PostgreSQL 6.3.2 手册

PostgreSQL 6.3.2 手册

涵盖正式发布的 v6.3

目录

插图清单

表格清单

摘要

Postgres 最初在加州大学伯克利分校计算机科学系开发，首创了许多对象关系概念，这些概念目前正在一些商业数据库中变得可用。它提供 SQL92/SQL3 语言支持、事务完整性和类型可扩展性。PostgreSQL 是这套原始伯克利代码的公有领域开源后继版本。

部分 I. 教程

面向新用户的介绍。

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，一直得到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

1.2.1. 伯克利 Postgres 项目

Postgres DBMS 的实现始于 1986 年。该系统的最初概念发表于 [[STON86]]，而最初数据模型的定义出现在 [[ROWE87]]。当时规则系统的设计描述于 [[STON87a]]。存储管理器的基本原理和体系结构则详细记述于 [[STON87b]]。

从那以后，Postgres经历了几个主要版本。第一个"演示系统"于 1987 年投入运行，并在 1988 年的 ACM-SIGMOD 大会上展出。我们在 1989 年 6 月向少数外部用户发布了版本 1，其描述见 [[STON90a]]。针对对第一个规则系统的批评 ([[STON89]])，规则系统被重新设计 ([[STON90b]])，版本 2 于 1990 年 6 月随新的规则系统一起发布。版本 3 出现于 1991 年，增加了对多个存储管理器的支持、一个改进的查询执行器和一个重写的重写规则系统。从那以后，各版本的重心大多放在可移植性和可靠性上。

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

Postgres已被用来实现许多不同的研究和生产应用。这些应用包括：一个金融数据分析系统、一个喷气发动机性能监控包、一颗小行星跟踪数据库、
一个医疗信息数据库以及若干地理信息系统。Postgres还曾在多所大学被用作教学工具。最后，Illustra Information Technologies[3]（Illustra 信息技术公司）接手了代码并将其商业化。Postgres于 1992 年底成为 Sequoia 2000[4] 科学计算项目的主要数据管理器。此外，外部用户社区的规模在 1993 年间几乎翻了一番。越来越明显的是，原型代码的维护和支持正在占用本应投入到数据库研究中的大量时间。为了减轻这一支持负担，该项目在版本 4.2 之后正式结束。

1.2.2. Postgres95

1994 年，Andrew Yu[5] 和 Jolly Chen[6] 为Postgres添加了 SQL 语言解释器，随后代码被发布到 Web 上自谋生路。Postgres95是这一原始伯克利代码的公共域、开源后代。

Postgres95是Postgres最后一个官方版本（4.2 版）的衍生版本。代码现在完全是 ANSI C，代码规模缩减了 25%。有大量改进性能和代码可维护性的内部变化。与 v4.2 相比，Postgres95 v1.0.x 在 Wisconsin 基准测试上快了大约 30-50%。除错误修复外，主要的增强包括：

- 查询语言Postquel已被替换为 SQL（在服务器中实现）。我们尚不支持子查询（可以用用户定义的SQL 函数来模仿）。聚合已被重新实现。我们还添加了对``GROUP BY''的支持。`libpq`接口仍可供C 程序使用。
- 除 monitor 程序外，我们还提供了一个新的程序（psql），它支持GNU `readline`。
- 我们添加了新的前端库`libpgtcl`，它支持基于Tcl的客户端。示例 shell pgtclsh 提供了新的 Tcl 命令，使tcl 程序能够与Postgres95后端交互。
- 大对象接口经过了彻底检修。我们只保留 Inversion 大对象作为存储大对象的唯一机制。（不要把这与已被移除的 Inversion 文件系统相混淆。）
- 实例级的规则系统已被移除。规则仍以重写规则的形式可用。
- 随源代码分发了一个简短的教程，介绍常规SQL特性以及我们自己的特性。
- 构建使用的是GNU make（而非BSD make）。此外，Postgres95可以用未打补丁的 gcc编译（double 的数据对齐已经修复）。

1.2.3. PostgreSQL

到了 1996 年，情况已经很清楚：“Postgres95” 这个名字经不起时间的考验。于是选择了一个新名字PostgreSQL，以反映原始 Postgres与具备SQL能力的较新版本之间的关系。与此同时，版本编号被重置为从 6.0 开始，使编号回到Postgres项目最初开始的序列。

Postgres95 v1.0.x 各版本的开发重点在于稳定后端代码。而在PostgreSQL的 v6.x 系列中，重点已从识别和理解后端中现存的问题转移到增强功能和能力上，不过各方面的工作都在继续。

主要的增强包括：

- 重要的后端特性已经实现，包括子查询、默认值、约束和触发器。
- 添加了更多符合SQL92的语言特性，包括主键、带引号的标识符、字符串字面量类型强制、类型转换，以及二进制和十六进制整数输入。

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

- 内建类型得到了改进，包括新的宽范围日期/时间类型和更多的几何类型支持。
- 后端代码整体速度提高了约 20%，后端启动速度降低了 80%。

1.3. 关于本版本

从现在起，我们用Postgres指代PostgreSQL。

PostgreSQL可以免费获取。本手册描述PostgreSQL的 6.3 版。

请查阅管理员指南以了解当前支持的机器列表。一般而言，PostgreSQL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

本手册集组织成几个部分：

教程

面向新用户的入门介绍。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的可扩展性、库接口以及应用设计问题。

管理员指南

安装和管理信息。所支持机器的列表。

开发者指南

面向Postgres开发者的信息。它面向那些为Postgres项目做贡献的人；
应用开发信息应出现在程序员指南中。

参考手册

关于命令语法的详细参考信息。目前这本手册还非常简略，
但最终应包含与手册页类似的信息。

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 pages

手册页包含关于命令语法的一般信息。

FAQs

常见问题（FAQ）文档涉及一般性问题以及一些平台特定的问题。

READMEs

一些贡献的软件包带有 README 文件。

Web 站点

Postgres[7]网站上有一些未出现在发行版中的信息。那里有一个mhonarc邮件列表归档，是许多主题的丰富资源。

邮件列表

Postgres Questions[8] 邮件列表是让用户问题得到解答的好地方。还有其他邮件列表可用；详情请查阅网页。

你自己！

Postgres是一个开源产品。因此，它依靠用户社区提供持续的支持。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。
请考虑把你的知识回馈出来。如果你学到了文档中没有的东西，把它写下来并贡献出去。
如果你给代码添加了特性，请贡献它。
即使是没有太多经验的人也可以对文档提供更正和小的修改，而这是一个很好的起点。
Postgres Documentation[9] 邮件列表就是开始的地方。

1.5. 版权和商标

[7] postgresql.org
[8] mailto:questions@postgresql.org
[9] mailto:docs@postgresql.org

第 2 章 体系结构

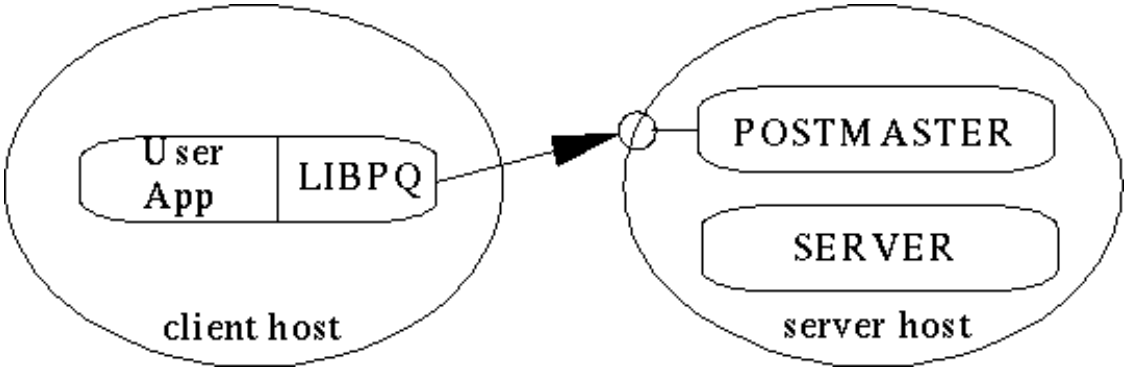
2.1. Postgres 体系结构概念

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 2.1），后者转而启动一个新的后端服务器进程

图 2.1. 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 3 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。

3.1. 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。

3.2. 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。

从 Postgres v6.3 起，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。

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'?
```

or

```
% 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 (1) postmaster 没有在运行，或者 (2) 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。

3.3. 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。

3.3.1. 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
WARN: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。

3.3.2. 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 (e.g. psql)，它允许你交互式地输入、编辑并执行 SQL 命令。

- 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[programmers-guide]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。

3.3.3. 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：

```
% destroydb 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 4 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 SQL 已有许多著作，包括 [MELT93] 和 [DATE97]。你应当知道，某些语言特性并不属于 ANSI 标准。

4.1. 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 `\?`。

4.2. 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。

4.3. 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变得区分大小写，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。

4.4. 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` 命令从平面（ASCII）文件装载大量数据。

4.5. 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;
```

```
+--------------+---------+---------+------+------------+
|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;
```

4.6. 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。

4.7. 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。

4.8. 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';
```

4.9. 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。

4.10. 使用聚合函数

与大多数其他查询语言一样，PostgreSQL 支持聚合函数。当前实现的 Postgres 聚合函数有一些限制。具体来说，虽然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），但聚合只能出现在查询的目标列表中，不能直接出现在限定条件（where 子句）里。举个例子，

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

是允许的，而

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

则不行。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);
```

聚合还可以带 group by 子句：

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

第 5 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 4 章。）

5.1. 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           char2
) 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 “*” 记法。

5.2. 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。

5.2.1. 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        char16[][]
);
```

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 char16 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"},{""}} |
+--------------------+
```

5.3. 时间旅行

从 Postgres v6.2 起，不再支持时间旅行。原因有几个：性能影响、存储大小，以及一个会在短时间内增长到无限大小的 pg_time 文件。

触发器等新特性允许在需要时模拟时间旅行的行为，而在不需要时（对大多数用户来说，大多数时候都不需要）不产生开销。更多信息见 `contrib` 目录中的示例。

> **时间旅行已废弃**
>
> 本节余下的文字只是暂时保留，直到能够用实现相同目的的新技术的语境重写为止。
> 有人志愿吗？—— thomas 1998-01-12

Postgres 支持时间旅行的概念。这个特性允许用户运行历史查询。例如，要查找 Mariposa 市当前的人口，可以查询：

```
SELECT * FROM cities WHERE name = 'Mariposa';

+---------+------------+----------+
|name     | population | altitude |
+---------+------------+----------+
```

```
|Mariposa | 1320       | 1953     |
+---------+------------+----------+
```

Postgres 会自动找到在当前时间有效的 Mariposa 记录版本。还可以给出一个时间范围。例如要查看 Mariposa 过去和现在的人口，可以查询：

```
SELECT name, population
    FROM cities['epoch', 'now']
    WHERE name = 'Mariposa';
```

其中"epoch"表示系统时钟的起点。

> **注意**
>
> 在 UNIX 系统上，这总是 1970 年 1 月 1 日 GMT 午夜。

如果你已经执行了目前为止的所有示例，那么上面的查询返回：

```
+---------+------------+
|name     | population |
+---------+------------+
|Mariposa | 1200       |
+---------+------------+
|Mariposa | 1320       |
+---------+------------+
```

时间范围的默认起点是系统可表示的最早时间，默认终点是当前时间；因此，
上面的时间范围可以缩写为 ``[,].''

5.4. 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 II. 用户指南

面向用户的信息。

目录

第 6 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 7 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。

7.1. 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：

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

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。

7.2. 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。

可以指定一个绝对路径名或者一个环境变量作为位置。
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。
关于预先配置的备选数据库位置，请咨询站点管理员。

> **注意**
>
> 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。

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。

要在 /alt/postgres/data 中创建一个数据存储区，请确保 /alt/postgres 已经存在。在命令行输入

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

Creating Postgres database system directory /alt/postgres/data/base
```

要使用环境变量 PGDATA2 做同样的事，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 /alt/postgres/data 中创建数据库，输入

```
% createdb -D /alt/postgres/data mydb
```

或者

```
% 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：

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

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：

```
% createdb -D /alt/postgres/data mydb
ERROR:  Unable to create database directory /alt/postgres/data/base/
mydb
createdb: database creation failed on mydb.
```

7.3. 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 Postgres 终端监视程序（例如 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。这一接口在第 ?? 节中有进一步讨论。

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 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=>
```

这个提示表明终端监视程序正在监听你，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，你可以获得各种 Postgres 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，在监视程序提示符下输入 `\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 “--” 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

7.3.1. 数据库权限

7.3.2. 表权限

TBD

7.4. 删除数据库

如果你是数据库 mydb 的数据库管理员，你可以使用下面的 UNIX 命令销毁它：

```
% destroydb 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 8 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。用户可以用在别处描述的 `define 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 8.1. 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
circle		二维平面上的圆
date	date	不含时间的日历日期
float4/8	float(p)	精度为 p 的浮点数
float8	real, double precision	双精度浮点数
int2	smallint	有符号二字节整数
int4	int, integer	有符号 4 字节整数
int4	decimal(p,s)	p <= 9、s = 0 的精确数值
int4	numeric(p,s)	p == 9、s = 0 的精确数值
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
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
time	time	一天中的时间
timespan	interval	通用时间段
timestamp	timestamp with time zone	日期/时间
varchar(n)	character varying(n)	变长字符串

表 8.2. 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 拥有位于 ORDBMS 发展前沿的特性。除了符合 SQL3 外，还支持 SQL92 的很大一部分。尽管我们力求 SQL92 兼容，但该标准中有一些情形考虑欠周，不应延续到后续标准中。

Postgres 不会花大力气去符合这些情形。不过这些情形往往很少使用且冷僻，典型用户不太可能遇到。

虽然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，但有些检查得并不特别严格。更重要的是，很少有运算符和函数（例如加法和乘法）做任何错误检查。因此，许多数值运算符可能（例如）静默地下溢或上溢。

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。

8.1. 数字类型

数值类型包括二字节和四字节整数以及四字节和八字节浮点数。

表 8.3. Postgres 数值类型

数值类型	存储尺寸	描述	范围
int2	2字节	固定精度	-32768 到 +32767
int4	4 字节	固定精度的常用选择	-2147483648 到 +2147483647
float4	4 字节	可变精度	7 位十进制小数
float8	8字节	可变精度	14 位十进制小数

精确数值`decimal` 和 `numeric` 拥有完整实现的语法，但目前（Postgres v6.3）只支持较小的精度和/或取值范围。

8.2. 货币类型

`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 类型应使用为 locale(7) 定义的货币惯例。

表 8.4. Postgres 数值类型

货币类型	存储尺寸	描述	范围
money	4 字节	固定精度	-21474836.48 到 +21474836.47

numeric 最终应取代 money 类型。它拥有完整实现的语法，但目前（Postgres v6.3）只支持较小的精度和/或取值范围，尚不能替代 money 类型。

8.3. 字符类型

SQL92 定义了两种主要的字符类型：`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声明上限。

表 8.5. 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

字符类型	存储尺寸	建议	描述
varchar(n)	(4+n) 字节	SQL92 兼容	有长度限制的变长

目前还有其他一些定长字符类型。它们不提供额外的功能，将来很可能被弃用。

表 8.6. Postgres 特殊字符类型

字符类型	存储尺寸	描述
char2	2字节	两个字符
char4	4 字节	四个字符
char8	8字节	八个字符
char16	16字节	十六个字符

8.4. 日期/时间类型

日期时间度量有两种基本类型：时钟时间和时间区间。两种量都具有连续性和平滑性，就像时间本身一样。Postgres 提供两种主要的面向用户的日期时间类型 `datetime` 和 timespan，以及相关的 SQL92 类型 date 和 time。

另外还有一些其他日期时间类型可用，大多出于历史原因。

表 8.7. Postgres 日期/时间类型

日期/时间类型	存储尺寸	建议	描述
abstime	4 字节	原始日期和时间	范围有限
date	4 字节	SQL92 类型	范围广
datetime	8字节	最佳通用日期和时间	范围广、精度高
interval	12字节	SQL92 类型	等价于 timespan
reltime	4 字节	原始时间区间	范围有限、精度低
time	4 字节	SQL92 类型	范围广
timespan	12字节	最佳通用时间区间	范围广、精度高
timestamp	4 字节	SQL92 类型	范围有限

表 8.8. Postgres 日期/时间范围

日期/时间类型	最早	最晚	精度
abstime	1901-12-14	2038-01-19	1秒
date	4713 BC	无限制	1日
datetime	4713 BC	无限制	1微秒到14位
interval	无限制	无限制	1微秒
reltime	-68年	+68年	1秒
time	00:00:00.00	23:59:59.99	1微秒
timespan	无限制	无限制	1微秒（14位）
timestamp	1901-12-14	2038-01-19	1秒

Postgres 力求在典型用法上与 SQL92 定义兼容。SQL92 标准对日期和时间类型及能力有一种奇怪的组合。例如，尽管 date 类型没有关联的时区，time 类型却可以有关联的时区。默认时区被

指定为相对 GMT/UTC 的固定偏移；然而，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会变化。

为消除这些困难，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。

在未来的版本中，日期时间类型的数量将会减少：当前 `datetime` 的实现将变成 `timestamp`，timespan 变成 interval，而 abstime 和 reltime（可能）被弃用，转而使用 `timestamp` 和 interval。SQL92 标准中日期时间定义的那些更晦涩的特性不太可能被追求。

8.4.1. 日期/时间样式

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

表 8.9. Postgres 日期样式

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

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。

表 8.10. Postgres 日期顺序惯例

样式说明	描述	示例
European	地区惯例	17/12/1997 15:37:16.00 MET
NonEuropean	地区惯例	12/17/1997 07:37:16.00 PST
US	地区惯例	12/17/1997 07:37:16.00 PST

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。

对 Postgres v6.3（及更早版本），默认的日期/时间风格是"non-European traditional Postgres"（非欧式传统 Postgres）。在未来的版本中，默认值可能变成 ISO-8601，它会减轻日期指定的歧义和 Y2K 排序问题。

8.4.2. 时区

Postgres 从底层操作系统获得时区支持。所有日期和时间在内部都以协调世界时（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。

- `set timezone` SQL 命令设置会话的时区。

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）。

8.4.3. 日期/时间输入

通用日期时间的输入可采用多种风格，包括 ISO 兼容、SQL 兼容、传统 Postgres 以及日期时间的其他排列组合。在解释可能有歧义的情况下（许多传统日期指定风格很可能如此），Postgres 使用一个风格设置来解决歧义。

大多数日期时间类型共用数据输入代码。对这些类型，输入可以采用多种多样的风格。对于数字日期表示，欧式与美式惯例可能不同，正确的解释要在输入数据之前使用 `set datestyle` 命令来获得。注意，风格设置并不妨碍在输入中使用各种风格；它主要用于确定输出风格和解决歧义。

提供了特殊值 `current'、`infinity' 和 `-infinity'。`infinity' 指定一个晚于任何其他有效时间的时间，而 `-infinity' 指定一个早于任何其他有效时间的时间。`current' 表示每当该值出现在计算中时就以当前时间替换。字符串 `now'、`today'、`yesterday'、`tomorrow' 和 `epoch' 可用于指定时间值。`now' 表示当前事务时间，它与 `current' 的区别在于当前时间会被立即替换进去。`epoch' 表示 Jan 1 00:00:00 1970 GMT。

表 8.11. 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
today	今天午夜
tomorrow	明天午夜
yesterday	昨天午夜

8.4.4. datetime

通用日期时间的输入可采用多种风格，包括 ISO 兼容、SQL 兼容、传统 Postgres（见“绝对时间”一节）以及日期时间的其他排列组合。输出风格可以是 ISO 兼容、SQL 兼容或传统 Postgres，默认设置为与 Postgres v6.0 兼容。

`datetime` 用下列语法指定：

```
Year-Month-Day [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
  YearMonthDay [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
     Month Day [ Hour : Minute : Second ] Year [AD,BC] [ Timezone ]
```

其中

`Year`（年）是 `4013 BC, ..., 非常大`
`Month`（月）是 `Jan, Feb, ..., Dec` 或 `1, 2, ..., 12`
`Day`（日）是 `1, 2, ..., 31`
`Hour`（时）是 `00, 02, ..., 23`
`Minute`（分）是 `00, 01, ..., 59`

    Second (秒) 是 00, 01, ..., 59 (闰秒为 60)
    Timezone (时区) 是 3 个字符或相对 GMT 的 ISO 偏移

有效日期从公元前 4013 年 11 月 13 日 00:00:00 GMT 到遥远的未来。时区要么是三个字符（如 "GMT" 或 "PST"），要么是与 GMT 的 ISO 兼容偏移（如太平洋标准时间时为 "-08" 或 "-08:00"）。日期在内部以格林尼治标准时间存储。输入和输出例程会在必要时把时间转换为服务器的本地时区。

