PostgreSQL从菜鸟到专家

翻译：洞庭湖的泥鳅

如有问题，请联系：

bambo.huang(at)gmail.com

**目录**

[引言 1](#_Toc322688823)

[第一章 PostgreSQL介绍 2](#_Toc322688824)

[基于数据编程 2](#_Toc322688825)

[静态数据 2](#_Toc322688826)

[用于数据存储的扁平文件 2](#_Toc322688827)

[重复单元和其他问题 3](#_Toc322688828)

[什么是数据库管理系统 3](#_Toc322688829)

[数据模型 4](#_Toc322688830)

[查询语言 6](#_Toc322688831)

[数据库管理系统的责任 7](#_Toc322688832)

[什么是PostgreSQL? 8](#_Toc322688833)

[PostgreSQL历史简介 8](#_Toc322688834)

[PostgreSQL架构 9](#_Toc322688835)

[通过PostgreSQL访问数据 9](#_Toc322688836)

[什么是开源？ 10](#_Toc322688837)

[相关资源 10](#_Toc322688838)

[第二章 关系数据库原理 11](#_Toc322688839)

[电子表格的局限性 11](#_Toc322688840)

[将数据存入数据库 13](#_Toc322688841)

[选择列 13](#_Toc322688842)

[为每个列选择数据类型 13](#_Toc322688843)

[标记行的唯一性 14](#_Toc322688844)

[在数据库中访问数据 14](#_Toc322688845)

[通过网络访问数据 15](#_Toc322688846)

[处理多用户访问 16](#_Toc322688847)

[数据分片和分块 16](#_Toc322688848)

[增加信息 18](#_Toc322688849)

[设计表 20](#_Toc322688850)

[基本数据类型 25](#_Toc322688851)

[处理未知的值：空值（NULL） 26](#_Toc322688852)

[回顾示例数据库 26](#_Toc322688853)

[摘要 27](#_Toc322688854)

[第三章 初步使用PostgreSQL 28](#_Toc322688855)

[在Linux和Unix系统中安装PostgreSQL 28](#_Toc322688856)

[在Linux中使用二进制文件安装PostgreSQL 28](#_Toc322688857)

[通过源码安装PostgreSQL 32](#_Toc322688858)

[在Linux和Unix上配置PostgreSQL 34](#_Toc322688859)

[在Windows中安装PostgreSQL 39](#_Toc322688860)

[使用Windows安装程序 40](#_Toc322688861)

[配置客户机访问 45](#_Toc322688862)

[建立示例数据库 46](#_Toc322688863)

[添加用户记录 46](#_Toc322688864)

[建立数据库 47](#_Toc322688865)

[建表 47](#_Toc322688866)

[移除表 49](#_Toc322688867)

[填充表 49](#_Toc322688868)

[摘要 52](#_Toc322688869)

[第四章 访问你的数据 53](#_Toc322688870)

[使用psql 53](#_Toc322688871)

[在Linux系统中启动 53](#_Toc322688872)

[在Windows系统中启动 54](#_Toc322688873)

[解决启动问题 54](#_Toc322688874)

[使用一些基本的psql命令 56](#_Toc322688875)

[使用SELECT语句 57](#_Toc322688876)

[覆盖列名 59](#_Toc322688877)

[控制行的顺序 59](#_Toc322688878)

[消除重复数据 61](#_Toc322688879)

[执行计算 63](#_Toc322688880)

[选择行 64](#_Toc322688881)

[使用更复杂的条件 66](#_Toc322688882)

[模式匹配 67](#_Toc322688883)

[限制结果集 68](#_Toc322688884)

[检查空值（NULL） 69](#_Toc322688885)

[检查时间和日期 70](#_Toc322688886)

[设置时间和日期的风格 70](#_Toc322688887)

[使用日期和时间函数 74](#_Toc322688888)

[多个表协同工作 75](#_Toc322688889)

[关联两个表 75](#_Toc322688890)

[给表赋予别名 80](#_Toc322688891)

[关联三个或更多的表 80](#_Toc322688892)

[SQL92的SELECT语法 84](#_Toc322688893)

[摘要 85](#_Toc322688894)

[第五章 PostgreSQL的命令行和图形界面工具 86](#_Toc322688895)

[psql 86](#_Toc322688896)

[启动psql 86](#_Toc322688897)

[在psql中输入命令 87](#_Toc322688898)

[使用命令历史 87](#_Toc322688899)

[在psql中执行脚本文件 88](#_Toc322688900)

[检查数据库 89](#_Toc322688901)

[psql命令行快速参考 89](#_Toc322688902)

[psql内部命令快速参考 90](#_Toc322688903)

[设置ODBC 92](#_Toc322688904)

[在Windows中安装ODBC驱动程序 92](#_Toc322688905)

[在Windows中建立一个数据源 94](#_Toc322688906)

[在Linux/Unix中安装ODBC驱动程序 96](#_Toc322688907)

[在Linux/Unix中建立一个数据源 96](#_Toc322688908)

[pgAdmin III 96](#_Toc322688909)

[安装pgAdmin III 96](#_Toc322688910)

[使用pgAdmin III 97](#_Toc322688911)

[phpPgAdmin 100](#_Toc322688912)

[安装phpPgAdmin 100](#_Toc322688913)

[使用phpPgAdmin 101](#_Toc322688914)

[Microsoft Access 103](#_Toc322688915)

[使用链接表 103](#_Toc322688916)

[输入数据及建立报表 106](#_Toc322688917)

[Microsoft Excel 108](#_Toc322688918)

[PostgreSQL相关工具的资源 111](#_Toc322688919)

[摘要 111](#_Toc322688920)

[第六章 数据交互 112](#_Toc322688921)

[添加数据到数据库中 112](#_Toc322688922)

[使用基本的INSERT语句 112](#_Toc322688923)

[使用更安全的插入语句 114](#_Toc322688924)

[插入数据到serial类型的列中 115](#_Toc322688925)

[插入空值 118](#_Toc322688926)

[使用\copy命令 120](#_Toc322688927)

[直接从另一个程序加载数据 122](#_Toc322688928)

[修改数据库中的数据 124](#_Toc322688929)

[使用UPDATE语句 125](#_Toc322688930)

[通过另一个表更新 126](#_Toc322688931)

[从数据库删除数据 127](#_Toc322688932)

[使用DELETE语句 127](#_Toc322688933)

[使用TRUNCATE语句 128](#_Toc322688934)

[摘要 129](#_Toc322688935)

[第七章 高级数据选择 131](#_Toc322688936)

[聚集函数 131](#_Toc322688937)

[count函数 132](#_Toc322688938)

[Min函数 139](#_Toc322688939)

[Max函数 139](#_Toc322688940)

[Sum函数 140](#_Toc322688941)

[Avg函数 140](#_Toc322688942)

[子查询 141](#_Toc322688943)

[返回多行记录的子查询 143](#_Toc322688944)

[相关子查询 144](#_Toc322688945)

[存在子查询（Existence Subqueries） 146](#_Toc322688946)

[UNION连接 147](#_Toc322688947)

[自连接 149](#_Toc322688948)

[外连接 150](#_Toc322688949)

[摘要 154](#_Toc322688950)

[第八章 数据定义 155](#_Toc322688951)

[数据类型 155](#_Toc322688952)

[布尔数据类型 155](#_Toc322688953)

[字符数据类型 157](#_Toc322688954)

[数字数据类型 159](#_Toc322688955)

[时间型数据类型 161](#_Toc322688956)

[数组 162](#_Toc322688957)

[数据操作 164](#_Toc322688958)

[在数据类型之间转换 165](#_Toc322688959)

[用于数据操作的函数 166](#_Toc322688960)

[魔法变量 167](#_Toc322688961)

[OID列 167](#_Toc322688962)

[表管理 168](#_Toc322688963)

[建表 168](#_Toc322688964)

[使用列约束 169](#_Toc322688965)

[使用表约束 172](#_Toc322688966)

[修改表结构 173](#_Toc322688967)

[删除表 176](#_Toc322688968)

[使用临时表 176](#_Toc322688969)

[视图 177](#_Toc322688970)

[建立视图 177](#_Toc322688971)

[删除和替换视图 180](#_Toc322688972)

[外键约束 180](#_Toc322688973)

[作为一个列的约束的外键 181](#_Toc322688974)

[作为一个表的约束的外键 181](#_Toc322688975)

[外键约束的选项 186](#_Toc322688976)

[摘要 188](#_Toc322688977)

[第九章 事务和锁 189](#_Toc322688978)

[什么是事务 189](#_Toc322688979)

[将数据改变分组到逻辑单元 189](#_Toc322688980)

[并发多用户访问数据 190](#_Toc322688981)

[ACID规则 191](#_Toc322688982)

[事务日志 191](#_Toc322688983)

[单用户情况下的事务 192](#_Toc322688984)

[包括多个表的事务 194](#_Toc322688985)

[事务和保存点 195](#_Toc322688986)

[事务的限制 198](#_Toc322688987)

[多用户情况下的事务 199](#_Toc322688988)

[实现隔离 199](#_Toc322688989)

[改变隔离级别 203](#_Toc322688990)

[使用显式和隐式的事务 203](#_Toc322688991)

[锁 203](#_Toc322688992)

[避免死锁 204](#_Toc322688993)

[显式锁定 206](#_Toc322688994)

[摘要 207](#_Toc322688995)

[第十章 函数、存储过程和触发器 208](#_Toc322688996)

[运算符 208](#_Toc322688997)

[运算符的优先级和结合性 209](#_Toc322688998)

[数学运算符 210](#_Toc322688999)

[比较和字符串运算符 211](#_Toc322689000)

[其他运算符 212](#_Toc322689001)

[内置函数 212](#_Toc322689002)

[过程语言 214](#_Toc322689003)

[PL/pgSQL入门 215](#_Toc322689004)

[函数重载 217](#_Toc322689005)

[列出函数 218](#_Toc322689006)

[删除函数 218](#_Toc322689007)

[引号 218](#_Toc322689008)

[解析一个存储过程 219](#_Toc322689009)

[函数参数 219](#_Toc322689010)

[注释 220](#_Toc322689011)

[声明 221](#_Toc322689012)

[赋值 223](#_Toc322689013)

[执行控制结构 225](#_Toc322689014)

[动态查询 231](#_Toc322689015)

[SQL函数 232](#_Toc322689016)

[触发器 233](#_Toc322689017)

[定义触发器过程 233](#_Toc322689018)

[建立触发器 233](#_Toc322689019)

[为什么使用存储过程和触发器 238](#_Toc322689020)

[摘要 239](#_Toc322689021)

# 引言

欢迎来到PostgreSQL数据库从入门到精通。

在我们生涯的早期，我们开始理解开源软件的质量。不仅通常情况下它们可以自由使用，而且它们提供极高的质量。如果你发现问题，你可以检查源代码，理解程序工作过程。如果你找到一个错误，你可以自己修复或找别人帮你修复。我们从1978年开始试用开源软件，包括优秀的GNU工具，包括GNU Emacs和GCC。我们从1993年开始使用Linux并愉快地使用Linux内核和GNU工具建立了一个完整的，自由的计算环境，并且使用X Window系统提供了一个图形用户界面。PostgreSQL是一个采取相同的开源理念的优秀的数据库系统（更多关于开源和自由的信息，请访问<http://www.opensource.org>）。

数据库是非常有用的东西。很多人发现“桌面数据库”在办公室和家里的小应用程序中非常有用。很多网站是数据驱动的，很多内容都由网页服务器后面的数据库提供。随着数据库的普及，我们觉得有必要写一本书介绍数据库理论和实践。

我们写这本书整体介绍数据库，全面覆盖现代的关系数据库的能以及怎么高效使用它们。使用 PostgreSQL作为他的数据库的人都没有觉得PostgreSQL在哪方面有什么不足。它支持优秀的数据库设计，非常有弹性和扩展性，且运行在几乎你所能想到的计算机上，包括Linux，Unix，Windows，Mac OS X，AIX，Solaris以及HP-UX。

对了，免得你好奇，我告诉你PostgreSQL念做“post-gres-cue-el”（而不是“post-gray-ess-cue-el”）。

本书大致分为三部分。第一部分包括入门，包括数据库概述（它们是什么以及它们用来干什么），尤其是PostgreSQL的概述（怎样获得，安装和启动以及使用）。如果你同时运行了示例，到第5章完成的时候，你将建立起第一个可工作的数据库并且可以使用一些工具来做一些有用的事情，例如输入数据和执行查询。

本书的第二部分深刻地探索关系数据库的核心：SQL查询语言。通过示例程序和“做一个尝试”章节，你将学到数据库编程的很多方面。，从简单的数据插入和修改，强大的各类查询到通过存储过程和触发器扩展数据库功能。本章最重要的内容是数据库无关，所以你从本章获取的知识能够让你在使用其他数据库时如鱼得水。当然，所有的用来说明的示例资源是用PostgreSQL来作为示例数据库。关于PostgreSQL的管理和数据库设计的习惯用法也将在本章完整讨论。

本书的第三部分关注于在你的程序中发挥PostgreSQL的能力。本章涵盖了通过大量的编程语言连接到数据库、执行查询以及处理结果集。无论你是使用PHP或者Perl开发动态网页、用Java或者C#开发企业级应用程序或者用C写一个客户端程序，你将找到你想要的章节。

这是PostgreSQL从入门到精通的第二版；第一版在2001年发布。从那时起，每章的内容都根据8.0版本的PostgreSQL做了升级。我们在本书中有机会补充了新的一章讨论通过C#访问PostgreSQL。

# PostgreSQL介绍

本书都是关于一个最近最成功的开源产品，一个名叫PostgreSQL的关系数据库。

数据库开发商和开源开发者都是PostgreSQL的热心拥护者。任何使用程序管理大量数据的人都可以从数据库中获得大量益处。PostgreSQL是一个非常优秀的关系数据库实现，全功能，开源且免费使用。

PostgreSQL 支持大量的主流开发语言，包括C，C++，Perl，Python，Java，Tcl以及PHP。它是最接近工业标准SQL92的查询语言，并且正在实现新的功能以兼容最新的SQL标准：SQL2003。PostgreSQL也获得数个奖项，包括三次被评为Linux Journal杂志编辑选择奖最佳数据库（2000,2003和2004年度）以及2004年度Linux新媒体奖最佳数据库系统。我们也许我们有点超越自我。你也许想知道到底PostgreSQL是什么，为什么你要使用它。

本章我们将设置场景为本书的剩余部分并提供一些关于数据库的概括性背景知识、不同类型的数据库、为什么他们非常有用以及PostgreSQL在哪些地方符合这些要求。

## 基于数据编程

基本上所有的非普通计算机程序操作大量的数据，而且大量的应用程序被开发用来处理数据而不是进行计算任务。有些人估计当今世界上80%的应用程序开发会通过某种方式连接到数据库中的复杂数据，所以数据库对于很多应用来说是一种非常重要的基础。

以数据编程的资源很丰富。大部分好的编程书籍都包含章节关于建立、存储和操作数据。有三本书（由Wrox发布）包含以数据编程的信息：

* 《Beginning Linux Programming》第三版覆盖了DBM库以及MySQL数据库系统。
* 《Professional Linux Programming》包含关于PostgreSQL和MySQL数据库系统的章节。
* 《Beginning Databases with MySQL》全面覆盖了MySQL数据库系统。

### 静态数据

数据有很多形态和大小，我们根据数据的不同性质处理数据。有些情况下，数据非常简单——也许仅仅是一个数字例如π的数值存在程序中用来画圆。程序本身可能包含这个硬编码的圆周率的值。我们把这种数据叫做静态数据，它永远不需要改变。

另一种静态数据的例子是欧洲一些国家的汇率。在欧元区，货币都按固定的保留小数点后6位的精度兑换到欧元。假设我们开发一个欧元区货币转换程序。他可能有一个硬编码的货币名称和基本汇率表，保存对应到欧元的兑换点数。这些汇率永远不会改变。不过还没完，因为这张表会随着欧元区国家数量的增长而增长。当一个国家注册到欧元区，这个国家货币对欧元的汇率将被设定，它将需要被加入这张表中。当这种情况发生，汇率转换程序需要修改，内置的货币表被修改，程序需要重建。这需要在每次货币表发生变化时做一次。

一个更好的办法是应用程序读取一个包含简单的货币信息的文件，文件可能包含货币的名称、国际符号以及汇率。之后在这个表需要改变时我们仅仅需要改变 这个文件，而不需要改变程序。

我们使用的这个数据文件没有特别的结构；它仅仅包含几行文本，包含一些数据让应用程序读取。它没有固定的结构。因此我们把这种文件叫做扁平文件。以下是我们的货币文件可能的样子：

France FRF 6.559570

Germany DEM 1.955830

Italy ITL 1936.270020

Belgium BEF 40.339901

### 用于数据存储的扁平文件

扁平文件对于很多应用类型来说非常有用。由于文件保持在可管理大小，我们很容易改变它，扁平文件可能非常符合我们的需求。

很多系统和程序，特别是UNIX平台，使用扁平文件存储数据或者进行数据交换。UNIX的密码文件就是一个例子，它通常情况下像以下的样子：

neil:\*:500:100:Neil Matthew:/home/neil:/bin/bash

nick:\*:501:100:Rick Stones:/home/rick:/bin/bash

这个例子包含一系列的信息和属性组成的记录。文件被设计成每行包含一个记录，整个文件则用来保存相关的记录到一起。某些情形下，这种模式还不够用，因而，我们需要增加扩展的功能来配合应用程序完成必须的工作。

### 重复单元和其他问题

假设我们决定扩展货币汇率程序（本章前面介绍过）来记录每个国家的语言，以及人口和面积。在扁平文件中，我们需要每行一个记录，每个记录由几个熟悉构成。记录中每个独立的属性总在固定的位置；例如货币符号总是第二个熟悉。所以我们可以认为按列读数据，每列总是相同类型的信息。

为了添加某个国家使用的语言，我们可能会认为我们只需为每一行要添加一个新列。当我们发现一些国家有不止一种官方语言时，我们碰到了一个问题。所以，在我们给Belgium的记录中，我们需要同时包含Flemish语和French语。对于Switzerland，我们需要添加四种语言。扁平文件在这是应该看起来像这样：

France FRF 6.559570 French 60424213 547030

Germany DEM 1.955830 German 82424609 357021

Italy ITL 1936.270020 Italian 58057477 301230

Belgium BEF 40.339901 Flemish French 10348276 30528

Switzerland CHF 1.5255 German French Italian Romansch 7450867 41290

这个问题被称为重复单元。我们现在的问题是重复的数据项（语言）是在记录内重复，也就是说数据可能会在列内重复，而不仅仅是记录。扁平的文件无法应付这类问题，因为它无法判断什么时候语言项列完了，开始下一个记录了。解决这个问题唯一的方法是在文件内添加一些结构，但这种情况下这个文件不再是扁平文件了。

这里有另一个示例。记录DVD详细信息的程序可能需要记录生产的年份，导演，风格以及演员列表。我们可以设计一个和Windows的.ini文件格式差不多的文件，就像这样：

[2001: A Space Odyssey]

year=1968

director=Stanley Kubrick

genre=science fiction

starring=Keir Dullea

starring=Leonard Rossiter

…

[Toy Story]

…

我们通过引入描述每个记录类型的标记解决了重复单元的问题。但是，现在你的程序为了获得需要的数据，需要读取和解析更复杂的文件。添加记录以及搜索这种结构就更困难了。我们怎样才能确保用于风格或分类的描述是从一个特定子集选择的。我们怎样才能很容易的搜索到Kubrick导演的电影？

由于对数据的需求越来越复杂，我们被迫写越来越多的代码用来读取和存储我们的数据。如果我们扩展我们的DVD管理程序，为DVD出租店主包含一些有用的信息——例如会员信息，租金，退租信息和预定信息，在扁平文件中管理所有这些信息的希望渺茫。

另外一个通用问题就是大小的问题。虽然包含结构的文本文件可以通过暴力扫描回答复杂的查询类似于“告诉我所有的最近三个月租过最少一次喜剧的会员”，不单代码很难编写，而且性能可能会很低。这是因为程序除了扫描整个文件来查找任何一片信息外没有别的办法，即使是仅仅返回一条数据的问题“谁主演了2001年的A Space Odyssey”。

我们需要的是一种在通用数据处理系统里用来存储和检索数据的方法，而不是总是发明一个个解决方案用于有点不同但非常相似的问题。

我们需要的是一个数据库和一个数据库管理系统。

## 什么是数据库管理系统

韦氏在线字典（Merriam-Webster online dictionary，<http://www.merriam-webster.com>）定义数据库为一个为（通过计算机）高速搜索和检索的而特别组织的大量数据集。

数据库管理系统（database management system，DBMS）通常是一套用来让程序开发人员从繁重的数据存储细节和管理中解脱出来的程序库、应用程序以及工具套件。它还提供了用来搜索和更新记录的能力。多年来，数据库管理系统发展出大量的特点用来解决各种各样特别的数据存储问题。

### 数据模型

从1960年代到1970年代，开发人员通过很多不同的方法建立数据库来解决重复单元的问题。这些方法从术语上说就是数据库系统模型。IBM在这方面的研究为这些模型提供了大量的基础，这些一直被用到现在。

一个早期的数据库系统设计的主要驱动力是效率。一个是系统效率更高的通用方法是强制使用固定长度的数据库记录。或至少每个记录有固定的元素（每行有固定的列数）。这可以有效的避免重复单元的问题。如果你是一个过程语言的程序员，你早就知道这种情况了，你可以将每个数据库记录读到一个简单的C数据结构中。现实世界中很少有这样的简单的情况，所以我们需要找到一些处理复杂结构的数据的方法。数据库系统设计者通过引入不同类型的数据库实现了这种需求。

#### 层次数据库模型

1960年代晚期IBM的信息管理系统介绍了数据库的层次模型。在这种模型中，把数据记录看成是其他记录的集合以解决重复单元的问题。

该模型可以用来比较描述如何制造一个复杂的产品组成的材料清单。例如，假设一辆车是一个底盘，一个机构，一个发动机，四个轮子组成。这些主要设备都进一步分解。包括一些汽缸发动机，汽缸头，曲轴。这些组件是进一步分解，直到我们得到螺母和螺栓构成的每一个汽车零件。

层次数据库模型到今天一直在使用，包括Software AG公司的适配数据库系统（Adaptable Database System，ADABAS）。

层次数据库系统有能力通过优化数据存储在某些具体问题上更加高效；例如判断哪个汽车使用一个特别的部件。

#### 网状数据库模型

网状模型在数据库中引入了指针。记录可以包含到其他记录的引用。例如，你可以为你公司的每个客户保留一条记录。每个客户随着时间的过去，给你发了很多订单（一个重复单元）。由于数据已经排序，所以客户记录只包含一个到一条订单的指针。每个订单记录包含该特定订单的订单数据，以及到其他订单记录的指针。

回到我们的货币程序，我们可以通过图1-1的样子通过记录结构终止记录。

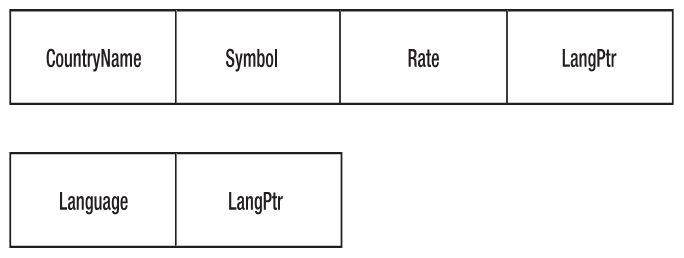


图1-1 货币程序记录类型

一旦数据被加载，我们使用一个语言用的链表（这里，就是网络模型）来终止该列信息，就像图1-2显示的一样。这里展示的两种不同的记录类型应该分别存储，各自一张表。

当然，为了在存储方面更有效率，在实际上数据库不会一次次重复语言名字，而可能包含一个第三方表用来存储语言名，并且针对语言名给一个标识符（通常是一个小整数），这个标识符用来在其他记录中指出对应的语言项目。它被叫做键。

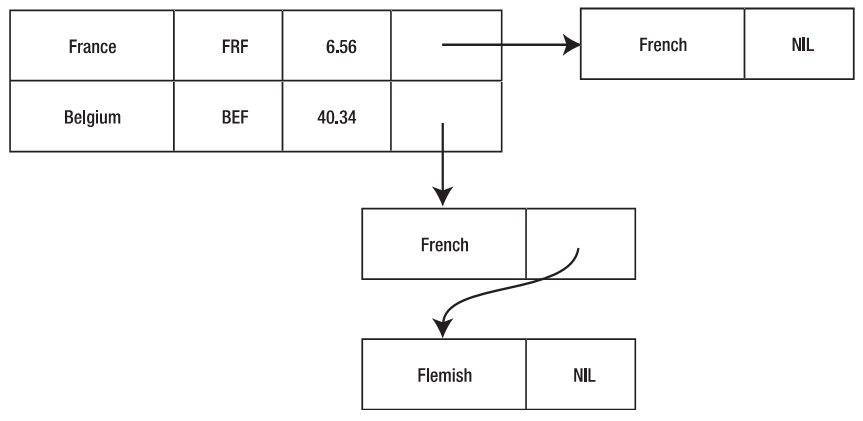


图1-2 汇率程序数据结构

一个网络模型数据库有一些很强的优势。如果你需要查找关联到某个记录的一种类型的所有的记录（本例中，一个国家的语言），你可以通过开始记录的指针快速找到他们。

当然，这也有一些不足。如果你需要列出讲法语的国家，你需要扫描所有国家的语言链表，这种情况在大数据库中非常慢。这可以通过使用另一个指向语言的指针链表来处理，但这很快就变得相当复杂，而且很明显不能作为一种通用的解决方案，因为你需要预先确定要设计多少指针。编写使用网状模型数据库的程序也非常令人厌恶，因为应用程序通常需要为记录的修改和删除而设置和管理指针。

#### 关系数据库模型

随着1970年E. F. Codd 提出 “一个大型共享数据仓库的数据的关系模型（A Relational Model of Data for Large Shared Data Banks，请阅读http://www.acm.org/classics/nov95/toc.html），数据库管理系统的理论发生了迅猛的发展。这篇革命性的文章介绍了关系的概念以及表格（通过其存储的数据）怎样用于表现现实世界的对象。

现在，一个比驱动最初的数据库设计和性能更值得关注的东西逐渐变得明晰：数据一致性。该模式比任何更多的早期模型更强调数据完整性的关系。参照完整性是指确保数据库中的数据在任何时候都合理，因此，例如，所有的订单都有客户（我们将在12章讨论数据库设计的时候更详细地讨论完整性）。

关系数据库里头的一个表里头的一条记录被认为是一个元组的组合，这是你将看到在PostgreSQL文档的某些部分使用的术语。一个元组是一个有序的组成部分或属性群，他们每一个都有一个定义的类型。

几个重要的规则定义一个关系数据库管理系统（RDBMs）。所有的元组群必须符合同样的模式，也就是它们必须拥有相同数量和类型的组成部分，以下是一个元组集的例子：

{"France", "FRF", 6.56}

{"Belgium", "BEF", 40.34}

每个元组集都有三个属性：国家名（字符串），货币（字符串）以及汇率（一个浮点数）。在关系数据库中，所有插入这个表的记录都必须使用相同的格式，所以以下的情况是不允许的：

{"Germany", "DEM"}

这里的属性太少。

{"Switzerland", "CHF", "French", "German", "Italian", "Romansch"}

这里的属性太多。

{1936.27, "ITL", "Italy"}

这里的属性类型错误（顺序错误）。

此外，对于元组集的表，不应该有重复的元组。这意味着任何正确设计的关系数据库里头的任何表格，不应该有任何重复的行或者记录。这似乎是一个相当苛刻的限制。例如，在一个记录客户的订单的系统里，它似乎不允许相同的客户订相同的产品两次。在下一章，我们会发现有一种简单的方法来解决这种需求，通过添加一个属性。

记录中的每个属性必须是原子的，也就是说，它必须是单独的一块数据，而不是另一条记录或者其他属性的列表。而且，表中的每条记录的相应属性必须相同。技术上来讲，这意味着他们必须来自于相同的值的集合或者域。实际上，它意味着他们必须是一个字符串，一个整数，一个浮点值或其他的数据库系统支持的类型。

用于从一个表中区从其他所有的记录中区分某一条特殊的记录的属性（或属性集）叫做主键。在关系数据库中，每个关系，或者表，必须有一个主键用来使一条记录唯一——也就是从表里的其他记录中区别出来。

最后一条确定一个关系数据库的构造的是参照完整性。就像我们开始说的，这要求数据库中的每一条数据在任何时候都有意义。数据库应用程序员必须确保他们的代码不会破坏数据库的这种完整性。想想我们删除一个客户的时候将发生的事情。如果我们尝试从客户表删除一条客户的信息，我们还需要从订单表删除他所有的记录。否则，我们将留下一些没有有效客户的订单记录。

我们将在后面的章节读到更多的关于关系数据库的理论。到这里，我们足以理解数据库的关系模型是基于针对集合和关系的一些数学概念，以及这种模型上的一些需要数据库系统监视的规则。

### 查询语言

关系数据库管理系统（RDBMS）诚然提供一些方法用于修改数据，但是它们真正的力量来源于它们有能力允许用户针对存储的数据以查询语句的形式提出问题。不同于很多早期的数据库设计，这些早期数据库经常针对数据需要回答的结果构造问题（查询），关系数据库对于在数据库设计时未知的问题的回答上更加灵活。

Codd关于关系数据库的提案使用以下事实：关系定义数据集，数据集能够被数学方式操作。他建议查询应该使用一个叫谓词演算的理论逻辑的分支，并且查询语言应该是它们的基础。这将使搜索和选择数据集前拥有所未有的能力。现代数据库系统，包括PostgreSQL，在一种富于表现力，易于学习的查询语言后隐藏所有的数学逻辑。

第一个实现的查询语言是QUEL，在1970年代晚期开发的Ingres数据库中使用。使用不同方法另一种查询语言是QBE（Query By Example，通过示例查询）.就在这同时，IBM的研发中心的一个团队开发了SQL（Structured Query Language，结构化查询语言），通常读作“sequel”。s

#### SQL标准和变化

SQL作为一种数据库查询语言的标准已经非常广泛地采用，并定义有一系列的国际标准定义。最通用的定义是在ISO/IEC 9075:1992中，“数据库语言SQL”中。这被简单的称为SQL92。这个标准替代了早期的SQL89标准。最新的SQL标准是ISO/IEC 9075:2003，简称为SQL2003。

当前，大多数关系数据库管理系统遵循SQL92标准，或者ANSI X3.135-1992（在美国登记的标准，只有几页封面不同）。SQL92有三个一致性级别：入门级SQL，动态SQL和完备级。到目前为止，最常见的一致性级别为入门级SQL。

注：PostgreSQL非常接近SQL92的入门级一致性，只有少量的轻微差异。开发人员保持密切关注有关标准，这使得PostgreSQL在每次发布版本后变得更加兼容标准。

今天，几乎所有有用的数据库系统支持SQL到一定程度。理论上，SQL扮演成一个很好的统一者，因为使用SQL作为数据库接口的数据库应用程序可以被移植到其他数据库，而仅仅需要花费少量的时间和精力。但商业压力决定了数据库厂商刻意制造自己的数据库与其他数据库的区别。由于SQL标准未定义现实世界中必不可少的用于执行数据库管理任务的命令，这导致了SQL的变化。所以，Oracle、SQL Server、PostgreSQL以及其他的数据库管理系统使用的SQL都不尽相同。

#### SQL命令类型

SQL语言由三种命令组成：

* 数据操作语言（Data Manipulation Language，DML）：这是在你90%时间内会使用的SQL。这些命令用来在数据库中插入、删除、更新、查询数据。
* 数据定义语言（Data Definition Language，DDL）：有些命令用来建表、定义关系以及控制数据库的其他结构方面的信息。
* 数据控制语言（Data Control Language，DCL）：这是一套通常用来控制对数据的访问许可的命令集，例如定义访问权限。很多数据库用户从来不会使用这些命令，因为这通常在大公司使用，而通常情况下会有一个或者多个数据库管理员专门管理数据库，他们的工作之一就是控制访问许可。

#### SQL命令简介

在本书中你将碰到大量的SQL。这里，我们将简单看几个例子作为介绍。我们将发现我们不需要担心我们缺少SQL基础知识而无法在这里使用。

以下有一些在数据库建立新表的SQL。以下为建立客户表的例子。

CREATE TABLE customer

(

customer\_id serial,

title char(4),

fname varchar(32),

lname varchar(32) not null,

addressline varchar(64),

town varchar(32),

zipcode char(10) not null,

phone varchar(16),

);

这张表需要一个由数据库自动生成的标记作为主键。它的类型是serial，这意味着每当一个客户被添加，一个唯一的customer\_id将被按顺序建立。客户的title是一个4字节的文本类型，zipcode有10个字节。其他的属性都是最超不超过定义长度的变长字符串，某些是必须存在的（被标记为not null）。

接下来，我们有一些SQL语句用来填充我们刚才建立的表。它们非常简单：

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone) VALUES('Mr','Neil','Matthew','5 Pasture Lane','Nicetown','NT3 7RT','267 1232');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone) VALUES('Mr','Richard','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');

SELECT语句是SQL的心脏。它用来建立一个匹配一组特定规则的记录组（或者记录的属性集）的结果集。这个规则可以是非常复杂的。这些结果集可以用来作为通过UPDATE语句修改或者通过DELETE语句删除的目标。

以下为一些SQL语句示例：

SELECT \* FROM customer

SELECT \* FROM customer, orderinfo

WHERE orderinfo.customer\_id = customer.customer\_id GROUP BY customer\_id

SELECT customer.title, customer.fname, customer.lname, COUNT(orderinfo.orderinfo\_id) AS "Number of orders"

FROM customer, orderinfo

WHERE customer.customer\_id = orderinfo.customer\_id

GROUP BY customer.title, customer.fname, customer.lname

这些SELECT语句列出所有的客户，所有客户的订单，并分别计算每个客户造出来的订单。我们将在第二章看到这些SQL语句的结果，并在第四章中学习更多关于SELECT的资料。

注：SQL命令的关键字例如SELECT和INSERT不区分大小写，所以它们可以任意使用大小写。在本书中，为了阅读方便，使用大写。

当你阅读本书的时候，我们将教你大量的，所以当你读完本书的时候，你将熟悉大量的SQL语句并知道怎么使用它们。

### 数据库管理系统的责任

正如我们前面所述，一个数据库管理系统包括构建数据库的一组程序以及使用它们的应用程序。一个数据库管理系统的职责包括以下内容：

* 建立数据库：有些系统将管理一个大的文件，并在它里面创建一个或多个数据库；其他系统可能使用许多操作系统文件或直接使用裸磁盘分区。用户不必担心这些文件的低层次结构，因为数据库管理系统提供了开发者和用户访问所需要的所有功能。
* 提供查询和修改能力：一个数据库管理系统将拥有一个请求符合特殊规则数据方法，例如某个客户的所有未完成的订单。在引入SQL标准之前，每个数据库系统查询这种结果的方法都大不相同。
* 多任务：如果数据库 被多个应用程序使用，或者同时被多个用户访问，数据库管理系统需要确保每个用户的请求被处理而不影响其他的用户。这意味着用户需要在其他人正在写入他需要读取（或者写入）的数据时排队等待。它允许在同一时间有很多并发读取。实际上，不同的数据库系统支持不同基本的多任务，甚至允许可配置的级别，就像我们在第九章讨论的。
* 维护一个审计线索：一个数据库管理系统将保存一个近段时间内数据库中所有变动的日志。它可以用于调查错误，但其实可能更重要，可以用于在系统故障（例如一个非计划的断电）后重建数据。数据备份和事务的审计线索可以用于在磁盘故障后完全恢复数据库。
* 管理数据库的安全性：一个数据库管理系统将提供访问控制，因此只有授权的用户可以操作数据库中的数据以及维护数据库本身（的属性，表和索引）的结构。通常情况下，每个数据库都会通过一个可以改变任何东西的超级用户定义不同级别的用户，从可以添加删除数据的用户到只可以读取数据的用户。数据库管理系统架需要有能力添加和删除用户，并指定数据库的哪种功能这个用户可以使用。
* 维护参照完整性：就像前面所说，很多数据库提供用于维护参照完整性（数据的正确性）的功能。它将在一个查询或者修改会破坏关系模型规则的时候报告一个错误。

## 什么是PostgreSQL?

现在是时候告诉你到底PostgreSQL是什么了。它是一个包含关系模型和支持SQL标准查询语言的DBMS（数据库管理系统）。

PostgreSQL也非常先进和可靠，并且性能非常高。它基本上可以在任何UNIX平台上运行，包含类UNIX系统，比如FreeBSD、Linux和Mac OS X。它也可以在Microsoft Windows NT/2000/2003服务器版本上运行，甚至可以在Windows XP上进行开发。并且，就像本章开始提及的，它免费且开源。

PostgreSQL可以与其他DBMS媲美。它基本上包含其他所有商业的或开源的数据库中你能找到的功能，甚至一些你找不到的功能。

PostgreSQL 包含以下功能（在PostgreSQL FAQ里面列出的）：

* 事务
* 子查询
* 视图
* 外键参照完整性
* 复杂的锁
* 用户自定义类型
* 继承
* 规则
* 多版本并发控制

从版本6.5开始，PostgreSQL就非常的稳定，这通过针对每个发布版本都通过大量的回归测试得以保证。7.x发布版比以前版本更接近SQL92标准，并去掉了令人讨厌的行数限制。

本书中使用的PostgreSQL，8.x版本，增加了以下新功能：

* Microsoft Windows原生支持
* 表空间
* 可以修改列类型
* 时间点恢复

PostgreSQL被证实在使用中非常可靠。每次发布都被严格控制，BETA都经过至少一个月的测试。通过庞大的用户社区和普及的源码访问，BUGs很快就被修复了。

PostgreSQL的性能在每次发布都有提升，最新的基准测试显示在某些条件下，它可以同商业产品相媲美。一些非全功能的数据库系统性能会比它高出一些，因为这些数据库在没有全功能带来的性能损耗。当然，对于足够简单的应用，可以使用扁平文件的数据库系！

### PostgreSQL历史简介

PostgreSQL可以回溯它的家族树到1977年的加州大学伯克利分校（University of California at Berkeley，UCB）。一个叫做Ingres的关系数据库由UCB在1977到1985年开发。Ingres是一个著名的UCB产出，在很多学院和研究团体的UNIX计算机上崭露头角。为了服务于商业市场，Ingres的源码由Relational Technologies/Ingres公司加以改良并成为第一个商用RDBMS。

注：今天，Ingres已经变成了CA-INGRES II，一个Computer Associates公司的产品。它最近也变成了开源产品。

同时，在UCB，从1986到1994年，一个名叫Postgres的关系数据库服务器被继续开发。同样，这份代码被Illustra公司拿去发展成一个商业化的产品（Illustra 公司后来被 Informix 公司并购，程序则整合到 Informix的 Universal Server 去了）。在1994年，Postgres被添加了SQL功能，且被易名为Postgres95。

到1996年，Postgres开始变得非常流行，开发人员决定开放它的开发到一个邮件列表，成功的实现了志愿者驱动的Postgres发展。这时候，Postgres经历了它的最后一次更名，替换“95”标记为更恰当的“SQL”，反映出它现在支持的查询语言标准。PostgreSQL诞生了。

现在，一个因特网上的开发团队使用其他开源软件像Perl，Apache和PHP的同样的方法开发着PostgreSQL。用户可以访问源码并贡献修复代码、增强功能和建议新功能。PostgreSQL的官方发布网站为<http://www.postgresql.org>。

已经有很多公司提供商业支持了。例如EnterpriseDB公司，期网站地址为：<http://www.enterprisedb.com>。

### PostgreSQL架构

PostgreSQL强壮的一个原因源于它的架构。和商业数据库一样，PostgreSQL可以用于C/S（客户/服务器）环境。这对于用户和开发人员有很多好处。

PostgreSQL安装核心是数据库服务端进程。它允许在一个独立服务器上。需要访问存储在数据库中的数据的应用程序必须通过数据库进程。这些客户端程序无法直接访问数据，即使它们和服务程序在同一台机器上。

注：PostgreSQL还不具有一些企业级商业数据库的负载均衡和提供扩展的可伸缩性和可恢复性的HA（High-Availability，高可用性）功能。在http://gborg.postgresql.org有一些PostgreSQL认可的项目针对增加这些功能在进行中，同时也有一些商业解决方案存在。

这种分开为客户端和服务器端的方式可以让应用程序分布式允许。你可以使用网络来分隔你的客户端和服务端，使开发的客户端程序适合用户的使用环境。例如，你可以在UNIX上实现数据库并建立运行在在Microsoft Windows上的客户端程序。图1-3显示一个典型的分布式PostgreSQL应用。

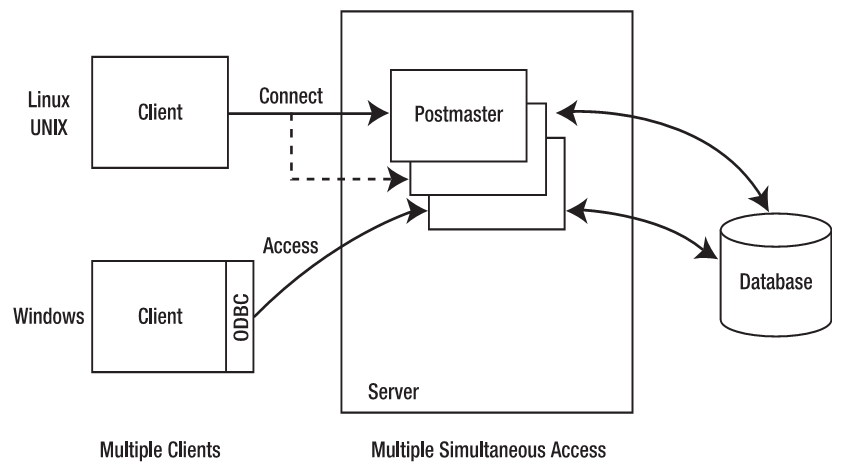


图1-3 PostgreSQL 架构

在途1-3中，你可以看到几个客户端通过网络连接到服务器。对于PostgreSQL，这需要是一个TCP/IP网络——一个局域网（local area network，LAN）或者甚至是因特网。客户端连接到数据库主进程（在图1-3中叫做postmaster），主进程为这个客户端的访问请求专门建立一个新的服务端进程提供服务。

服务器专注于数据处理，而不是尝试控制很多客户端访问服务器上共享目录中存储的数据，这让PostgreSQL高效的管理数据的完整性，即使在存在大量的用户的情况下。

客户端程序使用专有协议连接到PostgreSQL。然而，通过在客户端安装软件而为应用程序提供一个标准的接口是可能的，例如开放式数据库连接（ODBC，Open Database Connectivity）标准或者Java程序的Java数据库连接（JDBC）标准。ODBC驱动的存在运行很现存的应用程序使用PostgreSQL作为一个数据库，包括的微软的Office产品例如Excel和Access。你将在第三、第五和第十三到十八章了解不同的PostgreSQL连接方法。

PostgreSQL的C/S架构允许任务分工。非常适合于存储和访问大量数据的服务器主机可以用作安全的数据储存库。可以为客户端开发复杂的图形界面程序。另外，基于网页的前端可以通过建立网页模式的结果集到浏览器访问数据，而不需要另外的客户端软件。我们将在第五章和十五章回头讨论这些想法。

### 通过PostgreSQL访问数据

通过PostgreSQL，你可以通过好几种方法访问你的数据：

* 通过命令行程序执行SQL。本书自始至终都会用到这种方法。
* 直接嵌入SQL到你的应用程序（使用嵌入式SQL）。我们将在14章讨论怎样在C程序中使用它。
* 大量不同的程序语言使用功能调用（APIs）准备和执行SQL，扫描结果集以及执行更新。13章将覆盖PostgreSQL的C语言API。
* 通过ODBC（参考第3章）或者JDBC（参考17章）标准的驱动间接访问PostgreSQL数据库，或者使用标准库，例如Perl的DBI库（参考16章）。

## 什么是开源？

21世纪开始的时候，开源软件提供了大量的东西，PostgreSQL就是一个很好的例子。但是开源到底意味着什么呢？开源的条款在施加到软件上时有非常明确的意义。这意味着软件在提供的时候，同时提供了源码。这并不意味着没有其他条件被施加到软件的使用上。在某些情况下使用（开源）软件，一样需要得到许可。

开源许可赋予你使用、修改和重新发布它而不需要付许可费用的权力。这意味着你可以在你的机构中觉得合适的地方使用PostgreSQL。

如果你的开源软件有一些问题，因为你有源码，你既可以自己修复它，也可以将代码交给别人来帮忙修复。有很多商业公司提供对开源软件的支持，所以如果你选择使用开源产品，你不必感到被忽视。

有很多种开源许可的变种，一些比其他的更自由。但所有的都支持提供源码和允许重新发布。

最自由的是BSD软件许可（Berkeley Software Distribution），它提供“随便你怎么处置这个软件。”

最宽容的许可是伯克利软件分发（BSD，Berkeley Software Distribution）许可，它实际上说“用这个软件做任何你想做的。但不提供任何担保”。PostreSQL使用的软件许可（<http://www.postgresql.org/about/licence>）响应BSD软件许可的精神并套用它的说法，“允许以任何目的使用、拷贝、修改和重新发布这套软件以及文档，不需要任何费用，不需要签订任何书面协议，提供以上的版权通告以及这段和以下两个段落的文字在所有的拷贝中”。紧接着的段落时免责声明和保证书。

## 相关资源

有很多关于数据库的以及关于PostgreSQL的进一步信息的印刷的和在线资源。

对于更对的关于数据库的理论，可以查看位于<http://www.frick-cpa.com/ss7/default.htm>的David Frick的网站的数据库理论的章节。

PostgreSQL的官方网站位于<http://www.postgreSQL.org>，在这里你可以找到更多的关于PostgreSQL的历史，下载PostgreSQL的拷贝，浏览官方文档以及更多的东西（包括学习怎样发布PostgreSQL）。

PostgreSQL也是Red Hat Database的基础，也就是PostgreSQL- Red Hat版。你可以在在Red Hat的网站的http://sources.redhat.com/找到更多的关于PostgreSQL以及Red Hat为它开发的工具。

如需有关开放源码软件和自由软件原则的更多信息，请花点时间访问以下两个网站：<http://www.gnu.org>和<http://www.opensource.org>。

# 关系数据库原理

在本章中，我们将研究什么会组成一个数据库系统，特别是PostgreSQL这样的现实世界中非常有用的关系数据库系统。我们将从电子表格开始，它和关系数据库有很多相同的地方，同时也有重大的局限性。我们将学习像PostgreSQL一样的关系数据库怎么拥有比电子表格强悍的很多功能。顺带，我们将继续我们的非正式的对SQL的研究。

特别是，本章将讨论以下主题：

* 电子表格：它们的问题和局限
* 数据库怎样存储数据
* 怎样访问数据库中的数据
* 基本的数据库设计，用到多个表格
* 表之间的关系
* 一些基本数据类型
* NULL值，用来表示一个未知的值

## 电子表格的局限性

电子表格软件，例如Microsoft Excel，被广泛应用于数据存储和查阅。它很容易用不同的方法排序数据，并通过眼睛直接观察数据的内容和模式。

不幸的是，人们经常误解一个很好的用于查看和操作数据的工具为一个存储和共享复杂的甚至是关键业务数据的工具。这两种需求通常相差甚远。

很多人都熟悉一种或多种电子表格软件，并能够按照一定的行列的规则组织一些数据。图2-1显示了一个示例——一个保存客户数据的OpenOffice（<http://www.openoffice.org/>）电子表格。

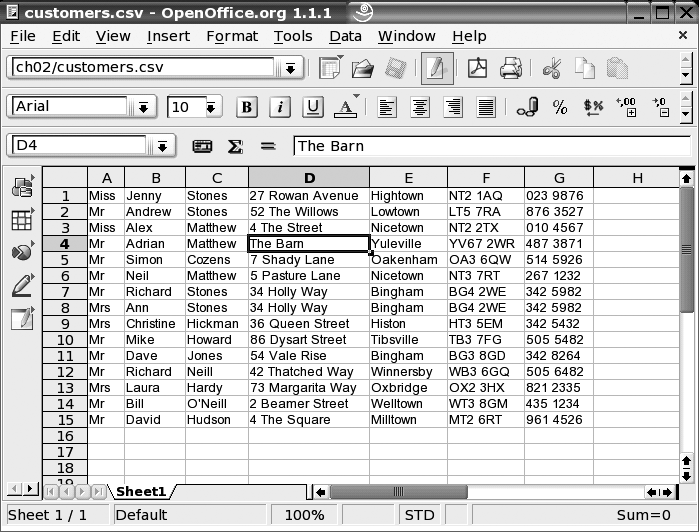


图2-1 简单的电子表格

当然，这样的信息容易被看到和修改。每个客户有单独的一行，而且客户的每段信息保存在一个单独的列中，就像图2-2中列出来的一样。一列和行的交集是一个单元格。

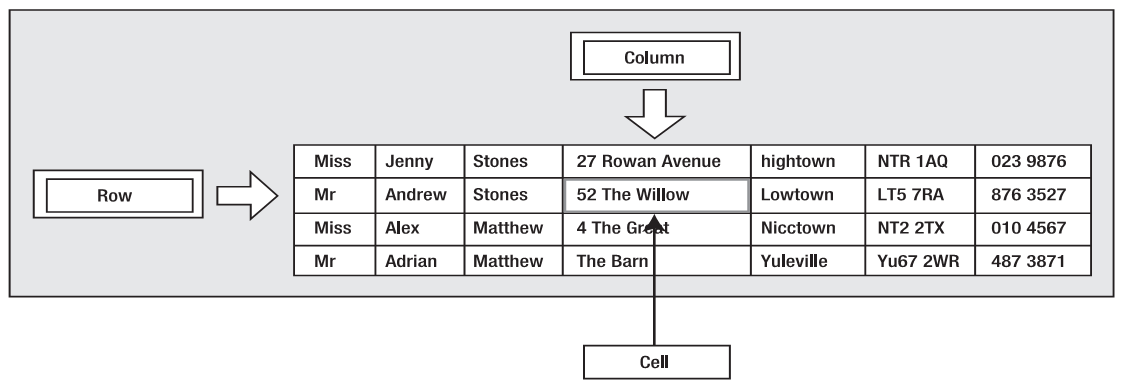


图2-2 一些电子表格的术语

这个简单的电子表格包含需要在我们开始设计数据库时顺手记下的几个功能。例如，姓和名分别在不同的列，这使按性和名排序的时候更容易。

所以，在电子表格里头保存客户信息有什么错？电子表格很好，因为你：

* 没有太多客户
* 每个客户没有太多的复杂细节
* 不需要保存任何其他的重复信息，例如若干客户的订单
* 不需要很多人同时更新信息
* 可以确保保存重要信息的电子表格定期得到备份

电子表格式一个了不起的主意，而且是一个可以解决很多类型问题的很好的工具。但是，就像你不会（或者至少不应该）尝试用牛刀杀鸡一样，有些时候电子表格你完成工具的恰当工具。

试想如果在一个有用成千上万客户的大公司将所有的客户信息保存在一张电子表格的主副本中将会怎样。在一个大公司，很可能很多人需要更新这个列表。虽然文件锁可以确保同一时间只有一个人可以修改这个列表，随着需要修改这个文件的人数的增多，他们需要花费更长的时间等待轮到自己修改这个列表。我们所想要的是让许多人同时读取，更新，添加和删除行，并让计算机确保没有冲突。显然，简单的文件锁定将不足以有效地处理这个问题。

电子表格的另一个问题是它们是绝对的二维的。假设我们还要存储每个客户的订单的细节。刚开始我们可以将订单细节紧贴着客户信息存放。当随着每个客户订单信息的增长，电子表格会变得越来越复杂。考虑到这样的结果，我们开始为每个客户添加一些基本订单信息，如图2-3所示。

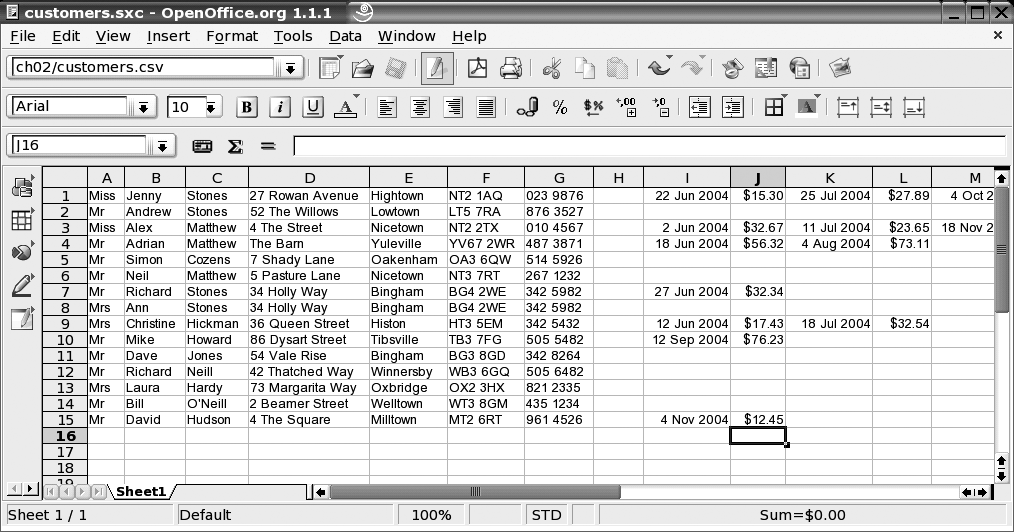


图2-3 保存有多条订单信息的电子表格

不幸的是，它看上去不在那么优雅了。我们现在又变长的行，且没有容易的方法计算么个客户会占用多少。甚至，我们会达到每条记录允许的最大的列数。这就是我们在前面章节碰到的重复组的问题。在电子表格中的多表格会有点帮助，但他们不是解决这个问题的理想方法。

电子表格受到的挑战

这是一个你怎样很容易超过电子表格限制的例子。一个熟人正尝试建立一个电子表格用来帮做小型生意的朋友存储信息。这个生意需要记录皮革项目，每个项目的价格不单依赖于生产它所需要的时间和劳动，同时还依赖于制造过程中所耗费的皮革的单位成本。物主需要购买不同批次的皮革，每个批次需的单价依赖于皮革的档次和购买时间。然后他们需要在他们制造皮革制品的时候在库存中使用先进先出的策略，通常皮革是按批购买的。我们的挑战是建立一个电子表格完成以下功能：

* 跟踪当前的库存价格
* 跟踪有多少批次不同级别的皮革在仓库中
* 跟踪当前被使用在生产某个项目上的批次和级别的皮革已经支付了多少钱

经过几天的努力，他们发现这种表面上简单的仓储管理需求出人意料的难以转换到一个电子表格。仓储记录多变的特征不适合电子表格的理念。

在这里我们想指出的是电子表格有其合适的位置，但也有其使用的限制。

## 将数据存入数据库

从表面上看，关系数据库，例如PostgreSQL，拥有很多类似于电子表格的地方。但是，当你了解数据库的底层结构，你可以发现它复杂得多，主要因为它有能力通过复杂的方法将表格关联到一起。它可以比电子表格有效地存储更多复杂的数据，并且它有用很多其他功能方便选择存储的数据。例如，数据库可以管理多个用户同时使用。

让我们看看存放我们简单的但表格客户列表到数据库，看这么做有什么好处。在后面的章节，我们将扩展它并看PostgreSQL怎么帮助我们解决客户订单的问题。

就像我们在前面章节看到的，数据库由表（tables）组成，或者用更正式的术语，关系（relations）。我们将在本书中使用表这个术语。表里头包含数据行（更正式的叫法是元组（tuples）），并且每条数据行都包含许多列（columns），或者叫做属性（attributes）。

首先，我们需要设计一个表来保存我们的客户信息。好消息是，电子表格的数据往往是一个几乎现成的解决方案，因为它按照一定的行和列保存数据。在开始建立一个基本的数据库表格前，我们需要确定三件事情：

* 我们需要多少个列来存储每个项目的属性？
* 每个属性（列）的数据类型是什么？
* 我们怎么区别不同的行包含的不同项目？

注意数据库的表的每行的顺序不影响数据。在单独的数据表格中，行的数据可能非常重要，但在数据库的表中，没有顺序。因为当你查看数据库的表存储的数据的时候，数据库可能会随意按照它选择的数据的顺序将数据给你，除非你特别告诉它你要按特殊的顺序排序数据。如果你需要按特殊的顺序查看数据，可以通过指定获取数据的顺序，而不需要例会它存储的方式。我们将在第四章的SELECT语句的ORDER BY从句中了解到怎么按顺序获取数据。

### 选择列

如果你回顾下图2-1中原始的客户信息电子表格，你会发现我们已经确定每个客户需要的合理列：名，姓，邮政编码等。所以，我们已经回答了我们应该有多少列的问题。

电子表格的行和数据库中的行最重要的不同是数据库里头的表格的列数对于所有的行都是相同的。这对于我们原始的电子表格的不是问题。

### 为每个列选择数据类型

第二步是为每列的数据确定类型。电子表格中允许每个单元格拥有不同的类型，在数据库的表中，每个列的类型必须相同。就像大多数编程语言一样，数据库使用类型来标记不同的数据值。平常，你需要知道所有的基本类型。主要的选择可以是整数，浮点数，定长文本，变长文本和日期。通常最容易的判断恰当的数据类型的方法是看看示例的数据。

在我们的客户数据中，所有的列的类型都可以是文本类型，即使电话号码看上去好像都是数字。将电话号码作为数字存储通常存在以下问题：很容易丢失前导的零，并阻止我们存储国际拨号符（+），不允许在区号前后写上括号等等。显然，电话号码远不止一串数字。回过头来，用字符串存储电话号码可能不是最好的选择，因为我们可能无意地插入各种其他字符，但至少会比使用数字类型要好点。初始设计可以在之后做优化。

我们会发现头衔（女士，先生，医生）的长度通常比较短——通常少于四个字符。类似的，邮政编码也有固定长度。因此，我们可以设置这些列为固定长度，但设置其他的所有列为变长的，因为比方说没有简单的办法判断一个人的名字有多长。

我们将在本章靠后的“基础数据类型”小节以及第八章讨论PostgreSQL的数据类型。

### 标记行的唯一性

我们在转换我们的电子表格到数据库表格的最后的问题有点微妙，它牵涉到数据库管理表和表之间的关系。我们需要确定什么使数据库中一条客户数据记录区别于另一条客户记录。换句话说，我们怎么区别我们的客户？在电子表格中，我们不趋向于关心区别客户的细节。但是，在数据库设计中，这是一个关键问题，因为关系数据库的规则需要从某个方面区别每条记录的唯一性。

最明显的解决方案好像是通过客户名来区别客户，但不幸的是，这通常不足以区别。因为很有可能两个客户拥有同样的名字。另外一项你可以选择的是电话号码，但问题是如果两个客户住在于一起呢？在这个时候，你可能建议使用名字和电话号码的组合。

当然，不大可能两个客户同时拥有相同的名字和电话号码，但是，这种方法不但很不雅观，而且还有另一个潜在问题。如果客户更换电话服务商，电话号码发生变化，将发生什么。在我们的定义中，唯一的客户必须是一个客户。因为他不同于我们已有的客户。当然，我们知道它是有不同的电话号码的旧客户。在数据库中，选择一个可能会变动的功能作为客户的唯一标识是个坏习惯，因为管理唯一标识的变动非常麻烦。

这种唯一标识的问题，经常出现在数据库设计中。我们应该做的是寻找一个主键（primary key）——一种最容易的用于区别一个客户的数据行于其他所有行的方法。遗憾的是，我们还没有成功，但所有的也没有失败，因为标准的解决方案是为每个客户分配一个唯一的数字。

我们简单地为每位客户分配一个唯一的数字，然后，爽，我们有一个明显的方法区别客户，而不管他们是否改变电话号码或者换到其他的住所，甚至改变他们的名字。这种在实际数据中没有办法选择其他列而附而加的作为一个唯一的键值的键，叫做代理键。在现实世界数据中即使存在一些特殊的数据，数据库中叶经常这样做，提供序列数据类型来帮助解决这类问题。我们将在后面的“基础数据类型”小姐讨论这个类型。

我们已经完成为我们的初始表格做一个数据库设计，现在是时候存储我们的数据到数据库中了。图2-4展示了在PostgreSQL数据库中我们的数据在windows或者Linux主机的终端窗口中通过一个简单的命令行工具psql显示出来的样子。

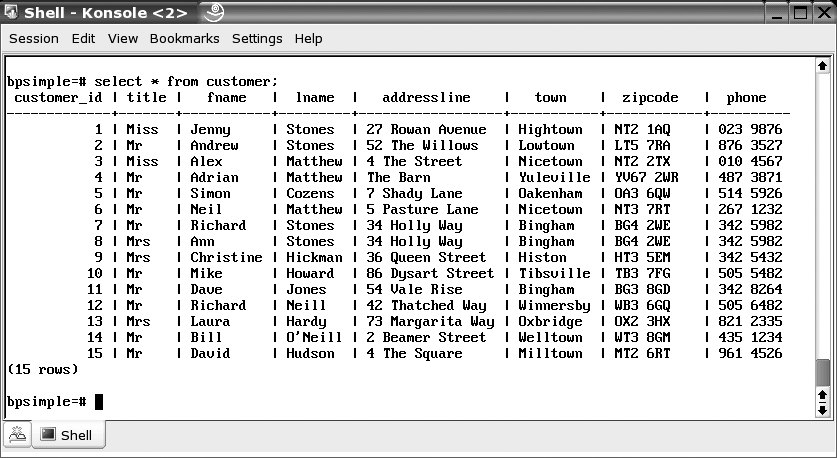


图2-4 通过命令行从数据库中查看我们的数据

注意我们添加了附加的一列，叫做customer\_id，作为我们参考客户的唯一方式。它是这个表中我们的主键。就像你所看见的，数据看上去就像一个电子表格，按行列布局。在后面的章中，我们将讲解在实际中定义数据库表格，存储和访问数据，但我们可以确信这些都不难。

## 在数据库中访问数据

你可以很容易地通过命令行工具psql查看你在PostgreSQL中的数据，如图2-4所示。但是，PostgreSQL不仅限于在命令行中使用。图2-5显示更友好的图形界面工具pgAdmin III，它是可以从<http://www.pgadmin.org> 获得的免费工具，它从8.0开始已经和Windows的PostgreSQL捆绑发布了。我们将在第五章看到更多的图形界面接口。

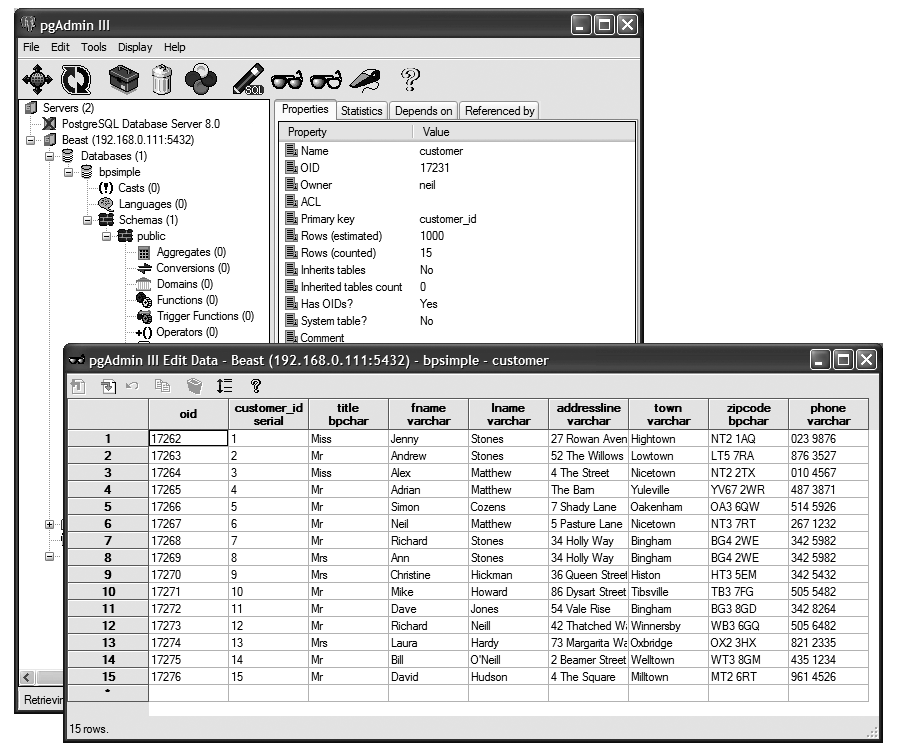


图2-5 通过pgAdmin III查看数据库中的客户数据

### 通过网络访问数据

当然，如果我们只可以在存储数据的同一台机器访问我们的数据，这情况和通过共享单个的电子表格给不同的用户没有太多改观。

PostgreSQL是一个机遇服务器的数据库，就像前面章节所说的，一旦配置完成，我们可以通过网络接受客户的请求。虽然客户端可以和数据库服务器在同一台机器，对于多用户访问来说，这实在是小菜一碟。对于微软的Windows用户，因为有ODBC驱动，所以我们可以使用任何支持ODBC的Windows桌面应用程序连接到保存我们数据的服务器。图2-6显示Windows上的微软Access软件访问一个在Linux主机上的PostgreSQL数据库。这就是通过ODBC连接经过网络访问的外部表。

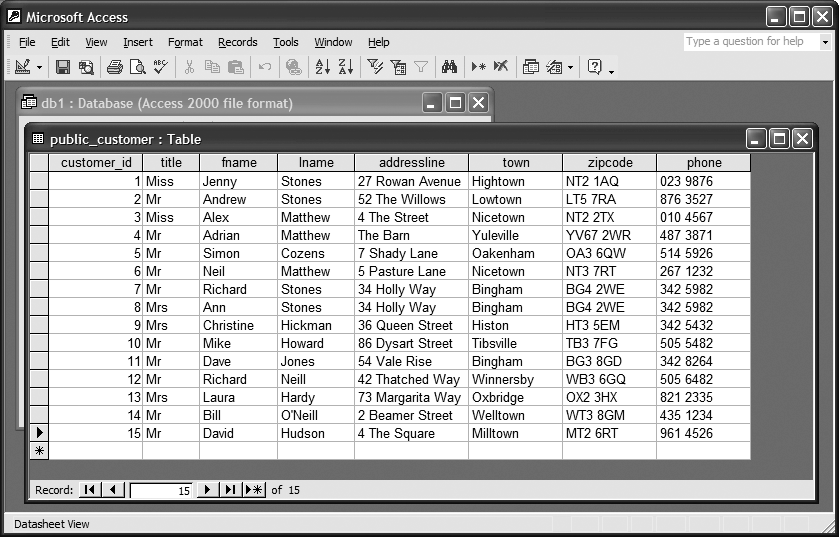


图2-6通过微软的Access访问以上相同的数据

现在我们可以同时从很多机器通过网络访问这些相同的数据。我们只有一份数据，保存在中央服务器中，可以通网络从过多个桌面程序同时访问。

我们将在第五章了解配置ODBC连接的技术细节。

### 处理多用户访问

和所有的关系数据库一样，PostgreSQL可以自动确保对数据库数据修改的冲突不会发生。它确保各个用户在使用数据的时候感觉好像访问全部的数据都不受限制，但在幕后，PostgreSQL监视着改变且避免同时更新的冲突。

这种表面上让很多用户可以同时读写同样的数据，但实际上确保其一致性的能力，对于数据库来说是非常重要的功能。当一个用户修改了一列，你要么看到它变化前的样子要么是变化后的样子，从不会看到修改一半的样子。

一个经典的例子是银行的数据库在两个账户间转账。如果在转账的时候，一些人正在生成一个汇报所有金额的报表，确保综述正确就非常重要了。对于报表来说钱在哪个账户在报表生成的时候无关紧要，但重要的是报表无法看到中间点，也就是一个账户计入借方但另一个账号还没计入贷方的时候。

像PostgreSQL一样的关系数据库都隐藏了任何中间状态，所以中间状态不会被其他用户发现。术语上说这叫隔离。报表操作从转账操作隔离开来，所以它看上去是在其之前或者之后发生，但绝不会同时发生。我们将在第九章讨论事务的时候回顾隔离的概念。

### 数据分片和分块

我们现在知道了当数据存在于表中后，访问它是多么容易。让我们首先看看我们实际上应该怎么处理数据。我们通常在大的数据集上会执行两类基本操作：选择符合特定值的集合的行和选择数据的部分列。在数据库术语中，他们分别叫做选择和投影。它们听上去有点复杂，但实际上选择和投影都非常简单。

#### 选择

让我们从选择开始，也就是我们选择行的子集。假设我们想知道我们住在Bingham的客户。让我们回到PostgreSQL的标准命令行工具psql来看我们怎么使用SQL语言让PostgreSQL获得我们需要的数据。我们要用的SQL命令非常简单：

SELECT \* FROM customer WHERE town = 'Bingham';

如果你键入你的SQL语句（通过命令行的工具psql），你需要在末尾加入一个分号。分号告诉psql已经到达命令的末尾了，因为很长的命令可能扩展到不止一行。在本书中我们通常会显示分号。

PostgreSQL通过返回customer表中所有的town为Bingham的行作为响应，就像图2-7所示。

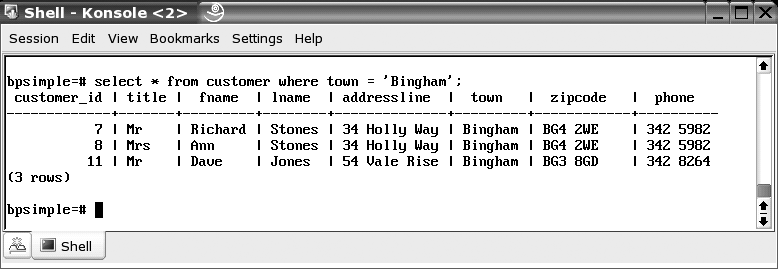


图2-7选择数据行的子集

所以，挑选表中某些行的行为就叫做选择。就像你所看到的，这非常容易。不用担心SQL语句的细节，我们将在第五章正式的回顾它。

#### 投影

现在我们来看看投影，也就是选择表中的某些列。假设我们仅仅需要选择客户表中的姓名。请记住我们分别把这两个列叫做fname和lname。选择名字的命令也非常简单：

SELECT fname, lname FROM customer;

