

第一部分 MySQL 的使用

第1章 MySQL 与 SQL 介绍

本章介绍 MySQL 关系数据库管理系统（RDBMS）及其所采用的结构化查询语言（SQL）。文中给出了应该掌握的基本术语和概念，并介绍了本书中使用的样例数据库，提供了怎样利用 MySQL 创建数据库并对其进行存取访问的指导。

在此，如果您对数据库不熟悉，可能还不能肯定是否需要一个数据库或是否能够使用一个数据库。或者，如果您对 MySQL 或 SQL 一无所知，需要一种入门性的指导，那么应该仔细阅读本章。对 MySQL 或对数据库系统具有一定经验的读者可能希望跳过这一章。但是，不管是否是初学者都应该阅读 1.2 节“一个样例数据库”，因为这一节中给出的数据库是熟悉数据库的用途和内容的一个最好的样例，本书将要反复地用到它。

1.1 MySQL 的用途

本节介绍 MySQL 的应用场合。提供 MySQL 能够做什么，以何种方式做的一个大致概念。如果您不需要了解数据库的用途，或许您已经在头脑中有了要解决什么问题的想法，只是希望知道怎样用 MySQL 来帮助解决它，那么可以跳到 1.2 节“一个样例数据库”。

数据库系统本质上是一种用来管理信息列表的手段。这些信息可来自不同的地方。例如，它可以代表研究数据、业务记录、顾客请求、运动数据统计、销售报告、个人爱好信息、人事记录、问题报告或学生成绩等。虽然数据库系统能够处理广泛的信息，但您不会仅仅是为用它而用它。如果一项工作很容易，那么就没有理由非得仅为了使用数据库而将数据库引入这项工作。杂货单就是一个很好的例子：开列一个购物清单，购买后在上面画叉，然后将它扔了，极不可能为此事使用一个数据库。即使您有一台便携式电脑，也只会为杂货单使用记事本，而不会启用数据库。

数据库系统的力量只在组织和管理的信息很庞大或很复杂，用手工处理极为繁重时才能显示出来。当然，每天处理数百万个业务的大公司可以使用数据库。但是，即使只涉及个人爱好的单一人员维护信息的小公司也可能需要数据库。不难想像由于在信息变得难于管理之前，使用了数据库而带来的好处。考虑下列情形：

- 您的木工店有几个员工。需要保存员工和工资记录，以便知道给谁付过工资，什么时候付的，并且必须对这些记录进行汇总以便能向税务部门报收益表。还需要明了您的公司雇人所做的工作以及对每项工作所做的安排。
- 您有一个汽车零部件的库房网，需要知道哪些库房中有给定的零件，以便能填写顾客订单。
- 作为玩具销售商，要特别关注所进货物是否流行。需要知道某项物品的当前销售曲线，以便能够估计是否需要增加库存量（对越来越流行的物品），或减少其库存量（从而用

不着存放一大堆销售不好的东西)。

- 多年课题研究收集的大量研究数据需要进行分析以便发表。希望对大量的原始数据进行加工, 得出结论性的信息, 并为更详细的统计分析筛选出观察样本子集。
- 您是位受欢迎的演讲者, 到全国各地的各种集会上进行演讲, 如在毕业典礼、商务会议、城市集会和行政大会上进行讲演。作了这么多讲演, 自己很难记住在什么地方讲了些什么, 因此一定很愿意保存过去讲演的记录, 以帮助准备以后的演说。如果您回到了一个以前曾作过演说的地方, 肯定不愿意作一个与上一次类似的演讲, 到过的地方都有一个记录能帮助您避免重复。您必定也愿意注意讲演受欢迎的程度。(您在“大都会狗窝”俱乐部所做的演讲“我为什么喜欢猫”不太成功, 那么下次去那儿时一定不希望再犯同样的错误。)
- 您是个教师, 需要知道学分和出勤情况。每当您进行测验或考试时, 都要记录学生们的学分。将考试成绩写在学分簿上很容易, 但以后利用这个学分簿却很费事。因此, 在学期末确定最终成绩时, 您宁可不进行学分排序, 而且宁可不汇总每个学生的学分。要统计出每个学生的缺旷课情况也不是一件简单的事。
- 您是某机构的秘书, 这个机构有一个庞大的会员姓名地址簿。(所谓机构可以是任何组织, 如一个专业团体、俱乐部、交响乐团或球迷俱乐部等。)您每年都要根据会员信息变化, 用字处理器进行编辑, 然后为每个会员们打印一个地址名录。

您厌倦了以这种方式维护这个地址簿, 因为它限止了您利用它可做的事。用它难于以不同的方式对各条目排序, 不能方便地选择每个条目的特定部分(如给出仅由姓名和电话号码组成的清单)。也不能查出某组会员, 如那些不久就需要更新其会员资格的人员, 如果可能的话, 应该取消为了找到哪些需要发送补充说明的会员而每个月都要查找所有条目的工作。

而且, 您一定不愿意自己做地址簿的编辑工作, 但是团体没有那么多的预算, 请人会产生问题。您听说过“无纸化办公”, 这是一种导致电子化保存记录的方法, 但您没有看到任何好处。现在会员记录是电子化的, 但具有讽刺意义的是, 除了地址簿的打印外, 没省多少事。

上述情形中有的涉及信息量较大, 有的涉及信息量较小。它们的共同特征都是所涉及的任务可由手工完成, 但是用数据库系统来做会有效得多。

使用如像 MySQL 这样的数据库系统希望看到什么样的效果呢? 这有赖于您的特定需求, 正如上面的例中所看到的那样, 其效果的差异是相当大的。我们来考虑一种常见的情形, 从而也是一种相当有代表性的数据库应用。

通常利用数据库管理系统来处理诸如人们用文件柜来完成的那样一类的任务。确实在某种意义上说, 数据库就像一个大文件柜, 只不过是一个内建的文件编排系统而已。电子化处理记录相对手工处理记录有很多优点。例如, 如果您在某种保存有客户记录的办公设施内工作, 那么 MySQL 可在某些方面向您提供帮助:

- 减少记录编档时间。不必为寻找增加新记录的位置而查看橱柜的所有抽屉。只要将记录放入文件编排系统, 并令文件编排系统为您将该记录放入正确的位置即可。
- 减少记录检索时间。在查找记录时, 不需要自己去寻看每个记录以找到含有所需信息的那个记录。假如您在一个牙科诊所中工作。如果想给所有近来未到诊断做过检查的病人发催询单, 只要求文件编排系统查找合适的记录即可。当然, 这样做会有别于吩咐别人去做。吩咐别人去做, 您只需说, “请确定哪些病人最近 6 个月内没来过。”

而使用数据库，则需要发出一串奇怪的“咒语”：

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

如果您从来没有看到过类似的东西，可能会感到相当吓人，但是在一两秒内就能得到结果远胜于用一个小时来查找，这应该是很有吸引力的。（不管怎样，不用太发愁。这些“咒语”用不了多久就会不奇怪了。事实上，只要您读完本章就能完全理解其含义。）

- 灵活的查找序列。不需要按记录存放的固定序列去查看它们（例如，按姓查找）。可以要求文件编排系统以任意的序列查出记录；如按姓、保险公司名、最后光临日期等提出记录。
- 灵活的输出格式。在查找到感兴趣的记录后，不需要手工拷贝其信息。可以让文件编排系统为您生成一份清单。有时，您可能只需要打印这些信息。有时，您又可能希望在其他程序中使用这些信息。（如，在生成误了看牙预约的病人清单后，可将这些信息送入一个字处理器，打出送给这些病人的通知单。）或者您只对汇总信息感兴趣，如对所选出记录数感兴趣。不必自己数它们；文件编排系统可自动生成汇总。
- 多个用户同时访问记录。对纸上的记录，如果两个人想同时查找一个记录，那么其中一个人必须等另一个人找完才能查找。MySQL 提供多个用户同时查找的能力，从而两个人可同时访问记录。
- 记录的远程访问与电子传输。纸面记录需要有该记录在手边才能使用，或者需要有人做拷贝再发送给您。而电子记录可以远程访问或进行电子化传输。如果您的牙医专家在多个诊所工作，那么他们可从自己的所在地访问您的记录，不需要给他们发快信。如果需要记录的某个人没有与您的数据库软件相同的软件，但有电子邮件，那么您可以选择所需的记录，用电子文档发送。

如果您以前使用过数据库管理系统，已经了解数据库的上述诸般好处，可能会想，怎样才能超越“取代文件柜”的用途。现在，数据库系统已经可以用来提供过去不能，直到最近才能够提供的服务。例如，许多机构以一种与 Web 结合的方式使用数据库，这种方式过去是做不到的。

假如您的公司有一个库存数据库，在顾客询问库房中是否有某项物品，它的价格是多少时，服务台人员使用这个数据库，这是数据库的一种较为传统的应用。但是，如果您的公司向顾客提供一个可供访问的 Web 站点，那么可以提供另一项服务，即：提供一个允许顾客确定物品价格和可得性的搜索页。这给顾客提供了他们所需的信息，提供的方法是让顾客自动地搜索存放在库存中的物品信息。顾客可以立即得到信息，不用听预先录好的音，或受服务台是否正在工作的限制。对于每个使用您的 Web 站点的顾客，所花的费用比服务台工作人员转接电话的费用还少。（或许，该 Web 站点已为这个付了费。）

还有比上述更好的利用数据库的方法。基于 Web 的库存查询请求可以不仅仅为顾客提供信息，而且还可以为您自己提供信息。该查询请求告诉您顾客在找什么，而查询的结果又可以让您知道能否满足他们的请求。您可能在不能满足顾客需求的方面丧失商机。因此，记录有关库存搜索的信息是很有意义的，如记录：顾客在找什么、库存有没有。然后，可以利用这些信息调整您的库存，更好地为顾客提供服务。

数据库的另一新用途是在 Web 页上做标题广告。我也和您一样不喜欢它们，但事实这是一种很流行的 MySQL 应用，可用 MySQL 来存储广告，然后检索它们为 Web 服务器的显

示而用。此外，MySQL 还可以用来进行跟踪，这种跟踪涉及哪些广告起了作用、它们被显示了多少次、哪个站点访问了它们等等信息。

因此，知道如何利用 MySQL 最好的办法是自己试试，为此目的您应该有一个试验性的数据库。

1.2 一个样例数据库

本节介绍一个样例数据库，这个数据库在本书各个部分都可能用到。在学习将 MySQL 投入工作时，这个数据库为您提供了参考的例子。我们主要从前面描述过的两种情形来给出例子：

- 机构的秘书方案。我们需要一些比“机构”更为明确的信息，所以现在就来构造一个，它具有这样一些特性：它由为了研究美国历史这个共同目的而聚集在一起的一群人组成（一时找不到更好的名称，就暂且称为美国历史同盟）。在交会费的基础上定期更新各会员的资格。会费构成了此同盟的活动经费，如出版报纸“美国编年历”。此联盟也有一个小 Web 站点，但开发出的功能不多。迄今这个站点只限于提供一些基本的信息，如本团体的性质，负责人是谁，什么样的人可以参加等。

- 学分保持方案。在学分时段中，需要管理被测试者、记录得分并赋予得分等级。然后确定最后的得分等级，将其与出勤率一道交给学校办公室。

现在让我们根据如下两个要求来进一步考虑这些情况：

- 必须确定希望从数据库中得到什么信息，即，希望达到什么目的。

- 必须计划好要向数据库输入什么，即将要保存什么数据。

或许，在考虑向数据库输入什么数据以前，逆向考虑一下需要从数据库输出什么数据。在能够对数据进行检索前，必须将数据送入数据库。但是，使用数据库的方法是受您的目标驱动的，这些方法与希望从数据库取出何种信息的关系较之与向数据库输入何种信息的关系更为紧密。除非打算以后使用这些信息，否则肯定不会浪费时间和精力将它们输入数据库。

1.2.1 美国历史同盟

这个方案的初期状况是您作为同盟的秘书，利用字处理文档维护会员清单。这样就生成一个打印的姓名地址录来说还是可以应付的，但是在利用这些信息做别的事时就会受到限制。假定您打算做下列工作：

- 希望能够利用该姓名地址录产生不同格式的输出，并且只给出相应用途所需的信息。目标之一是生成每年的打印姓名地址录，这是该同盟过去就需要的，您打算继续打印。除此之外，可以设想将姓名地址录中的信息派一些别的用途，如在全盟的年度宴会上所提供的节目单中给出一个当前的会员清单。这个应用涉及不同的信息集合。打印的姓名地址录中使用了每个会员条目的所有内容。而对于宴会节目单，只需要取出会员名字即可（如果采用字处理器要做到这一点有时是不太容易的）。

- 希望搜索姓名地址录查找其条目满足某些条件的会员。例如，希望知道哪些会员不久就需要更新其会员资格。另外涉及搜索的应用是由于需要维护每个会员的关键字列表而产生的。这些关键字描述了每个会员特别感兴趣的美国历史的某个方面（如内战、经济萧条、公民权利或托马斯·杰佛逊的生活等）。会员们有时会向您要一份与他们自

己有类似爱好的会员的清单，您一定乐于满足他们的这种要求。

- 希望让姓名地址名录在联盟的 Web 站点上联机使用。这对会员和您都是很有好处的。如果您能够将姓名地址录用某种合适的自动过程转换为 Web 页，则这个姓名地址录的联机版就可以以一种比打印版更及时的方式保持最新信息。而且如果能使这个联机姓名地址录可供搜索，那么会员就能够自己方便地查找信息了。例如，某个会员希望知道其他对内战感兴趣的会员，他就可以自己将这些会员找出而不用请您帮他查找，而您也不用花时间去做了。

我们清楚地知道，数据库并不是世界上最令人激动的东西，因此，我们也不打算狂热地声称，使用数据库可以促进创造性的思维。但是，当您停止将信息视为某种必须与之搏斗的东西（在用字处理文档时确实是这样的），并开始将其想像为某种可以相对容易地操纵的事物（正如希望用 MySQL 所做到的那样）时，您提出某种使用或表示信息的新方法的能力将会得到某种程度的解放，例如下面这些例子就是一些新方法：

- 如果数据库中的信息能够以联机姓名地址录的形式移到 Web 站点中，那么您可能会让信息以其他的方式流动。例如，如果会员能够联机编辑自己的条目，对数据库进行更新，那么您就不必自己做所有的编辑工作了，这样有助于使姓名地址录中的信息更为准确。
- 如果您在数据库中存储 Email 地址，那么可以利用它们来发送 Email 给那些相当长的一段时间没有更新自己的条目的会员。发出的消息可以向这些会员显示他们的条目内容，请他们查看，然后指示怎样利用 Web 站点提供的实用工具做所需的修改。
- 数据库不仅以关联到会员表的方式帮助使 Web 站点更为有用。比方说，联盟出版了一份报纸“美国编年史”，每一期中都有一个给小孩子的版面，内含历史试题。最近有几期主要集中在美国总统的传记上。联盟的 Web 站点也可以包含给孩子的版面，这样使试题联机。通过放置从数据库中取出的试题并让 Web 服务器对随机给出的问题进行查询，或许甚至可以使这个版面成为交互式的。

至此，您可能已经想起了许多数据库的用途，这使您有点不能自控了。在回到现实之前，您开始问一些特殊的问题：

- 这是不是有点野心勃勃了？在准备时是不是要做大量的工作？当然，如果只是想而不去做，则任何事情都很简单，我并不伪称上述所有事情实现起来都是微不足道的。然而，在本书结束时，我们所描述的这些事都实现了。只需记住一件事，没必要一次做完所有的事。我们将对工作进行分解，每次只做一部分。
- MySQL 能够完成所有这些事吗？不，它不能够。例如，MySQL 没有直接的 Web 能力。虽然由 MySQL 自身不能完成我们所讨论的每样事情，但是可以得到与 MySQL 一起工作的工具，从而完善和扩展了 MySQL 的能力。

我们将用 Perl 脚本语言和 DBI（数据库接口）Perl 模块来编写访问 MySQL 数据库的脚本。Perl 具有极为出色的文本处理能力，它允许以一种高度灵活的方式处理查询结果以产生各种格式的输出。例如，我们可以用 Perl 来生成多信息文本格式（RTF）的姓名地址录，这是一种可被所有字处理器读取的格式。

我们也可以使用另一种脚本语言 PHP。PHP 特别适合于编写 Web 应用，而且它与数据库一起工作。这使得能从 Web 页运行 MySQL 查询并生成包含数据库查询结果的新页。PHP 与

Apache（世界上最流行的 Web 服务器）一起工作得很好，这使得完成诸如给出一个搜索窗口并显示搜索结果之类的事情很容易。

MySQL 与这些工具集成得很好，并向您提供了以自己的方式组合它们的灵活性，可以进行选择以实现您的设想。不用受限于那些大肆推销的所谓“集成”功能而实际工作起来也只是彼此之间的固定组合。

- 最后，有一个大问题，那就是所有这些东西要花多少钱？首先，同盟的预算是有限的。回答是，大概什么钱也不用花，这可能会令您吃惊。如果您熟悉一般的数据库系统，就会知道，它们一般相当昂贵。但是，MySQL 一般是免费的。在某些环境下，确实不需要许可证，而且如果用户数量不限也只需花 \$200。（关于许可证的一般介绍请参阅前言，特定的细节可参阅 MySQL 参考指南。）我们将使用的其他工具（Perl、DBI、PHP、Apache）也是免费的，因此，所有东西都考虑到了，可以相当便宜地组成一个有用的系统。

开发这个数据库的操作系统的选择取决于您。我们介绍的所有软件都可运行在 UNIX 下，其中大多数可以运行在 Windows 下。作者推荐在 UNIX 下运行 MySQL 和其他工具。它们全都是发源于 UNIX 下，然后才转到 Windows 的。这表示它们的 Windows 版本成熟期较短，尚未经过彻底的测试和使用。

现在，让我们来考虑一下使用样例数据库的其他情形。

1.2.2 学分保存方案

初步的想法是，作为一个老师，有保存学分的职责。老师希望将学分处理从学分簿上的手工操作转到 MySQL 上用电子表示。在此情形下，想从数据库得到的是含在学分簿中的东西：

- 对于每次测验或测试，要记录学分。对测试，将学分排序，以便能确定每个字符（A、B、C、D 和 F）所代表等级的得分范围。
- 在学分时段结束时，计算每个学生的总得分，然后排序总的得分并根据它们确定得分等级。总的得分可能涉及权重计算，因为大概会希望使测试的得分比测验和得分权重更大。
- 在每个学分时段结束时，提供出勤信息给学校办公室。

目的是避免手工排序和汇总学分及出勤率记录。换句话说，希望 MySQL 在学分时段结束时对学分排序并完成每个学生的总分和缺课数的计算。为了达到这个目的，需要班级中的学生名册、每次测验和测试的分数以及学生缺课的日期。

1.2.3 样例数据库怎样才能满足需求

如果您对历史同盟或学分保存不太感兴趣，可能会奇怪为什么必须做这些例子呢？答案是这些样例方案本身并不是目的，只是用它们说明利用 MySQL 及其相关的工具能做什么事。

加上一点想像，您将会看到样例数据库的查询怎样应用到所希望解决的问题上。假设您在前面提到的牙科诊所上班，将会在本书中看到许多牙科方面的查询。例如，确定历史同盟的哪些会员需要立即更新他们的会员资格，这是一件类似于确定哪些病人近来没有来看牙医的事情。两者都是基于日期的查询，因此，一旦学会了编写会员更新的查询，便可以将该技

术用来编写更为感兴趣的延误的预约病人查询。

1.3 基本数据库术语

您可能会注意到，已经读了本书这么多页，但是还没有看到几句行话和术语。虽然我们大致提了一下怎样利用样例数据库，但事实上，关于什么是“数据库”，我们一点东西都还没有介绍。不过，我们现在打算设计该数据库，然后开始实现它，这样就不能再避而不谈数据库术语了。介绍数据库术语就是本节的目的。本节介绍的一些术语全书都要用到，因此必须对其熟悉。所幸的是，关系数据库中的许多概念是相当简单的。事实上，关系数据库的吸引力主要来源于其基本概念的简单性。

1.3.1 基本术语

在数据库世界中，MySQL 归类为关系数据库管理系统（RDBMS）。所谓关系数据库管理系统的含义如下：

- 数据库（RDBMS 中的“DB”）是存储信息的仓库，以一种简单的、规则的方式进行组织：

- 数据库中的数据集组织为表。
- 每个表由行和列组成。
- 表中每行为一个记录。
- 记录可包含几段信息；表中每一列对应这些信息中的一段。

- 管理系统（“MS”）是允许通过插入、检索、修改或删除记录来使用数据的软件。

- “关系”（“R”）一词表示一种特殊种类的 DBMS，它通过寻找相互之间的共同元素使存放在一个表中的信息关联到存放在另一个表中的信息。关系数据库的能力在于它能够从这些表中方便地取出数据，并将关联各表中的信息相结合得出问题的答案，这些答案只依据单个表的信息是不可能得到的。

这里有一个例子，示出了关系数据库怎样将数据组织成表并将一个表中的信息与另一个表中的信息相关联。假定您管理一个含有标题广告服务的 Web 站点。您与公司有协议，这些公司希望有人在拜访您的站点上的网页时显示他们的广告。每当一个拜访者点击您的页面一次，您就向该拜访者的浏览器提供了嵌在页面中的广告的一次服务，并且给公司估算一点费用。为了表示这些信息，要保存三个表（请参阅图 1-1）。一个是 company 表，它含有公司名、编号、地址和电话号码等列。另一个是 ad 表，它列出广告编号、拥有该广告的公司编号以及每次点击时的计费数。第三个 hit 表按广告编号记录广告点击次数以及广告提供服务的日期。

利用单个表的信息可以回答某些问题。为了确定签协议的公司数目，只需对 company 表中的行数计数即可。类似地，为了确定某个给定时间段中的点击次数，只需查看 hit 表即可。其他问题要更为复杂一些，而且必须考虑多个表以确定答案。例如，为了确定 Pickles 公司的每个广告在7月14日点击了多少次，应该按如下步骤使用这些表：

- 1) 查询 company 表中的公司名（Pickles, Inc）以找到公司编号（14）。
- 2) 利用公司编号查找 ad 表中匹配的记录以便能够确定相关的广告编号。有两个这样的广告，48 和 101。

3) 对 ad 表中匹配的记录，利用该记录中的广告编号查找 hit 表中在所需日期范围内的匹配记录，然后对匹配的记录进行计数。广告编号为 48 的匹配记录有三个，广告编号为 101 的匹配记录有两个。

听起来很复杂！而这正是关系数据库系统所擅长的。这种复杂性在某种程度可以说是一种幻觉，因为上述每一步只不过是一个简单的匹配操作，它通过将表的行中的值与另一个表的行中的值相匹配，把一个表与另一个表相关联。这个简单的操作可以各种方式使用来回答各种各样的问题。每个公司有多少个不同的广告？哪个公司的广告最受欢迎？每个广告带来的收入是多少？当前记账期中每个公司的总费用是多少？

现在我们已经介绍了关系数据库的理论，足以理解本书其余部分了，我们不必探究第三范式、实体关系图以及所有这一类的东西。如果您确实需要了解这些东西，那就太令人恐怖了，而且这也不是地方。建议您从阅读 C.J.Date 和 E.F.Codd 的某些书籍入手。

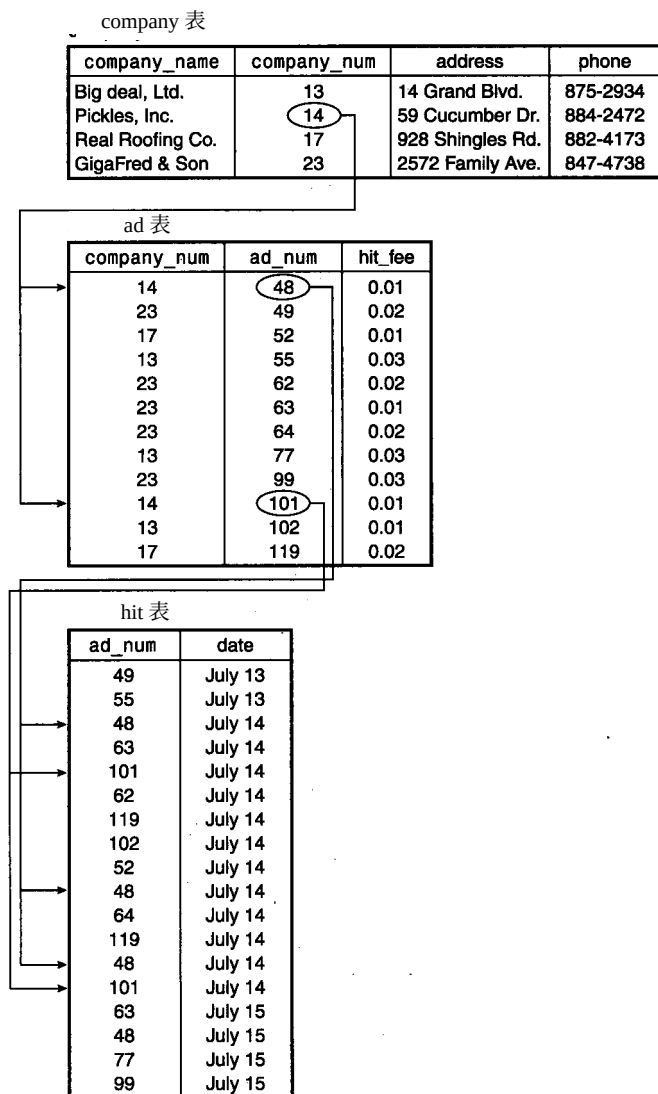


图1-1 标题广告的各表

1.3.2 查询语言术语

MySQL使用一种称为 SQL (Structured Query Language) 的语言。SQL 是当今的标准数据库语言, 所有主要的数据库系统都使用它。SQL 具有多种不同的语句, 所有语句都是以一种不枯燥并有用的方式设计来与数据库进行交互的。

正如其他语言一样, SQL 在初次接触时可能会令人感到有些古怪。例如, 为了创建一个表, 需要告诉 MySQL 表结构应该是什么样的。我们可能会根据图表来想像一个表, 但 MySQL 不会, 因此, 在创建表时需要告诉 MySQL 一些东西, 如下所示:

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

如果您不熟悉 SQL 语句, 可能会对这样的语句留下深刻的印象, 但您不必以程序员的身份来学习怎样有效地使用 SQL。如果逐步熟悉了 SQL 语言之后, 就会以一种不同的眼光来看待 CREATE TABLE 语句, 会认为它是一个有助于描述自己信息的伙伴, 而不是一种奇怪的胡言乱语。

1.3.3 MySQL 的体系结构术语

在您使用 MySQL 时, 实际正使用以下两个程序, 因为 MySQL 采用的是客户机/服务器体系结构:

- 数据库服务器是一个位于存放您的数据的机器上的程序。它监听从网络上传过来的客户机的请求并根据这些请求访问数据库的内容, 以便向客户机提供它们所要求的信息。
- 客户机是连接到数据库服务器的程序, 这些程序告诉服务器需要什么信息的查询。

MySQL 分发包括服务器和几个客户机程序。可根据要达到的目的来使用客户机。最常用的客户机程序为 mysql, 这是一个交互式的客户机程序, 它能发布查询并看到结果。其他的客户机程序有: mysqldump 和 mysqlimport, 分别转储表的内容到某个文件或将文件的内容导入某个表; mysqladmin 用来查看服务器的状态并完成管理任务, 如告诉服务器关闭等。如果具有标准的客户机不适合的应用, 那么 MySQL 还提供了一个客户机编程库, 可以编写自己的程序。客户机编程库可直接从 C 程序中调用, 如果希望使用 C 语言以外的其他语言, 还有几种其他的接口可用。

MySQL 的客户机/服务器体系结构具有如下好处:

- 服务器提供并发控制, 使两个用户不能同时修改相同的记录。所有客户机的请求都通过服务器处理, 服务器分类辨别谁准备做什么, 何时做。如果多个客户机希望同时访问相同的表, 它们不必互相裁决和协商, 只要发送自己的请求给服务器并让它仔细确定完成这些请求的顺序即可。
- 不必在数据库所在的机器上注册。MySQL 知道怎样在因特网上工作, 因此您可以在任何位置运行一个客户机程序, 此客户机程序可以连接到网络上的服务器。距离不是问题, 可从世界上的任何地方访问服务器。如果服务器位于澳大利亚的某台机器上, 那么当您带着自己的便携式电脑到冰岛去旅行时, 仍然可以访问自己的数据库。

这是否意味着任何人只要连接到因特网就可以访问您的数据？答案是否定的。MySQL 含有一个灵活的安全系统，只允许那些有权限访问数据的人访问。可以保证那些人只能做允许他们做的事。或许记账办公室的 Sally 能够读取和更新（修改）记录，而服务台的 Phil 只能查看记录。可以设置使用人员的权限。如果希望运行一个自含系统（独立系统），只要设置访问权限使客户机只能从服务器运行的主机上进行连接即可。

1.4 MySQL 教程

现在我们已经具备了所需的所有基础知识；可以将 MySQL 投入工作了！

本节提供一个教程，帮助熟悉 MySQL。在完成这个教程时，将创建一个样例数据库和这个数据库中的表，然后增加、检索、删除和修改信息与数据库进行交互。此外，在操作这个样例数据库的过程中，将能学到下列东西：

- 如何利用 mysql 客户机程序与 MySQL 通信。
- SQL 语言的基本语句。（如果您曾经使用过其他 RDBMS，从而熟悉 SQL，那么浏览一下这个教程，看看 SQL 的 MySQL 版与您熟悉的版本有何差别也是很好的。）

正如上一节所述，MySQL 采用客户机/服务器体系结构，其中服务器运行在存放数据库的机器上，而客户机通过网络连接到服务器。这个教程主要基于 mysql 客户机的应用。mysql 读取您的 SQL 查询，将它们发送给服务器，并显示结果。mysql 运行在 MySQL 所支持的所有平台上，并提供与服务器交互的最直接的手段，因此，它首先是一个逻辑上的客户机。

在本书中，我们将用 samp_db 作为样例数据库的名称。但是有可能在您完成本例子的过程中需要使用另一个数据库名。因为可能在您的系统上已经有某个人使用了 samp_db 这个名称，或者管理员给您指定了另一个数据库名称。在后面的例子中，无论是哪种情况，都用数据库的实际名称代替 samp_db。

表名可以像例子所显示的那样精确地使用，即使系统中的多个人都具有他们自己的样例数据库也是如此。顺便说一下，在 MySQL 中，如果有人使用了相同的表名也没什么关系。一旦各个用户都具有自己的数据库，MySQL 将一直保留这些数据库名，防止各用户互相干扰。

1.4.1 基本要求

为了试验这个教程中的例子，必须安装 MySQL。特别是必须具有对 MySQL 客户机和某个 MySQL 服务器的访问权。相应的客户机程序必须位于您的机器上。至少需要有 mysql 程序，最好还有 mysqlimport 程序。服务器也可以位于您的机器上，尽管这不是必须的。实际上，只要允许连接到服务器，那么服务器位于何位置都没有关系。

若服务器正巧运行在您的机器上，适当的客户机程序又已经安装，那么就可以开始试验了。如果您尚需设法搞到 MySQL，可参阅附录 A“获得和安装软件”的说明。如果您正自己安装 MySQL，可参阅这一章，或把它给管理员看。如果网络访问是通过一个因特网服务商（ISP）进行的，那么可查看该服务商是否拥有 MySQL。如果该 ISP 不提供 MySQL 服务，可查看附录 J“因特网服务商”以得到某些选择更适合的服务商的建议。

除 MySQL 软件外，还需要得到创建样例数据库及其表的权限。如果您没有这种权限，可以向 MySQL 管理员咨询。管理员可通过运行 mysql 并发布如下的命令提供这种权限：

```
GRANT ALL ON samp_db.* TO paul@localhost IDENTIFIED BY "secret"  
GRANT ALL ON samp_db.* TO paul@% IDENTIFIED BY "secret"
```

MySQL 与 mysql 的区别

为了避免混淆，应该说明，“MySQL”指的是整个 MySQL RDBMS，而“mysql”代表的是一个特定的客户机程序名。它们的发音都是相同的，但可通过不同的大小写字符和字体来区分。

关于发音，MySQL 的发音为“my-ess-queue-ell”。我们知道这是因为 MySQL 参考指南中是这样发音的。而 SQL 的发音为“sequel”或“ess-queue-ell”。我不认为哪个发音更好一些。愿意读哪个音都可以，不过在您对别人读的时候，他可能会用他认为是“正确”的发音对您进行纠正。

前一个命令在 paul 从 localhost（服务器运行在正运行的同一主机）连接时，允许它完全访问 samp_db 数据库及它的所有表。它还给出了一个口令 secret。第二个命令与第一个类似，但允许 paul 从任何主机上连接（“%”为通配符）。也可以用特定的主机名取代“%”，使 paul 只能从该主机上进行连接。（如果您的服务器允许从 localhost 匿名访问，由于服务器搜索授权表查找输入连接匹配的方式的原因，这样一个 GRANT 语句可能是必须的。）关于 GRANT 语句以及设置 MySQL 用户账号的更详细信息，可在第 11 章“常规的 MySQL 管理”找到。

1.4.2 取得样例数据库的分发包

这个教程在某些地方要涉及来自“样例数据库分发包”中的文件。有的文件含有帮助来设置样例数据库的查询或数据。为了得到这个分发包，可参阅附录 A。在打开这个分发包时，将创建一个名为 samp_db 的目录，此目录中含有所需的文件。无论您在哪个地方试验与样例数据库有关的例子，建议都移入该目录。

1.4.3 建立和中止与服务器的连接

为了连接到服务器，从外壳程序（即从 UNIX 提示符，或从 Windows 下的 DOS 控制台）激活 mysql 程序。命令如下：

```
% mysql options
```

其中的“%”在本书中代表外壳程序提示符。这是 UNIX 标准提示符之一；另一个为“\$”。在 Windows 下，提示符类似“c:\>”。

mysql 命令行的 options 部分可能是空的，但更可能的是发布一条类似如下的命令：

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

在激活 mysql 时，有可能不必提供所有这些选项；确切使用的命令请咨询 MySQL 管理员。此外，可能还需要至少指定一个名称和一个口令。

在刚开始学习 MySQL 时，大概会为其安全系统而烦恼，因为它使您难于做自己想做的事。（您必须取得创建和访问数据库的权限，任何时候连接到数据库都必须给出自己的名字和口令。）但是，在您通过数据库录入和使用自己的记录后，看法就会马上改变了。这时您会很欣赏 MySQL 阻止了其他人窥视（或者更恶劣一些，破坏！）您的资料。

下面介绍选项的含义：

■ -h host_name（可选择形式：--host=host_name）

希望连接的服务器主机。如果此服务器运行在与 mysql 相同的机器上，这个选项一般可省略。

■ **-u user_name** (可选择的形式：**--user=user_name**)

您的 MySQL 用户名。如果使用 UNIX 且您的 MySQL 用户名与注册名相同，则可以省去这个选项；mysql 将使用您的注册名作为您的 MySQL 名。

在 Windows 下，缺省的用户名为 ODBC。这可能不一定非常有用。可在命令行上指定一个名字，也可以通过设置 USER 变量在环境变量中设置一个缺省名。如用下列 set 命令指定 paul 的一个用户名：

```
set USER=paul
```

■ **-p** (可选择的形式：**--password**)

这个选项告诉 mysql 提示键入您的 MySQL 口令。注意：可用 **-pyour_password** 的形式 (可选择的形式：**--password=your_password**) 在命令行上键入您的口令。但是，出于安全的考虑，最好不要这样做。选择 **-p** 不跟口令告诉 mysql 在启动时提示您键入口令。例如：

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

```
Enter password:
```

在看到 Enter password: 时，键入口令即可。(口令不会显到屏幕，以免给别人看到。) 请注意，MySQL 口令不一定必须与 UNIX 或 Windows 口令相同。

如果完全省略了 **-p** 选项，mysql 就认为您不需要口令，不作提示。

请注意：**-h** 和 **-u** 选项与跟在它们后面的词有关，无论选项和后跟的词之间是否有空格。而 **-p** 却不是这样，如果在命令行上给出口令，**-p** 和口令之间一定不加空格。

例如，假定我的 MySQL 用户名和口令分别为 paul 和 secret，希望连接到在我注册的同一机器上运行的服务器上。下面的 mysql 命令能完成这项工作：

```
% mysql -u paul -p
```

```
Enter password: *****
```

在我键入命令后，mysql 显示 Enter password: 提示键入口令。然后我键入口令 (***** 表明我键入了 secret)。

如果一切顺利的话，mysql 显示一串消息和一个 “mysql>” 提示，表示它正等待我发布查询。完整的启动序列如下所示：

```
% mysql -u paul -p
```

```
Enter password: *****
```

```
Welcome to the MySQL monitor. Commands end with ; or \g.
```

```
Your MySQL connection id is 1805 to server version: 3.22.25-log
```

```
Type 'help' for help.
```

```
mysql>
```

为了连接到在其他某个机器上运行的服务器，需要用 **-h** 指定主机名。如果该主机为 pit-viper.snake.net，则相应的命令如下所示：

```
% mysql -h pit-viper.snake.net -u paul -p
```

在后面的说明 mysql 命令行的多数例子中，为简单起见，我们打算省去 **-h**、**-u** 和 **-p** 选项。并且假定您将会提供任何所需的选项。

有很多设置账号的方法，从而不必在每次运行 mysql 时都在连接参数中进行键入。这个问题在 1.5 节 “与 mysql 交互的技巧” 中介绍。您可能会希望现在就跳到该节，以便找到一些更易于连接到服务器的办法。

在建立了服务器的一个连接后，可在任何时候键入下列命令来结束会话：

```
mysql> QUIT
Bye
```

还可以键入 Control-D 来退出，至少在 UNIX 上可以这样。

1.4.4 发布查询

在连接到服务器后，就可以发布查询了。本节介绍有关与 mysql 交互应该了解的一些知识。

为了在 mysql 中输入一个查询，只需键入它即可。在查询的结尾处，键入一个分号（“；”）并按 Enter 键。分号告诉 mysql 该查询是完整的。（如果您喜欢键入两个字符的话，也可以使用 “\g” 终止查询。）

在键入一个查询之后，mysql 将其发送到服务器上。该服务器处理此查询并将结果送回 mysql，mysql 将此结果显示出来。

下面是一个简单的查询例子和结果：

```
mysql> SELECT NOW();
+-----+
| NOW() |
+-----+
| 1999-07-24 11:02:36 |
+-----+
1 row in set (0.00 sec)
```

它给出当前的日期和时间。（NOW() 函数本身并无多大用处，但可将其用于表达式中。如比较当前日期和其他日期的差异。）

mysql 还在结果中显示行数计数。本书在例子中一般不给出这个计数。

因为 mysql 需要见到分号才发送查询到服务器，所以在单一的行上不需要键入分号。如果有必要，可将一个查询分为几行，如下所示：

```
mysql> SELECT NOW(),
-> USER(),
-> VERSION()
-> ;
+-----+-----+-----+
| NOW() | USER() | VERSION() |
+-----+-----+-----+
| 1999-07-24 11:06:16 | paul@localhost | 3.23.1-alpha-log |
+-----+-----+-----+
```

请注意，在键入查询的第一行后，提示符从 ‘mysql’ 变成了 ‘->’；这表示 mysql 允许继续键入这个查询。这是一个重要的提示，因为如果在查询的末尾忘记了分号，此提示将有助于提醒您查询尚不完整。否则您会一直等下去，心里纳闷为什么 mysql 执行查询为什么这么长的时间还没完；而 mysql 也搞不清为什么结束查询的键入要花您那么多的时间！

大部分情况下，用大写字符、小写字符或大小写字符混合键入查询没什么关系。下列查询全是等价的：

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

本书中的例子用大写字符表示 SQL 关键字和函数名，用小写字符表示数据库、表和列名。

如果在查询中调用一个函数，在函数名和后跟的圆括号中间不允许有空格，例：

```
mysql> SELECT NOW();
+-----+
| NOW() |
+-----+
| 1999-07-17 12:44:52 |
+-----+
```

```
mysql> SELECT NOW ();
```

```
ERROR 1064: You have an error in your SQL syntax near '()' at line 1
```

这两个查询看上去差别不大，但第二个失败了，因为圆括号并没有紧跟在函数名的后面。

如果已经开始键入一个多行的查询，而又不想立即执行它，可键入 ‘\c’ 来跳过（放弃）它，如：

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

请注意，提示符又变回了 ‘mysql>’，这表示 mysql 为键入的新查询作好了准备。

可将查询存储在一个文件中并告诉 mysql 从文件中读取查询而不是等待键盘输入。可利用外壳程序键入重定向实用程序来完成这项工作。例如，如果在文件 my_file.sql 中存放有查询，可如下执行这些查询：

```
% mysql < my_file.sql
```

可用这种办法调用任何所需的文件。这里用后缀为 “.sql” 来表示该文件含有 SQL 语句。

执行 mysql 的这种方法将在输入数据到 samp_db 数据库时的“增加新记录”中使用。为了装载一个表，让 mysql 从某个文件中读取 INSERT 语句比每次用手工键入这些语句更为方便。

本教程的其余部分向您提供了许多可以自己试试的查询。这些查询以 ‘mysql>’ 提示为前导后跟结束分号，这些例子通常都给出了查询输出结果。可以按给出的形式键入这些查询，所得到的结果应该与自学材料中的相同。

给出的查询中无提示符的或无分号语句结束符的只是用来说明某个要点，不用执行它们。（如果愿意您可以试一下，但如果试的话，请记住给语句末尾加一个分号。）

本书后面的章节中，我们一般不给出 ‘mysql>’ 提示或 SQL 语句的分号。这样做的原因是为了可以在非 mysql 客户机程序的语言环境（如在 Perl 脚本中或 PHP 脚本中）中发布查询，在这些语言环境中，既无提示符也不需要分号。在专门针对 mysql 输入一个查询的场合会作出相应的说明。

1.4.5 创建数据库

现在开始创建 samp_db 样例数据库及其表，填充这些表并对包含在这些表中的数据进行一些简单的查询。

使用数据库涉及几个步骤：

- 1) 创建（初始化）数据库。
- 2) 创建数据库中的表。
- 3) 对表进行数据插入、检索、修改或删除。

检索现有数据是对数据库执行的最简单且常见的操作。另外几个最简单且常见的操作是

插入新数据、更新或删除现有数据。较少使用的操作是创建表的操作，而最不常用的操作是创建数据库。

我们将从头开始，先创建数据库，再插入数据，然后对数据进行检索。

为了创建一个新的数据库，用 `mysql` 连接到数据库然后发布 `CREATE DATABASE` 语句，此语句指定了数据库名：

```
mysql> CREATE DATABASE samp_db;
```

在创建表以及对这些表进行各种操作之前，必须先创建 `samp_db` 数据库。

创建数据库后，这个新创建的数据库并不是当前数据库。这可从执行下面的查询看出：

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

为了使 `samp_db` 成为当前数据库，发布 `USE` 语句即可：

```
mysql> USE samp_db
```

`USE` 为少数几个不需要终结符的语句之一，当然，加上终结符也不会出错。`HELP` 是另一个不需要终结符的语句。如果了解不需要终结符的语句有哪些，可发布 `HELP` 语句。

在发布了 `USE` 语句后，`samp_db` 成为缺省数据库：

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

使数据库成为当前数据库的另一个方法是在激活 `mysql` 时在命令行上指定它，如下所示：

```
% mysql samp_db
```

事实上，这是一个命名要使用的数据库的方法。如果需要连接参数可在数据库名前指定。例如，下列两个命令使我们能连接到在本地主机和 `pit-viper.snake.net` 上的 `samp_db` 数据库上：

```
% mysql -u paul -p samp_db
% mysql -h pit-viper.snake.net -u paul -p samp_db
```

除非另有指定，否则后面的例子都假定在激活 `mysql` 时，在命令行上给出 `samp_db` 使其成为当前数据库。如果激活数据库时忘了在命令行上指定数据库，只需发布 `USE samp_db` 语句即可。

1.4.6 创建表

本节中，我们将创建样例数据库 `samp_db` 所需的表。我们首先考虑美国历史同盟需要的表。然后再考虑学分保存方案所需的表。在某些数据库的书籍中，在这里要大讲分析与设计、实体—关系图、标准程序以及诸如此类的东西。这里确实也可以讲这些东西，但是我宁可只讲点实用的东西，比方说，我们的数据库应该是怎样的：数据库中将包含什么内容，每个表中有哪些数据以及由决定如何表示数据而带来的一些问题。

这里所作出的关于数据表示的选择并不是绝对的。在其他场合下，可能会选择不同的方式来表示类似的数据，这取决于应用的需要以及打算将数据派何用途。

1. 美国历史同盟所需的表

美国历史同盟的表设计相当简单：

- **总统(president)表。**此表含有描述每位总统的记录。同盟站点上的联机测验要使用这个表。
- **会员(member)表。**此表用来维护同盟每个会员的当前信息。这些信息将用来建立会员地址名录的书面和联机版本、发送会员资格更新提示等等。

(1) president表

president 表很简单，因此我们先讨论它。这个表将包含每位美国总统的一些基本信息：

- **姓名。**姓名在一个表中可用几种方式表示。如，可以用一个单一的列来存放完整的姓名，或者用分开的列来分别容纳名和姓。当然用单一的列更为简单，但是在使用上会带来一些限制，如：

- 如果先输入只有名的姓名，则不可能对姓进行排序。
 - 如果先输入只有姓的姓名，就不可能对具有名的姓名进行显示。
 - 难以对姓名进行搜索。例如，如果要搜索某个特定的姓，则必须使用一个特定的模式，并且查找与这个模式匹配的姓名。这样较之只查找姓效率更低和更慢。
- member 表将使用单独的名和姓的列以避免这些限制。

名列还存放中名（注：西方国家的姓名一般将名放在前，姓放在后，而且除了有名和姓外，有时还有中名，这是在位置上介于名和姓之间的中间名字）或首字母。这样应该不会削弱我们可能进行的任何一种排序，因为一般不可能对中名进行排序（或者甚至不会对名进行排序）。姓名即可以“Bush, George W.”格式显示，也可以“George W.Bush”格式显示。

还有一种稍显复杂一点的情形。一个总统（Jimmy Carter）在其姓名的末尾处有一个“Jr.”，这时怎样做？根据名字打印的格式，这个总统的姓名显示为“James E.Carter,Jr.”或“Carter, James E., Jr.”，“Jr.”与名和姓都没有关系，因此我们将建另外一个字段来存放姓名的后缀。这表明在试图确定怎样表示数据时，即使一个特殊的值也可能会带来问题。它也表明，为什么在将数据放入数据库前，尽量对数据值的类型进行了解是一个很好的想法。如果对数据了解不够，那么有可能在已经开始使用一个表后，不得不更改该表的结构。这不一定是个灾难，但通常应该避免。

- **出生地（城市和州）。**就像姓名一样，出生地也可以用单个列或多个列来表示。使用单列更为简单些，但正如姓名中的情形一样，独立的多个列使我们完成用单个列不方便完成的事情。例如，如果城市和州分别给出，查找各位总统出生在哪个州的记录就会更容易一些。
- **出生日期和死亡日期。**这里，唯一特殊的问题是我们不能要求都填上死亡日期，因为有的总统现在还健在。MySQL 提供了一个特殊的值 NULL，表示“无值”，可将其用在死亡日期列中以表示“仍然健在”。

(2) member 表

存储历史同盟会员清单的 member 表在每个记录都包含单个人员的基本描述信息这一点上，类似于 president 表。但是每个 member 的记录所含的列更多，member 表的各列如下：

- **姓名。**使用如 president 表一样的三个列来表示：姓、名（如果可能的话还有中名）、后缀。

■ **ID 号**。这是开始记录会员时赋给每个会员的唯一值。以前同盟未用 ID 号，但现在的记录做得更有系统性，所以最好开始使用 ID 号。（我希望您找到有利于使用 MySQL 并考虑到其他的将它用于历史同盟记录的方法。使用数字，将 member 表中的记录与其他与会员有关的表中的记录相关联要更容易一些。）

■ **截止日期**。会员必须定期更新他们的会员资格以免作废。对于某些应用，可能会用到最近更新的日期，但是近更新日期不适合于历史同盟。

会员资格可在可变的年数内（一般为一年、二年、三年或五年）更新，而最近更新的日期将不能表示下一次更新必须在何时进行。此外，历史同盟还允许有终生会员。我们可以用未来一个很长的日期来表示终生会员，但是用 NULL 似乎更为合适，因为“无值”在逻辑上对应于“永不终止”。

■ **电子邮件地址**。对于有电子邮件地址的会员，这将使他们能很容易地进行相互之间的通信。作为历史同盟秘书，这使您能电子化地发送更新通知给会员，而用不着发邮政信函。这比到邮局发送信函更容易，而且也不贵。还可以用电子邮件给会员发送他们的地址名录条目的当前内容，并要求他们在有必要时更新信息。

■ **邮政地址**。这是与没有电子邮件（或没有返回信息）的会员联络所需要的。将分别使用街道地址、城市、州和 Zip 号。街道地址列又可以用于有诸如 P.O. Box 123 而不是 123 Elm St. 的会员的信箱号。

我们假定所有同盟会员全都住在美国。当然，对于具有国际会员的机构，此假设过于简化了。如果希望处理多个国家的地址，还需要对不同国家的地址格式作一些工作。例如，这里的 Zip 号就不是一个国际标准，有的国家有省而不是州。

■ **电话号码**。与地址字段一样，这个列对于联络会员也是很有用的。

■ **特殊爱好的关键词**。假定每个会员一般都对美国历史都有兴趣，但可能有的会员对某些领域有特殊的兴趣。此列记录了这些特殊的兴趣。会员可以利用这个信息来找到其他具有类似兴趣的会员。

(3) 创建表

现在我们已经作好了创建历史同盟表的准备。我们用 CREATE TABLE 语句来完成这项工作，其一般格式如下：

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

其中 tbl_name 代表希望赋予表的名称。column_specs 给出表中列的说明，以及索引的说明（如果有的话）。索引能使查找更快；我们将在第 4 章“查询优化”中对其作进一步的介绍。

president 表的 CREATE TABLE 语句如下所示：

```
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
)
```

如果想自己键入这条语句，则调用 mysql，使 samp_db 为当前数据库：

```
% mysql samp_db
```

然后，键入如上所示的 CREATE TABLE 语句。（请记住，语句结尾要增加一个分号，否则 mysql 将不知道哪儿是语句的结尾。）

为了利用来自样例数据库分发包的预先写下的描述文件来创建 president 表，可从外壳程序运行下列命令：

```
% mysql samp_db < create_president.sql
```

不管用哪种方法调用 mysql，都应该在命令行中数据库名的前面指定连接参数（主机名、用户名或口令）。

CREATE TABLE 语句中每个列的说明由列名、类型（该列将存储的值的种类）以及一些可能的列属性组成。

president 表中所用的两种列类型为 VARCHAR 和 DATE。VARCHAR (n) 代表该列包含可变长度的字符（串）值，其最大长度为 n 个字符。可根据期望字符串能有多长来选择 n 值。state 定义为 VARCHAR(2)；即所有州名都只用其两个字符的缩写来表示。其他的字符串列则需要更长一些，以便存放更长的值。

我们使用过的其他列类型为 DATE。这种列类型表示该列存储的是日期值，这一点也不令人吃惊。而令人吃惊的是，日期的表示以年份开头。其标准格式为“YYYY-MM-DD”（例如，“1999-07-18”）。这是日期表示的 ANSI SQL 标准。

我们用于 president 表的唯一列属性为 NULL（值可以缺少）和 NOT NULL（必须填充值）。多数列是 NOT NULL 的，因为我们总要有它们的值。可有 NULL 值的两个列是 suffix（多数姓名没有后缀）和 death（有的总统仍然健在，所以没有死亡日期）。

member 表的 CREATE TABLE 语句如下所示：

```
CREATE TABLE member
(
    last_name VARCHAR(20) NOT NULL,
    first_name VARCHAR(20) NOT NULL,
    suffix VARCHAR(5) NULL,
    expiration DATE NULL DEFAULT "0000-00-00",
    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
)
```

将此语句键入 mysql 或执行下列外壳程序命令：

```
% mysql samp_db < create_member.sql
```

从列的类型来看，member 表并不很有趣：所有列中，除了一列之外，其他列都是可变长字符串。这个例外的列就是 expiration，为 DATE 型。终止日期值有一个缺省值为“0000-00-00”，这是一个非 NULL 的值，它表示未输入合法的日期值。这样做的原因是 expiration 可以是 NULL，它表示一个会员是终身会员。但是，因为此列可以为 NULL，除非另外指定一个不同的值，否则它将取缺省值“0000-00-00”。如果创建了一个新会员记录，但忘了指定终止日期，该会员将成为一个终身会员！通过采用缺省值“0000-00-00”的方法，避免了这个问题。它还向我们提供了一种手段，即可以定期地搜索这个值，以找出过去未正确输入终止日期的记录。

请注意，我们“忘了”放入会员 ID 号的列。这是专门为了以后练习使用 ALTER TABLE 语句而遗留下的。

现在让我们来验证一下 MySQL 是否确实如我们所期望的那样创建了表。在 mysql 中，发布下列查询：

```
mysql> DESCRIBE president;
```

Field	Type	Null	Key	Default	Extra
last_name	varchar(15)				
first_name	varchar(15)				
suffix	varchar(5)	YES		NULL	
city	varchar(20)				
state	char(2)				
birth	date			0000-00-00	
death	date	YES		NULL	

与 MySQL 3.23 一样，此输出还包括了显示访问权限信息的另一个列，这里没有给出，因为它使每行太长，不易显示。

这个输出结果看上去和我们所期望的非常一致，除了 state 列的信息显示它的类型为 CHAR(2)。这就有点古怪了，我们不是定义它为 VARCHAR(2) 了吗？是的，是这样定义的，但是 MySQL 已经悄悄地将此类型从 VARCHAR 换成了 CHAR。原因是为了使短字符串列的存储空间利用更为有效，这里不多讨论。如果希望详细了解，可参阅第 3 章中关于 ALTER TABLE 语句的介绍。但对这里的使用来说，两种类型没有什么差别。

如果发布一个 DESCRIBE member 查询，mysql 也会显示 member 表的类似信息。

DESCRIBE 在您忘了表中的列名、需要知道列的类型、了解列有多宽等的时候很有用。它对于了解 MySQL 存储表行中列的次序也很有用。列的这个存储次序在使用 INSERT 或 LOAD DATA 语句时非常重要，因为这些语句期望列值以缺省列的次序列出。

DESCRIBE 可以省写为 DESC，或者，如果您喜欢键入较多字符，则 DESCRIBE tbl_name 另一个等价的语句为 SHOW COLUMNS FROM tbl_name。

如果忘了表名怎么办？这时可以使用 SHOW TABLES。对于 samp_db 数据库，我们目前为止创建了两个表，其输出结果如下：

```
mysql> SHOW TABLES;
```

Tables_in_samp_db
member
president

如果您甚至连数据库名都记不住，可在命令行上调用 mysql 而不用给出数据库名，然后发布 SHOW DATABASES 查询：

```
mysql> SHOW DATABASES;
```

Database
menagerie
mysql
samp_db
test

数据库的列表在不同的服务器上是不同的，但是至少可以看到 `samp_db` 和 `mysql`；后一个数据库存放控制 MySQL 访问权限的授权表。

`DESCRIBE` 与 `SHOW` 查询具有可从外壳程序中使用的命令行等同物，如下：

```
% mysqlshow          与 SHOW DATABASES 一样列出所有数据库
% mysqlshow db_name    与 SHOW TABLES 一样列出给定数据库的表
% mysqlshow db_name tbl_name 与 DESCRIBE tbl_name 一样，列出给定表中的列
```

2. 用于学分保存方案的表

为了知道学分保存方案需要什么表，我们来看看在原来学分簿上是怎样记学分的。图 1-2 示出学分簿的一页。该页的主体是一个记录学分矩阵。还有一些对学分有意义的必要信息。学生名和 ID 号列在矩阵的一端。（为了简单好看，只列出了四个学生。）在矩阵顶端，记录了进行测验和测试的日期。图中示出 9 月 3 号、6 号、16 号和 23 号进行测验，9 月 9 号和 10 月 1 号进行测试。

为了利用数据库来记录这些信息，需要一个学分表。这个表中应该包含什么记录呢？很明显，每一行都需要有学生名、测验或测试的日期以及学分。图 1-3 示出了用这样的表表示的一些来自学分簿的学分。（日期以 MySQL 的表示格式“YYYY-MM-DD”表示。）

students		scores						
ID	name	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 学分簿样例

score 表

name	date	score
Billy	1999-09-23	15
Missy	1999-09-23	14
Johnny	1999-09-23	17
Jenny	1999-09-23	19
Billy	1999-10-01	67
Missy	1999-10-01	73
Johnny	1999-10-01	82
Jenny	1999-10-01	79

图1-3 初步的学分表设计

但是，以这种方式设置表似乎有点问题。好像少了点什么。请看图 1-3 中的记录，我们分辨不出是测验的学分还是测试的学分。如果测验和测试的学分权重不同，在确定最终的学分等级时知道学分的类型是很重要的。或许可以试着从学分的取值范围来确定学分的类型（测验的学分一般比测试的学分少），但是这样做很不方便，因为这需要进行判断，而且在数据中也不明显。

可以通过记录学分的类型来进行区分，如对学分表增加一列，此列包含“T”或“Q”以表示是“测试”或是“测验”，如图 1-4 所示。这具有使学分数据类型清晰易辨的优点。不利的地方是这个信息有点冗余。显然对具有同一给定日期的记录，学分的类型列总是取相同的值。9 月 23 日的学分总是为“Q”类型，而 10 月 1 日的学分其类型总是具有“T”类型。这样令人很不满意。如果我们以这种方式记录一组测验或测试的学分，不仅要为每个新记录输入相同的日期，而且还要一再重复地输入相同的学分类型。谁会希望一再输入冗余的信息呢？

我们可以试试另外一种表示。不在 `score` 表中记录学分类型，而是从日期上区分它们。我们可

score 表

name	date	score	type
Billy	1999-09-23	15	Q
Missy	1999-09-23	14	Q
Johnny	1999-09-23	17	Q
Jenny	1999-09-23	19	Q
Billy	1999-10-01	67	T
Missy	1999-10-01	73	T
Johnny	1999-10-01	82	T
Jenny	1999-10-01	79	T

图1-4 score 表的设计，包括学分类型

以做一个日期列表，用它来记录每个日期发生的“学分事件”（测验或测试）。然后将学分与这个事件列表中的信息结合，确定学分是测验学分还是测试学分。这只要将 score 表记录中的日期与 event 表中的日期相匹配得出事件类型即可。图 1-5 示出这个表的设计并演示了 score 表记录与 9 月 23 日这个日期相关联的工作。通过将 score 表中的记录与 event 表中记录相对应，我们知道这个学分来自测验。

score 表			event 表	
name	date	score	date	type
Billy	1999-09-23	15	1999-09-03	Q
Missy	1999-09-23	14	1999-09-06	Q
Johnny	1999-09-23	17	1999-09-09	T
Jenny	1999-09-23	19	1999-09-16	Q
Billy	1999-10-01	67	1999-09-23	Q
Missy	1999-10-01	73	1999-10-01	T
Johnny	1999-10-01	82		
Jenny	1999-10-01	79		

图1-5 score 和 event 表，按日期关联

这比根据某些猜测来推断学分类型要好得多；我们可以根据明确记录在数据库中的数据来直接得到学分类型。这也比在 score 表中记录学分类型更好，因为我们只需对每个类型记录一次。

但是，在第一次听到这种事情时（即结合使用多个表中的信息），可能会想，“嗯，这是一个好主意，但是不是要做很多工作呢？会不会使工作更复杂了？”

在某种程度上，这种想法是对的。处理两个记录表比处理一个要复杂。但是再来考察一下学分簿（见图 1-2）。不是也记录了两套东西吗？考虑下列事实：

- 在学分矩阵中用两个单元记录学分，其中每个单元都是按学生名字和日期（在矩阵的旁边和顶上）进行索引的。这代表了一组记录；与 score 表的作用相同。
- 怎样知道每个日期代表的事件类型呢？在日期上方写了字符“T”或“Q”！因此，也在矩阵顶上记录了日期和学分类型之间的关系。它代表第二组记录；与 event 表的作用相同。

换句话说，这里建议在两个表中记录信息与用学分簿记录信息所做的工作没什么不同。唯一不同的是，这两组信息在学分簿中不是那么明显地被分开。

在图 1-5 中所示的 event 表的设计中加了一个要求，那就是日期必须是唯一的，因为要用它连接 score 与 event 表的记录。换句话说，同一天不能进行两次测验，或者同一天不能进行一次测验和一次测试。否则，将会在 score 表中有两个记录并且在 event 表中也有两个记录，全都具有相同的日期，这时就不知道应如何将 score 的记录与 event 的记录进行匹配。

如果每天不多于一个学分事件，这就是一个永远不会出现的问题，可是事实并非如此简单。有时，一天中可能会有不止一个学分事件。我常听有的人说他们的数据，“那种古怪情况从不会出现。”然而，如果这种情况确实出现时，就必须重新设计表以适应这种情况引起的问题。

最好是预先考虑以后可能出现的问题，并预先准备好怎样处理他们。因此，我们假定有时可能会需要同一天记录两组学分。我们怎样处理呢？如果出现这种情况，问题并不难解决。只要对处理数据的方式作一点小的更改，就可使同一日期上有多个事件而不会引起问题：

- 1) 增加一个列到 event 表，并用它来给表中每个记录分配一个唯一的编号。实际上这就

给了每个事件一个唯一的 ID 号，因此我们称该列为 event_id 列。（如果觉得这好像是做傻事，可看一下图 1-2 中的学分簿，其中已经有这个特征了。事件 ID 正好与学分簿分数矩阵中列号相似。这个编号可能没有清晰地写在那儿并标上“事件 ID，”但是它确实在那儿。）

2) 当向 score 表中输入学分时，输入的是事件 ID 而不是日期。

这些改变的结果如图 1-6 所示。现在连接 score 和 event 表时，用的是事件 ID 而不是日期，而且不仅用 event 表来决定每个学分的类型，而且还用它来决定其日期。并且在 event 表中不再有日期必须唯一这个限制，而唯一的是事件 ID。这表示同一天可以有打测试和测验，而且能够在记录里边直接保存它们。（毫无疑问，学生们听到这个一定浑身发抖。）

不幸的是，从人的观点来看，图 1-6 中的表设计较前一个更不能令人满意。score 表也更为抽象一些，因为它包含的从直观上可以理解的列更少。而图 1-4 中此表的设计直观且容易理解，因为那个 score 表具有日期和学分类型的列。当前的 score 表如图 1-6 所示，日期和学分类型的列都没有了。这极大地去除了作为人能够很容易考虑的一切。谁希望看到其中有“事件 ID”的 score 表？如果有的话，也不代表我们大多数人。

score 表			event 表		
name	event_id	score	event_id	date	type
Billy	5	15	1	1999-09-03	Q
Missy	5	14	2	1999-09-06	Q
Johnny	5	17	3	1999-09-09	T
Jenny	5	19	4	1999-09-16	Q
Billy	6	67	5	1999-09-23	Q
Missy	6	73	6	1999-10-01	T
Johnny	6	82			
Jenny	6	79			

图1-6 score 表与 event 表，按事件 ID 关联

此时，可看到能够电子化地完成学分记录，且在赋予学分等级时不必做各种乏味的手工计算。但是，在考虑了如何实际在一个数据库中表示学分信息后，又会被怎样抽象和拆分成学分信息的表示难住了。

自然会产生一个问题：“根本不使用数据库可能会更好一些？或许 MySQL 不适合我？”正如您所猜测的那样，笔者将从否定的方面对这个问题进行回答，否则这本书就没必要再往下写了。不过，在考虑如何做一件工作时，应考虑各种情况并提问是否最好不使用数据库系统（如 MySQL）而使用一些别的东西（如电子表格等）：

■ 学分簿有行和列，而电子表格也有。这使学分簿和电子表格在概念上和外观上都非常类似。

■ 电子表格能够完成计算，可以利用一个计算字段来累计每个学生的学分。但是，要对测验和测试进行加权可能有点麻烦，但这也是可以办得到的。

另一方面，如果希望只查看某部分数据（如只查看学分或测试），进行诸如男孩与女孩的比较，或以一种灵活的方式显示合计信息等，情况又大有不同了。电子表格的功能显得要差一些，而关系数据库系统完成这些工作相当容易。

另外要考虑的一点是为了在关系数据库中进行表示而对数据进行抽象和分解，这个问题并不真的那么难以应付。只要考虑安排数据库使其不会以一种对您希望做的事无意义的方式来表示数据即可。但是，在确定了表示方式之后，就要靠数据库引擎来协调和表示数据了。您肯定不会希望将它视为一堆支离破碎的东西。

例如，在从 score 表中检索学时分，不希望看到事件 ID；但希望看到日期。这没有什么问题。数据库将会根据事件 ID 从 event 表中查找出日期。您还可能想要看看是测验的学分或测试的学分。这也不成问题。数据库将用相同的方法找出学分类型，也是利用事件 ID。请记住，这就是如像 MySQL 这样的关系数据库的优势所在，即，使一样东西与另一样东西相关联，以便从多个来源得出信息并以您实际想看到的形式提供出来。在学分保存数据的情况下，MySQL 确实利用事件 ID 将信息组合到了一起，而无需人工来完成这件事。

现在我们先来看看，如何使 MySQL 完成这种将一个东西与另一个东西相联系的工作。假定希望看到 1999年9月23号的学分，针对某个特定日期中给出的事件的分查询如下所示：

```
SELECT score.name, event.date, score.score, event.type
FROM score, event
WHERE event.date = "1999-09-23"
AND score.event_id = event.event_id
```

相当吓人，是吗？这个查询通过将 score 表的记录与 event 表的记录连接（关联）来检索学生名、日期、学分和学分的类型。其结果如下所示：

name	date	score	type
Billy	1999-09-23	15	Q
Missy	1999-09-23	14	Q
Johnny	1999-09-23	17	Q
Jenny	1999-09-23	19	Q

您肯定注意到了，它与图 1-4 中给出的表设计相同，而且不需要知道事件 ID 就可得出这个结果，只需指出感兴趣的日期并让 MySQL 查找出哪个学分记录具有该日期即可。如果您一直担心抽象和分解会使我们损失一些东西的话，看到这个世界，就不会有这种担心了。

当然，在考虑过查询后，您可能对其他别的东西产生担心。即，这个查询看上去有点长并且也有点复杂；是不是做了很多工作写出这样的东西只是为了查找某个给定日期的学分？是的，确实是这样。但是，在每次想要发布一个查询时，有几种方法可以避免键入多行的 SQL。一般情况下，一旦您决定如何执行这样一个查询并将它保存起来后，就可以按需要多次执行它。我们将在 1.5 节“与 mysql 交互的技巧”中介绍怎样完成这项工作。

在上述查询的介绍中，我们有点超前了。不过，这个查询比起我们要实际用来得出学分的查询是有点简单了。原因是，我们还要对表的设计作更多的修改。我们将采用一个唯一的学生 ID，而不在 score 表中记录学生名。（即，我们将使用来自学分簿的“ID”列的值而不是来自“Name”列的值。）然后，创建另一个称为 student 的表来存放 name 和 student_id 列（见图 1-7）。

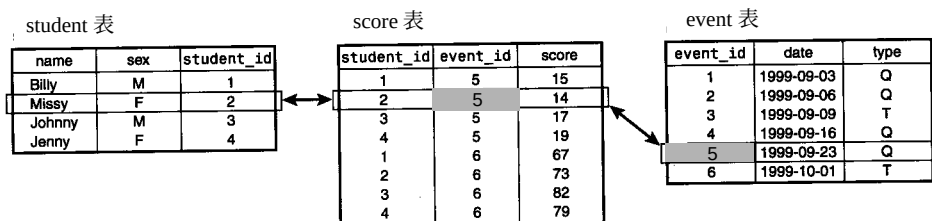


图1-7 student、score 和 event 表，按学生 ID 和事件 ID 关联

为什么要作出这种修改呢？只有一个原因，可能有两个学生有相同的名字。采用唯一的学生 ID 号可帮助区分他们的学分。（这与利用唯一的事件 ID 而不是日期来分辨出相同日期的测试或测验完全类似。）

在对表的设计作了这样的修改后，实际用来获得给定日期的学分查询变得更为复杂了一些，这个查询如下：

```
SELECT student.name, event.date, score.score, event.type
FROM event, score, student
WHERE event.date = "1999-09-23"
AND event.event_id = score.event_id
AND score.student_id = student.student_id
```

如果您不能立即清楚地读懂这个查询的意思的话，也不必担心。在进一步深入这个教程之后，就能看懂这个查询了。

将会从图1-7中注意到，在student表中增加了点学分簿中没有的东西。它包含了一个性别列。这便可以做一些简单的事情，如对班级中男孩和女孩的人数计数；也可以做一些更为复杂的事情，如比较男孩和女孩的学分。

我们已经设计完了学分保存的几乎所有的表。现在只需要另外一个表来记录出勤情况即可。这个表的内容相对较为直观，即，一个学生 ID 号和一个日期（见图1-8）。表中的每行表示特定的学生在给定的日期缺勤。在学分时段末，我们将调用 MySQL 的计数功能来汇总此表的内容，以便得出每个学生的缺勤数。

adsence 表

student_id	date
2	1999-09-02
4	1999-09-15
2	1999-09-20

图1-8 absence 表

既然现在已经知道学分保存的各个表的结构，现在可以创建它们了。

student 表的 CREATE TABLE 语句如下：

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

将上述语句键入 mysql 或执行下列外壳程序命令：

```
% mysql samp_db < create_student.sql
```

CREATE TABLE 语句创建了一个名为 student 的表，它含有三列，分别为：name、sex 和 student_id。

name 是一个可变长的字符串列，最多可存放 20 个字符。这个名字的表示比历史同盟表中所用的表示要简单，它只用了单一的列而不是分别的名和姓列。这是因为我们已经预先知道，不存在无需做额外的工作就使得在多个列上工作得更好的查询样例。

sex 表示学生是男孩还是女孩。这是一个 ENUM（枚举）列，表示只能取明确地列在说明中的值之一，这里列出的值为：“F”和“M”，分别表示女和男。在某列只具有一组有限值时，ENUM 类型非常有用。我们可以用 CHAR(1) 来代替它，但是 ENUM 更明确规定了列可以取什么值。如果对包括一个 ENUM 列的表发布一条 DESCRIBE tbl_name 语句，MySQL 将确切地显示可取的值有哪些。

顺便说一下，ENUM 列中的值不一定只是单个字符。此列还可以定义为 ENUM（‘female’，‘male’）。

student_id 为一个整数型列，它将包含唯一的 ID 号。通常，大概会从一个中心资料来源处（如学校办公室）取得学生的 ID 号，但在这里是我们自己定的。虽然 student_id 列只包含一个数，但其定义包括几个部分：

- INT 说明此列的值必须取整数（即无小数部分）。
- UNSIGNED 不允许负数。
- NOT NULL 表示此列的值必须填入。（任何学生都必须有一个 ID 号。）
- AUTO_INCREMENT 是 MySQL 中的一个特殊的属性。其作用为：如果在创建一个新的 student 表记录时遗漏了 student_id 的值（或为 NULL），MySQL 自动地生成一个大于当前此列中最大值的唯一 ID 号。在录入学生表时将用到这个特性，录入学生表时可以只给出 name 和 sex 的值，让 MySQL 自动生成 student_id 列值。
- PRIMARY KEY 表示相应列的值为快速查找进行索引，并且列中的每个值都必须是唯一的。这样可防止同一名字的 ID 出现两次，这对于学生 ID 来说是一个必须的特性。（不仅如此，而且 MySQL 还要求每个 AUTO_INCREMENT 列都具有一个惟一索引。）

如果您不理解 AUTO_INCREMENT 和 PRIMARY KEY 的含义，只要将其想像为一种为每个学生产生 ID 号的魔术方法即可。除了要求值唯一外，没有什么别的东西。

请注意：如果确实打算从学校办公室取得学生 ID 号而不是自动生成它们，则可以按相同的方法定义 student_id 列，只不过不定义 AUTO_INCREMENT 属性即可。

event 表如下定义：

```
CREATE TABLE event
(
    date DATE NOT NULL,
    type ENUM('T','Q') NOT NULL,
    event_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
)
```

将此语句键入 mysql 或执行下列外壳程序的命令：

```
% mysql samp_db < create_event.sql
```

所有列都定义为 NOT NULL，因为它们中任何一个值都不能省略。

date 列存储标准的 MySQL DATE 日期值，格式为“YYYY-MM-DD”（首先是年）。

type 代表学分类型。像 student 表中的 sex 一样，type 也是一个枚举列。所允许的值为“T”和“Q”，分别表示“测试”和“测验”。

event_id 是一个 AUTO_INCREMENT 列，类似于 student 表中的 student_id 列。采用 AUTO_INCREMENT 允许生成唯一的事件 ID 值。正如 student 表中的 student_id 列一样，与值的惟一性相比，某个特定的值并不重要。

score 表如下定义：

```
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)
)
```

将此语句键入 mysql 或执行下列外壳程序的命令：

```
% mysql samp_db < create_score.sql
```

score 为一个 INT（整型）列。即，假定学分值总是为一个整数。如果希望使学分值具有小数部分，如 58.5，应该采用浮点列类型，如 FLOAT 或 DECIMAL。

student_id 列和 event_id 列都是整型，分别表示每个学分所对应的学生和事件。通过利用它们来连接到 student 和 event 表，我们能够知道学生名和事件的日期。我们将两个列组成了 PRIMARY KEY。这保证我们不会对同一测验或测试重复一个学生的学分。而且，这样还很容易在以后更改某个学分。例如，在发现学分数录入错时，可以在利用 MySQL 的 REPLACE 语句放入一个新记录，替换掉旧的记录。不需要执行 DELETE 语句与 INSERT 语句；MySQL 自动替我们做了。

请注意，它是惟一的 event_id 和 student_id 的组合。在 score 表中，两者自身都可能不惟一。一个 event_id 值可有多个学分记录（每个学生对应一个记录），而每个 student_id 值都对应多个记录（每个测验和测试有一个记录）。

用于出勤情况的 absence 表如下定义：

```
CREATE TABLE absence
(
    student_id INT UNSIGNED NOT NULL,
    date DATE NOT NULL,
    PRIMARY KEY (student_id, date)
)
```

将此语句键入 mysql 或执行下列外壳程序的命令：

```
% mysql samp_db < create_absence.sql
```

student_id 和 date 列两者都定义为 NOT NULL，不允许省略值。应定义这两列的组合为主键，以免不当心建立了重复的记录。重要的是不要对同一天某个学生的缺旷进行重复计数。

1.4.7 增加新记录

至此，我们的数据库及其表都已经创建了，在下一节“检索信息”中，我们将看到怎样从数据库中取出数据。现在我们将一些数据放入表中。

在数据库中加入数据有几种方法。可通过发布 INSERT 语句手工将记录插入某个表中。还可以通过从某个文件读取它们来增加记录，在这个文件中，记录既可以是利用 LOAD DATA 语句或 mysqlimport 实用程序装入的原始数据值，也可以是预先写成可馈入 mysql 的 INSERT 语句的形式。

本节介绍将记录插入表的每种方法。您所应做的是演习各种方法以明了它们是如何起作用的。然后到本节结束处运行那儿给出的命令来清除表并重装它们。这样做，能够保证表中含有作者撰写下一节时所处理的相同记录，您也能得到相同的结果。

让我们开始利用 INSERT 语句来增加记录，这是一个 SQL 语句，需要为它指定希望插入数据行的表或将值按行放入的表。INSERT 语句具有几种形式：

■ 可指定所有列的值：

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

例如：

```
mysql> INSERT INTO student VALUES('Kyle','M',NULL);
mysql> INSERT INTO event VALUES("1999-9-3","Q",NULL);
```

“INTO”一词自 MySQL 3.22.5 以来是可选的。（这一点对其他形式的 INSERT 语句也成

立。) VALUES 表必须包含表中每列的值, 并且按表中列的存放次序给出。(一般, 这就是创建表时列的定义次序。如果不能肯定的话, 可使用 DESCRIBE tbl_name 来查看这个次序。)

在 MySQL 中, 可用单引号或双引号将串和日期值括起来。上面例子中的 NULL 值是用于 student 和 event 表中的 AUTO_INCREMENT 列的。(插入“错误”的值将导致下一个 student_id 或 event_id 号的自动生成。)

自 3.22.5 以来的 MySQL 版本允许通过指定多个值的列表, 利用单个的 INSERT 语句将几行插入一个表中, 如下所示:

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

例如:

```
mysql> INSERT INTO student VALUES('Abby','F',NULL),('Kyle','M',NULL);
```

这比多个 INSERT 语句的键入工作要少, 而且服务器执行的效率也更高。

- 可以给出要赋值的那个列, 然后再列出值。这对于希望建立只有几个列需要初始设置的记录是很有用的。

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

例如:

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

自 MySQL 3.22.5 以来, 这种形式的 INSERT 也允许多个值表:

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

在列的列表中未给出名称的列都将赋予缺省值。

- 自 MySQL 3.22.10 以来, 可以 col_name = value 的形式给出列和值。

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

例如:

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

在 SET 子句中未命名的行都赋予一个缺省值。

使用这种形式的 INSERT 语句不能插入多行。

将记录装到表中的另一种方法是直接从文件读取数据值。可以用 LOAD DATA 语句或用 mysqlimport 实用程序来装入记录。

LOAD DATA 语句起批量装载程序的作用, 它从一个文件中读取数据。可在 mysql 内使用它, 如下所示:

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

该语句读取位于客户机上当前目录中数据文件 member.txt 的内容, 并将其发送到服务器装入 member 表。

如果您的 MySQL 版本低于 3.22.15, 则 LOAD DATA LOCAL 不起作用, 因为那时从客户机读取数据的能力是在 LOAD DATA 上的。(没有 LOCAL 关键字, 被读取的文件必须位于服务器主机上, 并且需要大多数 MySQL 用户都不具备的服务器访问权限。)

缺省时, LOAD DATA 语句假定列值由 tab 键分隔, 而行则以换行符结束。还假定各个值是按列在表中的存放次序给出的。也有可能需要读取其他格式的文件, 或者指定不同的列次序。更详细的内容请参阅附录 D 的 LOAD DATA 的条款。

mysqlimport 实用程序起 LOAD DATA 的命令行接口的作用。从外壳程序调用

mysqlimport，它生成一个 LOAD DATA 语句：

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

mysqlimport 生成一个 LOAD DATA 语句，此语句使 member.txt 文件被装入 member 表。如果您的 MySQL 版本低于 3.22.15，这个实用程序不起作用，因为 --local 选项需要 LOAD DATA LOCAL。正如使用 mysql 一样，如果您需要指定连接参数，可在命令行上数据库名前指定它们。

mysqlimport 从数据文件名中导出表名（它将文件名第一个圆点前的所有字符作为表名）。例如，member.txt 将被装入 member 表，而 president.txt 将被装入 president 表。如果您有多个需要装入单个表的文件，应仔细地选择文件名，否则 mysqlimport 将不能使用正确的表名。对于如像 member1.txt 与 member2.txt 这样的文件名，mysqlimport 将会认为相应的表名为 member1 和 member2。不过，可以使用如 member.1.txt 和 member.2.txt 或 member.txt1 和 member.txt2 这样的文件名。

在试用过这些记录追加的方法后，应该清除各个表并重新装载它们，以便它们的内容与下一节假定的内容相同。

从外壳程序执行下列命令：

```
% mysql samp_db < insert_president.sql
% mysql samp_db < insert_member.sql
% mysql samp_db < insert_student.sql
% mysql samp_db < insert_score.sql
% mysql samp_db < insert_event.sql
% mysql samp_db < insert_absence.sql
```

每个文件都含有一个删除可能曾经插入到表中的记录的 DELETE 语句，后跟一组 INSERT 语句以初始化表的内容。如果不希望分别键入这些命令，可试一下下列语句：

```
% cat insert_*.sql | mysql samp_db
```

1.4.8 检索信息

现在各个表已经创建并装有数据了，因此让我们来看看可以对这些数据做点什么。

SELECT 语句允许以一般的方式检索和显示表中的信息。它可以显示表的整个内容：

```
SELECT * FROM president
```

或者只显示单个行中单个列的内容：

```
SELECT birth_date FROM president WHERE last_name = "Eisenhower"
```

SELECT 语句有几个子句（部件），可以根据需要用来检索感兴趣的信息。每个子句都可简单、可复杂，从而 SELECT 作为一个总的语句也繁简皆宜。但是，可以放心，本书中不会有花一个钟头来编写的长达数页的查询。（我在书中看到有很长的查询时，一般会立即跳过它们，因此我猜您也会这样。）

SELECT 语句的一般形式为：

SELECT 要选择的東西

FROM 一个或多个表

WHERE 数据必须满足的条件

记住，SQL 为一个自由格式的语言，因此在您编写 SELECT 查询时，语句的断行不必严格依照本书。

为了编写 SELECT 语句，只需指定需要检索什么，然后再选择某些子句即可。刚才给出的子句“FROM”、“WHERE”是最常用的，还有一些其他的子句，如 GROUP BY、ORDER BY 和LIMIT 等。

FROM 子句一般都要给出，但是如果不从表中选择数据，也可不给出。例如，下列查询只显示某些可以直接计算而不必引用任何表的表达式的值，因此不需要用 FROM 子句：

```
mysql> SELECT 2+2, "Hello, world", VERSION();
+-----+-----+-----+
| 2+2 | Hello, world | VERSION() |
+-----+-----+-----+
| 4 | Hello, world | 3.23.0-alpha-log |
+-----+-----+-----+
```

在确实使用一个 FROM 子句指定了要从其中检索数据的表时，SELECT 语句的最“普通”的格式是检索所有内容。用“*”来表示“所有列”。下面的查询将从 student 表中检索所有行并显示：

```
mysql> SELECT * FROM student;
+-----+-----+-----+
| name | sex | student_id |
+-----+-----+-----+
| Joseph | M | 1 |
| Kyle | M | 2 |
| Abby | F | 3 |
| Nathan | M | 4 |
...
```

各列按它们 MySQL 在表中存放的次序出现。该次序与发布 DESCRIBE student 语句时显示的列次序相同。（例子末尾的“...”表示此查询返回的输出行比这里显示的还要多。）

可明确地命名希望得到的一列或多列。如果只选择学生名，发布下列语句：

```
mysql> SELECT name FROM student;
+-----+
| name |
+-----+
| Joseph |
| Kyle |
| Abby |
| Nathan |
...
```

如果名字不止一列，可用逗号分隔它们。下列的语句与 SELECT * FROM student 等价，只是明确地指出了每一列：

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

可按任意次序给出列：

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

如果有必要，同一列甚至也可以给出多次，虽然这样做一般是没有意义的。

列名在 MySQL 中不区分大小写的。下面的查询是等同的：

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

数据库和表名有可能区分大小写的；这有取决于服务器主机上使用的文件系统。在 UNIX 上运行的服务器对数据库名和表名是区分大小写的，因为 UNIX 的文件名是区分大小写的。Windows 的文件名不区分大小写，因此运行在 Windows 上的服务器对数据库名和表名不区分大小写。

MySQL 允许您一次从多个表中选择列。我们将这个内容留到“从多个表中检索信息”小节去介绍。

1. 指定检索条件

为了限制 SELECT 语句检索出来的记录集，可使用 WHERE 子句，它给出选择行的条件。可通过查找满足各种条件的列值来选择行。

可查找数字值：

```
mysql> SELECT * FROM score WHERE score > 95;
```

student_id	event_id	score
4	3	96
5	3	97
7	3	97
13	6	96
14	6	99
22	6	97
24	6	97
26	6	96

也可以查找串值。（注意，一般串的比较是不区分大小写的。）

```
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.

可以查找日期值：

```
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 |
+-----+

```

可搜索组合值：

```
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 子句中的表达式可使用表 1-1 中的算术运算符、表 1-2 的比较运算符和表 1-3 的逻辑运算符。还可以使用圆括号将一个表达式分成几个部分。可使用常量、表列和函数来完成运算。在本教程的查询中，我们有时使用几个 MySQL 函数，但是 MySQL 的函数远不止这里给出的这些。请参阅附录 C，那里给出了所有 MySQL 函数的清单。

表1-1 算术运算符

运 算 符	说 明	运 算 符	说 明
+	加	*	乘
-	减	/	除

表1-2 比较运算符

运 算 符	说 明	运 算 符	说 明
<	小于	!= 或 <>	不等于
<=	小于或等于	>=	大于或等于
=	等于	>	大于

在用表达式表示一个需要逻辑运算的查询时，要注意别混淆逻辑与运算符与我们平常使用的“与”的含义。假如希望查找“出生在 Virginia 的总统与出生在 Maryland 的总统”。应该注意怎样表示“与”的关系，能写成如下查询吗？

```
SELECT last_name, first_name, state FROM president
WHERE state="VA" AND state="MA"
```

错了，因为这个查询的意思是“选择既出生在 Virginia 又出生在 Maryland 的总统”，不可能有同时出生在两个地点的总统，因此这个查询无意义。在英语中，可以用“and”表示这种选择，但在 SQL 中，应该用 OR 来连接两个条件，如下所示：

```
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    |
+-----+

```

表1-3 逻辑运算符

运 算 符	说 明
AND	逻辑与
OR	逻辑或
NOT	逻辑非

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 W.	MA

这有时是可以觉察到的，不仅仅是在编写自己的查询时可以觉察到，而且在为他人编写查询时也可以知道。最好是在他人描述想要检索什么时仔细听，但不一定使用相同的逻辑运算符将他人的描述转录成 SQL 语句。对刚才所举的例子，正确的英语等价描述为“选择出生在 Virginia 或者出生在 Maryland 的总统。”

2. NULL 值

NULL 值是特殊的；因为它代表“无值”。不可能以评估两个已知值的相同方式来将它与已知值进行评估。如果试图与通常的算术比较运算符一道使用 NULL，其结果是未定义的：

```
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，甚至不能将 NULL 与它自身比较，因为不可能知道比较两个未知值的结果：

```
mysql> SELECT NULL = NULL, NULL != NULL;
```

NULL = NULL	NULL != NULL
NULL	NULL

为了进行 NULL 值的搜索，必须采用特殊的语法。不能用 = 或 != 来测试等于 NULL 或不等于 NULL，取而代之的是使用 IS NULL 或 IS NOT NULL 来测试。

例如，因为我们将健在总统的死亡日期表示为 NULL，那么可按如下语句查找健在的总统：

```
mysql> SELECT last_name, first_name FROM president WHERE death IS NULL;
```

last_name	first_name
Ford	Gerald R
Carter	James E.
Reagan	Ronald W.
Bush	George W.
Clinton	William J.

为了找到具有后缀部分的姓名，可利用 IS NOT NULL：

```
mysql> SELECT last_name, first_name, suffix
-> FROM president WHERE suffix IS NOT NULL;
```

last_name	first_name	suffix
Carter	James E.	Jr.

MySQL3.23 及以后的版本具有一个特殊的 MySQL 专有的比较运算符“<=>”，即使是 NULL 与 NULL 的比较，它也是可行的。用这个比较运算符，可将前面的两个查询重写为：

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

```
+-----+-----+
| last_name | first_name |
+-----+-----+
| Ford      | Gerald R   |
| Carter    | James E.   |
| Reagan    | Ronald W.  |
| Bush      | George W.  |
| Clinton   | William J. |
+-----+-----+
```

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

```
+-----+-----+-----+
| last_name | first_name | suffix |
+-----+-----+-----+
| Carter    | James E.   | Jr.    |
+-----+-----+-----+
```

3. 对查询结果进行排序

有时我们注意到，在一个表装入初始数据后，对其发布一条 `SELECT * FROM tbl_name` 查询，检索出的行与这些行被插入的顺序是相同的。但不要认为这种情况是有规律的。如果在初始装入表后进行了行的删除和插入，就会发现服务器返回表的行次序被改变了。（删除记录在表中留下了未使用的“空位”，MySQL 在以后插入新记录时将会试图对其填补。）

缺省时，如果选择了行，服务器对返回行的次序不作任何保证。为了对行进行排序，可使用 `ORDER BY` 子句：

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

...

在 `ORDER BY` 子句中，可在列名之后利用 `ASC` 或 `DESC` 关键字指定排序是按该列值的升序或降序进行的。例如，为了按倒序（降序）名排列总统名，可如下使用 `DESC`：

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

...

如果在 `ORDER BY` 子句中，对某个列名既不指定 `ASC` 又不指定 `DESC`，则缺省的次序为升序。

在对可能包含 `NULL` 值的列进行排序时，如果是升序排序，`NULL` 值出现在最前面，如果是按降序排序，`NULL` 值出现在最后。

查询结果可在多个列上进行排序，而每个列的升序或降序可以互相独立。下面的查询从 president 表中检索行，并按出生的州降序、在每个州中再按姓氏的升序对检索结果进行排序：

```
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

4. 限制查询结果

如果一个查询返回许多行，但您只想看其中的几行，则可以利用 LIMIT 子句，特别是与 ORDER BY 子句结合时更是如此。MySQL 允许限制一个查询的输出为前 n 行。下面的查询选择了 5 位出生日期最早的总统：

```
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

如果利用 ORDER BY birth DESC 按降序排序，将得到 5 位最晚出生的总统。

LIMIT 也可以从查询结果中取出中间部分。为了做到这一点，必须指定两个值。第一个值为结果中希望看到的第一个记录（第一个结果记录的编号为 0 而不是 1）。第二个值为希望看到的记录个数。下面的查询类似于前面那个查询，但只显示从第 11 行开始的 5 个记录：

```
mysql> SELECT last_name, first_name, birth FROM president
-> ORDER BY birth LIMIT 10, 5;
```

last_name	first_name	birth
Tyler	John	1790-03-29
Buchanan	James	1791-04-23
Polk	James K.	1795-11-02
Fillmore	Millard	1800-01-07
Pierce	Franklin	1804-11-23

自 MySQL 3.23.2 以来，可按照一个公式来排序查询结果。例如，利用 ORDER BY RAND() 与 LIMIT 结合，从 president 表中随机抽取一个记录：

```
mysql> SELECT last_name, first_name FROM president
-> ORDER BY RAND() LIMIT 1;
```

```

+-----+-----+
| last_name | first_name |
+-----+-----+
| McKinley  | William    |
+-----+-----+

```

5. 计算并命名输出的列值

前面的多数查询通过从表中检索值已经产生了输出结果。MySQL 还允许作为一个公式的结果来计算输出列的值。表达式可以简单也可以复杂。下面的查询求一个简单表达式的值（常量）以及一个涉及几个算术运算符和两个函数调用的较复杂的表达式的值：

```

mysql> SELECT 17, FORMAT(SQRT(3*3+4*4),0);
+-----+-----+
| 17 | FORMAT(SQRT(3*3+4*4),0) |
+-----+-----+
| 17 | 5                        |
+-----+-----+

```

表达式也可以引用表列：

```

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        |
+-----+-----+
...

```

此查询把名和姓连接起来，中间间隔一个空格，将总统名形成一个单一字符串，而且将出生城市和州连接在一起，中间隔一个逗号，形成出生地。

在利用表达式来计算列值时，此表达式被用作列标题。如果表达式很长（如前面的一些查询样例中那样），那么可能会出现一个很宽的列。为了处理这种情况，此列可利用 AS name 结构来重新命名标题。这样的名称为列别名。用这种方法可使上面的输出更有意义，如下所示：

```

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        |
+-----+-----+
...

```

如果列的别名包含空格，需要用双引号括起来。

```

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      |
...
```

6. 使用日期

在 MySQL 中使用日期时要记住的是，在表示日期时首先给出年份。1999 年 7 月 27 日表示为“1999-07-27”，而不是像通常那样表示为“07-27-1999”或“27-07-1999”。

MySQL 提供了几种对日期进行处理的方法。可以对日期进行的一些运算如下：

- 按日期排序。（这点我们已经看到几次了。）
- 查找特定的日期或日期范围。
- 提取日期值的组成部分，如年、月或日。
- 计算日期的差。
- 日期增加或减去一个间隔得出另一日期。

下面给出一些日期运算的例子。

为了查找特定的日期，可使用精确的日期值或与其他日期值进行比较，将一个 DATE 列与有关的日期值进行比较：

```
mysql> SELECT * FROM event WHERE date = "1999-10-01";
+-----+-----+-----+
| date      | type | event_id |
+-----+-----+-----+
| 1999-10-01 | 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 |
+-----+-----+-----+
```

为了测试或检索日期的成分，可使用诸如 YEAR()、MONTH() 或 DAYOFMONTH() 这样的函数。例如，可通过查找月份值为 3 的日期，找出与笔者出生在相同月份（三月）的总统。

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

此查询也可以按月的名称写出：

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

为了更详细，详细到天，可组合测试 MONTH() 和 DAYOFMONTH() 以找出在笔者的生日出生的总统：

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

这是一种可用来生成类似报纸上娱乐部分所刊登的那种“这些人今天过生日”清单的查询。但是，不必按前面的查询那样插入一个特殊的日期。为了查找每年的今天出生的总统，只要将他们的生日与 CURRENT_DATE 进行比较即可：

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

可从一个日期减去另一个日期。这样可以知道日期期间的间隔，这对于确定年龄是非常有用的。例如，为了确定哪位总统活得最长，可将其逝世日期减去出生日期。为此，可利用函数 TO_DAYS() 将出生日期和逝世日期转换为天数，求出差，然后除以 365 得出大概的年龄：

```
mysql> SELECT last_name, first_name, birth, death,
-> FLOOR((TO_DAYS(death) - TO_DAYS(birth))/365) AS age
-> FROM president WHERE death IS NOT NULL
-> ORDER BY age DESC LIMIT 5;
+-----+-----+-----+-----+-----+
| last_name | first_name | birth      | death      | age |
+-----+-----+-----+-----+-----+
| 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  |
| Madison  | James      | 1751-03-16 | 1836-06-28 | 85  |
| Jefferson | Thomas     | 1743-04-13 | 1826-07-04 | 83  |
+-----+-----+-----+-----+-----+
```

此查询中所用的 FLOOR() 函数截掉了年龄的小数部分，得到一个整数。

得出日期之差，还可以确定相对于某个特定日期有多长时间。这样可以告诉历史同盟的会员，他们还有多久就应该更新自己的会员资格了。计算他们的截止日期和当前日期之差，如果小于某个阈值，则不久就需要更新了。下面的查询是查找需要在 60 天内更新的会员：

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

自 MySQL 3.22 以来，可使用 DATE_ADD() 或 DATE_SUB() 从一个日期计算另一个日期。这些函数取一个日期及时间间隔并产生一个新日期。例如：

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

本节中前面给出的一个查询选择 70 年代逝世的总统，它对选择范围的端点使用直接的日期值。该查询可以利用一个字符串日期和一个由开始日期和时间间隔计算出的结束日期来重写：

```
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

会员更新查询可根据 DATE_ADD() 写出如下：

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

本章前面给出了一个查询如下，确定不久要来检查但还没来诊所的牙科病人：

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

现在回过头来看，读者会更清楚这个查询的含义了。

7. 模式匹配

MySQL 允许查找与某个模式相配的值。这样，可以选择记录而不用提供精确的值。为了进行模式匹配运算，可使用特殊的运算符（LIKE 和 NOT LIKE），并且指定一个包含通配符的串。字符“_”匹配任意单个字符，而“%”匹配任意字符序列（包括空序列）。使用 LIKE 或 NOT LIKE 的模式匹配都是不区分大小写的。

下列模式匹配以“W”或“w”开始的姓：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE "W%";
```

last_name	first_name
Washington	George
Wilson	Woodrow

下列模式匹配是错误的：

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

此查询给出了一个常见的错误，它对一个算术比较运算符使用了模式。这种比较成功的惟一可能是相应的列确实包含串“W%”或“w%”。

下列模式匹配任意位置包含“W”或“w”的姓：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE "%W%";
```

last_name	first_name
Washington	George
Wilson	Woodrow


```
| Eisenhower | Dwight D. |
+-----+
```

下列模式匹配只含有四个字符的姓：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE "____";
```

MySQL 还提供基于扩展正规表达式的模式匹配。正规表达式在附录 C 的 REGEXP 运算符的介绍中描述。

8. 生成汇总

MySQL 所能做的最有用的事情是浓缩大量的原始数据行并对其进行汇总。当学会了利用 MySQL 来生成汇总时，它就变成了用户强有力的好帮手了，因为手工进行汇总是一项冗长的、费时的、易出错的工作。

汇总的一种简单的形式是确定在一组值中哪些值是唯一值。利用 DISTINCT 关键字来删除结果中的重复行。例如，总统出生的各个州可按如下找出：

```
mysql> SELECT DISTINCT state FROM president ORDER BY state;
```

```
+-----+
| state |
+-----+
| AK    |
| CA    |
| GA    |
| IA    |
| IL    |
| KY    |
| MA    |
| MO    |
| NC    |
| NE    |
| NH    |
| NJ    |
| NY    |
| OH    |
| PA    |
| SC    |
| TX    |
| VA    |
| VT    |
+-----+
```

其他的汇总形式涉及计数，可利用 COUNT() 函数。如果使用 COUNT (*), 它将给出查询所选择的行数。如果一个查询无 WHERE 子句，COUNT(*) 将给出表中的行数。

下列查询给出共有多少人当过美国总统：

```
mysql> SELECT COUNT(*) FROM president;
```

```
+-----+
| COUNT(*) |
+-----+
|         41 |
+-----+
```

如果查询有 WHERE 子句，COUNT(*) 将给出此子句选择多少行。下面的查询给出目前为止对班级进行了多少次测试：

```
mysql> SELECT COUNT(*) FROM event WHERE type = 'Q';
```

```
+-----+
| COUNT(*) |
```

```
+-----+
|      4 |
+-----+
```

COUNT(*) 对选中的行进行计数。而 COUNT(col_name) 只对非 NULL 值进行计数。下面的查询说明了这些差异：

```
mysql> SELECT COUNT(*),COUNT(suffix),COUNT(death) FROM president;
+-----+-----+-----+
| COUNT(*) | COUNT(suffix) | COUNT(death) |
+-----+-----+-----+
|      41 |             1 |           36 |
+-----+-----+-----+
```

这表示，总共有 41 位总统，他们中只有一个具有名字后缀，并且大多数总统都已去世。

自 MySQL 3.23.2 以来，可以将 COUNT() 与 DISTINCT 组合对选择结果集中不同的值进行计数。例如，为了对总统出生的不同州进行计数，可执行下列查询：

```
mysql> SELECT COUNT(DISTINCT state) FROM president;
+-----+
| COUNT(DISTINCT state) |
+-----+
|             19 |
+-----+
```

可以根据汇总列中单独的值对计数值进行分解。例如，您可能根据下列的查询结果知道班级中所有学生的人数：

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

但是，有多少是男孩？有多少是女孩？分别得出男孩、女孩的一种方法是分别对每种性别进行计数：

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

虽然这个方法可行，但是它很繁琐而且并不真正适合于可能有许多不同的值的列。考虑一下怎样以这种方式确定每个州出生的总统人数。您不得不找出有哪些州，从而不能省略 (SELECT DISTINCT state FROM president)，然后对每个州执行一个 SELECT COUNT(*) 查询。很显然，有些事是可以简化的。

所幸 MySQL 可以利用单个查询对一个列中不同的值进行计数。因此，针对学生表可以按下得出男孩和女孩的人数：

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

M	16
F	15

用同样形式的查询得出每个州出生的总统有多少：

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

state	COUNT(*)
AK	1
CA	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

如果以这种方法对值计数，GROUP BY 子句是必须的；它告诉 MySQL 在对值计数之前怎样进行聚集。如果将其省去，则要出错。

COUNT(*) 与 GROUP BY 一起用来对值进行计数比分别对每个不同的列值进行计数有更多的优点，这些优点是：

- 不必事先知道要汇总的列中有什么值。
- 不用编写多个查询，只需编写单个查询即可。
- 用单一查询就可以得出所有结果，因此可以对结果进行排序。

前两个优点对于更方便地表示查询很重要。第三个优点也较为重要，因为它提供了显示结果的灵活性。在使用 GROUP BY 子句时，其结果是在要分组的列上进行排序的，但是可以使用 ORDER BY 来按不同的次序进行排序。例如，如果想得到各州产生的总统人数，并按产生人数最多的州优先排出，可以如下使用 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
AK	1

如果希望进行排序的列是从计算得出的，则可以给该列一个别名，并在 ORDER BY 子句中引用这个别名。前面的查询说明了这一点；COUNT(*) 列的别名为 count。引用这样的列的另一种方法是引用它在输出结果中的位置。前面的查询可编写如下：

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

我不认为按位置引用列易读。如果增加、删除或重新排序输出列，必须注意检查 ORDER BY 子句，并且如果列号改变后还得记住它。别名就不存在这种问题。

如果想与计算出来的列一道使用 GROUP BY，正如 ORDER BY 一样，应该利用别名或列位置来引用它。下面的查询确定在一年的每个月中出生的总统人数：

```
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	3
8	August	4
9	September	1
10	October	6
11	November	5
12	December	3

利用列位置，此查询可编写如下：

```
SELECT MONTH(birth), MONTHNAME(birth), COUNT(*)
FROM president GROUP BY 2 ORDER BY 1;
```

例如，COUNT() 可与 ORDER BY 和 LIMIT 组合来查找 president 表中 4 个产生总统最多的州：

```
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

如果不想用 LIMIT 子句来限制查询输出，而是利用查找特定的 COUNT() 值来达到这个目的，可使用 HAVING 子句。下面的查询给出了产生两个以上总统的州：

```
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

从更为普遍的意义上说，这是一种在要查找的列中重复值时执行的查询类型。

HAVING 类似于 WHERE，但它是在查询结果已经选出后才应用的，用来缩减服务器实际送到客户机的结果。

除了 COUNT() 外还有许多汇总函数。MIN()、MAX()、SUM() 和 AVG() 函数在确定列的最大、最小、总数和平均值时都非常有用，甚至可以同时使用它们。下面的查询得出给定的测试和测验的各种数字特性。它还给出有多少学分参与了每个值的计算（有的学生可能缺旷或未计入）。

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

event_id	minimum	maximum	range	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

当然，如果您知道这些信息是来自测验的还是测试的，则它们就会更有意义。但是，为了产生那样的信息，还需要参考 event 表；我们将在下一节“从多个表中检索信息”讨论这个查询。

汇总信息是很有意思的，因为它们是那么有用，但不太好控制，容易走样。请看下列查询：

```
mysql> SELECT
-> state AS State,
-> AVG((TO_DAYS(death)-TO_DAYS(birth))/365) AS Age
-> FROM president WHERE death IS NOT NULL
-> GROUP BY state ORDER BY Age;
```

State	Age
KY	56.208219
VT	58.852055
NC	60.141096
OH	62.866145
NH	64.917808
NY	69.342466
NJ	71.315068
TX	71.476712
MA	72.642009
VA	72.822945
PA	77.158904
SC	78.284932
CA	81.336986
MO	88.693151
IA	90.254795

此查询选择已经去世的总统，按出生地对他们进行分组，并计算出他们逝世时的年龄，计算出平均年龄（每个州的），然后按平均年龄进行排序。换句话说，此查询按所出生地确定已故总统的平均寿命。

但这说明了什么呢？它仅仅说明您可写该查询，当然并不说明此查询是否值得写。并不是用一个数据库可以做的所有事情都同样有意义；但是，人们有时在发现可以利用自己的数据库进行查询时感到很开心。这可能说明关于转播运动会的不断增加的深奥的（空洞的）统计数据在过去几年里正在不断增多的原因。运动统计者可以使用他们的数据库来计算出某个队的历史纪录，而这些数字你可能感兴趣，也可能毫无兴致。

9. 从多个表中检索信息

到目前为止，我们所编写的查询都是从单个表中得到数据的。现在，我们将进行一件更为有趣的工作。以前笔者曾经提到过，关系 DBMS 的强大功能在于它能够将一样东西与另一样东西相关联，因为这样使得能够结合多个表中的信息来解答单个表不能解答的问题。本节介绍怎样编写这种查询。

在从多个表中选择信息时，需要执行一种称为连接（join）的操作。这是因为需要将一个表中的信息与其他表中的信息相连接来得出查询结果。即通过协调各表中的值来完成这项工作。

我们来研究一个例子。在前面的“学分保存方案”小节中，给出了一个检索特定日期的测验或测试学分的查询，但没有解释。现在可以进行解释了。这个查询实际涉及到三种连接方法，因此我们分两步进行研究。

第一步，我们构造一个对特定日期的学分进行选择查询，如下所示：

```
mysql> SELECT student_id, date, score, type
-> FROM event, score
-> WHERE date = "1999-09-23"
-> AND event.event_id = score.event_id;
```

student_id	date	score	type
1	1999-09-23	15	Q
2	1999-09-23	12	Q
3	1999-09-23	11	Q


```

5 | 1999-09-23 | 13 | Q |
6 | 1999-09-23 | 18 | Q |
...

```

此查询找出具有给定日期的记录，然后利用该记录中的事件 ID 查找具有相同事件 ID 的学分。对于每个匹配的事件记录和学分记录组合，显示学生 ID、学分、日期和事件类型。

此查询在两个重要方面不同于我们曾经编写过的其他查询。它们是：

■ FROM 子句给出了不止一个表名，因为我们要检索的数据来自不止一个表：

```
FROM event, score
```

■ WHERE 子句说明 event 和 score 表是由每个表中的 event_id 值的匹配连接起来的：

```
WHERE ... event.event_id = score.event_id
```

请注意，我们是怎样利用 tbl_name.col_name 语法引用列，以便 MySQL 知道引用的是哪些表的列。（event_id 出现在两个表中，如果不用表名来限定它的话将会出现混淆。）

此查询中的其他列（date、score、type）可单独使用而不用表名限定符，因为它们在表中只出现一次，从而不会出现含混。但是，一般在连接中我们对每个列都进行限定以便清晰地表示出每个列是属于哪个表。在完全限定的形式下，查询如下：

```

SELECT score.student_id, event.date, score.score, event.type
FROM event, score
WHERE event.date = "1999-09-23"
AND event.event_id = score.event_id

```

从现在起，我们将使用完全限定的形式。

第二步，我们利用 student 表完成查询以便显示学生名。（第一步中查询的输出给出了 student_id 字段，但是名字更有意义。）名字显示是利用 score 表和 student 表两者都具有 student_id 列，使它们中的记录可被连接这个事实来完成的。最终的查询如下：

```

mysql> SELECT student.name, event.date, score.score, event.type
-> FROM event, score, student
-> WHERE event.date = "1999-09-23"
-> AND event.event_id = score.event_id
-> AND score.student_id = student.student_id;

```

```

+-----+-----+-----+-----+
| name | date | score | type |
+-----+-----+-----+-----+
| Megan | 1999-09-23 | 15 | Q |
| Joseph | 1999-09-23 | 12 | Q |
| Kyle | 1999-09-23 | 11 | Q |
| Abby | 1999-09-23 | 13 | Q |
| Nathan | 1999-09-23 | 18 | Q |
...

```

此查询与前一个查询的差别在于：

■ student 表被增加到了 FROM 子句中，因为除了 event 表和 score 表外还用到了它。

■ student_id 列现在不明确了（因为现在有两个引用到的表都含有此列），因此必须限定为 score.student_id 或 student.student_id 以表明使用的是哪个表。

■ WHERE 子句有一个附加项，它说明根据学生 ID 将 score 表记录与 student 表记录进行匹配。

■ 此查询是显示学生名而不是学生 ID。（当然，如果愿意的话，可以两者都显示。）

利用此查询，可以加入任意日期，得到该日期的学分，用学生名和学分类型完善查询结果。不一定要了解关于学生 ID 或事件 ID 的情况。MySQL 小心地得出相关的 ID 值并利用它

们自动地使各表的行相配。

学分保存方案涉及的另一项工作是汇总学生的缺勤情况。缺勤情况是按学生 ID 和日期在 absence 表中记录的。为得到学生名（而不仅仅是 ID），我们需要根据 student_id 的值将 absence 表连接到 student 表。下面的查询给出了学生的 ID 号和名字以及缺勤计数：

```
mysql> SELECT student.student_id, student.name,
-> COUNT(absence.date) as absences
-> FROM student, absence
-> WHERE 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：虽然我们在 GROUP BY 子句中应用了一个限定符，但对于这个查询来说不是必须的。因为 GROUP BY 子句只引用选择表中（此查询的前两行）的列。在该处只有一个名为 student_id 的列，因此 MySQL 知道应该用哪个列。这个规则对 ORDER BY 子句也成立。

如果我们希望只了解哪些学生缺过勤，则此查询所产生的输出也是有用的。但是，如果我们将此清单交给学校办公室，他们可能会说，“其他的学生呢？我们需要每个学生的情况。”这是一个稍微有点不同的问题。它表示需要知道学生的缺勤数，即使没有缺勤的学生也需要知道。因为问题的不同，查询也应该不同。

为了解决上述问题，使用 LEFT JOIN 而不涉及 WHERE 子句中的学生 ID。LEFT JOIN 要求 MySQL 对从连接首先给出的表中选择每行生成一个输出行（即 LEFT JOIN 关键字左边给出的表）。由于首先给出 student 表，我们得到了每个学生的输出结果，即使是那些在 absence 表中未给出的学生也都包括在输出中。此查询如下：

```
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

...

前面，在“生成汇总”一节中，我们执行了一个查询，它生成 score 表中数据的数值特征。该查询的输出列出了事件 ID，但不包括学分日期或类型，因为我们不知道怎样将 score 表连接到 event 表以得到学分的日期和类型。现在可以做到了。下面的查询类似于早先的那个，但是它给出了学分的日期和类型而不只是简单的数字事件 ID：

```
mysql> SELECT
-> event.date,event.type,
-> MIN(score.score) AS minimum,
-> MAX(score.score) AS maximum,
-> MAX(score.score)-MIN(score.score)+1 AS range,
-> SUM(score.score) AS total,
-> AVG(score.score) AS average,
-> COUNT(score.score) AS count
-> FROM score, event
-> WHERE score.event_id = event.event_id
-> GROUP BY event.date;
```

date	type	minimum	maximum	range	total	average	count
1999-09-03	Q	9	20	12	439	15.1379	29
1999-09-06	Q	8	19	12	425	14.1667	30
1999-09-09	T	60	97	38	2425	78.2258	31
1999-09-16	Q	7	20	14	379	14.0370	27
1999-09-23	Q	8	20	13	383	14.1852	27
1999-10-01	T	62	100	39	2325	80.1724	29

可利用诸如 COUNT() 和 AVG() 这样的函数生成多个列上的汇总，即使这些列来自不同的表也是如此。下面的查询确定学分数，以及事件日期与学生性别的每种组合的平均学分。

```
mysql> SELECT event.date, student.sex,
-> COUNT(score) AS count, AVG(score) As average
-> FROM event, score, student
-> WHERE event.event_id = score.event_id
-> AND score.student_id = student.student_id
-> GROUP BY event.date, student.sex;
```

date	sex	count	average
1999-09-03	F	14	14.6429
1999-09-03	M	15	15.6000
1999-09-06	F	14	14.7143
1999-09-06	M	16	13.6875
1999-09-09	F	15	77.4000
1999-09-09	M	16	79.0000
1999-09-16	F	13	15.3077
1999-09-16	M	14	12.8571
1999-09-23	F	12	14.0833
1999-09-23	M	15	14.2667
1999-10-01	F	14	77.7857
1999-10-01	M	15	82.4000

我们可以使用一个类似的查询来完成学分保存方案的一个任务，即在学期末计算每个学生的总学分。相应的查询如下：

```
SELECT student.student_id, student.name,
SUM(score.score) AS total, COUNT(score.score) AS n
FROM event, score, student
WHERE event.event_id = score.event_id
AND score.student_id = student.student_id
GROUP BY score.student_id
ORDER BY total
```

不一定要连接必须用两个不同的表来完成。这似乎有点奇怪，但是确实可以将一个表连接到其自身。例如，可通过针对每个总统的出生地查看其他各个总统的出生地，确定几个

总统是否出生在相同城市。此查询如下：

```
mysql> SELECT p1.last_name, p1.first_name, p1.city, p1.state
-> FROM president AS p1, president AS p2
-> WHERE p1.city = p2.city AND p1.state = p2.state
-> AND (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

此查询有两个技巧性的东西：

■ 我们需要使用同一表的两个实例，因此建立了表的别名（p1、p2），并利用它们无歧义地引用表列。

■ 每个总统的记录与自身相匹配，但是我们不希望在输出中看到同一总统出再现两次。

WHERE 子句的第二行保证比较的记录为不同总统的记录，使记录不与自身匹配。

可以编写一个查找出生在同一天总统的类似查询。出生日期不能直接比较，因为那样会错过出生在不同年份的总统。我们用 MONTH() 和 DAYOFMONTH() 来比较出生日期的月和日，相应的查询如下：

```
mysql> SELECT p1.last_name, p1.first_name, p1.birthday
-> FROM president AS p1, president AS p2
-> WHERE MONTH(p1.birthday) = MONTH(p2.birthday)
-> AND DAYOFMONTH(p1.birthday) = DAYOFMONTH(p2.birthday)
-> 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

利用 DAYOFYEAR() 而不是 MONTH() 和 DAYOFMONTH() 将得出一个更为简单的查询，但是在比较闰年日期与非闰年日期时将会得出不正确的结果。

迄今所执行的连接结合了来自那些在某种意义上具有逻辑关系的表中的信息，但是只有您知道该关系无意义。MySQL 并不知道（或不关心）所连接的表相互之间是否相关。例如，可将 event 表连接到 president 表以找出在某个总统生日那天是否进行了测验或测试，此查询如下：

```
mysql> SELECT president.last_name, president.first_name,
-> president.birthday, event.type
-> FROM president, event
-> WHERE MONTH(president.birthday) = MONTH(event.date)
-> AND DAYOFMONTH(president.birthday) = DAYOFMONTH(event.date);
```

last_name	first_name	birth	type
Carter	James E.	1924-10-01	T

它产生了您所想要的东西。但说明了什么呢？这说明 MySQL 将愉快地制造出结果，至于这些结果是否有意义它不管。这是因为您使用的是计算机，所以它不能自动地判断查询的

结果有用或无用。无论如何，我们都必须为自己所做的事负责。

1.4.9 删除或更新现有记录

有时，希望除去某些记录或更改它们的内容。DELETE 和 UPDATE 语句令我们能做到这一点。

DELETE 语句有如下格式：

```
DELETE FROM tbl_name WHERE 要删除的记录
```

WHERE 子句指定哪些记录应该删除。它是可选的，但是如果不选的话，将会删除所有的记录。这意味着最简单的 DELETE 语句也是最危险的。

```
DELETE FROM tbl_name
```

这个查询将清除表中的所有内容。一定要当心！

为了删除特定的记录，可用 WHERE 子句来选择所要删除的记录。这类似于 SELECT 语句中的 WHERE 子句。例如，为了删除 president 表中所有出生在 Ohio 的总统记录，可用下列查询：

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

DELETE 语句中的 WHERE 子句的一个限制是只能够引用要删除记录的表中的列。

在发布 DELETE 语句以前，最好用 SELECT 语句测试一下相应的 WHERE 子句以确保实际删除的记录就是确实想要删除的记录（而且只删除这些记录）。假如想要删除 Teddy Roosevelt 的记录。下面的查询能完成这项工作吗？

```
DELETE FROM president WHERE last_name="Roosevelt"
```

是的，感觉上它能删除您头脑中打算删除的记录。但是，错了，实际上它也能删除 Franklin Roosevelt 的记录。如果首先用 WHERE 子句检查一下就安全了，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. |
+-----+-----+
```

从中可以看到条件还应该更特殊一些：

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

现在我们明白了能选择出所需记录的 WHERE 子句了，因此 DELETE 查询可正确地构造如下：

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

似乎删除一个记录需要做许多工作，不是吗？但是安全第一！（如果想使键盘输入工作尽量少，可利用拷贝和粘贴技术或采用输入行编辑技术。更详细的信息，请参阅“与 mysql

交互的技巧”一节。)

为了修改现有记录,可利用 UPDATE 语句,它具有下列格式:

```
UPDATE tbl_name SET 要更改的列  
WHERE 要更新的记录
```

这里的 WHERE 子句正如 DELETE 语句一样,是可选的,因此如果不指定的话,表中的每个记录都被更新。下面的查询将每个学生的名字都更改为“George”:

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

显然,对于这样的查询必须极为小心。

一般对正在更新的记录要更为小心。假定近来增加了一个新记录到历史同盟,但是只填写了此实体的少数几个列:

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

然后意识到忘了设置其会员终止日期。那么可如下进行设置:

```
mysql> UPDATE member  
-> SET expiration='2001-7-20'  
-> WHERE last_name='York' AND first_name='Jerome';
```

可同时更新多个列。下面的语句将更新 Jerome 的电子邮件和通信地址:

```
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';
```

还可以通过设置某列的值为 NULL (假设此列允许 NULL 值)“不设置”此列。如果在未来的某个时候 Jerome 决定支付成为终生会员的会员资格更新费,那么可以设置其记录的终止日期为 NULL (“永久”)以标记他为终生会员。具体设置如下:

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

正如 DELETE 语句一样,对于 UPDATE,用 SELECT 语句测试 WHERE 子句以确保选择正确的更新记录是一个好办法。如果选择条件范围太窄或太宽,就会使更新的记录太少或太多。

如果您试验过本节中的查询,那么必定已经删除和修改了 samp_db 表中的记录。在继续学习下一节的内容以前,应该撤消这些更改。按 1.4.7 节“增加新记录”最后的说明重新装载表的内容来完成这项工作。

1.4.10 改变表的结构

回顾我们创建历史同盟 member 表时缺了一个会员号列,因此我们可以进行一次 ALTER TABLE 语句的练习。需要用 ALTER TABLE,可以对表重新命名,增加或删除列,更改列的类型等等。这里给出的例子是关于怎样增加新列的。有关 ALTER TABLE 功能的详细内容,请参阅第3章。

增加会员号列到 member 表的主要考虑是,其值应该是唯一的,以免各会员条目混淆。AUTO_INCREMENT 列在此是很有用的,因为我们可以增加新的号码时令 MySQL 自动地生成唯一的号码。在 CREATE TABLE 语句中,这样一个列的说明如下:


```
member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
```

对于 ALTER TABLE，相应的句法也是类似的。可执行下列查询增加该列：

```
mysql> ALTER TABLE member  
-> ADD member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
```

我们已经有一个存放会员号的列，现在怎样分配会员号给 member 表中的现有记录呢？很容易！MySQL 已经做了这项工作。在增加一列到某个表时，MySQL 将会用缺省值初始化该列值。对于 AUTO_INCREMENT 列，每个行将会产生一个新的顺序号。

1.5 与 mysql 交互的技巧

本节介绍怎样更有效地且键入工作量较小地与 mysql 客户机程序进行交互。介绍怎样更简单地与服务器连接，以及怎样不用每次都从头开始键入查询。

1.5.1 简化连接过程

在激活 mysql 时，有可能需要指定诸如主机名、用户名或口令这样的连接参数。运行一个程序需要做很多输入工作，这很快就会让人厌烦。有几种方法可最小化所做的键入工作，使连接更为容易，它们分别为：

- 利用选项文件存储连接参数。
- 利用外壳程序的命令历史重复命令。
- 利用外壳程序的别名或脚本定义 mysql 命令行快捷键。

1. 利用选项文件

自版本3.22 以来，MySQL 允许在一个选项文件中存储连接参数。然后在运行 mysql 时就不用重复键入这些参数了；仅当您曾经在命令行上键入过它们时可以使用。这些参数也可以为其他 MySQL 客户机所用，如为 mysqlimport 所用。这也表示在使用这些程序时，选项文件减少了键入工作。

为了利用选项文件方法指定连接参数，可建立一个名为 ~/.my.cnf （即主目录中的一个名为 .my.cnf 的文件）。选项文件是一个无格式的文本文件，因此可用任何文本编辑器来创建它。文件的内容所下所示：

```
[client]  
host=serverhost  
user=yourname  
password=yourpass
```

[client] 行标记客户机选项组的开始；它后跟的所有行都是为 MySQL 客户机程序获得选项值准备的，这些行一直沿续到文件的结尾或另一不同的参数组的开始。在连接到服务器时，用指定的主机名、用户名和口令替换 serverhost、yourname 和 yourpass。对于笔者来说，.my.cnf 如下所示：

```
[client]  
host=pit-viper.snake.net  
user=paul  
password=secret
```

只有 [client] 行是必须的。定义参数值的行都是可选的；可以仅指定那些所需要的参数。例如，如果您的 MySQL 用户名与 UNIX 的登录名相同，则不需要包括 user 行。

在创建了 .my.cnf 文件后，设置其访问方式为某个限定值以保证别人不能读取它：

```
% chmod 600 .my.cnf
```

在 Windows 下，选项文件的内容是相同的，但其名称不同（c:\my.cnf），而且不调用 chmod 命令。

因为选项文件在版本 3.22 前未加到 MySQL，所以更早的版本不能使用它们。特别是在 Windows 下，您不能与共享 MySQL 分发一起得到的客户机使用选项文件，因为它是基于 MySQL 3.21 的。选项文件在注册过的 MySQL 的 Windows 版本下工作得很好，否则可以从 MySQL Web 站点取得更新的支持选项文件的客户机。

关于选项文件的详细内容可参阅附录 E “MySQL 程序参考”。

2. 利用外壳程序的命令历史

诸如 csh、tcsh 和 bash 这样的外壳程序会在一个历史列表中记下您的命令，并允许重复该列表中的命令。如果采用的是这样的外壳程序，其历史列表可帮助免除完整命令的键入。例如，如果最近调用了 mysql，可按如下命令再次执行它：

```
% !my
```

其中“！”告诉外壳程序搜索整个命令历史找到最近以“my”开头的命令，并像您打入的一样发布它。有的外壳程序还允许利用上箭头和下箭头键（或许是 Ctrl-P 和 Ctrl-N）在历史列表中上下移动。可用这种方法选择想要的命令，然后按 Enter 执行它。tcsh 和 bash 有这种功能，而其他外壳程序也可能有。可参阅相应的外壳程序以找到更多使用历史列表的内容。

3. 利用外壳程序的别名或脚本

如果使用的外壳程序提供别名功能，那么可以设置允许通过键入简短名调用长命令的命令快捷键。例如，在 csh 或 tcsh 中，可利用 alias 命令设置名为 samp_db 的别名，如下所示：

```
alias samp_db 'mysql -h pit-viper.snake.net -u paul -p samp_db'
```

而 bash 中的语法稍有不同：

```
alias samp_db='mysql -h pit-viper.snake.net -u paul -p samp_db'
```

可以定义一个别名使这两个命令等价：

```
samp_db  
mysql -h pit-viper.snake.net -u paul -p samp_db
```

显然，第一个比第二个更好键入。为了使这些别名在每次登录时都起作用，可将在外壳程序设置文件中放入一个 alias 命令（如，csh 放入 .cshrc，而 bash 放入 .bash_profile）。

快捷键的其他形式是建立利用适当的选项执行 mysql 的外壳程序脚本。在 UNIX 中，等价于 samp_db 别名的脚本文件如下所示：

```
#!/bin/sh  
exec mysql -h pit-viper.snake.net -u paul -p samp_db
```

如果笔者命名此脚本为 samp_db 并使其可执行（用 chmod +x samp_db），那么可以键入 samp_db 运行 mysql 并连接到笔者的数据库中。

在 Windows 下，可用批命令文件来完成相同的工作。命名文件 samp_db.bat，并在其中放入如下的行：

```
mysql -h pit-viper.snake.net -u paul -p samp_db
```

此批命令文件可通过在 DOS 控制台提示符下键入 samp_db 来执行，也可以双击它的 Windows 图标来执行。

如果访问多个数据库或连接到多个主机，则可以定义几个别名或脚本，每一个都用不同

的选项调用 `mysql`。

1.5.2 以较少的键入发布查询

`mysql` 是一个与数据库进行交互的极为有用的程序，但是其界面最适合于简短的、单行的查询。当然，`mysql` 自身并不关心某个查询是否分成多行，但是长的查询很不好键入。输入一条查询也不是很有趣的事，即使是一条较短的查询也是如此，除非发现有错误才愿意重新键入它。

有几种可用来避免不必要的键入或重新键入的技巧：

- 利用 `mysql` 的输入行编辑功能。
- 利用拷贝和粘贴。
- 以批方式运行 `mysql`。
- 利用现有数据来创建新记录以避免键入 `INSERT` 语句。

1. 利用 `mysql` 的输入行编辑器

`mysql` 具有内建的 GNU Readline 库，允许对输入行进行编辑。可以对当前录入的行进行处理，或调出以前输入的行并重新执行它们（原样执行或做进一步的修改后执行）。在录入一行并发现错误时，这是非常方便的；您可以在按 `Enter` 键前，在行内退格并进行修正。如果录入了一个有错的查询，那么可以调用该查询并对其进行编辑以解决问题，然后再重新提交它。（如果您在一行上键入了整个查询，这是最容易的方法。）

表1-4 中列出了一些非常有用的编辑序列，除了此表中给出的以外，还有许多输入编辑命令。利用因特网搜索引擎，应该能够找到 `Readline`手册的联机版本。此手册也包含在 `Readline` 分发包中，可在 <http://www.gnu.org/> 的 GNU Web 站点得到。

表1-4 `mysql` 输入编辑命令

键 序 列	说 明
Up 箭头, Ctrl-P	调前面的行
Down 箭头, Ctrl-N	调下一行
Left 箭头, Ctrl-B	光标左移（向后）
Right 箭头, Ctrl-F	光标右移（向前）
Escape Ctrl-B	向后移一个词
Escape Ctrl-F	向前移一个词
Ctrl-A	将光标移到行头
Ctrl-E	将光标移到行尾
Ctrl-D	删除光标下的字符
Delete	删除光标左边的字符
Escape D	删词
Escape Backspace	删除光标左边的词
Ctrl-K	删除光标到行尾的所有字符
Ctrl-_	撤消最后的更改；可以重复

下面的例子描述了输入编辑的一个简单的使用。假定用 `mysql` 输入了下列查询：

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

如果在按 `Enter` 前，已经注意到将“`president`”错拼成了“`persident`”，则可按左箭头或 `Ctrl-B` 多次移动光标到“`s`”的左边。然后按 `Delete` 两次删除“`er`”，键入“`re`”改正错误，并按 `Enter` 发布此查询。如果没注意到错拼就按了 `Enter`，也不会有问题。在 `mysql` 显示了错误

消息后, 按上箭头或 Ctrl-P 调出该行, 然后对其进行编辑。

输入行编辑在 mysql 的 Windows 版中不起作用, 但是可从 MySQL Web 站点取得免费的 cygwin_32 客户机分发版。在该分发版中的 mysqlc 程序与 mysql 一样, 但它支持输入行编辑命令。

2. 利用拷贝和粘贴发布查询

如果是在窗口环境下工作, 可将认为有用的查询文本保存在一个文件中并利用拷贝和粘贴操作很容易地发布这些命令。其工作过程如下:

1) 在 Telnet 窗口或 DOS 控制窗口中激活 mysql。

2) 在一个文档窗口打开包含查询的文件。(如笔者在 Mac OS 下使用 BBEdit, 在 UNIX 中使用 X Window System 下的 xterm 窗口中的 vi。)

3) 为了执行存放在文件中的某个查询, 选择并拷贝它。然后切换到 Telnet 窗口或 DOS 控制台, 并将该查询粘贴到 mysql。

这个过程写起来似乎有点令人讨厌, 但它是一个快速录入查询的很容易的方法, 实际使用时不用键入查询。

这个方法也允许在文档窗口中对查询进行编辑, 而且它允许拷贝和粘贴现有查询来构造一个新的查询。例如, 如果您经常从某个特定的表中选择行, 但是喜欢查看以不同方式存放的输出结果, 则可以在文档窗口中保存一个不同的 ORDER BY 子句的列表, 然后为任意的特定查询拷贝和粘贴想使用的那个句子。

也可按其他方向拷贝和粘贴(从 Telnet 到查询文件)。在 mysql 中录入行时, 它们被保存在您的主目录中的名为 .mysql_history 的文件中。如果您手工录入了一个希望保存起来今后使用的查询, 可退出 mysql, 在某个编辑器中打开 .mysql_history, 然后从 .mysql_history 拷贝和粘贴此查询到您的查询文件。

3. 以批方式运行 mysql

不一定必须交互式地运行 mysql。mysql 能够以非交互式(批)方式从某个文件中读取输入。这对于定期运行的查询是很有用的, 因为您一定不希望每次运行此查询时都要重新键入它。只要一次性地将其放入一个文件, 然后让 mysql 在需要时执行该文件的内容即可。

假定有一个查询查找 member 表的 interests 列, 以找出那些对美国历史的某个方面感兴趣的历史同盟会员。如查找对大萧条期有兴趣的会员, 可编写此查询如下(注意结尾处有一个分号, 从而 mysql 能够知道查询语句在何处结束):

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

将此查询放入文件 interests.sql 中, 然后按如下方法将其送入 mysql 执行:

```
% mysql samp_db < interests.sql
```

如果以批方式运行, 缺省时, mysql 以制表符分隔格式产生输出。如果想得到与交互式地运行 mysql 时相同的表格(“方框”)式输出, 可使用 -t 选项:

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

如果想要保存输出结果, 可将其送入文件:

```
% mysql -t samp_db < interests.sql > output_file
```

为了使用此查询来找出对 Thomas Jefferson 感兴趣的会员，可以编辑此查询文件将 depression 更改为 Jefferson 并再次运行 mysql。只要不很经常使用此查询，它工作得很好。如果经常使用，则需要更好的方法。使用此查询更为灵活的一种方法是将其放入一个外壳程序脚本中，此脚本从脚本命令行取一个参数并利用它来更改查询的文本。这样确定查询的参数，使得能够在运行脚本时指定令人感兴趣的关键词。为了了解这如何起作用，按如下编写一个较小的外壳程序脚本 interests.sh：

```
#!/bin/sh
if [ $# -ne 1 ]; then echo "Please specify one keyword"; exit; fi
mysql -t samp_db <<QUERY_INPUT
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE "%$1%"
ORDER BY last_name, first_name;
QUERY_INPUT
```

其中第二行保证在命令行上有一个关键字；它显示一条简短的消息，或者退出。在 <<QUERY_INPUT 和最后的 QUERY_INPUT 之间的所有内容成为 mysql 的输入。在查询文本中，外壳程序用来自命令行的关键字替换 \$1。（在外壳程序脚本中，\$1、\$2...为命令参数。）这使相应的查询反映了执行此脚本时在命令行上指定的关键字。

在能够运行此脚本前，必须使其可执行：

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

现在不需要在每次运行脚本时对其进行编辑了。只要在命令行上告诉它需要查找什么就行了。如下所示：

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

4. 利用现有数据来创建新记录

可以用 INSERT 语句每次一行地将新记录追加到表中，但是在通过手工键入 INSERT 语句建立几个新记录后，多数人都意识到应该有更好的追加记录的方法。一种选择是利用仅含有数据值的文件，然后利用 LOAD DATA 语句或 mysqlimport 实用程序从该文件中装入记录。

通常，可利用已经以某种格式存在的数据来建立数据文件。这些数据信息可能包含在电子表中，或许在某个其他数据库中，应该将它们转换到 MySQL。为了介绍起来简单，我们假定这些数据是在桌面微计算机的电子表中。

要将电子表数据从桌面微计算机中转换到您的 UNIX 账号下的某个文件中，可结合 Telnet 利用拷贝和粘贴。具体工作如下所示：

1) 打开UNIX 账号的一个 Telnet 连接。在 Mac OS 下，可利用诸如 Better Telnet 或 NCSA Telnet 这样的应用程序。在 Windows 下，可使用标准的 Telnet 程序。

2) 打开电子表，选择想转换的数据块，拷贝它。

3) 在 Telnet 窗中，键入下列命令开始获取数据到文件 data.txt。

```
% cat > data.txt
```

cat 命令等待输入。

4) 将从电子表拷贝来的数据粘贴到 Telnet 窗口。cat 认为您正在键入信息并忠实地将它写入到 data.txt 文件。

5) 在所有粘贴数据已经写入该文件后，如果光标停止在数据行的结尾处而不是停止在新行的开始，按 Enter。然后，按 Ctrl-D 以指示“文件结束”。cat 停止输入等待并关闭 data.txt

文件。

现在已经得到了包含有电子表中选择的数据块的 `data.txt` 文件，此文件已作好由 `LOAD DATA` 或 `mysqlimport` 加载到数据库的准备。

拷贝和粘贴是一种将数据传入 `UNIX` 文件的快速且简易的方法，但它最适合较小的数据集。量较大的数据可能会超出系统拷贝缓冲区。在这样的情况下，最好是以无格式文本（制表符分隔）的形式保存电子表。然后可利用 `FTP` 将相应文件从微机上传送到 `UNIX` 账号。转换文本模式（非二进制或影像模式）的文件以便行结束符转换为 `UNIX` 的行结束符。（`UNIX` 利用换行符、`Mac OS` 利用回车换行符、`Windows` 利用回车换行符/换行符对作为行结束符。）可告诉 `LOAD DATA` 或 `mysqlimport` 寻找什么换行符，但是在 `UNIX` 下，对含换行符的文件处理要更容易一些。

在转换了文件之后，应该检查一下在结尾处是否具空白行。如果有，应该将它们删除，否则在将该文件装载到数据库时，这些空白行将会转换为空白或畸形的记录。

来自电子表格以无格式文本保存的文件，或具有能括住包含空格的值的括号。为了在将该文件装入数据库时去掉这些括号，可利用 `LOAD DATA` 的 `FIELDS ENCLOSED BY` 子句，或利用 `mysqlimport` 的 `--fields - enclosed - by` 选项。更详细的信息请参看附录 D 中 `LOAD DATA` 的相应项。

1.6 向何处去

现在我们已经介绍了许多使用 `MySQL` 的知识。您已经知道了怎样设置数据库并创建表。能够将记录放入这些表中，并以各种方式对其进行检索，更改或删除。但是要掌握 `MySQL` 仍然有许多知识要学，本章中的教程仅仅给出了一些浅显的东西。通过考察我们的样例数据库就会明白这一点。我们创建了样例数据库及其表，并用一些初始的数据对其进行了填充。在这个工作过程中，我们明白了怎样编写查询，回答关于数据库中信息的某些问题，但是还有许多工作要做。

例如，我们没有方便的交互方式来输入学分保存方案的新学分记录，或输入历史同盟地址名录的会员条目。还没有方便的方法来编辑现有记录，而且我们仍然不能生成印刷或联机形式的同盟地址名录。这些任务以及一些其他的任务将在以后的各章中陆续地进行介绍，特别是在第 7 章“`Perl DBI API`”和第 8 章“`PHP API`”中将要进行详细地介绍。

下一步将阅读本书中哪部分取决于您对什么内容感兴趣。如果希望了解怎样完成已经以历史同盟和学分保存方案开始的工作，可看第一部分有关 `MySQL` 程序设计的内容。如果打算成为某个站点的 `MySQL` 管理员，本书的第三部分将对管理工作做较多的介绍。但是，笔者建议通过阅读第一部分中的其余各章，首先获得使用 `MySQL` 的一般背景知识。这些章节讨论了 `MySQL` 怎样处理数据，进一步提供有关语法和查询语句的用途，并且说明了怎样使查询运行得更快。不管您在什么环境中使用 `MySQL`，不管是运行 `mysql` 还是编写自己的程序，还是作为数据库管理员，用这些内容打下一个良好的基础将有助于您站在一个较高的起点上。

第2章 用 MySQL 处理数据

根据定义，数据库管理系统的目的就是管理数据。即使一条简单的 `SELECT 1` 语句也涉及表达式求值以产生一个整型数据值。

MySQL 中的每个数据值都有类型。例如，37.4 是一个数，而“abc”是一个串。有时，数据的类型是明显的，因为在使用 `CREATE TABLE` 语句时指定了作为表的组成部分定义的每个列的类型，如：

```
CREATE TABLE my_table
(
    int_col INT,           /* integer-valued column */
    str_col CHAR(20),      /* string-valued column */
    date_col DATE          /* date-valued column */
)
```

而有时，数据类型是不明确的，如在一个表达式中引用直接值时，将值传送给一个函数，或使用从该函数返回的值，如：

```
INSERT INTO my_table (int_col,str_col,date_col)
VALUES(14,CONCAT("a","b"),19990115)
```

`INSERT` 语句完成下列操作，这些操作全都涉及数据类型：

- 将整数值 14 赋给整数列 `int_col`。
- 将串值“a”和“b”传递给函数 `CONCAT()`。`CONCAT()` 返回串值“ab”，这个串值被赋予串列 `str_col`。
- 将整数值 19990115 赋给日期列 `date_col`。而这是不匹配的，因此，MySQL 将自动进行数据类型转换。

要有效地利用 MySQL，必须理解其怎样处理数据。本章描述了 MySQL 能够处理的数据类型，并讨论了在处理这些数据类型时所出现的问题，主要内容如下：

- 通用数据类型，包括 `NULL` 值。
- 特殊数据类型，以及描述每种列类型的属性。有些列类型是相当常见的，如 `CHAR` 串类型。而有的如 `AUTO_INCREMENT` 整型和 `TIMESTAMP` 日期类型，其性能很特殊，应该加以理解以免出错。
- 恰当地选择表的列类型。在创建表时，重要的是要了解怎样为自己的目的选择最好的类型，以及在几种类型都可以用于想要存储的值时选择一种类型。
- 表达式求值规则。MySQL 提供了许多可用于表达式的运算符和函数，以便对数据进行检索、显示和处理。表达式求值的规则包括类型转换规则，在一种类型的值用于另一类型的值的情况时需用到类型转换规则。

理解何时进行类型转换以及怎样进行转换很重要；有的转换没有意义而且会产生错误值。将串“13”赋给整数列结果为值 13，但是将串“abc”赋给该列得到 0 值，因为“abc”不是一个数。更坏的是，如果进行比较而不了解值的转换，可能会带来很大的危险，如在打算只对几行进行操作时，可能会更新或删除了表中的所有行。

附录 B 和附录 C 提供了 MySQL 列类型、运算和函数的更多信息。

2.1 MySQL 数据类型

MySQL 有几种数据类型，下面分别进行介绍。

1. 数值值

数值是诸如 48 或 193.62 这样的值。MySQL 支持说明为整数（无小数部分）或浮点数（有小数部分）的值。整数可按十进制形式或十六进制形式表示。

整数由数字序列组成。以十六进制形式表示的整数由“0x”后跟一个或多个十六进制数字（“0”到“9”及“a”到“f”）组成。例如，0x0a 为十进制的 10，而 0xffff 为十进制的 65535。十六进制数字不区分大小写，但其前缀“0x”不能为“0X”。即 0x0a 和 0x0A 都是合法的，但 0X0a 和 0X0A 不是合法的。

浮点数由一个阿拉伯数字序列、一个小数点和另一个阿拉伯数字序列组成。两个阿拉伯数字序列可以分别为空，但不能同时为空。

MySQL 支持科学表示法。科学表示法由整数或浮点数后跟“e”或“E”、一个符号（“+”或“-”）和一个整数指数来表示。1.34E+12 和 43.27e-1 都是合法的科学表示法表示的数。而 1.34E12 不是合法的，因为指数前的符号未给出。指数前的“e”也是一个合法的十六进制数字，因此有可能会弄错。

数值前可放一个负号“-”以表示负值。

2. (字符) 串值

串是诸如“Madison, Wisconsin”或“patient shows improvement”这样的值。既可用单引号也可用双引号将串值括起来。

串中可使用几个转义序列，它们用来表示特殊的字符，见表 2-1。每个序列以一个反斜杠（“\”）开始，指出临时不同于通常的字符解释。注意 NUL 字节与 NULL 值不同；NUL 为一个零值字节，而 NULL 为没有值。

表2-1 串转义序列

序 列	说 明	序 列	说 明
\0	NUL (ASCII 0)	\n	新行
\'	单引号	\r	回车
\"	双引号	\t	制表符
\b	退格	\\	反斜杠

要在串中包括一个引号，可有如下三种选择：

■ 如果串是用相同的引号括起来的，那么在串中需要引号的地方双写引号即可。如：

```
'I can't'
"He said, ""I told you so.""
```

■ 如果串是用另外的引号括起来的，则不需要双写相应引号。如：

```
"I can't"
'He said, "I told you so."'
```

■ 用反斜杠方式表示；这种方法不去管用来将串括起的是单引号还是双引号。如：

```
'I can\'t'
"I can\'t"
"He said, \"I told you so.\""
'He said, \"I told you so.\"'
```

在串的环境中，可用十六进制常数来指定串值。其语法与前面描述的数值值相同，但是每对十六进制的数字都被看作 ASCII 代码并转换为字符，其结果用于串。例如，0x616263作为串时为“abc”。

3. 日期和时间值

日期和时间是一些诸如“1999-06-17”或“12:30:43”这样的值。MySQL 还支持日期/时间的组合，如“1999-06-17 12:30:43”。要特别注意这样一个事实，即 MySQL 是按年-月-日的顺序表示日期的。MySQL 的初学者通常对这一点很惊奇，其实这是 ANSI SQL 的标准格式。可以利用 DATE_FORMAT() 函数以任意形式显示日期值，但是缺省显示格式首先显示年，而且输入值也必须首先给出年。

4. NULL 值

NULL 是一种“无类型”的值。它过去惯常表示的意思是“无值”、“未知值”、“丢失的值”、“溢出值”以及“没有上述值”等。可将 NULL 值插入表中、从表中检索它们，测试某个值是否是 NULL，但不能对 NULL 值进行算术运算（如果对 NULL 进行算术运算，其结果为 NULL）。

2.2 MySQL 的列类型

数据库中的每个表都是由一个或多个列构成的。在用 CREATE TABLE 语句创建一个表时，要为每列指定一个类型。列的类型比数据类型更为特殊，它仅仅是如“数”或“串”这样的通用类型。列的类型精确地描述了给定表列可能包含的值的种类，如 SMALLINT 或 VARCHAR(32)。

MySQL 的列类型是一种手段，通过这种手段可以描述一个表列包含什么类型的值，这又决定了 MySQL 怎样处理这些值。例如，数值值既可用数值也可用串的列类型来存放，但是根据存放这些值的类型，MySQL 对它们的处理将会有些不同。每种列类型都有几个特性如下：

- 其中可以存放什么类型的值。
- 值要占据多少空间，以及该值是否是定长的（所有值占相同数量的空间）或可变长的（所占空间量依赖于所存储的值）。
- 该类型的值怎样比较和存储。
- 此类型是否允许 NULL 值。
- 此类型是否可以索引。

我们将简要地考察一下 MySQL 列类型以获得一个总的概念，然后更详细地讨论描述每种列类型的属性。

2.2.1 列类型概述

MySQL 为除 NULL 值以外的所有通用数据类型的值都提供了列类型。在列是否能够包含 NULL 值被视为一种类型属性的意义上，可认为所有类型都包含 NULL 属性。

MySQL 有整数和浮点数值 的列类型，如表 2-2 所示。整数列类型可以有符号也可无符号。有一种特殊的属性允许整数列值自动生成，这对需要唯一序列或标识号的应用系统来说是非常有用的。

表2-2 数值列类型

类 型 名	说 明	类 型 名	说 明
TINYINT	非常小的整数	BIGINT	大整数
SMALLINT	较小整数	FLOAT	单精度浮点数
MEDIUMINT	中等大小的整数	DOUBLE	双精度浮点数
INT	标准整数	DECIMAL	一个串的浮点数

MySQL 串列类型如表 2-3 所示。串可以存放任何内容，即使是像图像或声音这样的绝对二进制数据也可以存放。串在进行比较时可以设定是否区分大小写。此外，可对串进行模式匹配（实际上，在 MySQL 中可以在任意列类型上进行模式匹配，但最经常进行模式匹配还是在串类型上）。

表2-3 串列类型

类 型 名	说 明
CHAR	定长字符串
VARCHAR	可变长字符串
TINYBLOB	非常小的 BLOB（二进制大对象）
BLOB	小 BLOB
MEDIUMBLOB	中等的 BLOB
LOB	大 BLOB
TINYTEXT	非常小的文本串
TEXT	小文本串
MEDIUMTEXT	中等文本串
LONGTEXT	大文本串
ENUM	枚举；列可赋予某个枚举成员
SET	集合；列可赋予多个集合成员

日期与时间列类型在表 2-4 中示出。对于临时值，MySQL 提供了日期（有或没有时间）、时间和时间戳（一种允许跟踪对记录何时进行最后更改的特殊类型）的类型。而且还提供了一种在不需要完整的日期时有效地表示年份的类型。

表2-4 日期与时间列类型

类 型 名	说 明
DATE	“YYYY-MM-DD”格式表示的日期值
TIME	“hh:mm:ss”格式表示的时间值
DATETIME	“YYYY-MM-DD hh:mm:ss”格式
TIMESTAMP	“YYYYMMDDhhmmss”格式表示的时间戳值
YEAR	“YYYY”格式的年份值

要创建一个表，应使用 CREATE TABLE 语句并指定构成表列的列表。每个列都有一个名字和类型，以及与每个类型相关的各种属性。下面是创建具有三个分别名为 f、c 和 i 的列的表 my_table 的例子：

```
CREATE TABLE my_table
(
  f FLOAT(10,4),
  c CHAR(15) NOT NULL DEFAULT "none",
  i TINYINT UNSIGNED NULL
)
```

定义一个列的语法如下：

```
col_name col_type [col_attributes] [general_attributes]
```

其中列名由 col_name 给出。列名可最多包含 64 个字符，字符包括字母、数字、下划线及美元符号。列名可以名字中合法的任何符号（包括数字）开头。但列名不能完全由数字组成，因为那样可能使其与数据分不开。MySQL 保留诸如 SELECT、DELETE 和 CREATE 这样的词，这些词不能用做列名。但是函数名（如 POS 和 MIN）是可以使用的。

列类型 col_type 表示列可存储的特定值。列类型说明符还能表示存放在列中的值的最大长度。对于某些类型，可用一个数值明确地说明其长度。而另外一些值，其长度由类型名蕴含。例如，CHAR(10) 明确指定了 10 个字符的长度。而 TINYBLOB 值隐含最大长度为 255 个字符。有的类型说明符允许指定最大的显示宽度（即显示值时使用多少个字符）。浮点类型允许指定小数位数，所以能控制浮点数的精度值为多少。

可以在列类型之后指定可选的类型说明属性，以及指定更多的常见属性。属性起修饰类型的作用，并更改其处理列值的方式，属性有以下类型：

- **专用属性用于指定列。**例如，UNSIGNED 属性只针对整型，而 BINARY 属性只用于 CHAR 和 VARCHAR。

- **通用属性除少数列之外可用于任意列。**可以指定 NULL 或 NOT NULL 以表示某个列是否能够存放 NULL。还可以用 DEFAULT def_value 来表示在创建一个新行但未明确给出该列的值时，该列可赋予值 def_value。def_value 必须为一个常量；它不能是表达式，也不能引用其他列。不能对 BLOB 或 TEXT 列指定缺省值。

如果想给出多个列的专用属性，可按任意顺序指定它们，只要它们跟在列类型之后、通用属性之前即可。类似地，如果需要给出多个通用属性，也可按任意顺序给出它们，只要将它们放在列类型和可能给出的列专用属性之后即可。

本节其余部分讨论每个 MySQL 的列类型，给出定义类型和描述它们的属性的语法，诸如取值范围和存储需求等。类型说明如在 CREATE TABLE 语句中那样给出。可选的信息由方括号 ([]) 给出。如，语法 MEDIUMINT[(M)] 表示最大显示宽度（指定为 M）是可选的。另一方面，对于 CHAR(M)，无方括号表示的 (M) 是必须的。

2.2.2 数值列类型

MySQL 的数值列类型有两种：

- **整型。**用于无小数部分的数，如 1、43、-3、0 或 -798432。可对正数表示的数据使用整数列，如磅的近似数、英寸的近似数，银河系行星的数目、家族人数或一个盘子里的细菌数等。

- **浮点数。**用于可能具有小数部分的数，如 3.14159、-.00273、-4.78、或 39.3E+4。可将浮点数列类型用于有小数点部分或极大、极小的数。可能会表示为浮点数的值有农作物平均产量、距离、钱数（如物品价格或工资）、失业率或股票价格等等。整型值也可以赋予浮点列，这时将它们表示为小数部分为零的浮点值。

每种数值类型的名称和取值范围如表 2-5 所示。各种类型值所需的存储量如表 2-6 所示。

CREATE TABLE 语句

本章中例子中大量使用了 CREATE TABLE 语句。您应该对此语句相当熟悉，因为我们在第1章中的教程部分使用过它。关于 CREATE TABLE 语句也可参阅附录 D。

表2-5 数值列类型的取值范围

类型说明	取值范围
TINYINT[(M)]	有符号值：-128 到 127 (-2^7 到 $2^7 - 1$) 无符号值：0 到 255 (0 到 $2^8 - 1$)
SMALLINT[(M)]	有符号值：-32768 到 32767 (-2^{15} 到 $2^{15} - 1$) 无符号值：0 到 65535 (0 到 $2^{16} - 1$)
MEDIUMINT[(M)]	有符号值：-8388608 到 8388607 (-2^{23} 到 $2^{23} - 1$) 无符号值：0 到 16777215 (0 到 $2^{24} - 1$)
INT[(M)]	有符号值：-2147683648 到 2147683647 (-2^{31} 到 $2^{31} - 1$) 无符号值：0 到 4294967295 (0 到 $2^{32} - 1$)
BIGINT[(M)]	有符号值：-9223372036854775808 到 9223373036854775807 (-2^{63} 到 $2^{63} - 1$) 无符号值：0 到 18446744073709551615 (0 到 $2^{64} - 1$)
FLOAT[(M, D)], FLOAT(4)	最小非零值：±1.175494351E - 38 最大非零值：±3.402823466E + 38
DOUBLE[(M, D)], FLOAT(8)	最小非零值：±2.2250738585072014E - 308 最大非零值：±1.7976931348623157E + 308
DECIMAL (M, D)	可变；其值的范围依赖于 M 和 D

表2-6 数值列类型的存储需求

类型说明	存储需求
TINYINT[(M)]	1 字节
SMALLINT[(M)]	2 字节
MEDIUMINT[(M)]	3 字节
INT[(M)]	4 字节
BIGINT[(M)]	8 字节
FLOAT[(M, D)], FLOAT(4)	4 字节
DOUBLE[(M, D)], FLOAT(8)	8 字节
DECIMAL (M,D)	M 字节 (MySQL < 3.23), M + 2 字节 (MySQL ≥ 3.23)

MySQL 提供了五种整型：TINYINT、SMALLINT、MEDIUMINT、INT 和 BIGINT。INT为INTEGER的缩写。这些类型在可表示的取值范围上是不同的。整数列可定义为 UNSIGNED 从而禁用负值；这使列的取值范围为 0 以上。各种类型的存储量需求也是不同的。取值范围较大的类型所需的存储量较大。

MySQL 提供三种浮点类型：FLOAT、DOUBLE 和 DECIMAL。与整型不同，浮点类型不能是 UNSIGNED 的，其取值范围也与整型不同，这种不同不仅在于这些类型有最大值，而且还有最小非零值。最小值提供了相应类型精度的一种度量，这对于记录科学数据来说是非常重要的（当然，也有负的最大和最小值）。

DOUBLE PRECISION[(M, D)] 和 REAL[(M, D)] 为 DOUBLE[(M, D)] 的同义词。而 NUMERIC(M, D) 为 DECIMAL(M, D) 的同义词。FLOAT(4) 和 FLOAT(8) 是为了与 ODBC 兼容而提供的。在 MySQL 3.23 以前，它们为 FLOAT(10, 2) 和 DOUBLE(16, 4) 的同义词。自

MySQL 3.23 以来, FLOAT(4) 和 FLOAT(8) 各不相同, 下面还要介绍。

在选择某种数值类型时, 应该考虑所要表示的值的范围, 只需选择能覆盖要取值的范围的最小类型即可。选择较大类型会对空间造成浪费, 使表不必要地增大, 处理起来没有选择较小类型那样有效。对于整型值, 如果数据取值范围较小, 如人员年龄或兄弟姐妹数, 则 TINYINT 最合适。MEDIUMINT 能够表示数百万的值并且可用于更多类型的值, 但存储代价较大。BIGINT 在全部整型中取值范围最大, 而且需要的存储空间是表示范围次大的整型 INT 类型的两倍, 因此只在确实需要时才用。对于浮点值, DOUBLE 占用 FLOAT 的两倍空间。除非特别需要高精度或范围极大的值, 一般应使用只用一半存储代价的 FLOAT 型来表示数据。

在定义整型列时, 可以指定可选的显示尺寸 M。如果这样, M 应该是一个 1 到 255 的整数。它表示用来显示列中值的字符数。例如, MEDIUMINT(4) 指定了一个具有 4 个字符显示宽度的 MEDIUMINT 列。如果定义了一个没有明确宽度的整数列, 将会自动分配给它一个缺省的宽度。缺省值为每种类型的“最长”值的长度。如果某个特定值的可打印表示需要不止 M 个字符, 则显示完全的值; 不会将值截断以适合 M 个字符。

对每种浮点类型, 可指定一个最大的显示尺寸 M 和小数位数 D。M 的值应该取 1 到 255。D 的值可为 0 到 30, 但是不应大于 M - 2。(如果熟悉 ODBC 术语, 就会知道 M 和 D 对应于 ODBC 概念的“精度”和“小数位数”) M 和 D 对 FLOAT 和 DOUBLE 都是可选的, 但对于 DECIMAL 是必须的。

在选项 M 和 D 时, 如果省略了它们, 则使用缺省值。下面的语句创建了一个表, 它说明了数值列类型的 M 和 D 的缺省值 (其中不包括 DECIMAL, 因为 M 和 D 对这种类型不是可选的):