8.4.5. timespan

通用时间区间的输入可采用多种语法，包括 ISO 兼容、SQL 兼容、传统 Postgres（见“相对时间”一节）以及时间区间的其他排列组合。输出格式可以是 ISO 兼容、SQL 兼容或传统 Postgres，默认设置为与 Postgres 兼容。月和年是“定性”时间区间，与日或小时等其他“定量”时间区间分开存储。在日期运算中，定性时间单位会在相关日期或时间的上下文中实例化。

时间区间用下列语法指定：

```
  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'、`millenium'，或这些单位的缩写和复数形式。
    Direction (方向) 是 `ago'。

8.4.6. abstime

绝对时间（abstime）是一种范围有限（±68 年）、精度有限（1 秒）的日期数据类型。可能更宜使用 datetime，因为它覆盖更大的范围且精度更高。

绝对时间用下列语法指定：

```
Month  Day [ Hour : Minute : Second ]  Year [ Timezone ]
```

其中

    Month (月) 是 Jan, Feb, ..., Dec
    Day (日) 是 1, 2, ..., 31
    Hour (时) 是 01, 02, ..., 24
    Minute (分) 是 00, 01, ..., 59
    Second (秒) 是 00, 01, ..., 59
    Year (年) 是 1901, 1902, ..., 2038

有效日期从 Dec 13 20:45:53 1901 GMT 到 Jan 19 03:14:04 2038 GMT。从版本 3.0 起，时间的读写不再使用格林尼治标准时间；输入和输出例程默认使用本地时区。所有允许用于 datetime 的特殊值也同样允许用于“绝对时间”。

8.4.7. reltime

相对时间 reltime 是一种范围有限（±68 年）、精度有限（1 秒）的时间区间数据类型。应优先使用 timespan，因为它覆盖更大的范围、精度更高，而且更重要的是能区分相对单位（月和年）与定量单位（日、小时等）。而 reltime 必须把一个月强制当作恰好 30 天，因此时间运算并不总是按预期工作。例如，把一个 reltime 的 year 加到 abstime 的 today 上，得到的不是从今天起一年的日期，而是从今天起 360 天的日期。

`reltime` 与其他时间区间类型共用输入和输出例程。`timespan` 一节对此有更详细的介绍。

8.4.8. `timestamp`

这是一种目前范围有限的绝对时间，与 abstime 数据类型非常相似。它与其他日期时间类型共用通用输入解析器。在未来的版本中，此类型将吸收 `datetime` 类型的能力，并向 SQL92 兼容的方向发展。

`timestamp` 使用与 `datetime` 相同的语法指定。

8.4.9. `interval`

`interval` 是一种 SQL92 数据类型，目前被映射到 `timespan` Postgres 数据类型。

8.4.10. tinterval

时间范围指定为：

```
[ 'abstime' 'abstime']
```

其中

`abstime` 是绝对时间格式的一个时间。

可以使用诸如 `current'、`infinity' 和 `-infinity' 这样的特殊 abstime 值。

8.5. 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 4 字节存储。

表 8.12. 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'

8.6. 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 8.13. 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

几何类型	存储尺寸	表示	描述
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。

8.6.1. 点

点用下列语法指定：

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

其中

`x` 是 `x` 轴坐标，以浮点数表示
`y` 是 `y` 轴坐标，以浮点数表示

8.6.2. 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

`(x1,y1)` 和
`(x2,y2)` 是线段的两个端点

8.6.3. 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

`(x1,y1)` 和 `(x2,y2)` 是方框的对角

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。

8.6.4. 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) 可用于在查询中选择这两种类型。

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

```
    (x1,y1),...,(xn,yn) 是第 1 到第 n 个点
    前导 "[" 表示开放路径
    前导 "(" 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。

8.6.5. 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

```
    (x1,y1),...,(xn,yn) 是第 1 到第 n 个点
```

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。

8.6.6. 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

```
    (x,y) 是圆的圆心
    r 是圆的半径
```

圆使用第一种语法输出。

第 9 章 运算符

Postgres 在系统类型上提供大量内建运算符。这些运算符声明在系统目录 pg_operator 中。pg_operator 中的每个条目都包括实现该运算符的过程的名称以及输入和输出类型的类 OIDs。

要查看 “||” 字符串连接运算符的所有变体，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)
```

表 9.1. Postgres 运算符

运算符	描述	用法
<	小于?	1 < 2
<=	小于等于?	1 <= 2
<>	不等于?	1 <> 2
=	等于?	1 = 1
>	大于?	2 > 1
>=	大于等于?	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*.'

表 9.2. Postgres 数值运算符

运算符	描述	用法
!	阶乘	3 !
!!	阶乘（左运算符）	!! 3
%	取模	5 % 4
%	截断	% 4.5
*	乘法	2 * 3
+	加法	2 + 3
-	减法	2 - 3
/	除法	4 / 2

运算符	描述	用法
:	自然指数	: 3.0
;	自然对数	(; 5.0)
@	绝对值	@ -5.0
^	幂运算	2.0 ^ 3.0
|/	平方根	|/ 25.0
||/	立方根	||/ 27.0

表 9.3. Postgres 几何运算符

运算符	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))'
#	多边形的点数	# '((1,0),(0,1),(-1,0))'
##	最近点	'(0,0)'::point ## '((2,0),(0,2))'::lseg
&&	重叠?	'((0,0),(1,1))'::box && '((0,0),(2,2))'::box
&<	在左侧重叠?	'((0,0),(1,1))'::box &< '((0,0),(2,2))'::box
&>	在右侧重叠?	'((0,0),(3,3))'::box &> '((0,0),(2,2))'::box
<->	间距	'((0,0),1)'::circle <-> '((5,0),1)'::circle
<<	在左边?	'((0,0),1)'::circle << '((5,0),1)'::circle
<^	在下方?	'((0,0),1)'::circle <^ '((0,5),1)'::circle
>>	在右边?	'((5,0),1)'::circle >> '((0,0),1)'::circle
>^	在上方?	'((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?	'(1,0)'::point ?- '(0,0)'::point
?-|	是垂直的?	'((0,0),(0,1))'::lseg ?-| '((0,0),(1,0))'::lseg
@-@	长度或周长	@-@ '((0,0),(1,0))'::path
?|	是竖直的?	'(0,1)'::point ?| '(0,0)'::point
?||	是平行的?	'((-1,0),(1,0))'::lseg ?|| '((-1,2),(1,2))'::lseg

运算符	描述	用法
@	包含于或位于	'(1,1)'::point @ '((0,0),2)'::circle
@@	圆心	@@ '((0,0),10)'::circle
~=	等于	'((0,0),(1,1))'::polygon ~= '((1,1),(0,0))'::polygon

时间区间数据类型 `tinterval` 是最初的日期时间类型留下的遗产，不像更现代的类型那样受到良好支持。此类型有几个运算符。

表 9.4. Postgres 时间区间运算符

运算符	描述	用法
#<	区间小于?	
#<=	区间小于等于?	
#<>	区间不等于?	
#=	区间等于?	
#>	区间大于?	
#>=	区间大于等于?	
<#>	转换为时间区间	
<<	区间小于?	
|	区间的起点	
~=	等于	
<?>	时间在区间内?	

用户可以用运算符名称调用运算符，例如：

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

或者，用户也可以直接调用实现这些运算符的函数。此时上面的查询将写作：

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

psql 有一个 `\dd` 命令用于显示这些运算符。

第 10 章 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。此外还有一些特定于类型的函数。
同时可通过运算符使用的函数仅以运算符的形式记载。

为 text 定义的一些函数也可用于 char() 和 varchar()。

对于 `date_part` 和 `date_trunc` 函数，参数可以是 `year'、`month'、`day'、`hour'、`minute' 和 `second'，以及更专门的量 `decade'、`century'、`millenium'、`millisecond' 和 `microsecond'。`date_part` 还允许用 `dow' 返回星期几，用 `epoch' 返回自 1970 年以来的秒数（对 `datetime`），或用 'epoch' 返回总经过秒数（对 `timespan`）。

表 10.1. 数学函数

函数	返回值	描述	示例
float(int)	float8	将整数转换为浮点数	float(2)
float4(int)	float4	将整数转换为浮点数	float4(2)
int	integer(float)	将浮点数转换为整数	integer(2.0)

许多字符串函数可用于 text、varchar() 和 char() 类型。目前有些函数只能用于 text 类型。

表 10.2. 字符串函数

函数	返回值	描述	示例
lower(text)	text	将文本转换为小写	lower('TOM')
lpad(text,int,text)	text	在左侧将字符串填充到指定长度	lpad('hi',4,'??')
ltrim(text,text)	text	从文本左侧去除字符	ltrim('xxxxtrim','x')
position(text,text)	text	提取指定的子串	position('high','ig')
rpad(text,int,text)	text	在右侧将字符串填充到指定长度	rpad('hi',4,'x')
rtrim(text,text)	text	从文本右侧去除字符	rtrim('trimxxxx','x')
substr(text,int[,int])	text	提取指定的子串	substr('hi there',3,5)
upper(text)	text	将文本转换为大写	upper('tom')

表 10.3. 日期/时间函数

函数	返回值	描述	示例
isfinite(abstime)	bool	如果这是一个有限时间则为 TRUE	isfinite('now'::abstime)
datetime(abstime)	datetime	转换为 datetime	datetime('now'::abstime)
datetime(date)	datetime	转换为 datetime	datetime('today'::date)
datetime(date,time)	datetime	转换为 datetime	datetime('1998-02-24'::datetime, '23:07'::time);
age(datetime,datetime)	timespan	保留月和年的区间	age('now','1957-06-13'::datetime)

函数	返回值	描述	示例
date_part(text,datetime)	float8	日期字段的指定部分	date_part('dow','now'::datetime)
date_trunc(text,datetime)	datetime	在指定单位处截断日期	date_trunc('month','now'::abstime)
isfinite(datetime)	bool	如果这是一个有限时间则为 TRUE	isfinite('now'::datetime)
abstime(datetime)	abstime	转换为 abstime	abstime('now'::datetime)
timespan(reltime)	timespan	转换为 timespan	timespan('4 hours'::reltime)
datetime(date,time)	datetime	转换为 datetime	datetime('1998-02-25'::date,'06:41'::time)
date_part(text,timespan)	float8	时间字段的指定部分	date_part('hour','4 hrs 3 mins'::timespan)
isfinite(timespan)	bool	如果这是一个有限时间则为 TRUE	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	转换为 reltime	reltime('4 hrs'::timespan)

表 10.4. 几何函数

函数	返回值	描述	示例
box(point,point)	box	将点转换为方框	box('(0,0)'::point,'(1,1)'::point)
area(box)	float8	方框的面积	area('((0,0),(1,1))'::box)
isopen(path)	bool	如果这是开放路径则为 TRUE	isopen('[(0,0),(1,1),(2,0)]'::path)
isclosed(path)	bool	如果这是封闭路径则为 TRUE	isclosed('((0,0),(1,1),(2,0))'::path)
circle(point,float8)	circle	转换为圆	circle('(0,0)'::point,2.0)
polygon(npts,circle)	polygon	转换为带 npts 个点的多边形	polygon(12,'((0,0),2.0)'::circle)
center(circle)	float8	对象的中心	center('((0,0),2.0)'::circle)
radius(circle)	float8	圆的半径	radius('((0,0),2.0)'::circle)
diameter(circle)	float8	圆的直径	diameter('((0,0),2.0)'::circle)
area(circle)	float8	圆的面积	area('((0,0),2.0)'::circle)

SQL92 定义了一些具有特定语法的函数。其中一些是用其他 Postgres 函数实现的。

表 10.5. SQL92 文本函数

函数	返回值	描述	示例
position(text in text)	int4	提取指定的子串	position('o' in 'Tom')

函数	返回值	描述	示例
substring(text [from int] [for int])	text	提取指定的子串	substring('Tom' from 2 for 2)
trim([leading|trailing|both] [text] from text)	text	从文本中去除字符	trim(both 'x' from 'x Tomx')

第 11 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。为了说明它们的用途，
我们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        char16[][]
);
```

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 char16 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"}}');
```

默认情况下，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。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 SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

这个查询检索所有员工的第三季度薪水：

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
```

```
+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

我们也可以访问数组的任意切片或子数组。下面的查询检索 Bill 周计划前两天的第一项。

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

+--------------------+
|schedule            |
+--------------------+
|{{"meeting"},{""}} |
+--------------------+
```

第 12 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           char2
) 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     |
+----------+----------+
```

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` 也是如此。

第 13 章 查询语言

> **注意**
>
> 本章必须深入讲解查询语言的每个领域。目前是教程的一份副本。- thomas 1998-01-12

Postgres 查询语言是 SQL3 的一个变种。它有许多扩展，例如可扩展的类型系统、继承、函数和产生式规则。那些都是从最初的 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 SQL 已有许多著作。例如可参考 [MELT93] 或 [DATE93]。你还应当知道，某些特性并不属于 ANSI 标准。

13.1. 概念

Postgres 中的基本概念是类（class），它是对象实例的一个命名集合。每个实例都具有同一组命名属性，而每个属性都有特定的类型。此外，每个实例都有一个在整个安装中唯一的永久对象标识符（OID）。由于 SQL 语法使用表的说法，我们将把术语表（table）和类（class）互换使用。同样，SQL 的行（row）就是实例（instance），而 SQL 的列（column）就是属性（attribute）。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。

13.2. 创建新类

你可以通过指定类名以及所有属性名及其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
);
```

注意关键字是大小写不敏感的，标识符通常也是大小写不敏感的。Postgres 允许 SQL92 的分隔标识符（由双引号包围的标识符）包含大小写混合以及空格、制表符等。

Postgres SQL 支持通常的 SQL 类型 `int`、`float`、`real`、`smallint`、`char(N)`、`varchar(N)`、`date`、`time`、`timestamp`，以及其他通用类型和一组丰富的几何类型。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，类型名不是语法关键字，除非为支持 SQL92 标准中的特殊情况所必需。到目前为止，Postgres 的 create 命令看起来与传统关系系统中创建表所用的命令完全一样。但我们马上就会看到，类具有一些关系模型扩展性质的属性。

13.3. 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')
```

你还可以使用 `copy` 命令从平面（ASCII）文件装载大量数据。这通常更快，因为数据作为单个原子事务直接读入（或写出）目标表。例如：

```
COPY INTO weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

其中源文件的路径名必须为后端服务器机器可访问，而不仅仅是客户端。

13.4. 查询类

weather 类可以用正常的关系选择和投影查询来查询。这通过 SQL 的 `select` 语句完成。该语句分为选择列表（列出要返回的属性的部分）和限定条件（指定限制条件的部分）。例如，要检索 weather 的所有行，键入：

```
SELECT * FROM WEATHER;
```

输出应为：

```
+--------------+---------+---------+------+------------+
|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 |
+--------------+---------+---------+------+------------+
```

你可以在选择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;
```

任意布尔操作符（`and`、`or` 和 `not`）都可用于任何查询的限定条件中。例如，

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

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

最后要说明的是，你可以指定 select 的结果以排序顺序返回，或者去除重复实例。

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

13.5. 重定向 SELECT 查询

任何 select 查询都可以被重定向到一个新类

```
SELECT * INTO TABLE temp FROM weather;
```

这构成一个隐式的 `create` 命令，按照 `select into` 命令的选择列表中指定的属性名和类型创建新类 temp。当然，随后我们可以对这个结果类执行任何可对其他类执行的操作。

13.6. 类之间的连接

到目前为止，我们的查询一次只访问一个类。查询可以一次访问多个类，
或者以某种方式访问同一个类，使得该类的多个实例同时被处理。
一次访问同一个或不同类的多个实例的查询称为连接查询。例如，
假设我们希望找出所有处于其他记录温度范围内的记录。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**
>
> 这只是一种概念模型。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。

我们可以用以下查询做到这一点：

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

+--------------+---------+---------+---------------+---------+-------
--+
|city          | temp_lo | temp_hi | city          | temp_lo | temp_hi
 |
+--------------+---------+---------+---------------+---------+-------
--+
|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：限定条件是为查询中指定的各类的笛卡尔积定义的真值表达式。
> 对于笛卡尔积中限定条件为真的那些实例，Postgres 计算并返回选择列表中指定的值。Postgres SQL 不为此类表达式中的重复值赋予任何意义。这意味着 Postgres 有时会重复计算同一选择列表多次；当布尔表达式用 "or" 连接时经常发生这种情况。要去除此类重复，必须使用 `select distinct` 语句。

在本例中，W1 和 W2 都是类 weather 一个实例的代理（surrogate），二者都取遍该类的所有实例。（用大多数数据库系统的术语来说，W1 和 W2 被称为范围变量。）查询可以包含任意数量的类名和代理。

13.7. 更新

你可以使用 update 命令更新现有实例。假设你发现从 11 月 28 日起所有温度读数都偏高 2 度，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';
```

13.8. 删除

删除使用 `delete` 命令执行：

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

所有属于 Hayward 的天气记录都被删除。对如下形式的查询应当小心

```
DELETE FROM classname;
```

不带限定条件时，`delete` 会直接删除给定类的所有实例，使其变空。系统在执行此操作前不会请求确认。

13.9. 使用聚合函数

与大多数其他查询语言一样，PostgreSQL 支持聚合函数。当前 Postgres 聚合函数的实现有一些限制。具体来说，虽然有计算 `count`、`sum`、`avg`（平均值）、`max`（最大值）和 `min`（最小值）等函数（在一组实例上）的聚合，聚合只能出现在查询的选择列表中，而不能直接出现在限定条件（where 子句）中。例如，

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

是允许的，而

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

则不行。不过，正如常见的情况，查询可以改写以达到预期的结果；这里使用子查询：

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

聚合还可以带 group by 子句：

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

第 14 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 15 章 psql

本节需要从原始的 psql 手册页转换而来。欢迎志愿者！

> **提示**
>
> 要更改结果的分页行为，请设置/取消设置你的 PAGER 环境变量。

15.1. 分页到屏幕

> **作者**
>
> 来自 Brett McCormick 1998-04-04 在邮件列表上的文章。

要影响 `psql` 输出的分页行为，请设置或取消设置你的 PAGER 环境变量。我总是必须先设置它才会暂停。当然，你必须在启动程序之前完成此操作。

在 csh/tcsh 或其他 C shell 中：

```
unsetenv PAGER
```

而在 sh/bash 或其他 Bourne shell 中：

```
unset PAGER
```

第 16 章 pgaccess

本节有待撰写。欢迎志愿者！

部分 III. 管理员指南

安装与维护信息。

目录

第 17 章 平台移植

17.1. 当前支持的平台

Postgres 可免费获取。本手册描述 Postgres 6.3 版。作者们在下列平台上编译和测试过 Postgres：

表 17.1. 支持的平台

OS	处理器	版本	报告日期	备注
AIX 4.1.x-4.2	RS6000	v6.3	1998-03-01	4.1.4.0、4.2 (Darren King[1]), 4.1.5 (Andreas Zeugswetter[2]); 3. 2.5 已在 v6.2.1 上确认 (Frank Dana[3])
BSDi	x86	v6.3	1998-03-01	(Bruce Momjian[4]
FreeBSD 2.2.x-3.x	x86	v6.3	1998-03-01	(Tatsuo Ishii[5], Marc Fournier[6])
NetBSD 1.3	x86	v6.3	1998-03-01	(Brook Milligan[7])
NetBSD 1.3	Sparc	v6.3	1998-03-01	(Tom I Helbek kmo[8])
NetBSD 1.3	VAX	v6.3	1998-03-01	(Tom I Helbek kmo[9])
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	(Brian E Gallew[10])
HPUX 10.20	PA-RISC	v6.3	1998-03-01	9.0.x 已在 v6.2. 1 上确认 (Stan Brown[11])
IRIX 6.x	MIPS	v6.3	1998-03-01	5.x 有所不同 (Andrew Martin[12])
Digital 4.0	Alpha	v6.3.2	1998-04-16	据报告在 DUnix/ v3.2g 上可用 (Pedro J. Lobo[13])
linux 2.0.x	Alpha	v6.3.2	1998-04-16	基本成功 (Ryan Kirkpatrick[14], Jeff Sturm [15])

[1] mailto:darrenk@insightdist.com
[2] mailto:Andreas.Zeugswetter@telecom.at
[3] mailto:danaf@ans.net
[4] mailto:maillist@candle.pha.pa.us
[5] mailto:t-ishii@sra.co.jp
[6] mailto:scrappy@hub.org
[7] mailto:brook@trillium.NMSU.Edu
[8] mailto:tih@hamartun.priv.no
[9] mailto:tih@hamartun.priv.no
[10] mailto:geek+@cmu.edu
[11] mailto:stanb@awod.com
[12] mailto:martin@biochemistry.ucl.ac.uk
[13] mailto:pjlobo@euitt.upm.es
[14] mailto:rkirkpat@nag.cs.colorado.edu
[15] mailto:jsturm@zenacomp.com