PostgreSQL通过返回恰当的列作为响应，如图2-8所示：

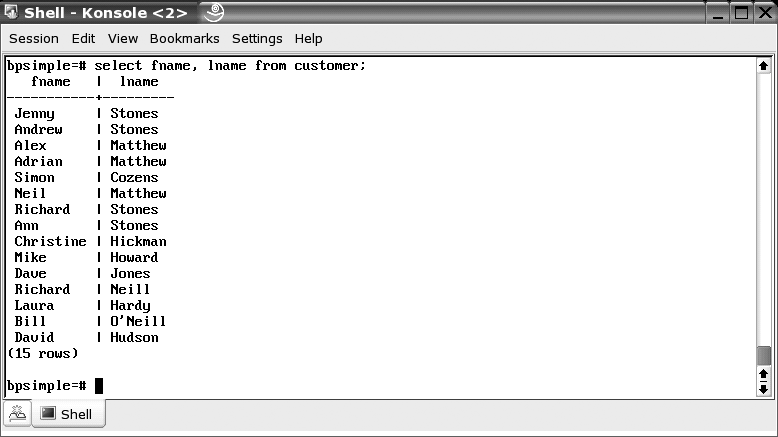


图2-8 选择数据列的子集

你当然可以假设某些时候我们需要在数据中同时执行以上两种操作；也就是说选择某些行中的某些列。这也可以通过非常简单的SQL实现。例如，假设我们需要知道住在Bingham的客户的姓名，我们只要简单的将以上两条SQL语句组合成一条简单的语句：

SELECT fname, lname FROM customer WHERE town = 'Bingham';

PostgreSQL如图2-9一样响应我们的请求：

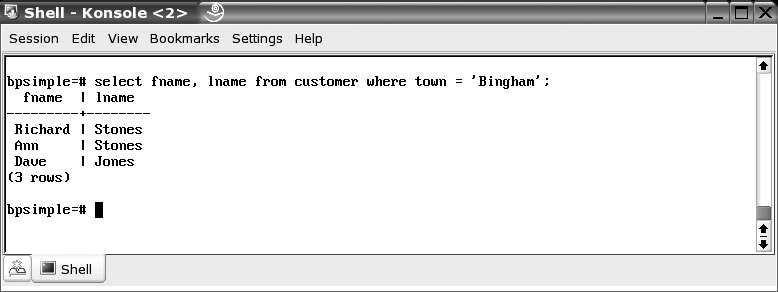


图2-9同时选择行和列的子集

有个东西需要非常注意。在很多传统编程语言例如C或者Java中，当在文件中查找数据的时候，我们需要编写一些代码扫描文件中的所有行，并每次找到我们需要的城镇的时候，打印出名字。虽然可以挤压代码让他们变成一行代码，它将是非常长和复杂的一行，不像这里的SQL那么简洁。这是因为C和Java等类似的语言从本质上说还是一种过程语言。

你在这些语言中指出计算机怎么工作，对于SQL，用术语讲这是一种描述性语言，你告诉计算机你要什么，PostgreSQL通过某些内部逻辑处理你要的任务。

如果你从没有使用过描述性语言你也许会觉得有点古怪，但是如果你开始使用这种想法，你会发现告诉计算机你要什么很明显是一种很好的想法，而不是告诉它要怎么做。你会觉得很奇怪为什么到现在为止你才遇到这么好的语言。

### 增加信息

到现在为止我们所碰到的都是通过我们的数据库模拟电子表格的单个工作表，而且我们也刚刚接触到SQL功能的表皮。就像我们将要在本书中看到的，PostgreSQL一类的关系数据库富有大量的有用功能，这让他们的工作能力大大超越了电子表格。其中一个数据库最重要的能力就是它们有能力将表与表之间的数据连接在一起，这就是我们现在将要读到的。

#### 使用多重表格

回到我们关于客户订单的问题，也就是在为每个客户存储扩展订单信息时突然让我们的简单客户表格变得非常凌乱的问题。在我们开始不知道我们的客户会有多少订单的时候要怎么存储客户的订单？你也许会从本章的标题就能猜测到，在关系数据库中解决这种问题的办法是增加一个表格存储这些信息。

就像我们设计客户表一样，我们从确定我们要存储的每个订单的信息开始。现在，让我们假设我们要存储下订单的客户的名字，下订单的日期，订单发出日期以及发货方式。和customer表一样，我们也需要为每条订单添加一个唯一参考数字，而不是假设哪个信息可能是唯一的。没有必要再存储所有的客户细节了。我们已经知道通过customer\_id，我们可以在customer表中找到客户的细节。

你也许觉得奇怪为什么我们忽略掉了订单的细节。当然，对于大多数客户，这是一个很重要的方面——他们想要知道他们订了什么内容。如果你认为这是一个和不知道客户有多少订单一样类似的问题，你非常正确。我们不知道每个订单有多少项目。重复组的问题还没远离你。我们将暂时把这个问题放下并在后面的“进行简单的数据库设计”章节解决这个问题。

图2-10显示了我们的订单信息表，显示了一些示例数据。而且是通过图形界面的pgAdmin III工具显示的。

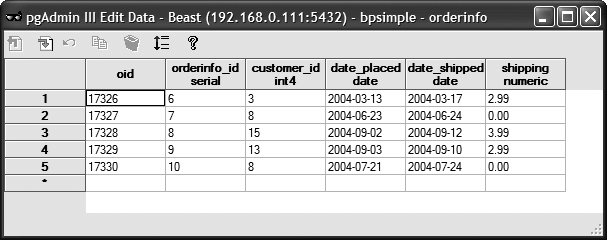


图2-10 在pgAdmin III中显示的一些订单信息

我们没有在这张表中放太多数据，因为少量的数据更容易做实验。你将发现一个扩展的列，叫做oid，它不属于我们的用户数据。这是PostgerSQL内部使用的一个特殊列。当前版本的PostgreSQL默认为表建立这个列，但在“SELECT \*”命令中它是被隐藏的。我们将在第八章讨论这个列。

#### 通过关联（Join）操作关联一个表

我们现在在数据库中存有我们客户的细节，以及它们订单的概要细节。在很多情况下，这和使用两张电子表格没有区别：一张存储客户细节，另一种存储订单细节。现在是时候关注我们怎么组合使用这两张表，开始发觉数据库的能力了。我们可以同时从这两个表选择数据。这就叫做关联（Join），在选择和投影之后，这是第三类最常用的资料检索操作SQL。

假设我们需要列出所有的订单以及下订单的客户。在像C一样的过程语言中，我们需要编写代码扫描其中一个表，也许从customer表开始，然后为每个客户打印他们的订单。这不难，但编写这些代码当然也非常费时和乏味。我敢保证你会很乐意知道我们可以通过SQL更容易的找到答案：通过关联操作。我们所要做的是告诉SQL三件事情：

* 我们需要的列
* 我们需要检索数据的表
* 这两个表之间怎么关联

我们需要的命令在前面的章节中出现过：

SELECT \* FROM customer, orderinfo WHERE customer.customer\_id = orderinfo.customer\_id;

你也许会猜想，它将从两个表中请求所有的列，并告诉SQL customer表的customer\_id列保存的信息与orderinfo表的customer\_id保存的信息一样。注意方便的table.column标记让我们让我们有能力同时指出表名和这个表的列。命令中的“\*”意味着所有的列。我们可以使用列的名字代替它来选择一些特殊的列，例如假设我们只需要名字和数量。

现在我们的数据库已有一些表格和数据，我们可以在图2-11中看看PostgreSQL的回应。

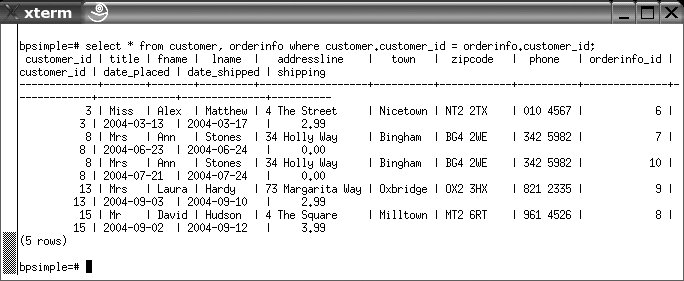


图2-11 通过一次操作从两个表中选择数据

这有点点凌乱，因为为了匹配窗口，数据被自动换行了，但是你可以看到PostgreSQL怎么响应我们的查询，而不需要我们指出怎么精确地解决问题。

再往前走一点点，看看我们可以用在这两个表上的义工更复杂的查询。假设我们想看看不同客户给我们下订单的频率。很明显这需要一个高级一点的SQL：

SELECT customer.title, customer.fname, customer.lname,

COUNT(orderinfo.orderinfo\_id) AS "Number of orders"

FROM customer, orderinfo

WHERE customer.customer\_id = orderinfo.customer\_id

GROUP BY customer.title, customer.fname, customer.lname;

这是个复杂点的SQL，你可以发现我们不需要告诉SQL怎样回答问题的细节；我们只要通过SQL精确地指出问题。我们也只需要子一条简单的语句中完成了它。图2-12显示了PostgreSQL怎么响应。

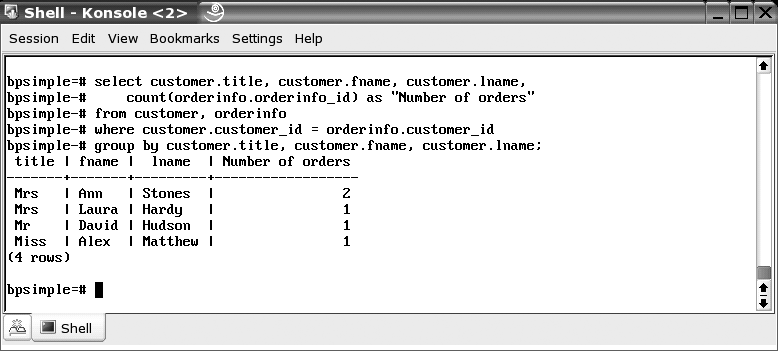


图2-12 查看订单数量

一些数据库专家喜欢通过直接在命令行工具的窗口中直接敲入SQL，它在某些时候非常有用，但它并不是所有人的爱好。如果你喜欢通过图形界面建立你的查询，这也不是问题。例如就像在本章早些时候指出的，你可以简单地通过ODBC驱动程序通过Windows的图形用户界面（GUI，Graphical User Interface）访问数据库。图2-12显示在Windows机器中的Acccess软件通过PostgreSQL的ODBC驱动程序连接到外部的表格，设计和执行的相同的查询。我们也会在第五章看到其他的GUI工具.。

在我们这个环境中，数据仍然存储在一台Linux主机中，但用户基本不需要知道这些技术细节。通常，在本书中，我们会在命令行中使用教学SQL，因为在进入更复杂的SQL命令前，你可以在这里学到很多基础知识。当然，也欢迎您使用GUI工具而不是命令行工具来构建你的SQL；这是你自己的选择。

### 设计表

到现在为止，我们在数据库中只有两个表，而且我们还没有真正讨论过关于我们怎么在每张表中做什么，除了在一些看似有理的非正式设计。这种包含表、列和关系的设计，正确的说法应该叫做模式（schema）。

如果数据很复杂，设计拥有数十个表的数据库模式将相当的有挑战性。数据库设计者因为善于完成这类工作而赚大把的钱。对于不到十个表格的关系简单的数据库，也许只需要按照一些基本的经验规则就可以得到很好的设计，而不需要非常正式的规则。

在本届，我们将看我们将开始建立的简单示例数据库，并指出用来决定我们需要哪些表的一些方法。

#### 理解一些优秀的基本规则（Basic Rules of Thumb）

当一个数据库被设计，它经常会被规范化；也就是说，一堆规则被应用来确保数据被按照一定的方式打破。在十二章中，我们将以正式的方式观察数据库设计。但作为启动，我们需要的只是一些简单的基本规则。这些规则只是用来帮助你理解我们将用来在以后的章节探索SQL和PostgreSQL的初始数据库。我们强烈建议你不仅仅只阅读这些规则然后就匆忙地设计一个有20个表格的数据库。至少要读完第12章你才能按照你的意愿做设计。

注：如果你对标准化形式非常感兴趣，我们建议你阅读Joe Celko的《SQL for Smarties》。它有一些各种各样优秀的标准定义，以及很多E.F.Codd博士提出的关系模型的规则以及SQL使用的示例。

##### 规则一：将数据拆分成列

第一条规则是将每一块信息或者数据属性放入单独一列。这对于很多人来说都很自然，可以假设他们很自然地都是这么想的。在我们原始的电子表格中，我们已经很自然地将每个客户的信息拆分成不同的列，例如名字将和邮政编码区别开。

在电子表格中，这条规则只是让对数据的工作更简单；例如，按邮政编码排序等。然而在数据库中，必须将数据差分成不同的属性。

为什么这在数据库中这么重要？从实践的观点上看，你很难说清楚你需要地址列中的第29到第35字符之间的数据，因为它碰巧是邮政编码。很有可能有这种规则不适用的地方，因而你可能取到错误的数据块。另一个需要将数据正确拆分的原因是因为每一列数据必须有相同的类型，而不像电子表格，它对数据列的类型很宽容。

##### 规则二：有一个唯一标记来标识每行

你会记起在本章开始的时候关于怎么标记电子表格里头每行数据的问题，我们纠结于什么可能是唯一的。就像说到的，这是因为没有主键。一般而言，不需要一个单独的列是唯一的，也许是两个列的组合，或者甚至是三个列才能唯一标记一行。你可能发现需要超过三列才能唯一标记一行数据，这可能很少见，也许可能是一个错误。

无论如何，必须有一个绝对必然的方法，当我查看某个特定的列或者一组列的内容时，我知道它将与表中其他所有的行有所不同。如果你无法找到这么一个列，或者需要找到超过三个列的组合来唯一标记每行，你需要增加一个列来完成这种目的。在我们的客户表中，我们添加了一个叫做customer\_id的列来标记每行。

##### 规则三：移除重复信息

回到我们尝试存储订单信息到客户表的时候，由于重复组的问题，表格看上去凌乱不堪。我们必须为每个客户重复订单信息很多次。这意味着我们永远无法知道订单需要多少列。在数据库中，表的列数实际上在设计的时候就已经固定了。所以我们必须在我们存储数据前预先确定我们需要多少列，每个列的类型是什么以及每个列的名字。永远不要尝试在单独的一行中存储重复的数据组。

围绕这一约束的解决方法就是我们针对订单和客户数据的方法：拆分数据到不同的表中。然后在你需要同时从两个表取数据时你可以关联这些表到一起。在我们的示例中，我们使用customer\_id来关联这两个表格。更正式的，我们有了一个多对一的关系；也就是说，我们可以从一个单独的客户处获得多条订单。

##### 规则四：正确地命名

这可能是最难很好实现的规则。我们怎么叫一个表或者列？表和列应该有简短且有意义的名字。如果你无法确定怎么叫一个东西，这通常意味着你的表和列设计不是很恰当。

很多数据库设计者都有他们自己的个人优秀规则或者命名约定用以确保表和列的命名在数据库中的一致性。不要让一些表名用单数而另一些用复数。例如，不要给一个表命名为office而另一个命名为departments，用office和department就好了。如果你为一个表的标识列的名字指定的规则为表名加“\_id”，请遵守它。如果你使用缩写，就总是使用缩写。如果表中的一个列是另一个表的键值（外键，我盟将在第十二章讲解），尽量使用相同的名字。在一个复杂的数据库中，如果命名方式不一致，将非常令人讨厌，例如customer\_id、customer\_ident、cust\_id以及cust\_no。

完成这表面上简单的正确命名这个目标非常有挑战性，但获得的结果是维护起来相当简单。

#### 建立一个简单的数据库设计

我们可以通过实体关系图画出我们的数据库设计，或者模式。对于我们的两个表的数据库，这样的关系图应该像图2-14一样。

注：一个实体关系图是用一个图形方式表现我们数据的逻辑结构。它可以帮助我们形象化表现我们不同的数据实体怎么关联到另一个实体。

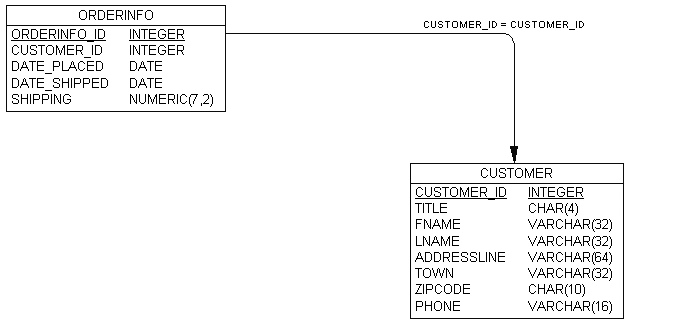


图2-14 单的实体关系图

这张图显示了我们的两张表，列和数据类型，以及每个列的大小，它还告诉我们customer\_id用来将两张表关联到一起。注意箭头从orderinfo表指向customer表，这意味着对于orderinfo的每一个条目，在customer表中最多只有一条对应的条目，但对于每个客户可能有多条订单。还要注意的是一些列有下划线，它指出这些列被确保为唯一的。这些列构成了这些表的主键。

你必须记清楚一对多关系是怎么关联的；如果在这里迷糊了，将带来一大堆问题。你还应该注意到我们很小心地命名用来关联两个表的列的名字均为customer\_id。这不是必要的。我们可以可以分别叫他们foo和bar如果我们愿意，但是就像前面小节所说的，命名一致性对于长期运行非常有帮助。

下一步是扩展我们非常简单的两表设计为稍微现实点的情况。我们将设计它为一个简单的订单管理数据库，叫做bpsimple。

#### 在两个表之上扩展

很明显，到现在为止我们拥有的信息缺乏每个订单的详细条目。你可能记得我们故意省略每条订单的实际条目，并承诺会回顾这个问题。现在是时候为每个订单增加实际条目了。

我们碰到的问题是我们预先无法知道每条订单将会有多少条条目。这和当初我们不知道一个客户将有多少条订单的问题一样。每条订单可能会有一条、两条、三条或者一百条条目。我们必须拆分客户的订单以及订单的内容信息。基本上，我们要做的就像图2-15中所展示的一样。

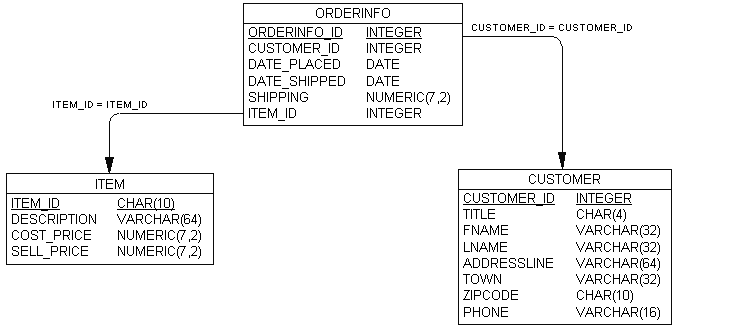


图2-15在客户和订单项目上的一个尝试

与customer和orderinfo一样，我们将这些信息拆分到两个表，然后将它们关联起来。然而，在这里出现了一个小问题。

如果你仔细考虑订单和订单项目的关系，你可能发现不仅仅orderinf表的每条记录关联到很多条目，而且在不同的客户订了相同条目的湿乎乎，每个条目可能出现在很多订单里头。

我们将在第十二章考虑这个问题，但是现在，你需要知道的就是我们有一个标准的解决方案用以解决这类难题。你可以在这两个表之间建立第三个表，用以实现多对多关系。这很容易实现但不容易讲明白，所以尽管先建立表orderline用来连接订单表和表中的项目，就像图2-16所示。

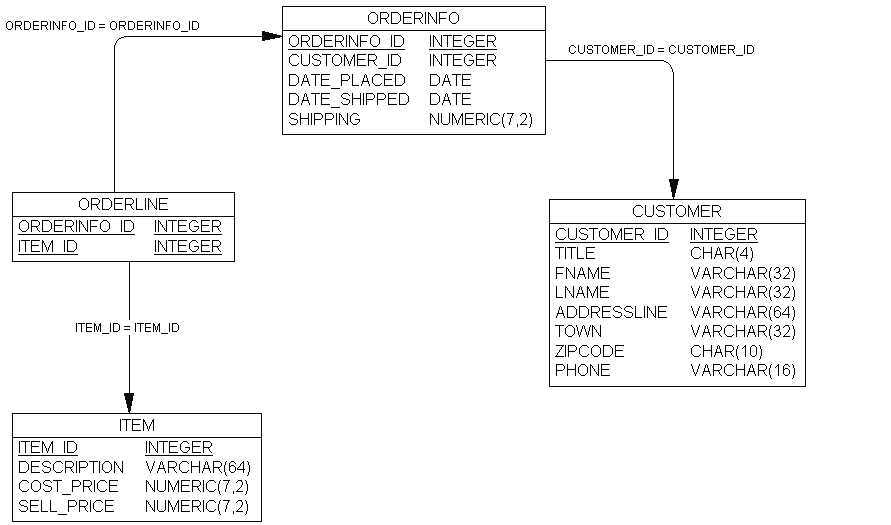


图2-16关联客户和订单

我们已经创建了一个每条记录都关联到一条订单记录的表。其每一行我们都可以通过orderinfo\_id列确定它的来源并且通过item\_id列确定它引用的条目。每个单独的项目都可以出现在多个订单行里头，且每个单独的订单都可以包含很多订单行。每条订单行只引用一个单独的条目，且每条订单行只会出现在一个订单中。

你还会发现我们不需要为标记每行而添加唯一的id列。这是因为组合的orderinfo\_id和item\_id总是唯一的。但是这里还有一个潜在的微妙问题。如果客户在一个订单里头订了一个条目两次将会发生什么？我们无法在orderline表里头输入另一行因为我们刚刚才说组合的orderinfo\_id和item\_id总是唯一的。我们是否需要添加另一个特殊的表来迎合包含不止一个相同项目的订单？幸运的是我们不需要那么做。有一个更简单的方法。我们只需要在orderline表里头添加数量列，这就令人满意了（查看下一节的图2-17）。

#### 完成初始化设计

我们在完成第一版的数据库主题结构设计完成前还需要存储两块数据。我们需要存储每个产品的条码，以及我们仓库中每个项目的存量。

很可能每项产品有不止一个条码，因为当制造商明显地改变产品外包装的时候，他们通常会修改条码。例如你可能看到打包的“赠送20%”（通常是指销售上的捆绑赠送包）。生产商通常会为这类推销包改变条码，但实际上产品没有改变。因此，我们可能有很多条码到一个项目的关系。我们增加了一个表存储条码，如图2-17所示。

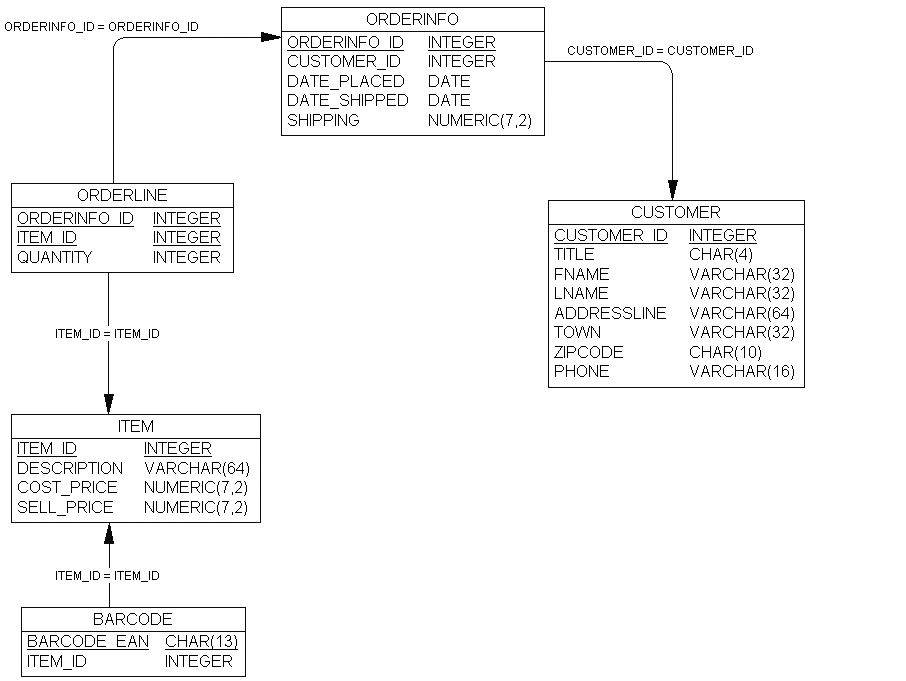


图2-17增加条码这个关系

注意从barcode表到item表的箭头的方向，因为可能每个条目有多个条码。同时还要注意barcode\_ean列是主键，因为每个条码都需要唯一的一行，并且每个独立的条目都可以有多个条码，但没有条码可以同时属于多个项目（EAN是欧洲的产品条码标准）。

我们最后需要添加的是我们仓库里头每项条目的存量。如果大多数条目是在仓库中且仓库信息都很基本，我们可以简单地在item表里存储库存量。但是，如果我们提供很多产品单只有少量库存，且我们需要存储很多关于仓库中项目的信息，这将行不通了。例如在仓库运营中，我们需要存储产地信息，批次号已经过期日期。如果我们的项目文件里头有500,000条记录，但仓库中只有前1,000条，这将非常浪费。这也有一个标准的解法，就是使用叫辅助表的表格。我们将通过这种方式存储仓储信息到我们的示例数据库中，如图2-18所示。

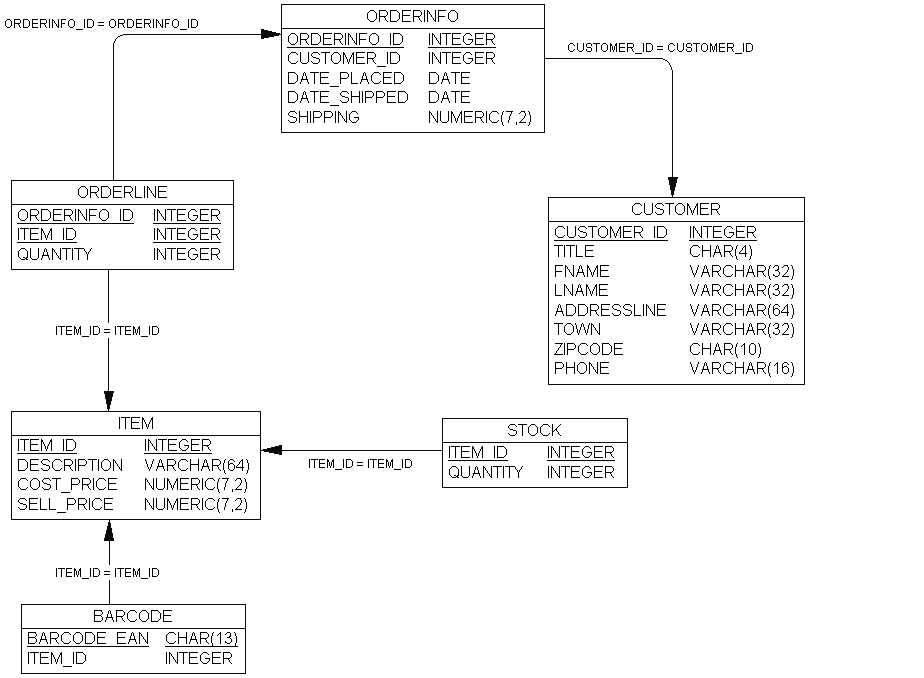


图2-19 bpsimple数据库的设计

我们建立了一个新表存储供给信息（本例中为库存），然后建立在仓库中的项目只需要的行，用来连接这些信息到主表中。注意stock表使用item\_id作为唯一键，它存储直接关联到项目的信息，使用item\_id关联到item表相关的行。箭头指向item表，因为它是主表，即使在本例中它不是一个多对一的关系。对于其他的表，下划线表明了主键（保证唯一的信息）。

按照现在的情况，我们的设计很明显过度复杂了，因为我们需要保存的扩展信息太少了。我们只是想演示模式设计的方法来看看它怎么做到我们的需求的，在本书后面的章节，我们将演示怎么访问像这里一样存在补充表格情况下的数据。对于那些喜欢偷偷往前翻的家伙，我们可以告诉你我们将使用的叫做外连接（outer join）。

注：在第八章，我们将看到我们怎么确保数据库中的表与表之间的这些关系被执行，在第十二章我们将再关注数据库设计的进一步细化。当我们到达第八章，我们将发现一些更高级的技术用于更好地管理数据库的一致性，而且我们将把这些增强设计加入到bpfinal模式中。

### 基本数据类型

在我们的示例数据库中，我们用到了一些基本的通用的数据类型，总结在表2-1中。当我们在下一章建表的时候这些将被转化成实际的PostgreSQL类型。

表2-1 示例数据库中的数据类型

| 数据类型 | 描述 |
| --- | --- |
| integer | 一个整数。 |
| serial | 一个整数，但是在新增行的时候会自动设置为一个唯一数字。这种类型我们通常会用在id列。本章的图表显示这种字段为整数是因为后台的数据库实际上是用的这种类型。 |
| char | 定长的字符数组，数组的长度为类型后面的括号中的数字。对于这种列类型，PostgreSQL总是存储规定长度的字符数。如果我们使用char(256)存储仅仅一个字符，数据库将存储256（至少，可能有其他辅助数据）个字节的数据，而且在检索数据的时候会返回这么多数据。 |
| varchar | 这也是一个字符数组，但就像它的名字暗示的一样，它是变长的。通常，在数据库中占用的空间和实际上需要存储的数据的长度一样。当你要求返回varchar类型的字段，它仅仅返回你存储长度的字符。能存储的最大长度由类型后面括号中的数字表示。 |
| date | 它允许你存储年月日信息。还有相关的类型允许我们像日期信息一样存储时间信息。我们将在后面点的第八章碰到它。 |
| numeric | 它允许你存储指定位数的数字（类型后面括号中的第一个数字）以及固定位数的小数（类型后面括号中的第二个数字）。在这里，number(7,2)将存储7位数字，其中小数点后有两位。 |

就像在本章早些时候标注的，由于添加一个特殊的唯一数据列在数据库中非常常见，所以大多数数据库都有一个内置的解决方案：一个叫做serial（序列号）类型数据类型。这种特殊的类型实际上是一个在有行插入表格时会自动增长的integer类型，它会在每行插入数据的时候赋予一个唯一的数字。当我们在有serial列的表里头加入一行数据时，我们不需要指定那列的值，而是让数据库自动赋予下一个值给它。大多数数据库在他们分配序列值的时候，从不考虑将要删除的行。分配的新值只会随着新行增加。我们将在第六章看到怎么处理serial类型数据失序的问题。

在第八章，我们将碰到PostgreSQL的其他数据类型，这些将让我们有一个机会重新审视现这些数据类型的选择。附录B提供PostgreSQL数据类型的概要说明。

### 处理未知的值：空值（NULL）

在我们示例数据库设计中的orderinfo表中，我们有一个下订单日期和一个发货日期，都是日期类型。在我们已经收到订单但还未发货的时候，我们该怎么办？我们该怎么存储发货日期？我们可以存储一个特殊的日期，一个守护值（sentinel value），用来告诉我们我们还未针对订单发货。在UNIX类型的系统，我们可以使用1970年1月1日，这是一个传统的UNIX系统开始计数的日期。那个日期绝对比任何我们想要存储到数据库的订单日期都早，所以我们总是可以知道那个日期意味着还未发货。

但是，在表中稀稀拉拉地使用特殊值值显示出糟糕的设计，有时候甚至是错误的。例如，一个新手参与这个项目并且不知道有一些特殊日期，他可能尝试计算下订单和发货间隔的平均耗时，如果这里有一些发货日期比下订单的日期更早，他可能得到一些非常奇怪的值。

幸运的是，所有的关系数据库都提供一个叫空值（NULL）的特殊值，用来表示当前是未知的。注意它并不表示零，或者空字符串，或者任何可以由字段的数据类型表现的东西。未知的值和零或者空字符串完全不同。实际上，空值甚至根本不是一个值。

空值的概念通常困扰很多数据库新用户（罗马人也对不存在的东西很困恼，所以罗马数字里头没有零）。在数据库术语中，空值通常指值还不知道，但它也有一些在这个意思之上的其他的甚至有一些微妙变化的意思。

要非常小心对待空值，因为它们会偶尔会出现在你面前并令你奇怪，通常是令人讨厌的。所以，在我们的orderinfo表中，我们可以在未发货前设置送货日期为空值，这意味着“现在还未知道”，这正是我们所需要的。有另一个关于NULL微妙的用法（不是很常用），它意味着“跟这行无关的”。假设你对别人做一个调查，问题之一是关于眼镜的颜色。对于不戴眼镜的人，这明显是一个无意义的问题。这就是空值会被用在记录这个列对于这一行无关的情况。

关于空值的一个特征是如果你比较两个空值，结果通常是未知的。这有时候会把人弄糊涂，但是如果你想清楚空值的意思就是不知道，那么对于测试两个未知的东西是否相等的结果也是未知就非常符合逻辑了。SQL有特殊的方法，通过IS NULL检查是否空值。这将允许你在你想要的时候查找以及测试空值。IS NULL将在第四章讨论。

空值的表现和常规的值的有些细微的不同。所以，有必要在你设计表的时候指出一些列不能存储空值。通常在你确保一个列永远不能接受空值的时候指定它为NOT NULL是个好主意，例如主键列。一些数据库设计者甚至提倡基本上完全禁止空值，但空值确实是有用的，所以我们通常提倡在一些特意选择的列中允许空值，这些列确实有一些真正的可能性需要未知的值。NOT NULL将在第八章详细讨论。

### 回顾示例数据库

在本章，我们已经完成设计一个可能可以用于小商店的叫做bpsimple的专用的简单数据库，用来管理客户、订单以及订单项（见本章稍早点的图2-18）。随着本书的前进，我们将使用这个数据库演示SQL和其他的PostgreSQL的功能。我们还将发觉我们现有设计的不足，并判断它在哪些方面可以得到改进。

这个我们在使用的简化的数据库拥有很多现实发布的数据库看上去拥有的很多元素；但是它却是在很多方面被简化了。例如，一个条目应该有一个完整的描述用于仓库存档，一个简单的描述用于销售时显示，以及另一个描述显示在货架边缘的标签上。我们存储的客户地址信息也非常简单。我们无法处理过长的包含省市甚至街道的地址。我们也无法处理海外订单。

通常在初始设计中做到合理可靠的基础并扩展比尝试迎合所有可能的需求更可行。这个数据库足够符合我们的初始需求。

下一章，我们将研究安装PostgreSQL，建立我们示例中的表并填充一些示例数据。

## 摘要

在本章中，我们仔细分析了一个简单的数据库表与单个的电子表格的相似之处，以及四个重要的区别：

* 一个列中的所有项目的类型都必须相同。
* 表中所有行的列数都必须相同。
* 必须想办法唯一标记每一行。
* 在数据库表不包含行的顺序，但电子表格中包含。

我们看到我们怎么扩展我们的数据库到多个表，这让我们能简单管理我们多对一的关系。我们给出一些非正式的rules of thumb来帮助你理解怎样将数据库设计构建起来。我们将在后面的章节以更严谨的方式回到数据库设计的主题。

我们还看到了怎样通过添加一个额外的表，将它拆分成两个一对多关系，实现现实世界中出现的多对多关系。

最后，我们扩展了我们的初始数据库设计，因此我们有了一个我们可以用于伴随本书向前的示例数据库，或者叫做模式。

下一章，我们将了解怎样使PostgreSQL在不同的平台中运行起来。

# 初步使用PostgreSQL

本章将讨论在各种操作系统中安装和配置PostgreSQL。如果你需要在一个Linux系统中安装它，预编译的二进制包比较容易安装。如果你是在运行一个UNIX系统或类UNIX系统，例如Linux，FreeBSD，AIX，Solaris，HP-UX或者Mac OS X，从源码编译PostgreSQL也不难。

我们也将讲到怎样在Windows平台上使用PostgreSQL 8.0里头介绍的Windows安装程序安装PostgreSQL。早期的版本可以安装在Windows中，但是这需要其他一些软件来建立类UNIX环境。我们强烈建议在Windows系统中使用8.0版本或更新的版本。

最后，我们将在以下的章节准备第2章讨论的示例。

本章将特别讨论以下主题：

* 在Linux上通过二进制文件安装PostgreSQL
* 在Linux上通过源码安装PostgreSQL
* 在Linux和UNIX系统上配置PostgreSQL
* 在Windows上安装和配置PostgreSQL
* 建立数据库，建立表，增加数据

## 在Linux和Unix系统中安装PostgreSQL

如果你安装较新发行版的Linux系统，你可能已经安装PostgreSQL或者在系统安装盘会带有它的安装包。如果没有，你可以在很多Linux发行版中通过RPM包安装PostgreSQL。另外，你可以在类UNIX系统中通过源码安装PostgreSQL。

### 在Linux中使用二进制文件安装PostgreSQL

可能在Linux中安装PostgreSQL最容易的方法就是通过二进制包安装。PostgreSQL的二进制文件对于很多Linux发行版来说都可以安装RPM （RPM包管理器，以前的Red Hat Package Manager）的安装包。在写本书的时候，在<http://www.postgresql.org/>有给以下平台使用的RPM安装包：

* Red Hat 9
* Red Hat Advanced Server 2.1
* Red Hat Enterprise Linux 3.0
* Fedora Core 1, 2（包括 64位平台), 3

你可以通过<http://www.rpmfind.net寻找其他Linux>发行版的安装包，包括以下发行版：

* SuSE Linux 8.2 and 9.x
* Conectiva Linux
* Mandrake
* Yellow Dog PPC

注：Debian Linux 用户可以通过apt-get安装PostgreSQL。

表3-1列出了PostgreSQL二进制安装包。为了安装数据库和客户端，你需要下载和安装至少base，libs和server包。

表3-1 PostgreSQL二进制包

|  |  |
| --- | --- |
| 包 | 描述 |
| postgresql | 包含客户端和工具的基础包 |
| postgresql-libs | 客户端需要的共享库 |
| postgresql-server | 建立和运行服务端的程序 |
| postgresql-contrib | 贡献的扩展程序 |
| postgresql-devel | 开发用的头文件和库 |
| postgresql-docs | 文档 |
| postgresql-jdbc | PostgreSQL的数据库连接库 |
| postgresql-odbc | PostgreSQL的ODBC接口库 |
| postgresql-pl | Perl的PostgreSQL服务器支持 |
| postgresql-python | Pyton的PostgreSQL服务器支持 |
| postgresql-tcl | Tcl的PostgreSQL服务器支持 |
| postgresql-test | PostgreSQL测试套件 |

实际的文件名会包含版本号。需要安装相同的版本的库，甚至是版本修订号。在版本号8.x.y的包中，x.y就是修订号。

#### 安装RPM包

安装RPM包，你可以使用以下方法：

* 使用RPM包管理程序。确保你以管理员（root）登录来执行安装。
* 使用你选择的图形界面包管理器，例如KPackage，来安装RPM包。
* 将所有的RPM文件放在同一个目录，并以管理员（root）身份执行以下命令来解压并安装所有的文件到恰当的位置：

$ rpm -U \*.rpm

你也可以安装和你的发行版绑定在一起的PostgreSQL包，例如Red Hat或SuSE Linux。例如，在Suse Linux 9.x中，你可以通过Yast2安装工具选择表3-1中的某个版本的PostgreSQL的安装包，如下图所示：

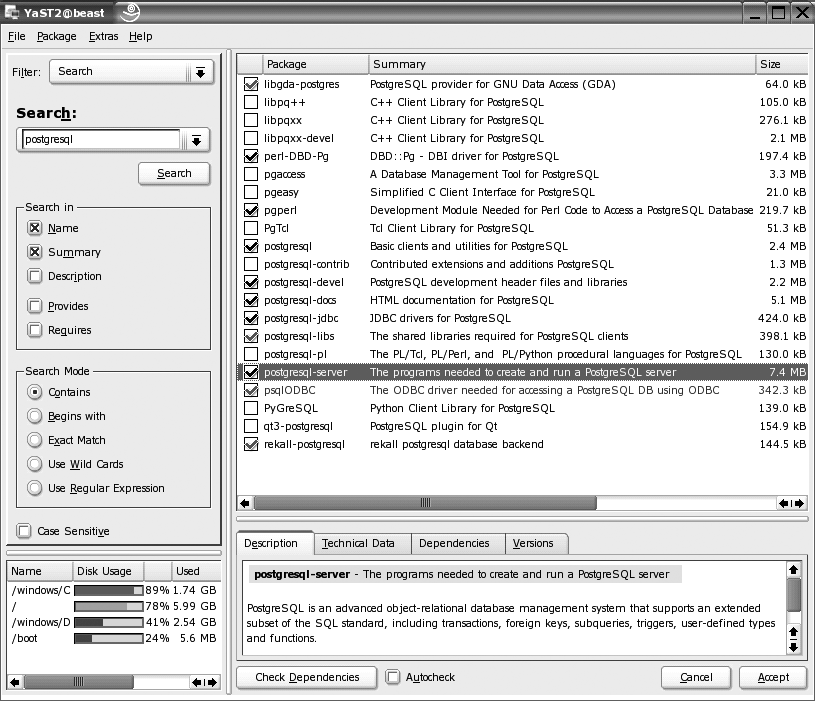


图3-1，通过Yast2从SuSE的安装包安装PostgreSQL

#### 升级到新版的PostgreSQL

PostgreSQL在不停的开发中，所以不停有新版产生。通过RPM包安装的很容易升级到最新的版本。完成省，仅仅需要使用-U参数而不是-i参数告诉rpm你在执行一次升级而不是首次安装：

$ rpm -U \*.rpm

但是，在执行升级前，你需要备份数据库中的现有数据。PostgreSQL官方网站的发行注记中会有一些在升级到最新版本前必须做的预防措施需要完成。备份现有数据库的细节将在本书11章讨论。

注：如果你是在安装一个新版的PostgreSQL到你现存的安装上，确保在执行升级前已经阅读新版的发行注记。在某些情况下，在升级后，必须备份和欢迎你的PostgreSQL数据库，例如新版本引用了新的数据存储方式。

#### PostgreSQL的安装解析

PostgreSQL的安装包含大量的应用程序、工具和数据目录。PostgreSQL的主应用程序（postmaster）包含为客户端访问数据提供服务的服务代码。工具例如pg\_ctl用来控制服务器激活后就必须一致运行的主服务进程。

PostgreSQL使用一个目录存放数据库所有的文件。这个目录不仅存放表和记录，还存放系统参数。一个典型的安装将包含表3-2中列出的PostgreSQL安装的所有组件，存放在PostgreSQL目录下的子目录中。通常情况下在/usr/local/pgsql（也是使用源码安装的默认位置，下节将讨论）。

表3-2 PostgreSQL的安装解析

|  |  |
| --- | --- |
| 目录 | 描述 |
| bin | 应用程序和工具，例如pg\_ctl和postmaster |
| data | 数据库本书，通过initdb初始化 |
| doc | HTML格式的文档 |
| include | 用于开发PostgreSQL应用程序的头文件 |
| lib | 用于开发PostgreSQL应用程序的库文件 |
| man | PostgreSQL的手册 |
| share | 配置文件示例 |

单目录的方式有个缺点：固定的程序和经常变动的数据存储在同一个地方，通常不理想。

PostgreSQL使用的文件分为两大类：

* 在数据库服务器运行的时候需要写入的文件，包括数据文件和日志。数据文件时系统的心脏，存储你的数据库的所有的信息。数据库服务器产生的日志文件包含关于数据库访问的信息，可以在解决故障的时候有很大的作用。在日志记录大开后，它将一直增长。
* 在数据库服务器运行的时候不需要写入的文件，实际上是只读文件。这些文件包含PostgreSQL的应用程序就像postmaster和pg\_ctl，这些东西一旦安装完成就不再改变（升级除外）。

为了更有效和简单的管理安装，你也许希望分开不同类别的文件。PostgreSQL提供灵活的方法分开存储应用程序、日志和数据，一些Linux发行版已经使这种灵活性生效了。例如，在SuSE Linux 9.x中，PostgreSQL应用程序和其他程序一样存储在/usr/bin，日志文件存放在/var/log/postgresql，数据文件存放在/var/lib/pgsql/data。这意味着非常容易安排备份从非关键文件区别出来的关键数据，例如日志文件。

其他发行版都有自己的文件分布规划。你可以使用rpm命令来列出某个包安装的文件。为了做到这个，使用以下的查询选项：

$ rpm -q -l postgresql-libs

/usr/lib/libecpg.so.5

/usr/lib/libecpg.so.5.1

...

/usr/share/locale/zh\_TW/LC\_MESSAGES/libpq.mo

为了查看所有安装的文件，你需要针对所有的用以安装完整的PostgreSQL的包运行rpm命令。不同的发行版的包名字可能有细微区别。例如，SuSE Linux使用名为pg\_serv作为服务器程序的包名称，所以查询选项应该像这样：

$ rpm -q -l pg\_serv

/etc/init.d/postgresql

/etc/logrotate.d/postgresql

...

/var/lib/pgsql/data/pg\_options

当然，你也可以选择一个图形界面的包管理工具，例如KDE桌面环境提供的KPackage。图3-2展示通过KPackage查看包内容的一个例子。

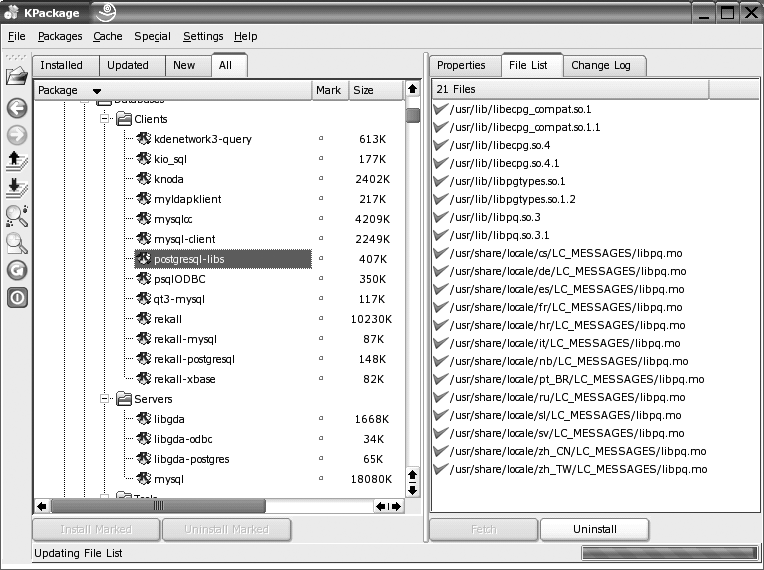


图3-2 使用KPackage查看包内容的示例

使用Linxu发行版安装的缺点是总是不是很清楚安装的东西在哪里。所以，如果你想升级到最新的版本，很难处理确保你已经清理掉了你原来执行的安装。一个解决方案是从源码安装PostgreSQL，就像下一节讨论的一样。加入你没有从源码安装的打算，你可以跳过下一节，直接跳到讨论PostgreSQL设置的“在Linux和Unix系统中设置PostgreSQL”小节。

### 通过源码安装PostgreSQL

就像前一章讲述的，在很多Linux发行版中你可以使用RPM包安装PostgreSQL。另外，你可以在任何UNIX兼容的系统中从源码创建和安装PostgreSQL，包括Mac OS X。

PostgreSQL的源码可以在<http://www.postgresql.org>找到。在这里，你将找到最新的源码以及下一个版本的beta测试版的源码。除非你很喜欢尝鲜，否则建议你使用最新的稳定版本。

你可以找到整个打包了的PostgreSQL的源码，可能是gzip压缩的tar打包文件，就像postgresql-9.0.0.tar.gz或者bzip2压缩的tar打包文件，就像postgresql-9.0.0.tar.bz2。在写本文的时候，PostgreSQL打包文件已经超过13MB了。

实际的文件名依赖于当前版本的修订号。

编译PostgreSQL相当简单。如果你熟悉编译开源产品，对你来说这里不会有任何意外。即使你是第一次编译和安装开源产品，你也不会觉得困难。

为了执行源码编译，你需要一个带有完整开发环境的Linux或UNIX系统。这包括C编译器和GNU的make工具（对于编译本数据库系统是必须的）。Linux发行版通常包含恰当的自由软件基金提供的GNU工具。这包含优秀的GNU C编译器（gcc），这也是Linux的标准编译器。绝大部分UNIX平台也都有GNU工具，我们推荐使用它们来编译PostgreSQL。你可以从<http://www.gnu.org>下载最新的工具。一旦你完成开发工具的安装，编译PostgreSQL就轻而易举了。

#### 解压源码

使用普通用户开始安装。拷贝打包的源码到一个合适的目录用于编译。这个目录不需要（实际上，不应该）在最终PostgreSQL安装的位置。一个可能的位置是你home目录中的义工字幕了。因为你不需要超级用户权限用来编译PostgreSQL；超级用户权限仅仅在建立完成后安装的时候需要用一下。我们通常解压源码到专门用来管理产品源码的目录/usr/src，但你可以解压源码到任何有足够空间用于编译的地方。大概需要90M多一点空间存放解压后的源码。

解压打包的源码的命令如下：

$ tar zxf postgresql-9.0.1.tar.gz

解压的过程将建立新的目录，目录名依赖于你编译的PostgreSQL版本。进入目录：

$ cd postgresql-8.0.0

提示：在这个目录中你可以找到一个叫INSTALL的文件，这个文件包含了编译相关指令的详细手册，有些少见的情况自动化的过程会失败。

#### 配置编译

建立过程使用一个叫configure的配置脚本针对你的特别平台来裁剪的建立参数。如果想接受所有默认参数，你可以简单的不带任何参数运行configure。以下为在一个Linux系统新运行configure的示例：

$ ./configure

checking build system type... i686-pc-linux-gnu

checking host system type... i686-pc-linux-gnu

checking which template to use... linux

checking whether to build with 64-bit integer date/time support... yes

checking whether NLS is wanted... no

checking for default port number... 5432

checking for block size... 8kB

checking for segment size... 1GB

checking for WAL block size... 8kB

checking for WAL segment size... 16MB

checking for gcc... gcc

...

configure脚本设置控制PostgreSQL生成方法的变量，统计编译的平台，C编译器提供的功能等。configure脚本将自动设置安装的位置。默认的PostgreSQL安装位置位于/usr/local/pgsql，包含应用程序和数据的子目录。

你可以使用configure的参数来改变默认位置，设置数据库服务器使用的网络端口以及附加的存储过程使用的服务端程序语言。这些选项在表3-3中列出。

表3-3 PostgreSQL配置脚本选项

|  |  |
| --- | --- |
| 选项 | 描述 |
| --prefix=*prefix* | 安装到prefix指向的目录；默认为/usr/local/pgsql |
| --bindir=*dir* | 安装应用程序到dir；默认为prefix/bin |
| --with-docdir=*dir* | 安装文档到dir；默认为prefix/doc |
| --with-pgport=*port* | 设置默认的服务器端网络连接服务TCP端口号 |
| --with-tcl | 为服务端提供Tcl存储过程支持 |
| --with-perl | 为服务端提供Perl存储过程支持 |
| --with-python | 为服务端提供Python存储过程支持 |

你可以通过—help参数查看configure所有的选项。

$ ./configure --help

`configure' configures PostgreSQL 9.0.1 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as

VAR=VALUE. See below for descriptions of some of the useful variables.

...

$

你在这一步可以不设置数据库和日志文件的位置。你通常可以在安装完成后启动服务进程的时候指定这些位置。

#### 构建软件

一旦编译完成，你可以通过make命令构建软件。PostgreSQL构建过程使用一套精密的makefile来控制编译过程。为了完成这个过程，我们建议你使用GNU版本的make工具。这是Linux默认的make工具。在其他的UNIX平台，你可能需要另外安装GNU make。通常这会用gmake这个名字，来区别于系统自带的make。在这里，我们说的make就是指GNU make。

$ make

...

All of PostgreSQL successfully made. Ready to install.

如果一切正常，我们将看到大量的编译过程。我们将最终得到一切都成功完成的好消息。

当make完成后，你需要将程序复制到目标位置。你可以使用make完成，但你需要先切换到超级用户：

$ su

# make install

...

PostgreSQL installation complete.

# exit

$

一旦软件构建完成并安装好，你可以通过pg\_config命令获得一份PostgreSQL系统的配置：

pg\_config --bindir | --includedir | --libdir | --configure | --version

pg\_config命令将回报PostgreSQL程序的安装目录（--bindir），C头文件的位置（--includedir）和库文件的目录（--libdir）以及PostgreSQL的版本（--version）：

$ pg\_config --version

PostgreSQL 9.0.1

$

编译时的配置参数可以通过pg\_config –configure查看。这将提供PostgreSQL服务程序配置用于编译的configure脚本的命令行参数。

以上就是安装PostgreSQL的内容。你现在已经有一套用于PostgreSQL数据库服务器的程序在你系统的适当位置了。

在这一点上，你和用安装包安装一样都处在相同的情况了。现在，是时候配置你安装的PostgreSQL了。

### 在Linux和Unix上配置PostgreSQL

在你安装好PostgreSQL后，无论是通过RPM包或者通过源码编译，你都需要执行一些步骤让它运行起来。第一步，你需要建立一个叫postgres的用户，然后你需要为数据库建立一个目录并初始化数据库结构。然后，你可以通过启动postmaster进程启动PostgreSQL了。

#### 建立postgres用户

PostgreSQL的数据库主进程postmaster是一个特别的程序。它负责处理所有的用户对所有数据库的访问。它必须允许用户访问自己的数据但在没授权的情况下不允许访问其他用户的数据。为了实现这个功能，它需要能够控制所有的数据文件，因而普通用户不允许直接访问这些文件。postmaster进程将通过检查访问数据的用户的赋权情况控制对数据文件的访问。

一般来说，PostgreSQL需要用一个非管理员用户运行，也就是说可以是任何普通用户；如果你在你的home目录里头安装了数据库，这个用户可以是你自己的用户。但是，PostgreSQL通常使用一个概念上的虚拟用户来完成数据访问。通常，这个一个叫做postgres的用户被建立用来管理这些数据文件，它不需要其他的访问权限。另外，postgres虚拟用户可以提供一些其他的安全措施，例如这个用户无法登录，所以别人无法非法访问这些数据。postmaster程序代表其他用户用这个用户去访问数据库文件。

因此，建立一个可运行的PostgreSQL系统的第一步就是建立postgres用户。每个系统建立用户的方法都不同。Linux可以通过root用户使用useradd命令添加：

# useradd postgres

其他的UNIX系统中尼可能需要建立home目录，修改配置文件或者运行相关的工具。请参考你的系统的文档来获得相关管理工具的细节。

#### 建立数据库目录

下一步，你必须通过root用户建立一个目录给PostgreSQL用来存放数据库，并将目录的所有者设置为postgres：

# mkdir /usr/local/pgsql/data

# chown postgres /usr/local/pgsql/data

在这里我们使用默认的位置给数据库。你可以选择在其他地方存储数据，就像我们前面在“PostgreSQL的安装解析”小节里讨论的一样。

#### 初始化数据库

通过initdb工具初始化PostgreSQL数据库，初始化时需要制定你文件系统中想要存储数据库的位置。这将做很多事，包括建立PostgreSQL需要运行的数据结构以及初始化一个可工作的数据库：template1。

你需要使用postgres用户来运行initdb工具。为了做到这点，最可靠的方法是完成两步，第一步是通过su命令切换到root用户，然后切换成postgres用户，就像以下所示（作为一个普通用户，你可能无法用其他用户身份运行程序，所以你必须先变成超级用户）：

$ su

# su - postgres

pg$

现在你运行的程序是以postgres用户运行的，并且你可以访问PostgreSQL的数据文件了。很明显，我们现实了postgres用户的shell的命令提示符pg$。

警告：不要为了图方便直接用root而不是postgres用户完成以上过程。由于安全原因，用root身份运行服务进程可能非常危险。如果这个进程有问题，可能导致外部通过网络非法访问你的系统。由于这个原因，postmaster将拒绝通过root运行。

通过initdb命令初始化数据库：

pg$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

The files belonging to this database system will be owned by user "postgres".

This user must also own the server process.

The database cluster will be initialized with locale en\_US.UTF-8.

The default database encoding has accordingly been set to UTF8.

The default text search configuration will be set to "english".

...

WARNING: enabling "trust" authentication for local connections

You can change this by editing pg\_hba.conf or using the -A option the

next time you run initdb.

Success. You can now start the database server using:

/usr/local/pgsql9/bin/postgres -D pgdata

or

/usr/local/pgsql9/bin/pg\_ctl -D pgdata -l logfile start

pg$

如果一切正常，你将在initdb命令的-D参数指向的位置拥有一个全新的空白数据库。

#### 配置连接权限

默认情况下，PostgreSQL不允许全面的远程访问。为了赋权给远程连接，你必须编辑配置文件pg\_hba.conf。这个文件存在于数据库文件的区域（在本例中，位于/usr/local/pgsql/），它包含允许或者拒绝特定用户连接到数据库的权限的配置记录。默认情况下，本地用户可以连接但远程用户不允许。文件格式非常简单，PostgreSQL自带的默认文件包含大量的有用的注释用于协助添加记录。你可以根据需要给单个的用户、主机、计算机组或者单独的数据库赋权。

例如，希望允许用户neil通过IP地址为192.168.0.3的主机连接到bpsimple数据库，添加以下行到pg\_hba.conf文件：

host bpsimple neil 192.168.0.3/32 md5

注意，在早于8.0版本的PostgreSQL中，pg\_hba.conf通过IP地址和子网掩码说明一个主机地址，所以之前的例子应该写成这样：

host bpsimple neil 192.168.0.3 255.255.255.255 md5

本例中，我们将添加一条记录来运行局域网中（本例中，子网为192.168.x.x）的任何计算机通过密码认证访问数据库。（如果你需要不同的访问策略，参考配置文件里头的注释）我们添加一行到pg\_hba.conf的末尾，就像这样：

host all all 192.168.0.0/16 md5

这意味着IP地址由192.168开头的计算机可以访问所有的数据库。此外，加入我们信任网络中的所有用户，我们可以通过指定使用trust标记不受限的访问方法作为访问策略，就像这样：

host all all 192.168.0.0/16 trust

PostgreSQL的postmaster服务进程读取配置文件postgresql.conf（也存在于数据目录中）来设置一系列的运行选项，包括（如果没有另外指定-D选项或者配置PGDATA环境变量）数据库数据文件的位置。这个配置文件被很好的注释了，如果你想修改任何设置，它都提供了向导。PostgreSQL的文档有一章讲述了运行配置。

例如，我们可以设置postgresql.conf文件中的listen\_addresses参数允许服务器监听网络连接，而不是通过-i选项：

listen\_addresses='\*'

实际上，我们推荐通过postgresql.conf设置配置参数来控制postmaster进程的行为。

#### 启动postmaster进程

现在，你可以启动服务进程了。再次提醒，你可以使用-D选项告诉postmaster数据库文件所在位置。如果你想允许网络上的用户访问你的数据，你可以使用-i选项启用远程访问（如果你没在postgresql.conf文件中启用listen\_addresses选项，就像前面所说的）：

pg$ /usr/local/pgsql/bin/postmaster -i -D /usr/local/pgsql/data >logfile 2>&1 &

这条命令启动postmaster，重定向进程输出到一个文件（名叫logfile，存放在postgres用户的home目录中），并且通过shell的2>&1合并标准输出和标准错误输出。你可以通过重定向输出到其他文件来选择不同的日志位置。

PostgreSQL提供的pg\_ctl工具提供了一种简单的方法启动、停止和重启（就是停止和启动）postmaster进程。如果PostgreSQL就像之前所属的那样完全由postgresql.conf文件配置，可以简单的使用以下命令启动、停止和重启：

pg\_ctl start

pg\_ctl stop

pg\_ctl restart

#### 连接到数据库

现在你可以通过尝试连接到数据库测试它是否正常工作了。psql工具是用来与数据库进行交互和进行简单的管理工作例如建立用户，建立数据库以及建表。在本章后面我们将用它来建立和填充数据库，在第5章将详细讲解它的功能。现在，你可以简单地尝试连接到一个数据库。以下的反馈显示你已经运行了postmaster：

pg$ /usr/local/pgsql/bin/psql

psql: FATAL 1: Database "postgres" does not exist in the system catalog.

