

PostgreSQL 从入门到精通

翻译：洞庭湖的泥鳅

如有问题，请联系：

[bambo.huang\(at\)gmail.com](mailto:bambo.huang(at)gmail.com)

目录

引言	7
第一章 POSTGRESQL 介绍.....	8
基于数据编程	8
静态数据.....	8
用于数据存储的扁平文件.....	9
重复单元和其他问题.....	9
什么是数据库管理系统.....	10
数据模型.....	10
查询语言.....	13
数据库管理系统的责任.....	15
什么是 POSTGRESQL?	16
PostgreSQL 历史简介.....	17
PostgreSQL 架构.....	17
通过 PostgreSQL 访问数据	18
什么是开源?	19
相关资源	19
第二章 关系数据库原理.....	20
电子表格的局限性	20
将数据存入数据库.....	22
选择列.....	23
为每个列选择数据类型.....	23
标记行的唯一性	23
在数据库中访问数据	25
通过网络访问数据.....	25
处理多用户访问.....	26
数据分片和分块.....	26
增加信息.....	28
设计表.....	31
基本数据类型.....	36
处理未知的值：空值（NULL）	37
回顾示例数据库.....	38
摘要	38

第三章 初步使用 POSTGRESQL.....	40
在 LINUX 和 UNIX 系统中安装 POSTGRESQL	40
在 <i>Linux</i> 中使用二进制文件安装 <i>PostgreSQL</i>	40
通过源码安装 <i>PostgreSQL</i>	45
在 <i>Linux</i> 和 <i>Unix</i> 上配置 <i>PostgreSQL</i>	48
在 WINDOWS 中安装 POSTGRESQL.....	54
使用 <i>Windows</i> 安装程序	54
配置客户机访问.....	60
建立示例数据库	61
添加用户记录.....	61
建立数据库.....	62
建表.....	63
移除表.....	65
填充表.....	65
摘要	69
第四章 访问你的数据.....	70
使用 PSQL	70
在 <i>Linux</i> 系统中启动.....	70
在 <i>Windows</i> 系统中启动	71
解决启动问题.....	71
使用一些基本的 <i>psql</i> 命令	74
使用 SELECT 语句	75
覆盖列名.....	77
控制行的顺序.....	78
消除重复数据.....	80
执行计算	82
选择行	84
使用更复杂的条件.....	86
模式匹配.....	88
限制结果集.....	89
检查空值 (NULL)	90
检查时间和日期	91
设置时间和日期的风格.....	91
使用日期和时间函数.....	95
多个表协同工作	97
关联两个表.....	97

给表赋予别名.....	102
关联三个或更多的表.....	103
SQL92 的 SELECT 语法.....	108
摘要.....	109
第五章 POSTGRESQL 的命令行和图形界面工具	110
PSQL	110
启动 <i>psql</i>	110
在 <i>psql</i> 中输入命令	111
使用命令历史.....	112
在 <i>psql</i> 中执行脚本文件	112
检查数据库.....	114
<i>psql</i> 命令行快速参考	114
<i>psql</i> 内部命令快速参考.....	116
设置 ODBC.....	117
在 Windows 中安装 ODBC 驱动程序.....	118
在 Windows 中建立一个数据源.....	120
在 Linux/Unix 中安装 ODBC 驱动程序.....	122
在 Linux/Unix 中建立一个数据源.....	122
PGADMIN III.....	122
安装 <i>pgAdmin III</i>	122
使用 <i>pgAdmin III</i>	123
PHPPGADMIN	126
安装 <i>phpPgAdmin</i>	127
使用 <i>phpPgAdmin</i>	127
MICROSOFT ACCESS	130
使用链接表.....	131
输入数据及建立报表.....	133
MICROSOFT EXCEL.....	135
POSTGRESQL 相关工具的资源	138
摘要	139
第六章 数据交互.....	140
添加数据到数据库中	140
使用基本的 <i>INSERT</i> 语句.....	140
使用更安全的插入语句.....	143
插入数据到 <i>serial</i> 类型的列中.....	144
插入空值.....	148

使用copy 命令.....	149
直接从另一个程序加载数据.....	152
修改数据库中的数据.....	155
使用UPDATE 语句.....	156
通过另一个表更新.....	158
从数据库删除数据.....	159
使用DELETE 语句.....	159
使用TRUNCATE 语句.....	160
摘要.....	161
第七章 高级数据选择.....	162
聚集函数.....	162
count 函数.....	163
Min 函数.....	171
Max 函数.....	172
Sum 函数.....	173
Avg 函数.....	173
子查询.....	174
返回多行记录的子查询.....	176
相关子查询.....	178
存在子查询 (Existence Subqueries)	180
UNION 连接.....	182
自连接.....	184
外连接.....	186
摘要.....	190
第八章 数据定义.....	192
数据类型.....	192
布尔数据类型.....	193
字符数据类型.....	195
数字数据类型.....	197
时间型数据类型.....	200
数组.....	201
数据操作.....	204
在数据类型之间转换.....	204
用于数据操作的函数.....	205
魔法变量.....	206

OID 列.....	207
表管理	208
建表.....	208
使用列约束.....	209
使用表约束.....	213
修改表结构.....	214
删除表.....	218
使用临时表.....	218
视图	218
建立视图.....	218
删除和替换视图.....	222
外键约束	222
作为一个列的约束的外键.....	223
作为一个表的约束的外键.....	224
外键约束的选项.....	230
摘要	232

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。

第一章 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，我们碰到了一个问題。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。扁平文件在这应该是看起来像这样：

```
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**。我们现在的问題是重复的数据项（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題，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。解决这个问題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。

CountryName	Symbol	Rate	LangPtr
-------------	--------	------	---------

Language	LangPtr
----------	---------

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。

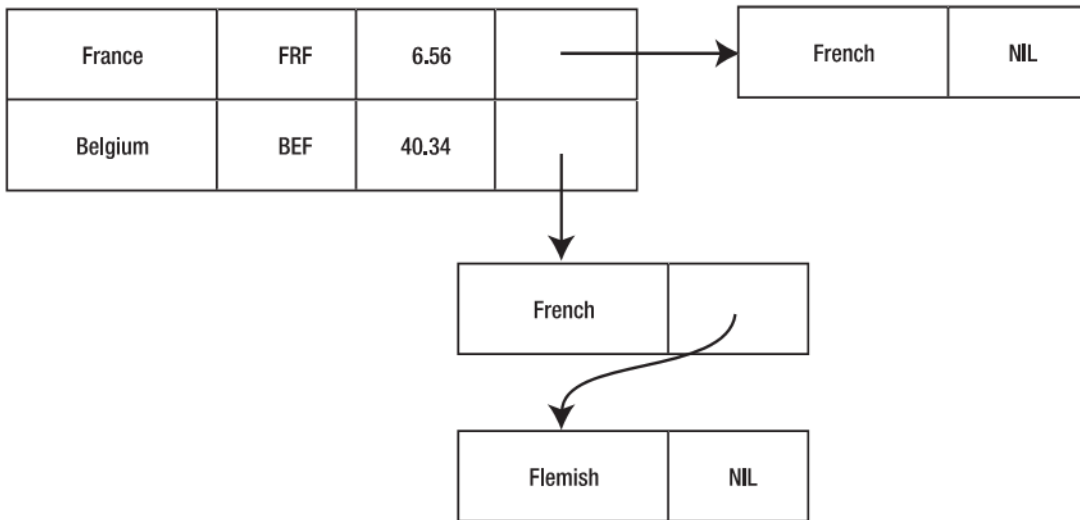


图 1-2 汇率程序数据结构

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，一个浮点值或其他数据库系统支持的类型。

用于从一个表中区分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做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他人在写入他需要读取（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。

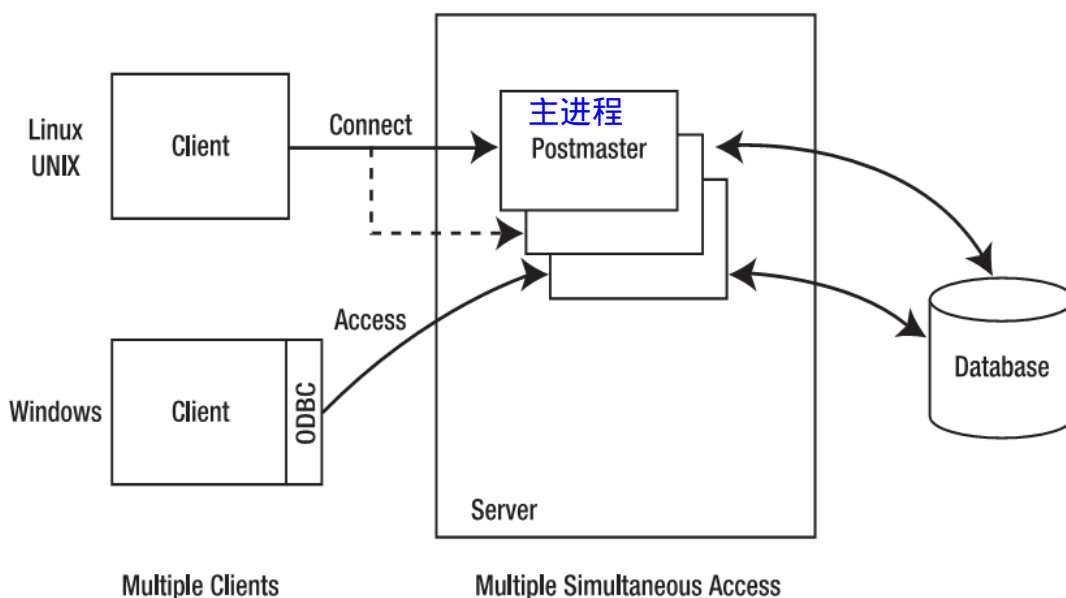


图 1-3 PostgreSQL 架构

在图 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”。PostgreSQL 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”。紧接着的段落是免责声明和保证书。

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 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>。

第二章关系数据库原理

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。

	A	B	C	D	E	F	G	H
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876	
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527	
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567	
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871	
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926	
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232	
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432	
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482	
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264	
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482	
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335	
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234	
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526	
16								
17								
18								
19								

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。一行和列的交集是一个单元格。

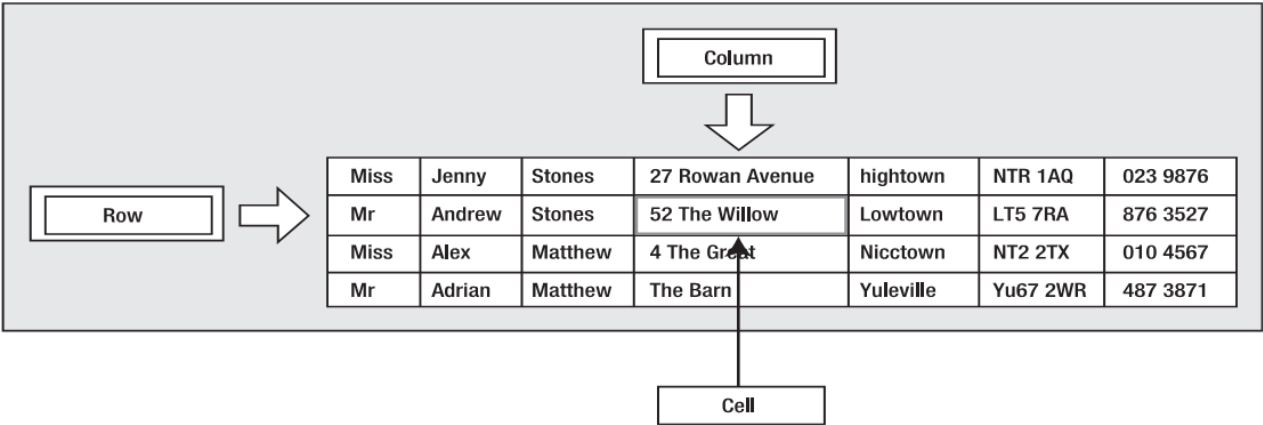


图 2-2 一些电子表格的术语

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

电子表格是一个了不起的主意，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。

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876		22 Jun 2004	\$15.30	25 Jul 2004	\$27.89	4 Oct 2
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527						
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567		2 Jun 2004	\$32.67	11 Jul 2004	\$23.65	18 Nov 2
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871		18 Jun 2004	\$56.32	4 Aug 2004	\$73.11	
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926						
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232						
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982		27 Jun 2004	\$32.34			
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982						
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432		12 Jun 2004	\$17.43	18 Jul 2004	\$32.54	
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482		12 Sep 2004	\$76.23			
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264						
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482						
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335						
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234						
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526		4 Nov 2004	\$12.45			
16													
17													
18													

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。

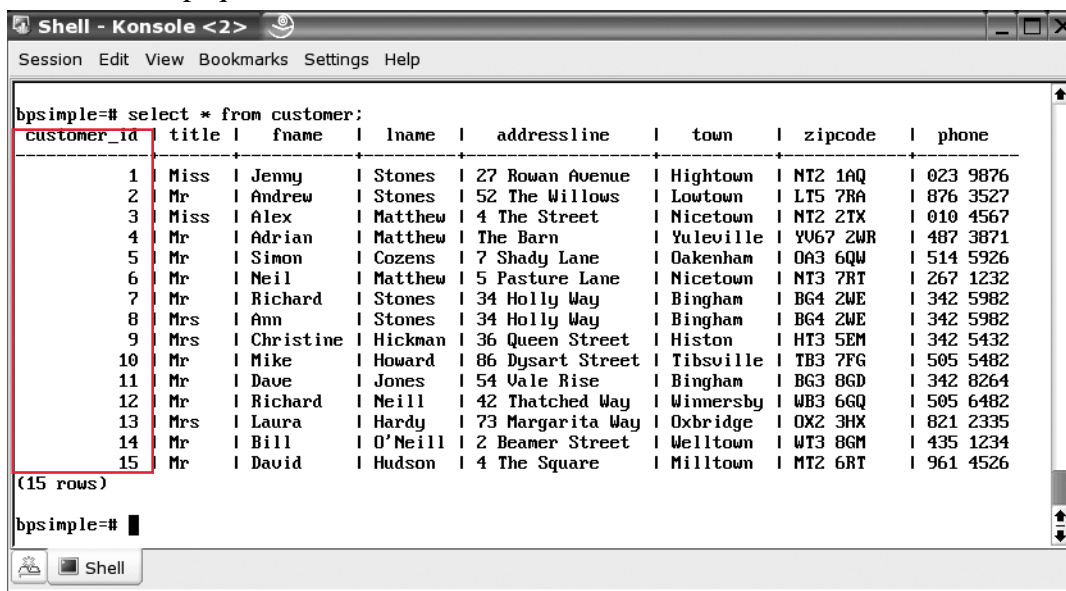
最明显的解决方案好像是通过客户名来区别客户，但不幸的是，这通常不足以区别。因为很有可能两个客户拥有同样的名字。另外一项你可以选择的是电话号码，但问题是如果两个客户住在一起呢？在这个时候，你可能建议使用名字和电话号码的组合。

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。



The screenshot shows a terminal window titled "Shell - Konsole <2>". Inside, a SQL query is executed: `bpsimple=# select * from customer;`. The result is displayed as a table with 9 columns: `customer_id`, `title`, `fname`, `lname`, `addressline`, `town`, `zipcode`, and `phone`. There are 15 rows of data. The `customer_id` column is highlighted with a red box. Below the table, it says "(15 rows)". At the bottom, the prompt `bpsimple=#` is shown.

customer_id	title	fname	lname	addressline	town	zipcode	phone
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
2	Mr	Andrew	Stones	52 The Willows	Lowntown	LT5 7RA	076 3527
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	021 2335
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526

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。

customer_id	title	fname	lname	addressline	town	zipcode	phone
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526

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

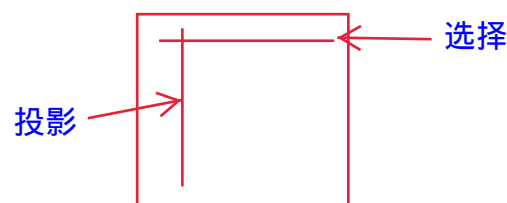
和所有的关系数据库一样，PostgreSQL 可以自动确保对数据库数据修改的冲突不会发生。它确保各个用户在使用数据的时候感觉好像访问全部的数据都不受限制，但在幕后，PostgreSQL 监视着改变且避免同时更新的冲突。

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。

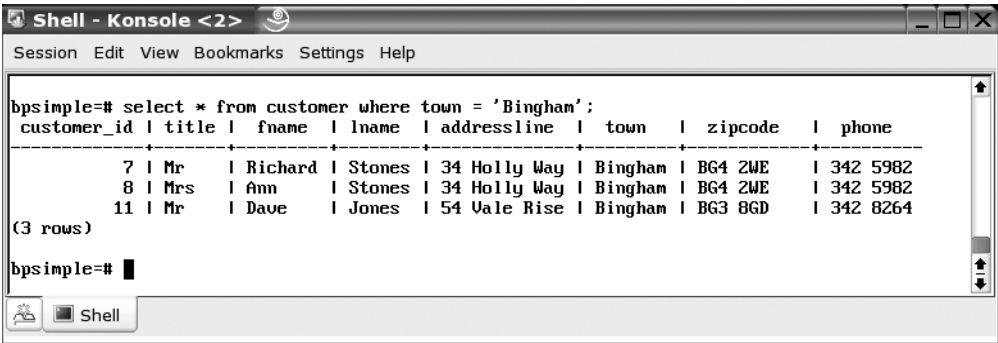


图 2-7 选择数据行的子集

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地回顾它。

投影

现在来看看投影，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：

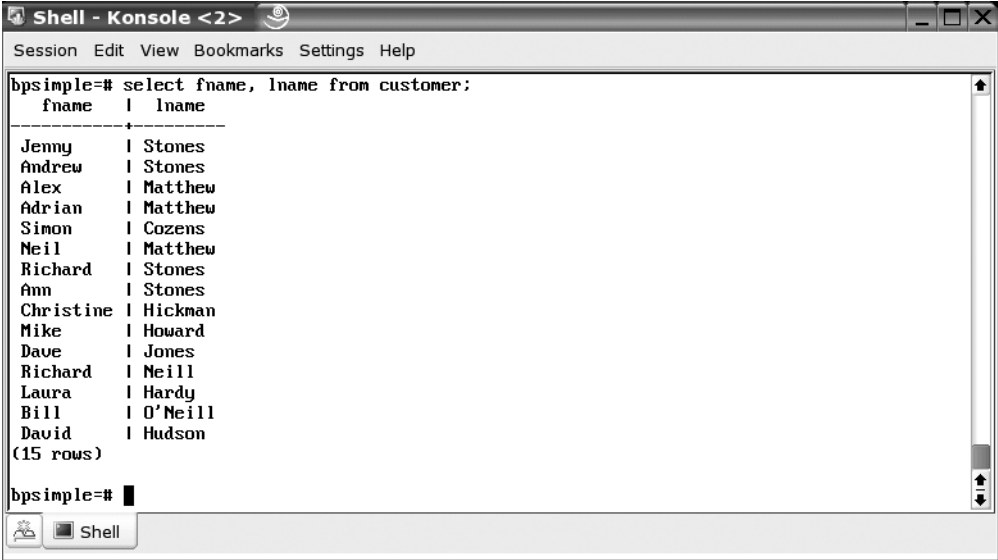
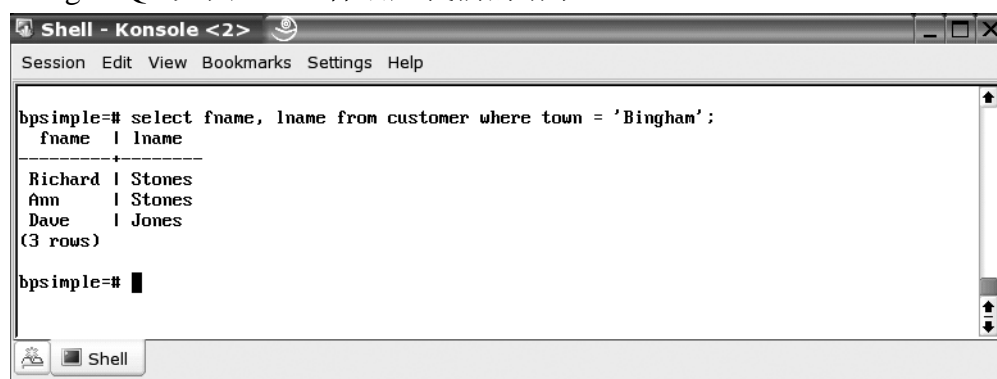


图 2-8 选择数据列的子集

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：



