
CREATE FUNCTION dept (name)
 RETURNS dept
 AS 'select * from
     dept where name=$1'
 LANGUAGE 'sql';
```

2.7.2. 函数表达式

函数表达式是一个合法 SQL 函数的名称，后跟放在圆括号内的参数列表：

function (*a_expr* [, *a_expr* ...])

例如，下面计算一个雇员薪水的平方根：

```
sqrt(emp.salary)
```

2.7.3. 聚合表达式

一个聚合表达式表示将聚合函数应用于查询选中的各行。
聚合函数将多个输入归约为单个输出值，例如输入的和或平均值。
聚合表达式的语法可以是以下形式之一：

aggregate_name (*expression*)
aggregate_name (ALL *expression*)
aggregate_name (DISTINCT *expression*)
aggregate_name (*)

其中 *aggregate_name* 是先前定义的聚合，而 *expression* 是任何本身不包含聚合表达式的表达式。

第一种形式的聚合表达式为所有使给定表达式产生非空值的输入行调用聚合。
第二种形式和第一种相同，因为 ALL 是默认选项。第三种形式为输入行中表达式的所有非空可区分值调用聚合。最后一种形式为每一个输入行调用一次聚合而不管值是空还是非空；
因为没有指定特定的输入值，它通常只对 count() 聚合有用。

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

2.7.4. 目标列表

一个目标列表是放在圆括号内、以逗号分隔的一个或多个元素的列表，
每个元素必须是下列形式：

```
a_expr [ AS result_attname ]
```

其中 *result_attname* 是要创建的属性的名称（在更新语句的情况下则是已存在的属性名）。如果 *result_attname* 不存在，那么 *a_expr* 必须只包含一个属性名，该属性名被假定是结果字段的名称。在 Postgres 中，只有当 *a_expr* 是一个属性时才使用默认命名。

2.7.5. 限定条件

一个限定条件由任意数量的子句构成，子句之间用逻辑操作符连接：

NOT
AND
OR

子句是一个在一组实例上求值为 boolean 的 *a_expr*。

2.7.6. From 列表

from 列表是以逗号分隔的from 表达式列表。每个"from 表达式"的形式是：

```
[ class_reference ] instance_variable
 {, [ class_ref ] instance_variable... }
```

其中 *class_reference* 的形式是

```
class_name [ * ]
```

"from 表达式"定义一个或多个实例变量，它们取值于 *class_reference* 所指示的类。通过在后面追加指示符星号（"*"），还可以让实例变量取值于继承层次中位于所指示类之下的所有类。

第 3 章 数据类型

描述 Postgres 中可用的内建数据类型。

Postgres 有一套丰富的本机数据类型可供用户使用。用户可以用 `CREATE TYPE` 命令向 Postgres 添加新类型。

在数据类型的语境下，下面几节将讨论 SQL 标准兼容性、移植问题和使用方法。某些 Postgres 类型直接对应于 SQL92 兼容类型。另一些情况下，由 SQL92 语法定义的数据类型被直接映射为 Postgres 本机类型。许多内建类型都有显而易见的外部格式。但也有一些类型要么为 Postgres 所独有（例如开放和闭合路径），要么有多种可选格式（例如日期和时间类型）。

表 3.1. Postgres 数据类型

Postgres 类型	SQL92 或 SQL3 类型	描述
bool	boolean	逻辑布尔值（真/假）
box		二维平面上的矩形框
char(n)	character(n)	定长字符串
cidr		IPv4 网络或主机地址
circle		二维平面上的圆
date	date	不含时间的日历日期
decimal	decimal(p,s)	p <= 9、s = 0 的精确数值
float4	float(p), p < 7	精度为 p 的浮点数
float8	float(p), 7 <= p < 16	精度为 p 的浮点数
inet		IPv4 网络或主机地址
int2	smallint	有符号2字节整数
int4	int, integer	有符号 4 字节整数
int8		有符号 8 字节整数
interval	interval	通用时间段
line		二维平面上的无限直线
lseg		二维平面上的线段
money	decimal(9,2)	美式货币
numeric	numeric(p,s)	p == 9、s = 0 的精确数值
path		二维平面上的开放和闭合几何路径
point		二维平面上的几何点
polygon		二维平面上的封闭几何路径
serial		用于索引和交叉引用的唯一 id
time	time	一天中的时间
timetz	time with time zone	一天中的时间，包括时区
timestamp	timestamp with time zone	日期/时间
varchar(n)	character varying(n)	变长字符串

> **注意**
>
> `cidr` 和 `inet` 类型设计为可处理任何 IP 类型，但当前实现只处理 ipv4。这里所有关于 ipv4 的内容在未来的版本中都将适用于 ipv6。

表 3.2. Postgres 函数常量

Postgres 函数	SQL92 常量	描述
getpgusername()	current_user	当前会话中的用户名
date('now')	current_date	当前事务的日期
time('now')	current_time	当前事务的时间
timestamp('now')	current_timestamp	当前事务的日期和时间

Postgres 的特性处于 ORDBMS 开发的前沿。除了符合 SQL3 之外，还支持 SQL92 的很大一部分。尽管我们力求 SQL92 兼容，但该标准有一些方面考虑欠周，不应延续到后续标准中。Postgres 不会花大力气去符合这些特性；不过它们往往只适用于很少使用或冷僻的场景，典型用户不太可能遇到。

大多数对应于基本类型（如整数和浮点数）的输入和输出函数会做一些错误检查。
为了提高执行速度，某些运算符和函数（例如加法和乘法）不做运行时错误检查。
例如在某些系统上，某些数据类型的数值运算符可能静默下溢或上溢。

有些输入和输出函数是不可逆的。也就是说，输出函数的结果与原始输入相比可能损失精度。

> **注意**
>
> 浮点数允许保留该类型的大部分固有精度（双精度通常 15 位，4 字节浮点通常 6 位）。底层为浮点字段的其他类型（如几何类型）具有类似的精度。

3.1. 数字类型

数值类型包括二字节和四字节整数，四字节和八字节浮点数，以及固定精度的小数。

表 3.3. Postgres 数值类型

数值类型	存储尺寸	描述	范围
decimal	可变	用户指定精度	约 8000 位
float4	4字节	可变精度	6 位十进制小数
float8	8字节	可变精度	15 位十进制小数
int2	2字节	定点数	-32768 到 +32767
int4	4字节	定点数的常用选择	-2147483648 到 +2147483647
int8	8字节	极大范围定点数	+/- > 18 位十进制小数
numeric	可变	用户指定精度	无限制
serial	4字节	标识符或交叉引用	0 到 +2147483647

数值类型有一整套相应的算术运算符和函数。更多信息请参阅数值操作符和数学函数。

int8 类型并非在所有平台上都可用，因为它依赖编译器对此的支持。

3.1.1. serial 类型

serial 类型是由 Postgres 用其他现有构件构造的一种特殊类型。它通常用于为表条目创建唯一标识符。在当前的实现中，指定

```
CREATE TABLE tablename (colname SERIAL);
```

等效于指定：

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
    (colname INT4 DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

> **小心**
>
> 为 serial 类型创建的隐式序列在表被删除时不会被自动删除。

支撑 serial 的隐式序列不会在包含 serial 类型的表被删除时自动删除。因此，按顺序执行下面的命令很可能会失败：

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

该序列会一直留在数据库中，直到用 DROP SEQUENCE 显式删除。

3.2. 货币类型

> **过时的类型**
>
> money 现已被弃用。请改用 numeric 或 decimal。在未来的版本中，money 类型可能会变成 numeric 类型之上一个感知区域设置的层。

money 类型支持以固定小数点表示的美式货币。如果 Postgres 编译时定义了 USE_LOCALE，则 money 类型应使用 locale(7) 定义的货币惯例。

表 3.4. Postgres 货币类型

货币类型	存储尺寸	描述	范围
money	4字节	定点数	-21474836.48 到 +21474836.47

numeric 将取代 money 类型，应优先使用它。

3.3. 字符类型

SQL92 定义两种主要字符类型：char 和 varchar。Postgres 支持这些类型，此外还有更通用的 text 类型；与 varchar 不同，它不要求为字段大小显式声明上限。

表 3.5. Postgres 字符类型

字符类型	存储尺寸	建议	描述
char	1字节	SQL92 兼容	单字符
char(n)	(4+n) 字节	SQL92 兼容	定长，空白填充
text	(4+x) 字节	最佳选择	变长
varchar(n)	(4+n) 字节	SQL92 兼容	有长度限制的变长

Postgres 中还有另一种定长字符类型。name 类型只有一个用途，即存储内部系统目录的名称。它并非供普通用户使用。其长度目前定义为 32 字节（31 个字符加终止符），但应使用 NAMEDATALEN 引用。该长度在编译时设定（因此可为特殊用途调整）；默认的最大长度在未来的版本中可能改变。

表 3.6. Postgres 特殊字符类型

字符类型	存储尺寸	描述
name	32字节	31 字符内部类型

3.4. 日期/时间类型

Postgres 支持全套 SQL 日期和时间类型。

表 3.7. Postgres 日期/时间类型

类型	描述	存储尺寸	最早	最晚	精度
timestamp	日期和时间	8字节	4713 BC	AD 1465001	1微秒 / 14位
timestamp with time zone	带时区的日期和时间	8字节	1903 AD	2037 AD	1微秒 / 14位
interval	用于时间间隔	12字节	-178000000年	178000000年	1微秒
date	仅日期	4字节	4713 BC	32767 AD	1日
time	仅一天中的时间	4字节	00:00:00.00	23:59:59.99	1微秒
time with time zone	仅一天中的时间	4字节	00:00:00.00+12	23:59:59.99-12	1微秒

注意

为确保与 Postgres 早期版本的兼容，我们还继续提供 datetime（等价于 timestamp）和 timespan（等价于 interval），但对它们的支持现在仅限于到 timestamp 和 interval 的隐式转换。abstime 和 reltime

类型是内部使用的低精度类型。不鼓励在新应用中使用这些类型，并建议在合适时把旧的迁移过去。这些内部类型中的任何一种或全部都可能在未来的版本中消失。

3.4.1. 日期/时间输入

日期和时间输入几乎可以采用任何合理的格式，包括 ISO-8601、SQL 兼容格式、传统 Postgres 格式等。日期输入中月和日的顺序可能有歧义，因此有一个设置用于指定歧义情况下如何解释。命令 `SET DateStyle TO 'US'` 或 `SET DateStyle TO 'NonEuropean'` 指定"月在前"的变体，命令 `SET DateStyle TO 'European'` 则设置"日在前"的变体。`ISO` 风格是默认值，但这一默认可以在编译时或运行时改变。

关于日期/时间输入的精确解析规则和可识别的时区，参见日期/时间支持。

请记住，任何日期或时间输入都必须像文本字符串一样用单引号括起来。

3.4.1.1. date

下面是 `date` 类型的一些可能输入。

表 3.8. Postgres 日期输入

示例	描述
January 8, 1999	无歧义
1999-01-08	ISO-8601 格式，推荐
1/8/1999	美式；在欧洲模式下读作 8 月 1 日
8/1/1999	欧式；在美式模式下读作 8 月 1 日
1/18/1999	美式；在任何模式下都读作 1 月 18 日
19990108	ISO-8601 年、月、日
990108	ISO-8601 年、月、日
1999.008	年和一年中的日子
99008	年和一年中的日子
January 8, 99 BC	公元前的 99 年

表 3.9. Postgres 月份缩写

月份	缩写
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

> **注意**
>
> 出于显而易见的原因，五月（`May`）没有明确的缩写。

表 3.10. Postgres 星期缩写

日	缩写
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

3.4.1.2. time

下面是有效的 `time` 输入。

表 3.11. Postgres 时间输入

示例	描述
04:05:06.789	ISO-8601
04:05:06	ISO-8601
04:05	ISO-8601
040506	ISO-8601
04:05 AM	和04:05一样；AM并不影响值
04:05 PM	和16:05一样；输入的小时必须 <= 12
z	和 00:00:00 一样
zulu	和 00:00:00 一样
allballs	和 00:00:00 一样

3.4.1.3. time with time zone

此类型由 SQL92 定义，但该定义存在根本性的缺陷，使这一类型几乎无法使用。在大多数情况下，`date`、`time` 和 `timestamp` 的组合应能提供任何应用所需的完整日期/时间功能。

`time with time zone` 接受所有对 `time` 类型合法的输入，再加上一个合法的时区，如下：

表 3.12. Postgres 带时区的时间输入

示例	描述
04:05:06.789-8	ISO-8601
04:05:06-08:00	ISO-8601
04:05-08:00	ISO-8601
040506-08	ISO-8601

更多时区的示例参见Postgres 时区输入。

3.4.1.4. timestamp

`timestamp` 类型的有效输入由一个日期和一个时间连接而成，其后可跟可选的 `AD` 或 `BC`，再后可跟可选的时区。（见下文。）因而

```
1999-01-08 04:05:06 -8:00
```

是一个符合 ISO 的有效 `timestamp` 值。此外，还支持广泛使用的格式

```
January 8 04:05:06 1999 PST
```

也被支持。

表 3.13. Postgres 时区输入

时区	描述
PST	太平洋标准时间
-8:00	PST的ISO-8601偏移
-800	PST的ISO-8601偏移
-8	PST的ISO-8601偏移

3.4.1.5. interval

`interval` 可用下列语法指定：

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
```

其中：`Quantity`（数量）是 ...、-1、0、1、2、...；`Unit`（单位）是 `second`、`minute`、`hour`、`day`、`week`、`month`、`year`、`decade`、`century`、`millennium`，
或者这些单位的缩写或复数形式；`Direction`（方向）可以是 `ago` 或为空。

3.4.1.6. 特殊值

下面的函数与 SQL 兼容，可用作相应数据类型的日期或时间输入：`CURRENT_DATE`、`CURRENT_TIME`、`CURRENT_TIMESTAMP`。

Postgres 还为方便使用支持几种特殊常量。

表 3.14. Postgres 特殊日期/时间常量

常量	描述
current	当前事务时间，延迟求值
epoch	1970-01-01 00:00:00+00（Unix系统时间0）
infinity	晚于其他有效时间
-infinity	早于其他有效时间
invalid	非法输入
now	当前事务时间

常量	描述
today	今天午夜
tomorrow	明天午夜
yesterday	昨天午夜

'now' 在插入值时解析，而 'current' 在每次取值时解析。所以在大多数应用中你可能想用 'now'。（当然，你真正想用的是 CURRENT_TIMESTAMP，它等价于 'now'。）

3.4.2. 日期/时间输出

输出格式可以用 SET DateStyle 设为四种风格之一：ISO-8601、SQL（Ingres）、传统 Postgres 和德式。默认是 ISO 格式。

表 3.15. Postgres 日期/时间输出风格

样式说明	描述	示例
'ISO'	ISO-8601 标准	1997-12-17 07:37:16-08
'SQL'	传统样式	12/17/1997 07:37:16.00 PST
'Postgres'	原始样式	Wed Dec 17 07:37:16 1997 PST
'German'	地区样式	17.12.1997 07:37:16.00 PST

date 和 time 风格的输出当然只是上述示例中相应的日期或时间部分。

SQL 风格有欧式和非欧式（美式）两种变体，它们决定月跟在日后还是日跟在月前。（另见上文“日期/时间输入”一节中关于此设置如何影响输入值解释的说明。）

表 3.16. Postgres 日期顺序惯例

样式说明	描述	示例
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

interval 的输出与输入格式相似，区别在于像 week 或 century 这样的单位会被转换为年和天。ISO 模式下输出形如

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

有几种方法可以影响日期/时间类型的外观：

- postmaster 启动时后端直接使用的 PGDATESTYLE 环境变量。
- 会话启动时前端 libpq 使用的 PGDATESTYLE 环境变量。
- SET DATESTYLE SQL 命令。

3.4.3. 时区

Postgres 力求在典型用法上与 SQL92 定义兼容。但 SQL92 标准对日期和时间类型及能力有一种奇怪的组合。两个明显的问题是：

- 尽管 date 类型没有关联的时区，time 类型却可以或确实有关联的时区。
- 默认时区被指定为相对 GMT/UTC 的固定整数偏移。

现实世界中的时区若不同时关联日期和时间就可能没有意义，
因为偏移量在一年中可能随夏令时边界而变化。

为解决这些困难，Postgres 只把时区关联到同时包含日期和时间的日期时间类型，而对只包含日期或只包含时间的类型假定使用本地时间。此外，时区支持源自底层操作系统的时区能力，
因此可以处理夏令时和其他预期行为。

对于 1902 到 2038 年之间的日期，Postgres 从底层操作系统获得时区支持（接近 Unix 风格系统的典型日期界限）。在此范围之外，所有日期都假定以协调世界时（UTC）指定和使用。

所有日期和时间在内部都以 UTC（也称为格林尼治标准时间 GMT）存储。时间在发送给客户端前端之前会被转换为数据库服务器上的本地时间，因此默认情况下采用服务器时区。

有几种方法可以影响时区行为：

- postmaster 启动时后端直接使用 TZ 环境变量作为默认时区。
- 在客户端设置的 PGTZ 环境变量由 libpq 用于在连接时向后端发送时区信息。
- SQL 命令 `SET TIME ZONE` 设置会话的时区。

如果指定了无效的时区，时区会变成 GMT（至少在大多数系统上如此）。

> **注意**
>
> 如果设置了编译选项 USE_AUSTRALIAN_RULES，那么 `EST` 指的是澳大利亚东部标准时间，其偏移为相对 UTC +10:00 小时。

3.4.4. 内部实现

Postgres 在所有日期/时间计算中使用儒略日。它们有一个好的特性：能正确预测/计算从公元前 4713 年之后的任何日期到遥远未来的任何日期，其假定是一年的长度为 365.2425 天。

19　世纪之前的日期约定读起来很有意思，但它们并不足够一致，不值得编码进日期/时间处理器。

3.5. 布尔类型

Postgres 支持 `bool` 作为 SQL3 布尔类型。`bool` 只能处于两种状态之一：'true'（真）或 'false'（假）。第三种状态 'unknown'（未知）未实现且 SQL3 也不建议使用；NULL 是一个有效的替代。`bool` 可用于任何布尔表达式，而布尔表达式的求值结果总是与此类型兼容。

`bool` 使用 1 字节存储。

表 3.17. Postgres 布尔类型

状态	输出	输入
真	't'	TRUE, 't', 'true', 'y', 'yes', '1'
假	'f'	FALSE, 'f', 'false', 'n', 'no', '0'

3.6. 几何类型

几何类型表示二维的空间物体。最基础的类型是点，它是所有其他类型的基础。

表 3.18. Postgres 几何类型

几何类型	存储尺寸	表示	描述
point	16字节	(x,y)	空间中的点
line	32字节	((x1,y1),(x2,y2))	无限直线
lseg	32字节	((x1,y1),(x2,y2))	有限线段
box	32字节	((x1,y1),(x2,y2))	矩形框
path	4+32n 字节	((x1,y1),...)	封闭路径（类似于多边形）
path	4+32n 字节	[(x1,y1),...]	开放路径
polygon	4+32n 字节	((x1,y1),...)	多边形（类似于封闭路径）
circle	24字节	<(x,y),r>	圆（圆心和半径）

有一套丰富的函数和运算符可用于执行各种几何操作，如缩放、平移、旋转以及求交。

3.6.1. 点

点是几何类型的基础二维构件。

`point` 用下列语法指定：

```
( x , y )
  x , y
```

其中的参数是

x

x 轴坐标，以浮点数表示。

y

y 轴坐标，以浮点数表示。

3.6.2. 线段

线段（`lseg`）由点对表示。

`lseg` 用下列语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中的参数是

(*x1*,*y1*)
(*x2*,*y2*)

线段的端点。

3.6.3. 方框

方框由作为其两个对角的点对表示。

box 用下列语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中的参数是

(*x1*,*y1*)
(*x2*,*y2*)

方框的对角。

方框使用第一种语法输出。输入时角点会被重新排序，先存左下角，后存右上角。
也可以输入方框的其他角点，但左下角和右上角由输入确定并存储。

3.6.4. 路径

路径由相连的点集表示。路径可以是"open"（开放）
的（集合中第一个点和最后一个点不相连），也可以是"closed"（封闭）
的（第一个点和最后一个点相连）。函数 popen(p) 和 pclose(p) 可用于强制路径开放或封闭，而函数 isopen(p) 和 isclosed(p) 可用于在查询中测试这两种类型。

path 用下列语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的参数是

(*x*,*y*)

组成路径的各线段的端点。前导方括号（"["）表示开放路径，前导圆括号（"("）表示封闭路径。

路径使用第一种语法输出。注意 Postgres v6.1 之前的版本对路径使用另一种格式：单个前导圆括号、一个"closed"（封闭）标志、一个点数的整数计数，然后是点列表和一个闭合圆括号。
内建函数 upgradepath 用于转换从 v6.1 之前的数据库转储并重新加载的路径。

3.6.5. 多边形

多边形由点集表示。多边形或许应被视为等价于封闭路径，但其存储方式不同，
并且有自己的一套支持例程。

polygon 用下列语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
```

```
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,   xn , yn )
  x1 , y1   , ... ,   xn , yn
```

其中的参数是

(*x*,*y*)

组成多边形边界的各线段的端点。

多边形使用第一种语法输出。注意 Postgres v6.1 之前的版本对多边形使用另一种格式：单个前导圆括号、x 轴坐标列表、y 轴坐标列表，再接一个闭合圆括号。内建函数 `upgradepoly` 用于转换从 v6.1 之前的数据库转储并重新加载的多边形。

3.6.6. 圆

圆由一个圆心和一个半径表示。

`circle` 用下列语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

其中的参数是

(*x*,*y*)

圆的圆心。

r

圆的半径。

圆的输出用第一种语法。

3.7. IPv4 网络与主机地址

`cidr` 类型存储以 CIDR（无类别域间路由）记法指定的网络。`inet` 类型以 CIDR 记法存储主机和网络，并通过表示上的一个简单变化来表示单纯的主机 TCP/IP 地址。

表 3.19. Postgres IPv4 类型

IPV4 类型	存储尺寸	描述	范围
cidr	可变	CIDR 网络	有效的 IPV4 CIDR 块
inet	可变	网络和主机	有效的 IPV4 CIDR 块

3.7.1. CIDR

`cidr` 类型保存一个 CIDR 网络。指定无类别网络的格式是 *x.x.x.x/y*，其中 *x.x.x.x* 是网络，*/y* 是网络掩码的位数。如果省略 */y*，则按较老的有类别命名系统的假定来计算，但至少会大到包含输入中写出的所有八位组。

下面是一些示例：

表 3.20. Postgres IP 类型示例

CIDR 输入	CIDR 显示
192.168.1	192.168.1/24
192.168	192.168.0/24
128.1	128.1/16
128	128.0/16
128.1.2	128.1.2/24
10.1.2	10.1.2/24
10.1	10.1/16
10	10/8

3.7.2. `inet`

`inet` 类型设计为在单个字段中保存关于一台主机的全部信息，包括它所在的 CIDR 式子网。注意，如果你想存储真正的 CIDR 网络，应使用 `cidr` 类型。`inet` 类型与 `cidr` 类型类似，区别在于主机部分的位可以非零。存在可用于提取该字段各组成元素的函数。

此类型的输入格式是 *`x.x.x.x/y`*，其中 *`x.x.x.x`* 是一台互联网主机，*`y`* 是网络掩码的位数。如果省略了 */`y`* 部分，则按 `/32` 对待。输出时，如果 */`y`* 部分为 `/32`，则不会打印它。这样只要省略位數部分，该类型就可以直接用作单纯的主机类型。

第 4 章 操作符

描述Postgres中可用的内建操作符。

Postgres为系统类型提供了大量的内建操作符。这些操作符声明在系统目录 pg_operator中。pg_operator中的每个项都包含实现该操作符的过程的名称以及输入和输出类型的类 OIDs。

要查看"||"字符串连接操作符的所有变体，可以试试

```
    SELECT oprleft, oprright, oprresult, oprcode
    FROM pg_operator WHERE oprname = '||';

oprleft|oprright|oprresult|oprcode
-------+--------+---------+-------
     25|      25|       25|textcat
   1042|    1042|     1042|textcat
   1043|    1043|     1043|textcat
(3 rows)
```

用户可以使用操作符名称来调用操作符，例如：

```
select * from emp where salary < 40000;
```

或者，用户也可以直接调用实现这些操作符的函数。在这种情况下，上面的查询将表达为：

```
select * from emp where int4lt(salary, 40000);
```

psql 有一个命令（\dd）用来显示这些操作符。

4.1. 词法优先级

操作符有优先级，目前优先级被硬编码在解析器中。
大多数操作符具有相同的优先级并且是左结合的。这可能导致不直观的行为；
例如布尔操作符"<"和">"的优先级与布尔操作符"<="和">="不同。

表 4.1. 操作符排序（优先级递减）

元素	优先级	描述
UNION	left	SQL select 构造
::		Postgres 类型转换
[ ]	left	数组定界符
.	left	表/列定界符
-	right	一元负号
:	right	求幂
|	left	间隔的起点
* / %	left	乘法、除法、取模
+ -	left	加法、减法
IS		测试 TRUE、FALSE、NULL

元素	优先级	描述
ISNULL		测试 NULL
NOTNULL		测试非 NULL
（所有其他操作符）		内建和用户定义
IN		集合成员
BETWEEN		包含
OVERLAPS		时间间隔重叠
LIKE		字符串模式匹配
< >		不等于
=	right	等于
NOT	right	逻辑非
AND	left	逻辑交
OR	left	逻辑并

4.2. 通用操作符

这里列出的操作符是为若干种原生数据类型定义的，从数值类型到日期/时间类型都有。

表 4.2. Postgres 操作符

操作符	描述	用法
<	小于吗？	1 < 2
<=	小于或等于吗？	1 <= 2
<>	不相等吗？	1 <> 2
=	相等吗？	1 = 1
>	大于吗？	2 > 1
>=	大于或等于吗？	2 >= 1
‖	连接字符串	'Postgre' ‖ 'SQL'
!!=	NOT IN	3 !!= i
~~	LIKE	'scrappy,marc,hermit' ~~ '%scrappy%'
!~~	NOT LIKE	'bruce' !~~ '%al%'
~	匹配（正则），区分大小写	'thomas' ~ '.*thomas.*'
~*	匹配（正则），不区分大小写	'thomas' ~* '.*Thomas.*'
!~	不匹配（正则），区分大小写	'thomas' !~ '.*Thomas.*'
!~*	不匹配（正则），不区分大小写	'thomas' !~* '.*vadim.*'

4.3. 数值操作符

表 4.3. Postgres 数值操作符

操作符	描述	用法
!	阶乘	3 !
!!	阶乘（左操作符）	!! 3
%	取模	5 % 4

操作符	描述	用法
%	截断	% 4.5
*	乘法	2 * 3
+	加法	2 + 3
-	减法	2 - 3
/	除法	4 / 2
:	自然指数	: 3.0
@	绝对值	@ -5.0
^	求幂	2.0 ^ 3.0
|/	平方根	|/ 25.0
||/	立方根	||/ 27.0

注意

两个操作符":"和";"现在已被弃用，将在下一个版本中移除。请改用等价的函数 exp() 和 ln()。

4.4. 几何操作符

表 4.4. Postgres 几何操作符

操作符	描述	用法
+	平移	'((0,0),(1,1))'::box + '(2.0,0)'::point
-	平移	'((0,0),(1,1))'::box - '(2.0,0)'::point
*	缩放/旋转	'((0,0),(1,1))'::box * '(2.0,0)'::point
/	缩放/旋转	'((0,0),(2,2))'::box / '(2.0,0)'::point
#	相交	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	多边形中的点数	# '((1,0),(0,1),(-1,0))'
##	最近邻点	'(0,0)'::point ## '((2,0),(0,2))'::lseg
&&	重叠吗?	'((0,0),(1,1))'::box && '((0,0),(2,2))'::box
&<	向左重叠吗?	'((0,0),(1,1))'::box &< '((0,0),(2,2))'::box
&>	向右重叠吗?	'((0,0),(3,3))'::box &> '((0,0),(2,2))'::box
<->	...之间的距离	'((0,0),1)'::circle <-> '((5,0),1)'::circle
<<	在其左侧吗?	'((0,0),1)'::circle << '((5,0),1)'::circle
<^	在其下方吗?	'((0,0),1)'::circle <^ '((0,5),1)'::circle

操作符	描述	用法
>>	在其右侧吗？	'((5,0),1)'::circle >> '((0,0),1)'::circle
>^	在其上方吗？	'((0,5),1)'::circle >^ '((0,0),1)'::circle
?#	相交或重叠	'((-1,0),(1,0))'::lseg ?# '((-2,-2),(2,2))'::box;
?-	是水平的吗？	'(1,0)'::point ?- '(0,0)'::point
?-|	是垂直的吗？	'((0,0),(0,1))'::lseg ?-| '((0,0),(1,0))'::lseg
@-@	长度或周长	@-@ '((0,0),(1,0))'::path
?|	是竖直的吗？	'(0,1)'::point ?| '(0,0)'::point
?||	是平行的吗？	'((-1,0),(1,0))'::lseg ?|| '((-1,2),(1,2))'::lseg
@	包含于或位于	'(1,1)'::point @ '((0,0),2)'::circle
@@	…的中心	@@ '((0,0),10)'::circle
~=	等同于	'((0,0),(1,1))'::polygon ~= '((1,1),(0,0))'::polygon

4.5. 时间间隔操作符

时间间隔数据类型`tinterval`是最初的日期/时间类型遗留下来的产物，
不像更现代的类型那样得到良好的支持。这种类型有若干个操作符。

表 4.5. Postgres 时间间隔操作符

操作符	描述	用法
#<	间隔小于吗？	
#<=	间隔小于或等于吗？	
#<>	间隔不相等吗？	
#=	间隔相等吗？	
#>	间隔大于吗？	
#>=	间隔大于或等于吗？	
<#>	转换为时间间隔	
<<	间隔小于吗？	
|	间隔的起点	
~=	等同于	
<?>	时间在间隔内吗？	

4.6. IP V4 CIDR 操作符

表 4.6. PostgresIP V4 CIDR 操作符

操作符	描述	用法
<	小于	'192.168.1.5'::cidr < '192.168.1.6'::cidr

操作符	描述	用法
<=	小于或等于	'192.168.1.5'::cidr <= '192.168.1.5'::cidr
=	等于	'192.168.1.5'::cidr = '192.168.1.5'::cidr
>=	大于或等于	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
>	大于	'192.168.1.5'::cidr > '192.168.1.4'::cidr
<>	不等于	'192.168.1.5'::cidr <> '192.168.1.4'::cidr
<<	包含于	'192.168.1.5'::cidr << '192.168.1/24'::cidr
<<=	包含于或等于	'192.168.1/24'::cidr <<= '192.168.1/24'::cidr
>>	包含	'192.168.1/24'::cidr >> '192.168.1.5'::cidr
>>=	包含或等于	'192.168.1/24'::cidr >>= '192.168.1/24'::cidr

4.7. IP V4 INET 操作符

表 4.7. PostgresIP V4 INET 操作符

操作符	描述	用法
<	小于	'192.168.1.5'::inet < '192.168.1.6'::inet
<=	小于或等于	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	等于	'192.168.1.5'::inet = '192.168.1.5'::inet
>=	大于或等于	'192.168.1.5'::inet >= '192.168.1.5'::inet
>	大于	'192.168.1.5'::inet > '192.168.1.4'::inet
<>	不等于	'192.168.1.5'::inet <> '192.168.1.4'::inet
<<	包含于	'192.168.1.5'::inet << '192.168.1/24'::inet
<<=	包含于或等于	'192.168.1/24'::inet <<= '192.168.1/24'::inet
>>	包含	'192.168.1/24'::inet >> '192.168.1.5'::inet
>>=	包含或等于	'192.168.1/24'::inet >>= '192.168.1/24'::inet

第 5 章 函数

描述 Postgres 中可用的内置函数。

> 许多数据类型都有可用于转换到其他相关类型的函数。此外，还有一些特定类型的函数。有些函数也可以通过操作符使用，并且可能只以操作符的形式记录。

5.1. SQL 函数

SQL 函数是由 SQL92 标准定义的、具有类似函数语法但不能实现为简单函数的结构。

表 5.1. SQL 函数

函数	返回	描述	例子
COALESCE(*list*)	non-NULL	返回列表中第一个非 NULL 值	COALESCE(rle, c2 + 5, 0)
NULLIF(*input*,*value*)	输入或 NULL	若 input = value 则返回 NULL，否则返回 input	NULLIF(c1, 'N/A')
CASE WHEN *expr* THEN *expr* [...] ELSE *expr* END	*expr*	返回第一个为真的 WHEN 子句的表达式	CASE WHEN c1 = 1 THEN 'match' ELSE 'no match' END

5.2. 数学函数

表 5.2. 数学函数

函数	返回	描述	例子
abs(float8)	float8	绝对值	abs(-17.4)
degrees(float8)	float8	弧度转角度	degrees(0.5)
exp(float8)	float8	e 的指定幂	exp(2.0)
ln(float8)	float8	自然对数	ln(2.0)
log(float8)	float8	以 10 为底的对数	log(2.0)
pi()	float8	基本常量	pi()
pow(float8,float8)	float8	数的指定幂	pow(2.0, 16.0)
radians(float8)	float8	角度转弧度	radians(45.0)
round(float8)	float8	舍入到最接近的整数	round(42.4)
sqrt(float8)	float8	平方根	sqrt(2.0)
cbrt(float8)	float8	立方根	cbrt(27.0)
trunc(float8)	float8	截断（朝零方向）	trunc(42.4)
float(int)	float8	把整数转换为浮点	float(2)
float4(int)	float4	把整数转换为浮点	float4(2)
integer(float)	int	把浮点转换为整数	integer(2.0)

上面为 FLOAT8 列出的大多数函数也可用于 NUMERIC 类型。

表 5.3. 超越数学函数

函数	返回	描述	例子
acos(float8)	float8	arccosine	acos(10.0)
asin(float8)	float8	arcsine	asin(10.0)
atan(float8)	float8	arctangent	atan(10.0)
atan2(float8,float8)	float8	arctangent	atan2(10.0,20.0)
cos(float8)	float8	cosine	cos(0.4)
cot(float8)	float8	cotangent	cot(20.0)
sin(float8)	float8	sine	cos(0.4)
tan(float8)	float8	tangent	tan(0.4)

5.3. 字符串函数

SQL92 用特定语法定义了一些字符串函数。其中一些是用其他内部字符串函数实现的。SQL92 支持的字符串类型是 char、varchar 和 text。

表 5.4. SQL92 字符串函数

函数	返回	描述	例子
char_length(string)	int4	字符串的长度	char_length('jose')
character_length(string)	int4	字符串的长度	char_length('jose')
lower(string)	string	把字符串转换为小写	lower('TOM')
octet_length(string)	int4	字符串的存储长度	octet_length('jose')
position(string in string)	int4	指定子串的位置	position('o' in 'Tom')
substring(string [from int] [for int])	string	提取指定子串	substring('Tom' from 2 for 2)
trim([leading|trailing| both] [string] from string)	string	移除字符串两侧的字符	trim(both 'x' from 'x Tomx')
upper(text)	text	把文本转换为大写	upper('tom')

text、varchar() 和 char() 类型还有许多其他的字符串函数可用。其中一些在内部用于实现 SQL92 字符串函数。

表 5.5. 字符串函数

函数	返回	描述	例子
char(text)	char	把 text 转换为 char 类型	char('text string')
char(varchar)	char	把 varchar 转换为 char 类型	char(varchar 'varchar string')
initcap(text)	text	把每个单词的首字母转换为大写	initcap('thomas')
lpad(text,int,text)	text	在左侧填充字符串到指定长度	lpad('hi',4,'??')
ltrim(text,text)	text	移除文本左侧的字符	ltrim('xxxxtrim','x')

函数	返回	描述	例子
textpos(text,text)	text	定位指定子串	position('high','ig')
rpad(text,int,text)	text	在右侧填充字符串到指定长度	rpad('hi',4,'x')
rtrim(text,text)	text	移除文本右侧的字符	rtrim('trimxxxx','x')
substr(text,int[,int])	text	提取指定子串	substr('hi there',3,5)
text(char)	text	把 char 转换为 text 类型	text('char string')
text(varchar)	text	把 varchar 转换为 text 类型	text(varchar 'varchar string')
translate(text,from,to)	text	转换字符串中的字符	translate('12345', '1', 'a')
varchar(char)	varchar	把 char 转换为 varchar 类型	varchar('char string')
varchar(text)	varchar	把 text 转换为 varchar 类型	varchar('text string')

显式为 text 定义的函数大多数也适用于 char() 和 varchar() 参数。

5.4. 日期/时间函数

日期/时间函数提供了一套强大的工具，用于操作各种日期/时间数据类型。
可以用这些类型做算术。

表 5.6. 日期/时间函数

函数	返回	描述	例子
abstime(timestamp)	abstime	转换为 abstime	abstime(timestamp 'now')
age(timestamp)	interval	保留月和年	age(timestamp '1957-06-13')
age(timestamp,timestamp)	interval	保留月和年	age('now', timestamp '1957-06-13')
date_part(text,timestamp)	float8	日期的部分	date_part('dow',timestamp 'now')
date_part(text,interval)	float8	时间的部分	date_part('hour',interval '4 hrs 3 mins')
date_trunc(text,timestamp)	timestamp	截断日期	date_trunc('month', abstime 'now')
interval(reltime)	interval	转换为 interval	interval(reltime '4 hours')
isfinite(timestamp)	bool	是否为有限时间?	isfinite(timestamp 'now')
isfinite(interval)	bool	是否为有限时间?	isfinite(interval '4 hrs')
reltime(interval)	reltime	转换为 reltime	reltime(interval '4 hrs')
timestamp(date)	timestamp	转换为 timestamp	timestamp(date 'today')
timestamp(date,time)	timestamp	转换为 timestamp	timestamp(timestamp '1998-02-24',time '23:07');

函数	返回	描述	例子
to_char(timestamp, text)	text	转换为字符串	to_char(timestamp '1998-02-24','DD');

对于 date_part 和 date_trunc 函数，参数可以是 `year`、`month`、`day`、`hour`、`minute` 和 `second`，还有更专门的量 `decade`、`century`、`millennium`、`millisecond` 和 `microsecond`。date_part 允许用 `dow` 返回星期几，用 `week` 返回 ISO 定义的年中的周，用 `epoch` 返回自 1970 年以来的秒数（对 timestamp），或返回总经过秒数（对 interval）。

5.5. 格式化函数

> **作者**
>
> 由 Karel Zak[1] 于 2000-01-24 编写。

Postgres 格式化函数提供了一套强大的工具，用于把各种数据类型（日期/时间、整数、浮点、numeric）转换为格式化的字符串以及把格式化的字符串转换回原始的数据类型。

> **注意**
>
> 所有格式化函数的第二个参数都是用于转换的模板。

表 5.7. 格式化函数

函数	返回	描述	例子
to_char(timestamp, text)	text	把 timestamp 转换为字符串	to_char(timestamp 'now','HH12:MI:SS')
to_char(int, text)	text	把 int4/int8 转换为字符串	to_char(125, '999')
to_char(float, text)	text	把 float4/float8 转换为字符串	to_char(125.8, '999D9')
to_char(numeric, text)	text	把 numeric 转换为字符串	to_char(numeric '-125.8', '999D99S')
to_date(text, text)	date	把字符串转换为 date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	date	把字符串转换为 timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	把字符串转换为 numeric	to_number('12,454.8-', '99G999D9S')

表 5.8. 用于日期/时间转换的模板

模板	描述
HH	小时（01-12）
HH12	小时（01-12）
HH24	小时（00-23）

[1] mailto:zakkr@zf.jcu.cz

模板	描述
MI	分钟（00-59）
SS	秒（00-59）
SSSS	自午夜以来的秒数（0-86399）
AM or A.M. or PM or P.M.	大写形式的上午/下午指示符
am or a.m. or pm or p.m.	小写形式的上午/下午指示符
Y,YYY	年份（4 位或更多数字）带逗号
YYYY	年份（4 位或更多数字）
YYY	年份的最后 3 位数字
YY	年份的最后 2 位数字
Y	年份的最后一位数字
BC or B.C. or AD or A.D.	大写形式的纪元指示符
bc or b.c. or ad or a.d.	小写形式的纪元指示符
MONTH	全大写的月名（9 个字符）
Month	首字母大写的月名全称（9 个字符）
month	全小写的月名（9 个字符）
MON	缩写的全大写月名（3 个字符）
Mon	缩写的首字母大写月名（3 个字符）
mon	缩写的全小写月名（3 个字符）
MM	月（01-12）
DAY	全大写的星期名（9 个字符）
Day	首字母大写的星期名全称（9 个字符）
day	全小写的星期名（9 个字符）
DY	缩写的全大写星期名（3 个字符）
Dy	缩写的首字母大写星期名（3 个字符）
dy	缩写的全小写星期名（3 个字符）
DDD	一年中的第几天（001-366）
DD	月中的日（01-31）
D	星期几（1-7；SUN=1）
W	月内的周
WW	一年中的周序号
CC	世纪（2 位数字）
J	儒略日（自公元前 4712 年 1 月 1 日以来的天数）
Q	季度
RM	以大写罗马数字表示的月份（I-XII；I=一月）
rm	以小写罗马数字表示的月份（I-XII；I=一月）

所有模板都允许使用前缀和后缀修饰符。前缀修饰符（如 `FM`）对其后紧跟的模板有效；后缀修饰符（如 `TH`）亦然。`FX` 前缀是一个全局修饰符。

表 5.9. 用于日期/时间 to_char() 的模板后缀

后缀	描述	例子
FM	填充模式前缀	FMMonth

后缀	描述	例子
TH	大写序数后缀	DDTH
th	小写序数后缀	DDTH
FX	固定格式全局选项（见下文）	FX Month DD Day
SP	拼写模式（尚未实现）	DDSP

用法说明：

- 如果不使用 FX 选项，to_timestamp 和 to_date 会跳过空白。FX 必须指定为模板中的第一项。
- 反斜杠（"\"）必须写成双反斜杠（"\\"）；例如 '\\HH\\MI\\SS'。
- 引号之间的双引号（""）会被跳过而不解析。如果想在输出中写双引号，必须在它前面加双反斜杠（'\\"），例如 '\\"YYYY Month\\"'。
- to_char 支持不带前导双引号（""）的文本，但引号之间的任何字符串都会被快速处理，可以保证它不会被解释为模板关键字（例如：'"Hello Year: "YYYY'）。

表 5.10. to_char(*numeric*) 的模板

模板	描述
9	具有指定位数的值
0	带前导零的值
.（句点）	小数点
,（逗号）	组（千位）分隔符
PR	尖括号中的负值
S	带负号的负值（使用区域设置）
L	货币符号（使用区域设置）
D	小数点（使用区域设置）
G	组分隔符（使用区域设置）
MI	指定位置上的负号（若数字 < 0）
PL	指定位置上的正号（若数字 > 0）
SG	指定位置上的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH or th	转换为序数
V	移动 n 位数字（见注释）
EEEE	科学计数法。现在不支持。

用法说明：

- 使用 SG、PL 或 MI 格式化的符号不是数字中的锚点；例如，to_char(-12, 'S9999') 产生 '-12'，但 to_char(-12, 'MI9999') 产生 '- 12'。Oracle 实现不允许在 9 前面使用 MI，而是要求 9 在 MI 之前。
- PL、SG 和 TH 是 Postgres 扩展。
- 9 指定一个位数与 9 的个数相同的值。如果没有可用数字则使用空格。

- TH 不转换小于零的值，也不转换十进制数。TH 是 Postgres 扩展。
- V 实际上把输入值乘以 10^*n*，其中 *n* 是 V 后面的数字个数。to_char 不支持 V 与小数点组合使用（例如不允许 "99.9V99"）。

表 5.11. to_char 示例

输入	输出
to_char(now(),'Day, HH12:MI:SS')	'Tuesday , 05:39:18'
to_char(now(),'FMDay, HH12:MI:SS')	'Tuesday, 05:39:18'
to_char(-0.1,'99.99')	' -.10'
to_char(-0.1,'FM9.99')	'-.1'
to_char(0.1,'0.9')	' 0.1'
to_char(12,'9990999.9')	' 0012.0'
to_char(12,'FM9990999.9')	'0012'
to_char(485,'999')	' 485'
to_char(-485,'999')	'-485'
to_char(485,'9 9 9')	' 4 8 5'
to_char(1485,'9,999')	' 1,485'
to_char(1485,'9G999')	' 1 485'
to_char(148.5,'999.999')	' 148.500'
to_char(148.5,'999D999')	' 148,500'
to_char(3148.5,'9G999D999')	' 3 148,500'
to_char(-485,'999S')	'485-'
to_char(-485,'999MI')	'485-'
to_char(485,'999MI')	'485'
to_char(485,'PL999')	'+485'
to_char(485,'SG999')	'+485'
to_char(-485,'SG999')	'-485'
to_char(-485,'9SG99')	'4-85'
to_char(-485,'999PR')	'<485>'
to_char(485,'L999')	'DM 485
to_char(485,'RN')	' CDLXXXV'
to_char(485,'FMRN')	'CDLXXXV'
to_char(5.2,'FMRN')	V
to_char(482,'999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8,'"Pre-decimal:"999" Post-decimal:" .999')	'Pre-decimal: 485 Post-decimal: .800'
to_char(12,'99V999')	' 12000'
to_char(12.4,'99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

5.6. 几何函数

几何类型 `point`、`box`、`lseg`、`line`、`path`、`polygon` 和 `circle` 有一大套原生支持的函数和操作符。

表 5.12. 几何函数

函数	返回	描述	例子
area(object)	float8	项目的面积	area(box '((0,0),(1,1))')
box(box,box)	box	交集框	box(box '((0,0),(1,1))', box '((0.5,0.5),(2,2))')
center(object)	point	项目的中心	center(box '((0,0),(1,2))')
diameter(circle)	float8	圆的直径	diameter(circle '((0,0),2.0)')
height(box)	float8	框的垂直尺寸	height(box '((0,0),(1,1))')
isclosed(path)	bool	是否为闭合路径?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	bool	是否为开放路径?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	float8	项目的长度	length(path '((-1,0),(1,0))')
pclose(path)	path	把路径转换为闭合	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	int4	点数	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	把路径转换为开放路径	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	float8	圆的半径	radius(circle '((0,0),2.0)')
width(box)	float8	水平尺寸	width(box '((0,0),(1,1))')

表 5.13. 几何类型转换函数

函数	返回	描述	例子
box(circle)	box	圆转为框	box('((0,0),2.0)'::circle)
box(point,point)	box	点转为框	box('(0,0)'::point,'(1,1)'::point)
box(polygon)	box	多边形转为框	box('((0,0),(1,1),(2,0))'::polygon)
circle(box)	circle	转换为圆	circle('((0,0),(1,1))'::box)
circle(point,float8)	circle	点转为圆	circle('(0,0)'::point,2.0)
lseg(box)	lseg	框的对角线转为 lseg	lseg('((-1,0),(1,0))'::box)
lseg(point,point)	lseg	点转为 lseg	lseg('(-1,0)'::point,'(1,0)'::point)

函数	返回	描述	例子
path(polygon)	point	多边形转为路径	path('((0,0),(1,1),(2,0))'::polygon)
point(circle)	point	中心	point('((0,0),2.0)'::circle)
point(lseg,lseg)	point	交集	point('((-1,0),(1,0))'::lseg, '((-2,-2),(2,2))'::lseg)
point(polygon)	point	中心	point('((0,0),(1,1),(2,0))'::polygon)
polygon(box)	polygon	12 点多边形	polygon('((0,0),(1,1))'::box)
polygon(circle)	polygon	12 点多边形	polygon('((0,0),2.0)'::circle)
polygon(*npts*,circle)	polygon	npts 多边形	polygon(12,'((0,0),2.0)'::circle)
polygon(path)	polygon	路径转为多边形	polygon('((0,0),(1,1),(2,0))'::path)

表 5.14. 几何升级函数

函数	返回	描述	例子
isoldpath(path)	path	测试路径是否为 v6.1 之前的形式	isoldpath('(1,3,0,0,1,1,2,0)'::path)
revertpoly(polygon)	polygon	转为 v6.1 之前形式	revertpoly('((0,0),(1,1),(2,0))'::polygon)
upgradepath(path)	path	转为 v6.1 之前形式	upgradepath('(1,3,0,0,1,1,2,0)'::path)
upgradepoly(polygon)	polygon	转为 v6.1 之前形式	upgradepoly('(0,1,2,0,1,0)'::polygon)

5.7. IP V4 函数

表 5.15. Postgres IP V4 函数

函数	返回	描述	例子
broadcast(cidr)	text	以文本形式构造广播地址	broadcast('192.168.1.5/24')
broadcast(inet)	text	以文本形式构造广播地址	broadcast('192.168.1.5/24')
host(inet)	text	以文本形式提取主机地址	host('192.168.1.5/24')
masklen(cidr)	int4	计算网络掩码长度	masklen('192.168.1.5/24')
masklen(inet)	int4	计算网络掩码长度	masklen('192.168.1.5/24')
netmask(inet)	text	以文本形式构造网络掩码	netmask('192.168.1.5/24')

第 6 章 类型转换

SQL 查询可能有意或无意地要求在同一表达式中混用不同的数据类型。Postgres 提供了丰富的机制来求值混合类型表达式。

在很多情况下，用户不需要了解类型转换机制的细节。但 Postgres 所做的隐式转换会影响查询的表面结果，用户或程序员可以用显式类型强制转换来定制这些结果。

本章介绍 Postgres 的类型转换机制和约定。关于特定数据类型及允许的函数和操作符的更多信息，请参阅用户指南和程序员指南中的相关章节。

程序员指南中有关于隐式类型转换和强制转换所用确切算法的更多细节。

6.1. 概述

SQL 是一种强类型语言。也就是说，每个数据项都有一个相关联的数据类型，它决定了该数据项的行为和允许的用法。Postgres 拥有一个可扩展的类型系统，比其他 RDBMS 实现更加通用和灵活。因此，Postgres 中大部分类型转换行为应当由一般规则而不是 ad-hoc（特定）启发式规则支配，以使混合类型表达式即使在有用户定义类型时也有意义。

Postgres 的扫描器/解析器只把词法元素解码成五种基本类别：整数、浮点数、字符串、名字和关键字。大多数扩展类型首先被词法化为字符串。SQL 语言定义允许用字符串指定类型名，这一机制可以用来让解析器走上正确的路径。例如，查询

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

中有两个字符串，类型分别为 `text` 和 `point`。如果没有指定类型，则最初会赋予占位类型 `unknown`，并在后面阶段按如下所述解析。

在 Postgres 解析器中有四种基本的 SQL 结构需要专门的类型转换规则：

操作符

Postgres 允许带左一元和右一元（单参数）操作符的表达式，也允许二元（双参数）操作符的表达式。

函数调用

Postgres 类型系统的很大一部分是围绕一套丰富的函数构建的。函数调用有一个或多个参数，对任何具体的查询，这些参数都必须与系统目录中可用的函数匹配。

查询目标

SQL 的 INSERT 语句把查询的结果放到表中。查询中的表达式必须与插入的目标列匹配，必要时还要转换成目标列的类型。

UNION 查询

由于 UNION SELECT 语句的所有查询结果必须出现在一组列中，每个 SELECT 子句的类型必须匹配并转换成统一的集合。

许多一般类型转换规则使用了建立在 Postgres 函数和操作符系统表之上的简单约定。转换规则中还包含一些启发式规则，以便更好地支持 SQL92 标准固有类型（如 `smallint`、`integer` 和 `float`）的约定。

Postgres 解析器使用的约定是：所有类型转换函数都接受一个源类型的参数，并且以目标类型的名称命名。满足这一标准的任何函数都被认为是有效的转换函数，
解析器可以把它当作转换函数使用。
这个简单的假设使解析器无需硬编码就能探索类型转换的可能性，
让扩展的用户定义类型透明地使用这些相同的特性。

解析器中还提供了一个额外的启发式规则，以便对 SQL 标准类型的正确行为做出更好的猜测。定义了五种类型类别：boolean、string、numeric、geometric 和用户自定义。除用户自定义外，每个类别都有一个用于解决候选歧义的"首选类型"。
每个"用户自定义"类型都是自己的"首选类型"，
因此只含一个用户定义类型的有歧义表达式（有多个候选解析解的表达式）
可以解析出单一的最佳选择，而含多个用户定义类型的表达式仍会有歧义并报错。

候选解只落在一个类型类别内的有歧义表达式很可能得到解析，
而候选跨越多个类别的有歧义表达式则很可能报错并请求用户澄清。

6.1.1. 指南

所有类型转换规则的设计都考虑了以下几条原则：

- 隐式转换绝不应有令人意外或不可预测的结果。
- 解析器没有先验知识的用户定义类型在类型层级中应该更"高"。在混合类型表达式中，
  固有类型总是应当转换成用户定义类型（当然，只在需要转换时如此）。
- 用户定义的类型之间没有关联。目前，Postgres 没有关于类型之间关系的信息，只有内置类型的硬编码启发式规则以及基于目录中可用函数的隐式关系。
- 如果查询不需要隐式类型转换，解析器或执行器就不应有额外的开销。也就是说，
  如果查询构造良好且类型已经匹配，查询就应当继续进行，不必在解析器上花费额外时间，
  也不必向查询中引入不必要的隐式转换函数。

  此外，如果某个查询通常需要为函数做隐式转换，
  而之后用户用正确的参数类型定义了一个显式函数，解析器就应使用这个新函数，
  不再用旧函数做隐式转换。

6.2. 操作符

6.2.1. 转换过程

操作符求值

1. 在 pg_operator 系统目录中检查精确匹配。

   a. (可选的) 如果二元操作符的一个参数是 `unknown`，就假定它与另一个参数的类型相同。

   b. 交换参数，寻找与一个指向自身为可交换的操作符的精确匹配。如果找到，
   就在解析树中交换参数并使用该操作符。

2. 寻找最佳匹配。

   a. (可选的) 列出所有同名的操作符。

b. 如果列表中只有一个操作符，输入类型可强制转换时就使用它，
类型不能强制转换时就报错。

c. 保留类型显式匹配最多的操作符。如果没有显式匹配则全部保留并进入下一步。
如果只剩一个候选，类型可强制转换时就使用它。

d. 如果有输入参数是"unknown"，把输入候选归类为 boolean、numeric、string、geometric 或用户自定义。如果类别混合，或者用户定义类型多于一个，就报错，因为没有更多线索就无法推断出正确的选择。如果只有一个类别，
就把"首选类型"赋给先前为"unknown"的输入列。

e. 选择类型精确匹配最多且与上一步中每个列类别的 "首选类型"匹配的候选。如果候选仍多于一个，或者一个也没有，就报错。

6.2.2. 示例

6.2.2.1. 求幂操作符

目录中只定义了一个求幂运算符，它接受 `float8` 参数。扫描器给这个查询表达式的两个参数都赋予了初始类型 `int4`：

```
tgl=> select 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

于是解析器对两个操作数都做类型转换，查询等价于

```
tgl=> select float8(2) ^ float8(3) AS "Exp";
 Exp
-----
   8
(1 row)
```

或

```
tgl=> select 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

> **注意**
>
> 最后这种形式的开销最小，因为没有调用任何函数来做隐式类型转换。
> 这对小查询不是问题，但可能影响涉及大表的查询的性能。

6.2.2.2. 字符串连接

类字符串的语法既用于处理字符串类型，也用于处理复杂的扩展类型。
未指定类型的字符串会与可能的操作符候选匹配。

一个参数未指定类型：

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
 Text and Unknown
------------------
 abcdef
(1 row)
```

在这种情况下，解析器会查看是否存在一个两边参数都接受text的操作符。既然存在，它就会假定第二个参数应解释为text类型。

未指定类型上的串接：

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

这种情况下没有关于使用哪个类型的初始提示，因为查询中没有指定类型。
于是解析器查找所有候选操作符，发现所有候选的所有参数都是 string 类型。它为这个查询选择 string 的"首选类型" text。

> **注意**
>
> 如果用户定义了一个新类型并为它定义了 "||" 操作符，这个查询就会不再按原样成功。解析器此时会有来自两个类别的候选类型，无法决定用哪一个。

6.2.2.3. 阶乘

这个例子说明了一个有趣的结果。传统上，阶乘运算符只为整数定义。Postgres 的运算符目录中阶乘只有一个条目，接受整数操作数。如果给定非整数的数值参数，Postgres 会试图把该参数转换成整数来求阶乘。

```
tgl=> select (4.3 !);
 ?column?
----------
       24
(1 row)
```

> **注意**
>
> 当然，这在数学上是可疑的结果，因为原则上非整数的阶乘没有定义。不过，
> 数据库的职责不是教数学，而是做数据操纵的工具。如果用户选择对浮点数求阶乘，
> Postgres 也会照办。

6.3. 函数

函数求值

1. 在 pg_proc 系统目录中检查精确匹配。

2. 寻找最佳匹配。

   a. 列出所有同名且参数个数相同的函数。

   b. 如果列表中只有一个函数，输入类型可强制转换时就使用它，
   类型不能强制转换时就报错。

   c. 保留类型显式匹配最多的函数。如果没有显式匹配则全部保留并进入下一步。
   如果只剩一个候选，类型可强制转换时就使用它。

   d. 如果有输入参数是"unknown"，把输入的候选参数归类为 boolean、numeric、string、geometric 或用户自定义。如果类别混合，或者用户定义类型多于一个，就报错，因为没有更多线索就无法推断出正确的选择。如果只有一个类别，
   就把"首选类型"赋给先前为"unknown"的输入列。

   e. 选择类型精确匹配最多且与上一步中每个列类别的 "首选类型"匹配的候选。如果候选仍多于一个，或者一个也没有，就报错。

6.3.1. 示例

6.3.1.1. 阶乘函数

pg_proc 目录中只定义了一个阶乘函数。因此下面的查询自动把 `int2` 参数转换为 `int4`：

```
tgl=> select int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

并且实际上被解析器变换为

```
tgl=> select int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

6.3.1.2. 子串函数

pg_proc 中声明了两个 `substr` 函数。但只有一个接受两个参数，类型是 `text` 和 `int4`。

如果用未指定类型的字符串常量调用，该类型会直接与唯一的候选函数类型匹配：

```
tgl=> select substr('1234', 3);
 substr
--------
     34
(1 row)
```

如果该字符串被声明为 `varchar` 类型（例如它来自一张表时就是这种情况），则解析器会试图把它强制转换成 `text`：

```
tgl=> select substr(varchar '1234', 3);
 substr
```

```
--------
      34
(1 row)
```

它被解析器变换成

```
tgl=> select substr(text(varchar '1234'), 3);
 substr
--------
      34
(1 row)
```

> **注意**
>
> 解析器中有一些启发式规则来优化 `char`、`varchar` 和 `text` 类型之间的关系。在这个例子中，`substr` 直接用 `varchar` 字符串调用，而不是插入显式的转换调用。

而且，如果用 `int4` 调用该函数，解析器会试图把它转换成 `text`：

```
tgl=> select substr(1234, 3);
 substr
--------
      34
(1 row)
```

实际执行为

```
tgl=> select substr(text(1234), 3);
 substr
--------
      34
(1 row)
```

6.4. 查询目标

目标求值

1. 检查是否与目标类型精确匹配。
2. 必要时尝试把表达式直接强制转换成目标类型。
3. 如果目标是定长类型（例如声明了长度的 `char` 或 `varchar`），则尝试寻找一个与类型同名、接受两个参数的尺寸调整函数，第一个参数是类型名，第二个是 integer 长度。

6.4.1. 示例

6.4.1.1. varchar 存储

对于声明为 `varchar(4)` 的目标列，下面的查询确保目标的大小正确：

```
tgl=> CREATE TABLE vv (v varchar(4));
```

```
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT * FROM vv;
  v
------
 abcd
(1 row)
```

6.5. UNION 查询

UNION 结构有些不同，它必须把可能不相似的类型匹配起来构成单一结果集。

UNION 求值

1. 检查所有结果的类型是否一致。
2. 把来自各 UNION 子句的每个结果强制转换成与第一个 SELECT 子句或目标列相同的类型。

6.5.1. 示例

6.5.1.1. 未充分指定的类型

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

6.5.1.2. 简单 UNION

```
tgl=> SELECT 1.2 AS "Float8" UNION SELECT 1;
 Float8
--------
      1
    1.2
(2 rows)
```

6.5.1.3. 转置 UNION

union 的类型被强制匹配 union 中第一个/顶端子句的类型：

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
 All integers
--------------
            1
            2
            3
(3 rows)
```

另一种解析器策略可以是选择这一组中的"最佳"类型，但由于解析器使用了精巧的递归技术，这更加困难。不过，在选择 into 表时会使用"最佳"类型：

```
tgl=> CREATE TABLE ff (f float);
CREATE
tgl=> INSERT INTO ff
tgl-> SELECT 1
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
INSERT 0 3
tgl=> SELECT f AS "Floating point" from ff;
  Floating point
------------------
                1
 2.20000004768372
              3.3
(3 rows)
```

第 7 章 索引和键

索引通常用于增强数据库性能。应该把索引定义在重复查询中用作限定条件的表列（或类属性）上。不当使用会导致性能变慢，因为在有索引的情况下，更新和插入的时间会增加。

索引也可以用来强制表的主键唯一。当索引被声明为 UNIQUE 时，将不允许存在多个具有相同索引项的表行。出于这种目的时，目标是确保数据一致性而不是改善性能，
因此上面关于不当使用的告诫并不适用。

可以定义两种形式的索引：

- 对于值索引，索引的键字段用列名指定；如果索引访问方法支持多列索引，可以指定多个列。

- 对于函数索引，索引定义在一个函数应用于单个类的一个或多个属性所得的结果上。
  即使函数使用多个输入字段，这也是一个单列索引（即函数结果）。
  函数索引可用于基于某些操作符快速访问数据，
  这些操作符通常需要经过某种变换才能应用到基础数据上。

Postgres 为索引提供了 btree、rtree 和 hash 访问方法。btree 访问方法是 Lehman-Yao 高并发 btree 的一种实现。rtree 访问方法使用 Guttman 的二次分裂算法实现标准 rtree。hash 访问方法是 Litwin 线性散列的一种实现。我们提到这些算法只是为了说明所有这些访问方法都是完全动态的，不需要定期优化（例如，与静态散列访问方法的情况不同）。

只要被索引属性参与的比较使用下列操作符之一，Postgres 查询优化器就会考虑使用 btree 索引：<, <=, =, >=, >

只要被索引属性参与的比较使用下列操作符之一，Postgres 查询优化器就会考虑使用 rtree 索引：<<, &<, &>, >>, @, ~=, &&

只要被索引属性参与的比较使用 = 操作符，Postgres 查询优化器就会考虑使用 hash 索引。

当前，只有 btree 访问方法支持多列索引。默认最多可以指定 16 个键（在构建 Postgres 时可以改变这个限制）。

可以为索引的每一列指定一个操作符类。操作符类确定索引针对该列使用哪些操作符。例如，
建立在四字节整数上的 btree 索引会使用`int4_ops`类；该操作符类包含针对四字节整数的比较函数。实际中，字段的默认操作符类通常就足够了。之所以需要操作符类，
主要是因为对于某些数据类型，可能存在不止一种有意义的排序方式。例如，
可能希望按绝对值或实部对复数数据类型排序。可以为该数据类型定义两个操作符类，
并在创建索引时选择合适的类。此外还有一些特殊用途的操作符类：

- 操作符类`box_ops`和`bigbox_ops`都支持`box`数据类型上的 rtree 索引。它们的区别在于，`bigbox_ops`会将框的坐标按比例缩小，以避免在对非常大的浮点坐标做乘法、
  加法和减法时出现浮点异常。如果你的矩形所在的范围大约在 20,000 个单位见方或更大，就应该使用`bigbox_ops`。

- `int24_ops`操作符类用于在 int2 数据上构建索引，并在查询限定条件中与 int4 数据进行比较。类似地，`int42_ops`支持在 int4 数据上构建索引，以便在查询中与 int2 数据进行比较。

下面的查询会显示所有已定义的操作符类：

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
```

```
FROM pg_am am, pg_amop amop,
     pg_opclass opc, pg_operator opr
WHERE amop.amopid = am.oid AND
      amop.amopclaid = opc.oid AND
      amop.amopopr = opr.oid
ORDER BY acc_name, ops_name, ops_comp
```

使用 DROP INDEX 来删除索引。

7.1. 键

作者

由 Herouth Maoz[1] 撰写。它最初于 1998-03-02 发表在用户邮件列表上，作为对问题 “What is the difference between PRIMARY KEY and UNIQUE constraints?” 的回答。

```
Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE
```

下面两者有什么区别：

```
PRIMARY KEY(fields,...) 和
UNIQUE (fields,...)
```

- 这是一个别名吗？
- 如果 `PRIMARY KEY` 已经是唯一的，那为什么还会有另一种名为 `UNIQUE` 的键？

主键是用来标识特定行的字段。例如，用社会保障号码标识一个人。

一个单纯的 UNIQUE 字段组合与标识行无关，它只是一种完整性约束。例如，我有若干链接集合。每个集合由一个唯一的编号标识，这个编号就是主键。这个键用于关系。

不过，我的应用还要求每个集合有一个唯一的名称。
为什么？为了让想要修改集合的人能够识别它。如果你有两个名为"Life Science"的集合，就很难知道究竟标记为 24433 的那个才是你需要的，而标记为 29882 的那个不是。

因此，用户通过名称来选择集合。我们因而在数据库中确保名称是唯一的。但是，
数据库中没有其他表通过集合名称与 collections 表关联，那样做会非常低效。

而且，集合名称尽管唯一，实际上并不定义这个集合！例如，如果有人决定把集合的名称从"Life Science"改为"Biology"，它仍然是同一个集合，只是名称不同而已。只要名称是唯一的，
这就没有问题。

所以：

- 主键：
  - 用于标识行并与其他数据相关联。
  - 不可能（或很难）更新。
  - 不应允许 NULL。

[1] herouth@oumail.openu.ac.il

- 唯一字段：
  - 用作访问行的另一种途径。
  - 可以更新，只要保持唯一。
  - 可以接受 NULL。

至于为什么标准 SQL 语法中没有显式定义非唯一键？你必须理解，索引是依赖于实现的。SQL 不定义实现，只定义数据库中数据之间的关系。Postgres 确实允许非唯一索引，但用于强制 SQL 键的索引总是唯一的。

因此，你可以按列的任意组合查询一个表，即使你在这些列上并没有索引。索引只是每种 RDBMS 提供给你的实现辅助手段，目的是让常用的查询执行得更高效。有些 RDBMS 还可能提供额外的手段，例如把键保存在主存中。它们会有一个特殊命令，例如

```
CREATE MEMSTORE ON <table> COLUMNS <cols>
```

（这不是一个真实存在的命令，只是一个例子。）

事实上，当你创建主键或唯一的字段组合时，SQL 规范中没有任何地方说会创建索引，也没有说按键检索数据会比顺序扫描更高效！

所以，如果你想用一个不唯一的字段组合作为辅助键，
其实完全不必指定任何东西——直接按那个组合开始检索就行了！不过，如果你想让检索高效，
就得求助于你的 RDBMS 供应商提供的手段——无论是索引、我虚构的 MEMSTORE 命令，还是一个智能的 RDBMS：它会根据你曾基于特定键组合发送过许多查询这一事实，在你不知情的情况下创建索引……（它从经验中学习）。

7.2. 部分索引

> **作者**
>
> 这出自 Paul M. Aoki[2] 于 1998-08-11 在邮件列表上对某个问题的回复。

部分索引是建立在表的一个子集上的索引；这个子集由一个谓词定义。Postgres 曾支持带任意谓词的部分索引。我相信 IBM 的 DB2 for AS/400 支持使用单子句谓词的部分索引。

部分索引的主要动机是：如果你提出的所有能够有利可图地使用索引的查询都落在某个特定范围内，那为什么要对整个表建索引、承担随之而来的空间/时间开销呢？（还有其他一些原因；
细节参见 Stonebraker, M, 1989b。）

构建、更新和查询部分索引的机制并不算太糟。麻烦的部分在于索引选择（我该建哪些索引？）和查询优化（我该用哪些索引？）；
也就是那些涉及判断哪些谓词能以某种有用的方式匹配工作负载/查询的部分。
对于熟悉数据库理论的人来说，这些问题基本上与相应的物化视图问题类似，
只是代价参数和公式不同。对于标准的有序 SQL 类型，这些问题在一般情况下就已经是难题；而对于黑盒扩展类型则是超级难题，因为选择率估计技术还太粗糙。

更多信息请参阅 Stonebraker, M, 1989b、Olson, 1993 和 Seshardri, 1995。

[2] aoki@CS.Berkeley.EDU

第 8 章 数组

> **注意**
>
> 这里必须变成一个关于数组行为的章节。有人志愿吗？- thomas 1998-01-12

Postgres允许把类的属性定义为变长的多维数组。
可以创建任何内建类型或用户定义类型的数组。为了说明它们的用途，我们创建这个类：

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

上面的查询将创建一个名为 sal_emp 的类，其中有一个 text 字符串（name）、一个 int4 的一维数组（pay_by_quarter，表示员工的季度薪水）以及一个 text 的二维数组（schedule，表示员工的周计划）。现在我们做一些 INSERTS；注意，向数组追加时，我们把值用花括号括起来并用逗号分隔。如果你了解 C，这很像初始化结构的语法。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

现在，我们可以在 sal_emp 上运行一些查询。首先，我们展示如何一次访问数组的一个元素。下面的查询检索第二季度薪水发生变化的员工的姓名：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

Postgres 对数组使用 "one-based"（从 1 开始）的编号约定——也就是说，一个有 n 个元素的数组从 array[1] 开始，以 array[n] 结束。

这个查询检索所有员工的第三季度薪水：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

我们也可以访问数组的任意切片或子数组。数组切片通过为一个或多个数组维度写 "lower subscript : upper subscript" 来表示。下面的查询检索 Bill 周计划前两天的第一项：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
--------------------
 {{"meeting"},{""}}
(1 row)
```

我们也可以写成

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

结果相同。

可以完整地替换一个数组值：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

或者在单个项上更新：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

或者在切片上更新：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

目前无法调整数组值的大小，只能完整替换；例如，我们不能通过对 array[5] 的单次赋值把一个四元素的数组值变成五元素的值。

CREATE TABLE 的语法允许定义定长数组：

```
CREATE TABLE tictactoe (
    squares   int4[3][3]
);
```

然而，当前的实现并不强制数组的大小限制——其行为与未指定长度的数组相同。

第 9 章 继承

我们来创建两个类。capitals 类包含各州首府，它们同时也是城市。自然地，capitals 类应当继承自 cities。

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

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

在这种情况下，capitals 的一个实例从其父类 cities 继承所有属性（name、population 和 altitude）。属性 name 的类型是 `text`，这是 Postgres 用于变长 ASCII 字符串的原生类型。属性 population 的类型是 `float`，这是 Postgres 用于双精度浮点数的原生类型。州首府有一个额外的属性 state，表示它所在的州。在 Postgres 中，一个类可以从零个或多个其他类继承，而查询既可以只引用一个类的所有实例，也可以引用一个类及其所有后代的所有实例。

> **注意**
>
> 继承层次实际上是一个有向无环图。

例如，下面的查询找出所有位于海拔 500 英尺或以上的城市：

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

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

另一方面，要找出所有位于海拔 500 英尺以上的城市（包括州首府）的名称，查询是：

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

它返回：

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

这里 cities 后面的 "*" 表示查询应在 cities 以及继承层次中位于 cities 之下的所有类上执行。我们已经讨论过的许多命令——`SELECT`、`UPDATE` 和 `DELETE`——都支持这种 "*" 记法，其他命令如 `ALTER TABLE` 也是如此。

第 10 章 PL/pgSQL - SQL 过程语言

PL/pgSQL 是Postgres数据库系统的一个可装载过程语言。

这个软件包最初由 Jan Wieck 编写。

10.1. 概述

PL/pgSQL 的设计目标是创建一种可装载的过程语言，它

- 可以用来创建函数和触发器过程，
- 向SQL语言中增加控制结构，
- 能执行复杂的计算，
- 继承所有用户定义的类型、函数和操作符，
- 可以被定义为服务器所信任，
- 易于使用。

PL/pgSQL 调用处理器在后端第一次调用函数时解析函数的源文本并生成一棵内部二进制指令树。生成的字节码在调用处理器中通过函数的对象 ID 来标识。这确保了通过 DROP/CREATE 序列更改函数会立即生效，而不需要建立新的数据库连接。

对于函数中使用的所有表达式和SQL语句，PL/pgSQL 字节码解释器都会使用 SPI 管理器的 SPI_prepare() 和 SPI_saveplan() 函数创建一个预备好的执行计划。这是在该语句第一次在 PL/pgSQL 函数中被处理时完成的。因此，一个包含大量可能需要执行计划的条件语句的函数，只会准备和保存那些在数据库连接的整个生命期内真正被用到的计划。

除了用户定义类型的输入/输出转换和计算函数之外，凡是能用 C 语言函数定义的东西也都可以用 PL/pgSQL 完成。你可以创建复杂条件计算函数，之后用它们来定义操作符，或在函数索引中使用它们。

10.2. 描述

10.2.1. PL/pgSQL 的结构

PL/pgSQL 语言不区分大小写。所有关键字和标识符都可以用大小写混合的形式使用。

PL/pgSQL 是一种面向块的语言。块的定义是

```
[<<label>>]
[DECLARE
    declarations]
BEGIN
    statements
END;
```

块的语句部分中可以有任意数量的子块。子块可以用来把变量隐藏在语句块的外部世界之外。
块前面的声明部分中声明的变量在每次进入该块时都会被初始化为它们的默认值，
而不是每次函数调用只初始化一次。

重要的一点是不要误解 PL/pgSQL 中用于给语句分组的 BEGIN/END 与用于事务控制的数据库命令的含义。函数和触发器过程不能开始或提交事务，而且Postgres没有嵌套事务。

10.2.2. 注释

PL/pgSQL 中有两种注释。双横线'--'开始一个延伸到行尾的注释。'/*'开始一个块注释，它延伸到下一个'*/'出现的位置。块注释不能嵌套，但双横线注释可以被包进块注释中，
而双横线也可以隐藏块注释的定界符'/*'和'*/'。

10.2.3. 声明

块或其子块中使用的所有变量、行和记录都必须在块的声明部分中声明，
唯一的例外是遍历一段整数值范围的 FOR 循环的循环变量。传递给 PL/pgSQL 函数的参数会用通常的标识符 $n 自动声明。声明的语法如下：

name [CONSTANT] *type*> [NOT NULL] [DEFAULT | := *value*];

声明一个指定基本类型的变量。如果变量被声明为 CONSTANT，它的值不能被更改。如果指定了 NOT NULL，赋予 NULL 值会导致一个运行时错误。由于所有变量的默认值都是 SQL的 NULL 值，所有声明为 NOT NULL 的变量还必须指定一个默认值。

默认值在每次函数被调用时都会被求值。因此把 '*now*'赋给一个 *datetime*类型的变量会让该变量拥有实际这次函数调用的时间，而不是函数被预编译为字节码时的时间。

nameclass%ROWTYPE;

声明一个具有给定类结构的行。类必须是数据库中已存在的表或视图的名字。
行的字段用点号记法访问。函数的参数可以是复合类型（完整的表行）。这种情况下，
对应的标识符 $n 将是一个行类型，但它必须用下面描述的 ALIAS 命令声明别名。行中只能访问表行的用户属性，不能访问 Oid 或其他系统属性（因为该行可能来自一个视图，而视图的行没有有用的系统属性）。

行类型的字段继承表中该字段的尺寸或精度（对于 char() 等数据类型）。

name RECORD;

记录与行类型类似，但它们没有预定义的结构。它们被用在选取和 FOR 循环中，保存 SELECT 操作产生的一个实际数据库行。同一个记录可以用于不同的选取。当记录中没有实际行时访问记录或试图给记录字段赋值都会导致运行时错误。

触发器中的 NEW 和 OLD 行会作为记录交给过程。这是必要的，因为在Postgres中同一个触发器过程可以处理不同表的触发器事件。

name ALIAS FOR $n;

为了让代码有更好的可读性，可以为函数的位置参数定义一个别名。

当复合类型作为参数传给函数时，这种别名声明是必需的。SQL 函数中的点号记法 $1.salary 在 PL/pgSQL 中是不允许的。

RENAME *oldname* TO *newname*;

更改变量、记录或行的名字。这在需要在触发器过程内部用另一个名字引用 NEW 或 OLD 时有用。

10.2.4. 数据类型

变量的类型可以是数据库中任何已有的基本类型。上面声明部分中的*type*定义为：

- Postgres基本类型
- *variable*%TYPE
- *class.field*%TYPE

*variable*是此前在同一函数中声明、在此处可见的变量的名字。

*class*是一个已存在的表或视图的名字，其中*field*是某个属性的名字。

使用*class.field*%TYPE 会让 PL/pgSQL 在后端生命期内第一次调用函数时查找属性的定义。假设有一张带 char(20) 属性的表和一些用局部变量处理其内容的 PL/pgSQL 函数。现在有人觉得 char(20) 不够了，转储表、删除它、把该属性重建为 char(40) 再恢复数据。哈--他忘了那些函数。函数内部的计算会把值截断到 20 个字符。但如果它们是用 *class.field*%TYPE 声明定义的，它们就会自动处理尺寸的变化，即使新表模式把该属性定义为 text 类型也没问题。

10.2.5. 表达式

PL/pgSQL 语句中使用的所有表达式都由后端的执行器处理。表面上包含常量的表达式实际上可能需要运行时求值（例如 datetime 类型的 'now'），因此 PL/pgSQL 解析器不可能识别除 NULL 关键字之外的真正常量值。所有表达式都通过执行查询

```
SELECT expression
```

使用 SPI 管理器来求值。在表达式中，变量标识符的出现会被参数替代，而变量的实际值则通过参数数组传递给执行器。PL/pgSQL 函数中使用的所有表达式都只准备和保存一次。

Postgres主解析器所做的类型检查对常量值的解释有一些副作用。详细说来，
下面这两个函数的行为是有差别的：

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

和

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime datetime;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
```

```
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

的行为。对于 logfunc1()，Postgres主解析器在为 INSERT 准备计划时就知道字符串'now'应当被解释为 datetime，因为 logtable 的目标字段就是该类型。于是它会在此刻把它变成一个常量，而这个常量值随后在后端的整个生命期内 logfunc1() 的所有调用中都被使用。不用说，这并不是程序员想要的结果。

对于 logfunc2()，Postgres主解析器不知道'now'应当变成什么类型，因此它返回一个包含字符串'now' 的 text 数据类型。在向局部变量 curtime 赋值期间，PL/pgSQL 解释器通过调用 text_out() 和 datetime_in() 函数把这个字符串转换成 datetime 类型。

Postgres主解析器所做的这种类型检查是在 PL/pgSQL 基本完成之后才实现的。这是 6.3 与 6.4 之间的一个差异，影响所有使用 SPI 管理器预备计划特性的函数。以上述方式使用局部变量是目前 PL/pgSQL 中让这些值被正确解释的唯一办法。

如果在表达式或语句中使用了记录字段，
这些字段的数据类型在同一表达式的各次调用之间不应改变。
在编写为多张表处理事件的触发器过程时要牢记这一点。

10.2.6. 语句

凡是没有被 PL/pgSQL 解析器按下面说明理解的东西，都会被放进一个查询并发送到数据库引擎执行。这样的查询不应返回任何数据。

赋值

把一个值赋给变量或行/记录字段写作

identifier := *expression*;

如果表达式的结果数据类型与变量的数据类型不匹配，或者变量具有已知的尺寸/精度（如 char(20)），结果值会被 PL/pgSQL 字节码解释器使用结果类型的输出函数和变量类型的输入函数隐式转换。注意这可能导致类型的输入函数产生运行时错误。

把一个完整的选取赋给记录或行可以这样完成

SELECT INTO *target expressions* FROM ...;

*target*可以是记录、行变量，或者由变量和记录/行字段组成的逗号分隔列表。

如果用一行或一个变量列表作为目标，所选中的值必须与目标的结构完全匹配，
否则会发生运行时错误。FROM 关键字后面可以跟 SELECT 语句允许的任何有效的限定条件、分组、排序等。

有一个名为 FOUND 的 bool 类型特殊变量，可以在 SELECT INTO 之后立即使用它来检查赋值是否成功。

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

如果选取返回多行，只有第一行会被移入目标字段。其余的行都会被无声地丢弃。

调用另一个函数

Prostgres数据库中定义的所有函数都返回一个值。因此，调用函数的通常方式是执行一个 SELECT 查询或做一次赋值（产生一个 PL/pgSQL 内部的 SELECT）。但有时人们并不关心函数的结果。

```
PERFORM query
```

通过 SPI 管理器执行一个'SELECT *query*' 并丢弃结果。像局部变量这样的标识符仍然会被替换为参数。

从函数返回

```
RETURN expression
```

函数终止，*expression*的值会被返回给上层执行器。函数的返回值不能没有定义。
如果控制到达函数顶层块的末尾而没有碰到 RETURN 语句，就会发生运行时错误。

表达式的结果会被自动转换成函数的返回类型，转换方式与赋值中描述的相同。

中止与消息

正如上面的例子所示，有一条 RAISE 语句可以把消息抛进 Postgres的 elog 机制。

```
RAISE level format'' [, identifier [...]];
```

在格式中，"%"用作后续逗号分隔标识符的占位符。可用的级别有　DEBUG（在生产运行的数据库中被无声地抑制）、NOTICE（写进数据库日志并转发给客户端应用）和
EXCEPTION（写进数据库日志并中止事务）。

条件语句

```
IF expression THEN
    statements
[ELSE
    statements]
END IF;
```

*expression*必须返回一个至少可以转换为布尔类型的值。

循环

有多种类型的循环。

```
[<<label>>]
LOOP
    statements
END LOOP;
```

一个必须由 EXIT 语句显式终止的无条件循环。可选的标签可以被嵌套循环的 EXIT 语句用来指定应该终止哪一层嵌套。

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

一个条件循环，只要*expression*的求值为真它就一直执行。

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

一个遍历一段整数值范围的循环。变量 *name*被自动创建为 integer 类型并且只存在于循环内部。给出范围下界和上界的两个表达式只在进入循环时被求值。迭代步长总是 1。

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

记录或行会被赋以 select 子句产生的所有行，并且语句对每一行都执行。如果循环被 EXIT 语句终止，最后赋值的行在循环之后仍然可以访问。

```
EXIT [ label ] [ WHEN expression ];
```

如果没有给出*label*，最内层的循环会被终止，接下来执行 END LOOP 之后的语句。如果给出了 *label*，它必须是当前或嵌套循环块某个上层的标签。此时被命名的循环或块被终止，控制转移到该循环/块对应的 END 之后的语句。

10.2.7. 触发器过程

PL/pgSQL 可以用来定义触发器过程。它们用通常的 CREATE FUNCTION 命令创建，是一个没有参数、返回类型为 OPAQUE 的函数。

用作触发器过程的函数有一些Postgres 特有的细节。

首先它们的顶层块声明部分中会自动创建一些特殊变量。它们是

NEW

数据类型 RECORD；在行级触发器中保存 INSERT/UPDATE 操作的新数据库行的变量。

OLD

数据类型 RECORD；在行级触发器中保存 UPDATE/DELETE 操作的旧数据库行的变量。

TG_NAME

数据类型 name；包含实际被触发的触发器名字的变量。

TG_WHEN

数据类型 text；依据触发器定义而是'BEFORE'或'AFTER'之一的字符串。

TG_LEVEL

数据类型 text；依据触发器定义而是'ROW'或'STATEMENT'之一的字符串。

TG_OP

数据类型 text；'INSERT'、'UPDATE'或'DELETE'之一的字符串，指出触发器实际是为哪个操作而触发。

TG_RELID

数据类型 oid；导致触发器调用的表的对象 ID。

TG_RELNAME

数据类型 name；导致触发器调用的表的名字。

TG_NARGS

数据类型 integer；CREATE TRIGGER 语句中给触发器过程的参数个数。

TG_ARGV[]

text 类型的数组；来自 CREATE TRIGGER 语句的参数。索引从 0 开始计数并且可以用表达式给出。无效的索引（< 0 或 >= tg_nargs）得到 NULL 值。

其次它们必须返回 NULL，或者返回一个恰好包含触发该触发器的表的结构的记录/行。AFTER 触发器总是返回 NULL 值也没有影响。BEFORE 触发器返回 NULL 时会通知触发器管理器跳过对这一实际行的操作。否则，返回的记录/行会在操作中替换被插入/更新的行。可以直接在 NEW 中替换单个值然后返回它，也可以构造一个完整的新记录/行来返回。

10.2.8. 异常

Postgres没有一个非常聪明的异常处理模型。每当解析器、规划器/
优化器或执行器判定一条语句无法再继续处理时，整个事务都会被中止，
系统跳回主循环去从客户端应用获取下一条查询。

可以钩进错误机制来注意到这种情况的发生。但目前无法判断究竟是什么导致了中止（输入/
输出转换错误、浮点错误、解析错误）。而且此时数据库后端可能处于不一致状态，
因此返回上层执行器或发出更多命令可能会损坏整个数据库。即便可以，
此时事务已中止的信息已经发送给了客户端应用，恢复操作没有任何意义。

因此，当 PL/pgSQL 在函数或触发器过程执行期间遇到中止时，它目前唯一会做的事情就是写一些额外的 DEBUG 级别日志消息，指出这件事发生在哪个函数中以及在哪里（行号和语句类型）。

10.3. 示例

这里只有几个函数用来展示编写 PL/pgSQL 函数有多么容易。想要更复杂的例子，程序员可以去看 PL/pgSQL 的回归测试。

用 PL/pgSQL 编写函数的一个痛苦细节是单引号的处理。CREATE FUNCTION 中函数的源文本必须是一个字面量字符串。字面量字符串内部的单引号必须加倍或者用反斜线引用。

我们仍在寻找优雅的替代方案。在此期间，应当像下面的例子那样把单引号加倍。
Postgres未来版本中对这一问题的任何解决方案都将是向上兼容的。

10.3.1. 一些简单的 PL/pgSQL 函数

下面两个 PL/pgSQL 函数与 C 语言函数讨论中的对应函数完全相同。

```
CREATE FUNCTION add_one (int4) RETURNS int4 AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION concat_text (text, text) RETURNS text AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

10.3.2. 操作复合类型的 PL/pgSQL 函数

这又是 C 函数中那个例子在 PL/pgSQL 中的等价物。

```
CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

10.3.3. PL/pgSQL 触发器过程

这个触发器保证，每当表中有行被插入或更新时，当前的用户名和时间都会被盖章到该行中。
而且它还保证给出了员工的名字并且薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary int4,
    last_date datetime,
    last_user name);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
```

```
            RAISE EXCEPTION ''empname cannot be NULL value'';
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION ''% cannot have NULL salary'', NEW.
empname;
        END IF;

        -- Who works for us when she must pay for?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative salary'', NEW.
empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := ''now'';
        NEW.last_user := getpgusername();
        RETURN NEW;
    END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

第 11 章 PL/Tcl - TCL 过程语言

PL/Tcl 是 Postgres 数据库系统的一种可加载过程语言，可以使用 Tcl 语言创建函数和触发器过程。

本包最初由 Jan Wieck 编写。

11.1. 概述

除某些限制外，PL/Tcl 提供了函数编写者在 C 语言中所具备的大部分能力。

一个良性的限制是，所有内容都在一个安全的 Tcl 解释器（Tcl-interpreter）中执行。除了安全 Tcl 受限的命令集之外，只提供了少数通过 SPI 访问数据库以及通过 elog() 发出消息的命令。没有办法访问数据库后端的内部，也无法像在 C 中那样在 Postgres 用户 ID 的权限下获得操作系统级访问。因此，任何非特权数据库用户都可以被允许使用这种语言。

另一个内在的限制是，Tcl 过程不能用来为新数据类型创建输入/输出（input-/output-）函数。

如果在安装过程的配置步骤中指定了 Tcl/Tk 支持，则 PL/Tcl 调用处理器的共享对象代码会自动构建并安装到 Postgres 的库目录中。

11.2. 描述

11.2.1. Postgres 函数与 Tcl 过程名

在 Postgres 中，只要参数个数或参数类型不同，就可以复用同一个函数名。这会与 Tcl 过程名冲突。为了在 PL/Tcl 中提供同样的灵活性，内部 Tcl 过程名中包含该过程的 pg_proc 行的对象 ID 作为其名称的一部分。因此，同一个 Postgres 函数的不同参数类型版本对 Tcl 来说也是不同的。

11.2.2. 在 PL/Tcl 中定义函数

要用 PL/Tcl 语言创建函数，可使用标准语法

```
CREATE FUNCTION funcname argument-types) RETURNS return-type AS '
    # PL/Tcl function body
' LANGUAGE 'pltcl';
```

在查询中调用该函数时，参数以变量 $1 ... $n 的形式提供给 Tcl 过程体。因此，一个小小的 max 函数——返回两个 int4 值中较大的一个——可以这样创建：

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE 'pltcl';
```

复合类型参数会作为 Tcl 数组提供给过程。数组中的元素名就是该复合类型的属性名。如果实际行中的某个属性为 NULL，它就不会出现在数组中！下面是一个在 PL/Tcl 中定义 overpaid_2 函数（见旧版 Postgres 文档）的例子

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
```

```
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

11.2.3. PL/Tcl 中的全局数据

有时（尤其是在使用稍后描述的 SPI 函数时）需要在两次过程调用之间保留某些全局状态数据。同一后端中执行的所有 PL/Tcl 过程共享同一个安全的 Tcl 解释器。为帮助防止 PL/Tcl 过程受到副作用影响，每个过程都可通过 upvar 命令访问一个数组。该变量的全局名称是过程的内部名称，局部名称是 GD。

11.2.4. PL/Tcl 中的触发器过程

在 Postgres 中，触发器过程定义为无参数、返回类型为 opaque 的函数。在 PL/Tcl 语言中也是如此。

来自触发器管理器的信息通过下列变量提供给过程体：

$TG_name

CREATE TRIGGER 语句中触发器的名称。

$TG_relid

导致触发器过程被调用的表的对象 ID。

$TG_relatts

表字段名构成的 Tcl 列表，前面带有一个空列表元素。因此，用 Tcl 的 lsearch 命令在列表中查找元素名时，返回的正编号与 pg_attribute 系统目录中字段的编号一样从 1 开始。

$TG_when

字符串 BEFORE 或 AFTER，取决于触发器调用的事件。

$TG_level

字符串 ROW 或 STATEMENT，取决于触发器调用的事件。

$TG_op

字符串 INSERT、UPDATE 或 DELETE，取决于触发器调用的事件。

$NEW

一个包含新表行值的数组，用于 INSERT/UPDATE 动作；对于 DELETE 则为空。

$OLD

一个包含旧表行值的数组，用于 UPDATE/DELETE 动作；对于 INSERT 则为空。

$GD

上文所述的全局状态数据数组。

$args

一个 Tcl 列表，包含 CREATE TRIGGER 语句中给出的过程参数。这些参数也可以在过程体中以 $1 ... $n 的形式访问。

触发器过程的返回值是字符串 OK 或 SKIP 之一，或者是由 'array get' Tcl 命令返回的列表。如果返回值为 OK，触发该触发器的正常操作（INSERT/UPDATE/DELETE）将会发生。显然，SKIP 告诉触发器管理器静默地取消该操作。'array get' 返回的列表告诉 PL/Tcl 向触发器管理器返回一个修改后的行，用该行代替 $NEW 中给出的行被插入（仅限 INSERT/UPDATE）。不用说，所有这些只在触发器是 BEFORE 且为 FOR EACH ROW 时才有意义。

下面是一个简单的触发器过程示例，它强制用表中的一个整数值记录对该行执行的更新次数（# of updates）。对于新插入的行，该值被初始化为 0，之后每次更新操作都将其递增：

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

CREATE TABLE mytab (num int4, modcnt int4, desc text);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

11.2.5. 从 PL/Tcl 访问数据库

在 PL/Tcl 过程体中可以使用下列命令来访问数据库：

elog *level msg*

发出一条日志消息。可用级别有 NOTICE、ERROR、FATAL、DEBUG 和 NOIND，与 `elog` C 函数类似。

quote *string*

将字符串中所有出现的单引号和反斜杠字符都加倍。在给 `spi_exec` 或 `spi_prepare` 的查询字符串中使用变量时应使用它（`spi_execp` 的值列表不需要）。设想类似下面的查询字符串

```
"SELECT '$val' AS ret"
```

其中 Tcl 变量 val 实际包含 "doesn't"。这将得到最终的查询字符串

```
"SELECT 'doesn't' AS ret"
```

这会在 spi_exec 或 spi_prepare 期间导致解析错误。它应当包含

```
"SELECT 'doesn''t' AS ret"
```

因此必须写成

```
"SELECT '[ quote $val ]' AS ret"
```

spi_exec ?-count *n*? ?-array *name*? *query* ?*loop-body*?

为查询调用解析器/规划器/优化器/执行器。可选的 -count 值告诉 spi_exec 该查询最多处理的行数。

如果查询是 SELECT 语句且给出了可选的 loop-body（一段像 foreach 语句中那样的 Tcl 命令体），则对选出的每一行求值一次，并且对 continue/break 的表现符合预期。所选字段的值被放入以列名命名的变量中。因此

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

会把变量 $cnt 设置为 pg_proc 系统目录中的行数。如果给出了选项 -array，列值将存储在以列名为索引、名为 'name' 的关联数组中，而不是各个单独的变量中。

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

会为 pg_class 的每一行打印一条 DEBUG 日志消息。spi_exec 的返回值是查询所影响的行数，如全局变量 SPI_processed 中所示。

spi_prepare *query typelist*

预备并保存一个查询计划供后续执行。它与 C 层的 SPI_prepare 有一点不同：该计划会被自动复制到顶层内存上下文。因此，目前没有办法预备一个计划而不保存它。

如果查询引用了参数，则必须以 Tcl 列表的形式给出类型名。spi_prepare 的返回值是一个查询 ID，供后续调用 spi_execp 时使用。示例见 spi_execp。

spi_exec ?-count *n*? ?-array*name*? ?-nulls*string*? *queryid* ?*value-list*? ?*loop-body*?

带变量替换地执行一个来自 spi_prepare 的已预备计划。可选的 -count 值告诉 spi_execp 该查询最多处理的行数。

-nulls 的可选值是一个由空格和 'n' 字符组成的字符串，告诉 spi_execp 哪些值是 NULL。如果给出，它的长度必须与值的个数完全相同。

queryid 就是 spi_prepare 调用返回的 ID。

如果调用 spi_prepare 时给出了类型列表，那么在 spi_execp 的 query 之后必须给出一个长度完全相同的 Tcl 值列表。如果 spi_prepare 上的类型列表为空，则必须省略该参数。

如果查询是 SELECT 语句，则对 loop-body 和所选字段的变量，发生的情况与上文为 spi_exec 所述的相同。

下面是一个使用已预备计划的 PL/Tcl 函数示例：

```
CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS '
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \\
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1
 AND num <= \\$2" \\
                int4 ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

注意，在创建函数的查询中，Tcl 应看到的每个反斜杠都必须写成双份，因为主解析器在 CREATE FUNCTION 上也会处理反斜杠。给 spi_prepare 的查询字符串内部应当用美元符号标记参数位置，以免 $1 被第一次函数调用时给出的值替换。

模块与 unknown 命令

PL/Tcl 对常用功能有特殊的支持。它识别两张魔表 pltcl_modules 和 pltcl_modfuncs。如果它们存在，模块 'unknown' 会在解释器创建之后立即被装载。每当调用一个未知的 Tcl 过程时，就会要求 unknown 过程检查该过程是否定义在某个模块中。如果是，则按需装载该模块。要启用这一行为，PL/Tcl 调用处理器编译时必须设置 -DPLTCL_UNKNOWN_SUPPORT。

PL/Tcl 源码的 modules 子目录中有用于维护这些表的支持脚本，其中包括必须最初安装的 unknown 模块的源码。

第 12 章 PL/perl - Perl 过程语言

本章描述如何编译、安装和使用 PL/Perl。

12.1. 概述

PL/Perl 允许你用 Perl 脚本语言编写函数，这些函数可以像 Postgres 内置的函数一样在 SQL 查询中使用。

PL/Perl 解释器是一个完整的 Perl 解释器。但为了提高系统的安全级别，某些操作已被禁用。

一般来说，受限制的是那些与环境交互的操作。这包括文件句柄操作、`require` 和 `use`（用于外部模块）。

应当注意这种安全并非绝对。事实上，几种拒绝服务攻击仍然可能-- 内存耗尽和无限循环是两个例子。

12.2. 构建和安装

假设 Postgres 源代码树的根在 $PGSRC，那么 Pl/perl 的源码位于 $PGSRC/src/pl/plperl。

要构建，只需做以下事情：

```
cd $PGSRC/src/pl/plperl
perl Makefile.PL
make
```

这将创建一个共享库文件。在 Linux 系统上它将被命名为 plperl.so。其他系统上的扩展名可能不同。

然后应当把共享库复制到 postgres 安装的 lib 子目录中。

createlang 命令用于把该语言安装到数据库中。如果安装到 template1，之后所有的数据库都会自动安装该语言。

12.3. 使用 PL/Perl

假设你有下面这个表：

```
CREATE TABLE EMPLOYEE (
    name text,
    basesalary int4,
    bonus int4 );
```

为了得到总报酬（基本工资 + 奖金），我们可以像下面这样定义一个函数：

```
CREATE FUNCTION totalcomp(int4, int4) RETURNS int4
    AS 'return $_[0] + $_[1]'
    LANGUAGE 'plperl';
```

注意，参数如你所料通过 `@_` 传给函数。另外，由于创建函数的 SQL 的引号规则，你可能会发现自己要比习惯上更频繁地使用扩展引号函数（qq[]、q[]、qw[]）。

现在我们可以像这样使用我们的函数：

```
SELECT name, totalcomp(basesalary, bonus) from employee
```

不过，我们也可以把整个元组传给函数：

```
CREATE FUNCTION empcomp(employee) returns int4
    AS 'my $emp = shift;
        return $emp->{''basesalary''} + $emp->{''bonus''};'
    LANGUAGE 'plperl';
```

元组以哈希的引用形式传递。键是元组中字段的名字。值是元组中对应字段的值。

新函数 `empcomp` 可以这样使用：

```
SELECT name, empcomp(employee) from employee;
```

第 13 章 多版本并发控制

多版本并发控制（MVCC）是一种在多用户环境下提升数据库性能的高级技术。Vadim Mikheev[1] 为 Postgres 提供了实现。

13.1. 介绍

与大多数使用锁进行并发控制的其他数据库系统不同，Postgres 通过使用多版本模型来维护数据一致性。这意味着，在查询数据库时，每个事务看到的都是某个较早时刻的数据快照（即一个数据库版本），而不受底层数据当前状态的影响。这样可以保护事务，使其看不到由（其他）并发事务在同一数据行上执行更新所产生的不一致数据，从而为每个数据库会话提供事务隔离。

多版本模型与锁模型的主要区别在于：在 MVCC 中，为查询（读取）数据而获取的锁不会与为写入数据而获取的锁冲突，因此读取永远不会阻塞写入，写入也永远不会阻塞读取。

13.2. 事务隔离

ANSI/ISO 的 SQL 标准根据并发事务之间必须防止的三个现象，定义了四个事务隔离级别。这些不受欢迎的现象是：

脏读

一个事务读取了由并发未提交事务写入的数据。

不可重复读

一个事务重新读取它之前读过的数据，发现这些数据已经被另一个已提交的事务修改。

幻读

一个事务重新执行一个返回满足某个搜索条件的行集合的查询，
发现满足该条件的额外行已经被另一个已提交的事务插入。

四种隔离级别及其对应行为描述如下。

表 13.1. Postgres 隔离级别

	脏读	**不可重复读**	**幻读**
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

Postgres 提供读已提交和可串行化两个隔离级别。

13.3. 读已提交隔离级别

读已提交是 Postgres 中的默认隔离级别。当事务使用这一隔离级别时，查询只能看到查询开始之前已经提交的数据，既看不到脏数据，也看不到查询执行期间并发事务提交的更改。

[1] mailto:vadim@krs.ru

如果查询在执行 UPDATE 语句（或 DELETE 或 SELECT FOR UPDATE）时返回的某行，正在被一个并发未提交事务更新，那么试图更新该行的第二个事务会等待另一个事务提交或回滚。若对方回滚，等待的事务可以继续更改该行。若对方提交（且该行仍然存在，即没有被另一个事务删除），查询会针对该行重新执行，以检查新行版本是否满足查询的搜索条件。如果新行版本满足查询搜索条件，该行将被更新（或删除或标记为更新）。

注意，SELECT 或（带查询的）INSERT 语句的执行结果不会受并发事务影响。

13.4. 可串行化隔离级别

可串行化提供最高的事务隔离。当事务使用可串行化级别时，查询只能看到事务开始之前已经提交的数据，既看不到脏数据，也看不到事务执行期间并发事务提交的更改。因此，该级别模拟串行的事务执行，就好像事务是一个接一个串行执行的，而不是并发执行的。

如果查询在执行 UPDATE（或 DELETE 或 SELECT FOR UPDATE）语句时返回的某行，正在被一个并发未提交事务更新，那么试图更新该行的第二个事务会等待另一个事务提交或回滚。若对方回滚，等待的事务可以继续更改该行。若并发事务提交，可串行化事务将被回滚并收到以下消息：

```
ERROR:  Can't serialize access due to concurrent update
```

因为可串行化事务不能修改在它开始之后被其他事务更改过的行。

> **注意**
>
> 注意，SELECT 或（带查询的）INSERT 的执行结果不会受并发事务影响。

13.5. 锁与表

Postgres 提供多种锁模式来控制对表中数据的并发访问。其中一些锁模式由 Postgres 在语句执行之前自动获取，另一些则供应用使用。事务中获取的所有锁模式（AccessShareLock 除外）在该事务期间一直保持。

除了锁之外，还使用短期的共享/排他闩锁来控制对共享缓冲池中表页的读写访问。闩锁在取到或更新一个元组后立即释放。

13.5.1. 表级锁

AccessShareLock

在被查询的表上自动获取的一种内部锁模式。Postgres 会在语句完成之后释放这些锁。

只与 AccessExclusiveLock 冲突。

RowShareLock

由 SELECT FOR UPDATE 以及 IN ROW SHARE MODE 的 LOCK TABLE 语句获取。

与 ExclusiveLock 和 AccessExclusiveLock 模式冲突。

RowExclusiveLock

由 UPDATE、DELETE、INSERT 以及 IN ROW EXCLUSIVE MODE 的 LOCK TABLE 语句获取。

与 ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ShareLock

由 CREATE INDEX 以及 IN SHARE MODE 的 LOCK TABLE 语句获取。

与 RowExclusiveLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ShareRowExclusiveLock

由 IN SHARE ROW EXCLUSIVE MODE 的 LOCK TABLE 语句获取。

与 RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 Access ExclusiveLock 模式冲突。

ExclusiveLock

由 IN EXCLUSIVE MODE 的 LOCK TABLE 语句获取。

与 RowShareLock、RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

AccessExclusiveLock

由 ALTER TABLE、DROP TABLE、VACUUM 和 LOCK TABLE 语句获取。

与 RowShareLock、RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

注意

只有 AccessExclusiveLock 会阻塞（不带 FOR UPDATE 的）SELECT 语句。

13.5.2. 行级锁

这些锁在行的内部字段被更新（或删除或标记为更新）时获取。Postgres 不会在内存中保存任何关于已修改行的信息，因此在无锁升级的情况下锁定的行数也没有限制。

不过，要注意 SELECT FOR UPDATE 会修改被选中的行以标记它们，因此会产生磁盘写入。

行级锁不影响数据查询。它们只用于阻塞同一行的写者。

13.6. 锁与索引

尽管 Postgres 对表数据提供了非阻塞的读写访问，但对于 Postgres 中实现的每一种索引访问方法，并非都能提供非阻塞的读写访问。

各种索引类型的处理方式如下：

GiST 和 R-树索引

读/写访问使用共享/排他的索引级锁。锁在语句完成之后释放。

Hash 索引

读/写访问使用共享/排他的页级锁。锁在页面处理完成之后释放。

页级锁比索引级锁产生更好的并发性，但可能发生死锁。

Btree

读/写访问使用短期的共享/排他页级闩锁。闩锁在索引元组被插入或取到之后立即释放。

B-树索引在不产生死锁条件的前提下提供最高的并发性。

13.7. 应用级别的数据一致性检查

由于 Postgres 中的读操作不加锁（无论事务隔离级别如何），一个事务读取的数据仍可能被另一个事务覆盖。换句话说，某行被 `SELECT` 返回，并不意味着该行在返回的那一刻（即语句或事务开始之后的某个时刻）确实存在，也不意味着该行在当前事务提交或回滚之前受到保护、不会被并发事务删除或更新。

要确保某一行确实存在，并保护它不受并发更新影响，就必须使用 `SELECT FOR UPDATE` 或适当的 `LOCK TABLE` 语句。把使用可串行化模式的应用从其他环境移植到 Postgres 时，应当考虑这一点。

> **注意**
>
> 在 6.5 版之前，Postgres 使用的是读锁，因此从较早的 Postgres 版本升级到 6.5（或更高）时，上述考虑同样适用。

第 14 章 设置你的环境

本节讨论如何设置你自己的环境，使你能够使用前端应用程序。我们假设 Postgres 已经被成功安装并启动；关于如何安装 Postgres，请参阅管理员指南和安装说明。

Postgres 是一个客户/服务器应用程序。作为用户，你只需要访问安装的客户端部分（客户端应用程序的一个例子是交互式监视程序 psql）。为简单起见，我们假设 Postgres 已经安装在目录 /usr/local/pgsql 中。因此，无论在哪里看到目录 /usr/local/pgsql，你都应该用 Postgres 实际安装所在的目录名来替换它。所有 Postgres 命令都安装在目录 /usr/local/pgsql/bin 中。因此，你应当把这个目录加到你的 shell 命令路径中。如果你使用 Berkeley C shell 的一个变体，例如 csh 或 tcsh，你可以把

```
set path = ( /usr/local/pgsql/bin path )
```

加到你主目录中的 .login 文件里。如果你使用 Bourne shell 的一个变体，例如 sh、ksh 或 bash，那么你可以把

```
$ PATH=/usr/local/pgsql/bin:$PATH
$ export PATH
```

加到你主目录中的 .profile 文件里。从现在起，我们将假设你已经把 Postgres 的 bin 目录加到了你的路径中。此外，我们在本文档中将频繁提到"设置一个 shell 变量"或"设置一个环境变量"。如果你没有完全理解上一段关于修改搜索路径的内容，在继续之前你应当查阅描述你的 shell 的 Unix 手册页。

如果你的站点管理员没有按默认方式完成设置，你可能还有一些工作要做。例如，
如果数据库服务器机器是一台远程机器，你就需要把 PGHOST 环境变量设置为数据库服务器机器的名字。环境变量 PGPORT 也可能必须设置。底线是：如果你试图启动一个应用程序而它抱怨无法连接到 postmaster，你应当立即咨询你的站点管理员，确保你的环境已正确设置。

第 15 章 管理数据库

> **注意**
>
> 这一节目前基本上是教程的一份简单改写的拷贝。需要扩充。- thomas 1998-01-12

虽然站点管理员负责 Postgres 安装的整体管理，但安装中的某些数据库可以由另一个人管理，这个人被指定为数据库管理员。这种职责分配发生在创建数据库时。
一个用户可以被授予创建数据库和/或创建新用户的显式权限。
被同时授予这两种权限的用户可以在 Postgres 内执行大多数管理任务，但默认不会拥有与站点管理员相同的操作系统权限。

数据库管理员指南更详细地讨论这些主题。

15.1. 数据库创建

数据库由从 Postgres 内部发出的 `create database` 命令创建。createdb 是一个命令行实用程序，用来在 Postgres 之外提供同样的功能。

无论哪种方法，要成功执行都必须有 Postgres 后端在运行，而且发出命令的用户必须是 Postgres 超级用户，或者已被超级用户授予了数据库创建权限。

要从命令行创建一个名为 `mydb` 的新数据库，输入

```
% createdb mydb
```

要在 psql 内做同样的事，输入

```
=> CREATE DATABASE mydb;
```

如果你没有创建数据库所需的权限，你会看到下列消息：

```
ERROR:  CREATE DATABASE: Permission denied.
```

Postgres 允许你在给定站点创建任意数量的数据库，并且你会自动成为你刚创建的数据库的数据库管理员。数据库名的第一个字符必须是字母，并且长度限制为 32 个字符。

15.2. 备选数据库位置

可以在安装的默认位置之外的位置创建数据库。请记住，
所有数据库访问实际上都是通过数据库后端进行的，
因此所指定的任何位置都必须是后端可以访问的。

备选数据库位置由一个给出预期存储位置绝对路径的环境变量创建和引用。
这个环境变量必须在后端启动之前已经定义，而且它指向的位置必须可被 postgres 管理员账号写入。关于预先配置的备选数据库位置，请咨询站点管理员。
任何有效的环境变量名都可以用来引用一个备选位置，不过建议使用带 `PGDATA` 前缀的变量名，以避免混淆以及与其他变量的冲突。

> **注意**
>
> 在以前的 Postgres 版本中，也允许使用绝对路径名来指定备选存储位置。虽然环境变量风格的指定方式更可取，因为它让站点管理员在管理磁盘存储时有更大的灵活性，
> 但也可以使用绝对路径来指定一个备选位置。管理员指南讨论了如何启用这一特性。

出于安全和完整性原因，所指定的任何路径或环境变量都会被追加一些额外的路径字段。
备选数据库位置必须通过运行 initlocation 来准备。

要使用环境变量 `PGDATA2`（本例中设为 `/alt/postgres`）创建一个数据存储区，请确保 `/alt/postgres` 已经存在并且可被 Postgres 管理员账号写入。然后，在命令行输入

```
% initlocation PGDATA2
Creating Postgres database system directory /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data/base
```

要从命令行在备选存储区 `PGDATA2` 中创建数据库，使用下面的命令：

```
% createdb -D PGDATA2 mydb
```

要在 psql 内做同样的事，输入

```
=> CREATE DATABASE mydb WITH LOCATION = 'PGDATA2';
```

如果你没有创建数据库所需的权限，你会看到下列消息：

```
ERROR:  CREATE DATABASE: permission denied
```

如果指定的位置不存在，或者数据库后端没有权限访问它或写入其下的目录，
你会看到下列消息：

```
ERROR:  The database path '/no/where' is invalid. This may be due
 to a character that is not allowed or because the chosen path isn't
 permitted for databases.
```

15.3. 访问数据库

一旦构造好数据库，就可以通过以下方式访问它：

- 运行 PostgreSQL 交互式终端 psql，它允许你交互式地输入、编辑和执行 SQL 命令。
- 使用 LIBPQ 子程序库编写 C 程序。这允许你从 C 中提交 SQL 命令，并把答案和状态消息返回给你的程序。这一接口在 PostgreSQL 程序员指南中有进一步讨论。

你也许想启动 psql，来试试本手册中的例子。可以通过输入以下命令为 mydb 数据库激活它：

```
% psql mydb
```

你会看到下面的欢迎消息：

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
```

```
        \g or terminate with semicolon to execute query
        \q to quit

mydb=>
```

这个提示表明 psql 正在监听你，你可以把 SQL 查询输入到由终端监视程序维护的工作区中。psql 程序响应以反斜杠字符 "\" 开头的转义码。例如，你可以通过输入下列内容获得各种 PostgreSQL SQL 命令的语法帮助：

```
mydb=> \h
```

一旦在工作区中输入完查询，你可以通过输入下列内容把工作区的内容传递给 Postgres 服务器：

```
mydb=> \g
```

这告诉服务器处理该查询。如果你用分号终止查询，"\g"就不是必需的。psql 会自动处理以分号终止的查询。要从文件（例如 myFile）中读取查询而不是交互式地输入，输入：

```
mydb=> \i fileName
```

要退出 psql 并返回 Unix，输入

```
mydb=> \q
```

然后 psql 将退出并把你返回到命令 shell。（关于更多转义码，在 psql 提示符下输入 \?。）空白（即空格、制表符和换行符）可以在 SQL 查询中自由使用。单行注释用 "--"表示。连字符之后直到行尾的所有内容都被忽略。多行注释以及行内注释用"/* ... */"表示。

15.3.1. 数据库权限

15.3.2. 表权限

TBD

15.4. 删除数据库

如果你是数据库 mydb 的拥有者，你可以使用下面的 Unix 命令销毁它：

```
% dropdb mydb
```

这个动作会物理地删除与该数据库关联的所有 Unix 文件，而且无法撤销，因此只应在深思熟虑之后才进行。

第 16 章 磁盘存储

本节尚待编写。FAQ 中有一些相关信息。有人愿意吗？- thomas 1998-01-11

第 17 章 理解性能

查询性能会受到许多因素的影响。其中一些可以由用户操纵，
而另一些则是系统底层设计的根本所在。

一些性能问题（例如索引创建和批量数据装载）在别处讨论。本章将讨论EXPLAIN命令，
并展示查询的细节如何影响查询计划，从而影响整体性能。

17.1. 使用EXPLAIN

> **作者**
>
> 由 Tom Lane 撰写，摘自 2000-03-27 的电子邮件。

计划阅读是一门值得写一篇教程的艺术，而我还没来得及写。下面是一些快速而粗糙的解释。

目前 EXPLAIN 给出的数字包括：

- 估计的启动代价（在输出扫描可以开始之前花费的时间，例如在 SORT 节点中执行排序的时间）。
- 估计的总代价（如果检索所有元组——实际可能不会，例如 LIMIT 会在付出总代价之前停止）。
- 该计划节点估计输出的行数。
- 该计划节点估计输出的行的平均宽度（字节）。

代价以磁盘页获取次数为单位度量。（CPU 工作量估计用一些相当任意的主观因子换算成磁盘页单位。如果你想试验这些因子，请参阅 SET参考页。）重要的是要注意，上层节点的代价包括其所有子节点的代价。同样重要的是要认识到，代价只反映规划器/优化器关心的东西。特别地，代价不考虑把结果元组传输到前端所花的时间——这可能是实际耗时的一个相当主要的因素，但规划器忽略它，因为它无法通过改变计划来改变它。
（我们相信每个正确的计划都会输出相同的元组集。）

输出行数有点棘手，因为它不是查询处理/扫描的行数——它通常更少，
反映在该节点上应用的所有 WHERE 子句约束的估计选择率。

平均宽度是相当虚假的，因为系统实际上并不知道变长列的平均长度。
我在考虑将来改进这一点，但可能不值得费这个麻烦，因为宽度的用途不多。

下面是一些例子（使用经过 vacuum analyze 的回归测试数据库，以及接近 7.0 的源代码）：

```
regression=# explain select * from tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

这是最直白的情况。如果你执行

```
select * from pg_class where relname = 'tenk1';
```

你会发现 tenk1 有 233 个磁盘页和 10000 个元组。所以代价估计为 233 次块读取（每次定义为 1.0），加上 10000 * cpu_tuple_cost（当前为 0.01，试试 `show cpu_tuple_cost`）。

现在让我们修改查询，加一个限定子句：

```
regression=# explain select * from tenk1 where unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1000 width=148)
```

由于 WHERE 子句，输出行数的估计下降了。（估计异常准确只是因为 tenk1 是一个特别简单的例子——unique1 列有 10000 个从 0 到 9999 的不同值，所以估计器在列的最小值和最大值之间做的线性插值正好命中。）然而，扫描仍然必须访问全部 10000 行，所以代价没有降低；实际上它还略微上升了，以反映检查 WHERE 条件额外花费的 CPU 时间。

再修改查询，进一步收紧限定：

```
regression=# explain select * from tenk1 where unique1 < 100;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.35 rows=100
 width=148)
```

你会看到，如果我们让 WHERE 条件足够有选择性，规划器最终会判定索引扫描比顺序扫描便宜。由于索引，这个计划只需访问 100 个元组，所以尽管每次单独的获取很昂贵，它仍然胜出。

再给限定加一个条件：

```
regression=# explain select * from tenk1 where unique1 < 100 and
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.60 rows=1
 width=148)
```

添加的子句 "stringu1 = 'xxx'" 降低了输出行数的估计，但没有降低代价，因为我们仍然要访问同一批元组。

让我们试着用一直在讨论的字段连接两个表：

```
regression=# explain select * from tenk1 t1, tenk2 t2 where t1.unique1
 < 100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..144.07 rows=100 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..89.35 rows=100 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..0.53 rows=1 width=148)
```

在这个嵌套循环连接中，外层扫描与倒数第二个例子中的索引扫描相同，
所以其代价和行数也相同，因为我们在该节点上应用的是 "unique1 < 100" WHERE 子句。"t1.unique2 = t2.unique2" 子句此时还不相关，所以它不影响外层扫描的行数。对于内层扫描，当前外层扫描元组的 unique2 值被代入内层索引扫描，产生一个形如 "t2.unique2 = 常量" 的索引限定。所以我们得到的内层扫描计划和代价，与例如 "explain select * from tenk2 where unique2 = 42" 得到的相同。循环节点的代价随后在外层扫描代价的基础上设定，加上每个外层元组一次的内层扫描重复（这里是 100 * 0.53），再加上一点用于连接处理的 CPU 时间。

在这个例子中，循环的输出行数是两个扫描行数的乘积，但一般并非如此，
因为一般你可能会有同时提到两个关系的 WHERE 子句，它们只能在连接点上应用，而不能应用于任一输入扫描。例如，如果我们加上 "WHERE ... AND t1.hundred < t2.hundred"，那会降低连接节点的输出行数，但不改变任一输入扫描。

我们可以通过强制规划器无视它认为获胜的策略来查看不同的计划（一个相当粗糙的工具，
但我们目前只有这个）：

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# explain select * from tenk1 t1, tenk2 t2 where t1.unique1
 < 100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=89.60..574.10 rows=100 width=296)
  ->  Seq Scan on tenk2 t2
              (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=89.35..89.35 rows=100 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..89.35 rows=100 width=148)
```

这个计划提议用同样的老索引扫描提取 tenk1 的 100 个感兴趣的行，把它们存入一个内存散列表，然后对 tenk2 做顺序扫描，在每个 tenk2 元组上探查散列表以寻找 "t1.unique2 = t2.unique2" 的可能匹配。读取 tenk1 并建立散列表的代价完全是散列连接的启动代价，因为在我们能开始读取 tenk2 之前不会得到任何元组。连接的总时间估计还包括探查散列表 10000 次的相当可观的 CPU 时间费用。但注意，我们没有收取 10000 次 89.35；在这种计划类型中散列表的建立只做一次。

第 18 章 填充数据库

作者

由 Tom Lane 撰写，取自一封日期为 1999-12-05 的电子邮件。

在首次填充一个数据库时，可能需要做大量的表插入。
这里有一些让这件事尽可能高效的技巧和技术。

18.1. 禁用自动提交

关闭自动提交，只在最后做一次提交。否则 Postgres会为每一条加入的记录做大量的工作。一般来说，在做批量插入时，你会希望关掉一些数据库特性来换取速度。

18.2. 使用 COPY FROM

使用`COPY FROM STDIN`用一条命令载入所有记录，而不是一系列 INSERT 命令。这会大幅减少解析、规划等开销。如果你这样做，那么就不必再折腾自动提交了。

18.3. 移除索引

如果你要装载的是一个新建的表，最快的办法是先创建表，用 COPY 批量装载，然后再创建该表需要的所有索引。在已有数据上创建索引比在装载每条记录时增量更新索引要快。

如果你是在向一个已有的表追加数据，可以先`DROP INDEX`，装载表，然后重建索引。当然，在索引缺失期间，数据库对其他用户的性能可能受到不利影响。

第 19 章 SQL 命令

这里是 Postgres 所支持的 SQL 命令的参考信息。

ABORT

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ]
```

输入

无。

输出

`ROLLBACK`

成功时返回的消息。

`NOTICE: ROLLBACK: no transaction in progress`

当前没有正在进行的事务时给出。

描述

`ABORT`回卷当前事务并使该事务所做的所有更新被丢弃。
这条命令的行为与SQL92命令`ROLLBACK` 完全相同，其存在仅出于历史原因。

注解

使用`COMMIT`来成功终止一个事务。

用法

中止所有更改：

```
ABORT WORK;
```

兼容性

SQL92

这条命令是一个因历史原因而保留的Postgres 扩展。`ROLLBACK`是等价的SQL92 命令。

ALTER GROUP

ALTER GROUP — 向组中添加用户、从组中移除用户

大纲

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

输入

name

要修改的组的名称。

username

要添加到组中或从组中移除的用户。这些用户名必须已经存在。

输出

`ALTER GROUP`

更改成功时返回的消息。

描述

`ALTER GROUP`用于更改向组中添加用户或从组中移除用户。只有数据库超级用户才能使用这条命令。把一个用户添加到组中并不会创建该用户。类似地，
从组中移除一个用户也不会删除该用户本身。

使用 CREATE GROUP 来创建一个新组，使用 DROP GROUP 来移除一个组。

用法

向组中添加用户：

```
ALTER GROUP staff ADD USER karl, john
```

从组中移除一个用户

```
ALTER GROUP workers DROP USER beth
```

兼容性

SQL92

`ALTER GROUP` 在 SQL92 中没有语句。角色的概念与之相似。

ALTER TABLE

ALTER TABLE — 修改表属性

大纲

```
ALTER TABLE table [ * ]
    ADD [ COLUMN ] column type
ALTER TABLE table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
ALTER TABLE table
    ADD table constraint definition
```

输入

table

要更改的现有表的名称。

column

新列或现有列的名称。

type

新列的类型。

newcolumn

现有列的新名称。

newtable

表的新名称。

table constraint definition

表的新表约束

输出

ALTER

列或表重命名后返回的消息。

ERROR

表或列不可用时返回的消息。

描述

ALTER TABLE 更改一个现有表的定义。ADD COLUMN 形式使用与 CREATE TABLE 相同的语法向表添加一个新列。ALTER COLUMN 形式允许你为列设置或移除默认值。注意，默认值只会应用于新插入的行。RENAME 子句更改表或列的名称，而不更改受影响表中包含的任何数据。因此，该命令执行后，表或列仍保持相同的类型和大小。ADD *table constraint definition* 子句使用与 CREATE TABLE 相同的语法向表增加一个新约束。

要更改表的模式，你必须拥有该表。

注意

关键字COLUMN只是噪声，可以省略。

“*” 跟在表名之后表示该语句应作用于该表以及继承层次中位于其下的所有表；默认情况下，
属性不会被添加到任何子类中，也不会在任何子类中被重命名。
在超类中添加或修改属性时应始终这样做。如果不这样做，对继承层次的查询（例如

```
SELECT NewColumn FROM SuperClass*
```

将无法工作，因为子类会缺少超类中已有的属性。

在当前实现中，新列的默认值和约束子句会被忽略。可以在之后使用 ALTER TABLE 的 SET DEFAULT 形式设置默认值。（你还必须用 UPDATE 把已有的行更新为新默认值。）

在当前实现中，只能向表添加 FOREIGN KEY 约束。要创建或移除唯一约束，请创建唯一索引（见 CREATE INDEX）。要添加检查约束，你需要用 CREATE TABLE 命令的其他参数重建并重新装载该表。

要更改类（class）的模式，你必须拥有它。不允许重命名系统目录模式的任何部分。
PostgreSQL 用户指南中还有关于继承的更多信息。

有关有效参数的进一步说明，请参见CREATE TABLE。

用法

要添加类型为 VARCHAR 的列到表中：

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

要重命名一个现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

要重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

为一个表增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address)
 REFERENCES addresses(address) MATCH FULL
```

兼容性

SQL92

`ADD COLUMN` 形式符合标准，但不支持默认值和约束，如上所述。`ALTER COLUMN` 形式完全符合标准。

SQL92 为 `ALTER TABLE` 语句规定了一些 Postgres 尚不直接支持的附加能力：

```
ALTER TABLE table DROP CONSTRAINT constraint { RESTRICT |
 CASCADE }
```

移除一个表约束（例如检查约束、唯一约束或外键约束）。要移除唯一约束，请删除一个唯一索引，要移除其他种类的约束，你需要用 CREATE TABLE 命令的其他参数重建并重新装载该表。

例如，要删除表 `distributors` 上的任何约束：

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT |
 CASCADE }
```

从表中移除一个列。目前，要移除一个现有列，必须重建并重新装载该表：

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

重命名列和表的子句是 Postgres 对 SQL92 的扩展。

ALTER USER

ALTER USER — 修改用户账户信息

大纲

```
ALTER USER username
    [ WITH PASSWORD 'password' ]
    [ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ VALID UNTIL 'abstime' ]
```

输入

username

要更改其详细信息的用户的名称。

password

此帐号要使用的新密码。

CREATEDB
NOCREATEDB

这些子句定义用户创建数据库的能力。如果指定了 CREATEDB，被定义的用户将被允许创建自己的数据库。使用 NOCREATEDB 将拒绝用户创建数据库的能力。

CREATEUSER
NOCREATEUSER

这些子句决定是否允许用户自己创建新用户。
此选项还会把该用户变为可以越过所有访问限制的超级用户。

abstime

该用户密码将过期失效的日期（以及可选的时间）。

输出

`ALTER USER`

更改成功时返回的消息。

`ERROR: ALTER USER: user "username" does not exist`

如果数据库不认识指定的用户，则返回此错误消息。

描述

`ALTER USER` 用于更改用户 Postgres 账户的属性。只有数据库超级用户才能用此命令更改权限和口令过期。普通用户只能更改自己的口令。

使用 CREATE USER 创建新用户，使用 DROP USER 删除用户。

用法

更改用户口令：

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

更改用户的有效期限（valid until）日期

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

更改用户的有效期限日期，指定他的授权应在 1998 年 5 月 4 日正午、使用比 UTC 早一小时的时区过期

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

授予用户创建其他用户和新数据库的能力。

```
ALTER USER miriam CREATEUSER CREATEDB;
```

兼容性

SQL92

SQL92 中没有 `ALTER USER` 语句。该标准把用户的定义留给实现。

BEGIN

BEGIN — 以链模式开始一个事务

大纲

```
BEGIN [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有效果。

输出

```
BEGIN
```

这表示一个新的事务已经启动。

```
NOTICE: BEGIN: already a transaction in progress
```

这表示已经有一个事务在进行中。当前事务不受影响。

描述

默认情况下，Postgres 以 unchained mode（非链模式，在其他数据库系统中也称为 “autocommit”（自动提交））执行事务。换句话说，
每条用户语句都在它自己的事务中执行，
并且在语句结束时隐式地执行一次提交（如果执行成功，否则执行一次回滚）。`BEGIN` 以链模式开启一个用户事务，即 `BEGIN` 命令之后的所有用户语句都将在单个事务中执行，直到显式的 COMMIT、ROLLBACK 或执行中止。链模式中的语句执行得快得多，因为事务的启动/提交需要可观的 CPU 和磁盘活动。在一个事务中执行多条语句对于在更改多个相关表时确保一致性也是必需的。

Postgres 中的默认事务隔离级别是 READ COMMITTED，其中事务内的查询只看到在查询执行之前提交的更改。因此，如果你需要更严格的事务隔离，就必须使用 `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`，紧跟在 `BEGIN` 之后。在 SERIALIZABLE 模式中，查询只会看到在整个事务开始之前（实际上，是在一个可串行化事务中第一条 DML 语句执行之前）提交的更改。

如果事务被提交，Postgres 将确保要么所有更新都完成，要么一个都不完成。事务具有标准的 ACID（原子性、一致性、隔离性、持久性）属性。

注意

关于在事务中锁定表的进一步信息，请参阅 LOCK。

使用 COMMIT 或 ROLLBACK 终止一个事务。

用法

开始一个用户事务：

```
BEGIN WORK;
```

兼容性

SQL92

BEGIN 是一种 Postgres 语言扩展。没有显式的 BEGIN 命令出现在 SQL92 中；事务的开启总是隐式的，它以 COMMIT 或 ROLLBACK 语句终止。