不要被上面显示的致命错误吓着。默认情况下，psql连接到本机的数据库并尝试用启动这个程序的用户的名称打开数据库。因为我们在这里没有建立叫postgres的数据库，所以连接失败。这象征着，postmaster进程运行了并且能够正常响应失败的细节。

为了指定连接的数据库，可以传递-d参数给psql。全新的PostgreSQL系统包含一些系统使用的数据库作为你需要新建的数据库的模板。其中有一个叫做template1。如果你需要，你可以连接到数据库这个数据库用来完成管理功能。

为了检查网络连接，你可以使用网络上其他机器安装的psql作为客户端，或者其他的PostgreSQL兼容的程序。在psql中，你可以使用-h选项指定主机（无论是名称还是IP地址），并指定一个系统数据库（如果你还没建立一个真正的数据库）。

remote$ psql -h 192.168.0.111 -d template1

Welcome to psql 8.0.0, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms

\h for help with SQL commands

\? for help with psql commands

\g or terminate with semicolon to execute query

\q to quit

template1=# \q

remote$

#### 配置自动启动

最后一步需要做的是安排postmaster服务进程在机器重启的时候自动启动。实际上你所有要做的就是确保postmaster在启动后被运行了。再次，对于Linux和UNIX变种系统，这里都有一些小标准可以遵从。请参考你的系统文档的细节。

如果你是从Linux发行版安装的PostgreSQL，启动脚本应该已经通过RPM包安装的时候完成配置了。在SuSE Linux中，PostgreSQL在系统进入多用户模式的时候通过/etc/rc.d/init.d下面的一个叫做postgresql的脚本启动。

如果你想自己写启动脚本，最简单的办法是建立一个简单的脚步使用你需要的参数启动postmaster，并且在一个自动启动的脚本里头添加一个到你的脚本的调用就OK，例如在/etc/rc.d里面的脚本。需要确保postmaster是由postgres用户启动的。以下是从源码安装的一个默认安装的PostgreSQL的可以完成以上工作的示例脚本：

#!/bin/sh

# Script to start and stop PostgreSQL

SERVER=/usr/local/pgsql/bin/postmaster

PGCTL=/usr/local/pgsql/bin/pg\_ctl

PGDATA=/usr/local/pgsql/data

OPTIONS=-i

LOGFILE=/usr/local/pgsql/data/postmaster.log

case "$1" in

start)

echo -n "Starting PostgreSQL..."

su -l postgres -c "nohup $SERVER $OPTIONS -D $PGDATA >$LOGFILE 2>&1 &"

;;

stop)

echo -n "Stopping PostgreSQL..."

su -l postgres -c "$PGCTL -D $PGDATA stop"

;;

\*)

echo "Usage: $0 {start|stop}"

exit 1

;;

esac

exit 0

注：在Debian Linux中，在su -l的地方，你需要使用su -。

建立一个包含以上脚本的可执行脚本。给它命名为MyPostgreSQL。使用chmod命令给它赋予执行权限，就像以下的情况：

# chmod a+rx MyPostgreSQL

然后，你需要安排脚本在服务器启动和关机的时候启动和停止PostgreSQL：

MyPostgreSQL start

MyPostgreSQL stop

对于使用System V类型的init脚本的系统（例如很多Linux发行版），你可以把脚本放在适当的位置。例如在SuSE Linux中，你应该把脚本放在/etc/rc.d/init.d/MyPostgreSQL，并且建立以下位置的软连接到这个脚本来实现在服务器进入和离开多用户模式的时候启动和停止PostgreSQL：

/etc/rc.d/rc2.d/S25MyPostgreSQL

/etc/rc.d/rc2.d/K25MyPostgreSQL

/etc/rc.d/rc3.d/S25MyPostgreSQL

/etc/rc.d/rc3.d/K25MyPostgreSQL

请参考你的系统文档关于启动脚本的部分的详细信息。

#### 停止PostgreSQL

PostgreSQL服务进程有序关闭非常重要，这将允许它将任何未写入数据库的数据写入数据库并释放它使用的共享内存资源。

为了安全地关闭数据库，可以通过postgres用户或者root用户使用pg\_ctrl工具这样做：

# /usr/local/pgsql/bin/pg\_ctl -D /usr/local/pgsql/data stop

如果有启动脚本，你可以像下面这样使用它们。

# /etc/rc.d/init.d/MyPostgreSQL stop

这些脚本能保证数据库在机器关机或者重启的时候能正常关闭。

相关资源

为了在使用PostgreSQL时方便一点，最好添加PostgreSQL应用程序路径到执行程序的搜索路径中，手册文件也需要这么做。在标准UNIX shell中，添加以下代码到你的启动脚本中（.profile or .bashrc）：

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

MANPATH=$MANPATH:/usr/local/pgsql/man

export PATH MANPATH

当前和最新测试版本的PostgreSQL的源码可以在<http://www.postgresql.org>找到。更多的PostgreSQL资源信息在本书附录G中列出来了。

## 在Windows中安装PostgreSQL

在本小节开始前，先给Windows用户一些好消息。虽然PostgreSQL是为类UNIX平台开发的，它却是可移植的。

现在已经可能在Windows上写PostgreSQL客户端程序，而且从7.1版开始，PostgreSQL可以编译安装和作为一个PostgreSQL服务器运行在Windows NT 4，2000，XP和Server 2003中。

从PostgreSQL 8.0开始，已经有了Windows本地版本了，为服务端和客户端程序提供了Windows的安装程序，这让在Windows下的安装非常轻松。在8.0版之前，Windows用户需要安装一些软件来在Windows上提供UNIX功能。

注：PostgreSQL 8.0支持Windows 2000，Windows XP和Windows Server 2003，由于Windows 95，98和Me不提供一些它需要的功能所以它无法在这些平台运行。它可以在Windows NT上跑起来，但必须手工安装，因为PostgreSQL的安装程序无法在Windows NT上正常运行。

看起来像Linux一样的开源的操作系统平台是PostgreSQL一样的开源数据库的自然归宿。实际上，我们不推荐在桌面版的Windows上运行生产用的数据库，但在Windows上安装有它的优点。例如在客户程序所在的机器上拥有PostgreSQL的工具套件例如psql在测试新装数据库和查找连接故障的时候非常有用。虽然你不需要在Windows上运行服务程序。在开发机上运行数据库服务器可以避免开发人员需要共享服务器实例而带来的问题。

### 使用Windows安装程序

EnterpriseDB公司（公司网站：http://www.enterprisedb.com）提供了Windows版的PostgreSQL安装程序。安装程序的名字类似于postgresql-9.0.1-1-windows.exe。安装程序需要系统安装了2.0或更高版本的Windows安装器。Windows XP以及以后的版本中已经包含了恰当的版本。如果有必要，Windows安装期可以从微软的官方网站http://www.microsoft.com下载（搜索“Windows Installer redistributable”）。

双击这个文件就可以启动安装程序，进入安装向导。在选择安装语言和阅读安装注记后，你将看到选择安装目录对话框，如图3-3所示。

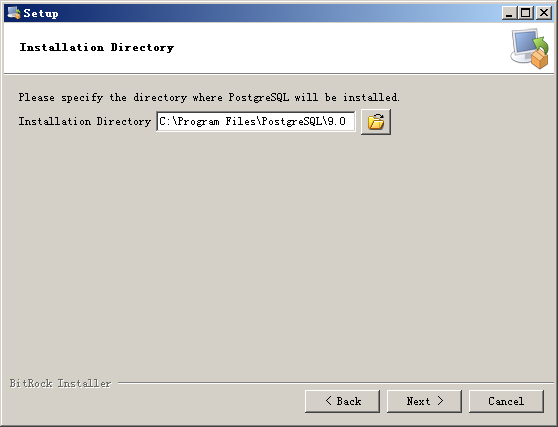


图3-3 PostgreSQL安装位置

在这一步，你可以选择你的PostgreSQL应用程序安装路径。通过点击打开文件夹按钮，选择安装路径。默认的安装路径为C:\Program Files\PostgreSQL\9.0。

点击下一步后，安装向导进入数据目录设置页面。如图3-4所示：

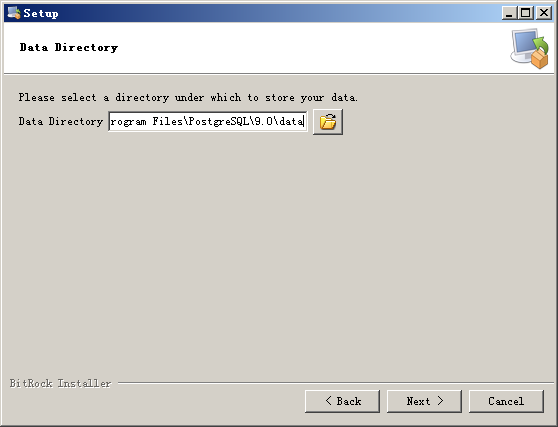


图3-4 PostgreSQL数据目录

这一步，你可以选择你的PostgreSQL数据存放的路径。通过点击打开文件夹按钮，选择存放路径。默认的存放路径为C:\Program Files\PostgreSQL\9.0\data。

点击下一步，安装程序进入填写密码页面，如图3-5所示：

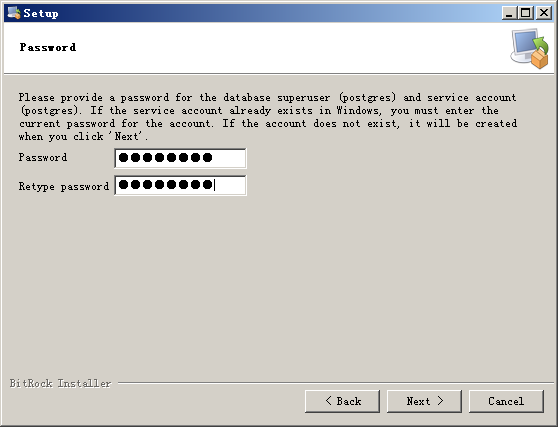


图3-5 输入postgres用户密码

和UNIX系统一样，PostgreSQL不允许以管理员身份运行程序，这避免了拥有管理员权限的用户运行接受网络连接的服务程序潜在的安全问题。如果发现了PostgreSQL的安全若得并被使用溢出攻击，这样做的最坏情况只是PostgreSQL管理的数据被损坏，而不是整台服务器被攻陷。安装程序会使用一个叫postgres的用户运行服务程序。在这一步，安装向导将记录postgres用户的密码。如果这台机器已经安装过PostgreSQL或者已经存在postgres用户，则输入的密码必须是这个用户现在的密码，否则，安装向导将建立postgres用户并设置密码为输入的密码。输入两次相同的密码以确保你记得你输入的密码。

点击下一步，安装向导要求你填入服务监听的端口，如图3-6所示：

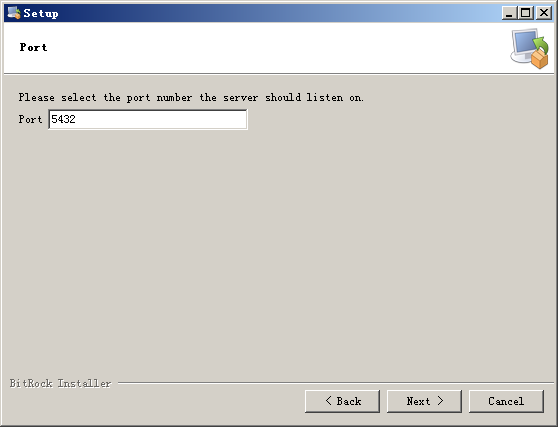


图3-6 输入服务监听端口

和UNIX下一样，默认的端口为5432，你可以设置为任何你想使用的端口。

下一步将设置新数据库的区域，如图3-7所示：

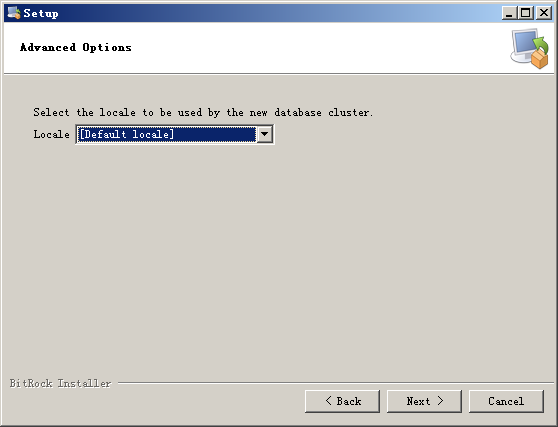


图3-7 选择新建数据库的区域

选择恰当的区域，点击下一步后，向导提示将开始安装数据库。再点击下一步，如果不出错，安装过程将进行并正常完成。完成安装后，数据库服务进程应该已经运行。服务进程postmaster.exe和postgres.exe可以在任务管理器中看到，如图3-8所示：

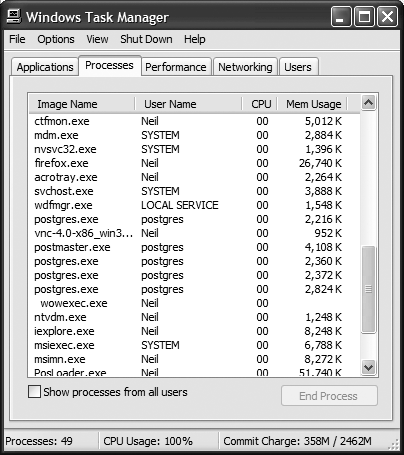


图3-8 PostgreSQL进程

PostgreSQL程序和工具被安装到开始菜单，如图3-8所示：

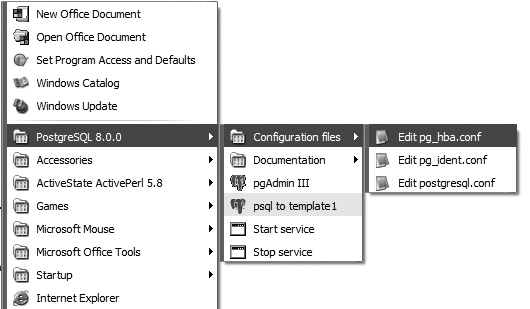


图 3-8 PostgreSQL程序菜单

### 配置客户机访问

为了配置远程主机和用户可以连接到PostgreSQL服务，你需要编辑pg\_hba.conf文件。文件包含大量注释记录用于远程访问的选项。在我们的安装示例中，我们允许局域网中任何主机的任何用户访问服务器上的数据库。为了达到这个目的，我们添加以下的一行记录到文件尾：

host all all 192.168.0.0/16 trust

这意味着所有IP地址由192.168开始的计算机可以访问所有的数据库。最简单的使配置生效的方法就是重启服务器。

Windows系统中的pg\_hba.conf文件和Linux和UNIX系统中的格式相同。其他的访问配置示例，请参考本章前面的“配置连接权限”小节。

## 建立示例数据库

现在我们已经让PostgreSQL运行起来了，我们将建立一个简单的数据库，我们给数据库起名为bpsimple，用来支持我们的客户订单表示例。这个数据库（与第8章建立的叫做bpfinal的改造版本）将在整个本书中使用。我们将在后面的章节全面讨论建立数据库和建表以及填充表的细节。在这里，我们只展示实现的步骤和SQL脚本，这样我们将有一个用于示例的数据库。在我们开始前，一个简单的检查PostgreSQL是否在运行的方法就是查找postmaster进程。在Windows系统中，通过任务管理器查找postmaster.exe进程。在UNIX和Linux系统中，运行以下的命令：

$ ps -el | grep post

如果有一个叫postmaster的进程在运行（名称可能是缩写显示），那么你已经在运行一个PostgreSQL服务程序了。

### 添加用户记录

在我们能建立一个数据库前，我们需要通过在系统中建立用户记录来告诉PostgreSQL有效的用户。PostgreSQL数据库系统的有效用户可以读数据，插入数据或者更新数据；建立自己的数据库以及控制对这些数据库管理的数据的访问。我们使用PostgreSQL的createuser工具建立用户记录。

在Linux和UNIX系统中，使用su命令（从root）变成PostgreSQL管理用户postgres。然后运行createuser注册用户。用户给出的用户登录名需要时有效的PostgreSQL用户。让我们为（UNIX/Linux的）用户neil建立数据库用户：

$ su

# su - postgres

pg$ /usr/local/pgsql/bin/createuser neil

Shall the new user be able to create databases? (y/n) y

Shall the new user be able to create new users (y/n) y

CREATE USER

pg$

在Windows系统中，打开一个命令行窗口并切换工作目录到PostgreSQL所在目录（在这里，我们按在默认位置：C:\Program Files\PostgreSQL\9.0），然后运行createuser.exe工具：

C:\Program Files\PostgreSQL\9.0\bin>createuser -U postgres -P neil

Enter password for new user:

Enter it again:

Shall the new user be allowed to create databases? (y/n) y

Shall the new user be allowed to create more new users? (y/n) y

Password:

CREATE USER

-U选项指出了你想要建立的新用户的编号。必须是PostgreSQL的用户才能建立用户，通常情况下就是postgres用户。-P选项通知createuser提示输入新用户的密码。

在这里，我们允许neil建立新数据库和建立新用户。本书中的一些例子中使用了另一个叫rick的用户，它允许建立数据库，但不允许新建用户。如果你想重新试试新建用户，现在就可以建立这个用户了。

一旦你建立了拥有这些权限的PostgreSQL用户，你就有能力建立bpsimple数据库了。

### 建立数据库

在Linux和UNIX系统中建立数据库，回到你自己的用户（非root用户）并运行以下命令：

$ /usr/local/pgsql/bin/createdb bpsimple

CREATE DATABASE

$

在Windows系统中，运行createdb.exe命令：

C:\Program Files\PostgreSQL\8.4\bin>createdb -U neil bpsimple

Password:

CREATE DATABASE

你先可以使用交互式终端工具psql（从本地）连接到服务器了。在Linux和UNIX系统中，使用以下命令：

$ /usr/local/pgsql/bin/psql -U neil -d bpsimple

psql (8.4.5)

Type "help" for help.

bpsimple =#

在Windows系统中，使用这个命令：

C:\Program Files\PostgreSQL\8.4\bin>psql -U neil -d bpsimple

Password:

Welcome to psql 8.4.5, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms

\h for help with SQL commands

\? for help with psql commands

\g or terminate with semicolon to execute query

\q to quit

Warning: Console codepage (850) differs from windows codepage (1252)

8-bit characters will not work correctly. See PostgreSQL

documentation "Installation on Windows" for details.

bpsimple=#

你也可以选择Windows开始菜单中的psql项目连接到template1数据库，然后在psql中切换数据库：

template1=# \c bpsimple

You are now connected to database "bpsimple".

bpsimple=#

现在你已经登录到PostgreSQL，准备好执行命令了。你可以使用\q命令退回到shell。

下一步，我们将使用一套SQL语句来建立和填充示例数据库。

### 建表

你可以在psql的命令提示符后面通过输入SQL命令在你的bpsimple数据库中建表。但是，更简单的办法是下载代码包并解压后，通过命令\i <文件名>执行这些命令（psql中的\i命令可以用来执行脚本文件，脚本文件是由数组SQL语句和其他PostgreSQL命令组成的文本文件）。由于命令都是纯文本，所以只要你愿意你都可以以通过你的文本编辑器编辑它。

输入下面的命令运行create\_tables-bpsimple.sql来建表：

bpsimple=# \i create\_tables-bpsimple.sql

CREATE TABLE

...

bpsimple=#

把所有的数据库模式（表，索引以及存储过程）写入脚本文件是一个好习惯。只有这样，如果数据库需要重新建立，你可以通过脚本完成。脚本也可以用于在任何模式需要更新时使用。

以下为建立我们需要的表的SQL（第2章设计的表）这些SQL可以在create\_tables-bpsimple.sql代码包里头找到：

CREATE TABLE customer

(

customer\_id serial ,

title char(4) ,

fname varchar(32) ,

lname varchar(32) NOT NULL,

addressline varchar(64) ,

town varchar(32) ,

zipcode char(10) NOT NULL,

phone varchar(16) ,

CONSTRAINT customer\_pk PRIMARY KEY(customer\_id)

);

CREATE TABLE item

(

item\_id serial ,

description varchar(64) NOT NULL,

cost\_price numeric(7,2) ,

sell\_price numeric(7,2) ,

CONSTRAINT item\_pk PRIMARY KEY(item\_id)

);

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL,

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id)

);

CREATE TABLE stock

(

item\_id integer NOT NULL,

quantity integer NOT NULL,

CONSTRAINT stock\_pk PRIMARY KEY(item\_id)

);

CREATE TABLE orderline

(

orderinfo\_id integer NOT NULL,

item\_id integer NOT NULL,

quantity integer NOT NULL,

CONSTRAINT orderline\_pk PRIMARY KEY(orderinfo\_id, item\_id)

);

CREATE TABLE barcode

(

barcode\_ean char(13) NOT NULL,

item\_id integer NOT NULL,

CONSTRAINT barcode\_pk PRIMARY KEY(barcode\_ean)

);

### 移除表

假设以后的某天，你需要删除所有的表并重新开始，命令集就在drop\_tables.sql，就像这样：

DROP TABLE barcode;

DROP TABLE orderline;

DROP TABLE stock;

DROP TABLE orderinfo;

DROP TABLE item;

DROP TABLE customer;

DROP SEQUENCE customer\_customer\_id\_seq;

DROP SEQUENCE item\_item\_id\_seq;

DROP SEQUENCE orderinfo\_orderinfo\_id\_seq;

警告你一声，如果你删掉表，你也将丢失表里的数据！

注：drop\_tables.sql脚本也明确地删除了叫做序列生成器的特殊的属性，它被PostgreSQL用来管理数据自动顺序增长的列。在PostgreSQL 8.0和以后的版本，序列生成器将在相关表被删除时被自动删除，但我们保留了命令用于兼容以前的版本。

如果你在建表后运行了这个脚本你需要在尝试往表里头填充数据前再次运行create\_tables-bpsimple.sql。

### 填充表

最后，我们需要往表里头添加一些数据，或者叫做填充表。示例数据在pop\_tablename.sql里头。如果你想使用自己的数据，你得到的结果将和本书提供的不同。所以，除非你确信，否则最好是使用我们提供的示例数据。

在这里使用换行符的必要性是为了将命令打印在打印页上。你可以每个每行一条命令。你必须包含结尾的分号，这告诉psql每行SQL的结束点。

#### customer表

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Miss','Jenny','Stones','27 Rowan Avenue','Hightown','NT2 1AQ','023 9876');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Andrew','Stones','52 The Willows','Lowtown','LT5 7RA','876 3527');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Miss','Alex','Matthew','4 The Street','Nicetown','NT2 2TX','010 4567');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Adrian','Matthew','The Barn','Yuleville','YV67 2WR','487 3871');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Simon','Cozens','7 Shady Lane','Oakenham','OA3 6QW','514 5926');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Neil','Matthew','5 Pasture Lane','Nicetown','NT3 7RT','267 1232');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Richard','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mrs','Ann','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mrs','Christine','Hickman','36 Queen Street','Histon','HT3 5EM','342 5432');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Mike','Howard','86 Dysart Street','Tibsville','TB3 7FG','505 5482');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Dave','Jones','54 Vale Rise','Bingham','BG3 8GD','342 8264');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Richard','Neill','42 Thatched Way','Winersby','WB3 6GQ','505 6482');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mrs','Laura','Hardy','73 Margarita Way','Oxbridge','OX2 3HX','821 2335');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Bill','O\'Neill','2 Beamer Street','Welltown','WT3 8GM','435 1234');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','David','Hudson','4 The Square','Milltown','MT2 6RT','961 4526');

#### item表

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Wood Puzzle', 15.23, 21.95);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Rubik Cube', 7.45, 11.49);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Linux CD', 1.99, 2.49);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Tissues', 2.11, 3.99);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Picture Frame', 7.54, 9.95);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Fan Small', 9.23, 15.75);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Fan Large', 13.36, 19.95);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Toothbrush', 0.75, 1.45);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Roman Coin', 2.34, 2.45);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Carrier Bag', 0.01, 0.0);

INSERT INTO item(description, cost\_price, sell\_price)

VALUES('Speakers', 19.73, 25.32);

#### barcode表

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6241527836173', 1);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6241574635234', 2);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6264537836173', 3);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6241527746363', 3);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('7465743843764', 4);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('3453458677628', 5);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6434564564544', 6);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('8476736836876', 7);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('6241234586487', 8);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('9473625532534', 8);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('9473627464543', 8);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('4587263646878', 9);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('9879879837489', 11);

INSERT INTO barcode(barcode\_ean, item\_id) VALUES('2239872376872', 11);

#### orderinfo表

INSERT INTO orderinfo(customer\_id, date\_placed, date\_shipped, shipping)

VALUES(3,'03-13-2000','03-17-2000', 2.99);

INSERT INTO orderinfo(customer\_id, date\_placed, date\_shipped, shipping)

VALUES(8,'06-23-2000','06-24-2000', 0.00);

INSERT INTO orderinfo(customer\_id, date\_placed, date\_shipped, shipping)

VALUES(15,'09-02-2000','09-12-2000', 3.99);

INSERT INTO orderinfo(customer\_id, date\_placed, date\_shipped, shipping)

VALUES(13,'09-03-2000','09-10-2000', 2.99);

INSERT INTO orderinfo(customer\_id, date\_placed, date\_shipped, shipping)

VALUES(8,'07-21-2000','07-24-2000', 0.00);

#### orderline表

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(1, 4, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(1, 7, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(1, 9, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(2, 1, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(2, 10, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(2, 7, 2);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(2, 4, 2);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(3, 2, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(3, 1, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(4, 5, 2);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(5, 1, 1);

INSERT INTO orderline(orderinfo\_id, item\_id, quantity) VALUES(5, 3, 1);

#### stock表

INSERT INTO stock(item\_id, quantity) VALUES(1,12);

INSERT INTO stock(item\_id, quantity) VALUES(2,2);

INSERT INTO stock(item\_id, quantity) VALUES(4,8);

INSERT INTO stock(item\_id, quantity) VALUES(5,3);

INSERT INTO stock(item\_id, quantity) VALUES(7,8);

INSERT INTO stock(item\_id, quantity) VALUES(8,18);

INSERT INTO stock(item\_id, quantity) VALUES(10,1);

在PostgreSQL系统运行，数据库建立表建立以及填充数据后，我们准备好继续探索PostgreSQL的功能了。

## 摘要

在本章中，我们了解了在Linux和类UNIX系统以及Windows中安装PostgreSQL数据库的一些选秀。最简单的方法是用某种预编译的二进制安装包。我们提供了在Linux系统中用安装包，类UNIX系统中用源码以及Windows系统中使用安装包的每一步的指令用于编译、安装、配置和确认安装一个可运行的环境。

最后，我们建立了一个我们将在本书剩下的部分中用于展示PostgreSQL系统功能的示例数据库。我们将从下章的访问你的数据开始，探索PostgreSQL系统。

# 访问你的数据

到现在为止，我们在本书中碰到SQL的情况都是非正式的。我们碰到过一些用不同方法获取数据的语句，以及一些用于建表和填充数据的SQL。

本章，我们将从SELECT语句开始，稍微正式地研究SQL。事实上，本章全部都是用于讲解SQL语句。你的第一映像可能认为用一章讲解一条SQL有点过分，但实际上SELECT语句是SQL语句的核心。一旦你理解了SELECT，你实际上已经完成学习SQL最难的部分。

在下一章，我们将讨论一些你可以使用的GUI客户端，但现在，我们将使用一个随PostgreSQL提供的简单的命令行工具psql来访问数据。

本章我们将涵盖以下主题：

* 使用psql命令与PostgreSQL数据库交互
* 使用一些简单的SELECT语句检索数据
* 通过覆盖列名提高输出的可读性
* 在检索数据的时候控制数据行的顺序
* 隐藏重复行
* 在检索数据的时候执行数学计算
* 为了方便而给数据库别名
* 使用模式匹配指出需要检索的数据
* 使用各种数据类型进行匹配
* 通过单个的SELECT语句在多个表中检索数据
* 在一条SELECT语句中关联三个甚至跟多的表

到现在为止，你应该已经有搭好PostgreSQL并且在运行了。通过这一章，我们将使用在第二章设计并在第三章生成的示例数据库。

## 使用psql

如果你跟着第三章里头的指令做了，现在你应该有一个你可以通过PostgreSQL的登录提示符访问的叫做bpsimple的数据库了。

注：除了进行一些特殊的数据库管理操作，你永远都不应该使用postgres用户访问PostgreSQL数据库。

### 在Linux系统中启动

如果你是在一个Linux系统中，并且你建立了一个普通的没有密码的用户，你可以通过在连接命令中包含用户名来启动psql来访问bpsimple数据库。例如，使用rick用户访问数据库，你可以输入：

$ psql -d bpsimple -U rick

你应该会看到以下内容：

Welcome to psql, the PostgreSQL interactive terminal.  
  
Type: \copyright for distribution terms  
 \h for help with SQL commands  
 \? for help on internal slash commands

\g or terminate with semicolon to execute query

\q to quit

bpsimple=>

现在我们已经准备好输入命令了。如果你建立的用户需要密码，你应该收到输入密码的提示，这依赖于实际的认证配置。我们将在第十一章解释更多关于认证的内容。

### 在Windows系统中启动

如果你是使用Windows，可以通过打开开始菜单的psql首先连接到template1。你将受到提示输入postgres用户密码的提示。在成功连接后，使用\c命令使用你自己的用户名（在这里为rick）切换到bpsimple数据库，如下所示：

template1=# \c bpsimple rick

You are now connected to database "bpsimple" as user "rick".

bpsimple=>

注意提示符从=#变成了=>，这意味着你不再拥有权限建立数据库。

你也可以选择自己在开始菜单建立这个命令的快捷方式。例如，我们通过IP地址（-h选项）作为risc用户（-U选项）连接到远程服务器的bpsimple数据库（-d选项）。

"C:\Program Files\PostgreSQL\8.4\bin\psql.exe" -h 192.168.0.3 -d bpsimple -U rick

如果你想作为管理员用户连接，你可以用postgres代替rick。如果你需要连接到本地的服务器，你可以忽略掉-h选项。

### 解决启动问题

如果psql啰嗦关于pg\_shadow的问题，可能是你没有在数据库建立你提供的用户。如果抱怨不知道这个用户或者缺乏许可，可能是你没有正确赋权。请参考第三章关于赋权的细节。

如果你还是没成功，现在最简单的修复问题的方法是删除数据库和用户并重建他们。现在的做法是使用\q命令退出psql命令，然后你就退出到Linux的命令提示符或者退出了Windows的命令行窗口。

之后，使用postgres用户重新连接到数据库。在Linux中，在命令行按照下面的方法做：

$ psql -d template1

在Windows中，使用开始菜单的快捷方式连接psql到template1数据库。

输入你在安装过程中输入的密码，然后你会看到以下的提示符：

template1=#

现在，可以通过以下的形式删除数据库和用户（本例中的用户为rick）：

template1=# DROP DATABASE bpsimple;  
DROP DATABASE  
template1=# DROP USER rick;  
DROP USER  
template1=#

然后重建用户并给出密码（本例中的密码为apress4789）：

template1=# CREATE USER rick WITH CREATEDB PASSWORD 'apress4789';

CREATE USER

template1=#

CREATEDB选项允许用户建立他们自己的数据库。

现在使用你新建的用户连接到数据库：

template1=# \c template1 rick

Password:

template1=#

你现用你的新用户（在本例中还是rick这个用户）连接到数据库template1。下一步是建立bpsimple数据库：

template1=> CREATE DATABASE bpsimple;

CREATE DATABASE

template1=>

使用新用户重新连接到bpsimple数据库：

template1=> \c bpsimple rick

You are now connected to database "bpsimple" as user "rick".

bpsimple=>

然后你需要回到上一章从“建表”小节到结尾位置的步骤，以便建立本章需要的示例数据库并填充数据。

如果你尝试建立数据库的时候看到下面的错误信息：

ERROR: source database "template1" is being accessed by other users

这意味着有其他的会话连接到了数据库template1——也许是另一个psql的会话或者pgAdmin III这样的图形界面程序。请确保你当前的psql会话是唯一使用template1数据库的会话并重试。

要检查你是否已经建立了bpsimple数据库里头的表，数据如\dt，然后敲回车键，然后你应该可以看到类似于以下的输出：

bpsimple=> \dt

List of relations

Schema | Name | Type | Owner

-------+--------------+-------+----------

public | barcode | table | rick

public | customer | table | rick

public | item | table | rick

public | orderinfo | table | rick

public | orderline | table | rick

public | stock | table | rick

(6 rows)

bpsimple=>

Owner列显示为你的登录名（本例中为rick）。

注：你可能看到一些名字为pg\_ts\_dict，pg\_ts\_parser，pg\_ts\_cfg以及pg\_ts\_cfgmap一类的表。这些表可能是一些可选的用户捐献的工具的附加表。你可以安全地忽略它们。

你也可以通过pgAdmin III浏览数据库时看到同样的信息，先是数据库bpsimple，然后是模式，然后是public，然后是表，就像图4-1所示。

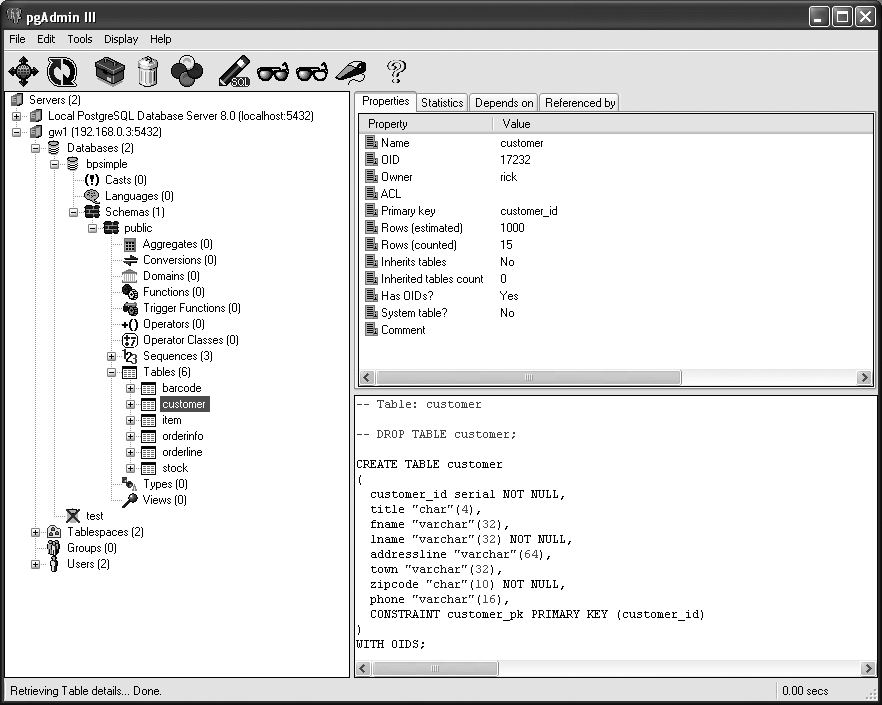


图4-1 通过pgAdmin III检查bpsimple数据库

我们将在第十一章覆盖绝大多数数据库和用户管理的细节。

### 使用一些基本的psql命令

在本章中我们只会使用很少的几个psql命令（我们将在第五章碰到全部的命令）。现在，你需要知道的命令就列在表4-1中。

表4-1 psql的基本命令

|  |  |
| --- | --- |
| 命令 | 描述 |
| \? | 获得帮助消息 |
| \do | 列出操作类型 |
| \dt | 列出表 |
| \dT | 列出类型 |
| \h <cmd> | 列出SQL命令的帮助；用实际的命令代替<cmd> |
| \i <filename> | 执行文件<filename>里头的命令 |
| \r | 重置缓冲器（忽略任何输入·····） |
| \q | 退出psql |

表4-1中列出的命令必须在输入回车后才能执行。

你还可以通过使用方向键回到之前的行并编辑它们。在Linux系统中，psql的这个功能依赖于GNU的readline基础设施，通常它已经被安装，但不保证总是已经安装。

现在我们已经准备好通过SQL命令访问PostgreSQL数据库。在下一章，我们将碰到一些可以使用PostgreSQL的图形界面工具，但是在本章，我们将使用psql工具。

注：如果你坚持使用图形界面工具，你可以先看看第五章。然后你可以回到本章。你应该可以通过任何图形界面工具通过直接输入SQL到PostgreSQL执行本章的所有例子，例如pgAdmin III（http://www.ggadmin.org/）。但是，我们建议至少在这章中使用命令行，因为知道如何使用命令行工具访问PostgreSQL的基本方法通常会非常方便。

## 使用SELECT语句

和所有关系数据库一样，我们通过SELECT语句从PostgreSQL里检索数据。它也许是最复杂的SQL语句，事实上它才是有效使用关系数据库的心脏。

让我们从简单的从一个表中的全部数据开始研究SELECT命令。我们通过很基础的形式的SELECT语句，指出一个列的列表，以及一个带有表名的FORM从句：

SELECT <逗号分隔的列的列表> FROM <表名>

如果我们不记得确定需要的列名，或者想要看到所有的列，我们可以直接使用一个星号（\*）来代替列的列表。

注：本书中，我们使用大写的SQL关键字以便让它们更突出。SQL不是区分大小写的，仅有少数数据库软件对于表名是区分大小写的。SQL数据库存储的数据是区分大小写的，所以字符串“Newtown”和“newtown”是不同的。

##### 尝试：从表中选择所有的列

我们开始于从item表中提取所有数据：

SELECT \* FROM item;

记住分号（;）对于psql非常有用，它可以告诉psql输入结束了。严格地说，它不是SQL的一部分。如果你愿意，你可以通过输入\g命令告诉psql SQL语句已经结束，这和分号的效果一样。如果你使用不同的工具发送SQL给PostgreSQL，你可以不需要任何一种终止符。

输入命令后，你会看到PostgreSQL的回应：

bpsimple=> SELECT \* FROM item;

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

1 | Wood Puzzle | 15.23 | 21.95

2 | Rubic Cube | 7.45 | 11.49

3 | Linux CD | 1.99 | 2.49

4 | Tissues | 2.11 | 3.99

5 | Picture Frame | 7.54 | 9.95

6 | Fan Small | 9.23 | 15.75

7 | Fan Large | 13.36 | 19.95

8 | Toothbrush | 0.75 | 1.45

9 | Roman Coin | 2.34 | 2.45

10 | Carrier Bag | 0.01 | 0.00

11 | Speakers | 19.73 | 25.32

(11 rows)

bpsimple=>

##### 解析

我们使用\*作为列名，简单的告诉PostgreSQL我们需要item表里头的所有列的数据。PostgreSQL正确地给了我们想要的数据，还整齐地将列名用管道符号（|）区分列的方式排列了数据。它还告诉我们检索到了多少行。

但假设我们不需要所有的列呢？通常，你应该告诉PostgreSQL甚至实际上所有的关系数据库仅仅检索你实际上需要的数据。检索每行数据的每列都会增加一些工作。没有必要让服务器做不必要的工作；有必要总是让事情清晰有效。

你还会发现，一旦你开始将SQL嵌入其他语言（参考第十四章），指定列名将保护你防止因数据库模式的改变而带来的问题。例如，如果你使用\*来检索所有的列而在代码被测试后一个列被插入到表中后，你会发现你处理的你想要的列中的数据是从另一个列里头来的了。如果你使用的列被删除了，那么你程序中的SQL将执行失败，因为列已经无法被检索了；但是，这是一个很容易找到并纠正的错误，而不需要应用程序在处理数据时访问错误的列的时候才发现。如果你指定列名，你可以在做数据库变更前搜索你所有的代码中看是否出现这个列名，来避免发生这类错误。

让我们试试只检索我们需要的列吧。就像前面的语法中见到的一样，我们通过指定我们需要的用逗号分隔的列做到。如果我们需要的列的顺序不是我们建表时指定的顺序，那么也没问题——我们可以按我们的需要指定列的顺序，然后数据库就会按照这个顺序返回数据。

##### 尝试：按照特定的顺序选择列

为了检索所有客户所在的城镇名字和他们的姓，我们必须指定城镇和姓所在的列的名字，当然，还要指出检索这些数据所在的表。以下是我们需要的语句以及PostgreSQL的响应：

bpsimple=> SELECT town, lname FROM customer;

town | lname

-----------+---------

Hightown | Stones

Lowtown | Stones

Nicetown | Matthew

Yuleville | Matthew

Oakenham | Cozens

Nicetown | Matthew

Bingham | Stones

Bingham | Stones

Histon | Hickman

Tibsville | Howard

Bingham | Jones

Winersby | Neill

Oxbridge | Hardy

Welltown | O'Neill

Milltown | Hudson

(15 rows)

bpsimple=>

##### 解析

PostgreSQL从我们指定的表返回了我们所有的数据行，但只返回了我们需要的列。它还按照我们在SELECT语句中指定的列的顺序返回列的数据。

### 覆盖列名

你会发现输出的结果使用列的名字作为输出的列标题。有时候它不易于阅读，尤其是当输出的列不是数据库的列，它没有名字。有一个非常简单的语法指定用于显示的列名（列的别名），方法是在SELECT语句的每个列后面添加AS "<*显示名*>"。你可以指定所有的列的别名，或者是仅仅选择几个。你不指定别名的地方，PostgreSQL就仅仅使用列名。

例如，通过添加有意义的名字改变前面输出，我们可以这样：

SELECT town, lname AS "Last Name" FROM customer;

我们将看在下一章看到它的示例。值得注意的是，在SQL92标准中，AS从句是可选的；但是，到8.0版本为止，PostgreSQL仍然需要AS关键字。

### 控制行的顺序

到现在为止我们已经按列检索到了我们需要的数据，但这些数据看起来还不是按最适当的方式显示的。我们看到的数据可能看上去是我们当初插入数据库的顺序，但这看上去很简单是因为我们还没有通过插入和删除行更新过数据。

就像我们在第二章提到的，不像电子表格，数据库中的行的顺序不需要指定。数据库服务器可以随意按照自己最高效的方法存储行，但这通常不是最自然的方法来查看数据。你看到的输出是按没有任何意义的顺序排序的，并且可能下次你重新请求这些数据的时候，它可能是按另一种顺序显示的。通常，数据将按其在数据库内部存储的顺序返回。包含PostgreSQL在内的所有SQL数据库，没有一个被要求按照一定的顺序返回数据，除非你在检索它们的时候指必须按顺序。

我们可以通过在SELECT语句后面添加ORDER BY从句控制SELECT语句显示的数据的顺序，ORDER BY从句指定我们需要的数据返回的顺序。语法如下所示：

SELECT <都好分隔的列名列表> FROM <表名> ORDER BY <列名> [ASC | DESC]

在最后面的有点奇怪的语法的意思是，在列名后面，我们可以选择ASC（升序排序）或者DESC（降序排序）之一。默认使用的是升序排序。之后数据按照我们指定的列和要求的顺序返回。

##### 尝试：给数据排序

在本例中，我们按town排序数据，同时我们将覆盖lname列的名字，就像我们上一节所说的，让输出看上去更易读。

以下是我们需要的命令以及PostgreSQL的响应：

bpsimple=> SELECT town, lname AS "Last Name" FROM customer ORDER BY town;

town | Last Name

-----------+-----------

Bingham | Stones

Bingham | Stones

Bingham | Jones

Hightown | Stones

Histon | Hickman

Lowtown | Stones

Milltown | Hudson

Nicetown | Matthew

Nicetown | Matthew

Oakenham | Cozens

Oxbridge | Hardy

Tibsville | Howard

Welltown | O'Neill

Winnersby | Neill

Yuleville | Matthew

(15 rows)

bpsimple=>

注意因为我们需要数据按升序排列，我们可以省略ASC，因为升序排列是默认的排序顺序。你可以看到，数据是按城镇的升序排列的。

##### 解析

现在，我们在我们最初的语句中做了两处变化。我们添加了AS从句用将第二列的名字改为“Last Name”，这让结果更易读，我们还添加了一个ORDER BY子句用于指定PostgreSQL返回数据给我们的顺序。

有时候，我们需要更进一步，按照不止一个列排序。例如，在之前的输出，虽然数据时按照town排序的，但没有按照Last Name排序。比方说我们可能发现所有的town为bingham的客户中，Jones的名字会出现在Stones的后面。

我们可以通过指定超过一个列的顺序细化输出的顺序。如果我们需要，我们甚至可以指定一列按升序排列同时另一列按降序排列。

##### 尝试：用ASC和DESC给数据排序

再试试我们的SELECT命令，只是这次，我们需要指定城镇名字降序排列，相同城镇的姓升序排列。

我们需要的语句和PostgreSQL的响应如下：

bpsimple=> SELECT town, lname AS 'Last Name' FROM customer

ORDER BY town DESC, lname ASC;

town | Last Name

-----------+-----------

Yuleville | Matthew

Winnersby | Neill

Welltown | O'Neill

Tibsville | Howard

Oxbridge | Hardy

Oakenham | Cozens

Nicetown | Matthew

Nicetown | Matthew

Milltown | Hudson

Lowtown | Stones

Histon | Hickman

Hightown | Stones

Bingham | Jones

Bingham | Stones

Bingham | Stones

(15 rows)

bpsimple=>

##### 解析

就像你看到的，PostgreSQL先将数据按town的降序排列，因为它是ORDER BY从句后面的第一个列。然后它将相同town里头有多条记录的数据按升序排列。这时候，虽然Bingham是在检索到的数据的末尾，但其中的客户的姓仍然是按升序排列的。

通常，你可以用来排序的列被强制限制于你选择用于输出的列（看上去不无道理）。但至少到当前版本的PostgreSQL未知，不强制执行这一标准限制，它可以接受用于ORDER BY后面的列不在你选择的列的列表里头的情况。但是，这是非标准SQL，所以我们建议你避免使用这个功能。

### 消除重复数据

你可能发现之前的输出有好几处重复行。例如，以下的城镇和姓出现了两次：

Nicetown | Matthew

Bingham | Stones

这里发生什么了？在原始数据中，确实在Nictown有两个叫做Matthew的客户和在Bingham有两个角Stones的客户。作为参考，以下为显示了名的相应行：

Nicetown | Matthew | Alex

Nicetown | Matthew | Neil

Bingham | Stones | Richard

Bingham | Stones | Ann

PostgreSQL分别列出了Nicetown和Matthew的两行，以及Bingham和Stones的两行，而且应该都是正确的。每个镇都有同名的两个客户。因为我们没有说要列出所有能够区别它们的列，所以它们看上去似乎完全一样。

数据库默认的行为是列出所有的行但这不总是我们所需要的。例如，我们可能只是列出有我们客户的城镇，也许是用来决定我们需要在哪里建立分发中心。基于我们现在的知识，我们可能尝试这么做：

bpsimple=> SELECT town FROM customer ORDER BY town;

town

-----------

Bingham

Bingham

Bingham

Hightown

Histon

Lowtown

Milltown

Nicetown

Nicetown

Oakenham

Oxbridge

Tibsville

Welltown

Winersby

Yuleville

(15 rows)

bpsimple=>

PostgreSQL列出了所有的城镇，在客户表中每出现一次就列了一次。这没错，但它可能不是不太是我们想要的。我们真正需要的列表是每个城镇出现一次；或者说，我们需要列出不同的镇。

在SQL中，你可以在SELECT语句中添加DISTINCT关键字来移除重复的行，语法如下：

SELECT DISTINCT <由逗号分隔的列的列表> FROM <表名>

和SELECT的所有从句一样，你可以将这个从句与其他从句组合起来使用，例如重命名列或者指定排序方式。

##### 尝试：使用DISTINCT

来尝试列出customer表里出现的所有不重复的town来。我们可以尝试使用以下的代码来获得回应：

bpsimple=> SELECT DISTINCT town FROM customer;

town

-----------

Bingham

Hightown

Histon

Lowtown

Milltown

Nicetown

Oakenham

Oxbridge

Tibsville

Welltown

Winersby

Yuleville

(12 rows)

bpsimple=>

##### 解析

关键字DISTINCT告诉PostgreSQL移除重复的行。注意输出已经按town排序了，这是因为PostgreSQL选择西安排序的方法实现DISTINCT从句。通常，我们不能假设数据总是被排序的。

如果你需要数据按照特殊的方式排序，你必须添加一个ORDER BY从句来指定顺序。

注意DISTINCT从句不是仅仅关联到某几个列。你只能去除你选择的所有列中的重复数据，而不能仅仅去除部分列的重复数据。例如，假设我们使用这种方式：

SELECT DISTINCT town, fname FROM customer;

我们仍将获得15行，因为有15种城镇和姓的组合。

这里要做一个小小的警告：虽然看上去总是在SELECT中使用DISTINCT是个好主意，在现实中，这是一个坏主意。第一，使用DISTINCT，意味着PostgreSQL需要在提取你的数据的时候需要做更多的事情来查重。除非你知道需要移除的重复数据，否则你不应该使用DISTINCT从句。第二个原因更加注重于实用。有时候DISTINCT会隐藏你SQL或者数据中的错误，而这种错误在显示重复行的时候却很容易看出。

警告：仅仅在你确定需要的时候使用DISTINC，因为它需要更多的工作且可能隐藏错误。

## 执行计算

我们还可以在数据被发送出来前对数据执行一些简单的计算。

假设我们需要显示我们item表里头的每个项目的价格，我们只需要简单的执行如下的SELECT语句：

bpsimple=> SELECT description, cost\_price FROM item;

description | cost\_price

---------------+-------------

Wood Puzzle | 15.23

Rubic Cube | 7.45

Linux CD | 1.99

Tissues | 2.11

Picture Frame | 7.54

Fan Small | 9.23

Fan Large | 13.36

Toothbrush | 0.75

Roman Coin | 2.34

Carrier Bag | 0.01

Speakers | 19.73

(11 rows)

bpsimple=>

假设我们需要按分查看它们的价格，我们可以在SQL中作简单的计算，就像这样：

bpsimple=> SELECT description, cost\_price \* 100 FROM item;

description | ?column?

---------------+--------------

Wood Puzzle | 1523.00

Rubic Cube | 745.00

Linux CD | 199.00

Tissues | 211.00

Picture Frame | 754.00

Fan Small | 923.00

Fan Large | 1336.00

Toothbrush | 75.00

Roman Coin | 234.00

Carrier Bag | 1.00

Speakers | 1973.00

(11 rows)

bpsimple=>

输出的小数点和奇怪的列名看上去有点古怪，所以让我们通过SQL的一个功能去掉小数点，同时明确地给出结果列的名字。我们使用cast函数转换列的类型，这与使用AS的给列命名的从句一同，让输出更好看一点：

bpsimple=> SELECT description, cast((cost\_price \* 100) AS int AS "Cost Price"

FROM item;

description | Cost Price

--------------+--------------

Wood Puzzle | 1523

Rubic Cube | 745

Linux CD | 199

Tissues | 211

Picture Frame | 754

Fan Small | 923

Fan Large | 1336

Toothbrush | 75

Roman Coin | 234

Carrier Bag | 1

Speakers | 1973

(11 rows)

bpsimple=>

我们将在本章稍后的“设置时间和日期格式”的小节详细讨论cast函数的功能。

## 选择行

本章到现在为止，我们总是工作在所有的数据行上，或者至少是所有的不同行。是时候看看我们怎样选择我们需要的行了。你可能不会觉得奇怪因为我们将要学习一个新的SELECT语句的从句：WHERE从句。

WHERE的最简单的语法如下：

SELECT <逗号分隔的列> FROM <表名> WHERE <条件>

可能有很多可以通过关键字AND、OR以及NOT组合的条件。

条件里头使用的标准的比较运算符列在表4-2中。比较运算符可以用在大多数类型中，包括数字和字符串，甚至一些特殊的条件例如日期比较的条件，这些我们将在本章稍后看到。

表4-2标准比较运算符

|  |  |
| --- | --- |
| 运算符 | 描述 |
| < | 小于 |
| <= | 小于或等于 |
| = | 等于 |
| >= | 大于或等于 |
| > | 大于 |
| <> | 不等于 |

我们将开始于一个简单的条件：选择住在Bingham的所有人：

bpsimple=> SELECT town, lname, fname FROM customer WHERE town = 'Bingham';

town | lname | fname

--------+--------+---------

Bingham | Stones | Richard

Bingham | Stones | Ann

Bingham | Jones | Dave

(3 rows)

bpsimple=>

这非常简单易懂，不是吗？注意需要用单引号包括字符串Bingham以表明它是一个字符串。还要注意Bingham因为是与数据库中的数据比较，因此它是区分大小写的。如果我们使用了town = 'bingham'，将没有数据返回。

我们可以有多个条件，通过AND，OR以及NOT组合起来，并添加括号使表达式看上去更清晰。我们也可以针对我们没有选择的列使用条件。你应该还记得这对于ORDER BY从句来说是不适用的。

##### 尝试：使用运算符

让我们尝试一些更复杂的条件集。假设我们需要看看住在Bingham或者Nicetown的title不是Mr.的客户。以下是我们需要的语句以及PostgreSQL的回应：

bpsimple=> SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr'

bpsimple-> AND (town = 'Bingham' OR town = 'Nicetown');

title | fname | lname | town

-------+-------+---------+----------

Miss | Alex | Matthew | Nicetown

Mrs | Ann | Stones | Bingham

(2 rows)

bpsimple=>

##### 解析

虽然它咋一眼看起来有点复杂，但这个语句实际上非常简单。第一部分就是我们普通的SELECT，列出我们在输出中需要看到的列。在WHERE从句之后，我们先检查title不是Mr.，然后使用AND连接我们下一个条件是否为真。第二个条件是town可以是Bingham或者Nicetown。注意我们需要用括号使需要分组的从句看起来更清晰。

你应该意识到PostgreSQL，甚至其他任何的关系数据库，不保证按照你写在SQL语句里头的从句的顺序处理。它只保证针对SQL提出的“问题”得出正确的答案。通常，关系数据库都有精密的优化器，它检查请求，然后确定能完成它的最优方法。优化器不是完美的，所以你经常需要通过不同的方式写语句以获得更好的运行效果。对于像这里的语句一样的相当简单的语句，我们可以很安全地假设优化器可以做得很好。

提示：如果你想知道PostgreSQL怎么处理SQL语句，你可以通过在SQL前头加EXPLAIN前缀让它告诉你。而不是执行这个语句。PostgreSQL将告诉你语句将怎么处理。

### 使用更复杂的条件

我们经常需要的一个功能是需要对字符串进行部分匹配。例如，我们需要找一个叫Bobert的人，但在数据库里头可能保存的是缩写的Rob或者Bob一类的。在SQL中有一些特殊的操作让我们针对字符串的一部分或者字符串的列表工作起来更轻松。

第一个新条件是IN，它允许我们针对一系列的条目进行检查，而不是使用一系列的OR条件。看看以下的语句：

SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND

(town = 'Bingham' OR town = 'Nicetown');

我们可以将它们重写为以下的样子：

SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND

town IN ('Bingham', 'Nicetown');

我们将获得相同的结果，虽然有可能输出的行的顺序可能不同，因为我们没有使用一个ORDER BY从句。在这里，IN从句除了可以简化表达式外没有特别的优势。当我们在第七章遇到子查询的时候，我们会再次使用IN，在那种情况下IN从句会提供更高级的优势。

下一个新的条件是BETWEEN，它允许我们通过指定边界检查一个值的范围。假设我们想要选择customer\_id在5和9之间的行。与其使用一系列的OR条件或者一个IN带着一堆值，不然我们写一个简单的条件如下：

bpsimple=> SELECT customer\_id, town, lname FROM customer WHERE customer\_id

BETWEEN 5 AND 9;

customer\_id | town | lname

-------------+----------+---------

5 | Oahenham | Cozens

6 | Nicetown | Matthew

7 | Bingham | Stones

8 | Bingham | Stones

9 | Histon | Hickman

(5 rows)

bpsimple=>

也可以针对字符串使用BETWEEN；但是，我们必须小心，因为有可能结果并不完全是你所期望的，而且你必须知道原因，就像之前提起的，因为字符串的比较是区分大小写的。

##### 尝试：使用复杂的条件

让我们尝尝使用BETWEEN比较字符串。假设我们想获得一个首字母为B和N之间的所有不同城镇的列表。所有的城镇的开始字母都是大写的，所以我们可以依照以下的写法：

bpsimple=> SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'N';

town

----------

Bingham

Hightown

Histon

Lowtown

Milltown

(5 rows)

bpsimple=>

如果你仔细地看结果，你会发现这个SQL没有达到如期的效果。Newtown在哪里？它当然是以N开始的，却没有被列出来。

#### 它为什么没有生效

这条语句没有达到我们效果的原因是PostgreSQL依照SQL标准，用填充你给的字符串到需要比较的字符串的长度然后再比较。所以当我们比较Newtown的时候，PostgreSQL用“N ”（N后面填充了六个空白）和Newtown，而空白在ASCII表中的位置再其他所有字符之前，所以判断的结果是Newtown比“N ”大，所以它不应该在结果列表中

#### 怎么让它生效

其实很容易让它像预期的一样工作。我们既可以通过在N后面添加字符z也可以在BETWEEN从句里使用字符表里头的下一个字符O，来避免搜索字符串时添加空白的行为带来的问题。当然，如果有一个镇确实叫做O，我们可能会错误地将它取出，所以你要小心使用这种方法。通常使用z比Z要好，因为z在ASCII表中出现在Z之后。于是，我们的SQL应该像这样：

SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'Nz';

注意我们没有在B之后加上z，因为B后添加空格确实能匹配到B开头的所有城镇，因为它是一个开始点而不是结束点。还要注意如果一个城镇开始于字符Nzz，我们仍将无法找到它，因为我们将用“Nz ”与Nzz比较，然后确定Nzz在Nz之后，因为字符串Nz字符串依旧被填充了一个空白，而空白的位置比我们比较的字符串相对位置的字符z的位置靠前。

这种比较确实有一些很微妙的行为，所以如果你确实要使用BETWEEN比较字符串，需要仔细考虑关于需要匹配的内容。

### 模式匹配

到现在为止的字符串比较操作到现在为止都运行得很好，但他们还不能实现现实世界的字符串模式匹配。SQL条件的模式匹配需要使用LIKE。

不幸的是，LIKE使用与其他我们知道的编程语言不同的字符串匹配规则集。但是，只要你记下了规则，就很容易使用了。当使用LIKE比较字符串，你可以使用百分号（%）表示任何字符串，你还可以使用下划线（\_）表示匹配一个字符。例如，匹配使用字母B开始的城镇，我们可以这么写：

... WHERE town LIKE 'B%'

匹配以字母e结束的名字，我们可以这么写：

... WHERE fname LIKE '%e';

匹配只有四个字符长的名字，我们可以使用四个下划线，像这样：

... WHERE fname LIKE '\_\_\_\_';

我们还可以在一个字符串里头组合这两种类型。

##### 尝试：模式匹配

让我们找找所有的第二个字符为a的客户。以下是能够做到这点的语句：

bpsimple=> SELECT fname, lname FROM customer WHERE fname LIKE '\_a%';

fname | lname

------+--------

Dave | Jones

Laura | Hardy

David | Hudson

(3 rows)

bpsimple=>

##### 解析

这个模式的第一部分“\_a”匹配一个任意字符开始切第二个字符为小写a的字符串。模式的第二部分“%”匹配任何剩下的字符。如果我们不适用尾部的%，就只有两个字符长的字符串被匹配到。

### 限制结果集

在到现在为止我们使用的例子中，返回结果的行数都比较小，因为只有很少的几行示例数据在我们的数据库中。在现实世界的数据库中，我们可以很容易找到数千行符合我们查询标准的数据。如果我们正在写我们的SQL，优化我们的语句，我们当然不想要几千行记录滚过我们的屏幕。只要一些用于检查我们逻辑的示例行就足够了。

PostgreSQL有一个非SQL标准的SELECT语句的扩展从句，LIMIT，可以在我们想限制返回的行数时给我们带来很大的帮助。

如果你在你的SELECT语句后面添加一个带数字的LIMIT从句，从第一行开始，只有最多你指定的行数的行将被返回。使用LIMIT的另一个稍微不同的方法是在与指定开始位置的OFFSET从句一起使用的时候。

用实际操作比用描述更能说明问题。以下我们只显示了五条符合条件的行：

bpsimple=> SELECT customer\_id, town FROM customer LIMIT 5;

customer\_id | town

-------------+-----------

1 | Hightown

2 | Lowtown

3 | Nicetown

4 | Yuleville

5 | Oahenham

(5 rows)

bpsimple=>

以下跳过了结果的前两行，直接返回之后的五行：

bpsimple=> SELECT customer\_id, town FROM customer LIMIT 5 OFFSET 2;

customer\_id | town

-------------+-----------

3 | Nicetown

4 | Yuleville

5 | Oahenham

6 | Nicetown

7 | Bingham

(5 rows)

bpsimple=>

也可以单独使用OFFSET，就像这样：

bpsimple=> SELECT customer\_id, town FROM customer OFFSET 12;

customer\_id | town

-------------+----------

13 | Oxbridge

14 | Welltown

15 | Milltown

(3 rows)

bpsimple=>

如果你想与其他的SELECT从句一起使用LIMIT，LIMIT从句应该总跟随在普通的SELECT语句之后，只有在你使用OFFSET的时候，在后面跟随OFFSET从句。

## 检查空值（NULL）

到现在为止，我们还没有一种方法检查一个列是否包含一个空值。我们可以检查它是否等于一个字符串或是否不等于一个字符串，这还不足够。

如果你还记得第二章讲到的空值是一个特殊的列的值表示不知道或者无关。。我们需要单独看看怎么检测空值，因为它需要特别的考虑以确保结果是想要的。

假设我们知道在我们的表testtab里头的整数列tryint里头存储了0,1或者NULL，我们可以通过以下语句检测它是不是0：

SELECT \* FROM testtab WHERE tryint = 0;

我们也可以通过以下语句检测它是不是1：

SELECT \* FROM testtab WHERE tryint = 1;

我们需要另一种方法检查这个值是不是NULL。PostgreSQL支持标准的SQL语法来检测一个值是否为NULL。我们使用IS NULL做到这点，像这样：

SELECT \* FROM testtab WHERE tryint IS NULL;

注意我们使用关键字IS，而不是=符号。

我们还可通过添加一个NOT来反转测试结果来测试这个值是不是NULL以外的值：

SELECT \* FROM testtab WHERE tryint IS NOT NULL;

为什么我们突然需要这个额外的语法？你可能很熟悉双值逻辑值，也就是一件事情要么是真要么是假。这里我们碰巧碰到了一个三值逻辑值，真、假以及未知。

不幸的是，NULL的未知这个属性，除了我们当前的可以直接检查是否为NULL外，还有一些其他的效果

假设我们在一些tryint列的值为NULL的表中运行下面的语句：

SELECT \* FROM testtab WHERE tryint = 1;

当tryint实际上为NULL时，tryint = 1意味着什么？我们可以问一个问题：“未知是否等于1？”这很有趣，我们不能确定语句的结果是假，也不能确定语句的结果是真。因此结果也是未知的，这样导致NULL不匹配这条语句。如果我们反转测试，比较tryint != 1，为NULL的行也无法被列出来，因为这个表达式也无法为真。这看上去难以理解，因为表面上我们使用两个相反的条件完成了两个测试，但还是无法从表中检索到那行数据。

注意NULL的这种特点很重要，因为太容易忘记NULL值。如果你在针对允许有NULL值的列使用条件得到了意外的结果集，很可能是有些行因为NULL值而引起了你的问题。

## 检查时间和日期

PostgreSQL有两种基本类型用来处理日期和时间信息：保存完整日期和时间信息的timestamp类型和保存年月日信息的date类型。PostgreSQL有一些内建的功能来帮助我们使用日期和时间，而且它们通常都是非常难以操作的。

这里，我们将集中讨论那些我们常用的功能（你可以通过在线文档找到所有的内置功能）。

在我们开始前，我们先要解决一个表面上没用但很容易引起混乱的问题：我们怎么指定一个日期？

当我们写下1/2/2005这样一个日期，我们的意思是什么？对于欧洲人，这意味着这是2005年的二月一日，但是对于美国佬，这意味着2005年一月二日。这是因为欧洲人按照DD/MM/YYYY的格式读取日期而美国人按照MM/DD/YYYY格式。ISO8601（被欧洲标准EN28601官方采用）指出的逻辑日期格式为YYYY-MM-DD（但在日常中很少见）又加重了这个混乱。

PostgreSQL为了迎合你的需要可以让你改变日期处理的方式，所以在我们开始检查日期和时间前，看看你怎样才能控制PostgreSQL在这方面的行为是一个明智之举。

### 设置时间和日期的风格

很遗憾，PostgreSQL设置日期和时间处理模式的功能表面上看好像有点奇怪。

有两个东西你可以控制：

* 日和月处理的顺序，有美国和欧洲两种样式
* 显示的格式；例如仅仅是数字或者更加文本化的输出

坦率地说，PostgreSQL使用一个变量来处理以上两项设定会导致一些混乱。好消息是PostgreSQL默认使用不容易混淆的ISO-8601样式输出日期和时间，它的标准格式是YYYY-MM-DD hh:mm:ss.ssTZD。它给了你年、月、日、时、分、以及精确到小数点后两位小数的秒，以及一个用于指定本地时间和UTC时间之间分钟或者小时修正的时区标识符。例如，一个完整的日期和时间应该像这样：2005-02-01 05:23:43.23+5，也就是2005年2月1日23凌晨5点23分43.23秒，时区在UTC之前5小时。如果你使用UTC时间，也就是没有时区信息，标准里头告诉你你应该使用一个Z（念做“Zulu”）来标记，虽然它看上去好像可以省略掉那个Z。

对于NN/NN/NNNN形式的输入，PostgreSQL默认认为月份在日期之前（美国样式）。例如2/1/05意味着2月1日。你还可以选择February 1,2005的格式，或者ISO的2005-02-01这样的格式。如果这些都是你需要的，那么你很幸运，你不再必须知道更多关于控制PostgreSQL接收和显示日期的内容，你可以往前跳到关于日期和时间的功能的小节。

默认样式实际上由数据目录里头的postgresql.conf文件控制，文件里头有一行类似于“datestyle = 'iso, mdy'”格式的内容。所以如果你想要，你可以改变全局的默认设置。

如果你需要更多控制处理日期的方法，PostgreSQL也允许，但是有一点点麻烦。令人迷糊的是有两个毫不相干的功能需要控制，而你需要用datestyle变量同时设置它们。可是千万记住，这就是所有要做。在日期被输入或者检索的时候，PostgreSQL用一种完全不依赖于任何用户想要的表现形式的内部格式存储日期。

给psql的命令的语法如下：

SET datestyle TO 'value';

为了设置月份和日期处理的顺序，你需要将datestyle的值设置成US或者European，分别表示月份在前（02/01/1997表示2月1日）和日期在前（01/02/1997表示2月1日）。

为了修改显示格式，你也需要设置datestyle，但是要设置成以下四个值之一：

* ISO，设置成ISO-8601标准，使用-作为分隔符，格式类似于1997-02-01
* SQL，用于传统样式，格式类似于02/01/1997
* Postgres，用于默认的PostgreSQL样式，格式类似于 Sat Feb 01
* Geman，用于德国样式，格式类似于01.02.1997

注：在最新的发布版本，PostgreSQL默认的日期和时间戳样式使用SQL样式。

你可以通过逗号分隔的一对参数设置datestyle。所以，例如我们想指定数据显示的样式为SQL样式且使用欧洲的样式转换月份和日期（日期在月份之前），我们使用如下的设置：

SET datestyle TO 'European, SQL';

我们可以在完整的安装的时候设置日期处理日期或者设置它为绘画的默认值，而不需要在每次会话中设置日期处理方式。如果你在完整安装的时候想设置日期输入的样式，你可以在postmaster主服务进程启动前设置PGDATESTYLE环境变量。在Linux设置这个环境变量，我们可以这样做：

PGDATESTYLE="European, SQL"

export PGDATESTYLE

更好的改变默认日期处理方法的方法是修改配置文件postgresql.conf（在安装目录的数据字目录中），依赖你的选择修改datastyle选项为datestyle = 'European, SQL'或者datestyle = 'iso, mdy'。需要重启服务器以使变动生效（reload方法可否生效？需要测试）。

如果你想要为不同的用户设置各自的日期样式，你应该为调用psql的每个本地用户设置环境变量。本地用户的设置将覆盖你设置的环境变量。

在我们演示她们怎么工作之前，我们顺便先介绍PostgreSQL的特别函数cast，它允许你将一种格式的数据转换成另一种格式。我们在这章对它做简单的介绍，当我们在学习SELECT语句中进行计算的时候，但它的功能远不止我们之前看到的转换整数。虽然PostgreSQL很善于类型纠正，而且你并不经常需要类型转换函数，但它们确实有时候很有用。以下是我们需用来进行日期转换的方法，首先是转换成日期：

cast('string' AS date)

另外是转换一个包含时间的值：

cast('string' AS timestamp)

我们也将用一个小小的技巧来使对这个功能的演示更容易。基本上每次你使用SELECT语句的时候，你都将从一个表取出数据。但是，你可以使用SELECT取根本不在表中的数据，就像这个例子：

bpsimple=> SELECT 'Fred';

?column?

--------------

Fred

(1 row)

bpsimple=>

PostgreSQL警告我们没有选择任何列，但它还是乐意接受没有表名的SELECT语法，并返回我们想要的字符串。

我们可以使用同样的方法来与cast函数一起工作，来看看PostgreSQL怎么处理日期和时间，而不需要建立一个临时表来做试验。

##### 尝试：设置日期格式

我们从环境变量PGDATESTYLE未设置开始，所以你可以看到默认的行为，然后设置日期样式，那么你可以看到事情的进展。我们输入ISO格式的日期，因为它是一种安全的可选方案，之后PostgreSQL将输出同样的格式。这些内容都不会有任何的歧义：

bpsimple=> SELECT cast('2005-02-1' AS date);

?column?

------------

2005-02-01

(1 row)

bpsimple=>

改变样式到US和SQL格式，我们仍然以ISO样式输入日期，这也不会有歧义；仍然是2月1日，但是现在输出看上去更传统，但也可能导致歧义，因为美国使用MM/DD/YYYY样式的输出：

bpsimple=> SET datestyle TO 'US, SQL';

SET VARIABLE

bpsimple=> SELECT cast('2005-02-1' AS date);

?column?

------------

02/01/2005

(1 row)

bpsimple=>

现在是时候顺便问问psql，内部的datestyle变量被设置成什么了：

bpsimple=> SHOW datestyle;

DateStyle

-----------

SQL, MDY

(1 row)

bpsimple=>

在旧版本的PostgreSQL中，输出会更详细，所以你看到的东西可能依赖于你使用的不同版本的PostgreSQL。

现在，来试试更多的格式：

bpsimple=> SET datestyle TO 'European, SQL';

SET

bpsimple=> SELECT cast('2005 02 1' AS date);

date

------------

01/02/2005

(1 row)

bpsimple=>

由于输出设置成European，日期显示格式为DD/MM/YYYY方式了。

然后回到ISO模式的输入输出：

bpsimple=> SET datestyle TO 'European, ISO';

SET

bpsimple=> SELECT cast('2005-02-1' AS date);

date

------------

2005-02-01

(1 row)

bpsimple=>

设置项European不再生效了，因为对于所有的区域，ISO都是一样的。

现在来看看时间。我们使用显示时间的timestamp类型。

bpsimple=> SELECT cast('2005-02-1' AS timestamp);

?column?

------------------------

2005-02-01 00:00:00

(1 row)

bpsimple=>

因为我们没有指定任何小时或分钟，所以它们默认都是0。

让我们试试PostgreSQL样式的输出：

bpsimple=> SET datestyle TO 'European, Postgres';

SET

bpsimple=> SELECT cast('2005-02-1' AS timestamp);

timestamp

--------------------------

Tue 01 Feb 00:00:00 2005

(1 row)

bpsimple=>

这样的输出相当明确且友好。

##### 解析

就像你看到的，我们可以改变日期和时间显示的方式，包括像01/02/2005一样模棱两可的输入字符串都被翻译过来了。

时区信息比日期格式更简单。如果你本地的环境变量TZ或者postgresql.conf文件中的配置项被正确设置，PostgreSQL可以毫不费力地管理好时区。

### 使用日期和时间函数

现在我们已经知道日期如何工作了，我们可以看几个有用的你可能在比较日期的时候需要的函数了：

* date\_part(units required, value to use) 允许你提取出日期的某一部分，例如月份。
* Now很简单地获得当前的日期和时间，它实际上和更标准的“魔法变量”current\_timestamp相等。

假设我们想要从我们的orderinfo表取出在九月下订单的行，我们仅仅需要以以下方式提问：

bpsimple=> SELECT \* FROM orderinfo WHERE date\_part('month',date\_placed)=9;

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping

--------------+-------------+-------------+--------------+----------

3 | 15 | 02-09-2004 | 12-09-2004 | 3.99

4 | 13 | 03-09-2004 | 10-09-2004 | 2.99

(2 rows)

bpsimple=>

PostgreSQL为我们提取出了正确的行。注意日期是按照ISO格式显示的。我们可以提取日期和时间戳的以下部分：

* Year（年）
* Month（月）
* Day（日）
* Hour（小时）
* Minute（分钟）
* Second（秒钟）

我们也可以使用用于数字比较的<>，<=，<，>，>=和=运算符来比较日期，以下是一个示例：

bpsimple=> SELECT \* FROM orderinfo WHERE date\_placed >= cast('2004 07 21' AS date);

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping

--------------+-------------+-------------+--------------+----------

3 | 15 | 02-09-2004 | 12-09-2004 | 3.99

4 | 13 | 03-09-2004 | 10-09-2004 | 2.99

5 | 8 | 21-07-2004 | 24-07-2004 | 0.00

(3 rows)

bpsimple=>

注意我们需要用cast来转换我们的字符串到日期，而且我们使用了明确的ISO样式的日期。

第二个函数now，简单地给我们当前的日期和时间，它非常方便，例如，我们插入客户通过电话下的订单的一行，或者通过Internet的实时操作：

bpsimple=> SELECT now(), current\_timestamp;

now | timestamptz

------------------------------------+------------------------------------

Sat 16 Oct 13:46:05.99938 2004 BST | Sat 16 Oct 13:46:05.99938 2004 BST

(1 row)

bpsimple=>

我们也可以对日期进行简单的计算。例如，计算从下订单到发货所间隔的天数，我们可以使用类似这样的查询：

bpsimple=> SELECT date\_shipped - date\_placed FROM orderinfo;

?column?