OS	处理器	版本	报告日期	备注
linux 2.0.x	x86	v6.3.2	1998-04-16	(Thomas Lockhart[16], Tatsuo Ishii[17])
linux 2.0.x	Sparc	v6.3	1998-03-01	(Tom Szybist[18])
mklinux	PPC	v6.3	1998-03-01	(Tatsuo Ishii[19])
SCO	x86	v6.3	1998-03-01	部分成功 (Billy G. Allie[20])
Solaris	x86	v6.3	1998-03-01	(Marc Fournier[21])
Solaris 2.5.1-2.6	x86	v6.3	1998-03-01	(Marc Fournier[22], Tatsuo Ishii[23])
SunOS 4.1.4	Sparc	v6.3	1998-03-01	已提交补丁 (Tatsuo Ishii[24])
SVR4	MIPS	v6.3	1998-03-01	与 v6.2.1 相似；"基本可用" (Frank Ridderbusch[25])
SVR4 4.4	m88k	v6.2.1	1998-03-01	打补丁后确认可用 (Doug Winterburn[26])
Unixware	x86	v6.3	1998-03-01	即 UNIVEL (Billy G. Allie[27])
NextStep	x86	v6.x	1998-03-01	仅客户端支持；v1.0.9 加补丁可用 (David Wetzel[28])

17.2. 未支持的平台

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 17.2. 可能不兼容的平台

OS	处理器	版本	报告日期	备注
MacOS	all	v6.3	1998-03-01	库不兼容；请使用 ODBC/JDBC
NetBSD	arm32	v6.3	1998-03-01	尚未可用 (Dave Millen[29])

[16] mailto:lockhart@alumni.caltech.edu
[17] mailto:t-ishii@sra.co.jp
[18] mailto:szybist@boxhill.com
[19] mailto:t-ishii@sra.co.jp
[20] mailto:Bill.Allie@mug.org
[21] mailto:scrappy@hub.org
[22] mailto:scrappy@hub.org
[23] mailto:t-ishii@sra.co.jp
[24] mailto:t-ishii@sra.co.jp
[25] mailto:ridderbusch.pad@sni.de
[26] mailto:dlw@seavme.xroads.com
[27] mailto:Bill.Allie@mug.org
[28] mailto:dave@turbocat.de
[29] mailto:dmill@globalnet.co.uk

OS	处理器	版本	报告日期	备注
NetBSD	m68k	v6.3	1998-03-01	Amiga、HP300、Mac；尚未可用 (Henry Hotz[30])
Ultrix	MIPS,VAX?	v6.x	1998-03-01	近期无报告；已过时?
Windows NT	all	v6.3	1998-03-01	库不兼容；客户端也许可行；请使用 ODBC/JDBC
Windows	x86	v6.3	1998-03-01	库不兼容；客户端也许可行；请使用 ODBC/JDBC

注意，使用第三方的 Posix 移植工具和库，前端在 Windows 上的移植显然是可能的。

[30] mailto:hotz@jpl.nasa.gov

第 18 章 安装

Postgres v6.3 的完整安装说明。

本过程基于 Postgres v6.3 的安装说明（见 `$PGROOT/INSTALL`）。关于 Postgres 的最新信息可在 www.postgresql.org[1] 找到。

下面的安装说明假定如下（除特别说明外）：

- 命令与 Unix 兼容。见下面的注记。
- 除特别说明外均使用默认值。
- 用户 postgres 是 Postgres 超级用户。
- 源代码路径为 /usr/src/pgsql（其他路径也可以）。
- 运行时路径为 /usr/local/pgsql（其他路径也可以）。

这些命令是在 RedHat Linux 4.2 版上使用 tcsh shell 测试的。除特别说明外，它们在大多数系统上很可能都能工作。像 ps 和 tar 这样的命令，在每个平台上应该使用哪些选项差异极大。输入这些命令之前请运用常识。

我们的 Makefile 需要 GNU make（本文档中称 “gmake”），并且假定 install 接受 BSD 选项。Makefile 中的 INSTALL 变量被设置为 BSD 兼容版本的 install。在某些系统上，你必须找一个 BSD 兼容的 install（例如 MIT X Window System 发行版自带的 bsdinst）。

18.1. 运行 Postgres 的要求

关于所支持平台的信息在另一章中。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 Postgres。

你应当至少有 8 MB 内存，以及至少 45 MB 磁盘空间来容纳源代码、二进制文件和用户数据库。安装之后可以将其减少到约 3 MB 加上用户数据库所需的空间。

18.2. 安装过程

Postgres 安装

对于全新安装或从旧版本 Postgres 升级：

1. 阅读任何最后时刻的信息和针对特定平台的移植说明。本文件末尾有关于 Ultrix4.x、Linux、BSD/OS 和 NeXT 的一些平台特定说明。目录 `/usr/src/pgsql/doc` 中还有其他文件，包括 FAQ-Irix 和 FAQ-Linux。也看看目录 ftp://ftp.postgresql.org/pub。如果该目录中有一个名为 INSTALL 的文件，那么该文件包含最新的安装信息。

   请注意，前面列表中的"已测试"平台只是意味着有人曾经花力气确保 Postgres 发行版能在该平台上不经修改地编译和运行。由于当前的开发者无法访问所有这些平台，
   其中一些可能因小问题而无法在当前版本中干净地编译并通过回归测试。
   任何此类已知问题及其解决方案都会公布在 ftp://ftp.postgresql.org/pub/INSTALL。

2. (可选的) 如果账号 postgres 尚不存在，则创建它。

3. 登入账号 postgres。

[1] http://www.postgresql.org

- 检查你是否有足够的磁盘空间。/usr/src/pgsql 大约需要 17 MB，/usr/local/pgsql（不含你的数据库）大约需要 2 MB，一个空数据库需要 1 MB。回归测试期间数据库会临时增长到约 20 MB。发行版 tar 文件还需要约 3 MB。

  因此我们建议，在安装和测试期间，/usr/local 之下保留远超 20 MB 的空闲空间，在包含数据库的磁盘分区上另外保留 25 MB 空闲空间。删除源文件、tar 文件和回归数据库之后，/usr/local/pgsql 需要 2 MB，空数据库需要 1 MB，再加上大约五倍于把数据库数据存成平面文件所需的空间。

  要检查磁盘空间，使用 `df -k`。

4. 从 Internet 上 Ftp 文件 ftp://ftp.postgresql.org/pub/postgresql-v6.3.tar.gz。把它存放在你的主目录中。

5. 某些平台使用 flex。如果你的系统使用 flex，请确保你有一个好的版本。要检查，输入 `flex --version`。

   如果找不到 flex 命令，那么你很可能不需要它。如果版本是 2.5.2、2.5.4 或更高，就没有问题。如果是 2.5.3 或低于 2.5.2，你就得升级 flex。可以从 ftp://prep.ai.mit.edu/pub/gnu/flex-2.5.4.tar.gz 获取。

   如果你需要 flex 却没有它或版本不对，那么当你尝试编译程序时会被告知。如果不确定是否需要，尽可以跳过这一步。如果确实需要，当你尝试编译时会被告知安装/升级 flex。

   要安装它，输入以下命令：

   ```
   cd
   gunzip -c flex-2.5.4.tar.gz | tar xvf -
   cd flex-2.5.4
   configure --prefix=/usr
   make
   make check
   # You must be root when typing the next line.
   make install
   cd
   rm -rf flex-2.5.4
   ```

   这将更新文件 /usr/man/man1/flex.1、/usr/bin/flex、/usr/lib/libfl.a、/usr/include/FlexLexer.h，并添加指向 flex 的链接 /usr/bin/flex++。

6. 如果你是在升级现有系统，请备份你的数据库。对于 alpha 和 beta 级的发布，数据库格式容易变化，往往每隔几周就变一次，除了 HACKERS 邮件列表中的一句快速评论之外没有任何通知。正式发布总是要求从旧版本进行转储/重载。因此跳过这一步是很糟糕的主意。
   还有，不要使用 v6.0 的 pg_dumpall 脚本，否则所有对象都将归 Postgres 超级用户所有。
   输入（gunzip 一行与后面一行作为一行输入）：

   ```
   cd
   gunzip -c postgresql-v6.3.tar.gz |
   tar xvf - src/bin/pg_dump/pg_dumpall
   chmod a+x src/bin/pg_dump/pg_dumpall
   src/bin/pg_dump/pg_dumpall > db.out
   rm -rf src
   ```

   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，否则不要这样做。

如果 pg_dumpall 命令似乎花了很长时间而你觉得它可能已经死掉，那么可以从另一个终端多次使用 "ls -l db.out" 看文件的大小是否在增长。

请注意，如果你是从 Postgres95 v1.09 之前的版本升级，那么你必须备份数据库、安装 Postgres95 v1.09、恢复数据库，然后再次备份。你还应当阅读文件 /usr/src/pgsql/migration/*。

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。

7. 如果你是在升级现有系统，则终止 postmaster。输入

```
ps -ax | grep postmaster
```

这会列出若干进程的进程号。输入下面这行，其中 "???" 替换为进程 "postmaster" 的进程 id。（不要使用进程 "grep postmaster" 的 id。）输入 kill ??? 其中 "???" 按上述方式替换。

8. 如果你是在升级现有系统，则把旧目录移开。如果磁盘空间紧张，
你可能得备份并删除这些目录。如果这样做，请把旧数据库保存在 /usr/local/pgsql/data 目录树中。至少要保存文件 /usr/local/pgsql/data/pg_hba.conf。

输入以下命令：su cd /usr/src mv pgsql pgsql_6_0 cd /usr/local mv pgsql pgsql_6_0 exit

如果你不使用 /usr/local/pgsql/data 作为数据目录（检查环境变量 PGDATA 是否被设置成了别的值），那么你也应该以同样的方式移动该目录。

9. 创建新的源代码目录和安装目录。你的安装中实际路径可以不同；
但在整个过程中要保持一致。输入

```
su
cd /usr/src
mkdir pgsql
chown postgres:postgres pgsql
cd /usr/local
mkdir pgsql
chown postgres:postgres pgsql
exit
```

10. 解压并解开新的源文件。输入

```
cd /usr/src/pgsql
gunzip -c ~/postgresql-v6.3.tar.gz | tar xvf -
```

11. 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源代码路径和安装路径（见下面的 --prefix 选项）。输入

```
cd /usr/src/pgsql/src
./configure
```

configure 程序会列出可用的模板文件并让你选择一个。很多时候会为你选好合适的模板文件，你只需按回车接受默认值。如果默认值不合适，则输入合适的模板文件名并按回车。
（如果你这样做了，请发电子邮件到 scrappy@hub.org，说明程序 './config.guess' 的输出以及模板文件应该是什么。）

选定模板文件后，你会被问到若干关于你的具体配置的问题。在上面的 configure 命令后面加上参数就可以跳过这些提问。下面的参数可以加在 configure 命令的末尾：

`--prefix=BASEDIR` 为 Postgres 配置的安装选择一个不同的基目录。默认为 /usr/local/pgsql。

`--enable-hba` 启用基于主机的认证（默认）

`--disable-hba` 禁用基于主机的认证

`--enable-locale` 启用 USE_LOCALE

`--disable-locale` 禁用 USE_LOCALE（默认）

`--enable-cassert` 启用 ASSERT_CHECKING

`--disable-cassert` 禁用 ASSERT_CHECKING（默认）

`--with-template=TEMPLATE` 使用模板文件 TEMPLATE —— 模板文件假定在目录 src/template 中，请到那里查找合适的值。（如果 configure 脚本找不到指定的模板文件，它会向你询问一个。）

`--with-pgport=PORT` 设置 postmaster 进程监听传入连接的端口。默认端口为 5432。

举个例子，下面是我在 Sparc Solaris 2.5 系统上使用的 configure 脚本，安装基目录为 /opt/postgres。

```
    ./configure --prefix=/opt/postgres \
 --with-template=sparc_solaris-gcc --with-pgport=5432 \
 --enable-hba --disable-locale
```

当然，在真实的 shell 中，你会把这三行都写在同一行上。

12. 编译程序。输入

```
cd /usr/src/pgsql/src
gmake all >& make.log &
tail -f make.log
```

但愿最后显示的一行是 "All of PostgreSQL is successfully made. Ready to install."。这时（或者更早，随你愿意）输入 control-C 退出 tail。（如果以后遇到问题，你可以检查文件 make.log 中的警告和错误消息。）

如果你的计算机没有 gmake（GNU make），那么在本说明的其余部分试着改用 make。

请注意，你可能会在 make.log 中发现许多警告消息。除非以后遇到问题，这些消息可以放心地忽略。

如果编译器失败并给出找不到 flex 命令的错误，则按前述方法安装 flex。然后回到本目录，输入 "make clean"，再重新编译。

13. 安装程序。输入

```
cd /usr/src/pgsql/src
gmake install >& make.install.log &
tail -f make.install.log
```

最后显示的一行将是 "gmake[1]: Leaving directory `/usr/src/pgsql/src/man'"。这时（或者更早，随你愿意）输入 control-C 退出 tail。

14. 如有必要，告诉 UNIX 如何找到你的共享库。如果你使用的是 Linux-ELF，做下面之一，最好是第一种：

    a. (可选的) 以 root 身份编辑文件 /etc/ld.so.conf。向该文件加入一行 `/usr/local/pgsql/lib` 然后运行命令 `/sbin/ldconfig`。

    b. (可选的) 在 bash shell 中，输入

    ```
    export LD_LIBRARY_PATH=/usr/local/pgsql/lib
    ```

    c. (可选的) 在 csh shell 中，输入

    ```
    setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
    ```

    请注意，上述命令在不同的操作系统之间可能差异极大。请查阅平台特定说明，例如 Ultrix4.x 和非 ELF Linux 的那些。

    如果你创建数据库时收到消息 "pg_id: can't load library 'libpq.so'"，那么上面的步骤就是必要的。照做之后，再试着创建数据库。

15. 如果尚未做过，则为使用 Postgres 准备账号 postgres。任何要使用 Postgres 的账号都必须同样地准备。（下面的说明针对 bash shell。对其他 shell 请相应调整。）

    把下面几行加入你的登录 shell 的 ~/.bash_profile：

    ```
    PATH=$PATH:/usr/local/pgsql/bin
    MANPATH=$MANPATH:/usr/local/pgsql/man
    PGLIB=/usr/local/pgsql/lib
    PGDATA=/usr/local/pgsql/data
    export PATH MANPATH PGLIB PGDATA
    ```

    在继续余下步骤之前，确保你已经定义了这些变量。最简单的做法是输入：

    ```
    source ~/.bash_profile
    ```

16. 创建数据库。不要以 root 身份做下面这件事！那会是一个重大的安全漏洞。输入

    ```
    initdb
    ```

17. 设置访问数据库系统的权限。编辑文件 /usr/local/pgsql/data/pg_hba.conf 来完成。说明包含在该文件中。（如果你的数据库不在默认位置，即如果 PGDATA 被设置为指向别处，那么该文件的位置也会相应改变。）完成后应把该文件重新设为只读。如果你是从 v6.0 升级，可以把旧数据库中的 pg_hba.conf 文件复制到新数据库的上面，而不必从头重做。

18. 你可以选择跳过回归测试。不过，我们认为跳过测试是个坏主意！

    文件 /usr/src/pgsql/src/test/regress/README 有运行和解释回归测试的详细说明。这里给出一个简短的版本：

    在后台启动 postmaster 守护进程，输入

```
cd
nohup postmaster > regress.log 2>&1 &
```

以你的 Postgres 超级用户账号（通常是账号 postgres）运行 postmaster。不要从 root 账号运行 postmaster。

19. 运行回归测试。输入

```
cd
cd /usr/src/pgsql/src/test/regress
gmake clean
gmake all runtest
```

如果这是你第一次运行测试，不需要输入 "gmake clean"。

你应该在屏幕上（同时也写入文件 ./regress.out）看到一系列语句，说明哪些测试通过、哪些测试失败。请注意，某些测试"失败"可能是正常的。对于失败的测试，用 diff 比较目录 ./results 和 ./expected 中的文件。如果 float8 失败了，输入类似这样的命令：

```
cd /usr/src/pgsql/src/test/regress
diff -w expected/float8.out results
```

"失败"的测试可能是由于错误消息措辞略有不同、输出格式不同、
未能为你的平台正确设置时区等原因而失败的。这类"失败"并不表示 Postgres 有问题。

对于 i686/Linux-ELF 平台，没有测试失败，因为它是 v6.3 回归测试的参考平台。

对于 SPARC/Linux-ELF 平台，使用 Postgres v6.2 的 970525 beta 版时下列测试"失败"：float8 和 geometry 因浮点数的微小精度差异而"失败"。select_views 产生大量不同的输出，但差异源于微小的浮点差异。

结论？如果你确实看到了失败，试着理解这些差异的性质，
然后判断这些差异是否会影响你对 Postgres 的预期用途。不过请记住，这很可能是迄今为止最稳固的 Postgres 版本，包含了来自 v6.2.1 的许多错误修复，而且以前版本的 Postgres 已经被成功地使用了一段时间。

运行测试之后，输入

```
destroydb regression
cd /usr/src/pgsql/src/test/regress
gmake clean
```

20. 按步骤 7 所述停止 postmaster。然后把时区恢复为正常设置。如果你是通过修改环境变量 TZ 来更改时区的，那么一种做法是从账号 postgres 注销后再重新登录。

21. 启动 postmaster 守护进程。输入

```
cd
nohup postmaster > server.log 2>&1 &
```

以你的 Postgres 超级用户账号（通常是账号 postgres）运行 postmaster。不要从 root 账号运行 postmaster。

22. 如果你还没有这样做，现在是修改你的计算机的好时机，
使其在你每次启动计算机时自动启动 postmaster。下面是一些关于如何做到这一点的建议，由各位用户贡献。无论你怎么做，postmaster 都必须由用户 postgres 而不是 root 运行。

这就是下面所有示例都先切换用户（su）到 postgres 的原因。这些命令还考虑到了 PATH 和 PGDATA 等环境变量可能没有被正确设置的事实。示例如下。请格外谨慎地使用它们。a) 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" b) 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。注意：与大多数其他例子不同，这个例子经过测试。c) 在 RedHat v4.0 Linux 中编辑文件 /etc/inittab，使其包含下面这一行：pg:2345:respawn:/bin/su - postgres -c "/usr/local/pgsql/bin/postmaster -D/usr/local/pgsql/data >> /usr/local/pgsql/server.log 2>&1" /dev/null（这个例子的作者说该例会在 postmaster 死掉时复活它，但他不知道是否有其他副作用。）d) Postgres 发行版的 contrib/linux 区域有一个示例 init.d 脚本，它与近期的 RedHat 软件包兼容并经过测试。

23. 如果你还没有这样做，现在是修改你的计算机进行定期维护的好时机。
下面的事情应当定期进行：a) 运行 SQL 命令 vacuum。这会清理你的数据库。b) 备份你的系统。（你或许应该保留最近几次的备份。）最好在此期间没有其他人使用系统。理想情况下，上述任务应该由一个 shell 脚本完成，由 cron 每夜或每周运行。可以看看 crontab 的手册页作为入门。（如果你这样做了，请把你的 shell 脚本用电子邮件发给我们一份。我们也想在自己的系统上这样做。）

24. 如果你是在升级现有系统，则装入你的旧数据库。输入

```
cd
psql -e template1 < db.out
```

如果你的 pre-v6.2 数据库使用了 path 或 polygon 几何数据类型，那么你需要升级包含这些类型的列。为此，输入（在 psql 内）

```
update YourTable set PathCol = UpgradePath(PathCol);
update YourTable set PolyCol = UpgradePoly(PolyCol);
...
vacuum;
```

UpgradePath() 检查一个 path 值是否与旧语法一致，不会更新未通过该检查的列。UpgradePoly() 无法验证一个 polygon 是否确实来自旧语法，但提供了 RevertPoly() 来撤销一次误用的升级。

25. 如果你是新用户，可以像下面描述的那样玩一玩 Postgres。

26. 清理你留下的东西。输入

```
rm -rf /usr/src/pgsql_6_0
rm -rf /usr/local/pgsql_6_0
# Also delete old database directory tree if it is not in
#  /usr/local/pgsql_6_0/data
rm ~/postgresql-v6.2.1.tar.gz
```

