TURING 图灵程序设计丛书 数据库系列

Addison Wesley

MySQL Fourth Edition

MySQL技术内幕

（第4版）

[美] Paul DuBois 著
杨晓云 王建桥 杨涛 译

- MySQL权威专家力作
- 全面、深入、细致入微
- 一册在手，别无他求

人民邮电出版社
POSTS & TELECOM PRESS

Paul DuBois

Sun公司MySQL文档团队的技术作者、开源社区和MySQL社区活跃的技术专家，同时也是一名数据库管理员。他曾参与过MySQL在线文档的编写工作。除本书外，他还著有*MySQL and Perl for the Web*、*MySQL Cookbook*、*Using csh and tcsh*以及*Software Portability with imake*等书。

TURING 图灵程序设计丛书 数据库系列

MySQL Fourth Edition

MySQL技术内幕

（第4版）

[美] Paul DuBois 著
杨晓云 王建桥 杨涛 译

人民邮电出版社
北京

图书在版编目（CIP）数据

MySQL技术内幕 : 第4版 / (美) 杜波依斯
(DuBois,P.) 著 ; 杨晓云, 王建桥, 杨涛译. -- 北京 :
人民邮电出版社, 2011.7
（图灵程序设计丛书）
书名原文: MySQL, Fourth Edition
ISBN 978-7-115-25595-2

Ⅰ. ①M… Ⅱ. ①杜… ②杨… ③王… ④杨… Ⅲ. ①
关系数据库－数据库管理系统，MySQL Ⅳ.
①TP311.138

中国版本图书馆CIP数据核字(2011)第101759号

内 容 提 要

本书介绍了 MySQL 的基础知识及其有别于其他数据库系统的独特功能，包括 SQL 的工作原理和 MySQL API 的相关知识；讲述了如何将 MySQL 与 Perl 或 PHP 等语言结合起来，为数据库查询结果生成动态 Web 页面，如何编写 MySQL 数据访问程序；详细讨论了数据库管理和维护、数据目录的组织和内容、访问控制、安全连接等。附录还提供了软件的安装信息，罗列了 MySQL 数据类型、函数、变量、语法、程序、API 等重要细节。

本书是一部全面的 MySQL 指南，对数据库系统感兴趣的读者都能从中获益。

图灵程序设计丛书

MySQL技术内幕（第4版）

◆ 著 [美] Paul DuBois
译 杨晓云 王建桥 杨 涛
责任编辑 王军花
执行编辑 谢灵芝

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：56.75
字数：1 519千字 2011年 7 月第 1 版
印数：1－3 000册 2011年 7 月北京第 1 次印刷

著作权合同登记号 图字：01-2008-5832号

ISBN 978-7-115-25595-2

定价：139.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

前　言

无论是在商业、科研和教育等方面的传统性应用项目里，还是作为因特网搜索引擎的后端支持，RDBMS（Relational Database Management System，关系数据库管理系统）在许多场合都是一种极其重要的工具。良好的数据库系统对于管理和访问信息资源来说至关重要，但很多企事业单位都没有足够的财力建立起自己的数据库系统。从历史上看，数据库系统一直是价格昂贵的产品，无论是软件本身还是后续的技术支持，供货商从来都是漫天要价。此外，为了获得令人满意的性能表现，数据库引擎往往对计算机硬件要求很高，而这又将使数据库系统的运营成本大大增加。

计算机硬件和软件在最近几年里的发展已经使这种情况得到了改善。小型桌面系统和服务器的价格越来越低，性能越来越高，而为它们编写高性能操作系统正成为一种潮流。这些操作系统有的可以从因特网免费获得，有的可以通过价格低廉的CD获得。它们包括BSD Unix操作系统的几种变体（如FreeBSD、NetBSD、OpenBSD等）以及各种各样的Linux发行版本（如Fedora、Debian、Gentoo、SuSE等）。

免费的操作系统因诸如GNU C编译器gcc之类的免费开发工具的发展而日臻完善。让任何人都能得到想要的软件，这正是开源运动的一部分。开源项目已经为我们提供了很多重要的软件产品，如因特网上使用范围最广的Web服务器Apache，以及广泛应用的通用脚本语言Perl、Python和Ruby，还有非常便于编写动态Web页面的PHP语言等。与此形成鲜明对比的是，如果决定采用某种专有的商业化解决方案，就不得不忍受供货商漫天要价，而且还极有可能根本看不到它的源代码。

开源运动也使免费的数据库软件和数据库系统越来越容易获得。例如MySQL就是一种免费的数据库系统，它是一种客户/服务器模式的关系数据库管理系统，最初起源于欧洲的斯堪的纳维亚半岛。MySQL由以下组件构成：一个SQL服务器、一些用来访问该服务器的客户程序、一套用来对数据库进行管理的软件工具，以及供用户自己编写程序的编程接口。

MySQL起源于Michael Widenius（外号Monty）在1979年为瑞典的TcX公司开发的一套名为UNIREG的数据库工具。到了1994年，TcX公司开始寻求一种能够用来开发Web应用的数据库服务器。TcX公司对几种商业化的服务器进行了测试，对它们在处理TcX公司的大数据表时的速度都不太满意。该公司还测试了mSQL，它缺少某些必要的功能。因此，Monty开始开发一种新的服务器。因为mSQL有一些免费的软件工具，所以新服务器的编程接口被有意设计成与mSQL所使用的编程接口非常相似。采用相似的编程接口将大大减少把那些免费的软件工具移植到MySQL的工作量。

到了1995年，Detron HB公司的David Axmark开始在因特网上推广和发行TcX公司研发的MySQL。David为MySQL编写了许多文档，增加了利用GNU组织的configure工具进行安装配置的功能。适用于Linux和Solaris系统的MySQL 3.11.1的二进制版本于1996年面世。如今，MySQL不仅能够在许多种计算机平台上运行，还同时提供二进制版本和源代码版本。MySQL在开源许可证和商业许可证下的发布、技术支持、监控服务和培训工作以前由MySQL AB公司专门负责。Sun公司在2008年收购了MySQL

AB公司，但保持了MySQL的开源特色（Sun公司的许多产品现在都可以在开源许可证下获得和使用）。

早期的MySQL广受欢迎的主要原因是它的速度和简单性，但因为缺少诸如事务处理（transaction）和外键支持（foreign key support）之类的高级功能，所以也有一些批评的声音。MySQL的开发和完善工作从未停止，发展至今，事务处理、外键支持、复制（replication）、子查询、存储过程、视图和触发器等功能都已被添至其中。这些功能让MySQL进入了企业级数据库软件的行列。结果，许多原来只考虑大型数据库系统而对MySQL不屑一顾的用户开始认真评估MySQL了。

MySQL的可移植性非常好，它可以运行在商业化操作系统（如Mac OS X、HP-UX和Windows）以及包括桌面电脑和企业级服务器的硬件平台上。此外，MySQL的运行性能绝不逊色于任何一种数据库系统，即使面对容纳着几十亿条数据记录的大型数据库，它也能游刃有余。在商业领域里，MySQL的地盘一直在扩大，这是因为许多公司的老板发现，与购买商业化许可证和技术支持服务相比，只需花一点零头就可以满足数据库处理需求。

未来我们将可以在功能强大但价格低廉的硬件设备上运行免费的操作系统，将有越来越多的人和商业机构在各种各样的硬件系统上拥有强大的计算能力和其他功能，MySQL则在其中起着重要作用。获得强大计算能力的经济成本的门槛正变得越来越低，大型数据库解决方案对普通用户和企业来说也已经不再是可望不可及的了。在过去，高性能的RDBMS只能出现在广大中小企业的梦想里，可现在，只需付出极低的成本和代价就能享用到这些东西。这一点对个人用户而言就更加突出了。就拿我本人来说吧，我有一台苹果笔记本电脑，在它的Mac OS X操作系统上，我同时使用着MySQL以及Perl、Apache和PHP。这使我能够随时随地工作，而这一解决方案的总成本只是笔记本电脑的价钱而已。

为什么要选用 MySQL

如果你正在寻求一种完全免费或者价格比较低廉的数据库管理系统，可以从MySQL、PostgreSQL和SQLite等软件中选择一个。在对MySQL和其他数据库系统进行评估之前，首先要弄清楚什么因素对自己最重要。你需要从运行性能、技术支持、特色功能（例如与SQL的兼容程度和可扩展性等）、许可证条件、购买价格等多方面进行全面的考虑。由此判断，MySQL在以下方面有比较吸引人的优势。

- **运行速度**。MySQL的运行速度相当快，MySQL开发人员相信它是目前最快的数据库系统。你可以在MySQL网站http://www.mysql.com/why-mysql/benchmarks/上的性能比较主页上查到有关数据。
- **易使用**。MySQL是一种简单易用的高性能数据库系统，与其他大型数据库系统相比，MySQL的安装和管理工作要容易得多。
- **查询语言支持**。MySQL支持SQL语言，SQL是各种现代数据库系统的首选查询语言。
- **功能丰富**。MySQL是多线程的，允许多个客户同时与服务器建立连接。每个客户都可以同时打开并使用多个数据库。你可以通过好几种办法（如命令行客户程序、Web浏览器、GUI客户程序等）对MySQL数据库进行交互式访问，在输入查询命令后立刻看到查询结果。此外，MySQL还准备了C、Perl、Java、PHP、Python和Ruby等多种语言的编程接口。你还可以通过支持ODBC（Open Database Connectivity，数据库开放连接，一种由微软公司开发的数据库通信协议）功能和.NET的应用程序来访问MySQL数据库。也就是说，你既可以选用现成的客户程序来访问MySQL数据库，也可以根据具体的应用来编写相关软件。
- **优异的联网和安防性能**。MySQL是完全网络化的数据库系统，用户可以从因特网上的任意地

点去访问它，因此你完全可以把你的数据拿出来与任何地方的任何人共享。同时，MySQL还具备完善的访问控制机制，这就将那些不应该看到你数据的人拒之门外。此外，为了提供更进一步的安防措施，MySQL还支持使用SSL（Secure Socket Layer，安全套接字层）协议的加密连接。

- **可移植性**。MySQL既能够运行在多种版本的Unix和Linux操作系统上，也能够运行在Windows和NetWare系统上。MySQL可以运行在各种硬件设备上，包括高端服务器。
- **短小精悍**。与某些数据库系统巨大的硬盘空间消耗量相比，MySQL发行版本的硬盘占用量相对要小得多。
- **成本低廉**。MySQL是一个开源项目，只要遵守GNU组织的GPL（General Public License）许可证条款，就可以任意使用。这意味着MySQL在大多数情况下都是免费的。其次，如果是喜欢或需要正规安排或是不想接受GPL许可证约束的组织，还有商业许可证可供选择。
- **来源广泛**。MySQL很容易获得，只要你有Web浏览器，就能从许多地方下载它。如果你想知道某个组件的工作原理，对它的某个算法感到好奇，或者想进行安全检查，你完全可以通过源代码来钻研它。如果你不喜欢它的某个组件，也完全可以自行加以修改。如果你自认为发现了一个bug，可以报告给相关开发人员。

MySQL的技术支持怎么样？这个问题问得好，不能提供支持的数据库系统没什么用。本书就是一种支持，希望它可以满足你在数据库方面的需要（本书既然是第4版了，就表明它能做到这一点）。你还可以利用其他一些MySQL的相关资源。

- MySQL的发行版本都带有《MySQL参考手册》（*MySQL Reference Manual*），有在线版和印刷版。MySQL用户对这本手册都给予了很高的评价。这一点非常重要，因为如果没有人知道如何使用，再好的软件产品也会贬值。
- 如果你想得到正规的培训或者专业的技术支持，可以报名参加Sun公司开设的培训课程，或者与该公司签订技术支持和跟踪服务合同。
- MySQL社区有一些非常活跃的邮件列表，任何人都能订阅。这些邮件列表有很多专家级的参与者，许多MySQL开发人员都是它的常客。作为提供技术支持的电子资源，它们被很多订阅者认为是物有所值。

MySQL大家庭（包括开发人员和普通用户在内）是一个团结互助的群体。贴在邮件列表上的求助帖子通常在几分钟内就会得到回复。如果有人报告说发现了一个bug并得到确认，开发人员就会马上发布一个修补方案并经由因特网迅速传遍整个社区。与此形成鲜明对照的是，某些大厂商提供的技术支持服务令人困惑，那种不得其门而入的感觉实在让人着急上火。你遇到过这样的情况吗？我遇到过。

如果你正打算挑选一种数据库产品，那么MySQL绝对是理想的候选。使用MySQL既无风险，也不需要花费金钱。如果你遇到了问题，还可以通过邮件列表寻求帮助。当然了，做这样的评估必定会花费一些时间，但无论你原来计划使用哪种数据库产品，反正都要花时间评估。与很多其他的数据库产品相比，安装和测试MySQL的时间肯定少得多。

如果已经在运行其他 RDBMS，该怎么办

如果你已经在使用某种数据库系统，但又颇受限制，那就绝对应该给MySQL一个机会。也许你觉得自己现有系统的性能不太好，也许它是一个专有产品而你又不想吊死在一棵树上，也许你想更换现

有的硬件设备而现有的软件系统却不支持，也许你现有的软件都是二进制代码而你更希望得到一种能够提供源代码的系统，也许你只是嫌它花钱太多……这一切都是你应该给MySQL一个机会的理由。你可以先通过本书熟悉一下MySQL的功能，再到MySQL邮件列表上提几个问题，然后再根据具体情况慎重抉择。

要明确的是，尽管所有主要的数据库引擎都支持SQL语言，但每种引擎支持不同的“方言”。请查阅本书关于MySQL所支持的SQL“方言”以及相关数据类型的章节。你也许会发现它们与你目前使用的RDBMS所支持的SQL版本区别太大，因而需要付出巨大的努力才能把你的应用程序迁移到MySQL系统上来。

当然，作为评估工作的一部分，应该先通过几个例子看看效果。这会让你在评估时获得宝贵的实际体验。MySQL的研发人员一直在努力让MySQL符合SQL语言标准，其效果之一就是让数据库应用程序迁移道路上的障碍随着时间的推移而不断减少，所以你的迁移工作很可能会比预期的容易许多。

MySQL 提供的软件工具

MySQL的发行版本都附带以下几种工具程序。

- **一个SQL服务器**。运转整个MySQL的引擎，对MySQL数据库的访问和操作都要通过它才能实现。
- **客户程序和工具程序**。其中包括一个供你直接提交查询并查看其结果的交互式客户程序以及几个用来对数据库站点进行管理和维护的工具程序。有一个工具程序用来监控MySQL服务器，另外几个工具程序则负责数据的导入、备份、数据表问题检查等。
- **一个供你自行开发应用程序的客户端库**。这个函数库是用C语言写的，所以你可以用C语言来编写客户程序。此外，这个函数库还提供了一些供其他语言（如Perl、PHP和Ruby）使用的第三方接口。

除MySQL本身提供的软件外，有很多聪明人也使用MySQL编写一些小程序来提高工作效率，并把自己的成果拿出来与大家分享。在这些第三方工具软件里，有些能帮助你更加得心应手地使用MySQL，还有一些把MySQL的功能进一步扩展到Web站点建设等方面中。

本书能让你学到哪些东西

通过阅读本书，可以高效地掌握MySQL的使用方法，从而高效地完成自己的工作。你将会学到怎样把信息资料录入数据库，怎样构造出查询语句以迅速获得有关问题的答案。

即使不是程序员，也可以学习和使用SQL。本书内容的重点之一就是介绍SQL的工作原理。但熟悉SQL的语法并不代表你掌握了SQL的使用技巧，所以本书的另一个重点就是介绍MySQL的独特功能及其用法。

你将学习如何把MySQL与其他软件工具结合起来。本书还将介绍如何通过MySQL与Perl或PHP语言来为数据库的查询结果生成动态Web页面，以及如何自行编写MySQL数据库访问程序。自行编写的程序会大大拓展MySQL的功能，满足应用项目的具体要求。

对于那些负责MySQL安装的人员，本书将为他们介绍有关职责及具体工作流程。你将学会如何建立用户账户，如何备份数据库，以及如何保证数据库的安全。

本书各章内容

本书内容分四部分。第一部分集中讨论数据库应用方面的概念。第二部分的重点是如何使用MySQL编写你们自己的程序。第三部分的目标读者是数据库管理员。第四部分是几个参考附录。

第一部分：MySQL 基础知识

第1章主要包括MySQL的用途与用法、交互式MySQL客户程序的使用方法、SQL基础知识和MySQL的常用功能。

如今，各种主流的RDBMS都能识别和理解SQL语言，但各种数据库引擎所使用的SQL语言彼此有着细微的差异。第2章重点介绍使MySQL有别于其他数据库系统的特色功能。

第3章主要包括MySQL为存储信息而提供的数据类型、各种类型的特点和局限性、它们的使用时机和使用方法，以及如何在相似的数据类型中作出选择，还有表达式的求值办法和各类型之间的转换机制等。

第4章讨论如何编写和使用存储在服务器端的SQL程序，包括各种存储函数、存储过程、触发器和事件。

第5章讨论如何使查询有效地运行。

第二部分：MySQL 的编程接口

第6章介绍MySQL提供的几种API（Application Programming Interface，应用程序编程接口）以及本书所涉及的几种API之间的详细比较。

第7章讲述如何利用MySQL的C客户端库所提供的API来编写C语言程序。

第8章探讨如何利用DBI模块编写Perl脚本，包括独立的命令行脚本和用于网站的CGI脚本。

第9章介绍如何利用PHP脚本语言和PHP数据对象（PDO）的数据库访问扩展来编写用来访问MySQL数据库的动态Web页面。

第三部分：MySQL 的系统管理

第10章介绍数据库管理员的工作职责，以及如何让数据库站点成功运行。

第11章详细介绍MySQL数据子目录（即MySQL用来存放各种数据库文件、日志文件和状态文件的地方）的组织布局和内容。

第12章阐述如何在操作系统开启和关闭时正确完成MySQL服务器的开启和关闭，如何在MySQL系统里建立用户账户，如何维护日志文件，如何配置存储引擎，如何优化数据库服务器，以及如何运行多个服务器，等等。

第13章介绍如何提高MySQL的安防水平以抵御各种入侵和破坏（可能来自数据库服务器主机的其他用户和网络客户端），如何配置你的MySQL服务器以支持SSL上的安全连接。

第14章阐释如何通过预防性措施来降低灾难的发生几率，如何备份数据库，如何在灾难真的发生时（即使采取了预防性措施）尽快恢复系统的运转。

第四部分：附录

附录A介绍如何获得并安装本书所提到的主要工具和示例数据库文件。

附录B详细说明MySQL数据类型。

附录C探讨在SQL语句中用来编写表达式的操作符和函数。
附录D介绍 MySQL服务器维护的各个变量和SQL语句变量的用法。
附录E描述MySQL支持的每个SQL语句。
附录F介绍MySQL发行版本所提供的程序。

说明 附录G、H、I需要上网获取。先访问www.informit.com/title/9780672329388，注册后可获取它们。也可以访问www.kitebird.com/mysql-book来获取它们[①]。

附录G介绍MySQL C客户端库所提供的数据类型和函数。
附录H讨论Perl DBI模块提供的方法和属性。
附录I介绍PDO扩展为在PHP中支持MySQL而提供的方法。

如何阅读本书

阅读本书的任何地方时，都应该同时尝试示例。这意味着你一定要先在计算机上安装MySQL，再安装示例数据库sampdb的有关文件，本书的许多示例都要用到sampdb数据库。获得和安装有关组件的办法与步骤可以在附录A里查到。

如果你是一位MySQL数据库系统或SQL语言的新手，请从本书的第1章开始学习。第1章介绍了MySQL与SQL的基本概念和使用入门，对加快本书后续章节的学习有很大帮助。然后再前进到第2章、第3章和第4章去学习如何描述和使用你自己的数据。这样，你就能有针对性地探索各种MySQL功能了。

即使你已经具备了一些SQL知识，也应该从第2章和第3章入手。不同的RDBMS系统所实现的SQL功能也不同，你应该首先弄清楚MySQL与你所熟悉的其他RDBMS系统有何区别。

如果你已经有了一些MySQL方面的经验但还需要进一步了解某些特定操作的原理，请把本书当做一本参考大全并根据需要有选择地查阅。你将发现书后的各个附录非常有价值。

如果你想编写能访问MySQL数据库的程序，请从第6章开始去学习有关API的章节。如果你想为自己的数据库开发一些便于使用的基于Web的前端访问程序，或者想为自己的数据库网站开发一些后端程序来增添动态内容，请阅读第8章和第9章。

如果要对MySQL和自己正在使用的RDBMS进行比较评估，本书的几个部分将有所帮助。如果想了解MySQL与你现有的SQL系统有何异同，请阅读本书第一部分中专门讨论数据类型和SQL语法的章节；如果你打算自己开发应用程序，请阅读第二部分中讨论编程的章节；如果你想了解MySQL需要何种级别的数据库管理功能，请阅读第三部分中有关管理的章节。如果你现在还没使用数据库，但正在对MySQL和其他数据库系统进行比较以作出选择，这些内容对你也将有很大的帮助。

书中涉及的软件及其版本

本书的第1版主要围绕MySQL 3.22版展开讨论并简要地介绍MySQL 3.23版。第2版把讨论范围扩大到了MySQL 4.0系列和MySQL 4.1系列的第一个发行版本。第3版讨论MySQL 4.1和MySQL 5.0中最早的几个发行版本。

本书是第4版，讨论的是MySQL 5.0。具体而言，本书将讨论MySQL 5.0和5.1版，以及MySQL 6.0

① 相应的中文译稿可在图灵网站（www.turingbook.com）本书主页上免费注册下载。——编者注

中最早发行的几个版本。本书的大部分内容仍适用于5.0和更早的版本，但我们不会特别指出特定于老版本的地方。

MySQL 5.0系列已经达到了通用阶段（即所谓的GA版），也就是说它已被认为能够稳定地运行在日常生产环境里。因为在MySQL 5.0系列的早期发行版本里有大量的修改，所以建议大家尽量选择最新的版本。在我编写本书的时候，5.0系列的最新版本是5.0.64。MySQL 5.1系列现处于备选版开发（Candidate Development）阶段，应该很快就会达到通用阶段。如果你想试试诸如事件调度器或XML支持之类的功能，你将需要MySQL 5.1。

如果你正在使用的MySQL版本早于5.0，本书讨论的以下几项功能将不可用。

- MySQL 5.0中增加的存储函数和过程、视图、触发器、脚本输入处理、真正的VARCHAR类型以及INFORMATION_SCHEMA。
- MySQL 5.1增加的事件调度器、分区、日志数据表和XML支持。

如果需要了解老版本，请访问MySQL官方文档网站http://dev.mysql.com/doc/，在那里可以查到每个版本的《参考手册》。

请注意以下几个没在本书里讨论的主题。

- 一些MySQL Connector组件，用户可通过它们访问Java、ODBC和.NET程序。
- NDB存储引擎和MySQL Cluster组件，它们用来提供以内存为介质的存储机制、高可用性和冗余。细节问题请查阅《MySQL参考手册》。
- 诸如MySQL Administrator和MySQL Query Brower之类的GUI（Graphical User Interface，图形化用户界面）工具。这些工具有助于在窗口环境里使用MySQL。

如果需要下载这些产品或查阅它们的文档，请访问http://www.mysql.com/products/或http://dev.mysql.com/doc/。

至于书中涉及的其他一些主要软件，目前比较常见的版本都应该可以满足书中示例的需要。（请注意：PDO数据库访问扩展必须使用PHP 5或更高版本，而在PHP 4环境下无法工作。）各主要软件的最新版本如下所示：

软件包	版本
Perl DBI模块	1.601
Perl DBD::mysql模块	4.007
PHP	5.2.6
Apache	2.0.63/2.2.8
CGI.pm	3.29

书中提到的所有软件都可以在因特网上找到。附录A介绍如何获得并在自己的系统上安装MySQL、Perl DBI、PHP和PDO、Apache、CGI.pm等软件，如何获得本书通篇使用的sampdb示例数据库（其中包含本书讲述程序设计时会用到的示例程序）。

如果读者使用的是Windows，我将假设你有Windows 2000、XP、2003或Vista之类相对较新的版本。本书里讨论的某些功能，例如命名管道和Windows服务，较早的版本（Windows 95、98或Me）不支持。

排版约定

本书的排版要求如下所示。

- 文件名和命令等都用Courier字体表示。

- 命令中需要由读者输入的部分用**Courier加粗**表示。
- 命令中需要由读者替换为自己选择的内容的部分用*Courier斜体字*表示。

在需要进行交互操作的例子里，我将假设你会把命令输入到终端窗口或控制台窗口。为反映出上下文环境，我将通过命令行提示符来表明所运行的命令的执行环境。比如说，对于通过mysql客户程序输入的SQL语句，相应的命令行提示符将是mysql>。对于通过命令解释器输入的命令，提示符将是%，这个提示符表示命令可以在Unix系统或者Windows系统下使用，但你们看到的提示符到底是什么要取决于命令解释器。（对Unix用户而言，命令解释器就是你的登录shell；对Windows用户而言，它是cmd.exe或command.exe程序）。#提示符的意义比较特殊，它表示命令通过su或sudo命令由Unix系统上的root用户执行，而C:\>提示符则表示Windows系统上的专用命令。

下面的例子给出了一条应该从命令解释器输入的命令。%是提示符，不需要输入。为了输入这条命令，需要依次输入粗体字字符并用你自己的用户名替换斜体字：

```
% mysql --user=user-name sampdb
```

在SQL语句里，SQL关键字和函数名都用大写英文字母，而数据库、数据表、数据列的名称则全部用小写字母。

在语法描述中，方括号[]表示内容可选，可选的内容以垂直线字符|分隔。方括号内的列表是可选的，其具体内容应该是该列表里的某一个数据项。花括号{}内的列表是必不可少的，必须从该列表里选择一个数据项。

其他资源

如果你没能在本书里找到问题的答案，该怎么办呢？以下是一些软件的网站。

软件包	官方Web站点
MySQL	http: //dev.mysql.com/doc/
Perl DBI	http: //dbi.perl.org/
PHP	http: //www.php.net /
Apache	http: //httpd.apache.org/
CGI.pm	http: //search.cpan.org/dist/CGI.pm/

这些网站提供的信息资源有参考手册、常见问题答疑文档（Frequently Asked-Question，FAQ）和各种邮件列表等。

- **参考手册**。MySQL发行版本中自带的主要文档。这些文档的格式有很多种，网上还有它们的在线版本和可下载版本。
  PHP的使用手册也有好几种格式。
- **手册页面**。DBI模块及其MySQL专用驱动程序DBD::mysql的文档可以从命令行使用perldoc命令查阅。试试perldoc DBI和perldoc DBD::mysql命令。DBI的文档侧重于基本概念，而其MySQL专用驱动程序的文档则侧重于与MySQL有关的各种具体功能。
- **FAQ文档**。DBI、PHP、Apache各有各的FAQ文档。
- **邮件列表**。本书所涉及的一些软件有它们各自的邮件列表。如果你打算使用某个工具软件，那最好订阅一份与之有关的邮件列表。使用邮件列表上的归档文件也是个好主意。如果你不熟悉某个软件工具，你的很多问题就可能是很多前人已经问过（并得到回答）无数次的了；

你不必再提出类似的问题，因为它们的答案几乎都能在邮件列表的归档文件中搜索到。不同的邮件列表有不同的订阅方式，下面这些URL地址可以为你提供相应的帮助。

软件包	邮件列表订阅站点
MySQL	http: //lists.mysql.com /
Perl DBI	http: //dbi.perl.org /support/
PHP	http: //www.php.net /mailing-lists.php
Apache	http: //httpd.apache.org /lists.html

- **其他网站**。除官方网站外，书中涉及的某些软件工具还另有一些提供其他信息（如示例程序的源代码和热门文章）的网站。这些网站大都可以通过官方网站上的链接找到。

目　录

第一部分　MySQL 基础知识

第 1 章 MySQL 和 SQL 入门

本章的主要内容是 MySQL 关系数据库管理系统和 MySQL 所使用的 SQL（Structured Query Language，结构化查询语言）的基础知识。本章将介绍大家应该掌握的术语和基本概念，描述书中示例所用到的 sampdb 示例数据库和 MySQL 数据库的创建与交互操作。

如果你在数据库系统方面是一个新手，不能肯定自己是否需要这种东西，就应该从本章开始学习。如果你完全不了解 MySQL 或 SQL，也应该从作为入门指南的本章入手。已经具备一定的 MySQL 或其他数据库系统使用经验的读者可以跳过本章内容。但我希望大家至少要看看 1.2 节，熟悉一下 sampdb 数据库的用途与内容，因为我们将在全书的示例中反复用到它。

1.1 MySQL 的用途

本节将描述 MySQL 数据库系统的用武之地，让大家了解 MySQL 能用来做哪些事情以及它们会对你的工作有什么样的促进和帮助作用。如果你用不着这些描述就已经信服了数据库的作用——也许你心里正有一个难题，急于让 MySQL 运转起来以解决它——不妨立刻前进到 1.2 节。

从本质上讲，数据库系统只不过是一套对大量信息进行管理的高效办法而已。信息有各种来源。例如，信息可以是科研数据、商业账目记录、客户定单、体育比赛成绩、销售业绩报告、个人爱好资料、人事档案记录、bug 报告、学生考试成绩，等等。虽然数据库系统能处理各种各样的信息，但单纯因为其本身而安装并使用它却不见得有必要。假如某项工作已经有了一个很好的解决方案，而你却在“为使用而使用”的心理驱使下引入了一个数据库系统，那就太不明智了。商品采购清单就是一个很好的例子：在出发前，先把想买的东西列在一张纸上；到商店后，每买到一样东西，就把它从清单上划掉；等采购完毕时，这张纸就可以丢掉不要了。几乎没有人会因为这种事情去动用数据库。如果你有掌上电脑，你应该会用掌上电脑的记事本功能来处理商品采购清单，而不会选用数据库。

当需要组织和管理的信息很多或者信息本身很复杂时，仍手工处理数据记录就会变得力不从心，而使用数据库系统则会让这些事变得轻而易举。对那些每天要处理上百万项交易的大公司来说，数据库不可或缺。但即便是某位用户出于个人爱好而收集整理信息这种小规模操作，也可能需要一个数据库。数据库可以帮上大忙的场景并不难想象，因为即便是很少的信息也很难管理。请考虑以下情况。

- 你开了一家木工厂并雇了几位员工。你需要维护一份员工名单和一份工资表，得把自己在何时给哪些员工发过工资的事记下来，还得把工资总数加起来好向政府有关部门报税。你还需要记录工厂接过的每一单木工活，以及每单木工活都由哪几位员工负责完成等事情。

- 你开了一家汽车配件连锁店。为了完成顾客的订单，你需要随时了解某个零件在哪一家分店里还有库存。
- 你在长年的科研工作中积累了大量的数据。为了发表研究成果，你必须对这些数据进行筛选和分析。这是一个沙里淘金般的工作，需要从大量原始数据生成汇总信息，提取典型的数据来进行统计分析，再根据分析结果推导出结论来。
- 你是一名教师，需要记录学生的考试成绩和出勤情况。每进行一次考试或测验，就把学生们的成绩记录下来。把成绩记到成绩本上并不复杂，但今后想分析这些成绩时可就麻烦了。对各次考试的成绩进行排序以确定分数线，学期结束时为每位学生计算总评成绩，统计学生们的出勤情况，等等，你一定不会手工完成这些工作。
- 你在某机构担任秘书一职，负责维护该机构的成员名录。（这个机构可能是专业团体、俱乐部、交响乐团、健身俱乐部等。）你每年都要为大家打印一份成员名录，每当有成员资料发生变化，就得用字处理软件修改这份文档。文档使你的很多好想法都无法实现，所以你对目前的状况感到很厌倦。很难对成员名录进行多种排序，很难从中选出指定的部分（例如只列出人名和电话号码），很难把符合某种条件的成员（如需要延续成员资格的人）都找出来（所以你每个月都得把这些成员一个不落地找出来并给他们寄去续会通知）。你听说过“无纸办公”，知道它指的是电子化的办公形式，但你并不了解它对你有什么好处。成员名录已经是电子化的了，可具有讽刺意味的是，除了把名录打印成册以外，任何其他类型的工作都不容易完成。

在上面列举的这些场景里，有的信息量很大，有的信息量很小。但它们有一个共同的特点，即这些工作原先都是手工完成的，但引入一个数据库系统将大幅提高工作效率。

那么，诸如 MySQL 之类的数据库系统会给你带来哪些特别的好处呢？这要看你的具体需要和目标到底是什么。在上面那些例子里，不同的场景有着不同的要求。下面，我们将以一个常见的情况为例来说明数据库的作用。数据库管理系统通常被人们用来取代文件柜，而事实上，数据库系统也像一个巨大的文件柜，只是它里面已经有很多预先建立好的存档功能。与手工方式相比，以电子化手段来管理信息的优势非常明显，同时也非常重要。我们来看一个例子，假设你要为牙科诊所管理顾客资料。下面是一些 MySQL 的存档功能，可以为你带来的巨大帮助。

缩短信息记录的录入时间。当需要添加一项新的信息记录时，你用不着拉开文件柜的各个抽屉以确定需要把这条记录添加到什么地方。你只需把这条记录提交给存档系统，让它把该记录存放到适当位置。

缩短信息记录的检索时间。当需要查找某条信息记录时，你用不着亲自拉开文件柜的各个抽屉就能找到你想要的资料。如果你想给最近没来参加定期检查的人们发一封提醒信，就完全可以让存档系统去把这些人的资料查找出来。当然，这与你让另一位员工“把最近 6 个月没来参加定期检查的人查出来”的情况是不同的。如果你有一个数据库，就可以直接用下面这条看起来很奇怪的语句完成这项工作：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

如果你从没见过类似的东西，这条语句看起来会吓人。以前需要花费一小时的时间才能得到结果，而现在你只需一两秒钟就能完成，这一点还是相当吸引人的。（现在，请不要被这条奇怪的语句吓倒。用不了多长时间，你就不再会对它感到陌生了。事实上，等你学完本章内容，就会明白这条语句到底有什么含义了。）

灵活的信息检索顺序。你用不着按照当初存储记录的顺序（例如按患者的姓氏顺序）来检索它们。你可以让你的存档系统按你希望的任意顺序来提取信息：按姓氏也行，按医疗保险公司名称的顺序也行，按最近一次就诊时间的顺序也行，等等。

灵活的输出格式。找到想要的资料后，你不必再把它们手工抄写下来。你可以让存档系统把它们生成为一份名单。有时候，你需要把信息打印出来；有时候，你可能想把它们用在另一个程序里。（例如，在生成一份最近没来参加定期诊断的患者名单后，你可以把这些资料送到一个文字处理软件里去打印催诊通知，然后再寄给那些患者。）也许你只对汇总信息（如总共有多少人没来参加定期诊断）感兴趣。有了数据库，你就用不着再亲自统计这些人数了，存档系统会轻而易举地为你生成汇总信息来告诉你。

信息记录能同时被多名员工使用。在“有纸办公”的年代，如果有两个人同时需要查看同一份资料，第二个人就必须等第一个人把资料放回原处之后才能拿到它。有了 MySQL，你就能让多位员工同时使用同一份资料了。

信息记录的远程访问和电子传输。“有纸办公”只允许你在信息资料的存放地点使用它们，或者让别人给你复印一份送过来。电子信息记录则允许对这些记录进行远程访问或者电子传输。如果你的牙科诊所设有分支机构，分支机构里的医护人员就能从他们自己的办公地点存取资料了。你不再需要通过信使来传送这些资料。如果别人想获得记录却又没有与你一样的数据库软件，那么，只要他能使用电子邮件，你就可以把他想要的资料找出来并通过电子手段传送给他。

如果你曾经使用过数据库管理系统，就会亲身体会到上面列举的种种好处，而你现在的想法可能已经超越“电子文件柜”的范畴了。很多企事业单位现在都把他们的数据库与网站结合起来使用，这是一个很好的例子。现在，假设你所在的公司有一个库存商品数据库，每当有顾客打电话来询问仓库里是否存有某种货物及其价格时，服务台员工就会用到这个数据库。这只是数据库比较传统的用法。如果你所在的公司还有一个可供顾客访问的网站，就可以增加一项新的服务项目——为顾客提供一个查询页面，让他们自己查询商品库存情况和价格信息。这样，顾客就能迅速地获得他们想要的资料，这些资料是从你的数据库里查出来的，而你为他们提供的库存商品查询功能又是自动完成的。顾客很快就能获得资料，不必再拿着电话筒听占线忙音，也不必再受你们公司上下班时间的限制。而每一位使用你们公司网络的顾客都替你省下了一小笔需要支付给服务台员工的工资。（光是如此节省下来的钱恐怕就能抵消网站本身的开支了。）

你还可以更进一步地发挥数据库的作用。基于 Web 的库存查询功能不仅能满足顾客的需求，对你的公司也有莫大的好处。这些查询能让你了解顾客想买哪些商品，而查询结果则能让你了解你是否能够满足顾客的要求。如果你的仓库里没有顾客想要的东西，你可能会失去这笔生意。所以把来自顾客的库存查询信息（他们想买什么，你是否有足够的库存等）记录下来不失为一个好主意。你可以根据这些信息来调整库存，向顾客提供更好的服务。

说了半天，MySQL 到底是怎样工作的呢？寻找这一答案的最佳办法是亲身体验一下。为此，我们需要先建立一个示例数据库。

1.2　示例数据库

本节描述本书后续各章中使用的两个示例数据库。在你学习 MySQL 使用方法的过程中，这两个数据库将充当各示例的信息源。这两个示例数据库是我们根据此前介绍的以下两种场景生成的。

- 你担任某机构秘书一职。这个机构有一些特点。它是由一些对美国历史很感兴趣的人自发组织起来的，我们不妨把它称为美国历史研究会。出于个人的爱好，那些会员将自愿定期交纳一定的会费以维持其会员身份。会员交纳上来的会费将用于支付研究会的开支，如会员刊物*Chronicles of U.S. Past*（美国历史年表）的印刷费用。这个研究会建有一个小规模的网站，但还没有得到充分的开发和利用，你可能想改变它。
- 考试分数记录。你是一名教师，在学期中负责考试与测验，记录考试分数，给学生打分。在学期末，你对学生的成绩作出总评，并把总评成绩和出勤情况上报给学校办公室。

下面，我们进一步分析这两种场景的需求。

- 你需要决定想从数据库得到哪些东西——也就是你想达到的目标。
- 你需要决定想把哪些东西放到数据库里——也就是你想记录的信息。

“你想从数据库得到哪些东西”位于“你想把哪些东西放到数据库里”的前面，这看起来似乎有些本末倒置，因为无论如何，你得先把数据放到数据库里才能对它们进行检索。事情是这样的，如何使用数据库主要取决于你想达到的目标，而目标又主要取决于你将从数据库检索出来的东西而不是你放到数据库里的东西。如果你今后根本不会使用你放入数据库里的信息，那就用不着浪费时间和精力把它们放进去了。

1.2.1 “美国历史研究会”场景

在这个场景里，你担任着这个研究会的秘书职务。眼下，你正使用一份文档维护这个研究会的会员名录。当然，利用文档来打印会员名录还是很容易完成的，可要是还想利用它再做些别的事情就没那么容易了。你有以下几个目标。

- 你希望能够根据不同情况把会员名录以其他格式输出，只输出符合特定要求的资料。首先，每年生成一份全体会员的名录——这是研究会的传统工作之一。你还想利用会员名录做一些其他工作，例如向研究会提供一份最新的会员名单以便邀请他们出席研究会的周年宴会。这两项工作需要用到的信息是不同的：全体会员名录需要用到每位会员的完整资料，而周年宴会只需用到会员的姓名（这项工作可不容易用文档来完成）。
- 你希望能够根据各种限制条件有选择地找出部分会员来。比如说，你想知道近期有哪些会员需要续交年费以保留其会员资格。另外，你还希望能够根据一些关键词把会员们分类，这些关键词代表每位会员对美国历史上的不同时期的兴趣，比如“Civil War”（南北战争）、“Depression”（经济大萧条）、“civil right”（民权法案）、“Thomas Jefferson”（托马斯·杰弗逊）总统的生平事迹，等等。会员有时希望你能为他们提供一份与自己志趣相投的其他会员的名单，而你也非常想满足这些会员的愿望。
- 你还想把研究会的会员名录发布到研究会的网站上去。这对会员和你都有好处。如果你能通过某种自动化的过程把会员名录转换为Web页面，在线版本的会员名录将总是最新的，这是纸张形式无法达到的。如果这份在线名录还支持检索功能，会员们就能轻松地自行查找他们感兴趣的信息了。比如说，如果某位会员想知道谁还对南北战争感兴趣，他完全可以自己去查询，用不着等你去帮他查找，而你也能抽出时间去做些别的事情。

我得承认，数据库并不是世界上最令人激动的东西，所以我也不打算鼓吹说使用数据库能够激发人们的创造性思维。不过，如果你不把信息看成是要解决的难题（和你看待文档一样），而是可以轻

松操作的东西（你希望使用 MySQL 时是这样），就肯定会发现很多使用这些信息的新方法，如下所示。

- 如果数据库中的信息能够以在线名录的形式添加到网站上，这些信息流肯定会发挥出其他的作用。比如说，如果会员能够以在线方式修改本人的资料来刷新数据库，就用不着再由你来负责录入工作了，会员资料也更准确。你一定不想自己做这些修改工作，而研究会也没有预算再雇一个人。
- 如果你把电子邮件地址也添加到数据库里，就可以通过电子邮件来提醒会员注意更新自己的资料。在发给会员的邮件里，你可以附上他们的现有资料，让他们自己去核查这些信息并通过研究会网站提供的有关功能进行必要的修改。
- 数据库将大大拓展研究会网站的用途，并不仅仅局限于会员名录。研究会有自己的会刊 *Chronicles of U.S. Past*，每期会刊里都有一个专为儿童准备的历史知识测验栏目。假设最近几期会刊里的小测验题目都集中在美国总统的生平事迹方面。你在研究会的网站上也可以开设一个类似的儿童栏目并以在线方式给出历史知识小测验题目。这个栏目甚至可以办成互动形式——小测验的题目都由Web服务器以实时方式直接从数据库里提取出来并展示给站点访客。

太让人兴奋了！你脑子里想到的这些数据库应用让你有点忘乎所以了。在重新回到现实之后，你要开始考虑以下一些实际的问题。

- 这是不是有点野心勃勃？实现起来工作量是否太大？

  要知道，空想总是要比实干来得容易，我也不想虚伪地告诉大家说这些设想都很容易实现。不过，等你学习完这本书的时候，我们现在勾勒出来的这些设想将全部成为现实。请记住这样一个道理：一蹴而就是不可能的。我们将把这些工作分解，然后逐一实现。
- MySQL能够胜任所有这些工作吗？

  不，它不能，至少单靠它自己不能。例如，MySQL 不具备直接开发 Web 程序的功能，你必须把它与其他软件开发工具结合起来才能完善和扩展它的功能。

  我们将利用 Perl 脚本语言和 Perl 语言中的 DBI（Database Interface，数据库接口）模块来编写用来访问 MySQL 数据库的脚本程序。Perl 有着强大的文本处理功能，能够对数据库的查询结果进行极其灵活的处理并生成各式各样的输出。例如，我们可以用 Perl 生成一份 RTF（Rich Text Format，富文本格式）的会员名录，而任何一种字处理软件都能识别出这种格式，也可以使用用于 Web 浏览器的 HTML 格式。

  我们还将用到另一种脚本语言，PHP。PHP 特别适合用来编写 Web 应用，同时它也很容易与数据库合作。这将使你能够从 Web 页面来开始 MySQL 查询，再把数据库的查询结果包含在一个新生成的页面里。PHP 与一些 Web 服务器软件（包括世界上最流行的 Web 服务器软件 Apache）配合得非常好，这使得提供查询页面和显示查询结果很容易。

  MySQL 能够非常好地与这些工具集成在一起，你可以灵活组合这些开发工具以达到你心中的目标。有些套装开发工具声称自己有着极高的“集成度”，被宣传得几乎是无所不能，可实际上却只能实现套装组件间的配合，与其他组件的配合并不理想。大家千万不要被蒙蔽。
- 最后，也是最重要的一个问题：这要花费多少钱？别忘了，美国历史研究会并没有多少预算。也许有点让人难以置信，可采用上述组合的解决方案的确没有什么成本。如果你有一些关于数据库系统方面的常识，就该知道它们通常都很昂贵。与此形成鲜明对比的是，MySQL 几乎总是免费的。即便是在需要技术支持服务和维护作业的企业级环境里，使用 MySQL 作为数据库系统的成本也是相对低廉的。（详见 www.mysql.com 网站。）我们将使用的其他工具（Perl、

DBI、PHP、Apache）都是免费的，因此，如果把所有事情都考虑进来的话，你花很小的成本就可以组建一个实用的系统。

你可以任意选择操作系统来开发你的数据库。我们将介绍的软件开发工具全都能在 UNIX（包括 BSD UNIX、Linux、Mac OS X 等）和 Windows 操作系统上使用，但专门针对 UNIX 或 Winows 的 shell 脚本或批处理脚本例外。

1.2.2 考试记分项目

现在我们再去看看另一场景中使用的示例数据库吧。在这个场景里，你是一位负责记录学生考试成绩的教师。你想把手工记录的学生成绩簿转换到一个使用 MySQL 的电子系统上去。在这个场景里，从数据库里获取信息的方式与你目前使用学生成绩簿时差不多，如下所示。

- 每次测验或考试之后，都要把考生的成绩记录下来。如果是考试，你还需要对分数进行排序以评定级别（A、B、C、D、F）分数线。
- 在学期结束时，把学生这一学期的总分数计算出来，对这些总分数进行排序，由此评定级别。在计算总分数的时候，你可能需要进行加权计算以区别考试成绩和测验成绩，因为考试通常要比测验重要。
- 在学期结束的时候，还要向学校办公室提交学生的考勤情况。

你的目标是不再以人工方式对学生们的考试分数和出勤情况进行排序和汇总。换句话说，你希望每次考试后的分数排序工作，以及各学期末的学生总评分和出勤情况的统计工作都能够交给 MySQL 去完成。为此，你需要知道班级里的学生名单、他们每次考试或测验的分数，以及缺勤的学生姓名。

1.2.3 关于示例数据库的说明

如果你对我们安排的“美国历史研究会”或者“考试记分项目”没有什么兴趣，你可能想知道它们和你有什么关系。要知道，这些场景只是一些例子，它们本身并不是我们的学习目标。它们只是我们在学习使用 MySQL 及相关工具时需要用到的一种载体。只要稍微动点脑筋，你就能看出这些示例数据库上的查询对解决你本人真正关心的具体问题有帮助。我们不妨假设你在我前面提到的那个牙科诊所里工作。虽然你在这本书里看不到多少与牙科医学有关的查询，但书中的很多查询都能运用到患者资料或办公室资料的管理工作中。比如说，在“美国历史研究会”场景里，你需要把近期必须续费才能保持其会员资格的会员找出来；而这与你把近期需要来牙科诊所进行定期检查的患者找出来的情况是类似的——它们都是与日期有关的查询。因此，只要你学会了如何写出一个用来找出哪些会员需要续费的查询，就可以运用类似的原理写出一个用来找出哪些患者需要就诊的查询来。

1.3 数据库基本术语

你大概注意到，你已经看了好多页了，却至今仍未在这本讲数据库的书中遇到太多专业术语和技术词汇。事实上，到目前为止，虽然已经粗略描述过示例数据库的用法，但我仍没说过一句关于什么是数据库的话。可是，既然我们将要设计一个数据库并开始实现它，我们就无法继续回避有关的术语，它们正是本节要介绍的内容。本节将描述书中使用的一些术语，希望大家能够掌握它们的含义。值得庆幸的是，与关系数据库有关的概念大都比较简单。事实上，人们喜欢关系数据库的很大一部分原因就在于它们的基本概念都很简明易懂。

1.3.1　数据库的组织结构

在数据库的世界里，MySQL 被划分到关系数据库管理系统（Relational Database Management System，RDBMS）的范畴内。我们可以把这个短语划分为以下几个部分。

- 数据库（database，即RDBMS里的DB）就是一个用来存放信息的仓库，它们构造简单，遵守一定的规则：
  - 数据库里的数据集合都存放在数据表（table）里；
  - 数据表由数据行（row）和数据列（column）构成；
  - 一个数据行就是数据表里的一条记录（record）；
  - 记录可以包含多个信息项，数据表里的每一个数据列都对应一个信息项。
- 管理系统（management system，即RDBMS里的MS）指的是用来对数据进行插入、检索、修改、删除等操作的软件。
- 关系（relational，即RDBMS里的R）表示RDBMS是DBMS中的一种，这种DBMS的专长就是把分别存放在两个数据表里的信息联系（即相互匹配）起来，而这种联系是通过查找两个数据表的共同元素来实现的。RDBMS的威力在于它能方便地抽取出数据表里的数据并把它们与其他相关数据表的信息结合起来，为那些单独利用某个数据表无法找到答案的问题提供答案。（事实上，“关系”的正式定义与这里有所不同，为此我向纯粹主义者们表示道歉，但这里的定义有助于解释RDBMS的用途。）

关系数据库是如何把数据组织到数据表里的？又是如何把来自不同数据表的信息联系在一起的呢？我们来看一个例子。假设你有一个包含横幅广告服务的网站，并与一些想登广告的公司签订了合同。每当有访客点击了你的某个页面时，你就会把一条广告嵌入到页面里，将其发送给访客的浏览器。同时，你还会向刊登这条广告的公司收取一笔小小的费用。这样就是一次广告点击。为了记录这些信息，你使用了 3 个数据表（如图 1-1 所示）。company 数据表由以下几个数据列构成：公司名称（company_name）、公司编号（company_num）、地址（address）和电话号码（phone）。ad（广告）数据表由以下几个数据列构成：广告编号（ad_num）、拥有该广告的公司的编号（company_num）和该广告每被点击一次的计费标准（hit_fee）。hit（点击情况）数据表由以下几个数据列构成：广告编号（ad_num）和该广告被发送给浏览者的日期（date）。

有些问题只用一个数据表就能得到答案。比如说，如果你想知道有多少家公司与你签订了广告合同，你只要数一数 company 数据表总共有多少行就行了。如果你想了解在某个给定的时间段内有多少次广告点击，也只需用到 hit 数据表。可有些问题会比较复杂，需要查询多个数据表才能找出答案。例如，在 7 月 14 日这一天里，Pickles 公司的各条广告分别被点击了多少次？要想回答出这个问题，就必须把这 3 个数据表都用上，如下所示。

(1) 在 company 数据表里根据公司名称（Pickles, Inc.）查出对应的公司编号（14）。

(2) 利用这个公司编号在 ad 数据表里查找与之匹配的记录以确定相关的广告编号。我们找到了两个符合条件的广告，编号是 48 和 101。

(3) 利用这两个广告编号从 hit 数据表里把落在给定日期范围内的匹配记录找出来，再对匹配到的记录个数进行统计。最后，我们查出编号为 48 的广告有 3 个匹配，编号为 101 的广告有 2 个匹配。

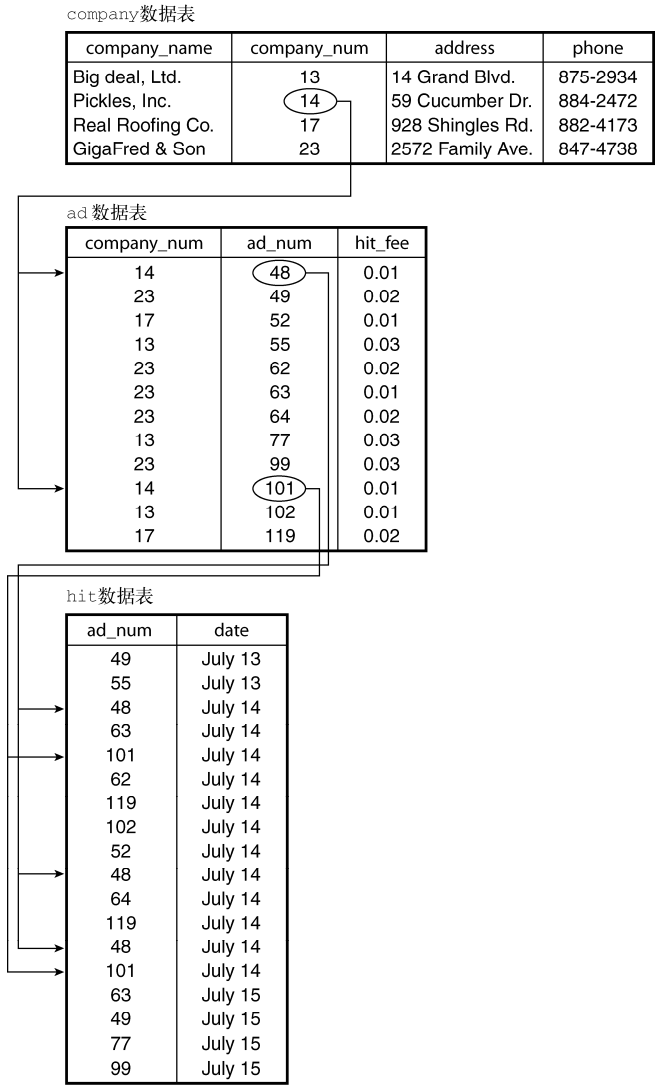


图 1-1 横幅广告数据表

听起来太复杂了，但这正是关系数据库系统所擅长的。而且，虽然看起来很复杂，但上面这几个步骤也只是几个简单的匹配操作而已：我们把一个数据表与另一个数据表联系起来，看前一个数据表的数据行取值是否与后一个数据表的数据行取值相匹配。把这几步简单的操作推广开来，我们就能找出各种问题的答案：各家公司分别有多少个不同的广告？哪家公司的广告最受欢迎？每个广告会带来多少收益？在本结算期内，每家公司应该支付你多少广告费？

现在，你对关系数据库理论的了解已经足以让你读懂本书后续章节的内容了，我也不想再用“第三范式”（Third Normal Form）、“实体联系图”（Entity Relationship Diagram）之类的枯燥概念去烦扰大家。（如果你想了解这些概念，我建议你去读读 C. J. Date 或 E. F. Codd 的著作。）

1.3.2　数据库查询语言

为了与 MySQL 交互，你需要使用一种名为 SQL（Structured Query Language，结构化查询语言）的语言。SQL 是今天的标准化数据库语言，在各种主流的数据库系统上都能使用（也有一些实现是供应商特有的）。SQL 中的各种语句使它能够高效率地与你的数据库进行互动。