---------

4

1

10

7

3

(5 rows)

bpsimple=>

这返回数据库中存储的这两天之间所间隔的天数。

注：可以在在线文档中找到更多关于PostgreSQL处理日期、时间、时区以及相关转换函数的细节。

## 多个表协同工作

到现在为止，你应该已经非常了解怎么从表里选择数据，选取我们要的列和行，以及怎样控制数据的顺序了。我们也理会到怎么执行简单的计算，进行类型转换以及处理特别的日期和时间格式了。

是时候开始来到SQL的一个最重要的功能，实际上，应该是关系数据库最重要功能：自动关联一个表的数据到另一个表的数据。好消息是此功能也是由SELECT语句完成，而且你需要进一步学习的关于SELECT的内容和与一个表操作绝对完全一样。

### 关联两个表

在我们研究通过SQL同时使用多个表之前，让我们先回顾下第二章关于关联表的简述。

你应该还记得我们有一个customer表，用来存储我们的客户信息，还有一个orderinfo表，存储他们下的订单细节信息。这允许我们只需要存储一次客户信息，而无论他们下了多少订单。我们通用他们公用的一块数据，也就是存储在这两个表中的customer\_id连接到这两个表。

如果我们仔细想想这样一个图片，我们可以想象customer表中的一行，它的customer\_id没有关联到orderinfo表中的任何一行，或者通过customer\_id关联到出现相同值的orderinfo表中的一行或者多行，就像图4-2说明的一样。

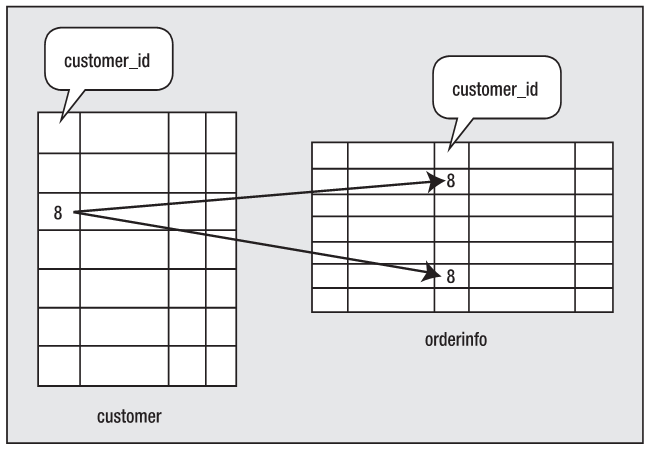


图4-2 customer表和orderinfo表的关系

我们可以说customer表中的customer\_id值为8的行关联到orderinfo表中相同customer\_id出现的两行。当然，我们不需要这两个列有相同的名字，但需要他们存储的内容都是客户编号，只是如果给它们不同的名字，会导致非常难以理解以及不协调。

如果我们需要找到客户Ann Stones的所有订单。逻辑上说，我们先在我们的customer表里头查找这个客户：

bpsimple=> SELECT customer\_id FROM customer WHERE fname = 'Ann'

AND lname = 'Stones';

customer\_id

-------------

8

(1 row)

bpsimple=>

现在，我们知道了customer\_id，我们可以查找这个客户的订单了：

bpsimple=> SELECT \* FROM orderinfo WHERE customer\_id = 8;

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping

--------------+-------------+-------------+--------------+----------

2 | 8 | 23-06-2004 | 24-06-2004 | 0.00

5 | 8 | 21-07-2004 | 24-07-2004 | 0.00

(2 rows)

bpsimple=>

这做到了，但是它需要两步，且我们需要在这两步间记住customer\_id。就像我们在第二章揭示的，SQL是一种说明性语言；也就是说，告诉SQL你要达到什么目标，而不需要明确定义怎么得到结果的步骤。我们刚才所做的是按照程序上的方法使用SQL。我们指定了两个不连续的步骤获得我们的结果，获得一个客户的订单。有没有简洁的方法通过一步获得所有结果呢？

实际上，在SQL中我们可以通过指出我们想知道Ann Stones的订单，且这些信息保存在通过customer\_id这个列关联的customer表和orderinfo表而一步做到。

做到这点的这个新一点的SQL语法是对于WHERE从句的一个扩展：

SELECT <列的列表> FROM <表的列表> WHERE <连接条件> AND <行选择条件>

这看上去有点复杂，但它实际上很简单。让我们将第一个例子做得简单点，假设我们知道客户的编号是8，且需要获取订单日期和客户的名字。我们需要指出我们要的列，客户的名字，下订单的日期，这两个表通过customer\_id列关联，并且我们只需要customer\_id为8的行。

你会突然发现我们有一个小问题。我们怎么告诉SQL我们需要使用哪一个customer\_id：是customer表里的那个还是orderinfo表里头的那个？虽然我们正打算说它们（列名）是相同，但不会总是这样，所以我们怎么处理在不同表里头的列名？我们可以简单的扩展语法指出列名：表名.列名。之后我们可以明确地描述我们数据库里头的每个列了。

通常，PostgreSQL非常宽容，如果一个列名只出现在SELECT语句里头的一个表里，我们也不需要明确指出表名。在本例中，我们将使用customer.fname，因为它更容易读，尤其是你在初学SQL的时候，虽然只使用fname也能够做到。因此，我们语句的第一部分需要需要是这样：

SELECT customer.fname, orderinfo.date\_placed FROM customer, orderinfo

这告诉PostgreSQL我们需要用到的列和表。

我们现在需要指出我们的条件。我们有两个不同的条件：customer\_id为8且这两个表是通过customer\_id相关的，或者说是需要连接的。就像我们在之前看到的关于多个条件一样，我们通过AND关键字指出多个条件必须为真来指出我们的要求：

WHERE customer.customer\_id = 8 AND customer.customer\_id = orderinfo.customer\_id;

注意我们需要通过使用“表名.列名”的语法告诉SQL明确的customer\_id，即使实际上检查这两个表中哪个表的customer\_id字段为8都无关紧要，因为我们还会指出它们俩都必须有相同的值。将它们组装在一起，我们需要的语句就是以下这样的：

bpsimple=> SELECT customer.fname, orderinfo.date\_placed

FROM customer, orderinfo

WHERE customer.customer\_id = 8 AND customer.customer\_id = orderinfo.customer\_id;

fname | date\_placed

-------+-------------

Ann | 2004-06-23

Ann | 2004-07-21

(2 rows)

bpsimple=>

这比分两步查询更优雅，不是吗？也许更重要的是，通过在一条单独的语句中指出问题，我们允许PostgreSQL完全优化检索数据的方法。

##### 尝试：关联表

现在我们知道原理了，让我们回到我们原始的问题，假设我们不知道customer\_id，我们要找到Ann Stones的所有订单。

我们仅仅知道一个名字，而不是客户编号；因此，我们的SQL又变得稍微有点复杂了。我们必须通过名字指出客户：

bpsimple=> SELECT customer.fname, orderinfo.date\_placed FROM customer, orderinfo

WHERE customer.fname = 'Ann' AND customer.lname = 'Stones'

AND customer.customer\_id = orderinfo.customer\_id;

fname | date\_placed

-------+-------------

Ann | 2004-06-23

Ann | 2004-07-21

(2 rows)

bpsimple=>

##### 解析

就像我们之前的例子中看到的，我们指出我们要的列（customer.fname，orderinfo.date\_placed），牵涉到的表（customer，orderinfo），选择条件（customer.fname = 'Ann' AND customer.lname = 'Stones'）以及这两个表怎么关系到一起的（customer.customer\_id = orderinfo.customer\_id）。

SQL为我们做了剩下的工作。它不管客户是没有订单、一条订单或者多条订单。SQL精确地执行SQL提出查询，提供它正确的符合条件的行，即使没有行符合条件。

让我们看看另一个示例。假设我们需要列出我们拥有所有的产品及其条码。你应该记得条码存在barcode表里头，而产品项目存在item表里头。这两个表通过各自的item\_id列关联。你可能还记得我们将它们拆分成两个表的原因是因为很多产品实际上有多个条码。

使用我们刚学到的技能来关联两个表，我们知道我们需要指出我们要的列，表名和它们怎么关联或连接到一起。我们还决定通过产品项目的价格排序：

bpsimple=> SELECT description, cost\_price, barcode\_ean FROM item, barcode

WHERE barcode.item\_id = item.item\_id ORDER BY cost\_price;

description | cost\_price | barcode\_ean

---------------+------------+---------------

Toothbrush | 0.75 | 6241234586487

Toothbrush | 0.75 | 9473625532534

Toothbrush | 0.75 | 9473627464543

Linux CD | 1.99 | 6264537836173

Linux CD | 1.99 | 6241527746363

Tissues | 2.11 | 7465743843764

Roman Coin | 2.34 | 4587263646878

Rubic Cube | 7.45 | 6241574635234

Picture Frame | 7.54 | 3453458677628

Fan Small | 9.23 | 6434564564544

Fan Large | 13.36 | 8476736836876

Wood Puzzle | 15.23 | 6241527836173

Speakers | 19.73 | 9879879837489

Speakers | 19.73 | 2239872376872

(14 rows)

bpsimple=>

这看上去很合理，除了有些产品羡慕看上去出现不止一次，而且我们不记得我们仓库里有两个不同的speakers。我们也不记得仓库里头有那么多商品项目了。到底怎么回事？

让我们使用新学到的SQL技巧统计下我们仓库里头有的商品数量：

bpsimple=> SELECT \* FROM item;

PostgreSQL回应的数据，显示11行（更有经验的SQL用户可以使用更高效的“SELECT count(\*) FROM item;”语句，这个函数将在第七章介绍）。

我们仅存有11项商品，但是我们早前的查询却找到14行，我们做错什么了么？

没有，发生这样的问题的原因是因为一些商品相，例如Toothbrush，每个单独的商品可能有很多不同的条码。PostgreSQL简单地为每个商品项针对每个条码重复了它们的信息，所以它列出了所有的条码以及属于它的每项商品。

你可以通过在SELECT语句中加入商品项编号来确认这一点，就像这样：

bpsimple=> SELECT item.item\_id, description, cost\_price, barcode\_ean

FROM item, barcode

WHERE barcode.item\_id = item.item\_id ORDER BY cost\_price;

item\_id | description | cost\_price | barcode\_ean

---------+---------------+------------+---------------

8 | Toothbrush | 0.75 | 6241234586487

8 | Toothbrush | 0.75 | 9473625532534

8 | Toothbrush | 0.75 | 9473627464543

3 | Linux CD | 1.99 | 6264537836173

3 | Linux CD | 1.99 | 6241527746363

4 | Tissues | 2.11 | 7465743843764

9 | Roman Coin | 2.34 | 4587263646878

2 | Rubic Cube | 7.45 | 6241574635234

5 | Picture Frame | 7.54 | 3453458677628

6 | Fan Small | 9.23 | 6434564564544

7 | Fan Large | 13.36 | 8476736836876

1 | Wood Puzzle | 15.23 | 6241527836173

11 | Speakers | 19.73 | 9879879837489

11 | Speakers | 19.73 | 2239872376872

(14 rows)

bpsimple=>

注意我们明确地指出item\_id从哪个表里头来，因为它同时出现在item表和barcode表里头。

现在很明确到底发生什么了。如果通过SELECT语句返回的数据看上去有点古怪，通过在SELECT语句里头添加所有的编号列来检查到底发生了什么通常是个好主意。

### 给表赋予别名

在本章早些时候，我们看到怎么通过使用AS给出更详细的名字改变输出的列名。如果你想要，你也有可能给表别名。这是种特别的方法，也就是你需要给一个表两个名字，但通常它是用于节省输入。你还会发现它通常用于GUI工具，它让SQL生成更容易。

给表以别名，你只需简单地在SQL的FROM从句部分的表名的后面紧跟别名。一旦你完成这个，你可以在剩下的SQL语句中使用别名，而不仅仅是原来的表名。

假设我们有以下简单的SQL：

SELECT lname FROM customer;

就像我们先去说过的，你可以通过在前面添加表名明确地指出列名，就像这样：

SELECT customer.lname FROM customer;

如果我们给表customer别名cu，我们可以用cu代替列的前缀，就像这样：

SELECT cu.lname FROM customer cu;

注意我们在表名后面直接添加了一个cu，然后又用cu作为列的前缀。

如果只牵涉到一个表，给表别名没太多意思。对于多个表，它就开始有点用了。考虑我们早前的查询：

SELECT customer.fname, orderinfo.date\_placed FROM customer, orderinfo WHERE

customer.fname = 'Ann' AND customer.lname = 'Stones' AND customer.customer\_id =

orderinfo.customer\_id;

是用表的别名，我们可以写成这样：

SELECT cu.fname, oi.date\_placed FROM customer cu, orderinfo oi

WHERE cu.fname = 'Ann' AND cu.lname = 'Stones' AND cu.customer\_id = oi.customer\_id;

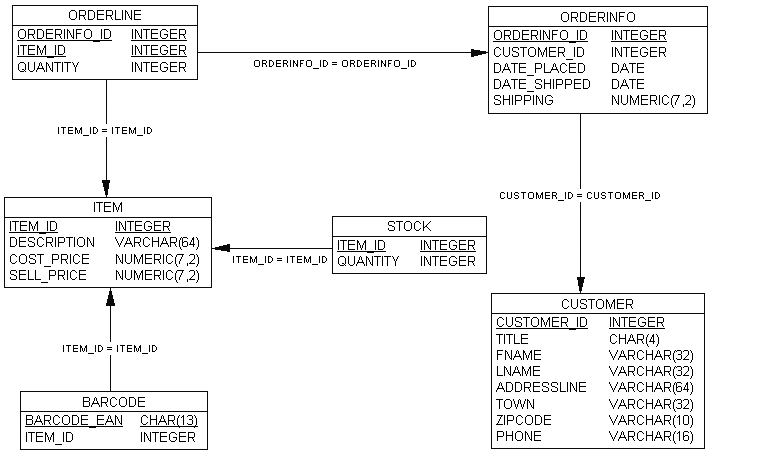
是用别名既可以使SQL清晰，也可以避免在复杂的表中反复输入冗长的表名。

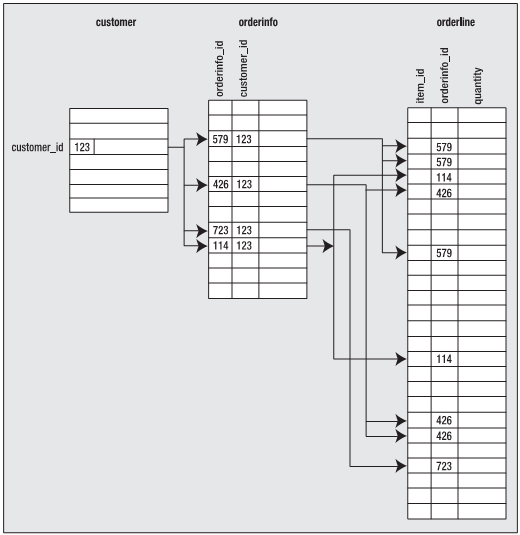
### 关联三个或更多的表

现在我们知道怎么关联两个表到一起了，我们能否将这种想法扩展到三个或者更多的表中？是的，我们可以。SQL是一个充满逻辑的语言，所以如果我们可以针对N项做一些事情，那么我们就可以针对N+1项做相同的事情。当然，你包含的表越多，PostgreSQL需要做的事情就越多，所以有很多表的查询可能非常慢，特别是很多表有大量行的情况下。

假设我们想关联客户信息到他们订的所有商品项的编号？

如果你看看图4-3里我们的数据库模式，你会发现我们需要使用三个表来获得客户订的所有商品项：customer，orderinfo以及orderline。用三个表重画我们早前的分析图，它看上去应该像图4-4一样。

  
图4-3数据库模式

  
图4-4三个相关表

在这里我们可以看到客户123匹配到orderinfo表的很多行——那些匹配到的orderinfo编号为579，426,723和114——它们中的每一行，按顺序关联到orderline表的一行或者很多行。注意customer表和orderline之间没有直接的关联关系。我们必须使用orderinfo表，因为它包含用来绑定客户和他们订单的信息。

##### 尝试：连接多个表

让我们先建立一个三个表的连接用于找寻Ann Stones的订单中的商品项，我们从我们需要的列开始：

SELECT customer.fname, customer.lname, orderinfo.date\_placed, orderline.item\_id,

orderline.quantity

然后我们列出相关的表：

FROM customer, orderinfo, orderline

然后我们指出customer表和orderinfo表是怎么关联的：

WHERE customer.customer\_id = orderinfo.customer\_id

我们还必须指出orderinfo表和orderline表是怎么关联的：

orderinfo.orderinfo\_id = orderline.orderinfo\_id

然后是我们的条件：

customer.fname = 'Ann' AND customer.lname = 'Stones';

将他们放在一起，并分多行输入它们（注意“bpsimple->”等待继续输入提示符），我们 得到这样的：

bpsimple=> SELECT customer.fname, customer.lname, orderinfo.date\_placed,

bpsimple-> orderline.item\_id,orderline.quantity

bpsimple-> FROM customer, orderinfo, orderline

bpsimple-> WHERE

bpsimple-> customer.customer\_id = orderinfo.customer\_id AND

bpsimple-> orderinfo.orderinfo\_id = orderline.orderinfo\_id AND

bpsimple-> customer.fname = 'Ann' AND

bpsimple-> customer.lname = 'Stones';

fname | lname | date\_placed | item\_id | quantity

-------+--------+-------------+---------+----------

Ann | Stones | 2004-06-23 | 1 | 1

Ann | Stones | 2004-06-23 | 4 | 2

Ann | Stones | 2004-06-23 | 7 | 2

Ann | Stones | 2004-06-23 | 10 | 1

Ann | Stones | 2004-07-21 | 1 | 1

Ann | Stones | 2004-07-21 | 3 | 1

(6 rows)

bpsimple=>

注意字符串间的空格不影响SQL，所以我们可以添加额外的空白和换行来使SQL更易读。程序psql只是给出一个等待继续输入的提示符“bpsimple->”，一直等待到我们输入一个分号，它才尝试执行。

你已经看到了从两个表变成三个表是多么容易，那么让我们的查询更进一步，列出我们的客户Ann Stones下的订单中的所有商品项的描述。为了完成这个，我们需要使用一个附加表item，来获得商品项的描述。那么剩下要写的查询也很像之前的：

bpsimple=> SELECT customer.fname, customer.lname, orderinfo.date\_placed,

bpsimple-> item.description, orderline.quantity

bpsimple-> FROM customer, orderinfo, orderline, item

bpsimple-> WHERE

bpsimple-> customer.customer\_id = orderinfo.customer\_id AND

bpsimple-> orderinfo.orderinfo\_id = orderline.orderinfo\_id AND

bpsimple-> orderline.item\_id = item.item\_id AND

bpsimple-> customer.fname = 'Ann' AND

bpsimple-> customer.lname = 'Stones';

fname | lname | date\_placed | description | quantity

-------+--------+-------------+-------------+----------

Ann | Stones | 2004-06-23 | Wood Puzzle | 1

Ann | Stones | 2004-06-23 | Tissues | 2

Ann | Stones | 2004-06-23 | Fan Large | 2

Ann | Stones | 2004-06-23 | Carrier Bag | 1

Ann | Stones | 2004-07-21 | Wood Puzzle | 1

Ann | Stones | 2004-07-21 | Linux CD | 1

(6 rows)

bpsimple=>

##### 解析

你一旦知道了三个表的连接如何工作，就没有困难扩展这个主意到更多的表。我们添加了商品项的描述到需要列出的列的列表中，添加item表到需要查询的表中，还添加关于如何关联item表到我们已有表的表的信息orderline.item\_id = item.item\_id。你将发现Wood Puzzle被列出了两次，因为它在不同的时刻被购买了两次。

在这个SELECT语句中，我们通过连接操作从我们使用的每个表中显示了至少一列。实际上并不需要这么做。如果我们只是需要客户名和商品项的描述，我们可以简单的选择不检索那些我们不需要的列。

一个只检索少量只需要的列的版本可能比早前的尝试的版本稍微高效些且一样够用：

SELECT customer.fname, customer.lname, item.description  
FROM customer, orderinfo, orderline, item  
WHERE customer.customer\_id = orderinfo.customer\_id  
 AND orderinfo.orderinfo\_id = orderline.orderinfo\_id  
 AND orderline.item\_id = item.item\_id  
 AND customer.fname = 'Ann'  
 AND customer.lname = 'Stones';

为了继续这个例子，让我们回到我们在前面章节学到的：怎样通过使用DISTINCT关键字移除重复信息。

##### 尝试：添加额外条件

假设我们想知道Ann Stones买了哪些类型的商品。我们所要列出来的只是他购买的商品项，按照描述排列。我们甚至不需要列出客户名，因为我们已经知道了（我们用它来选择数据）。我们仅仅需要选择item.description，我们还需要使用DISTINCT关键字，来确保Wood Puzzle只被列出来一次，虽然它被购买了好几次：

bpsimple=> SELECT DISTINCT item.description  
bpsimple-> FROM customer, orderinfo, orderline, item  
bpsimple-> WHERE customer.customer\_id = orderinfo.customer\_id  
bpsimple-> AND orderinfo.orderinfo\_id = orderline.orderinfo\_id  
bpsimple-> AND orderline.item\_id = item.item\_id  
bpsimple-> AND customer.fname = 'Ann'  
bpsimple-> AND customer.lname = 'Stones'  
bpsimple-> ORDER BY item.description;  
 description  
-------------  
 Carrier Bag  
 Fan Large  
 Linux CD  
 Tissues  
 Wood Puzzle  
(5 rows)  
  
bpsimple=>

#### 解析

我们只是简单的去处我们早前的SQL，移除我们不再需要的列，然后在SELECT后面添加关键字来确保每行只出现一次，然后在WHERE从句后面加入我们的ORDER BY条件。

这是关于SQL的最伟大的功能中的一个：一旦你学了一个功能，它就可以被通过某种方法被应用于这个功能。ORDER BY可以和跟一个单个表工作一样同时跟很多表工作。

## SQL92的SELECT语法

你可能注意到了WHERE从句实际上由两种稍微不同的任务。它不但指定确定哪一行我们需要检索的条件（customer.fname = 'anna'），而且指出多个表怎么互相关联（customer.customer\_id = orderinfo.customer\_id）。

多年来这不会引起任何问题，知道SQL标准委员会尝试扩展语法来帮助处理SQL被分配的越来越多的工作。当SQL92标准发布，一个新的格式的SELECT语法被添加用于区分这两种稍微有点不同的用法。这种新的语法（有时候，被叫做SQL92/99语法，或者ANSI语法）被很多SQL数据库延迟支持。微软是在其SQL Server 6.5中最早支持这种语法的，PostgreSQL在7.1版本中添加了对这种语法的支持，但是Oracle直到9才支持这种新语法。

新语法使用JOIN … ON语法来指定表之间的关系，而让WHERE从句专注于哪些行需要选择。新的语法将表的关联信息从SELECT语句的WHERE从句移动到FROM部分。所以语法由：

SELECT <列的列表> FROM <表的列表>

WHERE <连接条件> <行选择条件>

转变为：

SELECT <列的列表> FROM <表> JOIN <表> ON <连接条件>

WHERE <行选择条件>

它比看起来更简单——真的！假设我们想连接customer表和orderinfo表，他们共享customer\_id这个键。作为以下语句的替代者：

FROM customer, orderinfo WHERE customer.customer\_id = orderinfo.customer\_id

我们应这样写：

FROM customer JOIN orderinfo ON customer.customer\_id = orderinfo.customer\_id

这看上去有点罗嗦，但是它既清晰又易于扩展，就像我们将在第七章看到外连接的时候会了解到。

扩展到不止两个表也非常简单。对于我们之前的简单查询：

SELECT customer.fname, customer.lname, item.description

FROM customer, orderinfo, orderline, item

WHERE

customer.customer\_id = orderinfo.customer\_id AND

orderinfo.orderinfo\_id = orderline.orderinfo\_id AND

orderline.item\_id = item.item\_id AND

customer.fname = 'Ann' AND

customer.lname = 'Stones';

在SQL92语法中，会变成这样：

SELECT customer.fname, customer.lname, item.description

FROM customer

JOIN orderinfo ON customer.customer\_id = orderinfo.customer\_id

JOIN orderline ON orderinfo.orderinfo\_id = orderline.orderinfo\_id

JOIN item ON orderline.item\_id = item.item\_id

WHERE

customer.fname = 'Ann' AND

customer.lname = 'Stones';

两个版本的SQL语句将产生相同的结果。

但是，很多用户看上去更喜欢较早的语法，它们仍旧有效且对于很多SQL语句看上去更简洁。我们在这里提到更新的SQL92版本，所以你将熟悉这个语法，但是在本书中，我们通常继续使用旧样式的连接，除了在第七章我们碰到外连接的时候。

## 摘要

本章相当的长，但我们也覆盖了大量的内容。我们揭示了SELECT语句的一些细节，涵盖了如何选择列以及行，如果将结果排序，以及怎么隐藏重复的信息。我们还学了一点点关于日期类型，以及怎么配置PostgreSQL关于日期转换和显示的行为，以及怎么在条件语句里头使用日期。

然后我们来到SQL的核心：将表关联到一起的能力。在我们第一步通过SQL连接两个表后，我们发现将这扩展到三个表甚至四个表都是多么的简单。我们结束于重用我们之前章节获取的知识来优化我们的四个表关联查询，显示我们搜索的信息，并移除多余的列和重复的行。

好消息是我们现在已经见到过我们每天都会使用的SELECT语句，并且一旦你理解了SELECT语句，大部分剩下的SQL都是相当简单的了。我们将在第七章回到SELECT语句，来看一些你偶尔会用到的高级功能，但你会发现本章已经覆盖绝大部分现实世界中你需要使用的SQL了。

# PostgreSQL的命令行和图形界面工具

在先前的章节，为了开始学习，我们已经通过命令行工具psql建立并进行管理了一个PostgreSQL数据库。商用数据库通常都有一个和psql类似的命令行工具。例如，Oracle的这种工具叫做SQL\*Plus。

因为命令行工具通常比较完全，也就是说它们包含执行所有你需要功能的方法，但他们在用户友好度上有点不够。在另一方面，它们对图形显卡的要求不高，对内存等也一样。

在本章，我们将从仔细研究psql开始。然后，我们将覆盖怎么为PostgreSQL数据库设置一个ODBC数据源，这对一些在本章讲到的工具非常重要。然后我们会碰到一些用于PostgreSQL数据库的图形界面工具。其中一些工具也可以用于数据库管理，这是第十一章的主题。在本章，我们将专注于通常的数据库任务。

特别是，在本章我们将测试以下工具：

* psql
* ODBC
* pgAdmin III
* phpPgAdmin
* Microsoft Access
* Microsoft Excel

## psql

工具psql允许我们连接到数据库，执行查询以及管理数据库，包括使用SQL命令建立数据库，天剑新表以及输入和更新数据。

### 启动psql

就像我们看到的，我们通过指定我们需要连接的数据库启动psql。我们需要知道服务器的主机名和数据库监听的端口（如果它不是运行在默认的5432端口），外加有效的用户名和密码来进行连接。默认的数据库将是与本地主机当前登录的用户名相同的数据库。

如果要连接到服务器中其他名字的数据库，我们在调用psql的时候需要带上数据库名，就像这样：

$ psql -d bpsimple

我们可以通过设置环境变量PGDATABASE、PGUSER、PGHOST以及PGPORT分别覆盖默认的数据库名、用户名、主机名和监听端口。这些默认参数也可以通过psql的命令行参数-d、-U、-h和-p重新赋值。

注：我们可以通过连接到一个数据库仅仅启动psql。这在建立我们第一个数据库时产生了“鸡和蛋”的问题。我们需要一个用户和一个数据库用于连接。我们在第三章安装PostgreSQL的时候建立了一个默认用postgres，所以我们可以使用它连接并建立新用户和数据库。为了建立一个数据库，我们需要连接到一个随PostgreSQL安装而生成的特殊数据库template1。一旦连接到template1数据库，我们可以建立一个数据库，然后退出并重启psql或者使用\c命令重新连接到新数据库。

当psql启动后，如果在当前用户的家目录中存在它的启动文件.psqlrc且可读，psql将读取它。这个文件很类似于shell脚本的启动文件，并且包含很多psql的命令用于设置想要的行为，例如设置打印表格的格式选项以及其他选项。我们可以通过启动psql时加入-X选项阻止psql读取这个文件。

### 在psql中输入命令

一旦运行，psql将通过包含我们当前连接到的数据库名并跟随“=>”符号的提示符提示输入命令。对于在当前数据库拥有全部权限的用户，提示符被替换为“=#”。

psql的命令分为两种不同的类型：

* SQL命令：我们可以输入任何PostgreSQL支持的SQL语句给psql，然后它将执行它。
* 内部命令：有一些psql命令用于执行SQL不直接执行的命令例如列出存在的表和执行脚本。所有的内部命令都由一条反斜杠开始切不能被拆分成多行。

小提示：你可以通过执行内部命令\h查询一个全部支持的SQL命令的列表。可以使用\h <SQL命令>获得对某个特别命令的帮助。内部命令\?可以列出所有的内部命令。

输入到psql的SQL命令可以被分散成多行。当这种情况发生后，psql将转换它的提示符为->或者-#提示还需要更多的输入，就像这个例子：

$ psql -d bpsimple

...

bpsimple=# SELECT \*

bpsimple-# FROM customer

bpsimple-# ;

...

$

我们需要通过一个分号来告诉psql我们已经完成一条可能被拆分成多行的长SQL命令。注意分号不是一条SQL命令必须的部分，它只是用于告诉psql我们已经完成输入了。例如在上面列出的SELECT语句中，我们可能还需要在下一行添加一条WHERE从句。

我们也可以通过在启动psql时打开-S选项告诉psql我们将不会拆分我们的命令到多行。在这种情况下，我们不需要在我们命令的末尾添加分号。psql的提示将变为“^>”以提示我们处在单行模式。这将为我们节省少量输入并对执行一些SQL脚本有用。

### 使用命令历史

在支持历史记录的PostgreSQL平台，我们请求psql执行的每条命令被记录为历史，我们可以找回之前的命令用于运行或者编辑。使用方向键滚动命令历史并编辑命令。这个功能在你没有使用-n命令行选项关闭前都有效（或者没有针对你的平台编译这个功能）。

我们可以通过“\s”命令查看历史，或者通过“\s <文件名>”保存命令历史到文件。最后执行的查询被保存在查询缓冲区中。我们可以通过“\p”命令查看什么被保存在查询缓冲区并通过“\r”命令清除它。我们可以通过“\e”命令使用外部编辑器编辑它。默认的编辑器是vi（在Linux和UNIX中），但你可以在启动psql前设置EDITOR环境变量指定你自己喜欢的编辑器。我们可以简单地使用“\g”命令发送查询缓冲区的内容到服务器用以简单的重新执行一条查询。

### 在psql中执行脚本文件

我们可以收集一组psql命令（包括SQL和内部命令）到一个文件并把它当做一个简单的脚本使用。内部命令“\i”将从一个文件中读取一组psql命令。

这个命令对于建立和填充表格非常有用。我们在第三章我们使用它建立了我们简单的数据库bpsimple。以下是我们使用的脚本文件create\_tables-bpsimple.sql的一部分：

CREATE TABLE customer

(

customer\_id serial ,

title char(4) ,

fname varchar(32) ,

lname varchar(32) NOT NULL,

addressline varchar(64) ,

town varchar(64) ,

zipcode char(10) NOT NULL,

phone varchar(16) ,

CONSTRAINT customer\_pk PRIMARY KEY(customer\_id)

);

CREATE TABLE item

(

item\_id serial ,

description varchar(64) NOT NULL,

cost\_price numeric(7,2) ,

sell\_price numeric(7,2) ,

CONSTRAINT item\_pk PRIMARY KEY(item\_id)

);

我们依照惯例给脚本文件一个.sql的扩展名，并使用“\i”内部命令执行它们：

bpsimple=#\i create\_tables-bpsimple.sql

CREATE TABLE

CREATE TABLE

...

bpsimple=#

在本例中，脚本文件存放于我们启动psql的目录，但我们可以通过提供完整路径执行任何地方的脚本。

脚本的另一个用途是做简单的报表。如果我们想要注意数据库的增长，我们可以将一些命令写入一个脚本里面并安排它每隔一段时间运行一次。为了汇报客户数和订单数，我们可以建立包含以下几行命令的脚本文件report.sql并在一个psql会话中执行它：

SELECT count(\*) FROM customer;

SELECT count(\*) FROM orderinfo;

我们也可以选择使用“-f”命令行选项让psql命令执行脚本文件并退出：

$ psql -f report.sql bpsimple

count

--------

15

(1 row)

count

--------

5

(1 row)

$

如果访问数据库需要密码，psql将提示输入一个。我们可以通过“-U”选项指定一个不同的数据库用户给psql。

我们可以通过-o命令行选项直接要求将输出定向到一个文件，或者通过“\o”内部命令将当前会话的内容定向到一个文件或者过滤程序。例如，在一个psql会话中，我们可以通过输入以下命令建立一个包含我们所有客户的名叫customer.txt的文本文件：

bpsimple=# \o customers.txt

bpsimple=# SELECT \* FROM customer;

bpsimple=# \o

组后的不带文件名参数的“\o”命令停止重定向查询输出并关闭输出文件。

### 检查数据库

我们可以通过使用一些psql的内部命令浏览我们数据库的结构。这个结构包括构成数据库的表的名字和定义，可能已经建立的一些函数（存储过程和触发器），已经建立的用户等等。

“\d”命令列出了所有的关系——表、序列生成器和视图（如果在我们数据库里头有）。以下是一个示例：

bpsimple=# \d customer

Table "public.customer"

Column | Type | Modifiers

-------------+-----------------------+-----------------------------------

customer\_id | integer | not null default nextval(...)

title | character(4) |

fname | character varying(32) |

lname | character varying(32) | not null

addressline | character varying(64) |

town | character varying(32) |

zipcode | character(10) | not null

phone | character varying(16) |

Indexes:

"customer\_pk" PRIMARY KEY, btree (customer\_id)

bpsimple=#

“\dt”命令约束于只列出表。查看表5-2“内部命令快速参考”小节了解更多psql内部命令。

### psql命令行快速参考

psql的命令语法是：

psql [options] [dbname [username]]

psql命令行选项以及它们的意思在表5-1中列出。使用以下命令可以看到psql完整的选项列表：

$ psql --help

表5-1 psql命令行选项

| 选项 | 意义 |
| --- | --- |
| -a | 从脚本中响应所有输入 |
| -A | 取消表数据输出的对齐模式；功能与“-P format=unaligned”相同 |
| -c <查询> | 仅仅运行一个简单的查询（或者内部命令）然后退出 |
| -d <数据库名> | 指定连接到的数据库名（默认为$PGDATABASE或者当前登录用户名） |
| -e | 回显发送到服务器的查询 |
| -E | 显示内部命令生成的查询语句 |
| -f <文件名> | 执行一个文件中的查询，然后退出 |
| -F <字符串> | 指定列数据显示分隔符（默认为“|”）；功能与“-P fieldsep=<字符串>”相同 |
| -h <主机> | 指定数据库服务器主机（默认为$PGHOST或者本地主机） |
| -H | 设置表格输出模式为HTML；功能与“-P format=html”相同 |
| --help | 显示帮助，然后退出 |
| -l | 列出存在的数据库，然后退出 |
| -n | 禁用readline；阻止行编辑 |
| -o <文件名> | 将查询的输出发送到文件名指定文件（使用“|管道”的形式将输出发送到一个过滤程序） |
| -p <端口> | 指定数据库服务器的端口（默认为$PGPORT或者编译期设置的默认值，通常为4321） |
| -P var[=arg] | 设置打印选项var为arg（查看\pset命令） |
| -q | 以静默方式运行（没有任何消息，仅有查询的输出） |
| -R <字符串> | 设置记录的分隔符（默认为换行）；功能与“-P recordsep=<字符串>”相同 |
| -s | 设置为单步执行模式（每条查询都需要确认） |
| -S | 设置单行模式（每行结束就认为查询输入结束，而不是分号） |
| -t | 只打印行；功能与“-P tuples\_only”相同 |
| -T <文本> | 设置HTML表格标记选项（width，border等）；功能与“-P tableattr=<text>”相同 |
| -U <用户名> | 指定数据库用户（默认为$PGUSER或者当前登录的用户名） |
| -v name=value | 设置psql变量name的值为value |
| --version | 显示版本信息然后退出，也可以用“-V” |
| -W | 强制提示输入密码（如果需要密码，会自动执行） |
| -x | 开启扩展表格输出；功能与“-P expanded”相同 |
| -X | 不读取启动文件（~/.psqlrc） |

### psql内部命令快速参考

psql支持的内部命令在表5-2中列出。在很多版本的PostgreSQL中，这些命令有一些更易读的长模式（例如\list就是\l的长模式命令）。

表5-1 psql命令行选项

| 命令 | 意义 |
| --- | --- |
| \? | 列出所有的psql内部命令 |
| \a | 在表格对齐和非对齐模式之间切换。 |
| \c[onnect] [dbname|- [user]] | 连接到新的数据库；使用“-”作为数据库名指连接到默认数据库。可以user身份连接数据库 |
| \C <标题> | 设置输出表格的标题；功能与“\pset 标题”相同 |
| \cd <目录> | 改变工作目录 |
| \copy … | Perform SQL COPY with data stream to the client machine. |
| \copyright | 显示PostgreSQL的使用和发布条款 |
| \d <表> | 描述表（或者视图、索引、序列生成器） |
| \d{t|i|s|v} | 列出表/索引/序列生成器/视图 |
| \d{p|S|l} | 列出访问许可/系统表/大对象 |
| \da | 列出聚合体（aggregates） |
| \db | 列出表空间 |
| \dc | 列出conversions |
| \dC | 列出casts |
| \dd [对象] | 列出表、类型、函数或者操作的注释 |
| \dD | 列出domains |
| \df | 列出函数（自定义函数？？？）需要验证 |
| \dg | 列出groups |
| \dl | 列出大对象；也可以写作“\lo\_list” |
| \dn | 列出模式 |
| \do | 列出operators |
| \dT | 列出数据类型 |
| \du | 列出用户 |
| \e [file] | 使用外部编辑器编辑当前的查询缓冲区或者file指定的文件 |
| \echo <文本> | 将文本打印到标准输出 |
| \encoding <编码> | 设置客户端编码 |
| \f <分隔符> | 修改输出字段的分隔符 |
| \g [文件名] | 将查询的结果发送到后端（结果输出到文件或者|管道） |
| \h [命令] | 显示SQL命令的帮助；\*表示所有命令的详细说明 |
| \H | 开启HTML模式 |
| \i <文件名> | 从文件中读取并执行查询 |
| \l | 列出所有的数据库 |
| \lo\_export，\lo\_import，  \lo\_list，\lo\_unlink | 执行大对象操作 |
| \o [文件名] | 将所有的查询结果发送到文件或者|管道 |
| \p | 显示当前查询缓冲区的内容 |
| \pset <选项> | 设置表输出选项，可设置的选项可以是以下中的一个：format，border，expanded，fieldsep，footer，null，recordsep，tuples\_only，title，tableattr，pager |
| \q | 退出psql |
| \qecho <文本> | 将文本写入到查询输出流（参考\o命令） |
| \r | 重置（清空）查询缓冲区 |
| \s [文件名] | 打印历史或将历史存入文件中 |
| \set <变量> <值> | 设置内部变量 |
| \t | 只显示行（在该模式之间切换） |
| \T <标记> | 设置HTML表格的标记；功能和“\pset tableattr”相同 |
| \timing | 显示命令执行的时间（在显示和不显示这两种模式间切换） |
| \z | 列出对表、视图和序列生成器的访问许可 |
| \! [命令] | 切换到shell或者执行一个shell命令 |

## 设置ODBC

本章讨论的很多工具以及后面章节讨论的一些编程语言接口使用ODBC标准接口连接到PostgreSQL。ODBC定义了一个基于X/Open和ISO/IEC编程接口的通用的数据库接口。实际上，用于开放数据库互联的ODBC标准不仅仅限于微软的Windows客户端（虽然很多人经常这么认为）。很多语言包括C，C++，Ada，PH{，Perl以及Python编写的程序都可以使用ODBC。OpenOffice， Gnumeric，Microsoft Access和Microsoft Excel仅仅是可以使用ODBC的少量示例程。

为了在一个特定的客户机上使用ODBC，我们同时需要针对ODBC接口写的应用程序以及我们想要使用的特定数据库的驱动程序。PostgreSQL的ODBC驱动名叫psqlodbc，我们可以将它安装到我们的客户机上。通常，客户机运行在与服务器不同的主机上，甚至它们之间可能都不相同，所以我们需要针对不同的客户端平台编译不同的ODBC。例如，我们的数据库服务器可能在Linux上但是我们的客户端应用程序在Windows和Mac OS X上运行。

源码和Windows下的二进制安装程序可以在psqlODBC项目的首页http://gborg.postgresql.org/project/psqlodbc/中找到。

注：PostgreSQL的标准Windows安装程序也包含以版本的ODBC驱动程序，可以在安装数据库的同时安装到Windows服务器上。

### 在Windows中安装ODBC驱动程序

在微软的Windows中，ODBC驱动可以通过控制面板的管理工具中的数据源（ODBC）工具配置可见，如图5-1所示：

  
图5-1 ODBC数据源工具

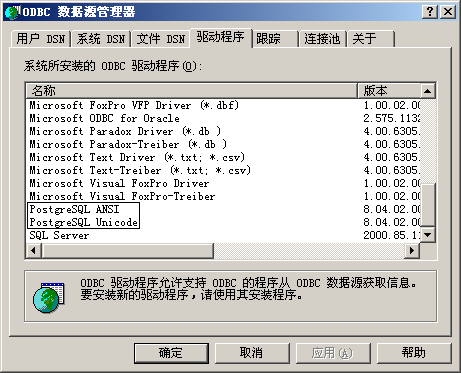
该程序的驱动程序页列出已安装的ODBC驱动，如图5-2所示：

  
图5-2已安装的ODBC驱动程序

我们需要执行两步来安装PostgreSQL驱动程序：

1. 从http://gborg.postgresql.org/project/psqlodbc下载适当版本的驱动程序。如果你的Windows已经包含了Microsoft Windows Installer，那么推荐你选择MSI版本的驱动程序，因为它小很多，否则，需要下载完整的安装包。两个版本的驱动程序都存放在压缩文件psqlodbc-07\_03\_0200.zip中。
2. 从下载的压缩包里解压安装程序文件，它可能是psqlodbc.msi或者psqlodbc.exe。双击安装程序并跟随指引安装PostgreSQL驱动程序。

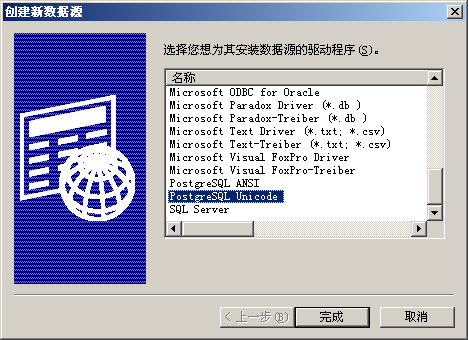
在这些这两步后，我们可以通过再次选择ODBC数据源管理器的驱动程序页查看PostgreSQL驱动是否出现在列表中来确定我们成功安装了这个驱动，就像图5-3所示：

  
图5-3 PostgreSQL驱动程序已经安装

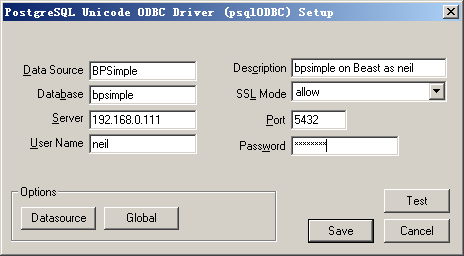
### 在Windows中建立一个数据源

现在我们可以使用面向ODBC的应用程序连接到PostgreSQL数据库了。为了让一个指定的数据库连接存在，我们需要通过以下的步骤建立一个数据源：

1. 选择ODBC数据源管理器的用户DSN页面建立一个当前用户的数据源（如果你选择系统DSN，你可以建立一个所有用户都能看到的数据源）。
2. 点击添加按钮来启动建立过程。一个用于选择数据源将使用哪个驱动程序的对话框将出现，如图5-4所示。

  
图5-4建立一个PostgreSQL数据源

1. 选择PostgreSQL驱动并点击完成。
2. 我们现在拥有了一个PostgreSQL驱动项目等待配置。一个驱动设置窗口将出现，用于让我们输入这个数据源的细节。如图5-5所示，填写数据源的名字和描述，设置网络连接。在这里，我们将通过服务器的IP地址（如果你运行了一个完全配置的命名服务器例如DNS或者WINS，你可以使用服务器的机器名）建立一个ODBC连接到我们运行在一台Linux服务器上的bpsimple数据库。我们还需要指定服务器上的用户名和密码用于访问我们使用的数据库。我们还需要选择适当的SSL数据链路模式。

  
图5-5 配置一个PostgreSQL数据源

提示：在ODBC驱动程序设置的对话框中有Datasource和Global附加选项。如果你想通过ODBC程序更新或者插入数据到PostgreSQL数据库中，你需要配置数据源来支持这个功能。要做到这点，点击Datasource按钮，打开高级选项对话框，确保对话框中的Read Only选项没有被选中。

1. 点击保存来完成设置。

我们已经准备好通过ODBC程序例如Microsoft Access和Excell访问我们的PostgreSQL数据库了，就像我们将在本章晚些时候讨论的一样。下一步，我们将看到一些开源的备选方案，我们从pgAdmin III开始。

### 在Linux/Unix中安装ODBC驱动程序

### 在Linux/Unix中建立一个数据源

需要自我完善

## pgAdmin III

pgAdmin是一个全功能的PostgreSQL数据库图形界面接口。它是自由软件，由社区在http://www.pgadmin.org管理。从网站上看，pgAdmin是“一个可以自由使用的强大的PostgreSQL数据库管理和开发平台”。它可以运行在Linux，FreeBSD和Windows 2000/XP/Vista/7中。运行在Mac OS X的版本正在开发中。

pgAdmin提供多种功能。通过它，我们可以做到以下的事情：

* 建立和删除表空间、数据库、表和模式
* 通过一个查询窗口执行SQL
* 将SQL查询的结果输出到文件
* 备份和恢复数据库或者单独的表
* 配置用户、组和权限
* 查看、编辑和插入表的数据

让我们看看怎么准备和运行这样一个多功能的工具吧。

### 安装pgAdmin III

随着pgAdmin III的发布，开发人员将这个程序的安装变得非常简单。前一个版本需要安装PostgreSQL的ODBC驱动程序来提供对数据库的访问，但是这种依赖已经被移除了。如果你还在使用旧版本的pgAdmin，我们强烈推荐你进行升级。

注PostgreSQL的标准Windows安装程序包含一个版本的pgAdmin III，它既可以与数据库一同安装在Windows服务器上，也可以在客户机上单独安装。

用于Windows2000/XP，FreeBSD，Debian Linux，Slackware Linux和其他支持RPM的Linux（例如Red Hat和SuSE Linux）的pgAdmin二进制安装包可以从<http://www.pgadmin.org/pgadmin3/download.php>处下载。

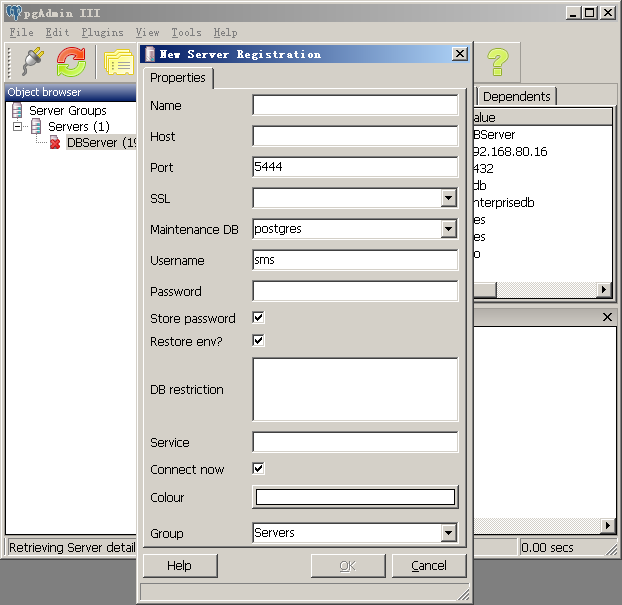
下载你需要运行pgAdmin III的系统的对应安装包并安装它。Windows安装包包含一个可执行安装程序，压缩在一个ZIP格式的压缩包内。安装后，你应该在Windows的开始菜单有一个新的pgAdmin III菜单项。

### 使用pgAdmin III

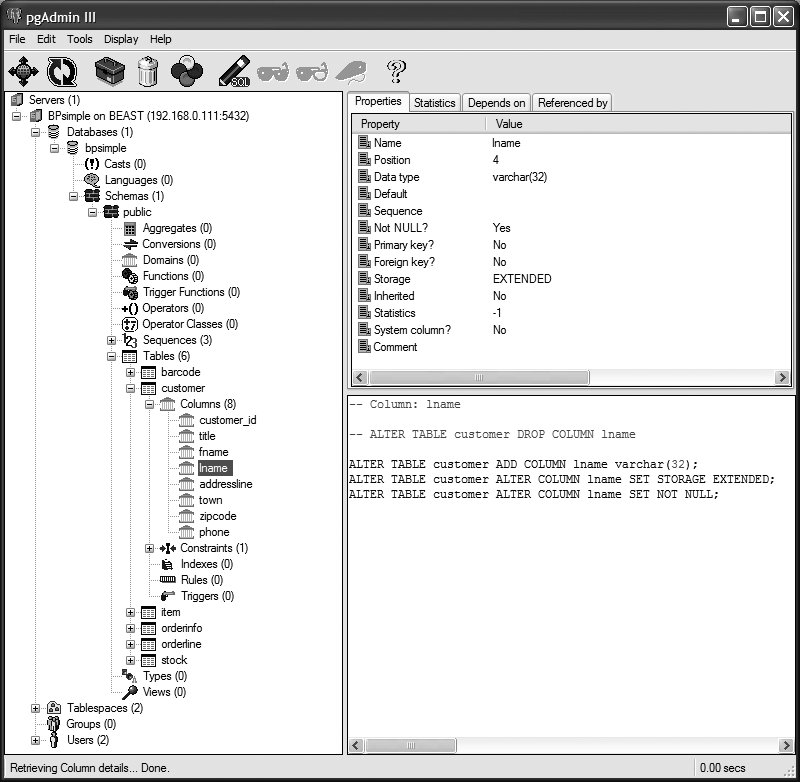
在我们认真地使用pgAdmin III之前，我们需要确保我们可以在我们需要管理的数据库中建立对象。这是因为pgAdmin III需要通过存储它自己的一些对象到服务器上扩充数据库。要使用pgAdmin III执行所有的管理功能，我们需要以一个拥有数据库全部权限的用户登录——也就是说，需要一个超级用户（superuser）。如果我们选择的用户没有超级用户权限，我们将得到一个错误信息。

提示：我们将在第十一章讨论用户和权限。如果你的PostgreSQL数据库是在Windows上按照默认设置安装的，你应该有一个叫做postgres的用户用于控制数据库，你可以尝试使用这个用户登录。如果你是跟随第三章的步骤在Linux或者UNIX上安装的，你将已经建立了一个合适的用户，在这里我们使用的用户名是neil。

我们可以通过pgAdmin III同时管理很多数据库，所以我们的第一步是建立一个服务器连接。从文件菜单选择添加服务器将弹出一个非常类似于早前建立ODBC连接的对话框。图5-6显示了一个到Linux服务器上的PostgreSQL数据库的连接。

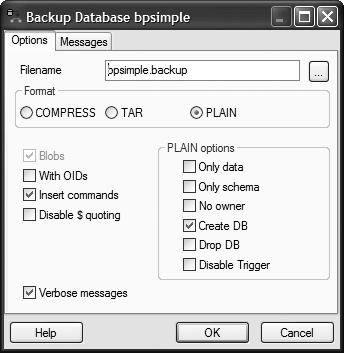
  
图5-6添加一个服务器连接到pgAdmin III中

一旦服务器连接建立完成，我们可以连接到数据库服务器并浏览数据库，表以及其他服务器提供的对象。图5-7显示了一个pgAdmin III浏览bpsimple数据库的表以及检查customer表的lname的示例。

  
图5-7使用pgAdmin III检查表的属性

pgAdmin III的一个可能非常有用的功能是它的备份和恢复功能。它提供一个到PostgreSQL的pg\_dump工具的简单接口，我们将在第十一章详细讲解pg\_dump工具。我们可以备份和恢复单独的表或者整个数据库。有一些选项控制怎么建立备份文件以及存放在哪里，以及如果必要我们将使用哪种方法恢复数据库（例如，可以使用\copy命令或者SQL的INSERT语句）。

要打开备份对话框，需要在需要备份的对象（数据库或者表）上右击并选择备份。图5-8显示了用于备份bpsimple数据库的备份对话框。

  
图5-8 pgAdmin III的备份对话框

我们将在在第十一章详细覆盖pgAdmin用于管理数据库的功能。

使用pgAdmin III的查询建立器，需要自己完善

好

## phpPgAdmin

一个基于web的用于管理PostgreSQL数据库的替代品是phpPgAdmin。这是一个应用安装在一个web服务器的应用程序（使用PHP编程语言），提供基于浏览器的接口用于管理数据库服务器。这个项目的主页位于<http://phppgadmin.sourceforge.net>。

通过phpPgAdmin，我们可以对我们的数据库执行很多任务，包括：

* 管理用户和族
* 建立表空间，数据库和模式
* 管理表、索引、约束、触发器、规则和权限
* 建立视图、序列生成器和函数
* 建立和执行报表
* 浏览表中的数据
* 执行独立的SQL
* 以很多格式导出表的数据：SQL，COPY（适合使用SQL COPY命令的数据），XML，XHTML，逗号分隔值的文件（CSV，comma-separated values），tab分隔值的文件以及pg\_dump
* 导入SQL脚本，拷贝数据、XML文件、CSV文件以及tab分隔值的文件

### 安装phpPgAdmin

安装phpPgAdmin非常简单。程序可以以多种格式的压缩包格式下载，包括ZIP和压缩的tar包（.tar.gz）。程序包需要被解压到由一个支持PHP编程语言的web服务器提供服务的目录中。一个常见的选择是配置了mod\_php扩展的Apache网页服务器。更多关于Apache和PHP的信息可以分别在<http://www.apache.org>和<http://www.php.net> 找到。很多Linux发行版提供了已经适当配置的Apache安装包。

phpPgAdmin唯一需要的配置是在它的配置文件conf/conf.inc.php里头设置一些变量。以下摘录了这个文件中需要配置用于设置phpPgAdmin管理在另一台数据库上的数据库的行。

// Display name for the server on the login screen

$conf['servers'][0]['desc'] = 'Beast';

// Hostname or IP address for server. Use '' for UNIX domain socket.

$conf['servers'][0]['host'] = '192.168.0.111';

// Database port on server (5432 is the PostgreSQL default)

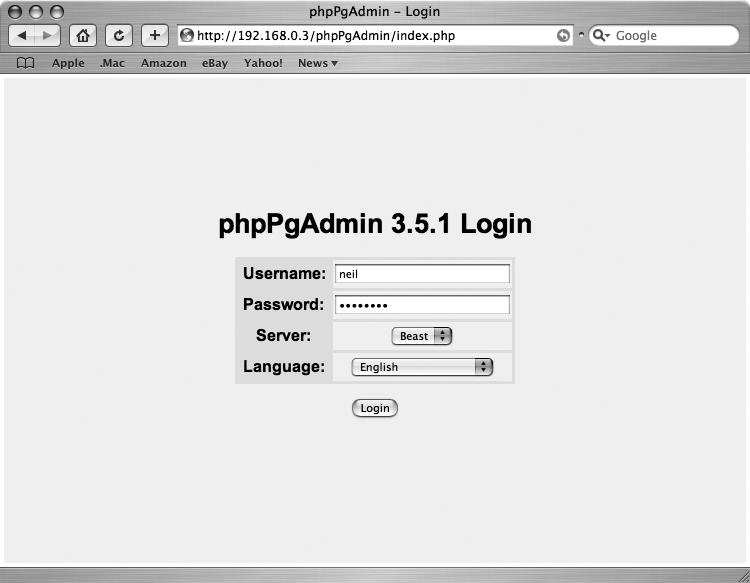
$conf['servers'][0]['port'] = 5432;

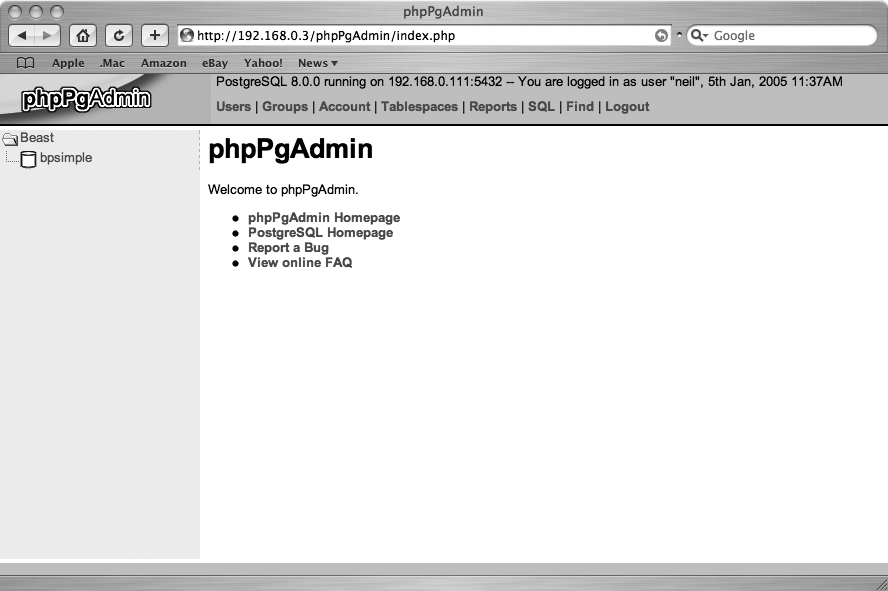
// Change the default database only if you cannot connect to template1

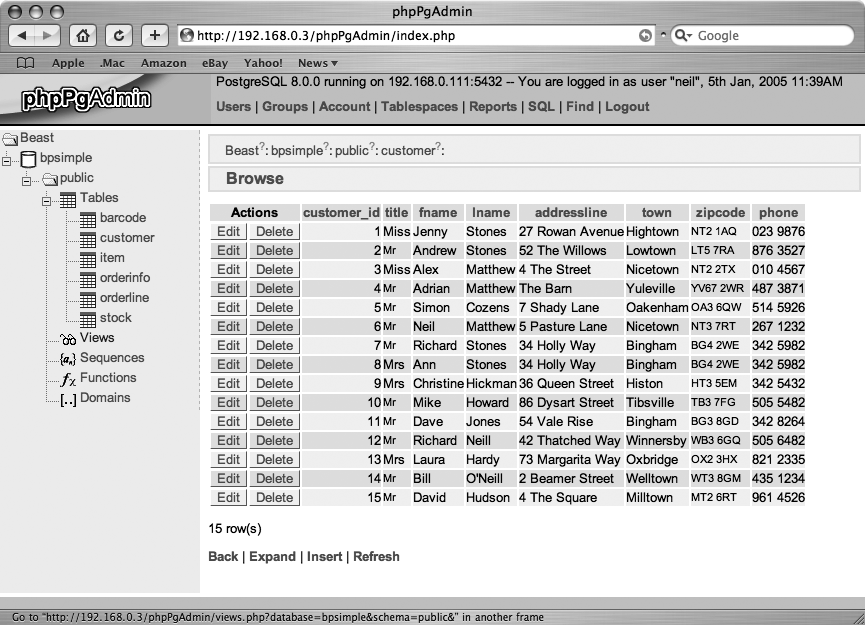
$conf['servers'][0]['defaultdb'] = 'template1';

### 使用phpPgAdmin

为了掩饰phpPgAdmin，Apache和PostgreSQL的跨平台能力，图5-9和图5-10显示一个运行在Apple Mac机器上的浏览器访问一个运行在Windows XP（位于地址192.168.0.3）上带有phpPgAdmin的Apache网页服务器，管理一台叫做Beast的位于192.168.0.111的Linux服务器上的数据库。图5-11展示了了customer表中的数据被显示的情况。浏览器的URL为<http://192.168.0.3/phpPgAdmin/index.php>。

  
图5-9 phpPgAdmin的登录界面

  
图5-10 phpPgAdmin的主界面

  
图5-11通过phpPgAdmin浏览表的数据

phpPgAdmin的一个可能非常有用的功能是它的数据输入功能。如果我们有一些数据想导入到一个PostgreSQL的表中，phpPgAdmin有很大的帮助。一种导入数据的方法是将数据存放入CSV格式的文件中。很多程序例如Microsoft Excel都能够将数据导出为这种格式。

让我们看一个简单的示例。假设从一个Excel的电子表格中，我们保存了bpsimple数据库的item表的一些行到有标题的CSV文件中。这意味着第一行存放着列名，之后是数据，就像这样：

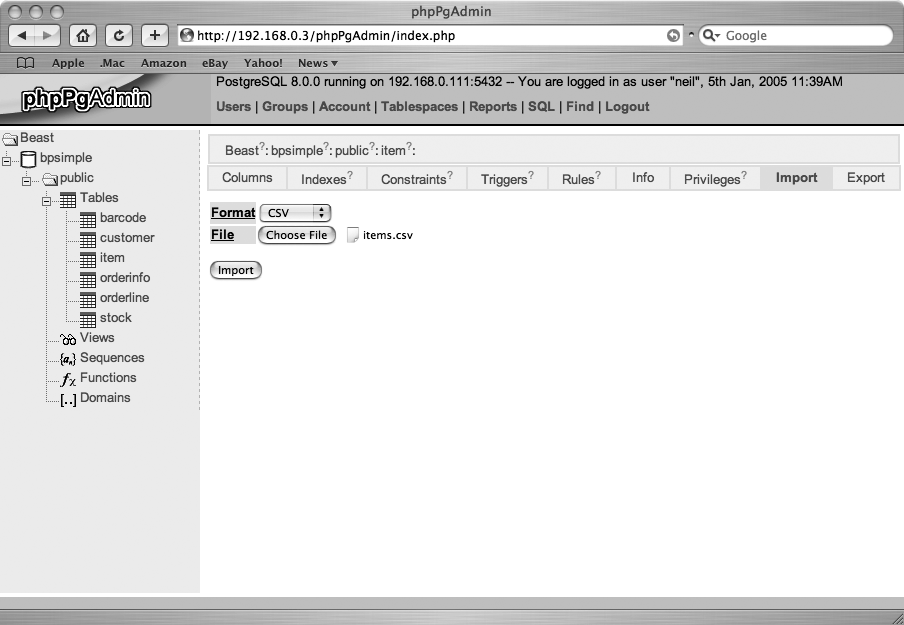
description,cost\_price,sell\_price

Wood Puzzle,15.23,21.95

Rubik Cube,7.45,11.49

Linux CD,1.99,2.49

我们可以先选定我们需要导入数据的表，，然后点击Import，然后选择需要导入文件的类型（在本例中，为CSV格式）和带入的文件名，就像图5-12所示。我们然后可以点击Import，然后（假设我们有权限）新的行将加入到我们数据库的表中。

  
图5-12使用phpPgAdmin导入数据

## Microsoft Access

虽然咋一看这是个古怪的念头，但我们确实可以使用Microsoft Access访问PostgreSQL。既然Access已经是一个数据库系统了，为什么我们还需要使用PostgreSQL来存储数据？而且既然这里有大量的可以与PostgreSQL协同工作的工具了，为什么我们还需要使用Microsoft Access？

首先，当开发一个基于数据库的系统时，我们需要考虑类似于数据卷，多用户并发，安全，健壮性和可靠性等相关需求。你可能会决定使用PostgreSQL因为它更符合你的安全模型，你的服务器平台以及你的数据增长预期。

虽然运行在UNIX或Linux服务器上的PostgreSQL是你理想的数据环境，但它可能不是对于你的用户以及它们的程序最好的或者最熟悉的环境。有可能需要允许用户使用类似于Access或者其他第三方工具为PostgreSQL建立报表或者数据登记表单。因为PostgreSQL有ODBC接口，这不但可行而且非常简单。

一旦你建立从Access到PostgreSQL的连接，你可以使用Access的全部功能来建立易用的PostgreSQL应用程序。在本章，我们将看到建立一个使用存储在远程的PostgreSQL服务器上数据的Access数据库，并基于这些数据生成一个简单的报表（我们假设你想当熟悉建立Access数据库和应用程序）。

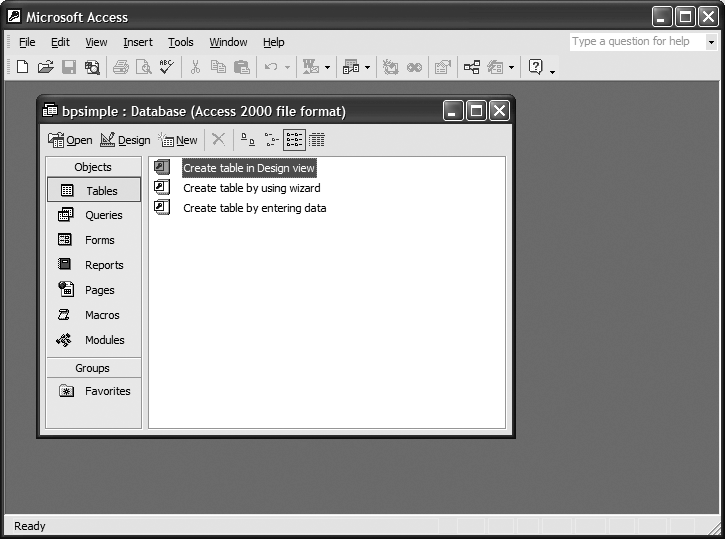
### 使用链接表

Access允许用户通过很多方法将一个表导入到一个数据库中，其中之一是通过叫链接表。这是一种在Access中表现为一个查询的表。数据在需要的时候从其他的数据源检索出来，而不是被拷贝到数据库中。这意味着当数据在外部数据库中发生变化，这种变化也会反映到Access中。

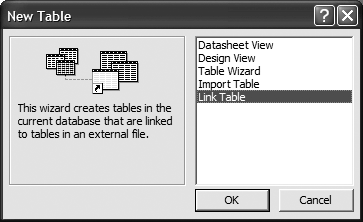
在bpsimple数据库中，我们有一个叫做item的表用来为我们销售的每项商品记录一个唯一标记，一个产品描述、一个成本价，以及一个售价。

作为一个示例，让我们按照步骤建立一个简单的Access数据用于更新我们示例数据库系统里存储的信息并且生成报表。

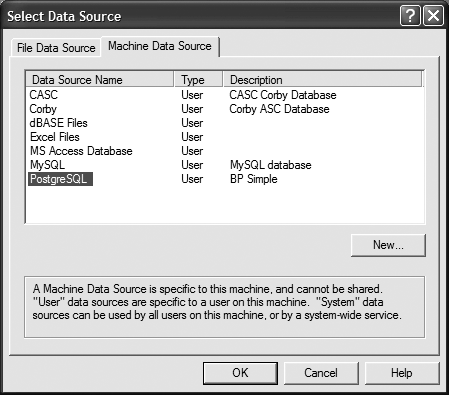
1. 在Access中，建立一个空数据库。点击窗口左边列表中的表格按钮，就像图5-13所示。

  
图5-13建立一个空白Access数据库

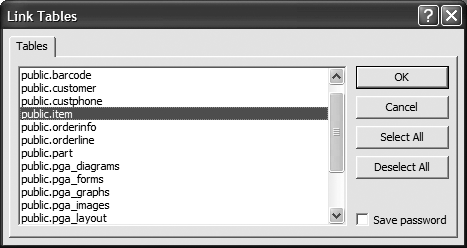
1. 点击新建，会弹出新建表对话框，选择链接表选项，如图5-14所示。

  
图5-14添加一个链接表

1. 在之后的连接对话框中，选择文件类型为ODBC数据库，之后弹出ODBC数据源选择对话框。选择机器数据源中恰当的PostgreSQL数据库连接，如图5-15所示。我们在本章靠前的“设置ODBC ”小节已经建立了一个合适的数据库连接。

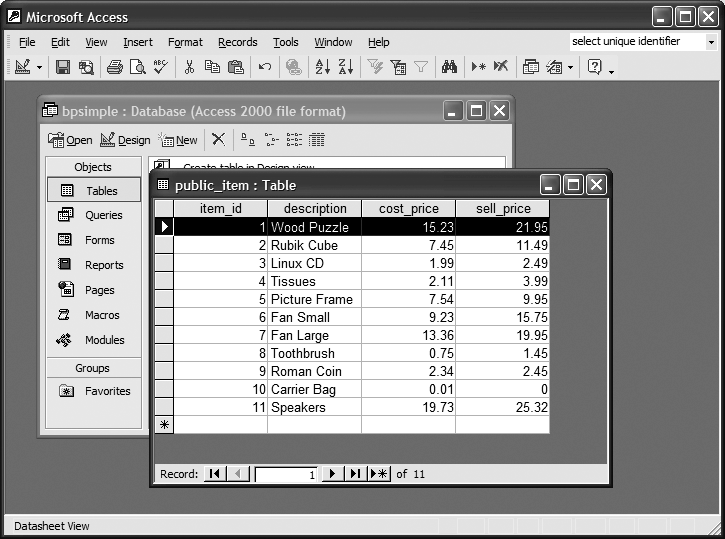
  
图5-15选择一个ODBC数据源

1. 当连接建立后，你将会看到一个远程数据库中存在的表的列表。你可以选择连接到这个列表中的一个或者多个表。在我们的示例中，我们将选择public.item来连接item表到我们的Access数据库，如图5-16所示。

  
图5-16选择要连接的表

注：在Access连接到一个表之前，它需要知道这个表中的哪个列可以用于唯一标记每条记录。也就是说，它需要知道哪个列扮演了主键的角色。对于这张表，item\_id列是主键，所以Access将选择它。对于没有定义主键的表，Access将提示你选择一个列。如果一个表有一个复合键，你可以选择不止一个列。

现在我们将发现Access数据库有一个新表，也叫做item，而且我们可以浏览和编辑它，就像数据存放在Access中一样。在展示在图5-17中。

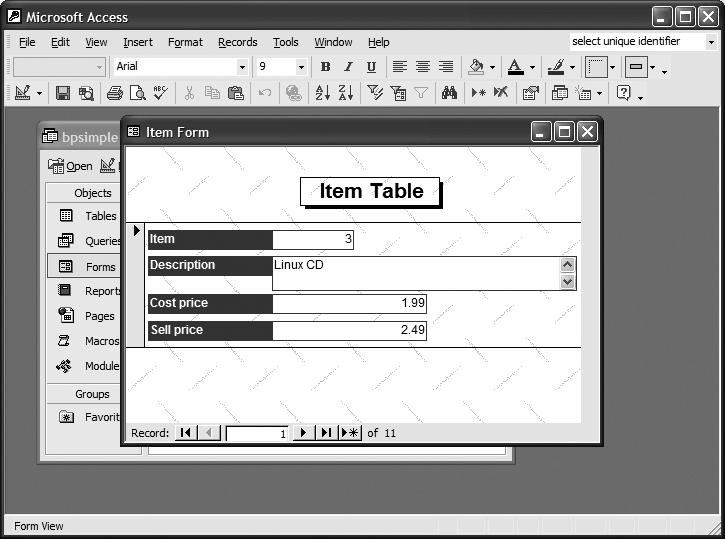
  
图5-17浏览一个链接表

以上就是关于怎么链接一个PostgreSQL数据库中的表到Access中的全部内容。

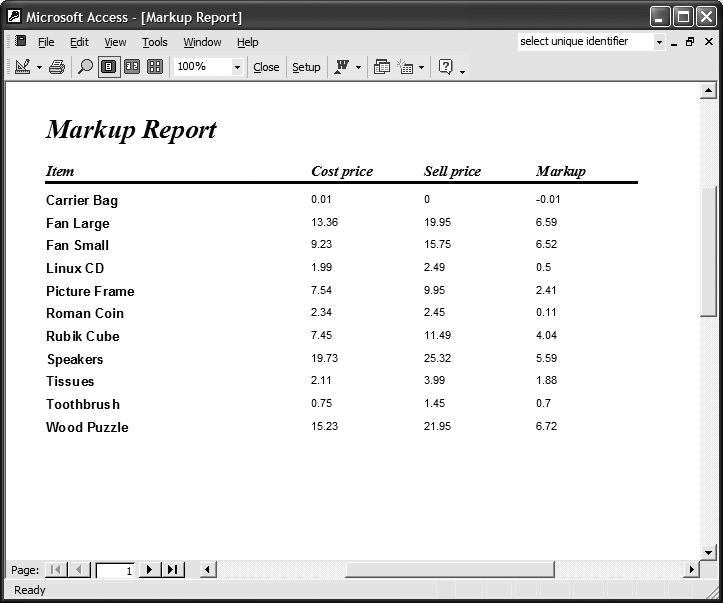
注：你可能在实际操作中看到的界面和以上图片中显示的略有不同，这依赖于你的Windows和Access的版本。如果你在表中看到一个额外的叫做oid的列，这是PostgreSQL内部的对象标识，可以被忽略掉。为了避免oid列被显示，请确定在ODBC数据源配置中去掉OID列的选项。

### 输入数据及建立报表

我们可以使用Access中的表浏览器检查PostgreSQL数据库中的数据以及添加更多行。图5-18显示了使用一个Access的数据登记表单来添加数据到item表中。我们可以使用Access的编程功能来建立更精细的数据数据登记应用程序来对输入的数据执行校验或者避免修改现有的数据。

  
图5-18一个简单的Access数据登记表单

建立报表也非常简单。使用Access的报表设计器针对存储在PostgreSQL中的数据生成报表，就像你针对其他Access表做报表一样。我们可以针对表中的数据选择一些列来回答问题。例如，图5-19显示了一个Access报表，显示了我们给item表中的产品添加的利润（也就是sell\_price（售价）和cost\_price（成本）之差）部分。

  
图5-19一个简单的Access报表

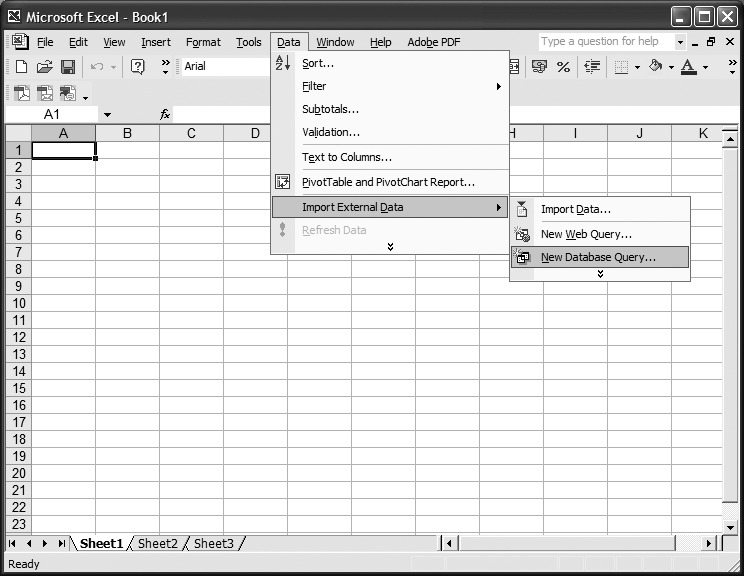
结合Microsoft Access和PostgreSQL增加了你建立数据库应用程序的选项。PostgreSQL的可伸缩性和可靠性以及Microsoft Access的易用性和你对它的熟悉程度的组合也许正是你所需要的。