27. 你可能想打印文档。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。alias gshp='gs -sDEVICE=laserjet -r300 -dNOPAUSE' export GS_LIB=/usr/share/ghostscript:/usr/share/ghostscript/fonts # Print out the man pages. man -a -t /usr/local/pgsql/man/*/* > manpage.ps gshp -sOUTPUTFILE=manpage.hp manpage.ps rm manpage.ps lpr -l -s -r manpage.hp # Print out the Postgres95 User Manual,

version 1.0, # Sept. 5, 1996. cd /usr/src/pgsql/doc gshp -sOUTPUTFILE=userguide.hp userguide.ps lpr -l -s -r userguide.hp 如果你是开发者，你可能还想打印 Postgres Implemention Guide（实现指南）1.0 版（1995 年 10 月 1 日）。这是一个 WWW 文档，位于 http://www.postgresql.org/docs/impguide。

28. Postgres 团队希望让 Postgres 在所有受支持的平台上都能工作。因此我们请你告诉我们你在你的系统上有没有让 Postgres 跑起来。请发邮件到 pgsql-ports@postgresql.org 告诉我们以下内容：- Postgres 的版本（v6.2.1、6.1.1、beta 970703 等）。- 你的操作系统（例如 RedHat v4.0 Linux v2.0.26）。- 你的硬件（SPARC、i486 等）。- 你是否干净地编译、安装并运行了回归测试？如果没有，你改了哪些源代码（即你应用的补丁、你做的修改等）、哪些测试失败了等。编译时出现许多警告是正常的。
这些不需要报告。

29. 现在按需要创建、访问和操纵数据库。编写访问数据库服务器的客户端程序。换句话说，尽情享受！

18.3. 玩转 Postgres

在 Postgres 安装完成、数据库系统创建好、postmaster 守护进程在运行、回归测试通过之后，你会想看看 Postgres 做点什么。这很容易。调用 Postgres 的交互界面 psql：

```
% psql template1
```

（psql 必须打开一个特定的数据库，但此时唯一存在的是 template1 数据库，它总是存在的。我们连接到它只是为了创建另一个数据库然后切换过去。）

psql 的响应是：

```
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

template1=>
```

创建数据库 foo：

```
template1=> create database foo;
CREATEDB
```

（养成写上 SQL 分号的习惯。在看到分号或 "\g" 之前 psql 不会执行任何东西，而且分号是分隔多条语句所必需的。）

现在连接到新数据库：

```
template1=> \c foo
connecting to new database: foo
```

（"斜杠"命令不是 SQL，所以不用分号。用 \? 可以看到所有斜杠命令。）

再创建一个表：

```
foo=> create table bar (i int4, c char(16));
```

```
CREATE
```

然后查看新表：

```
foo=> \d bar

Table    = bar
+----------------------------------+----------------------------------
+-------+
|               Field              |               Type
 | Length|
+----------------------------------+----------------------------------
+-------+
| i                                | int4
 |      4 |
| c                                | (bp)char
 |     16 |
+----------------------------------+----------------------------------
+-------+
```

依此类推。你明白意思了。

18.4. 下一步

有问题？Bug？反馈？首先阅读目录 /usr/src/pgsql/doc 中的文件。该目录中的 FAQ 可能特别有用。

如果 Postgres 在你的计算机上编译失败，那么填写文件 /usr/src/pgsql/doc/bug.template 中的表单，并邮寄到表单顶部指示的地址。

把问题邮寄到 pgsql-questions@postgresql.org。关于各种邮件列表的更多信息，见 http://www.postgresql.org 并查找邮件列表。

18.5. 移植说明

注意

对于某些移植，这些说明可能已经过时。

18.5.1. Ultrix4.x

你需要安装 libdl-1.1 包，因为 Ultrix 4.x 没有动态加载器。它可从 s2k-ftp.CS.Berkeley.EDU:pub/personal/andrew/libdl-1.1.tar.Z 获取

18.5.2. Linux

18.5.2.1. Linux ELF

Thomas G. Lockhart

回归测试参考机器是一台运行 linux-2.0.30/libc-5.3.12/RedHat-4.2 的双处理器 i686。linux-elf 移植可以干净地安装。更多细节见 Linux FAQ。

18.5.2.2. Linux a.out

对于非 ELF 的 Linux，必须获取 dld 库并安装到系统上。它为 Postgres 移植启用动态链接加载能力。dld 库可以从 sunsite linux 发行版获取。当前名称是 dld-3.2.5。Jalon Q. Zimmerman[2]

18.5.3. BSD/OS

对于 BSD/OS 2.0 和 2.01，你需要获取 GNU dld 库。

18.5.4. NeXT

v1.09 的 NeXT 移植由 Tom R. Hageman[3] 提供。它需要一个 SysV IPC 模拟库以及用于共享库和信号量功能的头文件。Tom 恰好销售这样的产品，可以联系他获取信息。他还表示将向公众提供 Postgres 的 NEXTSTEP 二进制发行版。联系 Info@RnA.nl 获取信息。

我们没有关于 NeXT 成功安装的最新报告（针对 v6.2.1）。不过，即使后端不受支持，客户端库也应该能工作。

[2] sneaker@powergrid.electriciti.com
[3] mailto:tom@basil.icce.rug.nl

第 19 章 运行时环境

图 19.1. Postgres 文件布局

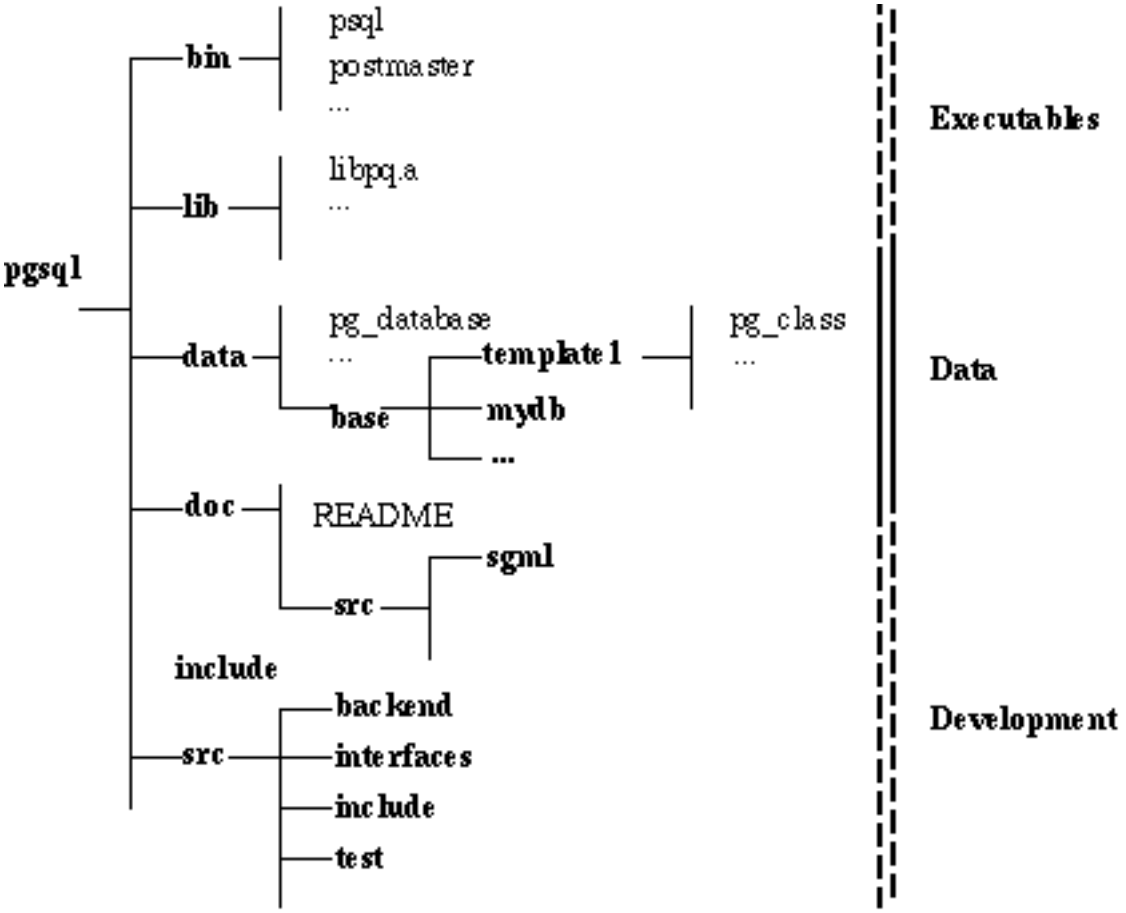


图 19.1展示了以默认方式安装时 Postgres 发行版的布局。为简单起见，我们假设 Postgres 安装在目录 `/usr/local/pgsql` 中。因此，无论在哪里看到目录 `/usr/local/pgsql`，你都应该用 Postgres 实际安装的目录名来替换它。所有 Postgres 命令都安装在目录 `/usr/local/pgsql/bin` 中。因此，你应当把这个目录加到你的 shell 命令路径中。如果你使用 Berkeley C shell 的一个变体，例如 csh 或 tcsh，你可以把

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

加到你主目录的 .login 文件中。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
```

加到你主目录的 .profile 文件中。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，在本文档中我们会频繁提到"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。PGPORT 环境变量也可能必须设置。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。

19.1. 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，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。在寻找可能的变通方法之后，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 工作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，就得到了我想要的东西！我只做了 LC_CTYPE 和 LC_COLLATE 的支持，但后来 LC_MONETARY 由其他人添加了。我收到了很多人的关于这个补丁的消息，于是我决定把它发给开发者，而（让我惊讶的是）它被并入了 postgresql 发行版。

人们经常抱怨 locale 对他们不工作。有几个常见的错误：

- 没有在编译前正确配置 postgresql。你必须用 --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。你必须在运行 postmaster 之前定义环境变量 $LC_CTYPE 和 $LC_COLLATE，因为后端从环境获取区域信息。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 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 的区域支持改过几次，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").

- locale 文件的位置错误！可能的位置：`/usr/lib/locale`（Linux、Solaris）、`/usr/share/locale`（Linux）、`/usr/lib/nls/loc`（DUX 4.0）查 man locale 找正确的位置。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 locale。

19.1.1. 好处是什么？

你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 locale，只需取消定义 USE_LOCALE 变量。

[1] http://www.sai.msu.su/~megera/postgres/

19.1.2. 坏处是什么？

使用 locale 有一个明显的坏处——速度！所以，只有在你真正需要时才使用 locale。

第 20 章 启动 postmaster

除非 postmaster 进程在运行，否则数据库什么也做不了。作为站点管理员，在启动 postmaster 之前，有一些事情你应当记住。这些内容在本手册题为"Administering Postgres"的章节中讨论。不过，如果 Postgres 是严格按照安装说明安装的，下面的简单命令就是你启动 postmaster 所需的全部：

```
% postmaster
```

postmaster 偶尔会打印出一些在排障时往往有帮助的消息。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 &
```

如果你不希望看到这些消息，可以输入

```
% postmaster -S
```

这样 postmaster 就是"S"ilent（静默）的。注意最后一个例子的末尾没有 and 符（"&"）。

第 21 章 添加和删除用户

createuser 使特定的用户能够访问 Postgres。destroyuser 移除用户并阻止他们访问 Postgres。注意，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 22 章 磁盘管理

22.1. 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。

可以指定一个绝对路径名或者一个环境变量作为位置。注意，出于安全和完整性原因，
所有这样指定的路径和环境变量都会被追加一些额外的路径字段。

> **注意**
>
> 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。

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，输入

```
% setenv PGDATA2 /home/postgres/data
```

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。

要在 `/home/postgres/data` 中创建一个数据存储区，请确保 `/home/postgres` 已经存在并且可写。然后在命令行输入

```
% 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
% destroydb test
```

第 23 章 排障

假设你的站点管理员已经正确地启动了 postmaster 进程并且授权你使用数据库，你（作为用户）就可以开始启动应用程序了。如前所述，你应当把 /usr/local/pgsql/bin 加到你的 shell 搜索路径中。在大多数情况下，这就是你在准备方面要做的全部。

如果你从一个 Postgres 命令（例如 psql 或 createdb）得到下列错误消息：

```
connectDB() failed: Is the postmaster running at 'localhost' on port
 '4322'?
```

这通常要么是因为 postmaster 没有在运行，要么是你试图连接到了错误的服务器主机。如果你得到下列错误消息：

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

则意味着站点管理员以错误的用户启动了 postmaster。让他以 Postgres 超级用户身份重启它。

第 24 章 管理数据库

现在 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。

24.1. 创建数据库

假设你想创建一个名为 mydb 的数据库。你可以使用下面的命令完成：

```
% createdb mydb
```

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 16 个字符。并非每个用户都有权成为数据库管理员。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。

24.2. 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（monitor 或 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。这一接口在第 ?? 节中有进一步讨论。

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 the Postgres interactive sql monitor:

  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: mydb

mydb=>
```

这个提示表明终端监视程序正在监听你，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 Postgres 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，backslash-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，在监视程序提示符下输入 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“--”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

24.3. 删除数据库

如果你是数据库 mydb 的数据库管理员，你可以使用下面的 UNIX 命令销毁它：

```
% destroydb 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 25 章 数据库恢复

本节尚待编写。有人愿意吗？

第 26 章 回归测试

回归测试的说明与分析。

PostgreSQL 回归测试是对由 Jolly Chen 和 Andrew Yu 开发的、嵌入在 PostgreSQL 中的 SQL 实现的一套全面测试。它既测试标准 SQL 操作，也测试 PostgreSQL 的扩展能力。

这些测试最近由 Marc Fournier 和 Thomas Lockhart 修订，现在被打包为功能单元，这应当使它们更易于运行、更易于解读。从 PostgreSQL v6.1 起，回归测试对每个正式发行版都是最新的。

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。这些测试目前是用一种简单的 "diff" 算法来评估的，对细微的系统差异很敏感。对于表面上失败的测试，检查差异可能会发现这些差异并不重要。

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）。

26.1. 回归测试环境

回归测试由 `make` 命令调用，该命令把一个 C 程序编译成当前目录中的一个共享库。当前目录中还会创建本地化的 shell 脚本。输出文件模板被整理成 `./expected/*.out` 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。

通常，回归测试应当以 pg_superuser 运行，因为 'src/test/regress' 目录及其子目录为 pg_superuser 所有。如果你以其他用户运行回归测试，那么 'src/test/regress' 目录树应当可被该用户写入。

postmaster 应当在系统时区设置为加利福尼亚州伯克利的情况下调用。回归测试脚本会自动完成这项设置。但是，它确实要求机器支持 PST8PDT 时区。

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
```

26.2. 目录布局

> **注意**
>
> 这里以后应当变成上一节中的一个表。

```
input/ .... .source files that are converted using 'make all' into
            some of the .sql files in the 'sql' subdirectory
```

```
 output/ ... .source files that are converted using 'make all' into
             .out files in the 'expected' subdirectory

 sql/ ...... .sql files used to perform the regression tests

 expected/ . .out files that represent what we *expect* the results
to
             look like

 results/ .. .out files that represent what the results *actually*
look
             like. Also used as temporary storage for table copy
testing.
```

26.3. 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 回归测试配置

对于全新安装或从先前版本的 Postgres 升级：

1. 构建回归测试。键入

   ```
   cd /usr/src/pgsql/src/test/regress
   gmake all
   ```

2. (可选的) 如果你之前调用过回归测试，用以下命令清理工作目录：

   ```
   cd /usr/src/pgsql/src/test/regress
   make clean
   ```

3. 文件 /usr/src/pgsql/src/test/regress/README 包含关于运行和解读回归测试的详细说明。这里给出一个简短的版本：

   如果 postmaster 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 &
   ```

   从你的 Postgres 超级用户账户（通常是 postgres 账户）运行 postmaster。

   注意

   不要从 root 账户运行 `postmaster`。

4. 运行回归测试。键入

```
cd /usr/src/pgsql/src/test/regress
gmake runtest
```

   如果这是你第一次运行测试，不需要键入 "gmake clean"。

5. 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，某些测试"失败"可能是正常的。对于失败的测试，用 diff 比较 ./results 和 ./expected 目录中的文件。如果 float8 失败了，键入类似下面的命令：

```
cd /usr/src/pgsql/src/test/regress
diff -w expected/float8.out results
```

6. 运行测试后，键入

```
destroydb regression
cd /usr/src/pgsql/src/test/regress
gmake clean
```

26.4. 回归分析

“失败” 的测试可能是由于错误消息、数学库或输出格式的细微差异而失败。这类"失败"并不表示 Postgres 有问题。

对于 i686/Linux-ELF 平台，没有测试失败，因为它是 v6.2.1 回归测试的参考平台。

对于 SPARC/Linux-ELF 平台，使用 Postgres v6.2 的 970525 beta 版时下列测试"失败"：float8 和 geometry 因浮点数的微小精度差异而"失败"。select_views 产生大量不同的输出，但差异源于微小的浮点差异。

结论是什么？如果你确实看到失败，请设法理解这些差异的本质，
然后判断这些差异是否会影响你对 Postgres 的预期使用。不过请记住，这很可能是迄今为止最坚实的 Postgres 发行版，它合并了来自 v6.1 的许多缺陷修复，而且先前版本的 Postgres 已经成功使用了一段时间。

26.4.1. 比较期望/实际输出

结果位于 ./results 目录中的文件里。这些结果可以用 'diff' 与 ./expected 目录中的结果进行比较。这些文件可能不会完全一致。下面几段试图解释这些差异。

26.4.2. 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。

26.4.3. OID 差异

PostgreSQL OID（对象标识符）出现在 'regress.out' 中的若干地方。OID 是唯一的 32 位整数，由 PostgreSQL 后端在每次插入或更新表行时生成。如果你在一个非全新的数据库上运行回归测试，或者多次运行它，报告的 OID 将具有不同的值。'misc.out' 中的下列 SQL 语句表现出了这种行为：QUERY: SELECT user_relns() AS user_relns ORDER BY user_relns; 'a,523676' 行就是由一个 OID 组成的。

26.4.4. 日期和时间差异

在许多受支持的平台上，你可以让 PostgreSQL 认为自己运行在与加利福尼亚州伯克利相同的时区中。详见关于如何运行回归测试的那一节。如果你不把时区环境显式设置为 PST8PDT，那么大多数日期和时间结果将反映你的本地时区，并且无法通过回归测试。
有些系统似乎不接受显式设置本地时区规则的推荐语法。某些使用公共域时区包的系统在 1970 年前的 PDT 时间上表现出小问题，把它们表示为 PST。

26.4.5. 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。这些差异出现在同一平台上使用不同操作系统的情况，即 Intel/86 上的 BSDI 和 SOLARIS；以及同一操作系统用于不同平台的情况，即 SPARC 和 Intel/86 上的 SOLARIS。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。

26.4.6. 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;
```

26.4.7. 随机差异

random.out 中至少有一个测试用例旨在产生随机结果。这会使 random 回归测试失败。键入

```
diff results/random.out expected/random.out
```

因此应当只产生一行或几行差异，但不同体系结构上的其他浮点差异可能造成多得多的差异。
见下面的发行说明。

26.4.8. “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。（见下面）原始的 `sample.regress.out` 文件来自 Jolly Chen 构建的 postgres-1.01 发行版，这里收录仅供参考。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。

第 27 章 发行说明

> **注意**
>
> 应包含 `migration/` 中的迁移说明。

发行说明尚未整合到新文档中。请查看发行目录树顶层的纯文本文件以及 `migration/` 目录以获取最新信息。

27.1. 版本 6.3

待定

27.2. 版本 6.2.1

> **注意**
>
> v6.2.1 是针对 v6.2 的缺陷修复和易用性发行版。只需少量说明。

27.3. 版本 6.2

> **注意**
>
> 此处应包含基于 Bruce 的发行总结的信息。

27.4. 版本 6.1

> **注意**
>
> 此处应包含基于 Bruce 的发行总结的信息。

回归测试已经为 PostgreSQL v6.1 发行版做了改编和大量修改。

三种新数据类型（datetime、timespan 和 circle）已添加到 PostgreSQL 的原生类型集中。点、框、路径和多边形的输出格式已在各数据类型间保持一致。misc.out 中的多边形输出只是相对于原始回归输出做了抽查以确认正确性。

PostgreSQL v6.1 引入了一个新的备选优化器，它使用遗传算法。当查询包含多个限定条件或多个表时（让优化器可以选择求值顺序），这些算法会给查询结果的排序引入随机行为。
若干回归测试已被修改为显式排序结果，因而对优化器的选择不敏感。
还有一些回归测试针对本身无序的数据类型（例如点和时间区间），
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）
的解释似乎在原始回归测试生成之后的某个时候发生了变化。当前的 `./expected/*.out` 文件反映的是这种新解释，而它可能并不正确！

float8 回归测试在至少某些平台上会失败。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所致。

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。

27.5. 计时结果

这些计时结果来自使用以下命令运行回归测试