> **注意**
>
> 许多关系数据库系统出于便利提供自动提交特性。

顺便一提，BEGIN 关键字在嵌入式 SQL 中用于不同的目的。在移植数据库应用程序时，建议注意事务语义。

SQL92 还要求 SERIALIZABLE 是默认的事务隔离级别。

CLOSE

CLOSE — 关闭一个游标

大纲

```
CLOSE cursor
```

输入

cursor

要关闭的打开游标的名称。

输出

CLOSE

游标成功关闭时返回的消息。

NOTICE PerformPortalClose: portal "*cursor*" not found

如果 *cursor* 没有被声明或者已经被关闭，就会给出这条警告。

描述

CLOSE释放与一个打开的游标相关联的资源。游标关闭后，不允许再对其执行任何后续操作。当不再需要游标时，应将其关闭。

当事务被 COMMIT 或 ROLLBACK 终止时，每个打开的游标都会被隐式关闭。

注意

Postgres 没有显式的 OPEN 游标语句；游标在声明时即被视为打开。请使用 DECLARE 语句声明游标。

用法

关闭游标 liahona：

```
CLOSE liahona;
```

兼容性

SQL92

CLOSE 与 SQL92 完全兼容。

CLUSTER

CLUSTER — 向服务器给出存储聚簇建议

大纲

```
CLUSTER indexname ON table
```

输入

indexname

一个索引的名称。

table

一个表的名称。

输出

`CLUSTER`

聚簇已成功完成。

```
ERROR: relation <tablerelation_number> inherits "table"
```

```
ERROR: Relation table does not exist!
```

描述

`CLUSTER`指示 Postgres近似地基于 *indexname*所指定的索引，对 *table*所指定的类进行聚簇。该索引必须已经定义在 *classname*上。

当一个类被聚簇时，它会基于索引信息被物理地重新排序。聚簇是静态的。换句话说，随着类被更新，这些更改不会被聚簇。系统不会尝试让新实例或更新过的元组保持聚簇。如果需要，可以通过再次发出该命令来手动重新聚簇。

注意

表实际上被按索引顺序复制到一个临时表中，然后改回原来的名称。因此，执行聚簇时会丢失所有 GRANT 权限和其他索引。

如果你在表中随机访问单个行，数据在堆表中的实际顺序并不重要。但如果你倾向于比其他数据更常访问某些数据，并且有一个索引把它们聚集在一起，使用`CLUSTER`会带来好处。

`CLUSTER`有帮助的另一个场合是使用索引从一个表中取出多行时。如果你从一个表中请求某个索引值范围，或者一个有多行匹配的单个索引值，`CLUSTER`会有帮助，因为一旦索引识别出第一行匹配所在的堆页，所有其他匹配的行很可能已经在同一个堆页上，这样就节省了磁盘访问并加快了查询。

有两种聚簇数据的方法。第一种是使用 CLUSTER命令，它使用你指定的索引的顺序对原表重新排序。在大表上这可能很慢，因为行是按索引顺序从堆中取出的，如果堆表是无序的，各项就散布在随机页面上，因此每移动一行就要检索一个磁盘页。Postgres有缓存，但大表的大部分放不进缓存。

另一种聚簇数据的方法是使用

```
SELECT columnlist INTO TABLE newtable
     FROM table ORDER BY columnlist
```

它在 ORDER BY 子句中使用 Postgres的排序代码来匹配索引，对于无序数据它要快得多。然后你删除旧表，使用 ALTER TABLE/RENAME把 *temp*改名为旧名称，并重建任何索引。唯一的问题是 OID 不会被保留。此后，CLUSTER应该会很快，因为堆数据大部分已经有序，并且使用的是现有的索引。

用法

基于其 salary 属性聚簇 employees 关系

```
CLUSTER emp_ind ON emp;
```

兼容性

SQL92

SQL92 中没有CLUSTER语句。

COMMENT

COMMENT — 给对象添加注释

大纲

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ]
  object_name |
  COLUMN table_name.column_name|
  AGGREGATE agg_name agg_type|
  FUNCTION func_name (arg1, arg2, ...)|
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

输入

object_name, *table_name*, *column_name*, *agg_name*, *func_name*, *op*, *trigger_name*

要加注释的对象的名称。

text

要添加的注释。

输出

COMMENT

表成功加上注释时返回的消息。

描述

COMMENT 给对象添加一条注释，注释可以用 psql 的 \dd 命令轻松检索。要移除一个注释，使用 NULL。对象被删除时，其注释也会被自动删除。

用法

给表 mytable 加注释：

```
COMMENT ON mytable IS 'This is my table.';
```

更多示例：

```
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON RULE my_rule IS 'Logs UPDATES of employee records';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
```

```
COMMENT ON TABLE my_table IS 'Employee Information';
COMMENT ON TYPE my_type IS 'Complex Number support';
COMMENT ON VIEW my_view IS 'View of departmental costs';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
COMMENT ON AGGREGATE my_aggregate float8 IS 'Computes sample
 variance';
COMMENT ON FUNCTION my_function (datetime) IS 'Returns Roman Numeral';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two
 text';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
```

兼容性

SQL92

SQL92 中没有 COMMENT。

COMMIT

COMMIT — 提交当前事务

大纲

```
COMMIT [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有任何作用。

输出

`COMMIT`

事务成功提交时返回的消息。

`NOTICE: COMMIT: no transaction in progress`

如果当前没有正在进行的事务。

描述

`COMMIT`提交当前事务。该事务所做的所有更改都会对其他会话可见，并且如果发生崩溃，也能保证其持久性。

注意

关键字 WORK 和 TRANSACTION 只是噪音，可以省略。

使用 ROLLBACK 中止一个事务。

用法

要使所有更改永久生效：

```
COMMIT WORK;
```

兼容性

SQL92

SQL92 只规定了 `COMMIT` 和 `COMMIT WORK` 两种形式。其他方面完全兼容。

COPY

COPY — 在文件和表之间复制数据

大纲

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

输入

BINARY

更改字段格式化的行为，强制所有数据以二进制而不是文本格式存储或读取。

table

一个现有表的名称。

WITH OIDS

为每一行复制内部唯一对象 id（OID）。

filename

输入或输出文件的绝对 Unix 路径名。

stdin

指定输入来自管道或终端。

stdout

指定输出到管道或终端。

delimiter

分隔输入或输出字段的字符。

null print

表示 NULL 值的字符串。默认是“\N”（反斜线-N）。例如你可能更喜欢空字符串。

> **注意**
>
> 在 copy in 中，任何匹配此字符串的数据项都会被存储为 NULL 值，所以应确保使用与 copy out 相同的字符串。

输出

`COPY`

复制成功完成。

`ERROR: reason`

复制因错误消息中所述的原因而失败。

描述

`COPY` 在 Postgres 表和标准文件系统文件之间移动数据。`COPY` 指示 Postgres 后端直接读取或写入文件。该文件必须对后端直接可见，且名称必须从后端的角度指定。如果指定了 `stdin` 或 `stdout`，数据通过客户端前端流经后端。

注解

BINARY 关键字强制所有数据以二进制而不是文本格式存储/读取。它比普通的复制命令略快，但通常不可移植，且生成的文件略大，尽管这一因素高度依赖于数据本身。

默认情况下，文本复制使用制表符（"\t"）作为分隔符。也可以用关键字短语 USING DELIMITERS 把分隔符改为任何其他单个字符。数据字段中恰好匹配分隔符的字符将被反斜线引用。

你必须对其值被 `COPY` 读取的任何表拥有 select 访问权限，并对正被 `COPY` 插入值的表拥有 insert 或 update 访问权限。对于被 `COPY` 读取或写入的任何文件，后端还需要适当的 Unix 权限。

关键字短语 USING DELIMITERS 指定一个用于列之间所有分隔符的单个字符。如果在分隔符字符串中指定了多个字符，只使用第一个字符。

> **提示**
>
> 不要把 `COPY` 与 psql 指令 `\copy` 混淆。

`COPY` 既不调用规则也不作用于列默认值。不过它确实会调用触发器。

`COPY` 在第一个错误处停止操作。这对 `COPY FROM` 不会导致问题，但在 `COPY TO` 中目标关系当然会被部分修改。失败的复制之后应当用 `VACUUM` 清理。

由于 Postgres 后端的当前工作目录通常与用户的工作目录不同，复制到文件“`foo`”（不带附加路径信息）可能给天真的用户带来意外的结果。在这种情况下，`foo` 最终会出现在 `$PGDATA/foo` 中。一般而言，指定要复制的文件时应使用后端服务器机器上所见的完整路径名。

用作 `COPY` 参数的文件必须位于数据库服务器机器上或可由其访问，方法是在本地磁盘上或网络文件系统上。

当使用从一台机器到另一台机器的 TCP/IP 连接并指定了目标文件时，目标文件将写入后端运行的机器而不是用户的机器。

文件格式

文本格式

当 `COPY TO` 不带 BINARY 选项使用时，生成的文件中每一行（实例）占一行，各列（属性）由分隔字符分隔。嵌入的分隔字符前面会有一个反斜线字符（"\"）。

属性值本身是由与各属性类型关联的输出函数生成的字符串。
类型的输出函数不应尝试生成反斜线字符；这将由 COPY 自身处理。

每个实例的实际格式是

```
<attr1><separator><attr2><separator>...<separator><attrn><newline>
```

如果指定了 WITH OIDS，oid 放在行的开头。

如果 COPY 把输出发送到标准输出而不是文件，完成时它将在单独一行上发送一个反斜线（"\"）和一个句点（"."）后紧跟一个换行符。类似地，如果 COPY 从标准输入读取，它将期待一个反斜线（"\"）和一个句点（"."）后跟一个换行符，作为一行上表示文件结束的前三个字符。不过，如果在找到这个特殊的文件结束模式之前遇到真正的 EOF，COPY 将终止（随后后端自身也终止）。

反斜线字符还有其他特殊含义。字面反斜线字符表示为两个连续的反斜线（"\\"）。
字面制表符表示为一个反斜线加一个制表符。字面换行符表示为一个反斜线加一个换行符。
在加载不是由 Postgres 生成的文本数据时，你需要把反斜线字符（"\"）转换为双反斜线（"\\"）以确保正确加载。

二进制格式

在 COPY BINARY 的情况下，文件的前四个字节是文件中的实例数。如果此数为零，COPY BINARY 命令将读取直到遇到文件结尾。否则，读满这个数目的实例后将停止读取。文件中的其余数据将被忽略。

文件中每个实例的格式如下。注意必须严格遵循此格式。下表中无符号四字节整数称为 uint32。

表 19.1. 二进制复制文件的内容

在文件的开头	
uint32	元组的数目
对于每个元组	
uint32	元组数据的总长度
uint32	oid（如果指定）
uint32	空属性的数目
[uint32,...,uint32]	属性编号（从 0 计数）
-	<tuple data>

二进制数据的对齐

在 Sun-3 上，2 字节属性按两字节边界对齐，所有更大的属性按四字节边界对齐。字符属性按单字节边界对齐。在大多数其他机器上，所有大于 1 字节的属性都按四字节边界对齐。注意变长属性前面有该属性的长度；数组只是数组元素类型的连续流。

用法

下面的例子把一个表复制到标准输出，使用竖线（"|"）作为字段分隔符：

```
COPY country TO stdout USING DELIMITERS '|';
```

把数据从一个 Unix 文件复制到表 "country":

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

下面是一段适合从 stdin 复制到表中的数据示例（所以最后一行有终止序列）：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

相同的数据在一台 Linux/i586 机器上以二进制格式输出。数据经过 Unix 工具 od -c 过滤后显示。该表有三个字段；第一个是 char(2)，第二个是 text。所有行的第三个字段都是空值。注意 char(2) 字段如何用空字节填充到四字节，text 字段前面有它的长度：

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
006  \0  \0  \0   A   F  \0  \0 017  \0  \0  \0   A   F   G   H
  A   N   I   S   T   A   N 023  \0  \0  \0 001  \0  \0  \0 002
 \0  \0  \0 006  \0  \0  \0   A   L  \0  \0  \v  \0  \0  \0   A
  L   B   A   N   I   A 023  \0  \0  \0 001  \0  \0  \0 002  \0
 \0  \0 006  \0  \0  \0   D   Z  \0  \0  \v  \0  \0  \0   A   L
  G   E   R   I   A
...             \n  \0  \0  \0   Z   A   M   B   I   A 024  \0
 \0  \0 001  \0  \0  \0 002  \0  \0  \0 006  \0  \0  \0   Z   W
 \0  \0  \f  \0  \0  \0   Z   I   M   B   A   B   W   E
```

兼容性

SQL92

SQL92 中没有 COPY 语句。

CREATE AGGREGATE

CREATE AGGREGATE — 定义一个新的聚合函数

大纲

```
CREATE AGGREGATE name ( BASETYPE = input_data_type
    [ , SFUNC1 = sfunc1, STYPE1 = state1_type ]
    [ , SFUNC2 = sfunc2, STYPE2 = state2_type ]
    [ , FINALFUNC = ffunc ]
    [ , INITCOND1 = initial_condition1 ]
    [ , INITCOND2 = initial_condition2 ] )
```

输入

name

要创建的聚合函数的名称。

input_data_type

此聚合函数所操作的输入数据类型。

sfunc1

一个状态转移函数，会为每个非 NULL 输入数据值调用。它必须是一个双参数函数，第一个参数的类型为 *state1_type*，第二个参数的类型为 *input_data_type*。该函数必须返回*state1_type*类型的值。此函数接受当前状态值 1 和当前输入数据项，并返回下一个状态值 1。

state1_type

聚合的第一个状态值的数据类型。

sfunc2

一个状态转移函数，会为每个非 NULL 输入数据值调用。它必须是一个单参数函数，参数类型为 *state2_type*，返回同一类型的值。此函数接受当前状态值 2，并返回下一个状态值 2。

state2_type

聚合的第二个状态值的数据类型。

ffunc

在遍历完所有输入数据之后调用的、用于计算聚合结果的最终函数。
如果使用了两个状态值，最终函数必须接受两个参数，类型分别为 *state1_type* 和 *state2_type*。如果只使用了一个状态值，最终函数必须接受单个该状态值类型的参数。
聚合的输出数据类型被定义为该函数的返回类型。

initial_condition1

状态值 1 的初始值。

initial_condition2

状态值 2 的初始值。

输出

CREATE

命令成功完成时返回的消息。

描述

CREATE AGGREGATE 允许用户或程序员通过定义新的聚合函数来扩展 Postgres 的功能。基本发行版中已经提供了一些针对基本类型的聚合函数，例如min(int4) 和avg(float8)。如果定义了新的类型，或者需要一个尚未提供的聚合函数，就可以使用CREATE AGGREGATE 来提供所需的功能。

聚合函数由其名称和输入数据类型标识。两个聚合如果操作于不同的输入类型，
则可以有相同的名称。为避免混淆，
不要创建与某个聚合同名且具有相同输入数据类型的普通函数。

一个聚合函数由一到三个普通函数构成：两个状态转移函数，*sfunc1* 和 *sfunc2*，以及一个最终计算函数，*ffunc*。它们的用法如下：

```
sfunc1( internal-state1, next-data-item ) ---> next-internal-state1
sfunc2( internal-state2 ) ---> next-internal-state2
ffunc(internal-state1, internal-state2) ---> aggregate-value
```

Postgres 会创建一个或两个临时变量（数据类型为 *stype1* 和/或 *stype2*），用来保存聚合的当前内部状态。对于每个输入数据项，都会调用状态转移函数来为内部状态值计算新值。
等到所有数据都处理完毕后，再调用一次最终函数来计算聚合的输出值。

如果两个转移函数都指定了，那么必须指定 *ffunc*。如果只使用一个转移函数，那么 *ffunc* 是可选的。未提供 *ffunc* 时的默认行为是返回所使用的内部状态值的结束值（因此聚合的输出类型与该状态值的类型相同）。

聚合函数还可以提供一个或两个初始条件，即所使用的内部状态值的初始值。它们在数据库中以text类型的字段的形式指定和存储，但它们必须是状态值数据类型常量的合法外部表示。
如果指定了 *sfunc1* 而没有指定 *initcond1* 值，那么系统不会在第一个输入项上调用 *sfunc1*；而是用第一个输入值初始化内部状态值 1，并从第二个输入项开始调用 *sfunc1*。这对于 MIN 和 MAX 这类聚合很有用。注意，使用这一特性的聚合在没有输入值时被调用会返回 NULL。对状态值 2 没有类似的机制；如果指定了 *sfunc2*，那么就需要一个 *initcond2*。

注意

使用DROP AGGREGATE 删除聚合函数。

CREATE AGGREGATE的参数可以按任意顺序书写，而不只是按上面展示的顺序。

可以指定具有不同状态函数和最终函数组合的聚合函数。例如，count聚合需要 *sfunc2*（一个递增函数）但不需要 *sfunc1* 或 *ffunc*；而 sum聚合需要 *sfunc1*（一个加法函数）但不需要 *sfunc2* 或 *ffunc*；而 avg 聚合则需要两个状态函数以及一个 *ffunc*（一个除法函数）来产生它的结果。无论如何，至少要定义一个状态函数，而且任何 *sfunc2* 都必须有一个对应的 *initcond2*。

用法

完整的使用示例请参阅PostgreSQL 程序员指南中关于聚合函数的章节。

兼容性

SQL92

`CREATE AGGREGATE` 是 Postgres 的语言扩展。SQL92 中没有`CREATE AGGREGATE`。

CREATE CONSTRAINT TRIGGER

CREATE CONSTRAINT TRIGGER — 创建一个支持约束的触发器

大纲

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

输入

name

约束触发器的名称。

events

该触发器应针对其触发的事件类别。

relation

触发关系的表名。

constraint

实际的约束说明。

attributes

约束属性。

func(*args*)

作为触发器处理的一部分调用的函数。

输出

CREATE CONSTRAINT

成功时返回的消息。

描述

CREATE CONSTRAINT TRIGGER 被 CREATE/ALTER TABLE 和 pg_dump 用来为参照完整性创建特殊的触发器。

它不打算用于一般用途。

CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]
```

输入

name

要创建的数据库的名称。

dbpath

在文件系统中存储新数据库的一个备选位置。注意事项见下文。

输出

`CREATE DATABASE`

命令成功完成时返回的消息。

`ERROR: user 'username' is not allowed to create/drop databases`

你必须拥有特殊的 CREATEDB 权限才能创建数据库。见 CREATE USER。

`ERROR: createdb: database "name" already exists`

如果已经存在一个具有所指定 *name* 的数据库，就会出现这个错误。

```
ERROR: Single quotes are not allowed in database names.
ERROR: Single quotes are not allowed in database paths.
```

数据库 *name* 和 *dbpath* 都不能包含单引号。这是为了确保创建数据库目录的 shell 命令能够安全执行而要求的。

`ERROR: The path 'xxx' is invalid.`

所指定的 *dbpath* 的展开（如何展开见下文）失败了。检查你输入的路径，或者确保你所引用的环境变量确实存在。

`ERROR: createdb: May not be called in a transaction block.`

如果有一个显式的事务块正在进行中，你就不能调用 `CREATE DATABASE`。必须先完成该事务。

```
ERROR: Unable to create database directory 'path'.
ERROR: Could not initialize database directory.
```

这些错误很可能与数据目录上的权限不足、磁盘已满或其他文件系统问题有关。
运行数据库服务器的用户必须能访问该位置。

描述

CREATE DATABASE 创建一个新的 Postgres 数据库。创建者成为新数据库的拥有者。

可以指定一个备选位置，例如，把数据库存储在另一块磁盘上。该路径必须已经用 initlocation 命令准备好。

如果路径包含斜杠，其前导部分会被解释为一个环境变量，服务器进程必须知道这个变量。这样数据库管理员就可以控制可以在哪些位置创建数据库。（惯例的选择例如 'PGDATA2'。）如果服务器编译时带 ALLOW_ABSOLUTE_DBPATHS（默认不带），那么以前导斜杠标识的绝对路径名（例如 '/usr/local/pgsql/data'）也是允许的。

注意

CREATE DATABASE 是一种 Postgres 语言扩展。

使用 DROP DATABASE 删除一个数据库。

程序 createdb 是围绕这个命令的 shell 脚本包装，为方便起见提供。

使用以绝对路径名指定的备选数据库位置会涉及安全和数据完整性问题，默认情况下，只有后端已知的环境变量才能被指定为备选位置。更多信息见《管理员指南》。

用法

创建一个新的数据库：

```
olly=>create database lusiadas;
```

要在备选区域 ~/private_db 中创建一个新数据库：

```
$mkdir private_db$initlocation ~/private_dbCreating Postgres database
 system directory /home/olly/private_db/base$psql ollyWelcome to psql,
 the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

olly=>CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/private_
db';CREATE DATABASE
```

兼容性

SQL92

SQL92 中没有 CREATE DATABASE 语句。数据库等价于目录，其创建是实现定义的。

CREATE FUNCTION

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS obj_file , link_symbol
    LANGUAGE 'C'
    [ WITH ( attribute [, ...] ) ]
```

输入

name

要创建的函数的名称。

ftype

函数参数的数据类型。输入类型可以是基本类型或复合类型，或者opaque。opaque 表示函数接受非法类型的参数，例如char *。

rtype

返回数据类型。输出类型可以指定为基本类型、复合类型、setof *type*或opaque。setof 修饰符表示该函数将返回一组结果项，而不是单个项。

attribute

关于函数的一条可选信息，用于优化。目前唯一支持的属性是 iscachable。iscachable 表示该函数在给定相同输入值时总是返回相同结果（也就是说，它不做数据库查找，也不使用不直接出现在其参数列表中的信息）。优化器利用iscachable 来判断预计算该函数的一次调用是否安全。

definition

定义函数的字符串；其含义取决于语言。它可以是内部函数名、对象文件的路径、SQL 查询，或者过程语言中的文本。

obj_file,*link_symbol*

当 C 语言源代码中的函数名与 SQL 函数名不同时，动态链接的 C 语言函数使用这种形式的AS 子句。字符串*obj_file*是包含可动态载入对象的文件名，而*link_symbol* 是对象的链接符号，它与 C 语言源代码中函数的名称相同。

langname

可以是 'C'、'sql'、'internal' 或 '*plname*'，其中 '*plname*' 是一种已创建过程语言的名称。详见 CREATE LANGUAGE。

输出

CREATE

如果命令成功完成，就返回这个结果。

描述

CREATE FUNCTION允许 Postgres用户向一个数据库注册一个函数。随后，这个用户被视为该函数的拥有者。

注意

关于编写外部函数的更多信息，请参考 PostgreSQL Programmer's Guide 中关于通过函数扩展Postgres主题的章节。

使用DROP FUNCTION 删除用户定义的函数。

Postgres允许函数"重载"；也就是说，只要几个不同函数的参数类型不同，
它们就可以使用同一个名称。但对internal和 C 语言函数，必须谨慎使用这一设施。

输入参数和返回值允许使用完整的SQL92类型语法。但是，类型规范的某些细节（例如 numeric类型的精度字段）由底层函数实现负责，CREATE FUNCTION命令会静默地忽略它们（即不识别也不强制执行）。

两个internal 函数如果 C 名相同，会在链接时引发错误。要解决这个问题，可以给它们起不同的 C 名（例如，把参数类型用作 C 名的一部分），然后在CREATE FUNCTION的 AS 子句中指定这些名字。如果 AS 子句留空则CREATE FUNCTION 假定函数的 C 名与 SQL 名相同。

当用 C 语言函数重载 SQL 函数时，给函数的每个 C 语言实例起一个不同的名字，然后在 CREATE FUNCTION语法中使用AS子句的另一种形式，确保重载的 SQL 函数名被解析到正确的动态链接对象。

C 函数不能返回一组值。

用法

创建一个简单的 SQL 函数：

```
CREATE FUNCTION one() RETURNS int4
    AS 'SELECT 1 AS RESULT'
    LANGUAGE 'sql';
SELECT one() AS answer;

     answer
--------
      1
```

这个示例通过调用一个用户创建的共享库中的例程来创建一个 C 函数。这个特定的例程计算一个校验位，如果函数参数中的校验位正确就返回 TRUE。它用于 CHECK 约束中。

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
```

```
    AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';

CREATE TABLE product (
    id        char(8) PRIMARY KEY,
    eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                      REFERENCES brandname(ean_prefix),
    eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

这个示例创建一个在用户定义（user defined）类型 complex 与内部类型 point 之间做类型转换的函数。该函数由一个从 C 源码编译出的动态载入对象实现。为了让Postgres自动找到类型转换函数，SQL 函数必须与返回类型同名，因此重载不可避免。在 SQL 定义中使用AS子句的第二种形式来重载函数名：

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE 'c';
```

该函数的 C 声明（decalaration）是：

```
Point * complex_to_point (Complex *z)
{
 Point *p;

 p = (Point *) palloc(sizeof(Point));
 p->x = z->x;
 p->y = z->y;

 return p;
}
```

兼容性

SQL92

CREATE FUNCTION是 Postgres语言扩展。

SQL/PSM

> **注意**
>
> PSM 表示持久存储模块（Persistent Stored Modules）。它是一种过程语言，最初希望在 1996 年底之前 PSM 能被批准为官方标准。截至 1998 年中，这还没有发生，但希望 PSM 最终能成为一个标准。

SQL/PSM CREATE FUNCTION的语法如下：

```
CREATE FUNCTION name
    ( [ [ IN | OUT | INOUT ] type [, ...] ] )
```

```
RETURNS rtype
LANGUAGE 'langname'
ESPECIFIC routine
SQL-statement
```

CREATE GROUP

CREATE GROUP — 创建一个新组

大纲

```
CREATE GROUP name
    [ WITH
     [ SYSID gid ]
     [ USER  username [, ...] ] ]
```

输入

name

组的名称。

gid

SYSID 子句可以用来选择新组的 Postgres 组 id。不过并不一定要这样做。

如果没有指定，默认将使用已分配的最大组 id 加一（从 1 开始）。

username

要包含在该组中的用户列表。这些用户必须已经存在。

输出

CREATE GROUP

命令成功完成时返回的消息。

描述

CREATE GROUP 将在数据库安装中创建一个新组。关于使用组进行认证的信息请参考管理员指南（adminstrator's guide）。你必须是数据库超级用户才能使用这条命令。

使用 ALTER GROUP 更改组的成员，使用 DROP GROUP 删除组。

用法

创建一个空组：

```
CREATE GROUP staff
```

创建一个带成员的组：

```
CREATE GROUP marketing WITH USER jonathan, david
```

兼容性

SQL92

SQL92 中没有 CREATE GROUP 语句。角色（roles）在概念上与组类似。

CREATE INDEX

CREATE INDEX — 构建一个二级索引

大纲

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( column [ ops_name ] [, ...] )
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( func_name( column [, ... ]) [ ops_name ] )
```

输入

UNIQUE

使系统在创建索引时（如果数据已存在）以及每次添加数据时检查表中的重复值。试图插入或更新会导致重复项的数据将产生一个错误。

index_name

要创建的索引名称。

table

要建立索引的表名。

acc_name

用于索引的访问方法的名称。默认访问方法是 BTREE。Postgres 为索引提供了三种访问方法：

BTREE

Lehman-Yao 高并发 B-树的一个实现。

RTREE

使用 Guttman 的二次分裂算法实现标准 R-tree。

HASH

Litwin 线性散列的一个实现。

column

表中一个列的名称。

ops_name

一个关联的操作符类。详见下文。

func_name

一个返回可索引值的函数。

输出

`CREATE`

索引成功创建时返回的消息。

`ERROR: Cannot create index: 'index_name' already exists.`

无法创建索引时出现此错误。

描述

`CREATE INDEX` 在指定的 *`table`* 上构建一个索引 *`index_name`*。

> **提示**
>
> 索引主要用于提升数据库性能。但使用不当会导致性能下降。

在上面所示的第一种语法中，索引的键字段以列名指定。如果索引访问方法支持多列索引，则可以指定多个字段。

在上面所示的第二种语法中，索引定义在把用户指定的函数 *`func_name`* 应用于单个类的一个或多个属性所得到的结果上。这些函数索引可用于基于操作符的快速数据访问，而这些操作符通常需要对基础数据做某种变换才能应用。

Postgres 为索引提供了 btree、rtree 和 hash 访问方法。btree 访问方法是 Lehman-Yao 高并发 btrees 的一个实现。rtree 访问方法使用 Guttman 的二次分裂算法实现标准 rtrees。hash 访问方法是 Litwin 线性散列的一个实现。我们提到这些算法只是为了说明所有这些访问方法都是完全动态的，不需要周期性地进行优化（例如静态 hash 访问方法就需要）。

使用 DROP INDEX 删除索引。

注解

每当一个被索引的属性出现在使用下列操作符之一的比较中时，PostgreSQL 的查询优化器就会考虑使用 B-tree 索引：<, <=, =, >=, >

每当一个被索引的属性出现在使用下列操作符之一的比较中时，PostgreSQL 的查询优化器就会考虑使用 R-tree 索引：<<, &<, &>, >>, @, ~=, &&

每当一个被索引的属性出现在使用 = 操作符的比较中时，Postgres 的查询优化器就会考虑使用 hash 索引。

目前，只有 btree 访问方法支持多列索引。默认最多可指定 16 个键（构建 Postgres 时可以更改此限制）。

索引的每一列都可以指定一个操作符类。操作符类标识索引用于该列的操作符。例如，四字节整数上的 B-tree 索引会使用 `int4_ops` 类；此操作符类包含四字节整数的比较函数。实际上，字段数据类型的默认操作符类通常已足够。操作符类存在的主要原因是，某些数据类型可能有多种有意义的排序方式。例如，我们可能希望按绝对值或实部对复数数据类型排序。我们可以为该数据类型定义两个操作符类，然后在创建索引时选择合适的类。还有一些具有特殊用途的操作符类：

- 操作符类 box_ops 和 bigbox_ops 都支持在 box 数据类型上建立 R-tree 索引。它们之间的区别是 bigbox_ops 会将 box 坐标缩小，以避免对非常大的浮点坐标做乘法、加法和减法时出现浮点异常。如果矩形所在字段约为 20,000 平方单位或更大，则应使用 bigbox_ops。
- int24_ops 操作符类适用于在 int2 数据上构建索引，并在查询条件中与 int4 数据进行比较。类似地，int42_ops 支持在要与查询中的 int2 数据进行比较的 int4 数据上构建索引。

下面的查询列出所有已定义的操作符类：

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_amop amop,
         pg_opclass opc, pg_operator opr
    WHERE amop.amopid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY acc_name, ops_name, ops_comp
```

用法

要在表 films 的字段 title 上创建一个 btree 索引：

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

兼容性

SQL92

CREATE INDEX 是 Postgres 的语言扩展。

SQL92 中没有 CREATE INDEX 命令。

CREATE LANGUAGE

CREATE LANGUAGE — 为函数定义一种新语言

大纲

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

输入

TRUSTED

TRUSTED指定该语言的调用处理器是安全的；也就是说，
它不会为无特权用户提供任何绕过访问限制的功能。如果注册语言时省略这个关键字，
则只有拥有 Postgres超级用户权限的用户能用这种语言创建新函数（就像'C'语言那样）。

langname

新过程语言的名称。语言名不区分大小写。过程语言不能覆盖 Postgres的内建语言之一。

HANDLER *call_handler*

*call_handler*是一个先前已注册函数的名称，该函数将被调用来执行 PL 过程。

comment

LANCOMPILER参数是将被插入到新的 pg_language项的LANCOMPILER属性中的字符串。
目前，Postgres完全不以任何方式使用这个属性。

输出

CREATE

如果语言成功创建，就返回此消息。

ERROR: PL handler function *funcname*() doesn't exist

如果找不到函数 *funcname*()，就返回此错误。

描述

使用CREATE LANGUAGE，Postgres用户可以向 Postgres注册一种新语言。随后，就可以用这种新语言定义函数和触发器过程。注册新语言的用户必须拥有 Postgres超级用户权限。

编写 PL 处理器

过程语言的调用处理器必须用 'C'之类的编译器语言编写，并作为不带参数、返回 opaque类型的函数注册到 Postgres中，这个类型是为未指定或未定义类型准备的占位类型。这可以防止调用处理器被当作函数从查询中直接调用。（不过，当要执行处理器所提供的语言中的某个 PL 函数时，实际调用中还是可以提供参数的。）

不过，当要执行处理器所提供的语言中的某个 PL 函数或触发器过程时，实际调用中必须提供参数。

- 从触发器管理器调用时，唯一的参数是过程的pg_proc项的对象 ID。来自触发器管理的所有其他信息都可以在全局CurrentTriggerData指针中找到。
- 从函数管理器调用时，参数是过程pg_proc项的对象 ID、给 PL 函数的参数个数、放在FmgrValues结构中的参数，以及一个指向布尔值的指针，函数通过它告诉调用方返回值是否为 SQL NULL 值。

获取pg_proc项并分析被调用过程的参数和返回类型，是调用处理器的责任。过程的 CREATE FUNCTION中的 AS 子句可以在pg_proc表项的prosrc属性中找到。它可以是过程语言自身的源文本（例如 PL/Tcl），可以是一个文件的路径名，也可以是任何能告诉调用处理器具体做什么的东西。

注意

使用CREATE FUNCTION创建函数。

使用DROP LANGUAGE删除过程语言。

参阅表pg_language以获得更多信息：

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

 lanname  | lanispl | lanpltrusted | lanplcallfoid | lancompiler
----------+---------+--------------+---------------+-------------
 internal | f       | f            |             0 | n/a
 C        | f       | f            |             0 | /bin/cc
 sql      | f       | f            |             0 | postgres
```

由于过程语言的调用处理器必须以 'C'语言注册到 Postgres中，它继承了 'C'函数的所有能力和限制。

目前，过程语言的定义一旦创建就不能更改。

用法

这是一个用 'C'编写的 PL 处理器模板：

```
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"        /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"
```

```
Datum
plsample_call_handler(
     Oid       prooid,
     int       pronargs,
     FmgrValues     *proargs,
     bool      *isNull)
{
     Datum          retval;
     TriggerData    *trigdata;

     if (CurrentTriggerData == NULL) {
          /*
           * Called as a function
           */

          retval = ...
     } else {
          /*
           * Called as a trigger procedure
           */
          trigdata = CurrentTriggerData;
          CurrentTriggerData = NULL;

          retval = ...
     }
     *isNull = false;
     return retval;
}
```

只需在省略号处再加上几千行代码，就能完成这个 PL 调用处理器。关于如何把它编译成可装载模块，见CREATE FUNCTION。

下面的命令随后注册这个示例过程语言：

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';
CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCOMPILER 'PL/Sample';
```

兼容性

SQL92

CREATE LANGUAGE是 Postgres扩展。SQL92中没有 CREATE LANGUAGE语句。

CREATE OPERATOR

CREATE OPERATOR — 定义一个新的用户操作符

大纲

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = type1 ] [, RIGHTARG = type2 ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
 )
```

输入

name

要定义的操作符。允许的字符见下文。

func_name

用于实现该操作符的函数。

type1

操作符左参数的类型（如果有的话）。对于左一元操作符，这个选项要省略。

type2

操作符右参数的类型（如果有的话）。对于右一元操作符，这个选项要省略。

com_op

该操作符的交换子。

neg_op

该操作符的求反器。

res_proc

该操作符的限制选择度估计函数。

join_proc

该操作符的连接选择度估算函数。

HASHES

表示该操作符可以支持哈希连接。

left_sort_op

如果此操作符可以支持归并连接，则为对该操作符左侧数据类型排序的操作符。

right_sort_op

如果此操作符可以支持归并连接，则为对该操作符右侧数据类型排序的操作符。

输出

CREATE

操作符创建成功时返回的消息。

描述

CREATE OPERATOR 定义一个新的操作符 *name*。定义操作符的用户将成为其所有者。

操作符的 *name* 是一个最多由 NAMEDATALEN-1（默认 31）个字符组成的序列，这些字符取自下面的列表：

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

对名称的选择有若干限制：

- "$" 和 ":" 不能定义为单字符操作符，尽管它们可以是多字符操作符名称的一部分。
- "--" 和 "/*" 不能出现在操作符名称的任何位置，因为它们会被视为注释的开始。
- 多字符操作符名称不能以 "+" 或 "-" 结尾，除非该名称中还至少包含下列字符之一：

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

  例如，@- 是允许的操作符名称，而 *- 不是。这一限制使 Postgres 能够解析符合 SQL 的查询而无需在记号之间加空格。

> **注意**
>
> 使用非 SQL 标准的操作符名称时，你通常需要用空格分隔相邻的操作符以避免歧义。例如，如果你定义了一个名为 "@" 的左一元操作符，你就不能写 X*@Y；你必须写 X* @Y，以确保 Postgres 把它读成两个操作符名而不是一个。

操作符 "!=" 在输入时会被映射为 "<>"，因此这两个名称始终等效。

LEFTARG 和 RIGHTARG 至少必须定义一个。对于二元操作符，两者都应定义。对于右一元操作符，只应定义 LEFTARG；对于左一元操作符，只应定义 RIGHTARG。

过程 *func_name* 必须此前已用 CREATE FUNCTION 定义，并且必须定义为接受正确数量的参数（一个或两个），类型为所指明的类型。

如果存在交换子操作符，就应当把它指出来，这样 Postgres 就可以在需要时反转操作数的顺序。例如，面积小于操作符 <<< 大概会有一个交换子操作符——面积大于 >>>。于是，查询优化器就可以自由地把：

```
box '((0,0),(1,1))'  >>> MYBOXES.description
```

转换为

```
MYBOXES.description <<< box '((0,0),(1,1))'
```

这使执行代码总能使用后一种表示，并在一定程度上简化了查询优化器。

类似地，如果存在求反器操作符，也应当把它指出来。假设存在一个操作符--面积等于 ===，同时也存在面积不等于 !==。求反器链接使查询优化器可以把

```
NOT MYBOXES.description === box '((0,0),(1,1))'
```

化简为

```
MYBOXES.description !== box '((0,0),(1,1))'
```

如果给出了交换子操作符的名称，Postgres 会在目录中搜索它。如果找到了，而它自己还没有交换子，那么该交换子的条目就会被更新，把新创建的操作符作为其交换子。
这对求反器同样适用。
这样做的目的是允许定义两个互为交换子或互为求反器的操作符：第一个操作符应当不带交换子或求反器（视情况而定）先定义；定义第二个操作符时，把第一个指名为交换子或求反器，
第一个就会作为副作用被更新。（从 Postgres 6.5 起，让两个操作符互相引用也可以。）

HASHES、SORT1 和 SORT2 选项的存在是为了支持查询优化器执行连接。Postgres 总是能通过迭代替换 [WONG76] 求一个连接的值（即处理一个由返回 boolean 的操作符分隔两个元组变量的子句）。此外，Postgres 还可以使用类似 [SHAP86] 的哈希连接算法；但它必须知道这一策略是否适用。当前的哈希连接算法只对表示相等测试的操作符是正确的；而且，
数据类型的相等必须意味着类型表示的按位相等。（例如，
一个包含对相等测试无关紧要的未用位的数据类型就不能做哈希连接。）HASHES 标志向查询优化器指明，此操作符可以安全地用于哈希连接。

类似地，两个排序操作符向查询优化器指明归并排序是否是可用的连接策略，
以及应该用哪些操作符来排序两个操作数类。只应为相等操作符提供排序操作符，
而且它们应分别指向左右两侧数据类型的小于操作符。

如果将来发现其他连接策略切实可行，Postgres 会修改优化器和运行时系统来使用它们，并在定义操作符时要求额外的说明。幸运的是，研究界并不经常发明新的连接策略，
而用户自定义连接策略带来的额外通用性被认为不值得为之增加复杂度。

RESTRICT 和 JOIN 选项帮助查询优化器估算结果大小。如果限定条件中出现了形如：

```
MYBOXES.description <<< box '((0,0),(1,1))'
```

的子句，那么 Postgres 可能必须估算 MYBOXES 中满足该子句的实例比例。函数 *res_proc* 必须是一个已注册的函数（即已用 `CREATE FUNCTION` 定义），它接受正确数据类型的参数并返回一个浮点数。查询优化器只需调用该函数，传入参数 `((0,0),(1,1))`，再把结果乘以关系的大小，就得到预期的实例数。

类似地，当操作符的两个操作数都含有实例变量时，查询优化器必须估算所得连接的大小。函数 join_proc 会返回另一个浮点数，把它乘以所涉及两个类的基数，就得到预期的结果大小。

函数

```
my_procedure_1 (MYBOXES.description, box '((0,0),(1,1))')
```

与操作符

```
MYBOXES.description === box '((0,0),(1,1))'
```

的区别在于：Postgres 会尝试优化操作符，并可在涉及操作符时决定使用索引来限制搜索空间。但系统不会尝试优化函数，函数是靠蛮力执行的。此外，函数可以有任意数量的参数，而操作符只能有一个或两个。

注意

更多信息参见 PostgreSQL 用户指南中关于操作符的章节。参见 `DROP OPERATOR` 以从数据库中删除用户定义的操作符。

用法

下面的命令为 BOX 数据类型定义一个新的操作符--面积相等。

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
);
```

兼容性

SQL92

`CREATE OPERATOR` 是 Postgres 的扩展。在 SQL92 中没有 `CREATE OPERATOR` 语句。

CREATE RULE

CREATE RULE — 定义一条新规则

大纲

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] [ action | NOTHING ]
```

输入

name

要创建的规则的名称。

event

事件是 select、update、delete 或 insert 之一。

object

Object 是 *table* 或 *table*.*column*。

condition

任意的 SQL WHERE 子句，只要 SQL 中允许出现实例变量，new 或 old 就可以出现在实例变量的位置上。

action

任意的 SQL 语句，只要 SQL 中允许出现实例变量，new 或 old 就可以出现在实例变量的位置上。

输出

CREATE

规则成功创建时返回的消息。

描述

Postgres 的规则系统允许定义在对数据库表或类进行插入、更新或删除时所执行的替代动作。目前，规则用于实现表视图。

一条规则的语义是：当单个实例被访问、插入、更新或删除时，会有一个旧实例（对于选择、更新和删除）和一个新实例（对于插入和更新）。如果 ON 子句中指定的 *event* 和 WHERE 子句中指定的 *condition* 对旧实例为真，该规则的 *action* 部分就会被执行。不过首先，旧实例和/或新实例中字段的值会替换到 old.*attribute-name* 和 new.*attribute-name* 上。

规则的 *action* 部分与触发激活它的用户命令使用相同的命令标识符和事务标识符执行。

注解

关于 SQL 规则有一点需要提醒。如果同一个类名或实例变量出现在规则的 *event*、*condition* 和 *action* 部分中，它们都会被视为不同的元组变量。更准确地说，new 和 old 是这些子句之间仅有的共享元组变量。例如，下面的两条规则具有相同的语义：

```
ON UPDATE TO emp.salary WHERE emp.name = "Joe"
    DO
 UPDATE emp SET ... WHERE ...


ON UPDATE TO emp-1.salary WHERE emp-2.name = "Joe"
    DO
 UPDATE emp-3 SET ...  WHERE ...
```

每条规则都可以带可选的 INSTEAD 标记。如果没有该标记，当规则 *condition* 部分中的 *event* 发生时，*action* 会随用户命令一起执行。或者，*action* 部分也可以替代用户命令执行。在后一种情况下，*action* 可以是关键字 NOTHING。

避免循环规则时务必特别注意。例如，尽管下面两条规则定义中的每一条都被 Postgres 接受，但 select 命令会导致 Postgres 报告错误，因为查询循环了太多次：

例 19.1. 循环重写规则组合的例子。

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT TO toyemp;


CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD
 SELECT TO emp;
```

这个从 EMP 中选择的尝试会导致 Postgres 报错，因为这些查询循环了太多次。

```
SELECT * FROM emp;
```

要在一个类上定义规则，你必须拥有该类的规则定义权限。使用 GRANT 和 REVOKE 来更改权限。

SQL 规则中的对象不能是数组引用，也不能带参数。

除 "oid" 字段外，系统属性不能在规则中的任何地方被引用。这尤其意味着实例的函数（例如 foo(emp)，其中 emp 是一个类）不能在规则中的任何地方被调用。

规则系统将规则文本和查询计划存储为文本属性。
这意味着如果规则加上其各种内部表示超过大约一页（8KB）的某个值，
规则的创建可能会失败。

用法

让 Sam 获得与 Joe 相同的薪水调整：

```
CREATE RULE example_1 AS
    ON UPDATE emp.salary WHERE old.name = "Joe"
    DO
 UPDATE emp
 SET salary = new.salary
 WHERE emp.name = "Sam";
```

在 Joe 接受薪水调整时，该事件将变为真，并且 Joe 的旧实例和提议的新实例对执行例程可用。因此，他的新薪水会被代入该规则的 action 部分，随后被执行。这样就把 Joe 的薪水传播给了 Sam。

让 Bill 在访问时获得 Joe 的薪水：

```
CREATE RULE example_2 AS
    ON SELECT TO EMP.salary
    WHERE old.name = "Bill"
    DO INSTEAD
 SELECT emp.salary
 FROM emp
 WHERE emp.name = "Joe";
```

拒绝 Joe 访问鞋部门员工的薪水（`current_user` 返回当前用户的名称）：

```
CREATE RULE example_3 AS
    ON
 SELECT TO emp.salary
    WHERE old.dept = "shoe" AND current_user = "Joe"
    DO INSTEAD NOTHING;
```

创建一个在玩具部门工作的员工的视图。

```
CREATE toyemp(name = char16, salary = int4);

CREATE RULE example_4 AS
    ON SELECT TO toyemp
    DO INSTEAD
     SELECT emp.name, emp.salary
 FROM emp
        WHERE emp.dept = "toy";
```

所有新员工的薪水必须不超过 5,000

```
CREATE RULE example_5 AS
    ON INERT TO emp WHERE new.salary > 5000
    DO
 UPDATE emp SET salary = 5000
 WHERE emp.oid = new.oid;
```

兼容性

SQL92

`CREATE RULE` 语句是 Postgres 的语言扩展。SQL92 中没有 `CREATE RULE` 语句。

CREATE SEQUENCE

CREATE SEQUENCE — 创建一个新的序列号发生器

大纲

```
CREATE SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

输入

seqname

要创建的序列的名称。

increment

INCREMENT *increment* 子句是可选的。正值将生成递增序列，负值生成递减序列。默认值为一（1）。

minvalue

可选子句 MINVALUE *minvalue* 决定序列可以生成的最小值。递增和递减序列的默认值分别是 1 和 -2147483647。

maxvalue

用可选子句 MAXVALUE *maxvalue* 决定序列的最大值。递增和递减序列的默认值分别是 2147483647 和 -1。

start

可选的 START *start* 子句让序列可以从任何地方开始。默认起始值：递增序列为 *minvalue*，递减序列为 *maxvalue*。

cache

CACHE *cache* 选项使序列号可以预分配并存储在内存中以便更快访问。最小值为 1（一次只能生成一个值，即不缓存），这也是默认值。

CYCLE

可选的 CYCLE 关键字可以让序列在递增或递减序列分别达到 *maxvalue* 或 *minvalue* 时继续。如果到达了限制，生成的下一个数将是 *minvalue* 或 *maxvalue*，视情况而定。

输出

CREATE

命令成功时返回的消息。

ERROR: Relation '*seqname*' already exists

指定的序列已存在。

```
ERROR: DefineSequence: MINVALUE (start) can't be >= MAXVALUE (max)
```

指定的起始值超出范围。

```
ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)
```

指定的起始值超出范围。

```
ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)
```

最小值和最大值不一致。

描述

`CREATE SEQUENCE` 将向当前数据库中输入一个新的序列号发生器。这包括创建并初始化一个名为 *seqname* 的新的单行表。该发生器将被发出命令的用户所"拥有"。

序列创建之后，可以用函数 `nextval(`*seqname*`)` 从序列获取一个新的数。函数 `currval('`*seqname*`')` 可用来确定当前会话中对指定序列最后一次调用 `nextval(`*seqname*`)` 所返回的数。函数 `setval('`*seqname*`', `*newvalue*`)` 可用来设置指定序列的当前值。下一次调用 `nextval(`*seqname*`)` 将返回给定的值加上序列增量。

Use a query like

```
SELECT * FROM seqname;
```

to get the parameters of a sequence. As an alternative to fetching the parameters from the original definition as above, you can use

```
SELECT last_value FROM seqname;
```

to obtain the last value allocated by any backend.

使用低级锁来支持对一个发生器的多个同时调用。

> **小心**
>
> 如果对一个将被多个后端并发使用的序列对象使用大于一的 cache 设置，可能得到意外的结果。每个后端会在一次访问序列对象期间分配并缓存若干连续的序列值，并相应地增大序列对象的 last_value。之后，该后端内接下来 cache-1 次 nextval 使用只是返回预分配的值，而不再接触共享对象。因此，当前会话中已分配但未使用的数字将会丢失。此外，虽然可以保证多个后端分配到互不相同的序列值，但综合考虑所有后端时，这些值可能不是按顺序生成的。（例如，cache 设为 10 时，后端 A 可能保留值 1..10 并返回 nextval=1，然后后端 B 可能保留值 11..20 并在后端 A 生成 nextval=2 之前返回 nextval=11。）因此，cache 设为一时，可以安全地假定 nextval 值是按顺序生成的；cache 设为大于一时，只应假定 nextval 的值互不相同，而不是纯粹按顺序生成的。另外，last_value 反映的是任何后端保留的最新值，无论它是否已被 nextval 返回。

注解

使用`DROP SEQUENCE`删除序列。

每个后端用自己的缓存存储已分配的数字。当前会话中已缓存但未使用的数字将会丢失，从而在序列中留下"空洞"。

用法

创建一个名为 `serial` 的递增序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从该序列中选择下一个数

```
SELECT NEXTVAL ('serial');

nextval
-------
    114
```

在 INSERT 中使用该序列：

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

在 COPY FROM 之后设置序列值：

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
    AS 'SELECT max(id) FROM distributors'
    LANGUAGE 'sql';
BEGIN;
    COPY distributors FROM 'input_file';
    SELECT setval('serial', distributors_id_max());
END;
```

兼容性

SQL92

`CREATE SEQUENCE` 是一种 Postgres 语言扩展。SQL92 中没有 `CREATE SEQUENCE` 语句。

CREATE TABLE

CREATE TABLE — Creates a new table

大纲

```
CREATE [ TEMPORARY | TEMP ] TABLE table (
  column type
  [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]
  [column_constraint_clause | PRIMARY KEY } [ ... ] ]
  [, ... ]
  [, PRIMARY KEY ( column [, ...] ) ]
  [, CHECK ( condition ) ]
  [, table_constraint_clause ]
  ) [ INHERITS ( inherited_table [, ...] ) ]
```

输入

TEMPORARY

该表仅为本次会话创建，并在会话退出时自动删除。在临时表存在期间，同名的现有永久表不可见。

table

要创建的新类或表的名称。

column

列的名称。

type

列的类型。这可以包括数组说明符。有关数据类型和数组的更多信息，请参阅 PostgreSQL 用户指南。

DEFAULT *value*

列的默认值。详见 DEFAULT 子句。

column_constraint_clause

可选的列约束子句指定一组完整性约束（测试），要使插入或更新操作成功，新条目或更新后的条目必须满足这些约束。每个约束都必须能求值为布尔表达式。尽管 SQL92 要求 *column_constraint_clause* 只引用该列，Postgres 却允许在单个列约束中引用多个列。详见列约束子句。

table_constraint_clause

可选的表 CONSTRAINT 子句指定一组完整性约束，要使插入或更新操作成功，新条目或更新后的条目必须满足这些约束。每个约束都必须能求值为布尔表达式。
单个约束中可以引用多个列。一个表只能指定一个 PRIMARY KEY 子句；PRIMARY KEY *column*（表约束）与 PRIMARY KEY（列约束）互斥。详见表约束子句。

INHERITS *inherited_table*

可选的 INHERITS 子句指定一组表名，该表自动从中继承所有字段。如果某个被继承的字段名出现多于一次，Postgres 会报错。Postgres 会自动让新表继承其在继承层次中上方表上的函数。

输出

`CREATE`

表创建成功时返回的消息。

`ERROR`

表创建失败时返回的消息。通常还伴随一些描述性文字，例如：`ERROR: Relation 'table' already exists`，如果指定的表在数据库中已存在，就会在运行时出现该错误。

`ERROR: DEFAULT: type mismatched`

如果默认值的数据类型与列定义的数据类型不匹配。

描述

`CREATE TABLE` 将向当前数据库中输入一个新的类或表。该表将由发出命令的用户“拥有”。

每个 *type* 可以是简单类型、复杂类型（集合）或数组类型。每个属性都可以指定为非空，并且都可以有默认值，默认值由 DEFAULT 子句指定。

> **注意**
>
> 从 Postgres 6.0 版起，不再强制同一属性内的数组维数一致。这一点在将来的版本中很可能会改变。

可选的 INHERITS 子句指定一组类名，该类自动从中继承所有字段。如果某个被继承的字段名出现多于一次，Postgres 会报错。Postgres 会自动让新类继承其在继承层次中上方类上的函数。函数的继承按照 Common Lisp Object System（CLOS）的惯例进行。

每个新表或类 *table* 都会自动创建为一种类型。因此，该类的一个或多个实例自动成为一种类型，并可用于 ALTER TABLE 或其他 `CREATE TABLE` 语句。

新表作为堆创建，不含初始数据。一个表最多可以有 1600 列（现实中，由于元组大小必须小于 8192 字节，这个上限还会更低），某些站点还可能把该上限配置得更低。表不能与系统目录表同名。

DEFAULT 子句

`DEFAULT` *value*

输入

value

默认值表达式可能的取值有：

- 字面值
- 用户函数
- niladic 函数（无参函数）

输出

无。

描述

DEFAULT 子句为一个列指定默认数据值（通过 CREATE TABLE 语句中的列定义）。默认值的数据类型必须与列定义的数据类型匹配。

如果一个 INSERT 操作涉及的列没有指定默认值，且没有为它提供显式数据值，那么将把 NULL 值赋给该列。Default *literal* 表示默认值就是指定的常量值。Default *niladic-function* 或 *user-function* 表示默认值是 INSERT 时该指定函数的值。

niladic 函数有两种类型：

niladic USER

CURRENT_USER / USER

见 CURRENT_USER 函数

SESSION_USER

见 CURRENT_USER 函数

SYSTEM_USER

未实现

niladic datetime

CURRENT_DATE

见 CURRENT_DATE 函数

CURRENT_TIME

见 CURRENT_TIME 函数

CURRENT_TIMESTAMP

见 CURRENT_TIMESTAMP 函数

用法

为列 `did` 和 `number` 指定常量作为默认值，并为列 `did` 指定一个字符串字面值：

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
    number   INTEGER DEFAULT 0,
```

```
    total    CASH DEFAULT '$0.0'
);
```

为列 did 指定一个已存在的序列作为默认值，并为列 name 指定一个字面值：

```
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

列 CONSTRAINT 子句

```
[ CONSTRAINT name ] { [
    NULL | NOT NULL ] | UNIQUE | PRIMARY KEY | CHECK constraint |
 REFERENCES
     reftable
     (refcolumn)
     [ MATCH matchtype ]
     [ ON DELETE action ]
     [ ON UPDATE action ]
     [ [ NOT ] DEFERRABLE ]
     [ INITIALLY checktime ] }
    [, ...]
```

输入

name

赋予完整性约束的一个任意名称。如果未指定 *name*，它将由表名和列名生成，这应当能保证 *name* 的唯一性。

NULL

该列允许包含 NULL 值。这是默认情况。

NOT NULL

该列不允许包含 NULL 值。这等价于列约束 CHECK (*column* NOT NULL)。

UNIQUE

该列必须具有唯一值。在 Postgres 中，这是通过在表上隐式创建一个唯一索引来强制实现的。

PRIMARY KEY

该列是一个主键，这蕴含着唯一性由系统强制，
并且其他表可以把该列当作行的唯一标识符来依赖。详见 PRIMARY KEY。

constraint

约束的定义。

描述

可选的约束子句指定约束（测试），要使插入或更新操作成功，
新条目或更新后的条目必须满足这些约束。每个约束都必须能求值为布尔表达式。
单个约束中可以引用多个属性。PRIMARY KEY 作为表约束使用与作为列约束使用互不相容。

约束是一条命名的规则：一个通过对在基表上执行的 INSERT、UPDATE 或 DELETE 操作的结果加以限制，来帮助定义有效值集合的 SQL 对象。

定义完整性约束有两种方式：稍后讲述的表约束，以及这里讲述的列约束。

列约束是作为列定义的一部分定义的完整性约束，一旦创建，在逻辑上就成为表约束。
可用的列约束有：

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

NOT NULL 约束

```
[ CONSTRAINT name ] NOT NULL
```

NOT NULL 约束指定一条规则：一列只能包含非空值。这只是列约束，不允许作为表约束。

输出

status

```
ERROR: ExecAppend: Fail to add null value in not null attribute "
column".
```

如果试图向一个带 NOT NULL 约束的列插入空值，就会在运行时出现该错误。

描述

用法

在表 `distributors` 上定义两个 NOT NULL 列约束，其中一个是命名约束：

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

UNIQUE 约束

```
[ CONSTRAINT name ] UNIQUE
```

输入

CONSTRAINT *name*

赋予约束的一个任意标签。

输出

status

```
ERROR: Cannot insert a duplicate key into a unique index.
```

如果试图向一个列插入重复值，就会在运行时出现该错误。

描述

UNIQUE 约束指定一条规则：表中一个或多个不同列组成的一组只能包含唯一值。

被指定列的列定义不必包含 NOT NULL 约束就可以纳入 UNIQUE 约束。一个不带 NOT NULL 约束的列中有多个空值并不违反 UNIQUE 约束。（这偏离了 SQL92 的定义，但是一个更合乎常理的约定。详见兼容性一节。）

每个 UNIQUE 列约束所命名的列，必须与为该表定义的任何其他 UNIQUE 或 PRIMARY KEY 约束所命名的列集合不同。

> **注意**
>
> Postgres 会自动为每个 UNIQUE 约束创建一个唯一索引，以确保数据完整性。详见 CREATE INDEX。

用法

为表 distributors 定义一个 UNIQUE 列约束。UNIQUE 列约束只能定义在表的一列上：

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40) UNIQUE
);
```

它等价于把以下内容指定为表约束：

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

CHECK 约束

```
[ CONSTRAINT name ] CHECK
    ( condition [, ...] )
```

输入

name

赋予约束的一个任意名称。

condition

任何产生布尔结果的合法条件表达式。

输出

status

ERROR: ExecAppend: rejected due to CHECK constraint "*table_column*".

如果试图向一个受 CHECK 约束约束的列插入非法值，就会在运行时出现该错误。

描述

CHECK 约束对列中允许的值指定一个限制。CHECK 约束也允许作为表约束。

SQL92 的 CHECK 列约束只能定义在表的一列上并只引用该列。Postgres 没有这一限制。

PRIMARY KEY 约束

[CONSTRAINT *name*] PRIMARY KEY

输入

CONSTRAINT *name*

约束的一个任意名称。

输出

ERROR: Cannot insert a duplicate key into a unique index.

如果试图向一个受 PRIMARY KEY 约束约束的列插入重复值，就会在运行时出现该错误。

描述

PRIMARY KEY 列约束指定一个表的某一列只能包含唯一（非重复）的非 NULL 值。被指定列的定义不必包含显式的 NOT NULL 约束就可以纳入 PRIMARY KEY 约束。

一个表只能指定一个 PRIMARY KEY。

注解

Postgres 会自动创建一个唯一索引来确保数据完整性。（见 CREATE INDEX 语句）

PRIMARY KEY 约束所命名的列集合应当与为同一表定义的任何 UNIQUE 约束所命名的列集合不同，否则会导致等价索引的重复和不必要的额外运行时开销。不过，Postgres 并不特别禁止这样做。

REFERENCES 约束

```
[ CONSTRAINT name ] REFERENCES reftable [ ( refcolumn ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

REFERENCES 约束指定一条规则：一个列的值会对照另一个列的值进行检查。REFERENCES 也可以作为 FOREIGN KEY 表约束的一部分来指定。

输入

CONSTRAINT *name*

约束的一个任意名称。

reftable

包含要对照检查的数据的表。

refcolumn

reftable 中要对照检查数据的列。如果未指定，则使用 *reftable* 的 PRIMARY KEY。

MATCH *matchtype*

共有三种匹配类型：MATCH FULL、MATCH PARTIAL，以及未指定时的默认匹配类型。MATCH FULL 不允许多列外键中的某一列为 NULL，除非所有外键列都为 NULL。默认的 MATCH 类型允许某些外键列为 NULL 而外键的其他部分不为 NULL。MATCH PARTIAL 目前不支持。

ON DELETE *action*

被引用表中被引用的行被删除时要执行的动作。有下列动作。

NO ACTION

如果违反外键则产生错误。这是默认值。

RESTRICT

与 NO ACTION 相同。

CASCADE

删除任何引用被删除行的行。

SET NULL

将引用列的值设置为 NULL。

SET DEFAULT

将引用列的值设置为它们的默认值。

ON UPDATE *action*

被引用表中被引用的列被更新为新值时要执行的动作。如果行被更新了，但被引用的列没有变化，则不执行任何动作。有下列动作。

NO ACTION

如果违反外键则产生错误。这是默认值。

RESTRICT

与 NO ACTION 相同。

CASCADE

将引用列的值更新为被引用列的新值。

SET NULL

将引用列的值设置为 NULL。

SET DEFAULT

将引用列的值设置为它们的默认值。

[NOT] DEFERRABLE

这控制约束是否可以推迟到事务结束检查。如果为 DEFERRABLE，SET CONSTRAINTS ALL DEFERRED 将使外键只在事务结束时才被检查。NOT DEFERRABLE 是默认值。

INITIALLY *checktime*

checktime 有两个可能的值，指定检查约束的默认时间。

DEFERRED

只在事务结束时检查约束。

IMMEDIATE

每条语句之后检查约束。这是默认值。

输出

status

`ERROR: name referential integrity violation - key referenced from table not found in reftable`

如果试图向一个列插入的值在被引用表中没有匹配的列，就会在运行时出现该错误。

描述

REFERENCES 列约束指定一个表的某一列只能包含与被引用表的被引用列中的值相匹配的值。

加入该列的值会按照给定的匹配类型，与被引用表和被引用列中的值进行匹配。此外，当被引用列的数据变化时，会对该列的匹配数据执行相应动作。

注解

目前 Postgres 只支持 MATCH FULL 和一个默认匹配类型。此外，被引用列应当是被引用表中某个 UNIQUE 约束的列，不过 Postgres 并不强制这一点。

表 CONSTRAINT 子句

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )
                     REFERENCES reftable
                      (refcolumn [, ...] )
                      [ MATCH matchtype ]
                      [ ON DELETE action ]
                      [ ON UPDATE action ]
                      [ [ NOT ] DEFERRABLE ]
                      [ INITIALLY checktime ]
```

输入

CONSTRAINT *name*

赋予完整性约束的一个任意名称。

column [, ...]

要为其定义唯一索引的列名，对于 PRIMARY KEY 还要加 NOT NULL 约束。

CHECK (*constraint*)

一个将被作为约束求值的布尔表达式。

输出

表约束子句可能的输出与列约束子句相应部分的输出相同。

描述

表约束是定义在一个基表的一列或多列上的完整性约束。“表约束”的四种变体是：

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

UNIQUE 约束

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

输入

CONSTRAINT *name*

赋予完整性约束的一个任意名称。

column

表中某一列的名称。

输出

status

ERROR: Cannot insert a duplicate key into a unique index

如果试图向一个列插入重复值，就会在运行时出现该错误。

描述

UNIQUE 约束指定一条规则：表中一个或多个不同列组成的一组只能包含唯一值。UNIQUE 表约束的行为与列约束相同，只是它还能跨越多列。

更多细节见 UNIQUE 列约束一节。

用法

为表 distributors 定义一个 UNIQUE 表约束：

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

PRIMARY KEY 约束

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

输入

CONSTRAINT *name*

约束的一个任意名称。

column [, ...]

表中一列或多列的名称。

输出

status

ERROR: Cannot insert a duplicate key into a unique index.

如果试图向一个受 PRIMARY KEY 约束约束的列插入重复值，就会在运行时出现该错误。

描述

PRIMARY KEY 约束指定一条规则：表中一个或多个不同列组成的一组只能包含唯一（非重复）的非空值。被指定列的列定义不必包含 NOT NULL 约束就可以纳入 PRIMARY KEY 约束。

PRIMARY KEY 表约束与相应的列约束类似，只是它还能涵盖多列。

更多信息见 PRIMARY KEY 列约束一节。

REFERENCES 约束

```
[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )
    REFERENCES reftable [ ( refcolumn [, ...] ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

REFERENCES 约束指定一条规则：一个列的值会对照另一个列的值进行检查。REFERENCES 也可以作为 FOREIGN KEY 表约束的一部分来指定。

输入

CONSTRAINT *name*

约束的一个任意名称。

column [, ...]

表中一列或多列的名称。

reftable

包含要对照检查的数据的表。

referenced column [, ...]

reftable 中要对照检查数据的一列或多列。如果未指定，则使用 *reftable* 的 PRIMARY KEY。

MATCH *matchtype*

共有三种匹配类型：MATCH FULL、MATCH PARTIAL，以及未指定时的默认匹配类型。MATCH FULL 不允许多列外键中的某一列为 NULL，除非所有外键列都为 NULL。默认的 MATCH 类型允许某些外键列为 NULL 而外键的其他部分不为 NULL。MATCH PARTIAL 目前不支持。

ON DELETE *action*

被引用表中被引用的行被删除时要执行的动作。有下列动作。

NO ACTION

如果违反外键则产生错误。这是默认值。

RESTRICT

与 NO ACTION 相同。

CASCADE

删除任何引用被删除行的行。

SET NULL

将引用列的值设置为 NULL。

SET DEFAULT

将引用列的值设置为它们的默认值。

ON UPDATE *action*

被引用表中被引用的列被更新为新值时要执行的动作。如果行被更新了，但被引用的列没有变化，则不执行任何动作。有下列动作。

NO ACTION

如果违反外键则产生错误。这是默认值。

RESTRICT

不允许更新被引用的行。

CASCADE

将引用列的值更新为被引用列的新值。

SET NULL

将引用列的值设置为 NULL。

SET DEFAULT

将引用列的值设置为它们的默认值。

[NOT] DEFERRABLE

这控制约束是否可以推迟到事务结束检查。如果为 DEFERRABLE，SET CONSTRAINTS ALL DEFERRED 将使外键只在事务结束时才被检查。NOT DEFERRABLE 是默认值。

INITIALLY *checktime*

checktime 有两个可能的值，指定检查约束的默认时间。

IMMEDIATE

每条语句之后检查约束。这是默认值。

DEFERRED

只在事务结束时检查约束。

输出

status

ERROR: *name* referential integrity violation - key referenced from *table* not found in *reftable*

如果试图向一个列插入的值在被引用表中没有匹配的列，就会在运行时出现该错误。

描述

FOREIGN KEY 约束指定一条规则：表中一个或多个不同列组成的一组与被引用表中一组不同的列相关联。

FOREIGN KEY 表约束与相应的列约束类似，只是它还能涵盖多列。

更多信息见 FOREIGN KEY 列约束一节。

用法

创建表 films 和表 distributors：

```
CREATE TABLE films (
     code      CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
     title     CHARACTER VARYING(40) NOT NULL,
     did       DECIMAL(3) NOT NULL,
     date_prod DATE,
     kind      CHAR(10),
     len       INTERVAL HOUR TO MINUTE
);


CREATE TABLE distributors (
     did      DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     name     VARCHAR(40) NOT NULL CHECK (name <> '')
);
```

创建一个带二维数组的表：

```
   CREATE TABLE array (
          vector INT[][]
          );
```

为表 films 定义一个 UNIQUE 表约束。UNIQUE 表约束可以定义在表的一列或多列上：

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个 CHECK 列约束：

```
CREATE TABLE distributors (
    did      DECIMAL(3) CHECK (did > 100),
    name     VARCHAR(40)
);
```

定义一个 CHECK 表约束：

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name > '')
);
```

为表 films 定义一个 PRIMARY KEY 表约束。PRIMARY KEY 表约束可以定义在表的一列或多列上：

```
CREATE TABLE films (
    code      CHAR(05),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

为表 distributors 定义一个 PRIMARY KEY 列约束。PRIMARY KEY 列约束只能定义在表的一列上（下面两个示例是等价的）：

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     CHAR VARYING(40),
    PRIMARY KEY(did)
);
```

```
CREATE TABLE distributors (
    did      DECIMAL(03) PRIMARY KEY,
    name     VARCHAR(40)
);
```

注解

CREATE TABLE/INHERITS 是 Postgres 的语言扩展。

兼容性

SQL92

除本地可见的临时表外，SQL92 还定义了 CREATE GLOBAL TEMPORARY TABLE 语句，以及一个可选的 ON COMMIT 子句：

```
CREATE GLOBAL TEMPORARY TABLE table ( column type [
    DEFAULT value ] [ CONSTRAINT column_constraint ] [, ...] )
    [ CONSTRAINT table_constraint ] [ ON COMMIT { DELETE | PRESERVE }
 ROWS ]
```

对于临时表，CREATE GLOBAL TEMPORARY TABLE 语句命名一个对其他客户端可见的新表，并定义该表的列和约束。

CREATE TEMPORARY TABLE 的可选 ON COMMIT 子句指定每当执行 COMMIT 时是否清空临时表中的行。如果省略 ON COMMIT 子句，则假定使用默认选项 ON COMMIT DELETE ROWS。

创建一个临时表：

```
CREATE TEMPORARY TABLE actors (
    id          DECIMAL(03),
    name        VARCHAR(40),
    CONSTRAINT actor_id CHECK (id < 150)
) ON COMMIT DELETE ROWS;
```

UNIQUE 子句

SQL92 为 UNIQUE 指定了一些额外的能力：

表约束定义：

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

列约束定义：

```
[ CONSTRAINT name ] UNIQUE
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

NULL 子句

NULL“约束”（实际上是一个非约束）是 Postgres 对 SQL92 的扩展，加入它是为了与 NOT NULL 子句对称。由于它是任何列的默认情况，它的出现只是噪音。

```
[ CONSTRAINT name ] NULL
```

NOT NULL 子句

SQL92 为 NOT NULL 指定了一些额外的能力：

```
[ CONSTRAINT name ] NOT NULL
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

CONSTRAINT 子句

SQL92 为约束指定了一些额外的能力，并且还定义了断言和域约束。

> **注意**
>
> Postgres 尚不支持域和断言。

断言是一种特殊类型的完整性约束，与其他约束共享同一名字空间。不过，
断言不一定像约束那样依赖于某个特定的基表，因此 SQL-92 提供了 CREATE ASSERTION 语句作为定义约束的另一种方法：

```
CREATE ASSERTION name CHECK ( condition )
```

域约束由 CREATE DOMAIN 或 ALTER DOMAIN 语句定义：

域约束：

```
[ CONSTRAINT name ] CHECK constraint
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

表约束定义：

```
[ CONSTRAINT name ] { PRIMARY KEY ( column, ... ) | FOREIGN
 KEY constraint | UNIQUE constraint | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

列约束定义：

```
[ CONSTRAINT name ] { NOT NULL | PRIMARY KEY | FOREIGN KEY constraint
 | UNIQUE | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

一个 CONSTRAINT 定义可以按任意顺序包含一个可推迟性属性子句和/或一个初始约束模式子句。

NOT DEFERRABLE

该约束必须在每条语句结束时检查。SET CONSTRAINTS ALL DEFERRED 对这种类型的约束没有影响。

DEFERRABLE

这控制约束是否可以推迟到事务结束检查。如果使用了 SET CONSTRAINTS ALL DEFERRED，或者约束被设置为 INITIALLY DEFERRED，这将使外键只在事务结束时才被检查。

SET CONSTRAINT 只改变当前事务的外键约束模式。

INITIALLY IMMEDIATE

只在事务结束时检查约束。这是默认值

INITIALLY DEFERRED

每条语句之后检查约束。

CHECK 子句

SQL92 为表约束或列约束中的 CHECK 指定了一些额外的能力。

表约束定义：

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

列约束定义：

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

PRIMARY KEY 子句

SQL92 为 PRIMARY KEY 指定了一些额外的能力：

表约束定义：

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

列约束定义：

```
[ CONSTRAINT name ] PRIMARY KEY
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

CREATE TABLE AS

CREATE TABLE AS — 创建一个新表

大纲

```
CREATE TABLE table [ (column [, ...] ) ]
     AS select_clause
```

输入

table

要创建的新表的名称。

column

一列的名称。可以使用逗号分隔的列名列表指定多个列名。

select_clause

一条有效的查询语句。允许语法的描述参阅 SELECT。

输出

可能的输出消息的摘要请参阅CREATE TABLE 和SELECT。

描述

CREATE TABLE AS使你可以根据一个现有表的内容创建一个表。它在功能上等价于 SELECT INTO，但语法或许更直接。

CREATE TRIGGER

CREATE TRIGGER — 创建一个新的触发器

大纲

```
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

Inputs

name

现有触发器的名称。

table

表的名称。

event

INSERT、DELETE 或 UPDATE 之一。

funcname

一个用户提供的函数。

Outputs

CREATE

触发器创建成功时返回此消息。

描述

CREATE TRIGGER 将把一个新触发器登记到当前数据库中。该触发器将与关系 *relname* 关联，并执行指定的函数 *funcname*。

可以指定触发器在操作试图作用于元组之前（BEFORE，即约束检查和 INSERT、UPDATE 或 DELETE 尝试之前）引发，或在操作尝试之后（AFTER，例如约束检查完毕且 INSERT、UPDATE 或 DELETE 已完成之后）引发。如果触发器在事件之前引发，它可以跳过对当前元组的操作，或者修改被插入的元组（仅适用于 INSERT 和 UPDATE 操作）。如果触发器在事件之后引发，那么所有更改（包括最后一次插入、更新或删除）对触发器都是"可见"的。

更多信息参见 PostgreSQL 程序员指南中关于 SPI 和触发器的章节。

注意

CREATE TRIGGER 是一种 Postgres 语言扩展。

只有关系的所有者才能在该关系上创建触发器。

截至当前版本（v7.0），STATEMENT 触发器尚未实现。

关于如何移除触发器，参见 DROP TRIGGER。

用法

在向表 films 追加或更新行之前，检查指定的发行商代码是否存在于 distributors 表中：

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors', 'did')
;
```

在取消一个发行商或更新其代码之前，移除对表 films 的所有引用：

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
 'did');
```

兼容性

SQL92

SQL92 中没有 CREATE TRIGGER。

上面的第二个例子也可以使用外键（FOREIGN KEY）约束来完成，如下所示：

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
    FOREIGN KEY(did) REFERENCES films
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

CREATE TYPE

CREATE TYPE — 定义一种新的基本数据类型

大纲

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT = output_
function
      , INTERNALLENGTH = { internallength | VARIABLE } [ ,
 EXTERNALLENGTH = { externallength | VARIABLE } ]
    [ , DEFAULT = "default" ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , SEND = send_function ] [ , RECEIVE = receive_function ]
    [ , PASSEDBYVALUE ] )
```

输入

typename

要创建的类型的名称。

internallength

一个字面值，指定新类型的内部长度。

externallength

一个字面值，指定新类型的外部长度。

input_function

一个由 CREATE FUNCTION 创建的函数的名称，它把数据从外部形式转换为该类型的内部形式。

output_function

一个由 CREATE FUNCTION 创建的函数的名称，它把数据从内部形式转换为适合显示的形式。

element

正在创建的类型是数组；该参数指定数组元素类型。

delimiter

数组的分隔符字符。

default

为表示"数据不存在"而显示的默认文本。

send_function

一个由 CREATE FUNCTION 创建的函数的名称，它把该类型的数据转换为适合传输到另一台机器的形式。

receive_function

一个由 CREATE FUNCTION 创建的函数的名称，它把该类型的数据从适合从另一台机器传输的形式转换为内部形式。

输出

CREATE

类型创建成功时返回的消息。

描述

CREATE TYPE 允许用户向 Postgres 注册一种新的用户数据类型，供当前数据库使用。定义类型的用户将成为其所有者。*typename* 是新类型的名称，在该数据库已定义的类型中必须唯一。

CREATE TYPE 要求在定义类型之前（用 create function）先注册两个函数。新基本类型的表示由 *input_function* 决定，它把该类型的外部表示转换为为该类型定义的操作符和函数可用的内部表示。自然，*output_function* 执行反向的转换。输入和输出函数都必须声明为接受一个或两个 "opaque" 类型的参数。

新的基本数据类型可以是定长的，这时 *internallength* 是一个正整数；也可以是变长的，这时 Postgres 假定新类型与 Postgres 内置的数据类型 "text" 具有相同的格式。要指明一个类型是变长的，把 *internallength* 设为 VARIABLE。外部表示用 *externallength* 关键字以类似的方式指定。

要指明一个类型是数组并指明它有数组元素，可以用 element 关键字指定数组元素的类型。例如，要定义一个 4 字节整数（"int4"）的数组，指定

```
ELEMENT = int4
```

要指明该类型数组上使用的分隔符，可以把 *delimiter* 设为某个特定的字符。默认分隔符是逗号（","）。

如果用户希望某个特定的位模式表示"数据不存在"，可以选择性地提供一个默认值。用 DEFAULT 关键字指定默认值。

可选参数 *send_function* 和 *receive_function* 用于请求 Postgres 服务的应用程序位于另一台机器上的情形。在这种情况下，Postgres 所在机器为该数据类型使用的格式可能与远程机器使用的格式不同。此时，服务器向客户端发送数据时把数据项转换为标准形式，
而在服务器从客户端接收数据时从标准格式转换为机器特定的格式，是恰当的做法。
如果不指定这两个函数，就假定该类型的内部格式在所有相关机器体系结构上都是可接受的。
例如，单个字符从 Sun-4 传到 DECstation 不需要转换，但许多其他类型需要。

可选标志 PASSEDBYVALUE 表示使用该数据类型的操作符和函数应按值而不是按引用传递参数。注意，内部表示超过 4 字节的类型不能按值传递。

对于新的基本类型，用户可以使用本节描述的相应设施定义操作符、函数和聚合。

数组类型

存在两个广义的内置函数 array_in 和 array_out，用于快速创建变长数组类型。这些函数可用于任何现有 Postgres 类型的数组。

大对象类型

"常规"的 Postgres 类型长度只能达到 8192 字节。如果你需要更大的类型，就必须创建大对象类型。这些类型的接口在 PostgreSQL 程序员指南中有详细讨论。所有大对象类型的长度永远是 VARIABLE。

示例

这条命令创建 box 数据类型，然后在类定义中使用该类型：

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

这条命令创建一个带整数元素的变长数组类型：

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
```

这条命令创建一个大对象类型并在类定义中使用它：

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

注意

类型名不能以下划线字符（"_"）开头，且长度只能是 31 个字符。这是因为 Postgres 会为每个基本类型悄悄创建一个数组类型，其名称由基本类型名前加下划线组成。

参见 `DROP TYPE` 以删除现有类型。

另请参阅 `CREATE FUNCTION`、`CREATE OPERATOR` 以及 PostgreSQL 程序员指南中关于大对象的章节。

兼容性

SQL3

`CREATE TYPE` 是一个 SQL3 语句。

CREATE USER

CREATE USER — 创建一个新数据库用户

大纲

```
CREATE USER username
    [ WITH
     [ SYSID uid ]
     [ PASSWORD 'password' ] ]
    [ CREATEDB    | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

输入

username

用户的名称。

uid

SYSID 子句可以用来选择正被创建的用户的 Postgres 用户 id。这些 ID 完全不必与 UNIX 用户 ID 相匹配，但有些人选择让这些数字保持一致。

如果没有指定，将默认使用已分配的最大用户 ID 加一。

password

设置用户的口令。如果你不打算使用口令认证，可以省略这个选项，
否则该用户将无法连接到做了口令认证的服务器。细节见 pg_hba.conf(5) 或管理员指南。

CREATEDB
NOCREATEDB

这些子句定义用户创建数据库的能力。如果指定了 CREATEDB，将被定义的用户将被允许创建他自己的数据库。使用 NOCREATEDB 将拒绝用户创建数据库的能力。如果这个子句被省略，默认使用 NOCREATEDB。

CREATEUSER
NOCREATEUSER

这些子句决定是否允许用户自己创建新用户。
这个选项还会让该用户成为可以越过所有访问限制的超级用户。
省略这个子句将把该用户的这个属性值设置为 NOCREATEUSER。

groupname

把该用户作为新成员插入其中的组的名称。

abstime

VALID UNTIL 子句设置一个绝对时间，超过该时间后用户的口令不再有效。如果省略这个子句，登录将永久有效。

输出

`CREATE USER`

命令成功完成时返回的消息。

描述

CREATE USER 将向一个 Postgres 实例添加一个新用户。关于管理用户和认证的信息，请参阅管理员指南。你必须是数据库超级用户才能使用这个命令。

使用 ALTER USER 更改用户的口令和权限，使用 DROP USER 删除一个用户。使用 `ALTER GROUP` 把用户添加到其他组中或从其他组中移除该用户。Postgres 附带一个脚本createuser，它具有与这个命令相同的功能（事实上，它调用这个命令），但可以从命令 shell 运行。

用法

创建一个没有口令的用户：

```
CREATE USER jonathan
```

创建一个带口令的用户：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4'
```

创建一个带口令的用户，其账号有效到 2001 年底。注意 2002 年到来一秒之后，该账号不再有效：

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL 'Jan 1 2002'
```

创建一个用户可以创建数据库的账号：

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB
```

兼容性

SQL92

SQL92 中没有 `CREATE USER` 语句。

CREATE VIEW

CREATE VIEW — 构造一个虚拟表

大纲

```
CREATE VIEW view AS SELECT query
```

输入

view

要创建的视图的名称。

query

一条 SQL 查询，它将提供视图的列和行。

关于有效参数的更多信息，参见 SELECT 语句。

输出

`CREATE`

视图创建成功时返回的消息。

`ERROR: Relation 'view' already exists`

如果指定的视图已存在于数据库中，就会出现这个错误。

`NOTICE create: attribute named "column" has an unknown type`

如果你不指定类型，创建出的视图就会带有一个未知类型的列。例如，
下面的命令会产生一条警告：

```
CREATE VIEW vista AS SELECT 'Hello World'
```

而这条命令则不会：

```
CREATE VIEW vista AS SELECT text 'Hello World'
```

描述

`CREATE VIEW` 将定义一个表或类的一个视图。该视图不会被物理物化。具体来说，会自动生成一条查询重写检索规则来支持对视图的检索操作。

注意

目前视图是只读的。

使用`DROP VIEW`语句删除视图。

用法

创建一个由所有喜剧（Comedy）影片组成的视图：

```
CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

 code  |           title           | did | date_prod  |  kind  | len
-------+---------------------------+-----+------------+--------+-----
--
 UA502 | Bananas                   | 105 | 1971-07-13 | Comedy | 01:22
 C_701 | There's a Girl in my Soup | 107 | 1970-06-11 | Comedy | 01:36
(2 rows)
```

兼容性

SQL92

SQL92 为 `CREATE VIEW` 语句规定了一些额外的能力：

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [ AS colname ] [, ...]
    FROM table [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

完整 SQL92 命令的可选子句有：

CHECK OPTION

此选项与可更新视图有关。视图上的所有 INSERT 和 UPDATE 都会被检查，以确保数据满足定义视图的条件。如果不满足，更新将被拒绝。

LOCAL

在该视图上检查完整性。

CASCADE

在该视图以及任何依赖视图上检查完整性。如果既未指定 CASCADE 也未指定 LOCAL，则假定使用 CASCADE。

DECLARE

DECLARE — 定义一个用于表访问的游标

大纲

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

输入

cursorname

用于后续 FETCH 操作的游标的名称。

BINARY

使游标以二进制而不是文本格式获取数据。

INSENSITIVE

SQL92 关键字，表示从游标检索的数据应不受其他进程或游标的更新的影响。由于 Postgres 中游标操作发生在事务内，情况总是如此。这个关键字没有任何效果。

SCROLL

SQL92 关键字，表示每次 FETCH 操作可以检索多行。由于 Postgres 总是允许这样做，这个关键字没有任何效果。

query

一个提供将由该游标管理的行的 SQL 查询。关于有效参数的更多信息请参阅 SELECT 语句。

READ ONLY

SQL92 关键字，表示游标将以只读模式使用。由于这是 Postgres 中唯一可用的游标访问模式，这个关键字没有任何效果。

UPDATE

SQL92 关键字，表示游标将用于更新表。由于 Postgres 目前不支持游标更新，这个关键字会引发一条信息性错误消息。

column

要更新的列。由于 Postgres 目前不支持游标更新，UPDATE 子句会引发一条信息性错误消息。

输出

SELECT

SELECT 成功运行时返回的消息。

```
NOTICE BlankPortalAssignName: portal "cursorname" already exists
```

如果 *cursorname* 已被声明，就会发生这个错误。

```
ERROR: Named portals may only be used in begin/end transaction blocks
```

如果游标不是在事务块内声明的，就会发生这个错误。

描述

DECLARE允许用户创建游标，游标可用于从较大的查询中一次取出少量行。游标可以使用 FETCH 以文本或二进制格式返回数据。

普通游标以文本格式（ASCII 或其他编码方案，取决于 Postgres 后端的构建方式）返回数据。由于数据本来就是以二进制格式存储的，系统必须做一次转换来产生文本格式。此外，
文本格式通常比对应的二进制格式更大。信息以文本形式返回后，
客户端应用可能需要把它转换为二进制格式才能处理。BINARY 游标以本机二进制表示形式返回数据。

例如，如果一个查询从整数列返回值一，用默认游标会得到字符串 '1'，而用二进制游标会得到一个等于 control-A（'^A'）的 4 字节值。

使用 BINARY 游标应当小心。诸如 psql 这样的用户应用不识别二进制游标，并期望数据以文本格式返回。

字符串表示与体系结构无关，而二进制表示在不同的机器体系结构之间可能不同，而且Postgres 不为二进制游标解决字节序或表示问题。因此，
如果你的客户端机器和服务器机器使用不同的表示（例如 "big-endian" 对 "little-endian"），你大概不想让数据以二进制格式返回。不过，二进制游标可能稍微高效一些，
因为服务器到客户端数据传输中的转换开销更小。

> **提示**
>
> 如果你想以 ASCII 显示数据，以 ASCII 取回它会省去客户端的一些工作。

注解

游标只在事务中可用。使用 BEGIN、COMMIT 和 ROLLBACK 来定义事务块。

在 SQL92 中，游标只在嵌入式 SQL（ESQL）应用中可用。Postgres 后端没有实现显式的 OPEN cursor 语句；游标在声明时即视为打开。不过，ecpg（Postgres 的嵌入式 SQL 预处理器）支持 SQL92 的游标约定，包括涉及 DECLARE 和 OPEN 语句的那些。

用法

声明一个游标：

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

兼容性

SQL92

SQL92 只允许在嵌入式 SQL 和模块中使用游标。Postgres 允许交互式地使用游标。SQL92 允许嵌入式或模块游标更新数据库信息。所有 Postgres 游标都是只读的。BINARY 关键字是一种 Postgres 扩展。

DELETE

DELETE — 从一个表中删除行

大纲

```
DELETE FROM table [ WHERE condition ]
```

输入

table

一个已存在表的名称。

condition

这是一个 SQL 选择查询，它返回要删除的行。

关于 WHERE 子句的进一步描述请参考 SELECT 语句。

输出

`DELETE` *count*

成功删除了项时返回的消息。*count* 是删除的行数。

如果 *count* 为 0，表示没有行被删除。

描述

`DELETE`从指定表中删除满足 WHERE 子句的行。

如果省略condition（WHERE 子句），则删除表中的所有行。结果是一个有效但为空的表。

> **提示**
>
> TRUNCATE 是一个 Postgres扩展，它提供了一种从表中删除所有行的更快机制。

要修改一个表，你必须拥有对它的写权限，同时对*condition*中读取其值的任何表拥有读权限。

用法

删除除音乐剧外的所有电影：

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;

 code  |           title            | did | date_prod  |  kind   | len
-------+----------------------------+-----+------------+---------+----
---
```

```
 UA501 | West Side Story           | 105 | 1961-01-03 | Musical | 02:
32
 TC901 | The King and I            | 109 | 1956-08-11 | Musical | 02:
13
 WD101 | Bed Knobs and Broomsticks | 111 |            | Musical | 01:
57
(3 rows)
```

清空表 `films`：

```
DELETE FROM films;
SELECT * FROM films;

 code | title | did | date_prod | kind | len
------+-------+-----+-----------+------+-----
(0 rows)
```

兼容性

SQL92

SQL92允许定位式 DELETE 语句：

```
DELETE FROM table WHERE
    CURRENT OF cursor
```

其中 *cursor* 标识一个已打开的游标。Postgres中的交互式游标是只读的。

DROP AGGREGATE

DROP AGGREGATE — 删除一个聚集函数的定义

大纲

```
DROP AGGREGATE name type
```

输入

name

一个已存在聚集函数的名称。

type

一个已存在聚集函数的类型。（关于数据类型的更多信息请参考 PostgreSQL User's Guide）。

输出

`DROP`

命令成功时返回的消息。

`NOTICE RemoveAggregate: aggregate 'agg' for 'type' does not exist`

如果指定的聚集函数在数据库中不存在，就会出现这条消息。

描述

`DROP AGGREGATE`将删除一个已存在的聚集定义。要执行这条命令，当前用户必须是该聚集函数的拥有者。

注意

使用 CREATE AGGREGATE 创建聚集函数。

用法

删除类型 `int4` 的 `myavg` 聚集函数：

```
DROP AGGREGATE myavg int4;
```

兼容性

SQL92

SQL92中没有 `DROP AGGREGATE`语句；这条语句是 Postgres 语言扩展。

DROP DATABASE

DROP DATABASE — 删除一个现有数据库

大纲

```
DROP DATABASE name
```

输入

name

要删除的现有数据库的名称。

输出

`DROP DATABASE`

命令成功时返回此消息。

`ERROR: user 'username' is not allowed to create/drop databases`

你必须拥有特殊的 CREATEDB 权限才能删除数据库。见 CREATE USER。

`ERROR: dropdb: cannot be executed on the template database`

`template1`数据库不能被删除。删除它对你没有好处。

`ERROR: dropdb: cannot be executed on an open database`

你不能连接在你将要删除的数据库上。请改为连接到`template1`或任何其他数据库，然后再次运行这条命令。

`ERROR: dropdb: database 'name' does not exist`

如果指定的数据库不存在，就会出现此消息。

`ERROR: dropdb: database 'name' is not owned by you`

你必须是数据库的所有者。作为所有者通常也意味着它是你创建的。

`ERROR: dropdb: May not be called in a transaction block.`

你必须先完成进行中的事务，然后才能调用这条命令。

`NOTICE: The database directory 'xxx' could not be removed.`

数据库已被删除（除非还出现了其他错误消息），但存储数据的目录无法删除。你必须手动删除它。

描述

`DROP DATABASE`删除现有数据库的目录项并删除包含数据的目录。它只能由数据库所有者（通常是创建它的用户）执行。

注解

当连接到目标数据库时无法执行这条命令。因此，改用 shell 脚本 dropdb 可能更方便，它是这条命令的一个包装。

关于如何创建数据库的信息，请参阅 CREATE DATABASE。

兼容性

SQL92

`DROP DATABASE`语句是 Postgres的语言扩展；SQL92中没有这样的命令。

DROP FUNCTION

DROP FUNCTION — 移除一个用户定义的 C 函数

大纲

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

name

现有函数的名称。

type

函数参数的类型。

Outputs

DROP

命令成功完成时返回的消息。

NOTICE RemoveFunction: Function "*name*" ("*types*") does not exist

如果指定的函数在当前数据库中不存在，则给出此消息。

描述

DROP FUNCTION 将删除对现有 C 函数的引用。要执行此命令，用户必须是该函数的所有者。必须指定函数的输入参数类型，因为只有具有给定名称和参数类型的函数才会被删除。

注意

关于创建聚合函数的信息，参见 CREATE FUNCTION。

系统不会检查依赖该函数的类型、操作符或访问方法是否已被先行删除。

用法

此命令删除平方根函数：

```
DROP FUNCTION sqrt(int4);
```

兼容性

SQL92

DROP FUNCTION 是一种 Postgres 语言扩展。

SQL/PSM

SQL/PSM 是一个实现函数可扩展性的提议标准。SQL/PSM 的 DROP FUNCTION 语句语法如下：

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

DROP GROUP

DROP GROUP — 移除一个组

大纲

```
DROP GROUP name
```

输入

name

现有组的名称。

输出

`DROP GROUP`

组成功删除时返回的消息。

描述

`DROP GROUP` 从数据库中移除指定的组。组中的用户不会被删除。

使用 CREATE GROUP 添加新组，使用 ALTER GROUP 更改组的成员。

用法

删除一个组：

```
DROP GROUP staff;
```

兼容性

SQL92

SQL92 中没有 `DROP GROUP`。

DROP INDEX

DROP INDEX — 从数据库中移除一个索引

大纲

```
DROP INDEX index_name
```

输入

index_name

要移除的该索引的名称。

输出

`DROP`

索引被成功删除时返回的消息。

`ERROR: index "index_name" nonexistent`

如果 *index_name* 不是数据库中的一个索引，就会出现这条消息。

描述

`DROP INDEX` 从数据库系统中删除一个现有的索引。要执行这个命令，你必须是指定索引的拥有者。

注意

`DROP INDEX` 是一种 Postgres 语言扩展。

关于如何创建索引的信息，请参阅 CREATE INDEX。

用法

这条命令将移除 `title_idx` 索引：

```
DROP INDEX title_idx;
```

兼容性

SQL92

SQL92 定义了访问通用关系数据库的命令。索引是一种实现相关的特性，因此 SQL92 语言中没有索引特定的命令或定义。

DROP LANGUAGE

DROP LANGUAGE — 移除一个用户定义的过程语言

大纲

```
DROP PROCEDURAL LANGUAGE 'name'
```

输入

name

现有过程语言的名称。

输出

DROP

语言成功删除时返回此消息。

ERROR: Language "*name*" doesn't exist

如果数据库中没有名为 *name* 的语言，就会出现这条消息。

描述

DROP PROCEDURAL LANGUAGE 将删除此前注册的名为 *name* 的过程语言的定义。

注意

DROP PROCEDURAL LANGUAGE 语句是一种 Postgres 语言扩展。

关于如何创建过程语言的信息，参见 CREATE LANGUAGE。

系统不会检查是否有使用该语言注册的函数或触发器过程仍然存在。
为了不删除并重建全部函数就能重新启用它们，必须把这些函数在 pg_proc 中的 prolang 属性调整为为该 PL 重建的 pg_language 条目的新对象 ID。

用法

这条命令删除 PL/Sample 语言：

```
DROP PROCEDURAL LANGUAGE 'plsample';
```

兼容性

SQL92

There is no DROP PROCEDURAL LANGUAGE in SQL92.

DROP OPERATOR

DROP OPERATOR — 从数据库中移除一个操作符

大纲

```
DROP OPERATOR id ( type | NONE [,...] )
```

输入

id

现有操作符的标识符。

type

函数参数的类型。

输出

`DROP`

命令成功时返回的消息。

`ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2' does not exist`

如果指定的二元操作符不存在，就会出现这条消息。

`ERROR: RemoveOperator: left unary operator 'oper' taking 'type' does not exist`

如果指定的左一元操作符不存在，就会出现这条消息。

`ERROR: RemoveOperator: right unary operator 'oper' taking 'type' does not exist`

如果指定的右一元操作符不存在，就会出现这条消息。

描述

`DROP OPERATOR` 从数据库中删除一个现有的操作符。要执行此命令，你必须是该操作符的所有者。

左一元或右一元操作符的左类型或右类型可以相应地指定为 `NONE`。

注意

`DROP OPERATOR` 语句是一种 Postgres 语言扩展。

关于如何创建操作符的信息，参见 CREATE OPERATOR。

删除任何依赖被删操作符的访问方法和操作符类是用户的责任。

用法

移除 int4 的幂操作符 a^n:

```
DROP OPERATOR ^ (int4, int4);
```

移除布尔值的左一元取反操作符（b !）：

```
DROP OPERATOR ! (none, bool);
```

移除 int4 的右一元阶乘操作符（! i）：

```
DROP OPERATOR ! (int4, none);
```

兼容性

SQL92

SQL92 中没有 DROP OPERATOR。

DROP RULE

DROP RULE — 从数据库中删除一个已存在的规则

大纲

```
DROP RULE name
```

输入

name

要删除的已存在规则的名称。

输出

`DROP`

成功时返回的消息。

`ERROR: RewriteGetRuleEventRel: rule "name" not found`

如果指定的规则不存在，就会出现这条消息。

描述

`DROP RULE`从指定的 Postgres 规则系统中删除一个规则。Postgres 会立即停止强制执行该规则，并从系统目录中清除其定义。

注意

`DROP RULE`语句是一种 Postgres 语言扩展。

关于如何创建规则的信息，请参考 `CREATE RULE`。

规则一旦被删除，对该规则所写入的历史信息的访问可能会随之消失。

用法

删除重写规则 `newrule`：

```
DROP RULE newrule;
```

兼容性

SQL92

SQL92 中没有 `DROP RULE`。

DROP SEQUENCE

DROP SEQUENCE — 移除一个现有序列

大纲

```
DROP SEQUENCE name [, ...]
```

输入

name

序列的名称。

输出

`DROP`

序列被成功删除时返回的消息。

`NOTICE: Relation "name" does not exist.`

指定的序列不存在时出现这条消息。

描述

`DROP SEQUENCE` 从数据库中移除序列号生成器。在当前把序列实现为特殊表的方式下，它的工作方式与 `DROP TABLE` 语句一样。

注意

`DROP SEQUENCE` 语句是一种 Postgres 语言扩展。

关于如何创建序列的信息，请参阅 `CREATE SEQUENCE` 语句。

用法

要从数据库中移除序列 `serial`：

```
DROP SEQUENCE serial;
```

兼容性

SQL92

SQL92 中没有 `DROP SEQUENCE`。

DROP TABLE

DROP TABLE — 从数据库中删除已存在的表

大纲

```
DROP TABLE name [, ...]
```

输入

name

要删除的已存在表或视图的名称。

输出

`DROP`

命令成功完成时返回的消息。

`ERROR Relation "name" Does Not Exist!`

如果指定的表或视图在数据库中不存在。

描述

`DROP TABLE`从数据库中删除表和视图。只有其拥有者才能删除一个表或视图。一个表可以通过使用 `DELETE`清空行，而不被删除。

如果被删除的表上有二级索引，会先删除它们。只删除一个二级索引不会影响底层表的内容。

注意

关于如何创建或修改表的信息，请参考 `CREATE TABLE`和 `ALTER TABLE`。

用法

删除两个表 `films` 和 `distributors`：

```
DROP TABLE films, distributors;
```

兼容性

SQL92

SQL92 为 DROP TABLE 规定了一些额外的能力：

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

保证只有没有任何依赖视图或完整性约束的表才能被删除。

CASCADE

所有引用它的视图或完整性约束也将被删除。

提示

目前，要删除一个被引用的视图，必须显式删除它。

DROP TRIGGER

DROP TRIGGER — 移除一个触发器的定义

大纲

```
DROP TRIGGER name ON table
```

输入

name

一个现有触发器的名称。

table

一个表的名称。

输出

`DROP`

触发器被成功删除时返回的消息。

`ERROR: DropTrigger: there is no trigger name on relation "table"`

指定的触发器不存在时出现这条消息。

描述

`DROP TRIGGER` 将移除对一个现有的触发器定义的所有引用。要执行这个命令当前用户必须是该触发器的拥有者。

注意

`DROP TRIGGER` 是一种 Postgres 语言扩展。

关于如何创建触发器的信息，请参阅 `CREATE TRIGGER`。

用法

销毁表 `films` 上的 `if_dist_exists` 触发器：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

SQL92

SQL92 中没有 `DROP TRIGGER` 语句。

DROP TYPE

DROP TYPE — 从系统目录中移除一个用户定义的类型

大纲

```
DROP TYPE typename
```

输入

typename

现有类型的名称。

输出

`DROP`

命令成功时返回的消息。

`ERROR: RemoveType: type 'typename' does not exist`

如果找不到指定的类型，就会出现这条消息。

描述

`DROP TYPE` 将从系统目录中删除一个用户类型。

只有类型的所有者才能删除它。

注意

DROP TYPE 语句是一种 Postgres 语言扩展。

关于如何创建类型的信息，参见 `CREATE TYPE`。

删除任何使用被删类型的操作符、函数、聚合、访问方法、子类型和类是用户的责任。

如果删除了内置类型，后端的行为将不可预料。

用法

要删除 `box` 类型：

```
DROP TYPE box;
```

兼容性

SQL3

`DROP TYPE` 是一个 SQL3 语句。

DROP USER

DROP USER — 删除一个用户

大纲

```
DROP USER name
```

输入

name

一个已存在用户的名称。

输出

`DROP USER`

成功删除用户时返回的消息。

`ERROR: DROP USER: user "name" does not exist`

用户名未找到时出现此消息。

`DROP USER: user "name" owns database "name", cannot be removed`

你必须先删除该数据库或更改其所有权。

描述

`DROP USER` 从数据库中删除指定的用户。它不会删除该用户拥有的表、视图或其他对象。如果该用户拥有任何数据库，则会得到一个错误。

使用 CREATE USER 来添加新用户，使用 ALTER USER 来更改用户的属性。Postgres 附带一个脚本 dropuser，它与这条命令功能相同（实际上它调用的就是这条命令），但可以在命令 shell 中运行。

用法

删除一个用户账户：

```
DROP USER jonathan;
```

兼容性

SQL92

SQL92 中没有 `DROP USER`。

DROP VIEW

DROP VIEW — 从数据库中删除一个现有视图

大纲

```
DROP VIEW name
```

输入

name

现有视图的名称。

输出

`DROP`

命令成功时返回的消息。

`ERROR: RewriteGetRuleEventRel: rule "_RET`*name*`" not found`

如果指定的视图在数据库中不存在，就会出现此消息。

描述

`DROP VIEW`从数据库中删除一个现有视图。要执行这条命令，你必须是该视图的所有者。

注解

Postgres的`DROP TABLE`语句也会删除视图。

关于如何创建视图的信息，请参阅`CREATE VIEW`命令。

用法

这条命令将删除名为`kinds`的视图：

```
DROP VIEW kinds;
```

兼容性

SQL92

SQL92为`DROP VIEW`规定了一些额外的能力：

```
DROP VIEW view { RESTRICT | CASCADE }
```

输入

RESTRICT

确保只有没有依赖视图或完整性约束的视图才能被销毁。

CASCADE

所有引用它的视图和完整性约束也将被删除。

注解

目前，要从Postgres数据库中删除一个被引用的视图，你必须显式地删除它。

END

END — 提交当前的事务

大纲

```
END [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有效果。

输出

`COMMIT`

事务被成功提交时返回的消息。

```
NOTICE: COMMIT: no transaction in progress
```

如果没有正在进行的事务。

描述

`END` 是一种 Postgres 扩展，是 SQL92 兼容的 COMMIT 的同义词。

注意

关键字 WORK 和 TRANSACTION 是噪声词，可以省略。

使用 ROLLBACK 中止一个事务。

用法

要使所有更改永久化：

```
END WORK;
```

兼容性

SQL92

`END` 是一种提供与 COMMIT 等价功能的 PostgreSQL 扩展。

EXPLAIN

EXPLAIN — 显示语句的执行计划

大纲

```
EXPLAIN [ VERBOSE ] query
```

输入

VERBOSE

显示详细查询计划的标志。

query

任何*query*。

输出

```
NOTICE: QUERY PLAN: plan
```

来自Postgres后端的显式查询计划。

```
EXPLAIN
```

查询计划显示完毕后发出的标志。

描述

这条命令显示 Postgres 规划器为给定查询生成的执行计划。执行计划显示查询引用的表将被如何扫描 —— 用普通的顺序扫描、索引扫描等 —— 以及当引用多个表时，将使用什么连接算法把每个输入表中所需的元组汇集到一起。

显示中最关键的部分是估计的查询执行代价，
即规划器对运行该查询所需时间的猜测（以磁盘页获取次数为单位度量）。
实际上会显示两个数字：返回第一个元组之前的启动时间，以及返回所有元组的总时间。
对大多数查询来说，总时间才是重要的，但在诸如 EXISTS 子查询这样的上下文中，规划器会选择最小的启动时间而不是最小的总时间（因为执行器在取到一个元组后无论如何都会停止）。
另外，如果你用 LIMIT 子句限制返回的元组数，规划器会在端点代价之间做适当的插值，来估计哪个计划实际上最便宜。

VERBOSE 选项输出计划树的完整内部表示，而不仅仅是一个摘要（并且也会发送到 postmaster 日志文件）。通常这个选项只对调试 Postgres 有用。

注解

关于优化器在Postgres中对代价信息的使用，目前只有很少的文档。
关于查询优化的代价估计的一般信息可以在数据库教科书中找到。更多信息请参阅
程序员指南中关于索引和遗传查询优化器的章节。

用法

要显示对一个只有单个`int4`列和 128 行的表的简单查询的查询计划：

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

对于同一个表，如果有一个支持查询上等值连接条件的索引，EXPLAIN将显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

最后，对于同一个表，如果有支持查询上等值连接条件的索引，EXPLAIN
对使用聚合函数的查询将显示如下内容：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

注意，这里显示的具体数字、甚至被选中的查询策略都可能随 Postgres 版本因规划器的改进而变化。

兼容性

SQL92

SQL92 中没有定义EXPLAIN语句。

FETCH

FETCH — 使用游标获取行

大纲

```
FETCH [ direction ] [ count ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ] {
 IN | FROM } cursor
```

输入

direction

selector 定义提取方向。它可以是下列之一：

FORWARD

提取下一个（或下一些）行。如果省略 *selector*，这是默认值。

BACKWARD

提取前一个（或前一些）行。

RELATIVE

为兼容 SQL92 而存在的噪声词。

count

count 决定提取多少行。它可以是下列之一：

#

一个有符号整数，指定要提取多少行。注意，负整数等价于反转 FORWARD 和 BACKWARD 的方向。

ALL

检索所有剩余的行。

NEXT

等价于指定数量 `1`。

PRIOR

等价于指定数量 `-1`。

cursor

一个已打开游标的名称。

输出

`FETCH`返回由指定游标定义的查询的结果。如果查询失败，将返回下列消息：

```
NOTICE: PerformPortalFetch: portal "cursor" not found
```

如果*cursor*此前未被声明。游标必须在事务块内声明。

```
NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE
```

Postgres不支持游标的绝对定位。

```
ERROR: FETCH/RELATIVE at current position is not supported
```

SQL92允许使用语法

```
FETCH RELATIVE 0 FROM cursor
```

反复检索游标"当前位置"上的行。

Postgres目前不支持这一概念；实际上，值零被保留用来表示要检索所有行，等价于指定 ALL 关键字。如果使用了 RELATIVE 关键字，Postgres 假定用户想要的是 SQL92行为，并返回这条错误消息。

描述

FETCH允许用户使用游标检索行。检索的行数由 # 指定。如果游标中剩余的行数少于 #，则只提取那些可用的行。用关键字 ALL 代替数字将检索游标中所有剩余的行。可以沿 FORWARD 和 BACKWARD 两个方向提取行。默认方向是 FORWARD。

> **提示**
>
> 行数允许指定为负数。负数等价于反转 FORWARD 和 BACKWARD 关键字的方向。例如，FORWARD -1与BACKWARD 1相同。

注意

注意，FORWARD 和 BACKWARD 关键字是 Postgres扩展。也支持 SQL92语法，即命令的第二种形式。关于兼容性问题的细节见下文。

Postgres不支持通过游标更新数据，因为把游标更新映射回基表通常不可行，VIEW 更新也是同样的情况。因此，用户必须发出显式的 UPDATE 命令来替换数据。

游标只能在事务内部使用，因为它们保存的数据跨越用户的多次查询。

使用 MOVE 改变游标位置。DECLARE 定义游标。关于事务的更多信息，请参阅 BEGIN、COMMIT 和 ROLLBACK。

用法

下面的例子使用游标遍历一个表。

```
-- set up and use a cursor:

BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

```
-- Fetch first 5 rows in the cursor liahona:
FETCH FORWARD 5 IN liahona;

 code  |          title           | did | date_prod  |  kind    | len
-------+--------------------------+-----+------------+----------+-----
--
 BL101 | The Third Man            | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen        | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                  | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                   | 103 | 1964-02-03 | Drama    | 02:28

-- Fetch previous row:
FETCH BACKWARD 1 IN liahona;

 code  | title   | did | date_prod | kind   | len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- close the cursor and commit work:

CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL92

> **注意**
>
> 游标的非嵌入式使用是Postgres 扩展。这里把游标的语法和用法与SQL92 定义的嵌入式游标形式相比较。

SQL92允许 FETCH 绝对定位游标，并允许把结果放入显式变量：

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

游标应被定位到指定的绝对行号。Postgres 中的所有行号都是相对编号，因此不支持这一能力。

:*variable*

目标主变量。

GRANT

GRANT — 把访问权限授予一个用户、一个组或所有用户

大纲

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

输入

privilege

可能的权限有：

SELECT

访问指定表/视图的所有列。

INSERT

向指定表的所有列插入数据。

UPDATE

更新指定表的所有列。

DELETE

从指定表中删除行。

RULE

在表/视图上定义规则（见 CREATE RULE 语句）。

ALL

授予所有权限。

object

要授予访问权限的对象的名称。可能的对象有：
- table
- view
- sequence

PUBLIC

代表所有用户的简写形式。

GROUP *group*

被授予权限的一个组 *group*。

username

被授予权限的用户的名称。PUBLIC 是代表所有用户的简写形式。

输出

CHANGE

如果成功则返回此消息。

ERROR: ChangeAcl: class "*object*" not found

如果指定的对象不可用，或者无法把权限授予指定的组或用户，则返回此消息。

描述

GRANT允许对象的创建者把特定权限授予所有用户（PUBLIC）或某个用户或组。
除非创建者在对象创建之后 GRANT 权限，除创建者之外的用户没有任何访问权限。

一旦用户拥有了对象上的某个权限，他就能够行使该权限。无需向对象的创建者 GRANT 权限，创建者自动持有所有权限，并且也可以删除对象。

注意

目前，在Postgres中要只对少数几个列授予权限，必须创建一个包含所需列的视图，
然后把权限授予该视图。

使用psql \z获取现有对象上权限的更多信息：

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

参阅 REVOKE 语句来撤销访问权限。

用法

把表 films 上的插入权限授予所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

把视图 kinds 上的所有权限授予用户 manuel：

```
GRANT ALL ON kinds TO manuel;
```

兼容性

SQL92

SQL92的 GRANT 语法允许为表中的单个列设置权限，也允许设置把相同权限授予他人的权限：

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

各字段与Postgres实现中的兼容，并有下列补充：

privilege

SQL92允许指定额外的权限：

SELECT

REFERENCES

允许在完整性约束中引用指定表/视图的部分或全部列。

USAGE

允许使用域、字符集、排序规则或转换。如果对象指定的不是表/视图，*privilege* 必须只指定 USAGE。

object

[TABLE] *table*

SQL92允许额外的无功能关键字TABLE。

CHARACTER SET

允许使用指定的字符集。

COLLATION

允许使用指定的排序序列。

TRANSLATION

允许使用指定的字符集转换。

DOMAIN

允许使用指定的域。

WITH GRANT OPTION

允许把相同的权限授予他人。

INSERT

INSERT — 向一个表中插入新行

大纲

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

输入

table

一个已存在表的名称。

column

*table*中一个列的名称。

expression

要赋予*column*的一个有效表达式或值。

query

一个有效的查询。有效参数的进一步描述请参考 SELECT 语句。

输出

`INSERT` *oid* `1`

只插入了一行时返回的消息。→ *oid* 是被插入行的数字 OID。

`INSERT 0` *#*

插入了多行时返回的消息。→ *#* 是插入的行数。

描述

`INSERT`允许我们向一个类或表中插入新行。可以一次插入一行，
也可以插入一个查询结果的多个行。目标列表中的列可以按任意顺序列出。

目标列表中没有出现的每个列将使用默认值插入，即声明的 DEFAULT 值或 NULL。如果向声明为 NOT NULL 的列插入 NULL，Postgres将拒绝这个新列。

如果每个列的表达式数据类型不正确，将尝试自动类型强制转换。

要向一个表追加数据，你必须拥有该表上的 insert 权限，并且对 WHERE 子句中指定的任何表拥有 select 权限。

用法

向表 `films` 插入一行：

```
INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82 minute')
;
```

在第二个例子中，列 date_prod 被省略，因此它将使用默认值 NULL：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

向表 distributors 插入一行；注意只指定了列 name，因此被省略的列 did 将被赋予其默认值：

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

从表 tmp 向表 films 插入若干行：

```
INSERT INTO films SELECT * FROM tmp;
```

插入数组（关于数组的更多信息请参考PostgreSQL User's Guide）：

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these queries create the same board attribute)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

兼容性

SQL92

INSERT与SQL92完全兼容。*query*子句特性方面可能存在的限制记录在 SELECT 中。

LISTEN

LISTEN — 监听某个通知条件上的响应

大纲

```
LISTEN name
```

输入

name

通知条件的名称。

输出

`LISTEN`

注册成功完成时返回的消息。

`NOTICE Async_Listen: We are already listening on name`

如果该后端已经注册了该通知条件。

描述

`LISTEN`把当前的 Postgres后端注册为通知条件 *name*的监听者。

每当命令 `NOTIFY name` 被调用时--无论是被本后端还是被连接到同一数据库的其他后端调用--所有当前正在监听该通知条件的后端都会被通知，
而每个后端又将进而通知它所连接的前端应用。更多信息请参阅 `NOTIFY`的讨论。

可以用`UNLISTEN`命令取消一个后端对给定通知条件的注册。此外，
后端的监听注册在后端进程退出时会被自动清除。

前端应用检测通知事件所必须使用的方法取决于它使用的 Postgres应用编程接口。使用基本的 libpq 库时，应用像发出普通 SQL 命令一样发出 `LISTEN`，然后必须周期性地调用例程 `PQnotifies`来查明是否收到了任何通知事件。其他接口（例如 libpgtcl）提供了处理通知事件的更高层方法；实际上，使用 libpgtcl 时，应用程序员甚至不必直接发出`LISTEN`或 `UNLISTEN`。更多细节请参阅你所使用的库的文档。

NOTIFY 中包含关于`LISTEN`和 `NOTIFY`用法的更广泛的讨论。

注解

name 可以是任何可作为名称使用的有效字符串；它不需要对应任何实际表的名称。如果 *notifyname* 被双引号括起来，它甚至不必是语法上有效的名称，而可以是长度不超过 31 个字符的任何字符串。

在Postgres的一些早期版本中，当*name* 不对应任何现有表名时，即使它在语法上是有效的名称，也必须用双引号把它括起来。现在已不再需要这样。

用法

在psql中配置并执行一个 listen/notify 序列：

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

兼容性

SQL92

`LISTEN` 在 SQL92 中没有。

LOAD

LOAD — 动态装载一个目标文件

大纲

```
LOAD 'filename'
```

输入

filename

用于动态装载的目标文件。

输出

LOAD

成功完成时返回的消息。

ERROR: LOAD: could not open file '*filename*'

找不到指定的文件时返回的消息。该文件必须对 Postgres后端可见，并指定适当的完整路径名，才能避免此消息。

描述

把一个目标（或".o"）文件装载到 Postgres后端地址空间中。文件一旦装载，该文件中的所有函数都可以被访问。这个函数用于支持用户定义的类型和函数。

如果文件没有用 LOAD装载，那么该文件将在函数第一次被Postgres调用时被自动装载。LOAD 也可以用来在目标文件被编辑并重新编译之后重新装载它。目前只支持由 C 语言文件创建的目标。

注解

已装载目标文件中的函数不应调用其他通过 LOAD 命令装载的目标文件中的函数。例如，文件A中的所有函数应当互相调用、调用标准库或数学库中的函数、或调用 Postgres 本身中的函数。它们不应调用另一个已装载文件 B中定义的函数。这是因为如果B被重新装载，Postgres 装载器无法把 A中函数的调用重定位到 B的新地址空间。但如果B没有被重新装载，则不会有问题。

目标文件必须编译为包含位置无关代码。例如，在 DECstation 上，编译要装载的目标文件时必须使用 /bin/cc 加上-G 0选项。

注意，如果你正在把Postgres 移植到新平台，LOAD 必须能够工作才能支持 ADT。

用法

装载文件/usr/postgres/demo/circle.o:

```
LOAD '/usr/postgres/demo/circle.o'
```

兼容性

SQL92

`LOAD` 在 SQL92 中没有。

LOCK

LOCK — 在事务内显式锁定一个表

大纲

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Inputs

name

要锁定的现有表的名称。

ACCESS SHARE MODE

> **注意**
>
> 这种锁模式会在被查询的表上自动获取。

这是限制性最低的锁模式。它只与 ACCESS EXCLUSIVE 模式冲突。它用于保护表不被并发的 `ALTER TABLE`、`DROP TABLE` 和 `VACUUM` 命令修改。

ROW SHARE MODE

> **注意**
>
> 由 `SELECT...FOR UPDATE` 自动获取。它是共享锁，但之后可以升级为 ROW EXCLUSIVE 锁。

与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

ROW EXCLUSIVE MODE

> **注意**
>
> 由 `UPDATE`、`DELETE` 和 `INSERT` 语句自动获取。

与 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

SHARE MODE

> **注意**
>
> 由 `CREATE INDEX` 自动获取。对整张表加共享锁。

与 ROW EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。此模式保护表不被并发更新影响。

SHARE ROW EXCLUSIVE MODE

> **注意**
>
> 这与 EXCLUSIVE MODE 类似，但允许别人持有 SHARE ROW 锁。

与 ROW EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

EXCLUSIVE MODE

> **注意**
>
> 此模式比 SHARE ROW EXCLUSIVE 限制性更强。它阻塞所有并发的 ROW SHARE/SELECT...FOR UPDATE 查询。

与 ROW SHARE、ROW EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

ACCESS EXCLUSIVE MODE

> **注意**
>
> 由 `ALTER TABLE`、`DROP TABLE`、`VACUUM` 语句自动获取。这是限制性最高的锁模式，与所有其他锁模式冲突，保护被锁表免受任何并发操作。

> **注意**
>
> 不带限定的 `LOCK TABLE`（即没有显式锁模式选项的命令）也会获取这种锁模式。

Outputs

`LOCK TABLE`

锁应用成功。

`ERROR` *`name`*`: Table does not exist.`

如果 *`name`* 不存在则返回此消息。

描述

`LOCK TABLE` 在事务期间控制对一个表的并发访问。Postgres 在可能时总是使用限制性最低的锁模式。`LOCK TABLE` 用于你可能需要更严格锁定的场合。

RDBMS 锁定使用下列术语：

EXCLUSIVE

排他锁，阻止授予其他锁。

SHARE

允许别人共享锁定。阻止 EXCLUSIVE 锁。

ACCESS

锁定表模式。

ROW

锁定单个行。

> **注意**
>
> 如果未指定 EXCLUSIVE 或 SHARE，则假定为 EXCLUSIVE。锁在事务期间存在。

例如，一个应用以 READ COMMITTED 隔离级别运行事务，并且需要确保表中的数据在事务期间保持存在。为此，你可以在查询之前对该表使用 SHARE 锁模式。这将保护数据不被并发更改，并为随后对该表的读取操作提供处于实际当前状态的数据，因为 SHARE 锁模式与写者取得的任何 ROW EXCLUSIVE 锁冲突，而你的 `LOCK TABLE name IN SHARE MODE` 语句会等待任何并发写操作提交或回滚。

> **注意**
>
> 要在以 SERIALIZABLE 隔离级别运行事务时读取处于真实当前状态的数据，你必须在执行任何 DML 语句之前执行一条 LOCK TABLE 语句，因为正是在那时事务确定哪些并发更改对自己可见。

除上述要求外，如果事务要更改表中的数据，就应取得 SHARE ROW EXCLUSIVE 锁模式，以防止两个并发事务都试图以 SHARE 模式锁定该表、然后再试图更改表中数据时出现死锁——两者（隐式）取得的 ROW EXCLUSIVE 锁模式都与并发的 SHARE 锁冲突。

继续讨论上面提出的死锁（两个事务互相等待）
问题：你应当遵循两条通用规则来防止死锁条件：

- 事务必须以相同的顺序对相同的对象获取锁。

  例如，如果一个应用先更新行 R1 再更新行 R2（在同一个事务中），那么第二个应用如果稍后要更新行 R1（在单个事务中），就不应先更新行 R2。相反，它应按照与第一个应用相同的顺序更新行 R1 和 R2。

- 只有两个冲突锁模式之一是自冲突的（即同一时刻只能被一个事务持有）时，
  事务才应同时获取这两个锁模式。如果涉及多种锁模式，
  事务应总是先获取限制性最高的模式。

  前面在讨论使用 SHARE ROW EXCLUSIVE 模式而非 SHARE 模式时已给出过这条规则的一个例子。

> **注意**
>
> Postgres 确实会检测死锁，并会回滚至少一个等待中的事务来解决死锁。

注意

`LOCK` 是一种 Postgres 语言扩展。

除 ACCESS SHARE/EXCLUSIVE 锁模式外，其他所有 Postgres 锁模式和 `LOCK TABLE` 语法都与 Oracle 中的对应语法兼容。

`LOCK` 只在事务内部起作用。

用法

演示在准备向外键表执行插入时，在主键表上获取一个 SHARE 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果未返回记录则执行 ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在准备执行删除操作时，在主键表上获取一个 SHARE ROW EXCLUSIVE 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

SQL92

SQL92 中没有 `LOCK TABLE`，而是使用 `SET TRANSACTION` 来指定事务的并发级别。我们也支持这一点；详见 SET。

MOVE

MOVE — 移动游标位置

大纲

```
MOVE [ direction ] [ count ]
    { IN | FROM } cursor
```

描述

MOVE允许用户把游标位置移动指定的行数。MOVE的工作方式类似FETCH命令，但它只定位游标而不返回行。

有关语法和用法的细节，请参阅 FETCH。

注意

MOVE是一个Postgres 语言扩展。

有效参数的描述请参阅 FETCH。定义游标请参阅 DECLARE。有关事务的更多信息，请参阅 BEGIN、COMMIT 和 ROLLBACK。

用法

建立并使用一个游标：

```
BEGIN WORK;
DECLARE liahona CURSOR  FOR SELECT * FROM films;
-- 跳过前 5 行:
MOVE FORWARD 5 IN liahona;
MOVE
-- 从游标 liahona 中提取第 6 行:
FETCH 1 IN liahona;
FETCH

 code  | title  | did | date_prod | kind   | len
-------+--------+-----+-----------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22| Action | 01:37
(1 row)
-- 关闭游标 liahona 并提交事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL92

SQL92中没有MOVE语句。作为替代，SQL92允许从游标的绝对位置 FETCH行，从而隐式地把游标移动到正确的位置。

NOTIFY

NOTIFY — 向所有正在监听某个通知条件的前端和后端发信号

大纲

```
NOTIFY name
```

Inputs

notifyname

要发信号的通知条件。

Outputs

NOTIFY

确认 notify 命令已执行。

Notify events

事件被送达正在监听的前端；每个前端应用是否响应以及如何响应取决于它的程序设计。

描述

NOTIFY 命令向当前数据库中每个此前已为指定通知条件执行过 LISTEN *notifyname* 的前端应用发送一个通知事件。

传递给前端的通知事件信息包括通知条件名和发出通知的后端进程的 PID。由数据库设计者来定义给定数据库中将要使用的条件名以及每个条件名的含义。

通常，通知条件名与数据库中某张表的名称相同，通知事件本质上意味着"我改动了这张表，看看它有什么新内容"。但 NOTIFY 和 LISTEN 命令并不强制这种关联。例如，数据库设计者可以用几个不同的条件名来表示对单个表不同种类的更改。

NOTIFY 为访问同一 Postgres 数据库的一组进程提供了一种简单的信号或 IPC（进程间通信）机制。通过使用数据库中的表，可以构建从通知者向监听者传递附加数据（而不仅仅是条件名）的更高层机制。

当NOTIFY用于表明某张特定表发生变化时，一种很有用的编程技巧是把NOTIFY放进由表更新触发的规则中。这样一来，每当表发生变化时就会自动发出通知，应用程序员也不容易忘记这么做。

NOTIFY 在若干重要方面与 SQL 事务交互。首先，如果在事务内部执行了 NOTIFY，通知事件直到且仅当事务提交时才会被送达。这是恰当的，因为如果事务中止，我们希望其中的所有命令都不产生效果，包括 NOTIFY。但如果期望通知事件立即送达，这可能会让人不安。其次，如果一个正在监听的后端在处于事务中时收到一个通知信号，该通知事件要到事务刚完成（提交或中止）之后才会送达其连接的前端。同样，其理由是：如果一个通知在某个后来被中止的事务内被送达，人们会希望以某种方式撤销该通知——但后端一旦把通知发给了前端，就无法"收回"它。因此通知事件只在事务之间送达。由此得出的结论是：使用 NOTIFY 进行实时信号传递的应用应尽量保持事务简短。

NOTIFY 在一个重要方面的行为类似 Unix 信号：如果在短时间内多次发出同一通知名的信号，接收者可能对多次执行的 NOTIFY 只收到一个通知事件。因此，依赖收到通知的数量并不是一个好主意。正确的做法是用 NOTIFY 唤醒需要关注某事的应用，而用一个数据库对象（例如序列）来跟踪发生了什么以及发生了多少次。

发送 NOTIFY 的前端自己同时也在监听同一通知名，这是很常见的情形。在这种情况下，它会像其他所有监听前端一样收到一个通知事件。根据应用逻辑，
这可能导致无用功——例如再次读取数据库表，找到的却正是该前端自己刚刚写入的更新。在 Postgres 6.4 及之后的版本中，可以通过检查发出通知的后端进程的 PID（在通知事件消息中提供）是否与自己的后端的 PID（可从 libpq 获得）相同来避免这种额外工作。二者相同时，该通知事件就是自己发的工作回弹回来，可以忽略。（尽管前一段有那样的说法，
但这是一种安全的技术。Postgres会把自身通知与来自其他后端的通知分开保存，
因此忽略自己的通知并不会让你错过来自外部的通知。）

注意

name 可以是任何作为名称有效的字符串；它不必对应任何实际表的名称。如果 *name* 用双引号括起，它甚至不必是语法上有效的名称，而可以是长度最多 31 个字符的任意字符串。

在 Postgres 的一些早期版本中，当 *name* 不对应任何现有表名时，即使语法上是有效的名称，也必须用双引号括起。这一要求已不再存在。

在 6.4 之前的 Postgres 版本中，通知消息中传递的后端 PID 总是前端自己的后端的 PID。因此在那些早期版本中，无法把自己的通知与其他客户的通知区分开。

用法

在psql中配置并执行一组 listen/notify 操作：

```
=> LISTEN virtual;
=> NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

兼容性

SQL92

SQL92 中没有 NOTIFY 语句。

REINDEX

REINDEX — 在独立 Postgres 下恢复损坏的系统索引

大纲

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

输入

TABLE

重建指定表的所有索引。

DATABASE

重建指定数据库的所有系统索引。

INDEX

重建一个指定的索引。

name

要重建索引的特定表/数据库/索引的名称。

FORCE

强制重建索引。不带此关键字时，除非目标索引已被标记为无效，否则 REINDEX 不会做任何事情。

输出

`REINDEX`

表成功重建索引后返回的消息。

描述

`REINDEX`用于恢复损坏的系统索引。为了运行 REINDEX 命令，必须关闭 postmaster，改为启动带 -O 和 -P 选项（忽略系统索引的选项）的独立 Postgres。注意，我们不能依赖系统索引来恢复系统索引。

用法

重建表`mytable`：

```
REINDEX TABLE mytable;
```

另外一些例子：

```
REINDEX DATABASE my_database FORCE;
```

```
REINDEX INDEX my_index;
```

兼容性

SQL92

REINDEX 在 SQL92 中没有。

RESET

RESET — 恢复会话的运行时参数为默认值

大纲

```
RESET variable
```

输入

variable

有关可用变量的更多信息，请参阅 SET。

输出

```
RESET VARIABLE
```

如果*variable*被成功重置为默认值，返回此消息。

描述

`RESET`把变量恢复为默认值。有关允许的取值和默认值，请参阅 SET。`RESET`是下面语句的替代形式

```
SET variable = DEFAULT
```

注意

另外还可以使用 SET 和 SHOW 来操作变量的值。

用法

把 DateStyle 设为默认值：

```
RESET DateStyle;
```

把 Geqo 设为默认值：

```
RESET GEQO;
```

兼容性

SQL92

SQL92中没有`RESET`。

REVOKE

REVOKE — 从一个用户、一个组或所有用户回收访问权限。

大纲

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP groupname | username }
```

输入

privilege

可能的权限有：

SELECT

访问一个指定表/视图的所有列的权限。

INSERT

向一个指定表的所有列插入数据的权限。

UPDATE

更新一个指定表的所有列的权限。

DELETE

从一个指定表删除行的权限。

RULE

在表/视图上定义规则的权限。（见 CREATE RULE）。

ALL

回收所有权限。

object

要回收访问权限的对象的名称。可能的对象有：
- 表
- 视图
- 序列

group

要从中回收权限的组的名称。

username

要从中回收权限的用户的名称。使用 PUBLIC 关键字指定所有用户。

PUBLIC

为所有用户回收指定的权限。

输出

CHANGE

成功时返回的消息。

ERROR

对象不可用或不可能从一个组或用户回收权限时返回的消息。

描述

REVOKE 允许一个对象的创建者回收先前授予所有用户（通过 PUBLIC）或某个用户或组的权限。

注意

关于现有对象上权限的更多信息，请参阅 psql 的 \z 命令：

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
     uname=arwR -- privileges granted to a user
     group gname=arwR -- privileges granted to a GROUP
     =arwR -- privileges granted to PUBLIC

     r -- SELECT
     w -- UPDATE/DELETE
     a -- INSERT
     R -- RULE
     arwR -- ALL
```

提示

目前，要创建一个 GROUP，你必须手动向表 pg_group 中插入数据：

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

用法

从所有用户回收表 films 上的插入权限：

```
REVOKE INSERT ON films FROM PUBLIC;
```

回收用户 manuel 在视图 kinds 上的所有权限：

```
REVOKE ALL ON kinds FROM manuel;
```

兼容性

SQL92

SQL92 的 REVOKE 语法有更多回收权限的能力，包括对表中个别列的权限：

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column [,
 ...] ) ]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

各个字段的细节请参阅 GRANT。

```
REVOKE GRANT OPTION FOR privilege [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

回收一个用户把指定权限授予其他人的权力。各个字段的细节请参阅 GRANT。

可能的对象有：

[TABLE] 表/视图
CHARACTER SET 字符集
COLLATION 排序规则
TRANSLATION 转换
DOMAIN 域

如果 user1 把一个权限带 WITH GRANT OPTION 地授予 user2，而 user2 又把它授予 user3，那么 user1 可以使用 CASCADE 关键字级联回收该权限。

如果 user1 把一个权限带 WITH GRANT OPTION 地授予 user2，而 user2 又把它授予 user3，那么当 user1 试图回收该权限时，如果他/她指定了 RESTRICT 关键字，回收将失败。

ROLLBACK

ROLLBACK — 中止当前的事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ]
```

输入

无。

输出

`ABORT`

成功时返回的消息。

`NOTICE: ROLLBACK: no transaction in progress`

如果当前没有正在进行的事务。

描述

`ROLLBACK` 回滚当前事务并使该事务所做的所有更新被丢弃。

注意

使用 COMMIT 成功地终止一个事务。ABORT 是 `ROLLBACK` 的一个同义词。

用法

要中止所有更改：

```
ROLLBACK WORK;
```

兼容性

SQL92

SQL92 只指定了 `ROLLBACK` 和 `ROLLBACK WORK` 两种形式。除此之外完全兼容。

SELECT

SELECT — 从表或视图中检索行

大纲

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

输入

expression

表的列名或一个表达式。

name

用 AS 子句为列或表达式指定另一个名称。此名称主要用于为列提供一个便于显示的标签。它也可以用于在 ORDER BY 和 GROUP BY 子句中引用该列的值。但 *name* 不能用于 WHERE 或 HAVING 子句；应改写该表达式。

TEMPORARY
TEMP

如果指定了 TEMPORARY 或 TEMP，则该表是为此会话唯一创建的，并在会话退出时自动删除。

new_table

如果指定了 INTO TABLE 子句，查询结果将存储在一个具有指定名称的新表中。目标表（*new_table*）会自动创建，在此命令执行之前必须不存在。更多信息请参阅 SELECT INTO。

> **注意**
>
> CREATE TABLE AS 语句同样会从一个 select 查询创建一个新表。

table

FROM 子句引用的现有表的名称。

alias

前面表的替代名称。它用于简洁或消除单个表内连接的歧义。

condition

一个结果为真或假的布尔表达式。见 WHERE 子句。

column

表的列名。

select

一个除 ORDER BY 和 LIMIT 子句外具有所有特性的 select 语句。

输出

Rows

由查询说明产生的完整行集合。

→ *count*

查询返回的行数。

描述

`SELECT` 将从一个或多个表返回行。选择的候选是满足 WHERE 条件的行；如果省略 WHERE，则所有行都是候选。(See WHERE 子句.)

`DISTINCT` 将从结果中消除重复行。`ALL`（默认）将返回所有候选行，包括重复行。

`DISTINCT ON` 消除在所有指定表达式上匹配的行，只保留每组重复行中的第一行。DISTINCT ON 表达式使用与 ORDER BY 项相同的规则解释；见下文。注意，除非用 `ORDER BY` 保证想要的行出现在最前，否则每个集合的"第一行"是不可预测的。例如，

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

它取回每个位置的最新天气报告。但如果我们没有用 ORDER BY 强制每个位置的时间值降序，就会得到每个位置年龄不可预测的报告。

GROUP BY 子句允许用户把一个表划分为在一个或多个值上匹配的行组。(See GROUP BY 子句.)

HAVING 子句允许只选择满足指定条件的那些行组。(See HAVING 子句.)

ORDER BY 子句使返回的行按指定的顺序排序。如果没有给出 ORDER BY，行按系统认为产生成本最低的任何顺序返回。(See ORDER BY 子句.)

UNION 操作符允许结果是所涉及查询返回的行的集合。(See UNION 子句.)

INTERSECT 操作符给出两个查询共同的行。(See INTERSECT 子句.)

EXCEPT 操作符给出第一个查询返回而第二个查询不返回的行。(See EXCEPT 子句.)

FOR UPDATE 子句允许 SELECT 语句对选中的行执行排他锁定。

LIMIT 子句允许把查询产生的行的一个子集返回给用户。(See LIMIT 子句.)

你必须对一个表拥有 SELECT 权限才能读取它的值（见 GRANT/REVOKE 语句）。

WHERE 子句

可选的 WHERE 条件具有如下一般形式：

```
WHERE boolean_expr
```

boolean_expr 可以是任何求值为布尔值的表达式。在很多情况下，这个表达式会是

exprcond_opexpr

或

log_opexpr

其中 *cond_op* 可以是 =、<、<=、>、>= 或 <> 之一，或像 ALL、ANY、IN、LIKE 这样的条件操作符，或本地定义的操作符，而 *log_op* 可以是 AND、OR、NOT 之一。SELECT 将忽略所有 WHERE 条件不返回 TRUE 的行。

GROUP BY 子句

GROUP BY 指定通过应用此子句导出的分组表：

```
GROUP BY column [, ...]
```

GROUP BY 把在分组列上共享相同值的所有选中行压缩为单行。聚合函数（如果使用了的话）会在组成每个组的所有行上进行计算，为每个组产生一个单独的值（而没有 GROUP BY 时，聚合会产生一个在所有被选中的行上计算出的单一值）。当存在 GROUP BY 时，SELECT 输出表达式引用未分组的列是无效的，除非是在聚合函数内部，因为对一个未分组的列可能会有多个可能的返回值。

GROUP BY 中的项也可以是输出列（SELECT 表达式）的名称或序号，或者是由输入列值构成的任意表达式。在有歧义时，GROUP BY 名称将被解释为输入列名而不是输出列名。

HAVING 子句

可选的 HAVING 条件具有如下一般形式：

```
HAVING cond_expr
```

其中 *cond_expr* 与为 WHERE 子句指定的相同。

HAVING 指定通过消除不满足 *cond_expr* 的组行而导出的分组表。HAVING 与 WHERE 不同：WHERE 在应用 GROUP BY 之前过滤单独的行，而 HAVING 过滤 GROUP BY 创建的组行。

cond_expr 中引用的每一列都应无歧义地引用一个分组列，除非该引用出现在聚合函数内。

ORDER BY 子句

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column 既可以是结果列名，也可以是序号。

序号指结果列的（从左到右的）位置。这一特性使得可以基于没有合适名称的列定义排序。这永远不是绝对必要的，因为总是可以用 AS 子句为结果列赋予一个名称，例如：

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

也可以按任意表达式 ORDER BY（对 SQL92 的扩展），包括不出现在 SELECT 结果列表中的字段。因此下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

注意，如果 ORDER BY 项是一个既匹配结果列名又匹配输入列名的简单名称，ORDER BY 将把它解释为结果列名。这与 GROUP BY 在相同情况下所作的选择相反。这种不一致是 SQL92 标准规定的。

还可以在 ORDER BY 子句中的每个列名后面添加关键字 DESC（降序）或 ASC（升序）。如果未指定，默认为 ASC。或者，可以指定一个特定的排序操作符名称。ASC 等价于 USING '<' 而 DESC 等价于 USING '>'。

UNION 子句

```
table_query UNION [ ALL ] table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

其中 table_query 指定不带 ORDER BY 或 LIMIT 子句的任何 select 表达式。

UNION 操作符允许结果是所涉及查询返回的行的集合。表示 UNION 的直接操作数的两个 SELECT 必须产生相同数量的列，并且对应的列必须是兼容的数据类型。

默认情况下，UNION 的结果不包含任何重复行，除非指定了 ALL 子句。

同一 SELECT 语句中的多个 UNION 操作符从左到右求值。注意 ALL 关键字不是全局性的，只应用于当前这对表结果。

INTERSECT 子句

```
table_query INTERSECT table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

其中 table_query 指定不带 ORDER BY 或 LIMIT 子句的任何 select 表达式。

INTERSECT 操作符给出两个查询共同的行。表示 INTERSECT 的直接操作数的两个 SELECT 必须产生相同数量的列，并且对应的列必须是兼容的数据类型。

同一 SELECT 语句中的多个 INTERSECT 操作符从左到右求值，除非圆括号另有指示。

EXCEPT 子句

```
table_query EXCEPT table_query
```

```
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

其中 table_query 指定不带 ORDER BY 或 LIMIT 子句的任何 select 表达式。

EXCEPT 操作符给出第一个查询返回而第二个查询不返回的行。表示 EXCEPT 的直接操作数的两个 SELECT 必须产生相同数量的列，并且对应的列必须是兼容的数据类型。

同一 SELECT 语句中的多个 EXCEPT 操作符从左到右求值，除非圆括号另有指示。

LIMIT 子句

```
    LIMIT { count | ALL } [ { OFFSET | , } start ]
    OFFSET start
```

其中 count 指定返回的最大行数，start 指定开始返回行之前要跳过的行数。

LIMIT 允许只取回由查询其余部分产生的行的一部分。如果给出了限制数，则返回不超过那么多的行。如果给出了偏移，则在开始返回行之前跳过那么多的行。

使用 LIMIT 时，最好使用把结果行约束为唯一顺序的 ORDER BY 子句。否则你会得到查询行的一个不可预测的子集——你可能要的是第十到第二十行，
但是按什么顺序的第十到第二十行？除非指定了 ORDER BY，否则你不知道顺序。

从 Postgres 7.0 起，查询优化器在生成查询计划时会把 LIMIT 考虑在内，所以很可能根据 LIMIT 和 OFFSET 的不同取值得到不同的计划（产生不同的行顺序）。因此，使用不同的 LIMIT/OFFSET 值选择查询结果的不同子集会得到不一致的结果，除非用 ORDER BY 强制可预测的结果顺序。这不是缺陷；这是 SQL 不承诺以任何特定顺序交付查询结果（除非用 ORDER BY 约束顺序）这一事实的固有后果。

用法

把表 films 与表 distributors 连接：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

           title           | did |   name           | date_prod  |
 kind
---------------------------+-----+------------------+------------+---
-------
 The Third Man             | 101 | British Lion     | 1949-12-23 |
 Drama
 The African Queen         | 101 | British Lion     | 1951-08-11 |
 Romantic
 Une Femme est une Femme   | 102 | Jean Luc Godard  | 1961-03-12 |
 Romantic
 Vertigo                   | 103 | Paramount        | 1958-11-14 |
 Action
 Becket                    | 103 | Paramount        | 1964-02-03 |
 Drama
 48 Hrs                    | 103 | Paramount        | 1982-10-22 |
 Action
```

```
 War and Peace             | 104 | Mosfilm          | 1967-02-12 |
 Drama
 West Side Story           | 105 | United Artists   | 1961-01-03 |
 Musical
 Bananas                   | 105 | United Artists   | 1971-07-13 |
 Comedy
 Yojimbo                   | 106 | Toho             | 1961-06-16 |
 Drama
 There's a Girl in my Soup | 107 | Columbia         | 1970-06-11 |
 Comedy
 Taxi Driver               | 107 | Columbia         | 1975-05-15 |
 Action
 Absence of Malice         | 107 | Columbia         | 1981-11-15 |
 Action
 Storia di una donna       | 108 | Westward         | 1970-08-15 |
 Romantic
 The King and I            | 109 | 20th Century Fox | 1956-08-11 |
 Musical
 Das Boot                  | 110 | Bavaria Atelier  | 1981-11-11 |
 Drama
 Bed Knobs and Broomsticks | 111 | Walt Disney      |            |
 Musical
(17 rows)
```

对所有影片的 len 列求和并按 kind 分组：

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
(5 rows)
```

对所有影片的 len 列求和，按 kind 分组，并显示少于 5 小时的组总计：

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL '5 hour';

 kind     | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
(2 rows)
```

下面两个例子是按照第二列（name）的内容对各个结果排序的相同方式：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
(13 rows)
```

这个例子展示如何获得表 distributors 和 actors 的并集，把结果限制为每个表中以字母 W 开头的行。只需要不重复的行，所以省略了 ALL 关键字：

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                          ...

SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%'

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

兼容性

扩展

Postgres 允许在查询中省略 FROM 子句。此特性是从最初的 PostQuel 查询语言保留下来的：

```
SELECT distributors.* WHERE name = 'Westwood';

 did | name
```

```
-----+----------
 108 | Westward
```

SQL92

SELECT 子句

在 SQL92 标准中，可选的关键字 "AS" 只是无意义的噪音，可以省略而不影响含义。Postgres 解析器在重命名列时要求这个关键字，因为类型可扩展特性在此上下文中会导致解析歧义。

DISTINCT ON 短语不是 SQL92 的一部分。LIMIT 和 OFFSET 也不是。

在 SQL92 中，ORDER BY 子句只能使用结果列名或编号，而 GROUP BY 子句只能使用输入列名。Postgres 对这两个子句都做了扩展，允许也使用另一种选择（但在有歧义时采用标准的解释）。Postgres 还允许两个子句指定任意表达式。注意，表达式中出现的名称将总是被当作输入列名，而不是结果列名。

UNION 子句

SQL92 的 UNION 语法允许一个附加的 CORRESPONDING BY 子句：

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

Postgres 不支持 CORRESPONDING BY 子句。

SELECT INTO

SELECT INTO — 从一个现有的表或视图创建一个新表

大纲

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

输入

所有输入字段在 SELECT 中有详细描述。

输出

所有输出字段在 SELECT 中有详细描述。

描述

`SELECT INTO` 从查询的结果创建一个新表。通常这个查询从现有的表提取数据，但允许任何 SQL 查询。

> **注意**
>
> CREATE TABLE AS 在功能上等价于 `SELECT INTO` 命令。

SET

SET — 设置会话的运行时参数

大纲

```
SET variable { TO | = } { value | 'value' | DEFAULT }
SET CONSTRAINTS {ALL | constraintlist} mode
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

输入

variable

可设置的全局参数。

value

参数的新值。DEFAULT 可用于指定把参数重置为默认值。允许使用字符串列表，但更复杂的构造可能需要加单引号或双引号。

可能的变量和允许的值有：

CLIENT_ENCODING | NAMES

设置多字节客户端编码。参数有：

value

把多字节客户端编码设为 *value*。指定的编码必须为后端所支持。

只有在构建 Postgres 的 configure 步骤中启用了 MULTIBYTE 支持，这个选项才可用。

DATESTYLE

设置日期/时间的表示风格。影响输出格式，某些情况下也会影响输入的解释。

ISO

使用 ISO 8601 风格的日期和时间

SQL

使用 Oracle/Ingres 风格的日期和时间

Postgres

使用传统的 Postgres 格式

European

数值日期表示使用 dd/mm/yyyy。

NonEuropean

数值日期表示使用 mm/dd/yyyy。

German

数值日期表示使用 dd.mm.yyyy。

US

与 NonEuropean 相同

DEFAULT

恢复默认值（ISO）

日期格式的初始化可以通过以下方式完成：

设置 PGDATESTYLE 环境变量。如果在基于 libpq 的客户端前端环境中设置了 PGDATESTYLE，libpq 会在连接启动时自动把 DATESTYLE 设为 PGDATESTYLE 的值。
使用选项 -o -e 运行 postmaster，把日期设为 European 约定。注意这只影响日期风格的某些组合；例如 ISO 风格不受这个参数影响。
更改 src/backend/utils/init/globals.c.

globals.c 中可以更改的变量有：

bool EuroDates = false | true
int DateStyle = USE_ISO_DATES | USE_POSTGRES_DATES | USE_SQL_DATES | USE_GERMAN_DATES

SEED

设置随机数生成器的内部种子。

value

供 random 目录函数使用的种子的值。有效的值是 0 到 1 之间的浮点数，它们随后会与 RAND_MAX 相乘。如果使用范围之外的数，该乘积会静默溢出。

种子也可以通过调用 setseed SQL 函数来设置：

```
SELECT setseed(value);
```

只有在构建 Postgres 的 configure 步骤中启用了 MULTIBYTE 支持，这个选项才可用。

SERVER_ENCODING

把多字节服务器编码设置为：

value

服务器编码的标识值。

只有在构建 Postgres 的 configure 步骤中启用了 MULTIBYTE 支持，这个选项才可用。

CONSTRAINTS

SET CONSTRAINTS 影响当前事务中约束求值的行为。SET CONSTRAINTS 在 SQL3 中规定，它允许下列参数：

constraintlist

以逗号分隔的可延迟约束名称列表。

mode

约束模式。允许的值是 DEFERRED 和 IMMEDIATE。

在 IMMEDIATE 模式下，外键约束在每条查询结束时检查。

在 DEFERRED 模式下，标记为 DEFERRABLE 的外键约束只在事务提交时检查，或者直到其模式被显式设为 IMMEDIATE。实际上这只对外键约束实施，因此不适用于 UNIQUE 或其他约束。

TIME ZONE
TIME ZONE

timezone 的可用值取决于你的操作系统。例如在 Linux 上，/usr/lib/zoneinfo 包含时区数据库。

下面是一些有效的 timezone 值：

PST8PDT

把时区设为加利福尼亚州

Portugal

把时区设为葡萄牙。

'Europe/Rome'

把时区设为意大利。

DEFAULT

把时区设为你的本地时区（即 TZ 环境变量的值）。

如果指定了无效的时区，时区会变成 GMT（至少在大多数系统上如此）。

上面所示的第二种语法允许用与 SQL92 SET TIME ZONE 类似的语法设置时区。LOCAL 关键字只是为兼容 SQL92 而提供的 DEFAULT 的另一种形式。

如果在基于 libpq 的客户端前端环境中设置了 PGTZ 环境变量，libpq 会在连接启动时自动把 TIMEZONE 设为 PGTZ 的值。

TRANSACTION ISOLATION LEVEL

设置当前事务的隔离级别。

READ COMMITTED

当前事务的查询只读取在查询开始之前已提交的行。READ COMMITTED 是默认值。

> **注意**
>
> SQL92 标准要求 SERIALIZABLE 为默认的隔离级别。

SERIALIZABLE

当前事务的查询只读取在本事务中第一条 DML 语句 (SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO) 执行之前已提交的行。

还有一些内部参数或优化参数也可以由 SET 命令指定：

PG_OPTIONS

设置各种后端参数。

RANDOM_PAGE_COST

设置优化器对非顺序获取磁盘页代价的估计。它以顺序获取一页代价的倍数来度量。

float8

把随机页面访问的代价设为指定的浮点值。

CPU_TUPLE_COST

设置优化器对查询期间处理每个元组代价的估计。
它以顺序获取一页代价的一个分数来度量。

float8

把每元组 CPU 处理的代价设为指定的浮点值。

CPU_INDEX_TUPLE_COST

设置优化器对索引扫描期间处理每个索引元组代价的估计。
它以顺序获取一页代价的一个分数来度量。

float8

把每索引元组 CPU 处理的代价设为指定的浮点值。

CPU_OPERATOR_COST

设置优化器对处理 WHERE 子句中每个操作符代价的估计。它以顺序获取一页代价的一个分数来度量。

float8

把每操作符 CPU 处理的代价设为指定的浮点值。

EFFECTIVE_CACHE_SIZE

设置优化器对磁盘缓存有效大小（即内核磁盘缓存中会用于 Postgres 数据文件的那部分）的假设。它以磁盘页为单位度量，通常每页 8KB。

float8

把假设的缓存大小设为指定的浮点值。

ENABLE_SEQSCAN

启用或禁用规划器对顺序扫描计划类型的使用。（不可能完全压制顺序扫描，
但把这个变量设为 OFF 会让规划器在还有其他方法可用时不倾向于使用它。）

ON

启用顺序扫描的使用（默认设置）。

OFF

禁用顺序扫描。

ENABLE_INDEXSCAN

启用或禁用规划器对索引扫描计划类型的使用。

ON

启用索引扫描的使用（默认设置）。

OFF

禁用索引扫描。

ENABLE_TIDSCAN

启用或禁用规划器对 TID 扫描计划类型的使用。

ON

启用 TID 扫描的使用（默认设置）。

OFF

禁用 TID 扫描。

ENABLE_SORT

启用或禁用规划器对显式排序步骤的使用。（不可能完全压制显式排序，但把这个变量设为 OFF 会让规划器在还有其他方法可用时不倾向于使用它。）

ON

启用排序的使用（默认设置）。

OFF

禁用排序。

ENABLE_NESTLOOP

启用或禁用规划器对嵌套循环连接计划的使用。（不可能完全压制嵌套循环连接，但把这个变量设为 OFF 会让规划器在还有其他方法可用时不倾向于使用它。）

ON

启用嵌套循环连接的使用（默认设置）。

OFF

禁用嵌套循环连接。

ENABLE_MERGEJOIN

启用或禁用规划器对合并连接计划的使用。

ON

启用合并连接的使用（默认设置）。

OFF

禁用合并连接。

ENABLE_HASHJOIN

启用或禁用规划器对哈希连接计划的使用。

ON

启用哈希连接的使用（默认设置）。

OFF

禁用哈希连接。

GEQO

设置使用遗传优化器算法的阈值。

ON

对涉及 11 个或更多表的语句启用遗传优化器算法。（这也是 DEFAULT 设置。）

ON=*#*

接受一个整数参数，对查询中涉及 *#* 个或更多表的语句启用遗传优化器算法。

OFF

禁用遗传优化器算法。

关于查询优化的更多信息，见程序员指南中的 GEQO 一章。

如果在基于 libpq 的客户端前端环境中设置了 PGGEQO 环境变量，libpq 会在连接启动时自动把 GEQO 设为 PGGEQO 的值。

KSQO

键集查询优化器使查询规划器把 WHERE 子句包含许多 OR 连接的 AND 子句（例如 "WHERE (a=1 AND b=2) OR (a=2 AND b=3) ..."）的查询转换为 UNION 查询。这种方法可能比默认实现更快，但不一定给出完全相同的结果，因为 UNION 隐式添加了 SELECT DISTINCT 子句来消除相同的输出行。KSQO 常用于配合 MicroSoft Access 之类的产品，它们往往生成这种形式的查询。

ON

启用这种优化。

OFF

禁用这种优化（默认设置）。

DEFAULT

等价于指定 SET KSQO=OFF。

KSQO 算法过去对带有许多 OR 连接 AND 子句的查询绝对必不可少，但在 Postgres 7.0 及以后版本中标准规划器已经能相当好地处理这些查询。

MAX_EXPR_DEPTH

设置解析器可以接受的最大表达式嵌套深度。默认值对任何正常查询都足够高，但需要时可以调高。（但如果调得太高，就有因栈溢出而导致后端崩溃的风险。）

integer

最大深度。

输出

SET VARIABLE

成功时返回的消息。

NOTICE: Bad value for *variable* (*value*)

如果命令未能设置指定的变量。

描述

SET 会在会话期间修改变量的配置参数。

当前值可以用 SHOW 获得，值可以用 RESET 恢复为默认值。参数和值不区分大小写。注意值字段总是以字符串形式指定，因此用单引号括起。

SET TIME ZONE 更改会话的默认时区偏移。SQL 会话总是以一个初始的默认时区偏移开始。SET TIME ZONE 语句用于更改当前 SQL 会话的默认时区偏移。

注解

SET *variable* 语句是一种 Postgres 语言扩展。

请参阅 SHOW 和 RESET 来显示或重置当前值。

用法

把日期样式设置为 ISO（参数不需要加引号）：

```
SET DATESTYLE TO ISO;
```

为有 4 个或更多表的查询启用 GEQO（注意用单引号处理值参数中的等号）：

```
SET GEQO = 'ON=4';
```

把 GEQO 设为默认值：

```
SET GEQO = DEFAULT;
```

把时区设为加州伯克利，使用双引号保留时区说明符的大写属性：

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 07:41:21-08
```

把时区设为意大利（注意需要用单引号或双引号处理特殊字符）：

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 17:41:31+02
```

兼容性

SQL92

SQL92 中没有通用的 `SET` *`variable`*（除了 `SET TRANSACTION ISOLATION LEVEL` 之外）。SQL92 中 `SET TIME ZONE` 的语法略有不同，只允许用单个整数值指定时区：

```
SET TIME ZONE { interval_value_expression | LOCAL }
```

SHOW

SHOW — 显示会话的运行时参数

大纲

```
SHOW keyword
```

输入

keyword

可用的变量更多信息请参考 SET。

输出

`NOTICE: variable is value`

成功时返回的消息。

`NOTICE: Unrecognized variable value`

如果 → `variable` 不存在则返回此消息。

`NOTICE: Time zone is unknown`

如果没有设置 `TZ` 或 `PGTZ` 环境变量。

描述

`SHOW` 将显示会话期间一个运行时参数的当前设置。

这些变量可以用 `SET` 语句设置，并且可以用 `RESET` 语句恢复为默认值。参数和值不区分大小写。

注解

另见 SET 和 RESET 来操纵变量的值。

用法

显示当前的 `DateStyle` 设置：

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

显示当前的遗传优化器（`geqo`）设置：

```
SHOW GEQO;
NOTICE:  GEQO is ON beginning with 11 relations
```

兼容性

SQL92

SQL92 中没有定义 `SHOW`。

TRUNCATE

TRUNCATE — 清空一个表

大纲

```
TRUNCATE [ TABLE ] name
```

输入

name

要截断的表的名字。

输出

`TRUNCATE`

如果表被成功截断，返回此消息。

描述

`TRUNCATE`快速移除一个表中的所有行。它的效果与不带条件的`DELETE`相同，但由于它并不实际扫描表，因此速度更快。这对大表最为有效。

用法

截断表`bigtable`：

```
TRUNCATE TABLE bigtable;
```

兼容性

SQL92

SQL92中没有`TRUNCATE`。

UNLISTEN

UNLISTEN — 停止监听通知

大纲

```
UNLISTEN { notifyname | * }
```

输入

notifyname

先前注册的通知条件的名称。

*

清除该后端当前的所有监听注册。

输出

→ UNLISTEN

语句已执行的确认（acknowledgement）。

描述

UNLISTEN 用于移除一个现有的NOTIFY注册。UNLISTEN 取消当前 Postgres会话作为通知条件 *notifyname*监听者的任何现有注册。特殊的条件通配符"*"取消当前会话的所有监听者注册。

NOTIFY 中包含关于LISTEN和 NOTIFY用法的更广泛的讨论。

注解

classname 不需要是有效的类名，而可以是任何可作为名称使用的、长度不超过 32 个字符的字符串。

后端不会因为你 UNLISTEN 一个你并没有在监听的东西而抱怨。每个后端退出时都会自动执行UNLISTEN *。

Postgres一些早期版本中，不对应实际表的 *classname*必须用双引号括起来的限制已不存在。

用法

订阅一个现有的注册：

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

一旦执行了 UNLISTEN，后续的 NOTIFY 命令会被忽略：

```
postgres=> UNLISTEN virtual;
UNLISTEN
postgres=> NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

兼容性

SQL92

`UNLISTEN` 在 SQL92 中没有。

UPDATE

UPDATE — 替换表中各列的值

大纲

```
UPDATE table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

输入

table

一个现有表的名称。

column

table 中一个列的名称。

expression

赋给列的一个有效表达式或值。

fromlist

一个 Postgres 非标准扩展，允许来自其他表的列出现在 WHERE 条件中。

condition

WHERE 子句的进一步描述请参阅 SELECT 语句。

输出

`UPDATE #`

成功时返回的消息。# 表示被更新的行数。如果 # 等于 0，则没有行被更新。

描述

`UPDATE` 更改所有满足条件的行中指定列的值。只有要修改的列才需要作为列出现在语句中。

数组引用使用与 SELECT 中相同的语法。也就是说，可以用单次查询替换单个数组元素、数组元素的一个范围或整个数组。

要修改表，你必须拥有对它的写权限，以及对 WHERE 条件中提到其值的任何表的读权限。

用法

把列 kind 上的单词"Drama"改为"Dramatic"：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
SELECT * FROM films WHERE kind = 'Dramatic' OR kind = 'Drama';
```

```
 code  |     title     | did | date_prod  |   kind   | len
-------+---------------+-----+------------+----------+-------
 BL101 | The Third Man | 101 | 1949-12-23 | Dramatic | 01:44
 P_302 | Becket        | 103 | 1964-02-03 | Dramatic | 02:28
 M_401 | War and Peace | 104 | 1967-02-12 | Dramatic | 05:57
 T_601 | Yojimbo       | 106 | 1961-06-16 | Dramatic | 01:50
 DA101 | Das Boot      | 110 | 1981-11-11 | Dramatic | 02:29
```

兼容性

SQL92

SQL92 为定位式 UPDATE 语句定义了一种不同的语法：

```
UPDATE table SET column = expression [, ...]
    WHERE CURRENT OF cursor
```

其中 *cursor* 标识一个打开的游标。

VACUUM

VACUUM — 清理并分析一个 Postgres 数据库

大纲

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

输入

VERBOSE

为每个表打印一份详细的清理活动报告。

ANALYZE

更新优化器用来确定执行查询最有效方式所使用的列统计信息。

table

要清理的指定表的名称。默认为所有表。

column

要分析的指定列的名称。默认为所有列。

输出

→ VACUUM

命令已被接受，数据库正在被清理。

NOTICE: --Relation *table*--

*table*的报告头。

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac
3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail.
 Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.
```

对*table*本身的分析。

```
NOTICE: Index index: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/
0 sec.
```

对目标表上某个索引的分析。

描述

VACUUM在 Postgres 中有两个目的：既是回收存储空间的手段，也是为优化器收集信息的手段。

VACUUM打开数据库中的每个类，清除回滚事务留下的记录，并更新系统目录中的统计信息。
所维护的统计信息包括所有类中存储的元组数和页数。

VACUUM ANALYZE收集表示每列数据 disbursion（离散度）的统计信息。当存在多条可能的查询执行路径时，这些信息很有价值。

周期性地运行VACUUM将提高数据库处理用户查询的速度。

注意

打开的数据库是 VACUUM 的目标。

我们建议对活跃的生产数据库每晚执行一次 VACUUMM，以持续清除过期的行。在把一个大类复制进 Postgres或删除大量记录之后，对受影响的表发出一条VACUUM ANALYZE查询可能是个好主意。这会用所有最近变更的结果更新系统目录，使 Postgres查询优化器在规划用户查询时做出更好的选择。

用法

下面是对回归数据库中的一个表运行VACUUM的例子：

```
regression=> vacuum verbose analyze onek;
NOTICE:  --Relation onek--
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen
 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail.
 Pages 0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 --> 25; Tuple(s) moved: 1000. Elapsed 0/
1 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
VACUUM
```

兼容性

SQL92

SQL92中没有VACUUM语句。

第 20 章 应用程序

这里是 Postgres 应用程序和支持工具的参考信息。

createdb

createdb — 创建一个新的 Postgres 数据库

大纲

```
createdb [ options ] dbname [ description ]
```

输入

-h, --host *host*

指定 postmaster 所在机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U, --username *username*

要连接使用的用户名。

-W, --password

强制提示输入密码。

-e, --echo

回显 createdb 生成并发送给后端的查询。

-q, --quiet

不显示响应。

-D, --location *datadir*

指定本次数据库安装的备选数据库位置。这是安装系统表的位置，
而不是这个具体数据库的位置，后者可能不同。

-E, --encoding *encoding*

指定该数据库使用的字符编码方案。

dbname

指定要创建的数据库的名称。该名称在本安装中的所有 Postgres 数据库中必须唯一。默认创建一个与当前系统用户同名的数据库。

description

可选地指定与新创建的数据库关联的注释。

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql .

输出

`CREATE DATABASE`

数据库创建成功。

`createdb: Database creation failed.`

（一目了然。）

`createdb: Comment creation failed. (Database was created.)`

无法创建数据库的注释/描述。数据库本身已经创建。可以以后用 SQL 命令 `COMMENT ON DATABASE` 创建注释。

If there is an error condition, the backend error message will be displayed. See CREATE DATABASE and psql for possibilities.

描述

createdb 创建一个新的 Postgres 数据库。执行这个命令的用户成为该数据库的所有者。

createdb 是围绕 SQL 命令 CREATE DATABASE 的一个 shell 脚本包装，它通过 Postgres 交互式终端 psql 执行。因此，用这种方法或其他方法创建数据库并没有什么特别之处。这意味着脚本必须能找到 psql，而且目标主机上有数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都适用。

用法

用默认的数据库服务器创建数据库 `demo`：

```
$ createdb demo
CREATE DATABASE
```

其响应与你直接运行 `CREATE DATABASE` SQL 命令所得到的相同。

用主机 eden 上的 postmaster、端口 5000 创建数据库 `demo`，使用 `LATIN1` 编码方案，并看一看底层查询：

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

createlang

createlang — 向 Postgres 数据库添加一种新的编程语言

大纲

```
createlang [ connection options ] [ langname [ dbname ] ]
createlang [ connection options ] --list|-l [ dbname ]
```

输入

createlang 接受下列命令行参数：

langname

指定要定义的后端编程语言的名称。如果命令行上没有指定 *langname*，createlang 会提示输入。

[-d, --dbname] *dbname*

指定应把该语言添加到哪个数据库。

-l, --list

显示目标数据库（必须指定）中已安装语言的列表。

createlang 还接受下列用于连接参数的命令行参数：

-h, --host *host*

指定 postmaster 所在机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U, --username *username*

要用来连接的用户名。

-W, --password

强制提示输入密码。

输出

大多数错误消息都是不言自明的。如果不是，请带 `--echo` 选项运行 createlang，并参见相应的 SQL 命令的详细说明。更多可能的情况也请检查 psql。

描述

createlang 是一个向 Postgres 数据库添加新的编程语言的实用工具。createlang 目前接受两种语言：`plsql` 和 `pltcl`。

虽然可以直接用若干 SQL 命令添加后端编程语言，但建议使用 createlang，因为它会执行若干检查而且使用起来容易得多。更多信息见 CREATE LANGUAGE。

注解

使用 droplang 可移除一个语言。

用法

To install `pltcl`:

```
$ createlang pltcl
```

createuser

createuser — 创建一个新的 Postgres 用户

大纲

```
createuser [ options ] [ username ]
```

输入

-h, --host *host*

指定运行 postmaster 的机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 unix 域套接字文件扩展名。

-e, --echo

显示 createdb 生成并发送给后端的查询。

-q, --quiet

不显示响应。

-d, --createdb

允许新用户创建数据库。

-D, --no-createdb

禁止新用户创建数据库。

-a, --adduser

允许新用户创建其他用户。

-A, --no-adduser

禁止新用户创建其他用户。

-P, --pwprompt

如果给出这个选项，createuser将提示输入新用户的口令。如果你不打算使用口令认证，则不需要它。

-i, --sysid *uid*

允许你为新用户选择一个非默认的用户 id。这并非必需，但有些人喜欢这样。

username

指定要创建的 Postgres 用户的名称。这个名称必须在所有 PostgreSQL用户之间唯一。

如果命令行上没有指定名称和其他缺少的信息，系统会提示你输入。

选项 -h、-p和 -e会按原样传递给 psql。psql的选项 -U和 -W也可用，但在这种场合使用它们可能造成困惑。

输出

CREATE USER

一切正常。

createuser: creation of user "*username*" failed

出了些问题。用户没有被创建。

如果出现错误情况，将显示后端错误消息。参见 CREATE USER 和 psql 了解各种可能。

描述

createuser创建一个新的 Postgres用户。只有 pg_shadow类中usesuper被设置的用户才能创建新的 PostgreSQL用户。

createuser是一个 shell 脚本，它通过 Postgres交互式终端 psql 包装了 SQL命令 CREATE USER。因此，通过这个或其他方法创建用户没有什么特别之处。这意味着脚本必须能找到 psql，并且目标主机上有一个数据库服务器在运行。此外，psql和 libpq前端库可用的任何默认设置和环境变量都会适用。

用法

在默认数据库服务器上创建用户 joe：

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

使用主机 eden 上端口 5000 的 postmaster 创建同一个用户 joe，避开提示并查看底层查询：

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

dropdb

dropdb — 移除一个现有的 Postgres 数据库

大纲

```
dropdb [ options ] dbname
```

输入

-h, --host *host*

指定 postmaster 正在运行的机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展。

-U, --username *username*

要连接的用户名。

-W, --password

强制口令提示。

-e, --echo

回显 dropdb 生成并发送给后端的查询。

-q, --quiet

不显示响应。

-i, --interactive

在做任何破坏性操作之前发出一个确认提示。

dbname

指定要移除的数据库的名称。该数据库必须是这个安装中现有的 Postgres 数据库之一。

选项 `-h`、`-p`、`-U`、`-W` 和 `-e` 会被原样传递给 psql。

输出

`DROP DATABASE`

数据库已被成功移除。

`dropdb: Database removal failed.`

某些地方出了问题。

如果出现错误情况，将显示后端错误消息。各种可能性请参阅 DROP DATABASE 和 psql。

描述

dropdb 销毁一个现有的 Postgres 数据库。执行这个命令的用户必须是数据库超级用户或该数据库的拥有者。

dropdb 是 SQL 命令 DROP DATABASE 的一个 shell 脚本包装，它通过 Postgres 交互式终端 psql 进行。因此，通过这个或其他方法删除数据库没有什么特别的。这意味着脚本必须能找到 psql，并且目标主机上有一个数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都适用。

用法

在默认数据库服务器上销毁数据库 demo：

```
$ dropdb demo
DROP DATABASE
```

使用主机 eden 上端口 5000 的 postmaster 销毁数据库 demo，带确认并看一眼底层查询：

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

droplang

droplang — 从 Postgres 数据库移除一种编程语言

大纲

```
droplang [ connection options ] [ langname [ dbname ] ]
droplang [ connection options ] --list|-l
```

输入

droplang 接受下列命令行参数：

langname

指定要移除的后端编程语言的名称。如果命令行上没有指定 *langname*，droplang 会提示输入。

[-d, --dbname] *dbname*

指定应从哪个数据库移除该语言。

-l, --list

显示目标数据库（必须指定）中已安装语言的列表。

droplang 还接受下列用于连接参数的命令行参数：

-h, --host *host*

指定 postmaster 所在机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U, --username *username*

要连接使用的用户名。

-W, --password

强制提示输入密码。

输出

大多数错误消息都是不言自明的。如果不是，请带 `--echo` 选项运行 droplang，并参见相应的 SQL 命令的详细说明。更多可能的情况也请检查 psql。

描述

droplang 是一个从 Postgres 数据库移除现有的编程语言的实用工具。droplang 目前接受两种语言：`plsql` 和 `pltcl`。

虽然后端编程语言可以直接用几条SQL命令移除，但建议使用droplang，
因为它会执行一些检查，而且用起来容易得多。更多信息见 DROP LANGUAGE。

注解

使用 createlang 可添加一个语言。

用法

要移除 `pltcl`：

```
$ droplang pltcl
```

dropuser

dropuser — 删除（移除）一个 Postgres 用户

大纲

```
dropuser [ options ] [ username ]
```

输入

-h, --host *host*

指定 postmaster 所在机器的主机名。

-p, --port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-e, --echo

回显 createdb 生成并发送给后端的查询。

-q, --quiet

不显示响应。

-i, --interactive

在实际移除用户之前提示确认。

username

指定要删除的 Postgres 用户的名称。该名称必须存在于 Postgres 安装中。如果命令行上没有指定名称，会提示你输入一个。

选项 `-h`、`-p` 和 `-e` 被原样传递给 psql。psql 的选项 `-U` 和 `-W` 也可用，但在此上下文中可能造成混淆。

输出

`DROP USER`

一切正常。

`dropuser: deletion of user "`*username*`" failed`

出了问题。用户没有被删除。

If there is an error condition, the backend error message will be displayed. See DROP USER and psql for possibilities.

描述

dropuser 移除一个现有的 Postgres 用户以及该用户所拥有的数据库。只有 `pg_shadow` 类中 `usesuper` 被设置的用户才能删除 Postgres 用户。

dropuser 是围绕 SQL 命令 DROP USER 的一个 shell 脚本包装，它通过 Postgres 交互式终端 psql 执行。因此，用这种方法或其他方法删除用户并没有什么特别之处。这意味着脚本必须能找到 psql，而且目标主机上有数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都适用。

用法

从默认的数据库服务器删除用户 `joe`：

```
$ dropuser joe
DROP USER
```

用主机 eden 上的 postmaster、端口 5000 删除用户 `joe`，带验证并看一看底层查询：

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

```
ecpg [ -v ] [ -t ] [ -I include-path ] [ -o outfile ]  file1 [ file2 ]
 [ ... ]
```

输入

ecpg接受下列命令行参数：

-v

打印版本信息。

-t

关闭自动事务（auto-transactin）模式。

-I *path*

指定一个额外的 include 路径。默认值是.、/usr/local/include、编译时定义的 Postgres include 路径（默认：/usr/local/pgsql/lib）以及 /usr/include。

-o

指定 ecpg 应将全部输出写入 outfile。如果没有给出这个选项，输出将写入 *name*.c（假定输入文件名为 *name*.pgc）。如果输入文件确实没有预期的 .pgc后缀，那么输出文件名将在输入文件名后追加上 .pgc。

file

要处理的文件。

输出

ecpg会创建一个文件或写入 stdout。

return value

ecpg 成功完成时向 shell 返回 0，出错时返回 -1。

描述

ecpg 是 C 语言和 Postgres的嵌入式 SQL 预处理器。它支持开发带有嵌入式 SQL 代码的 C 程序。

Linus Tolke[1]是 ecpg的原作者（直到 0.2 版）。Michael Meskes[2] 是ecpg目前的作者和维护者。Thomas Good[3] 是本文件所依据的 ecpg man 页最后一版的作者。

[1]linus@epact.se
[2]meskes@debian.org

用法

为编译做预处理

嵌入式 SQL 源文件在编译之前必须先做预处理：

```
ecpg [ -d ] [ -o file ] file.pgc
```

其中可选的 `-d` 标志打开调试。`.pgc` 扩展名是表示 ecpg 源的一种随意约定。

你可能想把预处理器的输出重定向到一个日志文件。

编译和链接

假定 Postgres 的可执行文件位于 `/usr/local/pgsql`，你需要这样编译和链接预处理后的源文件：

```
gcc -g -I /usr/local/pgsql/include [ -o file ] file.c -L /usr/local/
pgsql/lib -lecpg -lpq
```

语法

库

预处理器会在源文件开头加上两条指令：

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

变量声明

在 ecpg 源代码中声明的变量必须以以下内容开头：

```
EXEC SQL BEGIN DECLARE SECTION;
```

类似地，变量声明节必须以以下内容结束：

```
EXEC SQL END DECLARE SECTION;
```

> **注意**
>
> 在 2.1.0 版之前，每个变量必须单独一行声明。从 2.1.0 版起，多个变量可以在一行中声明：
>
> ```
> char  foo(16), bar(16);
> ```

[3] tomg@q8.nrnet.org

错误处理

SQL 通信区用以下方式定义：

```
EXEC SQL INCLUDE sqlca;
```

> **注意**
>
> sqlca是小写的。虽然可以遵循 SQL 惯例（即用大写来区分嵌入式 SQL 与 C 语句），但 sqlca（它包含 sqlca.h 头文件）MUST 是小写的。这是因为 EXEC SQL 前缀表明这个 INCLUDE 将由 ecpg 解析。ecpg 区分大小写（SQLCA.h 将找不到。）EXEC SQL INCLUDE 也可以用来包含其他头文件，只要遵守大小写敏感的规则。

sqlprint 命令与 EXEC SQL WHENEVER 语句一起使用，在整个程序中打开错误处理：

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

和

```
EXEC SQL WHENEVER not found sqlprint;
```

> **注意**
>
> 这不是EXEC SQL WHENEVER语句用法的一个详尽示例。更多用法示例可以在 SQL 手册中找到（例如 Groff 和 Weinberg 的`The LAN TIMES Guide to SQL'）。

连接到数据库服务器

使用下面的语句连接数据库：

```
EXEC SQL CONNECT dbname;
```

其中数据库名不带引号。在 2.1.0 版之前，数据库名必须放在单引号中。

也可以在 connect 语句中指定服务器名和端口名。语法是：

```
dbname[@server][:port]
```

或者

```
<tcp|unix>:postgresql://server[:port][/dbname][?options]
```

查询

一般来说，其他应用（如 psql）可接受的 SQL 查询都可以嵌入到你的 C 代码中。下面是一些示例。

创建表：

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
```

```
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

插入：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

单行 SELECT：

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

使用游标的 SELECT：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

注意

没有 `EXEC SQL PREPARE` 语句。

完整的结构定义 MUST 列在 declare 节内。

更多尚未实现的功能请看源码中的 `TODO` 文件。

pgaccess

pgaccess — Postgres 图形交互客户端

大纲

```
pgaccess [ dbname ]
```

输入

dbname

要访问的一个现有数据库的名称。

输出

描述

pgaccess 为 Postgres 提供一个图形界面，你可以在其中管理你的表、编辑它们、定义查询、序列和函数。

通过 tcl 访问 Postgres 的另一种方式是使用 pgtclsh 或 pgtksh。

pgaccess 可以：

- 在指定主机的指定端口上打开任何数据库，使用指定的用户名和口令。
- 执行 VACUUM。
- 把首选项保存在 `~/.pgaccessrc` 文件中。

对于表，pgaccess 可以：

- 打开多个表进行查看，最大 n 条记录（可配置）。
- 通过拖动垂直网格线调整列宽。
- 在单元格中折行显示文本。
- 编辑时动态调整行高。
- 为每个表保存表布局。
- 导入/导出到外部文件（SDF,CSV）。
- 使用过滤功能；输入像 price>3.14 这样的过滤器。
- 指定排序顺序；手动输入排序字段。
- 就地编辑；双击你想要更改的文本。
- 删除记录；指向该记录，按 Del 键。
- 添加新记录；用右键点击保存新行。
- 用向导创建表。
- 重命名和删除（drop）表。
- 检索表的信息，包括拥有者、字段信息、索引。

对于查询，pgaccess 可以：

- 定义、编辑和存储用户定义查询。
- 保存视图布局。
- 把查询存储为视图。

- 带可选的用户输入参数执行；例如

```
select * from invoices where year=[parameter "Year of selection"]
```

- 查看任何 select 查询的结果。
- 运行动作查询（insert、update、delete）。
- 使用带拖放支持和表别名的可视化查询构建器构造查询。

对于序列，pgaccess 可以：

- 定义新实例。
- 检查现有实例。
- 删除。

对于视图，pgaccess 可以：

- 通过把查询保存为视图来定义它们。
- 查看它们，带过滤和排序功能。
- 设计新视图。
- 删除（drop）现有视图。

对于函数，pgaccess 可以：

- 定义。
- 检查。
- 删除。

对于报表，pgaccess 可以：

- 从表生成简单报表（beta 阶段）。
- 更改字段和标签的字体、大小和样式。
- 从数据库装载和保存报表。
- 预览表、postscript 打印样例。

对于表单，pgaccess 可以：

- 打开用户定义表单。
- 使用表单设计模块。
- 使用查询部件访问记录集。

对于脚本，pgaccess 可以：

- 定义。
- 修改。
- 调用用户定义的脚本。

pgadmin

pgadmin — Postgres 的 Windows 95/98/NT 数据库管理与设计工具

大纲

```
pgadmin [ datasourcename [ username [ password ] ] ]
```

输入

datasourcename

一个已有的Postgres ODBC 系统数据源或用户数据源的名字。

username

指定的*datasourcename* 的一个有效用户名。

password

指定的*datasourcename*和 *username* 的一个有效口令。

输出

描述

pgadmin是一个用于设计、维护和管理 Postgres数据库的通用工具。它运行在 Windows 95/98 和 NT 之下。

它的特性包括：

- 任意 SQL 输入。
- 面向数据库、表、索引、序列、视图、触发器、函数和语言的信息浏览器和"创建器"。
- 用户、组和权限配置对话框。
- 带升级脚本生成的版本跟踪。
- Microsoft MSysConf 表的配置。
- 数据导入和导出向导。
- 数据库迁移向导。
- 面向数据库、表、索引、序列、语言和视图的预定义报表。

pgadmin与Postgres 分开发布，可以从 http://www.pgadmin.freeserve.co.uk 下载。

pg_ctl

pg_ctl — 启动、停止和重启 postmaster

大纲

```
pg_ctl [-w] [-D datadir][-p path] [-o "options"] start
pg_ctl [-w] [-D datadir] [-m [s[mart]|f[ast]|i[mmediate]]] stop
pg_ctl [-w] [-D datadir] [-m [s[mart]|f[ast]|i[mmediate]]
    [-o "options"] restart
pg_ctl [-D datadir] status
```

输入

-w

等待数据库服务器启动，方式是监视 pid 文件（PGDATA/postmaster.pid）的创建。60 秒后超时。

-D *datadir*

指定这个数据库安装的数据库位置。

-p *path*

指定 postmaster 可执行文件的路径。

-o "*options*"

指定直接传递给 postmaster 的选项。

参数通常用单引号或双引号括起来，以确保它们作为一个整体传递。

-m *mode*

指定关闭模式。

smart
s

smart 模式等待所有客户端退出登录。这是默认值。

f[ast]
f

Fast 模式向后端发送 SIGTERM，这意味着活动中的事务会被回滚。

immediate
i

Immediate 模式向后端发送 SIGUSR1 并让它们中止。在这种情况下，下次启动时将需要进行数据库恢复。

start

启动 postmaster。

stop

关闭 postmaster。

restart

重启 postmaster，执行一个停止/启动序列。

status

显示 postmaster 的当前状态。

输出

`pg_ctl: postmaster is *state* (pid: #)`

Postmaster 状态。

如果出现错误条件，将显示后端的错误消息。

描述

pg_ctl 是一个用于启动、停止或重启 postmaster 的工具。

用法

启动 postmaster

启动 postmaster：

```
> pg_ctl start
```

如果提供了 -w，pg_ctl 将等待数据库服务器启动，方式是监视 pid 文件（PGDATA/postmaster.pid）的创建，最多 60 秒。

调用 postmaster 的参数取自下列来源：

- postmaster 的路径：在命令搜索路径中寻找。
- 数据库目录：`PGDATA` 环境变量。
- 其他参数：`PGDATA/postmaster.opts.default`。

`postmaster.opts.default` 包含 postmaster 的参数。

注意 `postmaster.opts.default` 是由 initdb 从 Postgres 安装目录下的 `lib/postmaster.opts.default.sample` 安装的（`lib/postmaster.opts.default.sample` 是在安装 Postgres 时从 `src/bin/pg_ctl/postmaster.opts.default.sample` 复制而来的）。

要覆盖默认参数，你可以使用 `-D`、`-p` 和 `-o` 选项。

一个启动 postmaster 并阻塞到 postmaster 启动为止的例子：

```
> pg_ctl -w start
```

要指定 postmaster 二进制文件的路径，试试：

```
> pg_ctl -p /usr/local/pgsq/bin/postmaster start
```

对于一个使用端口 5433 并且不带 `fsync` 运行的 postmaster，使用：

```
> pg_ctl -o "-o -F -p 5433" start
```

停止 postmaster

```
> pg_ctl stop
```

停止 postmaster。使用 `-m` 开关可以控制后端如何关闭。`-w` 等待 postmaster 关闭。`-m` 指定关闭模式。

重启 postmaster

这几乎等价于先停止 postmaster 再重新启动它，区别只是停止之前使用的参数也会被使用。这是通过把参数保存在 $PGDATA/postmaster.opts 文件中实现的。`-w`、`-D`、`-m`、`-fast`、`-immediate` 和 `-o` 也可以用于重启模式，它们的含义与上面描述的相同。

以最简单的形式重启 postmaster：

```
> pg_ctl restart
```

重启 postmaster，等待它关闭并启动：

```
> pg_ctl -w restart
```

重启后使用端口 5433 并禁用 fsync：

```
> pg_ctl -o "-o -F -p 5433" restart
```

postmaster 状态

从 postmaster 获取状态信息：

```
> pg_ctl status
```

下面是 pg_ctl 的一个输出示例：

```
pg_ctl: postmaster is running (pid: 13718)
options are:
/usr/local/src/pgsql/current/bin/postmaster
-p 5433
-D /usr/local/src/pgsql/current/data
-B 64
-b /usr/local/src/pgsql/current/bin/postgres
-N 32
```

```
-o '-F'
```

pg_dump

pg_dump — Extract a Postgres database into a script file

大纲

```
pg_dump [ dbname ]
pg_dump [ -h host ] [ -p port ]
     [ -t table ]
     [ -a ] [ -c ] [ -d ] [ -D ] [ -i ] [ -n ] [ -N ]
     [ -o ] [ -s ] [ -u ] [ -v ] [ -x ]
     [ dbname ]
```

输入

pg_dump 接受下列命令行参数：

dbname

指定要抽取的数据库名称。*dbname* 默认为 USER 环境变量的值。

-a

只转储数据，不转储模式（定义）。

-c

创建之前先清除（删除）模式。

-d

将数据转储为正规的插入字符串。

-D

将数据转储为带属性名的插入

-i

忽略 pg_dump 与数据库服务器之间的版本不匹配。由于 pg_dump 对系统目录了解甚多，任何给定版本的 pg_dump 都只打算与相应版本的数据库服务器一起使用。如果你需要忽略版本检查，可以使用此选项（如果 pg_dump 因此而失败，可别怪没警告过你）。

-n

除非绝对必要，抑制标识符周围的双引号。如果标识符中使用了保留字，
这可能导致装载这份转储数据时出问题。这是 pg_dump v6.4 之前版本的默认行为。

-N

在标识符周围包含双引号。这是默认行为。

-o

为每个表转储对象标识符（OID）。

-s

只转储模式（定义），不转储数据。

-t *table*

只转储 *table* 的数据。

-u

使用密码认证。提示输入用户名和密码。

-v

指定详细模式

-x

不转储 ACL（grant/revoke 命令）和表所有权信息。

pg_dump 还接受下列用于连接参数的命令行参数：

-h *host*

指定运行 postmaster 的机器的主机名。默认使用本地 Unix 域套接字而不是 IP 连接。。

-p *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口，或本地 Unix 域套接字文件扩展名。端口号默认为 5432，或 PGPORT 环境变量的值（如果已设置）。

-u

使用密码认证。提示输入 *username* 和 *password*。

输出

pg_dump 会创建一个文件或写入 stdout。

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'?

pg_dump 无法连接到指定主机和端口上的 postmaster 进程。如果你看到这条消息，请确保 postmaster 运行在正确的主机上，并且你指定了正确的端口。如果你的站点使用认证系统，请确保你已获得所需的认证凭据。

Connection to database '*dbname*' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow'

你在关系 pg_shadow 中没有有效的条目，将不被允许访问 Postgres。请联系你的 Postgres 管理员。

dumpSequence(*table*): SELECT failed

你没有读取该数据库的权限。请联系你的 Postgres 站点管理员。

> **注意**
>
> pg_dump 内部执行 SELECT 语句。如果在运行 pg_dump 时遇到问题，请确保你能够使用例如 psql 从数据库中选择信息。

描述

pg_dump 是一个把 Postgres 数据库转储为包含查询命令的脚本文件的工具。脚本文件为文本格式，可用于重建数据库，甚至是在其他机器和其他架构上。pg_dump 会产生重新生成所有用户定义类型、函数、表、索引、聚合和操作符所需的查询。此外，
所有数据都以文本格式复制出来，因此既可以方便地重新复制进去，也可以导入编辑工具。

pg_dump 可用于转储数据库的内容，以便从一个 Postgres 安装迁移到另一个安装。运行 pg_dump 之后，应当检查输出脚本文件中是否有任何警告，尤其是考虑到下面列出的限制。

注意

pg_dump 有一些限制。这些限制大多源于难以从系统目录中抽取某些元信息。

- pg_dump 不理解部分索引。原因同上；部分索引谓词是以计划形式存储的。
- pg_dump 不处理大对象。大对象会被忽略，必须手工处理。
- 在进行只转储数据的转储时，pg_dump 会在插入数据之前发出禁用用户表上触发器的查询，并在数据插入完成后发出重新启用它们的查询。如果恢复在中途中止，
  系统目录可能会处于错误状态。

用法

要转储一个与用户同名的数据库：

```
% pg_dump > db.out
```

要重新装载这个数据库：

```
% psql -e database < db.out
```

pg_dumpall

pg_dumpall — 把所有 Postgres 数据库抽取到一个脚本文件

大纲

```
pg_dumpall
pg_dumpall [ -h host ] [ -p port ] [ -a ] [ -d ] [ -D ] [ -O ] [ -s ]
 [ -u ] [ -v ] [ -x ]
```

输入

pg_dumpall 接受下列命令行参数：

-a

只转储数据，不转储模式（定义）。

-d

把数据转储为正确的插入字符串。

-D

把数据转储为带属性名的插入

-n

除非绝对必要，抑制标识符两边的双引号。如果有保留字被用作标识符，
加载这些转储数据时可能出问题。

-o

转储每个表的对象标识符（OID）。

-s

只转储模式（定义），不转储数据。

-u

使用口令认证。提示输入用户名和口令。

-v

指定详细模式

-x

不转储 ACL（grant/revoke 命令）和表所有权信息。

pg_dumpall 还接受下列用于连接参数的命令行参数：

-h *host*

指定 postmaster 所在机器的主机名。默认使用本地 Unix 域套接字而不是 IP 连接。

-p *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。端口号默认为 5432，或 PGPORT 环境变量的值（如果设置了的话）。

-u

使用口令认证。提示输入 *username* 和 *password*。

输出

pg_dumpall 会创建一个文件或写到 stdout。

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'?

pg_dumpall could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database '*dbname*' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow'

你在关系 pg_shadow 中没有有效的条目，将不被允许访问 Postgres。请联系你的 Postgres 管理员。

dumpSequence(*table*): SELECT failed

你没有读取数据库的权限。请联系你的 Postgres 站点管理员。

注意

pg_dumpall 在内部执行 SELECT 语句。如果运行 pg_dumpall 遇到问题，请确保你能用例如 psql 从数据库选择信息。

描述

pg_dumpall 是一个把所有 Postgres 数据库转储到一个文件的实用工具。它还转储对所有数据库全局的 pg_shadow 表。pg_dumpall 在这个文件中包含在加载之前自动创建每个被转储数据库的相应命令。

pg_dumpall 接受所有 pg_dump 选项，但 -f、-t 和 *dbname* 应当省略。

关于这一功能的更多信息请参阅 pg_dump。

用法

转储所有数据库：

```
% pg_dumpall > db.out
```

提示

可以对 pg_dumpall 使用大多数 pg_dump 选项。

重新装载这个数据库：

```
% psql -e template1 < db.out
```

提示

重新加载时可以使用大多数 psql 选项。

psql

psql — Postgres 的交互式终端

大纲

```
psql [ options ] [ dbname [ user ] ]
```

概要

psql 是 Postgres 的一个基于终端的前端。它使你能够交互式地输入查询，将其发送给 Postgres，并查看查询结果。也可以从文件提供输入。此外，它还提供了若干元命令和多种类似 shell 的特性，以便于编写脚本和自动化执行各种任务。

描述

连接到数据库

psql 是一个常规的 Postgres 客户端应用程序。In order to connect to a database you need to know the name of your target database, the hostname and port number of the server and what user name you want to connect as. psql 可以通过命令行选项（即分别为 -d、-h、-p 和 -U）把这些参数告知它。如果遇到一个不属于任何选项的参数，它将被解释为数据库名（如果数据库名已经给出，则解释为用户名）。并非所有这些选项都是必需的，会应用默认值。If you omit the host name psql will connect via a UNIX domain socket to a server on the local host.
默认端口号在编译时确定。由于数据库服务器使用相同的默认值，因此在大多数情况下不必指定端口。默认用户名是你的 Unix 用户名，默认数据库名也是如此。请注意，你不能随意以任意用户名连接到任意数据库。数据库管理员应当已经告知你拥有的访问权限。为了少打一些字，
你还可以把环境变量 PGDATABASE、PGHOST、PGPORT 和 PGUSER 设置为适当的值。

如果由于任何原因（如权限不足、postmaster 未在服务器上运行等）无法建立连接，psql 将返回错误并终止。

输入查询

在正常操作时，psql 提供一个提示符，它是 psql 当前连接到的数据库名称后跟字符串 "=>". For example,

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

在提示符下，用户可以输入 SQL 查询。通常，当遇到表示查询结束的分号时，输入行被发送到后端。行尾并不会终止一条查询！因此为了清晰，查询可以分布在多行上。
如果查询被发送且没有错误，查询的结果会显示在屏幕上。

每当执行一条查询时，psql 还会轮询由 LISTEN and NOTIFY .

psql 元命令

你输入到 psql 中的任何以未加引号的反斜线开始的内容都是一个由 psql 自身处理的 psql 元命令。这些命令正是让 psql 在管理和编写脚本方面很有用的原因。元命令更常被称为斜线或反斜线命令。

psql命令的格式是用反斜线后面直接跟上一个命令动词，然后是一些参数。
参数与命令动词和其他参数之间用任意多个空白字符分隔开。

要在参数中包含空白，必须用单引号把它括起来。要在这类参数中包含单引号，
可以在它前面加一个反斜线。单引号中的内容还会进一步做类 C 的替换，即 \n（换行）、\t（制表符）、*digits*, \0*digits*, and \0x*digits* (the character with the given decimal, octal, or hexadecimal code).

如果未加引号的参数以冒号（:）开头，它会被当作一个变量，该变量的值将被用作参数。

用反引号（`）括起来的参数被当作传递给 shell 的命令行。命令的输出（去掉末尾的换行符）作为参数值。上述转义序列在反引号中也适用。

有些命令接受一个 SQL 标识符（如表名）作为参数。这些参数遵循 SQL 关于双引号的语法规则：
不带双引号的标识符被强制转换为小写。对于所有其他命令，双引号没有特殊含义，
会成为参数的一部分。

当遇到另一个未加引号的反斜线时，参数解析停止。这被当作新元命令的开始。特殊序列 \\（两个反斜线）标志参数的结束并继续解析 SQL 查询（如果有）。这样 SQL 和 psql 命令就可以在一行中自由混用。但无论如何，元命令的参数不能延续到行尾之后。

定义了下列元命令：

\a

如果当前表格输出格式是非对齐，则切换为对齐；否则设置为非对齐。
保留这条命令是为了向后兼容。通用的解决办法见 \pset。

\C [*title*]

设置作为查询结果打印的任何表的标题，或取消任何这样的标题。此命令等价于 \pset title *title*. (The name of this command derives from “caption”, as it was previously only used to set the caption in an HTML table.)

\connect (or \c) [*dbname* [*username*]]

建立到一个新数据库和/或以一个用户名的连接。之前的连接被关闭。If *dbname* is - the current database name is assumed.

如果省略 username，则假定为当前用户名。

作为一个特殊规则，不带任何参数的 \connect 将以默认用户连接到默认数据库（就像你不带任何参数启动 psql 时那样）。

如果连接尝试失败（用户名错误、访问被拒绝等），当且仅当 psql 处于交互模式时才会保留之前的连接。执行非交互式脚本时，处理将立即停止并报错。
选择这种区别一方面是为了防止用户打字错误提供便利，另一方面是一种安全机制，
防止脚本意外地在错误的数据库上操作。

\copy*table* [with oids] { from | to } *filename* | stdin | stdout [with delimiters '*characters*'] [with null as '*string*']

执行前端（客户端）复制。此操作执行的是一条 SQL COPY 命令，但不是由后端读取或写入指定的文件（后者因此需要后端访问权限和特殊的用户权限，
并且受限于后端可访问的文件系统），而是由 psql 读取或写入该文件，并在后端和本地文件系统之间传送数据。

该命令的语法类似于 SQL COPY 命令（细节见其描述）。注意，因此对 \copy 命令适用特殊的解析规则。特别是，变量替换规则和反斜线转义不适用。

提示

此操作不如 SQL COPY 命令高效，因为所有数据都必须通过客户端/服务器的 IP 或套接字连接。对于大量数据，另一种技术可能更可取。

注意

注意前端复制与后端复制对 stdin 和 stdout 的解释差异：在前端复制中它们总是指 psql 的输入和输出流。而在后端复制中 stdin 来自 COPY 自身的来源（例如用 -f 选项运行的脚本），stdout 指查询输出流（见下文的 \o 元命令）。

\copyright

显示 Postgres 的版权和分发条款。

\d*relation*

显示关系的所有列（关系可以是表、视图、索引或序列）、
它们的类型以及任何特殊属性（如 NOT NULL 或默认值，如果有）。如果该关系实际上是表，任何已定义的索引也会列出。如果该关系是视图，则还显示视图定义。

命令形式 \d+ 相同，但还会显示与表列关联的任何注释。

注意

如果 \d 不带任何参数调用，它等价于 \dtvs，后者会显示所有表、视图和序列的列表。这纯粹是为了方便。

\da [*pattern*]

列出所有可用的聚合函数及其操作的数据类型。如果指定了 *pattern*（正则表达式），则只显示匹配的聚合。

\dd [*object*]

显示对象（可以是一个正则表达式）的描述，如果没有给出参数则显示所有对象的描述。
(“Object” covers aggregates, functions, operators, types, relations (tables, views, indices, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
              Object descriptions
  Name   |   What   |         Description
---------+----------+-----------------------------
 version | function | PostgreSQL version string
(1 row)
```

对象的描述可以用 COMMENT ON SQL 命令生成。

> **注意**
>
> Postgres 把对象描述存储在 pg_description 系统表中。

`\df [ pattern ]`

列出可用的函数及其参数和返回类型。If *pattern* (a regular expression) is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description is shown.

`\distvS [ pattern ]`

这不是实际的命令名：字母 i、s、t、v、S 分别代表索引、序列、表、视图和系统表。你可以按任意顺序指定其中任意个或全部，来获得它们的列表及其所有者。

如果指定了 *pattern*，它是一个正则表达式，把列表限制为名称匹配的那些对象。如果在命令名后附加“+”，每个对象都会连同其关联的描述（如果有）一起列出。

`\dl`

这是`\lo_list`的别名，用于显示大对象列表。

`\do [ name ]`

列出可用的操作符及其操作数和返回类型。如果指定了 *name*，则只显示具有该名称的操作符。

`\dp [ pattern ]`

这是 \z 的别名，因它更具助记性而保留（display permissions，显示权限）。

`\dT [ pattern ]`

列出所有数据类型，或者只列出匹配*pattern*的类型。命令形式`\dT+`显示额外的信息。

`\edit` (or `\e`) [*filename*]

如果指定了 filename，则编辑该文件；编辑器退出后，其内容被复制回查询缓冲区。如果没有给出参数，则把当前查询缓冲区复制到一个临时文件，然后以相同的方式编辑。

新的查询缓冲区随后按照 psql 的正常规则重新解析，即把整个缓冲区当作单行对待。(Thus you cannot make “scripts” this way, use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

> **提示**
>
> psql 依次搜索环境变量 PSQL_EDITOR、EDITOR 和 VISUAL 来确定要使用的编辑器。如果它们都未设置，则运行 /bin/vi。

\echo*text* [...]

把参数打印到标准输出，参数之间以一个空格分隔并后跟一个换行符。
这在脚本的输出中穿插信息时很有用。例如：

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted -n the the trailing newline is not written.

> **提示**
>
> 如果你使用 \o 命令重定向了查询输出，可能希望用 \qecho 代替此命令。

\encoding [*encoding*]

如果使用多字节编码，则设置客户端编码。不带参数时，此命令显示当前编码。

\f [*string*]

设置非对齐查询输出的字段分隔符。默认是“|”（竖线符号）。
设置输出选项的通用方法另见 \pset。

\g [{ *filename* | |*command* }]

把当前查询输入缓冲区发送到后端，并可选地把输出保存到 filename 中，或把输出通过管道送到一个单独的 Unix shell 去执行 command。A bare \g is virtually equivalent to a semicolon. A \g with argument is a “one-shot” alternative to the \o command.

\help (or \h) [*command*]

给出指定 SQL 命令的语法帮助。如果未指定 command，psql 将列出所有可获得语法帮助的命令。如果 command 是星号（*），则显示所有 SQL 命令的语法帮助。

> **注意**
>
> 为简化输入，由多个词组成的命令不必加引号。因此输入 \help alter table 就可以。

\H

打开HTML查询输出格式。如果HTML格式已经打开，则切换回默认的对齐文本格式。
此命令是为兼容性和便利性而保留的；设置其他输出选项的方法见\pset。

\i*filename*

从文件 filename 读取输入并执行，就像从键盘输入一样。

> **注意**
>
> 如果想在读入时在屏幕上看到这些行，必须把变量 ECHO 设置为 all。

\l (or \list)

列出服务器中的所有数据库及其所有者。向命令名追加 + 还可以看到数据库的任何描述。如果你的 Postgres 安装编译时带有多字节编码支持，则还会显示每个数据库的编码方案。

\lo_export*loidfilename*

从数据库读取 OID 为 loid 的大对象并把它写入 filename。注意这条命令与服务器端函数 lo_export 有微妙的差别，后者以数据库服务器运行用户的权限在服务器的文件系统上操作。

> **提示**
>
> 使用 \lo_list 可以查出大对象的 OID。

> **注意**
>
> 有关所有大对象操作的重要信息，参见 LO_TRANSACTION 变量的描述。

\lo_import*filename* [*comment*]

把该文件存储为一个 Postgres 大对象。可以选择把给定的注释与该对象关联。Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the \lo_list command.

注意此命令与服务器端的 lo_import 有细微差别，因为它以本地用户身份在本地文件系统上操作，而不是服务器的用户和文件系统。

> **注意**
>
> 有关所有大对象操作的重要信息，参见 LO_TRANSACTION 变量的描述。

\lo_list

显示当前存储在数据库中的所有 Postgres 大对象的列表及其所有者。

`\lo_unlink`*loid*

从数据库删除 OID 为 loid 的大对象。

> **提示**
>
> 使用 \lo_list 可以查出大对象的 OID。

> **注意**
>
> 有关所有大对象操作的重要信息，参见 LO_TRANSACTION 变量的描述。

`\o` [{*filename* | `|` *command*}]

把以后的查询结果保存到文件 filename 中，或把以后的结果通过管道送到一个单独的 Unix shell 去执行 command。If no arguments are specified, the query output will be reset to `stdout`.

“Query results” 包括从数据库服务器获得的所有表、命令响应和通知，以及查询数据库的各种反斜线命令（如 \d）的输出，但不包括错误消息。

> **提示**
>
> 要在查询结果之间穿插文本输出，使用 \qecho。

`\p`

将当前查询缓冲区打印到标准输出。

`\pset`*parameter* [*value*]

此命令设置影响查询结果表输出的选项。*parameter* 描述要设置哪个选项。The semantics of *value* depend thereon.

可调整的打印选项有：

`format`

把输出格式设置为 unaligned、aligned、html 或 latex 之一。允许唯一的缩写。（也就是说一个字母就足够了。）

“Unaligned” 把一个元组的所有字段写在一行上，用当前活动的字段分隔符分隔。这用于创建可能要被其他程序读入的输出（制表符分隔、逗号分隔）。“Aligned” 模式是默认的标准、人类可读、格式良好的文本输出。The “HTML” and “LaTeX” 模式输出打算用相应标记语言包含进文档中的表。它们不是完整的文档！(This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

第二个参数必须是一个数字。一般来说，数字越大表格的边框和线越多，但这取决于具体的格式。In HTML mode, this will translate directly into the `border=`

. . . attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

在常规格式和扩展格式之间切换。启用扩展格式时，所有输出有两列，字段名在左、数据在右。如果数据在正常的“水平”模式下无法在屏幕上显示，这种模式会很有用。

所有四种输出格式都支持扩展模式。

`null`

第二个参数是一个每当字段为空值时应打印的字符串。默认是完全不打印任何内容，这很容易被误认为例如空字符串。因此，可以选择写 `\pset null '(null)'`.

`fieldsep`

指定在非对齐输出模式中使用的字段分隔符。
这样就可以创建例如以制表符或逗号分隔的输出，其他程序可能更喜欢这样。
要把制表符设置为字段分隔符，输入 `\pset fieldsep '\t'`。默认的字段分隔符是 `'|'`（竖线符号）。

`recordsep`

指定在非对齐输出模式中使用的记录（行）分隔符。默认是换行符。

`tuples_only` (or `t`)

在只显示元组和完整显示之间切换。完整显示可能显示额外的信息，如列标题、标题和各种脚注。在只显示元组模式下，只显示实际的表数据。

`title` [*`text`*]

为随后打印的任何表设置表标题。这可用于为输出提供描述性标签。如果不带参数，则取消标题。

> **注意**
>
> 此选项以前只影响 HTML 模式。现在可以在任何输出格式中设置标题。

`tableattr` (or `T`) [*`text`*]

允许你指定要放在 HTML table 标签内的任何属性。例如可以是 cellpadding 或 bgcolor。注意这里可能不应该指定 border，因为那已由 \pset border 处理。

`pager`

切换分页器的使用以输出表格。如果设置了环境变量 PAGER，输出将通过管道送到指定的程序。Otherwise `more` is used.

无论如何，psql 只在看起来合适时才使用分页器。这尤其意味着输出到终端且该表通常放不到屏幕上。由于打印例程的模块化特性，并不总是能预测实际会打印的行数。因此 psql 对何时使用分页器、何时不使用可能显得不太挑剔。

Illustrations on how these different formats look can be seen in the 示例 section.

> **提示**
>
> \pset 有多种快捷命令。参见 \a、\C、\H、\t、\T 和 \x。

> **注意**
>
> 不带参数调用 \pset 是一个错误。将来这个调用可能会显示所有打印选项的当前状态。

\q

退出 psql 程序。

\qecho*text* [...]

此命令与 \echo 相同，只是所有输出将写入由 \o 设置的查询输出通道。

\r

重置（清空）查询缓冲区。

\s [*filename*]

打印命令行历史或把它保存到 filename。如果省略 filename，则历史写到标准输出。This option is only available if psql is configured to use the GNU history library.

> **注意**
>
> 从 psql 7.0 版起，不再需要保存命令历史，因为程序终止时会自动完成。每次 psql 启动时也会自动加载历史。

\set [*name* [*value* [...]]]

把内部变量 *name* 设置为 *value*，如果给出多个值则设置为它们的串接。如果没有给出第二个参数，该变量只是被设置为没有值。要取消设置一个变量，使用 \unset 命令。

有效的变量名可以包含字符、数字和下划线。详见有关 psql 变量的小节。

尽管欢迎你把任何变量设置为任何值，psql 把若干变量视为特殊的。它们记录在有关变量的小节中。

> **注意**
>
> 此命令与 SQL 命令 SET 完全无关。

\t

切换输出中的列名标题和行数页脚的显示状态。这个命令等价于\pset tuples_only，提供它是为了使用方便。

\T*table_options*

允许你指定要放在 HTML 表输出模式的 table 标签中的选项。这条命令等价于 \pset tableattr *table_options*。

\w {*filename*|\|*command*}

把当前查询缓冲区输出到文件*filename*，或通过管道送给 Unix 命令*command*。

\x

切换扩展行格式模式。因此它等价于 \pset expanded。

\z [*pattern*]

产生数据库中所有表及其相应访问权限的列表。如果给出参数，它被当作一个正则表达式，只列出匹配它的表。

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
----------+-------------------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

这样阅读：

- "=r"：PUBLIC 拥有对该表的读（SELECT）权限。
- "joe=arwR"：用户 joe 拥有读、写（UPDATE、DELETE）、追加（INSERT）权限，以及在该表上创建规则的权限。
- "group staff=ar"：组 staff 拥有 SELECT 和 INSERT 权限。

GRANT 和 REVOKE 命令用于设置访问权限。

\! [*command*]

进入一个单独的 shell，或执行 shell 命令 *command*。参数不会被进一步解释；shell 会原样看到它们。

\?

获取有关斜线（\）命令的帮助信息。

命令行选项

如果做了相应配置，psql 既理解标准的 Unix 短选项，也理解 GNU 风格的长选项。后者并非在所有系统上都可用。

-a, --echo-all

在读入时把所有行打印到屏幕。这对脚本处理比交互模式更有用。This is equivalent to setting the variable ECHO to all.

-A, --no-align

切换到非对齐输出模式（默认输出模式是对齐的）。

-c, --command *query*

指定 psql 执行一个查询字符串 query，然后退出。这在 shell 脚本中很有用。

query 必须是一个后端完全可解析的查询字符串（即不包含 psql 专有的特性），或者是单个反斜线命令。因此不能用这个选项混合 SQL 和 psql 元命令。要那样做，可以把字符串用管道输送到 psql 中，像这样：`echo "\x \\ select * from foo;" | psql`.

-d, --dbname *dbname*

指定要连接的数据库的名称。这等效于在命令行上把 dbname 指定为第一个非选项参数。

-e, --echo-queries

显示发送到后端的所有查询。这等效于把变量 ECHO 设置为 queries。

-E, --echo-hidden

回显 \d 及其他反斜线命令生成的实际查询。如果你想在自己的程序中包含类似的功能，可以使用它。这等效于在 psql 内部设置变量 `ECHO_HIDDEN`。

-f, --file *filename*

使用文件 filename 作为查询来源，而不是交互式地读取查询。该文件处理完后，psql 终止。This in many ways equivalent to the internal command `\i`.

使用这个选项与写 `psql < filename`. 有细微差别。一般来说，两者都会如你期望的那样工作，但使用 -f 会启用一些不错的特性，例如带行号的错误消息。使用这个选项还可能略微降低启动开销。另一方面，使用 shell 输入重定向的变体（理论上）保证产生与你手工输入所有内容时完全相同的输出。

-F, --field-separator *separator*

使用*separator*作为字段分隔符。这等效于`\pset fieldsep`或者`\f`。

-h, --host *hostname*

指定 postmaster 正在运行的主机的主机名。不带此选项时，通信使用本地 Unix 域套接字进行。

-H, --html

打开 HTML 表格输出。这等效于 \pset format html 或 \H 命令。

-l, --list

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这类似于内部命令 `\list`。

-o, --output *filename*

把所有查询输出放到文件*filename*中。这等效于命令`\o`。

-p, --port *port*

指定 postmaster 用于监听连接的 TCP/IP 端口，或者在省略时指定本地 Unix 域套接字文件扩展名。默认是 PGPORT 环境变量的值，如果没有设置，则默认为编译时指定的端口，通常是 5432。

-P, --pset *assignment*

以 `\pset` 的形式指定打印选项。注意，这里必须用一个等号而不是空格来分隔名称和值。例如，要把输出格式设置为 LaTeX，可以写 `-P format=latex`。

-q

指定 psql 安静地工作。默认情况下，它会打印欢迎消息和各种提示信息。如果使用了这个选项，以上那些就都不会输出。这与 -c 选项配合很有用。Within psql you can also set the `QUIET` variable to achieve the same effect.

-R, --record-separator *separator*

把*separator*用作记录分隔符。这等效于`\pset recordsep`命令。

-s, --single-step

以单步模式运行。即每条查询发送到后端之前都会提示用户，并可选择取消执行。
可用它来调试脚本。

-S, --single-line

以单行模式运行，其中换行符与分号一样终止一条查询。

> **注意**
>
> 这种模式是为坚持使用它的用户提供的，但并不一定值得推荐。特别是，
> 如果在一行中混合了SQL和元命令，对于没有经验的用户来说，
> 它们的执行顺序未必总是清楚的。

-t, --tuples-only

关闭列名和结果行计数页脚等的打印。它完全等效于 \t 元命令。

-T, --table-attr *table_options*

指定要放在HTML `table`标签内的选项。详见`\pset`。

-u

让 psql 在连接数据库之前提示输入用户名和密码。

此选项已被废弃，因为它在概念上有缺陷。（提示输入非默认用户名，
与因为后端要求而提示输入密码，实际上是两件不同的事情。）建议你改用 -U 和 -W 选项。

-U, --username *username*

以用户 username 而不是默认用户连接到数据库。（当然，你必须有权限这样做。）

-v, --variable, --set *assignment*

执行变量赋值，类似于 \set 内部命令。注意在命令行上必须用等号分隔名称和值。要取消变量的设置，去掉等号即可。这些赋值在启动的非常早期阶段完成，
因此为内部目的保留的变量可能会在稍后被覆盖。

-V, --version

显示 psql 的版本。

-W, --password

要求 psql 在连接数据库之前提示输入密码。即使你随后用元命令 `\connect`.

As of version 7.0, psql automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a “hack”, the automatic recognition might mysteriously fail, hence this option to force a prompt. 如果没有发出密码提示而后端又要求密码认证，连接尝试将失败。

-x, --expanded

打开扩展行格式模式。这等效于命令 `\x`。

-X, --no-psqlrc

不读取启动文件 ~/.psqlrc。

-?, --help

显示有关 psql 命令行参数的帮助。

高级特性

变量

psql 提供类似于常见 Unix 命令 shell 的变量替换特性。此特性还比较新且不太成熟，但将来有扩展它的计划。变量只是名称/值对，其中值可以是任意长度的任意字符串。要设置变量，使用 psql 元命令 `\set`：

```
testdb=> \set foo bar
```

把变量 “foo” 设置为值 “bar” 。要获取变量的内容，
在名称前加冒号并用作任意斜线命令的参数：

```
testdb=> \echo :foo
bar
```

> **注意**
>
> \set 的参数遵循与其他命令相同的替换规则。因此你可以构造有趣的引用，例如 `\set :foo 'something'`，分别得到 Perl 著名的 “软链接” 或 PHP 著名的 “可变变量” 。不幸的是（或者幸运的是？），这些构造没有任何实际用处。但另一方面，`\set bar :foo` 是一种完全合法的复制变量的方法。

如果调用 \set 时不带第二个参数，该变量只是被设置但没有值。要取消设置（或删除）一个变量，使用命令 \unset。

psql's 的内部变量名可以由字母、数字和下划线以任意顺序、任意数量组成。若干常规变量被 psql. They indicate certain option settings that can be changed at runtime by altering

the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

`DBNAME`

The name of the database you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

`ECHO`

如果设置为 all，所有输入的或来自脚本的行在解析或执行之前都被写到标准输出。要在程序启动时指定这一点，使用开关 `-a`。如果设置为“`queries`”，psql 只是在所有查询发送到后端时打印它们。对应的选项是 `-e`。

`ECHO_HIDDEN`

设置了此变量后，当反斜线命令查询数据库时会先显示该查询。这样你就可以研究 Postgres 的内部实现，并在自己的程序中提供类似的功能。如果把该变量设置为值“noexec”，查询只会显示，而不会真正发送到后端执行。

`ENCODING`

当前的客户端多字节编码。如果你没有设置为使用多字节字符，此变量将始终包含 SQL_ASCII。

`HISTCONTROL`

如果此变量设置为 `ignorespace`，以空格开头的行不会进入历史列表。如果设置为值 `ignoredups`，与上一条历史行相同的行不会进入。`ignoreboth` 值组合这两个选项。如果未设置，或设置为上述值以外的其他值，交互模式下读入的所有行都会保存到历史列表中。

注意

此特性是从 bash 厚颜无耻地抄袭来的。

`HISTSIZE`

在命令历史中存储的命令数。默认值为 500。

注意

此特性是从 bash 厚颜无耻地抄袭来的。

`HOST`

你当前连接到的数据库服务器主机。每次连接到数据库（包括程序启动）时都会设置它，但可以取消设置。

IGNOREEOF

如果未设置，向 psql 的交互式会话发送 EOF 字符（通常是 Control-D）将终止应用程序。如果设置为数值，则在应用程序终止之前忽略那么多个 EOF 字符。如果该变量已设置但没有数值，默认为 10。

> **注意**
>
> 此特性是从 bash 厚颜无耻地抄袭来的。

LASTOID

最后受影响的 oid 的值，由 INSERT 或 lo_insert 命令返回。此变量只在下一条 SQL 命令的结果显示之前保证有效。

LO_TRANSACTION

如果你使用 Postgres 大对象接口来专门存储放不进单个元组的数据，所有操作都必须包含在一个事务块中。(See the documentation of the large object interface for more information.) Since psql has no way to tell if you already have a transaction in progress when you call one of its internal commands \lo_export, \lo_import, \lo_unlink it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the last case you must provide your own BEGIN TRANSACTION/COMMIT block or the results will be unpredictable (usually resulting in the desired action's not being performed in any case).

要选择你想做的操作，把此变量设置为“rollback”，“commit”，or “nothing”. 默认是 to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

ON_ERROR_STOP

默认情况下，如果非交互式脚本遇到错误（如格式错误的 SQL 查询或内部元命令），处理会继续。这是 psql 的传统行为，但有时并不理想。如果设置了此变量，脚本处理将立即终止。如果该脚本是从另一个脚本调用的，后者也会以同样的方式终止。If the outermost script was not called from an interactive psql session but rather using the -f option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

你当前连接到的数据库服务器端口。每次连接到数据库（包括程序启动）时都会设置它，但可以取消设置。

PROMPT1, PROMPT2, PROMPT3

它们指定 psql 发出的提示符应有的样子。见“提示符”下文。

QUIET

此变量等价于命令行选项 -q。在交互模式下它可能不太有用。

SINGLELINE

此变量由命令行选项 -S 设置。你可以在运行时取消设置或重置它。

SINGLESTEP

此变量等价于命令行选项 -s。

USER

你当前连接时所用的数据库用户。每次连接到数据库（包括程序启动）时都会设置它，但可以取消设置。

SQL Interpolation

psql 变量的另一个有用特性是你可以把它们替换（插值）到常规 SQL 语句中。The syntax for this is again to prepend the variable name with a colon (:).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table my_table. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

此功能的一个流行应用是在后续语句中引用最后插入的 OID 来构建外键场景。此机制的另一个可能用法是把文件的内容复制到字段中。先把文件装载到一个变量中，然后按上面的做法进行。

```
testdb=> \set content '\'' `cat my_file.txt` '\''
testdb=> INSERT INTO my_table VALUES (:content);
```

此方法的一个可能问题是 my_file.txt 可能包含单引号。需要对这些引号转义，以免在处理第三行时导致语法错误。这可以用 sed 程序做到：

```
testdb=> \set content `sed -e "s/'/\\\\\\'/g" < my_file.txt`
```

注意反斜杠的正确数量（6 个）！可以这样解决：psql 解析完这一行后，它把 sed -e "s/'/\\\'/g" < my_file.txt 传给 shell。shell 会在双引号内做自己的处理，然后带着参数 -e 和 s/'/\\'/g 执行 sed。sed 解析它时会把两个反斜杠替换为一个，然后执行替换。也许在某个时刻你曾觉得所有 Unix 命令使用相同的转义字符是很棒的事。而这还没有考虑你可能还必须转义所有反斜杠的事实，因为 SQL 文本常量同样要经过某些解释。在这种情况下，你或许更适合在外部预先准备好文件。

由于冒号可以合法地出现在查询中，适用以下规则：如果该变量未设置，字符序列“冒号+名称”不会被更改。在任何情况下，你都可以用反斜线转义冒号来保护它不被解释。
（变量的冒号语法是嵌入式查询语言（如 ecpg）的标准 SQL 语法。数组切片和类型转换的冒号语法是 Postgres 扩展，因此存在冲突。）

提示符

psql 发出的提示符可以按你的偏好定制。三个变量 PROMPT1、PROMPT2 和 PROMPT3 包含描述提示符外观的字符串和特殊转义序列。提示符 1 是 psql 请求新查询时发出的常规提示符。提示符 2 在查询输入期间期待更多输入时发出，因为查询未以分号结束或引号未闭合。提示符 3 在你运行 SQL COPY 命令并需要在终端上输入元组时发出。

相应的提示符变量的值按字面打印，除非遇到百分号（%）。根据下一个字符的不同，会替换为某些其他文本。Defined substitutions are:

`%M`

The full hostname (with domain name) of the database server (or “localhost” if hostname information is not available).

`%m`

数据库服务器的主机名（在第一个点后截断）。

`%>`

数据库服务器正在监听的端口号。

`%n`

你连接时所用的用户名（不是本地系统用户名）。

`%/`

当前数据库的名称。

`%~`

类似于 %/，但如果该数据库是你的默认数据库则输出 ~（波浪号）。

`%#`

如果当前用户是数据库超级用户则为 #，否则为 >。

`%R`

在提示符 1 中通常是“=”，单行模式下是“^”，会话与数据库断开时（例如 `\connect` 失败时）是“!”。在提示符 2 中该序列被替换为“-”、“*”、单引号或双引号，具体取决于 psql 期待更多输入是因为查询尚未结束、你在 `/* ... */` 注释内，还是你在带引号的字符串内而期待更多输入。在提示符 3 中该序列不解析为任何内容。

`%`*`digits`*

如果 *`digits`* 以 `0x` 开头，其余字符按十六进制数解释，并替换为相应代码的字符。如果第一位数字是 0，这些字符按八进制数解释，并替换为相应的字符。否则按十进制数假设。

`%:`*`name`*`:`

psql 变量 *`name`* 的值。详见“变量”一节。

`%`` `*`command`*` `

command 的输出，类似于普通的反引号替换。

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

> **注意**
>
> 这个特性是可耻地从tcsh抄袭过来的。

Miscellaneous

psql 正常结束时向 shell 返回 0；发生自身的致命错误（内存不足、找不到文件）时返回 1；与后端的连接变坏且会话不是交互式时返回 2；脚本中发生错误且设置了变量 ON_ERROR_STOP 时返回 3。

启动之前，psql 会尝试读取并执行文件 $HOME/.psqlrc 中的命令。它可用于按喜好设置客户端或服务器（使用 \set 和 SET 命令）。

GNU readline

psql 支持 readline 和 history 库以方便地进行行编辑和历史检索。命令历史保存在你的主目录下名为 `.psql_history` 的文件中，并在 psql 启动时重新装载。也支持 Tab 补全，不过补全逻辑并不自称是一个 SQL 解析器。可用时，psql 会自动构建为使用这些特性。如果由于某种原因你不喜欢 Tab 补全，可以把它关闭，方法是把以下内容放进你主目录下名为 `.inputrc` 的文件中：

```
$if psql
set disable-completion on
$endif
```

（这不是 psql 而是 readline 的特性。详细内容请阅读其文档。）

如果你安装了 readline 库但 psql 似乎没有使用它，必须确保 Postgres 的顶级 configure 脚本能找到它。`configure` 需要同时找到库 `libreadline.a`（或等价的共享库）以及头文件 `readline.h` 和 `history.h`（或 `readline/readline.h` 和 `readline/history.h`）。如果你把库和头文件安装在不常见的地方，必须把位置告诉 `configure`，例如：

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/
lib  ...
```

然后你必须重新编译 psql（不一定要重新编译整个代码树）。

GNU readline 库可以从 GNU 项目的 FTP 服务器 ftp://ftp.gnu.org 获取。

示例

> **注意**
>
> This section only shows a few examples specific to psql. 如果你想学习 SQL 或熟悉 Postgres，可能希望阅读发行版中包含的教程。

第一个例子显示如何把一条查询分布在多行输入上。注意提示符的变化。

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE
```

现在再看一下表定义：

```
testdb=> \d my_table
```

```
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

假设你已经在表中填入了数据并且想查看它：

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

Notice how the int4 colums in right aligned while the text column in left aligned. You can make this table look differently by using the `\pset` command.

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
```

```
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

附录

缺陷与问题

- 在早期，psql 允许第一个参数直接跟在（单字母）命令后面。为了兼容性这一点在一定程度上仍被支持，但我不打算在这里解释细节，因为不鼓励这种用法。但如果你收到奇怪的消息，请记住这一点。For example

  ```
  testdb=> \foo
  Field separator is "oo".
  ```

  可能不是人们期望的结果。

- psql 只与同版本的服务器顺畅配合。这并不意味着其他组合会立即失败，但可能出现或明显或微妙的问题。

- 在 copy in（数据发送到服务器）期间按 Control-C 的行为并不理想。如果你收到类似 “PQexec：你必须自己退出 COPY 状态” 的消息，只需输入 \c - - 重置连接。

pgtclsh

pgtclsh — Postgres TCL shell 客户端

大纲

```
pgtclsh [ dbname ]
```

输入

dbname

要访问的一个现有数据库的名称。

输出

描述

pgtclsh 为 Postgres 提供一个 TCL shell 接口。

通过 tcl 访问 Postgres 的另一种方式是使用 pgtksh 或 pgaccess。

pgtksh

pgtksh — Postgres 图形化 TCL/TK shell

大纲

```
pgtksh [ dbname ]
```

输入

dbname

要访问的现有数据库的名称。

输出

描述

pgtksh 为 Postgres提供一个图形化 TCL/TK shell 接口。

访问 Postgres 的另一种方式是使用 TCL，即使用 pgtclsh 或 pgaccess。

vacuumdb

vacuumdb — 清理并分析一个 Postgres 数据库

大纲

```
vacuumdb [ options ] [ --analyze | -z ]
    [ --alldb | -a ] [ --verbose | -v ]
    [ --table 'table [ ( column [,...] ) ]' ] [ [-d] dbname ]
```

Inputs

vacuumdb 接受下列命令行参数：

-d *dbname*
--dbname *dbname*

指定要清理或分析的数据库名。

-z
--analyze

为优化器的使用计算数据库的统计信息。

-a
--alldb

清理所有数据库。

-v
--verbose

在处理过程中输出详细信息。

-t *table* [(*column* [,...])]
--table *table* [(*column* [,...])]

只清理或分析 *table*。列名只能与 `--analyze` 选项一起指定。

> **提示**
>
> 如果你指定了要清理（vacuum）的列，可能需要向 shell 转义括号。

vacuumdb 还接受下列用于连接参数的命令行参数：

-h *host*
--host *host*

指定 postmaster 所在机器的主机名。

-p *port*
--port *port*

指定 postmaster 正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U *username*
--username *username*

要连接使用的用户名。

-W
--password

强制口令提示。

-e
--echo

回显 vacuumdb 生成并发送给后端的命令。

-q
--quiet

不显示响应。

Outputs

`VACUUM`

一切正常。

`vacuumdb: Vacuum failed.`

出了问题。vacuumdb 只是一个包装脚本。关于错误消息和潜在问题的详细讨论，参见 VACUUM 和 psql。

Description

vacuumdb 是一个用于清理 Postgres 数据库的工具。vacuumdb 还会生成供 Postgres 查询优化器使用的内部统计信息。

vacuumdb 是一个外壳脚本包装，它通过 Postgres 的交互式终端 psql 调用后端命令 VACUUM。通过这种方式清理数据库与其他方法没有实质差别。脚本必须能找到 psql，而且目标主机上必须有数据库服务器在运行。此外，适用于 psql 和 libpq 前端库的任何默认设置和环境变量也同样有效。

Usage

清理数据库 `test`：

```
$ vacuumdb test
```

要为优化器分析名为 `bigdb` 的数据库：

```
$ vacuumdb --analyze bigdb
```

要为优化器分析数据库 `xyzzy` 中表 `foo` 的单个列 `bar`：

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

第 21 章 系统应用程序

这里是 Postgres 服务器和支持工具的参考信息。

initdb

initdb — 创建一个新的 Postgres 数据库安装

大纲

```
initdb [ --pgdata|-D dbdir ]
    [ --sysid|-i sysid ]
    [ --pwprompt|-W ]
    [ --encoding|-E encoding ]
    [ --pglib|-L libdir ]
    [ --noclean | -n ] [ --debug | -d ] [ --template | -t ]
```

输入

--pgdata=*dbdir*
-D *dbdir*
PGDATA

这个选项指定在文件系统中何处存放数据库。这是 initdb 所需的唯一信息，但你可以通过设置 PGDATA环境变量来省去它，这样很方便，因为数据库服务器（postmaster）之后可以通过同一个变量找到数据库目录。

--sysid=*sysid*
-i *sysid*

选择数据库超级用户的系统 ID。默认为运行 initdb 的用户的有效用户 ID。超级用户的 sysid 是什么其实并不重要，但人们可能会选择从某个数字（如 0 或 1）开始编号。

--pwprompt
-W

让 initdb 提示输入数据库超级用户的口令。如果你不打算使用口令认证，这无关紧要。否则，在设置好口令之前，你将无法使用口令认证。

--encoding=*encoding*
-E *encoding*

选择模板数据库的多字节编码。这也会是你以后创建的任何数据库的默认编码，
除非在那里另行覆盖。要使用多字节编码特性，必须在构建时指定，
同时你还要选择这个选项的默认值。

还有其他一些较少使用的参数：

--pglib=*libdir*
-l *libdir*

initdb 需要一些输入文件来初始化数据库。这个选项告诉它到哪里去找这些文件。通常你不必操心这一点，因为 initdb 知道最常见的安装布局，会自己找到这些文件。如果需要显式指定它们的位置，系统会告诉你。如果发生这种情况，其中一个文件名为 global1.bki.source，传统上与其他文件一起安装在库目录（例如 /usr/local/pgsql/lib）中。

--template
-t

替换现有数据库系统中的 template1数据库，其他什么都不动。当你需要用来自更新版本Postgres的 initdb升级你的template1数据库时，或者当你的template1数据库因某些系统问题而损坏时，这很有用。通常，template1
的内容在数据库系统的整个生命期内保持不变。用 initdb加 --template 选项运行不会破坏任何东西。

--noclean
-n

默认情况下，当 initdb判定某个错误使它无法完全创建数据库系统时，它会删除在判定无法完成工作之前可能已创建的所有文件。这个选项抑制任何清理，因此对调试有用。

--debug
-d

打印引导后端的调试输出以及一些公众较少感兴趣的其他消息。引导后端是 initdb用来创建目录表的程序。这个选项会产生大量的输出。

输出

initdb会在指定的数据区域中创建作为完整安装的系统表和框架的文件。

描述

initdb创建一个新的 Postgres数据库系统。数据库系统是由同一个 Unix 用户管理、并由单个postmaster 管理的一组数据库的集合。

创建数据库系统包括：创建存放数据库数据的目录，
生成共享目录表（不属于任何特定数据库的表），以及创建 template1 数据库。当你创建新数据库时，template1 数据库中的所有内容都会被复制。其中包含为内建类型等内容填好的目录表。

你不能以 root 身份执行initdb。这是因为你同样不能以 root 身份运行数据库服务器，而服务器需要访问initdb创建的文件。此外，在初始化阶段，还没有用户和访问控制被安装，postgres 只会以当前 Unix 用户的名称连接，因此你必须在将拥有服务器进程的账户下登录。

虽然initdb会尝试创建相应的数据目录，但它很可能没有权限这样做。因此，最好在运行initdb之前先创建数据目录，并把它的所有权移交给数据库超级用户。

initlocation

initlocation — 创建一个辅助的 Postgres 数据库存储区

大纲

```
initlocation directory
```

输入

directory

你想让备选数据库放在你的 Unix 文件系统的什么地方？

输出

initlocation 将在指定的位置创建目录。

描述

initlocation 创建一个新的 Postgres 辅助数据库存储区。关于如何管理和使用辅助存储区，见 CREATE DATABASE 下的讨论。如果参数不包含斜杠并且不是有效的路径，就假定它是一个环境变量，并引用它。见末尾的例子。

要使用这个命令，你必须（例如用 'su'）以数据库超级用户身份登录。

用法

要在一个备选位置创建数据库，使用环境变量：

```
$ export PGDATA2=/opt/postgres/data
```

启动并停止 postmaster，使它看到 $PGDATA2 环境变量。系统的配置必须使 postmaster 每次启动时都能看到 $PGDATA2。

```
$ initlocation PGDATA2
$ createdb -D 'PGDATA2' 'testdb'
```

或者，如果你允许绝对路径，可以写：

```
$ initlocation /opt/postgres/data
$ createdb -D '/opt/postgres/data/testdb' testdb
```

ipcclean

ipcclean — 清理异常中止的后端留下的共享内存和信号量

大纲

```
ipcclean
```

输入

无。

输出

无。

描述

ipcclean 通过删除用户 `postgres` 拥有的所有实例，清理异常中止的后端留下的共享内存和信号量空间。只有 DBA 才应执行这个程序，因为在多用户执行期间运行它可能导致古怪的行为（即崩溃）。如果在启动 postmaster 或后端服务器时遇到诸如 `semget: No space left on device` 这样的消息，就应当执行这个程序。

如果在 postmaster 运行时执行这个命令，该 postmaster 分配的共享内存和信号量将被删除。这将导致该 postmaster 启动的后端服务器全面失效。

这个脚本是一个应急之作（hack），但自它写成以来的许多年里，
没有人提出过同样有效且可移植的解决方案。欢迎提出建议。

该脚本对 ipcs 实用程序的输出格式做了假设，这在不同的操作系统上可能不成立。因此，它可能无法在你的特定操作系统上工作。

pg_passwd

pg_passwd — 操纵平面口令文件

大纲

```
pg_passwd filename
```

描述

pg_passwd 是一个操纵Postgres 平面口令文件功能的工具。这种风格的口令认证在安装中不是必需的，而是多种受支持的安全机制之一。

在$PGDATA/pg_hba.conf中以与Ident 认证相同的风格指定口令文件：

```
host  unv     133.65.96.250   255.255.255.255 password passwd
```

上面这一行允许来自 133.65.96.250 的访问使用 $PGDATA/passwd中列出的口令。口令文件的格式遵循 /etc/passwd 和 /etc/shadow 的格式。第一个字段是用户名，第二个字段是加密后的口令。其余部分完全被忽略。因此下面的三个示例行指定的是相同的用户和口令对：

```
pg_guest:/nB7.w5Auq.BY:10031::::::
pg_guest:/nB7.w5Auq.BY:93001:930::/home/guest:/bin/tcsh
pg_guest:/nB7.w5Auq.BY:93001
```

把口令文件提供给 pg_passwd 命令。在上面描述的情况下，把工作目录切换到PGDATA后，执行下面的命令为pg_guest 指定新口令：

```
 % pg_passwd passwd
 Username: pg_guest
 Password:
 Re-enter password:
```

其中Password: 和Re-enter password: 提示要求输入相同的口令，且输入不会显示在终端上。原口令文件会被改名为 passwd.bk。

psql 使用-u 选项来调用这种风格的认证。

下面的几行展示了该选项的用法示例：

```
% psql -h hyalos -u unv
Username: pg_guest
Password:
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL
   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: unv
unv=>
```

Perl5 认证像这样使用新风格的Pg.pm:

```
$conn = Pg::connectdb("host=hyalos dbname=unv
                       user=pg_guest password=xxxxxxx");
```

更多细节请参阅 src/interfaces/perl5/Pg.pm。

Pg{tcl,tk}sh 认证像这样使用 pg_connect 命令的 -conninfo 选项:

```
% set conn [pg_connect -conninfo \\
            "host=hyalos dbname=unv \\
             user=pg_guest password=xxxxxxx "]
```

你可以通过执行下面的命令列出该选项的所有键:

```
% puts [ pg_conndefaults]
```

pg_upgrade

pg_upgrade — 允许从前一个版本升级而无需重新加载数据

大纲

```
pg_upgrade [ -f filename ] old_data_dir
```

描述

pg_upgrade 是一个用于从前一个 Postgres 版本升级而无需重新加载所有数据的工具。并非所有 Postgres 版本间的迁移都能用这种方式处理。请查阅发行说明了解与你的安装相关的细节。

用 pg_upgrade 升级 Postgres

1. 备份你现有的数据目录，最好用 pg_dumpall 做一次完整的转储。
2. 然后执行：

   ```
   % pg_dumpall -s >db.out
   ```

   转储出旧数据库的表定义而不含任何数据。
3. 停止旧的 postmaster 和所有后端。
4. 用 mv 把你旧的 pgsql `data/` 目录重命名为 `data.old/`。
5. 执行

   ```
   % make install
   ```

   以安装新的二进制文件。
6. 运行 initdb 创建一个包含新版本系统表的新 template1 数据库。
7. 启动新的 postmaster。（注意：在升级完成之前，绝对不能有用户连接到数据库。你可以临时不带 -i 启动 postmaster，和/或临时修改 `pg_hba.conf`。）
8. 把工作目录切换到 pgsql 主目录，然后输入：

   ```
   % pg_upgrade -f db.out data.old
   ```

   该程序会做一些检查以确保一切都配置正确，然后运行你的 db.out 脚本重建你原有的所有数据库和表，但不包含数据。接着它会把包含非系统表和索引的数据文件从 `data.old/` 物理移动到相应的 `data/` 子目录中，替换 db.out 脚本执行期间创建的空数据文件。
9. 如需允许用户登录，恢复你原来的 `pg_hba.conf`。
10. 停止并重启 postmaster。
11. 仔细检查升级后数据库的内容。如果发现问题，你需要用完整的 pg_dump 备份恢复。满意之后你可以删除 `data.old/` 目录。

12. 升级后的数据库将处于未清理（un-vacuumed）的状态。在开始生产工作之前，你可能想运行一次 `VACUUM ANALYZE`。

postgres

postgres — 运行一个 Postgres 单用户后端

大纲

```
postgres [ dbname ]
postgres [ -B nBuffers ] [ -C ] [ -D DataDir ] [ -E ] [ -F ]
    [ -O ] [ -P ] [ -Q ] [ -S SortSize ] [ -d [ DebugLevel ] ] [ -e ]
    [ -o ] [ OutputFile ] [ -s ] [ -v protocol ] [ dbname ]
```

输入

postgres 接受下列命令行参数：

dbname

可选参数 *dbname* 指定要访问的数据库的名称。*dbname* 默认为环境变量 USER 的值。

-B *nBuffers*

如果后端运行在 postmaster 之下，*nBuffers* 就是 postmaster 为它启动的后端服务器进程分配的共享内存缓冲区数量。如果后端独立运行，此选项指定要分配的缓冲区数量。
该值默认为 64 个缓冲区，每个缓冲区 8k 字节（或者 config.h 中 BLCKSZ 设置的值）。

-C

不显示服务器版本号。

-D *DataDir*

指定用作数据库目录树根的目录。如果未给出 -D，默认的数据目录名是环境变量 PGDATA 的值。如果未设置 PGDATA，则使用的目录是 $POSTGRESHOME/data。如果两个环境变量都未设置且未指定此命令行选项，则使用编译时设置的默认目录。

-E

回显所有查询。

-F

禁用每个事务之后的自动 fsync() 调用。此选项提升性能，但事务进行中发生的操作系统崩溃可能导致最近输入的数据丢失。没有 fsync() 调用时，数据由操作系统缓冲，晚些时候才写入磁盘。

-O

越过限制，使系统表结构可以被修改。这些表通常是表名以 "pg_" 开头的表。

-P

在扫描/更新系统元组时忽略系统索引。对系统表/索引执行 REINDEX 需要此选项。系统表通常是表名以 "pg_" 开头的表。

-Q

指定"安静"模式。

-S *SortSize*

指定内部排序和哈希在求助于临时磁盘文件之前可使用的内存量。
取值以千字节为单位指定，默认为 512 千字节。注意，对于一个复杂查询，可能有多个排序和/或哈希并行运行，每一个在被允许开始把数据放入临时文件之前，都可以使用多达 *SortSize* 千字节。

-d [*DebugLevel*]

可选参数 *DebugLevel* 决定后端服务器将产生的调试输出量。如果 *DebugLevel* 为一，postmaster 会跟踪所有连接流量，仅此而已。对于二级及更高级别，后端进程会打开调试，postmaster 显示更多信息，包括后端环境和进程流量。注意，如果没有为后端服务器指定发送其调试输出的文件，这些输出将出现在其父 postmaster 的控制终端上。

-e

此选项控制日期在进出数据库时如何被解释。如果给出了 -e 选项，传给前端进程和从前端进程传来的日期将被假定为"欧洲"格式（DD-MM-YYYY），否则日期被假定为"美国"格式（MM-DD-YYYY）。后端接受多种格式的日期，
对输入日期而言此开关主要影响有歧义情形的解释。更多信息见 PostgreSQL 用户指南。

-o *OutputFile*

把所有调试和错误输出发送到 *OutputFile*。如果后端运行在 postmaster 之下，错误消息仍会发送到前端进程以及 *OutputFile*，但调试输出发送到 postmaster 的控制终端（因为只有一个文件描述符能发送到实际的文件）。

-s

在每个查询结束时打印时间信息和其他统计信息。这对基准测试或调整缓冲区数量很有用。

-v *protocol*

指定此特定会话要使用的前端/后端协议的编号。

还有几个可以指定的其他选项，主要用于调试目的。这里列出它们仅供 Postgres 系统开发者使用。强烈不鼓励使用这些选项中的任何一个。而且，
这些选项中的任何一个都可能在任何时候消失或改变。

这些特殊用途的选项是：

-A n|r|b|Q\fIn\fP|X\fIn\fP

此选项产生极大量的输出。

-L

关闭锁定系统。

-N

禁用换行符作为查询分隔符。

-f [s | i | m | n | h]

禁止使用特定的扫描和连接方法：`s` 和 `i` 分别禁用顺序扫描和索引扫描，而 `n`、`m` 和 `h` 分别禁用嵌套循环、归并和哈希连接。

> **注意**
>
> 顺序扫描和嵌套循环连接都无法被完全禁用；`-fs`和 `-fn`选项只是在优化器有其他选择时，尽量不使用这些计划类型。

-i

阻止查询执行，但显示计划树。

-p *dbname*

向后端服务器表明它是由 postmaster 启动的，并对缓冲池管理、文件描述符等做出不同的假定。-p 之后的开关仅限于那些被认为"安全"的。

-t pa[rser] | pl[anner] | e[xecutor]

打印与每个主要系统模块相关的每个查询的计时统计信息。此选项不能与 `-s` 一起使用。

输出

直接执行后端服务器时你可能看到的错误消息近乎无穷多，其中最常见的可能是：

```
semget: No space left on device
```

如果看到这条消息，你应当运行 ipcclean 命令。做完之后再尝试启动 postmaster。如果仍然不行，你多半需要按照安装说明中的描述为共享内存和信号量配置内核。
如果你的内核共享内存和/或信号量限制特别小，
可能需要重新配置内核以增大其共享内存或信号量参数。

> **提示**
>
> 通过减小 -B 以降低 Postgres 的共享内存消耗，你或许可以推迟重新配置内核。

描述

Postgres 后端服务器可以直接从用户 shell 执行。这只应在 DBA 调试时进行，而不应在这组数据库的其他 Postgres 后端正由 postmaster 管理时进行。

这里解释的一些开关可以通过连接请求的"数据库选项"字段传给后端，
这样就能为特定后端设置它们而不必费力重启 postmaster。这对调试相关的开关特别方便。

可选参数 *dbname* 指定要访问的数据库的名称。*dbname* 默认为环境变量 `USER` 的值。

注意

处理共享内存问题的有用工具包括 ipcs(1)、ipcrm(1) 和 ipcclean(1)。另见 postmaster。

postmaster

postmaster — 运行 Postgres 多用户后端

大纲

```
postmaster [ -B nBuffers ] [ -D DataDir ] [ -N maxBackends ] [ -S ]
    [ -d DebugLevel ] [ -i ] [ -l ]
    [ -o BackendOptions ] [ -p port ] [ -n | -s ]
```

输入

postmaster 接受下列命令行参数：

-B *nBuffers*

设置 postmaster 为它启动的后端服务器进程分配使用的共享内存磁盘缓冲区数量。该值默认为 64 个缓冲区，每个缓冲区为 8k 字节（或者在 src/include/config.h 中为 BLCKSZ 设置的值）。

-D *DataDir*

指定用作数据库目录树根的目录。如果没有给出 -D，默认的数据目录名是环境变量 PGDATA 的值。如果未设置 PGDATA，则使用的目录是 $POSTGRESHOME/data。如果两个环境变量都未设置且未指定这个命令行选项，则使用编译时设置的默认目录。

-N *maxBackends*

设置该 postmaster 允许启动的最大后端服务器进程数。默认值为 32，但只要你的系统支持那么多进程，最高可设为 1024。（注意 -B 至少要为 -N 的两倍，因此增大 -N 时也需要增大 -B。）-N 的默认值和上限都可以在构建 Postgres 时更改（见 src/include/config.h）。

-S

指定 postmaster 进程应以静默模式启动。也就是说，它将与用户的（控制）tty 脱离、建立自己的进程组，并把标准输出和标准错误重定向到 /dev/null。

注意，使用这个开关会使问题的排查变得非常困难，因为这个 postmaster 及其子后端正常情况下会产生的所有跟踪和日志输出都会被丢弃。

-d *DebugLevel*

决定后端服务器将产生的调试输出数量。如果 *DebugLevel* 为一，postmaster 将跟踪所有连接流量。二级及更高的级别会打开越来越多的来自后端进程的调试输出，postmaster 显示的信息也更多，包括后端环境和进程流量。注意，除非把 postmaster 的标准输出和标准错误重定向到日志文件，否则所有这些输出都会出现在 postmaster 的控制 tty 上。

-i

允许客户端通过 TCP/IP（Internet 域）连接。不使用这个选项时，只接受本地 Unix 域套接字连接。

-l

启用使用 SSL 的安全连接。还需要 -i 选项。你必须以启用 SSL 的方式编译才能使用这个选项。

-o *BackendOptions*

指定的 postgres 选项（*BackendOptions*）会传递给该 postmaster 启动的所有后端服务器进程。如果选项字符串包含空格，整个字符串必须加引号。

-p *port*

指定 postmaster 用来监听前端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展名。默认为 PGPORT 环境变量的值，如果未设置 PGPORT，则默认为编译 Postgres 时确定的值（通常为 5432）。如果你指定的不是默认端口，所有前端应用（包括 psql）都必须用命令行选项或 PGPORT 指定同一端口。

还有两个额外的命令行选项可用于调试导致后端异常死亡的问题。这些选项控制 postmaster 在这种情况下的行为，两个选项都不用于日常操作。

这种情况下的常规策略是通知所有其他后端必须终止，然后重新初始化共享内存和信号量。
这是因为出错的后端在终止之前可能已经破坏了某些共享状态。

这些特殊情况的选项是：

-n

postmaster 不会重新初始化共享数据结构。有经验的系统程序员随后可以用调试器检查共享内存和信号量状态。

-s

postmaster 会通过发送信号 SIGSTOP 停止所有其他后端进程，但不会使它们终止。这让系统程序员可以手工收集所有后端进程的 core 转储。

输出

```
semget: No space left on device
```

如果你看到这条消息，应当运行 ipcclean 命令。之后再尝试启动 postmaster。如果仍然不行，你可能需要按照安装说明中的描述为共享内存和信号量配置内核。
如果你在单个主机上运行多个 postmaster 实例，或者内核的共享内存和/或信号量限制特别小，可能需要重新配置内核以增大其共享内存或信号量参数。

> **提示**
>
> 通过减小 -B 来降低 Postgres 的共享内存消耗，和/或减小 -N 来降低 Postgres 的信号量消耗，你可能可以推迟重新配置内核。

```
StreamServerPort: cannot bind to port
```

如果你看到这条消息，应当确保没有其他 postmaster 进程已在同一端口号上运行。判断这一点的最简单方法是使用命令

```
% ps -ax | grep postmaster
```

（用于 BSD 类系统），或

```
% ps -e | grep postmast
```

（用于 System V 类或 POSIX 兼容系统，如 HP-UX）。

如果你确信没有其他 postmaster 进程在运行却仍得到这个错误，请尝试用 `-p` 选项指定一个不同的端口。如果你终止 postmaster 后立即用同一端口重启它，也可能得到这个错误；这种情况下，只需等待几秒直到操作系统关闭该端口再重试即可。最后，
如果你指定的端口号被操作系统视为保留端口，也可能得到这个错误。例如，许多版本的 Unix 把 1024 以下的端口号视为受信任的，只允许 Unix 超级用户访问它们。

```
IpcMemoryAttach: shmat() failed: Permission denied
```

一个可能的解释是另一个用户试图在同一端口上启动 postmaster 进程，该进程获取了共享资源之后死掉了。由于 Postgres 的共享内存键基于分配给 postmaster 的端口号，如果单个主机上有多个安装，这种冲突很可能发生。如果没有其他 postmaster 进程在运行（见上文），运行 ipcclean 再试一次。如果有其他 postmaster 映像在运行，就必须找到这些进程的所有者，协调端口号的分配和/或删除不再使用的共享内存段。

描述

postmaster 管理前端和后端进程之间的通信，并分配共享缓冲池和 SysV 信号量（在没有测试并置指令的机器上）。postmaster 本身不与用户交互，应作为后台进程启动。

一个给定的 Postgres 安装中一次只应运行一个 postmaster。这里的安装指的是一个数据库目录和 postmaster 端口号。只有当每个 postmaster 各有独立的目录和端口号时，才能在一台机器上运行多个 postmaster。

注解

如果完全可以避免，不要在杀死 postmaster 时使用 `SIGKILL`。应改用 `SIGHUP`、`SIGINT` 或 `SIGTERM`（kill(1) 的默认信号）。使用

```
% kill -KILL
```

或其等价形式

```
% kill -9
```

将阻止 postmaster 在消亡前释放它持有的系统资源（例如共享内存和信号量）。请改用 `SIGTERM`，以免不得不手工清理（如前所述）。

处理共享内存问题的有用工具包括 ipcs(1)、ipcrm(1) 和 ipcclean(1)。

用法

要使用默认值启动 postmaster，键入：

```
% nohup postmaster >logfile 2>&1 &
```

这条命令将在默认端口（5432）上启动 postmaster。这是启动 postmaster 最简单、最常见的方式。

要用特定端口启动 postmaster：

```
% nohup postmaster -p 1234 &
```

这条命令将启动 postmaster 并通过端口 1234 通信。为了用 psql 连接这个 postmaster，你需要将它作为

```
% psql -p 1234
```

来运行，或者设置环境变量 PGPORT：

```
% setenv PGPORT 1234
% psql
```

部分 II. 管理员指南

安装与维护信息。

目录

第 22 章 平台移植

本手册描述 Postgres 7.0 版。Postgres 开发者社区已经在许多平台上编译和测试过 Postgres。最新信息请查阅网站[1]。

22.1. 当前支持的平台

截至发布时，下列平台已经过测试：

表 22.1. 支持的平台

OS	处理器	版本	报告日期	备注
AIX 4.3.2	RS6000	v7.0	2000-04-05	Andreas Zeugswetter[2]
BSDI 4.01	x86	v7.0	2000-04-04	Bruce Momjian[3]
Compaq Tru64 5.0	Alpha	v7.0	2000-04-11	Andrew McMurry[4]
FreeBSD 4.0	x86	v7.0	2000-04-04	Marc Fournier[5]
HPUX	PA-RISC	v7.0	2000-04-12	9.0x 和 10.20 皆可。Tom Lane[6]
IRIX 6.5.6f	MIPS	v6.5.3	2000-02-18	MIPSPro 7.3.1.1m N32 构建。Kevin Wheatley[7]
Linux 2.0.x	Alpha	v7.0	2000-04-05	带有已发布的补丁。Ryan Kirkpatrick[8]
Linux 2.2.x	armv4l	v7.0	2000-04-17	回归测试尚需完善。Mark Knox[9]
Linux 2.2.x	x86	v7.0	2000-03-26	Lamar Owens[10]
Linux 2.0.x	MIPS	v7.0	2000-04-13	Cobalt Qube. Tatsuo Ishii[11]
Linux 2.2.5	Sparc	v7.0	2000-04-02	Tom Szybist[12]
LinuxPPC R4	PPC603e	v7.0	2000-04-13	Tatsuo Ishii[13]
mklinux	PPC750	v7.0	2000-04-13	Tatsuo Ishii[14]
NetBSD 1.4	arm32	v7.0	2000-04-08	Patrick Welche[15]
NetBSD 1.4U	x86	v7.0	2000-03-26	Patrick Welche[16]

[1] http://www.postgresql.org/docs/admin/ports.htm
[2] mailto:Andreas.Zeugswetter@telecom.at
[3] mailto:maillist@candle.pha.pa.us
[4] mailto:andrew.mcmurry@astro.uio.no
[5] mailto:scrappy@hub.org
[6] mailto:tgl@sss.pgh.pa.us
[7] mailto:hxpro@cinesite.co.uk
[8] mailto:pgsql@rkirkpat.net
[9] mailto:segfault@hardline.org
[10] mailto:lamar.owen@wgcr.org
[11] mailto:t-ishii@sra.co.jp
[12] mailto:szybist@boxhill.com
[13] mailto:t-ishii@sra.co.jp
[14] mailto:t-ishii@sra.co.jp
[15] mailto:prlw1@newn.cam.ac.uk
[16] mailto:prlw1@newn.cam.ac.uk

OS	处理器	版本	报告日期	备注
NetBSD	m68k	v7.0	2000-04-10	Mac 8xx. Henry B. Hotz[17]
NetBSD/sparc	Sparc	v7.0	2000-04-13	Tom I Helbekkmo[18]
QNX 4.25	x86	v7.0	2000-04-01	Dr. Andreas Kardos[19]
SCO OpenServer 5	x86	v6.5	1999-05-25	Andrew Merrill[20]
SCO UnixWare 7	x86	v7.0	2000-04-18	见 FAQ。Billy G. Allie[21]
Solaris	x86	v7.0	2000-04-12	Marc Fournier[22]
Solaris 2.5.1-2.7	Sparc	v7.0	2000-04-12	Peter Eisentraut[23], Marc Fournier[24]
SunOS 4.1.4	Sparc	v7.0	2000-04-13	Tatsuo Ishii[25]
Windows/Win32	x86	v7.0	2000-04-02	客户端库或 ODBC/JDBC。无服务器端。Magnus Hagander[26]
WinNT/Cygwin	x86	v7.0	2000-03-30	使用 Cygwin 库。Daniel Horak[27]

注意

对于 Windows NT，Postgres 的服务器端移植使用 RedHat/Cygnus 的 Cygwin 库和工具集。对于 Windows 9x，由于操作系统的限制，没有可用的服务器端移植。

22.2. 未支持的平台

v6.3.x-v6.5.x 所列的平台应该也能在 v7.0 上工作，但在编制本列表时我们没有收到明确的确认。我们把它们列在这里，是想让你知道，只要稍加关注，这些平台可能是可以被支持的。

截至发布时，下列平台尚未针对 v7.0 或 v6.5.x 进行测试：

表 22.2. 未支持的平台

OS	处理器	版本	报告日期	备注
BeOS	x86	v7.0	2000-05-01	客户端支持即将到来？Adam Haberlach[28]

[17] mailto:hotz@jpl.nasa.gov
[18] mailto:tih@kpnQwest.no
[19] mailto:kardos@repas-aeg.de
[20] mailto:andrew@compclass.com
[21] mailto:Bill.Allie@mug.org
[22] mailto:scrappy@hub.org
[23] mailto:peter_e@gmx.net
[24] mailto:scrappy@hub.org
[25] mailto:t-ishii@sra.co.jp
[26] mha@sollentuna.net
[27] mailto:horak@sit.plzen-city.cz
[28] mailto:adam@newsnipple.com

OS	处理器	版本	报告日期	备注
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	v6.4 可能没问题。需要新的维护者。Brian E Gallew[29]
NetBSD-current	NS32532	v6.4	1998-10-27	日期数学运算有麻烦。Jon Buller[30]
NetBSD 1.3	VAX	v6.3	1998-03-01	v7.0 应该可以工作。Tom I Helbekkmo[31]
SVR4 4.4	m88k	v6.2.1	1998-03-01	v6.4.x 需要 TAS 自旋锁代码。Doug Winterburn[32]
SVR4	MIPS	v6.4	1998-10-28	无 64 位整数。Frank Ridderbusch[33]
Ultrix	MIPS, VAX	v6.x	1998-03-01	近期无报告；已过时？

有几个平台曾被尝试过，并据报道无法在标准发行版本上工作。
这里列出的另一些平台则没有提供足够的库支持，无法尝试。

表 22.3. 不兼容的平台

OS	处理器	版本	报告日期	备注
MacOS	all	v6.x	1998-03-01	库不兼容；请使用 ODBC/JDBC
NextStep	x86	v6.x	1998-03-01	仅客户端支持；v1.0.9 加补丁可用 David Wetzel[34]

[29] mailto:geek+@cmu.edu
[30] mailto:jonb@metronet.com
[31] mailto:tih@kpnQwest.no
[32] mailto:dlw@seavme.xroads.com
[33] mailto:ridderbusch.pad@sni.de
[34] mailto:dave@turbocat.de

第 23 章 配置选项

23.1. 配置参数（configure）

configure可用的全部参数可以通过键入以下命令获得：

```
$ ./configure --help
```

安装者可能对以下参数感兴趣：

```
Directories to install PostgreSQL in:
  --prefix=PREFIX         install architecture-independent files in
 PREFIX
                          [/usr/local/pgsql]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --libdir=DIR            object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
Features and packages:
  --disable-FEATURE       do not include FEATURE (same as --enable-
FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=
no)
--enable and --with options recognized:
  --with-template=template
                          use operating system template file
                              see template directory
  --with-includes=dirs    look for header files for tcl/tk, etc in
 DIRS
  --with-libraries=dirs   look for additional libraries in DIRS
  --with-libs=dirs        alternate spelling of --with-libraries
  --enable-locale         enable locale support
  --enable-recode         enable cyrillic recode support
  --enable-multibyte      enable multibyte character support
  --with-pgport=portnum   change default postmaster port
  --with-maxbackends=n    set default maximum number of server
 processes
  --with-tcl              build Tcl interfaces and pgtclsh
  --with-tclconfig=tcldir
                          tclConfig.sh and tkConfig.sh are in DIR
  --with-perl             build Perl interface and plperl
  --with-odbc             build ODBC driver package
  --with-odbcinst=odbcdir
                          change default directory for odbcinst.ini
  --enable-cassert        enable assertion checks (for debugging)
  --enable-debug          build with debugging symbols (-g)
  --with-CC=compiler
                          use specific C compiler
  --with-CXX=compiler
```

```
                          use specific C++ compiler
  --without-CXX           prevent building C++ code
```

有些系统在构建Postgres的某个特定特性时可能会遇到问题。例如，C++ 编译器损坏的系统可能需要指定--without-CXX来指示构建过程跳过 libpq++的构建。

如果你想使用未安装在系统标准搜索路径中的头文件或库来构建 Postgres，可以使用 --with-includes和 --with-libraries选项。例如，可以用它们来用一个实验版本的 Tcl 构建。如果需要为头文件或库指定多个非标准目录，可以这样写：

```
--with-includes="/opt/tcl/include /opt/perl5/include"
```

23.2. 构建参数（make）

许多与安装相关的参数可以在Postgres 安装的构建阶段设置。

在大多数情况下，这些参数应当放在专门用于此目的的文件 Makefile.custom中。默认的发行版不包含这个可选文件，因此你要用自己选择的文本编辑器创建它。升级安装时，
只需在构建之前把旧的 Makefile.custom 复制到新安装中即可。

或者，也可以在make命令行上设置变量：

```
make [ variable=value [...] ]
```

可以指定的许多变量中的几个如下：

POSTGRESDIR

安装树的顶层。

BINDIR

应用程序和实用工具的位置。

LIBDIR

目标库的位置，包括共享库。

HEADERDIR

头文件的位置。

ODBCINST

安装范围的psqlODBC（ODBC）配置文件的位置。

还有一些不那么常用的可选参数。
下面列出的许多参数适合在进行Postgres服务器代码开发时使用。

CFLAGS

为 C 编译器设置标志。应当用"+="赋值以保留相关的默认参数。

YFLAGS

为 yacc/bison 解析器设置标志。-v可以用来帮助诊断构建新解析器时的问题。应当用"+="赋值以保留相关的默认参数。

USE_TCL

启用 Tcl 接口的构建。

HSTYLE

用于从头构建文档的 DocBook HTML样式表。除非你正在根据doc/src/sgml/中与 DocBook 兼容的SGML源文档开发新文档，否则不会用到。

PSTYLE

用于从头构建印刷文档的 DocBook 样式表。除非你正在根据doc/src/sgml/中与 DocBook 兼容的SGML源文档开发新文档，否则不会用到。

下面是一个 PentiumPro Linux 系统的 Makefile.custom示例：

```
# Makefile.custom
# Thomas Lockhart 1999-06-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486 -O2

# documentation

HSTYLE= /home/tgl/SGML/db118.d/docbook/html
PSTYLE= /home/tgl/SGML/db118.d/docbook/print
```

23.3. 区域支持

注意

由 Oleg Bartunov 撰写。关于区域和俄语支持的更多信息，请见Oleg 的网页[1]。

在为俄罗斯莫斯科的一家公司做项目时，我遇到了 postgresql 不支持国家字母表的问题。在寻找可能的变通办法之后，我决定自己开发区域支持。我不是 C 程序员，但在使用 perl（调试）和 glimpse 时已经有一些区域编程的经验。在 Postgres源码树中挖掘了几天之后，我对 src/backend/utils/adt/varlena.c 和 src/backend/main/main.c 做了非常小的修改，就得到了我需要的东西！我当时只做了 LC_CTYPE和LC_COLLATE的支持，但后来其他人又添加了LC_MONETARY。我收到许多人关于这个补丁的消息，因此我决定把它发给开发者，
而（让我惊讶的是）它被并入了Postgres 发行版。

人们经常抱怨区域支持对他们不起作用。有几种常见的错误：

- 编译之前没有正确配置 postgresql。必须用 --enable-locale 选项运行 configure 来启用区域支持。启动 postmaster 时没有正确设置环境。必须在运行 postmaster 之前定义环境

[1] http://www.sai.msu.su/~megera/postgres/

变量 LC_CTYPE和LC_COLLATE，因为后端从环境中获取区域信息。我使用下面的 shell 脚本（runpostgres）：

```
#!/bin/sh

export LC_CTYPE=koi8-r
export LC_COLLATE=koi8-r
postmaster -B 1024 -S -D/usr/local/pgsql/data/ -o '-Fe'
```

并从 rc.local 中这样运行它：

```
/bin/su - postgres -c "/home/postgres/runpostgres"
```

- 操作系统的区域支持损坏（例如，Linux 下 libc 的区域支持曾多次变化，这引起了很多问题）。最新的 perl 也支持区域，如果区域损坏，perl -v会抱怨类似下面的内容：

```
8:17[mira]:~/WWW/postgres>setenv LC_CTYPE not_exist
8:18[mira]:~/WWW/postgres>perl -v
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LC_ALL = (unset),
    LC_CTYPE = "not_exist",
    LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

- 区域文件的位置错误！可能的位置包括：/usr/lib/locale（Linux、Solaris）、/usr/share/locale（Linux）、/usr/lib/nls/loc（DUX 4.0）。请查看man locale以找到正确的位置。在 Linux 下，我在/usr/lib/locale和 /usr/share/locale之间做了一个符号链接，以确保下一个 libc 不会破坏我的区域。

23.3.1. 有什么好处?