## Microsoft Excel

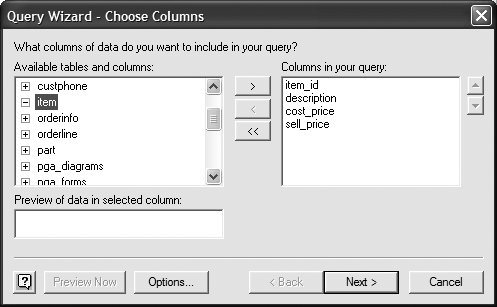
就像配合Microsoft Access使用一样，你也可以利用Microsoft Excel来对你安装的PostgreSQL添加功能。这和你使用Access工作的方法类似；你可以在你的电子表格中包含远程数据源（实际上是连接到远程数据源）。当数据发生变动后，你可以刷新电子表格让电子表格反映新数据。一旦你建立了一个基于PostgreSQL数据的电子表格，你可以使用Excel的功能例如图表来建立你的数据的图形化图示。

让我们将我们的报表示例从Access的报表扩展到一个图表，用以展示我们item表中的每个产品的利润。

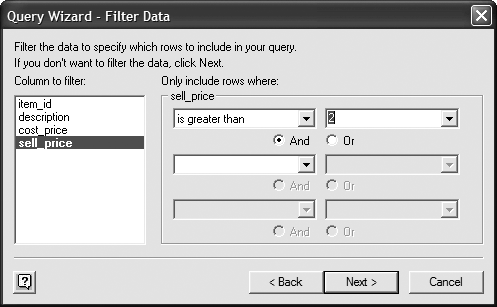
1. 我们需要告诉Excel电子表格的一部分需要链接到一个外部数据库表。我们从一个空白的电子表格开始，选择菜单选项来通过一个新的数据库查询导入外部数据，如图5-20所示。

  
图5-20导入数据到Excel中

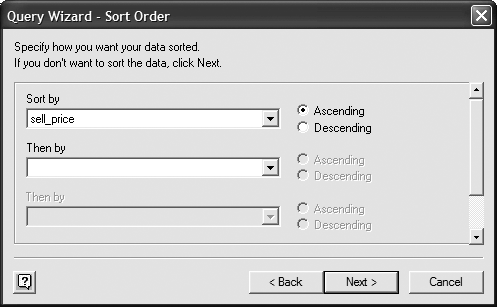
1. 出现一个ODBC数据源选择对话框让我们选择我们的数据源，就像Access中一样（如图5-15所示）。选择恰当的PostgreSQL数据库连接。
2. 当道数据库的连接建立后，你可以选择你想使用的表以及你想显示在电子表格中的列。在本例中，我们从item表中选择商品项标识符，描述和两个价格，如图5-21所示。

  
图5-21选择需要导入到Excel中的列

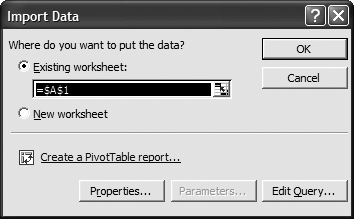
1. 如果你想限制在你的电子表格中显示的数据的行数，你可以通过下一步的对话框设置选择条件。在途5-22中，我们选择了那些售价高于2元的产品。

  
图5-22约束导入数据的行数

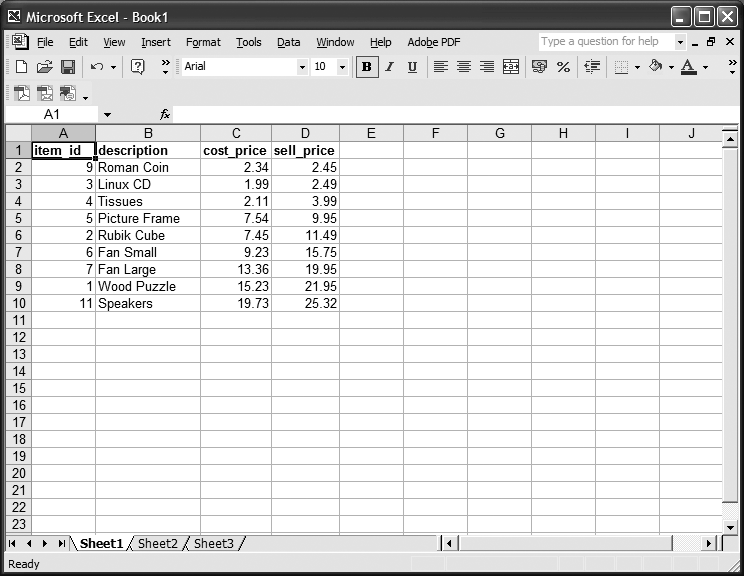
1. 最后，你可以选择选择按某个或者某些列，按照想要的方向排序。在本例中，我们选择按售价升序排列，如图5-23所示。

  
图5-23选择导入数据的排序标准

1. 在下一个对话框选择返回数据到Excel中。
2. 现在，你可以选择你希望导入的数据存放在你的电子表格的哪个位置了。最好是将从PostgreSQL中的表获得的数据存放在为它单独准备的工作版面中。这是因为你需要为数据库中数据的行数增长做准备。你将刷新电子表格，且需要空间用于数据的扩展。但是，在本例中，我们简单地让数据占据版面的左上角，如图5-24所示。

  
图5-24选择导入数据的位置

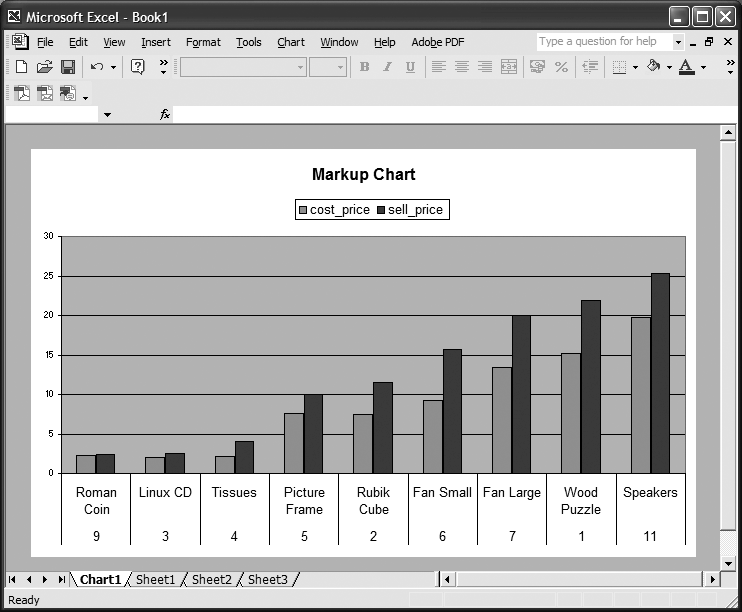
现在我们会发现数据出现在我们的工作版面中，如图5-25所示。

  
图5-25在Excel中查看导入的数据

我们可以使用这个电子表格针对数据执行一些计算。例如，我们可以通过使用适当的公式设置另一个列计算每个商品的销售利润。

警告：当数据库中的数据发生变化，Excel不会自动更新它获得版本的数据。为了保证Excel中看到的数据是准确的，你必须刷新数据。这可以简单的通过选择数据菜单项的刷新数据选项实现。

我们还可以利用Excel的一些功能为我们的PostgreSQL应用程序增值。在途5-26所示的示例中，我们增加了一个显示每项产品利润的图标。他可以简单地通过使用Excel的图表向导并选择图表的数据源为版面中PostgreSQL的数据区域来建立。当PostgreSQL数据库中的数据改变了，我们刷新电子表格，图表将随之自动改变。

  
图5-26使用PostgreSQL数据的一个Excel图表

## PostgreSQL相关工具的资源

查找PostgreSQL相关工具的一个好去处是pgFoundry，pgFoundry项目的网站位于<http://pgfoundry.org。GBorg>位于<http://gborg.postgresql.org/> 的网站也提供很多PostgreSQL相关的项目。有可能这项项目会被合并并通过<http://projects.postgresql.org>访问。

你可以在<http://techdocs.postgresql.org/guides/GUITools>找到一个支持PostgreSQL的图形界面工具列表。

一个正在开发中的叫做pgmonitor的PostgreSQL会话监控程序可以在<http://gborg.postgresql.org/project/pgmonitor>找到。这是一个Tcl/TK程序，允许你监控你的数据库的行为。它需要在数据库服务器上运行但它可以在你运行X Window系统的UNIX或者类UNIX系统的客户机上显示信息。

## 摘要

在本章中，我们看到了一些我们可以随意使用，最有效发挥PostgreSQL功能的工具。标准的发布带来了命令行工具psql，它有能力提供绝大多数我们可以用来建立和维护数据库的操作。

数据库管理可以在客户机上使用非常强大的pgAdmin III工具或者通过网络使用基于浏览器的phpPgAdmin工具完成。

我们可以使用Microsoft Office产品，包括Excel和Access，来操作和针对PostgreSQL数据库中保存的数据产生报表。这允许我们结合运行在UNIX或者Linux平台上的PostgreSQL系统的可伸缩性与健壮性和我们熟悉的工具。

现在我们回顾了一些PostgreSQL的工具，在下一章，我们将回到使用SQL处理PostgreSQL数据库中数据的主题，致力于研究插入、修改和删除数据。

# 数据交互

到现在为止，我们已经弄清楚了为什么一个关系数据库，特别是PostgreSQL是一个非常强大的用于组织和检索数据的工具。在前面的章节，我们研究了一些也可以用于管理PostgreSQL的图形界面工具，例如pgAdmin III。我们还学习了怎么在Microsoft Access中使用PostgreSQL以及通过使用Microsoft Excel添加更多功能。当然，如果没有在数据库中添加数据，它们没有一个对我们会有多大用处。在第三章，我们使用一些SQL脚本生成了我们的bpsimple数据库。

在本章，我们将跨越基本知识，学习更多关于处理数据的内容。我们将详细研究怎样插入数据到PostgreSQL数据库，更新已经存在于数据库中的数据以及从数据库删除数据。

随着本章的进展，我们将覆盖以下主题：

* 通过INSERT添加数据到数据库
* 插入数据到serial类型的列
* 插入NULL值
* 使用\copy命令从文本文件加载数据
* 直接从另一个程序加载数据
* 使用UPDATE更新数据库中的数据
* 使用DELETE从数据库中删除数据

## 添加数据到数据库中

也许出人意料，在我们第四章看到了复杂的SELECT语句之后，添加数据到一个PostgreSQL数据库相当容易。我们使用INSERT语句添加数据到PostgreSQL中。我们每次只可以加入数据到一个表中，而且通常我们是一次添加一行数据。

### 使用基本的INSERT语句

基本的INSERT SQL语句的语法非常简单：

INSERT INTO 表名 VALUES (每列的值的列表);

我们提供一个由逗号分隔的列的值的列表，它的顺序必须和表中列的顺序相同。

警告：虽然这个语法因为它的简单而非常诱人，但它也非常的危险，因为它依赖于具体的表结构，也就是列的顺序，但列的顺序可能由于数据库为支持更多的数据而修改。因此，我们强烈主张你避免使用这个语法，而是用稍后的“使用更安全的INSERT语句”小节介绍的更安全的语法。在更安全的语法中，列名和数据值一样被列出来了。我们在这里介绍简单的语法，因为你将看到它的通用用法，但是我们建议你避免这样使用它。

##### 尝试：使用INSERT语句

让我们添加一些新行到customer表中。我们需要做的第一件事是搞清楚正确的列的顺序。这和它在原来的建表命令CREATE TABLE中的列的顺序相同。如果我们无法访问这个表的建表SQL（不幸的是这很常见），那么我们可以使用命令行工具psql通过\d命令来查看这个表。假设我们想要在我们数据库中查看customer表的定义（存在于第三章），我们应该使用\d命令来请求显示它的描述。让我们现在就开始行动。

bpsimple=# \d customer

Table "public.customer"

Column | Type | Modifiers

-------------+-----------------------+------------------------------------------

-------------------------

customer\_id | integer | not null default nextval('public.customer

\_customer\_id\_seq'::text)

title | character(4) |

fname | character varying(32) |

lname | character varying(32) | not null

addressline | character varying(64) |

town | character varying(32) |

zipcode | character(10) | not null

phone | character varying(16) |

Indexes:

"customer\_pk" primary key, btree (customer\_id)

bpsimple=#

以上显示因为换行搞的有点不清晰，但它确实显示了我们需要的customer表的列的顺序。你可能会注意到customer\_id列的描述和我们第三章的CREATE TABLE语句中指出的不太一样。这是因为PostgreSQL就是通过这种方法实现serial类型的customer\_id的。现在，我们只需要知道它是整数型字段就好了。我们将在第八章讲解PostgreSQL怎么实现serial类型的列。

为了输入字符数据，我们必须确保它被单引号（'）包裹。数字不需要做任何特殊处理，对于空值，我们直接写NULL，或者在后面更复杂的INSERT语句中，我们直接针对那个字段不提供数据。

现在，我们知道了列的顺序了，我们可以这样写我们的INSERT语句：

bpsimple=# INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',

bpsimple-# '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');

INSERT 17331 1

bpsimple=#

你在INSERT之后看到的精确数字在你那里可能不同。真正重要的是PostgreSQL成功插入数据了。第一个数字实际上是PostgreSQL内部标识号，叫做OID，它通常是隐藏的。

注：OID（Object IDentification）是一个付给PostgreSQL中每一行的一个唯一数值，通常是不可见的。当你初始化数据库，一个计数器被建立。计数器用于唯一标记每一行。在这里，INSERT命令被执行后，17331就是新插入行的OID，1就是插入的行数。OID不是标准SQL的一部分，而且在表中一般不是连续的，所以我们强烈建议你知道它的存在但不要在应用程序中使用它。从8.0开始，PostgreSQL有选项来避免在表中建立OID，所以甚至它们的存在都是不可靠的。

我们可以通过SELECT轻松检查数据是否被正确插入，就像这样：

bpsimple=# SELECT \* FROM customer WHERE customer\_id > 15;

customer\_id | title | fname | lname | addressline | town | zipcode | phone

------------+-------+-------+-------+---------------+----------+---------+---------

16 | Mr | Gavyn | Smith | 23 Harlestone | Milltown | MT7 7HI | 746 3725

(1 row)

bpsimple=#

换行会因为你终端窗口的大小改变，但你应该可以看到数据被正常插入了。

假设我们需要插入另一行姓O'Rourke的。我们怎么处理数据中的单引号？如果在字符串中出现有一个单引号，我们需要在它之前写一个反斜杠。这个反斜杠叫做转义字符，它指明它后面的字符没有特殊的意义，就是数据的一部分。所以，要插入O’Rourke先生的数据，我们通过反斜杠转义他名字中的引号，像这样：

INSERT INTO customer VALUES(17, 'Mr', 'Shaun', 'O\'Rourke',

'32 Sheepy Lane', 'Milltown', 'MT9 8NQ', '746 3956');

然后检查插入的数据：

bpsimple=# SELECT \* FROM customer WHERE customer\_id > 15;

customer\_id | ti | fname | lname | addressline | town | zipcode | phone

------------+----+-------+----------+----------------+----------+---------+---------

16 | Mr | Gavyn | Smith | 23 Harlestone | Milltown | MT7 7HI | 746 3725

17 | Mr | Shaun | O'Rourke | 32 Sheepy Lane | Milltown | MT9 8NQ | 746 3956

(2 rows)

bpsimple=#

注：在某些情况下，为了便于页面输出，我们需要做一些小的变动。例如，在这里，我们将title缩短为ti。这些修正非常易读，而且我们可以确保这样做的目标很清晰。

##### 解析

我们使用INSERT语句往customer表里头添加数据，按照建表时列的顺序排列插入的值。要插入数字，则直接写数字。要插入字符串，则把字符串包含在单引号中。为了在插入的字符串里头包含单引号，我们必须在单引号之前添加一个反斜杠字符（\）。如果我们还需要插入一个反斜杠字符，那么我们需要写一对反斜杠，就像这样“\\”。

假设我们需要插入另一行地址有些特殊的数据，类似于Midtown Street A\33。我们怎么处理数据中已有的反斜杠？我们可以将一个反斜杠转义为两个反斜杠，就像这样：

INSERT INTO customer VALUES(18, 'Mr', 'Jeff', 'Baggott',

'Midtown Street A\\33', 'Milltown', 'MT9 8NQ', '746 3956');

以下是插入后的样子：

bpsimple=# SELECT \* FROM customer WHERE addressline='Midtown Street A\\33';

c\_id | ti | fname | lname | addressline | town | zipcode | phone

------+----+-------+---------+---------------------+----------+----------+----------

18 | Mr | Jeff | Baggott | Midtown Street A\33 | Milltown | MT9 8NQ | 746 3956

(1 row)

bpsimple=#

### 使用更安全的插入语句

当类似于刚才我们使用的INSERT语句在使用的时候，指定全部列或者将数据的顺序完全按照表的列顺序排列来插入数据不是很方便。这增加了一个风险因素，因为我们可能会意外地写一个错误的列顺序数据的INSERT语句。这可能导致加入错误数据到我们数据库中。

在上一个例子中，假设我们错误地交换了姓和名的列的位置。数据可以成功插入，因为这两个列都是文本列，PostgreSQL也无法检测我们的错误。如果我们后来需要按照按姓列出我们的客户，Gavyn将变成客户的姓，而不是Smith。

低质量的数据，或者完全错误的数据，是数据库的一个主要问题，所以我们会尽量执行很多预防措施来确保插入正确的数据。简单的错误对于我们只有十来行数据的数据库来说很容易发现，但是对于有成千上万客户数据的数据库，发现错误——尤其是不寻常名字——实际上会变得非常困难。

幸运的是，有一个即容易使用又更安全的INSERT语句的变体，就像这样：

INSERT INTO tablename( 列名的列表 ) VALUES ( 跟列的列表相对应的列的数值 );

在这个INSERT语句的变体中，我们必须列出列名以及那些列的相同顺序的数值，但这些可以和我们当初建表的顺序不同。使用这种变体，我们不再需要知道列在表中的顺序。我们也拥有了一个将要插入到表中的优美、清晰和基本上并行的列名的列表和数据列表。

##### 尝试：根据列名插入数据

让我们添加另一行到数据库，这次指定列名，就像这样：

INSERT INTO customer(customer\_id, title, fname, lname, addressline, ...)

VALUES(19, 'Mrs', 'Sarah', 'Harvey', '84 Willow Way', ...)

我们可以分多行输入一条INSERT语句，这让它更易读，更易于检查列名和对应数值的顺序是否相同。

让我们执行一个示例，分几行输入它以易于阅读：

bpsimple=# INSERT INTO

bpsimple-# customer(customer\_id, title, lname, fname, addressline, town,

bpsimple-# zipcode, phone)

bpsimple-# VALUES(19, 'Mrs', 'Harvey', 'Sarah', '84 Willow Way', 'Lincoln',

bpsimple-# 'LC3 7RD', '527 3739');

INSERT 22592 1

bpsimple=#

可以发现将字段名和需要插入的值对比是多么的简单。我们故意交换了fname和lname列的位置，仅仅用于展示这样是可行的。你可以使用任何你喜欢的列顺序；唯一要注意的就是插入的数值要匹配列出的列。

你还会发现psql的提示符在后续行的变动，它保持改变，直到通过输入分号结束命令。

提示：我们强烈推荐你使用带列名的INSERT语句，因为指明列名会让它更安全。

### 插入数据到serial类型的列中

到现在，是时候承认我们在customer\_id列上犯的一个小错误了。到本章这个时候，我们还没有讨论到如何插入部分列的数据到一个表而忽略其他的列。使用INSERT语句的第二种格式，也就是带列名的方法，我们可以做到这一点，并发现插入数据到带有serial类型的列的表中是多么重要。

你应该还记的从第二章碰到的特别特殊的数据类型serial，它实际上是一个整数，但是可以通过自动增长来给我们一种为每一行建立唯一ID数字的方法。本章到现在为止，我们插入数据，都为类型为serial的customer\_id自动提供了一个值。

让我们看看到现在为止customer表的数据：

bpsimple=# SELECT customer\_id, fname, lname, addressline FROM customer;

customer\_id | fname | lname | addressline

-------------+-----------+----------+---------------------

1 | Jenny | Stones | 27 Rowan Avenue

2 | Andrew | Stones | 52 The Willows

3 | Alex | Matthew | 4 The Street

4 | Adrian | Matthew | The Barn

5 | Simon | Cozens | 7 Shady Lane

6 | Neil | Matthew | 5 Pasture Lane

7 | Richard | Stones | 34 Holly Way

8 | Ann | Stones | 34 Holly Way

9 | Christine | Hickman | 36 Queen Street

10 | Mike | Howard | 86 Dysart Street

11 | Dave | Jones | 54 Vale Rise

12 | Richard | Neill | 42 Thatched Way

13 | Laura | Hardy | 73 Margarita Way

14 | Bill | O'Neill | 2 Beamer Street

15 | David | Hudson | 4 The Square

16 | Gavyn | Smith | 23 Harlestone

17 | Shaun | O'Rourke | 32 Sheepy Lane

18 | Jeff | Baggott | Midtown Street A\33

19 | Sarah | Harvey | 84 Willow Way

(19 rows)

bpsimple=#

当然，所有看上去都正常。但是，这里有一个小问题，由于强行给列customer\_id列赋值，我们无意间扰乱了PostgreSQL的内部序列计数器。

假设我们再尝试插入下一行，这次允许serial类型提供自动增量的customer\_id列的值：

bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,

bpsimple-# zipcode, phone)

bpsimple-# VALUES('Mr', 'Steve', 'Clarke', '14 Satview way', 'Lincoln',

bpsimple-# 'LC4 3ED', '527 7254');

ERROR: duplicate key violates unique constraint "customer\_pk"

bpsimple=#

很明显，有些东西出问题了，因为我们没有提供任何重复的值。发生问题的原因是因为在本章早些时候，当我们为customer\_id提供值的时候，我们跳过了PostgreSQL的serial类型列的自动分配ID的功能，这导致自动分配系统与表中的实际数据脱节。

警告：避免在插入数据的时候为serial类型的数据提供数值

脱节序列的问题相当少见，但通常是以下原因导致的：

* 你删除并重建了表，但你没有删除和重建序列生成器（PostgreSQL 8.0以及以后的版本会自动做这个事情）。
* 你混合了添加数据的方式——允许PostgreSQL通过serial类型的列自动生成值以及自己明确地为serial类型的列赋值。

在这个例子中，发生了后面的情况。我们把自己推进了混乱中，我们该怎么恢复呢？答案是我们需要帮PostgreSQL一把，将它内部的序列号改得和实际数据一致。

#### 访问序列生成器

当customer表被建立时，customer\_id列被定义为serial类型。你应该注意到PostgreSQL在当时给我们一些消息，说建立了一个customer\_customer\_id\_seq的序列生成器。而且，当我们使用\d要求PostgreSQL描述这个表的时候，我们发现这个列定义得很特殊：

customer\_id integer not null default nextval('customer\_customer\_id\_seq'::text)

PostgreSQL为这个列建立了一个特别的计数器，一个序列生成器，它可以用于产生唯一ID。注意这个序列生成器总是被命名为<表名>\_<列名>\_seq。这个列的默认行为被PostgreSQL自动指定为函数nextval(‘customer\_customer\_id\_seq’)的结果。当我们的INSERT语句没有提供这个列的数据，这个函数被PostgreSQL为我们自动执行。通过插入或提供数据到这个列，我们破坏了这种自动机制，因为如果提供了数据，函数将不会被调用。幸运的是，我们没有被迫从这个表删除所有数据并从头开始，因为PostgreSQL允许我们直接控制序列生成器。

当这样插入数据的时候，你通常会通过currval函数获得序列生成器的值：

currval('序列生成器名');

PostgreSQL将告诉你序列生成器当前的值：

bpsimple=# SELECT currval('customer\_customer\_id\_seq');

currval

---------

16

(1 row)

bpsimple=#

注：严格来说，currval告诉你的是上一次调用nextval后返回的值，所以为了让它工作，你需要要么插入一个新行或者直接在当前的psql会话中调用一次nextval函数。

就像你看到的，PostgreSQL认为最后一行的当前数字式16，但实际上，最后一行 是19。当我们尝试插入数据到customer表，空出customer\_id让PostgreSQL处理时，它尝试通过调用nextval函数为这个列提供一个值：

nextval('序列生成器名');

这个函数首先将提供的序列生成器的值加以，然后返回结果。我们可以这样直接尝试：

bpsimple=# SELECT nextval('customer\_customer\_id\_seq');

nextval

---------

17

(1 row)

bpsimple=#

我们当然可以通过反复针对序列生成器调用nextval以达到需要的值，但如果数值很大，这就帮不上太多忙了。作为替代方案，我们可以使用serval函数：

setval('序列生成器名', 新的值);

首先，我们需要弄清楚序列生成器的值应该是多少。这可以通过选择数据库中那个列的最大值来得到。要达到这一点，我们将使用max(列名)函数，它能够简单地告诉我们一个列的数值的最大值：

bpsimple=# SELECT max(customer\_id) FROM customer;

max

-----

19

(1 row)

bpsimple=#

PostgreSQL将回应它在customer表的customer\_id列中找到的最大值。（我们将在下一章详细讲解max()函数的更多细节）。现在，我们可以使用允许们设置一个序列生成器到任何值的函数setval()设置序列生成器了。这个表中当前最大值是19，而且序列生成器的值通常在它被使用前就增加了。因此，序列生成器通常应该和表中当前最大的值相同：

bpsimple=# SELECT setval('customer\_customer\_id\_seq', 19);

setval

--------

19

(1 row)

bpsimple=#

现在，序列生成器的数字正确了，我们可以插入我们的数据，并允许PostgreSQL提供serial类型的customer\_id列的值了：

bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,

bpsimple-# zipcode, phone) VALUES('Mr', 'Steve', 'Clarke', '14 Satview

bpsimple-# way', 'Lincoln', 'LC4 3ED', '527 7254');

INSERT 21459 1

bpsimple=#

成功了！PostgreSQL现在回到了正常状态，它将继续建立正确的序列值了。PostgreSQL 7.3以及以后的版本允许你在SELECT语句中使用DEFAULT关键字来指出插入的时候使用一个列的默认值，这在保持序列生成器的值的一致性非常有用。当我们显示使用customer\_id的值来添加行时，我们可以用以下的语句来代替：

INSERT INTO customer(customer\_id, title, fname, lname, addressline, town, zipcode, phone)

VALUES(DEFAULT, 'Mrs', 'Sarah', 'Harvey', '84 Willow Way', 'Lincoln', 'LC3 7RD', '527 3739');

在这里，customer\_id的默认值就是序列生成器的下一个值，因为customer\_id是一个serial类型的列。

我们将在第八章回到列的默认值这个主题。

### 插入空值

我们在第二章简短地提到空值可以通过INSERT语句插入到一个列中。现在让我们对这个做一点更细节的研究。

如果你在使用第一种格式的INSERT语句，也就是插入数据时按照表建立的时的列的顺序，你可以简单的在列的值的地方写NULL。注意你不能使用引号，因为它不是字符串。你还应该记住在SQL中NULL是一个特殊的未定义值，而不同于空字符串。

看看我们前一个例子：

INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',

'23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');

假设我们不知道姓。表定义中定义了fname列为允许NULL，所以加入不知道姓的数据时有效的。如果我们已经写了这个：

INSERT INTO customer VALUES(16, 'Mr', '', 'Smith',

'23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');

这不是我们故意的，因为我们想要先添加一个空串作为姓，也许暗示着Mr.Smith没有姓。我们的意思是使用NULL，因为我们不知道姓。

正确的INSERT语句应该是这样的：

INSERT INTO customer VALUES(16, 'Mr', NULL, 'Smith',

'23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');

注意NULL没有被引号包裹。如果使用了引号，fname将被设置成字符串“NULL”，而不是空值NULL。

使用第二种（更安全的）INSERT语句，也就是必须明确指明列名的情况，插入NULL会更容易，因为既不用列出列名也不用给他提供值，就像这样：

INSERT INTO customer(title, lname, addressline, town, zipcode, phone)

VALUES('Mr', 'Smith', '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');

注意fname列既没有被列出来，也没有一个值定义给他。我们也可以选择列出列，同时在值列表中给它一个NULL值。

尝试往一个不接受NULL值的列中加入NULL值是不会成功的。如果我们尝试加入一个没有名字列（lname）的客户：

bpsimple=# INSERT INTO customer(title, fname, addressline, town, zipcode,

bpsimple-# phone) VALUES('Ms', 'Gill', '27 Chase Avenue', 'Lowtown',

bpsimple-# 'LT5 8TQ', '876 1962');

ERROR: null value in column "lname" violates not-null constraint

bpsimple=#

注意我们没有为lname提供一个值，所以INSERT被拒绝了，因为customer表定义为那个列不允许NULL：

bpsimple=# \d customer

Table "public.customer"

Column | Type | Modifiers

-------------+-----------------------+------------------------------------------

-------------------------

customer\_id | integer | not null default nextval('public.customer

\_customer\_id\_seq'::text)

title | character(4) |

fname | character varying(32) |

lname | character varying(32) | not null

addressline | character varying(64) |

town | character varying(32) |

zipcode | character(10) | not null

phone | character varying(16) |

Indexes:

"customer\_pk" primary key, btree (customer\_id)

bpsimple=#

我们将在第八章我们将讲解我们怎么通过指定一个列的默认值，定义一个明确的默认值用于在插入数据时没有给出数值的列中。

### 使用\copy命令

虽然INSERT是用于添加数据到数据库的标准SQL方法，但它不总是最合适的。假设我们有大量的行需要加入到数据库，但是实际上的数据，可能是在一个电子表格里。将数据插入数据库的一种方法先是使用电子表格的导出功能，所以我们可以可能将电子表格导出为CSV（逗号分隔值）文件。然后我们可以使用类似于Eacs的文本编辑器，或者至少支持宏的其他工具，将所有的数据转换成INSERT语句。

考虑以下的数据：

Miss,Jenny,Stones,27 Rowan Avenue,Hightown,NT2 1AQ,023 9876

Mr,Andrew,Stones,52 The Willows,Lowtown,LT5 7RA,876 3527

Miss,Alex,Matthew,4 The Street,Nicetown,NT2 2TX,010 4567

我们可能转换它为一系列的INSERT语句，所以它看上去类似于这样：

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Miss','Jenny','Stones','27 Rowan Avenue','Hightown','NT2 1AQ','023 9876');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mr','Andrew','Stones','52 The Willows','Lowtown','LT5 7RA','876 3527');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Miss','Alex','Matthew','4 The Street','Nicetown','NT2 2TX','010 4567');

然后将它保存为有.sql扩展名的文本文件。

我们然后可以在psql中使用\i命令执行这个文件中的语句。这就是pop\_customer.sql文件工作的方法（我们在第三章使用这种方法构造我们的数据库）。注意这里我们需要PostgreSQL生成唯一键customer\_id的值。

但是，这样做不是非常方便。如果数据可以通过一种更通用的方法在扁平文件和数据库间移动会更好。在PostgreSQL中有好几种方法可以这么做。颇为难理解的是，他们都叫做copy命令。有一个PostgreSQL的命令叫做COPY，它可以通过扁平文件存储和还原数据，但它限于数据库管理员使用，因为文件必须在服务器上操作，而普通用户可能没有访问权限。另一个更有用的是通用的\copy命令，它基本上实现了COPY的全部功能，而且可以被任何人使用，而且数据是在客户机上读写的。所以基于SQL的COPY命令基本上是多余的。

注：SQL里头的COPY命令有一个优点：它明显比\copy命令快，因为它直接通过服务器进程执行。\copy命令是在客户进程中执行，有可能需要通过网络传输所有数据。而且COPY在发生错误的时候会更可靠。除非你有大量的数据，否则区别不会太明显。

\copy命令有以下语法用于导入数据：

\copy 表名 FROM '文件名'

[USING DELIMITERS '作为分隔符的单个字符']

[WITH NULL AS '代表NULL的字符串']

它看上去有点复杂，但它很容易使用。方括号“[]”中的部分是可选的，所以你只有在需要时使用它们。但是，注意文件名需要用单引号括起来。

选项[USING DELIMITERS '作为分隔符的单个字符']允许你指定输入文件中的每个列是怎么分隔的。默认情况下，输入文件的被假设为使用制表符（tab）分隔列的。在我们的例子中，我们假设我们从一个从电子表格里头导出的CSV文件开始。通常，CSV格式不是一个好的选择因为逗号可能出现在数据中，地址数据特别倾向于使用逗号字符。不幸的是，电子表格通产不提供合理的除了CSV文件之外的候选方案，所以你可能需要用好你拥有的。我们可以给出一个替代方案，使用管道符“|”，它经常被用作终止符，因为它极少出现在用户数据中。

选项[WITH NULL AS '代表NULL的字符串']允许你制定一个可以被翻译为NULL的字符串。默认情况下，假设为\N。注意在\copy命令中，你必须用单引号包裹这个字符串，因为这才能告诉PostgreSQL这是一个字符串，虽然在实际数据中不会有引号。所以如果你希望使用NOTHING作为空值加载到数据库，你应该使用选项“WITH NULL AS 'NOTHING'”。

那么例如如果我们不知道Mr.Hudson的名，数据就应该看起来是这样：

15,Mr,NOTHING,Hudson,4 The Square, Milltown,MT2 6RT,961 4526

当直接插入数据时，当心数据“干净”非常重要。你需要确保不丢失列，所有的引号字符都通过反斜杠被正确转义，没有二进制字符存在等。PostgreSQL会为你捕获大部分这种错误，仅加载有效数据，但是整理成千上万行基本上被完全加载的数据是一项缓慢、不可靠以及吃力不讨好的工作。因此在使用\copy命令尝试批量加载数据前尽量清理数据是非常值得的。

##### 尝试：使用\copy加载数据

让我们建立一个额外客户数据文件cust.txt，就像这样：

21,Miss,Emma,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245

22,Mr,Gavin,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245

23,Mr,Duncan,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245

你可以使用任何文本编辑器建立这个简单的cust.txt文件。为了简化操作，在这里不需要处理空值，所以我们仅仅需要指定逗号为列分隔符。执行以下命令加载这份数据：

\copy customer from 'cust.txt' using delimiters ','

注意这里在命令尾部没有分号“;”，因为他是一个由psql直接执行的命令，而不是SQL。Psql通过非常简短地回应“\.”，告诉我们所有操作成功。

然后执行以下语句：

SELECT \* FROM customer;

我们将看到额外的行被添加了。

但是，这里还有一个潜在的小问题。还记得序列号可能不同步的问题吗？不幸的是，使用\copy加载数据时这种问题可能发生的一种情况。让我们检查下在序列数字上发生了什么：

bpsimple=# SELECT max(customer\_id) FROM customer;

max

-----

23

(1 row)

bpsimple=# SELECT currval('customer\_customer\_id\_seq');

currval

---------

21

(1 row)

bpsimple=#

噢！！！存储在customer\_id中的最大值现在是23，所以下一个需要分配的ID应该是24，但是序列生成器将会为下一个数值分配22。不用担心，这很容易修复：

bpsimple=# SELECT setval('customer\_customer\_id\_seq', 23);

setval

--------

23

(1 row)

bpsimple=#

##### 解析

我们使用\copy命令直接加载从电子表格里导出的CSV格式的数据到我们的customer表。我们接下来需要修正用来为表中serial类型的customer\_id列生成customer\_id数字的序列生成器的数值，这个工作量明显比我们转换CSV格式的数据到一系列的INSERT语句要轻松得多。

### 直接从另一个程序加载数据

如果数据已经存在于桌面数据库中，例如Microsoft Access，那么有一种更简单的方法加载数据到PostgreSQL中。我们可以通过ODBC简单地附加PostgreSQL的表到Access数据库中，然后插入数据到PostgreSQL的表中。

通常，当你这么做的时候，你会发现你存在的数据不完全是你需要的，或者需要在插入到最终目标表的时候需要做一些加工。

即使数据的格式正确，直接将它插入数据库通常也不是一个好主意，相反的最好是先把数据加载到一个加载表，然后将数据从加载表传送到现实表中。使用一个中间的加载表是现实世界中应用程序将数据插入数据库的一个通用的做法，尤其是在原始数据的质量不稳定的情况下。数据先被加载进数据库的一个处理表，检查，如果有必要则修正，然后再将数据移入最终的表中。

通常，你需要写一个客户程序或者存储过程用于检查和修正数据，这将在第十章涉及。一旦数据准备好可以加载到最终表，我们可以使用另一个INSERT语句的变种，它允许我们在表和表之间移动数据，通过一条语句传输很多行。这是唯一的情况一条INSERT语句可以影响很多行。这就是INSERT INTO语句。

将数据从一个表插入另一个表的语法为：

INSERT INTO 表名(列名的列表) SELECT 普通的查询内容

##### 尝试：在表与表之间加载数据

假设我们有一个加载表tcust保存有一些额外用户信息需要加载到我们的customer表中。我们加载表的定义信息如下：

CREATE TABLE tcust

(

title char(4) ,

fname varchar(32) ,

lname varchar(32) ,

addressline varchar(64) ,

town varchar(32) ,

zipcode char(10) ,

phone varchar(16)

);

注意这个表没有任何的主键或者约束。为了交叉加载数据时加载数据到这个表更容易，我们通常这么做。移除约束让这么做更简单。还要注意除了customer\_id这个顺序数字的列以外的所有列都有了，因为PostgreSQL在加载数据的时候会为我们建立它。

假设我们已经加载一些数据到tcust表中（通过ODBC，\copy或其他方法），验证并修正了。然后，SELECT出来的内容就像这样：

bpsimple=# SELECT \* FROM tcust;

title | fname | lname | addressline | town | zipcode | phone

-------+-------+---------+----------------+-------–-+------------+----------

Mr | Peter | Bradley | 72 Milton Rise | Keynes | MK41 2HQ |

Mr | Kevin | Carney | 43 Glen Way | Lincoln | LI2 7RD | 786 3454

Mr | Brian | Waters | 21 Troon Rise | Lincoln | LI7 6GT | 786 7243

(3 rows)

bpsimple=#

要注意的第一件事是我们没法找到Mr.Bradley的电话号码。这是也不是一个问题。现在，让我们做个决定，我们不想加载这一行，但我们想加载所有其他的客户。在现实世界的情形中，我们可能尝试加载成千上万的新客户，而且很可能我们需要加载他们中的一些被验证或修正的分组。

这条INSERT语句的第一部分很容易写。我们将使用INSERT语句的完整语法，仅指定我们需要的列。这也通常是合理的选择：

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

注意我们没有指定加载customer\_id列。你应该还记得这个字段如果我们不处理他，PostgreSQL将自动为我们建立它的数值，这是一个比较安全的方法建立serial类型的数据。

我们现在需要写这个语句的SELECT部分了，它将为这个INSERT语句提供内容。记住我们还不想插入Mr.Bradley的信息，因为他的电弧号码是NULL，我们正在尝试找到他的号码。如果我们想要，我们可以加载Mr.Bradley的数据，因为电话号码字段可以接收NULL值。我们现在做的是针对数据使用一个比现实世界中需要的低级数据库规则更严格的规则。我们写一个这样的SELECT语句：

SELECT title, fname, lname, addressline, town, zipcode, phone FROM tcust

WHERE phone IS NOT NULL;

当然，这是一个完美有效的语句。让我们测试一下：

bpsimple=# SELECT title, fname, lname, addressline, town, zipcode, phone

bpsimple-# FROM tcust WHERE phone IS NOT NULL;

title | fname | lname | addressline | town | zipcode | phone

-------+-------+--------+---------------+--------+------------+----------

Mr | Kevin | Carney | 43 Glen Way | Lincoln | LI2 7RD | 786 3454

Mr | Brian | Waters | 21 Troon Rise | Lincoln | LI7 6GT | 786 7243

(2 rows)

bpsimple=#

这看上去正确。它找到我们需要的行，而且列的顺序和INSERT语句中的顺序也一样。所以我们现在可以将这两条语句放到一起执行，就这样：

bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,

bpsimple-# zipcode, phone) SELECT title, fname, lname, addressline, town,

bpsimple-# zipcode, phone FROM tcust WHERE phone IS NOT NULL;

INSERT 0 2

bpsimple=#

注意psql告诉我们插入了两条记录。现在，为了进一步的谨慎，让我们把customer表中的行取出来，仅仅是为了绝对确保它们被正确加载：

bpsimple=# SELECT customer\_id, fname, lname, addressline FROM customer WHERE

bpsimple-# town = 'Lincoln';

customer\_id fname | lname | addressline

-------------+-------+--------+----------------

19 | Sarah | Harvey | 84 Willow Way

20 | Steve | Clarke | 14 Satview way

24 | Brian | Waters | 21 Troon Rise

25 | Kevin | Carney | 43 Glen Way

(4 rows)

bpsimple=#

我们实际上获得了超过两行的数据，因为我们已经有来自Lincoln的客户了。可是我们可以看到，我们数据被正确插入了，且customer\_id的值也被建立了。

现在我们已经从tcust表加载了一些数据到customer表了，我们通常可以删掉tcust表中的那些数据了。为了示例的用途，我们暂时保留那些数据，将在后面的示例中删除它们。

##### 解析

我们指出了需要加载到customer表中的列，然后从tcust表中使用相同的顺序选择对应的数据集。我们没有指定需要加载customer\_id列，所以PostgreSQL使用它的序列生成器为我们生成唯一ID。

一个变通的方法，你会发现更容易，特别是有大量数据需要加载的时候，做法是在临时表加一个列，也许是叫做isvalid的类型为boolean的列。然后你将所有数据加载到临时表，然后使用我们在本章下一小节将正式讲解的UPDATE语句设置所有的isvalid为false：

UPDATE tcust SET isvalid = false;

我们没有指定一个WHERE从句；因此，所有的及路的isvalid列都被设置成false。我们继续在数据上做修改，在需要的地方修改。当我们确定一行已经修正和完成，我们就设置isvalid列为true。我们之后可以加载正确的数据，只选择那些isvalid为true的行：

bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

bpsimple-# SELECT title, fname, lname, addressline, town, zipcode, phone

bpsimple-# FROM tcust WHERE isvalid = true;

一旦完成了数据加载，我们可以使用以下语句将它们从tcust表删除：

DELETE FROM tcust WHERE isvalid = true;

然后继续处理tcust表里头剩下的数据。（我们将在本章接近最后的时候讨论DELETE语句）。

## 修改数据库中的数据

现在我们知道怎么通过INSERT语句将数据插入数据库，以及怎么使用SELECT将它检索出来。不幸的是，数据不宜一直保持不变。人们可能移动到不同的地址，改变电话号码等。我们需要一个方法来修改数据库中的数据。在PostgreSQL中，就像所有的基于SQL的数据库一样，这通过UPDATE语句实现。

### 使用UPDATE语句

UPDATE语句非常简单。它的语法是这样的：

UPDATE 表名 SET 列名 = 值 WHERE 条件

如果我们想一次性修改很多列，我们只需要简单的用逗号将赋值信息分隔开，像这样：

UPDATE customer SET town = 'Leicester', zipcode = 'LE4 2WQ' WHERE 一些条件

如果我们愿意，我们想同时修改多少列就修改多少列，只要让每个列出现一次。你会注意到你只能使用一个表名。这是因为SQL语法的原因。即使在一些情况下你需要修改两个分开的但相关的表，你需要写两个分开的UPDATE语句。你可以将这两条UPDATE语句放在一个事务中来确保要么两个操作都执行要么两个操作都没被执行。我们将在第九章近距离地讨论事务。

##### 尝试：使用UPDATE语句

假设我们现在查出了Mr.Bradley的电话号码，我们想更新我们的在线表customer。UPDATE语句的第一部分很容易：

UPDATE tcust SET phone = '352 3442'

现在我们需要指出需要修改的行，这也很简单：

WHERE fname = 'Peter' and lname = 'Bradley';

对于UPDATE语句，最好是经常检查WHERE从句。让我们就开始做吧：

bpsimple=# SELECT fname, lname, phone FROM tcust

bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';

fname | lname | phone

-------+---------+-------

Peter | Bradley |

(1 row)

bpsimple=#

我们可以看到我们需要修改的唯一一行被选中了，所以我们继续前进以，将两部分语句放到一起并执行它：

bpsimple=# UPDATE tcust SET phone = '352 3442'

bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';

UPDATE 1

bpsimple=#

PostgreSQL告诉我们一行被修改了。如果我们想要，我们可以重新执行我们的SELECT语句查看我们的结果是否正确。

##### 解析

我们分两步建立我们的UPDATE语句。首先，我们UPDATE语句的修改列数值的部分，然后我们写出WHERE从句来指出哪些行需要修改。在测试WHERE从句后，我们执行UPDATE语句，它修改了想要改变的行。

为什么我们这么小心测试WHERE从句并警告不要执行UPDATE语句的第一部分？答案是因为没有WHERE从句的UPDATE语句是绝对有效的。默认情况下，UPDATE语句会更新表中所有的行，一般情况下没有谁想这样。它也可能相当难纠正（如果WHERE从句有错）。

tcust是一个临时试验数据，所以让我们使用它测试下一个没有WHERE从句的UPDATE语句：

bpsimple=# UPDATE tcust SET phone = '999 9999';

UPDATE 3

bpsimple=#

注意psql会告诉我们有三行数据被更新。现在看看我们的数据：

bpsimple=# SELECT fname, lname, phone FROM tcust;

fname | lname | phone

-------+---------+----------

Kevin | Carney | 999 9999

Brian | Waters | 999 9999

Peter | Bradley | 999 9999

(3 rows)

bpsimple=#

这当然不是我们想要的！

警告：永远在执行UPDATE之前测试WHERE从句。WHERE从句中一个简单的错误会导致表中的很多甚至全部行被更新为相同的值。

如果你想要更新很多行，与其取出所有数据，不然简单的使用count(\*)检查有多少行匹配你的要求，我们将在下一章讲解count(\*)的细节。现在，你所需要知道的就是用count(\*)替换SELECT语句里的列名就会知道匹配我们要求的行有多少，而不是返回所有的数据行。实际上，这基本上就是count(\*)语句的全部内容了，但它在实际工作变得非常有用。以下是一个使用WHERE从句的SELECT语句用于检查有多少数据符合我们要求的例子：

bpsimple=# SELECT count(\*) from tcust

bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';

count

-------

1

(1 row)

bpsimple=#

这告诉我们WHERE从句充分限制到只有一行。当然，对于不同的数据，即使提供fname和lname也可能不能充分唯一标记一行。

### 通过另一个表更新

PostgreSQL有一个扩展用法允许通过另一个表更新，使用这样的语法：

UPDATE 表名 FROM 表名 WHERE 条件

这是对SQL标准的扩充。

##### 尝试：使用FROM更新

为了核对UPDATE的FROM选项的功能，我们需要先建立一个叫custphone的表，包含客户名和他们的电话号码。表结构为：

CREATE TABLE custphone

(

customer\_id serial,

fname varchar(32),

lname varchar(32) NOT NUL

phone\_num varchar(16)

);

也让我们插入一些数据到我们新建立的用于保存客户和他们电话号码的custphone表

bpsimple=# INSERT INTO custphone(fname, lname, phone\_num)

bpsimple-# VALUES('Peter', 'Bradley', '352 3442');

INSERT 22593 1

bpsimple=#

然后我们需要支持tcust表中需要更新的行：

bpsimple=# UPDATE tcust SET phone = custphone.phone\_num FROM custphone

bpsimple-# WHERE tcust.fname = 'Peter' AND tcust.lname = 'Bradley';

UPDATE 1

bpsimple=#

##### 解析

我们建立了一个新表包含客户的电话号码。然后我们插入数据到新建的表。最后，我们执行UPDATE语句，它根据要求修改了那行数据。

当UPDATE使用子查询来控制需要修改的行的时候，FROM从句允许在SET从句中包含从其他表来的列。事实上，FROM从句甚至都可以不需要。这是因为PostgreSQL默认建立一个到任何被用到的表的引用。

## 从数据库删除数据

本章我们需要学的最后内容就是从表中删除数据。有可能客户从不下订单，订单可能会被取消等，所以我们经常需要从数据库中删除数据。

### 使用DELETE语句

删除数据的常规方式是使用DELETE语句。它的语法类似于UPDATE语句：

DELETE FROM 表名 WHERE 条件

注意这里没有列的列表，因为DELETE是针对行的。如果你想要移除一个列的数据，你必须使用UPDATE语句来设置那个列的值为NULL或者其他适当的值。

现在我们已经从tcust表拷贝我们要的两条新客户数据到在线的customer表中，所以我们可以继续前进并从我们的tcust表删除不要的数据了。

##### 尝试：删除数据

我们知道在修改数据的时候遗忘WHERE从句有多么危险。我们能体会到意外删除数据会更非同小可，所以我们先从编写和通过SELECT语句检查我们的WHERE从句开始：

bpsimple=# SELECT fname, lname FROM tcust WHERE town = 'Lincoln';

fname | lname

-------+--------

Kevin | Carney

Brian | Waters

(2 rows)

bpsimple=#

很好——它列出了我们想要的两行。

现在我们可以在前面加上DELETE语句，经过最后的目视检查，执行它：

bpsimple=# DELETE FROM tcust WHERE town = 'Lincoln';

DELETE 2

bpsimple=#

警告：从数据库删除数据就是有这么简单，所以一定要非常小心

##### 解析

我们写了一个WHERE从句来选择我们需要从数据库中删除的数据并测试了它。然后我们执行一个DELETE语句删除了他们。

和UPDATE一样，DELETE一次只能工作在一个表中。如果我们需要从不止一个表中删除相关的行，我们需要使用事务，我们将在第九章碰到。

### 使用TRUNCATE语句

还有另一种方法从一个表删除数据。它从一个表中删除所有数据。除非它是包含在PostgreSQL 7.4以及以后版本中的事务中，否则你将没有办法恢复被删除的数据。这个命令是TRUNCATE，它的语法是：

TRUNCATE TABLE 表名

使用这个命令需要非常小心，只有当你非常确定要永久删除表中的数据才能使用。从某些方面说，它非常类似于删除表并重建它，除了它更容易操作且不会重置序列生成器。

##### 尝试：使用TRUNCATE语句

假设我们已经完成我们在表tcust中的操作，需要删除其中的所有数据。我们可以DROP这张表，但如果我们以后还会需要它，我们就必须重建它。作为替代操作，我们可以使用TRUNCATE，删除表中所有的行：

bpsimple=# TRUNCATE TABLE tcust;

TRUNCATE TABLE

bpsimple=# SELECT count(\*) FROM tcust;

count

-------

0

(1 row)

bpsimple=#

所有的行都被删除了。

##### 解析

TRUNCATE简单地将指定表的所有行删除。

如果你有一个达标，也许有成千上万行，并且希望删掉所有的行，默认情况下，PostgreSQL不会物理移除这些行，而是扫描全部行，并把每一行标记为已删除。这有助于在事务被回滚的时候恢复数据。即使在命令行中我们可能没有显示地启动事务，所有的命令在一个事务内被自动执行。扫描和标记表中上千行的行为执行起来非常缓慢。TRUNCATE语句非常高效地删除表的内容而不需要扫描数据。所以，对于非常大的表，它执行的效率比DELETE高很多。

注：有两种从表中删除全部数据的方法：没有WHERE的从句和TRUNCATE。虽然TRUNCATE不属于SQL92标准，但是一个非常通用的用于删除表中全部行的SQL语句。

你应该在大部分时候坚持使用DELETE，因为它是一个比较安全的删除数据的方法。虽然DELETE和TRUNCATE在某些情况下可能工作在不同的条件，例如有外键的表中。但是，在一些特殊的情况下当你想高效且不需恢复地删除一个表中的所有行，TRUNCATE是一个很好的方案。

## 摘要

在本章，我们看到了数据操作方法中除了SELECT之外的三种：使用INSERT命令插入数据，使用UPDATE命令修改数据以及使用DELETE命令移除数据。

我们学习了两种格式的INSERT命令，包括将数据包含在INSERT语句中或者从另一个表SELECT数据用于INSERT。我们看到了更长格式的INSERT语句会更安全点，因为所有相关的列都被列出来了，所有有更少的机会犯错误。我们还碰到INSERT的类似命令，更有用的PostgreSQL扩展的\copy命令，它允许数据直接从一个本地文件被插入到一个表中。

我们看到了你应该多么小心处理serial类型字段的序列计数器，以及怎么检查序列生成器的值，以及如果必要，改变它。我们看到，通常，最好允许PostgreSQL为你自动生成序列数字，而不是为serial类型的列提供一个数值。

我们看到了UPDATE和DELETE语句是多么简单，以及你怎样配合它们使用WHERE从句，就像和SELECT语句一样。我们还提到你应该经常使用SELECT测试UDPATE和DELETE的WHERE从句，因为这里导致的错误将很难纠正。

最后，我们看了TRUNCATE语句，一个从表中删除全部行非常高效的方法。因为这是一个不可恢复的删除，除非在事务中，否则它应该非常小心使用。

# 高级数据选择

在第四章，我们看了一些SELECT语句的细节以及我们怎么使用它来检索数据。这包括选择列，选择行，已经将表连接到一起。在前一章，我们学到了添加、更新和移除数据。在本章，我们回到SELECT语句，研究一些更高级的功能。你可能很少需要其中的一些功能，知道他们非常有用，因为你可以对SQL中可能有什么有一个很好的理解。

在本章，我们将碰到一些叫做聚集函数的函数，他们允许我们对一组记录获得结果。然后，我们会讲述一些更高级的连接功能，它们比我们在第四章讨论的简单的连接功能更能够控制我们查询结果。我们还会碰到在一个查询中使用几个SELECT语句的一整组查询，叫做子查询。最后，我们会讨论最重要的外连接，这也允许我们使用比我们刚才看到过的方法更灵活的方法连接几个表到一起。

通过本章，我们将覆盖以下主题：

* 合计函数
* 子查询
* UNION连接
* 自连接
* 外连接

## 聚集函数

在前面几章，我们使用了几个特殊的函数用来在选择中进行统计：max(列名)函数，告诉我们一个列中最大的值，count(\*)函数告诉我们一个表中有多少行。这些函数属于一小类叫做聚集函数的SQL的函数。这类函数在表7-1中列出了。

表7-1 聚集函数

| 聚集函数 | 描述 |
| --- | --- |
| count(\*) | 提供行的计数 |
| count(列名) | 提供指定字段中值不是NULL的行的计数 |
| min(列名) | 返回指定列中的最小值 |
| max(列名) | 返回指定列中的最大值 |
| sum(列名) | 返回指定列的值的合计总数 |
| avg(列名) | 返回指定列的值的平均数 |

聚集函记长非常有用且通常很容易使用。在本届，我们将介绍表7-1中列出的每个函数。PostgreSQL支持其他的聚集函数包括计算方差和标准差的函数。细节可以在PostgreSQL的文档中找到。

提示：psql的\da命令会列出PostgreSQL的聚集函数列表。

使用任何聚集函数SELECT语句都可以包含两个可选的从句：GROUP BY和HAVING。语法如下（在这里使用了count(\*)函数）：

SELECT count(\*), 列名列表 FROM 表名

WHERE 条件 [GROPY BY 列名 [HAVING 聚集条件]]

可选的GROUP BY从句是一个可以用于SELECT的附加条件。它通常在聚集函数被时候的时候才有用。它也可以用于提供类似于我们在第四章碰到的ORDER BY的功能，但仅用于聚集列。可选的HAVING从句允许我们通过GROPY BY选取的函数结果符合某些条件的特殊行。这些听上去都有点复杂，但实际上却非常简单，就像我们会在本章中看到的。

### count函数

我们从count函数开始，就像你在表7-1看到的，它有两种格式：count(\*)和count(列名)。

#### Count(\*)

Count(\*)函数提供结果集中行的计数。它在SELECT语句中扮演着一个特殊的列名。让我们尝试一个最简单的count(\*)来理解一些基本的概念。

注：在本章的例子中，和其他章一样，我们从我们示例数据库中干净的基本数据开始，所以读者可以沉浸在他们选择的任何一章。这意味着如果你接着上一章的示例数据继续，输出将稍微有点不同。你可以准备脚本删除表并重新建立，以及填充数据如果你想输出相同。

##### 尝试：使用count(\*)

假设我们想知道在customer表中有多少客户住在Bingham。我们可以简单的写一个这样的SQL：

SELECT \* FROM customer WHERE town = 'Bingham';

或者另一个更有效的返回更少数据的版本，我们可以写一个这样的SQL：

SELECT customer\_id FROM customer WHERE town = 'Bingham';

这做到了，但是是通过一个间接的方法。假设customer表包含成千上万的客户，其中有上千个住在Bingham的客户。在这种情况下，我们会检索到大量我们不需要的数据。函数count(\*)为我们解决了这个问题，它允许我们仅仅获得一行数据，包含我们选择的行数。

我们和往常一样写这个SQL，只是将选择的列用count(\*)代替，就像这样：

bpsimple=# SELECT count(\*) FROM customer WHERE town = 'Bingham';

count

-------

3

(1 row)

bpsimple=#

如果我们要知道所有客户的数量，我们只需要去掉WHERE从句：

bpsimple=# SELECT count(\*) FROM customer;

count

-------

15

(1 row)

bpsimple=#

你会发现我们只获得了包含总数的一行。如果你想检查结果，你只需要用customer\_id代替count(\*)来显示真实的数据就可以了。

##### 解析

Count(\*)函数允许我们获取一个对象的计数，而不是获得对象本身。它比获得数据本身提高了很多性能，因为我们不需要从数据库检索到我们不需要的数据，更坏的情况下，还要通过网络传送。

注：当你只需要行数时，你永远不要检索数据。

#### GROUP BY 和 count(\*)

如果我们想知道每个城镇有多少客户。我们可以可以先通过选择不同的城镇，然后分别统计每个城镇有多少客户。这是一个繁琐且单调的解决问题的方法。是不是在SQL中有一种公开的方法来简单解决这个问题呢？你可能想尝尝像这样的：

SELECT count(\*), town FROM customer;

这是基于我们现有知识的一个有理的猜测，但是PostgreSQL将产生一个错误信息，说它不是有效的SQL语法。你需要知道的额外的解决这个问题的一点语法是GROUP BY从句。

GROUP BY从句告诉PostgreSQL我们想使用一个聚集函数在一个列或者一组列的值改变时输出值并重置。它非常容易使用。你只需要在带有count(\*)的SELECT语句后简单地添加一个“GROUP BY 列名”从句。PostgreSQL将告诉你在表中这个列的每个值有多少条。

##### 尝试：使用GROUP BY

让我们尝试回答“每个城镇有多少客户？”这个问题。

第一步是写一条SQL检索计数和列名：

SELECT count(\*), town FROM customer;

然后添加GROUP BY从句，通过指定一个SQL查询告诉PostgreSQL在城镇每次改变的时候产生一个结果以及重置计数，就像这样：

SELECT count(\*), town FROM customer GROUP BY town;

以下是我们的行动：

bpsimple=# SELECT count(\*), town FROM customer GROUP BY town;

count | town

-------+-----------

1 | Milltown

2 | Nicetown

1 | Welltown

1 | Yuleville

3 | Bingham

1 | Histon

1 | Hightown

1 | Lowtown

1 | Tibsville

1 | Oxbridge

1 | Winnersby

1 | Oakenham

(12 rows)

bpsimple=#

就像你看到的，我们获得一个城镇的列表以及每个城镇的客户数量。

##### 解析

PostgreSQL按照GROUP BY从句里的列将结果排序。它然后保存一个行数的计数器，每次城镇名字改变，它就写出结果集并且重置计数器为零。你会认同这比写代码循环查询每个城镇来统计结果更容易。

如果我们需要，我们可以扩展这个想法到超过一个列，将我们所有选择的列都列在GROUP BY从句中。假设我们想知道两块信息：每个城镇有多少姓相同的客户。我们可以简单的在语句中的SELECT和GROUP BY部分添加lname：

bpsimple=# SELECT count(\*), lname, town FROM customer GROUP BY town, lname;

count | lname | town

-------+---------+-----------

1 | Hardy | Oxbridge

1 | Cozens | Oakenham

1 | Matthew | Yuleville

1 | Jones | Bingham

2 | Matthew | Nicetown

1 | O'Neill | Welltown

1 | Stones | Hightown

2 | Stones | Bingham

1 | Hudson | Milltown

1 | Hickman | Histon

1 | Neill | Winnersby

1 | Howard | Tibsville

1 | Stones | Lowtown

(13 rows)

bpsimple=#

注意Bingham列出了两次，因为在Bingham住有两个客户有不同的姓，分别是Jones和Stones。

还要注意输出是不排序的。在8.0版本之前的PostgreSQL会按town排序，然后是lname，因为这是他们在GROUP BY从句中的顺序。在PostgreSQL 8.0以及以后的版本，我们需要更明确地通过ORDER BY从句指定排序顺序。我们可以通过这样获取排序的输出：

bpsimple=# SELECT count(\*), lname, town FROM customer GROUP BY town, lname

bpsimple-# ORDER BY town, lname;

count | lname | town

-------+---------+-----------

1 | Jones | Bingham

2 | Stones | Bingham

1 | Stones | Hightown

1 | Hickman | Histon

1 | Stones | Lowtown

1 | Hudson | Milltown

2 | Matthew | Nicetown

1 | Cozens | Oakenham

1 | Hardy | Oxbridge

1 | Howard | Tibsville

1 | O'Neill | Welltown

1 | Neill | Winnersby

1 | Matthew | Yuleville

(13 rows)

bpsimple=#

#### HAVING从句和count(\*)

SELECT语句的最后一项可选部分是HAVING从句。这个从句对于SQL新手会有点感到迷惑，但它也不难使用。你只要记住HAVING是一种用于聚集函数的WHERE从句。我们使用HAVING来约束返回的结果 为针对特定的聚集的条件为真的行，例如count(\*) > 1。它的使用方法和我们通过列值限制行的方法一样。

警告：聚集无法使用在WHERE从句中。他们只能用在HAVING从句中。

让我们看一个例子。假设我们想知道有超过一个客户的城镇。我们可以使用count(\*)，然后直接查看相关的城镇。但是，在有成千上万的城镇的情况下，这不是一个合理的方案。作为替代，我们使用一个HAVING从句来约束结果为count(\*)大于一的行，就像这样：

bpsimple=# SELECT count(\*), town FROM customer

bpsimple-# GROUP BY town HAVING count(\*) > 1;

count | town

-------+----------

3 | Bingham

2 | Nicetown

(2 rows)

bpsimple=#

注意我们仍然需要使用GROUP BY从句，而且它出现在HAVING从句前。现在我们已经知道count(\*)，GROUP BY和HAVING的基本用法了，让我们将他们放到一起到一个大点的示例中。

##### 尝试：使用HAVING

假设我们考虑关于建立一个投递计划。我们需要知道我们所有客户的姓和所在城镇，除了Lincoln（也许它就是我们所在的本地城镇），而且我们只对城镇中客户名大于一个的客户感兴趣。

这没有它听上去难。我们只需要一点点建立我们的解决方案，这也通常是一个完成SQL的好方法。如果它看上去非常难，可以从一个简单点的，但类似的问题开始，然后扩展开始的方案知道解决你比较复杂的问题。事实上，面对问题，将它拆分为小的问题，然后分别解决每个小问题。

让我们从简单返回数据开始，而不是统计计数。我们按照town排序来让我们更容易看清楚究竟发生了什么：

bpsimple=# SELECT lname, town FROM customer

bpsimple=# WHERE town <> 'Lincoln' ORDER BY town;

lname | town

---------+-----------

Stones | Bingham

Stones | Bingham

Jones | Bingham

Stones | Hightown

Hickman | Histon

Stones | Lowtown

Hudson | Milltown

Matthew | Nicetown

Matthew | Nicetown

Cozens | Oakenham

Hardy | Oxbridge

Howard | Tibsville

O'Neill | Welltown

Neill | Winnersby

Matthew | Yuleville

(15 rows)

bpsimple=#

到目前为止，看起来不错，不是吗？

现在如果我们使用count(\*)来为了名进行计数，我们还需要针对lname和town使用GROUP BY：

bpsimple=# SELECT count(\*), lname, town FROM customer

bpsimple-# WHERE town <> 'Lincoln' GROUP BY lname, town ORDER BY town;

count | lname | town

-------+---------+-----------

2 | Stones | Bingham

1 | Jones | Bingham

1 | Stones | Hightown

1 | Hickman | Histon

1 | Stones | Lowtown

1 | Hudson | Milltown

2 | Matthew | Nicetown

1 | Cozens | Oakenham

1 | Hardy | Oxbridge

1 | Howard | Tibsville

1 | O'Neill | Welltown

1 | Neill | Winnersby

1 | Matthew | Yuleville

(13 rows)

bpsimple=#

事实上我们可以通过肉眼检查，但是我们已经接近完整的解决方案了，我们只需要添加一个HAVING从句来选取那些count(\*)大于一的行：

bpsimple=# SELECT count(\*), lname, town FROM customer

bpsimple-# WHERE town <> 'Lincoln' GROUP BY lname, town HAVING count(\*) > 1;

count | lname | town

-------+---------+----------

2 | Matthew | Nicetown

2 | Stones | Bingham

(2 rows)

bpsimple=#

正如你可以看到，当你把问题分解成几部分，解决方案会非常简单。

##### 解析

我们分三步解决了这个问题：

* 我们写一个简单的SELECT语句来选择我们感兴趣的行。
* 然后添加一个count(\*)函数以及一个GROUP BY从句，来统计唯一的lname和town组合。
* 最后我们添加了一个HAVING从句来抽取那些count(\*)大于一的行。

这种方法有一个小问题，在我们的小示例数据库中并不明显。在一个大数据库中，这个迭代开发方法有一些缺点。如果我们在一个由上千行客户数据的数据库中工作，我们在开发我们的查询的过程中需要等待我们的客户列表滚动很久。幸运的是， 通常有一个简单的方法通过使用数据的样本开发你的查询，使用主键。如果我们在我们所有的查询中使用条件WHERE customer\_id < 50，我们可以使用数据库中的customer\_id的前50条作为我们的样本。一旦我们满意了我们的SQL，我们可以简单地移除WHERE从句来针对整个数据库执行我们的方案。当然，我们需要留意样用于测试我们的SQL的本数据对完整的数据集有代表性且小心太少的样本可能无法完全执行我的SQL。

#### Count(列名)

一个count(\*)函数的微小的变种是使用一个列名替代“\*”。不同的是count(列名)统计表中提供的列值不为NULL的计数。

##### 尝试：count(列名)

假设我们添加了更多的数据到我们的customer表，一些新客户的电话号码为NULL：

INSERT INTO customer(title, fname, lname, addressline, town, zipcode)

VALUES('Mr','Gavyn','Smith','23 Harlestone','Milltown','MT7 7HI');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)

VALUES('Mrs','Sarah','Harvey','84 Willow Way','Lincoln','LC3 7RD','527 3739');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode)

VALUES('Mr','Steve','Harvey','84 Willow Way','Lincoln','LC3 7RD');

INSERT INTO customer(title, fname, lname, addressline, town, zipcode)

VALUES('Mr','Paul','Garrett','27 Chase Avenue','Lowtown','LT5 8TQ');

让我们查查我们有多少客户我们不知道电话号码：

bpsimple=# SELECT customer\_id FROM customer WHERE phone IS NULL;

customer\_id

-------------

16

18

19

(3 rows)

bpsimple=#

我们可以看到我们有三个客户我们不知道电话号码。让我们看看我们总共有多少客户：

bpsimple=# SELECT count(\*) FROM customer;

count

-------

19

(1 row)

bpsimple=#

总共有19个客户。现在如果我们想统计phone不为NULL的客户有多少，应该是16个：

bpsimple=# SELECT count(phone) FROM customer;

count

-------

16

(1 row)

bpsimple=#

##### 解析

Count(列名)和count(\*)唯一的区别是带列名的函数只统计这个列不是NULL的行，而\*格式的则统计所有的行。在其他方面，例如GROUP BY和HAVING，count(列名)和count(\*)的工作方式是相同的。

#### Count(DISTINCT 列名)

聚集函数count支持DISTINCT关键字，它约束函数只考虑一个列中值唯一的情况，不统计重复的。我们可以通过统计我们的customer表中不重复的城镇来说明它的这种行为：

bpsimple=# SELECT count(DISTINCT town) AS "distinct", count(town) AS "all"

bpsimple=# FROM customer;

distinct | all

----------+-----

12 | 15

(1 row)

bpsimple=#

在这里我们可以发现总共有15个城镇，但只有12个不重复的（Bingham和Nicetown出现了不止一次）。

现在，我们可以理解count(\*)以及学到了聚集函数的原理，我们可以将这相同的逻辑应用到其他聚集函数上了。

### Min函数

就像你预计的，min函数使用一个列名做参数且返回这个列中最小的值。对于numeric类型的列，结果应该和预期一样。对于时态类型，例如date的值，它返回最小的日期，日期既可以是过去也可以是未来。对于变长的字符串（varchar 类型），结果可能和预期有点不同：它在字符串右边添加空白后再进行比较。

警告：小心在varchar类型的列中使用min或者max，因为结果可能不是你预期的。

例如，假设我们想找到我们订单中收取的最小运费。我们可以使用min函数，就像这样：

bpsimple=# SELECT min(shipping) FROM orderinfo;

min

------

0.00

(1 row)

bpsimple=#

这显示最小的费用是零。

注意当我们在phone列尝试使用这个函数将会发生什么，因为我们知道这里有NULL值：

bpsimple=# SELECT min(phone) FROM customer;

min

----------

010 4567

(1 row)

bpsimple=#

你可能预期结果会是NULL，或者一个空串。因为NULL通常指位置，因此，min函数忽略NULL值。忽略NULL值是所有的聚集函数的一个特点，除了count(\*)（当然，是否一个电话号码是最小值又是另一个问题了）。