```
% 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计时的不确定性所致。

27.5.1. 版本 6.3

```
  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
```

27.5.2. 版本 6.1

```
  时间    系统
  06:12  Pentium Pro 180, 32MB, 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 IV. 程序员指南

关于扩展 Postgres 的信息。

目录

第 28 章 引言

本文档是 PostgreSQL[1] 数据库管理系统的程序员手册，该系统最初是在加州大学伯克利分校开发的。PostgreSQL 基于 Postgres release 4.2[2]。由 Michael Stonebraker 教授领导的 Postgres 项目曾获得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 公司的资助。

本手册的第一部分解释 Postgres 的可扩展性方式，并描述用户如何通过添加用户定义的类型、操作符、聚合以及查询语言和编程语言函数来扩展 Postgres。在对 Postgres 规则系统做一个极为简短的概述之后，我们讨论触发器和 SPI 接口。本手册最后详细描述各种语言的编程接口和支持库。

我们假定读者熟悉 UNIX 和 C 编程。

28.1. 版权和商标

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

第 29 章 体系结构

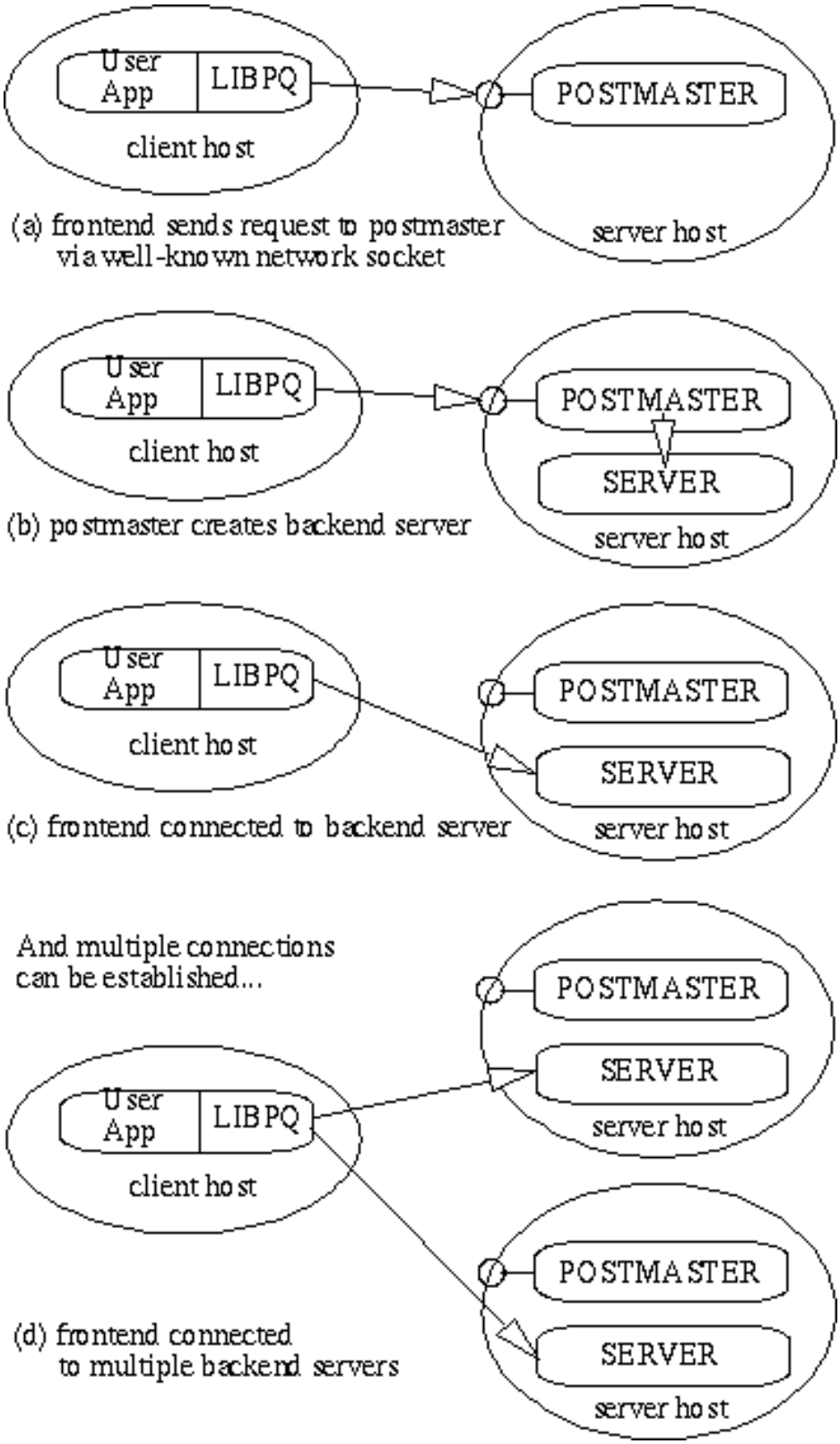
29.1. Postgres 体系结构概念

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 29.1(a))，后者再启动一个新的后端服务器进程 (图 29.1(b))

图 29.1. 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
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

并把前端进程连接到新服务器 (图 29.1(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 30 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

30.1. 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。

30.2. 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。

30.3. 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 30.1. 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 30.1. Postgres 的主要系统目录

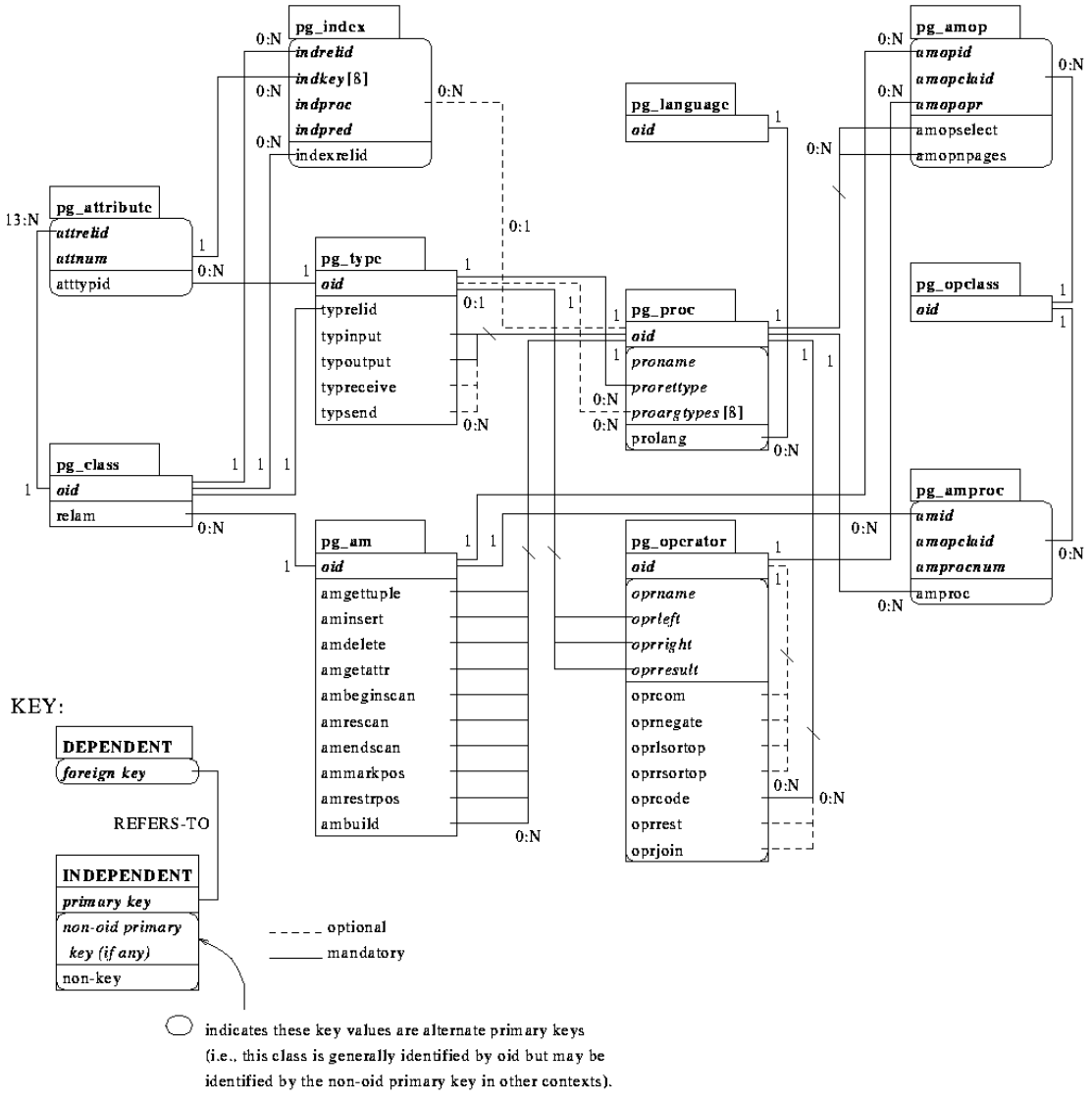


Figure 3. The major POSTGRES system catalogs.

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 30.1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 31 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 提供两类函数：查询语言函数（用 SQL 编写的函数）和编程语言函数（用 C 等编译型编程语言编写的函数）。每类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，两类函数都可以返回基本类型或复合类型。定义 SQL 函数更容易，所以我们从它开始。本节的示例也可以在 `funcs.sql` 和 `C-code/funcs.c` 中找到。

31.1. 查询语言（SQL）函数

31.1.1. 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      |
+-------+
```

31.1.2. 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.dept = 'toy';

     +-----+-------+
     |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,
  \'none\'::char16 AS dept;'
  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：

```
            WARN::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;
WARN: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 |
     +--+
```

31.2. 编程语言函数

31.2.1. 基本类型上的编程语言函数

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 char16 类型的示例实现：

```
/* 16-byte structure, passed by reference */
typedef struct {
    char data[16];
} char16;
```

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"
#include "utils/palloc.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"  /* for char16, etc. */
#include "utils/palloc.h" /* for palloc */
int
add_one(int arg)
{
    return(arg + 1);
}
char16 *
concat16(char16 *arg1, char16 *arg2)
{
    char16 *new_c16 = (char16 *) palloc(sizeof(char16));
    memset((void *) new_c16, 0, sizeof(char16));
    (void) strncpy(new_c16, arg1, 16);
    return (char16 *)(strncat(new_c16, arg2, 16));
}
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);
}
```

在 OSF/1 上我们会输入：

```
CREATE FUNCTION add_one(int4) RETURNS int4
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat16(char16, char16) RETURNS char16
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/obj/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）。

31.2.2. 复合类型上的编程语言函数

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 提供了一个从 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"  /* for char16, etc. */
#include "libpq-fe.h" /* for TUPLE */
bool
c_overpaid(TUPLE 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 char16 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。

31.2.3. 注意事项

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。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，因此通常也包含该文件是个好主意。

- 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，参见附录 A 的详细说明。

第 32 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。

32.1. 用户定义的类型

32.1.1. 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(WARN, "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。

32.1.2. 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，你将需要使用
Postgres的大对象接口。

第 33 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，即可用不同的参数数量和类型重复使用。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。
要创建一个把两个复数相加的操作符，可以按下面的方法做。
首先我们需要创建一个对新类型做加法的函数，然后就可以用该函数创建操作符。

```
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 = +
);
```

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。只要我们给系统足够的类型信息，
它就能自动确定该用哪些操作符。

```
SELECT (a + b) AS c FROM test_complex;

+----------------+
|c               |
+----------------+
|(5.2,6.05)      |
+----------------+
|(133.42,144.95) |
+----------------+
```

第 34 章 扩展 SQL：聚合

Postgres 中的聚合是按照状态转换函数来表示的。也就是说，聚合可以按照每当处理一个实例就被修改的状态来定义。有些状态函数在计算新状态时会查看实例中的特定值（即 create aggregate 语法中的 sfunc1），而另一些只跟踪自己的内部状态（sfunc2）。如果我们定义一个只使用 sfunc1 的聚合，定义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。我们将使用内置在 Postgres 中的 int4pl 来执行这个加法。

```
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)   |
+------------+
```

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')

SELECT my_count(*) as emp_count from EMP;

+-----------+
|emp_count |
+-----------+
|5         |
+-----------+
```

"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        |
+------------+
```

第 35 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。因此，
我们不会在这里试图解释 Postgres 规则系统的实际语法和运作方式，而是建议你阅读[ston90b]来理解其中一些要点以及 Postgres 规则系统的理论基础，然后再尝试使用规则。本节的讨论旨在提供 Postgres 规则系统的概览，并为用户指出有用的参考资料和示例。“查询重写”规则系统会修改查询，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[ong90]以及[ston90b]中有讨论。

第 36 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 30.1。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 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，以便按绝对值升序排序整数。

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 36.1. 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
amkind	当前未使用，但保留为占位符并置为 'o'
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）
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 |
         +----+
```

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 36.2. 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。该属性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。

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名称与一个 oid 相关联。在 pg_amop 中，每个 B-tree 操作符类都有上文的第 1 到第 5 号一组过程。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 是其他类中的外键。

```
INSERT INTO pg_opclass (opcname) VALUES ('complex_abs_ops');

SELECT oid, opcname
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';

        +------+--------------+
        |oid   | opcname      |
        +------+--------------+
        |17314 | int4_abs_ops |
        +------+--------------+
```

注意，你的 pg_opclass 实例的 oid 会不同！在本讨论中凡出现 17314 之处，你都应当用你得到的值替换。

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

代码的一部分看起来像这样：（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.sql。）

```
#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);
         }
```

下面有几点重要的事情正在发生。

首先，注意这里正在为 int4 定义小于、小于等于、等于、大于等于和大于操作符。所有这些操作符都已以 <、<=、=、>= 和 > 的名称为 int4 定义过。当然，新的操作符行为不同。为了确保 Postgres 使用这些新操作符而不是旧操作符，它们的命名必须与旧操作符不同。这是一个要点：你可以在 Postgres 中重载操作符，但仅当该操作符尚未为这些参数类型定义时才行。也就是说，如果你已经为 (int4, int4) 定义了 <，就不能再定义一次。Postgres 在你定义操作符时不检查这一点，所以要小心。为了避免这个问题，将为这些操作符使用奇怪的名字。如果你弄错了，访问方法在你尝试做扫描时很可能会崩溃。

另一个要点是所有操作符函数都返回布尔值。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。

```
CREATE FUNCTION complex_abs_eq(complex, complex)
              RETURNS bool
              AS 'PGROOT/tutorial/obj/complex.so'
              LANGUAGE 'c';
```

现在定义使用它们的操作符。如前所述，操作符名在所有取两个 int4 操作数的操作符中必须唯一。为了查看下面列出的操作符名是否已被占用，我们可以在 pg_operator 上做一次查询：

```
    /*
     * this query uses the regular expression operator (~)
     * to find three-character operator names that end in
     * the character &
     */
    SELECT *
     FROM pg_operator
     WHERE oprname ~ '^..&$'::text;
```

以查看你想要的名字是否已被用于你想要的类型。这里的重要内容是过程（即上面定义的 C 函数）以及限制和连接选择性函数。你应当直接使用下面所用的那些--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，

否则访问方法在试图使用该操作符时将会崩溃。你应当照抄 restrict 和 join 的名称，但使用你在上一步中定义的过程名。

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         )
```

注意，这里定义了对应于小于、小于等于、等于、大于和大于等于的五个操作符。

我们快完成了。最后要做的事情是更新 pg_amop 关系。为此，我们需要下列属性：

表 36.3. pg_amproc 模式

属性	描述
amopid	pg_am 中 B-tree 实例的 oid（== 403，见上文）
amopclaid	int4_abs_ops 的 pg_opclass 实例的 oid（== 你得到的那个替代 17314 的值，见上文）
amopopr	该操作符类的各操作符的 oid（我们马上就会取得）
amopselect, amopnpages	代价函数

代价函数供查询优化器用来决定在一次扫描中是否使用某个索引。幸运的是，
这些函数已经存在。我们将使用的两个函数是 btreesel（它估计 B-tree 的选择性）和 btreenpage（它估计一次搜索将在树中触及的页面数）。

所以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 int4 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';
```

```
+------+---------+
|oid   | oprname |
+------+---------+
|17321 | <       |
+------+---------+
|17322 | <=      |
+------+---------+
|17323 |  =      |
+------+---------+
|17324 | >=      |
+------+---------+
|17325 | >       |
+------+---------+
```

（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 17321 到 17325 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。我们可以查看操作符名称并挑出我们刚添加的那些。

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，操作符是按从小等于到大等于排序的。我们添加所需的实例：

```
INSERT INTO pg_amop (amopid, amopclaid,
 amopopr, amopstrategy,
 amopselect, amopnpages)
SELECT am.oid, opcl.oid, c.opoid, 3,
 'btreesel'::regproc, 'btreenpage'::regproc
FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
WHERE amname = 'btree'
 and opcname = 'complex_abs_ops'
 and c.oprname = '=';
```

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。

最后一步（终于！）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 int4_abs_cmp(int4, int4)
 RETURNS int4
 AS 'PGROOT/tutorial/obj/complex.so'
 LANGUAGE 'c';

SELECT oid, proname FROM pg_proc
 WHERE prname = 'int4_abs_cmp';

     +------+--------------+
     |oid   | proname      |
     +------+--------------+
     |17328 | int4_abs_cmp |
     +------+--------------+
```

（同样，你的 `oid` 号很可能会有所不同，你应当用你看到的值替换下文中的值。）回想 B-tree 实例的 oid 是 403 而 `int4_abs_ops` 的是 17314，我们可以像下面这样添加新实例：

```
INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
 VALUES ('403'::oid, -- btree oid
 '17314'::oid,       --  pg_opclass tuple
 '17328'::oid,       -- new pg_proc oid
 '1'::int2);
```

第 37 章 GiST Indices

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 38 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 linkedit。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 linkediting。注意，这一过程从 4.2 版开始已经改变。

> **提示**
>
> 旧的Postgres动态装载机制要求编写动态装载器的人深入了解可执行文件格式、
> 可执行指令在内存中的放置和对齐等方面的知识。那样的装载器往往又慢又有缺陷。从 4.2 版开始，Postgres的动态装载机制已被重写为使用操作系统提供的动态装载机制。这种做法通常比我们以前的动态装载机制更快、更可靠也更可移植。原因在于，
> 几乎所有现代版本的 UNIX 都使用动态装载机制来实现共享库，因此必然提供快速可靠的机制。另一方面，目标文件在被装载进 Postgres之前必须先做一点后处理。我们希望速度和可靠性上的大幅提升能弥补便利性上的轻微下降。

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，PGROOT/src/regress目录中的回归测试套件包含这一过程的若干可用示例。如果你照抄这些测试的做法，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。

38.1. ULTRIX

在 ULTRIX 下构建动态装载的目标文件非常容易。ULTRIX 没有任何 sharedlibrary 机制，因此对动态装载器接口没有任何限制。另一方面，我们不得不自己（重）
写一个不可移植的动态装载器，也无法使用真正的共享库。在 ULTRIX 下，唯一的限制是每个目标文件都必须用选项 -G 0 生成。（注意那是数字“0”而不是字母“O”。）例如，

```
# simple ULTRIX example
% cc -G 0 -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。

38.2. 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！

38.3. 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 标志产生所谓的“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
                   or
              % 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 39 章 触发器

虽然当前版本的 Postgres 有多种客户端接口，例如 Perl、Tcl、Python 和 C，它还缺少一种真正的过程语言（PL）。我们希望将来能有一种合适的 PL。在此期间，可以把 C 函数作为触发器动作调用。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。

39.1. 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 name> <BEFORE|AFTER> <INSERT|DELETE|UPDATE>
    ON <relation name> FOR EACH <ROW|STATEMENT>
    EXECUTE PROCEDURE <procedure name> (<function args>);
```

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。

下一个词决定函数是在事件之前还是之后被调用。

命令的下一个元素决定该函数将在哪些事件上触发。可以用 OR 分隔指定多个事件。

关系名决定该事件作用于哪个表。

FOR EACH 语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。

过程名是被调用的 C 函数。

参数通过 CurrentTriggerData 结构传给函数。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，函数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，它接受两个字段名作为参数，把当前用户写入其中一个字段，把当前时间戳写入另一个字段。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，触发的顺序是不可预测的。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）……

39.2. 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;
```

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(event) 如果触发器在之前（BEFORE）触发则返回 TRUE；
  TRIGGER_FIRED_AFTER(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
: 是一个指针，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
: 是一个指针，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
: 是一个指针，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

是一个指针，指向 Trigger 结构，它定义在 src/include/utils/rel.h 中：