可以对包含国家字母表字符的字符串使用 ~* 和 order by 操作符。非英语用户肯定需要它。如果你不想使用区域相关的东西，只需取消定义 USE_LOCALE 变量。

23.3.2. 有什么缺点?

使用区域有一个明显的缺点--速度！所以，只在真正需要时才使用区域。

23.4. Kerberos 认证

Kerberos是一种工业标准的安全认证系统，适合在公共网络上进行分布式计算。

23.4.1. 可用性

Kerberos 认证系统并不随Postgres一起发行。Kerberos的各个版本通常以操作系统厂商的可选软件的形式提供。此外，还可以通过 MIT Project Athena[2] 获取源码发行版。

[2] ftp://athena-dist.mit.edu

> **注意**
>
> 即使你的厂商提供了某个版本，你也可能希望获取 MIT 版本，因为有些厂商的移植版本被故意削弱或弄得与 MIT 版本无法互操作。

位于美国和加拿大之外的用户请注意，
Kerberos中实际加密代码的分发受美国政府出口法规的限制。

关于Kerberos的询问应当指向你的厂商或 MIT Project Athena[3]。注意，FAQL（常见问题列表）会定期张贴到 Kerberos邮件列表[4]（发送邮件订阅[5]）和USENET 新闻组[6]。

23.4.2. 安装

Kerberos本身的安装在 Kerberos 安装说明中有详细介绍。请确保服务器密钥文件（`srvtab`或`keytab`）能被Postgres 账户以某种方式读取。

通过把文件`src/Makefile.global`中的 `KRBVERS`变量设置为适当的值，Postgres及其客户端可以编译为使用 MIT Kerberos协议的版本 4 或版本 5。你还可以更改Postgres期望找到相关库、头文件和它自己的服务器密钥文件的位置。

编译完成后，必须把Postgres 注册为一个Kerberos 服务。关于注册服务的更多细节，请参阅 Kerberos 操作说明及相关的手册页。

23.4.3. 操作

初次安装之后，Postgres 在各方面都应当像正常的 Kerberos服务一样运行。关于认证使用的细节，请参阅postmaster和 psql的 PostgreSQL 用户指南参考章节。

在Kerberos版本 5 的钩子中，对用户和服务命名做了以下假设：

- 假定用户主体名（aname）的第一个组件中包含实际的 Unix/Postgres用户名。
- 假定Postgres服务有两个组件，即服务名和一个主机名，主机名按版本 4 的方式规范化（即去掉所有域名后缀）。

表 23.1. Kerberos 参数示例

参数	示例
user	frew@S2K.ORG
user	aoki/HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host	postgres_dbms/ucbvax@S2K.ORG

在 MIT 发布版本 5 的正式版本之后的某个时候，将不再支持版本 4。

[3] info-kerberos@athena.mit.edu
[4] mailto:kerberos@athena.mit.edu
[5] mailto:kerberos-request@athena.mit.edu
[6] news:comp.protocols.kerberos

第 24 章 系统布局

图 24.1. Postgres文件布局

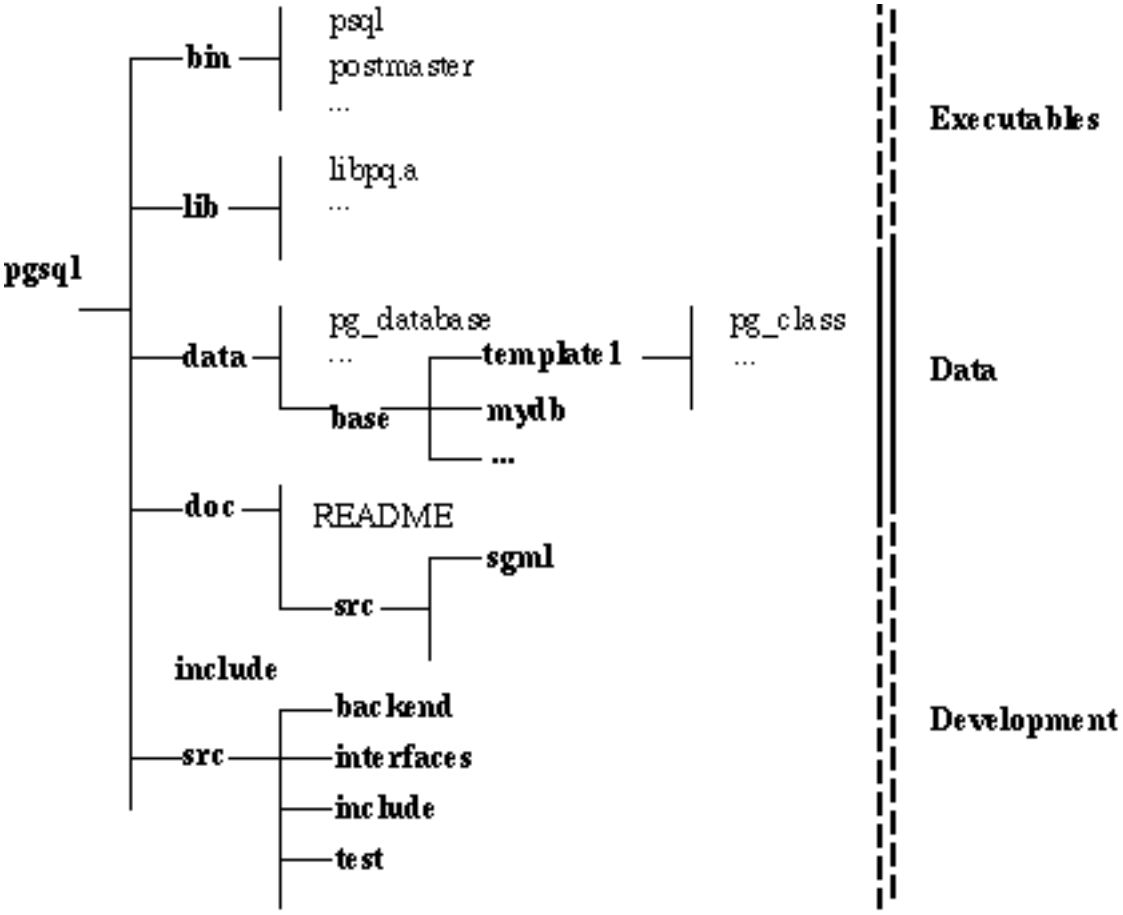


图 24.1展示了Postgres发行版按默认方式安装后的布局。为简单起见，我们假设Postgres 被安装在目录 `/usr/local/pgsql`中。因此，无论在哪里看到目录`/usr/local/pgsql`，你都应该把它替换为 Postgres实际安装所在的目录名。所有Postgres命令都被安装在目录 `/usr/local/pgsql/bin`中。因此，你应当把这个目录加到你的 shell 命令路径中。如果你使用 Berkeley C shell 的一个变体，例如 csh 或 tcsh，可以在你的主目录的 .login 文件中加入

```
set path = ( /usr/local/pgsql/bin path )
```

（上面的内容）。如果你使用 Bourne shell 的一个变体，例如 sh、ksh 或 bash，那么可以在

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

中加入这些内容（指你的主目录中的 .profile 文件）。从现在起，我们假设你已把 Postgres的 bin 目录加到了你的路径中。此外，在本文档中我们会频繁提到"设置 shell 变量"或"设置环境变量"。如果你没有完全理解上面关于修改搜索路径的段落，在继续之前你应当查阅描述你所用的 shell 的 Unix 手册页。

如果你没有按默认方式完成设置，可能还有一些工作要做。例如，
如果数据库服务器机器是一台远程机器，你需要把`PGHOST`

环境变量设置为数据库服务器机器的名字。环境变量 PGPORT也可能需要设置。底线是：如果你试图启动一个应用程序而它报怨无法连接到postmaster，
你必须回过头去确认你的环境已正确设置。

第 25 章 安装

PostgreSQL 7.0.3 的安装说明。

如果你还没有取得 PostgreSQL 发行版，可以从 ftp.postgresql.org[1] 获取，然后解包：

```
> gunzip postgresql-7.0.3.tar.gz
> tar -xf postgresql-7.0.3.tar
> mv postgresql-7.0.3 /usr/src
```

25.1. 开始之前

构建 PostgreSQL 需要 GNU make。它不能与其他 make 程序一起工作。在 GNU/Linux 系统上 GNU make 是默认工具，在其他系统上你可能会发现 GNU make 是以 `gmake` 的名称安装的。从现在起我们将用这个名称来指代 GNU make，无论它在你的系统上叫什么名字。要测试是否有 GNU make，输入

```
> gmake --version
```

如果你需要获取 GNU make，可以在 ftp://ftp.gnu.org 找到它。

关于所支持平台的最新信息位于 http://www.postgresql.org/docs/admin/ports.htm[2]。一般来说，大多数拥有现代库的 Unix 兼容平台都应该能够运行 PostgreSQL。发行版的 `doc` 子目录中有若干针对特定平台的 FAQ 和 README 文档，如果你遇到麻烦，不妨查阅。

尽管运行 PostgreSQL 所需的最小内存可以低至 8MB，但把内存扩展到 96MB 或更多会带来明显的速度提升。规律是：内存永远不嫌多。

检查你是否有足够的磁盘空间。编译期间源代码树大约需要 30 MB，安装目录大约需要 5 MB。一个空数据库占用约 1 MB，否则数据库占用的空间大约是容纳相同数据的平面文本文件的五倍。如果你运行回归测试，还会临时需要额外的 20MB。

要检查磁盘空间，使用

```
> df -k
```

考虑到当今硬盘的价格，在把数据库投入生产使用之前，
购置一块大而快的硬盘也许应该列入你的计划。

25.2. 安装过程

PostgreSQL 安装

对于全新安装或从旧版本 PostgreSQL 升级：

1. (可选的) 创建 PostgreSQL 超级用户账号。服务器将以这个用户的身份运行。用于生产环境时，你应当创建一个单独的、无特权的账号（通常使用 `postgres`）。如果你没有 root 权限或只是想随便玩玩，使用你自己的用户账号就够了。

[1] ftp://ftp.postgresql.org
[2] http://www.postgresql.org/docs/admin/ports.htm

以 root、bin 或任何其他拥有特殊访问权限的账号运行 PostgreSQL 都有安全风险；不要这样做。实际上 postmaster 会拒绝以 root 身份启动。

你不必在这个账号下进行构建和安装本身（尽管可以）。
需要以数据库超级用户身份登录时，我们会告诉你。

2. 为你的系统配置源代码。在这一步你可以为构建过程指定实际的安装路径，
并选择要安装哪些内容。进入 src 子目录并输入：

```
> ./configure
```

后面可以跟上你想给的任何选项。对于第一次安装，不带任何选项也应该没问题。
要获取完整的选项列表，输入：

```
> ./configure --help
```

一些较常用的选项有：

--prefix=BASEDIR

为 PostgreSQL 的安装选择一个不同的基目录。默认值为 /usr/local/pgsql。

--enable-locale

如果你想使用区域设置。

--enable-multibyte

允许使用多字节字符编码。这主要面向日语、韩语或汉语等语言。

--with-perl

构建 Perl 接口和 plperl 扩展语言。请注意 Perl 接口需要被安装到 Perl 模块的常规位置（通常在 /usr/lib/perl 下），因此你必须拥有 root 权限才能执行安装步骤。（通常最简单的做法是开始时不加 --with-perl，在完成 PostgreSQL 本身的安装之后再构建和安装 Perl 接口。）

--with-odbc

构建 ODBC 驱动程序包。

--with-tcl

构建需要 Tcl/Tk 的接口库和程序，包括 libpgtcl、pgtclsh 和 pgtksh。

3. 编译程序。输入

```
> gmake
```

编译过程可能需要 10 分钟到一个小时。你的耗时几乎肯定会有所不同。记住要使用 GNU make。

但愿最后显示的一行是

```
All of PostgreSQL is successfully made. Ready to install.
```

4. (可选的)　如果你想在安装之前测试新构建的服务器，可以在此时运行回归测试。回归测试是一套测试用例，用于验证 PostgreSQL 在你的机器上以开发者期望的方式运行。详细说明见回归测试。（务必使用"并行回归测试"方法，因为顺序方法只适用于已安装的服务器。）

5. 如果你不是在升级现有系统，跳到步骤 7。

   如果你运行的是 7.*，跳到步骤步骤 6。

   现在你需要备份现有的数据库。要转储你的数据库安装，输入：

   ```
   > pg_dumpall > db.out
   ```

   如果你想保留对象 id（oid），则在运行 pg_dumpall 时使用 -o 选项。但除非你有特殊理由这样做（例如在表中把 OID 用作键），否则不要这样做。

   务必使用你当前正在运行的版本中的 pg_dumpall 命令。7.0.3 的 pg_dumpall 不应被用于更旧的数据库。

   > **小心**
   >
   > 你必须确保数据库在备份期间没有被更新。必要时，关闭　postmaster，编辑文件 `/usr/local/pgsql/data/pg_hba.conf`　的权限使只有你能访问，然后把 postmaster 重新启动。

   除了使用 pg_dumpall 之外，常常也可以使用 pg_upgrade。

6. 如果你正在升级现有系统，现在就终止数据库服务器。输入

   ```
   > ps ax | grep postmaster
   ```

   或者

   ```
   > ps -e | grep postmaster
   ```

   （这两条中哪一条有效取决于你的系统。输错也不会造成危害。）
   这会列出若干进程的进程号，类似于这样：

   ```
   263  ?  SW   0:00 (postmaster)
   777  p1 S    0:00 grep postmaster
   ```

   输入下面这行，其中 *pid* 替换为 `postmaster` 进程的进程 id（上例中为 263）。（不要使用 "grep postmaster" 进程的 id。）

   ```
   > kill pid
   ```

提示

在开机时自动启动 PostgreSQL 的系统上，很可能有一个能完成同样事情的启动文件。例如，在 Redhat Linux 系统上你可能会发现

```
> /etc/rc.d/init.d/postgres.init stop
```

可以奏效。

如果你用过 pg_dumpall，把旧目录移开。输入以下命令：

```
> mv /usr/local/pgsql /usr/local/pgsql.old
```

（换成你自己的路径。）

7. 安装 PostgreSQL 的可执行文件和库。输入

   ```
   > gmake install
   ```

   这一步应当以你希望拥有已安装可执行文件的用户身份执行。
   这不必与数据库超级用户相同；有些人更喜欢让已安装的文件归 root 所有。

8. 如有必要，告诉你的系统如何找到新的共享库。具体做法因平台而异。
   最普遍可用的方法是设置环境变量 `LD_LIBRARY_PATH`：

   ```
   > LD_LIBRARY_PATH=/usr/local/pgsql/lib
   > export LD_LIBRARY_PATH
   ```

   （适用于 sh、ksh、bash、zsh）或

   ```
   > setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
   ```

   （适用于 csh 或 tcsh）。你可能想把它放进某个 shell 启动文件，例如 `/etc/profile`。

   在某些系统上，下面的方法是首选的，但你必须拥有 root 权限。编辑文件 `/etc/ld.so.conf`，加入一行

   ```
   /usr/local/pgsql/lib
   ```

   然后运行命令 `/sbin/ldconfig`。

   如果拿不准，请查阅你系统的手册页。如果以后你收到类似这样的消息

   ```
   psql: error in loading shared libraries
   libpq.so.2.1: cannot open shared object file: No such file or
    directory
   ```

   那么上面的步骤就是必要的。届时照做即可。

9. 如果你已把旧目录移开，现在创建数据库安装（工作数据文件）。为此你必须以 PostgreSQL 超级用户身份登录。以 root 身份是不行的。

```
> mkdir /usr/local/pgsql/data
> chown postgres /usr/local/pgsql/data
> su - postgres
> /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

-D 选项指定数据将要存放的位置。你可以使用任何想要的路径，它不必位于安装目录之下。只需确保在启动 initdb 之前超级用户账号能够写该目录（或者如果该目录尚不存在，能够创建它）。（如果你到目前为止一直是用 PostgreSQL 超级用户进行安装的，那么要在 root 所有的目录之下创建数据目录时，你可能需要临时以 root 身份登录。）

10. 上一步应该已经告诉你如何启动数据库服务器。现在就这样做。这个命令应该类似于

```
> /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

这会在前台启动服务器。要让它脱离到后台，可以使用 -S 选项，但这样你就看不到服务器产生的任何日志消息了。把服务器放到后台的更好方法是

```
> nohup /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data \
    </dev/null >>server.log 2>>1 &
```

11. (可选的) 如果你做过 pg_dumpall，把你的数据重新装回去：

```
> /usr/local/pgsql/bin/psql -d template1 -f db.out
```

你可能还想把旧的 pg_hba.conf 文件以及你为认证设置的其他任何文件（如口令文件）复制过来。

安装本身到此结束。为了让你的生活更有成效、更愉快，你应当看看下面这些可选步骤和建议：

- 设置一些环境变量会让生活更方便。首先，你可能想把 /usr/local/pgsql/bin（或等效路径）加入你的 PATH。为此，把下面内容加入你的 shell 启动文件，例如 ~/.bash_profile（或者 /etc/profile，如果你想让每个用户都受影响）：

```
> PATH=$PATH:/usr/local/pgsql/bin
```

此外，如果你在 PostgreSQL 超级用户的环境中设置了 PGDATA，就可以省略 postmaster 和 initdb 的 -D。

- 你可能想安装 man 和 HTML 文档。输入

```
> cd /usr/src/pgsql/postgresql-7.0.3/doc
> gmake install
```

这会把文件安装到 /usr/local/pgsql/doc 和 /usr/local/pgsql/man 之下。要让你的系统能找到 man 文档，你需要在某个 shell 启动文件中加入类似下面的一行：

```
> MANPATH=$MANPATH:/usr/local/pgsql/man
```

文档也有 Postscript 格式。如果你有 Postscript 打印机，或者你的机器已经用打印过滤器设置为可以接受 Postscript 文件，那么要打印用户指南只需输入

```
> cd /usr/local/pgsql/doc
> gunzip -c user.ps.tz | lpr
```

下面是如果你的系统上有 Ghostscript 并且向激光打印机打印时的做法。

```
> gunzip -c user.ps.gz \
    | gs -sDEVICE=laserjet -r300 -q -dNOPAUSE -sOutputFile=- \
    | lpr
```

各个系统的打印机设置千差万别。如果拿不准，请查阅你的手册或本地专家。

如果你完全初次接触 PostgreSQL，管理员指南很可能应该是你的第一读物，因为它包含如何设置数据库用户和认证的信息。

- 通常，你会想修改你的计算机，使其在每次启动时自动启动数据库服务器。这不是必需的；PostgreSQL 服务器可以在没有 root 干预的情况下成功地以非特权账号运行。

  不同的系统在开机时启动守护进程有不同的惯例，因此建议你熟悉它们。大多数系统都有文件 `/etc/rc.local` 或 `/etc/rc.d/rc.local`，把这样的命令放在那里几乎肯定不是坏主意。无论你怎么做，postmaster 都必须由 PostgreSQL 超级用户（`postgres`）运行，而不是由 root 或任何其他用户。因此你大概总想按照 `su -c '...' postgres` 的形式来组织你的命令行。

  保留服务器输出的日志可能是明智的。要以那种方式启动服务器，试试：

  ```
  > nohup su -c 'postmaster -D /usr/local/pgsql/data > server.log
   2>&1' postgres &
  ```

  下面是另外一些针对特定操作系统的建议。

  - 在 NetBSD 上编辑文件 rc.local，或在 SPARC Solaris 2.5.1 上编辑文件 rc2.d，使其包含下面这一行：

    ```
    > su postgres -c "/usr/local/pgsql/bin/postmaster -S -D /usr/
    local/pgsql/data"
    ```

  - 在 FreeBSD 2.2-RELEASE 中编辑 /usr/local/etc/rc.d/pgsql.sh，使其包含下面的行，并对它执行 chmod 755 和 chown root:bin。

    ```
    #!/bin/sh
    [ -x /usr/local/pgsql/bin/postmaster ] && {
        su -l pgsql -c 'exec /usr/local/pgsql/bin/postmaster
            -D/usr/local/pgsql/data
            -S -o -F > /usr/local/pgsql/errlog' &
        echo -n ' pgsql'
    }
    ```

你可以像上面那样放置换行。只要表达式未结束，shell 就足够聪明，会继续解析越过行尾的内容。exec 在 postmaster 进程之下节省了一层 shell，因此父进程是 init。

  - 在 RedHat Linux 中添加一个基于 `contrib/linux/` 中示例的文件 `/etc/rc.d/init.d/postgres.init`。然后从 `/etc/rc.d/rc5.d/S98postgres.init` 做一个指向该文件的软链接。

- 针对已安装的服务器运行回归测试（使用顺序测试方法）。
  如果你在安装之前没有运行过测试，现在绝对应该做。详细说明见回归测试。

要开始体验 Postgres，按上文说明设置好路径并启动服务器。要创建一个数据库，输入

```
> createdb testdb
```

然后输入

```
> psql testdb
```

连接到该数据库。在提示符下你可以输入 SQL 命令，开始体验。

第 26 章 在 Win32 上安装

在 Win32 上构建和安装 Postgres v6.4 客户端库的说明。

26.1. 构建这些库

Postgres 中包含的 makefile 是为 Microsoft Visual C++ 编写的，可能在其他系统上不能用。在其他情况下应当可以手动编译这些库。

要构建这些库，进入 `src` 目录，输入命令

```
copy include\config.h.win32 include\config.h
nmake /f win32.mak
```

这假设你的路径中有 Visual C++。

将构建出下列文件：

- `interfaces\libpq\Release\libpq.dll` - 可动态链接的前端库
- `interfaces\libpq\Release\libpqdll.lib` - 把你的程序链接到 libpq.dll 的导入库
- `interfaces\libpq\Release\libpq.lib` - 前端库的静态库版本
- `bin\psql\Release\psql.exe` - Postgresql 交互式 SQL 监视程序

26.2. 安装这些库

真正需要安装的只有该库的一部分，即 `libpq.dll` 库。在大多数情况下，这个文件应当放在 `WINNT\SYSTEM32` 目录中（在 Windows 95/98 系统上则放在 `WINDOWS\SYSTEM` 中）。如果这个文件是用安装程序安装的，应当使用文件中包含的 VERSIONINFO 资源带版本检查地安装，以确保不会覆盖较新版本的库。

如果你计划在这台机器上使用 libpq 进行开发，你必须把 `src\include` 和 `src\interfaces\libpq` 目录加到你的编译器设置的 include 路径中。

26.3. 使用这些库

要使用这些库，你必须把 `libpqdll.lib` 文件加到你的项目中（在 Visual C++ 中，只需右键点击项目并选择添加它）。

做完这些之后，就应该可以像在 Unix 平台上一样使用这个库了。

第 27 章 运行时环境

本章概述 Postgres 与操作系统的交互。

27.1. 在 Unix 上使用 Postgres

所有直接从 Unix shell 执行的 Postgres 命令都位于目录 `.../bin` 中。把这个目录包含在你的搜索路径中会让执行这些命令更容易。

每个站点都存在一组系统目录。其中包括一个类（`pg_user`），它为每个有效的 Postgres 用户包含一个实例。该实例指定了一组 Postgres 权限，例如充当 Postgres 超级用户的能力、创建/删除数据库的能力以及更新系统目录的能力。在把适当的实例安装到这个类中之前，Unix 用户无法用 Postgres 做任何事。有关系统目录的更多信息可以通过在相应的类上运行查询获得。

27.2. 启动 postmaster

postmaster 进程没有运行时，数据库什么也做不了。作为站点管理员，在启动 postmaster 之前有若干事项应当记住。这些在本手册的安装和配置小节中讨论。不过，如果 Postgres 是严格按照安装说明安装的，那么只需以下简单命令就应当足以启动 postmaster：

```
% postmaster
```

postmaster 偶尔会打印一些在排障时往往有帮助的消息。如果你想查看来自 postmaster 的调试消息，可以用 -d 选项启动它并把输出重定向到日志文件：

```
% postmaster -d > pm.log 2>&1 &
```

如果你不希望看到这些消息，可以输入

```
% postmaster -S
```

这样 postmaster 就是"S"（静默）的。这种情况下不需要与号（"&"），因为指定 -S 时 postmaster 会自动脱离终端。

27.3. 使用 pg_options

> **注意**
>
> 由 Massimo Dal Zotto[1] 贡献

可选文件 `data/pg_options` 包含后端用来控制跟踪消息和其他可调后端参数的运行时选项。后端收到 SIGHUP 信号时会重新读取该文件，因此无需重启 Postgres 就能即时更改运行时选项。此文件中指定的选项可以是跟踪包（`backend/utils/misc/trace.c`）使用的调试标志，
也可以是后端可用来控制自身行为的数值参数。

[1] mailto:dz@cs.unitn.it

所有 pg_options 在后端启动时初始化为零。新的或修改过的选项会被所有新启动的后端读取。要让更改对所有正在运行的后端生效，需要向 postmaster 发送 SIGHUP。该信号会被自动发送给所有后端。我们也可以通过直接向特定后端发送 SIGHUP，只为该后端激活更改。

pg_options 也可以用 Postgres 的 `-T` 开关指定：

```
postgres options -T "verbose=2,query,hostlookup-"
```

用于打印错误和调试消息的函数现在可以利用 syslog(2) 设施。打印到 stdout 或 stderr 的消息带有包含后端 pid 的时间戳前缀：

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

这种格式提高了日志的可读性，使人们能准确了解哪个后端在什么时间做什么。
它也让编写监视日志以检测数据库错误或问题、或计算事务时间统计的简单 awk 或 perl 脚本变得更容易。

打印到 syslog 的消息使用日志设施 LOG_LOCAL0。syslog 的使用可以由 syslog pg_option 控制。遗憾的是，许多函数直接调用 `printf()` 把消息打印到 stdout 或 stderr，这类输出无法重定向到 syslog，也无法带有时间戳。明智的做法是把所有对 printf 的调用替换为 PRINTF 宏，并把输出到 stderr 的做法改为使用 EPRINTF，这样我们就能以统一的方式控制所有输出。

`pg_options` 文件的格式如下：

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

注意 *`keyword`* 也可以是 `backend/utils/misc/trace.c` 中定义的选项名的缩写。

例 27.1. pg_options 文件

例如，我的 pg_options 文件包含以下值：

```
verbose=2
query
hostlookup
showportnumber
```

27.3.1. 可识别的选项

当前定义的选项有：

all

全局跟踪标志。允许的值有：

0

单独启用各跟踪消息

1

启用所有跟踪消息

-1

禁用所有跟踪消息

verbose

详尽程度标志。允许的值有：

0

无消息。这是默认值。

1

打印信息性消息。

2

打印更多信息性消息。

query

查询跟踪标志。允许的值有：

0

不打印查询。

1

在一行内打印压缩的查询。

4

打印完整查询。

plan

打印查询计划。

parse

打印解析器输出。

rewritten

打印重写后的查询。

pretty_plan

美化打印查询计划。

pretty_parse

美化打印解析器输出。

pretty_rewritten

美化打印重写后的查询。

parserstats

打印解析器统计。

plannerstats

打印规划器统计。

executorstats

打印执行器统计。

shortlocks

目前未用，但为将来启用特性所需。

locks

跟踪锁。

userlocks

跟踪用户锁。

spinlocks

跟踪自旋锁。

notify

跟踪通知函数。

malloc

目前未用。

palloc

目前未用。

lock_debug_oidmin

锁跟踪的最小关系 oid。

lock_debug_relid

如果非零，为锁跟踪的关系 oid。

lock_read_priority

目前未用。

deadlock_timeout

死锁检查计时器。

syslog

syslog 标志。允许的值有：

0

消息发到 stdout/stderr。

1

消息发到 stdout/stderr 和 syslog。

2

消息只发到 syslog。

hostlookup

在 ps_status 中启用主机名查找。

showportnumber

在 ps_status 中显示端口号。

nofsync

按后端禁用 fsync。

第 28 章 安全

数据库安全在若干个层面上加以处理：

- 数据库文件保护。数据库中存储的所有文件都受到保护，除Postgres超级用户账户外，任何其他账户都不能读取。
- 默认情况下，从客户端到数据库服务器的连接只允许通过本地 Unix 套接字进行，而不允许通过 TCP/IP 套接字。后端必须以`-i`选项启动才能允许非本地客户端连接。
- 可以通过`pg_hba.conf`文件（位于 `PG_DATA`中）按 IP 地址和/或用户名限制客户端连接。
- 客户端连接可以通过其他外部软件包进行认证。
- Postgres中的每个用户都被分配一个用户名和（可选的）口令。默认情况下，用户对不是他们创建的数据库没有写权限。
- 用户可以被分配到组，表访问可以基于组权限加以限制。

28.1. 用户认证

认证是后端服务器和postmaster 确保请求访问数据的用户确实是其声称的那个人的过程。所有调用Postgres的用户都会对照 `pg_user`类的内容进行检查，以确保他们被授权这样做。不过，用户实际身份的验证可以通过多种方式进行：

从用户 shell

> 从用户 shell 启动的后端服务器会先记录用户的（有效）用户 id，然后执行`setuid` 切换到用户*`postgres`*的用户 id。有效用户 id 被用作访问控制检查的依据。不进行其他认证。

从网络

> 如果Postgres系统构建为分布式，任何人都可以访问postmaster 进程的 Internet TCP 端口。DBA 配置 PGDATA 目录中的 pg_hba.conf 文件，根据发起连接的主机以及所连接的数据库来指定要使用的认证系统。可用认证系统的描述见pg_hba.conf(5)。当然，在 Unix 中基于主机的认证也不是万无一失的。意志坚定的入侵者也可能伪装来源主机。这些安全问题超出了Postgres的范围。

28.1.1. 基于主机的访问控制

基于主机的访问控制是 PostgreSQL 对哪些客户端被允许访问数据库以及这些客户端上的用户必须如何认证自己所实施的基本控制的名称。

每个数据库系统都包含一个名为 `pg_hba.conf`的文件（在其`PGDATA` 目录中），它控制谁可以连接到每个数据库。

每个访问数据库的客户端必须被`pg_hba.conf`中的某个条目覆盖。
否则来自该客户端的所有连接尝试都将被拒绝，并给出"User authentication failed"错误消息。

`pg_hba.conf`文件的一般格式是一组记录，每行一条。空行和以井号（"#"）开头的行被忽略。
一条记录由若干个以空格和/或制表符分隔的字段组成。

客户端可以使用 Unix 域套接字或 Internet 域套接字（即 TCP/IP）建立连接。使用 Unix 域套接字建立的连接由以下格式的记录控制：

```
local database authentication method
```

其中

database 指定该记录适用的数据库。值all 表示它适用于所有数据库。
authentication method 指定用户使用 Unix 域套接字连接到该数据库时必须使用的认证方法。各种方法在下面描述。

使用 Internet 域套接字建立的连接由以下格式的记录控制。

```
host database TCP/IP address TCP/IP mask authentication method
```

TCP/IP address 与指定的*TCP/IP mask* 以及连接客户端的 TCP/IP 地址分别做逻辑与。如果得到的两个值相等，则该记录用于此连接。如果一个连接匹配多条记录，则使用文件中最早的那条。*TCP/IP address*和 *TCP/IP mask* 都以点分十进制表示法指定。

如果一个连接未能匹配任何记录，则应用 reject认证方法（见下文）。

28.1.1.1. 认证方法

Unix 和 TCP/IP 域套接字都支持以下认证方法：

trust

无条件允许连接。

reject

无条件拒绝连接。

crypt

向客户端询问该用户的口令。口令以加密形式发送（使用crypt(3)），并与pg_shadow 表中保存的口令比较。如果口令匹配，则允许连接。

password

向客户端询问该用户的口令。口令以明文发送，并与pg_shadow表中保存的口令比较。如果口令匹配，则允许连接。可以在 password关键字后指定一个可选的口令文件，用它代替 pg_shadow 表来匹配提供的口令。见 pg_passwd。

以下认证方法仅支持 TCP/IP 域套接字：

krb4

使用 Kerberos V4 认证用户。

krb5

使用 Kerberos V5 认证用户。

ident

使用客户端上的 ident 服务器认证用户（RFC 1413）。可以在ident关键字后指定一个可选的映射名，它允许把 ident 用户名映射为 Postgres用户名。映射保存在文件 $PGDATA/pg_ident.conf中。

28.1.1.2. 示例

```
# Trust any connection via Unix domain sockets.
local trust
# Trust any connection via TCP/IP from this machine.
host all 127.0.0.1 255.255.255.255  trust
# We don't like this machine.
host all 192.168.0.10 255.255.255.0  reject
# This machine can't encrypt so we ask for passwords in clear.
host all 192.168.0.3 255.255.255.0  password
# The rest of this group of machines should provide encrypted
 passwords.
host all 192.168.0.0 255.255.255.0  crypt
```

28.2. 用户名和组

要定义一个新用户，运行 createuser实用程序。

要把一个或一组用户分配到一个新组，必须先定义组本身，再把用户分配到该组。在 Postgres中，这些步骤目前没有 create group命令支持。而是通过向 pg_group系统表插入适当的值来定义组，然后使用grant命令向组分配权限。

28.2.1. 创建用户

28.2.2. 创建组

目前还没有设置用户组的简便接口。你必须显式地插入/更新pg_group table。例如：

```
jolly=> insert into pg_group (groname, grosysid, grolist)
jolly=>     values ('posthackers', '1234', '{5443, 8261}');
INSERT 548224
jolly=> grant insert on foo to group posthackers;
CHANGE
jolly=>
```

pg_group中的字段有：

groname

组名。这是一个名称，应当纯粹由字母数字组成。不要包含下划线或其他标点。

grosysid

组 id。这是一个 int4。每个组的组 id 应当唯一。

grolist

属于该组的 pg_user id 列表。这是一个 int4[]。

28.2.3. 把用户分配到组

28.3. 访问控制

Postgres提供了一些机制，允许用户限制向其他用户提供的数据访问。

数据库超级用户

数据库超级用户（即设置了`pg_user.usesuper` 的用户）会静默地绕过下面描述的所有访问控制，但有两个例外：如果没有设置`pg_user.usecatupd`，
则不允许手动更新系统目录；而销毁系统目录（或修改其结构）任何时候都不允许。

访问权限

使用访问权限来限制对类的读取、写入和设置规则在grant/revoke(l)中介绍。

类的删除和结构修改

销毁或修改现有类结构的命令，例如`alter`、`drop table`和 `drop index`，只对类的所有者有效。如上所述，这些操作永远不允许用于系统目录。

28.4. 函数和规则

函数和规则允许用户把代码插入到后端服务器中，
而其他用户可能在不知情的情况下执行这些代码。因此，这两种机制都允许用户
特洛伊木马攻击他人而相对不受惩罚。唯一真正的保护是严格控制谁能定义函数（例如，向带 SQL 字段的关系写入）和规则。也建议在`pg_class`、`pg_user` 和`pg_group`上建立审计追踪和告警器。

28.4.1. 函数

除 SQL 之外的语言编写的函数都在后端服务器进程内部以用户*`postgres`*的权限运行（后端服务器以其真实和有效用户 id 都设置为 *`postgres`*的方式运行。用户有可能从受信任的函数内部更改服务器的内部数据结构。因此，除许多其他事情外，这样的函数可以绕过任何系统访问控制。
这是用户定义 C 函数固有的问题。

28.4.2. 规则

与 SQL 函数一样，规则总是以调用后端服务器的用户的身份和权限运行。

28.4.3. 注意事项

目前没有在Postgres内部显式支持加密数据的计划（不过没有什么能阻止用户在用户定义的函数内加密数据）。在前端/后端协议完全重写之前，也没有显式支持加密网络连接的计划。

用户名、组名和相关系统标识符（例如 `pg_user.usesysid`的内容）被假定在整个数据库中唯一。如果不是这样，可能出现不可预测的结果。

28.5. 安全的 TCP/IP 连接

作者

摘自Gene Selkov, Jr.[1] 1999-09-08 回答 Eric Marsden 一个问题的电子邮件。

可以使用ssh加密客户端与 Postgres服务器之间的网络连接。做得恰当的话，这应当能带来足够安全的网络连接。

ssh的文档提供了入门所需的大部分信息。请参阅 http://www.heimhardt.de/htdocs/ssh.html 以获得更好的理解。

分步说明只需两步。

通过 ssh 运行安全隧道

分步说明只需两步。

1. 像这样建立一条到后端机器的隧道：

   ```
   ssh -L 3333:wit.mcs.anl.gov:5432 postgres@wit.mcs.anl.gov
   ```

   -L 参数中的第一个数字 3333 是你这一端隧道的端口号。第二个数字 5432 是隧道的远端——你的后端正在使用的端口号。两个端口号之间的名称或地址属于服务器机器，ssh 的最后一个参数也是如此，后者还包含可选的用户名。如果没有用户名，ssh 将尝试你在客户端机器上当前登录所用的名字。你可以使用服务器机器接受的任何用户名，不一定非是与 postgres 相关的名字。

2. 现在你有了一个运行中的 ssh 会话，可以把 postgres 客户端连接到你本地主机上一步中指定的端口号。如果是 psql，你将需要另一个 shell，因为你在步骤 1中使用的 shell 会话现在被ssh占用了。

   ```
   psql -h localhost -p 3333 -d mpw
   ```

   注意，你必须指定-h参数才能使你的客户端使用 TCP 套接字而不是 Unix 套接字。如果你选择 5432 作为你这一端的隧道端口，可以省略端口参数。

[1] selkovjr@mcs.anl.gov

第 29 章 添加和删除用户

createuser 使特定的用户能够访问 Postgres。dropuser 移除用户并阻止他们访问 Postgres。

这些命令只影响与 Postgres 相关的用户；它们对用户在底层操作系统方面的其他权限或地位没有影响。

第 30 章 磁盘管理

30.1. 备选位置

可以在安装的默认位置之外的位置创建数据库。请记住，
所有数据库访问实际上都是通过数据库后端进行的，
因此所指定的任何位置都必须是后端可以访问的。

备选数据库位置由一个给出预期存储位置绝对路径的环境变量创建和引用。
这个环境变量必须在后端启动之前已经定义，并且必须可被 postgres 管理员账号写入。任何有效的环境变量名都可以用来引用一个备选位置，不过建议使用带 PGDATA 前缀的变量名，以避免混淆以及与其他变量的冲突。

> **注意**
>
> 在以前的 Postgres 版本中，也允许使用绝对路径名来指定一个备选存储位置。环境变量风格的指定方式更可取，因为它让站点管理员在管理磁盘存储时有更大的灵活性。
> 如果你更喜欢使用绝对路径，可以通过定义 "ALLOW_ABSOLUTE_DBPATHS" 并重新编译 Postgres 来做到。为此，要么把这一行
>
> ```
> #define ALLOW_ABSOLUTE_DBPATHS 1
> ```
>
> 加到文件 `src/include/config.h` 中，要么把
>
> ```
>  CFLAGS+= -DALLOW_ABSOLUTE_DBPATHS
> ```
>
> 指定到你的 `Makefile.custom` 中。

请记住，数据库的创建实际上是由数据库后端执行的。因此，
任何指定备选位置的环境变量都必须在后端启动之前已经定义。要定义一个指向 `/home/postgres/data` 的备选位置 PGDATA2，首先输入

```
% setenv PGDATA2 /home/postgres/data
```

来定义供后续命令使用的环境变量。通常，你会想在 Postgres 超级用户的 `.profile` 或 `.cshrc` 初始化文件中定义这个变量，以确保它在系统启动时被定义。任何环境变量都可以用来引用备选位置，不过最好让变量带 "PGDATA" 前缀，以消除混淆以及与其他变量冲突或覆盖其他变量的可能性。

要在 PGDATA2 中创建一个数据存储区，请确保 `/home/postgres` 已经存在并且可被 postgres 管理员写入。然后在命令行输入

```
% setenv PGDATA2 /home/postgres/data
% initlocation $PGDATA2
Creating Postgres database system directory /home/postgres/data

Creating Postgres database system directory /home/postgres/data/base
```

要测试新位置，输入下列命令创建一个数据库 test

```
% createdb -D PGDATA2 test
% dropdb test
```

第 31 章 管理数据库

如果 Postgres 的 postmaster 已经启动并正在运行，我们就可以创建一些数据库来做实验。这里我们描述管理数据库的基本命令。

31.1. 创建数据库

假设你想创建一个名为 mydb 的数据库。可以使用下面的命令完成：

```
% createdb dbname
```

Postgres 允许你在给定站点创建任意数量的数据库，并且你会自动成为你刚创建的数据库的数据库管理员。数据库名的第一个字符必须是字母，并且长度限制为 31 个字符。并非每个用户都有权限成为数据库管理员。如果 Postgres 拒绝为你创建数据库，那么站点管理员就需要授予你创建数据库的权限。如果发生这种情况，请咨询你的站点管理员。

31.2. 访问数据库

一旦构造好数据库，就可以通过以下方式访问它：

- 运行 Postgres 终端监视程序（psql），它允许你交互式地输入、编辑和执行 SQL 命令。
- 使用 `libpq` 子程序库编写 C 程序。这允许你从 C 中提交 SQL 命令，并把答案和状态消息返回给你的程序。这一接口在 PostgreSQL 程序员指南中有进一步讨论。

你也许想启动 psql 来试试本手册中的例子。可以通过输入以下命令为数据库 *dbname* 激活它：

```
psql dbname
```

你会看到下面的欢迎消息：

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

dbname=>
```

这个提示表明终端监视程序正在监听你，你可以在终端监视程序维护的工作区中输入 SQL 查询。psql 程序响应以反斜杠字符 "\" 开头的转义码。例如，你可以通过输入以下内容获得各种 Postgres SQL 命令的语法帮助：

```
dbname=> \h
```

一旦在工作区中输入完查询，你可以通过输入以下内容把工作区的内容传递给 Postgres 服务器：

```
dbname=> \g
```

这告诉服务器处理该查询。如果你用分号终止查询，就不需要 backslash-g。psql 会自动处理以分号终止的查询。要从文件中读取查询而不是交互式地输入，输入：

```
dbname=> \i filename
```

要退出 psql 并返回 Unix，输入

```
dbname=> \q
```

psql 将退出并把你返回到命令 shell。（关于更多转义码，在监视提示符下输入 backslash-h。）空白（即空格、制表符和换行符）可以在 SQL 查询中自由使用。单行注释用两个连字符（"--"）表示。连字符之后直到行尾的所有内容都被忽略。多行注释以及行内注释用 "/* ... */" 表示，这是从 Ingres 借来的约定。

31.3. 删除数据库

如果你是数据库 mydb 的数据库管理员，可以使用下面的 Unix 命令销毁它：

```
% dropdb dbname
```

这个动作会物理地删除与该数据库关联的所有 Unix 文件，而且无法撤销，因此只应在深思熟虑之后进行。

也可以在一个 SQL 会话中使用下面的命令销毁数据库：

```
> drop database dbname
```

31.4. 备份与恢复

小心

每个数据库都应当定期备份。由于 Postgres 在文件系统中管理它自己的文件，因此不建议依靠你文件系统的系统备份来做数据库备份；不能保证恢复之后文件处于可用的、一致的状态。

Postgres 提供两个用来备份系统的实用程序：pg_dump 用于备份单个数据库，pg_dumpall 用于一步备份你的整个安装。

可以使用下面的命令备份单个数据库：

```
% pg_dump dbname > dbname.pgdump
```

恢复时使用

```
cat dbname.pgdump | psql dbname
```

这项技术可以用来把数据库移动到新位置，以及重命名现有数据库。

31.4.1. 大型数据库

作者

由 Hannu Krosing[1] 于 1999-06-19 撰写。

由于 Postgres 允许表大于系统的最大文件大小，把表转储到一个文件可能有问题，因为得到的文件很可能大于你的系统所允许的最大尺寸。

由于 pg_dump 写到标准输出，你可以直接使用标准 *nix 工具绕过这个潜在问题：

- 使用压缩的转储：

```
% pg_dump dbname | gzip > filename.dump.gz
```

  重载时使用

```
% createdb dbname
% gunzip -c filename.dump.gz | psql dbname
```

  或

```
% cat filename.dump.gz | gunzip | psql dbname
```

- 使用 split：

```
% pg_dump dbname | split -b 1m - filename.dump.
```

  重载时使用

```
% createdb dbname
% cat filename.dump.* | pgsql dbname
```

当然，文件的名字（*filename*）和 pg_dump 输出的内容不必与数据库的名字匹配。而且，恢复后的数据库可以有任意的新名字，因此这个机制也适用于重命名数据库。

[1] hannu@trust.ee

第 32 章 故障排除

32.1. postmaster 启动失败

postmaster 启动失败有几个常见的原因。检查 postmaster 的日志文件，或者手工启动它（不重定向标准输出或标准错误）看看会出现什么错误消息。其中一些可能的错误消息是不言自明的，但也有一些不是：

```
FATAL: StreamServerPort: bind() failed: Address already in use
        Is another postmaster already running on that port?
```

这通常就像消息所提示的那样：你不小心在一个已经运行着 postmaster 的端口上又启动了第二个 postmaster。但是，如果内核错误消息不是"Address already in use"或类似措辞，则可能是别的问题。例如，试图在一个保留端口号上启动 postmaster 可能会得到类似

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
        Is another postmaster already running on that port?
```

```
IpcMemoryCreate: shmget failed (Invalid argument) key=5440001, size=
83918612, permission=600
FATAL 1:  ShmemCreate: cannot create region
```

这样的消息可能意味着你的内核对共享内存区大小的限制小于 Postgres 试图创建的缓冲区。（或者它可能意味着你的内核根本没有配置 SysV 风格的共享内存支持。）作为临时的变通办法，你可以尝试用比正常情况更少的缓冲区数目（-B 开关）启动 postmaster。不过你最终还是应该重新配置内核以增大允许的共享内存大小。如果在同一台机器上启动多个 postmaster 而它们的总空间请求超过内核限制，你也可能看到这条消息。

```
IpcSemaphoreCreate: semget failed (No space left on device) key=
5440026, num=16, permission=600
```

这样的消息并不意味着你的磁盘空间用完了；它意味着你的内核对 SysV 信号量数目的限制小于 Postgres 想要创建的数目。如上所述，你可以通过用较少的后端进程数目（-N 开关）启动 postmaster 来绕过这个问题，但最终你还是应该增大内核限制。

32.2. 客户端连接问题

一旦你有了一个正在运行的 postmaster，用客户端应用连接它仍然可能因为多种原因而失败。这里给出的示例错误消息来自基于较新版本 libpq 的客户端——基于其他接口库的客户端可能产生信息量或多或少不同的其他消息。

```
connectDB() -- connect() failed: Connection refused
Is the postmaster running (with -i) at 'server.joe.com' and accepting
 connections on TCP/IP port '5432'?
```

这是常见的"我找不到可以对话的 postmaster"失败。尝试 TCP/IP 通信时它像上面这样，而尝试通过 Unix 套接字连接本地 postmaster 时则像这样：

```
connectDB() -- connect() failed: No such file or directory
Is the postmaster running at 'localhost' and accepting connections on
 Unix socket '5432'?
```

最后一行有助于确认客户端是否在尝试连接它应该连接的地方。如果那里确实没有 postmaster 在运行，内核错误消息通常会是 "Connection refused"或"No such file or directory"，如图所示。（特别重要的是要认识到，在这种情况下的"Connection refused" 并不意味着 postmaster 收到了你的连接请求并拒绝了它——那种情况会产生不同的消息，如下所示。）其他错误消息如"Connection timed out"可能指示更根本的问题，比如缺乏网络连通性。

```
No pg_hba.conf entry for host 123.123.123.123, user joeblow, database
 testdb
```

如果你成功联系到了一个 postmaster 但它不想和你交谈，最有可能得到的就是这条消息。正如消息所提示的，postmaster 拒绝了连接请求，因为它在它的 pg_hba.conf 配置文件中没有找到授权条目。

```
Password authentication failed for user 'joeblow'
```

这样的消息表明你联系到了 postmaster，它愿意和你交谈，但前提是你通过 pg_hba.conf 文件中指定的认证方法。检查你提供的口令，或者如果错误消息提到 Kerberos 或 IDENT 认证类型之一，检查你的 Kerberos 或 IDENT 软件。

```
FATAL 1:  SetUserId: user 'joeblow' is not in 'pg_shadow'
```

这是认证失败的另一种变体：对给定的用户名还没有执行过 Postgres 的 create_user 命令。

```
FATAL 1:  Database testdb does not exist in pg_database
```

这个 postmaster 控制之下没有那个名字的数据库。注意，如果你不指定数据库名，默认使用你的 Postgres 用户名，而这未必是正确的选择。

```
NOTICE:  Unrecognized variable client_encoding
```

这不是错误；实际上它完全无害。如果你用一个编译时带 MULTIBYTE 支持的客户端去连接一个编译时不带该支持的服务器，就会在启动时看到这条消息。
（客户端试图告诉服务器它想要的字符集编码，但服务器根本不知道它在说什么。）
如果这条消息让你烦恼，可以使用一个与服务器编译选项相同的客户端。

32.3. 调试消息

postmaster 偶尔会打印出一些在故障排除时往往很有帮助的消息。如果你希望查看来自 postmaster 的调试消息，可以用 -d 选项启动它并把输出重定向到日志文件：

```
% postmaster -d > pm.log 2>&1 &
```

如果你不希望看到这些消息，可以键入

```
% postmaster -S
```

这样 postmaster 就会"S"静默。这种情况下不需要与号（"&"），因为指定 -S 时 postmaster 会自动脱离终端。

32.3.1. pg_options

> **注意**
>
> 由 Massimo Dal Zotto[1] 贡献。

可选文件 `data/pg_options` 包含后端用来控制跟踪消息和其他后端可调参数的运行时选项。这个文件的有趣之处在于，后端在收到 SIGHUP 信号时会重新读取它，因此无需重启 Postgres 就可以即时更改运行时选项。这个文件中指定的选项可以是跟踪包（`backend/utils/misc/trace.c`）使用的调试标志，也可以是后端用来控制其行为的数值参数。
新的选项和参数必须在 `backend/utils/misc/trace.c` 和 `backend/include/utils/trace.h` 中定义。

pg_options 也可以用 Postgres 的 `-T` 开关来指定：

```
postgres options -T "verbose=2,query,hostlookup-"
```

用于打印错误和调试消息的函数现在可以利用 syslog(2) 设施了。打印到 stdout 或 stderr 的消息带有一个还包含后端 pid 的时间戳前缀：

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

这种格式提高了日志的可读性，使人们能准确了解哪个后端在什么时间做什么。
它也使得编写简单的 awk 或 perl 脚本来监视日志以检测数据库错误或问题、或计算事务时间统计变得更容易。

打印到 syslog 的消息使用日志设施 LOG_LOCAL0。syslog 的使用可以用 syslog pg_option 来控制。遗憾的是，许多函数直接调用 `printf()` 把消息打印到 stdout 或 stderr，这样的输出不能被重定向到 syslog 也不能带时间戳。最好把所有对 printf 的调用都替换为 PRINTF 宏、把输出到 stderr 改为使用 EPRINTF，这样我们就能以统一的方式控制所有输出。

`pg_options` 文件的格式如下：

```
# comment
option=integer_value  # set value for option
option                # set option = 1
```

[1] mailto:dz@cs.unitn.it

```
option+                 # set option = 1
option-                 # set option = 0
```

注意，*keyword* 也可以是 backend/utils/misc/trace.c 中定义的选项名的缩写。

完整的选项关键字及其可能取值的列表请参考使用 pg_options。

第 33 章 数据库恢复

本节尚待编写。有人愿意吗？

第 34 章 回归测试

回归测试的说明与分析。

PostgreSQL 回归测试是对嵌入在 PostgreSQL 中的 SQL 实现的一套全面测试。它们既测试标准 SQL 操作，也测试 PostgreSQL 的扩展能力。

回归测试有两种不同的运行方式："sequential"（顺序）方法和 "parallel"（并行）方法。顺序方法依次运行每个测试脚本，而并行方法会启动多个服务器进程来并行运行成组的测试。
并行测试可以让人确信进程间通信和锁机制工作正常。另一个关键区别是，
顺序测试过程使用一个已安装的 postmaster，而并行测试过程测试的是已构建但尚未安装的系统。（并行测试脚本实际上会安装到一个临时目录中，并在那里启动一个私有的 postmaster。）

某些安装正确且功能完备的 PostgreSQL 安装可能会因浮点表示和时区支持的特有问题而在其中一些回归测试上"失败"。这些测试目前是通过与参考系统生成的输出做简单的 diff 比较来评估的，因此结果对细微的系统差异很敏感。当某项测试被报告为"失败"时，
务必检查期望结果与实际结果之间的差异；你很可能会发现这些差异并不重要。

回归测试最初由 Jolly Chen 和 Andrew Yu 开发，并经 Marc Fournier 和 Thomas Lockhart 大幅修订/重新打包。从 PostgreSQL v6.1 起，回归测试对每个正式发行版都是最新的。

34.1. 回归测试环境

下面的回归测试说明假定以下条件（除非另有说明）：

- 命令是 Unix 兼容的。见下面的说明。
- 除非另有说明，均使用默认值。
- 用户 postgres 是 Postgres 超级用户。
- 源代码路径是 /usr/src/pgsql（其他路径也是可能的）。
- 运行时路径是 /usr/local/pgsql（其他路径也是可能的）。

通常，回归测试应当以 postgres 用户运行，因为 'src/test/regress' 目录及其子目录为 postgres 用户所有。如果你以其他用户运行回归测试，那么 'src/test/regress' 目录树必须可被该用户写入。

过去必须把系统时区设置为 PST 来运行 postmaster，但现在不再需要了。你可以在正常的 postmaster 配置下运行回归测试。测试脚本会设置 PGTZ 环境变量以确保依赖时区的测试产生期望的结果。但是，你的系统必须提供对 PST8PDT 时区的库支持，否则依赖时区的测试将会失败。要验证你的机器确实支持这一点，请键入：

```
setenv TZ PST8PDT
date
```

上面的 "date" 命令应当以 PST8PDT 时区返回当前系统时间。如果 PST8PDT 数据库不可用，你的系统可能返回的是 GMT 时间。如果 PST8PDT 时区不可用，你可以显式设置时区规则：

```
setenv PGTZ PST8PDT7,M04.01.0,M10.05.03
```

回归测试区域的目录布局如下：

表 34.1. 目录布局

目录	描述
目录	描述

目录	描述
input	使用 `make all` 转换成 `sql` 子目录中某些 `.sql` 文件的源文件。
output	使用 `make all` 转换成 `expected` 子目录中 `.out` 文件的源文件。
sql	用于执行回归测试的 `.sql` 文件。
expected	表示我们期望结果是什么样子的 `.out` 文件。
results	包含结果实际是什么样子的 `.out` 文件。也用作表复制测试的临时存储。
tmp_check	由并行测试脚本创建的临时安装。

34.2. 回归测试过程

这些命令是在使用 bash shell 的 RedHat Linux 4.2 版上测试的。除非另有说明，它们在大多数系统上可能都能工作。像 `ps` 和 `tar` 这样的命令在各个平台上应使用的选项差异极大。键入这些命令之前请运用常识。

Postgres 回归测试

1. 准备回归测试所需的文件：

   ```
   cd /usr/src/pgsql/src/test/regress
   gmake clean
   gmake all
   ```

   如果这是你第一次运行测试，可以跳过 "gmake clean"。

   这一步会把一个带有 PostgreSQL 扩展函数的 C 程序编译成共享库。还会为需要本地化的测试创建本地化的 SQL 脚本和输出比较文件。本地化会把源文件中的宏替换为绝对路径名和用户名。

2. (可选的) 如果你打算使用测试已安装 postmaster 的 "sequential"（顺序）测试过程，请确保 postmaster 正在运行。如果它还没有运行，在一个可用的窗口中键入以下命令启动 postmaster

   ```
   postmaster
   ```

   或者键入以下命令让 postmaster 守护进程在后台运行

   ```
   cd
   nohup postmaster > regress.log 2>&1 &
   ```

   后者可能更可取，因为回归测试日志会相当长（在 Postgres 7.0 中约 60K），如果出现问题，你可能想查看它以寻找线索。

   注意

   不要从 root 账户运行 `postmaster`。

3. 运行回归测试。对于顺序测试，键入

```
cd /usr/src/pgsql/src/test/regress
gmake runtest
```

对于并行测试，键入

```
cd /usr/src/pgsql/src/test/regress
gmake runcheck
```

顺序测试只是使用你已经在运行的 postmaster 运行测试脚本。并行测试会把 Postgres 完整安装到一个临时目录，在其中启动一个私有 postmaster，然后运行测试脚本。最后它会杀掉这个私有 postmaster（但临时目录不会被自动删除）。

4. 你应当在屏幕上（同时也写入文件 ./regress.out）看到一系列说明哪些测试通过、哪些测试失败的语句。请注意，由于平台相关的变化，某些测试"失败"可能是正常的。
关于如何判断一个"失败"是否重要，详见下一节。

   某些测试，特别是 "numeric"，可能需要一段时间，尤其是在较慢的平台上。请保持耐心。

5. 运行测试并检查结果后，键入

```
cd /usr/src/pgsql/src/test/regress
gmake clean
```

以回收测试使用的临时磁盘空间。如果你运行的是顺序测试，还要键入

```
dropdb regression
```

34.3. 回归分析

回归测试的实际输出位于 `./results` 目录中的文件里。测试脚本使用 diff 把每个输出文件与存储在 `./expected` 目录中的参考输出进行比较。任何差异都会保存到 `./regression.diffs` 中供你检查。（如果愿意，你也可以自己运行 diff。）

这些文件可能不会完全一致。测试脚本会把任何差异报告为"失败"，
但这种差异可能源于错误消息措辞、数学库行为等方面的细微跨系统差异。这类"失败"并不表示 Postgres 有问题。

因此，有必要检查每个"失败"测试的实际差异，以确定是否真的存在问题。
下面几段试图为判断差异是否重要提供一些指导。

34.3.1. 错误消息差异

某些回归测试包含故意构造的非法输入值。错误消息可能来自 Postgres 代码，也可能来自宿主平台的系统例程。在后一种情况下，消息会因平台而异，但应反映相近的信息。
这类消息差异会导致回归测试显示为"失败"，但可以通过检查确认其有效性。

34.3.2. 日期和时间差异

大多数日期和时间结果依赖于时区环境。参考文件是为 PST8PDT 时区（加利福尼亚州伯克利）生成的，如果测试不以该时区设置运行，就会出现明显的失败。回归测试驱动程序会把环境变量 PGTZ 设置为 PST8PDT 以确保正确的结果。

如果你在夏令时切换日当天或其前一天或后一天运行测试，"timestamp" 测试中的一些查询会失败。这些查询假定昨天午夜、
今天午夜和明天午夜之间的间隔正好是二十四小时……如果其间夏令时生效或失效，
这就是错误的。

有些系统似乎不接受显式设置本地时区规则的推荐语法；在这类机器上你可能需要使用不同的 PGTZ 设置。

一些使用较老时区库的系统无法对 1970 年之前的日期应用夏令时修正，导致 1970 年前的 PDT 时间显示为 PST。这会导致测试结果中出现局部差异。

34.3.3. 浮点差异

某些测试涉及从表列计算 64 位（`float8`）数。已经观察到涉及 `float8` 列数学函数的结果存在差异。float8 和 geometry 测试尤其容易在不同平台之间出现微小差异。需要人工目测比较来确定这些差异的真实意义，它们通常出现在小数点右侧第 10 位。

某些系统发出 pow() 和 exp() 错误的方式与当前 Postgres 代码所预期的机制不同。

34.3.4. 多边形差异

若干测试涉及对加利福尼亚州奥克兰/伯克利街道地图地理数据的操作。
地图数据被表示为多边形，其顶点用一对 `float8` 数（十进制纬度和经度）表示。首先创建一些表并装入地理数据，然后创建一些使用多边形相交操作符（##）连接两个表的视图，
再对视图做一次 select。在比较不同平台的结果时，差异出现在小数点右侧第 2 或第 3 位。出现这些问题的 SQL 语句是：

```
QUERY: SELECT * from street;
QUERY: SELECT * from iexit;
```

34.3.5. 随机差异

"random" 测试脚本中至少有一种情况旨在产生随机结果。这会使 random 回归测试偶尔（也许每五到十次尝试一次）失败。键入

```
diff results/random.out expected/random.out
```

应当只产生一行或几行差异。除非随机测试在反复尝试中总是失败，否则不必担心。
（另一方面，如果在回归测试的多次尝试中随机测试从未被报告失败，你可能确实应该担心。）

34.3.6. "expected" 文件

`./expected/*.out` 文件改编自 Jolly Chen 等人提供的原始整体式 `expected.input` 文件。在各种开发机器上生成的这些文件的新版本经过仔细（？）检查后被替换了进来。许多开发机器运行的是 Ix86 硬件上的某种 Unix 变体（FreeBSD、Linux 等）。原始的 `expected.`

input 文件是在一台 SPARC Solaris 2.4 系统上使用 postgres5-1.02a5.tar.gz 源树创建的。它与一台 I386 Solaris 2.4 系统上创建的文件比较后，差异仅在于浮点多边形小数点右侧第 3 位。原始的 sample.regress.out 文件来自 Jolly Chen 构建的 postgres-1.01 发行版。它可能是在一台 DEC ALPHA 机器上创建的，因为 postgres-1.01 发行版中的 Makefile.global 有 PORTNAME=alpha。

34.4. 平台相关的比较文件

由于某些测试固有地会产生平台相关的结果，
我们提供了一种提供平台相关结果比较文件的方法。通常，同一种变化适用于多个平台；
与其为每个平台提供一个单独的比较文件，不如用一个映射文件来定义使用哪个比较文件。
因此，要消除特定平台上虚假的测试"失败"，你必须选择或制作一个变体结果文件，
然后向映射文件（即 "resultmap"）中添加一行。

映射文件中的每一行的形式是

```
testname/platformnamepattern=comparisonfilename
```

测试名就是特定回归测试模块的名称。平台名模式是一个 expr(1) 风格的模式（即一个在开头隐式带有 ^ 锚点的正则表达式）。它与 config.guess 输出的平台名进行匹配。比较文件名是替代结果比较文件的名称。

例如：int2 回归测试包含一个故意输入的、大到无法放入 int2 的值。所产生的具体错误消息是平台相关的；我们的参考平台发出

```
   ERROR:  pg_atoi: error reading "100000": Numerical result out of
range
```

但相当多的其他 Unix 平台发出

```
   ERROR:  pg_atoi: error reading "100000": Result too large
```

因此，我们提供了一个变体比较文件 int2-too-large.out，其中包含这种拼写的错误消息。为了在 HPPA 平台上消除虚假的"失败"消息，resultmap 中包含了

```
int2/hppa=int2-too-large
```

它会在任何 config.guess 的输出以 'hppa' 开头的机器上触发。resultmap 中的其他行为其他合适的平台选择变体比较文件。

第 35 章 发布说明

35.1. 版本 7.0.3

发行日期 2000-11-04。本次发布包含对 7.0.2 的多方面修复。

35.1.1. 迁移到版本 7.0.3

对于运行 7.0.* 的用户，不需要进行转储/恢复。

35.1.2. 变更

Jdbc 修复 (Peter)
大对象修复 (Tom)
修复 COPY WITH OIDS 的内存泄漏 (Tom)
修复反向索引扫描 (Tom)
修复 SELECT ... FOR UPDATE, 使其检查重复键 (Hiroshi)
为 configure 添加 --enable-syslog (Marc)
修复罕见情况下后端退出时中止事务的问题 (Tom)
修复启用多字节时的 psql \l+ (Tatsuo)
允许 PL/pgSQL 接受非 ASCII 标识符 (Tatsuo)
使 vacuum 总是刷新缓冲区 (Tom)
修复以允许在等待锁时取消 (Hiroshi)
修复用户认证代码中的内存分配问题 (Tom)
移除对 int4out() 的错误使用 (Tom)
修复 COALESCE 或 BETWEEN 中多个子查询的问题 (Tom)
修复某些情况下堆打开时触发器的失败 (Jeroen van Vianen)
修复不等号选择率的错误 (Tom)
修复 strcmp() 的错误使用 (Tom)
修复存储管理器访问文件末尾之后条目的缺陷 (Tom)
修复以在所有 smgr elog 消息中包含内核 errno 消息 (Tom)
修复构建时 '.' 不在 PATH 中的问题 (SL Baur)
修复文件描述符耗尽错误 (Tom)
修复以使 pg_dump 转储函数的 'iscachable' 标志 (Tom)
修复 Append 节点目标列表中的子选择 (Tom)
修复归并连接计划 (Tom)
修复 TRUNCATE 在带索引关系上的失败 (Tom)
避免写错误时整个数据库重启 (Hiroshi)
修复 nodeMaterial 以通过重算输出来遵守 chgParam (Tom)
修复元组的源和目标位于同一页时 VACUUM 移动更新元组链的问题 (Tom)
修复 user.c 的 CommandCounterIncrement (Tom)
修复 to_char() 中 AM/PM 边界的问题 (Karel Zak)
修复 TIME 聚合处理 (Tom)
修复 to_char() 以避免在 NULL 输入时核心转储 (Tom)
缓冲区修复 (Tom)
修复向 char() 数据类型插入/复制较长多字节字符串的问题 (Tatsuo)
修复中止时后端的崩溃 (Tom)

35.2. 版本 7.0.2

发行日期 2000-06-05。这是对 7.0.1 的重新打包，并添加了文档。

35.2.1. 迁移到版本 7.0.2

对于运行 7.* 的用户，不需要进行转储/恢复。

35.2.2. 变更

向 tarball 添加了文档。

35.3. 版本 7.0.1

发行日期 2000-06-01。这是针对 7.0 的清理发行版。

35.3.1. 迁移到版本 7.0.1

对于运行 7.0 的用户，不需要进行转储/恢复。

35.3.2. 变更

修复许多 CLUSTER 失败 (Tom)
允许 ALTER TABLE RENAME 用于索引 (Tom)
修复 plpgsql 以处理 datetime->timestamp 和 timespan->interval (Bruce)
新的 configure --with-setproctitle 开关以使用 setproctitle() (Marc、Bruce)
修复 6.5.3 以来 ResultSet 的差一错误，以及更多修复。
jdbc ResultSet 修复 (Joseph Shraibman)
优化器调整 (Tom)
为 pgaccess 修复 create user
修复 UNLISTEN 失败
IRIX 修复 (David Kaelbling)
QNX 修复 (Andreas Kardos)
降低 COPY IN 的锁级别 (Tom)
更改 libpqeasy 以使用 PQconnectdb() 风格的参数 (Bruce)
修复 pg_dump 以处理 OID 索引 (Tom)
修复小的内存泄漏 (Tom)
Solaris 上 createdb/dropdb 的修复 (Tatsuo)
修复非阻塞连接 (Alfred Perlstein)
修复 RENAME TABLE 失败后的不当恢复 (Tom)
安装时将 pg_ident.conf.sample 复制到 /lib 目录 (Bruce)
添加 SJIS UDC (NEC selection IBM 汉字) 支持 (Eiji Tokuya)
修复过长的 syslog 消息 (Tatsuo)
修复带引号的过长索引的问题 (Tom)
JDBC ResultSet.getTimestamp() 修复 (Gregory Krasnow & Floyd Marinescu)
ecpg 变更 (Michael)

35.4. 版本 7.0

发布于 2000-05-08。此发行版包含许多方面的改进，展示了 PostgreSQL 的持续成长。7.0 中的改进和修复比以往任何发行版都多。开发者们相信这是迄今最好的发行版；我们尽力只发布稳固的发行版，这一版也不例外。

此发行版的主要变更：

外键

外键现已实现，PARTIAL MATCH 外键除外。许多用户一直在请求这一特性，我们很高兴能提供它。

优化器全面翻修

延续一年前开始的工作，优化器得到了改进，能够更好地选择查询计划，
性能更快且内存使用更少。

更新的 psql

我们的交互式终端监视器 psql 已经更新，增加了多种新特性。详情参见 psql 手册页。

连接语法

现在支持 SQL92 连接语法，不过此发行版中仅支持 `INNER JOIN`。`JOIN`、`NATURAL JOIN`、`JOIN/USING` 和 `JOIN/ON` 都可用，列相关名也可用。

即将到来的特性

在 7.1 或 7.2 中，我们计划提供外连接、超长行的存储以及预写日志系统。

35.4.1. 迁移到版本 7.0

希望从任何以前的 Postgres 发行版迁移数据的用户，需要使用 pg_dump 进行转储/ 恢复。对于从 6.5.* 升级的用户，也可以改用 pg_upgrade 升级到此发行版；不过，完整的转储/重载安装始终是升级最稳健的方法。

新发行版需要考虑的接口和兼容性问题包括：

- 日期/时间类型 `datetime` 和 `timespan` 已被 SQL92 定义的类型 `timestamp` 和 `interval` 取代。尽管已经做了一些工作来简化过渡 —— 允许 Postgres 识别已废弃的类型名并将它们转换为新类型名 —— 但这一机制对你现有的应用程序可能无法完全透明。
- 优化器在查询代价估算方面得到了实质性改进。在某些情况下，
由于优化器为首选计划做出了更好的选择，查询时间会缩短。但是，
在少数情况下（通常涉及病态的数据分布），你的查询时间可能上升。如果你处理大量数据，
可能需要检查你的查询以验证性能。
- JDBC 和 ODBC 接口已升级并扩展。
- 字符串函数 `CHAR_LENGTH` 现在是原生函数。以前的版本将它转换为对 `LENGTH` 的调用，这可能与其他实现 `LENGTH` 的类型（例如几何类型）产生歧义。

35.4.2. 变更

缺陷修复

防止函数调用超出参数数量上限（`Tom`）

改进 CASE 结构 (Tom)
修复 SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
修复 SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
修复 GROUP BY 扫描缺陷 (Tom)
SQL 语法处理的改进 (Tom)
修复涉及 INSERT ... SELECT ... 的视图的问题 (Tom)
修复 SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
修复 INSERT ... SELECT 中的子选择 (Tom)
禁止 INSERT ... SELECT ... ORDER BY (Tom)
修复大于 2GB 的关系 (包括 vacuum) 的问题
改进系统表变更向其他后端的传播 (Tom)
改进用户表变更向其他后端的传播 (Tom)
修复复杂情况下临时表的处理 (Bruce、Tom)
允许在表打开时锁定表，改进并发可靠性 (Tom)
在 pg_dump 中正确为序列名加引号 (Ross J. Reedstrom)
防止他人正在访问时 DROP DATABASE
防止 GROUP BY 未处理任何行时返回行 (Tom)
修复 'SELECT COUNT(1) FROM table WHERE ...' (无匹配 WHERE 的行时) (Tom)
修复 pg_upgrade 使其适用于 MVCC (Tom)
修复 SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
修复 "f1 datetime DEFAULT 'now'" (Tom)
修复 DEFAULT 中使用 CURRENT_DATE 的问题 (Tom)
允许纯注释行，以及 ;;; 行。 (Tom)
改进磁盘写入失败、磁盘满之后的恢复 (Hiroshi)
修复表在 FROM 中提到但未连接的情况 (Tom)
允许不带聚合函数的 HAVING 子句 (Tom)
修复 "--" 注释且无末尾换行的问题，如 perl 接口中所见
改进 pg_dump 失败错误报告 (Bruce)
允许排序和哈希超过 2GB 文件大小 (Tom)
修复 pg_dump 转储继承规则的问题 (Tom)
修复 NULL 处理比较的问题 (Tom)
修复失败的 CREATE/DROP 命令导致的不一致状态 (Hiroshi)
修复带连字符的数据库名
防止 DROP INDEX 干扰其他后端 (Tom)
修复 verify_password() 中的文件描述符泄漏
修复 "Unable to identify an operator =$" 问题
修复 ODBC，使得启用 CommLog 和 Debug 时不再段错误 (Dirk Niggemann)
修复递归 exit 调用 (Massimo)
修复超长时区的问题 (Jeroen van Vianen)
使 pg_dump 保留主键信息 (Peter E)
防止带单引号的数据库名 (Peter E)
禁止在事务内 DROP DATABASE (Peter E)
ecpg 内存泄漏修复 (Stephen Birch)
修复 SELECT null::text、SELECT int4fac(null) 和 SELECT 2 + (null) (Tom)
Y2K 时间戳修复 (Massimo)
修复 VACUUM 的 'HEAP_MOVED_IN was not expected' 错误 (Tom)
修复带空格表/列的视图的问题 (Tom)
禁止对索引授权 (Peter E)
修复出错时自旋锁卡住的问题 (Hiroshi)
修复 Linux 上的 ipcclean
修复 NULL 约束条件的处理 (Tom)
修复 odbc 驱动中的内存泄漏 (Nick Gorham)
修复 UNION 表上的权限检查 (Tom)

修复以允许 SELECT 'a' LIKE 'a' (Tom)
修复 SELECT 1 + NULL (Tom)
CHAR 的若干修复
修复 numeric 类型的 log() (Tom)
废弃 ':' 和 ';' 操作符
允许清理临时表
禁止与新列同名的继承列
磁盘空间耗尽时恢复或强制失败 (Hiroshi)
修复 INSERT INTO ... SELECT 中 AS 列与结果列匹配的问题
修复 INSERT ... SELECT ... GROUP BY 按目标列而不是源列分组的问题 (Tom)
修复 CREATE TABLE test (a char(5) DEFAULT text '', b int4) 与 INSERT (Tom)
修复带 LIMIT 的 UNION
修复 CREATE TABLE x AS SELECT 1 UNION SELECT 2
修复 CREATE TABLE test(col char(2) DEFAULT user)
修复 CREATE TABLE ... DEFAULT 中类型不匹配的问题
修复 SELECT * FROM pg_class where oid in (0,-1)
修复 SELECT COUNT('asdf') FROM pg_class WHERE oid=12
防止能创建数据库的用户修改 pg_database 表 (Peter E)
修复 btree, 当键 > 1/2 (页 - 开销) 时给出有用的 elog (Tom)
修复向 DECIMAL(4,4) 字段插入 0.0 的问题 (Tom)

增强

新的 CLI 接口头文件 sqlcli.h, 基于 SQL3/SQL98
移除查询长度的所有限制, 行长度限制仍存在 (Tom)
将 jdbc 协议更新到 2.0 (Jens Glaser <jens@jens.de>)
添加 TRUNCATE 命令以快速截断关系 (Mike Mascari)
修复以赋予超级用户和 createdb 用户正确的目录更新权限 (Peter E)
允许 ecpg 布尔变量取 NULL 值 (Christof)
当无 NULL 指示符的变量取 NULL 值时发出 ecpg 错误 (Christof)
允许用 ^C 取消 COPY 命令 (Massimo)
添加 SET FSYNC 和 SHOW PG_OPTIONS 命令 (Massimo)
动态加载 C 函数的函数名重载 (Frankpitt)
向 libpq++ 添加 CmdTuples() (Vince)
新的 CREATE CONSTRAINT TRIGGER 和 SET CONSTRAINTS 命令 (Jan)
允许 CREATE FUNCTION/WITH 子句用于所有语言类型
configure --enable-debug 添加 -g (Peter E)
configure --disable-debug 移除 -g (Peter E)
允许更复杂的默认表达式 (Tom)
第一个真正可用的外键约束触发器功能 (Jan)
添加 FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
添加 FOREIGN KEY ... MATCH <unspecified> 引用动作 (Don Baccus)
允许对 ctid (物理堆位置) 使用 WHERE 限制 (Hiroshi)
将 pginterface 从 contrib 移到 interface 目录并更名为 pgeasy (Bruce)
更改 pgeasy connectdb() 的参数顺序 (Bruce)
要求 SELECT DISTINCT 目标列表包含所有 ORDER BY 列 (Tom)
添加 Oracle 的 COMMENT ON 命令 (Mike Mascari <mascarim@yahoo.com>)
libpq 的 PQsetNoticeProcessor 函数现在返回之前的钩子 (Peter E)
防止 PQsetNoticeProcessor 被设置为 NULL (Peter E)
使 COPY 中的 USING 变为可选 (Bruce)
允许目标列表中的子选择 (Tom)
允许比较操作符左侧的子选择 (Tom)
新的并行回归测试 (Jan)

将后端侧 COPY 写文件的权限从 666 改为 644 (Tom)
即使 PGDATA 目录已存在也强制其权限安全 (Tom)
添加 psql LASTOID 变量以返回最后插入的 oid (Peter E)
允许并发 vacuum 并移除 pg_vlock vacuum 锁文件 (Tom)
为 vacuum 添加权限检查 (Peter E)
libpq 新函数支持异步连接: PQconnectStart()、
  PQconnectPoll()、PQresetStart()、PQresetPoll()、PQsetenvStart()、
  PQsetenvPoll()、PQsetenvAbort (Ewan Mellor)
新的 libpq PQsetenv() 函数 (Ewan Mellor)
create/alter user 扩展 (Peter E)
$PGDATA 下新的 postmaster.pid 和 postmaster.opts (Tatsuo)
用于 create/drop user/db 的新脚本 (Peter E)
psql 全面翻修 (Peter E)
向 libpq 接口添加 const (Peter E)
新的 libpq 函数 PQoidValue (Peter E)
显示导致 GROUP BY 问题的特定非聚合 (Tom)
使对 pg_shadow 的更改重建 pg_pwd 文件 (Peter E)
添加 aggregate(DISTINCT ...) (Tom)
允许用标志控制 NULL 的 COPY 输入/输出 (Peter E)
使 postgres 用户默认有口令 (Peter E)
添加 CREATE/ALTER/DROP GROUP (Peter E)
所有管理脚本现在支持 --long 选项 (Peter E、Karel)
Vacuumdb 脚本现在支持 --all 选项 (Peter E)
ecpg 新的可移植 FETCH 语法
添加 ecpg 的 EXEC SQL IFDEF、EXEC SQL IFNDEF、EXEC SQL ELSE、EXEC SQL
 ELIF
       和 EXEC SQL ENDIF 指令
添加 pg_ctl 脚本以控制后端启动 (Tatsuo)
添加 postmaster.opts.default 文件以存储启动标志 (Tatsuo)
允许 --with-mb=SQL_ASCII
将索引键最大数量增加到 16 (Bruce)
将函数参数最大数量增加到 16 (Bruce)
允许配置索引键和参数的最大数量 (Bruce)
允许无特权用户更改自己的口令 (Peter E)
启用口令认证; 新用户必需 (Peter E)
禁止删除拥有数据库的用户 (Peter E)
将 initdb 选项 --with-mb 改为 --enable-multibyte
为 initdb 添加提示输入超级用户口令的选项 (Peter E)
允许复杂类型转换, 如 col::numeric(9,2) 和 col::int2::float8 (Tom)
更新 initdb、initlocation、pg_dump、ipcclean 的用户界面 (Peter E)
新的 pg_char_to_encoding() 和 pg_encoding_to_char() 函数 (Tatsuo)
libpq 非阻塞模式 (Alfred Perlstein)
改进未指定长度的类型转换中的类型转换
新的 plperl 内部编程语言 (Mark Hollomon)
允许 COPY IN 读取不以换行结尾的文件 (Tom)
指示长标识符何时被截断 (Tom)
允许聚合使用类型等价 (Peter E)
添加 Oracle 的 to_char()、to_date()、to_datetime()、to_timestamp()、to_
number()
       转换函数 (Karel Zak <zakkr@zf.jcu.cz>)
添加 SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
检查以确保 ORDER BY 与 DISTINCT 操作兼容 (Tom)
向 ODBC 添加 NUMERIC 和 int8 类型
改进 Append、Group、Agg、Unique 的 EXPLAIN 结果 (Tom)

添加 ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
允许在 PL/pgSQL 中使用 SELECT .. FOR UPDATE (Hiroshi)
即使到达 EOF 也启用反向顺序扫描 (Hiroshi)
添加布尔值的 btree 索引, >= 和 <= (Don Baccus)
COPY FROM 失败时打印当前行号 (Massimo)
识别 POSIX 时区, 例如 "PST+8" 和 "GMT-8" (Thomas)
添加 DEC 作为 DECIMAL 的同义词 (Thomas)
添加 SESSION_USER 作为 SQL92 关键字, 同 CURRENT_USER (Thomas)
实现 SQL92 列别名 (又称相关名) (Thomas)
实现 SQL92 连接语法 (Thomas)
使保留字 INTERVAL 可用作列标识符 (Thomas)
实现 REINDEX 命令 (Hiroshi)
接受聚合函数 SUM(ALL col) 中的 ALL (Tom)
防止 GROUP BY 使用列别名 (Tom)
新的 psql \encoding 选项 (Tatsuo)
允许 PQrequestCancel() 在等待锁状态下终止 (Hiroshi)
允许在所有情况下对负数取负
添加 ecpg 描述符 (Christof、Michael)
允许 CREATE VIEW v AS SELECT f1::char(8) FROM tbl
允许带长度的转换, 如 foo::char(8)
新的 libpq 函数 PQsetClientEncoding()、PQclientEncoding() (Tatsuo)
添加对 SJIS 用户定义字符的支持 (Tatsuo)
支持更大的视图/规则
使 libpq 的 PQconndefaults() 线程安全 (Tom)
禁用 // 作为注释以符合 ANSI, 应使用 -- (Tom)
允许视图上的列别名 CREATE VIEW name (collist)
修复带子查询的视图的问题 (Tom)
允许 UPDATE table SET fld = (SELECT ...) (Tom)
SET 命令选项不再需要引号
将 pgaccess 更新到 0.98.6
新的 SET SEED 命令
新的 pg_options.sample 文件
新的 SET FSYNC 命令 (Massimo)
允许创建表时使用 pg_descriptions
允许创建类型、列和函数时使用 pg_descriptions
允许 psql \copy 使用分隔符 (Peter E)
允许 psql 将空值打印为与 "" 不同的 [null] (Peter E)

类型

大量数组修复 (Tom)
允许裸列名作为数组下标使用 (Tom)
改进 int 和 float 常量的类型转换 (Tom)
int8 输入、范围检查和类型转换的清理 (Tom)
修复 SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') 现在是 255.255.255.255 而不是 0.0.0.0 (Oleg
Sharoiko)
在 NUMERIC 上添加 btree 索引 (Jan)
Perl 修复包含 NUL 字符的大对象 (Douglas Thomson)
ODBC 大对象修复 (free)
修复 cidr 数据类型的索引
修复以太网 MAC 地址 (macaddr 类型) 比较的问题
修复计算中发生溢出时日期/时间类型的问题 (Tom)
允许 int8 数组 (Peter E)

修复 NUMERIC 类型的舍入/溢出，如 NUMERIC(4,4) (Tom)
允许 NUMERIC 数组
修复 NUMERIC ceil() 和 floor() 函数中的缺陷 (Tom)
使 char_length()/octet_length 包含尾部空格 (Tom)
使 abstime/reltime 使用 int4 而不是 time_t (Peter E)
新的 lztext 数据类型用于压缩文本字段
修订处理 int 和 float 常量转换的代码 (Tom)
开始编写实现 BIT 和 BIT VARYING 类型的新代码 (Adriaan Joubert)
NUMERIC 现在接受科学记数法 (Tom)
NUMERIC 到 int4 进行舍入 (Tom)
正确地将 float4/8 转换为 NUMERIC (Tom)
允许与 NUMERIC 的类型转换 (Thomas)
使 ISO 日期样式 (2000-02-16 09:33) 成为默认 (Thomas)
添加 NATIONAL CHAR [VARYING] (Thomas)
允许 NUMERIC 的 round 和 trunc 接受负标度 (Tom)
新的 TIME WITH TIME ZONE 类型 (Thomas)
在 time 类型上添加 MAX()/MIN() (Thomas)
为 int8 添加 abs()、mod()、fac() (Thomas)
将 float8 的函数更名为 round()、sqrt()、cbrt()、pow() (Thomas)
为 float8 添加超越数学函数 (如 sin()、acos()) (Thomas)
为 NUMERIC 类型添加 exp() 和 ln()
将 NUMERIC 的 power() 更名为 pow() (Thomas)
改进的 TRANSLATE() 函数 (Edwin Ramirez、Tom)
允许 X=-Y 操作符 (Tom)
允许 SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
允许 LOCALE 在正则表达式搜索中使用索引 (Tom)
允许创建函数索引时使用默认类型

性能

防止大量 AND 和 OR 导致的指数级空间消耗 (Tom)
为系统列收集属性选择率值 (Tom)
减少聚合的内存使用 (Tom)
修复 LIKE 优化以在多字节编码下使用索引 (Tom)
修复 r-tree 索引优化器选择率 (Thomas)
改进优化器选择率计算和函数 (Tom)
优化存在许多相等键时的 btree 搜索 (Tom)
仅在存在索引时启用快速 LIKE 索引处理 (Tom)
重用索引页上重复项的空闲空间 (Tom)
改进哈希连接处理 (Tom)
防止结果已排序时降序排序 (Hiroshi)
允许索引扫描查询条件的交换 (Tom)
在需要 ORDER BY/GROUP BY 的情况下优先使用索引扫描 (Tom)
为提升性能，以固定大小的块分配大内存请求 (Tom)
通过减少内存分配请求改进 vacuum 的性能 (Tom)
实现常量表达式化简 (Bernard Frankpitt、Tom)
使用次要列确定索引扫描的起点 (Hiroshi)
防止内部排序时四倍占用磁盘空间 (Tom)
通过调用更少的函数加快排序 (Tom)
创建系统索引以匹配所有系统缓存 (Bruce、Hiroshi)
使系统缓存使用系统索引 (Bruce)
使所有系统索引唯一 (Bruce)
改进 pg_statistics 管理以提升 VACUUM 速度 (Tom)

降低后端缓存刷新频率 (Tom、Hiroshi)
COPY 现在重用先前的内存分配，改进性能 (Tom)
改进优化代价估算 (Tom)
改进优化器对范围查询 x > lowbound AND x < highbound 的估计 (Tom)
在适当处使用 DNF 而不是 CNF (Tom、Taral)
对 OR-of-AND WHERE 子句的进一步清理 (Tom)
在 OR 子句中使用索引 (x = 1 AND y = 2) OR (x = 2 AND y = 4) (Tom)
对随机索引页访问的更智能优化器计算 (Tom)
新的 SET 变量以控制优化器代价 (Tom)
基于 LIMIT、OFFSET 和 EXISTS 条件优化查询 (Tom)
减少优化器对连接路径的内部管理以加速 (Tom)
子查询的重大加速 (Tom)
fsync 未禁用时更少的 fsync 写 (Tom)
改进的 LIKE 优化器估计 (Tom)
防止仅 SELECT 查询中的 fsync (Vadim)
使索引创建使用 psort 代码，因为它现在更快 (Tom)
允许创建大于 1 Gig 的排序临时表

源代码树变更

修复 linux PPC 编译
新的通用表达式树遍历子例程 (Tom)
将 form() 改为 varargform() 以防止可移植性问题
改进 Alpha 上大整数的范围检查
清理 /include 目录中的 #include (Bruce)
添加检查 include 的脚本 (Bruce)
从 *.c 文件中移除不需要的 #include (Bruce)
酌情将 #include 改为使用 <> 和 "" (Bruce)
启用 libpq 的 Windows 编译
来自 Uncle George 的 Alpha 自旋锁修复 <gatgul@voicenet.com>
优化器数据结构的全面翻修 (Tom)
cygipc 库的修复 (Yutaka Tanida)
允许 pgsql 在较新的 Cygwin 快照上工作 (Dan)
新的目录版本号 (Tom)
添加 Linux ARM
将 heap_replace 更名为 heap_update
为 QNX 更新 (Dr. Andreas Kardos)
新的平台特定回归处理 (Tom)
将 oid8 更名为 oidvector, int28 更名为 int2vector (Bruce)
将所有 yacc 和 lex 文件纳入发行版 (Peter E.)
移除不再需要的 lextest (Peter E)
修复 Windows 上的 libpq 和 psql (Magnus)
内部将 datetime 和 timespan 更改为 timestamp 和 interval (Thomas)
修复 BSD/OS 上的 plpgsql
向回归测试添加 SQL_ASCII 测试用例 (Tatsuo)
configure --with-mb 现已废弃 (Tatsuo)
NT 修复
NetBSD 修复 (Johnny C. Lam <lamj@stat.cmu.edu>)
Alpha 编译的修复
新的多字节编码

35.5. 版本 6.5.3

发布于 1999-10-13。这基本上是 6.5.2 的清理版本。我们添加了 6.5.2 中缺失的新版 pgaccess，并安装了一个 NT 特定的修复。

35.5.1. 迁移到版本 6.5.3

对于运行 6.5.* 的用户，不需要进行转储/恢复。

35.5.2. 变更

更新版本的 pgaccess 0.98
NT 特定补丁
修复继承表上规则的转储

35.6. 版本 6.5.2

发布于 1999-09-15。这基本上是 6.5.1 的清理版本。我们修复了 6.5.1 用户报告的多种问题。

35.6.1. 迁移到版本 6.5.2

对于运行 6.5.* 的用户，不需要进行转储/恢复。

35.6.2. 变更

子查询+CASE 修复 (Tom)
为 solaris_i386 和 solaris_sparc 移植添加 SHLIB_LINK 设置 (Daren Sefcik)
WHERE 连接子句中 CASE 的修复 (Tom)
修复 BTScan 中止 (Tom)
修复对冗余 UNIQUE 和 PRIMARY KEY 索引的检查 (Thomas)
改进为可检查多列约束 (Thomas)
修复启用 MB 时 Win32 的构建问题 (Hiroki Kataoka)
允许 BSD yacc 和 bison 编译 pl 代码 (Bruce)
修复 SET NAMES 的功能
int8 修复 (Thomas)
修复 vacuum 的内存消耗 (Hiroshi、Tatsuo)
降低 vacuum 的总内存消耗 (Tom)
修复 timestamp(datetime)
规则反编译缺陷修复 (Tom)
修复 mkMakefile.tcldefs.sh.in 和 mkMakefile.tkdefs.sh.in 中的引号问题 (Tom)
这是为了重用 vacuum 释放的索引页空间 (Vadim)
为 pg_dump 记录 -x (Bruce)
修复规则反编译器中的一元操作符 (Tom)
在 mdtruncate() 期间注释掉多余段的 FileUnlink (Tom)
来自 Yu Cao 的 IRIX 链接修复 >yucao@falcon.kla-tencor.com<
修复 LIKE 中的逻辑错误：不应返回 LIKE_ABORT
（在到达模式末尾而文本未结束时）(Tom)
修复事务中止期间堆内存分配的不当清理 (Tom)
更新 pgaccess 0.98 版本

35.7. 版本 6.5.1

1999-07-15

这基本上是 6.5 的清理版本。我们修复了 6.5 用户报告的多种问题。

35.7.1. 迁移到版本 6.5.1

对于运行 6.5 的用户，不需要进行转储/恢复。

35.7.2. 变更

添加 NT README 文件
linux_ppc、IRIX、linux_alpha、OpenBSD、alpha 的可移植性修复
移除 QUERY_LIMIT，使用 SELECT...LIMIT
修复继承上的 EXPLAIN (Tom)
补丁允许对多段表进行清理 (Hiroshi)
R-Tree 优化器选择率修复 (Tom)
ACL 文件描述符泄漏修复 (Atsushi Ogawa)
新表达式子树代码 (Tom)
只读事务避免磁盘写 (Vadim)
修复最后事务中止时临时表的移除 (Bruce)
防止创建过大的元组 (Bruce)
plpgsql 修复
允许 32k - 64k 的端口号 (Bruce)
添加 ^ 优先级 (Bruce)
把名为 pg_temp 的排序文件改名为 pg_sorttemp (Bruce)
修复时间值中的微秒 (Tom)
教程源码清理
新 linux_m68k 移植
修复某些情形下 NULL 的排序 (Tom)
修复共享库依赖 (Tom)
修复影响子查询中 GROUP BY 的故障 (Tom)
修复一些编译器警告 (Tomoaki Nishiyama)
新增 Win1250（捷克语）支持 (Pavel Behal)

35.8. 版本 6.5

发布于 1999-06-09。此版本标志着开发团队对从 Berkeley 继承的源代码的掌握迈出了一大步。
你会看到我们现在能轻松添加重大特性，
这要归功于我们全球开发团队不断扩大的规模和日益丰富的经验。

以下是较显著变更的简要总结：

多版本并发控制（MVCC）

这移除了我们旧的表级锁定，代之以优于大多数商业数据库系统的锁定系统。
在传统系统中，每个被修改的行都被锁定直到提交，阻止其他用户读取。MVCC 利用 Postgre SQL 天生的多版本特性，允许读者在写入者活动期间继续读取一致的数据。写入者继续使用

紧凑的 pg_log 事务系统。这一切都不需要像传统数据库系统那样为每行分配锁。因此，基本上我们不再受简单表级锁定的限制；我们有了比行级锁定更好的东西。

来自 pg_dump 的热备份

pg_dump 利用新的 MVCC 特性，在数据库保持在线并可用于查询的同时给出一致的数据库转储/备份。

numeric 数据类型

我们现在有了真正的数字数据类型，精度由用户指定。

临时表

临时表保证在数据库会话内名称唯一，并在会话退出时销毁。

新 SQL 特性

我们现在支持 CASE、INTERSECT 和 EXCEPT 语句。新增了 LIMIT/OFFSET、SET TRANSACTION ISOLATION LEVEL、SELECT ... FOR UPDATE 以及改进的 LOCK TABLE 命令。

提速

得益于团队内的多种才能，我们继续加快 PostgreSQL 的速度。我们加速了内存分配、优化、表连接和行传输例程。

移植

我们继续扩大移植列表，这次包括 WinNT/ix86 和 NetBSD/arm32。

接口

大多数接口有了新版本，现有功能得到改进。

文档

文档中到处都有新的和更新的材料。SGI 和 AIX 平台有了新的 FAQ。教程有 Stefan Simkovics 编写的 SQL 入门信息。用户指南有涵盖 postmaster 和更多工具程序的参考页，新附录包含日期/时间行为的细节。管理员指南有 Tom Lane 编写的故障排除新章节。程序员指南有 Stefan 编写的查询处理描述，以及通过匿名 CVS 和 CVSup 获取 Postgres 源代码树的细节。

35.8.1. 迁移到版本 6.5

希望从任何以前的 Postgres 版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。pg_upgrade 不能用于升级到此版本，因为表的磁盘结构相对于以前的版本已更改。

新的多版本并发控制（MVCC）特性在多用户环境中可能给出略为不同的行为。
请阅读并理解以下章节，以确保你的现有应用程序能给出你需要的行为。

35.8.1.1. 多版本并发控制

由于 6.5 中的读者不锁定数据（无论事务隔离级别如何），一个事务读取的数据可能被另一个事务覆写。换句话说，如果 `SELECT` 返回了一行，并不意味着该行在被返回时（即语句或事务开始之后的某个时刻）确实存在，也不意味着该行在当前事务提交或回滚之前受到保护，
不被并发事务删除或更新。

为确保某行确实存在并保护它不被并发更新，必须使用 SELECT FOR UPDATE 或适当的 LOCK TABLE 语句。从以前的 Postgres 版本和其他环境移植应用程序时应考虑到这一点。

如果你在使用 contrib/refint.* 触发器做参照完整性，请记住上述内容。现在需要额外的技术。一种方法是：如果事务要更新/删除主键，使用 LOCK parent_table IN SHARE ROW EXCLUSIVE MODE 命令；如果事务要更新/插入外键，使用 LOCK parent_table IN SHARE MODE 命令。

> **注意**
>
> 注意，如果你以 SERIALIZABLE 模式运行事务，则必须在执行事务中的任何 DML 语句（SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO）之前执行上述 LOCK 命令。

当实现读取脏（未提交）数据的能力（无论隔离级别）和真正的参照完整性后，
这些不便将消失。

35.8.2. 变更

缺陷修复

修复 text<->float8 和 text<->float4 转换函数 (Thomas)
修复创建带大小写混合约束表的问题 (Billy)
更改 exp()/pow() 行为以在下溢/上溢时报错 (Jan)
修复 pg_dump -z 中的缺陷
内存越界清理 (Tatsuo)
修复 lo_import 崩溃 (Tatsuo)
调整数据类型名的处理以抑制双引号 (Thomas)
用类型强制匹配列和 DEFAULT (Thomas)
修复死锁，使其在一秒睡眠后只检查一次 (Bruce)
聚合和 PL/pgSQL 的修复 (Hiroshi)
修复子查询崩溃 (Vadim)
修复 libpq 函数 PQfnumber 和大小写不敏感名称的问题 (Bahman Rafatjoo)
修复大对象中间写入、无额外块和内存消耗的问题 (Tatsuo)
修复 pg_dump -d 或 -D 以及 INSERT 中特殊字符的引号问题
修复 dynahash 的严重问题 (Tom)
修复 INET/CIDR 可移植性问题
修复 ALTER TABLE ADD COLUMN 中选择率错误的问题 (Bruce)
修复执行器使不同列类型的归并连接可用 (Tom)
修复 Alpha OR 选择率缺陷
修复 OR 索引选择率问题 (Bruce)
修复 \d 为 char()/varchar() 显示正确长度的问题 (Ryan)
修复教程代码 (Clark)
改进 destroyuser 检查 (Oliver)
修复 Kerberos 的问题 (Rodney McDuff)
修复存在脏缓冲区时删除数据库的问题 (Bruce)
修复使 sequence nextval() 可区分大小写的问题 (Bruce)
修复 !!= 操作符
销毁数据库文件前丢弃缓冲区 (Bruce)
修复执行器两次求值函数的情形 (Tatsuo)
允许 sequence nextval 动作区分大小写 (Bruce)
修复优化器对负数索引不工作的问题 (Bruce)
修复执行器中 fjIsNull 的内存泄漏

修复聚合内存泄漏 (Erik Riedel)
允许为包含连字符的用户名授权
清理 inet 类型中的 NULL
清理系统表缺陷 (Tom)
修复 PAGER 和 \? 命令的问题 (Masaaki Sakaida)
把默认多段文件大小限制降为 1GB (Peter)
修复 CREATE OPERATOR 的转储 (Tom)
修复游标反向扫描 (Hiroshi Inoue)
修复使用 \i 时的 COPY FROM STDIN (Tom)
修复子查询在表达式内比较的问题 (Jan)
修复返回行时的错误报告处理 (Tom)
修复数组类型引用的问题 (Tom、Jan)
防止 UPDATE SET oid (Jan)
修复 pg_dump 使 -t 选项可处理大小写敏感的表名
特殊情形 GROUP BY 的修复 (Tom、Jan)
修复失败查询的内存泄漏 (Tom)
DEFAULT 现在支持大小写混合的标识符 (Tom)
修复表和索引 DROP/RENAME 的多段使用 (Ole Gjerde)
禁用 pg_dump 同时使用 -o 和 -d 选项 (Bruce)
允许 pg_dump 正确转储组权限 (Bruce)
修复 INSERT INTO table SELECT * FROM table2 中的 GROUP BY (Jan)
修复视图中的计算 (Jan)
修复数组索引上的聚合 (Tom)
修复 DEFAULT 处理值中需要过多引号的单引号的问题
修复非超级用户导入/导出大对象的安全问题 (Tom)
创建表的事务回滚现在正确清理 (Tom)
修复以允许长表名和列名生成正确的序列名 (Tom)

增强

添加 "vacuumdb" 工具
通过更好的内存分配加速 libpq (Tom)
EXPLAIN 显示所有使用的索引 (Tom)
实现 CASE、COALESCE、NULLIF 表达式 (Thomas)
新 pg_dump 表输出格式 (Constantin)
添加字符串 min()/max() 函数 (Thomas)
把新类型强制转换技术扩展到聚合 (Thomas)
新 moddatetime contrib (Terry)
更新到 pgaccess 0.96 (Constantin)
为单字节 "char" 类型添加例程 (Thomas)
改进的 substr() 函数 (Thomas)
改进的多字节处理 (Tatsuo)
多版本并发控制/MVCC (Vadim)
新 Serialized 模式 (Vadim)
修复超过 2GB 表的问题 (Peter)
新 SET TRANSACTION ISOLATION LEVEL (Vadim)
新 LOCK TABLE IN ... MODE (Vadim)
更新 ODBC 驱动 (Byron)
新 NUMERIC 数据类型 (Jan)
新 SELECT FOR UPDATE (Vadim)
处理输入值的 "NaN" 和 "Infinity" (Jan)
改进的日期/年份处理 (Thomas)
改进的后端连接处理 (Magnus)
日志文件的新选项 ELOG_TIMESTAMPS 和 USE_SYSLOG (Massimo)

新 TCL_ARRAYS 选项 (Massimo)
新 INTERSECT 和 EXCEPT (Stefan)
用于主键跟踪的新 pg_index.indisprimary (D'Arcy)
允许在创建前删除表的新 pg_dump 选项 (Brook)
行输出例程加速 (Tom)
新 READ COMMITTED 隔离级别 (Vadim)
新 TEMP 表/索引 (Bruce)
结果已排序时防止排序 (Jan)
新内存分配优化 (Jan)
允许 psql 执行 \p\g (Bruce)
允许多个规则动作 (Jan)
添加 LIMIT/OFFSET 功能 (Jan)
改进连接大量表时的优化器 (Bruce)
来自 S. Simkovics 硕士论文的新 SQL 介绍 (Stefan、Thomas)
来自 S. Simkovics 硕士论文的后端处理新介绍 (Stefan)
改进的 int8 支持 (Ryan Bradetich、Thomas、Tom)
int8 与 text/varchar 类型之间转换的新例程 (Thomas)
新浓密计划，其中连接元表 (Bruce)
默认启用右手查询 (Bruce)
允许在 configure 时设置可靠的最大后端数
(--with-maxbackends 和 postmaster 开关 (-N backends)) (Tom)
因优化器提速，GEQO 默认现为 10 表 (Tom)
为 MS-SQL 可移植性允许 NULL=Var (Michael、Bruce)
修改 contrib check_primary_key() 使其可为 "automatic" 或
"dependent" (Anand)
允许 psql 对视图的 \d 显示查询 (Ryan)
LIKE 提速 (Bruce)
Ecpg 修复/特性，参见 src/interfaces/ecpg/ChangeLog 文件 (Michael)
JDBC 修复/特性，参见 src/interfaces/jdbc/CHANGELOG (Peter)
使 % 操作符具有与 / 相同的优先级 (Bruce)
添加新 postgres -O 选项以允许系统表结构更改 (Bruce)
更新 contrib/pginterface/findoidjoins 脚本 (Tom)
带索引已删行清理 (vacuum) 的大幅提速 (Vadim)
允许非 SQL 函数根据参数运行不同版本 (Tom)
添加 -E 选项显示 \dt 等发送的实际查询 (Masaaki Sakaida)
在 psql 启动横幅中添加版本号 (Masaaki Sakaida)
新 contrib/vacuumlo 移除未被引用的大对象 (Peter)
新表大小初始化使未清理的表性能更好 (Tom)
改进连接被拒绝时的错误消息 (Tom)
支持 char() 和 varchar() 字段数组 (Massimo)
哈希代码的全面翻修以提高可靠性和性能 (Tom)
更新到 PyGreSQL 2.4 (D'Arcy)
更改调试选项使 -d4 和 -d5 产生不同的节点显示 (Jan)
新 pg_options: pretty_plan、pretty_parse、pretty_rewritten (Jan)
更好的系统表访问优化统计 (Tom)
更好地处理非默认块大小 (Massimo)
改进 GEQO 优化器内存消耗 (Tom)
UNION 现在支持对不在目标列表中的列 ORDER BY (Jan)
libpq++ 的重大改进 (Vince Vielhaber)
pg_dump 现在默认使用 -z (ACL) (Bruce)
后端缓存和内存提速 (Tom)
让 pg_dump 在一个快照事务中完成所有工作 (Vadim)
修复大对象内存泄漏和 pg_dump 问题 (Tom)
INET 类型现在比较时遵循网络掩码

使 VACUUM ANALYZE 只使用读锁 (Vadim)
允许 UNION 上的视图 (Jan)
pg_dump 现在能在活动数据库上生成一致快照 (Vadim)

源代码树变更

改进移植匹配 (Tom)
SunOS 可移植性修复
添加 Windows NT 后端移植并启用动态加载 (Magnus 和 Daniel Horak)
新移植到运行 Linux 的 Cobalt Qube (Mips) (Tatsuo)
移植到 NetBSD/m68k (Mr. Mutsuki Nakajima)
移植到 NetBSD/sun3 (Mr. Mutsuki Nakajima)
移植到 NetBSD/macppc (Toshimi Aoki)
修复 tcl/tk 配置 (Vince)
移除规则查询的 CURRENT 关键字 (Jan)
NT 动态加载现在可用 (Daniel Horak)
添加 ARM32 支持 (Andrew McMurry)
更好地支持 HP-UX 11 和 UnixWare
改进文件处理使其更统一，防止文件描述符泄漏 (Tom)
plpgsql 的新安装命令 (Jan)

35.9. 版本 6.4.2

1999-12-20

6.4.1 版本打包不当。此版本还包含一个额外的缺陷修复。

35.9.1. 迁移到版本 6.4.2

对于运行 6.4.* 的用户，不需要进行转储/恢复。

35.9.2. 变更

修复某些平台上日期时间常量的问题 (Thomas)

35.10. 版本 6.4.1

1999-12-18

这基本上是 6.4 的清理版本。我们修复了 6.4 用户报告的多种问题。

35.10.1. 迁移到版本 6.4.1

对于运行 6.4 的用户，不需要进行转储/恢复。

35.10.2. 变更

为 pg_dump 添加 -N 标志以强制在标识符周围加上双引号。这是
        默认行为 (Thomas)
修复 where 子句中 NOT 导致崩溃的问题 (Bruce)
EXPLAIN VERBOSE 核心转储修复 (Vadim)

修复 Linux 上的共享库问题
修复表存在性测试，以允许表名中包含
        大小写混合和空白（Thomas）
修复几个 pg_dump 缺陷
Configure 更好地匹配 template/.similar 条目（Tom）
把内建函数名从 SPI_* 改为 spi_*
OR WHERE 子句修复（Vadim）
修复大小写混合表名的多处问题（Billy）
contrib/linux/postgres.init.csh/sh 修复（Thomas）
libpq 内存越界修复
SunOS 修复（Tom）
更改 exp() 行为在下溢时报错（Thomas）
修复 pg_dump 的内存泄漏、继承约束、布局更改问题
把 pgaccess 更新到 0.93
修复 64 位平台上的原型
多字节修复（Tatsuo）
新 ecpg 手册页
内存越界修复（Tatsuo）
lo_import() 崩溃修复（Bruce）
改进对 install 程序的查找（Tom）
时区修复（Tom）
HPUX 修复（Tom）
用隐式类型强制转换匹配 DEFAULT 值（Thomas）
添加用于辅助单字节（内部）字符类型的例程（Thomas）
win32 下 libpq 编译修复（Magnus）
升级到 PyGreSQL 2.2（D'Arcy）

35.11. 版本 6.4

1998-10-30

此版本有许多新特性和改进。感谢我们的开发者和维护者，
自上一个版本以来系统的几乎每个方面都得到了关注。以下是简要而不完整的总结：

- 得益于 Jan Wieck 在重写规则系统中的大量新代码，视图和规则现在可以工作了。他还为程序员指南写了相关章节。
- Jan 还贡献了第二个过程语言 PL/pgSQL，与他在上个版本贡献的最初的 PL/pgTCL 过程语言相伴。
- 我们有 Tatsuo Ishii 提供的可选多字节字符集支持，以补充现有的区域设置支持。
- 客户端/服务器通信得到了清理，感谢 Tom Lane，对异步消息和中断的支持更好了。
- 解析器现在执行自动类型强制转换，以把参数匹配到可用的操作符和函数，
  并把列和表达式与目标列匹配。这使用了支持 Postgres 类型可扩展特性的通用机制。用户指南有涵盖此主题的新章节。
- 新增了三种数据类型。其中两种，inet 和 cidr，支持各种形式的 IP 网络、子网和机器寻址。某些平台上现在有 8 字节整数类型可用。详情见用户指南中的数据类型章节。第四种类型 serial 现在被解析器支持，作为 int4 类型、序列和唯一索引的混合体。
- 新增了更多 SQL92 兼容的语法特性，包括 INSERT DEFAULT VALUES
- 自动配置和安装系统得到了一些关注，应比以往在更多平台上更健壮。

35.11.1. 迁移到版本 6.4

希望从任何以前的 Postgres 版本迁移数据的用户，需要使用 pg_dump 或 pg_dumpall 进行转储/恢复。

35.11.2. 变更

缺陷修复

修复 PQsetdb/PQfinish 中的小内存泄漏 (Bryan)
移除 char2-16 数据类型，使用 char/varchar (Darren)
Pqfn 现在处理 NOTICE 消息 (Anders)
降低多后端时自旋锁的忙等开销 (dg)
卡死自旋锁检测 (dg)
修复 "ISO 风格" timespan 的解码和编码 (Thomas)
修复事务回滚后删除表的问题 (Vadim)
更改错误消息并移除无效的更新消息 (Vadim)
修复 COPY 数组检查的问题
修复 SELECT 1 UNION SELECT NULL
修复大对象调用中的缓冲区泄漏 (Pascal)
把 owner 从 oid 类型改为 int4 (Bruce)
修复 Oracle 兼容函数 btrim()、ltrim() 和 rtrim() 中的缺陷
修复共享失效缓存溢出 (Massimo)
防止失败的 COPY 中的文件描述符泄漏 (Bruce)
修复 libpgtcl 的 pg_select 中的内存泄漏 (Constantin)
修复超过 8 字符的用户名/口令问题 (Tom)
修复后端处理异步 NOTIFY 的问题 (Tom)
修复许多错误的系统表条目 (Tom)

增强

升级 ecpg 和 ecpglib，参见 src/interfaces/ecpc/ChangeLog (Michael)
在 EXPLAIN 中显示使用的索引 (Zeugswetter)
EXPLAIN 调用规则系统并显示被重写查询的计划 (Jan)
通过 configure 使许多数据类型和函数感知多字节 (Tatsuo)
新 configure --with-mb 选项 (Tatsuo)
新 initdb --pgencoding 选项 (Tatsuo)
新 createdb -E 多字节选项 (Tatsuo)
Select version(); 现在返回 PostgreSQL 版本 (Jeroen)
libpq 现在允许异步客户端 (Tom)
允许从客户端取消后端查询 (Tom)
psql 现在可用 Control-C 取消查询 (Tom)
libpq 用户不必发送哑查询即可获取 NOTIFY 消息 (Tom)
NOTIFY 现在发送发送者的 PID，可以判断是否是自己的 (Tom)
PGresult 结构现在包含相关的错误消息（如果有） (Tom)
定义 date_part() 的 "tz_hour" 和 "tz_minute" 参数 (Thomas)
添加 varchar 和 bpchar 之间转换的例程 (Thomas)
添加允许把 varchar 和 bpchar 调整到目标列大小的例程 (Thomas)
添加位标志以支持数据检索中的时区小时和分钟 (Thomas)
允许有效浮点数的更多变体（如 ".1"、"1e6"） (Thomas)
修复带前导空格的一元负号解析 (Thomas)
按 SQL92 规范实现 TIMEZONE_HOUR、TIMEZONE_MINUTE (Thomas)
检查并正确忽略 FOREIGN KEY 列约束 (Thomas)

按 SQL92 规范定义 USER 为 CURRENT_USER 的同义词 (Thomas)
启用 HAVING 子句但其他地方尚无修复。
使 "char" 类型成为 "char(1)" 的同义词 (实际实现为 bpchar) (Thomas)
为 DEFAULT 子句处理保存指定的字符串类型 (Thomas)
强制涉及不同数据类型的操作 (Thomas)
允许对不同类型列的某些索引使用 (Thomas)
添加自动类型转换能力 (Thomas)
清理大对象，使文件在打开时被截断 (Peter)
Readline 清理 (Tom)
允许 psql \f \ 用空格作为分隔符 (Bruce)
把 pg_attribute.atttypmod 传递给前端以获取列字段长度 (Tom、Bruce)
/contrib 中的 Msql 兼容库 (Aldrin)
移除 ORDER/GROUP BY 子句标识符必须
包含在目标列表中的要求 (David)
转换列以匹配 UNION 子句中的列 (Thomas)
移除 fork()/exec() 只做 fork() (Bruce)
Jdbc 清理 (Peter)
在 ps 命令行上显示后端状态 (只在某些平台上有效) (Bruce)
Pg_hba.conf 的 database 字段现在有 sameuser 选项
使 lo_unlink 接受 oid 参数而不是 int4
为无法处理我们宏的编译器新增 DISABLE_COMPLEX_MACRO (Bruce)
Libpgtcl 现在把 NOTIFY 作为 Tcl 事件处理，不必发送哑查询 (Tom)
libpgtcl 清理 (Tom)
向 libpgtcl 的 pg_result 命令添加 -error 选项 (Tom)
新区域设置补丁，参见 docs/README/locale (Oleg)
修复 pg_dump 使 CONSTRAINT 和 CHECK 语法正确 (ccb)
新 contrib/lo 代码用于移除孤立大对象 (Peter)
多字节特性的新 psql 命令 "SET CLIENT_ENCODING TO 'encoding'",
参见 /doc/README.mb (Tatsuo)
contrib/noupdate 代码用于撤销列上的更新权限
libpq 现在可以在 Windows 上编译 (Magnus)
在 libpq 中添加 PQsetdbLogin()
新 8 字节整数类型，由 configure 检查操作系统支持 (Thomas)
更好地支持带引号的表/列名 (Thomas)
在 pg_dump 中用双引号包围表和列名 (Thomas)
PQreset() 现在可用于口令 (Tom)
处理 GROUP BY 目标列表列号越界的情形 (David)
允许子查询中的 UNION
向 \d? 命令添加屏幕自动调整大小 (Bruce)
用 UNION 在一个查询中显示所有 \d? 结果 (Bruce)
添加 \d? 字段搜索特性 (Bruce)
Pg_dump 发出更少的 \connect 请求 (Tom)
使 pg_dump -z 标志更好地工作并在手册页中记录 (Tom)
添加对子查询和 union 完全支持的 HAVING 子句 (Stephan)
contrib/fulltextindex 中的全文索引例程 (Maarten)
事务 id 现在存储在共享内存中 (Vadim)
发出 COPY 命令时的新 PGCLIENTENCODING (Tatsuo)
支持 SQL92 语法 "SET NAMES" (Tatsuo)
支持 LATIN2-5 (Tatsuo)
添加 UNICODE 回归测试用例 (Tatsuo)
锁管理器清理，LLL 的新锁定模式 (Vadim)
允许 OR 子句使用索引 (Bruce)
允许 "SELECT NULL ORDER BY 1;"
Explain VERBOSE 打印计划，现在还把计划美化打印到

postmaster 日志文件 (Bruce)
向 \d 命令添加索引显示 (Bruce)
允许对函数 GROUP BY (David)
大对象的新 pg_class.relkind (Bruce)
把 libpq NOTICE 消息发送到其他位置的新方法 (Tom)
psql 的新 \w 写入命令 (Bruce)
新 /contrib/findoidjoins 扫描 oid 列以发现连接关系 (Bruce)
检查含常量的限制子句的有效索引时
允许考虑二进制兼容的索引 (Thomas)
/contrib/isbn_issn 中的新 ISBN/ISSN 代码
允许 NOT LIKE、IN、NOT IN、BETWEEN 和 NOT BETWEEN 约束 (Thomas)
新重写系统修复了规则和视图的许多问题 (Jan)

* 关系上的规则可用
* insert/update/delete 上的事件限定可用
* 新的 OLD 变量引用 CURRENT，CURRENT 未来将被移除
* 更新规则可在规则限定/动作中引用 NEW 和 OLD
* 视图上的 insert/update/delete 规则可用
* 现在支持多个规则动作，用括号包围
* 普通用户可以在有 RULE 权限的表上创建视图/规则
* 规则和视图继承创建者的权限
* 没有列级别的规则
* 没有 UPDATE NEW/OLD 规则
* 新 pg_tables、pg_indexes、pg_rules 和 pg_views 系统视图
* SELECT 规则只有单个动作
* 彻底的重写改造，也许在 6.5
* 处理子查询
* 处理视图上的聚合
* 处理 insert into select from view 可用

系统索引现在是多键的 (Bruce)
移除 Oidint2、oidint4 和 oidname 类型 (Bruce)
更多系统表查找使用系统缓存 (Bruce)
backend/pl 中的新后端编程语言 PL/pgSQL (Jan)
新 SERIAL 数据类型，自动创建序列/索引 (Thomas)
无需重新编译即可启用断言检查 (Massimo)
用户锁增强 (Massimo)
设置序列值的新 setval() 命令 (Massimo)
启动时若无 postmaster 运行则自动移除 unix 套接字文件 (Massimo)
条件跟踪包 (Massimo)
新 UNLISTEN 命令 (Massimo)
psql 和 libpq 现在可用 win32.mak 在 Windows 下编译 (Magnus)
Lo_read 不再存储尾随 NULL (Bruce)
标识符现在内部截断为 31 字符 (Bruce)
Createuser 选项现在可在命令行上使用
添加 64 位整数支持代码，configure 已测试，int8 类型 (Thomas)
防止失败 COPY 的文件描述符泄漏 (Bruce)
新 pg_upgrade 命令 (Bruce)
更新的 /contrib 目录 (Massimo)
新 CREATE TABLE DEFAULT VALUES 语句可用 (Thomas)
新 INSERT INTO TABLE DEFAULT VALUES 语句可用 (Thomas)
新 DECLARE 和 FETCH 特性 (Thomas)
libpq 的内部结构现在不导出 (Tom)
允许多达 8 键的索引 (Bruce)
移除 ARCHIVE 关键字，它不再使用 (Thomas)
pg_dump -n 标志以抑制标识符周围的引号

禁用视图的系统列 (Jan)
用于网络地址的新 INET 和 CIDR 类型 (TomH、Paul)
psql 输出中不再有双引号
pg_dump 现在转储视图 (Terry)
新 SET QUERY_LIMIT (Tatsuo、Jan)

源代码树变更

/contrib 清理 (Jun)
内联一些每行都调用的小函数 (Bruce)
Alpha/linux 修复
HP-UX 清理 (Tom)
多字节回归测试 (Soonmyung.)
从 configure 移除 --disabled 选项
定义 PGDOC 默认使用 POSTGRESDIR
使回归可选
移除 pgindent 的额外花括号代码 (Bruce)
添加 bsdi 共享库支持 (Bruce)
新 --without-CXX 支持 configure 选项 (Brook)
新 FAQ_CVS
更新 tools/backend 中的后端流程图 (Bruce)
把 atttypmod 从 int16 改为 int32 (Bruce、Tom)
为没有 Getrusage() 的平台提供修复 (Tom)
把 PQconnectdb、PGUSER、PGPASSWORD 添加到 libpq 手册页
NS32K 平台修复 (Phil Nelson、John Buller)
SCO 7/UnixWare 2.x 修复 (Billy 等)
Sparc/Solaris 2.5 修复 (Ryan)
Pgbuiltin.3 已过时，移到 doc 文件 (Thomas)
更多的文档 (Thomas)
Nextstep 支持 (Jacek)
Aix 支持 (David)
pginterface 手册页 (Bruce)
所有共享库都有版本号
把所有操作系统特定的共享库定义合并到一个文件
更智能的 TCL/TK 配置检查 (Billy)
更智能的 perl 配置 (Brook)
未找到 install 脚本时 configure 使用随附的 install-sh (Tom)
用于共享库配置的新 Makefile.shlib (Tom)

35.12. 版本 6.3.2

1998-04-07

这是 6.3.x 的缺陷修复版本。有关 6.3 版本系列新特性更完整的摘要，请参阅 6.3 版本的发行说明。

摘要：

- 修复某些平台（包括 Linux）的自动配置支持，该问题是 6.3.1 版本无意中引入的。
- 正确处理 BETWEEN 和 LIKE 子句左侧的函数调用。

对于运行 6.3 或 6.3.1 的用户，不需要进行转储/恢复。只需做一次 'make distclean'、'make' 和 'make install'。最后一步应在 postmaster 未运行时执行。你应重新链接任何使用 Postgres 库的自定义应用程序。

对于从 6.3 之前安装的升级，请参阅 6.3 版本的安装和迁移说明。

35.12.1. 变更

对 tcl/tk 的配置检测改进 (Brook Milligan、Alvin)
手册页改进 (Bruce)
BETWEEN 和 LIKE 修复 (Thomas)
修复 pg_dump 所用 psql \connect 的问题 (Oliver Elphick)
新 odbc 驱动
pgaccess 0.86 版
移除 qsort，现使用 libc 版本并清理 (Jeroen)
修复检测到的缓冲区越界 (Maurice Gittens)
修复 libpgtcl 中的缓冲区越界 (Randy Kunkee)
修复带 DISTINCT 或 ORDER BY 的 UNION (Bruce)
gettimeofday configure 检查 (Doug Winterburn)
修复 "indexes not used" 缺陷 (Vadim)
文档补充 (Thomas)
修复后端内存泄漏 (Bruce)
libreadline 清理 (Erwan MAS)
移除 DISTDIR (Bruce)
Makefile 依赖清理 (Jeroen van Vianen)
ASSERT 修复 (Bruce)

35.13. 版本 6.3.1

发布于 1998-03-23。摘要：

- 对多字节字符集的额外支持。
- 修复混合端序客户端和服务器的字节顺序。
- 对允许的 SQL 语法做小的更新。
- 改进安装时配置的自动检测。

对于运行 6.3 的用户，不需要进行转储/恢复。只需做一次 'make distclean'、'make' 和 'make install'。最后一步应在 postmaster 未运行时执行。你应重新链接任何使用 Postgres 库的自定义应用程序。

对于从 6.3 之前安装的升级，请参阅 6.3 版本的安装和迁移说明。

35.13.1. 变更

ecpg 清理/修复，现为 1.1 版 (Michael Meskes)
pg_user 清理 (Bruce)
pg_dump 和 tclsh 的大对象修复 (alvin)
修复多个相邻下划线的 LIKE
修复重定义内建函数的问题 (Thomas)
ultrix4 清理
升级到 pg_access 0.83
更新 CLUSTER 手册页
多字节字符集支持，参见 doc/README.mb (Tatsuo)
configure --with-pgport 修复
pg_ident 修复

修复后端通信的大端问题 (Kataoka)
修复 SUBSTR() 和 substring() (Jan)
若干 jdbc 修复 (Peter)
libpgtcl 改进，见 libptcl/README (Randy Kunkee)
修复 "Datasize = 0" 错误 (Vadim)
防止 \do 折行 (Bruce)
移除重复的俄文字符集条目
Sunos4 清理
允许 LOCK 和 SELECT INTO 中使用可选的 TABLE 关键字 (Thomas)
CREATE SEQUENCE 选项允许负整数 (Thomas)
添加 "PASSWORD" 作为允许的列标识符 (Thomas)
添加对 UNION 目标字段的检查 (Bruce)
修复 Alpha 移植 (Dwayne Bailey)
修复包含引号的 text 数组 (Doug Gibson)
Solaris 编译修复 (Albert Chin-A-Young)
更好地识别 tcl 与 tk 库和头文件 (Bruce)

35.14. 版本 6.3

发布于 1998-03-01。此版本有许多新特性和改进。以下是简要而不完整的总结：

- 许多新的 SQL 特性，包括完整的 SQL92 子查询能力（只缺目标列表子查询，其余一切都有了）。
- 支持用客户端环境变量指定时区和日期样式。
- 客户端/服务器连接的套接字接口。现在这是默认，所以你可能需要用 -i 标志启动 postmaster。
- 更好的口令认证机制。默认表权限已经更改。
- 旧式时间旅行已被移除。性能得到改进。

> **注意**
>
> Bruce Momjian 写了以下说明来介绍新版本。

我想提一些 6.3 的普遍问题。这里只是无法用一句话描述的大项。仍需查阅详细变更列表。

首先，我们现在有了子查询。既然有了它们，我想指出：没有子查询的 SQL 是一种非常受限的语言。子查询是一项重大特性，你应当检查自己的代码，
看看哪些地方子查询能为你的查询提供更好的解决方案。我想你会发现，
子查询的用途比你想象的更多。Vadim 用子查询让我们登上了 SQL 的大舞台，而且是功能完备的子查询。子查询唯一不能做的，是用在目标列表中。

其次，6.3 默认使用 Unix 域套接字而不是 TCP/IP。要启用来自其他机器的连接，必须使用新的 postmaster -i 选项，当然还要编辑 pg_hba.conf。此外，由于这个原因，pg_hba.conf 的格式已经更改。

第三，char() 字段现在允许比 varchar() 或 text 更快的访问。具体来说，text 和 varchar() 在访问该类型第一列之后的任何列时有性能损失。char() 过去也有这种访问损失，但现在没有了。这可能提示你重新设计一些表，特别是如果你有定义为 varchar() 或 text 的短字符列。这项和其他变更使 6.3 比以前的版本更快。

我们现在可以独立于任何 Unix 文件定义口令。有新的 SQL USER 命令。更多信息见 pg_hba.conf 手册页。还有一个新表 pg_shadow，用于存储用户信息和用户口令，默认只有 postgres 超级用户可以 SELECT。pg_user 现在是 pg_shadow 的视图，可被 PUBLIC SELECT。你的应用程序应继续使用 pg_user 而无需更改。

用户创建的表现在默认不再对 PUBLIC 授予 SELECT 权限。这样做是因为 ANSI 标准的要求。当然，你可以在表创建之后随意 GRANT 你想要的任何权限。系统表仍然可被 PUBLIC SELECT。

我们还有真正的死锁检测代码。不再有六十秒超时。而且新的锁定代码更好地实现了 FIFO，因此在重度使用期间资源饥饿应该会更少。

以前的版本中有很多关于文档不足的抱怨。Thomas 在此版本的许多新手册上投入了大量精力。请查看 doc/ 目录。

出于性能原因，时间旅行已被移除，但可以用触发器实现（参见 pgsql/contrib/spi/README）。请查看新的 \d 命令以了解类型、操作符等。此外，视图现在有自己的权限，不再基于底层表，因此必须单独设置视图的权限。查看 /pgsql/interfaces 了解与 Postgres 交互的一些新方式。