```
CREATE TABLE my_table
(
  itiny TINYINT, itiny_u TINYINT UNSIGNED,
  ismall SMALLINT, ismall_u SMALLINT UNSIGNED,
  imedium MEDIUMINT, imedium_u MEDIUMINT UNSIGNED,
  ireg INT, ireg_u INT UNSIGNED,
  ibig BIGINT, ibig_u BIGINT UNSIGNED,
  fp_single FLOAT, fp_double DOUBLE
)
```

如果在创建表之后使用 DESCRIBE my_table 语句, 则输出的 Field 和 Type 列如下所示 (注意, 如果用 MySQL 的 3.23 以前的版本运行这个查询, 则有一个小故障, 即 BIGINT 的显示宽度将是 21 而不是 20。):

Field	Type
itiny	tinyint(4)
itiny_u	tinyint(3) unsigned
ismall	smallint(6)
ismall_u	smallint(5) unsigned
imedium	mediumint(9)
imedium_u	mediumint(8) unsigned
ireg	int(11)
ireg_u	int(10) unsigned
ibig	bigint(20)
ibig_u	bigint(20) unsigned
fp_single	float(10,2)
fp_double	double(16,4)

每一个数字列都具有一个由列类型所决定的取值范围。如果打算插入一个不在列范围内的值，将会进行截取：MySQL 将剪裁该值为取值范围的边界值并使用这个结果。在检索时不进行值的剪裁。

值的剪裁根据列类型的范围而不是显示宽度进行。例如，一个 `SMALLINT(3)` 列显示宽度为 3 而取值范围为 -32768 到 32767。值 12345 比显示宽度大，但在该列的取值范围内，因此它可以插入而不用剪裁并且作为 12345 检索。值 99999 超出了取值范围，因此在插入时被剪裁为 32767。以后在检索中将以值 32767 检索该值。

一般赋予浮点列的值被四舍五入到这个列所指定的十进制数。如果在一个 `FLOAT(8, 1)` 的列中存储 1.23456，则结果为 1.2。如果将相同的值存入 `FLOAT(8, 4)` 的列中，则结果为 1.2346。这表示应该定义具有足够位数的浮点列以便得到尽可能精确的值。如果想精确到千分之一，那就不要定义使该类型仅有两位小数。

浮点值的这种处理在 MySQL 3.23 中有例外，`FLOAT(4)` 和 `FLOAT(8)` 的性能有所变化。这两种类型现在为单精度（4 字节）和双精度（8 字节）的类型，在其值按给出的形式存放（只受硬件的限制）这一点上说，这两种类型是真浮点类型。

`DECIMAL` 类型不同于 `FLOAT` 和 `DECIMAL`，其中 `DECIMAL` 实际是以串存放的。`DECIMAL` 可能的最大取值范围与 `DOUBLE` 一样，但是其有效的取值范围由 `M` 和 `D` 的值决定。如果改变 `M` 而固定 `D`，则其取值范围将随 `M` 的变大而变大。表 2-7 的前三行说明了这一点。如果固定 `M` 而改变 `D`，则其取值范围将随 `D` 的变大而变小（但精度增加）。表 2-7 的后三行说明了这一点。

表 2-7 M 与 D 对 `DECIMAL(M, D)` 取值范围的影响

类型说明	取值范围 (MySQL < 3.23)	取值范围 (MySQL ≥ 3.23)
<code>DECIMAL(4, 1)</code>	-9.9 到 99.9	-999.9 到 9999.9
<code>DECIMAL(5, 1)</code>	-99.9 到 999.9	-9999.9 到 99999.9
<code>DECIMAL(6, 1)</code>	-999.9 到 9999.9	-99999.9 到 999999.9
<code>DECIMAL(6, 2)</code>	-99.99 到 999.99	-9999.99 到 99999.99
<code>DECIMAL(6, 3)</code>	-9.999 到 99.999	-999.999 到 9999.999

给定的 `DECIMAL` 类型的取值范围取决于 MySQL 的版本。对于 MySQL 3.23 以前的版本，`DECIMAL(M, D)` 列的每个值占用 `M` 字节，而符号（如果需要）和小数点包括在 `M` 字节中。因此，类型为 `DECIMAL(5, 2)` 的列，其取值范围为 -9.99 到 99.99，因为它们覆盖了所有可能的 5 个字符的值。

正如 MySQL 3.23 一样，`DECIMAL` 值是根据 ANSI 规范进行处理的，ANSI 规范规定 `DECIMAL(M, D)` 必须能够表示 `M` 位数字及 `D` 位小数的任何值。例如，`DECIMAL(5, 2)` 必须能够表示从 -999.99 到 999.99 的所有值。而且必须存储符号和小数点，因此自 MySQL 3.23 以来 `DECIMAL` 值占 `M + 2` 个字节。对于 `DECIMAL(5, 2)`，“最长”的值（-999.99）需要 7 个字节。在正取值范围的一端，不需要正号，因此 MySQL 利用它扩充了取值范围，使其超过了 ANSI 所规范所要求的取值范围。如 `DECIMAL(5, 2)` 的最大值为 9999.99，因为有 7 个字节可用。

简而言之，在 MySQL 3.23 及以后的版本中，`DECIMAL(M, D)` 的取值范围等于更早版本中的 `DECIMAL(M + 2, D)` 的取值范围。

在 MySQL 的所有版本中，如果某个 `DECIMAL` 列的 `D` 为 0，则不存储小数点。这样做

的结果是扩充了列的取值范围，因为过去用来存储小数点的字节现在可用来存放其他数字了。

1. 数值列的类型属性

可对所有数值类型指定 `ZEROFILL` 属性。它使相应列的显示值用前导零来填充，以达到显示宽度。在希望确定列值总是以给定的数字位数显示时可利用 `ZEROFILL`。实际上，更准确地说是“一个给定的最小数目的数字位数”，因为比显示宽度更宽的值可完全显示而未被剪裁。使用下列语句可看到这一点：

```
CREATE TABLE my_table (my_zerofill INT(5) ZEROFILL)
INSERT INTO my_table VALUES(1),(100),(10000),(1000000)
SELECT my_zerofill FROM my_table
```

其中 `SELECT` 语句的输出结果如下。请注意最后一行值，它比列的显示宽度更宽，但仍然完全显示出来：

```
+-----+
| my_zerofill |
+-----+
|          00001 |
|          00100 |
|          10000 |
|        1000000 |
+-----+
```

如下所示两个属性只用于整数列：

■ **AUTO_INCREMENT**。在需要产生唯一标识符或顺序值时，可利用 `AUTO_INCREMENT` 属性。`AUTO_INCREMENT` 值一般从1开始，每行增加 1。在插入 `NULL` 到一个 `AUTO_INCREMENT` 列时，MySQL 插入一个比该列中当前最大值大 1 的值。一个表中最多只能有一个 `AUTO_INCREMENT` 列。

对于任何想要使用 `AUTO_INCREMENT` 的列，应该定义为 `NOT NULL`，并定义为 `PRIMARY KEY` 或定义为 `UNIQUE` 键。例如，可按下列任何一种方式定义 `AUTO_INCREMENT` 列：

```
CREATE TABLE ai (i INT AUTO_INCREMENT NOT NULL PRIMARY KEY)
CREATE TABLE ai (i INT AUTO_INCREMENT NOT NULL, PRIMARY KEY (i))
CREATE TABLE ai (i INT AUTO_INCREMENT NOT NULL, UNIQUE (i))
```

`AUTO_INCREMENT` 的性能将在下一小节“使用序列”中作进一步的介绍。

■ **UNSIGNED**。此属性禁用负值。将列定义为 `UNSIGNED` 并不改变其基本数据类型的取值范围；它只是前移了取值的范围。考虑下列的表说明：

```
CREATE TABLE my_table
(
    itiny TINYINT,
    itiny_u TINYINT UNSIGNED
)
```

`itiny` 和 `itiny_u` 两列都是 `TINYINT`列，并且都可取256个值，但是`itiny`的取值范围为 -128 到 127，而`itiny_u` 的取值范围为 0 到 255。`UNSIGNED` 对不取负值的列是非常有用的，如存入人口统计或出席人数的列。如果用常规的有符号列来存储这样的值，那么就只利用了该列类型取值范围的一半。通过使列为 `UNSIGNED`，能有效地成倍增加其取值范围。如果将列用于序列号，且将它设为 `UNSIGNED`，则可取原双倍的值。在指定以上属性之后（它们是专门用于数值列的），可以指定通用属性 `NULL` 或 `NOT`

NULL。如果未指定 NULL 或 NOT NULL，则缺省为 NULL。也可以用 DEFAULT 属性来指定一个缺省值。如果不指定缺省值，则会自动选择一个。对于所有数值列类型，那些可以包含 NULL 的列的缺省将为 NULL，不能包含 NULL 的列其缺省为 0。

下面的样例创建三个 INT 列，它们分别具有缺省值 -1、1 和 NULL：

```
CREATE TABLE t
(
    i1 INT DEFAULT -1,
    i2 INT DEFAULT 1,
    i3 INT DEFAULT NULL
)
```

2. 使用序列

许多应用程序出于标识的目的需要使用唯一的号码。需要唯一值的这种要求在许多场合都会出现，如：会员号、试验样品编号、顾客 ID、错误报告或故障标签等等。

AUTO_INCREMENT 列可提供唯一编号。这些列可自动生成顺序编号。本节描述 AUTO_INCREMENT 列是怎样起作用的，从而使您能够有效地利用它们而不至于出错。另外，还介绍了怎样不用 AUTO_INCREMENT 列来产生序列的方法。

(1) MySQL 3.23 以前的版本中的 AUTO_INCREMENT

MySQL 3.23 版以前的 AUTO_INCREMENT 列的性能如下：

- 插入 NULL 到 AUTO_INCREMENT 列，使 MySQL 自动地产生下一个序列号并将此序列号自动地插入列中。AUTO_INCREMENT 序列从 1 开始，因此插入表中的第一个记录得到为 1 的序列值，而后继插入的记录分别得到序列值 2、3 等等。一般，每个自动生成的值都比存储在该列中的当前最大值大 1。
- 插入 0 到 AUTO_INCREMENT 与插入 NULL 到列中的效果一样。插入一行而不指定 AUTO_INCREMENT 列的值也与插入 NULL 的效果一样。
- 如果插入一个记录并明确指定 AUTO_INCREMENT 列的一个值，将会发生两件事之一。如果已经存在具有该值的某个记录，则出错，因为 AUTO_INCREMENT 列中的值必须是惟一的。如果不存在具有该值的记录，那么新记录将被插入，并且如果新记录的 AUTO_INCREMENT 列中的值是新的最大值，那么后续行将用该值的下一个值。换句话说，也就是可以通过插入一个具有比当前值大的序列值的记录，来增大序列的计数器。

增大计数器会使序列出现空白，但这个特性也有用。例如创建一个具有 AUTO_INCREMENT 列的表，但希望序列从 1000 而不是 1 开始。则可以用后述的两种办法之一达到此目的。一个办法是插入具有明确序列值 1000 的第一个记录，然后通过插入 NULL 到 AUTO_INCREMENT 列来插入后续的记录。另一个办法是插入 AUTO_INCREMENT 列值为 999 的假记录。然后第一个实际插入的记录将得到一个序列号 1000，这时再将假记录删除。

- 如果将一个不合规定的值插入 AUTO_INCREMENT 列，将会出现难以预料的结果。
- 如果删除了在 AUTO_INCREMENT 列中含有最大值的记录，则此值在下一次产生新值时会再次使用。如果删除了表中的所有记录，则所有值都可以重用；相应的序列重新从 1 开始。
- REPLACE 语句正常起作用。

- UPDATE 语句按类似插入新记录的规则起作用。如果更新一个 AUTO_INCREMENT 列为 NULL 或 0, 则会自动将其更新为下一个序列号。如果试图更新该列为一个已经存在的值, 将出错 (除非碰巧设置此列的值为它所具有的值, 才不会出错, 但这没有任何意义)。如果更新该列的值为一个比当前任何列值都大的值, 则以后序列将从下一个值继续进行编号。
- 最近自动产生的序列编号值可调用 LAST_INSERT_ID() 函数得到。它使得能在其他不知道此值的语句中引用 AUTO_INCREMENT 值。LAST_INSERT_ID() 依赖于当前服务器会话中生成的 AUTO_INCREMENT 值; 它不受与其他客户机相关的 AUTO_INCREMENT 活动的影响。如果当前会话中没有生成 AUTO_INCREMENT 值, 则 LAST_INSERT_ID() 返回 0。

能够自动生成顺序编号这个功能特别有用。但是刚才介绍的 AUTO_INCREMENT 性能有两个缺陷。首先, 序列中顶上的记录被删除时, 序列值的重用使得难于生成可能删除和插入记录的应用的一系列单调 (严格递增) 值。其次, 利用从大于 1 的值开始生成序列的方法是很笨的。

(2) MySQL 3.23 版以后的 AUTO_INCREMENT

MySQL 3.23 对 AUTO_INCREMENT 的性能进行了下列变动以便能够处理上述问题:

- 自动顺序生成的值严格递增且不重用。如果最大的值为 143 并删除了包含这个值的记录, MySQL 继续生成下一个值 144。
- 在创建表时, 可以明确指定初始的序列编号。下面的例子创建一个 AUTO_INCREMENT 列 seq 从 1,000,000 开始的表:

```
CREATE TABLE my_table
  (seq INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY)
  AUTO_INCREMENT = 1000000
```

在一个表具有多个列时 (正如多数表那样), 最后的 AUTO_INCREMENT = 1000000 子句应用到哪一列是不会混淆的, 因为每个表只能有一个 AUTO_INCREMENT 列。

(3) 使用 AUTO_INCREMENT 应该考虑的问题

在使用 AUTO_INCREMENT 列时, 应该记住下列要点:

- AUTO_INCREMENT 不是一种列类型, 它只是一种列类型属性。此外, AUTO_INCREMENT 是一种只能用于整数类型的属性。MySQL 早于 3.23 的版本并不严格服从这个约束, 允许定义诸如 CHAR 这样的列类型具有 AUTO_INCREMENT 属性。但是只有整数类型作为 AUTO_INCREMENT 列正常起作用。
- AUTO_INCREMENT 机制的主要目的是生成一个正整数序列, 并且如果以这种方式使用, 则 AUTO_INCREMENT 列效果最好。所以应该定义 AUTO_INCREMENT 列为 UNSIGNED。这样做的优点是在到达列类型的取值范围上限前可以进行两倍的序列编号。在某些环境下, 也有可能利用 AUTO_INCREMENT 列来生成负值的序列, 但是我们不建议这样做。如果您决定要试一下, 应该保证进行充分的试验, 并且在升级到不同的 MySQL 版本时需要重新测试。笔者的经验表明, 不同的版本中, 负序列的性能并不完全一致。
- 不要认为对某个列定义增加 AUTO_INCREMENT 是一个得到无限的编号序列的奇妙方法。事实并非这样; AUTO_INCREMENT 序列受基础列类型的取值范围所限制。例如,

如果使用 TINYINT UNSIGNED 列, 则最大的序列号为 255。在达到这个界限时, 应用程序将开始出现“重复键”错误。

- MySQL 3.23 引入了不重用序列编号的新 AUTO_INCREMENT 性能, 并且允许在 CREATE TABLE 语句中指定一个初始的序列编号。这些性能在使用下列形式的 DELETE 语句删除了表中所有记录后可以撤消:

```
DELETE FROM tbl_name
```

在此情形下, 序列重新从1开始而不按严格的增量顺序继续增加。即使在 CREATE TABLE 语句中明确指定了一个初始的序列编号, 相应的序列也会从头开始。出现这种情形的原因在于 MySQL 优化完全删空一个表的 DELETE 语句的方法上; 它从头开始重新创建数据文件和索引文件而不是去删除每个记录, 这样就丢失了所有的序列号信息。如果要删除所有记录, 但希望保留序列信息, 可以取消优化并强制 MySQL 执行逐行的删除操作, 如下所示:

```
DELETE FROM tbl_name WHERE 1 > 0
```

如果使用的是3.23以上的版本, 怎样保持严格的增量序列? 方法之一是保持一个只用来生成 AUTO_INCREMENT 值的独立的表, 永远不从这个表中删除记录。在这种情况下, 独立表中的值永远不会重用。在主表中需要生成一个新记录时, 首先在序列编号表中插入一个 NULL。然后对希望包含序列编号的列使用 LAST_INSERT_ID() 的值将该记录插入主表, 如下所示:

```
INSERT INTO ai_tbl SET ai_col = NULL  
INSERT INTO main_tbl SET id=LAST_INSERT_ID() ...
```

如果想要编写一个生成 AUTO_INCREMENT 值的应用程序, 但希望序列从 100 而不是 1 开始。再假定希望这个程序可移植到所有 MySQL 版本。怎样来完成它呢?

如果可移植是一个目标, 那么不能依赖 MySQL 3.23 所提供的在 CREATE TABLE 语句中指定初始序列编号的功能。而是在想要插入一个记录时, 首先用下列语句检查表是否是空的:

```
SELECT COUNT(*) FROM tbl_name
```

这个步骤虽然是附加的, 但不会花费太多的时间, 因为没有 WHERE 子句的 SELECT COUNT(*) 是优化的, 返回很快。如果表是空的, 则插入记录并明确地对序列编号列指定值 100。如果表不空, 则对序列编号列值指定 NULL 使 MySQL 自动生成下一个编号。

此方法允许插入序列编号为 100、101 等的记录, 它不管 MySQL 是否允许指定初始序列值都能正常工作。如果要求序列编号即使是从表中删除了记录后也要严格递增, 则此方法不起作用。在这样的情形下, 可将此方法与前面描述的什么也不做只是用来产生用于主表的序列编号的辅助表技术结合使用。

为什么会希望从一个大于 1 的序列编号开始呢? 一个原因是想使所有序列编号全都具有相同的数字位数。如果需要生成顾客 ID 号, 并且希望不要多于一百万个顾客, 则可以从 1 000 000 开始编号。在对顾客 ID 值计数的数字位数改变之前, 可以追加一百万个顾客。

当然, 强制序列编号为一个固定宽度的另一个方法是采用 ZEROFILL 列。对于有的情形, 这样做有可能会出问题。例如, 如果在 Perl 或 PHP 脚本中处理具有前导零的序列编号, 则必须仔细地将它们只作为串使用; 如果将它们转换成数字, 前导零将会丢失。下面的短 Perl 脚本说明了处理编号时可能会出的问题:


```
#!/usr/bin/perl
$s = "00010"; # create "number" with leading zeroes
print "$s\n";
$s++; # use Perl's intelligent increment
print "$s\n";
$s += 1; # use $s in a numeric context
print "$s\n";
```

打印时，此脚本给出下列输出：

```
00010 Okay
00011 Okay
12 Oops!
```

Perl 的 ‘++’ 自动增量操作是很灵巧的而且可以利用串或数值建立序列值，但 ‘+=’ 操作只应用于数值。在所显示的输出中，可看到 ‘+=’ 引起串到数值的转换并且丢失了 \$s 值中的前导零。

序列不从1开始的另一个原因从技术的角度来说可能不值一提。例如，在分配会员号时，序列号不要从1开始，以免出现关于谁是第一号的政治争论。

(4) 不用 AUTO_INCREMENT 生成序列

生成序列号的另一个方法根本就不需要使用 AUTO_INCREMENT 列。它利用取一个参数的 LAST_INSERT_ID() 函数的变量来生成序列号。(这种形式在 MySQL 3.22.9. 中引入) 如果利用 LAST_INSERT_ID(expr) 来插入或更新一个列，则下一次不用参数调用 LAST_INSERT_ID() 时，将返回 expr 的值。换句话说，就像由 AUTO_INCREMENT 机制生成的那样对 expr 进行处理。这样使得能生成一个序列号，然后可在以后的客户会话中利用它，用不着取受其他客户机影响的值。

利用这种策略的一种方法是创建一个包含一个值的单行表，该值在想得到序列中下一个值时进行更新。例如，可创建如下的表：

```
CREATE TABLE seq_table (seq INT UNSIGNED NOT NULL)
INSERT INTO seq_table VALUES(0)
```

上面的语句创建了表 seq_table 并用包含 seq 值 0 的行对其进行初始化。可利用这个表产生下一个序列号，如下所示：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+1)
```

该语句取出 seq 列的当前值并对其加 1，产生序列中的下一个值。利用 LAST_INSERT_ID(seq + 1) 生成新值使它就像一个 AUTO_INCREMENT 值一样，而且此值可在以后的语句中通过调用无参数的 LAST_INSERT_ID() 来取出。即使某个其他客户机同时生成了另一个序列号，上述作用也不会改变，因为 LAST_INSERT_ID() 是客户机专用的。

如果希望生成增量不是 1 的编号序列或负增量的编号序列，也可以利用这个方法。例如，下面两个语句可以用来分别生成一个增量为 100 的编号序列和一个负的编号序列：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+100)
UPDATE seq_table SET seq = LAST_INSERT_ID(seq-1)
```

通过将 seq 列设置为相应的初始值，可利用这个方法生成以任意值开始的序列。

关于将此序列生成方法用于多个计数器的应用，可参阅第 3 章。

2.2.3 串列类型

MySQL 提供了几种存放字符数据的串类型。串常常用于如下这样的值：

```
"N. Bertram, et al."
"Pencils (no. 2 lead)"
"123 Elm St."
"Monograph Series IX"
```

在某种意义上，串实际是一种“通用”类型，因为可用它们来表示任意值。例如，可用串类型来存储二进制数据，如影像或声音，或者存储 gzip 的输出结果，即存储压缩数据。

对于所有串类型，都要剪裁过长的值使其适合于相应的串类型。但是串类型的取值范围很不同，有的取值范围很小，有的则很大。取值大的串类型能够存储近 4GB 的数据。因此，应该使串足够长以免您的信息被切断（由于受客户机 / 服务器通信协议的最大块尺寸限制，列值的最大限额为 24MB）。

表2-8给出了 MySQL 定义串值列的类型，以及每种类型的最大尺寸和存储需求。对于可变长的列类型，各行的值所占的存储量是不同的，这取决于实际存放在列中的值的长度。这个长度在表中用 L 表示。

表2-8 串列类型

类型说明	最大尺寸	存储需求
CHAR(M)	M 字节	M 字节
VARCHAR(M)	M 字节	L + 1 字节
TINYBLOB, TINYTEXT	2 ⁸ - 1 字节	L + 1 字节
BLOB, TEXT	2 ¹⁶ - 1 字节	L + 2 字节
MEDIUMBLOB, MEDIUMTEXT	2 ²⁴ - 1 字节	L + 3 字节
LONGBLOB, LONGTEXT	2 ³² - 1 字节	L + 4 字节
ENUM(“value1”, “value2”, ...)	65535 个成员	1 或 2 字节
SET(“value1”, “value2”, ...)	64 个成员	1、2、3、4 或 8 字节

L 以外所需的额外字节为存放该值的长度所需的字节数。MySQL 通过存储值的内容及其长度来处理可变长度的值。这些额外的字节是无符号整数。请注意，可变长类型的最大长度、此类型所需的额外字节数以及占用相同字节数的无符号整数之间的对应关系。例如，MEDIUMBLOB 值可能最多 2²⁴ - 1 字节长并需要 3 个字节记录其结果。3 个字节的整数类型 MEDIUMINT 的最大无符号值为 2²⁴ - 1。这并非偶然。

1. CHAR 和 VARCHAR 列类型

CHAR 和 VARCHAR 是最常使用的串类型。它们是有差异的，CHAR 是定长类型而 VARCHAR 是可变长类型。CHAR(M) 列中的每个值占 M 个字节；短于 M 个字节的值存储时在右边加空格（但右边的空格在检索时去掉）。VARCHAR(M) 列的值只用所必需的字节数来存放（结尾的空格在存储时去掉，这与 ANSI SQL 的 VARCHAR 值的标准不同），然后再加一个字节记录其长度。

如果所需的值在长度上变化不大，则 CHAR 是一种比 VARCHAR 好的选择，因为处理行长度固定的表比处理行长度可变的表的效率更高。如果所有的值长度相同，由于需要额外的字节来记录值的长度，VARCHAR 实际占用了更多的空间。

在 MySQL 3.23 以前，CHAR 和 VARCHAR 列用最大长度为 1 到 255 的 M 来定义。从 MySQL 3.23 开始，CHAR(0) 也是合法的了。在希望定义一个列，但由于尚不知道其长度，所以不想给其分配空间的情况下，CHAR(0) 列作为占位符很有用处。以后可以用 ALTER TABLE 来加宽这个列。如果允许其为 NULL，则 CHAR(0) 列也可以用来表示 on/off 值。这

样的列可能取两个值，NULL 和空串。CHAR(0) 列在表中所占的空间很小，只占一位。

除少数情况外，在同一个表中不能混用 CHAR 和 VARCHAR。MySQL 根据情况甚至会

将列从一种类型转换为另一种类型。这样做的原因如下：

- 行定长的表比行可变长的表容易处理（其理由请参阅 2.3 节“选择列的类型”）。
- 表行只在表中所有行为定长类型时是定长的。即使表中只有一列是可变长的，该表的行也是可变长的。
- 因为在行可变长时定长行的性能优点完全失去。所以为了节省存储空间，在这种情况下最好也将定长列转换为可变长列。

这表示，如果表中有 VARCHAR 列，那么表中不可能同时有 CHAR 列；MySQL 会自动地将它们转换为 VARCHAR 列。例如创建如下一个表：

```
CREATE TABLE my_table
(
  c1 CHAR(10),
  c2 VARCHAR(10)
)
```

如果使用 DESCRIBE my_table 查询，则其输出如下：

Field	Type	Null	Key	Default	Extra
c1	varchar(10)	YES		NULL	
c2	varchar(10)	YES		NULL	

请注意，VARCHAR 列的出现使 MySQL 将 c1 也转换成了 VARCHAR 类型。如果试图用 ALTER TABLE 将 c1 转换为 CHAR，将不起作用。将 VARCHAR 列转换为 CHAR 的惟一办法是同时转换表中所有 VARCHAR 列：

```
ALTER TABLE my_table MODIFY c1 CHAR(10), MODIFY c2 CHAR(10)
```

BLOB 和 TEXT 列类型像 VARCHAR 一样是可变长的，但是它们没有定长的等价类型，因此不能在表中与 BLOB 或 TEXT 列一起使用 CHAR 列。这时任何 CHAR 列都将被转换为 VARCHAR 列。

定长与可变长列混用的情形是在 CHAR 列短于 4 个字符时，可以不对其进行转换。例如，MySQL 不会将下面所创建的表中的 CHAR 列转换为 VARCHAR 列：

```
CREATE TABLE my_table
(
  c1 CHAR(2),
  c2 VARCHAR(10)
)
```

短于4个字符的列不转换的原因是，平均情况下，不存储尾空格所节省的空间被 VARCHAR 列中记录每个值的长度所需的额外字节所抵消了。实际上，如果所有列都短，MySQL 将会把所定义的所有列从 VARCHAR 转换为 CHAR。MySQL 这样做的原因是，这种转换平均来说不会增加存储需求，而且使表行定长，从而改善了性能。如果按如下创建一个表，VARCHAR 列全都会转换为 CHAR 列：

```
CREATE TABLE my_table
(
  c1 VARCHAR(1),
  c2 VARCHAR(2),
  c3 VARCHAR(3)
)
```

查看 DESCRIBE my_table 的输出可看到转换结果如下：

Field	Type	Null	Key	Default	Extra
c1	char(1)	YES		NULL	
c2	char(2)	YES		NULL	
c3	char(3)	YES		NULL	

2. BLOB 与 TEXT 列类型

BLOB 是一个二进制大对象，是一个可以存储大量数据的容器，可以使其任意大。在 MySQL 中，BLOB 类型实际是一个类型系列（TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB），除了在可以存储的最大信息量上不同外（请参阅表 2-8），它们是等同的。MySQL 还有一个 TEXT 类型系列（TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT）。除了用于比较和排序外，它们在各个方面都与相应的 BLOB 类型等同，BLOB 值是区分大小写的，而 TEXT 值不区分大小写。BLOB 和 TEXT 列对于存储可能有很大增长的值或各行大小有很大变化的值很有用，例如，字处理文档、图像和声音、混合数据以及新闻文章等等。

BLOB 或 TEXT 列在 MySQL 3.23 以上版本中可以进行索引，虽然在索引时必须指定一个用于索引的约束尺寸，以免建立出很大的索引项从而抵消索引所带来的好处。除此之外，一般不通过查找 BLOB 或 TEXT 列来进行搜索，因为这样的列常常包含二进制数据（如图像）。常见的做法是用表中另外的列来记录有关 BLOB 或 TEXT 值的某种标识信息，并用这些信息来确定想要哪些行。

使用 BLOB 和 TEXT 列需要特别注意以下几点：

- 由于 BLOB 和 TEXT 值的大小变化很大，如果进行的删除和更新很多，则存储它们的表出现高碎片率会很高。应该定期地运行 OPTIMIZE TABLE 减少碎片率以保持良好的性能。要了解更详细的信息请参阅第 4 章。
- 如果使用非常大的值，可能会需要调整服务器增加 max_allowed_packet 参数的值。详细的信息请参阅第 11 章“常规的 MySQL 管理”。如果需要增加希望使用非常大的值的客户机的块尺寸，可见附录 E“MySQL 程序参考”，该附录介绍了怎样对 mysql 和 mysqldump 客户机进行这种块尺寸的增加。

3. ENUM 和 SET 列类型

ENUM 和 SET 是一种特殊的串类型，其列值必须从一个固定的串集中选择。它们之间的主要差别是 ENUM 列值必须确实是值集中的一个成员，而 SET 列值可以包括集合中任意或所有的成员。换句话说，ENUM 用于互相排斥的值，而 SET 列可以从一个值的列表中选择多个值。

ENUM 列类型定义了一个枚举。可赋予 ENUM 列一个在创建表时指定的值列表中的选择的成员。枚举可具有最多 65 536 个成员（其中之一为 MySQL 保留）。枚举通常用来表示类别值。例如，定义为 ENUM(“N”，“Y”) 的列中的值可以是“N”或“Y”。或者可将 ENUM 用于诸如调查或问卷中的多项选择问题，或用于某个产品的可能尺寸或颜色等：

```
employees ENUM("less than 100","100-500","501-1500","more than 1500")
color ENUM("red","green","blue","black")
size ENUM("S","M","L","XL","XXL")
```

如果正在处理 Web 页中的选择, 那么可以利用 ENUM 来表示站点访问者在某页上的互相排斥的单选钮集合中进行的選擇。例如, 如果运行一个在线比萨饼订购服务系统, 可用 ENUM 来表示顾客订购的比萨饼形状:

```
crust ENUM("thin","regular","pan style")
```

如果枚举类别表示计数, 在建立该枚举时最重要的是选择合适的类别。例如, 在记录实验室检验中白血球的数目时, 可能会将计数分为如下的几组:

```
wbc ENUM("0-100","101-300",>300")
```

在某个测试结果以精确的计数到达时, 要根据该值所属的类别来记录它。但如果想将列从基于类别的 ENUM 转换为基于精确计数的整数时, 不可能恢复原来的计数。

在创建 SET 列时, 要指定一个合法的集合成员列表。在这种意义上, SET 类型与 ENUM 是类似的。但是 SET 与 ENUM 不同, 每个列值可由来自集合中任意数目的成员组成。集合中最多可有 64 个成员。对于值之间互斥的固定集合, 可使用 SET 列类型。例如, 可利用 SET 来表示汽车的可用选项, 如下所示:

```
SET("luggage rack","cruise control","air conditioning","sun roof")
```

然后, 特定的 SET 值将表示顾客实际订购哪些选项, 如下所示:

```
SET("cruise control,sun roof")
SET("luggage rack,air conditioning")
SET("luggage rack,cruise control,air conditioning")
SET("air conditioning")
SET("")
```

空串表示顾客未订购任何选项。这是一个合法的 SET 值。

SET 列值为单个串。如果某个值由多个集合成员组成, 那么这些成员在串中用逗号分隔。显然, 这表示不应该用含有逗号的串作为 SET 成员。

SET 列的其他用途是表示诸如病人的诊断或来自 Web 页的选择结果这样的信息。对于诊断, 可能会有一个向病人提问的标准症状清单, 而病人可能会表现出某些症状或所有的症状。对于在线比萨饼服务系统, 用于订购的 Web 页应该具有一组复选框, 用来表示顾客想在比萨饼上加的配料。

对 ENUM 或 SET 列的合法值列表的定义很重要, 例如:

- 正如上面所介绍的, 此列表决定了列的可能合法值。
- 可按任意的大小写字符插入 ENUM 或 SET 值, 但是列定义中指定的串的大小写字符决定了以后检索它们时的大小写。例如, 如果有一个 ENUM("Y", "N") 列, 但您在其中存储了 "y" 和 "n", 当您检索出它们时显示的是 "Y" 和 "N"。这并不影响比较或排序的状态, 因为 ENUM 和 SET 列是不区分大小写的。
- 在 ENUM 定义中的值顺序就是排序顺序。SET 定义中的值顺序也决定了排序顺序, 但是这个关系更为复杂, 因为列值可能包括多个集合成员。
- SET 定义中的值顺序决定了在显示由多个集合成员组成的 SET 列值时, 子串出现的顺序。

ENUM 和 SET 被归为串类型是由于在建立这些类型的列时, 枚举和集合成员被指定为串。但是, 这些成员在内部存放时作为数值, 而且同样可作为数值来处理。这表示 ENUM 和 SET 类型比其他的串类型更为有效, 因为通常可用数值运算而不是串运算来处理它们。而且这还表示 ENUM 和 SET 值可用在串或数值的环境中。

列定义中的 ENUM 成员是从 1 开始顺序编号的。(0 被 MySQL 用作错误成员, 如果以串的形式表示就是空串。) 枚举值的数目决定了 ENUM 列的存储大小。一个字节可表示 256 个值, 两个字节可表示 65 536 个值。(可将其与一字节和两字节的整数类型 TINYINT、UNSIGNED 和 SMALLINT UNSIGNED 进行对比。) 因此, 枚举成员的最大数目为 65 536 (包括错误成员), 并且存储大小依赖于成员数目是否多于 256 个。在 ENUM 定义中, 可以最多指定 65 535 (而不是 65 536) 个成员, 因为 MySQL 保留了一个错误成员, 它是每个枚举的隐含成员。在将一个非法值赋给 ENUM 列时, MySQL 自动将其换成错误成员。

下面有一个例子, 可用 mysql 客户机程序测试一下。它给出枚举成员的数值顺序, 而且还说明了 NULL 值无顺序编号:

```
mysql> CREATE TABLE e_table (e ENUM("jane","fred","will","marcia"));
mysql> INSERT INTO e_table
VALUES("jane"),("fred"),("will"),("marcia"),(""),(NULL);
mysql> SELECT e, e+0, e+1, e*3 FROM e_table;
```

e	e+0	e+1	e*3
jane	1	2	3
fred	2	3	6
will	3	4	9
marcia	4	5	12
	0	1	0
NULL	NULL	NULL	NULL

可对 ENUM 成员按名或者按编号进行运算, 例如:

```
mysql> SELECT e FROM e_table WHERE e="will";
```

e
will

```
mysql> SELECT e FROM e_table WHERE e=3;
```

e
will

可以定义空串为一个合法的枚举成员。与列在定义中的其他成员一样, 它将被赋予一个非零的数值。但是使用空串可能会引起某些混淆, 因为该串也被作为数值为 0 的错误成员。在下面的例子中, 将非法的枚举值 “x” 赋予 ENUM 列引起了错误成员的赋值。仅在以数值形式进行检索时, 才能够与空串区分开:

```
mysql> CREATE TABLE t (e ENUM("a","", "b"));
mysql> INSERT INTO t VALUES("a"),(""),("b"),("x");
mysql> SELECT e, e+0 FROM t;
```

e	e+0
a	1
	2
b	3
	0

SET 列的数值表示与 ENUM 列的表示有所不同，集成员不是顺序编号的。每个成员对应 SET 值中的一个二进制位。第一个集成员对应于 0 位，第二个成员对应于 1 位，如此等等。数值 SET 值 0 对应于空串。SET 成员以位值保存。每个字节的 8 个集合值可按此方式存放，因此 SET 列的存储大小是由集成员的数目决定的，最多 64 个成员。对于大小为 1 到 8、9 到 16、17 到 24、25 到 32、33 到 64 个成员的集合，其 SET 值分别占用 1、2、3、4 或 8 个字节。

用一组二进制位来表示 SET 正是允许 SET 值由多个集成员组成的原因。值中二进制位的任意组合都可以得到，因此，相应的值可由对应于这些二进制位的 SET 定义中的串组合构成。

下面给出一个说明 SET 列的串形式与数值形式之间关系的样例；数值以十进制形式和二进制形式分别给出：

```
mysql> CREATE TABLE s_table (s SET("jane","fred","will","marcia");
mysql> INSERT INTO s_table
VALUES("jane"),("fred"),("will"),("marcia"),(""),(NULL);
mysql> SELECT s, s+0, BIN(s+0) FROM s_table;
```

s	s+0	BIN(s+0)
jane	1	1
fred	2	10
will	4	100
marcia	8	1000
	0	0
NULL	NULL	NULL

如果给 SET 列赋予一个含有未作为集成员列出的子串的值，那么这些子串被删除，并将包含其余子串的值赋予该列。在赋值给 SET 列时，子串不需要按定义该列时的顺序给出。但是，在以后检索该值时，各成员将按定义时的顺序列出。假如用下面的定义定义一个 SET 列来表示家具：

```
SET("table","lamp","chair")
```

如果给这个列赋予“ chair, couch, table ”值，那么，“ couch ”被放弃，因为它不是集合的成员。其次，以后检索这个值时，显示为“ table, chair ”。之所以这样是因为 MySQL 针对所赋的值的每个子串决定各个二进制位并在存储值时将它们置为 1。“ couch ”不对应二进制位，则忽略。在检索时，MySQL 按顺序扫描各二进制位，通过数值值构造出串值，它自动地将子串排成定义列时给出的顺序。这个举动还表示，如果在一个值中不止一次地指定某个成员，但在检索时它也只会出现一次。如果将“ lamp, lamp, lamp ”赋予某个 SET 列，检索时也只会得出“ lamp ”。

MySQL 重新对 SET 值中的成员进行排序这个事实表示，如果用一个串来搜索值，则必须以正确的顺序列出各成员。如果插入“ chair, table ”，然后搜索“ chair, table ”，那么将找不到相应的记录；必须查找“ table, chair ”才能找到。

ENUM 和 SET 列的排序和索引是根据列值的内部值（数值值）进行的。下面的例子可能会显示不正确，因为各个值并不是按字母顺序存储的：

```
mysql> SELECT e FROM e_table ORDER BY e;
```

```

+-----+
| e      |
+-----+
| NULL   |
+-----+
| jane   |
| fred   |
| will   |
| marcia |
+-----+

```

NULL 值排在其他值前（如果是降序，将排在其他值之后）。

如果有一个固定的值集，并且希望按特殊的次序进行排序，可利用 ENUM 的排序顺序。在创建表时做一个 ENUM 列，并在该列的定义中以所想要的次序给出各枚举值即可。

如果希望 ENUM 按正常的字典顺序排序，可使用 CONCAT() 和排序结果将列转换成一个非 ENUM 串，如下所示：

```
mysql> SELECT CONCAT(e) as e_str FROM e_table ORDER BY e_str;
```

```

+-----+
| e_str |
+-----+
| NULL  |
+-----+
| fred  |
| jane  |
| marcia|
| will  |
+-----+

```

4. 串列类型属性

可对 CHAR 和 VARCHAR 类型指定 BINARY 属性使列值作为二进制串处理（即，在比较和排序操作区分大小写）。

可对任何串类型指定通用属性 NULL 和 NOT NULL。如果两者都不指定，缺省值为 NULL。但是定义某个串列为 NOT NULL 并不阻止其取空串。空值不同于遗漏的值，因此，不要错误地认为可以通过定义 NOT NULL 来强制某个串列只包含非空的值。如果要求串值非空，那么这是一个在应用程序中必须强制实施的约束条件。

还可以对除 BLOB 和 TEXT 类型外的所有串列类型用 DEFAULT 属性指定一个缺省值。如果不指定缺省值，MySQL 会自动选择一个。对于可以包含 NULL 的列，其缺省值为 NULL。对于不能包含 NULL 的列，除 ENUM 列外都为空串，在 ENUM 列中，缺省值为第一个枚举成员（对于 SET 类型，在相应的列不能包含 NULL 时其缺省值实际上是空集，不过这里空集等价于空串）。

2.2.4 日期和时间列类型

MySQL 提供了几种时间值的列类型，它们分别是：DATE、DATETIME、TIME、TIMESTAMP 和 YEAR。表2-9 给出了 MySQL 为定义存储日期和时间值所提供的这些类型，并给出了每种类型的合法取值范围。YEAR 类型是在 MySQL 3.22版本中引入的。其他类型在所有 MySQL 版本中都可用。每种时间类型的存储需求见表 2-10。

每个日期和时间类型都有一个“零”值，在插入该类型的一个非法值时替换成此值，见表2-11。这个值也是定义为 NOT NULL 的日期和时间列的缺省值。

表2-9 日期和时间列类型

类型说明	取值范围
DATE	“1000 - 01 - 01”到“9999 - 12 - 31”
TIME	“- 838:59:59”到“838:59:59”
DATETIME	“1000 - 01 - 01 00:00:00”到“9999 - 12 - 31 23:59:59”
TIMESTAMP[(M)]	19700101000000 到 2037 年的某个时刻
YEAR[(M)]	1901 到 2155

表2-10 日期和时间列类型的存储需求

类型说明	存储需求
DATE	3 字节 (MySQL 3.22 以前为 4 字节)
TIME	3 字节
DATETIME	8 字节
TIMESTAMP	4 字节
YEAR	1 字节

MySQL 表示日期时根据 ANSI 规范首先给出年份。例如，1999 年 12 月 3 日表示为“1999-12-03”。MySQL 允许在输入日期时有某些活动的余地。如能将两个数字的年份转换成四位数字的年份，而且在输入小于 10 的月份和日期时不用输入前面的那位数字。但是必须首先给出年份。平常经常使用的那些格式，如“12/3/99”或“3/12/99”，都是不正确的。MySQL 使用的日期表示规则请参阅“处理日期和时间列”小节。

表2-11 日期和时间列类型的“零”值

类型说明	零 值
DATE	“0000 - 00 - 00”
TIME	“00:00:00”
DATETIME	“0000- 00 - 00 00:00:00”
TIMESTAMP	0000000000000000
YEAR	0000

时间值按本地时区返回给服务器；MySQL 对返回给客户机的值不作任何时区调整。

1. DATE、TIME 和 DATETIME 列类型

DATE、TIME 和 DATETIME 类型存储日期、时间以及日期和时间值的组合。其格式为“YYYY - MM - DD”、“hh:mm:ss”和“YYYY - MM - DD hh:mm:ss”。对于 DATETIME 类型，日期和时间部分都需要；如果将 DATE 值赋给 DATETIME 列，MySQL 会自动地追加一个为“00:00:00”的时间部分。

MySQL 对 DATETIME 和 TIME 表示的时间在处理上稍有不同。对于 DATETIME，时间部分表示某天的时间。而 TIME 值表示占用的时间（这也就是为什么其取值范围如此之大而且允许取负值的原因）。用 TIME 值的最右边部分表示秒，因此，如果插入一个“短”（不完全）的时间值，如“12:30”到 TIME 列，则存储的值为“00:12:30”，即被认为是“12 分 30 秒”。如果愿意，也可用 TIME 列来表示天的时间，但是要记住这个转换规则以免出问题。为了插入一个“12 小时 30 分钟”的值，必须将其表示为“12:30:00”。

2. TIMESTAMP 列类型

TIMESTAMP 列以 YYYYMMDDhhmmss 的格式表示值，其取值范围从 19700101000000 到 2037 年的某个时间。此取值范围与 UNIX 的时间相联系，在 UNIX 的时间中，1970 年的第一天为“零天”，也就是所谓的“新纪元”。因此 1970 年的开始决定了 TIMESTAMP 取值范围的低端。其取值范围的上端对应于 UNIX 时间上的四字节界限，它可以表示到 2037 年的

值。(TIMESTAMP 值的上限将会随着操作系统为扩充 UNIX 的时间值所进行的修改而增加。这是在系统库一级必须提及的。MySQL 也将利用这些更改。)

TIMESTAMP 类型之所以得到这样的名称是因为它在创建或修改某个记录时，有特殊的记录作用。如果在一个 TIMESTAMP 列中插入 NULL，则该列值将自动设置为当前的日期和时间。在建立或更新一行但不明确给 TIMESTAMP 列赋值时也会自动设置该列的值为当前的日期和时间。但是，仅行中的第一个 TIMESTAMP 列按此方式处理，即使是行中第一个 TIMESTAMP 列，也可以通过插入一个明确的日期和时间值到该列（而不是 NULL）使该处理失效。

TIMESTAMP 列的定义可包含对最大显示宽度 M 的说明。表 2-12 给出了所允许的 M 值的显示格式。如果 TIMESTAMP 定义中省略了 M 或者其值为 0 或大于 14，则该列按 TIMESTAMP(14) 处理。取值范围从 1 到 13 的 M 奇数值作为下一个更大的偶数值处理。

TIMESTAMP 列的显示宽度与存储大小或存储在内部的值无关。TIMESTAMP 值总是以 4 字节存放并按 14 位精度进行计算，

表2-12 TIMESTAMP 显示格式

类型说明	显示格式
TIMESTAMP(14)	YYYYMMDDhhmmss
TIMESTAMP(12)	YYYYMMDDhhmm
TIMESTAMP(10)	YYMMDDhhmm
TIMESTAMP(8)	YYYYMMDD
TIMESTAMP(6)	YYMMDD
TIMESTAMP(4)	YYMM
TIMESTAMP(2)	YY

与显示宽度无关。为了明白这一点，按如下定义一个表，然后插入一些行，进行检索：

```
CREATE TABLE my_table
(
  ts TIMESTAMP(8),
  i INT
)
INSERT INTO my_table VALUES(19990801120000,3)
INSERT INTO my_table VALUES(19990801120001,2)
INSERT INTO my_table VALUES(19990801120002,1)
INSERT INTO my_table VALUES(19990801120003,0)
SELECT * FROM my_table ORDER BY ts, i
```

该 SELECT 语句的输出如下：

```
+-----+-----+
| ts      | i      |
+-----+-----+
| 19990801 | 3      |
| 19990801 | 2      |
| 19990801 | 1      |
| 19990801 | 0      |
+-----+-----+
```

从表面上看，出现的行排序有误，第一列中的值全都相同，所以似乎排序是根据第二列中的值进行的。这个表面反常的结果是由于事实上，MySQL 是根据插入 TIMESTAMP 列的全部 14 位值进行排序的。

MySQL 没有可在记录建立时设置为当前日期和时间、并从此以后保持不变的列类型。如果要实现这一点，可用两种方法来完成：

- 使用 TIMESTAMP 列。在最初建立一个记录时，设置该列为 NULL，将其初始化为当前日期和时间：

```
INSERT INTO tbl_name (ts_col, ...) VALUES(NULL, ...)
```

在以后无论何时更改此记录，都要明确地设置此列为其原有的值。赋予一个明确

的值使时间戳机制失效，因为它阻止了该列的值自动更新：

```
UPDATE tbl_name SET ts_col=ts_col WHERE ...
```

■ 使用 DATETIME 列。在建立记录时，将该列的值初始化为 NOW()：

```
INSERT INTO tbl_name (dt_col, ...) VALUES(NOW(), ...)
```

无论以后何时更新此记录，都不能动该列：

```
UPDATE tbl_name SET /* anything BUT dt_col here */ WHERE ...
```

如果想利用 TIMESTAMP 列既保存建立的时间值又保存最后修改的时间值，那么可用一个 TIMESTAMP 列来保存修改时间值，用另一个 TIMESTAMP 列保存建立时间值。要保证保存修改时间值的列为第一个 TIMESTAMP，从而在记录建立或更改时自动对其进行设置。使保存建立时间值的列为第二个 TIMESTAMP，并在建立新记录时将其初始化为 NOW()。这样第二个 TIMESTAMP 的值将反映记录建立时间，而且以后将不再更改。

3. YEAR 列类型

YEAR 是一个用来有效地表示年份值的 1 个字节的列类型。其取值范围为从 1901 到 2155。在想保存日期信息但又只需要日期的年份时可使用 YEAR 类型，如出生年份、政府机关选举年份等等。在不需要完全的日期值时，YEAR 比其他日期类型在空间利用上更为有效。

YEAR 列的定义可包括显示宽度 M 的说明，显示宽度应该为 4 或 2。如果 YEAR 定义中省略了 M，其缺省值为 4。

TINYINT 与 YEAR 具有相同的存储大小（一个字节），但取值范围不同。要使用一个整数类型且覆盖与 YEAR 相同的取值范围，可能需要 SMALLINT 类型，此类型要占两倍的空间。在所要表示的年份取值范围与 YEAR 类型的取值范围相同的情况下，YEAR 的空间利用率比 SMALLINT 更为有效。YEAR 相对整数列的另一个优点是 MySQL 将会利用 MySQL 的年份推测规则把 2 位值转换为 4 位值。例如，97 与 14 将转换为 1997 和 2014。但要认识到，插入数值 00 将得到 0000 而不是 2000。如果希望零值转换为 2000，必须指定其为串“00”。

4. 日期和时间列类型的属性

没有专门针对日期和时间列类型的属性。通用属性 NULL 和 NOT NULL 可用于任意日期和时间类型。如果 NULL 和 NOT NULL 两者都不指定，则缺省值为 NULL。也可以用 DEFAULT 属性指定一个缺省值。如果不指定缺省值，将自动选择一个缺省值。含有 NULL 的列的缺省值为 NULL。否则，缺省值为该类型的“零”值。

5. 处理日期和时间列

MySQL 可以理解各种格式的日期和时间值。DATE 值可按后面的任何一种格式指定，其中包括串和数值

表2-13 日期和时间类型的输入格式

类 型	允许的格式
DATETIME, TIMESTAMP	“YYYY - MM - DD hh:mm:ss”
	“YY - MM - DD hh:mm:ss”
DATE	“YYYYMMDDhhmmss”
	“YYMMDDhhmmss”
	YYYYMMDDhhmmss
	YYMMDDhhmmss
	“YYYY - MM - DD”
	“YY - MM - DD”
	“YYYYMMDD”
	“YYMMDD”
	YYYYMMDD
	YYMMDD
TIME	“hh:mm:ss”
	“hhmmss”
YEAR	hhmmss
	“YYYY”
	“YY”
	YYYY
	YY

形式。表2-13为每种日期和时间类型所允许的格式。

两位数字的年度值的格式用“歧义年份值的解释”中所描述的规则来解释。对于有分隔符的串格式，不一定非要用日期的“-”符号和时间的“:”符号来分隔，任何标点符号都可用作分隔符，因为值的解释取决于上下文，而不是取决于分隔符。例如，虽然时间一般是用分隔符“:”指定的，但 MySQL 并不会在一个需要日期的上下文中将含有“:”号的值理解成时间。此外，对于有分隔符的串格式，不需要为小于 10 的月、日、小时、分钟或秒值指定两个数值。下列值是完全等同的：

```
"2012-02-03 05:04:09"
"2012-2-03 05:04:09"
"2012-2-3 05:04:09"
"2012-2-3 5:04:09"
"2012-2-3 5:4:09"
"2012-2-3 5:4:9"
```

请注意，有前导零的值根据它们被指定为串或数有不同的解释。串“001231”将视为一个六位数字的值并解释为 DATE 的“2000-12-31”和 DATETIME 的“2000-12-31 00:00:00”。而数 001231 被认为 1231，这样的解释就有问题了。这种情形最好使用串值，或者如果要使用数值的话，应该用完全限定的值（即，DATE 用 20001231，DATETIME 用 200012310000）。

通常，在 DATE、DATETIME 和 TIMESTAMP 类型之间可以自由地赋值，但是应该记住以下一些限制：

- 如果将 DATETIME 或 TIMESTAMP 值赋给 DATE，则时间部分被删除。
- 如果将 DATE 值赋给 DATETIME 或 TIMESTAMP，结果值的时间部分被设置为零。
- 各种类型具有不同的取值范围。TIMESTAMP 的取值范围更受限制（1970 到 2037），因此，比方说，不能将 1970 年以前的 DATETIME 值赋给 TIMESTAMP 并得到合理的结果。也不能将 2037 以后的值赋给 TIMESTAMP。

MySQL 提供了许多处理日期和时间值的函数。要了解更详细的信息请参阅附录 C。

6. 歧义年份值的理解

对于所有包括年份部分的日期和时间类型（DATE、DATETIME、TIMESTAMP、YEAR），MySQL 将两位数字的年份转换为四位数字的年份。这个转换根据下列规则进行（在 MySQL 4.0 中，这些规则稍有改动，其中 69 将转换为 1969 而不是 2069。这是根据 X/Open UNIX 标准规定的规则作出的改动）：

- 00 到 69 的年份值转换为 2000 到 2069。
- 70 到 99 的年份值转换为 1970 到 1999。

通过将不同的两位数字值赋给一个 YEAR 列然后进行检索，可很容易地看到这些规则的效果。下面是检索程序：

```
mysql> CREATE TABLE y_table (y YEAR);
mysql> INSERT INTO y_table VALUES(68),(69),(99),(00);
mysql> SELECT * FROM y_table;
+-----+
| y      |
+-----+
| 2068   |
| 1969   |
| 1999   |
| 0000   |
+-----+
```


请注意，00 转换为 0000 而不是 2000。这是因为 0 是 YEAR 类型的一个完全合法的值；如果插入一个数值，得到的就是这个结果。要得到 2000，应该插入串“0”或“00”。可通过 CONCAT() 插入 YEAR 值来保证 MySQL 得到一个串而不是数。CONCAT() 函数不管其参数是串或数值，都返回一个串结果。

请记住，将两位数字的年份值转换为四位数字的年份值的规则只产生一种结果。在未给定世纪的情况下，MySQL 没有办法肯定两位数字的年份的含义。如果 MySQL 的转换规则不能得出您所希望的值，解决的方法很简单：即用四位数字输入年份值。

MySQL 有千年虫问题吗？

MySQL 自身是没有 2000 年问题的，因为它在内部是按四位数年份存储日期值的，并且由用户负责提供恰当的日期值。两位数字年份解释的实际问题不是 MySQL 带来的，而是由于有的人想省事，输入歧义数据所引起的问题。如果您愿意冒险，可以继续这样做。在您冒险的时候，MySQL 的猜测规则是可以使用的。但要意识到，很多时候您确实需要输入四位数字的年份。例如，president 表列出了 1700 年以来的美国总统，所以在此表中录入出生与死亡日期需要四位的年份值。这些列中的年份值跨了好几个世纪，因此，让 MySQL 从两位数字的年份去猜测是哪个世纪是不可能的。

2.3 选择列的类型

上一节描述了各种可供选择的 MySQL 的列类型及其属性，以及它们可存储的各种值，所占用的存储空间等等。但是在实际创建一个表时怎样决定用哪些类型呢？本节讨论在做出决定前应考虑的各种因素。

最“常用”的列类型是串类型。可将任何数据存储为串，因为数和日期都可以串的形式表示。但是为什么不将所有列都定义为串从而结束这里的讨论呢？让我们来看一个简单的例子。假定有一些看起来像数的值。可将它们表示为串，但应该这样做吗？这样做会发生什么事？

有一桩事不可避免，那就是可能要使用更多的空间，因为较串来说，数的存储更为有效。我们可能已经注意到，由于数和串处理方式的不同，查询结果也有所不同。例如，数的排序与串的排序就有所不同。数 2 小于数 11，但串“2”按字典顺序大于“11”。可用如下数值内容的列来搞清这个问题：

```
SELECT col_name + 0 as num ... ORDER BY num
```

将零加到该列强制得出一个数值，但是这样合理吗？一般可能不合理。将该列作为数而不是串具有几个重要的含义。它对每个列值实施串到数的转换，这是低效的。而且将该列的值转换为计算结果妨碍 MySQL 使用该列上的索引，降低了以后的查询速度。如果这些值一开始就是作为数值存储的，那么这些性能上的降低都不会出现。采用一种表示而不用另一种的简单选择实际上并不简单，它在存储需求、查询效率以及处理性能等方面都会产生重要的影响。

前面的例子说明，在选择列类型时，有以下几个问题需要考虑：

- 列中存储何种类型的值？这是一个显而易见的问题，但必须确定。可将任何类型的值表示为串，尤其当对数值使用更为合适的类型可能得到更好的性能时（日期和时间值

也是这样)。可见，对要处理的值的类型进行评估不一定是件微不足道的事，特别在数据是别人的数据时更是如此。如果正在为其他人建立一个表，搞清列中要存储的值的类型极为重要，必须提足够多的问题以便得到作出决定的充足的信息。

- 列值有特定的取值范围吗？如果它们是整数，它们总是非负值吗？如果这样，可采用 UNSIGNED 类型。如果它们是串，总能从定长值集中选出它们吗？如果这样，ENUM 或 SET 是很合适的类型。

在类型的取值范围与所用的存储量之间存在折衷。需有一个多“大”的类型？对于数，如果其取值范围有限，可以选择较小的类型，对取值范围几乎无限的数，应该选择较大的类型。对于串，可以使它们短也可以使它们长，但如果希望存储的值只含不到 10 个字符，就不应该选用 CHAR(255)。

- 性能与效率问题是什么？有些类型比另外一些类型的处理效率高。数值运算一般比串的运算快。短串比长串运行更快，而且磁盘消耗更小。定长类型比可变长类型的性能更好。
- 希望对值进行什么样的比较？对于串，其比较可以是区分大小写的，也可以不区分大小写。其选择也会影响排序，因为它是基于比较的。
- 计划对列进行索引吗？如果计划对列进行索引，那么将会影响您对列类型的选择，因为有的 MySQL 版本不允许对某些类型进行索引，例如不能对 BLOB 和 TEXT 类型进行索引。而且有的 MySQL 版本要求定义索引列为 NOT NULL 的，这使您不能使用 NULL 值。

现在让我们来更详细地考虑这些问题。这里要指出的是：在创建表时，希望作出尽可能好的列类型选择，但如果所作的选择其实际并不是最佳的，这也不会带来多大的问题。可用 ALTER TABLE 将原来选择的类型转换为更好的类型。在发现数据所含的值比原设想的大时，可像将 SMALLINT 更换成 MEDIUMINT 那样简单地对类型进行更换。有时这种更换也可能很复杂，例如将 CHAR 类型更换成具有特定值集的 ENUM 类型。在 MySQL 3.23 及以后的版本中，可使用 PROCEDURE ANALYSE() 来获得表列的信息，诸如最小值和最大值以及推荐的覆盖列中值的取值范围的最佳类型。这有助于确定使用更小的类型，从而改进涉及该表的查询的性能，并减少存储该表所需的空间量。

2.3.1 列中存储何种类型的值

在决定列的类型时，首先应该考虑该列的值类型，因为这对于所选择的类型来说具有最为明显的意义。通常，在数值列中存储数，在串列中存储串，在日期和时间列中存储日期和时间。如果数值有小数部分，那么应该用浮点列类型而不是整数类型，如此等等。有时也存在例外，不可一概而论。主要是为了有意义地选择类型，应该理解所用数据的特性。如果您打算存储自己的数据，大概对如何存储它们会有自己很好的想法。但是，如果其他人请您为他们建一个表，决定列类型有时会很困难。这不像处理自己的数据那么容易。应该充分地提问，搞清表实际应该包含何种类型的值。

如果有人告诉您，某列需要记录“降雨量”。那是一个数吗？或者它“主要”是一个数值，即，一般是但不总是编码成一个数吗？例如，在看电视新闻时，气象预报一般包括降雨量。有时是一个数（如“0.25”英寸的雨量），但是有时是“微量(trace)”降雨，意思是“雨根本

就不大”。这对气象预报很合适，但在数据库中怎样存储？有可能需要将“微量”量化为一个数，以便能用数值列类型来记录降雨量，或许需要使用串，以便可以记录“微量”这个词。或者可以提出某种更为复杂的安排，使用一个数值列和一个串列，如果填充一个列就让另一个列为 NULL。很明显，可能的话，应该避免最后这种选择；最后这种选择使表难于理解，使查询更为困难。

我们一般尽量以数值形式存储所有的行，而且只为了显示的需要才对它们进行转换。例如，如果小于 0.01 英寸的非零降雨量被视为微量，那么可以如下选择列值：

```
SELECT IF(precip>0 AND precip<.01,"trace",precip) FROM ...
```

对于金钱的计算，需要处理元和分部分。这似乎像浮点值，但 FLOAT 和 DOUBLE 容易出现舍入错误，除了只需要大致精确的记录外，这些类型可能不适合。因为人们对自己的钱都是很敏感的，最好是用一种能提供完善的精确性的类型，例如：

- 将钱表示为 DECIMAL(M, 2) 类型，选择 M 为适合于所需取值范围的最大宽度。这给出具有两位小数精度的浮点值。DECIMAL 的优点是将值表示为一个串，而且不容易出现舍入错误。不利之处是串运算比内部存储为数的值上的运算效率差。
- 可在内部用整数类型来表示所有的钱值。其优点是内部用整数来计算，这样会非常快。不利之处是在输入或输出时需要利用乘或除 100 对值进行转换。

有些数据显然是数值的，但必须决定是使用浮点类型还是使用整数类型。应该搞清楚所用的单位是什么以及需要什么样的精度。整个单元的精度都够吗？或者需要表示小数的单元吗？这将有助于您在整数列和浮点数列之间进行区分。例如，如果您正表示权重，那么如果记录的值为英磅，可以使用一个整形列。如果希望记录小数部分，就应该使用浮点列。在有的情况下，甚至会使用多个字段，例如：如果希望根据磅和盎司记录权重，则可以使用多个列。

高度 (height) 是另外一种数值类型，有如下几种表示方法：

- 诸如“6 英尺 2 英寸”可表示为“6-2”这样一个串。这种形式具有容易察看和理解的优点（当然比“74 英寸更好理解”），但是这种值很难用于数学运算，如求和或取平均值。
- 一个数值字段表示英尺，另一个数值字段表示英寸。这样的表示进行数值运算相对容易，但两个字段比一个字段难于使用。
- 只用一个表示英寸的数值段。这是数据库最容易处理的方式，但是这种方式意义最不明确。不过要记住，不一定要用与您惯常使用的那种格式来表示值。可以用 MySQL 的函数将值转换为看上去意义明显的值。因此，最后这种表示方法可能是表示高度的最好方法。

如果需要存储日期信息，需要包括时间吗？即，它们永远都需要包括时间吗？MySQL 不提供具有可选时间部分的日期类型：DATE 可不包含时间，而 DATETIME 必须包含时间。如果时间确实是可选的，那么可用一个 DATE 列记录日期，一个 TIME 列记录时间。允许 TIME 列为 NULL 并解释为“无时间”：

```
CREATE TABLE my_table
(
    date DATE NOT NULL,
    time TIME NULL
)
```

在用基于日期信息的主-细目关系连接两个表时，决定是否需要时间值特别重要。

假如您正在进行一项研究，包括一些对进入您的办公室的人进行测试的题目。在一个标准的初步测试集之后，您可能在同一天进行几个额外的测试，测试的选择视初步测试结果而定。您可能会利用一个主-细目关系来表示这些信息，其中题目的标识信息和标准的初步测试存储在一个主记录中，而其他测试保存为辅助细目表的行。然后基于题目 ID 与进行测试的日期将这两个表连接到一起。

在这种情况下必须回答的问题是，是否可以只用日期，或者是否需要既使用日期又使用时间。这个问题依赖于一个题目是否可以在同一天投入测试过程不止一次。如果是这样，那么应该记录时间（比方说，记录测试过程开始的时间），或者用 DATETIME 列，或者分别用 DATE 和 TIME 列（两者都必须填写）。如果一个题目一天测试了两次，没有时间值就不能将该题目的细目记录与适当的主记录进行关联。

我曾经听过有人声称“我不需要时间；我从不同一天把一道题测试两次”。有时他们是对的，但是我也看到过这些人后来在录入同一天测试多次的题目的数据后，反过来考虑怎样防止细目记录与错误的主记录相混。很抱歉，这时已经太迟了！

有时可以在表中增加 TIME 列来处理这个问题，不幸的是，除非有某些独立的数据源，如原书面记录，否则很难整理现有记录。此外，没办法消除细目记录的歧义，以便将它们关联到合适的主记录上。即使有独立的信息源，这样做也是非常乱的，很可能使已经编写来利用表的应用程序出问题。最好是向表的拥有者说明问题并保证在创建他们的表之前进行很好的描述。

有时具有一些不完整的数据，这会干扰列类型的选择。如果进行家谱研究，需要记录出生日期和死亡日期，有时会发现所能搜集到的数据中只是某人出生或死亡的年份，但没有确切的日期。如果使用 DATE 列，除非有完整的日期值，否则不能输入日期。如果希望能够记录所具有的任何信息，即使不完整也保存，那么可能必须保存独立的年、月、日字段。这样就可以输入所具有的日期成员并将没有的部分设为 NULL。在 MySQL 3.23 及以后的版本中，还允许 DATE 的日为 0 或者月和日部分为 0。这样“模糊”的日期可用来表示不完整的日期值。

2.3.2 列值有特定的取值范围吗

如果已经决定从通用类别上选择一种列类型，那么考虑想要表示的值的取值范围会有助于将您的选择缩减到该类别中特定的类型上。假如希望存储整数值。这些整数值的取值范围为 0 到 1000，那么可以使用从 SMALLINT 到 BIGINT 的所有类型。如果这些整数值的取值范围最多为 2 000 000，则不能使用 SMALLINT，其选择范围从 MEDIUMINT 到 BIGINT。需要从这个可能的选择范围中选取一种类型。

当然，可以简单地为想要存储的值选择最大的类型（如上述例子中选择 BIGINT）。但是，一般应该为所要存储的值选择足以存储它的最小的类型。这样做，可以最小化表占用的存储量，得到最好的性能，因为通常较小列的处理比较大列的快。

如果不知道所要表示的值的取值范围，那么必须进行猜测或使用 BIGINT 以应付最坏的情况。（请注意，如果进行猜测时使用了一个太小的类型，工作不会白做；以后可以利用

ALTER TABLE 来将此列改为更大一些的类型。)

在第1章中,我们为学分保存方案创建了一个 score 表,它有一个记录测验和测试学分的 score 列。为了讨论简单起见,创建该表时使用了 INT 类型,但现在可以看出,如果学分数在 0 到 100 的取值范围内,更好的选择应该是 TINYINT UNSIGNED,因为所用的存储空间较小。

数据的取值范围还影响列类型的属性。如果该数据从不为负,可使用 UNSIGNED 属性;否则就不能用它。

串类型没有数值列那样的“取值范围”,但它们有长度,需要知道该串可使用的列最大长度。如果串短于 256 个字符,可使用 CHAR、VARCHAR、TINYTEXT 或 TINYBLOB 等类型。如果想要更长的串,可使用 TEXT 或 BLOB 类型,而 CHAR 和 VARCHAR 不再是选项。

对于用来表示某个固定值集合的串列,可以考虑使用 ENUM 或 SET 列类型。它们可能是很好的选项,因为它们在内部是用数来表示的。这两个类型上的运算是数值化的,因此,比其他的串类型效率更高。它们还比其他串类型紧凑、节省空间。

在描述必须处理的值的范围时,最好的术语是“总是”和“决不”(如“总是小于 1000”或“决不为负”),因为它们能更准确地约束列类型的选择。但在未确证之前,要慎用这两个术语。特别是与其他人谈他们的数据,而他们开始乱用这两个术语时要注意。在有人说“总是”或“决不”时,一定要搞清他们说的确实是这个含义。有时人们说自己的数据总是有某种特定的性质,而其真正的含义是“几乎总是”。

例如,假如您为某些人设计一个表,而他们告诉您,“我们的测试学分数总是 0 到 100”。根据这个描述,您选择了 TINYINT 类型并使它为 UNSIGNED 的,因为值总是非负的。然而,您发现编码录入数据库的人有时用 -1 来表示“学生因病缺席”。呀,他们没告诉您这事。可能可以用 NULL 来表示 -1,但如果不能,必须记录 -1,这样就不能用 UNSIGNED 列了(只好用 ALTER TABLE 来补救!)。

有时关于这些情形的讨论可通过提一些简单的问题来简化,如问:曾经有过例外吗?如果曾经有过例外情况,即使是只有一次,也必须考虑。您会发现,和您讨论数据库设计的人总是认为,如果例外不经常发生,那么就没什么关系。然而在创建数据库时,就不能这样想了。需要提的问题并不是例外出现有多频繁,而是有没有例外?如果有,必须考虑进去。

2.3.3 性能与效率问题

列类型的选择会在几个方面影响查询性能。如果记住下几节讨论的一般准则,将能够选出有助于 MySQL 有效处理表的列类型。

1. 数值与串的运算

数值运算一般比串运算更快。例如比较运算,可在单一运算中对数进行比较。而串运算涉及几个逐字节的比较,如果串更长的话,这种比较还要多。

如果串列的值数目有限,应该利用 ENUM 或 SET 类型来获得数值运算的优越性。这两种类型在内部是用数表示的,可更为有效地进行处理。

例如替换串的表达。有时可用数来表示串值以改进其性能。例如,为了用点分四位数

(dotted-quad) 表示法来表示 IP 号, 如 192.168.0.4, 可以使用串。但是也可以通过用四字节的 UNSIGNED 类型的每个字节存储四位数的每个部分, 将 IP 号转换为整数形式。这即可以节省空间又可加快查找速度。但另一方面, 将 IP 号表示为 INT 值会使诸如查找某个子网的号码这样的模式匹配难于完成。因此, 不能只考虑空间问题; 必须根据利用这些值做什么来决定哪种表示更适合。

2. 更小的类型与更大的类型

更小的类型比更大的类型处理要快得多。首先, 它们占用的空间较小, 且涉及的磁盘活动开销也少。对于串, 其处理时间与串长度直接相关。

一般情况下, 较小的表处理更快, 因为查询处理需要的磁盘 I/O 少。对于定长类型的列, 应该选择最小的类型, 只要能存储所需范围的值即可。例如, 如果 MEDIUMINT 够用, 就不要选择 BIGINT。如果只需要 FLOAT 精度, 就不应该选择 DOUBLE。对于可变长类型, 也仍然能够节省空间。一个 BLOB 类型的值用 2 字节记录值的长度, 而一个 LONGBLOB 则用 4 字节记录其值的长度。如果存储的值长度永远不会超过 64KB, 使用 BLOB 将使每个值节省 2 字节 (当然, 对于 TEXT 类型也可以做类似的考虑)。

3. 定长与可变长类型

定长类型一般比可变长类型处理得更快:

- 对于可变长列, 由于记录大小不同, 在其上进行许多删除和更改将会使表中的碎片更多。需要定期运行 OPTIMIZE TABLE 以保持性能。而定长列就没有这个问题。
- 在出现表崩溃时, 定长列的表易于重新构造, 因为每个记录的开始位置是确定的。可变长列就没有这种便利。这不是一个与查询处理有关的性能问题, 但它必定能加快表的修复过程。

如果表中有可变长的列, 将它们转换为定长列能够改进性能, 因为定长记录易于处理。在试图这样做之前, 应该考虑下列问题:

- 使用定长列涉及某种折衷。它们更快, 但占用的空间更多。CHAR(n) 类型列的每个值总要占用 n 个字节 (即使空串也是如此), 因为在表中存储时, 值的长度不够将在右边补空格。而 VARCHAR(N) 类型的列所占空间较少, 因为只给它们分配存储每个值所需要的空间, 每个值再加一个字节用于记录其长度。因此, 如果在 CHAR 和 VARCHAR 列之间进行选择, 需要对时间与空间作出折衷。如果速度是主要关心的因素, 则利用 CHAR 列来取得定长列的性能优势。如果空间是关键, 应该使用 VARCHAR 列。
- 不能只转换一个可变长列; 必须对它们全部进行转换。而且必须使用一个 ALTER TABLE 语句同时全部转换, 否则转换将不起作用。
- 有时不能使用定长类型, 即使想这样做也不行。例如对于比 255 字符长的串, 没有定长类型。

4. 可索引类型

索引能加快查询速度, 因此, 应该选择可索引的类型。

5. NULL 与 NOT NULL 类型

如果定义一列为 NOT NULL, 其处理更快, 因为 MySQL 在查询处理中不必检查该列的值弄清它是否为 NULL, 表中每行还能节省一位。

避免列中有 NULL 可以使查询更简单, 因为不需要将 NULL 作为一种特殊情形来考虑。

通常，查询越简单，处理就越快。

所给出的性能准则有时是互相矛盾的。例如，根据 MySQL 能对行定位这一方面来说，包含 CHAR 列的定长行比包含 VARCHAR 列的可变长行处理快。但另一方面，它也将占用更多的空间，因此，会导致更多的磁盘活动。从这个观点来看，VARCHAR 可能会更快。作为一个经验规则，可假定定长列能改善性能，即使它占用更多的空间也如此。对于某个特殊的关键应用，可能会希望以定长和可变长两种方式实现一个表，并进行某些测试以决定哪种方式对您的特定应用来说更快。

2.3.4 希望对值进行什么样的比较

根据定义串的方式，可以使串类型以区分大小写或不区分大小写的方式进行比较和排序。表2-14 示出不区分大小写的每个类型及其等价的区分大小写类型。根据列定义中给不给出关键字 BINARY，有的类型（CHAR、VARCHAR）是二进制编码或非二进制编码的。其他类型（BLOB、TEXT）的“二进制化”隐含在类型名中。

表2-14 串类型是否区分大小写

非二进制类型（不区分大小写）	二进制类型（区分大小写）
CHAR(M)	CHAR(M) BINARY
VARCHAR(M)	VARCHAR(M) BINARY
TINYTEXT	TINYBLOB
TEXT	BLOB
MEDIUMTEXT	MEDIUMBLOB
LONGTEXT	LONGBLOB

请注意，二进制（区分大小写）类型仅在比较和排序行为上不同于相应的非二进制（不区分大小写）类型。任意串类型都可以包含任意种类的数据。特别是，TEXT 类型尽管在列类型名中称为“TEXT（文本）”，但它可以很好地存储二进制数据。

如果希望使用一个在比较时既区分大小写，又可不区分大小写的列。可在希望进行区分大小写的比较时，利用 BINARY 关键字强制串作为二进制串值。例如，如果 my_col 为一个 CHAR 列，可按不同的方式对其进行比较：

my_col = "ABC"	不区分大小写
BINARY my_col = "ABC"	区分大小写
my_col = BINARY "ABC"	区分大小写

如果有一个希望以非字典顺序存储的串值，可考虑使用 ENUM 列。ENUM 值的排序是根据列定义中所列出枚举值的顺序进行的，因此可以使这些值以任意想要的次序排序。

2.3.5 计划对列进行索引吗

使用索引可更有效地处理查询。索引的选择是第 4 章中的一个主题，但一般原则是将 WHERE 子句中用来选择行的列用于索引。

如果您要对某列进行索引或将该列包含在多列索引中，则在类型的选择上可能会有限定。在早于 3.23.2 版的 MySQL 发行版中，索引列必须定义为 NOT NULL，并且不能对 BLOB 或 TEXT 类型进行索引。这些限制在 MySQL 3.23.2 版中都撤消了，但如果您正使用一个更早的版本，不能或不愿升级，那么必须遵从这些约束。不过在下列情形中可以绕过它们：

- 如果可以指定某个值作为专用的值，那么能够将其作为与 NULL 相同的东西对待。对于 DATE 列，可以指定“0000 - 00 - 00”表示“无日期”。在串列中，可以指定空串代表“缺值”。在数值列中，如果该列一般只存储非负值，则可使用 -1。
- 不能对 BLOB 或 TEXT 类型进行索引，但如果串不超过 255 它符，可使用等价的 VARCHAR 列类型并对其进行索引。可将 VARCHAR(255) BINARY 用于 BLOB 值，将 VARCHAR(255) 用于 TEXT 值。

2.3.6 列类型选择问题的相互关联程度

不要以为列类型的选择是相互独立的。例如，数值的取值范围与存储大小有关；在增大取值的范围时，需要更多的存储空间，这会影响性能。另外，考虑选择使用 AUTO_INCREMENT 来创建一个存放唯一序列号的列有何含义。这个选择有几个结果，它们涉及列的类型、索引和 NULL 的使用，现列出如下：

- AUTO_INCREMENT 是一个应该只用于整数类型的列属性。它将您的选择限定在 TINYINT 到 BIGINT 之上。
- AUTO_INCREMENT 列应该进行索引，从而当前最大的序列号可以很快就确定，不用对表进行全部扫描。此外，为了防止序列号被重用，索引号必须是唯一的。这表示必须将列定义为 PRIMARY KEY 或定义为 UNIQUE 索引。
- 如果所用的 MySQL 版本早于 3.23.2，则索引列不能包含 NULL 值，因此，必须定义列为 NOT NULL。

所有这一切表示，不能像如下这样只定义一个 AUTO_INCREMENT 列：

```
my_col arbitrary_type AUTO_INCREMENT
```

应该按如下定义：

```
my_col integer_type AUTO_INCREMENT NOT NULL PRIMARY KEY
```

或如下定义：

```
my_col integer_type AUTO_INCREMENT NOT NULL, UNIQUE(my_col)
```

使用 AUTO_INCREMENT 得到的另一个结果是，由于它是用来生成一个正值序列的，因此，最好将 AUTO_INCREMENT 列定义为 UNSIGNED：

```
my_col integer_type UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY  
my_col integer_type UNSIGNED AUTO_INCREMENT NOT NULL, UNIQUE(my_col)
```

2.4 表达式求值和类型转换

MySQL 允许编写包括常量、函数调用和表列引用的表达式。这些值可利用不同类型的运算符进行组合，诸如算术运算符或比较运算符。表达式的项可用圆括号来分组。

表达式在 SELECT 语句的列选择列表和 WHERE 子句中出现得最为频繁，如下所示：

```
SELECT  
    CONCAT(last_name, ", ", first_name),  
    (TO_DAYS(death) - TO_DAYS(birth) / 365)  
FROM president  
WHERE  
    birth > "1900-1-1" AND DEATH IS NOT NULL
```

所选择的每列给出了一个表达式，如 WHERE 子句中所示的那样。表达式也出现在

DELETE 和 UPDATE 语句的 WHERE 子句中, 以及出现在 INSERT 语句的 VALUES() 子句中。

在 MySQL 遇到一个表达式时, 它对其求值得出结果。例如, $(4 * 3)/(4 - 2)$ 求值得 6。表达式求值可能涉及类型转换。例如, MySQL 在数 960821 用于需要日期值的环境时, 将其转换为日期 “1996-08-21”。

本节讨论怎样编写 MySQL 的表达式, 以及在表达式求值中 MySQL 所使用的类型转换规则。每个 MySQL 的运算符都介绍过了, 但 MySQL 有那么多的函数, 我们只接触过几个。每个运算符和函数的进一步介绍可参阅附录 C。

2.4.1 撰写表达式

表达式可以只是一个简单的常量, 如:

0	数值常量
“abc”	串常量

表达式可以进行函数调用。有的函数需要参数 (圆括号中有值), 而有的不需要。多个参数应该用逗号分隔。在调用一个函数时, 参数旁边可以有空格, 但在函数名与圆括号间不能有空格。下面是一些函数例子:

NOW()	无参数函数
STRCMP(“abc”, “def”)	有两个参数的函数
STRCMP(“abc”, “def”)	参数旁边有空格是合法的
STRCMP(“abc”, “def”)	函数名后跟空格是不合法的

如果函数名后有一个空格, MySQL 的分析程序可能会将函数名解释为一个列名 (函数名不是保留字, 如果需要的话, 可将它们用作列名)。其结果是出现一个语法错误。

表达式中可使用表列。最简单的情形是, 当某个列所属的表在上下文中是明确的, 则可简单地给出列名对该列进行引用。下面的每个 SELECT 语句中惟一地出了一个表名, 因此, 列的引用无歧义:

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

如果使用哪个表的列不明确, 可在列名前加上表名。如果使用哪个数据库中的表也不明确的话, 可在表名前加上数据库名。如果只是希望意思更明显, 也可以在无歧义的上下文中利用这种更为具体的表示形式, 如:

```
SELECT
    president.last_name, president.first_name,
    member.last_name, member.first_name
FROM president, member
WHERE president.last_name = member.last_name

SELECT samp_db.student.name FROM samp_db.student
```

总之, 可以组合所有这些值以得到更为复杂的表达式。

1. 运算符的类型

MySQL 有几种类型的运算符, 可用来连接表达式的项。算术运算符, 如表 2-15 所示, 一般包括加、减、乘、除以及模运算符。在两个操作数都是整数时, “+”、“-”和 “*” 算术运算用 BIGINT (64 位) 整数值来完成。而在结果预期为一个整数时, “/”和 “%”也是用

BIGINT（64 位）整数值来完成的。应该认识到，如果某个运算涉及更大的值，如结果超过 64 位，其结果不可预料。

表2-15 算术运算符

运 算 符	语 法	说 明
+	a + b	加；操作数之和
-	a - b	减；操作数之差
-	-a	一元减号；操作数取负
*	a * b	乘；操作数之积
/	a / b	除；操作数之商
%	a % b	模；操作数除后的余数

逻辑运算符如表2-16所示，对表达式进行估计以确定其为真（非零）或假（零）。MySQL 包含有 C 风格的“&&”、“||”和“!”运算符，可替换 AND、OR 和 NOT。要特别注意“||”运算符，ANSI SQL 指定“||”作为串连接符，但在 MySQL 中，它表示一个逻辑或运算。如果执行下面的查询，则返回数 0：

```
SELECT "abc" || "def" → 0
```

MySQL 为进行运算，将“abc”和“def”转换为整数，且两者都转换为 0，0 与 0 进行或运算，结果为 0。在 MySQL 中，必须用 CONCAT(“abc”，“def”)来完成串的连接。

表2-16 逻辑运算符

运 算 符	语 法	说 明
AND, &&	a AND B, a && b	逻辑与；如果两操作数为真，结果为真
OR,	a OR B, a b	逻辑或；任一操作数为真，结果为真
NOT, !	NOT a, !a	逻辑非；如果操作数为假，结果为真

位运算符如表2-17 所示，完成按位“与”和“或”，其中结果的每一位按两个操作数的对应位的逻辑 AND 或 OR 求值。还可以进行位的左移或右移。位运算用 BIGINT（64 位）整数值进行。

表2-17 位运算

运 算 符	语 法	说 明
&	a & b	按位 AND（与）；如果两个操作数的对应位为 1，则该结果位为 1
	a b	按位 OR（或）；如果两操作数的对应位中有一位为 1，则该结果位为 1
<<	a << b	将 a 左移 b 个二进制位
>>	a >> b	将 a 右移 b 个二进制位

比较运算符如表 2-18 所示，其中包括测试相对大小或数和串的顺序的运算符，以及完成模式匹配和测试 NULL 值的运算符。“<=>”运算符是 MySQL 特有的，在 MySQL 3.23 版本中引入。

表2-18 比较运算符

运 算 符	语 法	说 明
=	a = b	如果两操作数相等，为真
!=, <>	a != b, a <> b	如果两操作数不等，为真
<	a < b	如果 a 小于 b，为真
<=	a <= b	如果 a 小于等于 b，为真

(续)

运 算 符	语 法	说 明
>=	a >= b	如果 a 大于等于 b, 为真
>	a > b	如果 a 大于 b, 为真
IN	a IN (b1, b2, ...)	如果 a 为 b1, b2, ...中任意一个, 为真
BETWEEN	a BETWEEN b AND c	如果 a 在值 b 和 c 之间包括等于 b 和 c, 为真
LIKE	a LIKE b	SQL 模式匹配; 如果 a 与 b 匹配, 为真
NOT LIKE	a NOT LIKE b	SQL 模式匹配; 如果 a 与 b 不匹配, 为真
REGEXP	a REGEXP b	扩展正规表达式匹配; 如果 a 与 b 匹配, 为真
NOT REGEXP	a NOT REGEXP b	扩展正规表达式匹配; 如果 a 与 b 不匹配, 为真
<=>	a <=> b	如果两操作数相同 (即使为 NULL), 为真
IS NULL	a IS NULL	如果操作数为 NULL, 为真
IS NOT NULL	a IS NOT NULL	如果操作数不为 NULL, 为真

自 MySQL 3.23 版本起, 可使用 BINARY 运行符, 此运算符可用来将一个串转换为一个二进制串, 这个串在比较中是区分大小写的。下列的第一个比较是不区分大小写的, 但第二个和第三个比较是区分大小写的:

```
"abc" = "Abc"           → 1
BINARY "abc" = "Abc"    → 0
"abc" = BINARY "Abc"    → 0
```

没有相应的 NOT BINARY 计算。如果希望使一个列既能在区分大小写又能在不区分大小写的环境中使用, 则应该利用不区分大小写的列并对希望区分大小写的比较使用 BINARY。

对于利用二进制串类型 (CHAR BINARY、VARCHAR BINARY 和 BLOB 类型) 定义的列, 其比较总是区分大小写的。为了对这样的列类型实现不区分大小写的比较, 可利用 UPPER() 或 LOWER() 来转换成相同的大小写:

```
UPPER(col_name) < UPPER("Smith")
LOWER(col_name) < LOWER("Smith")
```

对于不区分大小写的串比较, 有可能把多个字符认为是相等的, 这取决于所用的字符集。例如 “e” 和 “é” 对于比较和排序操作可能是相同的。二进制 (区分大小写) 比较利用字符的 ASCII 值来完成。

模式匹配允许查找值而不必给出精确的直接值。MySQL 利用 LIKE 运算符和通配符 “%” (匹配任意的字符序列) 和 “_” (匹配任意单个字符), 提供 SQL 的模式匹配。MySQL 还基于类似于诸如 grep、sed 和 vi 等 UNIX 程序中所用的 REGEXP 运算符和扩展正规表达式, 提供模式匹配。为了完成模式匹配, 必须使用这些模式匹配运算符中的某一个; 不能使用 “=”。为了进行相反的模式匹配, 可使用 NOT LIKE 或 NOT REGEXP。

除了使用的模式运算符和模式字符不同外, 这两种模式匹配还在以下重要的方面存在差异:

- 除非至少有一个操作数为二进制串, 否则 LIKE 是不区分大小写的。REGEXP 是区分大小写的。(在 MySQL 3.23.4 以后的版本中, 除非至少有一个操作数是二进制串, 否则 REGEXP 是不区分大小写的。)
- 仅当整个串匹配, SQL 才是模式匹配的。仅当相应的模式在串中某一处出现, 正规表达式才匹配。

用于 LIKE 运算符的模式可以包括 “%” 和 “_” 通配符。例如, 模式 “Frank%” 与任何以 “Frank” 起头的串匹配:

```
"Franklin" LIKE "Frank%"      → 1
"Frankfurter" LIKE "Frank%"    → 1
```

通配符“%”与任何串匹配，其中包括与空字符串序列匹配，因此“Frank%”与“Frank”匹配：

```
"Frank" LIKE "Frank%"          → 1
```

这也表示模式“%”与任何串匹配，其中包括与空串匹配。但是，“%”不与 NULL 匹配。事实上，具有 NULL 操作数的任何模式匹配都将失败：

```
"Frank" LIKE NULL              → NULL
NULL LIKE "Frank%"            → NULL
```

MySQL 的 LIKE 运算符是不区分大小写的，除非它至少有一个操作数是二进制串。因此，缺省时“Frank%”与串“Frankly”和“frankly”匹配，但在二进制比较中，它只与其中之一匹配：

```
"Frankly" LIKE "Frank%"        → 1
"frankly" LIKE "Frank%"        → 1
BINARY "Frankly" LIKE "Frank%" → 1
BINARY "frankly" LIKE "Frank%" → 0
```

这不同于 ANSI SQL 的 LIKE 运算符，它是区分大小写的。

通配符可在模式中任何地方给出。“%bert”与“Englebert”、“Bert”和“Albert”匹配。“%bert%”也与所有这些串匹配，而且还与如像“Berthold”、“Bertram”和“Alberta”这样的串匹配。

LIKE 所允许的另一个通配符是“_”，它与单个字符匹配。“___”与三个字符的串匹配。“c_t”与“cat”、“cut”甚至“c_t”匹配（因为“_”与自身匹配）。

为了关掉“%”或“_”的特殊含义，与这些字符的直接实例相匹配，需要在它们前面放置一个斜杠（“\%”或“_”），如：

```
"abc" LIKE "a%c"              → 1
"abc" LIKE "a\%c"             → 0
"a%c" LIKE "a\%c"             → 1
```

MySQL 的另一种形式的模式匹配使用了正规表达式。运算符为 REGEXP 而不是 LIKE（RLIKE 为 REGEXP 的同义词）。最常用的正规表达式模式字符如下：

‘.’与任意单个字符匹配：

```
"abc" REGEXP "a.c"            → 1
```

‘[...]’与方括号中任意字符匹配。可列出由短划线“-”分隔的范围端点指定一个字符范围。为了否定这种区间的意义（即与未列出的任何字符匹配），指定“^”作为该区间的第一个字符即可：

```
"abc" REGEXP "[a-z]"          → 1
"abc" REGEXP "[^a-z]"          → 0
```

‘*’表示“与其前面字符的任意数目的字符匹配”，因此，如“x*”与任意数目的“x”字符匹配，例如：

```
"abcdef" REGEXP "a.*f"        → 1
"abc" REGEXP "[0-9]*abc"       → 1
"abc" REGEXP "[0-9][0-9]*"     → 0
```

“任意数目”包括 0 个实例，这也就是为什么第二个表达式匹配成功的原因。‘^pat’和‘pat\$’固定了一种模式匹配，从而模式 pat 只在它出现在串的前头时匹配，而‘^pat\$’只在

pat 匹配整个串时匹配，例如：

```
"abc" REGEXP "b"           → 1
"abc" REGEXP "^b"          → 0
"abc" REGEXP "b$"          → 0
"abc" REGEXP "^abc$"       → 1
"abcd" REGEXP "^abc$"      → 0
```

REGEXP 模式可从某个表列中取出，虽然如果该列包含几个不同的值时，这样做比常量模式慢。每当列值更改时，必须对模式进行检查并转换成内部形式。

MySQL 的正规表达式匹配还有一些特殊的模式字符。要了解更详细信息请参阅附录 C。

2. 运算符的优先级

当求一个表达式的值时，首先查看运算符以决定运算的先后次序。有的运算符具有较高的优先级；例如，乘和除以加和减的优先级更高。下面的两个表达式是等价的，因为“*”和“/”先于“+”和“-”计算：

```
1 + 2 * 3 - 4 / 5           → 6.2
1 + 6 - .8                  → 6.2
```

下面列出了运算符的优先级，从高到低。列在同一行中的运算符具有相同的优先级。优先级较高的运算符在优先级较低的运算符之前求值。

```
BINARY
NOT !
- (unary minus)
* / %
+ -
<< >>
&
|
< <= = <=> != <> >= > IN IS LIKE REGEXP RLIKE
BETWEEN
AND &&
OR ||
```

可用圆括号来忽略运算符的优先级并改变表达式的求值顺序，如：

```
1 + 2 * 3 - 4 / 5           → 6.2
(1 + 2) * (3 - 4) / 5       → -0.6
```

3. 表达式中的 NULL 值

请注意，在表达式中使用 NULL 值时，其结果有可能出现意外。下列准则将有助于避免出问题。

如果将 NULL 作为算术运算或位运算符的一个操作数，其结果为 NULL：

```
1 + NULL                    → NULL
1 | NULL                    → NULL
```

如果将 NULL 用于逻辑运算符，NULL 被认为是假：

```
1 AND NULL                  → 0
1 OR NULL                   → 1
0 AND NULL                  → 0
0 OR NULL                   → 0
```

NULL 作为任意比较运算符的操作数，除 <=>、IS NULL 和 IS NOT NULL 运算符（它们是专门扩展来处理 NULL 值的）外，将产生一个 NULL 结果。如：

```
1 = NULL                    → NULL
NULL = NULL                 → NULL
1 <=> NULL                   → 0
```

```
NULL <=> NULL          → 1
1 IS NULL              → 0
NULL IS NULL          → 1
```

如果给函数一个 NULL 参数,除了那些处理 NULL 参数的函数外,一般返回一个 NULL 结果。例如,IFNULL() 能够处理 NULL 参数并适当地返回真或假。STRCMP() 期望一个非 NULL 的参数;如果它发现传给它的是一个 NULL 参数,则返回 NULL 而不是真或假。

在排序操作中, NULL 值被归到一起。在升序排序中, NULL 将排在所有非 NULL 值之前(包括空串),而在降序排序中, NULL 将排在所有非 NULL 值之后。

2.4.2 类型转换

MySQL 根据所执行的操作类型,自动地进行大量的类型转换,任何时候,只要将一个类型的值用于需要另一类型值的场合,就会进行这种转换。下面是需要进行类型转换的原因:

- 操作数转换为适合于某种运算符求值的类型。
- 函数参数转换为函数所需的类型。
- 转换某个值以便赋给一个具有不同类型的表列。

下列表达式涉及类型转换。它由加运算符“+”和两个操作数 1 和“2”组成:

```
1 + "2"
```

其中操作数的类型不同,一个是数,另一个是串,因此,MySQL 对其中之一进行转换以便使它们两个具有相同的类型。但是应该转换哪一个呢?因为,“+”是一个数值运算符,所以 MySQL 希望操作数为数,因此,将串“2”转换为数 2。然后求此表达式的值得出 3。再举一例。CONCAT() 函数连接串产生一个更长的串作为结果。为了完成此工作,它将参数解释为串,而不管参数实际是何类型。如果传递给 CONCAT() 几个数,则它将把它们转换成串,然后返回这些串的连接,如:

```
CONCAT(1,2,3)          → "123"
```

如果作为表达式的组成部分调用 CONCAT(),可能会进行进一步的类型转换。考察下列表达式及其结果:

```
REPEAT('X',CONCAT(1,2,3)/10) → "XXXXXXXXXXXX"
```

CONCAT(1, 2, 3) 产生串“123”。表达式“123”/10 转换为 123/10,因为除是一个算术运算符。这个表达式的结果的浮点形式为 12.3,但 REPEAT() 需要整数的重复计数值,所以进行整除得 12。然后,REPEAT('X', 12) 产生一个含有 12 个‘X’字符的结果串。

一般原则是,MySQL 尽量将值转换为表达式所需要的类型,尽量避免由于值的类型不对而导致错误。根据上下文,MySQL 将在三种通用类型(数、串或日期与时间)之间进行值的转换。但是,值不能总是可以从一种类型转为另一种类型。如果被转换值不是给定类型的合法值,则此转换失败。将如“abc”这样不像数的东西转换为数,则结果为 0。将不像日期或时间的东西转换为日期或时间类型结果为该类型的“零”值。例如,将串“abc”转换为日期结果为“零”日期“0000-00-00”。而任何值都可以处理为串,因此,一般将某个值转换为串不会产生问题。

MySQL 也进行一些微小的类型转换。如果在整型环境中使用一个浮点值,此值将被转换,转换时进行四舍五入。也可以进行相反的工作;一个整数用作浮点数也不会有问题。

除非其内容显示表示一个数,否则十六进制常数一般作为串处理。在串上下文中,每对

十六进制数字转换为一个字符，其结果作为串。下面是一些转换的样例：

```
0x61                → "a"
0x61 + 0            → 97
CONCAT(0x61)        → "a"
CONCAT(0x61 + 0)    → "97"
```

相同的解释原理也应用到比较上；除非与其比较的是一个数，否则十六进制常量按串对待，例如：

```
10 = 0x0a          → 1
10 = 0x09          → 0
"\n" = 0x0a        → 1
"\n" = 0x0a + 0    → 0
("\n" = 0x0a) + 0  → 1
```

某些运算符可将操作数强制转换为它们所要的类型，而不管操作数是什么类型。例如，算术运算符需要数，并按此对操作数进行转换，参考如下运算：

```
3 + 4              → 7
"3" + 4            → 7
"3" + "4"          → 7
```

MySQL 不对整个串进行寻找一个数的查找；它只查看串的起始处。如果一个串不以数作为前导部分，其转换结果为 0。

```
"1973-2-4" + 0     → 1973
"12:14:01" + 0     → 12
"23-skidoo" + 0    → 23
"-23-skidoo" + 0   → -23
"carbon-14" + 0    → 0
```

请注意，MySQL 的串到数的转换规则自 3.23 版以后已经改变了。在该版本以前，类似于数的串被转换为四舍五入的整数值。自 3.23 版后，它们转换为浮点值，例如：

```
"-428.9" + 0       → -429 (MySQL < 3.23)
"-428.9" + 0       → -428.9 (MySQL ≥ 3.23)
```

逻辑和位运算符比算术运算符要求更为严格。它们不仅希望操作数为数，而且还要求是整数。这表示一个浮点数，如 .3，不被视为真，虽然它是非零的；这是因为在转换为整数时，.3 已经转换为 0 了。在下面的表达式中，除非各操作数有一个至少为 1 的值，否则各操作数不被认为是真。

```
0.3 OR .04         → 0
1.3 OR .04         → 1
0.3 AND .04        → 0
1.3 AND .04        → 0
1.3 AND 1.04       → 1
```

这种转换也出现在 IF() 函数中，此函数要求第一个参数为整数。为了恰当地对浮点值进行测试，最好是利用明确的比较。否则，小于 1 的值将被认为是假，例如：

```
IF(1.3, "non-zero", "zero") → "non-zero"
IF(0.3, "non-zero", "zero") → "zero"
IF(0.3>0, "non-zero", "zero") → "non-zero"
```

模式匹配运算符要求对串进行处理。这表示可将 MySQL 的模式匹配运算符用于数，因为 MySQL 会在试图进行的匹配中将它们转换成串。例如：

```
12345 LIKE "1%"      → 1
12345 REGEXP "1.*5"   → 1
```

大小比较运算符（“<”、“<=”、“=”等等）是上下文相关的；即，它们根据操作数的类

型求值。下面的表达式从数值上对操作数进行比较，因为操作符两边都是数。

```
2 < 11                                → 1
```

下面的表达式涉及串比较，因为其两边的操作数都是串：

```
"2" < "11"                            → 0
```

在下面的比较中，类型是混合的，因此，MySQL 按数比较它们。结果是两个表达式都为真：

```
"2" < 11                              → 1
```

```
2 < "11"                                → 1
```

在各个比较中，MySQL 根据下列规则对操作数进行转换：

- 除了“<=>”运算符外，涉及 NULL 值的比较其值为 NULL（除 NULL <=> NULL 为真外，“<=>”与“=”相同）。
- 如果两个操作数都是串，则按串进行字典顺序的比较。串比较利用服务器上有效的字符集进行。
- 如果两个操作数都为整数，则按整数进行数的比较。
- 不与数进行比较的十六进制常量按二进制串进行比较。
- 如果其中有一个操作数为 TIMESTAMP 或 DATETIME 值而另一个为常量，则按 TIMESTAMP 值进行比较。这样做将使比较对 ODBC 应用更好。
- 否则，两个操作数将按浮点值进行数的比较。注意，这包括一个串与一个数进行比较的情况。其中串被转换为数，如果该串转换后不是一个数，则结果为 0。例如，“14.3”转换为 14.3，但“L4.3”转换为 0。

1. 日期与时间的解释规则

MySQL 按表达式的环境将串和数自由地转换为日期和时间值，反之亦然。日期和时间值在数值上下文中转换为数；数在日期或时间上下文中转换为日期或时间。在将一个值赋予一个日期或时间列时，或在函数需要一个日期或时间值时，进行转换为日期或时间值的转换。

如果表 my_table 含有一个 DATE 列 date_col，下列语句是等价的：

```
INSERT INTO my_table SET date_col = "1997-04-13"
INSERT INTO my_table SET date_col = "19970413"
INSERT INTO my_table SET date_col = 19970413
```

TO_DAYS() 函数的参数在下面三个表达中为相同的值：

```
TO_DAYS("1997-04-10")                → 729489
TO_DAYS("19970410")                   → 729489
TO_DAYS(19970410)                     → 729489
```

2. 测试并强制进行类型转换

为了了解表达式中类型转换是怎样进行的，用 mysql 程序发布一条对表达式求值的 SELECT 语句如下：

```
mysql> SELECT 0x41, 0x41 + 0;
+-----+-----+
| 0x41 | 0x41 + 0 |
+-----+-----+
| A    | 65       |
+-----+-----+
```

正如您所想像的那样，笔者在撰写本章时，做了不少这种比较。

测试表达式的求值对于诸如 DELETE 或 UPDATE 这种修改记录的语句极为重要，因为需

要保证只涉及所需涉及的行。检查表达式的一个办法是，预先执行一条具有准备用于 DELETE 或 UPDATE 语句的相同 WHERE 子句，以验证该子句选择的行是正确的。假如表 my_table 具有一个含有下列值的 CHAR 列 char_col：

```
"abc"
"def"
"00"
"ghi"
"jkl"
"00"
"mno"
```

对于这些值，下列查询的作用是什么？

```
DELETE FROM my_table WHERE char_col = 00
```

原来的打算大概是想删除包含值“00”的那两行。但实际作用是删除了所有的行。之所以这样是由于 MySQL 的比较规则在起作用。char_col 为一个串列，但 00 没有用引号括起来，因此，它被作为数对待了。按 MySQL 的比较规则，涉及一个串与一个数的比较按两个数的比较来求值。随着 DELETE 查询的执行，char_col 的每个值被转换为 0，“00”也被转换为 0，因此，所有不类似数的串都转换成 0。从而，对于每一行，WHERE 子句都为真，因此，DELETE 语句清空了该表。显然，这是一种在执行 DELETE 前，应该用 SELECT 语句对 WHERE 子句进行测试的情况，这样将会示出表达式所选择的行太多了。如下所示：

```
mysql> SELECT char_col FROM my_table WHERE char_col = 00;
```

```
+-----+
| char_col |
+-----+
| "abc"    |
| "def"    |
| "00"     |
| "ghi"    |
| "jkl"    |
| "00"     |
| "mno"    |
+-----+
```

如果不能肯定某个值的使用方式，可以利用 MySQL 的表达式求值机制将该值强制转换为特定的类型：

■ 增加 + 0 或 + 0.0 到某项上以强制转换到一个数值：

```
0x65          → "e"
0x65 + 0      → 101
0x65 + 0.0    → 101.0
```

■ 利用 CONCAT() 将值转换为串：

```
14            → 14
CONCAT(14)    → "14"
```

■ 利用 ASCII() 得到字符的 ASCII 值：

```
"A"          → "A"
ASCII("A")    → 65
```

■ 利用 DATE_ADD() 强制转换串或数为日期：

```
19990101      → 19990101
DATE_ADD(19990101, INTERVAL 0 DAY) → "1999-01-01"
"19990101"    → "19990101"
```

```
DATE_ADD("19990101", INTERVAL 0 DAY)
```

```
→ "1999-01-01"
```

3. 超范围值或非法值的转换

超范围值或非法值的转换的基本原则为：无用输入，无用输出。如果不在存储日期前对其进行验证，那么可能会得到不喜欢的东西。下面给出一些 MySQL 处理超范围值或不合适值的一般原则，这些内容曾经在前面介绍过：

- 对于数值或 TIME 列，超出合法范围的值被剪裁为相应取值范围的最接近的数值并作为结果值存储。
- 对于非 ENUM 或 SET 的串列，太长的串被截为适合该列存储的最大长度的串。
ENUM 或 SET 列的赋值依赖于定义列时给出的合法值。如果赋予 ENUM 列一个未作为枚举成员给出的值，将会赋予一个错误成员（即，对应于零值成员的空串）。如果赋予 SET 列一个包含未作为集合成员给出的子串的值，那么，那些未作为集合成员给出的子串将被删除，并将剩余成员构成的值赋给该列。
- 对于日期或时间列，非法值被转换为该类型适当的“零”值（参阅表 2-11）。对于非 TIME 的日期和时间列，超出取值范围的值可转换为“零”值、NULL 或某种其他的值（换句话说，结果是不可预料的）。

这些转换都将作为 ALTER TABLE、LOAD DATA、UPDATE 和多行 INSERT 语句的警告信息报告。在 mysql 客户机中，这些信息显示在查询报告的状态行上。在编程语言中，可通过某些其他手段取得这个信息。如果使用的是 MySQL C API，那么可调用 mysql_info() 函数来获得这个信息。对于 Perl DBI API，可利用数据库连接的 mysql_info 属性。所提供的这个信息是警告信息的次数计数。为了知道更改了哪些行，可发布一条 SELECT ... INTO OUTFILE 查询，并将结果与原始行进行比较。

第3章 MySQL SQL 语法及其用法

为了与MySQL 服务器进行通信，必须熟练掌握 SQL。例如，在使用诸如 mysql 客户机这样的程序时，其功能首先是作为一种发送 SQL 语句给服务器执行的工具。而且，如果编写使用编程语言所提供的 MySQL 接口的程序时，也必须熟悉 SQL 语言，因为需要发送 SQL 语句与服务器沟通。

第1章“MySQL与SQL介绍”的教程介绍了许多 MySQL 功能。本章在该教程的基础上进一步对 MySQL的SQL的几个方面进行研究。它讨论了怎样引用数据库的要素，包括命名规则以及区分大小写约束的适用性。它还介绍了许多更为重要的 SQL 语句，诸如创建和删除数据库、表和索引的语句；利用连接检索数据的语句；提供关于数据库和表的信息的语句等。这里的介绍还强调了 MySQL 对标准 SQL 进行的某些扩充。

3.1 MySQL 中的SQL特征

MySQL 的 SQL 语句可分为几大类，如图3-1 所示。我们将在本章中介绍图 3-1中所示的前四类。MySQL的一些实用工具提供了与某些 SQL 语句的基本命令行接口的机制。例如，mysqlshow 就是 SHOW COLUMNS 语句的一个接口。本章中适当的地方也对这些等效的东西进行介绍。

未在本章介绍的一些语句将在其他章中介绍。例如，用于设置用户权限的 GRANT 和 REVOKE 语句在第11章“常规的MySQL 管理”中介绍。所有语句的引用语法在附录 D“SQL 语法参考”中列出。此外，还可以参看MySQL 参考指南（MySQL Reference Manual）以获得其他信息，特别是获得MySQL 最新版本中所作更改的信息。

本章最后一节介绍 MySQL缺少的功能，即一些其他数据库中有的而 MySQL 中无的功能。例如子选择、事务处理、引用完整性、触发器、存储过程以及视图。缺少这些功能是否意味着MySQL 不是一个“真正”的数据库系统？有些人是这样认为的，但据笔者的看法，这些功能的缺乏并未阻止大量人员使用它。这大概是因为，对于大多数应用来

创建、丢弃和选择数据库 CREATE DATABASE DROP DATABASE USE
创建、更改和丢弃表和索引 ALTER TABLE CREATE INDEX CREATE TABLE DROP INDEX DROP TABLE
取数据库、表和查询的有关信息 DESCRIBE EXPLAIN SHOW
从表中选择信息 SELECT
修改表中信息 DELETE INSERT LOAD DATA OPTIMIZE TABLE REPLACE UPDATE
管理语句 FLUSH GRANT KILL REVOKE
其他语句 CREATE FUNCTION DROP FUNCTION LOCK TABLES SET UNLOCK TABLES

图3-1 MySQL 支持的 SQL 语句

说，缺这些功能没什么关系。而其他一些需要这些功能的场合，也有相应的解决办法。例如，缺少级联删除表示从表中删除记录时，可能需要发布一条额外的查询。如果发现利用 `LOCK TABLES` 与 `UNLOCK TABLES` 语句，将各语句分为不中断执行组的 MySQL 功能已经足够，那么缺少事务处理支持对你来说可能不会产生什么影响。

（这里真正的问题不是缺少事务处理；而是自动回退以取消失败的语句。如果有一些应用具有复杂的财务事务处理，比如需要完成涉及必须作为一个组执行的几个互锁语句的处理，那么可能会考虑使用具有提交/回退能力的数据库，如使用 Postgres。）某些缺少的功能将在未来实现。如，MySQL 不支持子查询，但已计划在版本 3.24 中给出，或许您读到本书时它已经实现了。

3.2 MySQL 的命名规则

几乎每条 SQL 语句都在某种程度上涉及一个数据库或其组成成分。本节介绍引用数据库、表、列、索引和别名的语法规则。名称是区分大小写的，这里也对其进行了介绍。

3.2.1 引用数据库的成分

在用名称引用数据库的成分时，受到可使用的字符以及名称可具有的长度的限制。名称的形式还依赖于使用它们的上下文环境：

- **名称中可用的字符。**名称可由服务器所采用的字符集中任意字母、数字、“`_`”和“`$`”组成。名称可按上述任意字符包括数字起头。但是名称不能单独由数字组成，因为那样会使其与数值相混。MySQL 所提供的名称用一个数起始的能力是很不寻常的。如果使用了这样的一个名称，要特别注意包含“`E`”和“`e`”的名称，因为这两个字符可能会导致与表达式的混淆。`23e + 14` 表示列 23e 加 14，但是 `23e+14` 又表示什么？它表示一个科学表示法表示的数吗？

- **名称的长度。**数据库、表、列和索引的名称最多可由 64 个字符组成。别名最多可长达 256 个字符。

- **名称限定词。**为了引用一个数据库，只要指定其名称即可，如：

```
USE db_name
```

```
SHOW TABLES FROM db_name
```

其中 `db_name` 为所要引用的数据库名。要想引用一个表，可有两种选择。一种选择是使用由数据库名和表名组成的完全限定的表名，例如：

```
SHOW TABLES FROM db_name.tbl_name
```

```
SELECT * FROM db_name.tbl_name
```

其中，`tbl_name` 为要引用的表名。另一种选择是由表名自身来引用缺省（当前）数据库中的一个表。如果 `samp_db` 为缺省数据库中的一个表，下面的两个语句是等价的：

```
SELECT * FROM member
```

```
SELECT * FROM samp_db.member
```

其中 `member` 为数据库 `samp_db` 中的一个表。要引用一个列，有三种选择，它们分别为：完全限定、部分限定和非限定。完全限定名（如 `db_name.tbl_name.col_name`）是完全地指定。部分限定名（如 `tbl_name.col_name`）引用指定表中的列。非限定名（如 `col_name`）引用由环境上下文给出的表中的列。下面两个查询使用了相同的列名，但是 `FROM` 子句提供的上下文指定了从哪个表中选择列：

```
SELECT last_name, first_name FROM president  
SELECT last_name, first_name FROM members
```

虽然愿意的话，提供完全限定名也是合法的，但是一般不需要提供完全限定名，如果用 USE 语句选择了一个数据库，则该数据库将成为缺省数据库并在每一个非限定表引用中都隐含指向它。如果正使用一条 SELECT 语句，此语句只引用了一个表，那么该语句中的每个列引用都隐含指向这个表。只在所引用的表或数据库不能从上下文中确定时，才需要对名称进行限定。下面是一些会出现混淆的情形：

- 从多个数据库中引用表的查询。任何不在缺省数据库中的表都必须用“数据库名.表名”的形式引用，以便让 MySQL 知道在哪个数据库中找到该表。
- 从多个表中选择一列的查询，其中不止一个表含有具有该名称的列。

3.2.2 SQL 语句中的大小写规则

SQL 中的大小写规则在语句的不同部分是不同的，而且还取决于所引用的东西以及运行的操作系统。下面给出相应的说明：

- **SQL 关键字和函数名。**关键字与函数名是不区分大小写的。可按任意的大小写字符给出。下面的三条语句是等价的：

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

- **数据库与表名。**MySQL 中数据库和表名对应于服务器主机上的基本文件系统上的目录和文件。因此，数据库与表名是否区分大小写取决于主机上的操作系统处理文件名的方式。运行在 UNIX 上的服务器处理数据库名和表名是区分大小写的，因为 UNIX 的文件名是区分大小写的。而 Windows 文件名是不区分大小写的，所以运行在 Windows 上的服务器处理数据库名和表名也是不区分大小写的。

如果在 UNIX 服务器上创建一个某天可能会移到 Windows 服务器上的数据库，应该意识到这个特性：如果现在创建了两个分别名为 abc 和 ABC 的表，它们在 Windows 机器上将是没有区别的。避免这种情况发生的一种方法是选择一种字符（如小写），总是以这种字符创建数据库和表名。这样，在将数据库移到不同的服务器时，名称的大小写便不会产生问题。

- **列与索引名。**MySQL 中列和索引名是不区分大小写的。下面的查询都是等价的：

```
SELECT name FROM student  
SELECT NAME FROM student  
SELECT nAmE FROM student
```

- **别名。**别名是区分大小写的。可按任意的大小写字符说明一个别名（大写、小写或大小写混合），但是必须在任何查询中都以相同的大小写对其进行引用。

不管数据库、表或别名是否是区分大小写的，在同一个查询中的任何地方引用同一个名称都必须使用相同的大小写。对于 SQL 关键字、函数名列名和索引名没有这个要求。可在同一个查询中多个地方用不同的大小写对它们进行引用。当然，如果使用一致的大小写而不是“胡乱写”的风格（如 SelEcT NamE FrOm ...），相应的查询可读性要强得多。

3.3 创建、删除和选择数据库

MySQL 提供了三条数据库级的语句，它们分别是：CREATE DATABASE 用于创建数据

库，DROP DATABASE 用于删除数据库，USE 用于选择缺省数据库。

1. CREATE DATABASE 语句

创建一个数据库很容易；只要在 CREATE DATABASE 语句中给出其名称即可：

```
CREATE DATABASE db_name
```

其中限制条件是该数据库的名称必须是合法的，该数据库必须不存在，并且您必须有足够的权限来创建它。

2. DROP DATABASE 语句

删除数据库就像创建它一样容易，假如有权限，执行下列语句即可：

```
DROP DATABASE db_name
```

请注意，不要乱用 DROP DATABASE 语句，它将会删除数据库及其所有的表。在删除了一个数据库后，该数据库就永远没有了。换句话说，不要仅为了看看这条语句如何工作就试着执行该语句。如果管理员已经正常完成了数据库备份，那么删除的数据库可能还可以恢复。

请注意，数据库是由数据目录中的一个目录表示的。如果在该目录中放置了一些非表的数据文件，它们是不会被 DROP DATABASE 语句删除的。此时，该数据库目录自身也不被删除。

3. USE 语句

USE 语句选择一个数据库，使其成为服务器的给定连接的缺省（当前）数据库：

```
USE db_name
```

必须对数据库具有某种访问权限，否则不能使用它。为了使用数据库中的表而选择该数据库实际上不是必须的，因为可以利用 db_name.tbl_name 形式来引用它的表。但是，不必指定数据库限定词引用表要方便得多。

选择一个缺省数据库并不代表在连接的持续时间内它都必须是缺省的。可发布任意数目的 USE 语句在数据库之间进行任意地切换，只要具有使用它们的权限即可。选择一个数据库也不限制您只使用该数据库中的表。您仍然可以通过用数据库名限定表名的方法，引用其他数据库中的表。

在服务器的连接终止时，服务器关于缺省数据库的所有记忆都消失了。即，如果您再次连接到该服务器，它不会记住以前您所选择的数据库。事实上，假定 MySQL 是多线程的，可通过一个用户处理多个连接，用户可以按任何顺序连接或断开，让服务器对缺省数据库进行记忆的想法也是没有意义的。在这个环境中，“以前选择的数据库”这句话指什么并不清楚。

3.4 创建、删除、索引和更改表

可利用 CREATE TABLE、DROP TABLE 和 ALTER TABLE 语句创建表，然后，对它们进行删除，更改它们的结构。对于它们中的每一条语句，存在 MySQL 专有的扩充，这些扩充使各语句更为有用。CREATE INDEX 和 DROP INDEX 语句使您能够增加或删除现有表上的索引。

3.4.1 CREATE TABLE 语句

用 CREATE TABLE 语句创建表。此语句的完整语法是相当复杂的，因为存在那么多的

可选子句，但在实际中此语句的应用相当简单。如我们在第 1 章中使用的所有 CREATE TABLE 语句都不那么复杂。

有意思的是，大多数复杂东西都是一些子句，这些子句 MySQL 在分析后扔掉。参阅附录 D 可看到这些复杂的东西。看看 CREATE TABLE 语句的各项条款，注意该语句有多少语法是用于 REFERENCES、CONSTRAINT 和 CHECK 子句的。这些子句涉及外部键、引用完整性及输入值约束。MySQL 不支持这些功能，但它分析其语法使其更容易利用在其他数据库系统中建立的表定义。（可以用较少的编辑工作更容易地利用该代码。）如果您从头开始编写自己的表描述，可以完全不管这些子句。本节中我们对它们也不多做介绍。

CREATE TABLE 至少应该指出表名和表中列的清单。例如：

```
CREATE TABLE my_table
(
    name CHAR(20),
    age INT NOT NULL,
    weight INT,
    sex ENUM('F','M')
)
```

除构成表的列以外，在创建表时还可以说明它应该怎样索引。另一个选择是创建表时不进行索引，以后再增加索引。如果计划在开始将表用于查询前，用大量的数据填充此表，以后再创建索引是一个好办法。在插入每一行时更新索引较装载数据到一个未索引的表中然后再创建索引要慢得多。

我们已经在第1章中介绍了 CREATE TABLE 语句的基本语法，并在第2章讨论了怎样描述列类型。这里假定您已经读过了这两章，因此我们就不重复这些内容了。在本节下面，我们将介绍一些 MySQL 3.23 中对 CREATE TABLE 语句的重要扩充，这些扩充在构造表方面提供了很大的灵活性，这些扩充为：

- 表存储类型说明符。
- 仅当表不存在时才进行创建。
- 在客户机会话结束时自动删除临时表。
- 通过选择希望表存储的数据来创建一个表。

1. 表存储类型说明符

在 MySQL 3.23 之前，所有用户创建的表都利用的是 ISAM 存储方法。在 MySQL 3.23 中，可在 CREATE TABLE 语句的列的列表之后指定 TYPE = type，以三种类型明确地创建表。其中 type 可以为 MYISAM、ISAM 或 HEAP。例如：

```
CREATE TABLE my_tbl (i INT, c CHAR(20)) TYPE = HEAP
```

可用 ALTER TABLE 将表从一种类型转换到另一种类型：

```
ALTER TABLE my_tbl TYPE = ISAM
ALTER TABLE my_tbl TYPE = MYISAM
ALTER TABLE my_tbl TYPE = HEAP
```

将表转换为 HEAP 类型可能不是一个好主意，但是，如果希望表一直维持到服务器关闭，可以进行这个转换。HEAP 表在服务器退出之前，一直保留在内存中。

这三种表类型的一般特点如下：

- MyISAM 表。MyISAM 存储格式自版本 3.23 以来是 MySQL 中的缺省类型，它有下列特点：

- 如果操作系统自身允许更大的文件，那么文件比 ISAM 存储方法的大。
- 数据以低字节优先的机器独立格式存储。这表示可将表从一种机器拷贝到另一种机器，即使它们的体系结构不同也可以拷贝。
- 数值索引值占的存储空间较少，因为它们是按高字节优先存储的。索引值在低位字节中变化很快，因此高位字节更容易比较。
- AUTO_INCREMENT 处理比 ISAM 的表更好。详细内容在第2章讨论。
- 减少了几个索引限制。例如，可对含 NULL 值的列进行索引，还可以对 BLOB 和 TEXT 类型的列进行索引。
- 为了改善表的完整性检查，每个表都具有一个标志，在 myisamchk 对表进行过检查后，设置该标志。可利用 myisamchk - fast 跳过对自前次检查以来尚未被修改过表的检查，这样使此管理任务更快。表中还有一个指示表是否正常关闭的标志。如果服务器关闭不正常，或机器崩溃，此标志可用来检测出服务器启动时需要检查的表。
- ISAM 表。ISAM 存储格式是 MySQL 3.23 所用的最旧的格式，但当前仍然可用。通常，相对于 ISAM 表来说，宁可使用 MyISAM 表，因为它们的限制较少。对 ISAM 表的支持随着此存储格式被 MyISAM 表格式所支持很有可能会逐渐消失。
- HEAP 表。HEAP 存储格式建立利用定长行的内存中的表，这使表运行得非常快。在服务器停止时，它们将会消失。在这种意义上，这些表是临时的。但是，与用 CREATE TEMPORARY TABLE 所创建的临时表相比，HEAP 表是其他客户机可见的。HEAP 表有几个限制，这些限制对 MyISAM 或 ISAM 表没有，如下所示：
 - 索引仅用于“=”和“<=>”比较。
 - 索引列中不能有 NULL 值。
 - 不能使用 BLOB 和 TEXT 列。
 - 不能使用 AUTO_INCREMENT 列。

2. 创建不存在的表

要创建一个不存在的表，使用 CREATE TABLE IF NOT EXISTS 即可。在某种应用程序中，无法确定要用的表是否已经存在，因此，要创建这种表。IF NOT EXISTS 修饰符对于作为用 mysql 运行的批量作业的脚本极为有用。在这里，普通的 CREATE TABLE 语句工作得不是很好。因为作业第一次运行时，建立这些表，如果这些表已经存在，则第二次运行时将出错。如果用 IF NOT EXISTS 语句，就不会有问题。每一次运行作业时，像前面一样创建表。如果这些表已经存在，在第二次运行时，创建表失败，但不出错。这使得作业可以继续运行，就像创建表的企图已经成功了一样。

3. 临时表

可用 CREATE TEMPORARY TABLE 来创建临时表，这些表在会话结束时会自动消失。使用临时表很方便，因为不必费心发布 DROP TABLE 语句明确地删除这些表，而且如果您的会话不正常结束，这些表不会滞留。例如，如果某个文件中有一个用 mysql 运行的查询，您决定不等到其结束，那么可以在其执行的中途停止这个查询，而且毫无问题，服务器将删除所创建的任意临时表。

在旧版的 MySQL 中，没有真正的临时表，除了您在自己的头脑中认为它们是临时的除外。对于需要这样的表的应用程序，必须自己记住删除这些表。如果忘了删除，或在前面使

其存在的客户机中出现错误时，这些表在有人注意到并删除它们以前会一直存在。

临时表仅对创建该表的客户机可见。其名称可与一个现有的永久表相同。这不是错误，也不会使已有的永久表出问题。假如在 samp_db 数据库中创建了一个名为 member 的临时表。原来的 member 表变成隐藏的（不可访问），对 member 的引用将引用临时表。如果发布一条 DROP TABLE member 语句，这个临时表将被删除，而原来的 member 表“重新出现”。如果您简单地中断与服务器的连接而没有删除临时表，服务器会自动地删除它。下一次连接时，原来的 member 表再次可见。

名称隐藏机制仅在一个级别上起作用。即，不能创建两个具有同一个名称的临时表。

4. 利用 SELECT 的结果创建表

关系数据库的一个重要概念是，任何数据都表示为行和列组成的表，而每条 SELECT 语句的结果也都是一个行和列组成的表。在许多情况下，来自 SELECT 的“表”仅是一个随着您的工作在显示屏上滚动的行和列的图像。在 MySQL 3.23 以前，如果想将 SELECT 的结果保存在一个表中以便以后的查询使用，必须进行特殊的安排：

1) 运行 DESCRIBE 或 SHOW COLUMNS 查询以确定想从中获取信息的表中的列类型。

2) 创建一个表，明确地指定刚才查看到的列的名称和类型。

3) 在创建了该表后，发布一条 INSERT ... SELECT 查询，检索出结果并将它们插入所创建的表中。

在 MySQL 3.23 中，全都作了改动。CREATE TABLE ... SELECT 语句消除了这些浪费时间东西，使得能利用 SELECT 查询的结果直接得出一个新表。只需一步就可以完成任务，不必知道或指定所检索的列的数据类型。这使得很容易创建一个完全用所喜欢的数据填充的表，并且为进一步查询作了准备。

可以通过选择一个表的全部内容（无 WHERE 子句）来拷贝一个表，或利用一个总是失败的 WHERE 子句来创建一个空表，如：

```
CREATE TABLE new_tbl_name SELECT * FROM tbl_name
CREATE TABLE new_tbl_name SELECT * FROM tbl_name WHERE 1 = 0
```

如果希望利用 LOAD DATA 将一个数据文件装入原来的文件中，而不敢肯定是否具有指定的正确数据格式时，创建空拷贝很有用。您并不希望在第一次未得到正确的选项时以原来表中畸形的记录而告终。利用原表的空拷贝允许对特定的列和行分隔符用 LOAD DATA 的选项进行试验，直到对输入数据的解释满意时为止。在满意之后，就可以将数据装入原表了。

可结合使用 CREATE TEMPORARY TABLE 与 SELECT 来创建一个临时表作为它自身的拷贝，如：

```
CREATE TEMPORARY TABLE my_tbl SELECT * FROM my_tbl
```

这允许修改 my_tbl 的内容而不影响原来的内容。在希望试验对某些修改表内容的查询，而又不想更改原表内容时，这样做很有用。为了使用利用原表名的预先编写的脚本，不需要为引用不同的表而编辑这些脚本；只需在脚本的起始处增加 CREATE TEMPORARY TABLE 语句即可。相应的脚本将创建一个临时拷贝，并对此拷贝进行操作，当脚本结束时服务器会自动删除这个拷贝。

要创建一个作为自身的空拷贝的表，可以与 CREATE TEMPORARY ... SELECT 一起使用 WHERE 0 子句，例如：

```
CREATE TEMPORARY TABLE my_tbl SELECT FROM my_tbl WHERE 1 = 0
```

但创建空表时有几点要注意。在创建一个通过选择数据填充的表时，其列名来自所选择的列名。如果某个列作为表达式的结果计算，则该列的“名称”为表达式的文本。表达式不是合法的列名，可在 mysql 中运行下列查询了解这一点：

```
mysql> CREATE TABLE my_tbl SELECT 1;  
ERROR 1166 at line 1: Incorrect column name '1'
```

为了正常工作，可为该列提供一个合法的别称：

```
mysql> CREATE TABLE my_tbl SELECT 1 AS my_col;  
Query OK, 1 row affected (0.01 sec)
```

如果选择了来自不同表的具有相同名称的列，将会出现一定的困难。假定表 t1 和 t2 两者都具有列 c，而您希望创建一个来自两个表中行的所有组合的表。那么可以提供别名指定新表中惟一性的列名，如：

```
CREATE TABLE t3 SELECT t1.c AS c1, t2.c AS c2 FROM t1, t2;
```

通过选择数据进行填充来创建一个表并会自动拷贝原表的索引。

3.4.2 DROP TABLE 语句

删除表比创建表要容易得多，因为不需要指定有关其内容的任何东西；只需指定其名称即可，如：

```
DROP TABLE tbl_name
```

MySQL 对 DROP TABLE 语句在某些有用的方面做了扩充。首先，可在同一语句中指定几个表对它们进行删除，如：

```
DROP TABLE tbl_name1, tbl_name2, ...
```

其次，如果不能肯定一个表是否存在，但希望如果它存在就删除它。那么可在此语句中增加 IF EXISTS。这样，如果 DROP TABLE 语句中给出的表不存在，MySQL 不会发出错误信息。如：

```
DROP TABLE IF EXISTS tbl_name
```

IF EXISTS 在 mysql 所用的脚本中很有用，因为缺省情况下，mysql 将在出错时退出。例如，有一个安装脚本能够创建表，这些表将在其他脚本中继续使用。在此情形下，希望保证此创建表的脚本在开始运行时无后顾之忧。如果在该脚本开始处使用普通的 DROP TABLE，那么它在第一次运行时将会失败，因为这些表从未创建过。如果使用 IF EXISTS，就不会产生问题了。当表已经存在时，将它们删除；如果不存在，脚本继续运行。

3.4.3 创建和删除索引

索引是加速表内容访问的主要手段，特别对涉及多个表的连接的查询更是如此。这是第 4 章“查询优化”中的一个重要内容，第 4 章讨论了为什么需要索引，索引如何工作以及怎样利用它们来优化查询。本节中，我们将介绍索引的特点，以及创建和删除索引的语法。

1. 索引的特点

MySQL 对构造索引提供了很大的灵活性。可对单列或多列的组合进行索引。如果希望能从一个表的不同列中找出一个值，还可以在一个表上构造不止一个索引。如果某列为串类型而非 ENUM 或 SET 类型，可以选择只对该列最左边的 n 个字符进行索引。如果该列的前 n 个字符最具有唯一性，这样做一般不会牺牲性能，而且还会对性能有大的改善：用索引列的

前缀而非整个列可使索引更小且访问更快。

虽然随着 MySQL 的进一步开发创建索引的约束将会越来越少，但现在还是存在一些约束的。下面的表根据索引的特性，给出了 ISAM 表和 MyISAM 表之间的差别：

索引的特点	ISAM 表	MyISAM 表
NULL 值	不允许	允许
BLOB 和 TEXT 列	不能索引	能索引
每个表中的索引数	16	32
每个索引中的列数	16	16
最大索引行尺寸	256 字节	500 字节

从此表中可以看到，对于 ISAM 表来说，其索引列必须定义为 NOT NULL，并且不能对 BLOB 和 TEXT 列进行索引。MyISAM 表类型去掉了这些限制，而且减缓了其他的一些限制。两种表类型的索引特性的差异表明，根据所使用的 MySQL 版本的不同，有可能对某些列不能进行索引。例如，如果使用 3.23 版以前的版本，则不能对包含 NULL 值的列进行索引。

如果使用的是 MySQL 3.23 版或更新的版本，但表是过去以 ISAM 表创建的，可利用 ALTER TABLE 很方便地将它们转换为 MyISAM 存储格式，这样使您能利用某些较新的索引功能，如：

```
ALTER TABLE tbl_name TYPE = MYISAM
```

2. 创建索引

在执行 CREATE TABLE 语句时，可为新表创建索引，也可以用 CREATE INDEX 或 ALTER TABLE 来为一个已有的表增加索引。CREATE INDEX 是在 MySQL 3.23 版中引入的，但如果使用 3.23 版以前的版本，可利用 ALTER TABLE 语句创建索引（MySQL 通常在内部将 CREATE INDEX 映射到 ALTER TABLE）。

可以规定索引能否包含重复的值。如果不包含，则索引应该创建为 PRIMARY KEY 或 UNIQUE 索引。对于单列惟一索引，这保证了列不包含重复的值。对于多列惟一索引，它保证值的组合不重复。

PRIMARY KEY 索引和 UNIQUE 索引非常类似。事实上，PRIMARY KEY 索引仅是一个具有名称 PRIMARY 的 UNIQUE 索引。这表示一个表只能包含一个 PRIMARY KEY，因为一个表中不可能具有两个同名的索引。同一个表中可有多多个 UNIQUE 索引，虽然这样做意义不大。

为了给现有的表增加一个索引，可使用 ALTER TABLE 或 CREATE INDEX 语句。ALTER TABLE 最常用，因为可用它来创建普通索引、UNIQUE 索引或 PRIMARY KEY 索引，如：

```
ALTER TABLE tbl_name ADD INDEX index_name (column_list)
ALTER TABLE tbl_name ADD UNIQUE index_name (column_list)
ALTER TABLE tbl_name ADD PRIMARY KEY (column_list)
```

其中 tbl_name 是要增加索引的表名，而 column_list 指出对哪些列进行索引。如果索引由不止一列组成，各列名之间用逗号分隔。索引名 index_name 是可选的，因此可以不写它，MySQL 将根据第一个索引列赋给它一个名称。ALTER TABLE 允许在单个语句中指定多个表的更改，因此可以在同时创建多个索引。

CREATE INDEX 可对表增加普通索引或 UNIQUE 索引，如：

```
CREATE UNIQUE INDEX index_name ON tbl_name (column_list)
CREATE INDEX index_name ON tbl_name (column_list)
```

tbl_name、index_name 和 column_list 具有与 ALTER TABLE 语句中相同的含义。这里索引名不可选。不能用 CREATE INDEX 语句创建 PRIMARY KEY 索引。

要想在发布 CREATE TABLE 语句时为新表创建索引，所使用的语法类似于 ALTER TABLE 语句的语法，但是应该在您定义表列的语句部分指定索引创建子句，如下所示：

```
CREATE TABLE tbl_name
(
    ...
    INDEX index_name (column_list),
    UNIQUE index_name (column_list),
    PRIMARY KEY (column_list),
    ...
)
```

与 ALTER TABLE 一样，索引名对于 INDEX 和 UNIQUE 都是可选的，如果未给出，MySQL 将为其选一个。

有一种特殊情形：可在列定义之后增加 PRIMARY KEY 创建一个单列的 PRIMARY KEY 索引，如下所示：

```
CREATE TABLE my_tbl
(
    i INT NOT NULL PRIMARY KEY
)
```

该语句等价于以下的语句：

```
CREATE TABLE my_tbl
(
    i INT NOT NULL,
    PRIMARY KEY (i)
)
```

前面所有表创建样例都对索引列指定了 NOT NULL。如果是 ISAM 表，这是必须的，因为不能对可能包含 NULL 值的列进行索引。如果是 MyISAM 表，索引列可以为 NULL，只要该索引不是 PRIMARY KEY 索引即可。

如果对某个串列的前缀进行索引（列值的最左边 n 个字符），应用 column_list 说明符表示该列的语法为 col_name(n) 而不用 col_name。例如，下面第一条语句创建了一个具有两个 CHAR 列的表和一个由这两列组成的索引。第二条语句类似，但只对每个列的前缀进行索引：

```
CREATE TABLE my_tbl
(
    name CHAR(30),
    address CHAR(60),
    INDEX (name,address)
)

CREATE TABLE my_tbl
(
    name CHAR(30),
    address CHAR(60),
    INDEX (name(10),address(20))
)
```

在某些情况下，可能会发现必须对列的前缀进行索引。例如，索引行的长度有一个最大上限，因此，如果索引列的长度超过了这个上限，那么就可能需要利用前缀进行索引。在

MyISAM 表索引中，对 BLOB 或 TEXT 列也需要前缀索引。

对一个列的前缀进行索引限制了以后对该列的更改；不能在不删除该索引并使用较短前缀的情况下，将该列缩短为一个长度小于索引所用前缀的长度的列。

3. 删除索引

可利用 DROP INDEX 或 ALTER TABLE 语句来删除索引。类似于 CREATE INDEX 语句，DROP INDEX 通常在内部作为一条 ALTER TABLE 语句处理，并且 DROP INDEX 是在 MySQL 3.22 中引入的。删除索引语句的语法如下：

```
DROP INDEX index_name ON tbl_name  
ALTER TABLE tbl_name DROP INDEX index_name  
ALTER TABLE tbl_name DROP PRIMARY KEY
```

前两条语句是等价的。第三条语句只在删除 PRIMARY KEY 索引时使用；在此情形中，不需要索引名，因为一个表只可能具有一个这样的索引。如果没有明确地创建作为 PRIMARY KEY 的索引，但该表具有一个或多个 UNIQUE 索引，则 MySQL 将删除这些 UNIQUE 索引中的第一个。

如果从表中删除了列，则索引可能会受到影响。如果所删除的列为索引的组成部分，则该列也会从索引中删除。如果组成索引的所有列都被删除，则整个索引将被删除。

3.4.4 ALTER TABLE 语句

ALTER TABLE 语句是 MySQL 中一条通用的语句，可用它来做许多事情。我们已经看过了它的几种功能（创建和删除索引以及将表从一种存储格式转换为另一种存储格式）。本节中，我们将介绍它的一些其他功能。ALTER TABLE 的完整语法在附录 D 中介绍。

在发现某个表的结构不再反映所希望的东西时，ALTER TABLE 很有用处。可能希望用该表记录其他信息，或者它含有多余的值。或者有的列太小，或者其定义较实际需要来说太大，需要将它们改小以节省存储空间。或者发布 CREATE TABLE 语句时给出的表名不对。等等，诸如此类的问题，都可以用 ALTER TABLE 语句来解决。下面是一些例子：

- 您正操纵一个基于 Web 的问卷，将每份提交的问卷作为表中的一个记录。后来决定修改此问卷，增加一些问题。这时必须对表增加一些列以存放新问题。
- 您正在管理一个研究项目。用 AUTO_INCREMENT 列分配案例号来研究记录。您不希望经费延期太长产生多于 50 000 个以上的记录，因此，令该列的类型为 UNSIGNED SMALLINT，它能存储的最大惟一值为 65 535。但是，项目的经费延长了，似乎可能另外产生 50 000 个记录。这时，需要使该列的类型更大一些以便存储更多的件号。
- 大小的更改也可能是反方向的。可能创建了一个 CHAR(255) 列，但现在发现表中没有比 100 个字符更长的串。这时可缩短该列以节省存储空间。

ALTER TABLE 的语法如下：

```
ALTER TABLE tbl_name action, ...
```

每个 action 表示对表所做的一个修改。MySQL 扩充了 ALTER TABLE 语句，允许指定多个动作，各动作间以逗号分隔。这对于减少键盘输入很有用，但这个扩充的更为重要的原因是，除非能同时将所有 VARCHAR 列更改为 CHAR 列，否则不可能将表从行可变长的表更改为行定长的表。

下面的例子示出了某些 ALTER TABLE 的功能。

■ **对表重新命名。**这很简单；只需给出旧表名和新表名即可：

```
ALTER TABLE tbl_name RENAME AS new_tbl_name
```

在 MySQL 3.23 中有临时表，重命名一个临时表为数据库中已经存在的名称将隐藏原始表，只要临时表存在就会隐藏原始表。这类似于通过用相同的名字创建一个临时表来隐藏一个表的方法。

■ **更改列类型。**为了更改列的类型，可使用 CHANGE 或 MODIFY 子句。假如表 my_tbl 中的列为 SMALLINT UNSIGNED 的，希望将其更改为 MEDIUMINT UNSIGNED 的列。用下面的任何一个命令都可完成此项工作：

```
ALTER TABLE my_tbl MODIFY i MEDIUMINT UNSIGNED
ALTER TABLE my_tbl CHANGE i i MEDIUMINT UNSIGNED
```

为什么在 CHANGE 命令中给出列名两次？因为 CHANGE 可以做的而 MODIFY 不能做的一桩事是，除了更改类型外还能更改列名。如果希望在更改类型的同时重新将 i 命名为 j，可按如下进行：

```
ALTER TABLE my_tbl CHANGE i j MEDIUMINT UNSIGNED
```

重要的是命名了希望更改的列，并说明了一个包括列名的列的完整定义。即使不更改列名，也需要在定义中包括相应的列名。

更改列类型的一个重要原因是为了改善比较两个表的连接查询的效率。在两个列的类型相同时，比较更快。假如执行如下的查询：

```
SELECT ... FROM t1, t2 WHERE t1.name = t2.name
```

如果 t1.name 为 CHAR(10)，而 t2.name 为 CHAR(15)，此查询的运行速度没有它们两者都为 CHAR(15) 时的快。那么可以用下面的任一条命令更改 t1.name 使它们的类型相同：

```
ALTER TABLE t1 MODIFY name CHAR(15)
ALTER TABLE t1 CHANGE name name CHAR(15)
```

对于 3.23 以前的 MySQL 版本，所连接的列必须是同样类型的这一点很重要，否则索引不能用于比较。对于版本 3.23 或以上的版本，索引可用于不同的类型，但如果类型相同，查询仍然更快。

■ **将表从可变长行转换为定长行。**假如有一个表 chartbl 具有 VARCHAR 列，想要把它转换为 CHAR 列，看看能够得到什么样的性能改善。（定长行的表一般比变长行的表处理更快。）这个表如下创建：

```
CREATE TABLE chartbl (name VARCHAR(40), address VARCHAR(80))
```

这里的问题是需要相同的 ALTER TABLE 语句中一次更改所有的列。不可能一次一列地改完，或者说这个企图将不起作用。如果执行 DESCRIBE chartbl，会发现两个列仍然是 VARCHAR 的列！原因是如果每次更改一列，MySQL 注意到表仍然包含有可变长的列，则会把已经更改过的列重新转换为 VARCHAR 以节省空间。为了处理这个问题，应该同时更改所有 VARCHAR 列：

```
ALTER TABLE chartbl MODIFY name CHAR(40), MODIFY address CHAR(80)
```

现在 DESCRIBE 将显示该表包含的都是 CHAR 列。确实，这种类型的操作很重要，因为它使 ALTER TABLE 能在相同的语句中支持多个动作。

这里要注意，在希望转换这样的表时：如果表中存在 BLOB 或 TEXT 列将使转换表为定长行格式的企图失败。即使表中只有一个可变长的列都将会使表有可变长的行，

因为这些可变长的列类型没有定长的等价物。

- **将表从定长行转换为可变长的行。**虽然，`chartbl` 用定长行更快，但它要占用更多的空间，因此决定将它转换回原来的形式以节省空间。这种转换更为容易。只需将某个 `CHAR` 列转换为 `VARCHAR` 列，MySQL 就自动地转换其他的 `CHAR` 列。要想转换 `chartbl` 表，用下列任一条语句都可以：

```
ALTER TABLE chartbl MODIFY name VARCHAR(40)
ALTER TABLE chartbl MODIFY address VARCHAR(80)
```

- **转换表的类型。**如果从 MySQL 3.23 版以前的版本升级到 3.23 版或更高，那么可能会有一些原来创建为 `ISAM` 表的旧表。如果希望使它们为 `MyISAM` 格式，如下操作：

```
ALTER TABLE tbl_name TYPE = MYISAM
```

为什么要这样做呢？正如在“创建和删除索引”小节中所介绍的那样，一个原因是 `MyISAM` 存储格式具有某些 `ISAM` 格式没有的索引特性，例如能够对 `NULL` 值、`BLOB` 和 `TEXT` 列类型进行索引。另一个原因为，`MyISAM` 表是独立于机器的，因此可通过将它们直接拷贝来将它们移到其他机器上，即使那些机器具有不同的硬件体系结构也同样。这在第 11 章中将要作进一步的介绍。

3.5 获取数据库和表的有关信息

MySQL 提供了几条获取数据库和表中信息的语句。这些语句对于了解数据库的内容及了解自己表的结构很有帮助。还可以将它们作为使用 `ALTER TABLE` 的一种辅助手段；能够知道当前列是如何定义的，计划出怎样对列进行更改会更为容易。

`SHOW` 语句可用来获取数据库和表的几个方面的信息，它有如下用法：

<code>SHOW DATABASES</code>	列出服务器上的数据库
<code>SHOW TABLES</code>	列出当前数据库中的表
<code>SHOW TABLES FROM db_name</code>	列出指定数据库中的表
<code>SHOW COLUMNS FROM tbl_name</code>	显示指定表中列的信息
<code>SHOW INDEX FROM tbl_name</code>	显示指定表中索引的信息
<code>SHOW TABLE STATUS</code>	显示缺省数据库中表的说明信息
<code>SHOW TABLE STATUS FROM db_name</code>	显示指定数据库中表的说明信息

`DESCRIBE tbl_name` 和 `EXPLAIN tbl_name` 语句与 `SHOW COLUMNS FROM tbl_name` 功能相同。

`mysqlshow` 命令提供了某些与 `SHOW` 语句相同的信息，它允许从外壳程序中取得数据库和表的信息：

<code>% mysqlshow</code>	列出服务器上的数据库
<code>% mysqlshow db_name</code>	列出指定数据库中的表
<code>% mysqlshow db_name tbl_name</code>	显示指定表中列的信息
<code>% mysqlshow -- keys db_name tbl_name</code>	显示指定表中索引的信息
<code>% mysqlshow --status db_name</code>	显示指定数据库中表的说明信息

`mysqldump` 实用程序允许以 `CREATE TABLE` 语句的形式查看表的结构。（与 `SHOW COLUMNS` 语句比较，作者认为 `mysqldump` 命令的输出结果更容易阅读，而且这个输出还显示了表中的索引。）但如果使用 `mysqldump`，要保证用 `- no - data` 选项调用它，以防被表中数据把头搞晕。

```
% mysqldump --no-data db_name tbl_name
```

对于 mysqlshow 和 mysqldump, 两者都可以指定一些有用的选项 (如 -- host) 来与不同主机上的服务器进行连接。

3.6 检索记录

除非最终检索它们并利用它们来做点事情, 否则将记录放入数据库没什么好处。这就是 SELECT 语句的用途, 即帮助取出数据。SELECT 大概是 SQL 语言中最常用的语句, 而且怎样使用它也最为讲究; 用它来选择记录可能相当复杂, 可能会涉及许多表中列之间的比较。

SELECT 语句的语法如下:

SELECT <i>selection_list</i>	选择哪些列
FROM <i>table_list</i>	从何处选择行
WHERE <i>primary_constraint</i>	行必须满足什么条件
GROUP BY <i>grouping_columns</i>	怎样对结果分组
ORDER BY <i>sorting_columns</i>	怎样对结果排序
HAVING <i>secondary_constraint</i>	行必须满足的第二条件
LIMIT <i>count</i>	结果限定

除了词 “SELECT” 和说明希望检索什么的 *column_list* 部分外, 语法中的每样东西都是可选的。有的数据库还需要 FROM 子句。MySQL 有所不同, 它允许对表达式求值而不引用任何表:

```
SELECT SQRT(POW(3,2)+POW(4,2))
```

在第1章中, 我们对 SELECT 语句下了很大的功夫, 主要集中介绍了列选择的列表和 WHERE、GROUP BY、ORDER BY、HAVING 以及 LIMIT 子句。本章中, 我们将主要精力放在 SELECT 语句中最可能令人搞不清的方面, 即连接 (join) 上。我们将介绍 MySQL 支持的连接类型、它们的含义、怎样指定它们等。这样做将有助于更有效地使用 MySQL, 因为在许多情况下, 解决怎样编写查询的关键是确定怎样将表恰当地连接在一起。还应该参阅一下本章后面3.8节“解决方案随笔”。在那一节中将会找到解决几个 SQL 问题的方案, 它们多数都涉及 SELECT 语句这样或那样的功能。

使用 SELECT 的一个问题是, 在第一次遇到一种新的问题时, 并不总是能够知道怎样编写 SELECT 查询来解决它。但在解决以后, 再遇到类似的问题时, 可利用其中的经验。SELECT 大概是过去的经验在能够有效地使用中起很大作用的语句, 这是因为使用它的方法太多的原故。

在有了一定的经验后, 可将这些经验用于新问题, 您会发现自己思考问题类似于, “噢, 是的, 它就是一个 LEFT JOIN 问题。”或者, “啊哈, 这就是一个受各对索引列制约的三路线连接。”(指出这一点, 实际上我也感到有点不愿意。听到经验有帮助, 您可能受到一定的鼓舞。另外, 考虑到您最终能那样思考问题也会令自己有点惊讶。)

下几节中介绍怎样利用 MySQL 支持的连接操作的格式, 多数例子使用了下面的两个表。它们很小, 很简单, 足以很清楚地看出每种连接的效果。

表 t1		表 t2	
+	+	+	+
i1	c1	i2	c2
+	+	+	+
1	a	2	c

2	b	3	b
3	c	4	a

3.6.1 平凡连接

最简单的连接是平凡连接（trivial join），这种连接中只指定一个表。在此情况下，行从指定的表中选择。如：

```
SELECT ... FROM t1
```

i1	c1
1	a
2	b
3	c

有的作者根本就不考虑这种 SELECT 连接的形式，仅对从两个或多个表中检索记录的 SELECT 语句使用“连接”这个术语。本人认为那只是看法不同而已。

3.6.2 全连接

如果指定多个表，将各个表名用逗号分隔，就指定了全连接。例如，如果连接两个表，来自第一个表中的每行与第二个表中每行进行组合：

```
SELECT t1.*, t2.* FROM t1, 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

全连接也称为叉连接，因为每个表的每行都与其他表中的每行交叉以产生所有可能的组合。这也就是所谓的笛卡儿积。这样连接表潜在地产生数量非常大的行，因为可能得到的行数为每个表中行数之积。三个分别含有 100、200、300 行的表的全连接将产生 $100 \times 200 \times 300 = 6$ 百万行。即使各表很小，所得到的行数也会很大。在这样的情形下，通常要使用 WHERE 子句来将结果集减少为易于管理的大小。

如果在 WHERE 子句中增加一个条件使各表在某些列上进行匹配，此连接就是所谓的等连接（equi-join），因为只选择那些在指定列中具有相等的值的行。如：

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2
```

i1	c1	i2	c2
2	b	2	c
3	c	3	b

JOIN、CROSS JOIN 和 INNER JOIN 连接类型都与“,”连接操作符意义相同。

STRAIGHT_JOIN 与全连接类似, 但各表按 FROM 子句中指定的次序进行连接。一般情况下, 在全连接中 MySQL 优化程序自身完全不考虑安排各表的顺序, 以便使记录的检索更快。在有的场合, 优化程序将作出非优化的选择, 这样将忽略 STRAIGHT_JOIN 关键字。

在 SELECT 语句中, 可在两个位置给出 STRAIGHT_JOIN。一个位置是在 SELECT 关键字与选择列表之间, 将其放在这里对语句中所有全连接具有整体作用。另一个在 FROM 子句中。下面两条语句是等价的:

```
SELECT STRAIGHT_JOIN ... FROM table1, table2, table3 ...
SELECT ... FROM table1 STRAIGHT_JOIN table2 STRAIGHT_JOIN table3 ...
```

限定列引用

SELECT 语句中列的引用必须对 FROM 子句中指定的每个表是无歧义的。如果 FROM 子句中仅指定了一个表, 则无歧义存在, 因为所有列必须是该表的列。如果指定了多个表, 只出现在一个表中的列名也是无歧义的。但是, 如果某个列名出现在多个表中, 该列的引用必须用表名来限定, 用 tbl_name.col_name 语法来表明所指的是哪个表。如果表 my_tbl1 含有列 a 和 b, 表 my_tbl2 含有列 b 和 c, 则列 a 和 c 的引用是无歧义的, 但 b 的引用必须限定为 my_tbl1.b 或 my_tbl2.b, 如:

```
SELECT a, my_tbl1.b, my_tbl2.b, c FROM my_tbl1, my_tbl2 ...
```

有时, 表名限定符还不能解决列的引用问题。例如, 如果在一个查询中多次使用一个表, 用表名限定列名没有什么用处。在此情况下, 为表达您的想法可使用别名。给表指派一个别名, 利用这个别名来引用列, 其语法为: alias_name.col_name。下面的查询将表与自身进行连接, 给表指派了一个别名, 以便应付引用列时有歧义的情况:

```
SELECT my_tbl.col1, m.col2
FROM my_tbl, my_tbl AS m
WHERE my_tbl.col1 > m.col1
```

3.6.3 左连接

等价连接只给出两个表匹配的行。左连接也给出匹配行, 但它还显示左边表中有的但在右边表中无匹配的行。对于这样的行, 从右边表中选择的列都显示为 NULL。这样, 每一行都从左边表中选出。如果右边表中有一个匹配行, 则该行被选中。如果不匹配, 行仍然被选中, 但它是一个“假”行, 其中所有列被设置为 NULL。换句话说, LEFT JOIN 强制结果集包含对应左边表中每一行的行, 而不管左边表中的行在右边表中是否有匹配的行。匹配是根据 ON 或 USING() 子句中给出的列进行的。不管所连接的列是否具有相同的名称, 都可使用 ON。如:

```
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

USING() 子句类似于 ON，但连接列的名称必须在每个表中是相同的。下面的查询将 my_tbl1.b 连接到 my_tbl2.b：

```
SELECT my_tbl1.*, my_tbl2.* FROM my_tbl1 LEFT JOIN my_tbl2 USING (b)
```

在希望只查找出现在左边表而不出现在右边表中的行时，LEFT JOIN 极为有用。可通过增加一条查询右边表中具有 NULL 值的列的 WHERE 子句来完成这项工作。

```
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

一般不用担心选择为 NULL 的列，因为没有什么意思。真正要关心的是左边表中不匹配的列，如：

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2 WHERE t2.i2 IS NULL
```

i1	c1
1	a

利用 LEFT JOIN 时有一件事情需要提防，如果所连接的列未定义为 NOT NULL，将会在结果中得出一些无关的行。

LEFT JOIN 有几个同义词和变种。LEFT OUTER JOIN 为 LEFT JOIN 的一个同义词。LEFT JOIN 还有一个为 MySQL 所接受的 ODBC 表示如下（“oj”意为“outer join”）：

```
{ oj tbl_name LEFT OUTER JOIN tbl_name ON join_expr }
```

NATURAL LEFT JOIN 类似于 LEFT JOIN；它执行一个 LEFT JOIN，匹配左边表和右边表中具有相同名称的所有列。

有的数据库还有，RIGHT JOIN，但 MySQL 迄今还没有。

3.7 加注释

MySQL 允许在 SQL 代码中使用注释。这对于说明存放在文件中的查询很有用处。可用两种方式编写注释。以“#”号开头直到行尾的所有内容都认为是注释。另一种为 C 风格的注释。即，以“/*”开始，以“*/”结束的所有内容都认为是注释。C 风格的注释可跨多行，如：

```
# this is a single line comment
/* this is also a single line comment */
/* this, however,
   is a multiple line
   comment
*/
```

自 MySQL 3.23 版以来，可在 C 风格的注释中“隐藏”MySQL 特有的关键字，注释以“/*!”而不是以“/*”起头。MySQL 查看这种特殊类型注释的内部并使用这些关键字，但其他数据库服务器将这些关键字作为注释的一部分忽略。这样有助于编写由 MySQL 执行时利用 MySQL 特有功能的代码，而且该代码也可以不用修改就用于其他数据库服务器。下面的两条语句对于非 MySQL 的数据库服务器是等价的，但如果是 MySQL 服务器，将在第二条语句中执行一个 INSERT DELAYED 操作：

```
INSERT INTO absence (student_id,date) VALUES(13,"1999-09-28")
```

```
INSERT /*! DELAYED */ INTO absence (student_id,date) VALUES(13,"1999-09-28")
```

自 MySQL 3.23.3 以来,除了刚才介绍的注释风格外,还可以用两个短划线和—个空格 (“ -- ”) 来开始注释;从这两个短划线到行的结束的所有内容都作为注释处理。有的数据库以双短划线作为注释的起始。MySQL 也允许这样,但需要加—个空格以免产生混淆。

例如,带有如像 5--7 这样的表达式的语句有可能被认为包含—个注释,但不可能写 5-- 7 这样的表达式,因此,这是—个很有用的探索。然而,这仅仅是一个探索,最好不用这种风格的注释。

3.8 解决方案随笔

本节内容相当杂;介绍了怎样编写解决各种问题的查询。多数内容是在邮件清单上看到的解决问题的方案(谢谢清单上的那些朋友,他们为解决方案作了很多工作)。

3.8.1 将子选择编写为连接

MySQL 自 3.24 版本以来才具有子选择功能。这项功能的缺少是 MySQL 中—件常常令人惋惜的事,但有一件事很多人似乎没有认识到,那就是用子选择编写的查询通常可以用连接来编写。事实上,即使 MySQL 具有了子查询,检查用子选择编写的查询也是—件苦差事;用连接而不是用子选择来编写会更为有效。

1. 重新编写选择匹配值的子选择

下面是一个包含—个子选择查询的样例,它从 score 表中选择所有测试的学分(即,忽略测验的学分):

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM event WHERE type = "T")
```

可通过将其转换为—个简单的连接,不用子选择也可以编写出相同的查询,如下所示:

```
SELECT score.* FROM score, event
WHERE score.event_id = event.event_id AND event.type = "T"
```

下面的例子为选择女学生的学分:

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = "F")
```

可将其转换为连接,如下所示:

```
SELECT score.* FROM score, student
WHERE score.student_id = student.student_id AND student.sex = "F"
```

这里是一个模式,子选择查询如下形式:

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2 FROM table2a WHERE column2b = value)
```

这样的查询可转换为如下形式的连接:

```
SELECT table1.* FROM table1, table2
WHERE table1.column1 = table2.column2a AND table2.column2b = value
```

2. 重新编写选择非匹配值的子选择查询

另—种常用的子选择查询是查找—个表中有的而另—个表中没有的值。正如以前所看到的那样,“那些未给出的值”这—类的问题是 LEFT JOIN 可能有用的—个线索。下面的查询包含—个子选择(它寻找那些全勤的学生):

```
SELECT * FROM student
```



```
WHERE student_id NOT IN (SELECT student_id FROM absence)
```

此查询可利用 LEFT JOIN 重新编写如下：

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.student_id
WHERE absence.student_id IS NULL
```

一般来说，子选择查询的形式如下：

```
SELECT * FROM table1
WHERE column1 NOT EXISTS (SELECT column2 FROM table2)
```

具有这样形式的查询可重新编写如下：

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL
```

这假定 table2.column2 定义为 NOT NULL。

3.8.2 检查表中未给出的值

我们已经在3.6节“检索记录”中看到，在要想知道一个表中哪些值不出现在另一表中时，可对两个表使用 LEFT JOIN 并查找那些从第二个表中选中 NULL 的行。并用下列两个表举例：

表 t1		表 t2	
i1	c1	i2	c2
1	a	2	c
2	b	3	b
3	c	4	a

查找所有不出现在 t2.i2 列中的所有 t1.i1 值的 LEFT JOIN 如下：

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2 WHERE t2.i2 IS NULL
```

i1	c1
1	a

现在让我们来考虑一种更为困难的情况，“缺了哪些值”。对于第 1 章中提到的学分保存方案中，有一个列出学生的 student 表，一个列出已经出现过的学分事件的 event 表，以及列出每个学生的每次学分事件学分的 score 表。但是，如果一个学生在某个测试或测验的同一天病了，那么 score 表中将不会有这个学生的该事件的学分，因此，要进行测验或测试的补考。我们怎样查找这些缺少了的记录，以便能保证让这些学生进行补考？