```
typedef struct Trigger
{
 char  *tgname;
 Oid  tgfoid;
 func_ptr tgfunc;
 int16  tgtype;
 int16  tgnargs;
 int16  tgattr[8];
 char  **tgargs;
} Trigger;
```

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。

39.3. 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！

39.4. 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(WARN, "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(WARN, "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(WARN, "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：create table 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 40 章 服务器编程接口

Vadim Mikheev

服务器编程接口（SPI）是一次让用户能够在用户定义的 C 函数中运行 SQL 查询的尝试。鉴于当前版本的 Postgres 缺少一种合适的过程语言（PL），SPI 是编写服务器存储过程和触发器的唯一方法。将来，SPI 将被用作某种 PL 的“主力”。

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。

40.1. 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

XXX thomas 1997-12-24

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 — 执行由 SPI_saveplan 准备或返回的计划

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。

40.2. 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

40.3. 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！

40.4. 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。

40.5. 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 41 章 libpq

libpq 是 Postgres 的应用编程接口。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。本版本的文档描述 C 接口库。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 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。

41.1. 控制与初始化

下面的环境变量可用于设置默认的环境值，以避免把数据库名硬编码进应用程序：

- PGHOST 设置默认的服务器名。
- PGOPTIONS 设置 Postgres 后端的附加运行时选项。
- PGPORT 设置与 Postgres 后端通信所用的默认端口。
- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。
- PGDATABASE 设置默认的 Postgres 数据库名。
- 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。仅在启用了 Kerberos 认证时才会使用此环境变量。

41.2. 数据库连接函数

下面的例程用于从 C 程序建立与后端的连接。

- PQsetdbLogin 与后端建立一个新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                const char *pgport,
                const char *pgoptions,
                const char *pgtty,
                const char *dbName,
                const char *login,
                const char *pwd);
```

如果任何参数为 NULL，则检查相应的环境变量。如果环境变量也未设置，则使用硬编码的默认值。PQsetdbLogin 总是返回有效的 PGconn 指针。在通过连接发送查询之前，应当调用 PQstatus（见下文）确认连接已正确建立。libpq 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。

- PQsetdb 与后端建立一个新连接。

```
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()。

- PQconndefaults 返回连接选项。

  ```
  PQconninfoOption *PQconndefaults(void)

  struct PQconninfoOption
   {
    char   *keyword;   /* The keyword of the option */
    char   *environ;   /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;    /* Options value */
    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 options - don't
            create a field by default */
    int dispsize;  /* Field size in characters for dialog */
   };
  ```

  返回连接选项结构的地址。它可用来确定所有可用选项及其当前值。

- PQdb 返回连接的数据库名。

  ```
  char *PQdb(PGconn *conn)
  ```

- PQhost 返回连接的主机名。

  ```
  char *PQhost(PGconn *conn)
  ```

- PQoptions 返回连接中使用的 pgoptions。

  ```
  char *PQoptions(PGconn *conn)
  ```

- PQport 返回连接的 pgport。

  ```
  char *PQport(PGconn *conn)
  ```

- PQtty 返回连接的 pgtty。

  ```
  char *PQtty(PGconn *conn)
  ```

- PQstatus 返回连接的状态。状态可以是 CONNECTION_OK 或 CONNECTION_BAD。

  ```
  ConnStatusType *PQstatus(PGconn *conn)
  ```

- PQerrorMessage 返回与连接相关联的错误消息

  ```
  char *PQerrorMessage(PGconn* conn);
  ```

- PQfinish 关闭与后端的连接，同时释放 PGconn 结构使用的内存。调用过 PQfinish 之后，不应再使用该 PGconn 指针。

  ```
  void PQfinish(PGconn *conn)
  ```

- PQreset 重置与后端的通信端口。此函数将关闭到后端的 IPC 套接字连接，并尝试与同一个后端重新建立新连接。

  ```
  void PQreset(PGconn *conn)
  ```

- PQtrace 打开前端与后端之间所传消息的跟踪。这些消息会被回显到 debug_port 文件流。

  ```
  void PQtrace(PGconn *conn,
               FILE* debug_port);
  ```

- PQuntrace 关闭前端与后端之间所传消息的跟踪。

  ```
  void PQuntrace(PGconn *conn);
  ```

41.3. 查询执行函数

- PQexec 向 Postgres 提交一个查询。查询成功则返回 PGresult 指针，否则返回 NULL。如果返回了 NULL，可以用 PQerrorMessage 获取该错误的更多信息。

  ```
  PGresult *PQexec(PGconn *conn,
                   char *query);
  ```

  PGresult 结构封装了后端返回的查询结果。libpq 程序员应注意维护 PGresult 的抽象。请使用下面描述的访问函数检索查询结果。避免直接引用 PGresult 结构的字段，因为它们将来可能改变。

- PQresultStatus 返回查询的结果状态。PQresultStatus 可以返回下列值之一：

  ```
  PGRES_EMPTY_QUERY,
  PGRES_COMMAND_OK,  /* the query was a command */
  PGRES_TUPLES_OK,  /* the query successfully returned tuples */
  PGRES_COPY_OUT,
  PGRES_COPY_IN,
  PGRES_BAD_RESPONSE, /* an unexpected response was received */
  PGRES_NONFATAL_ERROR,
  PGRES_FATAL_ERROR
  ```

  如果结果状态为 PGRES_TUPLES_OK，就可以使用下面的例程检索查询返回的元组。

- PQntuples 返回查询结果中的元组（实例）数。

  ```
  int PQntuples(PGresult *res);
  ```

- PQnfields 返回查询结果中的字段（属性）数。

  ```
  int PQnfields(PGresult *res);
  ```

- PQfname 返回与给定字段编号相关联的字段（属性）名。字段编号从 0 开始。

  ```
  char *PQfname(PGresult *res,
  ```

```
                int field_index);
```

- PQfnumber 返回与给定字段名相关联的字段（属性）编号。

  ```
  int PQfnumber(PGresult *res,
                char* field_name);
  ```

- 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(PGresult *res,
              int field_num);
  ```

- PQfsize 返回与给定字段编号相关联的字段的大小，以字节计。如果返回的大小为 -1，则该字段是变长字段。字段编号从 0 开始。

  ```
  int2 PQfsize(PGresult *res,
               int field_index);
  ```

- PQgetvalue 返回字段（属性）值。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。如果查询是 BINARY 游标的结果，那么 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。此时由程序员负责把数据转换成正确的 C 类型。PQgetvalue 返回的值指向属于 PGresult 结构的存储空间。如果需要在 PGresult 结构本身的生存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。

  ```
  char* PQgetvalue(PGresult *res,
                   int tup_num,
                   int field_num);
  ```

- PQgetisnull 测试一个字段是否为 NULL 条目。

  ```
  int PQgetisnull(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

  如果字段包含 NULL，此函数返回 1；包含已知值则返回 0。

- PQgetlength 返回字段（属性）的长度，以字节计。如果字段是 struct varlena，这里返回的长度不含 varlena 的大小字段，即少 4 字节。

  ```
  int PQgetlength(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

- PQcmdStatus 返回与最后一条查询命令相关联的命令状态。

  ```
  char *PQcmdStatus(PGresult *res);
  ```

- PQcmdTuples 返回受最后一条命令影响的行数。

  ```
  const char *PQcmdTuples(PGresult *res);
  ```

  如果最后一条命令是 INSERT、UPDATE 或 DELETE，此函数返回一个包含受影响行数的字符串。如果最后一条命令是其他命令，返回空字符串。

- PQoidStatus 如果最后一条查询是 INSERT 命令，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(PGresult *res);
```

- 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 */
             PGresult* res,
             PQprintOpt* po);

struct _PQprintOpt
 {
  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 */
 };
```

此函数旨在取代现已过时的 PQprintTuples()。

- PQprintTuples 打印出所有的元组以及（可选的）属性名到指定的输出流。程序 psql 和 monitor 的输出都使用 PQprintTuples。

```
void PQprintTuples(PGresult* res,
                   FILE* fout,      /* output stream */
                   int printAttName,/* print attribute names or
 not*/
                   int terseOutput, /* delimiter bars or not?*/
                   int width);      /* width of column, variable
 width if 0*/
```

- PQdisplayTuples 打印出所有的元组以及（可选的）属性名到指定的输出流。

```
void PQdisplayTuples(
                     PGresult* res,
                     FILE* fout,           /* output stream */
                     int fillAlign,        /* space fill to align
 columns */
                     const char *fieldSep, /* field separator */
                  int printHeader,      /* display headers? */
               int quiet);           /* suppress print of row count
 at end */
```

PQdisplayTuples() 本意是取代 PQprintTuples()，而它又被 PQprint() 取代。

- PQclear 释放与 PGresult 关联的存储。每个查询结果在不再使用时都应妥善释放。不这样做将导致前端应用内存泄漏。

```
void PQclear(PQresult *res);
```

41.4. 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。

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               PQArgBlock *args,
               int nargs);
```

fnid 参数是要执行的函数的对象标识符。result_buf 是装载返回值的缓冲区。调用者必须已分配足够的空间来存储返回值。结果长度将返回到 result_len 所指向的存储中。如果结果将是整数值，则 result_is_int 应设为 1；否则应设为 0。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 resultStatus。不再需要结果时，调用者负责用 PQclear 释放 PGresult。

41.5. 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关系的兴趣。当另一个后端执行带该关系名的 NOTIFY 命令时，所有监听该关系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该关系传递。每当所连接的后端收到异步通知时，libpq 应用都会得到通知。但后端到前端的通信并不是异步的。通知随其他查询结果捎带而来。因此，
应用必须提交查询（哪怕是空查询）才能收到后端通知。实际上，libpq 应用必须轮询后端，看是否有待处理的通知信息。查询执行之后，前端可以调用 PQNotifies 查看后端是否有可用的通知数据。

- PQNotifies 从后端发来的尚未处理的通知列表中返回通知。如果后端没有待处理的通知则返回 NULL。PQNotifies 的行为类似弹出栈。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);
```

第二个示例程序演示了异步通知的使用。

41.6. 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，应用才能充分利用这一能力。

- 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。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单个字符 "." 组成，这表示后端服务器已发送完 copy 命令的结果。因此，如果应用可能收到超过 length-1 个字符的行，就必须非常仔细地检查 PQgetline 的返回值。../src/bin/psql/psql.c 中的代码包含正确处理 copy 协议的例程。

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

- PQputline 向后台服务器发送一个以空字符结尾的字符串。应用必须显式发送单个字符 "."，以告知后端它已发送完数据。

  ```
  void PQputline(PGconn *conn,
                 char *string);
  ```

- 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);
  ```

  ```
  PQexec(conn, "create table foo (a int4, b char16, d float8)");
  PQexec(conn, "copy foo from stdin");
  PQputline(conn, "3<TAB>hello world<TAB>4.5\n");
  PQputline(conn,"4<TAB>goodbye world<TAB>7.11\n");
  ...
  PQputline(conn,".\n");
  PQendcopy(conn);
  ```

41.7. 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)
  ```

41.8. 用户认证函数

如果用户已生成适当的认证凭据（例如获取 Kerberos ticket），前端/后端认证过程由 PQexec 处理，无需任何进一步的干预。下面的例程可由 libpq 程序调用来调整认证过程的行为。

- fe_getauthname 返回一个指向静态空间的指针，其中包含用户经过认证所用的名称。强烈建议应用用此例程代替对 getenv(3) 或 getpwuid(3) 的调用，因为经过认证的用户名完全有可能与 USER 环境变量的值或用户在 /etc/passwd 中的条目不同。

  ```
  char *fe_getauthname(char* errorMessage)
  ```

- `fe_setauthsvc` 指定 `libpq` 应使用认证服务 name 而不是其编译期默认值。此值通常取自命令行开关。

  ```
  void fe_setauthsvc(char *name,
                     char* errorMessage)
  ```

  认证尝试的任何错误消息都会返回到 errorMessage 参数中。

41.9. 缺陷

查询缓冲区长 8192 字节，超过该长度的查询将被静默截断。

41.10. 示例程序

41.10.1. 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.0, 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 (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"BEGIN command failed0);
            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 myportal CURSOR FOR select *
 from pg_database");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"DECLARE CURSOR command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          PQclear(res);

          res = PQexec(conn,"FETCH ALL in myportal");
          if (PQresultStatus(res) != PGRES_TUPLES_OK) {
            fprintf(stderr,"FETCH ALL command didn't return tuples
 properly0);
            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("0);

            /* 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("0);
            }

            PQclear(res);

            /* close the portal */
            res = PQexec(conn, "CLOSE myportal");
            PQclear(res);

            /* end the transaction */
            res = PQexec(conn, "END");
            PQclear(res);

            /* close the connection to the database and cleanup */
            PQfinish(conn);

          /*   fclose(debug); */
          }
```

41.10.2. 示例程序 2

```
          /*
           * testlibpq2.c
           *   Test of the asynchronous notification interface
           *
             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];

           * Then start up this program
           * After the program has begun, 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.0, db
Name);
    fprintf(stderr,"%s",PQerrorMessage(conn));
    exit_nicely(conn);
  }

  res = PQexec(conn, "LISTEN TBL2");
  if (PQresultStatus(res) != PGRES_COMMAND_OK) {
    fprintf(stderr,"LISTEN command failed0);
    PQclear(res);
    exit_nicely(conn);
  }
```

```
           /* should PQclear PGresult whenever it is no longer needed
 to avoid
              memory leaks */
           PQclear(res);

           while (1) {
               /* async notification only come back as a result of a
 query*/
               /* we can send empty queries */
               res = PQexec(conn, " ");
         /*      printf("res->status = %s0, pgresStatus[PQresult
Status(res)]); */
               /* check for asynchronous returns */
               notify = PQnotifies(conn);
               if (notify) {
                fprintf(stderr,
                     "ASYNC NOTIFY of '%s' from backend pid '%d'
 received0,
                     notify->relname, notify->be_pid);
                free(notify);
                break;
               }
               PQclear(res);
           }

           /* close the connection to the database and cleanup */
           PQfinish(conn);

         }
```

41.10.3. 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.0, db
Name);
              fprintf(stderr,"%s",PQerrorMessage(conn));
              exit_nicely(conn);
            }

            /* start a transaction block */
            res = PQexec(conn,"BEGIN");
            if (PQresultStatus(res) != PGRES_COMMAND_OK) {
              fprintf(stderr,"BEGIN command failed0);
              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 (PQresultStatus(res) != PGRES_COMMAND_OK) {
              fprintf(stderr,"DECLARE CURSOR command failed0);
              PQclear(res);
              exit_nicely(conn);
            }
            PQclear(res);

            res = PQexec(conn,"FETCH ALL in mycursor");
            if (PQresultStatus(res) != PGRES_TUPLES_OK) {
              fprintf(stderr,"FETCH ALL command didn't return tuples
 properly0);
              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] = %d0,
                    i, PQftype(res,i),
                    i, PQfsize(res,i));
            }
            for (i=0; i < PQntuples(res); i++) {
              int *ival;
              float *dval;
              int plen;
              POLYGON* pval;
              /*/
              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: got0, i);
            printf(" i = (%d bytes) %d,0,
                PQgetlength(res,i,i_fnum), *ival);
            printf(" d = (%d bytes) %f,0,
                PQgetlength(res,i,d_fnum), *dval);
            printf(" p = (%d bytes) %d points boundbox = (hi=%f/%f,
lo = %f,%f)0,
                PQgetlength(res,i,d_fnum),
                pval->npts,
                pval->boundbox.xh,
                pval->boundbox.yh,
                pval->boundbox.xl,
                pval->boundbox.yl);
          }

          PQclear(res);

          /* close the portal */
          res = PQexec(conn, "CLOSE mycursor");
          PQclear(res);

          /* end the transaction */
          res = PQexec(conn, "END");
          PQclear(res);

          /* close the connection to the database and cleanup */
          PQfinish(conn);

        }
```

部分 V. Reference

用户与程序员接口。

目录

第 42 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 43 章 Large Objects

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。

43.1. 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 UNIX 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。）

43.2. Inversion 大对象

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。

43.3. 大对象接口

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。该接口对动态装载的 C 函数与对 XXX LOST TEXT? WHAT SHOULD GO HERE?? 是一样的。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。

43.3.1. 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);
```

43.3.2. 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, text *filename)
```

filename 参数指定要导入为大对象的文件的 UNIX 路径名。

43.3.3. 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, text *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。

43.3.4. 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。

43.3.5. 向大对象写入数据

例程

```
int lo_write(PGconn *conn, int fd, char *buf, int 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。

43.3.6. 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。

43.3.7. 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。

43.4. 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';
```

43.5. 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。

43.6. 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
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ|INV_WRITE);
    if (lobjId == 0) {
     fprintf(stderr, "can't create large object");
    }

    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
     }
    }

    (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",
          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,"0);
            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",
                  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,"0);
            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",
                  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
                  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
                      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_filename0,
```

```
                    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.0,
database);
                fprintf(stderr,"%s",PQerrorMessage(conn));
                exit_nicely(conn);
               }

               res = PQexec(conn, "begin");
               PQclear(res);

               printf("importing file
           /*  lobjOid = importFile(conn, in_filename); */
               lobjOid = lo_import(conn, in_filename);
           /*
               printf("as large object %d.0, lobjOid);

               printf("picking out bytes 1000-2000 of the large object0)
;
               pickout(conn, lobjOid, 1000, 1000);

               printf("overwriting bytes 1000-2000 of the large object
with X's0);
               overwrite(conn, lobjOid, 1000, 1000);
           */

               printf("exporting large object to file
           /*    exportFile(conn, lobjOid, out_filename); */
               lo_export(conn, lobjOid,out_filename);

               res = PQexec(conn, "end");
               PQclear(res);
               PQfinish(conn);
               exit(0);
           }
```

第 44 章 ecpg - C 中的嵌入式 SQL

Linux Tolke
Michael Meskes

本文档描述 Postgres 的 C 语言嵌入式 SQL 包。它由 Linus Tolke[1] 和 Michael Meskes[2] 编写。

> **注意**
>
> 授予复制和使用的许可，方式与你被允许复制和使用 PostgreSQL 其余部分的方式相同。

44.1. 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。我见过和听说过的大多数嵌入式 SQL 预处理器都做了扩展，因此要在它们之间获得可移植性本来就很困难。我没有读过这个标准，但我希望我的实现偏离得不太远，并且有可能把为其他 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。

44.2. 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。

44.3. 如何使用 egpc

本节介绍如何使用 egpc 工具。

44.3.1. 预处理器

预处理器叫做 ecpg。安装后它位于 Postgres 的 `bin/` 目录中。

[1] mailto:linus@epact.se
[2] mailto:meskes@debian.org

44.3.2. 库

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, FILE *stream)

如果以第一个参数非零调用它，就会打开调试日志。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`）在除 `EXEC SQL COMMIT`、`EXEC SQL ROLLBACK`、`EXEC SQL CONNECT` 之外的所有 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。

44.3.3. 错误处理

为了能够检测来自 Postgres 服务器的错误，你需要在文件的 include 节中加入类似下面的一行：