与其他计算机语言一样，初次接触 SQL 的人往往会觉得它很奇怪。例如，在创建数据表的时候，你必须告诉 MySQL 这个数据表的结构是什么样的。很多人会把数据表想象成一个表格或者一幅图，但 MySQL 却不这么想，所以你在创建数据表时必须写出下面这样的代码：

```
CREATE TABLE company
(
  company_name CHAR(30),
  company_num  INT,
  address      CHAR(30),
  phone        CHAR(12)
);
```

这类语句往往会让 SQL 新手产生畏难情绪，不过请放心，不是程序员也能学会熟练运用 SQL。随着对这种语言的熟悉程度的加深，你对 CREATE TABLE 这类语句的看法也会悄悄地转变——它们不再是一团难以理清的乱麻，而是能帮助你描述信息的工具。

1.3.3　MySQL 的体系结构

MySQL 采用的是客户/服务器体系结构。因此，当你使用 MySQL 的时候，你实际是在使用两个程序。第一个程序是 MySQL 服务器程序，指的是 mysqld 程序，它运行在存放着你的数据库的机器上。它负责在网络上监听并处理来自客户的服务请求，根据这些请求去访问数据库的内容，再把有关信息回传给客户。另一个程序是 MySQL 客户程序，它们负责连接到数据库服务器，并通过向服务器发出查询命令来告知它们需要哪些信息。

MySQL 的大多数发行版本包括数据库服务器和几个客户程序（在 Linux 下使用 RPM 包时，有单独的服务器和客户 RPM 包，所以这两个都要安装）。你得根据自己的具体情况来选用一种客户程序。mysql 是最常用的客户程序，它是一个交互式的客户程序，你通过它发出查询命令并查看结果。mysqldump 和 mysqladmin 是两个主要用于数据库管理的客户程序，前者用来把数据表的内容导出到一个文件里，后者用来检查数据库服务器的工作状态和执行一些数据库管理方面的任务，例如通知数据库服务器停止运行等。MySQL 发行版本里往往还有其他的客户程序。MySQL 还提供了一个客户程序开发库，如果 MySQL 自带的标准客户程序不能满足你的应用要求，你就可以自行编写一些程序来解决问题。这个开发库可以从 C 语言程序里直接使用。如果你偏爱其他编程语言，还有其他语言（Perl、PHP、Python、Java、Ruby 等）的编程接口可供选用。

本书讨论的客户程序都是从命令行使用的。如果想试试使用 GUI（图形用户界面）并提供点击功能的工具，请访问 http://www.mysql.com/products/tools/。

MySQL 的“客户/服务器”体系结构有以下一些好处。

- 并发控制（concurrency control）由服务器提供，因而不会出现两个用户同时修改同一条记录的现象。来自客户的请求全都要经过服务器，由服务器来安排处理它们的先后顺序。即使出现多个客户同时请求访问同一个数据表的情况，也用不着由这些客户去发现对方并进行协商。它们只负责把请求发往服务器，而谁先谁后的事则完全由服务器去决定。

- 你不必非得在存放着你的数据库的那台机器上登录。MySQL 知道该如何在因特网上运行，所以你可以在任意地点运行 MySQL 客户程序，而这个客户程序能够通过网络寻找到服务器。地理距离根本不是问题，你可以从世界任何一个角落访问服务器。即使服务器位于澳大利亚而你带着一台笔记本电脑旅行到了冰岛，你也能访问到自己的数据库。可这是否意味着别人也能通过因特网看到你的数据呢？答案是“不”。MySQL 有一个灵活的安防系统，只有得到你授权的人才能访问你的数据。而且，你还可以进一步限制这些人只能做你允许他们做的事。比如说，财务部的 Sally 应该有查看和修改数据记录的权限，可服务部的 Phil 却只应该有查看它们的权限。总之，你可以把这种访问权限控制细化到每一个人。从另一方面讲，如果只想拥有一个完全属于你自己的系统，你也完全可以把访问权限设置成只允许客户程序从运行着服务器的那台主机上连接。

除原来以客户/服务器方式运行的 `mysqld` 服务器程序外，MySQL 又新增了一个库函数形式的服务器 libmysqld，你可以把它链接到程序里以编写出独立的基于 MySQL 的应用程序。因为它被嵌入在各个应用程序里，所以人们把 libmysqld 称为“嵌入式服务器库”（embedded server library）。嵌入式服务器与客户/服务器方案的主要区别是它不需要网络。这使我们能够方便地制作出这样一种“自给自足”的应用软件包来：它们对外部操作环境的要求更低，与数据库有关的操作完全由它自己来负责完成。这既是它的优点，也是它的局限性——如果机器里没有安装 MySQL 软件，其他软件包就无法访问该主机上的数据库了。

MySQL 与 mysql 的区别

为避免混淆，请注意，MySQL 指的是一个完整的 MySQL RDBMS，而 `mysql` 则是一个特定的客户程序的名字。它们的发音相同，但代表的却是不同的事物，所以本书要用大写和小写字母来区分它们。

说到发音，《MySQL 参考手册》给出的是“my-ess-queue-ell”。但 SQL 却有“sequel”和“ess-queue-ell”两种读法。

1.4 MySQL

好了，大家应该了解的预备知识也就是这么多。下面该让本书的主角 MySQL 出场了！

安排这样一节是为了帮助大家熟悉 MySQL 的基本用法。本节是这样安排的：首先创建示例数据库和几个必要的数据表，再通过示例数据库介绍如何对数据表里的信息进行插入、检索、删除、修改等操作。通过这些步骤，你将学到以下技能。

- MySQL能理解的SQL的基本知识。MySQL所使用的SQL语言与其他RDBMS使用的版本有着细微的差异。因此，即便你以前曾经接触过其他的RDBMS并有一定的SQL使用经验，也应该快速浏览一下本节的内容。
- 使用MySQL自带的标准客户程序与MySQL服务器通信。前面讲过，MySQL采用的是“客户/服务器”体系结构，服务器将运行在存放着数据库的主机上，而客户需要通过网络来连接到服务器上。本节的重点内容是`mysql`客户程序，它负责接收你发出的SQL查询命令，把它们发送到服务器执行，再把执行结果显示给你看。我们之所以会把`mysql`作为介绍重点，是因为它能在MySQL支持的任何一种平台上运行，是把你与数据库服务器直接联系在一起的纽带。本

节中的某些示例还用到了另外两个客户程序mysqlimport和mysqlshow。

我们给书中的示例数据库取名为 sampdb，但读者也许需要给它另外起个名字才行。比如说，也许在你的系统上已经有其他人把自己的数据库命名为 sampdb 了，或者你的 MySQL 管理员给这个示例数据库另外指定了一个名字。如果遇到这类情况，请把本书示例中的 sampdb 替换为你实际使用的示例数据库名称。

出现在本书示例中的数据表名称用不着任何改动就可使用，哪怕在你的系统里有多名用户都各自安装了一份示例数据库。在 MySQL 里，只要数据库的名字没有出现重复，数据库里的数据表是允许同名的。MySQL 将数据表限制在各自的数据库，防止互相干涉。

1.4.1　如何获得示例数据库

本节有时会用“sampdb 数据库发行版本”（或“sampdb 发行版本”，因为示例数据库的名字是 sampdb）来指称有关文件。这些文件里存放着用来安装示例数据库的查询命令和数据，它们的获取办法和安装步骤可以在附录 A 里查到。解压缩之后，安装过程将自动创建一个名为 sampdb 的子目录并把有关的文件存放到里面。顺便给大家提个建议：在拿 sampdb 数据库练手之前，最好先切换到这个子目录里。

如果想无论当前在哪个子目录都可以方便地运行 MySQL 程序，就应该把包含着那些程序的 MySQL bin 子目录添加到你的命令解释器的搜索路径里去。这很容易做到，按照本书附录 A 里给出的步骤把该子目录的路径名添加到你的 PATH 环境变量设置里去就可以了。

1.4.2　最低配置要求

要想试用本节中的示例，必须满足以下几项基本要求：

- 你的系统已经安装了MySQL软件；
- 你有一个用来连接数据库服务器的MySQL账户；
- 你有一个示例数据库。

MySQL 客户程序和 MySQL 服务器是必不可少的。MySQL 客户程序必须安装在你本人操作的机器里；服务器可以安装在你的机器里，也可以安装在别的主机里——只要你有对它的连接权限，MySQL 服务器可以放在任何地方。获得和安装 MySQL 的步骤请参考附录 A。如果你的网络连接需要经过一家 ISP（Internet Service Provider，因特网服务提供商），请提前查明该 ISP 提供的服务项目里有没有 MySQL。如果没有这个服务项目，并且该 ISP 也不准备安装它，请更换你的 ISP，选择提供 MySQL 的供应商。

除 MySQL 软件外，你还必须有一个 MySQL 账户。否则，你将无法连接 MySQL 服务器，也就无法创建你的示例数据库和其中的数据表。（如果你已经有了一个 MySQL 账户，可以直接拿过来用。但我建议你还是另外申请一个专供学习本书时使用的账户比较好。）

现在，我们遇到了一个“先有鸡，还是先有蛋”的问题：要想申请一个用来连接 MySQL 服务器的账户，你必须先连接到这个服务器上才行。一般来说，这需要在运行着 MySQL 服务器的主机上以 MySQL 的 root 用户身份登录，再用 CREATE USER 和 GRANT 语句新创建一个 MySQL 账户，并赋予数据库权限。如果 MySQL 服务器就安装在你自己的机器上并且正运转着，你本人就能以 root 身份连接上服务器并为自己新创建一个账户。在下面的示例里，我们给示例数据库 sampdb 增加了一个新

的管理员账户，新账户的用户名是 sampadm、口令是 secret。（你可以把它们改成别的，但要改就得把此处和本书其他有关内容里的用户名和口令都改过来。）

```
% mysql -p -u root
Enter password: ******
mysql> CREATE USER 'sampadm'@'localhost' IDENTIFIED BY 'secret';
Query OK, 0 rows affected (0.04 sec)
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'localhost';
Query OK, 0 rows affected (0.01 sec)
```

mysql 命令的-p 选项将会让 mysql 提示 MySQL 的 root 用户输入口令。你输入的口令将被显示为一串星号字符，即示例中的******。这里假设你已经为 MySQL 的 root 用户设置了口令。如果你还没有给它设置口令，请在提示 Enter Password:出现后直接按下 Enter 键。不过，root 用户没有口令是很大的安防漏洞，应该尽快给它设置一个。第 12 章讲述了 CREATE USER 和 GRANT 语句的其他信息，MySQL 用户账户的设置和口令的修改。

如果你打算今后就在运行着 MySQL 服务器的这台机器上连接 MySQL，就可以直接套用示例中的语句。这样你就能以用户名 sampadm 和口令 secret 连接上服务器，还能拥有 sampdb 数据库上的全部访问权限。注意：GRANT 语句并不能创建出数据库来（你可以在数据库创建之前赋予权限），我们稍后再来讨论数据库的创建问题。

如果你打算今后使用另一台计算机通过网络来连接 MySQL 服务器，就需要把示例中的 localhost 改为那台计算机的名字。举个例子，如果你将从主机 asp.snake.net 来连接 MySQL 服务器，就得把语句改写为下面这样：

```
mysql> CREATE USER 'sampadm'@'asp.snake.net' IDENTIFIED BY 'secret';
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'asp.snake.net';
```

如果你无法亲自操作服务器，也不能创建用户，那就得求助于 MySQL 管理员，让他为你建立一个新账户了。如果是这样，你就得把在本书各示例中出现的 sampadm、secret、sampdb 分别替换为管理员分配给你的用户名、口令和示例数据库名。

1.4.3 如何建立和断开与服务器的连接

通过 Unix 系统的 shell 提示符或者通过 Windows 下的 DOS 控制台用命令提示符调用 mysql 程序就能连接上 MySQL 服务器。这个命令如下所示：

```
% mysql  options
```

本书使用%作为命令提示符。事实上，%是 Unix 系统的一个标准提示符，另一个是$。在 Windows 下，有 C:>这样的提示符。（输入示例中的命令时，不用再输入提示符。）

这个 mysql 命令行里的 options 部分表示允许是空白。但下面这种形式的命令可能更多见一些：

```
% mysql -h host_name -p -u user_name
```

在执行 mysql 程序时，用不着把全部选项都写出来，但通常至少需要用户给出自己的用户名和口令来。下面是这几个选项的含义和用法。

- -h *host_name*（替换形式：-- host = *host_name*）
  待连接的服务器主机名。如果主机就是运行 mysql 客户程序的那台机器，此选项就可以省略。
- -u *user_name*（替换形式：-- user = *user_name*）

你的 MySQL 用户名。在 Unix 系统上，如果你的 MySQL 用户名与你的登录名完全一样，就可以省略这个选项——mysql 将自动把你的登录名用做你的 MySQL 用户名。

在 Windows 系统上，默认用户名是 ODBC。但这个默认名也许并不归你拥有。你可以通过命令行上的-u 选项明确地给出用户名，也可以通过环境变量 USER 来隐含地给出用户名。比如说，你可以用下面这条 set 命令设定一个名为 sampadm 的用户：

```
C:\> set USER=sampadm
```

如果已经通过 Control Panel（控制面板）中的 System（系统）页面设置了 USER 环境变量，该设置将影响到每一个控制台窗口，你就用不着再从命令提示符发出这个命令了。

- -p（替换形式：--password）

  这个选项的作用是通过 Enter password:让 mysql 提示你输入 MySQL 口令。比如说：

```
% mysql -h host_name -p -u user_name
Enter password:
```

  当看到提示 Enter password:时，请输入你的口令。（你输入的口令将不会显示在屏幕上，以免被你身后的人偷看到。）注意：MySQL 口令并不一定要与 Unix 或 Windows 口令相同。

  如果你省略了-p 选项，mysql 就将认为你不需要口令，也就不会提示你输入它了。

  这个选项的另一种形式是在命令行上直接给出口令，以-*pyour_pass*（替换形式：--password =*your_pass*，其中的 *your_pass* 就是你的口令）的形式直接敲入口令。但出于安全方面的考虑，最好别这样做，因为你身后的人会看到它。

  如果你确实想在命令行上直接敲入口令，有一点请特别注意：在 -p 和口令之间不允许有空格存在。-p 选项的这一特点（不需要输入空格）很容易与输入-h 和-u 选项时的情况弄混，无论选项与口令之间是否有空格，-h 和-u 都与其后的口令关联。

假设 MySQL 用户名和口令分别是 sampadm 和 secret，那么，如果 MySQL 服务器就运行在同一台主机上，就可以省略-h 选项和 mysql 命令而像下面这样去连接服务器：

```
% mysql -p -u sampadm
Enter password: ******
```

输入完这条命令后，mysql 将显示 Enter password:以提示你应输入口令。此时敲入口令（输入的 secret 将在屏幕上显示为 6 个星号字符******）。

如果一切正常，mysql 将显示欢迎消息和一个 mysql>提示以表明它在等你发出 SQL 查询命令。下面是完整的操作过程：

```
% mysql -p -u sampadm
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13762
Server version: 5.0.60-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

如果 MySQL 服务器运行在另一台机器上，就必须通过-h 选项来指定那台机器的主机名。假设那台主机的名字是 cobra.snake.net，就必须使用下面这样的命令：

```
% mysql -h cobra.snake.net -p -u sampadm
```

为简洁起见，我将在后面内容里出现的 mysql 命令行上省略-h、-p 和-u 选项，但大家在做练习或实际工作中请不要忘了输入它们。在运行其他 MySQL 程序（如 mysqlshow）时，你将用到这些选项。

在连接上 MySQL 服务器之后，我们随时都能通过 quit 命令来结束这次会话。如下所示：

```
mysql> quit
Bye
```

你也可以通过敲入字符串 exit 或\q 来退出。在 Unix 系统下，利用组合键 Ctrl-D 也可以退出。

在刚开始学习 MySQL 时，很多人都觉得它的安防系统给自己添了不少麻烦——你必须有足够的权限才能创建和访问数据库，你必须正确地给出用户名和口令才能连接上服务器。但在抛开教科书里的示例数据库而开始使用自己的数据记录之后，这些人的看法就会迅速改变，转而感激 MySQL 有这样一个能防止别人搜索或者（更糟糕）破坏自己信息的安防系统。

有些方法能让你把工作环境提前设置好，使你不必在每次运行 mysql 的时候都不得不在命令行输入一大堆的连接参数，1.5 节将会讨论这个问题。简化服务器连接过程最常见的办法是把连接参数存放到一个选项文件里。如果你想现在就去建立一个这样的文件，不妨直接跳到 1.5 节。

1.4.4 执行 SQL 语句

连接上服务器以后，你就可以发出查询命令让服务器执行了。本节将介绍一些与 mysql 交互的一般原则。

利用 mysql 来进行数据库查询很简单：先敲入有关命令，再在命令的末尾敲入一个分号字符（;）表示语句结束，再按下 Enter 键就行了。在你完成查询命令的输入之后，查询命令将由 mysql 送往服务器执行，服务器对查询进行处理并把其结果回送给 mysql，最后由 mysql 把查询结果显示在屏幕上。

下面这个简单的查询命令将让你看到系统的当前日期和时间：

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:51:23 |
+---------------------+
1 row in set (0.00 sec)
```

除使用分号外，终止语句的另一种方法是使用\g（表示 go）：

```
mysql> SELECT NOW()\g
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:51:28 |
+---------------------+
1 row in set (0.00 sec)
```

也可以使用\G，竖直排列显示结果，每行一个值：

```
mysql> SELECT NOW(), USER(), VERSION()\G
*************************** 1. row ***************************
    NOW(): 2008-03-21 10:51:34
   USER(): sampadm@localhost
VERSION(): 5.0.60-log
1 row in set (0.03 sec)
```

如果查询命令的输出行比较短，以\G 作为查询命令结束符的效果还不太明显。可万一输出行比较长，在屏幕上显示为好几行的时候，\G 结束符就能使屏幕输出内容更容易阅读一些。

如上所示，mysql 把查询结果、构成本次查询结果的数据行的个数以及用来处理本次查询所花费的时间依次显示出来。为简洁起见，我将在本书后面的示例里省略用来给出查询结果数据行总数的那一行。

因为 mysql 必须等待语句结束符，所以我们用不着把查询命令完整地写在同一个命令行上，我们可以用多个命令行来输入一条查询命令，如下所示：

```
mysql> SELECT NOW(),
    -> USER(),
    -> VERSION()
    -> ;
+---------------------+--------------------+------------+
| NOW()               | USER()             | VERSION()  |
+---------------------+--------------------+------------+
| 2008-03-21 10:51:37 | sampadm@localhost | 5.0.60-log |
+---------------------+--------------------+------------+
```

请注意，当我们输入查询命令的第一行时，提示符将从 mysql>变为->。这是在提醒你 mysql 认为你仍要继续输入查询命令。这种反馈非常重要，如果你遗漏了应该添加在查询命令末尾的分号，发生了变化的提示符将提醒你注意 mysql 仍在等待你继续输入。不然，就可能你这边在为 MySQL 经过了这么长的时间还没有完成你的查询而疑惑而烦躁，mysql 那边却在耐心地等待你把这条查询命令输入完！（mysql 还有几个别的提示符，我们将在附录 F 里介绍它们。）

如果你已经输入了好几行查询命令却不想执行它，可以敲入\c 来清除（即取消）它，如下所示：

```
mysql> SELECT NOW(),
    -> VERSION(),
    -> \c
mysql>
```

请注意，提示符将变回为 mysql>以表明 mysql 程序准备接收下一条查询命令。

与将一条语句输入成多行相反的是，在一行上输入多条语句，用终止符分隔。

```
mysql> SELECT NOW();SELECT USER();SELECT VERSION();
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:52:31 |
+---------------------+
+--------------------+
| USER()             |
+--------------------+
| sampadm@localhost |
+--------------------+
+------------+
| VERSION()  |
+------------+
| 5.0.60-log |
+------------+
```

在大多数情况下，查询命令允许以大写字母、小写字母或者大小写字母混用的形式来输入。比如

说，下面这几条查询命令就是等效的（虽然大小写不同）：

```
SELECT USER();
select user();
SeLeCt UsEr();
```

在后面的示例语句里，我们将用大写字母来写出 SQL 关键字和函数名，用小写字母来写出数据库、数据表和数据列的名字。

如果你想在语句里使用函数，请千万记住这样一件事：在函数名与它后面的括号中间不允许出现空格。有时空格会导致语法错误。

你可以把查询命令提前保存在一个文件里，再创建一个 SQL 脚本让 mysql 从那个文件而不是键盘来读取语句。这要用到 shell（操作系统的命令解释器）的输入重定向功能。比如说，如果把语句提前保存在一个名为 myfile.sql 的文件里，就可以像下面这样来执行它们（要指定任何必需的连接参数选项）：

```
% mysql < myscript.sql
```

这个文件的名字可以随便起。我喜欢给它们加上一个“.sql”后缀以表明这个文件里存放的是 SQL 语句。

这种利用 shell 的重定向功能来调用 mysql 的做法将在 1.4.7 节出现，我们将用这种办法把数据录入到 sampdb 数据库里。与一条一条地手工敲入一大堆 INSERT 语句相比，让 mysql 从某个文件里来读取它们要方便和快捷得多。

1.4 节的后续内容里有很多为大家练习而准备的 SQL 语句，它们以 mysql>提示符为标志，语句在提示符后面，且基本上都包括查询结果。如果你输入的语句与示例中的一模一样，那它们的查询结果也应该完全相同。但我在某些查询命令的前面没有给出提示符，它们主要用来阐明某个概念，不要求读者真的去执行它们。（当然，如果你愿意，试试它们也没什么坏处。但如果你打算用 mysql 来这样做，请不要忘记在它们的末尾加上一个分号作为结束符。）

1.4.5 创建数据库

我们的学习将从创建 sampdb 示例数据库与其中的数据表、把有关数据录入各数据表、利用这些数据表里的数据完成一些简单的查询任务开始逐步展开。使用数据库有以下几个步骤。

(1) 创建（初始化）一个数据库。

(2) 在数据库里创建各种数据表。

(3) 对数据表里的数据进行插入、检索、修改、删除等操作。

数据库上最常见的操作是对现有数据进行检索，比较常见的操作是插入新数据、修改或者删除现有数据，比较不常见的操作是创建数据表，最不常见的操作是创建数据库。但因为我们的学习要从一穷二白的状况开始，所以我们反而要从最不常见的数据库创建操作入手，再经过创建数据表和录入原始数据等步骤之后才能完成最常见的数据库操作——数据的检索。

创建新数据库的做法是：先用 mysql 连接上服务器，再用一条 CREATE DATABASE 语句给出新数据库的名字：

```
mysql> CREATE DATABASE sampdb;
```

只有在创建了 sampdb 数据库之后，才能创建该数据库里的各个数据表并对这些数据表的内容进

行各种操作。

那么，创建一个数据库是否就意味着把它选定为当前的默认数据库呢？答案是“否”。你可以用下面这条语句去核实一下：

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| NULL       |
+------------+
```

Null 意味着没有选择数据库。如果想把 sampdb 设置为当前的默认数据库，就需要发出一条 USE 语句：

```
mysql> USE sampdb;
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| sampdb     |
+------------+
```

选定默认数据库的另一种办法是在启动 mysql 的命令行上给出该数据库的名字：

```
% mysql sampdb
```

事实上，今后大家在选定数据库时用得最多的可能还是这后一种办法。如果需要指定服务器连接参数，请在命令行指定。比如说，下面两条命令将把 sampadm 用户连接到本地主机（如果没有指定主机名字，就是默认的）的 sampdb 数据库上去：

```
% mysql -p -u sampadm sampdb
```

如果需要连接到在远程主机上运行的 MySQL 服务器，就应在命令行指定主机：

```
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果没有特别说明，以后的示例都将假定你在启动 mysql 时已经在命令行上把 sampdb 数据库选定为当前的默认数据库。如果你在启动 mysql 时忘了在命令行选定这个数据库，请在 mysql>提示符处发出一条 USE sampdb 语句。

1.4.6　创建数据表

在本节里，我们将创建示例数据库 sampdb 里的各个数据表。首先创建“美国历史研究会”场景所需要的各种数据表，然后再创建“考试记分项目”所需要的各种数据表。在这个部分，有些数据库图书会大谈特谈*数据库的分析与设计*、*实体联系图*、*规范化过程*等概念。有些书专门讲解这些概念，而本书只讨论我们的数据库应该是什么样子——它应该包含哪些数据表，各数据表应该有什么样的内容，以及数据应该如何表示。

这里所选定的数据表示方式并不是绝对的，换在另一种场合，你可能会选用另一种方式来表示类似的数据——这应该由应用项目的具体要求和数据的具体用途来决定。

1. 美国历史研究会的数据表

美国历史研究会场景需要的数据表相当简单，如下所示。

- ❑ president（总统）数据表。用来保存关于美国历届总统的描述性记录。我们将利用它在研究会的网站上提供历史知识在线小测验（会刊儿童栏目中的打印的小测验的交互式模拟版）。
- ❑ member（会员）数据表。用来保存每一位会员的个人最新资料。研究会将利用这个数据表来完成制作会员名录（纸印刷品以及网上版本）、自动提醒会员续交会费等工作。

● president数据表

president 数据表比较简单，所以我们先来讨论它。这个表里包含着关于美国历届总统生平的基本信息。

- ❑ **姓名**。在数据表里，有好几种办法可以用来表示人的姓名。比如说，我们既可以把姓名保存在同一个数据列里，也可以把人的姓氏和名字分别保存在不同的数据列里。用同一个数据列来保存姓名当然要简单一些，但这种做法有一定的局限性，如下所示。
  - ■ 如果先输入名字，就无法按姓氏进行排序。
  - ■ 如果先输入姓氏，就无法按名在前姓在后的（英语国家）习惯顺序来显示它们。
  - ■ 很难对姓名进行查找。比如说，如果你想查找某个姓氏，就必须使用一个匹配模板（pattern）来查找与之匹配的姓名。与直接查找姓氏的做法相比，这种做法效率低且速度慢。

  为了避开这些限制，president 数据表将把总统们的姓氏和名字分别保存在不同的数据列里。

  我们把总统们的中间名或第一个名字也安排在名字数据列里。因为我们不太可能对中间名（也不太可能对第一个名字）排序，所以这应该不会影响到我们将对总统姓名进行的排序。这也不会影响到姓名的显示，因为无论是按“Bush, George W.”还是按“George W. Bush”的格式来显示，姓名里的中间名总是紧跟在名字的后面。

  还有一个问题需要考虑。有位总统（如 Jimmy Carter）的姓名后面还有一个“Jr.”。应该把这个东西放到哪儿呢？根据英语习惯，这位总统的名字既可以写成“James E. Carter, Jr.”，也可以写成“Carter, James E., Jr.”。这个“Jr.”只能出现在整个姓名的末尾，无法与名字或姓氏结合在一起。因此，我们决定另建一个数据列来保存这种姓名后缀。这种情况请大家务必注意：即使只有一个特例值，也会影响到你在选择数据表示形式时的决策。它同时还证明了这样一条经验：应该在事先对将被保存到数据库里去的数据值做尽可能深入的了解。如果你没有在事先对这些问题做周密的考虑，就可能会在启用数据库之后还不得不再去修改数据库结构。这种事情虽说不是什么灾难，但还是从一开始就尽量避免为好。
- ❑ **出生地（州和城市）**。与姓名的情况类似，这些信息也是既可以保存在同一个数据列，也可以保存在多个数据列。保存在同一个数据列的做法要简单些，但把它们分别保存在不同的数据列将使你的某些工作更容易完成。比如说，如果把州名与城市名分开放置，诸如“出生在某个州的总统有多少”之类的问题就更容易查询出来。
- ❑ **出生日期和逝世日期**。这里需要考虑的特殊情况是：不能要求必须填上逝世日期，因为有些总统还依然健在呢。MySQL有一个专用的特殊值NULL来对付这种“无数据”的情况，所以我们将在逝世日期列里用这个值来表示“依然健在”的情况。

● member数据表

从每条记录都保存着某个人的个人资料的角度看，用来存放美国历史研究会会员个人资料的 member 数据表与刚才介绍的 president 数据表差不多。但每个 member 数据表里还包含其他一些数据列，如下所示。

- **姓名**。我们将沿用president数据表的3个数据列（姓氏、名字和姓名后缀）表示法。
- **ID编号**。每个会员都会在其会员资格初次生效时分配到一个独一无二的编号。研究会以前从没对会员进行过编号，但既然打算从现在起对会员进行更系统化的管理，所以眼下正是一个好时机。（我希望大家会不断发现MySQL的好处并琢磨出会员资料更多的用途。当你需要把member数据表和其他与会员有关的数据表联系起来时，会员编号用起来比姓名要简便得多）。
- **失效日期**。会员必须定期续费才能保证其会员资格不会过期失效。在别的项目里，你可能需要把这个日期表示为“上次交费日期”，但这对美国历史研究会的情况不适用。会员资格的有效期是一个可变的数字（可以是一年、两年、三年或者五年），而“上次交费日期”并不能告诉你某个会员必须在何时交纳下一期会费。所以我们将保存会员资格的失效日期。此外，研究会还有一个终身会员制度。虽然可以用一个遥远的未来日期来代表这种情况，但特殊的NULL值更理想，因为用“无数据”来代表“永不失效”是非常合乎逻辑的。
- **电子邮件地址**。电子邮件地址将使兴趣相同的会员更容易交流。对于身为研究会秘书的你来说，这些地址将使你能够以电子方式向会员发出续费通知而不必再依靠普通信件。与跑到邮局去寄信相比，这种做法既方便又省钱。你还可以利用电子邮件把会员们的当前个人资料发送给他们做必要的修改。
- **邮政地址**。这是为那些无法通过电子邮件进行联络（或者没有回复你电子邮件）的会员而准备的。我们将把街道地址、城市名、州名和邮政编码分别保存在不同的数据列里。
  我们这里要假设全体会员都居住在美国。当然了，对于那些在世界各地都有会员的组织机构来说，这个假设过于简单了。如果涉及多个国家的地址，你就必须研究各种地址格式。比如说，邮政编码并没有一项国际标准，还有些国家被划分为省而不是州。
- **电话号码**。这些信息与地址列的作用相似，都是为了与会员联络。
- **会员兴趣关键字**。研究会的每位会员都对美国历史感兴趣，但他们的兴趣却可能集中在某些特定的历史时期上。这个数据列就是用来记录这种特殊兴趣的。会员可以利用这些信息来寻找与自己兴趣相同的其他会员。（严格说来，建立一个单独的表会更好，其中的行由一个关键字和相关成员ID组成。即便这也有弊端，我不想在这儿谈。）

● *美国历史研究会各数据表的创建*

下面，我们将开始创建“美国历史研究会”的各种数据表。我们要用 CREATE TABLE 语句来完成这一工作，这条语句的格式是：

```
CREATE TABLE tbl_name (column_specs);
```

其中，*tbl_name* 是给数据表起的名字，*column_specs* 则是该数据表里的各个数据列以及各种索引（如果有的话）的定义。索引能够加快信息的检索速度；我们将在第 5 章中介绍它们。

下面是对应于 president 数据表的 CREATE TABLE 语句：

```
CREATE TABLE president
(
  last_name  VARCHAR(15) NOT NULL,
  first_name VARCHAR(15) NOT NULL,
  suffix     VARCHAR(5) NULL,
  city       VARCHAR(20) NOT NULL,
  state      VARCHAR(2) NOT NULL,
  birth      DATE NOT NULL,
```

```
  death      DATE NULL
);
```

执行这条语句的方法有好几种。可以手动输入，也可以使用 sampdb 发行版的 create_president.sql 文件中预先写好的语句。

如果你打算亲自输入这条语句，请先用下面的命令来启动 mysql 客户程序，并把数据库 sampdb 设置为当前的默认数据库：

```
% mysql sampdb
```

然后再敲入上面的 CREATE TABLE 语句。不要漏掉语句末尾的分号，这样才能让 mysql 程序知道这条语句的结束位置。有缩进没有关系，你不需要在同一处换行。例如，你可以在同一行输入一条语句。

如果打算利用一个预先写好的描述文件来创建 president 数据表，可以使用 sampdb 发行版本里的 create_president.sql 文件。你可以在系统安装这个发行版本时所创建的 sampdb 子目录里找到这个文件。先切换到那个子目录，然后再执行下面这条命令：

```
% mysql sampdb < create_president.sql
```

不管如何启动 mysql 客户程序，都不要忘记在命令行里的命令名称之后加上必要的连接参数（主机名、用户名、口令等）。

CREATE TABLE 语句中的数据列定义由以下几部分组成：数据列的名字、数据类型（这个数据列是用来保存哪种数据的）和一些属性。

president 数据表用到了两种数据类型：VARCHAR(*n*)和 DATE。VARCHAR(*n*)的意思是这个数据列里存放着长度可变的字符（串）值，最多有 *n* 个字符。这个 *n* 要由你根据对字符串数据长度的预估来选定。比如说，我们把 state 数据列定义为 VARCHAR(2)类型，这是根据美国州名都可以被缩写为两个字母的事实而确定的。其他字符串类型的数据列所要容纳的值可能会比较长，所以它们要宽一些。

我们用到的另一种列类型是 DATE。这种类型的数据列用来保存日期值，这用不着多说，但大家千万要注意日期值的表示格式。MySQL 要求日期被表示为'*CCYY-MM-DD*'的格式，其中 *CC*、*YY*、*MM*、*DD* 分别代表世纪、年份、月份和日期。这是 SQL 中规定的日期表示标准（也叫做 ISO 8601 格式）。比如说，2002 年 7 月 18 日在 MySQL 里必须被表示为'2002-07-18'而不是'07-18-2002'或'18-07-2002'。

president 数据表还用到了两种数据列属性：NULL（表示无数据）和 NOT NULL（表示不得为空）。表中的大部分数据列都具有 NOT NULL 属性，因为我们必须在其中填上数据。具有 NULL 属性的数据列有两个，一个是 suffix（姓名后缀，大多数总统的姓名里都没有后缀），另一个是 death（逝世日期，有些总统还活着，用不着填逝世日期）。

下面是我们用来创建 member 数据表的 CREATE TABLE 语句：

```
CREATE TABLE member
(
  member_id  INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (member_id),
  last_name  VARCHAR(20) NOT NULL,
  first_name VARCHAR(20) NOT NULL,
  suffix     VARCHAR(5) NULL,
  expiration DATE NULL,
  email      VARCHAR(100) NULL,
  street     VARCHAR(50) NULL,
```

```
  city       VARCHAR(50) NULL,
  state      VARCHAR(2) NULL,
  zip        VARCHAR(10) NULL,
  phone      VARCHAR(20) NULL,
  interests  VARCHAR(255) NULL
);
```

你可以手动输入这些语句，也可以利用 sampdb 发行版本里的预编写文件 creat_member.sql，其中包含用于 member 表的 CREATE TABLE 语句。执行下面的命令：

```
% mysql sampdb < create_member.sql
```

在 member 数据表里，大部分数据列的类型都是可变长度的字符串。只有两个数据列例外：用来保存会员编号的 member_id 和用来保存失效日期的 expiration。

为了避免混淆不同的会员，数据列 member_id 里的值必须是独一无二的。这正是 AUTO_INCREMENT 数据列大显身手的地方——当我们往 member 数据表添加新记录时，MySQL 能在 member_id 列自动生成一个唯一的会员编号。虽然我们将要放到 member_id 数据列里的只是些数字，但它的定义却包含了好几个部分，如下所示。

- INT。表示这个数据列将用来保存整数值（没有小数部分的数字）。
- UNSIGNED。不允许出现负数。
- NOT NULL。必须填有数据，不得为空。（这意味着每个会员必须有一个会员号。）
- AUTO_INCREMENT。这是MySQL里的一个特殊属性。它表示数据列里存放的是序列编号。AUTO_INCREMENT机制的工作原理是这样的：当我们往member数据表里插入数据记录时，如果没有给出member_id列的值（或者给出的值是NULL），MySQL将自动生成下一个编号并赋值给这个数据列。这样，为新会员分配会员号的工作就简单了，因为MySQL可以替我们完成。

PRIMARY KEY 子句表示需要对 member_id 数据列创建索引以加快查找速度，同时也要求该数据列里的各个值都必须是唯一的。后者正好满足了我们对会员 ID 的编号要求，因为我们绝不希望误把同一个会员号分配给两个会员。此外，MySQL 本身也要求每一个具备 AUTO_INCREMENT 属性的数据列必须拥有某种形式的唯一化索引，若没有，数据表的定义就不合法。任何 PRIMARY KEY 列都必须是 NOT NULL，所以，如果在 member_id 定义中忽略了 NOT NULL，MySQL 将自动添加上去。

如果你现在还不能理解 AUTO_INCREMENT 和 PRIMARY KEY 的含义与作用，不妨把它们想象成一种用来为生成带索引的会员号的魔术好了。我们真正关心的是那些会员号是否都是唯一的，它们到底等于多少并不重要。（有关 AUTO_INCREMENT 数据列的使用请参见第 3 章。）

expiration 列是一个 DATE。它可以为 NULL 值，所以其默认值为 NULL。NULL 意味着可以不输入数据。原因就是前面提到的，expiration 可以为 NULL，表明成员具有终身会员资格。

现在，既然已经让 MySQL 创建了几个数据表，我们就该去检查一下它做得怎么样。在 mysql 里，我们可以用下面这条命令来查看 president 表的结构：

```
mysql> DESCRIBE president;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| last_name  | varchar(15) | NO   |     |         |       |
| first_name | varchar(15) | NO   |     |         |       |
| suffix     | varchar(5)  | YES  |     | NULL    |       |
| city       | varchar(20) | NO   |     |         |       |
```

```
| state      | varchar(2)  | NO   |     |         |       |
| birth      | date        | NO   |     |         |       |
| death      | date        | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

如果你发出的是 DESCRIBE member 命令，mysql 就将显示有关 member 数据表的类似信息。

DESCRIBE 是一个非常有用的命令，尤其是当你想不起数据列的名字、类型或数据长度等细节的时候。你还可以利用这条命令来查看各数据列在数据行里的存储先后顺序，这个顺序很重要，INSERT 或 LOAD DATA 等语句要求各数据列的值必须按它们默认的存储顺序依次列出。

能够用 DESCRIBE 命令查出来的信息也可以通过别的手段获得。你可以把它简写为 DESC，也可以把它写成 EXPLAIN 或 SHOW 语句。下面这些语句的作用是相同的：

```
DESCRIBE president;
DESC president;
EXPLAIN president;
SHOW COLUMNS FROM president;
SHOW FIELDS FROM president;
```

这些语句还允许你把输出内容限制为指定的数据列。比如说，如果你在 SHOW 语句的末尾加上一个 LIKE 子句，就只能看到与给定模板相匹配的那几个数据列的有关信息，如下所示：

```
mysql> SHOW COLUMNS FROM president LIKE '%name';
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| last_name  | varchar(15) | NO   |     |         |       |
| first_name | varchar(15) | NO   |     |         |       |
+------------+-------------+------+-----+---------+-------+
```

DESCRIBE president '%name'是等效的。这里使用的百分号（%）是一个特殊的通配符，1.4.9 节的第 7 小节将介绍它。

SHOW 语句还有其他几种用法，可以用来从 MySQL 获取各种信息。SHOW TABLES 能够列出当前默认数据库里的数据表。例如，我们已经在 sampdb 数据库里创建了两个数据表，于是输出为：

```
mysql> SHOW TABLES;
+------------------+
| Tables_in_sampdb |
+------------------+
| member           |
| president        |
+------------------+
```

另外，SHOW DATABASES 能够列出当前连接的服务器上的数据库，如下所示：

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| sampdb             |
| test               |
+--------------------+
```

服务器不同，这条语句列出来的数据库清单也就不同，但你至少应该能看到 information_schema 和 sampdb.information_schema。sampdb 数据库是我们刚创建的。你还会看到 test 数据库，它是在 MySQL 安装过程中创建的。根据你的访问权限，你还会看到 mysql 数据库，其中存放着各种用来控制 MySQL 访问权限的权限分配表。

客户程序 mysqlshow 提供命令行接口，用 SHOW 语句能查看到的信息也都能用 mysqlshow 程序查看到。

不带参数的 mysqlshow 程序将列出一份数据库清单：

```
% mysqlshow
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| sampdb             |
| test               |
+--------------------+
```

如果给它加上一个数据库名，mysqlshow 将列出一份给定数据库里的数据表清单：

```
% mysqlshow sampdb
Database: sampdb
+-----------+
|  Tables   |
+-----------+
| member    |
| president |
+-----------+
```

如果同时给出一个数据库名和数据表名，mysqlshow 将显示那个数据表里各数据列的信息——就像 SHOW FULL COLUMNS 语句那样。

2. 考试记分项目的数据表

要想确定考试记分项目需要用到哪些数据表，先要弄清楚怎样用纸质记分簿来记录考生成绩。请看图 1-2，假设这是纸质记分簿里的某一页，上面是一个记有考试分数的表格，其中还包含其他一些使考试分数更有意义的信息。学生的姓名和 ID 号列在表格的左侧（为简洁起见，我只列出了 4 位学生），考试或测验的举行日期则列在表格的顶部。表格显示，9 月的 3、6、16、23 日进行了测验，9 月 9 日和 10 月 1 日进行了考试。

学生		分数						
ID	姓名	Q 9/3	Q 9/6	T 9/9	Q 9/16	Q 9/23	T 10/1	...
1	Billy	14	10	73	14	15	67	...
2	Missy	17	10	68	17	14	73	...
3	Johnny	15	10	78	12	17	82	...
4	Jenny	14	13	85	13	19	79	...
...	...	...	...	...	...	...	...	...

图 1-2　纸质记分簿里的某一页

要想把这些信息记录到一个数据库里，就需要一个 score 数据表。那么，这个数据表里的各条记

录应该包含哪些信息呢？这个问题不难回答。在每个数据行里，需要列出学生的姓名、考试或测验的日期和学生的考试分数。图 1-3 给出了数据表里的一些考试分数。（注意，日期是按 MySQL 的日期表示法'CCYY-MM-DD'格式写出来的。）

score 数据表

name	date	score
Billy	2008-09-23	15
Missy	2008-09-23	14
Johnny	2008-09-23	17
Jenny	2008-09-23	19
Billy	2008-10-01	67
Missy	2008-10-01	73
Johnny	2008-10-01	82
Jenny	2008-10-01	79

图 1-3 最初的 score 数据表

可是，如此得到的数据表是有问题的，它丢失了某些信息。比如说，仔细看看图 1-3 中的行就会发现，我们无法区别考试分数与测验分数。一般说来，在评定学生们的期末总成绩时，考试分数与测验分数的比重是有一定区别的，所以有必要知道考分类型。当然了，我们可以根据某给定日期的分数范围（测验分数通常要比考试分数在数值上低很多）来推测出这个类型，但这种不用数据明确表明而纯粹依靠推理的做法会带来问题。

要想在每行记录里把考分类型区别开还是有办法的，例如，可以像图 1-4 那样给 score 数据表增加一个数据列，并用 T 或 Q 来分别代表 test（考试）或 quiz（测验）。这种做法的好处是考分类型能直接体现在数据上，坏处是这部分信息有些冗余。看看那些日期相同的行就能发现，考分分类（category）栏里的值全都一模一样。在 9 月 23 日，所有考分的类型全都是 Q；到了 10 月 1 日，所有考分的类型又全都是 T。这可有点太啰嗦了。要是按这种办法来记录学生们的考试分数，我们不光要反复多次地输入一个相同的日期，还不得不反复多次地输入一个相同的考分类型。有谁愿意反复输入这么多的冗余信息呢？

score 数据表

name	date	score	category
Billy	2008-09-23	15	Q
Missy	2008-09-23	14	Q
Johnny	2008-09-23	17	Q
Jenny	2008-09-23	19	Q
Billy	2008-10-01	67	T
Missy	2008-10-01	73	T
Johnny	2008-10-01	82	T
Jenny	2008-10-01	79	T

图 1-4 修改后的 score 数据表，增加了一个 type 列

我们应该想出一个更好的办法。与其把考分类型放到 score 数据表里，不如把它与考试日期对应起来。可以把考试日期列成一个表，再把各日期里发生的“考试事件”（测验或考试）记录在这个表里。这样，只要根据 score 表里的日期在 grade_event 表里查出当天的考试事件类型，我们就能知道某个分数是来自测验还是来自考试。图 1-5 给出了这种思路下的数据表布局，并以 2008 年 9 月 23 日为例画出了 score 表与 grade_event 表的对应关系。根据 score 数据表里的日期，我们从 grade_event 数据表里查出当天举行的是一次测验，所以 score 表里的那个分数是一次测验成绩。

score 数据表

name	date	score
Billy	2008-09-23	15
Missy	2008-09-23	14
Johnny	2008-09-23	17
Jenny	2008-09-23	19
Billy	2008-10-01	67
Missy	2008-10-01	73
Johnny	2008-10-01	82
Jenny	2008-10-01	79

grade_event 数据表

date	category
2008-09-03	Q
2008-09-06	Q
2008-09-09	T
2008-09-16	Q
2008-09-23	Q
2008-10-01	T

图 1-5　通过 date 列相联系的 score 与 grade_event 数据表

与通过推测来判断考分类型的做法相比，新办法要好得多了，因为现在能够从记录在数据库里的数据直接得出考试分数的类型。与把考分类型直接记录在 score 数据表里的做法相比，新办法也要好得多——我们总共只需记录一次考分类型，不必再为每个考分都要记录一次了。

不过，现在需要把多个数据表的信息结合起来才行。也许你和我一样，在第一次听说这种事的时候，可能会想：“嘿，这个主意可真够酷的。可这么多的数据表，想查什么东西会不会太费事？这会不会把事情搞得更复杂呢？”

从某种意义上讲，这种担心是有道理的。记录两个表当然要比记录一个表复杂。可仔细看看当初的记分簿（如图 1-2 所示），你不是已经在记录两套信息了吗？请注意以下两个事实。

- 把考试分数记录在表格的每一小格里，这些小格按学生姓名和考试日期排列（按姓名，由上往下排列；按日期，由左往右排列）。这正是我刚才所说的两套信息中的一套，与这套信息相对应的是 score 数据表里的内容。
- 你是怎么知道各日期所代表的事件类型的呢？你在记分簿里是这样做的：在日期的上面写上一个 T 或 Q，在表格的顶部把考试日期与考试类型关联起来。这正是我刚才所说的两套信息中的第二套，与这套信息相对应的是 grade_event 数据表里的内容。

换句话说，也许你本人还没有意识到这一点，但你在记分簿里做的事与我把信息放到两个数据表里的情况并没有什么差异。即便是有差异，也只是纸质记分簿里的两套信息没有明确地分别放置而已。

记分簿表格的例子体现出了人们对信息的思维方式，也反映出这样一个问题：把信息妥善地放到数据库里去并不是一件简单的事情。在日常生活中，人们习惯于把不同信息综合起来并作为一个整体来考虑。但数据库毕竟不是人类的大脑，这正是它们看起来过于人工化和不太自然的原因之一。习惯于把信息综合在一起的思维特点使我们有时很难意识到自己正使用着多种的信息而非一种。因此，以数据库系统的方式来表达数据往往很有挑战性。

图 1-5 里的 grade_event 数据表还隐含了这样一个要求：date 列里的日期必须是独一无二的。因为每一个日期都将被用来联系 score 和 grade_event 数据表里的某些数据记录。换句话说，它要求你不得在同一天进行两场测验（或者一次测验加一次考试）。如果你这样做了，那么，score 数据表里将会有两组考分记录、grade_event 数据表里将会有两条类型记录，都对应于同一个日期。这意味着通过日期的匹配关系来联系 score 记录和 grade_event 记录的做法将难以为继。

假如你每天最多只进行一场考试，那么这个问题就不成其为问题。但一天两场考试的情况真的永远都不会发生的吗？也许如此，心地善良的你应该不会对学生们苛刻到要对他们进行一天两场考试的程度。不过（希望大家别怪我多嘴），虽然经常有人说“这种奇怪的事情永远也不会发生”，可奇怪的事情却真的在某个时刻发生了。为了弥补这一漏洞，这些人就不得不加班加点地去重新设计数据表。

与其临时抱佛脚，不如防患于未然。提前预见到可能出现的各种问题并准备好应对措施将为你减少很多麻烦。因此，还是现在就对“你会记录同一天里的两组考试分数”的情况作一下分析比较好。应该如何解决这个问题呢？别担心，随着讨论的深入，这个问题将迎刃而解。只要对有关数据的布局结构作一个小小的改动，在同一天发生多次考试事件的事情就不再会引起麻烦。这些改动如下所示。

(1) 在 grade_event 表里增加一个数据列，利用它给 grade_event 表里的各个记录分配一个唯一的编号。从效果上讲，这等于是给各次考试事件分别赋予了一个唯一的 ID 编号。我们就给新增的这个数据列起名为 event_id（意思是事件编号）好了。（虽然看着有点奇怪，可这一做法却并不是什么新点子，图 1-2 中的记分簿表格其实已经用到了这个东西。记分簿表格分数记录部分的列序号就相当于这里的事件编号。虽说你没有把各列的序号明确地写出来并标明是“event ID”，但它的的确确存在。）

(2) 在把考试分数记到 score 数据表里去的时候，用考试事件的 ID 来代替考试日期。

完成上述改动后，我们得到了图 1-6 所示的结果。现在，score 和 grade_event 表必须用 event_id 而不是 date 来联系。你用 grade_event 表不仅能查出考分的类型，还能查出它具体发生在哪一天。最重要的是，grade_event 表中必须具备唯一性的不再是日期，而是事件编号。这意味着即使在一天之内进行了十几场考试和测验，也能条理清晰地把各场的分数全都记录下来。（你的学生肯定害怕听到这个消息。）

score数据表

name	event_id	score
Billy	5	15
Missy	5	14
Johnny	5	17
Jenny	5	19
Billy	6	67
Missy	6	73
Johnny	6	82
Jenny	6	79

grade_event数据表

event_id	date	category
1	2008-09-03	Q
2	2008-09-06	Q
3	2008-09-09	T
4	2008-09-16	Q
5	2008-09-23	Q
6	2008-10-01	T

图 1-6 通过事件编号列相联系的 score 与 grade_event 数据表

应该承认，图 1-6 里的表格不如前面那几个看起来顺眼。score 表变得越来越抽象，数据列的含义也越来越不容易看懂。请看图 1-4 里的 score 表，那里面既有考试日期又有考分类型，让人一眼就能看明白。但在图 1-6 里，这两个数据列却不见了，我们看到的是一个高度抽象化的信息表示形式。谁会愿意看一个包含 event_id 的 score 表呢？它对我们而言没多大意义。

此时此刻，我们来到了一个十字路口。此前，大家对电子化的考试记分系统充满希望，觉得很快就能从繁琐的评分工作中解脱出来。但在看过上面的讨论后，却发现单是把信息放到数据库里去的事就已经很不容易做到最好了。高度抽象的信息与它们所代表的事物似乎毫无联系，这往往会让人们产生一种畏难情绪。

这很自然地引出了一个问题：“干脆不用数据库会不会更好？也许 MySQL 不适合我。”我的回答大家肯定都能猜到，要不这本书就不会有这么厚了。但对读者来说，在项目开工前多考虑几种办法总是好的。你们应该问自己：是使用 MySQL 这样的数据库系统好，还是使用电子表格（spreadsheet）等其他办法好？从一方面看，

- 记分簿由行和列构成，电子表格也是如此，它们二者在概念和外观上都很相似；
- 电子表格程序能够进行计算，所以使用计算字段进行分数统计工作不难完成。测验分数和考试分数可能不太容易按不同权重来统计，但肯定有办法解决。

从另一方面看，如果你想只对一部分数据进行操作（比如只统计测验分数，或者只统计考试分数），或者想进行某种对比分析（比如男生与女生的成绩对比），或者想灵活地汇总和显示各种统计信息，事情就不同了。这些工作电子表格都不擅长，关系数据库系统则能大显身手。

往开处想，关系数据库中数据的高度抽象化也不是什么不得了的事。你只是在数据库建立之初需要考虑信息在数据库里的表示方式，按照最符合你目标的方式来设置它们。在数据库建立起来以后，信息数据的提取和显示工作将由数据库引擎按一定的逻辑来完成，你看到的是有意义的资料，而不是抽象的彼此无关的信息零件。

比如说，从 score 数据表检索学生分数时，你想看的是考试日期而不是事件编号。这很容易办到：数据库将根据事件编号从 grade_event 表里查出考试日期来给你看。如果你还想知道考试分数是来自测验还是来自考试，这也很容易办到，数据库也能根据事件编号查出考分类型。别忘了，MySQL 之类的关系数据库系统最擅长的本领是——把一样东西与另一样东西联系起来，从多个信息源把你最想知道的信息查找出来。在考试记分的例子里，MySQL 必须通过事件编号才能对信息进行关联和提取，但你（数据库的使用者）并不需要关心这类细节。

为了让大家提前了解如何让 MySQL 将各信息联系起来，我们准备了一个例子，假设你打算查看 2002 年 9 月 23 日的考试分数。下面这个查询将那天的考试分数查出来：

```
SELECT score.name, grade_event.date, score.score, grade_event.category
FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id
WHERE grade_event.date = '2008-09-23';
```

有点吓人吧？这个查询将把表 score 和表 event 结合（联系）起来并检索出学生姓名、考试日期、考试分数和考分类型等信息，下面是它的输出结果：

```
+--------+------------+-------+----------+
| name   | date       | score | category |
+--------+----------- +-------+----------+
| Billy  | 2008-09-23 |    15 | Q        |
| Missy  | 2008-09-23 |    14 | Q        |
| Johnny | 2008-09-23 |    17 | Q        |
| Jenny  | 2008-09-23 |    19 | Q        |
+--------+------------+-------+----------+
```

是不是觉得上面这个表格有点面熟？没错，它与图 1-4 里的表格格式是一模一样的。你不必知道事件编号就能得到这份查询结果。你指定了一个日期，MySQL 把该日期里的考试分数找了出来。总之，虽然数据库里的信息很抽象，与它们所代表的事物似乎也没有直接的联系，但这并不会影响它们的使用，数据库会根据你的查询把信息提取出来并显示为有意义的资料。

再仔细看看这个查询，大家也许又会产生一些新的问题。这个查询看起来太长太复杂。仅为查出某天的考试分数就要写这么多东西是不是太复杂了？是的。但是，有好几种办法能避免在输入查询命令的时候敲很多行内容。比较常见的做法是：一旦确定了某个查询的最终写法，就把它保存起来，以后在必要时就可以直接使用了。我们将在 1.5 节讨论这如何实现。

要不是为了让大家对查询过程有个了解，我是不想这么早就给出例子的。事实上，与我们真正用来检索考试分数的查询相比，刚才举的例子还算是简单的。因为我们还需要对数据表的布局再做一次较大的改动。首先，让 score 表不再包含学生姓名，我们将使用一个独一无二的学生 ID 编号。这其实就等于是用纸质记分簿里的 ID 栏而不是 Name 栏来构成 score 表。我们再新建一个名为 student