这是第一个真正需要为现有用户做解释的版本。在许多方面，这是必要的，
因为新版本移除了许多限制，人们以前使用的变通方法不再需要。

35.14.1. 迁移到版本 6.3

希望从任何以前的 Postgres 版本迁移数据的用户，需要使用 pg_dump 或 pg_dumpall 进行转储/恢复。

35.14.2. 变更

```
缺陷修复
---------
修复被 MOVE 实现破坏的二进制游标 (Vadim)
修复 tcl 库崩溃 (Jan)
修复数组处理的问题，来自 Gerhard Hintermayer
修复 acl 错误并移除重复的 pqtrace (Bruce)
修复 psql \e 对空文件的问题 (Bruce)
修复 varchar() 字段上的 textcat (Bruce)
修复 DBT Sendproc (Zeugswetter Andres)
修复 vacuum analyze 语法问题 (Bruce)
修复国际标识符的问题 (Tatsuo)
修复继承表上的聚合 (Bruce)
修复 substr() 的越界数据问题
修复 select 1=1 or 2=2、select 1=1 and 2=2 和 select sum(2+2) (Bruce)
修复 notty 输出以显示状态结果。-q 选项仍会关闭它 (Bruce)
修复 count(*)、带视图和多表的聚合以及 sum(3) (Bruce)
修复 cluster (Bruce)
修复 PQtrace 多次启动/停止的问题 (Bruce)
修复多种锁定问题，例如较新的锁等待者先于较旧的等待者获得锁、
        有写者等待锁时读锁持有者不共享锁、以及等待的写者没有
        优先于等待的读者 (Bruce)
修复从外部文件执行查询时 psql 的崩溃 (James)
修复多列 order by 且第一列含 NULL 值的问题 (Jeroen)
为 float8 和 int4 使用正确的哈希表支持函数 (Thomas)
重新启用 CREATE OPERATOR 语句中的 JOIN= 选项 (Thomas)
```

更改布尔操作符的优先级以符合预期行为 (Thomas)
对过大整数生成 elog(ERROR) (Bruce)
允许在约束子句中使用多参数函数 (Thomas)
检查布尔输入字面量 'true'、'false'、'yes'、'no'、'1'、'0',
　　无法识别时抛出 elog(ERROR) (Thomas)
大型对象的重要修复
修复 GROUP BY 显示重复项的问题 (Vadim)
修复 MergeJoin 中的索引扫描 (Vadim)

增强

带 EXISTS、IN、ALL、ANY 关键字的子查询 (Vadim、Bruce、Thomas)
新用户手册 (Thomas 等)
通过内联一些频繁调用的函数提速
真正的死锁检测，不再依赖超时 (Bruce)
添加 SQL92 "常量" CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、
　　CURRENT_USER (Thomas)
修改约束语法以符合 SQL92 (Thomas)
用索引实现 SQL92 PRIMARY KEY 和 UNIQUE 子句 (Thomas)
识别 FOREIGN KEY 的 SQL92 语法。抛出 elog 通知 (Thomas)
允许 NOT NULL UNIQUE 约束子句（以前各自单独允许） (Thomas)
允许对非常量使用 PostgreSQL 风格的类型转换 ("::") (Thomas)
添加对 SQL3 TRUE 和 FALSE 布尔常量的支持 (Thomas)
支持 IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE 的 SQL92 语法 (Thomas)
允许更短的布尔字面量字符串（如 "t"、"tr"、"tru"） (Thomas)
允许 SQL92 分隔标识符 (Thomas)
实现 SQL92 二进制和十六进制字符串解码 (b'10' 和 x'1F') (Thomas)
支持字面字符串类型强制转换的 SQL92 语法
　　(如 "DATETIME 'now'") (Thomas)
添加 int2、int4 和 OID 类型与 text 之间的转换 (Thomas)
构建索引时使用共享锁 (Vadim)
事务块内的用户查询完成后释放为其分配的内存,
　　此功能在 <= 6.2.1 中被关闭 (Vadim)
新 SQL 语句 CREATE PROCEDURAL LANGUAGE (Jan)
新 Postgres 过程语言 (PL) 后端接口 (Jan)
把 pg_dump -H 选项改名为 -h (Bruce)
添加 Java 对口令和欧洲日期的支持 (Peter)
为 LIKE 和 ~、!~ 操作使用索引 (Bruce)
添加 datetime 和 timespan 的哈希函数 (Thomas)
移除时间旅行 (Vadim、Bruce)
为 \d 和 \z 添加分页并修复 \i (Bruce)
向后端和前端库添加 Unix 域套接字支持 (Goran)
实现 CREATE DATABASE/WITH LOCATION 和 initlocation 工具 (Thomas)
允许更多 SQL92 和/或 Postgres 保留字作为列标识符 (Thomas)
增强对 SQL92 SET TIME ZONE... 的支持 (Thomas)
SET/SHOW/RESET TIME ZONE 使用 TZ 后端环境变量 (Thomas)
实现 SET keyword = DEFAULT 和 SET TIME ZONE DEFAULT (Thomas)
启用使用 TZ 环境变量的 SET TIME ZONE (Thomas)
把 PGDATESTYLE 环境变量添加到前端和后端初始化 (Thomas)
添加 PGTZ、PGCOSTHEAP、PGCOSTINDEX、PGRPLANS、PGGEQO
　　前端库初始化环境变量 (Thomas)
回归测试时区用 "setenv PGTZ PST8PDT" 自动设置 (Thomas)
添加 pg_description 表以存放表、列、操作符、类型和
　　聚合的信息 (Bruce)

把系统表/索引名 16 字符的限制增加到 32 字符 (Bruce)
重命名系统索引 (Bruce)
向 SET DATESTYLE 添加 'GERMAN' 选项 (Thomas)
定义带 "hh:mm:ss" 字段的 "ISO 风格" timespan 输出格式 (Thomas)
允许 delta time 的小数值 (如 '2.5 days') (Thomas)
更仔细地验证 delta time 的数字输入 (Thomas)
实现把年积日作为 date_part() 的可能输入 (Thomas)
定义 timespan_finite() 和 text_timespan() 函数 (Thomas)
移除归档内容 (Bruce)
允许有独立于系统口令文件的 pg_password 认证数据库 (Todd)
转储 ACL、GRANT、REVOKE 权限 (Matt)
定义 text、varchar 和 bpchar 字符串长度函数 (Thomas)
修复 Query 对继承的处理以及代价计算 (Bruce)
实现 CREATE TABLE/AS SELECT (SELECT/INTO 的替代) (Thomas)
允许约束中的 NOT、IS NULL、IS NOT NULL (Thomas)
实现 SELECT 的 UNION (Bruce)
向 INSERT 添加 UNION、GROUP、DISTINCT (Bruce)
varchar() 在磁盘上只存储必要的字节 (Bruce)
修复 BLOB 的问题 (Peter)
JDBC 超大补丁……变更列表见 README_6.3 (Peter)
从 PQconnectdb() 移除未使用的 "option"
新 LOCK 命令和描述死锁的锁手册页 (Bruce)
添加新 psql \da、\dd、\df、\do、\dS 和 \dT 命令 (Bruce)
增强 psql \z 以显示序列 (Bruce)
在 psql \d 表中显示 NOT NULL 和 DEFAULT (Bruce)
新 psql .psqlrc 文件启动 (Andrew)
修改 contrib/linux 中的示例启动脚本以显示 syslog (Thomas)
contrib/ip_and_mac 中 IP 和 MAC 地址的新类型 (TomH)
contrib/unixdate 中与日期/时间类型的 Unix 系统时间转换 (Thomas)
contrib 内容的更新 (Massimo)
向 DBD::Pg 添加 Unix 套接字支持 (Goran)
新 python 接口 (PyGreSQL 2.0) (D'Arcy)
新的前端/后端协议带有版本号和网络字节序 (Phil)
pg_hba.conf 的安全特性得到增强和记录，大量清理 (Phil)
CHAR() 现在比 VARCHAR() 或 TEXT 访问更快
ecpg 嵌入式 SQL 预处理器
降低系统列的开销 (Vadmin)
移除 pg_time 表 (Vadim)
添加 pg_type 属性以标识需要长度的类型 (bpchar、varchar)
COPY 命令失败时报告出错的行
允许对视图设置的权限独立于底层表。
        为安全起见，酌情对视图使用 GRANT/REVOKE (Jan)
表现在没有默认的 GRANT SELECT TO PUBLIC。你必须
        显式授予此类权限。
清理教程示例 (Darren)

源码树变更

添加新的 html 开发工具和 /tools/backend 中的流程图
修复 SCO 编译
Stratus 计算机移植 (Robert Gillies)
为 BSD44_derived & i386_solaris 添加 shlib 支持
使 configure 更加自动化 (Brook)
添加检查回归测试结果的脚本

将解析器函数拆分为更小的文件并分组 (Bruce)
把 heap_create 改名为 heap_create_and_catalog，把 heap_creatr
改名为 heap_create() (Bruce)
Sparc/Linux 的锁定补丁 (TomS)
移除 PORTNAME 并重组移植特定内容 (Marc)
添加优化器 README 文件 (Bruce)
移除优化器中的部分递归并清理那里的代码 (Bruce)
修复 NetBSD 锁定 (Henry)
修复 libptcl 的制作 (Tatsuo)
AIX 补丁 (Darren)
把 IS TRUE、IS FALSE、... 更改为使用 "=" 的表达式而不是
对 istrue() 或 isfalse() 的函数调用以允许优化 (Thomas)
各种 NetBSD/Sparc 相关修复 (TomH)
Alpha linux 锁定 (Travis、Ryan)
把 elog(WARN) 改为 elog(ERROR) (Bruce)
FreeBSD 的 FAQ (Marc)
将 PostODBC 源码树纳入标准发行版 (Marc)
HP/UX 10 与 9 的小补丁 (Stan)
新 pg_attribute.atttypmod 用于 varchar 长度等类型特定信息 (Bruce)
UnixWare 补丁 (Billy)
自旋锁 asm 的新 i386 'lock' (Billy)
移除对多路复用后端的支持
开始 OpenBSD 移植
开始 AUX 移植
开始 Cygnus 移植
在回归测试套件中添加字符串函数 (Thomas)
扩展一些以前截断为 16 字符的函数名 (Thomas)
移除不需要的 malloc() 调用并用 palloc() 替换 (Bruce)

35.15. 版本 6.2.1

1997-10-17

6.2.1 是 6.2 之上的缺陷修复和易用性版本。

摘要：

- 按照 SQL92，允许字符串跨行。
- 包含在表更新时插入用户名的示例触发器函数。

这是 6.2 之上的次要缺陷修复版本。从 6.2 之前的系统升级需要完整的转储/重载。请参阅 6.2 版本的发行说明了解操作说明。

35.15.1. 从版本 6.2 迁移到版本 6.2.1

这是次要缺陷修复版本。从 6.2 版本不需要转储/重载，但从 6.2 之前的任何版本需要。

从 6.2 版本升级时，如果你选择转储/重载，你会发现 avg(money) 现在计算正确。所有其他缺陷修复在更新可执行文件后生效。

避免转储/重载的另一种方法是在 psql 中使用以下 SQL 命令更新现有的系统表：

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

这需要对每个现有数据库执行，包括 template1。

35.15.2. 变更

允许 TIME 和 TYPE 作为列名 (Thomas)
允许更大范围的 true/false 布尔值 (Thomas)
支持 "now" 和 "current" 的输出 (Thomas)
正确处理 INSERT NULL 与 DEFAULT (Vadim)
修复缓冲区管理器中关系引用计数的问题 (Vadim)
像 ANSI 一样允许字符串跨行 (Thomas)
修复带 ORDER BY 的反向游标 (Vadim)
修复 avg(cash) 计算 (Thomas)
修复 ORDER/GROUP BY 中重复指定列的问题 (Vadim)
记录返回受影响行数的新 libpq 函数 PQcmdTuples (Bruce)
用于在 INSERT/UPDATE 时插入用户名的触发器函数 (Brook Milligan)

35.16. 版本 6.2

1997-10-02

希望从以前的 Postgres 版本迁移数据的用户，需要进行转储/恢复。

35.16.1. 从版本 6.1 迁移到版本 6.2

此迁移需要完整转储 6.1 数据库并在 6.2 中恢复。

注意，应使用 6.2 的 pg_dump 和 pg_dumpall 工具来转储 6.1 数据库。

35.16.2. 从版本 1.*x* 迁移到版本 6.2

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

35.16.3. 变更

缺陷修复

修复 pg_dump 对继承、序列、归档表的问题 (Bruce)
修复因移位、无符号和 Solaris 的错误原型
导致的溢出编译错误 (Diab Jerius)
修复几何线段算术中的缺陷（错误的交点计算） (Thomas)
检查几何对象在端点处的相交以避免舍入问题 (Thomas)
捕获无效的删除尝试 (Vadim)
把时间函数名改得更一致 (Michael Reifenberg)
检查除零 (Michael Reifenberg)
修复一个非常老的缺陷：命令本身可以看见
由该命令更改/插入的行（因此会导致对已更新
元组的多次更新等） (Vadim)
修复 SELECT null, 'fail' FROM pg_am (Patrick)
现在允许 SELECT NULL as EMPTY_FIELD (Patrick)
从 contrib/pginterface 中移除不需要的信号代码
修复 OR (where x != 1 or x isnull 不返回 x NULL 的行) (Vadim)
修复 time_cmp 函数 (Vadim)

修复 WHERE 子句中对首参数为非属性的函数的处理 (Vadim)
修复条目顺序与目标列表顺序不同时的 GROUP BY (Vadim)
修复 pg_dump 对无 sfunc1 聚合的处理 (Vadim)

增强

默认遗传优化器 GEQO 参数现在是 8 (Bruce)
允许在目标列表中使用带聚合的函数参数 (Vadim)
添加 JDBC 驱动作为接口 (Adrian & Peter)
pg_password 工具
返回 INSERT/UPDATE/DELETE 等插入/影响的行数 (Vadim)
用 CREATE TRIGGER 实现触发器 (SQL3) (Vadim)
SPI (服务器编程接口) 允许在 C 函数内
        执行查询 (Vadim)
实现 NOT NULL (SQL92) (Robson Paniago de Miranda)
纳入用于字符串处理、外连接和联合的保留字 (Thomas)
用独占状态实现扩展注释 ("/* ... */") (Thomas)
添加 "//" 单行注释 (Bruce)
移除对操作符名称中字符的一些限制 (Thomas)
实现表的 DEFAULT 和 CONSTRAINT (SQL92) (Vadim & Thomas)
添加文本连接操作符和函数 (SQL92) (Thomas)
支持 WITH TIME ZONE 语法 (SQL92) (Thomas)
支持 INTERVAL unit TO unit 语法 (SQL92) (Thomas)
定义类型 DOUBLE PRECISION、INTERVAL、CHARACTER、
        和 CHARACTER VARYING (SQL92) (Thomas)
定义类型 FLOAT(p) 和基础的 DECIMAL(p,s)、NUMERIC(p,s) (SQL92) (Thomas)
定义 EXTRACT()、POSITION()、SUBSTRING() 和 TRIM() (SQL92) (Thomas)
定义 CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP (SQL92) (Thomas)
为 UNION、HAVING、INNER 和 OUTER JOIN 添加语法和警告 (SQL92) (Thomas)
添加更多保留字，主要为符合 SQL92 (Thomas)
允许 timespan/reltime 类型的 hh:mm:ss 时间输入 (Thomas)
为 lseg、path、polygon 添加 center() 例程 (Thomas)
为 circle-polygon、polygon-polygon 添加 distance() 例程 (Thomas)
用轴交叉算法显式检查多边形内包含的
        点和多边形 (Thomas)
添加 circle-box 转换例程 (Thomas)
合并不同几何数据类型的冲突操作符 (Thomas)
将距离操作符 "<===>" 替换为 "<->" (Thomas)
把"上方"操作符 "!^" 替换为 ">^"，"下方"操作符 "!|" 替换为 "<^" (Thomas)
添加两端修剪文本、子串和字符串位置的例程 (Thomas)
添加转换例程 circle(box) 和 poly(circle) (Thomas)
允许内部排序存储在内存中而不是文件中 (Bruce & Vadim)
允许内部相同的类型上的函数和操作符成功执行 (Bruce)
在性能分析后加速后端启动 (Bruce)
为性能内联频繁调用的函数 (Bruce)
减少 open() 调用 (Bruce)
psql: 为 \h 和 \? 添加 PAGER，\C 修复
修复无 tty 时 psql 分页器的问题 (Bruce)
新 entab 工具 (Bruce)
用于引用完整性的通用触发器函数 (Vadim)
用于时间旅行的通用触发器函数 (Vadim)
用于 AUTOINCREMENT/IDENTITY 特性的通用触发器函数 (Vadim)
MOVE 实现 (Vadim)

源码树变更

HP-UX 10 补丁 (Vladimir Turin)
添加 SCO 支持 (Daniel Harris)
MkLinux 补丁 (Tatsuo Ishii)
把几何 box 术语从 "length" 改为 "width" (Thomas)
废弃几何代码中临时不存储的斜率字段 (Thomas)
从 INSTALL 移除重启说明 (Bruce)
先在 /usr/ucb 中查找 install (Bruce)
修复 c++ 复制示例代码 (Thomas)
在 psql 手册页添加 -o (Bruce)
防止 relname 未分配字符串长度被复制进数据库 (Bruce)
清理 NAMEDATALEN 的使用 (Bruce)
修复输出中超过 15 字符的 pg_proc 名 (Bruce)
添加 strNcpy() 函数 (Bruce)
移除一些不必要的 (void) 转换 (Bruce)
新的 interfaces 目录 (Marc)
用对 fd.c 函数的调用替换 fopen() 调用 (Bruce)
尽可能把函数设为 static (Bruce)
把未使用的函数包在 #ifdef NOT_USED 中 (Bruce)
移除时间戳支持中对 difftime() 的调用以修复 SunOS (Bruce & Thomas)
针对 Digital Unix 的变更
pg_dumpall 的可移植性修复 (Bruce)
把 pg_attribute.attnvals 改名为 attdisbursion (Bruce)
"intro/unix" 手册页现为 "pgintro" (Bruce)
"built-in" 手册页现为 "pgbuiltin" (Bruce)
"drop" 手册页现为 "drop_table" (Bruce)
添加 "create_trigger"、"drop_trigger" 手册页 (Thomas)
添加约束回归测试 (Vadim & Thomas)
添加注释语法回归测试 (Thomas)
添加 PGINDENT 和支持程序 (Bruce)
对所有 *.c 和 *.h 文件运行 PGINDENT 的大规模提交 (Bruce)
文件移到 /src/tools 目录 (Bruce)
SPI 和触发器编程指南 (Vadim & D'Arcy)

35.17. 版本 6.1.1

35.17.1. 从版本 6.1 迁移到版本 6.1.1

1997-07-22

这是次要缺陷修复版本。从 6.1 版本不需要转储/重载，但从 6.1 之前的任何版本需要。更多细节请参阅 6.1 的发行说明。

35.17.2. 变更

修复带选项的 SET (Thomas)
允许 pg_dump/pg_dumpall 保留所有表/对象的所有权 (Bruce)
新 psql \connect 选项允许只更改用户名而不更改数据库
修复 initdb --debug 选项 (Yoshihiko Ichikawa))
lextest 清理 (Bruce)
哈希修复 (Vadim)
修复日期/时间的月份边界算术 (Thomas)

修复某些移植的时区夏令时处理（Thomas、Bruce、Tatsuo）
timestamp 彻底改造为使用标准函数（Thomas）
日期/时间例程的其他代码清理（Thomas）
psql 的 \d 现在不区分大小写（Bruce）
psql 的反斜杠命令现在可以带尾部分号（Bruce）
修复使用 \g 时 psql 的内存泄漏（Bruce）
与服务器通信的字节序处理重大修复（Thomas、Tatsuo）
Solaris 汇编器和头文件的修复（Yoshihiko Ichikawa）
允许用户名中使用下划线（Bruce）
pg_dumpall 现在返回正确状态，可移植性修复（Bruce）

35.18. 版本 6.1

1997-06-08

回归测试已经为 Postgres 6.1 版本做了适配和大量修改。

新增了三种数据类型（datetime、timespan 和 circle），已加入到 Postgres 本地类型集中。点、框、路径和多边形在各数据类型间的输出格式已统一。misc.out 中的多边形输出只相对原始回归输出做了抽查。

Postgres 6.1 引入了使用遗传算法的新备选优化器。当查询包含多个限定条件或多个表时（优化器可选择求值顺序），这些算法会给查询结果的顺序引入随机行为。
已有几个回归测试被修改为显式排序结果，因此对优化器的选择不敏感。
少数回归测试针对本质上无序的数据类型（如点和时间间隔），涉及这些类型的测试显式地用 set geqo to 'off' 和 reset geqo 括起来。

数组说明符（原子值周围的花括号）
的解释似乎在最初的回归测试生成之后的某个时候发生了变化。当前的 ./expected/*.out 文件反映了这一新解释，它可能并不正确！

float8 回归测试至少在某些平台上失败。这是由于 pow() 和 exp() 实现的差异以及上溢/下溢条件所用的信号机制不同。

“random” 测试中的 “随机” 结果本应导致 “random” 测试 “失败”，因为回归测试是用简单的 diff 评估的。不过，在我的测试机（Linux/gcc/i686）上，“random” 似乎并不会产生随机结果。

35.18.1. 迁移到版本 6.1

此迁移需要完整转储 6.0 数据库并在 6.1 中恢复。

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

35.18.2. 变更

缺陷修复

库例程中的包长度检查
锁管理器优先级补丁
检查 float8 的下溢/上溢（Bruce）
多表连接修复（Vadim）
SIGPIPE 崩溃修复（Darren）

大对象修复 (Sven)
允许 btree 索引处理 NULL (Vadim)
时区修复 (D'Arcy)
无行时 select SUM(x) 可返回 NULL (Thomas)
内部优化器、执行器缺陷修复 (Vadim)
修复 < 或 <= 内层循环无行的问题 (Vadim)
防止连接索引子句重新交换 (Vadim)
修复多表的连接子句 (Vadim)
修复数组的哈希和哈希连接 (Vadim)
修复 abstime 类型的 btree (Vadim)
大对象修复 (Raymond)
修复哈希索引中的缓冲区泄漏 (Vadim)
修复 rtree 以用于内层扫描 (Vadim)
修复 gist 以用于内层扫描并清理 (Vadim、Andrea)
避免不必要的本地缓冲区分配 (Vadim、Massimo)
修复事务中止时本地缓冲区泄漏 (Vadim)
修复文件管理器内存泄漏并清理 (Vadim、Massimo)
修复存储管理器内存泄漏 (Vadim)
修复 btree 重复项处理 (Vadim)
修复 vacuum 导致已删元组复活的问题 (Vadim)
修复 SELECT varchar()/char() INTO TABLE 生成零长度字段的问题 (Bruce)
用 Purify 修复许多 psql、pg_dump 和 libpq 内存泄漏 (Igor)

增强

属性优化统计 (Bruce)
快得多的新 btree 批量装载代码 (Paul)
批量装载代码加入 BTREE UNIQUE (Vadim)
新锁调试代码 (Massimo)
libpg++ 的大规模变更 (Leo)
新 GEQO 优化器加速多表优化 (Martin)
对唯一键非唯一插入的新 WARN 消息 (Marc)
update x=-3 (无空格) 现在有效 (Bruce)
移除标识符的大小写敏感处理 (Bruce、Thomas、Dan)
调试后端现在美化打印树 (Darren)
新 Oracle 字符函数 (Edmund)
新明文口令函数 (Dan)
"no such class or insufficient privilege" 拆分为不同的消息 (Dan)
新 ANSI timestamp 函数 (Dan)
新 ANSI Time 和 Date 类型 (Thomas)
在后端中移动大数据块 (Martin)
多列 btree 索引 (Vadim)
新 SET var TO value 命令 (Martin)
读取时更新事务状态 (Dan)
字符类型的新区域设置 (Oleg)
新 SEQUENCE 序列号生成器 (Vadim)
现在可以使用 GROUP BY 函数 (Vadim)
重新组织回归测试 (Thomas、Marc)
新优化器操作权重 (Vadim)
新 psql \z 授予/许可选项 (Marc)
新 MONEY 数据类型 (D'Arcy、Thomas)
tcp 套接字通信速度改进 (Vadim)
VACUUM 的属性统计及特定列新选项 (Vadim)
许多几何类型改进 (Thomas、Keith)

附加回归测试 (Thomas)
新 datestyle 变量 (Thomas、Vadim、Martin)
更多用于排序的比较操作符 (Thomas)
新转换函数 (Thomas)
新更紧凑的 btree 格式 (Vadim)
允许 pg_dumpall 保留数据库所有权 (Bruce)
新 SET GEQO=# 和 R_PLANS 变量 (Vadim)
旧 (!GEQO) 优化器可使用右侧计划 (Vadim)
SQL 解析器类型检查改进 (Bruce)
新 SET、SHOW、RESET 命令 (Thomas、Vadim)
新 \connect database USER 选项
新 destroydb -i 选项 (Igor)
新 \dt 和 \di psql 命令 (Darren)
SELECT "\n" 现在转义换行 (A. Duursma)
从旧格式转换的新几何转换函数 (Thomas)

源代码树变更

新配置脚本 (Marc)
添加 readline 配置选项 (Marc)
移除操作系统特定的配置选项 (Marc)
新的操作系统特定模板文件 (Marc)
不再需要编辑 Makefile.global (Marc)
重新安排头文件 (Marc)
nextstep 补丁 (Gregor Hoffleit)
移除 WIN32 专用代码 (Bruce)
移除 postmaster -e 选项，现在只有 postgres -e 选项 (Bruce)
合并前后端中重复的库代码 (Martin)
现在可与 eBones、国际 Kerberos 一起工作 (Jun)
更多共享库支持
c++ 头文件清理 (Bruce)
对有缺陷的 flex 发出警告 (Bruce)
DG/UX、Ultrix、IRIX、AIX 可移植性修复

35.19. 版本 6.0

1997-01-29

希望从以前的 Postgres 版本迁移数据的用户，需要进行转储/恢复。

35.19.1. 从版本 1.09 迁移到版本 6.0

此迁移需要完整转储 1.09 数据库并在 6.0 中恢复。

35.19.2. 从 1.09 之前版本迁移到版本 6.0

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

35.19.3. 变更

缺陷修复

ALTER TABLE 缺陷—运行中的 postgres 进程需要重新读取表定义
允许对一个表或整个数据库运行 vacuum (Bruce)
数组修复
修复数组内存写的越界 (Kurt)
修复难以捉摸的 btree 范围/非范围缺陷 (Dan)
修复某些类型 (如 time 和 date) 上的哈希索引
修复 pg_log 大小爆炸的问题
修复 lo_export() 的权限 (Bruce)
修复未初始化的内存读取 (Kurt)
修复 ALTER TABLE ... char(3) 缺陷 (Bruce)
修复了几个小的内存泄漏
修复 EXPLAIN 的选项处理并更改了 full_path 选项名
修复组 acl 权限的输出
内存泄漏 (用 Purify 等工具查找并消灭) (Kurt)
规则系统的小改进
NOTIFY 修复
新的运行检查断言
彻底改造 parser/analyze 代码以正确报告错误并提高速度
Pg_dump -d 现在正确处理 NULL (Bruce)
防止 SELECT NULL 使服务器崩溃 (Bruce)
INSERT ... SELECT 列不匹配时正确报告错误
插入列名不正确时正确报告错误
psql \g filename 现在可用 (Bruce)
修复 psql 一行多语句多输出的问题
移除重复的系统 OID
SELECT * INTO TABLE . GROUP/ORDER BY 在表已存在时给出 unlink 错误 (Bruce)
几项对使后端崩溃查询的修复
插入字符串中起始引号的错误 (Bruce)
提交空查询现在返回空状态，而不只是 " " 查询 (Bruce)

增强

添加 EXPLAIN 手册页 (Bruce)
添加 UNIQUE 索引能力 (Dan)
添加主机名/用户级访问控制，而不只是主机名和用户
为 <> 添加 != 同义词 (Bruce)
允许 "select oid,* from table"
允许 BY、ORDER BY 按编号或按非别名的 table.column 指定列 (Bruce)
允许从前端 COPY (Bryan)
允许 GROUP BY 使用别名列名 (Bruce)
允许真正的压缩，而不只是同一页上的重用 (Vadim)
允许安装配置选项自动添加所有本地用户 (Bryan)
允许 libpq 区分文本值 '' 和 null (Bruce)
允许有 createdb 权限的非 postgres 用户 destroydb
允许限制谁可以创建 C 函数 (Bryan)
允许限制谁可以做后端 COPY (Bryan)
可以收缩表、pg_time 和 pg_log (Vadim & Erich)
更改调试级别 2 为只打印查询，更改调试标题布局 (Bruce)
把十进制常量的默认表示从 float4 改为 float8 (Bruce)
postmaster 启动时现在设置欧洲日期格式
找不到精确大小写时执行小写函数名
聚合/GROUP 处理的修复，允许 'select sum(func(x),sum(x+y) from z'
Gist 现在包含在发行版中 (Marc)
本地用户的 Ident 认证 (Bryan)

实现 BETWEEN 限定符 (Bruce)
实现 IN 限定符 (Bruce)
libpq 有 PQgetisnull() (Bruce)
libpq++ 改进
initdb 的新选项 (Bryan)
pg_dump 允许转储 OID (Bruce)
Pg_dump 为提高速度在表装载后创建索引 (Bruce)
Pg_dumpall 转储所有数据库和用户表
Pginterface 对 NULL 值的补充 (Bruce)
防止以 root 运行 postmaster
psql \h 和 \? 现在可读 (Bruce)
psql 允许行内任意位置的反斜杠和分号 (Bruce)
psql 更改查询中或引号中行的命令提示符 (Bruce)
psql char(3) 现在在 \d 输出中显示为 (bp)char (Bruce)
psql 返回码现在更准确 (Bryan?)
psql 更新的帮助语法 (Bruce)
重新检查并修复 vacuum (Vadim)
减小回归 diff 的大小，移除时区名差异 (Bruce)
移除编译期参数以支持二进制发行 (Bryan)
反转 HBA 掩码的含义 (Bryan)
本地用户的安全认证 (Bryan)
加快 vacuum (Vadim)
Vacuum 现在有 VERBOSE 选项 (Bruce)

源代码树变更

所有函数现在都有与调用比较的原型
允许从 Makefile.global 轻松禁用断言 (Bruce)
把代码中使用的 oid 常量改为 #define 名称
解耦 sparc 和 solaris 定义 (Kurt)
Gcc -Wall 干净编译，警告只来自无法修复的构造
主要的头文件重组/精简 (Marc)
编译失败时 make 现在停止 (Bryan)
Makefile 重构 (Bryan、Marc)
把 bsdi_2_1 合并到 bsdi (Bruce)
移除 Monitor 程序
从 Postgres95 改名为 PostgreSQL
新 config.h 文件 (Marc、Bryan)
PG_VERSION 现在设为 6.0 并被 postmaster 使用
可移植性补充，包括 Ultrix、DG/UX、AIX 和 Solaris
减少 #define 的数量，集中化 #define
移除系统表中的重复 OID (Dan)
移除重复的系统目录信息或报告不匹配 (Dan)
移除了许多操作系统特定的 #define
重构目标文件的生成/位置 (Bryan、Marc)
重构移植特定文件的位置 (Bryan、Marc)
纠正未使用/未初始化的变量

35.20. 版本 1.09

Unknown 抱歉，我们停止了记录 1.02 到 1.09 的变更。6.0 中列出的一些变更实际上包含在 1.02.1 到 1.09 版本中。

35.21. 版本 1.02

35.21.1. 从版本 1.02 迁移到版本 1.02.1

1996-08-01

这是 1.02.1 的新迁移文件。它包括 'copy' 变更和一个把旧 ascii 文件转换为新格式的脚本。

> **注意**
>
> 以下说明是为了帮助希望把数据库从 postgres95 1.01 和 1.02 迁移到 postgres95 1.02.1 的用户。
>
> 如果你全新开始使用 postgres95 1.02.1 而不需要迁移旧数据库，则无需再往下读。

为了把较旧的 postgres95 1.01 或 1.02 版本数据库升级到 1.02.1 版本，需要以下步骤：

1. 启动新的 1.02.1 postmaster

2. 将 1.02.1 新增的内建函数和操作符添加到 1.01 或 1.02 数据库中。方法是用新的 1.02.1 服务器运行你自己的 1.01 或 1.02 数据库，并应用本文件末尾附带的查询。这可以通过 `psql` 轻松完成。如果你的 1.01 或 1.02 数据库名为 `testdb`，并且你已把命令从本文件末尾剪出并保存到 `addfunc.sql` 中：

   ```
   % psql testdb -f addfunc.sql
   ```

   从 1.02 升级的用户在执行文件中的最后两条语句时会得到警告，因为它们已经存在于 1.02 中。这无需担心。

35.21.2. 转储/重载程序

如果你试图重新装载由旧版本生成的 pg_dump 或文本模式 `copy tablename to stdout` 输出，需要先对 ASCII 文件运行附带的 `sed` 脚本，再将其装入数据库。旧格式用 '.' 作为数据结束标记，而现在的数据结束标记是 '\.'。此外，空字符串现在装载为 '' 而非 NULL。完整细节见 copy 手册页。

```
sed 's/^\.$/\\./g' <in_file >out_file
```

如果你装载的是较旧的二进制 copy 或非 stdout 的 copy，则没有数据结束字符，因此无需转换。

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure =
 texticregexne);
```

35.21.3. 变更

源代码维护与开发
 * 全球志愿者团队
 * 源代码树现在在 ftp.ki.net 的 CVS 中

增强
 * psql（及底层 libpq 库）现在有更多输出格式化选项，
   包括 HTML
 * pg_dump 现在可以输出模式和/或数据，并有许多修复
   以提高完整性。
 * 管理 shell 脚本中用 psql 代替 monitor。
   monitor 将在下一个发行版中废弃。
 * 日期/时间函数得到增强
 * NULL 插入/更新/比较修复/增强
 * TCL/TK 库和 shell 修复为同时支持 tck7.4/tk4.0 和 tcl7.5/tk4.1

缺陷修复（多到几乎无法列举）
 * 索引
 * 存储管理
 * 解引用前检查 NULL 指针
 * Makefile 修复

新移植
 * 新增 SolarisX86 移植
 * 新增 BSD/OS 2.1 移植
 * 新增 DGUX 移植

35.22. 版本 1.01

35.22.1. 从版本 1.0 迁移到版本 1.01

1996-02-23

以下说明是为了帮助希望把数据库从 postgres95 1.0 迁移到 postgres95 1.01 的用户。

如果你全新开始使用 postgres95 1.01 而不需要迁移旧数据库，则无需再往下读。

为了在用 postgres95 1.0 版本创建的数据库上运行 postgres95 1.01 版本，需要以下步骤：

1. 把 src/Makefile.global 中 NAMEDATALEN 的定义改为 16，OIDNAMELEN 改为 20。

2. 决定是否要使用基于主机的认证。

   a. 如果要用，你必须在顶级数据目录（通常是 $PGDATA 的值）中创建名为 pg_hba 的文件。src/libpq/pg_hba 给出了示例语法。

   b. 如果不想使用基于主机的认证，可以注释掉 src/Makefile.global 中的这一行：

      ```
      HBA = 1
      ```

      注意，基于主机的认证默认开启，如果你不执行上面的 A 或 B 步骤，开箱即用的 1.01 将不允许你连接到 1.0 数据库。

3. 编译并安装 1.01，但不要执行 `initdb` 步骤。

4. 在做任何其他事情之前，终止你的 1.0 postmaster，并备份现有的 `$PGDATA` 目录。

5. 把 `PGDATA` 环境变量指向你的 1.0 数据库，但把路径设置为使用 1.01 二进制文件。

6. 把文件 `$PGDATA/PG_VERSION` 从 5.0 改为 5.1

7. 启动新的 1.01 postmaster

8. 将 1.01 新增的内建函数和操作符添加到 1.0 数据库中。方法是用新的 1.01 服务器运行你自己的 1.0 数据库，并应用随附保存在文件 1.0_to_1.01.sql 中的查询。这可以通过 `psql` 轻松完成。如果你的 1.0 数据库名为 `testdb`：

```
% psql testdb -f 1.0_to_1.01.sql
```

然后执行以下命令（从此处剪切并粘贴）：

```
-- add builtin functions that are new to 1.01

create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as
 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as
 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure =
 int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure =
 oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure =
 char2icregexeq);
```

```
create operator !~* (leftarg = char2, rightarg = text, procedure =
 char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure =
 char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure =
 char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure =
 char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure =
 char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure =
 char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure
 = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure =
 texticregexne);
```

35.22.2. 变更

不兼容之处:
 * 只要用户按照 MIGRATION_from_1.0_to_1.01 文件中概述的步骤操作，
   1.01 与 1.0 数据库向后兼容。
   如果未采取这些步骤，1.01 与 1.0 数据库不兼容。

增强:
 * 为 libpq 添加 PQdisplayTuples() 并更改 monitor 和 psql 使用它
 * 新增 NeXT 移植（需要 SysVIPC 实现）
 * 新增 CAST .. AS 语法
 * 添加了 ASC 和 DESC 关键字
 * 为 CREATE FUNCTION 添加 'internal' 作为可能的语言
   内部函数是已静态链接
   到 postgres 后端中的 C 函数。
 * 为系统标识符（表名、
   属性名等）添加了新类型 "name"。它取代了旧的 char16 类型。
   name 的长度由 src/Makefile.global 中的 NAMEDATALEN #define 设置
 * 一本描述查询语言的易读参考手册。
 * 添加了基于主机的访问控制。配置文件 ($PGDATA/pg_hba)
   用于保存配置数据。如果不想要基于主机的访问控制，
   请注释掉 src/Makefile.global 中的 HBA=1。
 * 更改正则表达式处理为统一使用 Henry Spencer 的正则代码，
   与平台无关。正则代码包含在发行版中
 * 添加了用于不区分大小写正则表达式的函数和操作符。
   操作符为 ~* 和 !~*。
 * pg_dump 使用 COPY 而非 SELECT 循环以获得更好性能

缺陷修复:
 * 修复了一个优化器缺陷，当在 WHERE 子句的比较中使用
   函数调用时会导致核心转储
 * 将所有 getuid 的使用改为 geteuid，以使用有效 uid
 * psql 使用 -c 时出错现在返回非零状态
 * 应用了公开补丁 1-14

35.23. 版本 1.0

35.23.1. 变更

1995-09-05

版权变更:
* Postgres 1.0 的版权已放宽为可自由修改并可用于任何目的。
  请阅读 COPYRIGHT 文件。感谢 Michael Stonebraker 教授使之成为可能。

不兼容之处:
* 日期格式必须为 MM-DD-YYYY (如果使用
欧洲风格，则为 DD-MM-YYYY) 。这遵循 SQL-92 规范。
* "delimiters" 现在是一个关键字

增强:
* 添加了 SQL LIKE 语法
* copy 命令现在接受可选的 USING DELIMITER 说明。
分隔符可以是任何单字符字符串。
* 添加了 IRIX 5.3 移植。
感谢 Paul Walmsley 等人。
* 更新 pg_dump 以配合新 libpq 工作
* psql 中添加了 \d
感谢 Keith Parks
* 对于使用 POSIX 正则的体系结构，正则性能
通过缓存预编译模式得到了改进。
感谢 Alistair Crooks
* 新版本的 libpq++
感谢 William Wanders

缺陷修复:
* 可以在 createuser 脚本中指定任意用户 id
* psql 中连接其他数据库的 \c 现在可以工作了。
* 修复了 float4inc() 的错误 pg_proc 条目
* 设置了 usecreatedb 字段的用户现在可以创建数据库，
而无需是 usesuper
* 当条目不再有任何权限时移除访问控制条目
* 修复了不可移植的 datetimes 实现
* 在 src/backend/Makefile 中添加 kerberos 标志
* libpq 现在支持 kerberos
* 用户手册中的排印错误已更正。
* 多索引的 btree 从未工作过，现在尝试使用时会
明确告知它们不可用

35.24. Postgres95 Beta 0.03

35.24.1. 变更

1995-07-21

不兼容变更:

```
 * BETA-0.3 与用以前版本创建的数据库不兼容
   （由于系统目录更改和索引结构更改）。
 * 双引号（"）作为字符串字面量的引号字符已被废弃;
   你需要将它们转换为单引号（'）。
 * 聚合的名称（如 int4sum）已按照
   SQL 标准重命名（如 sum）。
 * CHANGE ACL 语法被 GRANT/REVOKE 语法取代。
 * 浮点字面量（如 3.14）现在是 float4 类型（而不是以前版本的
   float8）；如果你依赖它
   是 float8 类型，可能需要进行类型转换。如果忽略类型转换而把
   浮点字面量赋给 float8 类型的字段，可能会得到错误的
   存储值!
 * LIBPQ 经过彻底翻新，前端应用程序
   可以连接多个后端
 * pg_user 中的 usesysid 字段已从 int2 改为 int4，以
   允许更宽范围的 Unix 用户 id。
 * netbsd/freebsd/bsd 操作系统移植已合并为
   单一的 BSD44_derived 移植。（感谢 Alistair Crooks）

SQL 标准符合性（以下详述使 postgres95
更符合 SQL-92 标准的更改）：
 * 以下 SQL 类型现在是内建的：smallint、int(eger)、float、real、
   char(N)、varchar(N)、date 和 time。

   以下是现有 postgres 类型的别名：
               smallint -> int2
               integer, int -> int4
               float, real  -> float4
   char(N) 和 varchar(N) 实现为截断的 text 类型。此外，
   char(N) 会做空白填充。
 * 用单引号（'）引用字符串字面量；''（除
   \' 外）也可用于在字符串中插入单引号
 * 使用 SQL 标准聚合名（MAX、MIN、AVG、SUM、COUNT）
   （此外，聚合现在可以重载，即你可以定义
   接受用户自定义类型的 MAX 聚合。）
 * 移除 CHANGE ACL。添加 GRANT/REVOKE 语法。
   - 可以使用 "GROUP" 关键字把权限授予组。
        例如：
                GRANT SELECT ON foobar TO GROUP my_group;
        关键字 'PUBLIC' 也受支持，表示所有用户。

        每次只能对一个用户或组授予或撤销
        权限。

        不支持 "WITH GRANT OPTION"。只有类所有者可以更改
        访问控制
   - 默认访问控制是给用户只读访问权限。
     你必须显式授予用户插入/更新权限。要更改
     这一点，请修改
               src/backend/utils/acl.h
     中定义 ACL_WORLD_DEFAULT 的那一行

缺陷修复：
 * 空表上的聚合不被运行的缺陷已修复。现在，
```

   在空表上运行的聚合将返回聚合的初始条件。因此，空表的 COUNT 现在会正确返回 0。
   空表的 MAX/MIN 将返回值为 NULL 的元组。
 * 允许在 monitor 中使用 \;
 * LISTEN/NOTIFY 异步通知机制现在可以工作
 * 规则动作体中的 NOTIFY 现在可以工作
 * 哈希索引可以工作了，访问方法总体上性能应该更好。
   创建大型 btree 索引应该快得多。（感谢 Paul
   Aoki）

其他更改与增强：
 * 添加了用于解释查询执行计划的 EXPLAIN 语句（例如 "EXPLAIN SELECT * FROM
 EMP" 会打印出该查询的执行计划）。
 * WARN 和 NOTICE 消息不再带时间戳。要打开错误消息的时间戳，请取消注释 src/
backend/utils/elog.h 中的这一行：
        /* define ELOG_TIMESTAMPS */
 * 访问控制被违反时，会给出消息
        "Either no such class or insufficient privilege"。
   这与找不到类时返回的消息相同。这可以阻止无权限的用户
   猜测受权限保护的类的存在。
 * 还做了一些用户不可见的
   系统目录更改。

libpgtcl 更改：
 * "pg_result" tcl 命令添加了 -oid 选项。
   pg_result -oid 返回最后插入元组的 oid。如果最后的命令不是 INSERT，则 pg_
result -oid 返回 ""。
 * 大对象接口以 pg_lo* tcl 命令的形式提供：
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

可移植性增强与新移植：
 * flex/lex 的问题已经解决。现在，在任何平台上都应该可以用 flex 代替 lex。
   我们不再根据你所用的平台假设你使用的词法分析器。
 * 现在支持 Linux-ELF 移植。已测试多种配置：已知以下配置可以工作：
        kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   所有内容均为 ELF 格式，

新实用程序：
 * ipcclean 已加入发行版
   通常不需要运行 ipcclean，但如果你的后端崩溃
   并留下共享内存段，ipcclean 会为你清理它们。

新文档：
 * 用户手册已经修订并添加了 libpq 文档。

35.25. Postgres95 Beta 0.02

35.25.1. 变更

1995-05-25

不兼容变更：
 * 创建数据库的 SQL 语句是 'CREATE DATABASE' 而不是
   'CREATEDB'。类似地，删除数据库是 'DROP DATABASE' 而不是 'DESTROYDB'。

但可执行文件名 'createdb' 和 'destroydb' 保持不变。

新工具:
 * pgperl - Postgres95 的 Perl (4.036) 接口
 * pg_dump - 把 postgres 数据库转储为包含查询命令的
脚本文件的工具。脚本文件为 ASCII 格式，可用于重建数据库，甚至在其他机器和其他体系结构上。（也适合把 Postgres 4.2 数据库转换为 Postgres95 数据库。）

以下移植已并入 postgres95-beta-0.02:
 * Alistair Crooks 的 NetBSD 移植
 * Mike Tung 的 AIX 移植
 * Jon Forrest 的 Windows NT 移植（内容更多但尚未完成）
 * Brian Gallew 的 Linux ELF 移植

postgres95-beta-0.02 中修复了以下缺陷:
 * COPY OUT 中换行未转义以及首属性为 '.' 时 COPY OUT 的问题
 * createuser 中无法键入回车以使用默认用户 id
 * 大表上的 SELECT DISTINCT 崩溃
 * Linux 安装问题
 * monitor 不允许使用 'localhost' 作为 PGHOST
 * psql 执行 \c 或 \l 时核心转储
 * src/bin/pgtclsh/Makefile 中缺少 "pgtclsh" 目标
 * libpgtcl 有硬编码的默认端口号
 * SELECT DISTINCT INTO TABLE 挂起
 * CREATE TYPE 不接受 'variable' 作为 internallength
 * 在 SELECT 中使用多个聚合时结果错误

35.26. Postgres95 Beta 0.01

1995-05-01

初始发行版。

35.27. 计时结果

这些计时结果来自使用以下命令运行回归测试

```
% cd src/test/regress
% make all
% time make runtest
```

Linux 2.0.27 下的计时在每次运行之间似乎有大约 5% 的波动，这大概是多任务系统调度的不确定性所致。

35.27.1. 版本 6.5

与以前的版本一样，各版本之间的计时不能直接比较，因为加入了新的回归测试。总体而言，6.5 比以前的版本更快。

禁用 fsync() 时的计时:

时间 系统

```
  02:00  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.
2.3 -O2 -m486
  04:38  Sparc Ultra 1 143MHz, 64MB, Solaris 2.6
```

启用 `fsync()` 时的计时：

```
  时间    系统
  04:21  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.
2.3 -O2 -m486
```

对于上面的 Linux 系统，使用 UW-SCSI 磁盘而非（较旧的）IDE 磁盘可使回归测试的速度提升 50%。

35.27.2. 版本 6.4beta

本版本的耗时与以前版本不能直接比较，因为纳入了一些额外的回归测试。不过总体而言，6.4 应比前一版本略快（谢谢 Bruce！）。

```
  时间    系统
  02:26  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.
1 -O2 -m486
```

35.27.3. 版本 6.3

本版本的耗时与以前版本不能直接比较，
因为纳入了一些额外的回归测试并移除了一些涉及时间旅行的过时测试。不过总体而言，6.3 比以前的版本快得多（谢谢 Bruce！）。

```
  时间    系统
  02:30  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.
1 -O2 -m486
  04:12  Dual Pentium Pro 180, 96MB, EIDE, Linux 2.0.30, gcc 2.7.2.1 -
O2 -m486
```

35.27.4. 版本 6.1

```
  时间    系统
  06:12  Pentium Pro 180, 32MB, EIDE, Linux 2.0.30, gcc 2.7.2 -O2 -
m486
  12:06  P-100, 48MB, Linux 2.0.29, gcc
  39:58  Sparc IPC 32MB, Solaris 2.5, gcc 2.7.2.1 -O -g
```

部分 III. 程序员指南

扩展 Postgres 的信息。

目录

第 36 章 体系结构

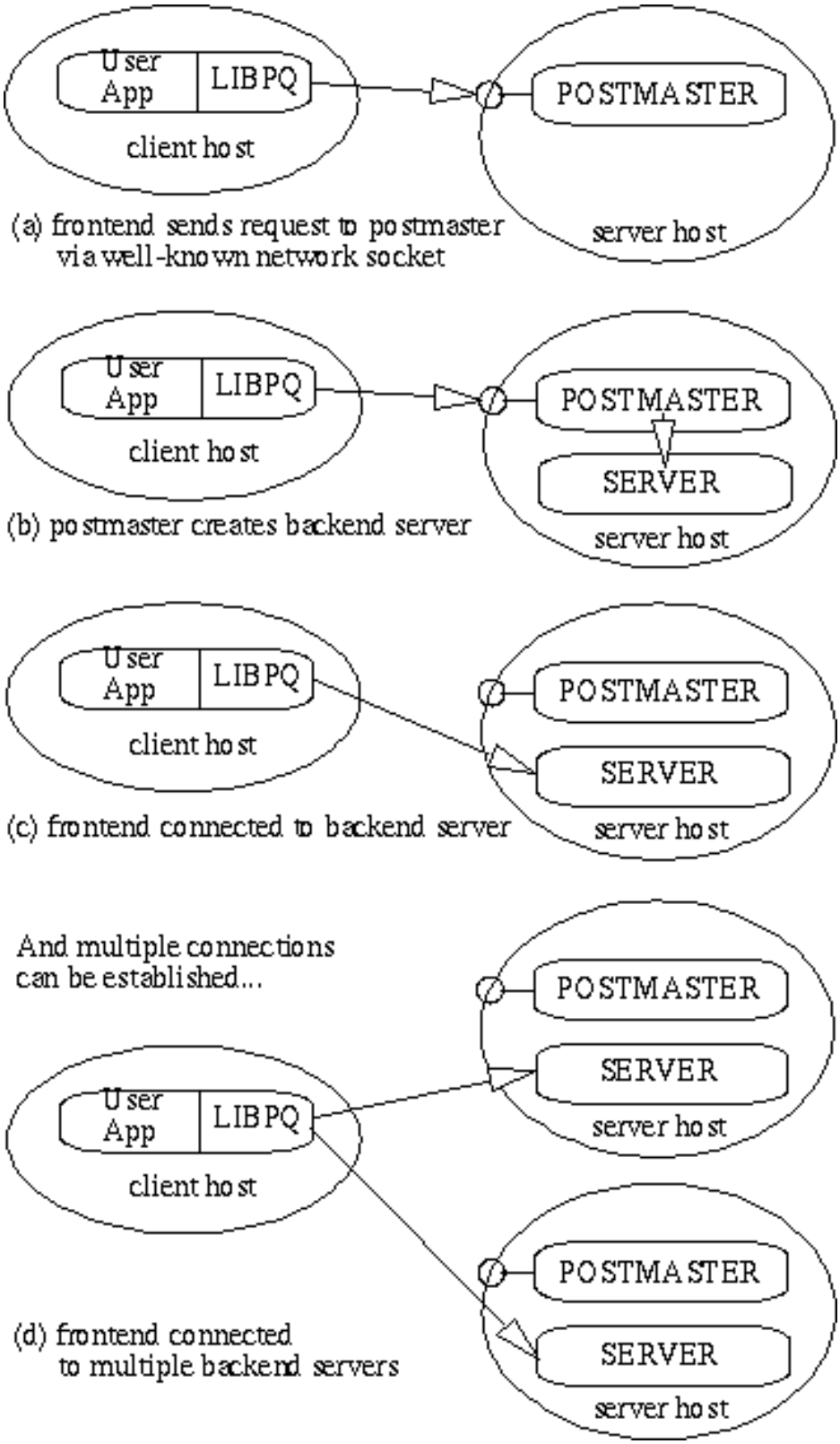
36.1. Postgres 体系结构概念

在继续之前，你应当理解 Postgres 系统的基本体系结构。理解 Postgres 各部分的交互方式会让下一章更容易理解。用数据库术语来说，Postgres 使用简单的"每用户一进程"客户端/服务器模型。一个 Postgres 会话由下列协作的 Unix 进程（程序）组成：

- 一个管理守护进程（postmaster），
- 用户的前端应用（例如 psql 程序），以及
- 一个或多个后端数据库服务器（postgres 进程本身）。

单个 postmaster 管理单个主机上给定的数据库集合。这样的数据库集合称为一个安装或站点。希望访问安装中某个数据库的前端应用调用该库。该库通过网络把用户请求发送给 postmaster (图 36.1(a))，后者再启动一个新的后端服务器进程 (图 36.1(b))

图 36.1. 如何建立连接


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
SERVER
(b) postmaster creates backend server
(c) frontend connected to backend server
And multiple connections can be established...
(d) frontend connected to multiple backend servers

并把前端进程连接到新服务器 (图 36.1(c))。从那时起，前端进程和后端服务器之间的通信不再需要 postmaster 干预。因此，postmaster 总是在运行并等待请求，而前端和后端进程则来来去去。libpq 库允许单个前端对后端进程建立多个连接。但前端应用仍然是一个单线程进程。libpq 目前不支持多线程的前端/后端连接。这一体系结构的一个含义是 postmaster 和后端总是运行在同一台机器（数据库服务器）上，而前端应用可以运行在任何地方。你应当牢记这一点，
因为在客户端机器上可以访问的文件在数据库服务器机器上可能无法访问（或者只能用不同的文件名访问）。你还应当知道 postmaster 和 postgres 服务器以 Postgres "superuser."（超级用户）的 user-id 运行。注意 Postgres 超级用户不必是任何特定的用户（例如名为 "postgres" 的用户），虽然许多系统就是这样安装的。而且，Postgres 超级用户绝对不应该是 Unix 超级用户 "root"！无论如何，与数据库相关的所有文件都应属于这个 Postgres 超级用户。

第 37 章 扩展 SQL：概览

在后面的各节中，我们将讨论如何通过添加下列内容来扩展 Postgres 的 SQL 查询语言：

- 函数
- 类型
- 操作符
- 聚集

37.1. 可扩展性如何运作

Postgres 之所以可扩展，是因为它的操作是目录驱动的。如果你熟悉标准的关系系统，就知道它们把关于数据库、表、列等的信息存储在通常所谓的系统目录中。
（有些系统称之为数据字典。）这些目录在用户看来是像其他类一样的类，但 DBMS 把它的内部簿记存储在其中。Postgres 与标准关系系统之间的一个关键区别是，Postgres 在其目录中存储的信息多得多——不仅有关于表和列的信息，还有关于其类型、函数、访问方法等的信息。
用户可以修改这些类，而由于 Postgres 把它的内部操作建立在这些类之上，这意味着 Postgres 可以由用户扩展。相比之下，传统数据库系统只能通过更改 DBMS 内部的硬编码过程或装载由 DBMS 厂商专门编写的模块来扩展。

Postgres 与大多数其他数据管理器的另一个不同之处在于，服务器可以通过动态装载把用户编写的代码纳入自身。也就是说，
用户可以指定一个实现新类型或函数的目标代码文件（例如编译好的 .o 文件或共享库），Postgres 会按需装载它。用 SQL 编写的代码加入服务器更是轻而易举。这种"即时"修改操作的能力使 Postgres 特别适合新应用和存储结构的快速原型开发。

37.2. Postgres 类型系统

Postgres 的类型系统可以从几个方面细分。类型分为基本类型和复合类型。基本类型是像 int4 这样用 C 之类的语言实现的类型。它们大体对应于通常所说的"抽象数据类型"；Postgres 只能通过用户提供的方法操作这类类型，并且只在用户描述它们的程度上理解这类类型的行为。
复合类型在用户创建类时创建。EMP 是复合类型的一个例子。

Postgres 只以一种方式存储这些类型（在存储类的所有实例的文件内），但用户可以从查询语言"窥视"这些类型的属性，并通过（例如）在属性上定义索引来优化它们的检索。Postgres 的基本类型进一步分为内置类型和用户定义类型。内置类型（如 int4）是编译进系统的那些。用户定义类型是用户以下面要描述的方式创建的类型。

37.3. 关于 Postgres 系统目录

介绍了基本的可扩展性概念之后，我们现在可以看看目录实际上是如何组织的。
现在可以跳过本节，但后面的某些章节如果没有这里给出的信息将无法理解，
所以请给这一页做个标记以便以后参考。所有系统目录的名称都以 pg_ 开头。下面的类包含可能对最终用户有用的信息。（还有很多其他系统目录，但很少需要直接查询它们。）

表 37.1. Postgres 系统目录

目录名	描述
pg_database	数据库
pg_class	类
pg_attribute	类属性

目录名	描述
pg_index	辅助索引
pg_proc	过程（C 和 SQL 都是）
pg_type	类型（基本和复合都是）
pg_operator	操作符
pg_aggregate	聚集和聚集函数
pg_am	访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_opclass	访问方法操作符类

图 37.1. Postgres 的主要系统目录

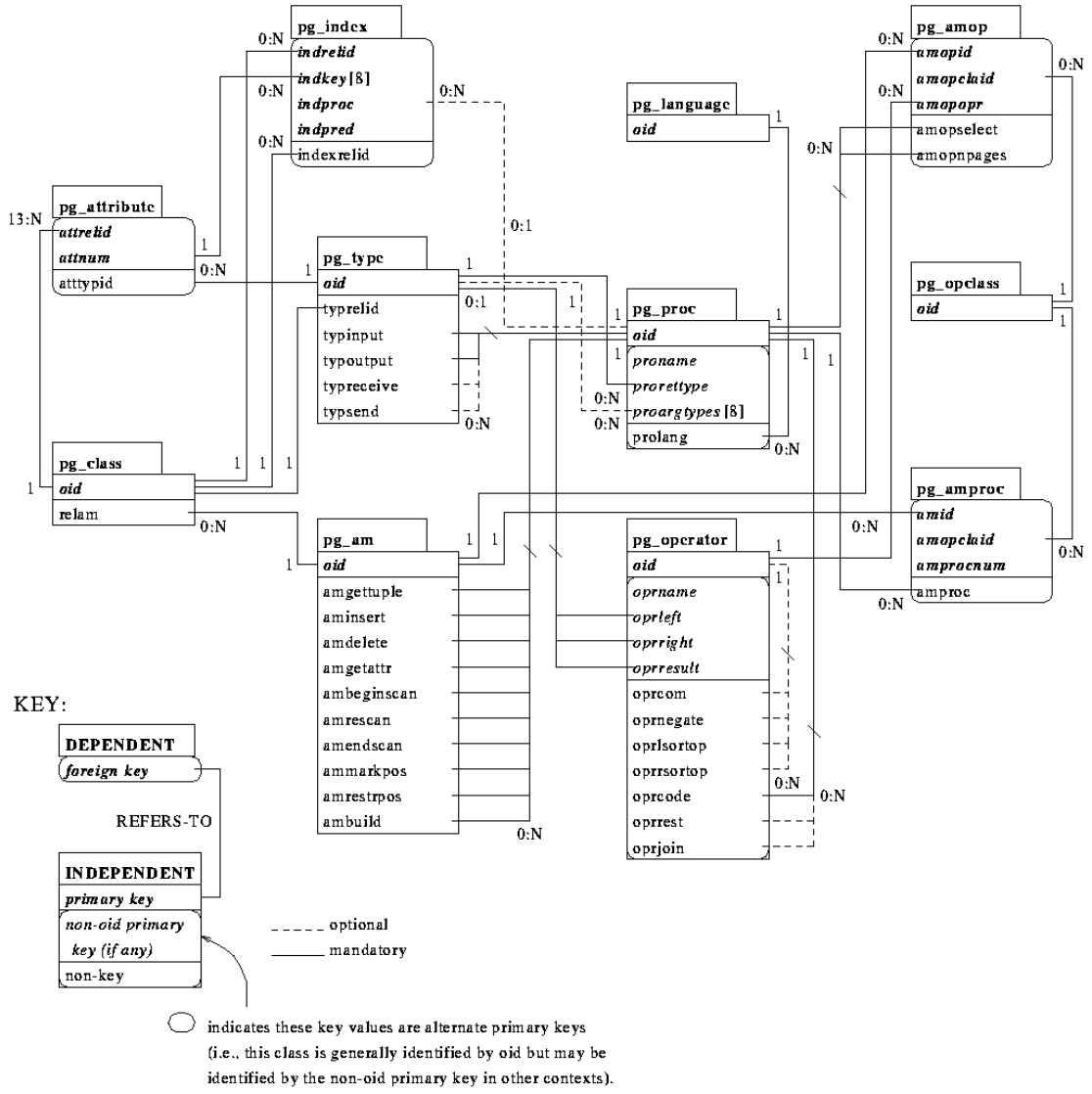


Figure 3. The major POSTGRES system catalogs.

参考手册对这些目录及其属性给出了更详细的解释。不过，图 37.1展示了系统目录中的主要实体及其关系。（不引用其他实体的属性没有显示，除非它们是主键的一部分。）
在你真正开始查看目录的内容并了解它们如何相互关联之前，这张图或多或少难以理解。
就目前而言，从这张图中要了解的要点如下：

- 在后面的几节中，我们将给出系统目录上的各种连接查询，
  它们显示我们扩展系统所需的信息。
  看这张图应该会让其中一些连接查询（它们通常是三路或四路连接）更容易理解，
  因为你将能看到查询中使用的属性在其他类中构成外键。
- 许多不同的特性（类、属性、函数、类型、访问方法等）在这个模式中紧密集成。一条简单的 create 命令就可能修改其中许多目录。
- 类型和过程是这个模式的核心。

  注意

  我们或多或少可以互换地使用过程和函数这两个词。

  几乎每个目录都包含对这两个类之一或两者的实例的引用。例如，Postgres 经常使用类型签名（例如函数和操作符的签名）来标识其他目录的唯一实例。
- 有许多属性和关系的含义是显而易见的，但也有许多（特别是那些与访问方法有关的）
  并非如此。pg_am、pg_amop、pg_amproc、pg_operator 和 pg_opclass 之间的关系特别难以理解，我们将在讨论了基本扩展之后（在把类型和操作符接口到索引的一节中）
  深入描述它们。

第 38 章 扩展 SQL：函数

事实证明，定义新类型的一部分工作就是定义描述其行为的函数。因此，
虽然可以不定义新类型就定义新函数，反过来却不成立。所以我们在描述如何向 Postgres 添加新类型之前，先描述如何添加新函数。

Postgres SQL 提供三种类型的函数：

- 查询语言函数（用 SQL 编写的函数）
- 过程语言函数（例如用 PLTCL 或 PLSQL 编写的函数）
- 编程语言函数（用 C 这样的编译型编程语言编写的函数）

每种函数都可以接受基本类型、复合类型或它们的某种组合作为参数。此外，
每种函数都可以返回基本类型或复合类型。定义 SQL 函数最容易，所以我们从它开始。本节的示例也可以在 `funcs.sql` 和 `funcs.c` 中找到。

38.1. 查询语言（SQL）函数

SQL 函数执行一个任意的 SQL 查询列表，返回列表中最后一个查询的结果。SQL 函数一般返回集合。如果其返回类型没有指定为 `setof`，则返回最后一个查询结果中的一个任意元素。

AS 之后的 SQL 函数体应是用空白字符分隔并用引号括起的查询列表。注意，查询中使用的引号必须转义，即在前面加两个反斜杠。

SQL 函数的参数可以在查询中用 $n 语法引用：$1 指第一个参数，$2 指第二个参数，依此类推。如果参数是复合类型，则可以使用点记法（例如 "$1.emp"）访问参数的属性或调用函数。

38.1.1. 示例

为说明一个简单的 SQL 函数，考虑下面这个可以用来给银行账户扣款的例子：

```
CREATE FUNCTION tp1 (int4, float8)
RETURNS int4
AS ' UPDATE bank
 SET balance = bank.balance - $2
        WHERE bank.acctountno = $1;
        SELECT 1;'
LANGUAGE 'sql';
```

用户可以执行如下命令给账户 17 扣款 $100.00：

```
select (x = TP1( 17,100.0));
```

下面的示例更有趣，它接受一个 EMP 类型的参数并检索多个结果：

```
select function hobbies (EMP) returns set of HOBBIES
    as 'select HOBBIES.* from HOBBIES
        where $1.name = HOBBIES.person'
    language 'sql';
```

38.1.2. 基本类型上的 SQL 函数

最简单的 SQL 函数没有参数，只是返回一个基本类型，例如 `int4`：

```
CREATE FUNCTION one()
RETURNS int4
AS 'SELECT 1 as RESULT;'
LANGUAGE 'sql';

SELECT one() AS answer;

     +-------+
     |answer |
     +-------+
     |1      |
     +-------+
```

注意我们为函数定义了一个目标列表（名为 RESULT），但调用该函数的查询的目标列表覆盖了函数的目标列表。因此结果被标记为 answer 而不是 one。

定义以基本类型为参数的 SQL 函数几乎同样容易。在下面的示例中，注意我们如何在函数内把参数引用为 $1 和 $2：

```
CREATE FUNCTION add_em(int4, int4)
RETURNS int4
AS 'SELECT $1 + $2;'
LANGUAGE 'sql';

SELECT add_em(1, 2) AS answer;

     +-------+
     |answer |
     +-------+
     |3      |
     +-------+
```

38.1.3. 复合类型上的 SQL 函数

指定带复合类型（如 EMP）参数的函数时，我们不仅要指明想要哪个参数（如上面用 $1 和 $2），还要指明该参数的属性。例如，取函数 double_salary，它计算你的薪资翻倍后的值：

```
CREATE FUNCTION double_salary(EMP)
RETURNS int4
AS 'SELECT $1.salary * 2 AS salary;'
LANGUAGE 'sql';

SELECT name, double_salary(EMP) AS dream
FROM EMP
WHERE EMP.cubicle ~= '(2,1)'::point;


     +-----+-------+
```

```
 |name | dream |
 +-----+-------+
 |Sam  | 2400  |
 +-----+-------+
```

注意语法 $1.salary 的使用。在进入返回复合类型的函数这一主题之前，我们必须先介绍投影属性的函数记法。简单的解释是：attribute(class) 和 class.attribute 这两种记法通常可以互换使用：

```
--
-- this is the same as:
--  SELECT EMP.name AS youngster FROM EMP WHERE EMP.age < 30
--
SELECT name(EMP) AS youngster
    FROM EMP
    WHERE age(EMP) < 30;

 +----------+
 |youngster |
 +----------+
 |Sam       |
 +----------+
```

然而如下文所见，情况并不总是如此。当我们想使用返回单个实例的函数时，
这种函数记法很重要。我们通过在函数内逐属性地组装整个实例来实现。这是一个返回单个 EMP 实例的函数示例：

```
CREATE FUNCTION new_emp()
RETURNS EMP
AS ' SELECT \'None\'::text AS name,
         1000 AS salary,
         25 AS age,
         \'(2,2)\'::point AS cubicle'
LANGUAGE 'sql';
```

在本例中，我们为每个属性都指定了常量值，但这些常量可以换成任何计算或表达式。
定义这样的函数可能有些棘手。较重要的一些注意事项如下：

- 查询中的目标列表顺序必须与定义该复合类型的 CREATE TABLE 语句（或执行 .* 查询时）中属性出现的顺序完全相同。
- 你必须（使用 ::）非常小心地对表达式进行类型转换，否则会看到如下错误：

  ```
  NOTICE::function declared to return type EMP does not retrieve (EMP.
  *)
  ```

- 调用返回实例的函数时，我们无法检索整个实例。我们必须要么从实例中投影出一个属性，要么把整个实例传递给另一个函数。

  ```
  SELECT name(new_emp()) AS nobody;

          +-------+
  ```

```
|nobody |
+-------+
|None   |
+-------+
```

- 一般而言，我们必须使用函数语法投影函数返回值的属性，
  原因就是解析器还不理解另一种（点）语法与函数调用组合使用做投影。

  ```
  SELECT new_emp().name AS nobody;
  NOTICE:parser: syntax error at or near "."
  ```

SQL 查询语言中任何命令的集合都可以打包到一起并定义为函数。这些命令既可以是更新（即 INSERT、UPDATE 和 DELETE），也可以是 SELECT 查询。但最后一条命令必须是一个 SELECT，它返回函数返回类型所指定的内容。

```
CREATE FUNCTION clean_EMP ()
RETURNS int4
AS ' DELETE FROM EMP
 WHERE EMP.salary <= 0;
 SELECT 1 AS ignore_this;'
LANGUAGE 'sql';

SELECT clean_EMP();

     +--+
     |x |
     +--+
     |1 |
     +--+
```

38.2. 过程语言函数

过程语言并不内建于 Postgres。它们由可装载模块提供。关于各语言的语法以及 AS 子句如何被 PL 处理器解释的细节，请参阅相应 PL 的文档。

标准的 Postgres 发行版中有两种过程语言可用（PLTCL 和 PLSQL），其他语言可以由用户定义。更多信息参阅过程语言。

38.3. 内部函数

内部函数是用 C 编写并已静态链接到 Postgres 后端进程的函数。AS 子句给出该函数的 C 语言名称，它不必与为 SQL 使用而声明的名称相同。（出于向后兼容的原因，空的 AS 字符串也被接受，表示 C 语言函数名与 SQL 名称相同。）通常，后端中出现的所有内部函数都在数据库初始化期间声明为 SQL 函数，但用户可以用 CREATE FUNCTION 为内部函数创建额外的别名。

38.4. 编译型（C）语言函数

用 C 编写的函数可以编译成动态可装载对象，用于实现用户自定义的 SQL 函数。用户自定义函数第一次在后端内被调用时，动态装载器把该函数的目标代码装载到内存中，

并把该函数链接到正在运行的 Postgres 可执行文件。CREATE FUNCTION 的 SQL 语法以两种方式之一把 SQL 函数链接到 C 源函数。如果 SQL 函数与 C 源函数同名，则使用该语句的第一种形式。AS 子句中的字符串参数是包含动态可装载编译对象的文件的完整路径名。如果 C 函数的名称与期望的 SQL 函数名不同，则使用第二种形式。此时 AS 子句接受两个字符串参数，第一个是动态可装载对象文件的完整路径名，第二个是动态装载器应搜索的链接符号。这个链接符号就是 C 源代码中的函数名。

> **注意**
>
> 动态装载的用户函数在第一次使用后会保留在内存中，
> 以后对该函数的调用只会产生符号表查找这一很小的开销。

指定对象文件的字符串（AS 子句中的字符串）应是该函数目标代码文件的完整路径，用引号括起。如果 AS 子句中使用了链接符号，该链接符号也应用单引号括起，并且应与 C 源代码中函数的名称完全相同。在 Unix 系统上，命令 nm 会打印动态可装载对象中的所有链接符号。（Postgres 不会自动编译函数；它必须在被 CREATE FUNCTION 命令使用之前编译好。更多信息见下文。）

38.4.1. 基本类型上的 C 语言函数

下表给出了将要装载到 Postgres 中的 C 函数的参数所需的 C 类型。“定义于”列给出（.../src/backend/ 目录中）实际定义等价 C 类型的头文件。但如果你包含 utils/builtins.h，这些文件会被自动包含。

表 38.1. 内置 Postgres 类型的等价 C 类型

内置类型	C 类型	定义于
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
cid	CID	include/postgres.h
datetime	(DateTime *)	include/c.h 或 include/postgres.h
int2	int2	include/postgres.h
int2vector	(int2vector *)	include/postgres.h
int4	int4	include/postgres.h
float4	float32 或 (float4 *)	include/c.h 或 include/postgres.h
float8	float64 或 (float8 *)	include/c.h 或 include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h
oid	oid	include/postgres.h
oidvector	(oidvector *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h

内置类型	C 类型	定义于
regproc	regproc 或 REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
timespan	(TimeSpan *)	include/c.h 或 include/postgres.h
tinterval	TimeInterval	utils/nabstime.h
uint2	uint16	include/c.h
uint4	uint32	include/c.h
xid	(XID *)	include/postgres.h

在内部，Postgres 把基本类型视为一个“内存块”。你针对某个类型定义的用户自定义函数转而定义了 Postgres 能对它进行操作的方式。也就是说，Postgres 只负责在磁盘上存储和检索数据，而使用你的用户自定义函数来输入、处理和输出数据。
基本类型可以有以下三种内部格式之一：

- 传值，定长
- 传引用，定长
- 传引用，变长

传值类型的长度只能是 1、2 或 4 字节（即使你的计算机支持其他大小的传值类型）。Postgres 本身只按传值方式传递整数类型。你应当小心地定义类型，使其在所有体系结构上具有相同的大小（以字节计）。例如，`long` 类型是危险的，因为它在某些机器上是 4 字节而在另一些机器上是 8 字节；而 `int` 类型在大多数 Unix 机器上是 4 字节（但在大多数个人电脑上不是）。Unix 机器上 `int4` 类型的一个合理实现可以是：

```
/* 4-byte integer, passed by value */
typedef int int4;
```

另一方面，任何大小的定长类型都可以按传引用方式传递。例如，下面是一个 Postgres 类型的示例实现：

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

在 Postgres 函数中传入和传出这类类型时只能使用指向它们的指针。最后，所有变长类型也必须以传引用方式传递。所有变长类型都必须以一个恰好 4 字节的长度字段开始，而该类型中要存储的所有数据都位于紧随该长度字段之后的内存中。
长度字段是结构的总长度（即它包含长度字段本身的大小）。我们可以这样定义 text 类型：

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

显然，这里所示的 data 字段不足以容纳所有可能的字符串；在 C 中无法声明这样的结构。操作变长类型时，我们必须小心分配正确数量的内存并初始化长度字段。例如，如果我们想在 text 结构中存储 40 字节，可以使用如下代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memmove(destination->data, buffer, 40);
...
```

既然我们已经讲完了基本类型的所有可能结构，我们可以展示一些真实函数的示例。设 `funcs.c` 内容如下：

```
#include <string.h>
#include "postgres.h"

/* By Value */

int
add_one(int arg)
{
    return(arg + 1);
}

/* By Reference, Fixed Length */

Point *
makepoint(Point *pointx, Point *pointy )
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* By Reference, Variable Length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    memset(new_t, 0, VARSIZE(t));
    VARSIZE(new_t) = VARSIZE(t);
    /*
```

```
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);        /* how many bytes */
    return(new_t);
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    memset((void *) new_text, 0, new_text_size);
    VARSIZE(new_text) = new_text_size;
    strncpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    strncat(VARDATA(new_text), VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return (new_text);
}
```

在 OSF/1 上我们会输入：

```
CREATE FUNCTION add_one(int4) RETURNS int4
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';
```

在其他系统上，我们可能必须让文件名以 .sl 结尾（表示它是共享库）。

38.4.2. 复合类型上的 C 语言函数

复合类型没有像 C 结构那样的固定布局。复合类型的实例可能包含空字段。此外，属于某个继承层次的复合类型可能具有与同一继承层次其他成员不同的字段。因此，Postgres 提供了从 C 访问复合类型字段的过程式接口。当 Postgres 处理一组实例时，每个实例都会作为一个 TUPLE 类型的不透明结构传入你的函数。假设我们想编写一个函数来回答查询

```
* SELECT name, c_overpaid(EMP, 1500) AS overpaid
  FROM EMP
  WHERE name = 'Bill' or name = 'Sam';
```

在上面的查询中，我们可以把 c_overpaid 定义为：

```
#include "postgres.h"
```

```
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current instance of EMP */
           int4 limit)
{
    bool isnull = false;
    int4 salary;
    salary = (int4) GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        return (false);
    return(salary > limit);
}
```

GetAttributeByName 是 Postgres 的系统函数，返回当前实例中的属性。它有三个参数：传给函数的 TUPLE 类型参数、所需属性的名称，以及一个描述该属性是否为空的返回参数。GetAttributeByName 会正确地对齐数据，因此你可以把它的返回值转换为期望的类型。例如，如果你有一个 name 类型的属性 name，GetAttributeByName 调用将形如：

```
char *str;
...
str = (char *) GetAttributeByName(t, "name", &isnull)
```

下面的查询让 Postgres 知道 c_overpaid 函数：

```
CREATE FUNCTION c_overpaid(EMP, int4)
RETURNS bool
AS 'PGROOT/tutorial/obj/funcs.so'
LANGUAGE 'c';
```

虽然有办法在 C 函数内构造新实例或修改现有实例，但这些方法过于复杂，本手册不予讨论。

38.4.3. 编写代码

现在我们转向编写编程语言函数这一更困难的任务。
请注意：本手册的这一部分不会使你成为程序员。在尝试为 Postgres 编写 C 函数之前，你必须对 C（包括指针和 malloc 内存管理器的使用）有良好的理解。虽然也许可以把用 C 以外语言编写的函数装载到 Postgres，但这通常很困难（即使在可能的时候），因为其他语言（如 FORTRAN 和 Pascal）往往不遵循与 C 相同的调用约定。也就是说，其他语言在函数之间传递参数和返回值的方式不同。因此，我们将假定你的编程语言函数是用 C 编写的。

带基本类型参数的 C 函数可以以直截了当的方式编写。如果把 *PGROOT*/src/backend/utils/builtins.h 作为头文件包含，内建 Postgres 类型的 C 等价物就可以在 C 文件中使用。这可以通过

```
#include <utils/builtins.h>
```

放在 C 源文件的顶部。

构建 C 函数的基本规则如下：

- Postgres 的大多数头（include）文件应当已安装在 *PGROOT*/include（见图 2）。你应当总是在 cc 命令行中包含

  ```
  -I$PGROOT/include
  ```

  。有时你可能发现需要服务器源码本身中的头文件（即需要一个我们疏忽没有安装到 include 中的文件）。那些情况下，你可能需要添加一个或多个

  ```
  -I$PGROOT/src/backend
  -I$PGROOT/src/backend/include
  -I$PGROOT/src/backend/port/<PORTNAME>
  -I$PGROOT/src/backend/obj
  ```

  （其中 <PORTNAME> 是移植的名称，例如 alpha 或 sparc）。

- 分配内存时，使用 Postgres 的例程 palloc 和 pfree，而不是相应的 C 库例程 malloc 和 free。palloc 分配的内存会在每个事务结束时自动释放，从而防止内存泄漏。

- 总是使用 memset 或 bzero 把结构体的字节清零。有几个例程（如哈希访问方法、哈希连接和排序算法）会计算结构体中原始位的函数。即使你初始化了结构体的所有字段，结构体中仍可能存在若干包含垃圾值的对齐填充字节（结构体中的空洞）。

- Postgres 的大多数内部类型都在 postgres.h 中声明，因此始终也包含该文件是个好主意。包含 postgres.h 时还会顺带为你包含 elog.h 和 palloc.h。

- 编译和装载你的目标代码使其能被动态装载到 Postgres 总是需要特殊的标志。关于如何针对你的特定操作系统进行操作，参见链接动态装载的函数的详细说明。

38.5. 函数重载

可以用同一个名称定义多个函数，只要它们接受的参数不同即可。换句话说，函数名可以被重载。函数也可以与属性同名。当复合类型上的函数与该复合类型的属性存在歧义时，总是使用属性。

38.5.1. 名字空间冲突

从 Postgres v7.0 开始，SQL CREATE FUNCTION 命令的 AS 子句的另一种形式可以把 SQL 函数名与 C 源代码中的函数名解耦。这现在是实现函数重载的首选技术。

38.5.1.1. v7.0 之前

对用 C 编写的函数，CREATE FUNCTION 中声明的 SQL 名称必须与 C 代码中函数的实际名称完全相同（因此它必须是合法的 C 函数名）。

这一限制有一个微妙的含义：虽然大多数操作系统的动态装载例程都乐于让你装载任意数量包含冲突（同名）函数名的共享库，但它们实际上可能以各种有趣的方式把装载搞砸。例如，如果你定义的动态装载函数恰好与 Postgres 内置函数同名，DEC OSF/1 的动态装载器会使 Postgres 调用其自身的函数，而不是让 Postgres 调用你的函数。因此，如果你希望你的函数能在不同体系结构上使用，我们建议不要重载 C 函数名。

有一个巧妙的技巧可以绕过刚才描述的问题。由于重载 SQL 函数没有问题，你可以先定义一组名称不同的 C 函数，然后定义一组同名的 SQL 函数包装器，它们接受适当的参数类型并调用相匹配的 C 函数。

另一种解决方案是不用动态装载，而是把你的函数静态链接到后端并声明为 INTERNAL 函数。此时这些函数的 C 名称必须各不相同，但可以用相同的 SQL 名称声明（当然，只要它们的参数类型不同）。这种方式避免了 SQL 包装函数的开销，代价是准备定制的后端可执行文件要花费更多精力。（此选项只在 6.5 及之后的版本中可用，因为更早的版本要求内部函数在 SQL 中的名称与 C 代码中的相同。）

第 39 章 扩展 SQL：类型

如前所述，Postgres中有两类类型：基本类型（用编程语言定义）和组合类型（实例）。
本节中直到索引接口之前的示例可以在 `complex.sql`和`complex.c`中找到。组合类型的示例在`funcs.sql`中。

39.1. 用户定义的类型

39.1.1. 用户定义类型所需的函数

用户定义的类型必须始终具有输入和输出函数。
这些函数决定该类型在字符串中如何表示（供用户输入和输出给用户）以及在内存中如何组织。
输入函数接受一个以空字符结尾的字符串作为输入，并返回该类型的内部（内存中）表示。
输出函数接受该类型的内部表示，并返回一个以空字符结尾的字符串。
假设我们想定义一个表示复数的 complex 类型。很自然，我们会选择用下面的C结构在内存中表示复数：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

并选择形如 (x,y) 的字符串作为外部字符串表示。这些函数通常不难编写，输出函数尤其如此。不过，有几点需要记住：

- 在定义你的外部（字符串）表示时，
  请记住：你最终必须为该表示编写一个完整而健壮的解析器作为输入函数！

  ```
  Complex *
  complex_in(char *str)
  {
      double x, y;
      Complex *result;
      if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
          elog(NOTICE, "complex_in: error in parsing
          return NULL;
      }
      result = (Complex *)palloc(sizeof(Complex));
      result->x = x;
      result->y = y;
      return (result);
  }
  ```

  输出函数可以简单地写成：

  ```
  char *
  complex_out(Complex *complex)
  {
      char *result;
      if (complex == NULL)
  ```

```
        return(NULL);
    result = (char *) palloc(60);
    sprintf(result, "(%g,%g)", complex->x, complex->y);
    return(result);
}
```

- 你应当尽量让输入和输出函数互为逆函数。否则，
  在你需要把数据转储到文件中再读回来时（比如读到另一台计算机上某人的数据库中），
  就会出现严重的问题。当涉及浮点数时，这是一个特别常见的问题。

要定义complex类型，我们需要在创建类型之前先创建 complex_in 和 complex_out 这两个用户定义的函数：

```
CREATE FUNCTION complex_in(opaque)
    RETURNS complex
    AS 'PGROOT/tutorial/obj/complex.so'
    LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
    RETURNS opaque
    AS 'PGROOT/tutorial/obj/complex.so'
    LANGUAGE 'c';

CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

如前面所讨论的，Postgres完全支持基本类型的数组。此外，
Postgres也支持用户定义类型的数组。当你定义一个类型时，
Postgres会自动提供对该类型数组的支持。由于历史原因，
数组类型的名称与用户定义类型相同，只是在前面加上下划线字符 _。组合类型不需要在其上定义任何函数，因为系统已经知道它们内部是什么样子。

39.1.2. 大对象

到此为止讨论的类型都是"小"对象——也就是说，它们的大小小于 8KB。

> **注意**
>
> 1024 长字 == 8192 字节。实际上，类型必须显著小于 8192 字节，因为Postgres的元组和页开销也必须放进这个 8KB 的限制内。实际能放下的值取决于机器体系结构。

如果你需要一个更大的类型，比如用于文档检索系统或存储位图，你就需要使用
Postgres的大对象接口，或者需要重新编译 Postgres后端以使用大于 8k 字节的内部存储块。。

第 40 章 扩展 SQL：操作符

Postgres 支持左一元、右一元和二元操作符。操作符可以重载；也就是说，同一个操作符名可用于参数数量和类型不同的多个操作符。如果出现歧义情况而系统无法确定该用哪个操作符，它会返回一个错误。你可能需要对左操作数和/或右操作数进行类型转换，帮助系统理解你想用哪个操作符。

每个操作符都是对某个完成实际工作的底层函数调用的"语法糖"；因此你必须先创建底层函数，然后才能创建操作符。然而，操作符不仅仅是语法糖，因为它还携带额外信息，帮助查询规划器优化使用该操作符的查询。本章的大部分篇幅将用于解释这些额外信息。

下面是创建一个把两个复数相加的操作符的例子。我们假设已经创建过 complex 类型的定义。首先需要一个完成工作的函数，然后就可以定义操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS '$PWD/obj/complex.so'
    LANGUAGE 'c';

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

现在我们可以这样做：

```
SELECT (a + b) AS c FROM test_complex;

+-----------------+
|c                |
+-----------------+
|(5.2,6.05)       |
+-----------------+
|(133.42,144.95)  |
+-----------------+
```

我们在这里展示了如何创建二元操作符。要创建一元操作符，只需省略 leftarg（左一元）或 rightarg（右一元）之一即可。procedure 子句和参数子句是 CREATE OPERATOR 中仅有的必选项。例子中展示的 COMMUTATOR 子句是给查询优化器的一个可选提示。关于 COMMUTATOR 和其他优化器提示的更多细节见下文。

40.1. 操作符优化信息

作者

由 Tom Lane 撰写。

Postgres 的操作符定义可以包含若干可选子句，告诉系统关于操作符行为的有用信息。只要合适就应当提供这些子句，因为它们能给使用该操作符的查询的执行带来可观的加速。
但如果你提供了它们，就必须确保它们是对的！错误地使用优化子句可能导致后端崩溃、微妙错误的输出或其他糟糕后果。如果不确定，你随时可以省略某个优化子句；
唯一的后果是查询可能比所需的更慢。

未来的 Postgres 版本可能会添加更多优化子句。这里描述的是 6.5 版理解的所有子句。

40.1.1. COMMUTATOR

如果给出 COMMUTATOR 子句，它指定一个与正在定义的操作符互为交换子的操作符。若对所有可能的输入值 x、y，都有 (x A y) 等于 (y B x)，则称操作符 A 是操作符 B 的交换子。注意，B 也同样是 A 的交换子。例如，对某种特定数据类型而言，'<' 和 '>' 操作符通常互为交换子，而操作符 '+' 通常与其自身可交换。但操作符 '-' 通常并不与任何操作符可交换。

被交换操作符的左参数类型与其交换子的右参数类型相同，反之亦然。因此，
只要给出交换子操作符的名称，Postgres 就足以查找到交换子，这也正是 COMMUTATOR 子句需要提供的全部内容。

定义一个与自身可交换的操作符时，直接定义即可。但定义一对可交换操作符时，
情况就稍微复杂一些：第一个要定义的操作符如何引用另一个尚未定义的操作符呢？这个问题有两种解决办法：

- 一种办法是在你定义的第一个操作符中省略 COMMUTATOR 子句，然后在第二个操作符的定义中给出它。由于 Postgres 知道可交换操作符成对出现，当它看到第二个定义时，会自动回过头来补填第一个定义中缺失的 COMMUTATOR 子句。

- 另一种更直接的办法是在两个定义中都包含 COMMUTATOR 子句。当 Postgres 处理第一个定义并发现 COMMUTATOR 引用了一个不存在的操作符时，系统会在系统的 pg_operator 表中为该操作符创建一个占位条目。这个占位条目只在操作符名称、左右参数类型和结果类型上拥有有效数据，因为那是 Postgres 此时所能推断出的全部。第一个操作符的目录条目会链接到这个占位条目。之后当你定义第二个操作符时，
  系统会用第二个定义中的额外信息更新该占位条目。
  如果你在占位条目被填充之前就尝试使用该占位操作符，只会得到一条错误消息。
  （注意：这一过程在 6.5 之前的 Postgres 版本中不能可靠工作，但现在它是推荐的做法。）

40.1.2. NEGATOR

如果给出 NEGATOR 子句，它指定一个与正在定义的操作符互为求反器的操作符。若操作符 A 和 B 都返回布尔结果，并且对所有可能的输入 x、y，都有 (x A y) 等于 NOT (x B y)，则称 A 是 B 的求反器。注意，B 也同样是 A 的求反器。例如，对大多数数据类型而言，'<' 和 '>=' 构成一对求反器。一个操作符永远不可能合法地成为其自身的求反器。

与交换子不同，一对一元操作符完全可能合法地被标记为彼此的求反器；这意味着对所有 x，都有 (A x) 等于 NOT (B x)，或者对右一元操作符有等价的关系。

操作符的求反器必须与该操作符本身具有相同的左和/或右参数类型，因此与 COMMUTATOR 一样，在 NEGATOR 子句中只需给出操作符名。

提供求反器对查询优化器很有帮助，因为它允许把形如 NOT (x = y) 的表达式化简为 x <> y。这种情况比你想象的更常见，因为其他整理过程可能会插入 NOT 操作。

可以使用上面解释的定义交换子对的相同方法，来定义成对的求反器操作符。

40.1.3. RESTRICT

如果给出 RESTRICT 子句，它指定该操作符的限制选择率估算函数。（注意，这里是函数名，而不是操作符名。）RESTRICT 子句只对返回 boolean 的二元操作符有意义。限制选择率估算器的作用，是猜测一张表中有多少比例的行会满足如下形式的 WHERE 子句条件：

```
field OP constant
```

（即针对当前操作符和某个特定常量值）。这会帮助优化器大致了解这种形式的 WHERE 子句会消掉多少行。（你可能在想：如果常量在左边会怎样？嗯，这正是 COMMUTATOR 的用途之一……）

编写新的限制选择率估算函数远远超出了本章的范围，但幸运的是，对你的许多操作符，
你通常可以直接使用系统的标准估算器之一。标准的限制估算器有：

```
eqsel  for =
neqsel  for <>
scalarltsel for < or <=
scalargtsel for > or >=
```

这些类别看起来可能有点奇怪，但仔细想想是有道理的。'=' 通常只会接受表中一小部分行；'<>' 通常只会排除一小部分行。'<' 接受的比例取决于给定常量落在该表列取值范围中的位置（而这恰好是 VACUUM ANALYZE 收集并提供给选择率估算器的信息）。对同一个比较常量，'<=' 接受的比例略大于 '<'，但两者足够接近，不值得区分，尤其是因为我们反正也不太可能比粗略猜测做得更好。类似的说明也适用于 '>' 和 '>='。

对于选择率很高或很低的操作符，即使它们并非真正的相等或不相等，你也常常可以凑合使用 eqsel 或 neqsel。例如，几何上的近似相等操作符就使用了 eqsel，其假设是它们通常只会匹配表中一小部分条目。

对于那些能够以某种合理方式转换为数值标量、从而可进行范围比较的数据类型，你可以使用 scalarltsel 和 scalargtsel。如果可能的话，请把该数据类型加入 src/backend/utils/adt/selfuncs.c 中函数 convert_to_scalar() 所理解的范围。（最终，这个函数应被通过 pg_type 表某一列标识的、按数据类型划分的函数所取代；但这件事目前还没有发生。）如果不这样做，系统仍然可以工作，但优化器的估算效果就不会像本可达到的那样好。

src/backend/utils/adt/geo_selfuncs.c 中还有为几何操作符设计的额外选择率函数：areasel、positionsel 和 contsel。在写作本文时它们只是存根，但你可能还是想用它们（甚至更好的是，改进它们）。

40.1.4. JOIN

如果给出 JOIN 子句，它指定该操作符的连接选择率估算函数。（注意，这里是函数名，而不是操作符名。）JOIN 子句只对返回 boolean 的二元操作符有意义。连接选择率估算器的作用，是猜测一对表中有多少比例的行会满足如下形式的 WHERE 子句条件：

```
table1.field1 OP table2.field2
```

（即针对当前操作符）。与 RESTRICT 子句一样，这也能极大地帮助优化器判断若干可能的连接顺序中哪一个可能耗时最少。

与前面一样，本章不会尝试解释如何编写连接选择率估算函数，
而只是建议你在适用时使用标准估算器之一：

```
eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons
```

40.1.5. HASHES

如果给出 HASHES 子句，它告诉系统可以对此操作符上的连接使用哈希连接方法。HASHES 只对返回 boolean 的二元操作符有意义，而且实践中该操作符最好是某种数据类型上的相等操作。

哈希连接背后的假设是：只有当左值和右值被哈希到同一个哈希码时，连接操作符才有可能返回 TRUE。如果两个值被放进不同的哈希桶，连接过程就根本不会去比较它们，
这实际上隐含假定连接操作符的结果必定为　　FALSE。因此，对于那些并不表示相等关系的操作符，指定 HASHES 永远没有意义。

事实上，仅逻辑上的相等也不够；该操作符最好表示纯粹的按位相等，
因为哈希函数是按值的内存表示计算的，而不管这些位是什么含义。例如，
时间间隔的相等就不是按位相等；时间间隔相等操作符认为两个时间间隔只要时长相同就相等，
无论其端点是否一致。这意味着在时间间隔字段上使用 "=" 的连接，如果实现为哈希连接，得到的结果会与其他实现方式不同，因为本应匹配的很大一部分值对会被哈希到不同的值，
永远不会被哈希连接比较。而如果优化器选择使用另一种连接，
相等操作符判定相等的所有值对都会被找到。我们不想要这种不一致，
因此我们没有把时间间隔的相等操作标记为可哈希。

还有一些与机器相关的情形可能让哈希连接出错。例如，
如果你的数据类型是一个可能含有无关填充位的结构，把它的相等操作符标记为 HASHES 是不安全的。（除非也许你把其他操作符写成保证未用的位始终为零。）另一个例子是 FLOAT 数据类型用于哈希连接是不安全的。在符合 IEEE 浮点标准的机器上，负零和正零是不同的值（位模式不同），但按定义它们比较相等。因此，如果把浮点相等标记为 HASHES，负零和正零可能不会被哈希连接配对，而任何其他连接过程都会把它们配上。

归根结底：你可能只应对那些（可以）用 memcmp() 实现的相等操作符使用 HASHES。

40.1.6. SORT1 与 SORT2

如果给出 SORT 子句，它们告诉系统可以对此操作符上的连接使用归并连接方法。指定其一就必须同时指定另一个。当前操作符必须是某对数据类型上的相等操作，而 SORT1 和 SORT2 子句分别命名左边和右边数据类型的排序操作符（'<' 操作符）。

归并连接基于这样的思想：把左右两边的表排好序，然后并行扫描它们。因此，
两个数据类型都必须能够全序化，
而且连接操作符必须只在处于排序次序中"相同位置"的值对上才可能成功。
实践中这意味着连接操作符的行为必须像相等一样。
但与哈希连接不同——哈希连接要求左右数据类型最好相同（或至少按位等价）——只要两个不同的数据类型在逻辑上兼容，就可以对它们做归并连接。例如，int2 对 int4 的相等操作符就是可归并连接的。我们只需要能把两个数据类型排成逻辑兼容次序的排序操作符。

指定归并排序操作符时，当前操作符和两个被引用的操作符都必须返回 boolean；SORT1 操作符的两个输入数据类型都必须等于当前操作符的左参数类型，SORT2 操作符的两个输入数据类型都必须等于当前操作符的右参数类型。（与 COMMUTATOR 和 NEGATOR 一样，这意味着只凭操作

符名就足以指定操作符，而且如果你碰巧先定义了相等操作符、后定义其他操作符，
系统也能够创建占位操作符条目。）

实践中你只应对 '=' 操作符写 SORT 子句，而且两个被引用的操作符应当总是命名为 '<'。试图对名称不是这些的操作符使用归并连接将导致不可救药的混乱，原因我们稍后就会看到。

对于你标记为可归并连接的操作符，还有一些附加限制。这些限制目前不会被 CREATE OPERATOR 检查，但如果其中任何一条不成立，归并连接在运行时可能失败：

- 可归并连接的相等操作符必须有交换子（两个数据类型相同时就是它自身，
  不同时则是一个相关的相等操作符）。
- 必须存在与可归并连接操作符本身具有相同左右输入数据类型的 '<' 和 '>' 排序操作符。这些操作符必须命名为 '<' 和 '>'；这件事你没有选择的余地，因为没有显式指定它们的条款。注意，如果左右数据类型不同，这两个操作符都不是任何一个 SORT 操作符。但它们最好与 SORT 操作符以兼容的方式对数据值排序，否则归并连接将无法工作。

第 41 章 扩展 SQL：聚合

Postgres 中的聚合函数被表示为状态值和状态转换函数。也就是说，聚合可以按照每当处理一个输入项就被修改的状态来定义。要定义一个新的聚合函数，需要为状态值选择一种数据类型、为状态选择一个初始值，以及一个状态转换函数。状态转换函数只是一个普通函数，也可以在聚合上下文之外使用。

实际上，为了更容易用现有函数构造有用的聚合，聚合可以有一个或两个独立的状态值、一个或两个更新这些状态值的转换函数，以及一个从最终状态值计算实际聚合结果的最终函数。

因此最多可能涉及四种数据类型：输入数据项的类型、聚合结果的类型，以及两个状态值的类型。聚合的用户只能看到输入和结果数据类型。

有些状态转换函数需要查看每个相继的输入来计算下一个状态值，而另一些则忽略具体的输入值，只是更新自己的内部状态。（第二类最有用的例子是对输入项数量的运行计数。）Postgres 的聚合机制把聚合的 sfunc1 定义为一个同时接收旧状态值和当前输入值的函数，而 sfunc2 则是一个只接收旧状态值的函数。

如果我们定义一个只使用 sfunc1 的聚合，得到的就是一个对每个实例的属性值计算运行函数的聚合。"Sum" 就是这类聚合的一个例子。"Sum" 从零开始，总是把当前实例的值加到它的运行总和上。例如，如果我们想让 Sum 聚合作用于复数的数据类型，我们只需要该数据类型的加法函数。聚合定义是：

```
CREATE AGGREGATE complex_sum (
    sfunc1 = complex_add,
    basetype = complex,
    stype1 = complex,
    initcond1 = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;

         +------------+
         |complex_sum |
         +------------+
         |(34,53.9)   |
         +------------+
```

（实践中我们只会把该聚合命名为 "sum"，让 Postgres 自行判断对复数列应用哪种 sum。）

如果我们只定义 sfunc2，指定的是一个对每个实例的属性值无关的运行函数计算的聚合。"Count" 是这类聚合最常见的例子。"Count" 从零开始，对每个实例向其运行总数加一，忽略实例的值。这里我们使用内置的 int4inc 例程为我们完成这项工作。该例程把输入加一。

```
CREATE AGGREGATE my_count (
    sfunc2 = int4inc, -- add one
    basetype = int4,
    stype2 = int4,
    initcond2 = '0'
);
```

"Average"（平均值）是一个既需要计算运行总和的函数又需要计算运行计数的函数的聚合例子。处理完所有实例后，聚合的最终答案是运行总和除以运行计数。我们使用前面用过的 int4pl 和 int4inc 例程，以及 Postgres 的整数除法例程 int4div 来计算总和除以计数。

```
CREATE AGGREGATE my_average (
    sfunc1 = int4pl,     --  sum
    basetype = int4,
    stype1 = int4,
    sfunc2 = int4inc,    -- count
    stype2 = int4,
    finalfunc = int4div, -- division
    initcond1 = '0',
    initcond2 = '0'
);

SELECT my_average(salary) as emp_average FROM EMP;

         +------------+
         |emp_average |
         +------------+
         |1640        |
         +------------+
```

更多细节参见 PostgreSQL 用户指南中的 `CREATE AGGREGATE`。

第 42 章 Postgres 规则系统

产生式规则系统在概念上很简单，但在实际使用时会涉及许多微妙之处。其中一些要点，以及Postgres规则系统的理论基础，可以在 [Stonebraker et al, ACM, 1990] 中找到。

有些其他数据库系统定义了主动型数据库规则。它们通常是存储过程和触发器，在Postgres中以函数和触发器的形式实现。

查询重写规则系统（下文简称“规则系统”）与存储过程和触发器完全不同。它会修改查询，使其将规则纳入考虑，然后把修改后的查询交给查询优化器执行。它非常强大，可用于查询语言过程、视图和版本等许多用途。这一规则系统的能力在 [Ong and Goh, 1990] 以及 [Stonebraker et al, ACM, 1990] 中有讨论。

42.1. 什么是查询树?

要理解规则系统如何工作，必须先知道它在何时被调用，以及它的输入和输出是什么。

规则系统位于查询解析器和优化器之间。它接收解析器的输出，即一棵查询树，以及来自 `pg_rewrite` 目录的重写规则；这些规则本身也是带有一些附加信息的查询树。它会生成零棵或多棵查询树作为结果。因此，它的输入和输出始终都是解析器本身也可能产生的东西，所以它看到的任何内容基本上都可以表示为一个SQL语句。

那么什么是查询树？它是SQL语句的一种内部表示，其中构成语句的各个部分被分别存储。如果以调试级别 4 启动 Postgres 后端，并在交互式后端界面中输入查询，就可以看到这些查询树。`pg_rewrite` 系统目录中的规则动作同样以查询树的形式存储。它们的格式与调试输出不同，但包含的却是完全相同的信息。

阅读查询树需要一些经验，我刚开始在规则系统上工作时也经历了一段艰难的时光。我记得我曾站在咖啡机前，看到目标列表里有一个杯子，rangetable 里有水和咖啡粉，而所有的按钮出现在一个条件表达式中。由于查询树的SQL表示已足以理解规则系统，本文不会讲解如何阅读它们。学习阅读查询树也许会有帮助，而且在后续的描述中需要用到这些命名约定。

42.1.1. 查询树的组成部分

在阅读本文中查询树的 SQL 表示时，需要能够辨认语句在查询树结构中被拆分成了哪些部分。查询树的组成部分是

命令类型

> 这是一个简单的值，说明了是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了解析树。

范围表

> 范围表是查询中使用的关系列表。在 SELECT 语句中，它们就是 FROM 关键字后面给出的关系。
>
> 每个范围表项标识一个表或视图，并说明在查询的其他部分以哪个名称来称呼它。在查询树中，范围表项是按编号而不是按名称引用的，因此即使像 SQL 语句中那样出现重名，这里也无关紧要。这种情况可能出现在规则的范围表被合并之后。本文中的示例不会涉及这种情形。

结果关系

这是范围表中的一个索引，用来标识查询结果应写入哪个关系。

SELECT 查询通常没有结果关系。SELECT INTO 这一特殊情况与 CREATE TABLE、INSERT ... SELECT 序列几乎相同，这里不再单独讨论。

在 INSERT、UPDATE 和 DELETE 查询中，结果关系就是要让修改生效的表（或视图！）。

目标列表

目标列表是定义查询结果的表达式列表。对于 SELECT，这些表达式构成查询的最终输出。它们对应于关键字 SELECT 和 FROM 之间的表达式。（* 只是某个关系全部属性名的缩写。）

DELETE 查询不需要目标列表，因为它们不产生任何结果。实际上优化器会向空目标列表加入一个特殊项，但这发生在规则系统之后，将在后面讨论。对规则系统而言，
目标列表是空的。

在 INSERT 查询中，目标列表描述的是将要进入结果关系的新行。结果关系中缺失的列将由优化器补上常量空值表达式。它由 VALUES 子句中的表达式，或 INSERT ... SELECT 里 SELECT 子句中的表达式组成。

在 UPDATE 查询中，目标列表描述要替换旧行的新行。这里，优化器会通过插入把旧行的值放到新行中的表达式来补上缺失列，并且也会像 DELETE 那样加上那个特殊项。它由查询中 SET attribute = expression 部分的表达式组成。

目标列表中的每一项都包含一个表达式，它可以是常量值、
指向范围表中某个关系属性的变量、一个参数，或者由函数调用、常量、变量、
操作符等构成的表达式树。

条件

查询的条件是一个表达式，与目标列表项中的表达式很相似。该表达式的结果值是布尔值，用来说明是否应对最终结果行执行相应操作（INSERT、UPDATE、DELETE 或 SELECT）。它就是 SQL 语句的 WHERE 子句。

其他

查询树的其他部分，如 ORDER BY 子句，这里不作关注。规则系统在应用规则时会替换其中的项，但这与规则系统的基本原理关系不大。GROUP BY 在出现在视图定义中时是一种特殊情况，仍有待文档说明。

42.2. 视图和规则系统

42.2.1. Postgres 中视图的实现

在 Postgres 中，视图通过规则系统实现。实际上，以下命令：

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面这两条命令基本没有区别：

```
CREATE TABLE myview (same attribute list as for mytab);
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

因为这正是 CREATE VIEW 命令在内部所做的事情。这会带来一些副作用。其中之一是，在 Postgres 系统目录中，视图的信息与表的信息完全相同。因此，对查询解析器而言，表和视图完全没有区别。它们是同一种东西——关系。这是目前最重要的一点。

42.2.2. SELECT 规则如何工作

ON SELECT 规则会在所有查询上作为最后一步应用，即使给出的命令是 INSERT、UPDATE 或 DELETE 也一样。它们与其他规则在语义上不同，因为它们是就地修改解析树，而不是创建新的解析树。因此我们先讨论 SELECT 规则。

目前，只能有一个动作，而且它必须是一个带有 INSTEAD 的 SELECT 动作。之所以有这个限制，是为了让规则足够安全，从而能够向普通用户开放；它也把 ON SELECT 规则限制为真正的视图规则。

本文的示例是两个进行一些计算的连接视图，以及另外一些依次使用它们的视图。
最初两个视图中的一个会在后面通过为 INSERT、UPDATE 和 DELETE 操作添加规则来定制，从而最终得到一个在行为上像真正的表、但又带有某些特殊功能的视图。作为入门示例，
这并不算简单，因此会让理解变得更难一些。但与其使用许多可能令人混淆的不同示例，
不如用一个示例逐步覆盖这里讨论的全部要点。

用于在这些示例上操作的数据库名为 al_bundy。你很快就会明白为什么取这个名字。它还需要安装过程语言 PL/pgSQL，因为我们需要一个返回两个整数值中较小者的 min() 函数。我们这样创建它：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS 'BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;' LANGUAGE 'plpgsql';
```

在前两节对规则系统的描述（descripitons）中，我们需要用到如下真实表：

```
CREATE TABLE shoe_data (
    shoename   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);

CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

我想我们大多数人都穿鞋，都能看出这确实是很有用的数据。当然，
世上有一些不需要鞋带的鞋，但这并不能让 Al 的日子好过一点，所以我们忽略这一点。

这些视图的创建方式如下：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建 `shoelace` 视图（这是我们这里最简单的一个例子）的 CREATE VIEW 命令会创建一个关系 shoelace，并在 `pg_rewrite` 中创建一项，说明只要查询的 rangetable 中引用了关系 shoelace，就必须应用一条重写规则。该规则没有规则条件（这在非 SELECT 规则中讨论，因为 SELECT 规则目前不能有规则条件），并且它是 INSTEAD。注意，规则条件与查询条件不是一回事！这里我们的规则动作带有一个查询条件。

规则动作本身是一棵查询树，它是视图创建命令中 SELECT 语句的一个精确副本。

> **注意**
>
> 你在 `pg_rewrite` 项中看到的、用于 NEW 和 OLD 的两个额外范围表项（在打印出来的查询树中，它们出于历史原因被命名为 *NEW* 和 *CURRENT*），与 SELECT 规则无关。

现在我们填充 `unit`、`shoe_data` 和 `shoelace_data`，然后 Al 在他的一生中第一次输入 SELECT：

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

这是 Al 在这些视图上能执行的最简单的 SELECT，因此我们借此机会说明视图规则的基础。'SELECT * FROM shoelace' 由解析器解释后，会生成如下 parsetree：

```
    SELECT shoelace.sl_name, shoelace.sl_avail,
           shoelace.sl_color, shoelace.sl_len,
           shoelace.sl_unit, shoelace.sl_len_cm
      FROM shoelace shoelace;
```

然后它会被交给规则系统。规则系统遍历 rangetable，检查 `pg_rewrite` 中是否有关系对应的规则。在处理 `shoelace` 的 rangetable 项时（到目前为止只有这一个），它会找到带有如下 parsetree 的规则 '_RETshoelace'：

```
SELECT s.sl_name, s.sl_avail,
           s.sl_color, s.sl_len, s.sl_unit,
           float8mul(s.sl_len, u.un_fact) AS sl_len_cm
      FROM shoelace *OLD*, shoelace *NEW*,
           shoelace_data s, unit u
     WHERE bpchareq(s.sl_unit, u.un_name);
```

注意，解析器把计算和限定条件都改写成了对相应函数的调用。
但这实际上并没有改变任何东西。重写的第一步是合并两个 rangetable。得到的 parsetree 如下：

```
    SELECT shoelace.sl_name, shoelace.sl_avail,
           shoelace.sl_color, shoelace.sl_len,
           shoelace.sl_unit, shoelace.sl_len_cm
      FROM shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data s,
           unit u;
```

第 2 步把规则动作中的限定条件加到该 parsetree 上，得到：

```
    SELECT shoelace.sl_name, shoelace.sl_avail,
           shoelace.sl_color, shoelace.sl_len,
           shoelace.sl_unit, shoelace.sl_len_cm
      FROM shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data s,
           unit u
     WHERE bpchareq(s.sl_unit, u.un_name);
```

第 3 步把 parsetree 中所有引用当前正在处理的 rangetable 项（即 `shoelace` 的那一项）的变量，替换为规则动作 targetlist 中的相应表达式。最终得到的查询是：

```
    SELECT s.sl_name, s.sl_avail,
           s.sl_color, s.sl_len,
           s.sl_unit, float8mul(s.sl_len, u.un_fact) AS sl_len_cm
      FROM shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data s,
           unit u
     WHERE bpchareq(s.sl_unit, u.un_name);
```

把它转换回人类用户会输入的真实 SQL 语句，读作：

```
    SELECT s.sl_name, s.sl_avail,
           s.sl_color, s.sl_len,
           s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;
```

这就是应用的第一条规则。在此过程中，rangetable 已经变大了。于是规则系统继续检查各个 rangetable 项。下一个是第 2 号（shoelace *OLD*）。关系 `shoelace` 有一条规则，但这个 rangetable 项没有被 parsetree 中的任何变量引用，因此被忽略。由于其余的 rangetable 项要么在 `pg_rewrite` 中没有规则、要么未被引用，检查到达 rangetable 末尾。重写至此完成，上面的结果就是交给优化器的最终结果。优化器会忽略那些未被 parsetree 中的变量引用的额外 rangetable 项，优化器/优化器产生的计划，将与 Al 直接输入上面的 SELECT 查询（而不是对视图的选择）完全相同。

现在我们让 Al 面对这样一个问题：布鲁斯兄弟（Blues Brothers）来到他的店里，想买一些新鞋；而且正如布鲁斯兄弟的本色那样，他们要穿一模一样的鞋。他们还想立刻就穿上，
所以还需要鞋带。

Al 需要知道，商店里目前哪些鞋子有颜色和尺码都匹配的鞋带，并且完全匹配的总双数大于等于二。我们教会（theach）他怎么做，于是他询问他的数据库：

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al 是一位鞋类行家，所以他知道只有 sh1 型号的鞋才合适（鞋带 sl7 是棕色的，而需要棕色鞋带的鞋是布鲁斯兄弟绝不会穿的鞋）。

这一次解析器的输出是如下 parsetree：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

首先应用的是用于 shoe_ready 关系的规则，它得到如下 parsetree：

```
SELECT rsh.shoename, rsh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(rsh.sh_avail, rsl.sl_avail) AS total_avail
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl
 WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
   AND (bpchareq(rsl.sl_color, rsh.slcolor)
        AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
        AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
       );
```

实际上，限定条件中的 AND 子句会是 AND 类型的操作符节点，各带一个左表达式和一个右表达式。但那样只会让本已不易读的东西更难读，而且还有更多规则要应用。
所以我只是把它们放进一些括号里，按添加的顺序把它们分组成逻辑单元，然后继续处理关系 shoe 的规则，因为它是下一个被引用且带有规则的 rangetable 项。应用该规则的结果是：

```
SELECT sh.shoename, sh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(sh.sh_avail, rsl.sl_avail) AS total_avail,
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl, shoe *OLD*,
       shoe *NEW*, shoe_data sh,
       unit un
 WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
        AND (bpchareq(rsl.sl_color, sh.slcolor)
             AND float8ge(rsl.sl_len_cm,
```

```
                            float8mul(sh.slminlen, un.un_fact))
                 AND float8le(rsl.sl_len_cm,
                            float8mul(sh.slmaxlen, un.un_fact))
               )
          )
      AND bpchareq(sh.slunit, un.un_name);
```

最后我们应用那条已经十分熟悉的 shoelace 规则（这一次是在一个稍微复杂一点的 parsetree 上），得到：

```
SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl, shoe *OLD*,
       shoe *NEW*, shoe_data sh,
       unit un, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE (     (int4ge(min(sh.sh_avail, s.sl_avail), 2)
             AND (bpchareq(s.sl_color, sh.slcolor)
                  AND float8ge(float8mul(s.sl_len, u.un_fact),
                               float8mul(sh.slminlen, un.un_fact))
                  AND float8le(float8mul(s.sl_len, u.un_fact),
                               float8mul(sh.slmaxlen, un.un_fact))
                 )
             )
         AND bpchareq(sh.slunit, un.un_name)
       )
   AND bpchareq(s.sl_unit, u.un_name);
```

我们再把它化简为与规则系统最终输出等价的真实 SQL 语句：

```
SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
  FROM shoe_data sh, shoelace_data s, unit u, unit un
 WHERE min(sh.sh_avail, s.sl_avail) >= 2
   AND s.sl_color = sh.slcolor
   AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
   AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
   AND sh.sl_unit = un.un_name
   AND s.sl_unit = u.un_name;
```

规则的递归处理把一个对视图的 SELECT 重写成了一个 parsetree，它恰好等价于在没有视图的情况下 Al 必须输入的内容。

> **注意**
>
> 规则系统目前对视图规则没有递归终止机制（对其他规则才有）。
> 这并不会造成太大问题，因为要把处理推入无限循环（使后端不断膨胀，
> 直至达到内存上限）的唯一方式，是先创建一些表，然后手工用 CREATE RULE 把视图规则设置成一个从另一个选择、而另一个又从这个选择的形式。如果使用 CREATE VIEW，这种

情况绝不会发生，因为在执行第一个 CREATE VIEW 时，第二个关系尚不存在，因此第一个视图无法从第二个视图选择。

42.2.3. 非 SELECT 语句中的视图规则

上面关于视图规则的描述没有触及 parsetree 的两个细节：commandtype 和 resultrelation。事实上，视图规则并不需要这些信息（informations）。

SELECT 的 parsetree 与其他任何命令的 parsetree 之间只有少数差别。显然，它们的 commandtype 不同，而且这一次 resultrelation 会指向结果应写入的那个 rangetable 项。除此之外，其余部分完全相同。因此，假设有两个表 t1 和 t2，都具有属性 a 和 b，那么下面两条语句的解析树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

几乎是一样的。

- 范围表中包含表 t1 和 t2 的项。
- 目标列表中都包含一个变量，该变量指向表 t2 的范围表项中的属性 b。
- 条件表达式会比较两个范围中的属性 a 是否相等。

结果是，这两个 parsetree 都会产生相似的执行计划。它们都是对这两个表的连接。对于 UPDATE，优化器会把 t1 中缺失的列补入 targetlist，最终的 parsetree 会读作：

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

因而，执行器在这个连接上运行时会产生与下面语句完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但在 UPDATE 中有个小问题。执行器并不关心它所做的连接的结果将被用于什么。它只是生成一个行结果集。一个是 SELECT 命令、另一个是 UPDATE 命令这一差别，是在执行器的调用者中处理的。调用者仍然知道（通过查看解析树）这是一个 UPDATE，也知道这个结果应写入表 t1。但那 666 行里的哪一行应当被新行替换？所执行的计划是一个带限定条件的连接，可能以未知顺序产生 0 到 666 行之间的任意数量的行。

为了解决这个问题，UPDATE 和 DELETE 语句的 targetlist 中会额外加入一项：当前元组 ID（ctid）。这是一个带特殊功能的系统属性。它包含该行所在的块号以及在块中的位置。已知表之后，利用 ctid 只需取出一个数据块，就可以在一个包含数百万行、大小为 1.5GB 的表中找到特定的那一行。把 ctid 加入 targetlist 后，最终结果集可以定义为：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在还要引入 Postgres 的另一个细节。目前，表行不会被覆盖，这也是 ABORT TRANSACTION 之所以很快的原因。在 UPDATE 中，新结果行会被插入表中（去掉 ctid 之后），而 ctid 所指向的行，其元组头中的 cmax 和 xmax 项会被设置为当前命令计数器和当前事务 ID。于是旧行被隐藏起来，事务提交后，清理器（vacuum）最终就可以真正移除它。

知道了那一切之后，我们就可以用完全相同的方式把视图规则应用到任何命令上。没有区别。

42.2.4. Postgres 中视图的能力

上文演示了规则系统如何把视图定义整合进原始 parsetree。在第二个示例中，从一个视图发出的简单 SELECT 最终生成了一个四表连接的 parsetree（unit 以不同名称使用了两次）。

42.2.4.1. 好处

用规则系统实现视图的好处在于：优化器能够在单个 parsetree 中同时看到哪些表必须被扫描、这些表之间的关系、来自视图的限制条件，以及原始查询自身的条件。
即使原始查询本身已经是对若干视图的连接，情况也仍然如此。现在，
优化器必须决定执行查询的最佳路径。优化器掌握的信息越多，这个决定就能做得越好。
Postgres实现的规则系统能够保证，到此为止，关于该查询的全部可用信息都已经集中在这里。

42.2.4.2. 隐忧

曾经有很长一段时间，Postgres 的规则系统被认为是坏的。不推荐使用规则，唯一能工作的部分是视图规则。而即使是这些视图规则也会带来问题，因为规则系统无法在 SELECT 之外的语句上正确应用它们（例如，一个使用了视图数据的 UPDATE 无法工作）。

在那段时间里，开发继续推进，解析器和优化器添加了许多特性。
规则系统与它们的能力越来越脱节，着手修复也越来越难。于是，没有人去做。

到了 6.4，有人锁上门、深吸一口气，把这该死的东西彻底翻修了一遍。出来的就是本文所描述的这种能力的规则系统。但仍有一些构造没有得到处理，还有一些因为 Postgres 查询优化器目前不支持而失败。

- 带聚合列的视图有严重的问题。限定条件中的聚合表达式必须放在子查询中使用。
  目前无法对两个各带一个聚合列的视图做连接，并在限定条件中比较这两个聚合值。
  眼下可以把这些聚合表达式放进带有适当参数的函数中，再在视图定义里使用它们。
- 目前不支持 union 的视图。把一个简单的 SELECT 重写成 union 很容易。但如果该视图是一个做更新的连接的一部分，就有点困难了。
- 不支持视图定义中的 ORDER BY 子句。
- 不支持视图定义中的 DISTINCT。

没有好的理由说明，优化器为什么不应该处理那些解析器由于 SQL 语法限制而永远无法产生的 parsetree 构造。作者希望这些条目将来会消失。

42.2.5. 实现上的副作用

用上面描述的规则系统来实现视图，有一个有趣的副作用。下面这个操作看起来不起作用：

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES ('sh5', 0, 'black');
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename  |sh_avail|slcolor
----------+--------+----------
sh1       |       2|black
sh3       |       4|brown
sh2       |       0|black
sh4       |       3|brown
```

```
(4 rows)
```

有趣的是，INSERT 的返回码给了我们一个对象 ID，并告诉我们已经插入了 1 行。但它并没有出现在 shoe_data 中。查看数据库目录可以发现，视图关系 shoe 的数据库文件现在似乎有了一个数据块。而这确实是真的。

我们还可以执行一条　DELETE，如果它不带限定条件，它会告诉我们已经删除了一些行，而下次 vacuum 运行将把该文件重置为零大小。

这种行为的原因在于，INSERT 的 parsetree 没有在任何变量中引用 shoe 关系。targetlist 只包含常量值。因此没有规则可应用，它原封不动地进入执行，行被插入。DELETE 也是如此。

要改变这种情况，我们可以定义规则来修改非 SELECT 查询的行为。这是下一节的主题。

42.3. INSERT、UPDATE 和 DELETE 上的规则

42.3.1. 与视图规则的区别

定义在 ON INSERT、UPDATE 和 DELETE 上的规则与前一节描述的视图规则完全不同。首先，它们的 CREATE RULE 命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- INSTEAD 关键字是可选的。
- 伪关系 NEW 和 OLD 可以派上用场。
- 它们可以有规则条件。

第二，它们不是就地修改解析树，而是创建零棵或多棵新解析树，并且可能丢弃原始查询树。

42.3.2. 这些规则如何工作

记住以下语法：

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

在后文中，“更新规则”是指定义在 ON INSERT、UPDATE 或 DELETE 上的规则。

当解析树的结果关系和命令类型分别等于 CREATE RULE 命令中给出的对象和事件时，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个解析树列表，初始时该列表为空。
动作可以有零个（NOTHING　关键字）、一个或多个。为简化说明，我们来看只有一个动作的规则。这个规则可以有条件，也可以没有条件；它可以是 INSTEAD，也可以不是。

什么是规则条件？它是一种限制，用来说明何时执行规则动作、何时不执行。这个条件只能引用 NEW 和/或 OLD 伪关系，它们基本上就是作为对象给出的那个关系，只是带有特殊含义。

因此，对于只有一个动作的规则，有以下四种情况会产生相应的解析树。

- 没有条件，也不是 INSTEAD：

    - 规则动作的解析树，其中附加了原始解析树的条件。
- 没有条件，但带有 INSTEAD：
    - 规则动作的解析树，其中附加了原始解析树的条件。
- 给出了条件，不是 INSTEAD：
    - 规则动作的解析树，其中附加了规则条件和原始解析树的条件。
- 给出了条件，带有 INSTEAD：
    - 规则动作的解析树，其中附加了规则条件和原始解析树的条件。
    - 原始解析树，其中附加了规则条件取反后的条件。

最后，如果规则不是 INSTEAD，就把未经更改的原始解析树加入列表。由于只有带条件的 INSTEAD 规则已经加入了原始解析树，因此，对于只有一个动作的规则，最终最多会得到两棵输出解析树。

由规则动作生成的解析树会再次送入重写系统，随后可能还会有更多规则被应用，
从而产生更多或更少的解析树。因此，规则动作中的解析树必须具有不同的命令类型，
或者具有不同的结果关系。否则，这种递归过程就会陷入循环。目前内置的递归上限是 10 次迭代。如果经过 10 次迭代之后仍有更新规则要应用，规则系统就会认为出现了多个规则定义之间的循环，并中止事务。

`pg_rewrite` 系统目录中动作里的解析树只是模板。由于它们可以引用 NEW 和 OLD 的范围表项，因此在使用前必须进行一些替换。对任何 NEW 的引用，都会先在原始查询的目标列表中查找相应项。如果找到了，就用该项的表达式替换该引用。否则，NEW 就与 OLD 含义相同。任何对 OLD 的引用，都会被替换为对作为 resultrelation 的那个 rangetable 项的引用。

42.3.2.1. 第一个规则：逐步分析

我们想跟踪 `shoelace_data` 关系中 sl_avail 列的变化。因此，我们建立一个日志表和一条规则，使其在 `shoelace_data` 上执行 UPDATE 时，每次都为我们写一些日志项。

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    name,          -- who did it
    log_when   datetime       -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail != OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    getpgusername(),
                                    'now'::text
                                );
```

一个有趣的细节是规则 INSERT 动作中把 'now' 转换为 text 类型。如果不这样做，解析器在 CREATE RULE 时就会看到 `shoelace_log` 中的目标类型是 datetime，并试图用它构造一个常量——而且会成功。于是一个常量 datetime 值会被存储在规则动作中，所有日志项的时间都

将是 CREATE RULE 语句的时间。这并不完全是我们想要的。这个转换使解析器改而构造出一个 datetime('now'::text)，它会在规则执行时才被求值。

现在 Al 执行了：

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6

al_bundy->     WHERE sl_name = 'sl7';
```

然后我们查看日志表：

```
al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这正是我们预期的结果。后台发生的事情如下。解析器创建了 parsetree（这一次突出显示的是原始 parsetree 中的部分，因为对更新规则而言，操作的基础是规则动作）：

```
UPDATE shoelace_data SET sl_avail = 6
      FROM shoelace_data shoelace_data
     WHERE bpchareq(shoelace_data.sl_name, 'sl7');
```

此时存在一条 ON UPDATE 规则 'log_shoelace'，它带有如下规则条件表达式：

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

以及一个动作：

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avail,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_log shoelace_log;
```

不要相信 pg_rules 系统视图的输出。它专门处理了 INSERT 中只有对 NEW 和 OLD 的引用的情形，输出 INSERT 的 VALUES 格式。实际上，INSERT ... VALUES 和 INSERT ... SELECT 在 parsetree 层面没有区别。它们都有 rangetable、targetlist，可能还有限定条件等。优化器稍后会决定为该 parsetree 创建哪种类型的执行计划：result、seqscan、indexscan、join 或其他。如果 parsetree 中不再有对 rangetable 项的引用（leftin），它就会变成一个 result 执行计划（即 INSERT ... VALUES 版本）。上面的规则动作确实（truely）两种结果都可能出现。

该规则是一条带条件的非 INSTEAD 规则，因此规则系统必须返回两个 parsetree：修改后的规则动作，以及原始 parsetree。第一步中，原始查询的 rangetable 会并入规则动作的 parsetree，得到：

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log;
```

第 2 步将规则条件加进去，因此结果集被限制为 sl_avail 发生变化的行：

```
INSERT INTO shoelace_log SELECT
```

```
        *NEW*.sl_name, *NEW*.sl_avai,
        getpgusername(), datetime('now'::text)
   FROM shoelace_data shoelace_data, shoelace_data *NEW*,
        shoelace_data *OLD*, shoelace_log shoelace_log
  WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

第 3 步把原始 parsetree 的条件加进去，把结果集进一步限制为只有那些原始 parsetree 会触及的行：

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

第 4 步把 NEW 引用替换为原始 parsetree 中的 targetlist 项，或者用结果关系中相应的变量引用来替换：

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(6, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

第 5 步把 OLD 引用替换为 resultrelation 引用：

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(6, shoelace_data.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

至此就完成了。化简到极致之后，规则系统的输出是一个包含两个 parsetree 的列表，它们等同于以下语句：

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE 6 != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

它们会按这个顺序执行，而这正是该规则所定义的行为。上述替换（subtitutions）以及附加的条件能够保证：如果原始查询是下面这样，

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

就不会写入任何日志项，因为这一次原始 parsetree 不包含 sl_avail 的 targetlist 项，NEW.sl_avail 会被 shoelace_data.sl_avail 替换，于是规则产生的额外查询是：

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, shoelace_data.sl_avail,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE shoelace_data.sl_avail != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

而该条件永远不可能为真。由于 INSERT ... SELECT 与 INSERT ... VALUES 在 parsetree 层面没有区别，如果原始查询修改多行，这种机制同样能够正常工作。因此，如果 Al 发出如下命令：

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上会更新四行（sl1、sl2、sl3 和 sl4）。但 sl3 本来就已经是 sl_avail = 0。这一次，原始 parsetree 的条件不同，因此规则会产生额外的 parsetree：

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 0,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE 0 != shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

这棵 parsetree 必然会插入三条新的日志项。这完全正确。

原始 parsetree 最后执行至关重要。Postgres 的 “traffic cop” （交通警察）会在两个 parsetree 的执行之间做一次命令计数器递增，使第二个能看到第一个所做的更改。如果先执行 UPDATE，那么所有行都已经被设为零，记日志的 INSERT 就找不到任何满足 0 != shoelace_data.sl_avail 的行了。

42.3.3. 与视图的协作

为了防止有人像前面提到的那样在视图关系上 INSERT、UPDATE 和 DELETE 不可见数据，一种简单的办法是让那些解析树直接被丢弃。我们创建如下规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

如果现在 Al 尝试对视图关系 `shoe` 执行这些操作中的任何一种，规则系统就会应用这些规则。由于这些规则没有动作而且是 INSTEAD，生成的解析树列表将为空；整个查询也就什么都不会做，因为规则系统处理完后已经没有任何内容可供优化或执行。

> **注意**
>
> 这一事实可能会让前端应用程序感到困惑，因为数据库上绝对什么都没有发生，
> 因而后端不会为该查询返回任何东西。在 libpq 中连 PGRES_EMPTY_QUERY 之类的东西都得不到。在 psql 中，什么都不会发生。这一点将来可能会改变。

另一种更完善的做法，是创建一些规则，把解析树重写成在真实表上执行正确操作的解析树。要在 shoelace 视图上做到这一点，我们创建下列规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data SET
           sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

现在有一批鞋带到货进入 Al 的商店，随附一份很长的部件清单。Al 不太擅长计算，因此我们不想让他手工更新 shoelace 视图。于是我们建立两个小表：一个让他可以插入部件清单中的项目，另一个则采用一个特殊技巧。创建命令如下：

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace SET
           sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在 Al 可以坐下来随便干点什么，直到

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
```

```
(3 rows)
```