### Max函数

说max函数除了与min函数相反外其他都一样一点也不奇怪。和你想象的一样，max函数使用一个列名作为参数且返回那个列的最大值。

例如，我们可以通过以下方法查找我们订单中收取的最大运费值：

bpsimple=# SELECT max(shipping) FROM orderinfo;

max

------

3.99

(1 row)

bpsimple=#

和min函数一样，NULL值被max忽略掉，就像这个示例：

bpsimple=# SELECT max(phone) FROM customer;

max

----------

961 4526

(1 row)

bpsimple=#

这就是差不多全部你需要知道的关于max的内容。

### Sum函数

Sum函数使用一个列名作为参数并提供列的内容的合计。和min和max一样，NULL值被忽略。

例如，我们可以这样获得所有订单中运费的总和：

bpsimple=# SELECT sum(shipping) FROM orderinfo;

sum

------

9.97

(1 row)

bpsimple=#

和count一样，sum函数支持DISTINCT变体。你可以让它只统计不重复值的和，所以多条值相同的行只会被加一次：

bpsimple=# SELECT sum(DISTINCT shipping) FROM orderinfo;

sum

------

6.98

(1 row)

bpsimple=#

注意，在实际中，对这种变体的实际应用很少。

### Avg函数

我们要看的最后一个聚集函数是avg，它使用一个列名做参数并返回这个列数值的平均值。和sum一样，它忽略NULL值。这里是一个示例：

bpsimple=# SELECT avg(shipping) FROM orderinfo;

avg

--------------

1.9940000000000000

(1 row)

bpsimple=#

avg函数也可以使用一个DISTINCT关键字来处理不重复的值：

bpsimple=# SELECT avg(DISTINCT shipping) FROM orderinfo;

avg

--------------

2.3266666666666667

(1 row)

bpsimple=#

注：在标准SQL和PostgreSQL的实现中，没有mode和median函数。但是几个商业厂商以扩展模式支持它们。

## 子查询

到现在为止，我们遇到的各种SQL语句都只有一个SELECT，我们能够看到一个完整级别的使用多种方法结合两个甚至更多SELECT语句的数据检索语句。

子查询是指一个SELECT查询的一个或多个WHERE条件中的SELECT语句。子查询应该较单个SELECT语句的查询更难理解，但是它们非常有用且将开辟数据查询标准的新领域。

假设我们想找到价格比平均价格高的商品项目。我们可以通过两步做到：使用带聚集函数的SELECT语句找到平均价格，然后在第二个SELECT语句中使用这个值来查找我们要的行（使用在第四章介绍的cast函数），就像这样：

bpsimple=# SELECT avg(cost\_price) FROM item;

avg

--------------

7.2490909090909091

(1 row)

bpsimple=# SELECT \* FROM item

bpsimple-# WHERE cost\_price > cast(7.249 AS numeric(7,2));

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

1 | Wood Puzzle | 15.23 | 21.95

2 | Rubik Cube | 7.45 | 11.49

5 | Picture Frame | 7.54 | 9.95

6 | Fan Small | 9.23 | 15.75

7 | Fan Large | 13.36 | 19.95

11 | Speakers | 19.73 | 25.32

(6 rows)

bpsimple=#

这看上去确实非常不雅。我们实际上想要的是将第一个查询的结果直接传递给第二个查询，而不需要记住并输入到第二个查询中。

解决方案是使用子查询。我们将第一部分查询放入括号并将它用作第二个查询的WHERE从句的一部分，就像这样：

bpsimple=# SELECT \* FROM item

bpsimple-# WHERE cost\_price > (SELECT avg(cost\_price) FROM item)

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

1 | Wood Puzzle | 15.23 | 21.95

2 | Rubik Cube | 7.45 | 11.49

5 | Picture Frame | 7.54 | 9.95

6 | Fan Small | 9.23 | 15.75

7 | Fan Large | 13.36 | 19.95

11 | Speakers | 19.73 | 25.32

(6 rows)

bpsimple=#

就像你看到的，我们获得了相同的结果，但不需要中间步骤或者cast函数，因为结果已经是正确的类型了。PostgreSQL先执行括号中的查询。在获得回答后，它再运行外面的查询，使用了里面查询的结果。

如果需要，我们可以在WHERE从句里头包含很多子查询。我们不限制于只能使用一个，虽然需要多个嵌套SELECT的情况非常少见。

##### 尝试：使用子查询

让我们试一个复杂点的例子。假设我们想知道那些成本高于平均成本但售价低于平售价的产品（这些指标表明我们的利润不是很好，所以我们希望不会有太多符合这些标准的项目）。一般情况下的查询将是这种形式：

SELECT \* FROM item

WHERE cost\_price > average cost price

AND sell\_price < average selling price

我们已经知道平均成本可以使用“SELECT avg(cost\_price) FROM item”获得。查找平均售价可以通过类似的样子做到，就是使用查询“SELECT avg(sell\_price) FROM item”。

如果我们将这三个查询放到一起，我们得到这个：

bpsimple=# SELECT \* FROM item

bpsimple-# WHERE cost\_price > (SELECT avg(cost\_price) FROM item) AND

bpsimple-# sell\_price < (SELECT avg(sell\_price) FROM item);

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

5 | Picture Frame | 7.54 | 9.95

(1 row)

bpsimple=#

也许一些人需要查看Picture Frame的价格来看看它是否正确！

##### 解析

PostgreSQL首先扫描查询并发现有两个在括号中的子查询。它独自评价每个子查询，然后在主查询被执行前，将答案放回到主查询WHERE从句中的适当部分。

我们也可以使用附加的WHERE从句或者ORDER BY从句。它们可以完美有效地使用更常规的条件结合到子查询中的WHERE条件中。

### 返回多行记录的子查询

到现在为止，我们只看到了返回一条结果的子查询，因为子查询中使用了一个聚集函数。子查询也可以返回零行或者更多行记录。

假设我们想知道我们库存的哪些商品项的成本价高于10.0。我们可以使用一条单独的SELECT语句，就像这样：

bpsimple=# SELECT s.item\_id, s.quantity FROM stock s, item i

bpsimple-# WHERE i.cost\_price > cast(10.0 AS numeric(7,2))

bpsimple-# AND s.item\_id = i.item\_id;

item\_id | quantity

---------+----------

1 | 12

7 | 8

(2 rows)

bpsimple=#

注意我们为了让查询更短，使用了表的别名（stock表变成了s，item表变成了i）。我们所做的就是连接两个表（s.item\_id = i.item\_id），同时添加一个关于item表的成本的条件（i.cost\_price > cast(10.0 AS NUMERIC(7,2))）。

我们也可以把这个写作一个查询，使用关键字IN来检测这个列表的值。在这个上下文中使用IN，我们首先需要写一个查询来返回item表中中成本高于10.0的item\_id的列表：

SELECT item\_id FROM item WHERE cost\_price > cast(10.0 AS NUMERIC(7,2));

我们还需要从stock表查询一些内容：

SELECT \* FROM stock WHERE item\_id IN 值的列表

我们可以把这两个查询放到一起，就像这样：

bpsimple=# SELECT \* FROM stock WHERE item\_id IN

bpsimple-# (SELECT item\_id FROM item

bpsimple(# WHERE cost\_price > cast(10.0 AS numeric(7,2)));

item\_id | quantity

---------+----------

1 | 12

7 | 8

(2 rows)

bpsimple=#

这显示了相同的结果。

就像传统的查询，我们可以使用NOT IN来反转条件，我们也可以使用WHERE从句和ORDER BY条件。

使用子查询或者关联查询都可以查出相同结果的情况很常见。但是，并不总是这样；并不是所有的子查询可以写成连接，所以理解它们很重要。

如果你用子查询的情况也可以用连接做到，你应该选择哪个？这里有两个问题需要考虑：可读性和性能。如果查询是你偶尔使用在小表上且执行很快，使用你觉得可读性最好的。如果它是一个非常常用的在大表上的查询，那么使用不同的方式编写并试验来查看那种格式是最好就非常值得了。你可能会发现查询优化器有能力对两种样式都进行优化，所以它们的性能都是一样的，所以可读性自然胜出。你还可能发现性能严重地依赖于你数据库中的实际护具，或者说它伴随着不同表的行数的变化发生急剧变化。

注意：小心测试SQL语句的性能。有很多你无法控制的变量，例如操作系统的数据缓存。

### 相关子查询

我们到现在为止看到的子查询都是我们先执行一个查询获得一个答案，然后再把它“安插”到第二个查询中。这两个查询是不相关联的，叫做不相关子查询。这是因为没有链接表关联在内部和外部查询之间。我们可能在两部分SQL语句中使用相同的表的相同的列，但它们只是通过子查询的结果在传递到主查询的WHERE从句后关联起来。

有另一组子查询，叫做相关子查询，查询的两部分的关系比较复杂。在相关子查询中，内部的SELECT中的一个表会连接到外部SELECT中，从而确定这两个查询之间的关系。这是一组非常强大的子查询，它往往不能简写成一个带连接简单的SELECT语句。一个相关的查询具有的一般形式是：

SELECT columnA from table1 T1

WHERE T1.columnB =

(SELECT T2.columnB FROM table2 T2 WHERE T2.columnC = T1.columnC)

为了易于理解，我们已经将它们写成了一些伪SQL。需要强调的是在外面的SELECT语句中的表T1也出现在内部的SELECT中。因此，内部和外部的查询被视为相关的。你会发现我们使用了表的别名。这非常重要，因为相关子查询的表名的规则相当复杂，稍有不慎就可能得到奇怪的结果

提示：我们强烈建议你在相关子查询中使用表的别名，因为他是最安全的方法。

当这个相关的子查询被执行时，会发生一些非常复杂的事情。首先，T1表中的一行数据会由外部的SELECT查询出，然后T1.columnB列被传递到内部的查询，内部的查询然后被执行，通过传入的信息从T2表查询数据。查询结果然后被返回到外部的查询，外部查询在移动到下一行之前完成对WHERE从句的评估。这在图7-1中被展示出来。

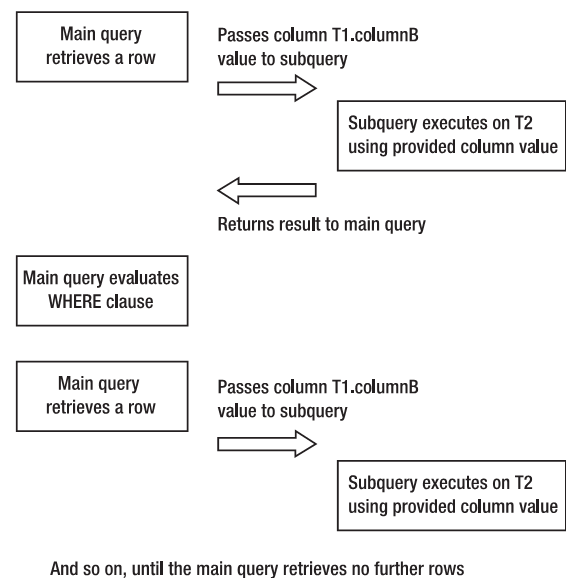


图7-1相关子查询的执行

如果你觉得它听起来有点冗长，那是因为它却是是这样的。相关的子查询通常执行起来非常没有效率。不过，它们偶尔也会解决一些相当复杂的问题。所以，最好知道它们的存在，即使你只是偶然使用它们。

##### 尝试：执行一个相关子查询

在一个简单的数据库，例如我们在用的这个，对相关子查询的需求很少，但我们仍然可以使用我们的示例数据库演示它们的用法。

假设我们想知道在Bingham的客户下的订单的日期。虽然我们可以用很老套的方法实现，但我们会使用相关子查询，就像这样：

bpsimple=# SELECT oi.date\_placed FROM orderinfo oi

bpsimple-# WHERE oi.customer\_id =

bpsimple-# (SELECT c.customer\_id from customer c

bpsimple(# WHERE c.customer\_id = oi.customer\_id and town = 'Bingham');

date\_placed

-------------

2000-06-23

2000-07-21

(2 rows)

bpsimple=#

##### 解析

查询开始于从orderinfo表选择一行数据。它之后在customer表上使用找到的customer\_id来执行子查询。子查询被执行，查找从外部查询中给出的customer中的行中城镇为Bingham的行。如果它找到一行，则传递customer\_id回到外部的查询，它再完成WHERE从句，如果WHERE从句为真，则打印date\_placed列。外部查询继续处理下一行，重复以上步骤。

也可能建立一个相关子查询，这个子查询在FROM从句中。以下是一个例子用来查找所有的在Bingham的给我们下过订单的客户的全部数据：

bpsimple=# SELECT \* FROM orderinfo o,

bpsimple=# (SELECT \* FROM customer c WHERE town = 'Bingham') c

bpsimple=# WHERE c.customer\_id = o.customer\_id;

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping |

customer\_id | title | fname | lname | addressline | town | zipcode | phone

--------------+-------------+-------------+--------------+----------+-----------

--+-------+-------+--------+--------------+---------+------------+----------

2 | 8 | 2004-06-23 | 2004-06-24 | 0.00 |

8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982

5 | 8 | 2004-07-21 | 2004-07-24 | 0.00 |

8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982

(2 rows)

bpsimple=#

子查询的结果作为主查询的一个表，也就意味着子查询产生了一个在Bingham的客户的结果集 。

现在你已经了解了怎么写相关子查询的。当你碰到一个你仿佛无法用常见的查询SQL解决的问题时，你可以发现相关子查询就是解决你的难题的答案。

### 存在子查询（Existence Subqueries）

另一种格式的子查询在WHERE中使用EXISTS关键字来检查是否存在，而不需要知道数据的内容。

如果我们想列出所有的下了订单的客户。在我们的示例数据库中没有太多。查询的第一部分很容易：

SELECT fname, lname FROM customer c;

注意我们给表customer一别名c，为子查询做准备。查询的下一步需要确定是否customer\_id也存在于orderinfo表中：

SELECT 1 FROM orderinfo oi WHERE oi.customer\_id = c.customer\_id;

有两个非常重要的方面需要在这里注意。首先，我们使用了一个通用的技巧。在我们需要执行查询但不需要返回结果集的地方，我们只要简单地在列名所在的位置使用1代替。这意味着如果找到任何数据，将返回1，这是一种简单有效的表达真值的方法。这是一种古怪的方法，所以我们先尝试一下它：

bpsimple=# SELECT 1 FROM customer WHERE town = 'Bingham';

?column?

----------

1

1

1

(3 rows)

bpsimple=#

它看上去有点古怪，但它确实有效。在这里不要使用count(\*)很重要；因为我们需要针对城镇是Bingham的每一行得到一个结果，而不仅仅是知道有多少客户是Bingham的。

第二个要注意的重要事情是我们在这个子查询中使用了customer表，它实际上是存在于主查询中。这就是它的相关性所在。和以前一样，我们给所有这些表别名。现在，我们需要将这两半放到一起了。

对于我们的查询，使用EXISTS才是将两个SELECT语句结合到一起的正确方法，因为我们只需要知道子查询是否返回了记录：

bpsimple=# SELECT fname, lname FROM customer c

bpsimple-# WHERE EXISTS (SELECT 1 FROM orderinfo oi

bpsimple(# WHERE oi.customer\_id = c.customer\_id);

fname | lname

-------+---------

Alex | Matthew

Ann | Stones

Laura | Hardy

David | Hudson

(4 rows)

bpsimple=#

EXISTS从句通常比其他类型的关联或者IN条件更高效。因此，在你选择怎么写一个子查询的时候，通常值得优先使用它而不是其他类型的连接。

## UNION连接

我们现在将开始弄清另一种结合多个SELECT语句到一起为我们提供更高级的查询能力的方法。让我们从一个我们需要解决的问题的示例开始。

在前一章，我们在添加数据到我们的主表customer时使用表tcust作为加载数据表。现在假设我们在加载新的客户数据，正处在新客户数据已经加载到tcust表进行清理但还未进入customer表的时候，我们被要求列出所有我们有客户的城镇，包括新数据。我们可能很有理由地指出因为我们还没有清理并加载客户数据到主表，我们无法确保新数据的准确性，所以这两个城市列表中的任何一个都可能不准确。然而，核对准确性可能不是很重要。也许我们所需要的只是一个大致的客户分布地理信息，而不是准确的数据。

我们可以通过从customer表中选择town，保存下来，再从tcust表里选择town，保存下来，然后结合这两个列表来解决这个问题。这确实看上去非常不雅，因为我们需要查询两个表，他们都包含一个城镇的列表，然后保存结果，之后在通过某些方法合并他们。

难道没有什么办法可以让我们自动联合这两个城镇的列表？就像你可以从本节的标题中获取到的，确实有个办法，叫做UNION连接。这些链接不是很常见，但是在一些条件下，它们却正是我们解决问题需要的，而且它们非常容易使用。

##### 尝试：使用UNION连接

让我们从放入一些数据到tcust表开始，所以它看上去就像这样：

bpsimple=# SELECT \* FROM tcust;

title| fname | lname | addressline | town | zipcode | phone

-----+---------+----------+----------------+----------+------------+---------

Mr | Peter | Bradley | 72 Milton Rise | Keynes | MK41 2HQ |

Mr | Kevin | Carney | 43 Glen Way | Lincoln | LI2 7RD | 786 3454

Mr | Brian | Waters | 21 Troon Rise | Lincoln | LI7 6GT | 786 7245

Mr | Malcolm | Whalley | 3 Craddock Way | Welltown | WT3 4GQ | 435 6543

(4 rows)

bpsimple=#

我们已经知道怎么从每个表中选择town。我们用一对简单的SELECT语句，就像这样：

SELECT town FROM tcust;

SELECT town FROM customer;

它们每一条语句都给出了一个城镇的列表，为了能结合他们，我们使用UNION关键字来连接这两个SELECT语句到一起：

SELECT town FROM tcust UNION SELECT town FROM customer;

我们输入我们的SQL语句，差分它为多行以让它更易读。注意psql提示符从=#转变成-#来表示这是继续输入的行，而且这里只有一个简单的分号在最后面，因为这只是一个单独的SQL语句：

bpsimple=# SELECT town FROM tcust

bpsimple-# UNION

bpsimple-# SELECT town FROM customer;

town

-----------

Bingham

Hightown

Histon

Keynes

Lincoln

Lowtown

Milltown

Nicetown

Oahenham

Oxbridge

Tibsville

Welltown

Winersby

Yuleville

(14 rows)

bpsimple=#

##### 解析

PostgreSQL从两个表中获取城镇的列表并结合它们到一个单独的列表。但是请注意，它消除了所有重复数据。如果我们想得到所有的城镇的列表，包括重复的，我们需要使用UNION ALL，而不是UNION。

这种结合SELECT语句的能力不限于一个列；我们可以结合城镇和邮政编码：

SELECT town, zipcode FROM tcust UNION SELECT town, zipcode FROM customer;

这将产生一个包含两个列的列表。它可能会更长点，因为它包含了邮政编码，因此这里有更多不重复的行被检索到。

UNION连接的使用有一些限制。你要连接的两个从两个表中查找列表的列必须有相同列数，而且选择的每个列必须都有相兼容的类型。

让我们看另一个例子用来连接两个不同但是兼容的列，title和town：

bpsimple=# SELECT title FROM customer

bpsimple-# UNION

bpsimple-# SELECT town FROM tcust;

title

----------

Keynes

Lincoln

Miss

Mr

Mrs

Welltown

(6 rows)

bpsimple=#

这个查询，虽然非常无意义，但是是有效的，因为PostgreSQL可以连接这两个列，即使title是一个固定长度的列而town是一个变长的列，因为他们都是字符串类型。例如如果我们尝试连接customer\_id和town，PostgreSQL会告诉我们无法做到，因为这两个列的类型不同。

总体来说，这就是你需要知道的所有关于UNION连接的内容。有时候，他们是一个方便用来从两个或多个表中联合数据的方法。

## 自连接

一种非常特殊的连接叫做自连接，它在我们想针对同一个表中的两个列使用连接时被使用。我们很少需要使用它，但偶尔它会非常有用。

假设我们卖的东西可以单独卖或者作为一套卖。作为示例，我们可以把桌子和椅子作为一套，也可以单独卖。我们想要做的是不但存储每个单独的项目，而且在它们作为一套销售时需要记录他们的关系。这通常被叫做零件分类（parts explosion），我们将在第12章再次碰到它。

让我们从建立一个可以不但保存商品项目的ID和描述，还能保存另一个商品ID的表开始，就像这样：

CREATE TABLE part (part\_id int, description varchar(32), parent\_part\_id INT);

我们将使用parent\_part\_id来存储作为组件的商品的ID。在本例中，一套桌椅的item\_id为1，椅子的item\_id为2，桌子的item\_id为3。相应的INSERT语句应该是这样：

bpsimple=# INSERT INTO part(part\_id, description, parent\_part\_id)

bpsimple-# VALUES(1, 'table and chairs', NULL);

INSERT 21579 1

bpsimple=# INSERT INTO part(part\_id, description, parent\_part\_id)

bpsimple-# VALUES(2, 'chair', 1);

INSERT 21580 1

bpsimple=# INSERT INTO part(part\_id, description, parent\_part\_id)

bpsimple-# VALUES(3, 'table', 1);

INSERT 21581 1

bpsimple=#

现在，我们已经存储好了数据了，但是我们怎么检索关于组成一个特殊组件的部件的信息呢？我们需要连接part表到它自己。这原来很容易。我们给表别名，然后我们可以写一个WHERE从句来针对相同的表使用不同的名字来处理：

bpsimple=# SELECT p1.description, p2.description FROM part p1, part p2

bpsimple-# WHERE p1.part\_id = p2.parent\_part\_id;

description | description

------------------+-------------

table and chairs | chair

table and chairs | table

(2 rows)

bpsimple=#

这成功运行了，但是结果有点点难以理解，因为我们的两个输出的列的名字相同。我们可以简单地通过使用AS给它们名字以纠正这点：

bpsimple=# SELECT p1.description AS "Combined", p2.description AS "Parts"

bpsimple-# FROM part p1, part p2 WHERE p1.part\_id = p2.parent\_part\_id;

Combined | Parts

------------------+-------

table and chairs | chair

table and chairs | table

(2 rows)

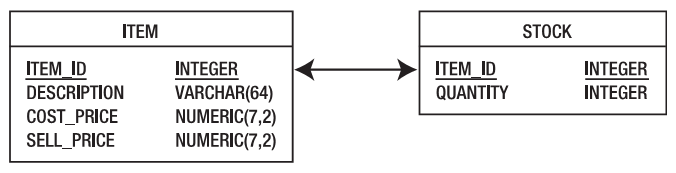
bpsimple=#

我们将在第十二章讨论怎么存储管理者/下属的关系到表中的时候再次看到自连接。

## 外连接

连一类连接为外连接。这种连接更接近于传统连接，但是它使用稍微不同的语法，这也是为什么我们推迟到现在才碰到它们的原因。

假设我们想知道我们销售的所有商品项目，说明我们的库存数量。这看似简单的要求以我们当前知道的SQL来做是惊人的困难的，虽然也可以做到。本例使用到了我们示例数据库中的item何stock表。就像你记得的，我们所有可以出售 的商品都保存在item表中，而我们有库存的商品存在stock表中，如图7-2所示：

  
图7-2 item和stock表的模式

让我们完成一个解决方案，从使用我们现在为止知道的SQL开始。让我们尝试使用一个单独的SELECT来连接两张表：

bpsimple=# SELECT i.item\_id, s.quantity FROM item i, stock s

bpsimple-# WHERE i.item\_id = s.item\_id;

item\_id | quantity

---------+----------

1 | 12

2 | 2

4 | 8

5 | 3

7 | 8

8 | 18

10 | 1

(7 rows)

bpsimple=#

很容易看到（因为我们碰巧知道item表中的item\_ids字段是顺序增长的，且没有间隙），一些item\_id丢失了。丢失的行为我们没有库存的那些商品，因为在item和stock表之间的连接无法连接到那些行，因为stock表没有那些item\_id的条目。我们可以使用一个子查询和一个IN从句找到这些丢失的行。

bpsimple=# SELECT i.item\_id FROM item i

bpsimple-# WHERE i.item\_id NOT IN

bpsimple-# (SELECT i.item\_id FROM item i, stock s

bpsimple(# WHERE i.item\_id = s.item\_id);

item\_id

---------

3

6

9

11

(4 rows)

bpsimple=#

我们可以把它翻译为“告诉我所有的item表中的item\_id，但排除那些出现在stock表中的”。

内部的SELECT语句就是我们前面使用的那个简单的语句，但这个时候，我们使用它返回的item\_id的列表作为另一个SELECT语句的一部分。主SELECT语句列出所有的item\_id，除了WHERE NOT IN从句排除了那些子查询中找到的那些item\_id。

那么现在我们有一个我们没有库存的商品的item\_id的列表，以及一个我们有库存的商品的item\_id的列表，只是是从不同的查询中检索到的。我们现在需要做的是连接这两个列表到一起，这是UNION链接的工作。但是，这里有一个小问题。我们的第一个语句返回两个列，item\_id和quantity，但是我们的第二个SELECT只返回item\_id，因为它们没有库存。我们需要添加一个伪列到第二个SELECT语句中，这样它才有和第一个SELECT一样的数量和类型的列。我们使用NULL。这里是我们完成了的查询：

SELECT i.item\_id, s.quantity FROM item i, stock s WHERE i.item\_id = s.item\_id

UNION

SELECT i.item\_id, NULL FROM item i WHERE i.item\_id NOT IN

(SELECT i.item\_id FROM item i, stock s WHERE i.item\_id = s.item\_id);

这看上去有点复杂，但我们还是先试试吧：

bpsimple=# SELECT i.item\_id, s.quantity FROM item i, stock s

bpsimple-# WHERE i.item\_id = s.item\_id

bpsimple-# UNION

bpsimple-# SELECT i.item\_id, NULL FROM item i

bpsimple-# WHERE i.item\_id NOT IN

bpsimple-# (SELECT i.item\_id FROM item i, stock s WHERE i.item\_id = s.item\_id);

item\_id | quantity

---------+----------

1 | 12

2 | 2

3 |

4 | 8

5 | 3

6 |

7 | 8

8 | 18

9 |

10 | 1

11 |

(11 rows)

bpsimple=#

在较早时候的SQL中，这可能是唯一的可以解决这类问题的办法，除了SQL89标准限制我们在第二个SELECT语句中使用NULL作为一个列。幸运的是，大多数厂家允许NULL，否者我们的日子可能更难过。如果我们不允许使用NULL，我们可以使用0（零）作为下一个方案。NULL更好，因为0可能导致误导；NULL可以总是表示为空白。

为了避免这种解决这类常见问题的相当复杂的解决方案，厂家设计出了外连接。不幸的是，因为这种类型的连接没有出现在标准中，所有的厂家发明出它们自己的解决方案，使用类似的方法，但是不同的语法。

Oracle和DB2使用一个在WHERE从句中加一个+标记的语法来指出一个表的所有值必须被显示出来（保留表），即使连接失败。Sybase在WHERE从句中使用\*=来指出保留表。这些语法都是相当简单，但不幸的是他们不同，对于你的SQL的移植性来说不好。

当SQL92标准出现后，它指出了一个非常通用的方法来实现外连接，为外连接带来了一个更合理的系统。厂家已经开始，但很慢地实现新的标准。（例如，Sybase 11和Oracle 8都在SQL92标准出现后出现，但都不支持它）PostgreSQL从7.1版开始实现了SQL92标准提供的这个方法。

注：如果你在运行一个7.1版本之前的PostgreSQL，你需要升级以便尝试本章中最后的例子。其实如果你还运行在一个比7.x更老的版本中，你值得升级了，因为8版本已经明显比旧版本提升了很多了。

SQL92标准中的外连接语法使用一个ON从句替代我们熟悉的WHERE从句来连接表，并添加了LEFT OUTER JOIN关键字。语法就像这样：

SELECT columns FROM table1

LEFT OUTER JOIN table2 ON table1.column = table2.column

连接到LEFT OUTER JOIN的表就是保留表，它的全部行将显示出来。

所以我们可以重写我们的查询，使用这个新语法：

SELECT i.item\_id, s.quantity FROM item i

LEFT OUTER JOIN stock s ON i.item\_id = s.item\_id;

是不是这样做看上简单到不像是真的？让我们运行一下它：

bpsimple=# SELECT i.item\_id, s.quantity FROM item i

bpsimple-# LEFT OUTER JOIN stock s ON i.item\_id = s.item\_id;

item\_id | quantity

---------+----------

1 | 12

2 | 2

3 |

4 | 8

5 | 3

6 |

7 | 8

8 | 18

9 |

10 | 1

11 |

(11 rows)

bpsimple=#

就像你看到的，结果与我们之前版本获得的结果一致。

你可以理解为什么大多数厂商他们需要实现一个外连接，即使它不是原始的SQL89标准。

也有一个相应的RIGHT OUTER JOIN，但LEFT OUT JOIN更常用（至少对于西方人，他们更习惯在左侧列出已知的项目而不是右边）。

##### 尝试：使用一个更复杂的条件

到现在为止我们使用的简单的LEFT OUTER JOIN已经非常好用了，但我们怎样添加更复杂的条件呢？

假设我们仅仅想知道stock表中的我们的库存多于两个的行，而且我们仅对售价高于5.0的行感兴趣。这是一个比较复杂的问题，因为我们要应用一条规则到item表（cost\_price > 5.0）以及另一条规则到stock表（quantity > 2），但我们还要从item表列出所有的item表中以上条件为真的行，即使没有库存。

我们所要做的仅仅是在左外连接表上结合ON条件，通过WHERE条件在表连接完成后限制返回的所有行。

在stock表上的条件是外连接的一部分。我们不想限制没有数量的行，所以我们将它作为ON条件的一部分：

ON i.item\_id = s.item\_id AND s.quantity > 2

对于item的条件，将应用于所有的行，我们使用一个WHERE从句：

WHERE i.cost\_price > cast(5.0 AS numeric(7,2));

把他们放到一起，我们得到了这个：

bpsimple=# SELECT i.item\_id, i.cost\_price, s.quantity FROM item i

bpsimple-# LEFT OUTER JOIN stock s

bpsimple-# ON i.item\_id = s.item\_id AND s.quantity > 2

bpsimple-# WHERE i.cost\_price > cast(5.0 AS numeric(7,2));

item\_id | cost\_price | quantity

---------+------------+----------

1 | 15.23 | 12

2 | 7.45 |

5 | 7.54 | 3

6 | 9.23 |

7 | 13.36 | 8

11 | 19.73 |

(6 rows)

bpsimple=#

##### 解析

我们使用LEFT OUTER JOIN从item表获得所有的值，选择性地连接到stock表中存在的且quantity大于2的行。这为我们得出了一组记录，这些记录从item表产生，但是quantity列（从stock表）要么为NULL要么是存在于stock表中且quantity大于2。WHERE从句然后被执行，它仅允许那些售价（从item表获得）大于5.0的行通过。

## 摘要

本章开始的时候学习了在SQL使用的聚集函数，用于从许多行中查询一个值。特别是，我们遇到了count(\*)函数，你会发现它可以广泛用于确定一个表中的行数。我们然后碰到了GROUP BY从句，它允许我们选择一组记录应用到聚集函数上，其后跟随着HAVING从句，它允许我们约束包含特定聚集值的记录。

下一步，我们学习了子查询，在这里，我们在一个查询中使用了另一个查询的结果。我们看到了一些简单的示例，并接触到了一个更难的查询——相关子查询——相同的列同时还出现在子查询中。

之后，我们还简短地学习了UNION连接，它允许我们结合两个查询的输出到一个简单的结果集。虽然它不是很常用，它有时候确实非常有用。最后，我们遇到了外连接，一个非常重要的功能，可以允许我们对两个表执行连接，即使在连接到第二个表失败的情况下，也能从第一个表检索出记录。

在本章，我们覆盖了SQL的一些难点。你已经看到了大量的SQL语法，所以如果你看到一些系统中存在的高级SQL，你理所当然应该理解它正在做什么。如果有些部分有点不清楚也不要着急。最好的真正理解SQL的方法是使用它，广泛地使用它。安装PostgreSQL，安装测试数据库以及一些示例数据并做实验。

在下一章，我们将学习一些数据类型的细节，建表，以及其他一些建立你自己数据库需要的信息。

# 数据定义

到现在为止，我们致力于研究PostgreSQL的工具和数据操作。虽然我们在本书老早的地方就建立了一个数据库，我们只是肤浅地了解了建表和PostgreSQL中的数据类型。我们通过使用主键和定义很少的几个不允许NULL值的列来简单定义我们的表。

在数据库中，我们最关注的就是数据的质量。针对数据执行非常严格的规则，通过数据库强制限制在最低级别，是我们可以用于保持数据一直的最好措施之一。这也是真正的数据库区别于简单的索引文件、电子表格等的功能之一。

在本章，我们将更深入地学习PostgreSQL中的数据类型以及如何操作它们。然后我们将学习如何管理表，包括如何使用约束，这使我们能够大大加强在添加数据到数据库中的表或从数据库中的表中删除数据时的规则。下一步，我们将简单学习视图。最后，我们将更深层次地学习外键约束并在建立我们更新版本的示例数据库时使用它们。我们将建立bpfinal数据库，我们将在之后的章节中使用这个示例数据库。

在本章，我们将覆盖以下主题：

* 数据类型
* 数据操作
* 表管理
* 视图
* 外键约束

## 数据类型

在最基本的层面上，PostgreSQL支持以下类型的数据：

* 布尔
* 字符
* 数字
* 时间（基于时钟）
* PostgreSQL扩展类型
* 二进制大对象（BLOB）

在这里，我们将学习它们每一个类型，除了非常少用的BLOB。如果你对BLOB类型感兴趣，可以在附录F查看怎么使用它。

### 布尔数据类型

布尔类型可能是最简单的类型。它只可以存储两个值，true和false，以及在值未知的时候，存储NULL。定义一个布尔类型的列的官方类型名为boolean，但它经常被缩写为简单的bool。

当数据被插入表的布尔类型的列中时，PostgreSQL能非常灵活地解释将翻译成true和false的值。表8-1提供了一个可以接收的值的列表以及它们对应的值。除了NULL，其他的任何值都将被拒绝。和SQL关键字一样，它们也是不区分大小写的；例如’TRUE’也将被翻译成布尔类型的ture。

表8-1 指定布尔值的方法

|  |  |
| --- | --- |
| 翻译为true | 翻译为false |
| '1' | '0' |
| 'yes' | 'no' |
| 'y' | 'n' |
| 'true' | 'false' |
| 't' | 'f' |

注：当PostgreSQL显示布尔类型列的内容是，它只会显示t,f和一个空格，分别代表true，false和NULL，而不管你设置这个列的值为什么（’true’，’y’，’t’等等）。因为PostgreSQL值存储这些可能状态中的一个，不是存储你实际设置的这个列的值，而仅仅是翻译后的值。

##### 尝试：使用布尔值

让我们建立一个简单的拥有bool列的表，然后用一些值做实验。与其用我们的bpsimple数据库使用我们的“真实”数据做测试，不然建立一个叫test的数据库。如果你使用了第三章中的示例，你应该已经建立了这个数据库了，只需要连接到它。如果没有，你可以像下面这样建立并连接到它：

bpsimple=> CREATE DATABASE test;

CREATE DATABASE

bpsimple=> \c test

You are now connected to database "test".

test=>

现在，我们建立一个叫testtype的表，拥有一个变长的字符串的列和一个布尔的列，插入一些数据，然后看看PostgreSQL存储了什么。以下是我们简短的psql会话：

test=> CREATE TABLE testtype (

test(> valused varchar(10),

test(> boolres bool

test(> );

CREATE TABLE

test=>

test=> INSERT INTO testtype VALUES('TRUE', TRUE);

INSERT 17862 1

test=> INSERT INTO testtype VALUES('1', '1');

INSERT 17863 1

test=> INSERT INTO testtype VALUES('t', 't');

INSERT 17864 1

test=> INSERT INTO testtype VALUES('no', 'no');

INSERT 17865 1

test=> INSERT INTO testtype VALUES('f', 'f');

INSERT 17866 1

test=> INSERT INTO testtype VALUES('Null', NULL);

INSERT 17867 1

test=> INSERT INTO testtype VALUES('FALSE', FALSE);

INSERT 17868 1

test=>

让我们检查以确认数据被插入了：

test=> SELECT \* FROM testtype;

valused | boolres

---------+---------

TRUE | t

1 | t

t | t

no | f

f | f

Null |

FALSE | f

(7 rows)

test=>

##### 解析

我们建立了一个有两个列的表testtype。第一个列保存一个字符串，第二个保存一个布尔值。我们然后插入数据到表中，每次都将第一个值设置为字符串，用以提示我们插入的值，第二个为相同的值，但是将被存储为布尔值。我们还插入一个NULL，来展示PostgreSQL（不像至少一个商业数据库）允许在boolean类型中存储NULL。然后我们取出这些数据，这显示了PostgreSQL如何翻译每一个我们传入的作为true，false或者NULL的值。

### 字符数据类型

在任何数据库中，字符数据类型可能是最广泛使用的类型。有三种字符数据类型，用于表示以下字符变量：

* 单个字符
* 固定长度字符串
* 长度可变字符串

有很多标准的SQL字符类型，但PostgreSQL也支持text类型，它类似于变长字符串，除了我们不需要定义任何上限来限制长度。但是这不是标准的SQL类型，所以要小心使用它。标准的类型可以使用char，char(n)和varchar(n)定义。表8-2列出了PostgreSQL的字符类型。

表8-2 PostgreSQL字符类型

|  |  |
| --- | --- |
| 定义 | 意义 |
| char | 单个字符 |
| char(n) | 一组长度固定为n的字符，长度不足用空白填充。如果你尝试存储一个过长的字符串，将会发生一个错误。 |
| varchar(n) | 一组长度不超过n的字符，长度不足也不需要填充。PostgreSQL扩展了SQL标准，允许指定没有长度的varchar，这实际上使长度不受限制。 |
| text | 实际上是一个长度不受限制的字符串，就像varchar一样，只是不需要定义最大长度。这是一个PostgreSQL针对SQL标准做的扩展。 |

如果让你选择三个标准类型用于存储字符串，你会选哪一个？通常，没有正式的答复。如果你知道你的数据库只会用在PostgreSQL中，你可以使用text类型，因为它很容易使用且不强制要你确定最大长度。它的最大长度只受限于PostgreSQL支持的行的最大大小。如果你使用的是早于7.1版本的PostgreSQL，行的大小限制在8KB（除非你从源码重新编译且改变了它）。从PostgreSQL 7.1开始，已经没有这个限制了。对于PostgreSQL 7.1和以后的版本，一个表中任何单个字段的长度的限制为1GB，实际上你永远不需要那么长的一个字符串。

主要缺点是text类型不是标准类型。所以，如果有很少的可能你有一天需要移植你的数据库到PostgreSQL之外的数据库，你应该避免使用text类型。通常，我们在本书中没有使用text类型，而更趋向于使用更标准的SQL类型varchar和char。

按惯例，char(n)在字符串的长度为固定或者行与行之间有稍微变化的时候使用，varchar(n)在字符串长度明显变化的时候使用。这是因为在一些数据库中，内部存储定长字符串的性能比变长的高很多，即使定长的需要存储一些不必要的字符。但是在内部，PostgreSQL使用相同的机制存储char和varchar类型。所以，对于PostgreSQL，使用哪种类型更多依赖于你自己的个人偏好。如果不同行之间的数据的长度明显不同，可以选择varchar(n)类型。还有，如果你不确定长度，可以使用varchar(n)。

就像boolean类型一样，所有的字符串类型可以包含NULL，除非你明确地定义这个列不允许NULL值。

##### 尝试：使用字符类型

让我们看看PostgreSQL的字符类型如何工作。首先，我们需要删除我们的testtype表，然后我们使用一些不同的列类型重建它：

test=> DROP TABLE testtype;

DROP TABLE

test=>

test=> CREATE TABLE testtype (

test(> singlechar char,

test(> fixedchar char(13),

test(> variablechar varchar(128)

test(> );

CREATE TABLE

test=>

test=> INSERT INTO testtype VALUES('F', '0-349-10177-9', 'The Wasp Factory');

INSERT 17871 1

test=> INSERT INTO testtype VALUES('S', '1-85723-457-X', 'Excession');

INSERT 17872 1

test=> INSERT INTO testtype VALUES('F', '0-349-10768-8', 'Whit');

INSERT 17873 1

test=> INSERT INTO testtype VALUES(NULL, '', 'T.B.D.');

INSERT 17874 1

test=> INSERT INTO testtype VALUES('L', 'A String that is too long', 'L');

ERROR: value too long for type character(13)

test=>

test=> SELECT \* FROM testtype;

singlechar | fixedchar | variablechar

------------+---------------+------------------

F | 0-349-10177-9 | The Wasp Factory

S | 1-85723-457-X | Excession

F | 0-349-10768-8 | Whit

| | T.B.D.

(4 rows)

test=> SELECT fixedchar, length(fixedchar), variablechar FROM testtype

test-> WHERE singlechar = 'S';

fixedchar | length | variablechar

---------------+--------+--------------

1-85723-457-X | 13 | Excession

(1 row)

test=> SELECT fixedchar, length(fixedchar), variablechar FROM testtype

test-> WHERE singlechar IS NULL;

fixedchar | length | variablechar

---------------+--------+--------------

| 0 | T.B.D.

(1 row)

test=>

##### 解析

我们建立了一个有三列的表，每个列对应一个标准的SQL类型。列singlechar保存单个的字符，fixedchar列保存刚好13个字符长度的字符串，variablechar列保存最长不超过128自己的字符串。我们然后储存不同的数据到几个列中，然后再检索它们一查看PostgreSQL是否正确存储了数据，虽然是在psql的输出中，你实际上看不到填充的字符。

我们尝试插入一个超过fixedchar列的字符串。这产生了一个错误，没有数据被插入。

我们检索了fixedchar列中字符串长度不同的行，使用内建的函数length()来检查它的大小。我们会在之后章的“用于数据操作的有用函数”小节中学习一些其他的用于操作数据的函数。

注：在PostgreSQL 8.0版本之前，length()函数在本例中总会返回13，因为存储类型char(n)是定长的且数据总是由空白填充了的，但是现在，length()函数忽略哪些空格并返回一个更有用的结果。

### 数字数据类型

PostgreSQL的数字数据类型比我们现在为止遇到的类型都稍复杂，但是它们也不是特别难理解。我们可以存储两类不同的数字到数据库中，整数和浮点数字。再细分，整数类型有一些子类型，包括serial类型（我们已经使用过，用于在一个表中建立一个唯一的值）和不同大小的整数，如表8-3所示。

表8-3 PostgreSQL整数数据类型

|  |  |  |
| --- | --- | --- |
| 子类型 | 标准名 | 描述 |
| Smalll integer | Smallint | 一个2字节的符号型整数，可以存储-32768到32767的数字 |
| Integer | Int | 一个4字节的符号型整数，可以存储-2147483648到2147473647的数字 |
| Serial |  | 和integer一样，除了它的值通常是由PostgreSQL自动输入的。 |

浮点数据也可以再细分，分为提供通用功能的浮点值和固定精度的数字，如表8-4所示。

表8-4 PostgreSQL浮点数据类型

|  |  |  |
| --- | --- | --- |
| 子类型 | 标准名 | 描述 |
| float | float(n) | 支持最少精度为n，存储为最多8字节的浮点数。 |
| float8 | real | 双精度（8字节）浮点数字 |
| numeric | numeric(p,s) | 拥有p个数字的实数，其中小数点后有s位。不像float，这始终是一个确切的数字，但工作效率比普通浮点数字低。 |
| money | numeric(9,2) | PostgreSQL特有的类型，但在其他数据库里也普遍存在。Money类型从PostgreSQL 8.0开始不赞成使用，且可能在以后版本中取消。你应该使用number类型代替。 |

被分成整数和浮点数据这两类的原因很容易理解，但numeric类型的用途不太明显。

浮点数据以科学计数法方式，使用尾数和幂存储。对于numeric类型，你可以在执行计算时同时指出精度和准确的存储的数字。你还可以指出小数点后的数字个数。实际的小数点位置变得非常自由了！

警告：最常见的错误时认为numeric(5,2)可以存储一个类似于12345.12的数字。这是错误的。在这里头存储的总归数字个数只有五个，所以定义为numeric(5,2)的数字在溢出前最大只能存储999.99。

PostgreSQL一般会捕捉对无法存储到字段中的值的插入，所以尝试插入很大的数字到任何类型的数字列将失败。

##### 尝试：使用数字类型

现在，我们可以实验数字数据类型。首先，我们需要删除我们的testtype表，然后使用一些不同的列类型来重新建立它：

test=> DROP TABLE testtype;

DROP TABLE

test=> CREATE TABLE testtype (

test(> asmallint smallint,

test(> anint int,

test(> afloat float(2),

test(> areal real,

test(> anumeric numeric(5,2)

test(> );

CREATE TABLE

test=> INSERT INTO testtype VALUES(2, 2, 2.0, 2.0, 2.0);

INSERT 17883 1

test=> INSERT INTO testtype VALUES(-100, -100, 123.456789, 123.456789, 123.456789);

INSERT 17884 1

test=> INSERT INTO testtype VALUES(-32768, -123456789, 1.23456789,

test-> 1.23456789, 1.23456789);

INSERT 17885 1

test=> INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,

test-> 23456789.123456789, 123456789.123456789);

ERROR: numeric field overflow

DETAIL: The absolute value is greater than or equal to 10^8 for field with

precision 5, scale 2.

test=>

test=> INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,

test-> 123456789.123456789, 123.123456789);

INSERT 17886 1

test=>

test=> SELECT \* FROM testtype;

asmallint | anint | afloat | areal | anumeric

-----------+------------+--------------+--------------+----------

2 | 2 | 2 | 2 | 2.00

-100 | -100 | 123.457 | 123.457 | 123.46

-32768 | -123456789 | 1.23457 | 1.23457 | 1.23

-32768 | -123456789 | 1.23457e+008 | 1.23457e+008 | 123.12

(4 rows)

test=>

##### 解析

我们建立了一个表，包含一个短整数列，一个普通整数列，一个浮点数字列，一个实数列，一个精度为5，小数点后为2的数字列。

你会发现float和real类型的行为非常相似，但是numeric列的行为有点不同。Numeric类型不是存储接近的数，而是在小数后面进行后超出固定长度的部分进行四舍五入。如果我们存储太大的数据到其中，INSERT将失败。还要注意float和real也会对数字四舍五入；例如123.456789被四舍五入为123.457。

### 时间型数据类型

时间型数据类型存储和时间相关的信息，在第四章，我们看到了如何控制数据的格式。PostgreSQL有一系列的与日期和时间相关的类型，如表8-5所示，但是我们在本书中通常只限制于SQL92标准的类型。

表8-5 PostgreSQL的时间数据类型

|  |  |
| --- | --- |
| 定义 | 意义 |
| date | 存储日期信息 |
| time | 存储时间信息 |
| timestamp | 存储日期和时间 |
| interval | 存储timestamp之间差别的信息 |
| timestamptz | PostgreSQL扩展的类型，存储包含时区信息的timestamp |

##### 特殊数据类型

由于PostgreSQL源于一个用于研究的数据库系统，PostgreSQL拥有一些少见的数据类型用于存储几何和网络数据类型，如表8-6所示。使用PostgreSQL的这些任何一种特殊功能都会使一个PostgreSQL数据库的可移植性变得非常的差，所以通常，我们趋向于避免这些扩展。要获得更多的关于这些类型的信息，查询PostgreSQL的文档，在“数据类型”部分。

表8-5 PostgreSQL的特殊数据类型

|  |  |
| --- | --- |
| 定义 | 意义 |
| box | 矩形盒子 |
| line | 一组点 |
| point | 一对几何学的数字 |
| lseg | 一条线段 |
| polygon | 一条封闭的几何线 |
| cidr或inet | 一个IPv4的地址，录入192.168.0.1 |
| macaddr | 以MAC地址（以太网卡物理地址） |

注：PostgreSQL也允许你使用SQL命令CREATE TYPE在数据库中建立你自己的类型。这通常不需要，而且在一定程度上，它是PostgreSQL独有的。更多的信息可以在官方文档中找到。注意建立你自己的类型可能导致数据库的模式非常限于PostgreSQL中使用，因为用户自定义类型无法被移植。

### 数组

PostgreSQL有另一个独特的功能：能够在表中存储数组。在SQL99标准之前，这不是一个标准的功能，所以在数据库实践中很少见。通常，一个数组需要通过使用一个附加表实现。但是，数组的能力有时候很有用，特别是当你需要存储固定数量的重复元素时，而且它非常容易使用。

建立数组的方法有两种：传统的PostgreSQL的方法和SQL99标准的方法。我们在这里会简单介绍这两种方法。

#### PostgreSQL样式的数组

要将一个表的列定义为数组，你可以简单地在类型后面添加[]；不需要定义元素的个数。如果你使用了大小来定义，PostgreSQL接受你的定义，但它不强制接受指定数量的元素。

##### 尝试：使用PostgreSQL语法定义数组

作为一个示例，假设我们决定要有一个雇员表，表中要包含一个指示他们工作日的指示器。通常，我们需要为每一天分配一个列，或者一个单独的表来储存工作日。在PostgreSQL中，我们可以简化这个工作，直接存储工作日的数组，就像这样：

test=> CREATE TABLE empworkday (

test(> refcode char(5),

test(> workdays int[]

test(> );

CREATE TABLE

test=>

这建立了一个有两个列的empworkday表：一个参考字符串以及一个叫做workdays的整数数组。要网数组列中插入值，我们需要用大括号分隔符包围由逗号分割的值的列表，就像这样：

test=> INSERT INTO empworkday VALUES('val01', '{0,1,0,1,1,1,1}');

INSERT 17892 1

test=> INSERT INTO empworkday VALUES('val02', '{0,1,1,1,1,0,1}');

INSERT 17893 1

test=>

我们可以一次性选择数组元素的所有值，就像这样：

test=> SELECT \* FROM empworkday;

refcode | workdays

---------+-----------------

val01 | {0,1,0,1,1,1,1}

val02 | {0,1,1,1,1,0,1}

(2 rows)

test=>

我们也可以通过给出数组的索引值来取出单个元素：

test=> SELECT workdays[2] FROM empworkday WHERE refcode = 'val02';

workdays

----------

1

(1 row)

test=>

##### 解析

PostgreSQL的行为很像传统的编程语言，存储一个数组的值，甚至还有不需要指出数组的大小的好处。如果你选择整个数组，PostgreSQL显示在花括号之间的所有的用逗号分隔的值。

有一个需要注意的事情是PostgreSQL的数组中第一个元素的索引值是1而不是0，而很多编程语言通常是0.如果你尝试选择一个不存在的数组元素，将返回NULL。

注：PostgreSQL也允许多维数组。要知道更多PostgreSQL关于数组的内容，请参考文档。

#### SQL99样式的数字

在SQL99标准中，新的数组定义语法被提出。这比PostgreSQL样式更明确的一种语法，必须指出元素的个数，而PostgreSQL在实现时是不强制执行这个标准的。

##### 尝试：使用SQL99语法的数组

让我们定义我们早前的表，使用SQL99样式的定义来做实验：

test=> DROP TABLE empworkday;

DROP TABLE

test=> CREATE TABLE empworkday (

test(> refcode char(5),

test(> workdays int array[7]

test(> );

CREATE TABLE

test=> INSERT INTO empworkday VALUES('val01', '{0,1,0,1,1,1,1}');

INSERT 17899 1

test=> INSERT INTO empworkday VALUES('val02', '{0,1,1,1,1,0,1}');

INSERT 17900 1

test=>

test=> SELECT \* FROM empworkday;

refcode | workdays

---------+-----------------

val01 | {0,1,0,1,1,1,1}

val02 | {0,1,1,1,1,0,1}

(2 rows)

test=>

test=> SELECT workdays[2] FROM empworkday WHERE refcode = 'val02';

workdays

----------

1

(1 row)

test=>

##### 解析

就像你看到的，SQL99样式的数组和PostgreSQL样式的数组基本一样。唯一的不同是定义的语法有点不同。

## 数据操作

PostgreSQL提供一些机制操作表中的数据。这里，我们将看到一些内置的函数和一些“神奇”变量。我们还会更仔细地查看PostgreSQL添加到表中的oid列。

### 在数据类型之间转换

我们经常需要在数据库中转换数据类型。类型转换可能很有用且有时候很必要，例如在处理日期和时间的时候。例如，我们可能需要处理从其他系统传入的以字符串方式进入数据库的日期值。转换这些字符串到日期数据类型将允许我们按日期查询，有时候将它们作为字符串处理无法满足我们的要求。

注：通常，你应该注意类型转换，因为程序中太多的类型转换可能意味着数据库设计中的缺陷。

关系数据库如何进行类型转换存在很大的差异。PostgreSQL使用cast转换符：

cast(column-name AS type-definition-to-convert-to)

另一种更简洁的双冒号语法可以用在SELECT语句中的简单的列名所在位置使用：

column-name::type-definition-to-convert-to

假设我们想要从我们原来的bpsimple数据库中的orderinfo表中以char(10)格式获取data数据，我们可以这样写：

SELECT cast(date\_placed AS char(10)) FROM orderinfo;

在我们的bpsimple数据库中执行它，我们可以得到：

bpsimple=> SELECT cast(date\_placed AS char(10)) FROM orderinfo;

date\_placed

-------------

2004-03-13

2004-06-23

2004-09-02

2004-09-03

2004-07-21

(5 rows)

bpsimple=>

我们可以用cast作为列值，而且我们可以通过提供列标题来给结果命名，就像我们将在下一个示例中一样。

##### 尝试：类型转换

假设我们想产生一个商品项的列表，以最接近的整数美元显示价格。我们可以简单地转换价格为integer类型。我们还将查询“原始”价格，来显示PostgreSQL已经对价格进行了四舍五入。

bpsimple=> SELECT sell\_price, sell\_price::int AS "Guide Price" FROM item

WHERE sell\_price > 5.0;

sell\_price | Guide Price

------------+-------------

21.95 | 22

9.95 | 10

15.75 | 16

19.95 | 20

25.32 | 25

11.49 | 11

(6 rows)

bpsimple=>

##### 解析

我们将sell\_price列转换为整数（sell\_price::int）并给它一个名字（AS "Guide Price"）。我们也可以使用case语法写出这个语句；这两种转换格式是可以互换的。

比较这两列，我们可以发现PostgreSQL如何进行四舍五入。在旧版本的PostgreSQL中，经常有必要进行数据类型之间的显式类型转换。通常，目前的版本会自动作出合理的转换。

注意并不是所有类型之间都可以进行转换。例如，你无法将date转换为一个integer。

### 用于数据操作的函数

PostgreSQL提供一些通用功能函数，你可以使用它们来操作列，它们被列在表8-7中。可以查看第10章获得更多关于PostgreSQL内建函数的信息。

表8-7 有用的数据操作函数

|  |  |
| --- | --- |
| 函数 | 描述 |
| length(column-name) | 返回一个字符串的长度 |
| trim(column-name) | 移除字符串开始和结尾的空格 |
| strpos(column-name, string) | 返回子串在列中的位置 |
| substr(column-name, position, length) | 根据指定位置和长度截取子串。第一个字符算作位置1 |
| round(column-name, length) | 根据指定小数点位置四舍五入一个数字 |
| abs(number) | 获得一个数字的绝对值 |

这些函数的用法和上小节讲解的cast的用法相同 。以下为一个使用substr和round函数的示例：

bpsimple=> SELECT substr(description, 3, 5), round(sell\_price, 1) FROM item;

substr | round

--------+-------

od Pu | 22.0

nux C | 2.5

ssues | 4.0

cture | 10.0

n Sma | 15.8

n Lar | 20.0

othbr | 1.5

man C | 2.5

rrier | 0.0

eaker | 25.3

bik C | 11.5

(11 rows)

bpsimple=>

### 魔法变量

有事，我们需要在数据库中存储一些关系到当前用户或者时间等等的信息，也许是为了实现一个审计线索。PostgreSQL提供一些“魔法”变量来完成这些。以下为其中一些最常用的：

* CURRENT\_DATE
* CURRENT\_TIME
* CURRENT\_TIMESTAMP
* CURRENT\_USER

你可以像使用列名一样使用它们，你甚至可以直接SELECT它们而不需要包含一个表名：

bpsimple=> SELECT item\_id, quantity, CURRENT\_TIMESTAMP FROM stock;

item\_id | quantity | timestamptz

---------+----------+-------------------------------

1 | 12 | 2004-10-19 18:03:14.500694+01

2 | 2 | 2004-10-19 18:03:14.500694+01

4 | 8 | 2004-10-19 18:03:14.500694+01

5 | 3 | 2004-10-19 18:03:14.500694+01

7 | 8 | 2004-10-19 18:03:14.500694+01

8 | 18 | 2004-10-19 18:03:14.500694+01

10 | 1 | 2004-10-19 18:03:14.500694+01

(7 rows)

bpsimple=> SELECT CURRENT\_USER, CURRENT\_TIME;

current\_user | timetz

--------------+--------------------

rick | 18:03:40.862712+01

(1 row)

bpsimple=>

这些魔法变量也可以用在INSERT和UPDATE语句中，就像以下示例：

INSERT INTO orderinfo(orderinfo\_id, customer\_id, date\_placed, date\_shipped,

shipping) VALUES (5, 8, CURRENT\_DATE, NULL, 0.0);

### OID列

你也许注意到了，每当我们插入数据，PostgreSQL使用一个随意的数字和一个行数作为响应。就像我们在第6章提到的，这个数字是一个内部参考数据，它是一个PostgreSQL为每一行保存的对象ID，通常为隐藏的列，名为oid。

大多数关系数据库没有这个列，或者即使它们这么做了，也是对用户无法访问的。在PostgreSQL中，我们可以通过显式地将它的名字放在查询表的SELECT语句中以查看它：

bpsimple=> SELECT oid, fname, lname FROM customer;

oid | fname | lname

-------+-----------+---------

19888 | Jenny | Stones

19889 | Andrew | Stones

19890 | Alex | Matthew

19891 | Adrian | Matthew

19892 | Simon | Cozens

19893 | Neil | Matthew

19894 | Richard | Stones

19895 | Ann | Stones

19896 | Christine | Hickman

19897 | Mike | Howard

19898 | Dave | Jones

19899 | Richard | Neill

19900 | Laura | Hardy

19901 | Bill | O'Neill

19902 | David | Hudson

(15 rows)

bpsimple=>

你的数据库中的oid列的值通常应该和以上是不同的。你也可以在ODBC驱动配置的时候看到关于OID的选项。你可以选择显式或者隐藏它。

可以通过设置postgresql.conf配置文件中的default\_with\_oids标志为false或者在建表的时候显式的指定WITHOUT OIDS来避免OID列被添加到你数据库的中用户表中。PostgreSQL 8.0默认是为用户表建立OID列。但是，以后发布的版本的PostgreSQL中可能默认不会建立OID列。因此，你永远都不应该依赖你数据库中的OID列，而且我们建议在你建表语句中避免使用WITH OIDS或者WITHOUT OIDS。

## 表管理

我们已经了解了PostgreSQL的数据类型，我们可以在我们建表的时候使用它们。我们在建立我们示例数据库的时候我们已经看到过CREATE TABLE这个SQL命令了，但我们将在这里将更正式地讲解它。我们还将探索一些扩展功能，例如临时表，在建立后改变表，以及在不需要的时候删除表。

### 建表

以下为建表的基本语法：

CREATE [TEMPORARY] TABLE table-name (  
 { column-name type [ column-constraint ] [,...] }  
 [ CONSTRAINT table-constraint ]  
) [ INHERITS (existing-table-name) ]

这看上去有点复杂，实际上它很简单。第一行通过使用CREATE TABLE简单说明你要建表，之后跟着的是表名和一个左括号。TEMPORARY允许你建立一个临时表，这将在本章较后的“使用临时表”小节讨论。

之后，你列出列名，它的类型以及一个可选的列约束。你基本上可以在你的表中使用无限多的列，每个列用逗号分隔。可选的列约束允许你指定这个列的扩展规则，而且你已经遇到过最常见的例子，NOT NULL。

在列的列表之后是一个表级的约束，它允许你写额外的表中的数据必须遵守的表级规则，例如一个列的值必须小于某个值。例如，星期几的列的值必须小于7。我们将在后面的小节讨论列和表的约束。

最后面是PostgreSQL的扩展，INHERITS，它允许一个建立一个新表，继承已经存在的表的列。新表除了包含指定的字段外，还包含在INHERITS关键字之后的表的所有字段。参考PostgreSQL文档获得更多关于使用INHERITS的信息。

提示：我们强烈建议你存储你建立数据库的命令到一个脚本中，并使用这个脚本来建立你的数据库。如果你需要修改数据库设计，修改脚本比重建数据库更轻松可靠。这样你就不需要尝试回想几个月（或者是几天）前你当初建立数据库的命令了。你会发现当初创建一个脚本并保持更新将为你带来数倍的回报。

### 使用列约束

经常需要对表中的列施加一些规则。我们已经有一些简单的例子，例如确保客户的姓不为空（NOT NULL）。有时候，我们需要施加一些规则用于管理数据的已知范围，例如确保保存付款率的列只允许大于一个最小值，或确保一个列是唯一的。对列使用约束允许我们在我们完整的应用的最低级别——数据库中——执行一些检查。

对于（hard-and-fast）基础规则，在数据库级别强制使用是一个很好的技术，因为它们不依赖于应用程序，所以任何应用程序的错误导致的进入的非法值将被数据库捕获。在建表时通过编写一个定义来施加规则比编写应用程序的逻辑代码来支持这些规则通常更容易。

表8-8显示了一些对你非常有用的最主要的约束。（也有一些更高级的约束，定义在PostgreSQL文档中。）我们在这里不会论述REFERENCES约束，而是在本章之后的“外键约束”小节中讨论。

表8-8 最主要的列约束

|  |  |
| --- | --- |
| 定义 | 意义 |
| NOT NULL | 列不允许存储NULL值 |
| UNIQUE | 列中存储的的值必须与其他行都不同。PostgreSQL允许你在定义为UNIQUE列上存储任意多个NULL值。 |
| PRIMARY KEY | 实际上是一个NOT NULL和UNIQUE的组合。每个表只能有一个列被标记为PRIMARY KEY（但你可以有多个列被同时标记为NOT NULL和UNIQUE）。如果你需要建立一个组合的主键（一个包含超过一个列的主键），你必须使用一个表级的约束，而不是列级的约束。 |
| DEFAULT default-value | 允许你在插入数据的时候提供一个默认值。（严格来说，这不是一个约束选项，但把它作为约束来考虑更容易理解。） |
| CHECK (condition) | 当插入或者更新数据的时候允许你进行一个条件检查。 |
| REFERENCES | 约束这个值必须为另一个独立的表的某个列中的某个值。 |

除了PRIMARY KEY，你可以在任何列中使用任何数量的约束。允许对列级约束命名，但当通常没人去做。

一个需要特备注意的是当一个NULL值添加到一个使用了UNIQUE约束的列中将发生的事情。PostgreSQL认为每个NULL都是唯一的，所以它允许你在一个定义为UNIQUE的列中拥有很多值为NULL的行。按照SQL标准，应该只允许一个NULL，所以这是一个稍微背离标准的地方。从论据上看，SQL标准更符合逻辑，因为如果NULL为未知，也就没有办法知道他们是否不同，但PostgreSQL的实现可能更现实。

##### 尝试：对列施加约束

最简单的理解列约束的方法是看它们的行为。让我们在我们之前建立的test数据库建立一个新表，并使用它来试验一些约束：

bpsimple=> \c test

You are now connected to database "test".

test=> CREATE TABLE testcolcons (

test(> colnotnull INT NOT NULL,

test(> colunique INT UNIQUE,

test(> colprikey INT PRIMARY KEY,

test(> coldefault INT DEFAULT 42,

test(> colcheck INT CHECK( colcheck < 42)

test(> );

NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index

"testcolcons\_pkey" for table "testcolcons"

NOTICE: CREATE TABLE / UNIQUE will create implicit index

"testcolcons\_colunique\_key" for table "testcolcons"

CREATE TABLE

test=>

你会发现PostgreSQL警告我们它已经建立了一些索引来执行PRIMARY KEY和UNIQUE约束。它还为我们生成了有意义的名字。

现在我们已经建立了一个在列中拥有数种约束的表，我们可以尝试插入一些数据并查看约束在实际中如何工作：

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey, coldefault,

test(> colcheck) VALUES(1,1,1,1,1);

INSERT 17497 1

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

test(> oldefault, colcheck) VALUES(2,2,2,2,2);

INSERT 17498 1

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

test(> coldefault, colcheck) VALUES(2,2,2,2,2);

ERROR: duplicate key violates unique constraint "testcolcons\_pkey"

test=>

插入失败了，因为索引testcolcons\_pkey发现了重复的值。这里我们需要用到一个常识，叫做testcolcons\_pkey的索引表示testcolcons的主键索引。

每个表只允许有一个主键；因此，对于叫做tablename\_pkey的索引不会有歧义。

但是，PostgreSQL允许我们插入两行NULL值到colunique列中（这看上去有点危险）：

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

coldefault, colcheck) VALUES(1,NULL,98,1,1);

INSERT 17503 1

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

coldefault, colcheck) VALUES(1,NULL,99,1,1);

INSERT 17504 1

test=>

如果我们使用实际的值，PostgreSQL将拒绝这条INSERT语句：

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

test(> coldefault, colcheck) VALUES(2,2,9,2,2);

ERROR: Cannot insert a duplicate key into unique index testcolcons\_colunique\_key

test=>

这时候，INSERT失败，因为索引testcolcons\_colunique\_key发现一个重复值。我们可以有很多定义为UNIQUE的列，所以PostgreSQL命名这些索引为tablename\_columnname\_key，这可以很清楚地显示出哪个列导致了这个问题：

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey,

test(> coldefault, colcheck) VALUES(3,3,3,3,100);

ERROR: new row for relation "testcolcons" violates check constraint

"testcolcons\_colcheck\_check"

test=>

这次，问题出在CHECK约束的colcheck列，因为我们尝试插入一个大于42的值。注意这个约束命名为tablename\_columnname\_check，所以问题源很容易定位：

test=> UPDATE testcolcons SET colunique = 1 WHERE colnotnull = 2;

ERROR: duplicate key violates unique constraint

"testcolcons\_colunique\_key"testcolcons\_colunique\_key

test=>

我们无法更新colunique的值，因为表中有一行的这个列有了这个值：

test=> INSERT INTO testcolcons(colnotnull, colunique, colprikey, colcheck)

test-> VALUES(3,3,3,41);

INSERT 17505 1

test=> SELECT \* FROM testcolcons;

colnotnull | colunique | colprikey | coldefault | colcheck

------------+-----------+-----------+------------+----------

1 | 1 | 1 | 1 | 1

2 | 2 | 2 | 2 | 2

1 | | 98 | 1 | 1

1 | | 99 | 1 | 1

3 | 3 | 3 | 42 | 41

(5 rows)

test=>

最后，我们没有为coldefault列提供一个值（注意它没有在列的列表中给出），我们会发现默认值被使用了。

如果我们想查看一个表上的约束，我们需要通知psql来列出它们，使用\d tablename命令，就像这样：

test=> \d testcolcons

Table "public.testcolcons"

Column | Type | Modifiers

------------+---------+------------

colnotnull | integer | not null

colunique | integer |

colprikey | integer | not null

coldefault | integer | default 42

colcheck | integer |

Indexes:

"testcolcons\_pkey" PRIMARY KEY, btree (colprikey)

"testcolcons\_colunique\_key" UNIQUE, btree (colunique)

Check constraints:

"testcolcons\_colcheck\_check" CHECK (colcheck < 42)

test=>

##### 解析

PostgreSQL使用多种方法实现约束。但无法控制约束检查的顺序。你获得的实际错误依赖于PostgreSQL的内部实现。你确定可以知道的是所有的约束都将在数据存储到数据库中之前得到检查。你也可以第9章讲解的使用事务来确保对数据库改变的请求集同时全部执行或者同时没有被执行。

### 使用表约束

表约束和列约束非常相似，区别就像它的名字描述的一样，将应用到表上，而不是一个独立的列中。有时候，我们需要指定特殊的约束，例如表级的主键而不是列级的。例如，我们看看我们的orderline表，我们需要使用两个列orderinfo\_id和item\_id一起作为一个键来标记一行，因为只有这几个列的组合才是唯一的。这种类型的约束必须在表级别描述。

四个表级的约束列在表8-9中。

表8-9 最主要的表约束

|  |  |
| --- | --- |
| 定义 | 意义 |
| UNIQUE(column-list) | 列的列表中存储的的值必须与其他行都不同。 |
| PRIMARY KEY | 实际上是一个NOT NULL和UNIQUE的组合。每个表只能有一个列被标记为PRIMARY KEY，无论是表约束还是列约束。 |
| CHECK (condition) | 当插入或者更新数据的时候允许你进行一个条件检查。 |
| REFERENCES | 约束这个值必须为另一个独立的表的某个列中的某个值。 |

就像你看到的，表级的约束和列级约束很相似。它们的区别为：

* 表级的约束可以针对不止一个列。
* 表级的约束在所有的列之后列出来。

##### 尝试：使用表级约束

让我们看看表级约束如何工作。首先，建立一个有一些约束的表：

test=> CREATE TABLE ttconst (

test(> mykey1 int,

test(> mykey2 int,

test(> mystring varchar(15),

test(> CONSTRAINT cs1 CHECK (mystring <> ''),

test(> CONSTRAINT cs2 PRIMARY KEY(mykey1, mykey2)

test(> );

NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "cs2"

for table "ttconst"

CREATE TABLE

test=>

注意，由于使用了列级约束，PostgreSQL建立了一个索引来执行主键约束。让我们从插入一些行开始：

test=> INSERT INTO ttconst VALUES(1,1,'Hello');

INSERT 19381 1

test=> INSERT INTO ttconst VALUES(1,2,'Bye');

INSERT 19382 1

test=>

现在尝试插入一行违反mystring不能为空串规则的数据：

test=> INSERT INTO ttconst VALUES(1,2,'');

ERROR: new row for relation "ttconst" violates check constraint "cs1"

test=>

表级CHECK约束和列级约束基本上完全一样，拒绝了这一行，因为字符串为空。

现在如果我们尝试插入一行违反mykey1和mykey2组合必须为唯一的约束，我们将看到约束cs2被执行：

test=> INSERT INTO ttconst VALUES(2,2,'Chow');

INSERT 19383 1

test=> INSERT INTO ttconst VALUES(2,2,'Chow');

ERROR: duplicate key violates unique constraint "cs2"

test=>

当两个mykey的值读相同的时候，这一行被拒绝了，因为违反了主键约束。

##### 解析

就像你看到的，表及约束和列级约束非常相似。通常，如果列级约束够用，建议使用列级约束。然而，当我们需要在同一个表中同时使用列级和表级约束的时候，例如在我们的bpsimple数据库中，为了一致性，我们建议在所有的表中使用表级主键约束。