的数据表来存放学生姓名（name）和学号（student_id），如图 1-7 所示。

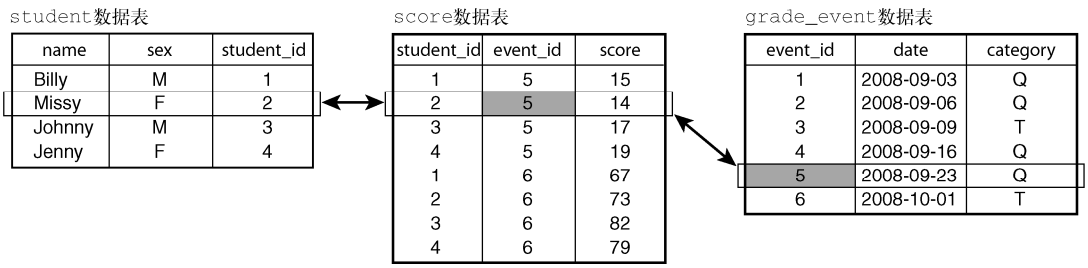


图 1-7　通过学生学号和事件编号相联系的 score、student 和 grade_event 数据表

为什么要做这样的改动呢？为了应对出现两名学生名字相同的情况，唯一的学生 ID 将有助于把他们区分开来。（这与我们不使用日期而使用唯一事件编号来区分在同一天进行的考试和测试的分数的道理是一样的。）在对数据表的布局做了上述改动之后，用来查询给定日期的考试分数的命令又变得稍微复杂了一些，如下所示：

```
SELECT student.name, grade_event.date, score.score, grade_event.category
FROM grade_event INNER JOIN score INNER JOIN student
ON grade_event.event_id = score.event_id
AND score.student_id = student.student_id
WHERE grade_event.date = '2008-09-23';
```

如果你现在还看不懂这个查询命令，请不要着急。大多数初学者都是如此。在 1.4 节的后半部分内容里，我们还会遇到这个查询命令，等到那时你就能看明白它了。真的，不开玩笑。

大家可能已经注意到我在图 1-7 里的 student 表里增加了一些记分簿里没有的东西，它多了一个 sex（性别）列。你可以利用这个数据列来统计班级里男生或女生的人数，也可以利用它来对男女生的成绩进行比较分析。

考试记分项目的数据表到这里就设计得差不多了。我们只需再增加一个用来记录缺勤情况的数据表就全部完成了。这个数据表的内容很简单：一个学生 ID 和一个日期（如图 1-8 所示）。这个数据表里的每一个数据行都代表当天缺勤的一位学生。等到学期结束的时候，我们将通过 MySQL 的统计功能来汇总这个表里的数据，把每位学生的缺勤次数查出来。

absence 数据表

student_id	date
2	2008-09-02
4	2008-09-15
2	2008-09-20

图 1-8　absence 数据表

- student数据表

好了，考试记分项目的数据表到这里就全部设计完成了，下一步就该创建它们了。下面是我们用来创建 student 数据表的 CREATE TABLE 语句：

```
CREATE TABLE student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = InnoDB;
```

注意观察，CREATE TABLE 语句中加入了一些新内容（结尾的 ENGINE 子句），稍后将解释它的

用途。

你可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_student.sql
```

上面这条 CREATE TABLE 语句将创建出一个名为的 student 且包含有 name、sex、student_id 等 3 个数据列的数据表。

name 是一个可变长度的字符串数据列，它最多可以容纳 20 个字符。这种人名表示法要比美国历史研究会场景中的数据表里使用多个数据列来分别保存人的姓氏和名字的情况来得简单，它只用了一个数据列。我之所以这样做是因为我知道考试记分项目不会出现必须用多个数据列来表示人名的查询操作。（因为这本书是我写的。但在实际中你可能需要使用多个数据列。）

sex 用来表明某位学生是男生还是女生。这是一个 ENUM（枚举）类型的数据列，其中的可取值只能是在该数据列的定义里枚举出来的那些值中的某一个：'F'代表女生，'M'代表男生。如果你想把某个数据列的可取值限制在一个元素个数有限的集合内，ENUM 就正好管用。当然了，我们也可以把这个数据列定义为 CHAR(1)，但 ENUM 能够更明确地把这个数据列只有有限个可取值的特点表示出来。如果你忘了它都有哪些可取值，可以发出一条 DESCRIBE 命令来查看。对于 ENUM 数据列，MySQL 将把它合法的枚举值都列出来，如下所示：

```
mysql> DESCRIBE student 'sex';
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| sex   | enum('F','M') | NO   |     |         |       |
+-------+---------------+------+-----+---------+-------+
```

ENUM 数据列的值不一定非得是一个字符。我们完全可以把 sex 数据列定义为：ENUM ('female', 'male')。

student_id 是一个整数类型的数据列，我们用它来保存学生的唯一编号。一般说来，学生们的学号应该从一个权威机构（如学校办公室）获得。但既然这只是一个示例性的数据表，我们不妨自己编造一些。我们使用了一个 AUTO_INCREMENT 数据列，对它的定义类似于在前面创建 member 数据表时对其 member_id 数据列的定义。

需要提醒大家的是，如果真的是从学校办公室获得学生 ID 而不是自动生成，就千万不要在定义 student_id 数据列时给它加上 AUTO_INCREMENT 属性。但为了避免出现重复的 ID 或 NULL ID 值，PRIMARY KEY 子句还是要保留下来的。

现在，CREATE TABLE 语句末尾的 ENGINE 子句是干什么用的？如果给出了这个子句，它将指定 MySQL 用来创建新数据表的存储引擎的名字。一种“存储引擎”就是一种用来管理某种特定类型的数据表的处理器。MySQL 有好几种存储引擎，每一种都有它自己的特性，我们将在 2.6.1 节对此展开讨论。

如果省略了 ENGINE 子句，MySQL 会替你选择一个默认引擎，它通常是 MyISAM。“ISAM”是“indexed sequential access method”（索引化顺序访问方法）的缩写，MyISAM 引擎在这种访问方法的基础上增加了一些 MySQL 独有的东西。因为我们刚才为“美国历史学会”创建数据表（president 和 member）的时候没有提供 ENGINE 子句，所以它们将是些 MyISAM 数据表（除非你曾重新配置过你的 MySQL 服务器，让其使用另外一种默认引擎）。至于那个考试成绩记录项目，我们明确地使用了

InnoDB 存储引擎。InnoDB 引擎通过引入“外键”概念而具备了保持“引用一致性”的特点。这意味着我们可以通过 MySQL 让数据表之间的关系满足一定的约束条件，而这对考试成绩记录项目中的数据表来说很有必要。

- 考试成绩与考试事件和学生是相关联的：如果某个学生ID和考试事件ID在student和grade_event数据表里尚不存在，就不应该把考试成绩录入到score数据表里去。
- 类似地，缺勤记录与学生相关联：如果某个学生ID在student数据表里尚不存在，就不应该把缺勤情况录入absence数据表。

为了满足这些条件，需要设置几个外键关系。“外”在这里的含义是“在另一个数据表里”，“外键”的含义是一个给定的键值必须与另一个数据表里的某个键值相匹配。这些概念会随着我们为考试成绩记录项目创建更多的数据表而变得越来越明晰。

- grade_event数据表

grade_event 数据表的定义如下所示：

```
CREATE TABLE grade_event
(
  date     DATE NOT NULL,
  category ENUM('T','Q') NOT NULL,
  event_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (event_id)
) ENGINE = InnoDB;
```

为了创建这个数据表，你既可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_grade_event.sql
```

date 数据列用来保存一个标准的 MySQL 日期值，必须写成 *'CCYY-MM-DD'* 的格式。

category 表示考试分数的类型。类似于 student 表里的 sex 列，category 也是一个枚举类型的数据列。它的可取值是'T'和'Q'，分别代表 test 和 quiz。

event_id 是一个 AUTO_INCREMENT 类型的数据列，并同时被定义为 PRIMARY KEY，它与 student 数据表里的 student_id 数据列情况类似。利用 AUTO_INCREMENT 属性，我们就能方便地生成唯一的事件编号了。与 student 数据表里的 student_id 数据列类似，这些编号到底是多少并不重要，重要的是它们必须是唯一的。

因为这些列中没有缺少任何一个，可将它们定义为 NOT NULL。

- score数据表

下面是用来创建 score 数据表的 CREATE TABLE 语句：

```
CREATE TABLE score
(
  student_id INT UNSIGNED NOT NULL,
  event_id   INT UNSIGNED NOT NULL,
  score      INT NOT NULL,
  PRIMARY KEY (event_id, student_id),
  INDEX (student_id),
  FOREIGN KEY (event_id) REFERENCES grade_event (event_id),
  FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

其中包含新内容：FOREIGN KEY 结构。稍后将介绍它。

为了创建这个数据表，既可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_score.sql
```

score 列是一个 INT，容纳整数值。如果想要纳入像 58.5 这样含小数部分的值，可以使用能表示它们的数据类型，如 DECIMAL 或 FLOAT。

student_id 和 event_id 数据列都是 INT（整数）类型的数据列，它们分别代表着每一个考试分数所对应的学生和考试事件。我们将通过它们把 student 和 grade_event 数据表联系起来以查出学生姓名和考试日期。student_id 和 event_id 数据列有一些需要注意的重点，如下所示。

- 我们已经把这两个数据列的组合设置为一个PRIMARY KEY。这确保了我们不会在某次考试或测验结束后重复录入某位学生的成绩。请注意，只有event_id和studemt_id的组合才具备我们需要的唯一性。在score数据表里，这两个ID值单独使用时都不具备唯一性的：同样的event_id值会出现在多个考试成绩数据行里（每位学生对应一个），同样的studemt_id值也会出现在多个数据行里（因为每位学生每参加一次考试或测验都会有一个成绩）。
- 每个ID数据列都需要一条FOREIGN KEY子句来定义它应该遵守的约束条件。这个子句的REFERENCES部分用来指定score数据列应该与哪个数据表里的哪个数据列相对应。event_id数据列上的约束条件是这个数据列里的每个值必须与grade_event数据表里的某个event_id值相匹配。类似地，score数据表里的每个student_id值必须与student数据表里的某个student_id值相匹配。

上面描述的 PRIMARY KEY 定义可以确保我们不会创建重复的考试成绩数据行，而 FOREIGN KEY 定义可以确保在我们的数据行不会有在 grade_event 或 student 数据表里并不存在的虚假 ID 值。

为什么 student_id 数据列上有一个索引？这是因为，对于出现在 FOREIGN KEY 定义里的每一个数据列，它要么本身有一个索引，要么是某个多数据列索引里第一个被列出的数据列。对于 event_id 数据列上的 FOREIGN KEY，该数据列在我们定义 PRIMARY KEY 时是第一个被列出来的。对于 student_id 数据列上的 FOREIGN KEY，就不能从 PRIMARY KEY 方面去找理由了，因为 student_id 数据列没被列在第一个。因此，我们需要在 student_id 数据列上另行创建一个索引。

值得一提的是，InnoDB 存储引擎其实会为出现在外键定义里的数据列自动创建一个索引，但它使用的索引定义不一定是你想要的（详见 2.14.1 节里的讨论），由你来明确地定义一个索引可以避免这个问题。

- absence数据表

下面是我们用来记录缺勤学生的 absence 数据表：

```
CREATE TABLE absence
(
  student_id INT UNSIGNED NOT NULL,
  date       DATE NOT NULL,
  PRIMARY KEY (student_id, date),
  FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

为了创建这个数据表，既可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_absence.sql
```

student_id 和 date 数据列都被定义为 NOT NULL，因为它们的内容都不允许缺失。为了避免这个数据表里出现重复的行，我决定把这两个数据列的组合也定义为一个主键。不管怎么说，把学生缺勤一天的情况统计为两次肯定是不公平的。

absence 表也包含一个外键关系，用来确保每个 student_id 值都与 student 表中的一个 student_id 值相匹配。

我们为考试成绩记录项目的数据表设置外键关系，是为了让那些约束条件能够在数据录入阶段发挥作用：只插入那些包含合法的考试事件 ID 值和学生 ID 值的数据行。不过，外键关系还有另外一种效果。它们会形成依赖关系，使你在创建和丢弃那些数据表的时候必须按照一定的顺序进行。

- score数据表依赖于grade_event和student数据表，所以在创建score数据表之前必须先创建出它们。类似地，adsence数据表依赖于student数据表，在创建adsence数据表时student数据表必须已经存在。
- 在丢弃数据表的时候，必须把上面的顺序倒过来。如果不先丢弃score数据表，就无法丢弃grade_event数据表；如果不先丢弃score和absence数据表，就无法丢弃student数据表。

注意 如果你的 MySQL 服务器因为某种原因不能提供 InnoDB 支持，你可以把考试成绩记录项目里的数据表创建为一些 MyISAM 数据表。把每一条 CREATE TABLE 语句里的 InnoDB 替换为 MyISAM 或者干脆省略 ENGINE 子句就能达到这一目的。不过，如果使用 MyISAM 数据表的话，本书后面的内容里用这些数据表去演示外键用法的例子就看不到效果了。

1.4.7 如何添加新的数据行

现在，我们已经把数据库和它里面的数据表都创建好了。接下来，我们需要往数据表里放一些行。但在此之前，我想先介绍一下如何查找数据表里的内容——在往里面放了一些记录之后，应该先看看自己做得怎么样吧。虽然我把有关检索操作的详细介绍安排在 1.4.9 节里，但你现在至少应该先把下面这条语句弄明白，它是用来查看名为 *tbl_name* 的数据表里的全部内容：

```
SELECT * FROM tbl_name;
```

例如：

```
mysql> SELECT * FROM student;
Empty set (0.00 sec)
```

现在，mysql 会报告说这个数据表是空的，但经过本小节中的几次示例操作之后，你就会看到不同的结果了。

往数据库添加数据的办法有好几种。你可以用 INSERT 语句以手工方式逐行插入到数据表里；也可以利用一个文件把行添加到数据表里，这个文件的内容既可以是一系列提前写好的 INSERT 语句(数据将通过客户程序 mysql 被加载到数据库里)，也可以是纯粹的数据值（将通过 LOAD DATA 语句或 mysqlimport 工具程序被加载到数据库里)。

本小节将介绍把记录插入到数据表的各种方法。大家应该对它们都进行练习，熟悉并掌握它们的工作原理和用法。练习完这些方法之后，转到 1.4.8 节，运行其中的命令。这些命令用来删除数据表，然后重建，再把书中一整套已知数据加载到里面去。这样，你的数据库里的内容就与我将在后面示例中用到的数据一样了，而你自己做示例练习时看到的结果也将会与书中给出的结果一致。(如果已经

知道如何插入数据行，可以直接跳到本节末尾去填充你的数据表。）

1. 利用INSERT语句添加数据

我们先来学习如何用 INSERT 语句来添加数据记录。这是一条 SQL 语句，用来指定你打算往哪个数据表插入一个数据行以及该数据行的各数据列的值。INSERT 语句有好几种形式。

(1) **你可以一次性地列出全部数据列的值**，如下所示：

```
INSERT INTO tbl_name VALUES(value1,value2,...);
```

例如：

```
mysql> INSERT INTO student VALUES('Kyle','M',NULL);
mysql> INSERT INTO grade_event VALUES('2008-09-03','Q',NULL);
```

在使用这个语法的时候，关键字 VALUES 后面的括号里必须为数据表的全体数据列准备好对应的值，这些值的先后顺序也必须与各数据列在数据表里的存储先后顺序保持一致。（这个顺序通常就是各数据列在用来创建这个数据表的 CREATE TABLE 语句里的出现顺序。）如果你拿不准数据列的先后顺序，可以先用一条*数据表名称*语句来查一下。

MySQL 里的字符串或日期值必须放在单引号或双引号里才能被引用，放在单引号里更标准些。NULL 值对应于 student 和 event 数据表里的 AUTO_INCREMENT 数据列。在一个 AUTO_INCREMENT 数据列里插入一个表示“无数据”的 NULL 值将使 MySQL 为这个数据列自动生成下一个序号。

在 MySQL 中可以用一条 INSERT 语句把多个数据行插入到数据表里去，具体语法如下：

```
INSERT INTO tbl_name VALUES(...),(...),... ;
```

例如：

```
mysql> INSERT INTO student VALUES('Avery','F',NULL),('Nathan','M',NULL);
```

与刚才必须使用多条 INSERT 语句的情况相比，这种做法不仅能让你少打不少字，还能提高服务器的执行效率。注意，括号内包含了每行的一组列值。下列语句是非法的，因为它没在括号内包含正确数目的值。

```
mysql> INSERT INTO student VALUES('Avery','F',NULL,'Nathan','M',NULL);
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

(2) **还可以直接对数据列进行赋值，先给出数据列的名字，再列出它的值**。这特别适用于你创建的记录只有少数几个数据列需要有初始值的情况。具体语法如下：

```
INSERT INTO tbl_name (col_name1,col_name2,...) VALUES(value1,value2,...);
```

例如：

```
mysql> INSERT INTO member (last_name,first_name) VALUES('Stein','Waldo');
```

这种形式的 INSERT 语句一次可以插入多个记录：

```
mysql> INSERT INTO student (name,sex) VALUES('Abby','F'),('Joseph','M');
```

没有在 INSERT 语句中出现的数据列将被赋予默认值。例如，上面两条语句没有给出 member_id 或 event_id 数据列的值，所以 MySQL 将把默认值 NULL 赋给它们。（又因为 member_id 和 event_id 都是 AUTO_INCREMENT 数据列，所以结局将是这两个数据列被赋值为 MySQL 自动生成的下一个序列号。这与你直接把 NULL 赋值给它们的效果是一样的。）

(3) **还可以用包含 col_name = value（而非 VALUES()列表）的 SET 子句对数据列赋值。**

```
INSERT INTO tbl_name SET col_name1=value1, col_name2=value2, ... ;
```

例如：

```
mysql> INSERT INTO member SET last_name='Stein',first_name='Waldo';
```

没有在 SET 子句里出现的数据列将被赋予默认值。这种形式的 INSERT 语句不允许一次插入多个数据行。

既然知道了 INSERT 语句的工作原理，你可以用它去核实一下我们建立的外键关系是不是真的能够阻止“坏”数据行被录入到 score 和 absence 数据表。随便找几个没在 grade_event 或 student 数据表里出现过的 ID 值编造些“坏”数据行，看能不能把它们插入到数据表里：

```
mysql> INSERT INTO score (event_id,student_id,score) VALUES(9999,9999,0);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`score`, CONSTRAINT `score_ibfk_1` FOREIGN
KEY (`event_id`) REFERENCES `grade_event` (`event_id`))
mysql> INSERT INTO absence SET student_id=9999, date='2008-09-16';
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`absence`, CONSTRAINT `absence_ibfk_1`
FOREIGN KEY (`student_id`) REFERENCES `student` (`student_id`))
```

错误消息表明限制在起作用。

2. 通过从文件中读取来添加新行

把数据记录加载到数据表里的另一种方法从一个文件里把它们直接读出来。例如，如果 sampdb 发行版本里有一个名为 insert_president.sql 的文件，文件内容是一系列用来把新行添加到 president 数据表里的 INSERT 语句，你就可以像下面这样直接执行它们：

```
% mysql Sampdb < insert_president.sql
```

如果你已经进入 mysql，可以用一条 SOURCE 命令读入这个文件，如下所示：

```
mysql> source insert_president.sql;
```

SOURCE 命令只能用在 MySQL 3.23.9 或更高的版本里。

如果文件里的记录项不是以 INSERT 语句而是以纯数据值的形式来存放的，我们可以利用 LOAD DATA 语句或 mysqlimport 工具程序来加载它们。

LOAD DATA 语句就像是一架大型装载机，它能把文件里的数据一次性地全部读到数据表里。这条语句要在 mysql 客户程序里使用：

```
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

假设数据文件 member.txt 就保存在你正使用着的客户主机上的当前子目录里，上面这条语句将读这个文件并把它的内容发送到服务器以加载到 member 数据表。（你可以在 sampdb 发行版本里找到这个 member.txt 文件。）

在默认的情况下，LOAD DATA 语句将假设各数据列的值以制表符分隔，各数据行以换行符分隔，数据值的排列顺序与各数据列在数据表里的先后顺序一致。但你完全可以用它来读取其他格式的数据文件或者按其他顺序来读取各数据列的值，有关细节请参阅附录 E 里的 LOAD DATA 条目。

LOAD DATA 中的 LOCAL 关键字可以使客户程序（本例中是 mysql）读取数据文件并发送到服务器以加载。如果省略了关键字 LOCAL，就表示数据文件是保存在服务器主机上的，而你必须拥有相应的 FILE 服务器访问权限才能把文件里的数据加载到数据表里去。但可惜的是，大多数 MySQL 用户都没

有这种权限。还应指定文件的完全路径，这样服务器才能找到它。

如果使用 LOAD DATA LOCAL 时得到下面的错误，LOCAL 功能在默认的情况下就可能处于禁用状态。

```
ERROR 1148 (42000): The used command is not allowed with this MySQL version
```

你可以试试加上--local-infile 选项再重新启动一次 mysql 程序的办法，比如：

```
% mysql --local-infile sampdb
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

如果这一招也不管用，就说明服务器端的 LOCAL 机制没有被激活。激活服务器端 LOCAL 机制的具体做法请参阅第 12 章。

加载数据文件的另一种方法是使用 mysqlimport 客户程序。当你在命令提示符下启动 mysqlimport 程序时，它会为你生成一条 LOAD DATA 语句：

```
% mysqlimport --local sampdb member.txt
```

此外，与你使用 mysql 客户程序时一样，如果还需要设定连接参数，请在命令行上把它们添加到数据库名称的前面。

就上面这条命令而言，mysqlimport 程序将生成一条能够把 member.txt 文件里的数据值加载到 member 数据表里去的 LOAD DATA 语句。这是因为 mysqlimport 程序是根据数据文件的名字来确定与之对应的数据表的，它将把文件名中第一个句号字符（.）之前的那个字符串用做数据表的名字。举例来说，它会把 member.txt 和 president.txt 文件里的数据分别加载到 member 和 president 数据表里去。这就要求你必须慎重选择数据文件的名字，要不然，mysqlimport 程序就会把数据错误地加载到别的数据表里去。我们来看一个例子：如果你想把 member1.txt 和 member2.txt 文件里的数据都加载到 member 数据表里去，但 mysqlimport 却会认为你想把这两个文件分别加载到名为 member1 和 member2 的两个数据表里去。为了避免出现这种混乱，你可以把这两个文件命名为 member.1.txt 和 member.2.txt，或者是 member.txt1 和 member.txt2。

1.4.8　将 sampdb 数据库重设为原来的状态

在练习完上面介绍的这几种数据行添加方法之后，为了顺利进行后面的学习，你应该重新创建和加载 sampdb 数据库里的数据表，把它们的内容恢复为原样。使用包含 sampdb 发布文件的目录中的 mysql 程序，写出以下语句：

```
% mysql sampdb
mysql> source create_member.sql;
mysql> source create_president.sql;
mysql> source insert_member.sql;
mysql> source insert_president.sql;
mysql> DROP TABLE IF EXISTS absence, score, grade_event, student;
mysql> source create_student.sql;
mysql> source create_grade_event.sql;
mysql> source create_score.sql;
mysql> source create_absence.sql;
mysql> source insert_student.sql;
mysql> source insert_grade_event.sql;
mysql> source insert_score.sql;
```

```
mysql> source insert_absence.sql;
```

如果你不喜欢输入这么多条命令，那么，在 Unix 系统上，请执行下面这条命令：

```
% sh init_all_tables.sh sampdb
```

在 Windows 系统上，请执行下面这条命令：

```
C:\> init_all_tables.bat sampdb
```

无论使用哪条命令，如果你还需要设定连接参数，请在命令行上把它们添加到命令名称的后面。

1.4.9 检索信息

现在，数据表都已经创建出来并加载了数据。下面看看这些数据都能派上哪些用场。SELECT 语句允许以你喜欢的方式检索和显示数据表里的信息。例如，可以像下面这样把整个数据表的内容都显示出来：

```
SELECT * FROM president;
```

也可以像下面这样只选取某个数据行里的某个数据列：

```
SELECT birth FROM president WHERE last_name = 'Eisenhower';
```

SELECT 语句还有好几个子句（也叫组成部分），它们的各种搭配能帮你查出你最感兴趣的信息。这些子句可以很简单，也可以很复杂，由它们搭配出来的 SELECT 语句也会相应地变得简单或者复杂。不过，请放心，在本书里，绝不会有长达数页让大家必须花费好几个钟头才能搞明白的查询命令。（如果在看书时遇到长长的查询命令，我通常会跳过它们，我想你也会如此。）

下面是 SELECT 语句的通用形式：

```
SELECT what to retrieve
FROM table or tables
WHERE conditions that data must satisfy;
```

在写 SELECT 语句时，先把你想检索的东西说清楚，再把可选子句写出来。上面两个子句（FROM 和 WHERE）是最常见的，其他子句包括 GROUP BY、ORDER BY 和 LIMIT 等。需要指出的是，SQL 语言对书写格式并没有严格的要求，所以在书写 SELECT 语句时，放置换行符的位置不必与本书示例中的一样。

FROM 子句一般都少不了，但如果你不需要给出数据表的名字，就不必把它写出来。比如说，下面这条语句只是计算一个表达式的值。因为这个计算不涉及任何数据表，所以没有必要把 FROM 子句写出来：

```
mysql> SELECT 2+2, 'Hello, world', VERSION();
+-----+--------------+------------+
| 2+2 | Hello, world | VERSION()  |
+-----+--------------+------------+
|   4 | Hello, world | 5.0.60-log |
+-----+--------------+------------+
```

当的确需要使用一个 FROM 子句来指定将从哪个数据表检索数据时，你还需要把想查看的数据列的名字列举出来。SELECT 语句最常用形式是用一个星号（表示所有数据列）作为数据列说明符。下面这条查询将把 student 数据表所有的数据列全都显示出来：

```
mysql> SELECT * FROM student;
+-----------+-----+------------+
```

```
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

有关数据列将按它们在数据表里的存储先后顺序显示出来。这个顺序与你用 DESCRIBE student 语句查看到的数据列排列顺序是一致的。(示例末尾处的省略号表示这个查询所返回的数据行比大家在这里看到的要多。)

你也可以把自己想要查看的数据列的名字逐个列出来。例如，如果你只想查看学生姓名，就应该使用下面这条语句：

```
mysql> SELECT name FROM student;
+-----------+
| name      |
+-----------+
| Megan     |
| Joseph    |
| Kyle      |
| Katie     |
...
```

如果需要列举多个数据列，请用逗号把它们分隔开。请看下面这条语句，它与 SELECT* FROM student 是等价的，但它把各数据列的名字都明确地列了出来：

```
mysql> SELECT name, sex, student_id FROM student;
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

你可以按任意顺序列举数据列的名字：

```
SELECT name, student_id FROM student;
SELECT student_id, name FROM student;
```

只要你愿意，甚至还可以重复列举数据列的名字，只是这样做通常没有多大的意义。

MySQL 允许你在一条 SELECT 语句里同时选取多个数据表里的数据列，这叫做数据表之间的联结(join)，我们将在第 4 小节里对此进行讨论。

MySQL 里的数据列名称不区分字母的大小写，所以下面这些查询都是等价的：

```
SELECT name, student_id FROM student;
SELECT NAME, STUDENT_ID FROM student;
SELECT nAmE, sTuDeNt_Id FROM student;
```

但需要注意的是，数据库和数据表的名字却可能需要区分字母的大小写，这取决于服务器主机上所使用的文件系统，以及 MySQL 的配置情况。比如说，Windows 文件名不区分大小写，所以运行在 Windows 系统上的服务器也就不区分数据库和数据表名字的大小写；Unix 文件名区分大小写，所以运

行在 Unix 系统上的服务器就将区分数据库和数据表名字的大小写。（属于 Unix 阵营的 Mac OS X 是个例外：HFS+文件系统不区分文件名的大小写，但 UFS 文件系统却区分。）如果你想让 MySQL 服务器不区分数据库和数据表名字中的大小写，可以对服务器进行配置，请参阅 11.2.5 节。

1. 指定检索条件

要想让 SELECT 语句只把满足特定条件的记录检索出来，就必须给它加上一个 WHERE 子句来设定数据行的检索条件。只有这样，你才能有选择地把数据列的取值满足特定要求的那些数据行挑选出来。你可以针对任何类型的值进行查找。例如，可以针对某些数值进行搜索：

```
mysql> SELECT * FROM score WHERE score > 95;
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          5 |        3 |    97 |
|         18 |        3 |    96 |
|          1 |        6 |   100 |
|          5 |        6 |    97 |
|         11 |        6 |    98 |
|         16 |        6 |    98 |
+------------+----------+-------+
```

也可以针对字符串值进行查找。对于默认的字符设置和排序，字符串的比较操作通常不区分字母的大小写：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='ROOSEVELT';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

还可以针对日期值进行查找：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < '1750-1-1';
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
+------------+------------+------------+
```

甚至还能针对不同类型的值的组合进行查找：

```
mysql> SELECT last_name, first_name, birth, state FROM president
    -> WHERE birth < '1750-1-1' AND (state='VA' OR state='MA');
+------------+------------+------------+-------+
| last_name  | first_name | birth      | state |
+------------+------------+------------+-------+
| Washington | George     | 1732-02-22 | VA    |
| Adams      | John       | 1735-10-30 | MA    |
| Jefferson  | Thomas     | 1743-04-13 | VA    |
+------------+------------+------------+-------+
```

WHERE 子句里的表达式允许使用算术运算符（见表1-1）、比较运算符（见表1-2）和逻辑运算符（见表1-3），还允许使用括号。你可使用常数、数据表的数据列和函数调用进行运算。在这本书的查询示例里用到的 MySQL 函数不算很多，但它们的数量其实并不少。MySQL 函数的完整清单请参阅附录 C。

表1-1　算术运算符

运算符	含　义
+	加法
-	减法
*	乘法
/	除法
DIV	整数除法
%	求余（整数除法后的剩余部分）

表1-2　比较运算符

运算符	含　义
<	小于
<=	小于或等于（不大于）
=	等于
<=>	等于（能够对NULL值进行比较）
<> 或 !=	不等于
>=	大于或等于（不小于）
>	大于

表1-3　逻辑运算符

运算符	含　义
AND	逻辑与
OR	逻辑或
XOR	逻辑异或
NOT	逻辑非

如果需要在查询语句里使用逻辑运算符，千万要注意一点：逻辑 AND 运算与人们日常生活中所说的“和”在含义上是不一样的。举个例子，假设你想把“出生于弗吉尼亚州和出生于马萨诸塞州的总统”查出来。这道问题里有一个“和”字，头脑简单的人会不假思索地写出一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' AND state='MA';
Empty set (0.36 sec)
```

这个查询的结果集显然是空的，没有把我们想要的东西找出来。为什么会这样呢？因为这条查询的真正含义是“把同时出生在弗吉尼亚州和马萨诸塞州的总统”找出来，而这种情况是不可能出现的。在日常生活里，你可以用“和”来表达你的查询条件，但在 SQL 里，你却必须把这两个条件用逻辑或操作符 OR 并列在一起，即：

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' OR state='MA';
+------------+-------------+-------+
| last_name  | first_name  | state |
+------------+-------------+-------+
| Washington | George      | VA    |
| Adams      | John        | MA    |
| Jefferson  | Thomas      | VA    |
| Madison    | James       | VA    |
| Monroe     | James       | VA    |
| Adams      | John Quincy | MA    |
```

```
| Harrison   | William H.  | VA    |
| Tyler      | John        | VA    |
| Taylor     | Zachary     | VA    |
| Wilson     | Woodrow     | VA    |
| Kennedy    | John F.     | MA    |
| Bush       | George H.W. | MA    |
+------------+-------------+-------+
```

请大家务必注意日常语言与 SQL 语句之间的这类差异——在为你自己编写查询命令时要注意，在为其他人编写查询命令时更要注意。一定要仔细倾听别人对查询内容的描述，然后根据描述正确地选用适当的 SQL 逻辑操作符。以刚才那个查询为例，正确的自然语言表述形式应该是：“把出生于弗吉尼亚州或者出生于马萨诸塞州的总统给找出来。”

你可能会发现像下面这样，在表达查询时，用 IN()操作符来查找几个值中的某一个会很方便。前面的查询可以使用 IN()写成下面这样：

```
SELECT last_name, first_name, state FROM president
WHERE state IN('VA','MA');
```

在比较一个列和一大组值时，IN()使用起来特别方便。

2. NULL值

NULL 是一个很特殊的值。它的含义是“无数据”或“未知数据”，所以不能用它与“有数据”的值进行运算或者比较。如果你试图用普通的算术比较运算符对 NULL 值进行操作，其结果将是不可预料的：

```
mysql> SELECT NULL < 0, NULL = 0, NULL <> 0, NULL > 0;
+----------+----------+-----------+----------+
| NULL < 0 | NULL = 0 | NULL <> 0 | NULL > 0 |
+----------+----------+-----------+----------+
|     NULL |     NULL |      NULL |     NULL |
+----------+----------+-----------+----------+
```

事实上，你甚至不应该把 NULL 值与它本身进行比较，因为两个表示“无数据”的未知值的比较结果也将是不可预料的：

```
mysql> SELECT NULL = NULL, NULL <> NULL;
+-------------+--------------+
| NULL = NULL | NULL <> NULL |
+-------------+--------------+
|        NULL |         NULL |
+-------------+--------------+
```

如果需要对 NULL 值进行查找，就必须使用一种特殊的语法。你不能使用=、<>或者!=来测试它们是相等还是不相等，你必须使用 IS NULL 或 IS NOT NULL 来判断。例如，如果你想把目前仍然健在的美国总统给查出来，就应该使用一条下面这样的查询命令，因为这些总统的逝世日期在 president 数据表里是用 NULL 值来表示的：

```
mysql> SELECT last_name, first_name FROM president WHERE death IS NULL;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Carter    | James E.    |
| Bush      | George H.W. |
```

```
| Clinton    | William J.  |
| Bush       | George W.   |
+------------+-------------+
```

如果想把没有姓名后缀的美国总统查出来，就应该在检索条件里使用 IS NOT NULL 进行判断：

```
mysql> SELECT last_name, first_name, suffix
    -> FROM president WHERE suffix IS NOT NULL;
+-----------+------------+--------+
| last_name | first_name | suffix |
+-----------+------------+--------+
| Carter    | James E.   | Jr.    |
+-----------+------------+--------+
```

专用的 MySQL 比较操作符<=>能完成 NULL 值与 NULL 值之间的比较。你可以用这个操作符把刚才的两条查询命令分别改写为：

```
SELECT last_name, first_name FROM president WHERE death <=> NULL;

SELECT last_name, first_name, suffix
FROM president WHERE NOT (suffix <=> NULL);
```

3. 如何对查询结果进行排序

只要你是 MySQL 用户，迟早会注意到这样一种情况：如果你创建了一个数据表并往里面加载了一些数据记录，当你发出一条“SELECT * FROM 数据表名称”语句时，数据记录在查询结果中的先后顺序通常与它们当初被插入时的先后顺序一致。这很符合人们的思维习惯，人们很自然地假设数据记录在查询结果中的先后顺序与它们当初被插入时的先后顺序相同。但这是不正确的，因为如果你在加载完数据表的初始数据之后又删除并插入了一些数据行，这些操作往往会改变数据行在服务器所返回的数据表检索结果中的先后顺序。（数据删除操作会在数据表里留下一些“空洞”，而 MySQL 会用你以后插入的新记录来尽量填补这些“空洞”。）

你真正可以信赖的原则是：从服务器返回的数据行的先后顺序没有任何保证，除非你事先设定。如果想让查询结果按你希望的先后顺序显示，就必须给查询命令增加一条 ORDER BY 子句。下面这条查询命令将把美国总统们的姓名按姓氏字母表顺序排列并显示出来：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name;
+------------+----------------+
| last_name  | first_name     |
+------------+----------------+
| Adams      | John           |
| Adams      | John Quincy    |
| Arthur     | Chester A.     |
| Buchanan   | James          |
...
```

ORDER BY 子句中的默认排序方式是升序排列。在 ORDER BY 子句中的数据列名字的后面加上关键字 ASC 或 DESC，就能使查询结果中的数据记录按指定数据列的升序或者降序排列，例如，如果你想让美国总统们的姓名按姓氏的逆序（降序）排列显示，就应该像下面这样加上一个 DESC 关键字：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name DESC;
```

```
+------------+---------------+
| last_name  | first_name    |
+------------+---------------+
| Wilson     | Woodrow       |
| Washington | George        |
| Van Buren  | Martin        |
| Tyler      | John          |
...
```

可以对查询结果按多个数据列进行排序，而每一个数据列又都可以互不影响地分别按升序或降序进行排列。下面这条针对 president 数据表的查询命令将把查询结果中的数据行按总统出生地所在州的逆序排列，而出生地所在州相同的总统姓名又将按其姓氏的升序排列：

```
mysql> SELECT last_name, first_name, state FROM president
    -> ORDER BY state DESC, last_name ASC;
+------------+---------------+-------+
| last_name  | first_name    | state |
+------------+---------------+-------+
| Arthur     | Chester A.    | VT    |
| Coolidge   | Calvin        | VT    |
| Harrison   | William H.    | VA    |
| Jefferson  | Thomas        | VA    |
| Madison    | James         | VA    |
| Monroe     | James         | VA    |
| Taylor     | Zachary       | VA    |
| Tyler      | John          | VA    |
| Washington | George        | VA    |
| Wilson     | Woodrow       | VA    |
| Eisenhower | Dwight D.     | TX    |
| Johnson    | Lyndon B.     | TX    |
...
```

对于包含 NULL 值的数据行，如果设定按升序排列，它们将出现在查询结果的开头；如果设定按降序排列，它们将出现在查询结果的末尾。如果你想让包含 NULL 值的数据行必须出现在查询结果的末尾，就必须额外增加一个排序数据列以区分 NULL 值和非 NULL 值。例如，如果你想按逝世日期的降序对总统们的姓名排序，那么健在（即逝世日期等于 NULL）的总统们的姓名将出现在查询结果的末尾，而如果想让后者出现在查询结果的开头，就应该使用一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, death FROM president
    -> ORDER BY IF(death IS NULL,0,1), death DESC;
+------------+---------------+------------+
| last_name  | first_name    | death      |
+------------+---------------+------------+
| Clinton    | William J.    | NULL       |
| Bush       | George H.W.   | NULL       |
| Carter     | James E.      | NULL       |
| Bush       | George W.     | NULL       |
| Ford       | Gerald R.     | 2006-12-26 |
| Reagan     | Ronald W.     | 2004-06-05 |
| Nixon      | Richard M.    | 1994-04-22 |
| Johnson    | Lyndon B.     | 1973-01-22 |
...
| Jefferson  | Thomas        | 1826-07-04 |
```

```
| Adams      | John           | 1826-07-04 |
| Washington | George         | 1799-12-14 |
+------------+----------------+------------+
```

IF()函数的作用是对紧随其后的表达式（第一参数）进行求值，再根据表达式求值结果的真假返回它的第二参数或第三参数。在刚才的第一个查询里，在遇到 NULL 值的时候，IF()函数的求值结果将是 0；在遇到非 NULL 值的时候，IF()函数的求值结果将是 1。最终结果是把包含 NULL 值的数据行全都放在了没有 NULL 值的数据行的前面。

4. 如何限制查询结果中的数据行个数

查询结果往往由很多个数据行构成，如果你只想看到其中的一小部分，可以给查询命令增加一条 LIMIT 子句，它与 ORDER BY 子句联合使用的效果往往更佳。MySQL 允许给查询结果中的数据行个数设置一个上限，如前 *n* 个数据行，如果查询结果里的数据行超过了这个数字，就只显示前 *n* 个数据行。下面这个查询将把出生日期最早的前 5 个总统列举出来：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth LIMIT 5;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

如果你使用了 ORDER BY birth DESC（即按逆序）来排序查询结果，你找出来的就将是出生日期最晚的 5 个总统：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Clinton   | William J.  | 1946-08-19 |
| Bush      | George W.   | 1946-07-06 |
| Carter    | James E.    | 1924-10-01 |
| Bush      | George H.W. | 1924-06-12 |
| Kennedy   | John F.     | 1917-05-29 |
+-----------+-------------+------------+
```

LIMIT 还允许从查询结果的中间部分抽出一部分数据记录。此时必须设定两个值，第一个值给出了要在查询结果的开头部分跳过的数据记录个数，第二个值则是需要返回的数据记录的个数。下面的查询与前一个很相似，但它返回的是跳过前 10 个数据记录之后的 5 个数据记录：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 10, 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Truman    | Harry S     | 1884-05-08 |
| Roosevelt | Franklin D. | 1882-01-30 |
```

```
| Hoover    | Herbert C.  | 1874-08-10 |
| Coolidge  | Calvin      | 1872-07-04 |
| Harding   | Warren G.   | 1865-11-02 |
+-----------+-------------+------------+
```

如果想从president数据表里随机抽取出一条或一组数据记录，可以联合使用LIMIT和ORDER BY RAND()子句，如下所示：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 1;
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Johnson   | Lyndon B.  |
+-----------+------------+
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 3;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Harding   | Warren G.   |
| Bush      | George H.W. |
| Jefferson | Thomas      |
+-----------+-------------+
```

5. 如何对输出列进行求值和命名

前面给出的查询示例的输出结果大都是直接检索自数据表的数据值。MySQL 还允许把表达式的计算结果当做输出列的值，而不引用数据表。表达式可以很简单，也可以很复杂。就拿下面这个查询来说吧，它有两个输出列，前一个输出列对应着一个非常简单的表达式（一个常数），而后一个输出列则对应着一个使用了多个算术运算符和多个函数调用的复杂表达式，它生成一个平方根，并将结果按 3 位小数位表示：

```
mysql> SELECT 17, FORMAT(SQRT(25+13),3);
+----+------------------------+
| 17 | FORMAT(SQRT(25+13),3) |
+----+------------------------+
| 17 | 6.164                  |
+----+------------------------+
```

数据表里的数据列名字也可以用在表达式里，如下所示：

```
mysql> SELECT CONCAT(first_name,' ',last_name),CONCAT(city,', ',state)
    -> FROM president;
+-----------------------------------+--------------------------+
| CONCAT(first_name,' ',last_name) | CONCAT(city,', ',state) |
+-----------------------------------+--------------------------+
| George Washington                 | Wakefield, VA           |
| John Adams                        | Braintree, MA           |
| Thomas Jefferson                  | Albemarle County, VA    |
| James Madison                     | Port Conway, VA         |
...
```

在这个查询里，我们对输出列的格式进行了设置：总统们的名字和姓氏合二为一，成为了一个以空格分隔的字符串；他们的出生城市和出生州则被合并为一个以逗号分隔的字符串。

如果输出列的值是某个表达式的结算结果，这个表达式就会成为这个输出列的名字并用作它在输出结果中的标题。如果表达式很长（例如上面这个查询示例），就会使输出列的宽度变得很大。为解决这一问题，你可以利用 AS name 短语给输出列另外取一个名字，我们把它们称为（输出）列的*别名*(alias)。例如，我们可以像下面这样把上面这个查询的输出结果改写得更有意义：

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS Name,
    -> CONCAT(city,', ',state) AS Birthplace
    -> FROM president;
+----------------------+--------------------------+
| Name                 | Birthplace               |
+----------------------+--------------------------+
| George Washington    | Wakefield, VA            |
| John Adams           | Braintree, MA            |
| Thomas Jefferson     | Albemarle County, VA     |
| James Madison        | Port Conway, VA          |
...
```

如果输出列的别名里包含空格符，就必须把它放到一组引号里，如下所示：

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS 'President Name',
    -> CONCAT(city,', ',state) AS 'Place of Birth'
    -> FROM president;
+----------------------+--------------------------+
| President Name       | Place of Birth           |
+----------------------+--------------------------+
| George Washington    | Wakefield, VA            |
| John Adams           | Braintree, MA            |
| Thomas Jefferson     | Albemarle County, VA     |
| James Madison        | Port Conway, VA          |
...
```

在为数据列提供别名时，关键字 AS 可以省略：

```
mysql> SELECT 1, 2 AS two, 3 three;
+---+-----+-------+
| 1 | two | three |
+---+-----+-------+
| 1 |   2 |     3 |
+---+-----+-------+
```

我个人更喜欢写出 AS。如果省略了它，稍不留神就会写出一个看起来完全合法但执行起来却结果迥异的查询命令来。比如说，你本打算编写一个查询命令去选择总统的姓名，只可惜你不小心漏掉了 first_name 和 last_name 数据列之间的逗号，写出了下面这样的语句：

```
mysql> SELECT first_name last_name FROM president;
+---------------+
| last_name     |
+---------------+
| George        |
| John          |
| Thomas        |
| James         |
...
```

这样一来，查询结果不再是两个输出列了。它只能显示 first_name 数据列的内容，last_name

被视为一个数据列别名而成为了输出列的表头。如果某个查询命令检索出来的输出列的个数不符合你的预期，并且输出列的名字也不符合你的预期，就应该去检查一下是不是在什么地方漏掉了数据列之间的逗号。

6. 与日期有关的问题

在与 MySQL 里的日期打交道的时候，千万要记住年份总是出现在最前面。2008 年 7 月 27 日将被表示为'2002-07-27'，而不是像在日常生活中那样被表示为'07-27-2002'或'27-07-2002'。

MySQL 已经为我们准备了一些日期操作，比较常见的有下面几种。

- 按日期排序。（我们已经见过几个这方面的例子了。）
- 查找某个日期或者某个日期范围。
- 提取日期值中的年、月、日等组成部分。
- 计算两个日期之间的时间距离。
- 用一个日期加上或减去一个时间间隔以求出另一个日期。

下面是一些与日期有关的查询示例。

如果你的查询与某特定日期有关，就得把一个 DATE 类型的数据列与你感兴趣的那个日期值进行比较，如下所示：

```
mysql> SELECT * FROM grade_event WHERE date = '2008-10-01';
+------------+----------+----------+
| date       | category | event_id |
+------------+----------+----------+
| 2008-10-01 | T        |        6 |
+------------+----------+----------+
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-01-01' AND death < '1980-01-01';
+-----------+------------+------------+
| last_name | first_name | death      |
+-----------+------------+------------+
| Truman    | Harry S    | 1972-12-26 |
| Johnson   | Lyndon B.  | 1973-01-22 |
+-----------+------------+------------+
```

日期中的年、月、日 3 部分可以用函数 YEAR()、MONTH()、DAYOFMONTH()分别分离出来。例如，下面这个查询将把生日在 3 月的美国总统查出来：

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

在这个查询里，还可以直接使用 3 月份的英文名称 March：

```
mysql> SELECT last_name, first_name, birth
```

```
    -> FROM president WHERE MONTHNAME(birth) = 'March';
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

再进一步，把 MONTH()和 DAYOFMONTH()函数结合起来使用，就能把生日是 3 月的某一天的总统查出来：

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3 AND DAYOFMONTH(birth) = 29;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Tyler     | John       | 1790-03-29 |
+-----------+------------+------------+
```

很多报纸的娱乐版都有“今日出生的名人”之类的栏目，而上面的查询就能生成一份这样的名单。不过，如果你的查询与“今天”有关，那大可不必像前面的例子那样使用一个具体的日期值。MySQL 为我们准备了一个 CURDATE()函数，它的返回值永远是“今天”的日期值。于是，无论“今日”是哪一天，将总统的生日与 CURDATE()比较，“今日出生的总统”就可以用下面这个查询找出来：

```
SELECT last_name, first_name, birth
FROM president WHERE MONTH(birth) = MONTH(CURDATE())
AND DAYOFMONTH(birth) = DAYOFMONTH(CURDATE());
```

如果你想知道两个日期值之间的时间间隔，拿它们做减法就行了。例如，如果你想知道哪位总统的寿命最长，就得用他们的逝世日期减去他们的出生日期。此时 TIMESTAMPDIFF()函数很有用，有接受一个参数，用以指定计算结果的单位，本例中是年，即 YEAR。如下所示：

```
mysql> SELECT last_name, first_name, birth, death,
    -> TIMESTAMPDIFF(YEAR, birth, death) AS age
    -> FROM president WHERE death IS NOT NULL
    -> ORDER BY age DESC LIMIT 5;
+-----------+------------+------------+------------+------+
| last_name | first_name | birth      | death      | age  |
+-----------+------------+------------+------------+------+
| Reagan    | Ronald W.  | 1911-02-06 | 2004-06-05 |   93 |
| Ford      | Gerald R.  | 1913-07-14 | 2006-12-26 |   93 |
| Adams     | John       | 1735-10-30 | 1826-07-04 |   90 |
| Hoover    | Herbert C. | 1874-08-10 | 1964-10-20 |   90 |
| Truman    | Harry S    | 1884-05-08 | 1972-12-26 |   88 |
+-----------+------------+------------+------------+------+
```

计算日期间隔的另一种方法是使用 TO_DAYS 函数，它将日期都转换为天数。这个函数可以计算出距某特定日期还有多长的时间。例如，可以这样找出需要在近期交纳会费的美国历史研究会会员：用会员资格的失效日期减去“今天”的日期，若其结果小于某个极值，就表明这位会员需要续费了。下面这个查询将把资格已经失效和需要在 60 天以内续交会费的会员查出来：

```
SELECT last_name, first_name, expiration FROM member
WHERE (TO_DAYS(expiration) - TO_DAYS(CURDATE())) < 60;
```

使用 TIMESTAMPDIFF()函数也可以完成，如下所示：

```
SELECT last_name, first_name, expiration FROM member
WHERE TIMESTAMPDIFF(DAY, CURDATE(), expiration) < 60;
```

日期值的用 DATE_ADD()或 DATE_SUB()函数来完成。这两个函数的输入参数是一个日期值和一个时间间隔值，返回结果则是一个新日期值。请看下面的例子：

```
mysql> SELECT DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+----------------------------------------+
| DATE_ADD('1970-1-1', INTERVAL 10 YEAR) |
+----------------------------------------+
| 1980-01-01                             |
+----------------------------------------+
mysql> SELECT DATE_SUB('1970-1-1', INTERVAL 10 YEAR);
+----------------------------------------+
| DATE_SUB('1970-1-1', INTERVAL 10 YEAR) |
+----------------------------------------+
| 1960-01-01                             |
+----------------------------------------+
```

在本小节的前半部分有一个用来查询“哪些美国总统逝世于 20 世纪 70 年代”的示例，里面用了两个确切的日期值来表示时间的起止点。利用日期值的加减法运算，原来的查询可以被改写为：起点日期仍使用一个确切的日期，但终点日期却是通过起点日期加上一个时间间隔而计算出来的。如下所示：

```
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-1-1'
    -> AND death < DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+-----------+------------+------------+
| last_name | first_name | death      |
+-----------+------------+------------+
| Truman    | Harry S    | 1972-12-26 |
| Johnson   | Lyndon B.  | 1973-01-22 |
+-----------+------------+------------+
```

用来查找“需要在近期交纳会费的会员”的查询可以用 DATE_ADD()改写为：

```
SELECT last_name, first_name, expiration FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL 60 DAY);
```

如果为 expiration 列编制索引，将会比先前的查询更有效率，第 5 章将会讨论原因。

在本章前面的内容里，我曾给出一个牙医诊所用来查找“哪些患者没来参加复查”的查询：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

你当时也许还看不懂这个查询，但现在总该弄明白了吧？

7. 模式匹配

MySQL 支持模式匹配操作，这使我们能够在没有给出精确比较值的情况下把有关的数据行查出来。模式匹配需要使用特殊的操作符（LIKE 和 NOT LIKE），还需要你提供一个包含通配字符的字符串。下划线字符“_”只能匹配一个字符，百分号字符“%”则能匹配任何一个字符序列（包括空序列在内）。

下面这个模式查询将把姓氏以字母 W 或 w 开头的总统姓名查找出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE 'W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
+------------+------------+
```

请大家再来看一个查询，它在使用模式匹配功能时出现了错误；因为它使用的不是 LIKE，而是带有算术比较操作符的模式。

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name = 'W%';
Empty set (0.00 sec)
```

上面这个查询的含义变成了“把姓氏是 W%或 w%的总统找出来”。

下面这个查询将把姓氏里有 W 或 w 字母（并不仅限于姓氏的第一个字母）的总统姓名列出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '%W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
| Eisenhower | Dwight D.  |
+------------+------------+
```

下面这个查询将把姓氏由且仅由 4 个字母构成的总统姓名给查出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '____';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Polk      | James K.    |
| Taft      | William H.  |
| Ford      | Gerald R.   |
| Bush      | George H.W. |
| Bush      | George W.   |
+-----------+-------------+
```

MySQL 还提供基于*正则表达式*（regular expression）和 REXEXP 操作符的另一种模式匹配形式，3.5.1 节中的第 1 小节和附录 C 将进一步讨论 LIKE 和 REGEXP 操作符。

8. 如何设置和使用SQL变量

MySQL 允许自定义变量。我们可以使用查询结果来设置变量，这使我们能够方便地把一些值保存起来以供今后查询。例如，你想知道有哪些总统出生在 Andrew Jackson 总统之前。你可以这样做：先检索出他的出生日期并保存到一个变量里，再把出生日期早于这个变量值的总统查找出来：

```
mysql> SELECT @birth := birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew';
```

```
+-----------------+
| @birth := birth |
+-----------------+
| 1767-03-15      |
+-----------------+
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < @birth ORDER BY birth;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

变量的命名语法是“@变量名”，赋值语法是在 SELECT 语句里使用一个“@变量名:= 值”形式的表达式。因此，上面的第一个查询负责把 Andrew Jackson 总统的出生日期查出来并把它赋值给一个名为@birth 的变量（这条 SELECT 语句的查询结果仍会被显示。把查询结果赋值给一个变量并不会使该查询的输出结果不显示）。第二个查询负责把出生日期早于@birth 变量值的总统查出来。

前面的问题其实可以通过一个联结或子查询在一个查询语句里得到解决，但这里不予讨论。而有时候使用一个变量可能更容易。

SET 语句也能用来对变量赋值，此时，“=”和“:=”都可以用做赋值操作符。如下所示：

```
mysql> SET @today = CURDATE();
mysql> SET @one_week_ago := DATE_SUB(@today, INTERVAL 7 DAY);
mysql> SELECT @today, @one_week_ago;
+------------+---------------+
| @today     | @one_week_ago |
+------------+---------------+
| 2008-03-21 | 2008-03-14    |
+------------+---------------+
```

9. 如何生成统计信息

MySQL 最有用的功能之一是它能够依据大量未经加工的数据生成多种统计汇总信息。大家知道，单纯依靠人工手段来生成统计信息是一项既枯燥又耗时还容易出错的工作。如果大家能掌握使用 MySQL 来生成各种统计信息的技巧，它就会成为你手中最具威力的信息处理工具。

找出一组数据里到底有多少种不同的取值是一项比较常见的统计工作，而关键字 DISTINCT 恰好能让我们把在查询结果中重复出现的数据行清除掉。例如，下面的查询将把美国历届总统的出生地所在州不加重复地列举出来：