与部件清单上的内容完全一致为止。我们快速查看一下当前数据，

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

把到货的鞋带入库：

```
al_bundy=> INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

然后检查结果：

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(8 rows)

al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 19:14:45 1998 MET DST
sl3       |      10|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl6       |      20|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl8       |      21|Al     |Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一条 INSERT ... SELECT 到这些结果，中间要经历相当长的一段过程。对它的描述将是本文中的最后一个（但不是最后一个例子 :-)）。首先是解析器的输出：

```
INSERT INTO shoelace_ok SELECT
       shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则 'shoelace_ok_ins'，它会把这一输出转换成：

```
UPDATE shoelace SET
       sl_avail = int4pl(shoelace.sl_avail, shoelace_arrive.arr_quant)
```

```
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace
 WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name);
```

同时会丢弃针对 shoelace_ok 的原始 INSERT。这个重写后的查询会再次交给规则系统，随后应用的规则 'shoelace_upd' 产生：

```
UPDATE shoelace_data SET
       sl_name = shoelace.sl_name,
       sl_avail = int4pl(shoelace.sl_avail, shoelace_arrive.arr_quant)
,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data
 WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);
```

这同样是一条 INSTEAD 规则，因此前一个解析树会被丢弃。注意，这个查询仍然使用视图 shoelace。但规则系统在这个循环中尚未完成，因此它会继续在其上应用规则 '_RETshoelace'，于是得到：

```
UPDATE shoelace_data SET
       sl_name = s.sl_name,
       sl_avail = int4pl(s.sl_avail, shoelace_arrive.arr_quant),
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

又有一条更新规则被应用，于是车轮继续转动，我们进入了第 3 轮重写。这一次应用的是规则 'log_shoelace'，这产生了额外的解析树：

```
INSERT INTO shoelace_log SELECT
       s.sl_name,
       int4pl(s.sl_avail, shoelace_arrive.arr_quant),
       getpgusername(),
       datetime('now'::text)
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
```

```
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
   AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant), s.sl_
avail);
```

到这里，规则系统已经没有更多规则可用，返回生成的解析树。于是我们最终得到两棵解析树，它们等同于以下 SQL 语句：

```
INSERT INTO shoelace_log SELECT
       s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       getpgusername(),
       'now'
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

UPDATE shoelace_data SET
       sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是：来自一个关系的数据被插入到另一个关系中，这个插入被改写为对第三个关系的更新，再被改写为对第四个关系的更新外加在第五个关系中记录该更新，
最后整个过程被化简为两个查询。

这里有个稍显难看的小细节。观察这两个查询就会发现，`shoelace_data` 关系在范围表中出现了两次，而实际上完全可以缩减成一次。优化器不会处理这一点，因此规则系统为 INSERT 输出的执行计划会是：

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

而省略那个额外的范围表项则会得到：

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
```

```
                ->  Seq Scan on shoelace_arrive
```

后者完全会在日志关系中产生相同的项。因此，规则系统导致了对 shoelace_data 关系的一次完全不必要的额外扫描。而在 UPDATE 中，同样的冗余扫描还会再发生一次。不过，能让这一切总体上工作起来，已经是一项相当艰巨的工作。

最后来演示一下 Postgres 规则系统及其威力。有一位漂亮的金发女郎卖鞋带。而 Al 永远不会意识到的是，她不仅漂亮，还很聪明--甚至有点太聪明了。于是，Al 不时会订购一些完全卖不出去的鞋带。这一次他订购了 1000 双品红色（magenta）鞋带，而且由于另一种鞋带暂时缺货、而他已承诺购买一些，他又为粉色（pink）的鞋带做好了数据库准备：

```
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

由于这种事经常发生，我们必须不时地找出那些绝对不适合任何鞋子的鞋带项。
我们可以每次都用一条复杂的语句来完成，也可以为此建立一个视图。这个视图是：

```
CREATE VIEW shoelace_obsolete AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    1000|magenta   |    40|inch    |    101.6
```

那 1000 双品红色鞋带，我们得先向 Al 讨了债才能把它们扔掉，不过那是另一个问题了。粉色的条目我们要删掉。为了给 Postgres 增加一点难度，我们不直接删除，而是再创建一个视图：

```
CREATE VIEW shoelace_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

然后这样执行：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             WHERE sl_name = shoelace.sl_name);
```

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
```

```
sl6       |       20|brown     |   0.9|m       |        90
(9 rows)
```

作用于一个视图上的 DELETE，其子查询条件总共使用了四个嵌套/连接的视图，其中一个视图自身又带有一个包含视图的子查询条件，并且还用到了计算得到的视图列，
最终仍会被重写成单棵解析树，从真正的表中删除所请求的数据。

我想在现实世界中，大概只有很少的场景会需要这样的构造。但知道它确实能工作，
总归让人安心。

> **实情是**
>
> 在写这一章的过程中，我又发现了一个 bug。不过修完那个 bug 之后，这一切居然能工作，倒是让我有点惊讶。

42.4. 规则和权限

由于 Postgres 规则系统会重写查询，因此可能访问原始查询中并未使用的其他表或视图。使用更新规则时，这甚至可能包括对表的写访问。

重写规则没有单独的所有者。关系（表或视图）
的所有者会自动成为为其定义的重写规则的所有者。Postgres 规则系统改变了默认访问控制系统的行为。由于规则而使用的关系，
会在重写期间根据其上定义了该规则的关系所有者的权限进行检查。这意味着，
用户只需要对其查询中明确命名的表/视图具有所需的权限。

例如，某个用户有一份电话号码列表，其中一部分是私人的，另一部分对办公室秘书有用。
该用户可以这样构造：

```
CREATE TABLE phone_data (person text, phone text, private bool);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

除了该用户本人（以及数据库超级用户）之外，没有人可以访问 phone_data 表。但由于 GRANT 的存在，秘书可以对 phone_number 视图执行 SELECT。规则系统会把对 phone_number 的 SELECT 重写成对 phone_data 的 SELECT，并附加只选取 private 为假的项的条件。由于该用户是 phone_number 的所有者，对 phone_data 的读访问会按照该用户的权限进行检查，于是查询被判定为允许。对 phone_number 本身的访问检查仍会执行，因此除秘书之外没有人能使用它。

权限是逐条规则检查的。因此，目前秘书是唯一能看到公开电话号码的人。
但秘书还可以再建立一个视图，并把该视图授予公众访问。这样，
任何人都可以通过秘书的视图看到 phone_number 中的数据。秘书做不到的是创建一个直接访问 phone_data 的视图（其实可以创建，但它不会起作用，因为每次访问都会在权限检查时中止事务）。而且，一旦用户发现秘书开放了其 phone_number 视图，用户就可以 REVOKE 秘书的访问权限。那样一来，对秘书视图的任何访问都会立即失败。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但实际上并不是。如果它不这样工作，
秘书也可以建立一个与 phone_number 具有相同列的表，每天把数据复制进去。那样一来，这些就是秘书自己的数据，他同样可以把访问权限授予任何人。GRANT 的含义本来就是“我信任你”。如果某个你信任的人做了上面的事，那么该重新考虑这份信任，然后使用 REVOKE 了。

这种机制对更新规则同样有效。在上一节的示例中，Al 数据库中那些表的所有者可以把 shoelace 视图上的 SELECT、INSERT、UPDATE 和 DELETE 权限授予 al。但对 shoelace_log 只授予 SELECT 权限。写日志记录的规则动作仍会被成功执行，Al 也能看到日志记录。但他不能伪造记录，也不能操纵或删除已有记录。

> **警告**
>
> GRANT ALL 目前会包含 RULE 权限。这意味着被授权的用户可以先删除规则、做出更改、再把规则装回去。我认为这一点应该尽快改变。

42.5. 规则与触发器

许多能够用触发器完成的事情，同样也可以用 Postgres 规则系统实现。目前不能用规则实现的内容是一些种类的约束。你可以放置一条带条件的规则：当某列的值没有出现在另一张表中时，把查询重写成 NOTHING。但那样做会悄无声息地丢弃数据，这并不是好主意。如果需要检查值的有效性，并在值无效时生成错误消息，目前就必须使用触发器。

另一方面，在视图上因 INSERT 而触发的触发器可以做到与规则相同的事情：把数据放到别处，并抑制向视图本身的插入。但在 UPDATE 或 DELETE 上它就无能为力了，因为视图关系中没有被扫描的真实数据，触发器因而永远不会被调用。这时只有规则才能解决问题。

对于两者都能实现的事情，哪一种更好取决于数据库的使用方式。
触发器会对每个受影响的行触发一次，而规则会修改解析树或生成额外的解析树。因此，
如果一条语句会影响很多行，那么发出一条额外查询的规则往往会比触发器更快，
因为触发器必须为每一行调用一次，并反复执行其操作。

例如：有两个表：

```
CREATE TABLE computer (
    hostname        text     -- indexed
    manufacturer    text     -- indexed
);

CREATE TABLE software (
    software        text,    -- indexed
    hostname        text     -- indexed
);
```

两个表都有数千行，并且 hostname 上的索引是唯一索引。hostname 列包含计算机的完整域名。规则或触发器应实现这样一个约束：从 software 中删除引用了被删除主机的行。由于从 computer 中删除的每一行都会调用触发器，它可以在一个准备并保存好的计划中使用语句：

```
DELETE FROM software WHERE hostname = $1;
```

并通过参数传入 hostname。规则则会写成：

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在我们来看不同类型的删除。对于下面这种情况：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

computer 表会通过索引扫描（很快），由触发器发出的查询也会是一次索引扫描（同样很快）。规则产生的额外查询则是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，优化器将创建一个计划：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

因此，触发器实现和规则实现之间的速度差异不会太大。在下一个删除场景中，我们想去掉全部 2000 台 hostname 以 'old' 开头的计算机。有两种查询可以做到这件事。其中一种是：

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

规则查询的计划将是：

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一种可能的查询是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

其执行计划为：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这说明，当多个条件表达式通过 AND 组合在一起时，优化器并没有意识到，对 computer 上 hostname 的限制同样也可以用于 software 上的索引扫描，而在该查询的正则表达式版本里它却能做到。触发器会对需要删除的 2000 台旧计算机中的每一台各调用一次，这意味着对 computer 做一次索引扫描，并对 software 做 2000 次索引扫描。规则实现则用两条走索引的查询完成这件事。不过，在顺序扫描这种情况下，规则是否仍然更快，还取决于 software 表的总体大小。即使所有用于查找的索引块很快都会进入缓存，通过 SPI 管理器执行 2000 条查询仍然要耗费一些时间。

我们要看的最后一个查询是：

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

同样，这也可能导致从 computer 中删除很多行。因此触发器又会向执行器触发很多条查询。但规则的计划又将是在两个索引扫描上的嵌套循环，只不过使用了 computer 上的另一个索引：

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

它来自规则的查询：

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
```

```
                    AND software.hostname = computer.hostname;
```

在上述任何一种情况下，规则系统产生的额外查询都或多或少与该查询影响的行数无关。

另一种情况是 UPDATE 中是否执行动作取决于某个属性是否发生变化。在 Postgres 6.4 版中，规则事件的属性说明被禁用了（它最迟会在 6.5 恢复，也许更早--敬请期待）。因此，目前创建像 shoelace_log　例子中那样规则的唯一方式是使用规则条件。这会导致一条额外的查询总是被执行，即使所关注的属性根本不可能变化，因为它没有出现在初始查询的目标列表中。
等这一功能重新启用后，它将成为规则相对触发器的又一个优势。在这种情况下，
触发器的优化按定义必然失败，
因为"其动作只在某个特定属性被更新时执行"这一事实隐藏在它的功能之中。
触发器的定义只允许在行级指定，因此每当某一行被触及，触发器就必须被调用来做决定。
而规则系统会通过查看目标列表知道这一点，并在属性未被涉及时完全抑制那条额外的查询。
所以，无论带不带条件，规则只会在可能有事可做时才执行其扫描。

只有当规则动作导致了规模很大且条件很差的连接（优化器无能为力的情形）时，
规则才会明显慢于触发器。规则是一把大锤。不加小心地使用大锤可能造成巨大的破坏。
但用对了手法，它们能把任何钉子一击入帽。

第 43 章 索引扩展接口

到目前为止所描述的过程使我们能够定义新类型、新函数以及新操作符。然而，
我们还不能在一个新类型或其操作符上定义一个二级索引（例如 B-tree、R-tree 或哈希访问方法）。

回头看图 37.1。右半部分显示了我们必须修改的目录，以便告诉 Postgres 如何在索引中使用用户定义的类型和/或用户定义的操作符（即 pg_am， pg_amop， pg_amproc， pg_operator 和 pg_opclass）。遗憾的是，没有一条简单的命令可以完成这件事。我们将通过一个贯穿始终的示例来演示如何修改这些目录：为 B-tree 访问方法定义一个新的操作符类，以便按绝对值升序存储和排序复数。

pg_am 类为每个用户定义的访问方法保存一个实例。堆访问方法的支持内置于 Postgres 中，但每个其他访问方法都在这里描述。其模式是

表 43.1. 索引模式

属性	描述
amname	访问方法的名称
amowner	属主在 pg_user 中实例的对象 id
amstrategies	该访问方法的策略数目（见下文）
amsupport	该访问方法的支持例程数目（见下文）
amorderstrategy	如果索引不提供排序顺序则为零，否则是描述该排序顺序的策略操作符的策略号
amgettuple	
aminsert	
...	访问方法接口例程的过程标识符。例如，打开、关闭访问方法以及从中获取实例的 regproc id 就出现在这里。

pg_am 中该实例的对象 ID 被用作许多其他类中的外键。你不需要向这个类添加新实例；你所关心的只是你想扩展的访问方法实例的对象 ID：

```
SELECT oid FROM pg_am WHERE amname = 'btree';

 oid
-----
 403
(1 row)
```

我们稍后会在 WHERE 子句中使用那条 SELECT。

amstrategies 属性的存在是为了标准化跨数据类型的比较。例如，B-tree 对键施加了严格的从小到大的顺序。由于 Postgres 允许用户定义操作符，Postgres 不能仅凭操作符名称（例如 ">" 或 "<"）就判断它是哪一类比较。事实上，某些访问方法根本不施加任何排序。例如，R-tree 表达一种矩形包含关系，而哈希数据结构只表达基于哈希函数值的按位相似性。Postgres 需要某种一致的方式，来接受你查询中的条件、查看操作符，然后判断是否存在可用的索引。这蕴含着 Postgres 需要知道，例如，"<=" 和 ">" 操作符划分一个 B-tree。Postgres 使用策略来表达操作符与它们可用于扫描索引的方式之间的这些关系。

定义一组新的策略超出了本讨论的范围，但我们将解释 B-tree 策略如何工作，因为你要添加一个新的操作符类就需要了解它。在 pg_am 类中，amstrategies 属性是该访问方法所定义的策略数目。对 B-tree 来说，这个数是 5。这些策略对应于

表 43.2. B-tree 策略

操作	索引号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

这个思路是：你需要把与上述比较对应的过程添加到 pg_amop 关系中（见下文）。访问方法代码可以使用这些策略号（不管数据类型是什么）来弄清如何划分 B-tree、计算选择性等等。现在还不用担心添加过程的细节；只需理解：对 int2, int4, oid, 以及 B-tree 可以操作的每个其他数据类型，都必须存在一组这样的过程。

有时，仅靠策略信息不足以让系统弄清如何使用索引。
某些访问方法还需要其他支持例程才能工作。例如，B-tree 访问方法必须能够比较两个键，并判断其中一个是大于、等于还是小于另一个。类似地，R-tree 访问方法必须能够计算矩形的交集、并集和大小。这些操作并不对应 SQL 查询中的用户条件；它们是访问方法内部使用的管理例程。

为了在所有 Postgres 访问方法之间一致地管理多样的支持例程，pg_am 包含一个名为 amsupport 的属性。该列记录一个访问方法所使用的支持例程数目。对 B-tree 来说，这个数是一——接受两个键并根据第一个键小于、等于还是大于第二个键而返回 -1、0 或 +1 的例程。

> **注意**
>
> 严格地说，该例程可以返回一个负数（< 0）、0 或一个非零正数（> 0）。

pg_am 中的 amstrategies 项只是所论访问方法定义的策略数目。小于、小于等于等的过程并不出现在 pg_am 中。类似地，amsupport 只是该访问方法所需的支持例程数目。实际的例程列在别处。

顺便说一下，amorderstrategy 项表明访问方法是否支持有序扫描。零表示不支持；如果支持，amorderstrategy 就是对应排序操作符的策略例程编号。例如，btree 的 amorderstrategy = 1，即它的"小于"策略号。

下一个我们关心的类是 pg_opclass。这个类的存在只是为了把一个操作符类名（或许还有一个默认类型）与一个操作符类 oid 相关联。一些现有的操作符类是 int2_ops, int4_ops, 和 oid_ops。你需要把带有你的操作符类名（例如 complex_abs_ops）的一个实例添加到 pg_opclass。这一实例的 oid 将是其他类（特别是 pg_amop）中的外键。

```
INSERT INTO pg_opclass (opcname, opcdeftype)
    SELECT 'complex_abs_ops', oid FROM pg_type WHERE typname =
 'complex';

SELECT oid, opcname, opcdeftype
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';
```

```
  oid   |     opcname     | opcdeftype
--------+-----------------+------------
 277975 | complex_abs_ops |     277946
(1 row)
```

注意，你的 pg_opclass 实例的 oid 会不同！不过不必担心。稍后我们会像这里获取类型的 oid 一样从系统中取得这个数。

上述示例假定你希望把这个新操作符类设为 complex 数据类型的默认索引操作符类。如果不是这样，只需向 opcdeftype 中插入零，而不是插入该数据类型的 oid：

```
INSERT INTO pg_opclass (opcname, opcdeftype) VALUES ('complex_abs_
ops', 0);
```

现在我们有了一个访问方法和一个操作符类。我们还需要一组操作符。
定义操作符的过程已在本手册前面讨论过。对于 Btree 上的 complex_abs_ops 操作符类，我们需要的操作符是：

```
absolute value less-than
absolute value less-than-or-equal
absolute value equal
absolute value greater-than-or-equal
absolute value greater-than
```

假设实现所定义函数的代码存储在文件 PGROOT/src/tutorial/complex.c 中

C 代码的一部分如下所示：（注意，在余下的示例中我们只展示相等操作符。其他四个操作符非常类似。细节请参阅 complex.c 或 complex.source。）

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

 bool
 complex_abs_eq(Complex *a, Complex *b)
 {
     double amag = Mag(a), bmag = Mag(b);
     return (amag==bmag);
 }
```

我们像下面这样让 Postgres 知道该函数：

```
CREATE FUNCTION complex_abs_eq(complex, complex)
              RETURNS bool
              AS 'PGROOT/tutorial/obj/complex.so'
              LANGUAGE 'c';
```

这里有几件重要的事情正在发生。

首先，注意这里正在为 complex 定义小于、小于等于、等于、大于等于和大于操作符。我们只能有一个名为（例如）= 并且两个操作数都取 complex 类型的操作符。在本例中我们没有其他用于

complex 的 = 操作符，但如果我们在构造一种实用的数据类型，我们可能希望 = 是复数的普通相等操作。在那种情况下，我们就需要为 complex_abs_eq 使用其他某个操作符名。

其次，尽管 Postgres 能处理名称相同但输入数据类型不同的操作符，C 却只能处理给定名称的一个全局例程。因此，我们不应该把 C 函数简单命名成 abs_eq 之类。通常，在 C 函数名中包含数据类型名称是个好习惯，这样就不会与其他数据类型的函数发生冲突。

第三，我们本可以把该函数的 Postgres 名称取为 abs_eq，并依靠 Postgres 通过输入数据类型把它与任何其他同名的 Postgres 函数区分开。为了让示例保持简单，这里我们让 C 层和 Postgres 层的函数使用相同的名称。

最后，注意这些操作符函数返回布尔值。访问方法依赖这一事实。（另一方面，
支持函数返回的是特定访问方法所期望的类型——在本例中就是一个有符号整数。）
文件中的最后一个例程是我们在讨论 pg_am 类的 amsupport 属性时提到的"支持例程"。我们稍后会用到它。现在先忽略它。

现在我们准备好定义操作符了：

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         )
```

这里的重要内容是过程名（即上面定义的 C 函数）以及限制和连接选择性函数。你应当直接使用示例中所用的选择性函数（见 complex.source）。注意，小于、等于和大于情形有不同的此类函数。必须提供这些函数，否则优化器将无法有效地使用该索引。

下一步是把这些操作符的项添加到 pg_amop 关系中。为此，我们需要刚才定义的这些操作符的 oid。我们将查找所有取两个 complex 类型操作数的操作符的名称，并挑出我们的那些：

```
    SELECT o.oid AS opoid, o.oprname
     INTO TABLE complex_ops_tmp
     FROM pg_operator o, pg_type t
     WHERE o.oprleft = t.oid and o.oprright = t.oid
      and t.typname = 'complex';

 opoid  | oprname
--------+---------
 277963 | +
 277970 | <
 277971 | <=
 277972 | =
 277973 | >=
 277974 | >
(6 rows)
```

（同样，你的某些 oid 号几乎肯定会有所不同。）我们感兴趣的操作符是 oid 从 277970 到 277974 的那些。你得到的值很可能不同，你应当用它们替换下文中的值。我们将用一条 select 语句来完成这件事。

现在我们准备好用新操作符类来更新 pg_amop 了。在整个讨论中最重要的一点是：在 pg_amop 中，操作符是按从小到大排序的。我们添加所需的实例：

```
    INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy)
        SELECT am.oid, opcl.oid, c.opoid, 1
        FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
        WHERE amname = 'btree' AND
            opcname = 'complex_abs_ops' AND
            c.oprname = '<';
```

然后对其他操作符照此办理，替换上文第三行中的 "1" 和最后一行中的 "<"。注意顺序："小于"是 1，"小于等于"是 2，"等于"是 3，"大于等于"是 4，"大于"是 5。

下一步是注册此前在讨论 `pg_am` 时描述过的"支持例程"。该支持例程的 `oid` 存储在 `pg_amproc` 类中，以访问方法 `oid` 和操作符类 `oid` 为键。首先，我们需要在 Postgres 中注册该函数（回想一下，我们把实现该例程的 C 代码放在了实现操作符例程的那个文件的底部）：

```
    CREATE FUNCTION complex_abs_cmp(complex, complex)
     RETURNS int4
     AS 'PGROOT/tutorial/obj/complex.so'
     LANGUAGE 'c';

    SELECT oid, proname FROM pg_proc
     WHERE proname = 'complex_abs_cmp';

  oid   |     proname
--------+-----------------
 277997 | complex_abs_cmp
(1 row)
```

（同样，你的 `oid` 号很可能会有所不同。）我们可以像下面这样添加新实例：

```
    INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
        SELECT a.oid, b.oid, c.oid, 1
            FROM pg_am a, pg_opclass b, pg_proc c
            WHERE a.amname = 'btree' AND
                b.opcname = 'complex_abs_ops' AND
                c.proname = 'complex_abs_cmp';
```

这样就完成了！（呼。）现在应该可以在 `complex` 列上创建并使用 btree 索引了。

第 44 章 索引代价估计函数

作者

由 Tom Lane[1] 于 2000-01-24 撰写。

注意

这些内容最终应成为关于编写新索引访问方法的更大章节的一部分。

每种索引访问方法都必须提供一个供规划器/优化器使用的代价估计函数。该函数的过程 OID 记录在访问方法的 `pg_am` 项的 `amcostestimate` 字段中。

注意

在 Postgres 7.0 之前，注册索引特定的代价估计函数使用的是另一种方案。

amcostestimate 函数会得到一个已判定可用于该索引的 WHERE 子句列表。它必须返回访问索引的代价估计以及 WHERE 子句的选择率估计（即在索引扫描期间将被检索的主表元组所占的比例）。对于简单情况，代价估计器几乎全部工作都可以通过调用优化器中的标准例程完成；设置 amcostestimate 函数的意义在于让索引访问方法提供索引类型特定的知识，以便在可能改进标准估计时加以利用。

每个 amcostestimate 函数必须具有如下签名：

```
void
amcostestimate (Query *root,
                RelOptInfo *rel,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity);
```

前四个参数是输入：

root

正在处理的查询。

rel

索引所在的关系。

index

索引本身。

[1] mailto:tgl@sss.pgh.pa.us

indexQuals

索引限定条件子句的列表（隐含 AND 连接）；NIL 列表表示没有可用的限定条件。

后三个参数是按引用传递的输出：

*indexStartupCost

设置为索引启动处理的代价

*indexTotalCost

设置为索引处理的总代价

*indexSelectivity

设置为索引选择率

注意，代价估计函数必须用 C 编写，而不能用 SQL 或任何可用的过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问代价应按 src/backend/optimizer/path/costsize.c 所用的单位计算：顺序磁盘块读取的代价为 1.0，非顺序读取的代价为 random_page_cost，而处理一个索引元组的代价通常取为 cpu_index_tuple_cost（这是一个用户可调的优化器参数）。此外，对于索引处理期间调用的任何比较操作符（尤其是 indexQuals 本身的求值），应收取适当的 cpu_operator_cost 倍数。

访问代价应包括扫描索引本身相关的所有磁盘和 CPU 代价，但不包括检索或处理由该索引标识的主表元组的代价。

"启动代价"是总扫描代价中在可以开始取第一个元组之前必须付出的部分。
对大多数索引来说可以取零，但启动代价很高的索引类型可能希望把它设为非零。

indexSelectivity 应设置为索引扫描期间将检索的主表元组的估计比例。对于有损索引，这通常会高于实际满足给定限定条件的元组所占的比例。

代价估计

典型的代价估计器按如下步骤进行：

1. 基于给定的限定条件，估计并返回将被访问的主表元组所占的比例。
   在没有索引类型特定知识的情况下，使用标准优化器函数 clauselist_selectivity()：

   ```
   *indexSelectivity = clauselist_selectivity(root, indexQuals,
                                              lfirsti(rel->relids));
   ```

2. 估计扫描期间将访问的索引元组数。对许多索引类型来说，它等于 indexSelectivity 乘以索引中的元组数，但也可能更多。（注意，索引的页数和元组数可以从 IndexOptInfo 结构中获得。）

3. 估计扫描期间将检索的索引页数。它可以就是 indexSelectivity 乘以索引的页数。

4. 计算索引访问代价。一个通用估计器可以这样做：

   ```
       /*
        * Our generic assumption is that the index pages will be read
        * sequentially, so they have cost 1.0 each, not random_page_
   cost.
   ```

```
        * Also, we charge for evaluation of the indexquals at each
     index tuple.
        * All the costs are assumed to be paid incrementally during
     the scan.
        */
       *indexStartupCost = 0;
       *indexTotalCost = numIndexPages +
           (cpu_index_tuple_cost + cost_qual_eval(indexQuals)) * num
    IndexTuples;
```

代价估计器函数的例子可以在 src/backend/utils/adt/selfuncs.c中找到。

按照惯例，amcostestimate函数的 pg_proc 项应显示：

```
prorettype = 0
pronargs = 7
proargtypes = 0 0 0 0 0 0 0
```

由于所有参数都没有 pg_type 中已知的类型，我们对全部参数使用零（"opaque"）。

第 45 章 GiST 索引

Gene Selkov

关于 GIST 的信息在 http://GiST.CS.Berkeley.EDU:8000/gist/ 关于不同索引和排序方案的更多内容在 http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/ 另外在伯克利数据库站点还有一些有意思的阅读材料：http://epoch.cs.berkeley.edu:8000/。

> **作者**
>
> 这段摘录来自 Eugene Selkov Jr.[1] 发送的一封电子邮件，其中包含关于 GiST 的有用信息。希望我们将来能学到更多并更新这些信息。- thomas 1998-03-01

好吧，我不能说完全理解了其中的原理，但我（几乎）成功地把 GiST 示例移植到了 linux。GiST 访问方法已经在 postgres 树中（`src/backend/access/gist`）。

伯克利的示例[2]附带这些方法的概述，并演示了二维矩形、多边形、
整数区间和文本的空间索引机制（另见伯克利的 GiST[3]）。在矩形示例中，使用 GiST 索引本应看到性能提升；它对我来说确实有效，但我没有规模足够大的矩形集合来验证这一点。
其他示例也都有效，只有多边形除外：执行

```
test=> create index pix on polytmp
test-> using gist (p:box gist_poly_ops) with (islossy);
ERROR:  cannot open pix

(PostgreSQL 6.3              Sun Feb  1 14:57:30 EST 1998)
```

我没搞懂这条错误消息的含义；它似乎是我们更应该去问开发者的问题（另见下面的注 4）。我这里想建议的是，你们当中的 linux 用户（linux==gcc?）去取上面提到的原始源码并应用我的补丁（见附件），然后告诉我们你们的感受。我觉得很酷，但周围有这么多能人，
我不想把它扣着不放。

关于这些源码的几点说明：

1. 我没能用上原始的（HPUX）Makefile，于是把古老的 postgres95 教程里的 Makefile 改了改来干活。我试着让它通用一些，但我写 Makefile 的水平很差——只是做了些照猫画虎的活儿。抱歉，不过我想它现在比原来的 makefile 稍微可移植一点了。

2. 我直接在 pgsql/src 之下构建了示例源码（只是把 tar 文件解压在那里）。前面提到的 Makefile 假定它位于 pgsql/src 下一级（在我们的例子中即 pgsql/src/pggist）。

3. 我对 *.c 文件的改动都只是关于 #include、函数原型和类型转换。除此之外，我只是扔掉了一堆未用的变量，并加了几个括号来取悦 gcc。希望我没有搞砸太多 :)

4. polyproc.sql 中有一条注释：

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- create index pix2 on polytmp using rtree (p poly_ops);
```

[1] mailto:selkovjr@mcs.anl.gov
[2] ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz
[3] http://gist.cs.berkeley.edu:8000/gist/

收到！！我以为它可能与几个版本之前的 Postgres 有关，于是试着执行了该查询。我的系统疯了，大约十分钟后我不得不干掉了 postmaster。

我会继续研究一阵子 GiST，但我也希望看到更多 R 树用法的示例。

第 46 章 链接动态装载的函数

在你创建并注册一个用户定义的函数之后，你的工作基本上就完成了。然而，
Postgres必须装载实现你的函数的目标代码（例如一个.o文件或一个共享库）。如前所述，
Postgres会在运行时按需装载你的代码。为了让你的代码能够被动态装载，
你可能必须以一种特殊的方式对它进行编译和链接编辑。
本节简要描述在你把用户定义的函数装载进运行中的Postgres服务器之前，
需要怎样进行编译和链接编辑。

如果你有具体问题，应该预期去阅读（反复阅读）C 编译器 cc(1) 和链接编辑器 ld(1) 的手册页。此外，contrib 区域（`PGROOT/contrib`）和 `PGROOT/src/test/regress`目录中的回归测试套件包含这一过程的若干可用示例。如果你照抄一个示例，就不会有任何问题。

下面将使用下列术语：

- 动态装载（Dynamic loading）是Postgres对一个目标文件所做的事情。目标文件被复制进运行中的Postgres服务器，文件中的函数和变量变得可供Postgres进程中的函数使用。Postgres使用操作系统提供的动态装载机制来完成这件事。
- 装载和链接编辑（Loading and link editing）是你对一个目标文件所做的事情，目的是产生另一种目标文件（例如一个可执行程序或一个共享库）。你使用链接编辑程序 ld(1) 来完成这件事。

下列一般性限制和说明也适用于下面的讨论：

- 传给 create function 命令的路径必须是绝对路径（即以“/”开头），并且指向运行Postgres服务器的机器上可见的目录。

> **提示**
>
> 相对路径实际上也能工作，
> 但它是相对于数据库所在目录的（前端应用通常看不到这个目录）。显然，
> 让路径相对于用户启动前端应用的目录是没有意义的，
> 因为服务器可能运行在一台完全不同的机器上！

- Postgres用户必须能够遍历传给 create function 命令的路径，并且能够读取该目标文件。这是因为Postgres服务器以Postgres用户身份运行，而不是以启动前端进程的用户身份运行。（让文件或上层目录对“postgres”用户不可读和/或不可执行是一个极其常见的错误。）
- 目标文件内定义的符号名彼此不能冲突，也不能与Postgres中定义的符号冲突。
- GNU C 编译器通常不提供使用操作系统动态装载器接口所需的特殊选项。在这种情况下，必须使用操作系统自带的 C 编译器。

46.1. Linux

在 Linux ELF 下，通过指定编译器标志 -fpic 就可以生成目标文件。

例如，

```
# simple Linux example
% cc -fpic -c foo.c
```

会产生一个名为*foo.o*的目标文件，随后可以把它动态装载进Postgres。
不需要再做额外的装载或链接编辑。

46.2. DEC OSF/1

在 DEC OSF/1 下，你可以取任意简单的目标文件，用带正确选项的 ld 命令处理它来产生一个共享目标文件。相应的命令看起来像这样：

```
# simple DEC OSF/1 example
% cc -c foo.c
% ld -shared -expect_unresolved '*' -o foo.so foo.o
```

得到的共享目标文件随后就可以装载进Postgres。在向 create function 命令指定目标文件名时，必须给出共享目标文件的名字（以 .so 结尾），而不是简单的目标文件。

> **提示**
>
> 实际上，只要是一个共享目标文件，Postgres并不关心你把它命名成什么。
> 如果你更喜欢用扩展名 .o 来命名共享目标文件，这对Postgres也没问题，只要确保把正确的文件名给了 create function 命令。换句话说，你只需保持一致。不过，从实用的角度看，我们不鼓励这种做法，因为你无疑会搞混哪些文件已经被做成了共享目标文件、哪些没有。例如，如果目标文件和共享目标文件都以 .o 结尾，就很难编写 Makefile 来自动完成链接编辑！

如果你指定的文件不是共享目标文件，后端将会挂起！

46.3. SunOS 4.x、Solaris 2.x 和 HP-UX

在 SunOS 4.x、Solaris 2.x 和 HP-UX 下，必须用特殊的编译器标志编译源文件来创建简单的目标文件，并且必须产生一个共享库。HP-UX 所需的步骤如下。HP-UX C 编译器的 +z 标志产生位置无关代码（PIC），+u 标志移除 PA-RISC 体系结构通常会强制施加的某些对齐限制。必须使用 HP-UX 链接编辑器并带 -b 选项把目标文件变成共享库。这听起来复杂，其实非常简单，因为相应的命令就是：

```
# simple HP-UX example
% cc +z +u -c foo.c
% ld -b -o foo.sl foo.o
```

与上一小节提到的 .so 文件一样，必须告诉 create function 命令哪个是要装载的正确文件（即必须给出共享库即 .sl 文件的位置）。在 SunOS 4.x 下，命令看起来像这样：

```
# simple SunOS 4.x example
% cc -PIC -c foo.c
% ld -dc -dp -Bdynamic -o foo.so foo.o
```

在 Solaris 2.x 下等价的命令行是：

```
# simple Solaris 2.x example
```

```
% cc -K PIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

或者

```
# simple Solaris 2.x example
% gcc -fPIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

链接共享库时，你可能必须在 ld 命令行上指定一些额外的共享库（通常是系统库，例如 C 库和数学库）。

第 47 章 触发器

Postgres 有多种服务器端函数接口。服务器端函数可以用 SQL、PLPGSQL、TCL 或 C 编写。除 SQL 外，触发器函数可以用这些语言中的任何一种编写。注意，当前版本不支持语句（STATEMENT）级触发器事件。目前可以把 BEFORE 或 AFTER 指定在一个元组的 INSERT、DELETE 或 UPDATE 上作为触发器事件。

47.1. 触发器创建

一旦触发器事件发生，触发器管理器（由执行器调用）就会初始化全局结构 TriggerData *CurrentTriggerData（下文描述），并调用触发器函数来处理该事件。

必须先创建触发器函数，然后才能创建触发器，它声明为不接受参数并返回 opaque 的函数。

创建触发器的语法如下：

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE | UPDATE [
 OR ... ] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure
     (args);
```

其中各个参数为：

trigger

触发器的名称在你将来需要删除该触发器时使用。它被用作 DROP TRIGGER 命令的一个参数。

BEFORE
AFTER

决定函数是在事件之前还是之后被调用。

INSERT
DELETE
UPDATE

命令的下一个元素决定哪些事件会触发该函数。可以用 OR 分隔指定多个事件。

relation

关系名指出该事件作用于哪个表。

ROW
STATEMENT

FOR EACH 子句决定触发器是针对每个受影响的行触发，还是在整条语句完成之前（或之后）触发。目前只支持 ROW 的情形。

procedure

过程名是要被调用的函数。

args

通过 CurrentTriggerData 结构传给函数的参数。向函数传递参数的目的，是为了让需求相似的不同触发器能够调用同一个函数。

另外，*procedure* 也可以用于触发不同的关系（这类函数被称为 "通用触发器函数"）。

作为同时使用上述两个特性的例子，可以有一个通用函数，它接受两个列名作为参数，把当前用户写入其中一个列，把当前时间戳写入另一个列。这样就可以在 INSERT 事件上编写触发器，例如自动跟踪某个事务表中记录的创建。如果用在 UPDATE 事件上，它还可以作为一个"最近更新"函数使用。

触发器函数向调用它的执行器返回 HeapTuple。对于在 INSERT、DELETE 或 UPDATE 操作之后触发的触发器，该返回值会被忽略，但它允许 BEFORE 触发器：

- 返回 NULL，以跳过对当前元组的操作（这样该元组就不会被插入/更新/删除）。
- 返回一个指向另一个元组的指针（仅 INSERT 和 UPDATE），该元组将被插入（如果是 UPDATE，则作为被更新元组的新版本），以取代原来的元组。

注意，CREATE TRIGGER 处理器不执行任何初始化。这一点将来会改变。另外，如果为同一关系上的同一事件定义了多个触发器，触发器的触发顺序是不可预测的。这一点将来也可能会改变。

如果触发器函数执行 SQL 查询（使用 SPI），那么这些查询可能会再次触发触发器。这就是所谓的级联触发器。对级联层数没有明确的限制。

如果一个触发器由 INSERT 触发，而又向同一个关系插入了一个新元组，那么该触发器会再次被触发。目前，对于这类情形没有提供任何同步（等）机制，但这一点将来可能改变。目前，回归测试中的函数 funny_dup17() 使用了一些技巧来停止对自身的递归（级联）……

47.2. 与触发器管理器的交互

如上所述，当函数被触发器管理器调用时，结构 TriggerData *CurrentTriggerData 非 NULL 且已初始化。因此，最好在开始时检查 CurrentTriggerData 是否为 NULL，并在取回信息之后立即把它设为 NULL，以防止并非来自触发器管理器的对触发器函数的调用。

struct TriggerData 定义在 src/include/commands/trigger.h 中：

```
typedef struct TriggerData
{
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
} TriggerData;
```

其成员的含义如下：

tg_event

描述引发该函数调用的事件。你可以使用下列宏来检查 `tg_event`：

TRIGGER_FIRED_BEFORE(tg_event)

如果触发器在之前（BEFORE）触发则返回 TRUE。

TRIGGER_FIRED_AFTER(tg_event)

如果触发器在之后（AFTER）触发则返回 TRUE。

TRIGGER_FIRED_FOR_ROW(event)

如果触发器针对行（ROW）级事件触发则返回 TRUE。

TRIGGER_FIRED_FOR_STATEMENT(event)

如果触发器针对语句（STATEMENT）级事件触发则返回 TRUE。

TRIGGER_FIRED_BY_INSERT(event)

如果触发器由 INSERT 引发则返回 TRUE。

TRIGGER_FIRED_BY_DELETE(event)

如果触发器由 DELETE 引发则返回 TRUE。

TRIGGER_FIRED_BY_UPDATE(event)

如果触发器由 UPDATE 引发则返回 TRUE。

tg_relation

一个指针，指向描述被触发关系的结构体。关于这个结构体的细节请参看 src/include/utils/rel.h。其中最令人感兴趣的是 tg_relation->rd_att（关系元组的描述符）和 tg_relation->rd_rel->relname（关系名。它不是 char*，而是 NameData。如果你需要名称的一个副本，可以使用 SPI_getrelname(tg_relation) 来获得 char*）。

tg_trigtuple

一个指针，指向引发该触发器的元组。这就是正在被插入（INSERT）、删除（DELETE）或更新（UPDATE）的元组。如果是 INSERT/DELETE，那么当你不想（INSERT 的情形下）用另一个元组替换该元组或跳过该操作时，就应当把它返回给执行器。

tg_newtuple

一个指针，指向元组的新版本（如果 UPDATE）；如果是 INSERT 或 DELETE 则为 NULL。当事件是 UPDATE 而你不想用另一个元组替换该元组或跳过该操作时，就应当把它返回给执行器。

tg_trigger

一个指针，指向 Trigger 结构，它定义在 src/include/utils/rel.h 中：

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    FmgrInfo    tgfunc;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    bool        tgdeferrable;
```

```
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

其中 tgname 是触发器的名称，tgnargs 是 tgargs 中参数的数量，tgargs 是一个指针数组，指向 CREATE TRIGGER 语句中指定的参数。其他成员仅供内部使用。

47.3. 数据更改的可见性

Postgres 的数据更改可见性规则：在一次查询执行期间，该查询自身（通过 SQL 函数、SPI 函数、触发器）所做的数据更改对查询扫描是不可见的。例如，在查询

```
INSERT INTO a SELECT * FROM a;
```

中，被插入的元组对 SELECT 扫描不可见。其效果是在数据库表内部复制该表自身（当然要受唯一索引规则的约束），而不会递归。

但请记住 SPI 文档中关于可见性的这段说明：

> 查询 Q 所做的更改，对所有在查询 Q 之后启动的查询都可见，无论这些查询是在 Q 内部（即 Q 执行期间）启动，还是在 Q 完成之后启动。

这对触发器同样成立，因此，虽然正在被插入的元组（tg_trigtuple）对 BEFORE 触发器中的查询不可见，但这个元组（刚刚插入的）对 AFTER 触发器中的查询可见，也对在此之后触发的 BEFORE/AFTER 触发器中的查询可见！

47.4. 示例

更多更复杂的示例见 `src/test/regress/regress.c` 和 `contrib/spi`。

下面是一个非常简单的触发器用法示例。函数 trigf 报告被触发关系 ttest 中的元组数，并且当查询试图向 x 插入 NULL 时跳过该操作（也就是说，它起到一个 NOT NULL 约束的作用，但不会中止事务）。

```
#include "executor/spi.h" /* this is what you need to work with SPI */
#include "commands/trigger.h" /* -"- and triggers */

HeapTuple  trigf(void);

HeapTuple
trigf()
{
 TupleDesc tupdesc;
 HeapTuple rettuple;
 char  *when;
 bool  checknull = false;
 bool  isnull;
 int  ret, i;

 if (!CurrentTriggerData)
  elog(NOTICE, "trigf: triggers are not initialized");
```

```
 /* tuple to return to Executor */
 if (TRIGGER_FIRED_BY_UPDATE(CurrentTriggerData->tg_event))
  rettuple = CurrentTriggerData->tg_newtuple;
 else
  rettuple = CurrentTriggerData->tg_trigtuple;

 /* check for NULLs ? */
 if (!TRIGGER_FIRED_BY_DELETE(CurrentTriggerData->tg_event) &&
  TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  checknull = true;

 if (TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  when = "before";
 else
  when = "after ";

 tupdesc = CurrentTriggerData->tg_relation->rd_att;
 CurrentTriggerData = NULL;

 /* Connect to SPI manager */
 if ((ret = SPI_connect()) < 0)
  elog(NOTICE, "trigf (fired %s): SPI_connect returned %d", when, ret)
;

 /* Get number of tuples in relation */
 ret = SPI_exec("select count(*) from ttest", 0);

 if (ret < 0)
  elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when, ret);

 i = SPI_getbinval(SPI_tuptable->vals[0], SPI_tuptable->tupdesc, 1,
 &isnull);

 elog (NOTICE, "trigf (fired %s): there are %d tuples in ttest", when,
 i);

 SPI_finish();

 if (checknull)
 {
  i = SPI_getbinval(rettuple, tupdesc, 1, &isnull);
  if (isnull)
   rettuple = NULL;
 }

 return (rettuple);
}
```

现在，编译并创建表 ttest (x int4):

```
create function trigf () returns opaque as
'...path_to_so' language 'c';
```

```
vac=> create trigger tbefore before insert or update or delete on
 ttest
for each row execute procedure trigf();
CREATE
vac=> create trigger tafter after insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> insert into ttest values (null);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

vac=> select * from ttest;
x
-
(0 rows)

vac=> insert into ttest values (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> select * from ttest;
x
-
1
(1 row)

vac=> insert into ttest select x * 2 from ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
vac=> select * from ttest;
x
-
1
2
(2 rows)

vac=> update ttest set x = null where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> update ttest set x = 4 where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> select * from ttest;
x
-
1
```

```
4
(2 rows)

vac=> delete from ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                        ^^^^^^^^
                              remember what we said about visibility.
DELETE 2
vac=> select * from ttest;
x
-
(0 rows)
```

第 48 章 服务器编程接口

Vadim Mikheev

服务器编程接口（SPI）使用户能够在用户定义的 C 函数中运行 SQL 查询。可用的过程语言（PL）提供了访问这些能力的另一种方法。

实际上，SPI 只是一组简化对解析器、规划器、优化器和执行器访问的本地接口函数。SPI 还做一部分内存管理工作。

为避免误解，我们用函数指 SPI 接口函数，而用过程指使用 SPI 的用户定义 C 函数。

SPI 过程总是由某个（上层）执行器调用，而 SPI 管理器使用执行器来运行你的查询。执行器在运行来自你的过程的查询时，还可能调用其他过程。

注意，如果在执行来自过程的查询期间事务被中止，控制将不会返回到你的过程。相反，所有工作都会被回滚，服务器会等待客户端的下一条命令。这一点在未来的版本中会被改变。

其他限制是无法执行 BEGIN、END 和 ABORT（事务控制语句）以及游标操作。这一点将来也会被改变。

成功时，SPI 函数返回非负结果（或者通过整型返回值返回，或者如下文所述存放在全局变量 SPI_result 中）。出错时，将返回负值或 NULL。

48.1. 接口函数

SPI_connect

SPI_connect — 将你的过程连接到 SPI 管理器。

大纲

```
int SPI_connect(void)
```

输入

无

输出

int

返回状态

→ SPI_OK_CONNECT

已连接时

→ SPI_ERROR_CONNECT

未连接时

描述

SPI_connect 打开到 Postgres 后端的连接。如果需要执行查询，就应调用此函数。有些 SPI 辅助函数可以在未连接的过程中调用。

如果 SPI_connect 是从已连接的过程中调用的，就可能得到 → SPI_ERROR_CONNECT 错误 —— 例如，你从一个已连接的过程直接调用另一个过程。实际上，虽然子过程将能够使用 SPI，但在子过程返回之后（如果子过程调用了 SPI_finish），你的父过程将无法继续使用 SPI。这是一种不好的做法。

用法

算法

SPI_connect 执行以下操作：

- 初始化用于查询执行和内存管理的 SPI 内部结构。

SPI_finish

SPI_finish — 将你的过程与 SPI 管理器断开连接。

大纲

```
SPI_finish(void)
```

输入

无

输出

int

→ SPI_OK_FINISH 已正确断开连接时

→ SPI_ERROR_UNCONNECTED 从未连接的过程中调用时

描述

SPI_finish 关闭到 Postgres 后端的现有连接。在完成通过 SPI 管理器所做的操作之后，就应调用此函数。

如果在当前没有有效连接的情况下调用 SPI_finish，可能会得到错误返回 → SPI_ERROR_UNCONNECTED。这并不是什么根本性的问题；它只表示 SPI 管理器没有做任何事情。

用法

已连接的过程必须以调用 SPI_finish 作为最后一步，否则可能得到不可预测的结果！注意，如果你中止了事务（通过 elog(ERROR)），则可以安全地跳过对 SPI_finish 的调用。

算法

SPI_finish 执行以下操作：

- 将你的过程与 SPI 管理器断开连接，并释放该过程自 SPI_connect 以来通过 palloc 所做的全部内存分配。这些分配不能再使用了！参见内存管理。

SPI_exec

SPI_exec — 创建一个执行计划（解析器+规划器+优化器）并执行查询。

大纲

```
SPI_exec(query, tcount)
```

输入

char *query

包含查询计划的字符串

int tcount

要返回的最大元组数

输出

int

→ SPI_OK_EXEC 已正确断开连接时
→ SPI_ERROR_UNCONNECTED 从未连接的过程中调用时
→ SPI_ERROR_ARGUMENT query 为 NULL 或 *tcount* < 0 时。
→ SPI_ERROR_UNCONNECTED 过程未连接时。
→ SPI_ERROR_COPY 执行 COPY TO/FROM stdin 时。
→ SPI_ERROR_CURSOR 执行 DECLARE/CLOSE CURSOR、FETCH 时。
→ SPI_ERROR_TRANSACTION 执行 BEGIN/ABORT/END 时。
→ SPI_ERROR_OPUNKNOWN 查询类型未知时（这不应发生）。

如果查询执行成功，则会返回下列（非负）值之一：

→ SPI_OK_UTILITY 执行了某个工具命令（例如 CREATE TABLE ...）时
→ SPI_OK_SELECT 执行了 SELECT（但不是 SELECT ... INTO!）时
→ SPI_OK_SELINTO 执行了 SELECT ... INTO 时
→ SPI_OK_INSERT 执行了 INSERT（或 INSERT ... SELECT）时
→ SPI_OK_DELETE 执行了 DELETE 时
→ SPI_OK_UPDATE 执行了 UPDATE 时

描述

SPI_exec 创建一个执行计划（解析器+规划器+优化器），并针对 *tcount* 个元组执行该查询。

用法

此函数只应从已连接的过程中调用。如果 *tcount* 为零，则会对查询扫描返回的所有元组执行该查询。使用 *tcount* > 0 可以限制该查询将要执行的元组数。例如，

```
SPI_exec ("insert into table select * from table", 5);
```

最多只允许向表中插入 5 个元组。如果查询执行成功，则会返回一个非负值。

> **注意**
>
> 你可以在一个字符串中传递多条查询，查询字符串也可能被 RULE 改写。SPI_exec 返回最后执行的那条查询的结果。

（最后一条）查询实际执行所处理的元组数，会通过全局变量 SPI_processed 返回（返回值不是 → SPI_OK_UTILITY 时）。如果返回了 → SPI_OK_SELECT 且 SPI_processed > 0，则可以使用全局指针 SPITupleTable *SPI_tuptable 访问选出的元组：另外注意，SPI_finish 会释放所有 SPITupleTable 并使它们不可再用！（参见内存管理）。

SPI_exec 可能返回下列（负）值之一：

→ SPI_ERROR_ARGUMENT query 为 NULL 或 *tcount* < 0 时。

→ SPI_ERROR_UNCONNECTED 过程未连接时。

→ SPI_ERROR_COPY 执行 COPY TO/FROM stdin 时。

→ SPI_ERROR_CURSOR 执行 DECLARE/CLOSE CURSOR、FETCH 时。

→ SPI_ERROR_TRANSACTION 执行 BEGIN/ABORT/END 时。

→ SPI_ERROR_OPUNKNOWN 查询类型未知时（这不应发生）。

算法

SPI_exec 执行以下操作：

- 将你的过程与 SPI 管理器断开连接，并释放该过程自 SPI_connect 以来通过 palloc 所做的全部内存分配。这些分配不能再使用了！参见内存管理。

SPI_prepare

SPI_prepare — 将你的过程连接到 SPI 管理器。

大纲

```
SPI_prepare(query, nargs, argtypes)
```

输入

query

查询字符串

nargs

输入参数的数量（$1 ... $nargs —— 与 SQL 函数中相同）

argtypes

指向输入参数的类型 OID 列表的指针

输出

void *

指向执行计划（解析器+规划器+优化器）的指针

描述

`SPI_prepare` 创建并返回一个执行计划（解析器+规划器+优化器），但不执行该查询。只应从已连接的过程中调用。

用法

nargs 是参数的数量（$1 ... $nargs —— 与 SQL 函数中相同），只有当查询中没有任何 $1 时，nargs 才可以为 0。

预备执行计划的执行有时会快得多，因此当同一条查询要执行很多次时，此特性可能会很有用。

`SPI_prepare` 返回的计划只能在当前这次过程调用中使用，因为 `SPI_finish` 会释放为计划分配的内存。参见 `SPI_saveplan`。

成功时会返回一个非空指针。否则，将得到 NULL 计划。无论成功还是出错，SPI_result 都会被设成与 SPI_exec 返回值类似的值；但如果 query 为 NULL，或 nargs < 0，或 nargs > 0 且 argtypes 为 NULL，则会将其设为 → `SPI_ERROR_ARGUMENT`。

SPI_saveplan

SPI_saveplan — 保存传入的计划

大纲

```
SPI_saveplan(plan)
```

输入

void **query*

传入的计划

输出

void *

执行计划的位置。不成功时为 NULL。

SPI_result

→ `SPI_ERROR_ARGUMENT` plan 为 NULL 时
→ `SPI_ERROR_UNCONNECTED` 过程未连接时

描述

`SPI_saveplan` 把由 `SPI_prepare` 准备的计划存储在安全内存中，使其不会被 `SPI_finish` 或事务管理器释放。

在当前版本的 Postgres 中，还无法把预备计划存储在系统目录中并在执行时从那里取出。这一点将在未来的版本中实现。作为替代，可以在当前会话中该过程的后续调用里复用预备计划。使用 `SPI_execp` 执行这个保存的计划。

用法

`SPI_saveplan` 把传入的计划（由 `SPI_prepare` 准备）保存在受保护的内存中，使其不会被 `SPI_finish` 和事务管理器释放，并返回指向该已保存计划的指针。可以把返回的指针保存在局部变量中。无论是准备计划时，还是在 SPI_execp 中使用已准备好的计划时（见下文），都请始终检查该指针是否为 NULL。

> **注意**
>
> 如果预备计划所引用的对象（关系、函数等）
> 在会话期间被删除（由你的后端或另一个进程删除），那么对该计划执行 `SPI_execp` 的结果将是不可预知的。

SPI_execp

SPI_execp — 执行来自 SPI_saveplan 的计划

大纲

```
SPI_execp(plan,
values,
nulls,
tcount)
```

输入

void *plan

执行计划

Datum *values

实际参数值

char *nulls

描述哪些参数取 NULL 的数组

'n' 表示允许 NULL
' ' 表示不允许 NULL

int tcount

该计划将要执行的元组数

输出

int

返回与 SPI_exec 相同的值，以及

→ SPI_ERROR_ARGUMENT*plan* 为 NULL 或 *tcount* < 0 时
→ SPI_ERROR_PARAM*values* 为 NULL，而 *plan* 在准备时带有参数时。

SPI_tuptable

成功时，其初始化方式与 SPI_exec 中相同

SPI_processed

成功时，其初始化方式与 SPI_exec 中相同

描述

SPI_execp 把由 SPI_prepare 准备的计划存储在安全内存中，使其不会被 SPI_finish 或事务管理器释放。

在当前版本的 Postgres 中，还无法把预备计划存储在系统目录中并在执行时从那里取出。这一点将在未来的版本中实现。作为权宜之计，可以在当前会话中该过程的后续调用里复用预备计划。使用 `SPI_execp` 执行这个保存的计划。

用法

如果 *`nulls`* 为 NULL，则 `SPI_execp` 假定所有值（如果有的话）都非 NULL。

> **注意**
>
> 如果预备计划所引用的对象（关系、函数等）
> 在会话期间被删除（由你的后端或另一个进程删除），那么对该计划执行 `SPI_execp` 的结果将是不可预知的。

48.2. 接口支持函数

下面介绍的所有函数既可由已连接的过程使用，也可由未连接的过程使用。

SPI_copytuple

SPI_copytuple — 在上层执行器上下文中创建元组的副本

大纲

```
SPI_copytuple(tuple)
```

输入

HeapTuple *tuple*

要复制的输入元组

输出

HeapTuple

复制得到的元组

→ 非 NULL*tuple* 不为 NULL 且复制成功时
→ NULL 仅当 *tuple* 为 NULL 时

描述

SPI_copytuple 在上层执行器上下文中创建元组的一个副本。参见内存管理一节。

用法

TBD

SPI_modifytuple

SPI_modifytuple — 修改关系的元组

大纲

```
SPI_modifytuple(rel, tuple , nattrs
, attnum , Values , Nulls)
```

输入

Relation *rel*

HeapTuple *tuple*

要修改的输入元组

int *nattrs*

attnum 中属性编号的数量

int * *attnum*

要被修改的属性的编号组成的数组

Datum * *Values*

指定属性的新值

char * *Nulls*

哪些属性为 NULL（如果有的话）

输出

HeapTuple

带修改内容的新元组

→ 非 NULL*tuple* 不为 NULL 且修改成功时
→ NULL 仅当 *tuple* 为 NULL 时

SPI_result

→ SPI_ERROR_ARGUMENT rel 为 NULL，或 tuple 为 NULL，或 natts ≤ 0，或 attnum 为 NULL，或 Values 为 NULL 时。
→ SPI_ERROR_NOATTRIBUTE attnum 中存在无效的属性编号时（attnum ≤ 0，或大于元组中的属性数量）

描述

SPI_modifytuple 在上层执行器上下文中修改元组。参见内存管理一节。

用法

成功时，返回指向新元组的指针。新元组在上层执行器上下文中分配（参见内存管理）。
传入的元组不会被修改。

SPI_fnumber

SPI_fnumber — 查找指定属性的属性编号

大纲

```
SPI_fnumber(tupdesc, fname)
```

输入

TupleDesc *tupdesc*

输入元组描述

char * *fname*

字段名

输出

int

属性编号

属性的有效基于 1 的索引编号
→ `SPI_ERROR_NOATTRIBUTE` 找不到指定名称的属性时

描述

`SPI_fnumber` 返回 fname 中指定名称的属性的属性编号。

用法

属性编号从 1 开始。

SPI_fname

SPI_fname — 查找指定属性的属性名称

大纲

```
SPI_fname(tupdesc, fname)
```

输入

TupleDesc *tupdesc*

输入元组描述

char * *fnumber*

属性编号

输出

char *

属性名称

fnumber 超出范围时为 NULL
出错时 SPI_result 被设置为 → `SPI_ERROR_NOATTRIBUTE`

描述

`SPI_fname` 返回指定属性的属性名称。

用法

属性编号从 1 开始。

算法

返回一个新分配的属性名称副本。

SPI_getvalue

SPI_getvalue — 返回指定属性的字符串值

大纲

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

输入

HeapTuple *tuple*

要检查的输入元组

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

char *

属性值，以下情况返回 NULL：

属性为 NULL
fnumber 超出范围（SPI_result 被设置为 → SPI_ERROR_NOATTRIBUTE）
没有可用的输出函数（SPI_result 被设置为 → SPI_ERROR_NOOUTFUNC）

描述

SPI_getvalue 返回指定属性值的外部（字符串）表示形式。

用法

属性编号从 1 开始。

算法

根据该值的需要分配内存。

SPI_getbinval

SPI_getbinval — 返回指定属性的二进制值

大纲

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

输入

HeapTuple *tuple*

要检查的输入元组

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

Datum

属性的二元值

bool * *isnull*

属性值是否为空的标志

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

SPI_getbinval 返回指定属性的二进制值。

用法

属性编号从 1 开始。

算法

不为该二进制值分配新的空间。

SPI_gettype

SPI_gettype — 返回指定属性的类型名称

大纲

```
SPI_gettype(tupdesc, fnumber)
```

输入

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

char *

指定属性编号的类型名称

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

SPI_gettype 返回指定属性类型名称的一个副本。

用法

属性编号从 1 开始。

算法

不为该二进制值分配新的空间。

SPI_gettypeid

SPI_gettypeid — 返回指定属性的类型 OID

大纲

```
SPI_gettypeid(tupdesc, fnumber)
```

输入

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

OID

指定属性编号的类型 OID

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

SPI_gettypeid 返回指定属性的类型 OID。

用法

属性编号从 1 开始。

算法

TBD

SPI_getrelname

SPI_getrelname — 返回指定关系的名称

大纲

```
SPI_getrelname(rel)
```

输入

Relation *rel*

输入关系

输出

char *

指定关系的名称

描述

`SPI_getrelname` 返回指定关系的名称。

用法

TBD

算法

把该关系名称复制到新的存储中。

SPI_palloc

SPI_palloc — 在上层执行器上下文中分配内存

大纲

```
SPI_palloc(size)
```

输入

Size *size*

要分配的存储空间的字节大小

输出

void *

指定大小的新存储空间

描述

`SPI_palloc` 在上层执行器上下文中分配内存。参见内存管理一节。

用法

TBD

SPI_repalloc

SPI_repalloc — 在上层执行器上下文中重新分配内存

大纲

```
SPI_repalloc(pointer, size)
```

输入

void * *pointer*

指向现有存储的指针

Size *size*

要分配的存储空间的字节大小

输出

void *

指定大小的新存储空间，内容从现有区域复制而来

描述

`SPI_repalloc` 在上层执行器上下文中重新分配内存。参见内存管理一节。

用法

TBD

SPI_pfree

SPI_pfree — 从上层执行器上下文中释放内存

大纲

```
SPI_pfree(pointer)
```

输入

void * *pointer*

指向现有存储的指针

输出

无

描述

`SPI_pfree` 在上层执行器上下文中释放内存。参见内存管理一节。

用法

TBD

48.3. 内存管理

服务器在内存上下文中分配内存，
使得在一个上下文中做出的分配可以通过销毁该上下文来释放，
而不影响在其他上下文中做出的分配。所有分配（通过 `palloc` 等）都是在被选为当前上下文的那个上下文中进行的。如果试图释放（或重新分配）不是在当前上下文中分配的内存，
将会得到不可预知的结果。

内存上下文的创建和切换属于 SPI 管理器内存管理的内容。

SPI 过程涉及两个内存上下文：上层执行器内存上下文和过程内存上下文（如果已连接）。

在过程连接到 SPI 管理器之前，当前内存上下文是上层执行器上下文，因此该过程自身在连接到 SPI 之前通过 `palloc`/`repalloc` 或由 SPI 辅助函数所做的所有分配都位于这个上下文中。

调用 `SPI_connect` 之后，当前上下文是过程的私有上下文。通过 `palloc`/`repalloc` 或由 SPI 辅助函数所做的所有分配（`SPI_copytuple`、`SPI_modifytuple`、`SPI_palloc` 和 `SPI_repalloc` 除外）都位于这个上下文中。

当过程（通过 `SPI_finish`）与 SPI 管理器断开连接时，当前上下文会恢复为上层执行器上下文，而在该过程内存上下文中分配的所有内存都会被释放，不能再使用！

如果想把某些东西返回给上层执行器，就必须在上层上下文中为它分配内存！

SPI 无法自动释放上层执行器上下文中的分配！

查询执行期间分配的内存会在该查询完成时被 SPI 自动释放！

48.4. 数据更改的可见性

Postgres 的数据更改可见性规则：在一次查询执行期间，该查询自身（通过 SQL 函数、SPI 函数、触发器）所做的数据更改对查询扫描是不可见的。例如，在查询 INSERT INTO a SELECT * FROM a 中，被插入的元组对 SELECT 的扫描不可见。其效果是在数据库表内部复制该表自身（当然要受唯一索引规则的约束），而不会递归。

查询 Q 所做的更改，对所有在查询 Q 之后启动的查询都可见，无论这些查询是在 Q 内部（即 Q 执行期间）启动，还是在 Q 完成之后启动。

48.5. 示例

这个 SPI 用法的示例演示了可见性规则。在 src/test/regress/regress.c 和 contrib/spi 中还有更复杂的示例。

这是一个非常简单的 SPI 用法示例。过程 execq 的第一个参数接受一条 SQL 查询，第二个参数接受 tcount，它使用 SPI_exec 执行该查询，并返回该查询所处理的元组数：

```
#include "executor/spi.h" /* this is what you need to work with SPI */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
 int ret;
 int proc = 0;

 SPI_connect();

 ret = SPI_exec(textout(sql), cnt);

 proc = SPI_processed;
 /*
  * If this is SELECT and some tuple(s) fetched -
  * returns tuples to the caller via elog (NOTICE).
  */
 if ( ret == SPI_OK_SELECT && SPI_processed > 0 )
 {
  TupleDesc tupdesc = SPI_tuptable->tupdesc;
  SPITupleTable *tuptable = SPI_tuptable;
  char buf[8192];
  int i;

  for (ret = 0; ret < proc; ret++)
  {
   HeapTuple tuple = tuptable->vals[ret];

   for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
    sprintf(buf + strlen (buf), " %s%s",
     SPI_getvalue(tuple, tupdesc, i),
     (i == tupdesc->natts) ? " " : " |");
   elog (NOTICE, "EXECQ: %s", buf);
```

```
  }
 }

 SPI_finish();

 return (proc);
}
```

现在，编译并创建该函数：

```
create function execq (text, int4) returns int4 as '...path_to_so'
 language 'c';

vac=> select execq('create table a (x int4)', 0);
execq
-----
    0
(1 row)

vac=> insert into a values (execq('insert into a values (0)',0));
INSERT 167631 1
vac=> select execq('select * from a',0);
NOTICE:EXECQ:  0 <<< 由 execq 插入

NOTICE:EXECQ:  1 <<< 由 execq 返回并被外层 INSERT 插入的值

execq
-----
    2
(1 row)

vac=> select execq('insert into a select x + 2 from a',1);
execq
-----
    1
(1 row)

vac=> select execq('select * from a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 <<< 0 + 2，只插入了一个元组 —— 正如所指定的那样

execq
-----
    3            <<< 10 只是最大值，3 才是实际的元组数
(1 row)

vac=> delete from a;
DELETE 3
vac=> insert into a values (execq('select * from a', 0) + 1);
INSERT 167712 1
vac=> select * from a;
```

```
x
-
1                <<< a 中没有元组 (0) + 1
(1 row)

vac=> insert into a values (execq('select * from a', 0) + 1);
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> select * from a;
x
-
1
2                <<< a 中原先只有一个元组 + 1
(2 rows)

--    这演示了数据更改可见性规则:

vac=> insert into a select execq('select * from a', 0) * x from a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> select * from a;
x
-
1
2
2                <<< 2 个元组 * 1 (第一个元组中的 x)
6                <<< 3 个元组 (刚插入的 2 + 1) * 2 (第二个元组中的 x)
(4 rows)             ^^^^^^^^
                     execq() 在不同调用中可见的元组
```

第 49 章 过程语言

Postgres 支持定义过程语言。对于用过程语言定义的函数或触发器过程，数据库并不内置关于如何解释函数源文本的知识。相反，这项任务会交给一个了解该语言细节的调用处理器。
调用处理器本身是一个特殊的编程语言函数，被编译进共享对象并按需装载。

编写新过程语言（PL）的调用处理器超出了本手册的范围。

49.1. 安装过程语言

过程语言的安装

在数据库中安装过程语言需要三个步骤。（对于标准发行版自带的语言，可以改用 shell 脚本 `createlang` 来代替手工完成各个细节。）

1. 该语言调用处理器的共享对象必须编译并安装。默认情况下，PL/pgSQL 的调用处理器会被构建并安装到数据库库目录中。如果配置了 Tcl/Tk 支持，PL/Tcl 的调用处理器也会被构建并安装到同一位置。

2. 必须用如下命令声明该调用处理器：

   ```
   CREATE FUNCTION handler_function_name ()
       RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE 'C';
   ```

   特殊返回类型OPAQUE会告诉数据库系统，该函数返回的不是某种已定义的SQL数据类型，并且不能在 SQL语句中直接使用。

3. 该 PL 必须用如下命令声明：

   ```
   CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'language-name'
       HANDLER handler_function_name
       LANCOMPILER 'description';
   ```

   可选关键字 TRUSTED 指明是否允许没有超级用户权限的普通数据库用户使用该语言创建函数和触发器过程。由于 PL 函数是在数据库后端内部执行的，TRUSTED 标记只应赋予那些不允许访问数据库后端内部或文件系统的语言。语言 PL/pgSQL 和 PL/Tcl 已知是受信任的。

示例

1. 下面的命令告诉数据库到哪里查找 PL/pgSQL 语言调用处理器函数的共享对象：

   ```
   CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
       '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
   ```

2. 而命令

   ```
   CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
       HANDLER plpgsql_call_handler
       LANCOMPILER 'PL/pgSQL';
   ```

则定义对于语言属性为 'plpgsql' 的函数和触发器过程，将调用先前声明的调用处理器函数。

PL 调用处理器函数有一个与普通 C 语言函数不同的特殊调用接口。传给调用处理器的参数之一是要执行的函数在 `pg_proc` 表条目中的对象 ID。调用处理器会检查各种系统目录来分析函数的调用参数及其返回数据类型。函数主体的源文本位于 `pg_proc` 的 prosrc 属性中。因此，PL 函数可以像 SQL 语言函数一样被重载。只要调用参数不同，就可以有多个同名的不同 PL 函数。

在 `template1` 数据库中定义的过程语言会自动在随后创建的所有数据库中定义。因此数据库管理员可以决定默认提供哪些语言。

部分 IV. 接口

用户与程序员接口。

目录

第 50 章 函数

用户可调用函数的参考信息。

注意

本节有待撰写。欢迎志愿者！

第 51 章 大对象

在 Postgres 中，数据值存储在元组中，而单个元组不能跨数据页。由于数据页的大小是 8192 字节，数据值大小的上限相对较低。为了支持更大的原子值的存储，Postgres 提供了大对象接口。该接口以面向文件的方式访问被声明为大类型的用户数据。本节描述 Postgres 大对象数据的实现以及编程和查询语言接口。

51.1. 历史注记

最初，Postgres 4.2 支持三种标准的大对象实现：作为 Postgres 之外的文件、作为 Postgres 管理的外部文件、以及作为存储在 Postgres 数据库内部的数据。这给用户带来了相当大的困扰。因此，在 PostgreSQL 中我们只支持把大对象作为存储在 Postgres 数据库内部的数据。尽管访问起来更慢，它提供了更严格的数据完整性。由于历史原因，这种存储方案被称为 Inversion 大对象。（本节中我们将把 Inversion 和大对象作为同义词混用，指同一事物。）

51.2. 实现特性

Inversion 大对象的实现把大对象拆分成"块"并把块存储在数据库的元组中。一个 B-tree 索引保证在做随机访问读写时能快速搜索到正确的块号。

51.3. 接口

下面描述 Postgres 提供的访问大对象的设施，既包括后端中作为用户自定义函数一部分的用法，也包括前端中作为使用该接口的应用一部分的用法。对于熟悉 Postgres 4.2 的用户来说，PostgreSQL 有一套新函数，提供了更连贯的接口。

> **注意**
>
> 所有对大对象的操作必须在一个 SQL 事务中进行。这一要求从 Postgres v6.5 起被严格执行，尽管在之前的版本中它一直是隐含的要求，忽略它会导致错误行为。

Postgres 的大对象接口模仿 Unix 文件系统接口设计，有与 `open(2)`、`read(2)`、`write(2)`、`lseek(2)` 等对应的函数。用户函数调用这些例程来只从大对象中检索感兴趣的数据。例如，假设存在一个存储面孔照片的名为 mugshot 的大对象类型，那么可以在 mugshot 数据上声明一个名为 beard 的函数。Beard 可以查看照片的下三分之一部分，并确定那里出现的胡须颜色（如果有的话）。整个大对象的值不需要被 beard 函数缓冲，甚至不需要被检查。大对象可以从动态装载的 C 函数或链接了该库的数据库客户端程序访问。Postgres 提供了一组支持打开、读取、写入、关闭和在大对象上定位的例程。

51.3.1. 创建一个大对象

例程

```
Oid lo_creat(PGconn *conn, int mode)
```

创建一个新的大对象。*mode* 是一个位掩码，描述新大对象的若干不同属性。这里列出的符号常量定义在 `$PGROOT/src/backend/libpq/libpq-fs.h` 中。访问类型（读、写或读写）由把 INV_READ 和 INV_WRITE 位"或"在一起来控制。如果大对象应当被归档

——也就是说，如果它的历史版本应当被定期移到一个特殊的归档关系中——那么应当设置 INV_ARCHIVE 位。掩码的低十六位是大对象应驻留的存储管理器编号。对于伯克利以外的站点，这些位应当始终为零。下面的命令创建一个（Inversion）大对象：

```
inv_oid = lo_creat(INV_READ|INV_WRITE|INV_ARCHIVE);
```

51.3.2. 导入一个大对象

要把一个 UNIX 文件导入为大对象，调用

```
Oid lo_import(PGconn *conn, const char *filename)
```

filename 指定要导入为大对象的文件的 Unix 路径名。

51.3.3. 导出一个大对象

要把一个大对象导出到 UNIX 文件，调用

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename)
```

lobjId 参数指定要导出的大对象的 Oid，filename 参数指定文件的 UNIX 路径名。

51.3.4. 打开一个现有的大对象

要打开一个现有的大对象，调用

```
int lo_open(PGconn *conn, Oid lobjId, int mode)
```

lobjId 参数指定要打开的大对象的 Oid。mode 位控制对象是以读 INV_READ)、写还是读写方式打开。大对象在创建之前无法打开。`lo_open` 返回一个大对象描述符，供稍后在 `lo_read`、`lo_write`、`lo_lseek`、`lo_tell` 和 `lo_close` 中使用。

51.3.5. 向大对象写入数据

例程

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len)
```

把 len 字节从 buf 写入大对象 fd。fd 参数必须是先前某个 `lo_open` 返回的。返回值是实际写入的字节数。发生错误时，返回值为负。

51.3.6. 从大对象读取数据

例程

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len)
```

从大对象 fd 中读取 len 字节到 byf。fd 参数必须是先前某个 `lo_open` 返回的。返回值是实际读取的字节数。发生错误时，返回值为负。

51.3.7. 在大对象上定位

要更改大对象上当前的读或写位置，调用

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

此例程把 fd 所描述大对象的当前位置指针移动到 offset 指定的新位置。whence 的有效值是 SEEK_SET、SEEK_CUR 和 SEEK_END。

51.3.8. 关闭一个大对象描述符

可以通过调用下面的函数来关闭一个大对象：

```
int lo_close(PGconn *conn, int fd)
```

其中 fd 是由 `lo_open` 返回的大对象描述符。成功时 `lo_close` 返回零。发生错误时，返回值为负。

51.3.9. 移除一个大对象

要从数据库中移除一个大对象，调用

```
Oid lo_unlink(PGconn *conn, Oid lobjId)
```

lobjId 参数指定要删除的大对象的 Oid。

51.4. 内建已注册函数

有两个内置的已注册函数 lo_import 和 lo_export，它们便于在 SQL 查询中使用。下面是一个用法的例子：

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, '/tmp/motd') from image
    WHERE name = 'beautiful image';
```

51.5. 从 LIBPQ 访问大对象

下面是一个示例程序，展示了 LIBPQ 中大对象接口的用法。程序的一部分被注释掉了，但为了读者受益仍保留在源码中。该程序可以在 `../src/test/examples` 中找到。使用 LIBPQ 中大对象接口的前端应用应当包含头文件 libpq/libpq-fs.h 并链接 libpq 库。

51.6. 示例程序

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
```

```
 *
 *
 * IDENTIFICATION
 *    /usr/local/devel/pglite/cvs/src/doc/manual.me,v 1.16 1995/09/01
 23:55:00 jolly Exp
 *
 *-------------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile *    import file "in_filename" into database as large
 object "lobjOid"
 *
 */
Oid importFile(PGconn *conn, char *filename)
{
    Oid lobjId;
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)  {   /* error */
     fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ|INV_WRITE);
    if (lobjId == 0) {
     fprintf(stderr, "can't create large object\n");
    }

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0) {
     tmp = lo_write(conn, lobj_fd, buf, nbytes);
     if (tmp < nbytes) {
         fprintf(stderr, "error while reading large object\n");
     }
    }
```

```
    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
    int nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    nread = 0;
    while (len - nread > 0) {
     nbytes = lo_read(conn, lobj_fd, buf, len - nread);
     buf[nbytes] = ' ';
     fprintf(stderr,">>> %s", buf);
     nread += nbytes;
    }
    fprintf(stderr,"\n");
    lo_close(conn, lobj_fd);
}

void overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
    int nwritten;
    int i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    for (i=0;i<len;i++)
     buf[i] = 'X';
    buf[i] = ' ';
```

```
    nwritten = 0;
    while (len - nwritten > 0) {
     nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
     nwritten += nbytes;
    }
    fprintf(stderr,"\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    export large object "lobjOid" to file "out_
filename"
 *
 */
void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT|O_WRONLY, 0666);
    if (fd < 0)  {   /* error */
     fprintf(stderr, "can't open unix file %s\n",
          filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0) {
     tmp = write(fd, buf, nbytes);
        if (tmp < nbytes) {
         fprintf(stderr,"error while writing %s\n",
              filename);
     }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
```

```
}

void
exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

int
main(int argc, char **argv)
{
    char *in_filename, *out_filename;
    char *database;
    Oid lobjOid;
    PGconn *conn;
    PGresult *res;

    if (argc != 4) {
     fprintf(stderr, "Usage: %s database_name in_filename out_filename
\n",
          argv[0]);
     exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made
 */
    if (PQstatus(conn) == CONNECTION_BAD) {
     fprintf(stderr,"Connection to database '%s' failed.\n", database)
;
     fprintf(stderr,"%s",PQerrorMessage(conn));
     exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);
```

```
    printf("overwriting bytes 1000-2000 of the large object with X's
\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid,out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

第 52 章 ecpg - C 中的嵌入式 SQL

Linus Tolke
Michael Meskes

本文介绍 Postgres 的一个 C 中的嵌入式 SQL 软件包。它由 Linus Tolke[1] 和 Michael Meskes[2] 编写。

> **注意**
>
> Permission is granted to copy and use in the same way as you are allowed to copy and use the rest of PostgreSQL.

52.1. Why Embedded SQL?

与处理 SQL 查询的其他方式相比，嵌入式 SQL 有一些小的优势。它负责了在 C 程序的变量之间移动信息的全部繁琐工作。许多 RDBMS 软件包都支持这种嵌入式语言。

有一个 ANSI 标准描述了这种嵌入式语言应当如何工作。ecpg 的设计尽可能满足这个标准。因此可以把为其他 RDBMS 软件包编写的嵌入式 SQL 程序移植到 Postgres，从而弘扬自由软件的精神。

52.2. 概念

你用 C 加上一些特殊的 SQL 内容编写程序。要声明可以在 SQL 语句中使用的变量，需要把它们放在特殊的 declare 区中。SQL 查询要使用特殊的语法。

编译之前，先用嵌入式 SQL C 预处理器处理文件，它把你使用的 SQL 语句转换成以变量为参数的函数调用。既传递用作 SQL 语句输入的变量，也传递将包含结果的变量。

然后你编译并在链接时与一个包含所用函数的特殊库链接。这些函数（实际上主要是一个函数）从参数中获取信息，用普通接口（`libpq`）执行 SQL 查询，并把结果放回专门用于输出的参数中。

然后你运行程序，当控制到达 SQL 语句时，该 SQL 语句就会针对数据库执行，你可以接着使用结果。

52.3. How To Use ecpg

本节介绍如何使用 ecpg 工具。

52.3.1. Preprocessor

预处理器叫做 ecpg。安装后它位于 Postgres 的 `bin/` 目录中。

[1] linus@epact.se
[2] meskes@debian.org

52.3.2. Library

ecpg 库叫做 `libecpg.a` 或 `libecpg.so`。此外，该库用 `libpq` 库与 Postgres 服务器通信，因此你必须用 *`-lecpg -lpq`* 链接你的程序。

库中有一些"隐藏的"方法，但它们有时可能非常有用。

- `ECPGdebug(int` *`on`*`, FILE *`*`stream`*`)` 在第一个参数非零时打开调试日志。调试日志输出到 *`stream`*。大多数 SQL 语句会记录其参数和结果。

  最重要的一个（`ECPGdo`）几乎在所有 SQL 语句上都会被调用，它同时记录展开后的字符串、即插入了所有输入变量的字符串，以及 Postgres 服务器的结果。在查找 SQL 语句中的错误时这非常有用。

- `ECPGstatus()` 这个方法在已连接到数据库时返回 TRUE，否则返回 FALSE。

52.3.3. 错误处理

为了能够检测来自 Postgres 服务器的错误，你可以在文件的 include 区中包含像下面这样的一行：

```
exec sql include sqlca;
```

这会定义一个结构以及一个名为 *`sqlca`* 的变量，如下：

```
struct sqlca
{
 char sqlcaid[8];
 long sqlabc;
 long sqlcode;
 struct
 {
  int sqlerrml;
  char sqlerrmc[70];
 } sqlerrm;
 char sqlerrp[8];
 long sqlerrd[6];
 /* 0: empty                                        */
 /* 1: OID of processed tuple if applicable         */
 /* 2: number of rows processed in an INSERT, UPDATE */
 /*    or DELETE statement                          */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 char sqlwarn[8];
 /* 0: set to 'W' if at least one other is 'W'      */
 /* 1: if 'W' at least one character string      */
 /*    value was truncated when it was              */
 /*    stored into a host variable.                 */
 /* 2: empty                                        */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
```

```
 /* 6: empty                                                */
 /* 7: empty                                                */
 char sqlext[8];
} sqlca;
```

如果上一条 SQL 语句发生了错误，则 *sqlca.sqlcode* 将非零。如果 *sqlca.sqlcode* 小于 0，这是某种严重的错误，例如数据库定义与所给的查询不匹配。如果大于 0，则是普通错误，例如表中没有所请求的行。

sqlca.sqlerrm.sqlerrmc 将包含一个描述错误的字符串。该字符串以源文件中的行号结尾。

可能发生的错误列表：

-12, Out of memory in line %d.

通常不发生。这表示你的虚拟内存已经耗尽。

-200, Unsupported type %s on line %d.

通常不发生。这表示预处理器生成了库不认识的内容。
也许你运行的预处理器和库的版本不兼容。

-201, Too many arguments line %d.

这表示 Postgres 返回的参数比我们拥有的匹配变量多。也许你在 `INTO :var1,:var2` 列表中漏掉了几个宿主变量。

-202, Too few arguments line %d.

这表示 Postgres 返回的参数比我们拥有的宿主变量少。也许你在 `INTO :var1,:var2` 列表中放了太多宿主变量。

-203, Too many matches line %d.

这表示查询返回了多行但指定的变量不是数组。你执行的 `SELECT` 很可能不唯一。

-204, Not correctly formatted int type: %s line %d.

这表示宿主变量的类型是 `int`，而 Postgres 数据库中的字段是另一种类型，包含不能被解释为 `int` 的值。库用 `strtol` 做这个转换。

-205, Not correctly formatted unsigned type: %s line %d.

这表示宿主变量的类型是 `unsigned int`，而 Postgres 数据库中的字段是另一种类型，包含不能被解释为 `unsigned int` 的值。库用 `strtoul` 做这个转换。

-206, Not correctly formatted floating point type: %s line %d.

这表示宿主变量的类型是 `float`，而 Postgres 数据库中的字段是另一种类型，包含不能被解释为 `float` 的值。库用 `strtod` 做这个转换。

-207, Unable to convert %s to bool on line %d.

这表示宿主变量的类型是 `bool`，而 Postgres 数据库中的字段既不是 't' 也不是 'f'。

-208, Empty query line %d.

Postgres 返回了 PGRES_EMPTY_QUERY，很可能因为查询确实是空的。

-220, No such connection %s in line %d.

程序试图访问一个不存在的连接。

-221, Not connected in line %d.

程序试图访问一个确实存在但未打开的连接。

-230, Invalid statement name %s in line %d.

你试图使用的语句还没有被准备。

-400, Postgres error: %s line %d.

某种 Postgres 错误。消息中包含来自 Postgres 后端的错误消息。

-401, Error in transaction processing line %d.

Postgres 向我们表示无法开始、提交或回滚事务。

-402, connect: could not open database %s.

连接数据库失败。

100, Data not found line %d.

这是一个"正常"错误，告诉你所查询的内容找不到，或者我们已经遍历完了游标。

52.4. Limitations

永远不会被包含的内容及其原因，或者这个概念做不到的事情。

Oracle 的单任务可能性

AIX 3 上的 Oracle 7.0 版使用共享内存段上由 OS 支持的锁，允许应用设计者以所谓单任务的方式链接应用。不是每个应用进程启动一个客户端进程，
而是数据库部分和应用部分运行在同一个进程中。在 Oracle 的后续版本中这已不再受支持。

这需要彻底重新设计 Postgres 的访问模型，而这种努力配不上获得的性能。

52.5. 从其他 RDBMS 软件包移植

ecpg 的设计遵循 SQL 标准。因此从标准的 RDBMS 移植应该不成问题。遗憾的是并不存在所谓标准的 RDBMS。所以 ecpg 也会尽量理解语法附加，只要它们不与标准冲突。

下面列出所有已知的不兼容之处。如果你发现未列出的问题，请通知 Michael Meskes[3]。不过要注意，我们只列出从其他 RDBMS 的预编译器到 ecpg 的不兼容，而不列出这些 RDBMS 不具备的额外 ecpg 特性。

FETCH 命令的语法

FETCH 命令的标准语法是：

FETCH [direction] [amount] IN|FROM *cursor name*.

[3] meskes@debian.org

然而 ORACLE 不使用关键字 IN 或 FROM。这个特性无法添加，因为会造成解析冲突。

52.6. Installation

从 0.5 版开始，ecpg 与 Postgres 一起发行。因此作为安装的一部分，你的预编译器、库和头文件默认应该已被编译和安装。

52.7. 给开发者

本节面向想要开发 ecpg 接口的人。它描述各部分如何工作。这里的定位是让本节包含给想深入了解的人的内容，而如何使用一节应该足以回答所有常规问题。所以，在研究 ecpg 的内部之前先读这一节。如果你对它究竟如何工作不感兴趣，请跳过本节。

52.7.1. 待办列表

这个版本的预处理器有一些缺陷：

Library functions

to_date 等不存在。不过 Postgres 自身有一些不错的转换例程。所以你大概不会想念它们。

结构与联合

结构和联合必须在 declare 区中定义。

缺失的语句

下列语句迄今尚未实现：

exec sql allocate

exec sql deallocate

SQLSTATE

消息 'no data found'

exec sql insert select from 语句中"no data"的错误消息应该是 100。

sqlwarn[6]

如果 SET DESCRIPTOR 语句中指定的 PRECISION 或 SCALE 值将被忽略，sqlwarn[6] 应为 'W'。

52.7.2. The Preprocessor

写入输出的前四行是 ecpg 固定添加的内容：其中两行是注释，另外两行是与库接口所必需的包含行。

随后预处理器只单遍工作，边读输入文件边写输出。
通常它只是把所有内容原样回显到输出而不再细看。

遇到 EXEC SQL 语句时它会介入，并根据语句的种类改写它们。The EXEC SQL statement can be one of these:

Declare sections

声明节以

```
exec sql begin declare section;
```

开始，以

```
exec sql end declare section;
```

结束。节中只允许变量声明。
在这个节中声明的每个变量还会连同其相应的类型一起按名字登记到一个变量列表中。

特别是结构或联合的定义也必须列在 declare 区内。否则 ecpg 无法处理这些类型，因为它根本不知道定义。

声明也会被回显到文件中，使该变量同时成为一个普通的 C 变量。

特殊类型 VARCHAR 和 VARCHAR2 会为每个变量转换成一个命名的结构。像这样的声明：

```
VARCHAR var[180];
```

is converted into

```
struct varchar_var { int len; char arr[180]; } var;
```

包含语句

include 语句的形式是：

```
exec sql include filename;
```

注意这不同于

```
#include <filename.h>
```

Instead the file specified is parsed by ecpg itself. So the contents of the specified file is included in the resulting C code. This way you are able to specify EXEC SQL commands in an include file.

连接语句

connect 语句的形式是：

```
exec sql connect to connection target;
```

它创建到指定数据库的一个连接。

connection target 可以用下列方式指定：

dbname[@server][:port][as *connection name*][user *user name*]

tcp:postgresql://server[:port][/dbname][as *connection name*][user *user name*]

unix:postgresql://server[:port][/dbname][as *connection name*][user *user name*]

character variable[as *connection name*][user *user name*]

character string[as *connection name*][*user*]

default

user

还有几种指定用户名的方式：

userid

userid/*password*

userid identified by *password*

userid using *password*

最后是 userid 和 password。它们各自可以是常量文本、字符变量或字符串。

断开语句

disconnect 语句的形式是：

```
exec sql disconnect [connection target];
```

它关闭到指定数据库的连接。

connection target 可以用下列方式指定：

connection name

default

current

all

打开游标语句

open cursor 语句的形式是：

```
exec sql open cursor;
```

它被忽略，不复制到输出中。

提交语句

commit 语句的形式是

```
exec sql commit;
```

在输出中被翻译为

```
ECPGcommit(__LINE__);
```

回滚语句

rollback 语句的形式是

```
exec sql rollback;
```

and is translated on the output to

```
ECPGrollback(__LINE__);
```

其他语句

其他 SQL 语句是其他以 `exec sql` 开始、以 ; 结束的语句。中间的一切都被当作 SQL 语句并解析变量替换。

当符号以冒号（:）开头时发生变量替换。
此时会在先前于声明节中声明的变量里查找具有该名字的变量，
并且根据该变量用于输入还是输出，把指向变量的指针写入输出以允许函数访问。

For every variable that is part of the SQL request the function gets another ten arguments:

The type as a special symbol.
A pointer to the value or a pointer to the pointer.
The size of the variable if it is a char or varchar.
Number of elements in the array (for array fetches).
The offset to the next element in the array (for array fetches)
The type of the indicator variable as a special symbol.
A pointer to the value of the indicator variable or a pointer to the pointer of the indicator variable.
0.
Number of elements in the indicator array (for array fetches).
The offset to the next element in the indicator array (for array fetches)

52.7.3. 一个完整的例子

下面是一个完整的例子，描述预处理器对文件 foo.pgc 的输出：

```
exec sql begin declare section;
int index;
int result;
exec sql end declare section;
...
exec sql select res into :result from mytable where index = :index;
```

被翻译成：

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "select  res  from mytable where index = ?
 ",
 ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

（本手册中的缩进是为了可读性而加的，并不是预处理器能做的。）

52.7.4. 库

库中最重要的函数是 `ECPGdo` 函数。它接受数量可变的参数。但愿我们不会碰到对 vararg 函数可接受的变量数量有限制的机器。这很容易累积到 50 个左右的参数。

参数有：

一个行号

这是原语句的行号，只用于错误消息。

一个字符串

这是要发出的 SQL 请求。该请求被输入变量修改，即那些编译时未知但要输入请求的变量。在变量应出现的位置，字符串中包含 ";"。

输入变量

如关于预处理器的一节所述，每个输入变量得到十个参数。

ECPGt_EOIT

一个枚举，表示不再有输入变量。

输出变量

如关于预处理器的一节所述，每个输入变量得到十个参数。These variables are filled by the function.

ECPGt_EORT

一个枚举，表示不再有变量。

所有 SQL 语句都在一个事务中执行，除非你发出提交事务。为了让这种自动事务运转起来，第一条语句或提交/回滚之后的第一条语句总是会开始一个事务。要在默认情况下禁用这个特性，请在命令行使用 `-t` 选项。

待完成：描述其他条目的条目。

第 53 章 libpq - C 库

libpq 是 Postgres 的 C 应用程序员接口。libpq 是一组库例程，客户端程序通过它们可以向 Postgres 后端服务器传送查询并接收这些查询的结果。libpq 也是其他几个 Postgres 应用接口的底层引擎，包括 libpq++（C++）、libpgtcl（Tcl）、Perl 和 ecpg。因此，即使你使用的是上述某个软件包，libpq 行为的某些方面对你来说也很重要。

本节末尾包含三个简短程序，展示如何编写使用 libpq 的程序。下列目录中还有几个完整的 libpq 应用程序示例：

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

使用 libpq 的前端程序必须包含头文件 libpq-fe.h，并且必须与 libpq 库链接。

53.1. 数据库连接函数

下面的例程用于建立与 Postgres 后端服务器的连接。一个应用程序可以同时打开多个后端连接（原因之一是需要访问多个数据库）。每个连接由一个 PGconn 对象表示，该对象通过 PQconnectdb() 或 PQsetdbLogin() 获得。注意，除非内存太少以至于连 PGconn 对象都无法分配，这些函数总是返回一个非空的对象指针。在通过连接对象发送查询之前，应当调用 PQstatus 函数检查连接是否成功建立。

- PQconnectdb 与数据库服务器建立一个新连接。

  ```
  PGconn *PQconnectdb(const char *conninfo)
  ```

  此例程使用从字符串 conninfo 中取得的参数打开一个新的数据库连接。与下面的 PQsetdbLogin() 不同，该例程的参数集可以在不改变函数签名的情况下扩展，因此应用程序编程首选使用此例程或其非阻塞版本 PQconnectStart / PQconnectPoll。传入的字符串可以为空以使用全部默认参数，也可以包含一个或多个用空白分隔的参数设置。

  每个参数设置的形式为 keyword = value。（要写入空值或包含空格的值，请用单引号包围，例如 keyword = 'a value'。值中的单引号必须写作 \'。等号两边的空格是可选的。）目前可识别的参数关键字有：

  host

  要连接的主机名。如果指定了非零长度的字符串，则使用 TCP/IP 通信。使用此参数会引起主机名查找。参见 hostaddr。

  hostaddr

  要连接的主机的 IP 地址。其格式应为 BSD 函数 inet_aton 等使用的标准点分数字形式。如果指定了非零长度的字符串，则使用 TCP/IP 通信。

  使用 hostaddr 代替主机名可以让应用避免主机名查找，这对于有时间约束的应用可能很重要。但 Kerberos 认证需要主机名。因此适用以下规则。如果指定了主机名而没有指定 hostaddr，则强制进行主机名查找。如果指定了 hostaddr 而没有指定主机名，hostaddr 的值给出远程地址；若使用了 Kerberos，这会引起反向名称查询。如果主机名和 hostaddr 都被指定，hostaddr 的值给出远程地址；主机名的值被忽略，但使用

Kerberos 时例外，此时该值用于 Kerberos 认证。注意，如果传给 libpq 的主机名不是位于 hostaddr 的机器的名称，认证很可能失败。

既没有主机名也没有主机地址时，libpq 将使用本地 Unix 域套接字连接。

`port`

服务器主机上要连接的端口号，或 Unix 域连接的套接字文件名扩展。

`dbname`

数据库名。

`user`

要以哪个用户名连接。

`password`

如果服务器要求密码认证则使用的密码。

`options`

要发送给服务器的跟踪/调试选项。

`tty`

用于后端可选调试输出的文件或 tty。

如果任何参数未指定，则检查相应的环境变量（见"环境变量"一节）。
如果环境变量也未设置，则使用硬编码的默认值。返回值是指向一个抽象 struct 的指针，该结构表示到后端的连接。

- `PQsetdbLogin` 与数据库服务器建立一个新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd)
```

这是 `PQconnectdb` 的前身，参数个数固定但功能相同。

- `PQsetdb` 与数据库服务器建立一个新连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName)
```

这是一个宏，以空指针作为 login 和 pwd 参数调用 `PQsetdbLogin()`。它主要是为了与旧程序保持向后兼容而保留。

- `PQconnectStartPQconnectPoll` 以非阻塞方式建立与数据库服务器的连接。

```
PGconn *PQconnectStart(const char *conninfo)

PostgresPollingStatusType *PQconnectPoll(PQconn *conn)
```

这两个例程用于打开到数据库服务器的连接，同时应用程序的执行线程不会在远程 I/O 上阻塞。

数据库连接使用传给 PQconnectStart 的字符串 `conninfo` 中的参数建立。该字符串的格式与上面 PQconnectdb 中描述的相同。

只要满足若干限制条件，PQconnectStart 和 PQconnectPoll 都不会阻塞：

- 恰当地使用 hostaddr 和 host 参数，确保不进行（正向）名称和反向名称查询。详情见上面 PQconnectdb 下对这些参数的说明。
- 如果调用 PQtrace，确保要写入跟踪信息的流对象不会阻塞。
- 在调用 PQconnectPoll 之前，自行确保套接字处于适当的状态，如下所述。

首先，调用 `conn=PQconnectStart("<connection_info_string>")`。如果 conn 为 NULL，说明 libpq 无法分配新的 PGconn 结构。否则返回一个有效的 PGconn 指针（尽管它尚未表示一个有效的数据库连接）。从 PQconnectStart 返回后，调用 status=PQstatus(conn)。如果 status 等于 CONNECTION_BAD，则 PQconnectStart 失败。

如果 PQconnectStart 成功，下一阶段是轮询 libpq，使其继续进行连接序列。循环如下：默认将连接视为'非活跃'。如果 PQconnectPoll 上次返回了 PGRES_POLLING_ACTIVE，则改将其视为'活跃'。如果 PQconnectPoll(conn) 上次返回 PGRES_POLLING_READING，则在 PQsocket(conn) 上为读取执行 select。如果上次返回 PGRES_POLLING_WRITING，则在 PQsocket(conn) 上为写入执行 select。如果尚未调用过 PQconnectPoll（即刚调用完 PQconnectStart 之后），则视同它上次返回的是 PGRES_POLLING_WRITING。如果 select 表明套接字已就绪，将其视为'活跃'。一旦确定此连接处于'活跃'状态，就再次调用 PQconnectPoll(conn)。如果这次调用返回 PGRES_POLLING_FAILED，连接过程已失败。如果返回 PGRES_POLLING_OK，连接已成功建立。

注意，用 select() 确保套接字就绪只是一个（常见的）例子；如果还有其他可用手段，例如 poll() 调用，当然可以改用它。

在连接期间的任何时刻，都可以调用 PQstatus 检查连接状态。如果返回 CONNECTION_BAD，则连接过程已失败；如果返回 CONNECTION_OK，则连接已就绪。如上所述，这两种状态都应能从 PQconnectPoll 的返回值中同样检测到。其他状态只在（且仅会在）异步连接过程中出现，它们指示连接过程的当前阶段，例如可用于向用户提供反馈。这些状态可能包括：

- CONNECTION_STARTED：等待建立连接。
- CONNECTION_MADE：连接成功；等待发送。
- CONNECTION_AWAITING_RESPONSE：等待 postmaster 的响应。
- CONNECTION_AUTH_OK：已通过认证；等待后端启动。
- CONNECTION_SETENV：正在协商环境。

注意，尽管这些常量会保留（为了维护兼容性），应用程序绝不应依赖它们按特定顺序出现、依赖它们全都出现，或依赖状态总是这些已记录的值之一。应用程序可以这样处理：

```
    switch(PQstatus(conn))
```

```
    {
        case CONNECTION_STARTED:
            feedback = "Connecting...";
     break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
     feedback = "Connecting...";
    }
```

注意，如果 PQconnectStart 返回非 NULL 指针，使用完毕后必须调用 PQfinish，以释放该结构和所有关联的内存块。即使 PQconnectStart 或 PQconnectPoll 的调用失败，也必须这样做。

当前，如果 libpq 编译时定义了 USE_SSL，PQconnectPoll 会阻塞。此限制将来可能被移除。

当前，在 Windows 下 PQconnectPoll 会阻塞，除非 libpq 编译时定义了 WIN32_NON_BLOCKING_CONNECTIONS。这段代码尚未在 Windows 下测试过，因此目前默认关闭。将来可能改变。

这些函数会把套接字置于非阻塞状态，就像调用过 `PQsetnonblocking` 一样。

- `PQconndefaults` 返回默认的连接选项。

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by
 default */
    int     dispsize;  /* Field size in characters for dialog */
}
```

返回一个连接选项数组。它可用来确定 PQconnectdb 的所有可用选项及其当前默认值。返回值指向一个 PQconninfoOption struct 数组，该数组以一个 keyword 指针为 NULL 的条目结尾。注意，默认值（"val" 字段）依赖于环境变量和其他上下文。调用者必须将连接选项数据视为只读。

处理完选项数组后，将它传递给 PQconninfoFree() 释放。如果不这样做，每次调用 PQconndefaults() 都会泄漏少量内存。

在 Postgres 7.0 之前的版本中，PQconndefaults() 返回指向静态数组的指针，而不是动态分配的数组。那样不是线程安全的，因此行为已被改变。

- PQfinish 关闭与后端的连接，同时释放 PGconn 对象使用的内存。

  ```
  void PQfinish(PGconn *conn)
  ```

  注意，即使与后端的连接尝试失败（由 PQstatus 指示），应用也应调用 PQfinish 释放 PGconn 对象使用的内存。调用过 PQfinish 之后，不应再使用该 PGconn 指针。

- PQreset 重置与后端的通信端口。

  ```
  void PQreset(PGconn *conn)
  ```

  此函数将关闭与后端的连接，并尝试使用先前使用的全部相同参数与同一个 postmaster 重新建立新连接。如果工作中的连接丢失，这可用于错误恢复。

- PQresetStartPQresetPoll 以非阻塞方式重置与后端的通信端口。

  ```
  int PQresetStart(PGconn *conn);

  PostgresPollingStatusType PQresetPoll(PGconn *conn);
  ```

  这些函数将关闭与后端的连接，并尝试使用先前使用的全部相同参数与同一个 postmaster 重新建立新连接。如果工作中的连接丢失，这可用于错误恢复。它们与上面的 PQreset 的不同之处在于以非阻塞方式运作。这些函数受到与 PQconnectStart 和 PQconnectPoll 相同的限制。

  调用 PQresetStart。如果返回 0，重置失败。如果返回 1，用与使用 PQconnectPoll 建立连接完全相同的方式，使用 PQresetPoll 对重置进行轮询。

libpq 应用程序员应注意维护 PGconn 的抽象。请使用下面的访问函数获取 PGconn 的内容。避免直接引用 PGconn 结构的字段，因为它们将来可能改变。（从 Postgres 6.4 版开始，libpq-fe.h 中甚至不再提供 struct PGconn 的定义。如果你有直接访问 PGconn 字段的旧代码，可以通过同时包含 libpq-int.h 继续使用它，但我们建议你尽快修改这些代码。）

- PQdb 返回连接的数据库名。

  ```
  char *PQdb(const PGconn *conn)
  ```

  PQdb 及其后面几个函数返回在建立连接时确定的值。这些值在 PGconn 对象的整个生命周期内保持不变。

- PQuser 返回连接的用户名。

  ```
  char *PQuser(const PGconn *conn)
  ```

- PQpass 返回连接的密码。

  ```
  char *PQpass(const PGconn *conn)
  ```

- PQhost 返回连接的服务器主机名。

  ```
  char *PQhost(const PGconn *conn)
  ```

- PQport 返回连接的端口。

  ```
  char *PQport(const PGconn *conn)
  ```

- PQtty 返回连接的调试 tty。

  ```
  char *PQtty(const PGconn *conn)
  ```

- PQoptions 返回连接中使用的后端选项。

  ```
  char *PQoptions(const PGconn *conn)
  ```

- PQstatus 返回连接的状态。

  ```
  ConnStatusType PQstatus(const PGconn *conn)
  ```

  状态可以是若干值之一。但在异步连接过程之外只能见到其中两个-- CONNECTION_OK 或 CONNECTION_BAD。正常的数据库连接状态为 CONNECTION_OK。失败的连接尝试由状态 CONNECTION_BAD 表示。通常，OK 状态会一直保持到调用 PQfinish 为止，但通信故障可能导致状态提前变为 CONNECTION_BAD。此时应用可以尝试调用 PQreset 来恢复。

  关于可能见到的其他状态码，参见 PQconnectStart 和 PQconnectPoll 的条目。

- PQerrorMessage 返回连接上最近一次操作所产生的错误消息。

  ```
  char *PQerrorMessage(const PGconn* conn);
  ```

  几乎所有 libpq 函数在失败时都会设置 PQerrorMessage。注意，按照 libpq 的惯例，非空的 PQerrorMessage 会包含一个末尾换行符。

- PQbackendPID 返回处理此连接的后端服务器的进程 ID。

  ```
  int PQbackendPID(const PGconn *conn);
  ```

  后端 PID 可用于调试，也可用于与 NOTIFY 消息（其中包含发出通知的后端的 PID）进行比较。注意，该 PID 属于数据库服务器主机上执行的进程，而不是本地主机上的进程！

53.2. 查询执行函数

与数据库服务器的连接成功建立后，此处描述的函数用于执行 SQL 查询和命令。

- PQexec 向 Postgres 提交一个查询并等待结果。

  ```
  PGresult *PQexec(PGconn *conn,
                   const char *query);
  ```

  返回一个 PGresult 指针，也可能返回 NULL 指针。除内存耗尽或无法将查询发送到后端等严重错误外，一般会返回非 NULL 指针。如果返回了 NULL，应将其视同 PGRES_FATAL_ERROR 结果。使用 PQerrorMessage 获取该错误的更多信息。

PGresult 结构封装了后端返回的查询结果。libpq 应用程序员应注意维护 PGresult 的抽象。请使用下面的访问函数获取 PGresult 的内容。避免直接引用 PGresult 结构的字段，因为它们将来可能改变。（从 Postgres 6.4 版开始，libpq-fe.h 中甚至不再提供 struct PGresult 的定义。如果你有直接访问 PGresult 字段的旧代码，可以通过同时包含 libpq-int.h 继续使用它，但我们建议你尽快修改这些代码。）

- PQresultStatus 返回查询的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res)
```

PQresultStatus 可以返回下列值之一:

- PGRES_EMPTY_QUERY —— 发送给后端的字符串为空。
- PGRES_COMMAND_OK —— 不返回数据的命令成功完成
- PGRES_TUPLES_OK —— 查询成功执行
- PGRES_COPY_OUT —— Copy Out（从服务器传出）数据传输已开始
- PGRES_COPY_IN —— Copy In（向服务器传入）数据传输已开始
- PGRES_BAD_RESPONSE —— 无法理解服务器的响应
- PGRES_NONFATAL_ERROR
- PGRES_FATAL_ERROR

如果结果状态为 PGRES_TUPLES_OK，就可以使用下面描述的例程检索查询返回的元组。注意，碰巧检索到零个元组的 SELECT 仍然显示 PGRES_TUPLES_OK。PGRES_COMMAND_OK 用于永远不会返回元组的命令（INSERT、UPDATE 等）。PGRES_EMPTY_QUERY 响应往往暴露客户端软件中的 bug。

- PQresStatus 把 PQresultStatus 返回的枚举类型转换为描述该状态码的字符串常量。

  ```
  char *PQresStatus(ExecStatusType status);
  ```

- PQresultErrorMessage 返回与查询相关联的错误消息；如果没有错误则返回空字符串。

  ```
  char *PQresultErrorMessage(const PGresult *res);
  ```

  在 PQexec 或 PQgetResult 调用之后紧接着，（连接上的）PQerrorMessage 会返回与（结果上的）PQresultErrorMessage 相同的字符串。但 PGresult 会保留其错误消息直到被销毁，而连接的错误消息会随后续操作而变化。想知道与某个特定 PGresult 相关联的状态时使用 PQresultErrorMessage；想知道连接上最新操作的状态时使用 PQerrorMessage。

- PQntuples 返回查询结果中的元组（实例）数。

  ```
  int PQntuples(const PGresult *res);
  ```

- PQnfields 返回查询结果中每个元组的字段（属性）数。

  ```
  int PQnfields(const PGresult *res);
  ```

- PQbinaryTuples 如果 PGresult 包含二进制元组数据则返回 1，包含 ASCII 数据则返回 0。

  ```
  int PQbinaryTuples(const PGresult *res);
  ```

  目前，只有从 BINARY 游标提取数据的查询才能返回二进制元组数据。

- PQfname 返回与给定字段下标相关联的字段（属性）名。字段下标从 0 开始。

  ```
  char *PQfname(const PGresult *res,
                int field_index);
  ```

- PQfnumber 返回与给定字段名相关联的字段（属性）下标。

```
int PQfnumber(const PGresult *res,
              const char *field_name);
```

如果给定的名称不匹配任何字段，返回 -1。

- PQftype 返回与给定字段编号相关联的字段类型。返回的整数是该类型的内部编码。字段编号从 0 开始。

```
Oid PQftype(const PGresult *res,
            int field_num);
```

可以查询系统表 pg_type 来获取各种数据类型的名称和属性。内置数据类型的 OID 定义在源码树的 src/include/catalog/pg_type.h 文件中。

- PQfsize 返回与给定字段下标相关联的字段的大小，以字节计。字段下标从 0 开始。

```
int PQfsize(const PGresult *res,
            int field_index);
```

PQfsize　返回数据库元组中为该字段分配的空间，换句话说，即服务器对该数据类型的二进制表示的大小。如果字段是变长的则返回 -1。

- PQfmod 返回与给定字段下标相关联的字段的类型专属修饰数据。字段下标从 0 开始。

```
int PQfmod(const PGresult *res,
           int field_index);
```

- PQgetvalue 返回 PGresult 中某个元组的单个字段（属性）值。元组和字段下标都从 0 开始。

```
char* PQgetvalue(const PGresult *res,
                 int tup_num,
                 int field_num);
```

对大多数查询而言，PQgetvalue 返回的是属性值的以空字符结尾的 ASCII 字符串表示。但如果 PQbinaryTuples() 为 1，PQgetvalue 返回的就是该类型以后端服务器内部格式表示的二进制表示（若字段为变长则不含大小字）。此时由程序员负责把数据转换成正确的 C 类型。PQgetvalue 返回的指针指向属于 PGresult 结构的存储空间。不应修改它；如果需要在 PGresult 结构的生命周期结束后继续使用该值，就必须显式地将它复制到其他存储空间。

- PQgetlength 返回字段（属性）的长度，以字节计。元组和字段编号从 0 开始。

```
int PQgetlength(const PGresult *res,
                int tup_num,
                int field_num);
```

这是该特定数据值的实际数据长度，即 PQgetvalue 所指对象的大小。注意，对于以 ASCII 表示的值，此大小与 PQfsize 报告的二进制大小几乎没有关系。

- PQgetisnull 测试一个字段是否为 NULL 条目。元组和字段下标从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int tup_num,
                int field_num);
```

如果字段包含 NULL，此函数返回 1；包含非 NULL 值则返回 0。（注意，对于 NULL 字段，PQgetvalue 返回的是空字符串而不是空指针。）

- PQcmdStatus 返回生成该 PGresult 的 SQL 命令的命令状态字符串。

  ```
  char * PQcmdStatus(const PGresult *res);
  ```

- PQcmdTuples 返回受 SQL 命令影响的行数。

  ```
  char * PQcmdTuples(const PGresult *res);
  ```

  如果生成该 PGresult 的 SQL 命令是 INSERT、UPDATE 或 DELETE，此函数返回一个包含受影响行数的字符串。如果是其他命令，返回空字符串。

- PQoidValue 如果 SQL 命令是一个 INSERT，返回所插入元组的对象 id。否则返回 InvalidOid。

  ```
  Oid PQoidValue(const PGresult *res);
  ```

  包含 libpq 头文件后，将定义类型 Oid 和常量 InvalidOid。它们都属于某种整数类型。

- PQoidStatus 如果 SQL 命令是一个 INSERT，返回一个含有所插入元组对象 id 的字符串。否则返回空字符串。

  ```
  char * PQoidStatus(const PGresult *res);
  ```

  此函数已被弃用，由 PQoidValue 取代，且不是线程安全的。

- PQprint 打印出所有的元组以及（可选的）属性名到指定的输出流。

  ```
  void PQprint(FILE* fout,      /* output stream */
               const PGresult *res,
               const PQprintOpt *po);

  struct {
      pqbool  header;      /* print output field headings and row
   count */
      pqbool  align;       /* fill align the fields */
      pqbool  standard;    /* old brain dead format */
      pqbool  html3;       /* output html tables */
      pqbool  expanded;    /* expand tables */
      pqbool  pager;       /* use pager for output if needed */
      char    *fieldSep;   /* field separator */
      char    *tableOpt;   /* insert to HTML table ... */
      char    *caption;    /* HTML caption */
      char    **fieldName; /* null terminated array of replacement
   field names */
  } PQprintOpt;
  ```

  此函数以前被 psql 用于打印查询结果，但现在已不再如此，此函数也不再被积极维护。

- PQclear 释放与 PGresult 关联的存储。每个查询结果在不再需要时都应通过 PQclear 释放。

  ```
  void PQclear(PQresult *res);
  ```

  PGresult　对象需要保留多久就可以保留多久；发出新查询时它不会消失，即使关闭连接也不会。要清除它，必须调用 PQclear。不这样做将导致前端应用内存泄漏。

- PQmakeEmptyPGresult 以给定的状态构造一个空的 PGresult 对象。

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是 libpq 分配并初始化空 PGresult 对象的内部例程。导出它是因为某些应用发现自己生成结果对象（特别是带有错误状态的对象）很有用。如果 conn 非 NULL 且 status 指示一个错误，连接的当前 errorMessage 会被复制到 PGresult。注意，最终也应对该对象调用 PQclear，就像对待 libpq 本身返回的 PGresult 一样。

53.3. 异步查询处理

PQexec 函数足以在简单的同步应用中提交查询。但它有几个主要缺陷：

- PQexec 会等待查询完成。应用可能有其他工作要做（例如维护一个用户界面），这种情况下它不会想要阻塞等待响应。
- 由于控制权埋在 PQexec 内部，前端很难在需要时决定尝试取消正在进行的查询。（这可以在信号处理器中完成，但别无他法。）
- PQexec 只能返回一个 PGresult 结构。如果提交的查询串包含多条 SQL 命令，除了最后一个 PGresult 之外都会被 PQexec 丢弃。

不喜欢这些限制的应用可以改用构成 PQexec 的底层函数：PQsendQuery 和 PQgetResult。

使用此功能的旧程序以及 PQputline 和 PQputnbytes 都可能在等待向后端发送数据时阻塞。为了解决这一问题，加入了 PQsetnonblocking 函数。

旧应用可以不使用 PQsetnonblocking 而保持旧的、可能阻塞的行为。新程序则可以使用 PQsetnonblocking 实现到后端的完全非阻塞连接。

- PQsetnonblocking 把连接的状态设置为非阻塞。

  ```
  int PQsetnonblocking(PGconn *conn)
  ```

  此函数可确保对 PQputline、PQputnbytes、PQsendQuery 和 PQendcopy 的调用不会阻塞；如果需要再次调用，它们会返回错误。

  当数据库连接被设置为非阻塞模式后又调用了 PQexec 时，它会临时把连接置为阻塞状态，直到该 PQexec 完成。

  预计不久的将来，libpq 的更多部分将变得对 PQsetnonblocking 功能安全。

- PQisnonblocking 返回数据库连接的阻塞状态。

  ```
  int PQisnonblocking(const PGconn *conn)
  ```

  连接设置为非阻塞模式时返回 TRUE，阻塞时返回 FALSE。

- PQsendQuery 向 Postgres 提交一个查询而不等待结果。查询成功发出返回 TRUE，否则返回 FALSE（此时用 PQerrorMessage 获取失败的更多信息）。

  ```
  int PQsendQuery(PGconn *conn,
                  const char *query);
  ```

  成功调用 PQsendQuery 后，调用一次或多次 PQgetResult 获取查询结果。在 PQgetResult 返回 NULL（表示查询已完成）之前，不得（在同一连接上）再次调用 PQsendQuery。

- PQgetResult 等待先前一次 PQsendQuery 的下一个结果并将其返回。查询完成且不再有结果时返回 NULL。

  ```
  PGresult *PQgetResult(PGconn *conn);
  ```

  必须反复调用 PQgetResult 直到它返回 NULL，表示查询已完成。（如果在没有活跃查询时调用，PQgetResult 会立即返回 NULL。）对 PQgetResult 返回的每个非空结果，应使用前文所述的同一批 PGresult 访问函数处理。用完每个结果对象后别忘了用 PQclear 释放。注意，只有当查询处于活跃状态且必要的响应数据尚未被 PQconsumeInput 读取时，PQgetResult 才会阻塞。

使用 PQsendQuery 和 PQgetResult 解决了 PQexec 的一个问题：如果查询字符串包含多条 SQL 命令，这些命令的结果可以分别获取。（顺便说一句，这也支持一种简单的重叠处理：前端可以处理某条查询的结果，同时后端继续处理同一查询串中后面的查询。）但调用 PQgetResult 仍会使前端阻塞，直到后端完成下一条 SQL 命令。恰当使用另外三个函数可以避免这一点：

- PQconsumeInput 如果后端有可用的输入，消费它。

  ```
  int PQconsumeInput(PGconn *conn);
  ```

  PQconsumeInput 正常返回 1 表示"no error"（没有错误）；发生问题时返回 0（此时 PQerrorMessage 会被设置）。注意，返回结果并不说明是否实际读取了输入数据。调用 PQconsumeInput 后，应用可以检查 PQisBusy 和/或 PQnotifies，看它们的状态是否发生了变化。

  即使应用尚未准备好处理结果或通知，也可以调用 PQconsumeInput。该例程会读取可用数据并保存到缓冲区中，从而使 select(2) 的可读就绪指示消失。因此应用可以用 PQconsumeInput 立即清除 select 的就绪条件，随后再从容地检查结果。

- PQisBusy 如果查询仍在忙碌则返回 1，也就是说 PQgetResult 会阻塞等待输入。返回 0 表示可以放心调用 PQgetResult 而不会阻塞。

  ```
  int PQisBusy(PGconn *conn);
  ```

  PQisBusy 本身不会尝试从后端读取数据；因此必须先调用 PQconsumeInput，否则忙碌状态永远不会结束。

- PQflush 尝试刷新排队发往后端的任何数据，成功（或发送队列为空）返回 0，因某种原因失败返回 EOF。

  ```
  int PQflush(PGconn *conn);
  ```

  在非阻塞连接上，调用 select 判断响应是否到达之前需要先调用 PQflush。返回 0 可确保没有已排队但尚未真正发送到后端的数据。只有使用了 PQsetnonblocking 的应用才需要这样做。

- PQsocket 获取后端连接套接字的文件描述符号。有效的描述符将 >= 0；结果为 -1 表示当前没有打开的后端连接。

  ```
  int PQsocket(const PGconn *conn);
  ```

  应当用 PQsocket 获取后端套接字描述符，为执行 select(2) 做准备。这使使用阻塞连接的应用可以同时等待后端响应或其他条件。如果 select(2) 的结果表明可以从后端套接字读取数据，就应调用 PQconsumeInput 读取数据；随后即可用 PQisBusy、PQgetResult 和/或 PQnotifies 处理响应。

非阻塞连接（使用了 PQsetnonblocking 的连接）在 PQflush 返回 0、表明没有缓冲数据等待发往后端之前，不应使用 select。

使用这些函数的典型前端会有一个主循环，用 select(2) 等待它必须响应的所有条件。其中一个条件是后端有可读的输入；用 select 的话说，就是由 PQsocket 标识的文件描述符上有可读数据。主循环检测到输入就绪时，应调用 PQconsumeInput 读取输入。然后可以调用 PQisBusy，若 PQisBusy 返回假（0），接着调用 PQgetResult。还可以调用 PQnotifies 检测 NOTIFY 消息（见下面的"异步通知"）。

使用 PQsendQuery/PQgetResult 的前端还可以尝试取消一个仍随后端处理中的查询。

- PQrequestCancel 请求 Postgres 放弃对当前查询的处理。

  ```
  int PQrequestCancel(PGconn *conn);
  ```

  取消请求成功发出则返回 1，否则返回 0。（若为 0，PQerrorMessage 会说明原因。）但成功发出并不能保证请求会生效。无论 PQrequestCancel 的返回值如何，应用都必须继续用 PQgetResult 按正常顺序读取结果。如果取消生效，当前查询会提前终止并返回一个错误结果。如果取消失败（例如因为后端已完成该查询的处理），则根本不会有可见的结果。

注意，如果当前查询是事务的一部分，取消将中止整个事务。

PQrequestCancel 可以在信号处理器中安全地调用。因此，如果取消的决定可以在信号处理器中做出，也可以将它与普通的 PQexec 配合使用。例如，psql 从 SIGINT 信号处理器中调用 PQrequestCancel，从而允许交互式地取消它通过 PQexec 发出的查询。注意，如果连接当前未打开或后端当前没有在处理查询，PQrequestCancel 将不起作用。

53.4. 快速路径

Postgres 提供一个快速路径接口来向后端发送函数调用。这是通向系统内部的一个天窗，可能成为潜在的安全漏洞。大多数用户不需要此特性。

- PQfn 通过快速路径接口请求执行一个后端函数。

  ```
  PGresult* PQfn(PGconn* conn,
                 int fnid,
                 int *result_buf,
                 int *result_len,
                 int result_is_int,
                 const PQArgBlock *args,
                 int nargs);
  ```

  fnid 参数是要执行的函数的对象标识符。result_buf 是存放返回值的缓冲区。调用者必须已分配足够的空间来存储返回值（这里不做检查！）。实际结果长度将返回到 result_len 所指向的整数中。如果预期结果是 4 字节整数，将 result_is_int 设为 1；否则设为 0。（将 result_is_int 设为 1 会告诉 libpq 在必要时做字节序交换，使值以适合客户端机器的 int 值送达。当 result_is_int 为 0 时，后端发送的字节串原样返回。）args 和 nargs 指定要传给函数的参数。

  ```
  typedef struct {
      int len;
      int isint;
      union {
          int *ptr;
          int integer;
  ```

```
    } u;
} PQArgBlock;
```

PQfn 总是返回有效的 PGresult*。使用结果前应检查其状态。不再需要结果时，调用者负责用 PQclear 释放 PGresult。

53.5. 异步通知

Postgres 通过 LISTEN 和 NOTIFY 命令支持异步通知。后端用 LISTEN 命令注册它对某个特定通知条件的兴趣（并可以用 UNLISTEN 命令停止监听）。当任何后端执行带该条件名的 NOTIFY 命令时，所有监听该条件的后端都会被异步通知。通知者不会向监听者传递任何附加信息。因此，通常需要通信的任何实际数据都通过一个数据库关系传递。
条件名通常与相关的关系同名，但并不要求必须有相关的关系。

libpq 应用把 LISTEN 和 UNLISTEN 命令作为普通 SQL 查询提交。随后，可以通过调用 PQnotifies() 检测 NOTIFY 消息的到达。

- PQnotifies 从后端发来的尚未处理的通知消息列表中返回下一条通知。没有待处理的通知时返回 NULL。一条通知被 PQnotifies 返回后即视为已处理，并会从通知列表中移除。

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char relname[NAMEDATALEN];        /* name of relation
                                       * containing data */
    int  be_pid;                      /* process id of backend */
} PGnotify;
```

处理完 PQnotifies 返回的 PGnotify 对象后，务必用 free() 释放它，以避免内存泄漏。

> **注意**
>
> 在 Postgres 6.4 及之后的版本中，be_pid 是发出通知的后端的 PID，而在较早的版本中它总是你自己的后端的 PID。

第二个示例程序演示了异步通知的使用。

PQnotifies() 并不真正读取后端数据；它只是返回先前被其他 libpq 函数吸收的消息。在 libpq 的早期版本中，确保及时收到 NOTIFY 消息的唯一方法是不断地提交查询（哪怕是空查询），然后在每次 PQexec() 之后检查 PQnotifies()。这种方法虽然仍然有效，但由于浪费处理能力已被弃用。

在没有有用查询可提交时检查 NOTIFY 消息的更好办法是调用 PQconsumeInput()，然后检查 PQnotifies()。可以用 select(2) 等待后端数据到达，这样除非有事可做就不会消耗 CPU。（获取用于 select 的文件描述符号的方法见 PQsocket()。）注意，无论你用 PQsendQuery/PQgetResult 提交查询还是直接用 PQexec，这样做都没有问题。但应记住在每次 PQgetResult 或 PQexec 之后检查 PQnotifies()，看查询处理期间是否有通知到来。

53.6. 与 COPY 命令相关的函数

Postgres 中的 COPY 命令可以选择从 libpq 所用的网络连接读取或向其写入。因此，需要能直接访问此网络连接的函数，应用才能利用这一能力。

只有在从 PQexec 或 PQgetResult 得到 PGRES_COPY_OUT 或 PGRES_COPY_IN 结果对象之后，才应执行这些函数。

- PQgetline 把一行由后端服务器传来的以换行符结尾的字符读入大小为 length 的缓冲区字符串。

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

  与 fgets(3) 一样，此例程最多把 length-1 个字符复制到 string 中；但与 gets(3) 相似，它会把结尾的换行符转换为空字符。PQgetline 在 EOF 处返回 EOF，整行已读完返回 0，缓冲区已满但结尾换行符尚未读到返回 1。

  注意，应用必须检查新的一行是否由两个字符 "\." 组成，这表示后端服务器已发送完 copy 命令的结果。如果应用可能收到超过 length-1 个字符的行，就需要小心确保正确识别 "\." 行（例如，不要把长数据行的结尾误当作终结行）。src/bin/psql/copy.c 中的代码包含正确处理 copy 协议的例程示例。

- PQgetlineAsync 把一行由后端服务器传来的以换行符结尾的字符读入缓冲区而不阻塞。

  ```
  int PQgetlineAsync(PGconn *conn,
                     char *buffer,
                     int bufsize)
  ```

  此例程类似于 PQgetline，但可供必须异步（即不阻塞地）读取 COPY 数据的应用使用。发出 COPY 命令并得到 PGRES_COPY_OUT 响应后，应用应交替调用 PQconsumeInput 和 PQgetlineAsync，直到检测到数据结束信号。与 PQgetline 不同，此例程自己负责检测数据结束。每次调用时，如果 libpq 的输入缓冲区中已有完整的一行以换行符结尾的数据行，或者到来的数据行太长放不进调用者提供的缓冲区，PQgetlineAsync 就会返回数据。否则，在该行剩余部分到达之前不返回数据。

  如果已识别到 copy 数据结束标记，此例程返回 -1；没有可用数据返回 0；返回正数时该数给出所返回数据的字节数。返回 -1 时，调用者接下来必须调用 PQendcopy，然后回到正常处理。
  返回的数据不会跨过换行符。可能时一次返回整行；
  但如果调用者提供的缓冲区太小装不下后端发来的一行，就会返回部分数据行。
  这可以通过检查最后返回的字节是否为 "\n" 来检测。返回的字符串不以空字符结尾。（如果想加上结尾空字符，务必让传入的 bufsize 比实际可用空间小一。）

- PQputline 向后台服务器发送一个以空字符结尾的字符串。成功返回 0，无法发送字符串返回 EOF。

  ```
  int PQputline(PGconn *conn,
                const char *string);
  ```

  注意，应用必须在最后一行显式发送两个字符 "\."，以告知后端它已发送完数据。

- PQputnbytes 向后台服务器发送一个不以空字符结尾的字符串。成功返回 0，无法发送字符串返回 EOF。

  ```
  int PQputnbytes(PGconn *conn,
                  const char *buffer,
                  int nbytes);
  ```

  它与 PQputline 完全一样，只是由于要发送的字节数是直接指定的，数据缓冲区不需要以空字符结尾。

- PQendcopy 与后端同步。此函数等待后端完成 copy。它应当在用 PQputline 向后端发送完最后一个字符串之后，或用 PGgetline 从后端收到最后一个字符串之后发出。必须发出此函数，否则后端可能与前端"out of sync"（失去同步）。从此函数返回后，后端就准备好接收下一条查询了。成功完成时返回值为 0，否则非零。

  ```
  int PQendcopy(PGconn *conn);
  ```

  例如：

  ```
  PQexec(conn, "create table foo (a int4, b char(16), d float8)");
  PQexec(conn, "copy foo from stdin");
  PQputline(conn, "3\thello world\t4.5\n");
  PQputline(conn,"4\tgoodbye world\t7.11\n");
  ...
  PQputline(conn,"\\.\n");
  PQendcopy(conn);
  ```

使用 PQgetResult 时，应用对 PGRES_COPY_OUT 结果的响应应是反复执行 PQgetline，看到终结行后再执行 PQendcopy。然后应回到 PQgetResult 循环，直到 PQgetResult 返回 NULL。类似地，PGRES_COPY_IN 结果由一系列 PQputline 调用加 PQendcopy 处理，然后回到 PQgetResult 循环。这样的安排可确保嵌入在一系列 SQL 命令中的 copy in 或 copy out 命令被正确执行。

较老的应用可能通过 PQexec 提交 copy in 或 copy out，并认为 PQendcopy 之后事务就完成了。只有当 copy in/out 是查询字符串中唯一的 SQL 命令时，这样做才是正确的。

53.7. libpq 跟踪函数

- PQtrace 把前端/后端通信的跟踪打开到调试文件流。