问题是要对所有的学分事件确定哪些学生没有某个学分事件的学分。换个说法，就是我们希望知道学生和事件的哪些组合不出现在学分表中。这就是我们希望 LEFT JOIN 所做的事。这个连接不像前例中那样简单，因为我们不仅仅要查找不出现在单列中的值；还需要查找两列的组合。

我们想要的这种组合是所有学生/事件的组合，它们由 student 表与 event 表的叉积产生：

```
FROM student, event
```

然后我们取出此连接的结果，与 score 表执行一个 LEFT JOIN 语句找出匹配者：

```
FROM student, event
```

```
LEFT JOIN score ON student.student_id = score.student_id
AND event.event_id = score.event_id
```

请注意，ON 子句使得 score 表中的行根据不同表中的匹配者进行连接。这是解决本问题的关键。LEFT JOIN 强制为由 student 和 event 表的叉连接生成的每行产生一个行，即使没有相应的 score 表记录也是这样。这些缺少的学分记录的结果行可通过一个事实来识别，就是来自 score 表的列将全是 NULL 的。我们可在 WHERE 子句中选出这些记录。来自 score 表的任何列都是这样，但因为我们查找的是缺少的学分，测试 score 列从概念上可能最为清晰：

```
WHERE score.score IS NULL
```

可利用 ORDER BY 子句对结果进行排序。两种最合理的排序分别是按学生和按事件进行，我们选择第一种：

```
ORDER BY student.student_id, event.event_id
```

现在需要做的就是命名我们希望在输出结果中看到的列。最终的查询如下：

```
SELECT
    student.name, student.student_id,
    event.date, event.event_id, event.type
FROM
    student, event
LEFT JOIN score ON student.student_id = score.student_id
AND event.event_id = score.event_id
WHERE
    score.score IS NULL
ORDER BY
    student.student_id, event.event_id
```

运行此查询得出如下结果：

name	student_id	date	event_id	type
Megan	1	1999-09-16	4	Q
Joseph	2	1999-09-03	1	Q
Katie	4	1999-09-23	5	Q
Devri	13	1999-09-03	1	Q
Devri	13	1999-10-01	6	T
Will	17	1999-09-16	4	Q
Avery	20	1999-09-06	2	Q
Gregory	23	1999-10-01	6	T
Sarah	24	1999-09-23	5	Q
Carter	27	1999-09-16	4	Q
Carter	27	1999-09-23	5	Q
Gabrielle	29	1999-09-16	4	Q
Grace	30	1999-09-23	5	Q

这里有一个问题要引起注意。此输出列出了学生的 ID 和事件的 ID。student_id 列出现在 student 和 score 表中，因此，开始您可能会认为选择列表可以给出 student.student_id 或 score.student_id。但实际不是这样，因为能够找到感兴趣记录的基础是所有学分表字段返回 NULL。选择 score.student_id 将只在输出中产生 NULL 值的列。类似的推理可应用到 event_id 列，它也出现在 event 和 score 表中。

3.8.3 执行 UNION 操作

如果想通过从具有相同结构的多个表中建立一个结果集，可在某些数据库系统中使用某

种 UNION 语句来实现。MySQL 没有 UNION（至少直到 3.24 版还没有），但有许多办法来解决这个问题，下面是两种可行的方案：

- 执行多个 SELECT 查询，每个表执行一个。如果不关心所选出行的次序，这样做就行了。
- 将每个表中的行选入一个临时存储表，然后选择该表的内容。这样可对行按所需的次序进行排序。在 MySQL 3.23 版及以后的版本中，可通过允许服务器创建存储表来解决这个问题。而且，还可以使该表为临时表，以便在您与服务器的会话结束时，自动删除该表。

在下面的代码中，我们明确地删除该表使服务器释放与其有关的资源。如果客户机会话将继续执行进一步的查询，这样做很有好处。为了取到更好的性能，还可以利用 HEAP（在内存中）表。

```
CREATE TEMPORARY TABLE hold_tbl TYPE=HEAP SELECT ... FROM table1 WHERE ...
INSERT INTO hold_tbl SELECT ... FROM table2 WHERE ...
INSERT INTO hold_tbl SELECT ... FROM table3 WHERE ...
...
SELECT * FROM hold_tbl ORDER BY ...
DROP TABLE hold_tbl
```

对于 3.23 版本，除了必须自己明确定义 hold_tbl 表中的列外，其想法是类似的，而且结尾处的 DROP TABLE 是强制性的，用来防止在以下客户机会话生命周期之后继续存在：

```
CREATE TABLE hold_tbl (column1 ..., column2 ..., ...)
TYPE=HEAP SELECT ... FROM table1 WHERE ...
INSERT INTO hold_tbl SELECT ... FROM table1 WHERE ...
INSERT INTO hold_tbl SELECT ... FROM table2 WHERE ...
INSERT INTO hold_tbl SELECT ... FROM table3 WHERE ...
SELECT * FROM hold_tbl ORDER BY ...
DROP TABLE hold_tbl
```

3.8.4 增加序列号列

如果用 ALTER TABLE 增加 AUTO_INCREMENT 列，则该列用序列号自动地填充。下面这组 mysql 会话中的语句示出了怎样创建一个表，在其中存放数据，然后增加一个 AUTO_INCREMENT 列：

```
mysql> CREATE TABLE t (c CHAR(10));
mysql> INSERT INTO t VALUES("a"),("b"),("c");
mysql> SELECT * FROM t;
+-----+
| c      |
+-----+
| a      |
| b      |
| c      |
+-----+
mysql> ALTER TABLE t ADD i INT AUTO_INCREMENT NOT NULL PRIMARY KEY;
mysql> SELECT * FROM t;
+-----+-----+
| c      | i      |
+-----+-----+
| a      | 1      |
| b      | 2      |
| c      | 3      |
+-----+-----+
```

3.8.5 对某个已有的列进行排序

如果有一个数值列，可对其按如下进行排序（或对其重排序，如果已对其排过序，但删除了行并且想要对值重新排序使其连续）：

```
ALTER TABLE t MODIFY i INT NULL
UPDATE t SET i = NULL
ALTER TABLE t MODIFY i INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY
```

但是有一种更容易的方法，那就是删除该列，然后再作为一个 AUTO_INCREMENT 列追加它。ALTER TABLE 允许指定多个活动，因此，上述工作可在单个语句中完成：

```
ALTER TABLE t
  DROP i,
  ADD i INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY
```

3.8.6 非正常次序的串

假如有一个表示体育机构人员的表，如橄榄球队，如果按人员职位进行排序，以便以特殊的顺序表示它，如：教练、教练助理、四分卫、流动后卫、接球员、巡逻员等。可将列定义为 ENUM 并按希望出现的顺序定义枚举元素。对该列的排序将会以所指定的顺序自动进行。

3.8.7 建立计数表

在第2章的“使用序列”小节中，我们介绍了怎样利用 LAST_INSERT_ID(expr) 生成一个序列。那个例子说明了怎样利用单列的表进行计数。那样做对于只需要单个计数器的情形能够满足需要，但是，如果需要几个计数器，该方法将会引起不必要的表重复。假如有一个 Web 站点并且想要在几个页面上放置“此页面已经被访问 nnn 次”这样的计数器。那么为每个具有一个计数器的页面建立一个单独的表就有些多余了。

避免创建多个计数器表的一种方法是建立一个两列的表。其中一列存放计数值；另一列存放计数器名。这时仍然可以使用 LAST_INSERT_ID() 函数，但可用计数器名来决定用哪一行。这个表如下所示：

```
CREATE TABLE counter
(
  count INT UNSIGNED,
  name varchar(255) NOT NULL PRIMARY KEY
)
```

其中计数器名为一个串，从而可以调用任何想要的计数器，我们将其定义为 PRIMARY KEY 以免名称重复。这里假定使用这个表的应用程序知道他们将使用的名称。对于前面所说的 Web 计数器，可通过利用文件树中每个页面的路径名作为其计数器名的方法，保证计数器名的唯一性。例如，要为站点的主页建立一个新计数器，可执行下列语句：

```
INSERT INTO counter (name) VALUES("index.html")
```

它用零值初始化称为“index.html”的计数器。为了生成序列中的下一个值，增加表中相应的计数值，然后用 LAST_INSERT_ID() 检索它：

```
UPDATE counter
  SET count = LAST_INSERT_ID(count+1)
  WHERE name = "index.html"
SELECT LAST_INSERT_ID()
```

另一种方法是不用 `LAST_INSERT_ID()` 增加计数器的值，如下所示：

```
UPDATE counter SET count = count+1 WHERE name = "index.html"  
SELECT count FROM counter WHERE name = "index.html"
```

然而，如果另一个客户在您发布 `UPDATE` 语句与 `SELECT` 语句之间增加了该计数器的值，则这种方法工作不正常。不过可在此两条语句的前后分别放置 `LOCK TABLES` 和 `UNLOCK TABLES`，在您使用该计数器时阻塞其他客户，以解决上述问题。但用 `LAST_INSERT_ID()` 方法完成同样的工作更为容易一些。因为它的值是客户专用的，您总能得到自己插入的值，而不是其他客户插入的值，而且不必阻塞其他客户使代码复杂化。

3.8.8 检查表是否存在

在应用程序内部知道一个表是否存在有时很有用。为了做到这一点，可使用下列任一条语句：

```
SELECT COUNT(*) FROM tbl_name  
SELECT * FROM tbl_name WHERE 1=0
```

如果指定的表存在，则上述两条语句都将执行成功，如果不存在，则都失败。它们是这种测试的很好的查询。它们执行速度快，所以不会费太多的时间。这种方法最适合您自己编写的应用程序，因为您可以测试查询的成功与失败并采取相应的措施。但在从 `mysql` 运行的批量脚本中不特别有用，因为发生错误时除了终止运行外不可能做任何事（或者可以忽略相应的错误，但是显然无法再运行该查询了）。

3.9 MySQL 不支持的功能

本节介绍其他数据库中有而 `MySQL` 中无的功能。它介绍省略了什么功能，以及在需要这些功能时怎么办。一般情况下，`MySQL` 之所以忽略某些功能是因为它们有负面性能影响。有的功能正在开发者的计划清单上，一旦找到一种方法可以实现相应的功能而又不致于影响良好性能的目标，就会对它们进行实现。

- 子选择。子选择是嵌套在另一个 `SELECT` 语句内的 `SELECT` 语句，如下面的查询所示：

```
SELECT * FROM score  
WHERE event_id IN (SELECT event_id FROM event WHERE type = "T")
```

子选择打算在 `MySQL 3.24` 中给出，到那时它们就不会忽略了。但到那时，许多用子选择撰写的查询也可以用连接来编写。请参阅 3.8.1 节“将子选择编写为连接”。

- 事务处理和提交/回退。事务处理是由其他客户机作为一个整体不中断执行的一组 `SQL` 语句。提交/回退功能允许规定数条语句作为一个整体执行或不执行。即，如果事务处理中的任何一条语句失败，那么直到该语句前执行的所有语句的作用都被撤消。

`MySQL` 自动进行单一 `SQL` 语句的同步以免客户机互相干扰。（例如，两个客户机不能对相同的表进行同时写入。）此外，可利用 `LOCK TABLES` 和 `UNLOCK TABLES` 将数条语句组成一个整体，这使您能够完成单条语句的并发控制所不能满足的操作。`MySQL` 与事务处理有关的问题是，它不能自动对数条语句进行组织，而且如果这些语句中有某一条失败后也不能对它们进行回退。

为了弄清事务处理为什么有用，可举例说明。假如您在服装销售业工作，无论何时，只要您的销售人员进行了一次销售，都要更新库存数目。下面的例子说明了在多

个销售人员同时更新数据库时可能出现的问题（假如初始的衬衫库存数目为 47）：

t1 销售人员1卖出3件衬衫

t2 销售人员检索当前衬衫计数（47）：

```
SELECT quantity FROM inventory WHERE item = "shirt"
```

t3 销售人员2卖出2件衬衫

t4 销售人员2检索当前衬衫计数（47）

```
SELECT quantity FROM inventory WHERE item = "shirt"
```

t5 销售人员1计算库存的新数目为 $47 - 3 = 44$ 并设置衬衫计数为 44：

```
UPDATE inventory SET quantity = 44 WHERE item = "shirt"
```

t6 销售人员2计算库存的新数目为 $47 - 2 = 45$ 并设置衬衫计数为 45：

```
UPDATE inventory SET quantity = 45 WHERE item = "shirt"
```

在这个事件序列结束时，您已经卖掉了 5 件衬衫，但库存数目却是 45 而不是 42。问题是如果在一条语句中查看库存而在另一条语句中更新其值，这是一个多语句的事务处理。第二条语句中所进行的活动取决于第一条语句中检索出的值。但是如果在重叠的时间范围内出现独立的事务处理，则每个事务处理的语句会纠缠在一起，并且互相干扰。在事务处理型的数据库中，每个销售人员的语句可作为一个事务处理执行，这样，销售人员2的语句在销售人员1的语句完成之前不会被执行。在MySQL中，可用两种方法达到这个目的：

■ 方法1：作为一个整体执行一组语句。可利用 `LOCK TABLES` 和 `UNLOCK TABLES`将语句组织在一起，并将它们作为一个原子单元执行：锁住所需使用的表，发布查询，然后释放这些锁。这样阻止了其他人在您锁住这些表时使用它们。利用表同步，库存情况如下所示：

t1 销售人员1卖出3件衬衫

t2 销售人员1请求一个锁并检索当前衬衫计数（47）

```
LOCK TABLES inventory WRITE
```

```
SELECT quantity FROM inventory WHERE item = "shirt"
```

t3 销售人员2卖出2件衬衫

t4 销售人员2试图取得一个锁：这被阻塞，因为销售人员 1 已经占住了锁：

```
LOCK TABLES inventory WRITE
```

t5 销售人员1计算库存的新数目为 $47 - 3 = 44$ 并设置衬衫计数为 44，然后释放锁：

```
UPDATE inventory SET quantity = 44 WHERE item = "shirt"
```

```
UNLOCK TABLES
```

t6 现在销售人员2的锁请求成功。销售人员2检索当前衬衫计数（44）

```
SELECT quantity FROM inventory WHERE item = "shirt"
```

t7 销售人员2计算库存的新数目为 $44 - 2 = 42$ ，设置衬衫计数为 42，然后释放锁：

```
UPDATE inventory SET quantity = 42 WHERE item = "shirt"
```

```
UNLOCK TABLES
```

现在来自两个事务处理的语句不混淆了，并且库存衬衫数也正确进行了设置。我们在这里使用了一个 `WRITE` 锁，因为我们需要修改 `inventory` 表。如果只是读取表，可使用 `READ` 锁。当您正在使用表时，这个锁允许其他客户机读取表。

在刚才举的例子中，销售人员2大概不会注意到执行速度上的差异，因为其中的事务处理都很短，执行速度很快。但是，作为一个具有普遍意义的规则，那就是应该尽

量避免长时间地锁住表。

如果您正在使用多个表，那么在您执行成组查询之前，必须锁住他们。如果只是从某个特定的表中读取数据，那么只需给该表加一个读出锁而不是写入锁。假如有一组查询，其中想对 inventory 表作某些更改，而同时需要从 customer 表中读取某些数据。在此情形下，inventory 表上需要一个写入锁，而 customer 表上需要一个读出锁：

```
LOCK TABLES inventory WRITE, customer READ
```

```
...
```

```
UNLOCK TABLES
```

这里要求您自己对表进行加锁和解锁。支持事务处理的数据库系统将会自动完成这些工作。但是，在作为一个整体执行的分组语句方面，无论在是否支持事务处理的数据库中都是相同的。

- 方法2：使用相对更新而不是绝对更新。要解决来自多个事务处理的语句混淆问题，应消除语句之间的依赖性。虽然这样做并不都总是可能的，它只针对我们的库存例子可行。对于方法1中所用的库存更新方法，其中事务处理需要查看当前库存数目，并依据销售衬衫的数目计算新值，然后更新衬衫的数目。有可能通过相对于当前衬衫数目进行计数更新，在一个步骤中完成工作。如下所示：

t1 销售人员1卖出3件衬衫

t2 销售人员1将衬衫计数减3：

```
UPDATE inventory SET quantity = quantity - 3 WHERE item =  
"shirt"
```

t3 销售人员2卖出2件衬衫

t4 销售人员2将衬衫计数减2：

```
UPDATE inventory SET quantity = quantity - 2 WHERE item =  
"shirt"
```

因此，这里根本不需要多条语句的事务处理，从而也不需要锁住表以模拟事务处理功能。如果所使用的事务处理类型与这里类似，那么就可以不用事务处理也能完成工作。

上面的例子说明了在特殊情形下怎样避免对事务处理功能的需求。但这并不是说不存在那种确实需要事务处理功能的场合。典型的例子是财务转账，其中钱从一个账户转到另一个账户。假如 Bill 给 Bob 开了一张 \$100 的支票，Bob 兑现了这张支票。Bill 的户头上应该减掉 \$100 而 Bob 的户头上应该增加相同数量的钱：

```
UPDATE account SET balance = balance - 100 WHERE name = "Bill"
```

```
UPDATE account SET balance = balance + 100 WHERE name = "Bob"
```

如果在这两条语句执行中，系统发生了崩溃，此事务处理就不完整了。具有真正事务处理和提交/回退功能的数据库系统能够处理这种情况（至少从理论上能够处理。您可能仍然必须判断遇到了哪些事务处理并重新发布它们，但至少不会担心事务只处理了一半）。在 MySQL 中，系统崩溃时可通过检查更新日志来判断事务处理的状态，虽然这可能需要对日志进行某种手工检查。

- 外部键和引用完整性。外部键允许定义一个表中的键与另一个表中的键相关，而引用完整性允许放置对包含外部键的表可以做什么的约束。例如，samp_db 样例数据库的 score 表中包含一个 student_id 列，我们用它来将学分记录关联到 student 表中的学生。score.student_id 将定义为支持此概念的数据库中的一个外部键，我们将在其上加上一

条约束，使不能为 student 表中不存在的学生输入学分记录。此外，应该允许级联删除，以便如果某个学生从 student 表中被删除后，该学生的任何学分记录将会自动地从 score 表中删除。

外部键有助于保持数据的一致性，而且还提供了某种方便的手段。MySQL 不支持外部键的原因主要是由于它对数据库的实现与维护有负作用。（MySQL 参考指南详细列出了这些原因。）注意，对于外部键的这种看法与其他数据库文献中的看法有些不同，有的数据库文献通常将它们描述成“基本的”。MySQL 的开发者并不赞同这个观点。如果您赞成，那么最好是考虑采用其他提供外部键支持的数据库。如果数据具有特别复杂的关系，您可能不希望担负在应用程序中实现这些相关性的工作。（即使这样的工作量要比增加几个额外的 DELETE 语句的工作量要稍少一些。）

除了在一定程度上能够在 CREATE TABLE 语句中分析 FOREIGN KEY 子句外，MySQL 不支持外部键。（这有助于使从其他数据库移植代码到 MySQL 更为容易。）MySQL 不强制让外部键作为一种约束，也不提供级联删除功能。

外部键强制实施的约束一般不难用程序逻辑来实现。有时，它只是一个怎样进行数据录入处理的问题。例如，为了将一个新记录插入 score 表，不太可能插入不存在的学生学分。显然，输入一组学分的方法应该是根据从 student 表得出的学生名单，对每个学生，取其学分并利用该学生的 ID 号产生一个 score 表的记录。对于这个过程，不存在录入一个不存在的学生的记录的可能。您不会凭空造出一个学分记录来插入 score 表。

要实现 DELETE 的级联效果，必须用自己的应用程序逻辑来完成。假如想要删除 13 号学生。这也隐含表示需要删除该学生的学分记录。在支持级联删除的数据库中，只需要用如下的语句就可以删除 student 表的记录和相应的 score 表的记录：

```
DELETE FROM student WHERE student_id = 13
```

而在 MySQL 中，必须明确地用 DELETE 语句自己进行第二个删除语句：

```
DELETE FROM student WHERE student_id = 13
```

```
DELETE FROM score WHERE student_id = 13
```

- 存储过程和触发器。存储过程是编译和存放在服务器中的 SQL 代码。它可在以后调用而无需从客户机发送并分析。可以对一个过程进行更改以影响使用它的任何客户机应用程序。触发器功能使一个过程在某个事件发生时被激活，如从表中删除某个记录时，激活相应的过程。例如，某个作为累计成分的记录被删除时，应该重新进行累计，使累计数反映最新情况。存储过程语言已列入了 MySQL 准备实现的计划。
- 视图。视图是一个逻辑概念，其功能像表但本身不是表。它提供了一种查看不同表中的列的途径，在查看时好像这些列属于同一个表一样。视图有时也称为虚表。MySQL 也准备实现视图功能。
- 记录级权限和锁定。MySQL 支持各种权限，从全局权限到数据库、表、列的权限。但它不支持记录级的权限。不过，可在应用程序中利用 GET_LOCK() 和 RELEASE_LOCK() 函数来实现协同记录锁。这个过程在附录 C“运行算符和函数参考”中相应的项目下介绍。
- “--”作为注释的开始。MySQL 不支持这种注释风格，因为它是一个有歧义的结构，虽然自 MySQL 3.23 以来，注释可用两个短划线加一个空格开始。更详细的信息，请参阅 3.7 节“加注释”。

第4章 查询优化

关系数据库的世界是一个表与集合、表与集合上的运算占统治地位的世界。数据库是一个表的集合，而表又是行和列的集合。在发布一条 `SELECT` 查询从表中进行检索行时，得到另一个行和列的集合。这些都是一些抽象的概念，对于数据库系统用来操纵表中数据的基本表示没有多少参考价值。另一个抽象概念是，表上的运算都同时进行；查询是一种概念性的集合运算，并且集合论中没有时间概念。

当然，现实世界是相当不同的。数据库管理系统实现了抽象的概念，但是在实际的硬件范围内要受到实际的物理约束。结果是，查询要花时间，有时要花很长的时间。而人类很容易不耐烦，不喜欢等待，因此我们丢下了集合上的那些瞬间的数学运算的抽象世界去寻求加速查询的方法。幸运的是，有几种加速运算的技术，可对表进行索引使数据库服务器查找行更快。可考虑怎样充分利用这些索引来编写查询。可编写影响服务器调度机制的查询，使来自多个客户机的查询协作得更好。我们思考基本硬件怎样运行，以便想出怎样克服其物理约束对性能进行改善的方法。

这些正是本章所要讨论的问题，其目标是优化数据库系统的性能，使其尽可能快地处理各种查询。`MySQL` 已经相当快了，但即使是最快的数据库，在人的设计下还能运行得更快。

4.1 使用索引

我们首先讨论索引，因为它是加快查询的最重要的工具。还有其他加快查询的技术，但是最有效的莫过于恰当地使用索引了。在 `MySQL` 的邮件清单上，人们通常询问关于使查询更快的问题。在大量的案例中，都是因为表上没有索引，一般只要加上索引就可以立即解决问题。但这样也并非总是有效，因为优化并非总是那样简单。然而，如果不使用索引，在许多情形下，用其他手段改善性能只会是浪费时间。应该首先考虑使用索引取得最大的性能改善，然后再寻求其他可能有帮助的技术。

本节介绍索引是什么、它怎样改善查询性能、索引在什么情况下可能会降低性能，以及如何为表选择索引。下一节，我们将讨论 `MySQL` 的查询优化程序。除了知道怎样创建索引外，了解一些优化程序的知识也是有好处的，因为这样可以更好地利用所创建的索引。某些编写查询的方法实际上会妨碍索引的效果，应该避免这种情况出现。（虽然并非总会这样。有时也会希望忽略优化程序的作用。我们也将介绍这些情况。）

4.1.1 索引的益处

让我们从一个无索引的表着手来考察索引是怎样起作用的。无索引的表就是一个无序的行集。例如，图4-1给出了我们在第1章“`MySQL` 与 `SQL` 介绍”中首先看到的 `ad` 表。这个表上没有索引，因此如果我们查找某个特定公司的行时，必须查看表中的每一行，看它是否与所需的值匹配。这是一个全表扫描，很慢，如果表中只有少数几个记录与搜索条件相匹配，则其效率是相当低的。

图4-2给出了相同的表，但在表的 `company_num` 列上增加了一个索引。此索引包含表中每行的一项，但此索引是在 `company_num` 上排序的。现在，不需要逐行搜索全表查找匹配的条款，而是可以利用索引进行查找。假如我们要查找公司 13 的所有行，那么可以扫描索引，结果得出3行。然后到达公司 14 的行，这是一个比我们正在查找的要大的号码。索引值是排序的，因此在读到包含 14 的记录时，我们知道不会再有匹配的记录，可以退出了。如果查找一个值，它在索引表中某个中间点以前不会出现，那么也有找到其第一个匹配索引项的定位算法，而不用进行表的顺序扫描（如二分查找法）。这样，可以快速定位到第一个匹配的值，以节省大量搜索时间。数据库利用了各种各样的快速定位索引值的技术，这些技术是什么并不重要，重要的是它们工作正常，索引技术是个好东西。

有人会问，为什么不只对数据文件进行排序，省掉索引文件？这样不也在搜索时产生相同的效果吗？问得好，如果只有单个索引时，是这样的。不过有可能会用到第二个索引，但同时以两种不同的方法对同一个数据文件进行排序是不可能的。（如，想要一个顾客名的索引，同时又要一个顾客 ID 号或电话号码的索引。）将索引文件作为一个与数据文件独立的实体就解决了这个问题，而且允许创建多个索引。此外，索引中的行一般要比数据文件中的行短。在插入或删除值时，为保持排序顺序而移动较短的索引值与移动较长的数据行相比更为容易。

ad 表

company_num	ad_num	hit_fee
14	48	0.01
23	49	0.02
17	52	0.01
13	55	0.03
23	62	0.02
23	63	0.01
23	64	0.02
13	77	0.03
23	99	0.03
14	101	0.01
13	102	0.01
17	119	0.02

图4-1 无索引的 ad 表

index	company_num	ad_num	hit_fee
13	14	48	0.01
13	23	49	0.02
13	17	52	0.01
14	13	55	0.03
14	23	62	0.02
17	23	63	0.01
17	23	64	0.02
23	13	77	0.03
23	23	99	0.03
23	14	101	0.01
23	13	102	0.01
23	17	119	0.02

图4-2 有索引的 ad 表

这个例子与 MySQL 索引表的方法相符。表的数据行保存在数据文件中，而索引值保存在索引文件中。一个表上可不止一个索引；如果确实不止一个索引，它们都保存在同一个索引文件中。索引文件中的每个索引由排过序的用来快速访问数据文件的键记录数组构成。

前面的讨论描述了单表查询中索引的好处，其中使用索引消除了全表扫描，极大地加快了搜索的速度。在执行涉及多个表的连接查询时，索引甚至会更有价值。在单个表的查询中，每列需要查看的值的数目就是表中行的数目。而在多个表的查询中，可能的组合数目极大，因为这个数目为各表中行数之积。

假如有三个未索引的表 `t1`、`t2`、`t3`，分别只包含列 `c1`、`c2`、`c3`，每个表分别由含有数值 1

到 1000 的 1000 行组成。查找对应值相等的表行组合的查询如下所示：

```
SELECT c1, c2, c3
FROM t1, t2, t3
WHERE c1 = c2 AND c1 = c3
```

此查询的结果应该为 1000 行，每个组合包含 3 个相等的值。如果我们在无索引的情况下处理此查询，则不可能知道哪些行包含那些值。因此，必须寻找出所有组合以便得出与 WHERE 子句相配的那些组合。可能的组合数目为 $1000 \times 1000 \times 1000$ （十亿），比匹配数目多一百万倍。很多工作都浪费了，并且这个查询将会非常慢，即使在如像 MySQL 这样快的数据库中执行也会很慢。而这还是每个表中只有 1000 行的情形。如果每个表中有一百万行时，将会怎样？很显然，这样将会产生性能极为低下的结果。如果对每个表进行索引，就能极大地加速查询进程，因为利用索引的查询处理如下：

1) 如下从表 t1 中选择第一行，查看此行所包含的值。

2) 使用表 t2 上的索引，直接跳到 t2 中与来自 t1 的值匹配的行。类似，利用表 t3 上的索引，直接跳到 t3 中与来自 t1 的值匹配的行。

3) 进到表 t1 的下一行并重复前面的过程直到 t1 中所有的行已经查过。

在此情形下，我们仍然对表 t1 执行了一个完全扫描，但能够在表 t2 和 t3 上进行索引查找直接取出这些表中的行。从道理上说，这时的查询比未用索引时要快一百万倍。

如上所述，MySQL 利用索引加速了 WHERE 子句中条件相配的行的搜索，或者说在执行连接时加快了与其他表中的行匹配的行的搜索。它也利用索引来改进其他操作的性能：

- 在使用 MIN() 和 MAX() 函数时，能够快速找到索引列的最小或最大值。
- MySQL 常常能够利用索引来完成 ORDER BY 子句的排序操作。
- 有时，MySQL 可避免对整个数据文件的读取。假如从一个索引数值列中选择值，而且选择不选择表中其他列。这时，通过对索引值的读取，就已经得到了读取数据文件所要得到的值。没有对相同的值进行两次读取的必要，因此，甚至无需涉及数据文件。

4.1.2 索引的弊端

一般情况下，如果 MySQL 能够知道怎样用索引来更快地处理查询，它就会这样做。这表示，在大多数情况下，如果您不对表进行索引，则损害的是您自己的利益。可以看出，作者描绘了索引的诸多好处。但有不便利之处吗？是的，有。实际上，这些缺点被优点所掩盖了，但应该对它们有所了解。

首先，索引文件要占磁盘空间。如果有大量的索引，索引文件可能会比数据文件更快地达到最大的文件尺寸。其次，索引文件加快了检索，但增加了插入和删除，以及更新索引列中的值的时间（即，降低了大多数涉及写入的操作的时间），因为写操作不仅涉及数据行，而且还常常涉及索引。一个表拥有的索引越多，则写操作的平均性能下降就越大。在 4.4 节“有效地装载数据”中，我们将更为详细地介绍这些性能问题，并讨论怎样解决。

4.1.3 选择索引

创建索引的语法已经在 3.4.3 节“创建和删除索引”中进行了介绍。这里，我们假定您已经阅读过该节。但是知道语法并不能帮助确定表怎样进行索引。要决定表怎样进行索引需要考虑表的使用方式。本节介绍一些关于怎样确定和挑选索引列的准则：

- **搜索的索引列，不一定是所要选择的列。**换句话说，最适合索引的列是出现在 WHERE 子句中的列，或连接子句中指定的列，而不是出现在 SELECT 关键字后的选择列表中的列：

```
SELECT
    col_a                ← 不适合作索引列
FROM
    Tbl1 LEFT JOIN tbl2
        ON tbl1.col_b = tbl2.col_c    ← 适合作索引列
WHERE
    col_d = expr          ← 适合作索引列
```

当然，所选择的列和用于 WHERE 子句的列也可能是相同的。关键是，列出现在选择列表中不是该列应该索引的标志。

出现在连接子句中的列或出现在形如 col1 = col2 的表达式中的列是很适合索引的列。查询中的 col_b 和 col_c 就是这样的例子。如果 MySQL 能利用连接列来优化一个查询，表示它通过消除全表扫描相当可观地减少了表行的组合。

- **使用惟一索引。**考虑某列中值的分布。对于惟一值的列，索引的效果最好，而具有多个重复值的列，其索引效果最差。例如，存放年龄的列具有不同值，很容易区分各行。而用来记录性别的列，只含有“M”和“F”，则对此列进行索引没有多大用处（不管搜索哪个值，都会得出大约一半的行）。

- **使用短索引。**如果对串列进行索引，应该指定一个前缀长度，只要有可能就应该这样做。例如，如果有一个 CHAR(200) 列，如果在前 10 个或 20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。对前 10 个或 20 个字符进行索引能够节省大量索引空间，也可能使查询更快。较小的索引涉及的磁盘 I/O 较少，较短的值比较起来更快。更为重要的是，对于较短的键值，索引高速缓存中的块能容纳更多的键值，因此，MySQL 也可以在内存中容纳更多的值。这增加了找到行而不用读取索引中较多块的可能性。（当然，应该利用一些常识。如仅用列值的第一个字符进行索引是不可能有多大的好处的，因为这个索引中不会有许多不同的值。）

- **利用最左前缀。**在创建一个 n 列的索引时，实际是创建了 MySQL 可利用的 n 个索引。多列索引可起几个索引的作用，因为可利用索引中最左边的列集来匹配行。这样的列集称为最左前缀。（这与索引一个列的前缀不同，索引一个列的前缀是利用该的前 n 个字符作为索引值。）

假如一个表在分别名为 state、city 和 zip 的三个列上有一个索引。索引中的行是按 state/city/zip 的次序存放的，因此，索引中的行也会自动按 state/city 的顺序和 state 的顺序存放。这表示，即使在查询中只指定 state 值或只指定 state 和 city 的值，MySQL 也可以利用索引。因此，此索引可用来搜索下列的列组合：

```
state, city, zip
state, city
state
```

MySQL 不能使用不涉及左前缀的搜索。例如，如果按 city 或 zip 进行搜索，则不能使用该索引。如果要搜索某个州以及某个 zip 代码（索引中的列1和列3），则此索引不能用于相应值的组合。但是，可利用索引来寻找与该州相符的行，以减少搜索范围。

■ **不要过度索引。**不要以为索引“越多越好”，什么东西都用索引是错的。每个额外的索引都要占用额外的磁盘空间，并降低写操作的性能，这一点我们前面已经介绍过。在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，索引越多，所花的时间越长。如果有一个索引很少利用或从不使用，那么会不必要地减缓表的修改速度。此外，MySQL 在生成一个执行计划时，要考虑各个索引，这也要费时间。创建多余的索引给查询优化带来了更多的工作。索引太多，也可能会使 MySQL 选择不到所要使用的最好索引。只保持所需的索引有利于查询优化。

如果想给已索引的表增加索引，应该考虑所要增加的索引是否是现有多列索引的最左索引。如果是，则就不要费力去增加这个索引了，因为已经有了。

■ **考虑在列上进行的比较类型。**索引可用于“<”、“<= ”、“= ”、“>= ”、“> ”和 BETWEEN 运算。在模式具有一个直接量前缀时，索引也用于 LIKE 运算。如果只将某个列用于其他类型的运算时（如 STRCMP()），对其进行索引没有价值。

4.2 MySQL 查询优化程序

在发布一个选择行的查询时，MySQL 进行分析，看是否能够对它进行优化，使它执行更快。本节中，我们将研究查询优化程序怎样工作。更详细的信息，可参阅 MySQL 参考指南中的“Getting Maximum Performance from MySQL”，该章描述了 MySQL 采用的各种优化措施。该章中的信息会不断变化，因为 MySQL 的开发者不断对优化程序进行改进，因此，有必要经常拜访一下该章，看看是否有可供利用的新技巧。（<http://www.mysql.com/> 处的 MySQL 联机参考指南在不断地更新。）

MySQL 查询优化程序利用了索引。当然，它也利用了其他信息。例如，如果发布下列查询，MySQL 将非常快地执行它，不管相应的表有多大：

```
SELECT * FROM tbl_name WHERE 1 = 0
```

在此情形中，MySQL 考察 WHERE 子句，如果认识到不可能有满足该查询的行，就不会对该表进行搜索。可利用 EXPLAIN 语句知道这一点，EXPLAIN 语句要求 MySQL 显示某些有关它应该执行一条 SELECT 查询，而实际没有执行的信息。为了使用 EXPLAIN，只需要 SELECT 语句前放置 EXPLAIN 即可，如下所示：

```
EXPLAIN SELECT * FROM tbl_name WHERE 1 = 0
```

EXPLAIN 的输出结果为：

```
+-----+
| Comment |
+-----+
| Impossible WHERE |
+-----+
```

通常，EXPLAIN 返回的信息比这个多，包括将用来扫描表的索引、将要使用的连接类型以及需要在每个表中扫描的行数估计等等。

4.2.1 优化程序怎样工作

MySQL 查询优化程序有几个目标，但其主要目标是尽量利用索引，而且尽量使用最具有限制性的索引以排除尽可能多的行。这样做可能会适得其反，因为发布一条 SELECT 语句的目的是寻找行，而不是拒绝它们。优化程序这样工作的原因是从要考虑的行中排除行越快，

那么找到确实符合给出标准的行就越快。如果能够首先进行最具限制性的测试，则查询可以进行得更快。假如有一个测试两列的查询，每列上都有一个索引：

```
WHERE col1 = "some value" AND col2 = "some other value"
```

还假定，与 col1 上的测试相符的有 900 行，与 col2 上的测试相符的有 300 行，而两个测试都通过的有 30 行。如果首先测试 col1，必须检查 900 行以找到也与 col2 值相符的 30 行。那么测试中有 870 将失败。如果首先测试 col2，要找到也与 col1 值相符的 30 行，只需检查 300 行。测试中有失败 270 次，这样所涉及的计算较少，磁盘 I/O 也较少。

遵循下列准则，有助于优化程序利用索引：

- 比较具有相同类型的列。在比较中利用索引列时，应该使用那些类型相同的列。例如，CHAR(10) 被视为与 CHAR(10) 或 VARCHAR(10) 相同，但不同于 CHAR(12) 和 VARCHAR(12)。INT 与 BIGINT 不同。在 MySQL 3.23 版以前，要求使用相同类型的列，否则列上的索引将不起作用。自 3.23 版后，不严格要求这样做，但相同的列类型比不同类型提供更好的性能。如果所比较的两列类型不同，可使用 ALTER TABLE 语句修改其中之一使它们的类型相配。
- 比较中应尽量使索引列独立。如果在函数调用或算术表达式中使用一个列，则 MySQL 不能使用这样的索引，因为它必须对每行计算表达式的值。有时，这是不可避免的，但很多时候，可以重新编写只取索引列本身的查询。

下面的 WHERE 子句说明了怎样进行这项工作。第一行中，优化程序将简化表达式 4/2 为值 2，然后使用 my_col 上的索引快速地找到小于 2 的值。而在第二个表达式中，MySQL 必须检索出每行的 my_col 值，乘以 2，然后将结果与 4 比较。没索引可用，因为列中的每个值都要检索，以便能对左边的表达式求值：

```
WHERE my_col < 4 / 2  
WHERE my_col * 2 < 4
```

让我们考虑另一个例子。假如有一个索引列 date_col。如果发布如下的查询，相应的索引未被使用：

```
SELECT * FROM my_tbl WHERE YEAR(date_col) < 1990
```

其中表达式并不将索引列与 1990 比较，而是将从列值计算出的值用于比较，而且必须计算每行的这个值。结果是，date_col 上的索引不可能得到使用。怎样解决？使用一个文字日期即可，这时将会使用 date_col 上的索引：

```
WHERE date_col < "1990-01-01"
```

但是假如没有特定的日期值，那么可能会对找到具有出现在距今一定天数内的日期的记录感兴趣。有几种方法来编写这样的查询，但并非所有方法都很好。三种可能的方法如下：

```
WHERE TO_DAYS(date_col) - TO_DAYS(CURRENT_DATE) < cutoff  
WHERE TO_DAYS(date_col) < cutoff + TO_DAYS(CURRENT_DATE)  
WHERE date_col < DATE_ADD(CURRENT_DATE, INTERVAL cutoff DAY)
```

其中第一行不能利用索引，因为必须为每行检索列，以便能够计算 TO_DAYS(date_col) 的值。第二行要好一些。cutoff 和 TO_DAYS(CURRENT_DATE) 两者都是常量，因此比较表达式的右边可在查询处理前由优化程序一次计算出来，而不是每行计算一次。但 date_col 列仍然出现在一个函数调用中，因此，没有使用索引。第三行是最好的方法。比较表达式的右边可在执行查询前作为常量一次计算出来，但现在其值是一个

日期。这个值可直接与 date_col 的值进行比较，不再需要转换为天数，可以利用索引。

- **在 LIKE 模式的起始处不要使用通配符。**有时，有的人会用下列形式的 WHERE 子句来搜索串：

```
WHERE col_name LIKE "%string%"
```

如果希望找到 string，不管它出现在列中任何位置，那么这样做是对的。但不要出于习惯在串的两边加“%”。如果实际要查找的只是出现在列的开始处的串，则不应该要第一个“%”号。例如，如果在一个包含姓的列中查找“Mac”起始的姓，应该编写如下的 WHERE 子句：

优化程序考虑模式中的开始的文字部分，然后利用索引找到相符合的行。不过宁

```
WHERE last_name LIKE "Mac%"
```

可写成如下的表达式，它允许使用 last_name 上的索引：

```
WHERE last_name >= "Mac" AND last_name < "Mad"
```

这种优化对使用 REGEXP 操作符的模式匹配不起作用。

- **帮助优化程序更好地评估索引的有效性。**缺省时，如果将索引列中的值与常量进行比较，优化程序将假定键值是均匀地分布在索引中的。优化程序还将对索引进行一个快速的检查，以估计在确定相应的索引是否应该用于常量的比较时要使用多少条目。可利用 myisamchk 或 isamchk 的 --analyze 选项给优化程序提供更好的信息，以便分析键值的分布。myisamchk 用于 MyISAM 表，isamchk 用于 ISAM 表。为了完成键值分析，必须能够登录到 MySQL 服务器主机中，而且必须对表文件具有写访问权限。
- **利用 EXPLAIN 检验优化程序操作。**检查用于查询中的索引是否能很快地排除行。如果不能，那么应该试一下利用 STRAIGHT_JOIN 强制按特定次序使用表来完成一个连接。查询的执行方式不那么显然；MySQL 可能会有很多理由不以您认为最好的次序使用索引。
- **测试查询的其他形式，而且不止一次地运行它们。**在测试一个查询的其他形式时，应该每种方法运行几次。如果对两个不同方法中的每种只运行查询一次，通常会发现第二个查询更快，因为来自第一个查询的信息在磁盘高速缓存中，不需要实际从磁盘上读出。还应该尽量在系统负载相对平稳的时候运行查询，以避免受系统中其他活动的影响。

4.2.2 忽略优化

这可能听起来有点奇怪，但在以下情况中，要废除 MySQL 的优化功能：

- **强迫 MySQL 慢慢地删除表的内容。**在需要完全删空一个表时，利用无 WHERE 子句的 DELETE 语句删除整个表的内容是最快的，如下所示：

```
DELETE FROM tbl_name
```

MySQL 对这种特殊情况的 DELETE 进行优化；它利用表信息文件中的表说明从头开始创建空数据文件和索引文件。这种优化使 DELETE 操作极快，因为 MySQL 无需单独地删除每一行。但在某些情况下，这样做会产生一些不必要的副作用：

- MySQL 报告所涉及的行为零，即使表不为空也是如此。很多时候这没有关系（虽然，如果事先没有思想准备，会感到困惑不解），但对于那些确实需要知道真实行数的应用程序来说，这是不恰当的。

- 如果表含有一个 AUTO_INCREMENT 列，则该列的顺序编号会以 1 从头开始。这是真实的事情，即使在 MySQL 3.23 中对 AUTO_INCREMENT 的处理进行了改进后也是这样。关于这个改进的介绍请参阅第2章中的“使用序列”小节。可增加 WHERE 1 > 0 子句对 DELETE 语句“不优化”。

```
DELETE FROM tbl_name WHERE 1 > 0
```

这迫使 MySQL 进行逐行的删除。相应的查询执行要慢得多，但将返回真正删除的行数。它还将保持当前的 AUTO_INCREMENT 序列的编号，不过只对 MyISAM 表（MySQL 3.23 以上的版本可用）有效。而对于 ISAM 表，序列仍将重置。

- **避免更新循环不终止。**如果更新一个索引列，如果该列用于 WHERE 子句且更新将索引值移入至今尚未出超的取值范围内时，有可能对所更新的行进行不终止的更新。假如表 my_tbl 有一个索引了的整数列 key_col。下列的查询会产生问题：

```
UPDATE my_tbl SET key_col = key_col+1 WHERE key_col > 0;
```

这个问题的解决方法是在 WHERE 子句中将 key_col 用于一个表达式，使 MySQL 不能使用索引：

```
UPDATE my_tbl SET key_col = key_col+1 WHERE key_col+0 > 0;
```

实际上，还有另外的方法，即升级到 MySQL 3.23.2 或更高的版本，它们已经解决了这样的问题。

- **以随机次序检索结果。**自 MySQL 3.23.3 以来，可使用 ORDER BY RAND() 随机地对结果进行排序。另一技术对 MySQL 更旧的版本很有用处，那就是选择一个随机数列，然后在该列上进行排序。但是，如果按如下编写查询，优化程序将会让您的愿望落空：

```
SELECT ..., RAND() as rand_col FROM ... ORDER BY rand_col
```

这里的问题是 MySQL 认为该列是一个函数调用，将认为相应的列值是一个常数，而对 ORDER BY 子句进行优化，使此查询失效。可在表达式中引用某个表列来蒙骗优化程序。例如，如果表中有一个名为 age 的列，可编写如下查询：

```
SELECT ..., age*0+RAND() as rand_col FROM ... ORDER BY rand_col
```

- **忽略优化程序的表连接次序。**可利用 STRIGHT_JOIN 强迫优化程序以特定的次序使用表。如果这样做，应该规定表的次序，使第一个表为从中选择的行数最少的表。（如果不能肯定哪个表满足这个要求，可将行数最多的表作为第一个表。）换句话说，应尽量规定表的次序，使最有限制性的选择先出现。排除可能的候选行越早，查询执行得就越快。要保证测试相应的查询两次；可能会有某些原因使优化程序不以您所想像的方式对表进行连接，并且 STRAIGHT_JOIN 也可能实际上不起作用。

4.3 列类型选择与查询效率

要选择有助于使查询执行更快的列，应遵循如下规则（这里，“BLOB 类型”应该理解为即包含 BLOB 也包含 TEXT 类型）：

- **使用定长列，不使用可变长列。**这条准则对被经常修改，从而容易产生碎片的表来说特别重要。例如，应该选择 CHAR 列而不选择 VARCHAR 列。所要权衡的是使用定长列时，表所占用的空间更多，但如果能够承担这种空间的耗费，使用定长行将比使用

可变长的行处理快得多。

- **在较短的列能够满足要求时不要使用较长的列。**如果正使用的是定长的 CHAR 列，应该使它们尽量短。如果列中所存储的最长值为 40 个字符，那么就不要再将其定义为 CHAR(255)；只要定义为 CHAR(40) 即可。如果能够使用 MEDIUMINT 而不是 BIGINT，表将会更小（磁盘 I/O 也较少），其值在计算中也可以处理得更快。
- **将列定义为 NOT NULL。**这样处理更快，所需空间更少。而且有时还能简化查询，因为不需要检查是否存在特例 NULL。
- **考虑使用 ENUM 列。**如果有一个只含有限数目的特定值的列，那么应该考虑将其转换为 ENUM 列。ENUM 列的值可以更快地处理，因为它们在内部是以数值表示的。
- **使用 PROCEDURE ANALYSE()。**如果使用的是 MySQL 3.23 或更新的版本，应该执行 PROCEDURE ANALYSE()，查看它所提供的关于表中列的信息：

```
SELECT * FROM tbl_name PROCEDURE ANALYSE()  
SELECT * FROM tbl_name PROCEDURE ANALYSE(16,256)
```

相应输出中有一列是关于表中每列的最佳列类型的建议。第二个例子要求 PROCEDURE ANALYSE() 不要建议含有多于 16 个值或取多于 256 字节的 ENUM 类型（可根据需要更改这些值）。如果没有这样的限制，输出可能会很长；ENUM 的定义也会很难阅读。

根据 PROCEDURE ANALYSE() 的输出，会发现可以对表进行更改以利用更有效的类型。如果希望更改值类型，使用 ALTER TABLE 语句即可。

- **将数据装入 BLOB。**用 BLOB 存储应用程序中包装或未包装的数据，有可能使原来需要几个检索操作才能完成的数据检索得以在单个检索操作中完成。而且还对存储标准表结构不易表示的数据或随时间变化的数据有帮助。在第 3 章 ALTER TABLE 语句的介绍中，有一个例子处理存储来自 Web 问卷的结果的表。该例子中讨论了在问卷中增加问题时，怎样利用 ALTER TABLE 向该表追加列。

解决该问题的另一个方法是让处理 Web 的应用程序将数据包装成某种数据结构，然后将其插入单个 BLOB 列。这样会增加应用程序对数据进行解码的开销（而且从表中检索出记录后要对其进行编码），但是简化了表的结构，并且不用在更改问卷时对表进行更改。

另一方面，BLOB 值也有自己的固有问题，特别是在进行大量的 DELETE 或 UPDATE 操作时更是如此。删除 BLOB 会在表中留下一个大空白，在以后将需用一个记录或可能是不同大小的多个记录来填充。

- **对容易产生碎片的表使用 OPTIMIZE TABLE。**大量进行修改的表，特别是那些含有可变长列的表，容易产生碎片。碎片不好，因为它在存储表的磁盘块中产生不使用的空间。随着时间的增长，必须读取更多的块才能取到有效的行，从而降低了性能。任意具有可变长行的表都存在这个问题，但这个问题对 BLOB 列更为突出，因为它们尺寸的变化非常大。经常使用 OPTIMIZE TABLE 有助于保持性能不下降。
- **使用合成索引。**合成索引列有时很有用。一种技术是根据其他列建立一个散列值，并将其存储在一个独立的列中，然后可通过搜索散列值找到行。这只对精确匹配的查询有效。（散列值对具有诸如“<”或“>=”这样的操作符的范围搜索没有用处）。在 MySQL 3.23 版及以上版本中，散列值可利用 MD5() 函数产生。

散列索引对 BLOB 列特别有用。有一事要注意，在 MySQL 3.23.2 以前的版本中，

不能索引 BLOB 类型。甚至是在 3.23.2 或更新的版本中，利用散列值作为标识值来查找 BLOB 值也比搜索 BLOB 列本身更快。

■ **除非有必要，否则应避免检索较大的 BLOB 或 TEXT 值。**例如，除非肯定 WHERE 子句能够将结果恰好限制在所想要的行上，否则 SELECT * 查询不是一个好办法。这样做可能会将非常大的 BLOB 值无目的地从网络上拖过来。这是存储在另一列中的 BLOB 标识信息很有用的另一种情形。可以搜索该列以确定想要的行，然后从限定的行中检索 BLOB 值。

■ **将 BLOB 值隔离在一个独立的表中。**在某些情况下，将 BLOB 列从表中移出放入另一个副表可能具有一定的意义，条件是移出 BLOB 列后可将表转换为定长行格式。这样会减少主表中的碎片，而且能利用定长行的性能优势。

4.4 有效地装载数据

很多时候关心的是优化 SELECT 查询，因为它们是最常用的查询，而且确定怎样优化它们并不总是直截了当。相对来说，将数据装入数据库是直截了当的。然而，也存在可用来改善数据装载操作效率的策略，其基本原理如下：

- 成批装载较单行装载更快，因为在装载每个记录后，不需要刷新索引高速缓存；可在成批记录装入后才刷新。
- 在表无索引时装载比索引后装载更快。如果有索引，不仅必须增加记录到数据文件，而且还要修改每个索引以反映增加了的新记录。
- 较短的 SQL 语句比较长的 SQL 语句要快，因为它们涉及服务器方的分析较少，而且还因为将它们通过网络从客户机发送到服务器更快。

这些因素中有一些似乎微不足道（特别是最后一个因素），但如果要装载大量的数据，即使是很小的因素也会产生很大的不同结果。我们可以利用上述的一般原理推导出几个关于如何最快地装载数据的实际结论：

- LOAD DATA（包括其所有形式）比 INSERT 效率高，因为其成批装载行。索引刷新较少，并且服务器只需分析和解释一条语句而不是几条语句。
- LOAD DATA 比 LOAD DATA LOCAL 效率更高。利用 LOAD DATA，文件必须定位在服务器上，而且必须具有 FILE 权限，但服务器可从磁盘直接读取文件。利用 LOAD DATA LOCAL，客户机读取文件并将其通过网络发送给服务器，这样做很慢。
- 如果必须使用 INSERT，应该利用允许在单个语句中指定多行的形式，例如：

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

可在语句中指定的行越多越好。这样会减少所需的语句数目，降低索引刷新量。

如果使用 mysqldump 生成数据库备份文件，应该使用 --extended-insert 选项，使转储文件包含多行 INSERT 语句。还可以使用 --opt（优化），它启用 --extended-insert 选项。反之，应该避免使用 mysqldump 的 --complete-insert 选项；此选项会导致 INSERT 语句为单行，执行时间更长，比不用 --complete-insert 选项生成的语句需要更多的分析。

- 使用压缩了的客户机/服务器协议以减少网络数据流量。对于大多数 MySQL 客户机，可以用 --compress 命令行选项来指定。它一般只用于较慢的网络，因为压缩需要占用大量的处理器时间。

■ 让 MySQL 插入缺省值；不要在 INSERT 语句中指定将以任意方式赋予缺省值的列。平均来说，这样做语句会更短，能减少通过网络传送给服务器的字符数。此外，语句包含的值较少，服务器所进行的分析和转换就会较少。

■ 如果表是索引的，则可利用批量插入（LOAD DATA 或多行的 INSERT 语句）来减少索引的开销。这样会最小化索引更新的影响，因为索引只需要在所有行处理过时才进行刷新，而不是在每行处理后就刷新。

■ 如果需要将大量数据装入一个新表，应该创建该表且在未索引时装载，装载数据后才创建索引，这样做较快。一次创建索引（而不是每行修改一次索引）较快。

■ 如果在装载之前删除或禁用索引，装入数据后再重新创建或启用索引可能使装载更快。

如果想对数据装载使用删除或禁用策略，一定要做一些实验，看这样做是否值得（如果将少量数据装入一个大表中，重建和索引所花费的时间可能比装载数据的时间还要长）。

可用 DROP INDEX 和 CREATE INDEX 来删除和重建索引。另一种可供选择的方法是利用 myisamchk 或 isamchk 禁用和启用索引。这需要在 MySQL 服务器主机上有一个帐户，并对表文件有写入权。为了禁用表索引，可进入相应的数据库目录，执行下列命令之一：

```
% myisamchk --keys-used=0 tbl_name
% isamchk --keys-used=0 tbl_name
```

对具有 .MYI 扩展名的索引文件的 MyISAM 表使用 myisamchk，对具有 .ISM 扩展名的索引文件的 ISAM 表使用 isamchk。在向表中装入数据后，按如下激活索引：

```
% myisamchk --recover --quick --keys-used=n tbl_name
% isamchk --recover --quick --keys-used=n tbl_name
```

n 为表具有的索引数目。可用 --description 选项调用相应的实用程序得出这个值：

```
% myisamchk --description tbl_name
% isamchk --description tbl_name
```

如果决定使用索引禁用和激活，应该使用第 13 章中介绍的表修复锁定协议以阻止服务器同时更改锁（虽然此时不对表进行修复，但要对它像表修复过程一样进行修改，因此需要使用相同的锁定协议）。

上述数据装载原理也适用于与需要执行不同操作的客户机有关的固定查询。例如，一般希望避免在频繁更新的表上长时间运行 SELECT 查询。长时间运行 SELECT 查询会产生大量争用，并降低写入程序的性能。一种可能的解决方法为，如果执行写入的主要是 INSERT 操作，那么先将记录存入一个临时表，然后定期地将这些记录加入主表中。如果需要立即访问新记录，这不是一个可行的方法。但只要能在一个较短的时间内不访问它们，就可以使用这个方法。使用临时表有两个方面的好处。首先，它减少了与主表上 SELECT 查询语句的争用，因此，执行更快。其次，从临时表将记录装入主表的总时间较分别装载记录的总时间少；相应的索引高速缓存只需在每个批量装载结束时进行刷新，而不是在每行装载后刷新。

这个策略的一个应用是进入 Web 服务器的 Web 页访问 MySQL 数据库。在此情形下，可能没有保证记录立即进入主表的较高权限。

如果数据并不完全是那种在系统非正常关闭事件中插入的单个记录，那么减少索引刷新的另一策略是使用 MyISAM 表的 DELAYED_KEY_WRITE 表创建选项（如果将 MySQL 用于某些数据录入工作时可能会出现这种情况）。此选项使索引高速缓存只偶尔刷新，而不是在每次插入后都要刷新。

如果希望在服务器范围内利用延迟索引刷新, 只要利用 `--delayed-key-write` 选项启动 `mysqld` 即可。在此情形下, 索引块写操作延迟到必须刷新块以便为其他索引值腾出空间为止, 或延迟到执行了一个 `flush-tables` 命令后, 或延迟到该索引表关闭。

4.5 调度与锁定问题

前面各段主要将精力集中在使个别的查询更快上。MySQL 还允许影响语句的调度特性, 这样会使来自几个客户机的查询更好地协作, 从而单个客户机不会被锁定太长的时间。更改调度特性还能保证特定的查询处理得更快。我们先来看一下 MySQL 的缺省调度策略, 然后来看看为改变这个策略可使用什么样的选项。出于讨论的目的, 假设执行检索 (SELECT) 的客户机程序为读取程序。执行修改表操作 (DELETE, INSERT, REPLACE 或 UPDATE) 的另一个客户机程序为写入程序。

MySQL 的基本调度策略可总结如下:

- 写入请求应按其到达的次序进行处理。
- 写入具有比读取更高的优先权。

在表锁的帮助下实现调度策略。客户机程序无论何时要访问表, 都必须首先获得该表的锁。可以直接用 `LOCK TABLES` 来完成这项工作, 但一般服务器的锁管理器会在需要时自动获得锁。在客户机结束对表的处理时, 可释放表上的锁。直接获得的锁可用 `UNLOCK TABLES` 释放, 但服务器也会自动释放它所获得的锁。

执行写操作的客户机必须对表具有独占访问的锁。在写操作进行中, 由于正在对表进行数据记录的删除、增加或更改, 所以该表处于不一致状态, 而且该表上的索引也可能需要作相应的更新。如果表处于不断变化中, 此时允许其他客户机访问该表会出问题。让两个客户机同时写同一个表显然不好, 因为这样会很快使该表不可用。允许客户机读不断变化的表也不是件好事, 因为可能在读该表的那一刻正好正在对它进行更改, 其结果是不正确的。

执行读取操作的客户机必须有一把防止其他客户机写该表的锁, 以保证读表的过程中表不出现变化。不过, 该锁无需对读取操作提供独占访问。此锁还允许其他客户机同时对表进行读取。读取不会更改表, 所有没必要阻止其它客户机对该表进行读取。

MySQL 允许借助几个查询限修饰符对其调度策略施加影响。其中之一是 `DELETE`、`INSERT`、`LOAD DATA`、`REPLACE` 和 `UPDATE` 语句的 `LOW_PRIORITY` 关键字。另一个是 `SELECT` 语句的 `HIGH_PRIORITY` 关键字。第三个是 `INSERT` 和 `REPLACE` 语句的 `DELAYED` 关键字。

`LOW_PRIORITY` 关键字按如下影响调度。一般情况下, 如果某个表的写入操作在表正被读取时到达, 写入程序被阻塞, 直到读取程序完成, 因为一旦某个查询开始, 就不能中断。如果另一读取请求在写入程序等待时到达, 此读取程序也被阻塞, 因为缺省的调度策略为写入程序具有比读取程序高的优先级。在第一个读取程序结束时, 写入程序继续, 在此写入程序结束时, 第二个读取程序开始。

如果写入请求为 `LOW_PRIORITY` 的请求, 则不将该写入操作视为具有比读取操作优先级高的操作。在此情形下, 如果第二个读取请求在写入程序等待时到达, 则让第二个读取操作排在等待的写入操作之前。仅当没有其他读取请求时, 才允许写入程序执行。这种调度的更改从理论上说, 其含义为 `LOW_PRIORITY` 写入可能会永远被阻塞。当正在处理前面的读

取请求时，只要另一个读取请求到达，这个新的请求允许排在 `LOW_PRIORITY` 写入之前。

`SELECT` 查询的 `HIGH_PRIORITY` 关键字作用类似。它使 `SELECT` 插在正在等待的写入操作之前，即使该写入操作具有正常的优先级。

`INSERT` 的 `DELAYED` 修饰符作用如下，在表的一个 `INSERT DELAYED` 请求到达时，服务器将相应的行放入一个队列，并立即返回一个状态到客户机程序，以便该客户机程序可以继续执行，即使这些行尚未插入表中。如果读取程序正在对表进行读取，那么队列中的行挂起。在没有读取时，服务器开始开始插入延迟行队列中的行。服务器不时地停下来看看是否有新的读取请求到达，并进行等待。如果是这样，延迟行队列将挂起，并允许读取程序继续。在没有其他的读取操作时，服务器再次开始插入延迟行。这个过程一直进行到延迟行队列空为止。

此调度修饰符并非出现在所有 `MySQL` 版本中。下面的表列出了这些修饰符和支持这些修饰符的 `MySQL` 版本。可利用此表来判断所使用的 `MySQL` 版本具有什么样的功能：

语句类型	开始出现的版本
<code>DELETE LOW_PRIORITY</code>	3.22.5
<code>INSERT LOW+PRIORITY</code>	3.22.5
<code>INSERT DELAYED</code>	3.22.15
<code>LOAD DATA LOW_PRIORITY</code>	3.23.0
<code>LOCK TABLES ... LOW_PRIORITY</code>	3.22.8
<code>REPLACE LOW_PRIORITY</code>	3.22.5
<code>REPLACE DELAYED</code>	3.22.15
<code>SELECT ... HIGH_PRIORITY</code>	3.22.9
<code>UPDATE LOW_PRIORITY</code>	3.22.5
<code>SET SQL_LOW_PRIORITY_UPDATES</code>	3.22.5

INSERT DELAYED 在客户机方的作用

如果其他客户机可能执行冗长的 `SELECT` 语句，而且您不希望等待插入完成，此时 `INSERT DELAYED` 很有用。发布 `INSERT DELAYED` 的客户机可以更快地继续执行，因为服务器只是简单地将要插入的行插入。

不过应该对正常的 `INSERT` 和 `INSERT DELAYED` 性能之间的差异有所认识。如果 `INSERT DELAYED` 存在语法错误，则向客户机发出一个错误，如果正常，便不发出信息。例如，在此语句返回时，不能相信所取得的 `AUTO_INCREMENT` 值。也得不到惟一索引上的重复数目的计数。之所以这样是因为此插入操作在实际的插入完成前返回了一个状态。其他还表示，如果 `INSERT DELAYED` 语句的行在等待插入中被排队，并且服务器崩溃或被终止（用 `kill -9`），那么这些行将丢失。正常的 `TERM` 终止不会这样，服务器会在退出前将这些行插入。

4.6 管理员的优化

前面各段介绍了普通的 `MySQL` 用户利用表创建和索引操作，以及利用查询的编写能够进行的优化。不过，还有一些只能由 `MySQL` 管理员和系统管理员来完成的优化，这些管理员在 `MySQL` 服务器或运行 `MySQL` 的机器上具有控制权。有的服务器参数直接适用于查询处理，可将它们打开。而有的硬件配置问题直接影响查询处理速度，应该对它们进行调整。

4.6.1 服务器参数

服务器有几个能够改变从而影响其操作的参数（或称变量）。有关服务器参数优化的综合介绍请参见第11章，但其中几个参数主要与查询有关，有必要在此提一下：

■ delayed_queue_size

此参数在执行其他 INSERT DELAYED 语句的客户机阻塞以前，确定来自 INSERT DELAYED 语句的放入队列的行的数目。增加这个参数的值使服务器能从这种请求中接收更多的行，因而客户机可以继续执行而不阻塞。

■ key_buffer_size

此参数为用来存放索引块的缓冲区尺寸。如果内存多，增加这个值能节省索引创建和修改的时间。较大的值使 MySQL 能在内存中存储更多的索引块，这样增加了在内存中找到键值而不用读磁盘块的可能性。

在 MySQL 3.23 版及以后的版本中，如果增加了键缓冲区的尺寸，可能还希望用 --init-file 选项启动服务器。这样能够指定一个服务器启动时执行的 SQL 语句文件。如果有想要存放在内存中的只读表，可将它们拷贝到索引查找非常快的 HEAP 表。

4.6.2 硬件问题

可利用硬件更有效地改善服务器的性能：

■ **在机器中安装更多的内存。** 这样能够增加服务器的高速缓存和缓冲区的尺寸，使服务器更经常地使用存放在内存中的信息，降低从磁盘取信息的要求。

■ **如果有足够的 RAM 使所有交换在内存文件系统中完成，那么应该重新配置系统，去掉所有磁盘交换设置。** 否则，即使有足以满足交换的 RAM，某些系统仍然要与磁盘进行交换。

■ **增加更快的磁盘以减少 I/O 等待时间。** 寻道时间是这里决定性能的主要因素。逐字地移动磁头是很慢的，一旦磁头定位，从磁道读块则较快。

■ **在不同的物理设备上设法重新分配磁盘活动。** 如果可能，应将您的两个最繁忙的数据库存放在不同的物理设备上。请注意，使用同一物理设备上的不同分区是不够的。这样没有帮助，因为它们仍将争用相同的物理资源（磁盘头）。移动数据库的过程在第 10 章中介绍。

在将数据重新放到不同设备之前，应该保证了解该系统的装载特性。如果在特定的物理设备上已经有了某些特定的主要活动，将数据库放到该处实际上可能会使性能更坏。例如，不要把数据库移到处理大量 Web 通信的 Web 服务器设备上。

■ **在设置 MySQL 时，应该配置其使用静态库而不是共享库。** 使用共享库的动态二进制系统可节省磁盘空间，但静态二进制系统更快（然而，如果希望装入用户自定义的函数，则不能使用静态二进制系统，因为 UDF 机制依赖于动态连接）。

第二部分 MySQL 编程接口

第5章 MySQL 程序设计介绍

在本书的这部分中，我们将讨论编写自己的访问 MySQL 数据库的程序所需要知道的内容。MySQL 有一组实用程序。例如，mysqldump 导出表的上下文和结构定义，mysqlimport 将数据文件加载到表中，mysqladmin 实现管理操作，mysql 可以使用户与服务器交互来执行任意的查询。每个标准的 MySQL 实用程序都倾向于小巧，重点放在程序可完成特定的、有限的功能。即使 mysql 也是如此，从感觉上说，mysql 比其他实用程序更灵活，因此可以用它来执行任何数量的各种查询，即它就是为允许向服务器直接发布 SQL 查询，并可查看查询结果这一单一目的而设计的。

MySQL 客户机这种有限的特性并不是缺点，而是特意设计的。程序是具有通用目的的实用程序；它们并不试图预料您所想做的所有可能的需要。MySQL 的开发者们不赞成编写大型的、臃肿的程序来试图做可能想去做的每件事情（而且这样做的结果将使程序中包括大量的您根本不关心的事情的代码）。然而，有时有些应用确实有常规客户机的能力所无法处理的需求。为了处理这些情况，MySQL 提供一个客户机编程库。这允许您编写自己的程序，满足您的应用程序可能具有的任何特定需求。通过允许您对 MySQL 服务器的访问，客户机的开放程度只受您自己想象力的限制了。

编写自己的程序可以获取如何特殊的能力呢？让我们比较一下 mysql 客户机和其没有附加代码的接口对 MySQL 服务器的访问：

- 可以定制输入处理。用 mysql 可以输入原始的 SQL 语句。用自己的程序，可以为用户提供使用起来更直观、更容易的输入方法。用程序可使用户不必知道 SQL —— 甚至不必知道在完成的任务中数据库承担的角色。

输入信息的采集可能是像命令行风格的提示和值读取这样基本的方式，或者可能是使用屏幕管理程序包（如 curses 或 S-Lang）、使用 Tcl/Tk 的 X 窗口或 Web 浏览器格式实现的基于屏幕输入那样复杂的方式。

对大多数人来说，通过填写一定的格式来指定搜索参数的形式比通过发布 SELECT 语句更容易。例如，一位房地产经纪入，要寻找一定价格范围、风格或位置的房屋，只要将查寻参数输入到表格中，就可以最小的代价得到符合条件的内容。输入新记录或更新已有记录也类似地考虑这种应用。在数据输入部门的键盘操作员应该不需要知道像 INSERT、REPLACE 或 UPDATE 这样的 SQL 语法。

在最终用户和 MySQL 服务器之间提出输入采集层的另一个原因是可以校验用户提供的输入。例如，可以检查数据，确认它们是符合 MySQL 要求的格式，或可以要求填写特定的区域。

- 可以定制输出。mysql 的输出基本上是无格式的；可以选择以制表符为分隔符或以表格

形式输出。如果想要使输出结果看起来比较好，则必须自己对它进行格式化。这些需求可能像打印“Missing”而不是 NULL 这样简单，也可能更复杂。考虑下面的报告：

State	City	Sales
AZ	Mesa	\$94,384.24
	Phoenix	\$17,328.28
	subtotal	\$117,712.52
CA	Los Angeles	\$118,198.18
	Oakland	\$38,838.36
	Subtotal	\$157,036.54
	TOTAL	\$274,749.06

这个报告包括几个特定的元素：

- 定制标题。
- 在 State 列中重复值的抑制以便只在更改时才将这些值打印出来。
- 小计和总计的计算。
- 数字格式，如 94384.24，打印为美元数量为 \$94,384.24。

对于一些任务，甚至可能不需要任何输出。您可能正在对计算向后插入到另一个数据库表中的结果进行简单地检索信息。除了用户运行这个查询以外，甚至可能还想将这个结果输出到其他地方。例如，如果正在提取姓名和电子邮件地址以自动地填入为批量电子邮件生成信件格式的过程中，则程序产生输出。但是该输出由邮件接受者的信息组成，而没有运行程序人员的信息。

- **可以在 SQL 自身施加的约束条件的环境下工作。** SQL 不是一种带有条件选择、循环和子例程的流程控制结构的过程语言。SQL 脚本包括一组从开始到结束一次一个的可执行语句，具有最低限度的错误检查。

如果在批处理模式中使用 mysql 执行 SQL 查询的一个文件，则 mysql 在出现第一个错误后退出，或者，如果指定 --force 选项，则不管出现多少错误，都不加选择地执行所有查询。程序可以围绕语句提供流程控制，以便可以有选择地适应查询的成功或失败。可以根据另一个查询的成功或失败来执行一个查询，或根据前一个查询的结果来决定下一步要做的事情。

SQL 具有非常有限的语句间的连续性，这点也被带到 mysql 中。使用一个查询的结果，并将它们应用于另一个查询中，或将多个查询结果联系在一起是困难的。LAST_INSERT_ID() 可用于获取由前一个语句最新生成的 AUTO_INCREMENT 值，仅仅是关于它的。

更一般的情况是，要想检索一组记录，然后使用每一条记录作为一系列复杂的进一步操作的基础是困难的。例如，检索一个消费者列表然后查询每个消费者的详细信用历史，对每个客户来说可能要包括若干个查询。在某些情况下，可能想开发票，在发票头写上需要联系的客户信息，然后按序列出每项条目。mysql 不适合这些类型的任务，因为可能需要依赖于前几个查询结果的若干查询，并且这些任务超出了 mysql 的布局设计的能力。

一般来说，除了 mysql 外，还需要工具来执行包括主细目关系和具有复杂输出格式需求的任务。程序提供将查询连接在一起的“胶”；并可用一个查询的输出作为另一个查询的输入。

- **可以将 MySQL 集成到任何应用程序中。**许多程序都利用数据库的能力提供信息。通

过发布一个快速查询，应用程序可以校验消费者号或检查一项条目是否在产品清单中。假设一个客户要寻找某些作者的所有书，则 Web 应用程序可以在数据库中查找它们，然后将结果显示在该客户的浏览器上。

通过使用调用带有包含 SQL 语句的输入文件的 mysql 的外壳脚本（shell script），可以实现一种初步的“集成”，然后，再使用其他UNIX实用程序加工这些输出。但是这可能变得很难看，特别是当任务变得更复杂时。当应用程序不断增长成为杂乱的修补工作时，它也可能产生一种“在工作，但觉得有错误”的感觉。此外，运行其他命令的外壳脚本的创建过程的开销可能超过您的预想。但它可能更有效率地与 MySQL 服务器直接交互，当在应用程序执行的每个阶段需要它的时候，都可以精确地提取想要的信息。

针对我们在第1章“MySQL 和 SQL 介绍”中安装的样例数据库 samp_db，我们已经列举了若干需要自己编写与 MySQL 服务器交互的节目的目标。这些目标中的一些显示在下面的列表中：

- 为打印而格式化 Historical League 目录。
- 考虑外观和联机目录的寻找。
- 通过电子邮件向成员发送补充通知。
- 使用 Web 浏览器很容易地将分数输入到学分册中。

在一些细节方面，我们将考虑的一个方面是将 MySQL 的能力与 Web 环境结合起来。MySQL 不直接提供对 Web 应用程序的支持，但通过组合带有适当的工具的 MySQL，通过 Web 可以很容易地访问数据库。使用 Web 服务器可以指定查询，向客户的浏览器报告结果。

将 MySQL 和 Web 结合可能有两个想法：

- 主要的兴趣在于数据库，只是想使用 Web 作为工具来获取对数据更容易的访问。在这样的想法下，数据库的位置是清楚且明显的，因为它是兴趣的焦点。例如，可以编写 Web 页来允许查看数据库所包含的表、表的结构，及表的内容。您打算使用 Web 服务器来提高对 MySQL 的访问能力。这可能也是 MySQL 管理者的观点。
- 主要的兴趣可能是 Web 站点，为了使站点的内容对访问者更有价值，您可能想使用 MySQL 作为一个工具。例如，如果为站点的访问者运行信息板或讨论清单，则可以使用 MySQL 保留信息的轨迹。在这种情况下，数据库的角色更微妙，访问者甚至可以不关心您必须提供给他服务器中执行的部分。您打算使用 MySQL 提高 Web 服务器的能力。这可能也是 Web 站点开发者的一个观点。

这些想法并不矛盾。例如，在 Historical League 情况下，我们想通过允许联机输入来作为成员获取访问成员目录内容的一种方法而使用 Web。提供对数据库的访问是 Web 的一个用法。同时，League 的 Web 站点在某些方面有些不完全，所以向站点增加目录内容，以便为成员提高站点的价值。增强站点所提供的服务是数据库的一种用法。

无论您如何看待 MySQL 与 Web 的结合，实现方法都是类似的，即将前台的 Web 站点与后台的 MySQL 连接，使用 Web 服务器作为媒介。Web 服务器将查询从用户发送到 MySQL 服务器，检索查询结果，然后将它们传送给客户，在浏览器上显示。

当然，不一定要联机处理数据，但这样做往往有好处，特别是与经过标准的 MySQL 客户机程序访问数据做比较时：

- 通过 Web 访问数据，人们可以使用他们喜欢的浏览器，在他们喜欢的平台上运行。他们不限制 MySQL 客户机程序所运行的系统。Web 浏览器更是这样，无论 MySQL 客

户机分布如何广泛。

- Web 界面的使用比独立命令行的 MySQL 客户机程序的使用更简单。
- Web 界面可以根据特殊应用程序的要求来定制。而 MySQL 客户机程序是用固定接口来完成基本功能的工具。
- 动态 Web 页面扩充了 MySQL 的能力，它可以做到用 MySQL 客户机很难做到或根本不可能做到的事情。例如，仅用 MySQL 客户机程序不可能真正地使一体化购买车辆的应用程序组合成整体。

任何编程语言都可用来编写基于 Web 的应用程序，但是，有些语言比其他语言更适合一些。请参阅 5.2 节“选择 API”，我们将看到这点。

5.1 MySQL 可用的 API

为了方便应用程序的开发，MySQL 提供了用 C 编程语言编写的客户机库，它允许从任何 C 程序的内部访问 MySQL 数据库。客户机库实现应用程序编程接口（API），API 定义客户机程序如何建立和执行与服务器的通信。

然而，使用 C 来编写 MySQL 程序并不受限制。许多其他语言处理器本身也是由 C 编写的，或具有使用 C 库的能力，所以 MySQL 客户机库提供了这个方法，由此，MySQL 对这些语言的约束可以建立在 C API 的上面。这就为与 MySQL 服务器通信而编写应用程序提供了许多选择。客户机程序的 API 是用 Perl、PHP、Java、Python、C++、Tcl 和其他一些语言编写的。有关最新的清单，请查看 MySQL 参考指南或 MySQL Web 站点，因为有时会增加用新语言编写的 API。

每种语言约束都定义自己的接口，特别是访问 MySQL 的规则。这里没有足够的时间来讨论 MySQL 可使用的每种 API，我们只讲述最流行的三种：

- **C 客户机库 API。**这是 MySQL 的基本编程接口。
- **Perl 通用目标脚本语言的 DBI（数据库接口）API。**DBI 是作为与其他模块在 DBD（数据库驱动程序）级接口的 Perl 模块来实现的，每个模块都提供对特定类型的数据库引擎的访问（当然，我们将讨论的特定的 DBD 模块也提供对 MySQL 的支持）。DBI 对 MySQL 的最普遍用法是编写由命令行来调用的独立的客户机，以及试图由 Web 服务器调用的脚本来提供 Web 对 MySQL 的访问。
- **PHP API。**PHP 是一种脚本语言，它提供了在 Web 页中嵌入程序的一种便利的方法。在发送以前，这样的页面由 PHP 来处理，它允许这些脚本生成动态的内容，如在页面中包括 MySQL 查询的结果。“PHP”原始的意思是个人主页（Personal Home Page），但是 PHP 的成长已经远远超过它简单的原始功能。PHP Web 站点现在使用的这个名称表示“PHP：超文本预处理程序（Hypertext Preprocessor）”，它像 GNU（是 GUN 而不是 UNIX）一样以同样的方式自我引用。

使用他人成果

当标准的 MySQL 客户机不能满足需要时，您并不总是需要编写自己的程序。其他一些人一直编写程序，而这些程序中有许多是可共享得到的。请参阅附录 I 中的一些样例。只要找到几个就能节省您的许多工作。

以上这三种 API 都有专门章节详细说明。本章只提供对 API 比较的概述，用来说明它们的基本特征，并给出对特定的应用程序可能选择某个而不是其他 API 的原因。

当然，不必只考虑某个 API，应了解每个 API，并用可以明智选择适合自己的 API。在包括若干组件的大项目中，可能使用多个 API，多种语言，这取决于每个子任务适合哪一种语言。

对于试图使用的任何一种 API，如果需要得到必需的软件，请参阅附录 A。

5.1.1 C API

C API 用于编译 C 程序上下文环境内部。它是一种客户机库，提供可用来与 MySQL 服务器对话的最低级别的接口——具有创建与服务器通信所需的能力。

DBI 和 PHP 的前身

DBI 的 Perl 前身是 Mysqlperl 模块 Mysql.pm。这个模块不再被支持，而且不应该用于新的 MySQL 的开发。有一件事需要明白，Mysqlperl 是依赖于 MySQL 的，但 DBI 不是。如果编写 MySQL 的 Perl 应用程序，然后，决定想用另外一种数据库引擎来使用它们，则移植 DBI 脚本比 Mysqlperl 脚本更容易一些，因为它们很少依赖于特定的数据库引擎。

如果获取了访问 MySQL 的一段 Perl 脚本，并发现它是用 Mysqlperl 而不是 DBI 编写的，则仍然可以使用 DBI。DBI 包括了对 Mysqlperl 的仿真支持，因此不需要安装两个程序包。

PHP 3 的前身是 PHP/FI 2.0（FI 代表“form interpreter，即格式解释程序”）。像 Mysqlperl 一样，PHP/FI 也是过时的，所以我们就不再进一步讨论它了。

MySQL C API 的起源

如果已经有编写 mSQL RDBMS 程序的经验，那么将注意到 MySQL C API 类似于 mSQL 相应的 C API。当 MySQL 的开发者们开始实现他们的 SQL 引擎时，许多有用的共享实用程序可用于 mSQL。要想花费最小的难度将那些 mSQL 实用程序移植为 MySQL 的实用程序是可能的，可有意地将 MySQL API 设计为与 mSQL API 类似（MySQL 甚至带有与 mSQL API 函数名称相应的 MySQL 名称的简单的文本替代品的 msql2mysql 脚本。这个操作相对烦琐，实际上也照顾了许多涉及为使用 MySQL 而转换 mSQL 程序的工作）。

MySQL 分发包提供的 C 客户机是基于这个 API 的。C 客户机库也作为 MySQL 对其他语言约束的基础来提供服务，但 Java API 是一个例外。例如，通过连接 MySQL C 客户机库代码（这个过程在附录 A 中通过 DBI 和 PHP 安装指导来举例说明），MySQL 可用 Perl DBI 模块专有的 MySQL 驱动程序和 PHP 代码。

5.1.2 Perl DBI API

DBI API 用于 Perl 脚本语言编写的应用程序的上下文环境内部。这种 API 在我们考虑的这三种 API 结构中是最高的，因为它可与许多数据库工作，而同时在脚本中可忽略许多特定数据库的细节。

DBI 经过使用两级结构的 Perl 模块来实现（请参阅图 5-1）：

■ **DBI(数据库接口)级**。为客户机脚本提供接口。这个级别提供的是抽象接口，并不是

指特定数据库引擎。

- **DBD(数据库驱动器) 级**。在这个级别由特定引擎的驱动程序来提供对各种数据库引擎的支持。

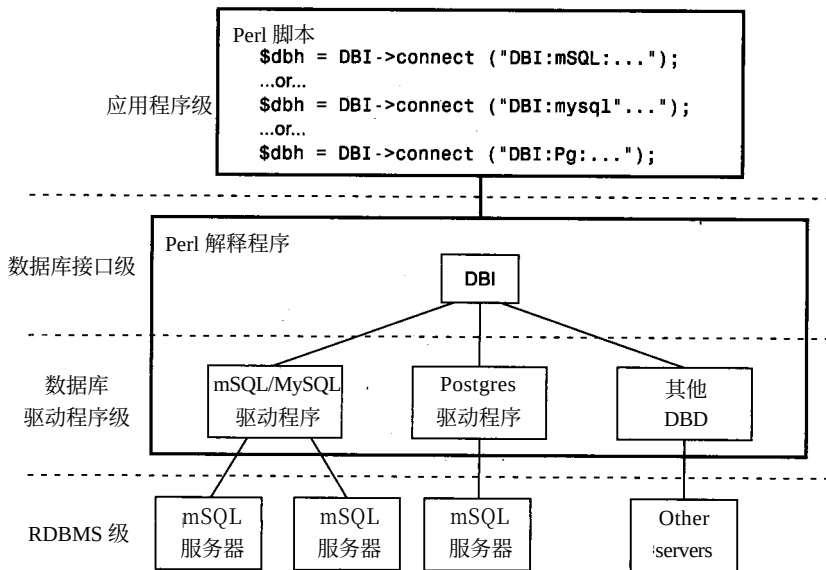


图5-1 DBI 体系结构

MySQL 对 DBI 的支持环境由 Msql-Mysql-modules 分发提供。这个模块在 DBD 级操作。可以从分发名称及图 5-1 中分辨它，一个驱动程序可以提供对一个以上的 RDBMS 的支持。Msql-Mysql-Modules 最初是为 mSQL 而编写的，后来扩展到 MySQL。这种影响类似于对 mSQL 和 MySQL 的 C API。由于设计的 MySQL C API 类似于 mSQL C API，所以将 mSQL DBD（使用 mSQL C API）扩展到对 MySQL 的使用很有意义。

DBI 体系结构编写应用程序的风格相对普通。当编写 DBI 脚本时，可使用一组标准的调用。DBI 级在 DBD 级调用适当的驱动程序来处理请求，对于想使用的特定数据库服务器通信中包括的特定问题，由驱动程序处理。DBD 级传送从服务器返回的数据，备份到 DBI 级，使数据出现在应用程序中。数据的格式与数据库的数据来源一致。

其结果得到这样一个接口，该接口从应用程序的编写者的观点隐藏了数据库引擎之间的差异，这样可使用多种不同的引擎——和驱动程序一样多。DBI 通过允许以统一风格访问每个数据库来提供一致性客户接口以增加可移植性。

当打开数据库时，出现由脚本编写的数据库专有的界面。当创建连接时，应指出使用哪个驱动程序。例如，要想使用 MySQL 数据库，应这样连接：

```
$dbh = DBI->connect ("DBI:mysql:...");
```

而要想使用 Postgres 或 mSQL，应这样连接：

```
$dbh = DBI->connect ("DBI:Pg:...");  
$dbh = DBI->connect ("DBI:mSQL:...");
```

连接以后，对该驱动程序不需要再做任何做特殊的引用。让 DBI 和该驱动程序解决数据库专有的细节。

无论如何这都是理论问题。然而，至少有两个因素与 DBI 脚本的可移植性矛盾：

- 在 RDBMS 引擎之间 SQL 的实现不同，为一个引擎编写的 SQL 另一个引擎根本不解是完全可能的。如果 SQL 相当通用，则脚本可在引擎之间作相应的移植。但如果 SQL 依赖于引擎，则脚本也是这样。例如，如果使用 MySQL 指定的 SHOW TABLES 语句，则该脚本不能用其他数据库执行。
- DBD 模块通常提供引擎专有类型的信息来允许脚本编写者使用特定数据库系统的特定功能。例如，MySQL DBD 提供访问查询结果中列属性的方法，如每列的最大长度、列是否是数值型的，等等。而这些属性在其他数据库中没有任何相似物。DBD 专有的特性与可移植性相背，通过使用它们，将 MySQL 编写的脚本用于其他数据库系统是困难的（然而，在第7章中，您将发现，笔者毫不费力地避免了由 MySQL DBD 提供的 MySQL 专有的结构。那是因为它应该知道那些结构是什么，以便可以决定自己是否使用它们）。

尽管存在数据库专有脚本的这两个因素，但以抽象方式提供数据库访问的 DBI 机制是完成可移植性的合理方式，只要您决定利用它多少次即可。

5.1.3 PHP API

像 Perl 一样，PHP 也是一种脚本语言。但它与 Perl 不同，PHP 很少作为通用目标语言来设计，而是作为编写 Web 应用程序的一种语言。PHP API 主要作为在 Web 页面中嵌入可执行脚本的一种方法来使用。这使 Web 的开发者们很容易用动态生成上下文环境来编写页面。当客户浏览器向 Web 服务器发送 PHP 页面的请求时，PHP 执行在该页面中它所发现的任何脚本，并用脚本的输出来替换它。该结果再送回浏览器。这就使浏览器中实际出现的页面根据请求的页面环境的不同而有所不同。例如，当在 Web 页面中嵌入下面简短的 PHP 脚本时，它出现所请求页面的主机 IP 地址：

```
<?php echo $REMOTE_ADDR; ?>
```

可以使用脚本为访问者提供基于数据库上下文环境的最新信息。下面的样例说明可用于 Historical League Web 站点的一个简单脚本。该脚本发布一个请求来确定当前的 League 的成员数目，并将该数目报告给访问该站点的人（如果出现错误，则该脚本不报告任何数目）：

```
<HTML>
<HEAD>
<TITLE>US Historical League</TITLE>
</HEAD>
<BODY>
<P>Welcome to the US Historical League Website.
<?php
$link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or exit ();
mysql_select_db ("samp_db")
    or exit ();
$result = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_array ($result))
    echo "<P>The League currently has " . $row[0] . " members";
mysql_free_result ($result);
?>
</BODY></HTML>
```


DBI和DBD的含义

尽管 DBI 级是独立于数据库的，而 DBD 级是依赖于数据库的，但那并不是“DBI”和“DBD”所代表的意义。它们的意思是“数据库接口”和“数据库驱动程序”。

PHP 脚本通常看起来像是带有嵌入在“<?php”和“?>”标识符中的脚本的 HTML 页面。一个页面可能包括若干个脚本。这为脚本的开发提供了一种非常灵活的方法。例如，如果您喜欢，可以编写一个正常的 HTML 页面来创建通用的页面框架，然后再增加脚本的内容。

对于不同的数据库引擎，PHP 对统一的接口不再作任何事情，DBI 也用这种方法。取而代之，每个引擎的接口看起来非常像相应的实现该引擎低级 API 的 C 库接口。例如，用于从 PHP 脚本内部访问 MySQL 的 PHP 函数的名称非常类似于 MySQL C 客户库中函数的名称。

5.2 选择API

本节介绍根据各种类型的应用程序选择 API 的方法，比较 C、DBI 和 PHP API 的能力，并给出它们相对的优点和缺点，并指出什么时候应选择哪一个。

首先应该指出，笔者不认为任一种语言优于其他语言。尽管笔者确实有自己的喜好，但还是统统使用它们。您也会有自己的喜好，像我的评论家一样。一个评论家会感觉应该强调 C 对 MySQL 编程的重要性，应将这种重要性上升到更重要的程度，而另一个评论家会认为 C 编程相当困难，应放弃使用它！您应当权衡本节中讨论的这些因素，得出自己的结论。

在对特定任务选择哪个 API 时，要考虑以下问题：

- **预期的执行环境。**期望使用应用程序的上下文环境。
- **性能。**当在 API 语言中编写时，如何使应用程序高效地执行。
- **开发的容易性。**如何便于 API 和它的语言编写应用程序。
- **可移植性。**除 MySQL 以外，应用程序是否还将用于其他数据库系统。

下面进一步分析每个问题。要注意这些因素的相互影响。例如，您想要一个运行良好的应用程序，但使用一个可快速开发该应用程序的语言也同等重要，即使该应用程序不能非常有效地运行也同样。

5.2.1 执行环境

当编写应用程序时，通常应考虑使用哪种环境。例如，该应用程序可能是从外壳程序中调用的报告生成器程序，或一个应付账目概要程序，在每月的月底作为 cron job 进行运行。从外壳程序或 cron 程序中运行的命令通常依赖它们自己，而且很少需要运行环境。另外，可以编写一个应用程序来试图由 Web 服务器调用。这样的程序期望能从它的运行环境中抽出非常特殊类型的信息：客户正在使用什么浏览器？在邮件清单订阅请求格式中输入什么参数？客户提供正确的口令访问我们个人信息了吗？

每种 API 语言都以它在这些不同的环境中适于编写应用程序而变化：

- C 是通用目标的语言，从理论上讲任何任务都可使用它。在实际中，C 倾向于用于更频繁的独立程序而不是对 Web 的编程。其原因可能是在 C 中不像在 Perl 或在 PHP 中那样容易地实现文本处理和内存管理，并且这些处理和管理在 Web 应用程序中大量地使用。

■ Perl, 像 C 一样, 适合于编写独立的程序。然而, 对于 Web 站点的开发, Perl 也是非常有用的, 例如通过使用 CGI.pm 模块。这使 Perl 成为编写连接 MySQL 和 Web 的应用程序的便利的语言。这样的应用程序可以经 CGI.pm 模块与 Web 接口, 并可以使用 DBI 与 MySQL 相互作用。

■ PHP 是设计用来编写 Web 应用程序的语言, 所以这个环境显然是最适合的。而且, 数据库访问是 PHP 最大的优势之一, 所以它是实现与 MySQL 相关的任务的 Web 应用程序最自然的选择。也可以将 PHP 作为一个独立的解释程序 (例如, 从外壳程序中运行脚本), 但不能非常频繁地使用它。

根据以上这些需要考虑的问题, 对于独立的应用程序, C 和 Perl 是最佳语言。对于 Web 应用程序, Perl 和 PHP 是最合适的。如果需要编写这两种类型的应用程序, 但又不会使用这些语言的任何一种, 并想用尽可能少的精力来学习, 则 Perl 可能是您最佳的选择。

5.2.2 性能

我们通常喜欢应用程序尽可能快地运行。然而, 实际上性能的重要性取决于所使用的程序的频率。对于一个月运行一次晚上定时工作的程序, 性能可能不是非常重要的。而对于在 Web 站点上一秒钟运行若干次的程序, 则每当排除一点无效性都会带来巨大的不同。后一种情况下, 在站点的有效性和请求中, 性能发挥着重要的作用。一个缓慢的站点是令用户苦恼的, 无论站点的内容如何, 如果您依靠站点作为一项收入来源, 则性能的降低直接影响收入。如果不能一次为多个连接提供服务, 访问者只会产生厌烦情绪而去其他的站点。

性能评价是一个复杂的问题。当编写特定的 API 时, 应用程序完成得好坏的最好指标是在这个 API 环境下编写并进行测试。而且最好的比较测试是在不同的 API 环境下多次运行该应用程序, 来比较每个版本。当然, 那不是一般的工作。一般来说, 您只想获取编写的应用程序。一旦它工作了, 如果它需要运行得更快, 您就可以考虑优化它, 使用更少的内存, 或有某些需要用其他方法提高的方面。但是, 至少有如下两个因素会影响性能:

■ 编译的程序比解释的程序运行得更快。

■ 对于在 Web 上下文环境中使用的解释语言, 在解释程序作为 Web 服务器自身的一部分而不是单独的过程模块被调用时, 性能更好。

1. 相对于解释语言的编译语言

编译的应用程序比用脚本语言编写的程序的同样版本效率更高、使用的内存更少, 并且执行得更快, 这是基本规律。这是由于执行脚本的语言的解释程序的开销问题。因为 C 是编译的, 而 Perl 和 PHP 是解释的, 所以 C 程序通常比 Perl 或 PHP 脚本运行得更快一些。

对于大量使用的程序, 通常用 C 是最好的选择。在 MySQL 分发包中包括的 mysql 命令行客户机程序就是最好的样例。

当然, 有一些因素能使这种明显的差别减小。对于一项任务, 可用 C 编写出更快的程序, 但也很有可能编写出低效率的 C 程序。用编译语言编写的程序并不自动地保证更好的性能。所以需要不断地考虑所做的事情。此外, 如果一个脚本化的应用程序需花费大部分时间来执行连接到解释程序引擎的 MySQL 客户机库例程的代码, 则编译程序和解释程序之间的差别将有所减少。

2. 相对于语言解释程序模块版本的独立程序

对于基于 Web 的应用程序，脚本语言解释程序通常以两种形式之一来使用，至少对 Apache 是这样，当编写 Web 应用程序时，Apache 是我们将使用的 Web 服务器：

- 可以安排 Apache 去调用这个解释程序作为单独的过程。当 Apache 需要运行 Perl 或 PHP 脚本时，它启动相应的程序，并告知它来执行该脚本。在这种情况下，Apache 使用该解释程序作为 CGI 程序，也就是说，它使用公共网关接口（Common Gateway Interface, CGI）协议与它们通信。
- 解释程序可用作直接连接到 Apache 二进制程序和作为其过程自身的一部分运行的模块。在 Apache 条件下，Perl 和 PHP 解释程序获得 `mod_perl` 和 `mod_php3` 模块的形式。

Perl 和 PHP 的提倡者们极力宣扬解释程序有速度优势，但所有的人都同意之所以喜欢解释程序是因为其运行的形式比语言本身有更大的诱惑力。在这两者中，解释程序作为模块运行比作为独立的 CGI 应用程序运行更快。

对于独立的应用程序，每当运行一个脚本时都必须启动该解释程序，所以将导致重大的创建过程的开销。当在已经运行 Apache 过程的内部作为模块使用时，解释程序可以立即从 Web 页面中访问。通过减少开销显著地提高了性能，并直接转换为快速处理获取的请求并发送它们的能力的增加。

独立解释程序启动的性能比模块解释程序的性能至少差一个数量级。当考虑 Web 页面服务包括少量处理的快速事务处理而不是具有许多处理时，解释程序启动的开销特别重要。如果花费许多时间只是为了启动而不是用于实际执行该脚本，则大部分资源一直处于等待状态。一天中的大部分时间可能花费在准备工作上，4 点到达，然后 5 点回家。

您可能想知道，为什么解释程序的模块版本由于必须一直启动 Apache 而能节省时间呢？。这个原因是，当 Apache 启动时，它立即产生一些子过程，用于处理收到的请求。当包括脚本执行的请求到达时，已经有 Apache 进程在准备等待去处理它。同样，Apache 的每个实例可服务于多个请求，所以该进程启动的开销只导致每组请求一次，而不是每个请求一次。

当 Perl 和 PHP 以模块形式安装时（像 `mod_perl` 和 `mod_php3`），哪一个完成得更好一些？那就是争论的主题，以下是适用于一般性的指南：

- Perl 将脚本转换为内部可编译的形式；而 PHP 却不这样。因此，一旦该脚本通过语法分析，则 Perl 可更快地执行它，特别是对于具有大量迭代的循环。
- `mod_perl` 可以运行脚本高速缓存来提高重复执行的脚本的性能。如果脚本在高速缓存中，则 Perl 可更快地开始执行脚本，因为它不需要再一次分析。否则，PHP 开始执行的该脚本的速度更快。
- `mod_perl` 比 PHP 有更大的内存覆盖区；利用 `mod_perl` 连接的 Apache 进程比利用 `mod_php3` 的更大。PHP 被假定必须在另一个进程的内部协同存在，并且在那个进程内部可以多次激活或撤消。Perl 被设计成从命令行作为独立程序运行，而不是作为被嵌入在 Web 服务器进程中的一个语言进行运行。这可能使它付出较大的内存覆盖区；Perl 是模块，因此它不运行在自身环境中。脚本的高速缓存和该脚本使用的附加 Perl 模块是付出较大的内存覆盖区的另外的因素。在这两种情况下，有更多的代码使用内存，并将运行的 Apache 进程保留在内存中。

在脚本运行速度方面，Perl 无论有什么可超过 PHP 的优势，都被 PHP 4 除掉了。PHP 4 在

它的能力和接口方面类似于 PHP 3，但它合并了 Zend，Zend 是一种更高性能的解释程序的引擎。

无论如何，所有的这些因素只导致 Perl 和 PHP 的模块版本之间性能比不同。无论您选择哪种语言，最重要的是尽可能地避免独立的解释程序。

解释程序的独立版本的确有一个优点超过它的模块版本，即可以安排它在不同的用户 ID 下运行。模块版本始终运行在与 Web 服务器相同的用户 ID 下，出于安全原因，该用户是一个典型的具有很少权限的账号。对于需要特殊权限的脚本来说不能很好地运行（例如，如果您需要能读写受保护的文件）。如果愿意，可以将模块方法和独立方法结合起来：缺省情况下使用模块版本，而在具有特定用户的权限的脚本的情况下使用独立版本。

降低 mod_perl 的内存需求

有些技术允许您只能对 mod_perl 使用某些的 Apache 进程。这样，只对运行 Perl 脚本的那些进程产生额外的内存开销。Apache Web 站点的 mod_perl 部分有可选择的各种策略的讨论（有关的详细信息，请参阅 <http://perl.apache.org/guide/>）。

综上考虑，也就是说，选择 Perl 还是 PHP，您应该试着从 Apache 模块中而不是通过调用一个单独的解释程序过程来使用它。只对不能由模块处理的那些情况使用独立的解释程序，例如需要特殊权限的脚本。对于这些实例，可以通过使用 Apache 的 suEXEC 机制在给定的用户 ID 下启动解释程序来处理脚本。

5.2.3 开发时间

刚才描述的这些因素影响应用程序的性能，但不能只考虑运行效率。时间及编程的简易性也是重要的，所以为 MySQL 编程选择 API 时要考虑的另一个因素是如何很快地开发出自己的应用程序。如果开发同样程序，用 Perl 脚本只需用 C 语言一半的时间，那您可能宁愿使用 Perl DBI API，而非 C API，即使开发出的应用程序运行速度不是非常快也如此。考虑程序的运行时间比考虑编写程序时花的时间更少一些通常是合理的，特别是对不经常运行的应用程序更是如此。您的一小时比机器的一小时要值钱得多！

一般来说，脚本语言编写程序更快，特别是得到应用程序的原型更快，这是由于以下两个原因：

第一，脚本语言提供更高级别的结构。这允许您在抽象的更高级别上进行思考，以便集中考虑要做什么，而不是考虑如何做。例如，Perl 的关联数组（散列）对于维护具有键/值关系（如学生 ID/学生姓名对）的数据节省了大量时间。C 没有这样的构造。如果想在 C 中实现这样的事情，则可能需要编写代码来处理许多低级的细节，其中包括内存管理和串操纵的问题，并且需要调试它，这也要花时间。

第二，脚本语言的开发周期的步骤较少。用 C 开发应用程序时，要经历通常的编辑—编译—测试的循环周期。每次修改程序时，在测试之前都必须重新编译它。而用 Perl 和 PHP，由于每次修改后不用编译就可以立即运行脚本，因此，开发周期可简化为编辑—测试。另一方面，编译程序对程序在严格的类型检查形式方面有更多的约束条件。编译程序施加的更多约束条件有助于避免在松散语言（如 Perl 和 PHP）中不易捕获的错误。在 C 中，如果您错误地拼写了变量的名称，则编译程序将警告您。PHP 不这样，Perl 也不这样，除非向它询问。当应用程序变得更大且更难于维护时，这些更严格的约束条件可能特别有用。

一般来说，在编译和解释语言之间权衡的是开发与性能的折衷：是想要使用编译语言开发程序，以便在运行时可以更快，但花费更多的时间来编写它？还是想要用脚本语言编写程序，以便在缩短编程时间，但要损失一些运行速度？

将两种方法合并起来也是可能的。编写一个脚本作为“第一个草案”来快速地开发出一个应用程序原型以测试其逻辑性，确定算法的可用性。如果这个程序有用，并且被频繁使用，则性能成为关注的焦点，这时可作为编译的应用程序重新对它编写代码。这样做给您带来两种方法的优点：快速得到应用程序的初始开发原型，同时得到最终产品的最佳性能。

从某种严格的意义来说，Perl DBI 和 PHP API 并没有给您在 C 客户机库中没有的能力。这是因为这两种 API 通过 MySQL C 库连接到 Perl 和 PHP 解释程序来获取对 MySQL 的访问。然而，对于嵌入 MySQL 的环境，C 与 Perl 或 PHP 有很大的不同。应考虑在与 MySQL 服务器相互作用时要做的事情并提问每个 API 语言如何帮助您完成这些事情。下面有一些样例：

■ **内存管理。**在C中，您发现自己对任何任务，包括动态分配数据结构都使用 malloc() 和 free() 来工作。Perl 和 PHP 可为您处理这些任务。例如，数组的大小自动地增加，并且可以使用动态长度的字符串而不用考虑内存管理。

■ **文本处理。**在这点 Perl 具有最大的开发能力，而 PHP 位于第二。比较起来，C 是非常初级的。

当然，可以用C编写自己的库，将内存管理和文本处理这样一些任务封装成更容易工作的函数。但是，然后还必须调试它们，并且您也想使自己的算法效率更高。在这两个方面，基本上可以断定，由于它们已经具有了通过许多双眼睛检查过的好处，对这些事情 Perl 和 PHP 中的算法一般是易于调试并且有合理的效率。通过利用其他人的工作可以节省您的时间（另一方面，如果解释程序偶尔有一个错误，您可能必须携带它，直到这个问题纠正为止。而用 C 编写时，可以更细地控制程序性能）。

这些语言的不同还在于它们的“安全性”。C API 提供对服务器最低级别的接口，而且强制的原则最少。从这种意义上说，它提供最低级的安全性网络。如果您超出正常顺序执行 API 函数，则可能获得一个“超出同步”的错误，或使程序崩溃。Perl 和 PHP 都提供了很好的保护。如果您没有以适当的顺序进行，则脚本失败，但是解释程序并不崩溃。在 C 程序中，出现崩溃错误的另一个非常可能的来源是动态分配内存和与它们相关的指针的使用。Perl 和 PHP 为您处理内存管理，所以您的脚本很少可能因为内存管理的错误而瘫痪。

开发时间受语言可用的外部支持的影响。C 可用的外部支持是将 MySQL C API 函数封装为更容易使用的实例的包装库的形式。这些库对于 C 和 C++ 都可用。Perl 无疑具有最大数量的附加软件，都是 Perl 模块的形式（与在 Apache 模块中具有的概念类似）。甚至有一个基础结构被设计来容易地定位并获取这些模块（即综合 Perl 归档网络 Comprehensive Perl Archive Network, CPAN）。使用 Perl 模块，不用编写代码就可以获取对所有类型的函数的访问。想要编写从数据库中生成报告的脚本，然后作为一个附件发送给某人吗？只要获取 MIME 模块中的一个，就立即具有附件生成的能力。

PHP 没有同样程度的外部支持（这并不令人惊奇，因为它是较新的语言）。也许所知道的最佳的附加软件是 PHP 基本库（PHP Base Library, PHPLIB）。根据名称和口令机制的一些排序，假设您正在编写需要限定只有经授权的用户才可以对某个 Web 页面访问的 Web 应用程

序。可以用任意语言编写对它的支持程序，但是如果使用 PHPLIB，则不必花费时间重新做这件事情。PHPLIB提供确认并且允许通过会话跟踪经授权的用户（从作为单个逻辑访问部分的给定客户机中连续页面的命中）。还可以分配给用户许可权，这允许您进行像定义具有更多权限的管理用户的工作。

5.2.4 可移植性

可移植性的问题与为访问 MySQL 引擎所编写的程序怎样才能容易地修改为使用不同引擎的程序有关。您可能不担心这个事情。然而，除非可以预知未来，否则，说“除了 MySQL 以外，我永远都不会将这个程序用在任何其他数据库上”可能有些冒险：假设您找到另一份工作，并想使用自己的旧程序，但您的新老板使用不同的数据库系统呢？

如果可移植性是需要优先考虑的事情，则应该考虑在 API 之间的区别：

- DBI API的可移植性最好，因为它独立于数据库是 DBI 设计的一个明确目标。
- PHP 的可移植性稍差，因为它不提供对 DBI 提供的各种数据库引擎的同样类型的统一接口。对每个支持数据库的 PHP 函数的调用类似于在相应的基础 C API 中的那些。稍有一些不同，但很少，需要更改您调用的数据库相关函数的名称。还有可能要修改一点应用程序的逻辑，因为不同数据库的借口并不都是以同样的方式工作的。
- C API 提供的数据库之间的可移植性最差。因为它生来就是为 MySQL 设计的。

当需要在同一个应用程序中访问多个数据库系统时，独立于数据库的可移植性特别重要。这可能包括像将数据从一个 RDBMS 移动到另一个 RDBMS 中的简单任务，或更复杂的任务，如基于从许多数据库系统中得到的信息生成报告。

DBI 和 PHP 都提供对访问多个数据库引擎的支持，所以对不同的数据库，甚至在不同的主机上，都可以很容易地同时连接到服务器上。然而，DBI 和 PHP 在对于从多个异构数据库系统中检索和处理数据的任务的适宜性方面有所不同。而 DBI 更好些，因为无论使用哪种数据库，它都使用一组单独的访问调用。假设想在 MySQL、mSQL 和 Postgres 数据库之间传送数据。使用 DBI，则使用这三种数据库的惟一不同之处在于用于连接到每个服务器的 DBI->connect()调用。而用 PHP时，可能需要有更复杂的脚本，该脚本将含有三组读取调用和三组写入调用。

多数据库应用程序的一个极好的例子是 MySQL 分发包中的 crash-me 脚本，它可测试许多不同的数据库服务器的能力。该脚本是用 DBI编写的，对于这样的应用程序，这种选择是很明显的，因为您可以同样的方式访问所有的数据库。

第6章 MySQL C API

MySQL提供用C编程语言编写的客户机库，可以用它编写访问 MySQL的客户机程序。这个库定义了应用程序编程接口，包括下面的实用程序：

- 建立和终止与服务器会话的连接管理例程。
- 构造查询的例程，将例程发送到服务器，并处理结果。
- 当其他C API调用失败时，确定错误准确原因的状态和错误报告函数。

本章介绍如何用客户机库编写自己的程序。我们要记住的一些要点是，自己的程序与MySQL 分发包中已有的客户机程序的一致性，代码的模块性和可重用性。本章假设您知道用C编程的一些知识，但并不一定是专家。

本章从简单到复杂粗略地开发了一系列的客户机程序。这个过程是第一部分开发了一个程序框架，该框架除了与服务器连接和断开以外不能作任何事情。这样做的原因是，尽管MySQL 客户机程序是为不同的目的而编写的，但它们都有一个共同点：即创建与服务器的连接。

我们将用以下步骤来建立这个程序框架：

1. 编写一些连接和断开的简要代码（客户机程序 1）。
2. 增加一些错误检查（客户机程序 2）。
3. 使连接代码模块化和可重用化（客户机程序 3）。
4. 增加获取运行时连接参数的能力（主机，用户，口令）（客户机程序4）。

这个框架一般是合理的，可以使用它作为编写任意数量的客户机程序的基础。开发它以后，我们将暂不考虑如何处理各种问题。首先，我们将讨论如何处理特定的硬编码的 SQL 语句，然后再开发用于处理任意语句的代码。在这之后，将查询处理的代码增加到客户机程序框架中，开发另一个程序（客户机程序 5），它类似于 mysql 客户机程序。

我们也将考虑（并解决）一些通用的问题，如“如何获取有关表的结构信息？”和“如何在数据库中插入图像？”

只有在需要时，本章才讨论客户机库的函数和数据类型。要想了解所有函数和类型的列表，请参阅附录 F“C API 参考”。可以用这个附录作为使用客户机库任何部分的进一步的背景信息的参考。

样例程序可以由联机下载得到，可以直接使用，而不必再键入它们。有关的指导，请参阅附录 A“获得和安装软件”。

在哪里寻找样例

MySQL 邮件清单的一个共同问题就是“我在哪里可以找到一些用 C 写的客户机样例？”。当然，这个答案是“就在本书里！”。但是，许多人好像并没有考虑的是 MySQL 分发包中包括了若干客户机程序（例如 mysql、mysqladmin 和 mysqldump），这些大部分都是用C编写的。因为这个分发包可以很容易地以源程序形式使用，所以 MySQL 提供非常少的样例客户机代码。因此，如果您还没有这样做，找个时间找到源程序分发包，在

客户机目录中查看这些程序。MySQL 客户机程序为共享软件，从那里可以为自己的程序自由地借用代码。

在本章提供的样例和 MySQL 分发包中包括的客户机程序之间，可以找到与自己编写程序时想做的事情相类似的代码。如果是这样，可以通过拷贝和修改已有的程序来重新使用代码。应该阅读本章，了解客户机库是如何工作的。然而，请记住，并不总是需要自己编写琐碎的每件事情（您将注意到，在本章编写程序的讨论中，代码的可重用性是目的之一）。通过使用其他人编好的程序，可以避免许多工作，那是最好的。

6.1 建立客户机程序的一般过程

本节介绍使用 MySQL 客户机库编译和连接程序所包括的步骤。不同的系统建立客户机程序的命令也有所不同，可能需要稍微修改一下这里介绍的命令。然而，这里的说明是通用的，应该能够将它用于几乎您编写的任何客户机程序中。

6.1.1 基本的系统需求

当您用 C 编写 MySQL 客户程序时，显然将需要一个 C 编译程序。这里说明的样例使用 gcc。除了自己的源文件以外，还将需要下列程序：

- MySQL 头文件。
- MySQL 客户机库。

MySQL 头文件和客户机库组成客户机编程的支持程序。它们可能已经安装到您的系统上。如果没有，应获取它们。如果 MySQL 从源程序分发包或二进制分发包中安装，则客户机可编程的支持程序应该已经作为该处理的一部分安装了。如果 MySQL 是从 RPM 文件中安装的，则除非安装了开发程序 RPM，否则就没有这种支持。如果需要安装 MySQL 头文件和库，请参阅附录 A。

6.1.2 编译和连接客户机程序

要想编译和连接客户机程序，就必须指定 MySQL 头文件和客户机库的位置，因为它们通常不安装在编译程序和连接程序缺省搜索的位置。对于下面的样例，假定头文件和客户机库的位置为 /usr/local/include/mysql 和 /usr/local/lib/mysql。

要想告知编译程序如何寻找 MySQL 头文件，则当将源文件编译为目标文件时，传送给它一个 -I/usr/local/include/mysql 参数。例如，可以使用这样的命令：

```
% gcc -c -I/usr/local/include/mysql myclient.c
```

要想告知连接程序在哪，可以找到客户机库和它的名称，当连接目标文件产生一个可执行的二进制文件时，传送 -L/usr/local/lib/mysql 和 -lmysqlclient 参数，如下所示：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient
```

如果客户机程序包括多个文件，则所有目标文件的名称都要列在连接命令上。如果连接步骤导致不能找到必需的 floor() 函数的错误，则通过在命令行的后面增加 -lm，连接到数学库：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient -lm
```

可能还需要增加其他的库。例如，在 Solaris 上可能需要 -lsocket -lnsl。

如果没有使用 make 建立程序，则建议您了解一下如何进行，以便不必手工地键入许多建立程序的命令。假设有一个客户机程序 myclient，包括两个源文件 main.c 和 aux.c，及一个头文件 myclient.h。一个简单的 Makefile 建立这个程序的代码，如下所示：

```
CC = gcc
INCLUDES = -I/usr/local/include/mysql
LIBS = -L/usr/local/lib/mysql -lmysqlclient

all: myclient

main.o: main.c myclient.h
    $(CC) -c $(INCLUDES) main.c
aux.o: aux.c myclient.h
    $(CC) -c $(INCLUDES) aux.c

myclient: main.o aux.o
    $(CC) -o myclient main.o aux.o $(LIBS)

clean:
    rm -f myclient main.o aux.o
```

如果是一个需要连接到数学库的系统，则更改 LIBS 的值，并将 -lm 加到最后：

```
LIBS = -L/usr/local/lib/mysql -lmysqlclient -lm
```

如果需要其他的库，如 -lsocket 和 -lnsl，则也要将这些库加到 LIBS 中。

使用 Makefile，无论何时修改何源文件，只简单地键入“make”就可以重新建立程序代码。那比键入一句长的 gcc 命令更容易，发生错误更少。

6.2 客户机程序1——连接到服务器

我们的第一个 MySQL 客户机程序很简单：连接到服务器、断开，并退出。它本身并不是非常有用，但是必须知道如何做它，因为实际上用 MySQL 数据库做任何事情都必须与服务器连接。这是一个公用的操作，开发创建连接的代码是编写每个客户机程序都将使用的代码。除此之外，这项任务带给我们一些简单开始的事情。以后，我们可以增加这个客户机来做一些更有用的事情。

我们第一个客户机程序的源代码，客户机程序 1，包括一个单独的文件，client.c：

```
/* client1.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name    NULL /* host to connect to (default = localhost) */
#define def_user_name    NULL /* user name (default = your login name) */
#define def_password     NULL /* password (default = none) */
#define def_db_name      NULL /* database to use (default = none) */

MYSQL *conn;           /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    conn = mysql_init (NULL);
    mysql_real_connect (
        conn,           /* pointer to connection handler */

```

```

        def_host_name, /* host to connect to */
        def_user_name, /* user name */
        def_password,  /* password */
        def_db_name,   /* database to use */
        0,              /* port (use default) */
        NULL,           /* socket (use default) */
        0);             /* flags (none) */
mysql_close (conn);
exit (0);
}

```

这个源文件以包括 `stdio.h` 和 `mysql.h` 开始。MySQL 客户机可能包括其他的头文件，但是一般来说至少有两个是最基本的。

主机名称、用户名称、口令和数据库名称的缺省值固定在代码内部，使事情变得简单。以后，我们将参数化这些值，以便可以在选项文件或命令行中指定它们。

程序的 `main()` 函数创建和终止与服务器的连接。建立连接使用如下两个步骤：

1) 调用 `mysql_init()` 来获取连接处理程序。MYSQL 数据类型是一个包括连接信息的结构。这种类型的变量称为连接处理程序。当我们将 `NULL` 传递给 `mysql_init()` 时，它分配一个 MYSQL 变量，初始化它，然后返回一个指向它的指针。

2) 调用 `mysql_real_connect()` 来创建与服务器的连接。`mysql_real_connect()` 可有任意数量的参数，例如：

- 连接处理程序的指针。这不能为 `NULL`；它应该是由 `mysql_init()` 返回的值。
- 服务器主机。如果指定 `NULL` 或主机“localhost”，则客户机连接到在本地主机使用 UNIX 套接字运行的服务器上。如果指定一个主机名称或主机的 IP 地址，则客户机连接到使用 TCP/IP 连接命名的主机上。

在 Windows 上，除了使用 TCP/IP 连接而不是用 UNIX 套接字以外，这种操作是类似的（在 Windows NT 上，如果主机为 `NULL`，则在 TCP/IP 以前，先试着使用一个指定的管道来连接）。

- 用户名称和口令。如果名称为 `NULL`，则客户机库将逻辑名称发送给服务器。如果口令为 `NULL`，则不发送口令。
- 端口号和套接字文件。这些指定为 0 或 `NULL`，来告知客户机库使用它的缺省值。如果不指定端口和套接字，则根据希望连接到的主机确定这些缺省值。附录 F 中的 `mysql_real_connect()` 的描述给出有关这些的详细情况。
- 标志值。因为我们不使用任何特定的连接操作，因此它是 0。这个参数可用的选项在附录 F 中的 `mysql_real_connect()` 的项目中讨论详细情况。

要想终止这个连接，可将连接处理程序的指针传递给 `mysql_close()`。当将连接处理程序传递给 `mysql_close()` 来终止这个连接时，由 `mysql_init()` 自动分配的连接处理程序自动地释放。

要想测试客户机程序 1，可使用本章前面建立客户机程序时给出的指导来编译和连接，然后运行它：

```
% client1
```

程序连接到服务器、断开并退出。这一点都不令人兴奋，但它是一个开始。然而，它只是一个开始，因为有两个重要的缺点：

- 客户机没有错误检查，所以并不真正地知道实际上它是否在工作！
 - 连接参数（主机名称，用户名称等）在源代码内部固定。如果允许用户通过指定选项文件或命令行中的参数来解决这个问题则更好一些。
- 这些问题的处理都不困难。我们将在下面专门解决它们。

6.3 客户机程序2——增加错误检查

我们的第二个客户机程序将像第一个客户机程序一样，但是将修改它们，考虑错误出现的可能性。“将错误检查作为读者的练习”这样的项目在编程文献中相当常见，这或许是因为检查错误相当令人讨厌。但是，我赞同这种观点，即 MySQL 客户机程序应该测试错误条件并适当地进行回应。由于某种原因，返回状态值的客户机库的调用做这些事情，而且您要承担忽略它们的后果。您最终还是要试图捕获由于没有错误检查而出现在程序中的错误，这些程序的用户会对程序运行如此不规律感到奇怪。

考虑我们的程序，客户机程序 1。如何知道它是否真正连接到服务器上？可以通过查看服务器的日志，找出与运行程序时间相应的 Connect 和 Quit 事件：

```
2005-10-21:02:14      20 Connect      password@localhost on
20 Quit
```

或者，可以查看 Access denied 消息：

```
2005-10-22:11:47      20 Connect      Access denied for user: 'password@localhost'
(using password: NO)
```

这条消息表示根本没有创建连接。不幸的是，客户机程序 1 没有告诉我们出现的这些结果。实际上它不能。它不能实现任何错误检查，所以它甚至不知道自己发生了什么事。无论如何，当然不一定必须查看日志来寻找是否能连接到服务器！让我们立刻改正它。

在 MySQL 客户机库中返回值的例程基本上以下列两种方式之一表示成功或失败：

- 成功时，值的指针函数返回一个非 NULL 指针，失败时返回 NULL（在这里 NULL 的意思是“C NULL 指针”，而不是“MySQL NULL 列值”）。

迄今为止，我们使用的客户机库的例程 `mysql_init()` 和 `mysql_real_connect()` 都用返回连接处理程序的指针来表示成功，NULL 表示失败。

- 整型数值的函数一般成功返回 0，失败返回非 0。不要测试特定的非 0 值，如 -1。因为当失败时，并不保证客户机库函数返回任何特定的值。有时，您可能会看到像如下的较旧的错误地测试返回值的代码：

```
if (mysql_XXX() != 0)          /* this test is incorrect */
    fprintf(stderr, "something bad happened\n");
```

这个测试可能工作，也可能不工作。MySQL API 不将任何非 0 错误的返回指定为特定的值，而只判断它（显然地）是否为 0。这个测试应该写成下面两段之一：

```
if (mysql_XXX())              /* this test is correct */
    fprintf(stderr, "something bad happened\n");
```

或如下所示：

```
if (mysql_XXX() != 0)         /* this test is correct */
    fprintf(stderr, "something bad happened\n");
```

这两个测试是等价的。如果审核 MySQL 的源代码，则可以发现，它基本上用第一种形式测试，因为这编写起来更简短。

不是每个 API 调用都返回值。我们使用的另一个客户机例程 `mysql_close()` 就不返回值（它如何失败？失败了又如何？无论如何，都要进行连接）。

当客户机库调用失败，并且需要有关失败的详细信息时，API 中的两个调用都是有用的。`mysql_error()` 返回包括错误信息的字符串，而 `mysql_errno()` 返回数值代码。应该在错误出现以后立刻调用它们，因为如果发布另一个返回状态的 API 调用，则从 `mysql_error()` 或 `mysql_errno()` 获取的任何错误信息都将来自于后面的调用。

一般来说，程序的用户查看错误字符串比查看错误代码更有启发。如果只报告两者中的一个，则建议报告字符串。出于全面考虑，本章的这个样例报告两个值。

考虑前述的讨论，我们将编写第二个客户机程序，即客户机程序 2。它类似于客户机程序 1，但是适当地增加了错误检查代码。源文件 `client2.c` 如下所示：

```
/* client2.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name    NULL /* host to connect to (default = localhost) */
#define def_user_name    NULL /* user name (default = your login name) */
#define def_password     NULL /* password (default = none) */
#define def_db_name      NULL /* database to use (default = none) */

MYSQL *conn;             /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);
    }
    if (mysql_real_connect (
        conn,             /* pointer to connection handler */
        def_host_name,    /* host to connect to */
        def_user_name,    /* user name */
        def_password,     /* password */
        def_db_name,      /* database to use */
        0,                /* port (use default) */
        NULL,             /* socket (use default) */
        0)                /* flags (none) */
        == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
            mysql_errno (conn), mysql_error (conn));
        exit (1);
    }
    mysql_close (conn);
    exit (0);
}
```

这个错误检查的逻辑是，如果失败，则 `mysql_init()` 和 `mysql_real_connect()` 都返回 NULL。请注意，尽管这个程序检查 `mysql_init()` 返回的值，但是，如果它失败，却不调用错误报告函数。这是因为当 `mysql_init()` 失败时，不能假设连接处理程序包括任何有意义的信息。

相反, 如果 `mysql_real_connect()` 失败了, 则连接处理程序并不反映有效的连接, 但是的确包括传送给错误报告函数的错误信息 (不要将该处理程序传送给任何其他客户机例程! 因为它们一般假设是一个有效连接, 所以您的程序可能崩溃)。

编译和连接客户机程序 2, 然后试着运行它:

```
% client2
```

如果客户机程序 2 没有别输出, 则连接成功。另一方面, 可能会如下所示:

```
% client2
```

```
mysql_real_connect() failed:
```

```
Error 1045 (Access denied for user: 'paul@localhost' (Using password: NO))
```

这个输出表示没有创建连接, 并说明为什么。或者, 它还表示我们的第一个程序, 即客户机程序 1, 没有成功地连接到服务器 (毕竟客户机程序 1 使用同样的连接参数)! 而在那时我们不知道, 因为客户机程序 1 没有错误检查。而客户机程序 2 做检查, 所以当出问题时, 它可以告知我们。这就是应该始终测试 API 函数返回值的原因。

MySQL 邮件清单问题经常是与错误检查有关的。典型的问题是“当发送这个查询时, 为什么我的程序崩溃了?”或“我的程序怎么没有返回任何东西?”在许多情况下, 在查询发布以前, 有疑问的程序不检查在发布该查询前是否成功地建立了连接, 或者不检查在试着检索结果前确保服务器成功执行该查询。不要假定每个客户机库都调用成功。

本章下面的例子完成错误检查, 而且也应该这样。看起来它好像有更多的工作, 但是从长远地运行来看, 它的工作实际上是少的, 因为您化费了更少的时间来捕获错综复杂的问题。在第 7 章“Perl DBI API”和第 8 章“PHP API”中, 也使用这种检查错误的方法。

现在, 当运行客户机 2 的程序时, 假设看到拒绝访问 (Access denied) 的消息。如何改正这个问题呢? 一种可能是将主机名称、用户名称和口令的 `#define` 行更改为允许访问服务器的值。这是有好处的, 在这个意义上, 至少应该能做一个连接。但是, 这些值是程序中的固定编码。所以笔者建议不要用这种方法, 特别是对口令值。当将自己的程序编译为二进制格式时, 您可能认为口令隐藏起来了, 但是, 如果有人在程序上运行 `strings`, 则它根本隐藏不住 (更不用说明读取访问源文件的人根本不用做一点工作, 就可以获取口令)。

在“客户机程序 4——运行时获取连接参数”一节中我们将处理访问的问题。首先, 笔者想说明编写连接代码的一些其他方法。

6.4 客户机程序 3——产生连接代码模块

对于我们的第三个客户机程序, 即客户机程序 3, 通过将它封装到函数 `do_connect()` 和 `do_disconnect()` 中, 将使连接和断开代码更加模块化, 这样可以很容易地由多个客户机程序使用。这提供一种选择, 可将连接代码精确地嵌入到 `main()` 函数中。无论如何, 对在应用程序过程中套用老调的任何代码都是一个好主意。将它放在可以通过多个程序访问的函数中, 而不是在每个程序中都编写一遍。如果修正这个函数中的一个错误或对这个函数作了一些改进, 则可只更改一次, 只要重新编译就可以使用这个函数的所有程序都被修正或利用这种改进。同样, 编写一些客户机程序, 以便在它们执行过程中可以若干次地连接和断开。如果将安装和卸载方法放在连接和断开的函数中, 则编写这样一个客户机更加容易。

封装策略如下所示:

1) 将公用代码分离到一个独立的源文件 (`common.c`) 的包装函数中。

- 2) 提供一个头文件, common.h, 其中包括该公共例程的原型。
- 3) 在使用公共例程的客户机源文件中包括 common.h。
- 4) 将公共源文件编译成目标文件。
- 5) 将公共目标文件连接到您的客户机程序中。

用这些策略, 让我们构造 do_connect() 和 do_disconnect()。

do_connect() 代替对 mysql_init() 和 mysql_real_connect() 的调用, 并替换错误打印的代码。除了不传递任何连接处理程序外, 您可以像 mysql_real_connect() 一样调用它。do_connect() 分配并初始化这个处理程序, 然后, 在连接后返回一个指向它的指针。如果 do_connect() 失败, 则在打印一个错误消息以后, 返回 NULL (那就是说, 调用 do_connect() 并获取返回值 NULL 的任何程序都可以简单地退出, 而不用担心打印消息的本身)。do_disconnect() 产生一个指向连接处理程序的指针, 并调用 mysql_close()。这里是 common.c 的代码:

```
#include <stdio.h>
#include <mysql.h>
#include "common.h"

MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
    MYSQL *conn; /* pointer to connection handler */

    conn = mysql_init (NULL); /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
    if (mysql_real_connect (conn, host_name, user_name, password,
                           db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    return (conn); /* connection is established */
}

void
do_disconnect (MYSQL *conn)
{
    mysql_close (conn);
}
```

common.h 声明 common.c 中这些例程的原型:

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags);

void
do_disconnect (MYSQL *conn);
```

要想访问公共例程, 应在源文件中包括 common.h。请注意, common.c 同样包括 common.h。那就是说, 如果 common.c 中的函数定义与头文件中的声明不匹配, 则立即得到一个编译程序警告。同样, 如果更改 common.c 中的调用次序而没有相应地更改 common.h,

则当重新编译 common.c 时，编译程序将发出警告。

有人会问为什么要发明包装函数 do_disconnect(), 而它使用得还这么少。do_disconnect() 和 mysql_close() 等价。但是假设在断开连接时，都有一些要执行的额外清除。则通过调用已经完全控制的包装函数，可以修改该包装函数来做需要的事情，对于所做的任何断开的操作，这种更改统一生效。如果直接调用 mysql_close(), 则不能做到这点。

在前面，笔者声称对在多个程序中或在单个程序内部多处使用的函数中，将代码封装成模块化代码是有好处的。前面介绍一个理由，还有一些理由参见下面的两个样例。

■ **样例1** 在 MySQL3.22以前的版本中，mysql_real_connect() 调用与它现在稍微有些不同：即没有数据库名称参数。如果想利用旧的 MySQL 客户机库使用 do_connect(), 则它不能工作。然而，可以修改 do_connect(), 使它可在3.22版以前的版本上运行。这就意味着，通过修改 do_connect(), 可以增加使用它的所有程序的可移植性。如果将这些连接代码直接嵌入到每个客户机中，则必须独立地修改它们中的每一个。

要想修正 do_connect(), 使它可以处理 mysql_real_connect() 的旧格式，那么就可以使用包括当前 MySQL 版本的 MySQL_VERSION_ID 宏。更改了的 do_connect() 测试 MySQL_VERSION_ID 值，并使用 mysql_real_connect() 的正确格式：

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
    MYSQL *conn; /* pointer to connection handler */

    conn = mysql_init (NULL); /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
    #if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200 /* 3.22 and up */
    if (mysql_real_connect (conn, host_name, user_name, password,
                           db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    #else /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                           port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    if (db_name != NULL) /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            fprintf (stderr, "mysql_select_db() failed:\nError %u (%s)\n",
                     mysql_errno (conn), mysql_error (conn));
        }
    }
}
```

```

        mysql_errno (conn), mysql_error (conn));
    mysql_close (conn);
    return (NULL);
}
}
#endif
return (conn);      /* connection is established */
}

```

除了下述两点以外，do_connect() 的这个修改过的版本和前一个版本在外观上是完全一样的：

- 它不将 db_name 参数传递给 mysql_real_connect() 较早的格式，因为那个版本没有这样的参数。
 - 如果数据库名称是非 NULL 的，则 do_connect() 调用 mysql_select_db() 使指定的数据库为当前数据库（这类似于没有 db_name 参数的效果）。如果没有选择这个数据库，则 do_connect() 打印一个错误消息，关闭连接，并返回 NULL 来表示失败。
- **样例2** 该样例是在对第一个样例的 do_connect() 做更改的基础上建立的。那些更改导致对错误函数 mysql_errno() 和 mysql_error() 的三组调用。每次都将报告问题的这些代码书写出来是非常讨厌的。除此之外，错误所打印出的代码看起来不舒服，读起来也困难。而读下面这样的代码就比较容易：

```
print_error (conn, " mysql_real_connect() failed ");
```

所以，让我们在 print_error() 函数中封装错误打印。即使 conn 为 NULL，也可以编写它来做一些明智的事情。也就是说，如果 mysql_init() 调用失败，可以使用 print_error()。而且没有混合调用（一些为 fprintf()，一些为 print_error()）。

我听到一些反对意见：“为了想报告一个错误而又不必每次都调用两个错误函数，所以使代码故意编写得难以阅读，以说明封装样例更好。其实不用真的写出所有的错误打印代码：只将它编写一次，然后当再次需要时就使用拷贝和粘贴即可。”这种观点是正确的，但我持反对意见，理由如下：

- 即使使用拷贝和粘贴，用较短的代码段进行起来也更容易。
- 每当报告错误时，无论是否愿意每次调用两种函数，将所有的错误报告代码书写得很长，会产生不一致性。将错误报告的代码放在容易调用的包装函数中，就可以减少这种想法并提高编码的一致性。
- 如果决定修改错误消息的格式，则只需要在一个地方而不是整个程序中做更改，这样就要容易许多。或者，如果决定将错误消息编写到日志文件中而不是（或除此以外还）编写到 stderr 中，则只须更改 print_error()，这就更容易。这种方法可能犯更少的错误，而且再一次减少了工作量和不一致的可能性。
- 当测试程序时，如果使用调试程序，将断点放在错误报告的函数中，则当它侦测出一个错误条件时，调试程序是使程序中断的一种便利方法。

以下是错误报告函数 print_error() 的使用举例：

```

void
print_error (MYSQL *conn, char *message)
{
    fprintf (stderr, "%s\n", message);
    if (conn != NULL)
    {

```

```

        fprintf(stderr, "Error %u (%s)\n",
                mysql_errno(conn), mysql_error(conn));
    }
}

```

Print_error()在common中, 所以为它增加一个原型到common.中:

```

void
print_error (MYSQL *conn, char *message);

```

现在, 可以修改 do_connect() 来使用 print_error() :

```

MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            int port_num, char *socket_name, unsigned int flags)
{
    MYSQL *mysql; /* pointer to connection handle */

    conn = mysql_init (0); /* allocate, initialize connection handle if
                             it failed == NULL */
    {
        print_error (NULL, "mysql_init() failed (probably not of security)");
        return (NULL);
    }

    /* determine MySQL_VERSION_ID. 33 MySQL_VERSION_ID >= 32000 /* 3.22 and up
    */

    if (mysql_real_connect (conn, host_name, user_name, password,
                           db_name, port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }

    /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                           db_name, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }

    if (db_name != NULL) /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            print_error (conn, "mysql_select_db() failed");
            mysql_close (conn);
            return (NULL);
        }
    }

    /* success */
    return (conn); /* connection is established */
}

```

主源文件 client3.c 与 client2.c 一样, 但是所有嵌入的连接和断开代码都利用调用包装函数来删除和替换了。如下所示:

```

/* client3.c */

#include <stdio.h>

```

```
#include <mysql.h>
#include "common.h"

#define def_host_name NULL /* host to connect to (default = localhost) */
#define def_user_name NULL /* user name (default = your login name) */
#define def_password NULL /* password (default = none) */
#define def_port_num 0 /* use default port */
#define def_socket_name NULL /* use default socket name */
#define def_db_name NULL /* database to use (default = none) */

MYSQL *conn; /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    conn = do_connect (def_host_name, def_user_name, def_password, def_db_name,
                      def_port_num, def_socket_name, 0);

    if (conn == NULL)
        exit (1);

    /* do the real work here */

    do_disconnect (conn);
    exit (0);
}
```

6.5 客户机程序4——在运行时获取连接参数

现在我们有容易修改的防止出现错误的连接代码，我们要了解一些如何做某些比使用 NULL 连接参数更灵巧的事情，如在运行时允许用户指定一些值。

客户机程序3由于固定连接参数方面的缺陷，要想更改那些值中的任何一个，都必须编辑源文件并重新编译。这十分不方便，特别是想使程序用于其他人时。

在运行时指定连接参数的一个通用的方法是使用命令行选项。MySQL 分发包中的程序接受两种形式的连接参数，如表 6-1 所示。

表6-1 标准的 MySQL 命令行选项

参 数	短 格 式	长 格 式
主机名称	-h host_name	--host=host_name
用户名称	-u user_name	--user=user_name
口令	-p 或 -pyour_password	--password 或 --password=your_password
端口号	-P port_num	--port=port_num
套接字名称	-S socket_name	--socket=socket_name

与标准的 MySQL 客户机程序一致，客户机程序将接受同样的格式。这很容易，那是因为客户机库包括了实现选项分析的函数。

除此之外，客户机程序具有从选项文件中抽取信息的能力。这允许将连接参数放在 `-. my.cnf`（也就是主目录中的 `.my.cnf` 文件）中，以便不用在命令行中指定它们。客户机库使检查 MySQL 选项文件和从它们中抽取任何相关的值变得非常容易。只在程序中增加几行代码，就可以使选项文件识别它，并且通过编写自己的代码而不必重新改造这个框架来进行操作。附录 E “MySQL 程序参考”中说明了选项文件的语法。

6.5.1 访问选项文件内容

使用 `load_default()` 函数为连接参数值读取选项文件，`load_default()` 寻找选项文件、分析任何感兴趣的可选组的内容，以及重新编写程序的参数向量（`argv[]` 数组），以便把来自于那些组的信息以命令行选项的形式放置在 `argv[]` 的开头。这就是说，在命令行指定出现的选项。因此，当分析命令选项时，就得到了作为常规选项分析循环部分的连接参数。选项加到 `argv[]` 的开头而不是加到末尾，所以，如果连接参数真的在命令行指定，它们要比 `load_defaults()` 增加的任何选项晚一些出现（因而忽略）。

下面的小程序 `show_argv` 显示了如何使用 `load_defaults()`，并举例说明了对参数向量如何做出这样的修改：

```
/* show_argv.c */

#include <stdio.h>
#include <mysql.h>

char *groups[] = { "client", NULL };

int
main (int argc, char *argv[])
{
    int i;

    my_init ();

    printf ("Original argument vector:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    load_defaults ("my", groups, &argc, &argv);

    printf ("Modified argument vector:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    exit (0);
}
```

该处理选项文件的代码包括：

- `groups[]` 是一个字符串数组，表示所感兴趣的选项文件组。对于客户机程序，始终至少指定 “client”（[client] 组）。数组的最后一个元素必须是 `NULL`。
- `my_init()` 是 `load_defaults()` 所需的执行一些设置操作的初始化例程。
- `load_defaults()` 有四个参数：选项文件的前缀（这里应该始终是 “my”），列出感兴趣的可选组的数组、程序参数的数目和向量的地址。不传数目和向量的值，而是传地址，因为 `load_defaults()` 需要改变它们的值。特别注意的是，虽然 `argv` 是一个指针，但还是要传 `&argv`，它是指针的地址。

`show_argv` 打印参数两次，第一次是在命令行指定它们的时候，第二次是在 `load_defaults()` 修改它们的时候。为了查看 `load_defaults()` 的运行效果，应确信在主目录中有一个具有 [client] 组指定设置的 `.my.cnf` 文件。假设 `.my.cnf` 文件如下：


```
[client]
user=paul
password=secret
host=some_host
```

如果是这种情况，则执行 `show_argv` 产生的输出结果如下：

```
% show_argv a b
Original argument vector:
arg 0: show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: show_argv
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

有可能会从不在命令行或 `~/.my.cnf` 文件中的 `show_argv` 所产生的输出结果中看到一些选项。如果是这样，它们或许是在系统范围的选项文件中指定的。在主目录中读取 `.my.cnf` 之前，`load_defaults()` 实际上是在 MySQL 数据目录中寻找 `/etc/my.cnf` 和 `my.cnf` 文件（在 Windows 中，`load_defaults()` 在 Windows 系统目录中寻找文件 `C:\my.cnf`、`C:\mysql\data\my.cnf` 和 `my.ini`）。

使用 `load_defaults()` 的客户机程序几乎始终是在选项组列表中指定“client”（以便从选项文件中获取任何通用的客户机设置），但是也可以为请求自己的程序请求特定值。可将下列代码：

```
char *groups[] = { "client", NULL };
```

修改为：

```
char *groups[] = { "show_argv", "client", NULL };
```

然后将 `[show_argv]` 组添加到 `~/.my.cnf` 文件中：

```
[client]
user=paul
password=secret
host=some_host
```

```
[show_argv]
host=other_host
```

有了这些改变，再次调用 `show_argv` 就得到了一个不同的结果，如下所示：

```
% show_argv a b
Original argument vector:
arg 0: show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: show_argv
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b
```

参数数组中选项值出现的顺序取决于它们在选项文件中列出的顺序，而不是选项组在 `group[]` 数组中列出的顺序。这意味着将可能在选项文件的 `[client]` 组之后指定程序专有的组。即如果在两个组中都指定了一个选项，程序专有的值将有更高的优先权。在这个例子中可以看到：在组 `[client]` 和 `[show_argv]` 中都指定了 `host` 选项，但是因为组 `[show_argv]` 在选项文件的最后出现，所以 `host` 值将在参数向量中出现并取得优先权。

`load_defaults()` 不是从环境设置中提取值，如果想使用环境变量的值，例如 `MYSQL_TCP_PORT` 或者 `MYSQL_UNIX_PORT`，就必须使用 `getenv()` 来自管理。我不想把这个管理能力增加到客户机中，但这里有一个例子，介绍了如何检查几个标准的与 MySQL 有关的环境变量值：

```
extern char *getenv();
char *p;
int port_num;
char *socket_name;

if ((p = getenv ("MYSQL_TCP_PORT")) != NULL)
    port_num = atoi (p);
if ((p = getenv ("MYSQL_UNIX_PORT")) != NULL)
    socket_name = p;
```

在标准 MySQL 客户机中，环境变量值的优先权比在选项文件或命令行指定值的优先权要低。如果检查环境变量的值，并要与约定保持一致，那么就要在调用 `load_default()` 或者处理命令行选项之前（不是之后）检查环境。

6.5.2 分析命令行参数

现在我们可以把所有的连接参数都放入参数向量，但需要一个分析该向量的方法。`getopt_long()` 函数就是为此目的设计的。

`getopt_long()` 设在 MySQL 客户机库的内部，因此，无论什么时候与库连接都可以访问它。源文件中要包含 `getopt.h` 头文件，可以把这个头文件从 MySQL 源分发包的 `include` 目录拷贝到正在开发的客户机程序所在的目录中。

load_defaults() 与安全

因为有些程序（如 `ps`）可以显示任何过程的参数列表，`load_defaults()` 将口令的文本放在参数列表中，所以您可能对它处理窥探的含意表示惊异。这没有问题，因为 `ps` 显示原始的 `argv[]` 内容，由 `load_defaults()` 创建的任何口令参数都指向为它自己分配的区域，这个区域并不是原始区域的一部分，所以 `ps` 看不见它。

另一方面，除非故意清除，否则在命令行指定的口令会在 `ps` 中出现。6.5.2节“分析命令行参数”介绍了如何去做。

下面的程序 `show_param` 使用 `load_defaults()` 读取选项文件，然后调用 `getopt_long()` 来分析参数向量。`show_param` 举例说明了通过执行以下操作参数处理的每个阶段发生了什么：

- 1) 建立主机名称、用户名称和口令的缺省值。
- 2) 打印原始连接参数和参数向量值。
- 3) 调用 `load_defaults()` 重新编写参数向量，反映选项文件内容，然后打印结果向量。
- 4) 调用 `getopt_long()` 处理参数向量，然后打印结果参数值和参数向量中的剩余部分。

show_param 允许使用各种指定的连接参数的方法进行试验（无论是在选项文件中还是在命令行中），并通过显示使用什么值进行连接来查看结果。当实际上我们把参数处理代码与连接函数 do_connect() 连到一起时，show_param 对于预知下一个客户机程序将要发生什么是很有用的。

以下是 show_param.c 的代码：

```
/* show_param.c */

#include <stdio.h>
#include <stdlib.h> /* needed for atoi() */
#include "getopt.h"

char *groups[] = { "client", NULL };

struct option long_options[] =
{
    {"host",      required_argument, NULL, 'h'},
    {"user",      required_argument, NULL, 'u'},
    {"password",  optional_argument, NULL, 'p'},
    {"port",      required_argument, NULL, 'P'},
    {"socket",    required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

int
main (int argc, char *argv[])
{
    char *host_name = NULL;
    char *user_name = NULL;
    char *password = NULL;
    unsigned int port_num = 0;
    char *socket_name = NULL;
    int i;
    int c, option_index;
    my_init ();

    printf ("Original connection parameters:\n");
    printf ("host name: %s\n", host_name ? host_name : "(null)");
    printf ("user name: %s\n", user_name ? user_name : "(null)");
    printf ("password: %s\n", password ? password : "(null)");
    printf ("port number: %u\n", port_num);
    printf ("socket name: %s\n", socket_name ? socket_name : "(null)");

    printf ("Original argument vector:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    load_defaults ("my", groups, &argc, &argv);

    printf ("Modified argument vector after load_defaults():\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                            &option_index)) != EOF)
    {
        switch (c)
        {
```

```

case 'h':
    host_name = optarg;
    break;
case 'u':
    user_name = optarg;
    break;
case 'p':
    password = optarg;
    break;
case 'P':
    port_num = (unsigned int) atoi (optarg);
    break;
case 'S':
    socket_name = optarg;
    break;
}

argc -= optind; /* advance past the arguments that were processed */
argv += optind; /* by getopt_long() */

printf ("Connection parameters after getopt_long():\n");
printf ("host name: %s\n", host_name ? host_name : "(null)");
printf ("user name: %s\n", user_name ? user_name : "(null)");
printf ("password: %s\n", password ? password : "(null)");
printf ("port number: %u\n", port_num);
printf ("socket name: %s\n", socket_name ? socket_name : "(null)");

    printf ("Argument vector after getopt_long():\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    exit (0);
}

```

为了处理参数向量，show_argv() 使用 getopt_long()，它在循环中调用：

To process the argument vector, show_argv uses getopt_long(), which you typically call in a loop:

```

while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                        &option_index)) != EOF)
{

```

getopt_long() 的前两个参数是程序的计数参数和向量参数，第三个参数列出了要识别的选项字符。这些是程序选项的短名称形式。选项字符后可以有冒号、双冒号或者无冒号，表示选项值必须跟在选项后面、可以跟在选项后面或者不能跟在选项后面。第四个参数 long_options 是一个指向可选结构数组的指针，每个可选结构为程序需要支持的选项指定信息。它的目标与第三个参数的可选字符串相类似。每个 long_options[] 结构有四个元素，其描述如下：

■ 选项的长名称。

■ 选项值。这个值可以是 required_argument、optional_argument 或者 no_argument，表明选项值是必须跟在选项后面、可以跟在选项后面，还是不能跟在选项后面（它们与第三个参数选项字符串中的冒号、双冒号或无冒号的作用相同）。

■ 标记参数。可用它存储变量指针。如果找到这个选项，getopt_long() 则把第四个参数指定的值存储到变量中去。如果标记是 NULL，getopt_long() 就把 optarg 变量指向下一个选项的任何值，并返回选项的短名称。long_options[] 数组为所有的选项指定了

NULL。那就是说，如果遇到 `getopt_long()`，就返回每个参数，以便我们可以在 `switch` 语句中来处理它。

■ **选项的短（单个字符）名称。** 在 `long_options[]` 数组中指定的短名称必须与作为第三个参数传递给 `getopt_long()` 的选项字符串所使用的字母相匹配，否则程序将不能正确处理命令行参数。

`long_options[]` 数组必须由一个所有元素都设为 0 的结构所终止。

`getopt_long()` 的第五个参数是一个指向 `int` 变量的指针。`getopt_long()` 把与最后遇到的选项相符合的 `long_options[]` 结构索引存储到变量中（`show_param` 不用这个值做任何事情）。

请注意，口令选项（指定为 `--password` 或者 `-p`）可以获得一个选项值，那就是说，如果使用长选项形式可指定为 `--password` 或者 `--password = your_pass`，如果使用短选项形式则指定为 `-p` 或者 `-pyour_pass`。可选字符串中“`p`”后面的双冒号和 `long_options[]` 数组中的 `optional_argument` 表示了口令值的可选特性。特别是，MySQL 客户机允许在命令行省略口令值，然后提示输入。这样避免了在命令行给出口令，它防止其他人通过偷窃看到口令。在编写下一个客户机程序（客户机程序 4）时，将把口令检查性能添加进去。

下面是 `show_param` 的调用示例和结果输出（假设 `~/.my.cnf` 一直与 `show_argv` 示例有相同的内容）：

```
% show_param -h yet_another_host x
Original connection parameters:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Original argument vector:
arg 0: show_param
arg 1: -h
arg 2: yet_another_host
arg 3: x
Modified argument vector after load_defaults():
arg 0: show_param
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: x
Connection parameters after getopt_long():
host name: yet_another_host
user name: paul
password: secret
port number: 0
socket name: (null)
Argument vector after getopt_long():
arg 0: x
```

输出结果说明从命令行得到主机名（忽略选项文件中的这个值），从选项文件中得到用户名和口令。`getopt_long()` 正确分析了选项是在短选项形式（`-h host_name`）中指定还是在长选项形式（`--user = paul`，`--password = secret`）中指定。

现在让我们去掉纯粹说明选项处理例程是如何工作的这一部分，把剩余部分作为根据选项文件或命令行提供的任何选项而连接到服务器的客户机的基础。源文件 `client4.c` 的代码如下：

```
/* client4.c */

#include <stdio.h>
#include <stdlib.h> /* for atoi() */
#include <mysql.h>
#include "common.h"
#include "getopt.h"

#define def_host_name    NULL /* host to connect to (default = localhost) */
#define def_user_name    NULL /* user name (default = your login name) */
#define def_password     NULL /* password (default = none) */
#define def_port_num     0    /* use default port */
#define def_socket_name  NULL /* use default socket name */
#define def_db_name      NULL /* database to use (default = none) */

char *groups[] = { "client", NULL };

struct option long_options[] =
{
    {"host",      required_argument, NULL, 'h'},
    {"user",      required_argument, NULL, 'u'},
    {"password",  optional_argument, NULL, 'p'},
    {"port",      required_argument, NULL, 'P'},
    {"socket",    required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

MYSQL *conn; /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    char *host_name = def_host_name;
    char *user_name = def_user_name;
    char *password = def_password;
    unsigned int port_num = def_port_num;
    char *socket_name = def_socket_name;
    char *db_name = def_db_name;
    char passbuf[100];
    int ask_password = 0;
    int c, option_index=0;
    int i;

    my_init ();
    load_defaults ("my", groups, &argc, &argv);

    while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                           &option_index)) != EOF)
    {
        switch (c)
        {
            case 'h':
                host_name = optarg;
                break;
            case 'u':
                user_name = optarg;
                break;
            case 'p':
                if (!optarg) /* no value given */
                    ask_password = 1;
        }
    }
```



```

else                /* copy password, wipe out original */
{
    (void) strncpy (passbuf, optarg, sizeof(passbuf)-1);
    passbuf[sizeof(passbuf)-1] = '\0';
    password = passbuf;
    while (*optarg)
        *optarg++ = ' ';
}
break;
case 'P':
    port_num = (unsigned int) atoi (optarg);
    break;
case 'S':
    socket_name = optarg;
    break;
}
}

argc -= optind; /* advance past the arguments that were processed */
argv += optind; /* by getopt_long() */

if (argc > 0)
{
    db_name = argv[0];
    --argc; ++argv;
}

if (ask_password)
    password = get_tty_password (NULL);

conn = do_connect (host_name, user_name, password, db_name,
                  port_num, socket_name, 0);
if (conn == NULL)
    exit (1);
/* do the real work here */

do_disconnect (conn);
exit (0);
}

```

与前面开发的客户机程序 1、客户机程序 2 和客户机程序 3 比较一下，客户机程序 4 具有一些以前没有的内容：

- 允许在命令行指定数据库名称，它紧跟在由 `getopt_long()` 分析的选项的后面。这与 MySQL 分发包中标准客户机的行为是一致的。
- 对口令值做了备份之后，删除参数向量中的任何口令值。这使时间窗口最小化，在时间窗口中命令行所指定的口令对于 `ps` 或其他系统状态程序是可见的（窗口缩到最小，但并没有删除。命令行指定的口令仍然不太安全）。
- 如果给出没有值的口令选项，则客户机程序提示用户用 `get_tty_password()` 输入口令。在客户机库中，这是一个实用程序，它提示输入口令而不在显示器上回应（客户机库充满了这样吸引人的东西。因为找到了相关的例程和使用它们的方法，所以有助于从 MySQL 客户机程序的源文件中的读取）。您可能会问：“为什么不只调用 `getpass()` 呢？”回答是，并不是所有的系统都有这个函数，如 Windows。`get_tty_password()` 可以在系统间移植，因为它被配置为适应各种不同系统。

客户机程序4按照指定的选项来响应。假设没有使事件复杂化的选项文件。如果无参数调用客户机程序4，则连接到 localhost，并把 UNIX 注册名和无口令传递到服务器中。相反，如果像介绍的那样调用客户机程序4，则提示输入口令（没有直接以 -p 开头的口令值），连接到 some_host，并将用户名 some_user 和键入的口令都传递到服务器：

```
% client4 -h some_host -u some_user -p some_db
```

客户机程序4也把数据库名 some_db 传递给 do_connect()，成为当前数据库。如果没有选项文件，则处理它的内容并用来改变参数连接。

早期，我们曾热衷于封装代码，创建包装函数，目的是断开与服务器的连接和从服务器的连接断开。询问是否把分析选项部分放置到包装函数中也是合理的。我想这是可能的，但并不想去做。选项分析代码与连接代码在程序间并不一致：程序经常支持除了标准选项之外的其他选项，不同的程序很可能支持其他选项的不同设置。这就使选项处理循环标准化的函数很难编写。而且，与连接的建立不同，在它的执行过程中程序可以希望进行多次（因而是好的封装候选者），而选项分析只在程序开始时执行一次。

迄今为止，我们所做的工作完成了每个 MySQL 客户机程序所必须做的事情：用适当的参数与服务器相连接。当然应该知道如何连接，现在知道怎么做了，并且处理的细节由客户机程序框架（client4.c）来实现，因此就不必再去考虑了。这就是说可以集中精力干真正感兴趣的事情——访问数据库的内容。应用程序中所有的真正功能将在 do_connect() 调用和 do_disconnect() 调用之间发生，但是我们现在所拥有的是用于建立可为许多不同客户机程序使用的基本框架。编写一个新程序，要做到以下几点：

- 1) 制作一个 client4.c 的备份。
- 2) 如果接受其他选项而不是 client4.c 支持的标准选项，那么修改处理选项循环。
- 3) 在连接和断开调用之间加上自己的应用程序代码。

这样就算完成了。

构造客户机程序框架的目的是，很容易地建立和断开连接，以便集中精力干真正想做的事情。

6.6 处理查询

我们已经知道了如何开始和结束与服务器的会话，现在应该看看如何控制会话。本节介绍了如何与服务器通信以处理查询。

执行的每个查询应包括以下步骤：

- 1) **构造查询**。查询的构造取决于查询的内容——特别要看是否含有二进制数据。
- 2) **通过将查询发送到服务器执行来发布查询**。
- 3) **处理查询结果**。这取决于发布查询的类型。例如，SELECT 语句返回数据行等待处理，INSERT 语句就不这样。

构造查询的一个要素就是使用哪个函数将查询发送到服务器。较通用的发布查询例程是 mysql_real_query()。该例程给查询提供了一个计数串（字符串加上长度）。必须了解查询串的长度，并将它们连同串本身一起传递给 mysql_real_query()。因为查询是一个计数的字符串，所以它的内容可能是任何东西，其中包括二进制数据或者空字节。查询不能是空终结串。

另一个发布查询的函数，mysql_query()，在查询字符串允许的内容上有更多的限制，但

更容易使用一些。传递到 `mysql_query()` 的查询应该是空终结串，这说明查询内部不能含有空字节（查询里含有空字节会导致错误地中断，这比实际的查询内容要短）。一般说来，如果查询包含任意的二进制数据，就可能包含空字节，因此不要使用 `mysql_query()`。另一方面，当处理空终结串时，使用熟悉的标准 C 库字符串函数构造查询是很耗费资源的，例如 `strcpy()` 和 `sprintf()`。

构造查询的另一个要素就是是否要执行溢出字符的操作。如果在构造查询时使用含有二进制数据或者其他复杂字符的值时，如引号、反斜线等，就需要使用这个操作。这些将在 6.8.2 节“对查询中有疑问的数据进行编码”中讨论。

下面是处理查询的简单轮廓：

```
if (mysql_query (conn, query) != 0)
{
    /* failure; report error */
}
else
{
    /* success; find out what effect the query had */
}
```

`mysql_query()` 和 `mysql_real_query()` 的查询成功都会返回零值，查询失败返回非零值。查询成功指服务器认为该查询有效并接受，而且能够执行，并不是指有关该查询结果。例如，它不是指 `SELECT` 查询所选择的行，或 `DELETE` 语句所删除的行。检查查询的实际结果要包括其他的处理。

查询失败可能有多种原因，有一些常见的原因如下：

- 含有语法错误。
- 语义上是非法的——例如涉及对表中不存在的列的查询。
- 没有足够的权利访问查询所引用的数据。

查询可以分成两大类：不返回结果的查询和返回结果的查询。`INSERT`、`DELETE` 和 `UPDATE` 等语句属于“不返回结果”类的查询，即使对修改数据库的查询，它们也不返回任何行。可返回的唯一信息就是有关受作用的行数。

`SELECT` 语句和 `SHOW` 语句属于“返回结果”类的查询；发布这些语句的目的就是要返回某些信息。返回数据的查询所生成的行集合称为结果集，在 MySQL 中表示为 `MYSQL_RES` 数据类型，这是一个包含行的数据值及有关这些值的元数据（如列名和数据值的长度）的结构。空的结果集（就是包含零行的结果）要与“没有结果”区分开。

6.6.1 处理不返回结果集的查询

处理不返回结果集的查询，用 `mysql_query()` 或 `mysql_real_query()` 发布查询。如果查询成功，可以通过调用 `mysql_affected_rows()` 找出有多少行需要插入、删除或修改。

下面的样例说明如何处理不返回结果集的查询：

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
    print_error ("INSERT statement failed");
}
else
{
    ...
}
```

```
printf ("INSERT statement succeeded: %lu rows affected\n",  
        (unsigned long) mysql_affected_rows (conn));  
}
```