```
mysql> SELECT DISTINCT state FROM president ORDER BY state;
+-------+
| state |
+-------+
| AR    |
| CA    |
| CT    |
| GA    |
| IA    |
```

```
| IL    |
| KY    |
| MA    |
| MO    |
| NC    |
| NE    |
| NH    |
| NJ    |
| NY    |
| OH    |
| PA    |
| SC    |
| TX    |
| VA    |
| VT    |
+-------+
```

另一项比较常见的统计工作是利用 COUNT()函数来计数。COUNT(*)能把你的查询到底选取了多少个数据行的情况告诉你。如果你的查询语句没有 WHERE 子句，就会选择所有行 COUNT(*)就会把数据表里的行数告诉你。下面这个查询能让我们知道“美国历史研究会”现在共有多少名会员：

```
mysql> SELECT COUNT(*) FROM member;
+----------+
| COUNT(*) |
+----------+
|      102 |
+----------+
```

如果查询语句带有 WHERE 子句，COUNT(*)会告诉你该子句到底匹配了多少个数据行。例如，下面这个查询能让我们知道你到目前为止已经进行过多少次考试：

```
mysql> SELECT COUNT(*) FROM grade_event WHERE category = 'Q';
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

COUNT(*)的统计结果是被选中的数据行的总数，而 COUNT(*数据列名称*)值则只统计全体非 NULL 值的个数。这二者之间的区别很容易从下面这个查询看出来：

```
mysql> SELECT COUNT(*), COUNT(email), COUNT(expiration) FROM member;
+----------+--------------+-------------------+
| COUNT(*) | COUNT(email) | COUNT(expiration) |
+----------+--------------+-------------------+
|      102 |           80 |                96 |
+----------+--------------+-------------------+
```

从上面的查询结果可以知道，member 数据表目前共有 102 条记录，其中的 80 条在 email 数据列里有一个值。我们还可以推断出研究会目前有 6 名终身会员（expiration 数据列里的 NULL 值表示这是一名终身会员，因为 102 条记录里有 96 条记录的 expiration 数据列里有非 NULL 值，剩下的 6 条记录就必然属于那些终身会员）。

COUNT()和 DISTINCT 联合起来可以统计出查询结果里到底有多少种不同的非 NULL 值。例如，如

果想知道美国总共有多少个州曾经有总统出生，下面这个查询就能告诉你：

```
mysql> SELECT COUNT(DISTINCT state) FROM president;
+------------------------+
| COUNT(DISTINCT state) |
+------------------------+
|                    20 |
+------------------------+
```

你可以对某个数据列进行全面的统计，也可以对该数据列做分门别类的统计。例如，如果你想知道班级里总共有多少名学生，可以使用下面这个查询：

```
mysql> SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|       31 |
+----------+
```

如果想知道班级里分别有多少名男生和女生又该怎么办呢？一种办法是分别对两种性别进行统计，如下所示：

```
mysql> SELECT COUNT(*) FROM student WHERE sex='f';
+----------+
| COUNT(*) |
+----------+
|       15 |
+----------+
mysql> SELECT COUNT(*) FROM student WHERE sex='m';
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
```

这个办法管用，但比较麻烦。如果数据列的取值有很多种的话，这个办法就算管用也不适用了。就拿统计出生于美国各个州的总统人数的问题来说吧：你先得一个不落地把有多少个不同的州出生过总统的情况统计出来(SELECT DISTINCT state FROM president),然后再用一系列 SELECT COUNT(*) 语句去分别统计出生于各州的总统人数。如此麻烦的事情我想是不会有多少人情愿去做的。

值得庆幸的是，MySQL 可以只用一个查询就把某数据列里的不同值分别出现过多少次的情况统计出来。还是用分别统计男、女学生人数的例子，使用 GROUP BY 子句：

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
+-----+----------+
```

分别统计出生于各州的美国总统人数的问题也可以用类似的办法来解决，如下所示：

```
mysql> SELECT state, COUNT(*) FROM president GROUP BY state;
```

```
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| AR    |        1 |
| CA    |        1 |
| CT    |        1 |
| GA    |        1 |
| IA    |        1 |
| IL    |        1 |
| KY    |        1 |
| MA    |        4 |
| MO    |        1 |
| NC    |        2 |
| NE    |        1 |
| NH    |        1 |
| NJ    |        1 |
| NY    |        4 |
| OH    |        7 |
| PA    |        1 |
| SC    |        1 |
| TX    |        2 |
| VA    |        8 |
| VT    |        2 |
+-------+----------+
```

如果需要进行这种分门别类的统计，GROUP BY 子句就必不可少，它的作用是让 MySQL 知道在统计之前应该如何对有关的数据记录分类。如果你忘了加上这个子句，就会收到一条出错信息。

与反复使用多个彼此近似的查询来分别统计某数据列不同取值出现次数的做法相比，把 COUNT(*) 函数与 GROUP BY 子句相结合的做法有很多优点：

- 在开始统计之前，我们不必知道将被统计的数据列里到底有多少种不同的取值；
- 我们只需使用一个而不是好几个查询；
- 因为只用一个查询就能把所有的结果都查出来，所以我们还能对输出进行排序。

前两项优点的重要性体现在它们有助于简化查询语句的书写。第三项优点的重要性则体现在它能让我们更灵活地显示查询结果。在默认的情况下，MySQL 会根据 GROUP BY 子句里的数据列对查询结果进行排序。但我们也可以通过 ORDER BY 子句按指定顺序排序。例如，如果你想按人数从多到少的顺序把有总统出生的美国各州查出来并排序输出，就可以增加一个如下所示的 ORDER BY 子句：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
| SC    |     1 |
```

```
| NH    |     1 |
| PA    |     1 |
| KY    |     1 |
| NJ    |     1 |
| IA    |     1 |
| MO    |     1 |
| CA    |     1 |
| NE    |     1 |
| GA    |     1 |
| IL    |     1 |
| AR    |     1 |
| CT    |     1 |
+-------+-------+
```

当你准备用来排序的输出列是某个汇总函数的计算结果时，不能直接在 ORDER BY 子句里引用函数，而应该先给这个输出列取一个别名，然后再把这个别名用在 ORDER BY 子句里。上面那个查询就是这样做的：我们给 COUNT(*)输出列取了一个别名叫 count。在 ORDER BY 子句里引用输出列的另一种办法是利用它在输出结果中的位置：

```
SELECT state, COUNT(*) FROM president
GROUP BY state ORDER BY 2 DESC;
```

用输出列的出现位置来设定排序顺序的做法在 MySQL 中允许的，但存在着弊病。

- 它很容易导致查询命令难以阅读，因为数字不如文字表意丰富。
- 每当添加、删除、或者调整了输出列的先后次序，都不得不重新检查ORDER BY子句以便对它们的位置编号作相应的更改。
- 在ORDER BY子句中引用列位置的语法，不再属于标准SQL的一部分，是不赞成使用的。

使用别名就不存在这种问题。

类似于 ORDER BY 子句的情况，如果你打算用 GROUP BY 子句对一个计算出来的输出列进行归类，可以使用输出列的别名或者它们在查询结果里的出现位置来设定。下面这个查询把不同月份出生的美国总统的人数分别统计了出来：

这个查询可以用输出列的出现位置改写如下：

```
mysql> SELECT MONTH(birth) AS Month, MONTHNAME(birth) AS Name,
    -> COUNT(*) AS count
    -> FROM president GROUP BY Name ORDER BY Month;
+-------+-----------+-------+
| Month | Name      | count |
+-------+-----------+-------+
|     1 | January   |     4 |
|     2 | February  |     4 |
|     3 | March     |     4 |
|     4 | April     |     4 |
|     5 | May       |     2 |
|     6 | June      |     1 |
|     7 | July      |     4 |
|     8 | August    |     4 |
|     9 | September |     1 |
|    10 | October   |     6 |
|    11 | November  |     5 |
|    12 | December  |     3 |
+-------+-----------+-------+
```

COUNT()函数还能与 ORDER BY 和 LIMIT 子句联合使用，例如，如果你想知道出生总统最多的前 4 个州是哪几个，可以对 president 数据表做如下查询：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC LIMIT 4;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
+-------+-------+
```

如果你不想用 LIMIT 子句来限制查询结果中的记录个数，而只是想把与某个特定 COUNT()值相对应的记录找出来，就需要使用一个 HAVING 子句。HAVING 子句与 WHERE 子句的相似之处是它们都是用来设定查询条件的，输出的行必须符合这些查询条件。HAVING 子句与 WHERE 子句的不同之处是 COUNT()之类的汇总函数的计算结果允许在 HAVING 子句里出现。请看下面这个查询，它能告诉我们说美国哪些州有两位或两位以上的总统出生：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state HAVING count > 1 ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
+-------+-------+
```

一般说来，带有 HAVING 子句的查询命令特别适合用来查找在某个数据列里重复出现的值（或者不重复出现的值——使用子句 HAVING count = 1 即可）。

除 COUNT()以外，MySQL 还有其他一些汇总函数。函数 MIN()、MAX()、SUM()和 AVG()能够得出某个数据列里的最小值、最大值、总和和平均值。MySQL 允许把它们同时用在同一个查询命令里。下面这个查询对学生们在各次考试或测验中的分数情况作了多种统计处理。结果显示各次考试中有考试成绩的总人数（不包括考试当天缺勤的学生）：

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS span,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id;
```

```
+----------+---------+---------+------+-------+---------+-------+
| event_id | minimum | maximum | span | total | average | count |
+----------+---------+---------+------+-------+---------+-------+
|        1 |       9 |      20 |   12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |   12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |   14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |   13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
+----------+---------+---------+------+-------+---------+-------+
```

很明显，如果还能知道 event_id 列的值代表的都是考试还是测验，这些信息的意义就更清晰明确了。我们可以做到这一点，但需要查询 grade_event 数据表，我们将在第 10 小节再次讨论这条查询命令。

如果你想输出“统计结果”，那就再增加一条 WITH ROLLUP 子句。这将使 MySQL 对数据行分组统计结果做进一步统计而得到所谓的“超级聚合”值。下面这个简单的例子是早前用来按性别统计学生人数的那条语句的基础上改写而来的，新增加的 WITH ROLLUP 子句将对两种性别的学生人数进行汇总并生成一个输出行：

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex WITH ROLLUP;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
| NULL|       31 |
+-----+----------+
```

在查询结果里，类别名称列（本例中是 sex 列）里的 NULL 值，表明与它同在一行的数值是它前面那些分组统计结果的汇总统计值。

WITH ROLLUP 子句还可以和其他聚合函数搭配使用。下面这条语句除了像几个段落之前那样对考试成绩进行了几种统计以外，还将生成一个额外的超级统计结果行：

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS span,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id
    -> WITH ROLLUP;
+----------+---------+---------+------+-------+---------+-------+
| event_id | minimum | maximum | span | total | average | count |
+----------+---------+---------+------+-------+---------+-------+
|        1 |       9 |      20 |   12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |   12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |   14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |   13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
```

```
|     NULL |       7 |     100 |   94 |  6376 | 36.8555 |   173 |
+----------+---------+---------+------+-------+---------+-------+
```

在上面这份输出结果里，最后一行显示的聚合值是根据它前面的所有分组统计结果值计算出来的。

WITH ROLLUP 子句的有用之处在于它能让你简单方便地获得一些额外的信息，如果不使用它，你恐怕要多进行一次查询才能达到同样的目的。只进行一次查询就能完成任务当然更有效率，因为这可以让 MySQL 服务器不必对数据进行两次甚至更多次的检索。如果 GROUP BY 子句指定了多列，WITH ROLLUP 会生成另外的包含高级统计值的超聚合行。

汇总函数功能很强大，用起来很有趣，但也正因如此，它们也很容易被滥用。请看这个查询：

```
mysql> SELECT
    -> state AS State,
    -> AVG(TIMESTAMPDIFF(YEAR, birth, death)) AS Age
    -> FROM president WHERE death IS NOT NULL
    -> GROUP BY state ORDER BY Age;
+-------+---------+
| State | Age     |
+-------+---------+
| KY    | 56.0000 |
| VT    | 58.5000 |
| NC    | 59.5000 |
| OH    | 62.2857 |
| NH    | 64.0000 |
| NY    | 69.0000 |
| NJ    | 71.0000 |
| TX    | 71.0000 |
| MA    | 72.0000 |
| VA    | 72.3750 |
| PA    | 77.0000 |
| SC    | 78.0000 |
| CA    | 81.0000 |
| MO    | 88.0000 |
| IA    | 90.0000 |
| NE    | 93.0000 |
| IL    | 93.0000 |
+-------+---------+
```

这个查询把所有已经逝世的总统都找了出来，按他们的出生地所在州进行分组，确定每个人逝世时的年龄，然后按各州求出他们逝世时的平均年龄，再按这个平均年龄排序后显示出来。换句话说，这个查询能让我们了解在同一个州出生的总统逝世时的平均年龄。

这又有什么意义呢？它能证明你有写出这类查询命令的能力，但并不能证明这个查询值得你去编写。数据库能让我们做很多事情，但并非每件事情都有意义。当人们发现自己能够利用数据库做很多事情时，他们往往会陷入一种为查询而查询的狂热。这种对统计数字漫无目标的热衷在最近几年的体育赛事电视转播中表示得尤其明显。利用他们的数据库，赛事统计人员能告诉你很多关于比赛你确实想知道的事情，但同时也能告诉你很多你根本就不想知道或者根本就没有想到过的事情。例如，你真的关心橄榄球队里哪个四分卫替补球员在他所属的球队领先对手 14 多分的情况下，在赛事第二节的最后两分钟里，在 15 码线区域内阻截对手达阵的次数最多吗？

10. 如何从多个数据表里检索信息

到目前为止，我们查询出来的信息都来自一个数据表。MySQL 的能力其实远不止此。前面说过，RDBMS（关系数据库管理系统）的威力在于它们能把一种东西与另一种东西关联起来，即能把来自多个数据表的信息结合在一起以解答单个数据表不足以解答的问题。本节将介绍涉及多个数据表的查询命令的编写方法。

当你打算从多个数据表选取信息时，有一种方法叫做*联结*（join）。这种叫法是因为必须把一个数据表与另一个数据表中的信息结合起来才能得到查询结果。联结操作是通过把两个（或多个）数据表里的同类数据进行匹配而完成的。多表操作的另一种方法就是将 SELECT 语句嵌套在另一个 SELECT 语句里，前者叫做*子查询*。本节将介绍这两种操作。

我们先来看一个关于联结的例子。在 1.4.6 节中第 2 小节里，我给出了一个用来检索给定日期考试或测验分数的查询命令，但我当时没有对它进行解释。现在是解释那条查询命令的时候了。那条查询命令实际上涉及 3 个数据表，即需要进行一次三方联结操作。现在，我们分两步来对它进行说明。首先，我们来构造一个能查出给定日期的分数的查询命令：

```
mysql> SELECT student_id, date, score, category
    -> FROM grade_event INNER JOIN score
    -> ON grade_event.event_id = score.event_id
    -> WHERE date = '2008-09-23';
+------------+------------+-------+----------+
| student_id | date       | score | category |
+------------+------------+-------+----------+
|          1 | 2008-09-23 |    15 | Q        |
|          2 | 2008-09-23 |    12 | Q        |
|          3 | 2008-09-23 |    11 | Q        |
|          5 | 2008-09-23 |    13 | Q        |
|          6 | 2008-09-23 |    18 | Q        |
...
```

这个查询先查出给定日期('2008-09-23')的 grade_event 行，再利用此行里的 event_id（考试事件编号）把 score 数据表里拥有同一 event_id 的考试分数都查出来。每找到一组彼此匹配的 grade_event 行和 score 行，就把学生的学号、考试分数、日期和考试事件的类型显示出来。

与以前介绍的查询命令相比，这个查询在以下几方面有着显著的区别。

- 在FROM子句里，我们列举了多个数据表的名字，因为我们要从多个数据表里检索信息：

  ```
  FROM grade_event INNER JOIN score
  ```

- 在ON子句里，我们给出了grade_event数据表和score数据表的联结条件：这两个数据表里的event_id列的值必须相互匹配：

  ```
  ON grade_event.event_id = score.event_id
  ```

请注意我在将 event_id 数据列命名为 grade_event、event_id 和 score.event_id 时所使用的 *tbl_name.col_name* 语法，这是为了让 MySQL 知道我们提到的数据表到底是哪几个。（因为两个数据表都有 event_id 数据列，所以如果不加上数据表的名字以区分，就会产生二义性。）但这个查询命令中的其他数据列（即 date、score、type）却用不着加上数据表的名字以进行区分，因为它们在不同的数据表里只出现一次，不可能产生二义性。

我个人喜欢在每个数据列的前面都加上数据表的名字以示区分。在今后涉及联结操作的查询示例

里，我将一直沿用这个习惯。在给每一个数据列都加上它们各自所属的数据表名字之后，这个查询将成为如下所示的样子：

```
SELECT score.student_id, grade_event.date, score.score, grade_event.category
FROM grade_event INNER JOIN score
ON grade_event.event_id = score.event_id
WHERE grade_event.date = '2008-09-23';
```

第一阶段的查询利用 grade_event 数据表把日期映射到一个考试事件编号，再利用这个考试事件编号把 score 数据表里与自己相匹配的考试分数找出来。这个查询的输出结果只能让我们看到 student_id 数据列的值，要是能把学生们的姓名直接列出来就更清晰易懂了。第二阶段，利用 student 数据表把学生们的学号映射为他们的姓名。score 和 student 数据表都有 student_id 数据列，两个数据表里的记录能够通过这个数据列被关联起来，利用这一事实就能把学生们的姓名也显示出来。如下所示：

```
mysql> SELECT
    -> student.name, grade_event.date, score.score, grade_event.category
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id
    -> WHERE grade_event.date = '2008-09-23';
+-----------+------------+-------+----------+
| name      | date       | score | category |
+-----------+------------+-------+----------+
| Megan     | 2008-09-23 |    15 | Q        |
| Joseph    | 2008-09-23 |    12 | Q        |
| Kyle      | 2008-09-23 |    11 | Q        |
| Abby      | 2008-09-23 |    13 | Q        |
| Nathan    | 2008-09-23 |    18 | Q        |
...
```

与以前介绍的查询命令相比，这个查询在以下几方面有着显著的区别。

- 在FROM子句里，除grade_event和score数据表外，我们又增加了student数据表。
- 在前一个查询里，student_id数据列不会产生二义性，所以我们当时既可以不给它加上数据表的名字（即写成student_id的形式），也可以给它加上数据表的名字（即写成score.student_id的形式）。但在这个查询里，因为score和student数据表都有student_id数据列，为了避免产生二义性，必须把它分别写成score.student_id和student.student_id以表明它们来自不同的数据表。
- ON子句里多了一个查询条件：score数据表里的行必须在student_id数据列上与student数据表里的行相匹配。

  ```
  ON ... score.student_id = student.student_id
  ```

- 查询结果将列出学生们的姓名而不是学号。（当然，如果你愿意，也可以把它们同时显示出来，只需要在输出列的列表加上student.student_id。）

只要对这个查询里的日期值进行替换，就能查出任何一天的考试分数、参加考试的学生名单以及考试的类型。你根本用不着知道学生学号或考试事件编号之类的东西，因为 MySQL 将自动查出有关的 ID 值并利用它们把你想要的信息找出来。

考试记分项目中的另一项工作是统计学生们的考试缺勤情况，并以学生学号和考试日期的形式记

录在 absence 数据表里。如果你想看到缺勤学生的姓名（而不仅仅是学号），就需要把 absence 表与 student 表通过 student_id 列的值联结起来。下面这个查询将列出缺勤学生的学号、姓名和他们的缺勤次数。

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student INNER JOIN absence
    -> ON student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-------+-----------+
| student_id | name  | absences |
+------------+-------+-----------+
|          3 | Kyle  |         1 |
|          5 | Abby  |         1 |
|         10 | Peter |         2 |
|         17 | Will  |         1 |
|         20 | Avery |         1 |
+------------+-------+-----------+
```

注意 在这个查询命令的 GROUP BY 子句里，数据列名字前面也加上了数据表的名字，但就这个查询而言，这并不是必要的。这是因为：GROUP BY 子句作用于输出列，而这里的查询结果只包含一个名为 student_id 的列，所以 MySQL 知道你指的是哪一个。

如果只想知道有哪些学生缺席，这个查询的结果将正是我们需要的。但如果你还要把这份名单交到学校办公室去，他们可能会问："其他学生的出勤情况呢？我们想知道每一名学生的出勤情况。"这是一个稍微不同的问题，他们想知道每一名学生（包括参加了考试的学生）的缺勤次数。但因为问法不同，回答这个问题使用的查询也就不同。

为了回答这个问题，我们可以用 LEFT JOIN 来代替普通的联结操作。LEFT JOIN 将使 MySQL 为联结操作中第一个数据表（即名称出现在关键字 LEFT JOIN 左边的那个数据表）里的每一个中选数据行生成一个输出行。只要先列出 student 数据表，我们就能得到全体学生（包括那些没有在 absence 表里出现过的学生）的考试出勤情况。这个查询的具体写法是这样的：在 FROM 子句里用关键字 LEFT JOIN（而不是逗号）来分隔各数据表的名字，再增加一个 ON 子句来指定两个数据表中的数据记录的匹配关系。如下所示：

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student LEFT JOIN absence
    -> ON student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-----------+-----------+
| student_id | name      | absences |
+------------+-----------+-----------+
|          1 | Megan     |         0 |
|          2 | Joseph    |         0 |
|          3 | Kyle      |         1 |
|          4 | Katie     |         0 |
|          5 | Abby      |         1 |
|          6 | Nathan    |         0 |
```

```
|          7 | Liesl     |        0 |
...
```

在前面的第 9 小节里，曾有一个对 score 数据表里的数据进行各种统计分析的查询。那个查询的输出结果列出了考试事件的编号，但因为我们当时还不知道如何联结 score 数据表与 grade_event 数据表才能把考试事件编号映射为考试的日期和类型，所以那份查询结果就没有包括考试的日期和类型。现在，我们知道应该怎样做了。下面这个查询与前面那个差不多，但当初那个简单的考试事件编号数字现在已经被取代为相应的日期和类型了：

```
mysql> SELECT
    -> grade_event.date,grade_event.category,
    -> MIN(score.score) AS minimum,
    -> MAX(score.score) AS maximum,
    -> MAX(score.score)-MIN(score.score)+1 AS span,
    -> SUM(score.score) AS total,
    -> AVG(score.score) AS average,
    -> COUNT(score.score) AS count
    -> FROM score INNER JOIN grade_event
    -> ON score.event_id = grade_event.event_id
    -> GROUP BY grade_event.date;
+------------+----------+---------+---------+------+-------+---------+-------+
| date       | category | minimum | maximum | span | total | average | count |
+------------+----------+---------+---------+------+-------+---------+-------+
| 2008-09-03 | Q        |       9 |      20 |   12 |   439 | 15.1379 |    29 |
| 2008-09-06 | Q        |       8 |      19 |   12 |   425 | 14.1667 |    30 |
| 2008-09-09 | T        |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
| 2008-09-16 | Q        |       7 |      20 |   14 |   379 | 14.0370 |    27 |
| 2008-09-23 | Q        |       8 |      20 |   13 |   383 | 14.1852 |    27 |
| 2008-10-01 | T        |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
+------------+----------+---------+---------+------+-------+---------+-------+
```

对涉及多个数据表的查询结果里的输出列也可以使用 COUNT()和 AVG()等汇总函数。请看下面这个查询，它将把与考试日期与考生性别的每一种组合相对应的考生人数（即考试分数的个数）和平均分数统计出来：

```
mysql> SELECT grade_event.date, student.sex,
    -> COUNT(score.score) AS count, AVG(score.score) AS average
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id
    -> GROUP BY grade_event.date, student.sex;
+------------+-----+-------+---------+
| date       | sex | count | average |
+------------+-----+-------+---------+
| 2008-09-03 | F   |    14 | 14.6429 |
| 2008-09-03 | M   |    15 | 15.6000 |
| 2008-09-06 | F   |    14 | 14.7143 |
| 2008-09-06 | M   |    16 | 13.6875 |
| 2008-09-09 | F   |    15 | 77.4000 |
| 2008-09-09 | M   |    16 | 79.0000 |
| 2008-09-16 | F   |    13 | 15.3077 |
| 2008-09-16 | M   |    14 | 12.8571 |
| 2008-09-23 | F   |    12 | 14.0833 |
```

```
| 2008-09-23 | M   |    15 | 14.2667 |
| 2008-10-01 | F   |    14 | 77.7857 |
| 2008-10-01 | M   |    15 | 82.4000 |
+------------+-----+-------+---------+
```

考试记分项目的另一项工作是计算每位学生的期末总成绩，这也可以用一个类似的查询来完成。下面就是完成这一工作的查询命令：

```
SELECT student.student_id, student.name,
SUM(score.score) AS total, COUNT(score.score) AS n
FROM grade_event INNER JOIN score INNER JOIN student
ON grade_event.event_id = score.event_id
AND score.student_id = student.student_id
GROUP BY score.student_id
ORDER BY total;
```

联结操作并非只能作用于不同的数据表，这乍听起来有点奇怪，但你确实能把某个数据表与它自身结合起来。例如，如果你想知道是否有两位（或者多位）总统出生于同一城市，就需要检查每位总统的出生地是否与其他总统的出生地一样。下面就是完成这一查询的命令：

```
mysql> SELECT p1.last_name, p1.first_name, p1.city, p1.state
    -> FROM president AS p1 INNER JOIN president AS p2
    -> ON p1.city = p2.city AND p1.state = p2.state
    -> WHERE (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY state, city, last_name;
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Adams     | John Quincy | Braintree | MA    |
| Adams     | John        | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

这个查询命令有两个地方特别值得注意，如下所示。

- 它需要两次用到同一个数据表，所以我们必须为它创建两个别名（p1和p2）才能把表中的同名数据列区别开来。有了列别名，在为表另定义别名时，AS关键字就是可选的。
- 每位总统的记录都将与其本身相匹配，但这并不是我们想要的输出结果。WHERE子句通过确保每位总统的记录只能与其他总统的记录比较，来剔除“记录与它本身相匹配”的情况。

用一个类似的查询可以查出在同一天出生的总统。可是，如果直接比较某两位总统的出生日期，就只能把同年同月同日出生的总统查出来，那些生日在同一天但出生年份不同的总统将不会出现在查询结果里。因此，我们必须用 MONTH() 和 DAYOFONTH() 函数比较出生日期值中的月份和日期，如下所示：

```
mysql> SELECT p1.last_name, p1.first_name, p1.birth
    -> FROM president AS p1 INNER JOIN president AS p2
    -> WHERE MONTH(p1.birth) = MONTH(p2.birth)
    -> AND DAYOFMONTH(p1.birth) = DAYOFMONTH(p2.birth)
    -> AND (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY p1.last_name;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Harding   | Warren G.  | 1865-11-02 |
```

```
| Polk       | James K.    | 1795-11-02 |
+------------+-------------+------------+
```

用 DAYOFYEAR()来代替 MONTH()与 DAYOFONTH()的组合会使我们得到一个稍微简单点儿的查询命令，但查询结果却可能是不正确的，因为没有考虑到闰年的因素。

进行多表检索的另一类办法是使用“子查询”，也就是把一条 SELECT 语句嵌套在另一条里。子查询有好几种类型，我们将在 2.9 节中进一步讨论。就目前而言，只要熟悉几个例子就可以了。我们不妨假设需要把出全勤的学生都找出来。这等价于把没在 absence 数据表里出现过的学生找出来，我们可以用如下所示的语句来达到目的：

```
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+------------+-----+------------+
| name       | sex | student_id |
+------------+-----+------------+
| Megan      | F   |          1 |
| Joseph     | M   |          2 |
| Katie      | F   |          4 |
| Nathan     | M   |          6 |
| Liesl      | F   |          7 |
...
```

嵌套于内层的 SELECT 语句用来生成一个在 absence 数据表里出现过的 student_id 值的集合，外层的 SELECT 语句负责把与该集合里的任何一个 ID 值都不匹配的 student 数据行检索出来。

利用子查询可以为 1.4.9 节的第 8 小节里提出的一个问题提供一个单语句解决方案，那个问题是哪些总统出生在 Andrew Jackson 之前。当初的解决方案使用了两条语句和一个用户变量，而我们现在可以用一个如下所示的子查询来解决它：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < (SELECT birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew');
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

内层 SELECT 语句用来确定 Andrew Jackson 的出生日期，外层 SELECT 语句负责检索生日早于该日期的总统。

1.4.10　如何删除或更新现有的数据行

有时候，你需要删除或修改一些现有的数据行，这就需要用到 DELETE 和 UPDATE 语句。本节的讨论重点就是这两条语句的用法。

DELETE 语句的基本格式如下所示：

```
DELETE FROM tbl_name
WHERE which rows to delete;
```

WHERE 子句是可选的，它指定哪些数据行会被删除掉；而如果没有 WHERE 子句，数据表里的全部行将都被删除掉。这意味着形式越简单的 DELETE 语句往往越危险，例如：

```
DELETE FROM tbl_name;
```

这条语句将把数据表里的内容删除得一干二净。请千万小心！如果只想删除满足部分数据记录，就必须用 WHERE 子句把它们先筛选出来。这与 SELECT 语句里用 WHERE 子句来避免选取整个数据表的做法类似。例如，如果想删除美国总统的记录，就应该使用下面这样的查询：

```
mysql> DELETE FROM president WHERE state='OH';
Query OK, 7 rows affected (0.00 sec)
```

如果不清楚某条 DELETE 语句到底会删掉哪些数据行，最好先把这条语句的 WHERE 子句放到一条 SELECT 语句里，看这条 SELECT 语句能查出哪些记录。这让你可以确认那些将被删除的行都是你想要删除的，既不多也不少。例如，如果想把 Teddy Roosevelt 总统的记录删掉，能不能使用下面这个查询呢？

```
DELETE FROM president WHRER last_name='Roosevelt';
```

这个语句确实会把 Teddy Roosevelt 总统的记录删掉，但它同时还会把 Franklin Roosevelt 总统的记录也删掉。因此，为保险起见，最好先把这个 WHERE 子句放到 SELECT 语句里检查一下，如下所示：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

从上面的查询结果可以看出，还需要对总统的名字做更细致的设定：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Roosevelt | Theodore   |
+-----------+------------+
```

现在，你应该知道要用什么样的 WHERE 子句才能选出打算删除的行了。下面是改正后的 DELETE 语句：

```
mysql> DELETE FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
```

不过，要是每删除一行都得这么办，那可就太麻烦了。但麻烦总比后悔好，对吧？如果遇到这类场合，可以通过复制加粘贴或者输入行编辑功能来尽可能地减少重复打字动作。1.5 节将对这方面的技巧进行介绍。

如果想修改现有行，就需要使用 UPDATE 语句，它的基本格式如下：

```
UPDATE tbl_name
```

```
SET which columns to change
WHERE which rows to update;
```

类似于 DELETE 语句中的情况，这里的 WHERE 子句也是可选的。如果没有给出 WHERE 子句，就表示该数据表里的每一条记录都需要修改。例如，下面这个查询将把每个学生的名字都改成 Georage：

```
mysql> UPDATE student SET name='George';
```

显然，必须谨慎对待这类查询，所以通常都加上一条 WHERE 子句来限制哪些行需要修改。我们来看一个“美国历史研究会”场景中的例子：研究会新吸收了一名会员，但在添加他的个人资料记录项时，只填写了几个数据列。

```
mysql> INSERT INTO member (last_name,first_name)
    -> VALUES('York','Jerome');
```

你发现自己忘了给他设置会员资格失效日期。可以用一条 UPDATE 语句来弥补，其中包含合适的 WHERE 子句来找到想要修改的行：

```
mysql> UPDATE member
    -> SET expiration='2009-7-20'
    -> WHERE last_name='York' AND first_name='Jerome';
```

可以用一条语句修改多个数据列。下面这条 UPDATE 语句将修改 Jerome 的电子邮件地址和普通邮政地址：

```
mysql> UPDATE member
    -> SET email='jeromey@aol.com', street='123 Elm St',
    -> city='Anytown', state='NY', zip='01003'
    -> WHERE last_name='York' AND first_name='Jerome';
```

如果某个数据列允许使用 NULL 值，可以把它设置为 NULL，从而使它处于“未设置”状态。例如，假如 Jerome 在日后一次性地交齐了足以让他成为终身会员的会费，就应该把他的会员资格失效日期设置为 NULL（意思是永不失效）：

```
mysql> UPDATE member
    -> SET expiration=NULL
    -> WHERE last_name='York' AND first_name='Jerome';
```

类似于 DELETE 语句，为了确保能不多不少地把你想要修改的记录全都筛选出来，最好先把 UPDATE 语句的 WHERE 子句放到一条 SELECT 语句里去检查。如果检索条件偏于严格或宽松，就会出现少修改或多修改了一些数据记录的情况。

如果你练习了本节里的查询命令，你的 sampdb 数据库里有关的数据表就会有一些记录删除或者修改掉了。在开始学习下一节之前，应该把那些改动都恢复到原状。如果真的需要重新加载那些数据表，请参考 1.4.8 节中的说明。

1.5　与客户程序 mysql 交互的技巧

本节将介绍一些与客户程序 mysql 交互的技巧，这些技巧能帮助我们更有效率地完成任务，让我们少打一些字。此外，还将学习到怎样更加方便迅速地连接服务器，怎样才能避免逐条输入查询命令，等等。

1.5.1 简化连接过程

在启动 mysql 程序时，通常都需要设定一些连接参数，如主机名、用户名或口令等。如果在每次启动 mysql 程序时都输入这么多的内容，你很快就会感到厌烦，也很容易打错字。其实，在连接 MySQL 服务器时，有好几种办法能减少必需的打字输入内容：

- 把连接参数保存在一个选项文件里；
- 利用shell的命令历史功能重复输入命令；
- 利用shell别名或脚本为mysql命令行定义一个快捷方式。

1. 使用一个选项文件

MySQL 允许把连接参数保存到一个选项文件（option file）里。这样，就用不着在每次启动 mysql 程序时都亲自输入这些参数了。系统将从选项文件里读入有关的参数，就好像已经在命令行上输入了它们一样。这样做的好处是其他 MySQL 客户程序（如 mysqlimport 或 mysqlshow）也能使用这些参数。换句话说，选项文件不仅能简化 mysql 程序的启动过程，也使很多其他程序的启动过程变得简单了。本节简要介绍如何设置选项文件以供客户程序使用，其他内容请参见 F.2.2 节。

在 Unix 系统上，你可以创建一个名为~/.my.cnf 的文件（即在登录主目录里创建一个名为.my.cnf 的文件）来作为你的选项文件。在 Windows 系统上，这个选项文件可以被创建为 MySQL 安装目录下的 my.ini 文件或者 C 盘根目录下的 C:\my.cnf 文件。这个选项文件是一个纯文本文件，所以可以用任何一种文本编辑器来创建。这个文件的内容应该是下面这个样子：

```
[client]
host=server_host
user=your_name
password=your_pass
```

其中，[client]是 MySQL 客户程序选项组的开始标记，由此往后直到文件结尾或下一个选项组开始标记的那些文本行，将逐一给出 MySQL 客户程序在启动时需要用到的选项值。请把其中的 server_host、your_name 和 your_pass 分别替换为你在连接 MySQL 服务器时使用的主机名、用户名和口令。例如，如果服务器在主机 cobra.snake.net 上运行，MySQL 用户名和口令分别是 sampadm 和 secret，下面就是.my.cnf 文件的内容：

```
[client]
host=cobra.snake.net
user=sampadm
password=secret
```

[client]是 MySQL 客户程序选项组的开始标记，它不能省略。但那些用来定义连接参数值的文本行却都是可选的，你可以只列举你需要的连接参数。例如，假如你使用的是 Unix 系统，而你的 MySQL 用户名又与你的 Unix 登录名一样，就不需要包括 user 那一行。默认主机是本地主机（localhost），你只打算连接到本地主机上的服务器，就不需要 host 那一行。

如果是在 Unix 系统上，那么在创建选项文件之后，还需要再多做一项工作：给这个选项文件设置访问权限，以保证别人不会读取或者修改它。下面两条命令都可以把这个文件的访问权限设置为只允许你本人访问：

```
% chmod 600 .my.cnf
% chmod u=rw,go-rwx .my.cnf
```

2. 利用shell的命令历史功能

有些 shell（如 tcsh 或 bash 等）能把你在命令行上输入过的命令记在一个命令历史清单里，你能查看并反复多次地选用其中的命令。如果 shell 具备这一功能，你就可以利用这份命令历史清单来避免敲入大段的命令内容。例如，如果最近使用过 mysql 客户程序，就可以像下面这样把它从命令历史清单里找出来并重新执行一遍：

```
% !my
```

感叹号字符的作用是：让 shell 从命令历史清单里把你最近输入过的、以 my 开头的命令找出来并重新执行一遍，就好像你在命令行上再次输入了这条命令一样。有些 shell 还可以用于利用键盘的上下箭头键（或 Ctrl-P、Ctrl-N 组合键）在命令历史清单里前后移动，当找到你想要执行的命令后，按下 Enter 键即可执行它。tcsh 和 bash 有这种功能，其他 shell 可能也有。你的 shell 是否具备命令历史功能以及该功能的具体使用方法可以在 shell 的帮助文档里查到。

3. 利用shell别名和脚本

如果 shell 允许使用别名机制，你就能把一个较短的命令映射为一条较长的命令。例如，如果使用的 shell 是 csh 或 tcsh，你就可以像下面这样用 alias 命令来定义出一个“新”的 sampdb 命令来，如下所示：

```
alias sampdb 'mysql -h cobra.snake.net -p -u sampadm sampdb'
```

如果使用的 shell 是 bash，定义语法将稍有不同：

```
alias sampdb='mysql -h cobra.snake.net -p -u sampadm sampdb'
```

完成别名定义工作之后，下面两条命令将完全等价：

```
% sampdb
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

很明显，第一个命令比第二个简短得多。如果想让这个别名在你每次登录系统时都能生效，就需要把 alias 命令放到 shell 的某个启动文件（如 tcsh 下的.tcshrc 文件，或者 bash 下的.bashrc 或.bash_profile 文件）里。

Windows 系统上也有类似的技巧：先为 mysql 程序创建一个快捷方式，再通过编辑该快捷方式属性把相关的连接参数包括进去。

还有一个办法能让你在调用命令时少打些字：创建一个脚本，用合适的选项执行 mysql。下面就是一个与刚才定义的命令别名 sampdb 等价的 shell 脚本（适用于 Unix 系统）：

```
#!/bin/sh
exec mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果把这个脚本命名为 sampdb 并（用 chmod +x sampdb 命令）设置为可执行，那么在命令提示符处敲入 sampdb 就能启动 mysql 程序并连接到 sampdb 数据库。

在 Windows 系统上可以利用批处理文件做同样的事情。创建一个名为 sampdb.bat 的批处理文件，再把下面这行文字输入其中：

```
mysql -h cobra.snake.net -p -u sampadm sampdb
```

此后，执行这个批处理文件的办法有两种：一是在控制台窗口敲入 sampdb，二是双击这个批处理文件的 Windows 图标。

如果需要访问多个数据库或连接到多个主机，不妨多定义几个别名、快捷方式或者脚本，让它们以不同的选项参数来启动 mysql 程序。

1.5.2 减少输入查询命令时的打字动作

从对数据库进行交互式查询的角度讲，mysql 是一个非常有用的程序，但它的操作界面却最适合用来输入短小的单行查询命令。虽说 mysql 本身并不关心我们输入的查询命令是否会延续好几行，但输入一条长长的查询命令却不是件会让人高兴的事。如果因为语法错误而不得不重新输入一遍，你应该会很郁闷。有好几种办法能帮助我们减少不必要的打字录入工作：

- 利用mysql的输入行编辑功能；
- 利用“复制+粘贴”来发出查询命令；
- 以批处理模式运行mysql程序。

1. 利用mysql的输入行编辑器

mysql 程序内建有 GNU Readline 库的输入行编辑功能。你可以编辑当前输入行的内容，也可以把以前的输入行调出来直接或经修改之后再次输入。当你在自己输入的命令行里发现了打字错误并想及时纠正时，这非常适用。在按 Enter 键之前，你可以把光标移到出错位置并改正那个打字错误。如果你在按下 Enter 键之后才发现有打字错误，可以把它调出来并在改正错误之后再次提交。（如果查询命令只有一行，改起来就更容易了。）

表 1-4 列出了一些有用的输入行编辑功能的按键序列，除了这些，还有很多常见的输入行编辑命令。你可以在 bash 使用手册介绍命令行编辑功能的有关章节里查到一份完整的清单，在线版 bash 使用手册可以在 GNU 项目的网站 http://www.gnu.org /manual/上找到。

表1-4 mysql程序的输入编辑命令

按键序列	含 义
上箭头键 或 Ctrl-P	调出前一个输入行
下箭头键 或 Ctrl-N	调出后一个输入行
左箭头键 或 Ctrl-B	向左移动光标
右箭头键 或 Ctrl-F	向右移动光标
Escape b	把光标向后移动一个单词
Escape f	把光标向前移动一个单词
Ctrl-A	把光标移到输入行的开头
Ctrl-E	把光标移到输入行的末尾
Ctrl-D	删除光标位置上的那个字符
Delete	删除光标前面（左侧）的那个字符
Escape D	删除单词
Escape Backspace	删除光标前面（左侧）的那个单词
Ctrl-K	删除从光标位置到输入行末尾的所有内容
Ctrl-_	取消前一次修改（可多次重复）

Readline 库没有适用于 Windows 平台的版本，所以在 Windows 平台上无法使用 Readline 库提供的编辑功能。幸好 Windows 本身支持表 1-5 里的编辑命令，因而在 mysql 工具的命令行上也可以使用它们。

表1-5　Windows平台上的输入编辑命令

按键序列	含　义
上箭头	调出前一行
下箭头	调出后一行
左箭头	光标左移一个字符（后退）
右箭头	光标右移一个字符（前进）
Ctrl + 左箭头	光标左移一个单词
Ctrl + 右箭头	光标右移一个单词
Home	光标移动到行首
End	光标移动到行尾
Delete	删除光标处的字符
Backspace（退格键）	删除光标左边的字符
Esc	删除整行
Page Up	调出最早输入的命令
Page Down	调出最后输入的命令
F3	调出最后输入的命令
F7	弹出命令菜单，用上箭头/下箭头键选择
F9	弹出命令菜单，用命令编号选择
F8，F5	循环显示命令列表

下面是输入行编辑功能一个简单的用法示例。假设你在 mysql 程序里输入了如下所示的查询命令：

```
mysql> SHOW COLUMNS FROM persident;
```

在按 Enter 键之前，如果你注意到自己把 president 错误地输成 persident 了，可以像下面这样修改查询。先按几次左箭头键把光标左移到字符 s 的位置上。按两次 Delete 或 Backspace 键，这两个键都可以删除你的系统上光标左侧的字符以删除 er，再重新输入 re 以改正错误。然后按下 Enter 键以提交修改后的查询命令。如果你没有在按下 Enter 键之前发现这个打字错误也不要紧。等看到 mysql 显示的出错信息后，按上箭头键调出刚刚输入的查询命令，然后再按上述过程修改就可以了。

2. 利用“复制+粘贴”来查询

如果你在一个窗口化的操作环境里工作，可以通过“复制+粘贴”操作把你认为有价值的查询命令保存到一个文件里供今后使用。整个操作步骤如下所示。

(1) 在一个终端窗口启动 mysql 程序。

(2) 在一个文档窗口里打开用来存放查询命令的文件（例如，在 Unix 系统上，我将使用 vi。在 Windows 系统上，我将使用 gvim）。

(3) 在文件里找到你想要执行的查询命令，选取并复制下来。再切换到终端窗口，把刚才复制下来的查询命令粘贴到 mysql 程序的提示符处。

这一过程看起来比较繁琐，但熟练掌握之后却相当快捷。它能让你不需打字，迅速地输入一条查询命令。

还可以把“复制+粘贴”操作反过来用（即把有关命令从终端窗口复制到你的查询命令存档文件里）。在 Unix 系统，当你在 mysql 程序里输入查询命令时，它们会被保存到你登录主目录里的一个名为.mysql_history 的文件里。如果你想把自己输入的某个查询命令保存起来供今后使用，可以这样

做：退出 mysql 程序，用一个文本编辑器打开.mysql_history 文件，找到这条查询命令并把它从.mysql_history 文件“复制+粘贴”到查询命令存档文件去。

3. 用mysql程序执行脚本文件

mysql 程序并非只能运行在交互模式下。mysql 程序能够以非交互（即批处理）模式运行并从文件里读取输入。如果你有一些需要定期运行的查询命令，这个技巧将特别有用，你再也用不着每次运行时都重新敲入它们了。只要把命令在文件中保存一次，就可以反复多次地让 mysql 程序根据需要执行它们了。

来看一个“美国历史研究会”场景中的查询示例。假设需要通过 member 数据表里的 interests 数据列来查找哪些会员对美国历史上的某个特定事件感兴趣。例如，为了了解哪些会员对 Great Depression（美国在 1930 年代的大萧条时期）感兴趣，可以编写下面这样的查询命令：

```
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%depression%'
ORDER BY last_name, first_name;
```

你把这个查询命令保存在 interests.sql 文件里，将文件馈入 mysql 程序里就可以运行它了：

```
% mysql  sampdb < interests.sql
```

在默认的情况下，以批处理模式运行的 mysql 程序的输出内容是以制表符来分隔的。如果想得到与你以交互方式运行 mysql 程序时的输出格式相同的整齐效果，就需要增加一个-t 选项，如下所示：

```
% mysql -t sampdb < interests.sql
```

如果想把输出结果保存起来，可以把它重定向到一个文件里，如下所示：

```
% mysql -t sampdb < interests.sql > interests.out
```

如果你已经在运行 mysql 了，可通过 source 命令执行文件内容：

```
mysql> source interests.sql
```

如果需要了解哪些会员对 Thomas Jefferson 总统的生平感兴趣，只需把 depression 改为 Jefferson 并再次运行 mysql 程序即可。不过，这个办法只有在你不需要非常频繁地查询时才显得有优势。如果必须频繁地运行某个查询命令，还需要找一个更好的办法。在 Unix 上，增加查询命令的灵活性的办法之一是把它保存为一个能够接受命令行参数的脚本，这个脚本将根据你给出的命令行参数对查询命令的具体内容作出修改，进而完成不同的查询任务。这样可以为查询确定参数，在运行脚本时你就可以指定 interests 值。以下面的 shell 脚本 interests.sh 为例：

```
#!/bin/sh
# interests.sh - find USHL members with particular interests
if [ $# -ne 1 ]; then echo 'Please specify one keyword'; exit; fi
mysql -t sampdb <<QUERY_INPUT
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%$1%'
ORDER BY last_name, first_name;
QUERY_INPUT
```

这个脚本程序的第 3 行确保命令行参数只有一个，否则，它就会打印一条简短的出错信息并退出执行。<<QUERY_INPUT 到脚本程序结尾处的 QUERY_INPUT 之间的文字将成为 mysql 程序的输入。shell 会把这段查询命令文本里的脚本变量$1 替换为你在命令行上给出的参数值（在脚本程序里，$1、$2

等变量依次对应该脚本的第 1 个、第 2 个命令行参数)。这样，运行这个脚本时，你在命令行上给出的参数值将成为查询命令中的检索关键字。

在运行这个脚本程序之前，还需要把它设置为可执行，如下所示：

```
% chmod +x interests.sh
```

现在，你再也用不着在每次运行这个脚本时都要先编辑了。只需通过命令行参数告诉它你想查找什么东西，就可以得到你想要的资料：

```
% ./interests.sh depression
% ./interests.sh Jefferson
```

可以在 sampdb 发行版本的 misc 子目录里找到这个 interests.sh 脚本，以及与之等价的 Windows 批处理文件 interests.bat。

说明　我强烈建议大家不要把这类脚本安装在共享区域里，因为它们不进行任何安全方面的检查，因而很容易遭到 SQL 注射攻击。万一有人用如下所示的命令行来调用脚本：

```
% ./interests.sh "Jefferson';DROP DATABASE sampdb;"
```

其后果将是把一条 DROP DATABASE 语句注射到脚本语句中成为 mysql 工具程序的输入，并真的会被执行。

1.6　后面各章的学习计划

通过本章的学习，相信大家对 MySQL 的使用方法已经有了一定的了解。你们应该掌握的技能包括：创建数据库和数据表，对数据表里的记录用多种方法插入、检索、修改、删除等操作。但本章只介绍了一些最浅显的概念，还有很多内容没有涉及。这一点可以从 sampdb 数据库的现状清楚地反映出来。我们创建了这个数据库和其中的数据表，还把一些原始数据填充到了数据表里。在学习过程中，我们还编写了一些查询命令，利用从数据库检索出来的信息解答了一些问题。但是，仍有很多事情在等着我们去做。例如，截止到目前，还没有一种简便的交互方式可以为考试记分项目插入新的考试分数记录和为“美国历史研究会”增加新的会员；不能方便地对现有数据记录进行修改；还没有生成“美国历史研究会”会员名录的打印版和在线版，等等。这些任务需要我们在今后各章（尤其是第 8 章和第 9 章）的学习过程中逐步完成。

如何开展后面的学习取决于读者对哪部分最感兴趣。如果你最想知道的是如何完成“美国历史研究会”和“考试记分项目”里的各项任务，本书的第二部分对 MySQL 应用程序的编写工作进行了讨论。如果你打算朝着 MySQL 数据库管理员的方向努力，本书的第三部分对管理工作进行了探讨。不过，我建议大家还是先按部就班地学完第一部分，多积累一些 MySQL 在使用方面的背景知识比较好。这些内容将帮助大家进一步了解 SQL 语句的语法和用法，明白 MySQL 怎样处理数据，怎样才能让查询执行得更快。对这些内容的扎实掌握将使你有能力胜任与 MySQL 有关的任何工作——无论是使用 mysql 程序编写自己的程序，还是作为一名称职的 MySQL 数据库管理员。

第 2 章

使用 SQL 管理数据

2

SQL（Structured Query Language，结构化查询语言）是 MySQL 服务器能够听懂的语言，是我们用来告诉 MySQL 服务器如何完成各种数据管理操作的手段。因此，要想有效地与 MySQL 服务器交流，就必须熟练掌握 SQL 语言。当你使用某个程序（如 `mysql` 客户工具）的时候，它在本质上只是一种能够让你把想要执行的 SQL 语句发送到服务器去的工具而已。如果你使用某种具备 MySQL 编程接口（如 Perl DBI 模块或 PHP PDO 扩展）的语言编写程序，你将能够通过发出 SQL 语句去与服务器进行交流。

第 1 章对 MySQL 的许多方面进行了简要的介绍，其中已经包括了 SQL 的一些基本用法。现在，我们将在这基础上从以下几个方面对 MySQL 所实现的 SQL 语言进行更详细的探讨：

- 改变SQL模式以影响MySQL服务器的行为；
- 各种数据库元素的命名规则；
- 使用多种字符集；
- 数据库、数据表和索引的创建和销毁；
- 获得关于数据库及其内容的信息；
- 使用联结、子查询和联合操作去检索数据；
- 创建视图以便从不同的角度去查看数据表里的数据；
- 多个数据表的删除和刷新操作；
- 利用事务处理机制一次性执行或撤销多条语句；
- 创建外键关系；
- 使用`FULLTEXT`搜索引擎。

上面列出的项目覆盖了 SQL 语言的众多应用领域。其他章节提供了更多与 SQL 相关的信息。

- 第4章讨论如何创建和使用存储函数（stored function）、存储过程（stored procedure）、触发器和事件。
- 第12章描述如何使用系统管理类语句如`GRANT`和`REVOKE`去管理用户账户。这一章还将讨论MySQL数据库的权限控制子系统，它控制着各个账户都允许执行哪些操作。
- 附录E列出了MySQL所实现的各种SQL语句的语法。它还讨论了在SQL语句中使用注释的语法。

你还可以参考《MySQL 参考手册》，它对了解 MySQL 最新版本中的更新非常有用。

2.1 MySQL 服务器的 SQL 模式

MySQL 服务器有一个名为 `sql_mode` 的系统变量可以让你调控其 SQL 模式，SQL 模式对 SQL 语

句的执行情况有多方面的影响。对这个变量可以作出全局性的设置，而各个客户可以通过改变这个模式来影响它们自己对 MySQL 服务器的连接。这意味着任何一个客户都可以在不影响其他客户的前提下改变 MySQL 服务器对自己的反应。

受 SQL 模式影响的行为包括在数据录入阶段如何处理非法数据、如何引用各种标识符，等等。下面的列表描述了几种可能的 SQL 模式设置值。

- STRICT_ALL_TABLES和STRICT_TRANS_TABLES将启用严格模式。在严格模式下，MySQL服务器在接受“坏”数据值方面将更加严格。（具体地说，它将拒绝“坏”数据值而不是把它们转换为最接近的合法值。）
- TRADITIONAL是另一种复合模式。它类似于严格模式，但启用了其他几种引入额外限制条件的模式以进行更加严格的数据检查。TRADITIONAL模式将导致MySQL服务器在处理“坏”数据值时更接近于传统的SQL服务器。
- ANSI_QUOTES告诉MySQL服务器把双引号识别为一个标识符引用字符。
- PIPES_AS_CONCAT将导致“||”字符串被视为一个标准的SQL字符串合并操作符，而不是“OR”操作符的一个同义词。
- ANSI是一种复合模式。它将同时启用ANSI_QUOTES、PIPES_AS_CONCAT和另外几种模式值，其结果是让MySQL服务器的行为比它的默认运行状态更接近于标准的SQL。

3.3 节将集中讨论会对数据录入环节中的错误或缺失值的处理行为产生影响的 SQL 模式值。附录 D 将对 sql_mode 变量的可取模式值做全面的描述。

在设置 SQL 模式时，需要给出一个由单个模式值或多个以逗号分隔的模式值构成的值；或者，给出一个空字符串以清除该值。模式值不区分字母的大小写。

如果想在启动 MySQL 服务器的时候设置 SQL 模式，可以在 mysqld 命令行上或是在某个选项文件里使用 sql_mode 选项。在命令行上，可以使用一个如下所示的设置项：

```
--sql-mode="TRADITIONAL"
--sql-mode="ANSI_QUOTES,PIPES_AS_CONCAT"
```

如果想在运行时改变 SQL 模式，可以使用一条 SET 语句来设置 sql_mode 系统变量。

任何一个客户都可以给它自己设置一个本次会话专用的 SQL 模式：