```
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

bpsimple=# select fname, lname from customer where town = 'Bingham';
  fname | lname
-----+-----
 Richard | Stones
    Ann  | Stones
    Dave | Jones
(3 rows)

bpsimple=#
```

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

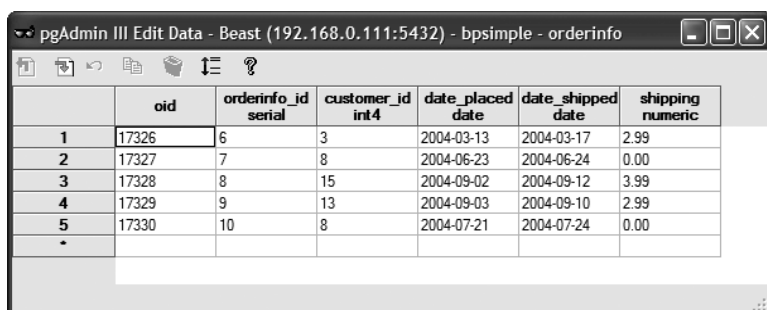
回到我们关于客户订单的问题，也就是在为每个客户存储扩展订单信息时突然让我们的简单客户表格变得非常凌乱的问题。在我们开始不知道我们的客户会有多少订单的时候要怎么存储客户的订单？你也许会从本章的标题就能猜测到，在关系数据库中解决这种问题的办法是增加一个表格存储这些信息。

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。



The screenshot shows the pgAdmin III interface with the 'orderinfo' table selected. The table has the following columns: oid, orderinfo_id serial, customer_id int4, date_placed date, date_shipped date, and shipping numeric. The data rows are as follows:

	oid	orderinfo_id serial	customer_id int4	date_placed date	date_shipped date	shipping numeric
1	17326	6	3	2004-03-13	2004-03-17	2.99
2	17327	7	8	2004-06-23	2004-06-24	0.00
3	17328	8	15	2004-09-02	2004-09-12	3.99
4	17329	9	13	2004-09-03	2004-09-10	2.99
5	17330	10	8	2004-07-21	2004-07-24	0.00

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 `PostgreSQL` 内部使用的一个特殊列。当前版本的 `PostgreSQL` 默认认为表建立这个列，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在我们客户的细节，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：

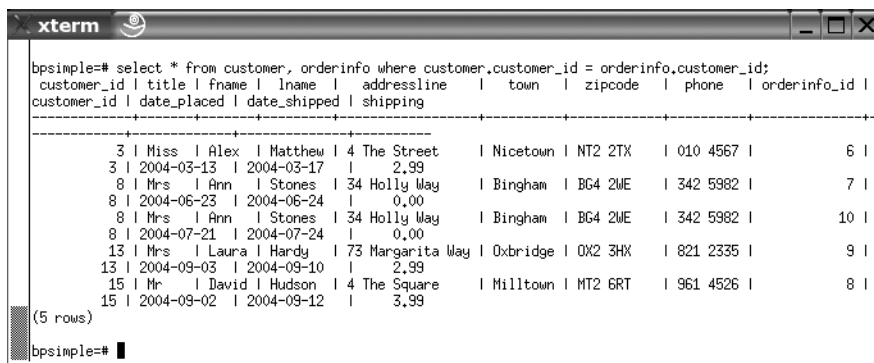
- 我们需要的列
- 我们需要检索数据的表
- 这两个表之间怎么关联

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。



```
bpsimple=# select * from customer, orderinfo where customer.customer_id = orderinfo.customer_id;
customer_id | title | fname | lname | addressline | town | zipcode | phone | orderinfo_id |
customer_id | date_placed | date_shipped | shipping
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 | Miss | Alex | Matthew | 4 The Street | Nicetown | NT2 2TX | 010 4567 | 6 |
3 | 2004-03-13 | 2004-03-17 | 2.99
8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982 | 7 |
8 | 2004-06-23 | 2004-06-24 | 0.00
8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982 | 10 |
8 | 2004-07-21 | 2004-07-24 | 0.00
13 | Mrs | Laura | Hardy | 73 Margarita Way | Oxbridge | OX2 3HX | 821 2335 | 9 |
13 | 2004-09-03 | 2004-09-10 | 2.99
15 | Mr | David | Hudson | 4 The Square | Milltown | MT2 6RT | 961 4526 | 8 |
15 | 2004-09-02 | 2004-09-12 | 3.99
(5 rows)
bpsimple=#
```

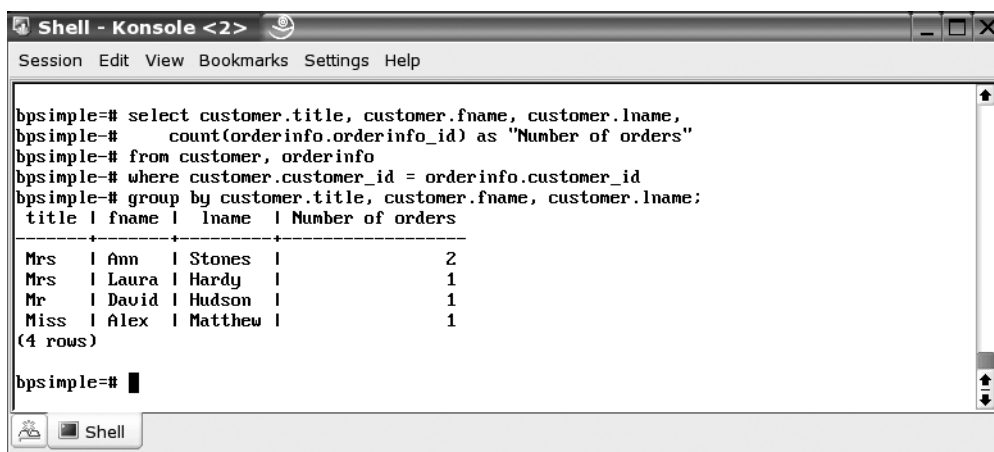
图 2-11 通过一次操作从两个表中选择数据

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。



```
bpsimple=# select customer.title, customer.fname, customer.lname,
bpsimple=#         count(orderinfo.orderinfo_id) as "Number of orders"
bpsimple=# from customer, orderinfo
bpsimple=# where customer.customer_id = orderinfo.customer_id
bpsimple=# group by customer.title, customer.fname, customer.lname;
 title | fname | lname | Number of orders
-----+-----+-----+-----
Mrs    | Ann   | Stones | 2
Mrs    | Laura | Hardy  | 1
Mr     | David | Hudson | 1
Miss   | Alex  | Matthew | 1
(4 rows)
bpsimple=#
```

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 Access 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_idnt`、`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。

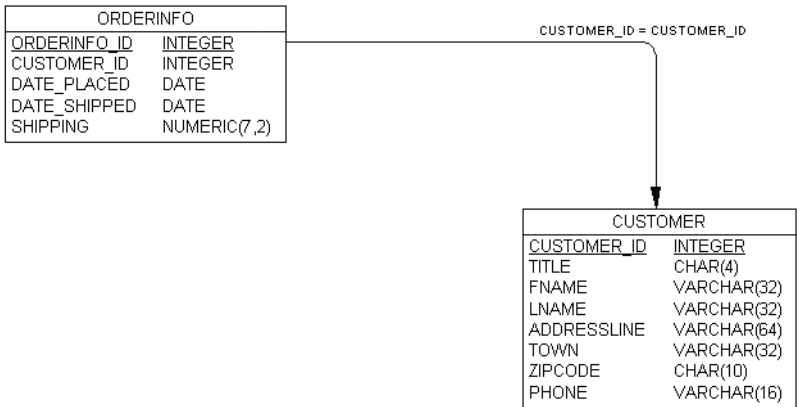


图 2-14 单的实体关系图

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。

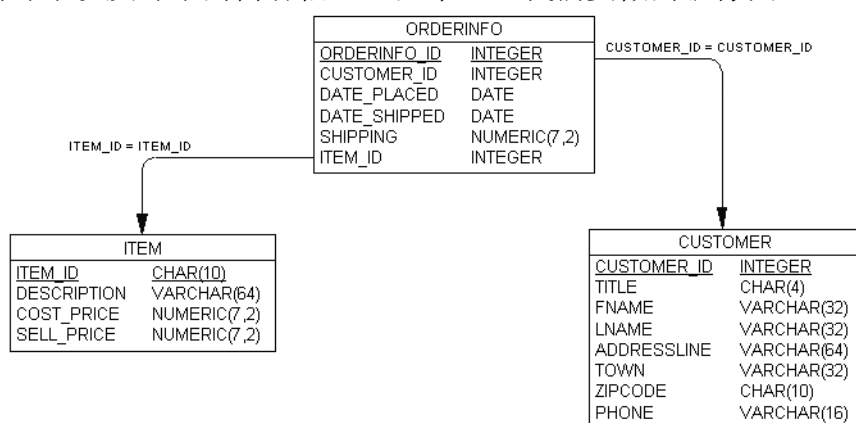


图 2-15 在客户和订单项目上的一个尝试

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。

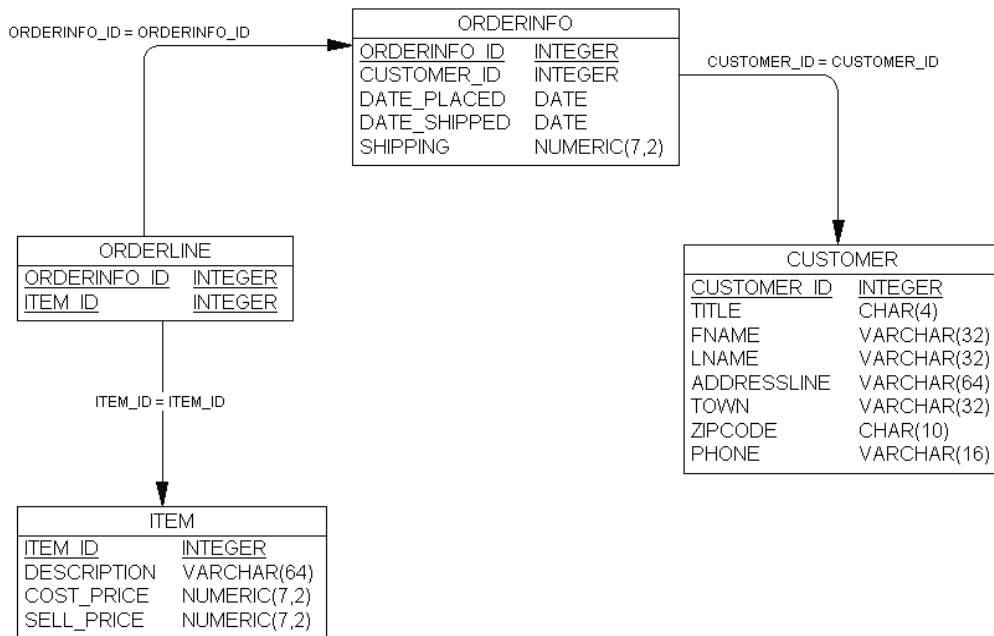


图 2-16 关联客户和订单

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。

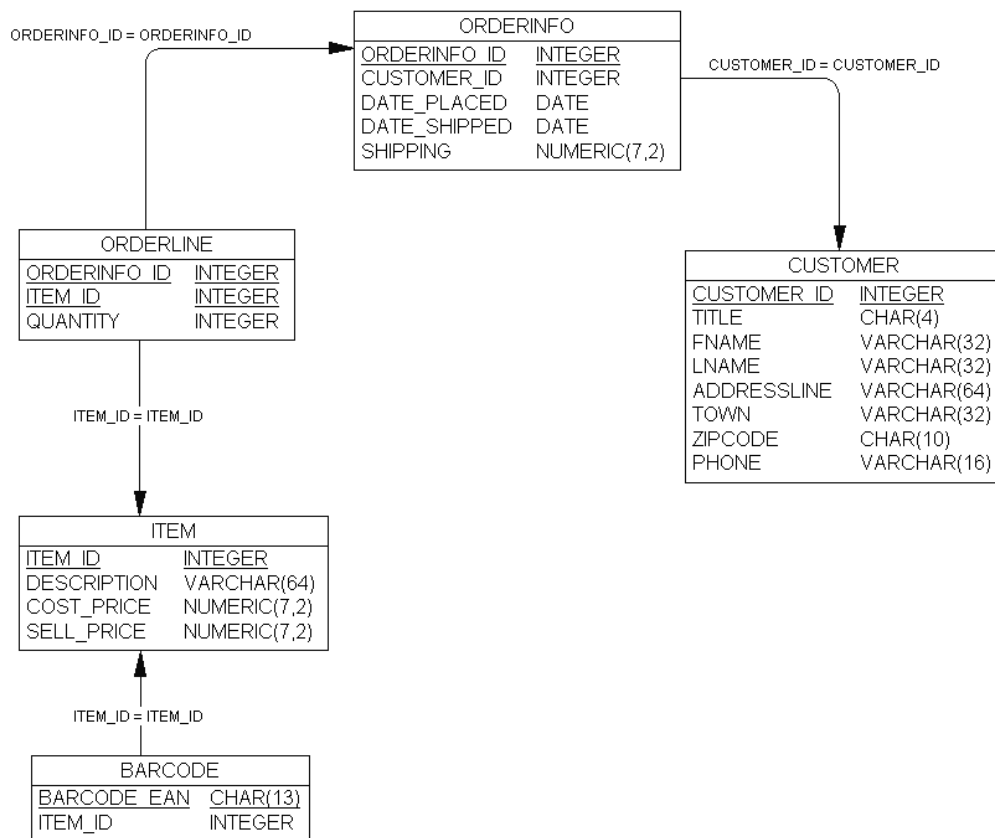


图 2-17 增加条码这个关系

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。

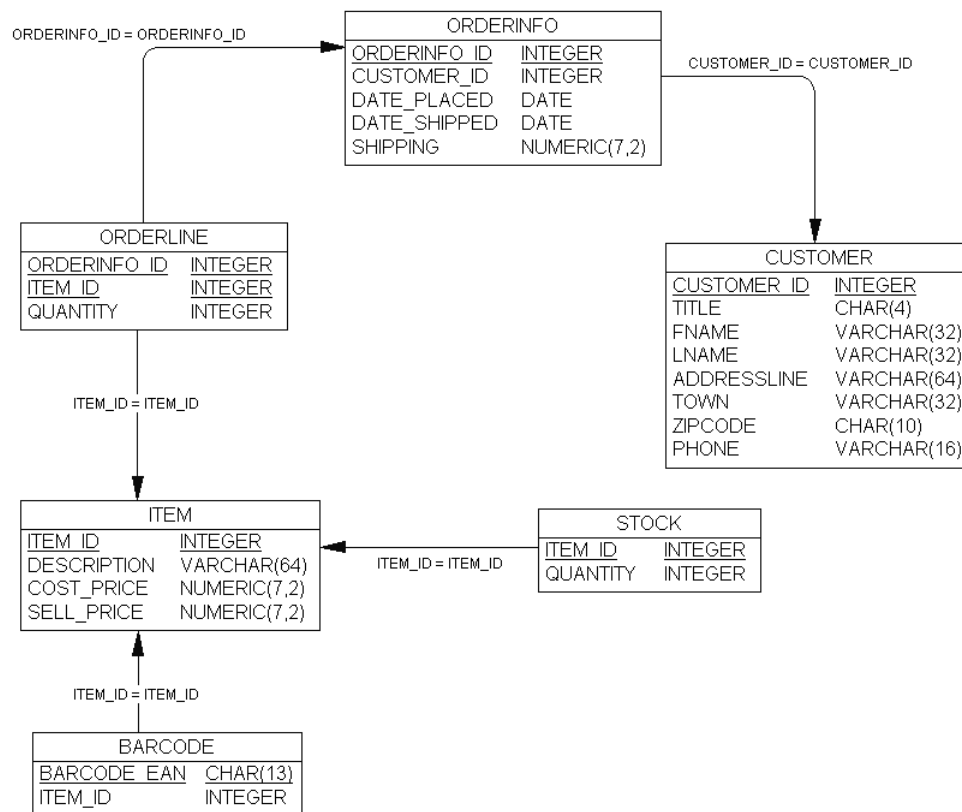


图 2-19 bpsimple 数据库的设计

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 varchar r	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。
numeri c	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显示糟糕的设计，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烦恼，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。

第三章初步使用 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	Python 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：

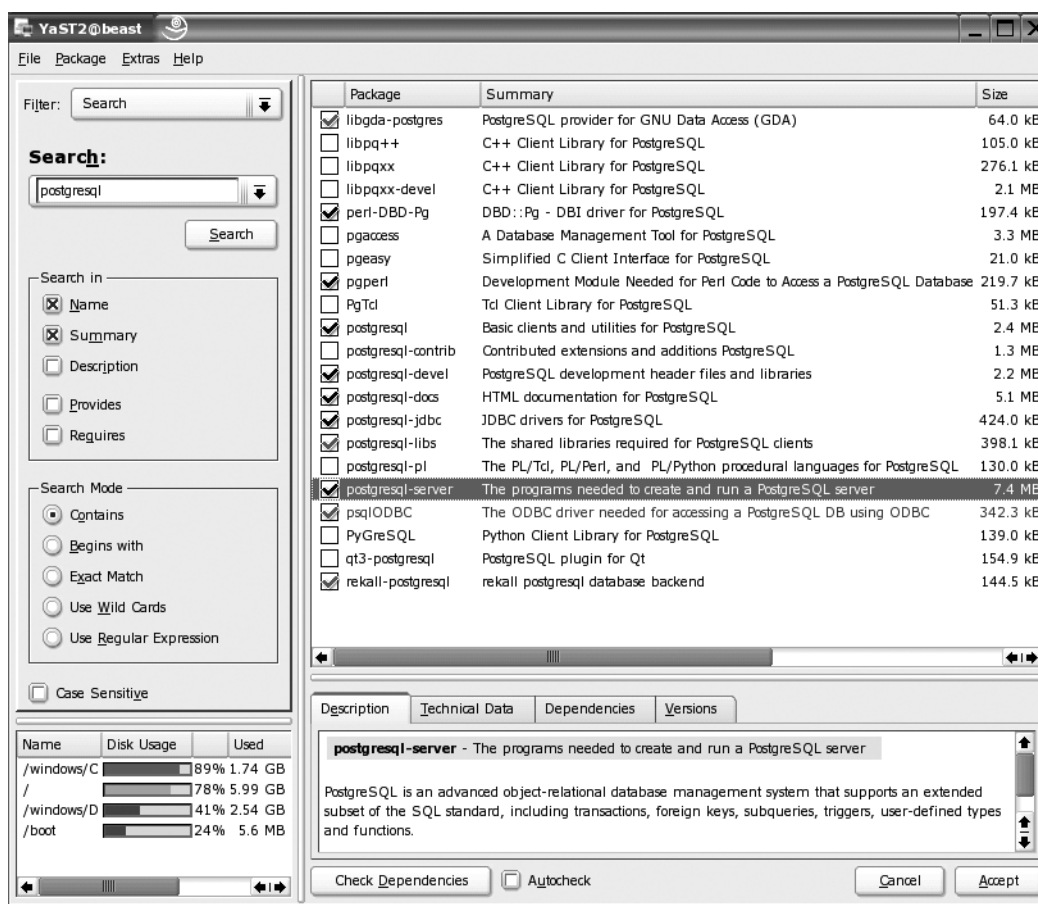


图 3-1，通过 Yast2 从 SuSE 的安装包安装 PostgreSQL

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 官方网站的发行笔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 <code>pg_ctl</code> 和 <code>postmaster</code>
data	数据库本书，通过 <code>initdb</code> 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。数据文件时系统的核心，存储你的数据库的所有信息。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。

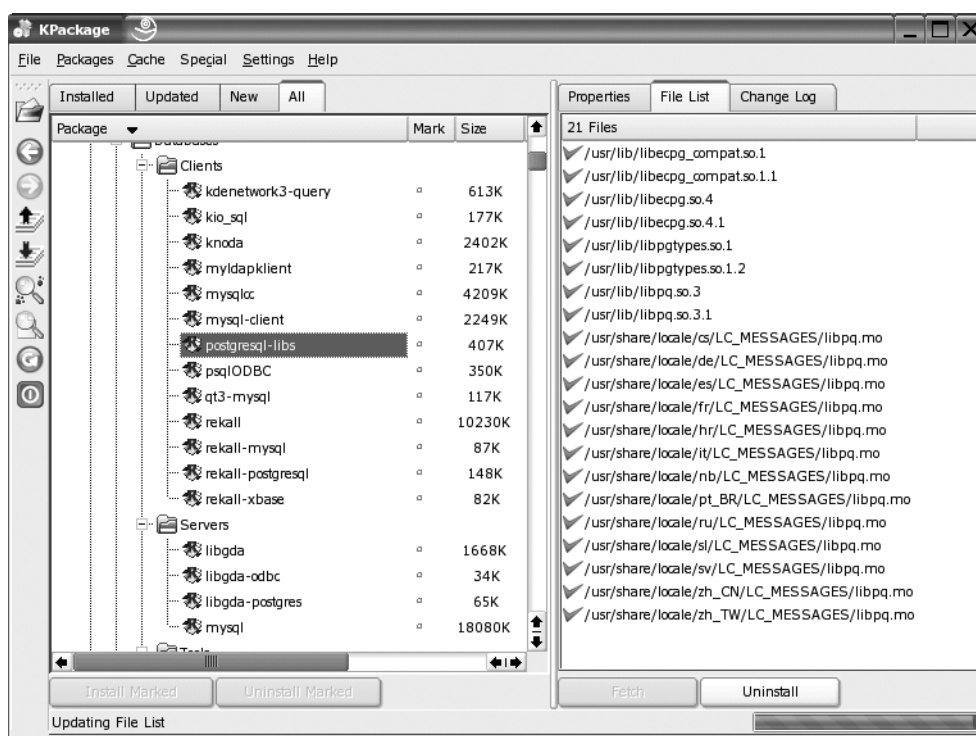


图 3-2 使用 KPackage 查看包内容的示例

使用 Linux 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 xzf 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 <code>prefix</code> 指向的目录；默认为 <code>/usr/local/pgsql</code>
--bindir=dir	安装应用程序到 <code>dir</code> ；默认为 <code>prefix/bin</code>
--with-docdir=dir	安装文档到 <code>dir</code> ；默认为 <code>prefix/doc</code>
--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 `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 系统中可能可能需要建立 `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，然后切换到 `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：

```
hostbpsimple    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：

```
hostbpsimple    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：

```
hostall 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：

```
hostall 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本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_ctl 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 或 .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。

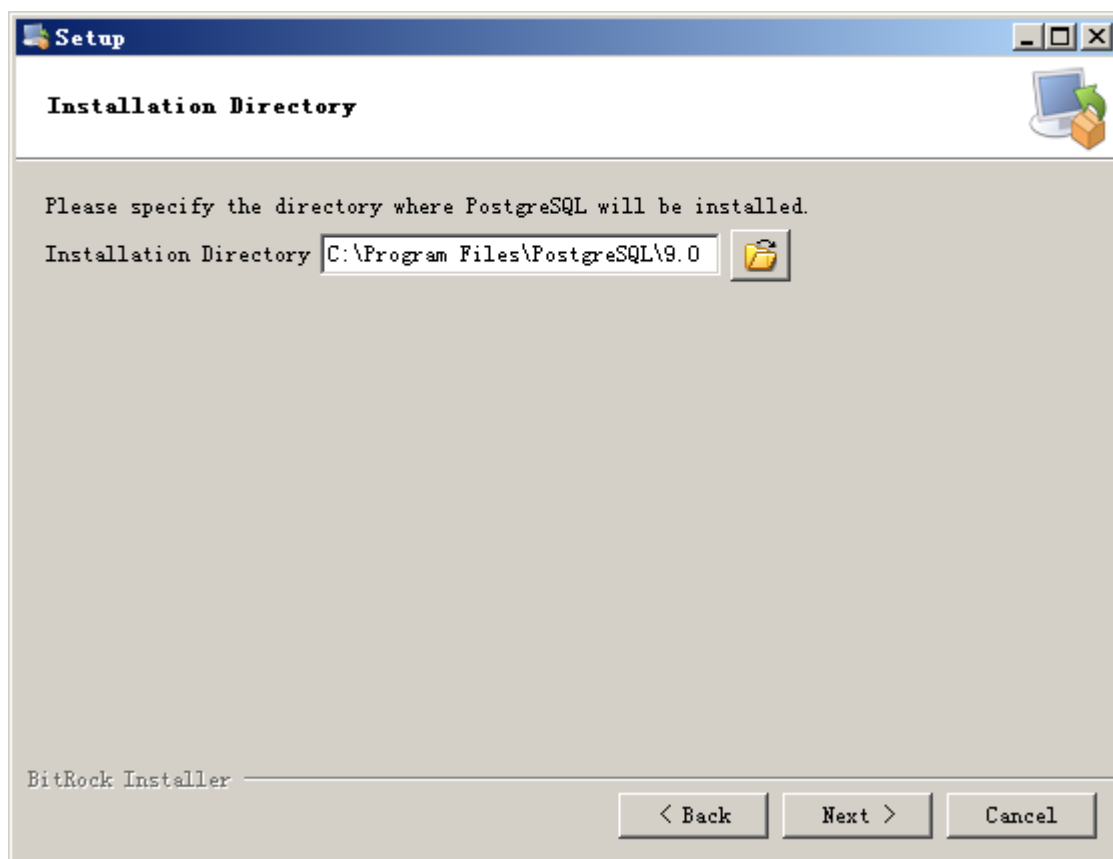


图 3-3 PostgreSQL 安装位置

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：

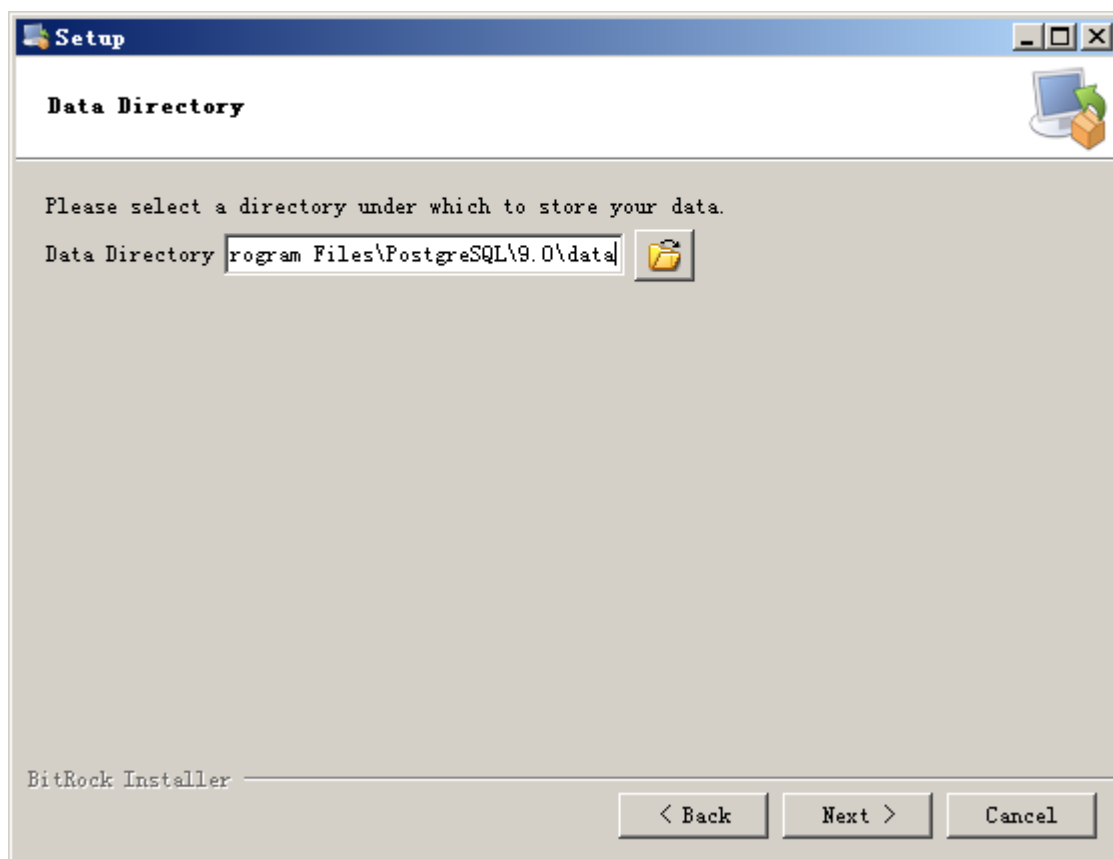


图 3-4 PostgreSQL 数据目录

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：

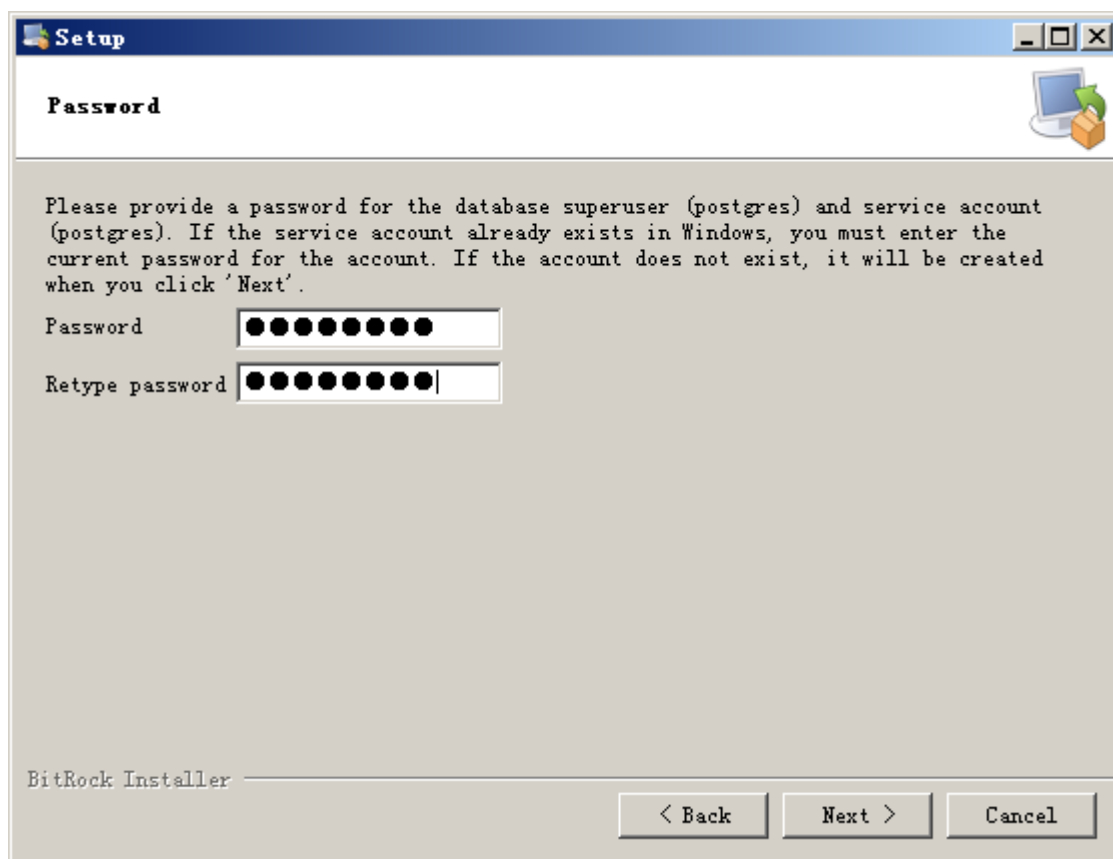


图 3-5 输入 postgres 用户密码

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 的安全漏洞并被使用溢出攻击，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：

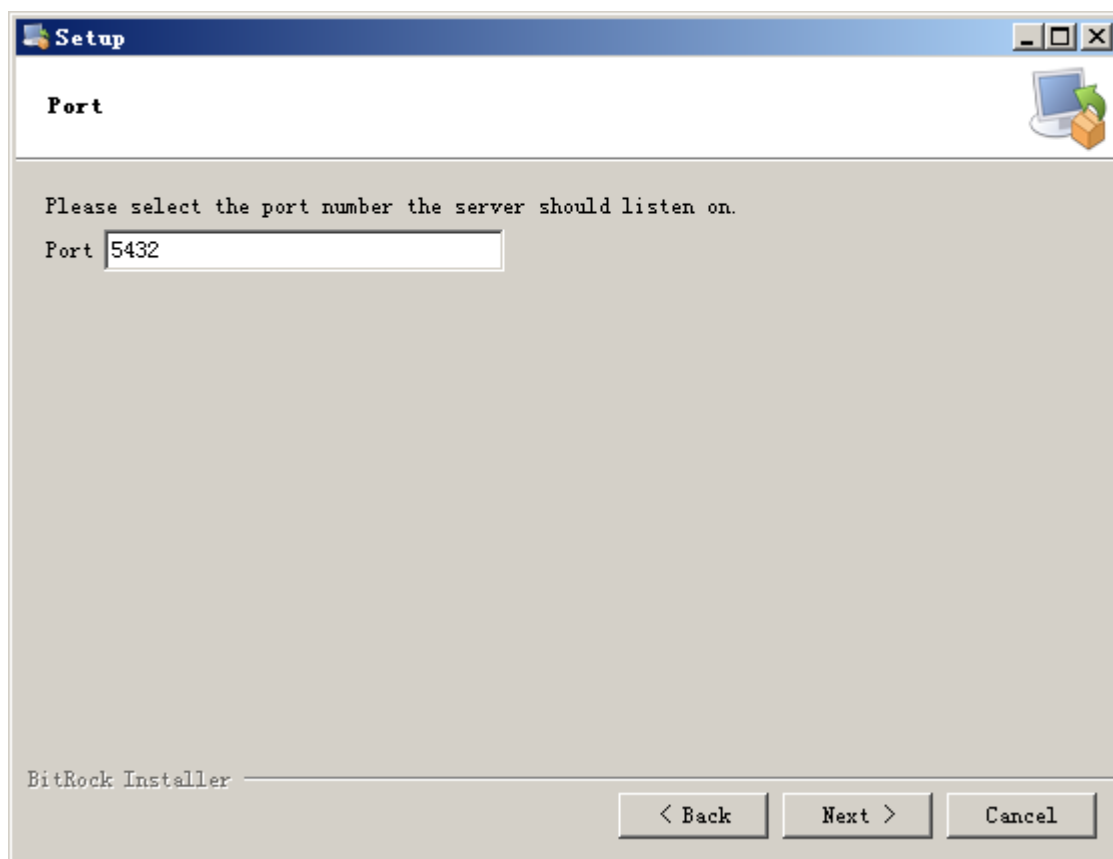


图 3-6 输入服务监听端口

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：

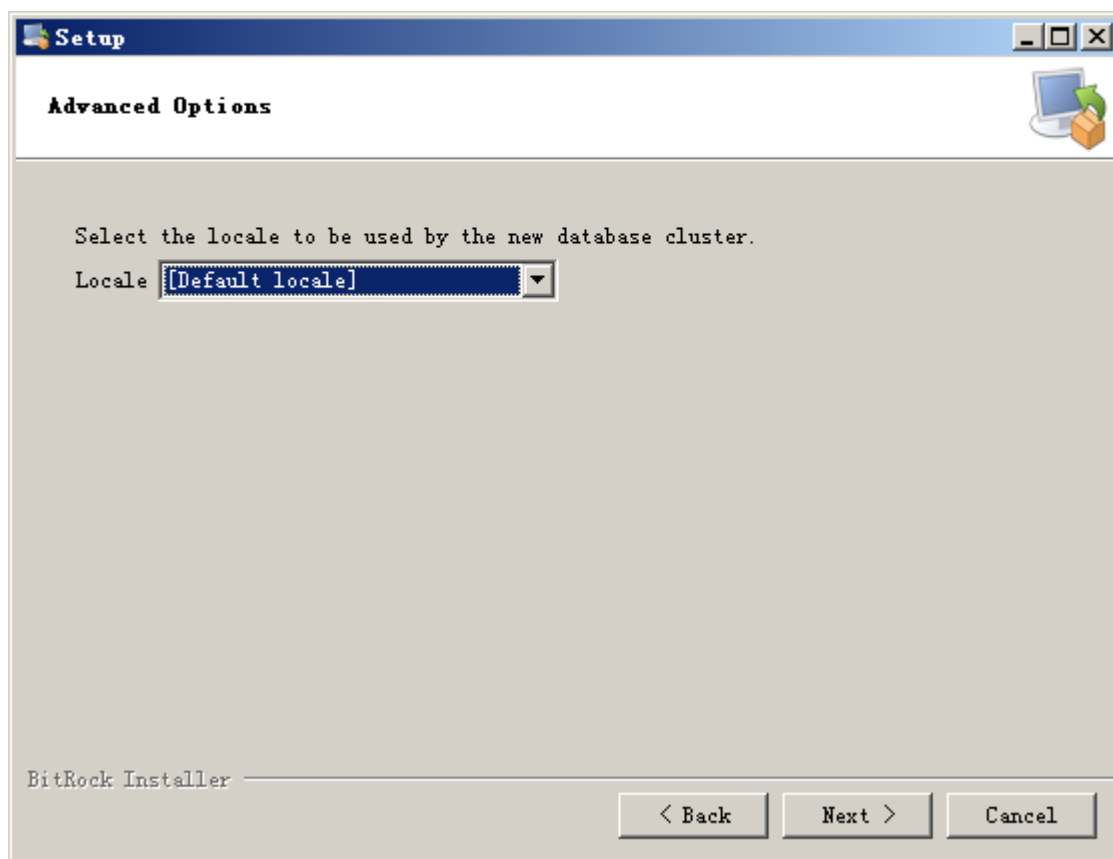


图 3-7 选择新建数据库的区域

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：

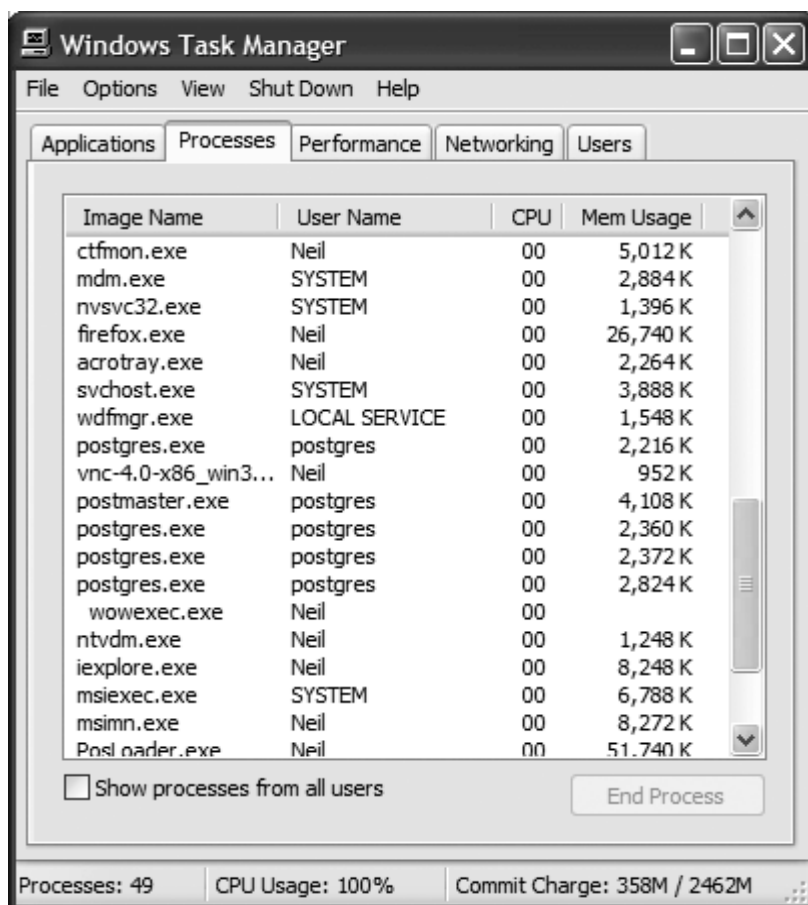


图 3-8 PostgreSQL 进程

PostgreSQL 程序和工具被安装到开始菜单，如图 3-8 所示：

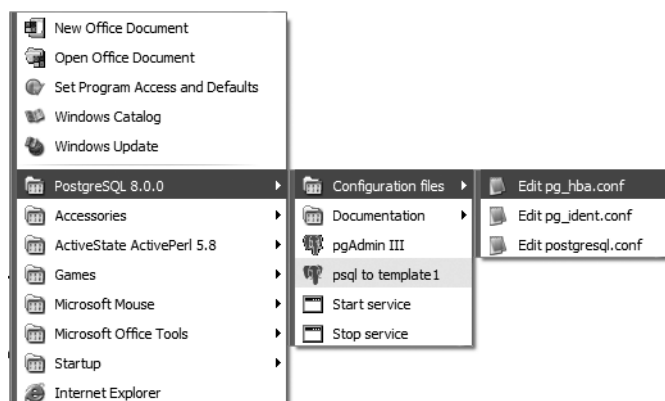


图 3-8 PostgreSQL 程序菜单

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：

```
hostall 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','Tibbsville','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。

第四章访问你的数据

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。

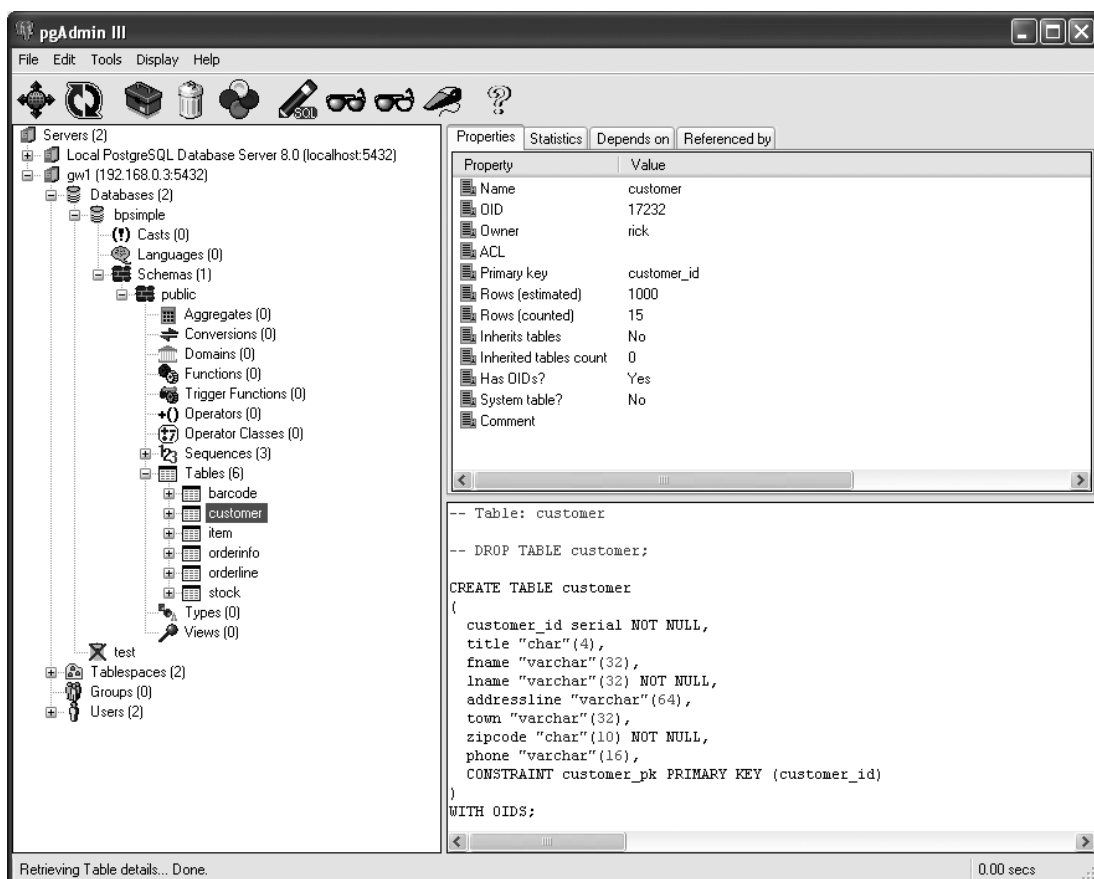


图 4-1 通过 pgAdmin III 检查 bpsimple 数据库

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	执行文件<filename>里头的命令
<filename>	

<code>\r</code>	重置缓冲器（忽略任何输入）
<code>\q</code>	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.pgadmin.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所有数据开始研究 SELECT 命令。我们通过很基础的形式 SELECT 语句，指出一个列的列表，以及一个带有表名的 FROM 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=>

它是如何实现的

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库，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
```

```
Tibbsville | Howard
```

```
Bingham | Jones
```

```
Winersby | Neill
```

```
Oxbridge | Hardy
```

```
Welltown | O'Neill
```

```
Milltown | Hudson
```

```
(15 rows)
```

```
bpsimple=>
```

它是如何实现

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。我们看到的數據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
Tibbsville	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。

它是如何实现

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
Tibbsville | Howard
Oxbridge | Hardy
Oakenham | Cozens
Niketown | Matthew
Niketown | Matthew
Milltown | Hudson
```

```
Lowtown | Stones
Histon  | Hickman
Hightown | Stones
Bingham | Jones
Bingham | Stones
Bingham | Stones
(15 rows)
```

bpsimple=>

它是如何实现

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  
Tibbsville  
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  
Tibbsville  
Welltown  
Winersby  
Yuleville  
(12 rows)
```

```
bpsimple=>
```

它是如何实现

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 **DISTINCT**，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操作符列在表 4-2 中。比较操作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 标准比较操作符

操作符	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：使用操作符

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=>
```

它是如何实现

虽然它乍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=>
```

它是如何实现的

这个模式的第一部分“_a”匹配一个任意字符开始第二个字符为小写 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，所以你看到的東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。

它是如何实现

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`=`操作符来比较日期，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          |          timestampz
-----+-----
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，以及如何控制数据的顺序了。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。

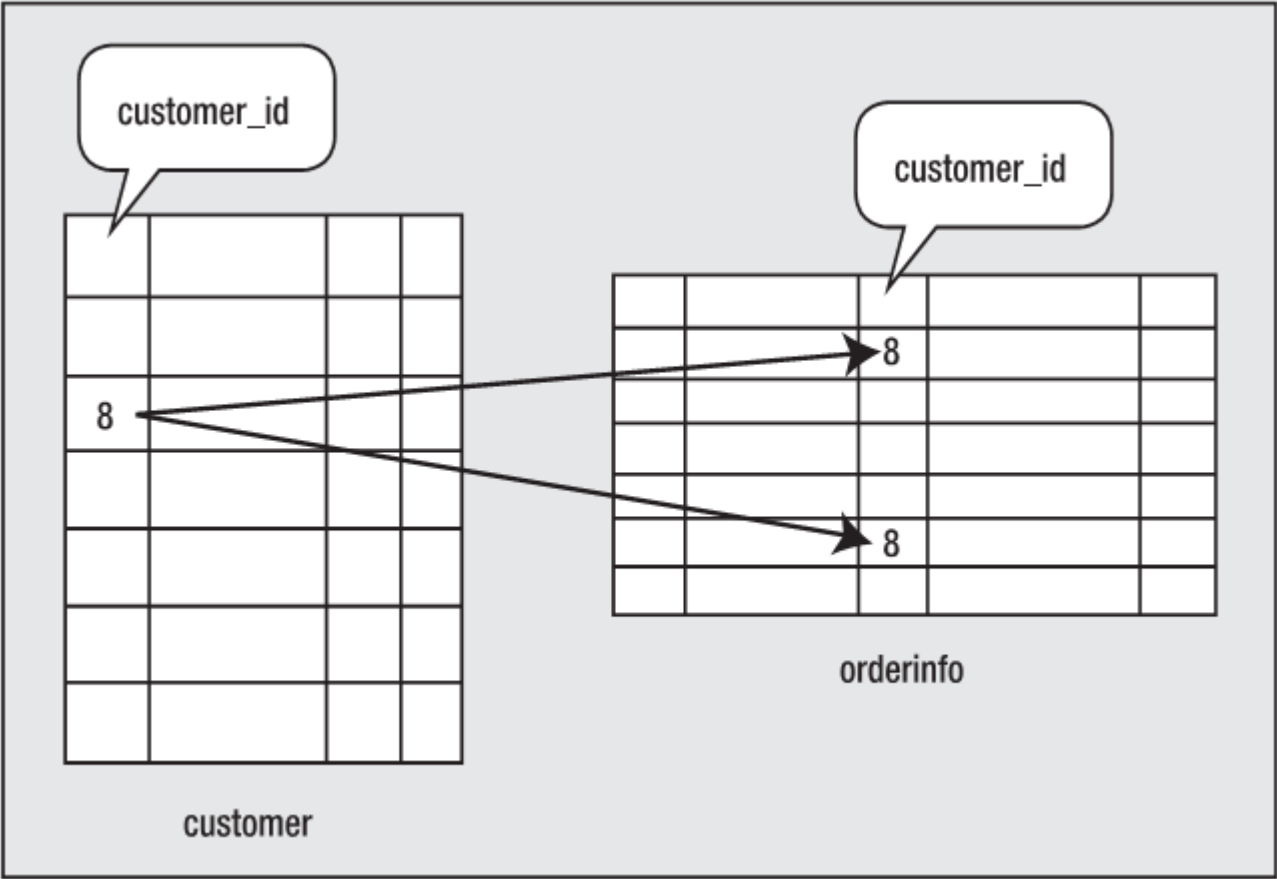


图 4-2 `customer` 表和 `orderinfo` 表的关系

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，通过在一个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=>
```

它是如何实现

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;
```

就像我们先前说过的，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。

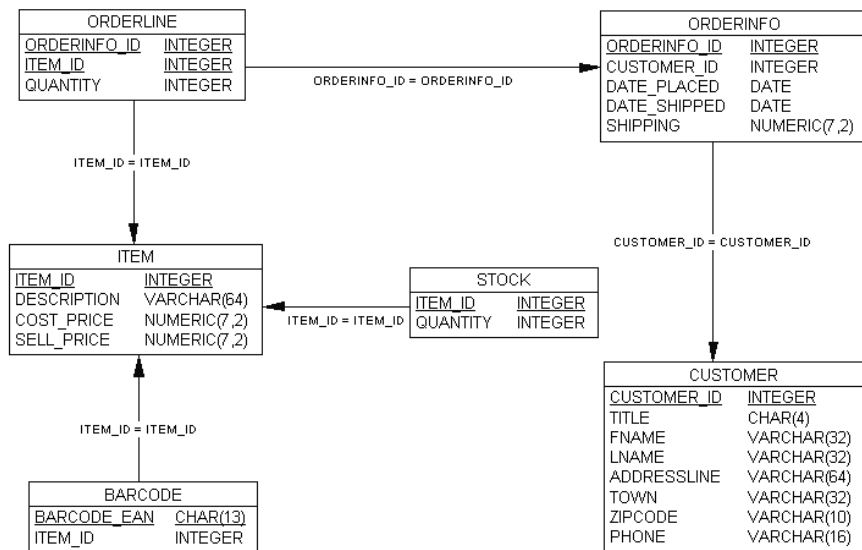


图 4-3 数据库模式

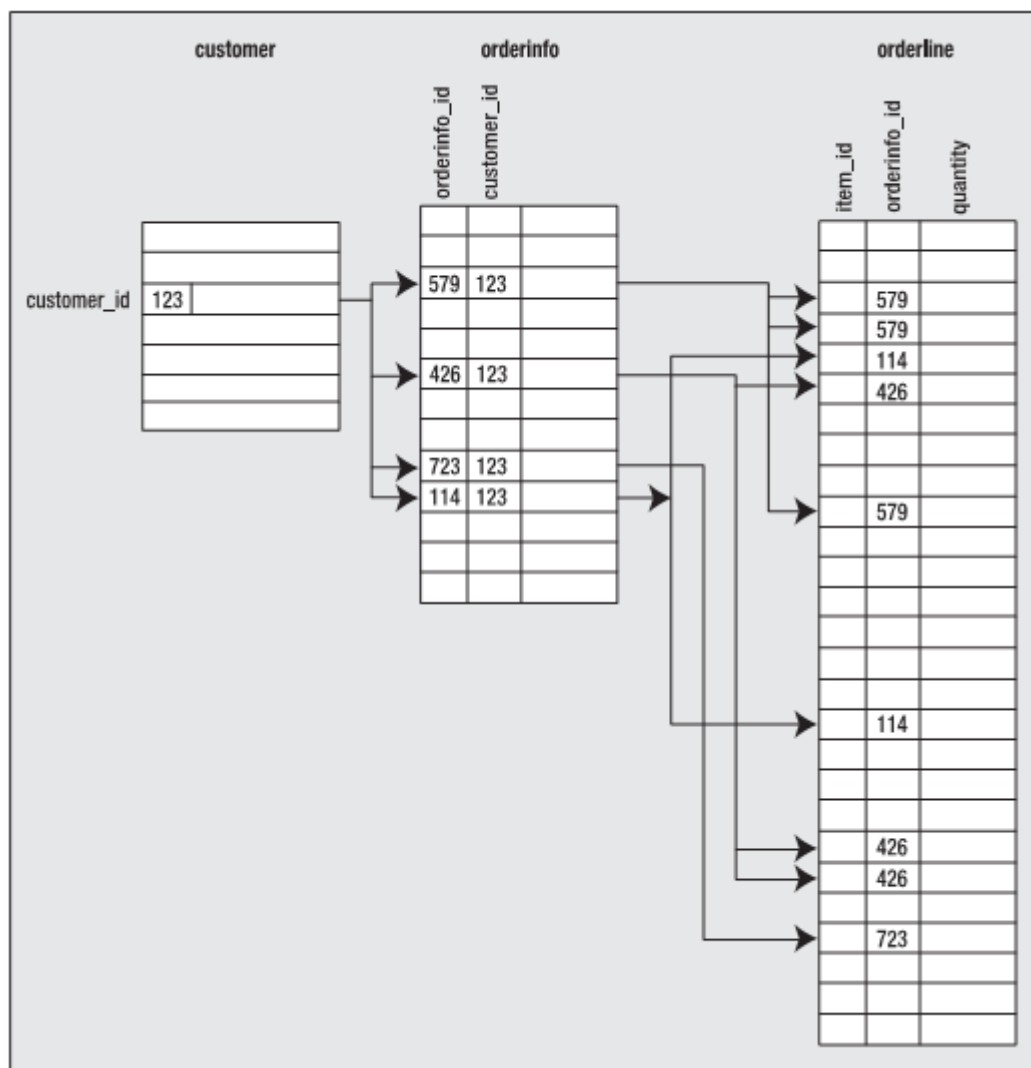


图 4-4 三个相关表

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=>

它是如何实现

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=>
```

它是如何实现：

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。

第五章 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

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

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：

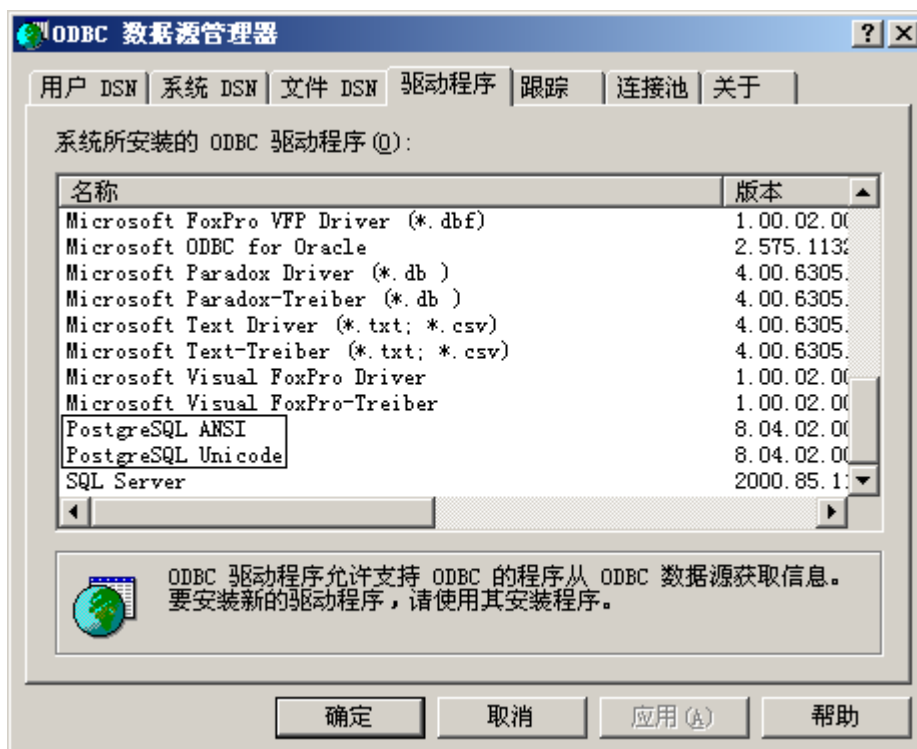


图 5-3 PostgreSQL 驱动程序已经安装

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

3. 选择 PostgreSQL 驱动并点击完成。
4. 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。

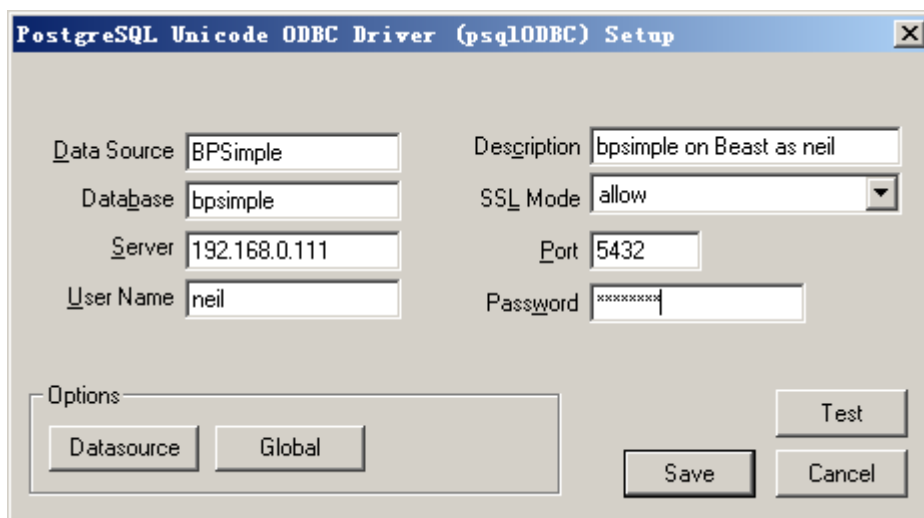


图 5-5 配置一个 PostgreSQL 数据源

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。

5. 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。

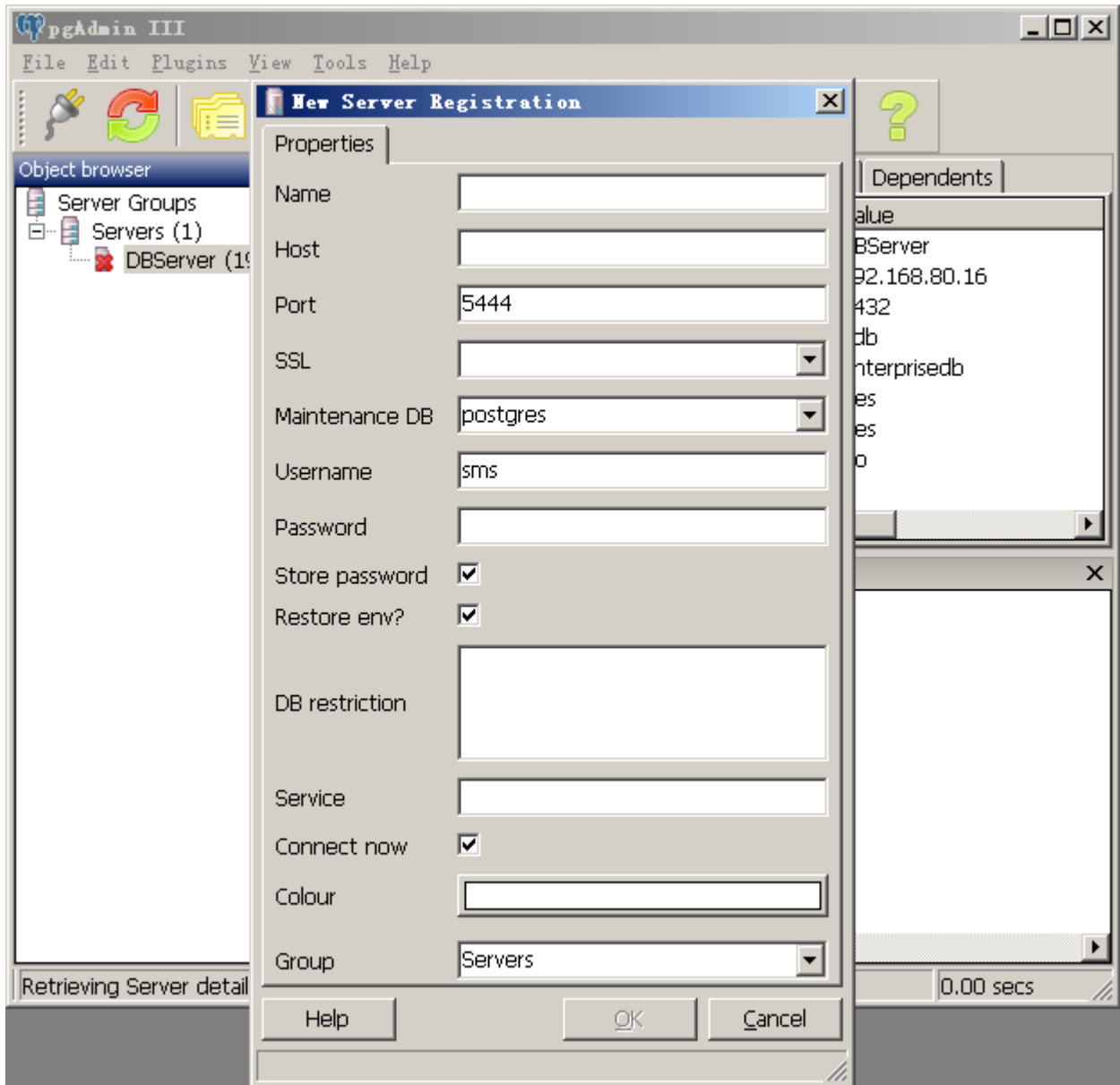


图 5-6 添加一个服务器连接到 pgAdmin III 中

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。

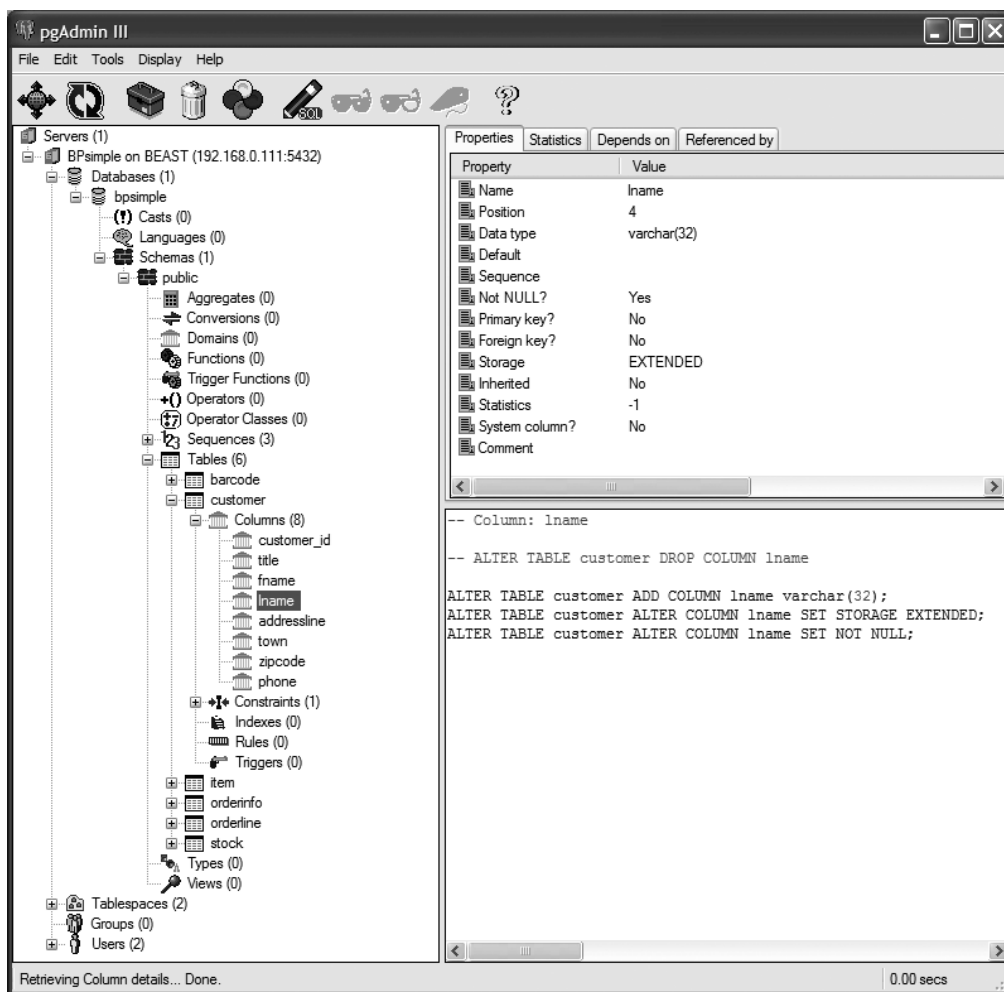


图 5-7 使用 pgAdmin III 检查表的属性

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

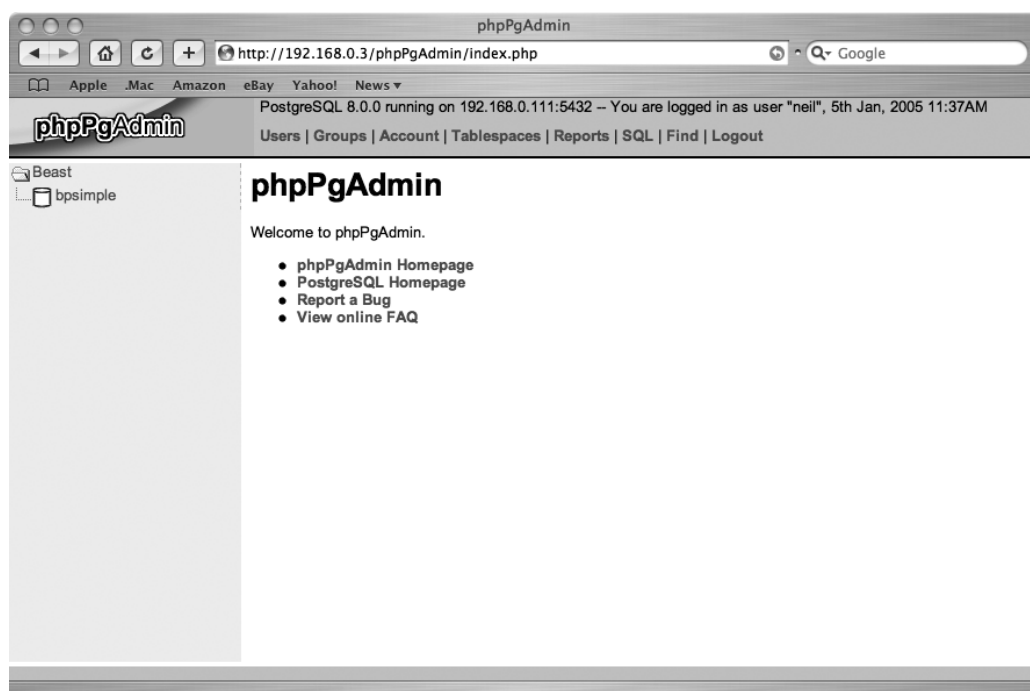


图 5-10 phpPgAdmin 的主界面

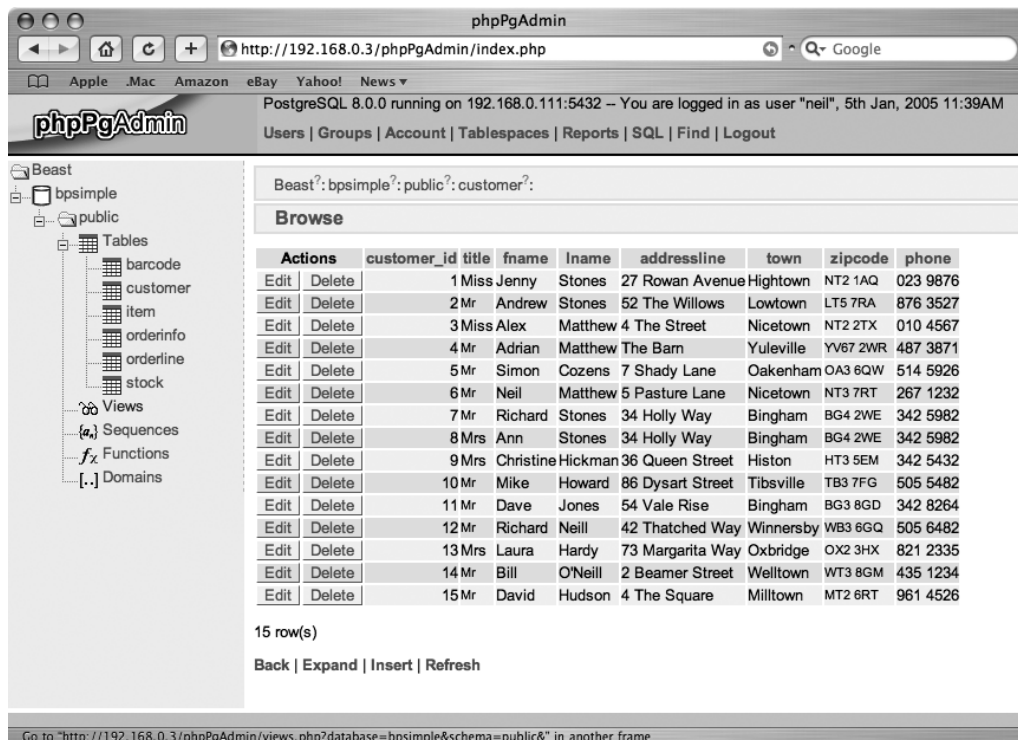


图 5-11 通过 phpPgAdmin 浏览表的数据

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。

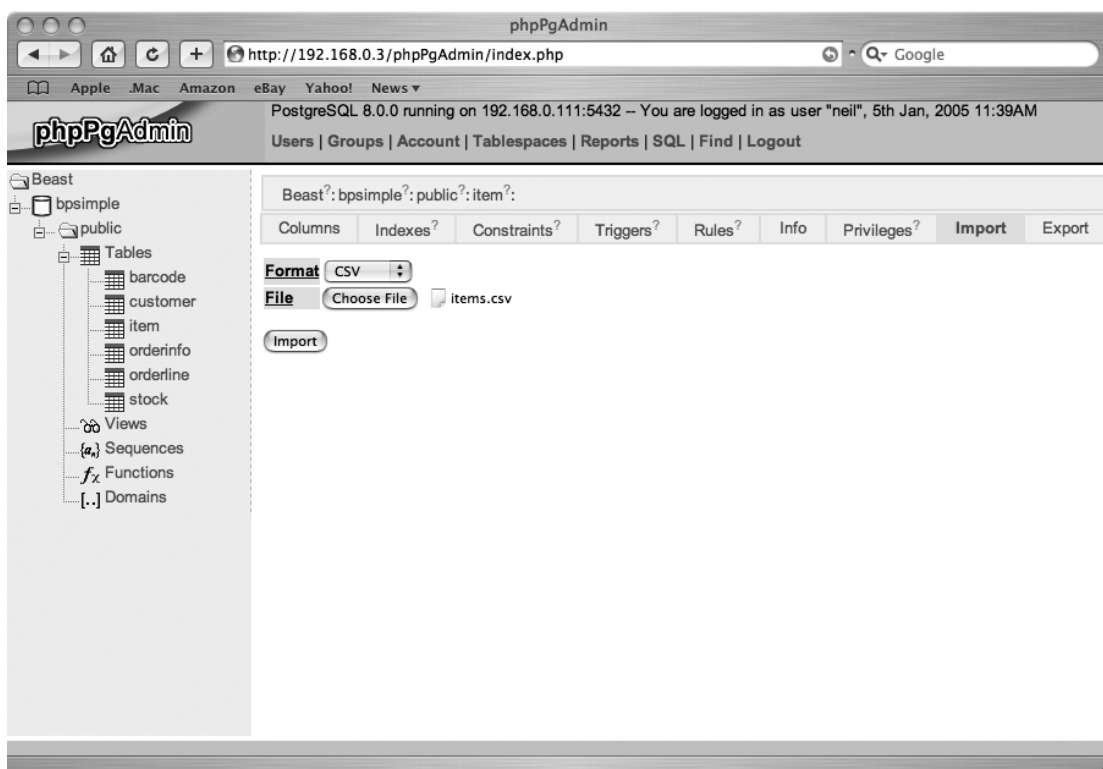


图 5-12 使用 phpPgAdmin 导入数据

Microsoft Access

虽然乍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。

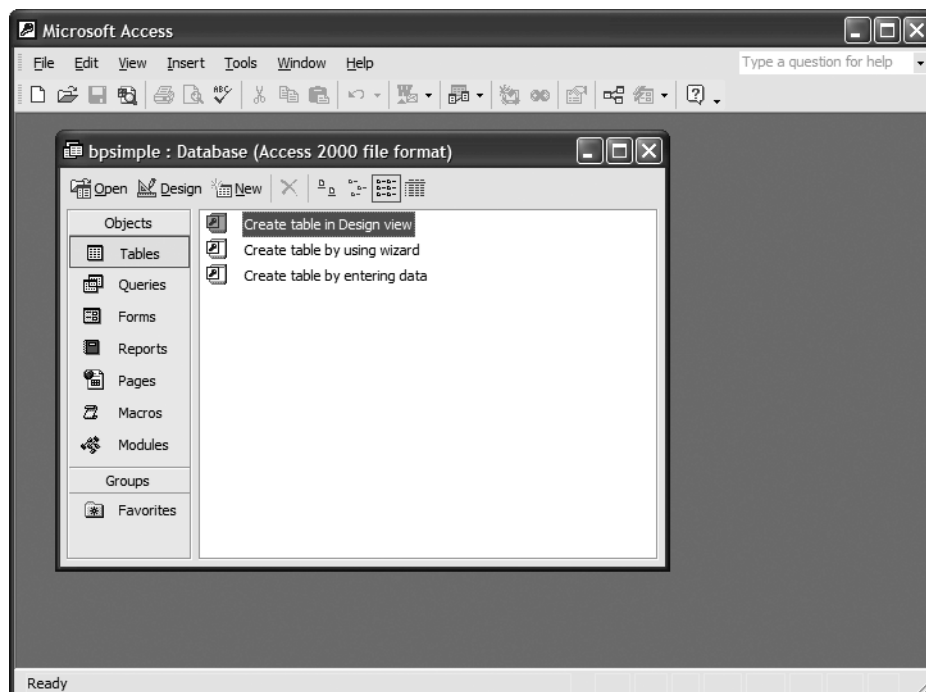


图 5-13 建立一个空白 Access 数据库

2. 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。

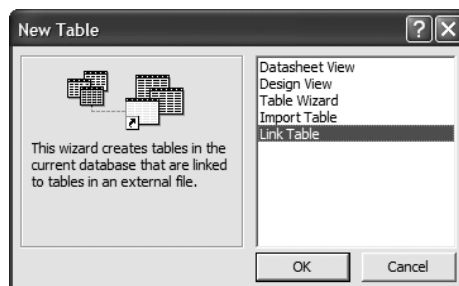


图 5-14 添加一个链接表

3. 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。

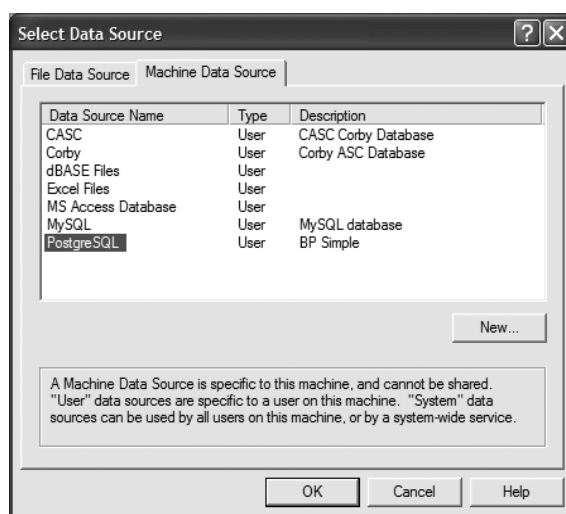


图 5-15 选择一个 ODBC 数据源

4. 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。

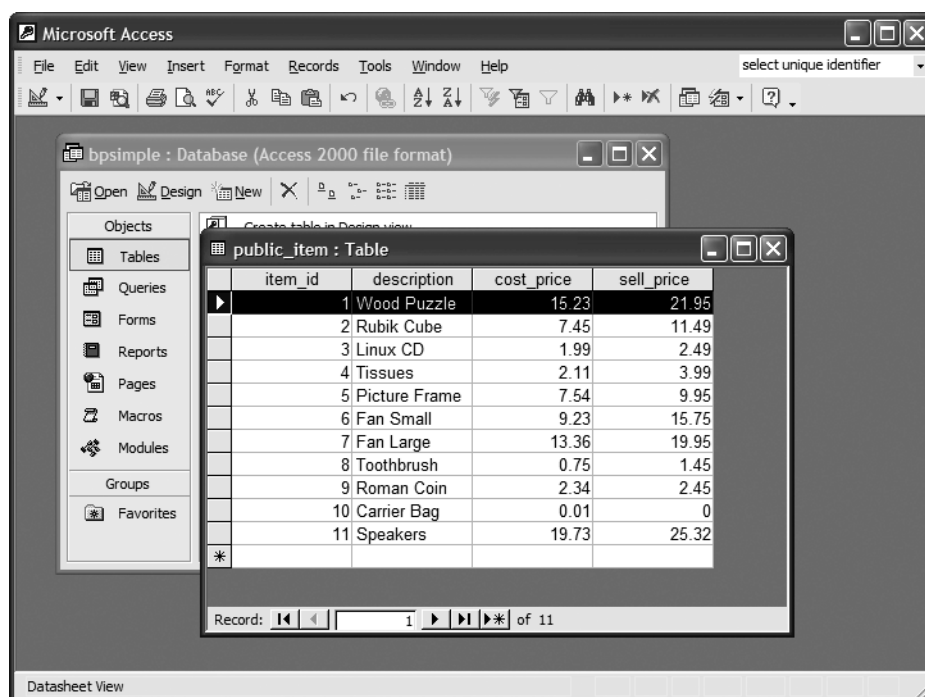


图 5-17 浏览一个链接表

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。

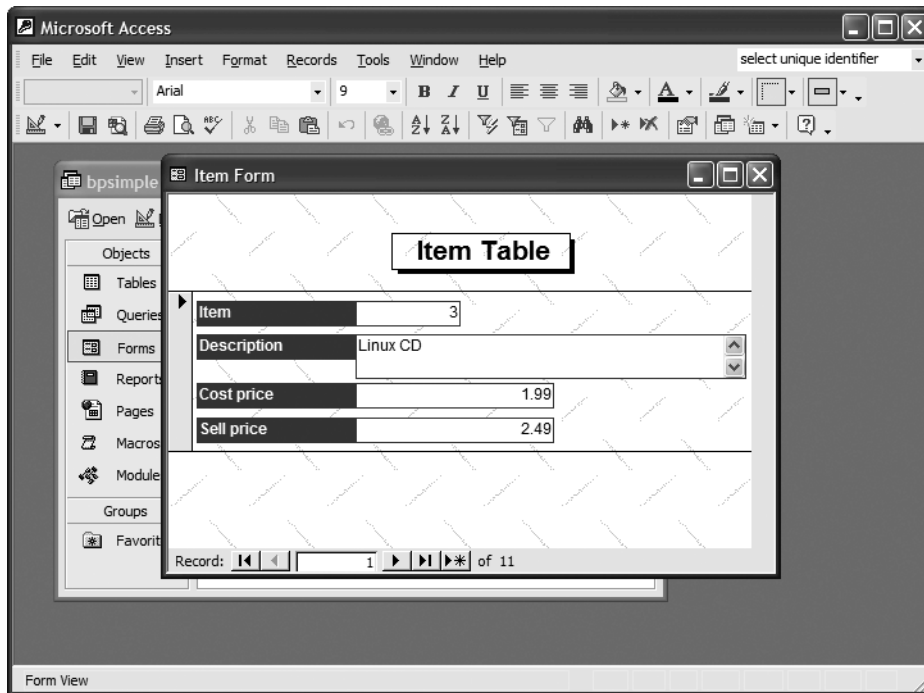


图 5-18 一个简单的 Access 数据登记表单

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。

The screenshot shows the Microsoft Access application window with a report titled 'Markup Report'. The report has a title bar 'Microsoft Access - [Markup Report]' and a 'select unique identifier' dropdown. The report content is a table with the following data:

Item	Cost price	Sell price	Markup
Carrier Bag	0.01	0	-0.01
Fan Large	13.36	19.95	6.59
Fan Small	9.23	15.75	6.52
Linux CD	1.99	2.49	0.5
Picture Frame	7.54	9.95	2.41
Roman Coin	2.34	2.45	0.11
Rubik Cube	7.45	11.49	4.04
Speakers	19.73	25.32	5.59
Tissues	2.11	3.99	1.88
Toothbrush	0.75	1.45	0.7
Wood Puzzle	15.23	21.95	6.72

The status bar at the bottom of the report window shows 'Page: 1' and 'Ready'.

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。

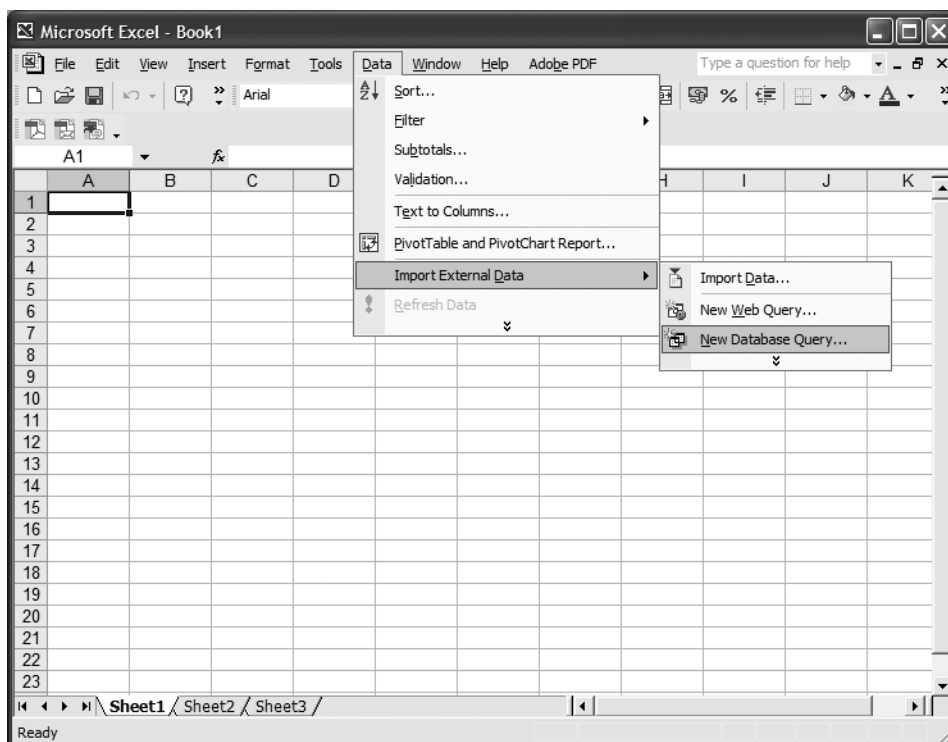


图 5-20 导入数据到 Excel 中

2. 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。
3. 当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。

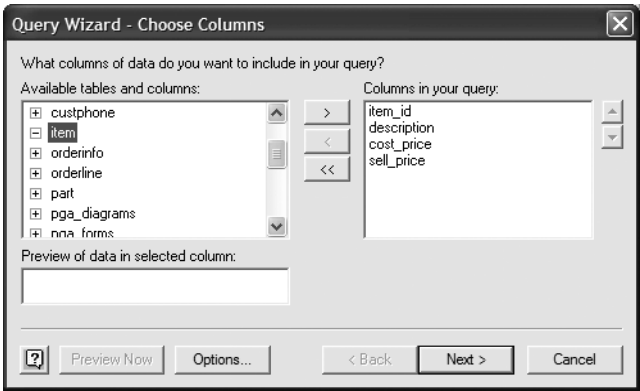


图 5-21 选择需要导入到 Excel 中的列

4. 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。在图 5-22 中，我们选择了那些售价高于 2 元的产品。

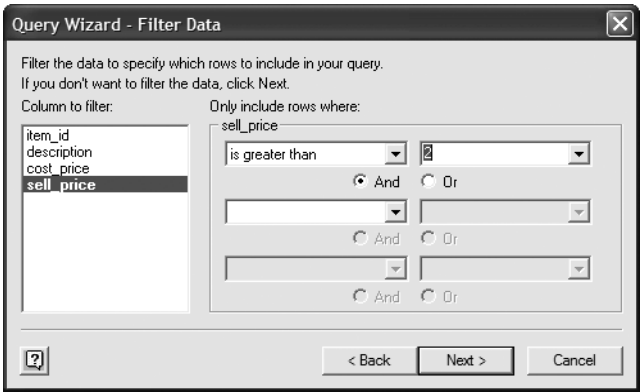


图 5-22 约束导入数据的行数

5. 最后，你可以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。

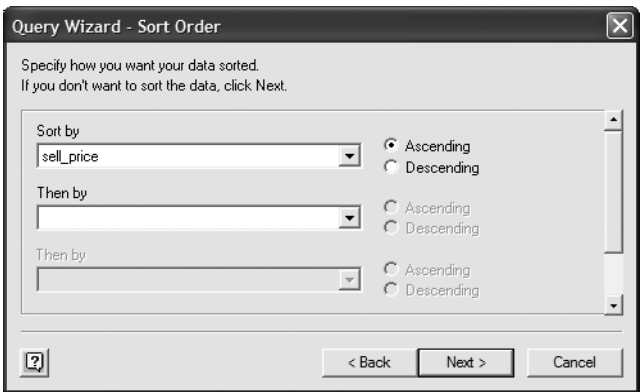


图 5-23 选择导入数据的排序标准

6. 在下一个对话框选择返回数据到 Excel 中。
7. 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。

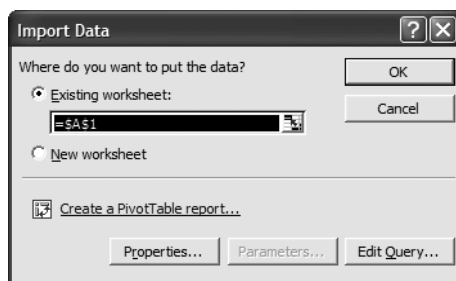
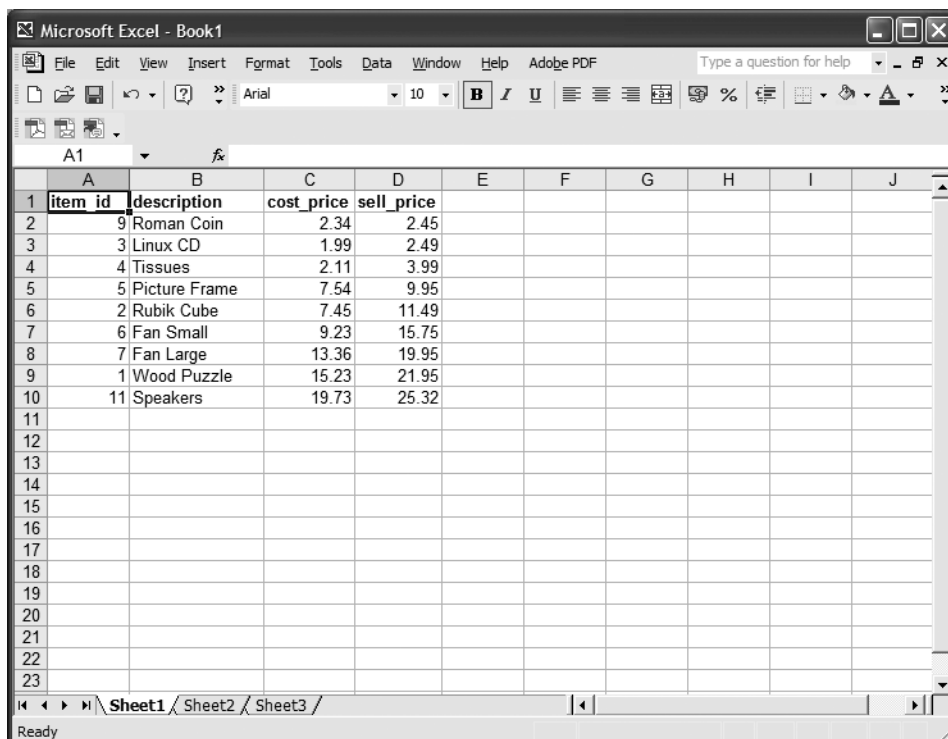


图 5-24 选择导入数据的位置

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。



	A	B	C	D	E	F	G	H	I	J
1	item_id	description	cost_price	sell_price						
2	9	Roman Coin	2.34	2.45						
3	3	Linux CD	1.99	2.49						
4	4	Tissues	2.11	3.99						
5	5	Picture Frame	7.54	9.95						
6	2	Rubik Cube	7.45	11.49						
7	6	Fan Small	9.23	15.75						
8	7	Fan Large	13.36	19.95						
9	1	Wood Puzzle	15.23	21.95						
10	11	Speakers	19.73	25.32						
11										
12										
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										

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。在图 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。

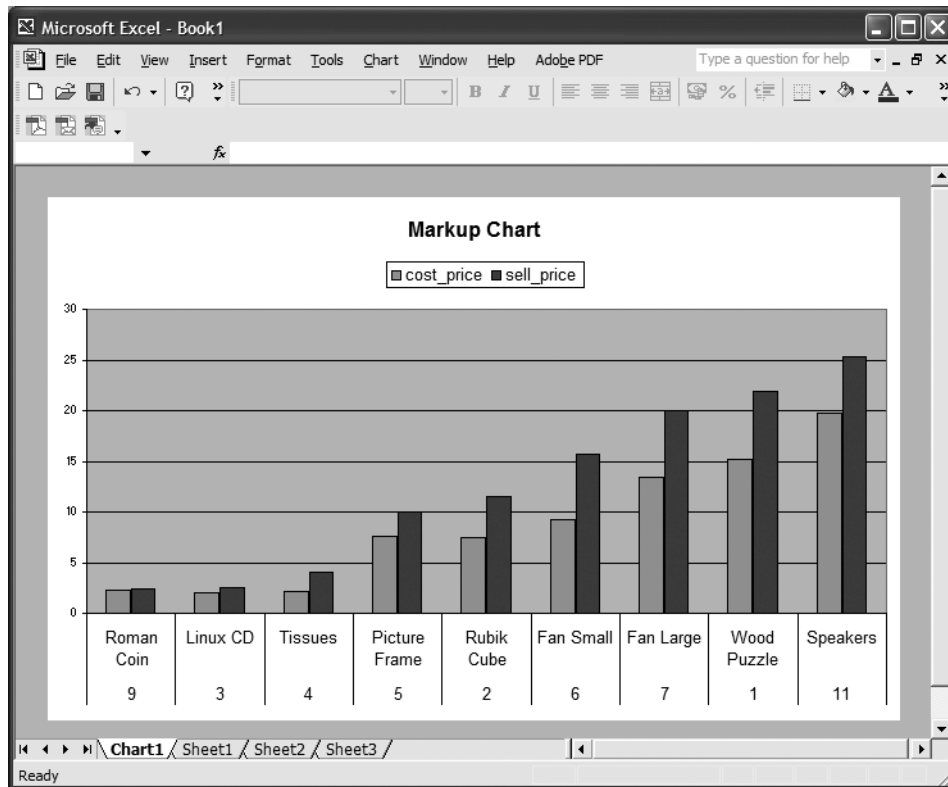


图 5-26 使用 PostgreSQL 数据的一个 Excel 图表

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。

第六章数据交互

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了解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。

它是如何实现

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是 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 `setval` 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

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，电子表格通常不提供合理的除了 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=#
```

它是如何实现

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
```

```
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。

它是如何实现

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。

它是如何实现的

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=#
```

它是如何实现

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这么简单，所以一定要非常小心

它是如何实现的

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。

它是如何实现

TRUNCATE 简单地将指定表的所有行删除。

如果你有一个表，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 FROM 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 UPDATE 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。

第七章高级数据选择

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
<code>count(*)</code>	提供行的计数
<code>count(列名)</code>	提供指定字段中值不是 NULL 的行的计数
<code>min(列名)</code>	返回指定列中的最小值
<code>max(列名)</code>	返回指定列中的最大值
<code>sum(列名)</code>	返回指定列的值的合计总数
<code>avg(列名)</code>	返回指定列的值的平均数

聚集函数非常有用且通常很容易使用。在本章，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 [GROUP 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使用的时候才有用。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 GROUP 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。

它是如何实现

`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。

它是如何实现

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 | Tibbsville  
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 | Tibbsville
```

```
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(*)` 来为名字进行计数，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 | Tibbsville
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。

它是如何实现

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=#
```

它是如何实现的

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。我们不受限于只能使用一个，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!

它是如何实现

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。

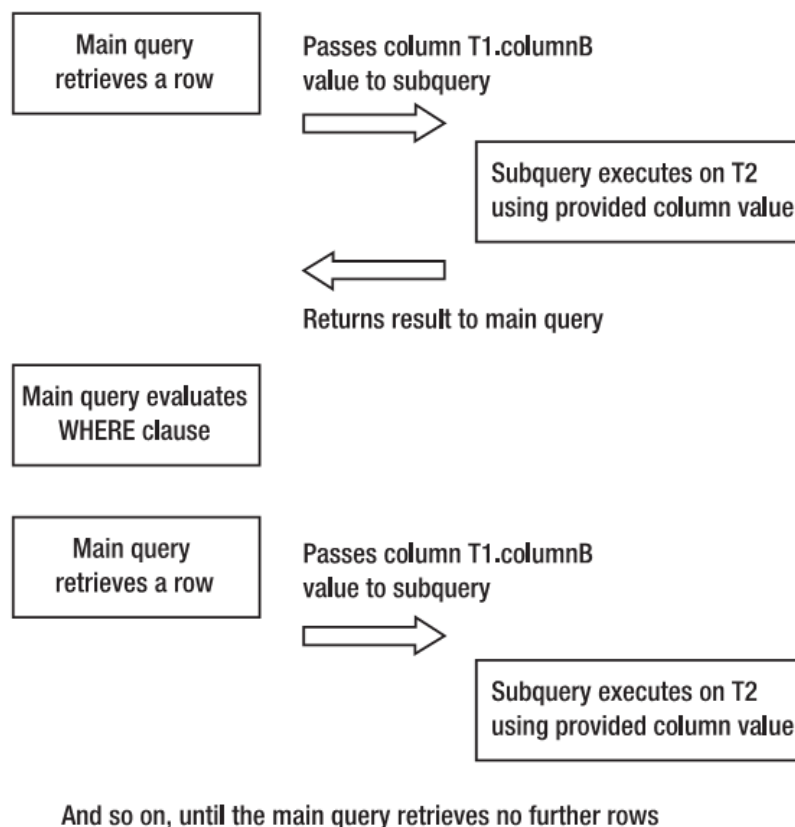


图 7-1 相关子查询的执行

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=#

它是如何实现

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并没有太多。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=#
```

它是如何实现

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：

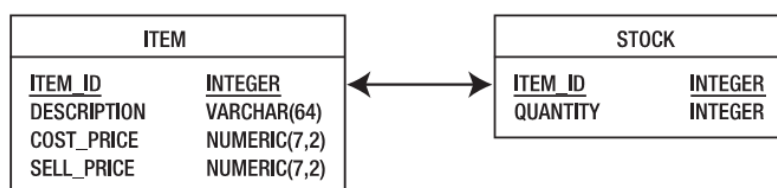


图 7-2 item 和 stock 表的模式

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，否则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—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	

```
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 OUTER 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=#
```

它是如何工作的

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。

第八章数据定义

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 `true`。

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=>

```

它是如何工作的

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 <code>n</code> 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 <code>n</code> 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 <code>varchar</code> ，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 <code>varchar</code> 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=>

```

它是如何工作的

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
Smallinteger	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 2147483647 的数字
Serial		和 <code>integer</code> 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 <code>n</code> ，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 <code>p</code> 个数字的实数，其中小数点后有 <code>s</code> 位。不像 <code>float</code> ，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。 <code>Money</code> 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 <code>number</code> 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)

它是如何工作的

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，如图 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 <code>timestamp</code> 之间差别的信息
timestampz	<code>PostgreSQL</code> 扩展的类型，存储包含时区信息的 <code>timestamp</code>

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。要往数组列中插入值，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=>
```

它是如何工作的

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=>
```

它是如何工作的

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=>
```

它是如何工作的

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
<code>length(column-name)</code>	返回一个字符串的长度
<code>trim(column-name)</code>	移除字符串开始和结尾的空格
<code>strpos(column-name, string)</code>	返回子串在列中的位置
<code>substr(column-name, position,</code>	根据指定位置和长度截取子串。第一个字符算作位置 1

length)

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 |      timestampz
```

```
-----+-----+-----
```

```
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` 列被添加到你的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。

一个需要特别注意的是当一个 **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=>
```

它是如何工作的

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 `orderid` 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 tconst (  
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 "tconst"  
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 tconst 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。

它是如何工作的

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，无论你怎么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 "ttconst"

```
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=>
```

它是如何工作的

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=>
```

它是如何工作的

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：

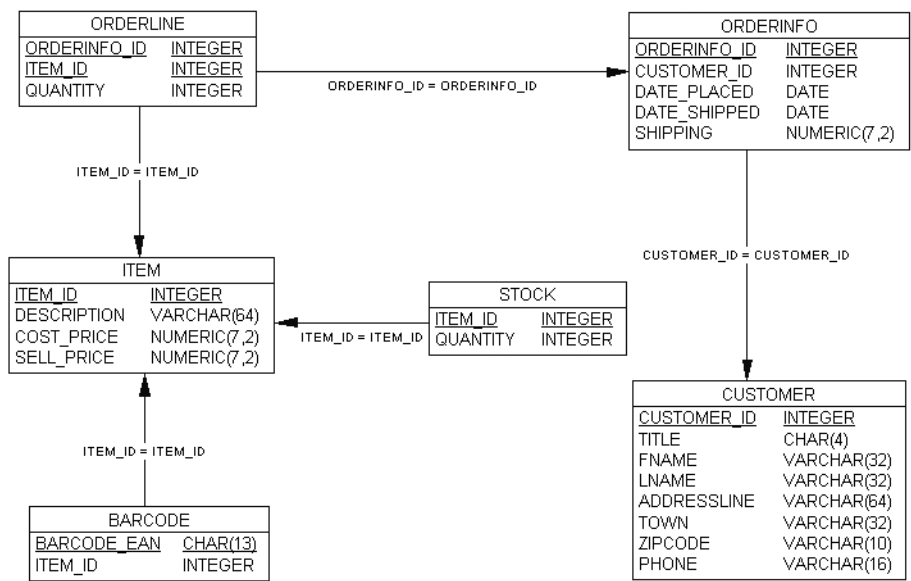


图 8-1 数据库模式设计

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，但实际上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完整性问题。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-definition` 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(customer_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=>
```

它是如何工作的

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 `item_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。