请注意在打印时 `mysql_affected_rows()` 的结果是如何转换为 `unsigned long` 类型的，这个函数返回一个 `my_ulonglong` 类型的值，但在一些系统上无法直接打印这个类型的值（例如，笔者观察到它可在 FreeBSD 下工作，但不能在 Solaris 下工作）。把值转换为 `unsigned long` 类型并使用 ‘%lu’ 打印格式可以解决这个问题。同样也要考虑返回 `my_ulonglong` 值的其他函数，如 `mysql_num_rows()` 和 `mysql_insert_id()`。如果想使客户机程序能跨系统地移植，就要谨记这一点。

`mysql_rows_affected()` 返回查询所作用的行数，但是“受作用的行”的含义取决于查询的类型。对于 `INSERT`、`DELETE` 和 `UPDATE`，是指插入、删除或者更新的行数，也就是 MySQL 实际修改的行数。如果行的内容与所要更新的内容相同，则 MySQL 就不再更新行。这就是说虽然可能选择行来更新（通过 `UPDATE` 语句的 `WHERE` 子句），但实际上该行可能并未改变。

对于 `UPDATE`，“受作用的行”的意义实际上是个争论点，因为人们想把它当成“被匹配的行”——即选择要更新的行数，即使更新操作实际上并未改变其中的值也是如此。如果应用程序需要这个信息，则当与服务器连接时可以用它来请求以实现这个功能。将 `CLIENT_FOUND_ROWS` 的 `flags` 值传递给 `mysql_real_connect()`。也可以将 `CLIENT_FOUND_ROWS` 作为 `flags` 参数传递给 `do_connect()`；它将把值传递给 `mysql_real_connect()`。

6.6.2 处理返回结果集的查询

通过调用 `mysql_query()` 和 `mysql_real_query()` 发布查询之后，返回数据的查询以结果集形式进行。在 MySQL 中实现它非常重要，`SELECT` 不是返回行的唯一语句，`SHOW`、`DESCRIBE` 和 `EXPLAIN` 都需要返回行。对所有这些语句，都必须在发布查询后执行另外的处理行操作。

处理结果集包括下面几个步骤：

- 通过调用 `mysql_store_result()` 或 `mysql_use_result()` 产生结果集。这些函数如果成功则返回 `MYSQL_RES` 指针，失败则返回 `NULL`。稍后我们将查看 `mysql_store_result()` 与 `mysql_use_result()` 的不同，以及选择其中一个而不选另一个时的情况。我们的样例使用 `mysql_store_result()`，它能立即从服务器返回行，并将它们存储到客户机中。
- 对结果集的每一行调用 `mysql_fetch_rows()`。这个函数返回 `MYSQL_ROW` 值，它是一个指向字符串数组的指针，字符串数组表示行中每列的值。要根据应用程序对行进行操作。可以只打印出列值，执行有关的统计计算，或者做些其他操作。当结果集中不再有行时，`mysql_fetch_rows()` 返回 `NULL`。
- 处理结果集时，调用 `mysql_free_result()` 释放所使用的内存。如果忽略了这一点，则应用程序就会泄露出内存（对于长期运行的应用程序，适当地解决结果集是极其重要的；否则，会注意到系统将由一些过程所取代，这些过程消耗着经常增长的系统资源量）。

下面的样例轮廓介绍了如何处理返回结果集的查询：

```

MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM mysql") != 0)
    print_error (conn, "mysql_query() failed");
else
{
    res_set = mysql_store_result (conn);    /* generate result set */
    if (res_set == NULL)
        print_error (conn, "mysql_store_result() failed");
    else
    {
        /* process result set, then deallocate it */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}

```

我们通过调用函数 `process_result_set()` 来处理每一行，这里有个窍门，因为我们并没有定义这个函数，所以需要这样做。通常，结果的处理集函数是基于下面的循环：

```

MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* do something with row contents */
}

```

从 `mysql_fetch_row()` 返回的 `MYSQL_ROW` 值是一个指向数值数组的指针，因此，访问每个值就是访问 `row[i]`，这里 `i` 的范围是从0到该行的列数减1。

这里有几个关于 `MYSQL_ROW` 数据类型的关键点需要注意：

- `MYSQL_ROW` 是一个指针类型，因此，必须声明类型变量为 `MYSQL_ROW row`，而不是 `MYSQL_ROW *row`。
- `MYSQL_ROW` 数组中的字符串是空终结的。但是，列可能含有二进制数据，这样，数据中就可能含有空字节，因此，不应该把值看成是空终结的。由列的长度可知列值有多长。
- 所有数据类型的值都是作为字符串返回的，即使是数字型的也是如此。如果需要该值为数字型，就必须自己对该字符串进行转换。
- 在 `MYSQL_ROW` 数组中，`NULL` 指针代表 `NULL`，除非声明列为 `NOT NULL`，否则应该经常检查列值是否为 `NULL` 指针。

应用程序可以利用每行的内容做任何想做的事，为了举例说明这一点，我们只打印由制表符隔开列值的行，为此还需要另外一个函数，`mysql_num_fields()`，它来自于客户机库；这个函数告知我们该行包括多少个值（列）。

下面就是 `process_result_set()` 的代码：

```

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
    MYSQL_ROW    row;
    unsigned int  i;

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {

```

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i] != NULL ? row[i] : "NULL");

    }
    fputc ('\n', stdout);
}
if (mysql_errno (conn) != 0)
    print_error (conn, "mysql_fetch_row() failed");
else
    printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}
```

process_result_set() 以制表符分隔的形式打印每一行（将 NULL 值显示为单词“NULL”），它跟在 被检索的行计数的后面，该计数通过调用 mysql_num_rows() 来计算。像 mysql_affected_rows() 一样，mysql_num_rows() 返回 my_ulonglong 值，因此，将值转换为 unsigned long 型，并用 ‘%lu’ 格式打印。

提取行的循环紧接在一个错误检验的后面，如果要用 mysql_store_result() 创建结果集，mysql_fetch_row() 返回的 NULL 值通常意味着“不再有行”。然而，如果用 mysql_use_result() 创建结果集，则 mysql_fetch_row() 返回的 NULL 值通常意味着“不再有行”或者发生了错误。无论怎样创建结果集，这个测试只允许 process_result_set() 检测错误。

process_result_set() 的这个版本是打印列值要求条件最低的方法，每种方法都有一定的缺点，例如假设执行下面的查询：

```
SELECT last_name, first_name, city, state FROM president
```

会得到下面的输出：

```
Adams John Braintree MA
Adams John Quincy Braintree MA
Arthur Chester A. Fairfield VT
Buchanan James Mercersburg PA
Bush George W. Milton MA
Carter James E. Jr Plains GA
Cleveland Grover Caldwell NJ
```

我们可以通过提供一些信息如列标签，及通过使这些值垂直排列，而使输出结果漂亮一点。为此，我们需要标签和每列所需的最宽的值。这个信息是有效的，但不是列数据值的一部分，而是结果集的元数据的一部分（有关数据的数据）。简单归纳了一下查询处理程序后，我们将在 6.6.6 节“使用结果集元数据”中给出较漂亮的显示格式。

打印二进制数据

对包含可能含有空字节的二进制数据的列值，使用 ‘%s’ printf() 格式标识符不能将它正确地打印；printf() 希望一个空终结串，并且直到第一个空字节才打印列值。对于二进制数据，最好用列的长度，以便打印完整的值，如可以用 fwrite() 或 putc()。

6.6.3 通用目标查询处理程序

前面介绍的处理查询样例应用了语句是否应该返回一些数据的知识来编写的。这是可能

的，因为查询固定在代码内部：使用 INSERT 语句时，它不返回结果，使用 SHOW TABLES 语句时，才返回结果。

然而，不可能始终知道查询用的是哪一种语句，例如，如果执行一个从键盘键入或来源于文件的查询，则它可能是任何的语句。不可能提前知道它是否会返回行。当然不想对查询做语法分析来决定它是哪类语句，总之，并不像看上去那样简单。只看第一个单词是不够的，因为查询也可能以注释语句开始，例如：

```
/* comment */ SELECT ...
```

幸运的是不必过早地知道查询类型就能够正确地处理它。用 MySQL C API 可编写一个能很好地处理任何类型语句的通用目标查询处理程序，无论它是否会返回结果。

在编写查询处理程序的代码之前，让我们简述一下它是如何工作的：

- 发布查询，如果失败，则结束。
- 如果查询成功，调用 mysql_store_result() 从服务器检索行，并创建结果集。
- 如果 mysql_store_result() 失败，则查询不返回结果集，或者在检索这个结果集时发生错误。可以通过把连接处理程序传递到 mysql_field_count() 中，并检测其值来区别这两种情况，如下：

- 如果 mysql_field_count() 非零，说明有错误，因为查询应该返回结果集，但却没有。这种情况发生有多种原因。例如：结果集可能太大，内存分配失败，或者在提取行时客户机和服务器之间发生网络中断。

这种过程稍微有点复杂之处就在于，MySQL 3.22.24 之前的早期版本中不存在 mysql_field_count()，它们使用的是 mysql_num_fields()。为编写 MySQL 任何版本都能运行的程序，在调用 mysql_field_count() 的文件中都包含下面的代码块：

```
#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID < 32224
#define mysql_field_count mysql_num_fields
#endif
```

这就将对 mysql_field_count() 的一些调用看作是比 MySQL 3.22.24 更早版本中的 mysql_num_fields() 的调用。

- 如果 mysql_field_count() 返回0，就意味着查询不返回结果（这说明查询是类似于 INSERT、DELETE、或 UPDATE 的语句）。
- 如果 mysql_store_result() 成功，查询返回一个结果集，通过调用 mysql_fetch_row() 来处理行，直到它返回 NULL 为止。

下面的列表说明了处理任意查询的函数，给出了连接处理程序和空终结查询字符串：

```
#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID < 32224
#define mysql_field_count mysql_num_fields
#endif
```

```
void
process_query (MYSQL *conn, char *query)
{
    MYSQL_RES *res_set;
    unsigned int field_count;
```

```
    if (mysql_query (conn, query) != 0) /* the query failed */
    {
        print_error (conn, "process_query() failed");
```

```
    return;
}

/* the query succeeded; determine whether or not it returns data */

res_set = mysql_store_result (conn);
if (res_set == NULL) /* no result set was returned */
{
    /*
     * does the lack of a result set mean that an error
     * occurred or that no result set was returned?
     */
    if (mysql_field_count (conn) > 0)
    {
        /*
         * a result set was expected, but mysql_store_result()
         * did not return one; this means an error occurred
         */
        print_error (conn, "Problem processing result set");
    }
    else
    {
        /*
         * no result set was returned; query returned no data
         * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
         * so just report number of rows affected by query
         */
        printf ("%lu rows affected\n",
                (unsigned long) mysql_affected_rows (conn));
    }
}
else /* a result set was returned */
{
    /* process rows, then free the result set */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
}
}
```

6.6.4 可选择的查询处理方法

process_query() 的这个版本有三个特性：

- 用 mysql_query() 发布查询。
 - 用 mysql_store_query() 检索结果集。
 - 没有得到结果集时，用 mysql_field_count() 把错误事件和不需要的结果集区别开来。
- 针对查询处理的这些特点，有如下三种方法：
- 可以用计数查询字符串和 mysql_real_query()，而不使用空终结查询字符串和 mysql_query()。
 - 可以通过调用 mysql_use_result() 而不是调用 mysql_store_result() 来创建结果集。
 - 可以调用 mysql_error() 而不是调用 mysql_field_count() 来确定结果集是检索失败还是仅仅没有设置检索。

可用以上部分或全部方法代替 process_query()。以下是一个 process_real_query() 函数，它与 process_query() 类似，但使用了所有三种方法：

```

void
process_real_query (MYSQL *conn, char *query, unsigned int len)
{
    MYSQL_RES *res_set;
    unsigned int field_count;

    if (mysql_real_query (conn, query, len) != 0) /* the query failed */
    {
        print_error (conn, "process_real_query () failed");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_use_result (conn);
    if (res_set == NULL) /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that an error
         * occurred or that no result set was returned?
         */
        if (mysql_errno (conn) != 0) /* an error occurred */
            print_error (conn, "Problem processing result set");
        else
        {
            /*
             * no result set was returned; query returned no data
             * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
             * so just report number of rows affected by query
             */
            printf ("%lu rows affected\n",
                    (unsigned long) mysql_affected_rows (conn));
        }
    }
    else /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}

```

6.6.5 mysql_store_result() 与 mysql_use_result() 的比较

函数 `mysql_store_result()` 与 `mysql_use_result()` 类似，它们都有连接处理程序参数，并返回结果集。但实际上两者间的区别还是很大的。两个函数之间首要的区别在于从服务器上检索结果集的行。当调用时，`mysql_store_result()` 立即检索所有的行，而 `mysql_use_result()` 启动查询，但实际上并未获取任何行，`mysql_store_result()` 假设随后会调用 `mysql_fetch_row()` 检索记录。这些行检索的不同方法引起两者在其他方面的不同。本节加以比较，以便了解如何选择最适合应用程序的方法。

当 `mysql_store_result()` 从服务器上检索结果集时，就提取了行，并为之分配内存，存储到客户机中，随后调用 `mysql_fetch_row()` 就再也不会返回错误，因为它仅仅是把行脱离了已经保留结果集的数据结构。`mysql_fetch_row()` 返回 `NULL` 始终表示已经到达结果集的末端。

相反，`mysql_use_result()` 本身不检索任何行，而只是启动一个逐行的检索，就是说必须

对每行调用 `mysql_fetch_row()` 来自己完成。既然如此, 虽然正常情况下, `mysql_fetch_row()` 返回 `NULL` 仍然表示此时已到达结果集的末端, 但也可能表示在与服务器通信时发生错误。可通过调用 `mysql_errno()` 和 `mysql_error()` 将两者区分开来。

与 `mysql_use_result()` 相比, `mysql_store_result()` 有着较高的内存和处理需求, 因为是在客户机上维护整个结果集, 所以内存分配和创建数据结构的耗费是非常巨大的, 要冒着溢出内存的危险来检索大型结果集, 如果想一次检索多个行, 可用 `mysql_use_result()`。

`mysql_use_result()` 有着较低的内存需求, 因为只需给每次处理的单行分配足够的空间。这样速度就较快, 因为不必为结果集建立复杂的数据结构。另一方面, `mysql_use_result()` 把较大的负载加到了服务器上, 它必须保留结果集中的行, 直到客户机看起来适合检索所有的行。这就使某些类型的客户机程序不适用 `mysql_use_result()` :

- 在用户的请求下提前逐行进行的交互式客户机程序 (不必仅仅因为用户需要喝杯咖啡而让服务器等待发送下一行)。
- 在行检索之间做了许多处理的客户机程序。

在所有这些情况下, 客户机程序都不能很快检索结果集的所有行, 它限制了服务器, 并对其他客户机程序产生负面的影响, 因为检索数据的表在查询过程中是读锁定的。要更新表的客户机或要插入行的任何客户机程序都被阻塞。

偏移由 `mysql_store_result()` 引起的额外内存需求对一次访问整个结果集带来相当的好处。结果集中的所有行都是有效的, 因此, 可以任意访问: `mysql_data_seek()`、`mysql_rowseek()` 和 `mysql_row_tell()` 函数允许以任意次序访问行。而 `mysql_use_result()` 只能以 `mysql_fetch_row()` 检索的顺序访问行。如果想要以任意次序而不是从服务器返回的次序来处理行, 就必须使用 `mysql_store_result()`。例如, 如果允许用户来回地浏览查询所选的行, 最好使用 `mysql_store_result()`。

使用 `mysql_store_result()` 可以获得在使用 `mysql_use_result()` 时无效的某些类型的列信息。通过调用 `mysql_num_rows()` 来获得结果集的行数, 每列中的这些值的最大宽度值存储在 `MYSQL_FIELD` 列信息结构的 `max_width` 成员中。使用 `mysql_use_result()`, 直到提取完所有的行, `mysql_num_rows()` 才会返回正确值, 而且 `max_width` 无效, 因为只有在每行的数据都显示后才能计算。

由于 `mysql_use_result()` 比 `mysql_store_result()` 执行更少的操作, 所以 `mysql_use_result()` 就强加了一个 `mysql_store_result()` 没有的需求: 即客户机对结果集中的每一行都必须调用 `mysql_fetch_row()`, 否则, 结果集中剩余的记录就会成为下一个查询结果集中的一部分, 并且发生“不同步”的错误。这种情形在使用 `mysql_store_result()` 时不会发生, 因为当函数返回时, 所有的行就已被获取。事实上, 使用 `mysql_store_result()` 就不必再自己调用 `mysql_fetch_row()`。对于所有感兴趣的事情就是是否得到一个非空的结果, 而不是结果所包含的内容的查询来说, 它是很有用的。例如, 要知道表 `my_tbl` 是否存在, 可以执行下面的查询:

```
SHOW TABLES LIKE "my_tbl"
```

如果在调用 `mysql_store_result()` 之后, `mysql_num_rows()` 的值为非零, 这个表就存在, 就不必再调用 `mysql_fetch_row()` (当然仍需调用 `mysql_free_result()`)。

如果要提供最大的灵活性, 就给用户选择使用任一结果集处理方法的选项。 `mysql` 和

mysqldump 是执行这个操作的两个程序，缺省时，使用 `mysql_store_result()`，但是如果指定 `--quick` 选项，则使用 `mysql_use_result()`。

6.6.6 使用结果集元数据

结果集不仅包括数据行的列值，而且还包括数据信息，这些信息成为元数据结果集，包括：

- 结果集中的行数和列数，通过调用 `mysql_num_rows()` 和 `mysql_num_fields()` 实现。
- 行中每列值的长度，通过调用 `mysql_fetch_lengths()` 实现。
- 有关每列的信息，例如列名和类型、每列值的最大宽度和列来源的表等。
`MYSQL_FIELD` 结构存储这些信息，通过调用 `mysql_fetch_fields()` 来获得它。附录 F 详细地描述了 `MYSQL_FIELD` 结构，并列出了提供访问列信息的所有函数。

元数据的有效性部分决定于结果集的处理方法，如在上节中提到的，如果要使用行计数或者列长度的最大值，就必须用 `mysql_store_result()` 而不是 `mysql_use_result()` 创建结果集。

结果集元数据对确定有关如何处理结果集非常有帮助：

- 列名和宽度信息对漂亮地生成带有列标题并垂直排列的格式化输出是非常有用的。
- 使用列计数来确定处理数据行的连续列值的循环所迭代的次数。如果要分配取决于结果集中已知的行数或列数的数据结构，就可以使用行或列计数。
- 可以确定列的数据类型。可以看出列是否是数字的，是否可能包括二进制数据等等。

在前面的6.6.1节“处理返回结果集的查询”中，我们编写了从结果集的行中以制表符分隔的形式打印出结果的 `process_result_set()` 程序。这对某些目的是很好的（例如要把数据输入到电子制表软件中），但对于可视化检查或打印输出，就不是一个漂亮的显示格式。回忆前面的 `process_result_set()` 版本，产生过这样的输出：

```
Adams John Braintree MA
Adams John Quincy Braintree MA
Arthur Chester A. Fairfield VT
Buchanan James Mercersburg PA
Bush George W. Milton MA
Carter James E. Jr Plains GA
Cleveland Grover Caldwell NJ
...
```

让我们在每列加上标题和边框来对 `process_result_set()` 做些修改，以生成表格式的输。这种修正版看上去更美观，输出的结果是相同的，如下所示：

last_name	first_name	city	state
Adams	John	Braintree	MA
Adams	John Quincy	Braintree	MA
Arthur	Chester A.	Fairfield	VT
Buchanan	James	Mercersburg	PA
Bush	George W.	Milton	MA
Carter	James E., Jr.	Plains	GA
Cleveland	Grover	Caldwell	NJ
...	...	...	...

显示算法的基本要点是这样的：

- 1) 确定每列的显示宽度。
- 2) 打印一列带有边框的列标题（由垂直竖线和前后的虚线分隔）。
- 3) 打印结果集每行的值、带边框的列（由垂直竖线分隔），并垂直排列，除此之外，打印正确的数字，将 NULL 值打印为单词“NULL”。

- 4) 最后，打印检索的行的计数。

该练习为结果集元数据的使用提供了一个很好的示范。为了显示所描述的输出，除了行所包含的数据值之外，我们还需了解许多有关结果集的内容。

您可能想，“这个描述听起来与 mysql 显示的输出惊人地相似”。是的，欢迎把 mysql 源代码和修正版的 `process_result_set()` 代码比较一下，它们是不同的，可以发现对同一问题使用两种方法是有指导作用的。

首先，我们需要确定每列的显示宽度，下面列出如何做这件事情。可观察到这些计算完全基于结果集元数据，无论行值是什么，它们都没有引用：

```
MYSQL_FIELD    *field;
unsigned int    i, col_len;

/* determine column display widths */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
        col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
        col_len = 4;    /* 4 = length of the word "NULL" */
    field->max_length = col_len;    /* reset column info */
}
```

列宽度通过结果集中列的 `MYSQL_FIELD` 结构的迭代来计算，调用 `mysql_fetch_seek()` 定位第一个结构，后续的 `mysql_fetch_field()` 调用返回指向连续列的结构指针。显示出来的列宽度是下面三个值中的最大值，其中每一个都取决于列信息结构中的元数据：

- `field->name` 的长度，也就是列标题的长度。
- `field->max_length`，列中最长的数据值的长度。
- 如果列中可能包括 NULL 值，则为字符串“NULL”的长度，`field->flag` 表明列是否包含 NULL。

请注意，已知要显示的列的宽度后，我们将这个值赋给 `max_length`，`max_length` 是从客户机库获取的结构中的一个成员。这种获取是允许的吗？或者 `MYSQL_FIELD` 结构的内容应该为只读？一般来说，是“只读的”，但是 MySQL 分发包中的一些客户机程序以同样的方式改变了 `max_length` 的值，因此，假设这也是正确的（如果更喜欢不改变 `max_length` 值的方法，则分配一个 `unsigned int` 值的数组，将计算的宽度存储到这个数组中）。

显示宽度的计算包括一个说明，回想当使用 `mysql_use_result()` 创建结果集时，`max_length` 没有意义。因为我们需要 `max_length` 来确定列值的显示宽度，所以该算法的正确操作需要使用 `mysql_store_result()` 产生的结果集（`MYSQL_FIELD` 结构的 `length` 成员告知列值可以取得的最大值，如果使用 `mysql_store_result()` 而不是 `mysql_use_result()` 的话，这可能是个工作环境）。

一旦知道了列的宽度，就可以准备打印，处理标题很容易；对于给定的列，只需使用由 field 指向的列信息结构，用已计算过的宽度打印出 name 成员。

```
printf (" %-*s |", field->max_length, field->name);
```

对于数据，我们对结果集中的行进行循环，在每次迭代时打印当前行的列值。从行中打印列值有些技巧，因为值可能是 NULL，也可能代表一个数（无论哪种情况都如实打印）。列值的打印如下，这里 row[i] 包括数据值和指向列信息的 field 指针：

```
if (row[i] == NULL)
    printf (" %-*s |", field->max_length, "NULL");
else if (IS_NUM (field->type))
    printf (" %-*s |", field->max_length, row[i]);
else
    printf (" %-*s |", field->max_length, row[i]);
```

如果 field->type 指明的列类型是数字型，如 INT、FLOAT 或者 DECIMAL，那么宏 IS_NUM 的值为真。

显示该结果集的最终的代码如下所示。注意，因为我们需要多次打印虚线，所以这段代码封装在它自己的函数中，函数 print_dashes() 是这样的：

```
void
print_dashes (MYSQL_RES *res_set)
{
    MYSQL_FIELD    *field;
    unsigned int    i, j;

    mysql_field_seek (res_set, 0);
    fputc ('+', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        for (j = 0; j < field->max_length + 2; j++)
            fputc ('-', stdout);
        fputc ('+', stdout);
    }
    fputc ('\n', stdout);
}

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
    MYSQL_FIELD    *field;
    MYSQL_ROW      row;
    unsigned int    i, col_len;

    /* determine column display widths */
    mysql_field_seek (res_set, 0);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        col_len = strlen (field->name);
        if (col_len < field->max_length)
            col_len = field->max_length;
        if (col_len < 4 && !IS_NOT_NULL (field->flags))
            col_len = 4; /* 4 = length of the word "NULL" */
        field->max_length = col_len; /* reset column info */
    }

    print_dashes (res_set);
    fputc ('|', stdout);
    mysql_field_seek (res_set, 0);
```

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    printf (" %-*s |", field->max_length, field->name);
}
fputc ('\n', stdout);
print_dashes (res_set);

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    mysql_field_seek (res_set, 0);
    fputc ('|', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        if (row[i] == NULL)
            printf (" %-*s |", field->max_length, "NULL");
        else if (IS_NUM (field->type))
            printf (" %-*s |", field->max_length, row[i]);
        else
            printf (" %-*s |", field->max_length, row[i]);
    }
    fputc ('\n', stdout);
}
print_dashes (res_set);
printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}
```

MySQL 客户机库提供了访问列信息结构的几种方法，例如，前面样例的代码多次使用如下形式的循环访问这些结构：

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    ...
}
```

然而，mysql_field_seek() 与 mysql_fetch_field() 的结合是获得 MYSQL_FIELD 结构的唯一途径，可在附录 F 中查看 mysql_fetch_field() 函数和 mysql_fetch_field_direct() 函数，寻找其他获得列信息结构的方法。

6.7 客户机程序5——交互式查询程序

让我们把迄今为止研究的诸多内容整理一下，编写一个简单的交互式客户机程序。它的功能包括可以进入查询，用通用目标查询处理程序 process_query 执行查询，并用前面研究过的显示格式 process_result_set() 显示查询结果。

客户机程序5在某些方面与 mysql 类似，虽然在几个特征上还是有所不同。客户机程序 5 在输入上有几个约束条件：

- 每个输入行必须包括一个完整的查询。
- 查询不会以分号或 ‘\g’ 为终止。
- 不识别类似 quit 的命令；而是用 Control-D 结束程序。

客户机程序 5 的编写几乎是完全微不足道的（不到 10 行的新代码）。客户机程序框架 (client4.c) 和写过的其他代码几乎提供了所需的每一件事，我们唯一要增加的是搜集输入行

并执行它们的循环。

为了建造客户机程序5, 首先把客户机程序框架 client4.c 拷贝到 client5.c 中, 然后把代码增加到 process_query()、process_result_set() 和 print_dashes() 中, 最后在 client5.c 的 main() 中寻找标有下列字符的行:

```
/* do the real work here */
```

然后用下面的循环替换它:

```
while (1)
{
    char    buf[1024];
    fprintf (stderr, "query> ");           /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL) /* read query */
        break;
    process_query (conn, buf);             /* execute query */
}
```

编译 client5.c 产生 client5.o, 将 client5.o 与 common.o 和客户机库连接, 生成客户机程序5, 到此就全部完成了。您就拥有了一个可执行任意查询并显示结果的交互式 MySQL 客户机程序。

6.8 其他主题

本节包括几个主题, 这些主题不完全适合于本章从 client1 到 client5 的开发中的任一小节的内容:

- 在使用结果集元数据帮助验证这些数据适合于计算之后, 使用结果集数据计算结果。
- 如何处理很难插入到查询中的数据。
- 如何处理图形数据。
- 如何获得表结构的信息。
- 常见的 MySQL 程序设计错误及如何避免。

6.8.1 在结果集上执行计算

迄今为止, 我们集中而主要地使用了结果集元数据来打印行数据, 但很明显, 除打印之外, 还需要使用数据做其他事情的时候。例如, 计算基于数据值的统计信息, 应用元数据确保数据适合它们要满足的需求。哪种类型的需求? 对于启动程序来说, 可能要校验一下正要执行数字计算的列实际上是否包含着数字!

下面的列表显示了一个简单函数 summary_stats(), 它获取结果集和列索引, 并产生列值的汇总统计。该函数还列出缺少数值的数量, 它是通过检查 NULL 来检测的。这些计算包括两个数据所必须满足的需求, summary_stats() 用结果集元数据来校验:

- 指定的列必须存在 (也就是说, 列索引必须在结果集列值的范围内)。
- 此列必须包括数字值。

如果这些条件不满足, 则 summary_stats() 只打印出错误消息并返回。代码如下:

```
void
summary_stats (MYSQL_RES *res_set, unsigned int col_num)
{
    MYSQL_FIELD *field;
    MYSQL_ROW row;
```

```

unsigned int    n, missing;
double  val, sum, sum_squares, var;

/* verify data requirements */
if (mysql_num_fields (res_set) < col_num)
{
    print_error (NULL, "illegal column number");
    return;
}
mysql_field_seek (res_set, 0);
field = mysql_fetch_field (res_set);
if (!IS_NUM (field->type))
{
    print_error (NULL, "column is not numeric");
    return;
}

/* calculate summary statistics */

n = 0;
missing = 0;
sum = 0;
sum_squares = 0;

mysql_data_seek (res_set, 0);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    if (row[col_num] == NULL)
        missing++;
    else
    {
        n++;
        val = atof (row[col_num]); /* convert string to number */
        sum += val;
        sum_squares += val * val;
    }
}
if (n == 0)
    printf ("No observations\n");
else
{
    printf ("Number of observations: %lu\n", n);
    printf ("Missing observations: %lu\n", missing);
    printf ("Sum: %g\n", sum);
    printf ("Mean: %g\n", sum / n);
    printf ("Sum of squares: %g\n", sum_squares);
    var = ((n * sum_squares) - (sum * sum)) / (n * (n - 1));
    printf ("Variance: %g\n", var);
    printf ("Standard deviation: %g\n", sqrt (var));
}
}

```

请注意在 `mysql_fetch_row()` 循环前面调用的 `mysql_data_seek()`。为获得同样的结果集，它允许多次调用 `summary_stats()`（假设要计算几列的统计值的话）。每次调用 `summary_stats()`，都要“重新回到”到结果集的开始（这里假设用 `mysql_store_result()` 创建结果集，如果用 `mysql_use_result()` 创建结果集就只能按顺序处理行，而且只能处理一次）。

`summary_stats()` 是个相对简单的函数，但它给我们一个提示，就是如何编写一个比较复杂的计算程序，如两个列的最小二乘回归或者标准统计，如 `t`-检验。

6.8.2 对查询中有疑问的数据进行编码

包括引号、空值和反斜线的数据值，如果把它们插入到查询中，在执行查询时就会产生一些问题。下面的讨论论述了这些难点，并介绍了解决的办法。

假设要建造一个 SELECT 查询，它基于由 name 指向的空终结串的内容：

```
char query[1024];
```

```
sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", name);
```

如果 name 的值类似于 “0 ' Malley,Brian”，这时进行的查询就是非法的，因为引号在引用的字符串里出现：

```
SELECT * FROM my_tbl WHERE name='0'Malley, Brian'
```

需要特别注意这个引号，以便使服务器不将它解释为 name 的结尾。一种方法是在字符串内使用双引号，这就是 ANSI SQL 约定。SQL 支持这个约定，也允许引号在反斜线后使用：

```
SELECT * FROM my_tbl WHERE name='0''Malley, Brian'
```

```
SELECT * FROM my_tbl WHERE name='0\'Malley, Brian'
```

另一个有问题之处是查询中任意二进制数据的使用，例如，在把图形存储到数据库这样的应用程序中会发生这种情况。因为二进制数值含有一些字符，把它放到查询中是不安全的。

为了解决这个问题，可使用 mysql_escape_string()，它可以对特殊字符进行编码，使其在引用的字符串中可以使用。mysql_escape_string() 认为的特殊字符是指空字符、单引号、双引号、反斜线、换行符、回车符和 Control-Z（最后一个在 Windows 语言环境中出现）。

什么时候使用 mysql_escape_string() 呢？最保险的回答是“始终”。然而，如果确信数据的形式并且知道它是正确的——可能因为预先执行了确认检查——就不必编码了。例如，如果处理电话号码的字符串，它完全由数字和短线组成，那么就不必调用 mysql_escape_string() 了，否则还是要调用。

mysql_escape_string() 对有问题字符进行编码是将它们转换为以反斜线开头的 2 个字符的序列。例如，空字符转换为 ‘\0’，这里的 0 是可打印的 ASCII 码 0，而不是空。反斜线、单引号和双引号分别转换为 ‘\\’、‘\’ 和 ‘\’。

调用 mysql_escape_string() 的过程如下：

```
to_len = mysql_escape_string (to_str, from_str, from_len);
```

mysql_escape_string() 对 from_str 进行编码，并把结果写入 to_str 中，还添加了空终结值，这样很方便，因为可以利用像 strcpy() 和 strlen() 这样的函数使用该结果串。

from_str 指向包括将要编码的字符串的 char 缓冲区，这个字符串可能包含任何内容，其中包括二进制数据。to_str 指向一个存在的 char 缓冲区，在这个缓冲区里，可以写入编码的字符串；不要传递未初始化的指针或 NULL 指针，希望由 mysql_escape_string() 分配空间。由 to_str 指向的缓冲区的长度至少是 (from_len*2)+1 个字节（很可能 from_str 中的每个字符都需要用 2 个字符来编码；额外的字节是空终结值）。

from_len 和 to_len 都是 unsigned int 值，from_len 表示 from_str 中数据的长度；提供这个长度是非常必要的，因为 from_str 可能包含空值字节，不能把它当作空终结串。从 mysql_escape_string() 返回的 to_len 值是作为结果的编码字符串的实际长度，没有对空终结值进行计数。

当 mysql_escape_string() 返回时，to_str 中编码的结果就可看作是空终结串，因为

from_str 中的空值都被编码为 ‘\0’。

为了重新编写构造 SELECT 的代码，使名称的值即使包含引号也能工作，我们进行下面的操作：

```
char query[1024], *p;

p = strcpy (query, "SELECT * FROM my_tbl WHERE name='");
p += strlen (p);
p += mysql_escape_string (p, name, strlen (name));
p = strcpy (p, "'");

是的，这很难看，如要简化一点，就要使用第二个缓冲区，如替换成下列内容：

char query[1024], buf[1024];

(void) mysql_escape_string (buf, name, strlen (name));
sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", buf);
```

6.8.3 图像数据的处理

mysql_escape_string() 的基本功能之一就是把图像数据加载到一个表中。本节介绍如何进行这项工作（这个讨论也适用于二进制数据的其他形式）。

假设想从文件中读取图像，并将它们连同唯一的标识符存储到表中。BLOB 类型对二进制数据来讲是个很好的选择，因此可以使用下面的表说明：

```
CREATE TABLE images
(
    image_id INT NOT NULL PRIMARY KEY,
    image_data BLOB
)
```

实际上，要想从文件中获取图像并放入 images 表，利用下面的函数 load_image() 可以实现，给出一个标识符号码和一个指向包括这个图像数据的打开文件的指针：

```
int
load_image (MYSQL *conn, int id, FILE *f)
{
    char          query[1024*100], buf[1024*100], *p;
    unsigned int   from_len;
    int            status;

    sprintf (query, "INSERT INTO images VALUES (%d,'", id);
    p = query + strlen (query);
    while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
    {
        /* don't overrun end of query buffer! */
        if (p + (2*from_len) + 3 > query + sizeof (query))
        {
            print_error (NULL, "image too big");
            return (1);
        }
        p += mysql_escape_string (p, buf, from_len);
    }
    (void) strcpy (p, "'");
    status = mysql_query (conn, query);
    return (status);
}
```

load_image() 不会分配非常大的查询缓冲区（100K），因此它只能处理相对较小的图形。

在实际的应用程序中，可以根据图形文件的大小动态地分配缓冲区。

处理从数据库中恢复的图形数据（或任何二进制数据）并不像开始把它放入时那样问题重重，因为在变量 `MYSQL_ROW` 中数据值的原始形式是有效的，通过调用 `mysql_fetch_length()`，这个长度也是有效的。必须将值看作是计数串，而不是空终结串。

6.8.4 获取表信息

MySQL 允许使用下面的查询获取有关表结构的信息（下面两者是等价的）：

```
DESCRIBE tbl_name  
SHOW FIELDS FROM tbl_name
```

与 `SELECT` 相类似，两个语句都返回结果集。为了在表中找出有关列，所需做的就是处理结果集中的行，从中获取有用的信息。例如，如果从 `mysql` 客户机上发布 `DESCRIBE images` 语句，就会返回这样的信息：

Field	Type	Null	Key	Default	Extra
image_id	int(11)		PRI	0	
image_data	blob	YES		NULL	

如果从自己的客户机上执行同样的查询，可以得到相同的信息（没有边框）。

如果只想要单个列的信息，则使用如下这个查询：

```
SHOW FIELDS FROM tbl_name LIKE "col_name"
```

此查询会返回相同的列，但只是一行（如果列不存在就不返回行）。

6.8.5 需要避免的客户机程序设计错误

本节讨论一些常见的 MySQL C API 程序设计错误，以及如何避免其发生（这些问题在 MySQL 邮件清单中会周期性地突然出现）。

1. 错误1——使用未初始化的连接处理程序指针

在本章的样例中，我们已经通过传递 `NULL` 参数调用了 `mysql_init()`，这就是让它分配并且初始化 `MYSQL` 结构，然后返回一个指针。另外一种方法是将指针传递到一个已有的 `MYSQL` 结构中。在这种情况下，`mysql_init()` 会将结构初始化并返回一个指针，而不必自己分配结构。如果要使用第二种方法，则要小心会出现一些微妙的问题。下面的讨论指出了需要注意的一些问题。

如果将一个指针传递给 `mysql_init()`，它应该实际指向某些东西。看下面的代码段：

```
main ()  
{  
    MYSQL *conn;  
    mysql_init (conn);  
    ...  
}
```

这个问题是，`mysql_init()` 得到了一个指针，但指针没有指向所知的任何地方。`conn` 是一个局部变量，因此在 `main()` 开始执行时它是一个能指向任何地方的未初始化的存储器，这就是说 `mysql_init()` 将使用指针，并可在内存的一些任意区域滥写。如果幸运的话，`conn` 将指向您的程序地址空间的外部，这样，系统将立即终止，使您能尽早意识到代码中出现的问题。

如果不幸的话, conn 将指向程序中以后才使用的一些数据的内部, 直到再次使用那个数据时才发现问题的地方远比执行程序时出现的问题多, 也更难捕捉到。

下面是一段有问题的代码:

```
MYSQL *conn;

main ()
{
    mysql_init (conn);
    mysql_real_connect (conn, ...)
    mysql_query(conn, "SHOW DATABASES");
    ...
}
```

此时, conn 是一个全局变量, 因此在程序启动前, 将它初始化为 0 (就是 NULL)。mysql_init() 遇到 NULL 参数, 因此初始化并分配一个新的连接处理程序。只要将 conn 传递给需要非NULL 连接处理程序的 MySQL C API 函数, 系统就会崩溃。这些代码段的修改就是确保 conn 有一个可知的值。例如, 可以将它初始化到已经分配的 MYSQL 结构地址中去:

```
MYSQL conn_struct, *conn = &conn_struct;
...
mysql_init (conn);
```

然而, 推荐的(较容易的!) 解决方案仅仅是将 NULL 显式地传递给 mysql_init(), 让该函数分配 MYSQL 结构, 并将返回值赋值给 conn:

```
MYSQL *conn;
...
conn = mysql_init (NULL);
```

无论如何不要忘记检验 mysql_init() 的返回值, 以确保它不是 NULL。

2. 错误2——有效结果集检验的失败

请记住检查希望得到的结果集的调用状态。下面的代码没有做到这一点:

```
MYSQL_RES *res_set;
MYSQL_ROW row;

res_set = mysql_store_result (conn);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* process row */
}
```

不幸地是, 如果 mysql_store_result() 失败, res_set 为 NULL, while 循环也不执行了, 应测试返回结果集函数的返回值, 以确保实际上在进行工作。

3. 错误3—— NULL 列值引起的失败

不要忘记检查mysql_fetch_row() 返回的数组 MYSQL_ROW 中列值是否为 NULL 指针。如果 row[i] 为 NULL, 则在一些机器上, 下面的代码就会引起崩溃:

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i]);
}
fputc ('\n', stdout);
```

该错误危害最大的部分是，有些 `printf()` 的版本很宽容地对 `NULL` 指针输出了 “(null)”，这就使错误很容易逃脱而没有把错误定位。如果把程序给了朋友，而他只有不太宽容 `printf()` 版本，程序就会崩溃，您的朋友会认为您是个无用的程序员。循环应该写成下面这样：

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);

    printf ("%s", row[i] != NULL ? row[i] : "NULL");
}
fputc ('\n', stdout);
```

不需要检查列值是否为 `NULL` 的惟一一次是当已经从列信息结构确定 `IS_NOT_NULL()` 为真时。

4. 错误4——传递无意义的结果缓冲区

需要您提供缓冲区的客户机库函数通常要使这些缓冲区真正存在，下面的代码违反了这个规则：

```
char *from_str = "some string";
char *to_str;
unsigned int len;

len = mysql_escape_string (to_str, from_str, strlen (from_str));
```

问题是什么呢？`to_str` 必须指向一个存在的缓冲区，而在这个样例中没有，因此，它指向了随意的位置。不要向 `mysql_escape_string` 传递无意义的指针作为 `to_str` 参数，否则它会恣意践踏内存。

第7章 Perl DBI API

本章介绍如何使用 Perl DBI 与 MySQL 接口。我们不讨论 DBI 的基本原理或体系结构。有关 DBI 这些方面的信息（特别是与 C 和 PHP API 的比较），请参阅第 5 章。

本章的举例动用了样例数据库 samp_db，使用了学分保存方案和历史同盟需要的表。想要从本章中取得最大收获，最好了解一些有关 Perl 的知识。如果不想这样，那么通过拷贝这里看到的样例代码，也能有所帮助，并可以编写自己的脚本，不过找一本好的 Perl 书，可能仍是一件非常有价值的投资。有这样一本书，名为《Programming Perl》，第二版是由 Wall、Christiansen、Schwartz 和 Potter（O’Reilly出版社1996出版）撰写的。（机械工业出版社1999年已出版了《Perl 5编程详解》——编者注。）

DBI 的当前版本为 1.13，但是此处的大部分介绍也可用于更早的 1.xx 版本。请注意，对所介绍的早期版本中没有出现的特性作了说明。

MySQL 的 DBI 需要至少为 5.004_05 的 Perl 版本。另外还必须安装 Mysql-Mysql 模块和 Data-Dumper Perl 模块，以及 MySQL C 客户机库和一些头文件。如果计划编写基于 Web 的 DBI 脚本，则要使用 CGI.pm 模块。本章中，这个模块用于与 Apache Web 服务器的连接。如果需要获得这样的程序包，请参阅附录 A。该附录中也给出了获得本章开发的样例脚本的说明。可以下载这些脚本，不必自己键入。

很大程度上，本章介绍 Perl DBI 的方法和变量只是出于讨论的需要。至于所有方法和变量的更全面的列表，请参阅附录 G。如果要使用 DBI 的任何部分，可以用该附录作为进一步研究的背景材料。可通过运行下面的命令来得到联机文档：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

在数据库驱动程序（DBD）级，MySQL 的驱动程序建立在 MySQL C 客户机库的基础之上，因而具有它的某些特性。有关该库的详细信息，请参阅第 6 章。

7.1 Perl 脚本的特点

Perl 脚本为文本文件，可以利用任何文本编辑器来创建它们。本章所有的 Perl 脚本都遵从 UNIX 的约定，第一行以 ‘#!’ 开始，接着是执行这个脚本要使用的程序路径名。第一行如下所示：

```
#!/usr/bin/perl
```

如果在您的系统中，路径名不是 Perl，如为 /usr/local/bin/perl5 或 /opt/bin/perl，则需要修改 ‘#!’ 行。否则，Perl 脚本不能在系统中正确运行。

在 ‘#!’ 之后含有一个空格，这是因为有的系统会将 ‘#!/’ 解释为 4 个字节的怪异数字，所以如果没有空格，则忽略这一行，这样，会将相应脚本作为外壳脚本来对待。

在 UNIX 系统中，应该使 Perl 脚本成为可执行文件，以便只要键入其名称就可执行。为使脚本成为可执行文件，对文件模式做如下更改即可：

```
% chmod +x script_name
```

如果在 Windows 下使用 ActiveState Perl, 则不必使脚本成为可执行文件, 可如下运行一个脚本:

```
C:\> perl script_name
```

7.2 Perl DBI 基础

本节提供 DBI 的背景信息——在编写自己的脚本和支持其他人编写的脚本时, 需要这些信息。如果已经熟悉 DBI, 则可以略过这节, 直接跳到 7.3 节“运行 DBI”。

7.2.1 DBI 数据类型

从某些方面来说, 使用 Perl DBI API 类似于使用第 6 章介绍的 C 客户机库。在使用 C 客户机库时, 主要依靠指向结构或数组的指针来调用函数和访问与 MySQL 相关的数据。在使用 DBI API 时, 除了函数称为方法, 指针称为引用外, 也调用函数和使用指向结构的指针。指针变量称为句柄, 句柄指向的结构称为对象。

DBI 使用若干种句柄。它们往往通过表 7-1 所示的惯用名称在 DBI 文件中引用。而惯用的非句柄变量的名称如表 7-2 所示。实际上, 在本章中, 我们并不使用每个变量名, 但是, 在阅读其他人编写的 DBI 脚本时, 了解它们是有用的。

表 7-1 惯用的 Perl DBI 句柄变量名

名 称	说 明
\$dbh	数据库对象的句柄
\$sth	语句 (查询) 对象的句柄
\$fh	打开文件的句柄
\$h	“通用”句柄; 其意义取决于上下文

表 7-2 惯用的 Perl DBI 非句柄变量的名称

名 称	说 明
\$rc	从返回真或假的操作中返回的代码
\$rv	从返回整数的操作中返回的值
\$rows	从返回行数的操作中返回的值
@ary	查询返回的表示一行值的数组 (列表)

7.2.2 一个简单的 DBI 脚本

让我们从一个简单脚本 dump_members 开始, 它举例说明了 DBI 程序设计中若干标准概念, 如与 MySQL 服务器的连接和断开、检索数据等。此脚本产生的结果为以制表符分隔形式列出的历史同盟成员。这个格式本身并不让人感兴趣: 在这里, 了解如何使用 DBI 比产生漂亮的输出更为重要。

dump_members 如下:

```
#!/usr/bin/perl
```

```
# dump_members - dump Historical League's membership list

use DBI;
use strict;

my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul"; # user name
my ($password) = "secret"; # password
my ($dbh, $sth); # database and statement handles
my (@ary); # array for rows returned by query

# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });

# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();

# read results of query, then clean up
while (@ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
$sth->finish ();

$dbh->disconnect ();
exit (0);
```

要想自己试验这个脚本，可以下载它（请参阅符录 A），或使用文本编辑器创建它，然后使之可执行，以便能运行。当然，可能至少需要更改一些连接参数（主机名、数据库名、用户名和口令）。本章中的其他 DBI 脚本也是这样。在参数缺省时，本章下载脚本的权限设置为只允许读。如果您将自己的 MySQL 用户名和口令放在它们之中，我建议将它们保留为这种方式，以便其他人不能读取这些值。以后，在 7.2.8 节“指定连接参数”中，我们将看到如何从选项文件中获得这些参数，而不是将它们直接放在脚本中。

现在，让我们逐行看完这个脚本。第一行是标准行，指出哪里可以找到 Perl 的指示器：

```
#!/usr/bin/perl
```

在本章将要讨论的脚本中，每个脚本都包含这行；以后不再说明。此脚本中至少应该含有一个简短的目的说明，这是一个好主意，所以下一行是一个注释，给阅读此脚本的人提供一个关于它做什么的线索：

```
# dump_members - dump Historical League's membership list
```

从‘#’字符到行尾部的文本为注释。有必要做一些练习，就是在整个脚本中编写一些注释来解释它们如何工作。

接下来是两个 use 行：

```
use DBI;
use strict;
```

use DBI 告知 Perl 解释程序它需要引入 DBI 模块。如果没有这一行，试图在脚本中做与 DBI 相关的任何事，都将出现错误。不需要指出想要哪个 DBD 级别的模块。在连接数据库时，DBI 会激活相应的模块。

use strict 告知 Perl，在使用它们之前需要声明变量。如果没有 use strict 行，也可以编写脚本，但是，它有助于发现错误，所以建议始终要包括这行。例如，置为严格模式时，如果

声明变量 `$my_var`，但是之后错误地用 `$mv_var` 来访问，则在运行这个脚本时，将获得下面的消息：

```
Global symbol "$mv_var" requires explicit package name at line n
```

这个消息会使您想，“怎么了？`$mv_var`？我从未使用过这种名称的变量！”，然后，找到脚本中的第 n 行，看是什么问题，并改正它。如果不用严格模式，Perl 不会给出 `$mv_var`；将只是简单地按具有 `undef`（未定义的）值的该名称创建一个新的变量，并毫无动静地使用它，然后，您会莫名其妙脚本为什么不工作。

因为我们在严格模式下操作，所以我们将定义脚本使用的变量：

```
my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul"; # user name
my ($password) = "secret"; # password
my ($dbh, $sth); # database and statement handles
my (@ary); # array for rows returned by query
```

现在我们准备连接数据库：

```
# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });
```

`connect()` 调用作为 `DBI->connect()` 来调用，因为它是 `DBI` 类的方法。不必真正知道它是什么意思；它只是一个使人头痛的面向对象的行话（如果的确想知道，那么它意味着

`connect()` 是“属于”`DBI` 的一个函数）。`connect()` 有若干参数：

■ **数据源**。（经常调用的数据源名称，或 DSN。）数据源格式由要使用的特定 `DBD` 模块需求来确定。对于 MySQL 驱动程序，允许的格式如下：

```
"DBI:mysql:db_name"
"DBI:mysql:db_name:host_name"
```

对于第一种格式，主机名缺省为 `localhost`（实际上有其他允许的数据源格式，我们将在后面 7.2.8 节“指定连接参数”中讨论）。“`DBI`”大写没关系，但是“`mysql`”必须小写。

■ **用户名和口令**。

■ **表示额外连接属性的可选参数**。这个参数控制 `DBI` 的错误处理行为，我们指定的看起来有点奇怪的构造启用了 `RaiseError` 属性。这导致 `DBI` 检查与数据库相关的错误，并显示消息，而且只要它检测到错误就退出（这就是为什么在 `dump_members` 脚本中的任何地方都没有看到错误检查代码的原因；`DBI` 将它全部处理了）。7.2.3 节“处理错误”包括了对错误响应的可选方法。

如果 `connect()` 调用成功，则它返回数据库句柄，我们分配给 `$dbh`（如果 `connect()` 失败，通常返回 `undef`。然而，因为我们在脚本中启用了 `RaiseError`，所以 `connect()` 不返回；但是，`DBI` 将显示一条错误消息，并且在出现错误时退出）。

连接到数据库后，`dump_members` 发布一条 `SELECT` 语句查询来检索全体成员列表，然后，执行一个循环来处理返回的每一行。这些行构成了结果集。

为了完成 `SELECT` 语句，首先需要准备，然后再运行它：

```
# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();
```

利用数据库句柄调用 `prepare()`；在执行前，它将 `SQL` 语句传递给预处理的驱动程序。实

际上，在这里某些驱动程序做了一些有关这条语句的事情。其他驱动程序只是记住它，直到调用 `execute()` 使这条语句被执行为止。从 `prepare()` 返回的值是一个语句句柄 `$sth`，如果出现错误，则为 `undef`。在进一步处理与这条语句相关的所有内容时，都使用这个语句句柄。

请注意，指定的这个查询没有分号结束符。您无疑有这样的（经过长时间使用 `mysql` 程序养成的）习惯，用 `;` 字符终止 SQL 语句。然而，在使用 DBI 时，最好打破这个习惯，因为分号经常导致查询出现语法错误而失败。向查询增加 `'\g'` 也类似，使用 DBI 时不要这样。

在调用一个方法而不用向它传递任何参数时，可以没有这个圆括号。下列两个调用是等价的：

```
$sth->execute ();  
$sth->execute;
```

我宁愿有圆括号，因为它使人感到这个调用看上去不像变量。您的选择就可能不同了。

调用 `execute()` 后，可以处理成员列表的行。在 `dump_members` 脚本中，提取行的循环简单地显示了每行的内容：

```
# read results of query, then clean up  
while (@ary = $sth->fetchrow_array ())  
{  
    print join ("\t", @ary), "\n";  
}  
$sth->finish ();
```

`fetchrow_array()` 返回含有当前行的列值的数组，在没有剩余的行时，返回一个空数组。这样，此循环提取了由 `SELECT` 语句返回的连续行，并显示列值之间用制表符分隔的每一行。在数据库中 `NULL` 作为 `undef` 值返回到 Perl 脚本，但是将它们显示为空字符串，而不是单词“`NULL`”。

请注意，制表符和换行符（表示为 `'\t'` 和 `'\n'`）括在双引号中。在 Perl 中，只解释出现在双引号内的转义符序列，不解释出现在单引号内的转义符序列。如果使用单引号，则输出将为字符串 `"\t"` 和 `"\n"`。

提取行的循环终止以后，调用 `finish()` 告知 DBI 不再需要语句句柄，并且释放分配给它的所有临时资源。实际上，除非只提取结果集的一部分（无论是设计的原因，还是因为出现一些问题），否则不需要调用 `finish()`。然而，在提取循环之后，`finish()` 始终是很保险的，我认为调用并执行 `finish()`，比区分何时需要，何时不需要更容易一些。

我们已经显示完了全部成员列表，所以我们可以从服务器上断开连接，并且退出：

```
$dbh->disconnect ();  
exit (0);
```

`dump_members` 示出了许多 DBI 程序的大多数通用概念，而且不必了解更多的知识，就可以着手编写自己的 DBI 程序。例如，要想写出一些其他表的内容，所需要做的只是更改传递给 `prepare()` 方法的 `SELECT` 语句的文本。而且实际上，如果了解这种技术的某些应用，可略过这部分，直接跳到 7.3 节“运行 DBI”中讨论如何生成历史同盟一年一度的宴会成员列表程序和 `League` 打印目录的部分。然而，DBI 提供许多其他有用的功能。下一节介绍了一些，以便能够在 Perl 脚本中看看如何完成比运行一条简单的 `SELECT` 语句更多的事情。

7.2.3 处理错误

在 `dump_members` 调用 `connect()` 方法时，应该启用 `RaiseError` 错误处理属性，以便这些

错误用一条错误消息就能自动地终止相应的脚本。也可以用其他方式处理这些错误。例如，可以自己检查错误而不必使用 DBI。

为了查看如何控制 DBI 的错误处理行为，我们来仔细查看一下 `connect()` 调用的最终参数。下面两个相关的属性是 `RaiseError` 和 `PrintError`：

- 如果启用 `RaiseError`（设为非零值），如果在 DBI 方法中出现错误，则 DBI 调用 `die()` 来显示一条消息并且退出。
- 如果启用 `PrintError`，在出现 DBI 错误时，DBI 会调用 `warn()` 来显示一条消息，但是相应脚本会继续执行。

缺省时，`RaiseError` 是禁用的，而 `PrintError` 启用。在此情况下，如果 `connect()` 调用失败，则 DBI 显示一条消息，而且继续执行。这样，如果省略 `connect()` 的四个参数，则得到缺省的错误处理行为，可以如下检查错误：

```
$dbh = DBI->connect ($dsn, $user_name, $password)
or exit (1);
```

如果出现错误，则 `connect()` 返回 `undef` 表示失败，并且触发对 `exit()` 的调用。因为 DBI 已经显示了错误消息，所以您就不一定要显示它了。

如果明确给出该错误检查属性的缺省值，可如下调用 `connect()`。

```
$dbh = DBI->connect ($dsn, $user_name, $password,
                    { RaiseError => 0, PrintError => 1 })
or exit (1);
```

这就需要更多的编写工作，但是即使对不经意的读者，处理错误行为也会更为明显。

如果想自己检查错误，并显示自己的消息，应该禁用 `RaiseError` 和 `PrintError`：

```
$dbh = DBI->connect ($dsn, $user_name, $password,
                    { RaiseError => 0, PrintError => 0 })
or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

变量 `$DBI::err` 和 `$DBI::errstr`，只用于所显示的 `die()` 调用中，有助于构造错误消息。它们含有 MySQL 错误代码和错误字符串，非常像 C API 函数中的 `mysql_errno()` 和 `mysql_error()`。

如果仅仅要 DBI 处理错误，以便不必自己检查它们，则启用 `RaiseError`：

```
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });
```

到目前为止，这是最容易的方法，并且是 `dump_members` 带来的。如果在脚本退出时，想要执行某种类型的清除代码，启用 `RaiseError` 可能是不恰当的，尽管在这种情况下，可以重新定义 `$_SIG{$_DIE_}` 句柄，可以做想做的事情。

避免启用 `RaiseError` 属性的另一个原因是 DBI 在它的消息中显示技术信息，如下：

```
disconnect(DBI::db=HASH(0x197aae4)) invalidates 1 active statement. Either
destroy statement handles or call finish on them before disconnecting.
```

对于编程者来说，这是好的信息，但对普通用户可能没有什么意义。在此情形，最好自己检查错误，以便可以显示对期望使用这个脚本的人更有意义的消息。或者也可在这里考虑重新定义 `$_SIG{$_DIE_}` 句柄。这样可能很有用，因为它允许启用 `RaiseError` 来使错误处理简单化，而不是用自己的消息替换 DBI 给出的缺省错误消息。为了提供自己的 `_DIE_` 句柄，可在执行任何 DBI 调用以前，进行下面的工作：

```
$SIG{__DIE__} = sub { die "Sorry, an error occurred\n"; };
```

也可以用普通的风格定义一个子例程，并利用这个子例程的引用来设置这个句柄值：

```
sub die_handler
{
    die "Sorry, an error occurred\n";
}
```

```
$SIG{__DIE__} = \&die_handler;
```

除了在 connect() 调用中逐字传递错误处理属性之外，还可以利用散列定义它们，并传递对这个散列的引用。有人发现以这种方式准备属性设置使脚本更容易阅读和编辑，但是在功能上这两种方法是相同的。下面是一个说明如何使用属性散列的样例：

```
%attr =
(
    PrintError => 0,
    RaiseError => 0
);
$dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
    or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

下面的脚本 dump_members2 举例说明了当要自己检查错误并显示自己的消息时，如何编写脚本。dump_member2 处理和 dump_members 一样的查询，但是明确地禁用 PrintError 和 RaiseError，然后测试每个 DBI 调用的结果。如果出现错误，在退出以前，脚本调用了子例程 bail_out() 显示消息及 \$DBI::err 和 \$DBI::errstr 的内容：

```
#!/usr/bin/perl

# dump_members2 - dump Historical League's membership list

use DBI;
use strict;

my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul"; # user name
my ($password) = "secret"; # password
my ($dbh, $sth); # database and statement handles
my (@ary); # array for rows returned by query
my (%attr) = # error-handling attributes
(
    PrintError => 0,
    RaiseError => 0
);

# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
    or bail_out ("Cannot connect to database");

# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name")
    or bail_out ("Cannot prepare query");
$sth->execute ()
    or bail_out ("Cannot execute query");

# read results of query
while (@ary = $sth->fetchrow_array ())
{
    . . . . .
}
```

```

    print join ("\t", @ary), "\n";
}
$DBI::err == 0
    or bail_out ("Error during retrieval");

# clean up
$sth->finish ()
    or bail_out ("Cannot finish query");
$dbh->disconnect ()
    or bail_out ("Cannot disconnect from database");
exit (0);

# bail_out() subroutine - print error code and string, then exit

sub bail_out
{
    my ($message) = shift;

    die "$message\nError $DBI::err ($DBI::errstr)\n";
}

```

除了 `bail_out()` 是退出而不是返回到调用者以外，`bail_out()` 类似于我们在第6章中为编写 C 程序使用的 `print_error()` 函数。每次想显示错误消息时，`bail_out()` 解除了写出 `$DBI::err` 和 `$DBI::errstr` 名称的麻烦。同样，通过封装显示到子例程的错误消息，可更改子例程使整个脚本中错误消息的格式一致。

`dump_member2` 脚本在提取行循环的后面有一个测试，这是 `dump_members` 所没有的。因为如果在 `fetchrow_array()` 中出现错误，`dump_members2` 不会自动地退出，所以人们判断循环是因为结果集读取完成而终止（正常终止），还是因为出现错误而终止做出确定是很困难的。当然，任何一种方式，循环都将终止，但是如果出现错误，则将删截脚本的输出。如果没有错误检查，运行该脚本的人将无法知道是否有错！如果自己检查错误，应该检查提取循环的结果。

7.2.4 处理不返回结果集的查询

`DELETE`、`INSERT`、`REPLACE`和`UPDATE` 等执行后不返回行的语句比 `SELECT`、`DESCRIB`、`EXPLAIN` 和 `SHOW` 等执行后返回行的语句的处理相对要容易一些。为处理一条非 `SELECT` 语句，利用数据库句柄，将它传递给 `do()`。`do()` 方法在一个步骤内准备和执行该查询。例如，开始输入一个新的成员，`Marcis Brown`，终止日期为 2002 年 6 月 3 日，可以这样做：

```

$rows = $dbh->do ("INSERT member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2002-6-3')");

```

`do()` 方法返回涉及行的计数，如果出现错误，则返回 `undef`。因为各种原因，可能出现错误（例如，这个查询可能是畸形的，或可能没有访问这个表的权力）。对于非 `undef` 的返回，注意那些没有受到影响的行的情况。当这种情况发生时，`do()` 不返回数字 0；而是返回字符串“0E0”（0的Perl科学计数法形式）。“0E0”在数值上等价于 0，但是，在条件测试中将其视为真，以便可以将其与早期的 `undef` 区别。如果 `do()` 返回 0，则区分是出现了错误（`undef`）还是“没有受到影响的行”这两种情况将更困难。使用下面的两个测试之一可以检查错误：

```

if (!defined ($rows)) { # error }
if (!$rows) { # error }

```

在数值环境中,“0E0”与0等价。下面的代码将正确地显示 \$rows 的任何非 undef 值的行数:

```
if (!$rows)
{
    print "error\n";
}
else
{
    $rows += 0; # force conversion to number if "0E0"
    print "$rows rows affected\n";
}
```

也可以用 printf() 使用 ‘%d’ 格式显示 \$row 来强制进行隐含的数字转换:

```
if (!$rows)
{
    print "error\n";
}
else
{
    printf "%d rows affected\n", $rows;
}
```

do() 方法等价于后跟 execute() 的 prepare()。前面的 INSERT 语句可以不调用 do(), 如下发布:

```
$sth = $dbh->prepare ("INSERT member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2002-6-3')");
$rows = $sth->execute ();
```

7.2.5 处理返回结果集的查询

本章提供了有关实现 SELECT 查询中提取行循环的若干选项的详细信息 (或其他类似于 SELECT 的返回行的查询,如 DESCRIBE、EXPLAIN 和 SHOW)。还讨论了如何获得结果中行数的计数值, 如何处理不需要循环的结果集, 以及如何一次检索整个结果集的全部内容等。

1. 编写提取行的循环

dump_members 脚本利用 DBI 方法的标准序列检索数据: prepare() 使驱动程序处理查询, execute() 开始执行这个查询, fetchrow_array() 提取结果集中的每一行, finish() 释放与这个查询相关的资源。

prepare()、execute() 和 finish() 是处理返回行的查询中非常标准的部分。然而, 对于提取的行, fetchrow_array() 实际上只是若干方法中的一种 (请参阅表 7-3)。

表7-3 DBI 提取行的方法

方 法 名	返 回 值
fetchrow_array()	行值的数组
fetchrow_arrayref()	对行值数组的引用
fetch()	与 fetchrow_arrayref() 相同
fetchrow_hashref()	对行值的散列引用, 列名键索引

下面的例子说示出了怎样使用每个提取行方法。这些例子在整个结果集的行中循环, 对

于每一行，显示由逗号分隔的列值。在某些情况下，编写这些显示代码还有一些更有效的方法，但是这些例子是以能够说明访问单个列值的语法的方式编写的。

可如下使用 `fetchrow_array()`：

```
while (@ary = $sth->fetchrow_array ())
{
    $delim = "";
    for ($i = 0; $i < @ary; $i++)
    {
        print $delim . $ary[$i];
        $delim = ",";
    }
    print "\n";
}
```

对 `fetchrow_array()` 的每个调用都返回行值数组，不再有行时，返回一个空数组。

选择将返回值分配给数组变量，可以在一组标量变量中提取列值。如果想使用比 `$ary[0]`、`$ary[1]` 等更有意义的变量名，就可以这样做。假设要在变量中检索名称和电子邮件值，可使用 `fetchrow_array()`，可以如下选择并提取行：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email"
    . " FROM member ORDER BY last_name");

$sth->execute ();
while (($last_name, $first_name, $suffix, $email)
    = $sth->fetchrow_array ())
{
    # do something with variables
}
```

当然，在以这种方式使用一系列变量时，必须保证查询按正确的次序选择列。DBI 不关心 `SELECT` 语句指定列的次序，所以正确地分配变量是您的职责。在提取行时，使用一种称为参数约束的技术，也可以使列值自动分配给单独的变量。

`fetchrow_arrayref()` 类似于 `fetchrow_array()`，但不返回包含当前行的列值的数组，而是返回这个数组的引用，在没有剩余行时，返回 `undef`。如下使用：

```
while ($ary_ref = $sth->fetchrow_arrayref ())
{
    $delim = "";
    for ($i = 0; $i < @{$ary_ref}; $i++)
    {
        print $delim . $ary_ref->[$i];
        $delim = ",";
    }
    print "\n";
}
```

通过数组引用 `$ary_ref` 访问数组元素。这类似于引用指针，所以使用了 `$ary_ref->[$i]` 而不是 `$ary[$i]`。要想引用整个数组，就要使用 `@{$ary_ref}` 结构。

`fetchrow_arrayref()` 不适合在列表中提取变量。例如，下面的循环不起作用：

```
while (($last_name, $first_name, $suffix, $email)
    = @{$sth->fetchrow_arrayref ()})
{
    # do something with variables
}
```

实际上，只要 `fetchrow_arrayref()` 提取一行，这个循环就能正确地运行。但是在没有更

多的行时, `fetchrow_arrayref()` 返回 `undef`, 并且 `@{undef}` 不合法 (它有些像在 C 程序中试图废弃一个 NULL 指针)。

提取行的第三个方法 `fetchrow_hashref()`, 如下使用:

```
{
    $delim = "";
    foreach $key (keys (%{$hashref}))
    {
        print $delim . $hashref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

对 `fetchrow_hashref()` 的每个调用都返回一个按列名索引的行值散列的引用, 在没有更多的行时, 返回 `undef`。在此情况下, 列值不按特定的次序出现; Perl 散列的成员是无序的。然而, 散列元素是按列名索引的, 所以 `$hashref` 提供了一个单独的变量, 可通过它按名称访问任何列值。这使得能按任意需要的次序来提取值 (或者它们中的任何子集), 而且不必知道 `SELECT` 查询检索的列的次序。例如, 如果想访问名称和电子邮件域, 可以如下进行:

```
while ($hashref = $sth->fetchrow_hashref ())
{
    $delim = "";
    foreach $key ("last_name", "first_name", "suffix", "email")
    {
        print $delim . $hashref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

如果希望将一行值传递给某个函数而又不需要这个函数知道 `SELECT` 语句中指定列的次序时, `fetchrow_hashref()` 是非常有用的。既然如此, 可以调用 `fetchrow_hashref()` 来检索行, 并且编写一个使用列名访问来自行散列值的函数。

如果使用 `fetchrow_hashref()`, 请记住下列警告:

- 如果性能很重要, 则 `fetchrow_hashref()` 并不是最好的选择, 因为它没有 `fetchrow_array()` 或 `fetchrow_arrayref()` 的效率。
- 作为散列键值使用的列名具有与 `SELECT` 语句中写出时相同的字符。在 MySQL 中, 列名不区分大小写, 所以此查询也是这样, 不管以大写字母还是小写字母给出列名, 查询结果都是一样的。但是 Perl 散列索引名是区分大小写的, 这可能会带来一些问题。为了避免潜在的大小写不匹配问题, 可通过传递 `NAME_lc` 或 `NAME_uc` 属性, 告知 `fetchrow_hashref()` 强迫列名为大写或小写:

```
$hash_ref = $sth->fetchrow_hashref ('NAME_lc'); # use lowercase names
$hash_ref = $sth->fetchrow_hashref ('NAME_uc'); # use uppercase names
```

- 散列对每个唯一的列名含有一个元素。如果正在执行从多个具有重叠名称的表中返回行的连接, 则不能访问所有的列值。例如, 如果发布下面的查询, `fetchrow_hashref()` 将返回只有一个元素的散列:

```
SELECT a.name, b.name FROM a, b WHERE a.name = b.name
```

2. 确定查询返回的行数

如何知道 SELECT 或类似于 SELECT 的查询返回的行数？一种方法是，当提取它们时，计算这些行的数量。实际上，这是知道 SELECT 查询返回多少行的唯一方便的方法。使用 MySQL 驱动程序，可以在调用 `execute()` 后利用语句句柄调用 `rows()` 方法，但是这对其他数据库引擎并不方便。而且即使就 MySQL 来说，如果已经设置了 `mysql_use_result` 属性，`rows()` 也不能返回正确的结果，直到提取了所有行（有关的详细信息，请参阅附录 G）。所以只能如提取行一样对它们进行计数。

3. 提取单行的结果

如果结果集只含单个行，则不需要运行循环来获得结果。假设要编写得出历史同盟成员当前数量的脚本 `count_members`。完成查询的代码如下所示：

```
# issue query
$sth = $dbh->prepare ("SELECT COUNT(*) FROM member");
$sth->execute ();

# read results of query, then clean up
$count = $sth->fetchrow_array ();
$sth->finish ();
$count = "can't tell" if !defined ($count);
print "$count\n";
```

SELECT 语句只返回一行，所以不需要循环；我们只调用 `fetchrow_array()` 一次。另外，因为我们只选择一列，所以甚至不需要将返回值分配给数组。当在标量环境中（单个值而不是所期望的一列）调用 `fetchrow_array()` 时，它返回这个行的第一列，如果没有更多的有效行，则返回 `undef`。

另一种期望最多有一个记录的查询是一个含有 `LIMIT 1` 来约束返回的行数的查询。其一般的用法是返回特定列含有最大或最小值的行。例如，下面的查询给出最近出生的总统姓名和出生日期：

```
$query = "SELECT last_name, first_name, birth"
        . " FROM president ORDER BY birth DESC LIMIT 1";
$sth = $dbh->prepare ($query);
$sth->execute ();

# read results of query, then clean up
($last_name, $first_name, $birth) = $sth->fetchrow_array ();
$sth->finish ();
if (!defined ($last_name))
{
    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

必须无提取循环的其他类型的查询利用 `MAX()` 或 `MIN()` 来选择单个值。但是在所有这些情况下，获得单个行结果的一种更容易的方法就是使用数据库句柄方法 `selectrow_array()`，它结合了 `prepare()`、`execute()` 并在单个调用中提取行。它返回一个数组（而不是一个引用），如果出现错误，则返回一个空数组。前一例子可利用 `selectrow_array()` 编写如下：

```
$query = "SELECT last_name, first_name, birth"
        . " FROM president ORDER BY birth DESC LIMIT 1";
($last_name, $first_name, $birth) = $dbh->selectrow_array ($query);
if (!defined ($last_name))
```

```

{
    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}

```

4. 处理完整的结果集

在使用提取循环时，DBI 不提供在结果集中随意查找的方法，或以任何次序而不是以循环返回的次序来处理行。同样，提取行以后，如果没有保存，前一行会丢失。这种做法并不一定合适以下情况：

- 以非连续的次序处理行。考虑一种情况，想以历史同盟的 president 表中列出的美国总统为主体，进行一些测验。如果希望每次测验时都以不同的次序提出问题，则可以从 president 表中选择所有行。然后，可能以任意的次序提取行来改变与所问问题有关的总统的次序。要想任意地提取一行，就必须同时访问所有的行。
- 只使用返回行的子集，对其进行随机选择。例如，当问及总统出生地时，要想出现多个选择的问题，则可以随便地提取一行来选择总统（正确的答案），然后再从取来干扰的选择中提取若干其他行。
- 即使确实以连续的次序去处理，也想紧紧抓住整个结果集。如果想经过这些行进行多个传递，这可能是必需的。例如，在统计计算中，可能先浏览一遍结果集，来估计数据的一些通用数字属性，然后再次检查这些行，来实现更加明确的分析。

可以用几个不同的方式作为一个整体访问结果集。可以完成这个常见的提取循环，并在提取它时保存每一行，可以使用一次返回整个结果集的方法。无论哪种方法都以在结果集中包括一行一行的矩阵作为结束，和选择的列一样多。可以以任何次序任意多次地处理矩阵的元素。下面的讨论说明这两种方法。

使用提取循环来捕获结果集的一种方法是使用 `fetchrow_array()` 并保存对这些行引用的数组。除了保存所有的行，然后显示矩阵举例说明了如何确定矩阵中的行数 and 列数，及如何访问矩阵的个别成员以外，下面的代码和 `dump_members` 中提取和显示的循环作用是一样的。

```

my (@matrix) = (); # array of array references

while (my @ary = $sth->fetchrow_array ()) # fetch each row
{
    push (@matrix, [ @ary ]); # save reference to just-fetched row
}
$sth->finish ();

# determine dimensions of matrix
my ($rows) = scalar (@matrix);
my ($cols) = ($rows == 0 ? 0 : scalar (@{$matrix[0]}));

for (my $i = 0; $i < $rows; $i++) # print each row
{
    my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix[$i][$j];
        $delim = ",";
    }
}

```

```
print "\n";
}
```

在确定矩阵的维数时，必须首先确定行数，因为无论这个矩阵是否为空，都可能计算列数。如果 \$rows 为0，则这个矩阵为空，并且 \$cols 也为0。否则，列数可能作为行数组中的元素数量来计算，用语法 @{\$matrix[\$i]} 来整体访问行 \$i。

在前述的样例中，我们提取每一行，然后保存对它的引用。可以设想调用 fetchrow_arrayref() 而不是直接地检索行引用可能更有效率：

```
my (@matrix) = (); # array of array references

while (my $ary_ref = $sth->fetchrow_arrayref ())
{
    # this will not work!
    push (@matrix, $ary_ref); # save reference to just-fetched row
}
$sth->finish ();
```

它不能正常工作，因为 fetchrow_arrayref() 重新使用了引用指向的数组。结果矩阵是一个引用的数组，数组中的每个元素都指向相同行——最后检索的行。因此，如果想一次提取一行，则使用 fetchrow_array() 而不是 fetchrow_arrayref()。

另一个选择是使用提取循环，可以使用返回整个结果集的 DBI 方法中的一个。例如，fetchall_arrayref() 返回对引用数组的引用，数组的每个元素都指向结果集中某行。这非常简单，但很有效，这个返回值是对矩阵的引用。要想使用 fetchall_arrayref()，则调用 prepare() 和 execute()，然后如下检索结果：

```
my ($matrix_ref); # reference to array of references

$matrix_ref = $sth->fetchall_arrayref (); # fetch all rows

# determine dimensions of matrix
my ($rows) = (!defined ($matrix_ref) ? 0 : scalar @{$matrix_ref});
my ($cols) = ($rows == 0 ? 0 : scalar @{$matrix_ref->[0]});

for (my $i = 0; $i < $rows; $i++) # print each row
{
    my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}
```

如果结果集为空，则 fetchall_arrayref() 返回一个对空数组的引用。如果出现错误，则结果为 undef，所以如果没有启用 RaiseError，则在开始使用它以前，要确保检查返回值。

行数和列数由矩阵是否为空来确定。如果想作为一个数组访问这个矩阵的整个行 \$i，应使用该语法 @{\$matrix_ref->[\$i]}。

使用 fetchall_arrayref() 来检索结果集当然比编写一个提取行的循环要更简单一些，尽管访问数组元素的语法是一个小技巧。一个与 fetchall_arrayref() 方法相类似，但却做了更多工作的方法是 selectall_arrayref()。这个方法为您完成了整个 prepare()、execute()、提取循环、finish() 序列。为了使用 selectall_arrayref()，应该利用数据库句柄直接将查询传递给它：

```
my ($matrix_ref); # reference to array of references

$matrix_ref =
    $dbh->selectall_arrayref ("SELECT last_name, first_name, suffix, email,"
        . "street, city, state, zip, phone FROM member ORDER BY last_name");

# determine dimensions of matrix
my ($rows) = (defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my ($cols) = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)          # print each row
{
    my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}
```

5. 检查 NULL 值

从数据库中检索数据时，可能需要区分值为 NULL、为 0 或者为空字符串的列。因为 DBI 返回 NULL 列值作为 undef，所以这种区分是容易的。然而，必须确保使用正确的测试。如果试用下面的代码段，则它所有三次显示都为“false!”：

```
$col_val = undef; if (!$col_val) { print "false!\n"; }
$col_val = 0;    if (!$col_val) { print "false!\n"; }
$col_val = "";   if (!$col_val) { print "false!\n"; }
```

而且，对于这两个测试，这段代码都显示“false!”：

```
$col_val = undef; if ($col_val eq "") { print "false!\n"; }
$col_val = "";    if ($col_val eq "") { print "false!\n"; }
```

下面这段代码效果相同：

```
$col_val = "";
if ($col_val eq "") { print "false!\n"; }
if ($col_val == 0) { print "false!\n"; }
```

要想区分 NULL 列值和非 NULL 列值，则使用 defined()。知道了没有出现 NULL 值之后，使用适当的测试可以在其他类型值之间加以区分。例如：

```
if (defined ($col_val)) { print "NULL\n"; }
elsif ($col_val eq "") { print "empty string\n"; }
elsif ($col_val == 0) { print "zero\n"; }
else { print "other\n"; }
```

以适当的次序完成这些测试是很重要的，因为如果 \$col_val 为空字符串，则第二个和第三个比较就都为真。如果颠倒比较的次序，则会错误地将空字符串标识为 0。

7.2.6 引用问题

迄今为止，我们已经利用引用字符串以最基本的方式构造了查询。在引用的字符串含有引用值时，会在 Perl 词汇一级产生问题。在插入或者选择含有引号、反斜杠或二进制数据的值时，在 SQL 中也可能出问题。如果指定一个查询作为 Perl 引用的字符串，则必须避免在查询字符串本身中出现引用字符：


```
$query = 'INSERT absence VALUES(14,\ '1999-9-16\')';
$query = "INSERT absence VALUES(14,\"1999-9-16\")";
```

Perl 和 MySQL 都允许用单引号或双引号引用字符串，所以混合使用引用字符有时可以避免这种无法引用引用字符自身的情况：

```
$query = 'INSERT absence VALUES(14,"1999-9-16")';
$query = "INSERT absence VALUES(14,'1999-9-16')";
```

然而，在 Perl 中，这两种类型的引号并不等价。只有在双引号内部才解释为变量引用。因此，当想通过在查询字符串中嵌入变量引用来构造查询时，单引号并不是非常有用的。例如，如果 \$var 的值为 14，则下面的两个字符串并不等价：

```
"SELECT * FROM member WHERE id = $var"
'SELECT * FROM member WHERE id = $var'
```

两个字符串的解释如下所示；显然，第一个字符串与希望传递给 MySQL 服务器的内容更为相像：

```
"SELECT * FROM member WHERE id = 14"
'SELECT * FROM member WHERE id = $var'
```

用双引号来引用字符串的另一个选择是使用 qq{} 结构，它告诉 Perl 在 ‘ qq{ ’ 和 ‘ } ’ 之间的每个字符都要看作为双引号括起的字符串（两个 q 表示“双引号”）。例如，下列两行是等价的：

```
$date = "1999-9-16";
$date = qq{1999-9-16};
```

使用 qq{} 时，构造查询不用过多考虑引号的问题，因为可以在这个查询字符串内自由地使用引号（单引号或双引号），而不用避开它们。此外，还解释了变量引用。qq{} 的这两种特性可用下面的 INSERT 语句来说明：

```
$id = 14;
$date = "1999-9-16";
$query = qq{INSERT absence VALUES($id,"$date")};
```

不一定使用 ‘ { ’ 和 ‘ } ’ 作为 qq 的分隔符。其他格式，如 qq() 和 qq//，也可以使用，只要封闭的分隔符不出现在字符串内即可。我喜欢用 qq{}，因为 ‘ { ’ 不像 ‘) ’ 或 ‘ / ’ 会出现在查询的文本内，并且在查询字符串的结尾也可能有问题。例如，‘) ’ 出现在所显示的 INSERT 语句的内部，所以 qq() 对于引用查询字符串来说不是一个有用的结构。

qq{} 结构能跨行，如果想让查询字符串在 Perl 代码中醒目，这很有用：

```
$id = 14;
$date = "1999-9-16";
$query = qq{
    INSERT absence VALUES($id,"$date")
};
```

如果希望将查询格式分为多个行，从而使它的可读性更强，这样也很有用。例如，dump_members 脚本中的 SELECT 语句如下：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
```

用 qq{} 编写如下：

```
$sth = $dbh->prepare (qq{
    SELECT
        last_name, first_name, suffix, email,
```

```

        street, city, state, zip, phone
    FROM member
    ORDER BY last_name
});

```

双引号字符串也可以跨行。但是，对于编写多行的字符串，我更喜欢用 qq{}。我发现当在一行中看到不匹配的引号时，我就自然地去想，“这会是语法错误吗？”，然后，我就会浪费时间去寻找相匹配的引号。

qq{} 结构在 Perl 词汇级注意了引用的问题，因此可将引号容易地放到字符串内，而不会使 Perl 搞混它们。然而，还必须考虑 SQL 级的语法。考虑向 member 表中插入一条记录：

```

$last = "O'Malley";
$first = "Brian";
$expiration = "2002-9-1";
$rows = $dbh->do (qq{
    INSERT member (last_name,first_name,expiration)
    VALUES('$last','$first','$expiration')
});

```

do() 发送给 MySQL 的字符串如下所示：

```

INSERT member (last_name,first_name,expiration)
VALUES('O'Malley','Brian','2002-9-1')

```

这是不合法的 SQL 语句，因为在单引号字符串内出现了单引号。在第 6 章中，我们遇到过类似的引用问题。在那里，我们使用 mysql_escape_string() 来处理这个问题。DBI 提供了一个类似的机制——在一条语句中，对想按字面使用的每个引用值，都调用 quote() 方法，并使用它的返回值。

前面的例子可编写如下：

```

$last = $dbh->quote ("O'Malley");
$first = $dbh->quote ("Brian");
$expiration = $dbh->quote ("2002-9-1");
$rows = $dbh->do (qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
});

```

现在，do() 发送给 MySQL 的字符串如下所示，具有出现在引用字符串内的可能对服务器转义的引号：

```

INSERT member (last_name,first_name,expiration)
VALUES('O\'Malley','Brian','2002-9-1')

```

请注意，在查询字符串中引用 \$last 和 \$first 时，不要增加括起来的引号；quote() 方法支持它们。如果增加了引号，则查询将出现过多的引号，如下面的例子所示：

```

$value = "paul";
$quoted_value = $dbh->quote ($value);

print "... WHERE name = $quoted_value\n";
print "... WHERE name = '$quoted_value'\n";

```

这些语句产生下面的输出：

```

... WHERE name = 'paul'
... WHERE name = ''paul''

```

7.2.7 占位符和参数约束

在前面各节中，我们通过把要插入或选择的值作为选择标准，直接放在查询字符串中构造了查询。不一定非要这样做。DBI允许在查询字符串内部放置一些称为占位符的特殊标记符，然后，在执行该查询时，将这些值代替那些标识符来使用。这样做的主要原因是提高性能，特别是在循环中反复执行某个查询的时候。

为了说明占位符如何工作，举例说明。假设学校新学期刚开始，打算清理学分薄的 student 表，然后利用包含在文件中的一列学生姓名将其初始化，使其包含新学生。不用占位符，可以如下这样删除现有表的内容，并装入新的姓名：

```
$dbh->do (qq{ DELETE FROM student } ); # delete existing rows
while (<>)                               # add new rows
{
    chomp;
    $_ = $dbh->quote ($_);
    $dbh->do (qq{ INSERT student SET name = $_ });
}
```

这样做效率很低，因为 INSERT 查询的基本格式每次都是相同的，并且在整个循环中，do() 每次都调用 prepare() 和 execute()。在进入这个循环以前，只调用一次 prepare() 来设置 INSERT 语句，并且在这个循环内部只调用 execute()，这样做效率更高一些。只调用一次 prepare()，可避免其他多次调用。DBI 允许我们这样做：

```
$dbh->do (qq{ DELETE FROM student } ); # delete existing rows
$stmt = $dbh->prepare (qq{ INSERT student SET name = ? });
while (<>)                               # add new rows
{
    chomp;
    $sth->execute ($_);
}
$stmt->finish();
```

请注意这个 INSERT 查询中的 ‘?’ 就是一个占位符。调用 execute() 时，将查询发送给服务器，传递这个值来代替占位符。一般来说，如果发现在循环内部调用了 do()，应该在循环前调用 prepare()，并在这个循环内部调用 execute() 更好一些。

有关占位符的一些注意事项：

- 在查询字符串内，不要在引号中封装占位符字符。如果这样做，不能识别为占位符。
- 不要使用 quote() 方法来指定占位符的值，否则将在插入的值中得到额外的引号。
- 在查询字符串中可以有一个以上的占位符，但是要确保占位符的标记符与传递给 execute() 的值一样多。
- 每个占位符都必须指定一个单独的值，而不是一列值。例如，不能运行这样的语句：

```
$sth = $dbh->prepare (qq{
    INSERT member last_name, first_name VALUES(?)
});
$stmt->execute ("Adams,Bill,2003-09-19");
```

必须这样：

```
$sth = $dbh->prepare (qq{
    INSERT member last_name, first_name VALUES(?,?,?)
});
$stmt->execute ("Adams","Bill","2003-09-19");
```

- 为了将 NULL 指定为占位符，应该使用 undef。
- 不要对关键字使用占位符。这样会出问题，因为占位符的值是由 quote() 自动处理的。关键字将被放在引号括起来的查询中，因此，这个查询会由于语法错误而失败。

除了在循环中提高效率以外，对于某些数据库引擎，可以从占位符的使用中获得其他的性能好处。某些引擎高速缓存了准备好的查询，以及为有效地运行这个查询所生成的计划。也就是说，如果以后这个服务器收到同样的查询，则它可以再次使用相应的计划而不用生成。查询高速缓存特别有助于复杂的 SELECT 语句，因为可能需要花费时间生成较好的执行计划。占位符提供了一个在高速缓存中寻找查询的好机会，因为它们使查询比直接在查询字符串中嵌入指定的列值来构造查询更通用。对于 MySQL，在这种方式下，占位符并不提高性能，因为没有高速缓存查询。然而，可能仍想使用占位符编写自己的查询；如果偶然将 DBI 脚本传递给支持查询高速缓存的引擎，则这个脚本比没有占位符时运行效率更高。

在查询运行时，允许在查询字符串中用占位符代替这些值。换句话说，可以参数化这个查询的“输入”。在提取行而不必将值赋给变量时，DBI 也提供一个称为参数约束的输出操作，允许通过检索自动进入这些变量的列值使“输出”参数化。

假设有一个查询，检索 member 表中的成员姓名。可以告诉 DBI 将选定列的值赋给 Perl 变量。在提取行时，变量利用相应的列值自动进行更新。下面是一个例子，说明如何将这些列约束到变量上，然后在提取循环中访问它们：

```
$sth = $dbh->prepare (qq{
    SELECT last_name, first_name, suffix
    FROM member
    ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_col (1, \$last_name);
$sth->bind_col (2, \$first_name);
$sth->bind_col (3, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch;
```

bind_col() 的每个调用都应该指定一个列号和一个希望与该列相联的变量的引用。列号从 1 开始。bind_col() 应该在 execute() 之后调用。

还有一种选择，就是单独调用 bind_col()，可以在 bind_columns() 的单个调用中传递全部变量引用：

```
$sth = $dbh->prepare (qq{
    SELECT last_name, first_name, suffix
    FROM member
    ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_columns (\$last_name, \$first_name, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch;
```

7.2.8 指定连接参数

建立服务器的连接的最直接的方法为，调用 connect() 方法时指定所有连接参数：

```
$data_source = "DBI:mysql:db_name:host_name";
$dbh->connect ($data_source, user_name, password);
```

如果遗漏连接参数，则 DBI 做下面的事情：

- 如果未定义数据源或未定义空字符串，则使用 DBI_DSN 环境变量。如果未定义用户名和口令，则使用 DBI_USER 和 DBI_PASS 环境变量（但如果它们为空字符串则不使用）。在 Windows 下，如果未定义用户名，则使用 user 变量。
- 如果遗漏了主机名，其缺省值为 localhost。
- 如果将用户名指定为 undef 或空字符串，则其缺省为 UNIX 的登录名称。在 Windows 下，用户名缺省为 ODBC。
- 如果将口令指定为 undef 或空字符串，则不传送口令。

通过将某些选项添加到字符串的初始部分，每个都在分号前面，可以在数据源中指定这些选项。例如，可以使用 mysql_read_default_file 选项来指定一个选项文件的路径名：

```
$data_source =  
"DBI:mysql:samp_db;mysql_read_default_file=/u/paul/.my.cnf";
```

当执行这个脚本时，它将从这个文件中读取连接参数。假设 /u/paul/.my.cnf 含有下面的内容：

```
[client]  
host=pit-viper.snake.net  
user=paul  
password=secret
```

然后 connect() 调用试图连接到 pit-viper-snake-net 上的 MySQL 服务器，并且用口令 secret 及用户名 paul 连接。如果想允许具有正确地设置选项文件的任何人使用您的脚本，则像这样指定数据源：

```
$data_source =  
"DBI:mysql:samp_db;mysql_read_default_file=$ENV{HOME}/.my.cnf";
```

\$ENV{HOME} 含有用户运行这个脚本的主目录的路径名，所以这个脚本使用的主机名、用户名和口令将会从每个用户自己的选项文件中抽取出来。以这种方式编写脚本，不必在这个脚本中逐字地嵌入连接参数。

还可以使用 mysql_read_default_group 选项，来指定一个选项文件组。这自动地导致读取用户的 .my.cnf 文件，并且除了 [client] 组以外，还允许读取一个指定的选项组。例如，如果在 DBI 脚本中具有指定的选项，则可以将它们列在 [dbi] 组中，然后以如下方式使用数据源值：

```
$data_source =  
"DBI:mysql:samp_db;mysql_read_default_group=dbi";
```

mysql_read_default_file 和 mysql_read_default_group 需要 MySQL 3.22.10 或更新的版本，以及 DBD::mysql 1.21.06 或更新的版本。有关指定的数据源字符串的选项的详细信息，请参阅附录 G。有关 MySQL 选项文件格式的详细信息，请参阅附录 E。

使用选项文件并不妨碍在 connect() 调用中指定连接参数（例如，如果想这个脚本作为特殊的用户来连接）。在 connect() 调用中指定的任何明确的主机名、用户名和口令值都将覆盖在选项文件中找到的连接参数。例如，想要脚本从命令行中分析 --host、--user 和 --password 选项，并使用那些值，如果给定，则优先于在选项文件中发现的任何内容。这是有用的，因为它是标准的 MySQL 客户机操作的方式。DBI 脚本将因此符合它的行为。

对于在本章中我们开发的保留在命令行中的脚本，我将使用一些标准的连接设置代码及卸载代码。我只在这里说明它一次，以便我们可以将精力集中在每个脚本的主体上，我们编写如下代码：

```

#! /usr/bin/perl

use DBI;
use strict;

# parse connection parameters from command line if given

use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive

# default parameters - all missing
my ($host_name, $user_name, $password) = (undef, undef, undef);

# GetOptions doesn't seem to allow -uuser_name form, only -u user_name?
GetOptions(
    # =s means a string argument is required after the option
    "host|h=s"      => \$host_name
    , "user|u=s"     => \$user_name
    # :s means a string argument is optional after the option
    , "password|p:s" => \$password
) or exit (1);

# solicit password if option specified without option value
if (defined ($password) && !$password)
{
    # turn off echoing but don't interfere with STDIN
    open (TTY, "/dev/tty") or die "Cannot open terminal\n";
    system ("stty -echo < /dev/tty");
    print STDERR "Enter password: ";
    chomp ($password = <TTY>);
    system ("stty echo < /dev/tty");
    close (TTY);
    print STDERR "\n";
}

# construct data source

my ($dsn) = "DBI:mysql:samp_db";
$dsn .= ":hostname=$host_name" if $host_name;
$dsn .= ";mysql_read_default_file=$ENV{HOME}/.my.cnf";

# connect to server
my (%attr) = ( RaiseError => 1 );
my ($dbh) = DBI->connect ($dsn, $user_name, $password, \%attr);

```

这个代码初始化 DBI，在命令行中查找连接参数，然后使用命令行中的或者在用户运行这个脚本的 `-.my.cnf` 文件中所找到的参数，连接到 MySQL 服务器。如果在主目录中设置 `.my.cnf` 文件，则当运行这个脚本时，不一定要输入任何连接参数（请记住，设置这种方式，以便没有其他人读取这个文件。有关的指导请参阅附录 E）。

我们脚本的最后部分也类似于从脚本到脚本；它简单地终止这个连接并退出：

```

$dbh->disconnect ();
exit (0);

```

当我们读到 Web 程序设计的部分，即 7.4 节“在 Web 应用程序中使用 DBI”时，将修改一些这个连接设置代码，但是基本的思想是类似的。

7.2.9 调试

当想调试有故障的 DBI 脚本时，通常使用两项技术，即单独使用一个或一前一后地配合使用。首先，在脚本的整个过程中编写显示语句。它允许将自己调试的输出设计为想要的方式，但必须手工地增加语句。其次，可以使用 DBI 的内建跟踪能力。这更加通用，但也更加系统，而且它在打开以后，则会自动地出现。DBI 跟踪也说明一些除此以外就无法获得的有关驱动程序的操作信息。

1. 使用显示语句调试

在 MySQL 邮件清单中，常见问题之一是：“有一个查询，当我在 mysql 中执行它时运行得很好，但是它不能在我的 DBI 脚本中工作，怎么回事？”寻找发布不同查询的 DBI 脚本和这个发问者所期望的一样是很平常的。如果在执行它之前显示查询，则可能会惊异地看到真正发送到这个服务器上的内容。假设将一个查询键入到 mysql 中（没有终止的分号）：

```
INSERT member (last_name,first_name,expiration)
VALUES("Brown","Marcia","2002-6-3")
```

然后，在 DBI 脚本中试着做相同的事情：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2002-6-3";
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($query);
```

尽管它是同样的查询，但它不能工作。不是吗？试着显示：

```
print "$query\n";
```

结果如下：

```
INSERT member (last_name,first_name,expiration)
VALUES(Brown,Marcia,2002-6-3)
```

从这个输出中，可以看到是您忘记了 VALUES() 列表中这些列值前后的引号。指定查询的正确方法如下：

```
$last = $dbh->quote ("Brown");
$first = $dbh->quote ("Marcia");
$expiration = $dbh->quote ("2002-6-3");
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
```

或者，可以使用占位符指定查询，并传递这些值，直接插入到 do() 方法中：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2002-6-3";
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES(?,?,?)
};
$rows = $dbh->do ($query, undef, $last, $first, $expiration);
```

不幸的是，当做这些的时候，使用显示语句不能看到完整查询的样子，因为直到调用 `do()` 才能估计占位符的值。当使用占位符时，跟踪可能对调试方法更有帮助。

2. 使用跟踪调试

当试图查出脚本不能正确工作的原因时，可以告知 DBI 来生成跟踪（调试）信息。跟踪级别范围从 0（关闭）到 9（最多信息）。一般来说，跟踪级别 1 和 2 是最有用的。级别 2 跟踪说明正在执行的查询文本（包括占位符替换的结果）、调用 `quote()` 的结果等等。这可能会对捕获问题有极大的帮助。

使用 `trace()` 方法，可以从独立的脚本内部控制跟踪，或者可以设置 `DBI_TRACE` 环境变量来影响所运行的所有 DBI 脚本的跟踪。

要想使用 `trace()` 调用，则传递一个跟踪级别参数，并可以有选择地再传递一个文件名。如果没有指定文件名，则所有的跟踪输出到 `STDERR` 中；否则，它就转到这个命名的文件中。一些样例如下：

```
DBI->trace (1)           Level 1 trace to STDERR
DBI->trace (2, "trace.out") Level 2 trace to "trace.out"
DBI->trace (0)           Turn trace output off
```

当调用 `DBI->trace()` 时，跟踪所有的 DBI 操作。一个更精细的方法是，可以用独立的处理级别启用跟踪。当没想好脚本中问题的位置，并对在那点出现的每件事的跟踪输出不想插手时，这是有帮助的。例如，如果特定的 `SELECT` 查询有问题，则可以跟踪与这个查询相关的语句句柄：

```
$sth = $dbh->prepare (qq{ SELECT ... }); # create the statement handle
$sth->trace (1);                        # enable tracing on the statement
$sth->execute ();
```

如果对任何 `trace()` 调用指定一个文件名参数，则无论对 DBI 作为整体还是单独的句柄，所有的跟踪输出都要到那个文件中。

要想对运行的所有 DBI 脚本全部都打开跟踪，则从命令解释程序中设置 `DBI_TRACE` 环境变量。它的语法取决于使用的命令解释程序：

```
% setenv DBI_TRACE value      For csh, tcsh
$ DBI_TRACE=value             For sh, ksh, bash
$ export DBI_TRACE
C:\> set DBI_TRACE=value      For Windows
```

`value` 的模式和所有命令解释程序的模式一样：数字 `n` 表示在级别 `n` 打开跟踪到 `STDERR` 中；文件名打开级别 2 跟踪到这个命名的文件，或 `n=file_name` 打开级别 `n` 跟踪到这个命名的文件中。下面的样例使用了 `csh` 语法：

```
% setenv DBI_TRACE 1          级别1跟踪到 STDERR 中
% setenv DBI_TRACE 1=trace.out 级别1跟踪到 "trace.out" 中
% setenv DBI_TRACE trace.out   级别2跟踪到 "trace.out" 中
```

如果打开跟踪到命令解释程序中的文件，则确保一旦解决了这个问题，就将它关闭。将调试输出增加到这个跟踪文件中，而不用重写它，所以如果不小心，则这个文件可能变得非常大。极其不好的想法是在命令解释程序的启动文件（如 `.cshrc`、`.login` 或 `.profile`）中定义 `DBI_TRACE`！在 UNIX 下，可以使用下面两个命令（`csh` 语法）之一关闭跟踪：

```
% setenv DBI_TRACE 0
% unsetenv DBI_TRACE
```

对于 sh、ksh 或 bash, 这样做：

```
$ DBI_TRACE=0
$ export DBI_TRACE
```

在 Windows 操作系统中, 可以使用下面两个命令之一关闭跟踪：

```
C:\> unset DBI_TRACE
C:\> set DBI_TRACE=0
```

7.2.10 使用结果集元数据

可以使用 DBI 来获得访问结果集元数据——也就是有关由查询选择行的描述信息。访问与结果集生成的查询所相关的语句句柄的属性来获得这个信息。提供这些属性中有一些是作为可用于横跨所有数据库驱动程序的标准 DBI 属性（如 NUM_OF_FIELDS, 结果集中列的数量）。另外一些是 MySQL 特定的, 由 DBD::mysql 所提供的 DBI 的 MySQL 驱动程序。这些属性, 如 mysql_max_length 告知了每列值的最大宽度, 不能用于其他数据库引擎。要想使用任何 MySQL 特定的属性, 都必须冒着使脚本不可移植到其他数据库的危险。另一方面, 它们可以使它更容易地获得想要的信息。

必须在适当时候请求元数据。一般来说, 直到调用 prepare() 和 execute() 之后, 结果集属性才能用于 SELECT 语句。除此之外, 在调用 finish() 之后, 属性可能变为无效。

让我们来看看如何使用 MySQL 的一个元数据属性 mysql_max_length, 与保留查询列名的 DBI 级别的 NAME 属性一起使用。我们可以将这些属性提供的信息合并起来, 编写一个脚本 box_out, 它以交互模式运行 mysql 客户机程序时获得的相同边框风格, 从 SELECT 查询产生输出。box_out 的主体如下（可以用任何其他的语句替换 SELECT 语句；编写输出的例程独立于特定的查询）：

```
my ($sth) = $dbh->prepare (qq{
    SELECT last_name, first_name, city, state
    FROM president ORDER BY last_name, first_name
});
$sth->execute (); # attributes should be available after this call

# actual maximum widths of column values in result set
my (@wid) = @{$sth->{mysql_max_length}};
# number of columns
my ($ncols) = scalar (@wid);

# adjust column widths if column headings are wider than data values
for (my $i = 0; $i < $ncols; $i++)
{
    my ($name_wid) = length ($sth->{NAME}->[$i]);

    $wid[$i] = $name_wid if $wid[$i] < $name_wid;
}

# print output
print_dashes (\@wid, $ncols);          # row of dashes
print_row ($sth->{NAME}, \@wid, $ncols); # column headings
print_dashes (\@wid, $ncols);          # row of dashes
while (my $ary_ref = $sth->fetch)
{
```

```

    print_row ($ary_ref, \@wid, $ncols);    # row data values
}
print_dashes (\@wid, $ncols);            # row of dashes
$sth->finish ();

```

用execute() 将这个查询初始化之后, 我们获得了所需的元数据。\$sth->{NAME} 和 \$sth->{mysql_max_length} 给出了列名和每列值的最大宽度。为了在这个查询中为列命名, 每个属性值都引用了一个数组, 这个数组含有结果集每列中的一个值。

剩余的计算非常类似于在第6章中开发的客户机程序5中所使用的那些内容。例如, 为避免偏离输出, 如果列的名比该列中任何数据值都宽, 则我们要向上调整列的宽度值。

输出函数 print_dashes() 和 print_row() 代码编写如下, 它们也类似于客户机程序5中相应的代码:

```

sub print_dashes
{
    my ($wid_ary_ref) = shift; # column widths
    my ($cols) = shift;       # number of columns

    print "+";
    for (my $i = 0; $i < $cols; $i++)
    {
        print "-" x ($wid_ary_ref->[$i]+2) . "+";
    }
    print "\n";
}

# print row of data. (doesn't right-align numeric columns)

sub print_row
{
    my ($val_ary_ref) = shift; # column values
    my ($wid_ary_ref) = shift; # column widths
    my ($cols) = shift;       # number of columns

    print "|";
    for (my $i = 0; $i < $cols; $i++)
    {
        printf "%-*s |", $wid_ary_ref->[$i],
            defined ($val_ary_ref->[$i]) ? $val_ary_ref->[$i] : "NULL";
    }
    print "\n";
}

```

box_out 的输出如下:

last_name	first_name	city	state
Adams	John	Braintree	MA
Adams	John Quincy	Braintree	MA
Arthur	Chester A.	Fairfield	VT
Buchanan	James	Mercersburg	PA
Bush	George W.	Milton	MA

我们的下一个脚本使用了列元数据来产生不同格式的输出。这个脚本 show_member, 允许快速浏览历史同盟成员项目, 而不用输入任何查询。给出成员的姓, 它就这样显示所选择的项目:

```
% show_member artel
```

```

last_name: Artel
first_name: Mike
suffix:
expiration: 2003-04-16
email: mike_artel@venus.org
street: 4264 Lovering Rd.
city: Miami
state: FL
zip: 12777
phone: 075-961-0712
interests: Civil Rights, Education, Revolutionary War
member_id: 63

```

使用成员资格号码，或者使用与若干姓相匹配的模式也可以调用 `show_members`。下面的命令说明成员号码为23的项目，和以字母“C”开始的姓的成员项：

```

% show_member 23
% show_member c%

```

`show_member` 脚本的主体如下所示。它使用了 `NAME` 属性来确定输出的每行所使用的标号和 `NUM_OF_FIELDS` 属性，找出这个结果集含有的列数：

```

my ($count) = 0;    # number of entries printed so far
my (@label) = ();   # column label array
my ($label_wid) = 0;

while (@ARGV)       # run query for each argument on command line
{
    my ($arg) = shift (@ARGV);
    my ($sth, $clause, $address);

    # default is last name search; do id search if argument is numeric
    $clause = "last_name LIKE " . $dbh->quote ($arg);
    $clause = "member_id = " . $dbh->quote ($arg) if $arg =~ /\d/;

    # issue query
    $sth = $dbh->prepare (qq{
        SELECT * FROM member
        WHERE $clause
        ORDER BY last_name, first_name
    });
    $sth->execute ();

    # get column names to use for labels and
    # determine max column name width for formatting
    # (only do this the first time through the loop, though)
    if ($label_wid == 0)
    {
        @label = @{$sth->{NAME}};
        foreach my $label (@label)
        {
            $label_wid = length ($label) if $label_wid < length ($label);
        }
    }

    # read and print query results, then clean up
    while (my @ary = $sth->fetchrow_array ())
    {
        # print newline before 2nd and subsequent entries
        print "\n" if ++$count > 1;
        foreach (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)

```

```
{
    # print label, and value if there is one
    printf "%-*s", $label_wid+1, $label[$i] . ":";
    print " " . $ary[$i] if $ary[$i];
    print "\n";
}
}
$sth->finish ();
}
```

无论区域是什么，`show_member` 的目的都是说明一个项目的全部内容。通过使用 `SELECT *` 来检索所有的列和 `NAME` 属性来看看它们是什么，即使从 `member` 表中增加或删除列，这个脚本也会工作而不用做修改。

如果不检索任何行就想知道一个表含有哪些列，则可以发布下面这条查询：

```
SELECT * FROM tbl_name WHERE 1 = 0
```

以正常方式调用 `prepare()` 和 `execute()` 之后，可以从 `@{$sth->{NAME}}` 中得到列名。然而，请注意，尽管使用“空”查询的这个小心窍门可以在 MySQL 下运行，但是它不可移植，而且并不是对所有的数据库引擎都可以工作的。

有关 DBI 和 `DBD::mysql` 所提供属性的详细信息，请参见附录 G。它完全可以使您确定是想通过避免 MySQL 特定的属性而为可移植性花费努力，还是在可移植性的开销方面利用它们。

7.3 运行 DBI

在这里，已经看到许多涉及 DBI 程序设计的概念，所以让我们继续做一些样例数据库能处理事情。最初，第 1 章简述了我们的目标。本章通过编写 DBI 脚本，我们将处理的那些问题在这里列出。

对于学分保存方案，我们想能够检索任何给定的测验或测试的分数。

对于历史联盟，我们想做下面的事情：

- 以不同格式产生成员目录。我们想在年度宴会程序中，以可以用于生成显示目录的格式使用一个只有名称的列表。
- 寻找不久就要更新其成员资格的 League 成员，然后发送电子邮件通知他们。
- 编辑成员项目（毕竟，在更新成员资格时，我们将要更新他们的终止日期）。
- 寻找分享共同兴趣的成员。
- 使这个目录联机。

对于这样一些任务，我们将编写从命令行上运行的脚本。其他任务，我们将在 7.4 节“在 Web 应用程序中使用 DBI”中创建脚本，可以与 Web 服务器配合使用。在本章的最后，我们将仍有许多有待完成的目标。将在第 8 章“PHP API”中，完成剩余的目标。

7.3.1 生成历史同盟目录

我们的目标之一是能以不同格式产生历史同盟目录的信息。我们将生成的最简单格式是一个年度宴会程序的成员名列表。那可能是一个简单的无格式文本列表。它将成为创建这个程序的一部分较大文档，所以，我们所需要的就是可以粘贴到文档中的一些内容。

对于可显示的目录，则需要一种比无格式文本更好的表示方法，原因是我们想把一些内

容更精细地格式化。这里一个合理的选择为 RTF (丰富的文本格式 Rich Text Format)，它是由 Microsoft 开发的一种格式，可以由许多字处理程序来识别。当然，Word 就是这种程序之一，但是许多其他的软件，如 WordPerfect 和 AppleWork 也是可以识别的。不同的字处理程序对 RTF 的支持程度也有所不同，但是我们将使用由即使对最低级别 RTF 都确信的任何字处理程序所支持的全部 RTF 规定的一个基本子集。

生成宴会列表和 RTF 目录格式的过程本质上是一样的：发布查询来检索这些项目，然后运行将每个项目提取和格式化的循环。给出了基本的相似之处，就能很好地避免编写两个分开的脚本。所以，我们编写一个单独的脚本 gen_dir，它可以以不同的格式从这个目录生成输出。我们可以这样组织这个脚本：

- 1) 在编写出项目内容之前，完成这个输出格式可能需要的任何初始化。宴会程序成员列表不需要任何特殊的初始化，但是我们需要为这个 RTF 版本编写一些初始的控制语言。
- 2) 提取和显示每个项目，将我们要输出的类型适当地格式化。
- 3) 处理完所有的项目之后，还要完成任何必需的清除和终止。除了这个 RTF 版本需要的一些关闭控制语言以外，宴会列表不需要特殊的处理。

将来，我们可能想使用这个脚本以其他格式编写输出，所以我们通过设置“转换盒”——每个输出格式都有一个元素的散列，使它成为可扩展的。每个元素都指定对给定格式生成适当输出的函数：初始化函数、编写项目函数和清除函数如下所示：

```
# switchbox containing formatting functions for each output format
my (%switchbox) =
(
    "banquet" =>
    {
        # functions for banquet list
        "init"    => undef,          # no initialization needed
        "entry"   => \&format_banquet_entry,
        "cleanup" => undef          # no cleanup needed
    },
    "rtf" =>
    {
        # functions for RTF format
        "init"    => \&rtf_init,
        "entry"   => \&format_rtf_entry,
        "cleanup" => \&rtf_cleanup
    }
);
```

由一个格式名（在这种情况下的“banquet”和“rtf”）标识转换盒的每个元素。我们将编写这个脚本，以便在运行它时可以在命令行中指定想要的格式：

```
% gen_dir banquet
% gen_dir rtf
```

通过以这种方式设置转换盒，我们可以很容易地增加新格式的性能：

- 1) 编写三个格式化函数。
- 2) 向转换盒增加一个指向那些函数的新元素。
- 3) 为了以新的格式产生输出，调用 gen_dir，并在命令行中指定这个格式名。

按照命令行中的第一个参数所选择的适当转换盒项目的代码如下所示。它是由于输出格式的名称为 %switchbox 散列中关键字。如果在转换盒中不存在这样的关键字，则这个格式是无效的。不需要这个代码中的硬连线格式；如果向转换盒增加新的格式，则自动地检测它。

如果在命令行中没有指定格式名，或者指定了一个无效的名称，则这个脚本产生错误消息，并显示一系列允许的名称：

```
# make sure one argument was specified on the command line
@ARGV == 1
    or die "Usage: gen_dir format_type\nAllowable formats: "
        . join (" ", sort (keys (%switchbox))) . "\n";

# determine proper switchbox entry from argument on command line
# if no entry was found, the format type was invalid
my ($func_hashref) = $switchbox{$ARGV[0]};

defined ($func_hashref)
    or die "Unknown format: $ARGV[0]\nAllowable formats: "
        . join (" ", sort (keys (%switchbox))) . "\n";
```

如果在命令行指定了一个有效的格式名，则前述的代码设置 `$func_hashref`。它的值将是指向选择了格式输出的编写函数的散列引用。然后我们可以运行这个选择项目的查询。之后，我们调用初始化函数、提取和显示这些项目，并激活清除函数：

```
# issue query
my ($sth) = $dbh->prepare (qq{
    SELECT * FROM member ORDER BY last_name, first_name
});
$sth->execute ();

# invoke initialization function if there is one
&{$func_hashref->{init}} if defined ($func_hashref->{init});

# fetch and print entries if there is an entry formatting function
if (defined ($func_hashref->{entry}))
{
    while (my $entry_ref = $sth->fetchrow_hashref ("NAME_lc"))
    {
        # pass entry reference to formatting function
        &{$func_hashref->{entry}} ($entry_ref);
    }
}
$sth->finish ();

# invoke cleanup function if there is one
&{$func_hashref->{cleanup}} if defined ($func_hashref->{cleanup});
```

因为某种原因，提取项目的循环使用了 `fetchrow_hashref()`。如果这个循环提取数组，则这个格式化函数必须知道列的次序。它可能通过访问 `$sth->{NAME}` 属性（它含有返回次序的列名）来得到，但为什么烦扰呢？通过使用散列引用，格式化函数将只能命名那些想使用 `$entry_ref->{col_name}` 的列值。那样效率就非常低，但它容易做到，并可用于想生成的任何格式，因为我们知道我们需要的任何域都在散列中。

剩余的工作就是为每种输出格式编写这些函数（也就是说，通过转换盒项目为这些函数命名）。

1. 生成宴会程序成员列表

对于这种输出格式，我们只想要成员的姓名。不需要初始化或清除调用。只需要一个项目格式化函数：

```
sub format_banquet_entry
```

```

{
my ($entry_ref) = shift;
my ($name);

$name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
if ($entry_ref->{suffix})      # there is a name suffix
{
    # no comma for suffixes of I, II, III, etc.
    $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
    $name .= " " . $entry_ref->{suffix} if $entry_ref->{suffix};
}
print "$name\n";
}

```

format_banquet_entry() 的参数是行的列值的散列引用。这个函数将名和姓连在一起，加上可能出现的任何后缀。这里的窍门是如“Jr.”或“Sr.”后缀的前面应该有一个逗号或空格，但是如“II”或“III”后缀的前面只能为一个空格：

```

Michael Alvis IV
Clarence Elgar, Jr.
Bill Matthews, Sr.
Mark York II

```

因为字母‘I’、‘V’和‘X’覆盖了所有生成的数字，从第1到第39，所以我们可以使用下面的测试来确定是否增加一个逗号：

```
$name .= "," unless $hash_ref->{suffix} =~ /^[IVX]+$/;
```

和名称放在一起的 format_banquet_entry() 的代码也是这个目录的 RTF 版本将需要的一些内容。然而，并不是复制 format_rtf_entry() 中的代码，让我们将它填入函数中：

```

sub format_name
{
my ($entry_ref) = shift;
my ($name);

$name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
if ($entry_ref->{suffix})      # there is a name suffix
{
    # no comma for suffixes of I, II, III, etc.
    $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
    $name .= " " . $entry_ref->{suffix} if $entry_ref->{suffix};
}
return ($name);
}

```

将确定名称的字符串放在 format_name() 函数中，将把 format_banquet_entry() 函数减少到几乎没有：

```

sub format_banquet_entry
{
    printf "%s\n", format_name ($_[0]);
}

```

2. 生成显示格式的目录

生成这个目录的 RTF 版本比生成宴会程序成员列表更要棘手一些。首先，我们需要从每个项目中显示更多的信息。其次，我们需要用每个项目产生一些 RTF 控制语言来完成我们想要的作用。RTF 文档的最小框架是这样的：

```

{\rtf0
{\fonttbl {\f0 Times;}}
\plain \f0 \fs24

```

```
...document content goes here...
}
```

这个文档用花括号 ‘ { ’ 和 ‘ } ’ 作为开始和结束。RTF 关键字用反斜线符号开始，并且文档的第一个关键字必须为 \rtfn, n 为这个文档对应的 RTF 规定的版本号。如果按我们的目的，0 就比较合适。

在这个文档的内部，我们指定字体表来说明这些项目所使用的字体。字体表信息列在组中，由含有前导的 \fonttbl 关键字和一些字体信息的花括号组成。在框架中说明的这个字体表把字体号 0 定义为 Times（我们只需要一个字体，但是如果显示得更好一些，可以使用多种字体）。

下面的一些指示设置了缺省格式风格：\plain 选择无格式的格式，\f0 选择字体 0（我们已经在字体表中定义为 Times），\fs24 设置字体大小为 12 个点阵（\fs 后面的数量表示半个点阵的大小）。设置页边空白并不是必需的；大多数的字处理程序将提供合理的缺省值。

要想得到一个非常简单的方法，可以将每个项目显示为一系列的行，每行上都有一个标号。如果对应于特定输出行的信息缺失，则忽略这个行（例如，没有电子邮件地址的成员没有显示“Email:”行）。一些行（如“Address:”行）由多个列（街道、城市、州、邮政编码）中的信息构成，所以这个脚本必须能够处理缺失值的各种组合。这里是我们将使用的输出格式的样例：

```
Name: Mike Artel
Address: 4264 Lovering Rd., Miami, FL 12777
Telephone: 075-961-0712
Email: mike_artel@venus.org
Interests: Civil Rights, Education, Revolutionary War
```

对于显示的格式化项目，RTF 的表示方法如下所示：

```
\b Name: Mike Artel\b0\par
Address: 4264 Lovering Rd., Miami, FL 12777\par
Telephone: 075-961-0712\par
Email: mike_artel@venus.org\par
Interests: Civil Rights, Education, Revolutionary War\par
```

要想使“Name:”行为粗体，则在它的前面加 \b（后面有个空格）来打开粗体，并用 \b0 来关闭粗体。每行在末端都有一个段标记符（\par）来告诉字处理程序移到下一行——没有太复杂的事情。

初始化函数产生前导 RTF 控制语言（请注意，两个反斜线符号获得输出中的一个反斜线符号）：

```
sub rtf_init
{
    print "{\\rtf0\n";
    print "{\\fonttbl {\\f0 Times;}}\n";
    print "\\plain \\f0 \\fs24\n";
}
```

类似地，清除函数产生终止控制语言（并不太多！）：

```
sub rtf_cleanup
{
    print "}\n";
}
```

真正的工作与格式化这个项目有关，即使这个任务相对简单。主要复杂点是将地址字符

串格式化，并确定应该显示哪个输出行：

```
sub format_rtf_entry
{
    my ($entry_ref) = shift;
    my ($address);

    printf "\\b Name: %s\\b0\\par\\n", format_name ($entry_ref);
    $address = "";
    $address .= $entry_ref->{street} if $entry_ref->{street};
    $address .= ", " . $entry_ref->{city} if $entry_ref->{city};
    $address .= ", " . $entry_ref->{state} if $entry_ref->{state};
    $address .= " " . $entry_ref->{zip} if $entry_ref->{zip};
    print "Address: $address\\par\\n" if $address;
    print "Telephone: $entry_ref->{phone}\\par\\n" if $entry_ref->{phone};
    print "Email: $entry_ref->{email}\\par\\n" if $entry_ref->{email};
    print "Interests: $entry_ref->{interests}\\par\\n" if $entry_ref->{interests};
    print "\\par\\n";
}
```

当然，不用限于这种特殊的格式化风格。可以更改如何显示任何域的方法，所以通过简单地更改 `format_rtf_entry()`，可以几乎任意地更改显示的目录。用它原始格式的目录（一个字处理文档），是多么不容易做的事情！

`gen_dir` 脚本现在完成了。通过运行以下这些命令，我们可以以任意一种输出格式生成这个目录：

```
% gen_dir banquet > names.txt
% gen_dir rtf > directory.rtf
```

在 Windows 中，我可以运行 `gen_dir`，则这些文件准备从基于 Windows 字处理程序的内部使用。在 UNIX 中，我可就以运行上面那些命令，然后将这些输出文件以邮件形式发给自己作为附件，以便可以从我的 Macintosh 中获取它们，并将它们加载到字处理程序中。我偶尔使用 `mutt` 邮寄程序，它允许使用 `-a` 选项从命令行指定附件。可以如下发送给自己一个具有这两个附加文件的消息：

```
% mutt -a names.txt -a directory.rtf paul@snake.net
```

其他邮寄程序可能也允许创建附件。或者，可以以其他意思传输这些文件，如 FTP。无论如何，在这些文件被放到想放的地方之后，读取这个名称列表，并将它粘贴到年度程序文档，或者在可识别 RTF 的任何字处理程序中读取 RTF 文件，这都是较容易的。DBI 使我们从 MySQL 中抽取想要的信息很容易，Perl 的文本处理能力使我们将这些信息放在指定的格式中很容易。MySQL 不提供信息输出的任何特殊方式，但没有关系，因为将 MySQL 的数据库处理能力集成到如 Perl 的语言中并不费力，而这些语言具有极好的文本处理能力。

7.3.2 发送成员资格更新通知

当作为字处理文档维护历史同盟目录时，确定需要通知哪个成员其成员资格应该更新，这是件耗时间并且容易出现错误的事情。既然我们在数据库中有信息，那么让我们看看如何自动地处理更新通知。我们想标识需要经过电子邮件更新的成员，这样我们就不必通过电话或邮件与他们联系了。

我们需要做的事情就是确定哪个成员在某些天以内快到更新的时间了。这个的查询涉及一个相对简单的日期计算：

```
SELECT ... FROM member
WHERE expiration < DATE_ADD(CURRENT_DATE, INTERVAL cutoff DAY)
```

cutoff 表示我们同意的可允许误差的天数。这个查询选择在几天之内快到更新时间的成员项目。作为特殊情况，终止点值为 0，寻找终止日期已过的成员（也就是说，实际上已经终止了的那些成员）。

我们标识了限制通知的这些记录之后，我们对它们应该怎么办呢？一个选择是直接从同样的脚本中发送邮件，但是，首先审阅不发送任何消息的列表可能有用。由于这个原因，我们将使用一个两阶段的方法：

- 阶段1：运行脚本 need_renewal 来标识需要更新的成员。可检查这个列表，或者可以使用它作为将更新通知发送到第 2 阶段的输入。
- 阶段2：运行脚本 renewal_notify，它通过电子邮件向成员发送“请更新”的通知。这个脚本应该通知您不具有电子邮件地址的成员，以便可以用其他方式与他们联系。

在此任务的第一部分中，need_renewal 脚本必须标识哪个成员需要更新。它的操作如下所示：

```
# default cutoff is 30 days; reset if numeric argument given
my ($cutoff) = 30;
$cutoff = shift (@ARGV) if @ARGV && $ARGV[0] =~ /^d+$/;

warn "Using cutoff of $cutoff days\n";

my ($sth) = $dbh->prepare (qq{
    SELECT
        member_id, email, last_name, first_name, expiration,
        TO_DAYS(expiration) - TO_DAYS(CURRENT_DATE) AS days
    FROM member
    WHERE expiration < DATE_ADD(CURRENT_DATE, INTERVAL ? DAY)
    ORDER BY expiration, last_name, first_name
});
$sth->execute ($cutoff);    # pass cutoff as placeholder value

while (my $hash_ref = $sth->fetchrow_hashref ())
{
    print join ("\t",
        $hash_ref->{member_id},
        $hash_ref->{email},
        $hash_ref->{last_name},
        $hash_ref->{first_name},
        $hash_ref->{expiration},
        $hash_ref->{days} . " days")
        . "\n";
}
$sth->finish ();
```

need_renewal 脚本的输出如下所示（因为是针对当前日期确定的结果，而您读这本书的时间和书写它的时间将是不同的，所以将获得不同的输出）：

```
89 g.steve@pluto.com    Garner Steve  1999-08-03  -32 days
18 york_mark@earth.com  York Mark     1999-08-24  -11 days
82 iohn edwards@venus.org Edwards John   1999-09-12   8 days
```

可以观察到，处于负数天数的那些成员资格需要更新。负数意味着我们已经过期了（当手工地维护记录时，就可能发生这种情况；有些人从缝隙中滑掉了。既然我们在数据库中有了一些信息，那么我们要寻找在前面丢失的几个人）！

更新通知任务的第二部分涉及了通过电子邮件发送通知的脚本 `renewal_notify`。要想使 `renewal_notify` 更容易使用，则我们可以使它支持三类命令行参数：成员关系 ID 号码，电子邮件地址和文件名。数值的参数表示成员资格 ID 值，带有字符 '@' 的参数表示电子邮件的地址。其他任何事情都解释为应该读取的文件名，以便找到他们的 ID 号码或电子邮件地址，可以直接在命令行中这样做，或者通过将它们在文件中列出来去做（特别是，可以使用 `need_renewal` 的输出作为 `renewal_notify` 的输入）。

对于要发送通知的每个成员，此脚本查找相应的 `member` 表项目，抽取电子邮件地址，并向那个地址发送一条消息。如果此项中没有电子邮件地址，则 `renewal_notify` 生成一条消息，通知您需要以一些其他方式与这些成员联系。

要想发送电子邮件，`renewal_notify` 打开与 `sendmail` 程序的管道，并将这封邮件推入此管道中（在 Windows 下不能这样操作，Windows 中没有 `sendmail`。可能需要寻找发送邮件的模块来代替它使用）。在此脚本开头附近，将 `sendmail` 的路径名设置为参数。可能需要更改该路径，因为 `sendmail` 的位置随系统的变化而变化：

```
# change path to match your system
my ($sendmail) = "/usr/lib/sendmail -t -oi";
```

主要参数处理循环的操作如下所示。如果在命令行没有指定参数，则我们读取标准的输出作为输入。否则，我们通过将参数传递给 `interpret_argument()`，将它分类为 ID 号、电子邮件地址或者文件名来处理每个参数：

```
if (@ARGV == 0)      # no arguments, read STDIN for values
{
    read_file (\*STDIN);
}
else
{
    while (my $arg = shift (@ARGV))
    {
        # interpret argument, with filename recursion
        interpret_argument ($arg, 1);
    }
}
```

函数 `read_file()` 读取了文件的内容（假设已经打开），并查看每行的第一个域（如果我们 将 `need_renewal` 的输出作为 `renewal_notify` 的输入，则每行都有若干域，但是我们只想查看第一个域）。

```
sub read_file
{
    my ($fh) = shift;
    my ($arg);

    while (defined ($arg = <$fh>))
    {
        # strip off everything past column 1, including newline
        $arg =~ s/\s.*//s;
        # interpret argument, without filename recursion
        interpret_argument ($arg, 0);
    }
}
```

`interpret_argument()` 函数将每个参数分类，以便确定它是 ID 号码、电子邮件地址还是文

件名。对于 ID 号码和电子邮件地址，它查找适当的成员项目，并将它传递给 `notify_member()`。我们必须注意由电子邮件所指定的成员。两个成员具有同样的地址是可能的（例如，丈夫和妻子），并且我们不想将一条消息发送给不能用这条消息的人。为了避免这一点，我们查找了与电子邮件地址相对应的成员的 ID 号码，来确保内容的正确。如果此地址和一个以上的 ID 号码匹配，则它是不确定的，我们在显示一条警告消息后忽略它。

如果参数看起来不像 ID 号码或电子邮件地址，则将它作为文件名读取为进一步的输入。在这里，我们也必须小心——为了避免无穷循环的可能性，如果我们已经读取一个文件，则我们不想再读取文件：

```
sub interpret_argument
{
  my ($arg, $recurse) = @_;
  my ($query, $ary_ref);

  if ($arg =~ /^(\d+)/)          # numeric membership ID
  {
    notify_member ($arg);
  }
  elsif ($arg =~ /\@/)          # email address
  {
    # get member_id associated with address
    # (there should be exactly one)
    $query = qq{ SELECT member_id FROM member WHERE email = ? };
    $ary_ref = $dbh->selectcol_arrayref ($query, undef, $arg);
    if (scalar (@{$ary_ref}) == 0)
    {
      warn "Email address $arg matches no entry: ignored\n";
    }
    elsif (scalar (@{$ary_ref}) > 1)
    {
      warn "Email address $arg matches multiple entries: ignored\n";
    }
    else
    {
      notify_member ($ary_ref->[0]);
    }
  }
  else                          # filename
  {
    if (!$recurse)
    {
      warn "filename $arg inside file: ignored\n";
    }
    else
    {
      open (IN, $arg) or die "Cannot open $arg: $!\n";
      read_file (\*IN);
      close (IN);
    }
  }
}
```

实际上，发送更新通知的 `notify_member()` 函数的代码如下所示。如果得出这个成员没有电子邮件地址，则什么也不做，但是 `notify_member()` 显示一条警告消息，以便知道需要以其他某种方式与该成员联系。可以调用具有这条消息中所显示的这个成员资格 ID 号码的 `show_member`，来查看全部项目——例如，找出这个成员的电话号码和通信地址。

```
# notify member that membership will soon be in arrears

sub notify_member
{
    my ($member_id) = shift;
    my ($query, $sth, $entry_ref, @col_name);

    warn "Notifying $member_id...\n";
    $query = qq{ SELECT * FROM member WHERE member_id = ? };
    $sth = $dbh->prepare ($query);
    $sth->execute ($member_id);
    @col_name = @{$sth->{NAME}};
    $entry_ref = $sth->fetchrow_hashref ();
    $sth->finish ();
    if (!$entry_ref)          # no member found!
    {
        warn "NO ENTRY found for member $member_id!\n";
        return;
    }
    open (OUT, "| $sendmail") or die "Cannot open mailer\n";
    print OUT <<EOF;
    To: $entry_ref->{email}
    Subject: Your USHL membership is in need of renewal
    Greetings. Your membership in the US Historical League is
    due to expire soon. We hope that you'll take a few minutes to
    contact the League office to renew your membership. The
    contents of your member entry are shown below. Please note
    particularly the expiration date.

    Thank you.

    EOF
    foreach my $col_name (@col_name)
    {
        printf OUT "%s: %s\n", $col_name, $entry_ref->{$col_name};
    }
    close (OUT);
}
```

用它可能获得更好的内容——例如，通过向 member 表中增加一列来记录最近更新的提示是何时发送出去的。这样做将有助于避免过于频繁地发送通知。实际上，我们只需假设不存在大约每月运行一次以上的程序。

现在运行这两个脚本，从而可以这样使用它们：

```
% need_renewal > junk
% (看一看 junk, 检查它是否合理)
% renewal_notify junk
```

要想通知单个的成员，可以通过 ID 号码或电子邮件地址指定它们：

```
% need_renewal 18 g.steve@pluto.com
```

7.3.3 历史同盟成员项目编辑

我们开始发送更新通知之后，假设我们通知的一些人将更新他们的成员资格是个安全的措施。当这种情况发生时，我们将需要一种更新其所具有的新的终止日期项的方法。下一章中，我们将开发一种方法，在 Web 浏览器上编辑成员记录，但是在这里，我们将建立一个命令行脚本 edit_member，允许用提示项的各部分新值的方法来更新项目。其操作如下：

- 如果在命令行上无参数调用，则 edit_member 假设您想输入一个新的号码，提示放在成员项目中的初始信息，并创建新的项目。

■如果在命令行上调用时带有成员 ID 号码,则edit_member 查找这个项目的已有内容,然后提示更新每一列。如果输入一列的值,则其替换当前的值。如果按 Enter 键,这列并不更改(如果不知道成员的 ID 号码,可以运行 show_member last_name 来查找其内容)。

如果只想更新成员的终止日期,则允许编辑全部项目的这种方式可能是不必要的过度行动。另一方面,类似这样的脚本也提供了一种简单的通用目的方式,来更新一个项目的任何部分而不必了解 SQL 的任何知识(一种特殊的情况为 edit_member 不允许更改 member_id 域,因为当创建一个项目时,自动地分配这个域,并且在以后不能更改)。

edit_member 需要了解的第一件事为 member 表中这些列的名称:

```
# get member table column names
my ($sth) = $dbh->prepare (qq{ SELECT * FROM member WHERE 1 = 0 });
$sth->execute ();
my (@col_name) = @{$sth->{NAME}};
$sth->finish ();
```

然后我们可以输入主体循环:

```
if (@ARGV == 0) # if no arguments, create new entry
{
    new_member (\@col_name);
}
else           # otherwise edit entries using arguments as member IDs
{
    my (@id);

    # save @ARGV, then empty it so that reads from STDIN
    # don't use the arguments as filenames
    @id = @ARGV;
    @ARGV = ();
    # for each ID value, look up the entry, then edit it
    while (my $id = shift (@id))
    {
        my ($entry_ref);

        $sth = $dbh->prepare (qq{
            SELECT * FROM member WHERE member_id = ?
        });
        $sth->execute ($id);
        $entry_ref = $sth->fetchrow_hashref ();
        $sth->finish ();
        if (!$entry_ref)
        {
            warn "No member with member ID = $id\n";
            next;
        }
        edit_member (\@col_name, $entry_ref);
    }
}
```

创建新成员项目的代码如下所示。它请求每个 member 表列,然后发布一条 INSERT 语句以增加一条新记录:

```
# create new member entry

sub new_member
{
    my ($col_name_ref) = shift;
    my ($entry_ref);
    my ($col_val, $query, $delim);
```

```

return unless prompt ("Create new entry? ") =~ /^y/i;
# prompt for new values; user types in new value,
# "null" to enter a NULL value, "exit" to exit
# early.
foreach my $col_name (@{$col_name_ref})
{
    next if $col_name eq "member_id"; # skip key field
    $col_val = col_prompt ($col_name, "", 0);
    next if $col_val eq ""; # user pressed Enter
    return if $col_val =~ /^exit$/i; # early exit
    $col_val = undef if $col_val =~ /^null$/i;
    $entry_ref->{$col_name} = $col_val;
}
# show values, ask for confirmation before inserting
show_member ($col_name_ref, $entry_ref);
return unless prompt ("\nInsert this entry? ") =~ /^y/i;

# construct an INSERT query, then issue it.
$query = "INSERT INTO member";
$delim = " SET "; # put "SET" before first column, "," before others
foreach my $col_name (@{$col_name_ref})
{
    # only specify values for columns that were given one
    next if !defined ($entry_ref->{$col_name});
    # quote() quotes undef as the word NULL (without quotes),
    # which is what we want.
    $query .= sprintf ("%s %s=%s", $delim, $col_name,
                        $dbh->quote ($entry_ref->{$col_name}));
    $delim = ",";
}
warn "Warning: entry not inserted?\n"
    unless $dbh->do ($query) == 1;
}

```

new_member() 所用的提示例程如下所示：

ask a question, prompt for an answer

```

sub prompt
{
    my ($str) = shift;

    print STDERR $str;
    chomp ($str = <STDIN>);
    return ($str);
}

# prompt for a column value; show current value in prompt if
# $show_current is true

sub col_prompt
{
    my ($name, $val, $show_current) = @_;
    my ($prompt, $str);

    $prompt = $name;
    $prompt .= " [$val]" if $show_current;
    $prompt .= ": ";
    print STDERR $prompt;
    chomp ($str = <STDIN>);
    return ($str ? $str : $val);
}

```

col_prompt() 带有 \$show_current 参数的原因是，当这个脚本用于更新项目时，我们也对已有成员项目请求的列值使用这个函数。当创建新的项目时，\$show_current 将为 0，因为当

前没有值可以显示。在编辑一个已有项目时，它将为非零。后一种情况中的提示将显示当前的值，用户可以简单地通过按 Enter 键来接受。

编辑已有成员的代码类似于创建新成员的代码。然而，我们有一个可操作的项目，所以提示例程显示当前项目的值，并且 edit_member() 函数发布一条 UPDATE 语句，而不是 INSERT 语句：

```
# edit existing contents of an entry

sub edit_member
{
  my ($col_name_ref, $entry_ref) = @_;
  my ($col_val, $query, $delim);

  # show initial values, ask for okay to go ahead and edit
  show_member ($col_name_ref, $entry_ref);
  return unless prompt ("\nEdit this entry? ") =~ /^y/i;
  # prompt for new values; user types in new value to replace
  # existing value, "null" to enter a NULL value, "exit" to exit
  # early, or Enter to accept existing value.
  foreach my $col_name (@{$col_name_ref})
  {
    next if $col_name eq "member_id"; # skip key field
    $col_val = $entry_ref->{$col_name};
    $col_val = "NULL" unless defined ($col_val);
    $col_val = col_prompt ($col_name, $col_val, 1);
    return if $col_val =~ /^exit$/i; # early exit
    $col_val = undef if $col_val =~ /^null$/i;
    $entry_ref->{$col_name} = $col_val;
  }
  # show new values, ask for confirmation before updating
  show_member ($col_name_ref, $entry_ref);
  return unless prompt ("\nUpdate this entry? ") =~ /^y/i;

  # construct an UPDATE query, then issue it.
  $query = "UPDATE member";
  $delim = " SET "; # put "SET" before first column, "," before others
  foreach my $col_name (@{$col_name_ref})
  {
    next if $col_name eq "member_id"; # skip key field
    # quote() quotes undef as the word NULL (without quotes),
    # which is what we want.
    $query .= sprintf ("%s %s=%s", $delim, $col_name,
                        $dbh->quote ($entry_ref->{$col_name}));
    $delim = ",";
  }
  $query .= " WHERE member_id = ?";
  warn "Warning: entry not updated?\n"
    unless $dbh->do ($query, undef, $entry_ref->{member_id}) == 1;
}
```

edit_member 的问题为它不进行任何输入值校验。对于 member 表中的大多数域，都没有什么校验——它们只是字符串域。但是对于 expiration 列，实际上应该检查输入值，以便确保它们看起来像日期。在一般目标的数据输入应用程序中，可能想抽取有关表的信息，以便确定它的所有列的类型。然后，可能按照那些类型上的约束条件来校验。那就比我在这里想探求的内容涉及得更多，所以我只在 col_prompt() 函数中增加一个快速方法，以便如果列名为“expiration”，则检查输入的格式。最低限度的日期值检查可以这样做：

```
sub col_prompt
{
  my ($name, $val, $show_current) = @_;
  my ($prompt, $sstr);
```



```

loop:
    $prompt = $name;
    $prompt .= " [$val]" if $show_current;
    $prompt .= ": ";
    print STDERR $prompt;
    chomp ($str = <STDIN>);
    # perform rudimentary check on the expiration date
    if ($str && $name eq "expiration") # check expiration date format
    {
        $str =~ /^d+[^d]\d+[^d]\d+$/
            or goto loop;
    }
    return ($str ? $str : $val);

```

这个模板测试了非数字字符分隔的三个序列的数字。这只是检查的一部分，因为它没有侦测如“1999-14-22”的值为无效。要想使脚本更好，则应该给它更严格的日期检查以及其他检查，如需要名和姓的域，就应该给非空值。

一些其他的改进可能是，如果没有更改列，则跳过这个更新，当用户正在编辑它时，如果其他一些人已经更改了这条记录，则通知这个用户。可以通过保存成员项目列的原始数据来做到这一点，然后，编写 UPDATE 语句来只更新那些已经更改的列。如果没有，则甚至不需要发布这条语句。同样，对于每个原始列值，可以编写 WHERE 子句来包括 AND col_name = col_val。如果其他一些人已经更改了这条记录，则这可能导致 UPDATE 失败，此时它的反馈为，两个人要同时更改这个项目。

7.3.4 寻找共同兴趣的历史同盟成员

历史同盟秘书的责任之一就是处理成员的请求，这些成员可能要求对美国历史领域内特殊时期或特殊人物（如在大萧条中或者亚伯拉罕·林肯的生命）感兴趣的其他人清单。当在字处理程序文档中维护这个目录时，使用字处理程序的“Find”功能，可以非常容易地找到这样的成员。然而，产生一系列只含有合格成员的项就要困难一些，因为它涉及大量的拷贝和粘贴。使用 MySQL，工作就变得容易得多，因为我们可以只运行如下这样的查询：

```

SELECT * FROM member WHERE interests LIKE "%lincoln%"
ORDER BY last_name, first_name

```

不幸的是，如果在 mysql 客户机程序运行这个查询，则结果看上去并不是非常好。让我们把少量的 DBI 脚本和生成较漂亮的输出的 interests 放在一起。首先，检查一下脚本，确保在命令行至少有一个命名的参数，因为如果没有一个命名的参数就没有内容可以搜索。然后，对于每个参数，脚本在 member 表的 interests 列上运行一个查询：

```

@ARGV or die "Usage: interests keyword\n";
search_members (shift (@ARGV)) while @ARGV;

```

为了搜索关键字字符串，我们在每一边都放了通配符‘%’，以便可以在 interests 列的任何地方都可以找到这个字符串。然后，我们显示相匹配的项：

```

sub search_members
{
    my ($interest) = shift;
    my ($sth, $count);

    print "Search results for keyword: $interest\n\n";
    $sth = $dbh->prepare (qq{
        SELECT * FROM member WHERE interests LIKE ?
        ORDER BY last_name, first_name
    });
    # look for string anywhere in interest field

```

```

$sth->execute ("% " . $interest . "%");
$count = 0;
while (my $hash_ref = $sth->fetchrow_hashref ())
{
    format_entry ($hash_ref);
    ++$count;
}
print "$count entries found\n\n";
}

```

这里没有出现 `format_entry()` 函数。它与 `gen_dir` 脚本的函数 `format_rtf_entry()` 在本质上是相同的, 但 `format_entry()` 函数去掉了 RTF 控制字。

7.3.5 联机历史同盟目录

在7.4节中, 我们将开始编写连接到 MySQL 服务器并抽取信息的脚本, 还要编写以 Web 页面形式在客户机的 Web 浏览器中出现的消息。那些脚本按照客户机请求动态地生成了 HTML。在我们到达那一点之前, 让我们通过编写生成能装载到 Web 服务器文档树中的静态 HTML 文档的 DBI 代码, 开始考虑有关的 HTML。以 HTML 格式创建的历史同盟目录是最好的选择, 因为我们的目标之一就是无论如何要使目录联机。

一般来说, HTML 文档有点像下面这样的结构:

<code><HTML></code>	← beginning of document
<code><HEAD></code>	← beginning of document head
<code><TITLE>My Page Title</TITLE></code>	← title of document
<code></HEAD></code>	← beginning of document head
<code><BODY></code>	← beginning of document body
<code><H1>My Level 1 Heading</H1></code>	← a level 1 heading
... content of document body ...	
<code></BODY></code>	← end of document body
<code></HTML></code>	← end of document

为了以这种格式生成目录, 编写完整的脚本对于你来讲并不必要。回想一下, 当我们编写 `gen_dir` 脚本时, 我们使用了可扩展的框架, 因此, 为了以其他格式产生目录而插入了代码。这意味着假如代码生成了 HTML 输出, 我们则需要编写文档初始化和清除的函数, 和格式化单独项一样。然后我们需要创建转换盒元素来指向这些函数。

只显示出的联机文档非常容易地分解为可以由初始化函数和清除函数处理的序言和收尾部分, 以及由项目格式化函数生成的中间部分。HTML 初始化函数生成级别1标题的每一部分, 而清除函数生成关闭 `</BODY>` 和 `</HTML>` 标记的部分:

```

sub html_init
{
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>US Historical League Member Directory</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>US Historical League Member Directory</H1>\n";
}

sub html_cleanup
{
    print "</BODY>\n";
    print "</HTML>\n";
}

```

一般来说，真正的工作在于格式化项目。但即使这样也不太困难。我们可以拷贝 `format_rtf_entry()` 函数，确保项目中的任何特殊字符都被编码，并且用 HTML 标出的标志替换 RTF 控制字：

```
sub format_html_entry
{
    my ($entry_ref) = shift;
    my ($address);

    # encode characters that are special in HTML
    foreach my $key (keys (%{$entry_ref}))
    {
        $entry_ref->{$key} =~ s/&/&amp;/g;
        $entry_ref->{$key} =~ s/"/&quot;/g;
        $entry_ref->{$key} =~ s/>/&gt;/g;
        $entry_ref->{$key} =~ s/</&lt;/g;
    }
    printf "<STRONG>Name: %s</STRONG><BR>\n", format_name ($entry_ref);
    $address = "";
    $address .= $entry_ref->{street} if $entry_ref->{street};
    $address .= ", " . $entry_ref->{city} if $entry_ref->{city};
    $address .= ", " . $entry_ref->{state} if $entry_ref->{state};
    $address .= " " . $entry_ref->{zip} if $entry_ref->{zip};
    print "Address: $address<BR>\n" if $address;
    print "Telephone: $entry_ref->{phone}<BR>\n" if $entry_ref->{phone};
    print "Email: $entry_ref->{email}<BR>\n" if $entry_ref->{email};
    print "Interests: $entry_ref->{interests}<BR>\n"
        if $entry_ref->{interests};
    print "<BR>\n";
}
```

现在我们把另一个元素加到转换盒中，指出编写 HTML 的函数，并且完成对 `gen_dir` 的更正：

```
# switchbox containing formatting functions for each output format
my (%switchbox) =
(
    "banquet" =>
        # functions for banquet list
        {
            "init"    => undef,          # no initialization needed
            "entry"    => \&format_banquet_entry,
            "cleanup"  => undef          # no cleanup needed
        },
    "rtf" =>
        # functions for RTF format
        {
            "init"    => \&rtf_init,
            "entry"    => \&format_rtf_entry,
            "cleanup"  => \&rtf_cleanup
        },
    "html" =>
        # functions for HTML format
        {
            "init"    => \&html_init,
            "entry"    => \&format_html_entry,
            "cleanup"  => \&html_cleanup
        }
);
```

为了产生 HTML 格式的目录，运行下面的命令并在 Web 服务器的文档树中安装结果输出文件：

```
% gen_dir html > directory.html
```

当更新目录时，可以再次运行命令来更新联机版本。另一个方案是建立周期性执行的 cron 作业。那就是说，联机目录将被自动地更新。例如，我可能使用类似于这个的 crontab 项在每天早晨4点运行 gen_dir：

```
0 4 * * * /u/paul/samp_db/gen_dir > /usr/local/apache/htdocs/directory.html
```

这个 cron 作业所运行的用户必须允许它们都执行位于 samp_db 目录中的脚本，并将文件编写到 Web 服务器的文档树中。

7.4 在 Web 应用程序中使用 DBI

迄今为止，我们编写的 DBI 脚本用于命令行环境中的命令解释程序，但 DBI 在其他环境下也是有用的，例如在基于 Web 的应用程序的开发中。当编写能从 Web 浏览器调用的 DBI 脚本时，就打开了新鲜而有趣的与数据库交互的性能。

例如，如果以表格的形式显示数据，则可以很容易地把每个列标题转换为可以选择的连接，以便将该列的数据重新排序。它允许单击一次就可以以不同的方式查看数据，而又不必键入任何查询。或者可以提供一种用户可以为数据库搜索而键入的标准格式，然后，显示含有搜索结果的页面。像这种简单的能力能够特别地改变为访问数据库内容而提供的交互性的水平。除此之外，Web 浏览器的显示能力比在终端窗口获得的能力要明显地更好一些，所以，输出也经常看起来更漂亮。

在这部分，我们将创建下面的基于 Web 的脚本：

- **samp_db 数据库中表的通用浏览器。**这与我们想对这个数据库完成的任何特定的任务无关，但是它举例说明了若干 Web 程序设计概念，并提供了一种查看这些表所含有的信息的方便方式。
- **允许我们查看任何给定的测验或测试分数的分数浏览器。**它作为回顾评分事件结果的快速方式是很方便的，并且当我们需要创建测试的等级曲线时，它是有用的，所以我们可以以字母等级来标记试卷。
- **寻找分享共同兴趣的历史同盟成员脚本。**通过允许用户输入搜索短语来完成它，然后在 member 表的 interests 域来搜索短语。我们已经编写了一个行命令脚本来做这些，但是，基于 Web 的版本提供了有指导意义的参考观点，允许对同一任务比较两种方法。

我们将使用 CGI.pm Perl 模块来编写这些脚本，这个模块是将 DBI 脚本连接到 Web 上最容易的方法（有关获得 CGI.pm 模块的说明，请参阅附录 A）。之所以称为 CGI.pm，是因为它有助于编写使用公共网关协议的脚本，这个协议定义了 Web 服务器如何与其他程序通信。CGI.pm 处理涉及了许多通用内务处理的任务细节，如收集通过 Web 服务器传递到脚本的作为输出的参数值。CGI.pm 也提供了生成 HTML 输出的便利方法，与编写自己原始的 HTML 标记相比，它减少了编写难看的 HTML 的机会。

在本章中，您将学到足够有关 CGI.pm 的知识来编写自己的 Web 应用程序，但是，当然不是它所包括的所有性能。要想学习有关这个模块的更多知识，请参阅 Lincoln Stein (John Wiley 1998 出版) 撰写的《Official Guide to Programming with CGI.pm》，或在以下网址查阅联机文档：

<http://stein.cshl.org/www/software/CGI/>

7.4.1 设置 CGI 脚本的 Apache

除了 DBI 和 CGI.pm 之外，编写基于 Web 的脚本还需要有一个以上的组件：Web 服务器。这里的说明适合 Apache 服务器使用脚本，但是，如果愿意，稍微改编一点这些说明，就可以使用不同的服务器。

一般来说，Apache 装置的各个部分位于 /usr/local/apache 目录。对我们的目的来讲，这个目录中最重要的子目录为 htdocs（HTML 文档树）、cgi-bin（可执行的脚本和 Web 服务器调用的程序），和 conf（配置文件）。这些目录也可能放在系统中的其他地方。如果是这样，则要对下面的注意事项做适当的调整。

应该验证 cgi-bin 目录不在 Apache 文档树的内部，以便它内部的这些脚本不能作为无格式文本来请求。这是个安全的防范方法。您也不愿意让怀有恶意的客户机程序检查您的脚本，通过提取这些脚本的文本并研究它们来作为安全的突破口。

要想安装以 Apache 方式使用的 CGI 脚本，则将它放在 cgi-bin 目录下，然后将这个脚本的所有权更改为运行 Apache 的用户，并将它的模式更改为对该用户为可执行的和只读的模式。例如，如果 Apache 以名称为 www 的用户方式运行，则使用下面的命令：

```
% chown www script_name
% chmod 500 script_name
```

可能需要用 www 或 root 运行这些命令。如果不允许在 cgi-bin 目录下安装脚本，则可以请求系统管理员代表您来这样做。

安装这个脚本之后，通过向 Web 服务器发送适当的 URL，可以请求浏览器上的这个脚本。典型的 URL 是这样的：

http://your.host.name/cgi-bin/script_name

从 Web 浏览器请求脚本会导致 Web 服务器执行它。返回脚本的输出，结果作为 Web 页面出现在浏览器中。

如果为寻求更好的性能而使用具有 mod_perl 的 CGI 脚本，则可以这样做：

- 1) 确保至少有以下版本的必需软件：Perl 5.004、CGI.pm 2.36 和 mod_perl 1.07。
- 2) 确保将 mod_perl 编译为 Apache 可执行的文件。
- 3) 建立一个存储脚本的目录。我使用了 /usr/local/apache/cgi-perl。cgi-bin 不应该位于 Apache 文档树的内部，出于同样的安全原因，cgi-perl 目录也不应该在那里。
- 4) 告知 Apache，与位于 cgi-perl 目录中的脚本 mod_perl 相关联：

```
Alias /cgi-perl/ /usr/local/apache/cgi-perl
```

```
<Location /cgi-perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
</Location>
```

如果正在使用 Apache 的当前版本，这个版本使用单个的配置文件，则将所有这些指示放在 httpd.conf 中。如果 Apache 的版本使用三个旧文件的方法来配置信息，则将 Alias 指示放入 srm.conf 中，将 Location 行放入 access.conf 中。对于 cgi-perl 目录，不要启用 mod_perl、PerlSendHeader 或 PerlSetupEnv 指示。这些由 CGI.pm 自动地处理，启用它们可能导致处理冲突。

mod_perl 脚本的 URL 与标准的 CGI 脚本的 URL 相类似。唯一的不同之处在于指定 cgi-perl 而不是 cgi-bin。

```
http://your.host.name/cgi-perl/script_name
```

有关的详细信息，请参阅下面地址的 Apache Web 站点的 mod_perl 区域：

```
http://perl.apache.org/
```

7.4.2 CGI.pm 的简要介绍

为了编写使用 CGI.pm 模块的 Perl 脚本，将 use 行放在这个脚本的开头附近，然后创建让您访问 CGI.pm 方法和变量的 CGI 对象：

```
use CGI;  
my ($cgi) = new CGI;
```

我们的 CGI 脚本使用了 CGI.pm 的性能，它通过使用 \$cgi 变量调用方法来实现。例如，为了生成级别1标题，我们将这样使用 h1() 方法：

```
print $cgi->h1 ("My Heading");
```

CGI.pm 也支持允许以函数调用它的方法的使用风格，而不用前导的 ‘ \$cgi-> ’。在这里，我没有使用这个语法，是因为 ‘ \$cgi-> ’ 符号更类似于使用 DBI 的方式，还因为它防止 CGI.pm 函数名与可以定义的任何函数名产生冲突。

1. 检查输入参数，并编写输出

CGI.pm 所做的事情之一就是照看所有丑陋的细节，这些细节涉及到收集由 Web 服务器向脚本提供的输入信息。为了获得那些信息，所需做的就是调用 param() 方法。可以如下获得所有可用的参数名：

```
my (@param) = $cgi->param ();
```

为了检索特定参数的值，只命名感兴趣的参数：

```
if (!$cgi->param ("my_param"))  
{  
    print "my_param is not set\n";  
}  
else  
{  
    printf "my_param value: %s\n", $cgi->param ("my_param");  
}
```

CGI.pm 还提供生成传送给客户机浏览器的输出方法。考虑下面的 HTML 文档：

```
<HTML>  
<HEAD>  
<TITLE>My Page Title</TITLE>  
</HEAD>  
<BODY>  
<H1>My Level-1 Heading</H1>  
<P>Paragraph 1.  
<P>Paragraph 2.  
</BODY>  
</HTML>
```

这个代码使用 \$cgi 来产生等价的文档：

```
print $cgi->header ();  
print $cgi->start_html (-title => "My Page Title");
```



```
print $cgi->h1 ("My Page Heading");
print $cgi->p ();
print "Paragraph 1.\n";
print $cgi->p ();
print "Paragraph 2.\n";
print $cgi->end_html ();
```

使用 CGI.pm 生成输出，而不是编写自己原始的 HTML，这样做的一些优点是，可以按逻辑单元考虑，而不是按单独的组成标识来考虑，而且 HTML 不太可能含有错误（我说“不太可能”的原因是 CGI.pm 不禁止做古怪的事情，如含有一列内部的标题）。除此之外，对于编写的非标记文本，CGI.pm 提供自动的字符转义，如 HTML 中指定的‘<’和‘>’。

如果愿意，CGI.pm 生成输出方法的使用并不排斥编写自己原始的 HTML。可以将这两种方法混合起来，组合调用具有生成文字标识的显示语句的 CGI.pm 方法。

2. 转义的 HTML 和 URL 文本

如果经 CGI.pm 方法，如 start_html() 或 h1()，编写非标记的文本，则自动地转义文本中的特定字符。例如，如果使用下面的语句生成标题，则标题文本中的‘&’字符将由 CGI.pm 转换为‘&’：

```
print $cgi->start_html (-title => "A, B & C");
```

如果不使用 CGI.pm 生成输出的方法编写非标记的文本，则可能应该先让它经过 escapeHTML()，以便确保可以正确地转义任何指定的字符。当构造可能含有特定字符的 URL 时也是这样，尽管在那种情况下应该使用 escape() 方法来代替它。使用适当的编码方法是很重要的，因为每种方法都将不同的字符集作为特殊的字符来对待，并使用彼此不同的格式来对待特殊的字符编码。考虑下面简短的 Perl 脚本：

```
#!/usr/bin/perl
use CGI;
$cgi = new CGI;

$s = "x<y, right?";
print $cgi->escapeHTML($s) . "\n"; # encode for use as HTML text
print $cgi->escape($s) . "\n";     # encode for use in a URL
```

如果运行这个脚本，则它生成下面的输出，从这里可以看到 HTML 文本的编码不同于 URL 的编码：

```
x&lt;y, right
x%3C%3Dy%2C%20right%3F
```

3. 编写多目的页面

编写基于 Web 的脚本来生成 HTML，而不是编写静态的 HTML 文档的主要原因之一是，根据调用方式，脚本可以产生不同类型的页面。我们将要编写的所有 CGI 脚本都有这种特性。每一个都像下面这样操作：

- 1) 当从浏览器第一次请求这个脚本时，它生成一个初始页面，允许选择想要的信息类型。
- 2) 当做了选择以后，重新调用这个脚本，但是，这次它在第二页检索，并显示请求的特定信息。

这里的主要问题是想从第一页的选择中确定第二页的内容，但是，通常 Web 页面是彼此独立的，除非安排某些特定排列的次序。这个窍门是让脚本生成页面，这个页面给参数设置一个值，告诉这个脚本的下一个调用想要的内容。当第一次调用这个脚本时，这个参数没有值；告诉这个脚本给出它的初始页面。当指出想看的信息内容时，这个页面再一次调用这个

脚本，但是，将参数设为指示这个脚本做什么的一个值。

将说明从页面传送回脚本有不同的方式。一种方式是提供一种用户填写的表格。当用户提交这张表格时，将它的内容提交给 Web 服务器。服务器将信息传递给脚本，这个脚本通过调用 `param()` 方法，能够找出提交的内容。这就是我们对第三个 CGI 脚本所做的事情（允许用户输入搜索历史同盟目录的关键字）。

对脚本指定说明的另外一种方法是，当请求脚本时，将信息作为发送到 Web 服务器的 URL 的一部分来传递。这就是我们对于 `samp_db` 表浏览器和分数浏览器脚本要做的事情。这种工作方式是脚本生成含有超链接的页面。选择一个连接，再次调用这个脚本，但是，这次指定参数值，这个参数值指示这个脚本做什么。实际上，这个脚本以不同的方式调用它本身，来提供不同类型的结果，这取决于用户所选择的连接。

脚本可以允许通过向浏览器向它自己的 URL 传送一个含有超链接的页面来调用它本身。例如，脚本 `my_script` 可以编写含有如下这样连接的页面：

```
<A HREF="/cgi-bin/my_script">Click Me!</A>
```

当用户敲入文本“Click Me!”时，用户浏览器就请求将 `my_script` 发送回 Web 服务器。当然，所有这些会导致脚本再次发送出同一个页面，因为它不支持其他信息。然而，如果将一个参数附加到 URL 上，则当用户选择这个连接时，将这个参数送回 Web 浏览器。服务器调用这个脚本，这个脚本可以调用 `param()` 来侦测设置的参数，并根据它的值采取行动。

为了把参数附到 URL 的末尾，加一个“?”字符放到名称/值的前面。为了附上多个参数，用字符“&”分隔。例如：

```
/cgi-bin/my_script?name=value  
/cgi-bin/my_script?name=value&name2=value2
```

为了构造带有附加参数的自引用的 URL，CGI 脚本应该通过调用 `script_name()` 方法获得自己的 URL 来开始，然后像按照如下方法添加参数：

```
$url = $cgi->script_name(); # get URL for script  
$url .= "?name=value";      # add first parameter  
$url .= "&name2=value2";     # add second parameter
```

在构造 URL 之后，通过使用 CGI.pm 的 `a()` 方法，可以生成一个包括它的超链接 `<A>` 标记：

```
print $cgi->a ({-href => $url}, "Click Me!");
```

通过检查一个简短的 CGI 脚本来查看如何工作会更容易。第一次调用时，下面的脚本 `flip_flop`，给出了一个含有单个超链接的称为页面 A 的页面。选择这个连接再次调用这个脚本，但是设置 `page` 参数，告诉它显示页面 B。页面 B 也包括对脚本的连接，但是 `page` 参数没有值。因此，在页面 B 中选择这个连接导致重新显示原始页面。随后的脚本调用将页面在脚本 A 和脚本 B 之间来回切换：

```
use CGI;  
  
my ($cgi) = new CGI;  
my ($url) = $cgi->script_name(); # this script's own URL  
  
print $cgi->header();  
if ($cgi->param ("page") ne "b") # display page A  
{  
    print $cgi->start_html (-title => "Flip-Flop: Page A");
```

```

print "This is Page A.<BR>To select Page B, ";
$url .= "?page=b";      # attach parameter to select page B
print $cgi->a ({-href => $url}, "click here");
}
else
    # display page B
{
    print $cgi->start_html (-title => "Flip-Flop: Page B");
    print "This is Page B.<BR>To select Page A, ";
    print $cgi->a ({-href => $url}, "click here");
}
print $cgi->end_html ();

```

如果另一个客户机程序出现并请求 flip_flop, 就给出初始页面, 因为不同客户机的浏览器并不互相影响。

实际上, \$url 的值被前面的样例设置成漂亮的风格。在把它们放在 URL 之后以免包括特殊字符时, 使用 escape() 方法对参数名和值进行编码是比较好的。这里有一个较好的方法来用附加的参数值来构造 URL :

```

$url = $cgi->script_name ();      # get URL for script
$url .= sprintf ("%s=%s",      # add first parameter
    $cgi->escape ("name"), $cgi->escape ("value"));
$url .= sprintf ("%s=%s",      # add second parameter
    $cgi->escape ("name2"), $cgi->escape ("value2"));

```

7.4.3 从 Web 脚本连接到 MySQL 服务器

我们在前一节“运行 DBI”中开发的命令行脚本, 为建立到 MySQL 服务器的连接共享了一个通用的前文。CGI 脚本也共享了一些代码, 但是有一些不同:

```

use DBI;
use CGI;
use strict;

# default connection parameters - all missing
my ($host_name, $user_name, $password) = (undef, undef, undef);
my ($db_name) = "samp_db";

# construct data source
my ($dsn) = "DBI:mysql:$db_name";
$dsn .= ":hostname=$host_name" if $host_name;
$dsn .= ";mysql_read_default_file=/usr/local/apache/conf/samp_db.cnf";

# connect to server
my (%attr) = ( RaiseError => 1 );
my ($dbh) = DBI->connect ($dsn, $user_name, $password, \%attr);

```

这个前文与命令行脚本使用的前文的不同之处在于以下几个方面:

- 第一部分现在含有一条 use CGI 语句。
- 不再分析命令行的参数。
- 代码仍然在可选文件中寻找连接参数, 但是, 在用户执行脚本的主目录中不使用 .my.cnf 文件 (也就 Web 服务器用户的主目录)。Web 服务器可能运行访问其他数据库的脚本, 没有理由假设所有脚本会使用同一连接参数。相反, 我们寻找不同位置存放的可选文件 (/usr/local/apache/conf/samp_db.cnf)。如果想使用不同的文件, 应该修改可选文件的路径名。

通过 Web 服务器调用的脚本作为 Web 服务器用户，而不是作为您来运行。这就提出了一些安全问题，因为在 Web 服务器接管之后您就不再控制了。应该把可选文件的所有权交给运行 Web 服务器的用户（可能是 www 或者 nobody 或者一些类似的用户），并将模式设置为 400 或 600，以便其他用户不能读取。不幸的是，可以安装这个 Web 服务器的脚本来执行的任何人仍然能够读取这个文件。他们要做的所有事情就是编写一个脚本，显式地打开可选文件，并在 Web 页面上显示它的内容。因为他们的脚本作为 Web 服务器用户来运行，所以它将有足够的权利来读取这个文件。

由于这个原因，创建一个对 samp_db 数据库具有只读（SELECT）权限的 MySQL 用户，然后在 samp_db.cnf 文件中列出这个用户的名称和口令，而不是您自己的名称和口令，这种行为是很谨慎的。作为有权修改数据库的表的用户，这种方式不会冒险允许脚本连接到数据库。第 11 章“常规的 MySQL 管理”，讨论了如何创建具有严格权限的 MySQL 用户账户。

另一种选择，可以在 Apache 的 suEXEC 机制下安排执行脚本。这就允许作为特殊权限的用户执行脚本，然后编写脚本，从只对那个用户为只读的可选文件中获得连接参数。例如，需要编写访问数据库的脚本，就可以这样做。

还有另外一种方法就是编写脚本，从客户机用户请求用户姓名和口令，并使用这些值建立到 MySQL 服务器的连接。这种方法对于为管理目的而创建脚本比对于为一般使用提供脚本更适合。无论如何，应该警惕用户名和口令请求的一些方法受到一些人的攻击，这些人可能您在您和服务端之间的网络上安放窃听器。

因为可以从前面的段落中搜集，所以 Web 脚本的安全性是个棘手的问题。很明显，应该多读一些有关安全的主题，因为它是一个大的主题，所以在这里我不能真正做得很全面。查看 Apache 手册中有关安全性的资料是一种好的方法。您也可以查找 WWW 安全性的 FAQ 说明，例如可以使用下面的网址：

<http://www.w3.org/Security/Faq/>

7.4.4 samp_db 数据库浏览器

对于第一个基于 Web 的应用程序，我们将开发一个简单的脚本——samp_db——允许查看 samp_db 数据库中存在的表，并从 Web 浏览器中交互式地检查这些表中的内容。samp_db 的工作方式如下：

- 当首次从浏览器中请求 samp_db 时，它连接到 MySQL 服务器，在 samp_db 数据库中检索一列表，并向浏览器发送一个页面，在这个页面中出现的每个表都作为可选择的连接。当选择这个页面中的一个表名时，浏览器就向 Web 服务器发送一个请求，请求 samp_browse 显示那个表的内容。
- 当调用 samp_browse 时，如果它收到从 Web 服务器发来的一个表名，则它就检索这个表的内容，并将信息显示在 Web 浏览器上。数据每列的标题就是表中列的名称。标题作为连接出现；如果选择它们中的一个，则浏览器就向 Web 服务器发送一个请求，显示同样的表，但按选择的列排序。

注意，这里有个警告：samp_db 表中的这些表相对较小，因此向浏览器发送表的全部内容并不是大问题。如果编辑 samp_db，显示包含大型表的不同数据库中的表，则应该考虑向行检索语句中增加一个 LIMIT 子句。

在 samp_browse 脚本的主体中，我们创建了 CGI 对象，并取消了 Web 页面的初始部分。然后检查是否按我们的假设，根据 tbl_name 参数值显示了一些特定的表：

```
my ($cgi) = new CGI;

# put out initial part of page
my ($title) = "$db_name Database Browser";
print $cgi->header ();
print $cgi->start_html (-title => $title);
print $cgi->h1 ($title);

# parameters to look for in URL
my ($tbl_name) = $cgi->param ("tbl_name");
my ($sort_column) = $cgi->param ("sort_column");

# if $tbl_name has no value, display a clickable list of tables.
# Otherwise, display contents of the given table. $sort_column, if
# set, indicates which column to sort by.
if (!$tbl_name)
{
    display_table_list ()
}
else
{
    display_table ($tbl_name, $sort_column);
}

print $cgi->end_html ();
```

很容易找出参数的值，因为 CGI.pm 做了找出 Web 服务器传递给这个脚本信息的全部工作。我们只需调用具有我们感兴趣的参数名的 param()，在 samp_browse 的主体中，这个参数为 tbl_name。如果它没有定义或者为空，则它就是这个脚本的初始调用，我们显示这个表列。否则，就显示由 tbl_name 参数命名的表的内容，由 sort_column 参数命名的列值排序。显示适当的信息之后，我们调用 end_html() 消除结束的 HTML 标志。

display_table_list() 函数生成初始页面。display_table_list() 检索这个表列并写出在每个单元中都含有一个数据库表名的单列的 HTML 表：

```
sub display_table_list
{
    my ($ary_ref, $url) = @_;

    print "Select a table by clicking on its name:<BR><BR>\n";

    # retrieve reference to single-column array of table names
    $ary_ref =
        $dbh->selectcol_arrayref (qq{ SHOW TABLES FROM $db_name });

    # display table with a border
    print "<TABLE BORDER>\n";
    print "<TR>\n";
    display_cell ("TH", "Table Name", 1);
    print "</TR>\n";
    foreach my $tbl_name (@{$ary_ref})
    {
        $url = $cgi->script_name ();
        $url .= sprintf ("%s", $cgi->escape ($tbl_name));
        print "<TR>\n";
        display_cell ("TD", $cgi->a ({-href => $url}, $tbl_name), 0);
    }
}
```

```

        print "</TR>\n";
    }
    print "</TABLE>\n";
}

```

display_table_list() 生成的页面含有如下连接：

```

/cgi-bin/samp_browse?tbl_name=absence
/cgi-bin/samp_browse?tbl_name=event
/cgi-bin/samp_browse?tbl_name=member
...