  ```
  void PQtrace(PGconn *conn
               FILE *debug_port)
  ```

- PQuntrace 关闭由 PQtrace 启动的跟踪。

  ```
  void PQuntrace(PGconn *conn)
  ```

53.8. libpq 控制函数

- PQsetNoticeProcessor 控制 libpq 产生的通知和警告消息的报告方式。

  ```
  typedef void (*PQnoticeProcessor) (void *arg, const char *message);

  PQnoticeProcessor
  PQsetNoticeProcessor(PGconn *conn,
                       PQnoticeProcessor proc,
                       void *arg);
  ```

默认情况下，libpq 把来自后端的"notice"消息以及它自己产生的少量错误消息打印到 stderr。可以通过提供一个对这些消息另做处理的回调函数来改变这一行为。
回调函数会收到错误消息的文本（包括末尾换行符），外加一个与传给 PQsetNoticeProcessor 的指针相同的 void 指针。（需要时可以用该指针访问应用特定的状态。）默认的通知处理器只是

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

要使用特定的通知处理器，只需在创建新的 PGconn 对象之后立即调用 PQsetNoticeProcessor。

返回值是指向前一个通知处理器的指针。如果传入 NULL 回调函数指针，则不采取任何动作，但仍返回当前指针。

设置通知处理器之后，应当预期：只要 PGconn 对象或由它生成的 PGresult 对象仍存在，就可能调用该函数。创建 PGresult 时，PGconn 当前的通知处理器指针会被复制到 PGresult 中，供 PQgetvalue 等例程在需要时使用。

53.9. 环境变量

下面的环境变量可用于选择默认的连接参数值；当调用代码没有直接指定值时，PQconnectdb 或 PQsetdbLogin 将使用这些值。它们有助于避免把数据库名硬编码进简单的应用程序。

- PGHOST 设置默认的服务器名。如果指定了非零长度的字符串，则使用 TCP/IP 通信。没有主机名时，libpq 将使用本地 Unix 域套接字连接。
- PGPORT 设置与 Postgres 后端通信所用的默认端口或本地 Unix 域套接字文件扩展名。
- PGDATABASE 设置默认的 Postgres 数据库名。
- PGUSER 设置连接数据库和认证所用的用户名。
- PGPASSWORD 设置当后端要求密码认证时使用的密码。
- PGREALM 设置与 Postgres 一起使用的 Kerberos realm（当它与本地 realm 不同时）。如果设置了 PGREALM，Postgres 应用将尝试对该 realm 中的服务器进行认证，并使用单独的 ticket 文件以避免与本地 ticket 文件冲突。仅当后端选择了 Kerberos 认证时才会使用此环境变量。
- PGOPTIONS 设置 Postgres 后端的附加运行时选项。
- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。

下面的环境变量可用于为每个 Postgres 会话指定用户级的默认行为：

- PGDATESTYLE 设置日期/时间表示的默认风格。
- PGTZ 设置默认时区。
- PGCLIENTENCODING 设置默认的客户端编码（如果配置 Postgres 时选择了 MULTIBYTE 支持）。

下面的环境变量可用于为每个 Postgres 会话指定默认的内部行为：

- PGGEQO 设置遗传优化器的默认模式。

关于这些环境变量的正确取值，请参阅 SET SQL 命令。

53.10. 线程行为

从 Postgres 7.0 起，libpq 是线程安全的，前提是没有两个线程同时尝试操作同一个 PGconn 对象。特别是，不能通过同一个连接对象从不同线程发出并发的查询。（如果需要运行并发查询，请建立多个连接。）

PGresult 对象在创建后是只读的，因此可以在线程之间自由传递。

已废弃的函数 PQoidStatus 和 fe_setauthsvc 不是线程安全的，不应在多线程程序中使用。PQoidStatus 可以由 PQoidValue 代替。而调用 fe_setauthsvc 则根本没有正当理由。

53.11. 示例程序

53.11.1. 示例程序 1

```
/*
 * testlibpq.c Test the C version of Libpq, the Postgres frontend
 * library.
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char        *pghost,
                *pgport,
                *pgoptions,
                *pgtty;
    char        *dbName;
    int          nFields;
    int          i,
                 j;

    /* FILE *debug; */

    PGconn      *conn;
    PGresult    *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
```

```
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = "template1";

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from pg_
database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
```

```
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
 properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the instances */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    /* fclose(debug); */
    return 0;

}
```

53.11.2. 示例程序 2

```
/*
 * testlibpq2.c
 *  Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
```

```
 * Populate a database with the following:
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 values (new.i); NOTIFY TBL2);
 *
 * and do
 *
 *   INSERT INTO TBL1 values (10);
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
```

```
    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d' received
\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }
```

```
    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

53.11.3. 示例程序 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d float4, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0,
 * 2.0)'::polygon);
 *
 * INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0,
 * 1.0)'::polygon);
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
```

```
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */

    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
```

```
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select *
from test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
```

```
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum),
 plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
        printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo =
 %f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

第 54 章 libpq - C++ 绑定库

libpq++ 是 Postgres 的 C++ API。libpq++ 是一组允许客户端程序连接到 Postgres 后端服务器的类。这些连接有两种形式：数据库类和大对象类。

数据库类用于操作数据库。你可以向 Postgres 后端服务器发送各种 SQL 查询并获取服务器的响应。

大对象类用于操作数据库中的大对象。虽然大对象实例可以向 Postgres 后端服务器发送普通查询，但它只适用于不返回任何数据的简单查询。应当把大对象看作一个文件流。
将来它应当表现得非常像 C++ 文件流 cin、cout 和 cerr。

本章基于 libpq C 库的文档。本节末尾列出了三个短程序作为 libpq++ 编程的示例（但不一定是良好编程的示例）。源代码发行版的 src/libpq++/examples 中有若干 libpq++ 应用的示例，包括本章三个示例的源代码。

54.1. 控制与初始化

54.1.1. 环境变量

下列环境变量可用于为环境设置默认值，避免把数据库名硬编码到应用程序中：

> **注意**
>
> 完整的可用连接选项列表见libpq - C 库。

下列环境变量可用于选择默认的连接参数值，如果调用代码没有直接指定值，PQconnectdb 或 PQsetdbLogin 将使用它们。这些变量有助于避免在简单的应用程序中硬编码数据库名。

> **注意**
>
> libpq++ 只使用环境变量或 PQconnectdb conninfo 风格字符串。

- PGHOST 设置默认的服务器名。如果指定了非零长度的字符串，则使用 TCP/IP 通信。如果没有主机名，libpq 将使用本地 Unix 域套接字连接。
- PGPORT 设置与 Postgres 后端通信的默认端口或本地 Unix 域套接字文件扩展名。
- PGDATABASE 设置默认的 Postgres 数据库名。
- PGUSER 设置用于连接数据库和进行认证的用户名。
- PGPASSWORD 设置在后端要求密码认证时使用的密码。
- PGREALM 设置与 Postgres 一起使用的 Kerberos 域（如果它与本地域不同）。如果设置了 PGREALM，Postgres 应用将尝试向该域的服务器认证，并使用单独的票据文件以避免与本地票据文件冲突。只有在后端选择了 Kerberos 认证时才会使用此环境变量。

- PGOPTIONS 为 Postgres 后端设置额外的运行时选项。
- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。

下列环境变量可用于为每个 Postgres 会话指定用户级的默认行为：

- PGDATESTYLE 设置日期/时间表示的默认风格。
- PGTZ 设置默认时区。

下列环境变量可用于为每个 Postgres 会话指定默认的内部行为：

- PGGEQO 设置遗传优化器的默认模式。

关于这些环境变量的正确取值，请参阅 SQL 命令 SET。

54.2. libpq++ 类

54.2.1. 连接类：PgConnection

连接类实际建立与数据库的连接，并被所有访问类继承。

54.2.2. 数据库类：PgDatabase

数据库类提供了带有到后端服务器连接的 C++ 对象。要创建这样的对象，首先需要可供后端访问的合适环境。下列构造函数负责从 C++ 程序建立到后端服务器的连接。

54.3. 数据库连接函数

- PgConnection 建立到后端数据库服务器的新连接。

```
PgConnection::PgConnection(const char *conninfo)
```

  虽然通常是从某个访问类中调用，但也可以通过创建 PgConnection 对象来建立到后端服务器的连接。

- ConnectionBad 返回到后端服务器的连接是成功还是失败。

```
int PgConnection::ConnectionBad()
```

  如果连接失败则返回 TRUE。

- Status 返回到后端服务器的连接的状态。

```
ConnStatusType PgConnection::Status()
```

  根据连接的状态返回 CONNECTION_OK 或 CONNECTION_BAD。

- PgDatabase 建立到后端数据库服务器的新连接。

```
PgDatabase(const char *conninfo)
```

在创建了 PgDatabase 之后，应当先确认到数据库的连接已经成功，然后再向该对象发送查询。这很容易做到：用 `Status` 或 `ConnectionBad` 方法获取 PgDatabase 对象的当前状态。

- `DBName` 返回当前数据库的名字。

```
const char *PgConnection::DBName()
```

- `Notifies` 从后端收到的未处理通知消息列表中返回下一条通知。

```
PGnotify* PgConnection::Notifies()
```

细节见 PQnotifies()。

54.4. 查询执行函数

- `Exec` 向后端服务器发送一条查询。通常更可取的是使用下面两个函数之一。

```
ExecStatusType PgConnection::Exec(const char* query)
```

返回查询的结果。可能得到下列状态值：

PGRES_EMPTY_QUERY
PGRES_COMMAND_OK（查询是一条命令时）
PGRES_TUPLES_OK（查询成功返回元组时）
PGRES_COPY_OUT
PGRES_COPY_IN
PGRES_BAD_RESPONSE（收到意外响应时）
PGRES_NONFATAL_ERROR
PGRES_FATAL_ERROR

- `ExecCommandOk` 向后端服务器发送一条命令查询。

```
int PgConnection::ExecCommandOk(const char *query)
```

命令查询成功时返回 TRUE。

- `ExecTuplesOk` 向后端服务器发送一条命令查询。

```
int PgConnection::ExecTuplesOk(const char *query)
```

命令查询成功时返回 TRUE。

- `ErrorMessage` 返回最后一条错误消息的文本。

```
const char *PgConnection::ErrorMessage()
```

- `Tuples` 返回查询结果中元组（实例）的数目。

```
int PgDatabase::Tuples()
```

- CmdTuples 返回 INSERT、UPDATE 或 DELETE 之后受影响的行数。如果是其他命令，返回 -1。

```
int PgDatabase::CmdTuples()
```

- Fields 返回查询结果中每个元组的字段（属性）数目。

```
int PgDatabase::Fields()
```

- FieldName 返回与给定字段索引相关联的字段（属性）名。字段索引从 0 开始。

```
const char *PgDatabase::FieldName(int field_num)
```

- FieldNum PQfnumber 返回与给定字段名相关联的字段（属性）索引。

```
int PgDatabase::FieldNum(const char* field_name)
```

如果给定的名称不匹配任何字段，则返回 -1。

- FieldType 返回与给定字段索引相关联的字段类型。返回的整数是该类型的内部编码。字段索引从 0 开始。

```
Oid PgDatabase::FieldType(int field_num)
```

- FieldType 返回与给定字段名相关联的字段类型。返回的整数是该类型的内部编码。字段索引从 0 开始。

```
Oid PgDatabase::FieldType(const char* field_name)
```

- FieldSize 返回与给定字段索引相关联的字段的字节大小。字段索引从 0 开始。

```
short PgDatabase::FieldSize(int field_num)
```

返回数据库元组中为该字段分配的空间（按字段编号给出）。换句话说，
就是服务器上该数据类型二进制表示的大小。如果字段是变长的，则返回 -1。

- FieldSize 返回与给定字段索引相关联的字段的字节大小。字段索引从 0 开始。

```
short PgDatabase::FieldSize(const char *field_name)
```

返回数据库元组中为该字段分配的空间（按字段名给出）。换句话说，
就是服务器上该数据类型二进制表示的大小。如果字段是变长的，则返回 -1。

- GetValue 返回 PGresult 中某个元组的单个字段（属性）值。元组和字段索引从 0 开始。

```
const char *PgDatabase::GetValue(int tup_num, int field_num)
```

对大多数查询来说，GetValue 返回的是属性值的以空字符结尾的 ASCII 字符串表示。但如果 BinaryTuples() 为 TRUE，GetValue 返回的是该类型以后端服务器内部格式表示的二进制表示（如果字段是变长的，则不包括长度字）。这时由程序员负责把数据造型和转换为正确的 C 类型。GetValue 返回的指针指向 PGresult 结构一部分的存储。不应修改它，如果要在 PGresult 结构自身的生存期之后使用该值，必须把它显式复制到其他存储中。BinaryTuples() 尚未实现。

- GetValue 返回 PGresult 中某个元组的单个字段（属性）值。元组和字段索引从 0 开始。

```
 const char *PgDatabase::GetValue(int tup_num, const char *field_
name)
```

对大多数查询来说，GetValue 返回的是属性值的以空字符结尾的 ASCII 字符串表示。但如果 BinaryTuples() 为 TRUE，GetValue 返回的是该类型以后端服务器内部格式表示的二进制表示（如果字段是变长的，则不包括长度字）。这时由程序员负责把数据造型和转换为正确的 C 类型。GetValue 返回的指针指向 PGresult 结构一部分的存储。不应修改它，如果要在 PGresult 结构自身的生存期之后使用该值，必须把它显式复制到其他存储中。BinaryTuples() 尚未实现。

- GetLength 返回字段（属性）的字节长度。元组和字段索引从 0 开始。

```
 int PgDatabase::GetLength(int tup_num, int field_num)
```

这是该数据值的实际数据长度，即 GetValue 所指向对象的大小。注意，对于以 ASCII 表示的值，这个大小与 PQfsize 报告的二进制大小关系不大。

- GetLength 返回字段（属性）的字节长度。元组和字段索引从 0 开始。

```
 int PgDatabase::GetLength(int tup_num, const char* field_name)
```

这是该数据值的实际数据长度，即 GetValue 所指向对象的大小。注意，对于以 ASCII 表示的值，这个大小与 PQfsize 报告的二进制大小关系不大。

- DisplayTuples 把所有元组以及可选的属性名打印到指定的输出流。

```
 void PgDatabase::DisplayTuples(FILE *out = 0, int fillAlign = 1,
 const char* fieldSep = "|",int printHeader = 1, int quiet = 0)
```

- PrintTuples 把所有元组以及可选的属性名打印到指定的输出流。

```
 void PgDatabase::PrintTuples(FILE *out = 0, int printAttName = 1,
 int terseOutput = 0, int width = 0)
```

- GetLine

```
 int PgDatabase::GetLine(char* string, int length)
```

- PutLine

```
 void PgDatabase::PutLine(const char* string)
```

- OidStatus

  ```
  const char *PgDatabase::OidStatus()
  ```

- EndCopy

  ```
  int PgDatabase::EndCopy()
  ```

54.5. 异步通知

Postgres 通过 LISTEN 和 NOTIFY 命令支持异步通知。后端用 LISTEN 命令注册它对某个被命名信号量的关注。当另一个后端执行同名的 NOTIFY 时，所有正在监听该被命名信号量的后端都会被异步通知。通知者不会向监听者传递任何额外的信息。因此，
通常任何需要通信的实际数据都是通过关系传递的。

> **注意**
>
> 过去，文档曾把异步通知所用的名称与关系或类关联起来。
> 但实际上实现中这两个概念并没有直接联系，
> 这个被命名的信号量并不需要事先定义一个对应的关系。

libpq++ 每当一个已连接的后端收到异步通知时，应用都会得到通知。但是，从后端到前端的通信并不是异步的。libpq++ 应用必须轮询后端，查看是否有待处理的通知信息。在执行一条查询之后，前端可以调用 PgDatabase::Notifies 查看后端当前是否有可用的通知数据。PgDatabase::Notifies 从后端的未处理通知列表中返回通知。如果没有来自后端的待处理通知，该函数返回 NULL。PgDatabase::Notifies 的行为像弹出一个栈。一旦某条通知被 PgDatabase::Notifies 返回，它就被视为已处理，并将从通知列表中移除。

- PgDatabase::Notifies 从服务器取回待处理的通知。

  ```
  PGnotify* PgDatabase::Notifies()
  ```

第二个示例程序演示了异步通知的用法。

54.6. 与 COPY 命令相关的函数

Postgres 中的 copy 命令有一些选项，可以读取或写入 libpq++ 所使用的网络连接。因此，需要一些函数来直接访问这个网络连接，使应用可以充分利用这一能力。

- PgDatabase::GetLine 把后端服务器传输的一行以换行符结尾的字符读入大小为 *length* 的缓冲区 *string*。

  ```
  int PgDatabase::GetLine(char* string, int length)
  ```

  与 Unix 系统例程 fgets (3) 一样，此例程最多复制 *length*-1 个字符到 *string*。但它又像 gets (3)，会把结尾的换行符转换为一个空字符。

PgDatabase::GetLine 在文件末尾返回 EOF，整行已读取时返回 0，而缓冲区已满但尚未读到终止换行符时返回 1。

注意，应用程序必须检查新行是否由单个句点（"."）组成，这表示后端服务器已经完成了 copy 结果的发送。因此，如果应用预期会收到长度超过 *length*-1 个字符的行，务必非常仔细地检查 PgDatabase::GetLine 的返回值。

- PgDatabase::PutLine 向后端服务器发送一个以空字符结尾的 *string*。

```
void PgDatabase::PutLine(char* string)
```

应用程序必须显式发送单个句点字符（"."），向后端表明它已完成数据的发送。

- PgDatabase::EndCopy 与后端同步。

```
int PgDatabase::EndCopy()
```

该函数等待后端完成对 copy 的处理。无论是用 PgDatabase::PutLine 向后端发送完最后一个字符串，还是用 PgDatabase::GetLine 从后端接收完最后一个字符串之后，都应当调用它。必须调用它，否则后端可能与前端"失去同步"。该函数返回后，
后端就准备好接收下一条查询了。

成功完成时返回值为 0，否则为非零。

例如：

```
PgDatabase data;
data.Exec("create table foo (a int4, b char(16), d float8)");
data.Exec("copy foo from stdin");
data.PutLine("3\tHello World\t4.5\n");
data.PutLine("4\tGoodbye World\t7.11\n");
&...
data.PutLine("\\.\n");
data.EndCopy();
```

第 55 章 pgtcl - TCL 绑定库

`pgtcl` 是一个 tcl 包，供前端程序与 Postgres 后端交互。它使 tcl 脚本可以使用 `libpq` 的大部分功能。

本包最初由 Jolly Chen 编写。

55.1. 命令

表 55.1. `pgtcl` 命令

命令	描述
pg_connect	打开一个到后端服务器的连接
pg_disconnect	关闭一个连接
pg_conndefaults	获取连接选项及其默认值
pg_exec	向后端发送一个查询
pg_result	操作查询的结果
pg_select	循环遍历 SELECT 语句的结果
pg_listen	为 NOTIFY 消息建立回调
pg_lo_creat	创建一个大对象
pg_lo_open	打开一个大对象
pg_lo_close	关闭一个大对象
pg_lo_read	读取一个大对象
pg_lo_write	写入一个大对象
pg_lo_lseek	在大对象中定位到某个位置
pg_lo_tell	返回大对象的当前寻位位置
pg_lo_unlink	删除一个大对象
pg_lo_import	把 Unix 文件导入为大对象
pg_lo_export	把大对象导出到 Unix 文件

这些命令将在后续页面中进一步描述。

pg_lo* 例程是 Postgres 大对象特性的接口。这些函数的设计模仿标准 Unix 文件系统接口中的类似文件系统函数。pg_lo* 例程应当在 BEGIN/END 事务块内使用，因为 pg_lo_open 返回的文件描述符只在当前事务内有效。pg_lo_import 和 pg_lo_export 必须在 BEGIN/END 事务块内使用。

55.2. 示例

下面是使用这些例程的一个小例子：

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
```

```
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY
 datname"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

55.3. pgtcl 命令参考信息

pg_connect

pg_connect — 打开一个到后端服务器的连接

大纲

```
pg_connect -conninfo connectOptions
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty]
  [-options optionalBackendArgs]
```

输入（新风格）

connectOptions

一个连接选项字符串，每个选项都写成 keyword = value 的形式。

输入（旧风格）

dbName

指定一个有效的数据库名。

[-host *hostName*]

指定 *dbName* 所在后端服务器的主机名。

[-port *portNumber*]

指定 *dbName* 所在后端服务器的 IP 端口号。

[-tty *pqtty*]

指定用于后端可选调试输出的文件或 tty。

[-options *optionalBackendArgs*]

指定 *dbName* 所在后端服务器的选项。

输出

dbHandle

成功时返回一个数据库连接的句柄。句柄以 "pgsql" 前缀开头。

描述

pg_connect 打开一个到 Postgres 后端的连接。

有两种可用的语法。在较老的一种中，每个可能的选项在 pg_connect 语句中都有一个单独的选项开关。在较新的一种中，提供的是单个选项字符串，其中可以包含多个选项值。
关于较新语法中可用选项的信息参见 pg_conndefaults。

用法

XXX thomas 1997-12-24

pg_disconnect

pg_disconnect — 关闭一个到后端服务器的连接

大纲

```
pg_disconnect dbHandle
```

输入

dbHandle

指定一个有效的数据库句柄。

输出

None

描述

`pg_disconnect` 关闭一个到 Postgres 后端的连接。

pg_conndefaults

pg_conndefaults — 获取关于默认连接参数的信息

大纲

```
pg_conndefaults
```

输入

无。

输出

option list

结果是一个列表，描述可能的连接选项及其当前默认值。
列表中的每个条目是如下格式的子列表：

{optname label dispchar dispsize value}

其中 optname 可用作 `pg_connect -conninfo` 中的一个选项。

描述

`pg_conndefaults` 返回关于 `pg_connect -conninfo` 中可用的连接选项及每个选项当前默认值的信息。

用法

pg_conndefaults

pg_exec

pg_exec — 向后端发送一个查询字符串

大纲

```
pg_exec dbHandle queryString
```

输入

dbHandle

指定一个有效的数据库句柄。

queryString

指定一个有效的 SQL 查询。

输出

resultHandle

如果 Pgtcl 未能获得后端响应，将返回一个 Tcl 错误。否则，会创建一个查询结果对象并返回其句柄。可以把这个句柄传给 `pg_result` 以获取查询的结果。

描述

`pg_exec` 向 Postgres 后端提交一个查询并返回结果。查询结果句柄以连接句柄开头，再加上一个句点和一个结果编号。

注意，没有 Tcl 错误并不证明查询成功！后端返回的错误消息会作为带有失败状态的查询结果被处理，而不是让 pg_exec 产生 Tcl 错误。

pg_result

pg_result — 获取关于查询结果的信息

大纲

```
pg_result resultHandle resultOption
```

输入

resultHandle

一个查询结果的句柄。

resultOption

指定若干可能选项之一。

选项

-status

结果的状态。

-error

错误消息（如果状态表示出错）；否则为空字符串。

-conn

产生该结果的连接。

-oid

如果命令是 INSERT，为所插入元组的 OID；否则为空字符串。

-numTuples

查询返回的元组数。

-numAttrs

每个元组中的属性数。

-list VarName

把结果赋给一个列表的列表。

-assign arrayName

把结果赋给一个数组，使用形如 (tupno,attributeName)的下标。

-assignbyidx arrayName ?appendstr?

把结果赋给一个数组，用第一个属性的值和其余属性的名作为键。如果给出了 *appendstr*，则把它追加到每个键之后。简而言之，每个元组中除第一个字段外的所有字段都存入数组，使用形如 (firstFieldValue,fieldNameAppendStr)的下标。

-getTuple tupleNumber

以列表形式返回所指元组的各字段。元组编号从零开始。

-tupleArray tupleNumber arrayName

把该元组的各字段存入数组 *arrayName*，以字段名为索引。元组编号从零开始。

-attributes

返回元组各属性名的列表。

-lAttributes

返回子列表的列表，每个元组属性对应一个 {name ftype fsize}。

-clear

清除该查询结果对象。

输出

结果取决于所选的选项，如上所述。

描述

`pg_result` 返回关于由先前的 `pg_exec` 创建的查询结果的信息。

查询结果可以保留任意长的时间，但用完之后，务必通过执行 `pg_result -clear` 将其释放。否则会造成内存泄漏，Pgtcl 最终会抱怨你创建了太多查询结果对象。

pg_select

pg_select — 循环遍历 SELECT 语句的结果

大纲

```
pg_select dbHandle queryString
  arrayVar queryProcedure
```

输入

dbHandle

指定一个有效的数据库句柄。

queryString

指定一个有效的 SQL select 查询。

arrayVar

用于存放返回元组的数组变量。

queryProcedure

对找到的每个元组运行的过程。

输出

resultHandle

返回结果是一个错误消息或一个查询结果的句柄。

描述

pg_select 向 Postgres 后端提交一个 SELECT 查询，并对结果中的每个元组执行一段给定的代码。*queryString* 必须是一条 SELECT 语句。其他语句都会返回错误。*arrayVar* 变量是循环中使用的数组名。对每个元组，*arrayVar* 都会以字段名作为数组下标填入该元组的字段值。然后执行 *queryProcedure*。

用法

如果表 "table" 有字段 "control" 和 "name"（或许还有其他字段），下面这样写即可：

```
pg_select $pgconn "SELECT * from table" array {
 puts [format "%5d %s" array(control) array(name)]
}
```

pg_listen

pg_listen — 设置或更改异步 NOTIFY 消息的回调

大纲

```
pg_listen dbHandle notifyName callbackCommand
```

输入

dbHandle

指定一个有效的数据库句柄。

notifyName

指定要开始或停止监听的 notify 条件名。

callbackCommand

如果提供且非空，给出匹配的通知到达时要执行的命令字符串。

输出

None

描述

pg_listen 创建、更改或取消对来自 Postgres 后端的异步 NOTIFY 消息的监听请求。带 *callbackCommand* 参数时，会建立请求，或替换已存在请求的命令字符串。不带 *callbackCommand* 参数时，则取消先前的请求。

pg_listen 请求建立之后，每当带有给定名称的 NOTIFY 消息从后端到达时，就执行指定的命令字符串。任何 Postgres 客户端应用程序发出引用该名称的 NOTIFY 命令时都会发生这种情况。（注意，该名称可以是数据库中已存在关系的名称，但不必一定如此。）命令字符串在 Tcl 空闲循环中执行。这是用 Tk 编写的应用程序的正常空闲状态。在非 Tk 的 Tcl shell 中，可以执行 update 或 vwait 来进入空闲循环。

使用 pg_listen 时，不应直接调用 SQL 语句 LISTEN 或 UNLISTEN。Pgtcl 会替你发出这些语句。但如果想自己发送 NOTIFY 消息，可以用 pg_exec 调用 SQL 的 NOTIFY 语句。

pg_lo_creat

pg_lo_creat — 创建一个大对象

大纲

```
pg_lo_creat conn mode
```

输入

conn

指定一个有效的数据库连接。

mode

指定大对象的访问模式

输出

objOid

所创建大对象的 oid。

描述

`pg_lo_creat` 创建一个 Inversion 大对象。

用法

mode 可以是 INV_READ、INV_WRITE、INV_ARCHIVE 的任意 OR 组合。OR 分隔字符是 "|"。

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

pg_lo_open

pg_lo_open — 打开一个大对象

大纲

```
pg_lo_open conn objOid mode
```

输入

conn

指定一个有效的数据库连接。

objOid

指定一个有效的大对象 oid。

mode

指定大对象的访问模式

输出

fd

供后续 pg_lo* 例程使用的文件描述符。

描述

`pg_lo_open` 打开一个 Inversion 大对象。

用法

模式可以是 "r"、"w" 或 "rw"。

pg_lo_close

pg_lo_close — 关闭一个大对象

大纲

```
pg_lo_close conn fd
```

输入

conn

指定一个有效的数据库连接。

fd

供后续 pg_lo* 例程使用的文件描述符。

输出

无

描述

`pg_lo_close` 关闭一个 Inversion 大对象。

用法

pg_lo_read

pg_lo_read — 读取一个大对象

大纲

```
pg_lo_read conn fd bufVar len
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

bufVar

指定一个有效的缓冲区变量，用于容纳大对象片段。

len

指定大对象片段的最大允许大小。

输出

无

描述

`pg_lo_read` 从大对象中读取至多 *len* 字节到名为 *bufVar* 的变量中。

用法

bufVar 必须是有效的变量名。

pg_lo_write

pg_lo_write — 写入一个大对象

大纲

```
pg_lo_write conn fd buf len
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

buf

指定要写入大对象的有效字符串变量。

len

指定要写入字符串的最大大小。

输出

无

描述

`pg_lo_write` 从变量 *buf* 向大对象写入至多 *len* 字节。

用法

buf 必须是要写入的实际字符串，而不是变量名。

pg_lo_lseek

pg_lo_lseek — 在大对象中定位到某个位置

大纲

```
pg_lo_lseek conn fd offset whence
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

offset

指定以字节计的从零开始的偏移量。

whence

whence 可以是 "SEEK_CUR"、"SEEK_END"、"SEEK_SET"

输出

无

描述

`pg_lo_lseek` 定位到距大对象开头 *`offset`* 字节处。

用法

`whence` 可以是 "SEEK_CUR"、"SEEK_END" 或 "SEEK_SET"。

pg_lo_tell

pg_lo_tell — 返回大对象的当前寻位位置

大纲

```
pg_lo_tell conn fd
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

输出

offset

以字节计的从零开始的偏移量，适合作为 `pg_lo_lseek` 的输入。

描述

`pg_lo_tell` 返回当前距大对象开头的 *offset*（以字节计）。

用法

pg_lo_unlink

pg_lo_unlink — 删除一个大对象

大纲

```
pg_lo_unlink conn lobjId
```

输入

conn

指定一个有效的数据库连接。

lobjId

大对象的标识符。XXX Is this the same as objOid in other calls?? - thomas 1998-01-11

输出

无

描述

`pg_lo_unlink` 删除指定的大对象。

用法

pg_lo_import

pg_lo_import — 从文件导入一个大对象

大纲

```
pg_lo_import conn filename
```

输入

conn

指定一个有效的数据库连接。

filename

Unix 文件名。

输出

无 XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

描述

`pg_lo_import` 读取指定的文件并把内容放入一个大对象。

用法

`pg_lo_import` 必须在 BEGIN/END 事务块内调用。

pg_lo_export

pg_lo_export — 把大对象导出到文件

大纲

```
pg_lo_export conn lobjId filename
```

输入

conn

指定一个有效的数据库连接。

lobjId

大对象标识符。XXX Is this the same as the objOid in other calls?? thomas - 1998-01-11

filename

Unix 文件名。

输出

无 XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

描述

`pg_lo_export` 把指定的大对象写入一个 Unix 文件。

用法

`pg_lo_export` 必须在 BEGIN/END 事务块内调用。

第 56 章 libpgeasy - 简化 C 绑定库

作者

由 Bruce Momjian 编写（pgman@candle.pha.pa.us[1]），最后更新于 2000-03-30。

pgeasy 让你能够干净地接口到 libpq 库，更像是 4GL 的 SQL 接口。

它由一组封装了 libpq 功能的简化 C 函数组成。这些函数是：

- PGresult *doquery(char *query);
- PGconn *connectdb(char *options);
- void disconnectdb();
- int fetch(void *param,...);
- int fetchwithnulls(void *param,...);
- void reset_fetch();
- void on_error_continue();
- void on_error_stop();
- PGresult *get_result();
- void set_result(PGresult *newres);
- void unset_result(PGresult *oldres);

许多函数会返回一个结构或值，因此在需要时你可以对结果做更多的工作。

基本上你用 `connectdb` 连接到数据库，用 `doquery` 发出查询，用 `fetch` 取回结果，最后用 `disconnectdb` 结束。

对于 `SELECT` 查询，`fetch` 允许你把指针作为参数传递，返回时这些变量将被你打开的二进制游标中的数据填充。如果你运行 pgeasy 客户端的系统与数据库服务器的体系结构不同，这些二进制游标就无法使用。如果你传递一个 NULL 指针参数，该列会被跳过。`fetchwithnulls` 允许你取回字段的 NULL 状态，方法是在每个结果指针之后传递一个 `int*`，字段为空时它返回真或假。你总是可以在 `doquery` 返回的 PGresult 指针上使用 libpq 函数。`reset_fetch` 把取回操作重置回开头。

`get_result`、`set_result` 和 `unset_result` 允许你同时处理多个结果集。

源代码目录中有各种各样的示例程序。

[1] mailto:pgman@candle.pha.pa.us

第 57 章 ODBC 接口

Tim Goeke
Thomas Lockhart

> **注意**
>
> 背景信息最初由 Tim Goeke[1] 提供

ODBC（开放数据库互连）是一个抽象的 API，它让你能够编写可与各种 RDBMS 服务器互操作的应用。ODBC 在前端应用和数据库服务器之间提供了一个与产品无关的接口，让用户或开发者能够编写在不同厂商的服务器之间可移植（可传输）的应用。

57.1. 背景

ODBC API在后端对应一个 ODBC兼容的数据源。它可以是任何东西，从一个文本文件到 Oracle或Postgres RDBMS。

后端访问来自ODBC驱动，或者说来自允许访问数据的厂商特定驱动。
psqlODBC就是这样一种驱动，此外还有其他可用的驱动，例如 OpenLink 的 ODBC驱动。

一旦你编写了一个ODBC应用，只要数据库模式相同，无论后端数据库来自哪个厂商，你应该都能连接到任何后端数据库。

例如，你可以有一台MS SQL Server服务器和一台数据完全相同的Postgres服务器。使用ODBC，你的 Windows 应用发出的调用将完全相同，后端数据源（在 Windows 应用看来）看起来也一样。

57.2. Windows应用

在现实世界中，驱动之间的差异以及ODBC支持水平的不同削弱了ODBC的潜力：

- Access、Delphi 和 Visual Basic 都直接支持ODBC。
- 在 C++（例如 Visual C++）下，你可以使用 C++ 的 ODBC API。
- 在 Visual C++ 中，你可以使用 CRecordSet 类，它把 ODBC API集封装在一个 MFC 4.2 类中。如果你在 Windows NT 下进行 Windows C++ 开发，这是最容易的路线。

57.2.1. 编写应用

“如果我为Postgres编写一个应用，我能否通过ODBC调用访问 Postgres服务器来编写它，还是说只有当 MS SQL Server 或 Access 之类的其他数据库程序需要访问数据时才这么做？”

ODBC API正是该走的路。关于Visual C++编程，你可以在 Microsoft 的网站或你的VC++文档中找到更多信息。

Visual Basic 和其他 RAD 工具有 Recordset 对象，它们直接使用 ODBC访问数据。使用数据感知控件，你可以（非常迅速地）快速链接到 ODBC后端数据库。

玩玩 MS Access 会帮你弄清楚这一切。试着使用 `File->Get External Data`。

[1] mailto:tgoeke@xpressway.com

提示

你必须先设置一个 DSN。

57.3. Unix 安装

ApplixWare有一个至少在部分平台上受到支持的ODBC数据库接口。在 Linux 下，已经演示过 ApplixWare v4.4.2 通过 Postgres发行版中包含的 psqlODBC驱动与 Postgres v7.0 配合工作。

57.3.1. 构建驱动

关于psqlODBC驱动（或任何 ODBC驱动）首先要注意的一点是：在 ODBC驱动要使用的系统上必须存在一个驱动管理器。Unix 上有一种免费软件的ODBC驱动管理器叫做iodbc，可以从网上多处获取，包括 AS200[2]。安装iodbc的说明超出了本文档的范围，但iodbc压缩 .shar 文件内有一个 `README`，它应该能解释如何让它跑起来。

话虽如此，你能为你的平台找到的任何驱动管理器都应该支持 psqlODBC驱动或任何 ODBC驱动。

psqlODBC的 Unix 配置文件最近被大幅重构，以便在受支持的平台上轻松构建，并为将来支持其他 Unix 平台留出余地。驱动的新配置和构建文件应当让在受支持平台上构建驱动变成一个简单的过程。目前这包括 Linux 和 FreeBSD，但我们希望其他用户能贡献必要的信息，迅速扩大可构建该驱动的平台数量。

实际上有两种构建驱动的方法，取决于你以何种方式得到它，
而这些差别归根结底只是在哪里以及如何运行 configure和make。驱动可以在一个独立的、仅客户端的安装中构建，也可以作为 Postgres主发行版的一部分来构建。如果你在多个异构平台上都有ODBC客户端应用，独立安装更方便。当目标客户端与服务器相同，
或客户端和服务器有相似的运行时配置时，集成安装更方便。

具体来说，如果你得到的psqlODBC驱动是 Postgres发行版的一部分（下文称之为 "集成"构建），那么你将在 Postgres发行版的顶层源目录中与其余库一起配置和构建ODBC驱动。如果你得到的是独立打包的驱动，那么就在你解开驱动源代码的目录中运行 configure 和 make。

集成安装

这个安装过程适用于集成安装。

1. 为src/configure指定 `--with-odbc`命令行参数：

```
% ./configure --with-odbc
% make
```

2. 重新构建Postgres发行版：

```
% make install
```

3. (可选的) 安装`PGROOT/contrib/odbc/odbc.sql`中提供的 ODBC 目录扩展：

[2] http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z

```
% psql -e template1 < $PGROOT/contrib/odbc/odbc.sql
```

把template1指定为目标数据库可以保证之后所有新建的数据库都拥有这些相同的定义。

一旦配置完成，ODBC驱动就会被构建并安装到为 Postgres系统其他组件定义的区域中。安装范围的ODBC配置文件会被放到 Postgres 目标树（POSTGRESDIR）的顶层目录。这可以在 make 命令行上覆盖，即

```
% make ODBCINST=filename install
```

v6.4 之前的集成安装

如果你的Postgres安装早于 v6.4、你手头有原始源代码树、而且你想使用最新版本的ODBC 驱动，那么可以试试这种安装形式。

1. 把输出的 tar 文件复制到目标系统并解开到一个干净的目录。
2. 在包含源代码的目录中输入：

```
% ./configure
% make
% make POSTGRESDIR=PostgresTopDir install
```

3. (可选的) 如果你想把组件安装到不同的树中，可以显式指定各个目的地：

```
% make BINDIR=bindir  LIBDIR=libdir  HEADERDIR=headerdir ODBCINST=
instfile install
```

独立安装

独立安装不与正常的Postgres发行版集成，也不在其上构建。它最适合为多个没有本地安装Postgres源代码树的异构客户端构建 ODBC驱动。

独立安装的库和头文件的默认位置分别是 /usr/local/lib和 /usr/local/include/iodbc。还有另一个系统范围的配置文件，它会被安装为/share/odbcinst.ini（如果/share存在）或 /etc/odbcinst.ini（如果 /share不存在）。

> **注意**
>
> 把文件安装到/share或/etc 需要系统 root 权限。Postgres的大多数安装步骤没有这个要求，你可以改选一个可由你的非 root Postgres超级用户账号写入的其他目的地。

1. 独立安装发行版可以从Postgres发行版构建，也可以从非 Unix 源代码的当前维护者 Insight Distributors[3] 获取。

   把 zip 或 gzip 压缩的 tar 文件复制到一个空目录。如果使用的是 zip 包，用命令解开它

   ```
   % unzip -a packagename
   ```

[3] http://www.insightdist.com/psqlodbc

-a选项是必要的，用于去掉源文件中的 DOS CR/LF 对。

如果你拿到的是 gzip 压缩的 tar 包，只需运行

```
% tar -xzf packagename
```

- (可选的) 要从Postgres主源代码树创建一个完整独立安装的 tar 文件：

2. 配置Postgres主发行版。
3. 创建 tar 文件：

```
% cd interfaces/odbc
% make standalone
```

4. 把输出的 tar 文件复制到目标系统。如果使用 ftp，务必以二进制文件方式传输。
5. 把 tar 文件解开到一个干净的目录。
6. 配置独立安装：

```
% ./configure
```

配置可以带选项进行：

```
% ./configure --prefix=rootdir --with-odbc=inidir
```

其中--prefix把库和头文件安装到目录 *rootdir*/lib和 *rootdir*/include/iodbc，而--with-odbc把odbcinst.ini安装到指定的目录。

注意这两个选项也可以在集成构建中使用，但要小心：在集成构建中使用时，--prefix 也会作用于你的Postgres安装的其余部分。--with-odbc只作用于配置文件 odbcinst.ini。

7. 编译并链接源代码：

```
% make ODBCINST=instdir
```

你也可以在'make'命令行上覆盖安装的默认位置。这只适用于库文件和头文件的安装。由于驱动需要知道 odbcinst.ini 文件的位置，试图覆盖指定其安装目录的环境变量多半会给你带来麻烦。最安全的做法就是让驱动把 odbcinst.ini 文件安装在默认目录，或你在 './configure'命令行上用 --with-odbc 指定的目录。

8. 安装源代码：

```
% make POSTGRESDIR=targettree install
```

要分别覆盖库和头文件的安装目录，你需要在 make install命令行上传入正确的安装变量。这些变量是LIBDIR、HEADERDIR和 ODBCINST。在 make 命令行上覆

盖 POSTGRESDIR会让LIBDIR和 HEADERDIR以你指定的新目录为根。ODBCINST独立于POSTGRESDIR。

下面是显式指定各个目的地的做法：

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

例如，在（用过./configure和 make之后）输入

```
% make POSTGRESDIR=/opt/psqlodbc install
```

会让库和头文件分别安装到/opt/psqlodbc/lib 和/opt/psqlodbc/include/iodbc目录。

而命令

```
% make POSTGRESDIR=/opt/psqlodbc HEADERDIR=/usr/local install
```

应该会让库安装到 /opt/psqlodbc/lib，头文件安装到 /usr/local/include/iodbc。如果结果不如预期，请联系某位维护者。

57.4. 配置文件

~/.odbc.ini 包含用户为 psqlODBC驱动指定的访问信息。该文件使用Windows注册表文件典型的约定，不过尽管有这个限制，还是可以让它工作的。

.odbc.ini文件有三个必需的节。第一个是 [ODBC Data Sources]，它是你想访问的每个数据库的任意名称和描述的列表。第二个必需的节是数据源说明，
每个数据库都会有一个这样的节。每个节必须用 [ODBC Data Sources]中给出的名称标记，并且必须包含下列条目：

```
Driver = POSTGRESDIR/lib/libpsqlodbc.so
Database=DatabaseName
Servername=localhost
Port=5432
```

> **提示**
>
> 记住Postgres数据库名通常是单个词，不带任何形式的路径名。
> Postgres服务器管理对数据库的实际访问，你只需要从客户端指定名字。

还可以插入其他条目来控制显示的格式。第三个必需的节是 [ODBC]，它必须包含InstallDir关键字，还可以包含其他选项。

下面是一个.odbc.ini文件的例子，展示三个数据库的访问信息：

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
```

```
Default = Postgres Stripped

[DataEntry]
ReadOnly = 0
Servername = localhost
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

57.5. ApplixWare

57.5.1. 配置

ApplixWare必须被正确配置，才能访问 Postgres的ODBC 软件驱动。

启用ApplixWare数据库访问

这些说明针对的是Linux上的 ApplixWare `4.4.2` 版。更详细的信息请参阅Linux Sys Admin在线图书。

1. 你必须修改`axnet.cnf`，让 `elfodbc`能找到`libodbc.so`（ODBC驱动管理器）共享库。这个库随 ApplixWare 发行版一起提供，但`axnet.cnf` 需要被修改为指向正确的位置。

   以 root 身份编辑文件 *`applixroot`*`/applix/axdata/axnet.cnf`。

   a. 在`axnet.cnf`的底部，找到以

      ```
      #libFor elfodbc /ax/...
      ```

   b. 把这一行改为

```
libFor elfodbc applixroot/applix/axdata/axshlib/lib
```

这会告诉elfodbc到这个目录中查找 ODBC支持库。通常 Applix安装在 /opt，因此完整路径是 /opt/applix/axdata/axshlib/lib，但如果你把Applix安装在了其他地方，请相应地修改路径。

2. 按照上面的说明创建.odbc.ini。你可能还想加上标志

```
TextAsLongVarchar=0
```

加到.odbc.ini的数据库特定部分，这样文本字段就不会显示为**BLOB**。

测试ApplixWare ODBC 连接

1. 启动Applix Data
2. 选择你感兴趣的Postgres数据库。
   a. 选择Query->Choose Server。
   b. 选择ODBC，然后点击Browse。你在.odbc.ini中配置的数据库应该会被显示出来。确保Host: field为空（如果不为空，axnet 会试图联系另一台机器上的 axnet 来查找数据库）。
   c. 在Browse弹出的框中选择数据库，然后点击OK。
   d. 在登录标识对话框中输入用户名和口令，然后点击OK。

   你应该会在数据窗口左下角看到 "Starting elfodbc server"。如果你得到一个错误对话框，请参见下面的调试一节。
3. 'Ready'消息会出现在数据窗口左下角。这表示你现在可以输入查询了。
4. 从 Query->Choose tables 中选择一张表，然后选择 Query->Query 来访问数据库。表中前 50 行左右应该会出现。

57.5.2. 常见问题

在试图通过Applix Data建立 ODBC连接时可能出现下列消息：

Cannot launch gateway on server

elfodbc找不到libodbc.so。检查你的axnet.cnf。

Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded

libodbc.so找不到 .odbc.ini中列出的驱动。核对设置。

Server: Broken Pipe

驱动进程由于某种其他问题已经终止。你可能没有最新版本的 Postgres ODBC包。

setuid to 256: failed to launch gateway

ApplixWare 4.4.1 的九月版（第一个在 Linux 下正式支持 ODBC的版本）在用户名长度超过八（8）个字符时会出问题。问题描述由 Steve Campbell[4]提供。

作者

由Steve Campbell[5] 于 1998-10-20 供稿。

axnet 程序的安全系统看起来有点可疑。axnet 代表用户做事，而在真正的多用户系统上，它其实应该以 root 权限运行（这样它才能在每个用户的目录中读/写）。不过我不太愿意这样建议，因为我们完全不知道这会带来什么安全漏洞。

57.5.3. 调试ApplixWare ODBC 连接

调试连接问题的一个好工具是 Unix 系统实用程序 strace。

用strace调试

1. 启动 applixware。
2. 对 axnet 进程启动strace。例如，如果

```
% ps -aucx | grep ax
```

shows

```
cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
```

然后运行

```
% strace -f -s 1024 -p 10432
```

3. 检查 strace 输出。

来自 Cary 的提示

ApplixWare的许多错误消息会发往 `stderr`，但我不确定`stderr` 被送到了哪里，所以strace是查明这一点的手段。

例如，在得到"`Cannot launch gateway on server`" 之后，我对 axnet 运行了 strace，得到

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No_such file or directory)
```

[4]mailto:scampbell@lear.com
[5]mailto:scampbell@lear.com

```
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No such file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
can't load library 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

所以发生的事情是 applix elfodbc 在搜索 libodbc.so 但找不到它。这就是为什么需要修改axnet.cnf。

57.5.4. 运行ApplixWare演示

要完成ApplixWare Data Tutorial，你需要创建教程中引用的示例表。用于创建这些表的 ELF 宏试图在许多数据库列上使用 NULL 条件，而Postgres目前不允许这样做。

要绕开这个问题，你可以这样做：

修改ApplixWare演示

1. 把`/opt/applix/axdata/eng/Demos/sqldemo.am`复制到一个本地目录。
2. 编辑这份`sqldemo.am`的本地副本：
   a. 搜索 'null_clause = "NULL"
   b. 把它改为 null_clause = ""
3. 启动Applix Macro Editor。
4. 从Macro Editor中打开 sqldemo.am 文件。
5. 选择`File->Compile and Save`。
6. 退出Macro Editor。
7. 启动Applix Data。
8. 选择`*->Run Macro`
9. 输入值"`sqldemo`"，然后点击`OK`。
   你应该能在数据窗口的状态行（左下角）看到进度。
10. 现在你应该可以访问演示表了。

57.5.5. 有用的宏

你可以把数据库登录和口令信息加到标准的 Applix 启动宏文件中。下面是一个`~/axhome/macros/login.am`文件的例子：

```
macro login
set_set_system_var@("sql_username@","tgl")
set_system_var@("sql_passwd@","no$way")
endmacro
```

小心

对于任何包含用户名和口令信息的文件，你都应当小心设置文件保护。

57.5.6. 支持的平台

psqlODBC已在Linux 上构建并测试过。有报告称在 FreeBSD 和 Solaris 上也成功了。对于其他已经支持Postgres的平台，基本代码没有已知的限制。

第 58 章 JDBC 接口

作者

由 Peter T. Mount[1] 编写，他是 JDBC 驱动的作者。

JDBC 是 Java 1.1 及更高版本的核心 API。它为兼容 SQL 的数据库提供了一组标准接口。

Postgres 提供一个 4 类 JDBC Driver。4 类表示该驱动用纯 Java 编写，并使用数据库自身的网络协议通信。因此，该驱动是平台无关的。一旦编译完成，就可以在任何平台上使用。

58.1. 构建 JDBC 接口

58.1.1. 编译驱动

驱动的源码位于源码树的 `src/interfaces/jdbc` 目录中。要编译，只需进入该目录并输入：

```
% make
```

完成后，你会在当前目录中找到归档文件 `postgresql.jar`。这就是 JDBC 驱动。

注意

你必须使用 make 而不是 javac，因为驱动出于性能原因使用了一些动态加载技术，而 javac 无法应对。`Makefile` 会生成该 jar 归档。

58.1.2. 安装驱动

要使用驱动，需要把 jar 归档 postgresql.jar 包含在 `CLASSPATH` 中。

58.1.2.1. 示例

我有一个应用程序使用 JDBC 驱动访问一个包含天体数据的大型数据库。该应用程序和 jdbc 驱动安装在 /usr/local/lib 目录中，而 java jdk 安装在 /usr/local/jdk1.1.6 中。

要运行该应用程序，我可以使用：

```
export CLASSPATH = /usr/local/lib/finder.jar:/usr/local/lib/
postgresql.jar:.
java uk.org.retep.finder.Main
```

载入驱动的内容将在本章稍后介绍。

[1] peter@retep.org.uk

58.2. 为 JDBC 准备数据库

由于 Java 只能使用 TCP/IP 连接，Postgres postmaster 必须以 -i 标志运行。

另外，必须配置 `pg_hba.conf` 文件。它位于 PGDATA 目录中。在默认安装中，该文件只允许通过 Unix 域套接字访问。要让 JDBC 驱动连接同一个 localhost，你需要加上类似这样的内容：

```
host          all         127.0.0.1     255.255.255.255   password
```

这样，从本地机器就可以用 JDBC 访问所有数据库。

JDBC Driver 支持 trust、ident、password 和 crypt 认证方法。

58.3. 使用驱动

本节并不是一份完整的 JDBC 编程指南，但应该能帮助你入门。更多信息请参阅标准的 JDBC API 文档。另外，也可以看看源码中附带的例子。这里使用的是其中的基本示例。

58.4. 导入 JDBC

任何使用 JDBC 的源码都需要导入 java.sql 包，使用：

```
import java.sql.*;
```

> **重要**
>
> 不要导入 postgresql 包。如果你这样做，你的源代码将无法编译，因为 javac 会产生混淆。

58.5. 载入驱动

在连接数据库之前，你需要载入驱动。有两种可用的方法，哪种最好取决于你的代码。

第一种方法中，你的代码使用 `Class.forName()` 方法隐式地载入驱动。对于 Postgres，你可以使用：

```
Class.forName("postgresql.Driver");
```

这会载入驱动，并且在载入时，驱动会自动向 JDBC 注册自身。

注意：`forName()` 方法可能抛出 `ClassNotFoundException`，所以如果驱动不可用，你需要捕获它。

这是最常用的方法，但它把你的代码限制为只使用 Postgres。如果你的代码将来可能访问另一个数据库，并且你不使用我们的扩展，那么建议使用第二种方法。

第二种方法在 JVM 启动时使用 -D 参数把驱动作为参数传递给它。例如：

```
% java -Djdbc.drivers=postgresql.Driver example.ImageViewer
```

在这个例子中，JVM 会尝试把载入驱动作为其初始化的一部分。完成后，就会启动 ImageViewer。

现在，这种方法是更好的一种，
因为它允许你的代码在不重新编译的情况下与其他数据库一起使用。唯一还需要改变的是 URL，这是接下来要介绍的内容。

最后一点。当你的代码随后尝试打开一个 Connection，并且你得到一个被抛出的 `No driver available` SQLException 时，这可能是驱动不在类路径中，或者参数中的值不正确造成的。

58.6. 连接到数据库

在 JDBC 中，数据库由一个 URL（统一资源定位符）表示。对于 Postgres，它采用下列形式之一：

- jdbc:postgresql:*database*
- jdbc:postgresql://*host*/*database*
- jdbc:postgresql://*hostport*/*database*

其中：

host

服务器的主机名。默认为“localhost”。

port

服务器监听的端口号。默认为 Postgres 的标准端口号（5432）。

database

数据库名。

要连接，你需要从 JDBC 获得一个 Connection 实例。为此，你可以使用 DriverManager.getConnection() 方法：

```
Connection db = DriverManager.getConnection(url,user,pwd);
```

58.7. 发出查询并处理结果

每当你想向数据库发出 SQL 语句时，你都需要一个 Statement 实例。一旦你有了一个 Statement，就可以使用 executeQuery() 方法发出查询。这将返回一个 ResultSet 实例，其中包含全部结果。

58.7.1. 使用 Statement 接口

使用 Statement 接口时必须考虑以下几点：

- 你可以按需多次使用一个 Statement 实例。你可以在打开连接后立即创建一个，并在该连接的整个生命周期中使用它。你必须记住，每个 Statement 只能存在一个 ResultSet。
- 如果你需要在处理一个 ResultSet 期间执行另一个查询，只需创建并使用另一个 Statement 即可。
- 如果你使用线程，而且有多个线程在使用数据库，就必须为每个线程使用一个单独的 Statement。如果你打算使用它们，请参阅本文档后面介绍线程和 Servlet 的章节，其中涵盖了一些重要的内容。

58.7.2. 使用 ResultSet 接口

使用 ResultSet 接口时必须考虑以下几点：

- 在读取任何值之前，你必须调用 `next()`。如果有结果它会返回 true，但更重要的是，它会让该行做好被处理的准备。
- 按照 JDBC 规范，你应该只访问一个字段一次。遵守这条规则是最安全的，尽管目前 Postgres 驱动允许你按需多次访问一个字段。
- 一旦用完一个 ResultSet，你必须通过调用 `close()` 来关闭它。
- 一旦你用创建某个 ResultSet 的 Statement 请求另一个查询，当前打开的实例就会被关闭。

下面是一个例子：

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from mytable");
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

58.8. 执行更新

要执行一个更新（或任何其他不返回结果的 SQL 语句），只需使用 `executeUpdate()` 方法：

```
st.executeUpdate("create table basic (a int2, b int2)");
```

58.9. 关闭连接

要关闭数据库连接，只需对 Connection 调用 close() 方法：

```
db.close();
```

58.10. 使用大对象

在 Postgres 中，大对象（也称为blob）用于存放数据库中无法存储在普通 SQL 表中的数据。它们以一个“表/索引”对的形式存储，并通过一个 OID 值从你自己的表中引用。

重要

对于 Postgres，你必须在一个 SQL 事务内访问大对象。虽然原则上一直如此，但直到 v6.5 发布之前都没有被严格执行。你可以通过以 `false` 作为输入参数调用 `setAutoCommit()` 方法来打开一个事务：

```
Connection mycon;
...
mycon.setAutoCommit(false);
... now use Large Objects
```

使用大对象有两种方法。第一种是标准的 JDBC 方式，这里介绍的就是它。另一种使用我们自己对 api 的扩展，它把 libpq 的大对象 API 呈现给 Java，提供了比标准方式更好的大对象访问能力。在内部，驱动正是使用该扩展来提供大对象支持的。

在 JDBC 中，访问它们的标准方式是使用 ResultSet 中的 getBinaryStream() 方法和 PreparedStatement 中的 setBinaryStream() 方法。这些方法让大对象表现为一个 Java 流，使你可以使用 java.io 包和其他包来操纵该对象。

例如，假设你有一个表存储图像的文件名，并且你有一个包含该图像本身的大对象：

```
create table images (imgname name,imgoid oid);
```

要插入一幅图像，你可以使用：

```
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("insert into images
 values (?,?)");
ps.setString(1,file.getName());
ps.setBinaryStream(2,fis,file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

在这个例子中，setBinaryStream 从一个流向大对象传输指定数量的字节，并把 OID 存入持有对它的引用的字段中。

取回一幅图像甚至更容易（我这里使用 PreparedStatement，但 Statement 同样可以使用）：

```
PreparedStatement ps = con.prepareStatement("select oid from images
 where name=?");
ps.setString(1,"myimage.gif");
ResultSet rs = ps.executeQuery();
if(rs!=null) {
    while(rs.next()) {
        InputStream is = rs.getBinaryInputStream(1);
        // use the stream in some way here
        is.close();
    }
```

```
    rs.close();
}
ps.close();
```

这里你可以看到大对象是作为一个 InputStream 取回的。你还会注意到，我们在处理结果中的下一行之前关闭了流。这是 JDBC 规范的一部分，该规范规定，当调用 ResultSet.next() 或 ResultSet.close() 时，返回的任何 InputStream 都会被关闭。

58.11. Postgres 对 JDBC API 的扩展

Postgres 是一个可扩展的数据库系统。你可以向后端添加自己的函数，这些函数随后可以在查询中被调用，你甚至可以添加自己的数据类型。

现在，由于这些是我们独有的设施，我们通过一组扩展 API 从 Java 支持它们。标准驱动核心中的某些特性（如大对象）实际上就是使用这些扩展实现的。

访问扩展

要访问其中一些扩展，你需要使用 `postgresql.Connection` 类中的一些额外方法。这种情况下，你需要把 `Driver.getConnection()` 的返回值强制转换。

例如：

```
    Connection db = Driver.getConnection(url,user,pass);

    // later on
    Fastpath fp = ((postgresql.Connection)db).getFastpathAPI();
```

```
Class postgresql.Connection

java.lang.Object
   |
   +----postgresql.Connection

   public class Connection extends Object implements Connection
```

这些是用来访问我们的扩展的额外方法。这里没有列出 `java.sql.Connection` 已定义的方法。

```
public Fastpath getFastpathAPI() throws SQLException
```

返回当前连接的 `Fastpath API`。

注意：这不是 `JDBC` 的一部分，而是允许访问 `postgresql` 后端本身的函数。

它主要供 `LargeObject API` 使用

最好的使用方式如下：

```
import postgresql.fastpath.*;
...
Fastpath fp = ((postgresql.Connection)myconn).getFastpathAPI();
```

其中 myconn 是一个到 postgresql 的已打开连接。

返回值:
允许访问 postgresql 后端函数的 Fastpath 对象。

抛出: SQLException
首次初始化时由 Fastpath 抛出

```
public LargeObjectManager getLargeObjectAPI() throws SQLException
```

返回当前连接的 LargeObject API。

注意: 这不是 JDBC 的一部分，而是允许访问 postgresql 后端本身的函数。

最好的使用方式如下:

```
import postgresql.largeobject.*;
...
LargeObjectManager lo =
((postgresql.Connection)myconn).getLargeObjectAPI();
```

其中 myconn 是一个到 postgresql 的已打开连接。

返回值:
实现该 API 的 LargeObject 对象

抛出: SQLException
首次初始化时由 LargeObject 抛出

```
public void addDataType(String type,
                        String name)
```

它允许客户端代码为 postgresql 较独特的数据类型之一添加处理器。通常，驱动不认识的数据类型会由 ResultSet.getObject() 以 PGobject 实例的形式返回。

此方法允许你编写一个继承 PGobject 的类，并告诉驱动要使用的类型名和类名。

它的缺点是，每次建立连接时都必须调用此方法。

注意: 这不是 JDBC 的一部分，而是一个扩展。

最好的使用方式如下:

```
...
((postgresql.Connection)myconn).addDataType("mytype","my.class.name"-
);
...
```

其中 myconn 是一个到 postgresql 的已打开连接。

处理类必须继承 postgresql.util.PGobject

参见:
PGobject

Fastpath

Fastpath 是 libpq C 接口中的一个 API，允许客户端机器在数据库后端上执行一个函数。大多数客户端代码不需要使用它，但之所以提供，是因为大对象 API 会用到它。

要使用它，需要用下面这行导入 postgresql.fastpath 包：

```
import postgresql.fastpath.*;
```

然后，在你的代码中需要获得一个 FastPath 对象：

```
Fastpath fp = ((postgresql.Connection)conn).getFastpathAPI();
```

它会返回一个与数据库连接关联的实例，你可以用它发出命令。这里必须把 Connection 转换为 postgresql.Connection，因为 getFastpathAPI() 是我们自己的方法，不属于 JDBC。

一旦有了 Fastpath 实例，就可以使用 fastpath() 方法执行一个后端函数。

Class postgresql.fastpath.Fastpath

```
java.lang.Object
   |
   +----postgresql.fastpath.Fastpath
```

public class Fastpath

extends Object

这个类实现 Fastpath api。

这是一种从 Java 应用内执行嵌入在 postgresql 后端中的函数的手段。

它基于 src/interfaces/libpq/fe-exec.c 文件

参见：

FastpathFastpathArg, LargeObject

方法

```
public Object fastpath(int fnid,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

向 PostgreSQL 后端发送一次函数调用

参数：

fnid - 函数 id
resulttype - 如果结果是整数则为 true，其他结果为 false
args - 传递给 fastpath 的 FastpathArg 参数

返回值：

无数据时为 null，整数结果时为 Integer，否则为 byte[]

抛出：SQLException

发生数据库访问错误时。

```
public Object fastpath(String name,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

按名称向 PostgreSQL 后端发送一次函数调用。

注意:
过程名到函数 id 的映射必须已经存在，通常来自早先对 addfunction() 的调用。这是推荐的调用方式，因为函数 id 在后端的不同版本之间可能改变。关于其工作方式的示例，参见 postgresql.LargeObject

参数:
name - 函数名
resulttype - 如果结果是整数则为 true，其他结果为 false
args - 传递给 fastpath 的 FastpathArg 参数

返回值:
无数据时为 null，整数结果时为 Integer，否则为 byte[]

抛出: SQLException
名称未知或发生数据库访问错误时。

参见:
LargeObject

```
public int getInteger(String name,
                      FastpathArg args[]) throws SQLException
```

这个便捷方法假定返回值是 Integer

参数:
name - 函数名
args - 函数参数

返回值:
整数结果

抛出: SQLException
发生数据库访问错误或没有结果时。

```
public byte[] getData(String name,
                      FastpathArg args[]) throws SQLException
```

这个便捷方法假定返回值是二进制数据

参数:
name - 函数名
args - 函数参数

返回值:
包含结果的 byte[] 数组

抛出：SQLException
发生数据库访问错误或没有结果时。

```
public void addFunction(String name,
                        int fnid)
```

它向我们的查找表中添加一个函数。

用户代码应使用基于查询的 addFunctions 方法，而不是硬编码 oid。函数的 oid 并不保证保持不变，即使在同一版本的不同服务器上也是如此。

参数：
name - 函数名
fnid - 函数 id

```
public void addFunctions(ResultSet rs) throws SQLException
```

它接受一个包含两列的 ResultSet。第 1 列是函数名，第 2 列是 oid。

它会读取整个 ResultSet，把值装入函数表。

记住：调用之后要 close() 该 resultset！！

关于函数名查找的实现说明：

PostgreSQL 把函数 id 及其对应名称存储在 pg_proc 表中。为了在本地加速，不是在需要时逐个查询该表，而是使用一个 Hashtable。同时，只有需要的函数才会被录入该表，以使连接时间尽可能快。

postgresql.LargeObject 类在启动时执行一次查询，并把返回的 Result Set 传给这里的 addFunctions() 方法。

完成之后，LargeObject api 就按名称引用这些函数。

不要以为手工把它们转换成 oid 就能行。目前是可以，但在开发过程中它们可能改变（V7.0 期间曾有相关讨论），因此这样实现是为了避免将来不必要的麻烦。

参数：
rs - ResultSet

抛出：SQLException
发生数据库访问错误时。

参见：
LargeObjectManager

```
public int getID(String name) throws SQLException
```

返回与名称关联的函数 id

如果没有为该名称调用过 addFunction() 或 addFunctions()，则会抛出 SQLException。

参数：

name - 要查找的函数名

返回值:
供 fastpath 调用的函数 ID

抛出: SQLException
函数未知时。

Class postgresql.fastpath.FastpathArg

```
java.lang.Object
   |
   +----postgresql.fastpath.FastpathArg
```

```
public class FastpathArg extends Object
```

每次 fastpath 调用都需要一个参数数组，其数量和类型取决于所调用的函数。

这个类实现提供此能力所需的方法。

关于使用示例，参见 postgresql.largeobject 包

参见:
Fastpath, LargeObjectManager, LargeObject

构造器

```
public FastpathArg(int value)
```

构造一个由整数值构成的参数

参数:
value - 要设置的 int 值

```
public FastpathArg(byte bytes[])
```

构造一个由字节数组构成的参数

参数:
bytes - 要存储的数组

```
public FastpathArg(byte buf[],
                   int off,
                   int len)
```

构造一个由字节数组一部分构成的参数

参数:
buf - 源数组
off - 数组内的偏移
len - 要包含的数据长度

```
public FastpathArg(String s)
```

构造一个由字符串构成的参数。

参数:
s - 要存储的字符串

几何数据类型

PostgreSQL 有一组能把几何特性存入表中的数据类型，涵盖单点、直线和多边形。

我们用 postgresql.geometric 包在 Java 中支持这些类型。

它包含继承 postgresql.util.PGobject 类的各个类。关于如何实现你自己的数据类型处理器，参见那个类。

```
Class postgresql.geometric.PGbox

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGbox

   public class PGbox extends PGobject implements Serializable,
Cloneable
```

它表示 postgresql 中的 box 数据类型。

变量

```
 public PGpoint point[]
```

这是该矩形的两个角点。

构造器

```
 public PGbox(double x1,
              double y1,
              double x2,
              double y2)
```

参数:
x1 - 第一个 x 坐标
y1 - 第一个 y 坐标
x2 - 第二个 x 坐标
y2 - 第二个 y 坐标

```
 public PGbox(PGpoint p1,
              PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
 public PGbox(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的矩形定义

抛出: SQLException
如果定义无效

```
public PGbox()
```

必需的构造器

方法

```
public void setValue(String value) throws SQLException
```

此方法设置该对象的值。子类应当覆盖它，但仍应调用它。

参数:
value - 对象值的字符串表示
抛出: SQLException
如果值对该类型无效则抛出

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 postgresql 所期望语法表示的 PGbox

Overrides:
getValue in class PGobject

Class postgresql.geometric.PGcircle

java.lang.Object

```
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable
```

它表示 postgresql 的 circle 数据类型，由一个点和一个半径组成

变量

```
 public PGpoint center
```

这是圆心点

```
public double radius
```

这是半径

构造器

```
 public PGcircle(double x,
                 double y,
                 double r)
```

参数:
x - 圆心的坐标
y - 圆心的坐标
r - 圆的半径

```
 public PGcircle(PGpoint c,
                 double r)
```

参数:
c - 描述圆心的 PGpoint
r - 圆的半径

```
 public PGcircle(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

```
 public PGcircle()
```

此构造器由驱动使用。

方法

```
 public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 postgresql 所期望语法表示的 PGcircle

Overrides:
getValue in class PGobject

```
Class postgresql.geometric.PGline

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable
```

它实现由两点构成的 line。后端目前尚未实现 line 类型，但这个类保证了在实现之后我们就能直接使用它。

变量

```
public PGpoint point[]
```

这就是这两个点。

构造器

```
public PGline(double x1,
              double y1,
              double x2,
              double y2)
```

参数:
x1 - 第一个点的坐标
y1 - 第一个点的坐标
x2 - 第二个点的坐标
y2 - 第二个点的坐标

```
public PGline(PGpoint p1,
              PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
public PGline(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

```
public PGline()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的线段定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

```
        Overrides:
                equals in class PGobject

 public Object clone()
```

必须覆盖此方法才能克隆该对象

```
        Overrides:
                clone in class PGobject

 public String getValue()
```

返回值:

按 postgresql 所期望语法表示的 PGline

```
        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGlseg

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGlseg

   public class PGlseg extends PGobject implements Serializable,
Cloneable
```

它实现由两点构成的 lseg（线段）

变量

```
 public PGpoint point[]
```

这就是这两个点。

构造器

```
 public PGlseg(double x1,
               double y1,
               double x2,
               double y2)
```

参数:

x1 - 第一个点的坐标
y1 - 第一个点的坐标
x2 - 第二个点的坐标
y2 - 第二个点的坐标

```
 public PGlseg(PGpoint p1,
               PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
public PGlseg(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

```
public PGlseg()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的线段定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 postgresql 所期望语法表示的 PGlseg

Overrides:
getValue in class PGobject

```
Class postgresql.geometric.PGpath

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpath

   public class PGpath extends PGobject implements Serializable,
Cloneable
```

它实现 path（一条多段线，可以是闭合的）

变量

```
 public boolean open
```

如果路径是开放的则为 true，闭合则为 false

```
 public PGpoint points[]
```

定义该路径的各个点

构造器

```
 public PGpath(PGpoint points[],
               boolean open)
```

参数:
points - 定义该路径的 PGpoint 数组
open - 如果路径是开放的则为 true，闭合则为 false

```
 public PGpath()
```

驱动所需

```
 public PGpath(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

方法

```
 public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的路径定义

抛出: SQLException
转换失败时

```
        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        参数:
                obj - 要比较的对象

        返回值:
                如果两个矩形相同则为 true

        Overrides:
                equals in class PGobject

 public Object clone()

          必须覆盖此方法才能克隆该对象

        Overrides:
                clone in class PGobject

 public String getValue()

          返回按 postgresql 所期望语法表示的该多边形

        Overrides:
                getValue in class PGobject

 public boolean isOpen()

     如果路径是开放的则返回 true

 public boolean isClosed()

     如果路径是闭合的则返回 true

 public void closePath()

     把路径标记为闭合

 public void openPath()

     把路径标记为开放

Class postgresql.geometric.PGpoint

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
```

Cloneable

它实现 java.awt.Point 的一个版本，只是用 double 表示坐标。

它映射到 postgresql 的 point 数据类型。

变量

public double x

该点的 X 坐标

public double y

该点的 Y 坐标

构造器

```
public PGpoint(double x,
               double y)
```

参数:
x - 坐标
y - 坐标

public PGpoint(String value) throws SQLException

当某点嵌入其他几何类型的定义中时，主要由那些类型调用此构造器。

参数:
value - PostgreSQL 语法的该点定义

public PGpoint()

驱动所需

方法

public void setValue(String s) throws SQLException

参数:
s - PostgreSQL 语法的该点定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

public boolean equals(Object obj)

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

public Object clone()

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

public String getValue()

返回值:
按 postgresql 所期望语法表示的 PGpoint

Overrides:
getValue in class PGobject

public void translate(int x,
int y)

按给定的量平移该点。

参数:
x - 在 x 轴上要加的整数增量
y - 在 y 轴上要加的整数增量

public void translate(double x,
double y)

按给定的量平移该点。

参数:
x - 在 x 轴上要加的 double 增量
y - 在 y 轴上要加的 double 增量

public void move(int x,
int y)

把该点移动到给定的坐标。

参数:
x - 整数坐标
y - 整数坐标

public void move(double x,
double y)

把该点移动到给定的坐标。

参数:

```
                x - double 坐标
                y - double 坐标

 public void setLocation(int x,
                         int y)
```

把该点移动到给定的坐标。相关描述参见 java.awt.Point

参数:

```
                x - 整数坐标
                y - 整数坐标
```

参见:

```
                Point

 public void setLocation(Point p)
```

把该点移动到给定的 java.awt.Point。相关描述参见 java.awt.Point

参数:

```
                p - 要移动到的点
```

参见:

```
                Point

Class postgresql.geometric.PGpolygon

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpolygon

   public class PGpolygon extends PGobject implements Serializable,
Cloneable
```

它实现 PostgreSQL 中的 polygon 数据类型。

变量

```
 public PGpoint points[]
```

定义该多边形的各个点

构造器

```
 public PGpolygon(PGpoint points[])
```

用一个 PGpoint 数组创建多边形

参数:

```
                points - 定义该多边形的各个点

 public PGpolygon(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

```
public PGpolygon()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的多边形定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 postgresql 所期望语法表示的 PGpolygon

Overrides:
getValue in class PGobject

大对象

标准 JDBC 规范支持大对象。但那个接口能力有限，而 PostgreSQL 提供的 api 允许像访问本地文件一样随机访问对象内容。

postgresql.largeobject 包向 Java 提供了 libpq C 接口的大对象 API。它由两个类组成：负责创建、打开和删除大对象的 LargeObjectManager，以及处理单个对象的 LargeObject。

Class postgresql.largeobject.LargeObject

```
java.lang.Object
   |
   +----postgresql.largeobject.LargeObject
```

```
public class LargeObject extends Object
```

这个类实现 postgresql 的大对象接口。

它提供运行该接口所需的基本方法，另有一对方法为该对象提供 InputStream 和 OutputStream 类。

通常，客户端代码会使用 ResultSet 中的 getAsciiStream、getBinaryStream 或 getUnicodeStream 方法，或 PreparedStatement 中的 setAsciiStream、setBinaryStream 或 setUnicodeStream 方法来访问大对象。

然而有时需要对大对象的更底层访问，这是 JDBC 规范不支持的。

关于如何访问大对象或如何创建大对象，参见 postgresql.largeobject.LargeObjectManager。

参见：

LargeObjectManager

变量

```
public static final int SEEK_SET
```

表示从文件开头定位

```
public static final int SEEK_CUR
```

表示从当前位置定位

```
public static final int SEEK_END
```

表示从文件末尾定位

方法

```
public int getOID()
```

返回值：

该大对象的 OID

```
public void close() throws SQLException
```

此方法关闭该对象。调用之后不得再对该对象调用方法。

抛出: SQLException
发生数据库访问错误时。

```
public byte[] read(int len) throws SQLException
```

从该对象读取一些数据，并以 byte[] 数组返回

参数:
len - 要读取的字节数

返回值:
包含所读数据的 byte[] 数组

抛出: SQLException
发生数据库访问错误时。

```
public void read(byte buf[],
                 int off,
                 int len) throws SQLException
```

从该对象读取一些数据到既有数组中

参数:
buf - 目标数组
off - 数组内的偏移
len - 要读取的字节数

抛出: SQLException
发生数据库访问错误时。

```
public void write(byte buf[]) throws SQLException
```

把一个数组写入该对象

参数:
buf - 要写入的数组

抛出: SQLException
发生数据库访问错误时。

```
public void write(byte buf[],
                  int off,
                  int len) throws SQLException
```

把数组中的一些数据写入该对象

参数:
buf - 目标数组
off - 数组内的偏移
len - 要写入的字节数

抛出: SQLException

发生数据库访问错误时。

```
public void seek(int pos,
                 int ref) throws SQLException
```

设置该对象内的当前位置。

这类似于标准 C 库中的 fseek() 调用，允许随机访问大对象。

参数:
pos - 对象内的位置
ref - SEEK_SET、SEEK_CUR 或 SEEK_END 之一
抛出: SQLException
发生数据库访问错误时。

```
public void seek(int pos) throws SQLException
```

设置该对象内的当前位置。

这类似于标准 C 库中的 fseek() 调用，允许随机访问大对象。

参数:
pos - 对象内相对开头的位置

抛出: SQLException
发生数据库访问错误时。

```
public int tell() throws SQLException
```

返回值:
对象内的当前位置

抛出: SQLException
发生数据库访问错误时。

```
public int size() throws SQLException
```

此方法效率不高，因为要知道对象的大小，只能定位到末尾、记录当前位置、再回到原位置。

将来会找到更好的方法。

返回值:
大对象的大小

抛出: SQLException
发生数据库访问错误时。

```
public InputStream getInputStream() throws SQLException
```

返回该对象的一个 InputStream。

随后这个 InputStream 可用于任何需要 InputStream 的方法。

抛出: SQLException
发生数据库访问错误时。

```
public OutputStream getOutputStream() throws SQLException
```

返回该对象的一个 OutputStream

随后这个 OutputStream 可用于任何需要 OutputStream 的方法。

抛出: SQLException
发生数据库访问错误时。

```
Class postgresql.largeobject.LargeObjectManager

java.lang.Object
   |
   +----postgresql.largeobject.LargeObjectManager

public class LargeObjectManager extends Object
```

这个类实现 postgresql 的大对象接口。

它提供允许客户端代码在数据库中创建、打开和删除大对象的方法。打开对象时会返回一个 postgresql.largeobject.LargeObject 实例，其方法随后允许访问该对象。

这个类只能由 postgresql.Connection 创建

要访问这个类，使用下面这段代码:

```
import postgresql.largeobject.*;
Connection  conn;
LargeObjectManager lobj;
... 打开连接的代码 ...
lobj = ((postgresql.Connection)myconn).getLargeObjectAPI();
```

通常，客户端代码会使用 ResultSet 中的 getAsciiStream、getBinaryStream 或 getUnicodeStream 方法，或 PreparedStatement 中的 setAsciiStream、setBinaryStream 或 setUnicodeStream 方法来访问大对象。

然而有时需要对大对象的更底层访问，这是 JDBC 规范不支持的。

关于如何操作大对象内容，参见 postgresql.largeobject.LargeObject。

参见:
LargeObject

变量

```
public static final int WRITE
```

此模式表示我们想写一个对象

```
public static final int READ
```

此模式表示我们想读一个对象

```
public static final int READWRITE
```

此模式是默认值，表示我们想对一个大对象进行读写访问

方法

```
public LargeObject open(int oid) throws SQLException
```

它根据 OID 打开一个已有大对象。此方法假定需要 READ 和 WRITE 访问（默认）。

参数:
oid - 大对象的

返回值:
提供对该对象访问的 LargeObject 实例

抛出: SQLException
出错时

```
public LargeObject open(int oid,
                        int mode) throws SQLException
```

它根据 OID 打开一个已有大对象

参数:
oid - 大对象的
mode - 打开模式

返回值:
提供对该对象访问的 LargeObject 实例

抛出: SQLException
出错时

```
public int create() throws SQLException
```

它创建一个大对象并返回其 OID。

新对象的属性默认为 READWRITE。

返回值:
新对象的 oid

抛出: SQLException
出错时

```
public int create(int mode) throws SQLException
```

它创建一个大对象并返回其 OID

参数:

mode - 描述新对象不同属性的位掩码

返回值:
新对象的 oid

抛出: SQLException
出错时

```
public void delete(int oid) throws SQLException
```

它删除一个大对象。

参数:
oid - 描述要删除的对象

抛出: SQLException
出错时

```
public void unlink(int oid) throws SQLException
```

它删除一个大对象。

它与 delete 方法相同，之所以提供是因为 C API 使用 unlink。

参数:
oid - 描述要删除的对象

抛出: SQLException
出错时

对象序列化
PostgreSQL 不是普通的 SQL 数据库。它比大多数其他数据库可扩展得多，并且支持它独有的面向对象特性。

其中一个后果是，可以让一个表引用另一个表中的某一行。例如:

```
test=> create table users (username name,fullname text);
CREATE
test=> create table server (servername name,adminuser users);
CREATE
test=> insert into users values ('peter','Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast',2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
--------+--------------
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
----------+---------
maidast   |  2610132
```

```
(1 row)
```

好，上面的例子表明我们可以把表名用作字段，该行的 oid 值就存储在这个字段中。

这与 Java 有什么关系？

在 Java 中，只要对象的类实现了 java.io.Serializable 接口，就可以把对象存储到流中。这一过程称为对象序列化，可用于把复杂对象存入数据库。

而在 JDBC 之下，你得用 LargeObject 来存储它们。然而，无法对这些对象执行查询。

postgresql.util.Serialize 类所做的事，就是提供一种把对象存成表、再从表中取回该对象的手段。大多数情况下你不需要直接访问这个类，而是使用 PreparedStatement.setObject() 和 ResultSet.getObject() 方法。这些方法会把对象的类名与数据库中的表比对。找到匹配时，就假定该对象是一个序列化对象，并从那个表取回它。在此过程中，如果对象还包含其他序列化对象，它会沿树递归。

听起来复杂？其实比我写的简单──只是难以解释。

唯一需要直接访问这个类的场合，是使用 create() 方法。驱动本身不使用它们，而是根据你要序列化的 Java 对象或类，向数据库发出一条或多条 "create table" 语句。

哦，最后一件事。如果你的对象里有这样一行：

```
  public int oid;
```

那么，当对象从表中取回时，它会被设为表内的 oid。之后，如果对象被修改并重新序列化，既有条目会被更新。

如果没有 oid 变量，那么对象被序列化时总是插入表中，表中任何既有条目都会保留。

序列化之前把 oid 设为 0 也会导致对象被插入。这使得可以在数据库中复制一个对象。

```
Class postgresql.util.Serialize

java.lang.Object
   |
   +----postgresql.util.Serialize

   public class Serialize extends Object
```

这个类利用 PostgreSQL 的面向对象特性来存储 Java 对象。做法是把 Java 类名映射为数据库中的一个表。新表中的每个条目代表该类的一个序列化实例。由于每个条目都有一个 OID（对象标识符），这个 OID 可以被包含在另一个表中。这太复杂，不在此展示，主文档中会有更详细的说明。

构造器

```
 public Serialize(Connection c,
                  String type) throws SQLException
```

它创建一个实例，可用于对 PostgreSQL 表中的 Java 对象做序列化/反序列化。

方法

```
public Object fetch(int oid) throws SQLException
```

它根据 OID 从表中取回一个对象

参数:
oid - 对象的 oid

返回值:
与 oid 对应的对象

抛出: SQLException
出错时

```
public int store(Object o) throws SQLException
```

它把一个对象存入表中并返回其 OID。

如果对象有一个名为 OID 的 int 且其值 > 0，就用该值作 OID，表将被更新。如果 OID 的值为 0，则会新建一行，并且 OID 的值会被设置到对象中。这使对象在数据库中的值可以更新。如果对象没有名为 OID 的 int，则直接存储对象。但若对象随后被取回、修改并再次存储，它的新状态会被追加到表中，而不会覆盖旧条目。

参数:
o - 要存储的对象（必须实现 Serializable）

返回值:
已存储对象的 oid

抛出: SQLException
出错时

```
public static void create(Connection con,
                          Object o) throws SQLException
```

驱动不使用此方法；它根据一个可序列化的 Java 对象创建表。应当在序列化任何对象之前使用它。

参数:
c - 到数据库的连接
o - 表所基于的对象

抛出: SQLException
出错时

返回值:
与 oid 对应的对象

抛出: SQLException
出错时

```
public int store(Object o) throws SQLException
```

它把一个对象存入表中并返回其 OID。

如果对象有一个名为 OID 的 int 且其值 > 0，就用该值作 OID，表将被更新。如果 OID 的值为 0，则会新建一行，并且 OID 的值会被设置到对象中。这使对象在数据库中的值可以更新。如果对象没有名为 OID 的 int，则直接存储对象。但若对象随后被取回、修改并再次存储，它的新状态会被追加到表中，而不会覆盖旧条目。

参数:
o - 要存储的对象（必须实现 Serializable）

返回值:
已存储对象的 oid

抛出: SQLException
出错时

```
public static void create(Connection con,
                          Object o) throws SQLException
```

驱动不使用此方法；它根据一个可序列化的 Java 对象创建表。应当在序列化任何对象之前使用它。

参数:
c - 到数据库的连接
o - 表所基于的对象

抛出: SQLException
出错时

```
public static void create(Connection con,
                          Class c) throws SQLException
```

驱动不使用此方法；它根据一个可序列化的 Java 对象创建表。应当在序列化任何对象之前使用它。

参数:
c - 到数据库的连接
o - 表所基于的类

抛出: SQLException
出错时

```
public static String toPostgreSQL(String name) throws SQLException
```

它把 . 替换为 _，从而把 Java 类名转换为 postgresql 表名

因此，类名中不能含有 _。

另一个限制是，完整类名（含包名）不能超过 31 个字符（这是 PostgreSQL 强加的限制）。

参数:
name - 类名

返回值：
PostgreSQL 表名

抛出：SQLException
出错时

```
public static String toClassName(String name) throws SQLException
```

它把 _ 替换为 .，从而把 postgresql 表名转换为 Java 类名

参数：
name - PostgreSQL 表名

返回值：
类名

抛出：SQLException
出错时

工具类

postgresql.util 包包含主驱动内部使用的类以及其他扩展使用的类。

Class postgresql.util.PGmoney

```
java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.util.PGmoney
```

```
   public class PGmoney extends PGobject implements Serializable,
Cloneable
```

它实现一个处理 PostgreSQL money 类型的类

变量

```
public double val
```

字段的值

构造器

```
public PGmoney(double value)
```

参数：
value - 字段的值

```
public PGmoney(String value) throws SQLException
```

当某点嵌入其他几何类型的定义中时，主要由那些类型调用此构造器。

参数：

value - PostgreSQL 语法的该点定义

public PGmoney()

驱动所需

方法

public void setValue(String s) throws SQLException

参数:
s - PostgreSQL 语法的该点定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

public boolean equals(Object obj)

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:
equals in class PGobject

public Object clone()

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

public String getValue()

返回值:
按 postgresql 所期望语法表示的 PGpoint

Overrides:
getValue in class PGobject

Class postgresql.util.PGobject

java.lang.Object
|
+----postgresql.util.PGobject

public class PGobject extends Object implements Serializable,
Cloneable

这个类用于描述 JDBC 标准所不知道的数据类型。调用 postgresql.Connection 可以让一个继承本类的类与一个命名类型关联。postgresql.geometric 包就是这样工作的。对任何没有自己处理器的类型，ResultSet.getObject() 都会返回这个类。因此，任何 postgresql 数据类型都得到支持。

构造器

```
public PGobject()
```

它由 postgresql.Connection.getObject() 调用来创建对象。

方法

```
public final void setType(String type)
```

此方法设置该对象的类型。

子类不应扩展它，因此它是 final 的

参数:
type - 描述该对象类型的字符串

```
public void setValue(String value) throws SQLException
```

此方法设置该对象的值。它必须被覆盖。

参数:
value - 对象值的字符串表示

抛出: SQLException
如果值对该类型无效则抛出

```
public final String getType()
```

由于它在对象生命周期内不能改变，因此是 final 的。

返回值:
该对象的类型名

```
public String getValue()
```

必须覆盖它，以 postgresql 所要求的形式返回对象的值。

返回值:
该对象的值

```
public boolean equals(Object obj)
```

必须覆盖它才能比较对象

参数:
obj - 要比较的对象

返回值:

如果两个矩形相同则为 true

Overrides:
equals in class Object

public Object clone()

必须覆盖此方法才能克隆该对象

Overrides:
clone in class Object

public String toString()

这里已定义它，因此用户代码不必覆盖。

返回值:
按 postgresql 所期望语法表示的该对象的值

Overrides:
toString in class Object

Class postgresql.util.PGtokenizer

java.lang.Object
|
+----postgresql.util.PGtokenizer

public class PGtokenizer extends Object

这个类用于对 postgres 的文本输出做分词。

本可以用 StringTokenizer 来做，但我们需要处理 '(' ')' '[' ']' '<' 和 '>' 的嵌套，几何数据类型会用到它们。

它主要由几何类使用，但对解析 postgresql 自定义数据类型的任何输出也有用。

参见:
PGbox, PGcircle, PGlseg, PGpath, PGpoint, PGpolygon

构造器

public PGtokenizer(String string,
char delim)

创建一个分词器。

参数:
string - 包含各词的字符串
delim - 分割各词的单个字符

方法

public int tokenize(String string,

```
                    char delim)
```

用一个新字符串和/或新分隔符重置该分词器。

参数:

```
string - 包含各词的字符串
delim - 分割各词的单个字符
```

```
public int getSize()
```

返回值:

可用词的数量

```
public String getToken(int n)
```

参数:

```
n - 词的编号 (0 ... getSize()-1)
```

返回值:

词的值

```
public PGtokenizer tokenizeToken(int n,
                                 char delim)
```

它基于我们的某个词返回一个新的分词器。几何数据类型用它处理嵌套的词（通常是 PGpoint）。

参数:

```
n - 词的编号 (0 ... getSize()-1)
delim - 要使用的分隔符
```

返回值:

基于该词的一个新 PGtokenizer 实例

```
public static String remove(String s,
                            String l,
                            String t)
```

移除一个字符串的前导/尾随字符串

参数:

```
s - 源字符串
l - 要移除的前导字符串
t - 要移除的尾随字符串
```

返回值:

不含前导/尾随字符串的字符串

```
public void remove(String l,
                   String t)
```

移除所有词的前导/尾随字符串

参数:

l - 要移除的前导字符串
t - 要移除的尾随字符串

public static String removePara(String s)

移除字符串开头和结尾的 (和)

参数:
s - 要从中移除的字符串

返回值:
不含 (或) 的字符串

public void removePara()

移除所有词开头和结尾的 (和)

返回值:
不含 (或) 的字符串

public static String removeBox(String s)

移除字符串开头和结尾的 [和]

参数:
s - 要从中移除的字符串

返回值:
不含 [或] 的字符串

public void removeBox()

移除所有词开头和结尾的 [和]

返回值:
不含 [或] 的字符串

public static String removeAngle(String s)

移除字符串开头和结尾的 < 和 >

参数:
s - 要从中移除的字符串

返回值:
不含 < 或 > 的字符串

public void removeAngle()

移除所有词开头和结尾的 < 和 >

返回值:
不含 < 或 > 的字符串

Class postgresql.util.Serialize

它已在前面"对象序列化"一节中说明。

Class postgresql.util.UnixCrypt

```
java.lang.Object
   |
   +----postgresql.util.UnixCrypt
```

public class UnixCrypt extends Object

这个类使我们能够在口令经网络流发送时对其加密

包含用于加密口令以及与 Unix 加密口令比对的静态方法。

原始源码见 John Dumas 的 Java Crypt 页面。

http://www.zeh.com/local/jfd/crypt.html

方法

```
public static final String crypt(String salt,
                                 String original)
```

根据明文口令和一个"盐"加密口令。

参数:
salt - 一个两字符字符串，表示用于以多种方式迭代加密引擎的盐。如果要生成新加密，此值应当随机化。 original - 要加密的口令。

返回值:
由 2 字符盐后接加密口令组成的字符串。

```
public static final String crypt(String original)
```

根据明文口令加密口令。此方法用 'java.util.Random' 类生成随机盐。

参数:
original - 要加密的口令。

返回值:
由 2 字符盐后接加密口令组成的字符串。

```
public static final boolean matches(String encryptedPassword,
                                    String enteredPassword)
```

检查 enteredPassword 加密后是否等于 encryptedPassword。

参数:
encryptedPassword - 已加密口令。假定前两个字符是盐。该字符串与 Unix /etc/passwd 文件中的形式相同。 enteredPassword - 用户（或其他途径）输入的口令。

返回值：

如果口令应视为正确则为 `true`。

在多线程或 `Servlet` 环境中使用驱动

许多 `JDBC` 驱动都有一个问题：同一时刻只能有一个线程使用一个连接——否则一个线程可能在另一个线程接收结果时发出查询，这对数据库引擎是坏事。

`PostgreSQL 6.4` 为整个驱动带来了线程安全。`6.3.x` 中标准 `JDBC` 是线程安全的，但 `Fastpath API` 不是。

因此，如果你的应用使用多线程（大多数像样的应用都会），就不必为确保同一时刻只有一个线程使用数据库而操心复杂方案。

如果一个线程在另一个线程正在使用连接时试图使用它，它会等那个线程完成当前操作。

若是标准 `SQL` 语句，操作就是发送语句并（完整地）取回任何 `ResultSet`。

若是 `Fastpath` 调用（即从 `LargeObject` 读取一块），就是发送并取回该块的时间。

对应用和小程序来说这没问题，但可能给 `servlet` 带来性能问题。

对 `servlet` 而言，连接可能承受重载。如果多个线程执行查询，每个都会停顿，这可能不是你想要的。

为解决此问题，建议创建一个连接池。

每当线程需要使用数据库时，它向一个管理器类请求 `Connection`。管理器把一个空闲连接交给线程并标记为忙。如果没有空闲连接，就新开一个。

线程用完后把连接还给管理器，管理器可以关闭它或把它放回池中。管理器还会检查连接是否仍存活，若已死则从池中移除。

所以在 `servlet` 场景下，用单连接还是连接池由你决定。池的好处是线程不会受单一网络连接造成的瓶颈影响；坏处是服务器负载增加，因为每个 `Connection` 都会创建一个后端。

这取决于你和你的应用需求。

58.12. 延伸阅读

如果你还没有读过，我建议你阅读 JDBC API 文档（随 Sun 的 JDK 提供）以及 JDBC 规范。两者都可以在 JavaSoft 的网站[2]上找到。

我自己的网站[3]包含本文档未收录的更新信息，并且还提供 v6.4 及更早版本的预编译驱动。

[2] http://www.javasoft.com
[3] http://www.retep.org.uk

第 59 章 Lisp 编程接口

pg.el是一个面向 emacs 的 Postgres套接字级接口。

作者

由 Eric Marsden[1] 于 1999 年 7 月 21 日撰写。

pg.el是一个面向 emacs（卓越的文本编辑器）的Postgres套接字级接口。该模块能够把一系列 SQL 类型类型强制转换为等价的 Emacs Lisp 类型。它目前既不支持 crypt 或 Kerberos 认证，也不支持大对象。

代码（版本 0.2）以 GNU GPL 许可发布，可从 Eric Marsden[2]获取

自上个版本以来的变更：

- 现在可以在 XEmacs 下工作（已在 Emacs 19.34 和 20.2 以及 XEmacs 20.4 上测试）
- 增加了提供数据库元信息的函数（数据库、表、列的列表）
- `pg:result' 的参数现在改为 :keywords
- 对 MULE 有抵抗力
- 更多自测试代码

请注意这是一个程序员的 API，不提供任何形式的用户界面。示例：

```
(defun demo ()
   (interactive)
   (let* ((conn (pg:connect "template1" "postgres" "postgres"))
          (res (pg:exec conn "SELECT * from scshdemo WHERE a = 42")))
     (message "status is %s"   (pg:result res :status))
     (message "metadata is %s" (pg:result res :attributes))
     (message "data is %s"     (pg:result res :tuples))
     (pg:disconnect conn)))
```

[1] mailto:emarsden@mail.dotcom.fr
[2] http://www.chez.com/emarsden/downloads/pg.el

部分 V. 开发者指南

开发者指南包括对设计决策的讨论以及对未来开发的建议。

目录

第 60 章 Postgres 源代码

60.1. 格式

源代码格式使用 4 列制表宽度，目前保留制表符（即，不把制表符展开为空格）。

对于 emacs，把下列内容（或类似内容）加入你的 ~/.emacs 初始化文件：

```
;; check for files with a path containing "postgres" or "pgsql"
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.[ch]\\'" .
 pgsql-c-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.cc\\'" .
 pgsql-c-mode) auto-mode-alist))

(defun pgsql-c-mode ()
  ;; sets up formatting for Postgres C code
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; set c-basic-offset to 4, plus
 other stuff
  (c-set-offset 'case-label '+)   ; tweak case indent to match PG
 custom
  (setq indent-tabs-mode t))      ; make sure we keep tabs when
 indenting
```

对于 vi，你的 ~/.vimrc 或等价文件应包含：

```
set tabstop=4
```

或在 vi 中等价地尝试

```
:set ts=4
```

文本浏览工具 more 和 less 可以这样调用

```
more -x4
less -x4
```

第 61 章 PostgreSQL 内部概述

作者

本章最初是 Simkovics, 1998（Stefan Simkovics 在维也纳技术大学完成的硕士论文，由 O.Univ.Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 指导）的一部分。

本章概述 Postgres 后端的内部结构。阅读以下各节之后，你应该能够大致了解一个查询是如何被处理的。不要期望在这里看到详细的描述（我认为，一份涵盖 Postgres 中使用的所有数据结构和函数的描述将超过 1000 页！）。本章旨在帮助读者理解后端从收到一个查询到发送结果为止，其内部总体上的控制流和数据流。

61.1. 查询的路径

这里将简要概述一个查询为得到结果必须经过的各个阶段。

1. 必须先建立从应用程序到 Postgres 服务器的连接。应用程序将查询发送给服务器，并接收服务器返回的结果。

2. 解析器阶段检查应用程序（客户端）传来的查询是否具有正确的语法，并创建一棵查询树。

3. 重写系统接收由解析器阶段创建的查询树，并查找任何可应用于该 querytree 的规则（存储在系统目录中），然后执行这些规则体中给出的转换。
   重写系统的一个应用在于视图的实现。

   每当对一个视图（即虚拟表）发出查询时，重写系统都会将用户的查询重写成一个改为访问视图定义中给出的基表的查询。

4. 规划器/优化器接收（重写后的）querytree，并创建一个将作为执行器输入的 queryplan。

   它首先生成所有能得到同一结果的可能路径。例如，如果待扫描的某个关系上有一个索引，那么该扫描就有两条路径：一种是简单的顺序扫描，另一种是使用该索引。
   接着会估算执行每个计划的代价，选出代价最低的计划并交回。

5. 执行器递归地遍历计划树，并以计划所表示的方式提取元组。执行器在扫描关系时会使用存储系统，执行排序和连接，计算限定条件，最后返回得到的元组。

在后续各节中，我们将更详细地介绍上述列出的每一项内容，以便更好地理解 Postgres 的内部控制和数据结构。

61.2. 连接是如何建立的

Postgres 使用一种简单的“每用户一个进程”客户端/服务器模型实现。在这种模型中，一个客户端进程恰好连接到一个服务器进程。由于我们per se（就其本身而言）并不知道会建立多少个连接，因此必须使用一个主进程，在每次收到连接请求时派生一个新的服务器进程。
这个主进程称为 `postmaster`，在指定的 TCP/IP 端口上监听传入连接。每当检测到连接请求时，`postmaster` 进程就会派生一个新的称为 `postgres` 的服务器进程。各个服务器任务（`postgres` 进程）之间使用信号量和共享内存通信，以确保并发数据访问期间的数据完整性。
图 \ref{connection} 展示了主进程 `postmaster`、服务器进程 `postgres` 和一个客户端应用程序之间的交互。

客户端进程既可以是 psql 前端（用于交互式 SQL 查询），也可以是任何使用 `libpg` 库实现的用户应用程序。注意，使用 ecpg（Postgres 面向 C 的嵌入式 SQL 预处理器）实现的应用程序同样使用这个库。

一旦连接建立起来，客户端进程就可以向后端（服务器）发送查询。查询以纯文本方式传输，也就是说，前端（客户端）不进行解析。服务器解析查询，创建一个执行计划，执行该计划，并通过已建立的连接把检索到的元组返回给客户端。

61.3. 解析器阶段

解析器阶段由两部分组成：

- 解析器定义在 `gram.y` 和 `scan.l` 中，它使用 Unix 工具 yacc 和 lex 构建。
- 转换过程，它会对解析器返回的数据结构做修改和补充。

61.3.1. 解析器

解析器必须检查查询串（以普通 ASCII 文本形式到达）的语法是否有效。如果语法正确，就构建一棵解析树并返回；否则返回错误。实现时使用了著名的 Unix 工具 lex 和 yacc。

词法分析器定义在文件 `scan.l` 中，负责识别标识符、SQL 关键字等。每找到一个关键字或标识符，就会生成一个词元并交给解析器。

解析器定义在文件 `gram.y` 中，由一组语法规则和动作组成，每当某条规则被触发时，对应动作就会执行。动作中的代码（实际上是 C 代码）用于构造解析树。

文件 `scan.l` 会被转换成 C 源文件 `scan.c`，所用程序是 lex；而 `gram.y` 则会被转换成 `gram.c`，所用程序是 yacc。这些转换完成之后，就可以使用普通的 C 编译器来构建解析器。绝不要修改这些生成出来的 C 文件，因为下一次调用 lex 或 yacc 时，它们都会被覆盖。

> **注意**
>
> 上述转换和编译通常都是借助 makefiles 自动完成的，这些文件随 Postgres 源代码发行版一同提供。

对 yacc 的详细描述，或者对 `gram.y` 中给出的语法规则的说明，都超出了本文的范围。有许多书籍和文档专门讨论 lex 和 yacc。在开始研究之前，你应该先熟悉 yacc，然后再去看 `gram.y` 中给出的语法，否则你将无法理解其中发生了什么。

为了更好地理解 Postgres 在处理查询时所使用的数据结构，我们用一个例子来说明这些数据结构在每个阶段所发生的变化。

例 61.1. 一个简单的 Select

这个例子包含下面这个简单的查询，它将被用于后续各节的各种描述和图示中。该查询假设 The Supplier Database（供应商数据库）中给出的各个表已经定义好。

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

图 \ref{parsetree} 展示了 `gram.y` 中的语法规则和动作为

这个例子包含下面这个简单的查询，它将被用于后续各节的各种描述和图示中。

该查询假设 The Supplier Database（供应商数据库）中给出的各个表已经定义好。

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

中给出的查询所构建的解析树（其中不含 where 子句的操作符树，后者见图 \ref{where_clause}，因为一张图放不下两个数据结构）。

树的顶层节点是一个 `SelectStmt` 节点。对于 SQL 查询 from 子句中出现的每一个表项，都会创建一个 `RangeVar` 节点，其中保存着别名的名称，以及一个指向 `RelExpr` 节点的指针，后者保存着关系的名称。所有 `RangeVar` 节点都被收集到一个列表中，挂接在 `SelectStmt` 节点的 `fromClause` 字段上。

对于 SQL 查询选择列表中出现的每一个表项，都会创建一个 `ResTarget` 节点，其中保存着一个指向 `Attr` 节点的指针。`Attr` 节点保存着该表项的关系名，以及一个指向 `Value` 节点的指针，后者保存着属性的名称。所有 `ResTarget` 节点都被收集到一个列表中，连接到 `SelectStmt` 节点的 `targetList` 字段上。

图 \ref{where_clause} 展示了为例

这个例子包含下面这个简单的查询，它将被用于后续各节的各种描述和图示中。

该查询假设 The Supplier Database（供应商数据库）中给出的各个表已经定义好。

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

中给出的 SQL 查询的 where 子句构建的操作符树，它挂接在 `SelectStmt` 节点的 `qual` 字段上。操作符树的顶层节点是一个表示 `AND` 运算的 `A_Expr` 节点。该节点有两个分别称为 `lexpr` 和 `rexpr` 的后继，分别指向两棵子树。挂接在 `lexpr` 上的子树表示限定条件 `s.sno > 2`，挂接在 `rexpr` 上的子树表示 `s.sno = se.sno`。对于每个属性，都会创建一个 `Attr` 节点，其中保存着关系的名称以及一个指向 `Value` 节点的指针，后者保存着属性的名称。对于查询中出现的常量项，则创建一个保存该值的 `Const` 节点。

61.3.2. 转换过程

转换过程以解析器返回的树作为输入，并递归地遍历它。如果找到一个 `SelectStmt` 节点，就把它转换为一个 `Query` 节点，后者将成为新数据结构的最顶层节点。图 \ref{transformed} 展示了转换后的数据结构（转换后的 where 子句部分见图 \ref{transformed_where}，因为一张图放不下所有部分）。

接下来检查 FROM 子句中的关系名是否为系统所知。对于每一个出现在系统目录中的关系名，都会创建一个 RTE 节点，其中包含关系名、别名和关系 id。从这时起，就用关系 id 来引用查询中给出的各个关系。所有 RTE 节点都被收集到范围表项列表中，该列表连接到 `Query` 节点的 `rtable` 字段上。如果在查询中检测到一个系统未知的关系名，将返回错误并且查询处理将被中止。

再接下来检查所用的属性名是否包含在查询给出的各个关系之中。对于找到的每一个属性，都会创建一个 TLE 节点，其中保存着一个指向 `Resdom` 节点（保存着列的名称）的指针和一个指

向 VAR 节点的指针。VAR 节点中有两个重要的编号：varno 字段给出包含当前属性的关系在上述范围表项列表中的位置；varattno 字段给出该属性在关系内部的位置。如果找不到某个属性的名称，将返回错误并且查询处理将被中止。

61.4. Postgres 规则系统

Postgres 提供了一个强大的规则系统，用于定义视图以及处理有歧义的视图更新。最初，Postgres 的规则系统由两种实现组成：

- 第一种实现使用元组级处理，并且深度实现于执行器内部。每当访问到单独一个元组时，规则系统就会被调用。这种实现在 1995 年被移除，当时 Postgres 项目的最后一个官方发行版被转换成了 Postgres95。
- 规则系统的第二种实现是一种称为查询重写的技术。
  重写系统是位于解析器阶段和规划器/优化器之间的一个模块。这种技术至今仍在使用。

有关 Postgres 系统中规则的语法和创建的信息，请参阅 The PostgreSQL User's Guide（PostgreSQL 用户指南）。

61.4.1. 重写系统

查询重写系统是位于解析器阶段和规划器/优化器之间的一个模块。
它处理解析器阶段返回的树（表示一个用户查询），如果存在必须应用于该查询的规则，
就把这棵树重写为另一种形式。

61.4.1.1. 实现视图的技术

下面我们将概述查询重写系统的算法。为了便于说明，我们以如何使用规则实现视图作为例子。

给定下面这条规则：

```
create rule view_rule
as on select
to test_view
do instead
   select s.sname, p.pname
   from supplier s, sells se, part p
   where s.sno = se.sno and
         p.pno = se.pno;
```

每当检测到对关系 test_view 的 select 时，上面给出的规则就会被触发。此时执行的将不是从 test_view 中选择元组，而是规则动作部分中给出的 select 语句。

再给定下面这个针对 test_view 的用户查询：

```
select sname
from test_view
where sname <> 'Smith';
```

下面列出每当出现针对 test_view 的用户查询时，查询重写系统所执行的各个步骤。（下面的列表只是对算法的非正式描述，仅用于基本理解。详细描述请参阅 Stonebraker et al, 1989。）

test_view 重写

1. 取规则动作部分中给出的查询。

2. 调整 targetlist，使其符合用户查询中给出的属性的数量和顺序。

3. 把用户查询 where 子句中给出的限定条件，加到规则动作部分所给查询的限定条件上。

依照上面给出的规则定义，用户查询将被重写成下面的形式（注意，
重写是在解析器阶段返回的用户查询内部表示上完成的，
但派生出的新数据结构将表示下面这个查询）：

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

61.5. 规划器/优化器

规划器/优化器的任务是创建一个最优的执行计划。
它首先把查询中出现的各个关系所有可能的扫描和连接方式组合起来。
所创建的所有路径都会得到同一结果，而优化器的任务就是估算执行每条路径的代价，
并找出哪一条代价最低。

61.5.1. 生成可能的计划

规划器/优化器根据查询中各关系上所定义索引的类型来决定应该生成哪些计划。
对一个关系总是可以执行顺序扫描，因此总会创建一个只使用顺序扫描的计划。
假设某个关系上定义了一个索引（例如一个 B-树索引），并且查询中包含限制条件 `relation.attribute OPR constant`。如果 `relation.attribute` 恰好匹配该 B-树索引的键，而且 `OPR` 不是 '<>'，就会再创建一个使用该 B-树索引扫描该关系的计划。如果还存在其他索引，并且查询中的限制条件恰好匹配某个索引的键，则还会考虑更多计划。

在找出扫描单个关系的所有可行计划之后，就会创建连接各关系的计划。规划器/优化器只考虑在 where 限定条件中存在相应连接子句（即存在类似 `where rel1.attr1=rel2.attr2` 这样的限制）的每两个关系之间的连接。对于规划器/优化器所考虑的每一对连接，都会生成所有可能的计划。三种可用的连接策略是：

- 嵌套迭代连接：左关系中找到的每一个元组，都会使右关系被扫描一次。
  这种策略实现起来很容易，但可能非常耗时。

- 归并排序连接：在连接开始之前，每个关系都会先按照连接属性排序。然后，
  考虑到两个关系都已经按连接属性排好序，把它们归并到一起。这种连接方式更有吸引力，
  因为每个关系只需扫描一次。

- 哈希连接：首先在右关系的连接属性上建立哈希。接着扫描左关系，
  并将找到的每个元组中的相应值作为哈希键，用来在右关系中定位元组。

61.5.2. 计划的数据结构

这里我们对计划中出现的节点做一点说明。图 \ref{plan} 展示了为例 \ref{simple_select} 中的查询生成的计划。

计划的顶层节点是一个 MergeJoin 节点，它有两个后继，一个挂接在 lefttree 字段上，另一个挂接在 righttree 字段上。每个子节点代表连接中的一个关系。如前所述，归并排序连接要求每个关系都已排序，因此我们在每个子计划中都会看到一个 Sort 节点。查询中给出的附加限定条件（s.sno > 2）被尽可能下推，挂接到相应子计划的叶子 SeqScan 节点的 qpqual 字段上。

MergeJoin 节点 mergeclauses 字段上挂接的列表包含有关连接属性的信息。mergeclauses 列表（以及 targetlist）中出现的 VAR 节点，其 varno 字段的值 65000 和 65001 表示：应当考虑的不是当前节点的元组，而是下一个“更深”节点（即各子计划的顶层节点）的元组。

注意，图 \ref{plan} 中出现的每一个 Sort 和 SeqScan 节点都有一个 targetlist，但由于空间不足，图中只画出了 MergeJoin 节点的那一个。

规划器/优化器的另一项任务是在 Expr 和 Oper 节点中固定操作符 id。如前所述，Postgres 支持多种不同的数据类型，甚至可以使用用户自定义的类型。为了能够维护数量庞大的函数和操作符，必须把它们存储在一张系统表中。每个函数和操作符都会得到一个唯一的操作符 id。根据限定条件等之中所使用属性的类型，必须使用相应的操作符 id。

61.6. 执行器

执行器接收由规划器/优化器创建的计划，并开始处理其顶层节点。对于我们的例子（例 \ref{simple_select} 中给出的查询）来说，顶层节点是一个 MergeJoin 节点。

在执行归并之前，必须先取到两个元组（每个子计划各一个）。
因此执行器会递归调用自身去处理这些子计划（从挂接在 lefttree 上的子计划开始）。新的顶层节点，也就是左子计划的顶层节点，是一个 SeqScan 节点，而在处理该节点本身之前又必须先取得一个元组。于是执行器再次递归调用自身，去处理 SeqScan 节点的 lefttree 上挂接的子计划。

现在新的顶层节点是一个 Sort 节点。由于排序必须在整个关系上进行，当 Sort 节点第一次被访问时，执行器开始从 Sort 节点的子计划取元组，并把它们排序后放入一个临时关系（内存中或文件中）。（之后再检查 Sort 节点时，总是只从排好序的临时关系中返回一个元组。）

每当 Sort 节点的处理需要一个新元组时，执行器就会被递归调用，去处理作为子计划挂接的 SeqScan 节点。在关系（内部通过 scanrelid 字段给出的值引用）中扫描下一个元组。如果该元组满足挂接在 qpqual 上的树所给出的限定条件，就把它交回；否则继续取下一个元组，直到限定条件被满足。如果关系中的最后一个元组已被处理完毕，就返回一个 NULL 指针。

在 MergeJoin 的 lefttree 交回一个元组之后，righttree 会以同样的方式被处理。如果两个元组都已就绪，执行器就处理 MergeJoin 节点。每当需要来自某个子计划的一个新元组时，都会递归调用执行器来取得它。如果能够构造出一个连接后的元组，就把它交回，
至此计划树的一次完整处理就结束了。

上述步骤会对每个元组执行一次，直到对 MergeJoin 节点的处理返回一个 NULL 指针，表明处理已经完成。

第 62 章 pg_options

Massimo Dal Zotto

> **注意**
>
> 由 Massimo Dal Zotto[1] 供稿

可选文件 `data/pg_options` 包含后端使用的运行时选项，用于控制跟踪消息和其他后端可调参数。这个文件的有趣之处在于，后端收到 SIGHUP 信号时会重新读取它，因此无需重启 Postgres 就可以即时更改运行时选项。该文件中指定的选项可以是跟踪包（`backend/utils/misc/trace.c`）使用的调试标志，也可以是后端用来控制其行为的数值参数。
新的选项和参数必须在 `backend/utils/misc/trace.c` 和 `backend/include/utils/trace.h` 中定义。

例如，假设我们想为文件 `foo.c` 中的代码添加条件跟踪消息和一个可调的数值参数。我们只需把常量 TRACE_FOO 和 OPT_FOO_PARAM 加入 `backend/include/utils/trace.h`：

```
/* file trace.h */
enum pg_option_enum {
    ...
    TRACE_FOO,    /* trace foo functions */
    OPT_FOO_PARAM,  /* foo tunable parameter */

    NUM_PG_OPTIONS              /* must be the last item of enum */
};
```

并在 `backend/utils/misc/trace.c` 中加入对应的一行：

```
/* file trace.c */
static char *opt_names[] = {
    ...
    "foo",              /* trace foo functions */
    "fooparam"           /* foo tunable parameter */
};
```

两个文件中的选项必须以完全相同的顺序指定。现在在 foo 源文件中就可以这样引用新的标志：

```
/* file foo.c */
#include "trace.h"
#define foo_param pg_options[OPT_FOO_PARAM]

int
foo_function(int x, int y)
{
    TPRINTF(TRACE_FOO, "entering foo_function, foo_param=%d", foo_
param);
```

[1] mailto:dz@cs.unitn.it

```
    if (foo_param > 10) {
        do_more_foo(x, y);
    }
}
```

使用私有跟踪标志的现有文件只需加上以下代码即可改造：

```
#include "trace.h"
/* int my_own_flag = 0; -- removed */
#define my_own_flag pg_options[OPT_MY_OWN_FLAG]
```

所有 pg_options 在后端启动时都被初始化为零。如果需要不同的默认值，必须在 `Postgres Main` 开头添加一些初始化代码。现在我们可以通过向 `data/pg_options` 文件写入值来设置 foo_param 并启用 foo 跟踪：

```
# file pg_options
...
foo=1
fooparam=17
```

新选项会在每个新后端启动时被读取。要让更改对所有正在运行的后端生效，需要向 postmaster 发送 SIGHUP。该信号会自动被发送到所有后端。也可以直接向某个特定后端发送 SIGHUP，只对它激活更改。

pg_options 也可以通过 Postgres 的 `-T` 开关来指定：

```
postgres options -T "verbose=2,query,hostlookup-"
```

用于打印错误和调试消息的函数现在可以使用 syslog(2) 设施。打印到 stdout 或 stderr 的消息都带有一个时间戳前缀，其中还包含后端 pid：

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

这种格式提高了日志的可读性，让人能准确了解哪个后端在什么时间做了什么。
它也让编写简单的 awk 或 perl 脚本来监控日志、检测数据库错误或问题、或计算事务时间统计变得更容易。

打印到 syslog 的消息使用日志设施 LOG_LOCAL0。是否使用 syslog 可以用 syslog pg_option 控制。遗憾的是，许多函数直接调用 `printf()` 把消息打印到 stdout 或 stderr，这样的输出既无法重定向到 syslog，也无法加上时间戳。最好把所有对 printf 的调用都替换为 PRINTF 宏，并把输出到 stderr 的调用改为使用 EPRINTF，这样我们就能以统一的方式控制所有输出。

新的 pg_options 机制比定义新的后端选项开关更方便，原因如下：

- 我们不必为每个想要控制的东西定义一个不同的开关。
  所有选项都以关键字的形式定义在数据目录中的一个外部文件里。
- 更改某个选项的设置不需要重启 Postgres。通常后端选项指定给 postmaster，并在每个后端启动时传递给它。现在它们从文件中读取。
- 我们可以在后端运行时即时更改选项。
  这样就可以只在问题出现时才激活调试消息来调查问题。还可以为可调参数尝试不同的值。

pg_options 文件的格式如下：

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

注意 *keyword* 也可以是 backend/utils/misc/trace.c 中定义的选项名的缩写。

关于当前支持的选项的完整列表，请参考管理员指南中关于运行时选项的章节。

一些使用私有变量和选项开关的现有代码已被改为使用 pg_options 特性，主要是在 postgres.c 中。最好把所有现有代码都改成这种方式，这样我们就可以去掉 Postgres 命令行上的许多开关，并拥有更多可调选项，而选项值集中放在一个地方。

第 63 章 数据库系统中的遗传查询优化

Martin Utesch, University of Mining and Technology

作者

由 Martin Utesch[1] 为德国弗赖贝格矿业和技术大学自动控制研究所编写。

63.1. 将查询处理看成是一个复杂的优化问题

在所有关系操作符中，最难处理和优化的是连接。随着查询中join数目的增加，
可能的查询计划数量会呈指数增长。为了处理单个连接而支持多种连接方法（例如 Postgres 中的嵌套循环、索引扫描和归并连接）来处理单个 join，以及作为关系访问路径的多种索引（例如 Postgres 中的 r-树、b-树和哈希），也进一步增加了优化工作量。

当前的Postgres查询优化器实现会在可选策略空间中执行近似穷举搜索。
这种查询优化技术不足以支持诸如人工智能这类需要大量查询的数据库应用领域。

德国弗赖贝格矿业和技术大学自动控制研究所在尝试将 Postgres 用作一个用于电网维护的基于知识的决策支持系统后端时遇到了一些问题。该 DBMS 需要为该基于知识系统的推理机处理大型 join 查询。

在探索可能查询计划空间内部的性能困难，产生了开发一种新优化技术的需求。

下文我们提出把遗传算法的实现作为数据库查询优化问题的一种可选方案。

63.2. 遗传算法（GA）

GA 是一种通过确定的、随机化的搜索进行工作的启发式优化方法。优化问题的可能解集合被视为由若干个体组成的种群。个体对其环境的适应程度由其适应度指定。

一个个体在搜索空间中的坐标由染色体表示，本质上是一组字符串。基因是染色体的一个片段，它编码某个待优化参数的值。基因的典型编码可以是二进制或整数。

通过模拟重组、变异和选择这些进化操作，可以找到平均适应度高于前代的新一代搜索点。

根据"comp.ai.genetic" FAQ 的说法，再怎么强调也不过分：GA并不是为了求解问题而进行的纯粹随机搜索。GA会使用随机过程，但其结果显然并非随机的（优于随机）。

```
Structured Diagram of a GA:
---------------------------

P(t)    generation of ancestors at a time t
P''(t)  generation of descendants at a time t

+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
```

[1] utesch@aut.tu-freiberg.de

```
| INITIALIZE P(t)                          |
+==========================================+
| evalute FITNESS of P(t)                  |
+==========================================+
| while not STOPPING CRITERION do          |
|   +--------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}        |
|   +--------------------------------------+
|   | P''(t) := MUTATION{P'(t)}            |
|   +--------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}   |
|   +--------------------------------------+
|   | evalute FITNESS of P''(t)            |
|   +--------------------------------------+
|   | t := t + 1                           |
+===+======================================+
```

63.3. Postgres 中的遗传查询优化（GEQO）

GEQO模块旨在以类似于旅行商问题（TSP）的方式解决查询优化问题。
可能的查询计划被编码为整数串。每个串表示查询中从一个关系到下一个关系的 `join` 顺序。例如，查询树

```
       /\
      /\ 2
     /\ 3
    4  1
```

会被编码为整数串 '4-1-3-2'，这意味着先连接关系 '4' 和 '1'，再连接 '3'，最后连接 '2'；其中 1、2、3、4 是 Postgres 中的 relid。

GEQO模块的部分内容改编自 D. Whitley 的 Genitor 算法。

Postgres 中 GEQO 实现的具体特征是：

- 采用稳态 GA（替换种群中适应度最低的个体，而不是整代替换），能够快速收敛到更优的查询计划。这对于在合理时间内处理查询至关重要；
- 采用边重组交叉，它特别适合在利用 GA 求解 TSP 时将边损失保持在较低水平；
- 不使用变异作为遗传操作符，因此无须借助修复机制来生成合法的 TSP 回路。

与 Postgres 查询优化器实现相比，GEQO 模块为 Postgres DBMS 带来以下益处：

- 通过非穷举搜索处理大型 `join` 查询；
- 改进了查询计划的代价大小估算，因为不再需要计划合并（GEQO 模块把一个查询计划的代价作为一个个体来评估）。

63.4. Postgres GEQO 的未来实现任务

63.4.1. 基本改进

63.4.1.1. 改进查询处理完成后的内存释放

对于大型 `join` 查询，遗传查询优化所花费的计算时间似乎只是 Postgres 通过例程 `Memory ContextFree`（文件 `backend/utils/mmgr/mcxt.c`）释放内存所用时间的一个小分数。调

试显示它卡在了例程 `OrderedElemPop`（文件 `backend/utils/mmgr/oset.c`）的一个循环中。使用普通 Postgres 查询优化算法处理长查询时也会出现同样的问题。

63.4.1.2. 改进遗传算法参数设置

在文件 `backend/optimizer/geqo/geqo_params.c` 的例程 `gimme_pool_size` 和 `gimme_number_generations` 中，我们必须为参数设置找到一种折中，以满足两个相互竞争的需求：

- 查询计划的最优性
- 计算时间

63.4.1.3. 为整数溢出寻找更好的解决方案

在文件 `backend/optimizer/geqo/geqo_eval.c` 的例程 `geqo_joinrel_size` 中，目前对 MAXINT 溢出的临时做法是把 Postgres 整数值 `rel->size` 设为其对数。对 `backend/nodes/relation.h` 中 `Rel` 的修改肯定会对整个 Postgres 实现产生严重影响。

63.4.1.4. 为内存耗尽寻找解决方案

当查询涉及的关系超过 10 个时可能发生内存耗尽。在文件 `backend/optimizer/geqo/geqo_eval.c` 中，例程 `gimme_tree` 会被递归调用。也许我忘了正确释放某些东西，但我不知道是什么。当然，`join` 的 `rel` 数据结构会随着装进去的关系越来越多而不断增长。欢迎建议 :-(

参考文献

GEQ 算法的参考信息。

The Hitch-Hiker's Guide to Evolutionary Computation . Jörg Heitkötter和David Beasley. InterNet resource .

comp.ai.genetic[1] 中的 FAQ 见 Encore[2]。

The Design and Implementation of the Postgres Query Optimizer . Z. Fong. University of California, Berkeley Computer Science Department .

文件 `planner/Report.ps`（在 'postgres-papers' 发行版中）。

Fundamentals of Database Systems . R. Elmasri和S. Navathe. The Benjamin/Cummings Pub., Inc. .

[1] news://comp.ai.genetic
[2] ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html

第 64 章 前端/后端协议

Phil Thompson

> **注意**
>
> 由 Phil Thompson 编写。协议 2.0 的更新由 Tom Lane 完成。

Postgres 使用基于消息的前端与后端之间的通信协议。该协议在 TCP/IP 以及 Unix 套接字之上实现。Postgres v6.3 在协议中引入了版本号。这样做的目的是仍然允许来自早期版本前端的连接，但本文档不涵盖那些早期版本使用的协议。

本文档描述该协议的 2.0 版，它在 Postgres v6.4 及更高版本中实现。

建立在此协议之上的更高层特性（例如 libpq 在连接建立后如何传递某些环境变量）在别处介绍。

64.1. Overview

三个主要组件是前端（运行在客户端上）以及 postmaster 和后端（运行在服务器上）。postmaster 和后端有不同的角色，但可以由同一个可执行程序实现。

前端向 postmaster 发送一个启动包。其中包括用户要连接的用户名和数据库名。然后 postmaster 使用这些信息以及 pg_hba.conf(5) 文件中的信息来确定它要求前端进一步发送什么认证信息（如果有的话），并相应地响应前端。

然后前端发送任何所需的认证信息。postmaster 验证这些信息后，响应前端已通过认证，并把连接移交给一个后端。然后后端发送一条消息指示启动成功（正常情况）
或失败（例如数据库名无效）。

随后的通信是前端与后端之间交换的查询和结果包。postmaster 不再参与普通的查询/结果通信。(However, the postmaster is involved when the frontend wishes to cancel a query currently being executed by its backend. Further details about that appear below.)

当前端希望断开连接时，它发送一个适当的包并关闭连接，而不等待后端的响应。

包以数据流的形式发送。第一个字节决定包的其余部分应期待什么。例外是前端发送给 postmaster 的包，它们由包长度后跟包本身组成。这一差异是历史原因造成的。

64.2. Protocol

本节描述消息流。有四种不同的流，取决于连接的状态：启动、查询、函数调用和终止。
还有针对通知响应和命令取消的特殊规定，它们可以在启动阶段之后的任何时间发生。

64.2.1. Startup

启动分为认证阶段和后端启动阶段。

最初，前端发送一个 StartupPacket。postmaster 使用这些信息和 pg_hba.conf(5) 文件的内容来确定前端必须使用什么认证方法。然后 postmaster 以下列消息之一响应：

ErrorResponse

然后 postmaster 立即关闭连接。

AuthenticationOk

然后 postmaster 移交给后端。postmaster 不再参与通信。

AuthenticationKerberosV4

然后前端必须与 postmaster 进行 Kerberos V4 认证对话（此处不描述）。如果成功，postmaster 以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationKerberosV5

然后前端必须与 postmaster 进行 Kerberos V5 认证对话（此处不描述）。如果成功，postmaster 以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationUnencryptedPassword

然后前端必须发送一个 UnencryptedPasswordPacket。如果密码正确，postmaster 以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationEncryptedPassword

然后前端必须发送一个 EncryptedPasswordPacket。如果密码正确，postmaster 以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

如果前端不支持 postmaster 请求的认证方法，那么它应当立即关闭连接。

发送 AuthenticationOk 之后，postmaster 尝试启动一个后端进程。由于这可能失败，或者后端可能在启动期间遇到失败，前端必须等待后端确认启动成功。前端在此点不应发送任何消息。
此阶段后端可能的消息有：

BackendKeyData

此消息在后端成功启动后发出。
它提供前端想要以后能够发出取消请求就必须保存的密钥数据。前端不应响应此消息，而应继续等待 ReadyForQuery 消息。

ReadyForQuery

后端启动成功。前端现在可以发出查询或函数调用消息。

ErrorResponse

后端启动失败。发送此消息后连接被关闭。

NoticeResponse

已发出一条警告消息。前端应显示该消息，但仍应继续等待 ReadyForQuery 或 ErrorResponse。

ReadyForQuery 消息与后端在每个查询周期之后将发出的消息相同。取决于前端的编码需要，既可以把 ReadyForQuery 视为一个查询周期的开始（然后 BackendKeyData 表示启动阶段的成功结束），也可以把 ReadyForQuery 视为启动阶段和每个后续查询周期的结束。

64.2.2. Query

查询周期由前端向 backend 发送 Query 消息发起。然后后端根据查询命令串的内容发送一个或多个响应消息，最后发送 ReadyForQuery 响应消息。ReadyForQuery informs the frontend that it may safely send a new query or function call.

后端可能发送的响应消息有：

CompletedResponse

一条 SQL 命令正常完成。

CopyInResponse

后端准备好把数据从前端复制到关系。前端然后应发送 CopyDataRows 消息。后端随后将以带 "COPY" 标签的 CompletedResponse 消息响应。

CopyOutResponse

后端准备好把数据从关系复制到前端。然后它发送 CopyDataRows 消息，再发送带 "COPY" 标签的 CompletedResponse 消息。

CursorResponse

查询是 insert(l)、delete(l)、update(l)、fetch(l) 或 select(l) 命令之一。如果事务已被中止，则后端发送带 "*ABORT STATE*" 标签的 CompletedResponse 消息。否则发送以下响应。

对于 insert(l) 命令，后端然后发送带 "INSERT *oid rows*" where *rows* is the number of rows inserted, and *oid* is the object ID of the inserted row if *rows* is 1, otherwise *oid* is 0.

对于 delete(l) 命令，后端然后发送带 "DELETE *rows*" where *rows* is the number of rows deleted.

对于 update(l) 命令，后端然后发送带 "UPDATE *rows*" where *rows* is the number of rows deleted.