```
exec sql include sqlca;
```

这会定义一个结构和名为 *`sqlca`* 的变量，如下所示：

```
struct sqlca {
    int sqlcode;
    struct {
        int sqlerrml;
        char sqlerrmc[1000];
    } sqlerrm;
} sqlca;
```

如果最后一条 SQL 语句发生了错误，*`sqlca.sqlcode`* 将非零。如果 *`sqlca.sqlcode`* 小于 0，则是某种严重错误，比如数据库定义与给定的查询不匹配。如果大于 0，则是普通错误，比如表中不包含所请求的行。

sqlca.sqlerrm.sqlerrmc 将包含一个描述错误的字符串。该字符串以 “line 23.” 结尾，其中行号是源文件中的行号（实际上是预处理器生成的文件中的行号，
但我希望将来能把它修正为输入文件中的行号。）

可能发生的错误列表：

-1, Unsupported type %s on line %d.

通常不应出现。这表明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。

-1, Too many arguments line %d.

预处理器出错了，生成了不正确的代码。

-1, Too few arguments line %d.

预处理器出错了，生成了不正确的代码。

-1, Error starting transaction line %d.

Postgres signalled to us that we cannot open the connection.

-1, 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。

1, 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，或者游标已经取到了末尾。

-1, To many matches line %d.

这表示查询返回了多行。你执行的 `SELECT` 可能不是唯一的。

-1, 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，其值无法解释为 `int`。库使用 `strtol` 做这一转换。

-1, 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，其值无法解释为 `unsigned int`。库使用 `strtoul` 做这一转换。

-1, 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，其值无法解释为 `float`。库使用 `strtod` 做这一转换。

-1, Too few arguments line %d.

这表示 Postgres 返回的记录多于我们匹配的变量。也许你在 `INTO :var1,:var2` 列表中漏掉了几个宿主变量。

-1, Too many arguments line %d.

这表示 Postgres 返回的记录少于我们的宿主变量数。也许你的 `INTO :var1,:var2` 列表中宿主变量太多了。

-1, Empty query line %d.

Postgres 返回了 PGRES_EMPTY_QUERY。

-1, Error: %s line %d.

这表示 Postgres 返回了 PGRES_NONFATAL_ERROR、PGRES_FATAL_ERROR 或 PGRES_BAD_RESPONSE 错误之一。是哪一个以及为什么在消息中有说明。

-1, Postgres error line %d.

Postgres 返回了库不知道如何处理的内容。这很可能是因为 Postgres 的版本与 ecpg 库的版本不匹配。

-1, Error committing line %d.

`COMMIT` 期间出错。`EXEC SQL COMMIT` 被转换为 Postgres 中的 `end` 操作，而该操作无法执行。

-1, Error rolling back line %d.

`ROLLBACK` 期间出错。`EXEC SQL ROLLBACK` 被转换为 Postgres 中的 `abort` 操作，而该操作无法执行。

-1, ECPGconnect: could not open database %s.

连接数据库失败。

44.4. 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。

44.5. 从其他 RDBMS 软件包移植

待由了解各种 RDBMS 软件包并实际移植过东西的人来写……

44.6. 安装

从 0.5 版开始，ecpg 与 Postgres 一起发行。因此你的预编译器、库和头文件应该已经被即时编译和安装。

44.7. 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。

44.7.1. 待办列表

这个版本的预处理器有一些缺陷：

预处理器输出

这些变量应该是 static 的。

预处理器无法对你的 SQL 语句做语法检查

你写的任何内容都会几乎原样复制给 Postgres，直到运行时你才能定位错误。

并非只限于字符串

ecpg 所使用的 PQ 接口（尤其是 PQexec 函数）依赖请求是以字符串形式构建的。在某些情况下，比如数据中包含空字符时，这会是一个严重的问题。

错误码

不同的错误应当有不同的错误编号，而不是全都用 -1。

库函数

to_date et al.

records

希望能够在 declare 区中以某种方式定义记录或结构，使该记录可以从数据库中的一行来填充。

这是一种每次处理整行的更简单方式。

数组操作

Oracle 有能提升速度的数组操作。在 ecpg 中实现它只是出于兼容性原因。要让它们提升速度，需要对 Postgres 内部机制比我更深入的理解。

指示符变量

Oracle 有指示符变量，用来说明一个值是 `null` 还是空。这大大简化了数组操作，并提供了一种绕过 `VARCHAR2` 处理中某些设计缺陷的方法（比如空字符串无法与 `null` 值区分）。我不确定这是 Oracle 的扩展还是 ANSI 标准的一部分。

typedefs

除了记录和数组这样的复杂类型外，typedef 也值得处理。

脚本转换

建立数据库需要一些包含表定义和其他配置参数的脚本。
如果你有一套旧数据库的这些脚本，你会希望直接应用它们得到一个以相同方式工作的 Postgres 数据库。

这种功能可以用一些转换脚本来实现，但速度永远无法以这种方式达成。要做到这一点，你需要对数据库的构造和使用有比脚本所能实现的更深入的理解。

44.7.2. 预处理器

前四行被写入输出。两行注释和两行与库接口所必需的 include 行。

然后预处理器单遍工作，边读输入文件边向输出写入。通常它只是把所有内容原样回显到输出，不做进一步检查。

当遇到 `EXEC SQL` 语句时，它会介入并根据其种类改变它们。`EXEC SQL` 语句可以是下列之一：

Declare sections

声明节以

```
exec sql begin declare section;
```

开始，以

```
exec sql end declare section;
```

结束。此节内只允许变量声明。
此节内声明的每个变量也按名称索引连同其对应类型登记到一个变量列表中。

声明也会被回显到文件中，使该变量同时成为普通的 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];
```

会被转换为

```
struct varchar_var { int len; char arr[180]; } var;
```

包含语句

include 语句的形式是：

```
exec sql include filename;
```

它被转换成

```
#include <filename.h>
```

连接语句

connect 语句的形式是：

```
exec sql connect 'database';
```

该语句被转换成

```
ECPGconnect("database");
```

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。此时会在先前在 declare 区中声明的变量里查找具有该名称的变量，并根据 SQL 语句是否知道它是用于输入还是输出的变量，把指向这些变量的指针写到输出中，供该函数访问。

对于作为 SQL 请求一部分的每个变量，函数还会得到另外五个参数。

作为特殊符号的类型
指向值的指针
如果变量是 varchar 则为变量的大小
数组中的元素个数（用于数组抓取）
到数组中下一个元素的偏移（用于数组抓取）

由于数组抓取尚未实现，最后两个参数其实并不重要。也许本可以把它们省去。

44.7.3. 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：

```
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
/* exec sql begin declare section */

 int index;
 int result;
/* exec sql end declare section */

...
    ECPGdo(__LINE__, "select res from mytable where index = ;;",
           ECPGt_int,&index,0,0,sizeof(int),
           ECPGt_EOIT,
           ECPGt_int,&result,0,0,sizeof(int),
           ECPGt_EORT );
```

(the indentation in this manual is added for readability and not something that the preprocessor can do.)

44.7.4. 库

库中最重要的函数是 `ECPGdo` 函数。它接受数量可变的参数。希望我们不会遇到对 varchar 函数可接受的变量数量有限制的机器。参数很容易累计到 50 个左右。

参数有：

一个行号

这是原语句的行号，只用于错误消息。

A string

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，每个输入变量得到五个参数。

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，每个输入变量得到五个参数。These variables are filled by the function.

ECPGt_EORT

一个枚举，表示不再有变量。

除非你发出提交事务命令，所有 SQL 语句都在一个事务中执行。其工作方式是：第一个事务，或提交/回滚之后的第一个事务，总是开始一个新事务。

待完成：描述其余条目的条目。

第 45 章 libpq

libpq 是 Postgres 的应用编程接口。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。本版本的文档描述 C 接口库。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 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。

45.1. 控制与初始化

下面的环境变量可用于设置默认的环境值，以避免把数据库名硬编码进应用程序：

- PGHOST 设置默认的服务器名。
- PGOPTIONS 设置 Postgres 后端的附加运行时选项。
- PGPORT 设置与 Postgres 后端通信所用的默认端口。
- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。
- PGDATABASE 设置默认的 Postgres 数据库名。
- 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。仅在启用了 Kerberos 认证时才会使用此环境变量。

45.2. 数据库连接函数

下面的例程用于从 C 程序建立与后端的连接。

- PQsetdbLogin 与后端建立一个新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                const char *pgport,
                const char *pgoptions,
                const char *pgtty,
                const char *dbName,
                const char *login,
                const char *pwd);
```

如果任何参数为 NULL，则检查相应的环境变量。如果环境变量也未设置，则使用硬编码的默认值。PQsetdbLogin 总是返回有效的 PGconn 指针。在通过连接发送查询之前，应当调用 PQstatus（见下文）确认连接已正确建立。libpq 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。

- PQsetdb 与后端建立一个新连接。

```
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()。

- PQconndefaults 返回连接选项。

  ```
  PQconninfoOption *PQconndefaults(void)

  struct PQconninfoOption
   {
    char   *keyword;   /* The keyword of the option */
    char   *environ;   /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;    /* Options value */
    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 options - don't
            create a field by default */
    int dispsize;  /* Field size in characters for dialog */
   };
  ```

  返回连接选项结构的地址。它可用来确定所有可用选项及其当前值。

- PQdb 返回连接的数据库名。

  ```
  char *PQdb(PGconn *conn)
  ```

- PQhost 返回连接的主机名。

  ```
  char *PQhost(PGconn *conn)
  ```

- PQoptions 返回连接中使用的 pgoptions。

  ```
  char *PQoptions(PGconn *conn)
  ```

- PQport 返回连接的 pgport。

  ```
  char *PQport(PGconn *conn)
  ```

- PQtty 返回连接的 pgtty。

  ```
  char *PQtty(PGconn *conn)
  ```

- PQstatus 返回连接的状态。状态可以是 CONNECTION_OK 或 CONNECTION_BAD。

  ```
  ConnStatusType *PQstatus(PGconn *conn)
  ```

- PQerrorMessage 返回与连接相关联的错误消息

  ```
  char *PQerrorMessage(PGconn* conn);
  ```

- PQfinish 关闭与后端的连接，同时释放 PGconn 结构使用的内存。调用过 PQfinish 之后，不应再使用该 PGconn 指针。

  ```
  void PQfinish(PGconn *conn)
  ```

- PQreset 重置与后端的通信端口。此函数将关闭到后端的 IPC 套接字连接，并尝试与同一个后端重新建立新连接。

  ```
  void PQreset(PGconn *conn)
  ```

- PQtrace 打开前端与后端之间所传消息的跟踪。这些消息会被回显到 debug_port 文件流。

  ```
  void PQtrace(PGconn *conn,
               FILE* debug_port);
  ```

- PQuntrace 关闭前端与后端之间所传消息的跟踪。

  ```
  void PQuntrace(PGconn *conn);
  ```

45.3. 查询执行函数

- PQexec 向 Postgres 提交一个查询。查询成功则返回 PGresult 指针，否则返回 NULL。如果返回了 NULL，可以用 PQerrorMessage 获取该错误的更多信息。

  ```
  PGresult *PQexec(PGconn *conn,
                   char *query);
  ```

  PGresult 结构封装了后端返回的查询结果。libpq 程序员应注意维护 PGresult 的抽象。请使用下面描述的访问函数检索查询结果。避免直接引用 PGresult 结构的字段，因为它们将来可能改变。

- PQresultStatus 返回查询的结果状态。PQresultStatus 可以返回下列值之一：

  ```
  PGRES_EMPTY_QUERY,
  PGRES_COMMAND_OK,  /* the query was a command */
  PGRES_TUPLES_OK,  /* the query successfully returned tuples */
  PGRES_COPY_OUT,
  PGRES_COPY_IN,
  PGRES_BAD_RESPONSE, /* an unexpected response was received */
  PGRES_NONFATAL_ERROR,
  PGRES_FATAL_ERROR
  ```

  如果结果状态为 PGRES_TUPLES_OK，就可以使用下面的例程检索查询返回的元组。

- PQntuples 返回查询结果中的元组（实例）数。

  ```
  int PQntuples(PGresult *res);
  ```

- PQnfields 返回查询结果中的字段（属性）数。

  ```
  int PQnfields(PGresult *res);
  ```

- PQfname 返回与给定字段编号相关联的字段（属性）名。字段编号从 0 开始。

  ```
  char *PQfname(PGresult *res,
  ```

```
                int field_index);
```

- PQfnumber 返回与给定字段名相关联的字段（属性）编号。

  ```
  int PQfnumber(PGresult *res,
                char* field_name);
  ```

- 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(PGresult *res,
              int field_num);
  ```

- PQfsize 返回与给定字段编号相关联的字段的大小，以字节计。如果返回的大小为 -1，则该字段是变长字段。字段编号从 0 开始。

  ```
  int2 PQfsize(PGresult *res,
               int field_index);
  ```

- PQgetvalue 返回字段（属性）值。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。如果查询是 BINARY 游标的结果，那么 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。此时由程序员负责把数据转换成正确的 C 类型。PQgetvalue 返回的值指向属于 PGresult 结构的存储空间。如果需要在 PGresult 结构本身的生存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。

  ```
  char* PQgetvalue(PGresult *res,
                   int tup_num,
                   int field_num);
  ```

- PQgetisnull 测试一个字段是否为 NULL 条目。

  ```
  int PQgetisnull(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

  如果字段包含 NULL，此函数返回 1；包含已知值则返回 0。

- PQgetlength 返回字段（属性）的长度，以字节计。如果字段是 struct varlena，这里返回的长度不含 varlena 的大小字段，即少 4 字节。

  ```
  int PQgetlength(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

- PQcmdStatus 返回与最后一条查询命令相关联的命令状态。

  ```
  char *PQcmdStatus(PGresult *res);
  ```

- PQcmdTuples 返回受最后一条命令影响的行数。

  ```
  const char *PQcmdTuples(PGresult *res);
  ```

  如果最后一条命令是 INSERT、UPDATE 或 DELETE，此函数返回一个包含受影响行数的字符串。如果最后一条命令是其他命令，返回空字符串。

- PQoidStatus 如果最后一条查询是 INSERT 命令，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(PGresult *res);
```

- 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 */
             PGresult* res,
             PQprintOpt* po);

struct _PQprintOpt
 {
  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 */
 };
```

此函数旨在取代现已过时的 PQprintTuples()。

- PQprintTuples 打印出所有的元组以及（可选的）属性名到指定的输出流。程序 psql 和 monitor 的输出都使用 PQprintTuples。

```
void PQprintTuples(PGresult* res,
                   FILE* fout,      /* output stream */
                   int printAttName,/* print attribute names or
 not*/
                   int terseOutput, /* delimiter bars or not?*/
                   int width);      /* width of column, variable
 width if 0*/
```

- PQdisplayTuples 打印出所有的元组以及（可选的）属性名到指定的输出流。

```
void PQdisplayTuples(
                     PGresult* res,
                     FILE* fout,           /* output stream */
                     int fillAlign,        /* space fill to align
 columns */
                     const char *fieldSep, /* field separator */
                  int printHeader,      /* display headers? */
               int quiet);           /* suppress print of row count
 at end */
```

PQdisplayTuples() 本意是取代 PQprintTuples()，而它又被 PQprint() 取代。

- PQclear 释放与 PGresult 关联的存储。每个查询结果在不再使用时都应妥善释放。不这样做将导致前端应用内存泄漏。

```
void PQclear(PQresult *res);
```

45.4. 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。

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               PQArgBlock *args,
               int nargs);
```

fnid 参数是要执行的函数的对象标识符。result_buf 是装载返回值的缓冲区。调用者必须已分配足够的空间来存储返回值。结果长度将返回到 result_len 所指向的存储中。如果结果将是整数值，则 result_is_int 应设为 1；否则应设为 0。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 resultStatus。不再需要结果时，调用者负责用 PQclear 释放 PGresult。

45.5. 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关系的兴趣。当另一个后端执行带该关系名的 NOTIFY 命令时，所有监听该关系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该关系传递。每当所连接的后端收到异步通知时，libpq 应用都会得到通知。但后端到前端的通信并不是异步的。通知随其他查询结果捎带而来。因此，
应用必须提交查询（哪怕是空查询）才能收到后端通知。实际上，libpq 应用必须轮询后端，看是否有待处理的通知信息。查询执行之后，前端可以调用 PQNotifies 查看后端是否有可用的通知数据。

- PQNotifies 从后端发来的尚未处理的通知列表中返回通知。如果后端没有待处理的通知则返回 NULL。PQNotifies 的行为类似弹出栈。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);
```

第二个示例程序演示了异步通知的使用。

45.6. 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，应用才能充分利用这一能力。

- 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。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单个字符 "." 组成，这表示后端服务器已发送完 copy 命令的结果。因此，如果应用可能收到超过 length-1 个字符的行，就必须非常仔细地检查 PQgetline 的返回值。../src/bin/psql/psql.c 中的代码包含正确处理 copy 协议的例程。

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

- PQputline 向后台服务器发送一个以空字符结尾的字符串。应用必须显式发送单个字符 "."，以告知后端它已发送完数据。

  ```
  void PQputline(PGconn *conn,
                 char *string);
  ```

- 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);
  ```

  ```
  PQexec(conn, "create table foo (a int4, b char16, d float8)");
  PQexec(conn, "copy foo from stdin");
  PQputline(conn, "3<TAB>hello world<TAB>4.5\n");
  PQputline(conn,"4<TAB>goodbye world<TAB>7.11\n");
  ...
  PQputline(conn,".\n");
  PQendcopy(conn);
  ```

45.7. 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)
  ```

45.8. 用户认证函数

如果用户已生成适当的认证凭据（例如获取 Kerberos ticket），前端/后端认证过程由 PQexec 处理，无需任何进一步的干预。下面的例程可由 libpq 程序调用来调整认证过程的行为。

- fe_getauthname 返回一个指向静态空间的指针，其中包含用户经过认证所用的名称。强烈建议应用用此例程代替对 getenv(3) 或 getpwuid(3) 的调用，因为经过认证的用户名完全有可能与 USER 环境变量的值或用户在 /etc/passwd 中的条目不同。

  ```
  char *fe_getauthname(char* errorMessage)
  ```

- `fe_setauthsvc` 指定 `libpq` 应使用认证服务 name 而不是其编译期默认值。此值通常取自命令行开关。

  ```
  void fe_setauthsvc(char *name,
                     char* errorMessage)
  ```

  认证尝试的任何错误消息都会返回到 errorMessage 参数中。

45.9. 缺陷

查询缓冲区长 8192 字节，超过该长度的查询将被静默截断。

45.10. 示例程序

45.10.1. 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.0, 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 (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"BEGIN command failed0);
            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 myportal CURSOR FOR select *
 from pg_database");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"DECLARE CURSOR command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          PQclear(res);

          res = PQexec(conn,"FETCH ALL in myportal");
          if (PQresultStatus(res) != PGRES_TUPLES_OK) {
            fprintf(stderr,"FETCH ALL command didn't return tuples
 properly0);
            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("0);

            /* 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("0);
            }

            PQclear(res);

            /* close the portal */
            res = PQexec(conn, "CLOSE myportal");
            PQclear(res);

            /* end the transaction */
            res = PQexec(conn, "END");
            PQclear(res);

            /* close the connection to the database and cleanup */
            PQfinish(conn);

          /*   fclose(debug); */
          }
```

45.10.2. 示例程序 2

```
          /*
           * testlibpq2.c
           *   Test of the asynchronous notification interface
           *
             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];

           * Then start up this program
           * After the program has begun, 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.0, db
Name);
    fprintf(stderr,"%s",PQerrorMessage(conn));
    exit_nicely(conn);
  }

  res = PQexec(conn, "LISTEN TBL2");
  if (PQresultStatus(res) != PGRES_COMMAND_OK) {
    fprintf(stderr,"LISTEN command failed0);
    PQclear(res);
    exit_nicely(conn);
  }
```

```
           /* should PQclear PGresult whenever it is no longer needed
 to avoid
              memory leaks */
           PQclear(res);

           while (1) {
               /* async notification only come back as a result of a
 query*/
               /* we can send empty queries */
               res = PQexec(conn, " ");
         /*      printf("res->status = %s0, pgresStatus[PQresult
Status(res)]); */
               /* check for asynchronous returns */
               notify = PQnotifies(conn);
               if (notify) {
                fprintf(stderr,
                     "ASYNC NOTIFY of '%s' from backend pid '%d'
 received0,
                     notify->relname, notify->be_pid);
                free(notify);
                break;
               }
               PQclear(res);
           }

           /* close the connection to the database and cleanup */
           PQfinish(conn);

         }
```

45.10.3. 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.0, db
Name);
              fprintf(stderr,"%s",PQerrorMessage(conn));
              exit_nicely(conn);
            }

            /* start a transaction block */
            res = PQexec(conn,"BEGIN");
            if (PQresultStatus(res) != PGRES_COMMAND_OK) {
              fprintf(stderr,"BEGIN command failed0);
              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 (PQresultStatus(res) != PGRES_COMMAND_OK) {
              fprintf(stderr,"DECLARE CURSOR command failed0);
              PQclear(res);
              exit_nicely(conn);
            }
            PQclear(res);

            res = PQexec(conn,"FETCH ALL in mycursor");
            if (PQresultStatus(res) != PGRES_TUPLES_OK) {
              fprintf(stderr,"FETCH ALL command didn't return tuples
 properly0);
              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] = %d0,
                    i, PQftype(res,i),
                    i, PQfsize(res,i));
            }
            for (i=0; i < PQntuples(res); i++) {
              int *ival;
              float *dval;
              int plen;
              POLYGON* pval;
              /*/
              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: got0, i);
            printf(" i = (%d bytes) %d,0,
                PQgetlength(res,i,i_fnum), *ival);
            printf(" d = (%d bytes) %f,0,
                PQgetlength(res,i,d_fnum), *dval);
            printf(" p = (%d bytes) %d points boundbox = (hi=%f/%f,
lo = %f,%f)0,
                PQgetlength(res,i,d_fnum),
                pval->npts,
                pval->boundbox.xh,
                pval->boundbox.yh,
                pval->boundbox.xl,
                pval->boundbox.yl);
          }

          PQclear(res);

          /* close the portal */
          res = PQexec(conn, "CLOSE mycursor");
          PQclear(res);

          /* end the transaction */
          res = PQexec(conn, "END");
          PQclear(res);

          /* close the connection to the database and cleanup */
          PQfinish(conn);