```
SET sql_mode = 'TRADITIONAL';
```

如果需要对 SQL 模式作全局性设置，需要加上 GLOBAL 关键字：

```
SET GLOBAL sql_mode = 'TRADITIONAL';
```

设置全局变量需要具备 SUPER 管理权限，新设置值将成为此后连接的所有客户的默认 SQL 模式。

如果想知道会话级或全局级 SQL 模式的当前值，可以使用如下所示的语句：

```
SELECT @@SESSION.sql_mode;
SELECT @@GLOBAL.sql_mode;
```

上述语句的返回值是由当前启用的所有模式以逗号分隔而构成的一列值。如果当前没有启用任何模式，则返回一个空值。

关于用户权限和设置/查看系统变量的其他信息，请参阅第 12 章。

2.2　MySQL 标识符语法和命名规则

几乎所有的 SQL 语句都需要以某种方式使用标识符来引用某个数据库或数据库所容纳的元素，

如数据表、视图、数据列、索引、存储例程、触发器或事件。在引用数据库的元素时，标识符必须遵守以下规则。

标识符里的合法字符。不加引号的标识符必须由系统字符集（utf8）中的字母和数字字符，再加上“_”和“$”字符构成。标识符的第一个字符可以是允许用在标识符里的任何一种字符，包括数字。不过，不加引号的标识符不允许完全由数字字符构成，因为那会使它与数值难以区分。MySQL 允许标识符以数字字符开头的做法在种类繁多的数据库系统中是不常见的。如果打算使用一个这样的标识符，必须特别留意它是否还包含着一个“E”或“e”字符，因为那样的组合很容易导致表达式出现歧义。比如说，表达式“23e + 14”（“+”号两边有空格）意味着给数据列 23e 加上 14，可是“23e + 14”又该如何解释呢？它是意味着同样的值，还是一个用科学计数法表示的数值呢？

标识符可以用反引号字符（“`”）括起来（加以界定），这意味着允许使用任意字符，只有取值为 0 或 255 的单字节例外：

```
CREATE TABLE `my table` (`my-int-column` INT);
```

当标识符是一个 SQL 保留字或者包含空格或其他特殊字符的时候，给它加上引号非常实用。给标识符加上引号让它可以完全由数字字符构成，这对不加引号的标识符来说是不允许的。如果想在一个加上引号的标识符里使用一个标识符引号字符，双写它即可。

在 MySQL 5.1.6 版之前，用于数据库和数据表的标识符还必须遵守另外两个限制条件，即使它们已经加上了引号。其一，不允许使用“.”字符，因为该字符在 `db_name.tbl_name` 或 `db_name.tbl_name.col_name` 格式的名称里被用作分隔符。其二，不允许使用 Unix 或 Windows 的路径名分隔字符（即“/”或“\”）。之所以不允许在数据库和数据表标识符里使用路径名分隔符，是因为数据库在硬盘上被表示为子目录，数据表在硬盘上被表示为至少一个文件。既然如此，这类标识符就只能由允许用在子目录名和文件名里的合法字符构成。不允许把 Unix 路径名分隔符用在 Windows 平台上（反之亦然），是为了让在运行于不同平台上的 MySQL 服务器之间迁移数据库和数据表的工作更容易。（如果允许人们在 Windows 平台上的数据表名字里使用斜线字符，就无法把它迁移到 Unix 平台上了，因为后一种平台上的文件名不允许包含斜线字符。）

从 MySQL 5.1.6 版开始，把 SQL 语句里的标识符映射为目录名和文件名的机制经过了修改，使得早期版本里的部分非法字符也可以用在标识符里。具体地说，只需给标识符加上引号，在其中使用路径名字符（“/”或“\”）和“.”字符就是合法的了。

你的操作系统可能会对数据库和数据表标识符有额外的要求，请参阅 11.2.6 节。

数据列和数据表名字的假名可以相当随意。如果打算使用的假名是一个 SQL 保留字、完全由数字构成，或者包含空格或其他特殊字符，就应该用标识符引号字符把它括起来。数据列假名还可以使用单引号或双引号。

MySQL 服务器的 SQL 模式。如果启用了 `ANSI_QUOTES` SQL 模式，可以用双引号来括住标识符（反引号仍允许使用）。

```
CREATE TABLE "my table" ("my-int-column" INT);
```

注意 启用 `ANSI_QUOTES` 还有额外效果——字符串值必须用单引号写出。如果使用了双引号，MySQL 服务器将把该值解释为标识符而不是字符串。

内建函数的名字一般来说都不是保留字，可以不加引号地用作标识符。不过，如果启用了IGNORE_SPACE SQL 模式，函数名将被视为保留字，还想使用它们作为标识符就必须给它们加上引号。

设置 SQL 模式的具体步骤请参阅 2.1 节。

标识符的长度。绝大多数标识符的最大长度是 64 个字符。假名的最大长度是 256 个字符。

标识符限定符。根据上下文，标识符可能需要加以限定，以明确它到底对应着什么。如果想指称一个数据库，把它的名字写出来即可：

```
USE db_name;
SHOW TABLES FROM db_name;
```

如果想指称一个数据表，有两种选择。

- 使用完整的数据表名，它由一个数据库标识符和一个数据表标识符构成：

```
SHOW COLUMNS FROM db_name.tbl_name;
SELECT * FROM db_name.tbl_name;
```

- 一个数据表标识符本身对应着默认（当前）数据库里的一个数据表。如果sampdb是默认数据库，下面的语句是等效的：

```
SELECT * FROM member;
SELECT * FROM sampdb.member;
```

如果没有选定数据库，就不能在没有给出数据库限定符的情况下引用某个数据表，因为这个数据表到底属于哪个数据库是不明确的。

对数据表名加以限定的考虑同样适用于视图（它们是“虚拟的”数据表）和存储程序的名字。

如果想指称一个数据列，有 3 种选择。

- 使用完整的数据列名，如*db_name.tbl_name.col_name*。
- 对于默认数据库里某给定数据表里的一个数据列，可以使用*tbl_name.col_name*形式的部分限定名。
- 只写出一个非完整名*col_name*表示该数据列属于上下文环境所确定的那个数据表。下面两个查询使用了相同的数据列名，但每条语句的FROM子句所提供的上下文表明了应该到哪一个数据表去选择这些数据列：

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

一般来说，没有必要提供完整的名字，但如果你愿意，那永远都是合法的。如果用一条 USE 语句选定了一个数据库，该数据库就将成为默认数据库并隐含在你此后引用的每一个不完整的数据表名字里。如果在一条 SELECT 语句里只引用了一个数据表，该数据表隐含在这条语句所包含的每一个数据列名字里。只有在无法根据上下文确定数据表或数据库的时候才必须使用完整的标识符。比如说，如果一条语句涉及多个数据库里的数据表，所有不在默认数据库里的数据表就必须以 *db_name.tbl_name* 的形式来给出，这样才能让 MySQL 知道哪个数据库包含着哪个数据表。同样的道理，如果某个查询涉及多个数据表并引用了一个在多个数据表里用到的数据列名字，就必须用一个数据表标识符对该数据列标识符进行限定，以明确打算使用的是哪一个数据列。

如果打算在引用一个完整名字时使用引号，就应该给该名字里的每一个标识符分别加上引号。如下所示：

```
SELECT * FROM `sampdb`.`member` WHERE `sampdb`.`member`.`member_id` > 100;
```

不要把这样的名字作为一个整体而只加上一组引号。下面这条语句是不正确的：

```
SELECT * FROM `sampdb.member` WHERE `sampdb.member.member_id` > 100;
```

在把某个保留字用作标识符时，必须给它加上引号，但在这个保留字紧跟在一个句号限定符后面时例外，因为已经可以根据上下文而确定这个保留字其实是一个标识符。

2.3 SQL 语句中的字母大小写问题

SQL 语句中的字母大小写规则随语句元素的不同而变化，同时还要取决于你正引用的事物和 MySQL 服务器主机上的操作系统。

SQL 关键字和函数名。关键字和函数名不区分字母的大小写。它们可以为任意的字母大小写组合。下面的语句将检索出同样的信息（但输出结果中的数据列标题将是不同的字母大小写组合）：

```
SELECT NOW();
select now();
sElEcT nOw();
```

数据库、数据表和视图的名字。在服务器主机上，MySQL 数据库和数据表用底层文件系统中的目录和文件表示。因而数据库和数据表名字的默认字母大小写情况将取决于服务器主机上的操作系统在文件名方面的规定。Windows 文件名不区分字母的大小写，所以运行在 Windows 主机上的 MySQL 服务器也就不区分数据库和数据表名字的字母大小写。运行在 Unix 主机上的 MySQL 服务器往往要区分数据库和数据表名字的字母大小写，因为 Unix 文件系统是区分字母大小写的。Mac OS X 平台上的 HFS+文件系统中的名字是个例外，不区分字母的大小写。

MySQL 使用一个文件来表示一个视图，所以刚才与数据表有关的讨论也同样适用于视图。

存储程序的名字。存储函数、存储过程和事件的名字不区分字母的大小写。触发器的名字要区分字母的大小写，这一点和标准的 SQL 行为是不一样的。

数据列和索引的名字。数据列和索引的名字在 MySQL 环境里不区分字母的大小写。下面的语句将检索出同样的信息：

```
SELECT name FROM student;
SELECT NAME FROM student;
SELECT nAmE FROM student;
```

假名的名字。在默认的情况下，数据表假名区分字母的大小写。可以使用任意的字母大小写组合（大写、小写或大小写混用）来给出一个假名，但如果需要在同一条语句里多次用到同一个假名，就必须让它们保持同样的字母大小写组合。如果 `lower_case_table_names` 变量是非零值，数据表假名将不区分字母的大小写。

字符串值。字符串值是否区分字母大小写，这取决于它是二进制还是非二进制，而非二进制字符串还要取决于字符集的排序方式。这对字符串常数值和字符串数据列的内容都不例外。关于这方面的更多信息请参阅 3.1.2 节。

当你在一台区分文件名字母大小写的机器上创建数据库和数据表时，应该这样思考字母大小写的问题：日后有没有可能需要把它们迁移到一台不区分文件名字母大小写的机器上去？假设你已经在一台 Unix 服务器上创建了两个名字分别是 `abc` 和 `ABC` 的数据表——这两个名字在这台服务器上是区别对待的，当你想把这两个数据表迁移到一台 Windows 机器上时就会遇到麻烦。因为新机器不区分字母的大小写，`abc` 和 `ABC` 将无法区别对待。在把数据表从一台 Unix 主服务器复制到一台 Windows 从服

务器时也会遇到麻烦。

避免字母大小写问题演变成棘手难题的办法之一，是选定一种字母大小写方案，一直遵照该方案去创建数据库和数据表。这样一来，等你日后想把某个数据库迁移到不同的服务器上时，名字的大小写问题就不存在了。我的建议是统一使用小写字母。这在你使用 InnoDB 数据表时也有益处，因为 InnoDB 引擎在其内部是把数据库和数据表的名字保存为小写字母的。

如果想统一使用小写字母来创建数据库和数据表的名字——就算没在 CREATE 语句里特意设定也能如此，可以通过设置 lower_case_table_name 系统变量来配置服务器。更多信息请参阅 11.2.6 节。

不管你的系统是否区分数据库或数据表名字的字母大小写，你在给定的查询里必须使用一致的大小写组合来引用它。SQL 关键字、函数名、数据列名和索引名不必如此拘泥，因为 MySQL 允许在查询命令里使用任意的字母大小写组合。不过，如果能坚持使用统一的风格而不是随意混搭的话，查询命令会有更好的可读性。

2.4　字符集支持

MySQL 不仅支持多种字符集，还允许对服务器、数据库、数据表、数据列或字符串常数级别的字符集作出互不影响的设定。比如说，如果想让某个数据表的数据列默认使用 latin1 字符集，但同时还包含一个 Hebrew 数据列和一个 Greek 数据列，你完全可以那样做。此外，还可以明确地设定排序方式。有哪些字符集和排序方式可供选择是可以查出来的，把数据从一种字符集转换为另一种也有章可循。

本节提供了关于 MySQL 的字符集支持的基本背景知识。第 3 章将更细致地讨论字符集、排序方式、二进制字符串和非二进制字符串以及如何定义和使用基于字符的数据表列。第 12 章将讨论如何对 MySQL 服务器支持的字符集进行配置。

MySQL 的字符集支持机制提供了以下一些功能。

- MySQL 服务器允许同时使用多种字符集。
- 一种给定的字符集可以有一种或多种排序方式。你可以为应用程序挑选一种最适用的排序方式。
- Unicode 支持由 utf8 和 ucs2 字符集提供，从 MySQL 6.0.4 版开始有更多的字符集可供选用。
- 你可以在服务器、数据库、数据表、数据列和字符串常数等级别设定字符集：
  - 服务器有一个默认的字符集。
  - CREATE DATABASE 语句可以用来设定数据库级字符集，ALTER DATABASE 语句可以改变之。
  - CREATE TABLE 和 ALTER TABLE 语句有专门用来设定数据表级和数据列级的子句（详见第 3 章）。
  - 用于字符串常数的字符集既可以通过上下文设定，也可以明确设定。
- 既有用来转换数据值的字符集的函数和操作符，也有用来判断数据值的字符集的函数和操作符。类似地，COLLATE 操作符可以用来改变某个字符串的排序方式，而 COLLATE() 函数将返回某给定字符串的排序方式。
- SHOW 语句和 INFORMATION_SCHEMA 数据表提供了关于可用字符集和排序方式的信息。
- 当你改变某个带索引的字符类型的数据列时，MySQL 服务器将自动地对索引进行重新排序。

在同一个字符串内不能混用不同的字符集，一个给定的数据列不能在不同的数据行使用不同的字

符集。不过，可以选用一种 Unicode 字符集（用单一编码方案表示多种语言的字符）去实现你期望的多语言支持。

2.4.1 字符集的设定

字符集和排序方式可以在多个级别进行设定，从 MySQL 服务器使用的默认字符集到用于个别字符串的字符集。

服务器的默认字符集和排序方式是在当初编译时内建的，但我们可以在启动服务器时使用 `--character-set-server` 和 `--collation-server` 选项，或是在服务器启动后设置 `character-set-server` 和 `collation-server` 系统变量来覆盖它们。如果只选定了字符集，它的默认排序方式就将成为服务器的默认排序方式。如果想选定一种排序方式，必须让它与字符集保持兼容。（判断排序方式与字符集是否兼容的办法，是看它的名字是否以字符集的名字开头。比如说，`utf8_danish_ci` 排序方式与 `utf8` 字符集相兼容、与 `latin1` 字符集不兼容。）

在创建数据库和数据表的 SQL 语句里，有两个子句专门用来设定数据库、数据表和数据列级别的字符集和排序方式：

```
CHARACTER SET charset
COLLATE collation
```

`CHARSET` 可以用作 `CHARACTER SET` 的同义词。*charset* 是服务器所支持的字符集的名字。而 `collation` 是该字符集的一种排序方式的名字。这些子句可以同时使用，也可以分开使用。在同时使用这两个子句的时候，必须让排序方式的名字与字符集保持兼容。如果只给出了 `CHARACTER SET` 子句，则意味着使用默认排序方式。如果只给出了 `COLLATE` 子句，则使用由给定排序方式的名字的开头部分确定的字符集。这些规则适用于以下几个级别。

- 如果想在创建数据库时为它设定一个默认的字符集和排序方式，需要使用如下所示的语句：

  ```
  CREATE DATABASE db_name CHARACTER SET charset COLLATE collation;
  ```

  如果没有对字符集或排序方式作出设定，服务器级别的默认设置将传递给这个数据库。
- 如果想为某个数据表设定默认的字符集和排序方式，可以在创建该数据表时利用`CHARACTER SET`和`COLLATION`数据表选项：

  ```
  CREATE TABLE tbl_name (...) CHARACTER SET charset COLLATE collation;
  ```

  如果没有对字符集或排序方式作出设定，数据库级别的默认设置将传递给这个数据表。
- 对于数据表里的某个数据列，可以使用`CHARACTER SET`和`COLLATION`属性为它指定一个字符集和排序方式。如下所示：

  ```
  c CHAR(10) CHARACTER SET charset COLLATE collation
  ```

  如果没有对字符集或排序方式作出设定，数据表级别的默认设置将传递给这个数据列。这些属性适用于 `CHAR`、`VARCHAR`、`TEXT`、`ENUM` 和 `SET` 数据类型。

还可以利用 `COLLATE` 操作符按照特定排序方式对字符串值排序。比如说，假设 `c` 是一个使用 `latin1` 字符集、`latin1_swedish_ci` 排序方式的数据列，但你想按照 `Spanish` 排序规则对它排序，可以这么做：

```
SELECT c FROM t ORDER BY c COLLATE latin1_spanish_ci;
```

2.4.2　确定可供选用的字符集和当前设置

如果想知道有哪些字符集和排序方式可供选用，可用使用下面这些语句：

```
SHOW CHARACTER SET;
SHOW COLLATION;
```

这两条语句都支持使用一个 LIKE 子句，把查询结果缩窄到名字与给定模式相匹配的那些字符集或排序方式。比如说，下面这条语句将只列出基于拉丁语的字符集：

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | cp1252 West European        | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

下面这条语句将只列出与 utf8 字符集相兼容的排序方式（排序方式的名字总是以字符集的名字开头）：

```
mysql> SHOW COLLATION LIKE 'utf8%';
+--------------------+---------+-----+---------+----------+---------+
| Collation          | Charset | Id  | Default | Compiled | Sortlen |
+--------------------+---------+-----+---------+----------+---------+
| utf8_general_ci    | utf8    |  33 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |  83 |         | Yes      |       1 |
| utf8_unicode_ci    | utf8    | 192 |         | Yes      |       8 |
| utf8_icelandic_ci  | utf8    | 193 |         | Yes      |       8 |
| utf8_latvian_ci    | utf8    | 194 |         | Yes      |       8 |
| utf8_romanian_ci   | utf8    | 195 |         | Yes      |       8 |
| utf8_slovenian_ci  | utf8    | 196 |         | Yes      |       8 |
...
```

从上面这些语句的输出结果里可以看出，每一种字符集最少拥有一种排序方式，并且有一种是它的默认排序方式。

关于可用字符集和排序方式的信息，还可以从 INFORMATION_SCHEMA 数据库中的 CHARACTER_SETS 和 COLLATIONS 数据表查到（请参阅 2.7 节）。

如果想显示 MySQL 服务器的当前字符集和排序方式的设置情况，可以使用 SHOW VARIABLES 语句：

```
mysql> SHOW VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | latin1 |
| character_set_connection | latin1 |
| character_set_database   | latin1 |
| character_set_filesystem | binary |
| character_set_results    | latin1 |
| character_set_server     | latin1 |
```

```
| character_set_system     | utf8   |
+--------------------------+--------+
mysql> SHOW VARIABLES LIKE 'collation\_%';
+----------------------+--------------------+
| Variable_name        | Value              |
+----------------------+--------------------+
| collation_connection | latin1_swedish_ci |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+--------------------+
```

这些系统变量中的某几个会对客户在与 MySQL 服务器建立连接后的通信情况产生影响。这方面的细节请参阅 3.1.2 节的第 2 小节。

2.4.3 Unicode 支持

之所以会有这么多种字符集，原因之一是人们为不同的人类语言制定了不同的字符编码方案。这就导致了几个问题。比如说，如果某给定字符在好几种人类语言里都存在，它在不同的编码方案里就有可能是用不同的数值来表示的。还有，不同的人类语言往往需要使用数目不同的字节去表示一个字符。latin1 字符集足够小，每个字符只需使用一个字节来表示，但有些语言（如日语和汉语）因为包含非常多的字符，它们需要使用多个字节来表示每个字符。

Unicode 的目标是提供一个统一的字符编码方案，让所有人类语言的字符集都能以一种统一的方式表示。

1. MySQL 6.0版之前的Unicode支持

在 MySQL 6.0.4 版之前，其 Unicode 支持仅包括 Basic Multilingual Plane（BMP，初级多语言方案）里的字符，最多只有 65 536 个字符。没被收录到 BMP 方案里的其他字符是没有任何支持的。Unicode 通过两种字符集提供了一个比较完善的解决方案。

- ucs2字符集对应着Unicode UCS-2编码方案。它使用两个字节来表示一个字符，高位字节排列在前。这种字符集无法表示需要用两个以上的字节才能表示的字符。UCS是Universal Character Set（通用字符集）的缩写。
- utf8字符集采用了一种长度可变的格式，使用一到三个字节来表示一个字符。它对应着UTF-8编码方案。UTF是Unicode Transformation Format（统一编码转换格式）的缩写。

2. MySQL 6.0版之后的Unicode支持

从 MySQL 6.0.4 版开始，其 Unicode 支持把 BMP 方案所遗漏的补充字符也收录了进来，这么做的效果如下所示。

- ucs2字符集在MySQL 6.0系列版本里未做改动，每个字符仍占两个字节。新增加的utf16和utf32字符集类似于utf8，但扩充了对补充字符的支持。在utf16字符集里，BMP字符仍占两个字节（和usc2字符集一样），补充字符占四个字节。在utf32字符集里，所有字符都占四个字节。
- 以前，每个utf8字符占一到三个字节。增加了补充字符之后，每个utf8字符占一到四个字节。
- 由MySQL 6.0以前的版本创建的、使用utf8字符集的数据库和数据表在MySQL 6.0系列版本里将按照utf8mb3字符集来显示。（比如说，如果使用SHOW CREATE TABLE语句去查看的话，你将看到utf8mb3。）除了名字本身的差异，MySQL 6.0系列版本中的utf8mb3字符集和6.0系列

之前的utf8字符集完全一样。

如果想把数据表从老 utf8 字符集（3 字节）转化为新 utf8（4 字节），可以在升级到 MySQL 6.0 以前先用 mysqldump 工具把数据表备份下来，等升级完后再重新加载导出文件。在升级完成以后，千万不要忘记运行 mysql_upgrade 工具以确保 mysql 数据库里的系统级数据表全都打好了补丁。

2.5　数据库的选定、创建、删除和变更

MySQL 提供了几个数据库级的语句：USE 用来选定一个默认数据库，CREATE DATABASE 用来创建数据库，DROP DATABASE 用来删除数据库，ALTER DATABASE 用来改变数据库的全局特性。

在后一种情况的语句中，关键字 SCHEMA 是 DATABASE 的同义词。

2.5.1　数据库的选定

USE 语句选定一个数据库并把它当做指定 MySQL 服务器连接上的默认（当前）数据库：

```
USE db_name;
```

要想选定一个数据库，就必须具备相应的访问权限，要不然无法选定它。

显式选定数据库不是必要的。如果你确实有访问该数据库的权限，那么即使你没有选择数据库，只要用数据库名字来限定数据表名字，你就可以使用其中的数据表。比如说，如果你没有事先选定 sampdb 数据库，却想检索其中 president 数据表里的内容，可以使用如下所示的查询语句：

```
SELECT * FROM sampdb.president;
```

不过，通常不带数据库限定词的数据表名字用起来要更方便一些。

选定默认数据库并不意味着它将在连接持续期间内一直是默认数据库。只要具备足够的访问权限，你可以多次使用 USE 语句在多个数据库之间任意切换。同时，选定一个数据库也并不意味着只能使用这个数据库里的数据表。即使已经把某个数据库选定为当前的默认数据库，也可以利用数据库标识符通过名字去访问其他数据库里的数据表。

当与服务器的连接终止时，该服务器上的默认数据库概念也就不复存在了。换句话说，当你再次连接上该服务器时，它并不会记得你上一次选定的默认数据库。

2.5.2　数据库的创建

要创建一个数据库，需要使用 CREATE DATABASE 语句：

```
CREATE DATABASE db_name;
```

执行数据库创建操作的先决条件是：数据库名字必须是合法的，这个数据库不能是已经存在的，你必须有足够的权限去创建它。

创建数据库时，MySQL 服务器会在它的数据目录里创建一个与该数据库同名的子目录，这个新目录称为数据库子目录。服务器还会在那个数据库目录里创建一个 db.opt 文件来保存数据库的属性。

CREATE DATABASE 语句有好几种可选的子句。它的完整语法如下所示：

```
CREATE DATABASE [IF NOT EXISTS] db_name
  [CHARACTER SET charset] [COLLATE collation];
```

在正常情况下，当你试图创建的数据库已经存在时，系统将报告出错。如果想避免这类错误，只在给定数据库尚不存在的前提下才创建它，请加上一条 IF NOT EXISTS 子句：

```
CREATE DATABASE IF NOT EXISTS db_name;
```

在默认情况下，服务器级别的字符集和排序方式将成为新建数据库的默认字符集和排序方式。可以使用 CHARACTER SET 和 COLLATE 子句对这些数据库属性作出明确的设置。如下所示：

```
CREATE DATABASE mydb CHARACTER SET utf8 COLLATE utf8_icelandic_ci;
```

如果只给出了 CHARACTER SET 子句而没有 COLLATE 子句，则意味着使用给定字符集的默认排序方式。如果只给出了 COLLATE 子句而没有 CHARACTER SET 子句，则意味着使用排序方式的名字的开头部分确定的字符集。

字符集必须是服务器所支持的，如 latin1 或 sjis。排序方式必须是给定字符集的一个合法的排序方式。关于字符集和排序方式的进一步讨论请参阅第 3 章。

MySQL 把数据库的字符集和排序方式等属性保存在相应的 db.opt 文件里。在创建新数据表时，如果你没有在新数据表的定义里为它指定一种默认的字符集和排序方式，数据库级别的默认设置就将成为新数据表的默认设置。

如果想查看现有数据库的定义，可以使用一条 SHOW CREATE DATABASE 语句：

```
mysql> SHOW CREATE DATABASE mydb\G
*************************** 1. row ***************************
       Database: mydb
Create Database: CREATE DATABASE `mydb`
                 /*!40100 DEFAULT CHARACTER SET utf8
                 COLLATE utf8_icelandic_ci */
```

2.5.3 数据库的删除

只要你有足够的权限，删除一个数据库和创建一个数据库同样简单，使用如下语句即可：

```
DROP DATABASE db_name;
```

注意，千万不要随意使用 DROP DATABASE 语句，这条语句将会删掉数据库和其中的所有东西，包括数据表、存储例程等，这个数据库也就永远消失了，除非定期地对数据库进行备份。

一个数据库就是 MySQL 数据目录里的一个子目录，这个子目录用于存放数据表视图和触发器等。如果 DROP DATABASE 语句失效，通常是因为那个数据库子目录里还包含有一些与数据库对象无关的文件。DROP DATABASE 语句不会删除这类文件，因而也就不删除那个数据库子目录。这就意味着如果写了 SHOW DATABASES 语句，尽管里面已经没有数据表了，可那个数据库子目录却依然存在。在这种情况下，如果真想删除那个数据库，就必须手动删除该数据库子目录里遗留的文件和子目录本身，然后再发出 DROP DATABASE 语句。

2.5.4 数据库的变更

使用 ALTER DATABASE 语句可以改变数据库的全局特性。就目前而言，数据库的全局特性还只有默认字符集和排序规则：

```
ALTER DATABASE [db_name] [CHARACTER SET charset] [COLLATE collation];
```

2.6　数据表的创建、删除、索引和变更

MySQL 允许创建、删除数据表或改变其结构，相应的 SQL 语句分别是 CREATE TABLE、DROP TABLE 和 ALTER TABLE。CREATE INDEX 和 DROP INDEX 语句用来给现有的数据表增加或删除索引。以下几节将详细解释这些语句，但我认为应该先讨论一下 MySQL 为管理不同类型的数据表而支持的几种存储引擎。

2.6.1　存储引擎的特征

MySQL 支持好几种*存储引擎*（storage engine，它们以前被称为“数据表处理器”）。由同一个存储引擎所实现的数据表具有一些共同的属性或特征。表 2-1 简要地描述了 MySQL 发行版本目前支持的几种存储引擎，稍后将讨论各存储引擎的功能细节。在 MySQL 5.0 和更高的版本里，表中除 Falcon 以外的所有存储引擎都可以直接使用，Falcon 存储引擎只能在 MySQL 6.0 里使用。

表 2-1　MySQL 支持的存储引擎

存储引擎	说　明
ARCHIVE	用于数据存档的引擎（数据行被插入后就不能再修改了）
BLACKHOLE	这种存储引擎的写操作是删除数据，读操作是返回空白记录
CSV	这种存储引擎在存储数据时以逗号作为数据项之间的分隔符
EXAMPLE	示例（存根）存储引擎
Falcon	用来进行事务处理的存储引擎
FEDERATED	用来访问远程数据表的存储引擎
InnoDB	具备外键支持功能的事务处理引擎
MEMORY	内存里的数据表
MERGE	用来管理由多个 MyISAM 数据表构成的数据表集合
MyISAM	默认的存储引擎
NDB	MySQL Cluster 专用存储引擎

有几种存储引擎还有同义名称，MERG_MyISAM 和 NDBCLUSTER 分别是 MERGE 和 NDB 的同义名称。MEMORY 和 InnoDB 存储引擎最早分别叫做 HEAP 和 Innobase，后者现在仍允许使用，但已不再建议使用。

在 MySQL 5.1 和更高的版本里，服务器采用了一种“可插入”的体系结构，它提供一套标准的接口用来在运行时动态地加载和卸载存储引擎。因此，由第三方开发的存储引擎可以被方便地集成到服务器里。

1. 查看有哪些存储引擎可供选用

给定一个服务器，真正使用哪几种存储引擎将取决于你的 MySQL 版本、在编译该服务器时使用的具体配置、启动该服务器时使用的选项，等等。配置和启动存储引擎的具体步骤参见 12.7 小节。

用 SHOW ENGINES 语句可以查出服务器都知道哪些存储引擎。该语句提供的信息可以帮助我们回答诸如“有哪些支持事务处理的存储引擎可供选用”之类的问题。以下输出内容使用的是 MySQL 5.0 版的格式：

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
 Engine: MyISAM
Support: DEFAULT
Comment: Default engine as of MySQL 3.23 with great performance
*************************** 2. row ***************************
 Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
*************************** 3. row ***************************
 Engine: InnoDB
Support: YES
Comment: Supports transactions, row-level locking, and foreign keys
...
```

Support 栏里的 YES 或 NO 代表该存储引擎是否可用，DISABLED 的意思是该存储引擎可用但它已被关闭，DEFAULT 表示这是服务器默认使用的存储引擎。一般来说，DEFAULT 存储引擎应该是可用的。

MySQL 5.1 版里的 SHOW ENGINES 语句要比 5.0 系列版本多几个与事务处理有关的数据列：

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
...
*************************** 8. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

Transaction 栏里的值表明存储引擎是否支持事务。XA 和 Savepoints 栏里的值表明某种存储引擎是否支持分布式事务处理（本书未讨论）和部分事务回滚。

在 MySQL 5.1 和更高的版本里，有一个名为 ENGINES 的 INFORMATION_SCHEMA 数据表，它提供的信息与 SHOW ENGINES 语句完全一样。你可以像下面这样使用该数据表查看有哪些支持事务处理的存储引擎可供选用（以下输出来自 MySQL 6.0，所以其中包括了 Falcon 存储引擎）：

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE TRANSACTIONS = 'YES';
+--------+
| ENGINE |
+--------+
| Falcon |
| InnoDB |
+--------+
```

2. 数据表在硬盘上的存储方式

你每创建一个数据表，MySQL 就会创建一个硬盘文件来保存该数据表的格式（也就是它的定义）。这个格式文件的基本名和数据表的名字一样，扩展名是.frm。比如说，如果数据表的名字是 t，其格式文件的名字就将是 t.frm。你创建的数据表属于哪个数据库，服务器就会在该数据库的数据库子目录里创建这个文件。.frm 文件的内容是不变的，不管是哪一个存储引擎在管理数据表，每个数据表也只有一个相应的.frm 文件。如果数据表的名字字符在文件名里会引起麻烦，SQL 语句里使用的数据表的名字有可能与相应的.frm 文件的基本名（表名）不一致。从 SQL 名字到文件名的映射规则参见 11.2.6 节。

具体到某个特定的存储引擎，它还会为数据表再创建几个特定的文件以存储其内容。对于给定的数据表，与之相关的所有文件都集中存放在这个数据表所在的数据库的数据库子目录里。表 2-2 列出了几种存储引擎为特定数据表创建的文件的扩展名。

表2-2　由存储引擎创建的数据表文件

存储引擎	硬盘上的文件
MyISAM	.MYD（数据）、.MYI（索引）
MERGE	.MRG（由各成员MyISAM数据表的名字构成的清单）
InnoDB	.ibd（数据和索引）
ARCHIVE	.ARZ（数据）、.ARM（元数据）
CSV	.CSV（数据）、.CSM（元数据）

对某些存储引擎而言，格式文件是与某特定数据表相关联的唯一文件。其他存储引擎会把数据表的内容保存到硬盘上的其他地方，或者使用一个或多个表空间（tablespace，由多个数据表所共享的存储区域），如下所示。

- MEMORY 数据表存放在内存里，不占用任何硬盘空间。
- 在默认的情况下，InnoDB 引擎会把数据表的数据和索引存储在它的共享表空间里。也就是说，所有 InnoDB 数据表的内容都集中保存在一个共享存储空间里，而不是与某个特定的数据表相关联的文件里。InnoDB 引擎只在你特意配置它来为每个数据表分别创建一个表空间时才会去创建.ibd 文件。
- Falcon 引擎把数据表的数据和索引保存在表空间文件里。有一个默认的 Falcon 表空间，但你也可以根据自己的需要创建其他表空间。这些表空间中的任何一个都可以容纳多个数据表的内容。
- BLACKHOLE 和 EXAMPLE 存储引擎实际上不存储任何数据，所以它们不需要创建任何文件。
- FEDERATED 引擎用于访问某远程 MySQL 服务器上的数据表，它本身不创建任何文件。
- 在接下来的几节里，我们将有选择地介绍几种 MySQL 存储引擎在功能和行为方面的特点。如果你想知道各存储引擎怎样以物理方式保存数据表，请参阅 11.2.3 节。

3. MyISAM存储引擎

MyISAM 存储引擎是 MySQL 默认使用的存储引擎，如果你没有把你的服务器配置成其他样子的话。下面是它的部分功能。

- MyISAM 存储引擎提供了键压缩功能。它使用某种压缩算法来保存连续的、相似的字符串索引值。此外，MyISAM 存储引擎还可以压缩相似的数值索引值，因为数值都是按照高位字节优先的办法来保存的。（低位字节的索引值的检索速度非常快，所以高位字节很容易压缩。）

如果你想激活数值压缩功能，请在创建 MyISAM 数据表时使用 PACK_KEYS=1 选项。

- 与其他存储引擎相比，MyISAM 存储引擎为 AUTO_INCREATMENT 数据列提供了更多的功能。关于这方面的详情请参见 3.4 节。
- 每个 MyISAM 数据表都有一个标志，服务器或 myisamchk 程序在检查 MyISAM 数据表时会对这个标志进行设置。MyISAM 数据表还有一个标志用来表明该数据表在上次使用后是不是被正常地关闭了。如果服务器意外宕机或机器崩溃，这个标志可以用来判断数据表是否需要检查和修复。如果你想让这种检查自动进行，需要在启动服务器时使用 --myisam-recover 选项。这会让服务器在每次打开一个 MyISAM 数据表时自动检查该数据表的标志并进行必要的数据表修复处理。
- MyISAM 存储引擎支持全文检索，但这需要通过 FULLTEXT 索引来实现。
- MyISAM 支持空间数据类型和 SPATIAL 索引。

4. MERGE存储引擎

MERGE 数据表提供了一种把多个 MyISAM 数据表合并为一个逻辑单元的手段。查询一个 MERGE 数据表相当于查询其所有的成员数据表。这种安排的好处之一是可以绕开文件系统对各个 MyISAM 数据表的最大长度的限制。

用来构成 MERGE 数据表的所有数据表必须有同样的结构。这意味着必须为各成员数据表里的数据列定义同样的名字、同样的类型和同样的顺序，索引也必须以同样的办法按同样的顺序定义。我们可以把经过压缩和未经过压缩的数据表混杂在一起而构成一个 MERGE 数据表（myisamchk 程序可以用来创建压缩数据表，请参见附录 F）。

2.6.2 节中的第 5 小节给出了一个例子。另外，分区数据表可以作为除 MERGE 数据表以外的另一种选择，而且其成员不限于 MyISAM 数据表。请参阅 2.6.2 节中的第 6 小节。

5. MEMORY存储引擎

MEMORY 存储引擎把数据表保存在内存里，这些数据表有着长度固定不变的数据行，这两个特点使得数据表的检索速度非常之快。

从某种意义上讲，MEMORY 数据表是临时性的。当服务器掉电时，表中的内容也就消失了——MEMORY 数据表在服务器重启之后仍会存在，只是它们的内容将是一片空白。MERORY 数据表的另一个特点是其内容对其他客户来说是可见的，这与用 CREATE TEMPORARY TABLE 语句创建出来的临时数据表形成了对照。

MERORY 数据表的如下特点使它们比其他类型的数据表更容易处理，所以检索速度非常快。

- 在默认的情况下，MERORY 数据表使用散列索引，利用这种索引进行“相等比较”的速度非常快，但进行“范围比较”的速度就慢多了。因此，散列索引只适合用在使用“=”和“<=>”操作符进行的比较操作里，不适合用在使用“<”或“>”操作符进行的比较操作里。出于同样的考虑，散列索引也不适合用在 ORDER BY 子句里。
- 存储在 MERORY 数据表里的数据行使用的是长度固定不变的格式，以此加快处理速度，这意味着你不能使用 BLOB 和 TEXT 这样的长度可变的数据类型。VARCHAR 是一种长度可变的类型，但因为它在 MySQL 内部被当做一种长度固定不变的 CHAR 类型，所以你可以在 MERORY 数据表里使用 VARCHAR 类型。

如果确实需要使用 MERORY 数据表和“<”、“>”或 BETWEEN 操作符进行某种比较以判断某个值是否落在某个范围内，可以使用 BTREE 索引来加快速度（请参阅 2.6.4 节中的第 2 小节）。

6. InnoDB存储引擎

InnoDB 存储引擎最早是由 Innobase Oy 公司开发的，该公司后来被 Oracle 公司收购。InnoDB 存储引擎有以下几种功能。

- 支持提交和回滚操作，确保数据在事务处理过程中万无一失。还可以通过创建保存点（savepoint）的办法来实现部分回滚（partial rollback）。
- 在系统崩溃后可以自动恢复。
- 外键和引用完整性支持，包括递归式删除和更新。
- 数据行级别的锁定和多版本共存，这使得 InnoDB 数据表在需要同时进行检索和更新操作的复杂查询里表现出非常好的并发性能。
- 在默认的情况下，InnoDB 存储引擎会把数据表集中存储在一个共享的表空间里，而不是像大多数其他存储引擎那样为不同的数据表创建不同的文件。InnoDB 表空间可以由多个文件构成，还可以包括多个原始分区。实际上，InnoDB 表空间就像是一个虚拟的文件系统，它存储和管理所有 InnoDB 数据表的内容。这样一来，数据表的长度就可以超过文件系统对各个文件的最大长度的限制。你也可以把 InnoDB 存储引擎配置成会为每个数据表分别创建一个表空间的样子，此时，每个数据表在它的数据库子目录里都有一个对应的.ibd 文件。

7. Falcon存储引擎

Falcon 存储引擎是从 MySQL 6.0 开始才有的，它有以下功能。

- 支持提交和回滚操作，确保数据在事务处理过程中万无一失。还可以通过创建保存点来实现部分回滚。
- 在系统崩溃后可以自动恢复。
- 灵活的锁定级别和多版本共存，这使得 Falcon 数据表在需要同时进行检索和更新操作的复杂查询里表现出非常好的并发性能。
- 在存储时对数据行进行压缩，在检索时对数据行进行解压缩以节省空间。
- 日常管理和维护方面的开销低。

8. FEDREATED存储引擎

FEDREATED 存储引擎的用途是访问其他 MySQL 服务器管理下的数据表。换句话说，FEDREATED 数据表的内容不是存放在本地主机里的。当你创建一个 FEDREATED 数据表时，需要指定一台运行着其他服务器程序的主机，并提供那个服务器的合法账户的用户名和口令。当你打算访问 FEDREATED 数据表时，本地服务器将使用这个账户连接那台远程服务器。2.6.2 节中的第 7 小节给出了一个这样的例子。

9. NDB存储引擎

NDB 是 MySQL 的集群（cluster）存储引擎。在这个存储引擎工作时，MySQL 服务器的作用是帮助其他进程访问 NDB 数据表，从整个集群的高度看，其行为像是一个客户。各集群结点上的进程通过彼此通信来管理内存中的数据表。为了减少冗余，数据表在集群进程中被复制。内存存储提供了高性能，集群机制提供了高度可用性，即使个别结点发生故障，系统也不会崩溃。

NDB 存储引擎的配置和使用超出了本书的讨论范围，这里就不再多说了。有兴趣的读者请自行研读《MySQL 参考手册》。

10. 其他存储引擎

MySQL 还有几种存储引擎是前面没有提到的，如下所示。

- ARCHIVE 存储引擎对数据进行归档。它最适合用来大批量地保存那些“写了就不改”的数据行。因此，它只支持很有限的几条 SQL 语句。INSERT 和 SELECT 语句没问题，但 REPLACE 语句的行为却永远像是 INSERT 语句，而 DELETE 或 UPDATE 语句根本不能用。为了节省空间，在存储时会对数据行进行压缩，在检索时再对它们进行解压缩。在 MySQL 5.1.6 之前的版本里，ARCHIVE 存储引擎根本不支持索引；即使是在 MySQL 5.1.6 版本里，每个 ARCHIVE 数据表最多也只能有一个带索引的 AUTO_INCREMENT 数据列；其他数据列还是不能带索引。
- BLACKHOLE 存储引擎创建的数据表有这样的行为特点：写操作其实是删除数据，而读操作是返回空白记录。
- CSV 存储引擎在存储数据时以逗号作为数据项之间的分隔符。它会在数据库子目录里为每个数据表创建一个.CSV 文件。这是一种普通文本文件，每个数据行占用一个文本行。CSV 存储引擎不支持索引。
- EXAMPLE 存储引擎是用来演示如何编写存储引擎的最小化样板。它的存在价值在于让开发人员通过查看其源代码去学习怎样才能正确地把存储引擎加载到服务器里。

11. 存储引擎的可移植性

从某种意义上讲，任何一个 MySQL 服务器所管理的任何数据表都可以移植到另一台服务器上去：先用 mysqldump 工具把它备份出来，然后把备份文本文件放到另一台服务器主机，并通过加载备份文件的办法重新创建该数据表。可移植性概念还有另一层含义，即*二进制可移植性*（binary portablity），指的是你可以直接把代表某个数据表的硬盘文件复制到另一台机器，并把它们安装到数据子目录下的相应地点，然后那台机器上的 MySQL 服务器就可以使用该数据表了。

数据表具备二进制可移植性的一项基本条件是源服务器和目标服务器的有关功能必须兼容。比如说，目标服务器必须支持用来管理数据表的存储引擎。如果目标服务器上没有适用的存储引擎，它将无法访问你在源服务器上使用那种存储引擎创建的数据表。

有些存储引擎创建的数据表具备良好的二进制可移植性，有些存储引擎则不然。下面是对各个存储引擎的二进制可移植性的总结。

- MyISAM 和 InnoDB 数据表的存储格式与机器无关，它们具备二进制可移植性——前提条件是你的处理器使用的是二进制补码整数算法和 IEEE 浮点格式。一般来说，只要使用的机器不是古怪少见的品牌，这两个前提条件就不会构成真正的问题。在实践中，只要使用的不是为某种专用设备而定制的嵌入式服务器，就不太容易因为硬件因素而遇到可移植性问题，这是因为专用设备所使用的处理器往往会有一些非标准的特性。
- MERGE 数据表的可移植性取决于其成员 MyISAM 数据表，只要那些 MyISAM 数据表是可移植的，MERGE 数据表就是可移植的。
- MEMORY 数据表不具备二进制可移植性，因为它们的内容都存储在内存里而不是硬盘上。
- CSV 数据表是二进制可移植的，因为.CSV 文件在本质上都是普通的文本文件。
- BLACKHOLE 数据表是二进制可移植的，因为它们根本不包含任何内容。
- 对于 FEDERATED 存储引擎，可移植性概念与之无关，因为 FEDERATED 数据表的内容都存储在另一个服务器上。
- Falcon 日志和表空间文件的存储格式与机器有关，它们只在两台机器的硬件特性完全相同时才是二进制可移植的。比如说，不能把一台低字节优先的机器里的 Falcon 文件移植到一台高字节优先的机器。

我们刚刚提到，在两台机器之间移植 MyISAM 和 InnoDB 数据表时，在二进制可移植性方面需要满足的前提条件是：将被移植的数据表不包含任何浮点数据列，或者两台机器使用的浮点数存储格式是一样的。这里所说的“浮点数”指的是 FLOAT 和 DOUBLE 类型，不包括 DECIMAL 类型。DECIMAL 数据列里的数据值的小数点位置是固定的，这种存储格式是可移植的。

对 InnoDB 存储引擎而言，二进制可移植性还有另外一个条件：数据库和数据表的名字应该由小写字母构成。InnoDB 存储引擎在其数据字典里把这些名字统一保存为小写字母格式，但.frm 文件名里的字母却与你在 CREATE TABLE 语句里使用的大小写情况完全一样。如果当初在创建数据库或数据表时使用的是大写字母，而现在你想把它们移植到一个对文件名区分大小写的平台上去，就有可能因为字母的大小写情况不匹配而遇到问题。

对 InnoDB 存储引擎而言，必须把所有 InnoDB 数据表作为一个整体来评估二进制可移植性，而不是只考虑某一个或某几个特定的 InnoDB 数据表。在默认的情况下，InnoDB 存储引擎会把由它负责管理的所有数据表集中存储在一个共享的表空间里，而不是为各数据表分别创建一个文件。因此，不应该只考虑某一个或某几个 InnoDB 数据表是不是可移植的，必须考虑 InnoDB 表空间文件是不是可移植的。这意味着关于浮点数的可移植性规则必须针对所有包含浮点数的 InnoDB 数据表进行考虑。要知道，即使把 InnoDB 存储引擎配置成把每个数据表分别存储在一个表空间里的样子，它也会把其数据字典里的数据项集中保存在那个共享的表空间里。

无论存储引擎的可移植性怎样，都要注意：不要在服务器关闭之后把数据表或表空间文件复制到另一台机器上，除非你能确定服务器的关机操作没有任何问题。这是因为，如果服务器意外关闭，在它关闭后得到的副本将无法确保数据的完整性不受影响。你打算复制的数据表有可能需要修复，也有可能还有一些事务信息仍保存在存储引擎的日志文件里，必须等它们被提交或回滚之后才能更新数据表。

在某些情况下，可以让一个运行中的服务器在我们复制数据表文件时不要使用有关的数据表。但一般而言，只要服务器正在运行并在刷新数据表，或者还有一些改动仍缓存在内存里，硬盘上的数据表内容就仍有可能发生变化，而数据表副本就有可能没有任何实用价值。如果想知道需要满足哪些条件才能在无须关闭服务器的前提下复制数据表，请参阅 14.2 节。

2.6.2　创建数据表

创建数据表需要用到 CREATE TABLE 语句。这条语句的完整语法相当复杂，因为它的可选子句实在是太多了。还好，在实际工作中用到的绝大多数 CREATE TABLE 语句都比较简单。比如说，第 1 章用到的绝大多数 CREATE TABLE 语句都算不上复杂。只要从最基本的语法形式开始循序渐进，就应该不会遇到太大的麻烦。

最简单的 CREATE TABLE 语句只需你给出一个数据表的名字和其中数据列的名单。比如说：

```
CREATE TABLE mytbl
(
  name   CHAR(20),
  birth  DATE NOT NULL,
  weight INT,
  sex    ENUM('F','M')
);
```

在创建数据表时，除了各数据列的定义，还可以指定如何为它创建索引。另一种做法是先创建一

个不带任何索引的数据表，过后再给它加上索引。对 MyISAM 数据表来说，如果在开始对它查询之前需要填充大量的数据，第二种策略往往更好。与先把数据加载到一个不带任何索引的 MyISAM 数据表里之后再去创建索引的做法相比，每插入一个数据行都刷新一次索引要慢得多。

我们在第 1 章已经对 CREATE TABLE 语句的基本语法进行了介绍。定义数据列的细节见第 3 章。在这一小节，我们将重点介绍 CREATE TABLE 语句的几种重要的变体，它们可以帮助你灵活地构造数据表，如下所示。

- 改变存储特性的数据表选项。
- 只在数据表不存在时才创建。
- 临时数据表，服务器会在客户会话结束时自动删除它们。
- 从另一个数据表或是从一次 SELECT 查询的结果来创建数据表。
- 使用 MERGE 数据表、分区数据表、FEDERATED 数据表。

1. 数据表选项

要想改变某个数据表的存储特性，在 CREATE TABLE 语句中的右括号之后加上一个或多个数据表选项即可。数据表选项的完整清单可以在附录 E 里对 CREATE TABLE 语句的描述里查到。

有一个数据表选项是 ENGINE = *engine_name*，它用来指定用哪种存储引擎来管理将要创建的数据表。比如说，如果想创建一个 MEMORY 或 InnoDB 数据表，就要写出如下所示的语句：

```
CREATE TABLE mytbl ( ... ) ENGINE = MEMORY;
CREATE TABLE mytbl ( ... ) ENGINE = InnoDB;
```

存储引擎的名字不区分字母的大小写。如果没有给出 ENGINE 选项，服务器将使用默认的存储引擎来创建数据表。内建的默认存储引擎是 MyISAM，但你可以通过使用--default-storage-engine 选项来启动服务器，使用另外一种默认的存储引擎。在服务器运行期间，还可以通过设置系统选项 storage-engine 改变默认的存储引擎。

在 MySQL 5.0 里，一个运行中的服务器可以同时使用的存储引擎是有限的，有几种存储引擎总是可用，另外几种就要由你来挑选和配置了。如果在一条 CREATE TABLE 语句里给出了一个服务器能够支持但在此时此刻不可用的存储引擎的名字，MySQL 将使用默认的存储引擎去创建那个数据表并生成一条警告信息。比如说，如果 ARCHIVE 存储引擎是服务器能够支持但在此时此刻不可用的，你在创建一个 ARCHIVE 数据表时就会看到如下所示的消息：

```
mysql> CREATE TABLE t (i INT) ENGINE = ARCHIVE;
Query OK, 0 rows affected, 1 warning (0.01 sec)
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1266 | Using storage engine MyISAM for table 't' |
+---------+------+-----------------------------------------+
```

如果给出的存储引擎的名字是服务器不支持的，它将报告一条出错消息。

在 MySQL 5.1 和更高的版本里，服务器使用了一种插入式的体系结构，这使得服务器可以在运行时动态地加载存储引擎。“服务器可以使用的存储引擎”现在的含义应该是“服务器当前已加载的存储引擎”。如果在创建数据表时给出了一个此时此刻尚未加载的存储引擎的名字，服务器将生成两条警告消息：

```
mysql> CREATE TABLE t (i INT) ENGINE = ARCHIVE;
Query OK, 0 rows affected, 2 warnings (0.01 sec)
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1286 | Unknown table engine 'ARCHIVE'          |
| Warning | 1266 | Using storage engine MyISAM for table 't' |
+---------+------+-----------------------------------------+
```

要想让某个数据表使用特定的存储引擎，就一定要给出相应的 ENGINE 数据表选项。默认的存储引擎可以改变，所以省略 ENGINE 选项有可能导致你预期使用的默认存储引擎与实际不一样。此外，一定要保证 CREATE TABLE 语句没有生成任何警告消息，与这条语句有关的警告消息基本上都是“某某存储引擎不可用，使用了默认的存储引擎”。

如果不想让 MySQL 在指定的存储引擎不可用时使用默认的存储引擎代替之，需要激活 NO_ENGINE_SUBSTITUTION SQL 模式。

如果想知道数据表使用的是哪一种存储引擎，发出一条 SHOW CREATE TABLE 语句并查看其输出内容里的 ENGINE 选项即可：

```
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `i` int(11) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

存储引擎还可以通过 SHOW TABLE STATUS 语句或 INFORMATION_SCHEMA.TABLES 数据表查看。

有些数据表选项只适用于特定的存储引擎。比如说，在创建 MEMORY 数据表时加上一个 MIN_ROWS=*n* 选项可能会很有用，它可以让 MEMORY 存储引擎对内存的使用情况进行优化：

```
CREATE TABLE mytbl ( ... ) ENGINE = MEMORY MIN_ROWS = 10000;
```

如果 MEMORY 引擎认为 MIN_ROWS 的值足够大，它在分配内存时就会使用大内存块以避免因为需要发出太多的内存分配调用而增加总体开销。

MAX_ROWS 和 AVG_ROW_LENGTH 选项可以帮你在某种程度上控制 MyISAM 数据表的大小。在默认的情况下，MyISAM 存储引擎在创建数据表时使用的内部数据行指针的长度允许数据表文件增长到 256 TB。如果你给出了 MAX_ROWS 和 AVG_ROW_LENGTH 选项，MyISAM 存储引擎就会根据这些信息为数据表选用一个适当的内部数据行指针长度，确保它至少能够容纳 MAX_ROWS 个数据行。

如果想改变现有的数据表的存储特性，在 ALTER TABLE 语句里写出相关的数据表选项即可。比如说，如果想把 mytbl 数据表现在使用的存储引擎改成 InnoDB，发出下面这条语句即可：

```
ALTER TABLE mytbl ENGINE = InnoDB;
```

如果想了解更多关于如何改变存储引擎的信息，请参阅 2.6.5 节。

2. 只创建原本没有的数据表

如果你只想创建原本没有的数据表，请使用 CREATE TABLE IF NOT EXIST 语句。这条语句可以让你的应用程序无须假设它需要用到的数据表是否已经存在。你的应用程序将去尝试创建那个数据表，无论它是否已经存在都不会影响应用程序的正常执行。IF NOT EXIST 短语在你打算通过 mysql

工具去执行的批处理脚本里非常有用。对于这种情况，普通的 CREATE TABLE 语句存在一个小问题：在你第一次运行批处理脚本时，它将正常地创建出那个数据表，但在第二次运行它的时候就会出错，因为那个数据表已经存在了。如果你使用了 IF NOT EXIST 短语，就不会发生这样的问题了：在第一次运行批处理脚本时，它将正常地创建出那个数据表。在第二次和以后运行这个脚本时，尝试创建那个数据表的操作将被毫无声息地忽略，不会产生任何错误，而你的批处理任务将继续往下执行，就像这次尝试取得了成功那样。