```

当调用 samp_browse 时，如果 tbl_name 参数有值，则这个脚本将这个值传递给 display_table()，连同按名称排序后的列名。如果没有命名的列，则我们按第一列排序（我们可以通过位置引用列，因而很容易地使用 ORDER BY 1 子句来完成）：

```

sub display_table
{
    my ($tbl_name, $sort_column) = @_;
    my ($sth, $url);

    # if sort column not specified, use first column
    $sort_column = "1" unless $sort_column;

    # present a link that returns user to table list page
    print $cgi->a ({-href => $cgi->script_name (), "Show Table List"});
    print "<BR><BR>\n";

    $sth = $dbh->prepare (qq{
        SELECT * FROM $tbl_name ORDER BY $sort_column
    });
    $sth->execute ();

    print "<B>Contents of $tbl_name table:</B><BR>\n";

    # display table with a border
    print "<TABLE BORDER>\n";
    # use column names for table headings; make each heading a link
    # that sorts output on the corresponding column
    print "<TR>\n";
    foreach my $col_name (@{$sth->{NAME}})
    {
        $url = $cgi->script_name ();
        $url .= sprintf ("?tbl_name=%s", $cgi->escape ($tbl_name));
        $url .= sprintf ("%sort_column=%s", $cgi->escape ($col_name));
        display_cell ("TH", $cgi->a ({-href => $url}, $col_name), 0);
    }
    print "</TR>\n";

    # display table rows
    while (my @ary = $sth->fetchrow_array ())
    {
        print "<TR>\n";
        foreach my $val (@ary)
        {
            display_cell ("TD", $val, 1);
        }
        print "</TR>\n";
    }

    $sth->finish ();
    print "</TABLE>\n";
}

```


表显示了与重新显示该表的连接相关的列标题的页面；这些连接包括 `sort_column` 参数，它显式地指定排序的列。例如，对于显示 `event` 表内容的页面，列标题连接看起来如下所示：

```
/cgi-bin/samp_browse?tbl_name=event&sort_column=date
/cgi-bin/samp_browse?tbl_name=event&sort_column=type
/cgi-bin/samp_browse?tbl_name=event&sort_column=event_id
```

`display_table_list()` 和 `display_table()` 都使用了 `display_cell()`，HTML 表中作为单元显示值的实用程序函数。这个函数使用了一个小窍门，就是将空值转换为不可分的空格（` `），因为在带有边框的表中，空单元不会正确地显示边框。将不可分的空格放入这个单元中解决了这个问题。`display_cell()` 还具有控制是否将单元值编码的第三个参数。这是必需的，因为调用 `display_cell()`，显示了一些已经编码的单元值，如含有 URL 信息的列标题：

```
# display a value in a table cell; put non-breaking
# space in "empty" cells so borders show up

sub display_cell
{
    my ($tag, $value, $encode) = @_;

    $value = $cgi->escapeHTML($value) if $encode;
    $value = "&nbsp;" unless $value;
    print "<$tag>$value</$tag>\n";
}
```

如果想编写更通用的脚本，可以将 `samp_browse` 更改为浏览多个数据库。例如，在脚本开始时，可以显示服务器上的一列数据库，而不是一个特定数据库中的一列表。然后，选出一个数据库，获得它的一列表，再从那里继续。

7.4.5 学分保存方案分数浏览器

每当我们输入测试的分数时，都需要生成一个有序的分数字列表，以便确定等级曲线，并分配字母等级。请注意，对于这个列表我们将做的所有事情就是显示它，以便能确定每个字母等级终止的位置。然后在返回给学生之前，在测试卷上标出等级分数。我们不在数据库中继续记录这个字母等级，因为在等级周期末尾的等级取决于数字分数，而不是字母等级。再请注意，严格地说，在创建检索分数的方法之前，就应该有一个输入分数的方法。我将输入分数的脚本一直保存到下一章。在这期间，在数据库中，我们已经从早期的等级周期部分中得到了几组分数。即使没有方便的分数输入方法，我们也可以使用具有那些分数的脚本。

我们浏览分数的脚本 `score_browse` 与 `samp_browse` 有些类似，但是希望查看给定测试或测验的分数这种更特定的目标。初始页面给出一列可以选择的可能的等级事件，允许用户选择它们中的任何一个，来查看与事件相关的分数。给定事件的分数按照高分在前的顺序按分数排序，因此可以显示出结果，并用它确定等级曲线。

`score_browse` 只需要检查一个参数 `event_id`，查看是否指定了特定事件。如果不是，则 `score_browse` 就显示 `event` 表中的行，以便用户可以选择其中的一个。否则，就显示与所选事件相关的分数：

```
# parameter that tells us which event to display scores for
my ($event_id) = $cgi->param("event_id");

# if $event_id has a value, display the event list.
# otherwise display the scores for the given event.
```

```

if (!$event_id)
{
    display_events ()
}
else
{
    display_scores ($event_id);
}

```

使用请求表列标题的列名，函数 `display_events()` 从 `event` 表中抽取信息，并以表格形式显示它。在每行的内部，显示 `event_id` 值，作为可以选择的连接，以触发检索相应事件分数的查询。每个事件的 URL 都只是到具有附加参数 `score_browse` 的路径，这个参数指定事件号码：

```
/cgi-bin/score_browse?event_id=number
```

`display_events()` 函数编写如下：

```

sub display_events
{
    my ($sth, $url);
    print "Select an event by clicking on its number:<BR><BR>\n";

    # get list of events
    $sth = $dbh->prepare (qq{
        SELECT event_id, date, type
        FROM event
        ORDER BY event_id
    });
    $sth->execute ();

    # display table with a border
    print "<TABLE BORDER>\n";
    # use column names for table column headings
    print "<TR>\n";
    foreach my $col_name (@{$sth->{NAME}})
    {
        display_cell ("TH", $col_name, 1);
    }
    print "</TR>\n";

    # associate each event id with a link that will show the
    # scores for the event; return rows using a hash to make
    # it easy to refer to the event_id column value by name.
    while (my $hash_ref = $sth->fetchrow_hashref ())
    {
        print "<TR>\n";
        $url = $cgi->script_name ();
        $url .= sprintf ("%event_id=%s",
            $cgi->escape ($hash_ref->{event_id}));
        display_cell ("TD",
            $cgi->a ({-href => $url}, $hash_ref->{event_id}), 0);
        display_cell ("TD", $hash_ref->{date}, 1);
        display_cell ("TD", $hash_ref->{type}, 1);
        print "</TR>\n";
    }
    $sth->finish ();
    print "</TABLE>\n";
}

```

当用户选择事件时，浏览器发送一个具有附加事件 ID 值的 `score_browse` 请求。

score_browse 找到 event_id 参数集，并调用 display_scores() 列出所有特定事件的分数。这个页面也显示了文本“Show Event List”，作为返回初始页面的连接，以使用户能很容易地返回事件列表页面。这个连接的 URL 引用了 score_browse 脚本，但不指定 event_id 参数的任何值。display_scores() 子程序如下所示：

```
sub display_scores
{
my ($event_id) = shift;
continued
my ($sth);

# a URL without any event_id parameter will
# cause the event list to be displayed.
print $cgi->a ({-href => $cgi->script_name ()}, "Show Event List");
print "<BR><BR>\n";

# select scores for the given event
$sth = $dbh->prepare (qq{
    SELECT
        student.name, event.date, score.score, event.type
    FROM
        student, score, event
    WHERE
        student.student_id = score.student_id
        AND score.event_id = event.event_id
        AND event.event_id = ?
    ORDER BY
        event.date ASC, event.type ASC, score.score DESC
});
$sth->execute ($event_id); # pass event ID as placeholder value

print "<B>Scores for event $event_id</B><BR>\n";

# display table with a border
print "<TABLE BORDER>\n";
# use column names for table column headings
print "<TR>\n";
foreach my $col_name (@{$sth->{NAME}})
{
    display_cell ("TH", $col_name, 1);
}
print "</TR>\n";

while (my @ary = $sth->fetchrow_array ())
{
    print "<TR>\n";
    display_cell ("TD", shift (@ary), 1) while @ary;
    print "</TR>\n";
}

$sth->finish ();
print "</TABLE>\n";
}
```

display_scores() 运行的查询与我们以前在第1章的1.4.8节中的“从多个表中检索信息”小节中开发的说明如何编写连接的查询极为类似。在那一章中，我们请求给定日期的分数，因为日期比事件的ID值更有意义。相反，当我们使用 score_browse 时，知道了精确的事件ID。那不是因为我们按照事件 ID 考虑（我们没有），而是因为脚本给了我们一系列可从中选择的事

件ID。可以看到这种类型的接口减少了了解特定细节的需要。我们不必了解事件的 ID；只需要识别出想要的事件。

7.4.6 历史同盟共同兴趣的搜索

samp_browse 和 score_browse 脚本通过在初始页面给出一列选择而允许用户做出选择，那个页面中的每个选择都是用特定参数值再次调用这个脚本的一个连接。允许用户做出选择的另外一种方法是，将含有可编辑域的表格放在页面中。当可选范围没有约束到一些容易确定的值的集合时，这种方法更加合适。我们的下一个脚本举例说明了请求用户输入的这种方法。

在7.3节“运行 DBI”中，我们为寻找共享特定兴趣的历史联盟成员构造了一个命令行脚本。然而，那个脚本并不是联盟成员已经访问的脚本；联盟秘书必须运行这个脚本，然后把结果邮寄给请求这个列表的成员。最好使这个搜寻性能更广泛，以便成员可以自己使用。编写 Web 脚本就是进行这种事的一种方法。

脚本 interests 把少量表格放到用户能够输入关键字的地方，然后搜索 member 表，寻找有满足条件的成员并显示结果。通过将通配符“ % ”加到关键字的两端，来执行这个搜索，以便在 interests 列值的任何地方都能找到。

在每个页面都显示关键字表格，以使用户可以立即输入新的搜索，甚至在显示搜索结果的页面中也可以。除此之外，还在关键字表格中显示前面页面的搜索字符串，以便如果用户想要运行类似的搜索，可以编辑这个字符串。这样就不必重新键入许多内容了：

```
# parameter to look for
my ($interest) = $cgi->param ("interest");

# Display a keyword entry form. In addition, if $interest is defined,
# search for and display a list of members who have that interest.
# Note that the current $interest value is displayed as the default
# value of the interest field in the form.

print $cgi->start_form (-method => "POST");
print $cgi->textfield (-name => "interest",
                    -value => $interest,
                    -size => 40);
print $cgi->submit (-name => "button", -value => "Search");
print $cgi->end_form ();

# run a search if a keyword was specified
search_members ($interest) if $interest;
```

脚本和自己交流信息与 samp_browse 或 score_browse 略有不同。interest 参数没有加到 URL 的末尾，而表格中的信息由浏览器编码，并作为 POST 请求的一部分发送出去。然而，CGI.pm 使如何发送信息成为不相关的；参数值仍然通过调用 param() 来获得。

实现搜索和显示结果的函数如下所示，没有显示格式化项的函数 format_html_entry()，因为它与 gen_dir 脚本中的相同：

```
sub search_members
{
my ($interest) = shift;
my ($sth, $count);

printf "Search results for keyword: %s<BR><BR>\n",
```

```
                                $cgi->escapeHTML ($interest);
$sth = $dbh->prepare (qq{
    SELECT * FROM member WHERE interests LIKE ?
    ORDER BY last_name, first_name
});
# look for string anywhere in interest field
$sth->execute ("% " . $interest . "%");
$count = 0;
while (my $hash_ref = $sth->fetchrow_hashref ())
{
    format_html_entry ($hash_ref);
    ++$count;
}
print $cgi->p (" $count entries found");
}
```

第8章 PHP API

PHP 是一种脚本语言，可以编写包含嵌入式代码的 Web 页面，只要访问页面就能执行这些代码，并且这些代码还能生成动态的内容作为输出的一部分发送到客户机的 Web 浏览器。本章描述了如何用 PHP 编写使用 MySQL 的基于 Web 的应用程序。有关 C 与 PHP 和 Perl DBI API 的比较，请参阅第5章“MySQL 程序设计介绍”。

本章的实例利用了样例数据库 samp_db，这个实例使用了为学分保存方案而创建的表和第1章“MySQL 和 SQL 介绍”中为历史同盟而创建的表。虽然在写本书时 PHP 4还处于β测试阶段，而当您读到本书时它可能已经发布了，但这里还是要涉及 PHP 3的内容。与 PHP 3兼容是 PHP 4 的一个明确设计目标，因此这里所说的有关 PHP 3的各个方面的都可应用到 PHP 4中。从 PHP 3 更改到 PHP 4 有一组移植注意事项。如果使用 PHP 4，就应该阅读那些注释。

本章假设的环境是：PHP 与 Apache Server 结合使用。必须安装 MySQL C 客户机库和头文件，因为建立 PHP 时需要这些文件，否则 PHP 不知道如何访问 MySQL 数据库。如果需要获得一些这样的软件，请参阅附录 A“获得和安装软件”。附录中还给出了获得本章开发的实例脚本的说明，您可以直接下载这些脚本。

在 UNIX 环境下，PHP 或者作为内建模块用于 Apache，该模块连接到 Apache 可执行的二进制程序上；或者作为独立的用于传统 CGI 程序的解释程序。在 Windows 环境下，虽然开发在 Windows NT 环境下运行的 Apache PHP 4 模块的工作正在进行之中，但是此刻的 PHP 只能作为一个独立程序来运行。

本章的大部分篇幅，都用来说明了在这里讨论所需要的 PHP 函数。要想较全面地了解所有关于 MySQL 的函数，请参阅附录 H“PHP API 参考”，也可以查阅 PHP 手册，它描述了 PHP 提供的全部函数，包括使用除 MySQL 以外的数据库的函数（使用 MySQL 时，PHP 不受限制的部分仅仅只有 DBI）。这个手册可以从 PHP Web 站点下载：<http://www.php.net/>。该 Web 站点也有从 PHP 3 移植到 PHP 4 的注意事项。

8.1 PHP 脚本的特点

PHP 脚本的文件名带有扩展名，该扩展名允许 Web 服务器识别文件名并执行 PHP 解释程序去处理它们。如果使用了不可识别的扩展名，则 PHP 脚本被当作纯文本。本章使用的扩展名为 .php，其他通用的扩展名为 .php3 和 .html。有关配置 Apache 来识别扩展名的说明，请参阅附录 A。在机器上，如果没有掌握 Apache 的安装，那么需要与系统管理员一起检查，找出可使用的适当扩展名。

8.2 PHP 基础

PHP 的基本功能就是解释一个脚本，来生成发送到客户机的 Web 页面。具有代表性的是，脚本包括逐字发送到客户机的 HTML 和作为程序执行的 PHP 代码的混合编码。无论代码生成什么样的输出，都会发送到客户机，因此客户机永远不会看到代码，它只能看到

结果的输出。

当 PHP 开始读取文件时，假设文件内容表示文字的 HTML，则它仅仅拷贝在那里找到的输出内容。当 PHP 解释程序遇到一个特殊的打开标记时，就从 HTML 模式切换到 PHP 代码模式，而作为要执行的 PHP 代码也开始解释文件。代码的结尾由另一个特殊的标记指出，解释程序在这个位置从代码模式切换回 HTML 模式。这就允许将静态的文本（HTML 部分）与动态产生的结果（PHP 代码部分的输出）相混合，产生依赖于调用环境变化的页面。例如，可以使用 PHP 脚本来处理表格的结果，在这个格式中，用户已经输入了数据库搜索的参数。由于格式填入内容的不同，所以每次搜索的参数可能也不同，因此当脚本执行搜索的时候，每个作为结果的页面将反映不同的搜索。

让我们通过一个非常简单的 PHP 脚本看一看它是如何工作的：

```
<HTML>
<BODY>
hello, world
</BODY>
</HTML>
```

这个脚本并不很有趣，因为它不包括 PHP 代码！因此您会问：那它有什么好处？这个问题的回答是：它有时有助于建立包括想要生成页面的 HTML 框架的脚本，然后再加入 PHP 代码。这是非常有效的，PHP 解释程序用于它是没有问题的。

为了在脚本中包括 PHP 代码，您可从用两个特殊标记（脚本开始处的 ‘<?php’ 和脚本结束处的 ‘?>’）把它与周围的文本区分开来。当 PHP 解释程序遇到开始的 ‘<?php’ 标记时，就从 HTML 模式切换到 PHP 模式，并解释它找到的任何 PHP 代码，直到看见结束的 ‘?>’ 标记为止。它产生的所有输出解释并替换了两个标记之间的脚本。将前面的实例再重新编写一下，它包括了少量的 PHP 代码，如下所示：

```
<HTML>
<BODY>
<?php print ("hello, world\n"); ?>
</BODY>
</HTML>
```

此时，代码部分是很小的，由单行组成。当解释代码时，产生了输出 “hello, world”，它作为输出部分发送到客户机浏览器。这样，这个脚本产生的 Web 页面与前面实例产生的 Web 页面一样，前面实例的脚本完全由 HTML 组成。

可以使用 PHP 代码产生 Web 页面的任何部分。我们已经看到了一个特别的实例，在那里整个脚本都由文字的 HTML 组成，而不包括 PHP 代码。另一个特别的实例是整个脚本都是 PHP 代码而不包括文字的 HTML：

```
<?php
print ("<HTML>\n");
print ("<BODY>\n");
print ("hello, world\n");
print ("</BODY>\n");
print ("</HTML>\n");
?>
```

这说明 PHP 在如何产生输出方面有很大的灵活性。但 PHP 也留下一个问题，那就是确定如何组合 HTML 和 PHP 代码才是合适的。不必把所有代码都放在一个地方，PHP 在这方面也很灵活。只要您高兴，就可以通过脚本在 HTML 和 PHP 代码模式之间进行转换。

PHP 脚本标记

除了本章实例中使用的标记之外，PHP 还支持其他的脚本标记。您可以在其他人编写的 PHP 代码中看到它们，或者可以自己使用这些标记。PHP 识别四种标记风格：

■ 缺省标记风格。这是 PHP 配置为缺省时使用的风格：

```
<?php print ("hello, world\n"); ?>
```

■ 简洁开标记风格。这个风格除了开标记较简洁外，其他与缺省风格相类似：

```
<? print ("hello, world\n"); ?>
```

■ 兼容 ASP 的风格。这个风格在 Active Server Page 环境内部是通用的：

```
<% print ("hello, world\n"); %>
```

■ <SCRIPT> 标记风格。如果使用与其他标记风格不同的 HTML 编辑器，这个风格是有用的。当然，它比较冗长，但是当您使用其他标记风格时，编辑器可能不处理 PHP 代码，这时您将发现它是必需的：

```
<SCRIPT LANGUAGE="php"> print ("hello, world\n"); </SCRIPT>
```

缺省时，简洁开标记风格和兼容 ASP 风格无效。请参阅附录 H 有关开启它们的说明。

独立的 PHP 脚本

当处理命令解释程序脚本或 Perl 脚本时，可以编写从命令行调用的独立的 PHP 脚本。这里有一个实例：

```
#!/usr/local/bin/php -q
<?php print ("hello, world\n"); ?>
```

前面的脚本可命名为 hello.php，利用 chmod +x 使之成为可执行的，并从命令解释程序中调用：

```
% hello.php
hello, world
```

本章中我们不编写任何独立的脚本。这里编写的所有实例都期望 Web 服务器调用它们来生成 Web 页面。

下一个脚本有一些实质性的内容，但仍然相当简短。它说明了如何较容易地从 PHP 访问 MySQL 数据库，并在 Web 页面中使用查询结果。此脚本在第 5 章表达得很简短，它形成了历史同盟 Web 站点主页的基础。在我们继续往下做的时候，应该使脚本更精巧一些，但直到现在为止，它所做的只是显示简短的欢迎消息和当前同盟会员资格的计数：

```
<HTML>
<HEAD>
<TITLE>US Historical League</TITLE>
</HEAD>
<BODY>
<P>Welcome to the US Historical League Website.
<?php
$link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or exit ();
mysql_select_db ("samp_db")
    or exit ();
$result = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_array ($result))
    echo "<P>The League currently has " . $row[0] . " members";
```

```
mysql_free_result ($result);  
?>  
</BODY></HTML>
```

欢迎消息就是一个静态文本，因此它作为文字的 HTML 是最容易编写的。另一方面，会员资格的计算是动态的，而且随时会更改，因此必须在它不工作时通过查询 samp_db 数据库中的 member 表来确定。

在开和闭脚本标记之间的代码文本执行一个简单的任务。首先，它打开了与 MySQL 服务器的连接，并使 samp_db 数据库成为缺省数据库。然后，它发送一个到服务器的查询来确定此刻历史同盟有多少成员（我们将它作为 member 表中的行数）。查询结果将作为包括会员资格计数消息的一部分显示出来，然后再做处理。

在处理过程中，如果任何一点发生错误，该脚本都会简单地退出。它由于太简单而不产生错误输出，因此把访问该 Web 站点的人们搞糊涂了（如果依靠 PHP 代码生成整个 Web 页面，当因错误而退出，且不产生任何输出时，可能会使访问页面的人非常恼火，因为有些浏览器将显示“这个页面没有数据”的对话框）。

让我们把该脚本分成几段，看看它是如何工作的。第一步是用 mysql_pconnect() 连接到服务器：

```
$link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret")  
or exit ();
```

mysql_pconnect() 把主机名、用户名和口令作为参数。连接建立成功，则返回连接标识符，如果发生错误，则返回 FALSE。如果连接失败，脚本立刻调用 exit() 来结束脚本，并且不再产生输出。

在 mysql_pconnect() 调用前的 ‘@’ 字符是什么意思呢？就是“请关闭”的意思。有些 PHP 函数在失败时除了返回状态代码之外还写出错误消息。在 mysql_pconnect() 情况下，失败的连接导致了下面的消息出现在发送到客户机浏览器的 Web 页面上：

```
Warning: MySQL Connection Failed: Access denied for user:  
'paul@pit-viper.snake.net' (Using password: YES)
```

这样很难看，参观我们站点的人们可能不知道它是如何造成的和该怎样处理它。把 ‘@’ 字符放到 mysql_pconnect() 调用的前面，就可以取消这个错误消息，以使我们能在返回值的基礎上自己选择如何处理错误。对于该脚本，如果发生错误，最好根本不产生属于会员资格计数的输出。这样，页面将只包含欢迎消息。

任何 PHP 函数都可以使用 ‘@’，但以笔者的经验来说，初始的 mysql_pconnect() 调用是最可能失败的一个。因此，本章的例子禁止来自该函数中的消息。

将名称和口令嵌入到所有人都能看到的脚本中，可能会使您紧张不安。是这样的，名称和口令出现在发送到客户机的 Web 页面上是对的，因为该脚本的内容由其输出的结果替换掉了。然而，如果 Web 服务器不知何故配置不当，没有识别出脚本需要由 PHP 代码来处理，它就会将脚本以纯文本发送出去，且会暴露连接参数。在 8.2.1 节“使用函数和 include 文件”中，将简要地处理这类情况。

mysql_pconnect() 返回的连接标识符可以传递到 PHP API 中的几个与 MySQL 相关的调用中。然而，对于这样的调用，标识符总是可选择的。例如，可以使用下列格式之一来调用 mysql_select_db()：

```
mysql_select_db ($db_name, $link);
```

```
mysql_select_db ($db_name);
```

mysql_pconnect() 与 mysql_connect() 的对比

函数 `mysql_pconnect()` 与函数 `mysql_connect()` 相似，都具有主机名、用户名和口令参数，并返回连接标识符或 `FALSE` 来说明连接是否成功。两个调用之间的不同在于：`mysql_pconnect()` 建立了一个持久的连接，而 `mysql_connect()` 建立了一个非持久的连接。与非持久连接不同，持久连接在脚本终止时不关闭。如果另外一个 PHP 脚本随后由同一个 Apache 子处理执行，并用同样的参数调用 `mysql_pconnect()`，将重新使用这个连接。这比关闭后再建立连接的效率要高。

如果忽略了从一些相关 MySQL 的 PHP 调用中得到的连接参数，调用就会使用最近打开的连接。这样，如果脚本只打开单个连接，那么在任何 MySQL 调用中永远不必指定连接参数——即连接是缺省的。这就是 C 或 DBI API 与 MySQL 程序设计的极大不同之处，因为它们没有这样的缺省。

笔者使用 `$link` 变量在简单的主页上编写了下面的连接代码，使 `mysql_pconnect()` 返回哪种类型的值更加清晰：

```
$link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret")  
or exit ();
```

然而，实际上，我们在脚本的其他地方都没有使用 `$link`，因此代码可更简单地写成：

```
@mysql_pconnect ("pit-viper.snake.net", "paul", "secret")  
or exit ();
```

假设连接建立成功，则下一步是选择一个数据库：

```
mysql_select_db ("samp_db")  
or exit ();
```

如果 `mysql_select_db` 失败，我们将会默默地退出。如果我们能够连接到服务器，并且数据库存在，那么似乎不可能发生错误，但是仍然要严格地检查问题并采取相应的行动。选择数据库之后，可将查询发送到服务器，提取结果，加以显示，然后释放结果集：

```
$result = mysql_query ("SELECT COUNT(*) FROM member")  
or exit ();  
if ($row = mysql_fetch_array ($result))  
    echo "<P>The League currently has " . $row[0] . " members";  
mysql_free_result ($result);
```

`mysql_query()` 函数将查询发送到服务器中去执行。查询不用分号或者 `'\g'` 终止。如果查询非法或因为某些原因不能执行，则 `mysql_query()` 返回 `FALSE`，否则返回一个结果集标识符。该标识符是我们能用来获得有关结果集信息的值。对于查询，该结果集由表示会员资格计数的单列值的单行组成。为得到这个值，我们可以把结果集标识符传给 `mysql_fetch_row()` 来获取行，将此行赋给变量 `$row`，并以 `$row[0]` 形式访问第一个元素（只有一个元素时也是这样）。

当处理完结果集时，将结果集传递给 `mysql_free_result()` 进行释放。实际上这种调用在我们的脚本中是不必要的，因为当脚本结束时，PHP 会自动地释放所有活动的结果集。

`mysql_free_result()` 有助于执行大型查询或大量查询的脚本。它防止大量内存的使用。

为了使用脚本，需要在某处安装它。本章将采用这样的约定：美国历史同盟在 Apache 文档树的最高一级中有自己的目录，称为 `ushl`，因此主页面脚本作为该树的 `ushl/index.php` 进行

安装。我们也将为学分保存方案开发脚本，因此给出目录 gp。如果 Web 站点主机是 pit-viper.snake.net，那么这两个目录中的页面将有如下开头的 URL：

```
http://pit-viper.snake.net/ush1/  
http://pit-viper.snake.net/gp/
```

例如，每个目录的主页面都可称为 index.php，并以如下方式进行访问：

```
http://pit-viper.snake.net/ush1/index.php  
http://pit-viper.snake.net/gp/index.php
```

8.2.1 使用函数和include 文件

PHP 脚本与 DBI 脚本的不同之处在于，PHP 脚本位于 Web 服务器文档树的内部，而 DBI 脚本位于 cgi-bin 目录中，这个目录在文档树的外部。这就提出了一个安全性问题：服务器配置不当的错误可能导致位于文档树内部的页面会以纯文本方式泄露给客户机。这意味着建立与 MySQL 服务器连接的用户名和口令如果在 PHP 脚本而非 DBI 脚本中使用，则将处于暴露给外界的很高的危险之中。

PHP 中的变量

在PHP中，可以通过简单地使用变量而使它们存在。主页脚本使用了三个变量：`$link`、`$result` 和 `$row`，没有一个变量会在所有地方都声明（声明变量的地方有上下文，如在函数内部引用全局变量时，我们随后会谈这个问题）。

变量由美元符号（‘\$’）为开头的标识符表示。无论它表示什么类型的值都是正确的，尽管对于数组和对象要添加一些额外内容来访问值的单个元素。如果变量 `$x` 表示单个值，例如，数字或字符串，则可以写成 `$x` 来访问它。如果 `$x` 表示有数字索引的数组，则可以写成 `$x[0]`、`$x[1]` 等等来访问它的元素。如果 `$x` 表示有关联索引的数组，如“yellow”或者“large”，则可以写成 `$x[“yellow”]`、`$x[“large”]` 来访问它的元素。

PHP 数组可以同时拥有数字索引的元素和相关索引的元素。例如，`$x[1]` 和 `$x[“large”]` 都能作为同一数组的元素。如果 `$x` 代表一个对象，则可写成 `$x->property_name` 来访问它所具有的属性。例如，`$x->yellow` 和 `$x->large` 都是 `$x` 的属性。作为属性名，数字是不合法的，因此 `$x->1` 在 PHP 中是不合法的。

初始的历史同盟主页脚本也存在这个问题，因为它包括 MySQL 用户名和口令的直接值。让我们用两个 PHP 性能：函数和 include 文件，把这些连接参数移到脚本外面。我们将编写函数 `samp_db_connect()`来建立这个连接，并把函数放到一个 include 文件中——不是主脚本部分的，但可以从脚本中引用的文件。这种方法的优点如下：

- **编写连接建立代码比较容易。** 不需要写出所有参数，就可以在连接后用 `samp_db_connect()` 选择数据库，使一个函数可以进行两个 PHP 函数的工作。由于你可以将精力集中于脚本的独特标记，而不必为连接建立代码分心，因此也使得脚本更加易于理解。
- **可从脚本中访问 include 文件，但可移到 Apache 文档树的外面。** 这使它的内容对于客户机来说是不可访问的，即使 Web 服务器配置不当，连接参数也不会暴露给它们。使用 include 文件，对于隐藏不想由 Web 服务器发送到站点的任何类型的敏感信息都是个良策。

虽然如此，但这并不意味着用户名和口令在任何意义上说都是安全的。如果没有采取预防措施，在 Web 服务器主机上注册的用户，能够直接读取 include 文件。请参阅

7.4.3节“从 Web 脚本连接到 MySQL 服务器”关于安装 DBI 配置文件所描述的预防措施，它们用于保护口令和用户名不受其他用户侵害。对 PHP 的 include 文件也要应用同样的防范措施。

PHP 语言上的影响

如果有 C 程序设计经验，则可能注意到：脚本中许多语法的结构与 C 程序设计中的非常类似。实际上，PHP 语法很大程度上来自于 C，因此这种相似处并不是巧合。如果有些 C 的背景知识，就可以将它的许多内容转换到 PHP。事实上，如果不能确信如何用 PHP 编写表达式或控制结构，则可以试用 C 中编写它们的方法，这很可能也是正确的。虽然 PHP 的基本部分主要在 C 中，但也包含了使用 Java 和 Perl 的成分。可以在注释语法中查看它，在那里允许以下形式：

```
# Perl-style comment from '#' to end of line
// C++- or Java-style comment from '//' to end of line
/* C-style comment between slash-star to star-slash */
```

Perl 的其他相似性包括 ‘.’ 字符串连接操作符（包括 ‘.=’ 作为额外的连接），变量引用和转义序列的方法是在双引号内而非单引号内解释的。

■ include 文件可以由多个脚本使用。这提高了代码的可重用性，使代码更加可维护。同时也允许对访问这个文件的每个脚本不费力地做出全局性的更改。例如，如果我们将数据库 samp_db 从 pit-viper 移动到 boa，则不必更改一簇单个脚本，而只要更改包含 samp_db_connect() 函数的 include 文件中 mysql_pconnect() 调用的主机名参数即可。

为了使用 include 文件，必须有存放它们的地方，而且必须使 PHP 找到它们。如果系统已经有了这样一个位置，则可以使用。如果没有，则使用下面的过程建立一个 include 文件的位置：

- 1) 创建一个目录来存放 PHP 的 include 文件。该目录不能位于 Web 服务器文档树内部！笔者使用 /usr/local/apache/php 的 PHP include 目录，它与我的文档树在同一层次上 (/usr/local/apache/htdocs)，而不是在其内部。

- 2) 通过完整的路径名或者告诉 PHP 在搜索时寻找哪个目录来引用 include 文件。后者的方法更方便些，因为如果我们使用了文件的基名，PHP 就会找到它（PHP include 文件与 C 头文件有些类似，其中包括的 PHP 将在多个目录中搜寻 include 文件，就像 C 预处理程序在多个目录中搜寻 C 头文件一样）。为了告诉 PHP 去哪里查看，修改 PHP 初始化文件（系统上的 /usr/local/lib/php3.ini）来改变 include_path 的值。如果它没有值，可以将它设置为新的包含路径的完整路径名：

```
include_path = "/usr/local/apache/php"
```

如果 include_path 已经有值了，则把新的目录加到那个值中：

```
include_path = "current_value:/usr/local/apache/php"
```

- 3) 创建想使用的 include 文件并将它放到 include 目录中。文件应该有一些有特点的名称，为了这个目的，这里我们使用 samp_db.inc。它的内容将在下面列出。对于我们这里开发的脚本，当连接到 MySQL 服务器上时，会一直使用 samp_db 数据库，因此连接函数 samp_db_connect() 也可以为我们选择 samp_db 数据库。如果连接成功并选择了这个数据库，这个函数就返回一个连接标识符；如果发生错误，则返回 FALSE。发生错误时将不打印消息，

并且允许调用者静静地退出，或者在环境允许时再打印消息。

```
<?php
# samp_db.inc
# samp_db sample database common functions
# Connect to the MySQL server using our top-secret name and password

function samp_db_connect ()
{
    $link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret");
    if ($link && mysql_select_db ("samp_db"))
        return ($link);
    return (FALSE);
}
?>
```

观察一下，`samp_db.inc` 文件的内容由 ‘`<?php`’ 和 ‘`?>`’ 括在一起。这是因为 PHP 是在 HTML 模式中开始读取文件的。如果没有这些标记 PHP 会把文件以纯文本发送出去，而不是作为 PHP 代码解释。如果想在文件中包含文字的 HTML 是很好的选择。但是，如果文件包含 PHP 代码，就必须在脚本标记内部封闭代码。

4) 使用下面的行从脚本中引用文件：

```
include ("samp_db.inc");
```

当 PHP 看到这一行时，就搜寻文件并读取内容。对于脚本的下列部分，文件中的任何事物都变成可访问的。

在建立了我们的 `include` 文件 `samp_db.inc` 之后，就可以修改历史同盟主页来引用 `include` 文件，并通过调用 `samp_db_connect()` 函数连接到 MySQL 服务器上：

```
<HTML>
<HEAD>
<TITLE>US Historical League</TITLE>
</HEAD>
<BODY>
<P>Welcome to the US Historical League Website.
<?php
include ("samp_db.inc");

samp_db_connect ()
    or exit ();
$result = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_array ($result))
    echo "<P>The League currently has " . $row[0] . " members";
mysql_free_result ($result);
?>
</BODY></HTML>
```

`include()` 与 `require()` 的对比

PHP 的 `require()` 性能与 `include()` 相类似。不同之处在于，对 `include()` 来说，在 `include()` 执行时文件每次都要进行读取和评估；而对于 `require()` 来说，文件只处理一次（实际上，文件内容替换了 `require()` 语句）。这意味着如果有包含这些指令之一的代码和可能执行多次的代码，则使用 `require()` 的效率比较高。另一方面，如果每次执行代码时想读取不同的文件，或者有通过一组文件迭代的循环，就使用 `include()`，因为可以给想要包括的文件名设置一个变量，当参数为 `include()` 时使用这个变量。

samp_db.inc 文件对其他函数也是有用的，我们可以将它作为各种其他事物的储藏库。实际上，还可以再创建两个函数放入到文件中。我们编写的每个脚本在页面的开头都会产生一组相当醒目的 HTML 标记，而另一组在结尾。不必在每个脚本中将它们逐字地写出，我们可以通过编写函数 html_begin() 和 html_end() 来做这些事。函数 html_begin() 能够提取几个指定了页面标题和头的参数。两个函数的代码如下：

```
# Put out initial HTML tags for page. $title and $header, if
# present, are assumed to have any special characters properly
# encoded.
```

```
function html_begin ($title, $header)
{
    print("<HTML>\n");
    print("<HEAD>\n");
    if ($title)
        print("<TITLE>$title</TITLE>\n");
    print("</HEAD>\n");
    print("<BODY>\n");
    if ($header)
        print("<H2>$header</H2>\n");
}
```

```
# put out final HTML tags for page.
```

```
function html_end ()
{
    print("</BODY></HTML>\n");
}
```

然后我们可以修改历史同盟主页来使用这两个新函数，如下所示：

```
<?php
include ("samp_db.inc");

$title = "US Historical League";
html_begin ($title, $title);
?>
<P>Welcome to the US Historical League Website.

<?php
samp_db_connect ()
    or exit ();
$result = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_array ($result))
    echo "<P>The League currently has " . $row[0] . " members";
mysql_free_result ($result);

html_end ();
?>
```

请注意代码被分成了两块，两块代码之间出现了欢迎消息的文字 HTML 文本。

产生页面开始和最后部分的函数用法给了我们一个重要的能力。如果想改变使用这些函数的页面头和尾的外观，可以在函数中包含一些代码。使用它们的每个脚本也都将自动地受到影响。例如，您可以把消息“Copyright USHL”放在每个历史同盟页面的底部。页面尾部函数 html_end() 会很容易地做到这一点。

8.2.2 一个简单的查询页面

已经嵌入到历史同盟主页中的脚本运行了一个只返回单个行的查询。下一个脚本介绍了如何处理多行的结果集。它获取并显示了 member 表中的内容。这就是第 7 章开发的 dump_members DBI 脚本的 PHP 等价物，因此称它为 dump_members.php。它与 DBI 版本的不同之处在于，它希望在 Web 环境中使用而不是在命令行中使用。由于这个原因，它需要产生 HTML 输出而不是简单地写出制表符分隔的文本。为了使行和列漂亮地排列，我们将以 HTML 表形式编写会员资格记录。脚本如下：

```
<?php
# dump_members.php - dump Historical League's membership list

include ("samp_db.inc");

$title = "US Historical League Member List";
html_begin ($title, $title);

samp_db_connect ()
    or die ("Cannot connect to server");

# issue query
$query = "SELECT last_name, first_name, suffix, email,"
        . "street, city, state, zip, phone FROM member ORDER BY last_name";
$result = mysql_query ($query)
    or die ("Cannot execute query");

print("<TABLE>\n");
# read results of query, then clean up
while ($row = mysql_fetch_row ($result))
{
    print("<TR>\n");
    for ($i = 0; $i < mysql_num_fields ($result); $i++)
    {
        # escape any special characters and print
        printf("<TD>%s</TD>\n", htmlspecialchars ($row[$i]));
    }
    print("</TR>\n");
}
mysql_free_result ($result);
print("</TABLE>\n");

html_end ();
?>
```

这个脚本使用了 die() 函数来打印消息，如果发生错误则退出（die() 函数与 exit() 函数类似，但是它在退出之前打印消息）。与我们在历史同盟主页中使用的静静地退出方法相比，这是一种不同的错误处理方法。在 dump_membes.php 中，我们希望看到一个特殊的结果，因此打印错误消息来说明发生的问题是有道理的。

脚本可以安装在 ushl 目录并用 http://pit-viper.snake.net/ushl/dump_members.php 访问。可以在历史同盟主页的新脚本中增加一个连接，以便人们了解它：

```
<?php
include ("samp_db.inc");

$title = "US Historical League";
html_begin ($title, $title);
```

```
?>

<P>Welcome to the US Historical League Website.

<?php
samp_db_connect ()
    or exit ();
$result = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_array ($result))
    echo "<P>The League currently has " . $row[0] . " members.";
mysql_free_result ($result);
?>

<P>
You can view the directory of members
<A HREF="dump_members.php">here</A>.

<?php
html_end ();
?>
```

8.2.3 处理查询结果

在这一节中，我们将更细致地检查如何执行 MySQL 查询并处理结果集。在 PHP 中，所有的查询都通过调用 `mysql_query()` 函数来发布，这个函数提取查询字符串和连接标识符作为参数。连接标识符是可选的，因此可以用下面任意一种形式调用 `mysql_query()`：

```
$result = mysql_query ($query, $link);    # use explicit connection
$result = mysql_query ($query);          # use default connection
```

对于不返回行的查询（非 `SELECT` 的查询，如 `DELETE`、`INSERT`、`REPLACE` 和 `UPDATE`），`mysql_query()` 返回 `TRUE` 或者 `FALSE` 说明查询成功或者失败。为了获得成功的查询，可以调用 `mysql_affected_rows()` 找出有多少行被改变（可能是删除、插入、替换或者更新）。

对于 `SELECT` 语句，`mysql_query()` 返回结果集标识符或者 `FALSE` 说明查询是成功或是失败。为了获得成功的查询，使用结果集标识符可以获得更多的有关结果集的信息。例如，可以找出结果集有多少行或列，或者提取这个结果集内部包括的行。

当 `mysql_query()` 返回 `FALSE`（也就是零）时，意味着查询失败——换句话说就是发生一些错误而不能执行查询。查询失败可能有下面几个原因：

- 查询可能是畸形的并包含语法错误。
- 查询依照语法可能是正确的，但是在语义上却是无意义的，例如要从不包含列的表中选择列时。
- 没有充分的权利执行查询。
- 由于网络问题，可能已经连接不到 MySQL 服务器主机。

在以上每种情况中（还有其他情况），`mysql_query()` 都返回 `FALSE`。如果想要知道错误的详细原因，就调用 `mysql_error()` 或者 `mysql_errno()` 获得错误消息字符串或者数字错误代码（请参阅8.2.4节“处理错误”一节）。

考虑 `mysql_query()` 造成的两种最常见的错误，返回值是行计数，或者它包含查询返回的数据，两者都是错误的。

1. 处理不返回结果集的查询

下面的代码使用 DELETE 说明了如何处理不返回任何行的查询：

```
$result = mysql_query ("DELETE FROM member WHERE member_id = 149");
if (!$result)
    print ("query failed\n");
else
    printf ("number of rows deleted: %d\n", mysql_affected_rows ());
```

如果有一个 ID 号为 149 的成员，MySQL 就删除记录，且 mysql_query() 返回 TRUE。如果没有这样的成员呢？这种情况下 mysql_query() 仍然返回 TRUE！这使将精力耗在误解 mysql_query() 的返回值是行计数的人们非常惊讶。两种情况都返回 TRUE，是因为不管实际上是否删除了一些行，查询都是合法的。由查询作用的行的数目则是完全不同的事。若要在查询成功之后提取值，可调用 mysql_affected_rows()。

2. 处理返回结果集的查询

下面的实例提供了 SELECT 查询处理的大致概况：

```
$result = mysql_query ("SELECT * FROM member");
if (!$result)
    print ("query failed\n");
else
{
    print ("number of rows returned: %d\n", mysql_num_rows ($result));
    while ($row = mysql_fetch_row ($result))
    {
        for ($i = 0; $i < mysql_num_fields ($result); $i++)
        {
            if ($i > 0)
                print (",");
            print ($row[$i]);
        }
        print ("\n");
    }
    mysql_free_result ($result);
}
```

不要假设 mysql_query() 会成功

在 PHP 邮件发送清单中，新的 PHP 用户会共同询问：执行脚本时为什么会发生下面的错误消息：

```
Warning: 0 is not a MySQL result index in file on line n
```

这个消息将为零的结果集标识符的值传递给了期望有效结果集的一些函数（如提取行的函数）。这意味着早时的 mysql_query() 调用返回了零值——就是 FALSE。换句话说就是 mysql_query() 失败，并且在由其他函数使用它之前，脚本编写程序对检查返回值并不烦恼。在使用 mysql_query() 时，要一直检测返回值。

如果查询失败，结果将为 FALSE，对这个结果我们只是打印一条消息（取决于环境，其他对错误的反应可能更合适）。如果查询成功，则 mysql_query() 返回结果集标识符。这个返回值在许多方法中是有用的（但不是作为行计数！）。结果集标识符可用于下列目标：

- 将它传递给 mysql_num_rows()，来确定结果集中的行数。
- 将它传递给 mysql_num_fields()，来确定结果集中的列数。
- 将它传递给提取行的例程，来提取结果集的连续行。这个实例使用了 mysql_fetch_

row(), 但是还有其他的选择, 我们马上就会看到它们。

- 将它传递给 `mysql_free_result()`, 来释放结果集并允许 PHP 处理一些与之相关的源文件。

在 `mysql_query()` 成功地执行 SELECT 查询之后 (请参见表 8-1), PHP 为检索结果集提供了几个提取行的函数。当不再有行时, 每个函数都得到一个结果集标识符作为参数并返回 FALSE。

表8-1 PHP 行提取函数

函 数 名	返 回 值
<code>mysql_fetch_row()</code>	一个数组, 由数字索引访问其元素
<code>mysql_fetch_array()</code>	一个数组, 由数字索引或相关索引访问其元素
<code>mysql_fetch_object()</code>	一个对象, 作为属性访问其元素

最基本的调用是 `mysql_fetch_row()`, 它返回结果集的下一行作为一个数组。数组的元素通过从 0 到 `mysql_num_fields()-1` 范围内的数字索引访问。下面的实例说明了如何在每一行都提取和打印值的简单循环中使用 `mysql_fetch_row()` :

```
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_row ($result))
{
    for ($i = 0; $i < mysql_num_fields ($result); $i++)
    {
        if ($i > 0)
            print ("\t");
        print ($row[$i]);
    }
    print ("\n");
}
```

变量 `$row` 是一个数组。可用 `$row[$i]` 访问它, `$i` 在这里是数字列索引。如果熟悉 PHP `count()` 函数, 可以试着用它而不要用 `mysql_num_fields()` 来确定每一行的列数。`count()` 只计算这个数组中已设置值的元素的数量, PHP 不是与 NULL 列值相对应的元素设置值。`count()` 对返回列数的度量是不可靠的, 因为那不是它想要的。它还用于另外两种提取行的函数。

第二个提取行的函数 `mysql_fetch_array()` 在表 8-1 中列出, 它与 `mysql_fetch_row()` 相类似, 但是由数字索引和相关索引返回的数组元素都是可靠的。换句话说, 可以通过数字或名称访问元素:

```
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_array ($result))
{
    printf ("%s %s\n", $row[0], $row[1]);
    printf ("%s %s\n", $row["first_name"], $row["last_name"]);
}
```

由 `mysql_fetch_array()` 返回的信息是 `mysql_fetch_row()` 返回的信息的扩展集。除此之外, 两个函数之间的不同性能是可以忽略不计的, 调用 `mysql_fetch_array()` 可以无特殊性能损耗。

第三个提取行的函数 `mysql_fetch_object()`, 返回结果集的下一行作为对象, 这意味着用

`$row->col_name` 语法访问行的元素。例如，如果从 `President` 表中检索 `last_name` 和 `first_name` 值，可以用 `$row->last_name` 和 `$row->first_name` 访问那些列：

```
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_object ($result))
    printf ("%s %s\n", $row->first_name, $row->last_name);
```

在查询结果中测试 NULL 值

可以使用 `isset()` 函数测试 `SELECT` 查询返回的列值是否为 `NULL`。例如，如果行包含在 `$row` 数组中，那么如果 `$row[$i]` 对应于 `NULL` 值，则 `isset($row[$i])` 就为 `FALSE`，如果 `$row[$i]` 为非 `NULL` 值，则 `isset($row[$i])` 就为 `TRUE`。相关的函数是 `empty()`，但是对于 `NULL` 和空字符串，`empty()` 返回的结果都是一样的，因此作为 `NULL` 值测试而言，这个函数是无用的。

如果查询包括计算的列怎么办？例如，发布一个作为表达式结果计算的返回值的查询：

```
SELECT CONCAT(first_name, " ", last_name) FROM president
```

这样编写的查询不适于使用 `mysql_fetch_object()`。选择的列名本身就是表达式，它不是合法的属性名。然而，可以通过给列赋予一个别名来提供合法的名称。下面的查询将列的别名命名为 `full_name`，如果用 `mysql_fetch_object()` 提取结果，就允许它以 `$row->full_name` 形式来访问：

```
SELECT CONCAT(first_name, " ", last_name) AS full_name FROM president
```

8.2.4 处理错误

PHP提供了三种处理错误的方法：

■ **用 ‘@’ 取消错误消息。** 可以对一些显示消息的函数使用 ‘@’。当我们调用 `mysql_pconnect()` 阻止来自于函数的错误消息不在发送到客户机的页面上出现时，就已经在做这一点了。

■ **使用 `error_reporting()` 函数。** 这个函数按下列级别将错误报告打开或者关闭：

错误级别	错误报告类型
1	正常函数错误
2	正常警告
4	分析程序错误
8	请注意

为了控制错误报告，可调用 `error_reporting()` 函数，且参数为想要激活的级别的总和。关闭级别1和级别2警告应该完全能够取消来源于 `MySQL` 的消息：

```
error_reporting (4 + 8);
```

你可能不想关闭有关分析错误的级别4警告，如果关掉了，可能要有一段艰难的时间用来调试它对脚本造成的改动！级别8警告常常被忽略，但有时它指出脚本中应该注意的问题，因此您可能也想把它激活。还有16和32错误级别，它们都来自于 `PHP` 核心发动机，而非函数，因此通常情况下不必考虑它们。

■ **使用 `mysql_error()` 和 `mysql_errno()`。** 这些函数报告了 `MySQL` 服务器返回的错误信息。它们与相应的 `C` API 调用相类似。`mysql_error()` 以字符串形式返回错误信息（如

果不发生错误就返回空字符串)。mysql_errno() 返回一个错误数字（如果不发生错误就返回0）。两个函数都有指定与 MySQL 服务器连接的连接标识符参数，在返回状态的连接上都返回最近调用的 MySQL 函数的错误信息。连接标识符是可选的，如果缺失，就使用最近打开的连接。例如，可以这样报告 mysql_query() 的错误：

```
way:
if (!($result = mysql_query ( ... )))
{
    print ("errno: " . mysql_errno());
    print ("error: " . mysql_error());
}
```

mysql_error() 和 mysql_errno() 的 PHP 版本在一个重要方面与 C API 中对应的部分不同。在 C 中，即使试图连接服务器失败，也会得到错误信息。相反，PHP 调用直到连接建立成功，才返回有用的连接信息。换句话说，如果连接失败，就不能用 mysql_error() 和 mysql_errno() 报告失败原因。如果要报告连接失败的特殊原因而不是普通原因，则必须做特殊的考虑。请参阅附录 H “PHP API 参考”，其中详细介绍了如何做。

当检测到错误时，本章的脚本打印了相当普通的错误信息，如“查询失败”。然而在开发脚本时，您会发现：加入一个 mysql_error() 调用对帮您发现错误的特殊原因是很有用的。

8.2.5 引用问题

在 PHP 中构造查询字符串时，知道引用问题是必要的，就像在 C 和 Perl 中一样。虽然函数名在各种语言中是不同的，但处理引用问题的方法却是类似的。假设正在构造一个，将新的记录插入到表中的查询，可以在值的前后加上引号插入到字符串列中：

```
$last = "O'Malley";
$first = "Brian";
$expiration = "2002-9-1";
$query = "INSERT member (last_name,first_name,expiration)"
        . " VALUES('$last','$first','$expiration')"
```

这里的问题是引用值的本身还包含着引号（“O’Malley”），如果将查询发送到 MySQL 服务器会导致语法错误。在 C 中我们调用 mysql_escape_string() 解决这个问题。在 Perl DBI 脚本中则使用 quote()。PHP 有一个 addslashes() 函数可以完成同样的事情。例如，调用 addslashes (“O’Malley”) 返回 “O’Malley” 值。将前面的实例做如下编写来解决引用问题：

```
$last = addslashes ("O'Malley");
$first = addslashes ("Brian");
$expiration = addslashes ("2002-9-1");
$query = "INSERT member (last_name,first_name,expiration)"
        . " VALUES('$last','$first','$expiration')"
```

DBI quote() 方法是把前后的引号加到字符串中。addslashes() 则不是，因此我们仍需在查询字符串中要插入值的周围将那些引号显式地指定出来。

当编写信息出现在 Web 页面上时也将发生引用问题。如果正在编写一个将作为 HTML 或 URL 的部分出现的字符串，而且这个字符串包含 HTML 或 URL 内部的特殊字符，最好将它编码。PHP 函数 htmlspecialchars() 和 urlencode() 可以做到这点，它们与 CGI.pm escapeHTML() 和 escape() 方法相类似。

8.3 运行 PHP

在本章的剩余部分中，将解决我们在第 1 章中为自己定下的目标：

- 对于学分保存方案，需要编写一个允许我们输入、编辑测试和测验分数的脚本。
- 对于历史同盟，需要开发一个有关美国总统的联机测验，使之成为交互式的，以便不做任何事情就可以为这个 Web 站点的访问者产生试题。
- 我们也想允许历史同盟成员联机编辑它们的目录项，使信息维持最新并减少我们自己编辑项的数量。

每个脚本都产生多个 Web 页面，并依靠在创建页面中嵌入的信息来在脚本的调用之间进行通信。

8.3.1 输入学生分数

在本节中，我们将把注意力转向学分保存方案。我们 Web 站点上的这个区域的 URL 是 `http://pit-viper.snake.net/gp/`，应该为它编写一个简短的主页 `index.php`，下面的页面就正在做这件事。它包括了与第 7 章编写的 `score_browser` 脚本的连接，因为这个脚本适合于学分保存方案。

```
<?php
include ("samp_db.inc");

$title = "Grade-Keeping Project";
html_begin ($title, $title);
?>

<P>
<A HREF="/cgi-bin/score_browser">View</A> scores for quizzes and tests

<?php
html_end ();
?>
```

现在让我们考虑如何设计和实现脚本 `score_entry.php`，它将让我们输入一组新的测试或测验分数，或者修改一组已经存在的分数。后者的性能对于处理由于生病或者其他原因缺席（或者，放弃这个想法以免输入分数失败）造成考试或测验比其他学生晚的学生的分数是必要的。分数项脚本的概要是这样的：

- 1) 最初的页面代表一系列已知的登记事件，并允许选择一个事件或者指定应该创建的新事件。
- 2) 如果选择创建一个新事件，脚本就给出允许指定日期和事件类型的页面。创建这个事件记录之后，脚本重新显示事件列表页面来显示这个新事件。
- 3) 当选择了事件后，脚本给出在顶部（事件 ID、日期、类型）显示事件信息的分数项页面，后接每个学生一项的列表。对于新事件，项将是空白的。对于已存在的事件，项将显示每个学生已存在的分数。选择提交按钮时，分数输入到 `score` 表中。

脚本需要执行几个不同的操作，这意味着我们需要从一个页面到另一个页面周而复始地传递状态变量，以便脚本在每次调用时能够知道假设要做什么。在 PHP 中很容易做到这一点，因为 PHP 处理作为 URL 参数传递的变量，并把它们转换为与参数具有相同名称的变量。例如，可以在脚本 URL 的末尾对参数 `action` 进行如下编码：

`http://pit-viper.snake.net/gp/score_entry.php?action=value`

当调用 `score_entry.php` 时, 参数 `action` 作为变量 `$action` 来编码, 这样就可以直接访问它了。这也适用于格式中的域。设想一个包括域 `name` 和 `address` 的表格, 当客户机传递表格时, Web 服务器就调用脚本访问表格的内容。脚本能够找出通过检查变量 `$name` 和 `$address` 的值而输入到表格中的值。对于包括许多域的表格, 全部给出唯一的命名是有困难的。PHP 很容易地把数组在表格中传入和传出。如果使用了如 `x[0]`、`x[1]` 等等的域名, 则 PHP 把它们作为 `$x` 数组的元素进行编码。可以将这些元素作为 `$x[0]`、`$x[1]` 等等来访问。

我们通过使用页面中的 `action` 参数, 可以将信息从 `score_entry.php` 脚本的一个调用传送到另一个调用, 并在脚本中用变量 `$action` 检查它的值。脚本的框架是这样的:

```
<?php
# score_entry.php - Score Entry script for grade-keeping project

include ("samp_db.inc");

# define action constants
define (INITIAL_PAGE, 0);
define (SOLICIT_EVENT, 1);
define (ADD_EVENT, 2);
define (DISPLAY_SCORES, 3);
define (ENTER_SCORES, 4);

/* ... put functions here ... */

$title = "Grade-Keeping Project - Score Entry";
html_begin ($title, $title);

samp_db_connect()
    or die ("Cannot connect to database server");

if (empty ($action))
    $action = INITIAL_PAGE;

switch ($action)      # what are we supposed to do?
{
case INITIAL_PAGE:    # present initial page
    display_events ();
    break;
case SOLICIT_EVENT:   # ask for new event information
    solicit_event_info ();
    break;
case ADD_EVENT:       # add new event to database
    add_new_event ();
    display_events ();
    break;
case DISPLAY_SCORES:  # display scores for selected event
    display_scores ();
    break;
case ENTER_SCORES:    # enter new or edited scores
    enter_scores ();
    display_events ();
    break;
default:
    die ("Unknown action code ($action)");
}
```

```
html_end ();
?>
```

变量 \$action 可以取若干值，我们已在 switch() 语句中测试过了（为避免在脚本中使用文字的数字，可以用 PHP 的 define() 构造来定义常量）。PHP switch() 语句与它在 C 中相应的部分相类似。在 score_entry.php 中，它用来确定采用什么操作，并且调用实现这个操作的函数。

检查一下每次处理一个操作的函数。第一个函数 display_events(), 检索来自 MySQL 的 event 表的行并加以显示。表的每一行都列出了事件 ID、日期和时间类型（测试或测验），还有编写事件 ID 作为可以选择用来修改事件分数的连接：

```
function display_events ()
{
    global $PHP_SELF;

    print ("Select an event by clicking on its number,\n");
    print ("or select New Event to create a new grade event:<BR><BR>\n");
    $query = "SELECT event_id, date, type"
        . " FROM event"
        . " ORDER BY event_id";
    $result = mysql_query ($query)
        or die ("Cannot execute query");
    print ("<TABLE BORDER>\n");
    print ("<TR>\n");
    display_cell ("TH", "Event ID", 1);
    display_cell ("TH", "Date", 1);
    display_cell ("TH", "Type", 1);
    print ("</TR>\n");

    # associate each event id with a link that will show the
    # scores for the event; use mysql_fetch_array() so we
    # can refer to columns by name.
    while ($row = mysql_fetch_array ($result))
    {
        print ("<TR>\n");
        $url = sprintf ("%s?action=%d&event_id=%d",
            $PHP_SELF, DISPLAY_SCORES, $row["event_id"]);
        display_cell ("TD",
            "<A HREF=\"$url\">" . $row["event_id"] . "</A>",
            0);
        display_cell ("TD", $row["date"], 1);
        display_cell ("TD", $row["type"], 1);
        print ("</TR>\n");
    }
    # add one more link for "new event"
    print ("<TR ALIGN=CENTER>\n");
    $url = sprintf ("%s?action=%d", $PHP_SELF, SOLICIT_EVENT);
    display_cell ("TD COLSPAN=3",
        "<A HREF=\"$url\">" . "New Event" . "</A>",
        0);
    print ("</TR>\n");

    print ("</TABLE>\n");
}
```

表中的连接用 \$PHP_SELF 来构造。这个变量包括了脚本自己的 URL，它为脚本再次调用自己提供了一个方便的方法。然而，请注意函数开始处的 global 行：

```
global $PHP_SELF;
```

在 PHP 函数中，全局变量是不可访问的，除非显式地声明要使用它们。没有 `global` 行，`$PHP_SELF` 将被看成局部变量（因为我们没有将值赋给它，因此是空的）。在函数内部，使用 `global` 来访问依靠 URL 参数或者作为表格域传递到脚本中的参数也是必需的。

用来生成表的函数 `display_cell()` 与第 7 章编写的同名 DBI 函数相类似。PHP 版本如下：

```
function display_cell ($tag, $value, $encode)
{
    if ($encode)
        $value = htmlspecialchars ($value);
    if ($value == "")
        $value = "&nbsp;";
    print ("<$tag>$value</$tag>\n");
}
```

如果在 `display_events()` 给出的表中选择了“New Event”连接，则脚本通过操作 `SOLICIT_EVENT` 进行再次调用。它引发了对 `solicit_event_info()` 的调用，这个函数显示了允许输入新事件信息的表格：

```
function solicit_event_info ()
{
    global $PHP_SELF;

    printf ("<FORM METHOD=\"post\" ACTION=\"%s?action=%d\">\n",
            $PHP_SELF, ADD_EVENT);
    print ("Enter information for new grade event:<BR><BR>\n");
    print ("Date: <INPUT TYPE=text NAME=\"date\">");
    print (" VALUE=\"\" SIZE=10> ");
    print ("Type: ");
    print ("<INPUT TYPE=\"radio\" NAME=\"type\" VALUE=\"T\" CHECKED=Test\n");
    print ("<INPUT TYPE=\"radio\" NAME=\"type\" VALUE=\"Q\">Quiz\n");
    print ("<BR><BR>\n");
    print ("<INPUT TYPE=\"submit\" NAME=\"button\" VALUE=\"Submit\">\n");
    print ("</FORM>\n");
}
```

由 `solicit_event_info()` 生成的表格包括输入数据的编辑域、指定新事件是测试还是测验的两个单选按钮、Submit 按钮。当递交表格时，`ADD_EVENT` 操作将调用 `score_entry.php`。调用 `add_new_event()` 函数在 `event` 表中输入一个新的行：

```
function add_new_event ()
{
    global $date, $type;

    if (empty ($date)) # make sure a date was entered
        die ("No date specified");
    $query = sprintf ("INSERT INTO event (date,type) VALUES(\"%s\", \"%s\")",
        addslashes ($date), addslashes ($type));
    if (!mysql_query ($query))
        die ("Could not add event");
}
```

在 `add_new_event()` 中，我们使用 `global` 访问在新事件项表格中使用的域值（`date` 和 `type`，用变量 `$date` 和 `$type` 访问）。做出最低限度的安全检查，确定数据为非空白之后，在 `event` 表中输入一个新记录。输入这个事件记录之后，主程序将再次显示事件列表，这样就可以选择新事件并开始输入分数了。

函数 `display_scores()` 为给定的事件查找已存在的分数，并列出显示他们的表格，包括学生姓名：

```
function display_scores ()
{
    global $PHP_SELF, $event_id;

    # select scores for the given event
    $query = "
        SELECT
            student.student_id, student.name, event.date,
            score.score AS score, event.type
        FROM student, event
        LEFT JOIN score ON student.student_id = score.student_id
        AND event.event_id = score.event_id
        WHERE event.event_id = $event_id
        ORDER BY student.name
    ";
    $result = mysql_query ($query)
        or die ("Cannot execute query");

    printf ("<FORM METHOD=\"post\" ACTION=\"%s?action=%d&event_id=%d\">\n",
        $PHP_SELF, ENTER_SCORES, $event_id);

    # print scores in a table, and print the event date and type
    # preceding the table. (however, we cannot print the date and
    # type until we've fetched the first row of the result set)

    $needheading = 1;
    while ($row = mysql_fetch_array ($result))
    {
        if ($needheading)
        {
            printf ("Event ID: %s, Event date: %s, Event type: %s\n",
                $event_id, $row["date"], $row["type"]);
            print ("<BR><BR>\n");
            print ("<TABLE BORDER>\n");
            print ("<TR>\n");
            display_cell ("TH", "Name", 1);
            display_cell ("TH", "Score", 1);
            print "</TR>\n";
            $needheading = 0;
        }
        print ("<TR>\n");
        display_cell ("TD", $row["name"], 1);
        $col_val =
            sprintf ("<INPUT TYPE=text NAME=\"score[%s]\"",
                $row["student_id"]);
        $col_val .=
            sprintf (" VALUE=\"%s\" SIZE=5><BR>\n",
                $row["score"]);
        display_cell ("TD", $col_val, 0);
        print ("</TR>\n");
    }

    print ("</TABLE>\n");
    print ("<BR>\n");
    print ("<INPUT TYPE=\"submit\" NAME=\"button\" VALUE=\"Submit\">\n");
    print "</FORM>\n";
}
```

display_scores() 用于检索所选事件的分数信息的查询并不是表之间的简单连接，因为它不会为事件中没有分数的学生选择行。特别是，对于新的事件，连接会选择无记录，这就有了一个空项表格！我们使用 LEFT JOIN 强迫为每个学生检索行，无论学生是否在 score 表中已经有了分数。与 display_scores() 用来检索来自于 MySQL 的分数记录相类似的查询背景，已在 3.8.2 节“检查表中未给出的值”中给出了介绍。那里的查询只选择缺失分数，这里的查询只选择特殊事件的分数。

分数在表格中使用了有名称的域进行编码，如 score[n]，这里的 n 是 student_id 的值。当表格送回 Web 服务器时，PHP 将这些域转换为 \$score 数组的元素，我们可以访问数组元素以恢复表格的内容。

当完成输入或者编辑分数，并提交给表格后，ENTER_SCORES 操作调用 score_entry.php，并且调用函数 enter_scores() 处理表格信息：

```
function enter_scores ()
{
    global $score, $event_id;

    $invalid = 0;
    $blank = 0;
    $nonblank = 0;
    while (list ($student_id, $newscore) = each ($score))
    {
        $newscore = trim ($newscore);
        if (empty ($newscore)) # no score, delete if present in table
        {
            ++$blank;
            $query = "DELETE FROM score"
                . " WHERE event_id = $event_id"
                . " AND student_id = $student_id";
        }
        else if (ereg ("^[0-9]+$", $newscore)) # must be integer
        {
            ++$nonblank;
            $query = "REPLACE INTO score (event_id,student_id,score)"
                . " VALUES($event_id,$student_id,$newscore)";
        }
        else
        {
            ++$invalid;
            continue;
        }
        if (!mysql_query ($query))
            die ("score entry failed, event_id $event_id, student_id $student_id");
    }
    printf ("Scores entered: %d<BR>\n", $nonblank);
    printf ("Scores missing: %d<BR>\n", $blank);
    printf ("Invalid scores: %d<BR>\n", $invalid);
    print ("<BR>\n");
}
```

学生 ID 的值和相关的分数通过迭代 PHP 的 each 函数的 \$score 数组来获得。每个分数处理如下：

- 如果分数是空白的，则表明什么也没有输入，但是我们还要试图删除这个分数，以免它以前曾经存在（也许以前我们为缺席的学生错误地输入了分数）。

■ 如果分数不是空白的，就对值进行一些根本的确认。用函数 `trim()` 去掉前后的空格之后，如果剩余部分是空白或者整数，就接受这个结果。然而，表格值通常作为字符串来编码，因此不能用 `is_long()` 或者 `is_int()` 检查值是否为整数。即使值只包括数字，这些函数也会返回 `FALSE`。既然如此，最好用模型匹配操作。如果字符串从开始到结束每个字符都是数字，则下面的测试为 `TRUE`：

```
ereg ("^[0-9]+$", $str)
```

如果分数检查完毕，我们就将它加到 `score` 表中。查询使用 `REPLACE` 而不用 `INSERT`，因为我们可能替换了已存在的分数而不是输入一个新的分数（`REPLACE` 在两种情况下都适用）。

注意 `score_entry.php` 脚本。现在所有的分数项和编辑项都能从 Web 浏览器执行。一个明显的缺点是：脚本没有提供安全措施，连接到 Web 服务器的任何人都可以对分数进行编辑。以后，我们用编辑历史同盟成员项编写的脚本来说明这个脚本所采取的简单确认方案。也可以使用 `PHPLIB` 程序包来提供更完善的确认。

8.3.2 美国总统测验

历史同盟 Web 站点的目标之一就是用它给出测验的在线版本，这类似于同盟在时事通信“美国编年史”的儿童部分发表的一些测验。实际上我们创建了 `president` 表，因此对基于历史的测验可以用它作为问题的来源。为了给出这个测验，我们将编写称为 `pres_quiz.php` 的脚本。

基本的想法是随机挑选一个总统，问一个关于他的问题，然后请求用户回答并且察看答案是否正确。为了简单一点，可以把主题限制为询问总统出生在哪里。另外一种简单的衡量就是以多个选择的格式给出这个问题。这对用户来讲很容易，他只需从一组选择中挑选一个，而不用将之键入等待回应。这对我们来讲也是容易的，因为我们不需做任何棘手的匹配字符串来检查用户可能键入的内容，而只需对用户的选择和我们寻找的值做一个简单的比较。

显示这个测验的脚本必须执行两个函数。第一个，对于它最初的调用，将从 `president` 表中查阅信息来生成并显示一个新的问题。第二个，如果脚本已经被调用是因为用户正提交一个回答，那么就需要检查这些答案并给出一些反馈信息来指出它是否正确。如果正确，脚本会生成并显示一个新的问题。如果回答不正确，将再次显示同一问题。

为了生成这些问题，我们将使用 `MySQL 3.23` 中出现的一个 `ORDER BY RAND()` 特性。使用这个函数就能从 `president` 表中随机地进行行选择。例如，为了随机地挑选总统的姓名和出生地，查询将执行这样的操作：

```
SELECT CONCAT(first_name, ' ', last_name) AS name,  
       CONCAT(city, ', ', state) AS place  
FROM president ORDER BY RAND() LIMIT 1
```

`name` 是选择的总统的名字，出生地是问题“总统出生在哪里？”的正确答案，我们还需要给出一些错误的选择，可用类似的查询：

```
SELECT DISTINCT CONCAT(city, ', ', state) AS place  
FROM president ORDER BY RAND()
```

从这个查询的结果中，我们选择了与正确答案不同的最前面的四个值。发布这个查询并检索结果的函数如下：

```

function setup_quiz ()
{
    # issue query to pick a president and get birthplace
    $query = "SELECT CONCAT(first_name, ' ', last_name) AS name,"
        . " CONCAT(city, ' ', state) AS place"
        . " FROM president ORDER BY RAND() LIMIT 1";
    $result = mysql_query ($query)
        or die ("Cannot execute query");
    $row = mysql_fetch_array ($result)
        or die ("Cannot fetch result");
    $name = $row["name"];
    $place = $row["place"];

    # Construct the set of birthplace choices to present.
    # Set up the $choice array containing five birthplaces, one
    # of which is the correct response.
    $query = "SELECT DISTINCT CONCAT(city, ' ', state) AS place"
        . " FROM president ORDER BY RAND()";
    $result = mysql_query ($query)
        or die ("Cannot execute query");
    $choice[] = $place; # initialize array with correct choice
    while (count ($choice) < 5 && $row = mysql_fetch_array ($result))
    {
        if ($row["place"] == $place)
            continue;
        $choice[] = $row["place"]; # add another choice
    }
    # randomize choices, then display form
    shuffle ($choice);
    display_form ($name, $place, $choice);
}

```

为了给出测验问题的信息，我们使用了显示总统姓名、一组列出可能选择的单选按钮和一个Submit按钮的表格。这个表格需要做两件事情：必须对客户机给出测验信息；当用户提交回答时必须将信息传送回 Web 服务器，以便检查回答是否正确。

为了安排表格执行这些操作，我们使用了隐藏域把测验信息包括在表格中。把域称为 name、place 和 choice，它们代表总统的姓名、出生地和一组可能的选择。使用 implode() 连接值和特殊字符，这样，这些选择可以很容易地作为单个字符串来编码（我们需要特殊字符，以便如果需要重新显示问题时可以用 explode() 分离字符串）。显示表格的函数如下：

```

function display_form ($name, $place, $choice)
{
    global $PHP_SELF;

    printf ("<FORM METHOD=\\"post\\" ACTION=\\"%s\\">\n", $PHP_SELF);
    hidden_field ("name", $name);
    hidden_field ("place", $place);
    hidden_field ("choice", implode ("#", $choice));
    printf ("Where was %s born?<BR><BR>\n", $name);
    for ($i = 0; $i < 5; $i++)
    {
        print ("<INPUT TYPE=\\"radio\\" NAME=\\"response\\">\n");
        printf ("VALUE=\\"%s\\">%s<BR>\n", $choice[$i], $choice [$i]);
    }
    print ("<BR><INPUT TYPE=\\"Submit\\" VALUE=\\"Submit\\">\n");
    print ("</FORM>\n");
}

```

函数hidden_field()为表格中的隐藏域编写了HTML:

```
function hidden_field ($name, $value)
{
    printf (<input type="hidden" name="%s" value="%s">\n",
           $name, $value);
}
```

当用户做出选择并提交表格时, 答案作为 response 域值在发送回 Web 服务器的表格内容中编码。我们可以通过检查变量 \$name、\$place 和 \$choice 发现 name、place 和 choice 域的值。这也给了我们一个方法, 指出是否是第一次调用脚本, 或者用户是否给以前显示过的表格提交了回答, 如果是第一次调用则不会设置那些变量。这样, 通过检查其中一个变量, 脚本的主体就决定了应该做的事情:

```
if (!$name)           # called for first time
    setup_quiz ();
else                  # user submitted response to form
    check_response ();
```

我们仍然需要编写 check_response() 函数来将用户的回答与正确答案做比较。我们将正确答案在表格的 place 域进行编码, 用户的回答则在表格的 response 域进行编码, 因此我们所要做的就是比较 \$place 和 \$response。在比较结果的基础上, 我们提供了一些反馈信息, 之后每次都生成显示一个新的问题, 或者再次显示相同的问题:

```
function check_response ()
{
    global $name, $place, $choice;
    global $response;

    if ($place == $response) # correct response; generate new question
    {
        print ("That is Correct!<br>\n");
        printf ("%s was born in %s.<br>\n", $name, $place);
        print ("Try the next question:<br><br>\n");
        setup_quiz();
    }
    else                      # incorrect response; redisplay question
    {
        print ("That is not correct. Please try again.<br><br>\n");
        $choice = explode ("#", $choice);
        display_form ($name, $place, $choice);
    }
}
```

这样, 我们就做完了。将 pres_quiz.php 的链接加到历史同盟主页上, 参观者可以做一下这个测验来测试他们的知识。

8.3.3 历史同盟联机成员项的编辑

最终的脚本 edit_member.php 允许历史同盟成员编辑他们自己的联机项。无论何时, 成员都可以校正或者更新他们的成员信息, 而不必向同盟部提交这些更改。这个性能使成员目录总是保持最新的, 而且减少了秘书的工作量。

我们需要采取的一个防范措施就是: 除了该项目的成员之外, 防止任何其他人修改项目。这意味着我们需要一些安全性的表单。作为一个简单的身份确认表单的示范, 我们将使用 MySQL 存放每个成员的口令, 并要求成员提供正确的口令以访问脚本给出的编辑表单。该脚

本操作如下：

- 当初次调用时，edit_script.php 给出包括成员 ID 和口令域的表单。
- 当提交初始表单时，脚本用成员 ID 作为关键字寻找相关的口令来搜索口令表。如果口令相符，脚本将从 member 表中查找成员项，并显示要编辑的内容。
- 当提交编辑过的表单后，我们就用表单的内容更新项。

edit_member.php 的框架如下所示：

```
<?php
# edit_member.php - Edit Historical League member entries via the Web

include ("samp_db.inc");

# define action constants
define (INITIAL_PAGE, 0);
define (DISPLAY_ENTRY, 1);
define (UPDATE_ENTRY, 2);

/* ... put functions here ... */

if (empty ($action))
    $action = INITIAL_PAGE;

$title = "US Historical League — Member Editing Form";
html_begin ($title, $title);

samp_db_connect ()
    or die ("Cannot connect to server");

switch ($action)
{
case INITIAL_PAGE:
    solicit_member_id ();
    break;
case DISPLAY_ENTRY:
    display_entry ();
    break;
case UPDATE_ENTRY:
    update_entry ();
    break;
default:
    die ("Unknown action code ($action)");
}

html_end ();
?>
```

初始页面显示了请求成员 ID 和口令的表单：

```
function solicit_member_id ()
{
global $PHP_SELF;

printf ("<FORM METHOD=\"post\" ACTION=\"%s?action=%d\">\n",
        $PHP_SELF, DISPLAY_ENTRY);
print ("Enter your membership ID number and password,\n");
print ("then select Submit.\n<BR><BR>\n");
print ("<TABLE>\n");
print ("<TR>");
print ("<TD>Member ID</TD><TD>");
print ("<INPUT TYPE=text NAME=\"member_id\" SIZE=10><BR>\n");
```



```

print ("</TD></TR>");
print ("<TR>");
print ("<TD>Password</TD><TD>");
print ("<INPUT TYPE=password NAME=\"password\" SIZE=10><BR>\n");
print ("</TD></TR>");
print ("</TABLE>\n");
print ("<INPUT TYPE=\"submit\" NAME=\"button\" VALUE=\"Submit\">\n");
print "</FORM>\n";
}

```

当然，我们还需要一些口令，一个简单的方法就是随机地生成它们。下面的语句建立 member_pass 表，然后，通过从随机数中生成 MD5 校验来为每个成员创建口令。您可以让成员们来选择他们自己的口令，也可以调用 mysql 并发布这些语句作为一种既快速又容易的方法：

```

CREATE TABLE member_pass
(
    member_id INT UNSIGNED NOT NULL PRIMARY KEY,
    password CHAR(8)
)
INSERT INTO member_pass (member_id, password)
SELECT
    member_id,
    LEFT(MD5(RAND()), 8) AS password
FROM member

```

我们将一个特殊项加到这个表中作为编号 0，它有一个用于管理的（超级用户）口令。可以使用这个口令访问所有想要访问的项：

```
INSERT INTO member_pass (member_id, password) VALUES(0, "secret");
```

在创建口令表之后，您可以停止使用第 7 章中编写的 samp_browse 脚本，该脚本允许任何人在 samp_db 数据库中浏览任何表的内容，其中包括 member_pass 表。

当成员输入 ID 和口令并提交该表单时，edit_member.php 显示该编辑的项：

```

function display_entry ()
{
    global $PHP_SELF;
    global $member_id, $password;

    $member_id = trim ($member_id);
    if (empty ($member_id))
        die ("No member ID specified");
    if (!ereg ("^[0-9]+$", $member_id)) # must be integer
        die ("Invalid member ID specified (must be number)");
    if (empty ($password))
        die ("No password specified");
    if (check_pass ($member_id, $password)) # regular member
        $admin = 0;
    else if (check_pass (0, $password)) # administrator
        $admin = 1;
    else
        die ("Invalid password");

    $query = "SELECT last_name, first_name, suffix, email,"
        . "street, city, state, zip, phone, interests,"
        . "member_id, expiration"
        . " FROM member"
        . " WHERE member_id = $member_id"
        . " ORDER BY last_name";
    $result = mysql_query ($query)
        or die ("Cannot execute query");
}

```

```

if (mysql_num_rows ($result) == 0)
    die ("No user with member_id = $member_id found");
if (mysql_num_rows ($result) > 1)
    die ("More than one user with member_id = $member_id found");

printf ("<FORM METHOD=\"post\" ACTION=\"%s?action=%d\">\n",
        $PHP_SELF, UPDATE_ENTRY);

hidden_field ("member_id", $member_id);
hidden_field ("password", $password);
print ("<TABLE>\n");
# read results of query and format for editing
$row = mysql_fetch_array ($result);
display_column ("Member ID", $row, "member_id", 0);
# allow user with admin password to edit expiration
display_column ("Expiration", $row, "expiration", $admin);
display_column ("Last name", $row, "last_name", 1);
display_column ("First name", $row, "first_name", 1);
display_column ("Suffix", $row, "suffix", 1);
display_column ("Email", $row, "email", 1);
display_column ("Street", $row, "street", 1);
display_column ("City", $row, "city", 1);
display_column ("State", $row, "state", 1);
display_column ("Zip", $row, "zip", 1);
display_column ("Phone", $row, "phone", 1);
display_column ("Interests", $row, "interests", 1);
print ("</TABLE>\n");
print ("<INPUT TYPE=\"submit\" NAME=\"button\" VALUE=\"Submit\">\n");
print "</FORM>\n";

```

display_entry() 需要做的第一件事就是校验口令。对于给定的成员 ID，如果表单中输入的口令与 member_pass 表中存放的口令相符，或者如果它与管理口令相符（即成员 0 的口令），edit_member.php 就显示编辑的项。口令检查函数 check_pass() 将执行一个简单的查询从 member_pass 表中移出一条记录：

```

function check_pass ($id, $pass)
{
    $query = "SELECT password FROM member_pass WHERE member_id = $id";
    if (!$result = mysql_query ($query))
        die ("Error reading password table");
    if (!$row = mysql_fetch_array ($result))
        return (FALSE);
    return ($row["password"] == $pass); # TRUE if password matches
}

```

因为不能修改它，所以编辑表单作为只读文本显示成员 ID 的值。对于正常的成员，截止日期也作为只读文本显示，因为不能让成员改动它。然而，如果给出管理口令，则截止日期就成为可编辑的，允许同盟秘书为成员更新日期来重新更新他们的会员资格。

member 表项的列由 display_column() 函数显示。它按照第三个参数值把列作为可编辑的文本或作为只读文本加到编辑表单中：

```

# Display a column of a member entry. $label is the visible label
# displayed to the user. $row is the array containing the entry.
# $col_name is the name of a column in the entry. $value is the
# column value. $editable is non-zero if the user is allowed to
# change the value. The value is displayed as non-editable text
# otherwise. Field names are constructed as row[col_name] so that
# when the form is submitted, values can be accessed using an array
# rather than a bunch of individual variables.

```

```
function display_column ($label, $row, $col_name, $editable)
{
    print("<TR>\n");
    printf("<TD>%s</TD>\n", htmlspecialchars ($label));
    $value = htmlspecialchars ($row[$col_name]);
    if ($editable) # display as edit field
    {
        $str = sprintf ("<INPUT TYPE=text NAME=\"row[%s]\" ", $col_name);
        $str .= sprintf (" VALUE=\"%s\" SIZE=\"%0\">\n", $value);
    }
    else # display as read-only text
        $str = $value;
    printf("<TD>%s</TD>\n", $str);
    print("</TR>\n");
}
```

display_entry() 函数在格式中作为隐藏字段嵌入了 member_id 和 password, 因此当成员提交编辑的项时将继续 edit_script.php 的下一个调用。这允许自动校验 ID 的口令, 而不用请求成员再次输入 (请注意, 我们的简单的确认身份的方法是以文本形式来回传递口令。通常这不是个好主意, 但是历史同盟不是对安全性要求很高的运作机构, 因此这种方法足够满足要求。如果在运行金融业务, 可能需要更强的安全性操作)。

更新项的函数如下:

```
function update_entry ()
{
    global $row, $member_id, $password;

    $member_id = trim ($member_id);
    if (empty ($member_id))
        die ("No member ID specified");
    if (!ereg ("^[0-9]+$", $member_id)) # must be integer
        die ("Invalid member ID specified (must be number)");
    if (!check_pass ($member_id, $password) && !check_pass (0, $password))
        die ("Invalid password");

    # We'll need a result set to use for assessing nullability of
    # member table columns. This query gives us one without
    # selecting any rows.
    $result = mysql_query ("SELECT * FROM member WHERE 1 = 0");
    if (!$result)
        die ("Cannot query member table");

    # iterate through each field in the form, using the values to
    # construct the UPDATE statement.

    $query = "UPDATE member ";
    $delim = "SET "; # put "SET" before first column, "," before others
    while (list ($col_name, $val) = each ($row))
    {
        $query .= "$delim $col_name =";
        $delim = ",";

        # if a value is empty, update the value with NULL if the
        # column is nullable. This prevents trying to put an
        # empty string in the expiration when it should be NULL,
        # for example.
        $val = trim ($val);
        if (empty ($val))
        {
```

```

        if (nullable ($result, $col_name))
            $query .= "NULL"; # enter NULL
        else
            $query .= "\""; # enter empty string
    }
    else
        $query .= "\"" . addslashes ($val) . "\"";
}
$query .= " WHERE member_id = $member_id";
if (mysql_query ($query) && mysql_affected_rows () > 0)
    print ("Entry updated successfully.\n");
else
    print ("Entry not updated.\n");
}

```

首先，重新校验口令，确定没人发送假表单来愚弄我们，然后更新项。更新时需要注意，因为如果表单中的字段是空白的，则可能需要作为 NULL 而不是作为空字符串输入。expiration 列就是这样的例子。NULL 的成员截止日期具有特殊的含义，即“终生会员”。如果将一个空字符串插入到此列中，值转换成“0000-00-00”，则成员不再具有终生会员资格。

为了处理这个问题，我们查找该列的元数据并检查它是作为 NULL 还是作为 NOT NULL 进行声明的。该信息由函数 mysql_fetch_field() 返回。不幸地是，此函数通过数值的索引查找列。在 member 表中按名称访问列会更方便，因此我们编写一个小函数 nullable()，它获取一个列名并查找相应的元数据对象：

```

# Determine whether or not the column with the given name in
# the result set is nullable.

function nullable ($result, $col_name)
{
    for ($i = 0; $i < mysql_num_fields ($result); $i++)
    {
        if (!($fld = mysql_fetch_field ($result, $i)))
            continue;
        if ($fld->name == $col_name)
            return (! $fld->not_null);
    }
    return (0);
}

```

mysql_fetch_field() 函数需要包含检查列所在表的结果集标识符。这可通过执行简单的不返回行的 SELECT 查询来获得。虽然该查询返回一个空结果集，但是，对于检索要评估 member 表中列的空性能 (nullability) 的元数据来说，这种方法足够了：

```
SELECT * FROM member WHERE 1 = 0
```

安装脚本，让成员们知道他们的口令，这样他们就能更新自己的成员信息了。

第三部分 MySQL 管理

第9章 MySQL 管理介绍

在运行数据库系统时，MySQL 的使用相当简单，且进行 MySQL 安装和使用所需的工作也很少。MySQL 的简单性可能就是它极为普及的原因，尤其是在非程序员人群中的普及。当然，它对于训练有素的计算机专业人员也是有帮助的，但肯定不是对运行一个成功的 MySQL 安装程序的需求。

然而，不论您是什么级别的专家，MySQL 的安装程序都不能自动运行。必须有人来监视它以确保它能顺利和有效地运行，有时还必须知道当问题出现时应该做什么。如果问题偶然地落到了您的头上，要想确保 MySQL 的正常，应继续阅读本书。

在本书的第三部分中，我们将讨论关于 MySQL 管理的各个方面。本章给出了您应当了解的，并包含在管理 MySQL 安装程序中的有关职责内容的概述，还提供了对这些职责的简单描述，并在后面的几章中给出执行它们的指导。

如果您是一位新手或毫无经验的 MySQL 管理员，可千万别让本章出现的冗长的职责清单吓着。下面小节中所列出的每个任务都是重要的，但是，您不需要马上学习它们。如果您希望这样做的话，可将本章节的内容作为参考来使用，当感觉需要知道其内容时可查找这些主题。

如果您有管理其他数据库系统的经验，将会发现：运行 MySQL 的安装程序在某些方面是类似的，您的经验也是用得着的。但是 MySQL 的管理有自己独特的需求，本书的这个部分将帮助您熟悉这些内容。

9.1 管理职责概述

MySQL 数据库系统由几部分组成。您应该熟悉这些组成部分的内容和每个部分的目的。这需要您了解所管理系统的特征以及帮助您进行管理的可用工具。如果您花时间去了解了要监督的内容，工作将会变得非常容易。为此，您应当使自己熟悉 MySQL 的以下几个方面：

- **MySQL 服务器。**服务器 mysql 执行数据库和表的所有操作。safe_mysqld 是一个相关的程序，它用于启动服务器、监控服务器和重新启动服务器。
- **MySQL 客户机和实用程序。**有几个 MySQL 程序，可用来帮助您与服务器进行通信和执行管理的任务。其中最重要的几个是：
 - mysql，一个交互式程序，允许将 SQL 语句发布到服务器上并浏览其结果。
 - mysqladmin，一个管理程序，允许执行诸如关闭服务器以及创建或删除数据库的工作。如果服务器运行不正常，还可以用 mysqladmin 来检查服务器的状态。
 - isamchk 和 myisamchk，这些实用程序帮助您完成表的分析和优化，以及在表损坏时进行崩溃恢复。

■ **mysqldump**，一个工具，用于备份数据库或将数据库拷贝到另一个服务器中。

- **服务器的语言，SQL**。有些管理职责只能用 **mysqladmin** 的命令行实用程序来完成，但是，如果您还能用服务器自己的语言来同服务器进行对话，那就更好了。作为简单的例子，您可能需要查找用户特权不按您所希望的方式进行工作的原因。没有任何替代品能够参与并与服务器直接通信。可通过使用 **mysql** 客户机程序发布能够检验授权表的 SQL 查询来做到这一点。如果您的 MySQL 版本还未引入 **GRANT** 语句，则需要使用 **mysql** 首先设置每个用户的权限。

如果您不知道 SQL 的任何内容，至少必须对 SQL 要有基本的了解。缺乏对 SQL 的熟悉只会给您带来困惑，而在学习 SQL 上所花费的时间将会得到成倍的回报。真正掌握 SQL 要花费一些时间，但掌握基本技能则很快。如果您需要了解对 SQL 和 **mysql** 命令行客户机的介绍内容，请参阅第 1 章的“MySQL 和 SQL 介绍”。

- **MySQL 数据目录**。数据目录是服务器存储其数据库和状态文件的所在。了解数据目录的结构及内容是很重要的，您可以知道服务器是怎样使用文件系统来表现数据库和表的，以及像日志这样的文件的存放位置和其内容。还应该了解在文件系统中管理磁盘空间分配的选项，当发现放置数据目录的文件系统过满时可以进行调整。

9.2 常规管理

常规管理主要指处理 **mysqld**、MySQL 服务器和提供给用户的访问服务器的操作。在履行该职责时，下列的任务是最重要的：

- **服务器的启动和关闭**。您应该能够从命令行中手工启动和终止服务器，并且在系统启动和关闭时知道怎样进行自动启动和关闭。如果服务器崩溃了或启动不正常的话，了解怎样使服务器再次运行也是重要的。
 - **用户账号维护**。应该了解 MySQL 用户和 UNIX 或 Windows 用户之间的区别。应该知道怎样通过指定哪些用户可以连接到服务器和从哪里进行连接来建立 MySQL 用户账号。还应该给新的用户建议合适的连接参数，以使他们成功地连接到服务器。弄清应怎样建立账号不是用户们的工作。
 - **日志文件维护**。应该了解可以维护的日志文件的类型，以及在何时和怎样完成日志文件的维护。日志的循环和终止对于防止日志填满文件系统是必要的。
 - **数据库备份和拷贝**。数据库备份对服务器系统的崩溃是至关重要的。应该能够将数据库恢复到崩溃时的状态，以便尽可能地减少数据的丢失。请注意，数据库备份与常规的系统备份不同，例如，可通过使用 UNIX 的 **dump** 程序来进行。与数据库表相对应的文件在系统备份发生时随服务器活动而变化，因此恢复那些文件将使您的表内部不一致。**mysqldump** 程序将产生对恢复数据库更有用的备份文件，并允许在不关闭服务器的状态下创建备份。
- 如果决定在更快的主机上运行数据库，或者想复制数据库，则需要拷贝其内容到另一台机器上。如果需要的话，应该了解进行这项操作的过程。数据库文件是依赖于系统的，因此您不能只拷贝这些文件。
- **服务器优化**。用户想要服务器以最佳状态运行。提高服务器运行性能的最简单方法是购买更多的内存或使磁盘速度更快。但是，这种直截了当的技术并不能代替对服务器

工作的了解。应该了解优化服务器操作所用的参数以及如何将这些参数应用在您的环境中。在某些站点中，大多数查询都是检索。而在另一些站点，插入和更新操作占据着优势。选择对哪些参数进行修改将受到站点查询的影响。

- **多服务器**。在某些环境中运行多服务器是有用的。如果保留当前的成品安装程序在适当的位置，或者为不同的用户组提供较好的保密性（后者与 ISP 尤其相关），则可以测试新的 MySQL 版本。对于这些情形，您应该了解怎样建立多个同时发生的安装。
- **MySQL 更新**。由于新的 MySQL 版本频繁出现，应该知道怎样始终跟上这些版本以便利用故障修复和新的特性。需要了解不进行版本升级的理由，并且掌握怎样在稳定版本和开发者版本之间进行选择。

9.3 安全性

当运行 MySQL 安装程序时，确保用户所存储的数据的安全性是很重要的。MySQL 管理员有责任控制对数据目录和服务器的访问，并应了解以下的问题：

- **文件系统的安全性**。UNIX 机器可能会使几个用户账号成为宿主账号，而这些账号都没有与 MySQL 相关的管理职责。确保这些账号没有对数据目录的访问是重要的。因为这样可以防止它们通过拷贝数据库表或移动数据库表，或者通过能够读取包含敏感信息的日志文件来损坏文件系统级的数据。您应该知道如何建立 MySQL 服务器的 UNIX 用户账号，如何建立该用户所拥有的数据目录，以及如何启动服务器以便利用该用户的权限运行。
- **服务器的安全性**。必须了解 MySQL 的安全系统是怎样进行工作的，以便在建立用户账号时授予适当的权限。通过网络连接到服务器的用户只允许做他们应该做的事情。您不要由于对安全系统的错误理解，将超级用户的访问权授予匿名用户。

9.4 数据库修复和维护

所有的 MySQL 管理员都希望避免处理破坏的或毁坏的数据表。但是愿望不能代替现实。以下几个步骤可以使您在问题发生时减少风险并学会怎样处理问题：

- **崩溃恢复**。如果尽管您做了最好的努力但灾难还是降临了，则应该知道如何修复或恢复表。很少会用到崩溃恢复，但当使用它时，它是一个令人讨厌的、高强度的工作（尤其是当您正在忙乱地修正某些内容时，电话铃响了或有人敲门）。然而，您必须知道怎样处理它，否则用户将会很不高兴。要熟悉 isamchk 和 myisamchk 的表的检查以及修复能力；要知道如何尽可能地从备份文件中恢复，并且知道怎样使用该更新日志来恢复备份之后所产生的修改。
- **预防性维护**。预防性维护的常规程序应适当地进行安置，以使数据库破坏和毁坏的可能性最小化。当然，您还要进行备份，但是，预防性维护将减少使用这些备份的机会。

以上综合地概括了作为 MySQL 管理员应承担的职责。第 10 章将详细讨论这些职责并提供操作过程，以便有效地履行这些职责。我们将首先讨论 MySQL 数据目录，这是您正在管理的资源，因此应该了解其布局和内容。然后再讨论常规的管理职责、MySQL 安全系统，以及维护和故障排除。

第10章 MySQL 数据目录

从概念上讲，大多数关系数据库系统都是类似的：它们都由一组数据库组成，且每个数据库都包含一组表。但是，所有的系统都有自己的管理数据的方法，MySQL 也不例外。

在缺省设置时，由 MySQL 服务器 `mysql` 管理的所有数据都存储在 MySQL 数据目录中。所有的数据库和提供有关服务器操作信息的状态文件也存储在那里。如果要执行 MySQL 安装的管理职责，应该熟悉数据目录的布局和使用。

本章包括以下内容：

- **怎样确定数据目录的位置。**需要了解这一点，以便有效地管理其内容。
- **服务器怎样组织和提供对它所管理的数据库和表的访问。**
- **在哪里查找由服务器产生的状态文件且文件中包含什么内容。**它们的内容提供了关于服务器运行的信息，如果您遇到问题，这些信息将是有用的。
- **如何改变数据目录或单个数据库的缺省布局或组织。**这对于管理系统的磁盘资源分配是重要的——例如，通过在驱动器中平衡磁盘的活动，或通过利用更多的可用空间将数据重定位到文件系统中。还可以使用这个知识来计划新数据库的布局。

即使不执行任何 MySQL 管理职责，也能通过阅读本章的内容而有所收获，它决不会影响您对如何运行服务器有更好的想法。

10.1 数据目录的位置

缺省数据库的位置编译在服务器中。如果您是在一个源程序分发包中安装 MySQL，典型的缺省位置可能是 `/usr/local/var`；如果在二进制分发包中安装 MySQL，则为 `/usr/local/mysql/data`；在 RPM 文件中安装，为 `/var/lib/mysql`。

数据目录的位置可以在启动服务器时通过 `--datadir = / path / to / dir` 明确地指定。如果您想将数据目录放置在其他地方而非缺省的位置，则这个选项是有用的。

作为一名 MySQL 管理员，您应该知道数据目录在哪里。如果运行多个服务器，那么您应该掌握所有数据目录的位置。但是，如果不知道目录的位置（或许您正在代替前一位管理员，而他留下的记录很糟糕），有几种方法可以用来查找它：

- **可使用 `mysqladmin` 变量直接从服务器中得到数据目录路径名。**在 UNIX 中，输出结果类似于如下所示：

```
% mysqladmin variables
+-----+-----+
| Variable_name | Value |
+-----+-----+
| back_log      | 5     |
| connect_timeout | 5     |
| basedir       | /var/local/ |
| datadir       | /usr/local/var/ |
| ...           |       |
```

该输出结果指明了服务器主机中数据目录的位置 /usr/local/var。

在 Windows 中，输出结果类似于如下所示：

```
C:\> mysqladmin variables
```

Variable_name	Value
back_log	5
connect_timeout	5
basedir	c:\mysql\
datadir	c:\mysql\data\
...	

如果正在运行多个服务器，它们将监听不同的 TCP/IP 端口号和套接字。可以通过提供合适的 --port 或 --socket 选项连接到每个服务器监听的端口和套接字上：

```
% mysqladmin --port=port_num variables
```

```
% mysqladmin --socket=/path/to/socket variables
```

mysqladmin 命令可在您连接服务器的任何一台主机上运行。如果需要连接到远程主机上的服务器，则使用 --host = host_name 选项：

```
% mysqladmin --host=host_name variables
```

在 Windows 中，您可以购买 Windows NT 服务器，它通过使用 --pipe 迫使一个指定的管道连接，并使用 --socket = pipe_name 指定该管道的名称，在该管道上进行监听。

```
C:\> mysqladmin --pipe --socket=pipe_name variables
```

■ 可使用 ps 来查看任何当前执行 mysql 进程的命令行。试一试下列的命令（根据您的系统所支持的 ps 版本）并查找显示在输出结果中的这些命令的 --datadir：

```
% ps axww | grep mysqld      BSD-style ps
```

```
% ps -ef | grep mysqld      System V-style ps
```

如果系统运行多个服务器（因为一次发现了多个数据目录位置），则 ps 命令将会特别有用。它的缺点是：ps 必须运行在服务器的主机上，并且除非 --datadir 选项在 mysqld 命令行中明确指定，否则将产生无用的信息。

■ 如果 MySQL 从源程序分发包中安装，可以检查其配置信息以确定数据目录的位置。

例如，在最高级的 Makefile 中该位置是可用的。但是，要小心：位置是 Makefile 中的变量 localstatedir 的值，而不是 datadir 的值。同样，如果分发包定位在 NFS 装配文件系统中，并且是用于为几个主机建立 MySQL 的，则配置信息反映最近建立分发包的主机。它可能不显示您感兴趣的主机的数据目录。

■ 如果前面的任何方法都不成功，可使用 find 搜索数据库文件。下列命令将搜索 .frm（描述）文件，它是 MySQL 安装程序的组成部分：

```
% find / -name "*.frm" -print
```

在本章的这些例子中，笔者将 MySQL 数据目录的位置表示为 DATADIR。您可以将其解释成为您自己的机器中的数据目录的位置。

10.2 数据目录的结构

MySQL 数据目录中包含由服务器管理的所有数据库和表。它们被组织成一个树状结构，该结构是通过 UNIX 或 Windows 文件系统的层次结构用简单的方式实现的：

■ 每个数据库对应该数据目录下的一个目录。

■ 数据库中的表对应数据库目录中的文件。

数据目录还包含几个由服务器生成的状态文件，如日志文件。这些文件提供了关于服务器运作的重要信息，对管理员是有用的，尤其是当问题出现且试图确定问题的原因时特别有用。例如，如果某个特定的查询毁坏了数据库，您可以通过检查日志文件来识别这个讨厌的查询。

10.2.1 MySQL 服务器怎样提供对数据的访问

数据目录中的一切都由一个单个的实体进行管理，即 MySQL 服务器的 mysqld。客户机程序不能直接操纵数据。而服务器提供了访问数据库的唯一的连结，它担当着客户机程序和所需数据之间的媒介（参见图 10-1）。

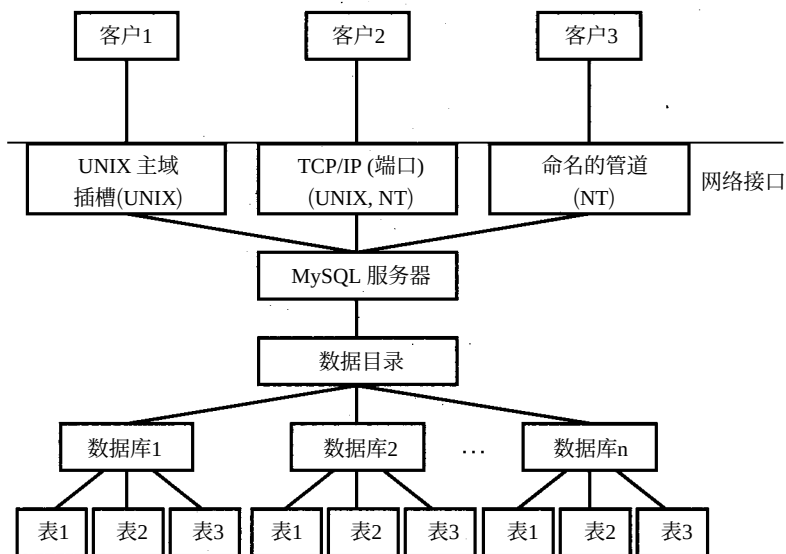


图10-1 MySQL 怎样控制对数据目录的访问

当启动服务器时，如果有任何请求，它都将打开日志文件，然后通过对网络连接的监听向数据目录展现网络接口。为了访问数据，客户机建立一个到服务器的连接，然后传达作为 SQL 查询的请求，以完成所期望的操作（例如，创建表、选择记录、更新记录）。服务器执行每个操作并将结果发送回客户机。服务器是多线程的，可以服务于多个并发的客户机的连接。但是，由于更新操作一次只能执行一个，因此实际上这些请求是顺序化的，两个客户机决不可能在同一时刻修改同一个记录。

在正常条件下，使服务器担当数据库访问的唯一仲裁者将提供对防止各种讹误的担保，这些讹误可导致多个进程同时访问数据库的表。然而，管理员应该知道：存在着服务器不具有对数据目录独占控制的时期：

■ **何时在单个数据目录中运行多个服务器。**通常情况下是运行一个单个的服务器来管理主机中的所有数据库，但是运行多个服务器也是可能的。如果这样做可以提供对多个独立的数据目录的访问，则不存在交互作用的问题。但是，有可能启动多个服务器并在相同的数据目录中指向它们。这是一个好主意，如果您想试试它，最好应确保系统提供了良好的文件锁定性能，否则服务器之间将不能协调地工作。如果将多个服务器

同时写入日志文件，则将会使日志文件成为混乱的来源（而不是有用信息的来源）。

- **何时运行 isamchk 和 myisamchk。** isamchk 和 myisamchk 实用程序用于表的维护、故障排除和修复。正如您所猜测的，由于这些实用程序能改变表的内容，所以如果在服务器运作的同时允许实用程序对表进行操作，将引起表的毁坏。了解怎样限制这种类型的交互作用以避免表毁坏是重要的。有关恰当使用这些程序的说明，请参阅第 13 章“数据库维护和修复”。

10.2.2 数据库的表示法

由 MySQL 管理的每个数据库都有自己的数据库目录，它们是数据目录的子目录，与所表示的数据库有相同的名称。例如，数据库 my_db 对应于数据库目录 DATADIR/my_db。

这个表示法使得几个数据库级的语句的实现几乎是微乎其微的。CREATE DATABASE db_name 使用只允许对 MySQL 服务器用户（服务器运行的 UNIX 用户）进行访问的所有权和方式，并在数据目录中创建一个空目录 db_name。这等价于以服务器主机中的服务器用户的身份通过执行下列命令手工创建数据库：

```
% mkdir DATADIR/db_name          创建数据库目录
% chmod 700 DATADIR/db_name       使它仅对 MySQL 服务器用户可访问
```

通过空目录表示新数据库的方法与其他数据库系统完全不同，那些数据库系统甚至要为“空”数据库创建许多控制文件或系统文件。

DROP DATABASE 语句也很容易实现。DROP DATABASE db_name 删除数据目录中的 db_name 目录以及其中的所有表文件。这个语句类似于下列命令：

```
%rm -rf DATADIR/db_name
```

其区别是，服务器只删除带有表的扩展名的文件。如果已经在该数据库目录中创建了其他的文件，服务器将使它们保持完整，并且不删除该目录本身。

SHOW DATABASE 只不过是对应于数据目录中的子目录名称的一个列表。有些数据库系统需要保留一个列出所有需要维护的数据库的主表，但是，在 MySQL 中没有这样的结构。由于数据目录结构的简单性，数据库的列表是隐含在该数据目录的内容中的，像主表这样的表可能会引起不必要的开销。

10.2.3 数据库表的表示法

数据库中的每个表在数据库目录中都作为三个文件存在：一个格式（描述）文件、一个数据文件和一个索引文件。每个文件的基名是该表名，扩展名指明该文件的类型。扩展名如表 10-1 所示。数据和索引文件的扩展名指明该表是否使用较老的 ISAM 索引或较新的 MyISAM 索引。

表10-1 MySQL 文件类型

文件类型	文件扩展名	文件内容
格式文件	.frm	描述表的结构（列、列类型、索引，等等）
数据文件	.ISD(ISAM) 或 .MYD(MyISAM)	包含表的数据——即它的行
索引文件	.ISD(ISAM) 或 .MYI(MyISAM)	包含数据文件中任何索引的索引树。无论该表有无索引，索引文件都存在

当发布定义一个表结构的 `CREATE TABLE tbl_name` 语句时，服务器创建 `tbl_name.frm` 文件，它包含该结构的内部编码。该语句还创建空的数据文件和索引文件，这些文件的初始信息表明没有记录和索引（如果 `CREATE TABLE` 语句包含索引说明，则该索引文件将反映这些索引）。描述表的文件的所有权和方式被设置为只允许对 MySQL 服务器用户的访问。

当发布 `ALTER TABLE` 语句时，服务器对 `tbl_name.frm` 重新编码并修改数据文件和索引文件的内容以反映由该语句表明结构变化。对于 `CREATE` 和 `DROP INDEX` 也是如此，因为服务器认为它们等价于 `ALTER TABLE` 语句。`DROP TABLE` 删除代表该表的三个文件。

尽管可以通过删除数据库目录中的对应某个表的三个文件来删除该表，但不能手工创建或更改表。例如，如果 `my_db` 是当前的数据库，`DROP TABLE my_tbl` 大致等价于下列命令：

```
% rm -f DATADIR/my_db/my_tbl.*
```

来自于 `SHOW TABLES my_db` 的输出结果正是 `my_db` 数据库目录中 `.frm` 文件基名的一个列表。某些数据库系统维护一个列出了数据库中的所有表的登记。但 MySQL 不这样做，因为没有必要，这个“登记”隐含在了数据目录的结构中。

10.2.4 数据库和表命名中的操作系统约束

MySQL 具有对数据库和表命名的一般规则：

- 名字可以由当前字符集中的字母数字字符以及下划线和美元符号（‘_’和‘\$’）组成。

- 名字最长可达 64 个字符。

但是，由于数据库和表的名字对应于目录和文件名，因此，对数据库运行的操作系统可以施加另外的约束。

首先，将数据库和表名限制为文件名中的合法字符。例如，按照 MySQL 的规则，名字中允许使用‘\$’，但是，如果操作系统不允许使用它，则不能在目录名或表名中使用。实际上，这与 UNIX 或 Windows 无关。可能会遇到的最大困难是在进行数据库管理时从外壳程序中直接命名。例如，如果给某个数据库定义了诸如 `$my_db` 的名字，该名字包括美元符号，对来自外壳程序命令行的该名字的任何引用都可由外壳程序解释为一个变量引用：

```
% ls $my_db
my_db: Undefined variable.
```

如果这种情况发生，必须将‘\$’字符换码，或使用引号来取消其特殊的含义：

```
% ls \my_db
% ls '$my_db'
```

如果要使用引号，则应使用单引号。双引号不能取消对变量的解释。

第二，尽管 MySQL 允许数据库和表的名字最大长度为 64 个字符，但名字的长度也要受到操作系统所允许长度的限制。通常，这不是什么问题，尽管在 UNIX 中您可能会进入有 14 个字符限制的旧版本 System V-ish 系统中。在这种情况下，数据库名字的有效限制为 14 个字符，表名的限制为 10 个字符，因为表示表的文件名称可用一个句点和三字符的扩展名终结。

第三，基础文件系统的大小写敏感性影响您对数据库和表的命名及引用。如果文件系统是区分大小写的（如 UNIX），则 `my_tbl` 和 `MY_TBL` 这两个名字涉及不同的表。如果文件系统不是区分大小写的（如 Windows），则 `my_tbl` 和 `MY_TBL` 是同一个表。您应当留意是否使

用了 UNIX 服务器来开发数据库，如果有可能的话，在某时应将该数据库移到 Windows 服务器上。

10.2.5 系统性能的数据目录结构的含义

数据目录的结构易于理解，因为它使用了文件系统的层次结构的方式。同时，该结构具有特定的性能含义，尤其是关于打开表示数据库表文件的操作。

这种数据目录结构的一个后果是，由于表由多个文件来表示，因此每个打开的表都需要多个文件描述符，而不是一个。服务器智能化地高速缓存这些描述符，但是一个繁忙的服务器可能会很轻易地耗尽描述符资源，如果服务器同时为许多并发的客户机连接服务或运行引用多个表的复杂查询的话。文件描述符在许多系统中都是匮乏的资源，尤其是将缺省的总进程（per-process）限制设置得相当低的系统。第11章“常规的 MySQL 管理”将提供有关估计所需描述符数量的信息，以及在需要时重新配置服务器或操作系统的信息。

由表自己的文件表示每个表的另一个后果是，表的打开时间随表的数量而增加。打开表的操作映射成由操作系统提供的文件打开操作，因此将受到系统的目录查找程序（directory-lookup routine）效率的影响。通常这不是个问题，但是，如果在数据库中需要大量的表时，它则是个要考虑的问题。

例如，如果想要得到10 000个表，则数据库目录中应该包含30 000个文件。对于这么多的文件，将会引起由于文件打开操作所花费的时间而使运行速度降低（Linux ext2 和 Solaris 文件系统存在这个问题）。如果这个问题涉及到利害关系，则应根据应用程序的需要明智地重新考虑表的结构，从而重新组织这些表。应查看一下是否真的需要这么多的表，因为有时应用程序会不必要地繁殖许多表。为每个用户创建一个单个表的应用程序将导致许多表的产生，其实所有这些表都有相同的结构。如果您想将这些表合并成一个表，可以通过增加另一列以标识每行所使用的用户来达到目的。如果这能使表的数量明显减少，则会相应提高应用程序的性能。

在数据库设计阶段，您必须考虑这个特定的阶段对于一个给定的应用程序是否是值得的。不按上面所描述的方法来合并表的原因如下：

- **增大的磁盘空间的需求。**合并表是为了减少所需表的数量（减少表打开的时间），但增加了另一列内容（增加磁盘空间的需求）。这是典型的空间与时间的折衷，您需要决定哪个因素更重要。如果认为速度极为重要，您或许愿意牺牲一点额外的磁盘空间。如果空间太紧张，则只能忍受使用多个表的时间。
- **安全性考虑。**这些可能会约束您的能力或对表合并的愿望。每个用户分别使用单独的表的一个原因是：使只有拥有表级权限的用户才能对每个表进行访问。如果合并了表，则所有用户数据都将在同一个表中出现。

MySQL 没有限制一个已知用户对特定行的访问的规定，因此，如果没有泄密访问控制就不能合并表。另一方面，如果所有的数据访问都由应用程序控制（用户不可能直接连接到服务器），则可以合并表并使用应用程序的逻辑强制合并后的行级访问。

MySQL 对于表的大小有其自己内部的限制，但是，由于它将表表示为文件，MySQL 还将受到文件尺寸最大值的限制，该最大值是由操作系统给出的。因此，有效的表尺寸最大值要小于 MySQL 的内部限制和系统文件尺寸的限制。

通常，随着时间的推移，对尺寸大小的约束将有所缓和。例如，IBM AIX4.1 有 2GB 文件大小的限制，但是在 AIX4.2 中该限制值大约为 64GB。在 MySQL 中内部的表大小限制值也随着最新版本的出现而增加。在 3.23 系列之前，内部的限制值为 4GB。从 3.23 系列起，该限制值大约为 9 000 000 太字节。表 10-2 说明了 MySQL 内部的表大小限制和 AIX 文件大小限制怎样相互作用来确定有效的表大小的最大值。类似的相互作用也可应用于其他的操作系统。

表10-2 MySQL 和 操作系统大小限制值之间的相互作用

MySQL 的版本	AIX 版本	表大小的最大值	约束因素
MySQL 3.22.22	AIX4.1	2GB	AIX 文件尺寸的最大值为 2GB
MySQL 3.22.22	AIX4.2	4GB	MySQL 表尺寸的最大值为 4GB
MySQL 3.23.0	AIX4.1	2GB	AIX 文件尺寸的最大值为 2GB
MySQL 3.23.0	AIX4.2	64GB	AIX 文件尺寸的最大值为 64GB

10.2.6 MySQL 的状态文件

除数据库目录外，MySQL 数据目录还包含许多状态文件。表 10-3 概括介绍了这些文件。大多数状态文件的缺省名称从服务器主机名字中生成，在此表中表示为 HOSTNAME。

表10-3 MySQL 状态文件

文件类型	缺省名	文件内容
进程 ID	HOSTNAME.pid	服务器进程 ID
错误日志	HOSTNAME.err	启动和关闭事件和错误状态
常规日志	HOSTNAME.log	连接/断开事件和查询信息
更新日志	HOSTNAME.nnn	修改表的内容或结构的所有查询的文本

服务器在启动时将它的进程 ID (PID) 写入 PID 文件，并在关闭时删除该文件。PID 文件是一种方法，用这种方法，其他的进程可以找到该服务器。例如，如果您在系统关闭时运行 mysql.server 脚本来关闭 MySQL 服务器，则该脚本将检查 PID 文件以确定它需要哪个进程来发送一个终止信号。

错误日志由 safe_mysqld 产生，作为服务器标准错误输出结果的重定向，它包含服务器写入 stderr 的所有消息。这意味着仅当通过调用 safe_mysqld 启动服务器时，错误日志才存在（总之，这是启动服务器的首选方法，因为，如果由于一个错误使错误日志存在，则 safe_mysqld 将重新启动服务器）。

常规日志和更新日志是可选的，可以用 --log 和 --log-update 服务器选项开启需要的日志类型。

常规进程提供有关服务器运作的常规信息：谁从哪里进行了连接，以及他们发布了什么查询。更新日志也提供查询信息，但仅仅是修改过的数据库内容的查询信息。更新日志的内容是一些 SQL 语句，这些语句可以通过将它们输入到 mysql 客户机程序来运行。如果出现崩溃且必须转到备份文件时，更新日志将是有用的，因为您能够通过将更新日志输入到服务器来重复这些自崩溃以来所完成的更新操作。这将使得数据库恢复到崩溃发生时所处的状态上。

下面是一个实例，它是作为一个短客户机会话的结果出现在常规日志中的信息中的，这个会话在 test 数据库中创建一个表，并插入一行到该表中，然后删除该表：

```
990509 7:34:09      492 Connect      paul@localhost on test
                  492 Query        show databases
                  492 Query        show tables
                  492 Field List  tbl_1
                  492 Field List  tbl_2
                  ...
990509 7:34:22      492 Query        CREATE TABLE my_tbl (val INT)
990509 7:34:34      492 Query        INSERT INTO my_tbl VALUES(1)
990509 7:34:38      492 Query        DROP TABLE my_tbl
990509 7:34:40      492 Quit
```

常规日志包含日期和时间、服务器线程 ID、事件类型以及特定事件信息的列。

同一个会话出现在如下的更新日志中：

```
use test;
CREATE TABLE my_tbl (val INT);
INSERT INTO my_tbl VALUES(1);
DROP TABLE my_tbl;
```

对于更新日志，日志的扩展格式是可用的，即使是用 `--log - long - format` 选项。扩展的日志提供有关谁何时发布查询的信息。当然，这将使用更多的磁盘空间，但是，如果您不将更新日志的内容与常规日志中的连接事件相联系就想知道谁正在做什么的话，扩展日志或许是可用的。

对于刚才显示出的会话，扩展日志将产生下列信息：

```
# Time: 990509 7:43:42
# User@Host: paul [paul] @ localhost []
use test;
CREATE TABLE my_tbl (val INT);
# User@Host: paul [paul] @ localhost []
INSERT INTO my_tbl VALUES(1);
# Time: 990509 7:43:43
# User@Host: paul [paul] @ localhost []
DROP TABLE my_tbl;
```

确保日志文件的安全且不被用户任意读取是个好注意。常规日志和更新日志都包含有诸如口令这样的敏感信息，这是因为它们包含了查询的文本。下面是您不想让任何人都能读取的日志项，因为它显示了 root 用户的口令：

```
990509 7:47:24      4 Query        UPDATE user SET Password=PASSWORD("secret")
                                WHERE user="root"
```

有关检查可设置数据目录许可权的信息，请参阅第 12 章。数据目录安全的简短指令由下列命令组成：

```
% chmod 700 DATADIR
```

以拥有该数据目录的 UNIX 用户身份来运行此命令。还要确保服务器以该用户身份运行，否则此命令不仅将其他用户排斥在该数据目录之外（您想要的），还将阻止服务器访问您的数据库（您不要的）。

状态文件出现在数据目录的最高级，就像数据库目录一样，因此您可能会想到那些文件的名字是否会相互混淆或者被误认为是数据库名（例如，当服务器正在执行 `SHOW DATABASE` 语句时）。答案是：不会的。状态和日志信息存储在文件中，而数据库是目录，因此可执行程序可以将它们与一个简单的 `stat()` 调用相区别（是服务器告诉它们怎样区分的）。如果您正在监视数据目录，则可以通过使用 `ls -l` 将状态文件从数据库目录中区分开来，并且

检查该模式信息的第一个字符以查看它是 ‘ - ’ 还是 ‘ d ’:

```
% ls -l DATADIR
total 31
drwxrwx--- 1 mysqladm mysqlgrp 1024 May 8 13:22 bigdb
drwxrwx--- 2 mysqladm mysqlgrp 1024 Dec 15 22:34 mysql
-rw-rw--- 1 mysqladm mysqlgrp 69 May 9 20:11 pit-viper.001
-rw-rw-r-- 1 mysqladm mysqlgrp 24168 May 9 20:11 pit-viper.err
-rw-rw--- 1 mysqladm mysqlgrp 4376 May 9 20:11 pit-viper.log
-rw-r--r-- 1 mysqladm mysqlgrp 5 May 9 20:11 pit-viper.pid
drwxrwx--- 7 mysqladm mysqlgrp 512 Sep 10 1998 sql-bench
drwxrwx--- 2 mysqladm mysqlgrp 512 May 9 07:44 test
```

您还可以通过查看名字而简单地告之：所有状态文件名都包含一个句点，但是数据库目录名没有句点（句点不是数据库名的合法字符）。

有关日志文件维护和循环技术的信息，请参阅第 11 章的内容。

10.3 重定位数据目录的内容

10.2 节讨论了在其缺省配置中的数据目录的结构。所有数据库和状态文件都包含在其中。但是，在确定数据目录内容的布局中管理员有某些职责。本节讨论为什么要移动数据目录的各个部分（甚至是字典本身）、可以移动什么，以及怎样进行这些移动。

MySQL 允许您重定位其中的数据目录或元素。这样做有几个原因：

- 可以用比缺省定位的文件系统更大的容量在文件系统中放置数据目录。
- 如果数据目录在繁忙的磁盘上，可以将其放置到较少使用的驱动器上，以平衡物理设备之间的磁盘活动。为了类似的原因，可以将数据库和日志文件放在不同的驱动器上，或在驱动器之间对数据库进行再分布。
- 您可以运行多个服务器，并且每个服务器都有属于自己的数据目录。这是一种解决总进程文件描述符限制问题的方法，尤其是当不能重新配置系统的核心以得到更高的限制值时。
- 某些系统将 PID 文件保存在诸如 /var/run 的目录中。为了系统运作的一致性，您可以将 MySQL PID 文件也放在那里。

10.3.1 重定位方法

有两种对数据目录重定位的方法：

- 可以在命令行或在一个选项文件的 [mysqld] 组上，在服务器启动时间指定一个选项。
- 可以移动要重定位的内容，然后在原始的位置中做一个指向新位置的 `symlink` (symbolic link, 符号链接)。

两种方法的任何一种都不能为您进行全部的重定位工作。表10-4 综合了可重定位的内容以及可用于重定位的方法。如果您使用一个选项文件，可以指定在全局选项文件/etc/my.cnf（Windows 中的 c:\my.cnf）中的选项。当前的 Windows 版本还访问系统目录（c:\windows 或 c:\NT）。

您还可以使用缺省数据目录的选项文件 `my.cnf`（该目录编译在服务器中）。笔者不建议使用此文件。如果要重定位数据目录本身，必须保持缺省数据目录的完整性，以便在数据目录中放置一个选项文件，该文件将说明服务器应该在哪里找到“真正”的数据目录！真乱。如果想要用一个选项文件来指定服务器的选项，则最好使用 `/etc/my.cnf`。

表10-4 重定位方法概括

重定位的实体	可使用的重定位方法	重定位的实体	可使用的重定位方法
全数据目录	启动选项或symlink	PID 文件	启动选项
单个的数据库目录	symlink	常规日志文件	启动选项
单个的数据库表	symlink	更新日志文件	启动选项

10.3.2 估计重定位的效果

在试图对任何东西进行重定位之前，检验该操作是否将具有期望的效果是一个好主意。笔者倾向于用 du、df 和 ls -l 命令来获得磁盘空间信息，但是所有的命令都依赖于对文件系统布局的正确理解。

以下例子将显示出一个神秘的中断以密切注意何时估计数据目录的重定位。假定数据目录是 /usr/local/var，并且想将其移动到 /var/mysql，因为 df 指出该 /var 文件系统有较多的可用空间（如下例所示）：

```
% df /usr /var
Filesystem 1K-blocks    Used   Avail Capacity  Mounted on
/dev/wd0s3e   396895   292126    73018     80%   /usr
/dev/wd0s3f  1189359  1111924   162287     15%   /var
```

重定位的数据目录能释放 /usr 文件系统中多少空间？为了查找空间数量，可使用 du -s 来查看该目录使用了多少空间：

```
% cd /usr/local/var
% du -s
133426 .
```

这大约为 130MB，应该对 /usr 产生相当大的变化。但这是真的吗？可在该数据目录 /usr 中试一下 df 命令：

```
% df /usr/local/var
Filesystem 1K-blocks    Used   Avail Capacity  Mounted on
/dev/wd0s3f  1189359  1111924   162287     15%   /var
```

真奇怪。我们请求的是包含 /usr/local/var 系统文件的可用空间，可为什么 df 报告了 var 的空间呢？下面的 ls -l 做出了回答：

```
% ls -l /usr/local
...
lrwxrwxr-x 1 root wheel 10 Dec 11 23:46 var -> /var/mysql
...
```

该输出结果表明 /usr/local/var 是对 /var/mysql 的一个 symlink。换句话说，数据目录已经被重定位到 /var 文件系统中，并且用指向 /var 文件系统的 symlink 所取代。有关通过移动数据目录到 /var 中来释放大量的 /usr 空间的工作就到此为止了！

教训：花几分钟的时间估计重定位的效果是一个有价值的投资。不用花很长的时间就会发现您可能不能达到自己的预期目标，这样可以使您避免浪费大量的移动数据目录的时间。

10.3.3 重定位数据目录

为了重定位数据目录，应关闭服务器，将数据目录移动到新的位置。然后应该或者删除原来的数据目录并用指向新位置的 symlink 来代替它，或者使用直接指明新位置的一个选项来重新启动服务器。表 10-5 列出了指定该位置的命令行和选项文件的语法。

表10-5 数据目录重定位的语法

选项来源	语 法
命令行	--datadir=/path/to/dir
选项文件	[mysqld] datadir=/path/to/dir

10.3.4 重定位数据库

数据库只能通过 `symlink` 方法来移动。为了重定位数据库，应关闭服务器，移动数据库目录。删除原来的数据库目录，用指向新位置的 `symlink` 来代替它，然后启动服务器。

下面的例子说明怎样将数据库 `bigdb` 移动到另一个位置：

```
% mysqladmin -u root -p shutdown
Enter password: *****
% cd DATADIR
% tar cf - bigdb | (cd /var/db; tar xf -)
% mv bigdb bigdb.orig
% ln -s /var/db/bigdb .
% safe_mysqld &
```

重定位的预防措施

在执行任何重定位操作之前应该关闭服务器，然后再重新启动它。对有些类型的重定位（如移动数据库目录），保持服务器的运行状态是可能的（尽管不建议这样做）。如果要这样做，您必须确保服务器没有访问将要移动的数据库。还应该确保在移动数据库之前发布了 `FLUSH TABLE` 语句，以便确保服务器关闭所有打开的表文件。不履行这些预防措施可能导致表的毁坏。

应该以数据目录所有者的身份来执行这些命令。为了安全起见，将原来的数据库目录重新命名为 `bigdb.orig`。在验证了服务器与重定位服务器正常工作之后，可以删除原来的目录：

```
% rm -rf bigdb.orig
```

10.3.5 重定位数据库表

对单个的表进行重定位不是好主意。可以通过将表的文件移动到另一个位置并在该数据库目录中创建指向这些文件的 `symlink` 来进行。但是，如果曾经发布过 `ALTER TABLE` 或 `OPTIMIZE TABLE` 语句，则所做的这些修改将被取消。

每个语句通过在数据库目录中创建一个实现变更和最优化的临时表进行操作，然后删除原来的表，将该临时表重新命名为原来的名称。其结果是：`symlink` 被删除，新的表回到数据库目录中，该目录是您移动表之前的原始表所在位置。因此，移出该数据库目录的旧的表文件仍然在原位置上——您甚至已经不记得它们的存在，但它们仍然占用着空间。同样，`symlink` 已经消失，因此，当您意识到所发生的一切时，如果已经不记得是在哪里移动的，将没有任何好办法去捕捉这些文件。

要想确保访问该表的任何人都不更改或优化该表是困难的（因而撤消任何企图的重定位），因此最好保留该数据库目录中的这些表。

10.3.6 重定位状态文件

可以用启动选项重定位 `PID` 文件、常规日志和更新日志。错误日志由 `safe_mysqld` 创建且不能够重定位（除非编辑 `safe_mysqld`）。

为了在另一个位置写状态文件，应关闭服务器，然后用指定新状态文件位置的恰当选项重新启动它。表10-6列出了每个文件的命令行和选项的语法。

删除一个重定位的数据库

您可以用 `DROP DATABASE` 语句删除一个数据库，但是旧版本的 MySQL 在删除已经重定位的数据库时是有难度的。该数据库中的表被正确地删除了，但在服务器试图删除该数据库目录时会出现错误，这是由于该目录是一个 `symlink` 而不是真正的目录。MySQL 管理员必须手工删除该数据库目录和指向它的 `symlink`。自 MySQL 3.23 以来，这个问题已经得到解决。

如果以绝对路径名指定一个状态文件的名称，则用该路径名创建该文件。否则，该文件在该数据目录下创建。例如，如果您指定 `--pid - file = /var/run/mysqld.pid`，则该 PID 文件为 `/var/run/mysqld.pid`。如果您指定 `--pid - file = mysqld.pid`，则该 PID 文件为 `DATADIR/mysqld.pid`。

如果指定一个没有带扩展名的更新日志，则 MySQL 在打开该更新日志时将生成顺序的名字。这些名字用 `.nnn` 扩展名创建，这里的 `.nnn` 是未被已有的更新日志文件使用过的第一个号码（如，`update.001`、`update.002`等等）。可以通过指定包含明确的扩展名来忽略顺序名字的生成，然后服务器将仅使用您指定的这个名字。

表10-6 状态文件重定位的语法

选项来源	语 法
命令行	<code>--pid - file = pidfile</code> <code>--log = logfile</code> <code>--log - update = updatefile</code>
选项文件	<code>[mysqld]</code> <code>pid - file = pidfile</code> <code>log = logfile</code> <code>log - update = updatefile</code>

第11章 常规的MySQL 管理

本章将讨论 MySQL 管理员为保持 MySQL 正常运行所需要完成的职责。这些职责包括确保服务器尽可能保持高性能地运行、设置用户账号以便客户机能够访问服务器、维护日志文件，以及执行数据库备份。为了得到更高的性能，管理员还可以运行多个服务器或修改服务器的操作参数。最后，由于 MySQL 正处于高速发展的状态，所以管理员必须能确定何对 MySQL 进行升级。其他重要的管理问题将在第 12 章“安全性”和第 13 章“数据库维护和修复”介绍。

有几个对 MySQL 管理员有用的程序包括在这些章节中：

- `mysqladmin` 执行各种各样的管理操作。
- `safe_mysqld` 和 `mysql.server` 是启动 MySQL 服务器 `mysqld` 的脚本。
- `mysqldump` 用于数据库备份和拷贝操作。
- `myisamchk` 和 `isamchk` 用于表的完整性检查和修复操作。

有关这些程序的附加信息，请参阅附录 E “MySQL 程序参考”。

11.1 新的 MySQL 安装的安全性

您可能因为刚刚按照附录 A “获得和安装软件”一节的说明安装了 MySQL 而正在阅读本章。如果是这样的话，则需要为 MySQL root 用户设置口令——对于一个新的 MySQL 安装来说，服务器正在以不安全的权限运行着。笔者假定您已经初始化了数据目录和包含授权表的 `mysql` 数据库。在 UNIX 中，可通过运行 `mysql_install_db` 脚本来进行。在 Windows 中，数据目录和 `mysql` 数据库通过运行服务器分包中的 `Setup` 程序初始化。笔者还假定服务器正在运行中。

当初次在机器上安装 MySQL 时，`mysql` 数据库的授权表的初始权限如下：

- 可以从本地主机中以 `root` 进行连接，不带任何口令。`root` 用户拥有所有的权限（其中包括管理权限），因此能做许多事情（顺便说一句，MySQL 超级用户和 UNIX 超级用户二者都有名字 `root`，这一事实是巧合的。它们相互之间没有任何关系）。
- 匿名的访问授予从本地主机上连接的用户，该主机拥有名为 `test` 的数据库或带有以 `'test_'` 开始的任何数据库。匿名用户可以利用这样的数据库做任何事，但没有管理权限。

无论连接的用户指定的是 `localhost` 主机名还是实际的主机名，从本地主机到服务器的连接都将是允许的。例如，如果服务器在 `pit-viper.snake.net` 上，则该主机上的客户机能够不使用口令而连接到该服务器，从而可利用下列之一的语句使用 `test` 数据库：

```
% mysql -h localhost test
% mysql -h pit-viper.snake.net test
```

您甚至可以不用口令就以 `root` 用户的身份连接到 MySQL，这一事实说明：初始的安装是不安全的。因此，作为 MySQL 管理员最初的行动之一应该是为 `root` 用户设置口令。然后，根据用来设置口令的方法，您还需要指示服务器重新加载授权的表，使服务器知道这些变化

(在服务器启动时, 它将这些表加载到内存中, 并且在没有意识到的情况下已经将它们改变。如果是这样的话, 必须明确告诉它重新读取这些表)。

对于 MySQL 3.22 以上的版本, 可以用 `mysqladmin` 程序按如下方式设置口令, 可用您选择的口令来替代下列命令中的 “`your password`”:

```
% mysqladmin -u root password "your password"
```

对于任何版本的 MySQL, 您都可以使用 `mysql` 程序并直接更新 `mysql` 数据库中的用户授权的表:

```
% mysql -u root mysql
mysql> UPDATE user SET Password=PASSWORD("your password")
-> WHERE User="root";
```

如果您有一个旧版本的 MySQL, 则可使用 `mysql` 和一个 `UPDATE` 语句, 这也包括 Windows 的共享软件版本。

在设置口令后, 要看看是否需要通过运行下列命令指示服务器重新加载授权的表:

```
% mysqladmin -u root status
```

如果服务器仍然允许不使用口令就以 `root` 用户身份进行连接, 则应告诉它重新读取授权的表 (MySQL 3.22 以上的版本还允许用 `mysqladmin flush.privileges` 命令和 `FLUSH PRIVILEGES` 的 SQL 语句重新加载表):

```
% mysqladmin -u root reload
```

在设置了 `root` 的口令 (并重新加载授权的表, 如果必要的话) 后, 当以 `root` 的身份连接到服务器时, 需要指定新的口令。

11.2 MySQL 服务器的启动和关闭

作为 MySQL 管理员, 一个普通的目标就是确保服务器尽可能地处于运行状态, 使得客户机能够随时访问它。但是, 有时最好关闭服务器 (例如, 如果正在进行数据库的重定位, 不希望服务器在该数据库中更新表)。保持服务器运行和偶尔关闭它的需求关系不是本书所解决的。但是我们至少可以讨论如何使服务器启动和停止, 以便您具备进行这两个操作的能力。

本章的说明只用于 UNIX 系统。如果正在运行 Windows 系统, 可以跳过本章, 因为附录 A “获得和安装软件” 一节中包含了所有需要的启动和关闭命令。

调用本章给出的命令

为了简洁, 在大多数情况中, 诸如 `mysqladmin`、`mysqldump` 等程序在本章中没有给出任何 `-h`、`-u` 或 `-p` 选项。笔者假定您将会用连接服务器所需的任何选项调用这些程序。

11.2.1 用无特权的用户账号运行 MySQL 服务器

在讨论如何启动服务器之前, 考虑一下在服务器启动时应该运行哪个账号。服务器可以手工和自动启动。如果手工启动, 则服务器以 UNIX 用户身份运行 (您恰好作为该用户进行了注册)。即, 如果笔者以 `paul` 进行注册并启动服务器, 则它将以 `paul` 身份运行。如果用 `su` 命令将用户切换到 `root` 然后启动服务器, 则服务器以 `root` 身份运行。

但是, 大多数时候可能都不会采用手工启动服务器。您很可能将安排服务器在系统引导时作为标准启动过程的一部分自动地运行。在 UNIX 中, 该启动过程由系统以 UNIX 的 `root`

用户的身份执行，该过程中启动的任何进程都用 root 的权限运行。

应该紧记 MySQL 服务器启动过程的两个目标：

- **要服务器以某些非 root 的用户身份启动。**通常，除非进程真的需要 root 访问权而 mysql 办不到，否则应限制任何进程的能力。
- **要服务器始终以同一个用户的身份运行。**服务器有时作为一个用户运行而有时又作为另一个用户运行时会产生矛盾。这将导致文件和目录以不同的所有权在该数据下被创建，甚至引起服务器不能访问数据库或表。以同一个用户的身份一致地运行服务器可以避免该问题。

为了以标准的、非特权的用户身份运行数据库，可按如下步骤执行该过程：

1) 选择用于运行服务器的账号。mysqld 可以以任何用户身份运行，但是很明显，它只为 MySQL 活动创建了一个单独的账号。您也可以为 MySQL 专门指定一个组。笔者将调用的这些用户和组的名字命名为 mysqladm 和 mysqlgrp。如果您使用了其他的名字，则在本书中有 mysqladm 和 mysqlgrp 的地方替换它们

如果您在自己的账号下安装了 MySQL 并且系统中没有特定的管理权限，则您可以在自己的 ID 用户下运行服务器。在这种情况下，应使用您自己的注册名和组名替代 mysqladm 和 mysqlgrp。

如果您利用 RPM 文件在 RedHat Linux 下安装了 MySQL，则该安装程序将在 mysql 名下自动创建了一个账号。应使用该名字替换 mysqladm。

2) 如果必要的话，可用系统常用的账号创建过程（account-creation）来创建服务器账号。这需要以 root 身份进行操作。

3) 关闭服务器（如果它在运行）。

4) 修改数据目录以及任何子目录和文件的所有权，使 mysqladm 用户拥有它们。例如，如果数据目录是 /usr/local/var，则可按以下设置 mysqladm 用户的所有权：

```
# cd /usr/local/var          移动到数据目录
# chown -R mysqladm.mysqlgrp  设置所有目录和文件的所有权
```

5) 修改数据目录以及任何子目录和文件的许可权，使得只有 mysqladm 用户能够访问它们。设置该方式以避免其他人员访问是一种好得安全预防措施。如果数据目录是 /usr/local/var，则可通过 mysqladm 用户按下列操作设置应具有的一切（您需要以 root 身份运行这些命令）：

```
# cd /usr/local/var          移动到数据目录
# chmod -R go -rwx          使所有一切只对 mysqladm 可访问
```

在设置数据目录及其内容的所有权和方式时，观察符号连接。您需要跟踪符号连接并修改所指向的文件或目录的所有权和方式。如果这些连接文件所定位的目录不属于您，则这样做可能会引起麻烦，因此您必须是 root 用户。

在完成前述过程后，应确保无论是作为 mysqladm 还是作为 root 用户注册都始终启动服务器。在后者中，要确保指定了 --user = mysqladm 的选项，使服务器可以将其用户 ID 切换到 mysqladm（该选项在系统启动过程中也可使用）。

--user 选项被增加到 MySQL 3.22 的 mysql 中。如果您的版本比 MySQL 3.22 旧，则在启动服务器并作为 root 用户运行时，可以使用 su 命令指示系统在指定账号下运行服务器。您需要阅读有关 su 的人工页，因为作为一个指定用户运行命令的语法被改变了。

11.2.2 启动服务器的方法

如果您已经确定了用来运行服务器的账号，则可以选择安排怎样启动服务器。可以从命令行手工运行，或在系统启动过程中自动运行服务器。有三种启动服务器的主要方法：

■ **直接调用 mysqld。**这或许是最小的命令方法。除了说明 `mysqld --help` 是一个有用的命令（用它可以查找您可利用其他启动方法使用的选项）外，笔者不打算进一步讨论它。

■ **调用 safe_mysqld 脚本。**`safe_mysqld` 试图确定服务器程序和数据目录的位置，然后利用反映这些位置的选项调用服务器。`safe_mysqld` 将服务器的标准错误输出重定向到数据目录的错误文件中，并以记录的形式出现。在启动服务器后，`safe_mysqld` 还监控服务器，并在其死机时重新启动。`safe_mysqld` 通常用于UNIX 的 BSD 风格的版本。

如果您曾经作为 `root` 或在系统启动程序中启动 `safe_mysqld`，则错误日志将由 `root` 拥有。如果您试着以非特权的用户身份调用 `safe_mysqld`，则可能引起“所有权被拒绝”的错误。删除该错误文件再试一次。

■ **调用 mysql.server 脚本。**通过运行 `safe_mysqld.mysql.server`，该脚本启动服务器。该脚本建议在使用 System V 启动/关闭系统的系统中使用。这个系统包括几个包含在机器登录或退出一个特定运行级时被调用的脚本的目录。它可以利用 `start` 或 `stop` 参数进行调用，以指明希望启动还是关闭服务器。

`safe_mysqld` 脚本被安装在 MySQL 安装目录的 `bin` 目录下，或者在 MySQL 源程序分发包的 `scripts` 目录中。`mysql.server` 脚本安装在 MySQL 安装目录的 `share/mysql` 目录下，或者在 MySQL 源程序分发包的 `support-files` 目录中。如果要使用它，应将其拷贝到合适的启动目录中。

对于 BSD 风格的系统，在 `/etc` 目录中有几个文件相对应，它们在引导期间开始服务。这些文件的名字通常以 ‘`rc`’ 开始，因此很可能会有一个名为 `rc.local`（或类似的名字）的文件来启动本地的安装服务。在这样的系统中，您可能要按如下方法添加一些行到 `rc.local` 文件中以启动服务器（如果路径与您系统中的不同，可将其修改成 `safe_mysqld`）：

```
if [ -x /usr/local/bin/safe_mysqld ]; then
    /usr/local/bin/safe_mysqld &
fi
```

对于 System V 风格的系统，可以通过将其放置在 `/etc` 下的合适的启动目录中来安装 `mysql.server`。如果您运行 Linux 并从 RPM 文件中安装了 MySQL，那么这此操作可能已经完成了。否则，应该在主启动脚本目录中安装该脚本，并在合适的运行级目录中设置对它的连接。您还可使该脚本仅对 `root` 用户可执行。

启动文件目录的布局随系统而变化，因此将需要全面检查来弄清系统是怎样组织它们的。例如，在 LinuxPPC 中，这些目录为 `/etc/rc.d/init.d` 和 `/etc/rc.d/rc3.d`。应该按如下方法安装该脚本：

```
# cp mysql.server /etc/rc.d/init.d
# cd /etc/init.d
# chmod 500 mysql.server
# cd /etc/rc.d/rc3.d
# ln -s ../init.d/mysql.server S99mysql
```

在 Solaris 中，主脚本目录为 `/etc/init.d`，运行级目录为 `/etc/rc2.d`，因此上述命令将替换为：


```
# cp mysql.server /etc/init.d
# cd /etc/init.d
# chmod 500 mysql.server
# cd /etc/rc2.d
# ln -s ../init.d/mysql.server S99mysql
```

在系统启动期间，S99mysql 脚本利用 start 参数自动调用。

如果您拥有 chkconfig 命令（它在 Linux 中很常用），则可用其帮助安装 mysql.server 脚本来代替手工运行上述的命令。

1. 指定启动选项

在启动服务器时，如果想要指定附加的启动选项，可用两种方法进行操作。您可以修改所使用的启动脚本（safe_mysqld 或 mysql.server），并在调用服务器的命令行中直接指定这些选项。您还可以在选项文件中指定选项。笔者建议，如果可能的话，应在全局选项文件中指定服务器选项。通常该文件的位置是 UNIX 中的 /etc/my.cnf 和 Windows 中的 c:\my.cnf（有关使用选项文件的细节，请参阅附录 E）。

某些种类的信息不能作为服务器的选项指定。为了这些选项，您可能需要修改 safe_mysqld。例如，如果服务器不能正确地拾取 GMT 中的本地时区（local time zone）和返回时间值，可以设置 TZ 环境变量以给该变量一个提示。如果用 safe_mysqld 或 mysql.server 启动服务器，可以将时区设置增加到 safe_mysqld 中。找到启动服务器的命令行，并在该行之前增加下列命令：

```
TZ=US/Central
export TZ
```

这个命令将 TZ 设置为 US Central 时区。您需要使用合适位置的时区。该语法是 Solaris 的，您的系统可能会有所不同。例如，设置 TZ 变量的另一个常用语法为：

```
TZ=CST6CDT
export TZ
```

如果修改了启动脚本，当下次安装 MySQL 时（如，升级到更新的版本），将失去这些修改，除非在之前将该启动脚本拷贝到了其他地方。在安装新的版本之后，将您的脚本与新安装的脚本进行比较，以便看看重新建立还需要做什么改动。

2. 在启动期间检查表

除了在系统引导时安排服务器的启动外，您还可以安装一个脚本来运行 mysamchk 和 isamchk，以便在服务器启动前对表进行检查。您可能打算在服务器崩溃后重新启动，但表可能已经损坏了。在服务器启动前检查这些表是发现问题的好办法。第 13 章包含了有关编写和安装这种脚本的细节。

11.2.3 关闭服务器

要想手工关闭服务器，可使用 mysqladmin：

```
% mysqladmin shutdown
```

要想自动关闭服务器，您不需要做特别的操作。BSD 系统通常会通过给进程发送一个 TERM 信号来关闭服务。进程或者对其作出反应，或者被随便地取消。当 mysqld 接收到信号时，它会通过终止来响应。对于利用 mysql.server 启动服务器的 System V-风格的系统，该关闭进程将调用带有 stop 参数的脚本来指示服务器进行关闭——当然，这是在假定您已经安装了 mysql.server 的情况下进行的。