        }
```

第 46 章 pgtcl

pgtcl 是一个 tcl 包，供前端程序与 Postgres 后端交互。pgtcl 不使用 libpq 库，而是通过前端-后端协议直接与后端通信。因此，它比构建在 libpq 之上的以前的 postgres->tcl 绑定更高效。此外，pgtcl 可以在单个前端应用程序中处理多个后端连接。

本包最初由 Jolly Chen 编写。

46.1. 命令

pg_lo* 例程是 Postgres 中 Inversion 大对象的接口。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。

表 46.1. 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_exec	向后端发送一个查询
pg_select	循环遍历 select 语句的结果
pg_result	操作查询的结果
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

有些命令与 libpq 中用于连接和查询操作的函数等价。

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。

46.2. 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_disconnect $conn
    return $datnames
}
```

46.3. 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 dbName [-host hostName]
  [-port portNumber] [-tty pqtty] [-options optionalBackendArgs]
```

输入

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*]

(need information thomas 1997-12-24)

[-options *optionalBackendArgs*]

指定 *dbName* 所在后端服务器的选项。

输出

dbHandle

返回结果是一个错误消息或一个数据库连接的句柄。句柄以 "pgp" 前缀开头

描述

`pg_connect` 打开一个到 Postgres 后端的连接。

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_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

queryHandle

返回结果是一个错误消息或一个查询结果的句柄。

描述

`pg_exec` 向 Postgres 后端提交一个查询并返回结果。句柄以 "pgp" 前缀开头。

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 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

queryHandle

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 后端提交一个查询并返回结果。*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，它都会用字段名作为关联索引填入该元组的查询结果。

用法

```
set DB "mydb"
set conn [pg_connect $DB]
pg_select $conn "SELECT * from table" array {
 puts [format "%5d %s" array(control) array(name)]
}
pg_disconnect $conn
```

pg_result

pg_result — 获取关于查询结果的信息

大纲

```
pg_result queryHandle resultOption
```

输入

queryHandle

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。

-oid

如果最后的查询是 insert，返回所插入元组的 oid

-conn

产生该结果的连接

-assign arrayName

把结果赋给一个数组

-numTuples

查询中的元组数

-attributes

返回元组属性的名/类型对的列表

-getTuple tupleNumber

以列表形式返回该元组的各值

-clear

清除结果缓冲区。此后不要重用

输出

queryHandle

返回结果是一个错误消息或一个查询结果的句柄。

描述

`pg_result` 返回关于一个查询的信息。

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 47 章 ODBC 接口

Tim Goeke

> **注意**
>
> 由 Tim Goeke[1] 供稿

ODBC 是一个抽象 API，让你能够使用 ODBC API 编写标准的"ODBC"代码。

47.1. 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，从一个文本文件到一个 Oracle 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。PostODBC 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，你都应该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 PostgreSQL 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。

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。

如果我为 PostgreSQL 编写一个应用，我能否通过 ODBC 调用访问 PostgreSQL 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？

再说一遍，ODBC API 集正是该走的路。你可以在 Microsoft 的网站或你的 Visual C++ 文档（如果你用的就是它）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

> **提示**
>
> 你必须先设置一个 DSN。

[1] mailto:tgoeke@xpressway.com

提示

PostgreSQL 的 datetime 类型会让 MS Access 出问题。

第 48 章 JDBC 接口

Postgres 有一个可用的 JDBC 接口。它使用 Java 语言代码所公认的工具在别处文档化。

部分 VI. 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 49 章 架构

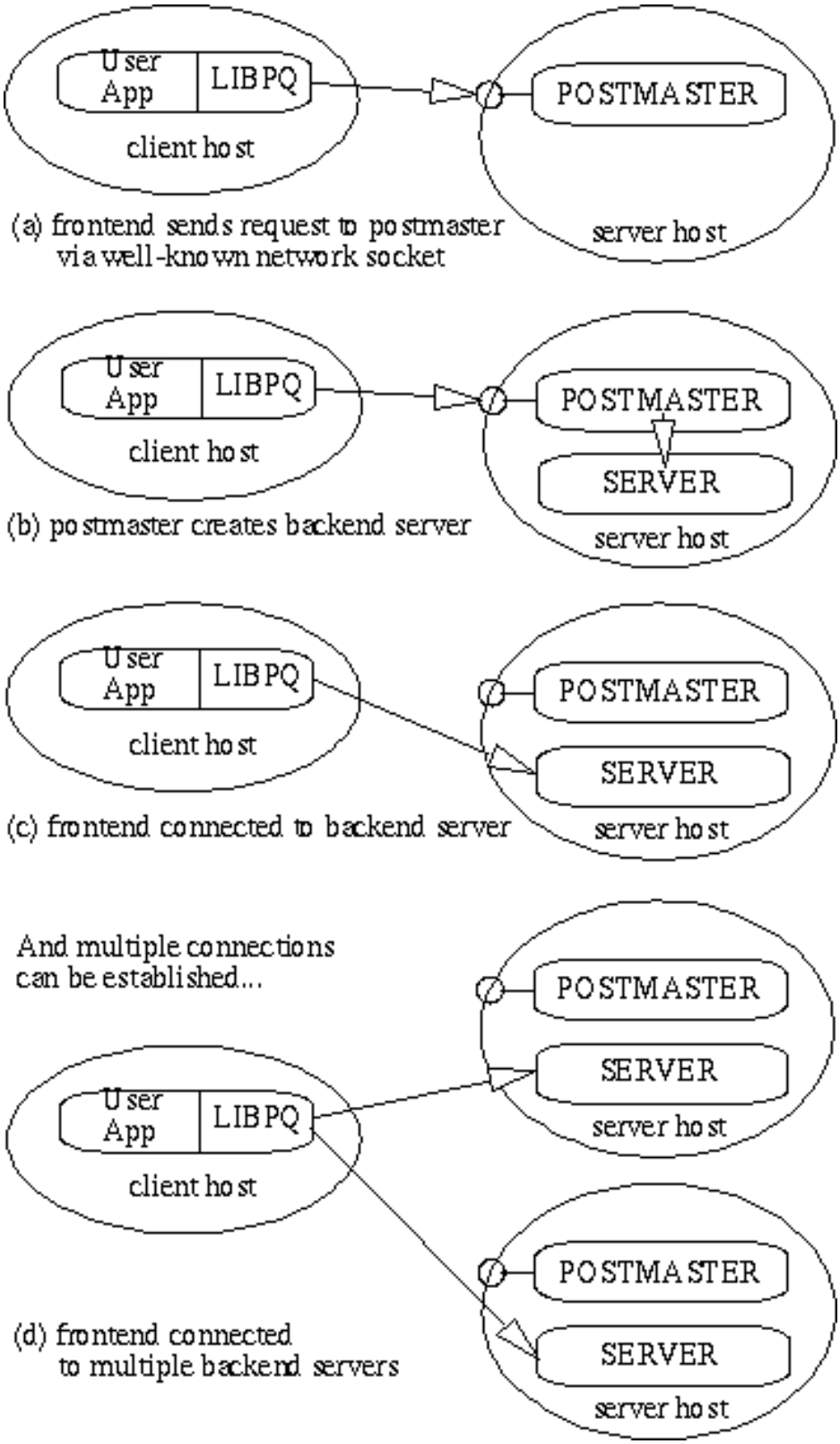
49.1. Postgres 体系结构概念

在继续之前，你应该先了解 Postgres 的基本体系结构。理解 Postgres 各部分之间如何交互，会让下一章的内容更加清晰。用数据库行话来说，Postgres 使用一种简单的“每用户一个进程”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：

- 一个监督守护进程（postmaster），
- 用户的前端应用程序（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 管理同一台主机上给定的一组数据库。这样一组数据库称为一个安装（installation）或站点（site）。
希望访问某个安装中特定数据库的前端应用程序调用相应的库。该库把用户请求通过网络发送给 postmaster（图 49.1(a)），后者再启动一个新的后端服务器进程（图 49.1(b)）

图 49.1. 连接是如何建立的



并把前端进程连接到这个新的服务器（图 49.1(c)）。从那一刻起，前端进程与后端服务器之间的通信不再经过 postmaster 干预。因此，postmaster 始终在运行并等待请求，而前端和后端进程则随需生灭。`libpq` 库允许单个前端与多个后端进程建立多条连接。但是，前端应用程序仍然是单线程进程。`libpq` 目前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 和后端总是运行在同一台机器（数据库服务器）上，而前端应用程序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 “超级用户”的用户 id 运行。注意，Postgres 超级用户并不需要是一个特殊的用户（例如名为“postgres”的用户）。而且，Postgres 超级用户绝对不应该是 UNIX 超级用户“root”！ 无论如何，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 50 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。

50.1. 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。

50.2. 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                           |
+===+======================================+
```

50.3. 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）。

50.4. Postgres GEQO 的未来实现任务

50.4.1. 基本改进

50.4.1.1. 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。

50.4.1.2. 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

50.4.1.3. 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。

50.4.1.4. 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 :-(

50.4.2. 进一步改进

在 Postgres 中启用浓密查询树处理；这可能会改进查询计划的质量。

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 51 章 前端/后端协议

Phil Thompson

> **注意**
>
> 由 Phil Thompson[1]

Postgres 使用基于消息的协议在前端和后端之间通信。该协议在 TCP/IP 上实现，也在 Unix 套接字上实现。Postgres v6.3 在协议中引入了版本号。这样做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所使用的协议。

本文档描述最初的带版本号的协议，即 v1.0。建立在该协议之上的更高层特性（例如 `libpq` 在连接建立之后如何传递某些环境变量）在别处介绍。

51.1. 概述

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 和后端角色不同，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。其中包括用户想要连接的用户名和数据库名。postmaster 然后使用它以及 pg_hba.conf(5) 文件中的信息，确定它要求前端进一步发送哪些认证信息（如果有的话），并相应地响应前端。

前端然后发送所需的任何认证信息。一旦 postmaster 验证通过，它就响应前端已通过认证，并移交给一个后端。

随后的通信是在前端和后端之间交换的查询和结果包。postmaster 不再参与通信。

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有什么内容。例外的是从前端发送到 postmaster 的包，它们由一个包长度和包本身组成。这一差异是历史原因造成的。

51.2. 协议

本节描述消息流。根据连接的状态，有四种不同类型的流：认证、查询、函数调用和终止。

51.2.1. 认证

前端发送一个 StartupPacket。postmaster 使用它以及 pg_hba.conf(5) 文件的内容确定前端必须使用哪种认证方法。postmaster 然后用下列消息之一响应：

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。

[1] mailto:phil@river-bank.demon.co.uk

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。

51.2.2. 查询

前端向后端发送一条 Query 消息。后端发送的响应取决于查询的内容。可能的响应如下。

CompletedResponse

查询正常完成。

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，后端随后发送一条带 "INSERT *oid rows*" 标签的 CompletedResponse 消息，其中 *rows* 是插入的行数，*oid* 在 *rows* 为 1 时是所插入行的对象 ID，否则 *oid* 为 0。

对于 delete(l) 命令，后端随后发送一条带 "DELETE *rows*" 标签的 CompletedResponse 消息，其中 *rows* 是删除的行数。

对于 update(l) 命令，后端随后发送一条带 "UPDATE *rows*" 标签的 CompletedResponse 消息，其中 *rows* 是更新的行数。

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

查询为空。

ErrorResponse

发生了一个错误。

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，即后端会立即发送另一条响应消息。

NotificationResponse

对一个先前执行过 listen(l) 命令的关系执行了 notify(l) 命令。通知是其他响应之外的附加内容，即后端会立即发送另一条响应消息。

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。

51.2.3. 函数调用

前端向后端发送一条 FunctionCall 消息。后端发送的响应取决于函数调用的结果。可能的响应如下。

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。

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，即后端会立即发送另一条响应消息。

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。

51.2.4. 终止

前端发送一条 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。

51.3. 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。即使缓冲区不够大而必须丢弃某些字符，前端也应始终读取完整的字符串。

> **注意**
>
> 8193 字节是允许的最大尺寸吗？

如果指定了 s，它就是字面值。例如 String、String("user")。

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')。

51.4. 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')

在发送它的上下文中（见 CopyInResponse）把该消息标识为一个 ASCII 行。

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，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，第 9 个字段对应第 2 个字节的第 8 位，依此类推。如果对应字段的值不是 NULL，则该位被置位。

然后，对每个非 NULL 值的字段，有下列内容：

Int32

指定字段值的大小，包括该大小本身。

Byte$n$

以 ASCII 字符指定字段本身的值。n 是上面的尺寸减 4。

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。

BinaryRow (B)

Byte1('B')

在其发送的上下文中（见 CopyOutResponse）标识此消息为二进制行。

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，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，第 9 个字段对应第 2 个字节的第 8 位，依此类推。如果对应字段的值不是 NULL，则该位被置位。

然后，对每个非 NULL 值的字段，有下列内容：

Int32

指定字段值的大小，不包括该大小本身。

Byte*n*

以二进制格式指定字段本身的值。n 是上面的尺寸。

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('D')

在其发送的上下文中（见 AsciiRow）标识此消息为 copy in 开始响应。

CopyOutResponse (B)

Byte1('B')

在其发送的上下文中（见 BinaryRow）标识此消息为 copy out 开始响应。

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')

标识此消息为空查询响应。

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

指定参数值的大小，不包括该大小本身。

Byte*n*

以二进制格式指定字段本身的值。n 是上面的尺寸。

FunctionResultResponse (B)

Byte1('V')

标识此消息为函数调用结果。

Byte1('G')

指定返回了实际结果。

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')

指定没有返回实际结果。

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

后端进程的进程 ID。

String

引发此通知的关系名称。

Query (F)

Byte1('Q')

标识此消息为查询。

String

查询本身。

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。

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，如果省略则默认为用户名。

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 52 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

部分 VII. 附录

其他相关信息。

目录

附录 A. 文档

Thomas Lockhart

Postgres 文档使用标准通用标记语言（SGML）DocBook[1] 文档类型定义（DTD）编写。

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。

> **注意**
>
> 这是三年来 Postgres 新文档的第一个发行版。内容和环境都还在变动和演化之中。

A.1. 引言

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）。

参阅 Introduction to DocBook[2]，那里有一份不错的 DocBook 特性 “快速入门” 摘要。DocBook Elements[3] 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[4] 和 Norm Walsh 的 Modular DocBook Stylesheets[5]。

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[6] 是 jade 输出的一种受支持格式，但当时未被使用，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。

A.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。

A.3. 构建文档

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
```

[1] http://www.ora.com/davenport/
[2] http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html
[3] http://www.ora.com/homepages/dtdparse/docbook/3.0/
[4] http://www.jclark.com/jade/
[5] http://www.berkshire.net/~norm/docbook/dsssl
[6] http://sunsite.unc.edu/pub/packages/TeX/systems/unix/

```
% 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
```

A.4. 工具集

A.4.1. jade

jade 当前的稳定版本是 1.0.1。

A.4.1.1. Linux 上的安装

安装 jade 及相关软件包的 RPM[7]。

A.4.1.2. 非 Linux 平台的安装

jade 还有一些其他打包发行形式。FreeBSD 似乎有一种可用。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。

对其他平台，安装 jade 及相关软件包的源码[8]并从零构建。

A.4.2. Modular Style Sheets

Modular Style Sheets 当前的稳定版本是 1.0.7。

A.5. v6.3 的硬拷贝生成

硬拷贝 Postscript 文档的生成方法是：把 SGML 源码转换为 RTF，然后导入 Applixware。经过少量清理（见下一节）后，把输出 “打印” 到一个 postscript 文件。

有些插图经过重画，以避免硬拷贝文档中出现位图 GIF 文件。有一幅关于系统目录的图足够复杂，没有时间重画。它用以下命令转换以适应页面：

```
% convert -v -geometry 400x400'>' figure03.gif con.gif
% convert -v -crop 400x380 con.gif connections.gif
```

A.5.1. RTF 清理过程

生成 Postscript 硬拷贝时必须处理若干事项：

Applixware RTF 清理

Applixware 在导入由 jade/MSS 生成的 RTF 方面似乎做得不完整。特别是，所有文本都被赋予 “Header1” 样式属性标签，尽管文本格式本身是可以接受的。另外，
目录页码引用的不是表中所列的节，而是指向 ToC 自身所在的页。

[7] ftp://ftp.cygnus.com/pub/home/rosalia/
[8] ftp://ftp.cygnus.com/pub/eichin/docware/SOURCES/

1. 输入以下命令生成 RTF 输入

   ```
   % cd doc/src/sgml
   % make tutorial.rtf
   ```

2. 在 Applix Words 中打开一个新文档，然后导入 RTF 文件。

3. 打印现有的目录，供下面几步标记使用。

4. 把插图插入文档。用居中边距按钮把每幅图在页面上居中。

   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。

5. 通读整个文档，调整分页和表格列宽。

6. 如果存在书目，Applix Words 似乎会把第一个标题之后的所有其余文本都标上下划线属性。选中所有其余文本，用下划线按钮关闭下划线，然后显式地为每个文档名和书名加下划线。

7. 通读整个文档，在 ToC 硬拷贝上标出每个 ToC 条目的实际页码。

8. 用正确的值替换 ToC 中右对齐的错误页码。这只需要每份文档几分钟。

9. 把文档保存为 Applix Words 原生格式，以便日后更容易地进行最后时刻的编辑。

10. 把文档导出为 Postscript 格式的文件。

11. 用 gzip 压缩 Postscript 文件。把压缩后的文件放入 `doc` 目录。

A.6. 备用工具集

sgml-tools 当前的稳定版本是 1.0.4。v1.0 发行版对目录树做了一些重构，以便更容易地支持更多文档样式，可能包括 DocBook。为 Postgres 评估过的 sgml-tools 版本只有 v0.99.0。

A.6.1. sgml-tools

安装 sgml-tools-0.99.0

把 sgml-tools-patches[9] 应用到 linuxdoc 样式。这些补丁修复了转换为 postscript 或 html 时表格格式和图形文件名的小问题。

A.6.2. sgml2latex

sgml2latex 当前的稳定版本是 1.4。我遗失了这个包的原始出处，所以暂时把它和这个示例放在一起发布。

安装 sgml2latex[10]。

A.6.3. latex

获取并安装 texmf、teTeX 或另一个提供完整 tex/latex 功能的软件包。

[9] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/sgml-tools-patches-0.99.0.tar.gz
[10] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/sgml2latex-format.1.4.tar.gz

把所需样式[11] linuxdoc-sgml.sty、linuxdoc-sgml-a4.sty isolatin.sty、qwertz.sty 和 null.sty 加入 texmf/tex/latex/tools/ 或适当的区域。

```
% cat latex-styles-0.99.0.tar.gz | (cd texmf/tex/latex/tools/; tar
 zxvf -)
```

运行 texhash 更新 tex 数据库。

[11] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/latex-styles-0.99.0.tar.gz

附录 B. 联系方式

B.1. 引言

B.2. 人物

- Thomas Lockhart[1] 负责 SQL 标准兼容性和文档。

[1] http://alumni.caltech.edu/~lockhart

SQL 参考文献

这里列出了一些关于 SQL 和 Postgres 的参考文献与读物。

SQL 参考书

SQL 特性的参考文本。

[bowman93] The Practical SQL Handbook. Using Structured Query Language. 3. Judity Bowman、Sandra Emerson和Marcy Damovsky. ISBN 0-201-44787-8. 1996. Addison-Wesley. 版权 © 1997 Addison-Wesley Longman, Inc..

[date97] A Guide to The SQL Standard. The SQL Standard. 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..

[melt93] Understanding the New SQL. 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。

PostgreSQL 专属文档

本节收录相关文档。

[admin-guide] The PostgreSQL. Administrator's Guide. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[programmers-guide] The PostgreSQL. Programmer's Guide. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[tutorial] The PostgreSQL. Reference Manual. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[users-guide] The PostgreSQL. Tutorial Introduction. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[reference] The PostgreSQL. User's Guide. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[yu95] The Postgres95. User Manual. YU95. 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。

[ong90] A Unified Framework for Version Modeling Using Production Rules in a Database System. ONG90. L. Ong和J. Goh. April, 1990. ERL Technical Memorandum M90/33. University of California, Berkeley CA.

[rowe87] The Postgres. Data Model. ROWE87. L. Rowe和M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

[ston86] The Design of Postgres. . STON86. M. Stonebraker和L. Rowe. Conference on Management of Data, Washington DC, May 1986.

[ston87a] The Design of the Postgres. Rules System. STON87a. M. Stonebraker、E. Hanson和C. H. Hong. Conference on Data Engineering, Los Angeles, CA, Feb. 1987.

[ston87b] The Postgres. Storage System. STON87b. M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

[ston89] A Commentary on the Postgres. Rules System. STON89. M. Stonebraker、M. Hearst和S. Potamianos. Record 18(3), Sept. 1989.

[ston90a] The Implementation of Postgres. . STON90a. M. Stonebraker、L. A. Rowe和M. Hirohama. Transactions on Knowledge and Data Engineering 2(1), March 1990.

[ston90b] On Rules, Procedures, Caching and Views in Database Systems. STON90b. M. Stonebraker和et. al.. Conference on Management of Data, June 1990.

索引