如果打算使用 IF NOT EXIST 短语，有个细节必须注意：MySQL 不会去比较 CREATE TABLE 语句里的数据表的结构与已经存在的那个数据表是否一致。即使已经存在一个名字相同、但结构不同的数据表，这条语句也不会报告出错，而你后面的操作可就会乱套了。如果不想冒这个险，可以先执行一条 DROP TABLE IF EXIST 语句，再执行一条不带 IF NOT EXIST 短语的 CREATE TABLE 语句。

3. 临时数据表

如果在数据表创建语句里加上 TEMPORARY 关键字，服务器将创建出一个临时的数据表，它在你与服务器的连接断开时自动消失：

```
CREATE TEMPORARY TABLE tbl_name ... ;
```

这么做的好处是你不必惦记还得发出一条 DROP TABLE 语句来删除那个数据表，即使你与服务器的连接意外断开了，那个数据表也不会成为“流浪汉”。比如说，假设你有一组复杂的查询命令保存在一个批处理文件里，你通过 mysql 工具运行了那个批处理文件但临时决定不等它执行完毕，你可以毫无顾虑地“杀”掉那个脚本，而服务器将删除由该脚本创建的所有临时数据表。

如果想使用某种存储引擎来创建临时数据表，给 CREATE TEMPORARY TABLE 语句加上 ENGINE 数据表选项即可。

服务器会在你的客户会话结束时自动删除一个 TEMPORARY 数据表，但它也完全可以在用完它之后显式地删除它，这可以让服务器尽快释放与之相关联的资源。如果你与服务器之间的会话还需要再持续一段时间，及时释放不再需要的资源是一种良好习惯，尤其是那些临时性的 MEMORY 数据表。

一个 TEMOPARY 数据表只对创建该数据表的客户是可见的。因为每个客户只能看到它自己创建的数据表，所以不同的客户可以各自创建一个名字相同的 TEMPORARY 数据表而不会发生冲突。

一个 TEMPORARY 数据表的名字允许与一个现有的永久性数据表相同。这不是一个错误，那个现有的永久性数据表也不会因此遭到损坏。只是创建这个 TEMPORARY 数据表的客户在此表存在期间将不再能够看见（也就是无法访问）那个永久性数据表而已。比如说，如果你在 sampdb 数据库中创建了一个名为 member 的 TEMPORARY 数据表，原有的 member 数据表将被隐藏起来，对 member 数据表的访问将作用于新建的 TEMPORARY 数据表。如果你发出一条 DROP TABLE member 语句，被删除的将是那个 TEMPORARY 数据表，原有的 member 数据表将重现在你眼前。如果你在没有删除那个 TEMPORARY 数据表的情况下断开了与服务器的连接，服务器将自动地替你删除它。等你下次连接的时候，原有的 member 数据表就又是可见的了。（如果把一个 TEMPORARY 数据表重新命名为另外一个名字，原有的数据表也会变成可见的。）

这种“隐姓埋名”机制只给你一次机会，因为你无法创建两个同名的 TEMPORARY 数据表。

在考虑要不要使用 TEMPORARY 数据表时，请注意以下几个因素。

- 如果客户程序会在与服务器的连接意外断开时自动重建连接，上次创建的 TEMPORARY 数据表在你重新连接上服务器时将不复存在。如果使用 TEMPORARY 数据表的目的是为了“隐藏”一

个与之同名的永久性数据表，那个永久性的数据表现在就会变成可用的，而这就带来了一定的风险。比如说，如果连接意外断开后立刻得到重建但你没有察觉，你发出的 DROP TABLE 语句将会导致那个永久性的数据表被删除。如果想避免这种问题，就应该使用 DROP TEMPORARY TABLE 语句来代替之。

- 因为 TEMPORARY 数据表只对创建它们的连接是可见的，所以如果使用了某种连接池机制，它们的用处就没有多大了，因为连接池机制不能保证你发出的每一条语句使用的都是同一条连接。
- 如果使用了连接池或永久性连接，你与 MySQL 服务器之间的连接在应用程序结束时就不一定会被关闭。那些机制可能会让连接保持打开状态供其他客户使用，而这意味着你创建的 TEMPORARY 数据表不一定会在你的应用程序结束时自动消失。

4. 从其他数据表或查询结果创建数据表

在某些场合，为某个数据表创建一份副本很有必要。比如说，你有一个数据文件，你想用 LOAD DATA 语句把它添加到某个数据表里，但你对用来给出数据格式的选项没有把握。万一那些选项设置得不正确，你就会把一些乱七八糟的数据行插入到原始数据表里。如果你有一份原始数据表的空白副本，你就可以放心大胆地通过尝试各种 LOAD DATA 选项的办法来确定那个数据文件使用的数据列和数据行分隔符是什么。等你认为来自数据文件的输入行得到了正确的解析之后，只需再次运行 LOAD DATA 语句并在该语句里给出那个原版数据表的名字，就可以把你的数据文件加载到那个原版数据表里去了。

在另外一些场合，把查询结果保存到一个数据表里要比让它们从显示器的屏幕一闪而过更符合我们的愿望。保存起来的查询结果使我们无需再次运行原始查询命令就可以使用它们，尤其是在需要对它们做进一步分析的时候。

MySQL 提供了两条语句来帮助我们从其他的数据表或是从查询结果创建新的数据表。这两条语句各有各的优点和缺点，如下所示。

- CREATE TABLE...LIKE 语句将创建一个新数据表作为原始数据表的一份空白副本。它将把原始数据表的结构丝毫不差地复制过来，不仅各数据列的所有属性都会得到保留，就连索引结构也会复制得一模一样。不过，因为新数据表的内容是一片空白，所以如果想填充它，就需要再使用一条语句（如 INSERT INTO...SELECT）。请注意，CREATE TABLE...LIKE 语句不能从原始数据表的数据列的一个子集创建出一个新数据表，它也不能使用除原始数据表以外的任何其他数据表里的数据列。
- CREATE TABLE...SELECT 语句可以从任意一条 SELECT 语句的查询结果创建新数据表。在默认的情况下，这条语句不会复制所有的数据列属性，如 AUTO_INCREMENT 等。通过选取数据到其中而创建新数据表也不会自动复制原始数据表里的所有索引，因为结果集本身不带索引。从另一方面讲，CREATE TABLE...SELECT 语句只需一条语句就可以完成创建和填充新数据表两项任务。它还可以用原始数据表的一个子集创建一个新数据表，并包括来自其他数据表的数据列或作为表达式结果而被创建出来的数据列。

使用 CREATE TABLE...LIKE 语句为一个现有的数据表创建一份空白副本的基本语法如下所示：

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

为数据表创建一份空白副本，再从原始数据表填充它，需要先使用一条 CREATE TABLE...LIKE 语句，再使用一条 INSERT INTO...SELECT 语句：

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

```
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```

如果想创建一个数据表作为它本身的一个临时副本，需要加上 TEMPORARY 关键字：

```
CREATE TEMPORARY TABLE new_tbl_name LIKE tbl_name;
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```


创建一个与原始数据表同名的 TEMPORARY 数据表，这在你打算使用一些语句去修改该数据表的内容、但又不想改变原始数据表的内容时会很有用。比如说，如果在事先编写好的脚本里使用的是原始数据表的名字，你无需改写脚本使用另外一个数据表，只要在脚本的开头加上一条 CREATE TEMPORARY TABLE 语句和一条 INSERT 语句就可以了。你的脚本将创建一份临时副本并在该份副本上进行各种操作，当脚本结束时，服务器会自动删除它。（不过，千万要注意上一小节提到的自动重建连接的问题。）

如果只想把原始数据表里的一部分数据行插入到新数据表里，可以增加一条 WHERE 子句来选取有关的数据行。下面的语句将创建一个名为 student_f 的新数据表，它只包含 student 数据表里的女学生的数据行：

```
CREATE TABLE student_f LIKE student;
INSERT INTO student_f SELECT * FROM student WHERE sex = 'f';
```

如果不关心新数据表是否保留了原始数据表里的数据列的精确定义，CREATE TABLE...SELECT 语句有时要比 CREATE TABLE...LIKE 语句更容易使用，因为前者只需一条语句就可以创建并填充新数据表：

```
CREATE TABLE student_f SELECT * FROM student WHERE sex = 'f';
```

用 CREATE TABLE...SELECT 语句创建出来的新数据表并非只能包含来自某一个数据表的数据列。你随时都可以在必要时用它快速创建一个新数据表来容纳任何一次 SELECT 查询的结果。这让我们可以非常简便快速地创建一个新数据表并让该数据表的内容都成为我们想要的数据（供后面的语句使用）。不过，如果不小心，新数据表可能会包含一些奇怪的数据列名字。当你通过选取数据到其中而创建数据表时，数据列的名字来自你正在选取的数据列。如果某个数据列是一个表达式的计算结果，该数据列的名字将是表达式的文本，而如此创建出来的数据表将包含一个不同寻常的数据列名字，如下所示：

```
mysql> CREATE TABLE mytbl SELECT PI() * 2;
mysql> SELECT * FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

这显然不理想，因为这样的数据列名字只能以一个用引号括起来的标识符的形式被直接引用：

```
mysql> SELECT `PI() * 2` FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

要想避免这个问题并提供一个便于使用的数据列名字，可以使用一个别名：

```
mysql> DROP TABLE mytbl;
mysql> CREATE TABLE mytbl SELECT PI() * 2 AS mycol;
mysql> SELECT mycol FROM mytbl;
+----------+
| mycol    |
+----------+
| 6.283185 |
+----------+
```

使用别名会带来一个小问题：如果你还从另一个数据表选取了一个同名的数据列，就会发生冲突。比如说，假设数据表 t1 和 t2 都有一个数据列 c，而你想从这两个数据表的所有数据行的组合里创建一个数据表。下面的语句将失败，因为它试图创建的数据表里有冲突——两个数据列的名字都是 c：

```
mysql> CREATE TABLE t3 SELECT * FROM t1 INNER JOIN t2;
ERROR 1060 (42S21): Duplicate column name 'c'
```

这个问题并不难解决，通过提供必要的别名让两个数据列在新数据表里各有一个独一无二的名字就行了：

```
mysql> CREATE TABLE t3 SELECT t1.c, t2.c AS c2
    -> FROM t1 INNER JOIN t2;
```

正如刚才提到的那样，CREATE TABLE...SELECT 语句的缺点之一是它不会把原始数据的所有特征全部复制到新数据表的结构里去。比如说，通过选取数据到其中来创建一个数据表不会把原始数据表里的索引复制过去，而且还可能失去数据列属性。可以保留下来的属性包括数据列是 NULL 还是 NOT NULL、字符集和排序方式、默认值和数据列注释。

在某些场合，你可以通过在语句的 SELECT 部分使用 CAST()函数的办法在新数据表里强制使用特定的属性。下面的 CREATE TABLE...SELECT 语句将强制由 SELECT 子句所生成的数据列被视为 INT UNSIGNED、TIME 和 DECIMAL(10,5)来对待，就像用 DESCRIBE 语句进行验证时看到的那样：

```
mysql> CREATE TABLE mytbl SELECT
    -> CAST(1 AS UNSIGNED) AS i,
    -> CAST(CURTIME() AS TIME) AS t,
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| i     | int(1) unsigned | NO   |     | 0       |       |
| t     | time             | YES  |     | NULL    |       |
| d     | decimal(10,5)    | NO   |     | 0.00000 |       |
+-------+------------------+------+-----+---------+-------+
```

允许的投射类型是 BINARY(二进制串)、CHAR、DATE、DATETIME、TIME、SIGNED、SIGNED INTEGER、UNSIGNED、UNSIGNED INTEGER 和 DECIMAL。

还可以在 CRAETE TABLE 部分提供明确的数据列定义，然后在 SELECT 部分使用那些定义去检索数据列。两个部分中的数据列按名字匹配，所以在 SELECT 部分往往需要提供一些必要的别名才能让它们正确地得到匹配：

```
mysql> CREATE TABLE mytbl (i INT UNSIGNED, t TIME, d DECIMAL(10,5))
    -> SELECT
    -> 1 AS i,
```

```
    -> CAST(CURTIME() AS TIME) AS t,
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| i     | int(10) unsigned | YES  |     | NULL    |       |
| t     | time             | YES  |     | NULL    |       |
| d     | decimal(10,5)    | YES  |     | NULL    |       |
+-------+------------------+------+-----+---------+-------+
```

提供明确的数据列定义的技巧使你能够创建具有特定精度和取值范围的数值数据列、与结果集中最长的值的宽度不同的字符数据列，等等。此外，请注意在这个例子里有几个数据列的Null和Default属性与前面几个例子里的情况不一样。如有必要，你可以在CREATE TABLE部分为这些属性提供明确的定义。

5. 使用MERGE数据表

MERGE存储引擎把一组MyISAM数据表当做一个逻辑单元来对待，让我们可以同时对它们进行查询。正如在前面2.6.1节所描述的，构成一个MERGE数据表的各成员MyISAM数据表必须具有完全一样的结构。每一个成员数据表里的数据列必须按照同样的顺序定义同样的名字和类型，索引也必须按照同样的顺序和同样的方式定义。

假设你有几个日志数据表，它们的内容是分别是这几年来每一年里的日志记录项，它们的定义都是下面这样，其中*CC*代表世纪，*YY*代表年份：

```
CREATE TABLE log_CCYY
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MyISAM;
```

假设日志数据表的当前集合包括log_2004、log_2005、log_2006、log_2007和log_2008，而你可以创建一个如下所示的MERGE数据表把它们归拢为一个逻辑单元：

```
CREATE TABLE log_merge
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MERGE UNION = (log_2004, log_2005, log_2006, log_2007, log_2008);
```

ENGINE选项的值必须是MERGE，UNION选项列出了将被收录在这个MERGE数据表里的各有关数据表。把这个MERGE数据表创建出来之后，你可以像对待任何其他的数据表那样查询它，只是每次查询都将同时作用于构成它的每一个成员数据表。下面这个查询可以让我们知道上述几个日志数据表里的数据行的总数：

```
SELECT COUNT(*) FROM log_merge;
```

下面这个查询用来确定在这几年里每年各有多少条日志记录项：

```
SELECT YEAR(dt) AS y, COUNT(*) AS entries FROM log_merge GROUP BY y;
```

除了便于同时引用多个数据表而无须发出多条查询命令，MERGE数据表还提供了以下一些便利。

- MERGE 数据表可以用来创建一个尺寸超过各个 MyISAM 数据表所允许的最大长度的逻辑单元。
- 你可以把经过压缩的数据表包括到 MERGE 数据表里。比如说，在某一年结束之后，你应该不会再往相应的日志文件里添加任何记录了，所以你可以用 myisampack 工具压缩它以节省空间，而 MERGE 数据表仍可以像以前那样使用。

MERGE 数据表也支持 DELETE 和 UPDATE 操作。INSERT 操作比较麻烦，因为 MySQL 需要知道应该把新数据行插入到哪一个成员数据表里去。在 MERGE 数据表的定义里可以包括一个 INSERT_METHOD 选项，这个选项的可取值是 NO、FIRST 和 LAST，它们的含义依次是 INSERT 操作是被禁止的、新数据行将被插入到在 UNION 选项里列出的第一个数据表或最后一个数据表。比如说，如下所示的定义将使对 log_merge 数据表的 INSERT 操作被当做对 log_2008 数据表——它是 UNION 选项所列出的最后一个数据表——的一个 INSERT 操作：

```
CREATE TABLE log_merge
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MERGE UNION = (log_2004, log_2005, log_2006, log_2007, log_2008)
INSERT_METHOD = LAST;
```

创建一个新的成员数据表 log_2009 并让它有着与其他 log_*CCYY* 数据表同样的结构，然后修改 log_merge 数据表的定义把 log_2009 包括进来即可：

```
log_2009:
CREATE TABLE log_2009 LIKE log_2008;
ALTER TABLE log_merge
UNION = (log_2004, log_2005, log_2006, log_2007, log_2008, log_2009);
```

6. 使用分区数据表

MySQL 5.1 及更高版本支持分区数据表（partitioned table）。分区在概念上与 MERGE 存储引擎很相似：它们都可以用来访问被分别存储在不同地点的多个数据表的内容。这两者之间的区别是：每个分区数据表都是一个货真价实的数据表，而不是一个用来列出各成员数据表的逻辑构造。此外，分区数据表可以使用 MyISAM 以外的存储引擎，而 MERGE 数据表只能用 MyISAM 数据表来构成。

通过对数据表的存储进行划分，分区数据表具有以下几个优点。

- 数据表的存储可以分布在多个设备上，这意味着我们可以通过建立某种 I/O 并行机制缩短访问时间。
- 优化器可以把检索操作限定在某个特定的分区或是同时搜索多个分区。

在创建一个分区数据表的时候，先像往常一样在 CREATE TABLE 语句里给出数据列和索引的清单，然后用一条 PARTITION BY 子句定义一个用来把数据行分配到各个分区的分区函数，再写出其他必须的分区选项即可。分区函数的作用类似于我们在创建 MERGE 数据表时使用的 INSERT_METHOD 选项，只是它更通用：它可以把新数据行分布到所有的分区，而 INSERT_METHOD 选项只能把所有的新数据行插入到同一个数据表。

分区函数把新数据行分配到不同分区的依据可以是取值范围、值的列表或散列值，如下所示。

- 根据取值范围进行分区。数据行所包含的值可以划分为一系列互不冲突的区间，比如日期、收入水平、重量等。

- 根据列表进行分区。每个分区分别对应一个明确的值的列表，比如邮政编码表、电话号码区号、身份证号码中的地区编号等。
- 根据散列值进行分区。根据数据行的键字计算出一个散列值，再根据这个散列值把数据行分布到各分区。你可以自行提供一个散列函数，也可以列出一组数据列让MySQL使用一个内建的散列函数去计算那些数据列的散列值。

分区函数必须具备这样一种确定性：同样的输入永远会把数据行分配到同一个分区。按照这一要求，诸如 RAND()和 NOW()之类的函数显然不适合用做分区函数。

作为一个简单的例子，让我们一起来为第 5 小节里的 MERGE 数据表创建一个分区数据表版本。那个名为 log_merge 的 MERGE 数据表由多个成员数据表构成，它们的内容分别是 2004 至 2008 年期间每一年里的日志记录项。相应的分区数据表将是一个被划分为多个分区的数据表。因为在构成日志记录项的数据里肯定会包含一个日期值，所以根据这个日期值的取值范围进行分区是最顺理成章的办法。我决定根据年份（也就是日期值里的“年”部分）来把数据行分配到一个给定的分区：

```
CREATE TABLE log_partition
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
)
PARTITION BY RANGE(YEAR(dt))
(
  PARTITION p0 VALUES LESS THAN (2005),
  PARTITION p1 VALUES LESS THAN (2006),
  PARTITION p2 VALUES LESS THAN (2007),
  PARTITION p3 VALUES LESS THAN (2008),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

根据上面的定义，2008 年和以后的日志记录项将被分配到 MAXVALUE 分区。2009 年时可以对这个分区再进行划分，把 2008 年的日志记录项转移到它们自己的一个分区里，把 2009 年和以后的日志记录项仍保留在 MAXVALUE 分区里，如下所示：

```
ALTER TABLE log_partition REORGANIZE PARTITION p4
INTO (
  PARTITION p4 VALUES LESS THAN (2009),
  PARTITION p5 VALUES LESS THAN MAXVALUE
);
```

在默认的情况下，分区被保存在分区数据表所属于的数据库的子目录里。若想将存储分配到其他位置（如另一个物理设备），需要使用 DATA_DIRECTORY 和 INDEX_DIRECTORY 分区选项。如果想了解关于这两个以及其他分区选项的语法的更多信息，请参阅附录 E 中对 CREATE TABLE 语句的描述。

7. 使用FEDERATED数据表

FEDERATED 存储引擎可以让你访问在其他主机上由另一个 MySQL 服务器实际管理的数据表。

假设你的本地服务器上有一个名为 sampdb 的数据库，在网址是 corn.snake.net 的主机上还有一个 MySQL 服务器也管理着一个同名的数据库，而你有一个账户可以访问那个远程服务器。这样一来，你就可以使用那个账户通过 FEDERATED 存储引擎在本地主机上使用位于远程主机的 sampdb 数据库。对于每一个你想如此访问的数据表，你必须创建一个与远程数据表有着同样数据列的 FEDE-

RATED 数据表，并给出相应的连接字符串让本地服务器知道如何连接远程服务器。

假设远程服务器上的 student 数据表有着如下所示的定义：

```
CREATE TABLE student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = InnoDB;
```

在本地服务器上创建一个相应的 FEDERATED 数据表必须使用同样的定义，还必须把 ENGINE 选项设置为 FEDERATED，再通过 CONNECTION 连接选项给出建立连接所必需的信息。（MySQL 5.0.13 之前的版本需要用 COMMENT 选项代替 CONNECTION 选项。）如下所示的定义将创建一个名为 federated_student 的数据表，它用来访问位于 corn.snake.net 的主机上的 student 数据表：

```
CREATE TABLE federated_student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = FEDERATED
CONNECTION = 'mysql://sampadm:secret@corn.snake.net/sampdb/student';
```

从 CONNECTION 选项所给出的连接字符串可以看到，用来访问远程服务器的 MySQL 账户的用户名和口令是 sampdb 和 secret。连接字符串的基本语法如下所示，方括号里是可选的信息。

```
mysql://user_name[:password]@host_name[:port_num]/db_name/tbl_name
```

把 federated_student 数据表创建出来之后，你就可以通过查询它而访问远程的 student 数据表了。你还可以通过对 federated_student 数据表进行 INSERT、UPDATE 和 DELETE 操作的办法去改变那个远程 student 数据表的内容。

这里有一个值得注意的问题：整个 CONNECTION 字符串（包括用户名和口令）对可以使用 SHOW CREATE TABLE 或类似语句去查看你的 FEDERATED 数据表的任何人来说都是可见的。从 MySQL 5.1.15 版开始，你可以避免这个问题：提前用 CREATE SERVER 语句创建一个存储服务器定义（这需要 SUPER 权限），然后在 CONNECTION 选项里写出该服务器的名字即可。下面这条语句定义了一个名为 corn_sampdb-server 的存储服务器：

```
CREATE SERVER corn_sampdb_server
FOREIGN DATA WRAPPER mysql
OPTIONS (
  USER 'sampadm',
  PASSWORD 'secret',
  HOST 'corn.snake.net',
  DATABASE 'sampdb'
);
```

MySQL 服务器将把这个定义保存为 mysql 数据库中的 servers 数据表的一个数据行。如果你想创建一个引用这个服务器定义的数据表，在如下所示的语句里通过 CONNECTION 选项给出远程服务器的名字即可：

```
CREATE TABLE federated_student2
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = FEDERATED
CONNECTION = 'corn_sampdb_server/student';
```


与把连接参数直接写在 CONNECTION 选项里的做法相比，使用服务器定义可以提高安全性，因为这种定义只有那些有权访问 mysql 数据库的用户才能查看到。此外，服务器定义还可以简化数据表的创建工作，共享着同样的连接参数的多个 FEDERATED 数据表可以使用同一个定义。

2.6.3 删除数据表

删除数据表要比创建它容易得多，因为你不需要给出任何关于其内容的格式的信息。你只需给出它的名字即可：

```
DROP TABLE tbl_name;
```

MySQL 为 DROP TABLE 语句提供了几种很有用的变体。如果需要删除多个数据表，把它们依次列在同一条语句里即可：

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

如果你不能肯定某个数据表是否已经存在，但如果它存在的话你就要删除它，请在语句里加上 IF EXIST 子句：

```
DROP TABLE IF EXISTS tbl_name;
```

IF EXIST 子句的作用是在给定的数据表不存在时不让这条语句报告错误。（但每出现一次这样的情况，服务器就会生成一条警告消息，你可以用 SHOW WARNINGS 语句去查看它们。）

IF EXIST 子句非常适合用在你将通过 mysql 客户工具去运行的脚本里。在默认的情况下，mysql 工具会在有错误发生时退出运行，而删除一个并不存在的数据表是一个错误。比如说，假设你有一个初始化脚本，它负责创建一些数据表供其他脚本做进一步处理。在这种场合里，你应该确保初始化脚本在开始运行时有一个干净的环境。如果你在这个脚本的开头使用的是普通的 DROP TABLE 语句，它在第一次运行时就会失败，因为它想要删除的数据表还都没有被创建出来。如果你使用了 IF EXIST 子句，就不会出问题了。如果数据表已经存在，它们将被删除；如果它们不存在，也不会发生错误，脚本将继续执行。

如果你只想删除临时数据表，加上 TEMPORARY 关键字即可：

```
DROP TEMPORARY TABLE tbl_name;
```

2.6.4 为数据表编制索引

索引是加快对数据表内容的访问速度的基本手段，尤其是在涉及多个数据表的关联查询里。这是一个非常重要的主题，我们在后面用了整整一章的篇幅来讨论为什么要使用索引，它们是如何工作的，以及怎样才能最大限度地利用索引去优化查询（参见第 5 章）。本小节的重点是介绍各种适用于不同数据表类型的索引的特性以及创建和删除索引的语法。

1. 存储引擎的索引特性

MySQL 提供了多种灵活的索引创建办法，如下所示。

- 你可以为单个数据列编制索引，也可以为多个数据列构造复合索引。
- 索引可以只包含独一无二的值，也可以包含重复的值。
- 你可以为同一个数据表创建多个索引并分别利用它们来优化基于不同数据列的查询。
- 对于 ENUM 和 SET 以外的字符串数据类型，可以只为数据列的一个前缀创建索引，也就是为对最左边的 n 个字符（对二进制字符串类型来说就是最左边的 n 个字节）创建索引。（对于 BLOB 和 TEXT 数据列，你只有在指定了前缀长度的情况下才能创建一个索引。）如果数据列在前缀长度范围内具有足够的独一无二性，查询性能通常不会受到影响，而是会得到改善：为数据列前缀而不是整个数据列编索引可以让索引本身更小并加快访问速度。

并非所有的存储引擎都能提供全部的索引功能。表 2-3 对 MySQL 的几种存储引擎的索引特性进行了汇总。这个表没有包括 MERGE 存储引擎，因为 MERGE 数据表是从一组 MyISAM 数据表创建出来的，它们有相似的索引特性。它也没有包括 ARCHIVE、BLACKHOLE、CSV 或 EXAMPLE 引擎，它们或者根本不支持索引，或者支持很有限。

表2-3　存储引擎的索引特性

索引特性	MyISAM	MEMOR	InnoDB	Falcon
是否允许使用NULL值	是	是	是	是
每个索引最多支持多少个数据列	16	16	16	16
每个数据表最多可以有多少个索引	64	64	64	64
索引项的最大长度（字节）	1000	1024/3072	1024/3072	1100
能否为数据列前缀创建索引	能	能	能	能
数据列前缀的最大长度（字节）	1000	1024/3072	767	1100
是否支持BLOB/TEXT索引	是	否	是	否
是否支持FULLETXT索引	是	否	否	否
是否支持SPATIAL索引	是	否	否	否
是否支持HASH索引	否	是	否	否

对于 MEMORY 和 InnoDB 存储引擎，索引项的最大长度在 MySQL 5.0.17/5.1.4 之前的版本里是 1024 个字节，在 5.0.17/5.1.4 及更高的版本里是 3 072 个字节。MEMORY 存储引擎的索引前缀的最大长度也是如此。

不同的存储引擎具备不同的索引特性，其中有这样一层含义：如果你要求某个索引必须具备某种特性，你将无法使用特定类型的数据表。比如说，如果你想使用 FULLTEXT 或 SAPTIAL 索引，就必须使用 MyISAM 数据表；如果你想对一个 TEXT 数据列编索引，就必须使用 MyISAM 或 InnoDB 数据表。

如果你想让一个现有的数据表改用另外一种有着你更需要的索引特性的存储引擎，可以使用 ALTER TABLE 语句来改变存储引擎。假设你正在使用一个 MyISAM 数据表但需要 InnoDB 或 Falcon 存储引擎提供的事务处理能力，你将需要使用如下所示的语句对数据表进行转换：

```
ALTER TABLE tbl_name ENGINE = InnoDB;
ALTER TABLE tbl_name ENGINE = Falcon;
```

2. 创建索引

MySQL 可以创建好几种索引，如下所示。

- 唯一索引。这种索引不允许索引项本身出现重复的值。对只涉及一个数据列的索引来说，这意味着该数据列不能包含重复的值。对涉及多个数据列的索引（复合索引）来说，这意味着那几个数据列的值的组合在整个数据表的范围内不能出现重复。
- 普通（非唯一）索引。这种索引的优点（从另一方面看是缺点）是允许索引值出现重复。
- FULLTEXT 索引。用来进行全文检索。这种索引只适用于 MyISAM 数据表。如果你想了解更多信息，请参阅 2.15 节。
- SPATIAL 索引。这种索引只适用于 MyISAM 数据表和空间（spatial）数据类型，对这种数据类型的描述见第 3 章。（对于其他支持空间数据类型的存储引擎，你可以创建非 SPATIAL 索引。）
- HASH 索引。这是 MEMORY 数据表的默认索引类型，但你可以改用 BTREE 索引来代替这个默认索引。

你可以在使用 CREATE TABLE 语句创建新数据表时创建索引。这方面的例子参见 1.4.6 节。用 ALTER TABLE 或 CREATE INDEX 语句可以给现有数据表添加索引。（MySQL 会在其内部把 CREATE INDEX 语句映射为 ALTER TABLE 操作。）

ALTER TABLE 语句比 CREATE INDEX 语句更灵活多能，因为它可以用来创建 MySQL 所能支持的任何一种索引。比如说：

```
ALTER TABLE tbl_name ADD INDEX index_name (index_columns);
ALTER TABLE tbl_name ADD UNIQUE index_name (index_columns);
ALTER TABLE tbl_name ADD PRIMARY KEY (index_columns);
ALTER TABLE tbl_name ADD FULLTEXT index_name (index_columns);
ALTER TABLE tbl_name ADD SPATIAL index_name (index_columns);
```

tbl_name 是你想添加索引的数据表的名字，*index_columns* 是你想加索引的一个或多个数据列。如果索引由多个数据列构成，要用逗号把它们的名字隔开。索引本身的名字 *index_name* 是可选的，如果你没有给出，MySQL 将根据第一个带索引的数据列给它挑选一个名字。

如果某个索引是一个 PRIMARY KEY 或 SPATIAL 索引，带索引的数据列必须具备 NOT NULL 属性。其他用途的索引都允许包含 NULL 值。

只需用逗号把它们彼此隔开，你就可以在同一条 ALTER TABLE 语句里包括多个对数据表的改动。这意味着你可以同时创建多个索引，这比使用多条 ALTER TABLE 语句逐个地添加索引的办法要快很多。

如果想限制某个索引只包含独一无二的值，可以把该索引创建为一个 PRIMARY KEY 或一个 UNIQUE 索引。这两种索引很相似，但有两点区别，如下所示。

- 每个数据表只能有一个 PRIMARY KEY。（这是因为 PRIMARY KEY 的名字总是 PRIMARY，而同一个数据表不允许有两个同名的索引。）
- PRIMARY KEY 不可以包含 NULL 值，而 UNIQUE 索引可以。而且，如果你允许某个 UNIQUE 索引包含 NULL 值，那它将可以包含多个 NULL 值。这是因为 MySQL 无法判断两个 NULL 值是否代表同样的东西，索引里的多个 NULL 值将被认为代表多个不同的东西。

除 PRIMARY KEY 以外，其他类型的索引几乎都可以用 CREATE INDEX 语句来添加：

```
CREATE INDEX index_name ON tbl_name (index_columns);
CREATE UNIQUE INDEX index_name ON tbl_name (index_columns);
CREATE FULLTEXT INDEX index_name ON tbl_name (index_columns);
CREATE SPATIAL INDEX index_name ON tbl_name (index_columns);
```

tbl_name、*index_name* 和 *index_columns* 的含义和 ALTER TABLE 语句里的一样。与 ALTER TABLE 语句不同的是，CREATE INDEX 语句里的 *index_name* 不是可选的，你也不能用一条 CREATE INDEX 语句创建多个索引。

在使用 CREATE TABLE 语句创建新数据表的同时为它创建索引的语法，和使用 ALTER TABLE 语句添加索引时用的语法很相似，这需要你在定义各数据列的基础上再增加一些索引创建子句：

```
CREATE TABLE tbl_name
(
  ... column definitions ...
  INDEX index_name (index_columns),
  UNIQUE index_name (index_columns),
  PRIMARY KEY (index_columns),
  FULLTEXT index_name (index_columns),
  SPATIAL index_name (index_columns),
  ...
);
```

类似于 ALTER TABLE 语句，*index_name* 在这里也是可选的。如果你省略了它，MySQL 将替你挑选一个。

作为一种特殊情况，你可以通过在某个数据列的定义的末尾加上一条 PRIMARY KEY 或 UNIQUE 子句的办法，来创建一个单数据列的 PRIMARY KEY 或 UNIQUE 索引。比如说，下面两条 CREATE TABLE 语句是等价的：

```
CREATE TABLE mytbl
(
  i INT NOT NULL PRIMARY KEY,
  j CHAR(10) NOT NULL UNIQUE
);

CREATE TABLE mytbl
(
  i INT NOT NULL,
  j CHAR(10) NOT NULL,
  PRIMARY KEY (i),
  UNIQUE (j)
);
```

MEMORY 数据表的默认索引类型是 HASH。利用散列索引进行精确值查询的速度非常快，这也是 MEMORY 数据表的典型用法。不过，如果你打算用一个 MEMORY 数据表进行范围比较（如 id < 100），散列索引的使用效果就没那么理想了。在遇到这类情况时，你最好创建一个 BTREE 索引来代替之，具体做法是在索引定义里增加一条 USING BTREE 子句：

```
CREATE TABLE namelist
(
  id   INT NOT NULL,
  name CHAR(100),
  INDEX USING BTREE (id)
) ENGINE = MEMORY;
```

如果只对某个字符串数据列的一个前缀编索引，在索引定义里命名数据列的语法是 *col_name*(*n*) 而不是简单的 *col_name*。*n* 的含义是索引应该包括数据列值的前 *n* 个字节（二进制字符串类型）或前 *n* 个字符（非二进制字符串类型）。比如说，用下面这条语句创建出来的数据表有一个 CHAR 数据列和一个 BINARY 数据列。它对 CHAR 数据列的前 10 个字符和 BINARY 数据列的前 15 个字节编了索引：

```
CREATE TABLE addresslist
(
  name    CHAR(30) NOT NULL,
  address BINARY(60) NOT NULL,
  INDEX (name(10)),
  INDEX (address(15))
);
```

在对某个字符串数据列的一个前缀编索引时，前缀长度的计量单位必须与该数据列的数据类型相同，就像数据列的长度那样。换句话说，二进制字符串以字节为单位，非二进制字符串以字符为单位。不过，索引项本身的最大长度在 MySQL 内部以字节为单位。这两种长度计量办法对单字节字符集来说没有什么区别，对多字节字符集来说则影响重大。如果某个非二进制字符串包含来自多字节字符集的字符，MySQL 会在索引所允许的字节长度范围内容纳尽可能多的字符。

- 在某些场合，你会发现对数据列前缀（而不是对整个数据列）编索引已经不是你想不想那样做的问题，你只能那样做。这类情况包括以下几种。
- BLOB 或 TEXT 数据列只能创建前缀型索引。
- 索引项本身的长度等于构成索引的各个数据列的索引部分的长度总和。如果这个长度超过了索引项本身所能容纳的最大字节数，你可以通过为数据列前缀编索引来“缩短”这个索引。比如说，假设某个 MyISAM 数据表使用了 latin1 单字节字符集并且包含 4 个 CHAR(255)数据列，数据列的名字是 c1 到 c4。此时，每个数据列的索引项的长度将占用 255 个字节，4 个数据列的索引项的总长度将有 1 020 个字节。可是，因为一个 MyISAM 索引项的最大长度是 1 000 个字节，所以你无法创建一个复合索引把那 4 个数据列的完整内容全部包括进来。解决问题的办法是对部分或全部数据列的一个比较短的前缀部分编索引。比如说，你可以只为前 250 个字符编索引。

FULLTEXT 索引所涉及的数据列是全部带索引的，没有前缀的说法。即使你为 FULLTEXT 索引所涉及的某个数据列指定了一个前缀长度，MySQL 也会忽略它。

可以像下面这样为空间数据类型的数据列（如 POINT 或 GEOMETRY）编索引。

- SPATIAL 索引只能用于 MYISAM 数据表，只能用于 NOT NULL 属性的数据列。所涉及的数据列是全部带索引的。
- 其他索引类型（INDEX、UNIQUE、PRIMNARY KEY）可以配合除 ARCHIVE 以外支持空间数据类型的任何存储引擎使用。不是 PRIMARY KEY 的组成部分的数据列都允许包含 NULL 值。除 POINT 数据列以外，索引所涉及的空间数据列以字节计算的前缀长度必须在创建索引时给出。

3. 删除索引

删除索引的工作可以用 DROP INDEX 或 ALTER TABLE 语句来完成。如果使用的是 DROP INDEX 语句，你必须给出要删除的索引的名字：

```
DROP INDEX index_name ON tbl_name;
```

如果你想用 DROP INDEX 语句删除一个 PRIMARY KEY，就必须以一个带引号的标识符的形式给出

名字 PRIMARY，如下所示：

```
DROP INDEX `PRIMARY` ON tbl_name;
```

这条语句没有任何歧义，因为每个数据表只有一个 PRIMARY KEY，它的名字也永远是 PRIMARY。

类似于 CREATE INDEX 语句，DROP INDEX 在 MySQL 内部将被当做一条 ALTER TABLE 语句来处理。上面这条 DROP INDEX 语句等价于下面两条 ALTER TABLE 语句：

```
ALTER TABLE tbl_name DROP INDEX index_name;
ALTER TABLE tbl_name DROP PRIMARY KEY;
```

如果你不知道某给定数据表的索引的名字，可以用 SHOW CREATE TABLE 或 SHOW INDEX 把它们查出来。

当你从数据表删除数据列时，索引也会隐式地受到影响。如果你删除的数据列是某个索引的组成部分，MySQL 将从索引里删除那个数据列。如果你把构成某个索引的数据列全都丢弃了，MySL 将删除整个索引。

2.6.5　改变数据表的结构

ALTER TABLE 语句的用途非常多。我们在本章前半部分内容里已经见识过它的一些本领了（如改变存储引擎、创建和删除索引等）。你还可以使用 ALTER TABLE 语句来重新命名数据表、添加或删除数据列、改变数据列的数据类型等。本小节只讨论这条语句的一部分功能，附录 E 将描述 ALTER TABLE 语句的完整语法。

当你发现某个数据表的结构不再满足需要时，ALTER TABLE 语句可以帮上你的大忙。比如说，也许是你想用这个数据表来记录一些额外的信息，也许这个数据表所包含的信息已经过时了，也许现有的数据列太小，也许是你当初定义的数据列宽度大到超出了你的需要，而你想把它们缩短一些以节省空间和改善查询性能。下面是 ALTER TABLE 语句会派上大用场的几种情况。

- 你管理着一个研究项目。你使用了一个 AUTO_INCREMENT 数据列来保存研究记录的编号。你最初的经费不是很充足，最多只能让你生成超过 5 万条记录，所以你决定使用 SMALLINT UNSIGNED 作为该数据列的数据类型，它最多能容纳 65 535 条彼此不重复的记录。没想到，这个项目的研究经费增加了，它现在足以让你再生成 5 万条记录。于是，你需要“加大”那个数据列的类型，以容纳更多的编号。
- 尺寸调整也可能会朝另一个方向发展。你创建了一个 CHAR(255)数据列，但后来发现数据表里的值没有超过 100 个字节长的。于是，你需要“缩短”那个数据列以节省空间。
- 你想让某个数据表改用另一种存储引擎以享受该引擎提供的功能。比如说，MyISAM 数据表不具备事务安全性，但你有一个应用程序需要进行事务处理。你可以让有关的数据表改用 InnoDB 或 Falcon 存储引擎，这两种引擎都支持事务处理。

下面是 ALTER TABLE 语句的语法：

```
ALTER TABLE tbl_name action [, action] ... ;
```

每个 action 代表一个你想对数据表进行的修改。有些数据库系统只允许一条 ALTER TABLE 语句完成一个改动，但 MySQL 允许用一条 ALTER TABLE 语句完成多个改动，只要用逗号把它们隔开即可。

提示 如果需要在使用 ALTER TABLE 语句之前查看一下数据表的当前定义，可以执行一条 SHOW CREATE TABLEA 语句。在执行完 ALTER TABLE 语句之后，你还可以用这条语句去验证一下你做的改动对数据表定义的影响是否符合你的预期。

下面的例子演示了 ALTER TABLE 语句的一些典型用法。

改变数据列的数据类型。如果想改变某个数据列的数据类型，可以使用 CHANGE 或 MODIFY 子句。假设 mytbl 数据表里的某个数据列的数据类型是 SMALLINT UNSIGNED，你想把它改成 MEDIUMINT UNSIGNED。下面两条命令都可以达到目的：

```
ALTER TABLE mytbl MODIFY i MEDIUMINT UNSIGNED;
ALTER TABLE mytbl CHANGE i i MEDIUMINT UNSIGNED;
```

为什么在使用 CHANGE 子句时需要写两遍数据列的名字呢？因为 CHANGE 子句能够（而 MODIFY 子句不能）做到的事情是在改变其数据类型的同时重新命名一个数据列。如果想在改变其数据类型的同时把数据列 i 重新命名为 k，你可以这样做：

```
ALTER TABLE mytbl CHANGE i k MEDIUMINT UNSIGNED;
```

在 CHANGE 子句里，需要先给出想改动的数据列的名字，然后给出它的新名字和新定义。因此，即使不想重新命名那个数据列，也需要把它的名字写两遍。

如果只想改变数据列的名字，不改变它的数据类型，先写出 CHANGE old_name new_name、再写出数据列的当前定义即可。

你可以为各数据列指定字符集，具体做法是在数据列的定义里使用 CHARACTER SET 属性来改变它的字符集：

```
ALTER TABLE t MODIFY c CHAR(20) CHARACTER SET ucs2;
```

改变数据类型的重要原因之一，是改善涉及多个数据表中的数据列比较的关联查询的性能。索引通常可以用来在关联查询里比较两个相似的数据列类型，而如果两个数据列的类型完全相同的话，比较的速度会更快。假设你正在运行一个如下所示的查询：

```
SELECT ... FROM t1 INNER JOIN t2 WHERE t1.name = t2.name;
```

如果 t1.name 是 CHAR(10)，t2.name 是 CHAR(15)，查询的速度将不如它们都是 CHAR(15)时那么快。你可以把它们改成相同的，下面两条语句都可以把 t1.name 改成你需要的样子：

```
ALTER TABLE t1 MODIFY name CHAR(15);
ALTER TABLE t1 CHANGE name name CHAR(15);
```

让数据表改用另一种存储引擎。如果想让数据表改用另一种存储引擎，用 ENGINE 子句给出新存储引擎的名字就行了：

```
ALTER TABLE tbl_name ENGINE = engine_name;
```

engine_name 是一个诸如 MyISAM、MEMORY 或 InnoDB 之类的名字，不区分字母的大小写。

让数据表改用另一种存储引擎的常见原因是让它具备事务安全性。假设你有一个 MyISAM 数据表，而一个用到了这个数据表的应用程序需要进行事务操作，包括意外故障发生时的事务回滚机制。MyISAM 数据表不支持事务处理，但你可以通过把它转换为一个 InnoDB 或 Falcon 数据表让它具备事务安全性：

```
ALTER TABLE tbl_name ENGINE = InnoDB;
ALTER TABLE tbl_name ENGINE = Falcon;
```

当你打算让数据表改用另一种存储引擎时，能否达到你的目的还要取决于新老两种存储引擎的功能是否兼容。例如，以下转换就是不允许的。

- 如果你有一个数据表包含着一个 BLOB 数据列，你将不能把它转换为使用 MEMORY 引擎，因为 MEMORY 引擎不支持 BLOB 数据列。
- 如果你有一个 MyISAM 数据表包含着 FULLTEXT 或 SPATIAL 索引，你将不能把它转换为使用另一种引擎，因为只有 MyISAM 支持这两种索引。

在以下条件下，你不应该使用 ALTER TABLE 语句让数据表改用另一种存储引擎。

- MEMORY 数据表存在于内存中，在服务器退出运行时将消失。因此，如果你希望某个数据表的内容在服务器重新启动后仍然存在，就不应该把它转换为 MEMORY 类型。
- 如果你使用了一个 MERGE 数据表来管理一组 MyISAM 数据表，就应该避免使用 ALTER TABLE 语句去改变个别 MyISAM 数据表的结构，除非你决定对所有的成员 MyISAM 数据表和那个 MERGE 数据表做出同样的修改。MERGE 数据表的正常使用需要其全体成员 MyISAM 数据表有着同样的结构。
- InnoDB 数据表可以被转换为使用另一种存储引擎。不过，如果你为你的 InnoDB 数据表定义了外键约束条件，那些约束条件在转换后将不复存在，因为只有 InnoDB 才支持外键。

重新命名一个数据表。用 RENAME 子句给数据表起一个新名字：

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

另一个办法是使用 RENAME TABLE 语句来重新命名数据表。下面是它的语法：

```
RENAME TABLE old_name TO new_name;
```

ALTER TABLE 语句每次只能重新命名一个数据表，而 RENAME TABLE 语句可以一次重新命名多个数据表。比如说，你可以像下面这样交换两个数据表的名字：

```
RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t2;
```

如果在重新命名一个数据表时在它的名字前面加上了数据库名前缀，就可以把它从一个数据库移动到另一个数据库。下面两条语句都可以把数据表 t 从 sampdb 数据库移到 test 数据库去：

```
ALTER TABLE sampdb.t RENAME TO test.t;
RENAME TABLE sampdb.t TO test.t;
```

不能把一个数据表重新命名为一个已有的名字。

如果重新命名的某个 MyISAM 数据表是某个 MERGE 数据表的成员，你必须重新定义那个 MERGE 数据表，让它使用那个 MyISAM 数据表的新名字。

2.7　获取数据库的元数据

MySQL 提供了好几种办法供我们获取关于数据库和数据库里各种对象（也就是数据库的元数据）的信息：

- 各种 SHOW 语句，如 SHOW DATABASES 或 SHOW TABLES；
- INFORMATION_SCHEMA 数据库里的数据表；
- 命令行程序，如 mysqlshow 或 mysqldump。

接下来的几节描述使用这些信息来源去访问元数据的具体做法。

2.7.1 用 SHOW 语句获取元数据

MySQL 提供的 SHOW 语句能够以多种方式显示数据库的元数据。SHOW 语句可以帮助我们及时了解数据库内容的变化情况，查看各有关数据表的结构。下面的例子演示了 SHOW 语句的几种典型用法。

列出服务器所管理的数据库：

```
SHOW DATABASES;
```

查看给定数据库的 CREATE DADABASE 语句：

```
SHOW CREATE DATABASE db_name;
```

列出默认数据库或给定数据库里的数据表：

```
SHOW TABLES;
SHOW TABLES FROM db_name;
```

SHOW TABLES 语句的输出报告里不包括 TEMPORARY 数据表。

查看数据表的 CREATE TABLE 语句：

```
SHOW CREATE TABLE tbl_name;
```

查看数据表里的数据列或索引信息：

```
SHOW COLUMNS FROM tbl_name;
SHOW INDEX FROM tbl_name;
```

DESCRIBE tbl_name 和 EXPLAIN tbl_name 语句是 SHOW COLUMNS FROM tbl_name 语句的同义语句。

查看默认数据库或某给定数据库里的数据表的描述性信息：

```
SHOW TABLE STATUS;
SHOW TABLE STATUS FROM db_name;
```

有几种 SHOW 语句还可以带有一条 LIKE 'pattern'子句，这个子句用来把 SHOW 语句的输出限制在给定范围。MySQL 将把'pattern'表达式解释为 SQL 模式，这种模式允许包含“%”和“_”通配符。比如说，下面这条语句可以把 student 数据表里名字以字符“s”开头的数据列查出来：

```
mysql> SHOW COLUMNS FROM student LIKE 's%';
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| sex        | enum('F','M')    | NO   |     |         |                |
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

如果需要在 LIKE 模式里实际使用某个通配符，必须在该字符的前面加上一个反斜线进行转义。这种情况常见于需要匹配“_”（下划线）字符时，该字符经常出现在数据库、数据表和数据列的名字里。

支持 LIKE 子句的所有 SHOW 语句都可以被改写为使用一条 WHERE 子句。WHERE 子句不影响 SHOW 语句的输报告里的数据列个数，它只改变 SHOW 语句输出的数据行的个数。WHERE 子句应该引用 SHOW 语句将输出的数据列。如果某个数据列的名字是一个保留字（如 KEY），就必须以带引号的标识符的方

式给出。下面这条语句用来确定 student 数据表里的主键是哪一个数据列：

```
mysql> SHOW COLUMNS FROM student WHERE `Key` = 'PRI';
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

有时候，如何能从一个应用程序里检查某个给定的数据表是否存在，这将很有用。可以用 SHOW TABLES 语句来达到这个目的（但别忘了 SHOW TABLES 语句的输出里不包括 TEMPORARY 数据表）：

```
SHOW TABLES LIKE 'tbl_name';
SHOW TABLES FROM db_name LIKE 'tbl_name';
```

如果 SHOW TABLES 列出了那个数据表的信息，就说明它存在。你还可以通过下面两条语句中的一条去检查某给定数据表是否存在，它们可以把 TEMPORARY 数据表也找出来：

```
SELECT COUNT(*) FROM tbl_name;
SELECT * FROM tbl_name WHERE FALSE;
```

如果数据表存在，这两条语句都将执行成功；如果不存在，两条语句都将失败。第一条语句最适用于 MyISAM 数据表，因为不带 WHERE 子句的 COUNT(*)函数已经得到了高度的优化。它不太适用于 InnoDB 数据表，因为对 InnoDB 数据表里的数据行的总数进行统计将导致一次全表扫描。第二条语句更通用一些，它在任何一种存储引擎下都执行得很快。这些语句最适合用在 Perl 或 PHP 等应用程序设计语言里，因为你可以测试你的查询是成功还是失败并采取相应的行动。它们不太适合用在你打算通过 mysql 工具运行的批处理脚本里，这是因为你在发生错误时除了让脚本退出运行以外不能做任何事情。（当然，你也可以忽略那个错误。但在发生错误之后，还有必要继续进行你的查询吗？）

如果你想知道各个数据表使用的是哪一种存储引擎，可以使用 SHOW TABLE STATUS 或 SHOW CREATE TABLE 语句。在这两条语句的输出报告里都有关于存储引擎指示的信息。

2.7.2　从 INFORMATION_SCHEMA 数据库获取元数据

获取数据库元信息的另一个办法是访问 INFORMATION_SCHEMA 数据库。INFORMATION_SCHEMA 数据库以 SQL 语言标准为基础。换句话说，对 INFORMATION_SCHEMA 数据库的访问机制是标准化的，虽然有一部分内容是 MySQL 特有的。这使得 INFORMATION_SCHEMA 数据库有着优于各种 SHOW 语句的可移植性，那些语句都是 MySQL 专用的。

对 INFORMATION_SCHEMA 数据库的访问需要通过 SELECT 语句来进行，所以这种访问可以非常灵活。在 SHOW 语句的输出报告里，数据列的个数是固定的，而且你无法把那些输出捕获到一个数据表里去。INFORMATION_SCHEMA 数据库就不同了，SELECT 语句可以选取特定的数据列进入输出报告，WHERE 子句可以让你通过各种表达式挑选你真正需要的信息。不仅如此，你还可以使用联结或子查询，可以使用 CREATE TABLE...SELECT 或 INSERT INTO...SELECT 语句把检索结果保存到另一个数据表做进一步处理。

可以把 INFORMATION_SCHEMA 数据库想象成一个虚拟的数据库，这个数据库里的数据表是一些由不同的数据库元数据构成的视图。如果你想知道 INFORMATION_SCHEMA 数据库都包含哪些数据表，请使用 SHOW TABLES 命令。下面的输出报告取材于 MySQL 5.1（5.0 版的数据表要少一些）。

```
mysql> SHOW TABLES IN INFORMATION_SCHEMA;
+---------------------------------------+
| Tables_in_information_schema          |
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ENGINES                               |
| EVENTS                                |
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| KEY_COLUMN_USAGE                      |
| PARTITIONS                            |
| PLUGINS                               |
| PROCESSLIST                           |
| REFERENTIAL_CONSTRAINTS               |
| ROUTINES                              |
| SCHEMATA                              |
| SCHEMA_PRIVILEGES                     |
| SESSION_STATUS                        |
| SESSION_VARIABLES                     |
| STATISTICS                            |
| TABLES                                |
| TABLE_CONSTRAINTS                     |
| TABLE_PRIVILEGES                      |
| TRIGGERS                              |
| USER_PRIVILEGES                       |
| VIEWS                                 |
+---------------------------------------+
```

下面是对各个 INFORMATION_SCHEMA 数据表的简要说明。

- SCHEMATA、TABLES、VIWS、ROUTINES、TRIGGERS、EVENTS、PARTITIONS、COLUMNS。关于数据库、数据表、视图、存储例程（stored routine）、触发器、数据库里的事件、数据表分区和数据列的信息。
- FILES。关于 NDB 硬盘数据文件的信息。
- TABLE_CONSTRAINS、KEY_COLUMN_USAGE。关于数据表和数据列上的约束条件的信息，如唯一化索引、外键等。
- STATISTICS。关于数据表索引特性的信息。
- REFERENTIAL_CONSTRAINS。关于外键的信息。
- CHARACTER_SETS、COLLATIONS、COLLATION_CHARACTER_SET_APPLICABILITY。关于所支持的字符集、每种字符集的排序方式、每种排序方式与它的字符集的映射关系的信息。
- ENGINES、PLUGINS。关于存储引擎和服务器插件的信息。
- USER_PRIVILEGES、SCHEMA_PRIVILEGES、TABLE_PRIVILEGES、COLUMN_PRIVILEGES。全局、数据库、数据表和数据列的权限信息，这些信息分别来自 mysql 数据库里的 user、db、tables_priv、column_priv 数据表。

- GLOBAL_VARIABLES、SESSION_VARIABLES、GLOBAL_STATUS、SESSION_STATUS。全局和会话级系统变量和状态变量的值。
- PROCESSLIST。关于在服务器内执行的线程的信息。

各个存储引擎还会在 INFORMATION_SCHEMA 数据库里增加它们专用的数据表。Falcon 存储引擎就是如此（如果它们已被激活）。

如果你想知道某给定 INFORMATION_SCHEMA 数据表都包含有哪些数据列，可以使用 SHOW COLLUMNS 或 DESCRIBE 语句去查看：