### 修改表结构

不幸的是，生活很复杂，无论你如何仔细收集需求并实现你的数据库，需要修改表设计的那一天总会到来。

在第6章我们找到了一个解决这个问题的办法，针对从已有的表中查询出来的数据使用INSERT INTO。我们可以使用以下的步骤：

* 建立一个和存在的表相同的新表。
* 使用INSERT INTO来从原始表中生成相同的数据到工作表。
* 删除原始表。
* 使用旧名称建立原来的表，但增加我们需要的改变。
* 再次使用INSERT INTO从功过表中生成数据到修改后的表。
* 删除工作表。

很明显，工作量很大，尤其是如果表包含大量的数据并且被触发器或者视图引用，而我们想要做的只是为表增加一个列。由于PostgreSQL遵从SQL标准，因此允许我们随时添加、删除和重命名列；也就是说，即使表中包含数据，你也可以重命名这个表。

也可以对表添加或者删除约束以及修改默认值；然而，由于一些特殊原因，这种操作有一些限制。例如，你不可以添加一个约束到包含违反这个约束的数据的表中。

注：在旧版本的PostgreSQL中有一个额外的限制，你不能建立一个拥有NOT NULL或者DEFAULT设置的新列，因为这个表中的数据已经存在了。如果确实有必要，做起来也不是很麻烦：添加这个列而不包含任何约束，更新表中的数据，按后添加需要的约束。从PostgreSQL 8.0开始，你可以添加拥有默认值的列，同时包含NOT NULL约束，只要你提供一个默认值。

要实现这些改变，我们使用ALTER TABLE命令。ALTER TABLE的语法非常简单，但有很多变种：

ALTER TABLE table-name ADD COLUMN column-name column-type  
ALTER TABLE table-name DROP COLUMN column-name  
ALTER TABLE table-name RENAME COLUMN old-column-name TO new-column-name  
ALTER TABLE table-name column-name TYPE new-type [ USING expression ]  
ALTER TABLE table-name ALTER COLUMN [SET DEFAULT value | DROP DEFAULT]  
ALTER TABLE table-name ALTER COLUMN [SET NOT NULL | DROP NOT NULL]  
ALTER TABLE table-name ADD CHECK check-expression  
ALTER TABLE table-name ADD CONSTRAINT name constraint-definition  
ALTER TABLE old-table-name RENAME TO new-table-name

添加到有数据的表中的列将为已有的行存储NULL值到这个列中。

##### 尝试：修改一个表

在我们看到一些ALTER TABLE语句前，让我们先检查下已有的ttconst表的结构：

test=> \d ttconst

Table "public.ttconst"

Column | Type | Modifiers

----------+-----------------------+-----------

mykey1 | integer | not null

mykey2 | integer | not null

mystring | character varying(15) |

Indexes:

"cs2" PRIMARY KEY, btree (mykey1, mykey2)

Check constraints:

"cs1" CHECK (mystring::text <> ''::text)

test=>

首先，我们添加一个新列：

test=> ALTER TABLE ttconst ADD COLUMN mydate DATE;

然后，我们重命名新添加的列：

test=> ALTER TABLE ttconst RENAME COLUMN mydate TO birthdate;

ALTER TABLE

test=> \d ttconst

Table "public.ttconst"

Column | Type | Modifiers

-----------+-----------------------+-----------

mykey1 | integer | not null

mykey2 | integer | not null

mystring | character varying(15) |

birthdate | date |

Indexes:

"cs2" PRIMARY KEY, btree (mykey1, mykey2)

Check constraints:

"cs1" CHECK (mystring::text <> ''::text)

test=>

现在，让我们尝试改变一些约束和其他规则：

test=> ALTER TABLE ttconst DROP CONSTRAINT cs1;

ALTER TABLE

test=> ALTER TABLE ttconst ADD CONSTRAINT cs3 UNIQUE(birthdate);

NOTICE: ALTER TABLE / ADD UNIQUE will create implicit index "cs3" for table "tt

const"

ALTER TABLE

test=> ALTER TABLE ttconst ALTER COLUMN mystring SET DEFAULT 'Hello';

ALTER TABLE

下一步，让我们看看最新的表定义：

test=> \d ttconst

Table "public.ttconst"

Column | Type | Modifiers

-----------+-----------------------+------------------------------------

mykey1 | integer | not null

mykey2 | integer | not null

mystring | character varying(15) | default 'Hello'::character varying

birthdate | date |

Indexes:

"cs2" PRIMARY KEY, btree (mykey1, mykey2)

"cs3" UNIQUE, btree (birthdate)

test=>

就像你看到的，新的规则正确被设置，这和我们在建表的时候就设置它们的效果一样。

也可以改变列的类型，条件是转换符合逻辑。这里，我们转换一个date到varchar：

test=> ALTER TABLE ttconst ALTER birthdate TYPE varchar(32);

ALTER TABLE

test=>

更常用的用法是修改列的大小；例如，增加一个varchar类型列的大小：

test=> ALTER TABLE ttconst ALTER mystring TYPE varchar(32);

ALTER TABLE

test=>

最后，我们重命名这个表：

test=> ALTER TABLE ttconst RENAME TO ttconst2;

ALTER TABLE

test=>

##### 解析

就像你看到的，ALTER TABLE是一个非常强大的命令，允许你修改存在的表结构，包括列和约束，甚至在它们已经包含数据的时候。

注：低于PostgreSQL 8.0的ALTER TABLE命令有更多限制，所以如果你在运行在一个很旧版本中，你可能发现一些选项不存在。

改变表结构的能力不应该被用来作为在初期表设计时缺乏对细节关注的借口。ALTER TABLE应该通常用于需求变更。在其他时候使用它通常暗示着你的原设计在你实际在数据库中建表前可能可以被改良。

你应该很警惕通过添加新列不断改变一个表的结构。新的列通常添加在表的末尾，所以很多不能很好的反应表的逻辑目的。假设我们在建立我们的customer表时忘记了一个title列，然后使用ALTER TABLE添加它。列会添加在末尾，这使customer表看上去有些古怪，因为一个人的称谓被放在电话号码之后。取而代之，你可能更愿意使用以下的过程来添加列：

* 使用临时的表名新建一个表，按最符合逻辑的顺序安排列名。
* 使用 INSERT INTO … SELECT …来复制需要改变的表的数据
* 删除旧表。
* 使用旧表的名字重命名新表。

你需要特别注意我们将在第10章讨论的序列生成器和触发器，在删除表和重命名表的时候，他们也需要被删除并重新建立。

### 删除表

删除表非常简单：

DROP TABLE table-name

一瞬间，你的表就消失了，当然也包括其中的数据。当然，你要非常小心使用这个命令。

### 使用临时表

到目前为止我们看到的所有的SQL示例都是通过一个简单的，偶尔有点复杂的SQL语句达到我们的目标的。通常，这是一个好习惯，因为就像我们所说的，SQL是一个说明性语言。如果你定义了你想要的，SQL找到最好的方法来为你获取结果。但是，有时候，做任何事情都用一条SQL语句是不可行的或者非常复杂的。在一些情况下，你需要保存临时结果。

通常，你需要的临时存储就是一个表，这样你就可以存储很多行数据了。当然，你可以总是建立一个表，处理你的业务，然后删除这个表，但有可能产生中间表偶尔没有被删除的风险，可能是应为你的应用程序有一个错误或者仅仅是因为一个直接针对数据库操作的交互式用户忘记了。最终的结果是流浪表，一些无用的有奇怪名字的表，遗留在你的数据库中。不幸的是，很难弄清楚那些表是用于中间工作的表，可以被删除，哪些是长期使用的表。

SQL提供一个非常简单的方法来解决这个问题：临时表。当你建表时，你可以使用CREATE TEMPORARY TABLE（你也可以使用同义词CREATE TEMP TABLE）而不是CREATE TABLE。这个表使用通常的方法为你创建，除了当你的会话结束，你到数据库的连接断开后，临时表会自动删除。

## 视图

当你有一个复杂的数据库，或者有时候当你有大量拥有不同许可的用户，你需要建立表的幻象，也就是视图。让我们通过一个示例弄清这个概念。

假设我们需要允许别人访问我们数据仓库，在我们的数据库中查找条码和对应的商品。当前，它们被拆分成两个表，item和barcode。从视图的观点上看，我们需要一个简单的窗口让别人访问数据，也许是通过第5章中的一些GUI工具。我们可以通过使用视图做到，而不是改变我们的设计。

### 建立视图

建立视图的语法非常简单：

CREATE VIEW name-of-view AS select-statement;

之后你可以像查询表一样查询这个视图。（在写本文的时候，在PostgreSQL中，默认情况下视图还是只读的。）你可以从视图中查询数据，就像从表中查询一样，而且可以连接到其他表中，也可以使用WHERE从句。每次你在视图中执行SELECT，数据都会重建，所以数据总是最新的。它不是一个在视图被建立的时候的冻结的拷贝。

注：在一些其他的数据库中，视图以及相关表中的数据可以被更新，就像表一样。

假设我们想建立一个视图显示item表的简化信息。我们只需要得到item\_id，description和sell\_price。对于的SELECT语句如下：

SELECT item\_id, description, sell\_price FROM item;

例如我们想要建立这个名为item\_price的视图，我们应该这么写：

CREATE VIEW item\_price AS SELECT item\_id, description, sell\_price FROM item;

之后，item\_price就能够像一个表一样用在一个SELECT语句中。

##### 尝试：建立一个视图

回到第5章，我们在定义item表时在定义价格的时候有点小麻烦。假设我们考虑将price定义为numeric(7,2)是正确的，我们仍然可以保持这个定义，但是可以通过使用一个带有cast的SELECT语句的视图达到目标。

让我们建立一个item表的视图，通过以下三项修改用户所见的内容：

* 我们需要隐藏cost\_price。
* 我们只需要提供简单的物品描述。
* 我们需要隐藏所有的昂贵的物品，例如那些价格在$20以上的东西。

我们可以简单地通过建立一个视图做到，就像这样：

bpsimple=> CREATE VIEW item\_price AS SELECT item\_id, description::varchar(10),

bpsimple-> sell\_price AS price FROM item WHERE sell\_price <= 20.0;

CREATE VIEW

bpsimple=>

现在，当我们从这个视图中SELECT数据，它的行为就像原来表的列的子集一样：

bpsimple=> SELECT \* FROM item\_price;

item\_id | description | price

---------+-------------+-------

3 | Linux CD | 2.49

4 | Tissues | 3.99

5 | Picture Fr | 9.95

6 | Fan Small | 15.75

7 | Fan Large | 19.95

8 | Toothbrush | 1.45

9 | Roman Coin | 2.45

10 | Carrier Ba | 0.00

2 | Rubik Cube | 11.49

(9 rows)

bpsimple=>

##### 解析

在我们的示例中，我们做了很多事。首先，我们通过转换description列为description::varchar(10)缩减它到10字节。下一步，我们通过不包含它在列的列表中，隐藏了成本价，同时耍了点小手段，将售价重命名为价格sell\_price AS price以使这里没有线索提示这个表里还包含了成本价。最后，我们约束视图返回的行为WHERE sell\_price <= 20.0。（在第11章，我们将讨论如何使用许可来使普通用户无法访问原来的item表。）

我们不限制在一个视图中只使用一个表。如果我们愿意，我们可以使用复杂的SQL语句来访问大量的表。

##### 尝试：从多个表创建一个视图

让我们建立一个视图来解决我们用简单的方法显示item和barcode表，隐藏价格信息并解决数据分布在两个表中的问题。我们将命名这个视图为all\_items：

bpsimple=> CREATE VIEW all\_items AS SELECT i.item\_id, i.description, b.barcode\_ean

bpsimple-> FROM item i, barcode b WHERE i.item\_id = b.item\_id;

CREATE VIEW

bpsimple=>

这建立了一个新的视图，我们可以像使用表一样用它：

bpsimple=> SELECT \* FROM all\_items;

item\_id | description | barcode\_ean

---------+---------------+---------------

1 | Wood Puzzle | 6241527836173

2 | Rubik Cube | 6241574635234

3 | Linux CD | 6264537836173

3 | Linux CD | 6241527746363

4 | Tissues | 7465743843764

5 | Picture Frame | 3453458677628

6 | Fan Small | 6434564564544

7 | Fan Large | 8476736836876

8 | Toothbrush | 6241234586487

8 | Toothbrush | 9473625532534

8 | Toothbrush | 9473627464543

9 | Roman Coin | 4587263646878

11 | Speakers | 9879879837489

11 | Speakers | 2239872376872

(14 rows)

bpsimple=>

注意，这和我们敲以下命令完全一样：

SELECT i.item\_id, i.description, b.barcode\_ean FROM item i, barcode b

WHERE i.item\_id = b.item\_id;

就像你看到的，它对最终用户隐藏了复杂性。

如果我们想列出我们数据库中的视图，我们可以使用\dv命令。命令\d name-of-view将描述视图，运行我们看到使用的SQL：

bpsimple=> \dv

List of relations

Schema | Name | Type | Owner

--------+------------+------+-------

public | all\_items | view | rick

public | item\_price | view | rick

(2 rows)

bpsimple=> \d all\_items

View "public.all\_items"

Column | Type | Modifiers

-------------+-----------------------+-----------

item\_id | integer |

description | character varying(64) |

barcode\_ean | character(13) |

View definition:

SELECT i.item\_id, i.description, b.barcode\_ean

FROM item i, barcode b

WHERE i.item\_id = b.item\_id;

bpsimple=>

我们建立了一个叫all\_items的视图，它的行为像一个表，除了它是通过一些隐藏的SQL建立的。

一些人趋向于认为视图是一个好东西，因而所有的表都应该隐藏在视图之后。虽然某些级别的数据隐藏通常很有用，但使用视图没有直接使用表有效，特别是如果定义视图的SQL非常复杂且使用超过一个表。将所有的表隐藏在视图之后的数据库的性能非常糟糕，而且用户无法优化他们的SQL的性能，也许是因为他们需要的列是在一个在一个视图中而导致了一个大表的关联。甚至也许用户只需要一个列，如果你强制他们使用视图，他们将执行视图之后复杂的SQL，降低性能。虽然视图对你有用，太多好东西可能反而是有害的。

### 删除和替换视图

要删除一个视图，如下操作：

DROP VIEW name-of-view

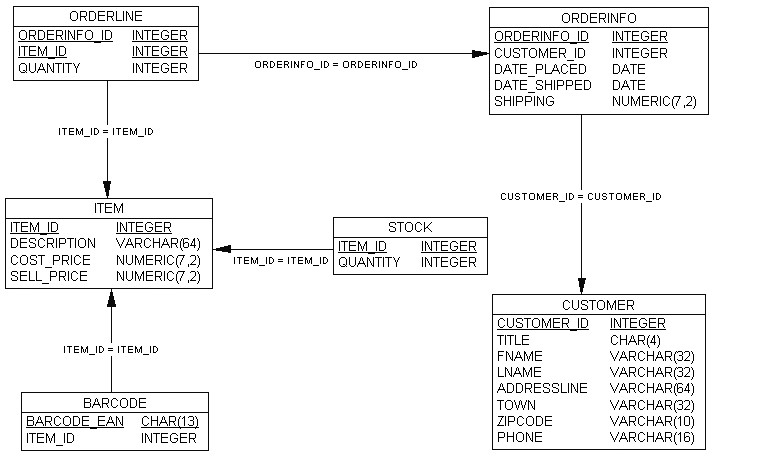
不像删除表，删除视图不会影响相关的数据。

如果你想要使用相同的名字和返回列替换一个现存的视图，你可以在一条语句中使用一个特别版本的语法来完成：

CREATE OR REPLACE VIEW name-of-view AS select-statement

## 外键约束

我们现在开始讲解一个最重要的类型的约束，叫做外键约束。在第2章，当我们画我们的bpsimple数据库的图解的时候，我们有一些表连接或者关联到其他表。图8-1显示这个数据库模式的设计：

  
图8-1数据库模式设计

你可以发现一个表中的列如何关联到另一个表中的列。例如，orderinfo表中的customer\_id关联到customer表中的customer\_id。所以，给出orderinfo\_id，我们可以使用同一行中的customer\_id找到订单相关的客户的名字和地址。我们已经学到了customer\_id为customer表的主键；用它可以对customer表中的行进行唯一标示。

这里有另一个重要的术语：orderinfo表中的customer\_id就是一个外键。这意味着虽然orderinfo表中的customer\_id不是这个表中的主键，它关联到的customer表中的列是customer表中的唯一键。注意这里没有反转的关系——customer表中没有哪个列是其他表中的唯一键。因此，我们可以认为customer表没有外键。

作为我们的数据库设计的内在结构的一部分，我们的目标是每个orderinfo表的customer\_id也必须出现在customer表中。在orderinfo表中的customer\_id叫做外键。我们想在数据库中执行这两个表之间关系的规则检查，因为它比在应用程序中进行数据完整性检查更有效，这就叫做外键约束。

当我们建立一个外键约束的时候，PostgreSQL将检查在特定表中的那个列被定义为唯一的。通常被外键引用的是其他表的主键。使用外键约束是一个很好的方法来确保表间的关系不会由于删除特定表的（被引用为另一个表的外键的）主键而被破坏。

一个表可以拥有多个外键。例如，在orderline表中，orderinfo\_id是一个外键，因为它连接到orderinfo表的orderinfo\_id这个主键，item\_id也是一个主键，因为它连接到item表的主键item\_id上。

在item表中，item\_id是item表的主键，因为它唯一标识了一行，它也是stock表的外键。一个列可以同时是主键和外键，且这暗指（通常是非强制性的）两个表的行的一对一的关系。

虽然在我们的示例数据库中没有任何数据库，但实际中也可以允许一对列组合为外键，就像orderinfo\_id和item\_id组合起来作为orderline表的主键一样。

这些关系是我们数据库中绝对的关键。如果我们有一行在orderinfo表中的数据无法在customer表中找到对应的customer\_id，我们就有一个重要的数据库整性问题。我们有一个无法找到下订单用户的订单。虽然我们可以使用应用程序的逻辑来确保我们的关系规则，然而就像我们之前说到的，这会更加安全，且通常更容易做到，只需要定义它们为数据库规则就可以做到。

当发现可以就像我们之前定义约束一样在列和表中定义这些外键关系，你不会觉得奇怪。这些通常在表建立的时候被做到，作为CREATE TABLE命令的一部分，使用REFERENCES类型的约束。也可以在建表之后再添加外键约束，使用ALTER TABLE table-name ADD CONSTRAINT name constraint-efinition语法。

我们将前往我们的bpsimple数据库，并建立一个bpfinal数据库，它实现了外键约束，来确保数据一致性。

### 作为一个列的约束的外键

这里是用于定义一个列为另一个表的外键的基本语法：

[CONSTRAINT arbitrary-name] existing-column-name type REFERENCES  
foreign-table-name(column-in-foreign-table)

为约束命名是可选的，但就像我们之后将看到的，它对理解出错信息非常有用。

要在orderinfo表的customer\_id列中定义一个外键约束，关联它到customer表，我们一桶使用REFERENCES关键字和外部表名和列，就像这样：

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL REFERENCES customer(customer\_id),

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id)

);

我们将很快看到REFERENCE约束的效果。

### 作为一个表的约束的外键

虽然你可以在列级定义外键约束，我们推荐在表级与主键约束一起定义它们。当多个列参与了这种关系的时候，你无法使用列级约束，所以在这种情况下，你必须使用表级约束。

提示：最好是使用表级约束，而不要混合私用表级和列级约束。

表级约束和列级约束的格式非常相似，只是它位于所有的列之后：

CONSTRAINT [arbitrary-name] FOREIGN KEY (column-list) REFERENCES  
foreign-table-name(column-list-in-foreign-table)

我们可以更新我们orderinfo表的定义来声明一个列customer\_id为一个外键的约束，因为它关联到customer表的主键列customer\_id。

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL,

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id),

CONSTRAINT orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id) REFERENCES

customer(customer\_id)

);

#### 为已有的表添加外键约束

在从头建表开始，让我们先简短地回顾下ALTER TABLE 命令，并查看我们如何使用它追加一个外键约束。

让我们先看看已有的表：

bpsimple=> \d orderinfo

Table "public.orderinfo"

Column | Type | Modifiers

--------------+--------------+--------------------------------------------

orderinfo\_id | integer | not null default nextval

('public.orderinfo\_orderinfo\_id\_seq'::text)

customer\_id | integer | not null

date\_placed | date | not null

date\_shipped | date |

shipping | numeric(7,2) |

Indexes:

"orderinfo\_pk" PRIMARY KEY, btree (orderinfo\_id)

bpsimple=>

然后我们改变这个表，添加一个新的外键约束：

bpsimple=> ALTER TABLE orderinfo ADD CONSTRAINT

orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id)

REFERENCES customer(customer\_id);

ALTER TABLE

bpsimple=>

让我们检查表是否被正确更新：

bpsimple=> \d orderinfo

Table "public.orderinfo"

Column | Type | Modifiers

--------------+--------------+--------------------------------------------

orderinfo\_id | integer | not null default nextval

('public.orderinfo\_orderinfo\_id\_seq'::text)

customer\_id | integer | not null

date\_placed | date | not null

date\_shipped | date |

shipping | numeric(7,2) |

Indexes:

"orderinfo\_pk" PRIMARY KEY, btree (orderinfo\_id)

Foreign-key constraints:

"orderinfo\_customer\_id\_fk" FOREIGN KEY (customer\_id) REFERENCES

customer(customer\_id)

bpsimple=>

看上去有点复杂，但我们可以明确地在末尾看到我们的新外键约束。

#### 建表时添加外键约束

是时候为我们的最初的bpsimple数据库带来飞跃，即开始设计我们最终版本的bpfinal数据库：

bpsimple=> CREATE DATABASE bpfinal;

CREATE DATABASE

bpsimple=> \c bpfinal

You are now connected to database "bpfinal".

bpfinal=>

现在我们已经准备好使用我们的新知识重建我们的表，使用外键约束来从数据库级别保证参照完整性。

我们必须从我们的customer表（相对于我们之前的设计不变）开始，因为如果它不存在，我们无法在orderinfo表中引用它。

bpfinal=> CREATE TABLE customer

bpfinal-> (

bpfinal(> customer\_id serial,

bpfinal(> title char(4),

bpfinal(> fname varchar(32),

bpfinal(> lname varchar(32) NOT NULL,

bpfinal(> addressline varchar(64),

bpfinal(> town varchar(32),

bpfinal(> zipcode char(10) NOT NULL,

bpfinal(> phone varchar(16),

bpfinal(> CONSTRAINT customer\_pk PRIMARY KEY

(customer\_id)

bpfinal(> );

NOTICE: CREATE TABLE will create implicit sequence "customer\_customer\_id\_seq"

for serial column "customer.customer\_id"

NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "customer\_pk"

for table "customer"

CREATE TABLE

bpfinal=>

现在我们已经有了customer表，我们也可以像以前一样使用\i命令来生成它：

bpfinal=> \i pop\_customer.sql

之后，我们可以建立我们的orderinfo表：

bpfinal=> CREATE TABLE orderinfo

bpfinal-> (

bpfinal(> orderinfo\_id serial,

bpfinal(> customer\_id integer NOT NULL,

bpfinal(> date\_placed date NOT NULL,

bpfinal(> date\_shipped date,

bpfinal(> shipping numeric(7,2) ,

bpfinal(> CONSTRAINT orderinfo\_pk PRIMARY KEY

(orderinfo\_id),

bpfinal(> CONSTRAINT orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id)

REFERENCES customer(customer\_id)

bpfinal(> );

NOTICE: CREATE TABLE will create implicit sequence "orderinfo\_orderinfo\_id\_seq"

for serial column "orderinfo.orderinfo\_id"

NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "orderinfo\_pk"

for table "orderinfo"

CREATE TABLE

bpfinal=>

让我们迅速检查下定义：

bpfinal=> \d orderinfo

Table "public.orderinfo"

Column | Type | Modifiers

--------------+--------------+--------------------------------------------------

orderinfo\_id | integer | not null default nextval('public.orderinfo\_orderi

nfo\_id\_seq'::text)

customer\_id | integer | not null

date\_placed | date | not null

date\_shipped | date |

shipping | numeric(7,2) |

Indexes:

"orderinfo\_pk" PRIMARY KEY, btree (orderinfo\_id)

Foreign-key constraints:

"orderinfo\_customer\_id\_fk" FOREIGN KEY (customer\_id) REFERENCES customer(cus

tomer\_id)

bpfinal=>

现在我们可以使用SQL脚本重建orderinfo表：

bpfinal=> \i pop\_orderinfo.sql

现在我们基本上回到了开始的地方，但有一个非常大的不同：orderinfo表有一个外键约束，用以说明orderinfo表中的行的customer\_id列引用了customer表中的customer\_id列。这意味着如果customer表中的行被orderinfo表中的数据引用了，则无法删除它。

##### 尝试：使用外键约束

我们将从检查orderinfo表中的customer\_id的数据开始：

bpfinal=> select orderinfo\_id, customer\_id from orderinfo;

orderinfo\_id | customer\_id

--------------+-------------

1 | 3

2 | 8

3 | 15

4 | 13

5 | 8

(5 rows)

bpfinal=>

我们现在知道在orderinfo表中有五行记录引用了customer表中的客户信息的customer\_id值，引用的客户的编号为3，8，13和15。只有四个客户被引用了，因为orderinfo\_id为2和5的行指向了同一个客户。

让我们从customer表中删除customer\_id为3的行：

bpfinal=> DELETE FROM customer WHERE customer\_id = 3;

ERROR: update or delete on "customer" violates foreign key constraint

"orderinfo\_customer\_id\_fk" on "orderinfo"

DETAIL: Key (customer\_id)=(3) is still referenced from table "orderinfo".

bpfinal=>

PostgreSQL阻止我们删除这一行。而且，请注意约束名orderinfo\_customer\_id\_fk允许我们更容易定位错误源。PostgreSQL甚至耐烦的告诉我们哪一个customer\_id键的值引起了这个问题，这虽然在本例中非常明显，但在其他情况下可能更复杂。PostgreSQL将允许我们删除不被orderinfo表的条目引用的行：

bpfinal=> DELETE FROM customer WHERE customer\_id = 4;

DELETE 1

bpfinal=>

##### 解析

在后台，PostgreSQL添加了一些额外的检查。我们尝试删除从customer表中的每一行时，数据库会检查这一行是否被另一个表引用——在本例中为orderinfo表。

命令中任何违反规则的尝试将被拒绝且数据不会改变。我们仍然可以删除一个用户，但我们必须首先确保这个客户没有任何订单。

PostgreSQL也会检查我们是否会插入一行引用了不存在的客户的行到orderinfo表中，如本例所示：

bpfinal=> INSERT INTO orderinfo(customer\_id, date\_placed, shipping)

bpfinal-> VALUES(250,'07-25-2000', 0.00);

ERROR: insert or update on table "orderinfo" violates foreign key constraint

"orderinfo\_customer\_id\_fk"

DETAIL: Key (customer\_id)=(250) is not present in table "customer".

bpfinal=>

很容易意识到我们在这里跨出了一大步。我们使用了非常有效的步骤通过数据库非常确保了表与表之间的关系。再也不存在orderinfo表中的行指向了不存在的客户的情况。

我们现在可以更新我们原来的建表脚本，对引用其他表的表添加外键约束：orderinfo，orderlines，stock和barcode表。唯一有点复杂的表是orderline表，它的orderinfo\_id列引用了orderinfo表，而intem\_id列引用了item表。但这不是问题，我们只需要简单地指定两个约束，每个列一个：

CREATE TABLE orderline

(

orderinfo\_id integer NOT NULL,

item\_id integer NOT NULL,

quantity integer NOT NULL,

CONSTRAINT orderline\_pk PRIMARY KEY(orderinfo\_id,

item\_id),

CONSTRAINT orderline\_orderinfo\_id\_fk FOREIGN KEY(orderinfo\_id) REFERENCES

orderinfo(orderinfo\_id),

CONSTRAINT orderline\_item\_id\_fk FOREIGN KEY(item\_id) REFERENCES item(item\_id)

);

在附件E中有我们最终版本的建库脚本。我们将在本书余下的buffer使用这个叫bpfinal的数据库。你也可以在<http://www.xxx.com>下载它。

当你使用这个数据库的时候，你会发现你也必须按照符合外键约束的顺序填充表；你现在无法在填充orderinfo表之前填充被引用的customer表。我们建议按照以下的顺序：

* customer
* item
* orderinfo
* orderline
* stock
* barcode

### 外键约束的选项

有可能碰到这样的情况，我们在orderinfo表中的项目引用了customer表，但我们需要修改customer\_id。实际情况是，我们无法轻松做到这一点，因为如果我们尝试修改customer\_id（实际上，这是一个坏主意，因为它是一个序列类型的列），orderinfo表的外键约束将阻止这么做，因为规则说明了orderinfo中的记录中的customer\_id必须总是引用到customer表中的项目的customer\_id。

我们也无法修改orderinfo表中的customer\_id，因为对应的项目在customer表中还不存在，而且我们也无法修改customer表中的项目，因为它被orderinfo表引用了。

当你需要短时间地在数据修改时违反外键约束时，SQL标准允许两种方法来解决这种情况，但交易完成之前，你必须恢复数据的完整性：

* 使约束变成延迟的。
* 定义外键约束规则时指出如何处理违例。

#### 延迟约束

第一种允许在某些情况下外键约束违例的方法是在外键约束之后添加INITIALLY DEFERRED关键字：

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL,

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id),

CONSTRAINT orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id)

REFERENCES customer(customer\_id) INITIALLY DEFERRED

);

这改变了外键约束试试的方法。通常，PostgreSQL会在对数据库进行任何变动前进行外键约束检查。但如果你使用了事物（我们将在下一章遇到）和INITIALLY DEFERRED，PostgreSQL允许违反外键约束，允许在事务中违反外键约束，并且这个违例将在事务结束前被修正。事实上，发生的事情是PostgreSQL挂起事务检查，直到将要完成当前事务。

就像我们将在第9章看到的，一个事务是指一组必须要么都被执行要么都不被执行的SQL命令。因此，我们可以启动一个事务，更新customer表的customer\_id，更新orderinfo表相关的customer\_id的值，提交事务，那么PostgreSQL会允许这么做的。它所要做的就是在事务完成的时候检查是否符合约束条件。

注：当然，你也可以只使用DEFERRED关键字，但在这种情况下，你需要使用命令SET CONSTRAINTS ALL DEFERRED，这样PostgreSQL默认只在事务结束的时候检查DEFERRED约束。要获得SET CONSTRAINTS选项的更多细节，请参考在线文档。

#### ON UPDATE和ON DELETE

另一个解决方案是在外键约束规则中指定在两种情况下发生违例的处理方法：UPDATE和DELETE操作。可能的两种动作为：

* 我们可以从有主键的表CASCADE变动。
* 我们可以使用SET NULL将列置为NULL，因为它不会指向到主表。

以下为一个示例：

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL,

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id),

**CONSTRAINT orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id)**

**REFERENCES customer(customer\_id) ON DELETE CASCADE**

);

本例告诉PostgreSQL，如果我们删除customer表中的一个被orderinfo表引用的customer\_id所在的行，则orderinfo表中的相应行也将被自动删除。这也许是我们特意选定的做法，但通常情况下这是一个非常危险的做法。最好的办法是确保应用程序按照正确的顺序删除行，所以我们应该确保在删除客户项前已经没有这个客户的订单了。

SET NULL选项通常和UPDATE和DELETE语句一起使用。它看起来应该是这样：

CREATE TABLE orderinfo

(

orderinfo\_id serial ,

customer\_id integer NOT NULL,

date\_placed date NOT NULL,

date\_shipped date ,

shipping numeric(7,2) ,

CONSTRAINT orderinfo\_pk PRIMARY KEY(orderinfo\_id),

**CONSTRAINT orderinfo\_customer\_id\_fk FOREIGN KEY(customer\_id)**

**REFERENCES customer(customer\_id) ON UPDATE SET NULL**

);

这里的意思是如果customer表中被引用的customer\_id被删除，则orderinfo表中相应的列将被置NULL。

你会发现在我们的表中，这无法工作。我们定义了customer\_id为NOT NULL，所以它无法被更新成NULL值。我们这么做的原因是我们不想orderinfo表中的customer\_id的值可能为NULL。毕竟，没有客户的订单意味着什么呢？可能是出错了。

这些选项可以被组合，所以你可以写一下的语句：

ON UPDATE SET NULL ON DELETE CASCADE

注意：使用ON UPDATE和ON DELETE要非常小心。强制应用程序开发人员按照正确的顺序编写UPDATE和DELETE代码并使用事务会更安全，而不要在其他表发生变化时使用CASCADE DELETE或者突然将某个列的值存储为NULL。

在第10章，我们将看到如何使用触发器和存储过程来实现相同的效果，但它会给我们对其他表的变动带来更好的控制。

## 摘要

在本章我们涵盖了大量的内容。我们从更正式地查看PostgreSQL支持的数据类型开始，尤其是常用的SQL标准类型，但也提及了一些PostgreSQL的不常用的扩展类型，例如数组。然后我们学习了如何操作列数据——转换数据类型，使用数据的子串，以及通过PostgreSQL的“神奇”变量访问信息。

之后我们学习了表的管理，集中尽力关注一个重要的内容：约束。我们知道了有两种方法定义约束：针对单一的列和在数据库级别。即使是简单的约束也可以帮助我们在数据库级别确保数据的完整性。

之后，我们看到如何使用视图来创建一个表的“幻象”。视图可以为用户访问数据提供一个简单的方法，也可以隐藏不想被其他人看到的一些数据。

我们最后的主题是最重要的一种约束：外键。它允许我们正式地在数据库中定义不同表之间的关系。最重要的是，它们允许我们强制执行这些规则，例如确保我们无法删除一个在其他表中有订单信息关联的客户的客户信息。

在学习如何确保数据库的参照整性后，我们建立了一个更新的数据库设计，bpfinal，我们将在本书剩余的部分使用它。

在下一章，我们将讨论事务和锁，它们在处理多个用户需要同时访问一个数据库时非常有用。

# 事务和锁

到目前为止，在本书中我们一直避免任何对PostgreSQL在多用户方面的深入讨论，简单的说明理想化的观点，和其他优秀的关系数据库一样，PostgreSQL隐藏了支持多用户并发的细节。它只是简单地提供了一个快速和高效的数据库服务器，对它的每个客户端提供服务，让每个客户端觉得是进行了独占的访问。特别是一些小的轻量级负载的数据库，这种理想化的观点在实际中基本上得以实现了。然而，事实上是，PostgreSQL虽然非常很强大，但也不会变魔术，为每个用户实现隔离需要幕后大量的工作。

在本章，我们将从两个方面了解数据库对多用户的支持：事务和锁。事务允许你收集许多不相关的数据库变动到一个工作单元。锁在不同用户同时改变数据库时避免冲突。

在本章，我们将涵盖以下主题：

* 事务的组成
* 在单用户数据库中使用事务的好处
* 多用户情况下的事务
* 行锁和表锁

## 什么是事务

就像我们在前一章说的，理想情况下的，你应该通过一个简单的说明性语句描述数据库的变动。但是，在现实世界的应用程序中，很快就会出现你需要对数据库做一系列无法用单个SQL语句描述的改动。虽然他们无法在一个语句中完成，你仍然需要所有对数据库的改变都是正确的。如果这组变动中的任何一部分出错，那么数据库就不应发生任何变动。换句话说，你需要执行一组不可分割的工作，它需要多个SQL语句被执行，且要求要么所有的SQL语句都执行成功，要么没有一条被执行。

关于事物最经典的例子是在两个银行账户间转账，也许账户存在于数据库中的不同表之间，因此一个账户被计入借方，另一个被计入贷方。如果你将一个账户计入借方后，由于某些原因无法将另一个账户计入贷方，你必须将钱退还给第一个账户，就像刚才扣减的操作没有发生过一样。如果钱在不同账户间转移时会偶尔消失，银行也就无法继续经营了。

基于ANSI SQL的数据库，包括PostgreSQL在内，通过事务做到这种要么成功要么失败的任务。一个事务是一个不可拆分的工作的逻辑单元。

### 将数据改变分组到逻辑单元

工作的逻辑单元是什么意思呢？它是一组对数据库的逻辑改变，他们要么必须都被执行，要么必须都失败，就像刚才示例中的在账户之间转账的例子。PostgreSQL中，这些改变由四个关键语法来控制：

* BEGIN开始一个事务
* SAVEPOINT savepointname告诉服务器需要记住事务的当前状态。这个语法只能在BEGIN之后且在COMMIT或者ROLLBACK之前；也就是说，必须在一个事务中。
* COMMIT表明所有的事务的元素都完成了，现在需要对所有的并行的事务和子事务进行持久化并对使其结果（在本事务之外）可访问。
* ROLLBACK [TO savepointname] 表明这个事务将被放弃，SQL事务中所有对数据的改变将被取消。数据库给所有用户的表现是从BEGIN开始的任何改变都没有发生，且事务被关闭。另一个带TO从句的版本允许我们回滚到一个命名的保存点，且不完成这个事务。

注意：ANSI SQL92表中没有定义BEGIN这个SQL语法。它定义了事务为自动开始的（因此这个语法是多余的），但这在很多关系数据库中是一个非常常见的扩展。SQL99添加了语句START TRANSACTION，它的效果和BEGIN一样。PostgreSQL从7.3开始就像支持BEGIN语法一样支持这个新的语法，但是我们保留了BEGIN语法，因为当前它更通用。

### 并发多用户访问数据

事务的第二方面是任何数据库的事务都将从同时在数据库中发生的其他事务隔离。在理想的情况下，对于每个事务，感觉就是它拥有对数据库的独占访问。不幸的是，就像我们将在本章靠后的内容所说的，当我们面对多用户情况下的事务时，必须采取一些折中的方案以保证实际应用中中达到高性能的目的。

让我们看另一个需要事务的不同例子。假设你想联机预订一张机票。你查找你想要的航班并发现一张机票。虽然你不知道，但这是这般飞机上的最后一张机票。当你在输入你信用卡的细节的时候，同一航班上另一位客户也尝试获得这张机票。你还没有支付你的这张机票时，另一个人在你仍在输入你的信用卡信息的时候看到了这个空位并使用他的会员账号预定了它。之后你提交并购买了这张机票，而且因为系统知道在你开始你的事务之前剩余一张座位，它错误地认为那张座位还存在且扣了你的信用卡的费用（当然，航空公司有更成熟的系统避免这种很基础的订票错误问题，但本例只是为了说明这个原理）。

你断开连接，确信你成功预订座位，甚至也许会去检查你的信用卡被成功扣费。但事实上，你购买了一个不存在的座位。在你事务被处理的时候，已经没有空位了。

预订流程的代码看起来就像这样：

查看是否有座位

如果有，给客户建议一个座位。

如果客户接受建议，则请求输入信用卡号。

委托银行进行信用卡交易。

从卡中扣钱。

分配座位。

根据被购买的座位数减少剩余的座位数。

任何时候如果只有一个用户使用这个系统，这个顺序事件的过程是完美的。问题是这里有两个客户。真正发生的事情由表9-1描述。

表9-1重叠的事件

|  |  |  |
| --- | --- | --- |
| 客户1 | 客户2 | 飞机上剩余的座位数 |
| 查看是否还有座位 |  | 1 |
|  | 查看是否还有座位 | 1 |
| 如果有，建议座位给客户 |  | 1 |
|  | 如果有，建议座位给客户 | 1 |
| 如果客户接受建议，要求输入信用卡号或会员账号 |  | 1 |
|  | 如果客户接受建议，要求输入信用卡号或会员账号 | 1 |
| 输入信用卡号码 | 输入会员账号 | 1 |
| 委托银行进行信用卡交易 |  | 1 |
|  | 检查会员账号是否存在 | 1 |
|  | 在新事务中更新账户 | 1 |
| 扣信用卡 | 分配座位 | 1 |
| 分配座位 | 根据被购买的座位数减少剩余的座位数 | 0 |
| 根据被购买的座位数减少剩余的座位数 |  | -1 |

我们该如何解决这种订票应用程序的问题呢？我们可以优化到在最接近扣钱的时候重新检查是否还存在座位，但是无论我们怎么检查，必然会发生“检查座位是否存在”的步骤和“扣钱”的步骤是分离的，即使只间隔很短的时间。

我们可以走到另一个极端去解决这个问题，同时只允许一个用户访问订票系统，但是这个性能会极低，可能会导致客户流失。

从应用程序术语上说，我们拥有的是一段临界代码——一小段需要独占访问数据的代码。我们可以使用信号灯或者类似的技术写我们的程序，来管理对临界代码的访问。这可能需要每个访问数据库的应用程序使用信号灯。然而，比起编写应用程序逻辑，通常数据库处理起这个问题来更容易。

从数据库术语上说，我们这里碰到的就是一个事务——从检查座位是否存在到从信用卡或者账户扣费再到分配座位这一组数据操作动作，这些必须在一个单元内完成。

### ACID规则

ACID是一个常用的缩写用于描述事务必须拥有的四个属性：

Atomic**（原子**性**）：一个事务，即使是在数据库中的一组不相干的操作，必须在同一个单元内发生。一个事务必须一次性发生，没有子集或者无意识的重复动作。在我们的银行示例中，转账必须是原子的。**从一个账户扣款并往另一个账户存款虽然是两个独立的动作但必须同时发生，即使需要更多连续的SQL语句。

Consistent（一致性）：在事务结尾的时候，系统必须保持一致状态。我们在第8章接触到了这个，当时我们看到我们可以定义一个延期约束；换句话说，约束应该指在事务完成的时候被检查。在我们银行的示例中，在事务结尾的时候，所有的账户必须如实反应被扣款和存款。

Isolated（隔离性）：这意味着对于每个事务，无论数据库中有多少事务在运行，都应该表现出不依赖于其他任何事务。在我们的机票预订示例中，事务处理的两个并发的用户必须表现出他们都是对数据库进行独占使用的。在现实中，我们知道在多用户数据库中，如果我们想让它符合这种合理的行为是不现实的，实际上这是现实世界中的实际问题中可以影响我们理想数据库最为显著的地方之一。我们将在本章稍后的“多用户下的事务”小节中讨论事务的隔离问题。

Durable（持久性）：一旦事务完成，必须保持完成状态。一旦在账户之间成功转账，必须保持转账完成的状态，即使在意外情况下运行数据库的主机掉电。在PostgreSQL中，和其他大多数关系型数据库一样，可以通过一个事务日志文件做到这一点，就像下一节所讲解的。事务的持久性不需要用户干预。

### 事务日志

就像在上一节提到的，事务日志文件被数据库内部使用以确事务的持久。事务日志文件的工作方式很简单。当事务执行后，改动不仅仅是写入数据库，还被写入日志。一旦事务完成，一个标记被写入用以说明事务已经完成，日志文件的数据被强制持久化存储，所以它是安全的，即使数据库服务器崩溃。数据库服务器可能因为某些原因在事务过程中崩溃，那么在服务器重启时，它有能力自动确保完成的事务被正确反映到数据库（通过重做（ROLLING FORWARD）在事务日志中但未进入数据库的事务）。服务器关机时在进行中的事务的改变不会体现在数据库中。

PostgreSQL维护的事务日志不仅记录所有的对数据库的改动，也记录如何撤销它们。很明显，这个文件会很快变得很大。一旦一个COMMIT语句在一个事务中执行，PostgreSQL就知道不再需要存储“撤销”信息了，因为数据库的改动已经是不可反转的了，至少对于数据库是这样的（应用程序可以执行附加代码来撤销改变）。

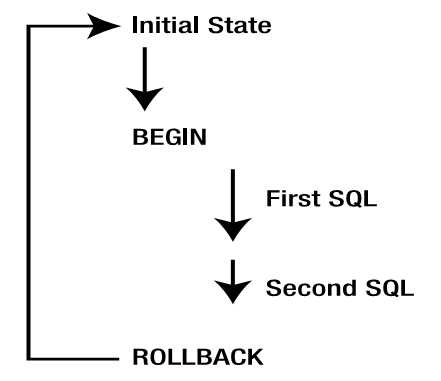
PostgreSQL使用了一种技术，让数据在写入磁盘中的表中之前先写入事务日志文件，因为它知道一旦数据被写入日志文件，它可以通过日志恢复表中数据的状态，即使系统可能在真实数据文件被修改前系统发生故障。这被叫做预写日志（Write Ahead Log，WAL），对此感兴趣的读者可以在PostgreSQL问的行中找到更多细节信息。

## 单用户情况下的事务

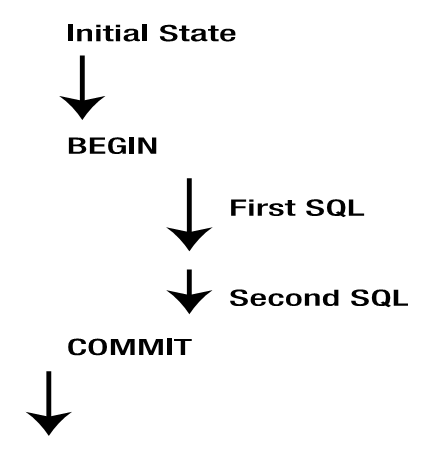
在我们学习事务更复杂的方面和数据库面对多用户并发操作中它们的行为之前，我们需要先看看它们在单用户情况下的行为。即使在这种非常简单的工作模式中，使用事务都是非常有用的。

使用事务最大的好处是它们允许你执行很多SQL语句，然后在之后的步骤，如果你确定需要，允许你撤销你刚才做的工作。另外，如果你的某条SQL语句失败，你可以撤销你的工作到预先定义的时刻。

使用事务，应用程序不需要关心如何存储对数据库的改变并撤销它们。只需要简单地高速数据库引擎一次性撤销所有的改变。逻辑上说，整个顺序由图9-1描画。

  
图9-1回滚一系列改变

如果你确定图9-1中在“Second SQL”后你所有对数据库的改动都是有效的，而且你希望应用它们到数据库以便他们能够持久，那么你需要做的就是用COMMIT语句替换ROLLBACK语句，如图9-2所示。

  
图9-2 提交一系列改变

在COMMIT后，所有对数据库的改变都被提交，且可以被认为持久写入到数据文件中，所以它们不会因为掉电或者应用程序错误而丢失。

##### 尝试：执行一个简单的事务

让我们尝试一个简单的事务，在其中我们改变一个表的一行，然后使用ROLLBACK语句来取消改变。我们将使用test数据库来做这个实验。首先，连接到test数据库（如果它不存在，只需要使用CREATE DATABASE test命令），然后建立一对简单的表进行实验：

bpfinal=> \c test

You are now connected to database "test".

test=> CREATE TABLE ttest1 (

test(> ival1 integer,

test(> sval1 varchar(64)

test(> );

CREATE TABLE

test=> CREATE TABLE ttest2 (

test(> ival2 integer,

test(> sval2 varchar(64)

test(> );

CREATE TABLE

test=>

现在我们可以尝试一个简单的事务了：

test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');

INSERT 17784 1

test=> BEGIN;

BEGIN

test=> UPDATE ttest1 SET sval1 = 'Dave' WHERE ival1 = 1;

UPDATE 1

test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;

sval1

-------

Dave

(1 row)

test=> ROLLBACK;

ROLLBACK

test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;

sval1

-------

David

(1 row)

test=>

##### 解析

我们最初插入了一行数据，存储的名字为“David”。然后我们使用BEGIN命令开始了一个事务。下一步，我们更新这一行的sval1列，设置名字为“Dave”。当我们在这一行上执行一个SELECT，它显示数据被改变了。我们然后调用ROLLBACK。PostgreSQL使用它的内部事务日志撤销从BEGIN开始的改变，所以下一次我们再SELECT这个行的时候，我们的改变被回滚了。

有趣的是，如果我们使用另一个psql会话并在更新David为Dave之后但在执行ROLLBACK之前立即查询数据库，我们会发现依旧是David在数据库中。这是因为PostgreSQL从未提交的数据库数据更新中隔离了用户，除了做出改变的用户。我们将在本章之后的“多用户情况下的事务”小节深入讨论。

### 包括多个表的事务

事务不限制于在一个表中或者只是更新数据。让我们看一个更复杂的包括多个表且包含UPDATE语句和INSERT语句的示例。

##### 尝试：在多个表中执行事务

让我们实验事务在多个表上的效果。首先，确保两个表都是空的，然后插入一行到第一个表中：

test=> DELETE FROM ttest1;

DELETE 1

test=> DELETE FROM ttest2;

DELETE 0

test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');

INSERT 17793 1

现在开始一个事务并做出一些改变：

test=> BEGIN;

BEGIN

test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');

INSERT 17794 1

test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;

UPDATE 1

test=> SELECT \* FROM ttest1;

ival1 | sval1

-------+--------

1 | Robert

(1 row)

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+--------

42 | Arthur

(1 row)

现在执行一个ROLLBACK并查看效果：

test=> ROLLBACK;

ROLLBACK

test=> SELECT \* FROM ttest1;

ival1 | sval1

-------+-------

1 | David

(1 row)

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+-------

(0 rows)

test=>

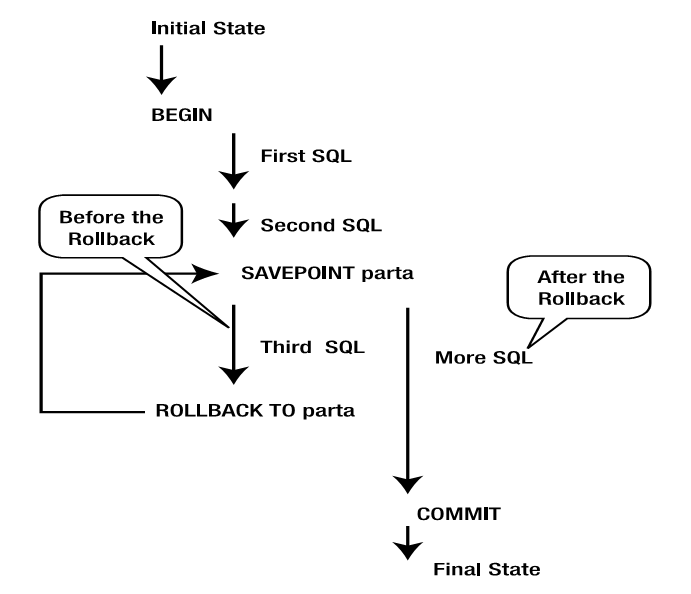
##### 解析

ROLLBACK导致INSERT语句添加的数据被移除，对ttest1表的UPDATE被撤销。这演示了一个事务将一些改变合并为一组可以跨多个表工作的情况。

### 事务和保存点

上一个示例使用了基本的事务语法，这也是大多数应用程序需要的。然而，保存点可以对你在事务中需要回滚到某个特殊的点的情况下非常有用。这需要扩展版本的事务语法，这带有一个命名的保存点和ROLLBACK TO命令。

如果我们需要回滚一个事务中的一些操作，我们可以建立一个我们待会儿可以回滚回去的命名的保存点，而不是回滚到我们从BEGIN语句开始的地方。图9-3描绘了这个场景。

  
图9-3 使用一个保存点

在图9-3的示例中，我们通过执行BEGIN语句开始我们的事务，然后执行两个SQL语句。之后我们建立一个叫做parta的保存点，然后执行了第三条语句。我们接着执行ROLLBACK TO parta语句，它实际上撤销了第三条SQL语句的效果。我们之后可以在执行COMMIT来实现数据库改变的持久化前执行一些其他的SQL。

##### 尝试：使用保存点

让我们看看保存点的实际运用。可以给保存点任何名字；我们在这里使用first，但实际上我们可以叫他Tux，Getreidegasse或者其他任何名字。

test=> DELETE FROM ttest1;

DELETE 1

test=> DELETE FROM ttest2;

DELETE 0

test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');

INSERT 17795 1

test=> BEGIN;

BEGIN

test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');

INSERT 17796 1

test=> SAVEPOINT first;

SAVEPOINT

test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;

UPDATE 1

test=> SELECT \* FROM ttest1;

ival1 | sval1

-------+--------

1 | Robert

(1 row)

test=> ROLLBACK TO first;

ROLLBACK

test=> SELECT \* FROM ttest1;

ival1 | sval1

-------+-------

1 | David

(1 row)

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+--------

42 | Arthur

(1 row)

test=>

此时我们依旧在事务中，所以我们依旧可以回滚到初始的BEGIN状态：

test=>

test=> ROLLBACK;

ROLLBACK

test=> SELECT \* FROM ttest1;

ival1 | sval1

-------+-------

1 | David

(1 row)

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+-------

(0 rows)

test=>

此时，通过调用ROLLBACK，事务回到了初始的BEGIN状态，事务此时已经完成了，我们不能调用另一个ROLLBACK或者COMMIT，直到另一个BEGIN语句被执行：

test=> INSERT INTO ttest2 (ival2, sval2) VALUES (99, 'Chris');

INSERT 17797 1

test=> COMMIT;

WARNING: there is no transaction in progress

COMMIT

test=>

而且，一旦我们通过COMMIT指定事务完成，则改变都被永久写入数据库，不再有办法回到以前：

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+-------

99 | Chris

(1 row)

test=> BEGIN;

BEGIN

test=> UPDATE ttest2 SET sval2 = 'Gill' WHERE ival2 = 99;

UPDATE 1

test=> COMMIT;

COMMIT

test=> ROLLBACK;

WARNING: there is no transaction in progress

ROLLBACK

test=> SELECT \* FROM ttest2;

ival2 | sval2

-------+-------

99 | Gill

(1 row)

test=>

##### 解析

在本例中，保存点允许我们回滚到一个事务的中间点或者回滚到事务的开始点。一旦一个ROLLBACK被执行，数据库看上去就好像被回滚掉的改变从未发生过一样。一旦一个事务被提交，则它无法再通过ROLLBACK撤销。、

### 事务的限制

虽然事务工作起来非常完美，但它也有一些限制。这涉及到嵌套、大小和持续时长。

#### 嵌套

在PostgreSQL（或大多数其他的关系型数据库中）你不能嵌套事务。在PostgreSQL中，如果你尝试在一个事务中执行一个BEGIN语句，PostgreSQL将产生一个警告消息，告诉你已经有一个事务在进行中。

一些数据库默默地接受多个BEGIN语句。COMMIT或者ROLLBACK语句将只针对第一个BEGIN有效，因此虽然看上去事务被嵌套了，但实际上，随后的BEGIN语句被忽略了。

#### 事务大小

建议尽量使用小事务。就像我们将在本章后面看到的，PostgreSQL（以及其他关系型数据库）为了确保事务从其他用户隔离出来，必须做大量的工作。这样做的结果是在事务中对数据库的调用部分频繁需要加锁，以确保事务被隔离。因此，你需要确保每个事务不会比它的实际需要大。在每个事务中包含大量的不必要的事务将导致在数据库中产生大量的锁，这将影响性能和其他用户对数据库的访问。我们将在本章靠后的“锁”小节讨论锁的细节。

#### 事务持续时长

事务不应该长时间保持打开状态。虽然PostgreSQL自动为你锁定数据库，但是长时间运行的事务通常防止其他用户访问由这个事务产生的数据，直到事务被提交或者回滚。因此，在需要与用户进行对话的时候你应该避免先启动事务。建议先从用户那里收集所有需要的细腻，然后在一个事务中处理这些信息，避免不可预期的用户响应时间。

假设一个用户一早在终端前使用一个蹩脚的程序启动了一个事务，然后让这个事务伴随着用户对数据库做了大量修改跑了一整天。随着用户在数据库中做工作，越来越多的东西被锁住，等待这些改变被提交。如果这个用户只在一天的结尾才提交数据，其他用户访问数据的能力将被严重压缩，而且整个应用程序可能在需要多用户的情况下变得不可用。

你还应该注意尽管一个COMMIT语句通常被快速的执行，因为它通常只需要做很少的事情，而回滚事务通常需要数据库做大量工作来使以前回到初始状态，因此需要一些时间执行。因此，如果你开始一个事务，花了两分钟时间执行所有的SQL，然后决定执行一个ROLLBACK或者取消所有改变，请别指望回滚能马上完成。它可能花费随便就花费超过两分钟的时间才能撤销所有的改变。

## 多用户情况下的事务

就像我们在本章前面部分看到的，支持多用户并发的工作的事务需要将用户从其他每个人隔离开来（ACID的I部分）。虽然PostgreSQL的默认处理隔离的行为就满足大多数情况，但依旧需要了解它的细节的情况。

### 实现隔离

关系型数据库中一个最难的方面是隔离不同的用户对数据库的更新。当然，如果我们不在乎性能，实现隔离也不难。简单地允许一个用户连接到数据库，并同时只允许一个事务，将确保在不同事务间完全隔离。不幸的是，多用户的性能将变得极差。事务隔离的难点是实现有效隔离的同时不明显影响性能或影响多用户访问数据库。

为了减少隔离对性能的影响，ANSI SQL标准定义了数据库可以实现的不同的数据库隔离级别。这允许数据库管理员在性能和针对数据库的用户的隔离程度进行做出平衡。通常，一个关系型数据库默认将最少实现这些级别中的一个，并允许用户最少指定另一个隔离级别来使用。

ANSI SQL标准根据在多用户数据库中当事务配合时可能出现的“不良现象（undesirable phenomena）”定义隔离级别。这些现象被称作脏读、不可重复读、幻读和更新丢失。让我们看看它们每个的意义以及ANSI隔离级别是如何定义的。

#### 脏读

脏读发生在一个事务中的SQL读到被其他事务改变的数据，但这个（其它）事务尚未提交其工作块中改变的数据。

就像我们早先提到的，一个事务是一个必须原子操作的逻辑单元或者工作块。要么事务中所有的元素都发生要么都不发生。直到一个事务被提交前，总有可能事务失败或者通过ROLLBACK命令放弃。因此，没有数据库中的其他用户可以看到COMMIT前改变的数据。

表9-2阐述不同的事务可以看到允许脏读和不允许脏读的情况下的customer\_id为15的客户的fname。

表9-2脏读

|  |  |  |  |
| --- | --- | --- | --- |
| 事务1 | 事务1看到的数据 | 其他允许脏读的事务看到的数据 | 其他禁止脏读的事务看到的数据 |
| BEGIN |  |  |  |
|  | David | David | David |
| UPDATE customer SET fname='Dave'  WHERE customer\_id = 15; |  |  |  |
|  | Dave | Dave | David |
| COMMIT |  |  |  |
|  | Dave | Dave | Dave |
| BEGIN |  |  |  |
|  | Dave | Dave | Dave |
| UPDATE customer SET fname = 'David'  WHERE customer\_id = 15; |  |  |  |
|  | David | David | Dave |
| ROLLBACK |  |  |  |
|  | Dave | Dave | Dave |

注意脏读允许其他事务看到未提交到数据库的数据。这意味着它们可以看到之后被抛弃的改变，因为可能使用ROLLBACK命令。

注：PostgreSQL不允许脏读。

#### 不可重复读

不可重复读非常类似于脏读，只是更加有限制性。一个不可重复读发生在党一个事务读了一些数据，之后重新读这些数据且发现它们被改变了。这里的问题远没有脏读严重，但是也不是很完美。一个不可重复读的阐述展示在表9-3中。

表9-3不可重复读

|  |  |  |  |
| --- | --- | --- | --- |
| 事务1 | 事务1看到的数据 | 其他允许不可重复读的事务看到的数据 | 其他禁止不可重复读的事务看到的数据 |
| BEGIN |  | BEGIN | BEGIN |
|  | David | David | David |
| UPDATE customer SET fname='Dave'  WHERE customer\_id = 15; |  |  |  |
|  | Dave | David | David |
| COMMIT |  |  |  |
|  | Dave | Dave | David |
|  |  | COMMIT | COMMIT |
|  |  | BEGIN | BEGIN |
| UPDATE customer SET fname = 'David'  WHERE customer\_id = 15; |  |  |  |
|  |  | Dave | Dave |

注意不可重复读意味着一个事务可以看到其他事务提交的改变，即使读的事务本身还没有提交。如果禁止了不可重复读，其他事务在提交自己的改变前无法看到数据库的改变。

默认情况下，PostgreSQL允许不可重复读，但就像我们在后面将看到的，我们可以改变这个默认行为。

#### 幻读

幻读和不可重复读非常类似，但是发生在一个新行出现在一个表中的时候，另一个事务在更新表，这个新行应该被更新了但实际上没有。

假设我们有两个事务在更新item表。第一个事务为所有的项目的售价增加一元，而第二个事务则增加一个新项目。这个过程在表9-4中描述。

表9-4幻读

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| BEGIN | BEGIN |
| UPDATE item SET sell\_price = sell\_price + 1; |  |
|  | INSERT INTO item(….) VALUES(…); |
| COMMIT |  |
|  | COMMIT |

由事务2添加的项目的售价应该为多少？INSERT语句在UPDATE语句提交前开始；但是，我们可能有理由期望它会比我们插入的价格高1。如果发生幻读，因为新行出现在事务1中的UPDATE探测到哪些行需要更新之后，那么新项目的价格不会增加1.

幻读非常少见，甚至很难做演示，所以通常你不需要关心它们。默认情况下，PostgreSQL允许幻读。

#### 更新丢失

更新丢失和以上三个示例有点区别，它通常是一个应用程序级别的问题，而跟关系型数据库的工作方式无关。更新丢失发生在两个不同的改变被写入数据库时，第二个更新导致第一个丢失。

假设两个用户都在使用一个基于屏幕的应用程序，用以更新item表。过程在表9-5中显示。

表9-5更新丢失

|  |  |  |  |
| --- | --- | --- | --- |
| 用户1 | 用户1看到的数据 | 用户2 | 用户2看到的数据 |
| 尝试将售价从21.95修改为22.55 |  | 尝试将成本从15.23修改为16.00 |  |
| BEGIN |  | BEGIN |  |
| SELECT cost\_price,  sell\_price FROM item  WHERE item\_id = 1; | 15.23, 21.95 | SELECT cost\_price,  sell\_price FROM item  WHERE item\_id = 1; | 15.23, 21.95 |
| UPDATE item SET cost\_price  = 15.23, sell\_price = 22.55  WHERE item\_id = 1; |  |  |  |
|  | 15.23, 22.55 |  |  |
| COMMIT |  |  |  |
|  |  |  | 15.23, 22.55 |
|  |  | UPDATE item SET  cost\_price = 16.00,  sell\_price = 21.95  WHERE item\_id = 1; |  |
|  | 15.23, 22.55 |  | 16.00, 21.95 |
|  |  | COMMIT |  |
|  | 16.00, 21.95 |  | 16.00, 21.95 |

被用户1改变的售价丢失了，这并不是因为一个数据库错误，而是因为用户2读取sell\_price后，“保存”了一会儿，然后将它写回数据库，这导致用户1做的改变被破坏了。数据库十分正确地隔离了两个操作，但应用程序还是丢失了数据。

有很多方法解决这个问题；不同的应用程序可能会选择恰当的方法。首先，应用程序应该尽量保持事务短小，除非绝对需要，否则不要保持事务为运行状态。第二步应用程序应该只回写它们改变的数据。这两步将避免很多更新丢失的发生，包括表9-5中的演示中的错误。

当然，存在两个用户都尝试修改sell\_price的情况；在这种情况下，其中一个改变可能会丢失。一个更通用的避免更新丢失的方法是在UPDATE语句中编码你将要改变的值，就像表9-6中描述的。

表9-6针对更新丢失的一个应用程序解决方法

|  |  |  |  |
| --- | --- | --- | --- |
| 用户1 | 用户1看到的数据 | 用户2 | 用户2看到的数据 |
| 尝试将售价从21.95修改为22.55 |  | 尝试将售价从21.95修改为22.99 |  |
| BEGIN |  | BEGIN |  |
| 读sell\_price（WHERE item\_id=1） | 21.95 | 读sell\_price（WHERE item\_id=1） | 21.95 |
| UPDATE item SET cost\_price = 15.23,  sell\_price = 22.55 WHERE item\_id = 1  AND sell\_price = 21.95; |  |  |  |
|  | 22.55 |  | 21.95 |
| COMMIT |  |  |  |
|  |  | UPDATE item SET cost\_price = 16.00,  sell\_price = 21.95 WHERE item\_id = 1  AND sell\_price = 21.95; |  |
|  |  | 由于行未找到而更新失败，因为sell\_price被改变了。 |  |

虽然这不是一个完美的解决方法，因为它只在第一个事务在第二个UPDATE语句运行前提交才能正确工作，但它可以有效减少更新丢失的风险。

#### ANSI隔离级别

ANSI标准针对这三种不良现象（脏读、不可重复读、幻读）的组合定义了一个数据库可用的不同隔离级别。这些级别在表9-7中列出。

表9-7 ANSI隔离级别和不良现象的对照

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| Read Uncommitted（读未提交） | 允许 | 允许 | 允许 |
| Read Committed（读已提交） | 不允许 | 允许 | 允许 |
| Repeatable Read（重复读） | 不允许 | 不允许 | 允许 |
| Serializable（串行化） | 不允许 | 不允许 | 不允许 |

就像你看到的，随着隔离级别从读未提交到最后的串行级别，发生的不良现象的类型减少了。

### 改变隔离级别

默认情况下，PostgreSQL的隔离模式被设置为READ COMMITTED，对应为表9-7中列出的Read Committed。存在的另一个级别为SERIALIZABLE，对应Serializable。

到8.4版本为止，PostgreSQL未实现中间的Repeatable Read和初始的Read Uncommitted。通常，Read Uncommitted由于其行为很烂，很少有数据库提供这个选项，而且罕有应用程序大胆（或者傻逼）到选择使用它。（实际上，PostgreSQL 8.4的SERIALIZABLE级别的实际实现为Repeatable Read，PostgreSQL把Read Uncommitted级别看做是Read Committed。）中间级别Repeatable Read只针对幻读提供额外保护，而这种情况也很少见，所以缺乏这种级别的后果也不严重。通常数据库只会提供可选项的子集，而支持Read Committed和Serializable是一个比较好的折中方案。

你可以通过使用SET TRANSACTION ISOLATION LEVEL命令改变隔离级别，以下为语法：

SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }

除非你有很好的理由改变它，否则我们建议你不要改变你的PostgreSQL数据库中的默认的隔离级别。

### 使用显式和隐式的事务

截至本章，我们已经通过明确地使用BEGIN和COMMIT（或ROLLBACK）来确定我们的事务。在本书较早的部分，我们尚未了解事务的时候，我们确能够开心地不使用BEGIN命令而改变我们的数据库。

默认情况下，PostgreSQL运行在自动提交模式，有时也被称为链式模式或隐式事务模式，在这种情况下每个能修改数据的SQL语句仿佛是在它的权限范围内的完整的事务中运行。这对在命令行实验或者允许新用户来体验而不需要太多SQL知识的情况非常有用。然而，它对于真正的应用程序却没那么好，因此我们需要显式地在事务中使用COMMIT或ROLLBACK语句。

在其他的关系型数据库管理系统中，实现模式有点不同，通常必须使用显式的命令来改变模式；例如，Sybase使用SET CHAINED，微软的SQL Server使用SET IMPLICIT\_TRANSACTIONS命令。

在PostgreSQL中，你所需要做的就是调用BEGIN命令，PostgreSQL将自动切换到一种接下来的命令在事务中的模式，知道你调用一个COMMIT或者ROLLBACK语句。

SQL标准认为SQL语句是在事务中执行的，认为事务在第一个SQL执行的时候就自动开始了，直到遇到一个COMMIT或ROLLBACK。因此，标准SQL没有定义BEGIN命令。然而，PostgreSQL显式地使用BEGIN使用执行事务的模式也很常用。

## 锁

大多数数据库实现了事务，特别是当隔离不同用户之间的事务的时候，使用锁来在用户间约束对数据的访问。简单来讲，有两种锁：

* 共享锁，允许其他用户读，但不允许修改数据
* 排他锁，甚至避免其他事务读数据

例如，服务器将锁定被一个事务更新的行直到事务完成，之后这些锁才被释放。这些都被自动完成，通常甚至不需要数据库用户关注锁的发生。

实现锁需要的实际机制和策略极度复杂，有很多不同类型的锁可以使用，具体的依赖于环境。PostgreSQL的文档描述了八种不同类型的锁。PostgreSQL还通过一种多版本模式实现了一种不同的事务隔离机制，这减少了锁的冲突，相对于其他模式明显提升了它的性能。

幸运的是，数据库用户通常只需要在两种情况注意锁：避免死锁（以及从死锁中恢复）以及通过应用程序显式加锁。

### 避免死锁

当两个不同的应用程序同时尝试修改相同的数据会发生什么呢？很容得到结果——只需要启动两个psql会话，并尝试在它们中改变相同的行。过程在表9-8中描述。

表9-8死锁

|  |  |
| --- | --- |
| 会话1 | 会话2 |
| 更新第14行 |  |
|  | 更新第15行 |
| 更新第15行 |  |
|  | 更新第14行 |

在这种情况下，两个会话都被阻塞了，因为每一个都在等待另一个释放。

这种行为可以作为推断PostgreSQL的默认隔离级别为Read Committed模式的线索。在并发性、性能、最大限度地减少一侧的锁的数量以及一致性和其他理想的行为之间有一个权衡。当你提高隔离级别，数据库中的多用户的并发性能将降低，就像在图9-4中指出的。

性能

隔离级别

图9-4 性能和隔离级别的反比关系

随着数据库的行为越来越智能，需要的锁的数量也增长了，不同用户间的并发性也降低，因此整体性能降低了。这是一个必须面对的负面消息。

通常，如果两个用户尝试访问相同的行，除了第二个用户必须等待第一个用户完成访问外，没有其他的实际影响。但一种更严重的情况是两个会话会互相阻塞。

##### 尝试：实验死锁

让我们用我们在第八章设计的bpfinal数据库模式做一个实验。开始两个psql会话，连接到bpfinal，并按照以下顺序执行命令：

|  |  |
| --- | --- |
| 会话1 | 会话2 |
| BEGIN |  |
|  | BEGIN |
| UPDATE customer SET fname = 'D' WHERE customer\_id = 15; |  |
|  | UPDATE customer SET fname = 'B' WHERE customer\_id = 14; |
| UPDATE customer SET fname = 'Bill' WHERE customer\_id = 14; |  |
|  | UPDATE customer SET fname = 'Dave' WHERE customer\_id = 15; |

你会发现两个会话都被阻塞住了，在短暂的暂停后，你将在其中一个会话中看到类似以下的消息：

ERROR: deadlock detected

DETAIL: Process 2018 waits for ShareLock on transaction 2788; blocked by

process 2017.