11.2.4 在不连接时收回服务器的控制

在某些环境中，由于不能连接到服务器，您需要用手工重新启动它。当然，这有点荒谬，因为一般是通过连接到服务器然后告知服务器终止来手工关闭服务器的。那么这种情况是怎样出现的？

首先，MySQL 的 root 口令可能得到了一个您不知道的值。这种情况可能是在修改口令时发生的——例如，如果在输入新的口令值时碰巧键入了一个不可见的控制字符。还有可能就是完全忘记了口令。

其次，对于 localhost 的连接通常是通过 UNIX 域的套接字文件进行的，它一般为 /tmp/mysql.sock。如果该套接字文件被删除了，则本地客户机将不能进行连接。如果系统偶尔运行了一个删除 /tmp 中的临时文件的 cron 作业，这种情况就可能会发生。

如果因为失去套接字文件而不能进行连接，可以通过重新启动服务器简单地进行恢复，因为服务器在启动期间重新建立了该文件。这里应知道的是，不能用该套接字建立连接（因为它已经不存在）而必须建立 TCP/IP 连接。例如，如果服务器的主机是 pit-viper.snake.net，则可以按如下方法进行连接：

```
% mysqladmin -p -u root -h pit-viper.snake.net shutdown
```

如果此套接字文件被 cron 作业删除，则问题将复发，直到您修改 cron 作业或使用另一个套接字文件为止。您可以用全局选项文件指定另一个套接字文件。例如，如果数据目录为 /usr/local/var，则可通过将以下行添加到 /etc/my.cnf 中来移动套接字文件到那里：

```
[mysqld]
socket=/usr/local/var/mysql.sock

[client]
socket=/usr/local/var/mysql.sock
```

路径名是为服务器和客户机程序二者所指定的，以便它们能使用相同的套接字文件。如果只对服务器设置路径名，客户机程序将仍然在旧的位置上查找套接字文件。在做出这个修改后应重新启动服务器，使它在新的位置创建套接字文件。

如果由于您忘记了 root 的口令或将其修改为一个您不知道的值而不能进行连接，则需要收回服务器的控制以便重新设置口令：

■ **关闭服务器。**如果您以 root 用户的身份在服务器主机上进行登录，可用 kill 命令终止服务器。通过使用 ps 命令或通过查看服务器的 PID 文件（通常放在数据目录中）能找出服务器的 ID 进程。

最好先试着用标准的 kill 命令取消服务器，该命令将一个 TERM 信号发送到服务器上，以查看服务器是否通过关闭信号来响应。也就是说，表和日志将被适当地刷新。如果服务器被堵塞并且没有响应正常的终止信号，可使用 kill -9 强制终止它。这是最后的一个方法，因为可能存在未刷新的更改，并且要承担非一致状态下将表保留下来的风险。

如果用 kill -9 终止服务器，应确保在重新启动服务器之前利用 myisamchk 和 isamchk 对表进行检查（参见第13章）。

■ **用 --skip-grant-tables 选项重新启动服务器。**该操作告诉服务器不要使用授权的表检查连接。这允许您作为 root 用户不用输入口令即可进行连接。在连接之后，修改 root 的口令。

■ **告诉服务器再利用** `mysqladmin flush-privileges` 使用授权表启动。如果您的 `mysqladmin` 版本不识别 `flush-privileges`，试着进行重新加载。

11.3 用户账号管理

MySQL 管理员应该知道怎样通过指定哪些用户可连接到服务器、从哪里进行连接，以及在连接时做什么，来设置 MySQL 用户账号。MySQL 3.22.11 引入了两个更容易进行这项工作的语句：`GRANT` 语句创建 MySQL 用户并指定其权限，`REVOKE` 语句删除权限。这两个语句充当 `mysql` 数据库中的授权表的前端，并提供直接操纵这些表内容的可选择的方法。

`GRANT` 和 `REVOKE` 语句影响以下四个表：

授权表

`user`
`db`
`tables_priv`
`columns_priv`

内容

可连接到服务器的用户和他们拥有的任何全局特权
数据库级的特权
表级特权
列级特权

还有第五个授权表 (`host`)，但它不受 `GRANT` 或 `REVOKE` 的影响。

当您为某个用户发布 `GRANT` 语句时，应在 `user` 表中为该用户创建一个项。如果该语句指定了所有全局特权（管理权限或用于所有数据库的权限），则这些指定也被记录在 `user` 表中。如果指定了数据库、表或列的权限，它们将记录在 `db`、`tables_priv` 和 `columns_priv` 表中。

使用 `GRANT` 和 `REVOKE` 语句比直接修改授权表更容易。但是，建议您最好通过阅读第 12 章来补充本章的内容，第 12 章中详细讨论了授权表。这些表非常重要，作为一位管理员应该了解这些表是怎样在 `GRANT` 和 `REVOKE` 语句级上工作的。

本节下面的部分将讨论如何设置 MySQL 用户的账号和授权，还将介绍如何取消权限以及从授权表中删除全部用户，并且将考虑一个困扰许多新的 MySQL 管理员的难题。

您还要考虑使用 `mysqlaccess` 和 `mysql_setpermission` 脚本，它们是 MySQL 分发包的组成部分。这些是 Perl 的脚本，它们提供了设置用户账号的 `GRANT` 语句的代用品。`mysql_setpermission` 需要具有 DBI 的支持环境。

11.3.1 创建新用户和授权

`GRANT` 语句的语法如下：

```
GRANT privileges (columns)
ON what
TO user IDENTIFIED BY "password"
WITH GRANT OPTION
```

要使用该语句，需要填写以下部分：

■ `privileges` 分配给用户的权限。下表列出了可在 `GRANT` 语句中使用的权限说明符：权限说明符权限允许的操作

ALTER	更改表和索引
CREATE	创建数据库和表
DELETE	删除表中已有的记录
DROP	删除数据库和表
INDEX	创建或删除索引
INSERT	插入新的记录到表中

REFERENCES	未用
SELECT	检索表中的已有记录
UPDATE	修改已有的表记录
FILE	读写服务器中的文件
PROCESS	查看有关线程在服务器中的执行或取消线程的信息
RELOAD	重新加载授权表或刷新日志、主机高速缓存或表高速缓存
SHUTDOWN	关闭服务器
ALL	任何事。ALL PRIVILEGES 是同义词
USAGE	一个特殊的“无权限”的权限

上表显示的第一组权限说明符适用于数据库、表和列。第二组说明符是管理特权。通常，这些权限的授予相当保守，因为它们会影响服务器的操作（例如，SHUTDOWN 特权不是按每天来分发的权限）。第三组说明符是特殊的。ALL 的意思是“所有的权限”，而 USAGE 的意思是“无权限”——即创建用户，但不授予任何的权限。

■ **columns** 权限适用的列。这是可选的，只来设置列专有的权限。如果命名多于一个列，则用逗号分开。

■ **what** 权限应用的级别。权限可以是全局的（适用于所有数据库和所有的表）、数据库专有的（适用于某个数据库中的所有表），或表专有的。可以通过指定一个 COLUMNS 子句将权限授予特定的列。

■ **user** 使用权限的用户。它由用户名和主机名组成。在 MySQL 中，不仅指定谁进行连接，还要指定从哪里连接。它允许您拥有两个带有相同名字的、从不同位置连接的用户。MySQL 允许在它们之间进行区别并相互独立地分配权限。

MySQL 的用户名就是您在连接到服务器时指定的名字。该名字与您的 UNIX 注册名或 Windows 名的没有必然连系。缺省设置时，客户机程序将使用您注册的名字作为 MySQL 的用户名（如果您不明确指定一个名字的话），但这只是一个约定。有关将 root 作为可以操作一切 MySQL 的超级用户名也是这样，就是一种约定。您也可以在授权表中将此名修改成 nobody，然后作为 nobody 用户进行连接，以执行需要超级用户特权的操作。

■ **password** 分配给该用户的口令。这是可选的。如果您不给新用户指定 IDENTIFIED BY 子句，该用户不分配口令（是非安全的）。对于已有的用户，任何指定的口令将替代旧口令。如果不指定新口令，用户的旧口令仍然保持不变。当您确实要使用 IDENTIFIED BY 时，该口令串应该是直接量，GRANT 将对口令进行编码。当用 SET PASSWORD 语句时，不要使用 PASSWORD() 函数。

WITH GRANT OPTION 子句是可选的。如果包含该子句，该用户可以将 GRANT 语句授予的任何权限授予其他的用户。可以使用该子句将授权的能力授予其他的用户。

用户名、口令以及数据库和表的名称在授权表项中是区分大小写的，而主机名和列名则不是。

通过查询某些问题，通常可以推断出所需的 GRANT 语句的类型：

- 谁可以进行连接，从哪里连接？
- 用户应具有什么级别的权限，这些权限适用于什么？
- 允许用户管理权限吗？

让我们来提问这些问题，同时看一些利用 GRANT 语句设置 MySQL 用户账号的例子。

1. 谁可以进行连接，从哪里连接

您可以允许用户在特定的主机或涉及范围很宽的一组主机中进行连接。在一个极端，如果知道用户将仅从那个主机中进行连接，则可限定对单个主机的访问：

```
GRANT ALL ON samp_db.* TO boris@localhost IDENTIFIED BY "ruby"  
GRANT ALL ON samp_db.* TO fred@ares.mars.net IDENTIFIED BY "quartz"
```

（符号 `samp_db.*` 含义是“在 `samp_db` 数据库中的所有表”）在另一个极端，您可能会有一个用户 `max`，他周游世界并需要能够从世界各地的主机中进行连接。在这种情况下，无论他从哪里连接您都将允许：

```
GRANT ALL ON samp_db.* TO max@% IDENTIFIED BY "diamond"
```

‘%’ 字符起通配符的作用，与 `LIKE` 模式匹配的含义相同，在上个语句中，它的意思是“任何主机”。如果您根本不给出主机名部分，则它与指定“%”的含义相同。因此，`max` 和 `max@%` 是等价的。这是设置一个用户最容易的方法，但安全性最小。

要想采取妥协的办法，可允许用户在一组有限的主机中进行连接。例如，要使 `mary` 从 `snake.net` 域的任何主机中进行连接，可使用 `%snake.net` 主机说明符：

```
GRANT ALL ON samp_db.* TO mary@%.snake.net IDENTIFIED BY "topaz"
```

该用户标识符的主机部分可用 IP 地址而不是主机名给出（如果愿意的话）。可以指定一个直接的 IP 地址或包含模式字符的地址。同样，自 MySQL 3.23 起，可以用一个网络掩码来指定 IP 号，网络掩码表明了用于该网络号的二进制位数：

```
GRANT ALL ON samp_db.* TO joe@192.168.128.3 IDENTIFIED BY "water"  
GRANT ALL ON samp_db.* TO ardis@192.168.128.% IDENTIFIED BY "snow"  
GRANT ALL ON samp_db.* TO rex@192.168.128.0/17 IDENTIFIED BY "ice"
```

第一条语句指明用户可进行连接的特定的主机。第二条语句指定 `129.168.128` Class C 子网的 IP 模式。在第三条语句中，`192.168.128.0/17` 指定一个 17 位二进制的网络号，并将任何主机与其 IP 地址的前 17 个二进制位中的 `192.168.128.0/17` 进行匹配。

如果 MySQL 抱怨您指定的用户值，则可能需要使用引号（但对用户名和主机名分别加引号）：

```
GRANT ALL ON samp_db.president TO "my friend"@boa.snake.net"
```

2. 用户应具有什么级别的权限，这些权限适用于什么

您可授予不同级别的权限。全局权限的功能最强，因为它们适用于任何数据库。为了使 `ethel` 成为可以进行一切操作的超级用户（其中包括可以对其他用户授权），发布下列语句：

```
GRANT ALL ON *.* TO ethel@localhost IDENTIFIED BY "coffee"  
WITH GRANT OPTION
```

`ON` 子句中 `*.*` 说明符的意思是“所有数据库，所有的表”，为保险起见，我们已经指定 `ethel` 只能从本地主机中连接。限制超级用户从哪些主机上进行连接通常是明智的做法，因为它限制住了其他用户对口令进行试探。

有些权限（`FILE`、`PROCESS`、`RELOAD` 和 `SHUTDOWN`）是管理权限，只能用 `NO *.*` 全局权限说明符来授予。如果希望的话，也可以不用授予数据库级的权限来授予这些权限。例如，下列语句建立了一个 `flush` 用户，它除了发布 `FLUSH` 语句外不做其他任何事情。在管理脚本中这可能是有用的，因为需要在这些脚本中执行诸如在日志文件循环期间刷新日志的操作：

```
GRANT RELOAD ON *.* TO flush@localhost IDENTIFIED BY "flushpass"
```

通常授予管理权限应该是保守的，因为具有这些权限的用户可能影响服务器的操作。

数据库级的权限适用于特定数据库中的所有表。这些权限使用 `ON db_name.*` 子句进行授予：

```
GRANT ALL ON samp_db.* TO billeracer.snake.net IDENTIFIED BY "rock"  
GRANT SELECT ON menagerie.* TO ro_user@% IDENTIFIED BY "dirt"
```

第一条语句将 bill 的所有权限授予 samp_db 数据库的任何表。第二条语句创建一个限制访问的用户 ro_user（只读用户），它可以访问 menagerie 数据库的所有表，但只能读取。也就是说，该用户只能发布 SELECT 语句。

怎样在授权表项中指定本地主机名

如果您使用服务器的主机名而非 localhost，通常存在从该服务器主机连接的问题。这可能是由于在授权表中指定名字的方法和名字分解器例程（name resolver routine）向程序报告名字的方法之间的错误匹配。如果分解器报告了一个非限定的名字（如 pit-viper），但授权表包含了具有全限定的名字的项（如 pit-viper.snake.net，反之亦然），则发生错误匹配。

为了确定这种情况是否正在系统中发生，可试着用 `-h` 选项连接到本地服务器，该选项指定了主机的名字。然后查看服务器的常规日志文件。它是怎样报告主机名的？是以非限定形式还是限定形式？不论它是哪种形式，都将告诉您在发布 GRANT 语句时需要怎样指定用户说明符的主机名部分。

可以同时列出许多被授予的单个权限。例如，如果想让用户能读取和修改已有表的内容，但又不允许创建新表或删除表，可按如下授权：

```
GRANT SELECT,INSERT,DELETE,UPDATE ON samp_db.* TO jennie@%  
IDENTIFIED BY "boron"
```

对于更小粒度（fine-grained）的访问控制，可以在单个表上授权，甚至在表的单个列上授权。当存在要对用户隐藏的表时，或者，当只允许用户修改特定列时，列专有的权限是有用的。假定历史同盟会中有一些志愿者利用您作为同盟会秘书应履行的职责来帮助您工作。这是一个好消息，但您决定首先给新的助手授予对 member 表只读的权限（该表中包含了会员资格的信息），然后再对他们增加授予该表的 expiration 列的列专有 UPDATE 权限。也就是说，您的助手可以在人们更新其会员资格时进行更改截止日期的工作。设置此 MySQL 用户的语句如下：

```
GRANT SELECT ON samp_db.member  
TO assistant@localhost IDENTIFIED BY "officehelp"  
GRANT UPDATE (expiration) ON samp_db.member  
TO assistant@localhost
```

第一条语句授予对整个 member 表的读访问权并设置口令。第二条语句增加 UPDATE 权限，但只是对 expiration 列。此时不必要再指定口令，因为在第一条语句中已经完成了。

如果想要为多个列授予列专有的权限，可指定一个列清单，并用逗号将这些列分隔。例如，为了给 assistant 用户增加对 member 表地址列的 UPDATE 权限，可以使用下列语句。新的权限将被增加到对该用户来说已经存在的所有列上：

```
GRANT UPDATE (street,city,state,zip) ON samp_db.member  
TO assistant@localhost
```

通常，不要给用户授予比实际需要更大的权限。但是，当您想要使用户能够创建存储中间结果的临时表，而又不允许用户在包含有他们不能修改的数据的数据库中这样做时，就有

了要在数据库上授予相当多的许可权限的理由。您可以建立一个单独的数据库（笔者称它为 tmp）并授予用户该数据库的所有权限。例如，如果想要 mars.net 域的主机中的任何用户都能够使用 tmp 数据库，可发布下列 GRANT 语句：

```
GRANT ALL ON tmp.* TO "@%.mars.net
```

在完成这些之后，用户可使用 tmp.tbl_name 格式的名字创建和引用 tmp 数据库中的表（用户说明符中的 “ ” 创建一个匿名用户项，任何用户都与空白用户名相匹配）。

3. 允许用户管理权限吗

通过授予数据库所有者数据库的所有权限并在操作时指定 WITH GRANT OPTION，可以允许数据库所有者控制对该数据库的访问。例如，如果要让 alicia 能在 big.corp.com 域的所有主机中进行连接并管理 sales 数据库中所有表的权限，应使用下列 GRANT 语句：

```
GRANT ALL ON sales.*  
TO alicia@%.big-corp.com IDENTIFIED BY "applejuice"  
WITH GRANT OPTION
```

实际上，WITH GRANT OPTION 子句允许将访问的权利授予给另一个用户。要知道，具有 GRANT 权限的两个用户可以相互授予自己的权限。如果只给一个用户授予 SELECT 权限而给另一个用户除 SELECT 外还授予了 GRANT 和其他的权限，则第二个用户可以使第一个用户“强大”起来。

11.3.2 取消权限和删除用户

为了收回某个用户的权限，可使用 REVOKE 语句。除了要用 FROM 替换 TO 并且没有 IDENTIFIED BY 或 WITH GRANT OPTION 子句外，REVOKE 的语法与 GRANT 语句非常相似：

```
REVOKE privileges (columns) ON what FROM user
```

user 部分必须与您想要取消其权限的用户的原始 GRANT 语句的 user 部分相匹配。privileges 部分不需要匹配，您可用 GRANT 语句授权，然后用 REVOKE 语句取消其中的一部分。REVOKE 语句只删除权限，不删除用户。用户的项仍然保留在 user 表中，即使您取消了该用户的所有权限也是如此。这意味着该用户仍然可连接到服务器上。要想删除整个用户，必须用 DELETE 语句将该用户的记录从 user 表中直接删除：

```
% mysql -u root mysql  
mysql> DELETE FROM user  
-> WHERE User = "user_name" and Host = "host_name";  
mysql> FLUSH PRIVILEGES;
```

DELETE 语句删除该用户的项，FLUSH 语句告诉服务器重新加载授权表（当使用 GRANT 或 REVOKE 语句，而不是直接修改授权表时，这些表将自动重新加载）。

11.4 节将讨论为什么删除 user 表项的原因。

一个权限难题，第一部分

下面是一个在 MySQL 邮件清单中反复出现的情况：一位新的 MySQL 管理员给某用户增加一个项，使用了主机名部分，该部分是用一个模式来指定的。例如：

```
GRANT ALL ON samp_db.* TO fred@%.snake.net IDENTIFIED BY "cocoa"
```

这里的意图是允许用户 fred 从 snake.net 域的所有主机中进行连接，并且具有对

samp_db 数据库的所有权限。事实上, fred 能够从那些主机中连接(除了服务器主机本身外)。当 fred 试着从服务器主机中进行连接时, 该企图以“访问被拒绝”的消息而告失败。即使用户指定了正确的口令也是如此。

如果授权表中包含了由 mysql_install_db 安装脚本安装的缺省项, 这种情况也会发生。其原因是, 当服务器验证 fred 连接的企图时, 一个匿名用户项 (anonymous-user entry) 比 fred 项优先。匿名用户项要求该用户不用口令来连接, 并且一个口令错误匹配发生。该问题的另一个背景将在第12章“权限难题, 第二部分”中给出。目前, 只要说修正此问题的方法是从 user 表中删除匿名用户项就足够了, 我们不能用 REVOKE, 因为该命令只删除权限。要想完全摆脱这些匿名项, 执行如下操作:

```
% mysql -u root mysql
mysql> DELETE FROM user where User="";
mysql> FLUSH PRIVILEGES;
```

现在, 当 fred 试图从本地主机连接时成功了。

11.4 日志文件维护

在 MySQL 服务器启动时, 它检查其命令行的操作, 来查看它是否应该执行登录并打开相应的日志文件(如果应该的话)。可以让服务器生成两种主要类型的日志文件:

- **常规日志文件。** 它报告客户机的连接、查询和其他各种各样的事件。它对于跟踪服务器的活动很有用: 谁正在连接、从哪里连接, 以及他们正在做什么。

- **更新日志。** 它报告修改数据库的查询。在此上下文中的术语“更新”不只涉及 UPDATE 语句, 还涉及修改数据库的所有语句。由于这个原因, 它包含了对 DELETE、INSERT、REPLACE、CREATE TABLE、DROP TABLE、GRANT 和 REVOKE 的查询记录。更新日志的内容以 SQL 语句的形式书写, 这些语句用作对 mysql 的输入。如果在崩溃后必须恢复表的话, 更新日志与备份是很有用的。您可以从备份文件中恢复数据库, 然后通过将更新日志作为对 mysql 的输入, 重新运行在该备份文件之后又修改数据库的任何查询。这样, 可将表恢复到崩溃时刻的状态。

为了使日志有效, 可使用 --log 选项开启常规日志, 并用 --log-update 选项开启更新日志。可以在 mysqld.safe_mysqld 或 mysql.server 的命令行中, 或在某个选项的 [mysqld] 组中指定这些选项。当日志有效时, 日志文件在缺省时被写到服务器的数据目录中。

笔者建议在首次使用 MySQL 时应使两种日志类型都有效。在获得一些使用 MySQL 的经验后, 可能会只用更新日志来对付, 以便减少磁盘空间的需求。

在使日志有效后, 要确保不用大量的日志信息将磁盘填满, 尤其是如果服务器正在处理大量的查询话。可使用日志文件循环和截止时间, 在避免日志文件无边界地增长的同时保持最近的几个日志是联机可用的。

日志文件循环工作如下。假定日志文件名为 log。在第一个循环中, log 被重新命名为 log.0, 且服务器开始写新的 log 文件。在第二次循环中, log.0 被重命名为 log.1, log 重命名为 log.0, 服务器开始写另一个新的 log 文件。这样, 每个文件循环通过名字 log.0、log.1, 等等。当文件到达循环的某一点时, 可以终止它。

更新日志和 LOAD DATA 语句

通常，当服务器执行 LOAD DATA 语句时，它只将该语句本身而不是被加载的行内容写到更新日志中。这意味着除非该数据文件仍然保持可访问，否则使用更新日志的恢复操作将是不完整的。为了确保这一点的安全，除非数据库已经备份，否则不应该删除数据文件。

系统备份

更新日志对于数据库恢复并不是任何时候都好，如果一个磁盘崩溃导致您失去了更新日志的话，应确保您执行定期的文件系统备份。将更新日志写到与存储数据库不相同的磁盘中也一个好主意。有关重新加载日志文件的介绍，请参阅第 10 章的“MySQL 数据目录”。

例如，如果您每天都循环日志，并且想保持一周的日志，则应保留 log.0 到 log.6。在下一个循环中，将通过令 log.5 覆盖 log.6 使其成为新的 log.6 来终止 log.6。这样，您就可以保留许多日志而又避免了它们超过磁盘的限度。

日志循环频率和保持的旧日志数量将依赖于服务器的繁忙程度（活动的服务器产生更多的日志信息）以及您希望为旧日志投入多少磁盘空间。当循环常规日志时，可以用 `mysqladmin flush-logs` 命令告诉服务器关闭当前的日志文件并打开新的日志文件。

执行常规日志循环的脚本类似如下（可修改它来反映您的日志基名和数据目录的位置，或许还有希望保留的旧日志的数量）：

```
#!/bin/sh

base=basename
datadir=DATADIR

cd $datadir
mv $base.5 $base.6
mv $base.4 $base.5
mv $base.3 $base.4
mv $base.2 $base.3
mv $base.1 $base.2
mv $base.0 $base.1
mv $base $base.0
mysqladmin flush-logs
```

最好从 `mysqladm` 账号中运行此脚本以确保日志文件属于那个用户。如果在 `.my.cnf` 选项文件中保留连接参数，您不需要在该脚本的 `mysqladmin` 命令中指定任何参数。如果您不这样的话可以建立一个受限用户，它除了发布刷新命令外什么也不做。然后可以以最小的风险在该脚本中放置这个用户的口令。如果想这样做，则该用户应只有 `RELOAD` 权限。例如，要想调用用户 `flush` 并分配一个口令 `flushpass`，可使用下列 `GRANT` 语句：

```
GRANT RELOAD ON *.* TO flush@localhost IDENTIFIED BY "flushpass"
```

当需要在脚本中执行刷新操作时，可以这样做：

```
mysqladmin -u flush -pflushpass flush-logs
```

在 Linux 中，最好用 `logrotate` 来安装 MySQL 分发包中的 `mysql-log-rotate` 脚本，而不是自己编写脚本。如果 `mysql-log-rotate` 不通过 RPM 文件自动安装，应查看 MySQL 分发包的 `support-files` 目录。

由于服务器处理更新日志文件的方法不同，日志文件的循环在更新日志与常规日志之间稍有不同。如果告诉服务器使用没有扩展名的更新日志文件名（如 `update`），则服务器将使用顺序的 `update.001`、`update.002` 等自动创建更新日志文件名。在服务器启动以及在日志刷新时，一个新的更新日志产生。如果您开启更新日志而没有指定文件名，服务器则使用主机名作为基名产生一个更新日志文件的序列。

当终止一个由这种方法生成的文件序列时，您或许想要根据其期限（最后被修改的时间）而非根据名字来终止它们。这样做的理由是由于您不知道 `flush-log` 命令将在何时发布，因此您不能指望在任何给定的时间周期内创建固定数量的更新日志。例如，如果用 `mysqldump` 备份表并使用 `--flush-logs` 选项，在该更新日志名序列中的一个新文件随每个备份一同创建。

对于带有由服务器自动产生的顺序文件名的更新日志，基于日志期限的终止脚本类似如下：

```
#! /bin/sh
```

```
find DATADIR -name "update.[0-9]*" -type f -mtime +6 | xargs rm -f  
mysqladmin flush-logs
```

`find` 命令定位并删除修改时间超过一个星期的更新日志文件。重要的是使用 `-name` 参数来对一个数字的文件扩展名进行测试，以避免删除由错误的 `update` 所指定的表。

还可以告诉服务器使用固定的更新日志文件名（如果希望的话），如果想用与常规日志相同的方法循环更新日志，这是有用的。要想使用固定的更新日志名，应指定一个包含扩展名的名字。例如，可以用 `--log-update=update.log` 选项启动服务器来使用名字 `update.log`。服务器将一直关闭并在接收 `flush-logs` 命令时打开该日志，但是服务器并不是每次都产生新的文件。在这种情况下，用于更新日志的日志循环脚本和用于常规日志的脚本仅在循环的文件基名上有所不同。

如果想自动执行日志循环和终止，可使用 `cron`。假定循环常规日志和更新日志的脚本为 `rotate-logs` 和 `rotate-update-logs`，且安装在 `/usr/user/mysql/bin` 目录中。以 `mysqladm` 用户进行注册，然后用以下命令编辑 `mysqladm` 用户的 `crontab` 文件：

```
% crontab -e
```

此命令允许编辑当前 `crontab` 文件的备份（如果之前没有这样做，则它可能为空）。按以下方法将行增加到该文件中：

```
0 4 * * * /usr/users/mysqladm/bin/rotate-logs  
0 4 * * * /usr/users/mysqladm/bin/rotate-update-logs
```

这个项告诉 `cron` 在每天早上 4 点运行此脚本。您可以改变时间或按需要进行调度。有关说明请参见 `crontab` 的人工页。

11.5 备份和拷贝数据库

重要的是在表丢失和毁坏时备份数据库。如果系统发生崩溃，您就能够将表恢复到崩溃时刻的状态，并尽可能不丢失数据。同样，错发 `DROP DATABASE` 或 `DROP TABLE` 命令的用户可能会向您请求进行数据恢复。有时，这是由 MySQL 管理员引起的破坏，管理员试图通过使用像 `vi` 或 `emacs` 这样的编辑器直接编辑表文件而毁坏了它们。这样做对表来说肯定是干了坏事。

备份数据库的两种主要方法是使用 `mysqldump` 程序或直接拷贝数据库文件（如使用 `cp`、

tar 或cpio)。每种方法都有自己的优点和缺点：

- **mysqldump** 与 MySQL 服务器联合进行操作。直接拷贝方法与服务器相脱离，因此必须采取措施确保在进行拷贝时没有客户机在修改这些表。这个问题与利用文件系统备份来备份数据库的问题相同：如果数据库表在文件系统备份时进行更新，则进行备份的表文件处于非一致的状态，并且对于今后恢复该表没有意义。文件系统备份和直接拷贝文件的区别是：对于后者，您具有控制备份进度的权利，因此可以采取确保服务器使表处于静止状态。
- **mysqldump** 比直接拷贝技术要慢。
- **mysqldump** 产生可移植到其他机器、甚至具有不同硬件结构的机器上的文本文件。直接拷贝文件不能够移植到其他机器上，除非要拷贝的表使用 **MyISAM** 存储格式。**ISAM** 表只能在具有相同硬件结构的机器之间进行拷贝。例如，将文件从 **SPARC** 的 **Solaris** 机器拷贝到 **Intel** 的 **Solaris** 机器（或者相反）是行不通的。由 **MySQL 3.23** 引进的 **MyISAM** 表存储格式可以解决这个问题，因为该格式与机器独立。因此，如果以下两个条件都满足的话，直接拷贝文件可以移植到具有不同硬件结构的机器上：即另一台机器上也必须运行 **MySQL 3.23** 以上的版本，并且文件必须表示成 **MyISAM** 表，而不是 **ISAM** 表。

不论选择哪种备份方法，都有某些原则，您必须坚持这些原则，才能确保在需要恢复数据库内容时得到最好的结果：

- 定期执行备份。设置一个时间表并坚持使用它。
- 告诉服务器运行更新日志。更新日志在您需要恢复崩溃后的数据库时给予帮助。在使用备份文件将数据库恢复到备份时刻的状态后，可以通过运行更新日志中的查询，重新运行备份之后所做的改变。这个操作将数据库中的表恢复到了崩溃时刻的状态。在文件系统备份语言中，数据库备份文件表示完全转储（**full dump**），而更新日志则表示增量转储。
- 使用一致和可理解的备份文件命名模式。像 **backup1**、**backup2** 等名字没有特殊的含义。当需要它执行恢复时，还得浪费时间去查看文件中的内容。您会发现使用数据库名和花时间去构造备份文件名是有帮助的。例如：

```
% mysqldump samp_db > /usr/archives/mysql/samp_db.1999-10-02
% mysqldump menagerie > /usr/archives/mysql/menagerie.1999-10-02
```

在产生备份文件后您可能需要将它们压缩。毕竟备份文件都比较大，所以您可能还需要终止备份文件以避免它们填满磁盘，这与终止日志文件类似。您可以用相同的技术终止备份文件：

- 用文件系统备份来备份您的备份文件。如果您遭受了一个完全崩溃，不仅毁坏了数据目录而且还破坏了包含数据库备份的磁盘驱动器，那将造成真正的麻烦。您还应该备份更新日志。
- 将备份文件放在与您的数据库不同的文件系统上。这将减少含有数据字典的文件系统被生成的备份文件填满的可能性。

创建备份的技术对于将数据库拷贝到另一个服务器上也是很有帮助的。将数据库转移到运行在另一个主机上的服务器是很平常的，您还可以将数据转移到运行在相同主机上的另一个服务器。如果正为一个新版本的 **MySQL** 运行服务器，并且想用成品服务器上的某些真

实数据来测试它时，可能会这样做。还有一种可能，那就是您得到了一台新的机器并要将所有的数据库移动到新机器上。

11.5.1 用 mysqldump 备份和拷贝数据库

当使用 mysqldump 程序产生数据库备份文件时，缺省设置是该文件的内容由 CREATE TABLE 语句组成，这些语句创建被转储的表以及包含表中的行数据的 INSERT 语句。换句话说，mysqldump 创建在今后可作为对 mysql 的输入使用的输出结果，以重建数据库。

可以将整个数据库按以下命令转储到单独的文本文件中：

```
% mysqldump samp_db > /usr/archives/mysql/samp_db.1999-10-02
```

该输出文件的开始类似如下：

```
# MySQL dump 6.0
#
# Host: localhost    Database: samp_db
#-----
# Server version      3.23.2-alpha-log
#
# Table structure for table 'absence'
#
CREATE TABLE absence (
  student_id int(10) unsigned DEFAULT '0' NOT NULL,
  date date DEFAULT '0000-00-00' NOT NULL,
  PRIMARY KEY (student_id,date)
);
#
# Dumping data for table 'absence'
#
INSERT INTO absence VALUES (3,'1999-09-03');
INSERT INTO absence VALUES (5,'1999-09-03');
INSERT INTO absence VALUES (10,'1999-09-06');
...
```

该文件的其余部分由更多的 INSERT 和 CREATE TABLE 语句组成。

如果想在生成备份时进行压缩，可替换成类似下列的命令：

```
% mysqldump samp_db | gzip > /usr/archives/mysql/samp_db.1999-10-02.gz
```

如果您有一个超大数据库，则该输出文件也将是极大的且管理起来很困难。如果您喜欢的话，可以通过在 mysqldump 命令的数据库名之后命名单个的表来转储这些表的内容。这个操作将该转储文件分成更小的、更多的可管理的文件。下面的例子将说明如何将 samp_db 的一些表转储到单个文件中：

```
% mysqldump samp_db student score event absence > gradebook.sql
% mysqldump samp_db member president > hist-league.sql
```

如果您正在生成备份文件并打算用这些备份文件来定期刷新另一个数据库的内容，则可能使用 --add-drop-table 选项。此选项告诉 mysqldump 将 DROP TABLE IF EXISTS 语句写到备份文件中。然后，当您取出该备份文件并将其加载到第二个数据库时，如果表已经存在将不会出现错误信息。如果您正在运行第二个数据库，可使用此技术利用从第一个数据库中的数据拷贝来定期地加载它。

如果您正在转储数据库使该数据库可以转换到另一个服务器上，则无须创建备份文件。应确保该数据库存在于另一台主机上，然后用一个管道使 mysql 直接读取 mysqldump 的输出

结果来转储数据库。例如，如果想要将 samp_db 数据库从 pit_viper.snake.net 拷贝到 boa.snake.net，操作如下：

```
% mysqladmin -h boa.snake.net create samp_db
% mysqldump samp_db | mysql -h boa.snake.net samp_db
```

稍后，如果想要在 boa.snake.net 中再次刷新该数据库，可跳过 mysqladmin 命令，但要将 --add-drop-table 增加到 mysqldump 中，以避免得到有关“表已经存在”的错误：

```
% mysqldump --add-drop-table samp_db | mysql -h boa.snake.net samp_db
```

mysqldump 的其他选项包括如下所示的几个：

- --flush-log 和 --lock-tables 的结合有助于检查数据库。--lock-tables 锁定所有正在转储的表，而 --flush-log 关闭并重新打开更新日志文件。如果正在产生后续的更新日志，则新的更新日志将只包含从备份的那一点开始修改数据库的查询。这时检查对于该备份时间的更新日志的检查点（然而，锁定所有的表对于备份期间客户机访问来说不太好，如果您有需要执行更新操作的客户机的话）。

如果用 --flush-logs 检查对于备份时间的更新日志检查点，最好转储整个数据库。如果转储单个文件，则将更新日志的检查点与备份文件同步是比较难的。在恢复操作中，您通常在总数据库（per-database）的基础上抽取更新日志的内容。对于抽取单个表的更新日志来说没有选项，因此您必须自己抽取它们。

- 缺省设置时，mysqldump 将表的全部内容在读之前读到内存中。这实际上不是必须的，事实上，如果您真的有大型表的话，这几乎是一个失败的方法。可以用 --quick 选项告诉 mysqldump 写每一行（只要是被检索的）。要想进一步优化该转储过程，可用 --opt 来代替 --quick。--opt 选项开启其他的选项，这些选项将加快转储数据和读回数据的速度。

由于快速备份的好处，使得用 --opt 执行备份成为最常用的方法。但是，要当心，--opt 选项有一个代价：--opt 所优化的是您的备份过程，而不是由其他客户机对数据库的访问。--opt 选项可防止任何人更新被锁定的正在转储的任何表。您会很容易地发现在常规数据库访问中在这一点上所做的努力。试着在一天中数据库通常最繁忙的时刻运行一个备份。这不会花费太多的时间。

- 与 --opt 作用有点相反的选项是 -delayed。该选项导致 mysqldump 写 INSERT DELAYED 语句而非 INSERT 语句。如果您将一个数据文件加载到另一个数据库中并且想要使该操作对其他查询（这些查询可能正在数据库中发生）造成的影响最小，则 --delayed 将有助于达到这个目的。

- --compress 选项有助于将数据库拷贝到另一台机器上，因为它可以减少网络传输中的字节数量。这里有一个例子，请注意，为了使程序与远程主机上的服务器进行通信（而不是与本地主机通信），给出了 --compress 选项：

```
% mysqldump --opt samp_db | mysql --compress -h boa.snake.net samp_db
```

mysqldump 有许多选项，详细信息请参考附录 E。

11.5.2 使用直接拷贝数据库备份和拷贝方法

不用 mysqldump 来备份数据库或表的另一种方法是直接拷贝表文件。通常可利用像 cp、tar 或 cpio 这样的实用程序来进行。本节的例子使用的是 cp。

当使用直接拷贝备份（direct-copy backup）方法时，必须确保没有使用这些表。如果在拷贝一个表的同时服务器正在修改它，则拷贝无效。

确保拷贝完整性的最好方法是关闭服务器，拷贝文件，然后重新启动服务器。如果不想关闭服务器，则应参考第 13 章，查阅有关在执行表检查点时锁定服务器的介绍。如果服务器在运行中，则相同的约束都适用于拷贝文件，您应该用同样的锁定协议使服务器保持静止状态。

假定服务器关闭，或者已经锁定了想要拷贝的表，下面的例子将说明怎样将整个 samp_db 数据库备份到备份目录中（DATADIR 代表服务器的数据目录）：

```
% cd DATADIR
% cp -r samp_db /usr/archive/mysql
```

单个表可按如下进行拷贝：

```
% cd DATADIR/samp_db
% cp member.* /usr/archive/mysql/samp_db
% cp score.* /usr/archive/mysql/samp_db
...
```

当完成备份时，可以重新启动服务器（如果已使它关闭），或者释放在表上施加的锁（如果保持服务器运行）。

要想用直接拷贝文件将数据库从一台机器拷贝到另一台机器，只要将这些文件拷贝到另一台服务器主机上的相应数据库上即可。应确保这些文件是对 MyISAM 表的或者两台机器都有相同的硬件结构。否则这些表在第二个主机上看起来好象有很奇怪的内容。还应该确保第二台主机的服务器不会在您安装这些表时去访问它们。

11.5.3 复制数据库

术语“复制”的含义简单地说有点像“拷贝数据库到另一个服务器”，或者是包含在主数据库的内容发生变化时次数据库的有效更新（live updating）的含义。如果想简单地将数据库拷贝到另一个服务器上，则可以使用在前面已经讨论的那些命令。自 MySQL 3.23 版本以来，就已经开始出现对基于有效更新的复制的支持。但它的功能仍未成熟，因此，在这方面笔者没有什么可讨论的，如果有兴趣，您可以注意一下当前的新版本，看看有些什么新的开发功能。

11.6 为数据恢复使用备份

数据库毁坏发生的原因有许多，且程度各不相同。如果幸运的话，可能是一两个表的小毁坏（例如，如果您的机器由于断电而暂时停机）。如果不是这样，可能需要置换整个的数据目录（例如，如果某个磁盘瘫痪而且数据目录在它上）。在其他情况下也需要恢复操作，例如，当用户错误地删除数据库或表时，或者错误地删除表的内容时。不论这些不幸的事件发生是由于什么原因，都需要恢复它们。

如果表被毁坏但没有丢失，可试着用 myisamchk 或 isamchk 来修复它们。如果修复实用程序能修复它们，就根本没有必要使用备份文件。表的修复过程将在第 13 章讨论。如果表被丢失或不能修复，则需要恢复它们。

恢复过程包括两个信息源：备份文件和更新日志。备份文件将表恢复到进行该备份时的状态。但是，在备份和故障发生这段时间中，表通常已经被修改。更新日志包含了用来完成

这些修改的查询。可以通过将更新日志作为对 mysql 的输入来重复这些查询（这就是为什么应该允许更新日志的原因。如果您还没有使更新日志有效，现在赶快做，并在进一步读取之前生成一个新的备份）。

恢复过程根据必须恢复的信息的多少而变化。事实上，恢复整个数据库比恢复单个的表要容易，因为对数据库应用更新日志比对表要容易。

11.6.1 恢复整个数据库

首先，如果要恢复的数据库是含有授权表的 mysql 数据库，将需要使用 `--skip-grant-tables` 选项运行服务器。否则，服务器将抱怨无法找到授权表。在恢复表之后，执行 `mysqladmin flush-privileges` 来告诉服务器加载授权表，并用它们启动。

- 将原数据库目录的内容拷贝到其他的地方。例如，您可能会在稍后用它们进行崩溃表的事后分析检查（post-mortem examination）。
- 用最新的备份文件重新加载数据库。如果您打算使用由 `mysqldump` 加载的文件，则需要将它们作为 mysql 的输入。如果打算使用从数据库中直接拷贝的文件（如，用 `tar` 或 `cp`），则将它们直接拷贝回到该数据库目录中。但是，在这种情况下，应该在拷贝这些文件之前关闭服务器，然后再重新启动它。
- 用更新日志重做在进行备份后又修改了数据库表的查询。对于所有可用的更新日志，可使用它作为 mysql 的输入。指定 `--one-database` 选项，使 mysql 只对想要恢复的数据库执行查询。如果您知道需要使用所有的更新日志文件，可在包含日志的目录中使用下列命令：

```
% ls -t -r -l update.[0-9]* | xargs cat | mysql --one-database db_name
```

`ls` 命令产生更新日志文件的单列列表，更新日志文件根据服务器生成的顺序进行排序（要知道，如果您修改了其中的任何文件，排序的顺序都将改变，这将导致更新日志按错误的顺序使用）。

您很可能必须使用某些更新日志。例如，如果自备份以来所产生的日志命名为 `update.392`、`update.393` 等等，可以重新运行它们中的命令：

```
% mysql --one-database db_name < update.392
% mysql --one-database db_name < update.393
...
```

如果正在运行恢复并打算使用更新日志恢复由于失策的 `DROP DATABASE`、`DROP TABLE` 或 `DELETE` 语句而丢失的信息，应确保先从更新日志中删除这些语句。

11.6.2 恢复单个的表

恢复单个表是很困难的。如果有通过 `mysqldump` 生成的备份文件并且它恰好不包含您想要的表数据，则需要抽取相关的行并用它们作为 mysql 的输入，这部分较容易。困难的是抽取应用于该表的更新日志的片段。您会发现：`mysql_find_rows` 实用程序在这方面有帮助，它可以从更新日志中抽取多行查询。

另一种可能性是用另一个服务器恢复整个数据库，然后将所要的该表的文件拷贝到原始数据库中。这实际很容易！在将文件拷贝回数据库目录时，应确保原始数据库的服务器关闭。

11.7 优化服务器

MySQL 服务器有几个影响其操作的参数（变量）。如果缺省的参数值不合适，可以将其修改为对服务器运行环境更合适的值。例如，如果您有大量的内存，可以告诉服务为磁盘和索引操作使用较大的缓冲区。这将使内存持有更多的信息并减少了必须进行的磁盘访问的数量。如果是一般的系统，可以告诉服务器使用较小的缓冲区，以防止它扰乱系统资源损害其他的进程。

系统变量的当前值可以通过执行 `mysqladmin variables` 命令来检查。变量可利用 `--set-variable var_name = value` 选项在命令行设置（`-o var_name = value` 是等价的）。如果要想设置几个变量，可使用多个 `--set-variable` 选项，还可以使用下列语法在一个选项文件的 `[mysqld]` 组中设置变量：

```
set-variable=var_name=value
```

在附录E的 `mysql` 程序的条款下给出了服务器变量的全部清单。有关性能优化比较常用的变量已在以下列表中给出。您还可以在 MySQL 参考手册的“从 MySQL 中获得最高性能”一章中找到该主题的附加讨论。

- `back_log` 引入的客户机连接请求的数量，这些请求在从当前客户机中处理时排队。如果您有一个很忙的站点，可以增加该变量的值。
- `delayed_queue_size` 此变量控制被排队的 `INSERT DELAYED` 语句中的行数。如果该队列已满，则更多的 `INSERT DELAYED` 将堵塞，直到队列有空间为止，这样可防止发布那些语句的客户机继续进行操作。如果您有许多执行这种 `INSERT` 的客户机，且发现它们正在堵塞，那么应增加该变量，使更多的客户机更快地进行工作（`INSERT DELAYED` 在4.5节“调度与锁定问题”中讨论）。
- `flush_time` 如果系统有问题并且经常锁死或重新引导，应将该变量设置为非零值，这将导致服务器按 `flush_time` 秒来刷新表的高速缓存。用这种方法来写出对表的修改将降低性能，但可减少表讹误或数据丢失的机会。

在 Windows 中，可以在命令行上用 `--flush` 选项启动服务器，以迫使表的修改在每次更新后被刷新。

- `key_buffer_size` 用于存放索引块缓冲区的大小。如果增加该变量值，将加快创建和修改索引操作的时间。值越大 MySQL 就越有可能在内存中查找键值，这将减少索引处理所需的磁盘访问次数。

在 MySQL 3.23 以前的版本中，该变量名为 `key_buffer`。MySQL 3.23 及后来的版本同时识别这两个名字。

- `max_allowed_packet` 客户机通信所使用的缓冲区大小的最大值。如果有客户机要发送大量的 BLOB 或 TEXT 的值，该服务器变量值可能需要增大。

客户机目前使用大小为 24MB 的缺省缓冲区。如果有使用较小缓冲区的旧客户机。可能需要使该客户机的缓冲区大一些。例如，`mysql` 可以按如下调用来指定一个 24MB 信息包的限制值：

```
mysql --set-variable max_allowed_packet=24M
```

- `max_connections` 服务器允许的客户机同时连接的最大数量。如果服务器繁忙，可能

需要增加该值。例如，如果您的 MySQL 服务器被 Web 服务器使用来处理由 DBI 或 PHP 脚本产生的查询，并且还有大量的 Web 通信，如果该变量设置太低的话，则您站点的访问者会发现请求被拒绝。

- `table_cache` 表的高速缓存的大小。增加该值可以使 `mysqld` 保持更多的表，同时打开并减少必须进行的文件打开和关闭操作的次数。

如果增加了 `max_connections` 或 `table_cache` 值的大小，服务器将需要大量的文件描述符。这将引起有关操作系统对文件描述符总进程数量限定的问题，在这种情况下您需要增加该限制值或逐步解决它。由于增加文件描述符数量的限制值，过程会发生变化，所以您可能会在一个运行脚本中使用 `ulimit` 命令时来这样做，该脚本可用于启动服务器，或用于重新配置您的系统。有些系统可以通过编辑系统描述文件来简单地配置和重新引导。对于其他一些系统，则必须编辑一个内核描述文件并重建该内核。如何继续进行下去，请参考您系统的文档。

解决总进程文件描述符限制的一个方法是：将数据目录分离成多个数据目录并运行多个服务器。这样，通过运行多个服务器使可用的描述符数量成倍增长。但另一方面，其他的复杂因素可能会引起问题。由于命名了两个服务器，您不能从一个单个的服务器上访问不同数据目录中的数据库，并且还需要在不同服务器之间复制授权表的权限，以使用户需要访问一个以上的服务器。

有两个变量是管理员为提高性能时常增加的，它们是 `record_buffer` 和 `sort_buffer`。这些缓冲区在连接和分类操作中使用，但其值是属于每个连接的。也就是说，每个客户机都获得属于自己的缓冲区。如果使这些变量的值很大，性能可能会由于昂贵的系统资源的消耗而遭受实际的损失。如果想要修改这些变量，先执行 `mysqladmin variables` 查看一下它们当前的值，然后增量调整其值。这个操作使您能估计为减少严重的性能降低所进行的修改的效果。

11.8 运行多个服务器

大多数人们都在指定的机器上运行单个 MySQL 服务器，但在有些情况下，运行多个服务器是有好处的：

- 您可以在成品服务器运行的状态下测试新版本的服务器。在这种情况下，将运行不同的服务器。
- 操作系统通常将总进程限制施加于打开文件的描述符数量上。如果系统提高该限制值有困难，则运行多个服务器是解决该限制的办法（例如，提高限制可能需要重新编译该内核，并且，如果该机器不是您的管辖范围，可能还不能这样做）。在这种情况下，您可能会运行相同服务器的多个实例。
- 互联网服务经常提供给顾客他们自己的 MySQL 安装程序，它需要单独的服务器。在这种情况下，您可能会运行相同服务器或不同服务器的多个实例，如果不同的顾客需要不同 MySQL 版本的话。

当然，同时运行几个服务器比只运行一个更复杂。如果您打算安装多个版本，则不能将它们都安装在相同的位置。当服务器运行时，某些参数对于每个服务器必须是唯一的。其中有些参数包括在服务器安装的地方：数据目录的路径名、TCP/IP 端口和 UNIX 域的套接字路径名，以及用于运行多个服务器的 UNIX 账号（如果不在相同的账号下运行所有的服务器的话）。如果决定运行多个服务器，则必须保持对正在使用的参数有良好说明，

以便不失去对运行的跟踪。

11.8.1 配置和安装多个服务器

如果打算运行不同版本的服务器而非相同服务器的多个实例，应该在不同的位置安装它们。如果安装二进制（不是 RPM）分发包，它们将在包含版本号的路径名下被安装。如果从源程序分发包中安装，使不同的分发包分开的最容易的办法是，当在每个版本的 MySQL 安装过程中运行 `configure` 时使用 `--with-prefix` 选项。这将导致在单独的目录下进行安装，并且可以连接该目录到该分发包的版本号上。例如，您可以按如下配置一个 MySQL 分发包，这里的 `version` 是 MySQL 的版本号：

```
% ./configure --with-prefix=/usr/local/mysql-version
```

`--with-prefix` 选项还将确定服务器的唯一数据目录。您可能会增加选项来配置其他的服务器专有的值，如 TCP/IP 端口和套接字路径名（`--with-tcp-port` 和 `--with-unix-socket`）。

如果打算运行相同服务器的多个实例，则服务器专用的任何选项将需要在运行时指定。

11.8.2 多个服务器的启动过程

启动多个服务器比使用单个服务器复杂，因为 `safe_mysqld` 和 `mysql.server` 二者均在单个服务器上工作良好。笔者建议您仔细研究 `safe_mysqld` 并用它作为启动过程的基础，除非您使用按自己的需求修改得更精细的拷贝。

必须处理的一个问题是如何在选项文件中指定选项。对于多个服务器，不能在总服务器基础上变化了的设置使用 `/etc/my.cnf`，只能为所有服务器都相同的设置使用该文件。如果每个服务器都有不同的数据目录位置，可以在每个服务器数据目录的 `my.cnf` 文件中指定服务器专有的参数。换句话说，应为所有服务器都使用的设置使用 `/etc/my.cnf`，并且为服务器专有的设置使用 `DATADIR/my.cnf`，这里的 `DATADIR` 随每个服务器变化。

指定服务器选项的另一种方法是：使用 `--defaults-file=path_name` 作为该命令行的第一个选项，以便告诉服务器从 `path_name` 指定的文件中读取选项。这样，可以将该文件中的服务器选项唯一地放置到那个服务器中，然后告诉服务器在启动时读取该文件。请注意，如果指定该选项，则没有任何通常的选项文件（如 `/etc/my.cnf`）将被使用。

11.9 更新 MySQL

MySQL 首次公开发行的是 3.11.1 版。目前流行的是 3.22 系列稳定版的发行版和 3.23 系列开发版的发行版。稳定版系列号总是比开发版系列的要小。在 3.23 稳定之后，将开始使用 3.24 作为开发系列。MySQL 开发者好像在夜以继日地工作着，版本更新相当频繁（一年有几次）。稳定版和开发版都发布更新版本。正在进行开发的速度给 MySQL 管理员带来了这样的问题：当新版本出现时是否应该对您已有的 MySQL 安装进行升级。本节提供了帮助您作出该项决定的一些指导。

在新版本出现时应该做的第一件事是查找它与前一个版本之间有什么不同。检查 MySQL 参考手册中的附录“变化说明”，使自己了解这些差别。然后给自己提出下列问题：

- 您在当前版本中遇到过新版本修正了的问题吗？
- 新版本中有您想要的附加功能吗？

■ 对于您使用的某些类型的操作其性能提高了吗？

如果对所有这些问题的回答都是否定的，则没有任何强制升级的必要。如果所有的回答是肯定的，可能要继续进行下一步工作。这时，一般应等待几天，观察 MySQL 邮件清单，看看是否有关于其他人使用新版本的报道。

可帮助您做出决定的某些要考虑的其他因素如下：

■ 稳定版系列的发行版通常都是对已有的错误进行修正，很少有新功能。通常在稳定版系列中升级的风险要比在开发版系列中的小（当然，如果正在运行开发版系列的服务器，可能根本不关心这个风险）。

■ 如果对 MySQL 进行升级，可能还要对用其内部的 MySQL C 客户机库建立的其他程序进行升级。例如，在 MySQL 升级之后，还需要重新建立 PHP、Apache 和 Perl DBD::mysql 模块，将新的客户机库连接到那些程序中（当所有 MySQL 相关的 DBI 和 PHP 脚本在您升级 MySQL 后开始转储核心时需要这样做）。重建这些程序通常不是什么了不起的事，但是，如果想避开它，最好别对 MySQL 进行。如果您使用静态连接程序而非动态连接程序，该问题的可能性将明显减少。但是，系统内存的需求将增加。

如果仍然不能肯定是否升级，则独立于当前的服务器来测试新的服务器总是可以的。可以通过或者将新服务器与成品服务器并行运行来测试，或者在另一台机器上安装新服务器进行测试。如果使用不同的机器，容易在服务器之间进行独立的维护。如果没有另一台用于测试的机器，可以在成品机上运行新的服务器。如果这样做，则必须用不同的参数值（如安装位置、数据目录以及服务器可以舰艇连接的网络端口和套接字）来运行新的服务器。

无论是哪种情况，您都可能会用已有数据库中的数据拷贝来测试新服务器。

如果作出了升级决定，应查看在 MySQL R 参考手册的“变化说明”附录中是否有关于升级必须采取的特殊步骤的说明。通常不会有，但总之最好检查一下。

不要害怕使用开发版的发行版

您可能不喜欢利用自己的成品数据库来使用开发的发行版，但是笔者鼓励您至少应试着用一下单独的测试服务器，或许再使用一下您的成品数据库备份。试用新的发行版的人越多，对发现错误就越有帮助。对于某些数据库产品，发现错误是一件可怕的事情。而对于 MySQL 来说，错误的报告是促进开发前进的重要因素，因为开发人员可根据用户团体所报告的问题进行实际修正。

第12章 安全性

本章主要讨论作为 MySQL 管理员的您，在维护 MySQL 安装的安全性和完整性方面能够做些什么。我们已经在第 11 章中略微谈到了一点安全性问题，如设置初始的 MySQL root 的口令的重要性以及怎样建立用户账号。这些内容是作为启动和运行安装过程的一部分被讨论的。在本章中，我们将更详细地讨论以下与安全性相关的问题：

- 为什么说安全性是重要的，应该警惕哪些攻击？
- 从服务器主机中的用户那里您将面临什么风险（内部安全性），能做什么？
- 从在网络上连接到服务器的客户机那里您将面临什么风险（外部安全性），能做什么？

MySQL 管理员有责任保护数据库内容的安全，使得记录只能由经过严格认证的那些用户访问。这包括内部安全性和外部安全性。

内部安全性关心文件系统级的问题，如保护 MySQL 数据目录免遭拥有运行服务器的机器账号的用户的攻击。但是，如果数据目录内容的文件许可权过分随意，有人可以将对应这些表的文件进行简单的替换的话，内部安全性就不能很好地确保适当建立对网络上客户机访问的授权表的控制。

外部安全性关心客户机从外部连接的问题，如防止 MySQL 服务器免遭通过网络进来的通过服务器的连接请求对数据库内容访问的攻击。要建立 MySQL 授权表使得它们不允许对服务器所管理的数据库的访问（除非提供了有效的名字和口令）。

本章提供了应该了解的有关问题的指导，并说明如何防止内部和外部级别中未认证的访问。

12.1 内部安全性：安全数据目录访问

MySQL 服务器提供了一个通过 mysql 数据库中的授权表来实现的灵活的权限系统。可以设置这些表的内容来允许或拒绝数据库对客户机的访问。这提供了关于未认证的网络访问数据的安全性。但是，如果服务器主机上的其他用户具有对该数据目录内容的直接访问权，则将不能对访问数据的网络建立良好的安全性。除非知道您是曾在运行 MySQL 服务器的机器上注册的惟一的一个人，否则需要关心在该机器上的其他用户获得对数据目录访问的可能性。

以下是您想要保护的内容：

- **数据库文件。**显然想要维护由服务器维护的数据库的保密性。数据库的所有者通常要考虑数据库内容的专有性。即使他们不考虑，也最多是使数据库的内容公共化，而不会使那些内容因数据库目录安全性低而被泄露。
- **日志文件。**常规和更新日志必须安全，因为它们包含了查询文本。这有相当的利害关系，因为具有日志文件访问的任何人都可以监控发生在数据库中的事务处理。

与日志文件有关的更为特殊的安全性问题是，像 GRANT 和 SET PASSWORD 这样的查询被记录在日志中了。常规和更新日志文件包含敏感的查询文本，其中包括了口令（MySQL 使用口令加密，但这只适用于在口令设置之后的连接建立。设置口令的

过程包含在 GRANT、INSERT 或 SET PASSWORD 这样的查询中，但这些查询以纯文本的形式被记录。) 如果一个攻击者具有对日志的读访问权，那他只需在日志中对 GRANT 或 PASSWORD 这样的词运行 grep 就能找到敏感信息。

显然，您不想让服务器主机上的其他用户拥有对数据目录文件的写访问权，因为那样的话，他们就可以在状态文件或数据库表上肆意践踏。但读访问也很危险。如果表文件可读取，那么窃取文件并使 MySQL 自己以纯文本的形式显示表的内容是微不足道的事。可按下列步骤进行：

1) 在服务器主机上安装您的 MySQL 服务器，但使用与正式服务器不同的端口、套接字和数据文件。

2) 运行 mysql_install_db 初始化您的数据目录。这将允许您作为 MySQL 的 root 用户访问服务器，因此您将具有完全控制服务器访问机制的权利。它还建立了一个 test 数据库。

3) 将您想窃取的表的相应文件拷贝到服务器数据目录下的 test 子目录中。

4) 启动作案服务器。您可以随意访问这些表。SHOW TABLES FROM test 将显示您拥有一个被窃取表的备份，SELECT * 将显示任何这些表的全部内容。

5) 如果更坏一点，打开服务器的匿名用户账号的许可权，使任何人都能从任何地方连接到该服务器来访问您的 test 数据库。现在，您已经向全世界公布了这些被偷窃的表。

考虑一下刚才的情况，然后颠倒过来想。您希望有人对您这样做吗？当然不要。

通过在数据目录中执行 ls -l 可以确定数据目录中是否包含非安全的文件或目录。应查看具有以开启的“组”或“其他”许可权的文件或目录。以下是一个非安全数据目录的部分列表，是该数据目录中的一部分数据库目录：

```
% ls -l
total 10148
drwxrwxr-x 11 mysqladm wheel      1024 May  8 12:20 .
drwxr-xr-x 22 root      wheel      512 May  8 13:31 ..
drwx----- 2 mysqladm mysqlgrp    512 Apr 16 15:57 menagerie
drwxrwxr-x 2 mysqladm wheel        512 Jan 25 20:43 mysql
drwxrwxr-x 7 mysqladm wheel        512 Aug 31 1998 sql-bench
drwxrwxr-x 2 mysqladm wheel      1536 May  6 06:11 test
drwx----- 2 mysqladm mysqlgrp    1024 May  8 18:43 tmp
...
```

正如您所看到的，有些数据库目录有正确的许可权，而有些则不是这样。本例中的情况是由于时间引起的。较老的服务器创建了限制较少的许可权，且较老的服务器与较新的服务器相比，在设置许可权方面不严格（请注意，有更多限制的目录，menager 和 tmp，都有更为新的日期）。MySQL 当前的版本确保这些文件只对服务器运行的用户可读。

让我们来安排这些许可权，使得只有服务器的用户才能访问它们。主要的保护手段来自 UNIX 文件系统本身提供的工具，这些工具可设置文件和目录的所有权及方式。操作步骤如下：

1) 定位到数据目录中：

```
% cd DATADIR
```

2) 设置该数据目录下所有文件的所有权为运行该服务器的账号所拥有（必须以 root 身份执行这一步）。在本书中，笔者对此账号的用户名和组名使用 mysqladm 和 mysqlgrp。可以用下列命令之一修改所有权：

```
# chown -R mysqladm.mysqlgrp .
# find . -follow -type d -print | xargs chown mysqladm.mysqlgrp
```

3) 修改数据目录和数据库目录的方式,使得它们仅对于 `mysqladm` 是可读的。这样防止了其他用户访问数据目录的内容。可以利用下列命令之一来进行,这些命令或者以 `root` 或者以 `mysqladm` 运行(后者更好,可以使作为 `root` 运行的命令数量最小化):

```
% chmod -R go-rwx .
% find . -follow -type d -print | xargs chmod go-rwx
```

4) 对 `mysqladm` 用户设置数据目录内容的所有权和方式。现在,您应该确保总是以 `mysqladm` 运行,因为它现在是唯一拥有该数据目录访问权的用户。作为非 `root` 用户运行服务器的过程已在第 11 章中介绍。

在上述步骤之后,将拥有以下许可权:

```
% ls -l
total 10148
drwxrwx--- 11 mysqladm mysqlgrp 1024 May 8 12:20 .
drwxr-xr-x 22 root wheel 512 May 8 13:31 ..
drwx----- 2 mysqladm mysqlgrp 512 Apr 16 15:57 menagerie
drwx----- 2 mysqladm mysqlgrp 512 Jan 25 20:43 mysql
drwx----- 7 mysqladm mysqlgrp 512 Aug 31 1998 sql-bench
drwx----- 2 mysqladm mysqlgrp 1536 May 6 06:11 test
drwx----- 2 mysqladm mysqlgrp 1024 May 8 18:43 tmp
...
```

12.2 外部安全性:安全网络访问

MySQL 安全性系统是灵活的。它允许以许多不同的方法设置用户访问权限。通常,可通过 `GRANT` 和 `REVOKE` 语句来进行,这些语句对控制客户机访问的授权表进行修改。但是,您拥有的可能是不支持这些语句的旧版本 MySQL (这些语句在 MySQL 3.22.11 以前的版本中没有使用),或者可能发觉用户的权限好像不是按希望地在工作。对于这样的情况,了解 MySQL 授权表的结构以及服务器怎样使用它们来决定访问许可权是有帮助的。您了解到这样一个程度,就可以通过直接修改授权表来增加、删除或修改用户的权限,还可以在检查表时诊断权限的问题。

笔者假定您已经阅读了第 11 章的“用户账号管理”一节,并理解 `GRANT` 和 `REVOKE` 语句是怎样工作的。`GRANT` 和 `REVOKE` 提供了建立 MySQL 账号和相关权限的便利方法,但是,它们只是一个前端。所有真正的操作都发生在 MySQL 授权表中。

12.2.1 MySQL 授权表的结构和内容

在网络上连接到服务器的客户机对 MySQL 数据库的访问是由授权表内容控制的。这些表定位在 `mysql` 数据库中,并在首次安装 MySQL 的过程中进行初始化(如附录 A 所描述的)。表12-1和表12-2示出列五个授权表,它们是 `user`、`db`、`host`、`tables_priv` 和 `columns_priv`。

表12-1 `user`、`db` 和 `host` 授权表的结构

访问列的活动范围		
user	db	host
Host	Host	Host

(续)

User	Db	Db
Password	User	
	数据库/表的权限列	
Alter_priv	Alter_priv	Alter_priv
Create_priv	Create_priv	Create_priv
Delete_priv	Delete_priv	Delete_priv
Drop_priv	Drop_priv	Drop_priv
Index_priv	Index_priv	Index_priv
Insert_priv	Insert_priv	Insert_priv
References_priv	References_priv	References_priv
Select_priv	Select_priv	Select_priv
Update_priv	Update_priv	Update_priv
	管理权限列	
user	db	host
File_priv	Grant_priv	Grant_priv
Grant_priv		
Process_priv		
Reload_priv		
Shutdown_priv		

授权表的内容按下列各项使用：

■ user user 表列出可连接到服务器和口令的用户，并指定用户拥有哪些全局（超级用户）权限（如果有的话）。任何在 user 表项中所允许的权限都是全局权限，并适用于所有的数据库。例如，如果在这里允许 DELETE 权限，则在该项中列出的用户可从任何表中删除记录。因此，在进行这项操作之前千万要小心，通常，最好在 user 表项中关闭所有的权限，而使用其他的、有更多限制的表来指定权限。对超级用户（如 root）则是一个例外，但一般很少有。

■ db db 表列出数据库以及哪些用户拥有访问这些数据库的权限。这里指定的权限适用于数据库中所有的表。

■ host host 表与 db表结合使用，在更细的级别上控制对特定主机的数据库访问权限。该表不受 GRANT 和 REVOKE 语句的影响，因此您会发现根本不会去使用它。

■ tables_priv tables_priv 表指定表级的权限。在这里所指定的权限适用于表中所有的列。

■ columns_priv columns_priv 表指定列级的权限。在这里所指定的权限适用于表中特定的列。

在12.2.4节“不用 GRANT 建立用户”中，我们将讨论 GRANT 语句怎样修改这些表以及怎样通过直接修改授权表来达到相同的结果。

tables_priv 和 columns_priv 表是在 MySQL 3.22.11 中引入的（与 GRANT 语句同时）。如

表12-2 table_priv 和 columns_priv 授权表的结构

访问列的活动范围	
tables_priv	columns_priv
Host	Host
Db	Db
User	User
Table_name	Table_name
Column_name	
	权限列
Table_priv	Column_priv

果您拥有 MySQL 的旧版本, 则 mysql 数据库只有 user、db 和 host 表。如果已经从旧版本升级到 3.22.11 以上的版本, 但仍没有 tables_priv 和 columns_priv 表的话, 可运行 mysql_fix_privileges_table 脚本创建它们。

没有 rows_priv 表, 因为 MySQL 不提供记录级的权限。例如, 不能使用户只局限于某些列中包含某个值的表中的那些行。如果需要这个能力, 必须编写应用程序。如果要想执行咨询记录级锁定 (advisory record-level locking), 可用附录 C 中介绍的 GET_LOCK() 函数进行。

授权表中包含两种类型的列: 作用域列 (scope column) 和权限列 (privilege column), 前者决定一个项何时可用, 后者决定一个项可授予哪些权限 (有些授权表中包含其他各式各样的列, 但在这里与我们无关)。

1. 授权表的作用域列

授权表作用域列指定表项何时使用。每个授权表项中都包含 User 和 Host 列, 以指明该项在特定的用户从特定的主机上连接时应用 (host 表是一个例外, 它被用于一种特定的、我们还未接触过的方法中)。其他表中包含附加的作用域列。例如, db 表中包含一个 Db 列以指明该项适用于哪个数据库。同样, tables_priv 和 columns_priv 表中包含使该作用域对数据库中特定表或表中的列缩小的作用域字段。

2. 授权表权限列

授权表中还包含权限列。这些列指明了哪些权限被在作用域列中指定的用户所拥有。MySQL 支持的权限显示在以下列表中。该列表使用用于 GRANT 语句的权限名。对于大部分来说, 在 user、db 和 host 表中权限列的名字与在第 11 章利用 GRANT 语句连接中所讨论的权限名显然是相同的。例如, Select_priv 列对应于 SELECT 权限。

3. 数据库和表的权限

以下权限适用于数据库和表的操作:

- ALTER 允许使用 ALTER TABLE 语句。这实际上是一个简单的一级权限, 您必须根据打算用该表做什么以拥有其他的权限。
- CREATE 允许创建数据库和表。这个权限不允许您创建索引。
- DELETE 允许从表中删除已有记录。
- DROP 允许删除数据库和表。不允许删除索引。
- INDEX 允许创建或删除表的索引。
- INSERT 允许在表中插入新的记录。
- REFERENCES 未使用。
- SELECT 允许用 SELECT 语句从表中检索数据。该权限对于不含表的 SELECT 语句是不必要的, 如 SELECT NOW() 或 SELECT 4/2。
- UPDATE 允许修改表中的已有记录。

4. 管理权限

下列权限适用于控制服务器操作或用户授权能力的管理操作:

- FILE 允许服务器在服务器主机上读写文件。该权限如果没有正当的原因则不应授予, 如12.2.3节“授权表应避免的风险”中所述, 它是危险的。服务器采取某些预防措施限定该权限在特定范围内使用。您只可以读世人都可读的文件, 因此不需要考虑任何方

式的保护。正在进行写操作的文件不允许是已经存在的。这可防止服务器覆盖重要的文件（如 /etc/passwd）或属于其他用户的数据库文件。例如，如果不施加该约束，您可能会替换全部mysql数据库中的授权表的内容。

如果您授予 FILE 权限，就不能作为 UNIX 的 root 用户运行服务器，因为 root 能在文件系统的任何地方创建新的文件。如果作为未授权用户运行服务器，则服务器只在该用户可访问到的目录中创建文件。

■ GRANT 允许将您所拥有的权限授予其他用户，其中包括 GRANT 权限。

■ PROCESS 允许通过使用 SHOW PROCESSLIST 语句或 mysqladmin processlist 命令查询有关正在服务器中运行的线程信息。该权限还允许利用 KILL 语句或 mysqladmin kill 命令取消线程。您始终可以查看或取消自己的线程。PROCESS 权限授予您对任何线程进行这些操作的能力。

■ RELOAD 允许执行多种服务器管理操作。可发布 FLUSH SQL 语句，还可执行 mysqladmin 的命令 reload、refresh、flush-hosts、flush-logs、flush-privileges 和 flush-tables。

通常这不是有危险的权限，尽管它实际上是管理权限。

■ SHUTDOWN 允许利用 mysqladmin shutdown 关闭服务器。

在 user、db 和 host 表中，每个权限都作为单独的列给出。这些列都是用 ENUM（“N”、“Y”）类型声明的，因此所有权限的缺省值为“N”（off）。tables_priv 和 columns_priv 表的权限由 SET 语句表示，它允许权限以使用单个列的任何组合形式指定。这两个表比其他三个表新，这就是为什么它们使用了更有效的表示方法的原因（user、db 和 host 有可能在将来被改造，也通过 SET 来表示权限）。

table_priv 表的 table_priv 列定义如下：

```
SET('Select','Insert','Update','Delete','Create','Drop',  
    'Grant','References','Index','Alter')
```

columns_priv 表的 Column_priv 列定义如下：

```
SET('Select','Insert','Update','References')
```

列权限比表权限少得多，因为在列级中有意义的操作很少。例如，您可以创建一个表，但不能创建孤立的列。

user 表中包含对某些权限的列，这些权限不出现在任何其他授权表中，如 File_priv、Process_priv、Reload_priv 和 Shutdown_priv 表。这些权限适用于执行与任何特定数据库或与表无关的服务器的操作。例如，使用户根据当前数据库关闭服务器是无意义的。

12.2.2 服务器如何控制客户机的访问

在使用 MySQL 时，客户机访问控制有两个阶段。第一阶段发生在连接服务器时。服务器查找 user 表看看是否能够找到与您的名字、正在连接的主机以及所提供的口令相匹配的项。如果不匹配，则不能连接。若匹配，则建立连接，然后继续进行第二阶段。在此阶段中，对于您发布的每个查询，服务器都会检查授权表以查看您是否具有充足的权限来执行该查询。第二阶段继续，直到关于该服务器的会话结束为止。

本节详细讨论 MySQL 用来将授权表项与输入的客户机连接请求和与查询相匹配的规则。这包括在授权表作用域列中合法的值的类型、在授权表中权限信息组合的方法，以及表项检

查的次序。

1. 作用域列的内容

某些作用域列需要直接量，但大多数列允许通配符或其他特殊的值。

■ **Host** 该列值可以是一个主机名或一个 IP 号。localhost 值的含义是本地主机，但仅当您实际使用 localhost 主机值而不是使用该主机名连接时才进行匹配。假定本地主机名为 pit-viper.snake.net，且在 user 表中有两个项，一个带有 Host 值或 localhost，而另一个带有 pit-viper.snake.net 值。带有 localhost 值的项仅当您连接到 localhost 时才进行匹配。另一个项仅当您连接到 pit-viper.snake.net 时才匹配。如果想让用户能够以任意一种方式进行连接，在 user 表中需要两个项。还可以用通配符指定 Host 的值。SQL 的模式字符 ‘ % ’ 和 ‘ _ ’ 也可以使用，并与查询中的 LIKE 操作符有相同的含义。SQL 模式（不允许 REGEX 模式）字符可用于名字和 IP 号。例如，%.wise.edu 将与 wise.edu 域中的任何主机相配。同样，192.168.% 与 192.168 类 B 子网中的任何主机相配，因而 192.168.3.% 与 192.168.3 类 C 子网中的任何主机相配。

值 ‘ % ’ 总是与任何主机相配，用于允许某个用户从任何地方进行连接。空白的 Host 的值与 ‘ % ’ 含义相同（例外：在 db 表中，空白的 Host 值含义是“检查 host 表的进一步的信息”。

自 MySQL 3.23 以来，还可以利用表明网络号的二进制位数的网络掩码指定 IP 号。例如，192.168.128.0/17 指定 17 位的网络号并用 IP 地址的前 17 位中的 192.168.128 与所有主机相配。

■ **User** 用户名必须或者是直接量或者是空白的（空）。空白值与任何用户相配。作为 User 值的 % 不意味着空白，相反，它与带有 % 的直接名字的用户相配，这可能不是您想要的东西。

当一个输入的连接对应 user 表进行检查并且相配的项包含一个空白的 User 值时，客户机被认为是匿名用户。

■ **Password** 口令值或者是空白的（空）或者是非空的，但不允许通配符。空白的口令并不是说与任何口令相配。它的意思是用户不必指定口令。口令以加密值而不是以直接值被存储。如果在 Password 列存储了一个直接的口令，则该用户将不能连接！GRANT 语句和 mysqladmin password 命令自动对口令进行加密，但是，如果使用像 INSERT、REPLACE、UPDATE 或 SET PASSWORD 这样的语句，则必须用 PASSWORD(“NEW_PASSWORD”) 而不是简单地用 “new_password” 来指定口令。

■ **Db** 在 columns_priv 和 tables_priv 表中，Db 值必须是直接的数据库名，不允许模式和空白名。在 dn 和 host 表中，Db 值可以用直接量指定，或通过使用指定通配符的 SQL 模式字符 ‘ % ’ 或 ‘ _ ’ 来指定。% 值或空白将与任何数据库相配。

■ **Table_name、Column_name** 这些列中的值必须是直接量的表名或列名，模式值和空白是不允许的。

有些作用域列由服务器视为区分大小写的，其他的则不是。这些规则已在表 12-3 中总结。特别注意，Table_name 值始终是区分大小写的，即使查询中的表名处理是根据服务器运行的文件系统，其大小写敏感性也是如此（在 UNIX 中区分大小写，而在 Windows 中不区分大小写）。

表12-3 授权表作用域列大小写敏感性

列	是否区分大小写	列	是否区分大小写
Host	No	Db	Yes
User	Yes	Table_name	Yes
Password	Yes	Column_name	No

2. 查询访问检查

每当您发布查询时，服务器都要检查您是否有足够的权限来执行查询。它是通过依次检查 user、db、table_priv 和 columns_priv 表来进行的，直到它确定您有适当的访问权或徒然地搜索了所有的表为止。更准确地说：

- 1) 在开头进行连接以查看所拥有的全局权限时，服务器检查相匹配的 user 表项。如果有全局权限且满足该查询，则服务器执行查询。
- 2) 如果全局权限不满足，则服务器在 db 表中寻找一个项，并将该项的权限增加到您的全局权限中。若结果满足该查询，则服务器执行查询。
- 3) 如果全局和数据库级权限的组合不够，服务器将保持不断地查找，首先在 tables_priv 表中，然后在 columns_priv 表中查找。
- 4) 如果在所有表检查后发现您没有许可权，服务器将拒绝执行查询。

口令是怎样存储在 user 表中的

口令在 user 表中以加密串的形式出现，因此您不知道用户的口令是什么，即使已经拥有对该表的访问权也是如此。通常，似乎认为 PASSWORD () 函数执行与 UNIX 口令所使用的类型相同的加密方法，其实并不是这样的。

当两种加密方法是单向的且不可逆时它们是相同的，但是，MySQL 不使用与 UNIX 相同的加密算法。这意味着，即使您使用 UNIX 口令作为 MySQL 的口令，也别希望加密的口令串相配。如果想对应用程序使用 UNIX 的加密方法，应使用 CRYPT() 函数而非 PASSWORD()。

在逻辑条件下，授权表由服务器按以下形式使用：

```
user OR db OR tables_priv OR columns_priv
```

您可能正在奇怪，为什么上面的描述只涉及到五个授权表中的四个。正击中要害，服务器实际检查访问许可权的方法如下：

```
user OR (db AND host) OR tables_priv OR columns_priv
```

笔者先给出了较简单的表达式，因为 host 表不受 GRANT 或 REVOKE 语句的影响。如果一直保持使用 GRANT 和 REVOKE 来管理用户权限，将不需要考虑 host 表。但下面还是给出了对该表的讨论，以便您有所了解：

- 当服务器检查数据库级的权限时，它查看客户机的 db 表的项。如果 Host 列值是空白，其含义是“检查 host 表查找哪些服务器可访问该数据库”。
- 服务器利用与来自 db 表项相同的 Db 列为项查看 host 表。如果 host 表项与客户机主机不相配，就不授予数据库级权限。如果这些项都与客户机正在连接的主机的 Host 列值相配，则 db 表的项和 host 表的项进行组合以产生客户机的数据库级权限。

但是，权限用逻辑 AND 进行组合，其意思是，除非客户机同时出现在两个项中，

否则它没有指定的权限。这样，可以在此 db 表项中授予一组基本的权限，然后用 host 表项有选择地对特定的主机禁止使用这些权限。例如，允许从域的所有主机中访问某个数据库，但是对于定位在非安全范围内的主机禁止使用数据库权限。

以上描述无疑要进行访问检查，这听起来好像是相当复杂的过程，尤其是考虑到服务器要为客户机发布的每一个单独的查询检查权限时。但是，该过程相当地快，因为服务器不对每个查询实际检查授权表的信息。相反，在服务器启动时，它将该表的内容读到内存中，然后用内存中 (in-memory) 的拷贝来检验查询。这对于访问检查操作性能产生了推进，但有相当重要的负面效应：如果您直接修改了授权表的内容，则服务器将不会注意到权限的改变。

例如，如果通过用 INSERT 语句增加一个新记录到 user 表中来增加了一个用户，在该项中指定的该用户将不能连接到该服务器。这是迷惑新管理员的问题（有时比较有经验的管理员也是如此），但是解决方案很简单：告诉服务器在您修改授权表后重新加载其内容即可。可以通过发布 FLUSH PRIVILEGES 语句或通过执行 mysqladmin flush-privileges（或 mysqladmin reload，如果是不支持 flush-privileges 的旧版本的话）来进行。

3. 作用域列匹配顺序

MySQL 服务器以一种特定的方式对授权表的项进行排序，然后试着通过按顺序查看所有的项来匹配输入的连接。所发现的第一个匹配确定被使用的项。重要的是要理解 MySQL 使用的排序顺序，尤其是对于 user 表的排序。这好像是在为难试图理解 MySQL 安全性的许多人们。

当服务器读取 user 表的内容时，它根据 Host 和 User 列的值对项进行排序。Host 值是域值（带有相同 Host 值的项被一起存放，然后按照 User 值进行排序）。但是，排序不是词典式的，或不完全是这样的。要记住的原则是直接值比模式值优先。这意味着如果正从 boa.snake.net 中进行连接并且带有 boa.snake.net 和 %.snake.net 的 Host 值，则直接值将是首选的。同样，%.snake.net 优先于 %.net，如此类推，%.net 优先于 %。IP 号也以这种方法进行匹配。对于从带有 192.168.3.14 的 IP 号的主机中连接的客户机，具有 192.168.3.4、192.168.3.%、192.168.%、192.% 和 % 的 Host 值将按此顺序匹配。

12.2.3 授权表应避免的风险

本节讨论在授权时应遵守的某些预防措施以及维护人员作出轻率选择所冒的风险。通常，给用户授超级用户权限应保守。也就是说，不要在 user 表中放置权限。取而代之的是使用其他的授权表来限制用户对特定数据库、表或列的权限。user 表中的权限可令用户影响服务器的操作或访问任何数据库中的任一表。

不要对 mysql 数据库授权。该数据库连接授权表，在该数据库上具有权限的用户可以修改这些表来获得任何其他数据库的权限。允许用户修改 mysql 数据库表的授权还能给该用户授予一个全局 GRANT 权限。如果该用户可直接修改表，就相当于可以发布任何的 GRANT 语句。

FILE 特权相当危险，不要轻易授权。以下是一个 FILE 特权用户的例子：

```
CREATE TABLE etc_passwd (pwd_entry TEXT);  
LOAD DATA INFILE "/etc/passwd" INTO TABLE etc_passwd;  
SELECT * FROM etc_passwd;
```

发布这些语句后，该用户拥有您的口令文件的内容。事实上，在服务器中，任何公用的

可读文件的内容都可以通过网络由任何有 FILE 权限的用户访问。

FILE 权限也可以用来对系统中的、不能用足够的限制文件许可权来建立的数据库进行折衷。这就是为什么应该将数据目录的内容设置成为只能由服务器可读的原因。如果与数据库表相对应的文件是全球范围内可读的，则不仅有该服务器主机账号的用户可读取它们，而且有 FILE 权限的任何客户机也都可通过网络连接并读取它们！下面的过程说明如何进行此项操作：

■ 创建有 LONGBLOB 列的表：

```
USE test
CREATE TABLE tmp (b LONGBLOB)
```

■ 用此表读取您要窃取的表的每个文件的内容，然后将该表的内容写到您自己的数据库文件中：

```
LOAD DATA INFILE "./other_db/x.frm" INTO TABLE tmp
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
SELECT * FROM tmp INTO OUTFILE "y.frm"
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
DELETE FROM tmp
LOAD DATA INFILE "./other_db/x.ISD" INTO TABLE tmp
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
SELECT * FROM tmp INTO OUTFILE "y.ISD"
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
DELETE FROM tmp
LOAD DATA INFILE "./other_db/x.ISM" INTO TABLE tmp
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
SELECT * FROM tmp INTO OUTFILE "y.ISM"
FIELDS ESCAPED BY "" LINES TERMINATED BY ""
```

■ 现在，您有了一个新表 y，它包含 other_db.x 的内容，您可以对它进行完全访问。

为了避免某人用同样的方式攻击您，应根据“内部安全性：安全数据目录访问”的指导设置数据目录的许可权。当启动服务器以限制用户利用 SHOW DATABASE 和 SHOW TABLES 对他们不能访问的数据库进行访问时，还可使用 -skip-show-database 选项。这将有助于防止用户查找不应访问的数据库和表。

ALTER 权限可能以您没有设想的任何方法被使用。假定您要 user1 能访问 table1 但不能访问 table2。但带有 ALTER 权限的用户可能会通过使用 ALTER TABLE 将 table2 重新命名为 table1 来打乱您的这一设想。

要小心使用 GRANT 权限。如果两个用户有不同的权限但都具有 GRANT 权限，则他们能使互相的访问权更强大。

12.2.4 不用 GRANT 建立用户

如果是 MySQL 3.22.11 以前的版本，则不能用 GRANT（或 REVOKE）语句建立用户和访问权限，但是可以直接修改授权表的内容。如果理解 GRANT 语句是怎样修改授权语句的，这个问题就很容易理解。可以通过手工发布 INSERT 语句自己来进行（INSERT 语句很难正确输入，但这是另外的问题。事实上这个问题就是为什么 GRANT 语句设计得如此易于使用的原因）。

当发布 GRANT 语句时，指定了用户名和主机号，可能还指定了口令。user 表项对该用

户建立，这些值记录在该项的 User、Host 和 Password 列中。需要仔细考虑的一件事是，GRANT 语句对口令进行加密，而 INSERT 则不能，它需要用 PASSWORD() 函数对 INSERT 语句中的口令进行加密。

如果已经指定了数据库级的权限，用户名和主机名将记录在 db 表项的 User 和 Host 列中。已授权的数据库记录在 Db 列中，所授予的权限记录在权限列中。

对于表级和列级权限，效果是类似的。在 tables_priv 和 columns_priv 表中创建项来记录用户名、主机名和数据库以及需要的表或者表和列。授予的权限记录在权限列中。

如果还记得上述讨论，即使不用 GRANT 语句也能够进行 GRANT 的工作。请记住，当直接修改授权表时，需要告诉服务器重新加载授权表，否则它将不会留意这些变化。可以通过执行 mysqladmin flush-privileges 或 mysqladmin reload 命令施加重新加载操作。如果忘记这步操作，服务器将不按您的设想进行工作。

以下 grant 语句创建有全部权限的超级用户，其中包括向其他用户授权的能力：

```
GRANT ALL ON *.* TO ethel@localhost IDENTIFIED BY "coffee"
WITH GRANT OPTION
```

该语句将创建 user 表的 ethel@localhost 的项，并开启所有权限，因为超级（全局）用户将存储在那里。用 INSERT 进行同样工作的语句是：

```
INSERT INTO user VALUES("localhost","ethel",PASSWORD("coffee"),
"Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y")
```

这是一个很难看的 INSERT 语句！您甚至会发现它在您的 MySQL 版本上不工作。授权表的结构可能已经改变，在 user 表中可能不会有 14 个权限列。此时，应使用 SHOW COLUMNS 来查找每个授权表包含些什么权限列，然后调整 INSERT 语句。

下列 GRANT 语句还创建具有超级用户状态的用户，但是只对单个的权限：

```
GRANT RELOAD ON *.* TO flush@localhost IDENTIFIED BY "flushpass"
```

您可能还记得我们曾在第 11 章建立 flush 用户时使用过该语句。本例中的 INSERT 语句比前一个更简单，可以很容易地列出列名并指定唯一的一个权限列。所有其他的列都将设置为缺省值“N”：

```
INSERT INTO user (Host,User,Password,Reload_priv)
VALUES("localhost","flush",PASSWORD("flushpass"),"Y")
```

数据库级的权限是用 ON db_name.* 子句而不是用 ON *.* 授予的：

```
GRANT ALL ON samp_db.* TO boris@localhost IDENTIFIED BY "ruby"
```

这些权限都不是全局的，因此不能存储在 user 表中。我们仍需要在 user 表中创建一个项（使得用户可以连接），但我们还需要创建一个 db 项来记录数据库级的权限：

```
INSERT INTO user (Host,User,Password)
VALUES("localhost","boris",PASSWORD("ruby"))
INSERT INTO db VALUES("localhost","samp_db","boris",
"Y","Y","Y","Y","Y","Y","N","Y","Y","Y")
```

“N”列是针对 GRANT 权限的，对于在末尾带 WITH GRANT OPTION 的数据库级的 GRANT 语句，应该将该列设置为“Y”。

为了设置表级或列级权限，对 tables_priv 或 columns_priv 表使用 INSERT 语句。当然，如果没有 GRANT 语句，也将没有这些表，因为它们都是同一时间出现在 MySQL 版本中的。如果有这些表但由于某种原因想手工操纵它们，那么您不能使用单个的列使权限有效。

您或者设置 `tables_priv.Table_priv` 或者设置 `columns_priv.Column_priv` 为一个 SET 值, 该 SET 值由您允许的权限组成。例如, 为了允许对某个表的 SELECT 和 INSERT 权限, 应在相对的 `tables_priv` 项中设置 `Table_priv` 列为 “Select, Insert” 值。

如果要想修改某个用户的权限 (该用户的 MySQL 账号已经存在), 应使用 UPDATE 而非 INSERT。无论是增加还是取消权限都是这样。为了删除一个完整的用户, 使用 DELETE 从该用户所在的每个授权表中删除其项。

如果要想避免发布查询来直接修改授权表, 应看一看与 MySQL 分发包一起的 `mysqlaccess` 和 `mysql_setpermissions` 脚本。

一个权限难题, 第二部分

在第11章的“一个权限难题, 第一部分”中, 我们看到了一种授权情况, 在这种情况下权限没有预想的效果。我们来简单地叙述一下问题, 经常与新的 MySQL 安装一起出现的一个问题是管理员想要为用户增加一项, 使用户们可从几个主机中进行连接。这样做最直接的方法是使用包含 ‘%’ 通配符的主机名说明符, 因此管理员用类似以下的语句创建一个用户:

```
GRANT ALL ON samp_db.* TO fred@%.snake.net IDENTIFIED BY "cocoa"
```

fred 碰巧有一个服务器主机的账号, 因此他试图从这里进行连接:

```
% mysql -u fred -pcocoa samp_db
ERROR 1045: Access denied for user: 'fred@localhost'
(Using password: YES)
```

为什么会这样? 为了继续理解, 您必须同时考虑 `mysql_install_db` 是怎样建立初始授权表和服务器怎样使用 `user` 表的项匹配客户机连接的。当通过运行 `mysql_install_db` 对数据库初始化时, 它创建带有如下 Host 和 User 值的 `user` 表的项:

Host	User
localhost	root
pit-viper.snake.net	root
localhost	
pit-viper.snake.net	

前两项通过指定 localhost 或主机名允许 root 连接到本地主机。后两项允许来自本地主机的匿名连接。当为 fred 增加此项时, 则 `user` 表将包含这些项:

Host	User
localhost	root
pit-viper.snake.net	root
localhost	
pit-viper.snake.net	
%.snake.net	fred

当服务器启动时, 它读取这些项并存储它们 (先主机, 然后是主机的用户), 其顺序是比较详细的值在前, 不详细的在后:

Host	User
------	------

(续)

localhost	root
localhost	
pit-viper.snake.net	root
pit-viper.snake.net	
%.snake.net	fred

带 localhost 的两项一起存储，有 root 的那个项放在前，因为这项比空白的更具体、更详细。带有 pit-viper.snake.net 的项以类似的方法一起存储。这些项都是不带任何通配符的直接 Host 值，因此它们都优先于 fred 的项进行存储，而 fred 项使用了一个通配符。特别地，匿名用户项在存储顺序上优先于 fred 的项。

此结果是，当 fred 试图从本地主机上进行连接时，带有空白用户名的项匹配先于在 Host 列中包含 %.snake.net 的项。该项的口令是空白的，因为缺省的匿名用户没有口令。由于 fred 在连接时指定了口令，则产生一个错误匹配和错误连接。

这里要记住的事情是，尽管使用通配符指定用户可连接的主机这种方式非常方便，但只要您在 user 表中保留匿名用户，就可能会出现从本地主机中连接的问题。

通常，笔者建议删除匿名用户项。这将使您的工作更轻松一些：

```
mysql> DELETE FROM user WHERE User = "";
```

如果想更彻底的话，可将其他授权表的匿名用户项也删除。有 User 列的表是 db、tables_priv 和 columns_priv。

本节给出的难题是一种特殊的情况，但包含了更为普遍的内容。如果某个用户的权限不按所希望的方式工作，则检查授权表，看看是否有某些项包含了比所讨论的该用户的项更详细的 Host 值，以及试图通过那个用户匹配连接的 Host 值。如果是这样的，问题的原因可能就是它。您可能需要使该用户的项更详细（或增加另一个项来包括更详细的情况）。

第13章 数据库维护和修复

理想的情况是，MySQL 从首次安装以来始终平稳地运行。但有时确实会由于各种原因而出现问题，其范围可以从电源断电到硬件故障到不正常地关闭 MySQL 服务器（如用 `kill -9` 终止服务器或机器崩溃）。诸如这样的情况大部分都超出您的控制范围，它们会导致数据库表的毁坏，尤其是在对表进行修改且未完全写入时所引起的。

本章的重点是检测 and 解决表的问题，而不论问题是如何引起的。对于表的检查和修复，MySQL 管理员最好的朋友是 `myisamchk` 和 `isamchk` 实用程序。这两个程序有好几个功能，我们已经在第4章讨论了怎样使用它们执行索引键的分布分析和索引的释放与激活。还可以使用它们检查表和修复有问题的表。这使您能在表变坏之前（使表不能使用之前）修正故障。

`myisamchk` 和 `isamchk` 提供的全部选项的清单在附录 E 中。有关其他的背景，请参阅 MySQL 参考指南的“维护 MySQL 安装”一章。

13.1 检查和维护数据库表

表的故障检测和修正的一般过程如下：

- 1) 检查出错的表。如果该表检查通过，则完成任务，否则必须修复它。
- 2) 在开始修复之前对表文件进行拷贝，以防万一。
- 3) 试着修复表。
- 4) 如果修复操作失败，从数据库备份和更新日志中恢复此表。

上述过程的最后一步假定您已经执行了数据库备份并允许更新日志有效。如果不是这样的话，系统将有危险。参考第 11 章查找一下怎样使用 `mysqldump` 和怎样开启更新日志。您肯定不想不可挽回地丢失一个表，因此，应努力地做备份。

在使用 `myisamchk` 或 `isamchk` 检查或修复表之前，应该满足一些初步需求：

- 建立常规的数据库备份过程并允许更新日志，以防事情越来越糟使表的毁坏不能修复。笔者好像在以前提醒过这一点？
- 在开始试验之前应先仔细地阅读本章的内容。尤其是不应该在阅读“避免与 MySQL 服务器交互作用”之前进行操作，因为它将讨论当您试图在一个表上执行检查或修复过程时服务器正在使用这个表所引起的问题。它还讨论怎样在服务器运行时防止那些问题发生。
- 当运行表检查或修复时，您应该被注册在运行 `mysql` 的账号下，因为您需要对表文件读写访问。

13.1.1 `myisamchk` 和 `isamchk` 的调用语法

MySQL 的 `myisamchk` 和 `isamchk` 实用程序很类似，多数时候它们可以用同样的方式使用。它们之间的主要区别是它们所使用的表的类型。对于 MyISAM 表，使用 `myisamchk`，而对于 ISAM 表，则使用 `isamchk`。您可以通过表的索引文件的扩展名来告诉表使用哪种存储

格式。扩展名“.MYI”表明是一个 MyISAM 表，而“.ISM”表明是 ISAM 表。

为了使用任一个实用程序，应指明您所检查或修复的表，以及指明要执行的操作类型的选项：

```
% myisamchk options tbl_name ...  
% isamchk options tbl_name ...
```

`tbl_name` 参数可以是表名也可以是该表的索引文件名。如果指定多个表，可以很容易地使用文件名模式来拾取目录中所有相应的文件：

```
% myisamchk options *.MYI  
% isamchk options *.ISM
```

不会因为告诉了错误的程序来检查某个表而使该表毁坏，但是除了发布一条警告消息外此程序不做任何事情。例如，下面的第一条语句将检查当前目录中的所有 MyISAM 表，而第二条语句只显示一条警告消息：

```
% myisamchk *.MYI          正确  
% myisamchk *.ISM          不正确——文件类型错
```

不论是 `myisamchk` 还是 `isamchk` 都不对表所在的位置做任何判断，因此，应该或者在包含表文件的目录中运行程序，或者指定表的路径名。这允许您将表文件拷贝到另一个目录中并用该拷贝进行操作。

13.1.2 检查表

`myisamchk` 和 `isamchk` 提供了表检查方法，这些方法在彻底检查表的程度方面有差异。通常用标准方法就足够了。如果标准检查报告没有发现错误而您仍然怀疑有毁坏（或许因为查询没有正常地工作），可能要执行更彻底的检查。要想用任意一个实用程序执行标准的表检查，则不用带任何选项直接调用即可：

```
% myisamchk tbl_name  
% isamchk tbl_name
```

为了执行扩充检查，使用 `--extend-check` 选项。该选项非常慢，但检查极为彻底。对于该表的数据文件中的每个记录，索引文件中的每个索引的相关键都被检查以确保它真正指向正确的记录。`myisamchk` 还有一个中间选项 `--medium-check`，它不如扩展检查彻底，但速度快。

如果对于 `--extend-check` 检查不报告错误，则可以肯定表是好的。如果您仍然感觉表有问题，那原因肯定在其他地方。应重新检查任何好像有问题的查询以验证查询是正确书写的。如果您认为问题可能是 MySQL 服务器的原因，应考虑整理一份故障报告或升级到新的版本上。

如果 `myisamchk` 或 `isamchk` 报告表有错误，应用下节中的说明修复它们。

13.1.3 修复表

表的修复是一项可怕的工作，如果具体问题非常独特则更难进行。然而，有一些常规的指导思想的过程，可以遵循它们来增加修正表的机会。通常，开始时可以用最快的修复方法，看看是否能修正故障。如果发现不行的话，可以逐步升级到更彻底的（但更慢的）修复方法上，直到故障被修复或您不能继续升级为止（实际上，大多数问题不用更大规模的和更慢的方法就能修正）。如果表不能修复，则从备份中恢复该表。有关使用备份文件和更新日志进行

恢复的指导在已第11章中给出。

1. 执行标准的表修复

为了修复一个表，执行下列步骤：

1) 试着用 `--recover` 选项修正表，但也可以用 `--quick` 选项试图只根据索引文件的内容进行恢复。这样将不触及数据文件：

```
% myisamchk --recover --quick tbl_name
% isamchk --recover --quick tbl_name
```

2) 如果问题仍存在，再试一下上一步的命令，但忽略 `--quick` 选项，以允许 `myisamchk` 或 `isamchk` 前进并修改数据文件：

```
% myisamchk --recover tbl_name
% isamchk --recover tbl_name
```

3) 如果还不工作，试一试 `--safe-recover` 修复方法。这种方法比普通的恢复方法要慢，但能够修正 `-recover` 方法不能修正的几个问题：

```
% myisamchk --safe-recover tbl_name
% isamchk --safe-recover tbl_name
```

如果 `myisamchk` 或 `isamchk` 由于一个 “Can 't create new temp file: file_name” 的错误消息在任何一步中停止，应该重复这个命令并增加 `--force` 选项以迫使清除临时文件。这个临时文件可能是从上一次失败的修复中留下的。

在修复表之前拷贝它们

在执行表修复前应该遵循的一个常规的预防措施是做该表的新拷贝。这种情况未必出现，但如果发生，则可以从拷贝文件中做该表的新的拷贝并试试另一种恢复方法。

2. 标准表修复方法失败时怎么办

如果标准的修复过程未能修复表，则索引文件可能在修复时丢失或毁坏。尽管未必可能，但还是有可能使表的描述文件丢失。不论哪种情况，都需要替换受影响的文件，然后再试试标准修复过程。

为了重新生成索引文件，可以使用下列过程：

1) 定位到包含崩溃表的数据库目录中。

2) 将该表的数据文件移到安全的地方。

3) 调用 `mysql` 并通过执行下列语句重新创建新的空表，该语句使用表的描述文件 `tbl_name.frm` 重新开始生成新的数据和索引文件：

```
mysql> DELETE FROM tbl_name;
```

4) 退出 `mysql`，将原始的数据文件移回到数据库目录中，替换刚建立的新的空文件。

5) 再试试标准表修复方法。

为了恢复该表的描述文件，可先从备份文件中恢复，然后再试着用标准修复方法。如果由于某些原因没有备份，但知道建立表的 `CREATE TABLE` 语句，则仍可以恢复该文件：

1) 定位到包含崩溃表的数据库目录中。

2) 将该表的数据文件移动到安全的地方。如果想要使用索引的话，还需将索引文件移走。

3) 调用 `mysql` 并发布 `CREATE TABLE` 语句建立该表。

4) 退出 `mysql`，将原始数据文件移回数据库目录中，替换刚才新建的数据文件。如果在步骤2移动了索引文件，则也要将其移回数据库目录中。

5) 再试试标准表修复方法。

13.1.4 避免与 MySQL 服务器交互作用

当您正在运行表的检查/修复实用程序时，您或许不想让 MySQL 服务器和实用程序同时访问一个表。如果两个程序都向表中写数据显然是一件坏事，但是，当一个程序在写入时另一个程序在读取也不是件好事。如果表正由一个程序写入，同时进行读取的另一个程序会被搞乱。

如果您关闭服务器，就可以保证在服务器和 `myisamchk` 或 `isamchk` 之间没有交互作用。但是管理员极不愿意使服务器完全地脱机，因为这使得没有故障的数据库和表也不可用。本节中讨论的过程将帮助您避免服务器和 `myisamchk` 或 `isamchk` 之间的交互作用。

服务器有两种类型的锁定方法。它使用内部锁定避免客户机的请求相互干扰——例如，避免客户机的 `SELECT` 查询被另一个客户机的 `UPDATE` 查询所干扰。服务器还使用外部锁定（文件级锁）来防止其他程序在服务器使用表时修改该表的文件。通常，在表的检查操作中服务器将外部锁定与 `myisamchk` 或 `isamchk` 组合使用。但是，外部锁定在某些系统中是禁用的，因为它不能可靠地进行工作。对运行 `myisamchk` 和 `isamchk` 所选择的过程取决于服务器是否能使用外部锁定。如果不使用，则必须使用内部锁定协议。

如果服务器用 `--skip-locking` 选项运行，则外部锁定禁用。该选项在某些系统中是缺省的，如Linux。可以通过运行 `mysqladmin variables` 命令确定服务器是否能够使用外部锁定。检查 `skip_locking` 变量的值并按以下方法进行：

- 如果 `skip_locking` 为 `off`，则外部锁定有效。您可以继续并运行任一个实用程序来检查表。服务器和实用程序将合作对表进行访问。但是，在运行任何一个实用程序之前，应该用 `mysqladmin flush-tables` 刷新表的高速缓存。为了修复表，应该使用表的修复锁定协议。
- 如果 `skip_locking` 为 `on`，则禁用外部锁定，但在 `myisamchk` 或 `isamchk` 检查或修复一个表时服务器并不知道，最好关闭服务器。如果坚持使服务器保持开启状态，需要确保在您使用此表时没有客户机来访问它。必须使用恰当的锁定协议告诉服务器使该表独处，并阻塞客户机对其访问。

这里所描述的锁定协议使用服务器的内部锁定机制，以防止服务器在您利用 `myisamchk` 或 `isamchk` 工作时访问表。通常的办法是调用 `mysql` 并对要检查或修复的表发布 `LOCK TABLE` 语句。然后，在 `mysql` 空闲时（即运行，但除了保持该表锁定外不用它做任何事情），运行 `myisamchk` 或 `isamchk`。在 `myisamchk` 或 `isamchk` 结束后，可以切换到 `mysql` 会话中并释放该锁以告诉服务器程序执行完毕此表可以再次使用了。

检查和修复的锁定协议有点区别。对于检查，您只需要获得读锁。在这种情况下，只能读取表，但不能修改它，因此它也允许其他客户机读取它。读锁足以防止其他客户机修改表。对于修复，您必须获得写锁以防止任何客户机在您对表进行操作时修改它。

锁定协议使用 `LOCK TABLE` 和 `UNLOCK TABLE` 语句获得并释放锁。协议还使用 `FLUSH TABLES` 告诉服务器刷新磁盘中任何未决的改变，并在通过表修复实用程序修改表后重新打开该表。您必须从单个 `mysql` 会话中执行所有 `LOCK`、`FLUSH` 和 `UNLOCK` 语句。如果锁定一个表然后退出 `mysql`，则该锁将释放，且运行 `myisamchk` 或 `isamchk` 将不再是安全的！

如果保持打开两个窗口的状态，且一个运行 mysql，而另一个运行 myisamchk 或 isamchk，则运行锁定过程将会变得很容易。这样允许您很容易地在程序之间进行切换。如果不是运行在视窗环境中，当运行 myisamchk 或 isamchk 时，将需要使用外壳程序的作业控制工具暂停和恢复 mysql。下面的指导显示出对 myisamchk 或 isamchk 的命令，可用与您正在使用的表相对应的那个命令。

1. 对检查操作锁定表

此过程只针对表的检查，不针对表的修复。在窗口 1 中，调用 mysql 并发布下列语句：

```
% mysql db_name
mysql> LOCK TABLE tbl_name READ;
mysql> FLUSH TABLES;
```

该锁防止其他客户机在检查时写入该表和修改该表。FLUSH 语句导致服务器关闭表的文件，它将刷新仍然在高速缓存中的任何未写入的改变。

当 mysql 空闲时，切换到窗口 2 并检查该表：

```
% myisamchk tbl_name
% isamchk tbl_name
```

当 myisamchk 或 isamchk 结束时，切换回到窗口 1 的 mysql 会话并释放该表锁：

```
mysql> UNLOCK TABLE;
```

如果 myisamchk 或 isamchk 指出发现该表的问题，将需要执行表的修复。

2. 对修复操作锁定表

修复表的锁定过程类似于检查表的过程，但有两个区别。第一，您必须得到写锁而非读锁。由于您将要修改表，因此根本不允許客户机对其进行访问。第二，必须在执行修复之后发布 FLUSH TABLE 语句，因为 myisamchk 和 isamchk 建立了新的索引文件，除非再次刷新该表的高速缓存否则服务器将不会注意到它：

```
% mysql db_name
mysql> LOCK TABLE tbl_name WRITE;
mysql> FLUSH TABLES;
```

利用 mysql 的空闲切换到窗口 2，做该表的数据库文件的拷贝，然后运行 myisamchk 或 isamchk：

```
% cp tbl_name.* /some/other/directory
% myisamchk --recover tbl_name
% isamchk --recover tbl_name
```

--recover 选项只是针对安装而设置的。这些特殊选项的选择将取决于您执行修复的类型。

myisamchk 或 isamchk 运行完成后，切换回到窗口 1 的 mysql 会话，再次刷新该表的高速缓存并释放表锁：

```
mysql> FLUSH TABLES;
mysql> UNLOCK TABLE;
```

13.1.5 快速运行 myisamchk 和 isamchk

myisamchk 和 isamchk 的运行可能会花很长时间，尤其是您正在处理一个大表或使用一个更广泛的检查或修复方法时。通过告诉这些程序在运行时使用更多的内存，能够提高它们的速度。这两个实用程序都有几个可设置的操作参数。其中最重要的是控制程序使用的缓冲区大小的变量：

变 量	含 义
key_buffer_size	用于存放索引块的缓冲区大小
read_buffer_size	读操作的缓冲区大小
sort_buffer_size	排序用的缓冲区大小
write_buffer_size	写操作的缓冲区大小

要想查看任一程序使用的这些变量的缺省值，可用 `--help` 选项运行该程序。要想指定其他的值，可在该命令上使用 `--set-variable variable=value` 或 `-O variable=value`。您可以将变量的名字简化成 `key`、`read`、`sort` 和 `write`。例如，可告诉 `myisamchk` 使用 16MB 的排序缓冲区和 1MB 的读写缓冲区，其调用如下：

```
% myisamchk -O sort=16M -O read=1M -O write=1M ...
```

`sort_buffer_size` 只能利用 `--recover` 选项来使用（而不是利用 `--safe_recover`），在这种情况下，`key_buffer` 不能使用。

减少服务器的停机时间

防止服务器访问（您正在处理的）表的另一种方法是在数据目录的外面使用该表文件的拷贝。这样并不能消除交互作用的问题，因为仍然必须防止服务器访问（并可能修改）正在进行拷贝的表。但是，如果您不愿意使服务器脱机的话，该路线可能是使服务器停机时间最小化的一种方法，这对您是有吸引力的。在将该表的文件拷贝到另一个目录时关闭服务器，然后恢复服务器。

myisamchk 的未来打算

`myisamchk` 的表检查和修复功能打算在 MySQL 3.23 版本系列的某个时候被合并到服务器中。如果这种打算实现，对表的检查和修复将更容易，因为服务器与 `myisamchk` 的交互问题将不再会出现。

同样，您能够告诉服务器在启动时检查表，因此在启动服务器前将不需要设置任何特殊的命令在引导期间执行。该程序不对 ISAM 表进行操作，因此在服务器获得表的检查和修复能力时，应考虑将 ISAM 表转换成 MyISAM 表。请查看新发行版的 MySQL 参考指南，了解在此范围内有什么新进展。可以用 `ALTER TABLE` 语句转换表的类型：

```
ALTER TABLE tbl_name TYPE = MYISAM
```

13.2 安排预防性的维护

应该考虑建立一个预防性维护的时间表，以协助自动检测问题，使得您可以采取措施进行修正：

- 1) 执行常规的数据库备份并允许更新日志。
- 2) 安排定期的常规表检查。通过检查表，将减少使用备份的机会。这个工作使用 `cron` 作业（一般从运行服务器所使用的该账号的 `crontab` 文件中调用）并且很容易实现。例如，如果您作为 `mysqladm` 用户运行服务器，则可以从 `mysqladm` 的 `crontab` 文件中建立定期检查。如果您不知道如何使用 `cron`，应使用下列命令查看相关的 UNIX 人工页：

```
% man cron
% man crontab
```

3) 在服务器启动前的系统引导期间检查数据库表。机器可能会因早期的崩溃重新启动。如果是这样的话，数据库表可能已被毁坏，应该对它进行检查。

为了运行自动的表检查，可以编写一个脚本，将目录改变为服务器数据目录并对所有数据库表运行 `myisamchk` 和 `isamchk`。我们将在下面讨论的脚本中同时使用这两个程序。如果您只有 MyISAM 表或只有 ISAM 表，则只需其中一个程序，可以将无关的那个程序从脚本中去除。

`myisamchk` 和 `isamchk` 都根据表检查的方式产生某些输出结果以便了解正在检查哪些表，甚至在没有问题时也是如此。对于 `crontab` 的项，除非表中有错误，否则通常将禁止输出结果（如果作业产生任何输出，`cron` 作业通常生成一个邮件消息，很少会收到没有任何问题的表检查作业的邮件）。如果用 `--silent` 选项调用任一个实用程序，它仅当发现问题时才产生输出。另外，`myisamchk` 支持 `--fast` 选项，该选项允许程序跳过自上次检查以来没有被修改过的任何表。

在服务器数据目录中检查所有表的一个简单的脚本如下（`DATADIR` 应该修改成对应您系统的值）：

```
#!/bin/sh
cd DATADIR
myisamchk --silent --fast */*.MYI
isamchk --silent */*.ISM
```

实用此脚本的一个潜在的问题是：如果有许多表，通配符模式 ‘`*/*.MYI`’ 或 ‘`*/*.ISM`’ 可能会由于 “too many arguments (过多的参数)” 错误使外壳程序爆炸。另一个可选择的脚本如下（同样，将 `DATADIR` 改变为对应您系统的值）：

```
#!/bin/sh
datadir=DATADIR
find $datadir -name "*.MYI" -print | xargs myisamchk --silent --fast
find $datadir -name "*.ISM" -print | xargs isamchk --silent
```

不论您选择哪种形式的脚本，笔者都假定您调用的是 `check_mysql_tables`，应确保及时改变该方式使它可执行，然后您手工调用它以验证它工作是否正常：

```
% chmod +x check_mysql_tables
% check_mysql_tables
```

在理想情况下应该没有输出结果。如果系统不支持外部锁定，有可能服务器将在您检查表时改变它。此时，脚本可能会把实际没问题的表报告成有问题的。这有点不幸，但比出现相反的问题要好：当出现某些故障时脚本报告无问题。如果系统支持外部锁定，则该问题就不会出现。

以下部分将说明如何建立脚本，使它通过 `cron` 并在系统启动期间自动执行。在这些小节的例子中，笔者假定脚本安装在 `/usr/users/mysaladm/bin` 中。您将需要调整适合自己系统的值。

如果在您的系统上运行了多个服务器，将需要修改该过程来检查每个服务器数据目录中的表。您可以使用不同的 `check_mysql_tables` 拷贝来进行，或通过修改它来接收一个命令行参数进行，该参数指定了想要检查的数据目录。

13.2.1 用 `cron` 定期检查表

假定要想对 `mysqldm` 用户从 `crontab` 文件中调用脚本 `check_mysql_tables`。先以该用户的身份进行注册，然后用下列命令编辑 `crontab` 文件：

```
% crontab -e
```

该命令带您进入带有当前 crontab 文件拷贝的编辑器中（如果以前没有编辑过，此文件可能是空的）。增加一行到文件中：

```
0 3 * * 0 /usr/users/mysqladm/bin/check_mysql_tables
```

它告诉 cron 在每个星期日的上午 3 时运行此选项。可以按要求改变时间或安排。有关这些选项的格式，请参阅 crontab 人工页。

13.2.2 在系统启动期间检查表

如果您正在使用 BSD 风格的系统，并且已经将服务器的启动命令增加到 /etc/rc.local 或类似的其他文件中，则可以在启动服务器前从对应的文件中调用 check_mysql_tables。

如果正在使用 System V 风格的启动方法从 /etc/rc.d 目录之一中调用 mysql.server，则过程稍有点复杂。这些目录中的启动脚本应该理解 start 和 stop 参数的含义，以便对系统的启动和关闭采取相应的操作。为了执行表的检查，我们可以编写这样的脚本：当参数为 start 时调用 check_mysql_tables，而当该参数为 stop 时不做任何事情。让我们来调用这样的一个脚本 mysql.check，其内容如下：

```
#!/bin/sh

# See how we were called.
case "$1" in
  start)
    echo -n "Checking MySQL tables: "
    if [ -x /usr/users/mysqladm/bin/check_mysql_tables ]; then
      /usr/users/mysqladm/bin/check_mysql_tables
    fi
    ;;
  stop)
    # don't do anything
    ;;
  *)
    echo "Usage: $0 {start|stop}"
    exit 1
esac

exit 0
```

现在，您已经可以使用一个过程来安装 mysql.check 了，该过程类似于在第 11 章所介绍的安装 mysql.server 的过程。必须给 mysql.check 一个运行级目录中较低的前缀号，才能使得它在 mysql.server 前运行。例如，如果在运行级目录中以 S99mysql.server 链接到 mysql.server，则应该以 S98mysql.check 链接到 mysql.check。

第四部分 附录

附录A 获得和安装软件

此附录介绍怎样获得和安装运行 MySQL 所需的软件和本书中介绍的第三方软件，如 Perl DBI 和 CGI.pm 支持环境、PHP、Apache 等。此附录中还说明了怎样获得与本书所用的 samp_db 数据库样例有关的文件。

本章的目的是将所有程序包的安装说明汇集在一起，但不取代各程序包附带的安装说明。事实上，我们鼓励阅读随程序包附带的说明。此附录提供了能满足多数情况的说明，但每个软件分发还含有在独立的安装过程失败时，怎样排除故障的说明。例如，MySQL 手册中有相当多的章节介绍其安装过程，还包括专门针对许多系统问题的解决方法。

A.1 选择安装的内容

如果还没有安装 MySQL，那么必须安装它，但还需要安装如下三个工具：

- 如果希望编写访问 MySQL 数据库的脚本，则必须安装 MySQL 客户机编程支持环境和 DBI 模块。如果打算编写提供对数据库进行 Web 访问的 DBI 脚本，还应该安装 CGI.pm 模块，并且需要一个 Web 服务器。本书所使用的是 Apache 服务器，但也可以使用其他一些服务器。
- 如果想要编写 PHP 脚本，则必须安装 MySQL 客户机编程支持环境与 PHP。一般 PHP 用于 Web 脚本编写，这表示还需要一个 Web 服务器。本书使用 Apache 服务器，因为 Apache 服务器具有 PHP 的模块支持环境。这表示可以安装 PHP 作为 Apache 系统的一个内部成分。如果您不能修改自己的 Apache 拷贝（例如，如果在 ISP 主机上工作而 Apache 是由 ISP 控制的），或者运行的是另一个不同的 Web 服务器，那么可以将 PHP 设置为一个独立程序，并且将 PHP 脚本作为一个标准的 CGI 脚本运行。将 PHP 作为 Apache 的一个模块安装所提供的性能比将它作为独立程序运行的性能更好。

许多安装程序包都可得到预编译程序。Linux 提供各种 RPM 文件。如果喜欢编译源程序，或者程序分包在您的平台上不可用，那么需要一个 C 编译器（MySQL 需要 C++）。MySQL 开发者建议使用 gcc 2.95.1 或更新的版本来建立 MySQL。egcs 在 1.03 以上的版本可以用。

所需要的其他工具还包括 gunzip 和 tar（如果利用压缩的 tar 文件进行安装）或 rpm（如果利用 RPM 文件进行安装）。

A.2 获得软件

每个程序包的主要分布点如下表所示。有的程序包可从多个位置得到：

程序包	位置
MySQL	http://www.mysql.com/ ftp://ftp.mysql.com/pub/mysql/
DBI	http://www.symbolstone.org/technology/perl/DBI/ http://www.mysql.com/Contrib/
PHP	http://www.php.net/ ftp://ftp.php.net/
Apache	http://www.apache.org/ ftp://ftp.apache.org/
CGI.pm	http://stein.cshl.org/WWW/software/CGI/

如果使用 FTP 传送文件，应该用二进制方式传送它们以免出错。

有许多 MySQL 的镜像站点。www.mysql.com 位于 Scandinavia，因此如果您选择一个离您最近的站点，可以缩短下载时间。下列地址列出了各镜像站点：

<http://www.mysql.com/mirrors.html>

对于 Perl 软件，可以用 <http://cpan.perl.org/> 找到所需模块的镜像。

选择要安装的版本

安装哪个程序包版本应该根据需求而定：

- 如果需要极大的稳定性，应该守旧一些，使用软件包的最新稳定（非 β）版。这样既能得到较新的功能，而又不像 β 版那样存在许多问题。
- 如果您喜欢冒险，或者是一个开发者，可能希望使用最新的 β 版。当然，如果极为需要的某些功能只能在 β 版中得到，那么除了使用该版本外没有其他选择。
- 对于 MySQL，预编译的系统和 RPM 分发版一般是优化建立的，它们比使用源程序分发版中的配置脚本自己完成更好。MySQL 开发者还利用某些使 MySQL 更快的商业优化编译器来建立分发版。因此这些分发版中的程序可能比您自己编译的程序更快。如果您想得到最好的性能，应该尽量使用预编译的分发版。

每个软件包的 Web 站点都指出哪些版本是最新的稳定发行版，哪些版本是最新的 β 发行版。它们还提供对以前版本的功能更改清单，以帮助决定哪个版本最适合于您。

如果您使用的是二进制的代码或 RPM 分发版，解开分发版就等于安装它，因为解开的文件会进入相应的目录。如果在分发版将文件装入保护目录，可能需要以 root 用户的身份打开分发版。

对于源程序分发版，可将其在用来进行编译的地方解开，然后再将软件安装到目的地址。可能需要以 root 身份完成安装步骤（但不是完成配置或编译步骤）。

订阅帮助邮件清单

在安装程序包时，订阅该程序包的综合讨论表是一个很好的想法，因为可以通过它提问题并得到答案。如果安装的是 β 发行版，应该准备参加并阅读与该软件有关的邮件，以便随时得到错误报告和解决方法。如果您不参加综合讨论，至少也应该订阅其通告，以便能接到新发行版的通知。预订邮件清单和使用它们的说明在引言中做过介绍。每个程序包的 Web 站点也提供预订信息。

如果利用源程序进行安装，有几个程序包配置有 configure 实用程序，这样容易在各种系统上设置和建立软件。如果建立失败，可能需要针对您原来给出的选项给 configure 提供不同的选项。这很有必要，因为要防止 configure 使用您前次运行它时保存的那些信息。可按如下方式清除原来保存的配置信息：

```
% make distclean
```

也可以按如下方式清除：

```
% rm config.cache  
% make clean
```

不管选用哪种方式，现在都可以重新运行 `configure` 了。

A.3 在 UNIX 上安装 MySQL

可得到几种版本的 MySQL 分发包。当前稳定的发行版有 3.22 版本系列。当前正在开发的版本为 3.23 系列。一般，应该使用系列中最高编号的版本。

MySQL 分发包可以以二进制代码、RPM 和源代码的格式得到。二进制代码和 RPM 分发包容易安装，但必须接受建立在分发包内的安装设计和缺省配置。源代码分发包安装很困难，因为必须对软件进行编译，但可对参数进行更多的控制。例如，可以只编译客户机程序的分发包，而不用管服务器，可以更改安装软件的目标位置等。

分发包包含有下列一个或多个组件：

- `mysqld` 服务器
- 客户机程序（`mysql`、`mysqladmin` 等）以及客户机编程支持环境（库和头文件）
- 文档
- 标准数据库
- 语言支持环境

源程序和二进制代码分发包包含有上述所有内容。每个 RPM 文件只包含其中一些内容，因此为了得到所需的东西，可能需要安装多个 RPM。

如果打算连接到其他机器上运行的服务器，则不需要安装服务器，但应该安装客户机软件：

- 如果不运行服务器，那么只需要客户机，以便能够连接到其他机器的服务器上。
- 如果您确实运行一个服务器，将希望能够从该服务器的主机对此服务器进行连接，而不是在其他具有客户机软件的机器上登录，然后再来测试您的服务器。

A.3.1 MySQL 安装综述

在 UNIX 上安装 MySQL 涉及下列步骤：

- 1) 创建一个用户的 UNIX 账号和服务器的组（如果正在安装服务器）。
- 2) 获得和打开想安装的分发包。如果使用的是源代码分发包，编译并安装它。
- 3) 运行 `mysql_install_db` 脚本来初始化数据目录和权限表（仅允许第一次安装）。
- 4) 启动服务器。
- 5) 参阅第 11 章，熟悉一般的管理过程。特别应该阅读服务器设置和关闭以及作为无特权用户运行服务器的段落。

A.3.2 创建 MySQL 用户账号

如果只需要运行 MySQL 客户机软件，而不打算运行 MySQL 服务器，可以跳过这一节。MySQL 服务器可作为系统上的任何 UNIX 用户运行，但出于安全和保密的原因，最好是

不以 root 身份运行服务器。建议创建一个独立的账号用于 MySQL 管理，并以该用户运行服务器。这样，可作为该用户登录，并在进行维护和故障排除的目录中具有完全的权限。创建用户账号的过程随系统的不同而不同。详细内容可参阅相应的文献。

本书将mysqladm和mysqlgrp分别用于 UNIX 用户和此账号的组名。如果打算安装 MySQL 只为自己使用，可以自己运行它，那么在本书中任何地方出现的 mysqladm 和 mysqlgrp都可换成您自己的登录名和组名。如果从 RPM 文件进行安装，随 RPM 的安装过程会自动建立一个名为 mysql 的用户。在这样的情况下，可以将 mysql 替换成 mysqladm。

使用独立、无特权帐户而不是运行 MySQL 的 root，其优点为：

- 如果不以 root 运行 MySQL，任何人都不能利用此服务器的安全漏洞获得 root 访问。
- 无特权的用户执行 MySQL 管理任务比用 root 执行更为安全。
- 服务器将创建 mysqladm 而非 root 拥有的文件。root在系统上拥有的文件越少越好。
- 将 MySQL 的活动分隔在它自己的账号中可以在概念上更为清晰，并且更容易看出系统上什么东西与 MySQL 相关。例如，在保存 core 文件的目录中，有 MySQL 用户 mysqladm 的一个独立文件。否则 MySQL 的 core 作业将位于 root 的文件中，将会和其他东西一起作为 root 周期性的执行。

A.3.3 获得和安装 MySQL 分发包

在下面的说明中，我们用版本代表 MySQL 的版本号，用平台代表在其上进行安装的平台名。它们也用在分发包文件名中，以便容易标识分发包，并将各个分发包区分开来。版本号是一些类似于 3.22.26 或 3.23.4-alpha 这样的东西，平台名是类似于 sgi-irix6.3-mips 或 dec-osf4-0b-alpha 这样的东西。

1. 安装二进制代码分发包

二进制代码分发包文件具有诸如 `mysql-version-platform.tar.gz` 这样的名字。获得理想版本和平台的分发包文件并将它放入指定的安装 MySQL 目录，如 `/usr/local`。

用下面的任意一条命令打开分发包（如果 tar 版本不支持非压缩分发包的 z 选项，应该使用第二条命令）：

```
% tar xzf mysql-version-platform.tar.gz
% gunzip < mysql-version-platform.tar.gz | tar xf -
```

打开分发包创建包含分发包内容的目录 `mysql-version-platform`。为了更容易引用这个目录，可创建一个符号连接：

```
% ln -s mysql-version-platform mysql
```

现在，如果在 `/usr/local` 目录下安装 MySQL，可引用的安装目录为 `/usr/local/mysql`。

如果只准备使用分发包所提供的客户机支持环境并且不运行服务器，则安装结束。如果是第一次安装 MySQL，可参阅 A.3.4 节“初始化数据目录和权限表”。如果是对已有安装进行更新，可参阅 A.3.5 节“启动服务器”。

2. 安装 RPM 分发包

RPM 文件可用于 Linux 系统上的 MySQL 安装。它们具有如下的文件名：

- `MySQL-version-platform.rpm` 服务器软件。
- `MySQL-client.version-platform.rpm` 客户机程序。

■ **MySQL-devel.version-platform.rpm** 编写客户机程序的开发支持环境（客户机库和头文件）。如果想使用或编写访问 MySQL 数据库的 Perl DBI 脚本，则需要用到它。

■ **MySQL-bench.version-platform.rpm** 标准和测试。需要 Perl 和 Mysql-Mysql-modules 模块。可在 www.mysql.com 上得到 Mysql-Mysql-modules 的一个 RPM 文件。

■ **MySQL-version.src.rpm** 服务器、客户机、标准和测试的源代码。

如果从 RPM 安装，不需要特定的目录，因为 RPM 文件含有指定将所包含的文件安装在何处的信息。对任何 RPM 文件 `rpm_file`，可用下列命令知道其内容将安装在何处：

```
% rpm -qpl rpm_file
```

为了安装一个 RPM 文件，使用下列命令：

```
% rpm -i rpm_file
```

MySQL 的各组成部分被划分为不同的 RPM 文件，因此可能需要安装不止一个 RPM。为了安装客户机支持环境，可使用下列命令：

```
% rpm -i MySQL-client.version-platform.rpm
```

安装服务器支持环境，使用下列命令：

```
% rpm -i MySQL-version-platform.rpm
```

如果打算使用客户机编程支持环境编写自己的程序，应该安装供开发用的 RPM 文件：

```
% rpm -i MySQL-devel.version-platform.rpm
```

如果打算使用分发包提供的客户机支持环境，而且不准使用服务器，MySQL 的安装就结束了。如果是第一次安装 MySQL，可参阅 A.3.4 节“初始化数据目录和权限表”。如果是对现有安装进行更新，可参阅 A.3.5 节“启动服务器”。

如果想从源代码 RPM 文件进行安装，用下面的命令就可以了：

```
% rpm --recompile MySQL-version.src.rpm
```

3. 安装源代码分发包

源代码分发包具有如 `mysql-version.tar.gz` 这样的名称，其中 `version` 为 MySQL 的版本号。选择想要将分发包放在其下的目录，并进入该目录。获得分发包文件并用下列命令之一打开它（如果 tar 版本不支持非压缩分发包的 `z` 选项，应该使用第二条命令）：

```
% tar xzf mysql-version.tar.gz
% gunzip < mysql-version.tar.gz | tar xf -
```

打开分发包创建包含分发包内容的目录 `mysql-version`，并进入此目录。

```
% cd mysql-version
```

在安装以前，需要配置和编译此分发包。如果各步骤失败，应该参阅 MySQL 参考指南的“安装 MySQL”一章，应该特别注意它所包含的有关各种机器专有的系统说明。

使用 `configure` 命令配置分发包：

```
% ./configure
```

可以为 `configure` 指定选项。为了得到可用的选项表，执行下列命令：

```
% ./configure --help
```

下面的清单给出了一些许多人认为很有用的配置选项：

■ **--without-server** 只建立客户机支持环境的配置（客户机程序或客户机库）。如果打算访问运行在其他机器上的服务器，应该选择此选项。

- `--prefix = path_name` 缺省时，安装根目录为 `/usr/local`。数据目录、客户机、服务器、客户机库以及头文件安装在根目录下的 `var`、`bin`、`libexec`、`lib` 和 `include` 目录中。如果想要更改安装的根目录，使用 `--prefix` 选项即可。
- `--localstatedir = path_name` 这个选项更改数据目录的位置。如果不希望将数据库保存在 `/usr/local/var` 之下，可使用这个选项。
- `--with-low-memory sql/sql_yacc.cc` 源代码文件需要大量编译内存，这有时会导致安装失败。这个问题的表现包括有关“fatal signal 11”或虚内存耗尽错误消息。`--with-low-memory` 选项用产生低内存消耗的选项来调用编译程序。

在执行了 `configure` 后，编译分发并安装它：

```
% make
% make install
```

如果未利用 `--prefix` 选项指定具有写入许可的目录，需要以 **root 身份** 执行安装命令。

如果只准备使用分发包所提供的客户机支持环境且不运行服务器，则 MySQL 的安装结束。如果是第一次安装 MySQL，可参阅 A.3.4 节。如果是对已有安装进行更新，可参阅 A.3.5 节“启动服务器”。

A.3.4 初始化数据目录和权限表

在能够利用 MySQL 安装之前，需要初始化含有控制网络对服务器进行访问的权限表的 `mysql` 数据库。这个步骤只在新安装以及打算运行服务器时需要。那些只进行客户机支持环境安装的人员可以跳过这一节。对于二进制代码分发，从安装目录（由分发包创建的 `bin` 目录的父级目录）执行命令。对于源代码分发，从分发包的高层目录执行命令。

DATADIR 是数据目录的路径名。一般情况下，作为 **root** 执行下列命令。如果作为 `mysqladm` 登录，或者由于打算自己运行 MySQL 而在自己的账号下安装它，则可**不以 root 的身份** 执行命令，而且可以跳过 `chown` 和 `chmod` 命令。

如下运行 `mysql_install_db` 脚本设置缺省权限表（如果利用 `RPM` 文件进行安装，则不需要这样做，因为 `mysql_install_db` 将会自动运行）：

```
# scripts/mysql_install_db
```

如果 `mysql_install_db` 失败，应该参阅 MySQL 参考指南的“安装 MySQL”一章，看看该章对于所遇到的问题是怎样说的，然后再试一次。注意，如果 `mysql_install_db` 未成功完成，它所建立的任意权限表可能都是不完整的。应该删除它们，因为 `mysql_install_db` 不会试图重建已经存在的表。可用如下语句删除整个 `mysql` 数据库。

```
# rm -rf DATADIR/mysql
```

在运行 `mysql_install_db` 之后，如果所安装的是 MySQL 3.22.10 以前的版本，应该关闭服务器（对于更新的版本，`mysql_install_db` 会自动完成这项工作）：

```
# bin/mysqladmin --user=root shutdown
```

在服务器关闭后，更改用户和组的拥有者该数据目录下的所有文件模式：

```
# chown -R mysqladm.mysqlgrp DATADIR
# chmod -R go-rwx DATADIR
```

`chown` 命令更改 MySQL 用户的拥有者，而 `chmod` 更改模式把除 `mysqladm` 外的所有人员都排斥在外。

A.3.5 启动服务器

这个步骤只在要运行服务器时需要。那些只需要客户机的安装可以跳过这一步。应该在前一节给出的相同目录中执行本节的命令。一般作为 root 用户运行命令。如果作为 mysqladm 用户登录,或者在自己的账号下安装 MySQL,那么可以不以 root 身份运行命令,而且应该省略 --user 选项。

用下列命令启动服务器:

```
# bin/safe_mysqld --user=mysqladm &
```

--user 选项告诉服务器作为 mysqladm 运行。如果想启用日志,使用下列命令:

```
# bin/safe_mysqld --user=mysqladm --log &
```

缺省的安装允许 MySQL root 用户不用口令进行连接。设口令是个好主意。还应该是在系统启动和关闭时,安排服务器启动和关闭。此外,推荐使用更新日志,因为这对数据恢复过程是很有用的。对于完成这些活动的说明,请参阅第 11 章。

A.3.6 安装 Perl DBI 支持环境

如果想要编写访问 MySQL 数据库的 Perl 脚本,应该安装 DBI 软件。DBI 要求已经安装过 MySQL 客户机编程支持环境,因为它要使用 MySQL C 客户机库。如果希望编写基于 Web 的 DBI 脚本,还可以安装 CGL.pm 模块。DBI 软件需要 Perl 的相当新的版本(5.004 或更新)。如果没有安装 Perl,请访问 <http://www.perl.com/>, 下载 Perl 分发版,并在安装 DBI 支持环境前安装它。

DBI 支持环境需要三个模块,可从源代码或 RPM 文件对它进行安装:

- Data-Dumper 方便地处理 Perl 数据结构的模块。

- DBI 主 DBI 驱动。

- Msql-Mysql-modules 连接到 MySQL 服务器时, DBI 使用的 MySQL 专用驱动。

如果从源代码进行安装,应该按前面清单中给出的次序安装这个模块。否则,下面安装说明中的测试步骤将会不正常。

所有三个模块的安装都是类似的。在从源代码进行安装时,利用下面的某个命令打开分发版(如果 tar 版本不支持非压缩分发版的 z 选项,应该使用第二条命令):

```
% tar xzf dist_file.tar.gz
% gunzip < dist_file.tar.gz | tar xf -
```

然后进入 tar 命令创建的分发版目录并运行下列命令(可能需要作为 root 运行这些安装步骤):

```
% perl Makefile.PL
% make
% make test
% make install
```

如果运行前面 Msql-Mysql-modules 分发版的命令,在生成 Makefile 时,perl 命令将提出下列问题:

- Which drivers do you want to install? 存在 MySQL 和 mSQL 的各种组合选择。除非还运行 mSQL,否则选择 MySQL 使其简单化。

- Do you want to install the MysqlPerl emulation? MysqlPerl 是 MySQL 的旧 Perl 接口, 已经作废。除非有旧的 MysqlPerl 脚本并希望在 DBI 模块中启用仿真支持环境, 否则回答为 no。
- Where is your MySQL installed? 这应该是包含 MySQL 头文件的祖父目录, 大概是 /usr/local 或 /usr/local/mysql, 除非将 MySQL 安装在非标准的位置才不是这样。
- Which database should I use for testing the MySQL drivers? 缺省为 test, 除非关闭了对它的匿名访问, 否则这样应该可以了。在关闭其匿名访问时, 需要给出一个您拥有访问权的数据库名, 然后为后面的问题指定一个合法的 MySQL 用户名和口令。
- On which host is the database running? 如果运行的是一个本地服务器, localhost 就可以了。如果不行, 指定一个您拥有访问权的服务器主机。在执行 make test 命令时, MySQL 服务器必须运行在这个主机上, 否则测试将失败。
- User name for connecting to the database?
Password for connecting to the database?

为测试连接到 MySQL 服务器的所用的用户名和口令。缺省设置时, 这两个问题都可以回答 undef。这样将使驱动作为匿名用户连接。如果需要非匿名连接, 可指定非空值。

如果安装 Perl 模块有问题, 可参阅相应分发包的 README 文件以及 DBI 邮件清单的邮件存档文件, 因为多数安装问题的答案都可从该处找到。

如果希望使用 CGI.PM, 它可能已经在 Perl 安装中给出了。试运行命令 perldoc CGI。如果显示 CGI 文档, 那么 CGI 已经安装。否则, 取得相应的分发包, 按与其他 Perl 模块相同的说明打开并安装。

如果为了使用基于 Web 的 DBI 脚本, 希望安装 mod_perl Apache 模块, 可访问下列位置的 Apache Web 站点的 mod_perl 区, 得到更多的信息:

<http://perl.apache.org/>

A.3.7 安装 PHP 和 Apache

按照下面的说明可以开始行动了。如果遇到问题, 可查看 PHP 分发包所含的 INSTALL 文件的“VERBOSE INSTALL”一节(阅读该文件是个不坏的主意, 它含有大量有用信息)。

这些说明都假定您有一个 Apache 源代码分发包可用, 而且将作为一个 Apache 模块运行 PHP。还假定您具有 Apache 1.3.9 和 PHP 3.0.12, 而且位于这样一个目录中, 在该目录下可定位含有源代码分发包的 apache_1.3.9 和 php-3.0.12 目录。如果您拥有不同的版本或位置, 在使用下列说明时, 应该替换成您的实际版本号 and 路径名。

要配置和建立 PHP, 可从下列命令入手(这个过程看上去有点怪, 因为要从 Apache 分发包开始, 但它是对的):

```
% cd apache_1.3.9
% ./configure
```

Apache 的 configure 命令是必须的, 因为 PHP 需要的某些头文件是在配置时生成的。如果不在 Apache 分发包中运行 configure, PHP 配置过程将不能找到文件。在配置了 Apache 后, 进入 PHP 分发包目录, 配置 PHP, 设置它, 并安装它。还可以安装 PHP 初始化文件的一个拷贝:


```
% cd ../php-3.0.12
% ./configure --with-mysql --with-apache=../apache_1.3.9 --enable-track-vars
% make
% make install
% cp php3.ini-dist /usr/local/lib/php3.ini
```

PHP 的 configure 命令告诉 PHP 您想要 MySQL 支持环境，并指出 Apache 源代码位于何处。--enable-track-vars 选项将窗口输入的自动转换转入可从 PHP 页内部容易地访问的变量内。make 命令编译和安装 PHP。有可能需要作为 root 安装初始化文件 php3.ini。

在作为一个模块建立 PHP 时，直接拷贝相应模块到 Apache 源代码树来安装它，以便可将它连接进 Apache 二进制代码内。在完成后，可按如下编译和安装 Apache：

```
% cd ../apache_1.3.9
% ./configure --activate-module=src/modules/php3/libphp3.a
% make
% make install
```

这些步骤配置 Apache，以便知道 PHP 模块文件，然后建立和安装它。应该编辑 Apache 配置文件 httpd.conf，以便 Apache 识别 PHP 脚本。PHP 识别是基于 PHP 脚本的文件扩展名进行的。例如，如果希望 Apache 识别 .php 和 .php3，在配置文件中放入下面的两行：

```
AddType application/x-httpd-php3 .php
AddType application/x-httpd-php3 .php3
```

所用的后缀依赖于打算怎样命名 PHP 脚本。.php 和 .php3 可能是最常用的。另一常用的后缀为 .phtml。如果愿意，可以将它们都启用。如果打算安装从使用这些扩展名的其他人那里得到的页面，最好都启用它们。

还可以告诉 Apache 在 URL 的结尾处未指定文件名时，将 index.php 或 index.php3 作为某个目录的缺省文件。您可能会在配置文件中发现如下的行：

```
DirectoryIndex index.html
```

将其更改如下：

```
DirectoryIndex index.html index.php index.php3
```

在编辑 Apache 配置文件后，结束 httpd 服务器（如果有一个正在运行），然后启动刚安装的新 httpd。在许多系统中，下面的命令（以 root 执行）完成此工作：

```
# /usr/local/apache/bin/apachectl stop
# /usr/local/apache/bin/apachectl start
```

还应该设置 Apache 在系统启动和关闭时启动和关闭。相应的说明请参阅 Apache 文档。一般情况下，它包括导入时运行 apachectl start，以及关闭时运行 apachectl stop。

A.4 在 Windows 上安装 MySQL

可在 Windows 95、Windows 98 或 Windows NT 下运行 MySQL。为了做到这一点，必须安装 TCP/IP 支持环境，而且 Winsock 软件必须至少为版本 2。

在 Windows 下可安装两种软件：

- 独立程序，如为 UNIX 安装的那种程序（mysqld 服务器与诸如 msyql 和 mysqladmin 这样的程序）。
- MyODBC，允许其他程序（如 Access）与 MySQL 服务器通信的 ODBC 的 MySQL 驱动。

Windows 分发包全都可从 MySQL 站点作为 zip 文件得到。为了打开这样的文件，只需双击它即可。如果这样不行，可使用诸如 Winzip 或 pkunzip 这样的程序来打开它。主要分发包如下：

- **mysqlwin.version.zip** 完全分发包（服务器和客户机）。
- **winclients-version.zip** 客户机软件（mysql、mysqladmin、mysqldump 等等）。如果不想在 Windows 下运行服务器，可使用这个软件。
- **myodbc-version-win95.zip**
myodbc-version-nt.zip
Windows 95（Windows 98）或 Windows NT 的 MyODBC 支持环境。
- **mysqlclient-version-cygwin-b20.tar.gz** 用 Cygnus 工具集编译的 MySQL 客户机。它包括 mysqlc，具有命令行历史编辑功能的一种 mysql 客户机版本。如果安装它，需从 c:\mysql\lib 拷贝库文件 cygwinb19.dll 到 Windows 系统目录。

A.4.1 安装仅有客户机的分发包或客户机/服务器分发包

如果有一个只包含客户机软件的 winclients.version.zip 分发包，解开它以产生 mysql 文件夹。将此文件夹定位到 c:\。

如果有包含服务器和客户机的 mysqlwin-version.zip 分发包，首先应该打开它。在结果文件夹中，执行 Setup 程序，在 c:\mysql 中安装 MySQL。

从下列服务器中进行选择：

服务器	说明
mysqld	标准服务器
mysqld-opt	为奔腾处理器进行优化的服务器
mysqld-nt	可作为在 Windows NT 下的服务安装的服务器

可在 NT 下运行任意的服务器，但其中最后一个只能在 NT 下运行。

要运行 mysqld 或 mysqld-opt，可如下启动该服务器：

```
C:\> C:\mysql\bin\mysqld  
C:\> C:\mysql\bin\mysqld-opt
```

要关闭此服务器，可利用 mysqladmin 实用程序：

```
C:\> C:\mysql\bin\mysqladmin -u root shutdown
```

在 Windows NT 下，可运行 mysqld-nt 作为一个服务：

```
C:\> C:\mysql\bin\mysqld-nt --install
```

如果以这种方式启动 mysqld-nt，可通过将选项放入选项文件 C:\my.cnf 中来指定它们。不能在命令行上指定这些参数。在 mysqld-nt 作为一个服务运行时，可利用下列命令启动或停止它：

```
C:\> net start mysql  
C:\> net stop mysql
```

可以使用 control Panel 文件夹中的 Services Control Manager 实用程序来启动和停止服务器，或者从命令行用 mysqladmin 实用程序来停止服务器：

```
C:\> C:\mysql\bin\mysqladmin -u root shutdown
```

为了作为一个独立的程序运行 mysqld-nt，按如下方式调用它：

```
C:\> C:\mysql\bin\mysqld-nt --standalone
```

在此情形中，如果想要的话，可在命令行上 `--standalone` 选项之后指定其他选项。为了关闭此服务器，使用 `mysqladmin`。

如果有关于运行服务器的问题，请参阅 MySQL 参考指南的“安装 MySQL”一章。

缺省安装允许 MySQL root 用户不用口令进行连接。关于设置一个口令的说明，请参阅第 11 章。

A.4.2 安装 MyODBC

打开恰当的分发包（一个用于 Windows 95 或 98，一个用于 Windows NT）。在产生的文件夹中运行 Setup 程序安装 MySQL ODBC 驱动。ODBC 控制面板也要安装，可将其用来配置此驱动。

如果在安装 MyODBC 中出现错误，如“Problems while copying MFC30.DLL”，则表示 MFC30.DLL 正被某个应用程序使用。在此情形下，以安全模式重新启动 Windows，然后执行 Setup 程序。

在运行 ODBC 控制面板时，将看到一个窗口，此窗口允许设置数据源名（DSN）。单击 User DSN 标签，然后单击 Add 按钮，会出现一个列出可用数据源驱动的窗口。从清单中选择 MySQL 驱动，单击 Finish 按钮。将看到录入数据源连接参数的窗口。填入适合于希望建立的连接的参数，然后单击 OK 按钮。例如，为了利用本书中最常用的服务器和用户账号设置 samp_db 数据库的数据源，按如下填入各字段：

字段名	字段值
Windows DSN name:	samp_db
MySQL host (name or IP):	pit-viper.snake.net
MySQL database name:	samp_db
User:	paul
Password:	secret

现在应该能够使用支持 ODBC 的程序访问 MySQL 数据库了。

A.4.3 安装 Perl DBI 支持环境

从 ActiveState Web 站点（<http://www.activestate.com/>）取得 ActiveState Perl 分包并安装它。然后，取得和安装所需的其他 Perl 模块。ppm（Perl Package Manager）程序就是用于此目的的。为了查找已经安装了哪些模块，可用下列命令：

```
C:\> C:\perl\bin\ppm info
```

然后使用下面所列的恰当命令安装所需模块。可能 CGI.pm 已经安装，大概还需要安装与 DBI 有关的程序包。

```
C:\> C:\perl\bin\ppm install Data-Dumper
C:\> C:\perl\bin\ppm install DBI
C:\> C:\perl\bin\ppm install DBD-mysql
C:\> C:\perl\bin\ppm install CGI
```

A.4.4 安装 PHP 和 Apache

在 Windows 下，PHP 是独立可执行的，而 MySQL 支持环境是一个 DLL 文件。为了完

成在 Windows 上安装 PHP，要做的最好的事情是访问下列的 Web 页：

<http://www.umesd.k12.or.us/php/win32install.html>

A.5 获得和安装 samp_db 样例数据库分发包

用来设置和访问 samp_db 样例数据库的文件可在下列地址得到：

<http://www.mysql.com/Contrib/Examples/>

检索分发包文件 samp_db.tar.gz，进入想安装它的目录，用下列命令之一打开它：

```
% tar xzf samp_db.tar.gz
% gunzip < samp_db.tar.gz | tar xf -
```

在打开分发包时，它将创建一个名为 samp_db 的目录，包含下列内容：

- 创建和装载 samp_db 数据库的文件。它们在第 1 章中使用。
- 第 6 章所用的 C 程序。
- 第 7 章所用的 Perl DBI 脚本和 samp_db.cnf 选项文件。
- 第 8 章所用的 PHP 脚本和 samp_db.inc 包含文件。

samp_db 目录含有一个 README 文件，应该参阅其有关使用分发包文件的说明。

附录B 列类型参考

本附录介绍 MySQL 提供的每种列类型。关于利用每种类型的详细说明请参阅第 2 章。除非另有说明，否则所列出的类型早在 MySQL 3.21.0 中就已经有了。

按下列约定给出类型名说明：

■ **方括号 ([])** 可选信息。

■ **M** 最大显示宽度。除非另有说明，否则 M 应该是一个 1 到 255 之间的整数。

■ **D** 有小数部分的类型的小数位数。D 为一个 0 到 30 之间的整数。D 应该小于等于 M - 2。否则，M 的值将调整为 D + 2。

在 ODBC 术语中，M 和 D 称为“精度”和“小数位数”。

为了描述每种类型，提供下列一个或多个信息：

■ **说明** 该类型的简短说明。

■ **允许的属性** 在 CREATE TABLE 或 ALTER TABLE 语句中与该类型相关的可选属性关键字。属性按字母顺序列出，但这个顺序并不一定与 CREATE TABLE 或 ALTER TABLE 中使用的顺序相对应。关于这些语句的语法请参阅附录 D。每种列类型在说明中列出的属性为通用属性之外的属性。

■ **允许的长度** 串类型的列值所允许的最大长度。

■ **取值范围** 对于数值或日期与时间类型，该类型可表示的值的范围。对于整数数值类型，给出两个取值范围，因为整数列可以是有符号或无符号的，有符号或无符号的取值范围是不同的。

■ **零值** 对于日期和时间类型，如果将非法值插入该列，则存入一个“零”值。

■ **缺省值** 类型说明中不明确给出 DEFAULT 属性时的缺省值。

■ **存储需求** 存储该类型的值所需的字节数。对于有的类型，这个值是固字的。而这个值在有些类型中根据存放在列中的值的不同是可变的。

■ **比较** 对于串类型，比较是否是区分大小写的。这对排序和索引有效，因为这些操作根据比较进行的。

■ **同义词** 该类型名的同义词。

■ **注释** 关于该类型的一些杂项说明。

■ **通用属性** 有的属性对所有类型或对于几乎所有类型都适用。它们列在这里，而不是列在每种类型的说明中。可对任何类型指定 NULL 或 NOT NULL 属性。还可对所有类型指定 DEFAULT default_value。

B.1 数值类型

MySQL 提供了整数和浮点数的数值类型。可根据所要表示的值的范围选择相应的类型。

对于整数类型，如果指定了 AUTO_INCREMENT 属性，则列必须为 PRIMARY KEY 或 UNIQUE 索引。将 NULL 插入 AUTO_INCREMENT 列将会插入一个大于该列中当前最大值

的值。如果指定了 UNSIGNED 属性，则相应的整数类型不允许取负值。

如果指定了 ZEROFILL 属性，则用前导零填充数值类型值以达到列的显示宽度。

1. TINYINT[(M)]

说明：非常小的整数

允许的属性：AUTO_INCREMENT, UNSIGNED, ZEROFILL

取值范围：-128 到 127 (-2^7 到 2^7-1)，或者如果为 UNSIGNED，则 0 到 255 (0 到 2^8-1)

缺省值：如果列可为 NULL；则为 NULL；如果列为 NOT NULL，则为 0

存储需求：1 字节

2. SMALLINT[(M)]

说明：小整数

允许的属性：AUTO_INCREMENT, UNSIGNED, ZEROFILL

取值范围：-32768 到 32767 (-2^{15} 到 $2^{15}-1$)，或者如果为 UNSIGNED，则 0 到 65535 (0 到 $2^{16}-1$)

缺省值：如果列可为 NULL；则为 NULL；如果列为 NOT NULL，则为 0

存储需求：2 字节

3. MEDIUMINT[(M)]

说明：中等大小的整数

允许的属性：AUTO_INCREMENT, UNSIGNED, ZEROFILL

取值范围：-8388608 到 8388607 (-2^{23} 到 $2^{23}-1$)，或者如果为 UNSIGNED，则 0 到 1677215 (0 到 $2^{24}-1$)

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 0

存储需求：4 字节

4. INT[(M)]

说明：标准大小的整数

允许的属性：AUTO_INCREMENT, UNSIGNED, ZEROFILL

取值范围：-2147483648 到 2147483647 (-2^{31} 到 $2^{31}-1$)，或者如果为 UNSIGNED，则 0 到 4294967295 (0 到 $2^{32}-1$)

缺省值：如果列为 NULL，则为 NULL；如果列为 NOT NULL，则为 0

存储需求：4 字节

同义词：INTEGER[(M)]

5. BIGINT[(M)]

说明：大整数

允许的属性：AUTO_INCREMENT, UNSIGNED, ZEROFILL

取值范围：-9223372036854775808 到 9223372036854775807 (-2^{63} 到 $2^{63}-1$)，或者如果为 UNSIGNED，则 0 到 18446744073709551615 (0 到 $2^{64}-1$)

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 0

存储需求：8 字节

6. FLOAT[(M, D)]

说明：小浮点数；单精度（精度小于 DOUBLE）

允许的属性：ZEROFILL

取值范围：最小非零值为 $\pm 1.75494351\text{E} - 38$ ；最大非零值为 $\pm 3.402823466\text{E} + 38$

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 0

存储需求：4 字节

同义词：MySQL 3.23 版以前，FLOAT(4) 为具有缺省 M 和 D 值的 FLOAT 的同义词。

注释：在 MySQL 3.23 以后，FLOAT(4) 为真浮点类型（值存储为硬件所允许的完整精度，而不四舍五入为缺省的小数位数）。

7. DOUBLE[(M, D)]

说明：大浮点数；双精度（比 FLOAT 更精确）

允许的属性：ZEROFILL

取值范围：最小非零值为 $\pm 2.2250738585072014\text{E} - 308$ ；最大非零值为 $\pm 1.7976931348623157\text{E} + 308$

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 0

存储需求：8 字节

同义词：DOUBLE PRECISION[(M, D)] 和 REAL[(M, D)] 为 DOUBLE[(M, D)] 的同义词。MySQL 3.23 版以前，FLOAT(8) 为具有缺省 M 和 D 值的 FLOAT 的同义词。

注释：在 MySQL 3.23 以后，FLOAT(8) 为真浮点类型（值存储为硬件所允许的完整精度，而不四舍五入为缺省的小数位数）。

8. DECIMAL(M, D)

说明：存储为串的浮点数（每位数字、小数点或“-”号都占 1 字节）。

允许的属性：ZEROFILL

取值范围：最大取值范围与 DOUBLE 相同；给定 DECIMAL 类型的有效取值范围由 M 和 D 决定。如果 D 为零，则列值无小数点或小数部分。

缺省值：如果列可为 NULL，则为 NULL，如果列为 NOT NULL，则为 0

存储需求：对于 MySQL 3.23 前的版本，为 M 字节，而 MySQL 3.23 以后的版本，为 M + 2 字节。

同义词：NUMERIC(M, D)

注释：在 MySQL 3.23 以后，M 的值为符合 ANSI SQL 标准，不包括符号字符或小数点所占的字节数。

B.2 串类型

MySQL 的串类型常用来存储文本，它不但是通用的而且还能存储任何数据。可用串类型来存储最大长度可变的值，而且可以选择在处理值时是否区分大小写。

1.CHAR(M)

说明：0 到 M 字节长的定长字符串。在 MySQL 3.23 版以前，M 应该为一个 1 到 255 之间的整数。而 MySQL 3.23 版以后，M 应该为一个 0 到 255 之间的整数。短于 M 个字符的串存储进右边补空格。长于 M 个字符的串存储时剪断为长度是 M 的串。在检索值时，去掉后跟的空格。

允许的属性：BINARY

允许的长度：0 到 M 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”

存储需求：M 字节

比较：不区分大小写（如果具有 BINARY 属性，则区分大小写）

2. VARCHAR(M)

说明：0 到 M 字节长的可变长字符串。M 应该为 1 到 255 之间的一个整数，或者自 MySQL 3.23 后为 0 到 255 之间的一个整数。存储时后跟的空格被去掉。存储时，大于 M 个字符的串剪断为 M 个字符。

允许的属性：BINARY

允许的长度：0 到 M 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”

存储需求：值的长度，加上 1 字节用来记录长度

比较：不区分大小写（如果具有 BINARY 属性，则区分大小写）

3. TINYBLOB

说明：小 BLOB 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 255 (0 到 2^8-1) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”

存储需求：值的长度加上用于记录长度的 1 个字节

比较：区分大小写

4. BLOB

说明：常规大小的 BLOB 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 65535 (0 到 $2^{16}-1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”

存储需求：值的长度加上用于记录长度的 2 个字节

比较：区分大小写

5. MEDIUMBLOB

说明：中等大小的 BLOB 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 16777215 (0 到 $2^{24}-1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”

存储需求：值的长度加上用于记录长度的 3 个字节

比较：区分大小写

6. LONGBLOB

说明：大 BLOB 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 4294967295 (0 到 $2^{32}-1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为“ ”

存储需求：值的长度加上用于记录长度的 4 个字节

比较：区分大小写

7. TINYTEXT

说明：小 TEXT 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 255 (0 到 $2^8 - 1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为“ ”

存储需求：值的长度加上用于记录长度的 2 个字节

比较：不区分大小写

8. TEXT

说明：常规大小的 TEXT 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 65535 (0 到 $2^{16} - 1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为“ ”

存储需求：值的长度加上用于记录长度的 2 个字节

比较：不区分大小写

9. MEDIUMTEXT

说明：中等大小的 TEXT 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 16777215 (0 到 $2^{24} - 1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为“ ”

存储需求：值的长度加上用于记录长度的 3 个字节

比较：不区分大小写

10. LONGTEXT

说明：大 TEXT 值

允许的属性：除通用属性外无其他属性

允许的长度：0 到 4294967295 (0 到 $2^{32} - 1$) 字节

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为“ ”

存储需求：值的长度加上用于记录长度的 4 个字节

比较：不区分大小写

11. ENUM(“ value1 ”, “ value2 ”, ...)

说明：枚举，列值可赋予值列表中的某个成员

允许的属性：除通用属性外无其他属性

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为第一个枚举值

存储需求：对 1 到 255 个成员的枚举 1 个字节，对 255 到 65535 个成员的枚举 2 个字节

比较：不区分大小写 (MySQL 3.22.1 版以前区分大小写)

12. SET(“ value1 ”, “ value2 ”, ...)

说明：集合，列值可赋予值列表中的零个或多个成员

允许的属性：除通用属性外无其他属性

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ ”（空集）

存储需求：1字节（1到8个成员的集合），2字节（9 到16个成员的集合），3字节（17到24个成员的集合），4 字节（25到32个成员的集合）或8字节（33到64个成员的集合）

比较：不区分大小写（MySQL 3.22.1 版以前区分大小写）

B.3 日期和时间类型

MySQL 提供以各种形式表示时间数据的类型。这些数据有日期和时间类型。有一种特殊的时间戳类型，它在记录更改时自动更新。还有一种用于存储年份的类型，在不需要完全的日期时使用。

1. DATE

说明：“ YYYY-MM-DD ” 格式的日期

允许的属性：除通用属性外无其他属性

取值范围：“ 1000-01-01 ” 到 “ 9999-12-31 ”

零值：“ 0000-00-00 ”

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ 0000-00-00 ”

存储需求：3 字节（MySQL 3.22 版以前为 4 字节）

2. TIME

说明：“ HH:MM:SS ” 格式的时间（负值为 “ - HH:MM:SS ”）；表示占用的时间，但也可作为日常时间

允许的属性：除通用属性外无其他属性

取值范围：“ -838:59:59 ” 到 “ 838:59:59 ”

零值：“ 00:00:00 ”

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ 00:00:00 ”

存储需求：3 字节

注释：虽然在非法值插入 TIME 列时，“ 00:00:00 ” 用作零值，但它也作为一个合法的值出现在正常的取值范围内

3. DATETIME

说明：以 “ YYYY-MM-DD hh:mm:ss ” 格式表示的日期和时间（两个部分都需要）

允许的属性：除通用属性外无其他属性

取值范围：“ 1000-01-01 00:00:00 ” 到 “ 9999-12-31 23:59:59 ”

零值：“ 0000-00-00 00:00:00 ”

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 “ 0000-00-00 00:00:00 ”

存储需求：8 字节

4. TIMESTAMP[(M)]

说明：以 YYYYMMDDHHMMSS 格式表示的时间（日期和时间）

允许的属性：除通用属性外无其他属性

取值范围：19700101000000 到 2037 年的某个时刻

缺省值：当前日期和时间。注意 DESCRIBE 和 SHOW COLLUMNS 报告缺省值为 NULL

存储需求：4 字节

注释：插入 NULL 到表的第一个 TIMESTAMP 列将会插入当前日期和时间。更改行中其他列的值使第一个 TIMESTAMP 列被更新为这个修改的日期和时间。在内部计算中存储和使用的值全都为 14 位字符精度，与显示宽度无关。如果指定了 NOT NULL 属性，则此属性不起作用

5. YEAR

说明：YYYY 格式表示的年份

允许的属性能：除通用属性外无其他属性

取值范围：1900 到 2155

零值：0000

缺省值：如果列可为 NULL，则为 NULL；如果列为 NOT NULL，则为 0000

存储需求：1 字节

注释：YEAR 是在 MySQL 3.22 版中引入的

附录C 运算符和函数参考

本附录列出所有可在 SQL 语句中使用的 MySQL 函数，还给出构造表达式的运算符。表达式求值常常涉及对表达式中值的类型转换。关于类型转换的环境及 MySQL 用来将值从一种类型转换为另一种类型的规则，可参阅第 2 章。

除非另有说明，否则所列出的运算符和函数在 MySQL 中得到支持至少可回溯到 MySQL 3.21.0 版。如果某个运算符或函数的功能与这里介绍的不同，可参阅 MySQL 参考指南中的注释。很可能是在旧的 MySQL 版本中有问题，但后来已经解决了。

运算符和函数的例子以下列形式给出：

expression → **result**

其中 expression 说明怎样使用运算符或函数，而 result 给出表达式求值的结果。例如：

LOWER("ABC") → **"abc"**

其含意是函数调用 LOWER("ABC") 得出结果串 "abc"。

可利用 mysql 程序试验本附录中所给出的例子。为试验上面的例子，调用 mysql，输入该例子，在其前面放上 SELECT，后面放一个分号，然后按 Enter 键。如下所示：

```
mysql> SELECT LOWER("ABC");
+-----+
| LOWER("ABC") |
+-----+
| abc          |
+-----+
```

MySQL 不需要具有 FROM 子句的 SELECT 语句，这使得以这种方式输入任意表达式对运算符和函数进行试验可以变得很容易（有的数据库系统不允许发布无 FROM 子句的 SELECT 语句，这是一个不好的限制）。

各个例子都包含有不能在其他地方示范的函数的完整表达式。“函数汇总”一节就是按这种方式书写的，因为这些函数除了只在特定的表中引用外是没有意义的。

函数名以及为单词形式的操作符（如 BETWEEN）可以由任意的大小写字符来指定。

有的函数参数重复出现，具有固定意义：

- *expr* 代表一个表达式，在不同的上下文中，可能代表数值、串或日期、时间表达式，而且可与常量、表列引用或其他表达式结合。
- *str* 代表一个串，可为字符串、串值表列引用或产生串的表达式。
- *n* 代表一个整数（还可以用字母表中与 *n* 相近的字母来表示）。
- *x* 代表一个浮点数（还可以用字母表中与 *x* 相近的字母来表示）。

其他一些参数名较少使用，因此在使用的地方再定义。运算符或函数调用序列的可选部分由方括号 ([]) 表示。

C.1 运算符

运算符用来结合表达式中的项以完成算术、比较、位逻辑或逻辑操作以及模式匹配。

C.1.1 运算符的优先级

运算符优先级的级别不同，这些级别在下面的清单中给出，从最高到最低。相同行上的运算符具有相同的优先级。给定优先级的运算符从左到右求值。较高优先级的运算符在较低优先级的运算符之前求值。

```

BINARY
NOT !
- (unary minus)
* / %
+ -
<< >>
&
|
< <= = <=> != <> >= > IN IS LIKE REGEXP RLIKE
BETWEEN
AND &&
OR ||

```

一元运算符（一元减、NOT 和 BINARY）较二元运算符更为紧密。即，它们与表达式中紧跟的项结合在一起，而不是与整个表达式结合在一起。如：

```

--2+3          → 1
-(2+3)         → -5

```

C.1.2 分组运算符

圆括号 (和) 可用于将表达式分组，以及改变运算符的决定表达式中各项求值顺序的缺省优先级。圆括号还可用来使表达式的含义明确，更好理解。如：

```

1 + 2 * 3 / 4          → 2.50
(((1 + 2) * 3) / 4)    → 2.25

```

C.1.3 算术运算符

这些运算符进行标准的算术运算。算术运算符对数而不是对串起作用（但是类似于数的串可自动转换这相应的数值值）。涉及 NULL 值的算术运算得出一个 NULL 结果。

■ + 求算术和。

```

2 + 2          → 4
3.2 + 4.7      → 7.9
"43bc" + "21d" → 64
"abc" + "def"  → 0

```

上面最后一个例子说明 “+” 不像在某些语言中那样可作为串连接运算符。这里，进行算术运算之前，串被转换为数，不是数的串转换为 0。连接串应该用 CONCAT() 函数。

■ - 在用于表达式的两项之间时，求两操作数之差。而用在单项之前时，求操作数的相反数（即，使项反号）。

```

10 - 7          → 3
-(10 - 7)       → -3

```

■ * 求操作数之积。

```

2 * 3          → 6
2.3 * -4.5     → -10.3

```


■ / 求操作数之商。零除得 NULL。

```
3 / 1          → 3.00
1 / 3          → 0.33
1 / 0          → NULL
```

■ % 模运算符；求 m 除以 n 的余数。m % n 与 MOD(m, n) 相同。与除一样，如果除数为零，则结果为 NULL。

```
12 % 4         → 0
12 % 5         → 2
12 % 0         → NULL
```

对于“+”、“-”和“*”，如果两边参数都是整数，则算术运算以 BIGINT 值进行（64 位整数）。这意味着较大值的表达式可能会超出 64 位整数计算的取值范围，其结果不可预料。如：

```
999999999999999999 * 999999999999999999 → -7527149226598858751
999999999999 * 999999999999 * 999999999999 → -1504485813132150785
18014398509481984 * 18014398509481984 → 0
```

对于“/”和“%”，BIGINT 值仅在结果转换为整数的上下文中使用。

C.1.4 比较运算符

如果比较为真，比较运算符返回 1；如果比较为假，返回 0，可对数或串进行比较，操作数根据下列规则进行转换：

- 对于“<=>”运算符的表达式，当比较中包含 NULL 值时，其结果都为 NULL（除 NULL <=> NULL 为真外，“<=>”与“=”一样）。
- 如果两个操作数皆为串，它们作为串按字典顺序比较。串比较按服务器上有效的字符集进行。
- 如果两个操作数为整数，它们按整数进行数值比较。
- 不与数进行比较的十六进制常量按二进制串进行比较。
- 如果一个操作数为 TIMESTAMP 或 DATETIME 值，而另一操作数是一个常量，则此两个操作数按 TIMESTAMP 值进行比较。这样使比较对 ODBC 应用更好。
- 如果前面任一规则都不适用，那么操作数按浮点数进行比较。注意，这包括串的比较和数的比较。串转换为一个数，如果串与数不类似，其转换结果为 0。例如，“14.3”转换为 14.3，但“L4.3”转换为 0。

下面的比较说明了这些规则：

```
2 < 12          → 1
"2" < "12"      → 0
"2" < 12        → 1
```

第一个比较涉及两个整数，所以进行数值比较。第二个比较涉及两个串，按字典顺序进行比较。第三个比较涉及一个串和一个数，因此它们按浮点值进行比较。

除非比较涉及二进制串，否则串比较是不区分大小写的。如果使用 BINARY 关键字或比较 CHAR BINARY、VARCHAR BINARY 或 BLOB 列的值，则进行区分大小写的比较。

■ = 如果操作数相等，其值为 1，否则为 0。

```
1 = 1          → 1
1 = 2          → 0
```

```

"abc" = "abc"           → 1
"abc" = "def"           → 0
"abc" = "ABC"           → 1
BINARY "abc" = "ABC"    → 0
BINARY "abc" = "abc"    → 1
"abc" = 0                → 1

```

“abc”既等于“abc”又等于“ABC”，因为缺省时，串比较是不区分大小写的。利用 BINARY 运算符，串比较是区分大小写的。“abc”等于 0，因为根据比较规则，它要转换为一个数值。而“abc”不类似于数，所以转换为 0。

■ **<=>** NULL-安全 (NULL-safe) 的等于运算符；在操作数相等时，其值为 1，即使操作数为 NULL 也是这样。它类似于“=”。

```

1 <=> 1                  → 1
1 <=> 2                  → 0
NULL <=> NULL            → 1
NULL = NULL              → NULL

```

最后两个例子显示了“=”和“<=>”处理 NULL 比较的差异。

■ **!= 或 <>** 如果操作数不相等，其值为 1，否则其值为 0。

```

3.4 != 3.4               → 0
"abc" <> "ABC"            → 0
BINARY "abc" <> "ABC"     → 1
"abc" != "def"           → 1

```

■ **<** 如果左操作数小于右操作数，其值为 1，否则其值为 0。

```

3 < 10                   → 1
105.4 < 10e+1            → 0
"abc" < "ABC"             → 0
"abc" < "def"            → 1

```

■ **<=** 如果左操作数小于等于右操作数，其值为 1，否则其值为 0。

```

"abc" <= "a"             → 0
"a" <= "abc"             → 1
13.5 <= 14                → 1
(3 * 4) - (6 * 2) <= 0    → 1

```

■ **>=** 如果左操作数大于等于右操作数，其值为 1，否则其值为 0。

```

"abc" >= "a"             → 1
"a" >= "abc"             → 0
13.5 >= 14                → 0
(3 * 4) - (6 * 2) >= 0    → 1

```

■ **>** 如果左操作数大于右操作数，其值为 1，否则其值为 0。

```

PI() > 3                  → 1
"abc" > "a"              → 1
SIN(0) > COS(0)          → 0

```

■ **expr BETWEEN min AND max** 如果 min 小于等于 expr 且 max 大于等于 expr，其值为

1. 如果操作数 expr、min 和 max 的类型都相同，此表达式等价于：

```

expr BETWEEN min AND max
(min <= expr AND expr <= max)

```

如果这些操作数类型不同，将进行类型转换，上面两个表达式可能不等价。 BETWEEN

利用根据 *expr* 的类型所决定的比较进行求值：

- 如果 *expr* 为一个串，则操作数按串进行字典顺序的比较。比较时区分大小写或不区分大小写，由 *expr* 是否二进制串来决定。
- 如果 *expr* 为一个整数，则操作数按整数进行比较。
- 如果上两个规则都不真，则操作数按浮点数进行数值比较。

```
"def" BETWEEN "abc" and "ghi"      → 1
"def" BETWEEN "abc" and "def"      → 1
13.3 BETWEEN 10 and 20             → 1
13.3 BETWEEN 10 and 13             → 0
2 BETWEEN 2 and 2                  → 1
"B" BETWEEN "A" and "a"            → 0
BINARY "B" BETWEEN "A" and "a"     → 1
```

BETWEEN 是在 MySQL 版本 3.21.2 中引入的。

■ *expr* IN (*value1*, *value2*, ...)

expr NOT IN (*value1*, *value2*, ...)