```
mysql> DESCRIBE INFORMATION_SCHEMA.CHARACTER_SETS;
+----------------------+-------------+------+-----+---------+-------+
| Field                | Type        | Null | Key | Default | Extra |
+----------------------+-------------+------+-----+---------+-------+
| CHARACTER_SET_NAME   | varchar(64) | NO   |     |         |       |
| DEFAULT_COLLATE_NAME | varchar(64) | NO   |     |         |       |
| DESCRIPTION          | varchar(60) | NO   |     |         |       |
| MAXLEN               | bigint(3)   | NO   |     | 0       |       |
+----------------------+-------------+------+-----+---------+-------+
```

如果你想查看某个 INFORMATION_SCHEMA 数据表里的信息，请使用 SELECT 语句。（INFORMATION_SCHEMA 数据库以及它里面的所有数据表和数据列的名字都不区分字母的大小写。）查看某给定 INFORMATION_SCHEMA 数据表里的所有数据列的通用查询语法如下所示：

```
SELECT * FROM INFORMATION_SCHEMA.tbl_name;
```

如果你想有选择地查看数据库元信息，加上一条相应的 WHERE 子句即可。

前面介绍了如何利用 SHOW 语句去检查某个数据表是否存在以及它使用的是哪一种存储引擎。INFORMATION_SCHEMA 数据表可以提供同样的信息。下面的查询利用 INFORMATION_SCHEMA 数据表去测试某个特定的数据表是否存在，如果那个数据表存在，返回 1；如果它不存在，返回 0。

```
mysql> SELECT COUNT(*) FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='member';
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

下面这个查询可以让你知道某个数据表使用的是哪一种存储引擎：

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='student';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

2.7.3　从命令行获取元数据

mysqlshow 命令提供的信息与相应的 SHOW 语句相同或近似，这使我们可以从我们的命令行提示符获取数据库和数据表信息。

服务器所管理的数据库：

```
% mysqlshow
```

列出某给定数据库里的数据表：

```
% mysqlshow db_name
```

查看某给定数据表里的数据列信息：

```
% mysqlshow db_name tbl_name
```

查看某给定数据表里的索引信息：

```
% mysqlshow --keys db_name tbl_name
```

查看某给定数据库里的数据表的描述性信息：

```
% mysqlshow --status db_name
```

mysqldump 客户程序能够让你看到 CREATE TABLE 语句（就像 SHOW CREATE TABLE 语句那样）所定义的数据表结构。在使用 mysqldump 程序去查看数据表结构时，千万不要忘记加上--no-data 选项，否则你看到的将是数据表里的数据！

```
% mysqldump --no-data db_name [tbl_name] ...
```

如果你只给出了数据库的名字而没有给出任何数据表的名字，mysqldump 将把该数据库里的所有数据表的结构呈现在你眼前。否则，它将只显示有名字的数据表的结构信息。

在使用 mysqlshow 和 mysqldump 程序时，不要忘记给出必要的连接参数选项，如--host、--user 或--password。

2.8 利用联结操作对多个数据表进行检索

如果只把数据存入数据库，而不对它们进行检索或是用它们来干些什么，这是没有什么意义的。那正是 SELECT 语句的目的所在：帮你找到你需要的数据。与 SQL 语言中的其他语句相比，SELECT 语句应该是最常用的，同时也应该是最不容易掌握的。用来筛选数据行的条件有时会非常复杂并需要比较多个数据表里的数据列。

SELECT 语句的基本语法如下所示：

```
SELECT select_list          # What columns to select
FROM table_list             # The tables from which to select rows
WHERE row_constraint        # What conditions rows must satisfy
GROUP BY grouping_columns # How to group results
ORDER BY sorting_columns  # How to sort results
HAVING group_constraint   # What conditions groups must satisfy
LIMIT count;                # Row count limit on results
```

除了用来表明你想得到什么样的输出的 SELECT 和 select_list 部分，这个语法中的所有东西都是可选的。有些数据库软件要求 FROM 子句也必不可少，但 MySQL 没有那样做，这使你可以在没有引用任何数据表的情况下对表达式进行求值：

```
SELECT SQRT(POW(3,2)+POW(4,2));
```

在第 1 章里，我们重点讨论了只涉及一个数据表的 SELECT 语句，把注意力主要集中在了输出数据列清单和 WHERE、GROUP BY、HAVING 和 LIMIT 子句上。本节将讨论 SELECT 语句最不容易掌握的

方面：编写联结查询，也就是编写从多个数据表检索数据的 SELECT 语句。我们将讨论 MySQL 所支持的联结操作的类型、它们的含义、如何使用它们等。这将帮助你更有效率地用好 MySQL。在许多时候，写出正确的查询命令的关键其实就是正确地把数据表联结在一起。

在使用SELECT语句时经常遇到这样一种情况：那是你一个从没见过的新问题，而编写一个SELECT查询去解决它并不总是那么轻而易举。不过，一旦你把它解决了，以后再遇到类似的问题时就有经验了。只有具备足够经验才能最有效地利用 SELECT 语句，原因很简单，需要用它来解决的问题实在是千变万化。

随着经验的积累，你会发现有很多新问题都可以通过联结操作轻松解决，你会发现自己在这样考虑："噢，这是一个左联结问题。"或者："啊哈，用一个三方联结，它加上一个键字数据列配对限制就可以搞定。"（我希望"经验可以帮上大忙"这句话让你受到鼓舞，否则你发现自己很难用那些古怪的术语思维。）

在接下来的讨论里，有许多用来演示如何使用 MySQL 所支持的联结操作的例子需要用到如下所示的 t1 和 t2 数据表：

```
数据表 t1        数据表 t2
+----+----+    +----+----+
| i1 | c1 |    | i2 | c2 |
+----+----+    +----+----+
|  1 | a  |    |  2 | c  |
|  2 | b  |    |  3 | b  |
|  3 | c  |    |  4 | a  |
+----+----+    +----+----+
```

之所以选择这样两个数据表是因为它们都很小，可以让我们看到联结操作的完整结果。

涉及多个数据表的 SELECT 语句的其他类型是子查询（嵌套在另一个 SELECT 语句里的 SELECT 语句）和 UNION 语句。我们将在 2.9 节和 2.10 节中分别讨论它们。

MySQL 支持的、与涉及多个数据表的检索操作密切相关的其他功能，包括根据某个数据表的内容来删除或刷新另一个数据表里的数据行。比如说，你可能需要从某个数据表里把它在另一个数据表里没有任何匹配的记录全部删掉，或者需要从某个数据表里把几个数据列复制到另一个数据表里去。2.12 节将讨论这些问题。

2.8.1　内联结

如果你在 SELECT 语句的 FROM 子句里列出了多个数据表，并用 INNER JOIN 将它们的名字隔开，MySQL 将执行内联结（inner join）操作，这将通过把一个数据表里的数据行与另一个数据表里的数据行进行匹配来产生结果。比如说，如果你像下面这样把 t1 和 t2 联结起来，t1 里的每一个数据行将与 t2 里的每一个数据行组合：

```
mysql> SELECT * FROM t1 INNER JOIN t2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  1 | a  |  2 | c  |
|  2 | b  |  2 | c  |
|  3 | c  |  2 | c  |
|  1 | a  |  3 | b  |
```

```
|  2 | b  |  3 | b  |
|  3 | c  |  3 | b  |
|  1 | a  |  4 | a  |
|  2 | b  |  4 | a  |
|  3 | c  |  4 | a  |
+----+----+----+----+
```


在这条语句里，SELECT *的含义是，从 FROM 子句所列出的每一个数据表里选取每一个数据列。你也可以把这写成 SELECT t1.*, t2.*：

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2;
```

如果你要选取所有的数据列，或者想按另外一种从左到右的顺序来显示它们，按照你希望的顺序依次列出各数据列（记得用逗号把它们隔开）就行了。

根据某个数据表里的每一个数据行与另一个数据表里的每一个数据行得到全部可能的组合的联结操作叫做生成笛卡儿积（cartesian product）。像这样来联结数据表有可能产生非常多的结果数据行，因为结果数据行的总数将是对每个数据表的数据行个数进行连续乘法而得到的积。假设你有 3 个数据表分别包含 100、200 和 300 个数据行，它们的交叉联结将返回 6 百万个（100 × 200 × 300）数据行。这可是个相当大的数字，而那 3 个数据表都很小。在这种情况下，通常需要增加一条 WHERE 子句以便把结果集压缩到一个更便于管理的尺寸。

如果你增加了一条如下所示的 WHERE 子句，让联结操作只对各数据表里的特定数据列里的值进行匹配，联结操作将只选取那些数据列里的值彼此相等的数据行：

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

CROSS JOIN 和 JOIN 联结类型类似于 INNER JOIN。比如说，下面的语句是等价的：

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 CROSS JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 JOIN t2 WHERE t1.i1 = t2.i2;
```

“,”（逗号）关联操作符的效果与 INNER JOIN 相似：

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
```

请注意，逗号操作符的优先级和其他联结类型不一样，有时还会导致语法错误，而其他联结类型没有这些问题。我个人认为应该尽量避免使用逗号操作符。

INNER JOIN、CROSS JOIN 和 JOIN（注意，不包括逗号操作符）还支持另外几种用来表明如何对数据表里的数据列进行匹配的语法变体，如下所示。

- 用一条 ON 子句代替 WHERE 子句。下面是一个使用了 ON 子句的 INNER JOIN 操作的例子：

  ```
  SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
  ```

  不管被联结的数据列是否同名，ON 子句都可以使用。
- 另一种语法是使用一个 USING()子句，它在概念上类似于 ON 子句，但要求被联结的数据列必须是同名的。比如说，如下所示的查询对 mytbl1.b 和 mytbl2.b 进行了联结：

```
SELECT mytbl1.*, mytbl2.* FROM mytbl1 INNER JOIN mytbl2 USING (b);
```

2.8.2　避免歧义：如何在联结操作中给出数据列的名字

在 SELECT 语句里给出的数据列的名字不允许产生歧义，而且必须来自 FROM 子句里的某个数据表。如果只涉及一个数据表，就不会产生歧义，被列出的所有数据列都来自那个数据表。如果涉及多个数据表，只在一个表中出现的数据列名称不会产生歧义。不过，如果某个数据列的名字出现在多个数据表里，在引用这个数据列时必须使用 tbl_name1.col_name 语法给它加上一个数据表的名字来表明它来自哪一个数据表。假设你的 mytbq1 数据表里包含数据列 a 和 b，mytbl2 数据表里包含数据列 b 和 c。对于这种情况，引用 a 或 c 都不会产生歧义，但在引用 b 时就必须把它限定为 mytbl1.b 或 mytbl2.b 了：

```
SELECT a, mytbl1.b, mytbl2.b, c FROM mytbl1 INNER JOIN mytbl2 ... ;
```

有时候，用数据表的名字进行限定仍不足以解决数据列的歧义问题。比如说，假设你正在进行一个自联结操作（也就是把一个数据表与它本身联结起来），这是在查询命令里多次用到同一个数据表（而不是用到多个数据表）的问题，用数据表的名字来限定数据列帮不上你的忙。在遇到这类问题时，数据表别名可以让你达到目的。你只需给该数据表的某个实例起一个别名，就可以通过 alias_name1.col_name 语法来引用该实例里的数据列了。下面的查询命令将把一个数据表与它自身联结起来。为了消除在引用数据列时可能产生的歧义问题，我们给该数据表的一个实例分配了一个别名：

```
SELECT mytbl.col1, m.col2 FROM mytbl INNER JOIN mytbl AS m
WHERE mytbl.col1 > m.col1;
```

2.8.3　左联结和右联结（外联结）

内联结只显示在两个数据表里都能找到匹配的数据行。外联结除了显示同样的匹配结果，还可以把其中一个数据表在另一个数据表里没有匹配的数据行也显示出来。外联结分左联结和右联结两种。本小节里的绝大多数例子使用的是 LEFT JOIN，意思是把左数据表在右数据表里没有匹配的数据行也显示出来。RIGHT JOIN 刚好与此相反，它将把右数据表在左数据表里没有匹配的数据行也显示出来，数据表的角色调换了一下。

LEFT JOIN 的工作情况是这样的：你给出用来匹配两个数据表里的数据行的数据列，当来自左数据表的某个数据行与来自右数据表的某个数据行匹配时，那两个数据行的内容就会被选取为一个输出数据行；如果来自左数据表的某个数据行在右数据表里找不到匹配，它也会被选取为一个输出数据行，此时与它联结的是一个来自右数据表的“假”数据行，这个“假”数据行的所有数据列都包含 NULL 值。

换句话说，在 LEFT JOIN 操作里，来自左数据表的每一个数据行在结果集里都有一个对应的数据行，不管它在右数据表里有没有匹配。在结果集里，在右数据表里没有匹配的结果数据行有这样一个特征：来自右数据表的所有数据列都是 NULL 值。这个特征可以让你知道右数据表里缺少了哪些数据行。这是一个既有趣又重要的特征，能够反映出各种各样的问题。还没有为哪些顾客指派服务代表？哪些库存商品一件也没卖出去？或者，就我们的 sampdb 数据库而言，哪些学生没有参加过某次特定的考试？哪些学生在 absence 数据表里没有任何记录（也就是说，哪些学生是全勤的）？

我们仍以刚才的 t1 和 t2 数据表为例：

```
数据表 t1       数据表 t2
+----+----+    +----+----+
| i1 | c1 |    | i2 | c2 |
+----+----+    +----+----+
|  1 | a  |    |  2 | c  |
|  2 | b  |    |  3 | b  |
|  3 | c  |    |  4 | a  |
+----+----+    +----+----+
```

如果我们对这两个数据表进行内联结以匹配 t1.i1 和 t2.i2，我们只能得到与值 2 和 3 相匹配的输出，因为只有它们才是这两个数据表里都有的值：

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

左联结操作会为 t1 数据表里每一个数据行生成一个输出数据行，不管它在 t2 里有没有匹配。为了写出相应的左联结查询命令，只要在上面这条语句里把两个数据表之间的 INNER JOIN 替换为 LEFT JOIN 即可：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
|  2 | b  |    2 | c    |
|  3 | c  |    3 | b    |
+----+----+------+------+
```

请注意，结果集里多出了一个 t1.i1 取值为 1 的输出行，它在 t2 里没有任何匹配。在这个输出行里，来自 t2 的所有数据列的值都是 NULL。

在使用 LEFT JOIN 时需要注意这样一个问题：如果右数据表里的数据列没有被全部定义成 NOT NULL，结果集里的数据行就可能不能反映真实情况。比如说，如果右数据表里的某个数据列允许取值为 NULL 并被收录在结果集里，用这个数据列里的 NULL 值来判断“在右数据表里没有匹配”就可能出问题。

正如刚才提到的那样，RIGHT JOIN 和 LEFT JOIN 的行为相似，只是把数据表的角色调换了一下而已。下面两条语句是等价的：

```
SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t2 RIGHT JOIN t1 ON t1.i1 = t2.i2;
```

下面的讨论将围绕 LEFT JOIN 展开，但同样适用于 RIGHT JOIN，只要调换一下数据表的角色就行了。

LEFT JOIN 很有用，尤其是在你只想找出在右数据表里没有匹配的左数据表的行时。具体地说，增加一条 WHERE 子句，让它把右数据表的数据列全部是 NULL 值的（也就是那些在一个数据表里有匹配，但在另一个数据表里没有匹配）数据行筛选出来：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
+----+----+------+------+
```

一般来说，在编写这样的查询命令时，你真正感兴趣的东西是左数据表里没被匹配的值。把来自右数据表值为 NULL 的数据列显示出来没有什么意义，你可以把它们从将被输出的数据列清单里剔除：

```
mysql> SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
|  1 | a  |
+----+----+
```

类似于 INNER JOIN，LEFT JOIN 也可以写成使用一个 ON 子句或一个 USING() 子句来给出匹配条件。与 INNER JOIN 的情况相同，ON 子句不管被联结的数据列是否同名都可以使用，USING() 子句要求被联结的数据列必须有同样的名字。

LEFT JOIN 还有几种同义词和变体。LEFT OUTER JOIN 是 LEFT JOIN 的一个同义词。MySQL 还支持一种 ODBC 风格的 LEFT OUTER JOIN 表示法，该表示法使用了花括号和 OJ（outer join）：

```
mysql> SELECT t1.* FROM { OJ t1 LEFT OUTER JOIN t2 ON t1.i1 = t2.i2 }
    -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
|  1 | a  |
+----+----+
```

NATURAL LEFT JOIN 类似于 LEFT JOIN。它将按照 LEFT JOIN 规则对左、右数据表里所有同名的数据列进行匹配。（也就是说，它不需要你给出任何 ON 或 USING 子句。）

正如刚才提到的那样，LEFT JOIN 非常适合用来解决“哪些值是缺失的”这个问题。接下来，我们将把这个原则应用到 sampdb 数据库上，来考虑一个比刚才那些使用 t1 和 t2 数据表的例子更复杂的例子。

对于记录学生学习情况的项目（请参见第 1 章），我们有一个 student 数据表来记录学生、一个 grade_event 数据表来记录已经发生的考试或测验事件，还有一个 score 数据表来列出每位学生的每次考试或测验成绩。如果某个学生在某次考试或小测验的当天生病了，score 数据表里就不会有该名学生在那次事件的成绩。因故误考的学生需要参加补考，可怎样才能把这些缺失的数据行找出来呢？

解决问题的关键是找出哪些学生在某次考试里没有任何成绩，这种查找需要为每一个考试事件执行一遍。换个技术味儿更浓的说法，我们需要找出哪些学生和事件的组合在 score 数据表里没有出现过。这种“哪些值没有出现过”的说法是需要对数据库进行 LEFT JOIN 查询的一种标志。这个联结查询可不像前几个例子那么简单：我们这次要找的不是哪个值没有出现在某个数据列，我们要找的是一些由两个数据列构成的组合。我们想要的组合是所有的“学生+事件”，这可以通过把 student 数据表

和 grade_event 数据表联结起来得到：

```
FROM student INNER JOIN grade_event
```

接下来，我们需要把这个联结结果与 score 数据表用一个 LEFT JOIN 操作联结起来以找出匹配的学生 ID/考试 ID 组合：

```
FROM student INNER JOIN grade_event
     LEFT JOIN score ON student.student_id = score.student.id
                    AND grade_event.event_id = score.event_id
```

请注意 ON 子句，score 数据表里的数据行将按照 ON 子句给出的匹配条件与刚才那个 INNER JOIN 操作的结果集进行左联结。这是解决这个问题的关键。这次 LEFT JOIN 操作将为对 student 和 grade_event 数据表进行内联结而得到的每一个数据行生成一个数据行，即使没有对应的 score 表里的行。在这次 LEFT JOIN 的结果集里，缺少考试分数的数据行可以根据来自 score 数据表的数据列将全部是 NULL 值这一事实被找出来。我们可以通过在 WHERE 子句里增加一个条件的办法把这些数据行找出来。来自 score 数据表的任何数据列都可以用在这个 WHERE 条件里，但因为我们正在寻找的是缺少考试分数的数据行，对 score 数据列进行测试当然最顺理成章：

```
WHERE score.score IS NULL
```

我们还可以用一个 ORDER BY 子句对结果进行排序。有两种顺序最符合逻辑：按每个学生参加的考试和按参加每次考试的学生，我选的是前者：

```
ORDER BY student.student_id, grade_event.event_id
```

现在，只需列出我们想在输出报告里看到的数据列就可以大功告成了。下面是最终的 SELECT 语句：

```
SELECT
  student.name, student.student_id,
  grade_event.date, grade_event.event_id, grade_event.category
FROM
  student INNER JOIN grade_event
  LEFT JOIN score ON student.student_id = score.student_id
                  AND grade_event.event_id = score.event_id
WHERE
  score.score IS NULL
ORDER BY
  student.student_id, grade_event.event_id;
```

下面是运行这个查询所得到的结果：

```
+-----------+------------+------------+----------+----------+
| name      | student_id | date       | event_id | category |
+-----------+------------+------------+----------+----------+
| Megan     |          1 | 2008-09-16 |        4 | Q        |
| Joseph    |          2 | 2008-09-03 |        1 | Q        |
| Katie     |          4 | 2008-09-23 |        5 | Q        |
| Devri     |         13 | 2008-09-03 |        1 | Q        |
| Devri     |         13 | 2008-10-01 |        6 | T        |
| Will      |         17 | 2008-09-16 |        4 | Q        |
| Avery     |         20 | 2008-09-06 |        2 | Q        |
| Gregory   |         23 | 2008-10-01 |        6 | T        |
| Sarah     |         24 | 2008-09-23 |        5 | Q        |
```

```
| Carter    |         27 | 2008-09-16 |        4 | Q        |
| Carter    |         27 | 2008-09-23 |        5 | Q        |
| Gabrielle |         29 | 2008-09-16 |        4 | Q        |
| Grace     |         30 | 2008-09-23 |        5 | Q        |
+-----------+------------+------------+----------+----------+
```

这里还有一个小细节需要解释一下。我们可以在输出报告里看到 student_id 和 event_id。student_id 数据列在 student 和 score 数据表里都有，所以你们也许会认为把输出列命名为 student.student_id 或 score.student_id 没什么区别。事实却并非如此：我们之所以能找到我们感兴趣的数据行，是因为那个 LEFT JOIN 操作所返回的 score 数据表里的数据列全部都是 NULL 值。如果选择 score.student_id，在输出报告里就会有一个全部都是 NULL 值的数据列。在决定显示来自哪个数据表的 event_id 数据列时也需要考虑这个因素：它在 grade_event 和 score 数据表里都出现了，但因为 score.event_id 值将全部是 NULL，所以在查询命令里需要选择 grade_event.event_id 作为输出列。

2.9　用子查询进行多数据表检索

MySQL 支持子查询（subquery），也就是把一条 SELECT 语句用括号括起来嵌入另一个 SELECT 语句。下面这个例子从 grade_event 数据表里找出对应于考试（'T'）的 event_id，再用它们去选取那些考试的成绩：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

子查询可以返回以下不同类型的信息。

- 标量子查询将返回一个值。
- 数据列子查询将返回一个由一个或多个值构成的数据列。
- 数据行子查询将返回一个由一个或多个值构成的数据行。
- 数据表子查询将返回一个由一个或多个数据行构成的数据表，数据行可以由一个或多个数据列构成。

子查询的结果可以用以下不同的办法进行测试。

- 标量子查询的结果可以用诸如“=”或“<”之类的相对比较操作符进行求值。
- 可以用 IN 和 NOT IN 操作符来测试某给定值是否包含在子查询的结果集里。
- 可以用 ALL、ANY 和 SOME 操作符把某给定值与子查询的结果集进行比较。
- 可以用 EXISTS 和 NOT EXISTS 操作符来测试子查询的结果集是否为空。

标量子查询是最严格的，它只产生一个值。也正是因为如此，标量子查询的适用范围也最大。从理论上讲，标量子查询可以出现在允许使用标量操作数的任何地方，你可以把它用做表达式里的一个操作数或函数的输入参数，还可以让它们出现在输出数据列的清单里。数据列、数据行和数据表子查询可以返回更多的信息，但只能用在不要求必须使用单个值的上下文里。

子查询既可以与主查询相关——需要引用和依赖于主查询里的值，也可以与之不相关。

除了 SELECT 语句，子查询还可以用在其他语句里。不过，如果把子查询用在一条会改变数据表内容的语句（INSERT、REPLACE、DELETE、UPDATE、LOAD DATA）里，目前还必须遵守这样一条限制：子查询不允许引用正在被修改的数据表。

有些子查询可以被改写为联结查询。如果你编写的查询命令有可能被拿到一个版本较早的MySQL服务器上去运行，或者如果你想比较一下MySQL优化器对联结查询和子查询的优化效果，就应该多掌握一些子查询的改写技巧。

在接下来的几节里，我们将讨论几种可以用来测试子查询结果的操作，以及如何编写与主查询相关的子查询，如何把子查询改写为联结查询。

2.9.1 子查询与关系比较操作符

=、<>、>、>=、<和<=操作符用来进行相对值比较。它们可以和标量子查询配合使用：用子查询返回的值来构造一个检索条件，再由主查询根据该条件做进一步检索。比如说，如果你想查看'2008-09-23'那天的小测验成绩，可以用一个标量子查询来确定那次小测验的event_id，再让主查询使用这个event_id去检索score数据表：

```
SELECT * FROM score
WHERE event_id =
(SELECT event_id FROM grade_event
   WHERE date = '2008-09-23' AND category = 'Q');
```

这种形式的语句有一个共同点：子查询的前面有一个值和一个相对比较操作符，子查询的这种用法要求它只返回一个值，也就是说，它必须是标量子查询；如果它返回了多个值，整条语句将以失败告终。有时候，为了确保子查询返回且只返回一个值，有必要用LIMIT 1对子查询的结果加以限制。

如果你遇到的问题可以通过在WHERE子句里使用某个聚合函数去解决，就应该考虑用标量子查询和相对比较操作符。比如说，如果你想知道president数据表里的哪位总统出生得最早，你可能会写出如下所示的语句：

```
SELECT * FROM president WHERE birth = MIN(birth);
```

可这个办法是行不通的——你不能在WHERE子句里使用聚合函数。WHERE子句的用途是确定应该选取哪些数据行，但MIN()的值只有在选取完数据行之后才能确定下来。不过，你可以像下面这样用一个子查询把最早的出生日期检索出来：

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

其他的聚合函数可以用来解决类似的问题。下面这条语句使用了一个子查询来选取在某次考试高于平均分数的分数：

```
SELECT * FROM score WHERE event_id = 5
AND score > (SELECT AVG(score) FROM score WHERE event_id = 5);
```

如果子查询返回的是一个数据行，你可以用一个数据行构造器把一组值（一条临时创建的记录）与子查询的结果进行比较。下面这条语句将返回一个数据行，它可以告诉我们哪几位总统和John Adams出生在同一个城市和州：

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) =
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Adams' AND first_name = 'John');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
```

```
+-----------+-------------+-----------+-------+
| Adams     | John        | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

你也可以使用 ROW(city, state)表示法，它等价于(city, state)。它们都可以用来生成一条临时记录。

2.9.2 IN 和 NOT IN 子查询

如果你的子查询将返回多个数据行，你可以用 IN 和 NOT IN 操作符来构造主查询的检索条件。IN 和 NOT IN 操作符的用途是测试一个给定的比较值有没有出现在一个特定的集合里。只要主查询里的数据行与子查询所返回的任何一个数据行匹配，IN 操作符的比较结果就将是 true。如果主查询里的数据行与子查询所返回的所有数据行都不匹配，NOT IN 操作符的比较结果将是 true。下面两条语句分别使用了 IN 和 NOT IN 操作符来查找在 absence 数据表里有缺勤记录和没有缺勤记录（也就是全勤）的学生：

```
mysql> SELECT * FROM student
    -> WHERE student_id IN (SELECT student_id FROM absence);
+-------+-----+------------+
| name  | sex | student_id |
+-------+-----+------------+
| Kyle  | M   |          3 |
| Abby  | F   |          5 |
| Peter | M   |         10 |
| Will  | M   |         17 |
| Avery | F   |         20 |
+-------+-----+------------+
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
...
```

IN 和 NOT IN 操作符还可以用在将返回多个数据列的子查询里。换句话说，你可以在数据表子查询里使用它们。此时，你需要使用一个数据行构造器来给出将与各数据列比较的比较值。

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) IN
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

IN 和 NOT IN 操作符其实是= ANY 和< > ALL 的同义词；详见下一节里的讨论。

2.9.3 ALL、ANY 和 SOME 子查询

ALL 和 ANY 操作符的常见用法是结合一个相对比较操作符对一个数据列子查询的结果进行测试。它们测试比较值是否与子查询所返回的全部或一部分值匹配。比如说，如果比较值小于或等于子查询所返回的每一个值，<= ALL 将是 true；只要比较值小于或等于子查询所返回的任何一个值，<= ANY 将是 true。SOME 是 ANY 的一个同义词。

下面这条语句用来检索最早出生的总统，具体做法是选取出生日期小于或等于 president 数据表里的所有出生日期（只有最早的出生日期满足这一条件）的那个数据行：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ALL (SELECT birth FROM president);
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
+------------+------------+------------+
```

下面这条语句的用处就不大了，它将返回所有的数据行，因为对于每个日期，至少有一个日期（它本身）大于或等于它：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ANY (SELECT birth FROM president);
+------------+---------------+------------+
| last_name  | first_name    | birth      |
+------------+---------------+------------+
| Washington | George        | 1732-02-22 |
| Adams      | John          | 1735-10-30 |
| Jefferson  | Thomas        | 1743-04-13 |
| Madison    | James         | 1751-03-16 |
| Monroe     | James         | 1758-04-28 |
...
```

当 ALL、ANY 或 SOME 操作符与“=”比较操作符配合使用时，子查询可以是一个数据表子查询。此时，你需要使用一个数据行构造器来提供与子查询所返回的数据行进行比较的比较值。

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) = ANY
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

前一节里提到过，IN 和 NOT IN 操作符是= ANY 和< > ALL 的简写。也就是说，IN 操作符的含义是“等于子查询所返回的某个数据行”，NOT IN 操作符的含义是“不等于子查询所返回的任何数据行”。

2.9.4 EXISTS 和 NOT EXISTS 子查询

EXISTS 和 NOT EXISTS 操作符只测试某个子查询是否返回了数据行：如果是，EXISTS 将是 true，NOT EXISTS 将是 false。下面两条语句演示了这两个操作符的用法。如果 absence 数据表是空白的，第一条语句将返回 0，第二条语句将返回 1：

```
SELECT EXISTS (SELECT * FROM absence);
SELECT NOT EXISTS (SELECT * FROM absence);
```

EXISTS 和 NOT EXISTS 操作符在与主查询相关的子查询里比较常见。详见 2.9.5 节里的讨论。

在使用 EXISTS 和 NOT EXISTS 操作符时，子查询通常使用“*”作为输出数据列清单。这两个操作符是根据子查询是否返回了数据行来判断真假的，不关心数据行所包含的内容是什么，所以没必要明确地列出数据列的名字。事实上，你可以在子查询的数据列选取清单里写出任何东西，但如果你想确保在子查询成功时返回一个 true 值的话，把它写成 SELECT 1 当然要比把它写成 SELECT *更保险。

2.9.5 与主查询相关的子查询

如下所示，子查询可以与主查询相关，也可以与之无关。

- 与主查询无关的子查询不引用主查询里的任何值。与主查询无关的子查询可以脱离主查询作为一条独立的查询命令去执行。比如说，下面这条语句里的子查询就是与主查询无关的，它只引用了数据表 t1，没有引用数据表 t2：

  ```
  SELECT j FROM t2 WHERE j IN (SELECT i FROM t1);
  ```

- 与主查询相关的子查询需要引用主查询里的值，所以必须依赖于主查询。因为这种联系，与主查询相关的子查询不能脱离主查询作为一条独立的查询命令去执行。比如说，下面这条语句里的子查询的作用是把t2数据表中数据列i的每一个值与数据表t1中数据列j的值相匹配：

  ```
  SELECT j FROM t2 WHERE (SELECT i FROM t1 WHERE i = j);
  ```

与主查询相关的子查询通常用在 EXISTS 和 NOT EXISTS 子查询里，这类子查询主要用来在某个数据表里查找在另一个数据表里有匹配数据行或没有匹配数据行的数据行。与主查询相关的子查询的工作情况是：把值从主查询传递到子查询，看它们是否满足在子查询里给出的条件。因此，如果数据列的名字有可能引起歧义的话（在不同的数据表里有同名的数据列），千万不要忘记使用数据表的名字来限定数据列。

下面的 EXISTS 子查询用来确定数据表之间的匹配——也就是在两个数据表里都有的值，而整个语句的用途是选取在 absence 数据表里至少有一条缺勤记录的学生：

```
SELECT student_id, name FROM student WHERE EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

NOT EXISTS 操作符用来寻找不匹配的值（在一个数据表里有但在其他数据表里没有的值）。下面这条语句将把没有缺勤记录的学生查找出来：

```
SELECT student_id, name FROM student WHERE NOT EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

2.9.6 FROM 子句中的子查询

子查询可以用在 FROM 子句里以生成一些值。在这种情况里，子查询的结果在行为上就像是一个

数据表。FROM 子句里的子查询可以参加关联操作，它的值可以在 WHERE 子句里被测试，等等。在使用这种子查询的时候，你必须提供一个数据表别名作为子查询的结果的名字：

```
mysql> SELECT * FROM (SELECT 1, 2) AS t1 INNER JOIN (SELECT 3, 4) AS t2;
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
```


2.9.7 把子查询改写为联结查询

有相当一部分使用了子查询的查询命令可以被改写为一个联结查询。有时候，联结查询要比子查询的执行效率更高，所以把子查询改写为联结查询是个不坏的主意。如果一条使用了子查询的 SELECT 语句需要很长时间执行完毕，就应该尝试把它改写为一个联结查询，看它是不是执行得更好。本节将介绍如何这样做。

1. 如何改写用来选取匹配值的子查询

下面这条示例语句包含一个子查询，它将从 score 数据表把考试（不包括小测验）成绩筛选出来：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

这条语句也可以写成不使用子查询的样子，只要把它转换为一个简单的联结操作就行了：

```
SELECT score.* FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id WHERE grade_event.category = 'T';
```

再来看一个例子。下面这个查询只选取女生们的考试成绩：

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = 'F');
```

这条语句可以被改写为下面的联结查询：

```
SELECT score.* FROM score INNER JOIN student
ON score.student_id = student.student_id WHERE student.sex = 'F';
```

看出其中的规律了吗？这几个例子里的子查询语句都是如下所示的形式：

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2a FROM table2 WHERE column2b = value);
```

这类查询可以被转换为如下所示的联结查询：

```
SELECT table1.* FROM table1 INNER JOIN table2
ON table1.column1 = table2.column2a WHERE table2.column2b = value;
```

在某些场合，子查询和关联查询可能返回不同的结果。这发生在 table2 包含多个 column2a 的实例的时候。子查询只为每个 column2a 值生成一个实例，联结操作会把它们都生成出来并导致在其输出里出现重复的数据行。如果你想防止这种重复，在编写联结查询命令时就要用 SELECT DISTINCT 来代替 SELECT。

2. 如何改写用来选取非匹配（缺失）值的子查询

子查询语句的另一种常见用法是检索在一个数据表里有、在另一个数据表里没有的值。前面看到过，与“哪些值没有出现”有关的问题通常都可以用 LEFT JOIN 来解决。下面是一个我们曾经见过的

子查询，它用来测试哪些学生没有出现在 absence 数据表里（查找全勤的学生）：

```
SELECT * FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

这条语句可以被改写为如下所示的 LEFT JOIN 查询命令：

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.student_id
WHERE absence.student_id IS NULL;
```

一般来说，如果子查询语句有如下所示的形式：

```
SELECT * FROM table1
WHERE column1 NOT IN (SELECT column2 FROM table2);
```

就可以把它改写为下面这样的联结查询：

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL;
```

注意，这里需要假设 table2.column2 被定义成 NOT NULL。

与 LEFT JOIN 相比，子查询具有比较直观和容易理解的优点。绝大多数人都可以毫无困难地理解“没被包含在……里面”的含义，它不是数据库编程技术带来的新概念。“左联结”这个概念就不同了，就算是数据库方面的专业人士也不见得都能把它解释透彻。

2.10　用 UNION 语句进行多数据表检索

如果想把多个查询的结果合并在一起创建一个结果集，可以使用 UNION 语句来做这件事。本节中的例子假设你有 t1、t2 和 t3 这 3 个数据表，如下所示：

```
mysql> SELECT * FROM t1;
+------+-------+
| i    | c     |
+------+-------+
|    1 | red   |
|    2 | blue  |
|    3 | green |
+------+-------+
mysql> SELECT * FROM t2;
+------+------+
| i    | c    |
+------+------+
|   -1 | tan  |
|    1 | red  |
+------+------+
mysql> SELECT * FROM t3;
+------------+------+
| d          | i    |
+------------+------+
| 1904-01-01 |  100 |
| 2004-01-01 |  200 |
| 2004-01-01 |  200 |
+------------+------+
```

数据表 t1 和 t2 有整数和字符数据列，t3 有日期和整数数据列。为了编写一条 UNION 语句把多个检索合并在一起，需要先写出几条 SELECT 语句（它们检索出来的数据列个数相同），然后把关键字 UNION 放到它们之间。比如说，下面这条语句将选取各数据表的整数数据列并把它们合并在一起：

```
mysql> SELECT i FROM t1 UNION SELECT i FROM t2 UNION SELECT i FROM t3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|   -1 |
|  100 |
|  200 |
+------+
```

UNION 语句有以下特性。

数据列的名字和数据类型。UNION 结果集里的数据列名字来自第一个 SELECT 语句里的数据列的名字。UNION 中的第二个和再后面的 SLEECT 语句必须选取个数相同的数据列，但各有关数据列不必有同样的名字或数据类型。（一般来说，同一条 UNION 语句里的各有关数据列会是相同的数据类型，但这并不是一项要求。如果它们不一样，MySQL 会进行必要的类型转换。）数据列是根据位置而不是根据名字进行匹配的，这也正是下面两条语句会返回不同结果的原因，虽然它们从两个数据表选取的是同样的值：

```
mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3;
+------+------------+
| i    | c          |
+------+------------+
|    1 | red        |
|    2 | blue       |
|    3 | green      |
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
+------+------------+
mysql> SELECT i, c FROM t1 UNION SELECT d, i FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

在每条语句里，结果中的每个数据列的类型是根据被选取的值而确定的。在第一条语句里，我们为第二个数据列选取的是字符串的日期，结果是一个字符串数据列。在第二条语句里，为第一个数据列选取的是整数和日期，为第二个数据列选取的是字符串和整数。在这两种情况里，结果都是一个字符串数据列。

重复数据行的处理。在默认的情况下，UNION 将从结果集里剔除重复的数据行：

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

t1 和 t2 都有一个包含着 1 和'red'值的数据行，但输出结果里只有一个这样的数据行。此外，t3 有两个包含'2004-01-01'和 200 的数据行，其中之一被剔除了。

UNION DISTINCT 是 UNION 的同义词，它们都只保留不重复的数据行。

如果你想保留重复的数据行，需要把每个 UNION 都改为 UNION ALL。

```
mysql> SELECT * FROM t1 UNION ALL SELECT * FROM t2 UNION ALL SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1          | red   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
| 2004-01-01 | 200   |
+------------+-------+
```

如果把 UNION（或 UNION DISTINCT）和 UNION ALL 混杂在一起使用，每个 UNION（或 UNION DISTINCT）操作将优先于它左边的任何 UNION ALL 操作。

ORDER BY 和 LIMIT 处理。如果你想将 UNION 结果作为一个整体进行排序，需要用括号把每一个 SELECT 语句括起来并在最后一个 SELECT 语句的后面加上一个 ORDER BY 子句。注意，因为 UNION 的结果数据列的名字来自第一个 SELECT 语句，所以你在 ORDER BY 子句里必须引用那些名字而不是引用来自最后一个 SELECT 语句的数据列名字：

```
mysql> (SELECT i, c FROM t1) UNION (SELECT i, d FROM t3)
    -> ORDER BY c;
+------+------------+
| i    | c          |
+------+------------+
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
|    2 | blue       |
|    3 | green      |
|    1 | red        |
+------+------------+
```

如果某个排序数据列有别名，位于 UNION 语句末尾的 ORDER BY 子句必须引用那个别名。此外，ORDER BY 不能引用数据表的名字。如果为了排序而需要以 *table_name.col_name* 的形式给出一个来

自第一个 SELECT 语句的数据列名字，必须给那个数据列起一个别名并在 ORDER BY 子句里引用那个别名。

类似地，如果你想限制 UNION 语句所输出的数据行的个数，可以在语句末尾加上一个 LIMIT 子句。


```
mysql> (SELECT * FROM t1) UNION (SELECT * FROM t2) UNION (SELECT * FROM t3)
    -> LIMIT 2;
+------+-------------+
| i    | c           |
+------+-------------+
| 1    | red         |
| 2    | blue        |
+------+-------------+
```

在 UNION 语句中，ORDER BY 和 LIMIT 还可以用在被括号括起来的各条 SELECT“子句”里。此时，它们将只作用于那条 SELECT 语句：

```
mysql> (SELECT * FROM t1 ORDER BY i LIMIT 2)
    -> UNION (SELECT * FROM t2 ORDER BY i LIMIT 1)
    -> UNION (SELECT * FROM t3 ORDER BY d LIMIT 2);
+------------+-------------+
| i          | c           |
+------------+-------------+
| 1          | red         |
| 2          | blue        |
| -1         | tan         |
| 1904-01-01 | 100         |
| 2004-01-01 | 200         |
+------------+-------------+
```

在用括号括起来的各个 SELECT 语句里的 ORDER BY 只能在 LIMIT 也出现时才能使用，以确定 LIMIT 将作用于哪些数据行。此时，它只影响那条 SELECT 语句的结果，不影响最终的 UNION 结果里的数据行的先后顺序。

如果你想在一组结构相同的 MyISAM 数据表上运行 UNION 类型的查询，建议你创建一个 MERGE 数据表并查询，因为编写针对单个 MERG 数据表的查询命令一般都要比编写相应的 UNION 语句容易一些。对 MERGE 数据表进行查询类似于用一条 UNOIN 语句从它的各成员数据表里把相应的数据列选取出来。具体地说，MERGE 数据表上的 SELECT 语句相当于 UNION ALL（不剔除重复的数据行），SELECT DISTINCT 相当于 UNION 或 UNION DISTINCT（剔除重复的数据行）。

2.11 使用视图

视图是一种虚拟的数据表，它们的行为和数据表一样，但并不真正包含数据。它们是用底层（真正的）数据表或其他视图定义出来的“假”数据表，用来提供查看数据表数据的另一种方法，这通常可以简化应用程序。

本节重点介绍视图的一些应用。这里没有讨论 DEFINER 子句，这个子句是存储程序和视图都使用的，它可以用来从信息安防的角度对视图数据的访问情况进行控制。关于 DEFINER 子句的详细讨论参见 4.5 节。

如果要选取某给定数据表的数据列的一个子集，把它定义为一个简单的视图是最方便的做法。比

如说，假设你经常需要从 president 数据表选取 last_name、first_name、city 和 state 等几个数据列，但不想每次都必须写出所有这些数据列，如下所示：

```
SELECT last_name, first_name, city, state FROM president;
```

你也不想使用 SELECT *，这虽然简单，但用*检索出来的数据列不都是你想要的。解决这个矛盾的办法是定义一个视图，让它只包括你想要的数据列：

```
CREATE VIEW vpres AS
SELECT last_name, first_name, city, state FROM president;
```

这个视图就像一个“窗口”，从中只能看到你想看的数据列。这意味着你可以在这个视图上使用 SELECT *，而你看到的将是你在视图定义里给出的那些数据列：

```
mysql> SELECT * FROM vpres;
+------------+---------------+---------------------+-------+
| last_name  | first_name    | city                | state |
+------------+---------------+---------------------+-------+
| Washington | George        | Wakefield           | VA    |
| Adams      | John          | Braintree           | MA    |
| Jefferson  | Thomas        | Albemarle County    | VA    |
| Madison    | James         | Port Conway         | VA    |
| Monroe     | James         | Westmoreland County | VA    |
...
```

如果你在查询某个视图时还使用了一个 WHERE 子句，MySQL 将在执行该查询时把它添加到那个视图的定义上以进一步限制其检索结果：

```
mysql> SELECT * FROM vpres WHERE last_name = 'Adams';
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Adams     | John        | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

在查询视图时还可以使用 ORDER BY、LIMIT 等子句，其效果与查询一个真正的数据表时的情况一样。

在使用视图时，你只能引用在该视图的定义里列出的数据列。也就是说，如果底层数据表里的某个数据列没在视图的定义里，你在使用视图的时候就不能引用它：

```
mysql> SELECT * FROM vpres WHERE suffix <> '';
ERROR 1054 (42S22): Unknown column 'suffix' in 'where clause'
```

在默认的情况下，视图里的数据列的名字与 SELECT 语句里列出的输出数据列相同。如果你想明确地改用另外的数据列名字，需要在定义视图时在视图名字的后面用括号列出那些新名字：

```
mysql> CREATE VIEW vpres2 (ln, fn) AS
    -> SELECT last_name, first_name FROM president;
```

此后，当你使用这个视图时，必须使用在括号里给出的数据列名字，而非 SELECT 语句里的名字：

```
mysql> SELECT last_name, first_name FROM vpres2;
ERROR 1054 (42S22) at line 1: Unknown column 'last_name' in 'field list'
mysql> SELECT ln, fn FROM vpres2;
```

```
+------------+---------------+
| ln         | fn            |
+------------+---------------+
| Washington | George        |
| Adams      | John          |
| Jefferson  | Thomas        |
| Madison    | James         |
| Monroe     | James         |
...
```

视图可以用来自动完成必要数学运算。1.4.9 节中的第 6 小节编写了一条语句来确定各位总统去世时的年龄，我们可以把同样的数学运算放到一个视图定义里进行：

```
mysql> CREATE VIEW pres_age AS
    -> SELECT last_name, first_name, birth, death,
    -> TIMESTAMPDIFF(YEAR, birth, death) AS age
    -> FROM president;
```

这个视图包含一个 age 数据列，它被定义成一个运算，从这个视图选取该数据列将检索出这个运算的结果：

```
mysql> SELECT * FROM pres_age;
+------------+---------------+------------+------------+------+
| last_name  | first_name    | birth      | death      | age  |
+------------+---------------+------------+------------+------+
| Washington | George        | 1732-02-22 | 1799-12-14 |   67 |
| Adams      | John          | 1735-10-30 | 1826-07-04 |   90 |
| Jefferson  | Thomas        | 1743-04-13 | 1826-07-04 |   83 |
| Madison    | James         | 1751-03-16 | 1836-06-28 |   85 |
| Monroe     | James         | 1758-04-28 | 1831-07-04 |   73 |
...
```

通过把年龄计算工作放到视图定义里完成，我们就用不着再在查询年龄值时写出那个公式了。有关的细节都隐藏在了视图里。

同一个视图可以涉及多个数据表，这使得联结查询的编写和运行变得更容易。下面定义的视图对 score、student 和 grade_event 数据表进行了联结查询：

```
mysql> CREATE VIEW vstudent AS
    -> SELECT student.student_id, name, date, score, category
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id;
```

当你从这个视图选取数据时，MySQL 将执行相应的联结查询并从多个数据表返回信息：

```
mysql> SELECT * FROM vstudent;
+------------+-----------+------------+-------+----------+
| student_id | name      | date       | score | category |
+------------+-----------+------------+-------+----------+
|          1 | Megan     | 2008-09-03 |    20 | Q        |
|          3 | Kyle      | 2008-09-03 |    20 | Q        |
|          4 | Katie     | 2008-09-03 |    18 | Q        |
|          5 | Abby      | 2008-09-03 |    13 | Q        |
|          6 | Nathan    | 2008-09-03 |    18 | Q        |
|          7 | Liesl     | 2008-09-03 |    14 | Q        |
```

```
|          8 | Ian       | 2008-09-03 |    14 | Q        |
...
```

这个视图可以让我们轻而易举地根据名字检索出某个学生的考试成绩：

```
mysql> SELECT * FROM vstudent WHERE name = 'emily';
+------------+-------+------------+-------+----------+
| student_id | name  | date       | score | category |
+------------+-------+------------+-------+----------+
|         31 | Emily | 2008-09-03 |    11 | Q        |
|         31 | Emily | 2008-09-06 |    19 | Q        |
|         31 | Emily | 2008-09-09 |    81 | T        |
|         31 | Emily | 2008-09-16 |    19 | Q        |
|         31 | Emily | 2008-09-23 |     9 | Q        |
|         31 | Emily | 2008-10-01 |    76 | T        |
+------------+-------+------------+-------+----------+
```

有些视图是可更新的，这意味着你可以通过对视图进行操作而在其底层数据表里插入、更新或删除数据行。下面是一个简单的例子：

```
mysql> CREATE TABLE t (i INT);
mysql> INSERT INTO t (i) VALUES(1),(2),(3);
mysql> CREATE VIEW v AS SELECT i FROM t;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
+------+
mysql> INSERT INTO v (i) VALUES(4);
mysql> DELETE FROM v WHERE i < 3;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    3 |
|    4 |
+------+
mysql> UPDATE v SET i = i + 1;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    4 |
|    5 |
+------+
```

要想让一个视图是可更新的，它必须直接映射到一个数据表上，它选取的数据列只能是数据表里数据列的简单引用（不能是表达式），视图里的单行操作必须对应于对其底层数据表里的一个单行操作。比如说，如果某个视图里有一个“汇总”数据列是用一个聚合函数计算出来的，这个视图里的每个数据行都将涉及其底层数据表里的多个数据行。这样的视图是不可更新的，因为你无法告诉 MySQL 应该更新其底层数据表里的哪一个数据行。

2.12 涉及多个数据表的删除和更新操作

有时候，我们可以根据某个数据表里数据行是否在另一个数据表里有匹配来删除它们，这很有用。类似地，用一个数据表里的数据行的内容去更新另一个数据表也很有用。本节讨论如何完成涉及多个数据表的 DELETE 和 UPDATE 操作。联结概念在用来完成这些操作的语句里扮演着极其重要的角色，这要求你对前面 2.8 节里讨论的内容有透彻的理解。

对于只涉及单个数据表的 DELETE 和 UPDATE 操作，被引用的数据列都来自同一个数据表，不需要使用数据表的名字对数据列的名字进行限定。比如说，如果要从数据表 t 里把 id 值大于 100 的数据行全部删掉，可以编写如下所示的语句：

```
DELETE FROM t WHERE id > 100;
```

如果需要根据某给定数据表里数据行与另一个数据表里的数据行之间的关系（而不是根据其自身的属性）来删除它们，你该怎么办？比如说，如果要从数据表 t 里把其 id 值可以在另一个数据表 t2 里找到的数据行删掉，该怎么办？

在编写一个涉及多个数据表的 DELETE 语句时，要把所涉及的数据表在 FROM 子句里全部列出来并把用来匹配各有关数据行（它们来自多个数据表）的检索条件写在 WHERE 子句里。下面这条语句将从数据表 t1 里把其 id 值可以在另一个数据表 t2 里找到的数据行全部删掉：

```
DELETE t1 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

请注意，如果某个数据列的名字出现在了多个数据表里，就有可能导致歧义问题，就需要用数据表的名字对它加以限定。

DELETE 语句有一种语法可以让我们一次删除多个数据表里的数据行。如果你想从两个数据表里把 id 值相匹配的数据行都删掉，必须在 DELETE 关键字的后面写出两个数据表的名字：

```
DELETE t1, t2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

如果你想删除的是不匹配的数据行，又该怎么办？在涉及多个数据表的 DELETE 语句里可以使用允许用在 SELECT 语句里的任何一种联结操作，所以我们可以按照编写一条从多个数据表选取不匹配数据行的 SELECT 语句的思路去思考。这通常都会归结到是选用 LEFT JOIN 还是选用 RIGHT JOIN 上。比如说，如果要从数据表 t1 里把在数据表 t2 里没有匹配的数据行找出来，你应该会写出一条如下所示的 SELECT 语句：

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

同样，从数据表 t1 找出并删除那些数据行的 DELETE 语句也要用到一个 LEFT JOIN 操作：

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

MySQL 还支持另一种涉及多个数据表的 DELETE 语法。这种语法使用一个 FROM 子句来列出将从中删除有关数据行的数据表，使用一个 USING 子句来联结各有关数据表以确定哪些数据行需要被删除。前面那几条涉及多个数据表的 DELETE 语句可以用这种语法改写为如下所示的样子：

```
DELETE FROM t1 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1, t2 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1 USING t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

编写涉及多个数据表的 UPDATE 语句的基本步骤与编写涉及多个数据表的 DELETE 语句的很相似。同样需要列出所涉及的全部数据表，同样需要用数据表的名字对数据列的名字进行必要的限定。我们

来看一个例子。假设 2008 年 9 月 23 日的测试里有一个问题是所有学生都没有答对的，而你后来发现这是因为你的标准答案有错误。于是，你决定给每位学生的考试成绩加上一分。下面这条涉及多个数据表的 UPDATE 语句可以完成这个工作：

```
UPDATE score, grade_event SET score.score = score.score + 1
WHERE score.event_id = grade_event.event_id
AND grade_event.date ='2008-09-23' AND grade_event.category = 'Q';
```

具体到这个问题，你也可以用一个基于子查询的单数据表更新操作来达到同样的目的：

```
UPDATE score SET score = score + 1
WHERE event_id = (SELECT event_id FROM grade_event
WHERE date = '2008-09-23' AND category = 'Q');
```

但其他类型的更新操作能不能用子查询来完成就不一定了。比如说，假设你不仅需要根据另一个数据表的内容来确定应该刷新某给定数据表里的哪些数据行，还需要把另一个数据表的数据列值复制到这个数据表里。下面这条语句将把符合条件的 t1.a 复制到 t2.a，而需要满足的条件是数据行有匹配的 id 数据列值：

```
UPDATE t1, t2 SET t2.a = t1.a WHERE t2.id = t1.id;
```

如果是对 InnoDB 数据表进行多数据表删除和刷新操作，你不必非得使用刚才介绍的语法。更好的办法是在数据表之间建立一个外键关系并给它加上 ON DELETE CASCADE 或 ON UPDATE CASCADE 约束条件。详见 2.14 节。

2.13　事务处理

事务（transaction）是作为一个不可分割的逻辑单元而被执行的一组 SQL 语句，如有必要，它们的执行效果可以被撤销。并非所有的语句每次都能执行成功，有些语句还会对数据产生永久性的影响。事务处理是通过提交（commit）和回滚（rollback）功能实现的。如果某个事务里的所有语句都执行成功了，提交该事务将把那些语句的执行效果永久性地记录到数据库里。如果在事务过程中发生错误，回滚该事务将把发生错误之前已经执行的语句全部取消，数据库将恢复到开始这次事务之前的状态。

提交和回滚机制使我们能够确保尚未全部完成的操作不会影响到数据库，不会让新旧数据混杂在一起让数据库呈不稳定状态。财务转账是一个典型的事务处理例子，即把钱从一个账户转到另一个账户。假设 Bill 给 Bob 开了一张 100 美元的支票，Bill 拿着这张支票去取钱。Bill 的账户应该减少 100 美元，Bob 的账户应该增加 100 美元：