Process 2017 waits for ShareLock on transaction 2789; blocked by process 2018.

bpfinal=>

看到这些错误信息的会话的更新操作被取消；另一个会话将继续。获得死锁消息的会话被回滚了，进行的改动丢失了。另一个会话可以继续执行一个COMMIT语句来使数据库的变动持久（或者执行一个ROLLBACK来放弃改变）。

##### 解析

这里发生的事情是PostgreSQL检测到了一个死锁，两个会话都因为等待对方而阻塞，没有谁可以继续。会话1锁住了15行，然后会话而发起且锁住了14行。会话1然后尝试锁住14行，但无法继续，因为这一行被会话2锁住了，然后会话2尝试修改行15，但无法进行因为它被会话1锁住了。在一小段时间后，PostgreSQL的死锁检测代码探测到发生了一个死锁并自动取消了一个事务。

无法进一步确定哪个会话将被PostgreSQL选择来取消。它将挑选一个认为做了最少工作的，但这不是非常科学。应用程序应该能够也必须小心避免死锁的发生。最简单的方法也是我们之前建议的：保持事务尽量短小。事务参与的行和表越少，则锁被锁定的时间越短，那么发生冲突的机会也就越少。

另一个技巧也很简单：让应用程序的代码使用相同的顺序处理表和行。在我们的示例中，如果两个会话都使用了相同的顺序来更新行，则不会发生这样的问题。会话1将可以完成两行的更新，而会话2在短暂的停止后，在会话1的事务完成后，可以继续执行。也可以写代码在发生死锁的时候重试，但最好还是设计你的程序来避免问题，而不是在失败后做重试。

### 显式锁定

偶尔，你可能发现PostgreSQL提供的自动锁定机制不满足你的需要。在这种情况下，你可能需要显式锁定一些行或者整张表。除非必要，否则你应该避免显式锁定。SQL标准甚至没有定义一种方法来锁定整张表。这个选项是一个PostgreSQL的扩展（你会发现这是在很多数据库中都有的一个常见扩展）。

有可能在一个事务中锁定一些行或者表。一旦事务完成，无论是COMMIT或者ROLLBACK，所有在事务中获得的锁将被自动释放。也没有办法在事务中显式释放锁，原因也很简单，在事务中释放一个被改变了的行上的锁可能导致其他应用再改变它，这将阻碍ROLLBACK恢复初始的改变。

#### 锁定行

最常见的需求是在对一些行进行改变前锁定他们。这也是一种避免死锁的方法。通过提前锁定你需要改变的行，你可以确保其他应用程序无法建立一个和你的改变相冲突的部分。

只需要在SELECT语句后面追加FOR UPDATE就可以锁定一些行，就像这个例子：

SELECT 1 FROM item WHERE sell\_price > 5.0 FOR UPDATE;

在事务中提供这行语句，我们将锁定所有sell\_price大于5的行。在本例中，我们不需要返回任何行，为了方便地返回最少的数据，我们简单地选择了1。

##### 尝试：锁定行

假设我们需要锁定customer表中所有的住在Nicetown的记录，因为我们需要修改电话号码（也许是因为区号被拆分成几个了）。我们需要确保我们能访问所有的行，但需要一些过程代码来逐行处理，计算新的电话号码。再次，我们使用第八章建立的bpfinal数据库来做尝试。

bpfinal=> BEGIN

BEGIN

bpfinal => SELECT customer\_id FROM customer WHERE town = 'Nicetown' FOR

UPDATE;

customer\_id

-------------

3

6

(2 rows)

bpfinal =>

这时候，customer\_id为3和6的行被锁定，然后我们可以通过在另一个psql会话中尝试UPDATE它们来测试这一点：

bpfinal l=> BEGIN;

BEGIN

bpfinal => UPDATE customer SET phone = '023 3376' WHERE customer\_id = 2;

UPDATE 1

bpfinal => UPDATE customer SET phone = '023 3267' WHERE customer\_id = 3;

这时候，第二个会话被阻塞了，直到我们敲击Ctrl+C来中断它，或者第一个事务提交或者回滚。

##### 解析

第一个会话使用SELECT … FOR UPDATE，导致customer\_id为3和6的行被锁定。其他的会话可以更新customer表中的其他行，但行3和6一直被锁定直锁定它们的事务结束。

#### 锁定表

在PostgreSQL中，也可以锁住表，虽然我们强烈建议你避免这样做，坚持使用SQL标准的机制来确保客户的隔离。

锁定表的语法如下：

LOCK [ TABLE ] table-name

LOCK [ TABLE ] table-name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE

LOCK [ TABLE ] table-name IN SHARE ROW EXCLUSIVE MODE

通常，程序使用这个简单的语法锁住表：

LOCK TABLE table-name

这和以下的效果相同：

LOCK TABLE table-name ACCESS EXCLUSIVE MODE

这避免了任何程序使用任何方法访问这张表。虽然很苛刻，但这可能是你在实际环境中需要一个表级锁时的行为。更多的关于不同类型锁的细节，请参考PostgreSQL的文档。

## 摘要

在本章，我们了解到了事务和锁。我们看到了即使在单用户的数据库中，事务都是非常有用，因为它们运行我们将一组SQL命令组成一个原子单元，让他们要么都发生要么都放弃。之后我们了解了事务在多用户环境中如何工作。

然后，我们查看了ANSI SQL表中条款中的不良现象，以及针对不同级别的不良行为定义的不同级别的事务一致性。我们还讨论了为何消除不良现象会导致性能的下降，因此有必要取得理想的行为和可接受的性能之间的平衡。

最后，我们看了下锁。我们看到了使用简单的技巧就可以减少死锁的发生的可能，也就是避免两个或者更多应用程序被卡住，等待另一个完成。

我们还讨论了显式锁，这允许我们在一个事务中锁定表中指定的行或者甚至是整个表。

虽然事务和锁不总是非常有意思的话题，但对它们的大体了解有助于写出有良好行为的应用。我们建立的应用不仅需要正确，还要对数据库的性能的影响最低，发挥PostgreSQL的多用户能力。

下一章，我们将浏览其他的方式来发挥PostgreSQL的能力，通过存储过程和触发器扩展它的行为。

# 函数、存储过程和触发器

在本章，我们将研究几种新的方法来扩展和增强到现在为止我们介绍的PostgreSQL。本章的大部分内容是PostgreSQL所特有的，虽然大部分商业的关系型数据库管理系统，例如Oracle也包含类似的功能。

首先，我们将看看PostgreSQL对SELECT语句的更深的支持，包括高级匹配以及允许我们在WHERE从句中构造复杂的测试的算数运算符。然后我们将查看这些操作如何在PostgreSQL中通过函数实现，并介绍一些可以增强我们SELECT语句表现力的附加函数。

PostgreSQL允许开发人员通过使用C语言建立新的功能并在数据库启动的时候加载它们到服务器中来扩展数据库服务器的功能。这种扩展可以是简单到一个扩展函数或者复杂到一个有自己规则的完整的编程语言。有很多扩展，被称为可加载过程语言，已经包含在标准的PostgreSQL的发布包里头。这些语言允许我们建立自己的函数，也就是存储过程，他们用起来比写C程序更快速和容易。在本章，我们将简要介绍其中一种可加载语言：PL/pgSQL。PL/pgSQL是PostgreSQL特有的，但类似的语言也存在于其他数据库中。例如，Oracle有PL/SQL，Sybase有Transact-SQL。

在数据库中当一些特殊条件发生的时候，存储过程也可以被PostgreSQL服务器自动执行。例如，当尝试从一个表中删除一行的时候，一个存储过程可以被执行删除其他表中的相关行来确保参照完整性，或者仅仅是避免删除发生。这些自主的行为就是触发器，我们也将在这里看到它们。

在本章，我们将涵盖以下主题：

* PostgreSQL的运算符
* PostgreSQL的内置函数
* 过程语言，特别是PL/pgsql
* 存储过程
* SQL函数
* 触发器

## 运算符

在第4章，我们已经在SELECT语句中看到并使用了一些简单的运算符。例如，我们可以使用数字比较运算符来限制查询的符合条件的行，例如售价大于4的项目：

bpfinal=# SELECT \* FROM item WHERE cost\_price > 4;

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

1 | Wood Puzzle | 15.23 | 21.95

2 | Rubik Cube | 7.45 | 11.49

5 | Picture Frame | 7.54 | 9.95

6 | Fan Small | 9.23 | 15.75

7 | Fan Large | 13.36 | 19.95

11 | Speakers | 19.73 | 25.32

(6 rows)

bpfinal=#

在这里，运算符 > 被用在cost\_price属性和一个给定的数字之间。

我们可以更进一步，包含其他的属性和运算符来建立更复杂的条件：

bpfinal=# SELECT \* FROM item WHERE (sell\_price\*100)%100 = 99;

item\_id | description | cost\_price | sell\_price

---------+-------------+------------+------------

4 | Tissues | 2.11 | 3.99

(1 row)

bpfinal=#

在这里，我们同时使用了乘法运算符和求余的运算符来列出哪些以.99结尾的售价的项目。

我们还可以使用正则表达式进行匹配：

bpfinal=# SELECT \* FROM item WHERE description ~\* '^[PR].\*E$';

item\_id | description | cost\_price | sell\_price

---------+---------------+------------+------------

2 | Rubik Cube | 7.45 | 11.49

5 | Picture Frame | 7.54 | 9.95

(2 rows)

bpfinal=#

在这里，运算符执行了一个不区分大小写的正则表达式匹配来寻找那些描述信息由小写或者大写的p或者r开头并且由e结尾的项目。

就像你看到的，PostgreSQL支持挺多运算符的。事实上，如果算上相同运算符的不同变化（也就是，认为比较整数和比较浮点数或比较字符串是不同的），有差不多600个运算符。

### 运算符的优先级和结合性

PostgreSQL的很多运算符的外观和行为与你在很多编程语言中看到的算数运算符非常相近。这些运算符在分析器中有硬编码的优先级来确定通过运算符结合的表达式中它们执行的顺序。通常，可以使用括号来覆盖优先级。

PostgreSQL允许在SELECT的WHERE从句之外使用运算符和函数：

bpfinal=# SELECT 1+2\*3;

?column?

----------

7

(1 row)

bpfinal=# SELECT (1+2)\*3 AS answer;

answer

--------

9

(1 row)

bpfinal=#

这里，我们可以看到表达式1+2\*3的结果为7，默认情况下，无名列名以?column?的形式显示。在第二个示例中，运算符的优先级被覆盖且结果的被命名为answer。

虽然大多数运算符的行为和你期望的完全一样（如果你有C或者其他编程语言的开发经验），但有一些运算符的优先级规则可能会有些古怪。在C中，布尔运算符的优先级低于算数运算符，所以通常需要使用括号来获取正确的运算符执行顺序。如果可能出问题，尽量显示地使用括号。

PostgreSQL的运算符也显示了其结合性，可以是左结合或者右结合，用来确定相同优先级的运算符的取值顺序。算数运算符的例如加减都是做结合的，也就是，1+2-3取值的顺序和(1+2)-3相同。其他的，例如布尔等于运算符，是右结合的，也就是说x=y=z的取值为x=(y=z)。

表10-1列出了大多数常用的PostgreSQL的语法优先级（降序）。

表10-1 PostgreSQL运算符的优先级

|  |  |  |
| --- | --- | --- |
| 运算符 | 结合性 | 意思 |
| ::  [] | 左 | 类型转换（与CAST相同）  数组选择 |
| . | 左 | 对象（模式，表，列）选择 |
| - | 右 | 一元减号（负数） |
| ^ | 左 | 幂 |
| \* / % | 左 | 乘法运算符 |
| + - | 左 | 加减运算符 |
| IS  IS NULL  NOT NULL  OR | 左 | 测试（TRUE，FALSE，UNKNOWN和NULL）  测试（NULL）  测试（非NULL）  逻辑或 |
| IN  BETWEEN  LIKE ILIKE SIMILAR  <>  = | 右 | 测试是否为一个集和的成员  测试是否包含在一个范围内  测试字符串匹配  测试不等于  测试等于 |
| NOT | 右 | 逻辑非 |
| AND | 左 | 逻辑与 |
| 所有其他的运算符 |  | 不在这里列出的用户定义的和内置的运算符有相同的优先级 |

注：旧版本的PostgreSQL的运算符支持计算自然对数和反对数（使用:和;），但这些已经被废弃并从最近的PostgreSQL发行版中移除，包括8.0版本。使用ln和exp函数代替。

### 数学运算符

PostgreSQL提供大量的数学运算符。最常用的数学运算符在表10-2中列出。它们都是左结合的。

表10-2 常用的PostgreSQL数学运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 示例 | 意思 |
| + | 2 + 3为5 | 加法 |
| - | 3 - 2为1 | 减法 |
| \* | 2 \* 3为6 | 乘法 |
| / | 3 / 2为1  3 / 2.0为1.5  3 / 2::float8为1.5 | 除法 |
| % | 22 % 7为1 | 求余 |
| ^ | 4 ^ 3为64 | 求幂 |
| & | 14 & 23为6 | 二进制AND |
| | | 14 | 23为31 | 二进制OR |
| # | 14 # 23为25 | 二进制XOR |
| >> | 128 >> 4为8 | 右移位 |
| << | 1 << 4为16 | 左移位 |

还有一些一元数学运算符，在表10-3中列出。

表10-3 常用的PostgreSQL数学运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 示例 | 意思 |
| ! | 4!为24 | 阶乘 |
| !! | !!4为24 | 阶乘（左运算符） |
| @ | @(-2)为2 | 绝对值 |
| |/ | |/64为8 | 平方根 |
| ||/ | ||/64为4 | 立方根 |
| ~ | ~15为16 | 二进制NOT |

通常，数学运算符的运行效果和预期的相同。PostgreSQL将使用和参数匹配的版本相同的运算符。所以当你使用一个整数进行除法运算，你将得到一个整数。当你使用浮点数进行除法运算，你将得到一个浮点数结果。就像表10-2显示的，要强制获得浮点数结果，其中一个参数应该被转换为一个浮点数。

### 比较和字符串运算符

PostgreSQL提供了一组常用的比较运算符，例如小于和大于。这些运算符可以用在PostgreSQL支持的很多类型上，所以例如，你可以使用大于运算符来通过字母顺序作为相对大小来比较字符串的大小。

比较运算符的结果为true或者false，psql用t或f显示。可用的比较运算符在表10-4中给出。

表10-4 PostgreSQL的比较运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 示例 | 意思 |
| < | 2 < 3  'axy' < 'azz' | 小于 |
| <= | 2 <= 3 | 小于或等于 |
| <>  != | 2 <> 3  2 != 3 | 不等于 |
| = | 3 = 1 + 2 | 等于 |
| > | 3 > 2 | 大于 |
| >= | 3 >= 2 | 大于或等于 |

在PostgreSQL中，字符串有它自有的一套运算符。有一些用于连接字符串和通过一些规则匹配字符串的运算符，在表10-5中总结。

表10-5 PostgreSQL的字符串运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 示例 | 意思 |
| || | 'abc' || 'def' 为 'abcdef' | 连接字符串 |
| ~~ | 'xyzzy' ~~ '%zz%' | LIKE的同义词 |
| !~~ | 'xyzzy' !~~ '%aa%' | NOT LIKE的同义词 |
| ~ | 'xyzzy' ~ 'y.\*y' | 正则表达式子串匹配；使用^开始或者$结尾来表示开始配或者结束匹配。 |
| ~\* | 'xyzzy' ~\* '^X.\*Y$' | 正则表达式匹配，不区分大小写 |
| !~ | 'xyzzy' !~ 'aa' | 不匹配（~的相反值） |
| !~\* | 'xyzzy' !~ 'AA' | 不匹配，不区分大小写（~\*的相反值） |

在正则表达式匹配中，字符串和正则表达式的匹配类似于使用UNIX的grep工具或者Perl的匹配运算符。

### 其他运算符

PostgreSQL提供大量的附加运算符来比较和操作PostgreSQL特有的数据类型例如points，circles，time intervals和IP地址。请参考PostgreSQL文档的第二章获得更多信息。

提示：PostgreSQL文档可以以HTML页面的方式安装，然后在本地通过浏览器打开 file://usr/local/pgsql/doc/html/index.html来阅读，或者在<http://www.postgresql.org>在线阅读。

所有的运算符都存放在数据库的pg\_operator表中，psql可以通过内部命令\do和\df来列出所有的运算符和函数。

## 内置函数

PostgreSQL提供了大量的内置函数，我们可以在SELECT表达式中使用它们。

提供的函数大体分为以下几类：

* 和上一节提供的运算符相同功能的函数
* 其他的数学函数
* 其他用于处理字符串中的字符的函数
* 处理日期和时间的函数
* 文本格式化的函数
* 处理PostgreSQL地理类型例如point和circle的函数
* IP地址的函数

内置函数（和用户自定义函数）被记录在PostgreSQL数据库的系统表pg\_proc中。在8.0和以后的版本中，这个表中有超过1700项。在psql中，你可以使用\df命令来列出所有函数和它的参数。你也可以使用\dd命令来显示指定的函数或者函数分组的注释。

很多内置的函数提供一个和每个数学和逻辑运算符相对应的函数。例如包含int4shl和float8mul对应了给integer的左移位运算符（<<）和浮点值的乘法运算符（\*）。表10-6列出了用于浮点数且返回一个浮点数的数学函数，除非另有说明。

表10-6 常用的PostgreSQL数学函数

|  |  |
| --- | --- |
| 函数 | 意思 |
| abs(x) | 绝对值 |
| degrees(r) | 弧度转化为角度 |
| radians(d) | 角度转换为弧度 |
| exp(x) | 自然反对数，以自然对数e为底的指数 |
| ln(x) | 自然对数 |
| log(x) | 以10为底的对数 |
| log(b,x) | 以b为底的对数 |
| mod(x,y) | x除y后的余数（还有个整数版本） |
| pi() | 返回π |
| pow(x,y) | 求x的y次方 |
| random() | 返回一个0和1.0之间的随机数 |
| round(x) | 四舍五入到最近的整数 |
| round(x,d) | 四舍五入到指定的小数位d（可以为负数哦） |
| trunc(x) | 取整到整数（相对于0） |
| trunc(x,d) | 取整到指定的小数位d |
| ceil(x) | 返回不小于给定数的最小整数 |
| floor(x) | 返回不大于给定数的最大整数 |
| sqrt(x) | 平方根 |
| cbrt(x) | 立方根 |
| float8(i) | 将一个整数转换为一个float8 |
| float4(i) | 将一个整数转换为一个float4 |
| int4(x) | 获取整数，四舍五入 |

也支持三角函数，列在表10-7中。所有的角参数和结果都是弧度。

表10-7 PostgreSQL的三角函数

|  |  |
| --- | --- |
| 函数 | 意思 |
| sin | 正弦 |
| cos | 余弦 |
| tan | 正切 |
| cot | 余切 |
| asin | 反正弦 |
| acos | 反余弦 |
| atan | 饭正切 |
| atan2 | 两个参数的饭正切，计算atan(a/b) |

PostgreSQL包含了标准SQL字符串函数，但有它自己的语法。这些函数列在表10-8中。对于这些函数，字符串可以是char，varchar或text。PostgreSQL使用自有的附加函数扩展了字符串管理功能。请参考PostgreSQL的文档获得更多信息。

表10-8常用的PostgreSQL字符串函数

|  |  |
| --- | --- |
| 函数 | 意思 |
| char\_length(s)  character\_length(s) | 字符串长度 |
| octet\_length(s) | 存储字符串需要的长度 |
| lower(s) | 转换为小写 |
| upper(s) | 转换为大写 |
| position(s1 in s2) | s1出现在s2中的位置 |
| substring(s from n fro m) | 从s的位置n开始获取长度为m的子串 |
| trim([leading | trailing | both] [s1] from s2) | 从s2中去除子串s1，可以是从开始、结尾或者同时，如果没给出s1则默认移除空白 |

一个在这里需要提及的重要的格式化函数是to\_char函数。它在PostgreSQL中扮演着C中的printf的角色，处理所有用于打印或者显示的格式化的值。它将通过一个日期模板格式化一个日期和时间值，而且可以使用很多不同的方法格式化数字值（包括罗马数字）。

注：想了解更多的关于PostgreSQL所有函数的信息，参考PostgreSQL的文档或者浏览PostgreSQL发布包源码中的回归测试代码。

## 过程语言

就像在本章介绍部分提及的，我们可以在PostgreSQL数据库中使用定义自己的函数。这对于我们想进行特殊的计算或查询并在很多地方重用非常有用。

新建函数的SQL是CREATE FUNCTION，使用以下的基本语法：

CREATE FUNCTION name ( [ ftype [, ...] ] )

RETURNS rtype

AS definition

LANGUAGE 'langname'

函数定义部分可以不按照这个顺序，一个流行的做法是在定义部分之前给出语言类型，就像这样：

CREATE FUNCTION name ( [ ftype [, ...] ] )

RETURNS rtype

LANGUAGE 'langname'

AS definition

注：有另一种CREATE FUNCTION允许编译的对象代码并入PostgreSQL服务器中，通常是用C语言编写的。

一个非常简单的用于将一个参数增加一的函数可以这样写：

CREATE FUNCTION add\_one(int4) RETURNS int4 AS '

BEGIN

RETURN $1 + 1;

END;

' LANGUAGE 'plpgsql';

函数的定义是一个可以被拆分成多行且可以用任何PostgreSQL支持的可加载过程语言编写的字符串。在这里，使用的是PL/pgSQL，在LANGUAGE从句中用plpgsql表示。PL/pgSQL是一种为在PostgreSQL中编写存储过程而特别开发的编程语言。

注：可加载过程语言PL/Tcl，PL/Perl和PL/Python允许你分别使用Tcl，Perl和Python编程语言建立PostgreSQL的扩展。

要处理一个过程语言，PostgreSQL必须首先通过包含一个处理函数（通常是用C编写的）来进行扩展。对于PL/pgSQL，处理程序包含在在作为一个共享库的发布包中。

当一个函数被建立，它的定义将被保存在数据库中。当函数第一次被调用的时候，定义内容被处理函数编译为可执行格式并执行。这意味着我们无法获得我们函数中的错误的提示，直到我们尝试使用它。

在我们选择可加载的语言来写自己的函数前，我们必须让PostgreSQL支持这种语言。这将是我们下一步需要为PL/pgSQL做的。

### PL/pgSQL入门

在本章，我们将使用PL/pgSQL作为可加载过程语言来编写示例存储过程。在标准的PostgreSQL安装中，PL/pgSQL的处理函数被包含在Unix和Linux的/usr/local/pgsql/lib目录中的plpgsql.so中，或者Microsoft Windows下的安装目录下的\lib下的plpgsql.dll中。

服务器上的每个PostgreSQL数据库都有自己的一个过程语言列表。当我们安装一个过程语言时，我们可以选择哪个数据库需要用这种语言运行存储过程。某种程度上说，这是一种安全策略，因为有可能创建的函数无意或恶意的占用CPU资源，例如无限循环。这可能形成拒绝服务（DoS）攻击，使服务器太忙不能响应新的请求。因此，默认情况下，PostgreSQL数据库没有安装过程语言。要使用PL/pgSQL，我们需要自己安装处理程序。

注：数据库管理员可以添加语言到template1数据库中，这样，所有新建的数据库都将默认拥有这种语言。

要在我们的bpfinal数据库中安装PL/pgSQL，我们可以通过psql使用CREATE LANG命令并加载共享库，显式地建立处理函数。这复杂到需要提供一个帮助脚本来完成，所以PostgreSQL提供了一个工具。需要用的命令是createlang：

createlang [options] langname dbname

相关的选项在表10-9中列出。

表10-9 createlang命令的选项

|  |  |
| --- | --- |
| 选项 | 意思 |
| -h, --host=hostname | 指定数据库服务器主机 |
| -p, --port=port | 指定数据库服务器端口 |
| -U, --username=username | 指定连接的用户名 |
| -W, --password | 提示密码 |
| -d, --dbname=dbname | 设置需要安装语言的数据库 |
| -L, --pglib=directory | 在directory中搜索语言解释文件 |
| -e, --echo | 显示发送到服务器的命令 |
| -l, --list | 列出所有当前安装的语言 |

普通用户无法往数据库添加语言，所以我们通常使用postgres超级用户连接：

$ createlang -U postgres plpgsql bpfinal

我们可以使用createlang命令检查语言是否已存在，或者使用psql或者pgAdmin查询系统表pg\_language：

$ createlang -l bpfinal

Procedural Languages

Name | Trusted?

---------+----------

plpgsql | yes

$ psql -d bpfinal

bpfinal=# SELECT \* FROM pg\_language;

lanname | lanispl | lanpltrusted | lanplcallfoid | lanvalidator | lanacl

----------+---------+--------------+---------------+--------------+---------------

internal | f | f | 0 | 2246 |

c | f | f | 0 | 2247 |

plpgsql | t | t | 17601 | 17602 |

sql | f | t | 0 | 2248 | {=U/postgres}

(4 rows)

bpfinal=#

注：createlang可以用于安装其他过程语言。例如，用于安装PL/Tcl的命令是createlang pltcl。

超级用户也可以通过psql执行DROP LANGUAGE命令来从数据库中移除语言支持：

bpfinal=# DROP LANGUAGE 'plpgsql';

DROP LANGUAGE

bpfinal=#

##### 尝试：建立第一个存储过程

我们现在已经准备好通过编写我们自己的函数来开始学习使用PL/pgSQL编写存储过程了。让我们从实现之前提到的add\_one函数开始检查是否所有东西都正常工作开始：

bpfinal=# CREATE FUNCTION add\_one (int4) RETURNS int4 as '

bpfinal'# BEGIN return $1 + 1; end;' language 'plpgsql';

CREATE FUNCTION

bpfinal=# SELECT add\_one(2) AS answer;

answer

--------

3

(1 row)

bpfinal=#

##### 解析

CREATE FUNCTION命令存储通过PL/pgSQL语言定义的add\_one函数到数据库。之后，用SELECT表达式执行它。注意PL/pgSQL的关键字例如BEGIN和LANGUAGE都是不区分大小写的，而且命令的布局也不影响。我们也可以这样定义这个函数：

bpfinal=# CREATE function

bpfinal-# add\_one(int4) RETURNS int4

bpfinal-# AS '

bpfinal'# BEGIN

bpfinal'# RETURN $1 + 1;

bpfinal'# END;

bpfinal'# '

bpfinal-# LANGUAGE 'plpgsql';

CREATE FUNCTION

bpfinal=#

### 函数重载

PostgreSQL把名称不同、参数个数不同或者参数类型不同的函数都视为不同的函数。如果有必要，我们可以建立另一个add\_one函数。来看看使用一个浮点数调用我们的add\_one函数会发生什么：

bpfinal=# SELECT add\_one(3.1);

ERROR: function add\_one(numeric) does not exist

HINT: No function matches the given name and argument types. You may need to add

explicit type casts.

bpfinal=#

PostgreSQL返回一个错误，因为无法找到使用浮点数作为参数的add\_one函数。

注：早期版本的PostgreSQL会通过自动类型转换将浮点数转换成整数来执行这个add\_one函数。3.1会被四舍五入到3，函数会返回4。

如果我们需要一个为浮点数增加值的函数，我们只需要定义另一个叫add\_one的函数：

bpfinal=# CREATE FUNCTION

bpfinal-# add\_one(float8) RETURNS float8

bpfinal-# AS '

bpfinal'# BEGIN

bpfinal'# RETURN $1 + 1;

bpfinal'# END;

bpfinal'# '

bpfinal-# LANGUAGE 'plpgsql';

CREATE FUNCTION

bpfinal=# SELECT add\_one(3.1);

add\_one

---------

4.1

(1 row)

bpfinal=#

这次，我们得到了想要的结果，因为PostgreSQL可以找到并执行相应版本的add\_one函数。这种行为，被称作函数重载，会非常有用，但是也会带来麻烦。为了保持函数唯一，我们必须通过指定他们的参数来参考它们。在本例中，有add\_one(int4)和add\_one(float8)两个函数可以参考。

提示：一个更方便的用来创建函数方式是编辑包含函数定义的脚本文件并使用psql的\i命令来读取他们。

### 列出函数

如果我们需要看加载到数据库中的函数的源码，我们可以使用psql的\df+内部命令或者查询用于保存存储过程的表pg\_proc：

bpfinal=# SELECT prosrc FROM pg\_proc WHERE proname = 'add\_one';

prosrc

--------

begin

return $1 + 1;

end;

begin

return $1 + 1;

end;

(2 rows)

bpfinal=#

### 删除函数

可以通过DROP FUNCTION函数来删除数据库中的函数。如果我们需要完全删除一个函数，必须确保从重载的函数中指定出正确的函数并逐个删除：

bpfinal=# DROP FUNCTION add\_one(int4);

DROP FUNCTION

bpfinal=# DROP FUNCTION add\_one(float8);

DROP FUNCTION

bpfinal=#

### 引号

使用PL/pgSQL编写存储过程的一个稍微有点复杂的问题是引号。由CREATE FUNCTION命令提供的整个函数的定义是一个单引号括起来的字符串。这意味着如果我们如果需要在函数定义中使用单引号，必须使用转义。我们通过使用两个连续的单引号来表示字符串中的一个单引号。

如果我们的过程使用一个用嵌入了转义了的单引号的用单引号括起来的字符串，我们也必须将他们转义。这可能最终导致使用连续的四个单引号（或更多）。这里有一个示例：

create function ... as ' ... return ''string with a single '''' in it ''; ...' ...

幸运的是，在PostgreSQL 8.0以及以后的版本中，提供了新功能美元符号引号。类似于Perl和UNIX/Linux的shell的工作方式，美元符号引号允许我们在开始和结束的引号之间选择使用一个字符串。通过选择一个合适的不会出现在我们存储过程中的字符串，我们不需要使用转义。美元符号引用指一个在$字符之间的长度为0到任意长度的字符串。所以，之前的示例看上去可以是这样：

create function ... as $$ ... return 'string with a single '' in it '; ... $$ ...

如果在我们的存储过程中出现了$$字符串，我们可以选择另一个美元符号引号：

create function ... as $WHAT$ ... return 'string with a $$ in it '; ... $WHAT$ ...

本章从这里开始的示例将使用美元符号引号。如果你还在使用8.0版本之前的PostgreSQL，我们建议你升级到最新版本。否则，你需要使用传统的引号。

## 解析一个存储过程

现在我们已经知道如何建立、执行和删除一个示例存储过程了，是时候进一步看看PL/pgSQL存储过程的结构的细节了。PL/pgSQL是一种块结构的语言，类似于Pascal或者C，包含变量定义和代码块。每一块有一个可选的标题；可以有一些变量声明，以及包含在BEGIN和END关键字之间的代码块。一个块的语法如下：

[<<label>>]

[DECLARE declarations]

BEGIN

statements

END;

注：PL/pgSQL不区分大小写。所有关键字和变量名都可以以任意大小写书写。

一个PL/pgSQL函数的定义是通过一个包含由引号（可以是单引号或者美元符号引号）括起来的代码块的CREATE FUNCTION语句定义的。

-- For all PostgreSQL versions

CREATE FUNCTION name ( [ ftype [, ...] ] )

RETURNS rtype

AS 'block definition'

LANGUAGE 'plpgsql';

-- For PostgreSQL 8.0 and later

CREATE FUNCTION name ( [ [arg] ftype [, ...] ] )

RETURNS rtype

AS $$

block definition

$$ LANGUAGE plpgsql;

在我们的示例中的定义部分，双破折号（--）引入了延伸到行尾的注释，它将在执行时被忽略。我们会很快学习使用注释。

### 函数参数

PL/pgSQL函数可以有零到多个参数，参数类型在函数名后的括号内给出。类型是PostgreSQL内建的类型，例如int4或者float8。所有的存储过程必须返回一个值，返回的类型由函数定义部分的RETURNS从句指定。

在函数体内，函数的参数依照被定义的顺序被称为$1，$2，以此类推。我们将会在稍后发现可以使用ALIAS定义来给参数命名。对于PostgreSQL 8.0，也可以在声明变量的时候给他们命名。

让我们研究下以下这个计算两个整数的浮点几何平均值的存储过程：

-- geom\_avg

-- get a geometric average of two integers

create function geom\_avg(int4, int4) returns float8 as $$

begin

return sqrt($1 \* $2::float8);

end;

$$ language plpgsql;

使用PostgreSQL 8.0的命名参数功能，我们可以这样定义这个函数：

-- geom\_avg

-- get a geometric average of two integers

create function geom\_avg(a int4, b int4) returns float8 as $$

begin

return sqrt(a \* b::float8);

end;

$$ language plpgsql;

注意我们转换了其中一个整数值到浮点值，因而我们确保传入了一个浮点的乘法结果值给sqrt函数。如果我们不这么做，我们将可能获得一个错误提示，告知sqrt(int4)不存在，当然在PostgreSQL 8.0版本之前可能会正常。在存在函数重载的情况下，使用类型转换将确保我们调用了正确版本的函数。

### 注释

就像你之前看到的，PL/pgSQL允许在函数定义部分添加注释。事实上有两种类型的注释：单行注释和块注释。标准的SQL单行注释由双破折号（--）表示。从双破折号开始直到行尾的内容都会被忽略：

-- This is a single line comment

create -- comments can

function -- come anywhere, and extend to the end of the line

块注释用于注释掉一大块介绍内容或者临时一段不需要的代码。语法和C和C++类似，被注释的内容被/\*和\*/包括：

/\*

This is a block comment used to describe

the use and behavior of the following function

\*/

create function blah() returns integer as $$

begin

/\* comment out call to func

func();

\*/

return 1;

end;

$$ language plpgsql;

块注释不能被嵌套，但你可以使用单行注释来避免块注释结束符被解析为块注释。

### 声明

PL/pgSQL函数可以声明用于函数的局部变量。每个变量可以是一个PostgreSQL内置类型、用户自定义类型或者对应到一个表的行的类型。

函数的变量声明可以放在函数定义部分或者函数函数块的DECLARE段。和块结构语言例如C和C++一样，声明的变量仅能用于对应的代码块以及包含的子块。内部块中声明名变量将隐藏对外部块中同名变量的访问：

DECLARE

n1 integer;

n2 integer;

BEGIN

-- can use n1 and n2 in here

n2 := 1;

DECLARE

n2 integer; -- hides the earlier n2

n3 integer;

BEGIN

-- can use n1, n2 and n3 in here

n2 := 2;

END;

-- n3 no longer available here

-- n2 still has value 1 here

END;

函数中所有的变量在被使用前必须被声明，除了我们之后会碰到的循环控制变量。变量名不能是PL/pgSQL的保留关键字。表10-10理出了PostgreSQL的PL/pgSQL的关键字：

表10-10 PostgreSQL的PL/pgSQL的关键字

|  |  |  |  |
| --- | --- | --- | --- |
| alias | assign | begin | close |
| constant | cursor | debug | declare |
| default | diagnostics | dotdot | else |
| elsif | end | exception | excute |
| exit | fetch | for | from |
| get | if | in | info |
| into | is | log | loop |
| next | not | notice | null |
| open | or | perform | raise |
| rename | result\_oid | return | return\_next |
| reserve | row\_count | select | then |
| to | type | when | warnning |
| while |  |  |  |

有很多方法来声明一个变量，这依赖于它的用法。这些声明将在下文讨论。

#### 别名（ALIAS）

最简单的声明访问是别名(ALIAS)，用于给某个函数参数一个名字。这有助于让我们的代码更有意义，而且代码在针对在修改参数数量或者顺序的时候显得更健壮。ALIAS声明的语法为：

name ALIAS FOR $n;

于是有了一个叫name的变量，它可以作为指定位置的参数的另一个名字使用。例如，我们可以改写geom\_avg函数为这样：

create function geom\_avg(integer, integer) returns float8 as '

declare

first alias for $1;

second alias for $2;

begin

return sqrt(first \* second::float8);

end;

' language 'plpgsql';

就像之前提及的，从PostgreSQL 8.0开始，我们可以在函数声明的时候为给定位置的参数建立别名。以下的定义和上面这个相同：

create function geom\_avg(first integer, second integer) returns float8 as $$

begin

return sqrt(first \* second::float8);

end;

$$ language plpgsql;

#### 重命名（RENAME）

也可以通过RENAME来重命名变量。这对于之后将提用到的触发器函数中非常有用，但通常情况下不建议使用，因为它会使代码很难懂。RENAME声明的语法如下：

RENAME original TO new;

#### 简单的变量声明

简单的变量声明可以通过给定名字，类型以及可选的初始值来实现。以下为语法：

name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } value];

CONSTANT修饰符声明变量不能被改变。因此它必须在声明时给出初值。

NOT NULL从句会让变量在被赋一个NULL值的时候让PostgreSQL抛出一个运行时错误。

初始值可以不是常量，他可以在每次函数被调用或进入者变量定义所在的块的时候被计算。例如，给一个timestamp类型的变量的初始值为now，结果是变量获得的时间是函数执行的时间，而不是被编译的时间。

类型可以是一个PostgreSQL内置的类型；也就是，你可以声明和另一个数据库项目相同的数据类型或者结构的变量。这样间接指定变量类型的好处是即使数据库发生了变动，存储过程还可以保持正确。语法如下：

builtintype

variable%TYPE

table.column%TYPE

以下有一些变量声明的示例：

n integer := 1;

mypi constant float8 := pi();

pizza\_pi mypi%TYPE;

mydesc item.description%type := 'extra large size pizza';

在示例中，mydesc的声明的结果是选择合适的类型用于处理当前数据库中的item表的description列的类型。如果这个列在声明变量时的定义为char(64)，而之后改为了char(80)，使用mydesc的代码仍然能够工作，PostgreSQL将建立正确的变量类型。

注：我们在这个示例中用的是双美元引号。如果使用单引号，我们需要使用两个单引号括起来的字符串来初始化mydesc，因为这个声明出现在由单引号括起来的函数定义字符串中。

#### 复合变量的声明

复合变量值对应到一个特定表的完整行的变量。它有对应到表的每个列的字段。我们可以在我们的存储过程中声明和使用复合变量，可以是rowtype或者record。

要声明一个符合变量，我们使用rowtype声明，语法如下：

name table%rowtype;

这个声明将得到一个有字段的变量，包含给定的表的每一个列。例如：

contact customer%rowtype;

将建立一个叫contact的变量，它拥有关联到customer表的所有列的字段。要使用字段，我们可以使用variable.field这样的语法。以下为一个示例代码段：

DECLARE

contact customer%rowtype;

address text;

BEGIN

contact.zipcode := 'XY1 6ZZ';

contact.fname := NULL;

address := contact.addressline || contact.town;

...

END;

第二种复合变量声明的方法是record类型。这是一种和rowtype非常类似的类型，但它在定义的时候不是基于某个的表的。这种record类型将拥有运行时赋给它的任何结构。Record在被不同表调用的触发器中非常有用，例如在一个通用的用于记录行被删除的存储过程中非常有用。它通常也可以用于存储SELECT语句的结果。Record声明非常简单：

name record;

我们将在讨论通过选择赋值和触发器部分了解更多关于record的内容。

### 赋值

PL/pgSQL的变量通过赋值语句获得新值。赋值语句的语法如下：

reference := expression;

reference是一个变量名或者一个复合类型的一个字段，例如rowtype或者record。表达式可以是静态的、另一个变量或者一个由运算符、类型转换和函数调用组成的复杂表达式的字段。以下是一些赋值的示例：

n1: = 23;

long\_variable\_names\_are\_OK := (n1 + 45)/2;

f2 := add\_one(n1)::float8 \* sqrt(2.0);

/\* Composite types may be assigned one field at a time

by referring to individual fields: \*/

contact.zipcode := 'AB12 3CD';

#### SELECT INTO语句

另一种赋值机制是对SELECT的扩展。我们可以使用SELECT INTO语句赋予一个变量，一个变量的列表，一整行变量或者一个记录。语法是对普通SELECT的扩展：

SELECT expressions INTO target [FROM ...];

以下是一些用SELECT来代替赋值运算符的简单示例：

SELECT sqrt(2.0) INTO sqrt2;

SELECT add\_one(n1) INTO n1;

SELECT 1,2,3,4 INTO n1, n2, n3, n4;

SELECT 'Mole', 'Adrian' INTO contact.lname, contact.fname;

如果我们为每个里的值指定了正确的顺序，我们可以一次性赋值给整个rowtype：

DECLARE

product item%rowtype;

BEGIN

select NULL, 'Widget', '1.45', '1.99' into product;

END;

当变量和赋的值的类型不同的时候，PostgreSQL将执行可以进行的合适的类型转换。在上面的示例中，成本和售价的类型都是numeric(7,2)，但是它们都能成功地被赋予字符串类型的值。

赋值都会被PostgreSQL服务器作为SELECT语句执行，即使是使用了:=运算符格式。我们可以通过使用带有FROM从句和可选的WHERE条件的强大的SELECT语句来基于数据库的内容进行赋值：

SELECT \* INTO product FROM item WHERE item\_id = 9;

我们必须小心SELECT语句只会返回一行，因为其他更多的行都会被悄悄地忽略掉；只有第一行被返回并赋给列出的变量。我们将在不久后看到我们可以安排代码来处理SELECT返回的多行数据。

也有可能SELECT不返回数据；这种情况下，赋值将不会被执行。要检测到这种情况，我们可以使用PostgreSQL特有的布尔变量，叫做FOUND，它在使用SELECT INTO时被赋值：

SELECT \* INTO product FROM item WHERE description ~~ '%Cube%';

IF NOT FOUND THEN

-- take some recovery action

END IF;

#### PERFORM语句

某些情况下，我们不需要获取SELECT语句的结果集，比如有可能是用于调用一个函数。这种情况下，我们可以计算表达式或查询，并放弃与执行的结果：

PERFORM query;

PERFORM语句实际上执行了SELECT查询并忽略了结果集。

### 执行控制结构

PL/pgSQL提供用于控制函数中执行流程的结构。他们包括返回，条件分支和循环。

#### 从函数返回

从函数返回值可以通过RETURN语句实现：

RETURN expression;

函数的处理流程在执行到这个表达式的时候停止。表达式的值将作为函数的结果传给调用者。值必须和函数声明的返回类型兼容，且它在有必要的情况下进行类型转换。

在PL/pgSQL中，函数必须返回值，如果在函数最外层代码块结束的时候没有执行RETURN，则会产生一个运行时错误。

#### 异常和消息

如果一些不可能继续的条件出现，函数的执行也可以被停止。函数可以抛出一个异常，而不是返回一个值。异常会导致PostgreSQL在日志中写入一个项目，并让存储过程立即终止。RAISE语句可以根据几个严重级别记录异常。

RAISE level 'format' [, variable ...];

在表10-11中，列出了PostgreSQL定义的几个级别的异常。

表10-11 PostgreSQL的异常级别

|  |  |
| --- | --- |
| 级别 | 行为 |
| DEBUG，LOG，INFO | 在日志中写一条消息（仅此而已，通常被忽略掉） |
| NOTICE，WARNNING | 在日志中写一条消息，并发送日志到应用程序 |
| EXCEPTION | 在日志中写一条消息，并终止存储过程 |

DEBUG级别对于在卡法中获取附加信息非常有用。NOTICE级别提供非致命错误的警告。警告会通知到客户端应用程序，如果有必要，可以通过NOTIFY机制获取消息。EXCEPTION级别对于致命错误非常有用，这种情况下，存储过程无法继续。要获取更多的配置异常的行为的信息，请参考PostgreSQL的文档。

字符串format是用来格式化错误消息的。使用这个字符串的时候，每个%字符被按顺序由之后的变量替换。与C里头的printf不同，在RAISE语句中，你不允许使用表达式，只能是标识符。

看看以下这个例子：

RAISE DEBUG 'The value of n is %', n;

这会导致在PostgreSQL的日志文件（在linux中通常是 /usr/local/pgsql/data/postmaster.log）增加一条日志记录，内容看起来就像是这样：

DEBUG: The value of n is 4

这里有一个示例存储过程：

create function scope() returns integer AS $$

DECLARE

n integer := 4;

BEGIN

RAISE DEBUG 'n is %', n;

return n;

END;

$$ language plpgsql;

如果我们在psql中执行这个存储过程，将没有消息显示：

bpfinal=# SELECT scope();

scope

-------

4

(1 row)

bpfinal=#

如果我们将RAISE语句中的严重级别提升到NOTICE，我们将在psql中看到一个消息，这和日志文件中的一样：

bpfinal=# SELECT scope();

NOTICE: n is 4

scope

-------

4

(1 row)

bpfinal=#

最后，在EXCEPTION级别，存储过程一早就终止了，并返回一个错误：

bpfinal=# SELECT scope();

ERROR: n is 4

bpfinal=#

#### 条件

PL/pgSQL支持很多类型的条件，用于针对一个测试执行两个或者多个语句，或者返回两个或多个结果。也许这是PL/pgSQL最有用的部分。

##### IF-THEN-ELSE

最常见的条件语句是IF语句，和其他编程语言一样。语法如下：

IF expression

THEN

statements

[ELSE

statements]

END IF;

如果expression的结果为true，则IF语句的THEN部分被执行。否则，如果存在（可选的）ELSE部分，这部分代码将被执行。

和其他编程语言一样，IF语句可以在THEN或者ELSE部分中嵌入其他的IF语句。

##### NULLIF和CASE

也有两个SQL条件函数根据一个测试来返回值。它们是NULLIF和CASE，它们可以用于存储过程和PostgreSQL的常规SQL中。NULLIF函数在输入符合指定值的时候返回NULL，否则返回输入的值。它的语法如下：

NULLIF(input, value)

如果对input=value的检测为true，它返回NULL，否则返回input。

CASE函数根据输入的值选择一组值中的一个。语法如下：

CASE

WHEN expression

THEN expression

...

ELSE expression

END;

在必要的情况下，可能有好多对WHEN/THEN。整个表达式返回WHEN表达式运算结果为true的部分对应的THEN部分的表达式。如果没有匹配到任何WHEN部分，则获取到ELSE部分的表达式被计算。例如，以下的示例根据n2的值为1,2或者其他值，将res设置为5,6或者7：

res := CASE

WHEN n2 = 1

THEN 5

WHEN n2 = 2

THEN 6

ELSE 7

END;

#### 循环

PL/pgSQL有一套特别丰富的被称为迭代控制结构（iterative control structure）循环机制，用于提供方法将代码执行多次。

最简单的循环是不带控制条件的无限循环，循环会一直运行直到碰到一个EXIT语句：

[<<label>>]

LOOP

statements

END LOOP;

所有的PL/pgSQL的循环结构都可以打标签，也就是可以被BEGIN...END包括。这个标签用于作为EXIT语句的执行对象，用于中断指定的循环（对于熟悉C语言的读者，这就是C语言所缺乏的多级退出版本的break）：

EXIT [label] [WHEN expression];

这将导致用lable作标签的循环被终止，程序从这个循环后的语句继续执行。这个label必须指向当前的循环或者一个静态的循环。如果没有给出标签，则终止当前循环。如果给出了WHEN从句，则EXIT只会在expression的结果为true的情况下才被执行。

这里有一个无限循环的示例：

<<indefinite>>

LOOP

n := n + 1;

EXIT indefinite WHEN n >= 10;

END LOOP;

##### WHILE循环

WHILE循环是对于无限循环的一个替代结构。它在条件保持为true的情况下一直执行一堆语句。语法如下：

[<<label>>]

WHILE expression

LOOP

statements

END LOOP;

##### FOR循环

我们可以通过FOR循环执行固定次数：

FOR name IN [REVERSE] from .. to

LOOP

statements

END LOOP;

这种类型的循环为整数表达式from和to之间的每个值执行一次循环体。这个循环建立了一个叫name的新变量。它依次获得范围内的每个值，在循环体执行的过程中每次增加一。

如果指定了REVERSE，则循环以相反的方向运行，也就是变量每次减一。以下为一个FOR循环的示例：

FOR cid IN 1 .. 15

LOOP

SELECT \* INTO row FROM customer

WHERE customer\_id = cid;

-- process a customer

END LOOP;

循环执行了15次，变量cid取值从1到15，每次取一个用户到row变量中。循环的上下界限可以是表达式，所以我们可以扫描整个customer表，而不是仅仅扫描15个，就像以下这样：

SELECT COUNT(\*) INTO ncustomers FROM customer;

FOR cid IN 1 .. ncustomers

...

一个更好的可选方案是FOR循环允许我们为任意的一个SELECT语句的结果逐行执行：

FOR row IN SELECT ...

LOOP

statements

END LOOP;

对于SELECT返回的每一行，变量row被赋值且循环体被执行。用于存储这一行的变量必须被预先定义为record或者rowtype类型。除非循环被EXIT终止，否则在循环会执行到最好一行被处理。

以下为在psql中执行的一段用于打印所有客户的姓的代码段示例：

DECLARE

row record;

BEGIN

FOR row IN SELECT \* FROM customer

LOOP

RAISE NOTICE 'Family Name is %', row.lname;

END LOOP;

...

END

至此，我们遇到了所有的PL/pgSQL中的所有程序结构，是时候把它们拼凑到一起使用了。

##### 尝试：建立一个存储过程

假设我们想使用我们的示例数据库来在我们库存较低的时候帮助我们从供货商订货。我们已经有了stock表来追踪我们要销售的项目。我们想有能力使用这些信息来自动向我们的供货商发起订单。

以下为一个实现了自动补货目标的第一步的存储过程。以下的reorder函数扫描stock表中小于指定值的项目。它为这种项目写一条记录到新建的reorders表中。

-- Drop if necessary and create a temporary table for raising orders

drop table reorders;

create table reorders

(

item\_id integer,

message text

);

-- reorders

-- scan the stock table to raise reorders of item low on stock

create function reorders(min\_stock int4) returns integer as $$

declare

reorder\_item integer;

reorder\_count integer;

stock\_row stock%rowtype;

msg text;

begin

select count(\*) into reorder\_count from stock

where quantity <= min\_stock;

for stock\_row in select \* from stock

where quantity <= min\_stock

loop

declare

item\_row item%rowtype;

begin

select \* into item\_row from item

where item\_id = stock\_row.item\_id;

msg = 'order more ' ||

item\_row.description || 's at ' ||

to\_char(item\_row.cost\_price,'99.99');

insert into reorders

values (stock\_row.item\_id, msg);

end;

end loop;

return reorder\_count;

end;

$$ language plpgsql;

保存以上代码为sproc.sql。

当我们建立这个函数并以最小库存级别为3执行它，我们获得的结果为3，表示有当前库存量小于等于3的项目为3条：

bpfinal=# \i sproc.sql

DROP TABLE

CREATE TABLE

CREATE FUNCTION

bpfinal=# SELECT reorders(3);

reorders

----------

3

(1 row)

bpfinal=#

补货表reorders被库存量不足的三个项目的标识符填充：

bpfinal=# SELECT \* FROM reorders;

item\_id | message

---------+-------------------------------------

2 | order more Rubik Cubes at 7.45

5 | order more Picture Frames at 7.54

10 | order more Carrier Bags at .01

(3 rows)

bpfinal=#

我们可以通过查询stock表来验证：

bpfinal=# SELECT \* FROM stock;

item\_id | quantity

---------+----------

1 | 12

2 | 2

4 | 8

5 | 3

7 | 8

8 | 18

10 | 1

(7 rows)

bpfinal=#

##### 解析

函数reorder使用一个SELECT语句来获取stock表中所有的低库存量的行。一个LOOP用于迭代处理SELECT的结果。循环体使用一个INSERT为结果集的每一行添加数据到reorders表。函数reorders返回的结果则是通过另一个SELECT语句从stock表获取到的符合条件的记录数。

我们可以使用CREATE TEMPORARY TABLE来建立reorders表。这种情况下，这种情况下，表会在我们推出数据库会话的时候被删除。我们在这里使用CREATE TABLE因而reorders表是持久的，因为我们可能在的订单程序可能和我们的库存检查是不在一个程序中的。

要完成自动订单系统，我们还需要分解项目并将订单提交给供货商。我们也许需要为每个产品分别设立最小库存级别并添加到item表中。统样，每项产品需要的订单量也是不同的，也许是基于之前的销售记录或者季节性因素。

### 动态查询

通常，一个存储过程里头的数据库查询是固定的或仅仅是带参数的。我们通常根据指定的列从一个表查询符合条件的数据或者根据新的列的值插入或者修改一行。这个可以通过SELECT，INSERT和UPDATE语句使用保存符合条件的值或者列的值到变量中来实现：

INSERT INTO reorders VALUES (stock\_row.item\_id, msg);

但有一些特殊情况我们需要可以使用变量的值来指定一个表或者列名来实现一个操作。PostgreSQL不允许这样，因为它需要可以仅仅优化查询一次，而不是每次只需查询的时候都做优化。但是，PL/pgSQL支持一个通用的EXECUTE语句允许我们执行任意的SQL语句，通过给出一个字符串：

EXECUTE query-string

这个查询的字符串可以是存储过程中使用之前碰到的字符串运算符动态生成的。需要特别注意的是字符串中的用引号括起来的名字和文字需要保证正确。有两个函数帮助我们做到这点。所有的表名和列名应该使用quote\_ident处理，它能为查询生成合适的字符串。值应该使用quote\_value处理。

以下为一个使用名称和值的变量建立通用更新语句的示例：

EXECUTE 'UPDATE '

|| quote\_ident(tablename)

|| ' SET '

|| quote\_ident(columnname)

|| ' = '

|| quote\_literal(columnvalue)

|| ' WHERE '

...;

注意这是一种低效的访问数据库的方法，因为任何生成的查询在其被执行的时候都必须被分析和生成执行计划。

动态查询也可以在FOR循环中用在通过SELECT迭代访问记录时候使用。语法如下：

FOR row IN EXECUTE query-string

LOOP

statements

END LOOP;

## SQL函数

虽然本章专注于讲解使用PL/pgSQL作为建立存储过程的一种方法，但也有可能使用SQL来建立函数。要做到这点，我们只需要指定存储过程语言为’sql’并使用PostgreSQL的SQL语句代替PL/pgSQL。

和PL/pgSQL一样，SQL函数也可以有参数，被依次映射到$1，$2等。在函数定义中，$1自动被函数调用的第一个参数代替，$2被第二个代替，依此类推。没有控制结构，你被限制在使用PostgreSQL的SQL语句。PL/pgSQL包含一些功能例如变量，条件判断和循环，而SQL函数只允许使用参数替换。SQL函数返回的值是最后一条SQL语句执行的结果，通常是一个SELECT。

使用SQL写存储过程的好处是你不需要加载PL/pgSQL语言处理模块到数据库中。

虽然没有循环，以下的SQL函数允许我们从函数中返回不止一行数据。

CREATE FUNCTION sqlf(text) RETURNS setof

如果我们生命函数的返回类型为setop类型，并使用了适当的SELECT，我们可以实现返回多行。以下函数返回指定城镇中所有的客户：

CREATE FUNCTION sqlf(text) RETURNS setof customer AS $$

SELECT \* FROM customer WHERE town = $1;

$$ language sql;

当我们在psql中运行这个函数，将返回三行数据：

bpfinal=# SELECT sqlf('Bingham');

sqlf

------------------------------------------------------------------------

(7,"Mr ",Richard,Stones,"34 Holly Way",Bingham,"BG4 2WE ","342 5982"

(8,"Mrs ",Ann,Stones,"34 Holly Way",Bingham,"BG4 2WE ","342 5982")

(11,"Mr ",Dave,Jones,"54 Vale Rise",Bingham,"BG3 8GD ","342 8264")

(3 rows)

bpfinal=#

早于8.0版本的PostgreSQL的psql无法一次处理一整行数据，所以会返回一个错误号或者返回对应行的OID。所有的版本都允许我们选择一个列。要实现它，我们使用语法column(function())来获取需要的列。这里，我们列出Bingham的客户的名字并给结果列一个名字：

bpfinal=# SELECT lname(sqlf('Bingham')) AS customer;

customer

----------

Stones

Stones

Jones

(3 rows)

bpfinal=#

## 触发器

在我们的存储过程示例中，我们开发了一个用于发现哪些产品需要进货的函数。这个函数写了一些提示消息到reorders表中。要让它用起来，我们需要确保这个过程在某些条件下被执行，也许是在每天的定时批量任务中做。但是最好能够找到一种方法自动确保reorders表总是最新的，而不需要编写我们的客户端程序来不停更新其中的项目。

在第八章，我们碰到了参照完整性这个概念——确保我们数据库中的数据总是有意义。例如，如果我们删除一个客户，我们需要确保所有的跟这个用户相关的订单历史记录同时被删除。我们在之前已经看到了PostgreSQL中确保这种完整性的约束。

对于一些应用程序，仅有约束是不够的。假设我们想要避免删除仍然有未完成订单的客户，而允许删除所有的已经发送订单的客户。

我们在第八章看到了列级和表级的约束可以用于确保更复杂的数据完整性的规则，但这些规则实际上是静态的。我们可以指定某个相关的行必须存在，或者执行一个当相关的行存在时某些行必须存在的规则。然而，我们我们无法指定复杂的条件，例如当其他条件也为true的时候一个行必须不存在。我们甚至无法在行被添加或者删除的时候进行其他复杂的用户定义行为。

解决这个问题的方法之一是使用触发器。通过触发器，我们可以让PostgreSQL在发生一些行为例如INSERT，DELETE或者UPDATE一个表的时候执行一个过程。

过程和触发器的组合给我们直接在数据库中执行相当复杂的业务规则的能力。就像在第八章说的，最好的确保数据的业务规则的地方是在数据库中。

要使用触发器，我们需要先定义一个触发器过程。然后我们可以创建触发器，它定义了什么时候触发器过程将被执行。

### 定义触发器过程

触发器在符合条件的时候被触发，然后会执行一种特殊类型的存储过程，叫做触发器过程。触发器过程和存储过程非常类似，但由于其特殊的调用方式，其编写会更加受限。

触发器过程是一种没有参数且返回类型为特殊的trigger类型的函数。

注：7.3版本之前的PostgreSQL使用返回类型opaque而不是trigger。返回类型opaque返回值类型用于无法被PostgreSQL直接操作的函数返回值。

PostgreSQL会在特定表发生改变的时候调用触发器过程。这个过程必须要么返回NULL要么返回符合触发器过程被调用的表的结构的一个行。

对于在UPDATE之后调用的AFTER触发器，建议触发器过程返回NULL。对于BEFORE触发器，返回的结果会直接用于更新操作。如果触发器过程返回NULL，则UPDATE不会被执行。如果数据行被返回了，它将用作UPDATE的数据源，在数据提交到数据库前给触发器过程机会改变它。

### 建立触发器

使用命令CREATE TRIGGER命令建立触发器，语法如下：

CREATE TRIGGER name { BEFORE | AFTER }

{ event [OR ...] }

ON table FOR EACH { ROW | STATEMENT }

EXECUTE PROCEDURE func ( arguments )

在这里，event可以是INSERT，DELETE或者UPDATE。

触发器有效地说：“在这个表上发生这个事件的时候调用这个存储过程”。

需要给触发器一个名字，用于在不需要这个触发器的时候删除它，语句如下：

DROP TRIGGER name ON table;

触发器在某些事件发生时被触发——DELETE，INSERT或者UPDATE。我们可以要求触发器在事件发生后被触发；在这种情况下，被调用的过程将可以访问原始数据（被更新和删除）和新数据（被插入和更新）。我们也可以要求触发器在事件发生前被触发。这种情况下，我们可以避免更新操作的发送，或者修改被插入或者更新的数据。我们可以通过使用OR分隔的事件列表指定不止一个事件。

正如我们所知，一些SQL语句可以影响多行数据。当多行更新导致触发器被触发，我们可以选择是否为更新的每一行或者为整个更新触发一次触发器。我们通过指定ROW来执行多次，否则，我们指定STATEMENT。

传递给函数的参数，可以用来区分类似的触发器，所以一个函数可以用于多个触发器。

要自动更新我们的reorders表，我们需要建立一个叫做reorder\_trigger的存储过程，并建立一个在表stock表被变更的时候调用它的触发器：

CREATE TRIGGER trig\_reorder

AFTER INSERT OR UPDATE ON stock

FOR EACH ROW EXECUTE PROCEDURE reorder\_trigger(3);

注意触发器过程（本例中的reorder\_trigger）必须在建立触发器前被建立。

我们使用触发器过程参数来传递一个最小库存阈值，在本例中为3。

##### 尝试：建立一个触发器

以下为一个触发器过程用于在stock表发生调整的时候更新reorders表的示例：

create function reorder\_trigger() returns trigger AS $$

declare

mq integer;

item\_record record;

begin

mq := tg\_argv[0];

raise notice 'in trigger, mq is %', mq;

if new.quantity <= mq

then

select \* into item\_record from item

where item\_id = new.item\_id;

insert into reorders

values (new.item\_id, item\_record.description);

end if;

return NULL;

end;

$$ language plpgsql;

现在我们有了触发器过程，我们可以这样定义一个触发器：

CREATE TRIGGER trig\_reorder

AFTER INSERT OR UPDATE ON stock

FOR EACH ROW EXECUTE PROCEDURE reorder\_trigger(3);

从一个脚本文件中加载这个函数和触发器的定义：

bpfinal=# \i sproc.sql

...

CREATE FUNCTION

CREATE FUNCTION1;

CREATE TRIGGER

bpfinal=#

然后尝试调整一个项目的库存让它降低到3或者更少：

bpfinal=# UPDATE stock SET quantity = 3 WHERE item\_id = 1;

NOTICE: in trigger, mq is 3

UPDATE 1

bpfinal=#

我们可以发现触发器被触发并发出一个通知给我们。对库存的修改执行了，同时触发器也更新了reorders表，添加了一个新行：

bpfinal=# SELECT \* FROM reorders;

item\_id | message

---------+-------------

1 | Wood Puzzle

(1 row)

bpfinal=#

##### 解析

触发器过程在stock表中发生INSERT或者UPDATE操作的时候总是被触发。它在UPDATE操作发生后检查库存量，如果库存小于给出的最小阈值，则添加一条记录到reorders表中。

你可能注意到了我们在触发器过程中使用了一大堆新的功能。首先，触发器过程的参数不在是$1，$2一类的了。自动触发的过程是通过一大批特殊变量之一实现参数传递的。这个以数组方式传递的参数名为TG\_ARGV，从TG\_ARGV[0]开始。触发器过程使用的特殊变量在表10-12中列出。

其次，在触发器过程中，有两个非常特殊的记录：OLD和NEW。它们包含来自于（对于ROW类型的触发器）被UPDATE影响的行所触发的触发器所影响的行的数据。就像你猜到的，OLD包含UPDATE之前的数据，而NEW包含更新后的数据（或者为BEFORE类型的触发器所建议的UPDATE操作的数据）。

表10-12 PostgreSQL触发器过程变量

|  |  |
| --- | --- |
| 变量 | 描述 |
| NEW | 包含新数据库行的记录 |
| OLD | 包含旧数据库行的记录 |
| TG\_NAME | 包含触发这个触发器过程的运行的触发器名字的变量 |
| TG\_WHEN | 包含文本’BEFORE’或者’AFTER’的变量，依赖于触发器类型 |
| TG\_LEVEL | 包含文本’ROW’或者’STATEMENT’的变量，依赖于触发器的定义 |
| TG\_OP | 包含文本’INSERT’，’DELETE’或者’UPDATE’的变量，依赖于导致触发器被触发的事件 |
| TG\_RELID | 触发器被激活的表所关联的对象ID |
| TG\_RELNAME | 触发器被触发的表的名字 |
| TG\_NARGS | 整数型变量，包含在触发器定义的时候传入的参数个数 |
| TG\_ARGV | 字符串数组，包含过程调用的参数，索引值从0开始；如果传入无效的索引值则返回NULL |

##### 尝试：建立另一个触发器

作为最终版本的示例，我们可以建立另一个触发器函数，它将避免在客户还存在未完成的订单的时候被删除。我们为将要被删除的客户检查orderinfo表中的订单的date\_shipped列是否为NULL，如果是，则不允许删除。如果没有未完成的订单，我们则允许删除继续，但是我们需要通过删除客户的旧订单来整理订单，还有相关订单的订单项目信息。

create function customer\_trigger() returns trigger AS $$

declare order\_record record;

begin

-- about to delete a customer

-- disallow if orders pending

select \* into order\_record from orderinfo

where customer\_id = old.customer\_id

and date\_shipped is NULL;

if not found

then

-- all OK, delete of customer can proceed

raise notice 'deletion allowed: no outstanding orders';

-- for referential integrity we have to tidy up

-- we will need to delete all completed orders

-- but first delete the information about the orders

for order\_record in select \* from orderinfo

where customer\_id = old.customer\_id

loop

delete from orderline

where orderinfo\_id = order\_record.orderinfo\_id;

end loop;

-- now delete the order records

delete from orderinfo

where customer\_id = old.customer\_id;

-- return the old record to allow customer to be deleted

return old;

else

-- orders present return NULL to prevent deletion

raise notice 'deletion aborted: outstanding orders present';

return NULL;

end if;

end;

$$ language plpgsql;

create trigger trig\_customer before delete on customer

for each row execute procedure customer\_trigger();

要验证这个触发器的行为，让我们检查一把。首先，标记一个订单的发货日期为NULL，表明订单还在处理中：

bpfinal=# UPDATE orderinfo SET date\_shipped = NULL WHERE orderinfo\_id = 3;

UPDATE 1

bpfinal=# SELECT \* FROM orderinfo;

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping

--------------+-------------+-------------+--------------+----------

1 | 3 | 2004-03-13 | 2004-03-17 | 2.99

2 | 8 | 2004-06-23 | 2004-06-24 | 0.00

4 | 13 | 2004-09-03 | 2004-09-10 | 2.99

5 | 8 | 2004-07-21 | 2004-07-24 | 0.00

3 | 15 | 2004-09-02 | | 3.99

(5 rows)

bpfinal=#

现在尝试删除客户编号为15的客户，而其编号为13的订单还未完成：

bpfinal=# DELETE FROM customer WHERE customer\_id = 15;

NOTICE: deletion aborted: outstanding orders present

DELETE 0

bpfinal=#

由于存在未完成订单，会返回一个通知。我们会发现，删除不影响任何一行，因为不允许删除继续。

删除没有未完成订单的客户将没有任何问题，只是我们会先删除orderinfo表中项目。这是因为我们在这个表中使用了约束来避免这种单独的删除。触发器很好的处理了这个问题，并为我我们提供了另一种方法来实现参照完整性问题。

编号为3的客户没有未完成的订单，所以我们可以直接删除客户，而让触发器做其他的收尾工作：

bpfinal=# DELETE FROM customer WHERE customer\_id = 3;

NOTICE: deletion allowed: no outstanding orders

DELETE 1

bpfinal=#

检查相关表，我们会发现删除客户后，不会剩余多余的垃圾数据：

bpfinal=# SELECT \* FROM orderinfo;

orderinfo\_id | customer\_id | date\_placed | date\_shipped | shipping

--------------+-------------+-------------+--------------+----------

2 | 8 | 2004-06-23 | 2004-06-24 | 0.00

4 | 13 | 2004-09-03 | 2004-09-10 | 2.99

5 | 8 | 2004-07-21 | 2004-07-24 | 0.00

3 | 15 | 2004-09-02 | | 3.99

(4 rows)

bpfinal=# SELECT \* FROM orderline;

orderinfo\_id | item\_id | quantity

--------------+---------+----------

2 | 1 | 1

2 | 10 | 1

2 | 7 | 2

2 | 4 | 2

3 | 2 | 1

3 | 1 | 1

4 | 5 | 2

5 | 1 | 1

5 | 3 | 1

(9 rows)

bpfinal=#

bpfinal=# SELECT customer\_id, fname, lname FROM customer;

customer\_id | fname | lname

-------------+-----------+---------

1 | Jenny | Stones

2 | Andrew | Stones

4 | Adrian | Matthew

5 | Simon | Cozens

6 | Neil | Matthew

7 | Richard | Stones

8 | Ann | Stones

9 | Christine | Hickman

10 | Mike | Howard

11 | Dave | Jones

12 | Richard | Neill

13 | Laura | Hardy

14 | Bill | O'Neill

15 | David | Hudson

(14 rows)

bpfinal=#

## 为什么使用存储过程和触发器

有很多原因要求使用存储过程和触发器，以下列出了一些：

* 提供集中校验：我们可以在一个地方对表的更新进行强制条件检查，而不依赖我们的客户端应用程序。如果条件需要改变，只需要修改一个地方。
* 追踪改变：我们可以使用触发器来建立一个审计线索，在表被更新的时候写另外一个表。这可以记录造成改变的用户，时间，日期甚至改变的数据。
* 增强安全：通过使用PostgreSQL的current\_user变量，我们可以执行我们自己的安全策略。
* 防止删除：我们可以使用一个触发器来标记数据为需要在之后删除，而不是在应用程序尝试删除它的时候就删除掉。
* 为客户端提供映射操作：我们可以使用触发器和存储过程来为我们的某些数据建立一个简单的，单表的映射，让我们用户对它们的操作跟容易。例如，我们可以建立一个表并映射到Microsoft Excel。当这个表被更新，我们可以更新“真实”表中的数据，其中可能有个表有很复杂的结构。

## 摘要

在本章，我们看到了一些可以用来扩展PostgreSQL的查询的功能的方法。我们已经看到PostgreSQL提供了很多运算符和函数来完善查询和提取信息。

PostgreSQL支持的过程语言允许我们使用PL/pgSQL，SQL和其他语言开发相当复杂的服务端过程。这为数据库服务器提供了实现复杂的不依赖于客户端的应用功能的机会。

存储在数据库中的存储过程可以被应用程序调用，或者被触发器在数据库的表发生改变时被自动调用。这给了我们另一种方法来确保参照完整性。

对于简单的参照完整性，最好是坚持使用约束，因为它们更简单、高效，不容易出错。而触发器和存储过程在你声明的约束变得非常复杂的时候或者你想要实现的约束对于声明方式太复杂的时候会带来很大帮助。

到此为止我们已经介绍了一些先进的PostgreSQL的技术，在下一章，我们将前进到如何保障PostgreSQL数据库的主题。