如果 *expr* 为列表中某个值，则 IN 的求值为 1，否则为 0。对于 NOT IN，求值正好相反。

下面的表达式是等价的：

```
expr NOT IN (value1, value2, ...)
NOT (expr IN (value1, value2, ...))
```

如果列表中所有值都是常量，MySQL 将对它们进行排序，并利用二分查找对 IN 进行测试，这是非常快的。

```
3 IN (1,2,3,4,5)                  → 1
"d" IN ("a","b","c","d","e")      → 1
"f" IN ("a","b","c","d","e")      → 0
3 NOT IN (1,2,3,4,5)              → 0
"d" NOT IN ("a","b","c","d","e")  → 0
"f" NOT IN ("a","b","c","d","e")  → 1
```

■ *expr* IS NULL

expr IS NOT NULL

如果 *expr* 的值为 NULL，则 IS NULL 求值为 1，否则为 0。IS NOT NULL 正好相反。下面的表达式是等价的：

```
expr IS NOT NULL
NOT (expr IS NULL)
```

IS NULL 和 IS NOT NULL 应该用来确定 *expr* 的值是否为 NULL。因为这时不能使用普通的比较运算符“=”和“!=”。

```
NULL IS NULL                      → 1
0 IS NULL                         → 0
NULL IS NOT NULL                  → 0
0 IS NOT NULL                     → 1
NOT (0 IS NULL)                   → 1
NOT (NULL IS NULL)                → 0
NOT NULL IS NULL                  → 1
```

最后一例得出该结果是因为 NOT 比 IS 的约束更紧（参阅“运算符优先级”一节）。

C.1.5 位运算符

位运算符利用 BIGINT 值（64 位整数）执行，它限定了运算的最大取值范围。涉及

NULL值的位运算产生 NULL 结果。

■ | 对操作数进行按位 OR (或)。

1 1	→ 1
1 2	→ 3
1 2 4 8	→ 15
1 2 4 8 15	→ 15

■ & 对操作数进行按位 AND (与)。

1 & 1	→ 1
1 & 2	→ 0
7 & 5	→ 5

■ << 将左操作数左移右操作数所指定的位数。移动负数位结果为零。

1 << 2	→ 4
2 << 2	→ 8
1 << 62	→ 4611686018427387904
1 << 63	→ - 9223372036854775808
1 << 64	→ 0

■ >> 将左操作数右移右操作数所指定的位数。移动负数位结果为零。

16 >> 3	→ 2
16 >> 4	→ 1
16 >> 5	→ 0

C.1.6 逻辑运算符

逻辑运算符（也称为布尔运算符）测试表达式的真和假。所有逻辑运算在真时返回 1，假时返回 0。逻辑运算解释非零操作数为真，0 操作数为假。NULL 值按运算符说明中所规定的处理。

逻辑运算符希望操作数为数值，因此在运算符执行之前，串操作数将被转换为数。

■ NOT 或 ! 逻辑否定。如果所跟的操作数为假，则其值为 1，否则其值为 0，而 NOT NULL 求值为 NULL。

NOT 0	→ 1
NOT 1	→ 0
NOT NULL	→ NULL
NOT 3	→ 0
NOT NOT 1	→ 1
NOT "1"	→ 0
NOT "0"	→ 1
NOT ""	→ 1
NOT "abc"	→ 1

■ OR 或 || 逻辑 OR。如果任一操作数为真（非零或非 NULL），则其值为 1，否则为 0。

0 OR 0	→ 0
0 OR 3	→ 1
4 OR 2	→ 1
1 OR NULL	→ 1

■ AND 或 && 逻辑 AND。如果两个操作数都为真（非零或非 NULL），则其值为 1，否则为 0。

0 AND 0	→ 0
0 AND 3	→ 0

```
4 AND 2          → 1
1 AND NULL       → 0
```

在 MySQL 中，“!”、“||”和“&&”表示逻辑运算，正如它们在 C 中一样。但要特别注意，“||”，不像在某些 SQL 版本中那样进行串的连接。这里连接串应该用 CONCAT()。

C.1.7 强制运算符

BINARY 使后跟的操作数被当作一个二进制串，从而使得涉及串的比较为区分大小写的。如果后跟的操作数为一个数，将被转换为串的形式：

```
"abc" = "ABC"          → 1
"abc" = BINARY "ABC"    → 0
BINARY "abc" = "ABC"    → 0
"2" < 12                → 1
"2" < BINARY 12         → 0
```

BINARY 产生数到串的转换，因此比较按字典序进行，因为这时两个操作数都是串。

BINARY 在 MySQL 3.23.0 中引入。

C.1.8 模式匹配运算符

MySQL 利用 LIKE 提供了 SQL 的模式匹配，利用 REGEXP 提供了扩展正规表达式模式匹配。除非要匹配的串或模式串为二进制串，否则 SQL 模式匹配是不区分大小写的。扩展正规表达式模式匹配总是区分大小写的。

SQL 模式匹配仅在模式与要匹配的整个串相匹配时才会成功。扩展正规表达式模式匹配仅在模式在串中任意地方找到才会成功。

■ str LIKE pat [ESCAPE 'c']

str NOT LIKE pat [ESCAPE 'c']

LIKE 完成一种简单的 SQL 模式匹配，如果模式串 pat 与整个串表达式 str 匹配，则其值为 1。如果不匹配，其值为 0。对于 NOT LIKE，正好相反。下列两个表达式是等价的：

```
str NOT LIKE pat [ESCAPE 'c']
NOT (str LIKE pat [ESCAPE 'c'])
```

如果任一个串为 NULL，则结果为 NULL。

在 SQL 模式中有两个字符具有特殊的含义，被用作通配符，说明如下：

字符	说明
%	匹配除 NULL 外的任意字符序列（包括空串）
_	匹配单个字符

模式可以包含任一通配符，或两个都包含：

```
"catnip" LIKE "cat%"      → 1
"dogwood" LIKE "%wood"    → 1
"bird" LIKE "____"        → 1
"bird" LIKE "___"         → 0
"dogwood" LIKE "%wo__"    → 1
```

使用 LIKE 的 SQL 模式匹配的大小写敏感性由被比较的串决定。一般情况下，比较是不区分大小写的。如果有一个串是二进制串，比较将是区分大小写的：

```
"abc" LIKE "ABC"           → 1
BINARY "abc" LIKE "ABC"    → 0
"abc" LIKE BINARY "ABC"    → 0
```

因为“%”与任意字符序列匹配，它甚至能匹配无字符的情况：

```
" " LIKE "%"           → 1
"cat" LIKE "cat%"      → 1
```

在 MySQL 中，可对数值表达式使用 LIKE：

```
50 + 50 LIKE "1%"       → 1
200 LIKE "2__"          → 1
```

为了按字面意义匹配一个通配符，应该在其前面放置一个转义字符“\”：

```
"100% pure" LIKE "100%"      → 1
"100% pure" LIKE "100\%"    → 0
"100% pure" LIKE "100\% pure" → 1
```

如果想使用非“\”的另一转义字符，可利用 ESCAPE 子句进行指定：

```
"100% pure" LIKE "100^%" ESCAPE '^' → 0
"100% pure" LIKE "100^% pure" ESCAPE '^' → 1
```

■ **str REGEXP pat**

str NOT REGEXP pat

REGEXP 执行扩展正规表达式模式匹配。扩展正规表达式类似于 UNIX 实用程序 grep 和 sed 所用的模式。可使用的模式序列如表 C-1 所示。

表 C-1 扩展正规表达式序列

序 列	说 明
^	在串的开始处匹配
\$	在串的结尾处匹配
.	匹配任意单个字符，包括换行符
[...]	匹配出现在方括号中的任意字符
[^...]	匹配不出现在方括号中的任意字符
e*	匹配模式元素 e 的零个或多个实例
e+	匹配模式元素 e 的1个或多个实例
e?	匹配模式元素 e 的零个或多个实例
e1 e2	匹配模式元素 e1 或 e2
e{m}	匹配模式元素 e 的m 个实例
e{m,}	匹配模式元素 e 的m 个或更多个实例 (
e{,n}	匹配模式元素 e 的0 到 n 个实例
e{m,n}	匹配模式元素 e 的m 到 n 个实例
(...)	将模式元素组成单一元素
other	无特定字符与它们自身相配

如果模式串 pat 与串表达式 str 匹配，则 REGEXP 求值为 1，否则为 0。NOT REGEXP 正好相反。下列两个表达式是等价的：

```
str NOT REGEXP pat
NOT (str REGEXP pat)
```


如果其中有一个串为 NULL，则结果为 NULL。

模式不需要与整个串匹配，只需要在串中某处找出它即可。

[...] 和 [^...] 结构指定字符类。在一个类中，字符的取值范围可用两个端点字符中间加一短划线来表示。如，[a-z] 与任何小写字母匹配，而 [0-9] 与任意数字匹配。为了能够在类中表示字符 “]”，它必须为相应类的第一个字符。为了能够表示字符 “-”，它必须是相应类的第一个或最后一个字符。而为了能够表示 “^”，它必须不是 “[” 之后的第一个字符。

还有其他几个与比较序列有关的更为复杂的特殊构造和等价类可用于字符类中。更详细的信息，请参阅 MySQL 参考指南。

如果在要匹配的串中的任何地方找到相应模式，则扩展正规表达式匹配成功，但可用 “^” 和 “\$” 强制模式只在串的开始或结尾处进行匹配。

"abcde" REGEXP "b"	→ 1
"abcde" REGEXP "^b"	→ 0
"abcde" REGEXP "b\$"	→ 0
"abcde" REGEXP "^a"	→ 1
"abcde" REGEXP "e\$"	→ 1
"abcde" REGEXP "^a.*e\$"	→ 1

扩展正规表达式模式匹配是区分大小写的：

"abc" REGEXP "ABC"	→ 0
"ABC" REGEXP "ABC"	→ 1

■ *str* RLIKE *pat*

str NOT RLIKE *pat*

RLIKE 和 NOT RLIKE 为 REGEXP 和 NOT REGEXP 的同义词。

对于串中的转义序列，MySQL 所用的语法类似于 C。例如，‘\n’、‘\t’ 和 ‘\\’ 分别解释为换行、制表和反斜杠。为了在模式中指定这样的字符，可使用双反斜杠，‘\\n’、‘\\t’ 和 ‘\\\\’。在对查询进行分析时，去掉一个反斜杠，而在进行模式匹配时，才对其余的转义序列进行解释。

C.2 函数

调用函数以执行计算并返回一个值。函数调用必须在函数名与其后的圆括号之间无空格。

NOW()	正确
NOW ()	不正确

如果将多个参数传递给一个函数，则参数必须用逗号分隔。函数参数周围允许有空格：

CONCAT(" abc ", " def ")	正确
CONCAT(" abc ", " def ")	也正确

C.2.1 比较函数

■ GREATEST(*expr1*, *expr2*, ...) 返回最大的参数，其中“最大”根据下列规则定义：

- 如果在整数环境中调用此函数，或者所有参数都是整数，则各参数按整数进行比较。
- 如果在浮点数环境中调用此函数，或者所有参数都是浮点数，则各参数按浮点数进行比较。
- 如果前两个规则都不真，则各参数按串进行比较。除非有的参数为二进制串，否

则此比较是不区分大小写的。

GREATEST(2,3,1)	→ 3
GREATEST(38.5,94.2,-1)	→ 94.2
GREATEST("a","ab","abc")	→ "abc"
GREATEST(1,3,5)	→ 5
GREATEST("A","b","C")	→ "C"
GREATEST(BINARY "A","b","C")	→ "b"

GREATEST() 在 MySQL 3.22.5 版中引入。在更早的版本中,用的是 MAX()。

■ IF(*expr1*, *expr2*, *expr3*) 如果 *expr1* 为真 (非 0 或 NULL), 返回 *expr2*, 否则返回 *expr3*。IF() 根据使用它的上下文返回数或串。

IF(1,"true","false")	→ "true"
IF(0,"true","false")	→ "false"
IF(NULL,"true","false")	→ "false"
IF(1.3,"non-zero","zero")	→ "non-zero"
IF(0.3,"non-zero","zero")	→ "zero"
IF(0.3 != 0,"non-zero","zero")	→ "non-zero"

expr 是作为整数求值的, 最后的三个例子说明, 如果不仔细, 结果将会搞得你不知所措。

1.3 转换为整数值 1, 因此为真。但 0.3 转换为整数值 0, 所以为假。最后的例子示出了使用浮点数的恰当方式: 即应该用比较来使用浮点数, 因为比较根据测试的结果返回 1 或 0。

■ IFNULL(*expr1*, *expr2*)

如果表达式 *expr1* 为 NULL, 则返回 *expr2*; 否则返回 *expr1*。IFNULL() 根据使用它的上下文返回一个数或串。

IFNULL(NULL,"null")	→ "null"
IFNULL("not null","null")	→ "not null"

■ INTERVAL(*n*, *n1*, *n2*, ...)

如果 $n < n1$, 返回 0, 如果 $n < n2$, 返回 1, 从此类推。或者如果 *n* 为 NULL, 返回 -1。其中值 *n1*, *n2*, ... 必须严格递增 (即 $n1 < n2 < \dots$), 因为使用的是快速二分搜索法。否则, INTERVAL() 的结果将不可预测。

INTERVAL(1.1,0,1,2)	→ 2
INTERVAL(7,1,3,5,7,9)	→ 4

■ ISNULL(*expr*) 如果表达式 *expr* 的值为 NULL, 返回 1; 否则返回 0。

ISNULL(NULL)	→ 1
ISNULL(0)	→ 0
ISNULL(1)	→ 0

■ LEAST(*expr1*, *expr2*, ...) 返回最小的参数, 其中“最小”利用与 GREATEST() 函数类似的比较规则来定义。

LEAST(2,3,1)	→ 1
LEAST(38.5,94.2,-1)	→ -1.0
LEAST("a","ab","abc")	→ "a"

LEAST() 在 MySQL 3.22.5 中引入。在更早的版本中用的是 MIN()。

■ STRCMP(*str1*, *str2*) 如果 *str1* 与 *str2* 相同, 返回 1, 如果不同, 返回 0。如果两个参数中有一个为 NULL, 则返回 NULL。该比较是区分大小写的。

STRCMP("a","a")	→ 0
STRCMP("a","A")	→ 1

C.2.2 数值函数

如果出现错误，数值函数返回 NULL。例如，如果将超出取值范围或非法的参数传递给函数，则函数将返回 NULL。

■ ABS(x) 返回 x 的绝对值。

ABS(13.5)	→ 13.5
ABS(-13.5)	→ 13.5

■ ACOS(x) 返回 x 的反余弦值，如果 x 不在 -1 到 1 的取值范围内，返回 NULL。

ACOS(1)	→ 0.000000
ACOS(0)	→ 1.570796
ACOS(-1)	→ 3.141593

ACOS() 在 MySQL 3.21.8 中引入。

■ ASIN(x) 返回 x 的反正弦值，如果 x 不在 -1 到 1 的取值范围内，返回 NULL。

ASIN(1)	→ 1.570796
ASIN(0)	→ 0.000000
ASIN(-1)	→ -1.570796

ASIN() 在 MySQL 3.21.8 中引入。

■ ATAN(x) 返回 x 的反正切值，如果 x 不在 -1 到 1 的取值范围内，返回 NULL。

ATAN(1)	→ 0.785398
ATAN(0)	→ 0.000000
ATAN(-1)	→ -0.785398

ASIN() 在 MySQL 3.21.8 中引入。

■ ATAN2(x, y) 与 ATAN(x) 类似，但它根据参数的符号决定返回值的象限。

ATAN2(1,1)	→ 0.785398
ATAN2(1,-1)	→ 2.356194
ATAN2(-1,1)	→ -0.785398
ATAN2(-1,-1)	→ -2.356194

■ GEILING(x) 返回不小于 x 的最小整数。

CEILING(3.8)	→ 4
CEILING(-3.8)	→ -3

■ COS(x) 返回 x 的余弦值， x 以弧度表示。

COS(0)	→ 1.000000
COS(PI())	→ -1.000000
COS(PI()/2)	→ 0.000000

COS() 在 MySQL 3.21.8 中引入。

■ COT(x) 返回 x 的余切值， x 以弧度表示。

COT(PI()/2)	→ 0.00000000
COT(PI()/4)	→ 1.00000000

COT() 在 MySQL 3.21.16 中引入。

■ DEGREES(x) 返回 x 的从弧度转换为度的值。

DEGREES(PI())	→ 180
DEGREES(PI()*2)	→ 360
DEGREES(PI()/2)	→ 90

DEGREES(-PI())

→ -180

DEGREES() 在 MySQL 3.21.16 中引入。

■ EXP(x) 返回 e^x ，其中 e 为自然对数的底。

EXP(1)

→ 2.718282

EXP(2)

→ 7.389056

EXP(-1)

→ 0.367879

1/EXP(1)

→ 0.36787944

■ FLOOR(x) 返回不大于 x 的最大整数（即，截去 x 的小数部分）。

FLOOR(3.8)

→ 3

FLOOR(-3.8)

→ -4

■ LOG(x) 返回 x 的自然对数（底为 e ）。

LOG(0)

→ NULL

LOG(1)

→ 0.000000

LOG(2)

→ 0.693147

LOG(EXP(1))

→ 1.000000

可利用 LOG() 来计算底为任意值 b 的 x 的对数为 LOG(x)/LOG(b)。

LOG(100)/LOG(10)

→ 2.00000000

LOG10(100)

→ 2.000000

■ LOG10(x) 返回底为 10 的 x 的对数。

LOG10(0)

→ NULL

LOG10(10)

→ 1.000000

LOG10(100)

→ 2.000000

■ MOD(m , n) 与 $m \% n$ 相同，请参阅“算术运算符”

■ PI() 返回 π 值。

PI()

→ 3.141593

PI() 在 MySQL 3.21.8 中引入。

■ POW(x , y) 返回 x^y ，即 x 的 y 次幂。

POW(2,3)

→ 8.000000

POW(2,-3)

→ 0.125000

POW(4,.5)

→ 2.000000

POW(16,.25)

→ 2.000000

■ POWER(x , y) 这是 POW() 的同义词。在 MySQL 3.21.16 中引入。

■ RADIANS(x) 返回 x 的从角度转换为弧度的值。

RADIANS(0)

→ 0

RADIANS(360)

→ 6.28319

RADIANS(-360)

→ -6.28319

RADIANS() 在 MySQL 3.21.16 中引入。

■ RAND()

RAND(n)

RAND() 返回一个范围在 0.0 到 1.0 之间的浮点数随机值。RAND(n) 功能相同，但利用 n 69 作为随机数的种子值。所有相同 n 值的调用都返回相同的结果，在需要可重复的数字序列时，可利用这个特性。

RAND(10)

→ 0.181091

RAND(10)

→ 0.181091

RAND()	→ 0.117195
RAND()	→ 0.358596
RAND(10)	→ 0.181091

请注意例子中给出参数和不给参数的那些连续调用所得到的随机数。

■ ROUND(x)

ROUND(x,d)
ROUND(x)

返回 x 四舍五入为一个整数的值。ROUND(x , d) 返回 x 四舍五入为一个 d 位小数的值。如果 d 为 0, 结果中没有小数点或小数部分。

ROUND(15.3)	→ 15
ROUND(15.5)	→ 16
ROUND(-33.27834,2)	→ -33.28
ROUND(1,4)	→ 1.0000

■ SIGN(x) 根据 x 的值为负、零或正, 分别返回 -1、0 或 1。

SIGN(15.803)	→ 1
SIGN(0)	→ 0
SIGN(-99)	→ -1

■ SIN(x) 返回 x 的正弦值, x 以弧度表示。

SIN(0)	→ 0.000000
SIN(PI())	→ 0.000000
SIN(PI()/2)	→ 1.000000

■ SIN() 在 MySQL 3.21.8 中引入。

■ SQRT(x) 返回 x 的非负平方根。

SQRT(625)	→ 25.000000
SQRT(2.25)	→ 1.500000
SQRT(-1)	→ NULL

■ TAN(x) 返回以弧度表示的 x 的正切值。

TAN(0)	→ 0.000000
TAN(PI()/4)	→ 1.000000

TAN() 在 MySQL 3.21.8 中引入。

■ TRUNCATE(x , d) 返回将 x 截成小数部分为 d 位的值。如果 d 为 0, 结果无小数部分。如果 d 小于 x 中的小数位数, 则用零填充所需的宽度。

TRUNCATE(1.23,1)	→ 1.2
TRUNCATE(1.23,0)	→ 1
TRUNCATE(1.23,4)	→ 1.2300

C.2.3 串函数

本节中多数函数返回串结果。而有的函数, 如 LENGTH(), 以串作为参数, 但返回数作为结果。对于那些根据串的位置处理串的函数, 其第一个(最左)字符的位置为 1 而不是 0。

■ ASCII(str) 返回串 str 最左边字符的 ASCII 码。如果 str 为空, 则返回 0。如果 str 为 NULL 则返回 NULL。

ASCII("abc")	→ 97
ASCII("")	→ 0

ASCII(NULL)

→ NULL

ASCII() 函数是在 MySQL 3.21.2 中引入的。

■ BIN(*n*) 以二进制串的形式返回 *n* 的值。下列两个例子是等价的：

BIN(*n*)CONV(*n*,10,2)

详细信息请参阅 CONV()。

■ CHAR(*n*₁, *n*₂, ...) 将各参数作为 ASCII 代码，返回各代码所代表字符连接成的字符串。忽略 NULL 参数。

CHAR(65)

→ "A"

CHAR(97)

→ "a"

CHAR(89,105,107,101,115,33)

→ "Yikes!"

CHAR() 函数是在 MySQL 3.21.0 中引入的。

■ CHARACTER_LENGTH(*str*) 此函数是 LENGTH() 的同义词。

■ COALESCE(*expr*₁, *expr*₂, ...) 返回列表 (*expr*₁, *expr*₂, ...) 中的第一个非空元素。

COALESCE(NULL,1/0,2,"a",45+97)

→ "2"

COALESCE() 是在 MySQL 3.23.3 中引入的。

■ CONCAT(*str*₁, *str*₂, ...) 返回由其所有参数连接而成的串。如果任一参数为 NULL，则返回 NULL。CONCAT() 可接受几个参数。

CONCAT("abc","def")

→ "abcdef"

CONCAT("abc")

→ "abc"

CONCAT("abc",NULL)

→ NULL

CONCAT("Hello",",",", "goodbye")

→ "Hello, goodbye"

■ CONV(*n*, *from_base*, *to_base*) 给出一个以 *from_base* 为基数的数 *n*，返回一个以 *to_base* 为基数表示的 *n* 的串。如果任一参数为 NULL，则结果为 NULL。*from_base* 与 *to_base* 应该是范围为2到36的整数。N 按 BIGINT 处理（64 位整数），但可能以串来指定，因为基数大于 10 的数可能包括非十进制数（这也是 CONV() 返回一个串的原因。对于基数为 11 到 36，其结果可能包含字符“A”到“Z”）。如果 *n* 在基数为 *from_base* 的数中不是一个合法的数，则结果为 0（例如，如果 *from_base* 为 16，而 *n* 为“abcdefg”，则结果为 0，因为“g”不是一个合法的十六进制数字）。

n 中的非十进制数字既可用大写也可以用小写表示。结果中的非十进制数字都以大写表示。缺省设置时，*n* 处理为一个无符号数。

将十六进制表示的14转换为二进制如下：

CONV("e",16,2)

→ "1110"

将二进制表示的255转换为八进制如下：

CONV(11111111,2,8)

→ "377"

CONV("11111111",2,8)

→ "377"

CONV() 是在 MySQL 3.22.4 中引入的。

■ ELT(*n*, *str*₁, *str*₂, ...) 从串 *str*₁、*str*₂、... 中返回第 *n* 个串。如果没有 *n* 个串，或第 *n* 个串为 NULL 或 *n* 为 NULL，则返回 NULL。第二个串的下标为1。ELT() 与 FIELD() 互为反函数。

ELT(3,"a","b","c","d","e")

→ "c"

ELT(0,"a","b","c","d","e")

→ NULL


```
ELT(6, "a", "b", "c", "d", "e")          → NULL.
ELT(FIELD("b", "a", "b", "c"), "a", "b", "c") → "b"
```

■ **EXPORT_SET**(*n*, *on*, *off*, [*separator*, [*bit_count*]]) 返回由串 *on* 和 *off* 组成以串 *separator* 分隔的串。*on* 用于 *n* 中置 1 的每一位, *off* 用于 *n* 中置 0 的每一位。*bit_count* 表示 *n* 中测试的最大位数。缺省的 *separator* 串为逗号, 缺省的 *bit_count* 为 64。

```
EXPORT_SET(7, "-", ".", "", 5)          → "++-.-"
EXPORT_SET(0xa, "1", "0", "", 6)         → "010100"
EXPORT_SET(97, "Y", "N", "", 8)          → "Y,N,N,N,N,Y,Y,N"
```

EXPORT_SET() 在 MySQL 3.23.2 中引入。

■ **FIELD**(*str*, *str1*, *str2*, ...)

在串列表 *str1*、*str2*、... 中查找 *str*, 返回其位置。如果无匹配串或 *str* 为 NULL, 则返回 0。第一个串的位置为 1。**FIELD()** 与 **ELT()** 互为反函数。

```
FIELD("b", "a", "b", "c")                → 2
FIELD("d", "a", "b", "c")                → 0
FIELD(NULL, "a", "b", "c")               → 0
FIELD(ELT(2, "a", "b", "c"), "a", "b", "c") → 2
```

■ **FIND_IN_SET**(*str*, *str_list*)

str_list 为一个包含以逗号分隔的子串的串 (即, 它类似一个 SET 值)。**FIND_IN_SET()** 返回 *str* 在 *str_list* 中的位置。如果 *str* 未出现在 *str_list* 中, 则返回 0。如果任一参数为 NULL, 则返回 NULL。第一个子串的位置为 1。

```
FIND_IN_SET("cow", "moose,cow,pig")       → 2
FIND_IN_SET("dog", "moose,cow,pig")       → 0
```

FIND_IN_SET() 在 MySQL 3.21.22 中引入。

■ **FORMAT**(*X*, *D*)

将数 *X* 转换为如 “nn,nn.nnn” 格式的带有 *D* 位十进制小数的串。如果 *D* 为 0, 则结果无小数部分。

```
FORMAT(1234.56789, 3)                    → "1,234.568"
FORMAT(999999.99, 2)                     → "999,999.99"
FORMAT(999999.99, 0)                     → "1,000,000"
```

注意最后一个例子中的四舍五入。

■ **HEX**(*n*)

以十六进制串形式返回 *n* 的值。下面两个表达式是等价的：

```
HEX(n)
CONV(n, 10, 16)
```

更多的信息请参阅 **CONV()** 的说明。

HEX() 是在 MySQL 3.22.4 中引入的。

■ **INSERT**(*str*, *pos*, *len*, *new_str*) 返回串 *str*, 其中用串 *new_str* 替换从 *pos* 开始 *len* 个字符的子串。如果 *pos* 超出范围则返回原串, 如果任一参数为 NULL, 则返回 NULL。

```
INSERT("nighttime", 6, 4, "fall")          → "nightfall"
INSERT("sunshine", 1, 3, "rain or ")       → "rain or shine"
INSERT("sunshine", 0, 3, "rain or ")       → "sunshine"
```

■ **INSTR**(*str*, *substr*) **INSTR()** 与 **LOCATE()** 的两参数形式相似, 只是参数颠倒了。下面

两个表达式是等价的：

```
INSTR(str, substr)
LOCATE(substr, str)
```

- **LCASE(str)** 返回串 *str*，其中所有字符转换为小写字符。如果 *str* 为 NULL，则返回 NULL。

```
LCASE("New York, NY")      → "new york, ny"
LCASE(NULL)                 → NULL
```

- **LEFT(str, len)** 返回串 *str* 中最左边的 *len* 个字符，如果没有 *len* 那么多个字符，则返回整个串。如果 *str* 为 NULL，则返回 NULL。如果 *len* 为 NULL，或小于 1，则返回空串。

```
LEFT("my left foot", 2)     → "my"
LEFT(NULL, 10)              → NULL
LEFT("abc", NULL)           → ""
LEFT("abc", 0)              → ""
```

- **LENGTH(str)** 返回串 *str* 的长度。

```
LENGTH("abc")               → 3
LENGTH("")                  → 0
LENGTH(NULL)                → NULL
```

- **LOCATE(substr, str)**

```
LOCATE (substr, str, pos)
```

LOCATE() 的两参数形式返回串 *substr* 在 *str* 中第一次出现的位置，如果 *substr* 未出现在 *str* 中，则返回 0。如果任一参数为 NULL，则返回 NULL。如果给出位置参数 *pos*，LOCATE() 从 *pos* 开始查找 *substr*。查找是区分大小写的。

```
LOCATE("b", "abc")          → 2
LOCATE("b", "ABC")          → 0
```

- **LOWER(str)** 此函数为 LCASE() 的同义词。

- **LPAD(str, len, pad_str)** 返回一个串，此串由串 *str* 左边加上数个 *pad_str* 串使其长度达到 *len* 个字符。如果 *str* 的长度已经是 *len*，则返回 *str*。

```
LPAD("abc", 12, "def")      → "defdefdefabc"
LPAD("abc", 10, ".")        → ".....abc"
LPAD("abc", 2, ".")         → "abc"
```

LPAD() 是在 MySQL 3.22.2 中引入的。

- **LTRIM(str)** 返回删除了左边空格的 *str* 串，如果 *str* 为 NULL，则返回 NULL。

```
LTRIM(" abc ")              → "abc "
```

- **MAKE_SET(n, bit0_str, bit1_str, ...)** 基于整数 *n* 和串 *bit0_str*、*bit1_str*、...，构造 SET 值（由逗号分隔的子串组成的一个串）。对于 *n* 值中置 1 的每位，相应的串包括在结果中（如果位 0 置 1，结果将包括 *bit0_str*，如此等等）。如果 *n* 为 0，则结果为空串。如果 *n* 为 NULL，则结果为 NULL。若串列表中任一参数为 NULL，则构造结果串时将其忽略。

```
MAKE_SET(8, "a", "b", "c", "d", "e") → "d"
MAKE_SET(7, "a", "b", "c", "d", "e") → "a,b,c"
MAKE_SET(2+16, "a", "b", "c", "d", "e") → "b,e"
MAKE_SET(2|16, "a", "b", "c", "d", "e") → "b,e"
```

`MAKE_SET(-1,"a","b","c","d","e")` → "a,b,c,d,e"

最后一个例子选择了每个串，因为 -1 使所有位都置 1。

`MAKE_SET()` 是在 MySQL 3.22.2 中引入的。

■ `MID(str, pos, len)` 返回串 `str` 中从 `pos` 开始长度为 `len` 的子串。如果任一参数为 `NULL`，则返回 `NULL`。

`MID("what a dull example",8,4)` → "dull"

■ `OCT(n)` 以八进制形式的串返回 `n` 的值。下面列两个表达式是等价的：

`OCT(n)`

`CONV(n,10,8)`

详细信息请参阅 `CON()` 的介绍。

`OCT()` 是在 MySQL 3.22.4 中引入的。

■ `OCTET_LENGTH(str)` 此函数为 `LENGTH()` 的同义词。

■ `POSITION(substr IN str)` 此函数与 `LOCATE()` 的两参数形式类似。下面的表达式是等价的：

`POSITION(substr IN str)`

`LOCATE(substr,str)`

■ `REPEAT(str, n)` 返回由串 `str` 重复 `n` 次所得的串。如果 `n` 不是正数或任一参数为 `NULL`，则返回空串。

`REPEAT("x",10)`

→ "xxxxxxxxxx"

`REPEAT("abc",3)`

→ "abcabcabc"

`REPEAT()` 是在 MySQL 3.21.10 中引入的。

■ `REPLACE(str, from_str, to_str)` 返回一个串，此串为将 `str` 中所有子串 `from_str` 替换成子串 `to_str` 所得的串。如果任一参数为 `NULL`，则返回 `NULL`。如果 `to_str` 为空，则结果是删除 `from_str`。如果 `from_str` 为空，则 `REPLACE()` 返回 `str` 不变。

`REPLACE("abracadabra","a","oh")`

→ "ohbrohcohdohbroh"

`REPLACE("abracadabra","a","")`

→ "brcdbr"

`REPLACE("abracadabra","", "x")`

→ "abracadabra"

■ `REVERSE(str)` 返回一个颠倒串 `str` 中所有字符得到的串。如果 `str` 为 `NULL`，则返回 `NULL`。

`REVERSE("abracadabra")`

→ "arbadacarba"

`REVERSE("Madam, I'm Adam")`

→ "madA m'I ,madaM"

`REVERSE()` 是在 MySQL 3.21.19 中引入的。

■ `RIGHT(str, len)` 返回串 `str` 中最右边长度为 `len` 个字符的子串。如果没有 `len` 个字符，则返回整个串。如果 `str` 为 `NULL`，则返回 `NULL`。如果 `len` 为 `NULL` 或小于 1，则返回空串。

`RIGHT("rightmost",4)`

→ "most"

■ `RPAD(str, len, pad_str)` 返回由串 `str` 右边补上数个 `pad_str` 串以达到 `len` 个字符长的串。如果 `str` 已包含 `len` 个字符，则返回 `str`。

`RPAD("abc",12,"def")`

→ "abcdefdefdef"

`RPAD("abc",10,".")`

→ "abc....."

`RPAD("abc",2,".")`

→ "abc"

`RPAD()` 是在 MySQL 3.22.2 中引入的。

■ **RTRIM(str)** 返回删除了右边空格的 *str* 串, 如果 *str* 为空, 则返回空。

RTRIM(" abc ") → " abc"

■ **SOUNDEX(str)** 返回由串 *str* 计算出来的声音串。忽略 *str* 中的非字母数字字符。超出范围“ A ”到“ Z ”以外的字符作为元音处理。

SOUNDEX("Cow") → "C000"
 SOUNDEX("Cowl") → "C400"
 SOUNDEX("Howl") → "H400"
 SOUNDEX("Hello") → "H400"

■ **SPACE(n)** 返回由 *n* 个空格组成的串, 如果 *n* 非负, 则为空集, 如果 *n* 为 NULL, 则为 NULL。

SPACE(6) → " "
 SPACE(0) → ""
 SPACE(NULL) → NULL

SPACE() 是在 MySQL 3.21.16 中引入。

■ **SUBSTRING(str, pos)**

SUBSTRING(str, pos, len)
 SUBSTRING(str FROM pos)
 SUBSTRING(str FROM pos FOR len)

返回串 *str* 中从 *pos* 开始的一个子串。如果给出 *len* 参数, 则返回该长度的一个子串, 否则返回 *str* 中以 *pos* 开始的右边的子串。

SUBSTRING("abcdef", 3) → "cdef"
 SUBSTRING("abcdef", 3, 2) → "cd"

下列表达式是等价的：

SUBSTRING(str, pos, len)
 SUBSTRING(str FROM pos FOR len)
 MID(str, pos, len)

■ **SUBSTRING_INDEX(str, delim, n)** 返回串 *str* 中的一个子串。如果 *n* 为正数, SUBSTRING_INDEX() 查找 *delim* 分隔串的第 *n* 次出现, 然后返回该分隔符左边的所有字符。如果 *n* 为负, 则从串 *str* 的右边向左计数, 查找 *delim* 的第 *n* 次出现, 然后返回该分隔符右边的所有字符。如果在 *str* 中未找到 *delim*, 则返回整个串。如果任意参数为 NULL, 则返回 NULL。

SUBSTRING_INDEX("jar.jar", "j", -2) → "ar.jar"
 SUBSTRING_INDEX(USER(), "@", 1) → "paul"

SUBSTRING_INDEX() 是在 MySQL 3.21.15 中引入的。

■ **TRIM([LEADING | TRAILING | BOTH] [trim_str] FROM str)** 从串 *str* 中去掉前导的和/或后跟的串 *trim_str*。如果指定 LEADING, 则去掉前导的 *trim_str*。如果指定 TRAILING, 则去掉后跟的 *trim_str*。如果指定 BOTH, 则去掉前导的和后跟的 *trim_str*。如果 LEADING、TRAILING 或 BOTH 都不给出, 缺省为 BOTH。如果不给出 *trim_str*, 则 TRIM() 去掉空格。

TRIM("^^" FROM "^^^XYZ^^") → "XYZ"
 TRIM(LEADING "^^" FROM "^^^XYZ^^") → "XYZ^^"
 TRIM(TRAILING "^^" FROM "^^^XYZ^^") → "^^^XYZ"
 TRIM(BOTH "^^" FROM "^^^XYZ^^") → "XYZ"

```
TRIM(BOTH FROM " ABC ") → "ABC"
TRIM(" ABC ") → "ABC"
```

TRIM() 是在 MySQL 3.21.12 中引入的。

■ UCASE(str) 将串 *str* 的所有字符转换为大写。如果 *str* 为 NULL, 则返回 NULL。

```
UCASE("New York, NY") → "NEW YORK, NY"
UCASE(NULL) → NULL
```

■ UPPER(str) 此函数为 UCASE() 的同义词。

C.2.4 日期和时间函数

日期和时间函数取各种形式的参数。一般情况下, 需要一个 DATE 参数的函数也接受 DATETIME 或 TIMESTAMP 参数并且忽略值的时间部分。

■ ADDDATE(date, INTERVAL *expr interval*) 此函数为 DATE_ADD() 的同义词。

■ CURDATE() 以 “YYYY-MM-DD” 格式返回一个串作为当前时间, 根据使用环境不同, 也可以返回 YYYYMMDD 格式的数值作为当前日期。

```
CURDATE() → "1999-08-10"
CURDATE() + 0 → 19990810
```

合法的日期和时间值

如果未向日期和时间函数提供合法的日期和时间值, 就不可能得到正确的结果。因此应该对参数进行仔细检查。

■ CURRENT_DATE 此函数是 CURDATE() 的同义词, 注意没有圆括号。

■ CURRENT_TIME 此函数为 CURTIME() 的同义词, 注意没有圆括号。

■ CURRENT_TIMESTAMP 此函数为 NOW() 的同义词, 注意没有圆括号。

■ CURTIME() 以 “hh:mm:ss” 格式的串返回当前时间, 根据上下文不同, 也可以 hhmmss 格式的数返回当前时间。

```
CURTIME() → "16:41:01"
CURTIME() + 0 → 164101
```

CURTIME() 是在 MySQL 3.21.12 中引入的。

■ DATE_ADD(date, INTERVAL *expr interval*) 给日期或日期和时间值 *date* 加上一个时间间隔, 然后返回其结果。 *expr* 指定要增加 (如果 *expr* 以 ‘-’ 号开始, 则为减去) 到 *date* 的时间值, 而 *interval* 指定 *expr* 的类型。如果 *date* 为 DATE 值, 则结果为 DATE 值, 在结果的计算中不涉及与时间有关的值, 否则, 结果是一个 DATETIME 的值。如果 *date* 不是一个合法的日期, 则返回 NULL。

```
DATE_ADD("1999-12-01", INTERVAL 1 YEAR) → "2000-12-01"
DATE_ADD("1999-12-01", INTERVAL 60 DAY) → "2000-01-30"
DATE_ADD("1999-12-01", INTERVAL -3 MONTH) → "1999-09-01"
DATE_ADD("1999-12-01 08:30:00", INTERVAL 12 HOUR) → "1999-12-01 20:30:00"
```

表C-2给出了允许的 *interval* 值、其意义以及每种间隔类型的格式。关键字 INTERVAL 和 *interval* 说明符可以大写或小写字符给出。

增加到日期的表达式 *expr* 可用数或串的形式指定, 如果包含非数字字符, 则必须以串的形式给出。分隔字符可以任意, 有可能为任一标点符号:

```
DATE_ADD("1999-12-01",INTERVAL "2:3" YEAR_MONTH) → "2002-03-01"
```

```
DATE_ADD("1999-12-01",INTERVAL "2-3" YEAR_MONTH) → "2002-03-01"
```

*expr*值的各部分根据 *interval*说明符所期望的部分从右至左进行匹配。例如， *HOURL_SECONDS* 期望的格式为 “*hh:mm:ss*”。*expr* 值为 “15:21” 解释为 “00:15:21” 而不是 “15:21:00”。

```
DATE_ADD("1999-12-01 12:00:00",INTERVAL "15:21" HOUR_SECOND)
→ "1999-12-01 12:15:21"
```

表C-2 DATE_ADD() 间隔类型

类 型	说 明	值的格式
SECOND	秒	<i>ss</i>
MINUTE	分	<i>mm</i>
HOUR	时	<i>hh</i>
DAY	日	<i>DD</i>
MONTH	月	<i>MM</i>
YEAR	年	<i>YY</i>
MINUTE_SECOND	分和秒	“ <i>mm:ss</i> ”
HOUR_MINUTE	时和分	“ <i>hh:mm</i> ”
HOUR_SECOND	时、分、秒	“ <i>hh:mm:ss</i> ”
DAY_HOUR	天和小时	“ <i>DD hh</i> ”
DAY_MINUTE	天、小时和分	“ <i>DD hh:mm</i> ”
DAY_SECOND	天、小时、分和秒	“ <i>DD hh:mm:ss</i> ”
YEAR_MONTH	年和月	“ <i>YY-MM</i> ”

如果 *interval* 为 YEAR、MONTH 或 YEAR_MONTH 并且结果中天的部分大于结果月份中的天数，则天数设置为该月中最大的天数。

```
DATE_ADD("1999-12-31",INTERVAL 2 MONTH) → "2000-02-29"
```

DATE_ADD 是在 MySQL 3.22.4 中引入的。

■ DATE_FORMAT(*date*, *format*) 根据格式串 *format* 格式化日期或日期和时间值 *date*，返回结果串。可用 DATE_FORMAT() 来格式化 DATE 或 DATETIME 值，以便得到所希望的格式。

```
DATE_FORMAT("1999-12-01","%M %e, %Y") → "December 1, 1999"
```

```
DATE_FORMAT("1999-12-01","The %D of %M") → "The 1st of December"
```

表C-3给出了可用于格式串中的说明符。

自MySQL 3.23.0 以来，在格式代码前要求 “%” 字符。在更早的版本中，允许 “%” 出现，但它是可选的。出现在格式串中但未列在表中的字符被逐字地拷贝到结果串中。

如果对 DATE 值引用时间说明符，则该值的时间部分处理为 “00:00:00”。

```
DATE_FORMAT("1999-12-01","%i") → "00"
```

DATE_FORMAT() 是在 MySQL 3.21.14 中引入的。

■ DATE_SUB(*date*, INTERVAL *expr interval*) 以与 DATE_ADD() 相同的方式求出日期数值，只是此函数从 *date* 中减去 *expr* 而非加上 *expr*。详细请参阅 DATE_ADD()。

```
DATE_SUB("1999-12-01",INTERVAL 1 MONTH) → "1999-11-01"
```

```
DATE_SUB("1999-12-01",INTERVAL "13-2" YEAR_MONTH) → "1986-10-01"
```

```
DATE_SUB("1999-12-01 04:53:12",INTERVAL "13-2" MINUTE_SECOND)
```


DATE_SUB("1999-12-01 04:53:12",INTERVAL "13-2" HOUR_MINUTE)
→ "1999-12-01 04:40:10"
→ "1999-11-30 15:51:12"

DATE_SUB() 是在 MySQL 3.22.4 中引入的。

表C-3 DATE_FORMAT() 格式说明符

说 明 符	说 明
%S, %s	两位数字形式的秒 (00,01, ..., 59)
%i	两位数字形式的分 (00,01, ..., 59)
%H	两位数字形式的小时, 24 小时 (00,01, ..., 23)
%h, %I	两位数字形式的小时, 12 小时 (01,02, ..., 12)
%k	数字形式的小时, 24 小时 (0,1, ..., 23)
%l	数字形式的小时, 12 小时 (1, 2, ..., 12)
%T	24 小时的时间形式 (hh:mm:ss)
%r	12 小时的时间形式 (hh:mm:ss AM 或 hh:mm:ss PM)
%p	AM 或 PM
%W	一周中每一天的名称 (Sunday, Monday, ..., Saturday)
%a	一周中每一天名称的缩写 (Sun, Mon, ..., Sat)
%d	两位数字表示月中的天数 (00, 01, ..., 31)
%e	数字形式表示月中的天数 (1, 2, ..., 31)
%D	英文后缀表示月中的天数 (1st, 2nd, 3rd, ...)
%w	以数字形式表示周中的天数 (0=Sunday, 1=Monday, ..., 6=Saturday)
%j	以三位数字表示年中的天数 (001, 002, ..., 366)
%U	周 (0, 1, 52), 其中 Sunday 为周中的第一天
%u	周 (0, 1, 52), 其中 Monday 为周中的第一天
%M	月名 (January, February, ..., December)
%b	缩写的月名 (January, February, ..., December)
%m	两位数字表示的月份 (01, 02, ..., 12)
%c	数字表示的月份 (1, 2, ..., 12)
%Y	四位数字表示的年份
%y	两位数字表示的年份
%%	直接值 “ % ”

■ DAYNAME(date) 返回包含日期值 date 的周日名串。

DAYNAME("1999-12-01") → "Wednesday"
DAYNAME("1900-12-01") → "Saturday"

DAYNAME() 是在 MySQL 3.21.23 中引入的。

■ DAYOFMONTH(date) 返回日期值 date 在月中的天数的数值, 范围从 1 到 31。

DAYOFMONTH("1999-12-01") → 1
DAYOFMONTH("1999-12-25") → 25

DAYMONTH() 是在 MySQL 3.21.22 中引入的。

■ DAYOFWEEK(date) 返回日期值 date 在每周中的天数的数值, 按 ODBC 标准星期天为1, 星期二为 2, ..., 星期六为 7。可参阅 WEEKDAY() 函数。

DAYOFWEEK("1999-12-05") → 1
DAYNAME("1999-12-05") → "Sunday"
DAYOFWEEK("1999-12-11") → 7
DAYNAME("1999-12-11") → "Saturday"

DAYWEEK() 是在 MySQL 3.21.15 中引入的。

■ DAYOFYEAR(*date*) 返回日期值 *date* 在年中的天数的数值, 范围从 1 到 366。

```
DAYOFYEAR("1999-12-01")      → 335
DAYOFYEAR("2000-12-31")      → 366
```

DAYOFYEAR() 是在 MySQL 3.21.22 中引入的。

■ EXTRACT(*interval* FROM *datetime*) 返回由 *interval* 表示的日期和时间值 *datetime* 的组成部分, *interval* 可以是 DATE_ADD() 所允许的任一间隔说明符。

```
EXTRACT(YEAR FROM "1999-12-01 13:42:19")      → 1999
EXTRACT(MONTH FROM "1999-12-01 13:42:19")     → 12
EXTRACT(DAY FROM "1999-12-01 13:42:19")       → 1
EXTRACT(HOUR_MINUTE FROM "1999-12-01 13:42:19") → 1342
EXTRACT(SECOND FROM "1999-12-01 13:42:19")    → 19
```

EXTRACT() 可用于 DATE 值, 这时若提取时间将得到 NULL。

EXTRACT() 是在 MySQL 3.23.0 中引入的。

■ FROM_DAYS(*n*) 给此函数一个表示自 0 年以来的日期数的数值 (一般由调用 TO_DAYS() 得到), 返回相应的日期。

```
TO_DAYS("1999-12-01")      → 730454
FROM_DAYS(730454 + 3)      → "1999-12-04"
```

FROM_DAYS() 只打算用于公历 (1582 年以来) 所包含的日期。

■ FROM_UNIXTIME(*unix_timestamp*)

FROM_UNIXTIME(*unix_timestamp*, *format*)

给此函数一个 UNIX 时间戳值 *unix_timestamp* (如由 UNIX_TIMESTAMP() 所返回的值), 返回一个以 “YYYY-MM-DD hh:mm:ss” 格式表示的日期和时间值, 或者根据使用环境不同返回一个 YYYYMMDDhhmmss 格式的数。如果给出 *format* 参数, 则返回值被格式化为一个串, 这与 DATE_FORMAT() 函数所做的一样。

```
FROM_UNIXTIME(934340541)      → "1999-08-10 22:02:21"
FROM_UNIXTIME(944028000)      → "1999-12-01 00:00:00"
FROM_UNIXTIME(944028000, "%Y") → "1999"
```

FROM_UNIXTIME() 是在 MySQL 3.21.5 中引入的。而两个参数的形式是在 MySQL 3.21.8 中引入的。

■ HOUR(*time*) 返回时间值 *time* 中小时的值, 范围为 0 到 23。

```
HOUR("12:31:58")      → 12
```

HOUR() 是在 MySQL 3.21.22 中引入的。

■ MINUTE(*time*) 返回时间值 *time* 中分的值, 范围为 0 到 59。

```
MINUTE("12:31:58")      → 31
```

MINUTE() 是在 MySQL 3.21.22 中引入的。

■ MONTH(*date*) 返回日期值 *date* 中月份的值, 范围为 1 到 12。

```
MONTH("1999-12-01")      → 12
```

MONTH() 是在 MySQL 3.21.22 中引入的。

■ MONTHNAME(*date*) 返回包含日期值 *date* 中的月份名的串。

```
MONTHNAME("1999-12-01")      → "December"
```

MONTHNAME() 是在 MySQL 3.21.23 中引入的。

- NOW() 返回 “YYYY-MM-DD hh:mm:ss” 格式的当前日期和时间串, 或者根据不同的上下文返回 YYYYMMDDhhmmss 格式的当前日期和时间的数值。

```
NOW() → "1999-08-10 18:51:43"
NOW() + 0 → 19990810185143
```

- PERIOD_ADD(period, n) 给时期值 period 加上 n 个月, 返回其结果。返回值的格式为 YYYYMM。而 period 参数可能是 YYYYMM 或 YYMM。两者都不是日期值。

```
PERIOD_ADD(199902, 12) → 200002
PERIOD_ADD(9902, -3) → 199811
```

- PERIOD_DIFF(period1, period2) 求周期值参数之差, 返回相差的月份数。两参数的格式可以是 YYYYMM 或 YYMM。两者都不是日期值。

```
PERIOD_DIFF(200002, 199902) → 12
PERIOD_DIFF(199811, 9902) → -3
```

- QUARTER(date) 返回日期值 date 中当年的季度值, 范围为 1 到 4。

```
QUARTER("1999-12-01") → 4
QUARTER("2000-01-01") → 1
```

QUARTER() 是在 MySQL 3.21.22 中引入的。

- SECOND(time) 返回时间值 time 中的秒数, 范围从 0 到 59。

```
SECOND("12:31:58") → 58
SECOND(123158) → 58
```

SECOND() 是在 MySQL 3.21.22 中引入的。

- SEC_TO_TIME(seconds) 给出秒数 seconds, 返回 “hh:mm:ss” 格式的时间串, 或根据上下文不同返回 hhmmss 格式的数。

```
SEC_TO_TIME(29834) → "08:17:14"
SEC_TO_TIME(29834) + 0 → 81714
```

SEC_TO_TIME() 是在 MySQL 3.21.5 中引入的。

- SUBDATE(date, INTERVAL expr interval) 此函数为 DATE_SUB() 的同义词。

- SYSDATE() 此函数为 NOW() 的同义词。

- TIME_FORMAT(time, format) 根据格式串 format 格式化时间值 time, 返回结果串。这里的格式化串与 DATE_FORMAT() 函数所用的相同, 但使用的只是那些与时间有关的说明符。其他说明产生 NULL 值或 0。

```
TIME_FORMAT("12:31:58", "%H %i") → "12 31"
TIME_FORMAT(123158, "%H %i") → "12 31"
```

TIME_FORMAT() 是在 MySQL 3.21.3 中引入的。

- TIME_TO_SEC(time) 给出一个表示所用去的时间值 time, 给出相应的秒数。

```
TIME_TO_SEC("08:17:14") → 29834
```

TIME_TO_SEC() 是在 MySQL 3.21.16 中引入的。

- TO_DAYS(date) 返回将日期值 date 转换为自 0 年以来的天数。此值可传递给 FROM_DAYS() 转换为一个日期值。

```
TO_DAYS("1999-12-01")           → 730454
FROM_DAYS(730454 - 365)          → "1998-12-01"
```

TODAYS() 只打算用于公历（1582 年以来）所包含的日期。

■ UNIX_TIMESTAMP()

UNIX_TIMESTAMP(date)

无参数调用时，返回自 UNIX 时间点（“1970-01-01 00:00:00” GMT）以来的秒数。在用日期值参数 *date* 调用时，返回 UNIX 时间点到该日期期间的秒数，日期 *date* 可以几种方式给出，这些方式分别为：DATE 或 DATETIME 串、TIMESTAMP 值，或本地时间的 YYYYMMDD 或 YYMMDD 格式的数。

```
UNIX_TIMESTAMP()                → 934341073
UNIX_TIMESTAMP("1999-12-01")    → 944028000
UNIX_TIMESTAMP(991201)          → 944028000
```

■ WEEK(date)

WEEK(date, first)

在用单个参数调用时，返回日期值 *date* 在当年中的星期数，其范围从 0 到 52。假定一周从星期天开始。在用两个参数调用时，返回相同的值，但 *first* 参数指出了一周从哪天开始。如果 *first* 为 0，则一周从星期天开始。如果 *first* 为 1，则一周从星期一开始。

```
WEEK("1999-12-05")              → 49
WEEK("1999-12-05", 0)           → 49
WEEK("1999-12-05", 1)           → 48
```

WEEK() 是在 MySQL 3.21.22 中引入的。其两参数的形式是在 MySQL 3.22.1 中引入的。

■ WEEKDAY(date) 返回日期值 *date* 为星期几的数值。星期几的数值从星期一的 0 到星期天的 6，可参阅 DAYOFWEEK() 函数。

```
WEEKDAY("1999-12-05")           → 6
DAYNAME("1999-12-05")           → "Sunday"
WEEKDAY("1999-12-13")           → 0
DAYNAME("1999-12-13")           → "Monday"
```

■ YEAR(date) 返回日期值 *date* 中的年份，其范围从 1000 到 9999。

```
YEAR("1999-12-01")              → 1999
```

YEAR() 是在 MySQL 3.21.22 中引入的。

C.2.5 汇总函数

汇总函数（summary function）也称为聚合函数（aggregate function）。它们根据一组值求出一个值。但结果值只根据选定行中非 NULL 值进行计算（也有例外，那就是 COUNT(*) 对所有的行进行计数）。汇总函数可用来汇总所有的列。或者在由其他列或列组合中的明确的值分组时，汇总多个值。请参阅第 1 章中的“生成汇总”一节。

对于该节中的例子，假定有一个表 *my_table*，含有一个整数列 *my_col*，该列有六行，其值分别为 1、3、5、7、9 和 NULL。

■ AVG(expr) 返回所选择行中所有非 NULL 值的 *expr* 的平均值。

```
SELECT AVG(my_col) FROM my_table → 5.0000
SELECT AVG(my_col)*2 FROM my_table → 10.0000
SELECT AVG(my_col*2) FROM my_table → 10.0000
```

■ **BIT_AND(*expr*)** 返回所选择行中所有非 NULL 值的 *expr* 的按位 AND 值。

```
SELECT BIT_AND(my_col) FROM my_table → 1
```

BIT_AND() 是在 MySQL 3.21.11 中引入的。

■ **BIT_OR(*expr*)** 返回所选择行中所有非 NULL 值的 *expr* 的按位 OR 值。

```
SELECT BIT_OR(my_col) FROM my_table → 15
```

BIT_OR() 是在 MySQL 3.21.11 中引入的。

■ **COUNT(*expr*)** 对于除 “*” 以外的任何参数，返回所选择集合中非 NULL 值的数的计数。对于参数 “*”，返回选择集合中所有行的计数，不管行中是否包含 NULL 值。

```
SELECT COUNT(my_col) FROM my_table → 5
SELECT COUNT(*) FROM my_table → 6
```

无 WHERE 子句的 COUNT(*) 是优化的，能非常快地返回 FROM 子句中指定的表中的记录数。在指定不止一个表时，COUNT(*) 返回各表中行数之积：

```
SELECT COUNT(*) FROM my_table AS m1, my_table AS m2
→ 36
```

■ **MAX(*expr*)** 返回所选择的行中所有非 NULL 值的 *expr* 的最大值。MAX() 也可以用于串，此时它返回字典序的最大值。

■ **MIN(*expr*)** 返回所选择的行中所有非 NULL 值的 *expr* 的最小值。MIN() 也可以用于串，此时它返回字典序的最小值。

```
SELECT MAX(my_col) FROM my_table → 9
```

```
SELECT MIN(my_col) FROM my_table → 1
```

■ **STD(*expr*)** 返回所选择行中所有非 NULL 值的 *expr* 的标准偏差。

```
SELECT STD(my_col) FROM my_table → 2.8284
```

■ **STDDEV(*expr*)** 此函数为 STD() 的同义词。

■ **SUM(*expr*)** 返回所选择行中所有非 NULL 值的 *expr* 之和。

```
SELECT SUM(my_col) FROM my_table → 25
```

C.2.6 杂项函数

本节中函数不属于上述所有类别。

■ **BENCHMARK(*count*, *expr*)** 重复求表达式 *expr* 的值 *count* 次。BENCHMARK() 是一个有些不寻常的函数，它是为在客户机程序中使用而设计的。它返回的值总是 0，因此返回的这个值没什么用途。其价值在于 mysql 显示查询结果后打印的占用时间：

```
mysql> SELECT BENCHMARK(100000,PASSWORD("secret"));
+-----+
| BENCHMARK(100000,PASSWORD("secret")) |
+-----+
| 0 |
+-----+
1 row in set (1.63 sec)
```

这里的时间只是服务器对表达式求值有多快的一个大致的估计，因为它表示的是客户机上的背景时间，而不是服务器上的 CPU 时间。这个时间受诸如服务器上的负载、此查询到达

时服务器是否处于可运行状态或换出等因素的影响。可以多次执行以观察代表性的值是什么。

BENCHMARK() 是在 MySQL 3.22.15 中引入的。

■ BIT_COUNT(*n*) 返回参数中置1的位数，返回值按 BIGINT 值对待（64 位整数）。

```
BIT_COUNT(0)           → 0
BIT_COUNT(1)           → 1
BIT_COUNT(2)           → 1
BIT_COUNT(7)           → 3
BIT_COUNT(-1)          → 64
```

■ DATABASE() 返回当前数据库名串，如果不存在当前数据库，则返回空串。

```
DATABASE()             → "samp_db"
```

■ DECODE(*str*, *password*) 给出一个作为调用 ENCODE() 所得结果的加密串 *str*，利用口令 *password* 串对其进行解密，并返回此结果串。

```
DECODE(ENCODE("secret","scramble"),"scramble") → "secret"
```

■ ENCODE(*str*, *password*) 利用口令串 *password* 对串 *str* 进行加密，返回二进制串形式的结果。此串可用 DECODE() 进行解码。因为加密后的结果是一个二进制串，因此若希望将它存储起来，应该利用 BLOB 列类型的列。

■ ENCRYPT(*str*)

```
ENCRYPT(str, salt)
```

对串 *str* 进行加密并返回结果串。这是一种不可逆的加密。如果给出 *salt* 参数，它应该是一个两字符的串。通过指定 *salt* 值，串的加密结果每次都相同。如果不给出 *salt* 参数，对 ENCRYPT() 的调用随时间不同产生不同的结果。

```
ENCRYPT("secret","AB") → "ABS5SGh1EL6bk"
ENCRYPT("secret","AB") → "ABS5SGh1EL6bk"
ENCRYPT("secret")       → "z1oPSN18WRwFA"
ENCRYPT("secret")       → "12FJgqDtV0g7Q"
```

ENCRYPT() 利用了 UNIX 的 crypt() 系统调用，并在两个方面依赖于它。首先，如果 crypt() 在您的系统上不可用，ENCRYPT() 总是返回 NULL。其次，ENCRYPT() 服从于 crypt() 在其上运行的系统的处理方式。特别是，crypt() 在某些系统上只处理串的前 8 个字符。

ENCRYPT() 是在 MySQL 3.21.12 中引入的。自 MySQL 3.22 以来，*salt* 可长于两个字符。

■ GET_LOCK(*str*, *timeout*) GET_LOCK() 与 RELEASE_LOCK() 联合使用以完成询问（协同）锁定。可利用这两个函数来编写基于一个协议锁名的状态进行协作的应用程序。

利用串 *str* 指定的锁名与 *timeout* 秒的超时值对 GET_LOCK() 进行调用。如果在规定的时间内成功获得相应的锁，则返回 1；如果由于超时使获得锁的企图失败，则返回 0；如果出错，则返回 NULL。*timeout* 值决定企图获得锁需要等待多长时间，而不是决定锁的持续时间。在获得锁后，该锁保持有效直到释放。

下面的调用请求一个名为“Nellie”的锁，对其等待 10 秒：

```
GET_LOCK("Nellie", 10)
```

此锁仅对该串自身起作用。它不锁定数据库、表或表中的任何行或列。换句话说就是，这个锁不阻止其他客户机对数据库表执行操作，这也就是 GET_LOCK() 锁定只是询问锁的原

因，它允许其他协同客户机决定该锁是否起作用。

对某个名称具有锁的客户机将阻塞其他客户机锁定该名称的企图（或多线程客户机内的其他线程的企图，这个多线程客户机维持对服务器的多个连接）。假定客户机 1 锁定了串“Nellie”。如果客户机 2 企图锁定相同的串，它将阻塞直到客户机 1 释放该锁或超时为止。在前一种情形中，客户机 2 将成功获得该锁，而后一种情况将会失败。

因为两个客户机不能同时锁定一个给定的串，所以在一个名称上有约定的数个应用程序可利用该名称的锁状态作为一个指示器，以指示何时执行与该名称有关的操作能保证安全。例如，可以基于某个表中的某行的唯一键值构造一个锁名，使得能够协同锁定该行。

通过用锁名调用 `RELEASE_LOCK()` 直接释放锁：

```
RELEASE_LOCK("Nellie")
```

如果成功地释放该锁，`RELEASE_LOCK()` 返回 1；如果该锁为其他连接所占有（只能释放自己所拥有的锁），返回 0；如果不存在这样的锁，返回 `NULL`。

客户机连接所占有的锁在该连接终止时，或者如果该客户机发布另一 `GET_LOCK()` 调用时，会自动释放（一个客户机连接一次只能锁定一个串）。上述的后一种情形中，所占用的锁在获得新锁前释放，即使新锁名与旧锁名相同也是如此。

`GET_LOCK(str, 0)` 可用作一个简单的查询，不用等待就可确定 `str` 上的锁是否有效（如果 `str` 当前未被锁定，这当然会锁住此串，不过要记住恰当地调用 `RELEASE_LOCK()` 来释放它）。

`GET_LOCK()` 是在 MySQL 3.21.7 中引入的。

■ `LAST_INSERT_ID()`

```
LAST_INSERT_ID(expr)
```

无参数时，返回最近在当前服务器会话中生成的 `AUTO_INCREMENT` 值，如果未曾生成过这样的值，则返回 0。在有参数的调用时，可将此函数用于 `UPDATE` 语句。其结果以与自动生成的值相同的方式处理，这对于生成序列是非常有用的。

更详细的内容请参阅第 2 章。对于这两种形式的 `LAST_INSERT_ID()`，其值都是由服务器在每次连接的基础上维持的，而且将不被其他客户机更改，即使那些导致建立新自动生成值的客户机也不能更改。

带参数的 `LAST_INSERT_ID()` 的形式是在 MySQL 3.22.9 中引入的。

■ `LOAD_FILE(file_name)` 此函数读文件 `file_name` 并将其内容作为一个串返回。文件必须位于服务器上，必须以绝对（完全）路径名指定，而且必须是完全可读的，以保证不用试图读取一个受保护的文件。因为该文件必须在服务器上，所以您必须具有 `file` 权限。如果上述任一条件不满足，`LOAD_FILE()` 将返回 `NULL`。

`LOAD_FILE()` 是在 MySQL 3.23.0 中引入的。

■ `MD5(str)` 基于 RSA DATA Security, Inc. MD5 Message_Digest 算法，计算串 `str` 的校验和。返回一个由 32 位十六进制数字组成的串。

```
MD5("secret") → "5ebe2294ecd0e0f08eab7690d2a6ee69"
```

`MD5()` 是在 MySQL 3.23.2 中引入的。

■ `PASSWORD(str)` 给出一个串 `str`，计算并返回 MySQL 授权表中所用形式的加密口令串。这是一种不可逆的加密。

`PASSWORD("secret")`

→ "428567f408994404"

注意, `PASSWORD()` 不采用与 UNIX 加密用户账号口令所用的加密算法相同的算法。如果需要那种加密, 可使用 `ENCRYPT()`。

■ `RELEASE_LOCK(str)` `RELEASE_LOCK()` 与 `GET_LOCK()` 联合使用。详细内容请参阅 `GET_LOCK()` 的描述。

`RELEASE_LOCK()` 是在 MySQL 3.21.7 中引入的。

■ `SESSION_USER()` 此函数为 `USER()` 的同义词。

■ `SYSTEM_USER()` 此函数为 `USER()` 的同义词。

■ `USER()` 返回一个表示当前客户机用户名的串。自 MySQL 3.22.1 以来, 此串的形式为 `user@host`, 其中 `user` 为用户名, 而 `host` 为客户机通过其连接到服务器的主机名。

`USER()`

→ "paul@localhost"

`SUBSTRING_INDEX(USER(), "@", 1)`

→ "paul"

`SUBSTRING_INDEX(USER(), "@", -1)`

→ "localhost"

■ `VERSION()` 返回描述服务器版本的串, 例如, "3.22.25.log"。可能看到的跟在服务器版本号后的后缀为 `-log` (登录打开)、`-debug` (服务器正运行调试模式) 或 `-demo` (服务器正按演示模式运行)。

`VERSION()`

→ "3.23.1-alpha-log"

`VERSION()` 是在 MySQL 3.21.13 中引入的。

附录D SQL 语法参考

此附录介绍了MySQL提供的每条 SQL 语句。还介绍了在 SQL 代码中进行注释的语法。注释用来编写说明性的文本，而不对应实际操作。在 MySQL 中注释还可用来隐藏特定的关键字（MySQL 执行这些关键字，但其他数据库忽略它们）。请参阅“注释语法”一节。

MySQL 正处于发展中，经常会对其进行改进。因此应该经常参阅联机 MySQL 参考指南，看看又增加了什么新功能。此指南可在 <http://www.mysql.com/> 处得到。

语法描述使用下列约定：

- 可选信息括在方括号 ([]) 中。如果某个列表括在方括号中，表示可以进行选择。
- 如果某个列表括在花括号 ({ }) 中，表示必须从中选择一项。
- 竖线 (|) 分开列表中的可选项。
- 省略号 (...) 表示该省略号前的项可以重复。

除非另有说明，否则这里给出的语句自 MySQL 3.22.0 版以来都有效。

D.1 SQL 语句

本节介绍每条 MySQL 的 SQL 语句的语法和含义。如果不具有使用某条语句的权限，则执行该语句将会失败。例如，如果没有访问数据库 db_name 的权限，执行 USE db_name 将会失败。

D.1.1 ALTER TABLE

`ALTER [IGNORE] TABLE tbl_name action_list`

ALTER TABLE 语句允许对表进行重命名或修改其结构。为了使用它，应该指定表名 *tbl_name*，然后给出要对表执行的一个或多个操作的说明。如果在新表的唯一键中出现重复值，IGNORE 关键字将起作用。这时如果不使用 IGNORE 关键字，则 ALTER TABLE 语句的作用将被取消。使用 IGNORE，唯一键值重复的行将被删除。

action_list 指定一个或多个操作，各操作之间用逗号分隔。各个操作依次执行。有如下的操作：

- `ADD [COLUMN] col_declaration [FIRST | AFTER col_name]` 给表增加一列。
col_declaration 为列定义，它与 CREATE TABLE 语句中所用的格式相同。如果给出 FIRST 关键字则增加的列成为表中第一列，如果给出了 AFTER *col_name*，则增加的列放置在指定的列后。如果未指定增加列的位置，则此列成为表中最后一列。

`ALTER TABLE member`

`ADD member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY`

`ALTER TABLE member`

`ADD member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
FIRST`

`ALTER TABLE member`

`ADD member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
AFTER suffix`

■ **ADD INDEX** [*index_name*] (*index_columns*) 给表增加索引。增加的索引基于 *index_columns* 中所指定的列，这些列都必须是表 *tbl_name* 中的列。如果指定了多列，列之间必须用逗号分隔。对于 **CHAR** 和 **VARCHAR** 列，可对列的前缀进行索引，即利用 *col_name*(*n*) 语法对列值的前 *n* 个字符进行索引。对于 **BLOB** 和 **TEXT** 列，必须指定前缀值，不允许对整个列进行索引。如果未指定索引名 *index_name*，那么系统会根据第一个索引列的名称自动选择一个索引名。

■ **ADD PRIMARY KEY** (*index_columns*) 在给定的列上增加一个主键。给此键的名称为 **PRIMARY**。如像在 **ADD INDEX** 子句中一样指定 *index_columns*。如果已经存在主键，则出错。

```
ALTER TABLE president ADD PRIMARY KEY (last_name, first_name)
```

■ **ADD UNIQUE** [*index_name*] (*index_columns*) 给表 *tbl_name* 增加一个唯一值索引。与 **ADD INDEX** 子句中一样指定 *index_name* 和 *index_columns*。

```
ALTER TABLE absence ADD UNIQUE id_date (student_id, date)
```

■ **ALTER [COLUMN]** *col_name* {**SET DEFAULT** *value* | **DROP DEFAULT**} 修改给定列的缺省值，或者对指定的值进行修改或者删除当前缺省值。在后一种情形中，将分配一个新的缺省值，如 **CREATE TABLE** 语句的相应条目所述。

```
ALTER TABLE event ALTER type SET DEFAULT "Q"  
ALTER TABLE event ALTER type DROP DEFAULT
```

■ **CHANGE [COLUMN]** *col_name col_declaration* 更改列的名称和定义。*col_name* 为列的当前名称，*col_declaration* 为定义，该列的定义将更改成此定义。*col_declaration* 与 **CREATE TABLE** 语句中相同的格式给出。注意该定义包括了新的列名，如果不想改动列名，可两次给出相同的列名。

```
ALTER TABLE student CHANGE name name VARCHAR(40)  
ALTER TABLE student CHANGE name student_name CHAR(30) NOT NULL
```

■ **DROP [COLUMN]** *col_name* 从表中删除指定的列。如果要删除的列是索引的组成部分，则这个列也将从这些索引中删除。如果删除了一个索引中所有的列，则此索引也被删除。

```
ALTER TABLE president DROP suffix
```

■ **DROP INDEX** *index_name* 从表中删除指定的索引。

```
ALTER TABLE member DROP INDEX name
```

■ **DROP PRIMARY KEY** 从表中删除主键。如果表中没有创建 **PRIMARY KEY** 的唯一索引，但有一个或多个 **UNIQUE** 索引，则删除这些索引中的第一个索引。

```
ALTER TABLE president DROP PRIMARY KEY
```

■ **MODIFY [COLUMN]** *col_declaration* 更改列的定义。利用与 **CREATE TABLE** 语句条目中列描述相同的格式给出列定义 *col_declaration*。此定义以列名开始，它表示怎样对列进行修改。**MODIFY** 是在 MySQL 3.22.16 中引入的。

```
ALTER TABLE student MODIFY name VARCHAR(40) DEFAULT "" NOT NULL
```

■ **RENAME [AS]** *new_tbl_name* 重新将表 *tbl_name* 命名为 *new_tbl_name*。

```
ALTER TABLE president RENAME prez
```

■ *table_options* 指定表排序选项，此排序选项可在 CREATE TABLE 语句的 *table_options* 部分给出。

```
ALTER TABLE score TYPE = MYISAM CHECKSUM = 1
```

D.1.2 CREATE DATABASE

```
CREATE DATABASE db_name
```

创建一个具有指定名称的数据库。如果要创建的数据库已经存在，或者没有创建它的适当权限，则此语句失败。

D.1.3 CREATE FUNCTION

```
CREATE [AGGREGATE] FUNCTION function_name
  RETURNS {STRING | REAL | INTEGER}
  SONAME shared_library_name
```

定义一个装入 mysql 数据库的 function 表中的用户定义函数（UDF）。*function_name* 为一个名称，SQL 语句利用此名称引用函数。跟在 RETURNS 后的关键字指出此函数的返回值的类型。*shared_library_name* 为包含可执行代码或函数的文件路径名。

AGGREGATE 关键字指定此函数为一个类似 SUM() 或 MAX() 的聚合函数。AGGREGATE 是在 MySQL 3.23.5 中引入的。

CREATE FUNCTION 要求服务器设置为一个动态连接库（不能是静态库），因为 UDF 机制要求动态连接。关于编写用户定义函数的说明，请参阅 MySQL 参考指南。

D.1.4 CREATE INDEX

```
CREATE [UNIQUE] INDEX index_name ON tbl_name (index_columns)
```

给表 *tbl_name* 增加一个名为 *index_name* 的索引。此语句根据有无 UNIQUE 关键字，分别按 ALTER TABLE ADD INDEX 或 ALTER TABLE ADD UNIQUE 语句处理。详细内容请参阅 ALTER TABLE 的相应条目。CREATE INDEX 是在 MySQL 3.22 中引入的。

D.1.5 CREATE TABLE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [[IGNORE | REPLACE] select_statement]
```

create_definition:

```
{ col_declaration
| PRIMARY KEY (index_columns)
| KEY [index_name] (index_columns)
| INDEX [index_name] (index_columns)
| UNIQUE [INDEX] [index_name] (index_columns)
| [CONSTRAINT symbol] FOREIGN KEY index_name (index_columns)
  [reference_definition]
| CHECK (expr) }
```

col_declaration:

```
col_name col_type
  [NOT NULL | NULL] [DEFAULT default_value]
```

```
[AUTO_INCREMENT] [PRIMARY KEY] [reference_definition]
```

```
reference_definition:
```

```
REFERENCES tbl_name [{index_columns}]  
[MATCH FULL | MATCH PARTIAL]  
[ON DELETE reference_option]  
[ON UPDATE reference_option]
```

```
reference_option:
```

```
{RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT}
```

CREATE TABLE 语句创建一个名为 *tbl_name* 的新表。如果给出 TEMPORARY 关键字, 则创建一个临时表, 该表仅在当前客户机连接终止前存在, 或者在发布 DROP TABLE 语句前存在。只有创建临时表的客户机才能看见此临时表。

通常, 企图创建一个已经存在的表时将产生错误。但在两种情况下不会出错。第一种情况, 如果给出 IF NOT EXISTS 子句, 则不创建已经存在的这个表, 但也不出错。第二种情况, 给出 TEMPORARY, 则原来名为 *tbl_name* 的表在临时表存在期间对客户机是隐藏的, 这样也不会出错。但原来的表对其他客户机仍保持可见。在下次客户机会话或对临时表明确发布 DROP TABLE 后, 原来的表再次可见。

create_definition 清单指定希望创建的列和索引。如果通过后跟 SELECT 语句的方法创建此表, 此清单可省略。 *table_options* 子句允许指定表的各种属性。如果给出后面的 *select_statement* (以任意 SELECT 语句形式), 则利用该 SELECT 语句返回的结果创建该表。后面的段落中将对这些子句进行更详细的介绍。

TEMPORARY 关键字、IF NOT EXISTS 子句、*table_options* 子句以及能够利用 SELECT 语句的结果创建表等都是在 MySQL 3.23 中引入的。

create_definition 可以是列定义、索引定义、FOREIGN KEY 子句、*reference_definition* 或 CHECK 子句。后三者在其数据库系统中是兼容的, 但其余的不行。

列定义 *col_declaration* 以列名 *col_name* 和类型 *col_type* 开始, 后面可跟几个可选的关键字。列类型可以是附录 B 中列出的任意类型。请参阅该附录中的类型专用属性。可跟在列类型之后的可选关键字如下:

- NULL 或 NOT NULL 指出该列是否可以包含 NULL 值。如果两者都不指出, 缺省为 NULL。
- DEFAULT *default_value* 给出该列的缺省值。这种属性不能用于 BLOB 或 TEXT 类型。如果无缺省说明, 自动分配一个缺省值。对于可取 NULL 值的列, 缺省值为 NULL。对于不能为 NULL 的列, 缺省值如下:
 - 对于数值列, 除 AUTO_INCREMENT 列外, 缺省值为 0。而对于 AUTO_INCREMENT 列, 其缺省值为该列的序列中的下一个数。
 - 对于非 TIMESTAMP 的日期和时间类型, 缺省值为该类型的“零”值(例如, DATE 类型的“零”值为“0000-00-00”)。对于 TIMESTAMP 类型, 缺省值为当前日期和时间。
 - 对于非 ENUM 的串类型, 缺省值为空串。对于 ENUM 类型, 缺省值为第一个枚举元素。
- AUTO_INCREMENT 此关键字仅用于整数列类型。AUTO_INCREMENT 列在插入

NULL 时是很特殊的, 这时插入的值实际为一个大于列中当前最大值的整数。AUTO_INCREMENT 值缺省时从 1 开始, 第一个值可用 AUTO_INCREMENT 表选项确明指定。这样的列还必须指定为 UNIQUE 索引或 PRIMARY KEY, 而且应该为 NOT NULL 列。每个表中至少有一个 AUTO_INCREMENT 列。

■ PRIMARY KEY 指定该列为一个 PRIMARY KEY。

■ UNIQUE 指定该列为 UNIQUE 索引列。自 MySQL 3.23 版以来都可指定此属性。

PRIMARY KEY、UNIQUE、INDEX 和 KEY 子句定义索引。PRIMARY KEY 与 UNIQUE 指定必须包含唯一值的索引。INDEX 和 KEY 意义相同, 它们都指定可以包含重复值的索引。索引基于 *index_columns* 中所指定的列, 其中的每个列都必须是 *tbl_name* 中的列。如果有多个列, 必须用逗号将它们分隔。对于 CHAR 和 VARCHAR 列, 可对列的前缀进行索引。如果未给出索引名 *index_name*, 系统会根据第一个索引列的名称自动选一个。

在 MySQL 3.23 以前的版本中, 索引列必须定义为 NOT NULL。自 MySQL 3.23 以来, 只有 PRIMARY KEY 中的列必须定义为 NOT NULL。

自 MySQL 3.23 以来可使用 *table_options* 子句。如果给出此子句, 它可以包含下面清单中的一个或多个选项。如果给出多个选项, 不应用逗号分隔它们。除非另有说明, 否则每个说明符都可用于所有表类型。

■ AUTO_INCREMENT = *n* 为表生成的第一个 AUTO_INCREMENT 值 (仅用于 MyISAM 表)。

■ AVG_ROW_LENGTH = *n* 表的近似平均行长。MySQL 用此选项来确定数据文件的最大尺寸, 只有具有 BLOB 或 TEXT 列的表需要此选项, 具有这两种列的表可能会增大超过 4GB。

■ CHECKSUM = {0 | 1} 如果设置为 1, MySQL 将对每个表行保持一个校验和。这样对于表的更新将会有某种轻微的损失, 但改善了表的校验过程 (仅用于 MyISAM 表)。

■ COMMENT = “*string*” 表的注释。最大长度为 60 个字符。

■ DELAY_KEY_WRITE = {0 | 1} 如果设置为 1, 表的索引高速缓存只是偶尔刷新, 而不是每个插入操作后都刷新 (只用于 MyISAM 表)。

■ MAX_ROWS = *n* 计划存储在表中的最多行数。

■ MIN_ROWS = *n*

计划存储在表中的最少行数。

■ PACK_KEYS = {0 | 1} 如果设置为 1, 索引块以较平常高的百分比进行压缩。其一般效果是更新性能受损, 但检索性能得以改善。如果设置为 0, 索引块不进行特殊的压缩。在此情形, 只有具有 8 个字符或更多字符的 CHAR 或 VARCHAR 键值进行压缩 (只用于 MyISAM 和 ISAM 表)。

■ PASSWORD = “*string*” 为加密表的描述文件指定一个口令。一般不起作用; 只对具有一定支持协议的顾客可用。

■ TYPE = {ISAM | MYISAM | HEAP} 指定表的存储格式。这些存储格式的特性在第 3 章的“表存储类型说明符”一节介绍。MySQL 自版本 3.23 以来的缺省格式为 MyISAM。对 MySQL 3.23 以前的版本, CREATE TABLE 总是以 ISAM 格式创建表。如果给出一个 *select_statement* 子句 (如一条 SELECT 查询), 该表利用查询结果的内容

进行创建。惟一索引中重复值的行根据指定的是 IGNORE 或 REPLACE，或者忽略或者替换现有行。如果两者都未指定，重复值时将出错，其余记录忽略。

下面创建一个具有三列的表。id 列为 PRIMARY KEY，其余两列组成一个索引：

```
CREATE TABLE customer
(
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    last_name CHAR(30) NOT NULL,
    first_name CHAR(20) NOT NULL,
    PRIMARY KEY (id),
    INDEX (last_name, first_name)
)
```

下面创建一个临时表并使其为一个 HEAP（内存中）表以加取得更快的速度：

```
CREATE TEMPORARY TABLE tmp_table
(id MEDIUMINT NOT NULL UNIQUE, name CHAR(40))
TYPE = HEAP
```

下面创建一个表作为另一表的拷贝：

```
CREATE TABLE prez_copy SELECT * FROM president
```

下面只用另一表的部分创建一个表：

```
CREATE TABLE prez_alive SELECT last_name, first_name, birth
FROM president WHERE death IS NULL
```

如果指定所要创建的表的创建定义并利用后缀 SELECT 语句进行填充，则此定义在表的内容已经插入表中时才使用。例如，可以定义将某个选定的列做成一个 PRIMARY KEY：

```
CREATE TABLE new_tbl (PRIMARY KEY (a)) SELECT a, b, c FROM old_tbl
```

D.1.6 DELETE

```
DELETE [LOW_PRIORITY] FROM tbl_name
[WHERE where_expr] [LIMIT n]
```

从表 *tbl_name* 中删除行。所删除的行是那些与表达式 *where_expr* 中给出条件相符的行。
例：

```
DELETE FROM score WHERE event_id = 14
DELETE FROM member WHERE expiration < CURRENT_DATE
```

如果省略 WHERE 子句，表中所有记录都将被删除。

除非不给出 WHERE 子句，否则 DELETE 将返回所删除记录的数目。在不给出 WHERE 子句的这种特殊情况下，MySQL 将优化删除过程，它将从头开始建立数据和索引文件，而不是一行一行地删除记录。这样做极快，但不给出删除的记录计数，返回的计数值为 0。为了得出真实的删除记录数，应该给出能选中所有记录的 WHERE 子句。例如：

```
DELETE FROM tbl_name WHERE 1 > 0
```

但这样做会有一定的性能损失。

指定 LOW_PRIORITY 使此语句被延迟至无客户机从表中读数据为止。LOW_PRIORITY 是在 MySQL 3.22.5 中引入的。

如果给出 LIMIT 子句，则值 *n* 规定了将要删除的最大行数。LIMIT 是在 MySQL 3.22.7 中引入的。

D.1.7 DESCRIBE

```
{DESCRIBE | DESC} tbl_name {col_name | pattern}
```

此语句提供与 SHOW COLUMNS 相同的信息。更详细的内容请参阅 SHOW 条目。

D.1.8 DROP DATABASE

```
DROP DATABASE [IF EXISTS] db_name
```

删除给定的数据库。在删除一个数据库后，它就永远没有了，因此要特别小心。如果要删除的数据库不存在，则此语句将失败（除非给出的 IF EXISTS）。在没有恰当的权限时，此语句也将失败。可利用 IF EXISTS 子句禁止在数据库不存在时通常将出现的错误消息。IF EXISTS 子句是在 MySQL 3.22.2 中引入的。

一个数据库由数据目录下的一个目录表示。如果在该目录中放置了非表文件，这些文件不会被 DROP DATABASE 语句删除。在此情形下，数据库目录本身也不会被删除。

D.1.9 DROP FUNCTION

```
DROP FUNCTION function_name
```

删除以前用 CREATE FUNCTION 装入的用户定义函数。

D.1.10 DROP INDEX

```
DROP INDEX index_name ON tbl_name
```

从表 tbl_name 中删除索引 index_name。此语句处理为一条 ALTER TABLE DROP INDEX 语句。详细内容请参阅 ALTER TABLE 条目。DROP INDEX 是在 MySQL 3.22 中引入的。

D.1.11 DROP TABLE

```
DROP TABLE [IF EXISTS] tbl_name [, tbl_name] ...
```

从数据库中删除指定的表。如果给出 IF EXISTS 子句，则删除不存在的表不会出错。IF EXISTS 是在 MySQL 3.22.2 中引入的。

D.1.12 EXPLAIN

```
EXPLAIN tbl_name  
EXPLAIN select_statement
```

此语句的第一种形式等价于 SHOW COLUMNS FROM tbl_name。详细信息请参阅 SHOW 语句的说明。

EXPLAIN 语句的第二种形式提供了 MySQL 将怎样执行跟在 EXPLAIN 关键字后的 SELECT 语句的信息。例如：

```
EXPLAIN SELECT score.* FROM score, event  
WHERE score.event_id = event.event_id AND event.event_id = 14
```

EXPLAIN 的输出由包含以下各列的一个或多个行组成：

■ table 输出的行所涉及的表

- **type** MySQL 将执行的连接类型。可能的类型从最好到最差分别为：`system`、`const`、`eq_ref`、`ref`、`range` 和 `ALL`。更好的类型限制更严，这意味着 MySQL 在进行检索时必须考虑的行更少。
- **possible_keys** MySQL 在查找 `rows` 列中指定的表中的行时所考虑的索引。NULL 值表示未发现索引。
- **key** MySQL 查找表中行时将实际使用的索引。NULL 值表示将不使用索引。
- **key_len** 将使用多长的索引。如果 MySQL 将使用最左前缀索引，这个值可能小于完全索引行的长度。
- **ref** MySQL 将对此值比较索引值。词 `const` 或 ‘`???`’ 表示该比较是针对常量进行的，一个列名表示一个列对列的比较。
- **rows** MySQL 为完成查询必须检查的表的行数估计。此列中值的积为必须要对所有表进行检查的行组合的总数估计。
- **extra** `Only index` 表示 MySQL 可以只利用索引检索表中的信息而不用查看数据文件。`where used` 表示使用 `SELECT` 语句的 `WHERE` 子句中的信息。

D.1.13 FLUSH

`FLUSH flush_option [, flush_option] ...`

刷新服务器所用的各种内部高速缓存。flush_option 可以是下列任意一项：

- **HOSTS** 刷新主机高速缓存。
- **LOGS** 通过关闭和重新打开刷新日志文件。
- **PRIVILEGES** 重新装载授权表。
- **STATUS** 重新初始化状态变量。
- **TABLES** 关闭高速缓存中任意打开的表。

`FLUSH` 语句是在 MySQL 3.22.9 中引入的。`STATUS` 刷新选项是在 MySQL 3.22.11 中引入的。

D.1.14 GRANT

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)] ] ...
ON {*. * | * | db_name.* | db_name.tbl_name | tbl_name}
TO user [, user ] ...
[WITH GRANT OPTION]
```

`GRANT` 语句给一个或多个 MySQL 用户授予访问权限。priv_type 值指出所授予的权限。它由从下列清单中选出的用逗号隔开的权限类型组成：

权限说明符	该权限允许的操作
<code>ALTER</code>	更改表和索引
<code>CREATE</code>	创建数据库和表
<code>DELETE</code>	从表中删除记录
<code>DROP</code>	删除数据库和表
<code>INDEX</code>	建立或删除索引
<code>INSERT</code>	在表中插入新记录
<code>REFERENCES</code>	未使用

权限说明符	该权限允许的操作
SELECT	检索表中记录
UPDATE	修改表中已有记录
FILE	读写服务器上的文件
PROCESS	查看正在服务器中执行的线程的信息或删除线程
RELOAD	重新装载授权表或刷新日志、主机高速缓存或表高速缓存
SHUTDOWN	关闭服务器
ALL	全部权限，ALL PRIVILEGES 为其同义词
USAGE	一种特殊的“无权限”权限

您总是可以查找或删除自己所拥有的线程。PROCESS 权限允许查找或删除任意线程。

ON 子句指出所授的权限范围有多广，如下表所示：

权限说明符	权限适用的级别
.	全局权限：所有数据库，所有表
*	如果未选择缺省数据库，则为全局权限，否则为当前数据库的数据库级权限
db_name.*	数据库级权限，指定数据库中所有表
db_name.tbl_name	表级权限，指定表中的所有列
tbl_name	表级权限，缺省数据库中指定表的所有列

如果在 ON 子句中指定了一个表，可通过在 *column_list* 子句中指定一个或多个用逗号分隔的列使权限为专门针对列的权限。

TO 子句指出一个或多个应授予权限的用户。每个用户由具有一个可选的 IDENTIFIED BY 子句的 *user_name@host_name* 中的说明符构成。*user_name* 可以是一个名字或空串（''）；后者指定一个匿名用户。*host_name* 可给出作为 localhost、主机名、IP 地址或作为与域名匹配的模式或网络编号。模式字符为“%”和“-”，与用于 LIKE 操作符的意义相同。无主机名指定的 *user_name* 等价于 *user_name@%*。自 MySQL 3.23 以来，*host_name* 可以是一个用 *n.n.n.n/m* 表示法的 IP 号/网络掩码对，其中 *n.n.n.n* 表示 IP 地址，而 *m* 表示用于网络编号的位数。

IDENTIFIED BY 子句，如果给出，将分配一个口令给用户。此口令应该以无格式文本指定，不使用 PASSWORD 函数，与为 SET PASSWORD 语句指定口令相反。如果用户已经存在而且指定了 IDENTIFIED BY 子句，则新的口令将替代旧口令。否则旧口令保持不变。

GRANT 语句是在 MySQL 3.22.11 中引入的。

下面的语句示范了使用 GRANT 语句的某些方法。其他例子请参阅第 11 章。

建立一个用户 paul，他能够从任何主机访问 samp_db 数据库中的所有表。下面的两个语句是等价的，因为在 user 标识符中缺少的主机名部分等价于“%”。

```
GRANT ALL ON samp_db.* TO paul IDENTIFIED BY "secret"
GRANT ALL ON samp_db.* TO paul@% IDENTIFIED BY "secret"
```

建立一个对 menagerie 数据库中的表具有只读权限的用户。该用户可以从 xyz.com 域中的任意主机进行连接：

```
GRANT SELECT ON menagerie.* TO lookonly@%.xyz.com
IDENTIFIED BY "ragweed"
```

建立对 samp_db 数据库中的 member 表具有全部权限的一个用户。该用户可从一个专门

的主机上进行连接：

```
GRANT ALL ON samp_db.member TO member_mgr@boa.snake.net  
IDENTIFIED BY "doughnut"
```

建立一个超级用户，他可以做任何事情，包括给其他用户授权，但他必须从本地主机进行连接：

```
GRANT ALL ON *.* TO superduper@localhost IDENTIFIED BY "homer"  
WITH GRANT OPTION
```

建立 menagerie 数据库的一个匿名用户，他能从本地主机不用口令进行连接：

```
GRANT ALL ON menagerie.* TO ""@localhost
```

D.1.15 INSERT

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]  
tbl_name [(column_list)] VALUES (expr [, expr] ...) [, (...)] ...  
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]  
tbl_name [(column_list)] SELECT ...  
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]  
tbl_name SET col_name=expr [, col_name=expr] ...
```

在一个已有的表 *tbl_name* 中插入行，返回所插入的行数。MySQL 3.22.5 以前的版本需要 INTO 关键字。

LOW_PRIORITY 使语句延迟至无客户机从表中读取数据时执行。LOW_PRIORITY 是在 MySQL 3.22.5 中引入的。

RELAYED 为以后的插入将行放入队列中，且此语句立即返回，以便客户机可以继续执行而不用等待。但在此情形下，LAST_INSERT_TO() 将不返回表中 AUTO_INCREMENT 列的 AUTO_INCREMENT 值。DELAYED 插入是在 MySQL 3.22.15 中引入的。

如果指定 IGNORE，现有行中唯一键值重复的行被丢弃。如果出现重复值而未指定 IGNORE，将出错且不再有更多的行被插入。IGNORE 是在 MySQL 3.22.10 中引入的。

INSERT 语句的第一种形式需要一个 VALUES() 列表，它给出所有要插入的值。如果不给出 *column_list*，VALUES() 列表必须对表中每列指定一个值。如果给出由一个或多个以逗号分隔的列构成的 *column_list*，则必须在 VALUES() 列表中给每列指定一个值。未在列表中指定的列被设置为它们的缺省值。自 MySQL 3.22.5 以来，可以给出多个值的列表，这允许在单个 INSERT 语句中插入多个行。自 MySQL 3.23.3 以来，VALUES() 列表可以为空，这将插入一个每列都设置为缺省值的行。

INSERT 语句的第二种形式根据 SELECT 语句检索记录并将它们插入 *tbl_name* 中。SELECT 语句必须选择 *tbl_name* 表的所有列，或者如果给出了列的列表，将选择 *column_list* 中所指定的所有列。如果给出列的列表，在该列表中未指定的任何列将被设置为缺省值。

自 MySQL 3.22.10 以来可使用 INSERT 的第三种形式，这种形式将由相应表达式给出值赋予 SET 子句中指定的列。未指定的列设置为缺省值。

```
INSERT INTO absence (student_id, date) VALUES(14,"1999-11-03"),(34,NOW())  
INSERT INTO absence SET student_id = 14, date = "1999-11-03"  
INSERT INTO absence SET student_id = 34, date = NOW()  
INSERT INTO score (student_id, score, event_id)  
SELECT student_id, 100 AS score, 15 AS event_id FROM student
```


D.1.16 KILL

`KILL thread_id`

取消具有给定 `thread_id` 的服务器线程。要取消线程，除非取消的线程是您自己所拥有的，否则必须具有 `PROCESS` 权限。

`mysqladmin kill` 命令完成相同的操作，但在命令行上允许指定多个线程的 `ID` 值。而 `KILL` 语句只允许指定一个 `ID`。

引语句是在 MySQL 3.22.9 中引入的。

D.1.17 LOAD DATA

```
LOAD DATA [LOW_PRIORITY] [LOCAL] INFILE 'file_name'
  [IGNORE | REPLACE]
  INTO TABLE tbl_name
  import_options
  [IGNORE n LINES]
  [(column_list)]
```

`LOAD DATA` 从文件 `file_name` 中读取记录并将它们成批装入表 `tbl_name` 中。这样比 `INSERT` 快。

`LOW_PRIORITY` 使本语句延迟至无客户机从表中读取数据为止。`LOW_PRIORITY` 是在 MySQL 3.23 中引入的。

一般情况下，此文件是由服务器在服务器主机上直接读入的。在此情形下，必须具有 `FILE` 权限而且相应文件必须或者位于缺省数据库的数据库目录中，或者为完全可读的。如果给出 `LOCAL`，则客户机在客户机主机上读取该文件并将其内容通过网络发送到服务器。在此情形下，不需要 `FILE` 权限。`LOCAL` 是在 MySQL 3.22.15 中开始起作用的。

如果不给出 `LOCAL`，服务器按如下方法对文件定位：

- 如果 `file_name` 为绝对路径，服务器从根目录开始查找该文件。
- 如果 `file_name` 为一个相对路径，则其解释要根据该名称是否包含单一成分来进行。如果是包含单一成分，服务器将在缺省数据库的数据库目录中查找该文件。如果该文件名包含多个成分，服务器将在服务器的数据目录中从头开始查找该文件。如果给出 `LOCAL`，则文件名如下解释：

- 如果 `file_name` 为绝对路径，则客户机从根目录开始查找文件。

- 如果 `file_name` 为一个相对路径，则客户机从当前目录开始查找文件。

在惟一索引上重复值的行根据给出的是 `IGNORE` 或 `REPLACE`，或者忽略或者替换现有行。如果两者都未指定，将出错，并忽略其余记录。

`import_options` 子句指出数据的格式。子句中可用的选项也适用于 `SELECT ... INTO OUTFILE` 语句。`import_options` 的语法为：

```
[FIELDS
  [TERMINATED BY 'string']
  [OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char' ]]
[LINES TERMINATED BY 'string']
```

`string` 和 `char` 值可能包括下列转义序列以给出特殊字符：

序列	说 明
\0	ASCII 0
\b	退格
\n	换行（不走纸）
\r	回车换行
\s	空格
\t	Tab（制表符）
\'	单引号
\"	双引号
\\	反斜杠

自 MySQL 3.22.10 以来，还可使用十六进制常量表示任意字符。例如，`LINES TERMINATED BY 0x02` 表示该行由 Ctrl-B（ASCII 2）字符结束。

如果给出 `FIELDS`，则至少应该给出 `TERMINATED BY`、`ENCLOSED BY` 或 `ESCAPED BY` 子句中的某一条，并且给出不止一个子句，必须按这里的顺序给出。如果 `FIELDS` 和 `LINES` 都给出，`FIELDS` 必须在 `LINE` 之前。`FIELDS` 子句的成分如下使用：

■ `TERMINATED BY` 给出限定行中值的字符。

■ `ENCLOSED BY` 指定引用字符，如果给出它，则将从字段值的最后去掉。不管是否给出 `OPTIONALLY`，都会出现这种情况。对于输出（`SELECT...INTO OUTFILE`），`ENCLOSED BY` 字符用来括起输出行中的字段值。如果给出 `OPTIONALLY`，则只将 `CHAR` 和 `VARCHAR` 列的值括起来。

为了在输入字段值内包含一个 `ENCLOSED BY` 字符，应该双写它或在其前面放置一个 `ESCAPED BY` 字符。否则，它将被理解为表示字段结束。对于输出，字段值中的 `ENCLOSED BY` 字符跟在一个 `ESCAPED BY` 字符之后。

■ `ESCAPED BY` 字符用来指出特殊字符的转义。在下面的例子中，假定转义字符为反斜杠（“\”）。对于输入，“\N”（反斜杠 N）解释为空格，而“\0”（反斜杠 ASCII “0”）解释为一个零值的字节。对于其他字符，转义字符被去掉，按字面使用所跟的字符。例如，“\"”解释为一个双引号，即使字段值是用双引号括起来的也是如此。

对于输出，用转义字符来将 `NULL` 和零值字节编码为“\N”“\0”。此外，`ESCAPED BY` 和 `ENCLOSED BY` 字符跟在转义字符之后，就像字段和行结束串的第一个字符一样。如果 `ESCAPED BY` 字符为空（`ESCAPED BY ''`），则不做转义工作。为了给出“\”的转义字符，可双写它（`ESCAPED BY '\\'`）。

`LINES TERMINATED BY` 值指出满足表示行结束的字符。

如果既不给出 `FIELDS` 又不给出 `LINES`，则将它们缺省指定为如下所示：

```
FIELDS
  TERMINATED BY '\\t'
  ENCLOSED BY ''
  ESCAPED BY '\\\'
LINES TERMINATED BY '\\n'
```

换句话说，行中的字段是用制表符限定的，无需括起来，反斜杠作为转义字符对待，而且行以换行符终止。

如果 `FIELDS` 子句的 `TERMINATED BY` 和 `ENCLOSED BY` 值皆为空，则使用定长的行

格式，字段之间不用定界符。列值利用列的显示宽度读取（或输出写）。例如，VARCHAR(15) 和 MEDIUMINT(5) 分别按 15 个字符和 5 个字符读取。NULL 值按空格串写入。

如果给出 IGNORE *n* LINES 子句，则输入的前 *n* 行被丢掉。例如，如果数据文件具有一个列标题行，不希望将其输入数据库表中，可使用 IGNORE 1 LINES 将其去掉：

```
LOAD DATA LOCAL INFILE "my_tbl.txt" INTO TABLE my_tbl IGNORE 1 LINES
```

如果不给出 *column_list*，则假定输入行包含表中每列的值。如果给出由一个或多个逗号分隔的列名清单，则输入行应该包含每个指定列的值。清单中没有的列设置为它们的缺省值。如果某个输入行中的值达不到所期望的值的数目，缺少的值的列将设置为其缺省值。

如果有一个在 Windows 中建立的用制表符限定的文本文件，可利用缺省的列分隔符，但行大概是用回车/换行对结束的。为了加载文件，应该给出不同的行结束符（' \r ' 表示一个回车换行，而 ' \n ' 表示换行）：

```
LOAD DATA LOCAL INFILE "my_tbl.txt" INTO TABLE my_tbl
  LINES TERMINATED BY "\r\n"
```

不幸的是，对于在 Windows 中建立的文件，如果建立该数据文件的程序使用了古怪的 MS-DOS 约定，在文件的结尾利用 Ctrl-Z 字符表示文件结束，则可能会在数据库中以不正确的记录作为结束记录。这种情况下，或者不用这样的程序来写入该文件，或装载文件后将不正确的记录删掉。

逗号分隔值（CSV）的文件在字段间有逗号，而字段可能用双引号括起来。假定行在结尾处有换行符，LOAD DATA 语句按如下装载一个文件：

```
LOAD DATA LOCAL INFILE "my_tbl.txt" INTO TABLE my_tbl
  FIELDS TERMINATED BY "," ENCLOSED BY "\"
```

下面的语句示出怎样读取一个字段由 Ctrl-A（ASCII 1）字符分隔，行用 Ctrl-B（ASCII 2）字符结束的文件。

```
LOAD DATA LOCAL INFILE "my_tbl.txt" INTO TABLE my_tbl
  FIELDS TERMINATED BY 0x01 LINES TERMINATED BY 0x02
```

D.1.18 LOCK TABLES

```
LOCK TABLES lock_list
```

获得在 *lock_list* 中指定的表上的一个锁，如果有必要，则一直等待获得所有锁。LOCK TABLE 为 LOCK TABLES 的同义词。

lock_list 由一个或多个逗号分隔的项构成，每个项的语法如下：

```
tbl_name [AS alias_name] {READ | [LOW_PRIORITY] WRITE}
```

READ 表示一个读锁，这个锁锁定其他希望写表的客户机，但允许其他客户机读取该表。WRITE 表示一个写锁，它阻塞所有其他客户机，不管它们是读表或还是写表。如果此请求正等待另一读此表的客户机，则 LOW_PRIORITY WRITE 锁请求允许其他读取程序读此表。这个锁直到无其他读取程序时才能获得。

LOCK TABLES 允许指定一个别名，以便在后面的查询中能利用此别名锁住准备引用的表。如果在某个查询中多次使用一个表，那么必须对表的每个实例获得一个锁，按需要锁定

别名。

LOCK TABLES 释放当前占用的任何锁。因而，为了锁定多个表，必须利用单一的 LOCK TABLES 语句锁定它们。一个客户机占用的任何锁，当此客户机终止时都将被自动释放。

LOW_PRIORITY 是在 MySQL 3.22.5 中引入的。

```
LOCK TABLES student READ, score WRITE, event READ
LOCK TABLE member READ
```

D.1.19 OPTIMIZE TABLE

```
OPTIMIZE TABLE tbl_name
```

释放表中浪费的空间。DELETE、REPLACE 与 UPDATE 语句可能会在表中产生无用的空间，特别是那种具有可变长行的表更是如此。OPTIMIZE TABLE 语句消除浪费的空间，得到一个所占空间更小的表。这条语句是在 MySQL 3.22.7 中引入的。其他表维护选项安排在 MySQL 3.23 系列中增加，因此对于最新增加的选项应该查看联机 MySQL 参考指南。

D.1.20 REPLACE

```
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name [(column_list)] VALUES (expr [, expr] ...) [, (...)] ...
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name [(column_list)] SELECT ...
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name SET col_name=expr [, col_name=expr] ...
```

REPLACE 语句类似于 INSERT，但有例外，在 INSERT 语句中，如果要插入的行具有一个与表中已有行重复的 UNIQUE 索引值，则在插入新行前将删除旧行。而 REPLACE 语句不这样，所以 REPLACE 语句中没有 IGNORE 子句选择。

D.1.21 REVOKE

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)] ...]
  ON {*. * | * | db_name. * | db_name.tbl_name | tbl_name}
  FROM user [, user] ...
```

此语句取消指定用户的权限。*priv_type*、*column_list* 和 *user* 子句按与 GRANT 语句相同的方式指定。ON 子句中也允许相同的说明符。

REVOKE 并不从 user 授权表中删除用户。这表示相应的用户仍然可以连接到 MySQL 服务器。为了删除所有用户，必须从授权表中手工删除用户的条目。

REVOKE 语句是在 MySQL 3.22.11 中引入的。

下面的 REVOKE 语句取消 superduper@localhost 的所有权限：

```
REVOKE ALL ON *.* FROM superduper@localhost
```

取消允许 member_mgr 用户修改 samp_db 数据库中 member 表的权限：

```
REVOKE INSERT,DELETE,UPDATE ON samp_db.member FROM member_mgr@boa.snake.net
```

取消本地主机上的匿名用户对 menagerie 数据库中单个表拥有的所有权限：

```
REVOKE ALL ON menagerie.pet FROM ""@localhost
```

D.1.22 SELECT

```

SELECT
  [select_options]
  select_list
  [INTO OUTFILE 'file_name' export_options]
  [FROM tbl_list]
  [WHERE where_expr]
  [GROUP BY column_list]
  [HAVING where_expr]
  [ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] , ...]
  [LIMIT [offset,] rows]
  [PROCEDURE procedure_name] ]

```

此语句从指定的表中检索行。实际上，除了 SELECT 关键字和 select_list 子句以外，本语句中所有成分都是可选的，这样考虑到了只是进行简单表达式求值的语句。例如：

```
SELECT 'one plus one =', 1+1
```

如果给出 select_options 子句，则它可能包含下列选项：

■ ALL

```

DISTINCT
DISTINCTROW

```

这几个关键字控制重复行是否返回，ALL 使所有行返回。这是缺省的，DISTINCT 和 DISTINCTROW 指出重复行应从结果中删除。

■ HIGH_PRIORITY 如果一般情况下必须等待，指定 HIGH_PRIORITY 将给语句一个更高的优先权。如果其他语句，如 INSERT 或 UPDATE 由于另外的语句正在读 SELECT 语句中给出的表而进行等待，这时 HIGH_PRIORITY 将使 SELECT 语句具有高于这些写语句的优先权。这只在您知道 SELECT 语句将快速执行且必须立即进行的情况下才这样做，因为这样做将把写语句放在后面执行。HIGH_PRIORITY 是在 MySQL 3.22.9 中引入的。

■ SQL_BIG_RESULT

```
SQL_SMALL_RESULT
```

这些关键字指定结果集大或小，它向优化程序提供了可用来更有效地处理查询的信息。

SQL_SMALL_RESULT 是在 MySQL 3.22.12 中引入的。而 SQL_BIG_RESULT 是在 MySQL 3.23 中引入的。

■ STRAIGHT_JOIN 强制表以 FROM 子句中指定的顺序进行连接。如果认为优化程序不能做出最好选择时，可使用这个关键字。

select_list 子句指定要返回的输出列。列之间应该用逗号分隔。这里的列可能涉及表列或表达式。任何列都可以用 AS alias_name 语法指定一个别名。然后此别名成为输出中的列名，而且还可以在 GROUP BY、ORDER BY 和 HAVING 子句中引用。但不能在 WHERE 子句中引用别名。

特殊的符号 * 表示“FROM 子句中指定的表中所有列”，而 tbl_name.* 表示“给定表中所有的列”。

SELECT 语句的结果可利用 INTO 'file_name' OUTFILE 子句写入一个文件中。您必须具有 FILE 权限，并且该文件必须不是已存在的文件。此文件名用与 LOAD DATA 读非 LOCAL

文件时所用的相同规则来解释。 `export_options` 子句的语法与 `LOAD DATA` 语句的 `import_options` 子句的相同。更详细的内容请参阅 `LOAD DATA` 条目。

`FROM` 子句指定一个或多个表，必须从这些表中选择行。使用 `SELECT` 语句时，MySQL 支持下列 `JOIN` 类型：

```
tbl_list:
    tbl_name
    tbl_list, tbl_name
    tbl_list [CROSS] JOIN tbl_name
    tbl_list INNER JOIN tbl_name
    tbl_list STRAIGHT_JOIN tbl_name
    tbl_list LEFT [OUTER] JOIN tbl_name ON conditional_expr
    tbl_list LEFT [OUTER] JOIN tbl_name USING (column_list)
    tbl_list NATURAL LEFT [OUTER] JOIN tbl_name
    { oj tbl_list LEFT OUTER JOIN tbl_name ON conditional_expr }
```

连接类型从指定的表中按下述描述选择行，虽然实际返回到客户机的行可能会受 `WHERE`、`HAVING` 或 `LIMIT` 子句所限定。

对于由其自身给出的表，`SELECT` 从此表中检索行。

如果指定多个表并用逗号将它们隔开，`SELECT` 返回这些表中行的所有可能组合。使用 `JOIN`、`CROSS JOIN` 或 `INNER JOIN` 等价于使用逗号。`STRAIGHT_JOIN` 也类似，但强制优化程序按表给出的顺序对表进行连接。如果认为优化程序不能做出最好的选择时，可以使用它。

`LEFT JOIN` 从连接的表中检索行，但强制对左边表中的每一行生成一行，即使右边表中没有匹配的行也是如此。在无匹配的行时，右边表中的列作为空值返回。根据 `ON conditional_expr` 子句或 `USING (column_list)` 子句中给定的条件确定匹配的行。`conditional_expr` 是一种可用于 `WHERE` 子句的表达式形式。`column_list` 由一个或多个逗号分隔的列名构成，其中每一列都必须是出现在两个连接表中的列。`LEFT OUTER JOIN` 等价于 `LEFT JOIN`。其语法也同样以 `oj` 开始（这是为与 `ODBC` 兼容而引入的）。

`NATURAL LEFT JOIN` 等价于 `LEFT JOIN USING (column_list)`，其中 `column_list` 给出两个表共有的每一列。

表可在 `FROM` 子句中用 `tbl_name alias_name` 或 `tbl_name AS alias_name` 语法指派别名。`SELECT` 语句的任何地方的引用一个别名化的列必须利用别名来进行。

`WHERE` 子句给出一个用于从 `FROM` 子句指定的表中选择的行的表达式。不能满足表达式所给出条件的行将被丢弃。结果还可以进一步由 `HAVING` 和 `LIMIT` 子句进行限制。

`GROUP BY column_list` 子句根据列表中给出的列对结果集进行分组。如果在 `select_list` 子句中给出诸如 `COUNT()` 或 `MAX()` 一类的汇总函数时，使用这个子句。列可用列名或别名或在 `select_list` 内用位置引用。列的位置从 1 开始编号。

`HAVING` 子句在行已经满足了 `WHERE` 子句中指定的条件后，给出一个用来限制行的辅助表达式。不满足 `HAVING` 条件的行被丢弃。对于涉及不能在 `WHERE` 子句中测试的汇总函数的表达式，`HAVING` 子句是很有用的。但如果条件既在 `WHERE` 子句中又在 `HAVING` 子句中都是合法的，最好将其放在 `WHERE` 子句内，它在 `WHERE` 子句中容易为优化程序所分析。

`ORDER BY` 确定怎样对结果进行排序。与 `GROUP BY` 一样，列可由列名或别名引用，

或者按列选择列表内的位置进行引用。输出列缺省时按升序排序。为了明确地指定列的排序次序,可在列指示符后跟 ASC (升序) 或 DESC (降序)。自 MySQL 3.23.2 以来,也可以指定一个表达式作为排序列。例如, ORDER BY RAND() 以随机次序返回行。

LIMIT 子句可用来从 SELECT 语句的结果中选择部分行。LIMIT n 返回前 n 行。LIMIT m, n 返回从第 m 行开始的 n 行。对于 LIMIT, 行从 0 而不是从 1 开始编号。

PROCEDURE 命名一个过程, 结果集在返回给客户之前先送给此过程。自 MySQL 3.23 以来, 可利用 PROCEDURE ANALYSE() 获得列选择表中给出的列中数据的特征。

下面的语句演示了某些使用 SELECT 语句的方法。更多的例子请参阅第 1 章。

选择表的所有内容:

```
SELECT * FROM president
```

选择所有内容, 但按名字进行排序:

```
SELECT * FROM president ORDER BY last_name, first_name
```

选择在 “1900-01-01” 出生或之后出生的总统的记录:

```
SELECT * FROM president WHERE birth >= "1900-01-01"
```

选择在 “1900-01-01” 出生或之后出生的总统的记录, 但按出生的顺序进行排序:

```
SELECT * FROM president WHERE birth >= "1900-01-01" ORDER BY birth
```

确定 member 表中的行给出哪些州:

```
SELECT DISTINCT state FROM member;
```

从 member 表中选择行并将列作为逗号分隔的值写入一个文件:

```
SELECT * INTO OUTFILE "/tmp/member.txt"
    FIELDS TERMINATED BY "," FROM member
```

选择某个特定学分事件的前五个学分:

```
SELECT * FROM score WHERE event_id = 9 ORDER BY score DESC LIMIT 5
```

D.1.23 SET

```
SET [OPTION] option_setting, ...
```

SET 语句用来指定各种选项。最好是省去词 OPTION, 因为它在 MySQL 的未来版本中将被删除。

option_setting 值可为下列清单中的任一项:

■ CHARACTER SET {charset_name | DEFAULT} 指定客户机所用的字符集。客户机接收和发送的串用这个字符集进行映射。当前可用的惟一字符集是 cp1251_koi8。字符集名 DEFAULT 恢复缺省字符集。

```
SET CHARACTER SET cp1251_koi8
SET CHARACTER SET DEFAULT
```

■ INSERT_ID = n 指定下一条 INSERT 语句插入一行时, AUTO_INCREMENT 列所用的值。

```
SET INSERT_ID = 1973
```

■ LAST_INSERT_ID = n 指定 LAST_INSERT_ID() 返回的值。此选项用于更新日志的处理。

```
SET LAST_INSERT_ID = 48731
```

- **PASSWORD** [FOR *user*] = **PASSWORD**(" *password* ") 在没有 FOR 子句时, 设置当前用户的口令为 “ *password* ”。在有 FOR 子句时, 设置给定用户的口令。为了能够设置其他用户的口令, 必须具有修改 *mysql* 数据库的权限。 *user* 是以 *user_name*@*host_name* 形式, 利用与 GRANT 语句所接受的 *user_name* 和 *host_name* 相同类型的值给出的。

```
SET PASSWORD = PASSWORD("secret")
SET PASSWORD FOR paul = PASSWORD("secret")
SET PASSWORD FOR paul@localhost = PASSWORD("secret")
SET PASSWORD FOR bill@%.bigcorp.com = PASSWORD("old-sneep")
```

- **SQL_AUTO_IS_NULL** = {0 | 1} 如果此选项设置为 1, 最后插入的包含一个 AUTO_INCREMENT 值的行可利用 WHERE *auto_inc_col* IS NULL 形式的 WHERE 子句来选择。如 Access 这样的 ODBC 程序使用此选项。此选项的缺省设置为 1, 它是在 MySQL 3.23.5 中引入的。

```
SET SQL_AUTO_IS_NULL = 0
```

- **SQL_BIG_SELECTS** = {0 | 1} 如果此选项设置为 1, 则允许返回多于 *max_join_size* 行的 SELECT 语句。否则, 查询将被中止。此选项的缺省设置为 1。

```
SET SQL_BIG_SELECTS = 0
```

- **SQL_BIG_TABLES** = {0 | 1} 如果此选项设置为 1, 则所有临时表都存放在磁盘上而不是存入在内存中。性能降低, 但需要大临时表的 SELECT 语句将不会出现 “table full (表满)” 错误。此选项的缺省设置为 0 (将临时表装在内存中)。此选项一般对于 MySQL 3.23 及以上的版本不需要。

```
SET SQL_BIG_TABLES = 1
```

- **SQL_LOG_OFF** = {0 | 1} 如果此选项设置为 1, 当前客户机的需求不记录在通常的日志文件中。如果设置为 0, 则允许客户机记录日志。如果要此语句起作用, 客户机必须具有 PROCESS 权限。对更新日志的记录不受影响。

```
SET SQL_LOG_OFF = 1
```

- **SQL_LOG_UPDATE** = {0 | 1} 除影响更新日志的记录和不影响常规日志外, 此选项类似于 SQL_LOG_OFF (包括要求 PROCESS 权限)。

SQL_LOG_UPDATE 是在 MySQL 3.22.5 中引入的。

```
SET SQL_LOG_UPDATE = 1
```

- **SQL_LOW_PRIORITY_UPDATES** = {0 | 1} 如果此选项设置为 1, 修改表内容的语句 (DELETE、INSERT、REPLACE、UPDATE) 将等待直到无 SELECT 语句活动时为止, 或等待直到该表挂起为止。在其他语句活动中到达的 SELECT 语句立即开始执行而不用等待较低优先级的修改语句。

SQL_LOW_PRIORITY_UPDATES 是在 MySQL 3.22.5 中引入的。

```
SET SQL_LOW_PRIORITY_UPDATES = 0
```

- **SQL_SELECT_LIMIT** = {*n* | DEFAULT} 指定从 SELECT 语句返回记录的最多个数。在一条语句中直接给出 LIMIT 子句将优于此选项。此选项的缺省值为 “不限制”。如果曾经做过更改, 则设置为 DEFAULT 将恢复缺省设置。

```
SET SQL_SELECT_LIMIT = 100000
SET SQL_SELECT_LIMIT = DEFAULT
```

■ `SQL_WARNINGS = {1 | 0}` 如果设置为 1，MySQL 即使对单行插入也报告警告计数。一般，只对插入多行的 `INSERT` 语句报告警告计数。此选项的缺省设置为 0。`SQL_WARNINGS` 是在 MySQL 3.22.11 中引入的。

```
SET SQL_WARNINGS = 1
```

■ `TIMESTAMP = {timestamp_value | DEFAULT}` 指定一个 `TIMESTAMP` 值。此选项用于更新日志处理。

```
SET TIMESTAMP = DEFAULT
```

D.1.24 SHOW

```
SHOW COLUMNS FROM tbl_name [FROM db_name] [LIKE pattern]
SHOW DATABASES [LIKE pattern]
SHOW GRANTS FOR user_name
SHOW INDEX FROM tbl_name [FROM db_name]
SHOW PROCESSLIST
SHOW STATUS
SHOW TABLE STATUS [FROM db_name] [LIKE pattern]
SHOW TABLES [FROM db_name] [LIKE pattern]
SHOW VARIABLES [LIKE pattern]
```

`SHOW` 的各种形式提供了关于数据库、表、列及索引的信息，或提供了关于服务器运行的信息。有几种形式具有一个可选的 `FROM db_name` 子句，允许指定要显示信息的数据库。如果不给出该子句，则针对的是缺省数据库。有的形式允许可选的 `LIKE pattern` 子句以限定与模式相匹配的值的输出。这里的模式应该为一个串或包含 “%” 或 “_” 字符的 SQL 模式。

1. SHOW COLUMNS

`SHOW COLUMNS` 语句列出给定表的列。`SHOW FIELDS` 为 `SHOW COLUMNS` 的同义词。

```
SHOW COLUMNS FROM president
SHOW FIELDS FROM president
SHOW COLUMNS FROM president FROM samp_db
SHOW COLUMNS FROM tables_priv FROM mysql LIKE "%priv"
```

`SHOW COLUMN` 的输出包含下面各列：

- **Field** 表列名。
- **Type** 列类型
- **Null** 列是否可包含 `NULL`。此列或者为 `YES`，或者为空白（无）。
- **Key** 列是否进行了索引。
- **Default** 列的缺省值。
- **Extra** 关于列的额外信息。
- **Privileges** 您对列所具有的权限。只在 MySQL 3.23 以后才显示此信息。

2. SHOW DATABASES

`SHOW DATABASES` 语句列出服务器主机上可用的数据库。

```
SHOW DATABASES
SHOW DATABASES LIKE "test%"
```

3. SHOW GRANTS

SHOW GRANTS 语句显示特定用户的权限信息，它应该以 `user_name$host_name` 的形式给出，利用与 GRANT 语句接受的 `user_name` 和 `host_name` 相同类型的值。SHOW GRANTS 是在 MySQL 3.23.4 中引入的。

```
SHOW GRANTS FOR root@localhost
SHOW GRANTS FOR "*"@pit-viper.snake.net
```

4. SHOW INDEX

SHOW INDEX 语句显示表索引的有关信息。SHOW KEYS 为 SHOW INDEX 的同义词。

```
SHOW INDEX FROM score
SHOW KEYS FROM score
SHOW INDEX FROM samp_db.score
SHOW INDEX FROM score FROM samp_db
```

SHOW INDEX 的输出包含下面的列：

- Table 包含相应索引的表名。
- Non_unique 如果索引可以包含重复值则此值为 1，否则为 0。
- Key_name 索引名。
- Seq_in_index 索引中列的数目。列从 1 开始编号。
- Column_name 列名。
- Collation 列在索引中的排序顺序。此值可能是 A（升序）、D（降序）或 NULL（未排序）。降序键至今尚不能使用，但以后会实现。
- Cardinality 索引中唯一值的数目。这个值通过运行具有 `--analyze` 选项的 `myisamchk` 或 `isamchk` 来更新。
- Sub_part 如果仅对列的一个前缀进行索引，则为此前缀的长度。如果对整个列都进行了索引，此列值为 NULL。
- Packed 键怎样压缩，如果未压缩，则此列的值为 NULL。此列仅在 MySQL 3.23 或以上的版本中显示。
- Comment 为关于索引的内部注释所保留。此列仅在 MySQL 3.23或以上的版本中显示。

5. SHOW PROCESSLIST

SHOW PROCESSLIST 显示服务器中执行的线程的信息。其输出包含以下的列：

- Id 客户机线程 ID 号。
- User 与线程有关的客户机名。
- Host 客户机从其进行连接的主机。
- db 线程的缺省数据库
- Command 线程执行的命令。
- Time 当前在线程中执行的命令所用的时间量，以秒计。
- State 在处理一条 SQL 语句时有关 MySQL 状态的信息。这个值对于报告 MySQL 的问题，或通过邮件清单提供一个关于为什么线程在某种状态下呆了很长时间的问题时，很有用。
- Info 正在执行的查询。

6. SHOW STATUS

SHOW STATUS 语句显示服务器的状态变量及其值：

- Aborted_clients 由于客户机未正常关闭连接而中止的客户机连接数目。
- Aborted_connects 企图连接到服务器的失败次数。
- Connections 企图连接到服务器的次数（包括成功的和失败的）。如果这个数相当大，可能的话应该考虑在您的客户机中使用永久连接。
- Created_tmp_tables 处理查询时创建的临时表的数目。
- Delayed_errors 处理 INSERT DELAYED 行时出现的错误数。
- Delayed_insert_threads 当前 INSERT DELAYED 处理程序的数目。
- Delayed_writes 已写入的 INSERT DELAYED 行的数目。
- Flush_commands 已执行的 FLUSH 命令数。
- Handler_delete 从表中删除一行的请求数目。
- Handler_read_first 从表中读取第一行的请求数目。
- Handler_read_key 基于索引值读取一行的请求数目。
- Handler_read_next 按索引顺序读取下一行的请求数目。
- Handler_read_rnd 基于行的位置读取一行的请求数目。如果这个数目比所有其他报告的数目大得多，表示或者执行了许多需要全表扫描的查询，或者未恰当地使用索引。
- Handler_update 更新表中一行的请求数目。
- Handler_write 在表中插入一行的请求数目。
- Key_blocks_used 索引高速缓存中使用的块数。
- Key_read_requests 从索引高速缓存中读一个块的请求数目。
- Key_reads 从磁盘物理读索引块的数目。
- Key_write_requests 写一个块到索引高速缓存的请求数目。
- Key_writes 物理写索引块到磁盘的数目。
- Max_used_connections 曾经同时打开的连接的最大数目。
- Not_flushed_delayed_rows 等待 INSERT DELAYED 查询写入的行数。
- Not_flushed_key_blocks 已经修改但未进行磁盘刷新的键高速缓存中的块数。
- Open_files 打开的文件的数目。
- Open_streams 打开的流的数目。
- Open_tables 打开的表的数目。
- Opened_tables 曾经打开过的表的数目。如果这个数太大，可能需要增加表高速缓存的尺寸。
- Questions 服务器曾经接到过的查询数目（包括成功的和未成功的查询）。
- Slow_queries 执行时间比 long_query_time 长的查询数目。
- Threads_connected 当前打开的连接数目。
- Threads_running 未睡眠的线程数目。
- Uptime 服务器启动以来的秒数。

7. SHOW TABLE STATUS

SHOW TABLE STATUS 语句显示数据库中表的描述性信息。SHOW TABLE STATUS 是在 MySQL 3.23 中引入的。

```
SHOW TABLE STATUS
SHOW TABLE STATUS FROM samp_db
SHOW TABLE STATUS FROM mysql LIKE "%priv"
```

SHOW TABLE STATUS 的输出包括以下各列：

- Name 表名。
- Type 表的类型，可能是 NISAM (ISAM)、MyISAM 或 HEAP。
- Row_format 行的存储格式，可以是 Fixed (定长行)、Dynamic (行可变长) 或 Compressed。
- Rows 表中行的数目。
- Avg_row_length 表行所用字节的平均数目。
- Data_length 表数据文件的实际字节数。
- Max_data_length 表数据文件可使用的最大字节数。
- Index_length 索引文件的实际字节数。
- Data_free 数据文件中未用的字节数。如果这个数非常大，应该对相应的表发布一条 OPTIMIZE TABLE 语句。
- Auto_increment AUTO_INCREMENT 列将产生的下一个值。
- Create_time 表创建的时间。
- Update_time 表最近被修改的时间。
- Check_time 最近用 myisamchk 检查或修复表的时间。如果该表从未进行过检查或修复，则此列的值为 NULL。
- Create_options 在创建该表的 CREATE TABLE 语句的 table_options 子句中给出的其他选项。
- Comment 创建表时给出的注释文本。

8. SHOW TABLES

SHOW TABLES 语句显示数据库中表的清单。

```
SHOW TABLES
SHOW TABLES FROM samp_db
SHOW TABLES FROM mysql LIKE "%priv"
```

9. SHOW VARIABLES

SHOW VARIABLES 语句显示服务器变量及其值的一个清单。这些变量在附录 E 的 mysqld 条目中描述。

```
SHOW VARIABLES
SHOW VARIABLES LIKE "%thread%"
```

D.1.25 UNLOCK TABLES

```
UNLOCK TABLES
```

此语句释放当前客户机所占用的所有表锁。UNLOCK TABLE 为 UNLOCK TABLES 的同义词。

D.1.26 UPDATE

```
UPDATE [LOW_PRIORITY] tbl_name
```



```
SET col_name=expr [, col_name=expr ] ...  
[WHERE where_expr] [LIMIT n]
```

修改表 `tbl_name` 中已有行的内容。被修改的行为那些由 `WHERE` 子句中给出的表达式选中的行。对于那些选中的行，`SET` 子句中给出的每列都被设置为相应表达式的值。

```
UPDATE member SET expiration = NULL, phone = "197-602-4832"  
WHERE member_id = 14
```

如果未给出 `WHERE` 子句，将更新表中所有行。

`UPDATE` 返回更新的行数。但是除非确实对行中某些列的值进行了更改，否则该行并不认为已进行了更新。设置某列的值为原来该列已经含有的值对行不产生影响。如果应用程序确实需要知道有多少行与 `WHERE` 子句相符而不管 `UPDATE` 是否实际修改了值，那么应该在建立与服务器的连接时，指定 `CLIENT_FOUND_ROWS` 标志。请参阅附录 F 中 `mysql_real_connect()` 函数的相应条目。

`LOW_PRIORITY` 使该语句延迟至无客户读取表时才进行更新。`LOW_PRIORITY` 是在 MySQL 3.22.5 中引入的。

如果给出 `LIMIT` 子句，值 `n` 将指定更新的行的最大数目。`LIMIT` 是在 MySQL 3.23.3 中引入的。

D.1.27 USE

```
USE db_name
```

选择 `db_name` 使其成为当前数据库（即不包含明确的数据库名的表引用中的缺省数据库）。如果数据库不存在或没有访问它的权限，则 `USE` 语句将会失败。

D.2 注释语法

本节介绍如何在 SQL 代码中撰写注释。并且指出 `mysql` 客户机程序在注释解释方面存在的缺陷。在利用 `mysql` 批量方式执行的查询文件中，常常使用注释，因此在编写这样的查询文件时要特别注意下列规定。

MySQL 服务器理解三种类型的注释：

- 从 “#” 号开始到行结束的任何字符都作为一个注释处理。这个语法与大多数外壳程序和 Perl 中的相同。
- “/*” 与 “*/” 之间的任何字符都作为一个注释处理。这种形式的注释可跨多行。这个语法与 C 语言中的相同。
- 自 MySQL 3.23.3 以来，可以 “-- ”（两个短划线和两个空格）开始一个注释，从这两个短划线到行结束的任何字符都作为一个注释处理。这样的注释风格（除了在两个短划线后不要空格）为某些其他数据库所用。MySQL 需要用空格作为区分字符，以使诸如 `value1 - value2`（其中 `value2` 是负的）这样的表达式不被作为注释处理。

在执行查询时，服务器忽略所有的注释。但有一个例外，那就是以 “/*！” 开始的 C 风格的注释应该特殊处理。这种注释的文本中应该包含 SQL 关键字，这些关键字将被 MySQL 当作注释所在行的语句成分进行处理。例如，服务器认为下面两行是等价的：

```
INSERT LOW_PRIORITY INTO my_tbl SET ...  
INSERT /*! LOW_PRIORITY */ INTO my_tbl SET ...
```

这种注释是为 MySQL 的特殊扩展与关键字而设计的。MySQL 能识别这些关键字，而其他 SQL 服务器会忽略它们。这使编写能利用 MySQL 特殊功能而又能为其他数据库系统所接受的查询更为容易。“/*!” 注释风格是在 MySQL 3.22.7 中引入的。

自 MySQL 3.22.26 以来，可在“/*!”之后跟一个版本号告诉 MySQL，除非服务器的版本号不小于该版本号，否则忽略该注释。下面 UPDATE 语句中的注释除非服务器的版本号为 3.23.3 或以上，否则将忽略：

```
UPDATE my_table SET my_col = 100 WHERE my_col < 100 /*!32303 LIMIT 100 */
```

mysql 客户机在理解注释方面的限制较 MySQL 服务器更多。如果某些结构以 C 风格注释出现，将会产生混淆，因为它使用的分析程序较服务器的简单。例如，出现在 C 风格注释内的引用字符将被视为一个串，并继续查找该串的结束符直到找出一个匹配的引号为止。下面的语句说明了这种情况：

```
mysql> SELECT /* I have no quote */ 1;
+---+
| 1 |
+---+
| 1 |
+---+
mysql> SELECT /* I've got a quote */ 1;
'>
```

mysql 分析第一条语句，不出问题，然后将它送服务器执行，并显示另一个“mysql>”提示符。第二条语句含有一个有未匹配引号的注释。结果，mysql 进入串分析模式。在键入回车键以后，它仍然处于串分析模式中。从它显示“'>”提示符就可以知道这一点。为了解决这个问题，键入一个匹配的引号后跟一个“\c”命令以结束此查询。关于 mysql 提示符的意义的详细信息，请参阅附录 E。

附录E MySQL 程序参考

本附录介绍以下列出的 MySQL 程序。附录的后部分详细介绍了每个程序，其中包括短描述、调用语法、所支持的选项，以及所具有的任何内部变量。

- `isamchk` 和 `myisamchk` 对表进行检查和修复、执行键分布分析以及释放和重新激活索引的实用程序。
 - `myisampack` 和 `pack_isam` 产生压缩的只读表的实用程序。
 - `mysql` 具有行编辑能力的交互式程序，可以将查询发送到 MySQL 服务器中，还可以批处理方式使用来执行存储在一个文件中的查询。
 - `mysql.server` 启动和关闭 MySQL 服务器的脚本。
 - `mysqlaccess` 测试访问权限的脚本。
 - `mysqladmin` 执行管理操作的实用程序。
 - `mysqlbug` 产生故障报告的脚本。
 - `mysqld` MySQL 服务器。该程序必须一直运行使得客户机具有对服务器所管理的数据库访问。
 - `mysqldump` 转储数据库表内容的实用程序。
 - `mysqlimport` 将数据批量装载到表中的实用程序。
 - `mysqlshow` 提供关于数据库或表的信息的实用程序。
 - `safe_mysqld` 启动和监控 MySQL 服务器的脚本。
- 在语法描述中，可选的信息由方括号（[]）指明。

E.1 指定程序选项

大多数 MySQL 程序能理解影响程序操作的一些选项。选项可在命令行中指定，或在选项文件中指定。另外，有些选项可能由设置环境的变量指定。命令行中指定的选项优先于其他方式指定的选项，选项文件中的选项优先于环境变量的值。

从 MySQL 程序中获得联机帮助

除 `mysqlbug`、`mysql.server` 和 `safe_mysqld` 外，每个 MySQL 程序都理解 `--help` 选项。该选项提供一种快速的方法从程序本身获得有关该程序的联机帮助。例如，如果您对如何使用 `mysqlimport` 没有把握，可按以下形式调用该程序：

```
% mysqlimport --help
```

? 选项与 `help` 相同，尽管外壳程序可以将其解释为文件名的通配符：

```
% mysqlimport -?
```

```
mysqlimport: No match.
```

如果发生这种情况，试着用下列命令进行替换：

```
% mysqlimport -\?
```

大多数选项同时有短（单字母）格式和长（完整的词）格式。 `-?` 和 `--help` 选项就是这样的例子。如果短格式选项后跟一个值，在大多数情况下该选项和值可能由空格分隔。例如，

当您指定一个文件名时, `-upaul` 等价于 `-u paul`。`-p` (口令) 选项是一个例外, 任何口令值都必须跟在 `-p` 的后面, 中间没有空格。

每个程序的描述中都列出了程序可以理解的所有选项。如果程序好像不认识描述中所列出的某个选项, 可能您的版本是该选项引入之前的旧版本 (仔细检查该语法以确信您没有指定错误的选项)。

E.1.1 标准的 MySQL 程序选项

有几个选项在多个 MySQL 程序中都有标准的含义。它们将在下面一次性列出, 而不是在每个程序的描述中给出, 在每个程序的“支持的标准选项”一节指明程序可以理解这些选项中的哪几个 (本节只列出长格式名, 但程序也理解相应的短格式选项, 除非另有所指)。

标准选项是:

- `-C, --compress` 允许客户机和服务器之间通信所使用的协议压缩技术 (如果服务器支持该技术的话)。该选项只由客户机使用。它是在 MySQL 3.22.3 中引入的。
- `## debug_options, --debug=debug_options` 开启调试输出。除非 MySQL 是用调试支持环境建立的, 否则该选项不起作用。`debug_options` 串由冒号分隔的选项组成。`d:t:o,file_name` 就是一个典型的值, 它允许调试、开启函数调用项并退出跟踪, 以及发送输出结果到文件 `file_name` 中。
如果您希望进行大量的调试, 应该检查 MySQL 源程序分发包中的文件 `dbug/dbug.c`, 它描述了可以使用的所有选项。
- `--defaults-file=path_name` 指定选项文件的路径名。通常, 程序在几个位置中搜索选项文件, 但是, 如果 `--defaults-file` 被指定, 则只能读取这个指定的文件。如果您使用该选项, 它必须是命令行中的第一个选项。该选项是在 MySQL 3.22.23 中引入的。
- `-, --` 告诉程序打印帮助消息并退出。
- `-h host_name, --host=host_name` 指定连接的主机 (即, 服务器正在运行的主机)。该选项由客户机程序使用。
- `--on-defaults` 取消任何选项文件的使用。如果使用该选项, 它必须是命令行的第一个选项。
- `-p[password], --password=[password]` 指定在连接到服务器时要使用的口令。如果 `password` 没有在该选项名后指定, 程序将请求输入口令。如果给出 `password`, 则必须紧跟在该选项名的后面, 中间没有空格。该选项只由客户机程序使用。
- `-w, --pipe` 使用一个指定的管道连接到服务器。该选项只能用于运行在 Windows 下的客户机程序。
- `-p port_num, --port=port_num` 对于客户机程序, 这是在连接到服务器时要使用的端口号。它用于 TCP/IP 的连接 (主机不是 localhost 的连接)。对于 `mysqld`, 该选项指定监听 TCP/IP 连接的端口。
- `--print-defaults` 如果您调用程序而在命令行中没有带该选项的话, 则打印将要使用的该选项的值。它列出了将从选项文件 (和环境变量中) 中拾取的值。`--print-defaults` 有助于验证选项文件设置的正确性。它也有助于识别 MySQL 正在使用的、而您从来就没有指定过的选项。可以使用 `--print-defaults` 确定选项是否是从某个选项文件中拾取

的。如果使用该选项，它必须在命令行的第一个位置。

- **-O var-value, --set-variable var=value** 几个程序都有可设置的变量（操作参数）。--set-variable 选项可以设置这些变量。var 是分配给变量的值。变量的值以字节为单位，表示缓冲区的大小或长度。值可以由精确的数字表示，或以带有后缀 ‘K’ 或 ‘M’ 的数字表示以表明千字节或兆字节。后缀不是大小写敏感的，‘k’ 和 ‘m’ 等价于 ‘K’ 和 ‘M’。每个程序的变量都在该附录的程序描述中给出，如果调用带有 --help 选项的程序时，还可以显示这些变量。
- **-s, --silent** 指定安静（silent）方式。这并不意味着程序是完全安静的，只不过它产生的数据结果比正常的要少。几个程序都允许指定该选项多次使得程序逐渐安静（该选项也可在选项文件中工作）。
- **-S path_name, --socket=path_name** 对于客户机程序，它是在连接到服务器时要使用的套接字文件的全路径名。套接字文件用于对 localhost 的连接。
- **-u user_name, --user=user_name** 对于客户机程序，它是在连接到服务器时要使用的用户名。如果没有指定该选项，则缺省值是您在 UNIX 中的注册名和 Windows 中的 ODBC。对于 mysqld，该选项指明（用于运行服务器的）UNIX 账号名（为使该选项更有效，服务器必须作为 root 来启动）。
- **-v, --verbose** 指定详细方式。程序将产生比常规更多的输出结果。几个程序都允许多次指定该选项使程序逐渐详细（它也可在选项文件中工作）。
- **-V, --version** 告诉程序打印版本信息串并退出。

E.1.2 选项文件

选项文件是在 MySQL 3.22 中引入的。它们允许存储 MySQL 程序选项，使得您不必每调用一个程序都在命令行输入选项。选项文件由 isamchk、myisamchk、myisampack、mysql、mysqladmin、mysqld、mysqldump、mysqlimport、mysql.server 和 pack_isam 读取。在 MySQL 安装目录的 share/mysql 目录中，或在源程序分发包的 support-files 目录中，可以找到一个实例选项文件 my-example.cnf。

选项文件中指定的任何选项都可以通过在命令行中直接指定该选项的另一个值而被忽略。

MySQL 程序通过咨询几个文件来查看选项，但是，丢失一个选项文件也不是什么错误。在 UNIX 中，可以对选项按给定的顺序查看下列文件：

文件名	内 容
/etc/my.cnf	全局选项
DATADIR/my.cnf	服务器专用选项
./my.cnf	用户专用选项

在 Windows 中可读取下列文件：

文 件 名	内 容
SYSTEMDIR \my.ini	全局选项
c:\my.cnf	全局选项
DATADIR \my.cnf	服务器专用选项

DATADIR 表示机器中的数据目录的路径名（这是编译到服务器中的路径名，它不能用 --

datadir 选项改变)。在 Windows 中, DATADIR 是 c:\mysql\data。SYSTEMDIR 代表 Windows 系统目录的路径名(通常可能是 C:\Windows 或 C:\WinNT)。尽管 Windows 的选项文件名使用 ‘\’ 作为路径名部分的分隔符来指定,但在文件中指定的路径名应该使用 ‘/’ 作为路径名分隔符来书写。如果要想用 ‘\’ 作为分隔符,应使用 ‘\\’,因为 ‘\’ 被认为是转义符。

全局选项文件由能够理解选项文件的所有程序使用。服务器的数据目录中的文件只能由相应的服务器使用。用户专用的文件由该用户所运行的程序使用。

选项按组指定,例子如下:

```
[client]
user=paul
password=secret
```

```
[mysql]
no-auto-rehash
```

```
[mysqlshow]
status
```

组名在方括号中给出,通常它对应一个程序名。在上例中, [mysql] 指明 mysql 客户机的选项组,而 [mysqlshow] 指明 mysqlshow 的选项组。专用的组名 [client] 允许指定可应用于所有客户机程序的选项。标准的 MySQL 客户机程序可同时查看到 [client] 组和与客户机名相同名字的组。例如,mysql 可查看到 [client] 和 [mysql] 组,而 mysqlshow 可查看到 [client] 和 [mysqlshow] 组。

跟随在组名下面的任何选项都与该组相关。一个选项文件可能包含任何数量的组,后列出的组优先于先列出的组。如果一个给定的选项在一个程序可以查看到的多个组中多次出现,则最后列出的那个值被采用。

每个选项都应在单独的一行上指定。行中的第一个词是选项名,必须用不带前导虚线的长名字格式指定(例如,如果要在命令行中指定压缩选项,可使用 -c 或 --compress,但在选项文件中只能使用 compress)。程序支持的任何长格式的选项都可以列在一个选项文件中。如果选项需要值,则用 ‘=’ 作为分隔符列出名字和值。

考虑下列命令行:

```
mysql --compress --user=paul --set-variable max_allowed_packet=24M
```

为了在一个选项文件的 [mysql] 组中指定同样的信息,可按以下格式书写:

```
[mysql]
compress
user=paul
set-variable=max_allowed_packet=24M
```

可注意到,在选项文件中,除了在变量名和值之间的 ‘=’ 外, set-variable 后面还跟一个 ‘=’ 字符。

选项文件的行如果为空,或以 ‘#’ 或 ‘;’ 开始,则该行被认为是注释而忽略。

某些转义符序列可用于选项值以指明特殊的字符:

序列	含义
\b	退格
\n	换行
\r	回车

\s	空格
\t	Tab
\\	反斜杠

小心不要将选项放在 [client] 组中, 因为这个组只能由一个单独的客户机来使用。例如, 如果将 mysql 专有的 skip-line-numbers 选项放在 [client] 组中, 将会发现其他的客户机程序 (如 mysqlimport) 不再工作 (您会得到错误消息, 后跟帮助消息)。反之, 将 skip-line-numbers 移到 [mysql] 组中, 一切正常。