```
UPDATE account SET balance = balance - 100 WHERE name = 'Bill';
UPDATE account SET balance = balance + 100 WHERE name = 'Bob';
```

可是，万一银行的计算机系统在这两条语句正在执行时发生了崩溃，整个操作将不完整。根据先执行的是哪一条语句，Bill 的账户可能少了 100 美元而 Bob 的账户金额没增加，或者 Bob 的账户多了 100 美元而 Bill 的账户金额没减少。这两种情况都不正确。如果没有使用事务机制，你将不得不以手动方式分析你的日志以查明崩溃发生时都有哪些操作正在进行，应该以及如何撤销或继续完成哪些操作，等等。事务机制提供的回滚操作可以让你正确地处理好这些问题，把发生错误之前已经执行完的语句的效果撤销掉。（作为善后工作的一部分，你还需要确定哪些事务需要再次执行，但至少你不必担心那些未能全部完成的事务会损害数据库的完整性了。）

事务的另一种用途是确保某个操作所涉及的数据行在你正在使用它们时不会被其他客户修改。

MySQL 在执行每一条 SQL 语句时都会自动地对该语句所涉及的资源进行锁定以避免各语句之间相互干扰。但这仍不足以保证每一个数据库操作总是能够得到预期的结果。要知道，有些数据库操作需要多条语句才能完成，而在此期间，不同的客户就有可能相互干扰。通过把多条语句定义为一个执行单元，事务机制可以防止在多客户环境里可能发生的并发问题。

事务机制的特性通常被概括为“ACID 原则”。ACID 是 Atomic（原子性）、Consistent（稳定性）、Isolated（孤立性）和 Durable（可靠性）的首字母缩写，它们分别代表事务机制应该具备的一个属性。

- **原子性**。构成一个事务的所有语句应该是一个独立的逻辑单元，要么全部执行成功，要么一个都不成功。你不能只执行它们当中的一部分。
- **稳定性**。数据库在事务开始执行之前和事务执行完毕之后都必须是稳定的。换句话说，事务不应该把你的数据库弄得一团糟。
- **隔离性**。事务不应该相互影响。
- **可靠性**。如果事务执行成功，它的影响将被永久性地记录到数据库里。

事务处理为数据库操作的结果提供了强有力的保证，但这需要以增加 CPU、内存和硬盘空间等方面的开销为代价。MySQL 提供了几种具备事务安全性的存储引擎（如 InnoDB 和 Falcon）和一些不具备事务安全性的存储引擎（如 MyISAM 和 MEMORY）。有些应用程序需要通过事务来实现，另外一些则不然，你可以根据具体情况挑选最适合的。一般来说，与金融有关的操作应该以事务方式完成，这是因为财务数据的完整性要比额外增加的开销成本更重要。从另一方面看，对于一个负责把 Web 页面的访问情况记入数据库表的应用程序来说，在主机崩溃时损失一些数据行应该是可以忍受的，你可以选用一种非事务存储引擎以避免事务处理所要求的额外开销。

2.13.1 利用事务来保证语句的安全执行

要想使用事务，就必须选用一种支持事务处理的存储引擎，如 InnoDB 或 Falcon。MyISAM 和 MEMORY 等其他存储引擎帮不上这个忙。如果你拿不准 MySQL 服务器是否支持任何事务存储引擎，请参见 2.6.1 节中的第 1 小节。

在默认的情况下，MySQL 从自动提交（autocommit）模式运行，这种模式会在每条语句执行完毕后把它作出的修改立刻提交给数据库并使之永久化。事实上，这相当于把每一条语句都隐含地当做一个事务来执行。如果你想明确地执行事务，需要禁用自动提交模式并告诉 MySQL 你想让它在何时提交或回滚有关的修改。

执行事务的常用办法是发出一条 START TRANSACTION（或 BEGIN）语句挂起自动提交模式，然后执行构成本次事务的各条语句，最后用一条 COMMIT 语句结束事务并把它们作出的修改永久性地记入数据库。万一在事务过程中发生错误，用一条 ROLLBACK 语句撤销事务并把数据库恢复到事务开始之前的状态。START TRANSACTION 语句“挂起”自动提交模式的含义是：在事务被提交或回滚之后，该模式将恢复到开始本次事务的 START TRANSACTION 语句被执行之前的状态。（如果自动提交模式原来是激活的，结束事务将让你回到自动提交模式；如果它原来是禁用的，结束当前事务将开始下一个事务。）

下面的例子演示了这个套路。首先，创建一个数据表供演示使用：

```
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE = InnoDB;
```

这个语句将创建一个 InnoDB 数据表，但你完全可以根据个人喜好选用一种不同的事务存储引擎。

接下来，用 START TRANSACTION 语句开始一次事务，往数据表里添加一些数据行，提交本次事务，然后看看数据表变成了什么样子：

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

你可以看到一些数据行已被记录到了数据表里。如果你另外启动一个 mysql 程序的实例并在插入之后、但提交之前选取数据表 t 的内容的话，你将看不到那些数据行。在第一个 mysql 进程发出 COMMIT 语句之前，那些数据行对第二个 mysql 进程来说是不可见的。

如果子事务过程中发生了一个错误，你可以用 ROLLBACK 语句把它删除。仍以数据表 t 为例，你可以通过发出下面这些语句看到这种情况：

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

第二条 INSERT 语句试图把一个其 name 值与一个现有的数据行重复的数据行插入到数据表里。因为 name 数据列有一个 UNIQUE 索引，所以这条语句将执行失败。在发出 ROLLBACK 语句之后，这个数据表只包含在这次失败事务之前被插入的两个数据行。准确地说，在这次事务里，在发生错误之前已经执行的 INSERT 语句被撤销了，它们的影响没有被记录到数据表里。

如果在事务过程中发出一条 START TRANSACTION 语句，它将隐含地提交当前事务，然后开始一个新的事务。

执行事务的另一个办法是利用 SET 语句直接改变自动提交模式的状态：

```
SET autocommit = 0;
SET autocommit = 1;
```

把 autocommit 变量设置为零将禁用自动提交模式，其效果是随后的任何语句都将成为当前事务的一部分，直到你发出一条 COMMIT 或 ROLLBACK 语句来提交或撤销它为止。如果使用这个办法，自动提交模式的状态将一直保持下去直到你把它设置回原来的状态，所以结束一个事务将开始下一个事务。你也可以通过重新激活自动提交模式的办法来提交一个事务。

我们来看看这个办法的工作情况。首先，创建一个和前面那个例子一样的数据表：

```
mysql> DROP TABLE t;
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE = InnoDB;
```

接下来，禁用自动提交模式，插入一些数据行，提交本次事务：

```
mysql> SET autocommit = 0;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

现在，有两个数据行被插入了数据表，但自动提交模式仍处于被禁用状态。如果你继续发出一些语句，它们将成为一个新事务的组成部分，它们的提交或撤销与第一个事务不会有任何关系。下面这些语句可以证明自动提交模式仍处于被禁用状态，并且 ROLLBACK 将撤销尚未被提交的语句：

```
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

要想重新激活自动提交模式，使用下面这条语句即可：

```
mysql> SET autocommit = 1;
```

正如刚才描述的那样，发出一条 COMIT 或 ROLLBACK 语句将结束当前事务，先禁用自动提交模式、再重新激活它也将结束当前事务。在其他坏境下，事务也会结束。SET autocommit、STRAT TRANSACTION、BEGIN、COMMIT 和 ROLLBACK 语句会明确地对事务产生影响，除它们以外，还有一些语句会对事务产生隐式影响，因为它们不能成为事务的一部分。一般来说，用来创建、改变或删除数据库或其中的对象的 DLL（Data Definition Language，数据定义语言）语句以及与锁定有关的语句都不能成为事务的一部分。比如说，如果你在事务过程中发出了下面这些语句之一，服务器将在执行该语句之前先提交当前事务：

```
ALTER TABLE
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOCK TABLES
RENAME TABLE
SET autocommit = 1 (if not already set to 1)
```

```
TRUNCATE TABLE
UNLOCK TABLES (if tables currently are locked)
```

如果你想知道你正在使用的 MySQL 版本都有哪些语句会隐含地提交当前事务，请查阅《MySQL 参考手册》。

事务提交前，客户连接的正常结束或意外中断也将导致事务结束。此时，服务器会自动回滚该客户正提交的所有事务。

如果客户程序在与服务器连接意外断开后再自动重建连接，新建的连接将被重置为激活自动提交模式的默认状态。

有许多实际问题需要依靠事务才能确保它们被正确地解决。比如说，假设你有一个用来记录学生成绩的应用项目，其中有一个 score 数据表，你发现有两名学生的考试分数输入错了，需要交换一下。输入有误的考试成绩如下所示：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    18 |
|          9 |        5 |    13 |
+------------+----------+-------+
```

要纠正这个错误，应把 8 号学生的成绩改成 13，把 9 号学生的成绩改为 18。只需两条简单的语句就可以完成：

```
UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
```

可是，你必须保证这两条语句是作为一个不可分割的逻辑单元而执行成功的。这是一个需要使用事务来解决的典型问题。下面是用 START TRANSACTION 语句开始一个事务来解决这个问题的具体做法：

```
mysql> START TRANSACTION;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
```

下面通过直接设置自动提交模式来完成同样的事情：

```
mysql> SET autocommit = 0;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
mysql> SET autocommit = 1;
```

两种办法都可以保证那两名学生的考试成绩被正确地交换过来：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    13 |
|          9 |        5 |    18 |
+------------+----------+-------+
```

2.13.2 使用事务保存点

MySQL 使你能够对一个事务进行部分回滚。这需要你在事务过程中使用 SAVEPOINT 语句设置一些称为保存点（savepoint）的标记。在后续的事务里，如果你想回滚到某个特定的保存点，在 ROLLBACK 语句里给出改保存点的名字就可以了。下面的语句演示了这个过程：

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
mysql> START TRANSACTION;
mysql> INSERT INTO t VALUES(1);
mysql> SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(2);
mysql> ROLLBACK TO SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(3);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| i    |
+------+
|    1 |
|    3 |
+------+
```

在执行完这些语句之后，数据表里只有第一条和第三条 INSERT 语句插入的数据行，没有第二条 INSERT 语句插入的数据行，它被那条含义是“回滚到 my_savepoint 保存点”的 ROLLBACK 语句给取消了。

2.13.3 事务的隔离性

因为 MySQL 是一个多用户数据库系统，所以不同的客户可能会在同一时间试图访问同一个数据表。诸如 MyISAM 之类的存储引擎使用了数据表级的锁定机制来保证不同的客户不能同时修改同一个数据表，但这种做法在更新量比较大的系统上会导致并发性能的下降。InnoDB 存储引擎采用了另一种策略，它使用了数据行级的锁定机制为客户对数据表的访问提供了更细致的控制：在某个客户修改某个数据行的同时，另一个客户可以读取和修改同一个数据表里的另一个数据行。如果有两个客户想同时修改某个数据行，先锁定该数据行的那个客户将可以先修改它。这比数据表级的锁定机制提供了更好的并发性能。不过，这里还有一个问题：一个客户的事务在何时才能看到另一个客户的事务作出的修改。

InnoDB 存储引擎实现的事务隔离级别机制能够让客户控制他们想看到其他事务作的哪些修改。它提供了多种不同的隔离级别以允许或预防在多个事务同时运行时可能发生的各种各样的问题，如下所示。

- 脏读（dirty read）。指某个事务所作出的修改在它尚未被提交时就可以被其他事务看到。其他事务会认为数据行已经被修改了，但对数据行作出修改的那个事务还有可能会被回滚，这将导致数据库里的数据发生混乱。
- 不可重复读取（nonrepeatable read）。指同一个事务使用同一条 SELECT 语句每次读取到的结果不一样。比如说，如果有一个事务执行了两次同一个 SELECT 语句，但另一个事务在这条 SELECT 语句的两次执行之间修改了一些数据行，就会发生这种问题。
- 幻影数据行（phantom row）。指某个事务突然看到了一个它以前没有见过数据行。比如说，如

果某个事务刚执行完一条 SELECT 语句就有另一个事务插入了一个新数据行，前一个事务再执行同一条 SELECT 语句时就可能多看到一个新的数据行，那就是一个幻影数据行。

为了解决这些问题，InnoDB 存储引擎提供了 4 种隔离级别。这些隔离级别用来确定允许某个事务看到与之同时执行的其他事务所作出的哪些修改，如下所示。

- READ UNCOMMITTED。允许某个事务看到其他事务尚未提交的数据行改动。
- READ COMMITTED。只允许某个事务看到其他事务已经提交的数据行改动。
- REPEATABLE READ。如果某个事务两次执行同一个 SELECT 语句，其结果是可重复的。换句话说，即使有其他事务在同时插入或修改数据行，这个事务所看到的结果也是一样的。
- SERIALIZABLE。这个隔离级别与 REPEATABLE READ 很相似，但对事务的隔离更彻底：某个事务正在查看的数据行不允许其他事务修改，直到该事务完成为止。换句话说，如果某个事务正在读取某些数据行，在它完成之前，其他事务将无法对那些数据行修改。

表 2-4 列出了这 4 种隔离级别以及它们是否允许脏读、不可重复读取或幻影数据行等问题。这个表格只适用于 InnoDB 存储引擎——REPEATABLE READ 隔离级别不能容忍幻影数据行。有些数据库系统的 REPEATABLE READ 隔离级别允许出现幻影数据行。

表2-4　隔离级别允许的问题

隔离级别	脏　读	不可重复读取	幻影数据行
READ UNCOMMITTED	是	是	是
READ COMMITTED	否	是	是
REPEATABLE READ	否	否	否
SERIALIZABLE	否	否	否

InnoDB 存储引擎默认使用的隔离级别是 REPEATABLE READ。这可以通过在启动服务器时使用 --transaction-isolation 选项或在服务器运行时使用 SET TRANSACTION 语句来改变。该语句有 3 种形式：

```
SET GLOBAL TRANSACTION ISOLATION LEVEL level;
SET SESSION TRANSACTION ISOLATION LEVEL level;
SET TRANSACTION ISOLATION LEVEL level;
```

SUPER 权限的客户可以使用 SET TRANSACTION 语句改变全局隔离级别的设置，该设置将作用于此后连接到服务器的任何客户。此外，任何客户都可以修改它自己的事务隔离级别，用 SET SESSION TRANSACTION 语句做出的修改将作用于在与服务器的本次会话里后续的所有事务，用 SET TRANSACTION 语句做出的修改只作用于下一个事务。客户在修改它自己的隔离级别时不需要任何特殊的权限。

本节里的绝大多数信息也适用于 Falcon 存储引擎。Falcon 和 InnoDB 存储引擎在这方面的主要区别是：Falcon 不支持 READ UNCOMMITTED 隔离级别，它目前也不支持 SERIALIZABLE 隔离级别（但 Falcon 开发团队正在为此而努力着）。

2.13.4　事务问题的非事务解决方案

在一个不支持事务的环境里，有些事务问题可以想办法解决，有些问题则毫无办法。下面将讨论哪些问题可以、哪些问题不可以在不使用事务的情况下得到解决。你可以利用这些信息去判断是否可

以把这里介绍的技巧用在某个应用程序里，以避免因为使用具备事务安全性的数据表而增加开销。

首先看看当有多个客户试图使用一些需要多条语句才能完成的操作去修改同一个数据库时，都会导致哪些并发问题。假设你开了一个服装店，你的销售管理软件会在售货员录入一份成交单据时自动更新相应的库存记录。下面是在同一时间卖出多件服装时可能发生的问题。为了便于描述，不妨假设你的衬衫库存记录的初始值是 47 件。

(1) 售货员 A 卖出了 3 件衬衫并把单据录入了电脑。销售管理软件开始更新数据库，它首先要选取当前的衬衫库存数量：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

(2) 与此同时，售货员 B 也卖出了两件衬衫并把单据录入了电脑。第二台电脑里的软件也开始更新数据库：

```
SELECT quantity FROM inventory WHERE item ='shirt';
```

(3) 第一台电脑计算出新的库存数量是 47−3=44 件，并对仓库里的衬衫数量作出了相应的修改：

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
```

(4) 第二台电脑计算出新的库存数量是 47−3=44 件，并对仓库里的衬衫数量作出了相应的修改：

```
UPDATE inventory SET quantity = 45 WHERE item = 'shirt';
```

在这一系列事情结束之后，你的售货员总共卖出了 5 件衬衫，这是个好消息。可是，衬衫的库存数量是 45 件，这就不对了，它应该是 42 件。导致这一问题的根源在于这里有一个多语句操作：一条语句用来查询库存数量，另一条语句用来更新这个值。发生在第二条语句里的动作依赖于第一条语句检索出来的值。如果不同的多语句操作在发生时间上出现重叠，它们就有可能彼此交叉和干扰。要想解决这个问题，就必须设法保证每个多语句操作在执行时都不会与其他的操作发生干扰。

明确地锁定数据表。把多条语句用 LOCK TABLES 和 UNLOCK TABLES 语句括起来就可以把它们当做一个单元来执行：锁定需要使用的所有数据表，发出你的语句，解除锁定。这可以防止其他人在你锁定有关数据表期间修改它们。利用这种锁定机制完成前面的衬衫库存数量更新操作的过程如下所述。

(1) 售货员 A 卖出了 3 件衬衫并把单据录入了电脑。你的销售管理软件开始更新数据库，它先锁定数据表并选取当前的衬衫库存数量（47）：

```
LOCK TABLES inventory WRITE;
SELECT quantity FROM inventory WHERE item = 'shirt';
```

这里需要使用 WRITE 锁，因为整个操作的最终目的是修改 inventory 数据表，需要对它进行写操作。

(2) 与此同时，售货员 B 也卖出了两件衬衫并把单据录入了电脑。第二台电脑里的软件也开始更新数据库，它也是先从锁定数据表开始：

```
LOCK TABLES inventory WRITE;
```

具体到这个例子，这条语句将被阻塞，因为售货员 A 已经锁定那个数据表了。

(3) 第一台电脑计算出新的库存数量是 47−3=44 件，它更新了衬衫件数并解除了锁定：

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
UNLOCK TABLES;
```

(4) 在第一台电脑解除了对数据表的锁定之后，第二台电脑的锁定请求成功了，它继续执行并检索出当前衬衫库存数量是 44 件：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

(5) 第二台电脑计算出新的库存数量是 47−3=44 件，它更新了衬衫件数并解除了锁定：

```
UPDATE inventory SET quantity = 42 WHERE item = 'shirt';
UNLOCK TABLES;
```

就这样，来自两个操作的各语句不再相互混杂，对库存数量的修改也就不会再出现错误了。

如果你正在使用多个数据表，在执行多语句操作之前必须把它们全部锁定。如果你只从某个特定的数据表读取数据，你只需要给它加上一个“读操作”锁就行了，用不着给它加上“写操作”锁。（“读操作”锁允许其他客户在你使用被锁定数据表时读取它，但不允许对之进行写操作。）比如说，假设你有一组查询是用来修改 inventory 数据表的，你还需要从 customer 数据表读取一些数据。此时，你需要给 inventory 数据表加上一把“写操作”锁，给 customer 数据表加上一把“读操作”锁：

```
LOCK TABLES inventory WRITE, customer READ;
... use the tables here ...
UNLOCK TABLES;
```

使用相对更新操作，不使用绝对更新操作。在先明确锁定数据表再更新库存的办法里，整个操作需要两条语句来完成，一条语句用来查询当前库存水平，另一条语句用来计算本次销售后的衬衫库存数量并对数据表进行相应的更新。让多个客户同时进行的操作互不干扰的另一个办法，是把整个操作压缩成只用一条语句来完成。这将消除多语句操作中各条语句之间的彼此依赖。并非所有的操作都可以只用一条语句完成，但就刚才那个更新衬衫库存数量的例子而言，这个策略是可行的。只要把计算衬衫库存数量的语句修改为使用相对于它的当前值就可以只用一个步骤完成库存更新操作了，如下所示。

(1) 售货员 A 卖出了 3 件衬衫，销售管理软件把衬衫的当前库存数量减去 3：

```
UPDATE inventory SET quantity = quantity - 3 WHERE item = 'shirt';
```

(2) 售货员 B 卖出了两件衬衫，销售管理软件把衬衫的当前库存数量减去 2：

```
UPDATE inventory SET quantity = quantity - 2 WHERE item = 'shirt';
```

这个办法的优点是数据库修改时不再需要使用多条语句。这消除了并发问题，所以不再需要明确地锁定数据表。如果你打算执行的某个操作与此相似，也许根本用不着借助事务机制就可以完成它。

刚才介绍的这两种非事务解决方案可以解决许多实际问题，但它们也有一定的局限性，如下所示。

- 并非所有的操作都可以被编写成一条相对更新语句。有些问题只有使用多条语句才能解决，你必须考虑和解决随之而来的并发问题。
- 在多语句操作期间锁定数据表可以避免客户之间相互干扰，但万一在操作过程中出现错误又会发生什么样的事情？此时，你肯定希望能够撤销此前已执行完毕的语句对数据库的影响，不让改了一半儿的数据留在数据库里造成数据库的不稳定。令人遗憾的是，虽然数据表锁定机制可以帮助你解决并发问题，但如果所涉及的数据表不支持事务处理，它们也不能为你的善后恢复工作提供多少帮助。
- 锁定机制需要你锁定和释放数据表。如果你改用了其他的数据表，千万不要忘记对 LOCK TABLES 语句作相应的修改。

如果这些问题对你的应用程序很重要，你应该选用具备事务安全性的数据表，事务机制可以帮你妥善处理所有这些问题。事务机制可以把一组语句当做一个不可分割的单元来执行，并防止客户之间彼此干扰，从而有效地管理好并发问题。它提供的回滚功能还可以在发生意外时避免尚未全部完成的操作损坏数据库。它还可以自行判断并获得必要的操作锁，让你不再为这些细节操心费力。

事务数据表和非事务数据表可以混用吗

在某次事务中混合使用事务数据表和非事务数据表是允许的，但最终的结果不一定符合你的期望。对非事务数据表进行操作的语句总是立刻生效，即便是自动提交模式处于禁用状态也是如此。事实上，非事务数据表永远待在自动提交模式下，每条语句都会在执行完毕后立刻提交。因此，如果你在一个事务里修改了一个非事务数据表，那么这个修改将无法撤销。

2.14 外键和引用完整性

利用外键（foreign key）关系可以在某个数据表里声明与另一个数据表里的某个索引相关联的索引。还可以把你想施加在数据表上的约束条件放到外键关系里，让系统根据这个关系里的规则来维护数据的引用完整性。比如说，sampdb 数据库里的 score 数据表包含一个 student_id 数据列，我们要用它把 score 数据表里的考试成绩与 student 数据表里的学生联系在一起。当我们在第 1 章里创建这些数据表时，我们在它们之间建立了一些明确的关系，其中之一是把 score.student_id 数据列定义为 student.student_id 数据列的一个外键。这可以确保只有那些在 studnt 数据表里存在 student_id 值的数据行才能被插入到 score 数据表里。换句话说，这个外键可以确保不会出现为一名并不存在的学生输入了成绩的错误。

外键不仅在数据行的插入操作中很有用，在删除和更新操作中也很有用。比如说，我们可以建立这样一个约束条件：在把某个学生从 student 数据表里删除时，score 数据表里与这个学生有关的所有数据行也将自动被删除。这被称为级联删除（cascaded delete），因为删除操作的效果就像瀑布（cascade）那样从一个数据表“流淌”到另外一个数据表。级联更新也是可能的。比如说，如果利用瀑布式更新在 student 数据表里改变了某个学生的 student_id，score 数据表里与这个学生相对应的所有数据行的这个值也应该发生相应的改变。

外键可以帮我们维护数据的一致性，它们用起来也很方便。如果不使用外键，就必须由你来负责保证数据表之间的依赖关系和维护它们的一致性，而这意味着你的应用程序必须增加一些必要的代码。在某些情况下，这只需要你额外发出几条 DELETE 语句以确保当你删除某个数据表里的数据行时，其他数据表里与之相对应的数据行也将随之一起被删除。但额外工作毕竟是额外工作，而且既然数据库引擎能够替你进行数据一致性检查，为什么不让它干呢？要是你的数据表有非常复杂的关系，由你在你的应用程序里通过代码去检查这些依赖关系就会变得很麻烦，而数据库系统提供的自动检查能力往往要比你本人考虑得更周全和更细致，也更简明实用。

在 MySQL 里，InnoDB 存储引擎提供了外键支持。本节将讨论如何设置 InnoDB 数据表以定义外键以及外键将如何影响你使用数据表的方式和方法。首先，有必要定义几个术语：

- 父表，包含原始键值的数据表；
- 子表，引用父表中的键值的相关数据表。

父表中的键值用来关联两个数据表。具体地说，子表中的某个索引引用父表中的某个索引。子表

的索引值必须匹配父表中的索引值或是被设置为 NULL 以表明在父表里不存在与之对应的数据行。子表里的索引就是所谓的“外键”，因为它们存在于父表的外部，但包含指向父表的值。外键关系可以被设置成不允许使用 NULL 值，此时所有的外键值都必须匹配父表里的某个值。

InnoDB 存储引擎通过这些规则来保证在外键关系里不会有不匹配的东西，这被称为*引用完整性*（referential integrity）。

2.14.1　外键的创建和使用

在子表里定义一个外键的语法如下所示：

```
[CONSTRAINT constraint_name]
FOREIGN KEY [fk_name] (index_columns)
  REFERENCES tbl_name (index_columns)
  [ON DELETE action]
  [ON UPDATE action]
  [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
```

这个语法本身很完备，但 InnoDB 存储引擎目前还没有实现所有的子句：它目前还不支持 MATCH 子句，即使你给出了一条 MATCH 子句，它也会被忽略。有几种 action 值目前只能被识别出来，但不会有任何效果。（除 InnoDB 以外的其他存储引擎在遇到 FOREIGN KEY 定义时不会报告错误，但会把它整个忽略掉。）

InnoDB 存储引擎目前能够识别和支持以下外键定义语法成分。

- CONSTRAINT 子句。如果给出，这个子句用来给外键约束关系起一个名字。如果你省略了它，InnoDB 存储引擎将创建一个名字。
- FOREIGN KEY 子句。列出子表里的被索引数据列，它们必须匹配父表里的索引值。fk_name 是外键的 ID。如果你给出了一个 fk_name，在 InnoDB 存储引擎能够为外键自动创建一个索引的情况下它将成为那个索引的名字；否则，它将被忽略。
- REFERENCES 子句。列出父表和父表中的索引数据列的名字，子表里的外键将引用这个子句所列出的父表数据列。在 REFERENCES 子句的 index_columns 部分列出的数据列的个数必须与在 FOREIGN KEY 子句的 index_columns 部分列出的数据列的个数相同。
- ON DELETE 子句。用来设定在父表里的数据行被删除时子表应该发生什么事。如果没有 ON DELETE 子句，其默认行为是拒绝从父表里删除仍有子表数据行在引用它们的数据行。如果你想明确地指定一种 action 值，请使用以下子句之一。
  - ON DELETE NO ACTION 和 ON DELETE RESTRICT 子句。它们的含义与省略 ON DELETE 子句一样。（有些数据库系统提供了延迟检查功能，而 NO ACTION 是一种延迟检查。在 MySQL 里，外键约束条件是被立刻检查的，所以 NO ACTION 和 RESTRICT 的含义完全一样。）
  - ON DELETE CASCADE 子句。在删除父表数据行时，子表里与之相关联的数据行也将被删除。本质上，删除效果将从父表蔓延到子表。这样一来，你只需删除父表里的数据行并让 InnoDB 存储引擎负责从子表删除相关数据行，就可以完成一个涉及多个数据表的删除操作了。
  - ON DELETE SET NULL 子句。在删除父表数据行时，子表里与之相关联的索引列将被设置为 NULL。如果你打算使用这个选项，就必须把在外键定义里列出的所有被编制索引的子表数据列定义为允许 NULL 值。（使用这个动作的一个隐含限制是你不能把外键定义为 PRIMARY KEY，因为主键不允许 NULL 值。）

- ON DELETE SET DEFAULT 子句。这个子句可以被识别出来，但目前尚未实现；InnoDB 存储引擎在遇到这个子句时将报告一个错误。

❑ ON UPDATE 子句。用来设定当父表里的数据行更新时子表应该发生什么事。如果没有 ON UPDATE 子句，其默认行为是拒绝插入或更新其外键值在父表索引里没有任何匹配的子表数据行，并阻止仍有子表数据行在引用着它们的父表索引值被更新。可供选用的 action 值及其效果与 ON DELETE 子句的相同。

如果你想建立一个外键关系，请遵守以下指示。

❑ 子表必须有这样一个索引。在定义该索引时，必须首先列出外键数据列。父表必须有这样一个索引：在定义该索引时，必须首先列出 REFERENCES 子句里的数据列。（换句话说，外键里的数据列在外键关系所涉及的两个数据表里都必须有索引。）在定义外键关系之前，你必须明确地创建出必要的父表索引。InnoDB 存储引擎将自动地在子表里为外键数据列（引用数据列）创建一个索引——如果在你用来创建子表的 CREATE TABLE 语句里没有包括一个这样的索引的话。不过，由 InnoDB 存储引擎自动创建的这种索引将是一个非唯一的索引，并且只包含外键数据列。如果你想让这个子表索引是一个 PRIMARY KEY 或 UNIQUE 索引，或者如果你想让它在外键数据列之外还包括其他的数据列，就只能由你本人明确地定义它了。

❑ 父表和子表索引里的对应数据列必须是兼容的数据类型。比如说，你不能让一个 INT 数据列去匹配一个 CHAR 数据列。对应的字符数据列必须是同样的长度。对应的整数数据列必须是同样的尺寸，并且必须要么都带符号，要么都被定义成 UNSIGNED。

❑ 你不能对外键关系里的字符串数据列的前缀编制索引。换句话说，对于字符串数据列，你必须对整个数据列编制索引，不能只对它的前几个字符编制索引。

在第 1 章里，我们为考试成绩管理项目创建了几个有着简单外键关系的数据表。我们现在来看一个比较复杂的例子。首先创建两个分别名为 parent 和 child 的数据表，其中 child 数据表包含一个外键，该外键引用 parent 数据表里的 par_id 数据列：

```
CREATE TABLE parent
(
  par_id    INT NOT NULL,
  PRIMARY KEY (par_id)
) ENGINE = INNODB;

CREATE TABLE child
(
  par_id    INT NOT NULL,
  child_id  INT NOT NULL,
  PRIMARY KEY (par_id, child_id),
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE = INNODB;
```

这个例子在定义外键时使用了 ON DELETE CASCADE 子句，它指定当 parent 数据表里的某个数据行被删除时，MySQL 将自动地从 child 数据表里把有匹配 par_id 值的数据行也删掉。ON UPDATE CASCADE 子句表明：如果 parent 数据表里的某个数据行的 par_id 值被改变了，MySQL 将自动地把 child 数据表里的所有匹配的 par_id 值也改成新值。

现在，在 parent 数据表里插入一些数据行，再在 child 数据表插入一些有着相关键值的数据行：

```
mysql> INSERT INTO parent (par_id) VALUES(1),(2),(3);
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
```

这些语句将导致如下所示的数据表内容，而 child 数据表里的每一个 par_id 值都分别匹配 parent 数据表里的一个 par_id 值：

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      1 |
|      2 |
|      3 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      1 |        1 |
|      1 |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

为了证明 InnoDB 存储引擎在插入新数据行时会遵守外键关系的约束，我们现在故意往 child 数据表插入一个“错误的”数据行，它的 par_id 值在 parent 数据表里没有匹配：

```
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`child`, CONSTRAINT `child_ibfk_1` FOREIGN
KEY (`par_id`) REFERENCES `parent` (`par_id`) ON DELETE CASCADE
ON UPDATE CASCADE)
```

想看看级联删除的效果？从 parent 数据表删除一个数据行试试：

```
mysql> DELETE FROM parent WHERE par_id = 1;
```

MySQL 将从父表删除这个数据行：

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      2 |
|      3 |
+--------+
```

同时，MySQL 还将把这个 DELETE 语句的效果蔓延到 child 数据表：

```
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
```

```
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

想看看级联更新的效果？对 parent 数据表里的某个数据行更新一下试试：

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id =2;
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      3 |
|    100 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      3 |        1 |
|    100 |        1 |
|    100 |        2 |
|    100 |        3 |
+--------+----------+
```

上面几个例子演示了 parent 数据表里的数据行删除和更新操作将导致 child 数据表里的相关数据行被级联删除或更新的情况。ON DELETE 和 ON UPDATE 子句还支持其他动作。比如说，你可以把 child 数据表里的数据行保留下来不删除，但它们的外键数据列将被设置为 NULL。要想做到这一点，你必须对 child 数据表的定义作一些必要的修改，如下所示。

- 使用 ON DELETE SET NULL 来代替 ON DELETE CASCADE。这将使 InnoDB 存储引擎把外键数据列（par_id）设置为 NULL 而不是删除那些数据列。
- 使用 ON UPDATE SET NULL 来代替 ON UPDATE CASCADE。这将使 InnoDB 存储引擎在 parent 数据表里的数据行被更新时把 child 数据表里的对应数据行的外键数据列（par_id）设置为 NULL。
- child 数据表里的 par_id 数据列最初被定义成 NOT NULL。这不能与 ON DELETE SET NULL 或 ON UPDATE SET NULL 配合使用，所以必须把这个数据列的定义改成允许有 NULL 值。
- child 数据表里的 par_id 数据列最初也被定义成 PRIMARY KEY 的一部分。因为 PRIMARY KEY 不允许包含 NULL 值，所以在把 child 数据表里的 par_id 数据列改成允许为 NULL 值的同时，还需要把那个 PRIMARY KEY 改成一个 UNIQUE 索引。UNIQUE 索引要求索引值必须是独一无二的——但 NULL 值除外，NULL 值可以在索引里出现多次。

想看看这些修改的效果吗？按最初的定义重新创建 parent 数据表并把同样的数据行加载到其中，然后用如下所示的新定义创建一个新的 child 数据表：

```
CREATE TABLE child
(
  par_id    INT NULL,
  child_id  INT NOT NULL,
  UNIQUE (par_id, child_id),
```

```
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
    ON DELETE SET NULL
    ON UPDATE SET NULL
) ENGINE = INNODB;
```

现在，在往 child 数据表插入新数据行时，child 数据表的行为和原来的定义基本一样：只有其 par_id 值在 parent 数据表里出现过的数据行才能被插入 child 数据表，否则将拒绝插入：

```
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails ('sampdb'.'child', CONSTRAINT 'child_ibfk_1' FOREIGN
KEY ('par_id') REFERENCES 'parent' ('par_id') ON DELETE SET NULL
ON UPDATE SET NULL)
```

请注意，往 child 数据表插入新数据行时与原来有一点区别：因为 par_id 数据列现在被定义成允许为 NULL 值，所以你现在可以把包含 NULL 值的新数据行插入 child 数据表而不会引起错误。另一个区别体现在从 parent 数据表删除数据行的时候，从 parent 数据表删除一个数据行，然后查看一下 child 数据表的内容就知道怎么回事了：

```
mysql> DELETE FROM parent WHERE par_id = 1;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

看到了吗？child 数据表里 par_id 数据列的值是 1 的数据行没有被删掉，但它们的 par_id 数据列的值都被设置成了 NULL，这正是 ON DELETE SET NULL 子句的效果。

对 parent 数据表里的数据行进行刷新将有类似的效果：

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id = 2;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        1 |
|   NULL |        2 |
|   NULL |        2 |
|   NULL |        3 |
|      3 |        1 |
+--------+----------+
```

如果你想查看某个 InnoDB 数据表都有哪些外键关系，可以使用 SHOW CREATE TABLE 或 SHOW TABLE STATUS 语句。

如果你在试图创建一个带有外键关系的数据表时遇到问题，可以使用 SHOW ENGINE INNODB STATUS 语句查看完整的出错消息。

2.14.2 如果不能使用外键该怎么办

如果 MySQL 服务器没有 InnoDB 支持，或者如果你正在使用另一种存储引擎（因为 InnoDB 不能提供你需要的功能，如 FULLTEXT 索引或空间数据类型），你就享受不到外键带来的好处。在这类情况下，怎样才能保证数据表之间的关系是正确和稳定的呢？

外键施加的约束条件并不难通过编程逻辑来实现。有时候，只要换个办法来录入数据就可以解决问题。以考试成绩管理项目中的 student 和 score 数据表为例，它们现在是通过一个基于每个表中 student_id 值的外键关系关联在一起。如果我们当初把它们创建为 MyISAM 数据表而不是 InnoDB 数据表，因为 MyISAM 数据表不支持外键，这两个数据表之间的关系将是隐式的而不是显式的。在某次考试或小测验之后，你将需要把一组新的考试成绩添加到数据库里，而你必须保证不会把其 student_id 值没列在 student 数据表里的 score 数据行添加进来。

从某种意义上讲，这只不过是采用正确办法录入数据。在需要输入一组新的考试成绩时，为了避免给一名并不存在的学生输入考试成绩，应该考虑编写一个应用小程序来帮助你：这个小程序依次列出 student 数据表里的每位学生，读取每名学生的考试成绩，使用该名学生的 ID 为 score 数据表生成一个新数据行。这个办法可以确保你永远也不会为一名并不存在的学生输入一个新数据行。不过，如果是由你本人手动发出一系列 INSERT 语句的话，就仍有可能出现“坏”数据行，而 InnoDB 数据表和外键可以保证不会发生这种错误。

从 student 数据表删除一个数据行该怎么办？假设你想删除编号是 13 的学生，这意味着应该从 score 数据表里把与这名学生有关的所有数据行都删掉。如果这两个数据表之间有一个指定级联删除的外键关系，你只要用如下所示的语句删除 student 数据表里的数据行就没事了，MySQL 将替你完成从 scorer 数据表删除相关数据行的工作：

```
DELETE FROM student WHERE student_id = 13;
```

在没有外键支持的情况下，要想获得与 ON DELETE CASCADE 同样的效果，就必须由你本人明确地在所有相关的数据表里把所有相关的数据行都删除干净才行：

```
DELETE FROM student WHERE student_id = 13;
DELETE FROM score WHERE student_id = 13;
```

另一个办法是使用一个涉及多个数据表的删除语句，这可以让你只用一条语句就获得与 ON DELETE CASCADE 同样的效果。但这里需要注意一个不容易察觉的陷阱。比如说，下面这条语句乍看起来毫无问题，其实却考虑得不够周全：

```
DELETE student, score FROM student INNER JOIN score
ON student.student_id = score.student_id WHERE student.student_id = 13;
```

这条语句的问题是：万一被删除的学生在 score 数据表里没有任何成绩记录，它将执行失败。此时，因为 WHERE 子句找不到任何匹配，所以 student 数据表不会有任何数据行被删除。具体到这个例子，LEFT JOIN 更适用，因为它可以把 student 数据表里应该被删除的的数据行全都找出来，包括那些在 score 数据表里没有匹配数据行：

```
DELETE student, score FROM student LEFT JOIN score USING (student_id)
WHERE student.student_id = 13;
```

2.15　使用 FULLTEXT 索引

MySQL 具备全文搜索的能力。全文搜索引擎可以在不使用模板匹配操作的情况下查找单词或短语。全文搜索分为 3 种模式，如下所示。

- 自然语言模式。把搜索字符串解释为一系列单词并查找包含这些单词的数据行。
- 布尔模式。把搜索字符串解释为一系列单词，但允许使用一些操作符字符来“修饰”这些单词以表明特定的要求，如某给定单词必须出现（或不出现）在匹配数据行里，某个数据行必须包含一个精确的短语，等等。
- 查询扩展模式。这种搜索分两阶段进行。第一阶段是自然语言搜索，第二阶段使用原来的搜索字符串加上在第一次搜索中找到的相关度最高的匹配数据行再进行一次搜索。这扩大了搜索范围，它可以把与原来的搜索字符串相关、但用原来的搜索字符串匹配不到的数据行也找出来。

要想对某个数据表进行全文搜索，必须事先为它创建一个 FULLTEXT 索引，这种索引具有以下特点。

- 全文搜索的基础是 FULLTEXT 索引，这种索引只能在 MyISAM 数据表里创建。FULLTEXT 索引只能由 CHAR、VARCHAR 和 TEXT 这几种类型的数据列构成。
- 全文搜索将忽略“常见的”单词，而“常见”在这里的含义是“至少在一半的数据行里出现过”。千万不要忘记这个特点，尤其是在你准备对数据表进行全文搜索测试时。你至少要在测试数据表里插入 3 个数据行。如果那个数据表只有一个或两个数据行，它里面的每个单词将至少有 50%的出现几率，所以对它进行全文搜索将不会有任何结果。
- 全文搜索还将忽略一些常用单词，如“the”、“after”和“other”等，这些单词被称为“休止单词”，MySQL 在进行全文搜索时总是会忽略它们。
- 太短的单词也将被忽略。在默认的情况下，“太短”指少于 4 个字符。但你可以通过重新配置服务器的办法把这个最小长度设置为其他值。
- 全文搜索对“单词”的定义是：由字母、数字、撇号（如“it’s”中的“’”）和下划线字符构成的字符序列。这意味着字符串“full-blood”将被解释为包含“full”和“blood”两个单词。全文搜索匹配整个单词，而不是单词的一部分。只要在一个数据行里找到了搜索字符串里的任何单词，FULLTEXT 引擎就会认为这个数据行与搜索字符串是匹配的。在此基础上，布尔式全文搜索还允许你加上一些额外的要求，比如说，所有的单词都必须出现（不论顺序）才认为是匹配，（在搜索一条短语时）单词顺序必须与在搜索字符串里列出的一致，等等。布尔搜索还可以用来匹配**不**包含特定单词的数据行，或者通过添加一个通配符来匹配以一个给定前缀开头的所有单词。
- FULLTEXT 索引可以为一个或多个数据列而创建。如果它涉及多个数据列，基于该索引的搜索将在所有数据列上同时进行。反过来说，在进行全文搜索时，你给出的数据列清单必须和某个 FULLTEXT 索引所匹配的那些数据列精确匹配。比如说，如果你需要分别搜索 col1、col2 以及“col1 和 col2”，你将需要创建 3 个索引：col1 和 col2 各有一个，“col1 和 col2”有一个。

接下来的例子演示了全文搜索的使用方法。我们将先创建几个 FULLTEXT 索引，然后用 MATCH 操作符对它们进行一些查询。用来创建数据表并把一些样板数据加载到其中的脚本可以在 sampdb 数据

库的 fulltext 子目录里找到。

FULLTEXT 索引的具体创建过程与其他索引大同小异。你可以在创建一个新数据表的同时在 CREATE TABLE 语句里定义它们，也可以在数据表被创建出来以后再用 ALTER TABLE 或 CREATE INDEX 语句添加它们。因为 FULLTEXT 索引要求你必须使用 MyISAM 数据表，所以如果你正在创建一个需要使用全文搜索的 MyISAM 数据表，不妨利用一下 MyISAM 存储引擎的这个特点来加快点儿速度。在加载数据时，先填充数据表、再添加索引的办法要比先创建索引再加载数据的办法快得多。假设你有一个名为 apothegm.txt 的数据文件，其内容是一些名人名言：

```
Aeschylus              Time as he grows old teaches many lessons
Alexander Graham Bell  Mr. Watson, come here. I want you!
Benjamin Franklin      It is hard for an empty bag to stand upright
Benjamin Franklin      Little strokes fell great oaks
Benjamin Franklin      Remember that time is money
Miguel de Cervantes    Bell, book, and candle
Proverbs 15:1          A soft answer turneth away wrath
Theodore Roosevelt     Speak softly and carry a big stick
William Shakespeare    But, soft! what light through yonder window breaks?
Robert Burton          I light my candle from their torches.
```

如果按“名人”、“名言”和“名人加名言”进行搜索，你需要创建 3 个 FULLTEXT 索引：两个数据列各有一个，它们加起来有一个。下面这些语句将创建、填充一个名为 apothegm 的数据表，并为它编制索引：

```
CREATE TABLE apothegm (attribution VARCHAR(40), phrase TEXT) ENGINE = MyISAM;
LOAD DATA LOCAL INFILE 'apothegm.txt' INTO TABLE apothegm;
ALTER TABLE apothegm
  ADD FULLTEXT (phrase),
  ADD FULLTEXT (attribution),
  ADD FULLTEXT (phrase, attribution);
```

2.15.1 全文搜索：自然语言模式

把数据表创建出来之后，对它进行自然语言模式的全文搜索：用 MATCH 操作符列出将被搜索的数据列、用 AGAINST()给出搜索字符串。如下所示：

```
mysql> SELECT * FROM apothegm WHERE MATCH(attribution) AGAINST('roosevelt');
+--------------------+-----------------------------------+
| attribution        | phrase                            |
+--------------------+-----------------------------------+
| Theodore Roosevelt | Speak softly and carry a big stick |
+--------------------+-----------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+-------------------+-------------------------------------------+
| attribution       | phrase                                    |
+-------------------+-------------------------------------------+
| Benjamin Franklin | Remember that time is money               |
| Aeschylus         | Time as he grows old teaches many lessons |
+-------------------+-------------------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
```

```
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
```

在最后一个例子里，请注意查询是如何在不同的数据列里找出包含搜索单词的数据行的，它展示了利用 FULLTEXT 索引同时搜索多个数据列的能力。还请注意，我们在查询命令里列出数据列的顺序是 attribution, phrase，而在创建索引时用的是（phrase, attribution）；这是为了告诉你这个顺序不重要。重要的是必须有一个 FULLTEXT 索引精确地包含你在查询命令里列出的数据列。

如果只想看看某个搜索可以匹配到多少数据行，请使用 COUNT(*)：

```
mysql> SELECT COUNT(*) FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

当你在 WHERE 子句里使用 MATCH 表达式时，自然语言模式的全文搜索的输出数据行按照相关程度递减的顺序排序。相关度以非负浮点数来表示，零代表“毫不相关”。要想查看这些值，在输出数据列清单里加上一个 MATCH 表达式即可：

```
mysql> SELECT phrase, MATCH(phrase) AGAINST('time') AS relevance
    -> FROM apothegm;
+-------------------------------------------------+------------------+
| phrase                                          | relevance        |
+-------------------------------------------------+------------------+
| Time as he grows old teaches many lessons       | 1.3253291845322 |
| Mr. Watson, come here. I want you!              |                0 |
| It is hard for an empty bag to stand upright    |                0 |
| Little strokes fell great oaks                  |                0 |
| Remember that time is money                     | 1.3400621414185 |
| Bell, book, and candle                          |                0 |
| A soft answer turneth away wrath                |                0 |
| Speak softly and carry a big stick              |                0 |
| But, soft! what light through yonder window breaks? |                0 |
| I light my candle from their torches.           |                0 |
+-------------------------------------------------+------------------+
```

自然语言模式的搜索能够找到包含任何一个搜索单词的数据行，所以下面这个查询将把包含单词“hard”或“soft”的数据行都找出来：

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('hard soft');
+---------------------+-------------------------------------------------+
| attribution         | phrase                                          |
+---------------------+-------------------------------------------------+
| Benjamin Franklin   | It is hard for an empty bag to stand upright    |
| Proverbs 15:1       | A soft answer turneth away wrath                |
| William Shakespeare | But, soft! what light through yonder window breaks? |
+---------------------+-------------------------------------------------+
```

自然语言模式是默认的全文搜索模式，在 MySLQ 5.1 和更高版本里，可以通过在搜索字符串的后

面加上 IN NATURAL LANGUAGE MODE 的办法明确地指定这个模式。下面这条语句和前一个例子做的事情完全一样。

```
SELECT * FROM apothegm WHERE MATCH(phrase)
AGAINST('hard soft' IN NATURAL LANGUAGE MODE);
```


2.15.2 全文搜索：布尔模式

全文搜索的布尔模式可以让我们控制多单词搜索操作中的许多细节。要想进行这种模式的搜索，需要在 AGAINST()函数里在搜索字符串的后面加上 IN BOOLEAN MODE 短语。布尔模式的全文搜索有以下特点。

- “50%规则”不再起作用，即使是在超过一半的数据行里出现过的单词也可以被这种搜索匹配出来。
- 查询结果不再按照相关程度排序。
- 在搜索一个短语时，你可以要求所有单词必须按照某种特定的顺序出现。如果是搜索一个短语，需要把构成该短语的所有单词用双引号括起来；如果数据行包含的单词按照给定的顺序排列，才被认为是一个匹配。

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('"bell book and candle"' IN BOOLEAN MODE);
+---------------------+-------------------------+
| attribution         | phrase                  |
+---------------------+-------------------------+
| Miguel de Cervantes | Bell, book, and candle |
+---------------------+-------------------------+
```

- 布尔模式的全文搜索还可以在没被包括在FULLTEXT索引里的数据列上进行，但这要比搜索有FULLTEXT索引的数据列慢很多。

在进行布尔搜索时，还可以给搜索字符串里的单词加上一些修饰符。在单词的前面加上一个加号表示该单词必须出现在匹配数据行里，而加上一个减号表示该单词不得出现在匹配数据行里。比如说，搜索字符串'bell'匹配包含“bell”的数据行，而在布尔模式里，搜索字符串'+bell -candle'只匹配包含单词“bell”、不包含单词“candle”的数据行：

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('+bell -candle' IN BOOLEAN MODE);
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
```

```
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+------------------------------------+
```

后缀的星号“*”将被解释为一个通配符，带星号后缀的搜索单词将匹配以它开头的所有单词。比如说，'soft*'将匹配“soft”、“softly”、“softness”等：

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('soft*' IN BOOLEAN MODE);
+---------------------+-----------------------------------------------------+
| attribution         | phrase                                              |
+---------------------+-----------------------------------------------------+
| Proverbs 15:1       | A soft answer turneth away wrath                    |
| William Shakespeare | But, soft! what light through yonder window breaks?|
| Theodore Roosevelt  | Speak softly and carry a big stick                  |
+---------------------+-----------------------------------------------------+
```

注意，星号通配符不能用来匹配比最小索引单词长度更短的单词。

在附录 C 里，你可以在介绍 MATCH 操作符的部分查到全部的布尔模式修饰符。

和自然语言模式的全文搜索情况相似，布尔模式的全文搜索也将忽略所有的休止单词，就算是把它们标为“必须出现”也是如此。比如说，搜索字符串'+Alexander +the +great'将找出包含“Alexander”和“great”的数据行，“the”会因为它是一个休止单词而被忽略。

2.15.3　全文搜索：查询扩展模式

全文搜索的查询扩展模式将进行两个阶段的搜索。第一遍搜索和普通的自然语言搜索一样，在这次搜索里找到的相关程度最高的数据行里的单词将被用在第二阶段。这些数据行里的单词加上原来那些搜索单词将被用来进行第二遍搜索。因为搜索单词的集合变大了，所以在最终结果里往往会多出一些在第一阶段没被找到、但与第一阶段的检索结果有一定关系的数据行。

要想进行这种搜索，需要在搜索字符串的后面加上 WITH QUERY EXPANSION 短语。下面的例子提供了一个演示。第一条查询命令将进行一次自然语言搜索。第二条查询命令将进行一次查询扩展搜索，这次多找到了一个数据行，但该数据行不包含原始搜索字符串里的任何单词。该数据行会被匹配出来的原因是它包含单词“candle”，这个单词出现在了被自然语言搜索找到的某个数据行里。

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book');
+-----------------------+------------------------------------+
| attribution           | phrase                             |
+-----------------------+------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle             |
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+------------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book' WITH QUERY EXPANSION);
+-----------------------+-------------------------------------+
| attribution           | phrase                              |
+-----------------------+-------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle              |
| Alexander Graham Bell | Mr. Watson, come here. I want you!  |
```

```
| Robert Burton          | I light my candle from their torches. |
+------------------------+---------------------------------------+
```


2.15.4 配置全文搜索引擎

有几个全文搜索参数是可配置的，可以通过设置系统变量的办法来改变。用来为 FULLTEXT 索引设定单词最小和最大长度的参数是 ft_min_word_len 和 ft_max_word_len。长度超出这两个参数所定义的范围的单词在创建 FULLTEXT 索引时将被忽略。默认的最小值和最大值分别是 4 个和 84 个字符。

假设你想把最小单词长度从 4 改成 3，请按以下步骤进行。

(1) 把 ft_min_word_len 变量设置为 3，重启服务器。如果你想让这个设置在服务器每次重启后都能生效，最好的办法是把这个设置放到某个选项文件里，如/etc/my.cnf 文件：

```
[mysqld]
ft_min_word_len=3
```

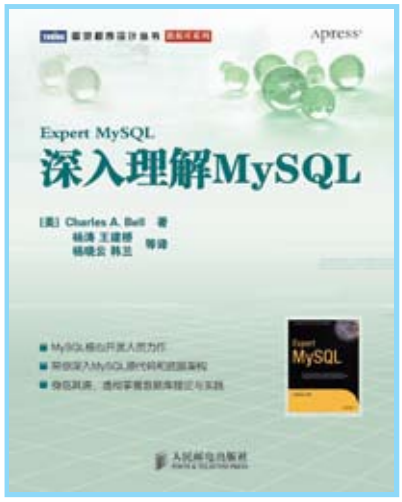
(2) 对于那些已经有 FULLTEXT 索引的现有数据表，你必须重新建立那些索引。你可以先删除、再重新创建，但更简便且同样有效的办法是进行一次快速修复操作：

```
REPAIR TABLE tbl_name QUICK;
```

(3) 在改变参数后创建的新 FULLTEXT 索引将自动使用新值。

关于如何设置系统变量的讨论见附录 D。使用选项文件的细节见附录 E。

说明 如果某个数据表有 FULLTEXT 索引，在使用 myisamchk 工具程序为该数据表重建索引的时候就必须注意一些与 FULLTEXT 索引有关的事项，详见附录 F 对 myisamchk 工具的描述。


Apress
Expert MySQL
深入理解MySQL
[美] Charles A. Bell 著
杨涛 王建桥
杨晓云 韩兰 等译
MySQL核心开发人员力作
帮你深入MySQL源代码和底层架构
借鉴其长，透彻掌握数据库理论与实践
人民邮电出版社


MySQL Crash Course
MySQL必知必会
[英] Ben Forta 著
刘晖霞 钟鸣 译
《SQL必知必会》作者新作
Amazon全五星评价
学习与参考首选
人民邮电出版社


Apress
Beginning PHP and MySQL
From Novice to Professional
PHP与MySQL
程序设计
（第4版）
[美] W. Jason Gilmore 著
朱涛江 等译
人民邮电出版社

“这是我读过的所有技术书中最好的一本，强烈推荐！”

——ACCU的*C Vu*杂志主编Gregory Haley

“本书是最权威的用户指南和参考手册，有了它，在MySQL数据库的日常操作和维护方面，你就会高枕无忧。”

——*Web Techniques*杂志主编Eugene Kim

MySQL Fourth Edition

MySQL技术内幕（第4版）

MySQL是一个开源关系数据库管理系统，现在越来越受到人们的青睐，应用范围也越来越广。MySQL的速度和易用性使其特别适用于Web站点或应用软件的数据库后端的开发工作。

本书是MySQL方面名副其实的圣经级著作，全面介绍了如何高效地使用和管理MySQL。书中主要介绍了如何将数据写入数据库，如何格式化查询，如何使用MySQL和PHP（或者Perl）生成动态网页，如何编写可以访问MySQL数据库的程序，如何管理MySQL服务器等。

Addison Wesley
www.informit.com

图灵网站：www.turingbook.com　热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议　计算机/数据库/MySQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-25595-2
定价：139.00元