对于 fetch(l) 或 select(l) 命令，后端发送一条 RowDescription 消息。然后对返回给前端的每一行，跟着一条 AsciiRow 或 BinaryRow 消息（取决于是否指定了二进制游标）。最后，后端发送一条带 "SELECT" 标签的 CompletedResponse 消息。

EmptyQueryResponse

识别到一个空查询串。（需要特别区分这种情况是历史原因。）

ErrorResponse

发生了一个错误。

ReadyForQuery

查询串的处理完成。发送单独的消息来指示这一点，因为查询串可能包含多条 SQL 命令。(CompletedResponse marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

发出了与查询有关的警告消息。通知是其他响应之外的附加内容，
即后端将继续处理该命令。

每当期待任何其他类型的消息时，前端必须准备好接受 ErrorResponse 和 NoticeResponse 消息。

实际上，即使前端不期待任何类型的消息（即后端名义上空闲）时，NoticeResponse 也可能到达。（特别是，后端可能被其 postmaster 命令终止。在这种情况下它会在关闭连接之前发送一条 NoticeResponse。）建议前端在发出任何新命令之前检查此类异步通知。

此外，如果前端发出任何 listen(l) 命令，那么它必须准备好在任何时间接受 NotificationResponse 消息；见下文。

64.2.3. 函数调用

函数调用周期由前端向后端发送一条 FunctionCall 消息来启动。后端随后根据函数调用的结果发送一条或多条响应消息，最后发送一条 ReadyForQuery 响应消息。ReadyForQuery 告知前端，可以安全地发送新的查询或函数调用。

后端可能发送的响应消息有：

ErrorResponse

发生了一个错误。

FunctionResultResponse

函数调用已执行并返回了结果。

FunctionVoidResponse

函数调用已执行但没有返回结果。

ReadyForQuery

函数调用处理完成。ReadyForQuery将总是被发送，
不管是成功完成处理还是发生一个错误。

NoticeResponse

发出了与函数调用有关的警告消息。通知是其他响应之外的附加内容，
即后端将继续处理该命令。

每当期待任何其他类型的消息时，前端必须准备好接受 ErrorResponse 和 NoticeResponse 消息。Also, if it issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

64.2.4. 通知响应

如果前端发出 listen(l) 命令，那么每当为相同的通知名称执行 notify(l) 命令时，后端都会发送一条 NotificationResponse 消息（不要与 NoticeResponse 混淆！）。

通知响应在协议的任何位置（启动之后）都是允许的，除非在另一个后端消息之内。Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

A notify(l) command has been executed for a name for which a previous listen(l) command was executed. Notifications may be sent at any time.

可能值得指出，listen 和 notify 命令中使用的名称不必与 SQL 数据库中关系（表）的名称有任何关系。通知名称只是任意选择的条件名称。

64.2.5. 取消进行中的请求

在处理查询期间，前端可以通过向 postmaster 发送一个适当的请求来请求取消该查询。出于实现效率的原因，
取消请求不直接发送给后端：我们不希望后端在查询处理期间不断检查来自前端的新输入。
取消请求应当相对少见，所以我们把它们做得稍微繁琐一些，以避免正常情况下的性能损失。

要发出取消请求，前端打开一个到 postmaster 的新连接并发送 CancelRequest 消息，而不是通常通过新连接发送的 StartupPacket 消息。postmaster 将处理此请求然后关闭连接。出于安全原因，不对取消请求消息作直接回复。

CancelRequest 消息将被忽略，除非它包含与连接启动期间传递给前端相同的密钥数据（PID 和密钥）。如果该请求与某个当前正在执行的后端的 PID 和密钥匹配，postmaster 向该后端发出信号以中止当前查询的处理。

取消信号可能有效也可能无效——例如，如果它到达时后端已经处理完查询，
那么它就不会有效果。如果取消有效，则会导致当前命令提前终止并附带一条错误消息。

这么做是对安全性和效率通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。
它必须继续等待后端对查询的响应。发出一个取消仅仅是增加了当前查询快些结束的可能性，
同时也增加了当前查询会伴随着一条错误消息失败而不是成功执行的可能性。

由于取消请求是发送给 postmaster 而不是通过常规的前端/后端通信链路，任何进程都可能发出取消请求，而不只是其查询要被取消的那个前端。
这在构建多进程应用方面可能带来一些灵活性上的好处。它也引入了安全风险，
即未授权的人可能试图取消查询。
这一安全风险通过要求在取消请求中提供动态生成的密钥来解决。

64.2.6. Termination

正常的优雅终止过程是前端发送 Terminate 消息并立即关闭连接。收到该消息后，后端立即关闭连接并终止。

不优雅的终止可能由于任一端的软件故障（即核心转储）而发生。
如果前端或后端看到连接意外关闭，应当清理并终止。The frontend has the option of launching a new backend by recontacting the postmaster, if it doesn't want to terminate itself.

64.3. 消息数据类型

本节描述消息中使用的基本数据类型。

Int*n*(*i*)

一个按网络字节序的 n 位整数。如果指定了 i，它就是字面值。Eg. Int16, Int32(42).

LimString*n*(*s*)

恰好 n 字节的字符数组，解释为以 '\0' 结尾的字符串。如果空间不足则省略 '\0'。如果指定了 s，它就是字面值。例如 LimString32、LimString64("user")。

String(*s*)

常规的以 '\0' 结尾的 C 字符串，没有长度限制。如果指定了 s，它就是字面值。例如 String、String("user")。

> **注意**
>
> 后端可返回的字符串的长度没有预定义的限制。
> 前端的一个好编码策略是使用可扩展的缓冲区，
> 这样任何能装进内存的东西都可以接受。如果做不到这一点，
> 就读取完整的字符串并丢弃固定大小缓冲区装不下的尾部字符。

Byte*n*(*c*)

恰好 n 字节。如果指定了 c，它就是字面值。例如 Byte、Byte1('\n')。

64.4. 消息格式

本节描述每种消息的详细格式。每种消息可以由前端（F）、postmaster/后端（B）或两者（F & B）发送。

AsciiRow (B)

Byte1('D')

标识此消息为 ASCII 数据行。（先前的 RowDescription 消息定义了行中字段的数量及其数据类型。）

Byte*n*

一个位图，行中每个字段一位。第 1 个字段对应第 1 个字节的第 7 位（MSB），第 2 个字段对应第 1 个字节的第 6 位，第 8 个字段对应第 1 个字节的第 0 位（LSB），第 9 个字段对应第 2 个字节的第 7 位，依此类推。如果对应字段的值不是 NULL，则每个位被置位。如果字段数不是 8 的倍数，位图中最后一个字节的剩余部分就被浪费了。

然后，对每个非 NULL 值的字段，有下列内容：

Int32

Specifies the size of the value of the field, including this size.

Byte*n*

以 ASCII 字符指定字段本身的值。n 是上面的尺寸减 4。字段数据中没有尾随的 '\0'；如果前端需要，必须自己添加一个。

AuthenticationOk (B)

Byte1('R')

标识此消息为认证请求。

Int32(0)

指定认证成功。

AuthenticationKerberosV4 (B)

Byte1('R')

标识此消息为认证请求。

Int32(1)

指定需要 Kerberos V4 认证。

AuthenticationKerberosV5 (B)

Byte1('R')

标识此消息为认证请求。

Int32(2)

指定需要 Kerberos V5 认证。

AuthenticationUnencryptedPassword (B)

Byte1('R')

标识此消息为认证请求。

Int32(3)

指定需要未加密的密码。

AuthenticationEncryptedPassword (B)

Byte1('R')

标识此消息为认证请求。

Int32(4)

指定需要加密的密码。

Byte2

加密密码时使用的盐。

BackendKeyData (B)

Byte1('K')

标识此消息为取消密钥数据。如果前端希望以后能够发出 CancelRequest 消息，就必须保存这些值。

Int32

此后端的进程 ID。

Int32

此后端的密钥。

BinaryRow (B)

Byte1('B')

标识此消息为二进制数据行。（先前的 RowDescription 消息定义了行中字段的数量及其数据类型。）

Byte*n*

一个位图，行中每个字段一位。第 1 个字段对应第 1 个字节的第 7 位（MSB），第 2 个字段对应第 1 个字节的第 6 位，第 8 个字段对应第 1 个字节的第 0 位（LSB），第 9 个字段对应第 2 个字节的第 7 位，依此类推。如果对应字段的值不是 NULL，则每个位被置位。如果字段数不是 8 的倍数，位图中最后一个字节的剩余部分就被浪费了。

然后，对于每个具有非 NULL 值的字段，有以下内容：

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

以二进制格式指定字段本身的值。*n* is the above size.

CancelRequest (F)

Int32(16)

包的尺寸（字节）。

Int32(80877102)

取消请求码。该值经过选择，使其高 16 位包含 1234，低 16 位包含 5678。(To avoid confusion, this code must not be the same as any protocol version number.)

Int32

目标后端的进程 ID。

Int32

目标后端的密钥。

CompletedResponse (B)

Byte1('C')

标识此消息为完成响应。

String

命令标签。这通常是（但不总是）标识完成了哪条 SQL 命令的单个词。

CopyDataRows (B & F)

这是一个行流，其中每行以 Byte1('\n') 结束。然后跟随序列 Byte1('\\')、Byte1('.')、Byte1('\n')。

CopyInResponse (B)

Byte1('G')

标识此消息为 Start Copy In 响应。前端现在必须发送 CopyDataRows 消息。

CopyOutResponse (B)

Byte1('H')

标识此消息为 Start Copy Out 响应。此消息后面将跟随 CopyDataRows 消息。

CursorResponse (B)

Byte1('P')

标识此消息为游标响应。

String

游标的名称。如果游标是隐式的，则为“blank”。

EmptyQueryResponse (B)

Byte1('I')

标识此消息为对空查询串的响应。

String("")

未使用。

EncryptedPasswordPacket (F)

Int32

包的尺寸（字节）。

String

加密（使用 crypt()）的密码。

ErrorResponse (B)

Byte1('E')

标识此消息为错误。

String

错误消息本身。

FunctionCall (F)

Byte1('F')

标识此消息为函数调用。

String("")

未使用。

Int32

指定要调用的函数的对象 ID。

Int32

指定提供给函数的参数数量。

然后，对每个参数，有下列内容：

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

以二进制格式指定字段本身的值。*n* is the above size.

FunctionResultResponse (B)

Byte1('V')

标识此消息为函数调用结果。

Byte1('G')

指定返回了非空结果。

Int32

指定结果的值的尺寸，不包括此尺寸本身。

Byte*n*

以二进制格式指定结果本身的值。n 是上面的尺寸。

Byte1('0')

未使用。（严格说来，FunctionResultResponse 和 FunctionVoidResponse 是同一个东西，只是消息的某些部分是可选的。）

FunctionVoidResponse (B)

Byte1('V')

标识此消息为函数调用结果。

Byte1('0')

指定返回了空结果。

NoticeResponse (B)

Byte1('N')

标识此消息为通知。

String

通知消息本身。

NotificationResponse (B)

Byte1('A')

标识此消息为通知响应。

Int32

发出通知的后端进程的进程 ID。

String

引发此通知的条件名称。

Query (F)

Byte1('Q')

标识此消息为查询。

String

查询串本身。

ReadyForQuery (B)

Byte1('Z')

标识消息类型。每当后端准备好进入新的查询周期时发送 ReadyForQuery。

RowDescription (B)

Byte1('T')

标识此消息为行描述。

Int16

指定一行中的字段数（可以为零）。

Then, for each field, there is the following:

String

指定字段名。

Int32

指定字段类型的对象 ID。

Int16

指定类型尺寸。

Int32

指定类型修饰符。

StartupPacket (F)

Int32(296)

包的尺寸（字节）。

Int32

协议版本号。高 16 位是主版本号。低 16 位是次版本号。

LimString64

数据库名，如果为空则默认为用户名。

LimString32

用户名。

LimString64

要由 postmaster 传递给后端的任何附加命令行参数。

LimString64

未使用。

LimString64

后端应用于调试消息的可选 tty。

Terminate (F)

Byte1('X')

标识此消息为终止。

UnencryptedPasswordPacket (F)

Int32

包的尺寸（字节）。

String

未加密的密码。

第 65 章 Postgres 信号

Massimo Dal Zotto

> **注意**
>
> 由 Massimo Dal Zotto[1] 贡献

Postgres 使用下列信号在 postmaster 和后端之间通信：

表 65.1. Postgres 信号

信号	postmaster 动作	服务器动作
SIGHUP	kill(*,sighup)	read_pg_options
SIGINT	die	cancel query
SIGQUIT	kill(*,sigterm)	handle_warn
SIGTERM	kill(*,sigterm), kill(*,9), die	die
SIGPIPE	ignored	die
SIGUSR1	kill(*,sigusr1), die	quickdie
SIGUSR2	kill(*,sigusr2)	async notify (SI flush)
SIGCHLD	reaper	ignored (alive test)
SIGTTIN	ignored	
SIGTTOU	ignored	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptionHandler

> **注意**
>
> "`kill(*,signal)`" 表示向所有后端发送一个信号。

与旧信号处理方式相比的主要变化是：用 SIGQUIT 而不是 SIGHUP 来处理 warn，用 SIGHUP 重新读取 pg_options 文件，以及把发送给 postmaster 的 SIGHUP、SIGTERM、SIGUSR1 和 SIGUSR2 重定向到所有活动后端。这样，发送给 postmaster 的这些信号就可以自动发送给所有后端，而无需知道它们的 pid。要关闭 postgres，只需向 postmaster 发送一个 SIGTERM，它会自动停止所有后端。

SIGUSR2 信号还用于防止 SI 缓存表溢出，这种情况发生在某个后端长时间不处理 SI 缓存时。当一个后端检测到 SI 表在 70% 满时，它就向 postmaster 发送一个信号，后者会唤醒所有空闲的后端并让它们刷写缓存。

程序员对信号的典型用法可能如下：

```
# stop postgres
kill -TERM $postmaster_pid
```

[1] mailto:dz@cs.unitn.it

```
# kill all the backends
kill -QUIT $postmaster_pid

# kill only the postmaster
kill -INT $postmaster_pid

# change pg_options
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $postmaster_pid

# change pg_options only for a backend
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $backend_pid
cat old_pg_options > $DATA_DIR/pg_options
```

第 66 章 gcc 默认优化

Brian Gallew

> **注意**
>
> 由 Brian Gallew[1] 贡献

把 gcc 配置成默认使用某些标志只是编辑 `/usr/local/lib/gcc-lib/`*`platform`*`/`*`version`*`/specs` 文件的一件简单事。这个文件的格式相当简单。文件分成若干节，每节三行。第一行是 "*`section_name`*:"（例如 "*asm:"）。第二行是标志列表，第三行为空。

最容易的修改是把想要的默认标志追加到相应节的列表中。作为示例，假设我在一台 '486 上运行 linux，gcc 2.7.2 安装在默认位置。在文件 /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs 的第 13 行处我找到下面这一节：

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

如你所见，没有任何默认标志。如果我希望 C 代码的编译总是使用 "-m486 -fomit-frame-pointer"，我会把它改成这样：

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

如果我还想为另一台闲置的较旧 linux 机器生成 386 代码，就必须把它改成这样：

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

这样就会总是省略帧指针，并且在命令行上未指定 -m386 时生成 486 优化的代码。

实际上你可以用 specs 文件做相当多的定制。但始终要记住，这些更改是全局的，会影响系统的所有用户。

[1] mailto:geek+@cmu.edu

第 67 章 后端接口

后端接口（BKI）文件是这样一些脚本：它们是运行在特殊"引导"模式下的 Postgres 后端的输入，这种模式使后端能够在数据库系统尚不存在的情况下执行数据库功能。因此 BKI 文件可以用于最初创建数据库系统。initdb 正是使用 BKI 文件来完成这件事：创建数据库系统。不过，initdb 的 BKI 文件是内部生成的。它使用 `global1.bki.source` 和 `local1.template1.bki.source` 这两个文件来生成它们，这两个文件可以在 Postgres 的"库"目录中找到。它们作为安装 Postgres 的一部分被安装到那里。这些 .source 文件作为 Postgres 构建过程的一部分，由一个名为 genbki 的构建程序构建。genbki 接收 Postgres 源文件作为输入，这些源文件同时充当 genbki 的输入，用以构建表以及描述这些表的 C 头文件。

相关信息可以在下列文档中找到：initdb、createdb，以及 SQL 命令 `CREATE DATABASE`。

67.1. BKI 文件格式

Postgres 后端按下面的描述解释 BKI 文件。如果 `global1.bki.source` 文件在手边作为示例，这段说明会更容易理解。（如上面所解释的，这个 .source 文件并不完全是一个 BKI 文件，但你还是能猜出得到的 BKI 文件会是什么样子。）

命令由命令名后跟以空格分隔的参数组成。以 "$" 开头的命令参数会被特殊处理。如果前两个字符是 "$$"，那么第一个 "$" 被忽略，该参数随后按正常方式处理。如果 "$" 后面跟的是空格，那么它被当作一个 NULL 值。否则，"$" 后面的字符被解释为一个宏的名字，使该参数被替换为该宏的值。如果这个宏未定义，则是一个错误。

宏用

```
define macro macro_name = macro_value
```

定义，用

```
undefine macro macro_name
```

取消定义，重新定义使用与 define 相同的语法。

下面是通用命令和宏命令的列表。

67.2. 通用命令

OPEN *classname*

打开名为 *classname* 的类以便进一步操作。

CLOSE [*classname*]

关闭名为 *classname* 的已打开类。如果 *classname* 尚未打开，则这是一个错误。如果没有给出 *classname*，则关闭当前打开的类。

PRINT

打印当前打开的类。

INSERT [OID=*oid_value*] (*value1 value2* ...)

用 *value1*、*value2*、等作为属性值，用 *oid_value* 作为其 OID，向已打开的类中插入一个新的实例。如果 *oid_value* 不为零（0），则该值将被用作该实例的对象标识符。否则，这是一个错误。

INSERT (*value1 value2* ...)

同上，但由系统生成一个唯一的对象标识符。

CREATE *classname* (*name1* = *type1* [,*name2* = *type2*[,...]])

创建一个名为 *classname* 的类，其属性在圆括号中给出。

OPEN (*name1* = *type1* [,*name2* = *type2*[,...]]) AS *classname*

打开一个名为 *classname* 的类用于写入，但不把它的存在记录到系统目录中。（这主要是为了帮助引导。）

DESTROY *classname*

销毁名为 *classname* 的类。

DEFINE INDEX *indexname* ON *class_name* USING *amname* (*opclass attr* | (*function* (*attr*))

用 *amname* 访问方法在名为 *classname* 的类上创建一个名为 *indexname* 的索引。要索引的字段称为 *name1*、*name2* 等，而要使用的操作符集合分别为 *collection_1*、*collection_2* 等。

> **注意**
>
> 这最后一句话没有引用示例中的任何东西。应该改成有意义的内容。- Thomas 1998-08-04

67.3. 宏命令

DEFINE FUNCTION *macro_name* AS *rettype function_name*(*args*)

为一个名为 *macro_name* 的函数定义函数原型，该函数的类型为 *rettype* 的值由执行带参数 *args* 的 *function_name* 计算得出，参数以类似 C 的方式声明。

DEFINE MACRO *macro_name* FROM FILE *filename*

定义一个名为 *macro_name* 的宏，它的值从名为 *filename* 的文件中读出。

67.4. 调试命令

> **注意**
>
> 关于调试命令的这一节在原始文档中是被注释掉的。Thomas 1998-08-05

r

随机打印打开的类。

m -1

切换时间信息的显示。

m 0

把检索设置为当前时间（now）。

m 1 Jan 1 01:00:00 1988

把检索设置为指定时间的快照。

m 2 Jan 1 01:00:00 1988, Feb 1 01:00:00 1988

把检索设置为指定时间范围的快照。如果想要无界的时间范围，
任一时间都可以用空格代替。

&A *classname natts name1 type1 name2 type2 ...*

向类 *classname* 添加 *natts* 个名为 *name1*、*name2*、等的属性，其类型分别为 *type1*、*type2*、等。

&RR *oldclassname newclassname*

把 *oldclassname* 类重命名为 *newclassname*。

&RA classname oldattname newattname *classname oldattname newattname*

把名为 *classname* 的类中名为 *oldattname* 的属性重命名为 *newattname*。

67.5. 示例

下面的命令集将创建 pg_opclass 类，它包含 *int_ops* 集合，作为一个 OID 为 *421* 的对象，然后打印该类并关闭它。

```
create pg_opclass (opcname=name)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```

第 68 章 页文件

对数据库文件默认页格式的描述。

本节概述 Postgres 类使用的页格式。用户定义的访问方法不必使用这种页格式。

在下面的说明中，假定一个字节（byte）包含 8 位。此外，术语项（item）指存储在 Postgres 类中的数据。

68.1. 页结构

下表展示了普通 Postgres 类和 Postgres 索引类（例如 B-tree 索引）中的页面是如何组织的。

表 68.1. 页面布局示例

项	描述
itemPointerData	
filler	
itemData...	
未分配空间	
ItemContinuationData	
特殊空间	
``ItemData 2''	
``ItemData 1''	
ItemIdData	
PageHeaderData	

每个页面的前 8 个字节由页头（PageHeaderData）组成。在页头中，前三个 2 字节整数字段（lower、upper 和 special）分别表示到未分配空间开始处、到未分配空间结束处以及到特殊空间开始处的字节偏移。特殊空间是页面末尾的一个区域，它在页面初始化时分配，
并包含某种访问方法特有的信息。页头的最后 2 个字节 opaque 编码页面大小和页面内部碎片的信息。每个页面都存储页面大小，
因为缓冲池中的帧在一个类内可以逐帧细分为大小相等的页面。
内部碎片信息用来帮助确定何时应进行页面重组。

页头之后是项标识符（ItemIdData）。新的项标识符从未分配空间的前四个字节分配。
由于项标识符在被释放之前从不移动，其索引可用来指示一个项在页面上的位置。事实上，
Postgres 创建的每个指向项的指针（ItemPointer）都由一个帧号和一个项标识符的索引组成。
项标识符包含到项开始处的字节偏移、以字节为单位的长度以及一组影响其解释的属性位。

项本身存储在从未分配空间末尾向后分配的空间中。通常不对项进行解释。
但当项太长而无法放在单个页面上或希望对项进行分片时，项会被切分，
每一片按下述方式作为不同的项处理。第一片到倒数第二片放在项延续结构（
ItemContinuationData）中。该结构包含指向下一片的 itemPointerData 和这一片本身。最后一片按正常方式处理。

68.2. 文件

`data/`

共享（全局）数据库文件的位置。

`data/base/`

本地数据库文件的位置。

68.3. 缺陷

页格式将来可能改变，以提供对大对象更高效的访问。

本节包含的细节不足以对编写新的访问方法提供任何帮助。

部分 VI. 教程

新用户入门。

目录

第 69 章 SQL

本章介绍关系数据库背后的数学概念。这不是必读内容，所以如果你觉得陷入困境，
或者想直接看一些简单的例子，尽管跳到下一章，等有了更多时间和耐心时再回来。
这些内容本来就应该是有趣的！

这些材料最初是 Stefan Simkovics 的硕士论文（Simkovics, 1998）的一部分。

SQL 已成为最流行的关系查询语言。名称 "SQL" 是 Structured Query Language（结构化查询语言）的缩写。1974 年，Donald Chamberlin 等人在 IBM 研究院定义了语言 SEQUEL（Structured English Query Language，结构化英语查询语言）。该语言于 1974-75 年首次在一个名为 SEQUEL-XRM 的 IBM 原型中实现。1976-77 年定义了修订版 SEQUEL，称为 SEQUEL/2，随后名称改为 SQL。

1977 年，IBM 开发了一个名为 System R 的新原型。System R 实现了 SEQUEL/2（即现在的 SQL）的一个很大的子集，并且在项目期间对 SQL 做了大量修改。System R 被安装在许多用户站点，既有 IBM 内部站点，也有一些选定的客户站点。得益于 System R 在这些用户站点的成功和认可，IBM 开始基于 System R 技术开发实现 SQL 语言的商业产品。

在随后的几年里，IBM 以及其他许多厂商都发布了 SQL 产品，例如 SQL/DS（IBM）、DB2（IBM）、ORACLE（Oracle Corp.）、DG/SQL（Data General Corp.）和 SYBASE（Sybase Inc.）。

SQL 如今也是一个官方标准。1982 年，美国国家标准协会（ANSI）授权其数据库委员会 X3H2 制定标准关系语言的提案。该提案于 1986 年获得批准，本质上由 SQL 的 IBM 方言构成。1987 年，这一 ANSI 标准又被国际标准化组织（ISO）接受为国际标准。这一最初的标准版本的 SQL 常被非正式地称为 "SQL/86"。1989 年，原始标准得到扩展，这一新标准同样常被非正式地称为 "SQL/89"。同样在 1989 年，还制定了一个相关标准，称为 Database Language Embedded SQL（ESQL，数据库语言嵌入式 SQL）。

ISO 和 ANSI 委员会多年来一直在定义一个大幅扩展的原始标准版本，它被非正式地称为 SQL2 或 SQL/92。该版本于 1992 年底成为批准的标准——"International Standard ISO/IEC 9075:1992, Database Language SQL"。当人们提到"the SQL standard"（SQL 标准）时，通常指的就是 SQL/92。SQL/92 的详细描述见 Date and Darwen, 1997。在撰写本文时，一个被非正式地称为 SQL3 的新标准正在制定中。计划让 SQL 成为图灵完备的语言，即所有可计算的查询（例如递归查询）都将成为可能。这是一项非常复杂的任务，因此新标准的完成不会早于 1999 年。

69.1. 关系数据模型

如前所述，SQL 是一种关系语言。也就是说，它基于 E.F. Codd 于 1970 年首次发表的关系数据模型。我们稍后（在关系数据模型的形式化定义中）给出关系模型的形式化描述，
但首先我们想从更直观的角度看一看它。

关系数据库是这样一种数据库：在它的用户看来，它是一个
表的集合（而且除了表之外没有别的东西）。表由行和列组成，每一行代表一条记录，
每一列代表表中记录的一个属性。供应商与零件数据库展示了一个由三张表组成的数据库示例：

- SUPPLIER 是一张存储供应商编号（SNO）、名称（SNAME）和所在城市（CITY）的表。
- PART 是一张存储零件编号（PNO）、名称（PNAME）和价格（PRICE）的表。
- SELLS 存储哪个供应商（SNO）销售哪个零件（PNO）的信息。在某种意义上，它起着把另外两张表连接起来的作用。

例 69.1. 供应商与零件数据库

```
SUPPLIER:                    SELLS:
 SNO |  SNAME  |  CITY        SNO | PNO
-----+---------+--------     -----+-----
 1   |  Smith  | London        1  |  1
 2   |  Jones  | Paris         1  |  2
 3   |  Adams  | Vienna        2  |  4
 4   |  Blake  | Rome          3  |  1
                               3  |  3
                               4  |  2
PART:                          4  |  3
 PNO |  PNAME  |  PRICE        4  |  4
-----+---------+---------
 1   |  Screw  |   10
 2   |  Nut    |    8
 3   |  Bolt   |   15
 4   |  Cam    |   25
```

表 PART 和 SUPPLIER 可以看作实体，而 SELLS 可以看作特定零件与特定供应商之间的联系。

正如我们稍后将看到的，SQL 就是在刚才定义的这类表上操作的，但在此之前，我们先学习关系模型的理论。

69.2. 关系数据模型的形式化定义

关系模型背后的数学概念是集合论的关系，即一组域的笛卡尔积的子集。
正是这个集合论意义上的关系赋予了该模型名字（不要把它与实体-联系模型中的联系相混淆）。
形式上，域只是一个值的集合。例如整数集合就是一个域。长度为 20 的字符串集合和实数集合也是域的例子。

域 D_1、D_2、... D_k 的笛卡尔积记作 $D_1 \times D_2 \times ... \times D_k$，它是满足 $v_1 \in D_1$、$v_2 \in D_2$、... $v_k \in D_k$ 的所有 k-元组 v_1、v_2、... v_k 的集合。

例如，当我们有 k=2, D_1={0,1} 和 D_2={a,b,c} 时，$D_1 \times D_2$ 是 {(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)}。

关系是一个或多个域的笛卡尔积的任意子集：$R \subseteq D_1 \times D_2 \times ... \times D_k$。

例如 {(0,a),(0,b),(1,a)} 是一个关系；它实际上是上文提到的 $D_1 \times D_2$ 笛卡尔积的一个子集。

关系的成员称为元组。某个笛卡尔积 $D_1 \times D_2 \times ... \times D_k$ 上的每个关系被称为元数为 k，因而它是 k 元组的一个集合。

关系可以被看作一张表（我们前面已经这样做了，回忆一下供应商与零件数据库），
其中每个元组由一行表示，而每一列对应元组的一个分量。给列命名（称为属性）就引出了
关系模式的定义。

关系模式R 是属性的一个有限集 $A_1, A_2, ... A_k$。对每个属性 A_i（1 <= i <= k）都有一个域 D_i，属性的值即取自该域。我们常把关系模式写成 R(A_1, A_2, ... A_k)。

> **注意**
>
> 关系模式只是一种模板，而关系是关系模式的一个实例。
> 关系由元组组成（因此可以看作一张表）；关系模式则不然。

69.2.1. 域与数据类型

上一节我们经常谈到域。回想一下，形式上域只是一个值的集合（例如整数集或实数集）。
在数据库系统的语境中，我们常常谈论数据类型而不是域。定义一张表时，
我们必须决定要包含哪些属性。此外还必须决定属性值将要存储哪种数据。例如，表 `SUPPLIER` 中 `SNAME` 的值将是字符串，而 `SNO` 将存储整数。我们通过为每个属性指定一个数据类型来定义这一点。`SNAME` 的类型将是 `VARCHAR(20)`（这是长度 <= 20 的字符串的 SQL 类型），`SNO` 的类型将是 `INTEGER`。指定数据类型的同时，我们也就为属性选定了一个域。`SNAME` 的域是所有长度 <= 20 的字符串的集合，`SNO` 的域是所有整数的集合。

69.3. 关系数据模型中的操作

在上一节（关系数据模型的形式化定义）中，我们定义了关系模型的数学概念。
现在我们知道如何用关系数据模型存储数据了，
但还不知道对这些表做些什么才能从数据库中检索内容。
例如有人可能询问销售零件'Screw'的所有供应商的名称。为此，
人们定义了两种相当不同的用于表达关系上操作的记法：

- 关系代数，一种代数记法，查询通过将专门的操作符应用于关系来表达。
- 关系演算，一种逻辑记法，查询通过表述答案中的元组必须满足的某些逻辑限制来表达。

69.3.1. 关系代数

关系代数由 E. F. Codd 于 1972 年提出。它由一组关系上的操作组成：

- SELECT（σ）：从关系中提取满足给定限制的元组。设 *R* 是一张包含属性 *A* 的表。$\sigma_{A=a}(R) = \{t \in R \mid t(A) = a\}$ 其中 `t` 表示 *R* 的一个元组，而 `t(A)` 表示元组 `t` 的属性 *A* 的值。
- PROJECT（π）：从关系中提取指定的属性（列）。设 `R` 是一个包含属性 `X` 的关系。$\pi_X(R) = \{t(X) \mid t \in R\}$，其中 `t(X)` 表示元组 `t` 的属性 `X` 的值。
- PRODUCT（×）：构建两个关系的笛卡尔积。设 `R` 是元数为 k_1 的表，`S` 是元数为 k_2 的表。`R × S` 是所有这样的 $k_1 + k_2$ 元组的集合：其前 k_1 个分量构成 `R` 中的一个元组，后 k_2 个分量构成 `S` 中的一个元组。
- UNION（∪）：构建两个表的集合论并集。给定表 `R` 和 `S`（两者的元数必须相同），并集 `R ∪ S` 是属于 `R` 或 `S` 或两者的元组的集合。
- INTERSECT（∩）：构建两个表的集合论交集。给定表 `R` 和 `S`，`R ∩ S` 是既属于 `R` 又属于 `S` 的元组的集合。我们同样要求 `R` 和 `S` 元数相同。
- DIFFERENCE（– 或 ⧵）：构建两个表的集合差。设 `R` 和 `S` 仍是两张元数相同的表。`R - S` 是属于 `R` 但不属于 `S` 的元组的集合。

- JOIN（∏）：通过公共属性连接两张表。设 R 是具有属性 A、B 和 C 的表，S 是具有属性 C、D 和 E 的表。两个关系有一个公共属性，即属性 C。R ∏ S = $\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$. 我们在这里做了什么？首先计算笛卡尔积 R × S。然后选出公共属性 C 的值相等的那些元组（$\sigma_{R.C = S.C}$）。现在我们得到一张包含两次属性 C 的表，再通过投影去掉重复的列来纠正这一点。

例 69.2. 一个内连接

让我们看一看执行连接所需的各个步骤所产生的表。给定下面两张表：

```
R:                  S:
 A | B | C           C | D | E
---+---+---         ---+---+---
 1 | 2 | 3           3 | a | b
 4 | 5 | 6           6 | c | d
 7 | 8 | 9
```

首先计算笛卡尔积 R × S，得到：

```
R x S:
 A | B | R.C | S.C | D | E
---+---+-----+-----+---+---
 1 | 2 |  3  |  3  | a | b
 1 | 2 |  3  |  6  | c | d
 4 | 5 |  6  |  3  | a | b
 4 | 5 |  6  |  6  | c | d
 7 | 8 |  9  |  3  | a | b
 7 | 8 |  9  |  6  | c | d
```

经过选择 $\sigma_{R.C=S.C}(R \times S)$ 后，得到：

```
 A | B | R.C | S.C | D | E
---+---+-----+-----+---+---
 1 | 2 |  3  |  3  | a | b
 4 | 5 |  6  |  6  | c | d
```

为去掉重复的列 S.C，我们用下面的操作把它投影出去：$\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$ 并得到：

```
 A | B | C | D | E
---+---+---+---+---
 1 | 2 | 3 | a | b
 4 | 5 | 6 | c | d
```

- DIVIDE（÷）：设 R 是具有属性 A、B、C、D 的表，S 是具有属性 C 和 D 的表。除法定义为：

$$R \div S = \{t \mid \forall t_s \in S \ \exists t_r \in R$$

使得 $t_r(A,B)=t \wedge t_r(C,D)=t_s\}$ 其中 $t_r(x,y)$ 表示表 R 中仅由分量 x 和 y 组成的一个元组。注意元组 t 只由关系 R 的分量 A 和 B 组成。

给定下面的表

```
R:                    S:
 A | B | C | D         C | D
---+---+---+---       ---+---
 a | b | c | d         c | d
 a | b | e | f         e | f
 b | c | e | f
 e | d | c | d
 e | d | e | f
 a | b | d | e
```

R ÷ S 的结果推导为

```
 A | B
---+---
 a | b
 e | d
```

关于关系代数更详细的描述和定义，参见 [Ullman, 1988] 或 [Date, 1994]。

例 69.3. 一个使用关系代数的查询

回想一下，我们阐述所有这些关系操作符，是为了能够从数据库中检索数据。
让我们回到上一节（关系数据模型中的操作）中的例子：有人想知道销售零件 `Screw` 的所有供应商的名称。使用关系代数，这个问题可以用下面的操作来回答：

$$\pi_{SUPPLIER.SNAME}(\sigma_{PART.PNAME='Screw'}(SUPPLIER \prod SELLS \prod PART))$$

我们把这样的操作称为查询。如果对示例表（供应商与零件数据库）求值上述查询，
将得到以下结果：

```
 SNAME
-------
 Smith
 Adams
```

69.3.2. 关系演算

关系演算基于一阶逻辑。关系演算有两种变体：

- 域关系演算（DRC），其中变量代表元组的分量（属性）。
- 元组关系演算（TRC），其中变量代表元组。

我们只想讨论元组关系演算，因为它是大多数关系语言的基石。关于 DRC（以及 TRC）的详细讨论，参见 Date, 1994 或 Ullman, 1988。

69.3.3. 元组关系演算

TRC 中使用的查询具有如下形式：

```
x(A) | F(x)
```

其中 x 是元组变量，A 是一个属性集合，F 是一个公式。结果关系由满足 F(t) 的所有元组 t(A) 组成。

如果我们想用 TRC 回答例子一个使用关系代数的查询中的问题，可以表述如下查询：

```
{x(SNAME) | x ∈ SUPPLIER ∧
    ∃ y ∈ SELLS ∃ z ∈ PART (y(SNO)=x(SNO) ∧
    z(PNO)=y(PNO) ∧
    z(PNAME)='Screw')}
```

对供应商与零件数据库中的表求值该查询，得到的结果与一个使用关系代数的查询中的相同。

69.3.4. 关系代数与关系演算

关系代数和关系演算具有相同的表达能力；
即所有能用关系代数表达的查询也都能用关系演算表达，反之亦然。这一点最早由 E. F. Codd 于 1972 年证明。该证明基于一个算法（"Codd 归约算法"），通过它可以把关系演算的任意表达式归约为语义等价的关系代数表达式。更详细的讨论参见 Date, 1994 和 Ullman, 1988。

有时人们说，基于关系演算的语言比基于关系代数的语言"层次更高"或"更具声明性"，
因为代数（部分地）指定了操作的顺序，
而演算把确定最有效求值顺序的工作留给了编译器或解释器。

69.4. SQL 语言

与大多数现代关系语言一样，SQL 基于元组关系演算。因此，每个能用元组关系演算（或者等价地，关系代数）表述的查询也都能用 SQL 表述。不过，SQL 也有一些超出关系代数或演算范围的能力。下面列出 SQL 提供的一些不属于关系代数或演算的附加特性：

- 用于插入、删除或修改数据的命令。
- 算术能力：在 SQL 中可以使用算术运算和比较，例如：

  ```
  A < B + 3.
  ```

  注意 + 或其他算术操作符既不出现在关系代数中，也不出现在关系演算中。
- 赋值和打印命令：可以打印由查询构造出的关系，也可以把计算得到的关系赋给一个关系名。
- 聚合函数：可以对关系的列应用诸如平均值、求和、最大值等操作，以获得单一的量。

69.4.1. Select

SQL 中最常用的命令是用于检索数据的 SELECT 语句。其语法为：

```
SELECT [ALL|DISTINCT]
    { * | expr_1 [AS c_alias_1] [, ...
     [, expr_k [AS c_alias_k]]]}
    FROM table_name_1 [t_alias_1]
     [, ... [, table_name_n [t_alias_n]]]
    [WHERE condition]
    [GROUP BY name_of_attr_i
     [,... [, name_of_attr_j]] [HAVING condition]]
```

```
    [{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
    [ORDER BY name_of_attr_i [ASC|DESC]
     [, ... [, name_of_attr_j [ASC|DESC]]]];
```

下面我们将用各种示例来说明 SELECT 语句的复杂语法。示例所用的表定义于供应商与零件数据库。

69.4.1.1. 简单 Select

下面是一些使用 SELECT 语句的简单示例：

例 69.4. 带限定条件的简单查询

要检索表 PART 中属性 PRICE 大于 10 的所有元组，我们写出如下查询：

```
SELECT * FROM PART
    WHERE PRICE > 10;
```

并得到表：

```
 PNO |   PNAME   |  PRICE
-----+---------+--------
  3  |  Bolt    |   15
  4  |  Cam     |   25
```

在 SELECT 语句中使用 "*" 将交付该表的所有属性。如果只想检索表 PART 的属性 PNAME 和 PRICE，我们使用如下语句：

```
SELECT PNAME, PRICE
    FROM PART
    WHERE PRICE > 10;
```

此时结果为：

```
 PNAME   |  PRICE
--------+--------
 Bolt    |   15
 Cam     |   25
```

注意，SQL 的 SELECT 对应于关系代数中的"投影"而不是"选择"（更多细节见关系代数）。

WHERE 子句中的限定条件也可以用关键字 OR、AND 和 NOT 进行逻辑连接：

```
SELECT PNAME, PRICE
    FROM PART
    WHERE PNAME = 'Bolt' AND
          (PRICE = 0 OR PRICE < 15);
```

将得到结果：

```
 PNAME   |  PRICE
---------+--------
 Bolt    |   15
```

目标列表和 WHERE 子句中可以使用算术运算。例如，如果我们想知道取一个零件的两件要花多少钱，可以用如下查询：

```
SELECT PNAME, PRICE * 2 AS DOUBLE
    FROM PART
    WHERE PRICE * 2 < 50;
```

我们得到：

```
 PNAME   |  DOUBLE
---------+---------
 Screw   |    20
 Nut     |    16
 Bolt    |    30
```

注意，关键字 AS 之后的单词 DOUBLE 是第二列的新标题。此技术可用于目标列表的每个元素，为结果列指派新标题。这个新标题通常称为别名。别名不能在查询的其余部分中使用。

69.4.1.2. 连接

下面的示例演示了SQL中如何实现连接。

要通过公共属性连接 SUPPLIER、PART 和 SELLS 三张表，我们表述如下语句：

```
SELECT S.SNAME, P.PNAME
    FROM SUPPLIER S, PART P, SELLS SE
    WHERE S.SNO = SE.SNO AND
          P.PNO = SE.PNO;
```

并得到以下表作为结果：

```
 SNAME | PNAME
-------+-------
 Smith | Screw
 Smith | Nut
 Jones | Cam
 Adams | Screw
 Adams | Bolt
 Blake | Nut
 Blake | Bolt
 Blake | Cam
```

在 FROM 子句中，我们为每个关系引入了一个别名，因为这些关系之间存在同名属性（SNO 和 PNO）。现在只需在属性名前加上别名和一个点号，就可以区分这些同名属性。连接的计算方式与一个内连接中所示的相同。首先导出笛卡尔积 SUPPLIER × PART × SELLS 然后只选出满足 WHERE 子句所给条件的元组（即同名属性必须相等）。最后把除 S.SNAME 和 P.PNAME 之外的所有列都投影出去。

69.4.1.3. 聚合函数

SQL 提供聚合运算符（例如 AVG、COUNT、SUM、MIN、MAX），它们接受属性名作为参数。聚合运算符的值是在整张表的指定属性（列）的所有值上计算的。如果查询中指定了组，
则只在该组的值上计算（见下一节）。

例 69.5. 聚合

如果我们想知道表 PART 中所有零件的平均价格，使用以下查询：

```
SELECT AVG(PRICE) AS AVG_PRICE
    FROM PART;
```

结果是：

```
 AVG_PRICE
-----------
   14.5
```

如果我们想知道表 PART 中存储了多少个零件，我们使用如下语句：

```
SELECT COUNT(PNO)
    FROM PART;
```

并得到：

```
 COUNT
-------
   4
```

69.4.1.4. 按组聚合

SQL 允许把表的元组划分成组。然后就可以把上面描述的聚合运算符应用到这些组上（即聚合运算符的值不再是对指定列的所有值计算，而是对一个组的所有值计算。这样，
聚合运算符会对每个组分别求值。）

把元组划分成组的工作通过关键字 `GROUP BY` 后跟定义组的属性列表来完成。如果有 `GROUP BY` A_1, …, A_k，我们就把关系划分为若干组，使得两个元组处于同一组当且仅当它们在所有属性 A_1, …, A_k 上都一致。

例 69.6. 聚合

如果我们想知道每个供应商销售多少个零件，可以表述如下查询：

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
    FROM SUPPLIER S, SELLS SE
    WHERE S.SNO = SE.SNO
    GROUP BY S.SNO, S.SNAME;
```

并得到：

```
 SNO | SNAME | COUNT
```

```
-----+-------+-------
  1  | Smith |   2
  2  | Jones |   1
  3  | Adams |   2
  4  | Blake |   3
```

现在来看看这里发生了什么。首先导出表 SUPPLIER 和 SELLS 的连接：

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
   1   |  Smith  |   2
   2   |  Jones  |   4
   3   |  Adams  |   1
   3   |  Adams  |   3
   4   |  Blake  |   2
   4   |  Blake  |   3
   4   |  Blake  |   4
```

接下来，把在 S.SNO 和 S.SNAME 两个属性上都一致的元组放在一起，将元组划分为组：

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
                 |   2
-------------------------
   2   |  Jones  |   4
-------------------------
   3   |  Adams  |   1
                 |   3
-------------------------
   4   |  Blake  |   2
                 |   3
                 |   4
```

在本例中我们得到了四个组，现在可以对每个组应用聚合运算符 COUNT，从而得到上面给出的查询的最终结果。

注意，要让使用 GROUP BY 和聚合运算符的查询结果有意义，分组所用的属性也必须出现在目标列表中。其他未出现在 GROUP BY 子句中的属性只能通过聚合函数来选择。另一方面，不能对出现在 GROUP BY 子句中的属性使用聚合函数。

69.4.1.5. Having

HAVING 子句的作用与 WHERE 子句很像，用于只考虑满足 HAVING 子句中限定条件的那些组。HAVING 子句中允许的表达式必须涉及聚合函数。每个只使用普通属性的表达式都属于 WHERE 子句。另一方面，每个涉及聚合函数的表达式都必须放到 HAVING 子句中。

例 69.7. Having

如果我们只想要销售多于一个零件的供应商，我们使用如下查询：

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
```

```
    FROM SUPPLIER S, SELLS SE
    WHERE S.SNO = SE.SNO
    GROUP BY S.SNO, S.SNAME
    HAVING COUNT(SE.PNO) > 1;
```

并得到：

```
 SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  3  | Adams |   2
  4  | Blake |   3
```

69.4.1.6. 子查询

在 WHERE 和 HAVING 子句中，凡是期望一个值的地方都允许使用子查询（子 SELECT）。此时该值必须先通过求值子查询得出。子查询的使用扩展了 SQL 的表达能力。

例 69.8. 子选择

如果我们想知道所有比名为 'Screw' 的零件价格更高的零件，我们使用如下查询：

```
SELECT *
    FROM PART
    WHERE PRICE > (SELECT PRICE FROM PART
                   WHERE PNAME='Screw');
```

结果是：

```
 PNO |  PNAME  |  PRICE
-----+---------+--------
  3  |  Bolt   |   15
  4  |  Cam    |   25
```

观察上面的查询可以看到关键字 SELECT 出现了两次。第一次在查询的开头--我们称之为外层 SELECT--另一次在 WHERE 子句中，它开始一个嵌套查询--我们称之为内层 SELECT。对外层 SELECT 的每个元组，都必须求值内层 SELECT。每次求值后我们就知道了名为 'Screw' 的元组的价格，从而可以检查当前元组的价格是否更大。

如果我们想知道没有销售任何零件的供应商（例如，以便能从数据库中删除这些供应商），使用：

```
SELECT *
    FROM SUPPLIER S
    WHERE NOT EXISTS
        (SELECT * FROM SELLS SE
         WHERE SE.SNO = S.SNO);
```

在本例中结果将为空，因为每个供应商都至少销售一个零件。注意我们在内层 SELECT 的 WHERE 子句中使用了来自外层 SELECT 的 S.SNO。如上所述，子查询对外层查询的每个元组求值，即 S.SNO 的值总是取自外层 SELECT 的当前元组。

69.4.1.7. Union, Intersect, Except

这些运算计算由两个子查询导出的元组的并集、交集和集合论差集。

例 69.9. Union, Intersect, Except

下面的查询是 UNION 的例子：

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNAME = 'Jones'
    UNION
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNAME = 'Adams';
```

给出结果：

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

下面是 INTERSECT 的一个例子：

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 1
    INTERSECT
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 2;
```

给出结果：

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
```

查询两部分都返回的唯一元组是 $SNO=2$ 的那一个。

最后是 EXCEPT 的例子：

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 1
    EXCEPT
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 3;
```

给出结果：

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

69.4.2. 数据定义

SQL 语言中包含一组用于数据定义的命令。

69.4.2.1. 创建表

数据定义最基本的命令是创建新关系（新表）的命令。CREATE TABLE 命令的语法为：

```
CREATE TABLE table_name
    (name_of_attr_1 type_of_attr_1
     [, name_of_attr_2 type_of_attr_2
     [, ...]]);
```

例 69.10. 创建表

要创建供应商与零件数据库中定义的表，使用以下 SQL 语句：

```
CREATE TABLE SUPPLIER
    (SNO   INTEGER,
     SNAME VARCHAR(20),
     CITY  VARCHAR(20));


CREATE TABLE PART
    (PNO   INTEGER,
     PNAME VARCHAR(20),
     PRICE DECIMAL(4 , 2));


CREATE TABLE SELLS
    (SNO INTEGER,
     PNO INTEGER);
```

69.4.2.2. SQL 中的数据类型

下面是 SQL 支持的一些数据类型的列表：

- INTEGER：有符号全字二进制整数（31 位精度）。
- SMALLINT：有符号半字二进制整数（15 位精度）。
- DECIMAL (p[,q])：有符号压缩十进制数，精度为 p 位数字，并假定其中 q 位在小数点右边。($15 \geq p \geq q \geq 0$). 如果省略 q，则假定为 0。
- FLOAT：有符号双字浮点数。
- CHAR(n)：长度为 n 的定长字符串。

- VARCHAR(*n*)：最大长度为 *n* 的变长字符串。

69.4.2.3. 创建索引

索引用来加快对关系的访问。如果关系 R 在属性 A 上有一个索引，那么检索所有满足 *t*(A) = *a* 的元组 *t* 时，所需时间大致与这类元组 *t* 的数量成正比，而不再与 R 的大小成正比。

要在 SQL 中创建索引，使用 CREATE INDEX 命令。其语法为：

```
CREATE INDEX index_name
    ON table_name ( name_of_attribute );
```

例 69.11. 创建索引

要在关系 SUPPLIER 的属性 SNAME 上创建名为 I 的索引，使用以下语句：

```
CREATE INDEX I ON SUPPLIER (SNAME);
```

创建的索引会被自动维护，即每当有新元组插入关系 SUPPLIER 时，索引 I 都会随之调整。注意，存在索引时用户能察觉到的唯一变化是速度的提高。

69.4.2.4. 创建视图

视图可以被看作一张虚拟表，即一张在数据库中并不物理存在、但在用户看来好像存在的表。
相比之下，当我们谈论基表时，表的每一行在物理存储中的某处确实有一个物理存储的对应物。

视图没有自己的、物理上分离且可区分的存储数据。相反，
系统把视图的定义（即如何访问物理存储的基表以物化该视图的规则）
存储在系统目录的某处（见系统目录）。关于实现视图的不同技术的讨论，参见 SIM98。

在 SQL 中使用 CREATE VIEW 命令定义视图。语法为：

```
CREATE VIEW view_name
    AS select_stmt
```

其中 *select_stmt* 是一个如Select中所定义的有效的 select 语句。注意，创建视图时并不执行 *select_stmt*，它只是被存储在系统目录中，每当对视图进行查询时才被执行。

给定下面的视图定义（我们再次使用供应商与零件数据库中的表）：

```
CREATE VIEW London_Suppliers
    AS SELECT S.SNAME, P.PNAME
        FROM SUPPLIER S, PART P, SELLS SE
        WHERE S.SNO = SE.SNO AND
              P.PNO = SE.PNO AND
              S.CITY = 'London';
```

现在我们就可以像使用另一张基表一样使用这个虚拟关系 London_Suppliers：

```
SELECT * FROM London_Suppliers
    WHERE P.PNAME = 'Screw';
```

它将返回以下表：

```
 SNAME | PNAME
-------+-------
 Smith | Screw
```

为了计算这个结果，数据库系统必须先隐蔽地访问基表 SUPPLIER、SELLS 和 PART。它通过对这些基表执行视图定义中给出的查询来完成这一步。之后，再应用（针对视图的查询中给出的）附加限定条件，即可得到结果表。

69.4.2.5. Drop Table、Drop Index、Drop View

要销毁一个表（包括该表中存储的所有元组），使用 DROP TABLE 命令：

```
DROP TABLE table_name;
```

要销毁 SUPPLIER 表，使用以下语句：

```
DROP TABLE SUPPLIER;
```

要销毁一个索引，使用 DROP INDEX 命令：

```
DROP INDEX index_name;
```

最后，要销毁一个给定的视图，使用命令 DROP VIEW：

```
DROP VIEW view_name;
```

69.4.3. 数据操纵

69.4.3.1. Insert Into

一旦创建了表（见创建表），就可以用 `INSERT INTO` 命令向其中填入元组。语法为：

```
INSERT INTO table_name (name_of_attr_1
    [, name_of_attr_2 [,...]])
    VALUES (val_attr_1 [, val_attr_2 [, ...]]);
```

要向关系 SUPPLIER（来自供应商与零件数据库）插入第一个元组，使用以下语句：

```
INSERT INTO SUPPLIER (SNO, SNAME, CITY)
    VALUES (1, 'Smith', 'London');
```

要向关系 SELLS 插入第一个元组，使用：

```
INSERT INTO SELLS (SNO, PNO)
    VALUES (1, 1);
```

69.4.3.2. Update

要更改关系中元组的一个或多个属性值，使用 UPDATE 命令。其语法为：

```
UPDATE table_name
    SET name_of_attr_1 = value_1
        [, ... [, name_of_attr_k = value_k]]
    WHERE condition;
```

要修改关系 PART 中零件'Screw'的属性 PRICE 的值，使用：

```
UPDATE PART
    SET PRICE = 15
    WHERE PNAME = 'Screw';
```

名为'Screw'的元组的属性 PRICE 的新值现在是 15。

69.4.3.3. Delete

要从特定的表中删除元组，使用 DELETE FROM 命令。语法为：

```
DELETE FROM table_name
    WHERE condition;
```

要删除表 SUPPLIER 中名为'Smith'的供应商，使用以下语句：

```
DELETE FROM SUPPLIER
    WHERE SNAME = 'Smith';
```

69.4.4. 系统目录

在每个 SQL 数据库系统中，都使用系统目录来记录数据库中定义了哪些表、视图、索引等。这些系统目录可以像普通关系一样被查询。例如，有一个用于视图定义的目录，
该目录存储视图定义中的查询。每当对视图进行查询时，系统首先从目录中取出
视图定义查询并物化该视图，然后再继续处理用户的查询（更详细的描述见 Simkovics, 1998）。关于系统目录的更多信息，参见 Date, 1994。

69.4.5. 嵌入式 SQL

本节将概述如何把 SQL 嵌入到宿主语言（例如 C）中。我们想从宿主语言中使用 SQL 有两个主要原因：

- 有些查询无法用纯 SQL 表述（即递归查询）。要能够执行这类查询，我们需要一种表达能力比 SQL 更强的宿主语言。
- 我们只是想从用宿主语言编写的应用程序访问数据库（例如，
一个带图形用户界面的订票系统用 C 编写，而关于还剩哪些票的信息存储在可以用嵌入式 SQL 访问的数据库中）。

在宿主语言中使用嵌入式 SQL 的程序由宿主语言的语句和嵌入式 SQL（ESQL）语句组成。每条 ESQL 语句都以关键字 `EXEC SQL` 开头。ESQL 语句由预编译器转换成宿主语言的语句（预编译器通常会插入对库例程的调用，由这些例程执行各种 SQL 命令）。

纵观Select中的示例可以意识到，查询的结果常常是一个元组的集合。
而大多数宿主语言并非为操作集合而设计，因此我们需要一种机制来访问 SELECT 语句返回的元组集合中的每一个元组。这一机制可以通过声明一个游标来提供。之后我们就可以用 FETCH 命令检索一个元组并把游标移到下一个元组。

关于嵌入式 SQL 的详细讨论，参见 Date and Darwen, 1997、Date, 1994 或 Ullman, 1988。

第 70 章 体系结构

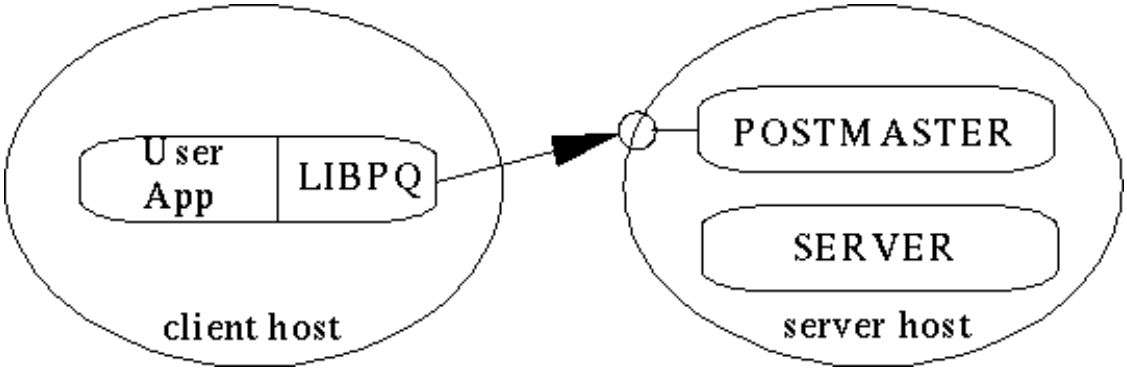
70.1. Postgres 体系结构概念

在我们开始之前，你应当理解基本的 Postgres 系统体系结构。理解 Postgres 的各个部分如何交互将使下一章更容易理解。用数据库术语来说，Postgres 使用一种简单的 "每用户一进程"客户端/服务器模型。一个 Postgres 会话由下列相互协作的 Unix 进程（程序）组成：

- 一个监管守护进程（postmaster），
- 用户的前端应用（例如 psql 程序），以及
- 一个或多个后端数据库服务器（postgres 进程本身）。

一个 postmaster 管理单个主机上一个给定的数据库集合。这样的数据库集合称为一个安装或站点。希望访问安装内某个给定数据库的前端应用对库进行调用。库通过网络把用户请求发送给 postmaster（图 70.1），后者转而启动一个新的后端服务器进程

图 70.1. 如何建立一个连接



并把前端进程连接到新的服务器。从那时起，前端进程和后端服务器进行通信而无需 postmaster 干预。因此，postmaster 总是在运行，等待请求，而前端和后端进程则来来去去。

`libpq` 库允许单个前端建立到多个后端进程的连接。但是，前端应用仍然是单线程进程。目前 `libpq` 不支持多线程的前端/后端连接。这种体系结构的一个含义是 postmaster 和后端总是在同一台机器（数据库服务器）上运行，而前端应用可以运行在任何地方。你应当牢记这一点，因为在客户机上可以访问的文件在数据库服务器机器上可能无法访问（或者只能用不同的文件名访问）。

你还应当知道，postmaster 和 postgres 服务器以 Postgres "超级用户"的用户 ID 运行。注意 Postgres 超级用户并不非得是特殊用户（例如名为 "postgres" 的用户）。而且，Postgres 超级用户绝对不应当是 Unix 超级用户（"root"）！任何情况下，与某个数据库相关的所有文件都应当属于这个 Postgres 超级用户。

第 71 章 从头开始

新用户如何开始使用 Postgres。

使用 Postgres 所需的一些步骤可以由任何 Postgres 用户执行，而另一些必须由站点数据库管理员完成。站点管理员是安装软件、创建数据库目录并启动 postmaster 进程的人。这个人不必是 Unix 超级用户（"root"）或计算机系统管理员；任何人都可以在没有任何特殊账户或权限的情况下安装和使用 Postgres。

如果你要自己安装 Postgres，请参阅管理员指南中的安装说明，安装完成后再回到本指南。

在本手册中，任何以字符 "`%`" 开头的示例都是应当在 Unix shell 提示符下键入的命令。以字符 "`*`" 开头的示例是 Postgres 查询语言 Postgres SQL 中的命令。

71.1. 设置你的环境

本节讨论如何设置你自己的环境，以便使用前端应用。我们假定 Postgres 已经成功安装并启动；关于如何安装 Postgres，请参阅管理员指南和安装说明。

Postgres 是一个客户端/服务器应用。作为用户，你只需要访问安装中的客户端部分（客户端应用的一个例子是交互式监控器 psql）。为简单起见，我们假定 Postgres 安装在目录 `/usr/local/pgsql` 中。因此，凡是看到目录 `/usr/local/pgsql` 之处，你都应该把它替换为 Postgres 实际安装的目录名。所有 Postgres 命令都安装在目录 `/usr/local/pgsql/bin` 中。因此，你应当把这个目录加入你的 shell 命令路径。如果你使用 Berkeley C shell 的变体（如 csh 或 tcsh），可以在你家目录的 `.login` 文件中加入

```
% set path = ( /usr/local/pgsql/bin path )
```

。如果你使用 Bourne shell 的变体（如 sh、ksh 或 bash），则应当把

```
% PATH=/usr/local/pgsql/bin:$PATH
% export PATH
```

加入你家目录的 .profile 文件。从现在起，我们假定你已把 Postgres 的 bin 目录加入了路径。此外，本文档会频繁提到“设置 shell 变量”或“设置环境变量”。如果你没有完全理解上一段关于修改搜索路径的内容，应当先查阅描述你所用的 shell 的 Unix 手册页，然后再继续。

如果你的站点管理员没有按默认方式完成设置，你可能还需要做一些额外工作。例如，
如果数据库服务器所在的机器是远程机器，你就需要把 PGHOST 环境变量设置为数据库服务器机器的名称。环境变量 PGPORT 也可能需要设置。归根结底就是：如果你试着启动某个应用程序，而它抱怨无法连接到 postmaster，你应当立即咨询站点管理员，以确保你的环境已正确设置。

71.2. 启动交互式监控器（psql）

假定你的站点管理员已正确启动 postmaster 进程并授权你使用数据库，你（作为用户）就可以开始启动应用了。如前所述，你应当把 `/usr/local/pgsql/bin` 加入你的 shell 搜索路径。在大多数情况下，这就是你在准备方面需要做的全部工作。

支持两种不同风格的连接。站点管理员或者选择允许 TCP/IP 网络连接，或者把数据库访问限制为仅本地（同机）套接字连接。如果你在连接数据库时遇到问题，这些选择就变得重要了，
因为你需要确认自己选择的是被允许的连接选项。

如果你从某个 Postgres 命令（如 psql 或 createdb）得到下面的错误消息：

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting
 connections
    at 'UNIX Socket' on port '5432'?
```

或者

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

通常是因为

- postmaster 没有在运行，或者
- 你试图连接到错误的服务器主机。

如果你得到下面的错误消息：

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

这表示站点管理员以错误的用户身份启动了 postmaster。让他以 Postgres 超级用户身份重新启动它。

71.3. 管理数据库

既然 Postgres 已经启动并运行，我们可以创建一些数据库来做实验。这里我们描述管理数据库的基本命令。

大多数 Postgres 应用假定，如果没有指定数据库名，它就与你的计算机账户名相同。

如果你的数据库管理员在设置你的账户时没有给它创建数据库的权限，
那么她应当已经告诉你你的数据库名是什么。如果是这种情况，
你可以跳过关于创建和删除数据库的小节。

71.3.1. 创建数据库

比如说你想创建一个名为 mydb 的数据库。你可以用下面的命令完成：

```
% createdb mydb
```

如果你没有创建数据库所需的权限，你会看到：

```
% createdb mydb
NOTICE:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres 允许你在一个给定站点上创建任意数量的数据库，并且你会自动成为你所创建数据库的数据库管理员。数据库名必须以字母开头，长度限制为 32 个字符。并非每个用户都有成为数据库管理员的授权。如果 Postgres 拒绝为你创建数据库，那么站点管理员需要授予你创建数据库的权限。如果发生这种情况，请咨询你的站点管理员。

71.3.2. 访问数据库

构建好数据库后，你可以通过以下方式访问它：

- 运行 Postgres 终端监控器程序（如 psql），它允许你交互式地输入、编辑并执行 SQL 命令。
- 使用现有的原生前端工具，如 pgaccess 或 ApplixWare（通过 ODBC）来创建和操作数据库。
- 使用带有 Postgres 受支持接口的语言（如 perl 或 tcl）。其中一些语言还有方便而强大的 GUI 工具包，可以帮助你构建自定义应用。上面提到的 pgaccess 就是这样一种用 tk/tcl 编写的应用，可以用作示例。
- 用 C 语言编写使用 LIBPQ 子例程库的程序。这允许你从 C 提交 SQL 命令，并把答案和状态消息返回给你的程序。这个接口在PostgreSQL 程序员指南中进一步讨论。

你可能想启动 psql 来试一下本手册中的示例。为 mydb 数据库启动它只需键入命令：

```
% psql mydb
```

你会看到下面的欢迎消息：

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

这个提示符表示终端监控器正在等待你的输入，你可以把 SQL 查询键入由终端监控器维护的工作区中。psql 程序会响应以反斜杠字符 "\" 开头的转义码。例如，键入下面的命令可以得到各种 Postgres SQL 命令语法的帮助：

```
mydb=> \h
```

把查询输入工作区后，你可以通过键入下面的命令把工作区的内容传递给 Postgres 服务器：

```
mydb=> \g
```

这会告诉服务器处理该查询。如果你用分号终止查询，则不需要 "\g"。psql 会自动处理以分号终止的查询。要从文件（比如 myFile）读取查询而不是交互式输入，键入：

```
mydb=> \i fileName
```

要退出 psql 并返回 Unix，键入

```
mydb=> \q
```

这样 psql 就会退出并返回到命令 shell。（要了解更多转义码，在监控器提示符下键入 `\h`。）在 SQL 查询中可以自由使用空白（即空格、制表符和换行符）。单行注释用 "`--`" 表示。从双划线到行尾的所有内容都会被忽略。多行注释和行内注释用 "`/* ... */`" 表示。

71.3.3. 删除数据库

如果你是数据库 mydb 的数据库管理员，你可以用下面的 Unix 命令删除它：

```
% dropdb mydb
```

这个操作会从物理上删除与该数据库关联的所有 Unix 文件，而且无法撤销，因此务必三思而后行。

第 72 章 查询语言

Postgres 查询语言是下一代标准草案 SQL3 的一个变种。它对 SQL92 有许多扩展，例如可扩展的类型系统、继承、函数和产生式规则。这些都是从最初的 Postgres 查询语言 PostQuel 继承下来的特性。本节概述如何使用 Postgres SQL 执行简单的操作。本手册只是让你对我们这种风格的 SQL 有个概念，绝不是 SQL 的完整教程。关于 SQL92 已有许多著作，包括 Melton and Simon, 1993 和 Date and Darwen, 1997。你应当知道，某些语言特性是对 ANSI 标准的扩展。

72.1. 交互式监视器

在后面的例子中，我们假设你已按上一小节的说明创建了 mydb 数据库并启动了 psql。本手册中的例子也可以在 `/usr/local/pgsql/src/tutorial/` 中找到。如何使用它们请参考该目录下的 `README` 文件。要开始教程，请执行以下操作：

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: postgres

mydb=> \i basics.sql
```

`\i` 命令从指定的文件中读入查询。`-s` 选项让你进入单步模式，它会在把查询发送给后端之前暂停。本节使用的查询在文件 `basics.sql` 中。

psql 有多种用于显示系统信息的 `\d` 命令。更多细节请查阅这些命令；要查看列表，请在 psql 提示符下输入 `\?`。

72.2. 概念

Postgres 中的基本概念是类，它是一组有名字的对象实例。每个实例都有相同的命名属性集合，每个属性都有特定的类型。此外，每个实例都有一个永久的对象标识符（OID），
它在整个安装范围内唯一。由于 SQL 语法说的是表，我们将把表和类作为同义词混用。同样，SQL 的行是实例，SQL 的列就是属性。如前所述，类被组织成数据库，而由单个 postmaster 进程管理的一组数据库构成一个安装或站点。

72.3. 创建新类

你可以通过指定类名以及所有属性的名称和类型来创建一个新类：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
```

```
);
```

注意关键字和标识符都不区分大小写；标识符可以用双引号括起来以保留大小写，这是 SQL92 所允许的。Postgres 的 SQL 支持常见的 SQL 类型 int、float、real、smallint、char(N)，varchar(N)、date、time 和 timestamp，还有其他一些通用类型以及一套丰富的几何类型。我们后面会看到，Postgres 可以用任意数量的用户自定义数据类型来定制。因此，类型名不是语法上的关键字，除非为了支持 SQL92 标准中的特殊情形所必需。到目前为止，Postgres 的 CREATE 命令看上去与传统关系系统中建表的命令完全一样。但我们马上就会看到，类还有一些性质是对关系模型的扩展。

72.4. 用实例填充类

INSERT 语句用于向类中填充实例：

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

你也可以用 COPY 从平面（ASCII）文件装载大量数据。这通常更快，因为数据是作为单个原子事务直接读写目标表的。例如：

```
COPY weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

其中源文件的路径名必须是后端服务器机器（而不是客户端）能访问到的，
因为文件是由后端服务器直接读取的。

72.5. 查询类

weather 类可以用普通的关系选择和投影查询来查询。这由 SQL 的 SELECT 语句完成。该语句分为目标列表（列出要返回的属性的部分）和限定条件（指定任何限制的部分）两部分。例如，要检索 weather 的所有行，输入：

```
SELECT * FROM weather;
```

输出应该是：

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

你可以在目标列表中指定任意表达式。例如：

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

查询的限定条件中允许使用任意布尔运算符（AND、OR 和 NOT）。例如，

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

结果为：

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

最后提一句，你可以指定选择的结果按排序顺序返回，或者去掉重复的实例。

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

72.6. 重定向 SELECT 查询

任何 SELECT 查询都可以重定向到一个新类：

```
SELECT * INTO TABLE temp FROM weather;
```

这构成一条隐式的 CREATE 命令，用 SELECT INTO 命令目标列表中指定的属性名和类型创建新类 temp。当然，之后我们可以像对其他类一样对结果类执行任何操作。

72.7. 类之间的连接

到目前为止，我们的查询一次只访问一个类。查询可以同时访问多个类，
也可以用让类的多个实例同时被处理的方式访问同一个类。
一次访问同一个或多个类的多个实例的查询称为连接查询。举个例子，
假设我们希望找出所有位于其他记录温度范围之内的记录。实际上，我们需要把每个 EMP 实例的 temp_lo 和 temp_hi 属性与所有其他 EMP 实例的 temp_lo 和 temp_hi 属性进行比较。

> **注意**
>
> 这只是一个概念模型。实际的连接可能以更高效的方式执行，但这对用户是不可见的。

我们可以用下面的查询做到：

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

+--------------+-----+------+---------------+-----+------+
```

```
|city          | low | high | city          | low | high |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
```

注意

这种连接的语义是：限定条件是针对查询中所指各类的笛卡尔积定义的一个真值表达式。对于笛卡尔积中使限定条件为真的那些实例，Postgres 计算并返回目标列表中指定的值。Postgres 的 SQL 不对此类表达式中的重复值赋予任何意义。这意味着 Postgres 有时会多次重复计算同一目标列表；当布尔表达式用 "or" 连接时经常出现这种情况。要去掉这样的重复，必须使用 `SELECT DISTINCT` 语句。

在这个例子中，`W1` 和 `W2` 都是类 weather 中某个实例的替身，二者都遍历该类的所有实例。（用大多数数据库系统的术语说，`W1` 和 `W2` 被称为范围变量。）查询可以包含任意数量的类名和替身。

72.8. 更新

你可以用 `UPDATE` 命令更新现有的实例。假设你发现从 11 月 28 日起所有温度读数都偏了 2 度，你可以这样更新数据：

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

72.9. 删除

删除是用 `DELETE` 命令执行的：

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

所有属于 Hayward 的天气记录都被删除了。对如下形式的查询应当保持警惕

```
DELETE FROM classname;
```

没有限定条件时，`DELETE` 会直接删除给定类中的所有实例，使其变空。系统在此之前不会请求确认。

72.10. 使用聚合函数

与大多数其他关系数据库产品一样，PostgreSQL 支持聚合函数。聚合函数从多个输入行计算出单个结果。例如，有一些聚合可以计算一组实例的 `count`（计数）、`sum`（求和）、`avg`（平均值）、`max`（最大值）和 `min`（最小值）。

理解聚合与 SQL 的 WHERE 和 HAVING 子句之间的交互很重要。WHERE 与 HAVING 的根本区别在于：WHERE 在分组和聚合计算之前选择输入行（因此它控制哪些行进入聚合计算），而 HAVING 在分组和聚合计算之后选择分组行。因此，WHERE 子句不得包含聚合函数；试图用聚合来决定哪些行作为聚合的输入是没有意义的。另一方面，HAVING 子句总是包含聚合函数。（严格说来，你也可以写不使用聚合的 HAVING 子句，但那是浪费；同样的条件放在 WHERE 阶段会更高效。）

例如，我们可以用下面的语句找出所有记录中最低温度读数的最高值：

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

如果我们想知道这个读数出现在哪个（哪些）城市，我们可能会试试

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

但这行不通，因为聚合 max 不能用在 WHERE 中。不过，正如常见的情况，可以把查询改写成另一种形式来达到想要的结果，这里用的是子选择：

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

这样是可以的，因为子选择是一个独立的计算，它独立于外层选择所做的事情计算自己的聚合。

聚合与 GROUP BY 子句结合使用也非常有用。例如，我们可以用下面的语句得到每个城市观测到的最低温度的最大值：

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

这会为每个城市输出一行。我们可以用 HAVING 过滤这些分好组的行：

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

这只会对那些有零下读数的城市给出相同的结果。最后，如果我们只关心名字以 "P" 开头的城市，可以这样写：

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city like 'P%'
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

注意，我们可以把城市名的限制条件放在 WHERE 中，因为它不需要聚合。这比把该限制加到 HAVING 更高效，因为我们避免了对所有未通过 WHERE 检查的行做分组和聚合计算。

第 73 章 Postgres SQL 的高级特性

在介绍了使用 Postgres SQL 访问数据的基础知识之后，我们现在来讨论 Postgres 区别于传统数据管理器的那些特性。这些特性包括继承、
时间旅行以及非原子数据值（数组值和集合值属性）。本节的示例也可以在教程目录的 `advance.sql` 中找到。（关于如何使用它，请参阅第 72 章。）

73.1. 继承

让我们创建两个类。capitals 类包含也是城市的州首府。自然，capitals 类应当从 cities 继承。

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

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

在这种情况下，capitals 的一个实例会从它的父类 cities 继承全部属性（name、population 和 altitude）。属性 name 的类型是 `text`，这是 Postgres 原生的一种变长 ASCII 字符串类型。属性 population 的类型是 `float`，一种 Postgres 原生的双精度浮点数类型。州首府有一个额外的属性 state，用来表示所在的州。在 Postgres 中，一个类可以从零个或多个其他类继承，而一个查询既可以引用一个类的所有实例，也可以引用一个类及其所有后代的所有实例。

> **注意**
>
> 继承层次结构是一个有向无环图。

例如，下面的查询会找出所有海拔在 500 英尺或更高的城市：

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

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

另一方面，要找出所有海拔超过 500 英尺的城市名称（包括州首府），查询是：

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

它返回：

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

这里，cities 后面的"*"表示该查询应当在 cities 以及继承层次中位于 cities 之下的所有类上运行。我们已经讨论过的许多命令（SELECT、UPDATE 和 DELETE）以及 ALTER 等其他命令都支持这种使用"*"的继承记法。

73.2. 非原子值

关系模型的信条之一是关系的属性是原子的。Postgres 没有这个限制；属性本身可以包含可以从查询语言访问的子值。例如，你可以创建由基本类型构成的数组属性。

73.2.1. 数组

Postgres 允许把一个实例的属性定义为定长或变长的多维数组。可以创建任何基本类型或用户定义类型的数组。为了说明它们的用法，我们首先创建一个带有基本类型数组的类。

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

上面的查询会创建一个名为 SAL_EMP 的类，它带有一个 text 字符串（name）、一个 int4 一维数组（pay_by_quarter，表示雇员按季度的薪水）和一个 text 二维数组（schedule，表示雇员的每周日程）。现在我们做一些 INSERTS；注意，向数组追加内容时，我们把值放在花括号内并用逗号分隔。如果你了解 C，这很像初始化结构的语法。

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

默认情况下，Postgres 对数组使用"从 1 开始"的编号约定——即一个有 n 个元素的数组从 array[1] 开始到 array[n] 结束。现在，我们可以在 SAL_EMP 上运行一些查询。首先，我们演示如何一次访问数组的一个元素。下面的查询检索薪水在第二季度发生变化的雇员的姓名：

```
SELECT name
    FROM SAL_EMP
    WHERE SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

下面的查询检索所有雇员的第三季度薪水：

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

我们还可以访问数组的任意切片或子数组。下面的查询检索 Bill 的日程中一周头两天的第一项。

```
SELECT SAL_EMP.schedule[1:2][1:1]
    FROM SAL_EMP
    WHERE SAL_EMP.name = 'Bill';

+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

73.3. 更多高级特性

Postgres 还有许多特性是这个面向 SQL 新用户的入门教程中没有涉及的。这些特性在用户指南和程序员指南中有更详细的讨论。

部分 VII. 附录

其他相关信息。

目录

附录 UG1. 日期/时间支持

UG1.1. 时区

Postgres 必须拥有用于时区解码的内部表格式信息，因为没有标准的 *nix 系统接口可用于访问通用的跨时区信息。底层操作系统确实被用来为输出提供时区信息。

表 UG1.1. Postgres 识别的时区

时区	与 UTC 的偏移	描述
NZDT	+13:00	新西兰夏令时间
IDLE	+12:00	国际日期变更线以东
NZST	+12:00	新西兰标准时间
NZT	+12:00	新西兰时间
AESST	+11:00	澳大利亚东部夏季标准时间
ACSST	+10:30	中澳夏季标准时间
CADT	+10:30	中澳夏令时间
SADT	+10:30	南澳夏令时间
AEST	+10:00	澳大利亚东部标准时间
EAST	+10:00	澳东标准时间
GST	+10:00	关岛标准时间，苏联 9 区
LIGT	+10:00	澳大利亚墨尔本
ACST	+09:30	中澳标准时间
CAST	+09:30	中澳标准时间
SAT	+9:30	南澳标准时间
AWSST	+9:00	澳大利亚西部夏季标准时间
JST	+9:00	日本标准时间，苏联 8 区
KST	+9:00	韩国标准时间
WDT	+9:00	西澳夏令时间
MT	+8:30	摩鹿加时间
AWST	+8:00	澳大利亚西部标准时间
CCT	+8:00	中国沿海时间
WADT	+8:00	西澳夏令时间
WST	+8:00	西澳标准时间
JT	+7:30	爪哇时间
WAST	+7:00	西澳标准时间
IT	+3:30	伊朗时间
BT	+3:00	巴格达时间
EETDST	+3:00	东欧夏令时间
CETDST	+2:00	中欧夏令时间
EET	+2:00	东欧，苏联 1 区
FWT	+2:00	法国冬令时间
IST	+2:00	以色列标准时间

时区	与 UTC 的偏移	描述
MEST	+2:00	中欧夏令时间
METDST	+2:00	中欧夏令时间
SST	+2:00	瑞典夏令时间
BST	+1:00	英国夏令时间
CET	+1:00	中欧时间
DNT	+1:00	Dansk Normal Tid
DST	+1:00	Dansk Standard Time (?)
FST	+1:00	法国夏令时间
MET	+1:00	中欧时间
MEWT	+1:00	中欧冬令时间
MEZ	+1:00	中欧时区
NOR	+1:00	挪威标准时间
SET	+1:00	塞舌尔时间
SWT	+1:00	瑞典冬令时间
WETDST	+1:00	西欧夏令时间
GMT	0:00	格林尼治标准时间
WET	0:00	西欧
WAT	-1:00	西非时间
NDT	-2:30	纽芬兰夏令时间
ADT	-03:00	大西洋夏令时间
NFT	-3:30	纽芬兰标准时间
NST	-3:30	纽芬兰标准时间
AST	-4:00	大西洋标准时间（加拿大）
EDT	-4:00	美国东部夏令时间
ZP4	-4:00	GMT +4 小时
CDT	-5:00	美国中部夏令时间
EST	-5:00	美国东部标准时间
ZP5	-5:00	GMT +5 小时
CST	-6:00	美国中部标准时间
MDT	-6:00	美国山地夏令时间
ZP6	-6:00	GMT +6 小时
MST	-7:00	美国山地标准时间
PDT	-7:00	美国太平洋夏令时间
PST	-8:00	美国太平洋标准时间
YDT	-8:00	育空夏令时间
HDT	-9:00	夏威夷/阿拉斯加夏令时间
YST	-9:00	育空标准时间
AHST	-10:00	阿拉斯加-夏威夷标准时间
CAT	-10:00	阿拉斯加中部时间
NT	-11:00	诺姆时间
IDLW	-12:00	国际日期变更线以西

UG1.1.1. 澳大利亚时区

澳大利亚时区及其命名变体占了 Postgres 时区查找表中整整四分之一的条目。它们与美国定义的常用时区存在两处命名冲突：`CST` 和 `EST`。

如果设置了编译器选项 USE_AUSTRALIAN_RULES，那么 `CST` 和 `EST` 将按澳大利亚约定解释。

表 UG1.2. Postgres 澳大利亚时区

时区	与 UTC 的偏移	描述
CST	+10:30	澳大利亚中部标准时间
EST	+10:00	澳大利亚东部标准时间

UG1.1.2. 日期/时间输入解释

日期/时间类型都使用一组公共的例程进行解码。

日期/时间输入解释

1. 将输入字符串拆分为词元，并把每个词元归类为字符串、时间、时区或数字。
   a. 如果词元包含冒号（":"），则它是一个时间字符串。
   b. 如果词元包含连字符（"-"）、斜杠（"/"）或点（"."），则它是一个日期字符串，并且可能带有文本形式的月份。
   c. 如果词元是纯数字，那么它要么是一个单个字段，要么是一个 ISO-8601 拼接式日期（例如 "19990113" 表示 1999 年 1 月 13 日）或时间（例如 141516 表示 14:15:16）。
   d. 如果词元以加号（"+"）或减号（"-"）开头，那么它要么是一个时区，要么是一个特殊字段。
2. 如果词元是一个文本字符串，则将其与可能的字符串匹配。
   a. 对该词元做二分查找表匹配，看它是否为特殊字符串（例如 `today`）、星期名（例如 `Thursday`）、月份名（例如 `January`）或噪声词（例如 `on`）。

      设置字段值和字段位掩码。例如，为 `today` 设置年、月、日，并为 `now` 额外设置时、分、秒。
   b. 如果没有找到，则做一次类似的二分查找表匹配，尝试将该词元与时区匹配。
   c. 如果没有找到，则抛出错误。
3. 该词元是一个数字或数字字段。
   a. 如果多于 4 位数字，并且之前还没有读取到其他日期字段，则将其解释为"拼接式日期"（例如 `19990118`）。8 位和 6 位数字被解释为年、月、日，而 7 位和 5 位数字则分别被解释为年、年积日。
   b. 如果词元是三位数字并且已经解码了年份，则解释为年积日。
   c. 如果是四位或更多位数字，则解释为年份。
   d. 如果处于欧洲日期模式，并且还没有读取日字段，并且该值小于或等于 31，则解释为日。

e. 如果还没有读取月份字段，并且该值小于或等于 12，则解释为月。

f. 如果还没有读取日字段，并且该值小于或等于 31，则解释为日。

g. 如果是两位数字或四位及更多位数字，则解释为年份。

h. 否则，抛出错误。

4. 如果指定了 BC，则将年份取反并偏移一后再用于内部存储（格里高利历中没有 0 年，因此从数值上说，1BC 会变成年 0）。

5. 如果没有指定 BC，并且年份字段的长度为两位，则将年份调整为 4 位。如果该字段小于 70，则加上 2000；否则加上 1900。

> **提示**
>
> 格里高利历公元 1-99 年可以使用带前导零的 4 位数字输入（例如，0099 表示公元 99 年）。以前的 Postgres 版本接受三位数字和单个数字的年份，但从 v7.0 开始，规则已被收紧以减少歧义的可能性。

UG1.2. 历法的历史

> **注意**
>
> 由 José Soares[1] 贡献。

儒略日由法国学者 Joseph Justus Scaliger（1540-1609）发明，其名称大概取自 Scaliger 的父亲、意大利学者 Julius Caesar Scaliger（1484-1558）。天文学家用儒略纪为自公元前 4713 年 1 月 1 日以来的每一天分配一个唯一的编号。这就是所谓的儒略日（JD）。JD 0 表示从 UTC 公元前 4713 年 1 月 1 日正午到 UTC 公元前 4713 年 1 月 2 日正午的 24 小时。

"儒略日"（Julian Day）与"儒略积日"（Julian Date）并不相同。儒略历由 Julius Caesar 于公元前 45 年引入。直到 1582 年各国开始改用格里高利历之前，它在西方世界一直被广泛使用。在儒略历中，回归年近似为 365 1/4 天 = 365.25 天。这会在大约 128 天里累计 1 天的误差。不断累积的历法误差促使教皇格里高利十三世按照特伦托会议的指示改革历法。

在格里高利历中，回归年近似为 365 + 97 / 400 天 = 365.2425 天。因此，回归年相对于格里高利历大约每 3300 年偏移一天。

365+97/400 这一近似值是通过每 400 年设置 97 个闰年来实现的，具体规则如下：

能被 4 整除的年份是闰年。
但是，能被 100 整除的年份不是闰年。
但是，能被 400 整除的年份仍是闰年。

因此，1700、1800、1900、2100 和 2200 年不是闰年，而 1600、2000 和 2400 年是闰年。相比之下，在较老的儒略历中只有能被 4 整除的年份是闰年。

[1] jose@sferacarta.com

1582 年 2 月发布的教皇诏书规定，应从 1582 年 10 月删去 10 天，使 10 月 15 日紧接在 10 月 4　日之后。意大利、波兰、葡萄牙和西班牙执行了这一规定。其他天主教国家随后不久也照办，但新教国家不愿改变，而希腊正教国家直到本世纪初才改历。
大不列颠及其领地（包括现在的美国）在 1752 年执行了这一改革。因此 1752 年 9 月 2 日之后紧跟着的是 1752 年 9 月 14 日。这就是 Unix 系统的 cal 会输出下面内容的原因：

```
% cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **注意**
>
> SQL92 指出，“在`datetime literal`的定义中，`datetime value`受格里高利历下日期和时间的自然规则约束”。1752-09-03 到 1752-09-13 之间的日期虽然在某些国家被教皇敕令取消，但它们符合“自然规则”，因而是合法的日期。

世界上不同地区发展出了不同的历法，其中许多都早于格里高利体系。例如，
中国历法的起源可以追溯到公元前 14 世纪。传说黄帝在公元前 2637 年发明了这种历法。中华人民共和国在民用方面使用格里高利历。中国历法则用于确定节日。

附录 DG1. CVS 代码库

Marc Fournier
Tom Lane
Thomas Lockhart

Postgres 源代码使用 CVS 代码管理系统存储和管理。

至少有两种方法——匿名 CVS 和 CVSup——可以把 CVS 代码树从 Postgres 服务器拉取到你的本地机器上。

DG1.1. CVS 树的组织

> **作者**
>
> 由 Marc G. Fournier[1] 写于 1998-11-05。

命令 `cvs checkout` 有一个标志 `-r`，让你检出模块的某个特定版本。有了这个标志，例如在将来任何时候检索组成模块 `tc' 的 1.0 版本的源码都很容易：

```
$ cvs checkout -r REL6_4 tc
```

比如说，如果有人声称那个版本里有一个 bug，而你在当前工作副本中找不到它，这就很有用。

> **提示**
>
> 你也可以用 `-D` 选项检出模块在任一给定日期时的状态。

当你用同一个标记标记多个文件时，
可以把这个标记想成"一条画在文件名与版本号矩阵中穿过的曲线"。假设我们有 5 个文件，版本如下：

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

那么标记 "`TAG`" 将引用 file1-1.2、file2-1.3 等等。

> **注意**
>
> 创建发行分支时，除了在命令中加上 -b 选项之外，其余都是一回事。

[1] mailto:scrappy@hub.org

于是，为了创建 v6.4 发行版，我做了以下操作：

```
$ cd pgsql
$ cvs tag -b REL6_4
```

这会为 RELEASE 树创建标记和分支。

现在，对于拥有 CVS 写权限的人来说就太简单了。首先创建 RELEASE 和 CURRENT 两个子目录，免得把两者弄混。然后执行：

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

这会得到两棵目录树：`RELEASE/pgsql` 和 `CURRENT/pgsql`。从那时起，CVS 会跟踪哪棵目录树对应仓库的哪个分支，并允许对两棵树独立地进行更新。

如果你只在 `CURRENT` 源码树上工作，那么一切照旧，就像我们开始标记发行分支之前那样做即可。

在某个分支上完成初始检出之后

```
$ cvs checkout -r REL6_4
```

你在该目录结构中做的任何操作都局限于该分支。如果你对该目录结构打了一个补丁，
然后在其内部执行

```
cvs commit
```

那么补丁会被应用到该分支上，而且只应用到该分支。

DG1.2. 通过匿名 CVS 获取源码

如果你想定期跟上当前的源码，可以从我们的 CVS 服务器获取它们，然后不时地用 CVS 检索更新。

匿名 CVS

1. 你需要一份本地安装的 CVS（并发版本控制系统），可以从 http://www.cyclic.com/ 或任何 GNU 软件归档站点获取。我们目前推荐 1.10 版（写作时最新的版本）。许多系统默认就装有较新版本的 cvs。

2. 首次登录到 CVS 服务器：

   ```
   $ cvs -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot login
   ```

   系统会提示你输入口令；输入 'postgresql'。这只需要做一次，因为口令会保存在你主目录的 .cvspass 中。

3. 获取 Postgres 源码：

```
cvs -z3 -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot co -
P pgsql
```

这会把 Postgres 源码安装到你当前所在目录的 `pgsql` 子目录中。

> **注意**
>
> 如果你的因特网链路很快，可能不需要 `-z3`（它让 CVS 对传输的数据使用 gzip 压缩）。但在调制解调器速度的链路上，它能带来非常大的收益。

这个初始检出比直接下载一个 `tar.gz` 文件稍慢一些；如果你用的是 28.8K 调制解调器，预计要花 40 分钟左右。CVS 的优势要等到你以后想更新文件集时才会显现出来。

4. 每当你想更新到最新的 CVS 源码时，`cd` 进入 `pgsql` 子目录，然后执行

```
$ cvs -z3 update -d -P
```

这只会获取自你上次更新以来的变更。通常只要几分钟就能完成更新，
即使是在调制解调器速度的链路上。

5. 你可以在主目录中创建一个包含以下内容的 `.cvsrc` 文件来省些输入：

```
cvs -z3
update -d -P
```

这会为所有 cvs 命令提供 `-z3` 选项，并为 cvs update 提供 `-d` 和 `-P` 选项。这样你只需输入

```
$ cvs update
```

即可更新你的文件。

> **小心**
>
> 某些较旧版本的 CVS 有一个 bug，会导致所有检出的文件在你的目录中以全局可写的方式存放。如果发现这种情况，可以执行类似这样的命令
>
> ```
> $ chmod -R go-w pgsql
> ```
>
> 来正确设置权限。这个 bug 在 CVS 1.9.28 版中已经修复。

CVS 还能做许多别的事情，例如获取 Postgres 源码的早期版本而不是最新开发版本。更多信息请查阅 CVS 附带的手册，或参见 http://www.cyclic.com/ 上的在线文档。

DG1.3. 通过 CVSup 获取源码

检索 Postgres 源码树的另一种匿名 CVS 替代方案是 CVSup。CVSup 由 John Polstra[2] 开发，用于为 FreeBSD 项目[3]分发 CVS 仓库和其他文件树。

使用 CVSup 的一个主要优点是它能可靠地把整个 CVS 仓库复制到你的本地系统上，使 `log` 和 `diff` 等 cvs 操作可以快速本地进行。其他优点包括与 Postgres 服务器的快速同步——这得益于一种高效的流式传输协议，它只发送上次更新以来的变更。

DG1.3.1. 准备 CVSup 客户端系统

CVSup 工作需要两个目录区域：一个本地 CVS 仓库（如果你获取的是快照而非仓库，那就只是一个目录区域；见下文），以及一个本地 CVSup 簿记区域。它们可以共存于同一棵目录树中。

决定你想把本地 CVS 仓库放在哪里。最近我们在其中一台系统上把仓库建在了 `/home/cvs/`，但以前曾把它放在 `/opt/postgres/cvs/` 的 Postgres 开发树下。如果你打算把仓库放在 `/home/cvs/`，那么把

```
setenv CVSROOT /home/cvs
```

放进你的 `.cshrc` 文件，或者根据你使用的 shell 在 `.bashrc` 或 `.profile` 文件中放入类似的一行。

cvs 仓库区域必须先初始化。设置好 `CVSROOT` 之后，一条命令即可完成：

```
$ cvs init
```

之后列出 `CVSROOT` 目录时，你至少应该看到一个名为 `CVSROOT` 的目录：

```
$ ls $CVSROOT
CVSROOT/
```

DG1.3.2. 运行 CVSup 客户端

确认 cvsup 在你的路径中；在大多数系统上可以输入下面的命令来检查：

```
which cvsup
```

然后只需像下面这样运行 cvsup：

```
$ cvsup -L 2 postgres.cvsup
```

其中 `-L 2` 会启用一些状态消息，让你能监视更新的进度，而 *`postgres.cvsup`* 是你为 CVSup 配置文件指定的路径和名称。

[2] mailto:jdp@polstra.com
[3] http://www.freebsd.org

下面是一个针对特定安装修改过的 CVSup 配置文件，它维护一个完整的本地 CVS 仓库：

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@alumni.caltech.edu 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or by
 default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory points to where CVSup will store its 'bookmarks'
 file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory points to where CVSup will store the actual
 distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

下面是从 Postgres ftp 站点[4]建议使用的 CVSup 配置文件，它只获取当前快照：

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.
```

[4] ftp://ftp.postgresql.org/pub/CVSup/README.cvsup

```
# base directory points to where CVSup will store its 'bookmarks'
 file(s)
*default base=/usr/local/pgsql

# prefix directory points to where CVSup will store the actual
 distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

DG1.3.3. 安装 CVSup

CVSup 有源码、预构建二进制以及 Linux RPM 等形式。用二进制比从源码构建容易得多，主要因为构建需要那个能力很强但体积庞大的 Modula-3 编译器。

从二进制安装 CVSup

如果 Postgres ftp 站点[5]上发布了你所用平台的二进制，或者你运行的是 FreeBSD（CVSup 有对应的 port），就可以使用预构建的二进制。

> **注意**
>
> CVSup 最初是作为分发 FreeBSD 源码树的工具开发的。它以"port"的形式提供；对于运行 FreeBSD 的人来说，如果这些信息不足以说明如何获取和安装它，请在此贡献一份操作说明。

在写作本文时，可用的二进制包括 Alpha/Tru64、ix86/xBSD、HPPA/HPUX-10.20、MIPS/irix、ix86/linux-libc5、ix86/linux-glibc、Sparc/Solaris 和 Sparc/SunOS。

1. 取回适合你平台的 cvsup 二进制 tar 文件（作为客户端不需要 cvsupd）。

   a. (可选的) 如果你运行的是 FreeBSD，安装 CVSup 的 port。

   b. (可选的) 如果是其他平台，到 Postgres ftp 站点[6]查找并下载相应的二进制。

2. 检查 tar 文件以确认其内容和目录结构（如果有的话）。至少对 linux 的 tar 文件来说，里面只包含静态二进制和手册页，没有任何目录包装。

   a. 如果二进制位于 tar 文件的顶层，那么直接把 tar 文件解包到你的目标目录即可：

   ```
   $ cd /usr/local/bin
   $ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
   ```

[5] ftp://postgresql.org/pub
[6] ftp://postgresql.org/pub

```
$ mv cvsup.1 ../doc/man/man1/
```

b. 如果 tar 文件里有目录结构，那么把 tar 文件解包到 /usr/local/src 中，再把二进制移动到上面所说的合适位置。

3. 确保新的二进制在你的路径中。

```
$ rehash
$ which cvsup
$ set path=(path to cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

DG1.3.4. 从源码安装

从源码安装 CVSup 并不完全轻松，主要因为大多数系统需要先安装 Modula-3 编译器。这个编译器有 Linux RPM、FreeBSD 软件包或源码等形式。

> **注意**
>
> 从纯净源码安装 Modula-3 大约需要 200MB 磁盘空间，删除源码后缩小到大约 50MB。

Linux 安装

1. 安装 Modula-3。

a. 从 Polytechnique Montréal[7] 获取 Modula-3 发行包，他们正在积极维护最初由 DEC 系统研究中心[8]开发的代码基础。PM3 的 RPM 发行包压缩后约 30MB。在写作本文时，1.1.10-1 版可以在 RH-5.2 上干净地安装，而 1.1.11-1 版显然是为另一个发行版（RH-6.0？）构建的，无法在 RH-5.2 上运行。

> **提示**
>
> 这个 rpm 打包包含很多个 RPM 文件，所以你多半想把它们放到一个单独的目录中。

b. 安装 Modula-3 的 rpm：

```
# rpm -Uvh pm3*.rpm
```

2. 解包 cvsup 发行包：

```
# cd /usr/local/src
# tar zxf cvsup-16.0.tar.gz
```

[7] http://m3.polymtl.ca/m3
[8] http://www.research.digital.com/SRC/modula-3/html/home.html

3. 构建 cvsup 发行包，禁用 GUI 界面特性以避免依赖 X11 库：

```
# make M3FLAGS="-DNOGUI"
```

如果你想构建一个静态二进制以便移到可能没装 Modula-3 的系统上，试试：

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```

4. 安装构建好的二进制：

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

附录 DG2. 文档

Thomas Lockhart
Tom Ivar Helbekkmo

文档的目的是让 Postgres 更容易学习、使用和扩展。。文档集应当描述 Postgres 的系统、语言和接口。它应当能够回答常见问题，并让用户能够自行找到这些答案，
而不必求助于邮件列表支持。

DG2.1. 文档路线图

Postgres 有四种主要的文档格式：

- 纯文本，用于安装前信息。
- HTML，用于在线浏览和参考。
- 硬拷贝（Postscript 或 PDF），用于深入阅读和参考。
- 手册页，用于快速参考。

表 DG2.1. Postgres 文档产品

文件	描述
./COPYRIGHT	版权声明
./INSTALL	安装说明（文本由 sgml->rtf->text 生成）
./README	入门信息
./register.txt	make 期间的注册消息
./doc/bug.template	缺陷报告模板
./doc/postgres.tar.gz	集成文档（HTML）
./doc/programmer.ps.gz	程序员指南（Postscript）
./doc/programmer.tar.gz	程序员指南（HTML）
./doc/reference.ps.gz	参考手册（Postscript）
./doc/reference.tar.gz	参考手册（HTML）
./doc/tutorial.ps.gz	入门介绍（Postscript）
./doc/tutorial.tar.gz	入门介绍（HTML）
./doc/user.ps.gz	用户指南（Postscript）
./doc/user.tar.gz	用户指南（HTML）

有手册页可用，Postgres 源代码树各处还有大量纯文本 README 类型的文件。

DG2.2. 文档项目

打包的文档有 HTML 和 Postscript 两种格式。它们作为标准 Postgres 安装的一部分提供。我们在这里讨论如何使用文档源码以及如何生成文档包。

文档源码使用纯文本文件的 SGML 标记编写。DocBook SGML 的目的是让作者能够指定技术文档的结构和内容（使用 DocBook DTD），并由文档样式定义这些内容如何被渲染成最终形式（例如使用 Norm Walsh 的 Modular Style Sheets）。

参阅 Introduction to DocBook[1]，那里有一份不错的 DocBook 特性“快速入门”摘要。DocBook Elements[2] 为 DocBook 的特性提供了强大的交叉参考。

本文档集使用若干工具构建，包括 James Clark 的 jade[3] 和 Norm Walsh 的 Modular DocBook Stylesheets[4]。

目前，硬拷贝是通过把 jade 输出的富文本格式（RTF）导入 ApplixWare 做少量格式修正，然后导出为 Postscript 文件来生成的。

TeX[5] 是 jade 输出的一种受支持格式，但目前未被使用，原因有几点，包括在提交硬拷贝之前无法做小的格式修正，以及 TeX 样式表对表格的支持普遍不足。

DG2.3. 文档源码

文档源码包括纯文本文件、手册页和 html。不过，Postgres 的新文档大多将使用标准通用标记语言（SGML）DocBook[6] 文档类型定义（DTD）编写。现有文档的大部分已经或将被转换为 SGML。

SGML 的目的是让作者能够指定文档的结构和内容（例如使用 DocBook DTD），并由文档样式定义这些内容如何被渲染成最终形式（例如使用 Norm Walsh 的样式表）。

文档积累自多个来源。随着我们把现有文档整合成一个连贯的文档集，较旧的版本将逐渐过时，并会从发行包中删除。但是，这不会立即发生，也不会同时发生在所有文档上。为了简化过渡，并帮助引导开发者和作者，我们定义了一份过渡路线图。

DG2.3.1. 文档结构

目前有五份用 DocBook 编写的独立文档。每份文档都有一个容器源文档，它定义 DocBook 环境和其他文档源文件。这些主源文件位于 `doc/src/sgml/`，文档使用的许多其他源文件也在那里。主源文件有：

postgres.sgml

这是集成文档，把所有其他文档作为部分包含在内。它以 HTML 格式生成输出，因为浏览器界面让你只需点击就能在全部文档之间轻松跳转。其他文档同时提供 HTML 和硬拷贝两种格式。

tutorial.sgml

入门教程，带示例。不包括编程主题，旨在帮助不熟悉 SQL 的读者。这是“入门”文档。

user.sgml

用户指南。包括数据类型和用户级接口的信息。这是放置“为什么”类信息的地方。

reference.sgml

参考手册。包括 Postgres SQL 语法。这是放置“怎么做”类信息的地方。

[1] http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html
[2] http://www.ora.com/homepages/dtdparse/docbook/3.0/
[3] http://www.jclark.com/jade/
[4] http://www.nwalsh.com/docbook/dsssl/
[5] http://sunsite.unc.edu/pub/packages/TeX/systems/unix/
[6] http://www.ora.com/davenport/

programming.sgml

程序员指南。包括 Postgres 可扩展性以及编程接口的信息。

admin.sgml

管理员指南。包括安装说明和发行说明。

DG2.3.2. 样式与约定

DocBook 有一组丰富的标签和构造，其中直接而明显地适用于良构文档的百分比出人意料地高。Postgres 文档集最近才改写为 SGML，不久将来会从文档集中选出若干节，作为 DocBook 用法的示范性示例加以维护。此外，下面还会给出 DocBook 标签的简短摘要。

DG2.3.3. SGML 编写工具

当前的 Postgres 文档集是使用纯文本编辑器（或 emacs/psgml，见下文）编写的，内容用 SGML DocBook 标记。

SGML 和 DocBook 的开源编写工具并不多。最常见的工具集是带 psgml 功能扩展的 emacs/xemacs 编辑软件包。在某些系统（例如 RedHat Linux）上，典型的完整安装会包含这些工具。

DG2.3.3.1. emacs/psgml

emacs（以及 xemacs）有一种 SGML 主模式。正确配置后，它可以让你用 emacs 插入标签并检查标记的一致性。

把以下内容放入你的 `~/.emacs` 环境文件（把路径名调整为适合你系统的值）：

```
; ********** for SGML mode (psgml)

(setq sgml-catalog-files "/usr/lib/sgml/CATALOG")
(setq sgml-local-catalogs "/usr/lib/sgml/CATALOG")

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

并在同一文件中为 SGML 向（已有的）auto-mode-alist 定义中加入一项：

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

每个 SGML 源文件的末尾都有下面这一块：

```
!-- Keep this comment at the end of the file
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
sgml-indent-step:1
```

```
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
--
```

Postgres 发行包中包含一个已解析的 DTD 定义文件 `reference.ced`。你可能会发现

使用 emacs/psgml 时，处理这些分别保存书籍各部分的文件，有一种方便的方式：在编辑时插入适当的 DOCTYPE 声明。例如，当前这个源码文件是一章附录，因此可以将它指定为 DocBook 文档的 “appendix” 实例，把第一行写成这样：

```
!doctype appendix PUBLIC "-//Davenport//DTD DocBook V3.0//EN"
```

这意味着任何读取 SGML 的东西都能正确处理它，而且我可以用 “nsgmls -s docguide.sgml” 验证该文档。

DG2.4. 构建文档

GNU make 用于从 DocBook 源码构建文档。有一些环境定义可能需要为你的安装设置或修改。`Makefile` 会寻找 `doc/../src/Makefile` 和（隐式地）寻找 `doc/../src/Makefile.custom` 来获取环境信息。在我的系统上，`src/Makefile.custom` 看起来像

```
# Makefile.custom
# Thomas Lockhart 1998-03-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486
YFLAGS+= -v

# documentation

HSTYLE= /home/lockhart/SGML/db143.d/docbook/html
PSTYLE= /home/lockhart/SGML/db143.d/docbook/print
```

其中 HSTYLE 和 PSTYPE 分别决定 HTML 和硬拷贝（打印）样式表的 `docbook.dsl` 路径。这些样式表文件名是针对 Norm Walsh 的 Modular Style Sheets 的；如果使用其他样式表，可以把 HDSL 和 PDSL 定义为样式表的完整路径和文件名，就像上面为 HSTYLE 和 PSTYLE 所做的那样。在许多系统上，这些样式表会出现在安装于 `/usr/lib/sgml/`、`/usr/share/lib/sgml/` 或 `/usr/local/lib/sgml/` 的软件包中。

HTML 文档包可以通过输入以下命令从 SGML 源码生成

```
% cd doc/src
% make tutorial.tar.gz
% make user.tar.gz
% make admin.tar.gz
% make programmer.tar.gz
% make postgres.tar.gz
% make install
```

这些包可以从主文档目录安装，输入

```
% cd doc
% make install
```

DG2.5. 手册页

我们使用 docbook2man 工具把 DocBook `REFENTRY` 页面转换为适合手册页的 *roff 输出。在撰写本文时，该工具需要打补丁才能在 Postgres 标记上成功运行，而且我们添加了少量新功能，允许在输出文件名中设置手册页的节号。

docbook2man 用 perl 编写，需要 CPAN 包 `SGMLSpm` 才能运行。此外，它还需要有 nsgmls 可用，后者包含在 jade 发行包中。安装这些包之后，只需运行

```
$ cd doc/src
$ make man
```

就会在 `doc/src` 目录中生成一个 tar 文件。

docbook2man 安装过程

1. 安装 docbook2man 包，可从 http://shell.ipoline.com/~elmert/comp/docbook2X/ 获得
2. 安装 SGMLSpm perl 模块，可从 CPAN 镜像获得。
3. 如果你的 jade 安装中还没有 nsgmls，请安装它。

DG2.6. v7.0 的硬拷贝生成

硬拷贝 Postscript 文档的生成方法是：先把 SGML 源码转换为 RTF，然后导入 ApplixWare-4.4.1。经过少量清理（见下一节）后，把输出“打印”到一个 postscript 文件。

DG2.6.1. 文本硬拷贝

`INSTALL` 和 `HISTORY` 每个发行版都会更新。由于历史原因，这些文件是纯文本的，但它们派生自较新的 SGML 源码。

纯文本生成

`INSTALL` 和 `HISTORY` 都是从现有的 SGML 源码生成的。它们从同一个中间 RTF 文件中提取。

1. 输入以下命令生成 RTF：

   ```
   % cd doc/src/sgml
   % make installation.rtf
   ```

2. 把 `installation.rtf` 导入 Applix Words。
3. 设置页宽和页边距。
   a. 在 File.PageSetup 中把页宽调整为 10 英寸。

b. 选中全部文本。用标尺把右边距调整为 9.5 英寸。这样最大列宽就是 79 个字符，在 80 列上限目标之内。

4. 把文档中不需要的部分去掉。

   对于 `INSTALL`，删除文本末尾除当前发行版之外的所有发行说明。对于 `HISTORY`，删除发行说明之前的所有文本，保留并修改标题和目录（ToC）。

5. 把结果导出为 “ASCII Layout” 。

6. 用 emacs 或 vi 清理 `INSTALL` 中的表格信息。删除移植贡献者的 “mailto” URL 以缩小列的高度。

DG2.6.2. Postscript 硬拷贝

在生成 Postscript 硬拷贝时需要处理若干方面的问题，包括 RTF 修复、目录（ToC）生成和分页调整。

Applixware RTF 清理

硬拷贝过程不可或缺的组成部分 jade 没有为正文文本指定默认样式。过去，这个未确诊的问题导致目录（ToC）生成过程极其漫长。不过，在 ApplixWare 方面的大力帮助下，症状已得到诊断，并且有了可用的变通方法。

1. 输入以下命令生成 RTF 输入（例如）：

```
% cd doc/src/sgml
% make tutorial.rtf
```

2. 修复 RTF 文件，使其正确指定所有样式，特别是默认样式。该字段可以用 vi 或者下面这个小小的 sed 过程添加：

```
#!/bin/sh
# fixrtf.sh
# Utility to repair slight damage in RTF files generated by jade
# Thomas Lockhart <lockhart@alumni.caltech.edu>
#
for i in $* ; do
  mv $i $i.orig
  cat $i.orig | sed 's#\\stylesheet#\\stylesheet{\\s0 Normal;}#' >
 $i
done

exit
```

   该脚本会在 `{\s0 Normal;}` 的位置添加它作为文档的第 0 号样式。按照 ApplixWare 的说法，RTF 标准不允许添加隐式的第 0 号样式，不过 M$Word 恰好能处理这种情况。

3. 在Applix Words中打开一个新文档，然后导入RTF文件。

4. 使用 ApplixWare 生成新目录（ToC）。

   a. 从第一行第一个字符开始到最后一行最后一个字符为止，选中现有的 ToC 行。

b. 使用 `Tools.BookBuilding.CreateToC`构建新的 ToC。选择前三级标题。这会用 ApplixWare 原生的 ToC 替换从 RTF 导入的现有行。

c. 使用 `Format.Style` 调整 ToC 格式，依次选择三种 ToC 样式，并调整`First` 和`Left` 的缩进。使用以下数值：

表 DG2.2. 目录的缩进格式

样式	首行缩进（英寸）	左缩进（英寸）
`TOC-Heading 1`	`0.6`	`0.6`
`TOC-Heading 2`	`1.0`	`1.0`
`TOC-Heading 3`	`1.4`	`1.4`

5. 在整个文档中完成以下工作：

   - 调整分页。
   - 调整表格列宽。
   - 把插图插入文档。使用 ApplixWare 工具栏上的居中边距按钮把每幅图在页面上居中。

> **注意**
>
> 并非所有文档都有插图。你可以在 SGML 源文件中 grep 字符串“`graphic`”，来找出文档中可能带有插图的部分。有少数插图在文档的不同部分重复出现。

6. 用正确的值替换 ToC 中 Examples 和 Figures 部分右对齐的页码。这只需要几分钟（每份文档）。
7. 如果存在书目，从每个条目中删除短形式引用标题。Norm Walsh 的 DocBook 样式表似乎会把这些打印出来，尽管这只是紧随其后的信息的一个子集。
8. 将文档保存为 Applix Words 原生格式，以便日后更容易地进行最后时刻的编辑。
9. 把文档“打印”到 PostScript 格式的文件。
10. 使用 gzip 压缩 Postscript 文件。把压缩后的文件放入 `doc` 目录。

DG2.7. 工具集

我们记录了处理文档所需各种工具的三种安装方法的经验。第一种是在 Linux 上从 RPM 安装，第二种是从 FreeBSD port 安装，最后一种是从各工具的原始发行包进行通用安装。下面将分别介绍。

这些工具可能还有其他打包发行形式。请把软件包状态报告到 docs 邮件列表，我们会把相关信息收录在这里。

DG2.7.1. Linux RPM 安装

对 RedHat 兼容的 Linux 系统来说，最简单的安装方式是使用 Cygnus 的 Mark Galassi 开发的 RPM 集。也可以像后面一节描述的那样从源码安装。

安装 RPM

1. 安装 Jade 及相关软件包的 RPM[7]。
2. (可选的) 安装 Norm Walsh 的最新样式表。视 RPM 的新旧程度而定，最新样式表可能比 RPM 中包含的有显著改进。
3. 更新你的 `src/Makefile.custom`，加入指向样式表的 HSTYLE 和 PSTYLE 定义。

DG2.7.2. FreeBSD 安装

FreeBSD 上有一整套文档工具的 ports。事实上，postgresql.org ––文档每天晚上都会自动更新的那台机器––就是一台 FreeBSD 机器。

安装 FreeBSD Ports

1. 要在 FreeBSD 上构建文档，需要安装若干 ports。

```
% cd /usr/ports/devel/gmake && make install
% cd /usr/ports/textproc/docproj && make install
% cd /usr/ports/textproc/docbook && make install
% cd /usr/ports/textproc/dsssl-docbook-modular && make install
```

2. (可选的) 设置环境变量以访问 jade 工具集。

> **注意**
>
> 在 postgresql.org 的 FreeBSD 机器上并不需要这一步，所以你可能不必这么做。

```
export SMGL_ROOT=/usr/local/share/sgml
SGML_CATALOG_FILES=/usr/local/share/sgml/jade/catalog
SGML_CATALOG_FILES=/usr/local/share/sgml/html/catalog:$SGML_
CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/iso8879/catalog:$SGML_
CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/transpec/catalog:$SGML_
CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/docbook/catalog:$SGML_
CATALOG_FILES
export SGML_CATALOG_FILES
```

（这是 sh/bash 语法；csh/tcsh 请作相应调整。）

3. make 需要一些特殊参数，或者把它们加入你的 Makefile.custom：

```
HSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/html/
PSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/print/
```

当然，构建时你需要使用 gmake 而不是普通的“make”。

[7] ftp://ftp.cygnus.com/pub/home/rosalia/

DG2.7.3. Debian 安装

Debian 有一整套文档工具的软件包。

安装 Debian 软件包

1. 安装 jade、docbook 和 unzip：

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

2. (可选的) 安装最新的样式表。

   a. (可选的) 确认已安装 unzip，或安装该软件包：

   ```
   apt-get install unzip
   ```

   b. 从 http://www.nwalsh.com/docbook/dsssl 抓取最新的样式表 zip 包，并解压到某个位置（可以是 /usr/share）。

3. 编辑 src/Makefile.custom，加入适当的 HSTYLE 和 PSTYLE 定义：

```
HSTYLE= /usr/share/docbook/html
PSTYLE= /usr/share/docbook/print
```

DG2.7.4. 手动安装工具

这里简要过一遍获取和安装软件的过程，你需要这些软件来用 Emacs 编辑 DocBook 源码，并用 Norman Walsh 的 DSSSL 样式表处理它，生成 HTML 和 RTF。

获取 SGML 和 DocBook 工具最简单的方法，可能是从 sgmltools[8] 获取 sgmltools。sgmltools 需要 GNU 版本的 m4。要确认你有正确的 m4 版本可用，试试

```
gnum4 --version
```

如果你安装 GNU m4，用 gnum4 这个名字安装，sgmltools 就会找到它。安装之后，你将拥有 sgmltools、jade 和 Norm Walsh 的 DocBook 样式表。下面说明如何单独安装这些工具。

DG2.7.4.1. 先决条件

你需要：

- 一个可用的 GCC 2.7.2 安装
- 一个可用的 Emacs 19.19 或更高版本安装
- 一个 Unix 的 unzip 程序用来解包

你必须获取：

[8] http://www.sgmltools.org/

- James Clark 的 Jade[9]（撰写本文时，`jade1_1.zip` 中的 1.1 版是当前版本）
- DocBook 3.0 版[10]
- Norman Walsh 的 Modular Stylesheets[11]（最初生成这些文档时使用的是 1.19 版）
- Lennart Staflin 的 PSGML[12]（撰写本文时，`psgml-1.0.1.tar.gz` 中的 1.0.1 版可用）

重要 URL：

- Jade 网页[13]
- DocBook 网页[14]
- Modular Stylesheets 网页[15]
- PSGML 网页[16]
- Steve Pepper 的 Whirlwind Guide[17]
- Robin Cover 的 SGML 软件数据库[18]

DG2.7.4.2. 安装 Jade

安装 Jade

1. 阅读上面列出的 URL 处的安装说明。
2. 把发行包解压到一个合适的位置。使用的命令大致是

   ```
   unzip -aU jade1_1.zip
   ```

3. Jade 不是用 GNU autoconf 构建的，所以你需要自己编辑 `Makefile`。既然 James Clark 已经为此准备好了他的套件，一个好办法是在 Jade 发行包的主目录下建一个构建目录（也许以你的机器架构命名），把主目录中的 `Makefile` 文件复制进去，在那里编辑它，然后在那里运行 `make`。

   不过，`Makefile` 确实需要编辑。主目录中有一个名为 `Makefile.jade` 的文件，构建 Jade（而不是构建 Jade 所基于的 SGML 解析器套件 SP）时，可以用 `make -f Makefile.jade` 来使用它。不过我们建议你不要那样做，因为需要修改的内容超出了 `Makefile.jade` 的范围，反正你也得编辑其中某一个。

   通读 `Makefile`，阅读 James 的说明并按需编辑。有若干变量需要设置。下面收集汇总了最重要的一些变量及其典型值：

   ```
   prefix = /usr/local
   ```

[9] ftp://ftp.jclark.com/pub/jade/
[10] http://www.ora.com/davenport/docbook/current/docbk30.zip
[11] http://nwalsh.com/docbook/dsssl/
[12] ftp://ftp.lysator.liu.se/pub/sgml/
[13] http://www.jclark.com/jade/
[14] http://www.ora.com/davenport/
[15] http://nwalsh.com/docbook/dsssl/
[16] http://www.lysator.liu.se/projects/about_psgml.html
[17] http://www.infotek.no/sgmltool/guide.htm
[18] http://www.sil.org/sgml/publicSW.html

```
XDEFINES = -DSGML_CATALOG_FILES_DEFAULT=\"/usr/local/share/sgml/
catalog\"
XLIBS = -lm
RANLIB = ranlib
srcdir = ..
XLIBDIRS = grove spgrove style
XPROGDIRS = jade
```

注意其中指定了查找 SGML 支持文件默认 catalog 的位置 —— 你可能想把它改成更适合你自己安装的值。如果你的系统不需要上面这些数学库和 `ranlib` 命令的设置，就让它们保持 `Makefile` 中的原样。

4. 输入 `make` 构建 Jade 和各种 SP 工具。

5. 软件构建完成后，`make install` 会完成显而易见的那一步。

DG2.7.4.3. 安装 DocBook DTD 工具包

安装 DocBook DTD 工具包

1. 你会想把构成 DocBook DTD 工具包的文件放在你构建 Jade 时让它查找的目录中；如果你遵循了上面的建议，那就是 `/usr/local/share/sgml/`。除了 DocBook 实际文件外，你还需要放好一个 `catalog` 文件，用于把文档类型规范和外部实体引用映射到该目录中的实际文件。你还需要 ISO 字符集映射，可能还有一或多个版本的 HTML。

   安装各种 DTD 和支持文件并建立 `catalog` 文件的一种办法是，把它们全部收进上面提到的目录，用一个名为 `CATALOG` 的文件描述它们全部，然后创建文件 `catalog` 作为指向前者的 catalog 指针，给它这样一行内容：

   ```
   CATALOG /usr/local/share/sgml/CATALOG
   ```

2. `CATALOG` 文件随后应包含三类行。第一类是（可选的）SGML 声明，即：

   ```
   SGMLDECL docbook.dcl
   ```

   接下来，对 DTD 和实体文件的各项引用必须解析。对于 DocBook 文件，这些行看起来像这样：

   ```
   PUBLIC "-//Davenport//DTD DocBook V3.0//EN" docbook.dtd
   PUBLIC "-//USA-DOD//DTD Table Model 951010//EN" cals-tbl.dtd
   PUBLIC "-//Davenport//ELEMENTS DocBook Information Pool V3.0//EN"
    dbpool.mod
   PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy V3.0//
   EN" dbhier.mod
   PUBLIC "-//Davenport//ENTITIES DocBook Additional General Entities
    V3.0//EN" dbgenent.mod
   ```

3. 当然，包含这些内容的文件随 DocBook 工具包一起提供。注意这些行每行的最后一项都是文件名，这里没有带路径。你当然可以把文件放在主 SGML 目录的子目录中，并修改 `CATALOG` 文件中的引用。DocBook 还引用 ISO 字符集实体，所以你需要获取并安装它们（它们可从多个来源获得，通过上面列出的 URL 很容易找到），并为它们全部配上 catalog 条目，例如：

```
PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" ISO/ISOlat1
```

注意这里的文件名包含目录名，表明我们把 ISO 实体文件放在了名为 `ISO` 的子目录中。同样，你获取的实体包应附带正确的 catalog 条目。

DG2.7.4.4. 安装 Norman Walsh 的 DSSSL 样式表

安装 Norman Walsh 的 DSSSL 样式表

1. 阅读上面列出的 URL 处的安装说明。
2. 要安装 Norman 的样式表，只需把发行包解压到合适的位置。一个合适的位置是 `/usr/local/share`，这会把工具包放到 `/usr/local/share/docbook` 目录树下。命令大致是

   ```
   unzip -aU db119.zip
   ```

3. 测试安装的一种办法是构建 PostgreSQL 用户指南的 HTML 和 RTF 形式。

   a. 要构建 HTML 文件，进入 SGML 源目录 `doc/src/sgml`，执行

   ```
   jade -t sgml -d /usr/local/share/docbook/html/docbook.dsl -D ..
   /graphics postgres.sgml
   ```

   `book1.htm` 是输出的顶层节点。。

   b. 要生成可导入你喜欢的字处理系统并打印的 RTF 输出，输入：

   ```
   jade -t rtf -d /usr/local/share/docbook/print/docbook.dsl -D ..
   /graphics postgres.sgml
   ```

DG2.7.4.5. 安装 PSGML

安装 PSGML

1. 阅读上面列出的 URL 处的安装说明。
2. 解开发行文件，运行 configure、make 和 make install，把字节编译文件和 info 库放到位置。
3. 然后把以下几行加入你的 `/usr/local/share/emacs/site-lisp/site-start.el` 文件，让 Emacs 在需要时正确载入 PSGML：

   ```
   (setq load-path
         (cons "/usr/local/share/emacs/site-lisp/psgml" load-path))
   (autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t)
   ```

4. (可选的) 如果你想在编辑 HTML 时也使用 PSGML，还要加上：

   ```
   (setq auto-mode-alist
   ```

```
(cons '("\\.s?html?\\'" . sgml-mode) auto-mode-alist))
```

5. (可选的) 使用 PSGML 有一点需要注意：它的作者假定你的主 SGML DTD 目录是 `/usr/local/lib/sgml`。如果像本章示例中那样使用 `/usr/local/share/sgml`，你就必须对此作出补偿。

   a. (可选的) 你可以设置 `SGML_CATALOG_FILES` 环境变量。

   b. (可选的) 你可以定制你的 PSGML 安装（它的手册说明了方法）。

   c. (可选的) 你甚至可以在编译和安装 PSGML 之前编辑源文件 `psgml.el`，把硬编码的路径改成你自己的默认值。

DG2.7.4.6. 安装 JadeTeX

如果你愿意，还可以安装 JadeTeX，用 TeX 作为 Jade 的格式化后端。注意它仍是相当粗糙的软件，生成的打印输出不如 RTF 后端的质量。不过它仍然工作得不错，尤其是对不使用表格的较简单文档而言；而且由于 JadeTeX 和样式表都在持续改进，它将来肯定会越来越好。

要安装和使用 JadeTeX，你需要可用的 TeX 和 LaTeX2e 安装，包括受支持的 tools 和 graphics 宏包、Babel、AMS 字体和 AMS-LaTeX、PSNFSS扩展及 “35 种字体” 配套包、用于生成 PostScript 的 dvips 程序、宏包 fancyhdr、hyperref、minitoc、url 和 ot2enc，当然还有 JadeTeX 本身。所有这些都可以在邻近的 CTAN 站点找到。

JadeTeX 在撰写本文时并没有附带多少安装指南，但有一个 `makefile` 展示了需要什么。它还包含一个 `cooked` 目录，你会在其中找到它所需的部分宏包 —— 但不是全部，也不完整，至少我们上次查看时是这样。

在构建 `jadetex.fmt` 格式文件之前，你可能需要编辑 `jadetex.ltx` 文件，把 Babel 的配置改为适合你所在地区的形式。要改的那一行看起来像

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

显然，你应该只列出你确实需要、并且已为 Babel 配置好的语言。

JadeTeX 可用后，你就应该能为 PostgreSQL 手册生成并排版 TeX 输出，办法是（如上所述，在 `doc/src/sgml` 目录中）给出命令

```
jade -t tex -d /usr/local/share/docbook/print/docbook.dsl -D ../
graphics postgres.sgml
jadetex postgres.tex
jadetex postgres.tex
dvips postgres.dvi
```

当然，这样做时，TeX 会在第二轮运行中停下来，并告诉你它的容量已被超出。就我们所知，这是 JadeTeX 生成交叉引用信息的方式造成的。当然，TeX 可以用更大的数据结构尺寸编译。具体细节因安装而异。

DG2.8. 备用工具集

sgml-tools v2.x 支持 jade 和 DocBook。

参考书目

这里列出了一些关于 SQL 和 Postgres 的参考文献与读物。

来自最初的 Postgres 开发团队的一些白皮书和技术报告，可在加州大学伯克利分校计算机科学系网站[1]上获取。

SQL 参考书

SQL 特性的参考文本。

The Practical SQL Handbook . Bowman et al, 1996 . Using Structured Query Language . 3. Judith Bowman、Sandra Emerson和Marcy Darnovsky. ISBN 0-201-44787-8. 1996. Addison-Wesley. 版权 © 1996 Addison-Wesley Longman, Inc..

A Guide to the SQL Standard . Date and Darwen, 1997 . A user's guide to the standard database language SQL . 4. C. J. Date和Hugh Darwen. ISBN 0-201-96426-0. 1997. Addison-Wesley. 版权 © 1997 Addison-Wesley Longman, Inc..

An Introduction to Database Systems . Date, 1994 . 6. C. J. Date. 1. 1994. Addison-Wesley. 版权 © 1994 Addison-Wesley Longman, Inc..

Understanding the New SQL . Melton and Simon, 1993 . A complete guide. Jim Melton和Alan R. Simon. ISBN 1-55860-245-3. 1993. Morgan Kaufmann. 版权 © 1993 Morgan Kaufmann Publishers, Inc..

Abstract

SQL 特性的易读参考。

Principles of Database and Knowledge . Base Systems . Ullman, 1988 . Jeffrey D. Ullman. 1. Computer Science Press . 1988 .

PostgreSQL 专属文档

本节收录相关文档。

The PostgreSQL. Administrator's Guide . The Administrator's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Developer's Guide . The Developer's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Programmer's Guide . The Programmer's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Tutorial Introduction . The Tutorial . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. User's Guide . The User's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/

Enhancement of the ANSI SQL Implementation of PostgreSQL . Simkovics, 1998 . Stefan Simkovics. with support by . O.Univ.Prof.Dr.. Georg. Gottlob. Univ.Ass. Mag.. Katrin. Seyr. .

讨论 SQL 的历史和语法，并描述向 Postgres 中加入 INTERSECT 和 EXCEPT 构造的过程。在维也纳技术大学的 O.Univ.Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 的支持下作为硕士论文完成。

November 29, 1998. Department of Information Systems, Vienna University of Technology .

The Postgres95. User Manual . Yu and Chen, 1995 . A. Yu和J. Chen. . Sept. 5, 1995. University of California, Berkeley CA.

会议论文和期刊文章

本节收录文章和简报。

Partial indexing in POSTGRES: research project . Olson, 1993 . Nels Olson. 1993. UCB Engin T7. 49.1993 O676. University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System . Ong and Goh, 1990 . L. Ong和J. Goh. April, 1990. ERL Technical Memorandum M90/33. University of California, Berkeley CA.

The Postgres. Data Model . Rowe and Stonebraker, 1987 . L. Rowe和M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

Generalized partial indexes [2]. Seshardri, 1995 . P. Seshadri和A. Swami. Eleventh International Conference on Data Engineering, March 1995. 1995. Cat. No.95CH35724. IEEE Computer Society Press.

The Design of Postgres. . Stonebraker and Rowe, 1986 . M. Stonebraker和L. Rowe. Conference on Management of Data, Washington DC, May 1986.

The Design of the Postgres. Rules System. Stonebraker, Hanson, Hong, 1987 . M. Stonebraker、E. Hanson和C. H. Hong. Conference on Data Engineering, Los Angeles, CA, Feb. 1987.

The Postgres. Storage System . Stonebraker, 1987 . M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

A Commentary on the Postgres. Rules System . Stonebraker et al, 1989. M. Stonebraker、M. Hearst和S. Potamianos. Record 18(3), Sept. 1989.

The case for partial indexes (DBMS) [3]. Stonebraker, M, 1989b. M. Stonebraker. Record 18(no. 4):4-11, Dec. 1989.

The Implementation of Postgres. . Stonebraker, Rowe, Hirohama, 1990 . M. Stonebraker、L. A. Rowe和M. Hirohama. Transactions on Knowledge and Data Engineering 2(1), March 1990.

On Rules, Procedures, Caching and Views in Database Systems . Stonebraker et al, ACM, 1990 . M. Stonebraker和et al. Conference on Management of Data, June 1990.

[2] http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
[3] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf