PostgreSQL 從菜鳥到專家

翻譯：洞庭湖的泥鰍
如有問題，請聯繫：
bambo.huang(at)gmail.com

目錄

引言 ..... 錯誤! 尚未定義書籤。

第一章 POSTGRESQL 介绍 ..... 錯誤! 尚未定義書籤。

基于数据编程 ..... **錯誤! 尚未定義書籤。**
静态数据 ..... ***錯誤! 尚未定義書籤。***
用于数据存储的扁平文件 ..... ***錯誤! 尚未定義書籤。***
重复单元和其他问题 ..... ***錯誤! 尚未定義書籤。***
什么是数据库管理系统 ..... **錯誤! 尚未定義書籤。**
数据模型 ..... ***錯誤! 尚未定義書籤。***
查询语言 ..... ***錯誤! 尚未定義書籤。***
数据库管理系统的责任 ..... ***錯誤! 尚未定義書籤。***
什么是 POSTGRESQL? ..... **錯誤! 尚未定義書籤。**
PostgreSQL 历史简介 ..... ***錯誤! 尚未定義書籤。***
PostgreSQL 架构 ..... ***錯誤! 尚未定義書籤。***
通过 PostgreSQL 访问数据 ..... ***錯誤! 尚未定義書籤。***
什么是开源？ ..... **錯誤! 尚未定義書籤。**
相关资源 ..... **錯誤! 尚未定義書籤。**

第二章 关系数据库原理 ..... 錯誤! 尚未定義書籤。

电子表格的局限性 ..... **錯誤! 尚未定義書籤。**
将数据存入数据库 ..... **錯誤! 尚未定義書籤。**
选择列 ..... ***錯誤! 尚未定義書籤。***
为每个列选择数据类型 ..... ***錯誤! 尚未定義書籤。***
标记行的唯一性 ..... ***錯誤! 尚未定義書籤。***
在数据库中访问数据 ..... **錯誤! 尚未定義書籤。**
通过网络访问数据 ..... ***錯誤! 尚未定義書籤。***
处理多用户访问 ..... ***錯誤! 尚未定義書籤。***
数据分片和分块 ..... ***錯誤! 尚未定義書籤。***
增加信息 ..... ***錯誤! 尚未定義書籤。***
设计表 ..... ***錯誤! 尚未定義書籤。***
基本数据类型 ..... ***錯誤! 尚未定義書籤。***
处理未知的值：空值（NULL） ..... ***錯誤! 尚未定義書籤。***
回顾示例数据库 ..... ***錯誤! 尚未定義書籤。***
摘要 ..... **錯誤! 尚未定義書籤。**

第三章 初步使用 POSTGRESQL ..... 錯誤! 尚未定義書籤。

在 LINUX 和 UNIX 系统中安装 POSTGRESQL ..... **錯誤! 尚未定義書籤。**
在 Linux 中使用二进制文件安装 PostgreSQL ..... ***錯誤! 尚未定義書籤。***
通过源码安装 PostgreSQL ..... ***錯誤! 尚未定義書籤。***
在 Linux 和 Unix 上配置 PostgreSQL ..... ***錯誤! 尚未定義書籤。***
在 WINDOWS 中安装 POSTGRESQL ..... **錯誤! 尚未定義書籤。**
使用 Windows 安装程序 ..... ***錯誤! 尚未定義書籤。***
配置客户机访问 ..... ***錯誤! 尚未定義書籤。***
建立示例数据库 ..... **錯誤! 尚未定義書籤。**
添加用户记录 ..... ***錯誤! 尚未定義書籤。***
建立数据库 ..... ***錯誤! 尚未定義書籤。***
建表 ..... ***錯誤! 尚未定義書籤。***
移除表 ..... ***錯誤! 尚未定義書籤。***
填充表 ..... ***錯誤! 尚未定義書籤。***

摘要 ..... 錯誤! 尚未定義書籤。

第四章 访问你的数据 ..... 錯誤! 尚未定義書籤。

使用 PSQL ..... 錯誤! 尚未定義書籤。
在 Linux 系统中启动 ..... ***錯誤! 尚未定義書籤。***
在 Windows 系统中启动 ..... ***錯誤! 尚未定義書籤。***
解决启动问题 ..... ***錯誤! 尚未定義書籤。***
使用一些基本的 psql 命令 ..... ***錯誤! 尚未定義書籤。***
使用 SELECT 语句 ..... 錯誤! 尚未定義書籤。
覆盖列名 ..... ***錯誤! 尚未定義書籤。***
控制行的顺序 ..... ***錯誤! 尚未定義書籤。***
消除重复数据 ..... ***錯誤! 尚未定義書籤。***
执行计算 ..... 錯誤! 尚未定義書籤。
选择行 ..... 錯誤! 尚未定義書籤。
使用更复杂的条件 ..... ***錯誤! 尚未定義書籤。***
模式匹配 ..... ***錯誤! 尚未定義書籤。***
限制结果集 ..... ***錯誤! 尚未定義書籤。***
检查空值（NULL） ..... 錯誤! 尚未定義書籤。
检查时间和日期 ..... 錯誤! 尚未定義書籤。
设置时间和日期的风格 ..... ***錯誤! 尚未定義書籤。***
使用日期和时间函数 ..... ***錯誤! 尚未定義書籤。***
多个表协同工作 ..... 錯誤! 尚未定義書籤。
关联两个表 ..... ***錯誤! 尚未定義書籤。***
给表赋予别名 ..... ***錯誤! 尚未定義書籤。***
关联三个或更多的表 ..... ***錯誤! 尚未定義書籤。***
SQL92 的 SELECT 语法 ..... 錯誤! 尚未定義書籤。
摘要 ..... 錯誤! 尚未定義書籤。

第五章 POSTGRESQL 的命令行和图形界面工具 ..... 錯誤! 尚未定義書籤。

PSQL ..... 錯誤! 尚未定義書籤。
启动 psql ..... ***錯誤! 尚未定義書籤。***
在 psql 中输入命令 ..... ***錯誤! 尚未定義書籤。***
使用命令历史 ..... ***錯誤! 尚未定義書籤。***
在 psql 中执行脚本文件 ..... ***錯誤! 尚未定義書籤。***
检查数据库 ..... ***錯誤! 尚未定義書籤。***
psql 命令行快速参考 ..... ***錯誤! 尚未定義書籤。***
psql 内部命令快速参考 ..... ***錯誤! 尚未定義書籤。***
设置 ODBC ..... 錯誤! 尚未定義書籤。
在 Windows 中安装 ODBC 驱动程序 ..... ***錯誤! 尚未定義書籤。***
在 Windows 中建立一个数据源 ..... ***錯誤! 尚未定義書籤。***
在 Linux/Unix 中安装 ODBC 驱动程序 ..... ***錯誤! 尚未定義書籤。***
在 Linux/Unix 中建立一个数据源 ..... ***錯誤! 尚未定義書籤。***
PGADMIN III ..... 錯誤! 尚未定義書籤。
安装 pgAdmin III ..... ***錯誤! 尚未定義書籤。***
使用 pgAdmin III ..... ***錯誤! 尚未定義書籤。***
PHPPGADMIN ..... 錯誤! 尚未定義書籤。
安装 phpPgAdmin ..... ***錯誤! 尚未定義書籤。***
使用 phpPgAdmin ..... ***錯誤! 尚未定義書籤。***
MICROSOFT ACCESS ..... 錯誤! 尚未定義書籤。
使用链接表 ..... ***錯誤! 尚未定義書籤。***
输入数据及建立报表 ..... ***錯誤! 尚未定義書籤。***

MICROSOFT EXCEL ..... 錯誤! 尚未定義書籤。
POSTGRESQL 相关工具的资源 ..... 錯誤! 尚未定義書籤。
摘要 ..... 錯誤! 尚未定義書籤。

第六章 数据交互 ..... 錯誤! 尚未定義書籤。

添加数据到数据库中 ..... 錯誤! 尚未定義書籤。
使用基本的 INSERT 语句 ..... ***錯誤! 尚未定義書籤。***
使用更安全的插入语句 ..... ***錯誤! 尚未定義書籤。***
插入数据到 serial 类型的列中 ..... ***錯誤! 尚未定義書籤。***
插入空值 ..... ***錯誤! 尚未定義書籤。***
使用\copy 命令 ..... ***錯誤! 尚未定義書籤。***
直接从另一个程序加载数据 ..... ***錯誤! 尚未定義書籤。***
修改数据库中的数据 ..... 錯誤! 尚未定義書籤。
使用 UPDATE 语句 ..... ***錯誤! 尚未定義書籤。***
通过另一个表更新 ..... ***錯誤! 尚未定義書籤。***
从数据库删除数据 ..... 錯誤! 尚未定義書籤。
使用 DELETE 语句 ..... ***錯誤! 尚未定義書籤。***
使用 TRUNCATE 语句 ..... ***錯誤! 尚未定義書籤。***
摘要 ..... 錯誤! 尚未定義書籤。

第七章 高级数据选择 ..... 錯誤! 尚未定義書籤。

聚集函数 ..... 錯誤! 尚未定義書籤。
count 函数 ..... ***錯誤! 尚未定義書籤。***
Min 函数 ..... ***錯誤! 尚未定義書籤。***
Max 函数 ..... ***錯誤! 尚未定義書籤。***
Sum 函数 ..... ***錯誤! 尚未定義書籤。***
Avg 函数 ..... ***錯誤! 尚未定義書籤。***
子查询 ..... 錯誤! 尚未定義書籤。
返回多行记录的子查询 ..... ***錯誤! 尚未定義書籤。***
相关子查询 ..... ***錯誤! 尚未定義書籤。***
存在子查询（Existence Subqueries） ..... ***錯誤! 尚未定義書籤。***
UNION 连接 ..... 錯誤! 尚未定義書籤。
自连接 ..... 錯誤! 尚未定義書籤。
外连接 ..... 錯誤! 尚未定義書籤。
摘要 ..... 錯誤! 尚未定義書籤。

第八章 数据定义 ..... 錯誤! 尚未定義書籤。

数据类型 ..... 錯誤! 尚未定義書籤。
布尔数据类型 ..... ***錯誤! 尚未定義書籤。***
字符数据类型 ..... ***錯誤! 尚未定義書籤。***
数字数据类型 ..... ***錯誤! 尚未定義書籤。***
时间型数据类型 ..... ***錯誤! 尚未定義書籤。***
数组 ..... ***錯誤! 尚未定義書籤。***
数据操作 ..... 錯誤! 尚未定義書籤。
在数据类型之间转换 ..... ***錯誤! 尚未定義書籤。***
用于数据操作的函数 ..... ***錯誤! 尚未定義書籤。***
魔法变量 ..... ***錯誤! 尚未定義書籤。***
OID 列 ..... ***錯誤! 尚未定義書籤。***
表管理 ..... 錯誤! 尚未定義書籤。
建表 ..... ***錯誤! 尚未定義書籤。***
使用列约束 ..... ***錯誤! 尚未定義書籤。***

使用表约束.....***錯誤! 尚未定義書籤。***
修改表结构.....***錯誤! 尚未定義書籤。***
删除表.....***錯誤! 尚未定義書籤。***
使用临时表.....***錯誤! 尚未定義書籤。***
视图.....錯誤! 尚未定義書籤。
建立视图.....***錯誤! 尚未定義書籤。***
删除和替换视图.....***錯誤! 尚未定義書籤。***
外键约束.....錯誤! 尚未定義書籤。
作为一个列的约束的外键.....***錯誤! 尚未定義書籤。***
作为一个表的约束的外键.....***錯誤! 尚未定義書籤。***
外键约束的选项.....***錯誤! 尚未定義書籤。***
摘要.....錯誤! 尚未定義書籤。

第九章 事务和锁.....錯誤! 尚未定義書籤。

什么是事务.....**錯誤! 尚未定義書籤。**
将数据改变分组到逻辑单元.....***錯誤! 尚未定義書籤。***
并发多用户访问数据.....***錯誤! 尚未定義書籤。***
ACID 规则.....***錯誤! 尚未定義書籤。***
事务日志.....***錯誤! 尚未定義書籤。***
单用户情况下的事务.....**錯誤! 尚未定義書籤。**
包括多个表的事务.....***錯誤! 尚未定義書籤。***
事务和保存点.....***錯誤! 尚未定義書籤。***
事务的限制.....***錯誤! 尚未定義書籤。***
多用户情况下的事务.....**錯誤! 尚未定義書籤。**
实现隔离.....***錯誤! 尚未定義書籤。***
改变隔离级别.....***錯誤! 尚未定義書籤。***
使用显式和隐式的事务.....***錯誤! 尚未定義書籤。***
锁.....**錯誤! 尚未定義書籤。**
避免死锁.....***錯誤! 尚未定義書籤。***
显式锁定.....***錯誤! 尚未定義書籤。***
摘要.....**錯誤! 尚未定義書籤。**

第十章 函数、存储过程和触发器.....錯誤! 尚未定義書籤。

运算符.....**錯誤! 尚未定義書籤。**
运算符的优先级和结合性.....***錯誤! 尚未定義書籤。***
数学运算符.....***錯誤! 尚未定義書籤。***
比较和字符串运算符.....***錯誤! 尚未定義書籤。***
其他运算符.....***錯誤! 尚未定義書籤。***
内置函数.....**錯誤! 尚未定義書籤。**
过程语言.....**錯誤! 尚未定義書籤。**
PL/pgSQL 入门.....***錯誤! 尚未定義書籤。***
函数重载.....***錯誤! 尚未定義書籤。***
列出函数.....***錯誤! 尚未定義書籤。***
删除函数.....***錯誤! 尚未定義書籤。***
引号.....***錯誤! 尚未定義書籤。***
解析一个存储过程.....**錯誤! 尚未定義書籤。**
函数参数.....***錯誤! 尚未定義書籤。***
注释.....***錯誤! 尚未定義書籤。***
声明.....***錯誤! 尚未定義書籤。***
赋值.....***錯誤! 尚未定義書籤。***
执行控制结构.....***錯誤! 尚未定義書籤。***

动态查询 ..... ***錯誤! 尚未定義書籤。***
SQL 函数 ..... **錯誤! 尚未定義書籤。**
触发器 ..... **錯誤! 尚未定義書籤。**
定义触发器过程 ..... ***錯誤! 尚未定義書籤。***
建立触发器 ..... ***錯誤! 尚未定義書籤。***
为什么使用存储过程和触发器 ..... **錯誤! 尚未定義書籤。**
摘要 ..... **錯誤! 尚未定義書籤。**

引言

歡迎來到 PostgreSQL 資料庫從入門到精通。

在我們生涯的早期，我們開始理解開源軟體的品質。不僅通常情況下它們可以自由使用，而且它們提供極高的品質。如果你發現問題，你可以檢查原始程式碼，理解程式工作過程。如果你找到一個錯誤，你可以自己修復或找別人幫你修復。我們從 1978 年開始試用開源軟體，包括優秀的 GNU 工具，包括 GNU Emacs 和 GCC。我們從 1993 年開始使用 Linux 並愉快地使用 Linux 內核和 GNU 工具建立了一個完整的，自由的計算環境，並且使用 X Window 系統提供了一個圖形化使用者介面。PostgreSQL 是一個採取相同的開源理念的優秀的資料庫系統（更多關於開源和自由的資訊，請訪問 http://www.opensource.org）。

資料庫是非常有用的東西。很多人發現“桌面資料庫”在辦公室和家裡的小應用程式中非常有用。很多網站是資料驅動的，很多內容都由網頁伺服器後面的資料庫提供。隨著資料庫的普及，我們覺得有必要寫一本書介紹資料庫理論和實踐。

我們寫這本書整體介紹資料庫，全面覆蓋現代的關聯式資料庫的能以及怎麼高效使用它們。使用 PostgreSQL 作為他的資料庫的人都沒有覺得 PostgreSQL 在哪方面有什麼不足。它支援優秀的資料庫設計，非常有彈性和擴展性，且運行在幾乎你所能想到的電腦上，包括 Linux，Unix，Windows，Mac OS X，AIX，Solaris 以及 HP-UX。

對了，免得你好奇，我告訴你 PostgreSQL 念做“post-gres-cue-el”（而不是“post-gray-ess-cue-el”）。

本書大致分為三部分。第一部分包括入門，包括資料庫概述（它們是什麼以及它們用來幹什麼），尤其是 PostgreSQL 的概述（怎樣獲得，安裝和啟動以及使用）。如果你同時運行了示例，到第 5 章完成的時候，你將建立起第一個可工作的資料庫並且可以使用一些工具來做一些有用的事情，例如輸入資料和執行查詢。

本書的第二部分深刻地探索關聯式資料庫的核心：SQL 查詢語言。通過示例程式和“做一個嘗試”章節，你將學到資料庫程式設計的很多方面。，從簡單的資料插入和修改，強大的各類查詢到通過存儲過程和觸發器擴展資料庫功能。本章最重要的內容是資料庫無關，所以你從本章獲取的知識能夠讓你在使用其他資料庫時如魚得水。當然，所有的用來說明的示例資源是用 PostgreSQL 來作為示例資料庫。關於 PostgreSQL 的管理和資料庫設計的習慣用法也將在本章完整討論。

本書的第三部分關注於在你的程式中發揮 PostgreSQL 的能力。本章涵蓋了通過大量的程式設計語言連接到資料庫、執行查詢以及處理結果集。無論你是使用 PHP 或者 Perl 開發動態網頁、用 Java 或者 C#開發企業級應用程式或者用 C 寫一個用戶端程式，你將找到你想要的章節。

這是 PostgreSQL 從入門到精通的第二版；第一版在 2001 年發佈。從那時起，每章的內容都根據 8.0 版本的 PostgreSQL 做了升級。我們在本書中有機會補充了新的一章討論通過 C#訪問 PostgreSQL。

第一章PostgreSQL 介紹

本書都是關於一個最近最成功的開源產品，一個名叫 PostgreSQL 的關聯式資料庫。

資料庫開發商和開源開發者都是 PostgreSQL 的熱心擁護者。任何使用程式管理大量資料的人都可以從資料庫中獲得大量益處。PostgreSQL 是一個非常優秀的關聯式資料庫實現，全功能，開源且免費使用。

PostgreSQL 支援大量的主流開發語言，包括 C，C++，Perl，Python，Java，Tcl 以及 PHP。它是最接近工業標準 SQL92 的查詢語言，並且正在實現新的功能以相容最新的 SQL 標準：SQL2003。PostgreSQL 也獲得數個獎項，包括三次被評為 Linux Journal 雜誌編輯選擇獎最佳資料庫（2000,2003 和 2004 年度）以及 2004 年度 Linux 新媒體獎最佳資料庫系統。我們也許我們有點超越自我。你也許想知道到底 PostgreSQL 是什麼，為什麼你要使用它。

本章我們將設置場景為本書的剩餘部分並提供一些關於資料庫的概括性背景知識、不同類型的資料庫、為什麼他們非常有用以及 PostgreSQL 在哪些地方符合這些要求。

基於資料程式設計

基本上所有的非普通電腦程式操作大量的資料，而且大量的應用程式被開發用來處理資料而不是進行計算任務。有些人估計當今世界上 80%的應用程式開發會通過某種方式連接到資料庫中的複雜資料，所以資料庫對於很多應用來說是一種非常重要的基礎。

以資料程式設計的資源很豐富。大部分好的程式設計書籍都包含章節關於建立、存儲和運算元據。有三本書（由 Wrox 發佈）包含以資料程式設計的資訊：

- 《Beginning Linux Programming》第三版覆蓋了 DBM 庫以及 MySQL 資料庫系統。
- 《Professional Linux Programming》包含關於 PostgreSQL 和 MySQL 資料庫系統的章節。
- 《Beginning Databases with MySQL》全面覆蓋了 MySQL 資料庫系統。

靜態資料

資料有很多形態和大小，我們根據資料的不同性質處理資料。有些情況下，資料非常簡單——也許僅僅是一個數字例如 π 的數值存在程式中用來畫圓。程式本身可能包含這個硬編碼的圓周率的值。我們把這種資料叫做靜態資料，它永遠不需要改變。

另一種靜態資料的例子是歐洲一些國家的匯率。在歐元區，貨幣都按固定的保留小數點後 6 位的精度兌換到歐元。假設我們開發一個歐元區貨幣轉換程式。他可能有一個硬編碼的貨幣名稱和基本匯率表，保存對應到歐元的兌換點數。這些匯率永遠不會改變。不過還沒完，因為這張表會隨著歐元區國家數量的增長而增長。當一個國家註冊到歐元區，這個國家貨幣對歐元的匯率將被設定，它將需要被加入這張表中。當這種情況發生，匯率轉換程式需要修改，內置的貨幣表被修改，程式需要重建。這需要在每次貨幣表發生變化時做一次。

一個更好的辦法是應用程式讀取一個包含簡單的貨幣資訊的檔，檔可能包含貨幣的名稱、國際符號以及匯率。之後在這個表需要改變時我們僅僅需要改變 這個檔，而不需要改變程式。

我們使用的這個資料檔案沒有特別的結構；它僅僅包含幾行文本，包含一些資料讓應用程式讀取。它沒有固定的結構。因此我們把這種檔叫做扁平檔。以下是我們的貨幣檔可能的樣子：

```
France     FRF 6.559570
Germany  DEM1.955830
Italy  ITL   1936.270020
Belgium    BEF 40.339901
```

用於資料存儲的扁平檔

扁平檔對於很多應用類型來說非常有用。由於檔保持在可管理大小，我們很容易改變它，扁平檔可能非常符合我們的需求。

很多系統和程式，特別是 UNIX 平臺，使用扁平檔存儲資料或者進行資料交換。UNIX 的密碼檔就是一個例子，它通常情況下像以下的樣子：

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

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

這個例子包含一系列的資訊和屬性組成的記錄。檔被設計成每行包含一個記錄，整個檔則用來保存相關的記錄到一起。某些情形下，這種模式還不夠用，因而，我們需要增加擴展的功能來配合應用程式完成必須的工作。

重複單元和其他問題

假設我們決定擴展貨幣匯率程式（本章前面介紹過）來記錄每個國家的語言，以及人口和麵積。在扁平檔中，我們需要每行一個記錄，每個記錄由幾個熟悉構成。記錄中每個獨立的屬性總在固定的位置；例如貨幣符號總是第二個熟悉。所以我們可以認為按列讀數據，每列總是相同類型的資訊。

為了添加某個國家使用的語言，我們可能會認為我們只需為每一行要添加一個新列。當我們發現一些國家有不止一種官方語言時，我們碰到了一個問題。所以，在我們給 Belgium 的記錄中，我們需要同時包含 Flemish 語和 French 語。對於 Switzerland，我們需要添加四種語言。扁平檔在這是應該看起來像這樣：

```
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
```

這個問題被稱為重複單元。我們現在的問題是重複的資料項目（語言）是在記錄內重複，也就是說資料可能會在列內重複，而不僅僅是記錄。扁平的檔無法應付這類問題，因為它無法判斷什麼時候語言項列完了，開始下一個記錄了。解決這個問題唯一的方法是在檔內添加一些結構，但這種情況下這個檔不再是扁平檔了。

這裡有另一個示例。記錄 DVD 詳細資訊的程式可能需要記錄生產的年份，導演，風格以及演員列表。我們可以設計一個和 Windows 的.ini 檔案格式差不多的檔，就像這樣：

```
[2001: A Space Odyssey]
year=1968
director=Stanley Kubrick
genre=science fiction
starring=Keir Dullea
starring=Leonard Rossiter
…
[Toy Story]
…
```

我們通過引入描述每個記錄類型的標記解決了重複單元的問題。但是，現在你的程式為了獲得需要的資料，需要讀取和解析更複雜的檔。添加記錄以及搜索這種結構就更困難了。我們怎樣才能確保用於風格或分類的描述是從一個特定子集選擇的。我們怎樣才能很容易的搜索到 Kubrick 導演的電影？

由於對資料的需求越來越複雜，我們被迫寫越來越多的代碼用來讀取和存儲我們的資料。如果我們擴展我們的 DVD 管理程式，為 DVD 出租店主包含一些有用的資訊——例如會員資訊，租金，退租資訊和預定資訊，在扁平檔中管理所有這些資訊的希望渺茫。

另外一個通用問題就是大小的問題。雖然包含結構的文字檔可以通過暴力掃描回答複雜的查詢類似於“告訴我所有的最近三個月租過最少一次喜劇的會員”，不單代碼很難編寫，而且性能可能會很低。這是因為程式除了掃描整個檔來查找任何一片資訊外沒有別的辦法，即使是僅僅返回一條資料的問題“誰主演了 2001 年的 A Space Odyssey”。

我們需要的是一種在通用資料處理系統裡用來存儲和檢索資料的方法，而不是總是發明一個個解決方案用於有點不同但非常相似的問題。

我們需要的是一個資料庫和一個資料庫管理系統。

什麼是資料庫管理系統

韋氏線上字典（Merriam-Webster online dictionary，http://www.merriam-webster.com）定義資料庫為一個為（通過電腦）高速搜索和檢索的而特別組織的大量資料集。

資料庫管理系統（database management system，DBMS）通常是一套用來讓程式開發人員從繁重的資料存儲細節和管理中解脫出來的程式庫、應用程式以及工具套件。它還提供了用來搜索和更新記錄的能力。多年來，資料庫管理系統發展出大量的特點用來解決各種各樣特別的資料存儲問題。

資料模型

從 1960 年代到 1970 年代，開發人員通過很多不同的方法建立資料庫來解決重複單元的問題。這些方法從術語上說就是資料庫系統模型。IBM 在這方面的研究為這些模型提供了大量的基礎，這些一直被用到現在。

一個早期的資料庫系統設計的主要驅動力是效率。一個是系統效率更高的通用方法是強制使用固定長度的資料庫記錄。或至少每個記錄有固定的元素（每行有固定的列數）。這可以有效的避免重複單元的問題。如果你是一個過程語言的程式師，你早就知道這種情況了，你可以將每個資料庫記錄讀到一個簡單的 C 資料結構中。現實世界中很少有這樣的簡單的情況，所以我們需要找到一些處理複雜結構的資料的方法。資料庫系統設計者通過引入不同類型的資料庫實現了這種需求。

層次資料庫模型

1960 年代晚期 IBM 的資訊管理系統介紹了資料庫的層次模型。在這種模型中，把資料記錄看成是其他記錄的集合以解決重複單元的問題。

該模型可以用來比較描述如何製造一個複雜的產品組成的材料清單。例如，假設一輛車是一個底盤，一個機構，一個發動機，四個輪子組成。這些主要設備都進一步分解。包括一些汽缸發動機，汽缸頭，曲軸。這些元件是進一步分解，直到我們得到螺母和螺栓構成的每一個汽車零件。

層次資料庫模型到今天一直在使用，包括 Software AG 公司的適配資料庫系統（Adaptable Database System，ADABAS）。

層次資料庫系統有能力通過優化資料存儲在某些具體問題上更加高效；例如判斷哪個汽車使用一個特別的部件。

網狀資料庫模型

網狀模型在資料庫中引入了指標。記錄可以包含到其他記錄的引用。例如，你可以為你公司的每個客戶保留一條記錄。每個客戶隨著時間的過去，給你發了很多訂單（一個重複單元）。由於資料已經排序，所以客戶記錄只包含一個到一條訂單的指標。每個訂單記錄包含該特定訂單的訂單資料，以及到其他訂單記錄的指標。

回到我們的貨幣程式，我們可以通過圖 1-1 的樣子通過記錄結構終止記錄。

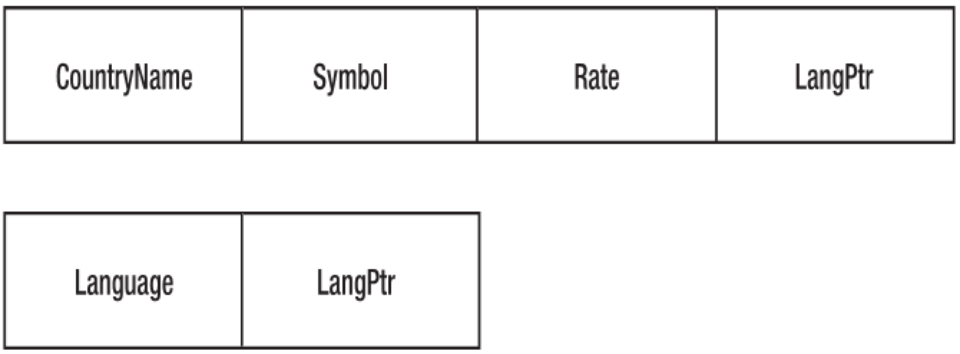


圖 1-1 貨幣程式記錄類型

一旦資料被載入，我們使用一個語言用的鏈表（這裡，就是網路模型）來終止該列資訊，就像圖 1-2 顯示的一樣。這裡展示的兩種不同的記錄類型應該分別存儲，各自一張表。

當然，為了在存儲方面更有效率，在實際上資料庫不會一次次重複語言名字，而可能包含一個協力廠商表用來存儲語言名，並且針對語言名給一個識別字（通常是一個小整數），這個識別字用來在其他記錄中指出對應的語言專案。它被叫做鍵。

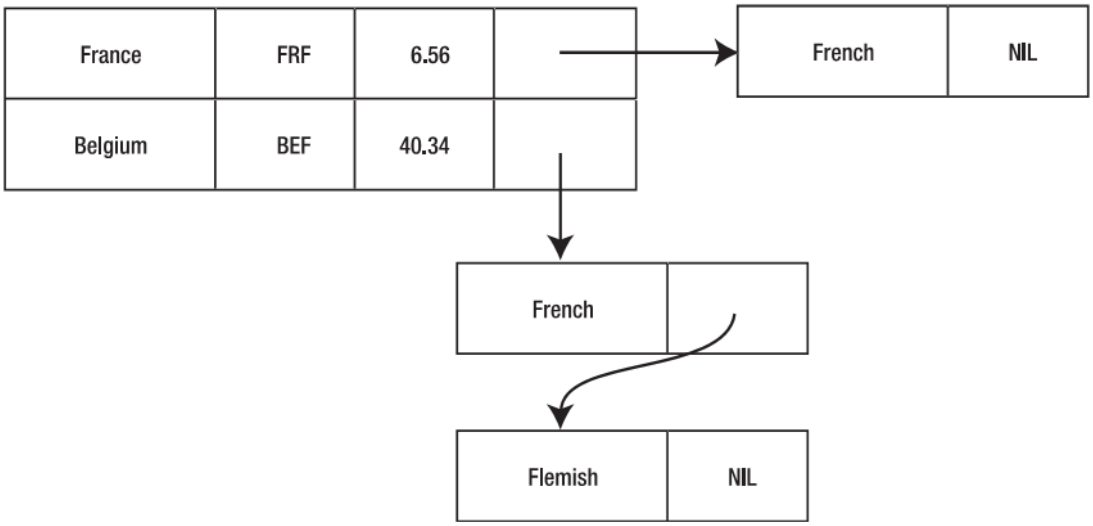


圖 1-2 匯率程式資料結構

一個網路模型資料庫有一些很強的優勢。如果你需要查找關聯到某個記錄的一種類型的所有的記錄（本例中，一個國家的語言），你可以通過開始記錄的指標快速找到他們。

當然，這也有一些不足。如果你需要列出講法語的國家，你需要掃描所有國家的語言鏈表，這種情況在大資料庫中非常慢。這可以通過使用另一個指向語言的指標鏈表來處理，但這很快就變得相當複雜，而且很明顯不能作為一種通用的解決方案，因為你需要預先確定要設計多少指針。編寫使用網狀模型資料庫的程式也非常令人厭惡，因為應用程式通常需要為記錄的修改和刪除而設置和管理指針。

關聯式資料庫模型

隨著 1970 年 E. F. Codd 提出 “一個大型共用資料倉庫的資料的關係模型(A Relational Model of Data for Large Shared Data Banks，請閱讀 http://www.acm.org/classics/nov95/toc.html），資料庫管理系統的理論發生了迅猛的發展。這篇革命性的文章介紹了關係的概念以及表格（通過其存儲的資料）怎樣用於表現現實世界的物件。

現在，一個比驅動最初的資料庫設計和性能更值得關注的東西逐漸變得明晰：資料一致性。該模式比任何更多的早期模型更強調資料完整性的關係。參照完整性是指確保資料庫中的資料在任何時候都合理，因此，例如，所有的訂單都有客戶（我們將在 12 章討論資料庫設計的時候更詳細地討論完整性）。

關聯式資料庫裡頭的一個表裡頭的一條記錄被認為是一個元組的組合，這是你將看到在 PostgreSQL 文檔的某些部分使用的術語。一個元組是一個有序的組成部分或屬性群，他們每一個都有一個定義的類型。

幾個重要的規則定義一個關聯式資料庫管理系統（RDBMs）。所有的元組群必須符合同樣的模式，也就是它們必須擁有相同數量和類型的組成部分，以下是一個元組集的例子：

```
{"France", "FRF", 6.56}
{"Belgium", "BEF", 40.34}
```

每個元組集都有三個屬性：國家名（字串），貨幣（字串）以及匯率（一個浮點數）。在關聯式資料庫中，所有插入這個表的記錄都必須使用相同的格式，所以以下的情況是不允許的：

```
{"Germany", "DEM"}
```

這裡的屬性太少。

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

這裡的屬性太多。

```
{1936.27, "ITL", "Italy"}
```

這裡的屬性類型錯誤（順序錯誤）。

此外，對於元組集的表，不應該有重複的元組。這意味著任何正確設計的關聯式資料庫裡頭的任何表格，不應該有任何重複的行或者記錄。這似乎是一個相當苛刻的限制。例如，在一個記錄客戶的訂單的系統裡，它似乎不允許相同的客戶訂相同的產品兩次。在下一章，我們會發現有一種簡單的方法來解決這種需求，通過添加一個屬性。

記錄中的每個屬性必須是原子的，也就是說，它必須是單獨的一塊資料，而不是另一條記錄或者其他屬性的清單。而且，表中的每條記錄的相應屬性必須相同。技術上來講，這意味著他們必須來自於相同的值的集合或者域。實際上，它意味著他們必須是一個字串，一個整數，一個浮點值或其他的資料庫系統支持的類型。

用於從一個表中區從其他所有的記錄中區分某一條特殊的記錄的屬性（或屬性集）叫做主鍵。在關聯式資料庫中，每個關係，或者表，必須有一個主鍵用來使一條記錄唯一——也就是從表裡的其他記錄中區別出來。

最後一條確定一個關聯式資料庫的構造的是參照完整性。就像我們開始說的，這要求資料庫中的每一條資料在任何時候都有意義。資料庫應用程式師必須確保他們的代碼不會破壞資料庫的這種完整性。想想我們刪除一個客戶的時候將發生的事情。如果我們嘗試從客戶表刪除一條客戶的資訊，我們還需要從訂單表刪除他所有的記錄。否則，我們將留下一些沒有有效客戶的訂單記錄。

我們將在後面的章節讀到更多的關於關聯式資料庫的理論。到這裡，我們足以理解資料庫的關係模型是基於針對集合和關係的一些數學概念，以及這種模型上的一些需要資料庫系統監視的規則。

查詢語言

關聯式資料庫管理系統（RDBMS）誠然提供一些方法用於修改資料，但是它們真正的力量來源於它們有能力允許使用者針對存儲的資料以查詢語句的形式提出問題。不同於很多早期的資料庫設計，這些早期資料庫經常針對資料需要回答的結果構造問題（查詢），關聯式資料庫對於在資料庫設計時未知的問題的回答上更加靈活。

Codd 關於關聯式資料庫的提案使用以下事實：關係定義資料集，資料集能夠被數學方式操作。他建議查詢應該使用一個叫謂詞演算的理論邏輯的分支，並且查詢語言應該是它們的基礎。這將使搜索和選擇資料集前擁有所未有的能力。現代資料庫系統，包括 PostgreSQL，在一種富於表現力，易於學習的查詢語言後隱藏所有的數學邏輯。

第一個實現的查詢語言是 QUEL，在 1970 年代晚期開發的 Ingres 資料庫中使用。使用不同方法另一種查詢語言是 QBE（Query By Example，通過示例查詢）.就在這同時，IBM 的研發中心的一個團隊開發了 SQL（Structured Query Language，結構化查詢語言），通常讀作“sequel”。s

SQL 標準和變化

SQL 作為一種資料庫查詢語言的標準已經非常廣泛地採用，並定義有一系列的國際標準定義。最通用的定義是在 ISO/IEC 9075:1992 中，“資料庫語言 SQL”中。這被簡單的稱為 SQL92。這個標準替代了早期的 SQL89 標準。最新的 SQL 標準是 ISO/IEC 9075:2003，簡稱為 SQL2003。

當前，大多數關聯式資料庫管理系統遵循 SQL92 標準，或者 ANSI X3.135-1992（在美國登記的標準，只有幾頁封面不同）。SQL92 有三個一致性級別：入門級 SQL，動態 SQL 和完備級。到目前為止，最常見的一致性級別為入門級 SQL。

注：PostgreSQL 非常接近 SQL92 的入門級一致性，只有少量的輕微差異。開發人員保持密切關注有關標準，這使得 PostgreSQL 在每次發佈版本後變得更加相容標準。

今天，幾乎所有有用的資料庫系統支持 SQL 到一定程度。理論上，SQL 扮演成一個很好的統一者，因為使用 SQL 作為資料庫介面的資料庫應用程式可以被移植到其他資料庫，而僅僅需要花費少量的時間和精力。但商業壓力決定了資料庫廠商刻意製造自己的資料庫與其他資料庫的區別。由於 SQL 標準未定義現實世界中必不可少的用於執行資料庫管理任務的命令，這導致了 SQL 的變化。所以，Oracle、SQL Server、PostgreSQL 以及其他的資料庫管理系統使用的 SQL 都不盡相同。

SQL 命令類型

SQL 語言由三種命令組成：

- 資料操作語言（Data Manipulation Language，DML）：這是在你 90%時間內會使用的 SQL。這些命令用來在資料庫中插入、刪除、更新、查詢資料。
- 資料定義語言（Data Definition Language，DDL）：有些命令用來建表、定義關係以及控制資料庫的其他結構方面的資訊。
- 資料控制語言（Data Control Language，DCL）：這是一套通常用來控制對資料的訪問許可的命令集，例如定義存取權限。很多資料庫使用者從來不會使用這些命令，因為這通常在大公司使用，而通常情況下會有一個或者多個資料庫管理員專門管理資料庫，他們的工作之一就是控制訪問許可。

SQL 命令簡介

在本書中你將碰到大量的 SQL。這裡，我們將簡單看幾個例子作為介紹。我們將發現我們不需要擔心我們缺少 SQL 基礎知識而無法在這裡使用。

以下有一些在資料庫建立新表的 SQL。以下為建立客戶表的例子。

```
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),
);
```

這張表需要一個由資料庫自動生成的標記作為主鍵。它的類型是 serial，這意味著每當一個客戶被添加，一個唯一的 customer_id 將被按順序建立。客戶的 title 是一個 4 位元元組的文本類型，zipcode 有 10 個位元組。其他的屬性都是最超不超過定義長度的變長字串，某些是必須存在的（被標記為 not null）。

接下來，我們有一些 SQL 語句用來填充我們剛才建立的表。它們非常簡單：

```
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 語句是 SQL 的心臟。它用來建立一個匹配一組特定規則的記錄組（或者記錄的屬性集）的結果集。這個規則可以是非常複雜的。這些結果集可以用來作為通過 UPDATE 語句修改或者通過 DELETE 語句刪除的目標。

以下為一些 SQL 語句示例：

```
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
```

這些 SELECT 語句列出所有的客戶，所有客戶的訂單，並分別計算每個客戶造出來的訂單。我們將在第二章看到這些 SQL 語句的結果，並在第四章中學習更多關於 SELECT 的資料。

注：SQL 命令的關鍵字例如 SELECT 和 INSERT 不區分大小寫，所以它們可以任意使用大小寫。在本書中，為了閱讀方便，使用大寫。

當你閱讀本書的時候，我們將教你大量的，所以當你讀完本書的時候，你將熟悉大量的 SQL 語句並知道怎麼使用它們。

資料庫管理系統的責任

正如我們前面所述，一個資料庫管理系統包括構建資料庫的一組程式以及使用它們的應用程式。一個資料庫管理系統的職責包括以下內容：

- 建立資料庫：有些系統將管理一個大的檔，並在它裡面創建一個或多個資料庫；其他系統可能使用許多作業系統檔或直接使用裸磁碟分割。使用者不必擔心這些檔的低層次結構，因為資料庫管理系統提供了開發者和使用者訪問所需要的所有功能。
- 提供查詢和修改能力：一個資料庫管理系統將擁有一個請求符合特殊規則資料方法，例如某個客戶的所有未完成的訂單。在引入 SQL 標準之前，每個資料庫系統查詢這種結果的方法都大不相同。
- 多工：如果資料庫 被多個應用程式使用，或者同時被多個使用者訪問，資料庫管理系統需要確保每個使用者的請求被處理而不影響其他的用戶。這意味著使用者需要在其他人正在寫入他需要讀取（或者寫入）的資料時排隊等待。它允許在同一時間有很多併發讀取。實際上，不同的資料庫系統支持不同基本的多工，甚至允許可配置的級別，就像我們在第九章討論的。
- 維護一個審計線索：一個資料庫管理系統將保存一個近段時間內資料庫中所有變動的日誌。它可以用於調查錯誤，但其實可能更重要，可以用於在系統故障（例如一個非計畫的斷電）後重建資料。資料備份和事務的審計線索可以用於在磁片故障後完全恢復資料庫。
- 管理資料庫的安全性：一個資料庫管理系統將提供存取控制，因此只有授權的使用者可以操作資料庫中的資料以及維護資料庫本身（的屬性，表和索引）的結構。通常情況下，每個資料庫都會通過一個可以改變任何東西的超級用戶定義不同級別的用戶，從可以添加刪除資料的使用者到只可以讀取資料的使用者。資料庫管理系統架需要有能力添加和刪除使用者，並指定資料庫的哪種功能這個用戶可以使用。
- 維護參照完整性：就像前面所說，很多資料庫提供用於維護參照完整性（資料的正確性）的功能。它將在一個查詢或者修改會破壞關係模型規則的時候報告一個錯誤。

什麼是 PostgreSQL?

現在是時候告訴你到底 PostgreSQL 是什麼了。它是一個包含關係模型和支援 SQL 標準查詢語言的 DBMS（資料庫管理系統）。

PostgreSQL 也非常先進和可靠，並且性能非常高。它基本上可以在任何 UNIX 平臺上運行，包含類 UNIX 系統，比如 FreeBSD、Linux 和 Mac OS X。它也可以在 Microsoft Windows NT/2000/2003 伺服器版本上運行，甚至可以在 Windows XP 上進行開發。並且，就像本章開始提及的，它免費且開源。

PostgreSQL 可以與其他 DBMS 媲美。它基本上包含其他所有商業的或開源的資料庫中你能找到的功能，甚至一些你找不到的功能。

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

- 事務
- 子查詢
- 視圖
- 外鍵參照完整性
- 複雜的鎖
- 用戶自訂類型
- 繼承
- 規則
- 多版本併發控制

從版本 6.5 開始，PostgreSQL 就非常的穩定，這通過針對每個發佈版本都通過大量的回歸測試得以保證。7.x 發佈版比以前版本更接近 SQL92 標準，並去掉了令人討厭的行數限制。

本書中使用的 PostgreSQL，8.x 版本，增加了以下新功能：

- Microsoft Windows 原生支持
- 表空間
- 可以修改列類型
- 時間點恢復

PostgreSQL 被證實在使用中非常可靠。每次發佈都被嚴格控制，BETA 都經過至少一個月的測試。通過龐大的用戶社區和普及的源碼訪問，BUGs 很快就被修復了。

PostgreSQL 的性能在每次發佈都有提升，最新的基準測試顯示在某些條件下，它可以同商業產品相媲美。一些非全功能的資料庫系統性能會比它高出一些，因為這些資料庫在沒有全功能帶來的性能損耗。當然，對於足夠簡單的應用，可以使用扁平檔的資料庫系！

PostgreSQL 歷史簡介

PostgreSQL 可以回溯它的家族樹到 1977 年的加州大學伯克利分校（University of California at Berkeley，UCB）。一個叫做 Ingres 的關聯式資料庫由 UCB 在 1977 到 1985 年開發。Ingres 是一個著名的 UCB 產出，在很多學院和研究團體的 UNIX 電腦上嶄露頭角。為了服務於商業市場，Ingres 的源碼由 Relational Technologies/Ingres 公司加以改良並成為第一個商用 RDBMS。

注：今天，Ingres 已經變成了 CA-INGRES II，一個 Computer Associates 公司的產品。它最近也變成了開源產品。

同時，在 UCB，從 1986 到 1994 年，一個名叫 Postgres 的關聯式資料庫伺服器被繼續開發。同樣，這份代碼被 Illustra 公司拿去發展成一個商業化的產品(Illustra 公司後來被 Informix 公司並購，程式則整合到 Informix的 Universal Server 去了)。在 1994 年，Postgres 被添加了 SQL 功能，且被易名為 Postgres95。

到 1996 年，Postgres 開始變得非常流行，開發人員決定開放它的開發到一個郵寄清單，成功的實現了志願者驅動的 Postgres 發展。這時候，Postgres 經歷了它的最後一次更名，替換“95”標記為更恰當的“SQL”，反映出它現在支援的查詢語言標準。PostgreSQL 誕生了。

現在，一個網際網路上的開發團隊使用其他開源軟體像 Perl，Apache 和 PHP 的同樣的方法開發著 PostgreSQL。用戶可以訪問源碼並貢獻修復代碼、增強功能和建議新功能。PostgreSQL 的官方發佈網站為 http://www.postgresql.org。

已經有很多公司提供商業支援了。例如 EnterpriseDB 公司，期網站地址為：http://www.enterprisedb.com。

PostgreSQL 架構

PostgreSQL 強壯的一個原因源於它的架構。和商業資料庫一樣，PostgreSQL 可以用於 C/S（客戶/伺服器）環境。這對於用戶和開發人員有很多好處。

PostgreSQL 安裝核心是資料庫服務端進程。它允許在一個獨立伺服器上。需要訪問存儲在資料庫中的資料的應用程式必須通過資料庫進程。這些用戶端程式無法直接訪問資料，即使它們和服務程式在同一台機器上。

注：PostgreSQL 還不具有一些企業級商業資料庫的負載均衡和提供擴展的可伸縮性和可恢復性的 HA（High-Availability，高可用性）功能。在 http://gborg.postgresql.org 有一些 PostgreSQL 認可的專案針對增加這些功能在進行中，同時也有一些商業解決方案存在。

這種分開為用戶端和伺服器端的方式可以讓應用程式分散式允許。你可以使用網路來分隔你的用戶端和服務端，使開發的用戶端程式適合使用者的使用環境。例如，你可以在 UNIX 上實現資料庫並建立運行在在 Microsoft Windows 上的用戶端程式。圖 1-3 顯示一個典型的分散式 PostgreSQL 應用。

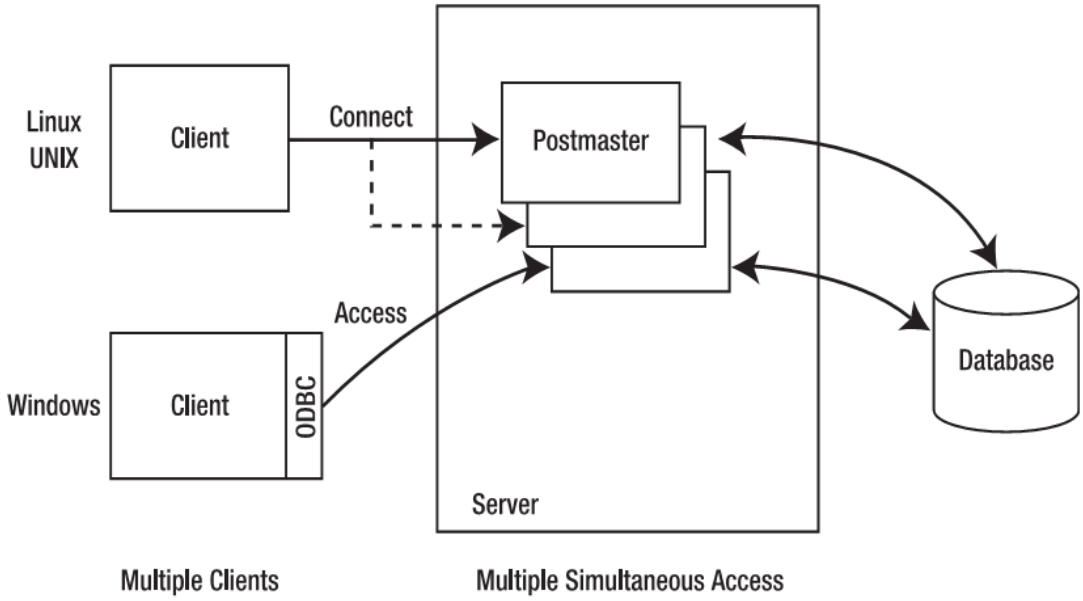


圖 1-3 PostgreSQL 架構

在途 1-3 中，你可以看到幾個用戶端通過網路連接到伺服器。對於 PostgreSQL，這需要是一個 TCP/IP 網路——一個局域網（local area network，LAN）或者甚至是網際網路。用戶端連接到資料庫主進程（在圖 1-3 中叫做 postmaster），主進程為這個用戶端的訪問請求專門建立一個新的服務端進程提供服務。

伺服器專注於資料處理，而不是嘗試控制很多用戶端訪問伺服器上共用目錄中存儲的資料，這讓 PostgreSQL 高效的管理資料的完整性，即使在存在大量的使用者的情況下。

用戶端程式使用專有協定連接到 PostgreSQL。然而，通過在用戶端安裝軟體而為應用程式提供一個標準的介面是可能的，例如開放式資料庫連接（ODBC，Open Database Connectivity）標準或者 Java 程式的 Java 資料庫連接（JDBC）標準。ODBC 驅動的存在運行很現存的應用程式使用 PostgreSQL 作為一個資料庫，包括的微軟的 Office 產品例如 Excel 和 Access。你將在第三、第五和第十三到十八章瞭解不同的 PostgreSQL 連接方法。

PostgreSQL 的 C/S 架構允許任務分工。非常適合於存儲和訪問大量資料的伺服器主機可以用作安全的資料儲存庫。可以為用戶端開發複雜的圖形介面程式。另外，基於網頁的前端可以通過建立網頁模式的結果集到流覽器訪問資料，而不需要另外的用戶端軟體。我們將在第五章和十五章回頭討論這些想法。

通過 PostgreSQL 訪問資料

通過 PostgreSQL，你可以通過好幾種方法訪問你的資料：

- 通過命令列程式執行 SQL。本書自始至終都會用到這種方法。
- 直接嵌入 SQL 到你的應用程式（使用嵌入式 SQL）。我們將在 14 章討論怎樣在 C 程式中使用它。
- 大量不同的程式語言使用功能調用（APIs）準備和執行 SQL，掃描結果集以及執行更新。13 章將覆蓋 PostgreSQL 的 C 語言 API。
- 通過 ODBC（參考第 3 章）或者 JDBC（參考 17 章）標準的驅動間接訪問 PostgreSQL 資料庫，或者使用標準庫，

例如 Perl 的 DBI 庫（參考 16 章）。

什麼是開源？

21 世紀開始的時候，開源軟體提供了大量的東西，PostgreSQL 就是一個很好的例子。但是開源到底意味著什麼呢？開源的條款在施加到軟體上時有非常明確的意義。這意味著軟體在提供的時候，同時提供了源碼。這並不意味著沒有其他條件被施加到軟體的使用上。在某些情況下使用（開源）軟體，一樣需要得到許可。

開源許可賦予你使用、修改和重新發佈它而不需要付許可費用的權力。這意味著你可以在你的機構中覺得合適的地方使用 PostgreSQL。

如果你的開源軟體有一些問題，因為你有源碼，你既可以自己修復它，也可以將代碼交給別人來幫忙修復。有很多商業公司提供對開源軟體的支援，所以如果你選擇使用開源產品，你不必感到被忽視。

有很多種開源許可的變種，一些比其他的更自由。但所有的都支援提供源碼和允許重新發佈。

最自由的是 BSD 軟體許可（Berkeley Software Distribution），它提供“隨便你怎麼處置這個軟體。”

最寬容的許可是伯克利軟體分發（BSD，Berkeley Software Distribution）許可，它實際上說“用這個軟體做任何你想做的。但不提供任何擔保”。PostreSQL 使用的軟體許可（http://www.postgresql.org/about/licence）回應 BSD 軟體許可的精神並套用它的說法，“允許以任何目的使用、拷貝、修改和重新發佈這套軟體以及文檔，不需要任何費用，不需要簽訂任何書面協定，提供以上的版權通告以及這段和以下兩個段落的文字在所有的拷貝中”。緊接著的段落時免責聲明和保證書。

相關資源

有很多關於資料庫的以及關於 PostgreSQL 的進一步資訊的印刷的和線上資源。

對於更對的關於資料庫的理論，可以查看位於 http://www.frick-cpa.com/ss7/default.htm 的 David Frick 的網站的資料庫理論的章節。

PostgreSQL 的官方網站位於 http://www.postgreSQL.org，在這裡你可以找到更多的關於 PostgreSQL 的歷史，下載 PostgreSQL 的拷貝，流覽官方文檔以及更多的東西（包括學習怎樣發佈 PostgreSQL）。

PostgreSQL 也是 Red Hat Database 的基礎，也就是 PostgreSQL- Red Hat 版。你可以在在 Red Hat 的網站的 http://sources.redhat.com/找到更多的關於 PostgreSQL 以及 Red Hat 為它開發的工具。

如需有關開放源碼軟體和自由軟體原則的更多資訊，請花點時間訪問以下兩個網站：http://www.gnu.org 和 http://www.opensource.org。

第二章關聯式資料庫原理

在本章中，我們將研究什麼會組成一個資料庫系統，特別是 PostgreSQL 這樣的現實世界中非常有用的關聯式資料庫系統。我們將從試算表開始，它和關聯式資料庫有很多相同的地方，同時也有重大的局限性。我們將學習像 PostgreSQL 一樣的關聯式資料庫怎麼擁有比試算表強悍的很多功能。順帶，我們將繼續我們的非正式的對 SQL 的研究。

特別是，本章將討論以下主題：

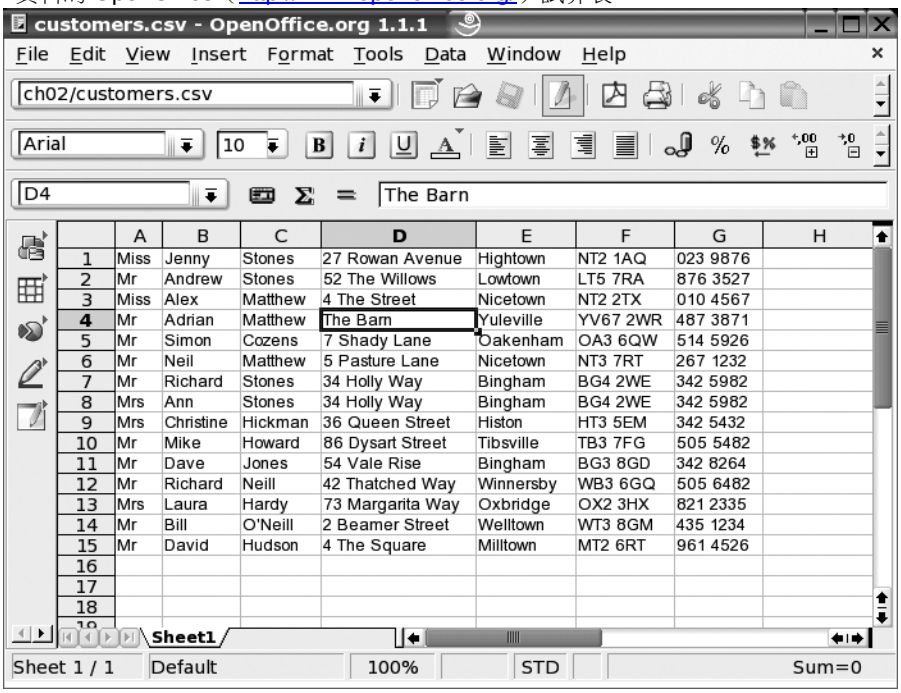
- 試算表：它們的問題和局限
- 資料庫怎樣存儲資料
- 怎樣訪問資料庫中的資料
- 基本的資料庫設計，用到多個表格
- 表之間的關係
- 一些基底資料型別
- NULL 值，用來表示一個未知的值

試算表的局限性

試算表軟體，例如 Microsoft Excel，被廣泛應用於資料存儲和查閱。它很容易用不同的方法排序資料，並通過眼睛直接觀察資料的內容和模式。

不幸的是，人們經常誤解一個很好的用於查看和運算元據的工具為一個存儲和共用複雜的甚至是關鍵業務資料的工具。這兩種需求通常相差甚遠。

很多人都熟悉一種或多種試算表軟體，並能夠按照一定的行列的規則組織一些資料。圖 2-1 顯示了一個示例——一個保存客戶資料的 OpenOffice（http://www.openoffice.org/）試算表。



	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	Tibsville	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								

圖 2-1 簡單的試算表

當然，這樣的資訊容易被看到和修改。每個客戶有單獨的一行，而且客戶的每段資訊保存在一個單獨的列中，就像圖 2-2 中列出來的一樣。一列和行的交集是一個儲存格。

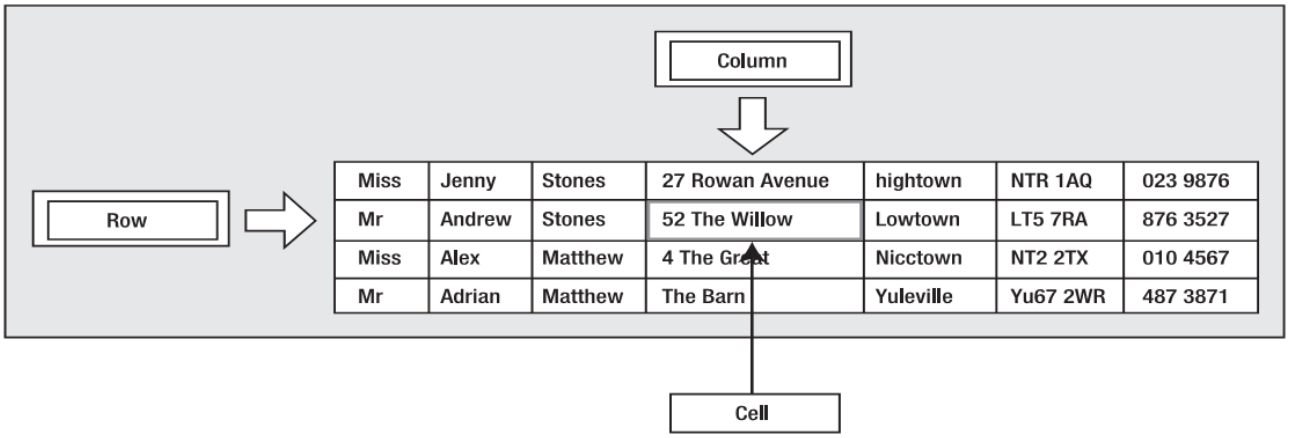


圖 2-2 一些試算表的術語

這個簡單的試算表包含需要在我們開始設計資料庫時順手記下的幾個功能。例如，姓和名分別在不同的列，這使按性和名排序的時候更容易。

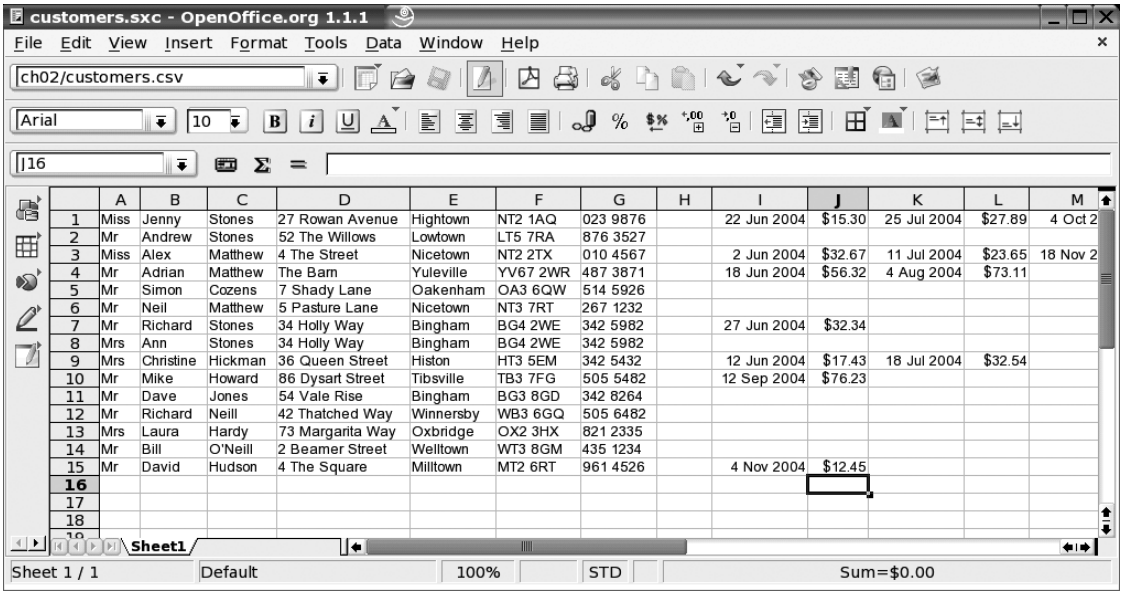
所以，在試算表裡頭保存客戶資訊有什麼錯？試算表很好，因為你：

- 沒有太多客戶
- 每個客戶沒有太多的複雜細節
- 不需要保存任何其他的重複資訊，例如若干客戶的訂單
- 不需要很多人同時更新資訊
- 可以確保保存重要資訊的試算表定期得到備份

試算表式一個了不起的主意，而且是一個可以解決很多類型問題的很好的工具。但是，就像你不會（或者至少不應該）嘗試用牛刀殺雞一樣，有些時候試算表你完成工具的恰當工具。

試想如果在一個有用成千上萬客戶的大公司將所有的客戶資訊保存在一張試算表的主副本中將會怎樣。在一個大公司，很可能很多人需要更新這個清單。雖然檔鎖可以確保同一時間只有一個人可以修改這個清單，隨著需要修改這個檔的人數的增多，他們需要花費更長的時間等待輪到自己修改這個列表。我們所想要的是讓許多人同時讀取，更新，添加和刪除行，並讓電腦確保沒有衝突。顯然，簡單的文件鎖定將不足以有效地處理這個問題。

試算表的另一個問題是它們是絕對的二維的。假設我們還要存儲每個客戶的訂單的細節。剛開始我們可以將訂單細節緊貼著客戶資訊存放。當隨著每個客戶訂單資訊的增長，試算表會變得越來越複雜。考慮到這樣的結果，我們開始為每個客戶添加一些基本訂單資訊，如圖 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	Tibsville	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													

圖 2-3 保存有多條訂單資訊的試算表

不幸的是，它看上去不在那麼優雅了。我們現在又變長的行，且沒有容易的方法計算麼個客戶會佔用多少。甚至，我們會達到每條記錄允許的最大的列數。這就是我們在前面章節碰到的重複組的問題。在試算表中的多表格會有點説明，但他們不是解決這個問題的理想方法。

試算表受到的挑戰

這是一個你怎樣很容易超過試算表限制的例子。一個熟人正嘗試建立一個試算表用來幫做小型生意的朋友存儲資訊。這個生意需要記錄皮革專案，每個專案的價格不單依賴於生產它所需要的時間和勞動，同時還依賴於製造過程中所耗費的皮革的單位成本。物主需要購買不同批次的皮革，每個批次需的單價依賴於皮革的檔次和購買時間。然後他們需要在他們製造皮革製品的時候在庫存中使用先進先出的策略，通常皮革是按批購買的。我們的挑戰是建立一個試算表完成以下功能：

- 跟蹤當前的庫存價格
- 跟蹤有多少批次不同級別的皮革在倉庫中
- 跟蹤當前被使用在生產某個項目上的批次和級別的皮革已經支付了多少錢

經過幾天的努力，他們發現這種表面上簡單的倉儲管理需求出人意料的難以轉換到一個試算表。倉儲記錄多變的特徵不適合試算表的理念。

在這裡我們想指出的是試算表有其合適的位置，但也有其使用的限制。

將資料存入資料庫

從表面上看，關聯式資料庫，例如 PostgreSQL，擁有很多類似於試算表的地方。但是，當你瞭解資料庫的底層結構，你可以發現它複雜得多，主要因為它有能力通過複雜的方法將表格關聯到一起。它可以比試算表有效地存儲更多複雜的資料，並且它有用很多其他功能方便選擇存儲的資料。例如，資料庫可以管理多個使用者同時使用。

讓我們看看存放我們簡單的但表格客戶清單到資料庫，看這麼做有什麼好處。在後面的章節，我們將擴展它並看 PostgreSQL 怎麼說明我們解決客戶訂單的問題。

就像我們在前面章節看到的，資料庫由表（tables）組成，或者用更正式的術語，關係（relations）。我們將在本書中使用表這個術語。表裡頭包含資料行（更正式的叫法是元組（tuples）），並且每條資料行都包含許多列（columns），或者叫做屬性（attributes）。

首先，我們需要設計一個表來保存我們的客戶資訊。好消息是，試算表的資料往往是一個幾乎現成的解決方案，因為它按照一定的行和列保存資料。在開始建立一個基本的資料庫表格前，我們需要確定三件事情：

- 我們需要多少個列來存儲每個專案的屬性？
- 每個屬性（列）的資料類型是什麼？
- 我們怎麼區別不同的行包含的不同項目？

注意資料庫的表的每行的順序不影響資料。在單獨的資料表格中，行的資料可能非常重要，但在資料庫的表中，沒有順序。因為當你查看資料庫的表存儲的資料的時候，資料庫可能會隨意按照它選擇的資料的順序將資料給你，除非你特別告訴它你要按特殊的順序排序資料。如果你需要按特殊的順序查看資料，可以通過指定獲取資料的順序，而不需要例會它存儲的方式。我們將在第四章的 SELECT 語句的 ORDER BY 從句中瞭解到怎麼按順序獲取資料。

選擇列

如果你回顧下圖 2-1 中原始的客戶資訊試算表，你會發現我們已經確定每個客戶需要的合理列：名，姓，郵遞區號等。所以，我們已經回答了我們應該有多少列的問題。

試算表的行和資料庫中的行最重要的不同是資料庫裡頭的表格的列數對於所有的行都是相同的。這對於我們原始的試算表的不是問題。

為每個列選擇資料類型

第二步是為每列的資料確定類型。試算表中允許每個儲存格擁有不同的類型，在資料庫的表中，每個列的類型必須相同。就像大多數程式設計語言一樣，資料庫使用類型來標記不同的資料值。平常，你需要知道所有的基本類型。主要的選擇可以是整數，浮點數，定長文本，變長文本和日期。通常最容易的判斷恰當的資料類型的方法是看看示例的資料。

在我們的客戶資料中，所有的列的類型都可以是文本類型，即使電話號碼看上去好像都是數位。將電話號碼作為數位存儲通常存在以下問題：很容易丟失前導的零，並阻止我們存儲國際撥號符（+），不允許在區號前後寫上括弧等等。顯然，電話號碼遠不止一串數字。回過頭來，用字串存儲電話號碼可能不是最好的選擇，因為我們可能無意地插入各種其他字元，但至少會比使用數位類型要好點。初始設計可以在之後做優化。

我們會發現頭銜（女士，先生，醫生）的長度通常比較短——通常少於四個字元。類似的，郵遞區號也有固定長度。因此，我們可以設置這些列為固定長度，但設置其他的所有列為變長的，因為比方說沒有簡單的辦法判斷一個人的名字有多長。

我們將在本章靠後的“基礎資料類型”小節以及第八章討論 PostgreSQL 的資料類型。

標記行的唯一性

我們在轉換我們的試算表到資料庫表格的最後的問題有點微妙，它牽涉到資料庫管理表和表之間的關係。我們需要確定什麼使資料庫中一條客戶資料記錄區別於另一條客戶記錄。換句話說，我們怎麼區別我們的客戶？在試算表中，我們不趨向於關心區別客戶的細節。但是，在資料庫設計中，這是一個關鍵問題，因為關聯式資料庫的規則需要從某個方面區別每條記錄的唯一性。

最明顯的解決方案好像是通過客戶名來區別客戶，但不幸的是，這通常不足以區別。因為很有可能兩個客戶擁有同樣的名字。另外一項你可以選擇的是電話號碼，但問題是如果兩個客戶住在於一起呢？在這個時候，你可能建議使用名字和電話號碼的組合。

當然，不大可能兩個客戶同時擁有相同的名字和電話號碼，但是，這種方法不但很不雅觀，而且還有另一個潛在問題。如果客戶更換電話服務商，電話號碼發生變化，將發生什麼。在我們的定義中，唯一的客戶必須是一個客戶。因為他不同於我們已有的客戶。當然，我們知道它是有不同的電話號碼的舊客戶。在資料庫中，選擇一個可能會變動的功能作為客戶的唯一標識是個壞習慣，因為管理唯一標識的變動非常麻煩。

這種唯一標識的問題，經常出現在資料庫設計中。我們應該做的是尋找一個主鍵（primary key）——一種最容易的用於區別一個客戶的資料行於其他所有行的方法。遺憾的是，我們還沒有成功，但所有的也沒有失敗，因為標準的解決方案是為每個客戶分配一個唯一的數位。

我們簡單地為每位元客戶分配一個唯一的數位，然後，爽，我們有一個明顯的方法區別客戶，而不管他們是否改變電話號碼或者換到其他的住所，甚至改變他們的名字。這種在實際資料中沒有辦法選擇其他列而附而加的作為一個唯一的鍵值的鍵，叫做代理鍵。在現實世界資料中即使存在一些特殊的資料，資料庫中葉經常這樣做，提供序列資料類型來幫助解決這類問題。我們將在後面的“基礎資料類型”小姐討論這個類型。

我們已經完成為我們的初始表格做一個資料庫設計，現在是時候存儲我們的資料到資料庫中了。圖 2-4 展示了在 PostgreSQL 資料庫中我們的資料在 windows 或者 Linux 主機的終端視窗中通過一個簡單的命令列工具 psql 顯示出來的樣子。

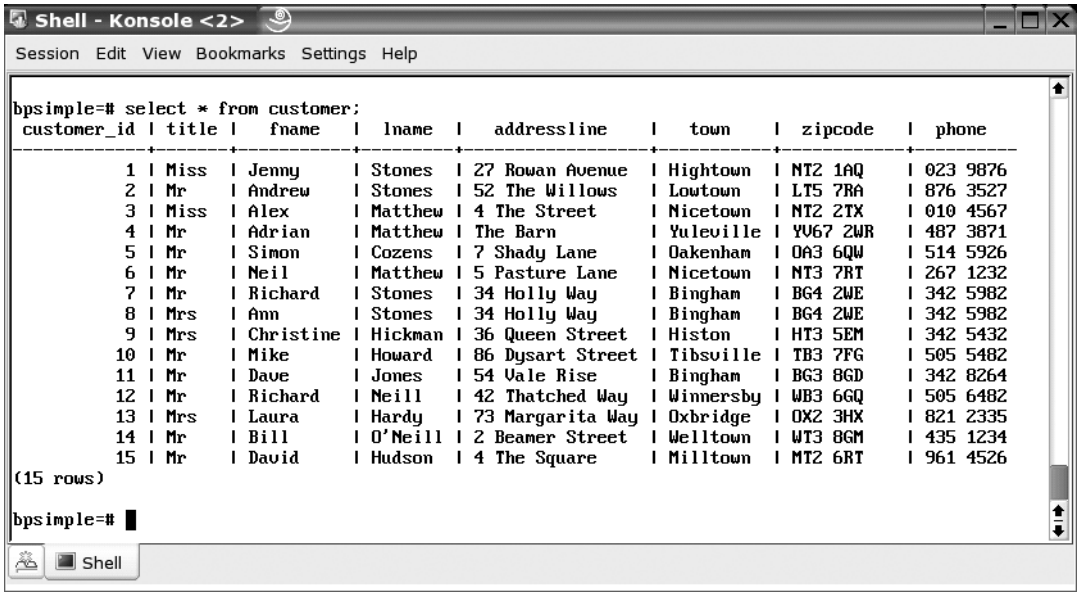


圖 2-4 通過命令列從資料庫中查看我們的資料

注意我們添加了附加的一列，叫做 customer_id，作為我們參考客戶的唯一方式。它是這個表中我們的主鍵。就像你所看見的，資料看上去就像一個試算表，按行列佈局。在後面的章中，我們將講解在實際中定義資料庫表格，存儲和訪問資料，但我們可以確信這些都不難。

在資料庫中訪問資料

你可以很容易地通過命令列工具 psql 查看你在 PostgreSQL 中的資料，如圖 2-4 所示。但是，PostgreSQL 不僅限於在命令列中使用。圖 2-5 顯示更友好的圖形介面工具 pgAdmin III，它是可以從 http://www.pgadmin.org 獲得的免費工具，它從 8.0 開始已經和 Windows 的 PostgreSQL 捆綁發佈了。我們將在第五章看到更多的圖形介面介面。

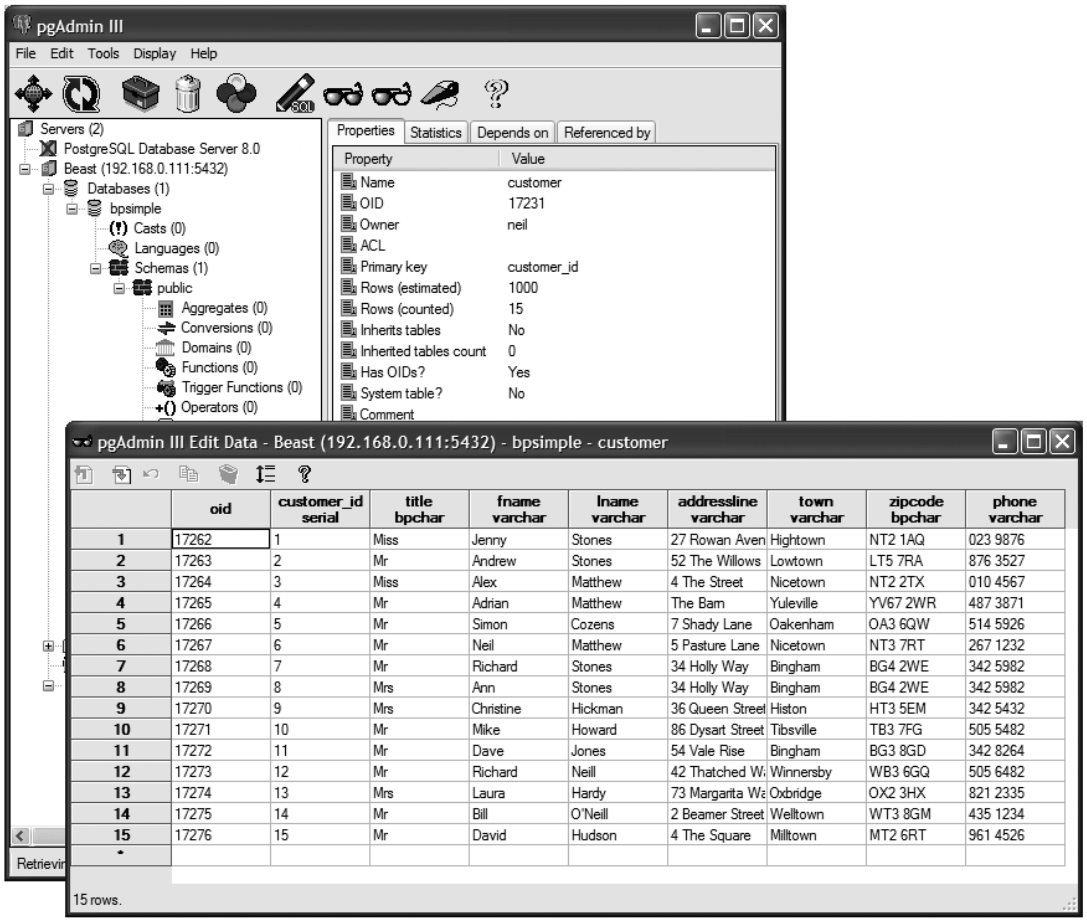
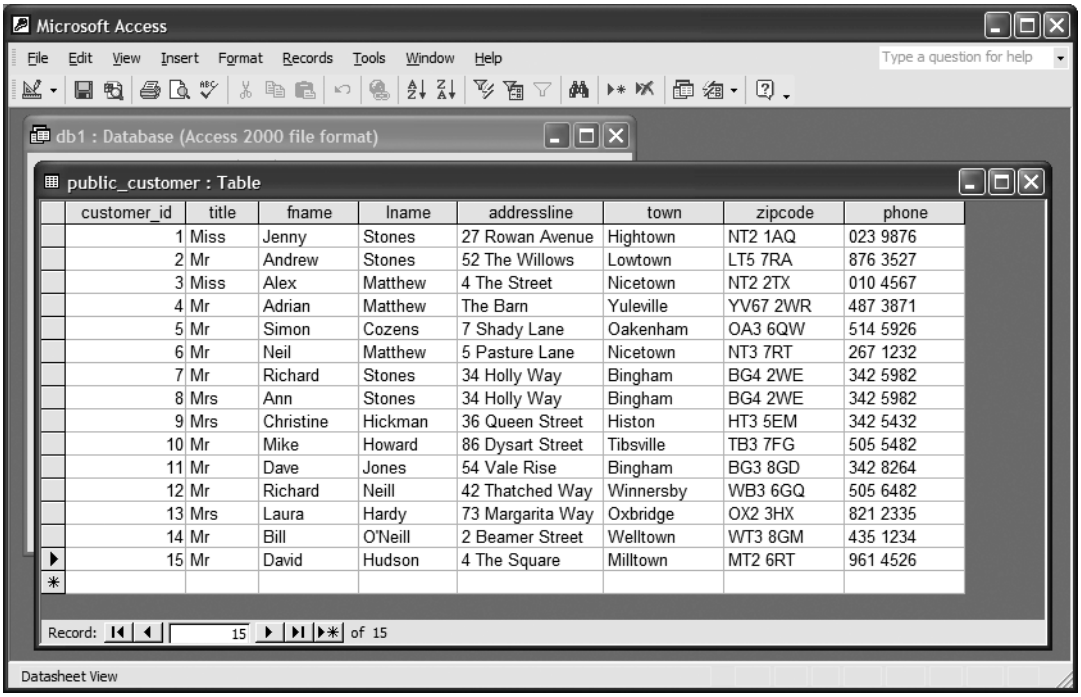


圖 2-5 通過 pgAdmin III 查看資料庫中的客戶資料

通過網路訪問資料

當然，如果我們只可以在存儲資料的同一台機器訪問我們的資料，這情況和通過共用單個的試算表給不同的用戶沒有太多改觀。

PostgreSQL 是一個機遇伺服器的資料庫，就像前面章節所說的，一旦配置完成，我們可以通過網路接受客戶的請求。雖然用戶端可以和資料庫伺服器在同一台機器，對於多用戶訪問來說，這實在是小菜一碟。對於微軟的 Windows 用戶，因為有 ODBC 驅動，所以我們可以使用任何支援 ODBC 的 Windows 桌面應用程式連接到保存我們資料的伺服器。圖 2-6 顯示 Windows 上的微軟 Access 軟體訪問一個在 Linux 主機上的 PostgreSQL 資料庫。這就是通過 ODBC 連接經過網路訪問的外部表。



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	Tibsville	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

圖 2-6 通過微軟的 Access 訪問以上相同的資料

現在我們可以同時從很多機器通過網路訪問這些相同的資料。我們只有一份資料，保存在中央伺服器中，可以通網路從過多個桌面程式同時訪問。

我們將在第五章瞭解配置 ODBC 連接的技術細節。

處理多用戶訪問

和所有的關聯式資料庫一樣，PostgreSQL 可以自動確保對資料庫資料修改的衝突不會發生。它確保各個使用者在使用資料的時候感覺好像訪問全部的資料都不受限制，但在幕後，PostgreSQL 監視著改變且避免同時更新的衝突。

這種表面上讓很多用戶可以同時讀寫同樣的資料，但實際上確保其一致性的能力，對於資料庫來說是非常重要的功能。當一個用戶修改了一列，你要麼看到它變化前的樣子要麼是變化後的樣子，從不會看到修改一半的樣子。

一個經典的例子是銀行的資料庫在兩個帳戶間轉帳。如果在轉帳的時候，一些人正在生成一個彙報所有金額的報表，確保綜述正確就非常重要了。對於報表來說錢在哪個帳戶在報表生成的時候無關緊要，但重要的是報表無法看到中間點，也就是一個帳戶計入借方但另一個帳號還沒計入貸方的時候。

像 PostgreSQL 一樣的關聯式資料庫都隱藏了任何中間狀態，所以中間狀態不會被其他用戶發現。術語上說這叫隔離。報表操作從轉帳操作隔離開來，所以它看上去是在其之前或者之後發生，但絕不會同時發生。我們將在第九章討論事務的時候回顧隔離的概念。

數據分片和分塊

我們現在知道了當資料存在於表中後，訪問它是多麼容易。讓我們首先看看我們實際上應該怎麼處理資料。我們通常在大的資料集上會執行兩類基本操作：選擇符合特定值的集合的行和選擇資料的部分列。在資料庫術語中，他們分別叫做選擇和投影。它們聽上去有點複雜，但實際上選擇和投影都非常簡單。

選擇

讓我們從選擇開始，也就是我們選擇行的子集。假設我們想知道我們住在 Bingham 的客戶。讓我們回到 PostgreSQL 的標準命令列工具 psql 來看我們怎麼使用 SQL 語言讓 PostgreSQL 獲得我們需要的資料。我們要用的 SQL 命令非常簡單：

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

如果你鍵入你的 SQL 語句（通過命令列的工具 psql），你需要在末尾加入一個分號。分號告訴 psql 已經到達命令的末

尾了，因為很長的命令可能擴展到不止一行。在本書中我們通常會顯示分號。

PostgreSQL 通過返回 customer 表中所有的 town 為 Bingham 的行作為回應，就像圖 2-7 所示。

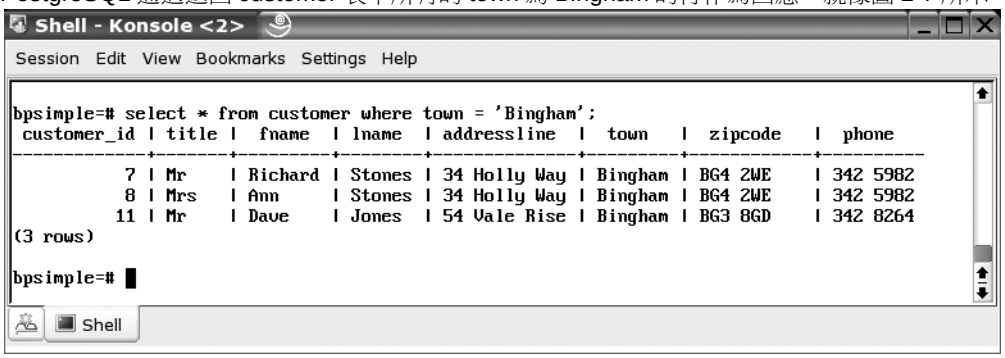


圖 2-7 選擇資料行的子集

所以，挑選表中某些行的行為就叫做選擇。就像你所看到的，這非常容易。不用擔心 SQL 語句的細節，我們將在第五章正式的回顧它。

投影

現在我們來看看投影，也就是選擇表中的某些列。假設我們僅僅需要選擇客戶表中的姓名。請記住我們分別把這兩個列叫做 fname 和 lname。選擇名字的命令也非常簡單：

```
SELECT fname, lname FROM customer;
```

PostgreSQL 通過返回恰當的列作為回應，如圖 2-8 所示：

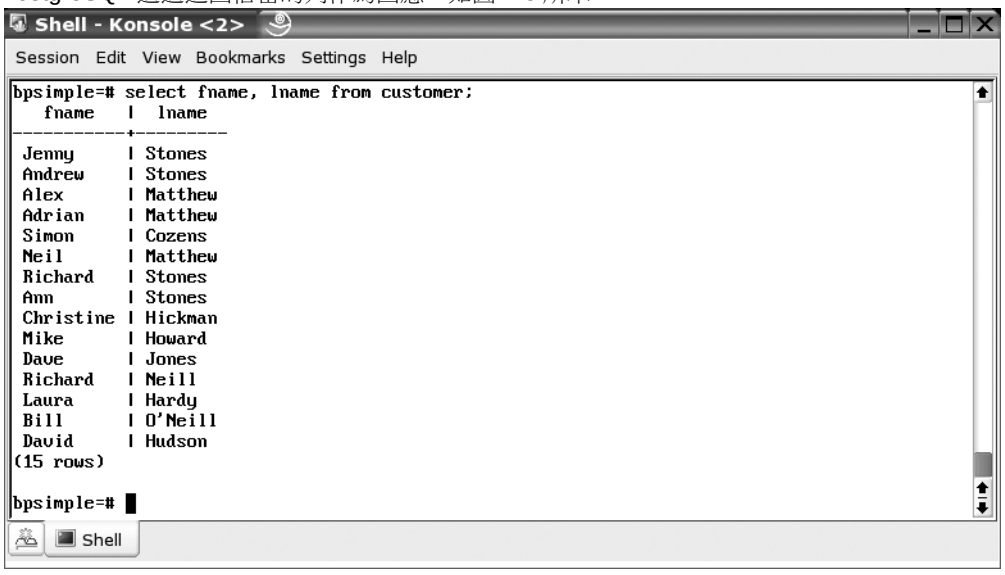


圖 2-8 選擇資料列的子集

你當然可以假設某些時候我們需要在資料中同時執行以上兩種操作；也就是說選擇某些行中的某些列。這也可以通過非常簡單的 SQL 實現。例如，假設我們需要知道住在 Bingham 的客戶的姓名，我們只要簡單的將以上兩條 SQL 語句組合成一條簡單的語句：

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

PostgreSQL 如圖 2-9 一樣響應我們的請求：

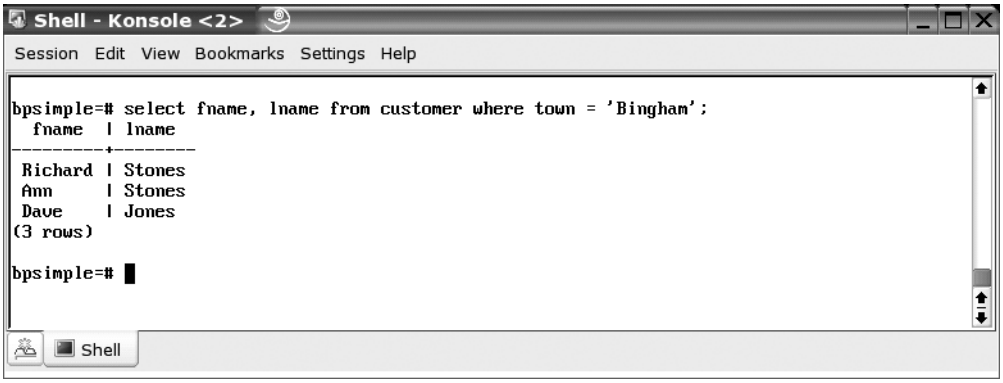


圖 2-9 同時選擇行和列的子集

有個東西需要非常注意。在很多傳統程式設計語言例如 C 或者 Java 中，當在檔中查找資料的時候，我們需要編寫一些代碼掃描檔中的所有行，並每次找到我們需要的城鎮的時候，列印出名字。雖然可以擠壓代碼讓他們變成一行代碼，它將是非常長和複雜的一行，不像這裡的 SQL 那麼簡潔。這是因為 C 和 Java 等類似的語言從本質上說還是一種過程語言。

你在這些語言中指出電腦怎麼工作，對於 SQL，用術語講這是一種描述性語言，你告訴電腦你要什麼，PostgreSQL 通過某些內部邏輯處理你要的任務。

如果你從沒有使用過描述性語言你也許會覺得有點古怪，但是如果你開始使用這種想法，你會發現告訴電腦你要什麼很明顯是一種很好的想法，而不是告訴它要怎麼做。你會覺得很奇怪為什麼到現在為止你才遇到這麼好的語言。

增加資訊

到現在為止我們所碰到的都是通過我們的資料庫類比試算表的單個工作表，而且我們也剛剛接觸到 SQL 功能的表皮。就像我們將要在本書中看到的，PostgreSQL 一類的關聯式資料庫富有大量的有用功能，這讓他們的工作能力大大超越了試算表。其中一個資料庫最重要的能力就是它們有能力將表與表之間的資料連接在一起，這就是我們現在將要讀到的。

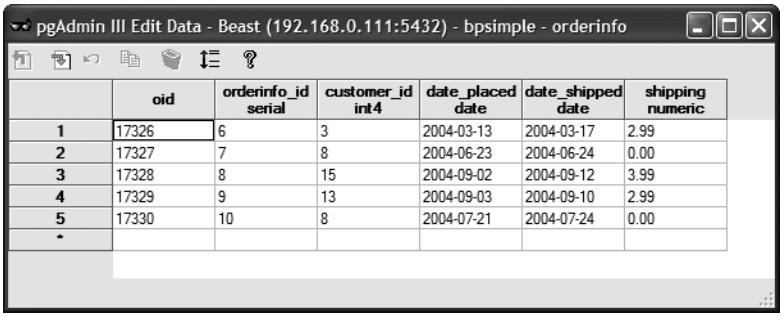
使用多重表格

回到我們關於客戶訂單的問題，也就是在為每個客戶存儲擴展訂單資訊時突然讓我們的簡單客戶表格變得非常凌亂的問題。在我們開始不知道我們的客戶會有多少訂單的時候要怎麼存儲客戶的訂單？你也許會從本章的標題就能猜測到，在關聯式資料庫中解決這種問題的辦法是增加一個表格存儲這些資訊。

就像我們設計客戶表一樣，我們從確定我們要存儲的每個訂單的資訊開始。現在，讓我們假設我們要存儲下訂單的客戶的名字，下訂單的日期，訂單發出日期以及發貨方式。和 customer 表一樣，我們也需要為每條訂單添加一個唯一參考數位，而不是假設哪個資訊可能是唯一的。沒有必要再存儲所有的客戶細節了。我們已經知道通過 customer_id，我們可以在 customer 表中找到客戶的細節。

你也許覺得奇怪為什麼我們忽略掉了訂單的細節。當然，對於大多數客戶，這是一個很重要的方面——他們想要知道他們訂了什麼內容。如果你認為這是一個和不知道客戶有多少訂單一樣類似的問題，你非常正確。我們不知道每個訂單有多少專案。重複組的問題還沒遠離你。我們將暫時把這個問題放下並在後面的“進行簡單的資料庫設計”章節解決這個問題。

圖 2-10 顯示了我們的訂單資訊表，顯示了一些示例資料。而且是通過圖形介面的 pgAdmin III 工具顯示的。



	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
*						

圖 2-10 在 pgAdmin III 中顯示的一些訂單資訊

我們沒有在這張表中放太多資料，因為少量的資料更容易做實驗。你將發現一個擴展的列，叫做 oid，它不屬於我們的使用者資料。這是 PostgerSQL 內部使用的一個特殊列。當前版本的 PostgreSQL 默認為表建立這個列，但在“SELECT *”

命令中它是被隱藏的。我們將在第八章討論這個列。

通過關聯（Join）操作關聯一個表

我們現在在資料庫中存有我們客戶的細節，以及它們訂單的概要細節。在很多情況下，這和使用兩張試算表沒有區別：一張存儲客戶細節，另一種存儲訂單細節。現在是時候關注我們怎麼組合使用這兩張表，開始發覺資料庫的能力了。我們可以同時從這兩個表選擇資料。這就叫做關聯（Join），在選擇和投影之後，這是第三類最常用的資料檢索操作 SQL。

假設我們需要列出所有的訂單以及下訂單的客戶。在像 C 一樣的過程語言中，我們需要編寫代碼掃描其中一個表，也許從 customer 表開始，然後為每個客戶列印他們的訂單。這不難，但編寫這些代碼當然也非常費時和乏味。我敢保證你會很樂意知道我們可以通過 SQL 更容易的找到答案：通過關聯操作。我們所要做的是告訴 SQL 三件事情：

- 我們需要的列
- 我們需要檢索資料的表
- 這兩個表之間怎麼關聯

我們需要的命令在前面的章節中出現過：

```
SELECT * FROM customer, orderinfo WHERE customer.customer_id = orderinfo.customer_id;
```

你也許會猜想，它將從兩個表中請求所有的列，並告訴 SQL customer 表的 customer_id 列保存的資訊與 orderinfo 表的 customer_id 保存的資訊一樣。注意方便的 table.column 標記讓我們讓我們有能力同時指出表名和這個表的列。命令中的 “*” 意味著所有的列。我們可以使用列的名字代替它來選擇一些特殊的列，例如假設我們只需要名字和數量。

現在我們的資料庫已有一些表格和資料，我們可以在圖 2-11 中看看 PostgreSQL 的回應。

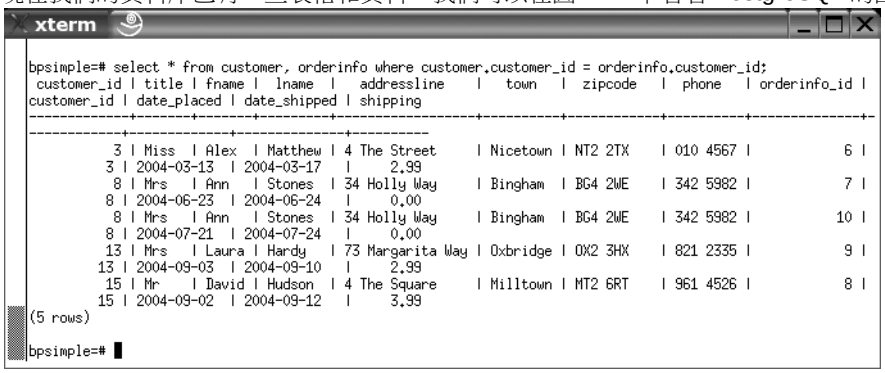


圖 2-11 通過一次操作從兩個表中選擇資料

這有點點凌亂，因為為了匹配視窗，資料被自動換行了，但是你可以看到 PostgreSQL 怎麼響應我們的查詢，而不需要我們指出怎麼精確地解決問題。

再往前走一點點，看看我們可以用在這兩個表上的義工更複雜的查詢。假設我們想看看不同客戶給我們下訂單的頻率。很明顯這需要一個高級一點的 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;
```

這是個複雜點的 SQL，你可以發現我們不需要告訴 SQL 怎樣回答問題的細節；我們只要通過 SQL 精確地指出問題。我們也只需要子一條簡單的語句中完成了它。圖 2-12 顯示了 PostgreSQL 怎麼回應。

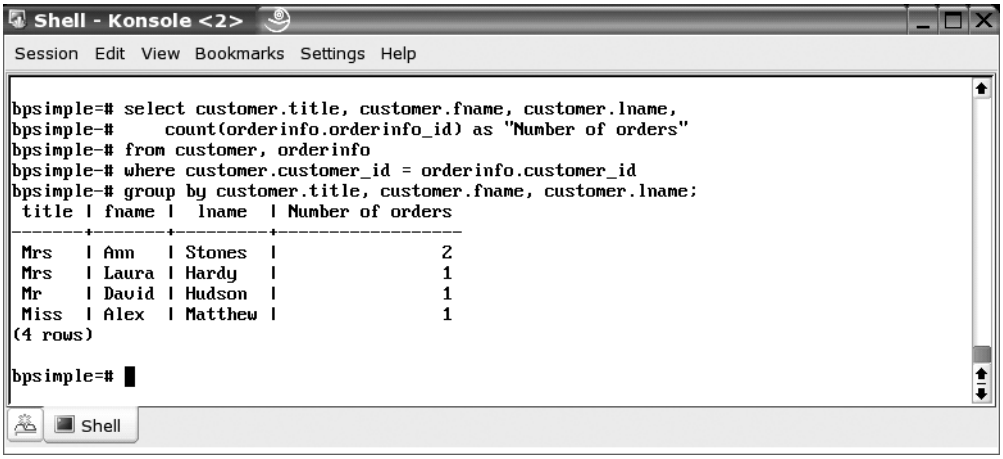


圖 2-12 查看訂單數量

一些資料庫專家喜歡通過直接在命令列工具的視窗中直接敲入 SQL，它在某些時候非常有用，但它並不是所有人的愛好。如果你喜歡通過圖形介面建立你的查詢，這也不是問題。例如就像在本章早些時候指出的，你可以簡單地通過 ODBC 驅動程式通過 Windows 的圖形化使用者介面（GUI，Graphical User Interface）訪問資料庫。圖 2-12 顯示在 Windows 機器中的 Acccess 軟體通過 PostgreSQL 的 ODBC 驅動程式連接到外部的表格，設計和執行的相同的查詢。我們也會在第五章看到其他的 GUI 工具.。

在我們這個環境中，資料仍然存儲在一台 Linux 主機中，但使用者基本不需要知道這些技術細節。通常，在本書中，我們會在命令列中使用教學 SQL，因為在進入更複雜的 SQL 命令前，你可以在這裡學到很多基礎知識。當然，也歡迎您使用 GUI 工具而不是命令列工具來構建你的 SQL；這是你自己的選擇。

設計表

到現在為止，我們在資料庫中只有兩個表，而且我們還沒有真正討論過關於我們怎麼在每張表中做什麼，除了在一些看似有理的非正式設計。這種包含表、列和關係的設計，正確的說法應該叫做模式（schema）。

如果資料很複雜，設計擁有數十個表的資料庫模式將相當的有挑戰性。資料庫設計者因為善於完成這類工作而賺大把的錢。對於不到十個表格的關係簡單的資料庫，也許只需要按照一些基本的經驗規則就可以得到很好的設計，而不需要非常正式的規則。

在本屆，我們將看我們將開始建立的簡單示例資料庫，並指出用來決定我們需要哪些表的一些方法。

理解一些優秀的基本規則（Basic Rules of Thumb）

當一個資料庫被設計，它經常會被規範化；也就是說，一堆規則被應用來確保資料被按照一定的方式打破。在十二章中，我們將以正式的方式觀察資料庫設計。但作為啟動，我們需要的只是一些簡單的基本規則。這些規則只是用來說明你理解我們將用來在以後的章節探索 SQL 和 PostgreSQL 的初始資料庫。我們強烈建議你不僅僅只閱讀這些規則然後就匆忙地設計一個有 20 個表格的資料庫。至少要讀完第 12 章你才能按照你的意願做設計。

注：如果你對標準化形式非常感興趣，我們建議你閱讀 Joe Celko 的《SQL for Smarties》。它有一些各種各樣優秀的標準定義，以及很多 E.F.Codd 博士提出的關係模型的規則以及 SQL 使用的示例。

規則一：將數據拆分成列

第一條規則是將每一塊資訊或者資料屬性放入單獨一列。這對於很多人來說都很自然，可以假設他們很自然地都是這麼想的。在我們原始的試算表中，我們已經很自然地將每個客戶的資訊拆分成不同的列，例如名字將和郵遞區號區別開。

在試算表中，這條規則只是讓對資料的工作更簡單；例如，按郵遞區號排序等。然而在資料庫中，必須將資料差分成不同的屬性。

為什麼這在資料庫中這麼重要？從實踐的觀點上看，你很難說清楚你需要位元址列中的第 29 到第 35 字元之間的資料，因為它碰巧是郵遞區號。很有可能有這種規則不適用的地方，因而你可能取到錯誤的資料塊。另一個需要將資料正確拆分的原因是因為每一列資料必須有相同的類型，而不像試算表，它對資料列的類型很寬容。

規則二：有一個唯一標記來標識每行

你會記起在本章開始的時候關於怎麼標記試算表裡頭每行資料的問題，我們糾結於什麼可能是唯一的。就像說到的，這是因為沒有主鍵。一般而言，不需要一個單獨的列是唯一的，也許是兩個列的組合，或者甚至是三個列才能唯一標記一行。你可能發現需要超過三列才能唯一標記一行資料，這可能很少見，也許可能是一個錯誤。

無論如何，必須有一個絕對必然的方法，當我查看某個特定的列或者一組列的內容時，我知道它將與表中其他所有的行有所不同。如果你無法找到這麼一個列，或者需要找到超過三個列的組合來唯一標記每行，你需要增加一個列來完成這種目的。在我們的客戶表中，我們添加了一個叫做 customer_id 的列來標記每行。

規則三：移除重複資訊

回到我們嘗試存儲訂單資訊到客戶表的時候，由於重複組的問題，表格看上去凌亂不堪。我們必須為每個客戶重複訂單資訊很多次。這意味著我們永遠無法知道訂單需要多少列。在資料庫中，表的列數實際上在設計的時候就已經固定了。所以我們必須在我們存儲資料前預先確定我們需要多少列，每個列的類型是什麼以及每個列的名字。永遠不要嘗試在單獨的一行中存儲重複的資料組。

圍繞這一約束的解決方法就是我們針對訂單和客戶資料的方法：拆分數據到不同的表中。然後在你需要同時從兩個表取資料時你可以關聯這些表到一起。在我們的示例中，我們使用 customer_id 來關聯這兩個表格。更正式的，我們有了一個多對一的關係；也就是說，我們可以從一個單獨的客戶處獲得多條訂單。

規則四：正確地命名

這可能是最難很好實現的規則。我們怎麼叫一個表或者列？表和列應該有簡短且有意義的名字。如果你無法確定怎麼叫一個東西，這通常意味著你的表和列設計不是很恰當。

很多資料庫設計者都有他們自己的個人優秀規則或者命名約定用以確保表和列的命名在資料庫中的一致性。不要讓一些表名用單數而另一些用複數。例如，不要給一個表命名為 office 而另一個命名為 departments，用 office 和 department 就好了。如果你為一個表的識別欄位的名字指定的規則為表名加“_id”，請遵守它。如果你使用縮寫，就總是使用縮寫。如果表中的一個列是另一個表的鍵值（外鍵，我盟將在第十二章講解），儘量使用相同的名字。在一個複雜的資料庫中，如果命名方式不一致，將非常令人討厭，例如 customer_id、customer_ident、cust_id 以及 cust_no。

完成這表面上簡單的正確命名這個目標非常有挑戰性，但獲得的結果是維護起來相當簡單。

建立一個簡單的資料庫設計

我們可以通過實體關係圖畫出我們的資料庫設計，或者模式。對於我們的兩個表的資料庫，這樣的關係圖應該像圖 2-14 一樣。

注：一個實體關係圖是用一個圖形方式表現我們資料的邏輯結構。它可以說明我們形象化表現我們不同的資料實體怎麼關聯到另一個實體。

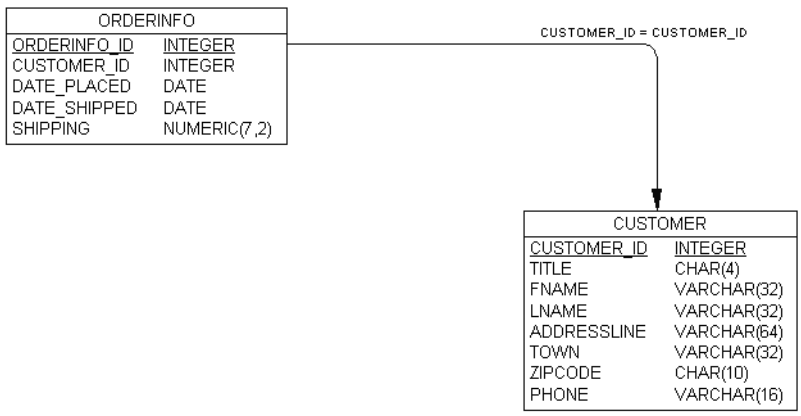


圖 2-14 單的實體關係圖

這張圖顯示了我們的兩張表，列和資料類型，以及每個列的大小，它還告訴我們 customer_id 用來將兩張表關聯到一起。注意箭頭從 orderinfo 表指向 customer 表，這意味著對於 orderinfo 的每一個條目，在 customer 表中最多只有一條對應的條目，但對於每個客戶可能有多條訂單。還要注意的是一些列有底線，它指出這些列被確保為唯一的。這些列構成了這些表的主鍵。

你必須記清楚一對多關聯性是怎麼關聯的；如果在這裡迷糊了，將帶來一大堆問題。你還應該注意到我們很小心地命名用來關聯兩個表的列的名字均為 customer_id。這不是必要的。我們可以可以分別叫他們 foo 和 bar 如果我們願意，但是就像前面小節所說的，命名一致性對於長期運行非常有幫助。

下一步是擴展我們非常簡單的兩表設計為稍微現實點的情況。我們將設計它為一個簡單的訂單管理資料庫，叫做 bpsimple。

在兩個表之上擴展

很明顯，到現在為止我們擁有的資訊缺乏每個訂單的詳細條目。你可能記得我們故意省略每條訂單的實際條目，並承諾會回顧這個問題。現在是時候為每個訂單增加實際條目了。

我們碰到的問題是我們預先無法知道每條訂單將會有多少條條目。這和當初我們不知道一個客戶將有多少條訂單的問題一樣。每條訂單可能會有一條、兩條、三條或者一百條條目。我們必須拆分客戶的訂單以及訂單的內容資訊。基本上，我們要做的就像圖 2-15 中所展示的一樣。

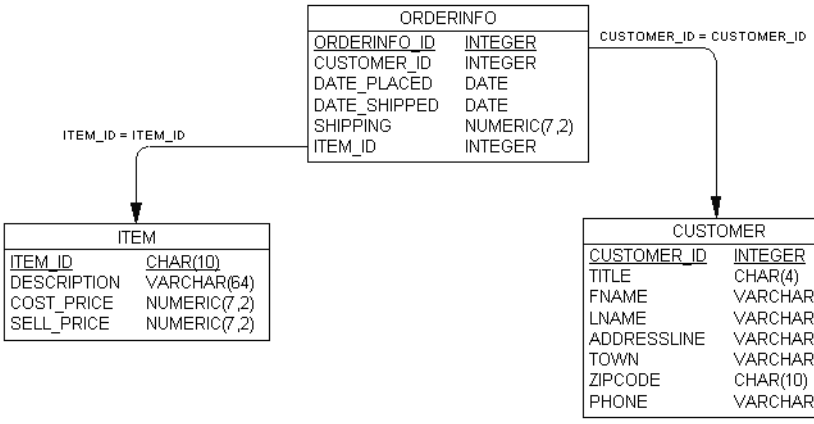


圖 2-15 在客戶和訂單專案上的一個嘗試

與 customer 和 orderinfo 一樣，我們將這些資訊拆分到兩個表，然後將它們關聯起來。然而，在這裡出現了一個小問題。

如果你仔細考慮訂單和訂單專案的關係，你可能發現不僅僅 orderinf 表的每條記錄關聯到很多條目，而且在不同的客戶訂了相同條目的濕乎乎，每個條目可能出現在很多訂單裡頭。

我們將在第十二章考慮這個問題，但是現在，你需要知道的就是我們有一個標準的解決方案用以解決這類難題。你可以在這兩個表之間建立第三個表，用以實現多對多關係。這很容易實現但不容易講明白，所以儘管先建立表 orderline 用來連接訂單表和表中的專案，就像圖 2-16 所示。

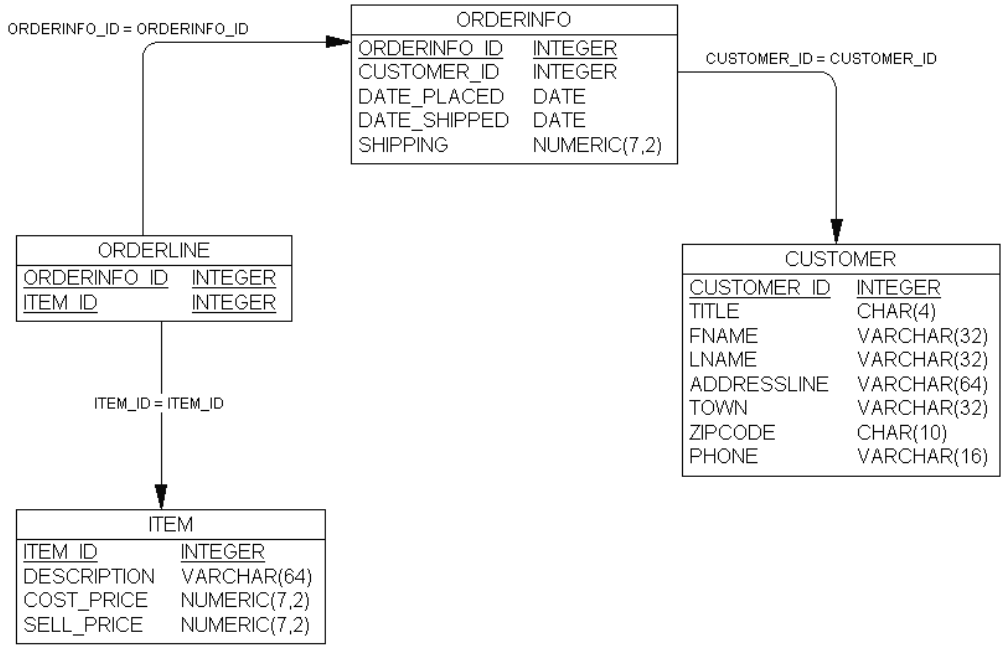


圖 2-16 關聯客戶和訂單

我們已經創建了一個每條記錄都關聯到一條訂單記錄的表。其每一行我們都可以通過 orderinfo_id 列確定它的來源並且通過 item_id 列確定它引用的條目。每個單獨的專案都可以出現在多個訂單行裡頭，且每個單獨的訂單都可以包含很多訂單行。每條訂單行只引用一個單獨的條目，且每條訂單行只會出現在一個訂單中。

你還會發現我們不需要為標記每行而添加唯一的 id 列。這是因為組合的 orderinfo_id 和 item_id 總是唯一的。但是這裡還有一個潛在的微妙問題。如果客戶在一個訂單裡頭訂了一個條目兩次將會發生什麼？我們無法在 orderline 表裡頭輸入另一行因為我們剛剛才說組合的 orderinfo_id 和 item_id 總是唯一的。我們是否需要添加另一個特殊的表來迎合包含不止一個相同專案的訂單？幸運的是我們不需要那麼做。有一個更簡單的方法。我們只需要在 orderline 表裡頭添加數量列，這就令人滿意了（查看下一節的圖 2-17）。

完成初始化設計

我們在完成第一版的資料庫主題結構設計完成前還需要存儲兩塊資料。我們需要存儲每個產品的條碼，以及我們倉庫中每個項目的存量。

很可能每項產品有不止一個條碼，因為當製造商明顯地改變產品外包裝的時候，他們通常會修改條碼。例如你可能看到打包的“贈送 20%”（通常是指銷售上的捆綁贈送包）。生產商通常會為這類推銷包改變條碼，但實際上產品沒有改變。因此，我們可能有很多條碼到一個項目的關係。我們增加了一個表存儲條碼，如圖 2-17 所示。

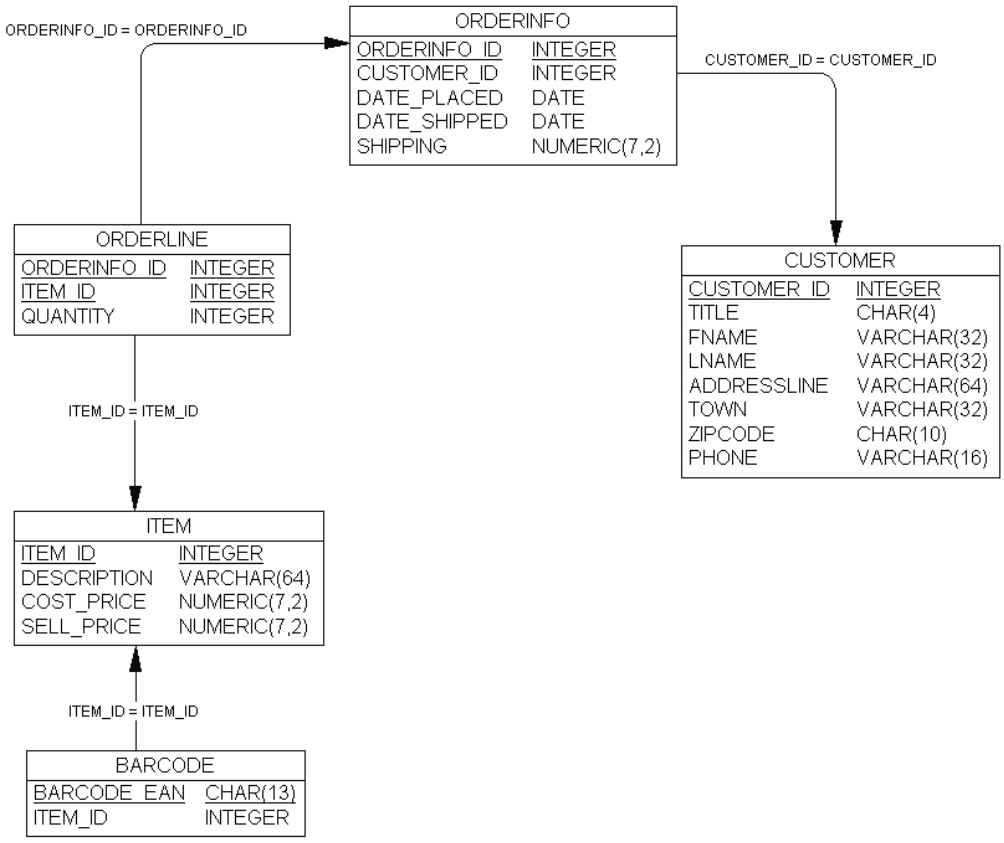


圖 2-17 增加條碼這個關係

注意從 barcode 表到 item 表的箭頭的方向，因為可能每個條目有多個條碼。同時還要注意 barcode_ean 列是主鍵，因為每個條碼都需要唯一的一行，並且每個獨立的條目都可以有多個條碼，但沒有條碼可以同時屬於多個專案（EAN 是歐洲的產品條碼標準）。

我們最後需要添加的是我們倉庫裡頭每項條目的存量。如果大多數條目是在倉庫中且倉庫資訊都很基本，我們可以簡單地在 item 表裡存儲庫存量。但是，如果我們提供很多產品單只有少量庫存，且我們需要存儲很多關於倉庫中專案的資訊，這將行不通了。例如在倉庫運營中，我們需要存儲產地資訊，批次號已經過期日期。如果我們的專案檔案裡頭有 500,000 條記錄，但倉庫中只有前 1,000 條，這將非常浪費。這也有一個標準的解法，就是使用叫輔助表的表格。我們將通過這種方式存儲倉儲資訊到我們的示例資料庫中，如圖 2-18 所示。

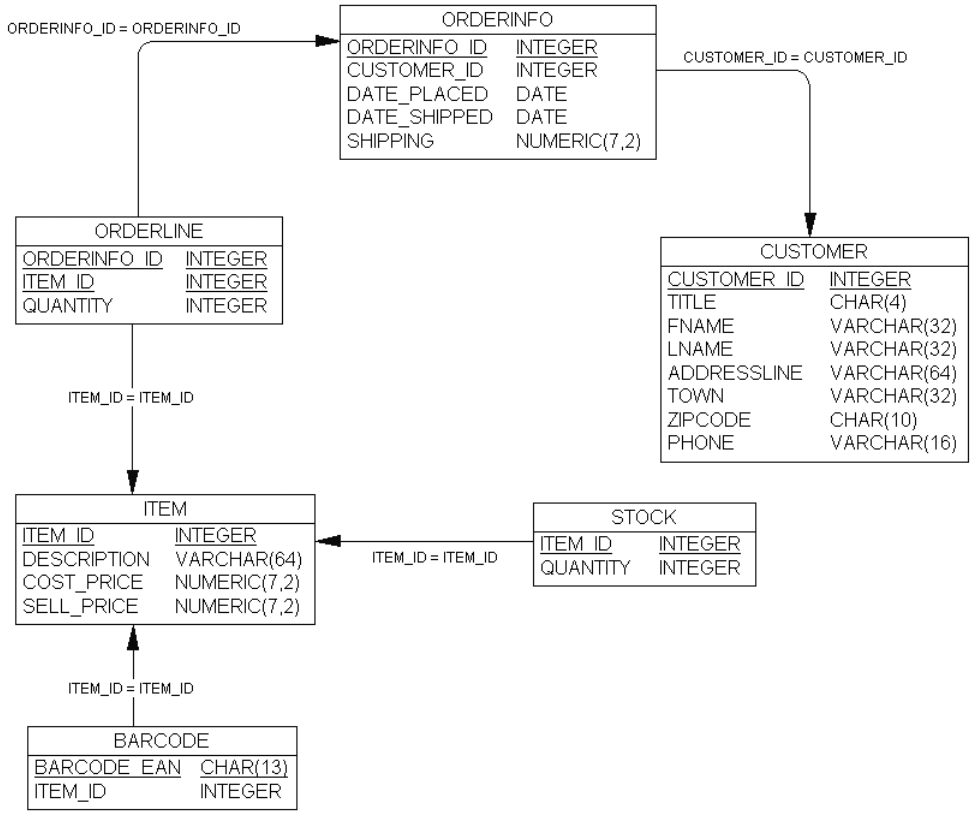


圖 2-19 bpsimple 資料庫的設計

我們建立了一個新表存儲供給資訊（本例中為庫存），然後建立在倉庫中的項目只需要的行，用來連接這些資訊到主表中。注意 stock 表使用 item_id 作為唯一鍵，它存儲直接關聯到專案的資訊，使用 item_id 關聯到 item 表相關的行。箭頭指向 item 表，因為它是主表，即使在本例中它不是一個多對一的關係。對於其他的表，底線表明了主鍵（保證唯一的資訊）。

按照現在的情況，我們的設計很明顯過度複雜了，因為我們需要保存的擴展資訊太少了。我們只是想演示模式設計的方法來看看它怎麼做到我們的需求的，在本書後面的章節，我們將演示怎麼訪問像這裡一樣存在補充表格情況下的資料。對於那些喜歡偷偷往前翻的傢夥，我們可以告訴你我們將使用的叫做外連接（outer join）。

注：在第八章，我們將看到我們怎麼確保資料庫中的表與表之間的這些關係被執行，在第十二章我們將再關注資料庫設計的進一步細化。當我們到達第八章，我們將發現一些更高級的技術用於更好地管理資料庫的一致性，而且我們將把這些增強設計加入到 bpfinal 模式中。

基底資料型別

在我們的示例資料庫中，我們用到了一些基本的通用的資料類型，總結在表 2-1 中。當我們在下一章建表的時候這些將被轉化成實際的 PostgreSQL 類型。

表 2-1 示例資料庫中的資料類型

資料類型	描述
integer	一個整數。
serial	一個整數，但是在新增行的時候會自動設置為一個唯一數字。這種類型我們通常會用在 id 列。本章的圖表顯示這種欄位為整數是因為後臺的資料庫實際上是用的這種類型。
char	定長的字元陣列，陣列的長度為類型後面的括弧中的數字。對於這種列類型，PostgreSQL 總是存儲規定長度的字元數。如果我們使用 char(256)存儲僅僅一個字元，資料庫將存儲 256（至少，可能有其他輔助資料）個位元組的資料，而且在檢索資料的時候會返回這麼多資料。
varchar	這也是一個字元陣列，但就像它的名字暗示的一樣，它是變長的。通常，在資料庫中佔用的空間和實際上需要存儲的資料的長度一樣。當你要求返回 varchar 類型的欄位，它僅僅返回你存儲長度的字元。能存儲的最大長度由類型後面括弧中的數字表示。

date	它允許你存儲年月日資訊。還有相關的類型允許我們像日期資訊一樣存儲時間資訊。我們將在後麵點的第八章碰到它。
numeric	它允許你存儲指定位元數的數位（類型後面括弧中的第一個數字）以及固定位元數的小數（類型後面括弧中的第二個數字）。在這裡，number(7,2)將存儲 7 位元數位，其中小數點後有兩位。

就像在本章早些時候標注的，由於添加一個特殊的唯一資料列在資料庫中非常常見，所以大多數資料庫都有一個內置的解決方案：一個叫做 serial（序號）類型資料類型。這種特殊的類型實際上是一個在有行插入表格時會自動增長的 integer 類型，它會在每行插入資料的時候賦予一個唯一的數字。當我們在有 serial 列的表裡頭加入一行資料時，我們不需要指定那列的值，而是讓資料庫自動賦予下一個值給它。大多數資料庫在他們分配序列值的時候，從不考慮將要刪除的行。分配的新值只會隨著新行增加。我們將在第六章看到怎麼處理 serial 類型資料失序的問題。

在第八章，我們將碰到 PostgreSQL 的其他資料類型，這些將讓我們有一個機會重新審視現這些資料類型的選擇。附錄 B 提供 PostgreSQL 資料類型的概要說明。

處理未知的值：空值（NULL）

在我們示例資料庫設計中的 orderinfo 表中，我們有一個下訂單日期和一個發貨日期，都是日期類型。在我們已經收到訂單但還未發貨的時候，我們該怎麼辦？我們該怎麼存儲發貨日期？我們可以存儲一個特殊的日期，一個守護值（sentinel value），用來告訴我們我們還未針對訂單發貨。在 UNIX 類型的系統，我們可以使用 1970 年 1 月 1 日，這是一個傳統的 UNIX 系統開始計數的日期。那個日期絕對比任何我們想要存儲到資料庫的訂單日期都早，所以我們總是可以知道那個日期意味著還未發貨。

但是，在表中稀稀拉拉地使用特殊值值顯示出糟糕的設計，有時候甚至是錯誤的。例如，一個新手參與這個專案並且不知道有一些特殊日期，他可能嘗試計算下訂單和發貨間隔的平均耗時，如果這裡有一些發貨日期比下訂單的日期更早，他可能得到一些非常奇怪的值。

幸運的是，所有的關聯式資料庫都提供一個叫空值（NULL）的特殊值，用來表示當前是未知的。注意它並不表示零，或者空字串，或者任何可以由欄位的資料類型表現的東西。未知的值和零或者空字串完全不同。實際上，空值甚至根本不是一個值。

空值的概念通常困擾很多資料庫新使用者（羅馬人也對不存在的東西很困惱，所以羅馬數字裡頭沒有零）。在資料庫術語中，空值通常指值還不知道，但它也有一些在這個意思之上的其他的甚至有一些微妙變化的意思。

要非常小心對待空值，因為它們會偶爾會出現在你面前並令你奇怪，通常是令人討厭的。所以，在我們的 orderinfo 表中，我們可以在未發貨前設置送貨日期為空值，這意味著“現在還未知道”，這正是我們所需要的。有另一個關於 NULL 微妙的用法（不是很常用），它意味著“跟這行無關的”。假設你對別人做一個調查，問題之一是關於眼鏡的顏色。對於不戴眼鏡的人，這明顯是一個無意義的問題。這就是空值會被用在記錄這個列對於這一行無關的情況。

關於空值的一個特徵是如果你比較兩個空值，結果通常是未知的。這有時候會把人弄糊塗，但是如果你想清楚空值的意思就是不知道，那麼對於測試兩個未知的東西是否相等的結果也是未知就非常符合邏輯了。SQL 有特殊的方法，通過 IS NULL 檢查是否空值。這將允許你在你想要的時候查找以及測試空值。IS NULL 將在第四章討論。

空值的表現和常規的值的有些細微的不同。所以，有必要在你設計表的時候指出一些列不能存儲空值。通常在你確保一個列永遠不能接受空值的時候指定它為 NOT NULL 是個好主意，例如主鍵列。一些資料庫設計者甚至提倡基本上完全禁止空值，但空值確實是有用的，所以我們通常提倡在一些特意選擇的列中允許空值，這些列確實有一些真正的可能性需要未知的值。NOT NULL 將在第八章詳細討論。

回顧示例資料庫

在本章，我們已經完成設計一個可能可以用於小商店的叫做 bpsimple 的專用的簡單資料庫，用來管理客戶、訂單以及訂單項（見本章稍早點的圖 2-18）。隨著本書的前進，我們將使用這個資料庫演示 SQL 和其他的 PostgreSQL 的功能。我們還將發覺我們現有設計的不足，並判斷它在哪些方面可以得到改進。

這個我們在使用的簡化的資料庫擁有很多現實發佈的資料庫看上去擁有的很多元素；但是它卻是在很多方面被簡化了。例如，一個條目應該有一個完整的描述用於倉庫存檔，一個簡單的描述用於銷售時顯示，以及另一個描述顯示在貨架邊緣的標籤上。我們存儲的客戶位址資訊也非常簡單。我們無法處理過長的包含省市甚至街道的位址。我們也無法處理海外訂單。

通常在初始設計中做到合理可靠的基礎並擴展比嘗試迎合所有可能的需求更可行。這個資料庫足夠符合我們的初始需求。

下一章，我們將研究安裝 PostgreSQL，建立我們示例中的表並填充一些示例資料。

摘要

在本章中，我們仔細分析了一個簡單的資料庫表與單個的試算表的相似之處，以及四個重要的區別：

- 一個列中的所有項目的類型都必須相同。
- 表中所有行的列數都必須相同。
- 必須想辦法唯一標記每一行。
- 在資料庫表不包含行的順序，但試算表中包含。

我們看到我們怎麼擴展我們的資料庫到多個表，這讓我們能簡單管理我們多對一的關係。我們給出一些非正式的 rules of thumb 來說明你理解怎樣將資料庫設計構建起來。我們將在後面的章節以更嚴謹的方式回到資料庫設計的主題。

我們還看到了怎樣通過添加一個額外的表，將它拆分成兩個一對多關聯性，實現現實世界中出現的多對多關係。

最後，我們擴展了我們的初始資料庫設計，因此我們有了一個我們可以用於伴隨本書向前的示例資料庫，或者叫做模式。

下一章，我們將瞭解怎樣使 PostgreSQL 在不同的平臺中運行起來。

第三章初步使用 PostgreSQL

本章將討論在各種作業系統中安裝和配置 PostgreSQL。如果你需要在一個 Linux 系統中安裝它，預編譯的二進位包比較容易安裝。如果你是在運行一個 UNIX 系統或類 UNIX 系統，例如 Linux，FreeBSD，AIX，Solaris，HP-UX 或者 Mac OS X，從源碼編譯 PostgreSQL 也不難。

我們也將講到怎樣在 Windows 平臺上使用 PostgreSQL 8.0 裡頭介紹的 Windows 安裝程式安裝 PostgreSQL。早期的版本可以安裝在 Windows 中，但是這需要其他一些軟體來建立類 UNIX 環境。我們強烈建議在 Windows 系統中使用 8.0 版本或更新的版本。

最後，我們將在以下的章節準備第 2 章討論的示例。

本章將特別討論以下主題：

- 在 Linux 上通過二進位檔案安裝 PostgreSQL
- 在 Linux 上通過源碼安裝 PostgreSQL
- 在 Linux 和 UNIX 系統上配置 PostgreSQL
- 在 Windows 上安裝和配置 PostgreSQL
- 建立資料庫，建立表，增加資料

在 Linux 和 Unix 系統中安裝 PostgreSQL

如果你安裝較新發行版本的 Linux 系統，你可能已經安裝 PostgreSQL 或者在系統安裝盤會帶有它的安裝包。如果沒有，你可以在很多 Linux 發行版本中通過 RPM 包安裝 PostgreSQL。另外，你可以在類 UNIX 系統中通過源碼安裝 PostgreSQL。

在 Linux 中使用二進位檔案安裝 PostgreSQL

可能在 Linux 中安裝 PostgreSQL 最容易的方法就是通過二進位包安裝。PostgreSQL 的二進位檔案對於很多 Linux 發行版本來說都可以安裝 RPM （RPM 包管理器，以前的 Red Hat Package Manager）的安裝包。在寫本書的時候，在 http://www.postgresql.org/有給以下平臺使用的 RPM 安裝包：

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

你可以通過 http://www.rpmfind.net 尋找其他 Linux 發行版本的安裝包，包括以下發行版本：

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

注：Debian Linux 用戶可以通過 apt-get 安裝 PostgreSQL。

表 3-1 列出了 PostgreSQL 二進位安裝包。為了安裝資料庫和用戶端，你需要下載和安裝至少 base，libs 和 server 包。

表 3-1 PostgreSQL 二進位包

包	描述
postgresql	包含用戶端和工具的基礎包
postgresql-libs	用戶端需要的共用庫
postgresql-server	建立和運行服務端的程式
postgresql-contrib	貢獻的擴展程式
postgresql-devel	開發用的標頭檔和庫
postgresql-docs	文檔
postgresql-jdbc	PostgreSQL 的資料庫連接庫
postgresql-odbc	PostgreSQL 的 ODBC 介面庫
postgresql-pl	Perl 的 PostgreSQL 伺服器支援
postgresql-python	Pyton 的 PostgreSQL 伺服器支援
postgresql-tcl	Tcl 的 PostgreSQL 伺服器支援
postgresql-test	PostgreSQL 測試套件

實際的檔案名會包含版本號。需要安裝相同的版本的庫，甚至是版本修訂號。在版本號 8.x.y 的包中，x.y 就是修訂號。

安裝 RPM 包

安裝 RPM 包，你可以使用以下方法：

- 使用 RPM 包管理程式。確保你以管理員（root）登錄來執行安裝。
- 使用你選擇的圖形界麵包管理器，例如 KPackage，來安裝 RPM 包。
- 將所有的 RPM 檔放在同一個目錄，並以管理員（root）身份執行以下命令來解壓並安裝所有的檔到恰當的位置：

```
$ rpm -U *.rpm
```

你也可以安裝和你的發行版本綁定在一起的 PostgreSQL 包，例如 Red Hat 或 SuSE Linux。例如，在 Suse Linux 9.x 中，你可以通過 Yast2 安裝工具選擇表 3-1 中的某個版本的 PostgreSQL 的安裝包，如下圖所示：

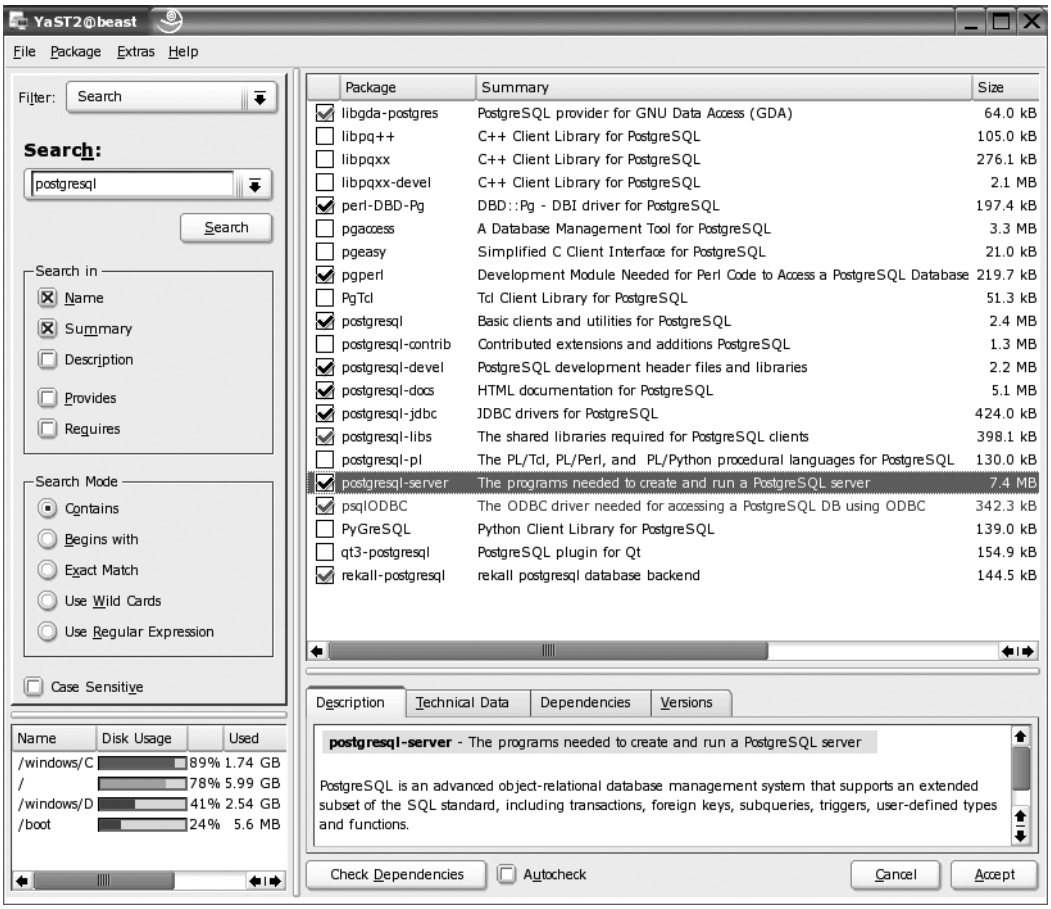


圖 3-1，通過 Yast2 從 SuSE 的安裝包安裝 PostgreSQL

升級到新版的 PostgreSQL

PostgreSQL 在不停的開發中，所以不停有新版產生。通過 RPM 包安裝的很容易升級到最新的版本。完成省，僅僅需要使用-U 參數而不是-i 參數告訴 rpm 你在執行一次升級而不是首次安裝：

```
$  rpm -U *.rpm
```

但是，在執行升級前，你需要備份資料庫中的現有資料。PostgreSQL 官方網站的發行注記中會有一些在升級到最新版本前必須做的預防措施需要完成。備份現有資料庫的細節將在本書 11 章討論。

注：如果你是在安裝一個新版的 PostgreSQL 到你現存的安裝上，確保在執行升級前已經閱讀新版的發行注記。在某些情況下，在升級後，必須備份和歡迎你的 PostgreSQL 資料庫，例如新版本引用了新的資料存儲方式。

PostgreSQL 的安裝解析

PostgreSQL 的安裝包含大量的應用程式、工具和資料目錄。PostgreSQL 的主應用程式（postmaster）包含為用戶端訪問資料提供服務的服務代碼。工具例如 pg_ctl 用來控制伺服器啟動後就必須一致運行的主服務進程。

PostgreSQL 使用一個目錄存放資料庫所有的檔。這個目錄不僅存放表和記錄，還存放系統參數。一個典型的安裝將包含表 3-2 中列出的 PostgreSQL 安裝的所有元件，存放在 PostgreSQL 目錄下的子目錄中。通常情況下在/usr/local/pgsql（也是使用源碼安裝的默認位置，下節將討論）。

表 3-2 PostgreSQL 的安裝解析

目錄	描述
bin	應用程式和工具，例如 pg_ctl 和 postmaster
data	資料庫本書，通過 initdb 初始化
doc	HTML 格式的文檔
include	用於開發 PostgreSQL 應用程式的標頭檔
lib	用於開發 PostgreSQL 應用程式的庫檔
man	PostgreSQL 的手冊
share	設定檔示例

單目錄的方式有個缺點：固定的程式和經常變動的資料存儲在同一個地方，通常不理想。

PostgreSQL 使用的檔分為兩大類：

- 在資料庫伺服器運行的時候需要寫入的檔，包括資料檔案和日誌。資料檔案時系統的心臟，存儲你的資料庫的所有的資訊。資料庫伺服器產生的日誌檔包含關於資料庫訪問的資訊，可以在解決故障的時候有很大的作用。在日誌記錄大開後，它將一直增長。
- 在資料庫伺服器運行的時候不需要寫入的檔，實際上是唯讀檔案。這些檔包含 PostgreSQL 的應用程式就像 postmaster 和 pg_ctl，這些東西一旦安裝完成就不再改變（升級除外）。

為了更有效和簡單的管理安裝，你也許希望分開不同類別的檔。PostgreSQL 提供靈活的方法分開存儲應用程式、日誌和資料，一些 Linux 發行版本已經使這種靈活性生效了。例如，在 SuSE Linux 9.x 中，PostgreSQL 應用程式和其他程式一樣存儲在/usr/bin，日誌檔存放在/var/log/postgresql，資料檔案存放在/var/lib/pgsql/data。這意味著非常容易安排備份從非關鍵檔區別出來的關鍵資料，例如日誌檔。

其他發行版本都有自己的文件分佈規劃。你可以使用 rpm 命令來列出某個包安裝的檔。為了做到這個，使用以下的查詢選項：

```
$ 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
```

為了查看所有安裝的檔，你需要針對所有的用以安裝完整的 PostgreSQL 的包運行 rpm 命令。不同的發行版本的包名字可能有細微區別。例如，SuSE Linux 使用名為 pg_serv 作為伺服器程式的包名稱，所以查詢選項應該像這樣：

```
$ rpm -q -l pg_serv
/etc/init.d/postgresql
/etc/logrotate.d/postgresql
...
/var/lib/pgsql/data/pg_options
```

當然，你也可以選擇一個圖形介面的包管理工具，例如 KDE 桌面環境提供的 KPackage。圖 3-2 展示通過 KPackage 查看包內容的一個例子。

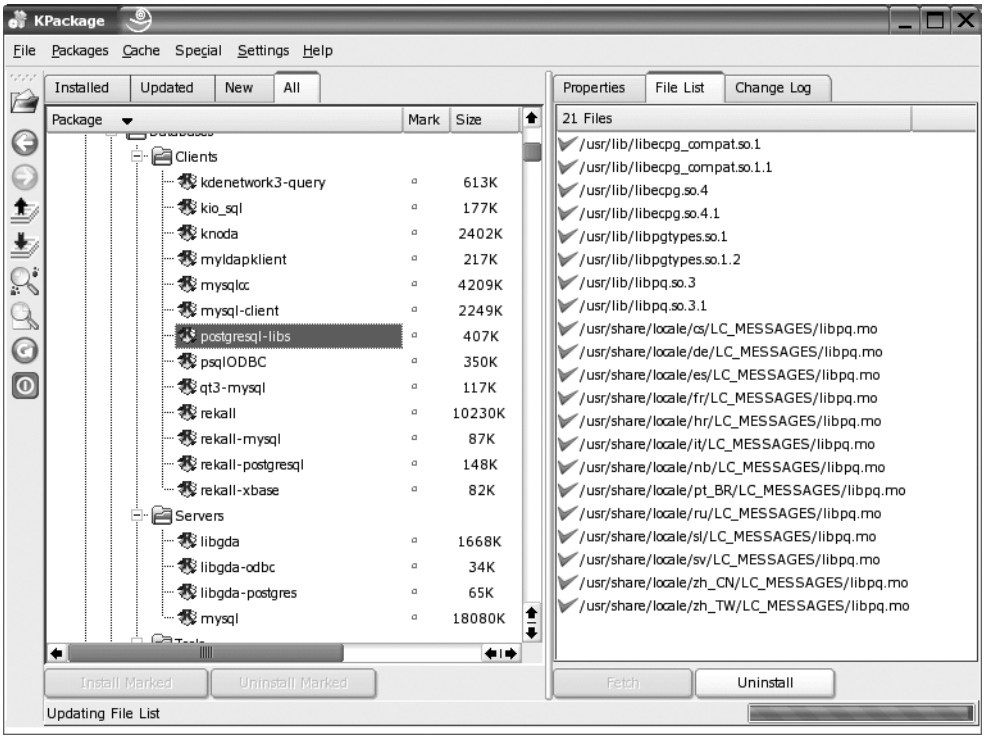


圖 3-2 使用 KPackage 查看包內容的示例

使用 Linxu 發行版本安裝的缺點是總是不是很清楚安裝的東西在哪裡。所以，如果你想升級到最新的版本，很難處理確保你已經清理掉了你原來執行的安裝。一個解決方案是從源碼安裝 PostgreSQL，就像下一節討論的一樣。加入你沒有從源碼安裝的打算，你可以跳過下一節，直接跳到討論 PostgreSQL 設置的“在 Linux 和 Unix 系統中設置 PostgreSQL”小節。

通過源碼安裝 PostgreSQL

就像前一章講述的，在很多 Linux 發行版本中你可以使用 RPM 包安裝 PostgreSQL。另外，你可以在任何 UNIX 相容的系統中從源碼創建和安裝 PostgreSQL，包括 Mac OS X。

PostgreSQL 的源碼可以在 http://www.postgresql.org 找到。在這裡，你將找到最新的源碼以及下一個版本的 beta 測試版的源碼。除非你很喜歡嘗鮮，否則建議你使用最新的穩定版本。

你可以找到整個打包了的 PostgreSQL 的源碼，可能是 gzip 壓縮的 tar 打包檔，就像 postgresql-9.0.0.tar.gz 或者 bzip2 壓縮的 tar 打包檔，就像 postgresql-9.0.0.tar.bz2。在寫本文的時候，PostgreSQL 打包檔已經超過 13MB 了。

實際的檔案名依賴於當前版本的修訂號。

編譯 PostgreSQL 相當簡單。如果你熟悉編譯開源產品，對你來說這裡不會有任何意外。即使你是第一次編譯和安裝開源產品，你也不會覺得困難。

為了執行源碼編譯，你需要一個帶有完整開發環境的 Linux 或 UNIX 系統。這包括 C 編譯器和 GNU 的 make 工具（對於編譯本資料庫系統是必須的）。Linux 發行版本通常包含恰當的自由軟體基金提供的 GNU 工具。這包含優秀的 GNU C 編譯器（gcc），這也是 Linux 的標準編譯器。絕大部分 UNIX 平臺也都有 GNU 工具，我們推薦使用它們來編譯 PostgreSQL。你可以從 http://www.gnu.org 下載最新的工具。一旦你完成開發工具的安裝，編譯 PostgreSQL 就輕而易舉了。

解壓源碼

使用普通用戶開始安裝。拷貝打包的源碼到一個合適的目錄用於編譯。這個目錄不需要（實際上，不應該）在最終 PostgreSQL 安裝的位置。一個可能的位置是你 home 目錄中的義工字幕了。因為你不需要超級使用者許可權用來編譯 PostgreSQL；超級用戶許可權僅僅在建立完成後安裝的時候需要用一下。我們通常解壓源碼到專門用來管理產品源碼的目錄/usr/src，但你可以解壓源碼到任何有足夠空間用於編譯的地方。大概需要 90M 多一點空間存放解壓後的源碼。

解壓打包的源碼的命令如下：

```
$ tar zxf postgresql-9.0.1.tar.gz
```

解壓的過程將建立新的目錄，目錄名依賴於你編譯的 PostgreSQL 版本。進入目錄：

```
$ cd postgresql-8.0.0
```

提示：在這個目錄中你可以找到一個叫 INSTALL 的檔，這個檔包含了編譯相關指令的詳細手冊，有些少見的情況自動化的過程會失敗。

配置編譯

建立過程使用一個叫 configure 的配置腳本針對你的特別平臺來裁剪的建立參數。如果想接受所有默認參數，你可以簡單的不帶任何參數運行 configure。以下為在一個 Linux 系統新運行 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 腳本設置控制 PostgreSQL 生成方法的變數，統計編譯的平臺，C 編譯器提供的功能等。configure 腳本將自動設置安裝的位置。預設的 PostgreSQL 安裝位置位於/usr/local/pgsql，包含應用程式和資料的子目錄。

你可以使用 configure 的參數來改變預設位置，設置資料庫伺服器使用的網路埠以及附加的存儲過程使用的服務端程式語言。這些選項在表 3-3 中列出。

表 3-3 PostgreSQL 配置腳本選項

選項	描述
--prefix=*prefix*	安裝到 prefix 指向的目錄；默認為/usr/local/pgsql
--bindir=*dir*	安裝應用程式到 dir；默認為 prefix/bin
--with-docdir=*dir*	安裝文檔到 dir；默認為 prefix/doc
--with-pgport=*port*	設置預設的伺服器端網路連接服務 TCP 埠號
--with-tcl	為服務端提供 Tcl 存儲過程支援

--with-perl	為服務端提供 Perl 存儲過程支援
--with-python	為服務端提供 Python 存儲過程支援

你可以通過—help 參數查看 configure 所有的選項。

```
$ ./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.
...
$
```

你在這一步可以不設置資料庫和日誌檔的位置。你通常可以在安裝完成後啟動服務進程的時候指定這些位置。

構建軟體

一旦編譯完成，你可以通過 make 命令構建軟體。PostgreSQL 構建過程使用一套精密的 makefile 來控制編譯過程。為了完成這個過程，我們建議你使用 GNU 版本的 make 工具。這是 Linux 默認的 make 工具。在其他的 UNIX 平臺，你可能需要另外安裝 GNU make。通常這會用 gmake 這個名字，來區別於系統自帶的 make。在這裡，我們說的 make 就是指 GNU make。

```
$ make
...
All of PostgreSQL successfully made. Ready to install.
```

如果一切正常，我們將看到大量的編譯過程。我們將最終得到一切都成功完成的好消息。

```
當 make 完成後，你需要將程式複製到目標位置。你可以使用 make 完成，但你需要先切換到超級使用者：
$ su
# make install
...
PostgreSQL installation complete.
# exit
$
```

一旦軟體構建完成並安裝好，你可以通過 pg_config 命令獲得一份 PostgreSQL 系統的配置：

```
pg_config --bindir | --includedir | --libdir | --configure | --version
```

pg_config 命令將回報 PostgreSQL 程式的安裝目錄(--bindir)，C 標頭檔的位置(--includedir)和庫檔的目錄(--libdir)以及 PostgreSQL 的版本（--version）：

```
$ pg_config --version
PostgreSQL 9.0.1
$
```

編譯時的配置參數可以通過 pg_config –configure 查看。這將提供 PostgreSQL 服務程式配置用於編譯的 configure 腳本的命令列參數。

以上就是安裝 PostgreSQL 的內容。你現在已經有一套用於 PostgreSQL 資料庫伺服器的程式在你系統的適當位置了。

在這一點上，你和用安裝包安裝一樣都處在相同的情況了。現在，是時候配置你安裝的 PostgreSQL 了。

在 Linux 和 Unix 上配置 PostgreSQL

在你安裝好 PostgreSQL 後，無論是通過 RPM 包或者通過源碼編譯，你都需要執行一些步驟讓它運行起來。第一步，你需要建立一個叫 postgres 的使用者，然後你需要為資料庫建立一個目錄並初始化資料庫結構。然後，你可以通過啟動 postmaster 進程啟動 PostgreSQL 了。

建立 postgres 用戶

PostgreSQL 的資料庫主進程 postmaster 是一個特別的程式。它負責處理所有的使用者對所有資料庫的訪問。它必須允許使用者訪問自己的資料但在沒授權的情況下不允許訪問其他使用者的資料。為了實現這個功能，它需要能夠控制所有的資料檔案，因而普通用戶不允許直接訪問這些檔。postmaster 進程將通過檢查訪問資料的使用者的賦權情況控制對資料檔案的訪問。

一般來說，PostgreSQL 需要用一個非管理員使用者運行，也就是說可以是任何普通用戶；如果你在你的 home 目錄裡頭安裝了資料庫，這個使用者可以是你自己的用戶。但是，PostgreSQL 通常使用一個概念上的虛擬使用者來完成資料訪問。通常，這個一個叫做 postgres 的用戶被建立用來管理這些資料檔案，它不需要其他的存取權限。另外，postgres 虛擬用戶可以提供一些其他的安全措施，例如這個用戶無法登錄，所以別人無法非法訪問這些資料。postmaster 程式代表其他使用者用這個使用者去訪問資料庫檔。

因此，建立一個可運行的 PostgreSQL 系統的第一步就是建立 postgres 用戶。每個系統建立使用者的方法都不同。Linux 可以通過 root 用戶使用 useradd 命令添加：

```
# useradd postgres
```

其他的 UNIX 系統中尼可能需要建立 home 目錄，修改設定檔或者運行相關的工具。請參考你的系統的文檔來獲得相關管理工具的細節。

建立資料庫目錄

下一步，你必須通過 root 使用者建立一個目錄給 PostgreSQL 用來存放資料庫，並將目錄的所有者設置為 postgres：

```
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
```

在這裡我們使用預設的位置給資料庫。你可以選擇在其他地方存儲資料，就像我們前面在 “PostgreSQL 的安裝解析” 小節裡討論的一樣。

初始化資料庫

通過 initdb 工具初始化 PostgreSQL 資料庫，初始化時需要制定你檔案系統中想要存儲資料庫的位置。這將做很多事，包括建立 PostgreSQL 需要運行的資料結構以及初始化一個可工作的資料庫：template1。

你需要使用 postgres 使用者來運行 initdb 工具。為了做到這點，最可靠的方法是完成兩步，第一步是通過 su 命令切換到 root 用戶，然後切換成 postgres 用戶，就像以下所示（作為一個普通用戶，你可能無法用其他使用者身份運行程式，所以你必須先變成超級用戶）：

```
$ su
# su - postgres
pg$
```

現在你運行的程式是以 postgres 使用者運行的，並且你可以訪問 PostgreSQL 的資料檔案了。很明顯，我們現實了 postgres 用戶的 shell 的命令提示符 pg$。

警告：不要為了圖方便直接用 root 而不是 postgres 用戶完成以上過程。由於安全原因，用 root 身份運行服務進程可能非常危險。如果這個進程有問題，可能導致外部通過網路非法訪問你的系統。由於這個原因，postmaster 將拒絕通過 root 運行。

通過 initdb 命令初始化資料庫：

```
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，你將在 initdb 命令的-D 參數指向的位置擁有一個全新的空白資料庫。

配置連接許可權

預設情況下，PostgreSQL 不允許全面的遠端存取。為了賦權給遠端連接，你必須編輯設定檔 pg_hba.conf。這個檔存在於資料庫檔的區域（在本例中，位於/usr/local/pgsql/），它包含允許或者拒絕特定使用者連接到資料庫的許可權的配置記錄。預設情況下，本地使用者可以連接但遠端用戶不允許。檔案格式非常簡單，PostgreSQL 自帶的默認檔包含大量的有用的注釋用於協助添加記錄。你可以根據需要給單個的使用者、主機、電腦組或者單獨的資料庫賦權。

例如，希望允許用戶 neil 通過 IP 位址為 192.168.0.3 的主機連接到 bpsimple 資料庫，添加以下行到 pg_hba.conf 文件：

```
host    bpsimple    neil    192.168.0.3/32  md5
```

注意，在早於 8.0 版本的 PostgreSQL 中，pg_hba.conf 通過 IP 位址和子網路遮罩說明一個主機位址，所以之前的例子應該寫成這樣：

```
host    bpsimple    neil    192.168.0.3 255.255.255.255 md5
```

本例中，我們將添加一條記錄來運行局域網中（本例中，子網為 192.168.x.x）的任何電腦通過密碼認證訪問資料庫。（如果你需要不同的訪問策略，參考設定檔裡頭的注釋）我們添加一行到 pg_hba.conf 的末尾，就像這樣：

```
host    all all 192.168.0.0/16  md5
```

這意味著 IP 位址由 192.168 開頭的電腦可以訪問所有的資料庫。此外，加入我們信任網路中的所有使用者，我們可以通過指定使用 trust 標記不受限的存取方法作為訪問策略，就像這樣：

```
host    all all 192.168.0.0/16  trust
```

PostgreSQL 的 postmaster 服務進程讀取設定檔 postgresql.conf（也存在於資料目錄中）來設置一系列的運行選項，包括（如果沒有另外指定-D 選項或者配置 PGDATA 環境變數）資料庫資料檔案的位置。這個設定檔被很好的注釋了，如果你想修改任何設置，它都提供了嚮導。PostgreSQL 的文檔有一章講述了運行配置。

例如，我們可以設置 postgresql.conf 檔中的 listen_addresses 參數允許伺服器監聽網路連接，而不是通過-i 選項：

```
listen_addresses='*'
```

實際上，我們推薦通過 postgresql.conf 設置配置參數來控制 postmaster 進程的行為。

啟動 postmaster 進程

現在，你可以啟動服務進程了。再次提醒，你可以使用-D 選項告訴 postmaster 資料庫檔所在位置。如果你想允許網路上的使用者訪問你的資料，你可以使用-i 選項啟用遠端存取（如果你沒在 postgresql.conf 檔中啟用 listen_addresses 選項，就像前面所說的）：

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

這條命令啟動 postmaster，重定向進程輸出到一個檔（名叫 logfile，存放在 postgres 使用者的 home 目錄中），並且通過 shell 的 2>&1 合併標準輸出和標準錯誤輸出。你可以通過重定向輸出到其他檔來選擇不同的日誌位置。

PostgreSQL 提供的 pg_ctl 工具提供了一種簡單的方法啟動、停止和重啟（就是停止和啟動）postmaster 進程。如果 PostgreSQL 就像之前所屬的那樣完全由 postgresql.conf 檔配置，可以簡單的使用以下命令啟動、停止和重啟：

```
pg_ctl start
pg_ctl stop
pg_ctl restart
```

連接到資料庫

現在你可以通過嘗試連接到資料庫測試它是否正常工作了。psql 工具是用來與資料庫進行交互和進行簡單的管理工作例如建立使用者，建立資料庫以及建表。在本章後面我們將用它來建立和填充資料庫，在第 5 章將詳細講解它的功能。現在，你可以簡單地嘗試連接到一個資料庫。以下的回饋顯示你已經運行了 postmaster：

```
pg$ /usr/local/pgsql/bin/psql
psql: FATAL 1: Database "postgres" does not exist in the system catalog.
```

不要被上面顯示的致命錯誤嚇著。預設情況下，psql 連接到本機的資料庫並嘗試用啟動這個程式的使用者的名稱打開資料庫。因為我們在這裡沒有建立叫 postgres 的資料庫，所以連接失敗。這象徵著，postmaster 進程運行了並且能夠正常回應失敗的細節。

為了指定連接的資料庫，可以傳遞-d 參數給 psql。全新的 PostgreSQL 系統包含一些系統使用的資料庫作為你需要新建的資料庫的範本。其中有一個叫做 template1。如果你需要，你可以連接到資料庫這個資料庫用來完成管理功能。

為了檢查網路連接，你可以使用網路上其他機器安裝的 psql 作為用戶端，或者其他的 PostgreSQL 相容的程式。在 psql 中，你可以使用-h 選項指定主機（無論是名稱還是 IP 位址），並指定一個系統資料庫（如果你還沒建立一個真正的資料庫）。

```
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$
```

配置自動啟動

最後一步需要做的是安排 postmaster 服務進程在機器重啟的時候自動啟動。實際上你所有要做的就是確保 postmaster 在啟動後被運行了。再次，對於 Linux 和 UNIX 變種系統，這裡都有一些小標準可以遵從。請參考你的系統文檔的細節。

如果你是從 Linux 發行版本安裝的 PostgreSQL，啟動腳本應該已經通過 RPM 包安裝的時候完成配置了。在 SuSE Linux 中，PostgreSQL 在系統進入多使用者模式的時候通過/etc/rc.d/init.d 下麵的一個叫做 postgresql 的腳本啟動。

如果你想自己寫啟動腳本，最簡單的辦法是建立一個簡單的腳步使用你需要的參數啟動 postmaster，並且在一個自動啟動的腳本裡頭添加一個到你的腳本的調用就 OK，例如在/etc/rc.d 裡面的腳本。需要確保 postmaster 是由 postgres 用戶啟動的。以下是從源碼安裝的一個默認安裝的 PostgreSQL 的可以完成以上工作的示例腳本：

```
#!/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 -。

建立一個包含以上腳本的可執行腳本。給它命名為 MyPostgreSQL。使用 chmod 命令給它賦予執行許可權，就像以

下的情況：

```
# chmod a+rx MyPostgreSQL
```

然後，你需要安排腳本在伺服器啟動和關機的時候啟動和停止 PostgreSQL：

```
MyPostgreSQL start
MyPostgreSQL stop
```

對於使用 System V 類型的 init 腳本的系統(例如很多 Linux 發行版本)，你可以把腳本放在適當的位置。例如在 SuSE Linux 中，你應該把腳本放在/etc/rc.d/init.d/MyPostgreSQL，並且建立以下位置的軟連接到這個腳本來實現在伺服器進入和離開多使用者模式的時候啟動和停止 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
```

請參考你的系統文檔關於啟動腳本的部分的詳細資訊。

停止 PostgreSQL

PostgreSQL 服務進程有序關閉非常重要，這將允許它將任何未寫入資料庫的資料寫入資料庫並釋放它使用的共用記憶體資源。

為了安全地關閉資料庫，可以通過 postgres 使用者或者 root 用戶使用 pg_ctrl 工具這樣做：

```
# /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

如果有啟動腳本，你可以像下面這樣使用它們。

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

這些腳本能保證資料庫在機器關機或者重啟的時候能正常關閉。

相關資源

為了在使用 PostgreSQL 時方便一點，最好添加 PostgreSQL 應用程式路徑到執行程式的搜索路徑中，手冊檔也需要這麼做。在標準 UNIX shell 中，添加以下代碼到你的啟動腳本中（.profile or .bashrc）：

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

當前和最新測試版本的 PostgreSQL 的源碼可以在 http://www.postgresql.org 找到。更多的 PostgreSQL 資源資訊在本書附錄 G 中列出來了。

在 Windows 中安裝 PostgreSQL

在本小節開始前，先給 Windows 用戶一些好消息。雖然 PostgreSQL 是為類 UNIX 平臺開發的，它卻是可移植的。

現在已經可能在 Windows 上寫 PostgreSQL 用戶端程式，而且從 7.1 版開始，PostgreSQL 可以編譯安裝和作為一個 PostgreSQL 伺服器運行在 Windows NT 4，2000，XP 和 Server 2003 中。

從 PostgreSQL 8.0 開始，已經有了 Windows 本地版本了，為服務端和用戶端程式提供了 Windows 的安裝程式，這讓在 Windows 下的安裝非常輕鬆。在 8.0 版之前，Windows 使用者需要安裝一些軟體來在 Windows 上提供 UNIX 功能。

注：PostgreSQL 8.0 支援 Windows 2000，Windows XP 和 Windows Server 2003，由於 Windows 95，98 和 Me 不提供一些它需要的功能所以它無法在這些平臺運行。它可以在 Windows NT 上跑起來，但必須手工安裝，因為 PostgreSQL 的安裝程式無法在 Windows NT 上正常運行。

看起來像 Linux 一樣的開源的作業系統平臺是 PostgreSQL 一樣的開來源資料庫的自然歸宿。實際上，我們不推薦在桌上出版的 Windows 上運行生產用的資料庫，但在 Windows 上安裝有它的優點。例如在客戶程式所在的機器上擁有 PostgreSQL 的工具套件例如 psql 在測試新裝資料庫和查找連接故障的時候非常有用。雖然你不需要在 Windows 上運行服務程式。在開發機上運行資料庫伺服器可以避免開發人員需要共用伺服器實例而帶來的問題。

使用 Windows 安裝程式

EnterpriseDB 公司（公司網站：http://www.enterprisedb.com）提供了 Windows 版的 PostgreSQL 安裝程式。安裝程式的名字類似於 postgresql-9.0.1-1-windows.exe。安裝程式需要系統安裝了 2.0 或更高版本的 Windows 安裝器。Windows XP 以及以後的版本中已經包含了恰當的版本。如果有必要，Windows 安裝期可以從微軟的官方網站 http://www.microsoft.com 下載（搜索 “Windows Installer redistributable”）。

按兩下這個檔就可以啟動安裝程式，進入安裝嚮導。在選擇安裝語言和閱讀安裝注記後，你將看到選擇安裝目錄對話方塊，如圖 3-3 所示。

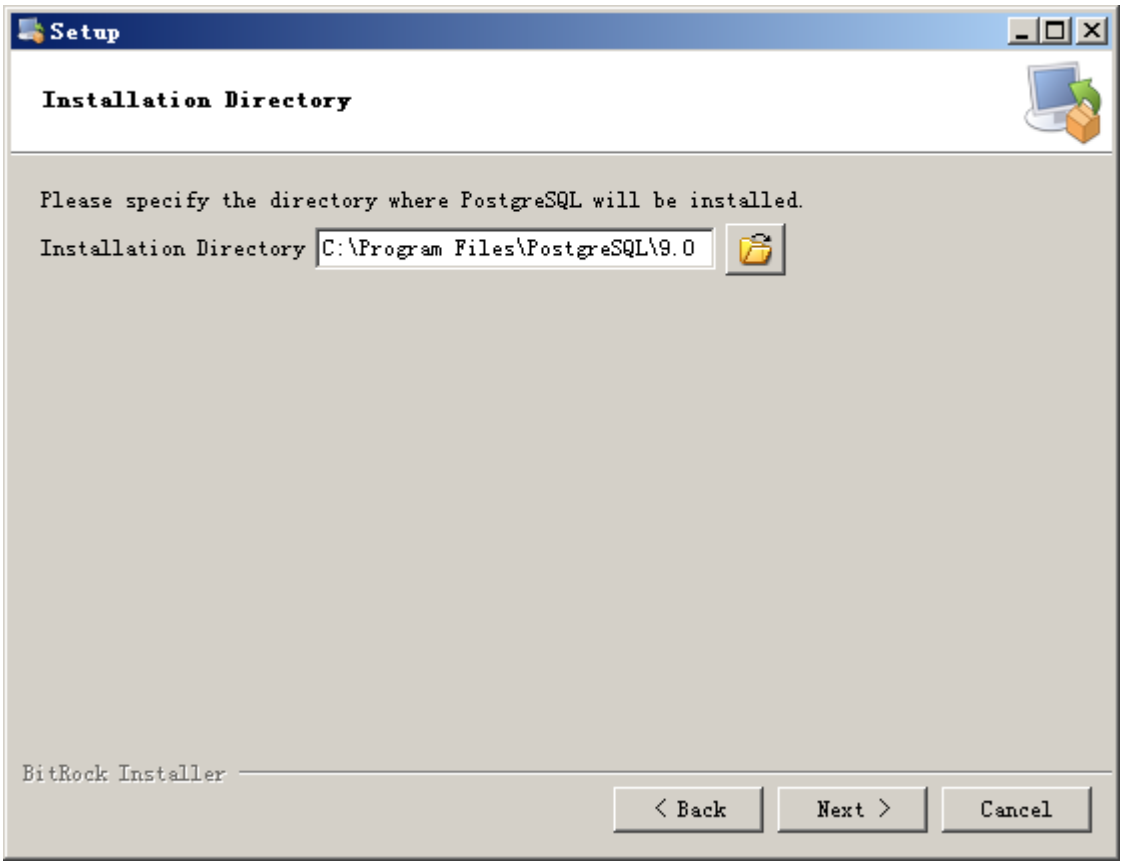


圖 3-3 PostgreSQL 安裝位置

在這一步，你可以選擇你的 PostgreSQL 應用程式安裝路徑。通過點擊打開資料夾按鈕，選擇安裝路徑。預設的安裝路徑為 C:\Program Files\PostgreSQL\9.0。

點擊下一步後，安裝嚮導進入資料目錄設置頁面。如圖 3-4 所示：

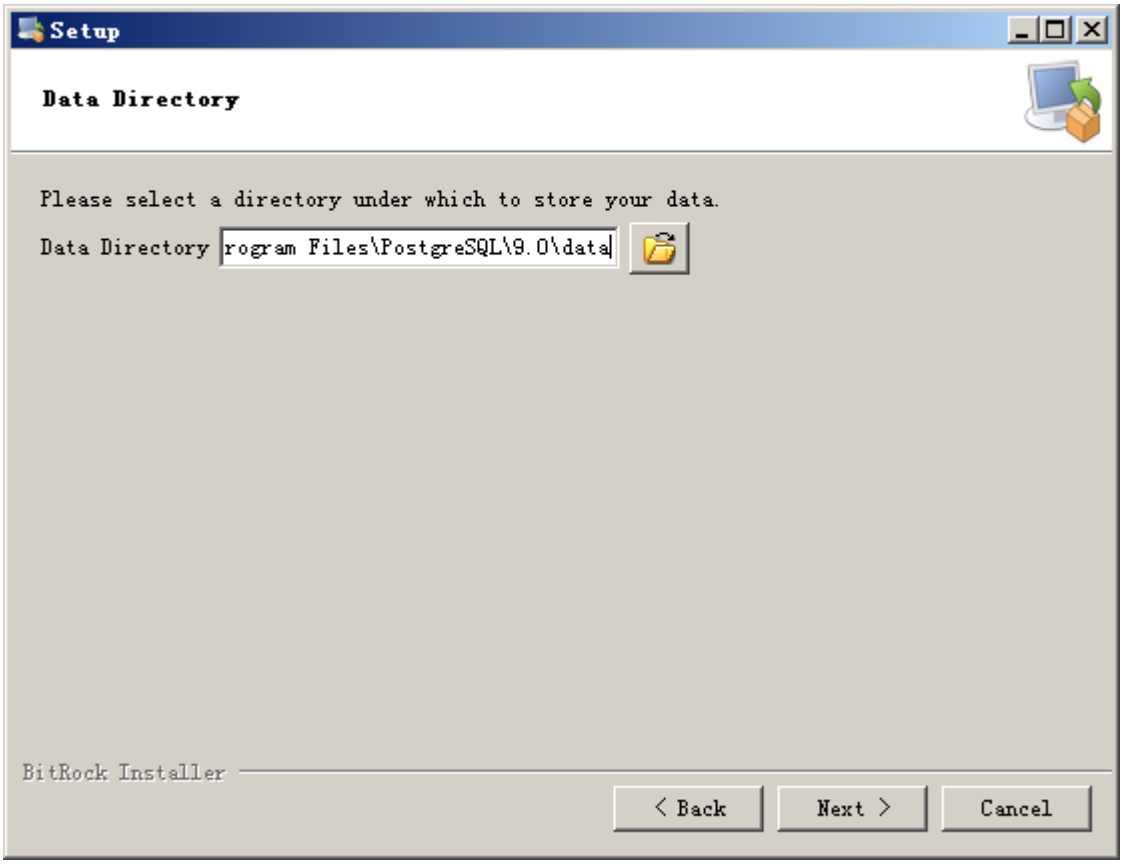


圖 3-4 PostgreSQL 資料目錄

這一步，你可以選擇你的 PostgreSQL 資料存放的路徑。通過點擊打開資料夾按鈕，選擇存放路徑。預設的存放路徑為 C:\Program Files\PostgreSQL\9.0\data。

點擊下一步，安裝程式進入填寫密碼頁面，如圖 3-5 所示：

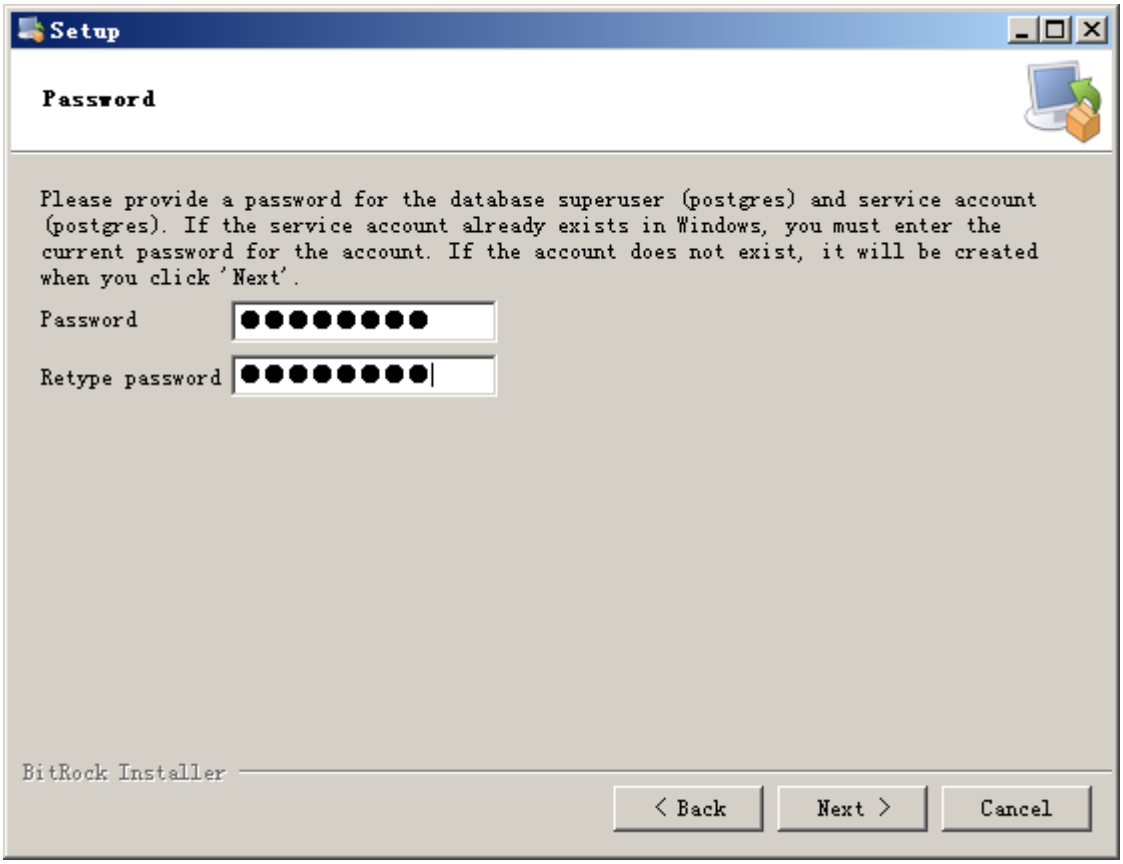


圖 3-5 輸入 postgres 使用者密碼

和 UNIX 系統一樣，PostgreSQL 不允許以管理員身份運行程式，這避免了擁有管理員許可權的使用者運行接受網路連接的服務程式潛在的安全問題。如果發現了 PostgreSQL 的安全若得並被使用溢出攻擊，這樣做的最壞情況只是 PostgreSQL 管理的資料被損壞，而不是整台伺服器被攻陷。安裝程式會使用一個叫 postgres 的使用者運行服務程式。在這一步，安裝嚮導將記錄 postgres 使用者的密碼。如果這台機器已經安裝過 PostgreSQL 或者已經存在 postgres 使用者，則輸入的密碼必須是這個使用者現在的密碼，否則，安裝嚮導將建立 postgres 使用者並設置密碼為輸入的密碼。輸入兩次相同的密碼以確保你記得你輸入的密碼。

點擊下一步，安裝嚮導要求你填入服務監聽的埠，如圖 3-6 所示：

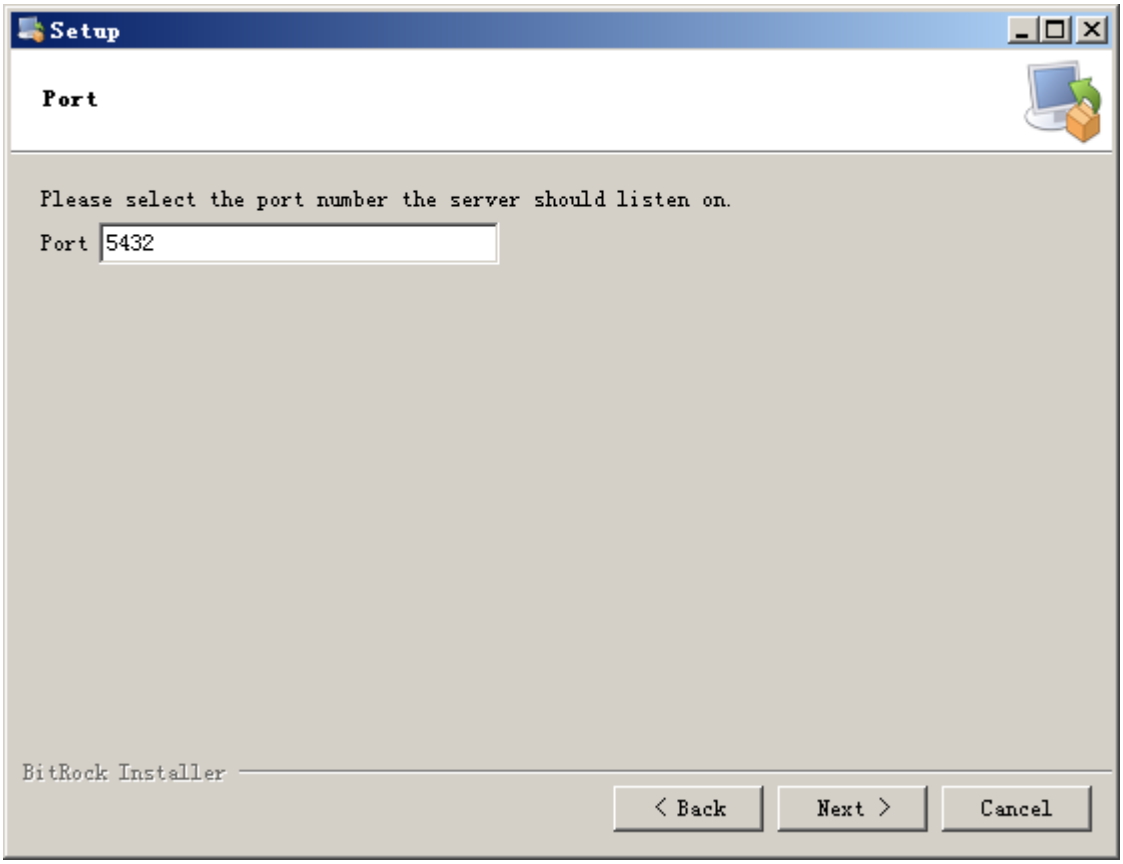


圖 3-6 輸入服務監聽埠

和 UNIX 下一樣，默認的埠為 5432，你可以設置為任何你想使用的埠。

下一步將設置新資料庫的區域，如圖 3-7 所示：

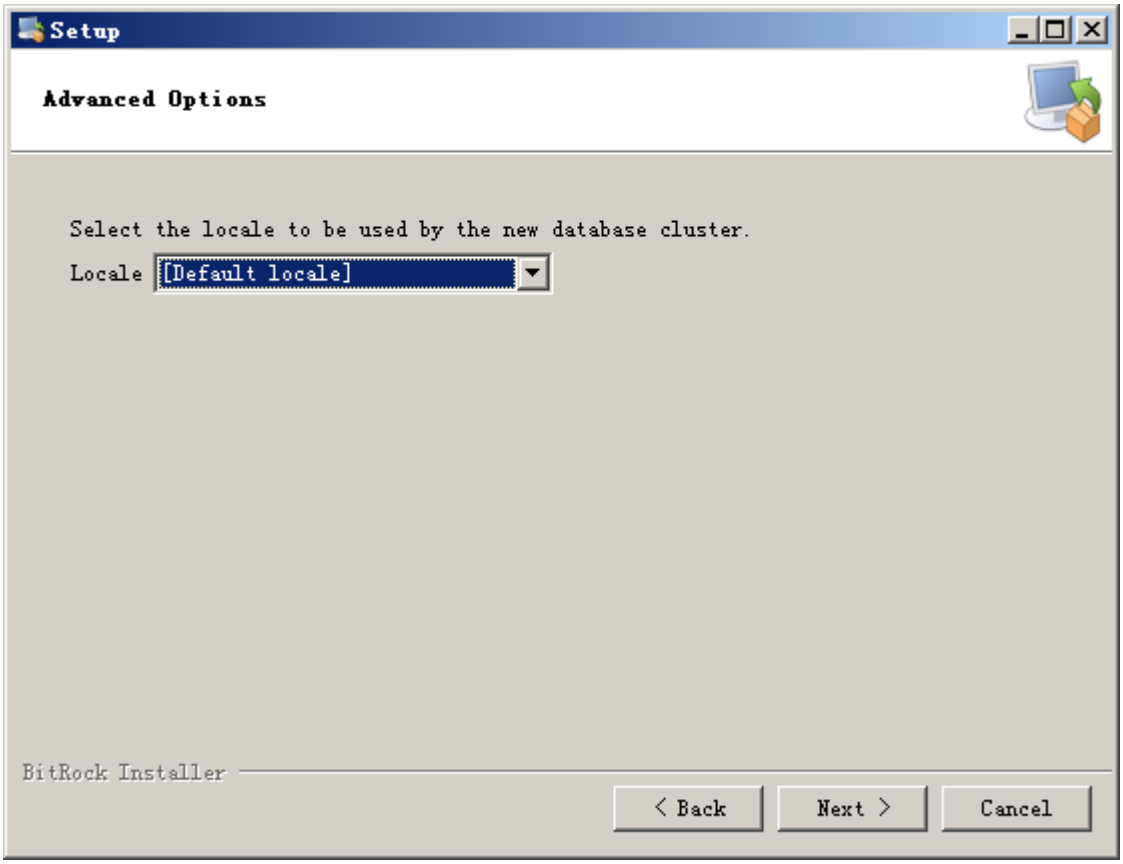


圖 3-7 選擇新建資料庫的區域

選擇恰當的區域，點擊下一步後，嚮導提示將開始安裝資料庫。再點擊下一步，如果不出錯，安裝過程將進行並正常完成。完成安裝後，資料庫服務進程應該已經運行。服務進程 postmaster.exe 和 postgres.exe 可以在工作管理員中看到，如圖 3-8 所示：

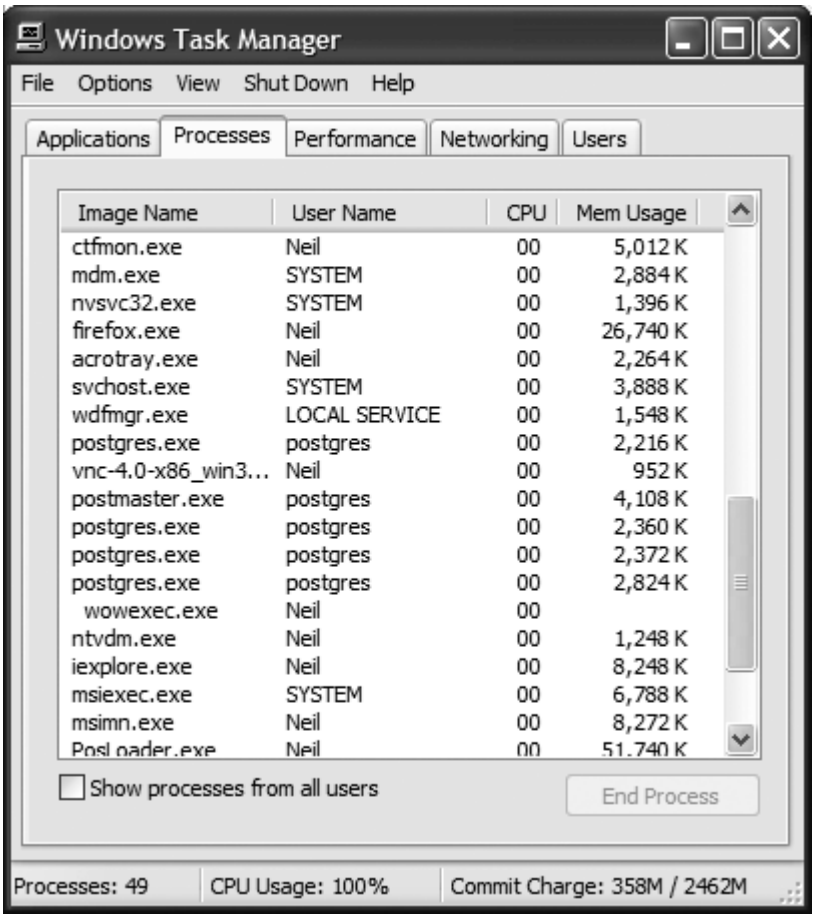


圖 3-8 PostgreSQL 進程

PostgreSQL 程式和工具被安裝到開始功能表，如圖 3-8 所示：

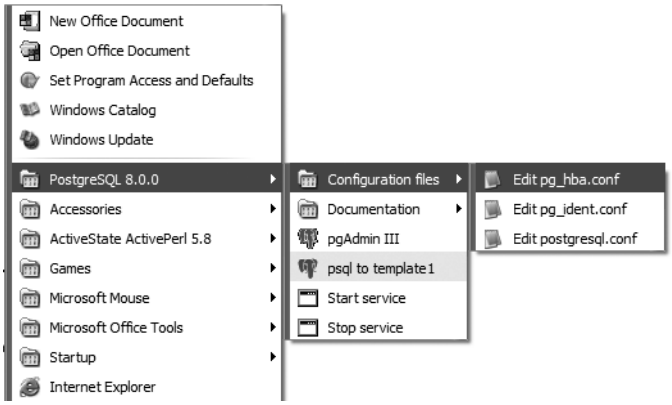


圖 3-8 PostgreSQL 程式功能表

配置客戶機訪問

為了配置遠端主機和使用者可以連接到 PostgreSQL 服務，你需要編輯 pg_hba.conf 文件。檔包含大量注釋記錄用於遠端存取的選項。在我們的安裝示例中，我們允許局域網中任何主機的任何使用者訪問伺服器上的資料庫。為了達到這個目的，我們添加以下的一行記錄到檔案結尾：

```
host    all all 192.168.0.0/16  trust
```

這意味著所有 IP 位址由 192.168 開始的電腦可以訪問所有的資料庫。最簡單的使配置生效的方法就是重啟伺服器。

Windows 系統中的 pg_hba.conf 檔和 Linux 和 UNIX 系統中的格式相同。其他的訪問配置示例，請參考本章前面的“配置連接許可權”小節。

建立示例資料庫

現在我們已經讓 PostgreSQL 運行起來了，我們將建立一個簡單的資料庫，我們給資料庫起名為 bpsimple，用來支援我們的客戶訂單表示例。這個資料庫（與第 8 章建立的叫做 bpfinal 的改造版本）將在整個本書中使用。我們將在後面的章節全面討論建立資料庫和建表以及填充表的細節。在這裡，我們只展示實現的步驟和 SQL 腳本，這樣我們將有一個用於示例的資料庫。在我們開始前，一個簡單的檢查 PostgreSQL 是否在運行的方法就是查找 postmaster 進程。在 Windows 系統中，通過工作管理員查找 postmaster.exe 進程。在 UNIX 和 Linux 系統中，運行以下的命令：

```
$ ps -el | grep post
```

如果有一個叫 postmaster 的進程在運行（名稱可能是縮寫顯示），那麼你已經在運行一個 PostgreSQL 服務程式了。

添加使用者記錄

在我們能建立一個資料庫前，我們需要通過在系統中建立使用者記錄來告訴 PostgreSQL 有效的用戶。PostgreSQL 資料庫系統的有效使用者可以讀數據，插入資料或者更新資料；建立自己的資料庫以及控制對這些資料庫管理的資料的訪問。我們使用 PostgreSQL 的 createuser 工具建立使用者記錄。

在 Linux 和 UNIX 系統中，使用 su 命令（從 root）變成 PostgreSQL 管理用戶 postgres。然後運行 createuser 註冊用戶。使用者給出的使用者登錄名需要時有效的 PostgreSQL 使用者。讓我們為（UNIX/Linux 的）使用者 neil 建立資料庫使用者：

```
$ 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 系統中，打開一個命令列視窗並切換工作目錄到 PostgreSQL 所在目錄（在這裡，我們按在默認位置：C:\Program Files\PostgreSQL\9.0），然後運行 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 選項指出了你想要建立的新用戶的編號。必須是 PostgreSQL 的使用者才能建立使用者，通常情況下就是 postgres 使用者。-P 選項通知 createuser 提示輸入新使用者的密碼。

在這裡，我們允許 neil 建立新資料庫和建立新使用者。本書中的一些例子中使用了另一個叫 rick 的使用者，它允許建立資料庫，但不允許新建用戶。如果你想重新試試新建用戶，現在就可以建立這個用戶了。

一旦你建立了擁有這些許可權的 PostgreSQL 用戶，你就有能力建立 bpsimple 資料庫了。

建立資料庫

在 Linux 和 UNIX 系統中建立資料庫，回到你自己的用戶（非 root 用戶）並運行以下命令：

```
$ /usr/local/pgsql/bin/createdb bpsimple
CREATE DATABASE
$
```

在 Windows 系統中，運行 createdb.exe 命令：

```
C:\Program Files\PostgreSQL\8.4\bin>createdb -U neil bpsimple
Password:
CREATE DATABASE
```

你先可以使用互動式終端工具 psql（從本地）連接到伺服器了。在 Linux 和 UNIX 系統中，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 系統中，使用這個命令：

```
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=#
```

你也可以選擇 Windows 開始功能表中的 psql 專案連接到 template1 資料庫，然後在 psql 中切換資料庫：

```
template1=# \c bpsimple
You are now connected to database "bpsimple".
bpsimple=#
```

現在你已經登錄到 PostgreSQL，準備好執行命令了。你可以使用\q 命令退回到 shell。

下一步，我們將使用一套 SQL 語句來建立和填充示例資料庫。

建表

你可以在 psql 的命令提示符後面通過輸入 SQL 命令在你的 bpsimple 資料庫中建表。但是，更簡單的辦法是下載代碼包並解壓後，通過命令\i <檔案名>執行這些命令（psql 中的\i 命令可以用來執行指令檔，指令檔是由陣列 SQL 語句和其他 PostgreSQL 命令組成的文字檔）。由於命令都是純文字，所以只要你願意你都可以以通過你的文字編輯器編輯它。

輸入下麵的命令運行 create_tables-bpsimple.sql 來建表：

```
bpsimple=# \i create_tables-bpsimple.sql
CREATE TABLE
...
bpsimple=#
```

把所有的資料庫模式（表，索引以及存儲過程）寫入指令檔是一個好習慣。只有這樣，如果資料庫需要重新建立，你可以通過腳本完成。腳本也可以用於在任何模式需要更新時使用。

以下為建立我們需要的表的 SQL（第 2 章設計的表）這些 SQL 可以在 create_tables-bpsimple.sql 代碼包裡頭找到：

```
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

假設以後的某天，你需要刪除所有的表並重新開始，命令集就在 drop_tables.sql，就像這樣：

```
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;
```

警告你一聲，如果你刪掉表，你也將丟失表裡的資料！

注：drop_tables.sql 腳本也明確地刪除了叫做序列生成器的特殊的屬性，它被 PostgreSQL 用來管理資料自動順序增長的列。在 PostgreSQL 8.0 和以後的版本，序列生成器將在相關表被刪除時被自動刪除，但我們保留了命令用於相容以前的版本。

如果你在建表後運行了這個腳本你需要在嘗試往表裡頭填充資料前再次運行 create_tables-bpsimple.sql。

填充表

最後，我們需要往表裡頭添加一些資料，或者叫做填充表。示例數據在 pop_tablename.sql 裡頭。如果你想使用自己的資料，你得到的結果將和本書提供的不同。所以，除非你確信，否則最好是使用我們提供的示例資料。

在這裡使用分行符號的必要性是為了將命令列印在列印頁上。你可以每個每行一條命令。你必須包含結尾的分號，

這告訴 psql 每行 SQL 的結束點。

customer 表

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

item 表

```
INSERT INTO item(description, cost_price, sell_price)
VALUES('Wood Puzzle', 15.23, 21.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Rubik Cube', 7.45, 11.49);
INSERT INTO item(description, cost_price, sell_price)
```

```
VALUES('Linux CD', 1.99, 2.49);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Tissues', 2.11, 3.99);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Picture Frame', 7.54, 9.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Fan Small', 9.23, 15.75);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Fan Large', 13.36, 19.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Toothbrush', 0.75, 1.45);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Roman Coin', 2.34, 2.45);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Carrier Bag', 0.01, 0.0);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Speakers', 19.73, 25.32);
```

barcode 表

```
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241527836173', 1);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241574635234', 2);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6264537836173', 3);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241527746363', 3);
INSERT INTO barcode(barcode_ean, item_id) VALUES('7465743843764', 4);
INSERT INTO barcode(barcode_ean, item_id) VALUES('3453458677628', 5);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6434564564544', 6);
INSERT INTO barcode(barcode_ean, item_id) VALUES('8476736836876', 7);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241234586487', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9473625532534', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9473627464543', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('4587263646878', 9);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9879879837489', 11);
INSERT INTO barcode(barcode_ean, item_id) VALUES('2239872376872', 11);
```

orderinfo 表

```
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(3,'03-13-2000','03-17-2000', 2.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(8,'06-23-2000','06-24-2000', 0.00);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
```

```
VALUES(15,'09-02-2000','09-12-2000', 3.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(13,'09-03-2000','09-10-2000', 2.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(8,'07-21-2000','07-24-2000', 0.00);
```

orderline 表

```
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 4, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 7, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 9, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 10, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 7, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 4, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(3, 2, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(3, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(4, 5, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(5, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(5, 3, 1);
```

stock 表

```
INSERT INTO stock(item_id, quantity) VALUES(1,12);
INSERT INTO stock(item_id, quantity) VALUES(2,2);
INSERT INTO stock(item_id, quantity) VALUES(4,8);
INSERT INTO stock(item_id, quantity) VALUES(5,3);
INSERT INTO stock(item_id, quantity) VALUES(7,8);
INSERT INTO stock(item_id, quantity) VALUES(8,18);
INSERT INTO stock(item_id, quantity) VALUES(10,1);
```

在 PostgreSQL 系統運行，資料庫建立表建立以及填充資料後，我們準備好繼續探索 PostgreSQL 的功能了。

摘要

在本章中，我們瞭解了在 Linux 和類 UNIX 系統以及 Windows 中安裝 PostgreSQL 資料庫的一些選秀。最簡單的方法是用某種預編譯的二進位安裝包。我們提供了在 Linux 系統中用安裝包，類 UNIX 系統中用源碼以及 Windows 系統中使用安裝包的每一步的指令用於編譯、安裝、配置和確認安裝一個可運行的環境。

最後，我們建立了一個我們將在本書剩下的部分中用於展示 PostgreSQL 系統功能的示例資料庫。我們將從下章的訪問你的資料開始，探索 PostgreSQL 系統。

第四章訪問你的資料

到現在為止，我們在本書中碰到 SQL 的情況都是非正式的。我們碰到過一些用不同方法獲取資料的語句，以及一些用於建表和填充資料的 SQL。

本章，我們將從 SELECT 語句開始，稍微正式地研究 SQL。事實上，本章全部都是用於講解 SQL 語句。你的第一映射可能認為用一章講解一條 SQL 有點過分，但實際上 SELECT 語句是 SQL 語句的核心。一旦你理解了 SELECT，你實際上已經完成學習 SQL 最難的部分。

在下一章，我們將討論一些你可以使用的 GUI 用戶端，但現在，我們將使用一個隨 PostgreSQL 提供的簡單的命令列工具 psql 來訪問資料。

本章我們將涵蓋以下主題：

- 使用 psql 命令與 PostgreSQL 資料庫交互
- 使用一些簡單的 SELECT 語句檢索資料
- 通過覆蓋列名提高輸出的可讀性
- 在檢索資料的時候控制資料行的順序
- 隱藏重複行
- 在檢索資料的時候執行數學計算
- 為了方便而給資料庫別名
- 使用模式匹配指出需要檢索的資料
- 使用各種資料類型進行匹配
- 通過單個的 SELECT 語句在多個表中檢索資料
- 在一條 SELECT 語句中關聯三個甚至跟多的表

到現在為止，你應該已經有搭好 PostgreSQL 並且在運行了。通過這一章，我們將使用在第二章設計並在第三章生成的示例資料庫。

使用 psql

如果你跟著第三章裡頭的指令做了，現在你應該有一個你可以通過 PostgreSQL 的登錄提示符訪問的叫做 bpsimple 的資料庫了。

> 注：除了進行一些特殊的資料庫管理操作，你永遠都不應該使用 postgres 使用者訪問 PostgreSQL 資料庫。

在 Linux 系統中啟動

如果你是在一個 Linux 系統中，並且你建立了一個普通的沒有密碼的使用者，你可以通過在連接命令中包含用戶名來啟動 psql 來訪問 bpsimple 資料庫。例如，使用 rick 使用者訪問資料庫，你可以輸入：

```
$ psql -d bpsimple -U rick
```

你應該會看到以下內容：

```
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=>
```

現在我們已經準備好輸入命令了。如果你建立的使用者需要密碼，你應該收到輸入密碼的提示，這依賴於實際的認證配置。我們將在第十一章解釋更多關於認證的內容。

在 Windows 系統中啟動

如果你是使用 Windows，可以通過打開開始功能表的 psql 首先連接到 template1。你將受到提示輸入 postgres 使用者密碼的提示。在成功連接後，使用\c 命令使用你自己的用戶名（在這裡為 rick）切換到 bpsimple 資料庫，如下所示：

```
template1=# \c bpsimple rick
You are now connected to database "bpsimple" as user "rick".
bpsimple=>
```

注意提示符從=#變成了=>，這意味著你不再擁有許可權建立資料庫。

你也可以選擇自己在開始功能表建立這個命令的快捷方式。例如，我們通過 IP 位址（-h 選項）作為 risc 使用者（-U 選項）連接到遠端伺服器的 bpsimple 資料庫（-d 選項）。

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

如果你想作為管理員用戶連接，你可以用 postgres 代替 rick。如果你需要連接到本地的伺服器，你可以忽略掉-h 選項。

解決啟動問題

如果 psql 囉嗦關於 pg_shadow 的問題，可能是你沒有在資料庫建立你提供的使用者。如果抱怨不知道這個用戶或者缺乏許可，可能是你沒有正確賦權。請參考第三章關於賦權的細節。

如果你還是沒成功，現在最簡單的修復問題的方法是刪除資料庫和使用者並重建他們。現在的做法是使用\q 命令退出 psql 命令，然後你就退出到 Linux 的命令提示符或者退出了 Windows 的命令列視窗。

之後，使用 postgres 使用者重新連接到資料庫。在 Linux 中，在命令列按照下面的方法做：

```
$ psql -d template1
```

在 Windows 中，使用開始功能表的快捷方式連接 psql 到 template1 資料庫。

輸入你在安裝過程中輸入的密碼，然後你會看到以下的提示符：

```
template1=#
```

現在，可以通過以下的形式刪除資料庫和使用者（本例中的用戶為 rick）：

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

然後重建使用者並給出密碼（本例中的密碼為 apress4789）：

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

CREATEDB 選項允許使用者建立他們自己的資料庫。

現在使用你新建的使用者連接到資料庫：

```
template1=# \c template1 rick
Password:
template1=#
```

你現用你的新用戶（在本例中還是 rick 這個使用者）連接到資料庫 template1。下一步是建立 bpsimple 資料庫：

```
template1=> CREATE DATABASE bpsimple;
CREATE DATABASE
template1=>
```

使用新使用者重新連接到 bpsimple 資料庫：

```
template1=> \c bpsimple rick
You are now connected to database "bpsimple" as user "rick".
bpsimple=>
```

然後你需要回到上一章從“建表”小節到結尾位置的步驟，以便建立本章需要的示例資料庫並填充資料。

如果你嘗試建立資料庫的時候看到下面的錯誤資訊：

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

這意味著有其他的會話連接到了資料庫 template1——也許是另一個 psql 的會話或者 pgAdmin III 這樣的圖形介面程式。請確保你當前的 psql 會話是唯一使用 template1 資料庫的會話並重試。

要檢查你是否已經建立了 bpsimple 資料庫裡頭的表，資料如\dt，然後敲回車鍵，然後你應該可以看到類似於以下的輸出：

```
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 列顯示為你的登錄名（本例中為 rick）。

注：你可能看到一些名字為 pg_ts_dict，pg_ts_parser，pg_ts_cfg 以及 pg_ts_cfgmap 一類的表。這些表可能是一些可選的用戶捐獻的工具的附加表。你可以安全地忽略它們。

你也可以通過 pgAdmin III 流覽資料庫時看到同樣的資訊，先是資料庫 bpsimple，然後是模式，然後是 public，然後是表，就像圖 4-1 所示。

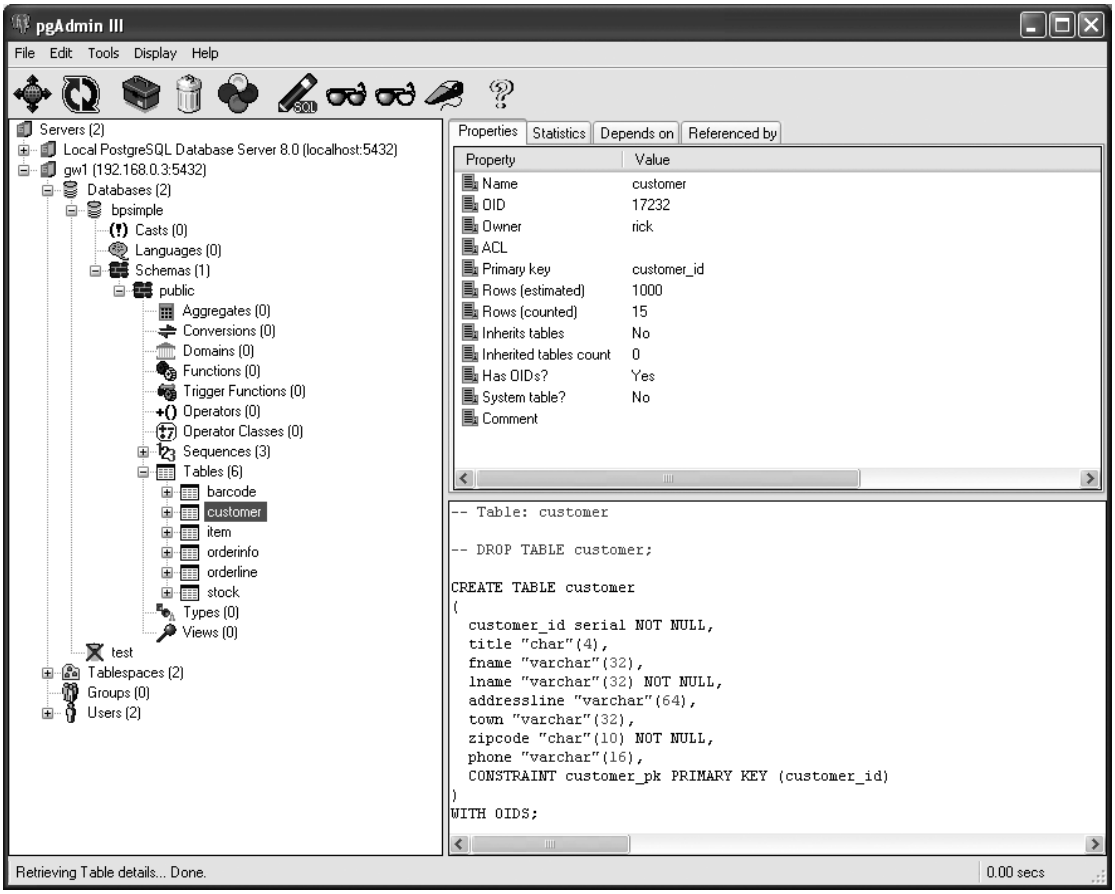


圖 4-1 通過 pgAdmin III 檢查 bpsimple 資料庫

我們將在第十一章覆蓋絕大多數資料庫和使用者管理的細節。

使用一些基本的 psql 命令

在本章中我們只會使用很少的幾個 psql 命令（我們將在第五章碰到全部的命令）。現在，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
\?	獲得說明消息
\do	列出操作類型
\dt	列出表
\dT	列出類型
\h <cmd>	列出 SQL 命令的幫助；用實際的命令代替<cmd>
\i <filename>	執行檔<filename>裡頭的命令
\r	重置緩衝器（忽略任何輸入……）
\q	退出 psql

表 4-1 中列出的命令必須在輸入回車後才能執行。

你還可以通過使用方向鍵回到之前的行並編輯它們。在 Linux 系統中，psql 的這個功能依賴於 GNU 的 readline 基礎設施，通常它已經被安裝，但不保證總是已經安裝。

現在我們已經準備好通過 SQL 命令訪問 PostgreSQL 資料庫。在下一章，我們將碰到一些可以使用 PostgreSQL 的圖形介面工具，但是在本章，我們將使用 psql 工具。

注：如果你堅持使用圖形介面工具，你可以先看看第五章。然後你可以回到本章。你應該可以通過任何圖形介面工具通過直接輸入 SQL 到 PostgreSQL 執行本章的所有例子，例如 pgAdmin III（http://www.ggadmin.org/）。但是，我們建議至少在這章中使用命令列，因為知道如何使用命令列工具訪問 PostgreSQL 的基本方法通常會非常方便。

使用 SELECT 語句

和所有關聯式資料庫一樣，我們通過 SELECT 語句從 PostgreSQL 裡檢索資料。它也許是最複雜的 SQL 語句，事實上它才是有效使用關聯式資料庫的心臟。

讓我們從簡單的從一個表中的全部資料開始研究 SELECT 命令。我們通過很基礎的形式的 SELECT 語句，指出一個列的列表，以及一個帶有表名的 FORM 從句：

```
SELECT <逗號分隔的列的列表> FROM <表名>
```

如果我們不記得確定需要的列名，或者想要看到所有的列，我們可以直接使用一個星號（*）來代替列的列表。

注：本書中，我們使用大寫的 SQL 關鍵字以便讓它們更突出。SQL 不是區分大小寫的，僅有少數資料庫軟體對於表名是區分大小寫的。SQL 資料庫存儲的資料是區分大小寫的，所以字串“Newtown”和“newtown”是不同的。

嘗試：從表中選擇所有的列

我們開始於從 item 表中提取所有資料：

```
SELECT * FROM item;
```

記住分號（;）對於 psql 非常有用，它可以告訴 psql 輸入結束了。嚴格地說，它不是 SQL 的一部分。如果你願意，你可以通過輸入\g 命令告訴 psql SQL 語句已經結束，這和分號的效果一樣。如果你使用不同的工具發送 SQL 給 PostgreSQL，你可以不需要任何一種結束字元。

輸入命令後，你會看到 PostgreSQL 的回應：

```
bpsimple=> SELECT * FROM item;
 item_id |  description   | cost_price | sell_price
---------+----------------+------------+------------
       1 | Wood Puzzle    |      15.23 |      21.95
       2 | Rubic Cube     |       7.45 |      11.49
       3 | Linux CD       |       1.99 |       2.49
       4 | Tissues        |       2.11 |       3.99
       5 | Picture Frame  |       7.54 |       9.95
       6 | Fan Small      |       9.23 |      15.75
       7 | Fan Large      |      13.36 |      19.95
       8 | Toothbrush     |       0.75 |       1.45
       9 | Roman Coin     |       2.34 |       2.45
      10 | Carrier Bag    |       0.01 |       0.00
      11 | Speakers       |      19.73 |      25.32
```

```
(11 rows)
bpsimple=>
```

解析

我們使用*作為列名，簡單的告訴 PostgreSQL 我們需要 item 表裡頭的所有列的資料。PostgreSQL 正確地給了我們想要的資料，還整齊地將列名用管道符號（|）區分列的方式排列了資料。它還告訴我們檢索到了多少行。

但假設我們不需要所有的列呢？通常，你應該告訴 PostgreSQL 甚至實際上所有的關聯式資料庫僅僅檢索你實際上需要的資料。檢索每行資料的每列都會增加一些工作。沒有必要讓伺服器做不必要的工作；有必要總是讓事情清晰有效。

你還會發現，一旦你開始將 SQL 嵌入其他語言（參考第十四章），指定列名將保護你防止因資料庫模式的改變而帶來的問題。例如，如果你使用*來檢索所有的列而在代碼被測試後一個列被插入到表中後，你會發現你處理的你想要的列中的資料是從另一個列裡頭來的了。如果你使用的列被刪除了，那麼你程式中的 SQL 將執行失敗，因為列已經無法被檢索了；但是，這是一個很容易找到並糾正的錯誤，而不需要應用程式在處理資料時訪問錯誤的列的時候才發現。如果你指定列名，你可以在做資料庫變更前搜索你所有的代碼中看是否出現這個列名，來避免發生這類錯誤。

讓我們試試只檢索我們需要的列吧。就像前面的語法中見到的一樣，我們通過指定我們需要的用逗號分隔的列做到。如果我們需要的列的順序不是我們建表時指定的順序，那麼也沒問題——我們可以按我們的需要指定列的順序，然後資料庫就會按照這個順序返回資料。

嘗試：按照特定的順序選擇列

為了檢索所有客戶所在的城鎮名字和他們的姓，我們必須指定城鎮和姓所在的列的名字，當然，還要指出檢索這些資料所在的表。以下是我們需要的語句以及 PostgreSQL 的回應：

```
bpsimple=> SELECT town, lname FROM customer;
   town     |  lname
------------+---------
Hightown   | Stones
 Lowtown    | Stones
 Nicetown   | Matthew
 Yuleville  | Matthew
 Oakenham   | Cozens
 Nicetown   | Matthew
 Bingham    | Stones
 Bingham    | Stones
 Histon     | Hickman
 Tibsville  | Howard
 Bingham    | Jones
 Winersby   | Neill
 Oxbridge   | Hardy
 Welltown   | O'Neill
 Milltown   | Hudson
(15 rows)
```

```
bpsimple=>
```

解析

PostgreSQL 從我們指定的表返回了我們所有的資料行，但只返回了我們需要的列。它還按照我們在 SELECT 語句中指定的列的順序返回列的資料。

覆蓋列名

你會發現輸出的結果使用列的名字作為輸出的列標題。有時候它不易於閱讀，尤其是當輸出的列不是資料庫的列，它沒有名字。有一個非常簡單的語法指定用於顯示的列名（列的別名），方法是在 SELECT 語句的每個列後面添加 AS "<*顯示名*>"。你可以指定所有的列的別名，或者是僅僅選擇幾個。你不指定別名的地方，PostgreSQL 就僅僅使用列名。

例如，通過添加有意義的名字改變前面輸出，我們可以這樣：

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

我們將看在下一章看到它的示例。值得注意的是，在 SQL92 標準中，AS 從句是可選的；但是，到 8.0 版本為止，PostgreSQL 仍然需要 AS 關鍵字。

控制行的順序

到現在為止我們已經按列檢索到了我們需要的資料，但這些資料看起來還不是按最適當的方式顯示的。我們看到的資料可能看上去是我們當初插入資料庫的順序，但這看上去很簡單是因為我們還沒有通過插入和刪除行更新過資料。

就像我們在第二章提到的，不像試算表，資料庫中的行的順序不需要指定。資料庫伺服器可以隨意按照自己最高效的方法存儲行，但這通常不是最自然的方法來查看資料。你看到的輸出是按沒有任何意義的順序排序的，並且可能下次你重新請求這些資料的時候，它可能是按另一種順序顯示的。通常，資料將按其在資料庫內部存儲的順序返回。包含 PostgreSQL 在內的所有 SQL 資料庫，沒有一個被要求按照一定的順序返回資料，除非你在檢索它們的時候指必須按順序。

我們可以通過在 SELECT 語句後面添加 ORDER BY 從句控制 SELECT 語句顯示的資料的順序，ORDER BY 從句指定我們需要的資料返回的順序。語法如下所示：

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

在最後面的有點奇怪的語法的意思是，在列名後面，我們可以選擇 ASC（昇冪排序）或者 DESC（降冪排序）之一。默認使用的是昇冪排序。之後資料按照我們指定的列和要求的順序返回。

嘗試：給數據排序

在本例中，我們按 town 排序資料，同時我們將覆蓋 lname 列的名字，就像我們上一節所說的，讓輸出看上去更易讀。

以下是我們需要的命令以及 PostgreSQL 的回應：

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

```
 Bingham   | Stones
 Bingham   | Stones
 Bingham   | Jones
 Hightown  | Stones
 Histon    | Hickman
 Lowtown   | Stones
 Milltown  | Hudson
 Nicetown  | Matthew
 Nicetown  | Matthew
 Oakenham  | Cozens
 Oxbridge  | Hardy
 Tibsville | Howard
 Welltown  | O'Neill
 Winnersby | Neill
 Yuleville | Matthew
(15 rows)

bpsimple=>
```

注意因為我們需要資料按昇冪排列，我們可以省略 ASC，因為昇冪排列是預設的排序順序。你可以看到，資料是按城鎮的昇冪排列的。

解析

現在，我們在我們最初的語句中做了兩處變化。我們添加了 AS 從句用將第二列的名字改為“Last Name”，這讓結果更易讀，我們還添加了一個 ORDER BY 子句用於指定 PostgreSQL 返回資料給我們的順序。

有時候，我們需要更進一步，按照不止一個列排序。例如，在之前的輸出，雖然資料時按照 town 排序的，但沒有按照 Last Name 排序。比方說我們可能發現所有的 town 為 bingham 的客戶中，Jones 的名字會出現在 Stones 的後面。

我們可以通過指定超過一個列的順序細化輸出的順序。如果我們需要，我們甚至可以指定一列按昇冪排列同時另一列按降冪排列。

嘗試：用 ASC 和 DESC 給數據排序

再試試我們的 SELECT 命令，只是這次，我們需要指定城鎮名字降冪排列，相同城鎮的姓昇冪排列。

我們需要的語句和 PostgreSQL 的響應如下：

```
bpsimple=> SELECT town, lname AS 'Last Name' FROM customer
ORDER BY town DESC, lname ASC;
   town    | Last Name
-----------+-----------
 Yuleville | Matthew
 Winnersby | Neill
 Welltown  | O'Neill
 Tibsville | Howard
```

```
 Oxbridge  | Hardy
 Oakenham  | Cozens
 Nicetown  | Matthew
 Nicetown  | Matthew
 Milltown  | Hudson
 Lowtown   | Stones
 Histon    | Hickman
 Hightown  | Stones
 Bingham   | Jones
 Bingham   | Stones
 Bingham   | Stones
(15 rows)

bpsimple=>
```

解析

就像你看到的，PostgreSQL 先將資料按 town 的降冪排列，因為它是 ORDER BY 從句後面的第一個列。然後它將相同 town 裡頭有多條記錄的資料按昇冪排列。這時候，雖然 Bingham 是在檢索到的資料的末尾，但其中的客戶的姓仍然是按昇冪排列的。

通常，你可以用來排序的列被強制限制於你選擇用於輸出的列（看上去不無道理）。但至少到當前版本的 PostgreSQL 未知，不強制執行這一標準限制，它可以接受用於 ORDER BY 後面的列不在你選擇的列的清單裡頭的情況。但是，這是非標準 SQL，所以我們建議你避免使用這個功能。

消除重復資料

你可能發現之前的輸出有好幾處重複行。例如，以下的城鎮和姓出現了兩次：

```
Nicetown  | Matthew
Bingham   | Stones
```

這裡發生什麼了？在原始資料中，確實在 Nictown 有兩個叫做 Matthew 的客戶和在 Bingham 有兩個角 Stones 的客戶。作為參考，以下為顯示了名的相應行：

```
Nicetown  | Matthew | Alex
Nicetown  | Matthew | Neil
Bingham   | Stones  | Richard
Bingham   | Stones  | Ann
```

PostgreSQL 分別列出了 Nicetown 和 Matthew 的兩行，以及 Bingham 和 Stones 的兩行，而且應該都是正確的。每個鎮都有同名的兩個客戶。因為我們沒有說要列出所有能夠區別它們的列，所以它們看上去似乎完全一樣。

資料庫預設的行為是列出所有的行但這不總是我們所需要的。例如，我們可能只是列出有我們客戶的城鎮，也許是用來決定我們需要在哪裡建立分發中心。基於我們現在的知識，我們可能嘗試這麼做：

```
bpsimple=> SELECT town FROM customer ORDER BY town;
   town
-----------
```

```
 Bingham
 Bingham
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
 Nicetown
 Nicetown
 Oakenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(15 rows)

bpsimple=>
```

PostgreSQL 列出了所有的城鎮，在客戶表中每出現一次就列了一次。這沒錯，但它可能不是不太是我們想要的。我們真正需要的清單是每個城鎮出現一次；或者說，我們需要列出不同的鎮。

在 SQL 中，你可以在 SELECT 語句中添加 DISTINCT 關鍵字來移除重複的行，語法如下：

```
SELECT DISTINCT <由逗號分隔的列的列表> FROM <表名>
```

和 SELECT 的所有從句一樣，你可以將這個從句與其他從句組合起來使用，例如重命名列或者指定排序方式。

嘗試：使用 DISTINCT

來嘗試列出 customer 表裡出現的所有不重複的 town 來。我們可以嘗試使用以下的代碼來獲得回應：

```
bpsimple=> SELECT DISTINCT town FROM customer;
   town
-----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
 Nicetown
 Oakenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(12 rows)
```

```
bpsimple=>
```

解析

關鍵字 DISTINCT 告訴 PostgreSQL 移除重複的行。注意輸出已經按 town 排序了，這是因為 PostgreSQL 選擇西安排序的方法實現 DISTINCT 從句。通常，我們不能假設資料總是被排序的。

如果你需要資料按照特殊的方式排序，你必須添加一個 ORDER BY 從句來指定順序。

注意 DISTINCT 從句不是僅僅關聯到某幾個列。你只能去除你選擇的所有列中的重復資料，而不能僅僅去除部分列的重復資料。例如，假設我們使用這種方式：

```
SELECT DISTINCT town, fname FROM customer;
```

我們仍將獲得 15 行，因為有 15 種城鎮和姓的組合。

這裡要做一個小小的警告：雖然看上去總是在 SELECT 中使用 DISTINCT 是個好主意，在現實中，這是一個壞主意。第一，使用 DISTINCT，意味著 PostgreSQL 需要在提取你的資料的時候需要做更多的事情來查重。除非你知道需要移除的重復資料，否則你不應該使用 DISTINCT 從句。第二個原因更加注重於實用。有時候 DISTINCT 會隱藏你 SQL 或者資料中的錯誤，而這種錯誤在顯示重複行的時候卻很容易看出。

警告：僅僅在你確定需要的時候使用 DISTINC，因為它需要更多的工作且可能隱藏錯誤。

執行計算

我們還可以在資料被發送出來前對資料執行一些簡單的計算。

假設我們需要顯示我們 item 表裡頭的每個項目的價格，我們只需要簡單的執行如下的 SELECT 語句：

```
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=>
```

假設我們需要按分查看它們的價格，我們可以在 SQL 中作簡單的計算，就像這樣：

```
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=>
```

輸出的小數點和奇怪的列名看上去有點古怪，所以讓我們通過 SQL 的一個功能去掉小數點，同時明確地給出結果列的名字。我們使用 cast 函數轉換列的類型，這與使用 AS 的給列命名的從句一同，讓輸出更好看一點：

```
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=>
```

我們將在本章稍後的“設置時間和日期格式”的小節詳細討論 cast 函數的功能。

選擇行

本章到現在為止，我們總是工作在所有的資料行上，或者至少是所有的不同行。是時候看看我們怎樣選擇我們需要的行了。你可能不會覺得奇怪因為我們將要學習一個新的 SELECT 語句的從句：WHERE 從句。

WHERE 的最簡單的語法如下：

```
SELECT <逗號分隔的列> FROM <表名> WHERE <條件>
```

可能有很多可以通過關鍵字 AND、OR 以及 NOT 組合的條件。

條件裡頭使用的標準的比較運運算元列在表 4-2 中。比較運運算元可以用在大多數類型中，包括數位和字串，甚至一些特殊的條件例如日期比較的條件，這些我們將在本章稍後看到。

表 4-2 標準比較運運算元

運運算元	描述
<	小於
<=	小於或等於
=	等於
>=	大於或等於
>	大於
<>	不等於

我們將開始於一個簡單的條件：選擇住在 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=>
```

這非常簡單易懂，不是嗎？注意需要用單引號包括字串 Bingham 以表明它是一個字串。還要注意 Bingham 因為是與資料庫中的資料比較，因此它是區分大小寫的。如果我們使用了 `town = 'bingham'`，將沒有資料返回。

我們可以有多個條件，通過 AND，OR 以及 NOT 組合起來，並添加括弧使運算式看上去更清晰。我們也可以針對我們沒有選擇的列使用條件。你應該還記得這對於 ORDER BY 從句來說是不適用的。

嘗試：使用運運算元

讓我們嘗試一些更複雜的條件集。假設我們需要看看住在 Bingham 或者 Nicetown 的 title 不是 Mr.的客戶。以下是我們需要的語句以及 PostgreSQL 的回應：

```
bpsimple=> SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr'
bpsimple-> AND (town = 'Bingham' OR town = 'Nicetown');
title   | fname |  lname   |   town
-------+-------+---------+----------
 Miss   | Alex  | Matthew | Nicetown
 Mrs    | Ann   | Stones   | Bingham
(2 rows)

bpsimple=>
```

解析

雖然它咋一眼看起來有點複雜，但這個語句實際上非常簡單。第一部分就是我們普通的 SELECT，列出我們在輸出中需要看到的列。在 WHERE 從句之後，我們先檢查 title 不是 Mr.，然後使用 AND 連接我們下一個條件是否為真。第二個條件是 town 可以是 Bingham 或者 Nicetown。注意我們需要用括弧使需要分組的從句看起來更清晰。

你應該意識到 PostgreSQL，甚至其他任何的關聯式資料庫，不保證按照你寫在 SQL 語句裡頭的從句的連續處理。它只保證針對 SQL 提出的“問題”得出正確的答案。通常，關聯式資料庫都有精密的優化器，它檢查請求，然後確定能完成它的最優方法。優化器不是完美的，所以你經常需要通過不同的方式寫語句以獲得更好的運行效果。對於像這裡的語句一樣的相當簡單的語句，我們可以很安全地假設優化器可以做得很好。

提示：如果你想知道 PostgreSQL 怎麼處理 SQL 語句，你可以通過在 SQL 前頭加 EXPLAIN 首碼讓它告訴你。而不是執行這個語句。PostgreSQL 將告訴你語句將怎麼處理。

使用更複雜的條件

我們經常需要的一個功能是需要對字串進行部分匹配。例如，我們需要找一個叫 Bobert 的人，但在資料庫裡頭可能保存的是縮寫的 Rob 或者 Bob 一類的。在 SQL 中有一些特殊的操作讓我們針對字串的一部分或者字串的清單工作起來更輕鬆。

第一個新條件是 IN，它允許我們針對一系列的條目進行檢查，而不是使用一系列的 OR 條件。看看以下的語句：

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND
(town = 'Bingham' OR town = 'Nicetown');
```

我們可以將它們重寫為以下的樣子：

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND
town IN ('Bingham', 'Nicetown');
```

我們將獲得相同的結果，雖然有可能輸出的行的順序可能不同，因為我們沒有使用一個 ORDER BY 從句。在這裡，IN 從句除了可以簡化運算式外沒有特別的優勢。當我們在第七章遇到子查詢的時候，我們會再次使用 IN，在那種情況下 IN 從句會提供更高級的優勢。

下一個新的條件是 BETWEEN，它允許我們通過指定邊界檢查一個值的範圍。假設我們想要選擇 customer_id 在 5 和 9 之間的行。與其使用一系列的 OR 條件或者一個 IN 帶著一堆值，不然我們寫一個簡單的條件如下：

```
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=>
```

也可以針對字串使用 BETWEEN；但是，我們必須小心，因為有可能結果並不完全是你所期望的，而且你必須知道原因，就像之前提起的，因為字串的比較是區分大小寫的。

嘗試：使用複雜的條件

讓我們嘗嘗使用 BETWEEN 比較字串。假設我們想獲得一個首字母為 B 和 N 之間的所有不同城鎮的列表。所有的城鎮的開始字母都是大寫的，所以我們可以依照以下的寫法：

```
bpsimple=> SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'N';
   town
----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
(5 rows)

bpsimple=>
```

如果你仔細地看結果，你會發現這個 SQL 沒有達到如期的效果。Newtown 在哪裡？它當然是以 N 開始的，卻沒有被列出來。

它為什麼沒有生效

這條語句沒有達到我們效果的原因是 PostgreSQL 依照 SQL 標準，用填充你給的字串到需要比較的字串的長度然後再比較。所以當我們比較 Newtown 的時候，PostgreSQL 用“N　”（N 後面填充了六個空白）和 Newtown，而空白在 ASCII 表中的位置再其他所有字元之前，所以判斷的結果是 Newtown 比“N　”大，所以它不應該在結果列表中

怎麼讓它生效

其實很容易讓它像預期的一樣工作。我們既可以通過在 N 後面添加字元 z 也可以在 BETWEEN 從句裡使用字元表裡頭的下一個字元 O，來避免搜索字串時添加空白的行為帶來的問題。當然，如果有一個鎮確實叫做 O，我們可能會錯誤地將它取出，所以你要小心使用這種方法。通常使用 z 比 Z 要好，因為 z 在 ASCII 表中出現在 Z 之後。於是，我們的 SQL 應該像這樣：

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

注意我們沒有在 B 之後加上 z，因為 B 後添加空格確實能匹配到 B 開頭的所有城鎮，因為它是一個開始點而不是結束點。還要注意如果一個城鎮開始於字元 Nzz，我們仍將無法找到它，因為我們將用“Nz ”與 Nzz 比較，然後確定 Nzz 在 Nz 之後，因為字串 Nz 字串依舊被填充了一個空白，而空白的位置比我們比較的字串相對位置的字元 z 的位置靠前。

這種比較確實有一些很微妙的行為，所以如果你確實要使用 BETWEEN 比較字串，需要仔細考慮關於需要匹配的內容。

模式匹配

到現在為止的字串比較操作到現在為止都運行得很好，但他們還不能實現現實世界的字串模式匹配。SQL 條件的模

式匹配需要使用 LIKE。

不幸的是，LIKE 使用與其他我們知道的程式設計語言不同的字串匹配規則集。但是，只要你記下了規則，就很容易使用了。當使用 LIKE 比較字串，你可以使用百分號（%）表示任何字串，你還可以使用底線（_）表示匹配一個字元。例如，匹配使用字母 B 開始的城鎮，我們可以這麼寫：

```
... WHERE town LIKE  'B%'
```

匹配以字母 e 結束的名字，我們可以這麼寫：

```
... WHERE fname LIKE '%e';
```

匹配只有四個字元長的名字，我們可以使用四個底線，像這樣：

```
... WHERE fname LIKE '____';
```

我們還可以在一個字串裡頭組合這兩種類型。

嘗試：模式匹配

讓我們找找所有的第二個字元為 a 的客戶。以下是能夠做到這點的語句：

```
bpsimple=> SELECT fname, lname FROM customer WHERE fname LIKE '_a%';
fname | lname
------+--------
Dave  | Jones
Laura | Hardy
David | Hudson
(3 rows)

bpsimple=>
```

解析

這個模式的第一部分“_a”匹配一個任意字元開始切第二個字元為小寫 a 的字串。模式的第二部分“%”匹配任何剩下的字元。如果我們不適用尾部的%，就只有兩個字元長的字串被匹配到。

限制結果集

在到現在為止我們使用的例子中，返回結果的行數都比較小，因為只有很少的幾行示例資料在我們的資料庫中。在現實世界的資料庫中，我們可以很容易找到數千行符合我們查詢標準的資料。如果我們正在寫我們的 SQL，優化我們的語句，我們當然不想要幾千行記錄滾過我們的螢幕。只要一些用於檢查我們邏輯的示例行就足夠了。

PostgreSQL 有一個非 SQL 標準的 SELECT 語句的擴展從句，LIMIT，可以在我們想限制返回的行數時給我們帶來很大的幫助。

如果你在你的 SELECT 語句後面添加一個帶數字的 LIMIT 從句，從第一行開始，只有最多你指定的行數的行將被返回。使用 LIMIT 的另一個稍微不同的方法是在與指定開始位置的 OFFSET 從句一起使用的時候。

用實際操作比用描述更能說明問題。以下我們只顯示了五條符合條件的行：

```
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=>
```

以下跳過了結果的前兩行，直接返回之後的五行：

```
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=>
```

也可以單獨使用 OFFSET，就像這樣：

```
bpsimple=> SELECT customer_id, town FROM customer OFFSET 12;
 customer_id |   town
-------------+----------
          13 | Oxbridge
          14 | Welltown
          15 | Milltown
(3 rows)

bpsimple=>
```

如果你想與其他的 SELECT 從句一起使用 LIMIT，LIMIT 從句應該總跟隨在普通的 SELECT 語句之後，只有在你使用 OFFSET 的時候，在後面跟隨 OFFSET 從句。

檢查空值（NULL）

到現在為止，我們還沒有一種方法檢查一個列是否包含一個空值。我們可以檢查它是否等於一個字串或是否不等於一個字串，這還不足夠。

如果你還記得第二章講到的空值是一個特殊的列的值表示不知道或者無關。。我們需要單獨看看怎麼檢測空值，因為它需要特別的考慮以確保結果是想要的。

假設我們知道在我們的表 testtab 裡頭的整數列 tryint 裡頭存儲了 0,1 或者 NULL，我們可以通過以下語句檢測它是不是 0：

```
SELECT * FROM testtab WHERE tryint = 0;
```

我們也可以通過以下語句檢測它是不是 1：

```
SELECT * FROM testtab WHERE tryint = 1;
```

我們需要另一種方法檢查這個值是不是 NULL。PostgreSQL 支援標準的 SQL 語法來檢測一個值是否為 NULL。我們使用 IS NULL 做到這點，像這樣：

```
SELECT * FROM testtab WHERE tryint IS NULL;
```

注意我們使用關鍵字 IS，而不是=符號。

我們還可通過添加一個 NOT 來反轉測試結果來測試這個值是不是 NULL 以外的值：

```
SELECT * FROM testtab WHERE tryint IS NOT NULL;
```

為什麼我們突然需要這個額外的語法？你可能很熟悉雙值邏輯值，也就是一件事情要麼是真要麼是假。這裡我們碰巧碰到了一個三值邏輯值，真、假以及未知。

不幸的是，NULL 的未知這個屬性，除了我們當前的可以直接檢查是否為 NULL 外，還有一些其他的效果

假設我們在一些 tryint 列的值為 NULL 的表中運行下麵的語句：

```
SELECT * FROM testtab WHERE tryint = 1;
```

當 tryint 實際上為 NULL 時，tryint = 1 意味著什麼？我們可以問一個問題：“未知是否等於 1？”這很有趣，我們不能確定語句的結果是假，也不能確定語句的結果是真。因此結果也是未知的，這樣導致 NULL 不匹配這條語句。如果我們反轉測試，比較 tryint != 1，為 NULL 的行也無法被列出來，因為這個運算式也無法為真。這看上去難以理解，因為表面上我們使用兩個相反的條件完成了兩個測試，但還是無法從表中檢索到那行資料。

注意 NULL 的這種特點很重要，因為太容易忘記 NULL 值。如果你在針對允許有 NULL 值的列使用條件得到了意外的結果集，很可能是有些行因為 NULL 值而引起了你的問題。

檢查時間和日期

PostgreSQL 有兩種基本類型用來處理日期和時間資訊：保存完整日期和時間資訊的 timestamp 類型和保存年月日資訊的 date 類型。PostgreSQL 有一些內建的功能來幫助我們使用日期和時間，而且它們通常都是非常難以操作的。

這裡，我們將集中討論那些我們常用的功能（你可以通過線上文檔找到所有的內置功能）。

在我們開始前，我們先要解決一個表面上沒用但很容易引起混亂的問題：我們怎麼指定一個日期？

當我們寫下 1/2/2005 這樣一個日期，我們的意思是什麼？對於歐洲人，這意味著這是 2005 年的二月一日，但是對於美國佬，這意味著 2005 年一月二日。這是因為歐洲人按照 DD/MM/YYYY 的格式讀取日期而美國人按照 MM/DD/YYYY 格式。ISO8601（被歐洲標準 EN28601 官方採用）指出的邏輯日期格式為 YYYY-MM-DD（但在日常中很少見）又加重了這個混亂。

PostgreSQL 為了迎合你的需要可以讓你改變日期處理的方式，所以在我們開始檢查日期和時間前，看看你怎樣才能控制 PostgreSQL 在這方面的行為是一個明智之舉。

設置時間和日期的風格

很遺憾，PostgreSQL 設置日期和時間處理模式的功能表面上看好像有點奇怪。

有兩個東西你可以控制：

- 日和月處理的順序，有美國和歐洲兩種樣式
- 顯示的格式；例如僅僅是數位元或者更加文本化的輸出

坦率地說，PostgreSQL 使用一個變數來處理以上兩項設定會導致一些混亂。好消息是 PostgreSQL 默認使用不容易混淆的 ISO-8601 樣式輸出日期和時間，它的標準格式是 YYYY-MM-DD hh:mm:ss.ssTZD。它給了你年、月、日、時、分、以及精確到小數點後兩位小數的秒，以及一個用於指定本地時間和 UTC 時間之間分鐘或者小時修正的時區識別字。例如，

一個完整的日期和時間應該像這樣：2005-02-01 05:23:43.23+5，也就是 2005 年 2 月 1 日 23 凌晨 5 點 23 分 43.23 秒，時區在 UTC 之前 5 小時。如果你使用 UTC 時間，也就是沒有時區資訊，標準裡頭告訴你你應該使用一個 Z（念做“Zulu”）來標記，雖然它看上去好像可以省略掉那個 Z。

對於 NN/NN/NNNN 形式的輸入，PostgreSQL 默認認為月份在日期之前（美國樣式）。例如 2/1/05 意味著 2 月 1 日。你還可以選擇 February 1,2005 的格式，或者 ISO 的 2005-02-01 這樣的格式。如果這些都是你需要的，那麼你很幸運，你不再必須知道更多關於控制 PostgreSQL 接收和顯示日期的內容，你可以往前跳到關於日期和時間的功能的小節。

預設樣式實際上由資料目錄裡頭的 postgresql.conf 檔控制，檔裡頭有一行類似於“datestyle = 'iso, mdy'”格式的內容。所以如果你想要，你可以改變全域的默認設置。

如果你需要更多控制處理日期的方法，PostgreSQL 也允許，但是有一點點麻煩。令人迷糊的是有兩個毫不相干的功能需要控制，而你需要用 datestyle 變數同時設置它們。可是千萬記住，這就是所有要做。在日期被輸入或者檢索的時候，PostgreSQL 用一種完全不依賴於任何用戶想要的表現形式的內部格式存儲日期。

給 psql 的命令的語法如下：

```
SET datestyle TO 'value';
```

為了設置月份和日期處理的順序，你需要將 datestyle 的值設置成 US 或者 European，分別表示月份在前（02/01/1997 表示 2 月 1 日）和日期在前（01/02/1997 表示 2 月 1 日）。

為了修改顯示格式，你也需要設置 datestyle，但是要設置成以下四個值之一：

- ISO，設置成 ISO-8601 標準，使用-作為分隔符號號，格式類似於 1997-02-01
- SQL，用於傳統樣式，格式類似於 02/01/1997
- Postgres，用於預設的 PostgreSQL 樣式，格式類似於 Sat Feb 01
- Geman，用於德國樣式，格式類似於 01.02.1997

注：在最新的發佈版本，PostgreSQL 預設的日期和時間戳記樣式使用 SQL 樣式。

你可以通過逗號分隔的一對參數設置 datestyle。所以，例如我們想指定資料顯示的樣式為 SQL 樣式且使用歐洲的樣式轉換月份和日期（日期在月份之前），我們使用如下的設置：

```
SET datestyle TO 'European, SQL';
```

我們可以在完整的安裝的時候設置日期處理日期或者設置它為繪畫的預設值，而不需要在每次會話中設置日期處理方式。如果你在完整安裝的時候想設置日期輸入的樣式，你可以在 postmaster 主服務進程啟動前設置 PGDATESTYLE 環境變數。在 Linux 設置這個環境變數，我們可以這樣做：

```
PGDATESTYLE="European, SQL"
export PGDATESTYLE
```

更好的改變默認日期處理方法的方法是修改設定檔 postgresql.conf（在安裝目錄的資料字目錄中），依賴你的選擇修改 datastyle 選項為 datestyle = 'European, SQL'或者 datestyle = 'iso, mdy'。需要重啟伺服器以使變動生效（reload 方法可否生效？需要測試）。

如果你想要為不同的使用者設置各自的日期樣式，你應該為調用 psql 的每個本地使用者設置環境變數。本地使用者的設置將覆蓋你設置的環境變數。

在我們演示她們怎麼工作之前，我們順便先介紹 PostgreSQL 的特別函數 cast，它允許你將一種格式的資料轉換成另一種格式。我們在這章對它做簡單的介紹，當我們在學習 SELECT 語句中進行計算的時候，但它的功能遠不止我們之前看到的轉換整數。雖然 PostgreSQL 很善於類型糾正，而且你並不經常需要類型轉換函數，但它們確實有時候很有用。以下是我們需用來進行日期轉換的方法，首先是轉換成日期：

```
cast('string' AS date)
```

另外是轉換一個包含時間的值：

```
cast('string' AS timestamp)
```

我們也將用一個小小的技巧來使對這個功能的演示更容易。基本上每次你使用 SELECT 語句的時候，你都將從一個

表取出資料。但是，你可以使用 SELECT 取根本不在表中的資料，就像這個例子：

```
bpsimple=> SELECT 'Fred';
?column?
--------------
Fred
(1 row)

bpsimple=>
```

PostgreSQL 警告我們沒有選擇任何列，但它還是樂意接受沒有表名的 SELECT 語法，並返回我們想要的字串。

我們可以使用同樣的方法來與 cast 函數一起工作，來看看 PostgreSQL 怎麼處理日期和時間，而不需要建立一個臨時表來做試驗。

嘗試：設置日期格式

我們從環境變數 PGDATESTYLE 未設置開始，所以你可以看到默認的行為，然後設置日期樣式，那麼你可以看到事情的進展。我們輸入 ISO 格式的日期，因為它是一種安全的可選方案，之後 PostgreSQL 將輸出同樣的格式。這些內容都不會有任何的歧義：

```
bpsimple=> SELECT cast('2005-02-1' AS date);
  ?column?
------------
2005-02-01
(1 row)

bpsimple=>
```

改變樣式到 US 和 SQL 格式，我們仍然以 ISO 樣式輸入日期，這也不會有歧義；仍然是 2 月 1 日，但是現在輸出看上去更傳統，但也可能導致歧義，因為美國使用 MM/DD/YYYY 樣式的輸出：

```
bpsimple=> SET datestyle TO 'US, SQL';
SET VARIABLE
bpsimple=> SELECT cast('2005-02-1' AS date);
?column?
------------
02/01/2005
(1 row)

bpsimple=>
```

現在是時候順便問問 psql，內部的 datestyle 變數被設置成什麼了：

```
bpsimple=> SHOW datestyle;
 DateStyle
-----------
 SQL, MDY
(1 row)

bpsimple=>
```

在舊版本的 PostgreSQL 中，輸出會更詳細，所以你看到的東西可能依賴於你使用的不同版本的 PostgreSQL。

現在，來試試更多的格式：

```
bpsimple=> SET datestyle TO 'European, SQL';
SET
bpsimple=> SELECT cast('2005 02 1' AS date);
    date
------------
 01/02/2005
(1 row)

bpsimple=>
```

由於輸出設置成 European，日期顯示格式為 DD/MM/YYYY 方式了。

然後回到 ISO 模式的輸入輸出：

```
bpsimple=> SET datestyle TO 'European, ISO';
SET
bpsimple=> SELECT cast('2005-02-1' AS date);
    date
------------
 2005-02-01
(1 row)

bpsimple=>
```

設置項 European 不再生效了，因為對於所有的區域，ISO 都是一樣的。

現在來看看時間。我們使用顯示時間的 timestamp 類型。

```
bpsimple=> SELECT cast('2005-02-1' AS timestamp);
        ?column?
------------------------
 2005-02-01 00:00:00

(1 row)
bpsimple=>
```

因為我們沒有指定任何小時或分鐘，所以它們默認都是 0。

讓我們試試 PostgreSQL 樣式的輸出：

```
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=>
```

這樣的輸出相當明確且友好。

解析

就像你看到的，我們可以改變日期和時間顯示的方式，包括像 01/02/2005 一樣模棱兩可的輸入字串都被翻譯過來了。

時區資訊比日期格式更簡單。如果你本地的環境變數 TZ 或者 postgresql.conf 檔中的配置項被正確設置，PostgreSQL 可以毫不費力地管理好時區。

使用日期和時間函數

現在我們已經知道日期如何工作了，我們可以看幾個有用的你可能在比較日期的時候需要的函數了：

- date_part(units required, value to use) 允許你提取出日期的某一部分，例如月份。
- Now 很簡單地獲得當前的日期和時間，它實際上和更標準的“魔法變數” current_timestamp 相等。

假設我們想要從我們的 orderinfo 表取出在九月下訂單的行，我們僅僅需要以以下方式提問：

```
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 為我們提取出了正確的行。注意日期是按照 ISO 格式顯示的。我們可以提取日期和時間戳記的以下部分：

- Year（年）
- Month（月）
- Day（日）
- Hour（小時）
- Minute（分鐘）
- Second（秒鐘）

我們也可以使用用於數字比較的<>，<=，<，>，>=和=運運算元來比較日期，以下是一個示例：

```
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=>
```

注意我們需要用 cast 來轉換我們的字串到日期，而且我們使用了明確的 ISO 樣式的日期。

第二個函數 now，簡單地給我們當前的日期和時間，它非常方便，例如，我們插入客戶通過電話下的訂單的一行，或者通過 Internet 的即時操作：

```
bpsimple=> SELECT now(), current_timestamp;
```

```
                now                  |            timestamptz
-------------------------------------+-----------------------------------
 Sat 16 Oct 13:46:05.99938 2004 BST | Sat 16 Oct 13:46:05.99938 2004 BST
(1 row)

bpsimple=>
```

我們也可以對日期進行簡單的計算。例如，計算從下訂單到發貨所間隔的天數，我們可以使用類似這樣的查詢：

```
bpsimple=> SELECT date_shipped - date_placed FROM orderinfo;
 ?column?
---------
        4
        1
       10
        7
        3
(5 rows)

bpsimple=>
```

這返回資料庫中存儲的這兩天之間所間隔的天數。

注：可以在線上文檔中找到更多關於 PostgreSQL 處理日期、時間、時區以及相關轉換函數的細節。

多個表協同工作

到現在為止，你應該已經非常瞭解怎麼從表裡選擇資料，選取我們要的列和行，以及怎樣控制資料的順序了。我們也理會到怎麼執行簡單的計算，進行類型轉換以及處理特別的日期和時間格式了。

是時候開始來到 SQL 的一個最重要的功能，實際上，應該是關聯式資料庫最重要功能：自動關聯一個表的資料到另一個表的資料。好消息是此功能也是由 SELECT 語句完成，而且你需要進一步學習的關於 SELECT 的內容和與一個表操作絕對完全一樣。

關聯兩個表

在我們研究通過 SQL 同時使用多個表之前，讓我們先回顧下第二章關於關聯表的簡述。

你應該還記得我們有一個 customer 表，用來存儲我們的客戶資訊，還有一個 orderinfo 表，存儲他們下的訂單細節資訊。這允許我們只需要存儲一次客戶資訊，而無論他們下了多少訂單。我們通用他們公用的一塊資料，也就是存儲在這兩個表中的 customer_id 連接到這兩個表。

如果我們仔細想想這樣一個圖片，我們可以想像 customer 表中的一行，它的 customer_id 沒有關聯到 orderinfo 表中的任何一行，或者通過 customer_id 關聯到出現相同值的 orderinfo 表中的一行或者多行，就像圖 4-2 說明的一樣。

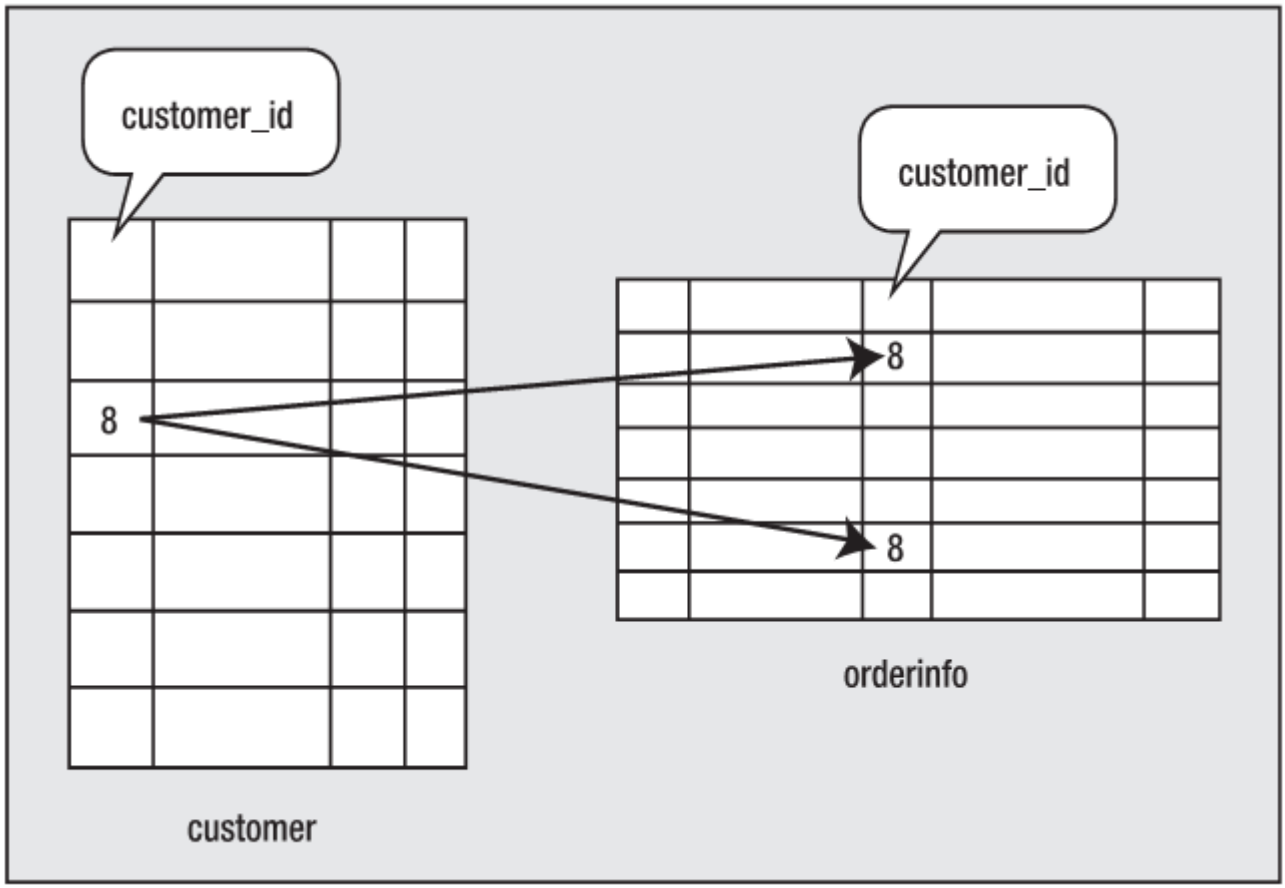


圖 4-2 customer 表和 orderinfo 表的關係

我們可以說 customer 表中的 customer_id 值為 8 的行關聯到 orderinfo 表中相同 customer_id 出現的兩行。當然，我們不需要這兩個列有相同的名字，但需要他們存儲的內容都是客戶編號，只是如果給它們不同的名字，會導致非常難以理解以及不協調。

如果我們需要找到客戶 Ann Stones 的所有訂單。邏輯上說，我們先在我們的 customer 表裡頭查找這個客戶：

```
bpsimple=> SELECT customer_id FROM customer WHERE fname = 'Ann'
AND lname = 'Stones';
customer_id
-------------
           8
(1 row)

bpsimple=>
```

現在，我們知道了 customer_id，我們可以查找這個客戶的訂單了：

```
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=>
```

這做到了，但是它需要兩步，且我們需要在這兩步間記住 customer_id。就像我們在第二章揭示的，SQL 是一種說明性語言；也就是說，告訴 SQL 你要達到什麼目標，而不需要明確定義怎麼得到結果的步驟。我們剛才所做的是按照程式上的方法使用 SQL。我們指定了兩個不連續的步驟獲得我們的結果，獲得一個客戶的訂單。有沒有簡潔的方法通過一步獲得所有結果呢？

實際上，在 SQL 中我們可以通過指出我們想知道 Ann Stones 的訂單，且這些資訊保存在通過 customer_id 這個列關聯的 customer 表和 orderinfo 表而一步做到。

做到這點的這個新一點的 SQL 語法是對於 WHERE 從句的一個擴展：

```
SELECT <列的列表> FROM <表的列表> WHERE <連接條件> AND <行選擇條件>
```

這看上去有點複雜，但它實際上很簡單。讓我們將第一個例子做得簡單點，假設我們知道客戶的編號是 8，且需要獲取訂單日期和客戶的名字。我們需要指出我們要的列，客戶的名字，下訂單的日期，這兩個表通過 customer_id 列關聯，並且我們只需要 customer_id 為 8 的行。

你會突然發現我們有一個小問題。我們怎麼告訴 SQL 我們需要使用哪一個 customer_id：是 customer 表裡的那個還是 orderinfo 表裡頭的那個？雖然我們正打算說它們（列名）是相同，但不會總是這樣，所以我們怎麼處理在不同表裡頭的列名？我們可以簡單的擴展語法指出列名：表名.列名。之後我們可以明確地描述我們資料庫裡頭的每個列了。

通常，PostgreSQL 非常寬容，如果一個列名只出現在 SELECT 語句裡頭的一個表裡，我們也不需要明確指出表名。在本例中，我們將使用 customer.fname，因為它更容易讀，尤其是你在初學 SQL 的時候，雖然只使用 fname 也能夠做到。因此，我們語句的第一部分需要需要是這樣：

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo
```

這告訴 PostgreSQL 我們需要用到的列和表。

我們現在需要指出我們的條件。我們有兩個不同的條件：customer_id 為 8 且這兩個表是通過 customer_id 相關的，或者說是需要連接的。就像我們在之前看到的關於多個條件一樣，我們通過 AND 關鍵字指出多個條件必須為真來指出我們的要求：

```
WHERE customer.customer_id = 8 AND customer.customer_id = orderinfo.customer_id;
```

注意我們需要通過使用“表名.列名”的語法告訴 SQL 明確的 customer_id，即使實際上檢查這兩個表中哪個表的 customer_id 欄位為 8 都無關緊要，因為我們還會指出它們倆都必須有相同的值。將它們組裝在一起，我們需要的語句就是以下這樣的：

```
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=>
```

這比分兩步查詢更優雅，不是嗎？也許更重要的是，通過在一條單獨的語句中指出問題，我們允許 PostgreSQL 完全優化檢索資料的方法。

嘗試：關聯表

現在我們知道原理了，讓我們回到我們原始的問題，假設我們不知道 customer_id，我們要找到 Ann Stones 的所有訂單。

我們僅僅知道一個名字，而不是客戶編號；因此，我們的 SQL 又變得稍微有點複雜了。我們必須通過名字指出客戶：

```
bpsimple=> SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo
WHERE customer.fname = 'Ann' AND customer.lname = 'Stones'
AND customer.customer_id = orderinfo.customer_id;
 fname | date_placed
-------+-------------
 Ann   | 2004-06-23
 Ann   | 2004-07-21
(2 rows)

bpsimple=>
```

解析

就像我們之前的例子中看到的，我們指出我們要的列（customer.fname，orderinfo.date_placed），牽涉到的表（customer，orderinfo），選擇條件（customer.fname = 'Ann' AND customer.lname = 'Stones'）以及這兩個表怎麼關係到一起的（customer.customer_id = orderinfo.customer_id）。

SQL 為我們做了剩下的工作。它不管客戶是沒有訂單、一條訂單或者多條訂單。SQL 精確地執行 SQL 提出查詢，提供它正確的符合條件的行，即使沒有行符合條件。

讓我們看看另一個示例。假設我們需要列出我們擁有所有的產品及其條碼。你應該記得條碼存在 barcode 表裡頭，而產品專案存在 item 表裡頭。這兩個表通過各自的 item_id 列關聯。你可能還記得我們將它們拆分成兩個表的原因是因為很多產品實際上有多個條碼。

使用我們剛學到的技能來關聯兩個表，我們知道我們需要指出我們要的列，表名和它們怎麼關聯或連接到一起。我們還決定通過產品專案的價格排序：

```
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=>
```

這看上去很合理，除了有些產品羨慕看上去出現不止一次，而且我們不記得我們倉庫裡有兩個不同的 speakers。我們也不記得倉庫裡頭有那麼多商品項目了。到底怎麼回事？

讓我們使用新學到的 SQL 技巧統計下我們倉庫裡頭有的商品數量：

```
bpsimple=> SELECT * FROM item;
```

PostgreSQL 回應的資料，顯示 11 行（更有經驗的 SQL 用戶可以使用更高效的 “SELECT count(*) FROM item;” 語句，這個函數將在第七章介紹）。

我們僅存有 11 項商品，但是我們早前的查詢卻找到 14 行，我們做錯什麼了麼？

沒有，發生這樣的問題的原因是因為一些商品相，例如 Toothbrush，每個單獨的商品可能有很多不同的條碼。PostgreSQL 簡單地為每個商品項針對每個條碼重複了它們的資訊，所以它列出了所有的條碼以及屬於它的每項商品。

你可以通過在 SELECT 語句中加入商品項編號來確認這一點，就像這樣：

```
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=>
```

注意我們明確地指出 item_id 從哪個表裡頭來，因為它同時出現在 item 表和 barcode 表裡頭。

現在很明確到底發生什麼了。如果通過 SELECT 語句返回的資料看上去有點古怪，通過在 SELECT 語句裡頭添加所有的編號列來檢查到底發生了什麼通常是個好主意。

給表賦予別名

在本章早些時候，我們看到怎麼通過使用 AS 給出更詳細的名字改變輸出的列名。如果你想要，你也有可能給表別名。這是種特別的方法，也就是你需要給一個表兩個名字，但通常它是用於節省輸入。你還會發現它通常用於 GUI 工具，它讓 SQL 生成更容易。

給表以別名，你只需簡單地在 SQL 的 FROM 從句部分的表名的後面緊跟別名。一旦你完成這個，你可以在剩下的 SQL 語句中使用別名，而不僅僅是原來的表名。

假設我們有以下簡單的 SQL：

```
SELECT lname FROM customer;
```

就像我們先去說過的，你可以通過在前面添加表名明確地指出列名，就像這樣：

```
SELECT customer.lname FROM customer;
```

如果我們給表 customer 別名 cu，我們可以用 cu 代替列的首碼，就像這樣：

```
SELECT cu.lname FROM customer cu;
```

注意我們在表名後面直接添加了一個 cu，然後又用 cu 作為列的首碼。

如果只牽涉到一個表，給表別名沒太多意思。對於多個表，它就開始有點用了。考慮我們早前的查詢：

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo WHERE
customer.fname = 'Ann' AND customer.lname = 'Stones' AND customer.customer_id =
orderinfo.customer_id;
```

是用表的別名，我們可以寫成這樣：

```
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;
```

是用別名既可以使 SQL 清晰，也可以避免在複雜的表中反復輸入冗長的表名。

關聯三個或更多的表

現在我們知道怎麼關聯兩個表到一起了，我們能否將這種想法擴展到三個或者更多的表中？是的，我們可以。SQL 是一個充滿邏輯的語言，所以如果我們可以針對 N 項做一些事情，那麼我們就可以針對 N+1 項做相同的事情。當然，你包含的表越多，PostgreSQL 需要做的事情就越多，所以有很多表的查詢可能非常慢，特別是很多表有大量行的情況下。

假設我們想關聯客戶資訊到他們訂的所有商品項的編號？

如果你看看圖 4-3 裡我們的資料庫模式，你會發現我們需要使用三個表來獲得客戶訂的所有商品項：customer，orderinfo 以及 orderline。用三個表重畫我們早前的分析圖，它看上去應該像圖 4-4 一樣。

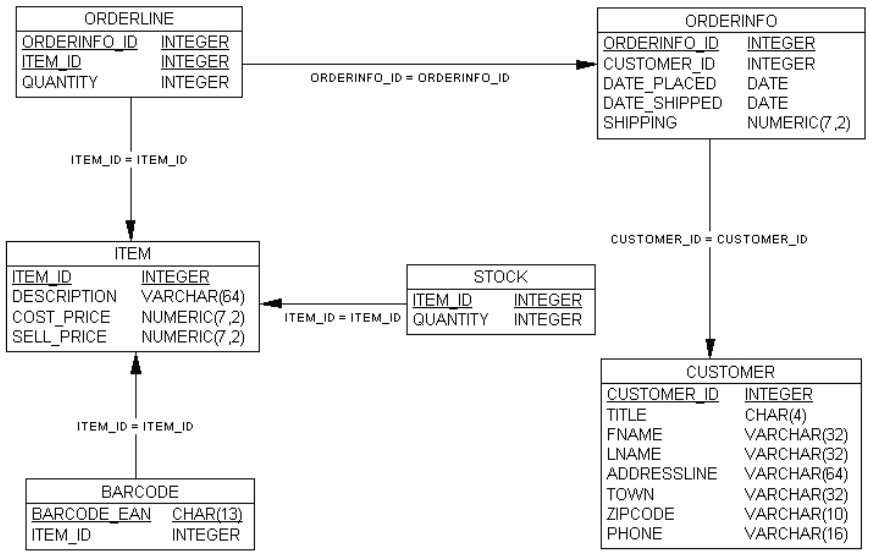


圖 4-3 資料庫模式

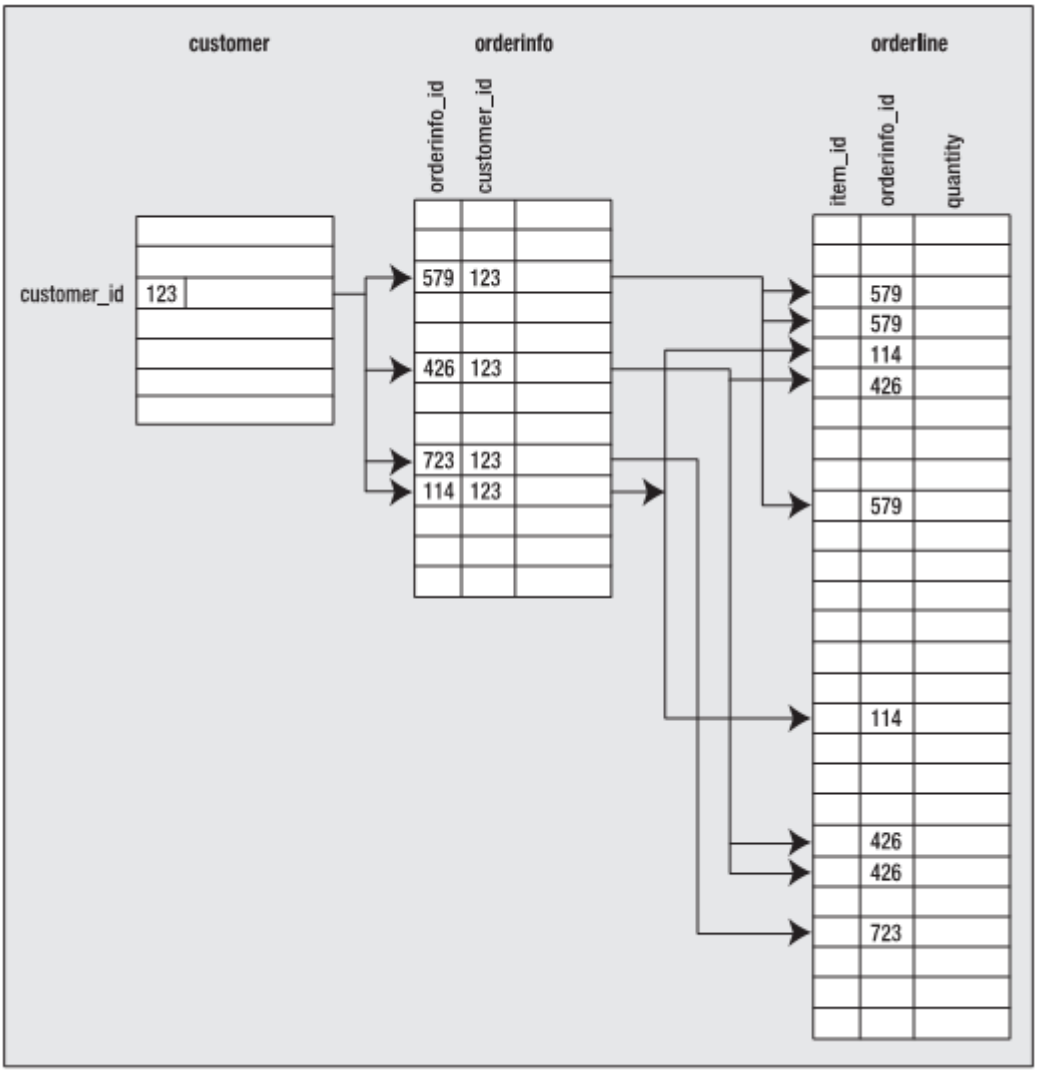


圖 4-4 三個相關表

在這裡我們可以看到客戶 123 匹配到 orderinfo 表的很多行——那些匹配到的 orderinfo 編號為 579，426,723 和 114——它們中的每一行，按順序關聯到 orderline 表的一行或者很多行。注意 customer 表和 orderline 之間沒有直接的關聯關係。我們必須使用 orderinfo 表，因為它包含用來綁定客戶和他們訂單的資訊。

嘗試：連接多個表

讓我們先建立一個三個表的連接用於找尋 Ann Stones 的訂單中的商品項，我們從我們需要的列開始：

```
SELECT customer.fname, customer.lname, orderinfo.date_placed, orderline.item_id,
orderline.quantity
```

然後我們列出相關的表：

```
FROM customer, orderinfo, orderline
```

然後我們指出 customer 表和 orderinfo 表是怎麼關聯的：

```
WHERE customer.customer_id = orderinfo.customer_id
```

我們還必須指出 orderinfo 表和 orderline 表是怎麼關聯的：

```
orderinfo.orderinfo_id = orderline.orderinfo_id
```

然後是我們的條件：

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

將他們放在一起，並分多行輸入它們（注意“bpsimple->”等待繼續輸入提示符），我們 得到這樣的：

```
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=>
```

注意字串間的空格不影響 SQL，所以我們可以添加額外的空白和換行來使 SQL 更易讀。程式 psql 只是給出一個等待繼續輸入的提示符“bpsimple->”，一直等待到我們輸入一個分號，它才嘗試執行。

你已經看到了從兩個表變成三個表是多麼容易，那麼讓我們的查詢更進一步，列出我們的客戶 Ann Stones 下的訂單中的所有商品項的描述。為了完成這個，我們需要使用一個附加表 item，來獲得商品項的描述。那麼剩下要寫的查詢也很像之前的：

```
bpsimple=> SELECT customer.fname, customer.lname, orderinfo.date_placed,
bpsimple-> item.description, orderline.quantity
bpsimple-> FROM customer, orderinfo, orderline, item
bpsimple-> WHERE
bpsimple-> customer.customer_id = orderinfo.customer_id AND
bpsimple-> orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-> orderline.item_id = item.item_id AND
bpsimple-> customer.fname = 'Ann' AND
bpsimple-> customer.lname = 'Stones';
 fname | lname  | date_placed | description | quantity
-------+--------+-------------+-------------+----------
 Ann   | Stones | 2004-06-23  | Wood Puzzle |        1
 Ann   | Stones | 2004-06-23  | Tissues     |        2
 Ann   | Stones | 2004-06-23  | Fan Large   |        2
 Ann   | Stones | 2004-06-23  | Carrier Bag |        1
 Ann   | Stones | 2004-07-21  | Wood Puzzle |        1
 Ann   | Stones | 2004-07-21  | Linux CD    |        1
(6 rows)

bpsimple=>
```

解析

你一旦知道了三個表的連接如何工作，就沒有困難擴展這個主意到更多的表。我們添加了商品項的描述到需要列出的列的清單中，添加 item 表到需要查詢的表中，還添加關於如何關聯 item 表到我們已有表的表的資訊 orderline.item_id = item.item_id。你將發現 Wood Puzzle 被列出了兩次，因為它在不同的時刻被購買了兩次。

在這個 SELECT 語句中，我們通過連接操作從我們使用的每個表中顯示了至少一列。實際上並不需要這麼做。如果我們只是需要客戶名和商品項的描述，我們可以簡單的選擇不檢索那些我們不需要的列。

一個隻檢索少量只需要的列的版本可能比早前的嘗試的版本稍微高效些且一樣夠用：

```
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';
```

為了繼續這個例子，讓我們回到我們在前面章節學到的：怎樣通過使用 DISTINCT 關鍵字移除重複資訊。

嘗試：添加額外條件

假設我們想知道 Ann Stones 買了哪些類型的商品。我們所要列出來的只是他購買的商品項，按照描述排列。我們甚至不需要列出客戶名，因為我們已經知道了（我們用它來選擇資料）。我們僅僅需要選擇 item.description，我們還需要使

用 DISTINCT 關鍵字，來確保 Wood Puzzle 只被列出來一次，雖然它被購買了好幾次：

```
bpsimple=> SELECT DISTINCT  item.description
bpsimple-> FROM customer, orderinfo, orderline, item
bpsimple-> WHERE customer.customer_id = orderinfo.customer_id
bpsimple-> AND orderinfo.orderinfo_id = orderline.orderinfo_id
bpsimple-> AND orderline.item_id = item.item_id
bpsimple-> AND customer.fname = 'Ann'
bpsimple-> AND customer.lname = 'Stones'
bpsimple-> ORDER BY item.description;
 description
-------------
 Carrier Bag
 Fan Large
 Linux CD
 Tissues
 Wood Puzzle
(5 rows)

bpsimple=>
```

解析

我們只是簡單的去處我們早前的 SQL，移除我們不再需要的列，然後在 SELECT 後面添加關鍵字來確保每行只出現一次，然後在 WHERE 從句後面加入我們的 ORDER BY 條件。

這是關於 SQL 的最偉大的功能中的一個：一旦你學了一個功能，它就可以被通過某種方法被應用於這個功能。ORDER BY 可以和跟一個單個表工作一樣同時跟很多表工作。

SQL92 的 SELECT 語法

你可能注意到了 WHERE 從句實際上由兩種稍微不同的任務。它不但指定確定哪一行我們需要檢索的條件（customer.fname = 'anna'），而且指出多個表怎麼互相關聯（customer.customer_id = orderinfo.customer_id）。

多年來這不會引起任何問題，知道 SQL 標準委員會嘗試擴展語法來說明處理 SQL 被分配的越來越多的工作。當 SQL92 標準發佈，一個新的格式的 SELECT 語法被添加用於區分這兩種稍微有點不同的用法。這種新的語法（有時候，被叫做 SQL92/99 語法，或者 ANSI 語法）被很多 SQL 資料庫延遲支援。微軟是在其 SQL Server 6.5 中最早支援這種語法的，PostgreSQL 在 7.1 版本中添加了對這種語法的支援，但是 Oracle 直到 9 才支援這種新語法。

新語法使用 JOIN … ON 語法來指定表之間的關係，而讓 WHERE 從句專注於哪些行需要選擇。新的語法將表的關聯資訊從 SELECT 語句的 WHERE 從句移動到 FROM 部分。所以語法由：

```
SELECT <列的列表> FROM <表的列表>
    WHERE <連接條件> <行選擇條件>
```

轉變為：

```
SELECT <列的列表> FROM <表> JOIN <表> ON <連接條件>
    WHERE <行選擇條件>
```

它比看起來更簡單——真的！假設我們想連接 customer 表和 orderinfo 表，他們共用 customer_id 這個鍵。作為以下語句的替代者：

```
FROM customer, orderinfo WHERE customer.customer_id = orderinfo.customer_id
```

我們應這樣寫：

```
FROM customer JOIN orderinfo ON customer.customer_id = orderinfo.customer_id
```

這看上去有點囉嗦，但是它既清晰又易於擴展，就像我們將在第七章看到外連接的時候會瞭解到。

擴展到不止兩個表也非常簡單。對於我們之前的簡單查詢：

```
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 語法中，會變成這樣：

```
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';
```

兩個版本的 SQL 語句將產生相同的結果。

但是，很多使用者看上去更喜歡較早的語法，它們仍舊有效且對於很多 SQL 語句看上去更簡潔。我們在這裡提到更新的 SQL92 版本，所以你將熟悉這個語法，但是在本書中，我們通常繼續使用舊樣式的連接，除了在第七章我們碰到外連接的時候。

摘要

本章相當的長，但我們也覆蓋了大量的內容。我們揭示了 SELECT 語句的一些細節，涵蓋了如何選擇列以及行，如果將結果排序，以及怎麼隱藏重複的資訊。我們還學了一點點關於日期類型，以及怎麼配置 PostgreSQL 關於日期轉換和顯示的行為，以及怎麼在條件陳述式裡頭使用日期。

然後我們來到 SQL 的核心：將表關聯到一起的能力。在我們第一步通過 SQL 連接兩個表後，我們發現將這擴展到三個表甚至四個表都是多麼的簡單。我們結束於重用我們之前章節獲取的知識來優化我們的四個表關聯查詢，顯示我們搜索的資訊，並移除多餘的列和重複的行。

好消息是我們現在已經見到過我們每天都會使用的 SELECT 語句，並且一旦你理解了 SELECT 語句，大部分剩下的 SQL 都是相當簡單的了。我們將在第七章回到 SELECT 語句，來看一些你偶爾會用到的高級功能，但你會發現本章已經覆蓋絕大部分現實世界中你需要使用的 SQL 了。

第五章PostgreSQL 的命令列和圖形介面工具

在先前的章節，為了開始學習，我們已經通過命令列工具 psql 建立並進行管理了一個 PostgreSQL 資料庫。商用資料庫通常都有一個和 psql 類似的命令列工具。例如，Oracle 的這種工具叫做 SQL*Plus。

因為命令列工具通常比較完全，也就是說它們包含執行所有你需要功能的方法，但他們在用戶友好度上有點不夠。在另一方面，它們對圖形顯卡的要求不高，對記憶體等也一樣。

在本章，我們將從仔細研究 psql 開始。然後，我們將覆蓋怎麼為 PostgreSQL 資料庫設置一個 ODBC 資料來源，這對一些在本章講到的工具非常重要。然後我們會碰到一些用於 PostgreSQL 資料庫的圖形介面工具。其中一些工具也可以用於資料庫管理，這是第十一章的主題。在本章，我們將專注於通常的資料庫任務。

特別是，在本章我們將測試以下工具：

- psql
- ODBC
- pgAdmin III
- phpPgAdmin
- Microsoft Access
- Microsoft Excel

psql

工具 psql 允許我們連接到資料庫，執行查詢以及管理資料庫，包括使用 SQL 命令建立資料庫，天劍新表以及輸入和更新資料。

啟動 psql

就像我們看到的，我們通過指定我們需要連接的資料庫啟動 psql。我們需要知道伺服器的主機名稱和資料庫監聽的埠（如果它不是運行在默認的 5432 埠），外加有效的用戶名和密碼來進行連接。預設的資料庫將是與本地主機當前登錄的用戶名相同的資料庫。

如果要連接到伺服器中其他名字的資料庫，我們在調用 psql 的時候需要帶上資料庫名，就像這樣：

```
$ psql -d bpsimple
```

我們可以通過設置環境變數 PGDATABASE、PGUSER、PGHOST 以及 PGPORT 分別覆蓋預設的資料庫名、用戶名、主機名稱和監聽埠。這些默認參數也可以通過 psql 的命令列參數-d、-U、-h 和-p 重新賦值。

注：我們可以通過連接到一個資料庫僅僅啟動 psql。這在建立我們第一個資料庫時產生了“雞和蛋”的問題。我們需要一個使用者和一個資料庫用於連接。我們在第三章安裝 PostgreSQL 的時候建立了一個默認用 postgres，所以我們可以使用它連接並建立新使用者和資料庫。為了建立一個資料庫，我們需要連接到一個隨 PostgreSQL 安裝而生成的特殊資料庫 template1。一旦連接到 template1 資料庫，我們可以建立一個資料庫，然後退出並重啟 psql 或者使用\c 命令重新連接到新資料庫。

當 psql 啟動後，如果在當前使用者的家目錄中存在它的開機檔案.psqlrc 且可讀，psql 將讀取它。這個檔很類似於 shell 腳本的開機檔案，並且包含很多 psql 的命令用於設置想要的行為，例如設置列印表格的格式選項以及其他選項。我們可

以通過啟動 psql 時加入-X 選項阻止 psql 讀取這個檔。

在 psql 中輸入命令

一旦運行，psql 將通過包含我們當前連接到的資料庫名並跟隨“=>”符號的提示符提示輸入命令。對於在當前資料庫擁有全部許可權的使用者，提示符被替換為“=#”。

psql 的命令分為兩種不同的類型：

- SQL 命令：我們可以輸入任何 PostgreSQL 支援的 SQL 語句給 psql，然後它將執行它。
- 內部命令：有一些 psql 命令用於執行 SQL 不直接執行的命令例如列出存在的表和執行腳本。所有的內部命令都由一條反斜線開始切不能被拆分成多行。

小提示：你可以通過執行內部命令\h 查詢一個全部支援的 SQL 命令的列表。可以使用\h <SQL 命令>獲得對某個特別命令的幫助。內部命令\?可以列出所有的內部命令。

輸入到 psql 的 SQL 命令可以被分散成多行。當這種情況發生後，psql 將轉換它的提示符為->或者-#提示還需要更多的輸入，就像這個例子：

```
$ psql -d bpsimple
...
bpsimple=# SELECT *
bpsimple-# FROM customer
bpsimple-# ;
...
$
```

我們需要通過一個分號來告訴 psql 我們已經完成一條可能被拆分成多行的長 SQL 命令。注意分號不是一條 SQL 命令必須的部分，它只是用於告訴 psql 我們已經完成輸入了。例如在上面列出的 SELECT 語句中，我們可能還需要在下一行添加一條 WHERE 從句。

我們也可以通過在啟動 psql 時打開-S 選項告訴 psql 我們將不會拆分我們的命令到多行。在這種情況下，我們不需要在我們命令的末尾添加分號。psql 的提示將變為“^>”以提示我們處在單行模式。這將為我們節省少量輸入並對執行一些 SQL 腳本有用。

使用命令歷史

在支援歷史記錄的 PostgreSQL 平臺，我們請求 psql 執行的每條命令被記錄為歷史，我們可以找回之前的命令用於運行或者編輯。使用方向鍵滾動命令歷史並編輯命令。這個功能在你沒有使用-n 命令列選項關閉前都有效（或者沒有針對你的平臺編譯這個功能）。

我們可以通過“\s”命令查看歷史，或者通過“\s <檔案名>”保存命令歷史到檔。最後執行的查詢被保存在查詢緩衝區中。我們可以通過“\p”命令查看什麼被保存在查詢緩衝區並通過“\r”命令清除它。我們可以通過“\e”命令使用外部編輯器編輯它。默認的編輯器是 vi（在 Linux 和 UNIX 中），但你可以在啟動 psql 前設置 EDITOR 環境變數指定你自己喜歡的編輯器。我們可以簡單地使用“\g”命令發送查詢緩衝區的內容到伺服器用以簡單的重新執行一條查詢。

在 psql 中執行指令檔

我們可以收集一組 psql 命令（包括 SQL 和內部命令）到一個檔並把它當做一個簡單的腳本使用。內部命令“\i”將從一個檔中讀取一組 psql 命令。

這個命令對於建立和填充表格非常有用。我們在第三章我們使用它建立了我們簡單的資料庫 bpsimple。以下是我們使用的指令檔 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)
);
```

我們依照慣例給指令檔一個.sql 的副檔名，並使用“\i”內部命令執行它們：

```
bpsimple=#\i create_tables-bpsimple.sql
CREATE TABLE
CREATE TABLE
...
bpsimple=#
```

在本例中，指令檔存放於我們啟動 psql 的目錄，但我們可以通過提供完整路徑執行任何地方的腳本。

腳本的另一個用途是做簡單的報表。如果我們想要注意資料庫的增長，我們可以將一些命令寫入一個腳本裡面並安排它每隔一段時間運行一次。為了彙報客戶數和訂單數，我們可以建立包含以下幾行命令的指令檔 report.sql 並在一個 psql 會話中執行它：

```
SELECT count(*) FROM customer;
SELECT count(*) FROM orderinfo;
```

我們也可以選擇使用“-f”命令列選項讓 psql 命令執行指令檔並退出：

```
$ psql -f report.sql bpsimple
  count
--------
     15
```

```
(1 row)
  count
--------
      5
(1 row)
$
```

如果訪問資料庫需要密碼，psql 將提示輸入一個。我們可以通過“-U”選項指定一個不同的資料庫使用者給 psql。

我們可以通過-o 命令列選項直接要求將輸出定向到一個檔，或者通過“\o”內部命令將當前會話的內容定向到一個檔或者過濾程式。例如，在一個 psql 會話中，我們可以通過輸入以下命令建立一個包含我們所有客戶的名叫 customer.txt 的文字檔：

```
bpsimple=# \o customers.txt
bpsimple=# SELECT * FROM customer;
bpsimple=# \o
```

組後的不帶檔案名參數的“\o”命令停止重定向查詢輸出並關閉輸出檔。

檢查資料庫

我們可以通過使用一些 psql 的內部命令流覽我們資料庫的結構。這個結構包括構成資料庫的表的名字和定義，可能已經建立的一些函數（存儲過程和觸發器），已經建立的用戶等等。

“\d”命令列出了所有的關係——表、序列生成器和視圖（如果在我們資料庫裡頭有）。以下是一個示例：

```
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”命令約束於只列出表。查看表 5-2“內部命令快速參考”小節瞭解更多 psql 內部命令。

psql 命令列快速參考

psql 的命令語法是：

```
psql [options] [dbname [username]]
```

psql 命令列選項以及它們的意思在表 5-1 中列出。使用以下命令可以看到 psql 完整的選項清單：

```
$ psql --help
```

表 5-1 psql 命令列選項

選項	意義
-a	從腳本中回應所有輸入
-A	取消表資料輸出的對齊模式；功能與“-P format=unaligned”相同
-c <查詢>	僅僅運行一個簡單的查詢（或者內部命令）然後退出
-d <資料庫名>	指定連接到的資料庫名（預設為$PGDATABASE 或者當前登錄用戶名）
-e	回顯發送到伺服器的查詢
-E	顯示內部命令生成的查詢語句
-f <檔案名>	執行一個檔中的查詢，然後退出
-F <字串>	指定列資料顯示分隔符號號（默認為“|”）；功能與“-P fieldsep=<字串>”相同
-h <主機>	指定資料庫伺服器主機（預設為$PGHOST 或者本地主機）
-H	設置表格輸出模式為 HTML；功能與“-P format=html”相同
--help	顯示說明，然後退出
-l	列出存在的資料庫，然後退出
-n	禁用 readline；阻止行編輯
-o <檔案名>	將查詢的輸出發送到檔案名指定檔（使用“|管道”的形式將輸出發送到一個過濾程式）
-p <埠>	指定資料庫伺服器的埠（預設為$PGPORT 或者編譯期設置的預設值，通常為 4321）
-P var[=arg]	設置列印選項 var 為 arg（查看\pset 命令）
-q	以靜默方式運行（沒有任何消息，僅有查詢的輸出）
-R <字串>	設置記錄的分隔符號號（預設為換行）；功能與“-P recordsep=<字串>”相同
-s	設置為單步執行模式（每條查詢都需要確認）
-S	設置單行模式（每行結束就認為查詢輸入結束，而不是分號）
-t	只打印行；功能與“-P tuples_only”相同
-T <文本>	設置 HTML 表格標記選項（width，border 等）；功能與“-P tableattr=<text>”相同
-U <用戶名>	指定資料庫使用者（預設為$PGUSER 或者當前登錄的用戶名）
-v name=value	設置 psql 變數 name 的值為 value
--version	顯示版本資訊然後退出，也可以用“-V”
-W	強制提示輸入密碼（如果需要密碼，會自動執行）
-x	開啟擴展表格輸出；功能與“-P expanded”相同
-X	不讀取開機檔案（~/.psqlrc）

psql 內部命令快速參考

psql 支持的內部命令在表 5-2 中列出。在很多版本的 PostgreSQL 中，這些命令有一些更易讀的長模式（例如\list 就是\l 的長模式命令）。

表 5-1 psql 命令列選項

命令	意義
\?	列出所有的 psql 內部命令
\a	在表格對齊和非對齊模式之間切換。
\c[onnect] [dbname|- [user]]	連接到新的資料庫；使用“-”作為資料庫名指連接到預設資料庫。可以 user 身份連接資料庫
\C <標題>	設置輸出表格的標題；功能與“\pset 標題”相同
\cd <目錄>	改變工作目錄
\copy …	Perform SQL COPY with data stream to the client machine.
\copyright	顯示 PostgreSQL 的使用和發佈條款
\d <表>	描述表（或者視圖、索引、序列生成器）
\d{t|i|s|v}	列出表/索引/序列生成器/視圖
\d{p|S|l}	列出訪問許可/系統表/大物件
\da	列出聚合體（aggregates）
\db	列出表空間
\dc	列出 conversions
\dC	列出 casts
\dd [對象]	列出表、類型、函數或者操作的注釋
\dD	列出 domains
\df	列出函數（自訂函數？？？）需要驗證
\dg	列出 groups
\dl	列出大對象；也可以寫作“\lo_list”
\dn	列出模式
\do	列出 operators
\dT	列出資料類型
\du	列出用戶
\e [file]	使用外部編輯器編輯當前的查詢緩衝區或者 file 指定的檔
\echo <文本>	將文本列印到標準輸出
\encoding <編碼>	設置用戶端編碼
\f <分隔符號號>	修改輸出欄位的分隔符號號
\g [檔案名]	將查詢的結果發送到後端（結果輸出到檔或者|管道）
\h [命令]	顯示 SQL 命令的說明；*表示所有命令的詳細說明
\H	開啟 HTML 模式
\i <檔案名>	從檔中讀取並執行查詢
\l	列出所有的資料庫
\lo_export，\lo_import，\lo_list，\lo_unlink	執行大物件操作
\o [檔案名]	將所有的查詢結果發送到檔或者|管道
\p	顯示當前查詢緩衝區的內容
\pset <選項>	設置表輸出選項，可設置的選項可以是以下中的一個：format，border，expanded，fieldsep，footer，null，recordsep，tuples_only，title，tableattr，pager
\q	退出 psql

\qecho <文本>	將文本寫入到查詢輸出流（參考\o 命令）
\r	重置（清空）查詢緩衝區
\s [檔案名]	列印歷史或將歷史存入檔中
\set <變數> <值>	設置內部變數
\t	只顯示行（在該模式之間切換）
\T <標記>	設置 HTML 表格的標記；功能和“\pset tableattr”相同
\timing	顯示命令執行的時間（在顯示和不顯示這兩種模式間切換）
\z	列出對表、視圖和序列生成器的訪問許可
\! [命令]	切換到 shell 或者執行一個 shell 命令

設置 ODBC

本章討論的很多工具以及後面章節討論的一些程式設計語言介面使用 ODBC 標準介面連接到 PostgreSQL。ODBC 定義了一個基於 X/Open 和 ISO/IEC 程式設計介面的通用的資料庫介面。實際上，用於開放資料庫互聯的 ODBC 標準不僅僅限於微軟的 Windows 用戶端（雖然很多人經常這麼認為）。很多語言包括 C，C++，Ada，PH{，Perl 以及 Python 編寫的程式都可以使用 ODBC。OpenOffice， Gnumeric，Microsoft Access 和 Microsoft Excel 僅僅是可以使用 ODBC 的少量示例程。

為了在一個特定的客戶機上使用 ODBC，我們同時需要針對 ODBC 介面寫的應用程式以及我們想要使用的特定資料庫的驅動程式。PostgreSQL 的 ODBC 驅動名叫 psqlodbc，我們可以將它安裝到我們的客戶機上。通常，客戶機運行在與伺服器不同的主機上，甚至它們之間可能都不相同，所以我們需要針對不同的用戶端平臺編譯不同的 ODBC。例如，我們的資料庫伺服器可能在 Linux 上但是我們的用戶端應用程式在 Windows 和 Mac OS X 上運行。

源碼和 Windows 下的二進位安裝程式可以在 psqlODBC 專案的首頁 http://gborg.postgresql.org/project/psqlodbc/中找到。

注：PostgreSQL 的標準 Windows 安裝程式也包含以版本的 ODBC 驅動程式，可以在安裝資料庫的同時安裝到 Windows 伺服器上。

在 Windows 中安裝 ODBC 驅動程式

在微軟的 Windows 中，ODBC 驅動可以通過控制台的管理工具中的資料來源（ODBC）工具配置可見，如圖 5-1 所示：

圖 5-1 ODBC 資料來源工具

該程式的驅動程式頁列出已安裝的 ODBC 驅動，如圖 5-2 所示：

圖 5-2 已安裝的 ODBC 驅動程式

我們需要執行兩步來安裝 PostgreSQL 驅動程式：

1. 從 http://gborg.postgresql.org/project/psqlodbc 下載適當版本的驅動程式。如果你的 Windows 已經包含了 Microsoft Windows Installer，那麼推薦你選擇 MSI 版本的驅動程式，因為它小很多，否則，需要下載完整的安裝包。兩個版本的驅動程式都存放在壓縮檔 psqlodbc-07_03_0200.zip 中。
2. 從下載的壓縮包裡解壓安裝程式檔，它可能是 psqlodbc.msi 或者 psqlodbc.exe。按兩下安裝程式並跟隨指引安裝 PostgreSQL 驅動程式。

在這些這兩步後，我們可以通過再次選擇 ODBC 資料來源管理器的驅動程式頁查看 PostgreSQL 驅動是否出現在列表中來確定我們成功安裝了這個驅動，就像圖 5-3 所示：

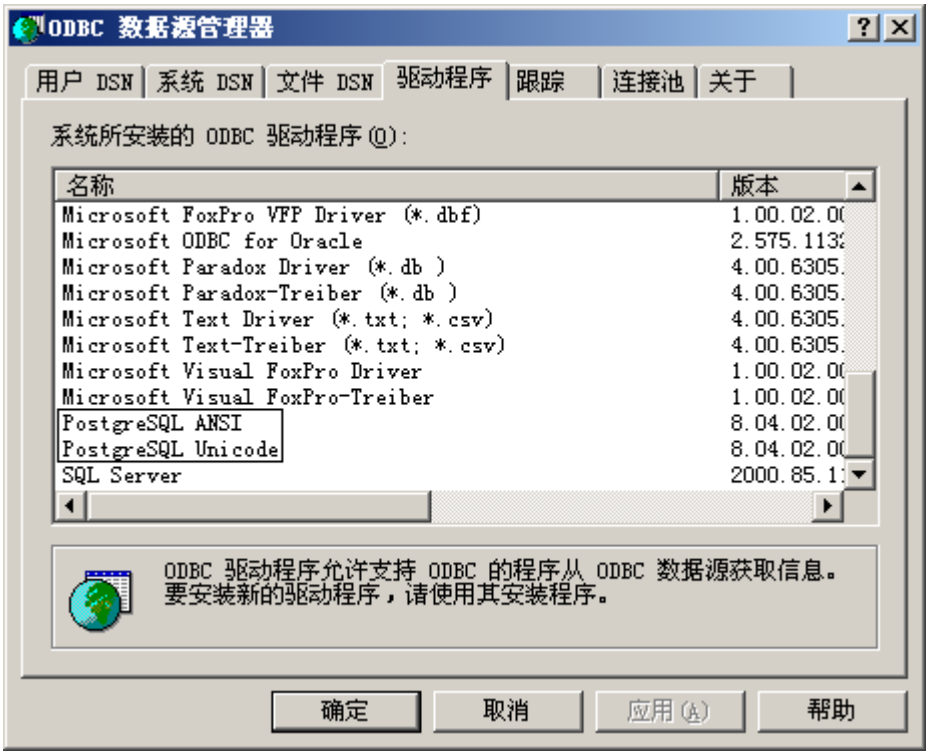


圖 5-3 PostgreSQL 驅動程式已經安裝

在 Windows 中建立一個資料來源

現在我們可以使用面向 ODBC 的應用程式連接到 PostgreSQL 資料庫了。為了讓一個指定的資料庫連接存在，我們需要通過以下的步驟建立一個資料來源：

1. 選擇 ODBC 資料來源管理器的使用者 DSN 頁面建立一個當前使用者的資料來源（如果你選擇系統 DSN，你可以建立一個所有用戶都能看到的資料來源）。
2. 點擊添加按鈕來啟動建立過程。一個用於選擇資料來源將使用哪個驅動程式的對話方塊將出現，如圖 5-4 所示。

圖 5-4 建立一個 PostgreSQL 資料來源

3. 選擇 PostgreSQL 驅動並點擊完成。
4. 我們現在擁有了一個 PostgreSQL 驅動專案等待配置。一個驅動設置視窗將出現，用於讓我們輸入這個資料來源的細節。如圖 5-5 所示，填寫資料來源的名字和描述，設置網路連接。在這裡，我們將通過伺服器的 IP 位址（如果你運行了一個完全配置的命名伺服器例如 DNS 或者 WINS，你可以使用伺服器的機器名）建立一個 ODBC 連接到我們運行在一台 Linux 伺服器上的 bpsimple 資料庫。我們還需要指定伺服器上的用戶名和密碼用於訪問我們使用的資料庫。我們還需要選擇適當的 SSL 資料連結模式。

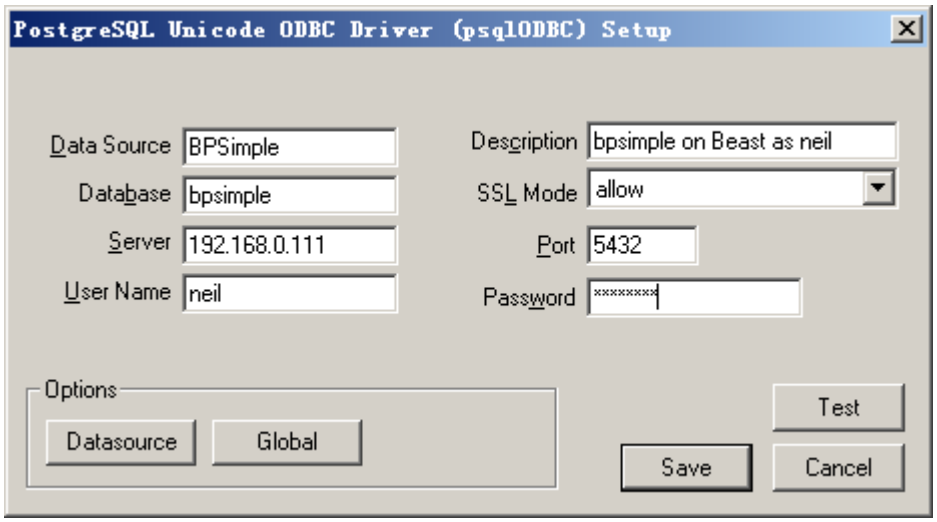


圖 5-5 配置一個 PostgreSQL 資料來源

提示：在 ODBC 驅動程式設置的對話方塊中有 Datasource 和 Global 附加選項。如果你想通過 ODBC 程式更新或者插入資料到 PostgreSQL 資料庫中，你需要配置資料來源來支援這個功能。要做到這點，點擊 Datasource 按鈕，打開高級選項對話方塊，確保對話方塊中的 Read Only 選項沒有被選中。

5. 點擊保存來完成設置。

我們已經準備好通過 ODBC 程式例如 Microsoft Access 和 Excell 訪問我們的 PostgreSQL 資料庫了，就像我們將在本章晚些時候討論的一樣。下一步，我們將看到一些開源的備選方案，我們從 pgAdmin III 開始。

在 Linux/Unix 中安裝 ODBC 驅動程式

在 Linux/Unix 中建立一個資料來源

需要自我完善

pgAdmin III

pgAdmin 是一個全功能的 PostgreSQL 資料庫圖形介面介面。它是自由軟體，由社區在 http://www.pgadmin.org 管理。從網站上看，pgAdmin 是“一個可以自由使用的強大的 PostgreSQL 資料庫管理和開發平臺”。它可以運行在 Linux，FreeBSD 和 Windows 2000/XP/Vista/7 中。運行在 Mac OS X 的版本正在開發中。

pgAdmin 提供多種功能。通過它，我們可以做到以下的事情：

- 建立和刪除表空間、資料庫、表和模式
- 通過一個查詢視窗執行 SQL
- 將 SQL 查詢的結果輸出到檔
- 備份和恢復資料庫或者單獨的表
- 配置使用者、組和許可權
- 查看、編輯和插入表的資料

讓我們看看怎麼準備和運行這樣一個多功能的工具吧。

安裝 pgAdmin III

隨著 pgAdmin III 的發佈，開發人員將這個程式的安裝變得非常簡單。前一個版本需要安裝 PostgreSQL 的 ODBC 驅動程式來提供對資料庫的訪問，但是這種依賴已經被移除了。如果你還在使用舊版本的 pgAdmin，我們強烈推薦你進行升級。

注 PostgreSQL 的標準 Windows 安裝套裝程式含一個版本的 pgAdmin III，它既可以與資料庫一同安裝在 Windows 伺服器上，也可以在客戶機上單獨安裝。

用於 Windows2000/XP，FreeBSD，Debian Linux，Slackware Linux 和其他支持 RPM 的 Linux（例如 Red Hat 和 SuSE Linux）的 pgAdmin 二進位安裝包可以從 http://www.pgadmin.org/pgadmin3/download.php 處下載。

下載你需要運行 pgAdmin III 的系統的對應安裝包並安裝它。Windows 安裝包包含一個可執行安裝程式，壓縮在一個 ZIP 格式的壓縮包內。安裝後，你應該在 Windows 的開始功能表有一個新的 pgAdmin III 功能表項目。

使用 pgAdmin III

在我們認真地使用 pgAdmin III 之前，我們需要確保我們可以在我們需要管理的資料庫中建立物件。這是因為 pgAdmin III 需要通過存儲它自己的一些物件到伺服器上擴充資料庫。要使用 pgAdmin III 執行所有的管理功能，我們需要以一個擁有資料庫全部許可權的使用者登錄——也就是說，需要一個超級使用者（superuser）。如果我們選擇的用戶沒有超級用戶許可權，我們將得到一個錯誤資訊。

提示：我們將在第十一章討論用戶和許可權。如果你的 PostgreSQL 資料庫是在 Windows 上按照默認設置安裝的，你應該有一個叫做 postgres 的使用者用於控制資料庫，你可以嘗試使用這個用戶登錄。如果你是跟隨第三章的步驟在 Linux 或者 UNIX 上安裝的，你將已經建立了一個合適的用戶，在這裡我們使用的用戶名是 neil。

我們可以通過 pgAdmin III 同時管理很多資料庫，所以我們的第一步是建立一個伺服器連接。從檔功能表選擇添加伺服器將彈出一個非常類似于早前建立 ODBC 連接的對話方塊。圖 5-6 顯示了一個到 Linux 伺服器上的 PostgreSQL 資料庫的連接。

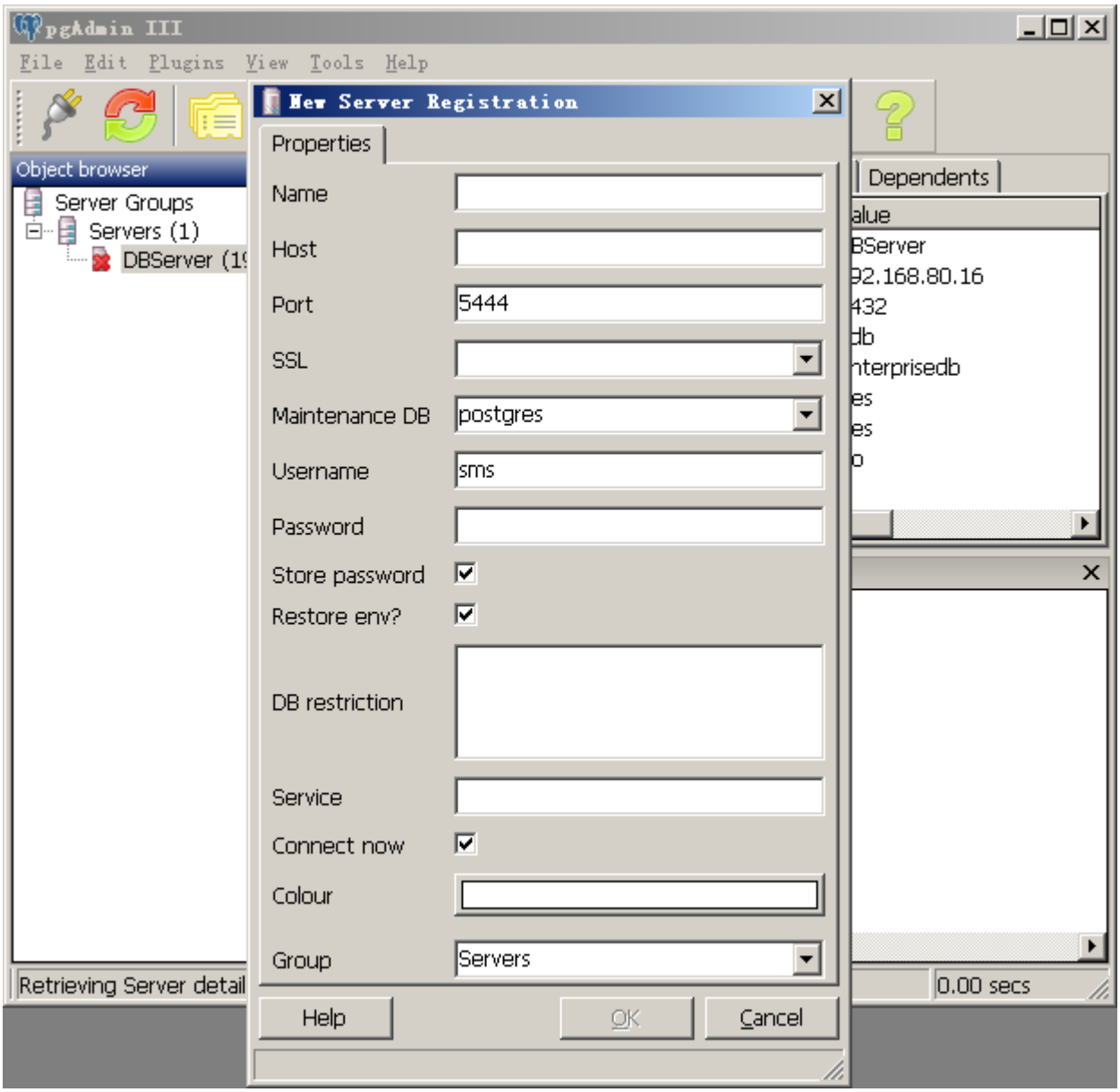


圖 5-6 添加一個伺服器連接到 pgAdmin III 中

一旦伺服器連接建立完成，我們可以連接到資料庫伺服器並流覽資料庫，表以及其他伺服器提供的物件。圖 5-7 顯示了一個 pgAdmin III 流覽 bpsimple 資料庫的表以及檢查 customer 表的 lname 的示例。

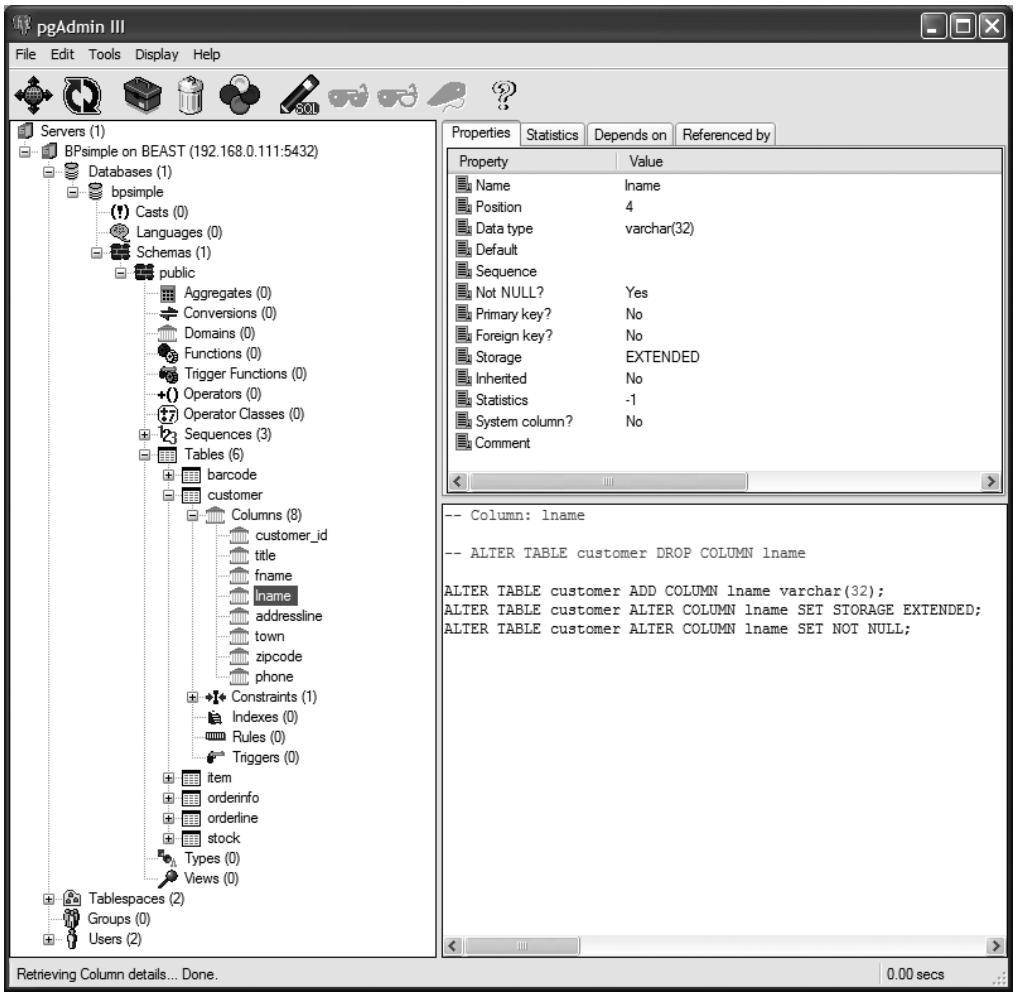


圖 5-7 使用 pgAdmin III 檢查表的屬性

pgAdmin III 的一個可能非常有用的功能是它的備份和恢復功能。它提供一個到 PostgreSQL 的 pg_dump 工具的簡單介面，我們將在第十一章詳細講解 pg_dump 工具。我們可以備份和恢復單獨的表或者整個資料庫。有一些選項控制怎麼建立備份檔案以及存放在哪裡，以及如果必要我們將使用哪種方法恢復資料庫（例如，可以使用\copy 命令或者 SQL 的 INSERT 語句）。

要打開備份對話方塊，需要在需要備份的物件（資料庫或者表）上右擊並選擇備份。圖 5-8 顯示了用於備份 bpsimple 資料庫的備份對話方塊。

圖 5-8 pgAdmin III 的備份對話方塊

我們將在在第十一章詳細覆蓋 pgAdmin 用於管理資料庫的功能。

使用 pgAdmin III 的查詢建立器，需要自己完善
好

phpPgAdmin

一個基於 web 的用於管理 PostgreSQL 資料庫的替代品是 phpPgAdmin。這是一個應用安裝在一個 web 伺服器的應用程式（使用 PHP 程式設計語言），提供基於流覽器的介面用於管理資料庫伺服器。這個項目的主頁位元於 http://phppgadmin.sourceforge.net。

通過 phpPgAdmin，我們可以對我們的資料庫執行很多工，包括：

- 管理用戶和族
- 建立表空間，資料庫和模式
- 管理表、索引、約束、觸發器、規則和許可權
- 建立視圖、序列生成器和函數
- 建立和執行報表
- 流覽表中的資料
- 執行獨立的 SQL
- 以很多格式匯出表的資料：SQL，COPY（適合使用 SQL COPY 命令的資料），XML，XHTML，逗號分隔值的檔（CSV，comma-separated values），tab 分隔值的檔以及 pg_dump
- 導入 SQL 腳本，拷貝資料、XML 檔、CSV 檔以及 tab 分隔值的檔

安裝 phpPgAdmin

安裝 phpPgAdmin 非常簡單。程式可以以多種格式的壓縮包格式下載，包括 ZIP 和壓縮的 tar 包（.tar.gz）。套裝程式需要被解壓到由一個支援 PHP 程式設計語言的 web 伺服器提供服務的目錄中。一個常見的選擇是配置了 mod_php 擴展的 Apache 網頁伺服器。更多關於 Apache 和 PHP 的資訊可以分別在 http://www.apache.org 和 http://www.php.net 找到。很多 Linux 發行版本提供了已經適當配置的 Apache 安裝包。

phpPgAdmin 唯一需要的配置是在它的設定檔 conf/conf.inc.php 裡頭設置一些變數。以下摘錄了這個檔中需要配置用於設置 phpPgAdmin 管理在另一台資料庫上的資料庫的行。

```
// 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

為了掩飾 phpPgAdmin，Apache 和 PostgreSQL 的跨平臺能力，圖 5-9 和圖 5-10 顯示一個運行在 Apple Mac 機器上的流覽器訪問一個運行在 Windows XP（位於位址 192.168.0.3）上帶有 phpPgAdmin 的 Apache 網頁伺服器，管理一台叫做 Beast 的位於 192.168.0.111 的 Linux 伺服器上的資料庫。圖 5-11 展示了了 customer 表中的資料被顯示的情況。流覽器的 URL 為 http://192.168.0.3/phpPgAdmin/index.php。

圖 5-9 phpPgAdmin 的登錄介面

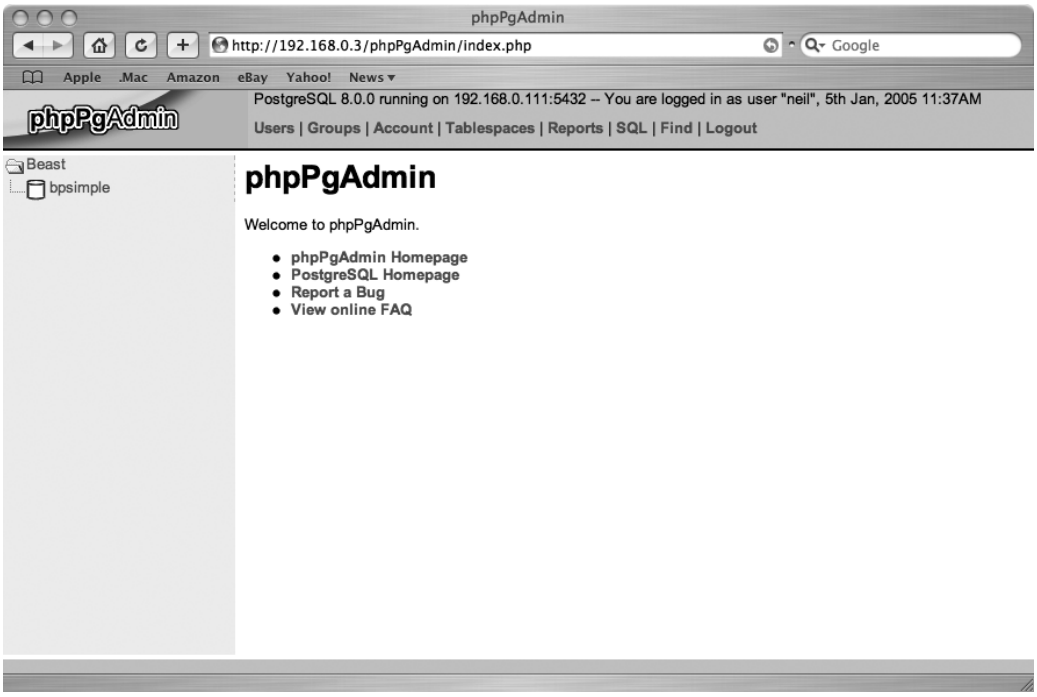


圖 5-10 phpPgAdmin 的主介面

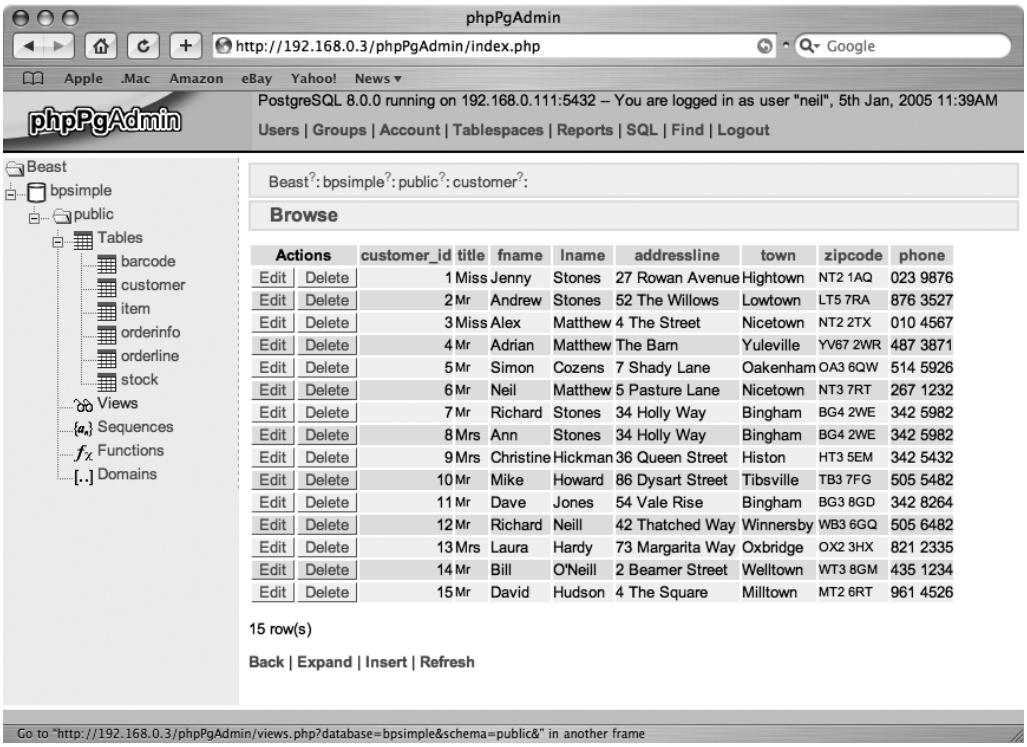


圖 5-11 通過 phpPgAdmin 流覽表的資料

phpPgAdmin 的一個可能非常有用的功能是它的資料登錄功能。如果我們有一些資料想導入到一個 PostgreSQL 的表中，phpPgAdmin 有很大的幫助。一種導入資料的方法是將資料存放入 CSV 格式的檔中。很多程式例如 Microsoft Excel 都能夠將資料匯出為這種格式。

讓我們看一個簡單的示例。假設從一個 Excel 的試算表中，我們保存了 bpsimple 資料庫的 item 表的一些行到有標題的 CSV 文件中。這意味著第一行存放著列名，之後是資料，就像這樣：

```
description,cost_price,sell_price
Wood Puzzle,15.23,21.95
Rubik Cube,7.45,11.49
Linux CD,1.99,2.49
```

我們可以先選定我們需要導入資料的表，，然後點擊 Import，然後選擇需要導入檔的類型（在本例中，為 CSV 格式）和帶入的檔案名，就像圖 5-12 所示。我們然後可以點擊 Import，然後（假設我們有許可權）新的行將加入到我們資料庫的表中。

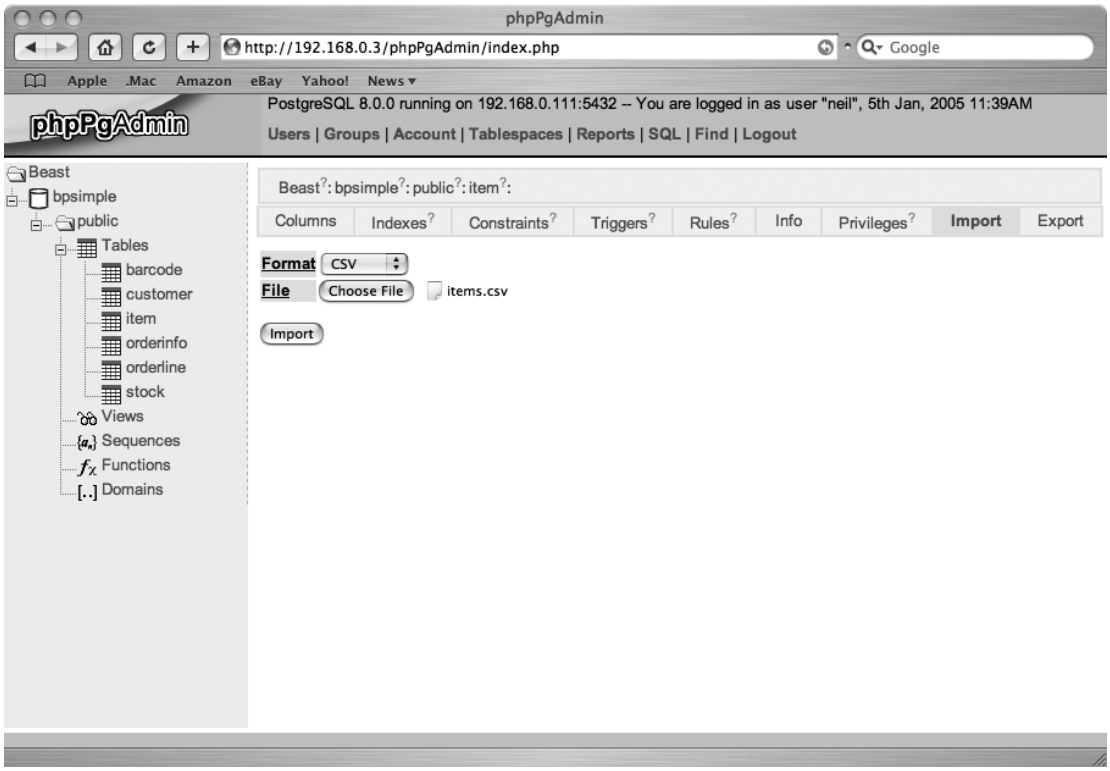


圖 5-12 使用 phpPgAdmin 導入資料

Microsoft Access

雖然咋一看這是個古怪的念頭，但我們確實可以使用 Microsoft Access 訪問 PostgreSQL。既然 Access 已經是一個資料庫系統了，為什麼我們還需要使用 PostgreSQL 來存儲資料？而且既然這裡有大量的可以與 PostgreSQL 協同工作的工具了，為什麼我們還需要使用 Microsoft Access？

首先，當開發一個基於資料庫的系統時，我們需要考慮類似於資料卷，多使用者併發，安全，健壯性和可靠性等相關需求。你可能會決定使用 PostgreSQL 因為它更符合你的安全模型，你的伺服器平臺以及你的資料增長預期。

雖然運行在 UNIX 或 Linux 伺服器上的 PostgreSQL 是你理想的資料環境，但它可能不是對於你的使用者以及它們的程式最好的或者最熟悉的環境。有可能需要允許使用者使用類似於 Access 或者其他協力廠商工具為 PostgreSQL 建立報表或者資料登記表單。因為 PostgreSQL 有 ODBC 介面，這不但可行而且非常簡單。

一旦你建立從 Access 到 PostgreSQL 的連接，你可以使用 Access 的全部功能來建立易用的 PostgreSQL 應用程式。在本章，我們將看到建立一個使用存儲在遠端的 PostgreSQL 伺服器上資料的 Access 資料庫，並基於這些資料生成一個簡單的報表（我們假設你想當熟悉建立 Access 資料庫和應用程式）。

使用連結表

Access 允許用戶通過很多方法將一個表導入到一個資料庫中，其中之一是通過叫連結表。這是一種在 Access 中表現為一個查詢的表。資料在需要的時候從其他的資料來源檢索出來，而不是被拷貝到資料庫中。這意味著當資料在外部資料庫中發生變化，這種變化也會反映到 Access 中。

在 bpsimple 資料庫中，我們有一個叫做 item 的表用來為我們銷售的每項商品記錄一個唯一標記，一個產品描述、一個成本價，以及一個售價。

作為一個示例，讓我們按照步驟建立一個簡單的 Access 資料用於更新我們示例資料庫系統裡存儲的資訊並且生成報表。

1. 在 Access 中，建立一個空資料庫。點擊視窗左邊清單中的表格按鈕，就像圖 5-13 所示。

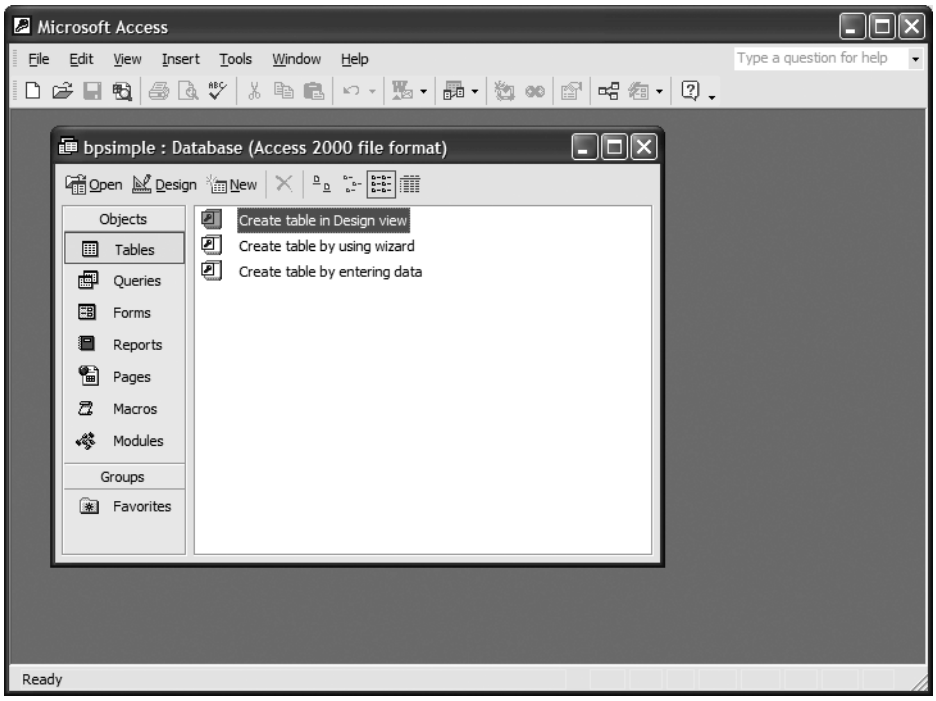


圖 5-13 建立一個空白 Access 資料庫

2. 點擊新建，會彈出新建表對話方塊，選擇連結表選項，如圖 5-14 所示。

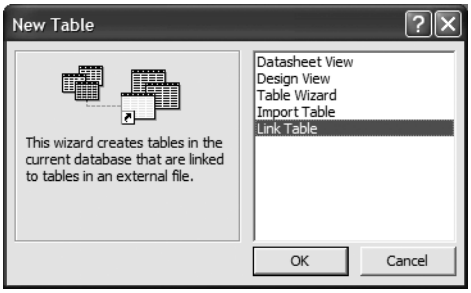


圖 5-14 添加一個連結表

3. 在之後的連接對話方塊中，選擇檔案類型為 ODBC 資料庫，之後彈出 ODBC 資料來源選擇對話方塊。選擇機器資料來源中恰當的 PostgreSQL 資料庫連接，如圖 5-15 所示。我們在本章靠前的“設置 ODBC ”小節已經建立了一個合適的資料庫連接。

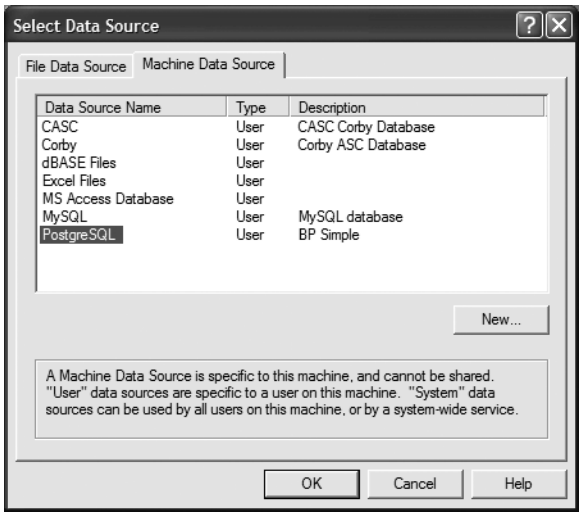


圖 5-15 選擇一個 ODBC 資料來源

4. 當連接建立後，你將會看到一個遠端資料庫中存在的表的列表。你可以選擇連接到這個列表中的一個或者多個表。在我們的示例中，我們將選擇 public.item 來連接 item 表到我們的 Access 資料庫，如圖 5-16 所示。

圖 5-16 選擇要連接的表

注：在 Access 連接到一個表之前，它需要知道這個表中的哪個列可以用於唯一標記每條記錄。也就是說，它需要知道哪個列扮演了主鍵的角色。對於這張表，item_id 列是主鍵，所以 Access 將選擇它。對於沒有定義主鍵的表，Access 將提示你選擇一個列。如果一個表有一個複合鍵，你可以選擇不止一個列。

現在我們將發現 Access 資料庫有一個新表，也叫做 item，而且我們可以流覽和編輯它，就像資料存放在 Access 中一樣。在展示在圖 5-17 中。

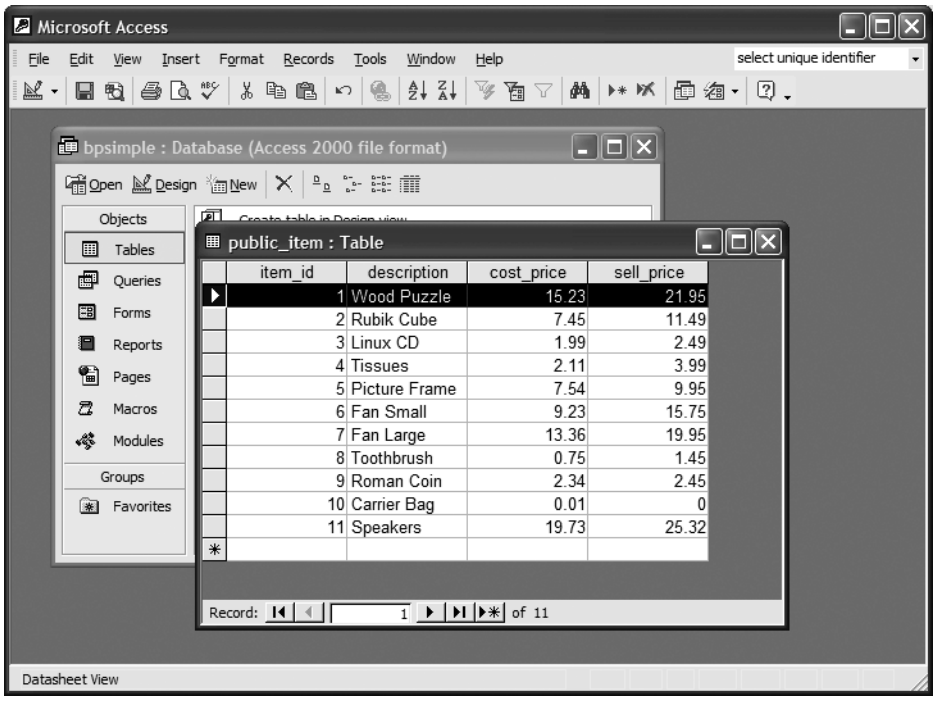


圖 5-17 流覽一個連結表

以上就是關於怎麼連結一個 PostgreSQL 資料庫中的表到 Access 中的全部內容。

注：你可能在實際操作中看到的介面和以上圖片中顯示的略有不同，這依賴於你的 Windows 和 Access 的版本。如果你在表中看到一個額外的叫做 oid 的列，這是 PostgreSQL 內部的物件標識，可以被忽略掉。為了避免 oid 列被顯示，請確定在 ODBC 資料來源配置中去掉 OID 列的選項。

輸入資料及建立報表

我們可以使用 Access 中的表流覽器檢查 PostgreSQL 資料庫中的資料以及添加更多行。圖 5-18 顯示了使用一個 Access 的資料登記表單來添加資料到 item 表中。我們可以使用 Access 的程式設計功能來建立更精細的資料資料登記應用程式來對輸入的資料執行校驗或者避免修改現有的資料。

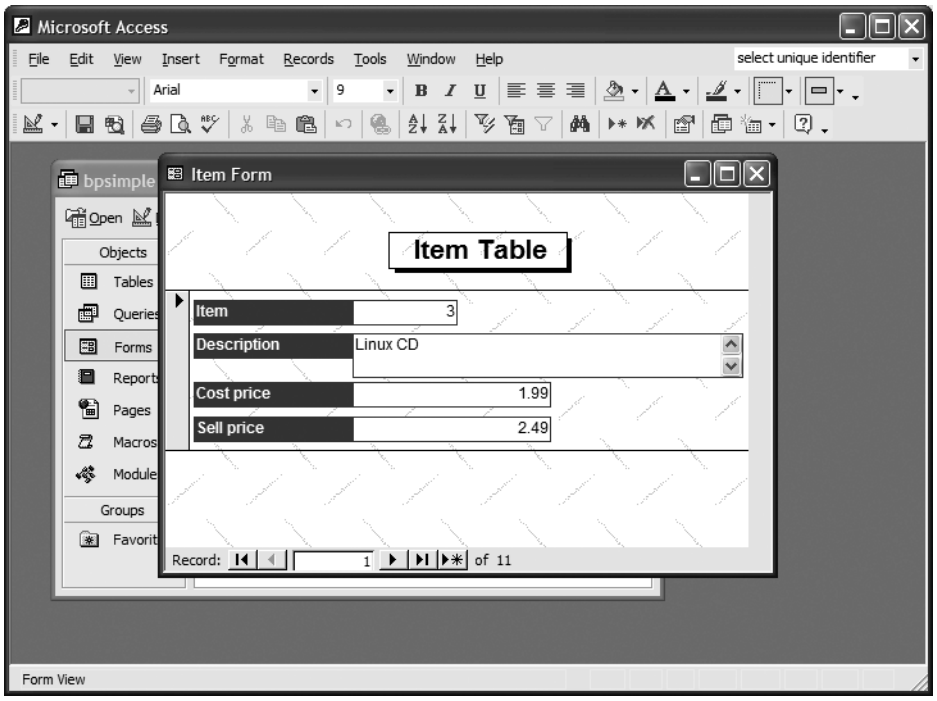


圖 5-18 一個簡單的 Access 資料登記表單

建立報表也非常簡單。使用 Access 的報表設計器針對存儲在 PostgreSQL 中的資料生成報表，就像你針對其他 Access 表做報表一樣。我們可以針對表中的資料選擇一些列來回答問題。例如，圖 5-19 顯示了一個 Access 報表，顯示了我們給 item 表中的產品添加的利潤（也就是 sell_price（售價）和 cost_price（成本）之差）部分。

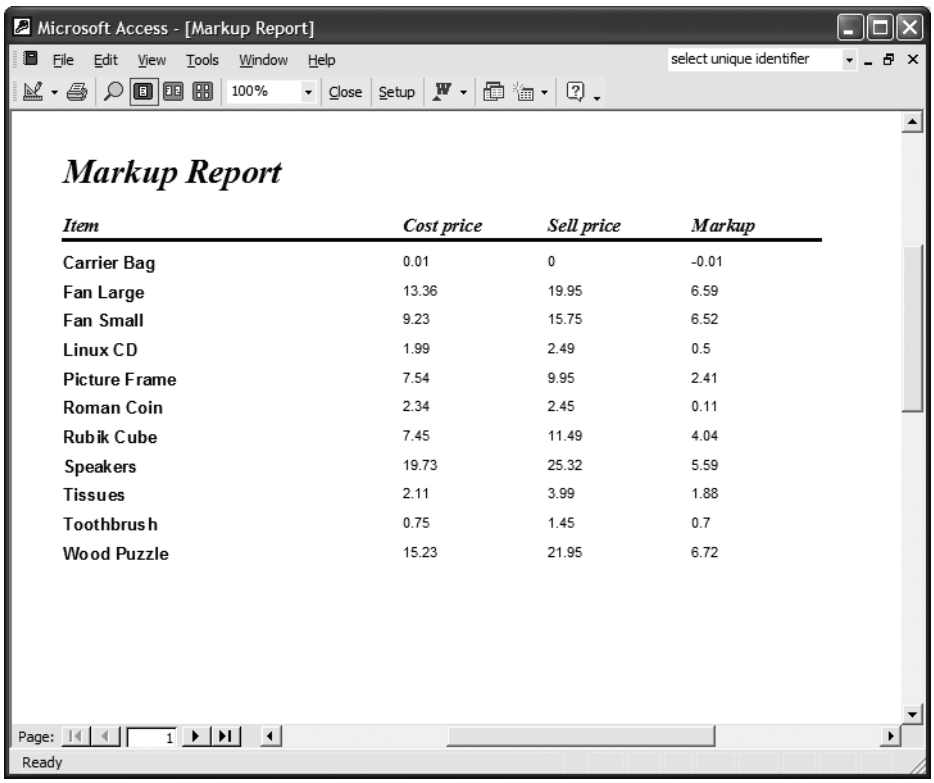


圖 5-19 一個簡單的 Access 報表

結合 Microsoft Access 和 PostgreSQL 增加了你建立資料庫應用程式的選項。PostgreSQL 的可伸縮性和可靠性以及

Microsoft Access 的易用性和你對它的熟悉程度的組合也許正是你所需要的。

Microsoft Excel

就像配合 Microsoft Access 使用一樣，你也可以利用 Microsoft Excel 來對你安裝的 PostgreSQL 添加功能。這和你使用 Access 工作的方法類似；你可以在你的試算表中包含遠端資料來源（實際上是連接到遠端資料來源）。當資料發生變動後，你可以刷新試算表讓試算表反映新資料。一旦你建立了一個基於 PostgreSQL 資料的試算表，你可以使用 Excel 的功能例如圖表來建立你的資料的圖形化圖示。

讓我們將我們的報表示例從 Access 的報表擴展到一個圖表，用以展示我們 item 表中的每個產品的利潤。

1. 我們需要告訴 Excel 試算表的一部分需要連結到一個外部資料庫表。我們從一個空白的試算表開始，選擇功能表選項來通過一個新的資料庫查詢導入外部資料，如圖 5-20 所示。

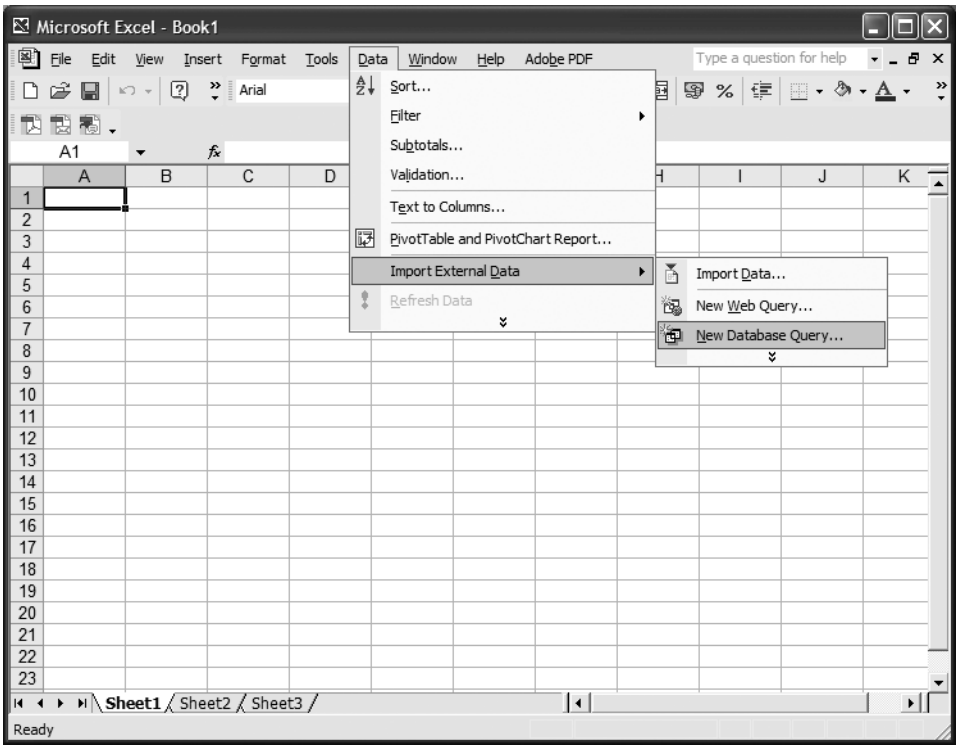


圖 5-20 導入數據到 Excel 中

2. 出現一個 ODBC 資料來源選擇對話方塊讓我們選擇我們的資料來源，就像 Access 中一樣（如圖 5-15 所示）。選擇恰當的 PostgreSQL 資料庫連接。
3. 當道資料庫的連接建立後，你可以選擇你想使用的表以及你想顯示在試算表中的列。在本例中，我們從 item 表中選擇商品項識別字，描述和兩個價格，如圖 5-21 所示。

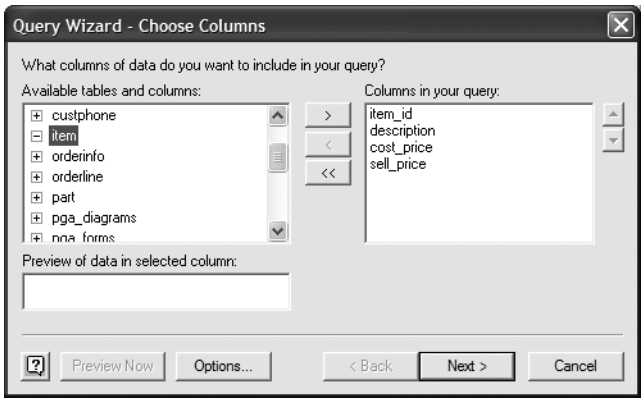


圖 5-21 選擇需要導入到 Excel 中的列

4. 如果你想限制在你的試算表中顯示的資料的行數，你可以通過下一步的對話方塊設置選擇條件。在途 5-22 中，我們選擇了那些售價高於 2 元的產品。

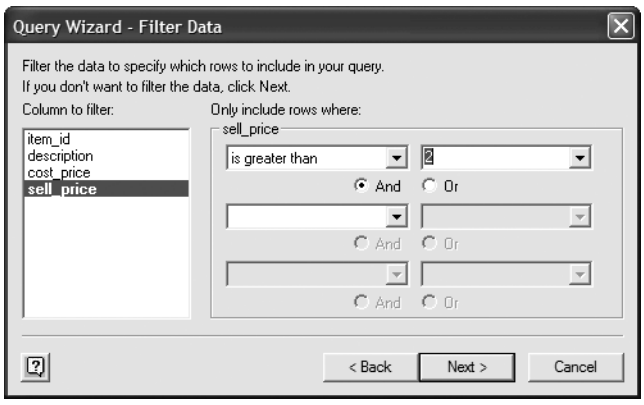


圖 5-22 約束導入資料的行數

5. 最後，你可以選擇選擇按某個或者某些列，按照想要的方向排序。在本例中，我們選擇按售價昇冪排列，如圖 5-23 所示。

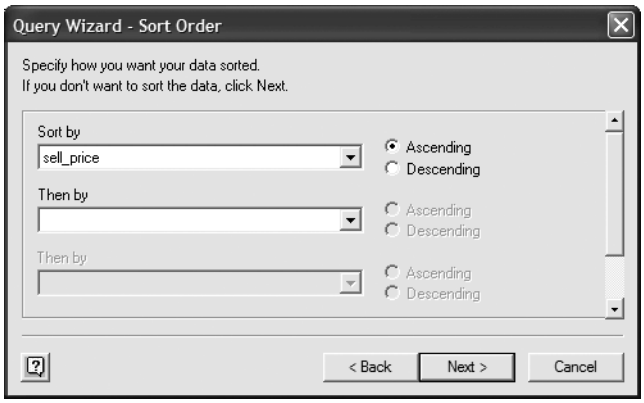


圖 5-23 選擇導入資料的排序標準

6. 在下一個對話方塊選擇返回資料到 Excel 中。
7. 現在，你可以選擇你希望導入的資料存放在你的試算表的哪個位置了。最好是將從 PostgreSQL 中的表獲得的資料存放在為它單獨準備的工作版面中。這是因為你需要為資料庫中資料的行數增長做準備。你將刷新試算表，

且需要空間用於資料的擴展。但是，在本例中，我們簡單地讓資料佔據版面的左上角，如圖 5-24 所示。

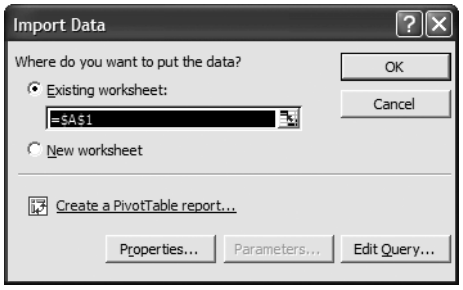
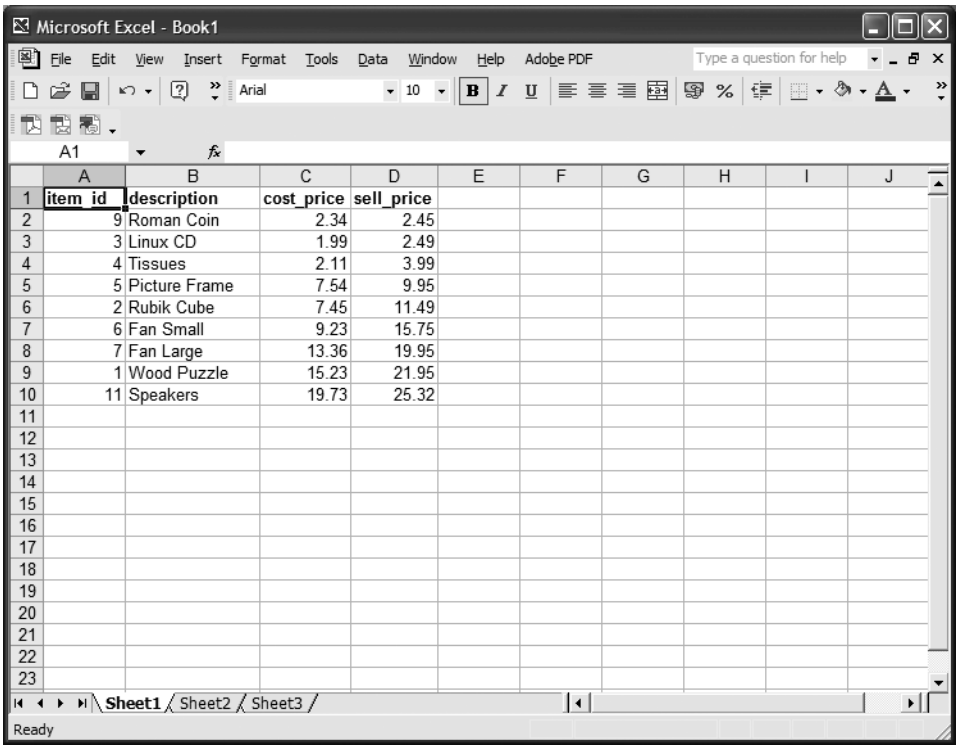


圖 5-24 選擇導入資料的位置

現在我們會發現資料出現在我們的工作版面中，如圖 5-25 所示。



	A	B	C	D
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

圖 5-25 在 Excel 中查看導入的數據

我們可以使用這個試算表針對資料執行一些計算。例如，我們可以通過使用適當的公式設置另一個列計算每個商品的銷售利潤。

警告：當資料庫中的資料發生變化，Excel 不會自動更新它獲得版本的資料。為了保證 Excel 中看到的資料是準確的，你必須刷新資料。這可以簡單的通過選擇資料功能表項目的刷新資料選項實現。

我們還可以利用 Excel 的一些功能為我們的 PostgreSQL 應用程式增值。在途 5-26 所示的示例中，我們增加了一個顯示每項產品利潤的圖示。他可以簡單地通過使用 Excel 的圖表嚮導並選擇圖表的資料來源為版面中 PostgreSQL 的資料區域來建立。當 PostgreSQL 資料庫中的資料改變了，我們刷新試算表，圖表將隨之自動改變。

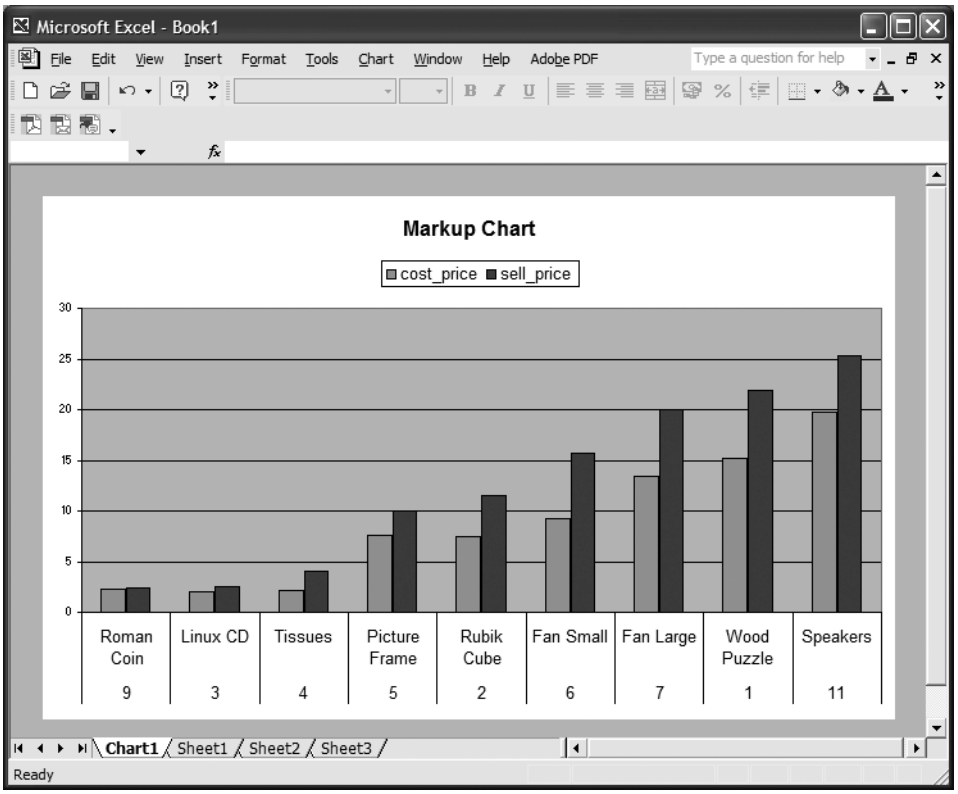


圖 5-26 使用 PostgreSQL 資料的一個 Excel 圖表

PostgreSQL 相關工具的資源

查找 PostgreSQL 相關工具的一個好去處是 pgFoundry，pgFoundry 項目的網站位元於 http://pgfoundry.org。GBorg 位於 http://gborg.postgresql.org/ 的網站也提供很多 PostgreSQL 相關的項目。有可能這項專案會被合併並通過 http://projects.postgresql.org 訪問。

你可以在 http://techdocs.postgresql.org/guides/GUITools 找到一個支援 PostgreSQL 的圖形介面工具清單。

一個正在開發中的叫做 pgmonitor 的 PostgreSQL 會話監控程式可以在 http://gborg.postgresql.org/project/pgmonitor 找到。這是一個 Tcl/TK 程式，允許你監控你的資料庫的行為。它需要在資料庫伺服器上運行但它可以在你運行 X Window 系統的 UNIX 或者類 UNIX 系統的客戶機上顯示資訊。

摘要

在本章中，我們看到了一些我們可以隨意使用，最有效發揮 PostgreSQL 功能的工具。標準的發佈帶來了命令列工具 psql，它有能力提供絕大多數我們可以用來建立和維護資料庫的操作。

資料庫管理可以在客戶機上使用非常強大的 pgAdmin III 工具或者通過網路使用基於流覽器的 phpPgAdmin 工具完成。

我們可以使用 Microsoft Office 產品，包括 Excel 和 Access，來操作和針對 PostgreSQL 資料庫中保存的資料產生報表。這允許我們結合運行在 UNIX 或者 Linux 平臺上的 PostgreSQL 系統的可伸縮性與健壯性和我們熟悉的工具。

現在我們回顧了一些 PostgreSQL 的工具，在下一章，我們將回到使用 SQL 處理 PostgreSQL 資料庫中資料的主題，致力於研究插入、修改和刪除資料。

第六章資料交互

到現在為止，我們已經弄清楚了為什麼一個關聯式資料庫，特別是 PostgreSQL 是一個非常強大的用於組織和檢索資料的工具。在前面的章節，我們研究了一些也可以用於管理 PostgreSQL 的圖形介面工具，例如 pgAdmin III。我們還學習了怎麼在 Microsoft Access 中使用 PostgreSQL 以及通過使用 Microsoft Excel 添加更多功能。當然，如果沒有在資料庫中添加資料，它們沒有一個對我們會有多大用處。在第三章，我們使用一些 SQL 腳本生成了我們的 bpsimple 資料庫。

在本章，我們將跨越基本知識，學習更多關於處理資料的內容。我們將詳細研究怎樣插入資料到 PostgreSQL 資料庫，更新已經存在於資料庫中的資料以及從資料庫刪除資料。

隨著本章的進展，我們將覆蓋以下主題：

- 通過 INSERT 添加資料到資料庫
- 插入資料到 serial 類型的列
- 插入 NULL 值
- 使用\copy 命令從文字檔載入資料
- 直接從另一個程式載入資料
- 使用 UPDATE 更新資料庫中的資料
- 使用 DELETE 從資料庫中刪除資料

添加資料到資料庫中

也許出人意料，在我們第四章看到了複雜的 SELECT 語句之後，添加資料到一個 PostgreSQL 資料庫相當容易。我們使用 INSERT 語句添加資料到 PostgreSQL 中。我們每次只可以加入資料到一個表中，而且通常我們是一次添加一行資料。

使用基本的 INSERT 語句

基本的 INSERT SQL 語句的語法非常簡單：

```
INSERT INTO 表名 VALUES (每列的值的列表);
```

我們提供一個由逗號分隔的列的值的清單，它的順序必須和表中列的順序相同。

警告：雖然這個語法因為它的簡單而非常誘人，但它也非常的危險，因為它依賴於具體的表結構，也就是列的順序，但列的順序可能由於資料庫為支援更多的資料而修改。因此，我們強烈主張你避免使用這個語法，而是用稍後的“使用更安全的 INSERT 語句”小節介紹的更安全的語法。在更安全的語法中，列名和資料值一樣被列出來了。我們在這裡介紹簡單的語法，因為你將看到它的通用用法，但是我們建議你避免這樣使用它。

嘗試：使用 INSERT 語句

讓我們添加一些新行到 customer 表中。我們需要做的第一件事是搞清楚正確的列的順序。這和它在原來的建表命令 CREATE TABLE 中的列的順序相同。如果我們無法訪問這個表的建表 SQL（不幸的是這很常見），那麼我們可以使用命令列工具 psql 通過\d 命令來查看這個表。假設我們想要在我們資料庫中查看 customer 表的定義（存在于第三章），我們

應該使用\d 命令來請求顯示它的描述。讓我們現在就開始行動。

```
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=#
```

以上顯示因為換行搞的有點不清晰，但它確實顯示了我們需要的 customer 表的列的順序。你可能會注意到 customer_id 列的描述和我們第三章的 CREATE TABLE 語句中指出的不太一樣。這是因為 PostgreSQL 就是通過這種方法實現 serial 類型的 customer_id 的。現在，我們只需要知道它是整數型欄位就好了。我們將在第八章講解 PostgreSQL 怎麼實現 serial 類型的列。

為了輸入字元資料，我們必須確保它被單引號(')包裹。數位元不需要做任何特殊處理，對於空值，我們直接寫 NULL，或者在後面更複雜的 INSERT 語句中，我們直接針對那個欄位不提供資料。

現在，我們知道了列的順序了，我們可以這樣寫我們的 INSERT 語句：

```
bpsimple=# INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',
bpsimple-# '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
INSERT 17331 1

bpsimple=#
```

你在 INSERT 之後看到的精確數字在你那裡可能不同。真正重要的是 PostgreSQL 成功插入資料了。第一個數位實際上是 PostgreSQL 內部標識號，叫做 OID，它通常是隱藏的。

注：OID（Object IDentification）是一個付給 PostgreSQL 中每一行的一個唯一數值，通常是不可見的。當你初始化資料庫，一個計數器被建立。計數器用於唯一標記每一行。在這裡，INSERT 命令被執行後，17331 就是新插入行的 OID，1 就是插入的行數。OID 不是標準 SQL 的一部分，而且在表中一般不是連續的，所以我們強烈建議你知道它的存在但不要在應用程式中使用它。從 8.0 開始，PostgreSQL 有選項來避免在表中建立 OID，所以甚至它們的存在都是不可靠的。

我們可以通過 SELECT 輕鬆檢查資料是否被正確插入，就像這樣：

```
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=#
```

換行會因為你終端視窗的大小改變，但你應該可以看到資料被正常插入了。

假設我們需要插入另一行姓 O'Rourke 的。我們怎麼處理資料中的單引號？如果在字串中出現有一個單引號，我們需要在它之前寫一個反斜線。這個反斜線叫做轉義字元，它指明它後面的字元沒有特殊的意義，就是資料的一部分。所以，要插入 O'Rourke 先生的資料，我們通過反斜線轉義他名字中的引號，像這樣：

```
INSERT INTO customer VALUES(17, 'Mr', 'Shaun', 'O\'Rourke',
    '32 Sheepy Lane', 'Milltown', 'MT9 8NQ', '746 3956');
```

然後檢查插入的資料：

```
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：在某些情況下，為了便於頁面輸出，我們需要做一些小的變動。例如，在這裡，我們將 title 縮短為 ti。這些修正非常易讀，而且我們可以確保這樣做的目標很清晰。

解析

我們使用 INSERT 語句往 customer 表裡頭添加資料，按照建表時列的順序排列插入的值。要插入數位，則直接寫數位。要插入字串，則把字串包含在單引號中。為了在插入的字串裡頭包含單引號，我們必須在單引號之前添加一個反斜線字元（\）。如果我們還需要插入一個反斜線字元，那麼我們需要寫一對反斜線，就像這樣“\\”。

假設我們需要插入另一行位元址有些特殊的資料，類似於 Midtown Street A\33。我們怎麼處理資料中已有的反斜線？我們可以將一個反斜線轉義為兩個反斜線，就像這樣：

```
INSERT INTO customer VALUES(18, 'Mr', 'Jeff', 'Baggott',
    'Midtown Street A\\33', 'Milltown', 'MT9 8NQ', '746 3956');
```

以下是插入後的樣子：

```
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=#
```

使用更安全的插入語句

當類似於剛才我們使用的 INSERT 語句在使用的時候，指定全部列或者將資料的順序完全按照表的列順序排列來插

入資料不是很方便。這增加了一個風險因素，因為我們可能會意外地寫一個錯誤的列順序資料的 INSERT 語句。這可能導致加入錯誤資料到我們資料庫中。

在上一個例子中，假設我們錯誤地交換了姓和名的列的位置。資料可以成功插入，因為這兩個列都是文本列，PostgreSQL 也無法檢測我們的錯誤。如果我們後來需要按照按姓列出我們的客戶，Gavyn 將變成客戶的姓，而不是 Smith。

低品質的資料，或者完全錯誤的資料，是資料庫的一個主要問題，所以我們會儘量執行很多預防措施來確保插入正確的資料。簡單的錯誤對於我們只有十來行資料的資料庫來說很容易發現，但是對於有成千上萬客戶資料的資料庫，發現錯誤——尤其是不尋常名字——實際上會變得非常困難。

幸運的是，有一個即容易使用又更安全的 INSERT 語句的變體，就像這樣：

```
INSERT INTO tablename( 列名的列表 ) VALUES ( 跟列的列表相對應的列的數值 );
```

在這個 INSERT 語句的變體中，我們必須列出列名以及那些列的相同順序的數值，但這些可以和我們當初建表的順序不同。使用這種變體，我們不再需要知道列在表中的順序。我們也擁有了一個將要插入到表中的優美、清晰和基本上並行的列名的清單和資料清單。

嘗試：根據列名插入資料

讓我們添加另一行到資料庫，這次指定列名，就像這樣：

```
INSERT INTO customer(customer_id, title, fname, lname, addressline, ...)
VALUES(19, 'Mrs', 'Sarah', 'Harvey', '84 Willow Way', ...)
```

我們可以分多行輸入一條 INSERT 語句，這讓它更易讀，更易於檢查列名和對應數值的順序是否相同。

讓我們執行一個示例，分幾行輸入它以易於閱讀：

```
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=#
```

可以發現將欄位元名和需要插入的值對比是多麼的簡單。我們故意交換了 fname 和 lname 列的位置，僅僅用於展示這樣是可行的。你可以使用任何你喜歡的列順序；唯一要注意的就是插入的數值要匹配列出的列。

你還會發現 psql 的提示符在後續行的變動，它保持改變，直到通過輸入分號結束命令。

> 提示：我們強烈推薦你使用帶列名的 INSERT 語句，因為指明列名會讓它更安全。

插入資料到 serial 類型的列中

到現在，是時候承認我們在 customer_id 列上犯的一個小錯誤了。到本章這個時候，我們還沒有討論到如何插入部分列的資料到一個表而忽略其他的列。使用 INSERT 語句的第二種格式，也就是帶列名的方法，我們可以做到這一點，並發現插入資料到帶有 serial 類型的列的表中是多麼重要。

你應該還記的從第二章碰到的特別特殊的資料類型 serial，它實際上是一個整數，但是可以通過自動增長來給我們一種為每一行建立唯一 ID 數字的方法。本章到現在為止，我們插入資料，都為類型為 serial 的 customer_id 自動提供了一個值。

讓我們看看到現在為止 customer 表的資料：

```
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=#
```

當然，所有看上去都正常。但是，這裡有一個小問題，由於強行給列 customer_id 列賦值，我們無意間擾亂了 PostgreSQL 的內部序列計數器。

假設我們再嘗試插入下一行，這次允許 serial 類型提供自動增量的 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=#
```

很明顯，有些東西出問題了，因為我們沒有提供任何重複的值。發生問題的原因是因為在本章早些時候，當我們為 customer_id 提供值的時候，我們跳過了 PostgreSQL 的 serial 類型列的自動分配 ID 的功能，這導致自動分配系統與表中的實際資料脫節。

警告：避免在插入資料的時候為 serial 類型的資料提供數值

脫節序列的問題相當少見，但通常是以下原因導致的：

- 你刪除並重建了表，但你沒有刪除和重建序列生成器（PostgreSQL 8.0 以及以後的版本會自動做這個事情）。
- 你混合了添加資料的方式——允許 PostgreSQL 通過 serial 類型的列自動生成值以及自己明確地為 serial 類型

的列賦值。

在這個例子中，發生了後面的情況。我們把自己推進了混亂中，我們該怎麼恢復呢？答案是我們需要幫 PostgreSQL 一把，將它內部的序號改得和實際資料一致。

訪問序列生成器

當 customer 表被建立時，customer_id 列被定義為 serial 類型。你應該注意到 PostgreSQL 在當時給我們一些消息，說建立了一個 customer_customer_id_seq 的序列生成器。而且，當我們使用\d 要求 PostgreSQL 描述這個表的時候，我們發現這個列定義得很特殊：

```
customer_id integer not null default nextval('customer_customer_id_seq'::text)
```

PostgreSQL 為這個列建立了一個特別的計數器，一個序列生成器，它可以用於產生唯一 ID。注意這個序列生成器總是被命名為<表名>_<列名>_seq。這個列的預設行為被 PostgreSQL 自動指定為函數 nextval('customer_customer_id_seq') 的結果。當我們的 INSERT 語句沒有提供這個列的資料，這個函數被 PostgreSQL 為我們自動執行。通過插入或提供資料到這個列，我們破壞了這種自動機制，因為如果提供了資料，函數將不會被調用。幸運的是，我們沒有被迫從這個表刪除所有資料並從頭開始，因為 PostgreSQL 允許我們直接控制序列生成器。

當這樣插入資料的時候，你通常會通過 currval 函數獲得序列生成器的值：

```
currval('序列生成器名');
```

PostgreSQL 將告訴你序列生成器當前的值：

```
bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
---------
      16
(1 row)

bpsimple=#
```

注：嚴格來說，currval 告訴你的是上一次調用 nextval 後返回的值，所以為了讓它工作，你需要要麼插入一個新行或者直接在當前的 psql 會話中調用一次 nextval 函數。

就像你看到的，PostgreSQL 認為最後一行的當前數字式 16，但實際上，最後一行 是 19。當我們嘗試插入資料到 customer 表，空出 customer_id 讓 PostgreSQL 處理時，它嘗試通過調用 nextval 函數為這個列提供一個值：

```
nextval('序列生成器名');
```

這個函數首先將提供的序列生成器的值加以，然後返回結果。我們可以這樣直接嘗試：

```
bpsimple=# SELECT nextval('customer_customer_id_seq');
 nextval
---------
      17
(1 row)

bpsimple=#
```

我們當然可以通過反復針對序列生成器調用 nextval 以達到需要的值，但如果數值很大，這就幫不上太多忙了。作為替代方案，我們可以使用 serval 函數：

```
setval('序列生成器名', 新的值);
```

首先，我們需要弄清楚序列生成器的值應該是多少。這可以通過選擇資料庫中那個列的最大值來得到。要達到這一點，我們將使用 max(列名)函數，它能夠簡單地告訴我們一個列的數值的最大值：

```
bpsimple=# SELECT max(customer_id) FROM customer;
 max
-----
  19
(1 row)

bpsimple=#
```

PostgreSQL 將回應它在 customer 表的 customer_id 列中找到的最大值。（我們將在下一章詳細講解 max()函數的更多細節）。現在，我們可以使用允許們設置一個序列生成器到任何值的函數 setval()設置序列生成器了。這個表中當前最大值是 19，而且序列生成器的值通常在它被使用前就增加了。因此，序列生成器通常應該和表中當前最大的值相同：

```
bpsimple=# SELECT setval('customer_customer_id_seq', 19);
 setval
--------
     19
(1 row)

bpsimple=#
```

現在，序列生成器的數字正確了，我們可以插入我們的資料，並允許 PostgreSQL 提供 serial 類型的 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 現在回到了正常狀態，它將繼續建立正確的序列值了。PostgreSQL 7.3 以及以後的版本允許你在 SELECT 語句中使用 DEFAULT 關鍵字來指出插入的時候使用一個列的預設值，這在保持序列生成器的值的一致性非常有用。當我們顯示使用 customer_id 的值來添加行時，我們可以用以下的語句來代替：

```
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');
```

在這裡，customer_id 的預設值就是序列生成器的下一個值，因為 customer_id 是一個 serial 類型的列。

我們將在第八章回到列的預設值這個主題。

插入空值

我們在第二章簡短地提到空值可以通過 INSERT 語句插入到一個列中。現在讓我們對這個做一點更細節的研究。

如果你在使用第一種格式的 INSERT 語句，也就是插入資料時按照表建立的時的列的順序，你可以簡單的在列的值的地方寫 NULL。注意你不能使用引號，因為它不是字串。你還應該記住在 SQL 中 NULL 是一個特殊的未定義值，而不同於空字串。

看看我們前一個例子：

```
INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',
```

```
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

假設我們不知道姓。表定義中定義了 fname 列為允許 NULL，所以加入不知道姓的資料時有效的。如果我們已經寫了這個：

```
INSERT INTO customer VALUES(16, 'Mr', '', 'Smith',
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

這不是我們故意的，因為我們想要先添加一個空串作為姓，也許暗示著 Mr.Smith 沒有姓。我們的意思是使用 NULL，因為我們不知道姓。

正確的 INSERT 語句應該是這樣的：

```
INSERT INTO customer VALUES(16, 'Mr', NULL, 'Smith',
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

注意 NULL 沒有被引號包裹。如果使用了引號，fname 將被設置成字串“NULL”，而不是空值 NULL。

使用第二種（更安全的）INSERT 語句，也就是必須明確指明列名的情況，插入 NULL 會更容易，因為既不用列出列名也不用給他提供值，就像這樣：

```
INSERT INTO customer(title, lname, addressline, town, zipcode, phone)
VALUES('Mr', 'Smith', '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

注意 fname 列既沒有被列出來，也沒有一個值定義給他。我們也可以選擇列出列，同時在值列表中給它一個 NULL 值。

嘗試往一個不接受 NULL 值的列中加入 NULL 值是不會成功的。如果我們嘗試加入一個沒有名字列（lname）的客戶：

```
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=#
```

注意我們沒有為 lname 提供一個值，所以 INSERT 被拒絕了，因為 customer 表定義為那個列不允許 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=#
```

我們將在第八章我們將講解我們怎麼通過指定一個列的預設值，定義一個明確的預設值用於在插入資料時沒有給出數值的列中。

使用\copy 命令

雖然 INSERT 是用於添加資料到資料庫的標準 SQL 方法，但它不總是最合適的。假設我們有大量的行需要加入到資料庫，但是實際上的資料，可能是在一個試算表裡。將資料插入資料庫的一種方法先是使用試算表的匯出功能，所以我們可以可能將試算表匯出為 CSV（逗號分隔值）文件。然後我們可以使用類似於 Eacs 的文字編輯器，或者至少支援宏的其他工具，將所有的資料轉換成 INSERT 語句。

考慮以下的資料：

```
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
```

我們可能轉換它為一系列的 INSERT 語句，所以它看上去類似於這樣：

```
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');
```

然後將它保存為有.sql 副檔名的文字檔。

我們然後可以在 psql 中使用\i 命令執行這個檔中的語句。這就是 pop_customer.sql 檔工作的方法（我們在第三章使用這種方法構造我們的資料庫）。注意這裡我們需要 PostgreSQL 生成唯一鍵 customer_id 的值。

但是，這樣做不是非常方便。如果資料可以通過一種更通用的方法在扁平檔和資料庫間移動會更好。在 PostgreSQL 中有好幾種方法可以這麼做。頗為難理解的是，他們都叫做 copy 命令。有一個 PostgreSQL 的命令叫做 COPY，它可以通過扁平檔存儲和還原資料，但它限於資料庫管理員使用，因為檔必須在伺服器上操作，而普通用戶可能沒有存取權限。另一個更有用的是通用的\copy 命令，它基本上實現了 COPY 的全部功能，而且可以被任何人使用，而且資料是在客戶機上讀寫的。所以基於 SQL 的 COPY 命令基本上是多餘的。

注：SQL 裡頭的 COPY 命令有一個優點：它明顯比\copy 命令快，因為它直接通過伺服器進程執行。\copy 命令是在客戶進程中執行，有可能需要通過網路傳輸所有資料。而且 COPY 在發生錯誤的時候會更可靠。除非你有大量的資料，否則區別不會太明顯。

\copy 命令有以下語法用於導入資料：

```
\copy 表名 FROM '檔案名'
    [USING DELIMITERS '作為分隔符號號的單個字元']
    [WITH NULL AS '代表 NULL 的字串']
```

它看上去有點複雜，但它很容易使用。方括號“[]”中的部分是可選的，所以你只有在需要時使用它們。但是，注意檔案名需要用單引號括起來。

選項[USING DELIMITERS '作為分隔符號號的單個字元']允許你指定輸入檔中的每個列是怎麼分隔的。預設情況下，輸入檔的被假設為使用定位字元（tab）分隔列的。在我們的例子中，我們假設我們從一個從試算表裡頭匯出的 CSV 檔開始。通常，CSV 格式不是一個好的選擇因為逗號可能出現在資料中，位址資料特別傾向於使用逗號字元。不幸的是，試算表通產不提供合理的除了 CSV 檔之外的候選方案，所以你可能需要用好你擁有的。我們可以給出一個替代方案，使用管道符“|”，它經常被用作結束字元，因為它極少出現在使用者資料中。

選項[WITH NULL AS '代表 NULL 的字串']允許你制定一個可以被翻譯為 NULL 的字串。預設情況下，假設為\N。注意在\copy 命令中，你必須用單引號包裹這個字串，因為這才能告訴 PostgreSQL 這是一個字串，雖然在實際資料中不會有引號。所以如果你希望使用 NOTHING 作為空值載入到資料庫，你應該使用選項“WITH NULL AS 'NOTHING'”。

那麼例如如果我們不知道 Mr.Hudson 的名，資料就應該看起來是這樣：

```
15,Mr,NOTHING,Hudson,4 The Square, Milltown,MT2 6RT,961 4526
```

當直接插入資料時，當心資料“乾淨”非常重要。你需要確保不丟失列，所有的引號字元都通過反斜線被正確轉義，沒有二進位字元存在等。PostgreSQL 會為你捕獲大部分這種錯誤，僅載入有效資料，但是整理成千上萬行基本上被完全載入的資料是一項緩慢、不可靠以及吃力不討好的工作。因此在使用\copy 命令嘗試批量載入資料前儘量清理資料是非常值得的。

嘗試：使用\copy 載入資料

讓我們建立一個額外客戶資料檔案 cust.txt，就像這樣：

```
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
```

你可以使用任何文字編輯器建立這個簡單的 cust.txt 文件。為了簡化操作，在這裡不需要處理空值，所以我們僅僅需要指定逗號為列分隔符號號。執行以下命令載入這份資料：

```
\copy customer from 'cust.txt' using delimiters ','
```

注意這裡在命令尾部沒有分號“;”，因為他是一個由 psql 直接執行的命令，而不是 SQL。Psql 通過非常簡短地回應“\.”，告訴我們所有操作成功。

然後執行以下語句：

```
SELECT * FROM customer;
```

我們將看到額外的行被添加了。

但是，這裡還有一個潛在的小問題。還記得序號可能不同步的問題嗎？不幸的是，使用\copy 載入資料時這種問題可能發生的一種情況。讓我們檢查下在序列數字上發生了什麼：

```
bpsimple=# SELECT max(customer_id) FROM customer;
 max
-----
  23
(1 row)

bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
---------
      21
(1 row)

bpsimple=#
```

噢！！！存儲在 customer_id 中的最大值現在是 23，所以下一個需要分配的 ID 應該是 24，但是序列生成器將會為下一個數值分配 22。不用擔心，這很容易修復：

```
bpsimple=# SELECT setval('customer_customer_id_seq', 23);
 setval
--------
     23
(1 row)
```

```
bpsimple=#
```

解析

我們使用\copy 命令直接載入從試算表裡匯出的 CSV 格式的資料到我們的 customer 表。我們接下來需要修正用來為表中 serial 類型的 customer_id 列生成 customer_id 數字的序列生成器的數值，這個工作量明顯比我們轉換 CSV 格式的資料到一系列的 INSERT 語句要輕鬆得多。

直接從另一個程式載入資料

如果資料已經存在於桌面資料庫中，例如 Microsoft Access，那麼有一種更簡單的方法載入資料到 PostgreSQL 中。我們可以通過 ODBC 簡單地附加 PostgreSQL 的表到 Access 資料庫中，然後插入資料到 PostgreSQL 的表中。

通常，當你這麼做的時候，你會發現你存在的資料不完全是你需要的，或者需要在插入到最終目標表的時候需要做一些加工。

即使資料的格式正確，直接將它插入資料庫通常也不是一個好主意，相反的最好是先把資料載入到一個載入表，然後將資料從載入表傳送到現實表中。使用一個中間的載入表是現實世界中應用程式將資料插入資料庫的一個通用的做法，尤其是在原始資料的品質不穩定的情況下。資料先被載入進資料庫的一個處理表，檢查，如果有必要則修正，然後再將資料移入最終的表中。

通常，你需要寫一個客戶程式或者存儲過程用於檢查和修正資料，這將在第十章涉及。一旦資料準備好可以載入到最終表，我們可以使用另一個 INSERT 語句的變種，它允許我們在表和表之間移動資料，通過一條語句傳輸很多行。這是唯一的情況一條 INSERT 語句可以影響很多行。這就是 INSERT INTO 語句。

將資料從一個表插入另一個表的語法為：

```
INSERT INTO 表名(列名的列表) SELECT 普通的查詢內容
```

嘗試：在表與表之間載入資料

假設我們有一個載入表 tcust 保存有一些額外使用者資訊需要載入到我們的 customer 表中。我們載入表的定義資訊如下：

```
CREATE TABLE tcust
(
    title                       char(4)                 ,
    fname                       varchar(32)             ,
    lname                       varchar(32)             ,
    addressline                 varchar(64)             ,
    town                        varchar(32)             ,
    zipcode                     char(10)                ,
    phone                       varchar(16)
);
```

注意這個表沒有任何的主鍵或者約束。為了交叉載入資料時載入資料到這個表更容易，我們通常這麼做。移除約束讓這麼做更簡單。還要注意除了 customer_id 這個順序數位的列以外的所有列都有了，因為 PostgreSQL 在載入資料的時

候會為我們建立它。

假設我們已經載入一些資料到 tcust 表中（通過 ODBC，\copy 或其他方法），驗證並修正了。然後，SELECT 出來的內容就像這樣：

```
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們沒法找到 Mr.Bradley 的電話號碼。這是也不是一個問題。現在，讓我們做個決定，我們不想載入這一行，但我們想載入所有其他的客戶。在現實世界的情形中，我們可能嘗試載入成千上萬的新客戶，而且很可能我們需要載入他們中的一些被驗證或修正的分組。

這條 INSERT 語句的第一部分很容易寫。我們將使用 INSERT 語句的完整語法，僅指定我們需要的列。這也通常是合理的選擇：

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

注意我們沒有指定載入 customer_id 列。你應該還記得這個欄位如果我們不處理他，PostgreSQL 將自動為我們建立它的數值，這是一個比較安全的方法建立 serial 類型的資料。

我們現在需要寫這個語句的 SELECT 部分了，它將為這個 INSERT 語句提供內容。記住我們還不想插入 Mr.Bradley 的資訊，因為他的電弧號碼是 NULL，我們正在嘗試找到他的號碼。如果我們想要，我們可以載入 Mr.Bradley 的資料，因為電話號碼欄位可以接收 NULL 值。我們現在做的是針對資料使用一個比現實世界中需要的低級資料庫規則更嚴格的規則。我們寫一個這樣的 SELECT 語句：

```
SELECT title, fname, lname, addressline, town, zipcode, phone FROM tcust
WHERE phone IS NOT NULL;
```

當然，這是一個完美有效的語句。讓我們測試一下：

```
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=#
```

這看上去正確。它找到我們需要的行，而且列的順序和 INSERT 語句中的順序也一樣。所以我們現在可以將這兩條語句放到一起執行，就這樣：

```
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 告訴我們插入了兩條記錄。現在，為了進一步的謹慎，讓我們把 customer 表中的行取出來，僅僅是為了絕對確保它們被正確載入：

```
bpsimple=# SELECT customer_id, fname, lname, addressline FROM customer WHERE
bpsimple-# town = 'Lincoln';
 customer_id   fname | lname  |  addressline
-------------+-------+--------+----------------
          19 | Sarah | Harvey | 84 Willow Way
          20 | Steve | Clarke | 14 Satview way
          24 | Brian | Waters | 21 Troon Rise
          25 | Kevin | Carney | 43 Glen Way
(4 rows)

bpsimple=#
```

我們實際上獲得了超過兩行的資料，因為我們已經有來自 Lincoln 的客戶了。可是我們可以看到，我們資料被正確插入了，且 customer_id 的值也被建立了。

現在我們已經從 tcust 表載入了一些資料到 customer 表了，我們通常可以刪掉 tcust 表中的那些資料了。為了示例的用途，我們暫時保留那些資料，將在後面的示例中刪除它們。

解析

我們指出了需要載入到 customer 表中的列，然後從 tcust 表中使用相同的順序選擇對應的資料集。我們沒有指定需要載入 customer_id 列，所以 PostgreSQL 使用它的序列生成器為我們生成唯一 ID。

一個變通的方法，你會發現更容易，特別是有大量資料需要載入的時候，做法是在臨時表加一個列，也許是叫做 isvalid 的類型為 boolean 的列。然後你將所有資料載入到臨時表，然後使用我們在本章下一小節將正式講解的 UPDATE 語句設置所有的 isvalid 為 false：

```
UPDATE tcust SET isvalid = false;
```

我們沒有指定一個 WHERE 從句；因此，所有的及路的 isvalid 列都被設置成 false。我們繼續在資料上做修改，在需要的地方修改。當我們確定一行已經修正和完成，我們就設置 isvalid 列為 true。我們之後可以載入正確的資料，只選擇那些 isvalid 為 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;
```

一旦完成了資料載入，我們可以使用以下語句將它們從 tcust 表刪除：

```
DELETE FROM tcust WHERE isvalid = true;
```

然後繼續處理 tcust 表裡頭剩下的資料。（我們將在本章接近最後的時候討論 DELETE 語句）。

修改資料庫中的資料

現在我們知道怎麼通過 INSERT 語句將資料插入資料庫，以及怎麼使用 SELECT 將它檢索出來。不幸的是，資料不宜一直保持不變。人們可能移動到不同的位址，改變電話號碼等。我們需要一個方法來修改資料庫中的資料。在 PostgreSQL 中，就像所有的基於 SQL 的資料庫一樣，這通過 UPDATE 語句實現。

使用 UPDATE 語句

UPDATE 語句非常簡單。它的語法是這樣的：

```
UPDATE 表名 SET 列名 = 值  WHERE 條件
```

如果我們想一次性修改很多列，我們只需要簡單的用逗號將賦值資訊分隔開，像這樣：

```
UPDATE customer SET town = 'Leicester', zipcode = 'LE4 2WQ' WHERE 一些條件
```

如果我們願意，我們想同時修改多少列就修改多少列，只要讓每個列出現一次。你會注意到你只能使用一個表名。這是因為 SQL 語法的原因。即使在一些情況下你需要修改兩個分開的但相關的表，你需要寫兩個分開的 UPDATE 語句。你可以將這兩條 UPDATE 語句放在一個事務中來確保要麼兩個操作都執行要麼兩個操作都沒被執行。我們將在第九章近距離地討論事務。

嘗試：使用 UPDATE 語句

假設我們現在查出了 Mr.Bradley 的電話號碼，我們想更新我們的線上表 customer。UPDATE 語句的第一部分很容易：

```
UPDATE tcust SET phone = '352 3442'
```

現在我們需要指出需要修改的行，這也很簡單：

```
WHERE fname = 'Peter' and lname = 'Bradley';
```

對於 UPDATE 語句，最好是經常檢查 WHERE 從句。讓我們就開始做吧：

```
bpsimple=# SELECT fname, lname, phone FROM tcust
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
 fname |  lname  | phone
-------+---------+-------
 Peter | Bradley |
(1 row)

bpsimple=#
```

我們可以看到我們需要修改的唯一一行被選中了，所以我們繼續前進以，將兩部分語句放到一起並執行它：

```
bpsimple=# UPDATE tcust SET phone = '352 3442'
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
UPDATE 1
bpsimple=#
```

PostgreSQL 告訴我們一行被修改了。如果我們想要，我們可以重新執行我們的 SELECT 語句查看我們的結果是否正確。

解析

我們分兩步建立我們的 UPDATE 語句。首先，我們 UPDATE 語句的修改列數值的部分，然後我們寫出 WHERE 從句來指出哪些行需要修改。在測試 WHERE 從句後，我們執行 UPDATE 語句，它修改了想要改變的行。

為什麼我們這麼小心測試WHERE從句並警告不要執行UPDATE語句的第一部分？答案是因為沒有WHERE從句的 UPDATE 語句是絕對有效的。預設情況下，UPDATE 語句會更新表中所有的行，一般情況下沒有誰想這樣。它也可能相

當難糾正（如果 WHERE 從句有錯）。

tcust 是一個臨時試驗資料，所以讓我們使用它測試下一個沒有 WHERE 從句的 UPDATE 語句：

```
bpsimple=# UPDATE tcust SET phone = '999 9999';
UPDATE 3
bpsimple=#
```

注意 psql 會告訴我們有三行資料被更新。現在看看我們的資料：

```
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=#
```

這當然不是我們想要的！

警告：永遠在執行 UPDATE 之前測試 WHERE 從句。WHERE 從句中一個簡單的錯誤會導致表中的很多甚至全部行被更新為相同的值。

如果你想要更新很多行，與其取出所有資料，不然簡單的使用 count(*)檢查有多少行匹配你的要求，我們將在下一章講解 count(*)的細節。現在，你所需要知道的就是用 count(*)替換 SELECT 語句裡的列名就會知道匹配我們要求的行有多少，而不是返回所有的資料行。實際上，這基本上就是 count(*)語句的全部內容了，但它在實際工作變得非常有用。以下是一個使用 WHERE 從句的 SELECT 語句用於檢查有多少資料符合我們要求的例子：

```
bpsimple=# SELECT count(*) from tcust
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
 count
-------
     1
(1 row)

bpsimple=#
```

這告訴我們 WHERE 從句充分限制到只有一行。當然，對於不同的資料，即使提供 fname 和 lname 也可能不能充分唯一標記一行。

通過另一個表更新

PostgreSQL 有一個擴展用法允許通過另一個表更新，使用這樣的語法：

```
UPDATE 表名 FROM 表名 WHERE 條件
```

這是對 SQL 標準的擴充。

嘗試：使用 FROM 更新

為了核對 UPDATE 的 FROM 選項的功能，我們需要先建立一個叫 custphone 的表，包含客戶名和他們的電話號碼。表結構為：

```
CREATE TABLE custphone
(
    customer_id                     serial,
    fname                           varchar(32),
    lname                           varchar(32) NOT NUL
    phone_num                       varchar(16)
);
```

也讓我們插入一些資料到我們新建立的用於保存客戶和他們電話號碼的 custphone 表

```
bpsimple=# INSERT INTO custphone(fname, lname, phone_num)
bpsimple-# VALUES('Peter', 'Bradley', '352 3442');
INSERT 22593 1
bpsimple=#
```

然後我們需要支援 tcust 表中需要更新的行：

```
bpsimple=# UPDATE tcust SET phone = custphone.phone_num FROM custphone
bpsimple-# WHERE tcust.fname = 'Peter' AND tcust.lname = 'Bradley';
UPDATE 1
bpsimple=#
```

解析

我們建立了一個新表包含客戶的電話號碼。然後我們插入資料到新建的表。最後，我們執行 UPDATE 語句，它根據要求修改了那行資料。

當 UPDATE 使用子查詢來控制需要修改的行的時候，FROM 從句允許在 SET 從句中包含從其他表來的列。事實上，FROM 從句甚至都可以不需要。這是因為 PostgreSQL 默認建立一個到任何被用到的表的引用。

從資料庫刪除資料

本章我們需要學的最後內容就是從表中刪除資料。有可能客戶從不下訂單，訂單可能會被取消等，所以我們經常需要從資料庫中刪除資料。

使用 DELETE 語句

刪除資料的常規方式是使用 DELETE 語句。它的語法類似於 UPDATE 語句：

```
DELETE FROM 表名 WHERE 條件
```

注意這裡沒有列的列表，因為 DELETE 是針對行的。如果你想要移除一個列的資料，你必須使用 UPDATE 語句來設置那個列的值為 NULL 或者其他適當的值。

現在我們已經從 tcust 表拷貝我們要的兩條新客戶資料到線上的 customer 表中，所以我們可以繼續前進並從我們的 tcust 表刪除不要的資料了。

嘗試：刪除資料

我們知道在修改資料的時候遺忘 WHERE 從句有多麼危險。我們能體會到意外刪除資料會更非同小可，所以我們先從編寫和通過 SELECT 語句檢查我們的 WHERE 從句開始：

```
bpsimple=# SELECT fname, lname FROM tcust WHERE town = 'Lincoln';
 fname | lname
-------+--------
 Kevin | Carney
 Brian | Waters
(2 rows)

bpsimple=#
```

很好——它列出了我們想要的兩行。

現在我們可以在前面加上 DELETE 語句，經過最後的目視檢查，執行它：

```
bpsimple=# DELETE FROM tcust WHERE town = 'Lincoln';
DELETE 2
bpsimple=#
```

警告：從資料庫刪除資料就是有這麼簡單，所以一定要非常小心

解析

我們寫了一個 WHERE 從句來選擇我們需要從資料庫中刪除的資料並測試了它。然後我們執行一個 DELETE 語句刪除了他們。

和 UPDATE 一樣，DELETE 一次只能工作在一個表中。如果我們需要從不止一個表中刪除相關的行，我們需要使用事務，我們將在第九章碰到。

使用 TRUNCATE 語句

還有另一種方法從一個表刪除資料。它從一個表中刪除所有資料。除非它是包含在 PostgreSQL 7.4 以及以後版本中的事務中，否則你將沒有辦法恢復被刪除的資料。這個命令是 TRUNCATE，它的語法是：

```
TRUNCATE TABLE 表名
```

使用這個命令需要非常小心，只有當你非常確定要永久刪除表中的資料才能使用。從某些方面說，它非常類似於刪除表並重建它，除了它更容易操作且不會重置序列生成器。

嘗試：使用 TRUNCATE 語句

假設我們已經完成我們在表 tcust 中的操作，需要刪除其中的所有資料。我們可以 DROP 這張表，但如果我們以後還會需要它，我們就必須重建它。作為替代操作，我們可以使用 TRUNCATE，刪除表中所有的行：

```
bpsimple=# TRUNCATE TABLE tcust;
TRUNCATE TABLE
bpsimple=# SELECT count(*) FROM tcust;
 count
-------
     0
(1 row)

bpsimple=#
```

所有的行都被刪除了。

解析

TRUNCATE 簡單地將指定表的所有行刪除。

如果你有一個達標，也許有成千上萬行，並且希望刪掉所有的行，預設情況下，PostgreSQL 不會物理移除這些行，而是掃描全部行，並把每一行標記為已刪除。這有助於在事務被回滾的時候恢復資料。即使在命令列中我們可能沒有顯示地啟動事務，所有的命令在一個事務內被自動執行。掃描和標記表中上千行的行為執行起來非常緩慢。TRUNCATE 語句非常高效地刪除表的內容而不需要掃描資料。所以，對於非常大的表，它執行的效率比 DELETE 高很多。

注：有兩種從表中刪除全部資料的方法：沒有 WHERE 的從句和 TRUNCATE。雖然 TRUNCATE 不屬於 SQL92 標準，但是一個非常通用的用於刪除表中全部行的 SQL 語句。

你應該在大部分時候堅持使用 DELETE，因為它是一個比較安全的刪除資料的方法。雖然 DELETE 和 TRUNCATE 在某些情況下可能工作在不同的條件，例如有外鍵的表中。但是，在一些特殊的情況下當你想高效且不需恢復地刪除一個表中的所有行，TRUNCATE 是一個很好的方案。

摘要

在本章，我們看到了資料操作方法中除了 SELECT 之外的三種：使用 INSERT 命令插入資料，使用 UPDATE 命令修改資料以及使用 DELETE 命令移除數據。

我們學習了兩種格式的 INSERT 命令，包括將資料包含在 INSERT 語句中或者從另一個表 SELECT 資料用於 INSERT。我們看到了更長格式的 INSERT 語句會更安全點，因為所有相關的列都被列出來了，所有有更少的機會犯錯誤。我們還碰到 INSERT 的類似命令，更有用的 PostgreSQL 擴展的\copy 命令，它允許資料直接從一個本地檔被插入到一個表中。

我們看到了你應該多麼小心處理 serial 類型欄位的序列計數器，以及怎麼檢查序列生成器的值，以及如果必要，改變它。我們看到，通常，最好允許 PostgreSQL 為你自動生成序列數字，而不是為 serial 類型的列提供一個數值。

我們看到了 UPDATE 和 DELETE 語句是多麼簡單，以及你怎樣配合它們使用 WHERE 從句，就像和 SELECT 語句一樣。我們還提到你應該經常使用 SELECT 測試 UDPATE 和 DELETE 的 WHERE 從句，因為這裡導致的錯誤將很難糾正。

最後，我們看了 TRUNCATE 語句，一個從表中刪除全部行非常高效的方法。因為這是一個不可恢復的刪除，除非在

事務中，否則它應該非常小心使用。

第七章高級資料選擇

在第四章，我們看了一些 SELECT 語句的細節以及我們怎麼使用它來檢索資料。這包括選擇列，選擇行，已經將表連接到一起。在前一章，我們學到了添加、更新和移除數據。在本章，我們回到 SELECT 語句，研究一些更高級的功能。你可能很少需要其中的一些功能，知道他們非常有用，因為你可以對 SQL 中可能有什麼有一個很好的理解。

在本章，我們將碰到一些叫做聚集函數的函數，他們允許我們對一組記錄獲得結果。然後，我們會講述一些更高級的連接功能，它們比我們在第四章討論的簡單的連接功能更能夠控制我們查詢結果。我們還會碰到在一個查詢中使用幾個 SELECT 語句的一整組查詢，叫做子查詢。最後，我們會討論最重要的外連接，這也允許我們使用比我們剛才看到過的方法更靈活的方法連接幾個表到一起。

通過本章，我們將覆蓋以下主題：

- 合計函數
- 子查詢
- UNION 連接
- 自連接
- 外連接

聚集函數

在前面幾章，我們使用了幾個特殊的函數用來在選擇中進行統計：max(列名)函數，告訴我們一個列中最大的值，count(*)函數告訴我們一個表中有多少行。這些函數屬於一小類叫做聚集函數的 SQL 的函數。這類函數在表 7-1 中列出了。

表 7-1 聚集函數

聚集函數	描述
count(*)	提供行的計數
count(列名)	提供指定欄位中值不是 NULL 的行的計數
min(列名)	返回指定列中的最小值
max(列名)	返回指定列中的最大值
sum(列名)	返回指定列的值的合計總數
avg(列名)	返回指定列的值的平均數

聚集函記長非常有用且通常很容易使用。在本屆，我們將介紹表 7-1 中列出的每個函數。PostgreSQL 支援其他的聚集函數包括計算方差和標準差的函數。細節可以在 PostgreSQL 的文檔中找到。

提示：psql 的\da 命令會列出 PostgreSQL 的聚集函數清單。

使用任何聚集函數 SELECT 語句都可以包含兩個可選的從句：GROUP BY 和 HAVING。語法如下（在這裡使用了 count(*)函數）：

```
SELECT count(*), 列名列表 FROM 表名
WHERE 條件 [GROPY BY 列名 [HAVING 聚集條件]]
```

可選的 GROUP BY 從句是一個可以用於 SELECT 的附加條件。它通常在聚集函數被時候的時候才有用。它也可以用於提供類似於我們在第四章碰到的 ORDER BY 的功能，但僅用於聚集列。可選的 HAVING 從句允許我們通過 GROPY BY 選取的函數結果符合某些條件的特殊行。這些聽上去都有點複雜，但實際上卻非常簡單，就像我們會在本章中看到的。

count 函數

我們從 count 函數開始，就像你在表 7-1 看到的，它有兩種格式：count(*)和 count(列名)。

Count(*)

Count(*)函數提供結果集中行的計數。它在 SELECT 語句中扮演著一個特殊的列名。讓我們嘗試一個最簡單的 count(*)來理解一些基本的概念。

注：在本章的例子中，和其他章一樣，我們從我們示例資料庫中乾淨的基本資料開始，所以讀者可以沉浸在他們選擇的任何一章。這意味著如果你接著上一章的示例資料繼續，輸出將稍微有點不同。你可以準備腳本刪除表並重新建立，以及填充資料如果你想輸出相同。

嘗試：使用 count(*)

假設我們想知道在 customer 表中有多少客戶住在 Bingham。我們可以簡單的寫一個這樣的 SQL：

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

或者另一個更有效的返回更少數據的版本，我們可以寫一個這樣的 SQL：

```
SELECT customer_id FROM customer WHERE town = 'Bingham';
```

這做到了，但是是通過一個間接的方法。假設 customer 表包含成千上萬的客戶，其中有上千個住在 Bingham 的客戶。在這種情況下，我們會檢索到大量我們不需要的資料。函數 count(*)為我們解決了這個問題，它允許我們僅僅獲得一行資料，包含我們選擇的行數。

我們和往常一樣寫這個 SQL，只是將選擇的列用 count(*)代替，就像這樣：

```
bpsimple=# SELECT count(*) FROM customer WHERE town = 'Bingham';
 count
-------
     3
(1 row)

bpsimple=#
```

如果我們要知道所有客戶的數量，我們只需要去掉 WHERE 從句：

```
bpsimple=# SELECT count(*) FROM customer;
 count
-------
    15
(1 row)

bpsimple=#
```

你會發現我們只獲得了包含總數的一行。如果你想檢查結果，你只需要用 customer_id 代替 count(*)來顯示真實的資料就可以了。

解析

Count(*)函數允許我們獲取一個物件的計數，而不是獲得物件本身。它比獲得資料本身提高了很多性能，因為我們不需要從資料庫檢索到我們不需要的資料，更壞的情況下，還要通過網路傳送。

> 注：當你只需要行數時，你永遠不要檢索資料。

GROUP BY 和 count(*)

如果我們想知道每個城鎮有多少客戶。我們可以可以先通過選擇不同的城鎮，然後分別統計每個城鎮有多少客戶。這是一個繁瑣且單調的解決問題的方法。是不是在 SQL 中有一種公開的方法來簡單解決這個問題呢？你可能想嘗嘗像這樣的：

```
SELECT count(*), town FROM customer;
```

這是基於我們現有知識的一個有理的猜測，但是 PostgreSQL 將產生一個錯誤資訊，說它不是有效的 SQL 語法。你需要知道的額外的解決這個問題的一點語法是 GROUP BY 從句。

GROUP BY 從句告訴 PostgreSQL 我們想使用一個聚集函數在一個列或者一組列的值改變時輸出值並重置。它非常容易使用。你只需要在帶有 count(*)的 SELECT 語句後簡單地添加一個“GROUP BY 列名”從句。PostgreSQL 將告訴你在表中這個列的每個值有多少條。

嘗試：使用 GROUP BY

讓我們嘗試回答“每個城鎮有多少客戶？”這個問題。

第一步是寫一條 SQL 檢索計數和列名：

```
SELECT count(*), town FROM customer;
```

然後添加 GROUP BY 從句，通過指定一個 SQL 查詢告訴 PostgreSQL 在城鎮每次改變的時候產生一個結果以及重置計數，就像這樣：

```
SELECT count(*), town FROM customer GROUP BY town;
```

以下是我們的行動：

```
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，我們獲得一個城鎮的清單以及每個城鎮的客戶數量。

解析

PostgreSQL 按照 GROUP BY 從句裡的列將結果排序。它然後保存一個行數的計數器，每次城鎮名字改變，它就寫出結果集並且重置計數器為零。你會認同這比寫代碼迴圈查詢每個城鎮來統計結果更容易。

如果我們需要，我們可以擴展這個想法到超過一個列，將我們所有選擇的列都列在 GROUP BY 從句中。假設我們想知道兩塊資訊：每個城鎮有多少姓相同的客戶。我們可以簡單的在語句中的 SELECT 和 GROUP BY 部分添加 lname：

```
bpsimple=# SELECT count(*), lname, town FROM customer GROUP BY town, lname;
 count |  lname  |   town
-------+---------+-----------
     1 | Hardy   | Oxbridge
     1 | Cozens  | Oakenham
     1 | Matthew | Yuleville
     1 | Jones   | Bingham
     2 | Matthew | Nicetown
     1 | O'Neill | Welltown
     1 | Stones  | Hightown
     2 | Stones  | Bingham
     1 | Hudson  | Milltown
     1 | Hickman | Histon
     1 | Neill   | Winnersby
     1 | Howard  | Tibsville
     1 | Stones  | Lowtown
(13 rows)

bpsimple=#
```

注意 Bingham 列出了兩次，因為在 Bingham 住有兩個客戶有不同的姓，分別是 Jones 和 Stones。

還要注意輸出是不排序的。在 8.0 版本之前的 PostgreSQL 會按 town 排序，然後是 lname，因為這是他們在 GROUP BY 從句中的順序。在 PostgreSQL 8.0 以及以後的版本，我們需要更明確地通過 ORDER BY 從句指定排序順序。我們可以通過這樣獲取排序的輸出：

```
bpsimple=# SELECT count(*), lname, town FROM customer GROUP BY town, lname
bpsimple-# ORDER BY town, lname;
 count |  lname  |   town
-------+---------+-----------
     1 | Jones   | Bingham
     2 | Stones  | Bingham
     1 | Stones  | Hightown
```

```
     1 | Hickman | Histon
     1 | Stones  | Lowtown
     1 | Hudson  | Milltown
     2 | Matthew | Nicetown
     1 | Cozens  | Oakenham
     1 | Hardy   | Oxbridge
     1 | Howard  | Tibsville
     1 | O'Neill | Welltown
     1 | Neill   | Winnersby
     1 | Matthew | Yuleville
(13 rows)

bpsimple=#
```

HAVING 從句和 count(*)

SELECT 語句的最後一項可選部分是 HAVING 從句。這個從句對於 SQL 新手會有點感到迷惑，但它也不難使用。你只要記住 HAVING 是一種用於聚集函數的 WHERE 從句。我們使用 HAVING 來約束返回的結果 為針對特定的聚集的條件為真的行，例如 count(*) > 1。它的使用方法和我們通過列值限制行的方法一樣。

> 警告：聚集無法使用在 WHERE 從句中。他們只能用在 HAVING 從句中。

讓我們看一個例子。假設我們想知道有超過一個客戶的城鎮。我們可以使用 count(*)，然後直接查看相關的城鎮。但是，在有成千上萬的城鎮的情況下，這不是一個合理的方案。作為替代，我們使用一個 HAVING 從句來約束結果為 count(*) 大於一的行，就像這樣：

```
bpsimple=# SELECT count(*), town FROM customer
bpsimple-# GROUP BY town HAVING count(*) > 1;
 count |   town
-------+----------
     3 | Bingham
     2 | Nicetown
(2 rows)

bpsimple=#
```

注意我們仍然需要使用 GROUP BY 從句，而且它出現在 HAVING 從句前。現在我們已經知道 count(*)，GROUP BY 和 HAVING 的基本用法了，讓我們將他們放到一起到一個大點的示例中。

嘗試：使用 HAVING

假設我們考慮關於建立一個投遞計畫。我們需要知道我們所有客戶的姓和所在城鎮，除了 Lincoln（也許它就是我們所在的本地城鎮），而且我們只對城鎮中客戶名大於一個的客戶感興趣。

這沒有它聽上去難。我們只需要一點點建立我們的解決方案，這也通常是一個完成 SQL 的好方法。如果它看上去非常難，可以從一個簡單點的，但類似的問題開始，然後擴展開始的方案知道解決你比較複雜的問題。事實上，面對問題，

將它拆分為小的問題，然後分別解決每個小問題。

讓我們從簡單返回資料開始，而不是統計計數。我們按照 town 排序來讓我們更容易看清楚究竟發生了什麼：

```
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=#
```

到目前為止，看起來不錯，不是嗎？

現在如果我們使用 count(*)來為了名進行計數，我們還需要針對 lname 和 town 使用 GROUP BY：

```
bpsimple=# SELECT count(*), lname, town FROM customer
bpsimple-#  WHERE town <> 'Lincoln' GROUP BY lname, town ORDER BY town;
 count |  lname  |   town
-------+---------+-----------
     2 | Stones  | Bingham
     1 | Jones   | Bingham
     1 | Stones  | Hightown
     1 | Hickman | Histon
     1 | Stones  | Lowtown
     1 | Hudson  | Milltown
     2 | Matthew | Nicetown
     1 | Cozens  | Oakenham
     1 | Hardy   | Oxbridge
     1 | Howard  | Tibsville
     1 | O'Neill | Welltown
     1 | Neill   | Winnersby
     1 | Matthew | Yuleville
(13 rows)
```

```
bpsimple=#
```

事實上我們可以通過肉眼檢查，但是我們已經接近完整的解決方案了，我們只需要添加一個 HAVING 從句來選取那些 count(*)大於一的行：

```
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，當你把問題分解成幾部分，解決方案會非常簡單。

解析

我們分三步解決了這個問題：

- 我們寫一個簡單的 SELECT 語句來選擇我們感興趣的行。
- 然後添加一個 count(*)函數以及一個 GROUP BY 從句，來統計唯一的 lname 和 town 組合。
- 最後我們添加了一個 HAVING 從句來抽取那些 count(*)大於一的行。

這種方法有一個小問題，在我們的小示例資料庫中並不明顯。在一個大資料庫中，這個反覆運算開發方法有一些缺點。如果我們在一個由上千行客戶資料的資料庫中工作，我們在開發我們的查詢的過程中需要等待我們的客戶清單滾動很久。幸運的是， 通常有一個簡單的方法通過使用資料的樣本開發你的查詢，使用主鍵。如果我們在我們所有的查詢中使用條件 WHERE customer_id < 50，我們可以使用資料庫中的 customer_id 的前 50 條作為我們的樣本。一旦我們滿意了我們的 SQL，我們可以簡單地移除 WHERE 從句來針對整個資料庫執行我們的方案。當然，我們需要留意樣用於測試我們的 SQL 的本資料對完整的資料集有代表性且小心太少的樣本可能無法完全執行我的 SQL。

Count(列名)

一個 count(*)函數的微小的變種是使用一個列名替代 “*” 。不同的是 count(列名)統計表中提供的列值不為 NULL 的計數。

嘗試：count(列名)

假設我們添加了更多的資料到我們的 customer 表，一些新客戶的電話號碼為 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');
```

讓我們查查我們有多少客戶我們不知道電話號碼：

```
bpsimple=# SELECT customer_id FROM customer WHERE phone IS NULL;
 customer_id
-------------
          16
          18
          19
(3 rows)

bpsimple=#
```

我們可以看到我們有三個客戶我們不知道電話號碼。讓我們看看我們總共有多少客戶：

```
bpsimple=# SELECT count(*) FROM customer;
 count
-------
    19
(1 row)

bpsimple=#
```

總共有 19 個客戶。現在如果我們想統計 phone 不為 NULL 的客戶有多少，應該是 16 個：

```
bpsimple=# SELECT count(phone) FROM customer;
 count
-------
    16
(1 row)

bpsimple=#
```

解析

Count(列名)和 count(*)唯一的區別是帶列名的函數只統計這個列不是 NULL 的行，而*格式的則統計所有的行。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)

聚集函數 count 支援 DISTINCT 關鍵字，它約束函數只考慮一個列中值唯一的情況，不統計重複的。我們可以通過統計我們的 customer 表中不重複的城鎮來說明它的這種行為：

```
bpsimple=# SELECT count(DISTINCT town) AS "distinct", count(town) AS "all"
bpsimple=# FROM customer;
 distinct | all
```

```
----------+-----
       12 |  15
(1 row)

bpsimple=#
```

在這裡我們可以發現總共有 15 個城鎮，但只有 12 個不重複的（Bingham 和 Nicetown 出現了不止一次）。

現在，我們可以理解 count(*)以及學到了聚集函數的原理，我們可以將這相同的邏輯應用到其他聚集函數上了。

Min 函數

就像你預計的，min 函數使用一個列名做參數且返回這個列中最小的值。對於 numeric 類型的列，結果應該和預期一樣。對於時態類型，例如 date 的值，它返回最小的日期，日期既可以是過去也可以是未來。對於變長的字串（varchar 類型），結果可能和預期有點不同：它在字串右邊添加空白後再進行比較。

> 警告：小心在 varchar 類型的列中使用 min 或者 max，因為結果可能不是你預期的。

例如，假設我們想找到我們訂單中收取的最小運費。我們可以使用 min 函數，就像這樣：

```
bpsimple=# SELECT min(shipping) FROM orderinfo;
 min
------
 0.00
(1 row)

bpsimple=#
```

這顯示最小的費用是零。

注意當我們在 phone 列嘗試使用這個函數將會發生什麼，因為我們知道這裡有 NULL 值：

```
bpsimple=# SELECT min(phone) FROM customer;
   min
----------
 010 4567
(1 row)

bpsimple=#
```

你可能預期結果會是 NULL，或者一個空串。因為 NULL 通常指位置，因此，min 函數忽略 NULL 值。忽略 NULL 值是所有的聚集函數的一個特點，除了 count(*)（當然，是否一個電話號碼是最小值又是另一個問題了）。

Max 函數

說 max 函數除了與 min 函數相反外其他都一樣一點也不奇怪。和你想像的一樣，max 函數使用一個列名作為參數且返回那個列的最大值。

例如，我們可以通過以下方法查找我們訂單中收取的最大運費值：

```
bpsimple=# SELECT max(shipping) FROM orderinfo;
```

```
 max
------
 3.99
(1 row)

bpsimple=#
```

和 min 函數一樣，NULL 值被 max 忽略掉，就像這個示例：

```
bpsimple=# SELECT max(phone) FROM customer;
   max
----------
 961 4526
(1 row)

bpsimple=#
```

這就是差不多全部你需要知道的關於 max 的內容。

Sum 函數

Sum 函數使用一個列名作為參數並提供列的內容的合計。和 min 和 max 一樣，NULL 值被忽略。

例如，我們可以這樣獲得所有訂單中運費的總和：

```
bpsimple=# SELECT sum(shipping) FROM orderinfo;
 sum
------
 9.97
(1 row)

bpsimple=#
```

和 count 一樣，sum 函數支援 DISTINCT 變體。你可以讓它只統計不重複值的和，所以多條值相同的行只會被加一次：

```
bpsimple=# SELECT sum(DISTINCT shipping) FROM orderinfo;
 sum
------
 6.98
(1 row)

bpsimple=#
```

注意，在實際中，對這種變體的實際應用很少。

Avg 函數

我們要看的最後一個聚集函數是 avg，它使用一個列名做參數並返回這個列數值的平均值。和 sum 一樣，它忽略 NULL

值。這裡是一個示例：

```
bpsimple=# SELECT avg(shipping) FROM orderinfo;
     avg
--------------
 1.9940000000000000
(1 row)

bpsimple=#
```

avg 函數也可以使用一個 DISTINCT 關鍵字來處理不重複的值：

```
bpsimple=# SELECT avg(DISTINCT shipping) FROM orderinfo;
     avg
--------------
 2.3266666666666667
(1 row)

bpsimple=#
```

注：在標準 SQL 和 PostgreSQL 的實現中，沒有 mode 和 median 函數。但是幾個商業廠商以擴展模式支援它們。

子查詢

到現在為止，我們遇到的各種 SQL 語句都只有一個 SELECT，我們能夠看到一個完整級別的使用多種方法結合兩個甚至更多 SELECT 語句的資料檢索語句。

子查詢是指一個 SELECT 查詢的一個或多個 WHERE 條件中的 SELECT 語句。子查詢應該較單個 SELECT 語句的查詢更難理解，但是它們非常有用且將開闢資料查詢標準的新領域。

假設我們想找到價格比平均價格高的商品項目。我們可以通過兩步做到：使用帶聚集函數的 SELECT 語句找到平均價格，然後在第二個 SELECT 語句中使用這個值來查找我們要的行（使用在第四章介紹的 cast 函數），就像這樣：

```
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=#
```

這看上去確實非常不雅。我們實際上想要的是將第一個查詢的結果直接傳遞給第二個查詢，而不需要記住並輸入到第二個查詢中。

解決方案是使用子查詢。我們將第一部分查詢放入括弧並將它用作第二個查詢的 WHERE 從句的一部分，就像這樣：

```
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，我們獲得了相同的結果，但不需要中間步驟或者 cast 函數，因為結果已經是正確的類型了。PostgreSQL 先執行括弧中的查詢。在獲得回答後，它再運行外面的查詢，使用了裡面查詢的結果。

如果需要，我們可以在 WHERE 從句裡頭包含很多子查詢。我們不限制於只能使用一個，雖然需要多個嵌套 SELECT 的情況非常少見。

嘗試：使用子查詢

讓我們試一個複雜點的例子。假設我們想知道那些成本高於平均成本但售價低於平售價的產品（這些指標表明我們的利潤不是很好，所以我們希望不會有太多符合這些標準的項目）。一般情況下的查詢將是這種形式：

```
SELECT * FROM item
WHERE cost_price > average cost price
AND sell_price < average selling price
```

我們已經知道平均成本可以使用“SELECT avg(cost_price) FROM item”獲得。查找平均售價可以通過類似的樣子做到，就是使用查詢“SELECT avg(sell_price) FROM item”。

如果我們將這三個查詢放到一起，我們得到這個：

```
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=#
```

也許一些人需要查看 Picture Frame 的價格來看看它是否正確！

解析

PostgreSQL 首先掃描查詢併發現有兩個在括弧中的子查詢。它獨自評價每個子查詢，然後在主查詢被執行前，將答案放回到主查詢 WHERE 從句中的適當部分。

我們也可以使用附加的 WHERE 從句或者 ORDER BY 從句。它們可以完美有效地使用更常規的條件結合到子查詢中的 WHERE 條件中。

返回多行記錄的子查詢

到現在為止，我們只看到了返回一條結果的子查詢，因為子查詢中使用了一個聚集函數。子查詢也可以返回零行或者更多行記錄。

假設我們想知道我們庫存的哪些商品項的成本價高於 10.0。我們可以使用一條單獨的 SELECT 語句，就像這樣：

```
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=#
```

注意我們為了讓查詢更短，使用了表的別名（stock 表變成了 s，item 表變成了 i）。我們所做的就是連接兩個表（s.item_id = i.item_id），同時添加一個關於 item 表的成本的條件（i.cost_price > cast(10.0 AS NUMERIC(7,2))）。

我們也可以把這個寫作一個查詢，使用關鍵字 IN 來檢測這個列表的值。在這個上下文中使用 IN，我們首先需要寫一個查詢來返回 item 表中中成本高於 10.0 的 item_id 的列表：

```
SELECT item_id FROM item WHERE cost_price > cast(10.0 AS NUMERIC(7,2));
```

我們還需要從 stock 表查詢一些內容：

```
SELECT * FROM stock WHERE item_id IN 值的列表
```

我們可以把這兩個查詢放到一起，就像這樣：

```
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=#
```

這顯示了相同的結果。

就像傳統的查詢，我們可以使用 NOT IN 來反轉條件，我們也可以使用 WHERE 從句和 ORDER BY 條件。

使用子查詢或者關聯查詢都可以查出相同結果的情況很常見。但是，並不總是這樣；並不是所有的子查詢可以寫成連接，所以理解它們很重要。

如果你用子查詢的情況也可以用連接做到，你應該選擇哪個？這裡有兩個問題需要考慮：可讀性和性能。如果查詢是你偶爾使用在小表上且執行很快，使用你覺得可讀性最好的。如果它是一個非常常用的在大表上的查詢，那麼使用不同的方式編寫並試驗來查看那種格式是最好就非常值得了。你可能會發現查詢最佳化工具有能力對兩種樣式都進行優化，所以它們的性能都是一樣的，所以可讀性自然勝出。你還可能發現性能嚴重地依賴於你資料庫中的實際護具，或者說它伴隨著不同表的行數的變化發生急劇變化。

注意：小心測試 SQL 語句的性能。有很多你無法控制的變數，例如作業系統的資料緩存。

相關子查詢

我們到現在為止看到的子查詢都是我們先執行一個查詢獲得一個答案，然後再把它“安插”到第二個查詢中。這兩個查詢是不相關聯的，叫做不相關子查詢。這是因為沒有連結表關聯在內部和外部查詢之間。我們可能在兩部分 SQL 語句中使用相同的表的相同的列，但它們只是通過子查詢的結果在傳遞到主查詢的 WHERE 從句後關聯起來。

有另一組子查詢，叫做相關子查詢，查詢的兩部分的關係比較複雜。在相關子查詢中，內部的 SELECT 中的一個表會連接到外部 SELECT 中，從而確定這兩個查詢之間的關係。這是一組非常強大的子查詢，它往往不能簡寫成一個帶連接簡單的 SELECT 語句。一個相關的查詢具有的一般形式是：

```
SELECT columnA from table1 T1
WHERE T1.columnB =
  (SELECT T2.columnB FROM table2 T2 WHERE T2.columnC = T1.columnC)
```

為了易於理解，我們已經將它們寫成了一些偽 SQL。需要強調的是在外面的 SELECT 語句中的表 T1 也出現在內部的 SELECT 中。因此，內部和外部的查詢被視為相關的。你會發現我們使用了表的別名。這非常重要，因為相關子查詢的表名的規則相當複雜，稍有不慎就可能得到奇怪的結果

提示：我們強烈建議你在相關子查詢中使用表的別名，因為他是最安全的方法。

當這個相關的子查詢被執行時，會發生一些非常複雜的事情。首先，T1 表中的一行資料會由外部的 SELECT 查詢出，然後 T1.columnB 列被傳遞到內部的查詢，內部的查詢然後被執行，通過傳入的資訊從 T2 表查詢資料。查詢結果然後被返回到外部的查詢，外部查詢在移動到下一行之前完成對 WHERE 從句的評估。這在圖 7-1 中被展示出來。

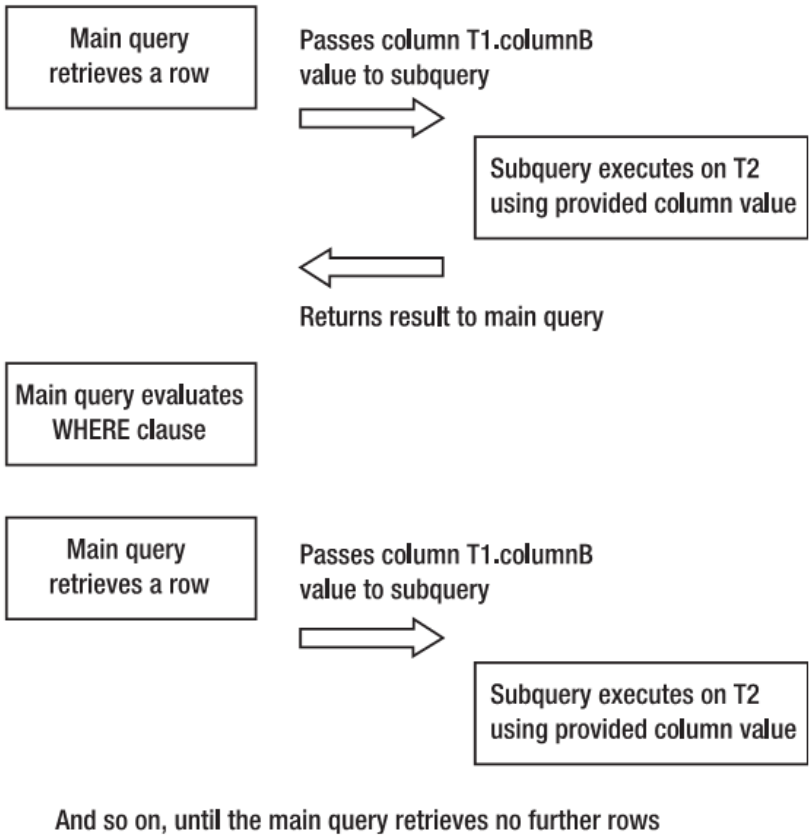


圖 7-1 相關子查詢的執行

如果你覺得它聽起來有點冗長，那是因為它卻是是這樣的。相關的子查詢通常執行起來非常沒有效率。不過，它們偶爾也會解決一些相當複雜的問題。所以，最好知道它們的存在，即使你只是偶然使用它們。

嘗試：執行一個相關子查詢

在一個簡單的資料庫，例如我們在用的這個，對相關子查詢的需求很少，但我們仍然可以使用我們的示例資料庫演示它們的用法。

假設我們想知道在 Bingham 的客戶下的訂單的日期。雖然我們可以用很老套的方法實現，但我們會使用相關子查詢，就像這樣：

```
bpsimple=# SELECT oi.date_placed FROM orderinfo oi
bpsimple-#  WHERE oi.customer_id =
bpsimple-#  (SELECT c.customer_id from customer c
bpsimple(#   WHERE c.customer_id = oi.customer_id and town = 'Bingham');
 date_placed
-------------
 2000-06-23
 2000-07-21
(2 rows)
```

```
bpsimple=#
```

解析

查詢開始於從 orderinfo 表選擇一行資料。它之後在 customer 表上使用找到的 customer_id 來執行子查詢。子查詢被執行，查找從外部查詢中給出的 customer 中的行中城鎮為 Bingham 的行。如果它找到一行，則傳遞 customer_id 回到外部的查詢，它再完成 WHERE 從句，如果 WHERE 從句為真，則列印 date_placed 列。外部查詢繼續處理下一行，重複以上步驟。

也可能建立一個相關子查詢，這個子查詢在 FROM 從句中。以下是一個例子用來查找所有的在 Bingham 的給我們下過訂單的客戶的全部資料：

```
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=#
```

子查詢的結果作為主查詢的一個表，也就意味著子查詢產生了一個在 Bingham 的客戶的結果集 。

現在你已經瞭解了怎麼寫相關子查詢的。當你碰到一個你仿佛無法用常見的查詢 SQL 解決的問題時，你可以發現相關子查詢就是解決你的難題的答案。

存在子查詢（Existence Subqueries）

另一種格式的子查詢在 WHERE 中使用 EXISTS 關鍵字來檢查是否存在，而不需要知道資料的內容。

如果我們想列出所有的下了訂單的客戶。在我們的示例資料庫中沒有太多。查詢的第一部分很容易：

```
SELECT fname, lname FROM customer c;
```

注意我們給表 customer 一別名 c，為子查詢做準備。查詢的下一步需要確定是否 customer_id 也存在於 orderinfo 表中：

```
SELECT 1 FROM orderinfo oi WHERE oi.customer_id = c.customer_id;
```

有兩個非常重要的方面需要在這裡注意。首先，我們使用了一個通用的技巧。在我們需要執行查詢但不需要返回結果集的地方，我們只要簡單地在列名所在的位置使用 1 代替。這意味著如果找到任何資料，將返回 1，這是一種簡單有效的表達真值的方法。這是一種古怪的方法，所以我們先嘗試一下它：

```
bpsimple=# SELECT 1 FROM customer WHERE town = 'Bingham';
 ?column?
```

```
----------
         1
         1
         1
(3 rows)

bpsimple=#
```

它看上去有點古怪，但它確實有效。在這裡不要使用 count(*)很重要；因為我們需要針對城鎮是 Bingham 的每一行得到一個結果，而不僅僅是知道有多少客戶是 Bingham 的。

第二個要注意的重要事情是我們在這個子查詢中使用了 customer 表，它實際上是存在於主查詢中。這就是它的相關性所在。和以前一樣，我們給所有這些表別名。現在，我們需要將這兩半放到一起了。

對於我們的查詢，使用 EXISTS 才是將兩個 SELECT 語句結合到一起的正確方法，因為我們只需要知道子查詢是否返回了記錄：

```
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 從句通常比其他類型的關聯或者 IN 條件更高效。因此，在你選擇怎麼寫一個子查詢的時候，通常值得優先使用它而不是其他類型的連接。

UNION 連接

我們現在將開始弄清另一種結合多個 SELECT 語句到一起為我們提供更高級的查詢能力的方法。讓我們從一個我們需要解決的問題的示例開始。

在前一章，我們在添加資料到我們的主表 customer 時使用表 tcust 作為載入資料表。現在假設我們在載入新的客戶資料，正處在新客戶資料已經載入到 tcust 表進行清理但還未進入 customer 表的時候，我們被要求列出所有我們有客戶的城鎮，包括新資料。我們可能很有理由地指出因為我們還沒有清理並載入客戶資料到主表，我們無法確保新資料的準確性，所以這兩個城市列表中的任何一個都可能不準確。然而，核對準確性可能不是很重要。也許我們所需要的只是一個大致的客戶分佈地理資訊，而不是準確的資料。

我們可以通過從 customer 表中選擇 town，保存下來，再從 tcust 表裡選擇 town，保存下來，然後結合這兩個列表來解決這個問題。這確實看上去非常不雅，因為我們需要查詢兩個表，他們都包含一個城鎮的列表，然後保存結果，之後在通過某些方法合併他們。

難道沒有什麼辦法可以讓我們自動聯合這兩個城鎮的列表？就像你可以從本節的標題中獲取到的，確實有個辦法，叫做 UNION 連接。這些連結不是很常見，但是在一些條件下，它們卻正是我們解決問題需要的，而且它們非常容易使用。

嘗試：使用 UNION 連接

讓我們從放入一些資料到 tcust 表開始，所以它看上去就像這樣：

```
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=#
```

我們已經知道怎麼從每個表中選擇 town。我們用一對簡單的 SELECT 語句，就像這樣：

```
SELECT town FROM tcust;
SELECT town FROM customer;
```

它們每一條語句都給出了一個城鎮的列表，為了能結合他們，我們使用 UNION 關鍵字來連接這兩個 SELECT 語句到一起：

```
SELECT town FROM tcust UNION SELECT town FROM customer;
```

我們輸入我們的 SQL 語句，差分它為多行以讓它更易讀。注意 psql 提示符從=#轉變成-#來表示這是繼續輸入的行，而且這裡只有一個簡單的分號在最後面，因為這只是一個單獨的 SQL 語句：

```
bpsimple=# SELECT town FROM tcust
bpsimple-# UNION
bpsimple-# SELECT town FROM customer;
   town
-----------
 Bingham
 Hightown
 Histon
 Keynes
 Lincoln
 Lowtown
 Milltown
 Nicetown
 Oahenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(14 rows)

bpsimple=#
```

解析

PostgreSQL 從兩個表中獲取城鎮的列表並結合它們到一個單獨的列表。但是請注意，它消除了所有重復資料。如果我們想得到所有的城鎮的列表，包括重複的，我們需要使用 UNION ALL，而不是 UNION。

這種結合 SELECT 語句的能力不限於一個列；我們可以結合城鎮和郵遞區號：

```
SELECT town, zipcode FROM tcust UNION SELECT town, zipcode FROM customer;
```

這將產生一個包含兩個列的列表。它可能會更長點，因為它包含了郵遞區號，因此這裡有更多不重複的行被檢索到。

UNION 連接的使用有一些限制。你要連接的兩個從兩個表中查找列表的列必須有相同列數，而且選擇的每個列必須都有相相容的類型。

讓我們看另一個例子用來連接兩個不同但是相容的列，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=#
```

這個查詢，雖然非常無意義，但是是有效的，因為 PostgreSQL 可以連接這兩個列，即使 title 是一個固定長度的列而 town 是一個變長的列，因為他們都是字串類型。例如如果我們嘗試連接 customer_id 和 town，PostgreSQL 會告訴我們無法做到，因為這兩個列的類型不同。

總體來說，這就是你需要知道的所有關於 UNION 連接的內容。有時候，他們是一個方便用來從兩個或多個表中聯合資料的方法。

自連接

一種非常特殊的連接叫做自連接，它在我們想針對同一個表中的兩個列使用連接時被使用。我們很少需要使用它，但偶爾它會非常有用。

假設我們賣的東西可以單獨賣或者作為一套賣。作為示例，我們可以把桌子和椅子作為一套，也可以單獨賣。我們想要做的是不但存儲每個單獨的專案，而且在它們作為一套銷售時需要記錄他們的關係。這通常被叫做零件分類（parts explosion），我們將在第 12 章再次碰到它。

讓我們從建立一個可以不但保存商品項目的 ID 和描述，還能保存另一個商品 ID 的表開始，就像這樣：

```
CREATE TABLE part (part_id int, description varchar(32), parent_part_id INT);
```

我們將使用 parent_part_id 來存儲作為元件的商品的 ID。在本例中，一套桌椅的 item_id 為 1，椅子的 item_id 為 2，桌子的 item_id 為 3。相應的 INSERT 語句應該是這樣：

```
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=#
```

現在，我們已經存儲好了資料了，但是我們怎麼檢索關於組成一個特殊元件的部件的資訊呢？我們需要連接 part 表到它自己。這原來很容易。我們給表別名，然後我們可以寫一個 WHERE 從句來針對相同的表使用不同的名字來處理：

```
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=#
```

這成功運行了，但是結果有點點難以理解，因為我們的兩個輸出的列的名字相同。我們可以簡單地通過使用 AS 給它們名字以糾正這點：

```
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=#
```

我們將在第十二章討論怎麼存儲管理者/下屬的關係到表中的時候再次看到自連接。

外連接

連一類連接為外連接。這種連接更接近於傳統連接，但是它使用稍微不同的語法，這也是為什麼我們推遲到現在才碰到它們的原因。

假設我們想知道我們銷售的所有商品項目，說明我們的庫存數量。這看似簡單的要求以我們當前知道的 SQL 來做是驚人的困難的，雖然也可以做到。本例使用到了我們示例資料庫中的 item 何 stock 表。就像你記得的，我們所有可以出售的商品都保存在 item 表中，而我們有庫存的商品存在 stock 表中，如圖 7-2 所示：

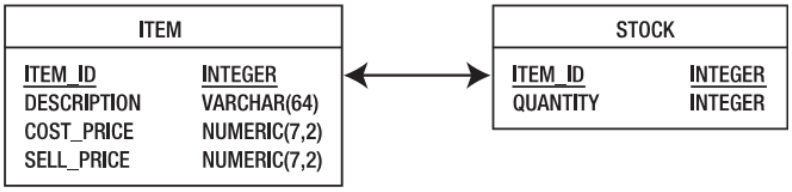


圖 7-2 item 和 stock 表的模式

讓我們完成一個解決方案，從使用我們現在為止知道的 SQL 開始。讓我們嘗試使用一個單獨的 SELECT 來連接兩張表：

```
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（因為我們碰巧知道 item 表中的 item_ids 欄位是順序增長的，且沒有間隙），一些 item_id 丟失了。丟失的行為我們沒有庫存的那些商品，因為在 item 和 stock 表之間的連接無法連接到那些行，因為 stock 表沒有那些 item_id 的條目。我們可以使用一個子查詢和一個 IN 從句找到這些丟失的行。

```
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=#
```

我們可以把它翻譯為“告訴我所有的 item 表中的 item_id，但排除那些出現在 stock 表中的”。

內部的 SELECT 語句就是我們前面使用的那個簡單的語句，但這個時候，我們使用它返回的 item_id 的列表作為另一個 SELECT 語句的一部分。主 SELECT 語句列出所有的 item_id，除了 WHERE NOT IN 從句排除了那些子查詢中找到的那些 item_id。

那麼現在我們有一個我們沒有庫存的商品的 item_id 的列表，以及一個我們有庫存的商品的 item_id 的列表，只是是

從不同的查詢中檢索到的。我們現在需要做的是連接這兩個列表到一起，這是 UNION 連結的工作。但是，這裡有一個小問題。我們的第一個語句返回兩個列，item_id 和 quantity，但是我們的第二個 SELECT 只返回 item_id，因為它們沒有庫存。我們需要添加一個偽列到第二個 SELECT 語句中，這樣它才有和第一個 SELECT 一樣的數量和類型的列。我們使用 NULL。這裡是我們完成了的查詢：

```
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);
```

這看上去有點複雜，但我們還是先試試吧：

```
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=#
```

在較早時候的 SQL 中，這可能是唯一的可以解決這類問題的辦法，除了 SQL89 標準限制我們在第二個 SELECT 語句中使用 NULL 作為一個列。幸運的是，大多數廠家允許 NULL，否者我們的日子可能更難過。如果我們不允許使用 NULL，我們可以使用 0（零）作為下一個方案。NULL 更好，因為 0 可能導致誤導；NULL 可以總是表示為空白。

為了避免這種解決這類常見問題的相當複雜的解決方案，廠家設計出了外連接。不幸的是，因為這種類型的連接沒有出現在標準中，所有的廠家發明出它們自己的解決方案，使用類似的方法，但是不同的語法。

Oracle 和 DB2 使用一個在 WHERE 從句中加一個+標記的語法來指出一個表的所有值必須被顯示出來（保留表），即使連接失敗。Sybase 在 WHERE 從句中使用*=來指出保留表。這些語法都是相當簡單，但不幸的是他們不同，對於你的 SQL 的移植性來說不好。

當 SQL92 標準出現後，它指出了一個非常通用的方法來實現外連接，為外連接帶來了一個更合理的系統。廠家已經開始，但很慢地實現新的標準。（例如，Sybase 11 和 Oracle 8 都在 SQL92 標準出現後出現，但都不支持它）PostgreSQL 從 7.1 版開始實現了 SQL92 標準提供的這個方法。

注：如果你在運行一個 7.1 版本之前的 PostgreSQL，你需要升級以便嘗試本章中最後的例子。其實如果你還運行在一個比 7.x 更老的版本中，你值得升級了，因為 8 版本已經明顯比舊版本提升了很多了。

SQL92 標準中的外連接語法使用一個 ON 從句替代我們熟悉的 WHERE 從句來連接表，並添加了 LEFT OUTER JOIN 關鍵字。語法就像這樣：

```
SELECT columns FROM table1
  LEFT OUTER JOIN table2 ON table1.column = table2.column
```

連接到 LEFT OUTER JOIN 的表就是保留表，它的全部行將顯示出來。

所以我們可以重寫我們的查詢，使用這個新語法：

```
SELECT i.item_id, s.quantity FROM item i
  LEFT OUTER JOIN stock s ON i.item_id = s.item_id;
```

是不是這樣做看上簡單到不像是真的？讓我們運行一下它：

bpsimple=# SELECT i.item_id, s.quantity FROM item i

bpsimple-# LEFT OUTER JOIN stock s ON i.item_id = s.item_id;

```
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       3 |
       4 |        8
       5 |        3
       6 |
       7 |        8
       8 |       18
       9 |
      10 |        1
      11 |
(11 rows)

bpsimple=#
```

就像你看到的，結果與我們之前版本獲得的結果一致。

你可以理解為什麼大多數廠商他們需要實現一個外連接，即使它不是原始的 SQL89 標準。

也有一個相應的 RIGHT OUTER JOIN，但 LEFT OUT JOIN 更常用（至少對於西方人，他們更習慣在左側列出已知的項目而不是右邊）。

嘗試：使用一個更複雜的條件

到現在為止我們使用的簡單的 LEFT OUTER JOIN 已經非常好用了，但我們怎樣添加更複雜的條件呢？

假設我們僅僅想知道 stock 表中的我們的庫存多於兩個的行，而且我們僅對售價高於 5.0 的行感興趣。這是一個比較複雜的問題，因為我們要應用一條規則到 item 表（cost_price > 5.0）以及另一條規則到 stock 表（quantity > 2），但我們還要從 item 表列出所有的 item 表中以上條件為真的行，即使沒有庫存。

我們所要做的僅僅是在左外連接表上結合 ON 條件，通過 WHERE 條件在表連接完成後限制返回的所有行。

在 stock 表上的條件是外連接的一部分。我們不想限制沒有數量的行，所以我們將它作為 ON 條件的一部分：

```
ON i.item_id = s.item_id AND s.quantity > 2
```

對於 item 的條件，將應用於所有的行，我們使用一個 WHERE 從句：

```
WHERE i.cost_price > cast(5.0 AS numeric(7,2));
```

把他們放到一起，我們得到了這個：

```
bpsimple=# SELECT i.item_id, i.cost_price, s.quantity FROM item i
bpsimple-# LEFT OUTER JOIN stock s
bpsimple-# ON i.item_id = s.item_id AND s.quantity > 2
bpsimple-# WHERE i.cost_price > cast(5.0 AS numeric(7,2));
 item_id | cost_price | quantity
---------+------------+----------
       1 |      15.23 |       12
       2 |       7.45 |
       5 |       7.54 |        3
       6 |       9.23 |
       7 |      13.36 |        8
      11 |      19.73 |
(6 rows)

bpsimple=#
```

解析

我們使用 LEFT OUTER JOIN 從 item 表獲得所有的值，選擇性地連接到 stock 表中存在的且 quantity 大於 2 的行。這為我們得出了一組記錄，這些記錄從 item 表產生，但是 quantity 列（從 stock 表）要麼為 NULL 要麼是存在於 stock 表中且 quantity 大於 2。WHERE 從句然後被執行，它僅允許那些售價（從 item 表獲得）大於 5.0 的行通過。

摘要

本章開始的時候學習了在 SQL 使用的聚集函數，用於從許多行中查詢一個值。特別是，我們遇到了 count(*)函數，你會發現它可以廣泛用於確定一個表中的行數。我們然後碰到了 GROUP BY 從句，它允許我們選擇一組記錄應用到聚集函數上，其後跟隨著 HAVING 從句，它允許我們約束包含特定聚集值的記錄。

下一步，我們學習了子查詢，在這裡，我們在一個查詢中使用了另一個查詢的結果。我們看到了一些簡單的示例，並接觸到了一個更難的查詢——相關子查詢——相同的列同時還出現在子查詢中。

之後，我們還簡短地學習了 UNION 連接，它允許我們結合兩個查詢的輸出到一個簡單的結果集。雖然它不是很常用，它有時候確實非常有用。最後，我們遇到了外連接，一個非常重要的功能，可以允許我們對兩個表執行連接，即使在連接到第二個表失敗的情況下，也能從第一個表檢索出記錄。

在本章，我們覆蓋了 SQL 的一些難點。你已經看到了大量的 SQL 語法，所以如果你看到一些系統中存在的高級 SQL，你理所當然應該理解它正在做什麼。如果有些部分有點不清楚也不要著急。最好的真正理解 SQL 的方法是使用它，廣泛地使用它。安裝 PostgreSQL，安裝測試資料庫以及一些示例資料並做實驗。

在下一章，我們將學習一些資料類型的細節，建表，以及其他一些建立你自己資料庫需要的資訊。

第八章資料定義

到現在為止，我們致力於研究 PostgreSQL 的工具和資料操作。雖然我們在本書老早的地方就建立了一個資料庫，我們只是膚淺地瞭解了建表和 PostgreSQL 中的資料類型。我們通過使用主鍵和定義很少的幾個不允許 NULL 值的列來簡單定義我們的表。

在資料庫中，我們最關注的就是資料的品質。針對資料執行非常嚴格的規則，通過資料庫強制限制在最低級別，是我們可以用於保持資料一直的最好措施之一。這也是真正的資料庫區別於簡單的索引檔、試算表等的功能之一。

在本章，我們將更深入地學習 PostgreSQL 中的資料類型以及如何操作它們。然後我們將學習如何管理表，包括如何使用約束，這使我們能夠大大加強在添加資料到資料庫中的表或從資料庫中的表中刪除資料時的規則。下一步，我們將簡單學習視圖。最後，我們將更深層次地學習外鍵約束並在建立我們更新版本的示例資料庫時使用它們。我們將建立 bpfinal 資料庫，我們將在之後的章節中使用這個示例資料庫。

在本章，我們將覆蓋以下主題：

- 資料類型
- 資料操作
- 表管理
- 視圖
- 外鍵約束

資料類型

在最基本的層面上，PostgreSQL 支援以下類型的資料：

- 布林
- 字元
- 數字
- 時間（基於時鐘）
- PostgreSQL 擴展類型
- 二進位大物件（BLOB）

在這裡，我們將學習它們每一個類型，除了非常少用的 BLOB。如果你對 BLOB 類型感興趣，可以在附錄 F 查看怎麼使用它。

布林資料類型

布林類型可能是最簡單的類型。它只可以存儲兩個值，true 和 false，以及在值未知的時候，存儲 NULL。定義一個布林類型的列的官方類型名為 boolean，但它經常被縮寫為簡單的 bool。

當資料被插入表的布林類型的列中時，PostgreSQL 能非常靈活地解釋將翻譯成 true 和 false 的值。表 8-1 提供了一個可以接收的值的列表以及它們對應的值。除了 NULL，其他的任何值都將被拒絕。和 SQL 關鍵字一樣，它們也是不區分大小寫的；例如'TRUE'也將被翻譯成布林類型的 ture。

表 8-1 指定布林值的方法

翻譯為 true	翻譯為 false
'1'	'0'
'yes'	'no'
'y'	'n'
'true'	'false'
't'	'f'

注：當 PostgreSQL 顯示布林類型列的內容是，它只會顯示 t,f 和一個空格，分別代表 true，false 和 NULL，而不管你設置這個列的值為什麼（'true'，'y'，'t'等等）。因為 PostgreSQL 值存儲這些可能狀態中的一個，不是存儲你實際設置的這個列的值，而僅僅是翻譯後的值。

嘗試：使用布林值

讓我們建立一個簡單的擁有 bool 列的表，然後用一些值做實驗。與其用我們的 bpsimple 資料庫使用我們的“真實”資料做測試，不然建立一個叫 test 的資料庫。如果你使用了第三章中的示例，你應該已經建立了這個資料庫了，只需要連接到它。如果沒有，你可以像下面這樣建立並連接到它：

```
bpsimple=> CREATE DATABASE test;
CREATE DATABASE
bpsimple=> \c test
You are now connected to database "test".
test=>
```

現在，我們建立一個叫 testtype 的表，擁有一個變長的字串的列和一個布林的列，插入一些資料，然後看看 PostgreSQL 存儲了什麼。以下是我們簡短的 psql 會話：

```
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=>
```

讓我們檢查以確認資料被插入了：

```
test=> SELECT * FROM testtype;
 valused | boolres
---------+---------
 TRUE    | t
 1       | t
 t       | t
 no      | f
 f       | f
 Null    |
 FALSE   | f
(7 rows)

test=>
```

解析

我們建立了一個有兩個列的表 testtype。第一個列保存一個字串，第二個保存一個布林值。我們然後插入資料到表中，每次都將第一個值設置為字串，用以提示我們插入的值，第二個為相同的值，但是將被存儲為布林值。我們還插入一個 NULL，來展示 PostgreSQL（不像至少一個商業資料庫）允許在 boolean 類型中存儲 NULL。然後我們取出這些資料，這顯示了 PostgreSQL 如何翻譯每一個我們傳入的作為 true，false 或者 NULL 的值。

字元資料類型

在任何資料庫中，字元資料類型可能是最廣泛使用的類型。有三種字元資料類型，用於表示以下字元變數：

- 單個字元
- 固定長度字串
- 長度可變字串

有很多標準的 SQL 字元類型，但 PostgreSQL 也支援 text 類型，它類似於變長字串，除了我們不需要定義任何上限來限制長度。但是這不是標準的 SQL 類型，所以要小心使用它。標準的類型可以使用 char，char(n)和 varchar(n)定義。表 8-2 列出了 PostgreSQL 的字元類型。

表 8-2 PostgreSQL 字元類型

定義	意義
char	單個字元
char(n)	一組長度固定為 n 的字元，長度不足用空白填充。如果你嘗試存儲一個過長的字串，將會發生一個錯誤。
varchar(n)	一組長度不超過 n 的字元，長度不足也不需要填充。PostgreSQL 擴展了 SQL 標準，允許指定沒有長度的 varchar，這實際上使長度不受限制。
text	實際上是一個長度不受限制的字串，就像 varchar 一樣，只是不需要定義最大長度。這是一個 PostgreSQL 針對 SQL 標準做的擴展。

如果讓你選擇三個標準類型用於存儲字串，你會選哪一個？通常，沒有正式的答覆。如果你知道你的資料庫只會用在 PostgreSQL 中，你可以使用 text 類型，因為它很容易使用且不強制要你確定最大長度。它的最大長度只受限於 PostgreSQL 支持的行的最大大小。如果你使用的是早於 7.1 版本的 PostgreSQL，行的大小限制在 8KB（除非你從源碼重新編譯且改變了它）。從 PostgreSQL 7.1 開始，已經沒有這個限制了。對於 PostgreSQL 7.1 和以後的版本，一個表中任何單個欄位的長度的限制為 1GB，實際上你永遠不需要那麼長的一個字串。

主要缺點是 text 類型不是標準類型。所以，如果有很少的可能你有一天需要移植你的資料庫到 PostgreSQL 之外的資料庫，你應該避免使用 text 類型。通常，我們在本書中沒有使用 text 類型，而更趨向於使用更標準的 SQL 類型 varchar 和 char。

按慣例，char(n)在字串的長度為固定或者行與行之間有稍微變化的時候使用，varchar(n)在字串長度明顯變化的時候使用。這是因為在一些資料庫中，內部存儲定長字串的性能比變長的高很多，即使定長的需要存儲一些不必要的字元。但是在內部，PostgreSQL 使用相同的機制存儲 char 和 varchar 類型。所以，對於 PostgreSQL，使用哪種類型更多依賴於你自己的個人偏好。如果不同行之間的資料的長度明顯不同，可以選擇 varchar(n)類型。還有，如果你不確定長度，可以使用 varchar(n)。

就像 boolean 類型一樣，所有的字串類型可以包含 NULL，除非你明確地定義這個列不允許 NULL 值。

嘗試：使用字元類型

讓我們看看 PostgreSQL 的字元類型如何工作。首先，我們需要刪除我們的 testtype 表，然後我們使用一些不同的列類型重建它：

```
test=> DROP TABLE testtype;
DROP TABLE
test=>
test=> CREATE TABLE testtype (
test(>     singlechar      char,
test(>     fixedchar       char(13),
test(>     variablechar    varchar(128)
test(> );
CREATE TABLE
test=>
test=> INSERT INTO testtype VALUES('F', '0-349-10177-9', 'The Wasp Factory');
INSERT 17871 1
test=> INSERT INTO testtype VALUES('S', '1-85723-457-X', 'Excession');
INSERT 17872 1
test=> INSERT INTO testtype VALUES('F', '0-349-10768-8', 'Whit');
INSERT 17873 1
test=> INSERT INTO testtype VALUES(NULL, '', 'T.B.D.');
INSERT 17874 1
test=> INSERT INTO testtype VALUES('L', 'A String that is too long', 'L');
ERROR:  value too long for type character(13)
test=>
test=> SELECT * FROM testtype;
 singlechar |   fixedchar   |   variablechar
------------+---------------+-----------------
```

```
 F          | 0-349-10177-9 | The Wasp Factory
 S          | 1-85723-457-X | Excession
 F          | 0-349-10768-8 | Whit
            |               | T.B.D.
(4 rows)

test=> SELECT fixedchar, length(fixedchar), variablechar FROM testtype
test-> WHERE singlechar = 'S';
   fixedchar   | length | variablechar
---------------+--------+--------------
 1-85723-457-X |     13 | Excession
(1 row)

test=> SELECT fixedchar, length(fixedchar), variablechar FROM testtype
test-> WHERE singlechar IS NULL;
   fixedchar   | length | variablechar
---------------+--------+--------------
               |      0 | T.B.D.
(1 row)

test=>
```

解析

我們建立了一個有三列的表，每個列對應一個標準的 SQL 類型。列 singlechar 保存單個的字元，fixedchar 列保存剛好 13 個字元長度的字串，variablechar 列保存最長不超過 128 自己的字串。我們然後儲存不同的資料到幾個列中，然後再檢索它們一查看 PostgreSQL 是否正確存儲了資料，雖然是在 psql 的輸出中，你實際上看不到填充的字元。

我們嘗試插入一個超過 fixedchar 列的字串。這產生了一個錯誤，沒有資料被插入。

我們檢索了 fixedchar 列中字串長度不同的行，使用內建的函數 length()來檢查它的大小。我們會在之後章的“用於資料操作的有用函數”小節中學習一些其他的用於運算元據的函數。

注：在 PostgreSQL 8.0 版本之前，length()函數在本例中總會返回 13，因為存儲類型 char(n)是定長的且資料總是由空白填充了的，但是現在，length()函數忽略哪些空格並返回一個更有用的結果。

數字資料類型

PostgreSQL 的數字資料類型比我們現在為止遇到的類型都稍複雜，但是它們也不是特別難理解。我們可以存儲兩類不同的數位到資料庫中，整數和浮點數位。再細分，整數類型有一些子類型，包括 serial 類型（我們已經使用過，用於在一個表中建立一個唯一的值）和不同大小的整數，如表 8-3 所示。

表 8-3 PostgreSQL 整數資料類型

子類型	標準名	描述

Smalll integer	Smallint	一個 2 位元組的符號型整數，可以存儲-32768 到 32767 的數位
Integer	Int	一個 4 位元組的符號型整數，可以存儲-2147483648 到 2147473647 的數位
Serial		和 integer 一樣，除了它的值通常是由 PostgreSQL 自動輸入的。

浮點數據也可以再細分，分為提供通用功能的浮點值和固定精度的數位，如表 8-4 所示。

表 8-4 PostgreSQL 浮點資料類型

子類型	標準名	描述
float	float(n)	支援最少精度為 n，存儲為最多 8 位元組的浮點數。
float8	real	雙精度（8 位元組）浮點數位
numeric	numeric(p,s)	擁有 p 個數字的實數，其中小數點後有 s 位。不像 float，這始終是一個確切的數字，但工作效率比普通浮點數字低。
money	numeric(9,2)	PostgreSQL 特有的類型，但在其他資料庫裡也普遍存在。Money 類型從 PostgreSQL 8.0 開始不贊成使用，且可能在以後版本中取消。你應該使用 number 類型代替。

被分成整數和浮點數據這兩類的原因很容易理解，但 numeric 類型的用途不太明顯。

浮點數據以科學計數法方式，使用尾數和冪存儲。對於 numeric 類型，你可以在執行計算時同時指出精度和準確的存儲的數位。你還可以指出小數點後的數位個數。實際的小數點位置變得非常自由了！

警告：最常見的錯誤時認為 numeric(5,2)可以存儲一個類似於 12345.12 的數字。這是錯誤的。在這裡頭存儲的總歸數位個數只有五個，所以定義為 numeric(5,2)的數字在溢出前最大只能存儲 999.99。

PostgreSQL 一般會捕捉對無法存儲到欄位中的值的插入，所以嘗試插入很大的數位到任何類型的數位列將失敗。

嘗試：使用數位類型

現在，我們可以實驗數字資料類型。首先，我們需要刪除我們的 testtype 表，然後使用一些不同的列類型來重新建立它：

```
test=> DROP TABLE testtype;
DROP TABLE

test=> CREATE TABLE testtype (
test(>     asmallint    smallint,
test(>     anint        int,
test(>     afloat       float(2),
test(>     areal        real,
test(>     anumeric     numeric(5,2)
test(> );
CREATE TABLE

test=> INSERT INTO testtype VALUES(2, 2, 2.0, 2.0, 2.0);
INSERT 17883 1
test=> INSERT INTO testtype VALUES(-100, -100, 123.456789, 123.456789, 123.456789);
INSERT 17884 1
```

```
test=> INSERT INTO testtype VALUES(-32768, -123456789, 1.23456789,
test-> 1.23456789, 1.23456789);
INSERT 17885 1
test=> INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,
test-> 23456789.123456789, 123456789.123456789);
ERROR:  numeric field overflow
DETAIL:  The absolute value is greater than or equal to 10^8 for field with
precision 5, scale 2.
test=>
test=> INSERT INTO testtype VALUES(-32768, -123456789, 123456789.123456789,
test-> 123456789.123456789, 123.123456789);
INSERT 17886 1
test=>
test=> SELECT * FROM testtype;
 asmallint |   anint    |    afloat    |    areal     | anumeric
-----------+------------+--------------+--------------+----------
         2 |          2 |            2 |            2 |     2.00
      -100 |       -100 |      123.457 |      123.457 |   123.46
    -32768 | -123456789 |      1.23457 |      1.23457 |     1.23
    -32768 | -123456789 | 1.23457e+008 | 1.23457e+008 |   123.12
(4 rows)

test=>
```

解析

我們建立了一個表，包含一個短整數列，一個普通整數列，一個浮點數字列，一個實數列，一個精度為 5，小數點後為 2 的數字列。

你會發現 float 和 real 類型的行為非常相似，但是 numeric 列的行為有點不同。Numeric 類型不是存儲接近的數，而是在小數後面進行後超出固定長度的部分進行四捨五入。如果我們存儲太大的資料到其中，INSERT 將失敗。還要注意 float 和 real 也會對數字四捨五入；例如 123.456789 被四捨五入為 123.457。

時間型資料類型

時間型資料類型存儲和時間相關的資訊，在第四章，我們看到了如何控制資料的格式。PostgreSQL 有一系列的與日期和時間相關的類型，如表 8-5 所示，但是我們在本書中通常只限制於 SQL92 標準的類型。

表 8-5 PostgreSQL 的時間資料類型

定義	意義
date	存儲日期資訊
time	存儲時間資訊
timestamp	存儲日期和時間

interval	存儲 timestamp 之間差別的資訊
timestamptz	PostgreSQL 擴展的類型，存儲包含時區資訊的 timestamp

特殊資料類型

由於 PostgreSQL 源於一個用於研究的資料庫系統，PostgreSQL 擁有一些少見的資料類型用於存儲幾何和網路資料類型，如表 8-6 所示。使用 PostgreSQL 的這些任何一種特殊功能都會使一個 PostgreSQL 資料庫的可攜性變得非常的差，所以通常，我們趨向於避免這些擴展。要獲得更多的關於這些類型的資訊，查詢 PostgreSQL 的文檔，在“資料類型”部分。

表 8-5 PostgreSQL 的特殊資料類型

定義	意義
box	矩形盒子
line	一組點
point	一對幾何學的數字
lseg	一條線段
polygon	一條封閉的幾何線
cidr 或 inet	一個 IPv4 的位址，錄入 192.168.0.1
macaddr	以 MAC 位元址（乙太網卡物理位元址）

注：PostgreSQL 也允許你使用 SQL 命令 CREATE TYPE 在資料庫中建立你自己的類型。這通常不需要，而且在一定程度上，它是 PostgreSQL 獨有的。更多的資訊可以在官方文檔中找到。注意建立你自己的類型可能導致資料庫的模式非常限於 PostgreSQL 中使用，因為用戶自訂類型無法被移植。

陣列

PostgreSQL 有另一個獨特的功能：能夠在表中存儲陣列。在 SQL99 標準之前，這不是一個標準的功能，所以在資料庫實踐中很少見。通常，一個陣列需要通過使用一個附加表實現。但是，陣列的能力有時候很有用，特別是當你需要存儲固定數量的重複元素時，而且它非常容易使用。

建立陣列的方法有兩種：傳統的 PostgreSQL 的方法和 SQL99 標準的方法。我們在這裡會簡單介紹這兩種方法。

PostgreSQL 樣式的陣列

要將一個表的列定義為陣列，你可以簡單地在類型後面添加[]；不需要定義元素的個數。如果你使用了大小來定義，PostgreSQL 接受你的定義，但它不強制接受指定數量的元素。

嘗試：使用 PostgreSQL 語法定義陣列

作為一個示例，假設我們決定要有一個雇員表，表中要包含一個指示他們工作日的指示器。通常，我們需要為每一天分配一個列，或者一個單獨的表來儲存工作日。在 PostgreSQL 中，我們可以簡化這個工作，直接存儲工作日的陣列，就像這樣：

```
test=> CREATE TABLE empworkday (
test(>     refcode char(5),
test(>     workdays int[]
test(> );
CREATE TABLE
test=>
```

這建立了一個有兩個列的empworkday表:一個參考字串以及一個叫做workdays的整數陣列。要網陣列列中插入值，我們需要用大括弧分隔符號號包圍由逗號分割的值的列表，就像這樣：

```
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=>
```

我們可以一次性選擇陣列元素的所有值，就像這樣：

```
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=>
```

我們也可以通過給出陣列的索引值來取出單個元素：

```
test=> SELECT workdays[2] FROM empworkday WHERE refcode = 'val02';
 workdays
----------
        1
(1 row)

test=>
```

解析

PostgreSQL 的行為很像傳統的程式設計語言，存儲一個陣列的值，甚至還有不需要指出陣列的大小的好處。如果你選擇整個陣列，PostgreSQL 顯示在花括弧之間的所有的用逗號分隔的值。

有一個需要注意的事情是 PostgreSQL 的陣列中第一個元素的索引值是 1 而不是 0，而很多程式設計語言通常是 0. 如果你嘗試選擇一個不存在的陣列元素，將返回 NULL。

注：PostgreSQL 也允許多維陣列。要知道更多 PostgreSQL 關於陣列的內容，請參考文檔。

SQL99 樣式的數位

在 SQL99 標準中，新的陣列定義語法被提出。這比 PostgreSQL 樣式更明確的一種語法，必須指出元素的個數，而 PostgreSQL 在實現時是不強制執行這個標準的。

嘗試：使用 SQL99 語法的陣列

讓我們定義我們早前的表，使用 SQL99 樣式的定義來做實驗：

```
test=> DROP TABLE empworkday;
DROP TABLE
test=> CREATE TABLE empworkday (
test(>     refcode char(5),
test(>     workdays int array[7]
test(> );
CREATE TABLE
test=> INSERT INTO empworkday VALUES('val01', '{0,1,0,1,1,1,1}');
INSERT 17899 1
test=> INSERT INTO empworkday VALUES('val02', '{0,1,1,1,1,0,1}');
INSERT 17900 1
test=>
test=> SELECT * FROM empworkday;
 refcode |    workdays
---------+-----------------
 val01   | {0,1,0,1,1,1,1}
 val02   | {0,1,1,1,1,0,1}
(2 rows)

test=>
test=> SELECT workdays[2] FROM empworkday WHERE refcode = 'val02';
 workdays
----------
        1
(1 row)

test=>
```

解析

就像你看到的，SQL99 樣式的陣列和 PostgreSQL 樣式的陣列基本一樣。唯一的不同是定義的語法有點不同。

資料操作

PostgreSQL 提供一些機制動作表中的資料。這裡，我們將看到一些內置的函數和一些“神奇”變數。我們還會更仔細地查看 PostgreSQL 添加到表中的 oid 列。

在資料類型之間轉換

我們經常需要在資料庫中轉換資料類型。類型轉換可能很有用且有時候很必要，例如在處理日期和時間的時候。例如，我們可能需要處理從其他系統傳入的以字串方式進入資料庫的日期值。轉換這些字串到日期資料類型將允許我們按日期查詢，有時候將它們作為字串處理無法滿足我們的要求。

> 注：通常，你應該注意類型轉換，因為程式中太多的類型轉換可能意味著資料庫設計中的缺陷。

關聯式資料庫如何進行類型轉換存在很大的差異。PostgreSQL 使用 cast 轉換符：

```
cast(column-name AS type-definition-to-convert-to)
```

另一種更簡潔的雙冒號語法可以用在 SELECT 語句中的簡單的列名所在位置使用：

```
column-name::type-definition-to-convert-to
```

假設我們想要從我們原來的 bpsimple 資料庫中的 orderinfo 表中以 char(10)格式獲取 data 資料，我們可以這樣寫：

```
SELECT cast(date_placed AS char(10)) FROM orderinfo;
```

在我們的 bpsimple 資料庫中執行它，我們可以得到：

```
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=>
```

我們可以用 cast 作為列值，而且我們可以通過提供列標題來給結果命名，就像我們將在下一個示例中一樣。

嘗試：類型轉換

假設我們想產生一個商品項的列表，以最接近的整數美元顯示價格。我們可以簡單地轉換價格為 integer 類型。我們還將查詢“原始”價格，來顯示 PostgreSQL 已經對價格進行了四捨五入。

```
bpsimple=> SELECT sell_price, sell_price::int AS "Guide Price" FROM item
WHERE sell_price > 5.0;
 sell_price | Guide Price
------------+-------------
      21.95 |          22
```

```
       9.95 |          10
      15.75 |          16
      19.95 |          20
      25.32 |          25
      11.49 |          11
(6 rows)
bpsimple=>
```

解析

我們將 sell_price 列轉換為整數（sell_price::int）並給它一個名字（AS "Guide Price"）。我們也可以使用 case 語法寫出這個語句；這兩種轉換格式是可以互換的。

比較這兩列，我們可以發現 PostgreSQL 如何進行四捨五入。在舊版本的 PostgreSQL 中，經常有必要進行資料類型之間的顯式類型轉換。通常，目前的版本會自動作出合理的轉換。

注意並不是所有類型之間都可以進行轉換。例如，你無法將 date 轉換為一個 integer。

用於資料操作的函數

PostgreSQL 提供一些通用功能函數，你可以使用它們來操作列，它們被列在表 8-7 中。可以查看第 10 章獲得更多關於 PostgreSQL 內建函數的資訊。

表 8-7 有用的資料操作函數

函數	描述
length(column-name)	返回一個字串的長度
trim(column-name)	移除字串開始和結尾的空格
strpos(column-name, string)	返回子串在列中的位置
substr(column-name, position, length)	根據指定位置和長度截取子串。第一個字元算作位置 1
round(column-name, length)	根據指定小數點位置四捨五入一個數字
abs(number)	獲得一個數字的絕對值

這些函數的用法和上小節講解的 cast 的用法相同 。以下為一個使用 substr 和 round 函數的示例：

```
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=>
```

魔法變數

有事，我們需要在資料庫中存儲一些關係到當前使用者或者時間等等的資訊，也許是為了實現一個審計線索。PostgreSQL 提供一些“魔法”變數來完成這些。以下為其中一些最常用的：

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER

你可以像使用列名一樣使用它們，你甚至可以直接 SELECT 它們而不需要包含一個表名：

```
bpsimple=> SELECT item_id, quantity, CURRENT_TIMESTAMP FROM stock;
 item_id | quantity |          timestamptz
---------+----------+-------------------------------
       1 |       12 | 2004-10-19 18:03:14.500694+01
       2 |        2 | 2004-10-19 18:03:14.500694+01
       4 |        8 | 2004-10-19 18:03:14.500694+01
       5 |        3 | 2004-10-19 18:03:14.500694+01
       7 |        8 | 2004-10-19 18:03:14.500694+01
       8 |       18 | 2004-10-19 18:03:14.500694+01
      10 |        1 | 2004-10-19 18:03:14.500694+01
(7 rows)

bpsimple=> SELECT CURRENT_USER, CURRENT_TIME;
 current_user |       timetz
--------------+--------------------
 rick         | 18:03:40.862712+01
(1 row)
bpsimple=>
```

這些魔法變數也可以用在 INSERT 和 UPDATE 語句中，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

你也許注意到了，每當我們插入資料，PostgreSQL 使用一個隨意的數字和一個行數作為響應。就像我們在第 6 章提到的，這個數位是一個內部參考資料，它是一個 PostgreSQL 為每一行保存的物件 ID，通常為隱藏的列，名為 oid。

大多數關聯式資料庫沒有這個列，或者即使它們這麼做了，也是對用戶無法訪問的。在 PostgreSQL 中，我們可以通

過顯式地將它的名字放在查詢表的 SELECT 語句中以查看它：

```
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=>
```

你的資料庫中的 oid 列的值通常應該和以上是不同的。你也可以在 ODBC 驅動配置的時候看到關於 OID 的選項。你可以選擇顯式或者隱藏它。

可以通過設置 postgresql.conf 設定檔中的 default_with_oids 標誌為 false 或者在建表的時候顯式的指定 WITHOUT OIDS 來避免 OID 列被添加到你資料庫的中使用者表中。PostgreSQL 8.0 默認是為用戶表建立 OID 列。但是，以後發佈的版本的 PostgreSQL 中可能默認不會建立 OID 列。因此，你永遠都不應該依賴你資料庫中的 OID 列，而且我們建議在你建表語句中避免使用 WITH OIDS 或者 WITHOUT OIDS。

表管理

我們已經瞭解了 PostgreSQL 的資料類型，我們可以在我們建表的時候使用它們。我們在建立我們示例資料庫的時候我們已經看到過 CREATE TABLE 這個 SQL 命令了，但我們將在這裡將更正式地講解它。我們還將探索一些擴展功能，例如臨時表，在建立後改變表，以及在不需要的時候刪除表。

建表

以下為建表的基本語法：

```
CREATE [TEMPORARY] TABLE table-name (
    { column-name type [ column-constraint ] [,...] }
    [ CONSTRAINT table-constraint ]
) [ INHERITS (existing-table-name) ]
```

這看上去有點複雜，實際上它很簡單。第一行通過使用 CREATE TABLE 簡單說明你要建表，之後跟著的是表名和一個左括弧。TEMPORARY 允許你建立一個臨時表，這將在本章較後的“使用臨時表”小節討論。

之後，你列出列名，它的類型以及一個可選的列約束。你基本上可以在你的表中使用無限多的列，每個列用逗號分隔。可選的列約束允許你指定這個列的擴展規則，而且你已經遇到過最常見的例子，NOT NULL。

在列的列表之後是一個表級的約束，它允許你寫額外的表中的資料必須遵守的表級規則，例如一個列的值必須小於某個值。例如，星期幾的列的值必須小於 7。我們將在後面的小節討論列和表的約束。

最後面是 PostgreSQL 的擴展，INHERITS，它允許一個建立一個新表，繼承已經存在的表的列。新表除了包含指定的欄位外，還包含在 INHERITS 關鍵字之後的表的所有欄位。參考 PostgreSQL 文檔獲得更多關於使用 INHERITS 的資訊。

提示：我們強烈建議你存儲你建立資料庫的命令到一個腳本中，並使用這個腳本來建立你的資料庫。如果你需要修改資料庫設計，修改腳本比重建資料庫更輕鬆可靠。這樣你就不需要嘗試回想幾個月（或者是幾天）前你當初建立資料庫的命令了。你會發現當初創建一個腳本並保持更新將為你帶來數倍的回報。

使用列約束

經常需要對表中的列施加一些規則。我們已經有一些簡單的例子，例如確保客戶的姓不為空（NOT NULL）。有時候，我們需要施加一些規則用於管理資料的已知範圍，例如確保保存付款率的列只允許大於一個最小值，或確保一個列是唯一的。對列使用約束允許我們在我們完整的應用的最低級別——資料庫中——執行一些檢查。

對於（hard-and-fast）基礎規則，在資料庫級別強制使用是一個很好的技術，因為它們不依賴於應用程式，所以任何應用程式的錯誤導致的進入的非法值將被資料庫捕獲。在建表時通過編寫一個定義來施加規則比編寫應用程式的邏輯代碼來支援這些規則通常更容易。

表 8-8 顯示了一些對你非常有用的最主要的約束。（也有一些更高級的約束，定義在 PostgreSQL 文檔中。）我們在這裡不會論述 REFERENCES 約束，而是在本章之後的“外鍵約束”小節中討論。

表 8-8 最主要的列約束

定義	意義
NOT NULL	列不允許存儲 NULL 值
UNIQUE	列中存儲的的值必須與其他行都不同。PostgreSQL 允許你在定義為 UNIQUE 列上存儲任意多個 NULL 值。
PRIMARY KEY	實際上是一個 NOT NULL 和 UNIQUE 的組合。每個表只能有一個列被標記為 PRIMARY KEY（但你可以有多個列被同時標記為 NOT NULL 和 UNIQUE）。如果你需要建立一個組合的主鍵（一個包含超過一個列的主鍵），你必須使用一個表級的約束，而不是列級的約束。
DEFAULT default-value	允許你在插入資料的時候提供一個預設值。（嚴格來說，這不是一個約束選項，但把它作為約束來考慮更容易理解。）
CHECK (condition)	當插入或者更新資料的時候允許你進行一個條件檢查。
REFERENCES	約束這個值必須為另一個獨立的表的某個列中的某個值。

除了 PRIMARY KEY，你可以在任何列中使用任何數量的約束。允許對列級約束命名，但當通常沒人去做。

一個需要特備註意的是當一個 NULL 值添加到一個使用了 UNIQUE 約束的列中將發生的事情。PostgreSQL 認為每個 NULL 都是唯一的，所以它允許你在一個定義為 UNIQUE 的列中擁有很多值為 NULL 的行。按照 SQL 標準，應該只允許一個 NULL，所以這是一個稍微背離標準的地方。從論據上看，SQL 標準更符合邏輯，因為如果 NULL 為未知，也就沒有辦法知道他們是否不同，但 PostgreSQL 的實現可能更現實。

嘗試：對列施加約束

最簡單的理解列約束的方法是看它們的行為。讓我們在我們之前建立的 test 資料庫建立一個新表，並使用它來試驗一些約束：

```
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=>
```

你會發現 PostgreSQL 警告我們它已經建立了一些索引來執行 PRIMARY KEY 和 UNIQUE 約束。它還為我們生成了有意義的名字。

現在我們已經建立了一個在列中擁有數種約束的表，我們可以嘗試插入一些資料並查看約束在實際中如何工作：

```
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=>
```

插入失敗了，因為索引 testcolcons_pkey 發現了重複的值。這裡我們需要用到一個常識，叫做 testcolcons_pkey 的索引表示 testcolcons 的主鍵索引。

每個表只允許有一個主鍵；因此，對於叫做 tablename_pkey 的索引不會有歧義。

但是，PostgreSQL 允許我們插入兩行 NULL 值到 colunique 列中（這看上去有點危險）：

```
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=>
```

如果我們使用實際的值，PostgreSQL 將拒絕這條 INSERT 語句：

```
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=>
```

這時候，INSERT 失敗，因為索引 testcolcons_colunique_key 發現一個重複值。我們可以有很多定義為 UNIQUE 的列，所以 PostgreSQL 命名這些索引為 tablename_columnname_key，這可以很清楚地顯示出哪個列導致了這個問題：

```
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=>
```

這次，問題出在 CHECK 約束的 colcheck 列，因為我們嘗試插入一個大於 42 的值。注意這個約束命名為 tablename_columnname_check，所以問題源很容易定位：

```
test=> UPDATE testcolcons SET colunique = 1 WHERE colnotnull = 2;
ERROR:  duplicate key violates unique constraint
"testcolcons_colunique_key"testcolcons_colunique_key
test=>
```

我們無法更新 colunique 的值，因為表中有一行的這個列有了這個值：

```
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=>
```

最後，我們沒有為 coldefault 列提供一個值（注意它沒有在列的列表中給出），我們會發現預設值被使用了。

如果我們想查看一個表上的約束，我們需要通知 psql 來列出它們，使用\d tablename 命令，就像這樣：

```
test=> \d testcolcons
    Table "public.testcolcons"
   Column   |  Type   | Modifiers
------------+---------+-----------
 colnotnull | integer | not null
 colunique  | integer |
 colprikey  | integer | not null
 coldefault | integer | default 42
 colcheck   | integer |
Indexes:
```

```
    "testcolcons_pkey" PRIMARY KEY, btree (colprikey)
    "testcolcons_colunique_key" UNIQUE, btree (colunique)
Check constraints:
    "testcolcons_colcheck_check" CHECK (colcheck < 42)
test=>
```

解析

PostgreSQL 使用多種方法實現約束。但無法控制約束檢查的順序。你獲得的實際錯誤依賴於 PostgreSQL 的內部實現。你確定可以知道的是所有的約束都將在資料存儲到資料庫中之前得到檢查。你也可以第 9 章講解的使用事務來確保對資料庫改變的請求集同時全部執行或者同時沒有被執行。

使用表約束

表約束和列約束非常相似，區別就像它的名字描述的一樣，將應用到表上，而不是一個獨立的列中。有時候，我們需要指定特殊的約束，例如表級的主鍵而不是列級的。例如，我們看看我們的 orderline 表，我們需要使用兩個列 orderinfo_id 和 item_id 一起作為一個鍵來標記一行，因為只有這幾個列的組合才是唯一的。這種類型的約束必須在表級別描述。

四個表級的約束列在表 8-9 中。

表 8-9 最主要的表約束

定義	意義
UNIQUE(column-list)	列的清單中存儲的的值必須與其他行都不同。
PRIMARY KEY	實際上是一個 NOT NULL 和 UNIQUE 的組合。每個表只能有一個列被標記為 PRIMARY KEY，無論是表約束還是列約束。
CHECK (condition)	當插入或者更新資料的時候允許你進行一個條件檢查。
REFERENCES	約束這個值必須為另一個獨立的表的某個列中的某個值。

就像你看到的，表級的約束和列級約束很相似。它們的區別為：

- 表級的約束可以針對不止一個列。
- 表級的約束在所有的列之後列出來。

嘗試：使用表級約束

讓我們看看表級約束如何工作。首先，建立一個有一些約束的表：

```
test=> CREATE TABLE ttconst (
test(>     mykey1 int,
test(>     mykey2 int,
test(>     mystring varchar(15),
test(>     CONSTRAINT cs1 CHECK (mystring <> ''),
test(>     CONSTRAINT cs2 PRIMARY KEY(mykey1, mykey2)
test(> );
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "cs2"
```

```
for table "ttconst"
CREATE TABLE
test=>
```

注意，由於使用了列級約束，PostgreSQL 建立了一個索引來執行主鍵約束。讓我們從插入一些行開始：

```
test=> INSERT INTO ttconst VALUES(1,1,'Hello');
INSERT 19381 1
test=> INSERT INTO ttconst VALUES(1,2,'Bye');
INSERT 19382 1
test=>
```

現在嘗試插入一行違反 mystring 不能為空串規則的資料：

```
test=> INSERT INTO ttconst VALUES(1,2,'');
ERROR:  new row for relation "ttconst" violates check constraint "cs1"
test=>
```

表級 CHECK 約束和列級約束基本上完全一樣，拒絕了這一行，因為字串為空。

現在如果我們嘗試插入一行違反 mykey1 和 mykey2 組合必須為唯一的約束，我們將看到約束 cs2 被執行：

```
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=>
```

當兩個 mykey 的值讀相同的時候，這一行被拒絕了，因為違反了主鍵約束。

解析

就像你看到的，表及約束和列級約束非常相似。通常，如果列級約束夠用，建議使用列級約束。然而，當我們需要在同一個表中同時使用列級和表級約束的時候，例如在我們的 bpsimple 資料庫中，為了一致性，我們建議在所有的表中使用表級主鍵約束。

修改表結構

不幸的是，生活很複雜，無論你如何仔細收集需求並實現你的資料庫，需要修改表設計的那一天總會到來。

在第 6 章我們找到了一個解決這個問題的辦法，針對從已有的表中查詢出來的資料使用 INSERT INTO。我們可以使用以下的步驟：

- 建立一個和存在的表相同的新表。
- 使用 INSERT INTO 來從原始表中生成相同的資料到工作表。
- 刪除原始表。
- 使用舊名稱建立原來的表，但增加我們需要的改變。
- 再次使用 INSERT INTO 從功過表中生成資料到修改後的表。
- 刪除工作表。

很明顯，工作量很大，尤其是如果表包含大量的資料並且被觸發器或者視圖引用，而我們想要做的只是為表增加一個列。由於 PostgreSQL 遵從 SQL 標準，因此允許我們隨時添加、刪除和重命名列；也就是說，即使表中包含資料，你

也可以重命名這個表。

也可以對表添加或者刪除約束以及修改預設值；然而，由於一些特殊原因，這種操作有一些限制。例如，你不可以添加一個約束到包含違反這個約束的資料的表中。

注：在舊版本的 PostgreSQL 中有一個額外的限制，你不能建立一個擁有 NOT NULL 或者 DEFAULT 設置的新列，因為這個表中的資料已經存在了。如果確實有必要，做起來也不是很麻煩：添加這個列而不包含任何約束，更新表中的資料，按後添加需要的約束。從 PostgreSQL 8.0 開始，你可以添加擁有預設值的列，同時包含 NOT NULL 約束，只要你提供一個預設值。

要實現這些改變，我們使用 ALTER TABLE 命令。ALTER TABLE 的語法非常簡單，但有很多變種：

```
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
```

添加到有資料的表中的列將為已有的行存儲 NULL 值到這個列中。

嘗試：修改一個表

在我們看到一些 ALTER TABLE 語句前，讓我們先檢查下已有的 ttconst 表的結構：

```
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，我們添加一個新列：

```
test=> ALTER TABLE ttconst ADD COLUMN mydate DATE;
```

然後，我們重命名新添加的列：

```
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=>
```

現在，讓我們嘗試改變一些約束和其他規則：

```
test=> ALTER TABLE ttconst DROP CONSTRAINT cs1;
ALTER TABLE
test=> ALTER TABLE ttconst ADD CONSTRAINT cs3 UNIQUE(birthdate);
NOTICE:  ALTER TABLE / ADD UNIQUE will create implicit index "cs3" for table "tt
const"
ALTER TABLE
test=> ALTER TABLE ttconst ALTER COLUMN mystring SET DEFAULT 'Hello';
ALTER TABLE
```

下一步，讓我們看看最新的表定義：

```
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，新的規則正確被設置，這和我們在建表的時候就設置它們的效果一樣。

也可以改變列的類型，條件是轉換符合邏輯。這裡，我們轉換一個 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，增加一個 varchar 類型列的大小：

```
test=> ALTER TABLE ttconst ALTER mystring TYPE varchar(32);
ALTER TABLE
test=>
```

最後，我們重命名這個表：

```
test=> ALTER TABLE ttconst RENAME TO ttconst2;
ALTER TABLE
test=>
```

解析

就像你看到的，ALTER TABLE 是一個非常強大的命令，允許你修改存在的表結構，包括列和約束，甚至在它們已經包含資料的時候。

注：低於 PostgreSQL 8.0 的 ALTER TABLE 命令有更多限制，所以如果你在運行在一個很舊版本中，你可能發現一些選項不存在。

改變表結構的能力不應該被用來作為在初期表設計時缺乏對細節關注的藉口。ALTER TABLE 應該通常用於需求變更。在其他時候使用它通常暗示著你的原設計在你實際在資料庫中建表前可能可以被改良。

你應該很警惕通過添加新列不斷改變一個表的結構。新的列通常添加在表的末尾，所以很多不能很好的反應表的邏輯目的。假設我們在建立我們的 customer 表時忘記了一個 title 列，然後使用 ALTER TABLE 添加它。列會添加在末尾，這使 customer 表看上去有些古怪，因為一個人的稱謂被放在電話號碼之後。取而代之，你可能更願意使用以下的過程來添加列：

- 使用臨時的表名新建一個表，按最符合邏輯的順序安排列名。
- 使用 INSERT INTO … SELECT …來複製需要改變的表的資料
- 刪除舊表。
- 使用舊表的名字重命名新表。

你需要特別注意我們將在第 10 章討論的序列生成器和觸發器，在刪除表和重命名表的時候，他們也需要被刪除並重新建立。

刪除表

刪除表非常簡單：

```
DROP TABLE table-name
```

一瞬間，你的表就消失了，當然也包括其中的資料。當然，你要非常小心使用這個命令。

使用臨時表

到目前為止我們看到的所有的 SQL 示例都是通過一個簡單的，偶爾有點複雜的 SQL 語句達到我們的目標的。通常，這是一個好習慣，因為就像我們所說的，SQL 是一個說明性語言。如果你定義了你想要的，SQL 找到最好的方法來為你獲取結果。但是，有時候，做任何事情都用一條 SQL 語句是不可行的或者非常複雜的。在一些情況下，你需要保存臨時結果。

通常，你需要的臨時存儲就是一個表，這樣你就可以存儲很多行資料了。當然，你可以總是建立一個表，處理你的業務，然後刪除這個表，但有可能產生中間表偶爾沒有被刪除的風險，可能是應為你的應用程式有一個錯誤或者僅僅是因為一個直接針對資料庫操作的互動式使用者忘記了。最終的結果是流浪表，一些無用的有奇怪名字的表，遺留在你的資料

庫中。不幸的是，很難弄清楚那些表是用於中間工作的表，可以被刪除，哪些是長期使用的表。

SQL 提供一個非常簡單的方法來解決這個問題：臨時表。當你建表時，你可以使用 CREATE TEMPORARY TABLE（你也可以使用同義詞 CREATE TEMP TABLE）而不是 CREATE TABLE。這個表使用通常的方法為你創建，除了當你的會話結束，你到資料庫的連接斷開後，臨時表會自動刪除。

視圖

當你有一個複雜的資料庫，或者有時候當你有大量擁有不同許可的使用者，你需要建立表的幻象，也就是視圖。讓我們通過一個示例弄清這個概念。

假設我們需要允許別人訪問我們資料倉庫，在我們的資料庫中查找條碼和對應的商品。當前，它們被拆分成兩個表，item 和 barcode。從視圖的觀點上看，我們需要一個簡單的視窗讓別人訪問資料，也許是通過第 5 章中的一些 GUI 工具。我們可以通過使用視圖做到，而不是改變我們的設計。

建立視圖

建立視圖的語法非常簡單：

```
CREATE VIEW name-of-view AS select-statement;
```

之後你可以像查詢表一樣查詢這個視圖。（在寫本文的時候，在 PostgreSQL 中，預設情況下視圖還是唯讀的。）你可以從視圖中查詢資料，就像從表中查詢一樣，而且可以連接到其他表中，也可以使用 WHERE 從句。每次你在視圖中執行 SELECT，資料都會重建，所以資料總是最新的。它不是一個在視圖被建立的時候的凍結的拷貝。

> 注：在一些其他的資料庫中，視圖以及相關表中的資料可以被更新，就像表一樣。

假設我們想建立一個視圖顯示 item 表的簡化資訊。我們只需要得到 item_id，description 和 sell_price。對於的 SELECT 語句如下：

```
SELECT item_id, description, sell_price FROM item;
```

例如我們想要建立這個名為 item_price 的視圖，我們應該這麼寫：

```
CREATE VIEW item_price AS SELECT item_id, description, sell_price FROM item;
```

之後，item_price 就能夠像一個表一樣用在一個 SELECT 語句中。

嘗試：建立一個視圖

回到第 5 章，我們在定義 item 表時在定義價格的時候有點小麻煩。假設我們考慮將 price 定義為 numeric(7,2)是正確的，我們仍然可以保持這個定義，但是可以通過使用一個帶有 cast 的 SELECT 語句的視圖達到目標。

讓我們建立一個 item 表的視圖，通過以下三項修改使用者所見的內容：

- 我們需要隱藏 cost_price。
- 我們只需要提供簡單的物品描述。
- 我們需要隱藏所有的昂貴的物品，例如那些價格在$20 以上的東西。

我們可以簡單地通過建立一個視圖做到，就像這樣：

```
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=>
```

現在，當我們從這個視圖中 SELECT 資料，它的行為就像原來表的列的子集一樣：

```
bpsimple=> SELECT * FROM item_price;
 item_id | description | price
---------+-------------+-------
       3 | Linux CD    |  2.49
       4 | Tissues     |  3.99
       5 | Picture Fr  |  9.95
       6 | Fan Small   | 15.75
       7 | Fan Large   | 19.95
       8 | Toothbrush  |  1.45
       9 | Roman Coin  |  2.45
      10 | Carrier Ba  |  0.00
       2 | Rubik Cube  | 11.49
(9 rows)

bpsimple=>
```

解析

在我們的示例中，我們做了很多事。首先，我們通過轉換 description 列為 description::varchar(10)縮減它到 10 位元組。下一步，我們通過不包含它在列的列表中，隱藏了成本價，同時耍了點小手段，將售價重命名為價格 sell_price AS price 以使這裡沒有線索提示這個表裡還包含了成本價。最後，我們約束視圖返回的行為 WHERE sell_price <= 20.0。（在第 11 章，我們將討論如何使用許可來使普通用戶無法訪問原來的 item 表。）

我們不限制在一個視圖中只使用一個表。如果我們願意，我們可以使用複雜的 SQL 語句來訪問大量的表。

嘗試：從多個表創建一個視圖

讓我們建立一個視圖來解決我們用簡單的方法顯示 item 和 barcode 表，隱藏價格資訊並解決資料分佈在兩個表中的問題。我們將命名這個視圖為 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=>
```

這建立了一個新的視圖，我們可以像使用表一樣用它：

```
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，這和我們敲以下命令完全一樣：

```
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，它對最終用戶隱藏了複雜性。

如果我們想列出我們資料庫中的視圖，我們可以使用\dv 命令。命令\d name-of-view 將描述視圖，運行我們看到使用的 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=>
```

我們建立了一個叫 all_items 的視圖，它的行為像一個表，除了它是通過一些隱藏的 SQL 建立的。

一些人趨向於認為視圖是一個好東西，因而所有的表都應該隱藏在視圖之後。雖然某些級別的資料隱藏通常很有用，

但使用視圖沒有直接使用表有效，特別是如果定義視圖的 SQL 非常複雜且使用超過一個表。將所有的表隱藏在視圖之後的資料庫的性能非常糟糕，而且使用者無法優化他們的 SQL 的性能，也許是因為他們需要的列是在一個在一個視圖中而導致了一個大表的關聯。甚至也許用戶只需要一個列，如果你強制他們使用視圖，他們將執行視圖之後複雜的 SQL，降低性能。雖然視圖對你有用，太多好東西可能反而是有害的。

刪除和替換視圖

要刪除一個視圖，如下操作：

```
DROP VIEW name-of-view
```

不像刪除表，刪除視圖不會影響相關的資料。

如果你想要使用相同的名字和返回列替換一個現存的視圖，你可以在一條語句中使用一個特別版本的語法來完成：

```
CREATE OR REPLACE VIEW name-of-view AS select-statement
```

外鍵約束

我們現在開始講解一個最重要的類型的約束，叫做外鍵約束。在第 2 章，當我們畫我們的 bpsimple 資料庫的圖解的時候，我們有一些表連接或者關聯到其他表。圖 8-1 顯示這個資料庫模式的設計：

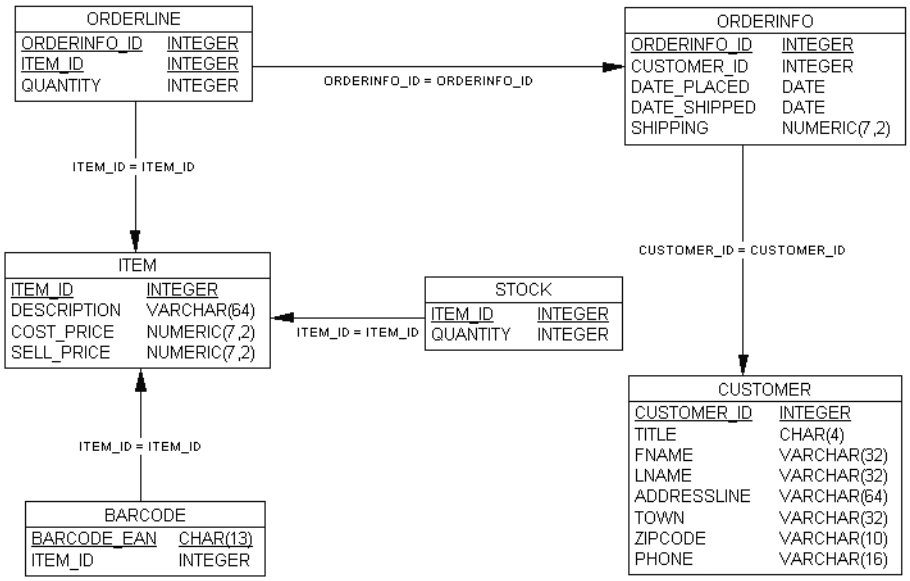


圖 8-1 資料庫模式設計

你可以發現一個表中的列如何關聯到另一個表中的列。例如，orderinfo 表中的 customer_id 關聯到 customer 表中的 customer_id。所以，給出 orderinfo_id，我們可以使用同一行中的 customer_id 找到訂單相關的客戶的名字和位址。我們已經學到了 customer_id 為 customer 表的主鍵；用它可以對 customer 表中的行進行唯一標示。

這裡有另一個重要的術語：orderinfo 表中的 customer_id 就是一個外鍵。這意味著雖然 orderinfo 表中的 customer_id 不是這個表中的主鍵，它關聯到的 customer 表中的列是 customer 表中的唯一鍵。注意這裡沒有反轉的關係——customer 表中沒有哪個列是其他表中的唯一鍵。因此，我們可以認為 customer 表沒有外鍵。

作為我們的資料庫設計的內在結構的一部分，我們的目標是每個 orderinfo 表的 customer_id 也必須出現在 customer 表中。在 orderinfo 表中的 customer_id 叫做外鍵。我們想在資料庫中執行這兩個表之間關係的規則檢查，因為它比在應用程式中進行資料完整性檢查更有效，這就叫做外鍵約束。

當我們建立一個外鍵約束的時候，PostgreSQL 將檢查在特定表中的那個列被定義為唯一的。通常被外鍵引用的是其他表的主鍵。使用外鍵約束是一個很好的方法來確保表間的關係不會由於刪除特定表的（被引用為另一個表的外鍵的）主鍵而被破壞。

一個表可以擁有多個外鍵。例如，在 orderline 表中，orderinfo_id 是一個外鍵，因為它連接到 orderinfo 表的 orderinfo_id 這個主鍵，item_id 也是一個主鍵，因為它連接到 item 表的主鍵 item_id 上。

在 item 表中，item_id 是 item 表的主鍵，因為它唯一標識了一行，它也是 stock 表的外鍵。一個列可以同時是主鍵和外鍵，且這暗指（通常是非強制性的）兩個表的行的一對一的關係。

雖然在我們的示例資料庫中沒有任何資料庫，但實際中也可以允許一對列組合為外鍵，就像 orderinfo_id 和 item_id 組合起來作為 orderline 表的主鍵一樣。

這些關係是我們資料庫中絕對的關鍵。如果我們有一行在 orderinfo 表中的資料無法在 customer 表中找到對應的 customer_id，我們就有一個重要的資料庫整性問題。我們有一個無法找到下訂單使用者的訂單。雖然我們可以使用應用程式的邏輯來確保我們的關係規則，然而就像我們之前說到的，這會更加安全，且通常更容易做到，只需要定義它們為資料庫規則就可以做到。

當發現可以就像我們之前定義約束一樣在列和表中定義這些外鍵關係，你不會覺得奇怪。這些通常在表建立的時候被做到，作為 CREATE TABLE 命令的一部分，使用 REFERENCES 類型的約束。也可以在建表之後再添加外鍵約束，使用 ALTER TABLE table-name ADD CONSTRAINT name constraint-efinition 語法。

我們將前往我們的 bpsimple 資料庫，並建立一個 bpfinal 資料庫，它實現了外鍵約束，來確保資料一致性。

作為一個列的約束的外鍵

這裡是用於定義一個列為另一個表的外鍵的基本語法：

[CONSTRAINT arbitrary-name] existing-column-name type REFERENCES
foreign-table-name(column-in-foreign-table)

為約束命名是可選的，但就像我們之後將看到的，它對理解出錯資訊非常有用。

要在 orderinfo 表的 customer_id 列中定義一個外鍵約束，關聯它到 customer 表，我們一桶使用 REFERENCES 關鍵字和外部表名和列，就像這樣：

```
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)
);
```

我們將很快看到 REFERENCE 約束的效果。

作為一個表的約束的外鍵

雖然你可以在列級定義外鍵約束，我們推薦在表級與主鍵約束一起定義它們。當多個列參與了這種關係的時候，你無法使用列級約束，所以在這種情況下，你必須使用表級約束。

> 提示：最好是使用表級約束，而不要混合私用表級和列級約束。

表級約束和列級約束的格式非常相似，只是它位於所有的列之後：

CONSTRAINT [arbitrary-name] FOREIGN KEY (column-list) REFERENCES
foreign-table-name(column-list-in-foreign-table)

我們可以更新我們 orderinfo 表的定義來聲明一個列 customer_id 為一個外鍵的約束，因為它關聯到 customer 表的主鍵列 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)
);
```

為已有的表添加外鍵約束

在從頭建表開始，讓我們先簡短地回顧下 ALTER TABLE 命令，並查看我們如何使用它追加一個外鍵約束。

讓我們先看看已有的表：

```
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=>
```

然後我們改變這個表，添加一個新的外鍵約束：

```
bpsimple=> ALTER TABLE orderinfo ADD CONSTRAINT
    orderinfo_customer_id_fk FOREIGN KEY(customer_id)
    REFERENCES customer(customer_id);
ALTER TABLE
bpsimple=>
```

讓我們檢查表是否被正確更新：

```
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=>
```

看上去有點複雜，但我們可以明確地在末尾看到我們的新外鍵約束。

建表時添加外鍵約束

是時候為我們的最初的 bpsimple 資料庫帶來飛躍，即開始設計我們最終版本的 bpfinal 資料庫：

```
bpsimple=> CREATE DATABASE bpfinal;
CREATE DATABASE
bpsimple=> \c bpfinal
You are now connected to database "bpfinal".
bpfinal=>
```

現在我們已經準備好使用我們的新知識重建我們的表，使用外鍵約束來從資料庫級別保證參照完整性。

我們必須從我們的 customer 表（相對於我們之前的設計不變）開始，因為如果它不存在，我們無法在 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=>
```

現在我們已經有了 customer 表，我們也可以像以前一樣使用\i 命令來生成它：

```
bpfinal=> \i pop_customer.sql
```

之後，我們可以建立我們的 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=>
```

讓我們迅速檢查下定義：

```
bpfinal=> \d orderinfo
                                  Table "public.orderinfo"
    Column    |     Type     |                              Modifiers
--------------+--------------+-------------------------------------------------------
 orderinfo_id | integer      | not null default nextval('public.orderinfo_orderi
nfo_id_seq'::text)
 customer_id  | integer      | not null
 date_placed  | date         | not null
 date_shipped | date         |
 shipping     | numeric(7,2) |
Indexes:
```

```
    "orderinfo_pk" PRIMARY KEY, btree (orderinfo_id)
Foreign-key constraints:
    "orderinfo_customer_id_fk" FOREIGN KEY (customer_id) REFERENCES customer(cus
tomer_id)

bpfinal=>
```

現在我們可以使用 SQL 腳本重建 orderinfo 表：

```
bpfinal=> \i pop_orderinfo.sql
```

現在我們基本上回到了開始的地方，但有一個非常大的不同：orderinfo 表有一個外鍵約束，用以說明 orderinfo 表中的行的 customer_id 列引用了 customer 表中的 customer_id 列。這意味著如果 customer 表中的行被 orderinfo 表中的資料引用了，則無法刪除它。

嘗試：使用外鍵約束

我們將從檢查 orderinfo 表中的 customer_id 的資料開始：

```
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=>
```

我們現在知道在 orderinfo 表中有五行記錄引用了 customer 表中的客戶資訊的 customer_id 值，引用的客戶的編號為 3，8，13 和 15。只有四個客戶被引用了，因為 orderinfo_id 為 2 和 5 的行指向了同一個客戶。

讓我們從 customer 表中刪除 customer_id 為 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 阻止我們刪除這一行。而且，請注意約束名 orderinfo_customer_id_fk 允許我們更容易定位錯誤源。PostgreSQL 甚至耐煩的告訴我們哪一個 customer_id 鍵的值引起了這個問題，這雖然在本例中非常明顯，但在其他情況下可能更複雜。PostgreSQL 將允許我們刪除不被 orderinfo 表的條目引用的行：

```
bpfinal=> DELETE FROM customer WHERE customer_id = 4;
DELETE 1

bpfinal=>
```

解析

在後臺，PostgreSQL 添加了一些額外的檢查。我們嘗試刪除從 customer 表中的每一行時，資料庫會檢查這一行是否被另一個表引用——在本例中為 orderinfo 表。

命令中任何違反規則的嘗試將被拒絕且資料不會改變。我們仍然可以刪除一個用戶，但我們必須首先確保這個客戶沒有任何訂單。

PostgreSQL 也會檢查我們是否會插入一行引用了不存在的客戶的行到 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=>
```

很容易意識到我們在這裡跨出了一大步。我們使用了非常有效的步驟通過資料庫非常確保了表與表之間的關係。再也不存在 orderinfo 表中的行指向了不存在的客戶的情況。

我們現在可以更新我們原來的建表腳本，對引用其他表的表添加外鍵約束：orderinfo，orderlines，stock 和 barcode 表。唯一有點複雜的表是 orderline 表，它的 orderinfo_id 列引用了 orderinfo 表，而 intem_id 列引用了 item 表。但這不是問題，我們只需要簡單地指定兩個約束，每個列一個：

```
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 中有我們最終版本的建庫腳本。我們將在本書餘下的 buffer 使用這個叫 bpfinal 的資料庫。你也可以在 http://www.xxx.com 下載它。

當你使用這個資料庫的時候，你會發現你也必須按照符合外鍵約束的順序填充表；你現在無法在填充 orderinfo 表之前填充被引用的 customer 表。我們建議按照以下的順序：

- customer
- item
- orderinfo
- orderline
- stock
- barcode

外鍵約束的選項

有可能碰到這樣的情況，我們在 orderinfo 表中的項目引用了 customer 表，但我們需要修改 customer_id。實際情況是，我們無法輕鬆做到這一點，因為如果我們嘗試修改 customer_id（實際上，這是一個壞主意，因為它是一個序列類型的列），orderinfo 表的外鍵約束將阻止這麼做，因為規則說明瞭 orderinfo 中的記錄中的 customer_id 必須總是引用到 customer 表中的項目的 customer_id。

我們也無法修改 orderinfo 表中的 customer_id，因為對應的項目在 customer 表中還不存在，而且我們也無法修改 customer 表中的項目，因為它被 orderinfo 表引用了。

當你需要短時間地在資料修改時違反外鍵約束時，SQL 標準允許兩種方法來解決這種情況，但交易完成之前，你必須恢復資料的完整性：

- 使約束變成延遲的。
- 定義外鍵約束規則時指出如何處理違例。

延遲約束

第一種允許在某些情況下外鍵約束違例的方法是在外鍵約束之後添加 INITIALLY DEFERRED 關鍵字：

```
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
);
```

這改變了外鍵約束試試的方法。通常，PostgreSQL 會在對資料庫進行任何變動前進行外鍵約束檢查。但如果你使用了事物（我們將在下一章遇到）和 INITIALLY DEFERRED，PostgreSQL 允許違反外鍵約束，允許在事務中違反外鍵約束，並且這個違例將在事務結束前被修正。事實上，發生的事情是 PostgreSQL 掛起事務檢查，直到將要完成當前事務。

就像我們將在第 9 章看到的，一個事務是指一組必須要麼都被執行要麼都不被執行的 SQL 命令。因此，我們可以啟動一個事務，更新 customer 表的 customer_id，更新 orderinfo 表相關的 customer_id 的值，提交事務，那麼 PostgreSQL 會允許這麼做的。它所要做的就是在事務完成的時候檢查是否符合約束條件。

注：當然，你也可以只使用 DEFERRED 關鍵字，但在這種情況下，你需要使用命令 SET CONSTRAINTS ALL DEFERRED，這樣 PostgreSQL 預設只在事務結束的時候檢查 DEFERRED 約束。要獲得 SET CONSTRAINTS 選項的更多細節，請參考線上文檔。

ON UPDATE 和 ON DELETE

另一個解決方案是在外鍵約束規則中指定在兩種情況下發生違例的處理方法：UPDATE 和 DELETE 操作。可能的兩種動作為：

- 我們可以從有主鍵的表 CASCADE 變動。
- 我們可以使用 SET NULL 將列置為 NULL，因為它不會指向到主表。

以下為一個示例：

```
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
);
```

本例告訴 PostgreSQL，如果我們刪除 customer 表中的一個被 orderinfo 表引用的 customer_id 所在的行，則 orderinfo 表中的相應行也將被自動刪除。這也許是我們特意選定的做法，但通常情況下這是一個非常危險的做法。最好的辦法是確保應用程式按照正確的順序刪除行，所以我們應該確保在刪除客戶項前已經沒有這個客戶的訂單了。

SET NULL 選項通常和 UPDATE 和 DELETE 語句一起使用。它看起來應該是這樣：

```
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
);
```

這裡的意思是如果 customer 表中被引用的 customer_id 被刪除，則 orderinfo 表中相應的列將被置 NULL。

你會發現在我們的表中，這無法工作。我們定義了 customer_id 為 NOT NULL，所以它無法被更新成 NULL 值。我們這麼做的原因是我們不想 orderinfo 表中的 customer_id 的值可能為 NULL。畢竟，沒有客戶的訂單意味著什麼呢？可能是出錯了。

這些選項可以被組合，所以你可以寫一下的語句：

```
ON UPDATE SET NULL ON DELETE CASCADE
```

注意：使用 ON UPDATE 和 ON DELETE 要非常小心。強制應用程式開發人員按照正確的順序編寫 UPDATE 和 DELETE 代碼並使用事務會更安全，而不要在其他表發生變化時使用 CASCADE DELETE 或者突然將某個列的值存儲為 NULL。

在第 10 章，我們將看到如何使用觸發器和存儲過程來實現相同的效果，但它會給我們對其他表的變動帶來更好的控制。

摘要

在本章我們涵蓋了大量的內容。我們從更正式地查看 PostgreSQL 支持的資料類型開始，尤其是常用的 SQL 標準類型，但也提及了一些 PostgreSQL 的不常用的擴展類型，例如陣列。然後我們學習了如何操作列資料——轉換資料類型，使用資料的子串，以及通過 PostgreSQL 的“神奇”變數訪問資訊。

之後我們學習了表的管理，集中盡力關注一個重要的內容：約束。我們知道了有兩種方法定義約束：針對單一的列和在資料庫級別。即使是簡單的約束也可以說明我們在資料庫級別確保資料的完整性。

之後，我們看到如何使用視圖來創建一個表的“幻象”。視圖可以為使用者訪問資料提供一個簡單的方法，也可以隱藏不想被其他人看到的一些資料。

我們最後的主題是最重要的一種約束：外鍵。它允許我們正式地在資料庫中定義不同表之間的關係。最重要的是，它們允許我們強制執行這些規則，例如確保我們無法刪除一個在其他表中有訂單資訊關聯的客戶的客戶資訊。

在學習如何確保資料庫的參照整性後，我們建立了一個更新的資料庫設計，bpfinal，我們將在本書剩餘的部分使用它。

在下一章，我們將討論事務和鎖，它們在處理多個使用者需要同時訪問一個資料庫時非常有用。

第九章事務和鎖

到目前為止，在本書中我們一直避免任何對 PostgreSQL 在多用戶方面的深入討論，簡單的說明理想化的觀點，和其他優秀的關聯式資料庫一樣，PostgreSQL 隱藏了支持多用戶併發的細節。它只是簡單地提供了一個快速和高效的資料庫伺服器，對它的每個用戶端提供服務，讓每個用戶端覺得是進行了獨佔的訪問。特別是一些小的羽量級負載的資料庫，這種理想化的觀點在實際中基本上得以實現了。然而，事實上是，PostgreSQL 雖然非常很強大，但也不會變魔術，為每個使用者實現隔離需要幕後大量的工作。

在本章，我們將從兩個方面瞭解資料庫對多使用者的支援：事務和鎖。事務允許你收集許多不相關的資料庫變動到一個工作單元。鎖在不同使用者同時改變資料庫時避免衝突。

在本章，我們將涵蓋以下主題：

- 事務的組成
- 在單使用者資料庫中使用事務的好處
- 多使用者情況下的事務
- 行鎖和表鎖

什麼是事務

就像我們在前一章說的，理想情況下的，你應該通過一個簡單的說明性語句描述資料庫的變動。但是，在現實世界的應用程式中，很快就會出現你需要對資料庫做一系列無法用單個 SQL 語句描述的改動。雖然他們無法在一個語句中完成，你仍然需要所有對資料庫的改變都是正確的。如果這組變動中的任何一部分出錯，那麼資料庫就不應發生任何變動。換句話說，你需要執行一組不可分割的工作，它需要多個 SQL 語句被執行，且要求要麼所有的 SQL 語句都執行成功，要麼沒有一條被執行。

關於事物最經典的例子是在兩個銀行帳戶間轉帳，也許帳戶存在於資料庫中的不同表之間，因此一個帳戶被計入借方，另一個被計入貸方。如果你將一個帳戶計入借方後，由於某些原因無法將另一個帳戶計入貸方，你必須將錢退還給第一個帳戶，就像剛才扣減的操作沒有發生過一樣。如果錢在不同帳戶間轉移時會偶爾消失，銀行也就無法繼續經營了。

基於 ANSI SQL 的資料庫，包括 PostgreSQL 在內，通過事務做到這種要麼成功要麼失敗的任務。一個事務是一個不可拆分的工作的邏輯單元。

將資料改變分組到邏輯單元

工作的邏輯單元是什麼意思呢？它是一組對資料庫的邏輯改變，他們要麼必須都被執行，要麼必須都失敗，就像剛才示例中的在帳戶之間轉帳的例子。PostgreSQL 中，這些改變由四個關鍵語法來控制：

- BEGIN 開始一個事務
- SAVEPOINT savepointname 告訴伺服器需要記住事務的當前狀態。這個語法只能在 BEGIN 之後且在 COMMIT 或者 ROLLBACK 之前；也就是說，必須在一個事務中。
- COMMIT 表明所有的事務的元素都完成了，現在需要對所有的並行的事務和子事務進行持久化並對使其結果（在本事務之外）可訪問。
- ROLLBACK [TO savepointname] 表明這個事務將被放棄，SQL 事務中所有對資料的改變將被取消。資料庫給所有使用者的表現是從 BEGIN 開始的任何改變都沒有發生，且事務被關閉。另一個帶 TO 從句的版本允許我們

回滾到一個命名的保存點，且不完成這個事務。

注意：ANSI SQL92 表中沒有定義 BEGIN 這個 SQL 語法。它定義了事務為自動開始的（因此這個語法是多餘的），但這在很多關聯式資料庫中是一個非常常見的擴展。SQL99 添加了語句 START TRANSACTION，它的效果和 BEGIN 一樣。PostgreSQL 從 7.3 開始就像支援 BEGIN 語法一樣支援這個新的語法，但是我們保留了 BEGIN 語法，因為當前它更通用。

併發多使用者訪問資料

事務的第二方面是任何資料庫的事務都將從同時在資料庫中發生的其他事務隔離。在理想的情況下，對於每個事務，感覺就是它擁有對資料庫的獨佔訪問。不幸的是，就像我們將在本章靠後的內容所說的，當我們面對多使用者情況下的事務時，必須採取一些折中的方案以保證實際應用中中達到高性能的目的。

讓我們看另一個需要事務的不同例子。假設你想連線預訂一張機票。你查找你想要的航班並發現一張機票。雖然你不知道，但這是這般飛機上的最後一張機票。當你在輸入你信用卡的細節的時候，同一航班上另一位元客戶也嘗試獲得這張機票。你還沒有支付你的這張機票時，另一個人在你仍在輸入你的信用卡資訊的時候看到了這個空位並使用他的會員帳號預定了它。之後你提交並購買了這張機票，而且因為系統知道在你開始你的事務之前剩餘一張座位，它錯誤地認為那張座位還存在且扣了你的信用卡的費用（當然，航空公司有更成熟的系統避免這種很基礎的訂票錯誤問題，但本例只是為了說明這個原理）。

你斷開連接，確信你成功預訂座位，甚至也許會去檢查你的信用卡被成功扣費。但事實上，你購買了一個不存在的座位。在你事務被處理的時候，已經沒有空位了。

預訂流程的代碼看起來就像這樣：

```
查看是否有座位
如果有，給客戶建議一個座位。
如果客戶接受建議，則請求輸入信用卡號。
委託銀行進行信用卡交易。
從卡中扣錢。
分配座位。
根據被購買的座位數減少剩餘的座位數。
```

任何時候如果只有一個使用者使用這個系統，這個順序事件的過程是完美的。問題是這裡有兩個客戶。真正發生的事情由表 9-1 描述。

表 9-1 重疊的事件

客戶 1	客戶 2	飛機上剩餘的座位數
查看是否還有座位		1
	查看是否還有座位	1
如果有，建議座位給客戶		1
	如果有，建議座位給客戶	1
如果客戶接受建議，要求輸入信用卡號或會員帳號		1
	如果客戶接受建議，要求輸入信用卡號或會員帳號	1
輸入信用卡號碼	輸入會員帳號	1
委託銀行進行信用卡交易		1
	檢查會員帳號是否存在	1
	在新事務中更新帳戶	1

扣信用卡	分配座位	1
分配座位	根據被購買的座位數減少剩餘的座位數	0
根據被購買的座位數減少剩餘的座位數		-1

我們該如何解決這種訂票應用程式的問題呢？我們可以優化到在最接近扣錢的時候重新檢查是否還存在座位，但是無論我們怎麼檢查，必然會發生“檢查座位是否存在”的步驟和“扣錢”的步驟是分離的，即使只間隔很短的時間。

我們可以走到另一個極端去解決這個問題，同時只允許一個使用者訪問訂票系統，但是這個性能會極低，可能會導致客戶流失。

從應用程式術語上說，我們擁有的是一段臨界代碼——一小段需要獨佔訪問資料的代碼。我們可以使用信號燈或者類似的技術寫我們的程式，來管理對臨界代碼的訪問。這可能需要每個訪問資料庫的應用程式使用信號燈。然而，比起編寫應用程式邏輯，通常資料庫處理起這個問題來更容易。

從資料庫術語上說，我們這裡碰到的就是一個事務——從檢查座位是否存在到從信用卡或者帳戶扣費再到分配座位這一組資料操作動作，這些必須在一個單元內完成。

ACID 規則

ACID 是一個常用的縮寫用於描述事務必須擁有的四個屬性：

Atomic（原子性）：一個事務，即使是在資料庫中的一組不相干的操作，必須在同一個單元內發生。一個事務必須一次性發生，沒有子集或者無意識的重複動作。在我們的銀行示例中，轉帳必須是原子的。從一個帳戶扣款並往另一個帳戶存款雖然是兩個獨立的動作但必須同時發生，即使需要更多連續的 SQL 語句。

Consistent（一致性）：在事務結尾的時候，系統必須保持一致狀態。我們在第 8 章接觸到了這個，當時我們看到我們可以定義一個延期約束；換句話說，約束應該指在事務完成的時候被檢查。在我們銀行的示例中，在事務結尾的時候，所有的帳戶必須如實反應被扣款和存款。

Isolated（隔離性）：這意味著對於每個事務，無論資料庫中有多少事務在運行，都應該表現出不依賴於其他任何事務。在我們的機票預訂示例中，交易處理的兩個併發的用戶必須表現出他們都是對資料庫進行獨佔使用的。在現實中，我們知道在多使用者資料庫中，如果我們想讓它符合這種合理的行為是不現實的，實際上這是現實世界中的實際問題中可以影響我們理想資料庫最為顯著的地方之一。我們將在本章稍後的“多使用者下的事務”小節中討論事務的隔離問題。

Durable（持久性）：一旦事務完成，必須保持完成狀態。一旦在帳戶之間成功轉帳，必須保持轉帳完成的狀態，即使在意外情況下運行資料庫的主機掉電。在 PostgreSQL 中，和其他大多數關係型數據庫一樣，可以通過一個事務日誌檔做到這一點，就像下一節所講解的。事務的持久性不需要使用者幹預。

事務日誌

就像在上一節提到的，事務日誌檔被資料庫內部使用以確事務的持久。事務日誌檔的工作方式很簡單。當事務執行後，改動不僅僅是寫入資料庫，還被寫入日誌。一旦事務完成，一個標記被寫入用以說明事務已經完成，日誌檔的資料被強制持久化存儲，所以它是安全的，即使資料庫伺服器崩潰。資料庫伺服器可能因為某些原因在事務過程中崩潰，那麼在伺服器重啟時，它有能力自動確保完成的事務被正確反映到資料庫（通過重做（ROLLING FORWARD）在事務日誌中但未進入資料庫的事務）。伺服器關機時在進行中的事務的改變不會體現在資料庫中。

PostgreSQL 維護的事務日誌不僅記錄所有的對資料庫的改動，也記錄如何撤銷它們。很明顯，這個檔會很快變得很大。一旦一個 COMMIT 語句在一個事務中執行，PostgreSQL 就知道不再需要存儲“撤銷”資訊了，因為資料庫的改動已經是不可反轉的了，至少對於資料庫是這樣的（應用程式可以執行附加代碼來撤銷改變）。

PostgreSQL 使用了一種技術，讓資料在寫入磁片中的表中之前先寫入事務日誌檔，因為它知道一旦資料被寫入日誌檔，它可以通過日誌恢復表中資料的狀態，即使系統可能在真實資料檔案被修改前系統發生故障。這被叫做預寫日誌（Write Ahead Log，WAL），對此感興趣的讀者可以在 PostgreSQL 問的行中找到更多細節資訊。

單使用者情況下的事務

在我們學習事務更複雜的方面和資料庫面對多使用者併發操作中它們的行為之前，我們需要先看看它們在單使用者情況下的行為。即使在這種非常簡單的工作模式中，使用事務都是非常有用的。

使用事務最大的好處是它們允許你執行很多 SQL 語句，然後在之後的步驟，如果你確定需要，允許你撤銷你剛才做的工作。另外，如果你的某條 SQL 語句失敗，你可以撤銷你的工作到預先定義的時刻。

使用事務，應用程式不需要關心如何存儲對資料庫的改變並撤銷它們。只需要簡單地高速資料庫引擎一次性撤銷所有的改變。邏輯上說，整個順序由圖 9-1 描畫。

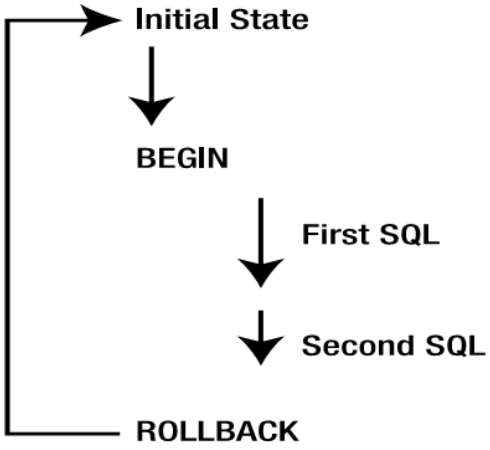


圖 9-1 回滾一系列改變

如果你確定圖 9-1 中在“Second SQL”後你所有對資料庫的改動都是有效的，而且你希望應用它們到資料庫以便他們能夠持久，那麼你需要做的就是用 COMMIT 語句替換 ROLLBACK 語句，如圖 9-2 所示。

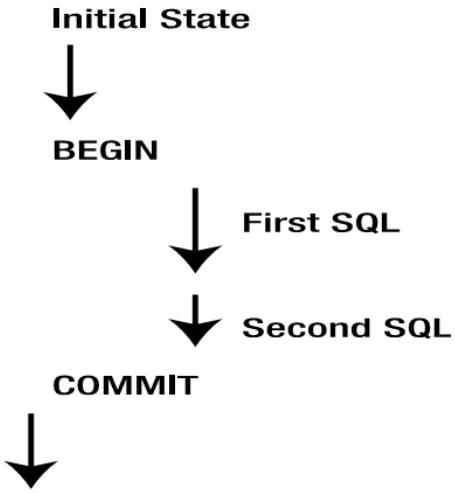


圖 9-2 提交一系列改變

在 COMMIT 後，所有對資料庫的改變都被提交，且可以被認為持久寫入到資料檔案中，所以它們不會因為掉電或者應用程式錯誤而丟失。

嘗試：執行一個簡單的事務

讓我們嘗試一個簡單的事務，在其中我們改變一個表的一行，然後使用 ROLLBACK 語句來取消改變。我們將使用 test 資料庫來做這個實驗。首先，連接到 test 資料庫（如果它不存在，只需要使用 CREATE DATABASE test 命令），然後建立一對簡單的表進行實驗：

```
bpfinal=> \c test
You are now connected to database "test".
test=> CREATE TABLE ttest1 (
test(>  ival1 integer,
test(>  sval1 varchar(64)
test(> );
CREATE TABLE
test=> CREATE TABLE ttest2 (
test(>  ival2 integer,
test(>  sval2 varchar(64)
test(> );
CREATE TABLE
test=>
```

現在我們可以嘗試一個簡單的事務了：

```
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
INSERT 17784 1
test=> BEGIN;
BEGIN
test=> UPDATE ttest1 SET sval1 = 'Dave' WHERE ival1 = 1;
UPDATE 1
test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;
 sval1
-------
 Dave
(1 row)

test=> ROLLBACK;
ROLLBACK
test=> SELECT sval1 FROM ttest1 WHERE ival1 = 1;
 sval1
-------
 David
(1 row)

test=>
```

解析

我們最初插入了一行資料，存儲的名字為“David”。然後我們使用 BEGIN 命令開始了一個事務。下一步，我們更新這一行的 sval1 列，設置名字為“Dave”。當我們在這一行上執行一個 SELECT，它顯示資料被改變了。我們然後調用 ROLLBACK。PostgreSQL 使用它的內部事務日誌撤銷從 BEGIN 開始的改變，所以下一次我們再 SELECT 這個行的時候，我們的改變被回滾了。

有趣的是，如果我們使用另一個 psql 會話並在更新 David 為 Dave 之後但在執行 ROLLBACK 之前立即查詢資料庫，我們會發現依舊是 David 在資料庫中。這是因為 PostgreSQL 從未提交的資料庫資料更新中隔離了使用者，除了做出改變的用戶。我們將在本章之後的“多使用者情況下的事務”小節深入討論。

包括多個表的事務

事務不限制於在一個表中或者只是更新資料。讓我們看一個更複雜的包括多個表且包含 UPDATE 語句和 INSERT 語句的示例。

嘗試：在多個表中執行事務

讓我們實驗事務在多個表上的效果。首先，確保兩個表都是空的，然後插入一行到第一個表中：

```
test=> DELETE FROM ttest1;
DELETE 1
test=> DELETE FROM ttest2;
DELETE 0
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
INSERT 17793 1
```

現在開始一個事務並做出一些改變：

```
test=> BEGIN;
BEGIN
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');
INSERT 17794 1
test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;
UPDATE 1
test=> SELECT * FROM ttest1;
 ival1 | sval1
-------+--------
     1 | Robert
(1 row)

test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+--------
    42 | Arthur
```

```
(1 row)
```

現在執行一個 ROLLBACK 並查看效果：

```
test=> ROLLBACK;
ROLLBACK
test=> SELECT * FROM ttest1;
 ival1 | sval1
-------+-------
     1 | David
(1 row)

test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+-------
(0 rows)

test=>
```

解析

ROLLBACK 導致 INSERT 語句添加的資料被移除，對 ttest1 表的 UPDATE 被撤銷。這演示了一個事務將一些改變合併為一組可以跨多個表工作的情況。

事務和保存點

上一個示例使用了基本的事務語法，這也是大多數應用程式需要的。然而，保存點可以對你在事務中需要回滾到某個特殊的點的情況下非常有用。這需要擴展版本的事務語法，這帶有一個命名的保存點和 ROLLBACK TO 命令。

如果我們需要回滾一個事務中的一些操作，我們可以建立一個我們待會兒可以回滾回去的命名的保存點，而不是回滾到我們從 BEGIN 語句開始的地方。圖 9-3 描繪了這個場景。

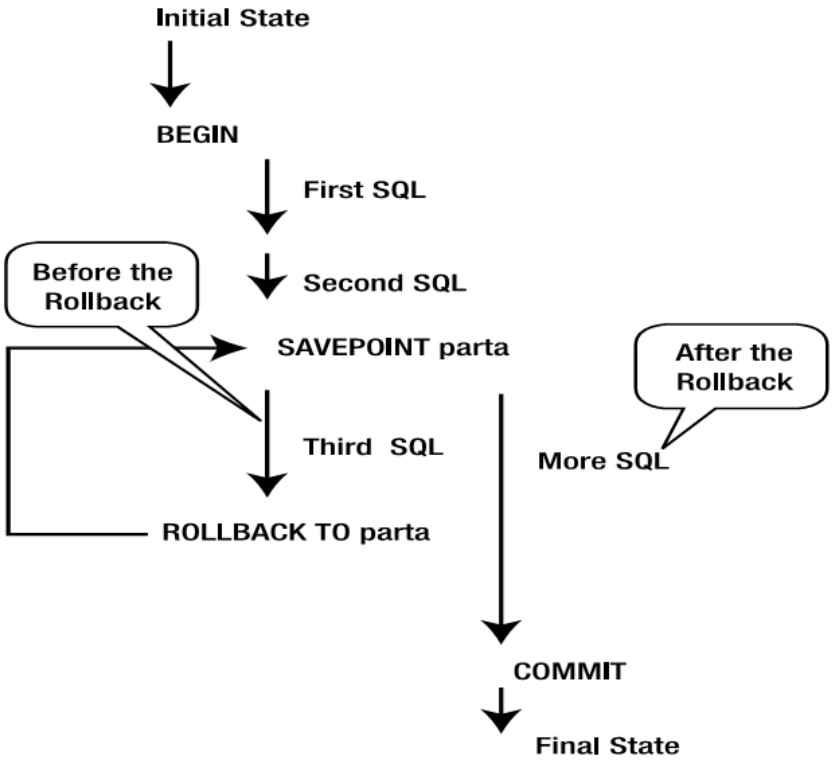


圖 9-3 使用一個保存點

在圖 9-3 的示例中，我們通過執行 BEGIN 語句開始我們的事務，然後執行兩個 SQL 語句。之後我們建立一個叫做 parta 的保存點，然後執行了第三條語句。我們接著執行 ROLLBACK TO parta 語句，它實際上撤銷了第三條 SQL 語句的效果。我們之後可以在執行 COMMIT 來實現資料庫改變的持久化前執行一些其他的 SQL。

嘗試：使用保存點

讓我們看看保存點的實際運用。可以給保存點任何名字；我們在這裡使用 first，但實際上我們可以叫他 Tux，Getreidegasse 或者其他任何名字。

```
test=> DELETE FROM ttest1;
DELETE 1
test=> DELETE FROM ttest2;
DELETE 0
test=> INSERT INTO ttest1 (ival1, sval1) VALUES (1, 'David');
INSERT 17795 1
test=> BEGIN;
BEGIN
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (42, 'Arthur');
INSERT 17796 1
test=> SAVEPOINT first;
SAVEPOINT
test=> UPDATE ttest1 SET sval1 = 'Robert' WHERE ival1 = 1;
```

```
UPDATE 1
test=> SELECT * FROM ttest1;
ival1 | sval1
-------+--------
     1 | Robert
(1 row)

test=> ROLLBACK TO first;
ROLLBACK
test=> SELECT * FROM ttest1;
 ival1 | sval1
-------+-------
     1 | David
(1 row)

test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+--------
    42 | Arthur
(1 row)

test=>
```

此時我們依舊在事務中，所以我們依舊可以回滾到初始的 BEGIN 狀態：

```
test=>
test=> ROLLBACK;
ROLLBACK
test=> SELECT * FROM ttest1;
 ival1 | sval1
-------+-------
     1 | David
(1 row)

test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+-------
(0 rows)

test=>
```

此時，通過調用 ROLLBACK，事務回到了初始的 BEGIN 狀態，事務此時已經完成了，我們不能調用另一個 ROLLBACK 或者 COMMIT，直到另一個 BEGIN 語句被執行：

```
test=> INSERT INTO ttest2 (ival2, sval2) VALUES (99, 'Chris');
INSERT 17797 1
test=> COMMIT;
WARNING:  there is no transaction in progress
```

```
COMMIT
test=>
```

而且，一旦我們通過 COMMIT 指定事務完成，則改變都被永久寫入資料庫，不再有辦法回到以前：

```
test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+-------
    99 | Chris
(1 row)

test=> BEGIN;
BEGIN
test=> UPDATE ttest2 SET sval2 = 'Gill' WHERE ival2 = 99;
UPDATE 1
test=> COMMIT;
COMMIT
test=> ROLLBACK;
WARNING:  there is no transaction in progress
ROLLBACK
test=> SELECT * FROM ttest2;
 ival2 | sval2
-------+-------
    99 | Gill
(1 row)

test=>
```

解析

在本例中，保存點允許我們回滾到一個事務的中間點或者回滾到事務的開始點。一旦一個 ROLLBACK 被執行，資料庫看上去就好像被回滾掉的改變從未發生過一樣。一旦一個事務被提交，則它無法再通過 ROLLBACK 撤銷。、

事務的限制

雖然事務工作起來非常完美，但它也有一些限制。這涉及到嵌套、大小和持續時長。

嵌套

在 PostgreSQL（或大多數其他的關係型數據庫中）你不能嵌套事務。在 PostgreSQL 中，如果你嘗試在一個事務中執行一個 BEGIN 語句，PostgreSQL 將產生一個警告消息，告訴你已經有一個事務在進行中。

一些資料庫默默地接受多個 BEGIN 語句。COMMIT 或者 ROLLBACK 語句將只針對第一個 BEGIN 有效，因此雖然看上去事務被嵌套了，但實際上，隨後的 BEGIN 語句被忽略了。

事務大小

建議儘量使用小事務。就像我們將在本章後面看到的，PostgreSQL（以及其他關係型數據庫）為了確保事務從其他使用者隔離出來，必須做大量的工作。這樣做的結果是在事務中對資料庫的調用部分頻繁需要加鎖，以確保事務被隔離。因此，你需要確保每個事務不會比它的實際需要大。在每個事務中包含大量的不必要的事務將導致在資料庫中產生大量的鎖，這將影響性能和其他使用者對資料庫的訪問。我們將在本章靠後的“鎖”小節討論鎖的細節。

事務持續時長

事務不應該長時間保持打開狀態。雖然 PostgreSQL 自動為你鎖定資料庫，但是長時間運行的事務通常防止其他使用者訪問由這個事務產生的資料，直到事務被提交或者回滾。因此，在需要與使用者進行對話的時候你應該避免先啟動事務。建議先從使用者那裡收集所有需要的細膩，然後在一個事務中處理這些資訊，避免不可預期的用戶回應時間。

假設一個使用者一早在終端前使用一個蹩腳的程式啟動了一個事務，然後讓這個事務伴隨著使用者對資料庫做了大量修改跑了一整天。隨著使用者在資料庫中做工作，越來越多的東西被鎖住，等待這些改變被提交。如果這個使用者只在一天的結尾才提交資料，其他使用者訪問資料的能力將被嚴重壓縮，而且整個應用程式可能在需要多使用者的情況下變得不可用。

你還應該注意儘管一個 COMMIT 語句通常被快速的執行，因為它通常只需要做很少的事情，而回滾事務通常需要資料庫做大量工作來使以前回到初始狀態，因此需要一些時間執行。因此，如果你開始一個事務，花了兩分鐘時間執行所有的 SQL，然後決定執行一個 ROLLBACK 或者取消所有改變，請別指望回滾能馬上完成。它可能花費隨便就花費超過兩分鐘的時間才能撤銷所有的改變。

多使用者情況下的事務

就像我們在本章前面部分看到的，支援多使用者併發的工作的事務需要將使用者從其他每個人隔離開來（ACID 的 I 部分）。雖然 PostgreSQL 的默認處理隔離的行為就滿足大多數情況，但依舊需要瞭解它的細節的情況。

實現隔離

關係型數據庫中一個最難的方面是隔離不同的使用者對資料庫的更新。當然，如果我們不在乎性能，實現隔離也不難。簡單地允許一個使用者連接到資料庫，並同時只允許一個事務，將確保在不同事務間完全隔離。不幸的是，多使用者的性能將變得極差。事務隔離的難點是實現有效隔離的同時不明顯影響性能或影響多使用者訪問資料庫。

為了減少隔離對性能的影響，ANSI SQL 標準定義了資料庫可以實現的不同的資料庫隔離級別。這允許資料庫管理員在性能和針對資料庫的使用者的隔離程度進行做出平衡。通常，一個關係型數據庫默認將最少實現這些級別中的一個，並允許用戶最少指定另一個隔離級別來使用。

ANSI SQL 標準根據在多使用者資料庫中當事務配合時可能出現的“不良現象（undesirable phenomena）”定義隔離級別。這些現象被稱作髒讀、不可重複讀、幻讀和更新丟失。讓我們看看它們每個的意義以及 ANSI 隔離級別是如何定義的。

髒讀

髒讀發生在一個事務中的 SQL 讀到被其他事務改變的資料，但這個（其它）事務尚未提交其工作塊中改變的資料。

就像我們早先提到的，一個事務是一個必須原子操作的邏輯單元或者工作塊。要麼事務中所有的元素都發生要麼都不發生。直到一個事務被提交前，總有可能事務失敗或者通過 ROLLBACK 命令放棄。因此，沒有資料庫中的其他使用者可以看到 COMMIT 前改變的資料。

表 9-2 闡述不同的事務可以看到允許髒讀和不允許髒讀的情況下的 customer_id 為 15 的客戶的 fname。

表 9-2 髒讀

事務 1	事務 1 看到的資料	其他允許髒讀的事務看到的資料	其他禁止髒讀的事務看到的資料
BEGIN			
	David	David	David
UPDATE customer SET fname='Dave' WHERE customer_id = 15;			
	Dave	Dave	David
COMMIT			
	Dave	Dave	Dave
BEGIN			
	Dave	Dave	Dave
UPDATE customer SET fname = 'David' WHERE customer_id = 15;			
	David	David	Dave
ROLLBACK			
	Dave	Dave	Dave

注意髒讀允許其他事務看到未提交到資料庫的資料。這意味著它們可以看到之後被拋棄的改變，因為可能使用 ROLLBACK 命令。

> 注：PostgreSQL 不允許髒讀。

不可重複讀

不可重複讀非常類似於髒讀，只是更加有限制性。一個不可重複讀發生在黨一個事務讀了一些資料，之後重新讀這些資料且發現它們被改變了。這裡的問題遠沒有髒讀嚴重，但是也不是很完美。一個不可重複讀的闡述展示在表 9-3 中。

表 9-3 不可重複讀

事務 1	事務 1 看到的資料	其他允許不可重複讀的事務看到的資料	其他禁止不可重複讀的事務看到的資料
BEGIN		BEGIN	BEGIN
	David	David	David
UPDATE customer SET fname='Dave' WHERE customer_id = 15;			
	Dave	David	David

COMMIT			
	Dave	Dave	David
		COMMIT	COMMIT
		BEGIN	BEGIN
UPDATE customer SET fname = 'David' WHERE customer_id = 15;			
		Dave	Dave

注意不可重複讀意味著一個事務可以看到其他事務提交的改變，即使讀的事務本身還沒有提交。如果禁止了不可重複讀，其他事務在提交自己的改變前無法看到資料庫的改變。

預設情況下，PostgreSQL 允許不可重複讀，但就像我們在後面將看到的，我們可以改變這個默認行為。

幻讀

幻讀和不可重複讀非常類似，但是發生在一個新行出現在一個表中的時候，另一個事務在更新表，這個新行應該被更新了但實際上沒有。

假設我們有兩個事務在更新 item 表。第一個事務為所有的專案的售價增加一元，而第二個事務則增加一個新專案。這個過程在表 9-4 中描述。

表 9-4 幻讀

事務 1	**事務 2**
BEGIN	BEGIN
UPDATE item SET sell_price = sell_price + 1;	
	INSERT INTO item(….) VALUES(…);
COMMIT	
	COMMIT

由事務 2 添加的專案的售價應該為多少？INSERT 語句在 UPDATE 語句提交前開始；但是，我們可能有理由期望它會比我們插入的價格高 1。如果發生幻讀，因為新行出現在事務 1 中的 UPDATE 探測到哪些行需要更新之後，那麼新項目的價格不會增加 1.

幻讀非常少見，甚至很難做演示，所以通常你不需要關心它們。預設情況下，PostgreSQL 允許幻讀。

更新丟失

更新丟失和以上三個示例有點區別，它通常是一個應用程式級別的問題，而跟關係型數據庫的工作方式無關。更新丟失發生在兩個不同的改變被寫入資料庫時，第二個更新導致第一個丟失。

假設兩個使用者都在使用一個基於螢幕的應用程式，用以更新 item 表。過程在表 9-5 中顯示。

表 9-5 更新丟失

用戶 1	**使用者 1 看到的資料**	**用戶 2**	**使用者 2 看到的資料**
嘗試將售價從 21.95 修改為 22.55		嘗試將成本從 15.23 修改為 16.00	
BEGIN		BEGIN	
SELECT cost_price,	15.23, 21.95	SELECT cost_price,	15.23, 21.95

sell_price FROM item WHERE item_id = 1;		sell_price FROM item WHERE item_id = 1;	
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1;			
	15.23, 22.55		
COMMIT			
			15.23, 22.55
		UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1;	
	15.23, 22.55		16.00, 21.95
		COMMIT	
	16.00, 21.95		16.00, 21.95

被用戶 1 改變的售價丟失了，這並不是因為一個資料庫錯誤，而是因為使用者 2 讀取 sell_price 後，“保存”了一會兒，然後將它寫回資料庫，這導致使用者 1 做的改變被破壞了。資料庫十分正確地隔離了兩個操作，但應用程式還是丟失了資料。

有很多方法解決這個問題；不同的應用程式可能會選擇恰當的方法。首先，應用程式應該儘量保持事務短小，除非絕對需要，否則不要保持事務為運行狀態。第二步應用程式應該只回寫它們改變的資料。這兩步將避免很多更新丟失的發生，包括表 9-5 中的演示中的錯誤。

當然，存在兩個用戶都嘗試修改 sell_price 的情況；在這種情況下，其中一個改變可能會丟失。一個更通用的避免更新丟失的方法是在 UPDATE 語句中編碼你將要改變的值，就像表 9-6 中描述的。

表 9-6 針對更新丟失的一個應用程式解決方法

用戶 1	**使用者 1 看到的資料**	**用戶 2**	**使用者 2 看到的資料**
嘗試將售價從 21.95 修改為 22.55		嘗試將售價從 21.95 修改為 22.99	
BEGIN		BEGIN	
讀 sell_price（WHERE item_id=1）	21.95	讀 sell_price（WHERE item_id=1）	21.95
UPDATE item SET cost_price = 15.23, sell_price = 22.55 WHERE item_id = 1 AND sell_price = 21.95;			
	22.55		21.95
COMMIT			
		UPDATE item SET cost_price = 16.00, sell_price = 21.95 WHERE item_id = 1 AND sell_price = 21.95;	
		由於行未找到而更新失敗，因為 sell_price 被改變了。	

雖然這不是一個完美的解決方法，因為它只在第一個事務在第二個 UPDATE 語句運行前提交才能正確工作，但它可以有效減少更新丟失的風險。

ANSI 隔離級別

ANSI 標準針對這三種不良現象（髒讀、不可重複讀、幻讀）的組合定義了一個資料庫可用的不同隔離級別。這些級別在表 9-7 中列出。

表 9-7 ANSI 隔離級別和不良現象的對照

隔離級別	髒讀	不可重複讀	幻讀
Read Uncommitted（讀未提交）	允許	允許	允許
Read Committed（讀已提交）	不允許	允許	允許
Repeatable Read（重複讀）	不允許	不允許	允許
Serializable（序列化）	不允許	不允許	不允許

就像你看到的，隨著隔離級別從讀未提交到最後的串列級別，發生的不良現象的類型減少了。

改變隔離級別

預設情況下，PostgreSQL 的隔離模式被設置為 READ COMMITTED，對應為表 9-7 中列出的 Read Committed。存在的另一個級別為 SERIALIZABLE，對應 Serializable。

到 8.4 版本為止，PostgreSQL 未實現中間的 Repeatable Read 和初始的 Read Uncommitted。通常，Read Uncommitted 由於其行為很爛，很少有資料庫提供這個選項，而且罕有應用程式大膽（或者傻逼）到選擇使用它。（實際上，PostgreSQL 8.4 的 SERIALIZABLE 級別的實際實現為 Repeatable Read，PostgreSQL 把 Read Uncommitted 級別看做是 Read Committed。）中間級別 Repeatable Read 只針對幻讀提供額外保護，而這種情況也很少見，所以缺乏這種級別的後果也不嚴重。通常資料庫只會提供可選項的子集，而支援 Read Committed 和 Serializable 是一個比較好的折中方案。

你可以通過使用 SET TRANSACTION ISOLATION LEVEL 命令改變隔離級別，以下為語法：

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

除非你有很好的理由改變它，否則我們建議你不要改變你的 PostgreSQL 資料庫中的預設的隔離級別。

使用顯式和隱式的事務

截至本章，我們已經通過明確地使用 BEGIN 和 COMMIT（或 ROLLBACK）來確定我們的事務。在本書較早的部分，我們尚未瞭解事務的時候，我們確能夠開心地不使用 BEGIN 命令而改變我們的資料庫。

預設情況下，PostgreSQL 運行在自動提交模式，有時也被稱為鏈式模式或隱式事務模式，在這種情況下每個能修改資料的 SQL 語句仿佛是在它的許可權範圍內的完整的事務中運行。這對在命令列實驗或者允許新使用者來體驗而不需要太多 SQL 知識的情況非常有用。然而，它對於真正的應用程式卻沒那麼好，因此我們需要顯式地在事務中使用 COMMIT 或 ROLLBACK 語句。

在其他的關係型數據庫管理系統中，實現模式有點不同，通常必須使用顯式的命令來改變模式；例如，Sybase 使用 SET CHAINED，微軟的 SQL Server 使用 SET IMPLICIT_TRANSACTIONS 命令。

在 PostgreSQL 中，你所需要做的就是調用 BEGIN 命令，PostgreSQL 將自動切換到一種接下來的命令在事務中的模式，知道你調用一個 COMMIT 或者 ROLLBACK 語句。

SQL 標準認為 SQL 語句是在事務中執行的，認為事務在第一個 SQL 執行的時候就自動開始了，直到遇到一個 COMMIT 或 ROLLBACK。因此，標準 SQL 沒有定義 BEGIN 命令。然而，PostgreSQL 顯式地使用 BEGIN 使用執行事務的模式也很常用。

鎖

大多數資料庫實現了事務，特別是當隔離不同使用者之間的事務的時候，使用鎖來在使用者間約束對資料的訪問。簡單來講，有兩種鎖：

- 共用鎖，允許其他用戶讀，但不允許修改資料
- 排他鎖，甚至避免其他事務讀數據

例如，伺服器將鎖定被一個事務更新的行直到事務完成，之後這些鎖才被釋放。這些都被自動完成，通常甚至不需要資料庫使用者關注鎖的發生。

實現鎖需要的實際機制和策略極度複雜，有很多不同類型的鎖可以使用，具體的依賴於環境。PostgreSQL 的文檔描述了八種不同類型的鎖。PostgreSQL 還通過一種多版本模式實現了一種不同的事務隔離機制，這減少了鎖的衝突，相對於其他模式明顯提升了它的性能。

幸運的是，資料庫使用者通常只需要在兩種情況注意鎖：避免鎖死（以及從鎖死中恢復）以及通過應用程式顯式加鎖。

避免鎖死

當兩個不同的應用程式同時嘗試修改相同的資料會發生什麼呢？很容得到結果——只需要啟動兩個 psql 會話，並嘗試在它們中改變相同的行。過程在表 9-8 中描述。

表 9-8 鎖死

會話 1	**會話 2**
更新第 14 行	
	更新第 15 行
更新第 15 行	
	更新第 14 行

在這種情況下，兩個會話都被阻塞了，因為每一個都在等待另一個釋放。

這種行為可以作為推斷 PostgreSQL 的預設隔離級別為 Read Committed 模式的線索。在併發性、性能、最大限度地減少一側的鎖的數量以及一致性和其他理想的行為之間有一個權衡。當你提高隔離級別，資料庫中的多使用者的併發性能將降低，就像在圖 9-4 中指出的。

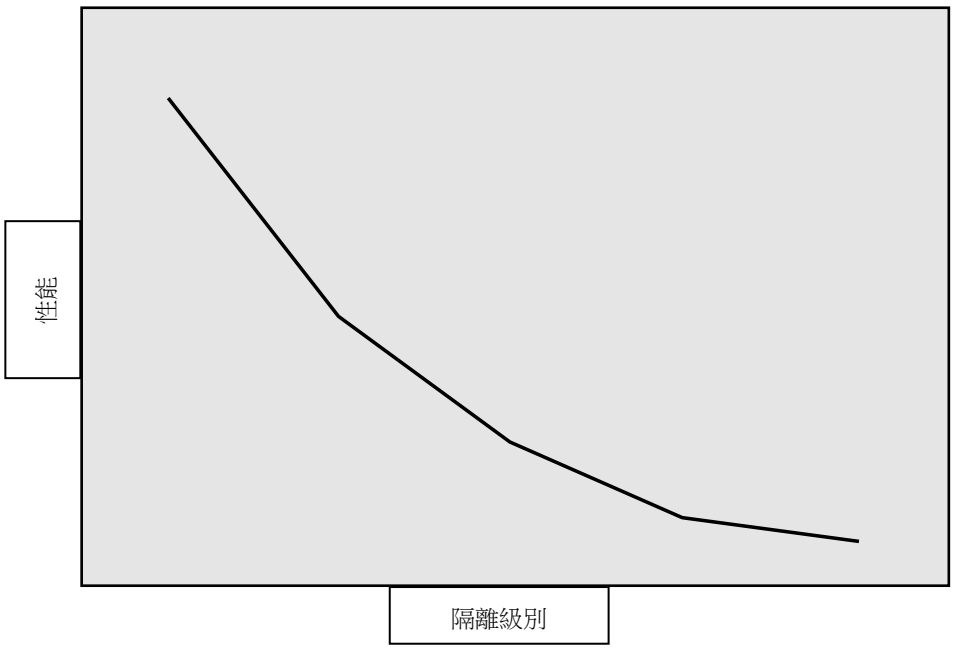


圖 9-4 性能和隔離級別的反比關係

隨著資料庫的行為越來越智慧，需要的鎖的數量也增長了，不同用戶間的併發性也降低，因此整體性能降低了。這是一個必須面對的負面消息。

通常，如果兩個用戶嘗試訪問相同的行，除了第二個用戶必須等待第一個用戶完成訪問外，沒有其他的實際影響。但一種更嚴重的情況是兩個會話會互相阻塞。

嘗試：實驗鎖死

讓我們用我們在第八章設計的 bpfinal 資料庫模式做一個實驗。開始兩個 psql 會話，連接到 bpfinal，並按照以下循序執行命令：

會話 1	**會話 2**
BEGIN	
	BEGIN
UPDATE customer SET fname = 'D' WHERE customer_id = 15;	
	UPDATE customer SET fname = 'B' WHERE customer_id = 14;
UPDATE customer SET fname = 'Bill' WHERE customer_id = 14;	
	UPDATE customer SET fname = 'Dave' WHERE customer_id = 15;

你會發現兩個會話都被阻塞住了，在短暫的暫停後，你將在其中一個會話中看到類似以下的消息：

```
ERROR:  deadlock detected
DETAIL:  Process 2018 waits for ShareLock on transaction 2788; blocked by
```

```
    process 2017.
Process 2017 waits for ShareLock on transaction 2789; blocked by process 2018.
bpfinal=>
```

看到這些錯誤資訊的會話的更新操作被取消；另一個會話將繼續。獲得鎖死消息的會話被回滾了，進行的改動丟失了。另一個會話可以繼續執行一個 COMMIT 語句來使資料庫的變動持久（或者執行一個 ROLLBACK 來放棄改變）。

解析

這裡發生的事情是 PostgreSQL 檢測到了一個鎖死，兩個會話都因為等待對方而阻塞，沒有誰可以繼續。會話 1 鎖住了 15 行，然後會話而發起且鎖住了 14 行。會話 1 然後嘗試鎖住 14 行，但無法繼續，因為這一行被會話 2 鎖住了，然後會話 2 嘗試修改行 15，但無法進行因為它被會話 1 鎖住了。在一小段時間後，PostgreSQL 的鎖死檢測代碼探測到發生了一個鎖死並自動取消了一個事務。

無法進一步確定哪個會話將被 PostgreSQL 選擇來取消。它將挑選一個認為做了最少工作的，但這不是非常科學。應用程式應該能夠也必須小心避免鎖死的發生。最簡單的方法也是我們之前建議的：保持事務儘量短小。事務參與的行和表越少，則鎖被鎖定的時間越短，那麼發生衝突的機會也就越少。

另一個技巧也很簡單：讓應用程式的代碼使用相同的連續處理表和行。在我們的示例中，如果兩個會話都使用了相同的順序來更新行，則不會發生這樣的問題。會話 1 將可以完成兩行的更新，而會話 2 在短暫的停止後，在會話 1 的事務完成後，可以繼續執行。也可以寫代碼在發生鎖死的時候重試，但最好還是設計你的程式來避免問題，而不是在失敗後做重試。

顯式鎖定

偶爾，你可能發現 PostgreSQL 提供的自動鎖定機制不滿足你的需要。在這種情況下，你可能需要顯式鎖定一些行或者整張表。除非必要，否則你應該避免顯式鎖定。SQL 標準甚至沒有定義一種方法來鎖定整張表。這個選項是一個 PostgreSQL 的擴展（你會發現這是在很多資料庫中都有的一個常見擴展）。

有可能在一個事務中鎖定一些行或者表。一旦事務完成，無論是 COMMIT 或者 ROLLBACK，所有在事務中獲得的鎖將被自動釋放。也沒有辦法在事務中顯式釋放鎖，原因也很簡單，在事務中釋放一個被改變了的行上的鎖可能導致其他應用再改變它，這將阻礙 ROLLBACK 恢復初始的改變。

鎖定行

最常見的需求是在對一些行進行改變前鎖定他們。這也是一種避免鎖死的方法。通過提前鎖定你需要改變的行，你可以確保其他應用程式無法建立一個和你的改變相衝突的部分。

只需要在 SELECT 語句後面追加 FOR UPDATE 就可以鎖定一些行，就像這個例子：

```
SELECT 1 FROM item WHERE sell_price > 5.0 FOR UPDATE;
```

在事務中提供這行語句，我們將鎖定所有 sell_price 大於 5 的行。在本例中，我們不需要返回任何行，為了方便地返回最少的資料，我們簡單地選擇了 1。

嘗試：鎖定行

假設我們需要鎖定 customer 表中所有的住在 Nicetown 的記錄，因為我們需要修改電話號碼（也許是因為區號被拆分成幾個了）。我們需要確保我們能訪問所有的行，但需要一些過程代碼來逐行處理，計算新的電話號碼。再次，我們使用第八章建立的 bpfinal 資料庫來做嘗試。

```
bpfinal=> BEGIN
BEGIN
bpfinal => SELECT customer_id FROM customer WHERE town = 'Nicetown' FOR
    UPDATE;
 customer_id
-------------
           3
           6
(2 rows)

bpfinal =>
```

這時候，customer_id 為 3 和 6 的行被鎖定，然後我們可以通過在另一個 psql 會話中嘗試 UPDATE 它們來測試這一點：

```
bpfinal l=> BEGIN;
BEGIN
bpfinal => UPDATE customer SET phone = '023 3376' WHERE customer_id = 2;
UPDATE 1
 bpfinal => UPDATE customer SET phone = '023 3267' WHERE customer_id = 3;
```

這時候，第二個會話被阻塞了，直到我們敲擊 Ctrl+C 來中斷它，或者第一個事務提交或者回滾。

解析

第一個會話使用 SELECT … FOR UPDATE，導致 customer_id 為 3 和 6 的行被鎖定。其他的會話可以更新 customer 表中的其他行，但行 3 和 6 一直被鎖定直鎖定它們的事務結束。

鎖定表

在 PostgreSQL 中，也可以鎖住表，雖然我們強烈建議你避免這樣做，堅持使用 SQL 標準的機制來確保客戶的隔離。

鎖定表的語法如下：

LOCK [TABLE] table-name

LOCK [TABLE] table-name IN [ROW | ACCESS] { SHARE | EXCLUSIVE } MODE

LOCK [TABLE] table-name IN SHARE ROW EXCLUSIVE MODE

通常，程式使用這個簡單的語法鎖住表：

LOCK TABLE table-name

這和以下的效果相同：

LOCK TABLE table-name ACCESS EXCLUSIVE MODE

這避免了任何程式使用任何方法訪問這張表。雖然很苛刻，但這可能是你在實際環境中需要一個表級鎖時的行為。更多的關於不同類型鎖的細節，請參考 PostgreSQL 的文檔。

摘要

在本章，我們瞭解到了事務和鎖。我們看到了即使在單使用者的資料庫中，事務都是非常有用，因為它們運行我們將一組 SQL 命令組成一個原子單元，讓他們要麼都發生要麼都放棄。之後我們瞭解了事務在多使用者環境中如何工作。

然後，我們查看了 ANSI SQL 表中條款中的不良現象，以及針對不同級別的不良行為定義的不同級別的事務一致性。我們還討論了為何消除不良現象會導致性能的下降，因此有必要取得理想的行為和可接受的性能之間的平衡。

最後，我們看了下鎖。我們看到了使用簡單的技巧就可以減少鎖死的發生的可能，也就是避免兩個或者更多應用程式被卡住，等待另一個完成。

我們還討論了顯式鎖，這允許我們在一個事務中鎖定表中指定的行或者甚至是整個表。

雖然事務和鎖不總是非常有意思的話題，但對它們的大體瞭解有助於寫出有良好行為的應用。我們建立的應用不僅需要正確，還要對資料庫的性能的影響最低，發揮 PostgreSQL 的多用戶能力。

下一章，我們將流覽其他的方式來發揮 PostgreSQL 的能力，通過存儲過程和觸發器擴展它的行為。

第十章函數、存儲過程和觸發器

在本章，我們將研究幾種新的方法來擴展和增強到現在為止我們介紹的 PostgreSQL。本章的大部分內容是 PostgreSQL 所特有的，雖然大部分商業的關係型數據庫管理系統，例如 Oracle 也包含類似的功能。

首先，我們將看看 PostgreSQL 對 SELECT 語句的更深的支援，包括高級匹配以及允許我們在 WHERE 從句中構造複雜的測試的算數運運算元。然後我們將查看這些操作如何在 PostgreSQL 中通過函數實現，並介紹一些可以增強我們 SELECT 語句表現力的附加函數。

PostgreSQL 允許開發人員通過使用 C 語言建立新的功能並在資料庫啟動的時候載入它們到伺服器中來擴展資料庫伺服器的功能。這種擴展可以是簡單到一個擴展函數或者複雜到一個有自己規則的完整的程式設計語言。有很多擴展，被稱為*可載入過程語言*，已經包含在標準的 PostgreSQL 的發佈包裡頭。這些語言允許我們建立自己的函數，也就是*存儲過程*，他們用起來比寫 C 程式更快速和容易。在本章，我們將簡要介紹其中一種可載入語言：PL/pgSQL。PL/pgSQL 是 PostgreSQL 特有的，但類似的語言也存在於其他資料庫中。例如，Oracle 有 PL/SQL，Sybase 有 Transact-SQL。

在資料庫中當一些特殊條件發生的時候，存儲過程也可以被 PostgreSQL 伺服器自動執行。例如，當嘗試從一個表中刪除一行的時候，一個存儲過程可以被執行刪除其他表中的相關行來確保參照完整性，或者僅僅是避免刪除發生。這些自主的行為就是*觸發器*，我們也將在這裡看到它們。

在本章，我們將涵蓋以下主題：

- PostgreSQL 的運運算元
- PostgreSQL 的內置函數
- 過程語言，特別是 PL/pgsql
- 存儲過程
- SQL 函數
- 觸發器

運運算元

在第 4 章，我們已經在 SELECT 語句中看到並使用了一些簡單的運運算元。例如，我們可以使用數字比較運運算元來限制查詢的符合條件的行，例如售價大於 4 的專案：

```
bpfinal=# SELECT * FROM item WHERE cost_price > 4;
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubik Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
      11 | Speakers      |      19.73 |      25.32
(6 rows)

bpfinal=#
```

在這裡，運運算元 > 被用在 cost_price 屬性和一個給定的數位之間。

我們可以更進一步，包含其他的屬性和運運算元來建立更複雜的條件：

```
bpfinal=# SELECT * FROM item WHERE (sell_price*100)%100 = 99;
 item_id | description | cost_price | sell_price
---------+-------------+------------+------------
       4 | Tissues     |       2.11 |       3.99
(1 row)

bpfinal=#
```

在這裡，我們同時使用了乘法運運算元和求餘的運運算元來列出哪些以.99 結尾的售價的項目。

我們還可以使用規則運算式進行匹配：

```
bpfinal=# SELECT * FROM item WHERE description ~* '^[PR].*E$';
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       2 | Rubik Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
(2 rows)

bpfinal=#
```

在這裡，運運算元執行了一個不區分大小寫的規則運算式匹配來尋找那些描述資訊由小寫或者大寫的 p 或者 r 開頭並且由 e 結尾的專案。

就像你看到的，PostgreSQL 支持挺多運運算元的。事實上，如果算上相同運運算元的不同變化（也就是，認為比較整數和比較浮點數或比較字串是不同的），有差不多 600 個運運算元。

運運算元的優先順序和結合性

PostgreSQL 的很多運運算元的外觀和行為與你在很多程式設計語言中看到的算數運運算元非常相近。這些運運算元在分析器中有硬編碼的優先順序來確定通過運運算元結合的運算式中它們執行的順序。通常，可以使用括弧來覆蓋優先順序。

PostgreSQL 允許在 SELECT 的 WHERE 從句之外使用運運算元和函數：

```
bpfinal=# SELECT 1+2*3;
 ?column?
----------
        7
(1 row)

bpfinal=# SELECT (1+2)*3 AS answer;
 answer
--------
      9
(1 row)

bpfinal=#
```

這裡，我們可以看到運算式 1+2*3 的結果為 7，預設情況下，無名列名以?column?的形式顯示。在第二個示例中，運運算元的優先順序被覆蓋且結果的被命名為 answer。

雖然大多數運運算元的行為和你期望的完全一樣（如果你有 C 或者其他程式設計語言的開發經驗），但有一些運運算元的優先順序規則可能會有些古怪。在 C 中，布林運運算元的優先順序低於算數運運算元，所以通常需要使用括弧來獲取正確的運運算元執行順序。如果可能出問題，儘量顯示地使用括弧。

PostgreSQL 的運運算元也顯示了其結合性，可以是左結合或者右結合，用來確定相同優先順序的運運算元的取值順序。算數運運算元的例如加減都是做結合的，也就是，1+2-3 取值的順序和(1+2)-3 相同。其他的，例如布林等於運運算元，是右結合的，也就是說 x=y=z 的取值為 x=(y=z)。

表 10-1 列出了大多數常用的 PostgreSQL 的語法優先順序（降冪）。

表 10-1 PostgreSQL 運運算元的優先順序

運運算元	結合性	意思
::		類型轉換（與 CAST 相同）
[]	左	陣列選擇
.	左	物件（模式，表，列）選擇
-	右	一元減號（負數）
^	左	冪
* / %	左	乘法運運算元
+ -	左	加減運運算元
IS		測試（TRUE，FALSE，UNKNOWN 和 NULL）
IS NULL		測試（NULL）
NOT NULL		測試（非 NULL）
OR	左	邏輯或
IN		測試是否為一個集和的成員
BETWEEN		測試是否包含在一個範圍內
LIKE ILIKE SIMILAR		測試字串匹配
<>		測試不等於
=	右	測試等於
NOT	右	邏輯非
AND	左	邏輯與
所有其他的運運算元		不在這裡列出的用戶定義的和內置的運運算元有相同的優先順序

注：舊版本的 PostgreSQL 的運運算元支援計算自然對數和反對數（使用:和;），但這些已經被廢棄並從最近的 PostgreSQL 發行版本中移除，包括 8.0 版本。使用 ln 和 exp 函數代替。

數學運運算元

PostgreSQL 提供大量的數學運運算元。最常用的數學運運算元在表 10-2 中列出。它們都是左結合的。

表 10-2 常用的 PostgreSQL 數學運運算元

運運算元	示例	意思
+	2 + 3 為 5	加法
-	3 - 2 為 1	減法

*	2 * 3 為 6	乘法
/	3 / 2 為 1 3 / 2.0 為 1.5 3 / 2::float8 為 1.5	除法
%	22 % 7 為 1	求餘
^	4 ^ 3 為 64	求冪
&	14 & 23 為 6	二進位 AND
|	14 | 23 為 31	二進位 OR
#	14 # 23 為 25	二進位 XOR
>>	128 >> 4 為 8	右移位
<<	1 << 4 為 16	左移位

還有一些一元數學運運算元，在表 10-3 中列出。

表 10-3 常用的 PostgreSQL 數學運運算元

運運算元	示例	意思
!	4!為 24	階乘
!!	!!4 為 24	階乘（左運運算元）
@	@(-2)為 2	絕對值
|/	|/64 為 8	平方根
||/	||/64 為 4	立方根
~	~15 為 16	二進位 NOT

通常，數學運運算元的運行效果和預期的相同。PostgreSQL 將使用和參數匹配的版本相同的運運算元。所以當你使用一個整數進行除法運算，你將得到一個整數。當你使用浮點數進行除法運算，你將得到一個浮點數結果。就像表 10-2 顯示的，要強制獲得浮點數結果，其中一個參數應該被轉換為一個浮點數。

比較和字串運運算元

PostgreSQL 提供了一組常用的比較運運算元，例如小於和大於。這些運運算元可以用在 PostgreSQL 支援的很多類型上，所以例如，你可以使用大於運運算元來通過字母順序作為相對大小來比較字串的大小。

比較運運算元的結果為 true 或者 false，psql 用 t 或 f 顯示。可用的比較運運算元在表 10-4 中給出。

表 10-4 PostgreSQL 的比較運運算元

運運算元	示例	意思
<	2 < 3 'axy' < 'azz'	小於
<=	2 <= 3	小於或等於
<> !=	2 <> 3 2 != 3	不等於
=	3 = 1 + 2	等於
>	3 > 2	大於
>=	3 >= 2	大於或等於

在 PostgreSQL 中，字串有它自有的一套運運算元。有一些用於連接字串和通過一些規則匹配字串的運運算元，在表 10-5 中總結。

表 10-5 PostgreSQL 的字串運運算元

運運算元	示例	意思
||	'abc' || 'def' 為 'abcdef'	連接字串
~~	'xyzzy' ~~ '%zz%'	LIKE 的同義詞
!~~	'xyzzy' !~~ '%aa%'	NOT LIKE 的同義詞
~	'xyzzy' ~ 'y.*y'	規則運算式子串匹配;使用^開始或者$結尾來表示開始配或者結束匹配。
~*	'xyzzy' ~* '^X.*Y$'	規則運算式匹配，不區分大小寫
!~	'xyzzy' !~ 'aa'	不匹配（~的相反值）
!~*	'xyzzy' !~ 'AA'	不匹配，不區分大小寫（~*的相反值）

在規則運算式匹配中，字串和規則運算式的匹配類似於使用 UNIX 的 grep 工具或者 Perl 的匹配運運算元。

其他運運算元

PostgreSQL 提供大量的附加運運算元來比較和操作 PostgreSQL 特有的資料類型例如 points，circles，time intervals 和 IP 地址。請參考 PostgreSQL 文檔的第二章獲得更多資訊。

提示：PostgreSQL 文檔可以以 HTML 頁面的方式安裝，然後在本地通過流覽器打開 file://usr/local/pgsql/doc/html/index.html 來閱讀，或者在 http://www.postgresql.org 線上閱讀。

所有的運運算元都存放在資料庫的 pg_operator 表中，psql 可以通過內部命令\do 和\df 來列出所有的運運算元和函數。

內置函數

PostgreSQL 提供了大量的內置函數，我們可以在 SELECT 運算式中使用它們。

提供的函數大體分為以下幾類：

- 和上一節提供的運運算元相同功能的函數
- 其他的數學函數
- 其他用於處理字串中的字元的函數
- 處理日期和時間的函數
- 文本格式化的函數
- 處理 PostgreSQL 地理類型例如 point 和 circle 的函數
- IP 位址的函數

內置函數（和使用者自訂函數）被記錄在 PostgreSQL 資料庫的系統表 pg_proc 中。在 8.0 和以後的版本中，這個表中有超過 1700 項。在 psql 中，你可以使用\df 命令來列出所有函數和它的參數。你也可以使用\dd 命令來顯示指定的函數或者函數分組的注釋。

很多內置的函數提供一個和每個數學和邏輯運運算元相對應的函數。例如包含 int4shl 和 float8mul 對應了給 integer 的左移位運運算元（<<）和浮點值的乘法運運算元（*）。表 10-6 列出了用於浮點數且返回一個浮點數的數學函數，除非另有說明。

表 10-6 常用的 PostgreSQL 數學函數

函數	意思

abs(x)	絕對值
degrees(r)	弧度轉化為角度
radians(d)	角度轉換為弧度
exp(x)	自然反對數，以自然對數 e 為底的指數
ln(x)	自然對數
log(x)	以 10 為底的對數
log(b,x)	以 b 為底的對數
mod(x,y)	x 除 y 後的餘數（還有個整數版本）
pi()	返回 π
pow(x,y)	求 x 的 y 次方
random()	返回一個 0 和 1.0 之間的亂數
round(x)	四捨五入到最近的整數
round(x,d)	四捨五入到指定的小數位 d（可以為負數哦）
trunc(x)	取整到整數（相對於 0）
trunc(x,d)	取整到指定的小數位 d
ceil(x)	返回不小於給定數的最小整數
floor(x)	返回不大於給定數的最大整數
sqrt(x)	平方根
cbrt(x)	立方根
float8(i)	將一個整數轉換為一個 float8
float4(i)	將一個整數轉換為一個 float4
int4(x)	獲取整數，四捨五入

也支持三角函數，列在表 10-7 中。所有的角參數和結果都是弧度。

表 10-7 PostgreSQL 的三角函數

函數	意思
sin	正弦
cos	余弦
tan	正切
cot	餘切
asin	反正弦
acos	反余弦
atan	飯正切
atan2	兩個參數的飯正切，計算 atan(a/b)

PostgreSQL 包含了標準 SQL 字串函數，但有它自己的語法。這些函數列在表 10-8 中。對於這些函數，字串可以是 char、varchar 或 text。PostgreSQL 使用自有的附加函數擴展了字串管理功能。請參考 PostgreSQL 的文檔獲得更多資訊。

表 10-8 常用的 PostgreSQL 字串函數

函數	意思
char_length(s) character_length(s)	字串長度
octet_length(s)	存儲字串需要的長度
lower(s)	轉換為小寫
upper(s)	轉換為大寫

position(s1 in s2)	s1 出現在 s2 中的位置
substring(s from n fro m)	從 s 的位置 n 開始獲取長度為 m 的子串
trim([leading | trailing | both] [s1] from s2)	從 s2 中去除子串 s1，可以是從開始、結尾或者同時，如果沒給出 s1 則默認移除空白

一個在這裡需要提及的重要的格式化函數是 to_char 函數。它在 PostgreSQL 中扮演著 C 中的 printf 的角色，處理所有用於列印或者顯示的格式化的值。它將通過一個日期範本格式化一個日期和時間值，而且可以使用很多不同的方法格式化數位值（包括羅馬數字）。

注：想瞭解更多的關於 PostgreSQL 所有函數的資訊，參考 PostgreSQL 的文檔或者流覽 PostgreSQL 發佈包源碼中的回歸測試代碼。

過程語言

就像在本章介紹部分提及的，我們可以在 PostgreSQL 資料庫中使用定義自己的函數。這對於我們想進行特殊的計算或查詢並在很多地方重用非常有用。

新建函數的 SQL 是 CREATE FUNCTION，使用以下的基本語法：

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
```

函式定義部分可以不按照這個順序，一個流行的做法是在定義部分之前給出語言類型，就像這樣：

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    LANGUAGE 'langname'
    AS definition
```

注：有另一種 CREATE FUNCTION 允許編譯的物件代碼併入 PostgreSQL 伺服器中，通常是用 C 語言編寫的。

一個非常簡單的用於將一個參數增加一的函數可以這樣寫：

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

函數的定義是一個可以被拆分成多行且可以用任何 PostgreSQL 支援的可載入過程語言編寫的字串。在這裡，使用的是 PL/pgSQL，在 LANGUAGE 從句中用 plpgsql 表示。PL/pgSQL 是一種為在 PostgreSQL 中編寫存儲過程而特別開發的程式設計語言。

注：可載入過程語言 PL/Tcl，PL/Perl 和 PL/Python 允許你分別使用 Tcl，Perl 和 Python 程式設計語言建立 PostgreSQL 的擴展。

要處理一個過程語言，PostgreSQL 必須首先通過包含一個處理函數（通常是用 C 編寫的）來進行擴展。對於 PL/pgSQL，處理常式包含在在作為一個共用庫的發佈包中。

當一個函數被建立，它的定義將被保存在資料庫中。當函數第一次被調用的時候，定義內容被處理函數編譯為可執行格式並執行。這意味著我們無法獲得我們函數中的錯誤的提示，直到我們嘗試使用它。

在我們選擇可載入的語言來寫自己的函數前，我們必須讓 PostgreSQL 支援這種語言。這將是我們下一步需要為 PL/pgSQL 做的。

PL/pgSQL 入門

在本章，我們將使用 PL/pgSQL 作為可載入過程語言來編寫示例存儲過程。在標準的 PostgreSQL 安裝中，PL/pgSQL 的處理函數被包含在 Unix 和 Linux 的/usr/local/pgsql/lib 目錄中的 plpgsql.so 中，或者 Microsoft Windows 下的安裝目錄下的\lib 下的 plpgsql.dll 中。

伺服器上的每個 PostgreSQL 資料庫都有自己的一個過程語言清單。當我們安裝一個過程語言時，我們可以選擇哪個資料庫需要用這種語言運行存儲過程。某種程度上說，這是一種安全性原則，因為有可能創建的函數無意或惡意的佔用 CPU 資源，例如無限迴圈。這可能形成拒絕服務（DoS）攻擊，使伺服器太忙不能回應新的請求。因此，預設情況下，PostgreSQL 資料庫沒有安裝過程語言。要使用 PL/pgSQL，我們需要自己安裝處理常式。

注：資料庫管理員可以添加語言到 template1 資料庫中，這樣，所有新建的資料庫都將預設擁有這種語言。

要在我們的 bpfinal 資料庫中安裝 PL/pgSQL，我們可以通過 psql 使用 CREATE LANG 命令並載入共用庫，顯式地建立處理函數。這複雜到需要提供一個說明腳本來完成，所以 PostgreSQL 提供了一個工具。需要用的命令是 createlang：

```
createlang [options] langname dbname
```

相關的選項在表 10-9 中列出。

表 10-9 createlang 命令的選項

選項	意思
-h, --host=hostname	指定資料庫伺服器主機
-p, --port=port	指定資料庫伺服器埠
-U, --username=username	指定連接的用戶名
-W, --password	提示密碼
-d, --dbname=dbname	設置需要安裝語言的資料庫
-L, --pglib=directory	在 directory 中搜索語言解釋檔
-e, --echo	顯示發送到伺服器的命令
-l, --list	列出所有當前安裝的語言

普通使用者無法往資料庫添加語言，所以我們通常使用 postgres 超級用戶連接：

```
$ createlang -U postgres plpgsql bpfinal
```

我們可以使用 createlang 命令檢查語言是否已存在，或者使用 psql 或者 pgAdmin 查詢系統表 pg_language：

```
$ createlang -l bpfinal
Procedural Languages
  Name   | Trusted?
---------+----------
 plpgsql | yes
$ psql -d bpfinal
bpfinal=# SELECT * FROM pg_language;
 lanname  | lanispl | lanpltrusted | lanplcallfoid | lanvalidator |    lanacl
----------+---------+--------------+---------------+--------------+---------------
 internal | f       | f            |             0 |         2246 |
```

```
 c          | f       | f            |             0 |         2247 |
 plpgsql    | t       | t            |         17601 |        17602 |
 sql        | f       | t            |             0 |         2248 | {=U/postgres}
(4 rows)

bpfinal=#
```

注：createlang 可以用於安裝其他過程語言。例如，用於安裝 PL/Tcl 的命令是 createlang pltcl。

超級用戶也可以通過 psql 執行 DROP LANGUAGE 命令來從資料庫中移除語言支援：

```
bpfinal=# DROP LANGUAGE 'plpgsql';
DROP LANGUAGE

bpfinal=#
```

嘗試：建立第一個存儲過程

我們現在已經準備好通過編寫我們自己的函數來開始學習使用 PL/pgSQL 編寫存儲過程了。讓我們從實現之前提到的 add_one 函數開始檢查是否所有東西都正常工作開始：

```
bpfinal=# CREATE FUNCTION add_one (int4) RETURNS int4 as '
bpfinal'# BEGIN return $1 + 1; end;' language 'plpgsql';
CREATE FUNCTION
bpfinal=# SELECT add_one(2) AS answer;
 answer
--------
      3
(1 row)

bpfinal=#
```

解析

CREATE FUNCTION 命令存儲通過 PL/pgSQL 語言定義的 add_one 函數到資料庫。之後，用 SELECT 運算式執行它。注意 PL/pgSQL 的關鍵字例如 BEGIN 和 LANGUAGE 都是不區分大小寫的，而且命令的佈局也不影響。我們也可以這樣定義這個函數：

```
bpfinal=# CREATE function
bpfinal-# add_one(int4) RETURNS int4
bpfinal-# AS '
bpfinal'# BEGIN
bpfinal'#   RETURN $1 + 1;
bpfinal'# END;
bpfinal'# '
bpfinal-# LANGUAGE 'plpgsql';
CREATE FUNCTION
```

```
bpfinal=#
```

函數重載

PostgreSQL 把名稱不同、參數個數不同或者參數類型不同的函數都視為不同的函數。如果有必要，我們可以建立另一個 add_one 函數。來看看使用一個浮點數調用我們的 add_one 函數會發生什麼：

```
bpfinal=# SELECT add_one(3.1);
ERROR:  function add_one(numeric) does not exist
HINT:  No function matches the given name and argument types. You may need to add
explicit type casts.

bpfinal=#
```

PostgreSQL 返回一個錯誤，因為無法找到使用浮點數作為參數的 add_one 函數。

注：早期版本的 PostgreSQL 會通過自動類型轉換將浮點數轉換成整數來執行這個 add_one 函數。3.1 會被四捨五入到 3，函數會返回 4。

如果我們需要一個為浮點數增加值的函數，我們只需要定義另一個叫 add_one 的函數：

```
bpfinal=# CREATE FUNCTION
bpfinal-# add_one(float8
) RETURNS float8
bpfinal-# AS '
bpfinal'# BEGIN
bpfinal'#   RETURN $1 + 1;
bpfinal'# END;
bpfinal'# '
bpfinal-# LANGUAGE 'plpgsql';
CREATE FUNCTION

bpfinal=# SELECT add_one(3.1);
 add_one
---------
     4.1
(1 row)

bpfinal=#
```

這次，我們得到了想要的結果，因為 PostgreSQL 可以找到並執行相應版本的 add_one 函數。這種行為，被稱作函數重載，會非常有用，但是也會帶來麻煩。為了保持函數唯一，我們必須通過指定他們的參數來參考它們。在本例中，有 add_one(int4)和 add_one(float8)兩個函數可以參考。

提示：一個更方便的用來創建函數方式是編輯包含函式定義的指令檔並使用 psql 的\i 命令來讀取他們。

列出函數

如果我們需要看載入到資料庫中的函數的源碼，我們可以使用 psql 的\df+內部命令或者查詢用於保存存儲過程的表 pg_proc：

```
bpfinal=# SELECT prosrc FROM pg_proc WHERE proname = 'add_one';
 prosrc
--------

begin
        return $1 + 1;
end;



begin
        return $1 + 1;
end;

(2 rows)

bpfinal=#
```

刪除函數

可以通過 DROP FUNCTION 函數來刪除資料庫中的函數。如果我們需要完全刪除一個函數，必須確保從重載的函數中指定出正確的函數並逐個刪除：

```
bpfinal=# DROP FUNCTION add_one(int4);
DROP FUNCTION

bpfinal=# DROP FUNCTION add_one(float8);
DROP FUNCTION

bpfinal=#
```

引號

使用 PL/pgSQL 編寫存儲過程的一個稍微有點複雜的問題是引號。由 CREATE FUNCTION 命令提供的整個函數的定義是一個單引號括起來的字串。這意味著如果我們如果需要在函式定義中使用單引號，必須使用轉義。我們通過使用兩個連續的單引號來表示字串中的一個單引號。

如果我們的過程使用一個用嵌入了轉義了的單引號的用單引號括起來的字串，我們也必須將他們轉義。這可能最終導致使用連續的四個單引號（或更多）。這裡有一個示例：

```
create function ... as ' ... return ''string with a single '''' in it ''; ...' ...
```

幸運的是，在 PostgreSQL 8.0 以及以後的版本中，提供了新功能*美元符號引號*。類似於 Perl 和 UNIX/Linux 的 shell 的工作方式，美元符號引號允許我們在開始和結束的引號之間選擇使用一個字串。通過選擇一個合適的不會出現在我們存儲過程中的字串，我們不需要使用轉義。美元符號引用指一個在$字元之間的長度為 0 到任意長度的字串。所以，之前的示例看上去可以是這樣：

```
create function ... as $$ ... return 'string with a single '' in it '; ... $$ ...
```

如果在我們的存儲過程中出現了$$字串，我們可以選擇另一個美元符號引號：

```
create function ... as $WHAT$ ... return 'string with a $$ in it '; ... $WHAT$ ...
```

本章從這裡開始的示例將使用美元符號引號。如果你還在使用 8.0 版本之前的 PostgreSQL，我們建議你升級到最新版本。否則，你需要使用傳統的引號。

解析一個存儲過程

現在我們已經知道如何建立、執行和刪除一個示例存儲過程了，是時候進一步看看 PL/pgSQL 存儲過程的結構的細節了。PL/pgSQL 是一種塊結構的語言，類似於 Pascal 或者 C，包含變數定義和代碼塊。每一塊有一個可選的標題；可以有一些變數聲明，以及包含在 BEGIN 和 END 關鍵字之間的代碼塊。一個塊的語法如下：

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

> 注：PL/pgSQL 不區分大小寫。所有關鍵字和變數名都可以以任意大小寫書寫。

一個 PL/pgSQL 函數的定義是通過一個包含由引號（可以是單引號或者美元符號引號）括起來的代碼塊的 CREATE FUNCTION 語句定義的。

```
-- For all PostgreSQL versions
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS 'block definition'
    LANGUAGE 'plpgsql';
-- For PostgreSQL 8.0 and later
CREATE FUNCTION name ( [ [arg] ftype [, ...] ] )
    RETURNS rtype
    AS $$
       block definition
    $$ LANGUAGE plpgsql;
```

在我們的示例中的定義部分，雙破折號（--）引入了延伸到行尾的注釋，它將在執行時被忽略。我們會很快學習使用注釋。

函數參數

PL/pgSQL 函數可以有零到多個參數，參數類型在函數名後的括弧內給出。類型是 PostgreSQL 內建的類型，例如 int4

或者 float8。所有的存儲過程必須返回一個值，返回的類型由函式定義部分的 RETURNS 從句指定。

在函數體內，函數的參數依照被定義的順序被稱為$1，$2，以此類推。我們將會在稍後發現可以使用 ALIAS 定義來給參數命名。對於 PostgreSQL 8.0，也可以在聲明變數的時候給他們命名。

讓我們研究下以下這個計算兩個整數的浮點幾何平均值的存儲過程：

```
-- geom_avg
-- get a geometric average of two integers
create function geom_avg(int4, int4) returns float8 as $$
begin
        return sqrt($1 * $2::float8);
end;
$$ language plpgsql;
```

使用 PostgreSQL 8.0 的具名引數功能，我們可以這樣定義這個函數：

```
-- geom_avg
-- get a geometric average of two integers
create function geom_avg(a int4, b int4) returns float8 as $$
begin
        return sqrt(a * b::float8);
end;
$$ language plpgsql;
```

注意我們轉換了其中一個整數值到浮點值，因而我們確保傳入了一個浮點的乘法結果值給 sqrt 函數。如果我們不這麼做，我們將可能獲得一個錯誤提示，告知 sqrt(int4)不存在，當然在 PostgreSQL 8.0 版本之前可能會正常。在存在函數重載的情況下，使用類型轉換將確保我們調用了正確版本的函數。

注釋

就像你之前看到的，PL/pgSQL 允許在函式定義部分添加注釋。事實上有兩種類型的注釋：單行注釋和塊注釋。標準的 SQL 單行注釋由雙破折號（--）表示。從雙破折號開始直到行尾的內容都會被忽略：

```
-- This is a single line comment
create   -- comments can
function -- come anywhere, and extend to the end of the line
```

塊注釋用於注釋掉一大塊介紹內容或者臨時一段不需要的代碼。語法和 C 和 C++類似，被注釋的內容被/*和*/包括：

```
/*
        This is a block comment used to describe
        the use and behavior of the following function
*/
create function blah() returns integer as $$
begin
        /* comment out call to func
        func();
        */
        return 1;
end;
```

```
$$ language plpgsql;
```

塊注釋不能被嵌套，但你可以使用單行注釋來避免塊注釋結束符被解析為塊注釋。

聲明

PL/pgSQL 函數可以聲明用於函數的區域變數。每個變數可以是一個 PostgreSQL 內置類型、用戶自訂類型或者對應到一個表的行的類型。

函數的變數聲明可以放在函式定義部分或者函數函數塊的 DECLARE 段。和塊結構語言例如 C 和 C++一樣，聲明的變數僅能用於對應的代碼塊以及包含的子塊。內部塊中聲明名變數將隱藏對外部塊中同名變數的訪問：

```
DECLARE
        n1 integer;
        n2 integer;
BEGIN
        -- can use n1 and n2 in here
        n2 := 1;
        DECLARE
                n2 integer; -- hides the earlier n2
                n3 integer;
        BEGIN
                -- can use n1, n2 and n3 in here
                n2 := 2;
        END;
        -- n3 no longer available here
        -- n2 still has value 1 here
END;
```

函數中所有的變數在被使用前必須被聲明，除了我們之後會碰到的迴圈控制變數。變數名不能是 PL/pgSQL 的保留關鍵字。表 10-10 理出了 PostgreSQL 的 PL/pgSQL 的關鍵字：

表 10-10 PostgreSQL 的 PL/pgSQL 的關鍵字

alias	assign	begin	close
constant	cursor	debug	declare
default	diagnostics	dotdot	else
elsif	end	exception	excute
exit	fetch	for	from
get	if	in	info
into	is	log	loop
next	not	notice	null
open	or	perform	raise
rename	result_oid	return	return_next
reserve	row_count	select	then
to	type	when	warnning
while			

有很多方法來聲明一個變數，這依賴于它的用法。這些聲明將在下文討論。

別名（ALIAS）

最簡單的聲明訪問是別名(ALIAS)，用於給某個函數參數一個名字。這有助於讓我們的代碼更有意義，而且代碼在針對在修改參數數量或者順序的時候顯得更健壯。ALIAS 聲明的語法為：

```
name ALIAS FOR $n;
```

於是有了一個叫 name 的變數，它可以作為指定位置的參數的另一個名字使用。例如，我們可以改寫 geom_avg 函數為這樣：

```
create function geom_avg(integer, integer) returns float8 as '
declare
    first alias for $1;
    second alias for $2;
begin
    return sqrt(first * second::float8);
end;
' language 'plpgsql';
```

就像之前提及的，從 PostgreSQL 8.0 開始，我們可以在函式宣告的時候為給定位置的參數建立別名。以下的定義和上面這個相同：

```
create function geom_avg(first integer, second integer) returns float8 as $$
begin
        return sqrt(first * second::float8);
end;
$$ language plpgsql;
```

重命名（RENAME）

也可以通過 RENAME 來重命名變數。這對於之後將提用到的觸發器函數中非常有用，但通常情況下不建議使用，因為它會使代碼很難懂。RENAME 聲明的語法如下：

```
RENAME original TO new;
```

簡單的變數聲明

簡單的變數聲明可以通過給定名字，類型以及可選的初始值來實現。以下為語法：

```
name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } value];
```

CONSTANT 修飾符聲明變數不能被改變。因此它必須在聲明時給出初值。

NOT NULL 從句會讓變數在被賦一個 NULL 值的時候讓 PostgreSQL 拋出一個執行階段錯誤。

初始值可以不是常量，他可以在每次函數被調用或進入者變數定義所在的塊的時候被計算。例如，給一個 timestamp 類型的變數的初始值為 now，結果是變數獲得的時間是函數執行的時間，而不是被編譯的時間。

類型可以是一個PostgreSQL內置的類型;也就是,你可以聲明和另一個資料庫專案相同的資料類型或者結構的變數。這樣間接指定變數類型的好處是即使資料庫發生了變動，存儲過程還可以保持正確。語法如下：

```
builtintype
variable%TYPE
```

```
table.column%TYPE
```

以下有一些變數聲明的示例：

```
n integer := 1;
mypi constant float8 := pi();
pizza_pi mypi%TYPE;
mydesc item.description%type := 'extra large size pizza';
```

在示例中，mydesc 的聲明的結果是選擇合適的類型用於處理當前資料庫中的 item 表的 description 列的類型。如果這個列在聲明變數時的定義為 char(64)，而之後改為了 char(80)，使用 mydesc 的代碼仍然能夠工作，PostgreSQL 將建立正確的變數類型。

注：我們在這個示例中用的是雙美元引號。如果使用單引號，我們需要使用兩個單引號括起來的字串來初始化 mydesc，因為這個聲明出現在由單引號括起來的函式定義字串中。

複合變數的聲明

複合變數值對應到一個特定表的完整行的變數。它有對應到表的每個列的欄位。我們可以在我們的存儲過程中聲明和使用複合變數，可以是 rowtype 或者 record。

要聲明一個符合變數，我們使用 rowtype 聲明，語法如下：

```
name table%rowtype;
```

這個聲明將得到一個有欄位的變數，包含給定的表的每一個列。例如：

```
contact customer%rowtype;
```

將建立一個叫 contact 的變數，它擁有關聯到 customer 表的所有列的欄位。要使用欄位，我們可以使用 variable.field 這樣的語法。以下為一個示例程式碼片段：

```
DECLARE
        contact customer%rowtype;
        address text;
BEGIN
        contact.zipcode := 'XY1 6ZZ';
        contact.fname := NULL;
        address := contact.addressline || contact.town;
...
END;
```

第二種複合變數聲明的方法是 record 類型。這是一種和 rowtype 非常類似的類型，但它在定義的時候不是基於某個的表的。這種 record 類型將擁有運行時賦給它的任何結構。Record 在被不同表調用的觸發器中非常有用，例如在一個通用的用於記錄行被刪除的存儲過程中非常有用。它通常也可以用於存儲 SELECT 語句的結果。Record 聲明非常簡單：

```
name record;
```

我們將在討論通過選擇賦值和觸發器部分瞭解更多關於 record 的內容。

賦值

PL/pgSQL 的變數通過設定陳述式獲得新值。設定陳述式的語法如下：

```
reference := expression;
```

reference 是一個變數名或者一個複合類型的一個欄位，例如 rowtype 或者 record。運算式可以是靜態的、另一個變數或者一個由運運算元、類型轉換和函式呼叫組成的複雜運算式的欄位。以下是一些賦值的示例：

```
n1: = 23;
long_variable_names_are_OK := (n1 + 45)/2;
f2 := add_one(n1)::float8 * sqrt(2.0);
/* Composite types may be assigned one field at a time
   by referring to individual fields: */
contact.zipcode := 'AB12 3CD';
```

SELECT INTO 語句

另一種賦值機制是對 SELECT 的擴展。我們可以使用 SELECT INTO 語句賦予一個變數，一個變數的清單，一整行變數或者一個記錄。語法是對普通 SELECT 的擴展：

```
SELECT expressions INTO target [FROM ...];
```

以下是一些用 SELECT 來代替設定運運算元的簡單示例：

```
SELECT sqrt(2.0) INTO sqrt2;
SELECT add_one(n1) INTO n1;
SELECT 1,2,3,4 INTO n1, n2, n3, n4;
SELECT 'Mole', 'Adrian' INTO contact.lname, contact.fname;
```

如果我們為每個裡的值指定了正確的順序，我們可以一次性賦值給整個 rowtype：

```
DECLARE
        product item%rowtype;
BEGIN
        select NULL, 'Widget', '1.45', '1.99' into product;
END;
```

當變數和賦的值的類型不同的時候，PostgreSQL 將執行可以進行的合適的類型轉換。在上面的示例中，成本和售價的類型都是 numeric(7,2)，但是它們都能成功地被賦予字串類型的值。

賦值都會被 PostgreSQL 伺服器作為 SELECT 語句執行，即使是使用了:=運運算元格式。我們可以通過使用帶有 FROM 從句和可選的 WHERE 條件的強大的 SELECT 語句來基於資料庫的內容進行賦值：

```
SELECT * INTO product FROM item WHERE item_id = 9;
```

我們必須小心 SELECT 語句只會返回一行，因為其他更多的行都會被悄悄地忽略掉；只有第一行被返回並賦給列出的變數。我們將在不久後看到我們可以安排代碼來處理 SELECT 返回的多行資料。

也有可能 SELECT 不返回資料；這種情況下，賦值將不會被執行。要檢測到這種情況，我們可以使用 PostgreSQL 特有的布林變數，叫做 FOUND，它在使用 SELECT INTO 時被賦值：

```
SELECT * INTO product FROM item WHERE description ~~ '%Cube%';
IF NOT FOUND THEN
-- take some recovery action
END IF;
```

PERFORM 語句

某些情況下，我們不需要獲取 SELECT 語句的結果集，比如有可能是用於調用一個函數。這種情況下，我們可以計算運算式或查詢，並放棄與執行的結果：

```
PERFORM  query;
```

PERFORM 語句實際上執行了 SELECT 查詢並忽略了結果集。

執行控制結構

PL/pgSQL 提供用於控制函數中執行流程的結構。他們包括返回，條件分支和迴圈。

從函數返回

從函數返回值可以通過 RETURN 語句實現：

```
RETURN expression;
```

函數的處理流程在執行到這個運算式的時候停止。運算式的值將作為函數的結果傳給調用者。值必須和函式宣告的返回類型相容，且它在有必要的情況下進行類型轉換。

在 PL/pgSQL 中，函數必須返回值，如果在函數最外層代碼塊結束的時候沒有執行 RETURN，則會產生一個執行階段錯誤。

異常和消息

如果一些不可能繼續的條件出現，函數的執行也可以被停止。函數可以拋出一個異常，而不是返回一個值。異常會導致 PostgreSQL 在日誌中寫入一個專案，並讓存儲過程立即終止。RAISE 語句可以根據幾個嚴重級別記錄異常。

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

在表 10-11 中，列出了 PostgreSQL 定義的幾個級別的異常。

表 10-11 PostgreSQL 的異常級別

級別	行為
DEBUG，LOG，INFO	在日誌中寫一條消息（僅此而已，通常被忽略掉）
NOTICE，WARNNING	在日誌中寫一條消息，並發送日誌到應用程式
EXCEPTION	在日誌中寫一條消息，並終止存儲過程

DEBUG 級別對於在卡法中獲取附加資訊非常有用。NOTICE 級別提供非致命錯誤的警告。警告會通知到用戶端應用程式，如果有必要，可以通過 NOTIFY 機制獲取消息。EXCEPTION 級別對於致命錯誤非常有用，這種情況下，存儲過程無法繼續。要獲取更多的配置異常的行為的資訊，請參考 PostgreSQL 的文檔。

字串 format 是用來格式化錯誤消息的。使用這個字串的時候，每個%字元被按順序由之後的變數替換。與 C 裡頭的 printf 不同，在 RAISE 語句中，你不允許使用運算式，只能是識別字。

看看以下這個例子：

```
RAISE DEBUG 'The value of n is %', n;
```

這會導致在 PostgreSQL 的日誌檔（在 linux 中通常是 /usr/local/pgsql/data/postmaster.log）增加一條日誌記錄，內

容看起來就像是這樣：

```
DEBUG: The value of n is 4
```

這裡有一個示例存儲過程：

```
create function scope() returns integer AS $$
DECLARE
    n integer := 4;
BEGIN
    RAISE DEBUG 'n is %', n;
    return n;
END;
$$ language plpgsql;
```

如果我們在 psql 中執行這個存儲過程，將沒有消息顯示：

```
bpfinal=# SELECT scope();
 scope
-------
     4
(1 row)

bpfinal=#
```

如果我們將 RAISE 語句中的嚴重級別提升到 NOTICE，我們將在 psql 中看到一個消息，這和日誌檔中的一樣：

```
bpfinal=# SELECT scope();
NOTICE:  n is 4
 scope
-------
     4
(1 row)

bpfinal=#
```

最後，在 EXCEPTION 級別，存儲過程一早就終止了，並返回一個錯誤：

```
bpfinal=# SELECT scope();
ERROR:  n is 4

bpfinal=#
```

條件

PL/pgSQL 支持很多類型的條件，用於針對一個測試執行兩個或者多個語句，或者返回兩個或多個結果。也許這是 PL/pgSQL 最有用的部分。

IF-THEN-ELSE

最常見的條件陳述式是 IF 語句，和其他程式設計語言一樣。語法如下：

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

如果 expression 的結果為 true，則 IF 語句的 THEN 部分被執行。否則，如果存在（可選的）ELSE 部分，這部分代碼將被執行。

和其他程式設計語言一樣，IF 語句可以在 THEN 或者 ELSE 部分中嵌入其他的 IF 語句。

NULLIF 和 CASE

也有兩個 SQL 條件函數根據一個測試來返回值。它們是 NULLIF 和 CASE，它們可以用於存儲過程和 PostgreSQL 的常規 SQL 中。NULLIF 函數在輸入符合指定值的時候返回 NULL，否則返回輸入的值。它的語法如下：

```
NULLIF(input, value)
```

如果對 input=value 的檢測為 true，它返回 NULL，否則返回 input。

CASE 函數根據輸入的值選擇一組值中的一個。語法如下：

```
CASE
    WHEN expression
    THEN expression
    ...
ELSE expression
END;
```

在必要的情況下，可能有好多對 WHEN/THEN。整個運算式返回 WHEN 運算式運算結果為 true 的部分對應的 THEN 部分的運算式。如果沒有匹配到任何 WHEN 部分，則獲取到 ELSE 部分的運算式被計算。例如，以下的示例根據 n2 的值為 1,2 或者其他值，將 res 設置為 5,6 或者 7：

```
res := CASE
        WHEN n2 = 1
            THEN 5
        WHEN n2 = 2
            THEN 6
        ELSE 7
END;
```

迴圈

PL/pgSQL 有一套特別豐富的被稱為反覆運算控制結構（iterative control structure）迴圈機制，用於提供方法將代碼執行多次。

最簡單的迴圈是不帶控制條件的無限迴圈，迴圈會一直運行直到碰到一個 EXIT 語句：

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

```
END LOOP;
```

所有的 PL/pgSQL 的迴圈結構都可以打標籤，也就是可以被 BEGIN...END 包括。這個標籤用於作為 EXIT 語句的執行物件，用於中斷指定的迴圈（對於熟悉 C 語言的讀者，這就是 C 語言所缺乏的多級退出版本的 break）：

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

這將導致用 lable 作標籤的迴圈被終止，程式從這個迴圈後的語句繼續執行。這個 label 必須指向當前的迴圈或者一個靜態的迴圈。如果沒有給出標籤，則終止當前迴圈。如果給出了 WHEN 從句，則 EXIT 只會在 expression 的結果為 true 的情況下才被執行。

這裡有一個無限迴圈的示例：

```
<<indefinite>>
LOOP
        n := n + 1;
        EXIT indefinite WHEN n >= 10;
END LOOP;
```

WHILE 迴圈

WHILE 迴圈是對於無限迴圈的一個替代結構。它在條件保持為 true 的情況下一直執行一堆語句。語法如下：

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

FOR 迴圈

我們可以通過 FOR 迴圈執行固定次數：

```
FOR name IN [REVERSE] from .. to
LOOP
    statements
END LOOP;
```

這種類型的迴圈為整數運算式 from 和 to 之間的每個值執行一次循環體。這個迴圈建立了一個叫 name 的新變數。它依次獲得範圍內的每個值，在循環體執行的過程中每次增加一。

如果指定了 REVERSE，則迴圈以相反的方向運行，也就是變數每次減一。以下為一個 FOR 迴圈的示例：

```
FOR cid IN 1 .. 15
LOOP
        SELECT * INTO row FROM customer
        WHERE customer_id = cid;
        -- process a customer
END LOOP;
```

迴圈執行了 15 次，變數 cid 取值從 1 到 15，每次取一個使用者到 row 變數中。迴圈的上下界限可以是運算式，所以我們可以掃描整個 customer 表，而不是僅僅掃描 15 個，就像以下這樣：

```
SELECT COUNT(*) INTO ncustomers FROM customer;
```

```
FOR cid IN 1 .. ncustomers
...
```

一個更好的可選方案是 FOR 迴圈允許我們為任意的一個 SELECT 語句的結果逐行執行：

```
FOR row IN SELECT ...
LOOP
    statements
END LOOP;
```

對於 SELECT 返回的每一行，變數 row 被賦值且循環體被執行。用於存儲這一行的變數必須被預先定義為 record 或者 rowtype 類型。除非迴圈被 EXIT 終止，否則在迴圈會執行到最好一行被處理。

以下為在 psql 中執行的一段用於列印所有客戶的姓的程式碼片段示例：

```
DECLARE
    row record;
BEGIN
    FOR row IN SELECT * FROM customer
    LOOP
        RAISE NOTICE 'Family Name is %', row.lname;
    END LOOP;
...
END
```

至此，我們遇到了所有的 PL/pgSQL 中的所有程式結構，是時候把它們拼湊到一起使用了。

嘗試：建立一個存儲過程

假設我們想使用我們的示例資料庫來在我們庫存較低的時候幫助我們從供應商訂貨。我們已經有了 stock 表來追蹤我們要銷售的項目。我們想有能力使用這些資訊來自動向我們的供應商發起訂單。

以下為一個實現了自動補貨目標的第一步的存儲過程。以下的 reorder 函數掃描 stock 表中小於指定值的專案。它為這種專案寫一條記錄到新建的 reorders 表中。

```
-- Drop if necessary and create a temporary table for raising orders
drop table reorders;
create table reorders
(
  item_id integer,
  message text
);

-- reorders
-- scan the stock table to raise reorders of item low on stock
create function reorders(min_stock int4) returns integer as $$
declare
    reorder_item integer;
    reorder_count integer;
    stock_row stock%rowtype;
    msg text;
```

```
begin
    select count(*) into reorder_count from stock
           where quantity <= min_stock;
    for stock_row in select * from stock
                     where quantity <= min_stock
    loop
        declare
            item_row item%rowtype;
        begin
            select * into item_row from item
               where item_id = stock_row.item_id;
           msg = 'order more ' ||
                        item_row.description || 's at ' ||
                        to_char(item_row.cost_price,'99.99');
           insert into reorders
                values (stock_row.item_id, msg);
        end;
    end loop;
    return reorder_count;
end;
$$ language plpgsql;
```

保存以上代碼為 sproc.sql。

當我們建立這個函數並以最小庫存級別為 3 執行它，我們獲得的結果為 3，表示有當前庫存量小於等於 3 的專案為 3 條：

```
bpfinal=# \i sproc.sql
DROP TABLE
CREATE TABLE
CREATE FUNCTION
bpfinal=# SELECT reorders(3);
 reorders
----------
        3
(1 row)
bpfinal=#
```

補貨表 reorders 被庫存量不足的三個項目的識別字填充：

```
bpfinal=# SELECT * FROM reorders;
 item_id |               message
---------+-------------------------------------
       2 | order more Rubik Cubes at   7.45
       5 | order more Picture Frames at   7.54
      10 | order more Carrier Bags at    .01
(3 rows)
bpfinal=#
```

我們可以通過查詢 stock 表來驗證：

```
bpfinal=# SELECT * FROM stock;
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       4 |        8
       5 |        3
       7 |        8
       8 |       18
      10 |        1
(7 rows)
bpfinal=#
```

解析

函數 reorder 使用一個 SELECT 語句來獲取 stock 表中所有的低庫存量的行。一個 LOOP 用於反覆運算處理 SELECT 的結果。循環體使用一個 INSERT 為結果集的每一行添加資料到 reorders 表。函數 reorders 返回的結果則是通過另一個 SELECT 語句從 stock 表獲取到的符合條件的記錄數。

我們可以使用 CREATE TEMPORARY TABLE 來建立 reorders 表。這種情況下，這種情況下，表會在我們推出資料庫會話的時候被刪除。我們在這裡使用 CREATE TABLE 因而 reorders 表是持久的，因為我們可能在的訂單程式可能和我們的庫存檢查是不在一個程式中的。

要完成自動訂單系統，我們還需要分解專案並將訂單提交給供應商。我們也許需要為每個產品分別設立最小庫存級別並添加到 item 表中。統樣，每項產品需要的訂單量也是不同的，也許是基於之前的銷售記錄或者季節性因素。

動態查詢

通常，一個存儲過程裡頭的資料庫查詢是固定的或僅僅是帶參數的。我們通常根據指定的列從一個表查詢符合條件的資料或者根據新的列的值插入或者修改一行。這個可以通過 SELECT，INSERT 和 UPDATE 語句使用保存符合條件的值或者列的值到變數中來實現：

```
INSERT INTO reorders VALUES (stock_row.item_id, msg);
```

但有一些特殊情況我們需要可以使用變數的值來指定一個表或者列名來實現一個操作。PostgreSQL 不允許這樣，因為它需要可以僅僅優化查詢一次，而不是每次只需查詢的時候都做優化。但是，PL/pgSQL 支持一個通用的 EXECUTE 語句允許我們執行任意的 SQL 語句，通過給出一個字串：

```
EXECUTE query-string
```

這個查詢的字串可以是存儲過程中使用之前碰到的字串運運算元動態生成的。需要特別注意的是字串中的用引號括起來的名字和文字需要保證正確。有兩個函數說明我們做到這點。所有的表名和列名應該使用 quote_ident 處理，它能為查詢生成合適的字串。值應該使用 quote_value 處理。

以下為一個使用名稱和值的變數建立通用更新語句的示例：

```
EXECUTE 'UPDATE '
|| quote_ident(tablename)
|| ' SET '
    || quote_ident(columnname)
```

```
    || ' = '
    || quote_literal(columnvalue)
    || ' WHERE '
...;
```

注意這是一種低效的訪問資料庫的方法，因為任何生成的查詢在其被執行的時候都必須被分析和生成執行計畫。

動態查詢也可以在 FOR 迴圈中用在通過 SELECT 反覆運算訪問記錄時候使用。語法如下：

```
FOR row IN EXECUTE query-string
LOOP
    statements
END LOOP;
```

SQL 函數

雖然本章專注於講解使用 PL/pgSQL 作為建立存儲過程的一種方法，但也有可能使用 SQL 來建立函數。要做到這點，我們只需要指定存儲過程語言為'sql'並使用 PostgreSQL 的 SQL 語句代替 PL/pgSQL。

和 PL/pgSQL 一樣，SQL 函數也可以有參數，被依次映射到$1，$2 等。在函式定義中，$1 自動被函式呼叫的第一個參數代替，$2 被第二個代替，依此類推。沒有控制結構，你被限制在使用 PostgreSQL 的 SQL 語句。PL/pgSQL 包含一些功能例如變數，條件判斷和迴圈，而 SQL 函數只允許使用參數替換。SQL 函數返回的值是最後一條 SQL 語句執行的結果，通常是一個 SELECT。

使用 SQL 寫存儲過程的好處是你不需要載入 PL/pgSQL 語言處理模組到資料庫中。

雖然沒有迴圈，以下的 SQL 函數允許我們從函數中返回不止一行資料。

```
CREATE FUNCTION sqlf(text) RETURNS setof
```

如果我們生命函數的返回類型為 setop 類型，並使用了適當的 SELECT，我們可以實現返回多行。以下函數返回指定城鎮中所有的客戶：

```
CREATE FUNCTION sqlf(text) RETURNS setof customer AS $$
        SELECT * FROM customer WHERE town = $1;
$$ language sql;
```

當我們在 psql 中運行這個函數，將返回三行資料：

```
bpfinal=# SELECT sqlf('Bingham');
                                  sqlf
------------------------------------------------------------------------
 (7,"Mr  ",Richard,Stones,"34 Holly Way",Bingham,"BG4 2WE   ","342 5982"
 (8,"Mrs ",Ann,Stones,"34 Holly Way",Bingham,"BG4 2WE   ","342 5982")
 (11,"Mr  ",Dave,Jones,"54 Vale Rise",Bingham,"BG3 8GD   ","342 8264")
(3 rows)


bpfinal=#
```

早於 8.0 版本的 PostgreSQL 的 psql 無法一次處理一整行資料，所以會返回一個錯誤號或者返回對應行的 OID。所有的版本都允許我們選擇一個列。要實現它，我們使用語法 column(function())來獲取需要的列。這裡，我們列出 Bingham 的客戶的名字並給結果列一個名字：

```
bpfinal=# SELECT lname(sqlf('Bingham')) AS customer;
 customer
----------
```

```
 Stones
 Stones
 Jones
(3 rows)

bpfinal=#
```

觸發器

在我們的存儲過程示例中，我們開發了一個用於發現哪些產品需要進貨的函數。這個函數寫了一些提示消息到 reorders 表中。要讓它用起來，我們需要確保這個過程在某些條件下被執行，也許是在每天的定時批量任務中做。但是最好能夠找到一種方法自動確保 reorders 表總是最新的，而不需要編寫我們的用戶端程式來不停更新其中的專案。

在第八章，我們碰到了參照完整性這個概念——確保我們資料庫中的資料總是有意義。例如，如果我們刪除一個客戶，我們需要確保所有的跟這個使用者相關的訂單歷史記錄同時被刪除。我們在之前已經看到了 PostgreSQL 中確保這種完整性的約束。

對於一些應用程式，僅有約束是不夠的。假設我們想要避免刪除仍然有未完成訂單的客戶，而允許刪除所有的已經發送訂單的客戶。

我們在第八章看到了列級和表級的約束可以用於確保更複雜的資料完整性的規則，但這些規則實際上是靜態的。我們可以指定某個相關的行必須存在，或者執行一個當相關的行存在時某些行必須存在的規則。然而，我們我們無法指定複雜的條件，例如當其他條件也為 true 的時候一個行必須不存在。我們甚至無法在行被添加或者刪除的時候進行其他複雜的用戶定義行為。

解決這個問題的方法之一是使用*觸發器*。通過觸發器，我們可以讓 PostgreSQL 在發生一些行為例如 INSERT，DELETE 或者 UPDATE 一個表的時候執行一個過程。

過程和觸發器的組合給我們直接在資料庫中執行相當複雜的業務規則的能力。就像在第八章說的，最好的確保資料的業務規則的地方是在資料庫中。

要使用觸發器，我們需要先定義一個觸發器過程。然後我們可以創建觸發器，它定義了什麼時候觸發器過程將被執行。

定義觸發器過程

觸發器在符合條件的時候被觸發，然後會執行一種特殊類型的存儲過程，叫做*觸發器過程*。觸發器過程和存儲過程非常類似，但由於其特殊的調用方式，其編寫會更加受限。

觸發器過程是一種沒有參數且返回類型為特殊的 trigger 類型的函數。

注：7.3 版本之前的 PostgreSQL 使用返回類型 opaque 而不是 trigger。返回類型 opaque 返回數值型別用於無法被 PostgreSQL 直接操作的函數返回值。

PostgreSQL 會在特定表發生改變的時候調用觸發器過程。這個過程必須要麼返回 NULL 要麼返回符合觸發器過程被調用的表的結構的一個行。

對於在 UPDATE 之後調用的 AFTER 觸發器，建議觸發器過程返回 NULL。對於 BEFORE 觸發器，返回的結果會直接用於更新操作。如果觸發器過程返回 NULL，則 UPDATE 不會被執行。如果資料行被返回了，它將用作 UPDATE 的資料來源，在資料提交到資料庫前給觸發器過程機會改變它。

建立觸發器

使用命令 CREATE TRIGGER 命令建立觸發器，語法如下：

```
CREATE TRIGGER name { BEFORE | AFTER }
    {  event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE  func  ( arguments )
```

在這裡，event 可以是 INSERT，DELETE 或者 UPDATE。

觸發器有效地說：“在這個表上發生這個事件的時候調用這個存儲過程”。

需要給觸發器一個名字，用於在不需要這個觸發器的時候刪除它，語句如下：

```
DROP TRIGGER name ON table;
```

觸發器在某些事件發生時被觸發——DELETE，INSERT 或者 UPDATE。我們可以要求觸發器在事件發生後被觸發；在這種情況下，被調用的過程將可以訪問原始資料（被更新和刪除）和新資料（被插入和更新）。我們也可以要求觸發器在事件發生前被觸發。這種情況下，我們可以避免更新操作的發送，或者修改被插入或者更新的資料。我們可以通過使用 OR 分隔的事件列表指定不止一個事件。

正如我們所知，一些 SQL 語句可以影響多行資料。當多行更新導致觸發器被觸發，我們可以選擇是否為更新的每一行或者為整個更新觸發一次觸發器。我們通過指定 ROW 來執行多次，否則，我們指定 STATEMENT。

傳遞給函數的參數，可以用來區分類似的觸發器，所以一個函數可以用於多個觸發器。

要自動更新我們的 reorders 表，我們需要建立一個叫做 reorder_trigger 的存儲過程，並建立一個在表 stock 表被變更的時候調用它的觸發器：

```
CREATE TRIGGER trig_reorder
AFTER INSERT OR UPDATE ON stock
FOR EACH ROW EXECUTE PROCEDURE reorder_trigger(3);
```

注意觸發器過程（本例中的 reorder_trigger）必須在建立觸發器前被建立。

我們使用觸發器過程參數來傳遞一個最小庫存閾值，在本例中為 3。

嘗試：建立一個觸發器

以下為一個觸發器過程用於在 stock 表發生調整的時候更新 reorders 表的示例：

```
create function reorder_trigger() returns trigger AS $$
declare
    mq integer;
    item_record record;
begin
    mq := tg_argv[0];
    raise notice 'in trigger, mq is %', mq;
    if new.quantity <= mq
    then
        select * into item_record from item
        where item_id = new.item_id;
        insert into reorders
          values (new.item_id, item_record.description);
    end if;
```

```
    return NULL;
end;
$$ language plpgsql;
```

現在我們有了觸發器過程，我們可以這樣定義一個觸發器：

```
CREATE TRIGGER trig_reorder
AFTER INSERT OR UPDATE ON stock
FOR EACH ROW EXECUTE PROCEDURE reorder_trigger(3);
```

從一個指令檔中載入這個函數和觸發器的定義：

```
bpfinal=# \i sproc.sql
...
CREATE FUNCTION
CREATE FUNCTION1;
CREATE TRIGGER

bpfinal=#
```

然後嘗試調整一個項目的庫存讓它降低到 3 或者更少：

```
bpfinal=# UPDATE stock SET quantity = 3 WHERE item_id = 1;
NOTICE:  in trigger, mq is 3
UPDATE 1
bpfinal=#
```

我們可以發現觸發器被觸發並發出一個通知給我們。對庫存的修改執行了，同時觸發器也更新了 reorders 表，添加了一個新行：

```
bpfinal=# SELECT * FROM reorders;
 item_id |   message
---------+-------------
       1 | Wood Puzzle
(1 row)

bpfinal=#
```

解析

觸發器過程在 stock 表中發生 INSERT 或者 UPDATE 操作的時候總是被觸發。它在 UPDATE 操作發生後檢查庫存量，如果庫存小於給出的最小閾值，則添加一條記錄到 reorders 表中。

你可能注意到了我們在觸發器過程中使用了一大堆新的功能。首先，觸發器過程的參數不在是$1，$2 一類的了。自動觸發的過程是通過一大批特殊變數之一實現參數傳遞的。這個以陣列方式傳遞的參數名為 TG_ARGV，從 TG_ARGV[0] 開始。觸發器過程使用的特殊變數在表 10-12 中列出。

其次，在觸發器過程中，有兩個非常特殊的記錄：OLD 和 NEW。它們包含來自於（對於 ROW 類型的觸發器）被 UPDATE 影響的行所觸發的觸發器所影響的行的資料。就像你猜到的，OLD 包含 UPDATE 之前的資料，而 NEW 包含更新後的資料（或者為 BEFORE 類型的觸發器所建議的 UPDATE 操作的資料）。

表 10-12 PostgreSQL 觸發器過程變數

變數	描述

NEW	包含新資料庫行的記錄
OLD	包含舊資料庫行的記錄
TG_NAME	包含觸發這個觸發器過程的運行的觸發器名字的變數
TG_WHEN	包含文本'BEFORE'或者'AFTER'的變數，依賴於觸發器類型
TG_LEVEL	包含文本'ROW'或者'STATEMENT'的變數，依賴於觸發器的定義
TG_OP	包含文本'INSERT'，'DELETE'或者'UPDATE'的變數，依賴於導致觸發器被觸發的事件
TG_RELID	觸發器被啟動的表所關聯的物件 ID
TG_RELNAME	觸發器被觸發的表的名字
TG_NARGS	整數型變數，包含在觸發器定義的時候傳入的參數個數
TG_ARGV	字串陣列，包含程式呼叫的參數，索引值從 0 開始；如果傳入無效的索引值則返回 NULL

嘗試：建立另一個觸發器

作為最終版本的示例，我們可以建立另一個觸發器函數，它將避免在客戶還存在未完成的訂單的時候被刪除。我們為將要被刪除的客戶檢查 orderinfo 表中的訂單的 date_shipped 列是否為 NULL，如果是，則不允許刪除。如果沒有未完成的訂單，我們則允許刪除繼續，但是我們需要通過刪除客戶的舊訂單來整理訂單，還有相關訂單的訂單專案資訊。

```
create function customer_trigger() returns trigger AS $$
declare order_record record;
begin
    -- about to delete a customer
    -- disallow if orders pending
    select * into order_record from orderinfo
    where customer_id = old.customer_id
           and date_shipped is NULL;
    if not found
    then
        -- all OK, delete of customer can proceed
        raise notice 'deletion allowed: no outstanding orders';
        -- for referential integrity we have to tidy up
        -- we will need to delete all completed orders
        -- but first delete the information about the orders
        for order_record in select * from orderinfo
            where customer_id = old.customer_id
        loop
            delete from orderline
               where orderinfo_id = order_record.orderinfo_id;
        end loop;

        -- now delete the order records
        delete from orderinfo
               where customer_id = old.customer_id;
        -- return the old record to allow customer to be deleted
        return old;
```

```
    else
        -- orders present return NULL to prevent deletion
        raise notice 'deletion aborted: outstanding orders present';
        return NULL;
    end if;
end;
$$ language plpgsql;

create trigger trig_customer before delete on customer
for each row execute procedure customer_trigger();
```

要驗證這個觸發器的行為，讓我們檢查一把。首先，標記一個訂單的發貨日期為 NULL，表明訂單還在處理中：

```
bpfinal=# UPDATE orderinfo SET date_shipped = NULL WHERE orderinfo_id = 3;
UPDATE 1
bpfinal=# SELECT * FROM orderinfo;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            1 |           3 | 2004-03-13  | 2004-03-17   |     2.99
            2 |           8 | 2004-06-23  | 2004-06-24   |     0.00
            4 |          13 | 2004-09-03  | 2004-09-10   |     2.99
            5 |           8 | 2004-07-21  | 2004-07-24   |     0.00
            3 |          15 | 2004-09-02  |              |     3.99
(5 rows)

bpfinal=#
```

現在嘗試刪除客戶編號為 15 的客戶，而其編號為 13 的訂單還未完成：

```
bpfinal=# DELETE FROM customer WHERE customer_id = 15;
NOTICE:  deletion aborted: outstanding orders present
DELETE 0

bpfinal=#
```

由於存在未完成訂單，會返回一個通知。我們會發現，刪除不影響任何一行，因為不允許刪除繼續。

刪除沒有未完成訂單的客戶將沒有任何問題，只是我們會先刪除 orderinfo 表中項目。這是因為我們在這個表中使用了約束來避免這種單獨的刪除。觸發器很好的處理了這個問題，並為我我們提供了另一種方法來實現參照完整性問題。

編號為 3 的客戶沒有未完成的訂單，所以我們可以直接刪除客戶，而讓觸發器做其他的收尾工作：

```
bpfinal=# DELETE FROM customer WHERE customer_id = 3;
NOTICE:  deletion allowed: no outstanding orders
DELETE 1

bpfinal=#
```

檢查相關表，我們會發現刪除客戶後，不會剩餘多餘的垃圾資料：

```
bpfinal=# SELECT * FROM orderinfo;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            2 |           8 | 2004-06-23  | 2004-06-24   |     0.00
```

```
            4 |          13 | 2004-09-03  | 2004-09-10   |     2.99
            5 |           8 | 2004-07-21  | 2004-07-24   |     0.00
            3 |          15 | 2004-09-02  |              |     3.99
(4 rows)

bpfinal=# SELECT * FROM orderline;
 orderinfo_id | item_id | quantity
--------------+---------+----------
            2 |       1 |        1
            2 |      10 |        1
            2 |       7 |        2
            2 |       4 |        2
            3 |       2 |        1
            3 |       1 |        1
            4 |       5 |        2
            5 |       1 |        1
            5 |       3 |        1
(9 rows)

bpfinal=#
bpfinal=# SELECT customer_id, fname, lname FROM customer;
 customer_id |   fname    |  lname
-------------+------------+---------
           1 | Jenny      | Stones
           2 | Andrew     | Stones
           4 | Adrian     | Matthew
           5 | Simon      | Cozens
           6 | Neil       | Matthew
           7 | Richard    | Stones
           8 | Ann        | Stones
           9 | Christine  | Hickman
          10 | Mike       | Howard
          11 | Dave       | Jones
          12 | Richard    | Neill
          13 | Laura      | Hardy
          14 | Bill       | O'Neill
          15 | David      | Hudson
(14 rows)

bpfinal=#
```

為什麼使用存儲過程和觸發器

有很多原因要求使用存儲過程和觸發器，以下列出了一些：

- 提供集中校驗：我們可以在一個地方對表的更新進行強制條件檢查，而不依賴我們的用戶端應用程式。如果條件需要改變，只需要修改一個地方。
- 追蹤改變：我們可以使用觸發器來建立一個審計線索，在表被更新的時候寫另外一個表。這可以記錄造成改變的使用者，時間，日期甚至改變的資料。
- 增強安全：通過使用 PostgreSQL 的 current_user 變數，我們可以執行我們自己的安全性原則。
- 防止刪除：我們可以使用一個觸發器來標記資料為需要在之後刪除，而不是在應用程式嘗試刪除它的時候就刪除掉。
- 為用戶端提供映射操作：我們可以使用觸發器和存儲過程來為我們的某些資料建立一個簡單的，單表的映射，讓我們用戶對它們的操作跟容易。例如，我們可以建立一個表並映射到 Microsoft Excel。當這個表被更新，我們可以更新“真實”表中的資料，其中可能有個表有很複雜的結構。

摘要

在本章，我們看到了一些可以用來擴展 PostgreSQL 的查詢的功能的方法。我們已經看到 PostgreSQL 提供了很多運運算元和函數來完善查詢和提取資訊。

PostgreSQL 支援的過程語言允許我們使用 PL/pgSQL，SQL 和其他語言開發相當複雜的服務端過程。這為資料庫伺服器提供了實現複雜的不依賴於用戶端的應用功能的機會。

存儲在資料庫中的存儲過程可以被應用程式調用，或者被觸發器在資料庫的表發生改變時被自動調用。這給了我們另一種方法來確保參照完整性。

對於簡單的參照完整性，最好是堅持使用約束，因為它們更簡單、高效，不容易出錯。而觸發器和存儲過程在你聲明的約束變得非常複雜的時候或者你想要實現的約束對於聲明方式太複雜的時候會帶來很大幫助。

到此為止我們已經介紹了一些先進的 PostgreSQL 的技術，在下一章，我們將前進到如何保障 PostgreSQL 資料庫的主題。