由读取 [client] 组的所有 MySQL 程序可理解的选项是 compress、debug、help、host、password、pipe port、select、user 和 version (当然, 将 help 或 version 放在选项文件中没有太大的意义)。

保持用户专用的选项文件为专有的

要确保每个用户专用的选项文件由适当的用户所拥有, 并且将该方式设置为 600 或 400, 使得其他的用户不能读取它。除了使用该文件的用户外, 不要将 MySQL 用户名和口令信息暴露给任何人。

E.1.3 环境变量

MySQL 程序可查看到几个环境变量的值以获得选项设置。环境变量的优先权较低, 用环境变量指定的选项可能会因在选项文件或命令行中所指定的选项而被忽略。

MySQL 程序检查下列环境变量:

- **MYSQL_DEBUG** 调试时使用的选项。除非 MySQL 是用调试支持环境建立的, 否则该变量没有作用。设置 MYSQL_DEBUG 类似于使用 --debug 选项。
- **MYSQL_PWD** 建立对 MySQL 服务器的连接时使用的口令。设置 MYSQL_PWD 类似于使用 --password 选项。
- **MYSQL_TCP_PORT** 对于客户, 它是建立服务器的 TCP/IP 连接时使用的端口号。对于 mysqld, 它是监听 TCP/IP 连接的端口。设置 MYSQL_TCP_PORT 类似于使用 --port 选项。
- **MYSQL_UNIX_PORT** 对于客户机, 它是建立对运行在 localhost 上的服务器的 UNIX 域套接字连接时使用的套接字文件路径。对于 mysqld, 它是监听本地连接的套接字。设置 MYSQL_UNIX_PORT 类似于使用 --socket 选项。
- **TMPDIR** 创建临时文件的目录路径名。设置该变量类似于使用 --tmpdir 选项。
- **USER** 它是在对服务器连接时使用的 MySQL 用户名。该变量只由运行在 Windows 中的客户机程序使用, 设置它类似于使用 --user 选项。

mysql 客户机检查下列环境变量的值:

- **MYSQL_HISTFILE** 用于存储命令行历史的文件的名字。如果该变量没有被设置, 则缺省值为 \$HOME/.mysql_history, 这里的 \$HOME 是主目录的位置。
- **MYSQL_HOST** 在建立对服务器的连接时连接到的主机。设置该变量类似于使用 --host 选项。

E.2 isamchk 和 myisamchk

这些实用程序允许检查和修复毁坏的表，显示表的信息，执行索引键值分布分析，以及禁止或允许索引。第 4 章提供了关于键分析和禁用索引的详细信息。第 13 章提供了有关表的检查和修复的详细信息。

myisamchk 用于使用较新的 MyISAM 存储格式的表。这些表有带有 .MYD 和 .MYI 后缀的数据文件名和索引文件名。可对使用旧 ISAM 存储格式的表使用 isamchk。这些表有带有 ISD 和 ISM 后缀的数据文件名和索引文件名。如果告诉任一个实用程序对一个错误类型的表进行操作，它将打印警告消息并忽略该表。

不要在 MySQL 服务器正访问一个表时执行该表的检查或修复！第 13 章讨论了如何防止服务器在 myisamchk 或 isamchk 对表进行处理时使用该表。

E.2.1 用法

```
isamchk [options] tbl_name[.ISM] ...
myisamchk [options] tbl_name[.MYI] ...
```

如果没有选项，这些实用程序将检查指定表的错误。否则，表将按照特定选项的含义进行处理。tbl_name 参数可以是表名或该文件的索引文件名（.ISM 或 .MYI）。使用索引文件名是方便的，因为您可以使用文件名的通配符，对在单个命令中指定存储类型的所有表进行操作。例如，可以按如下检查某个目录中的所有表：

```
% isamchk *.ISD
% myisamchk *.MYI
```

这些实用程序不对表文件的位置在哪里做判断。您必须指定这些文件的路径名（如果它们不在当前目录中）。由于表文件不被假定是存放在服务器的数据目录中，因此可以将表文件拷贝到另一个目录中并对这些拷贝而非原始文件进行操作。

有些选项涉及到索引的编号。索引从 1 开始编号。可以发布 SHOW INDEX 查询或使用 mysqlshow --key 命令来查找指定表的索引编号。key_name 列按 myisamchk 和 isamchk 查看索引的顺序列出索引。

MYSQL_PWD 是危险因素

使用 MYSQL_PWD 变量来存储口令产生了一个危险因素，这是因为系统中的其他用户可以轻易地发现口令的值。例如，PS 实用程序将显示其他用户的环境变量的设置。

E.2.2 isamchk 和 myisamchk 支持的标准选项

```
--debug          --set-variable  --verbose
--help           --silent        --version
```

--silent 选项表示只打印错误的消息。在任一个程序带有 --description 或 --extend-check 选项时，或在 myisamchk 带有 --check 选项时，--verbose 选项都将打印更多的信息。可以通过多次指定 --silent 和 --verbose 选项来增加效果。

E.2.3 isamchk 和 myisamchk 共有的选项

isamchk 和 myisamchk 有许多共有的选项：

- **-a, --analyze** 指定键分布分析。它可以帮助服务器更快地执行基于索引的查找和连接。通过再次运行带有 **--description** 和 **--verbose** 选项的 **isamchk** 和 **myisamchk** 进行分析后, 可获得有关键分布的信息。
- **-b n, --block-search=n** 打印包含块 *n* 的块头部的表行的开始信息。它只用于调试操作。
- **-d, --description** 打印关于表的描述信息。
- **-e, --extend-check** 执行一个扩展的表检查。该选项很少需要使用, 因为 **isamchk** 和 **myisamchk** 用不扩展的检查方式应该能发现错误。
- **-f, --force** 迫使表被检查或修复, 即使该表的临时文件已经存在也是如此。通常, 如果 **isamchk** 和 **myisamchk** 发现一个 *tbl_name.TMD* 文件, 在打印一个错误消息后将简单地退出, 因为那表示该程序的另一个实例正在运行。但是, 如果您在程序运行时取消它前面的调用, 该文件还能够存在, 在这种情况下该文件被安全地删除 (另一种可选的方法是, 手工删除该临时文件)。如果在检查表时使用 **--force**, 程序将利用 **-recover** 自动重新启动来检查任何有问题的表。
- **-i, --information** 打印关于表的内容的统计信息。
- **-k n, --keys-used=n** 与 **--recover** 一起使用。告诉 MySQL 只更新前 *n* 个索引。换句话说, 它释放编号大于 *n* 的索引。该选项可用来提高 **INSERT**、**DELETE** 和 **UPDATE** 操作的性能。如果指定索引号为 0 将释放所有的索引。如果指定索引号为最大的索引编号将激活所有的索引。
- **-l, --no-symlinks** 如果一个表参数是符号连接, 则通常 **symlink** 所指向的表要修复。当指定该选项时, 不跟随 **symlink**, 而是将 **symlink** 替换成该文件的新版本。
- **-q, --quick** 该选项与 **--recover** 一起使用将比单独使用 **--recover** 时修复得快。当给出这两个选项时不触及数据文件。为了迫使程序修改数据文件 (即使发现重复的键值), 指定 **--quick** 选项两次。如果使用该选项两次, 肯定首先有一个数据文件备份。
- **-r, --recover** 执行常规的恢复操作。它可以修正大多数问题, 除了在索引文件中发现的重复值 (它们应该是惟一的) 外。
- **-o, --safe-recover** 使用比 **--recover** 方法要慢的一种恢复方法, 但它能够修复 **--recover** 不能修复的问题。
- **-S, --sort-index** 将索引块排序以提高块顺序读取的速度。
- **-R n, --sort-recoerds=n** 按照记录在索引 *n* 中的顺序对数据记录进行排序。基于该给定索引的后续检索应该更快。当首次对表执行该操作时, 它可能很慢, 因为记录将要重新排序。
- **-u, --unpack** 解开一个打包文件, **myisamchk** 可以解开利用 **myisamchk** 打包的 **MyISAM** 文件。**isamchk** 可以解开利用 **pack_isam** 打包的 **ISAM** 文件。该选项可以用来将压缩的只读文件表转换成可修改的格式。
- **-w, --wait** 如果一个表被锁定, 要一直等待它可用为止。如果没有 **--wait** 选项, 程序将为锁等待 10 秒中, 如果没有得到任何的锁, 将打印错误消息。

E.2.4 myisamchk 专有的选项

尽管 **isamchk** 没有专有的选项, 但 **myisamchk** 有几个专有的选项。

- `-c, --check` 检查表的错误。如果不指定任何选项, 该选项是缺省的。
- `-D n, --data-file-length=n` 当重新建立一个已经装满的数据文件时, 该选项指定该数据文件应该允许增长的最大长度 (当文件到达 MySQL 所施加的大小限制或由操作系统的大小约束所施加的大小限制时, 这种情况会发生。它还发生在行数达到内部表的数据结构所施加的限制时)。该值以字节为单位指定。该选项只在与 `-recover` 或 `--safe-recover` 一起使用时有效。
- `-F, --fast` 不检查自上次检查以来没有被修改的表。
- `-m, --medium-check` 用一种比扩展检查更快的但稍有点不彻底的方法来检查表 (`myisamchk` 帮助消息声称此方法“只能”发现 99.99% 的错误)。这种检查方法应该满足大多数的环境。medium 检查方法通过计算索引中键的 CRC 值进行工作, 并将这些键与从数据文件的索引列中所计算出的 CRC 值进行比较。
- `-T, --read-only` 不标记已经检查的表
- `-A [n], --set-auto-increment[=n]` 迫使 AUTO_INCREMENT 值起始为 `n` (或为较高的值, 如果该表已经包含带有与 `n` 一样大小的 AUTO_INCREMENT 值的记录)。如果没有指定任何值, 则该选项将下一个 AUTO_INCREMENT 设置为一个比当前最大值更大的值。
- `-t path_name, --tmpdir=path_name` 指定用于临时文件的目录路径名。如果该变量没有被设置, 则缺省值为 TMPDIR 环境变量的值或 /tmp。

E.2.5 isamchk 和 myisamchk 的变量

以下的 `isamchk` 和 `myisamchk` 变量可以用 `--set-variable` 选项来设置:

- `key_buffer_size` 索引块所使用的缓冲区的大小。
- `read_buffer_size` 读缓冲区的大小。
- `write_buffer_size` 写缓冲区的大小。
- `sort_buffer_size` 键值的存储操作所使用的缓冲区大小 (该变量用于 `--recover`, 但不用于 `--safe-recover`)。
- `sort_key_blocks` 该变量与索引使用的 B-树结构的深度有关。一般不需要修改它。
- `decode_bits` 在对压缩的表进行解码时使用的比特位。值越大导致操作的速度越快, 但将需要更多的内存。一般情况下使用缺省值就足够了。

E.3 myisampack 和 pack_isam

`myisampack` 和 `pack_isam` 实用程序产生压缩的只读表。它们可以使存储需求减少 40% 至 70% 以维持快速的记录访问。`myisampack` 将 MyISAM 表打包并处理所有列类型。`pack_isam` 将 ISAM 表打包并只处理不包含 BLOB 或 TEXT 列的表。

缺省设置时, `myisampack` 和 `pack_isam` 不包括在 MySQL 中, 但可用于订购了 10 个或更多许可证的客户, 或者是有扩展支持级或更高级别上的支持协议的客户。这些是可以改变的, 以便使程序在与 MySQL 分发包相同的条件下可用 (有关当前的支持环境和许可信息, 请参阅 MySQL 参考手册)。

所有的 MySQL 版本都不需要读取已经用这些实用程序打包的表。这使得它们对这样的

应用特别有用：即希望分发一个表，该表包含归档的或百科全书式的信息，并且是只读的，无须更新。如果想将一个打包文件转换到解包的和可修改的格式，可以使用 `myisamchk --unpack`（对于 MyISAM 表）或 `isamchk --unpack`（对于 ISAM 表）来进行。

`myisampack` 和 `pack_isam` 对数据文件进行打包，但不对索引文件打包。为了更新索引，可在运行 `myisampack` 后运行 `myisampack -rq`，或在运行 `pack_isam` 后运行 `isamchk -rq`。

E.3.1 用法

```
myisampack [options] tbl_name ...
pack_isam [options] tbl_name ...
```

`tbl_name` 参数可以是表名也可以是该表的索引文件名（对于 MyISAM 表是 `.MYI` 文件，对于 ISAM 表是 `.ISM` 文件）。如果您不在那个目录中，应指定表所在的目录路径名。

E.3.2 `myisampack` 和 `pack_isam` 支持的标准选项

```
--debug          --silent          --version
--help           --verbose
```

E.3.3 `myisampack` 和 `pack_isam` 共有的选项

- `-b, --backup` 告诉 `myisampack` 或 `pack_isam` 在对 `tbl_name` 打包前做每个 `tbl_name` 参数的备份，备份文件名为 `tbl_name.OLD`。
- `-f, --force` 强迫表打包，即使作为结果的打包文件比原始文件大或者该表的临时文件已经存在也是如此。通常，`myisampack` 或 `pack_isam` 如果发现一个文件 `tbl_name.TMD`，则在打印一个错误消息后退出，这是因为那已经表明程序的另一个实例正在运行。但是，如果在其运行时取消程序的上一次调用，该文件可能还存在，这时该文件可以被安全地删除。如果您了解这种情况，可使用 `--force` 告诉任一个实用程序对该表打包，即使临时文件存在。
- `-j join_tbl, --join=join_tbl` 将命令行指定的所有表连接到单个的打包表中，即 `join_tbl`。所有被合并的表必须有相同的结构（列名、类型和索引必须相同）。
- `-t, --test` 以测试方式运行。运行打包测试，并打印有关在实际对表打包时应获得的信息。
- `-T path_name, --tmpdir=path_name` 指定临时文件使用的目录路径名。
- `-w, --wait` 等待并重新测试表是否在使用中。如果当在打包时表可能会更新，就不要对其进行打包。

E.3.4 `pack_isam` 专有的选项

尽管 `myisampack` 没有专有的选项，但 `pack_isam` 有一个，即 `-p n, --packlength=n`。该选项允许 `pack_isam` 使用一个 n 字节的记录长度存储尺寸，这里的 n 是一个从 1 到 3 的整数。`pack_isam` 试图自动地确定记录打包表中每个记录长度所需要的字节数。在某些情况中，它能够确定可使用的较小长度。如果是这样的话，`pack_isam` 为该结果发布一条消息。你可以

再次运行 `pack_isam` 并用 `--packlength` 选项明确指定较小的长度，以达到节省某些附加的空间。

E.4 mysql

`mysql` 客户机是一个交互式程序，它允许对服务器进行连接，发布查询并浏览结果。如果您将该命令的输入重定向为从一个文件中读取，`mysql` 还能以批操作方式进行使用以执行存储在该文件中的查询。例如：

```
% mysql -u paul -p -h pit-viper.snake.net samp_db < my_query_file
```

在交互式方式中，当 `mysql` 启动时，它显示一个提示符 ‘`mysql>`’ 来表示正在等待输入。要想发布一个查询，在这里键入 `mysql`（如果需要可使用多行），然后键入 ‘`;`’（分号）或 ‘`/g`’ 表示该查询的末尾。`mysql` 将该查询发送到服务器上，显示结果，然后打印另一个提示符以表示正准备接收另一个查询。

`mysql` 通过改变提示符来表示它正在等待您输入什么：

<code>mysql></code>	等待一个新查询的首行
<code>-></code>	等待当前查询的下一行
<code>'></code>	等待当前查询中的单引号串的结束
<code>"></code>	等待当前查询中双引号串的结束

‘`'>`’ 和 ‘`">`’ 提示符表示您已经在上一行中开始了一个单引号或双引号串，并且还未输入终结引号。通常，它在您忘记终止一个串时会发生。如果这种情况出现，要想离开串集合方式，可输入相配的引号（即由提示符指定的引号），后面跟 ‘`\c`’ 以取消当前的查询。

当 `mysql` 用于交互式方式时，它在历史文件中保存查询。该文件在缺省时为 `$HOME/.mysql_history`，或者可通过设置 `MYSQL_HISTORY` 环境变量来明确指定文件名。查询可以从该命令的历史中重新调用并重新发布，而不论是否进一步进行了编辑。以下列表出一些编辑命令（箭头键在 Windows 下不工作）：

键 序 列	含 义
上箭头, Ctrl-P	调用上一行
下箭头, Ctrl-N	调用下一行
左箭头, Ctrl-B	向左移动光标（向后）
右箭头, Ctrl-F	向右移动光标（向前）
Escape Ctrl-B	向后移动一个词
Escape Ctrl-F	向前移动一个词
Ctrl-A	移动光标到行头
Ctrl-E	移动光标到行尾
Ctrl-D	删除光标下的字符
Delete	删除光标左边的字符
Escape D	删除词
Escape 空格	删除光标左边的词
Ctrl-K	删除从光标至行尾的所有内容
Ctrl_	撤消上次的修改，可以重复

某些选项禁止使用历史文件。通常是一些表示 `mysql` 非交互式使用的选项，如 `--batch`、`--html` 和 `--quick`。

E.4.1 用法

mysql [options] [db_name]

如果指定一个 *db_name* 参数, 则该数据库成为您会话的当前 (缺省的) 数据库。如果没有指定 *db_name* 参数, 则 mysql 的启动不带当前数据库, 您需要或者限定所有的表带数据库名进行引用, 或者发布 `USE db_name` 语句指定一个缺省的数据库。

E.4.2 mysql 支持的标准选项

--compress	--password	--silent	--version
--debug	--pipe	--socket	
--help	--port	--user	
--host	--set-variable	--verbose	

--silent 和 --verbose 可以指定多次以增加效果。

E.4.3 mysql 专有的选项

- **-B, --batch** 指定 mysql 应该以批方式运行。查询结果以制表符分隔形式显示 (每行在一个单独的显示行上, 列值之间用制表符分隔)。这对于生成要想导入到另一个程序 (如电子表格) 的结果特别方便。查询结果包括缺省时的一个列标题的初始行。为了禁用这些标题, 可使用 `--skip-column-names` 选项。
- **-T, --debug-info** 在程序终止时打印调试信息。
- **-e, query, --execute=query** 执行查询并退出。应该用引号将该查询括起, 以防外壳程序作为多命令行的参数来解释它。通过在查询串中使用分号来分隔查询, 可以指定多个查询。
- **-f, --force** 通常, 当 mysql 从文件中读取查询时, 如果出现错误则退出。该选项促使 mysql 保持正在处理的查询, 而不管是否有错误。
- **-H, --html** 产生 HTML 的输出结果。该选项是在 MySQL 3.22.26 中引入的。
- **-i, --ignore-space** 指示服务器忽略在函数名和导入参数列表的 ‘ (’ 字符之间的空格。通常, 函数名后面必须紧跟括号, 中间没有空格。
- **-A, --no-auto-rehash** 当 mysql 启动时, 它对数据库、表和列名进行散列以构造供名字快速完成使用的数据结构 (您可以在输入一个查询时键入名字的初始部分, 然后按 Tab 键。mysql 将完成该名字, 除非该名字不明确)。该选项禁止散列计算, 这样可使 mysql 启动得更快, 尤其是有许多表的时候。如果想要在启动 mysql 后使用名字自动完成, 可在 ‘mysql>’ 提示符下使用 `rehash` 命令。
- **-o, --one-database** 该选项是在从更新日志文件的内容中更新数据库时使用。它指示 mysql 只更新缺省数据库 (该数据库在命令行指定)。对其他数据库的更新将被忽略。如果命令行未指定任何数据库, 则不执行任何更新。
- **-q, --quick** 通常, mysql 在显示查询之前从服务器中检索整个查询结果。该选项使每行在检索时显示, 它使用很少的内存并可以成功地完成一些大的查询。但是, 该选项不应该对交互式使用。如果用户暂停输出或暂停 mysql, 服务器将继续等待, 可能干扰其他客户机。

- `-r, --raw` 书写没有扩展任何特殊字符的列值。该选项与 `--batch` 选项组合使用。
- `-N, --skip-column-names` 禁止列名作为查询结果的列标题显示。也可以通过指定 `--silent` 选项两次来达到相同的效果。
- `-L, --skip-line-numbers` 在出现错误时不写行号。
- `-t, --table` 以表格形式产生输出结果，每行值由横杠分隔并垂直排列。
- `-n, --unbuffered` 在每个查询后刷新用于与服务器通信的缓冲区。
- `-E, --vertical` 垂直打印查询结果，即，结果集的每个数据行（row）作为一个行（line）集显示，每列一行。每行由一个列名和值组成。在结果集中表示行号的行先于每个数据行的显示。垂直显示格式在查询产生很长的行时很有用。
如果没有指定该选项，可以通过用 ‘ \G ’ 而非 ‘ ; ’ ‘ \g ’ 终止单个的查询，来对这些查询开启垂直显示格式。
该选项是在 MySQL 3.22.5 中引入的。
- `-w, --wait` 如果不能建立对服务器的连接，则等待并重试。

E.4.4 mysql 的变量

以下 mysql 的变量可用 `--set-variable` 选项来设置：

- `max_allowed_packet` 用于服务器和客户机之间通信的缓冲区大小的最大值。
- `net_buffer_length` 用于服务器和客户机之间通信的缓冲区的初始大小。该缓冲区最大可扩展到 `max_allowed_packet` 字节。

E.4.5 mysql 命令

mysql 除了允许向服务器发送 SQL 语句外，本身还认识几个命令。每个命令都必须在一个单独的行中给出；行的结尾不需要分号。大多数这些命令都有一个长格式（由词组成）和一个短格式（由虚线后跟单字符）。长格式的命令不区分大小写。短格式的命令必须使用下列清单中的大小写字母指定：

- `clear, \c` 清除（取消）当前查询。当前查询是您正在输入的查询。该命令不取消已经发送到服务器的查询以及 mysql 正在为其显示输出结果的查询。
- `connect [db_name [host_name]], \r [db_name [host_name]]` 连接到指定主机上的指定数据库。如果数据库名或主机名忽略了，则使用当前 mysql 会话中所使用过的最新值。
- `edit, \e` 编辑当前查询。要使用的编辑器由 `EDITOR` 和 `VISUAL` 环境变量（或者是 `vi`，如果它们都没有被设置的话）的值决定。
- `ego, \G` 将当前查询发送到服务器中并垂直排列地显示其结果。该命令是在 MySQL 3.22.11 中引入的。
- `exit` 同 `quit`。
- `go, \g, ;` 将当前查询发送到服务器中并显示其结果。
- `help, \h, ?` 显示描述可用的 mysql 命令的帮助消息。
- `print, \p` 打印当前查询（是查询文本本身，并非执行该查询所得到的结果）。
- `quit, \q` 退出 mysql。
- `rehash, \#` 重新计算数据库、表和列名自动完成所需要的信息。参见关于 `--no-auto-`

rehash 选项的描述。

- `status, \s` 检索和显示服务器的状态信息。如果要想检查服务器的版本和当前数据库, 该选项有用。
- `use db_name, \u db_name` 选择指定的数据库为当前数据库。

E.5 mysql.server

`mysql.server` 是一个脚本, 它通过调用 `safe_mysqld` 启动和停止 `mysqld` 服务器。

用法

```
mysql.server start
mysqlserver stop
```

通常, `mysql.server` 用在 System V 风格的系统中并安装在 `/etc` 目录下的一个运行时目录中。系统通过调用带有 `start` 参数的该脚本在系统引导期间启动服务器。系统通过调用带 `stop` 参数的脚本在系统关闭期间关闭服务器。该脚本还可以用相应的参数手工调用以启动或关闭服务器。

E.6 mysqlaccess

该脚本允许连接到服务器、检索权限信息, 以及测试指定用户权限的结果。它使用 `mysql` 数据库中的 `user`、`db` 和 `host` 表的拷贝来进行 (但 `mysqlaccess` 不能用来试验表或列的权限)。也可以将您在这些临时表中做的改变提交回 `mysql` 数据库的实际表中。

要想使用 `mysqlaccess`, 必须有足够的访问授权表的权限。

E.6.1 用法

```
mysqlaccess [host_name [user_name [db_name]]] options
```

E.6.2 mysqlaccess 支持的标准选项

```
--host --password --user --version
```

E.6.3 mysqlaccess 专有的选项

- `-b, --brief` 以单行格式显示结果。
- `--commit` 将临时授权表拷贝回到 `mysql` 数据库中。然后必须执行 `mysqladmin flush-privileges` 命令, 使服务器注意到这些变化。
- `--copy` 将授权表加载到临时表中。
- `-d db_name, --db=db_name` 指定数据库名。
- `--debug=n` 指定调试级别。`n` 应为 0 到 3 的整数。
- `--howto` 显示某些例子, 说明如何使用 `mysqlaccess`。
- `--old_server` 说明该服务器的版本比 MySQL 3.21 的旧。这样将使 `mysqlaccess` 对发送到服务器的查询做某些调整。
- `--plan` 显示 `mysqlaccess` 未来发行版中将计划增加的功能清单。

- `--preview` 显示实际和临时授权表之间的权限差别。
- `--relnotes` 打印 `mysqlaccess` 发行版的说明。
- `-H host_name, --rhost=host_name` 指定要连接的远程服务器。
- `--rollback` 撤消对临时授权表所做的改变。
- `-P password, --spassword=password` 为 MySQL 超级用户（具有足够的修改授权表的权限的用户）指定口令。
- `-U user_name, --superuser=user_name` 指定 MySQL 超级用户的用户名。
- `-t, --table` 按表格形式显示结果。

E.7 mysqladmin

`mysqladmin` 实用程序与 MySQL 服务器进行通信来执行各种管理操作。可以使用 `mysqladmin` 从服务器的操作中获得信息或控制服务器的操作、设置口令，以及创建或删除数据库。

E.7.1 用法

```
mysqladmin [options] command ...
```

E.7.2 mysqladmin 支持的标准选项

<code>--compress</code>	<code>--host</code>	<code>--port</code>	<code>--user</code>
<code>--debug</code>	<code>--password</code>	<code>--silent</code>	<code>--version</code>
<code>--help</code>	<code>--pipe</code>	<code>--socket</code>	

如果 `mysqladmin` 不能连接到服务器的话，`--silent` 将导致 `mysqladmin` 默默地退出。

E.7.3 mysqladmin 专有的选项

- `-f, --force` 该选项有两个作用。首先，它可使 `mysqladmin` 不对 `db_name` 命令进行确认。其次，当在该命令行上指定多个命令时，`mysqladmin` 试图执行每个命令，即使出现了错误。通常，`mysqladmin` 将在第一个错误出现后退出。
- `--relative` 显示在以 `--sleep` 方式使用时的当前的和以前的值之间的差别。通常，该选项只与 `extended-status` 命令一起工作。
- `-i n, --sleep=n` 以 n 秒的间隔重复执行命令行中指定的命令。
- `-t n, --timeout=n` 在当试图连接到服务器时在超时前等待 n 秒。该选项是在 MySQL 3.22.1 版本中引入的。
- `-w [n], --wait=[n]` 指定不能建立对服务器的连接时的等待和重试次数。如果未指定 n 的值，则缺省值为 1。

E.7.4 mysqladmin 的命令

在命令行的任何参数之后，可以指定一个或多个以下的命令。命令的名字可缩短成一个词头，只要该词头是惟一的即可。例如，`processlist` 可以缩短成 `process` 或 `proc`，但不能是 `p`。

有几个命令等价于 SQL 语句，它们在描述中说明。有关这些 SQL 语句的含义的详细信息，请参阅附录 D。

- **create db_name** 创建指定名称的新数据库。

该命令类似于 SQL 语句 `CREATE DATABASE db_name`。

- **drop db_name** 删除指定名的数据库和在该数据库中的任何表。要小心使用该命令，因为不能使该数据库再恢复。除非给出 `--force` 选项，否则 `mysqladmin` 要对该命令进行确认。

该命令类似于 SQL 语句 `DROP DATABASE db_name`。

- **extended-status** 显示服务器状态变量的名字和值。

该命令类似于 SQL 语句 `SHOW STATUS`。

`extended-status` 命令是在 MySQL 3.22.10 中引入的。

- **flush-hosts** 刷新主机高速缓存。

该命令类似于 SQL 语句 `FLUSH HOSTS`。

- **flush-logs** 刷新（关闭并重新打开）日志文件。

该命令类似于 SQL 语句 `FLUSH LOGS`。

- **flush-privileges** 重新加载授权表。

该命令类似于 SQL 语句 `FLUSH PRIVILEGES`。

- **flush-status** 清除状态变量（它将几个计数器复位）。

该命令类似于 SQL 语句 `FLUSH STATUS`

- **flush-tables** 刷新表的高速缓存。

该命令类似于 SQL 语句 `FLUSH TABLES`。

- **kill id,id, ...** 取消由给定的标识符号指定的服务器线程。如果指定多个号，则 ID 列表应该不包含空格以便它不会与跟在 `kill` 命令后面的另一个命令相混淆。要想查找当前有哪些线程在运行，可使用 `mysqladmin processlist`。

该命令类似于对每个线程 ID 发布的 SQL 语句 `KILL`。

- **password new_password** 当运行 `mysqladmin` 命令时为连接到服务器的 MySQL 用户改变口令。新的口令是 `new_password`。您必须能作为 MySQL 用户（是将要修改口令的用户）先连接到服务器，这起到了对您是否知道旧口令的验证。

该命令类似于 SQL 语句 `SET PASSWORD`。

`password` 命令是在 MySQL 3.22 中引入的。

- **ping** 检查 MySQL 服务器是否在运行。

- **processlist** 显示当前正在运行的服务器线程的列表。

该命令类似于 SQL 语句 `SHOW PROCESSLIST`。

- **refresh** 刷新表的高速缓存并关闭和重新打开日志文件。

- **reload** 重新加载授权表。

该命令类似于 SQL 语句 `FLUSH PRIVILEGES`。

- **shutdown** 指示服务器关闭。

- **status** 显示服务器简短的状态消息。

- **variables** 显示服务器变量的名字和值。

该命令类似于 SQL 语句 `SHOW VARIABLES`。

- **version** 检索并显示服务器版本信息串。

E.8 mysqlbug

MySQL 参考手册的第 2 章包含提交故障报告的详细过程。该过程后面的帮助信息确保您的报告是有用的并包含有足够的信息以解决所报告的问题。该过程的关键部分是使用 `mysqlbug` 脚本。它可用来在您发现有关 MySQL 的问题时创建和发送一个故障报告到 MySQL 邮件清单中。`mysqlbug` 收集有关系统和 MySQL 配置的信息，然后带您进入一个编辑器，其中包含要发送的该邮件消息的内容。编辑该消息以便尽可能地增加有关正在报告的该问题的信息，然后保存该消息的变化并退出编辑器。`mysqlbug` 将询问是否要发送该报告，确认后发出。

请使用 `mysqlbug` 来报告故障，但不要轻易地使用它。在许多情况中，“故障”其实根本就不是故障，或者是一个已经编入参考手册中的某个问题的报告。例如，有一个作为故障被报告了几次的问题是：语句 `DELETE FROM tbl_name` 报告了一个受影响的行值为零。这是事实，但它是预期的行为。它也在参考手册中做了说明，如果您想要得到真正的计数就不应该使用这个查询。MySQL 邮件清单档案是另一个有用的信息来源，可以在 <http://www.mysql.com/doc.html> 找到它们。

用法

```
mysqlbug [address]
```

故障报告缺省时被发送到 MySQL 的邮件清单中。如果您在命令行指定一个电子邮件地址，报告将送至该地址。您可以指定自己的地址来测试 `mysqlbug`，而不是发送报告到邮件清单中。对于首次使用该脚本并且不清楚它是如何工作的您来说，这个主意不坏。

E.9 mysqld

`mysqld` 是 MySQL 服务器。它向客户机程序提供数据库访问，因此，它必须一直运行，否则客户机不能使用由该服务器管理的数据库。当 `mysqld` 启动时，它打开网络端口进行监听，然后等待客户机的连接。`mysqld` 是多线程的并使用一个单独的线程处理每个客户机的连接以提供客户机间的并发性。写给服务器的查询被自动执行，当服务器开始执行像这样的查询时，它将不执行含有该表的其他查询，直到当前查询结束为止。由于这个原因，两个客户机不能同时在表中修改同一行。

E.9.1 用法

```
mysqld [options]
```

E.9.2 mysqld 支持的标准选项

```
--debug      --port      --user
--help       --socket    --version
```

请注意，尽管它支持 `--socket`，但不支持 `-S`，因为服务器使用 `-Sg` 选项，其含义与套接字文件无关。

如果给出 `--user` 选项，它指定运行服务器时要使用的账号的用户名。此时，当服务器启动

时，它从口令文件中查找该用户和该账号组 ID 的值，然后修改其用户和相配的组 ID。这样，服务器将使用与该用户相关的权限进行运行，而非 root 的权限（为了使 --user 选项有效，服务器必须作为 root 启动，否则它将不能改变其用户的 ID）。该选项是在 MySQL 3.22 中引入的。

E.9.3 mysqld 专有的选项

- **-b *path_name*, --basedir=*path_name*** 指定 MySQL 安装目录的路径名。如果它们以相对的路径名给出，则许多其他的路径名被认为是在该目录下的。
- **--big-tables** 通过将所有临时结果保存到磁盘而不是将其保存在内存中，允许处理大的结果集。这样避免了大多数作为内存不足的结果的“table full（表已满）”消息出现，以便存放大的结果集。该选项自 MySQL 3.23 以来不再需要。
- **--bind-address=*ip_addr*** 连接指定的 IP 地址。通常，mysql 与服务器运行的主机的主机缺省 IP 地址连接。如果主机有多个地址的话，该选项可用来选择一个要连接的地址。
- **--bootstrap** 该选项在您首次安装 MySQL 时由安装脚本使用。
- **--chroot=*path_name*** 运行固定到特定的根目录上的 MySQL 服务器。该选项指定该目录的路径名。有关在 chroot()-ed 环境中运行的详细信息，请参阅 chroot() UNIX 人工页。该选项是在 MySQL 3.22.2 中引入的。
- **-h *path_name*, --datadir=*path_name*** 指定到 MySQL 数据目录的路径名。
- **--default-table-type=*type*** 指定缺省的表存储类型。*type* 值应该为 isam、myisam 或 heap（该值不区分大小写的）。如果该选项没有指定，则将缺省使用 MyISAM 表。该选项是在 MySQL 3.23 中引入的。
- **--enable-locking** 对于像 Linux 这样的系统，允许外部锁定（文件系统锁定），缺省为 --skip-locking。该选项是在 MySQL 3.22.4 中引入的。
- **-T [*n*], --exit-info[=*n*]** 在程序终止时打印调试信息。该选项是在 MySQL 3.22 中引入的。
- **--init-file=*file_name*** 指定在启动时要执行的 SQL 语句的文件。
- **-L, --language=*lang_name*** 用指定的语言向客户机显示错误消息。通常，*lang_name* 将是 english 或 german 这样的值。
- **-l [*path_name*], --log[=*path_name*]** 开启常规日志文件。常规日志包含有关客户机连接和查询的信息。如果没有给出 *path_name*，则该日志文件名为 *host_name.log*，这里的 *host_name* 是服务器主机的名字。如果 *path_name* 作为相对路径给出，它将被认为是在数据目录下的。
- **--log-isam[=*file_name*]** 开启索引文件。该选项只用于调试 ISAM/MyISAM 的操作。
- **--log-long-format** 将附加信息写到更新日志中。该选项是在 MySQL 3.22.7 中引入的。
- **--log-update[=*path_name*]** 开启更新日志文件。更新日志包含修改数据库表的任何查询的文本。如果没有给出 *path_name*，则更新文件名为 *host_name.nnn*，这里的 *host_name* 是服务器主机的名字，*nnn* 是比前一个更新日志大 1 的序号。

如果给出 *path_name* 且最后的部分中没有扩展名，则服务器增加一个格式为 *nnn* 的数值扩展名（如前面所描述的）。如果路径名的最后部分中包含扩展名，则该名字作为更新日志文件名使用，而无须修改。如果 *path_name* 以相对路径给出，则它被认为是在数据目录下的。

- `--low-priority-updates` 给予写操作比检索低的优先权。该选项是在 MySQL 3.23 中引入的（从 MySQL 3.22.5 到 MySQL 3.23，它被称为 `--low-priority-inserts`）。
- `-n, --new` 使用新的、可能不安全的例程。这些例程是还未表明稳定性的 MySQL 的功能。该选项只有在您喜欢冒险时使用。
- `-o, --old-protocol` 使用 MySQL 3.21 以前的客户机/服务器通信使用的协议。该选项在服务器与版本很旧的客户机程序进行通信时可能需要。
- `--one-thread` 使用单个线程运行，用于在 Linux 中的调试，因为通常最少使用三个线程。该选项是在 MySQL 3.22.2 中引入的。
- `--pid-file=path_name` 当 mysql 启动时，它将其进程的 ID (PID) 写入文件中。该选项指定 PID 文件的路径名。该文件可以由确定服务器进程号的其他进程使用，尤其是为发送一个信号的目的。例如，mysql.server 在它向服务器发送一个关闭信号时读取该文件。
- `--safe-mode` 该选项类似于 `--skip-new`，但禁止更多的东西。如果 MySQL 出现不稳定或复杂的查询产生不正确的结果时可试一试该选项。如果使用该选项提高服务器的操作，则在使用 mysqlbug 报告您遇到的问题时应注释该情况。
- `-Sg, --skip-grant-tables` 不使用授权表检验客户机的连接。该选项给予任何客户机做任何事情的所有访问权。通过发布 mysqladmin flush-privileges 命令可以指示服务器开始使用授权表。
- `--skip-host-cache` 禁止使用主机名高速缓存。
- `--skip-locking` 禁止使用系统锁定（文件系统锁定）。如果使用该选项，则必须在使用 isamchk 或 myisamchk 修复表之前关闭服务器，否则将有毁坏这些表的危险。
- `--skip-name-resolve` 不解决主机名。如果指定该选项，授权表必须由 IP 号或以 localhost 指定主机
- `--skip-networking` 不允许 TCP/IP 的连接。客户机必须使用 localhost 从本地主机中连接。
- `--skip-new` 不使用新的、可能不安全的例程。参见 `--new`。
- `--skip-show-database` 不允许未授权用户发布 SHOW DATABASES 查询或在用户没有任何访问权的数据库上使用 SHOW TABLES。该选项时在 MySQL 3.23 中引入的。
- `--skip-thread-priority` 通常，修改表的查询比检索数据的查询更优先。如果不希望这样，该选项使得服务器将不给予不同类型的查询以不同的优先级。
- `-t path_name, --tmpdir=path_name` 指定临时文件使用的目录路径名。该选项是在 MySQL 3.22.4 中引入的。

Windows 专有的选项如下：

- `--console` 为错误消息显示一个控制台窗口。该选项是在 MySQL 3.22.4 中引入的。
- `--flush` 在每次更新后刷新磁盘上的所有表。这减少了崩溃事件中表讹误的风险，但严重降低了性能。该选项是在 MySQL 3.22.9 中引入的。
- `--install` 安装作为服务的 mysqld-nt（只对 Windows NT）。
- `--remove` 删除作为服务的 mysqld-nt（只对 Windows NT）。
- `--standalone` 启动作为独立程序的而非作为服务的 mysqld-nt（只对 Windows NT）。

E.9.4 mysqld 的变量

要查看 mysql 将使用哪些缺省值，可用下列命令：

```
% mysqld --help
```

要查看当前正运行的 mysqld 使用了哪些值，可用以下命令：

```
% mysqladmin variables
```

通过发布 SHOW VARIABLES 查询还可以检查当前变量的值。能使用 --set-variable 选项设置的 mysqld 变量如下：

- **back_log** 未决的连接请求数量，这些请求可能在处理当前连接时排队等待。
- **connect_timeout** 在初始连接信号交换时 mysql 等待报文分组的秒数。
- **delayed_insert_timeout** 当 INSERT DELAYED 操作的处理完成插入排队的行时，它等待该变量值这么多秒钟，看看是否有任何新的 INSERT DELAYED 行到达。如果有，它将处理它们，否则终止。
- **delayed_insert_limit** 来自 INSERT DELAYED 语句中的行数，这些行在被插入到表中之前检查是否有针对该表的任何新的 SELECT 语句到达。如果有，则插入操作暂停，让检索操作执行。
- **delayed_queue_size** 为 INSERT DELAYED 语句排队等待的行数。如果队列满，将阻塞后面的 INSERT DELAYED 语句，直到队列中有空间为止。
- **flush_time** 如果该变量有非零值，则关闭表，每 flush_time 秒向磁盘刷新一次未决变化。Windows 的缺省值为 30 秒；使用 --flush 选项可以改变该值。
- **join_buffer_size** 全连接缓冲区的尺寸（即不使用索引来完成的连接的缓冲区）。在 MySQL 3.23 之前该变量称为 join_buffer。
- **key_buffer_size** 索引块使用的缓冲区的大小。该缓冲区在连接处理程序之间共享。该选项在 MySQL 3.23 之前称为 key_buffer。
- **long_query_time** 定义一个“慢”查询的秒数。任何花费的时间比该值长的查询将引起 Slow_queries 计数器增加。
- **max_allowed_packet** 服务器和客户机之间进行通信所使用的缓冲区尺寸的最大值。分配给该缓冲区的初始值为 net_buffer_length 字节，但可根据需要增长到最多 max_allowed_packet 字节。
- **max_connections** 所允许的同时发生的客户机连接的最大数量。
- **max_connect_errors** 在阻塞主机后面的连接请求之前允许的该主机不成功连接的最大次数。使用这个变量是因为某人可能打算从该主机中闯入。FLUSH HOST 语句或 mysqladmin flush-host 命令可以用来清除该主机的高速缓存以便重新使阻塞的主机有效。
- **max_delayed_insert_threads** 将要启动处理 INSERT DELAYED 语句的处理程序的最大个数。在已经到达处理程序最大值的情况下被接收的任何这样的语句将认为是非 DELAYED 语句。
- **max_join_size** MySQL 优化器估计一个连接将返回多少行。如果该估计超过 max_join_size 行，将返回错误。如果用户喜欢编写其返回的行数无规律的 SELECT 语句，则该变量会有帮助。
- **max_sort_length** 使用每个值的第一个 max_sort_length 字节存储 BLOB 或 TEXT 的值。
- **net_buffer_length** 服务器和客户机之间进行通信所使用的缓冲区大小的初始值。该缓

缓冲区最大可扩充到 `max_allowed_packet` 字节。

- `record_buffer` 执行连续表扫描的线程所使用的缓冲区大小。
- `sort_buffer` 执行排序操作的线程（`GROUP BY` 或 `ORDER BY`）所使用的缓冲区的大小。每个线程一个值。通常，如果您有许多同时进行排序的线程，则使该值太大是不明智的（超过 1MB）。
- `table_cache` 可打开的表的最大数量。该高速缓存在线程之间共享。
- `tmp_table_size` 临时表所允许的最大字节数。如果表超过这个大尺寸，`mysqld` 将向客户机返回一个错误消息。
- `thread_stack` 每个线程的堆栈大小。
- `wait_timeout` 在服务器关闭一个连接之前该保持连接空闲的秒数。

E.10 mysqldump

`mysqldump` 程序将数据库表的内容写到文本文件中。这些文件可用于各种目的，如数据库备份、移动数据库到另一个服务器中，或基于已有数据库的内容建立测试数据库。

在缺省设置时，每个转储表的输出结果由重新创建该表的 `CREATE TABLE` 语句组成，后跟一组重新加载该表的内容的 `INSERT` 语句。如果指定 `--tab` 选项，表的内容将以行的形式写入数据文件，并且创建表的 `SQL` 语句写入一个单独的文件中。

E.10.1 用法

```
mysqldump [options] db_name [tbl_name] ...
```

如果没有指明跟在数据库名 `db_name` 后的任何表，则该数据库中的所有表都进行转储。否则只转储指定的表。

使用 `mysqldump` 最常用的方法如下：

```
% mysqldump --opt db_name > backup_file
```

E.10.2 mysqldump 支持的标准选项

<code>--compress</code>	<code>--host</code>	<code>--port</code>	<code>--user</code>
<code>--debug</code>	<code>--password</code>	<code>--set-variable</code>	<code>--verbose</code>
<code>--help</code>	<code>--pipe</code>	<code>--socket</code>	<code>--version</code>

E.10.3 mysqldump 专有的选项

下列选项控制怎样操作 `mysqldump`。下一节“数据格式选项”将介绍可与 `--tab` 选项组合使用来指定数据文件格式的选项。

- `--add-drop-table` 在每个 `CREATE TABLE` 语句之前增加 `DROP TABLE IF EXISTS` 语句。
- `--add-locks` 在加载每个表数据的 `INSERT` 语句组前后增加 `LOCK TABLE` 和 `UNLOCK TABLE`。
- `--allow-keywords` 允许创建是关键字的列名。
- `-c, --complete-insert` 使用 `INSERT` 语句指明要插入的每个列。
- `--delayed-insert` 写 `INSERT DELAYED` 语句。
- `-e, --extended-insert` 写插入多行的 `INSERT` 语句。

- `-F, --flush-logs` 在开始转储前刷新服务器日志文件。
- `-f, --force` 继续，即使有出现错误。
- `--full` 增加对 `mysqldump` 产生的 `CREATE TABLE` 语句的附加信息，如表的类型、起始的 `AUTO_INCREMENT`，等等。这是您可以在 `CREATE TABLE` 语法的 `table_options` 部分指定的信息（参见附录 D）。
- `-l, --lock-tables` 在转储表之前得到对所有表的锁。
- `-t, --no-create-info` 不写 `CREATE TABLE` 语句。
- `-d, --no-data` 不写表数据。
- `--opt` 优化表的转储速度并写转储文件，该文件的重新加载速度是最佳的。该选项开启 `--add-drop-table`、`--add-locks`、`--all --extended-insert`、`--quick` 和 `--lock-tables`。
- `-q, --quick` 在缺省时，`mysqldump` 将表的全部内容读到内存中，然后再进行写操作。该选项使每行（只要一从服务器中读出来）被写到输出文件中。
如果使用该选项，不应该挂起 `mysqldump`；这会导致服务器等待，干扰其他的客户机。
- `-T dump_dir, --tab=dump_dir` 该选项使 `mysqldump` 对每个表写两个文件，文件的位置为 `dump_dir`。该目录必须是已经存在的。对于每个表 `tbl_name`，写文件 `dump_dir/table_name.txt` 和 `dump_dir/table_name.sql`，前者包含该表的数据，后者是该表的 `CREATE TABLE` 语句。使用该选项必须具有 `FILE` 的权限。

为了避免混淆，当使用该选项时最好在服务器主机上运行 `mysqldump`。否则，有些文件被写在服务器上而另一些则写在客户机中。对于 `*.sql` 文件，`dump_dir` 用于服务器主机中，对于 `*.txt` 文件，`dump_dir` 用于客户机。`*.sql` 文件将属于用来运行服务器的账号，`*.txt` 文件将属于您自己。

缺省设置时，数据文件包含换行终止（`newline-terminated`）格式的行的，由制表符分隔的列值组成。该格式可以用“数据格式选项”中介绍的选项改变。

- `-w where_clause, --where=where_clause` 只转储通过由 `where_clause` 指定的 `WHERE` 子句选择的记录。应该用引号将该子句括起来，以防止外壳程序将其解释为多个命令行的参数。该选项是在 MySQL 3.22.7 中引入的。

E.10.4 数据格式选项

如果指定 `--tab` 或 `-T` 选项为每个表生成单独的数据文件，则有几个附加的选项。您可能需要用适当的引用字符将该选项的值括起来。这些选项类似于 `LOAD DATA` 语句的数据格式选项。参见附录 D 的 `LOAD DATA` 条目。

- `--fields-enclosed-by=char` 指明列值应括在指定的字符中，通常是引号字符。缺省时列值不括在任何字符中。
- `--field-escaped-by=char` 为转义特殊的字符指定转义符。缺省时无转义符。
- `--fields-optionally-enclosed-by=char` 表明列值应括在指定的字符中，通常使用引号字符。该字符用于非数值的列。缺省时列值不用任何字符括起。
- `--fields-terminated-by=char` 指定文件所使用的列值分隔符。缺省设置时，列值由制表符分隔。
- `--lines-terminated-by=str` 指定表示输出结果行末尾的串（可以是多字符的）、缺省时用换行符。

E.10.5 mysqldump 的变量

以下 mysqldump 变量可用 --set-variables 选项设置：

- max_allowed_packet 用于服务器和客户机之间通信的缓冲区尺寸的最大值。
- net_buffer_length 用于服务器和客户机之间通信的缓冲区的初始大小。该缓冲区最大可扩充到 max_allowed_packet 字节。

E.11 mysqlimport

mysqlimport 实用程序是一个批量加载器，它将文本文件的内容读到已有表中。它对 LOAD DATA 的 SQL 语句起到命令行接口的作用，是将行输入到表中的有效方法。

E.11.1 用法

```
mysqlimport [options] db_name file_name ...
```

db_name 参数指定数据库，该数据库中包含您要加载数据的表。要加载数据的表由文件名参数确定。对于每个文件名，去除其任何扩展名，留下的基名就作为要将文件加载进来的表名。例如，mysqlimport 将加载 president.txt 的内容到 president 表中。

E.11.2 mysqlimport 支持的标准选项

--compress	--host	--port	--user
--debug	--password	--silent	--verbose
--help	--pipe	--socket	--version

E.11.3 mysqlimport 专有的选项

下列选项控制 mysqlimport 如何处理输入文件。下一节“数据格式选项”将介绍用来指定输入文件数据格式的选项。

- -d, --delete 在加载任何数据到表中之前清空每个表。
- -f, --force 继续加载行，即使出现错误。
- -i, --ignore 当一个输入行包含某个唯一键值，且该值已存在于表中，则保留已有的行并丢弃该输入行。
- -L, --local 缺省时，mysqlimport 允许服务器读数据文件，这意味着文件必须在服务器主机中加载，并且您必须有 FILE 权限。指定 --local 选项将指示 mysqlimport 读取数据文件本身并将它发送到服务器中。如果您没有 FILE 权限，在另一台机器上运行 mysqlimport 与在服务器主机上相比，以及包括在服务器主机上，该选项尽管工作但较慢。
--local 选项是在 MySQL3.22.15 中引入的
- -l, --lock-tables 在加载数据到表中以前锁定每个表。
- -r, --replace 当一个输入行包含一个唯一键值，且该值已经存在于表中，则可用输入行替换已有的行。

E.11.4 数据格式选项

缺省时，mysqlimport 假定数据文件包含换行终止行，并由制表符分隔的值组成。可以使

用下列选项更改成希望的格式。可能需要用适当的引用符号将该选项值括起。这些选项类似于 LOAD DATA 语句的数据格式选项。参见附录 D 的 LOAD DATA 条目。

- `--fields-enclosed-by=char` 指明列值应括在指定的字符中，通常用引号。缺省时，假定列值不括在任何字符中。
- `--fields-escaped-by=char` 指定用于转义特殊字符的转义符。缺省表示无转义符。
- `--fields-optionally-enclosed-by=char` 表示列值应括在指定字符中，通常是引号。
- `--fields-terminated-by=char` 指定分隔列值的字符。缺省时，假定列值由制表符分隔。
- `--lines-terminated-by=str` 指定结束输入行的串（可以是多字符的）。缺省时假定由换行符终止行。

E.12 mysqlshow

mysqlshow 显示数据库、数据库中的表，或有关表中列或索引的信息。它担当 SHOW SQL 语句的命令行接口的作用。

E.12.1 用法

```
mysqlshow [options] [db_name [tbl_name [col_name]]]
```

如果没有指定数据库名，mysqlshow 将列出服务器主机中所有数据库。如果指定数据库名但未指定表名，则列出该数据库中所有的表。如果数据库和表名都被指定，但未指定列名，则列出该表的所有列。如果所有的名字都被指定，mysqlshow 将显示有关指定列的信息。

如果最后的参数中包含外壳程序通配符（‘*’或‘?’），输出结果将被限制为与该通配符相配的值。‘*’和‘?’被视为 LINK 操作符的‘%’和‘_’SQL 通配符。

E.12.2 mysqlshow 支持的标准选项

<code>--compress</code>	<code>--host</code>	<code>--port</code>	<code>--version</code>
<code>--debug</code>	<code>--password</code>	<code>--socket</code>	
<code>--help</code>	<code>--pipe</code>	<code>--user</code>	

E.12.3 mysqlshow 专有的选项

- `-i, --status` 显示由 SHOW TABLE STATUS 语句显示的相同类型的表的信息。
`--status` 选项是在 MySQL 3.23 中引入的。
- `-k, --keys` 显示除表列信息外的表索引信息。该选项只在指定表名时才有意义。

E.13 safe_mysqld

safe_mysqld 是启动 mysql 服务器和监控程序的一个脚本。如果服务器死机，safe_mysqld 重新启动它。

用法

safe_mysqld 理解与 mysql 相同的选项。

附录F C API 参 考

本附录介绍 MySQL 客户机库的C语言应用程序编程接口。API 由一组函数以及一组用于函数的数据类型组成，这些函数与 MySQL 服务器进行通信并访问数据库。

本附录是一个参考，因此它只包括简要的代码段来说明客户机库的使用。有关完整的客户机程序和编写说明，请参阅第 6 章。

F.1 编译和连接

在源程序级，客户机库定义在 `mysql.h` 头文件中，在源文件中可包括：

```
#include <mysql.h>
```

为了指示编译器在哪里查找该文件，需要指定 `-I/path/to/include/dir` 选项。例如，如果您的 MySQL 头文件被安装在 `/usr/include/mysql` 或 `/usr/local/mysql/include` 中，可以通过使用以下的命令编译源文件：

```
% gcc -I/usr/include/mysql -c myprog.c
% gcc -I/usr/local/mysql/include -c myprog.c
```

如果需要使用其他的 MySQL 头文件，它们应放在与 `mysql.h` 相同的目录中。例如，`mysql_com.h` 包含解释查询结果元数据的常量和宏指令（`mysql_com.h` 通过 `mysql.h` 来包含，因此，尽管可能会直接查看该文件中的内容，但通过包括 `mysql.h` 您可以简单地访问它）。头文件 `errmsg.h` 和 `mysqld_error.h` 中包含错误代码的常量。

在目标级，客户机库作为 `mysqlclient` 库被提供。为了将该库连接到程序中，需要在连接命令中指定 `-lmysqlclient`。您或许还需要用 `-L/path/to/lib/dir` 选项指示连接器在哪里查找该库。例如：

```
% gcc -o myprog myprog.o -L/usr/lib/mysql -lmysqlclient
% gcc -o myprog myprog.o -L/usr/local/mysql/lib -lmysqlclient
```

F.2 C API 数据类型

MySQL 客户机库的数据类型被设计来表示在与服务器的会话期间您所处理的实体。有针对连接本身的、查询结果的、结果中的一行的以及元数据（描述有关构成列结果的信息）的类型。

请注意，在以下讨论中，术语“列”和“字段”是同义词。

F.2.1 标量数据类型

MySQL 的标量数据类型表示像超大整数、逻辑值和字段偏移量这样的值。

■ `my_ulonglong` 长整型，用于函数的返回值，如 `mysql_affected_rows()`、`mysql_num_rows()` 和 `mysql_insert_id()`，这些函数都返回行计数或其他可能的大数值。

为了打印 `my_ulonglong` 值，可以定义它为 `unsigned long`（无符号长整型）并使用 ‘`%lu`’ 的格式。如果不这样做，在某些系统中该值将不能正确地打印。

■ `my_bool` 逻辑型，用于 `mysql_change_user()` 和 `mysql_eof()` 的返回值。

- **MYSQL_FIELD_OFFSET** 该数据类型用于 `mysql_field_tell()` 和 `mysql_field_seek()` 函数，表示当前结果集中 **MYSQL_FIELD** 结构集合中的偏移量。

F.2.2 非标量数据类型

MySQL 的非标量类型表示结构或数组。所有的 **MYSQL** 和 **MYSQL_RES** 结构都被认为是一个“黑匣子”。换言之，您只能引用结构本身，而不能引用结构中的元素。**MYSQL_ROW** 和 **MYSQL_FIELD** 类型则没有这样的限制，您可以直接访问它们来获得作为查询结果返回的数据和元数据。

- **MYSQL** 主要的客户机库类型是 **MYSQL** 结构的，它用于连接处理程序。处理程序中包含有关服务器的连接状态的信息。初始化一个 **MYSQL** 结构，然后经过该结构到一个连接，来打开服务器的会话。在建立该连接后，可以使用该处理程序发布查询、生成结果集、获得错误信息，等等。

■ **MYSQL_RES** 像 **SELECT** 或 **SHOW** 这样的查询借助于一个结果集将数据返回给客户机，这个结果集是用 **MYSQL_RES** 结构表示的。该结构中包含由查询返回的行的信息。在得到结果集后，可以调用 API 函数得到结果集数据（集合的每行中的数据值）或元数据（关于该结果的信息，如由多少列、类型和长度等等）。

- **MYSQL_ROW** **MYSQL_ROW** 类型包含一个数据行的值，表示为计数字节串（counted byte）的数组。所有值（甚至是数字）都以串的形式返回，除了当行中的值为 **NULL** 时它通过 **CNULL** 指针以 **MYSQL_ROW** 结构表示外。

行中值的数目由 `mysql_num_fields()` 指定。行中 *i*-th 列值由 `row[i]` 指定，这里 *i* 的范围从 0 到 `mysql_num_fields(res_set)-1` (`res_set` 是指向 **MYSQL_RES** 结果集的一个指针)。请注意，**MYSQL_ROW** 类型已经是一个指针，因此可以这样声明一个行变量：

```
MYSQL_ROW row; /*正确*/
```

但不能这样声明：

```
MYSQL_ROW *row; /*不正确*/
```

MYSQL_ROW 类型的值有终结的空值（terminating null），因此，非二进制值可以认为是空终结串。但是，包含二进制数据的数据值可以在内部包含空字节并且被看作是计数串。为了得到行中值的长度，这样调用 `mysql_fetch_lengths()`：

```
unsigned long *length;  
length = mysql_fetch_lengths (res_set);
```

对于 **NULL** 值来说，`mysql_fetch_lengths()` 返回的长度为零。

- **MYSQL_FIELD** 客户机库使用 **MYSQL_FIELD** 结构表示关于该结果集中列的元数据——每列一个结构。**MYSQL_FIELD** 结构的数目由 `mysql_num_fields()` 指定。您可以通过 `mysql_num_field()` 访问连续的字段结构，或用 `mysql_field_tell()` 和 `mysql_field_seek()` 在结构之间来回移动。

MYSQL_FIELD 结构对于显示或解释数据行的内容很有用。它类似于：

```
typedef struct st_mysql_field {  
    char *name;  
    char *table;  
    char *def;
```

```
enum enum_field_types type;
unsigned int length;
unsigned int max_length;
unsigned int flags;
unsigned int decimals;
} MYSQL_FIELD;
```

MYSQL_FIELD 结构的成员由如下含义：

- **name** 列名，为空终结串。如果该列值作为一个表达式的结果被计算，则 **name** 是以串形式表示的该表达式。如果给列或表达式一个列的别名，则 **name** 是该别名。例如，下列的查询产生 “my_col”、“4*(my_col+1)” 和 “mc” 的 **name** 值：

```
SELECT my_col, 4*(my_col+1), my_col AS mc ...
```

- **table** 列所在的表的名字，为空终结串。如果列值作为表达式的值被计算，则 **table** 是空串。例如，如果发布类似下面的查询，第一列 (my_col) 的表名是 my_tbl，而第二列 (my_col+0) 的表名为空串：

```
SELECT my_col, my_col+0 FROM my_tbl ...
```

- **def** 列的缺省值，为空终结串。MYSQL_FIELD 结构的该成员仅在调用 mysql_list_fields() 时设置，其他情况为 NULL（列缺省值可以通过发布 DESCRIBE tbl_name 或 SHOW FIELDS FROM tbl_name 查询和检查该结果集来获得）。
- **type** 列的类型。作为表达式结果被计算的列的类型从该表达式元素的类型中确定。例如，如果 my_col 是 VARCHAR(20) 列，则 **type** 是 FIELD_TYPE_VAR_STRING，而 LENGTH(my_col) 的 **type** 为 FIELD_TYPE_LONGLONG。

可能的 **type** 值列在 mysql_com.h 中并在表F-1 中示出。

表F-1 MYSQL_FIELD 的 type 成员值

type 值	列 类 型
FIELD_TYPE_BLOB	BLOB 或 TEXT
FIELD_TYPE_DATE	DATE
FIELD_TYPE_DATETIME	DATETIME
FIELD_TYPE_DECIMAL	DECIMAL 或 NUMERIC
FIELD_TYPE_DOUBLE	DOUBLE 或 REAL
FIELD_TYPE_ENUM	ENUM
FIELD_TYPE_FLOAT	FLOAT
FIELD_TYPE_INT24	MEDIUMINT
FIELD_TYPE_LONG	INT
FIELD_TYPE_LONGLONG	BIGINT
FIELD_TYPE_NULL	NULL
FIELD_TYPE_SET	SET
FIELD_TYPE_SHORT	SMALLINT
FIELD_TYPE_STRING	CHAR
FIELD_TYPE_TIME	TIME
FIELD_TYPE_TIMESTAMP	TIMESTAMP
FIELD_TYPE_TINY	TINYINT
FIELD_TYPE_VAR_STRING	VARCHAR
FIELD_TYPE_YEAR	YEAR

您可能在旧的源文件中看到对 FIELD_TYPE_CHAR 的引用，它是一字节的类型，现在称 FIELD_TYPE_TINY。

■ length 列的长度，该长度在创建该表的 CREATE TABLE 语句中指定。作为表达式结果被计算的列的长度从该表达式的元素中确定。

■ max_length 实际显示在结果集中的最长的列值长度。例如，如果结果集中的列包含值“Bill”、“Jack”和“Belviders”，则该列的 max_length 值为 9。

由于 max_length 值只在所有行已经知晓后才能确定，因此它只对用 mysql_store_result() 创建的结果集有意义。对于用 mysql_use_result() 所创建的结果集，max_length 为 0。

■ flags flags 成员指定列的属性。属性由 flags 值中的单个的比特位表示。这些比特位可以通过表 F-2 中的比特掩码常量进行测试。例如，为了确定列值是否为 UNSIGNED，应测试 flags 的值：

```
if (field -> flags & UNSIGNED_FLAG)
    printf ( " %s values are UNSIGNED \n " , field -> name);
```

表F-2 MYSQL_FIELD 的 flags 成员值

flag 值	含 义
AUTO_INCREMENT_FLAG	列具有 AUTO_INCREMENT 属性
BINARY_FLAG	列具有 BINARY 属性
MULTIPLE_KEY_FLAG	列是NON-UNIQUE 索引的一部分
NOT_NULL_FLAG	列不能包含 NULL 值
PRI_KEY_FLAG	列是 PRIMARY KEY 的一部分
UNIQUE_KEY_FLAG	列是 UNIQUE 索引的一部分
UNSIGNED_FLAG	列具有 UNSIGNED 属性
ZEROFILL_FLAG	列具有 ZEROFILL 属性

有些 flags 常量指定列的类型而非列属性，但现在它们不受欢迎，因为可使用 field->type 确定列的类型。表 F-3 列出这些不受赞成的常量。

表F-3 不赞成的 MYSQL_FIELD 的 flags 成员值

flags 值	含 义
BLOB_FLAG	列包含 BLOB 值
ENUM_FLAG	列包含 ENUM 值
SET_FLAG	列包含 SET 值
TIMESTAMP_FLAG	列包含 TIMESTAMP 值

■ decimals 数值列的十进制小数位，非数值列的 decimals 为零。例如，DECIMAL (8,3) 列的 decimals 值为 3，而 BLOB 列为 0 值。

F.2.3 存取器宏指令

mysql.h 包含一些宏指令，它们允许您更方便地测试 MYSQL_FIELD 成员。IS_NUM() 测试 type 成员，其他的测试 flags 成员。

如果列值有一个数值类型，则 IS_NUM() 为真（非零）：

```
if (IS_NUM (field->type))
    printf ("Field %s is numeric\n", field->name);
```

如果列是主键的一部分，则 IS_PRI_KEY() 为真：

```
if (IS_PRI_KEY (field->flags))
    printf ("Field %s is part of primary key\n", field->name);
```

如果列不包含 NULL 值，则 IS_NOT_NULL() 为真：

```
if (IS_NOT_NULL (field->flags))
    printf ("Field %s values cannot be NULL\n", field->name);
```

如果列是 BLOB 或 TEXT，则 IS_BLOB() 为真。但是，该宏指令测试了不受赞成的 flags 成员的 BLOB_FLAG 比特位，因此 IS_BLOB() 也不受欢迎。

F.3 C API 函数

客户机库函数可以分为以下几类：

- 连接管理例程，用于建立和终止对服务器的连接。
- 状态和错误报告例程，用于获得错误代码和消息。
- 查询构造和运行例程，用于构造查询并发送它们到服务器中。
- 结果设置处理例程，用于处理来自返回数据的查询的结果。
- 信息例程，用于提供有关客户机、服务器、协议版本和当前连接的信息。
- 管理例程，用于控制服务器的操作。
- 调试例程，用于产生调试信息。
- 考虑废除的不赞成的例程。

每个类别中的客户机库函数将在以下小节中详细介绍，并按字母顺序列出。某些参数重复出现在一些函数的描述中并具有标准的含义：

- conn 是一个指向服务器连接的 MYSQL 连接处理程序的指针。
- res_set 是一个指向 MYSQL_RES 结果集结构的指针。
- field 是一个指向 MYSQL_FIELD 列的信息结构的指针。
- row 是结构集的一个 MYSQL_ROW 数据行。

简单地说，这些参数将不在函数的描述中再讨论，您可以认为它们的含义是刚才指定的。

除非另有说明，否则可以假定任何给出的函数早在 MySQL 3.21.10 时就出现在客户机库中。

F.3.1 连接管理例程

这些函数允许建立和终止对服务器的连接，设置影响连接建立的选项，以及重新建立超时的连接。

■ my_bool

```
mysql_change_user (MYSQL *conn, char *user_name, char *password, char
*db_name);
```

修改由 conn 所指定的连接中的用户和缺省的数据库。对于不包括数据库说明符的表的引

用来说，数据库是缺省的。如果 db_name 为 NULL，则不选择任何缺省数据库。

如果允许用户对该服务器的连接，则 mysql_change_user() 返回真值，并且，如果指定数据库，则用户具有访问该数据库的许可权。否则该函数失败且当前的用户和数据库仍然保持不变。

mysql_change_user() 是在 MySQL 3.23.3 中引入的。

■ void

mysql_close (MYSQL *conn);

关闭由 conn 指定的连接。当完成一个服务器会话时调用该例程。如果该连接处理程序由 mysql_init() 自动分配，则 mysql_close() 解除该分配。

如果打开一个连接的企图失败，则不要调用 mysql_close()。

■ MYSQL *

mysql_init (MYSQL *conn);

初始化一个连接处理程序并返回指向它的指针。如果 conn 指向一个已有的 MYSQL 结构，则 mysql_init() 初始化该处理程序并返回其地址：

```
MYSQL conn_struct, *conn;
conn = mysql_init (&conn_struct);
```

如果 conn 为 NULL，则 mysql_init() 分配一个新的处理程序，对它进行初始化，并返回其地址：

```
MYSQL *conn;
conn = mysql_init (NULL);
```

如果 mysql_init() 失败，它会返回 NULL。如果 mysql_init() 不能分配一个新的处理程序，这种情况可能会发生。

如果 mysql_init() 分配该处理程序，则 mysql_close() 在您关闭连接时自动解除它。

mysql_init() 是在 MySQL 3.22.1 中引入的。

■ int

mysql_options (MYSQL *conn, enum mysql_option option, char *arg);

该函数允许您制作比单独使用 mysql_real_connect() 时可能的更精确的连接性能。应在 mysql_init() 后以及 mysql_real_connect() 前调用它。如果想要设置几个选项，可以多次调用 mysql_options。

option 参数指定您要设置哪个连接选项。设置该选项的附加功能（如果有的话）由 arg 参数指定。（请注意 arg 是一个指针）如果不需要任何附加信息，则 arg 为 NULL。

以下选项是可用的：

■ MYSQL_INIT_COMMAND 指定在连接到服务器后要执行的查询。arg 是包含该查询的空终结串。该查询也在重新连接后运行（例如，如果调用 mysql_ping()）。由该查询返回的任何结果集都被丢弃。

■ MYSQL_OPT_COMPRESS 指定连接应该使用的压缩客户机/服务器协议（如果服务器支持该协议）。arg 为 NULL。

它还可能在调用 mysql_real_connect() 时指定压缩。

■ MYSQL_OPT_CONNECT_TIMEOUT 指定连接超时（以秒为单位）。arg 是指向一个

包含超时值的 unsigned int 的指针。

- **MYSQL_OPT_NAMED_PIPE** 说明对服务器的连接应该使用指定的管道。arg 为 NULL。该选项只针对 Windows 95/98/NT 的客户机并且只针对 Windows NT 服务器的连接。
- **MYSQL_READ_DEFAULT_FILE** 指定一个选项文件以读取连接参数。选项从文件的 [client] 组中读取。
- **MYSQL_READ_DEFAULT_GROUP** 指定要从选项文件中读取的组，该文件用 **MYSQL_READ_DEFAULT_FILE** 命名。读取除 [client] 组外的指定组。如果没有指定任何选项文件，则客户机库将查找标准选项文件并读它们。

如果多次调用 `mysql_options()` 来设置一个指定的选项，则 `mysql_real_connect()` 使用为该选项最近设置的指定值。

`mysql_options()` 运行成功时返回零，如果 option 值未知则返回非零值。

下列选项可用在选项文件中（在 [client] 组或在用 **MYSQL_READ_DEFAULT_GROUP** 选项指定的组中）：

```
compress
database=db_name
debug
host=host_name
init-command=query
password=your_password
pipe
port=port_num
return-found-rows
socket=socket_name
timeout=seconds
user=user_name
```

下例中的 `mysql_options()` 调用使用压缩协议和 10 秒钟的超时设置连接选项，以便使 `mysql_real_connect()` 与指定的管道连接。它读取 `c:\my.cnf.special` 中的 [client] 和 [mygroup] 组的信息。在该连接建立时，执行 `SET SQL_BIG_TABLES` 语句。

```
MYSQL *conn;
unsigned int timeout;

if ((conn = mysql_init (NULL)) == NULL)
    ... deal with error ...

timeout = 10;
mysql_options (conn, MYSQL_OPT_CONNECT_TIMEOUT, (char *) &timeout);
mysql_options (conn, MYSQL_OPT_COMPRESS, NULL);
mysql_options (conn, MYSQL_OPT_NAMED_PIPE, NULL);
mysql_options (conn, MYSQL_READ_DEFAULT_FILE, "C:\my.cnf.special");
mysql_options (conn, MYSQL_READ_DEFAULT_GROUP, "mygroup");
mysql_options (conn, MYSQL_INIT_COMMAND, "SET SQL_BIG_TABLES=1");
if (mysql_real_connect (conn, ...) == NULL)
    ... deal with error ...
```

`mysql_options()` 是在 MySQL 3.22.1 中引入的。

MYSQL_INIT_COMMAND、**MYSQL_READ_DEFAULT_FILE** 和 **MYSQL_READ_DEFAULT_GROUP** 是在 MySQL 3.22.10 中引入的。

■ int

```
mysql_ping (MYSQL *conn);
```

检查由 conn 指定的连接是否仍在。如果不在，mysql_ping() 使用与建立初始连接所使用的相同的参数重新连接。因而，在没有成功地调用 mysql_real_connect() 之前不应该调用 mysql_ping()。

mysql_ping() 是在 MySQL3.22.1 中引入的。

■ MYSQL *

```
mysql_real_connect (MYSQL *conn, char *host_name,
                    char *user_name, char
                    *password, char *db_name,
                    unsigned int port_num, char
                    *socket_name,
                    unsigned int flags);
```

连接到服务器并返回一个指向该连接处理程序的指针。conn 应该是指向一个已有连接处理程序的指针，该处理程序由 mysql_init() 初始化。该处理程序的地址是连接成功时的返回值。如果出现错误则返回 NULL。

如果该连接失败，您可以传递 conn 处理程序的值到 mysql_errno() 和 mysql_error() 中，以获得错误信息。但是，不应该传递 conn 值到可能影响成功连接的任何其他的客户机库例程中。

host_name 是要连接的服务器。表 F-4 示出该客户机试图如何对 UNIX 和 Windows 客户机的不同的 host_name 值进行连接（如果使用 UNIX 套接字或指定的管道进行连接，则 socket_name 参数指定该套接字或管道名）。

表F-4 基于服务器主机名类型的客户机连接类型

主机名值	UNIX 连接类型	Windows 连接类型
主机名	对指定主机的 TCP/IP 连接	对指定主机的 TCP/IP 连接
IP 号	对指定主机的 TCP/IP 连接	对指定主机的 TCP/IP 连接
localhost	对本地主机的 UNIX 套接字连接不使用	对本地主机的 TCP/IP 连接对本地主机的指定的管道连接
NULL	对本地主机的 UNIX 套接字连接	对本地主机（除了 Windows NT）的 TCP/IP 连接，在退到 TCP/IP 前首先尝试指定的管道连接

user_name 是 MySQL 的用户名。如果它为 NULL，则客户机库发送一个缺省名。在 UNIX 中，该缺省值为您的注册名。在 Winodws 中，该缺省值是像在 USER 环境变量中所指定的名字（如果该变量被设置的话），否则为“ODBC”。

password 是口令。如果它为 NULL，则只能在 user 授权表包含与用户名和正在连接的主机相配的项并且该项有空白口令时进行连接。

db_name 是要使用的数据库。如果它为 NULL，则没有初始数据库可选择。

port_num 是 TCP/IP 连接使用的端口号。如果它为 0，则使用缺省的端口号。

socket_name 是对本地主机进行连接所使用的套接字名。如果它为 NULL，则使用缺省的套接字名。

端口号和套接字名根据 host_name 的值进行使用。如果您正在连接到本地主机，则 mysql_real_connect() 试图使用 UNIX 域套接字（在 UNIX 下）或指定的管道（在 Windows

下) 尝试一个连接。否则, 它用 TCP/IP 连接。

flags 可以为 0, 表示无选项, 或为一个或更多的值, 这些值在表 F-5 中示出。这些选项影响服务器的操作。

表F-5 mysql_real_connect() 的 flags 值

flag 值	服务器操作的效果
CLIENT_FOUND_ROWS	对于 UPDATE 查询, 返回相配的行数而非变动的行数
CLIENT_NO_SCHEMA	不允许 db_name.tbl_name.col_name 的语法
CLIENT_COMPRESS	使用压缩通信协议, 如果服务器支持该协议的话
CLIENT_ODBC	将客户机看作是 ODBC 客户机

如果指定 CLIENT_NO_SCHEMA, 服务器在查询中只允许引用格式 tbl_name.col_name、tbl_name 或 col_name。

flag 值是比特位值, 因此可以用 ‘|’ + ‘+’ 组合它们, 例如, CLIENT_COMPRESS|CLIENT_ODBC 或 CLIENT_COMPRESS+CLIENT_ODBC。

mysql_real_connect() 是在 MySQL 3.21.10 中引入的。db_name 参数是在 3.22.0 版本中引入的。而初始化 MYSQL 参数的 mysql_init() 是在 3.22.1 版本中引入的。

F.3.2 状态和错误报告例程

本节中的函数允许您确定和报告错误的原因。

■ unsigned int

mysql_errno (MYSQL *conn);

返回最近调用的客户机库例程的错误代码 (返回一个状态)。如果没有出现错误, 则错误代码为 0, 否则为非零值。MySQL 头文件 errmsg.h 和 mysqld_error.h 列出了可能的错误代码。

```
if (mysql_errno (conn) == 0)
    printf ("Everything is okay\n");
else
    printf ("Something is wrong!\n");
```

mysql_errno() 是在 MySQL 3.21.7 中引入的。

■ char

mysql_error (MYSQL *conn);

对于返回一个状态的最近调用的客户机库例程, 返回包含错误消息的一个空终结串。如果没有错误, 则该返回值是空串 (这是长度为零的串 "", 而不是 NULL 指针)。尽管通常是在已经知道出现错误后调用 mysql_error(), 但该返回值本身可以用来检测错误的发生:

```
if (mysql_error (conn)[0] == '\0') /* empty string? */
    printf ("Everything is okay\n");
else
    printf ("Something is wrong!\n");
```

mysql_error() 是在 MySQL 3.21.7 中引进的。

F.3.3 查询结构和执行例程

本节中的函数允许您将查询发送到服务器中。mysql_escape_string() 帮助您通过需要特殊

处理的转义字符来构造查询。

■ unsigned int

```
mysql_escape_string (char *to_str, char *from_str, unsigned int
from_len);
```

将包含特殊字符的串进行编码使得它可用于 SQL 语句中。表F-6 列出了被认为是特殊的字符和它们的编码。

表F-6 mysql_escape_string() 字符编码

特殊字符	编 码
NUL (ASCII 0)	\0
反斜杠	\\
单引号	\'
双引号	\"
换行	\n
回车	\r
Ctrl-Z	\Z

要编码的缓冲区指定为记数串。from_str 指向该缓冲区，from_len 指定缓冲区中的字节数。mysql_escape_string() 将结果写入通过 to_str 指向的该缓冲区中，并增加一个空字节。to_str 必须指向一个已有的缓冲区，该缓冲区至少为 (from_len*2) +1 长（最坏的情况下，from_str 的所有字符都以一个二字符的序列进行编码，并且还需要终结空的空间）。

mysql_escape_string() 返回编码串的长度，但不终结空字节记数。

编码串不包含内部的空值而是空终结的，因此您可以利用函数（如 strlen() 或 strcat()）使用它。

当在程序中书写字母串时，要小心不要将 C 程序设计语言的词汇转义约定与由 mysql_escape_string() 完成的编码相混淆。考虑下面的例子以及所产生的输出结果：

```
to_len = mysql_escape_string (to_str, "\0\\\'\"\\n\\r\\032", 7);
printf ("to_len = %d, to_str = %s\\n", to_len, to_str);
```

输出结果为：

```
to_len = 14, to_str = \0\\\'\"\\n\\r\Z
```

to_str 的打印值非常像做为 mysql_escape_string() 调用的第二个参数所指定的串，但其实是极为不同的。

■ int

```
mysql_query (MYSQL *conn, char *query_string);
```

如果给定一个指定为空终结串的查询，mysql_query() 将该查询发送到服务器中运行。该串不包含二进制数据；尤其是不包含空字节，mysql_query() 将空字节解释为查询的结束。如果查询包含二进制数据，则用 mysql_real_query 来代替 mysql_query()。

该查询必须由单个的 SQL 语句组成，并不应以分号（；）或 ‘\g’ 终结（‘；’ 和 ‘\g’ 是 mysql 客户机程序的约定，而不是客户机库的约定。）

mysql_query() 运行成功时返回零，失败时返回非零值。一个成功的查询是服务器合法接

受的并且运行无错的查询。成功并不蕴涵着有关受影响或被返回的行数的任何内容。

■ int

```
mysql_real_query (MYSQL *conn, char *query_string, unsigned int
length);
```

如果给定一个指定为空终结串的查询，mysql_real_query() 将该查询发送到服务器中运行。该串可以包含二进制数据（包括空字节）。该查询的文本由 query_string 给出，长度由 length 指定。

该查询必须由单个的 SQL 语句组成，并不应以分号（‘；’）或‘\g’终结（‘；’和‘\g’是 mysql 客户机程序的约定，而不是客户机库的约定）。

mysql_real_query() 运行成功时返回零，失败时返回非零值。一个成功的查询是服务器合法接受的并且运行无错的查询。成功并不蕴涵着有关受影响或被返回的行数的任何内容。

■ int

```
mysql_select_db (MYSQL *conn, char *db_name);
```

选择由 db_name 指定的数据库为当前数据库，它成为没有包含明确的数据库说明符的表引用的缺省数据库。您必须具有访问该数据库的许可权，否则，mysql_select_db() 将失败。mysql_select_db() 运行成功时返回零，不成功时返回非零值。

F.3.4 处理例程的结果集

当一个查询产生结果集时，本节的函数允许您检索该集合访问它的内容。mysql_store_result() 和 mysql_use_result() 函数创建该结果集，并且在使用本节的任何其他函数之前必须调用这两个函数的任一个。表 F-7 对这两个函数进行了比较。

表F-7 mysql_store_result() 和 mysql_use_result() 的比较

mysql_store_result()	mysql_use_result()
结果集中的所有行都由 mysql_store_result() 本身提取	mysql_use_result() 初始化该结果集，但服从 mysql_fetch_row() 的行检索
使用更多内存；所有行都存储存客户机中	使用很少的内存；一次存储一行
速度慢，因为包括分配内存给整个结果集的开销	速度快，因为对当前行分配所需的内存
mysql_fetch_row() 中的 NULL 值表明结果集的结尾，决不是表明一个错误	mysql_fetch_row() 的 NULL 值表明结果集的结尾或一个错误，因为通信的失败可以破坏对当前记录的检索
mysql_num_rows() 可以在调用 mysql_store_result() 之后的任何时候进行调用	mysql_num_rows() 仅当所有行已经提取后返回正确的行记录
mysql_affected_rows() 是 mysql_num_rows() 的同义词	不能使用 mysql_affected_rows()
结果集行的随机访问可能利用 mysql_data_seek()、mysql_row_seek() 和 mysql_row_tell()	没有对结果集的随机访问；行必须按照由服务器返回的顺序进行处理；mysql_data_seek()、mysql_row_seek() 和 mysql_row_tell() 不使用
根本不必调用 mysql_fetch_row()，因为所有的行已经被检索	必须调用 mysql_fetch_row() 提取所有的行，否则“剩余”行将漏入下一个查询的结果中，导致“不一致”错误。
表进行读锁定，因为不再需要提取数据行	表可以暂时读锁定，如果客户机在检索中的话，将试图修改表的客户机锁在外面
结果集 MYSQL_FIELD 结构的 max_length 成员实际在结果集中为列设置最长的值	max_length 不设置为任何有意义的值，因为它直到所有行都检索完后才知道

■ my_ulonglong

mysql_affected_rows (MYSQL *conn);

返回由最近的 DELETE、INSERT、REPLACE 或 UPDATE 查询修改的行数。对于这样的查询，mysql_affected_row() 在对 mysql_query() 成功调用之后立即被调用。还可以在发布一个有返回行的语句后调用该函数。这时，该函数的作用与 mysql_num_rows() 的一样，并遵从相同的约束条件（有意义的值），以及遵从附加的约束条件（在使用 mysql_use_result() 时 mysql_affected_rows() 无意义）。如果没有发布任何查询，或如果该查询返回行但没有任何行被选中，mysql_affected_rows() 则返回零。返回值大于零表明修改的（对于 DELETE、INSERT、REPLACE、UPDATE）或返回的（对于有返回行的查询）行数，返回值为 -1 表明有错，或者是在发布有返回值的查询之后、但在实际检索结果集之前您错误地调用了 mysql_affected_rows()。因为 mysql_affected_rows() 返回一个无符号值，所以应该通过定义其结果为符号值来完成比较：

```
if ((long) mysql_affected_rows (conn) == -1)
    fprintf (stderr, "Error!\n");
```

如果您已经指明客户机应该返回与 UPDATE () 查询相配的行数，则 mysql_affected_rows() 将返回该值，而非实际已修改的行数（如果要修改的列与新值相同，MySQL 将不对行进行更新）。通过指定选项文件中的 return-found-rows，或者通过将 flags 参数中的 CLIENT_FOUND_ROWS 值传送到 mysql_real_connect()，可以选择该性能。

请参见“标量数据类型”有关打印 my_ulonglong 类型值的说明。

■ void

mysql_data_seek (MYSQL_RES *res_set, unsigned int offset);

搜索结果集的特定行。offset 值的范围从 0 到 mysql_num_rows(res_set)-1。如果 offset 超出范围，则结果是不可预知的。

mysql_data_seek() 要求整个结果集已经检索，因此只有在该结果集通过 mysql_store_result()（而不是通过 mysql_use_result()）创建后才能使用。

■ MYSQL_FIELD*

mysql_fetch_field (MYSQL_RES *res_set);

返回包含关于结果集中某列的信息（元数据）的结构。在成功地执行有返回行的查询后，对 mysql_fetch_field() 的第一个调用返回有关第一列的信息。后续的调用将返回有关第一列的后续列信息，如果没有那么多的列，返回值为 NULL。

还可以调用 mysql_field_tell() 确定当前列的位置，或调用 mysql_field_seek() 选择在下一个 mysql_fetch_field() 调用中要返回的指定列。

下例检索第一个 MYSQL_FIELD，然后提取连续的列信息结构：

```
MYSQL_FIELD *field;
unsigned int i;

mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    printf ("column %u: name = %s max_length = %lu\n",
           i, field->name, field->max_length);
}
```

■ MYSQL_FIELD *

```
mysql_fetch_fields (MYSQL_RES res_set);
```

返回结果集的所有列信息结构的一个数组。您可以按如下形式对它们进行访问：

```
MYSQL_FIELD    *field;
unsigned int    i;

field = mysql_fetch_fields (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    printf ("column %u: name = %s max_length = %lu\n",
            i, field[i].name, field[i].max_length);
}
```

将它与 `mysql_fetch_field()` 的例子进行比较。请注意，尽管两个函数都返回同一类型的值，但访问每个函数的这些值的语法稍有不同。

■ MYSQL_FIELD *

```
mysql_fetch_field_direct (MYSQL_RES *res_set, unsigned int field_num);
```

如果给定一个列的索引，则返回该列的信息结构。 `field_num` 值的范围从 0 到 `mysql_num_fields()` - 1。如果 `field_num` 超出范围，则该结果是不可预知的。

以下例子直接访问 `MYSQL_FIELD` 的结构：

```
MYSQL_FIELD    *field;
unsigned int    i;

for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field_direct (res_set, i);
    printf ("column %u: name = %s max_length = %lu\n",
            i, field->name, field->max_length);
}
```

`mysql_fetch_field_direct()` 在 MySQL 3.23 版本之前还不能完全地工作。

■ unsigned long *

```
mysql_fetch_lengths (MYSQL_RES *res_set);
```

返回指向一个 `unsigned long` 值的数组的指针，这些值表示结果集的当前行中列的长度。每当调用 `mysql_fetch_row()` 时，或长度与数据值不一致时，必须调用 `mysql_fetch_lengths()`。

NULL 值的长度为 0，但长度为零并不表示其本身是 NULL 值。因为空串的长度也为零，您必须检查该数据值是否是空指针来区别这两种情况。

下例示出当前行的长度和值，如果该值为 NULL 则打印“NULL”：

```
unsigned long *length;

length = mysql_fetch_lengths (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    printf ("length is %lu, value is %s\n",
            length[i], (row[i] != NULL ? row[i] : "NULL"));
}
```

`mysql_fetch_lengths()` 是在 MySQL 3.20.5 中引入的。在 MySQL 3.22.7 之前，

mysql_fetch_lengths() 的返回类型为 unsigned int。

■ MYSQL_ROW

mysql_fetch_row (MYSQL_RES *res_set);

返回指向结果集下一行的指针，该行以一个串数组表示（除 NULL 列值表示为 NULL 指针外）。该行的 i-th 值是该值数组的 i-th 成员。

所有数据类型的值（甚至是数值类型的）都以串形式返回。如果要想利用某个值完成一个数值数值计算，必须将其转换——例如用 atoi() 或 atof()。

当数据集中不再有行时，mysql_fetch_row() 返回 NULL（如果使用 mysql_use_result() 开始逐行的结果集检索，并且出现通信错误的话，mysql_fetch_row() 也返回 NULL。）

数据值是空终结串，但不应将含有二进制数据的值看作时空终结串。应将它们看成时计数串（要想确定列值的长度，调用 mysql_fetch_lengths()）。

以下代码显示出如何循环通过一行数据值，以及如何确定每个值是否为 NULL：

```
MYSQL_ROW      row;
unsigned int    i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        printf ("column %u: value is %s\n",
                i, (row[i] == NULL ? "NULL" : "not NULL"));
    }
}
```

为了确定列值的类型，使用存储在 MYSQL_FIELD 列信息结构中的该列的元数据，可以通过调用 mysql_fetch_field()、mysql_fields() 或 mysql_fetch_field_direct() 得到。

■ unsigned int

mysql_field_count (MYSQL *conn);

返回指定连接中的最近查询的列的数目。该函数通常是在 mysql_store_result() 或 mysql_use_result() 返回 NULL 时使用。mysql_field_count() 指示您结果集是否已经返回。返回值为 0 表明无结果集并无错误。非零值表明有所希望的列，或者是一个错误（因为没有返回任何内容）。

下例说明怎样为检测错误使用 mysql_field_count()：

```
res_set = mysql_store_result (conn);
if (res_set == NULL) /* no result set was returned */
{
    /*
     * does the lack of a result set mean that an error
     * occurred or that no result set should be expected?
     */
    if (mysql_field_count (conn) > 0)
    {
        /*
         * a result set was expected, but mysql_store_result()
         * did not return one; this means an error occurred
         */
    }
}
```

```

    printf ("Problem processing result set\n");
}
else
{
    /*
     * a result set was not expected; query returned no data
     * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
     * so just report number of rows affected by query
     */
    printf ("%lu rows affected\n",
            (unsigned long) mysql_affected_rows (conn));
}
}
else /* a result set was returned */
{
    /* ... process rows here ... */
    mysql_free_result (res_set);
}

```

mysql_field_count() 是在 MySQL 3.22.24 中引入的。在该版本以前，mysql_numfields() 的作用与之相同。要想编写可使用任何 MySQL 版本的代码，应在使用 mysql_field_count() 的任何文件中包括以下代码段：

```

#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID<32224
#define mysql_field_count mysql_num_fields
#endif

```

当源程序在旧的 MySQL 版本下编译时，此段代码将 mysql_field_count() 影射到 mysql_num_fields() 中。

■ MYSQL_FIELD_OFFSET

```
mysql_field_seek (MYSQL_RES *res_set, MYSQL_FIELD_OFFSET offset);
```

搜索由 offset 指定的列信息结构。对 mysql_fetch_field() 的下次调用将返回该指定列的信息结构。offset 不是列的索引，它是 MYSQL_FIELD_OFFSET 值，是从对 mysql_field_tell() 或 mysql_field_seek() 的调用中得到的。

要想复位到第一列，传递一个为 0 的 offset 值。

■ MYSQL_FIELD_OFFSET

```
mysql_field_tell (MYSQL_RES *res_set);
```

返回当前的列信息结构的偏移量。该值可传递到 mysql_field_seek() 中。

■ void

```
mysql_free_result (MYSQL_RES *res_set);
```

释放结果集使用的内存。必须为您使用的每个结果集调用 mysql_free_result()。通常，通过调用 mysql_store_result() 或 mysql_use_result() 生成结果集。但是，有些客户机函数间接地生成结果集，因此您也有责任释放那些集合。这些函数是 mysql_list_dbs()、mysql_list_fields()、mysql_list_processes() 和 mysql_list_tables()。

■ char*

```
mysql_info (MYSQL *conn);
```

返回一个串，该串包含有关最近执行的下列类型的查询所实现的效果的信息。该串的格式紧跟在每个查询后给出：

```

ALTER TABLE ...
    Records: 0 Duplicates: 0 Warnings: 0
INSERT INTO ... SELECT ...
    Records: 0 Duplicates: 0 Warnings: 0
INSERT INTO ... VALUES (...),(...),...
    Records: 0 Duplicates: 0 Warnings: 0
LOAD DATA ...
    Records: 0 Deleted: 0 Skipped: 0 Warnings: 0
UPDATE ...
    Rows matched: 0 Changed: 0 Warnings: 0

```

当然，这些数字将根据您执行的特定的查询而变化。

对于未在上述列表中出现的语句，`mysql_info()` 返回 `NULL`。仅当 `INSERT INTO ... VALUES` 中包含多于一个值的列表时，`mysql_info()` 对该语句返回非 `NULL` 值。

由 `mysql_info()` 返回的串是用服务器所使用的语言的，因此不要指望可以通过查找某些词对它进行语法分析。

■ `my_ulonglong`

`mysql_insert_id (MYSQL conn);`

返回由最近执行的给定连接上的查询所产生的 `AUTO_INCREMENT` 值。如果没有执行查询或者前一查询不产生 `AUTO_INCREMENT` 值，则返回零。这意味着您应该在一个期待产生新值的查询后立即调用 `mysql_insert_id()`。如果任何其他查询在该查询和您要想使用该值的那一刻之间介入，`mysql_insert_id()` 的值将由介入的查询复位。

零返回值不同于任何合法的 `AUTO_INCREMENT` 值，因为这些值从 1 开始（例外：如果您创建 `AUTO_INCREMENT` 列然后插入一个直接量的负数，则序列将从该数字开始，并且最终达到作为该序列的合法成员的 0。在这种情况下假定您知道正在做什么，因为 `mysql_insert_id()` 返回一个无符号数并且您要想玩弄这个返回值）。

请注意，`mysql_insert_id()` 的特点与 SQL 函数 `LAST_INSERT_ID()` 的不同。`mysql_insert_id()` 在客户机中维护并对每个查询进行设置。`LAST_INSERT_ID()` 的值在服务器中维护并将在所有查询中保持。

由 `mysql_insert_id()` 返回的值是连接专用的，它不受其他连接中的 `AUTO_INCREMENT` 活动的影响。

请参阅 F.2.1 节“标量数据类型”中关于打印 `my_ulonglong` 类型值的说明。

■ `unsigned int`

`mysql_num_fields (MYSQL_RES *res_set);`

返回结果集的列数目。`mysql_num_rows()` 常常用来循环通过该集合的当前行的列，如下例所示：

```

MYSQL_ROW    row;
unsigned int  i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        /* do something with row[i] here ... */
    }
}

```

在 MySQL 3.22.24 以前, `mysql_num_fields()` 还用来完成现在由 `mysql_field_count()` 所函数完成的功能, 即, 测试来自 `mysql_store_result()` 或 `mysql_use_result()` 中的 `NULL` 返回值是否表示有错。这就是为什么在旧的源程序代码中, 有时将看到 `mysql_num_fields()` 正在用指向一个连接处理程序的指针进行调用, 而不是用指向结果集的指针。 `mysql_num_fields()` 通常可用于这两种方式。利用连接处理程序的 `mysql_num_fields()` 的使用方式现在不受赞成。取而代之, 应该用 `mysql_field_count()` 来编写程序。在该函数的描述中说明了如何使用它, 甚至对于较旧的 MySQL 版本的。

■ `my_ulonglong`

`mysql_num_rows (MYSQL_RES *res_set);`

返回结果集中的行数。如果是用 `mysql_store_result()` 产生结果集, 则可以在那以后任何时候调用 `mysql_num_rows()` :

```
if ((res_set = mysql_store_result (conn)) == NULL)
{
    /* mysql_num_rows() can be called now */
}
```

如果是用 `mysql_use_result()` 产生结果集, 则 `mysql_num_rows()` 在您提取所有行之前不返回正确的值 :

```
if ((res_set = mysql_use_result (conn)) == NULL)
{
    /* mysql_num_rows() cannot be called yet */
    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        /* mysql_num_rows() still cannot be called */
    }
    /* mysql_num_rows() can be called now */
}
```

请参阅“标量数据类型”一节关于打印 `my_ulonglong` 类型值的说明。

■ `MYSQL_ROW_OFFSET`

`mysql_row_seek (MYSQL_RES *res_set, MYSQL_ROW_OFFSET offset);`

搜索结果集中的一个特定行。 `mysql_row_seek()` 类似于 `mysql_data_seek()`, 但 `offset` 值不是行号。 `offset` 必须是一个从对 `mysql_row_tell()` 或 `mysql_row_seek()` 的调用中得到的值。

`mysql_row_seek()` 返回前一行的偏移量。

`mysql_row_seek()` 要求整个结果集已经被检索过, 因此仅当该结果集是由 `mysql_store_result()` 而不是由 `mysql_use_result()` 创建时才能使用。

■ `MYSQL_ROW_OFFSET`

`mysql_row_tell (MYSQL_RES *res_set);`

返回表示结果集中当前行位置的偏移量。它不是行号, 该值只能被传递到 `mysql_row_seek()` 中, 而不传递到 `mysql_data_seek()` 中。

`mysql_row_tell()` 要求整个结果集已经检索, 因此, 仅当该结果集是由 `mysql_store_result()` 而不是由 `mysql_use_result()` 创建时才能使用。

■ `MYSQL_RES *`

`mysql_store_result (MYSQL *conn);`

它跟在一个成功查询的后面, 返回结果集并将其存储在客户机中。如果该查询不返回任何数据

或者没有出现错误，则返回NULL。当 `mysql_store_result()` 返回 NULL 时，调用 `mysql_field_count()` 或错误报告函数中的一个来确定结果集是否不是所期望的，或者是否出现了错误。

当处理完结果集后，将其传递到 `mysql_free_result()` 中进行释放。

请参阅表F-7 中的 `mysql_store_result()` 和 `mysql_use_result()` 的比较。

■ MYSQL_RES*

`mysql_use_result (MYSQL *conn);`

紧跟在一个运行成功的查询后面开始结果集的检索，但不检索任何数据行本身。必须调用 `mysql_fetch_row()` 来逐个地提取这些行。如果该查询不返回数据或出现错误，则返回 NULL。当 `mysql_use_result()` 返回 NULL 时，调用 `mysql_field_count()` 或错误报告函数中的一个来确定结果集是否不是所期望的，或者是否出现了错误。

当处理完结果集后，将其传递到 `mysql_free_result()` 中进行释放。

请参阅表F-7 中的 `mysql_store_result()` 和 `mysql_use_result()` 的比较。

`mysql_store_result()` 和 `mysql_use_result()` 都是用来检索结果集的，但它们影响您使用其他结果集处理函数。

F.3.5 信息例程

这些函数提供关于客户机、服务器、协议版本和当前连接的信息。

■ char *

`mysql_get_client_info (void);`

返回一个空终结串，它描述客户机的版本，如 “ 3.22.25 ”。

■ char *

`mysql_get_host_info (MYSQL *conn);`

返回一个空终结串，它描述当前的连接，如 “ Localhost via UNIX socket ” 或 “ your.host.com via TCP/IP ”。

■ unsigned int

`mysql_get_proto_info (MYSQL *conn);`

返回一个数字，它指明用于当前连接的客户机 / 服务器协议的版本。

■ char *

`mysql_get_server_info (MYSQL *conn);`

返回一个空终结串，它描述服务器的版本，如 “ 3.22.25-log ”。跟在服务器版本号后面的后缀是 -log（日志是开启的）、-debug（服务器正在以调试方式运行），或 -demo（服务器正以演示方式运行）。

■ char *

`mysql_stat (MYSQL *conn);`

返回一个空终结串，它包含服务器状态信息或 NULL（如果出现错误的话）。该串的格式根据变化而改变。通常它如下所示：

```
Uptime: 864034  Threads: 1  Questions: 32736  Slow queries: 50  Opens: 1428
Flush tables: 1  Open tables: 61
```

这些值的解释如下：

- Uptime 是服务器已经打开的秒数。
- Threads 是当前运行在服务器上的线程的数目。
- questions 是服务器已经执行的查询数目。
- Slow queries 是所花费时间比服务器的 long_query_time 参数长的查询数目。
- Opens 是服务器已经打开的表的数目。
- Flush tables 是已经执行 FLUSH、REFRESH 和 RELOAD 命令的次数。
- Open tables 是服务器当前已经打开的表的数目。

由 `mysql_stat()` 返回的信息与 `mysqladmin status` 命令报告的信息相同。（您认为 `mysqladmin` 应该在哪里得到这些信息？）

■ unsigned long
`mysql_thread_id (MYSQL*Conn);`

返回服务器与当前连接相关的线程数。可以使用该数目作为 `mysql_kill()` 的标识符。

在您需要该值之前执行 `mysql_thread_id()` 不是好主意。如果您得到该值并存储它，打算在后面使用，则该值可能会不正确。这种情况可能发生在连接断开后又重新建立时（如用 `mysql_ping()`），因为服务器将分配一个新的线程标识符。

F.3.6 管理例程

本节的函数允许您控制服务器运作方式。

■ int
`mysql_kill (MYSQL *conn, unsigned long thread_id);`

取消由 `thread_id` 标识的服务器线程。

如果有 `PROCESS` 权限，则可以取消任何线程。否则，只能取消属于自己的线程。

`mysql_kill()` 在运行成功时返回零，失败时返回非零值。

■ int
`mysql_refresh (MYSQL *conn, unsigned int options);`

该函数在效果上类似于 `SQL FLUSH` 语句，除了您可以指示服务器立即刷新几种类型的事物以外。options 值应该是表 F-8 所示的一个或多个值。

表F-8 `mysql_refresh()` 选项

选 项 值	服务器采取的行动
REFRESH_GRANT	重新加载授权表的内容
REFRESH_LOG	开始新的常规和更新日志（它们通常是打开的）
REFRESH_TABLES	关闭所有打开的表
REFRESH_HOSTS	刷新主机的高速缓存
REFRESH_STATUS	将状态变量复位到零

表F-8 中的选项是比特位值，因此可以用 ‘|’ 或 ‘+’ 将它们组合，例如，`REFRESH_LOG|REFRESH_TABLES` 或 `REFRESH_LOG+REFRESH_TABLES`。

有关刷新操作的详细信息，请参阅附录 D 中对 `FLUSH` 语句的描述。

`mysql_refresh()` 运行成功时返回零，不成功时返回非零值。

■ int

```
mysql_shutdown (MYSQL *conn);
```

指示服务器关闭。您必须具有 SHUTDOWN 权限才能进行。

mysql_shutdown() 运行成功时返回零，不成功返回非零值。

F.3.7 调试例程

这些函数允许您产生该连接的客户机端或服务器端的调试信息。这需要编译 MySQL 以支持调试（在您配置 MySQL 分发时使用 --with-debug 选项）。

■ void

```
mysql_debug (char *debug_string);
```

用串 debug_string 执行 DBUG_PUSH 选项。该串的格式在 MySQL 参考手册中介绍。

要使用 mysql_debug(), 客户机库必须用调试支持环境进行编译。

■ int

```
mysql_dump_debug_info (MYSQL *conn);
```

指示服务器将调试信息写到日志中。您必须具有 PROCESS 权限才能进行。

mysql_dump_debug_info() 运行成功时返回零，不成功返回非零值。

F.3.8 不受赞成的例程

由于有更好的方法可用来进行相同的工作，因此，MySQL 客户机库中包括大量的目前不受赞成的函数。大多数这些函数都通过将等价的查询传递到 mysql_query() 中被取代。例如，mysql_create_db(“db_name”) 可以用下列调用取代：

```
mysql_query (conn, “CREATE DATABASE db_name”)
```

有几个函数，如 mysql_connect() 和 mysql_eof(), 都是不赞成的，因为它们已经由可以做更多工作的或提供更多信息的函数取代。

随着时间的推移，MySQL 对 SQL 语句理解得越多，不受赞成的函数就越多。例如，当增加 SQL FLUSH PRIVILEGES 语句时，mysql_reload() 不受赞成。下列描述指明了每个函数不受赞成的 MySQL 版本以及执行每个函数的现在更好的方法。如果客户机库比所列出的 MySQL 版本还旧，当然仍必须使用不赞成的函数。

如果想要对未来打算，应该避免使用本节所列出的所有函数。这些中的一部分或全部将在 MySQL 4.0 中消失。

■ MYSQL *

```
mysql_connect (MYSQL *conn, char *host_name,  
              char *user_name, char *password);
```

这是 mysql_real_connect() 的前身。事实上，现在它作为对 mysql_real_connect() 的调用被实施。

该函数自 MySQL 3.22.0 以来不受赞成。

■ int

```
mysql_create_db (MYSQL *conn, char *db_name);
```

用 db_name 指定的名字创建数据库。这个功能现在可以通过利用 mysql_query() 发布

CREATE DATABASE 命令来完成。

mysql_create_db() 运行成功时返回零，不成功返回非零值。

该功能自 MySQL 3.21.15 以来一直不受赞成。

■ int

mysql_drop_db (MYSQL *conn, char *db_name);

删除 db_name 指定的数据库。这个功能现在可通过用 mysql_query() 发布 DROP DATABASE 命令来进行。

mysql_drop_db() 运行成功时返回零，失败返回非零值。

该函数自 MySQL 3.21.15 以来不受赞成。

■ my_bool

mysql_eof (MYSQL_RES *res_set);

如果已经到达结果集的末尾返回非零值，如果出错返回零。在您使用开始结果集检索的 mysql_use_result() 和同时提取数据行的 mysql_fetch_row() 的组合时使用 mysql_eof()。对于 mysql_use_result(), mysql_fetch_row() 的 NULL 返回值意味着或者到达集合的末尾或者是出错。mysql_eof() 在两个结果之间进行辨别。

mysql_errno() 和 mysql_error() 现在可用来达到相同的效果，尽管它们实际返回更多的信息（它们指出已经出现的任何错误的原因，而不是只简单地指出是否有错）。

该函数自 MySQL 3.21.7 以来一直不受赞成。

■ MYSQL_RES *

mysql_list_dbs (MYSQL *conn, char *wild);

返回列出服务器中数据库名的结果集或 NULL（如果有错）。该列表中包含与 wild 指明的 SQL 常规表达式（可能包含通配符 ‘ % ’ 和 ‘ _ ’）相配的所有数据库，或者 wild 为 NULL 时的所有数据库。您有责任调用 mysql_free() 释放结果集。

mysql_list_dbs() 产生的列表可通过利用 mysql_query() 执行 SHOW DATABASE 命令然后处理该结果集来获得。

该函数自 MySQL 3.22.0 以来一直不受赞成。

■ MYSQL_RES *

mysql_list_fields (MYSQL *conn, char *tbl_name, char *wild);

返回在给定表中显示出列名的结果集或 NULL（如果有错的话）。该列表包含与 wild 指明的 SQL 常规表达式（可能包含通配符 ‘ % ’ 和 ‘ _ ’）相配的所有列名，或者 wild 为 NULL 时的所有列。您有责任调用 mysql_free() 释放结果集。

mysql_list_field() 产生的列表可通过利用 mysql_query() 执行 SHOW COLUMNS 命令然后处理该结果集来获得。

该函数自 MySQL 3.22.0 以来一直不受赞成。

■ MYSQL_RES *

mysql_list_processes (MYSQL *conn);

返回包含在服务器中运行的进程列表的结果集或 NULL（如果有错的话）。如果具有 PROCESS 特权，该列表将包含所有服务器进程。如果没有，该列表只包含属于自己的进程。您有责任调用 mysql_free() 释放结果集。

`mysql_list_processes()` 产生的列表可通过利用 `mysql_query()` 执行 `SHOW PROCESSLIST` 查询然后处理该结果集来获得。

该函数自 MySQL 3.22.0 以来一直不受赞成。

■ `MYSQL_RES *`

`mysql_list_tables` (`MYSQL *conn`, `char *wild`);

返回列出当前数据库的表的结果集或 `NULL`（如果有错的话）。该列表包含与 `wild` 指明的 SQL 常规表达式（可能包含通配符 ‘ % ’ 和 ‘ _ ’）相匹配的所有表，或者 `wild` 为 `NULL` 时的所有表。您有责任调用 `mysql_free()` 释放结果集。

`mysql_list_table()` 产生的列表可通过利用 `mysql_query()` 执行 `SHOW TABLES` 命令然后处理该结果集来获得。

该函数自 MySQL 3.22.0 以来一直不受赞成。

■ `int`

`mysql_reload` (`MYSQL *conn`);

指示服务器重新加载授权表。这个功能现在可以通过利用 `mysql_query()` 发布 `FLUSH PRIVILEGES` 查询来进行。必须具有 `RELOAD` 权限才能使用 `mysql_reload()`。

`mysql_reload()` 运行成功时返回零，失败返回非零值。

该函数自 MySQL 3.21.9 以来一直不受赞成。

附录G Perl DBI API 参考

本附录介绍了 Perl DBI 应用程序编程接口。API 由与数据库服务器通信及从 Perl 脚本访问数据库的一组方法和属性组成。本附录还描述了由 DBD::mysql 提供的针对 MySQL 进行的 DBI 扩充，即 MySQL 数据库驱动程序。

有一些 DBI 的方法和属性不在这里介绍，因为它们不用于 MySQL，或者因为它们还是新的，实验性的方法，而随着开发的进行，这些方法可能有变动，甚至被删除。一些 MySQL 特定的 DBD 方法由于已经过时，所以也不在这里讨论。如果了解有关新的或过时的方法的详细情况，请参阅 DBI 文档或 MySQL DBD 文档，运行下面的命令，我们可以获得这些文档：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

本附录仅供参考，所以只含有说明 Perl DBI API 用法的简要代码段。完整的客户机脚本和编写它们时的注意事项，请参阅第 7 章。

G.1 编写脚本

使用 DBI 模块的每个 Perl 脚本都必须包括下面的行：

```
use DBI;
```

特定的 DBD 级的模块不需要包括 use 行，因为在与服务器进行连接时，DBI 负责激活相应的模块。

一般来说，DBI 脚本使用 connect() 方法打开连接，发布查询，然后用 disconnect() 关闭连接。这些方法通常发布各种查询。具有代表性的非 SELECT 查询由 do() 方法实现。具有代表性的 SELECT 查询则将这个查询传递给 prepare()，然后调用 execute()，最后在重复调用提取行的方法的循环中，如在 fetchrow_array() 或 fetchrow_hashref() 中，检索查询结果，每次检查一行。

G.2 DBI 方法

这里描述的方法的编写格式与用于附录 F 中的 C 函数和附录 H 中的 PHP 函数的编写格式略有不同。那些附录中的函数以原型方式编写，明确地列出了返回值的类型和参数类型。一般来说，Perl 以同样的方式使用原型，所以这里的说明通过变量指定了返回值的类型和参数类型，变量的开始字符表明值的类型：‘\$’ 为标量，‘@’ 为数组，‘%’ 为散列（关联的数组）。除此之外，列出的以 ‘\’ 开始的任何参数，应该作为对变量的引用来传递，而不是传递变量本身。以 ‘_ref’ 结束的变量名称表示变量值为一个引用。

在这个附录中，有几个用于特殊含意的变量，如表 G-1 所示。

许多方法都接受散列参数 %attr，该参数包括影响相应方法的工作方式的属性。这个散列应该通过引用来传递。可以以两种方式来做这件事。一种方式是在调用此方法之前，设置散

列值 %attr 的内容，然后再将它传递给此方法。

```
my (%attr) = (AttrName1 => value1, AttrName2 => value2);
$ret_val = $h->method (...,\%attr);
```

另一种方式为在此方法调用中直接提供匿名的散列：

```
$ret_val = $h->method (...,{AttrName1 => value1, AttrName2 => value2});
```

表G-1 常用的 Perl DBI 变量名称

名 称	含 意
\$drh	驱动程序对象的句柄
\$dbh	数据库对象的句柄
\$sth	语句（查询）对象的句柄
\$fh	打开文件的句柄
\$h	“通用”句柄；其含意有赖于上下文
\$rc	从返回真或假的操作中返回的代码
\$rv	从返回整数的操作中返回的值
\$rows	从返回行计数的操作中返回的值
@ary	表示查询返回的一行值的数组（列表）

方括号（[]）表示可选的信息。其中使用方法或函数的方式由调用序列来表示。‘DBI->’表示 DBI 类方法，‘DBI::’表示 DBI 函数，‘\$DBI::’表示 DBI 变量。对于使用句柄来调用的方法，相应的句柄名称表示方法的范围。‘\$dbh->’表示数据库名称方法，‘\$sth->’表示语句句柄方法，‘\$h->’表示可以用各种不同类型的句柄调用的方法。下面是一个调用序列的实例。

```
@row_ary = $dbh->selectrow_array ($statement, [\%attr [, @bind_values]]);
```

这表示 selectrow_array() 方法是作为数据库句柄方法调用的，因为利用了 ‘\$dbh->’ 来调用它。其中，参数为 \$statement（标量值）、%attr（应该作为引用来传递的散列，因为以 ‘\’ 开头来表示）和 @bind_values（数组）。第二个和第三个参数为可选参数。返回值为数组。

这种方法的说明指出如果出现错误，返回什么样的值，而且仅当禁用 RaiseError 属性时才返回错误。如果启用 RaiseError，则此脚本将自动终止。

在后面的说明中，将用术语“SELECT 查询”来表示 SELECT 查询或返回行的任何其他查询，如 DESCRIBE、EXPLAIN 或 SHOW。

G.2.1 DBI 类方法

在本节中，方法的 %attr 参数可能用来指定方法处理的属性。就 MySQL 来说，这些属性中最重要的是 PrintError 和 RaiseError。例如，要想在出现 DBI 错误时，使自动脚本的作用终止，应该启用 RaiseError：

```
$dbh = DBI->connect ($data_source, $user_name, $password, {RaiseError => 1});
```

PrintError 和 RaiseError 在“通用句柄属性”一节中介绍。

```
■ @ary = DBI->available_drivers ([ $quiet]);
```

返回可得到的 DBI 驱动程序的一个列表。如果发现多个驱动程序具有同样的名称，则发布一条警告。给 \$quiet 参数传递值 1 可避免出现这条警告。

■ `$dbh = DBI->connect ($data_source, $user_name, $password, [, \%attr]);`

`connect()` 创建与数据库服务器的连接，并返回数据库句柄。如果连接失败，则该句柄为 `undef`。要想终止成功创建的连接，用 `connctet()` 返回的数据库句柄调用 `disconnect()` 方法即可。

对于 `connect()`，在 `@attr` 参数中指定的任何属性都会在全局影响 DBI 的处理。

可以用几种格式给定数据源。但第一部分总是 “DBI:mysql”，可以用任何大小写字母给出 “DBI”，而驱动程序名称 “mysql” 必须为小写字母。在数据源的驱动程序名称部分后的所有字符都由该驱动程序解释，所以下面讨论中说明的语法除 `DBD::mysql` 外，不一定任何驱动程序都要用。也可以在数据源字符串的开头部分、驱动程序名之后，指定数据库名和主机名：

```
$data_source = "DBI:mysql:db_name";  
$data_source = "DBI:mysql:db_name:host_name";
```

可以用 `db_name` 或 `database=db_name` 指定数据库名。可以用 `host_name` 或 `host=host_name` 指定主机名。

接着数据源字符串的初始部分，可以用 `attribute=value` 的格式指定若干选项。每个选项的前面都必须有一个分号。MySQL 驱动程序允许下面的选项：

■ `host=host_name` 要连接的主机。也可以使用 `host_name:port_num` 格式来指定端口号。

■ `port=port_num` 要连接的端口号。

■ `mysql_compression=1` 这个选项在客户机和 MySQL 服务器之间启用压缩通信。

`mysql_compression` 需要 MySQL 3.22.3 或更新的版本，以及 `DBD::mysql` 1.19.20 或更新的版本。

■ `mysql_read_default_file=file_name` 缺省设置时，DBI 脚本不读取连接参数的 MySQL 选项文件。此选项允许读取指定的选项文件。文件名称应该为完整的路径名（否则，将会解释为相对于在其中执行脚本的目录，这样会得到不一致的结果）。

为了只要用户运行该脚本就读取主目录中的 `.my.cnf` 文件，应该指定该文件名为 `$ENV{HOME}/.my.cnf`。如果期望多个用户使用同一脚本，并想让他们自己连接而不是作为此脚本中固定的一个用户来连接时，这样做很有用。

`mysql_read_default_file` 需要 MySQL 3.22.10 或更高的版本，以及 `DBD::mysql` 1.21.06 或更高的版本。

■ `mysql_read_default_group=group_name` 如果读取连接参数的选项文件，则缺省时读取 `[client]` 组。除了 `[client]` 组以外，`mysql_read_default_group` 选项还允许读取一个指定组。例如，`mysql_read_default_group=dbi` 指定 `[dbi]` 组应该像 `[client]` 组一样读取。出现在多个组中的参数的优先权在附录 E 中介绍。

`mysql_read_default_group` 需要 MySQL 3.22.10 或更高的版本，需要 `DBD::mysql` 1.21.06 或更高的版本。

■ `mysql_socket=socket_name` 如果缺省套接字路径不合适，则本选项指定了用于连接到 `localhost` 的 UNIX 域套接字的路径名。

`mysql_socket` 需要 MySQL 3.21.15 或更高的版本。DBI 考虑连接参数的若干环境变量：

■ 如果没有定义数据源或者数据源为空，则使用 `DBI_DSN` 变量的值。

- 如果该数据源中缺少驱动程序名称, 则使用 DBI_DRIVER 变量的值。
- 如果未定义 connect() 调用的 user_name 或 password 参数, 则使用 DBI_USER 和 DBI_PASS 变量的值。如果这些参数为空字符串, 则不必使用。

就安全而言, 使用 DBI_PASS 有些冒险, 所以在安全性很重要的情况下, 都不应该使用它(依靠像 ps 这样的命令, 其他用户可以看到环境变量)。

如果查阅了所有的信息源之后, 仍然不知道某些连接参数, 则 DBI 使用缺省值。如果未指定主机名称, 则其缺省值为 localhost。如果未指定用户名称, 则其在 UNIX 下的缺省值为登录名称, Windows 下为 ODBC。如果不给出口令, 则无缺省值, 不发送口令。

■ \$drh = DBI->install_driver (\$driver_name);

激活 DBD 级别的驱动程序, 并返回驱动程序句柄。对于 MySQL, 这个驱动程序名称为 “mysql” (必须为小写字母)。一般来说, 不一定要使用这种方法, 因为在调用 connect() 方法时, DBI 会自动激活恰当的驱动程序。然而, 如果使用 func() 方法来实现管理操作, 应该可以使用 install_driver() (请参阅 “MySQL 的特定管理方法”)。

G.2.2 数据库句柄方法

本节中的这些方法可通过数据库句柄进行调用, 在通过调用 connect() 方法获得这样一个句柄后, 可以使用这些方法。

本节所使用方法的 %attr 参数可用于特定的方法处理属性。对于 MySQL, 这些属性中最重要的是 PrintError 和 RaiseError。例如, 如果在像这样的特定查询的处理过程中, DBI 出现错误, 可如下启用 RaiseError 使脚本自动终止:

\$rows = \$dbh->do (\$statement, {RaiseError => 1});

在 “通用句柄属性” 一节中讨论 PrintError 和 RaiseError。

■ \$rc = \$dbh->disconnect();

终止与此句柄相关的连接。如果该脚本退出时, 连接仍然有效, 则显示一条警告, 并自动终止此连接。

■ \$rows = \$dbh->do (\$statement [, \%attr [, @bind_values]);

准备并运行 \$statement 表示的查询。返回值为受影响的行数 —— 如果不知道行数, 则返回 -1, 如果出现错误, 则返回 undef。如果受影响的行数为 0, 则返回值为字符串 “0E0”, 作为数值它与 0 等价, 但在判断时它为真。

不检索行的语句, 如 DELETE、INSERT、REPLACE 或 UPDATE, 主要使用 do()。如果对 SELECT 语句使用它, 则不会获得返回的语句句柄, 也不能提取任何行。

当语句包括占位符 (在此查询字符串内部由 ‘?’ 字母表示) 时, 使用 @bind_values。@bind_values 为给占位符赋值的值的列表。它必须和占位符有一样多的值。如果只指定赋值的值, 但没有指定属性, 则将 undef 作为 \%attr 参数的值来传递。

■ \$rc = \$dbh->ping();

检查与服务器的连接是否仍然有效, 并相应返回真或假。

■ \$sth = \$dbh->prepare (\$statement [, \%attr]);

为以后的执行所准备的由 \$statement 表示的查询, 并返回一个语句句柄。返回的句柄可用于 execute(), 以便执行该查询。

■ `$str = $dbh->quote ($value [, $data_type]);`

处理字符串以实现 SQL 语句中特定字符的引用和转义,以便在执行这条语句时,该字符串不引起语法错误。例如,“‘ I ’ m happy ’”(没有双引号)返回字符串“ I ’ m happy”。如果 \$value 为 undef,则它返回字符串“ NULL”(没有引号)。

一般来说, \$data_type 参数不是必需的,因为 MySQL 将查询中指定为字符串的值自动地转换为其他类型。可以将 \$data_type 指定为特殊类型值的提示——例如, DBI::SQL_INTEGER 指出 \$value 表示一个整数。

不要使用具有打算利用占位符插入到查询中的值的 quote()。DBI会自动引用这样的值。

■ `$ary_ref = $dbh->selectall_arrayref ($statement [, \%attr [, @bind_values]]);`

执行由 \$statement 指定的查询,并结合 prepare()、execute() 和 fetchall_arrayref() 返回结果。如果出现错误,则返回 undef。

如果 \$statement 参数是以前准备的语句,则省略 prepare() 步骤。

@bind_values 参数和 do() 方法中的该参数具有同样的意义。

■ `@ary_ref = $dbh->selectcol_arrayref ($statement [, \%attr [, @bind_values]]);`

执行由 \$statement 指定的查询,并通过组合 prepare() 和 execute() 返回结果的第一列。返回结果作为对含有每行第一列的数组的引用。如果出现错误,则返回 undef。

如果 \$statement 参数是以前准备的语句,则省略 prepare() 步骤。

@bind_values 参数和 do() 方法中的该参数具有同样的意义。

■ `@row_ary = $dbh->selectrow_array ($statement [, \%attr [, @bind_values]]);`

执行由 \$statement 指定的查询,并结合 prepare()、execute() 和 fetchall_arrayref() 返回结果的第一行。

如果参数 \$statement 是以前准备的语句,则省略 prepare() 步骤。

如果在列表的上下文中调用时, selectrow_array() 返回代表行值的数组,或者,如果出现错误,则返回空数组。在标量的上下文中, selectrow_array() 返回这个数组的第一个元素的值(行的第一列)。如果出现错误,则返回 undef。

@bind_values 参数和 do() 方法中的相应参数具有同样的意义。

G.2.3 语句句柄方法

本节中的这些方法通过语句句柄来调用,句柄可通过调用 prepare() 获得。

■ `$rc = $sth->bind_col ($col_num, \ $var_to_bind);`

将 SELECT 查询的给定列与 Perl 变量相联系,将它作为引用传递。\$col_num 的范围为 1 到查询选择的列数。每次提取行时,这个变量用列值自动更新。

bind_col() 应该在 execute() 之前及 prepare() 之后调用。

如果列号范围不在 1 到查询选择的列数之间,则 bind_col() 返回假。

■ `$rc = $sth->bind_columns (\ $var_to_bind1, \ $var_to_bind2, ...);`

将一系列变量与由准备好的 SELECT 语句返回的列相联系,请参阅 bind_col() 方法的说明。

如果引用的数量与查询选择的列数不匹配, 则 `bind_columns()` 返回假。

■ `$rv = $sth->bind_param ($n, $value [, \%attr]);`

`$rv = $sth->bind_param ($n, $value [, $bind_type]);`

在一个语句中, 将值与占位符 ‘?’ 相联系。应该在 `execute()` 之前及 `prepare()` 之后调用它。

`$n` 指定了占位符的数量, 应该限定 `$value` 值, 而且该值范围应该为 1 到占位符的数量。为了限定 `NULL` 值, 可传递 `undef`。

参数 `\%attr` 或者 `$bind_type` 可作为要联系的值的类型提示。例如, 要指定表示整数的值, 可以用下面两种方式调用 `bind_param()` :

```
$rv = $sth->bind_param ($n, $value , { TYPE => DBI::SQL_INTEGER });
```

```
$rv = $sth->bind_param ($n, $value , DBI::SQL_INTEGER);
```

缺省值是将变量作为 `VARCHAR` 类型。这通常就足够了, 因为 `MySQL` 将查询中字符串的值转换为所需的其他数据类型。

■ `$rows = $sth->dump_results ($maxlen [, $line_sep [, $field_sep [, $fh]]]);`

从语句句柄 `$sth` 中提取所有的行, 通过调用实用函数 `DBI::neat_list()` 将他们格式化, 并将他们打印到给定的文件句柄中。返回提取的行数。

`$maxlen`、`$line_sep`、`$field_sep`和`$fh`的缺省值分别为35、“\n”、“,”和`STDOUT`。

■ `$rv = $sth->execute (@bind_values);`

执行准备好的语句。如果该语句执行成功, 则返回真, 如果发生错误, 则返回 `undef`。

参数 `@bind_values` 与 `do()` 方法中的有相同的意义。

■ `$ary_ref = $sth->fetch();`

`fetch()` 是 `fetchrow_arrayref()` 的别名。

■ `$tbl_ary_ref = $sth->fetchall_arrayref ($slice_array_ref);`

`$tbl_ary_ref = $sth->fetchall_arrayref ($slice_hash_ref);`

从语句句柄 `$sth` 中提取所有行, 并返回数组的引用, 这个数组包含提取的每行的一个引用。数组中每个引用的意义取决于所传递的参数。没有参数或者只有数组部分引用参数, 则 `$tbl_ary_ref` 的每个元素都是包括结果集的一行值的数组引用。对于散列部分的引用参数, `$tbl_ary_ref` 的每个元素就是对包含结果集的一行值的散列引用。

■ `@ary_ref = $sth->fetchrow_array();`

当在一个列表的范围内调用时, `fetchrow_array()` 返回包含结果集下一行列值的数组, 如果不再有行或者发生错误, 则 `fetchrow_array()` 返回一个空数组。在标量上下文中, `fetchrow_array()` 返回数组第一个元素的值 (那就是说, 行的第一列), 如果不再有行或者发生错误, 则 `fetchrow_array()` 返回 `undef`。

通过检查 `$sth->err()`, 可以将结果集正常结束与出现错误区分开来。零值表明已经无错误地到达了结果集的末尾。

■ `@ary_ref = $sth->fetchrow_arrayref();`

返回一个包括结果集的下一行列值的数组引用。如果不再有行或者发生错误, 则返回 `undef`。

通过检查 `$sth->err()`, 可以将结果集正常结束与出现错误区分开来。零值表明已经无错误地到达了结果集的末尾。

■ `$shash_ref = $sth->fetchrow_hashref ([ $name ] );`

返回包括结果集的下一行列值的散列引用。如果不再有行或者发生错误，则返回 `undef`。散列是索引值是列名称，散列的元素是列值。

对于散列的关键值，指定变量 `$name` 说明使用的语句句柄属性。缺省值为“`NAME`”。这可能导致查询中的列名称不区分大小写的问题，但是散列键是区分大小写的。要强迫散列键为大写字母或者小写字母，可以指定“`NAME_lc`”或“`NAME_uc`”的 `$name` 值。

通过检查 `$sth->err()`，可以将结果集正常结束与出现错误区分开来。零值表明已经无错误地到达了结果集的末尾。

■ `$src = $sth->finish();`

释放有关语句句柄的任何资源。通常不必显式地调用这个方法，但是如果只提取部分结果集，则调用 `finish()` 使 DBI 了解已经提取了数据。调用 `finish()` 可能使语句属性无效，最好在调用 `execute()` 之后立即访问它们。

■ `$srv = $sth->rows();`

返回与 `$sth` 相关的语句所作用的行数，如果发生错误，则返回 `-1`。使用这个方法主要用于不返回行的语句。对于 `SELECT` 语句，不能依赖 `rows()` 方法在提取行时统计行数。

G.2.4 通用句柄方法

本节中的这些方法不是专用于特定类型的句柄的。可用驱动程序、数据库或语句句柄来调用它们。

■ `$h->err()`

返回最近调用的驱动程序操作的数字错误代码。0 表示没有错误。

■ `$h->errstr()`

返回最近调用的驱动程序操作的字符串错误消息。空字符串表示没有错误。

■ `DBI->trace ($trace_level [, $trace_filename]);`

`$h->trace ($trace_level [, $trace_filename]);`

设置跟踪级别。跟踪提供有关 DBI 操作的信息。跟踪级别的范围从 0（关闭）到 9（最多信息）。通过作为 DBI 类方法或独立的句柄调用跟踪，跟踪可以启用脚本内部的所有 DBI 操作：

`DBI->trace (2);` 打开脚本跟踪

`$sth->trace (2);` 打开句柄跟踪

通过设置 `DBI_TRACE` 环境变量，也可以对运行的所有 DBI 脚本在全局级别启用跟踪。缺省时，跟踪输出到 `STDERR`。提供的 `$filename` 参数可以直接将结果输出到不同的文件。将输出添加到这个文件的任何已有内容后面。

每个跟踪调用导致来自所有跟踪的句柄中的输出进入相同的文件。如果文件已命名，则所有跟踪就输出到那个文件。如果没有命名的文件，则所有跟踪输出到 `STDERR`。

■ `DBI->trace_msg ($str [, $min_level])`

`$h->trace_msg ($str [, $min_level])`

如果跟踪这个句柄或如果在 DBI 级启用跟踪，则编写这个跟踪输出的消息。如果启用 DBI 级的跟踪，则 `trace_msg()` 可以作为 `DBI->trace_msg()` 来调用，编写消息。只有在跟踪级

别至少为这个级别时，才可以提供 \$min_level 参数来指定应该编写的消息。

G.2.5 MySQL 的特定管理方法

本节介绍 DBI 作为直接访问驱动程序的手段所供的 func() 函数方法。

```
$src = $drh->func('createdb',
                 $db_name, $host_name, $user_name, $password, 'admin');
$src = $drh->func('dropdb',
                 $db_name, $host_name, $user_name, $password, 'admin');
$src = $drh->func('shutdown',
                 $host_name, $user_name, $password, 'admin');
$src = $drh->func('reload',
                 $host_name, $user_name, $password, 'admin');

$src = $dbh->func('createdb', $db_name, 'admin');
$src = $dbh->func('dropdb', $db_name, 'admin');
$src = $dbh->func('shutdown', 'admin');
$src = $dbh->func('reload', 'admin');
```

通过驱动程序句柄或通过数据库句柄访问 func() 方法。驱动程序句柄与打开的连接无关，所以，如果以这种方式访问 func()，则必须提供允许这个方法创建连接的主机名称、用户名称和口令的参数。如果用数据库句柄访问 func()，则不需要那些参数。如果需要，可以像下面这样获得驱动程序句柄：

```
$drh = DBI->install_driver("mysql"); # ("mysql" must be lowercase)
```

- **createdb** 创建由 \$db_name 指定的数据库。要这样做，必须对该数据库拥有 CREATE 权限。
- **dropdb** 删除由 \$db_name 指定的数据库。要这样做，必须对该数据库拥有 DROP 权限。当心，如果删除了一个数据库，则它将会消失，且再也不能恢复。
- **shutdown** 关闭服务器。必须具有 SHUTDOWN 权限。
- **reload** 告诉服务器重新加载授权表。如果直接使用 DELETE、INSERT 或 UPDATE 而不是使用 GRANT 或 REVOKE 来修改这个授权表的内容，则这是必需的。要使用 reload，必须具有 RELOAD 权限。

G.3 DBI 实用程序函数

这些函数如像 DBI::func_name() 而不是 DBI->func_name() 那样调用。

■ **@bool = DBI::looks_like_number (@array);**

产生一列值，并返回一个数组，列表的每个元素都为数组的一个成员。每个成员表示相应的参数是否为数字：如果是，则返回真，如果不是，则返回假，如果没有定义参数或参数为空，则返回 undef。

■ **\$str = DBI::neat (\$value [, \$maxlen]);**

返回含有 \$value 参数的格式化表示的字符串。字符串被引用；数字则不被引用（但是，请注意，引用的数字被看作是字符串）。未定义的值作为 undef 报告，不可打印的字符作为 ‘.’ 报告。

\$maxlen 参数控制结果的最大长度。如果结果比 \$maxlen 长，则将 \$maxlen 缩短——加

上4个字符和省略号。如果 `$maxlen` 为0、`undef`、或省略，则缺省值为 `$DBI::neat_maxlen` (400)。

对查询结构不要使用 `neat()`。如果需要实现引用或转义，应该使用 `quote()`。

■ `$str = DBI::neat_list (\@listref [, $maxlen [, $field_sep]]);`

对第一个元素指向的列表的每个元素调用 `neat()`，将它们与分隔符字符串 `$field_sep` 连接起来，并作为字符串返回结果。

`$maxlen` 参数用于单独的参数，而不是用于调用 `neat()` 的结果字符串。

如果省略 `$field_sep`，则缺省值为“，”。

G.4 DBI属性

DBI 在几个级别上提供了属性信息。大多数属性与数据库句柄相关，或者与语句句柄相关，但不是与两者都相关。一些属性，例如 `PrintError` 和 `RaiseError`，可能既与数据库句柄相关，又与语句句柄相关。一般来说，每个句柄都有自己的属性，但是一些保留错误信息的属性，如 `err` 和 `errstr`，与最近使用的句柄是动态相关的。

G.4.1 通用句柄属性

这些属性可以用于单独的句柄，或者对于取得影响该方法操作的参数的方法，在 `%attr` 参数中进行指定。如果用于 `connect()` 方法，则在整个脚本中将全局性地影响 DBI 的处理。

■ `$h->{ 'ChopBlanks' }`；确定提取行的方法是否从 `CHAR` 列值中剪裁掉后面的空白。

对于大多数数据库驱动程序来说，缺省值为禁用 `ChopBlanks`，但是，对 `MySQL` 不会产生影响，因为服务器始终去掉 `CHAR` 值后的空白。

■ `$h->{ 'PrintError' }`；如果启用，则出现有关 DBI 的错误时会显示警告消息。缺省为启用 `PrintError`。

■ `$h->{ 'RaiseError' }`；如果启用，则出现与 DBI 有关的错误会使脚本自动终止。缺省为启用 `RaiseError`。

G.4.2 动态属性

这些属性与最近使用的句柄有关，在下面的描述中用 `$h` 来表示：

■ `$DBI::err` 与 `$h->err()` 调用相同。

■ `$DBI::errstr` 与 `$h->errstr()` 调用相同。

■ `$DBI::rows` 与 `$h->rows()` 调用相同。

G.4.3 MySQL特定的数据库句柄属性

这些属性是专门针对 DBI `MySQL` 驱动程序 `DBD::mysql` 的。

■ `$str = $dbh->{ 'info' }`；此属性含有与 C API 函数 `mysql_info()` 返回的信息相同的信息。请参阅附录 F 中该函数的描述。

■ `$rv = $dbh->{ 'mysql_insertid' }`；`AUTO_INCREMENT` 的值是最近在与 `$dbh` 有关的连接上生成的。

■ `$rv = $dbh->{ 'thread_id' }`；与 `$dbh` 相关的连接的线程数。

G.4.4 语句句柄属性

通常，这些属性用于 SELECT 查询，而且在将查询传递给 prepare() 获得了语句句柄，而且为这个句柄调用了 execute() 时，这些属性才有效。此外，finish() 可能会使某些语句属性无效。

许多属性具有作为数组值引用的值，查询的每列都有一个值。数组中元素的数量由属性 \$sth->{ ' NUM_OF_FIELDS ' } 给出。语句属性 stmt_attr 是对数组的引用，可以用 @{\$sth->{stmt_attr}} 访问整个数组，或者由如下数组元素的循环来访问：

```
for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
{
    $value = $sth->{stmt_attr}->{$i};
}
```

- \$sth->{ ' NAME ' }; 对表示每列的名称的字符串数组的引用。这些名称的大小写字符与 SELECT 语句中给出的相同。
- \$sth->{ ' NAME_lc ' }; 表示每列名称的字符串数组的引用。返回的名称为小写字符串。
- \$sth->{ ' NAME_uc ' }; 表示每列名称的字符串数组的引用。返回的名称为大写字符串。
- \$sth->{ ' NULLABLE ' }; 表示每列是否可以为 NULL 值的数组引用。每个元素的值可为0（非）、1（是）或2（不知道）。至少，DBI 文档是这样说的。如果列不为 NULL，则它实际显示该值为空字符串。
- \$sth->{ ' NUM_OF_FIELDS ' }; 准备好的语句将返回的列数，或对非 SELECT 语句返回0。
- \$sth->{ ' NUM_OF_PARAMS ' }; 准备好的语句中的占位符数量。
- \$sth->{ ' PRECISION ' }; 表示每列精度的值的数组引用。DBI 采用 ODBC 意义中的“精度”，对MySQL意味着该列的最大宽度。对于数值列，这是显示宽度。对于字符串列，它是该列的最大长度。
- \$sth->{ ' SCALE ' }; 表示每列的范围的值的数组引用。DBI 采用 ODBC 意义中的“范围”，对MySQL意味着浮点列的小数位置。对其他列来说，这个范围为 0。
- \$sth->{ ' Statement ' }; 与 \$sth 相关的语句文本。在进行任何占位符替换以前，这个文本是 prepare() 所看到的。
- \$sth->{ ' TYPE ' }; 表示每列的数值类型的值的数组引用。

G.4.5 MySQL 特定的语句句柄属性

大多数语句句柄属性都被认为是只读的，并且应该在 execute() 调用之后访问。但 mysql_store_result 和 mysql_use_result 例外。DBD::mysql 提供控制您的脚本所用的结果集处理风格的功能。这条语句的句柄属性 mysql_store_result 和 mysql_use_result 选择C API 函数 mysql_store_result() 和 mysql_use_result() 的结果集处理的行为。有关这两个函数的讨论及它们的不同，请参阅附录F。

缺省设置时，DBI 使用 mysql_store_result()，但是可以启用 mysql_use_result 属性，它告

诉 DBI 使用 `mysql_use_result()`。在 `prepare()` 之后及 `execute()` 之前做这件事：

```
$sth = $dbh->prepare (...);
$sth->{mysql_use_result} = 1;
$sth->execute();
```

DBD::mysql 的旧版本中可用的若干 MySQL 专用属性现在不赞成使用，已经由更新更好的格式替换，如表 G-2 所示。如果 DBD::mysql 是旧版本，不支持这些更新的属性，则试着用这些不赞成的格式（否则应升级为更高的版本）。

请注意，`insertid` 为语句句柄属性，而它的首选形式 `mysql_insertid` 是数据库句柄属性。

表 G-2 不赞成使用的 MySQL 专用属性

不赞成的属性	首选的属性
<code>insertid</code>	<code>mysql_insertid</code>
<code>is_blob</code>	<code>mysql_is_blob</code>
<code>is_key</code>	<code>mysql_is_key</code>
<code>is_not_null</code>	<code>mysql_is_not_null</code>
<code>is_num</code>	<code>mysql_is_num</code>
<code>is_pri_key</code>	<code>mysql_is_pri_key</code>
<code>length</code>	<code>PRECISION</code>
<code>max_length</code>	<code>mysql_max_length</code>
<code>table</code>	<code>mysql_table</code>

■ `$sth->{ 'mysql_is_blob' }`；表示每列是否为 BLOB 类型的值的数组引用。

■ `$sth->{ 'mysql_is_key' }`；表示每列是否为非唯一键的值的数组引用。

■ `$sth->{ 'mysql_is_not_null' }`；表示每列是否可为 NULL 的值的数组引用。假值表示该列可包括 NULL 值。这个属性的信息为 NULLABLE 属性信息中“反转”的信息。

■ `$sth->{ 'mysql_is_num' }`；表示每列是否可为数值类型的值的数组引用。

■ `$sth->{ 'mysql_is_pri_key' }`；表示每列是否可为 PRIMARY KEY 的组成部分的值的数组引用。

■ `$sth->{ 'mysql_max_length' }`；表示结果集中列值实际的最大长度的值的数组引用。

■ `$sth->{ 'mysql_store_result' }`；如果启动 `mysql_store_result`（设为 1），则利用 `mysql_store_result` C API 函数，而非 `mysql_store_result` 从 MySQL 服务器中检索结果集。关于这两个函数及它们之间区别的介绍请参阅附录 F。

如果设置 `mysql_store_result` 属性，则在 `prepare()` 调用之后及 `execute()` 调用之前再设置。

■ `$sth->{ 'mysql_table' }`；表示列来自其中的表名的值的数组引用。计算列的表名为空字符串。

■ `$sth->{ 'mysql_type' }`；表示结果集中每列的 MySQL 类型的数的值的数组引用。

■ `$sth->{ 'mysql_type_name' }`；表示结果集中每列的 MySQL 类型的名的值的数组引用。

■ `$sth->{ 'mysql_use_result' }`；如果启动 `mysql_use_result`（设为 1），则使用

mysql_use_result C API 函数, 而不是 mysql_use_result 从 MySQL 服务器中检索结果集。请参阅附录F关于这两个函数及它们之间区别的讨论。

请注意, 使用这个属性会导致一些属性成为非法的, 如 mysql_max_length。虽然在以任何方式提取它们的时候最好对行进行计数, 但是, 它也会使 rows() 方法的使用无效。

如果设置 mysql_use_result 属性, 则在 prepare() 调用之后及 execute() 调用之前再设置。

G.5 DBI 环境变量

DBI 考虑了几个环境变量, 如表 G-3 所示。除了 DBI_TRACE 之外, 所有变量都由 connect() 方法使用。DBI_TRACE 由 trace() 方法使用。

表G-3 DBI 环境变量

名 称	含 义
DBI_DRIVER	DBI 级的驱动程序名 (MySQL的 “mysql”)
DBI_DSN	数据源名
DBI_PASS	口令
DBI_TRACE	跟踪级别和/或跟踪输出文件
DBI_USER	用户名称

附录 H PHP API 参考

本附录描述了 MySQL 的 PHP 应用程序编程接口 (API)。API 由与 MySQL 服务器通信和访问数据库的一组函数组成。

由于这个附录是参考性的, 因此它只包含说明 PHP API 应用的简短代码段。要想获得它们的全部客户机脚本和编写注意事项, 请参阅第 8 章。这里介绍的函数是完全适合 MySQL 的函数。PHP 手册 (现在大约有 800 页) 包括了 600 多页参考材料, 因此很明显, 这个附录说明的内容还不到 PHP 性能概要的一小部分。要获得完整的 PHP 手册, 请访问下面的 PHP Web 网站:

<http://www.php.net/>

H.1 编写 PHP 脚本

PHP 脚本是可以包括 HTML 和 PHP 代码混合的无格式文本文件。对脚本的解释生成一个 Web 页面, 作为发送到客户机的输出。HTML 不用解释就可拷贝到输出中。PHP 代码需要进行解释, 并用该代码所生成的输出 (可能没有输出) 来替换。

PHP 开始在 HTML 模式下解释文件。可以使用表明 PHP 代码开始和结束的特殊标记进行 PHP 代码模式的转换。在一个文件中可以在此两种模式之间进行多次切换。PHP 支持四种类型的标记, 虽然其中一些标记在使用它们时需要明确启用。一种方法就是在 PHP 初始化文件 php3.ini 中将它们打开。这个文件的位置由系统决定, 通常位于 /usr/local/lib 中。

PHP 支持下面的标记风格:

■ 缺省的标记风格, 用 '`<?php`' 和 '`?>`' 标记:

```
<?php echo ("Some PHP code here"); ?>
```

■ 简短打开的标记风格, 用 '`<?`' 和 '`?>`' 标记:

```
<? echo ("Some PHP code here"); ?>
```

这种标记风格由 PHP 初始文件的一个命令来启用:

```
short_open_tag = On;
```

■ 与活动服务器页面兼容的风格, 用 '`<%`' 和 '`%>`' 标记:

```
<% echo ("Some PHP code here"); %>
```

ASP 风格的标记由 PHP 初始文件的一个指令来启用:

```
asp_tags = On;
```

对 ASP 标记的支持是在 PHP 3.0.4 中引入的。

■ 如果使用的是不能支持其他标记的 HTML 编辑器, 可以使用 `<SCRIPT>` 和 `</SCRIPT>` 标记:

```
<SCRIPT LANGUAGE="php"> echo ("Some PHP code here"); </SCRIPT>
```

H.2 函数

下面的描述讨论了与 MySQL 相关的 PHP 的每个函数。在方括号 ([]) 中指出可选参数。

许多函数用一个可选的 `link_id` 参数（连接标识符）表示与 MySQL 服务器的连接。如果函数调用中没有连接标识符，则使用最近打开的连接。如果没有指定连接，并且也没有打开的连接，有的函数会设法建立连接。

参数 `result_id` 指出了结果集标识符，一般是由 `mysql_db_query()` 或者 `mysql_query()` 返回的。

除了返回状态值之外，如果出现错误，一些函数还会产生错误消息。在 Web 环境下，这个消息出现在发送到客户机浏览器的输出中。一般来说，在函数名称的前面加一个字符 '@' 取消了（可能是隐含的）函数产生的错误消息。例如，要想从 `mysql_pconnect()` 调用中取消错误消息，以便可以以更合适的方式报告失败，则可以如下进行：

```
<?php
$link = @mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
?>
```

另一种方法是利用函数 `error_reporting()` 来取消错误消息：

```
<?php
error_reporting (0); # suppress all error messages
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
?>
```

本附录中的许多实例脚本在建立了与 MySQL 服务器的连接之后，都将打印“Connected successfully”。它们这样做的原因是要确认脚本打印了一些输出，以免要自己去试这个脚本。没有产生输出的 PHP 脚本在一些浏览器中引发了“页面不包括数据”的警告。

在下面的描述中，术语“SELECT 查询”用来指 SELECT 查询或者其他返回行的查询，如 DESCRIBE、EXPLAIN 或者 SHOW。

H.2.1 连接管理例程

本节的例程允许打开和关闭对 MySQL 服务器的连接。

■ int

mysql_close ([int link_id]);

关闭由 `link_id` 标识的与 MySQL 服务器的连接。如果没有指定连接，则 `mysql_close()` 关闭最近打开的连接。如果成功，则 `mysql_close()` 返回真，失败则返回假。对由 `mysql_pconnect()` 打开的永久连接，`mysql_close()` 忽略相应的关闭请求，只是返回真。如果要关闭一个连接，就应该用 `mysql_connect()` 而不是 `mysql_pconnect()` 来打开它。

```
<?php
$link = mysql_connect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
mysql_close ($link);
?>
```

■ int

mysql_connect ([string host_name] [, string user_name [, string password]]);

在主机 `host_name` 上对给定口令的用户 `user_name` 打开与 MySQL 服务器的连接。连接成功，则返回与新连接相关的连接标识符（正数），如果出现错误，则返回假。

自 PHP 3.0B4 起，主机名称参数就由可选的端口号以 “`host_name:port_num`” 形式指定。自 PHP 3.0.10 起，如果主机名称是 `localhost`，则主机名称参数就可以由可选路径以 “`localhost:socket_name`” 形式指定 UNIX 领域的套接字路径。应该将套接字指定为完整的路径名称。

如果没有主机名称参数，则缺省值为 `localhost`。如果没有用户名称，或者用户名称为空，则缺省值为 PHP 运行的用户名称（这是 Web 正在处理的用户名，好像 PHP 按 Apache 模块运行一样，或者是 PHP 正在处理的用户名，好像 PHP 作为一个独立程序运行一样）。如果没有口令参数或者口令参数为空，就发送空口令。

在打开连接时，如果用同样参数（主机名称、用户名称和口令）调用 `mysql_connect()`，不会产生新的连接，`mysql_connect()` 将返回已有的连接标识符。

这个连接可以通过调用 `mysql_close()` 来关闭。在脚本终止时，如果连接是打开的，则自动关闭该连接。

```
<?php
$link = mysql_connect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
mysql_close ($link);
?>
```

■ int

```
mysql_pconnect ([string host_name] [, string user_name [, string
password]]);
```

`mysql_pconnect()` 除了打开一个永久的连接之外，与 `mysql_connect()` 类似，即，在脚本终止时，这个连接仍然是打开的。当连接打开时，如果用同样的连接参数（主机名称、用户名称和口令）调用 `mysql_pconnect()`，将再次使用连接。这就避免了删除和再次打开连接的开销，并且比非永久的连接效率更高。

永久连接只在 PHP 作为在脚本终止之后继续运行的 Web 服务器内部的模块执行时有意义。在独立的 PHP 版本执行的脚本中，当脚本终止时连接关闭，因为 PHP 处理也终止了。

在永久的连接上调用 `mysql_close()` 是无意义的，在此情形下，`mysql_close()` 返回真，但是连接仍然是打开的。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
?>
```

H.2.2 状态报告和错误报告例程

函数 `mysql_errno()` 和 `mysql_error()` 返回与 MySQL 有关的 PHP 函数的错误号或错误信息。然而，不会从无有效连接标识符的任何函数得到错误信息。这表示它们对报告失败的 `mysql_errno()` 和 `mysql_error()` 调用的结果没有用处，因为在成功建立连接之前，不能得到连

接标识符。如果想获得连接失败的 MySQL 错误消息, 可用 PHP 初始化文件的一条命令来启用变量 `track_errors` :

```
track_errors = On;
```

然后, 如果 PHP 按 Apache 模块运行, 则重新启动 Web 服务器。完成之后, 可通过访问变量 `$php_errormsg` 获得连接失败的错误字符串:

```
<?php
$link = @mysql_connect("badhost","baduser", "badpass")
    or die ("Could not connect: " . $php_errormsg);
print ("Connected successfully");
?>
```

■ int

```
mysql_errno ([int link_id]);
```

对于给定的连接, 返回最近返回状态的与 MySQL 相关的函数错误号。零值表示未出现错误。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("query failed, error code = " . mysql_errno ());
?>
```

■ string

```
mysql_error ([int link_id]);
```

对于给定的连接, 返回含有最近返回状态的与 MySQL 相关的函数的错误消息字符串。空值意味着未出现错误。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("query failed, error message = " . mysql_error ());
?>
```

H.2.3 查询的构造与执行例程

本节的例程用来发布到 MySQL 服务器的查询。

■ int

```
mysql_db_query (string db_name, string query [, int link_id]);
```

`mysql_db_query()` 除了提取一个额外的数据库名称参数, 并在执行查询之前使它成为缺省的数据库之外, 与 `mysql_query()` 类似 (可将下面的实例与 `mysql_query()` 的实例做一下比较)。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
```

```

print ("Connected successfully");
$query = "SELECT * FROM president";
$result = mysql_db_query ("samp_db", $query)
    or die ("Query failed");

```

?>

■ int

mysql_list_dbs ([int link_id]);

返回结果标识符，标识给定的连接中这个服务器知道的由数据库名称所组成的结果集，结果集中的每行都是一个数据库名。如果出现错误，则返回假。不必选择缺省的数据库。结果集可由任何通用的提取行的函数或者由 `mysql_tablename()` 来处理。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
$result = mysql_list_dbs ()
    or die ("Query failed");
print ("Databases (using mysql_fetch_row()):<BR>\n");
while ($row = mysql_fetch_row ($result))
    printf ("%s<BR>\n", $row[0]);
$result = mysql_list_dbs ()
    or die ("Query failed");
print ("Databases (using mysql_tablename()):<BR>\n");
for ($i = 0; $i < mysql_num_rows ($result); $i++)
    printf ("%s<BR>\n", mysql_tablename ($result, $i));

```

?>

■ int

mysql_list_fields (string db_name, string tbl_name [, int link_id]);

对于包括表中有关列的信息的结果集，返回一个结果标识符，如果出现错误，则返回假。不必选择缺省的数据库。db_name 和 tbl_name 参数指定了感兴趣的数据库和表。结果标识符用于函数 `mysql_field_flags()`、`mysql_field_len()`、`mysql_field_name()` 和 `mysql_field_type()`。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
$result = mysql_list_fields ("samp_db", "member")
    or die ("Query failed");
print ("member table column information:<BR>\n");
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,\n", mysql_field_name ($result, $i));
    printf (" len %d,\n", mysql_field_len ($result, $i));
    printf (" type %s,\n", mysql_field_type ($result, $i));
    printf (" flags %s\n", mysql_field_flags ($result, $i));
    print "<BR>\n";
}

```

?>

■ int

mysql_list_tables (string db_name [, int link_id]);

在给定的数据库名称中，对于由表的名称组成的结果集，返回一个结果标识符，结果集的每行都是一个表名称。如果出现错误，则返回假。不必选择缺省的数据库。结果集可以由任何通用的提取行的函数或者由 `mysql_tablename()` 来处理。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
or die ("Could not connect");
$result = mysql_list_tables ("samp_db")
or die ("Query failed");
print ("samp_db tables (using mysql_fetch_row()):<BR>\n");
while ($row = mysql_fetch_row ($result))
    printf ("%s<BR>\n", $row[0]);
$result = mysql_list_tables ("samp_db")
or die ("Query failed");
print ("samp_db tables (using mysql_tablename()):<BR>\n");
for ($i = 0; $i < mysql_num_rows ($result); $i++)
    printf ("%s<BR>\n", mysql_tablename ($result, $i));
?>
```

■ int

mysql_query (string query [, int link_id]);

在给定的连接上，将查询字符串发送给 MySQL 服务器。对于 DELETE、INSERT、REPLACE 和 UPDATE 语句，如果 `mysql_query()` 成功，则返回真；如果出现错误，则返回假。对于成功的查询，可以调用 `mysql_affected_rows()` 找出所修改的行数。

对 SELECT 语句，如果成功，则 `mysql_query()` 返回一个正的结果集标识符，如果出现错误，则返回假。对于成功的查询，结果集标识符可用于提取 `result_id` 参数的各种结果集处理函数。可以将这个标识符传递给 `mysql_free_result()` 以释放与结果集相关的任何资源。

一个“成功的”查询是无错误执行的查询，但是成功并不意味着查询会返回一些行。下面的查询是非常合法的，只是不返回行：

```
SELECT * FROM president WHERE 1 = 0
```

查询失败可能有许多原因。例如，可能在语句构成上是畸形的、语法是非法的或者不符合规定的，因为您不具有查询中指定表的访问权。

如果没有指定连接标识符，就使用最近打开的连接。如果不存在当前连接，则 `mysql_query()` 试图打开一个连接，就像无参数地调用 `mysql_connect()` 一样。如果连接失败，则 `mysql_query()` 失败。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
or die ("Could not connect");
print ("Connected successfully");
mysql_select_db ("samp_db")
or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
or die ("Query failed");
?>
```

H.2.4 结果集处理例程

本节的例程用来检索查询的结果。它们也提供了有关结果信息的访问，如涉及多少行的

信息，或者结果集列的元数据信息等。

■ int

mysql_affected_rows ([int link_id]);

在给定的连接中，返回由最近的 DELETE、INSERT、REPLACE 或者 UPDATE 语句所作用的行数。如果没有行被修改，则 mysql_affected_rows() 返回 0，如果出现错误，则返回 -1。

在 SELECT 查询之后，mysql_affected_rows() 返回所选择的行数。但一般是与 SELECT 语句一道使用 mysql_num_rows()。

<?php

```
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "INSERT INTO member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2002-6-3')";
$result = mysql_query ($query)
    or die ("Query failed");
printf ("%d row%s inserted\n",
        mysql_affected_rows (),
        mysql_affected_rows () == 1 ? "" : "s");
```

?>

■ int

mysql_data_seek (int result_id, int row_num);.

由 SELECT 查询返回的每个结果集都有一个行游标，指示下一个提取行的函数 (mysql_fetch_array()、mysql_fetch_object() 或者 mysql_fetch_row()) 调用将返回哪一行。mysql_data_seek() 将给定结果集的指针设置到给定的行。行号的范围为 0 到 mysql_num_rows() - 1。如果行号合法，则 mysql_data_seek() 返回真，否则返回假。

<?php

```
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
# fetch rows in reverse order
for ($i = mysql_num_rows ($result) - 1; $i >=0; $i--)
{
    if (!mysql_data_seek ($result, $i))
    {
        printf ("Cannot seek to row %d\n", $i);
        continue;
    }
    if (!($row = mysql_fetch_object ($result)))
        continue;
    printf ("%s %s<BR>\n", $row->last_name, $row->first_name);
}
mysql_free_result ($result);
```

?>

■ array


```
mysql_fetch_array (int result_id [, int result_type]);
```

作为一个数组返回给定结果集的下一行。如果没有更多的行，则返回假。数组含有按数值列索引和按列名称关联的键索引所存储的值。换句话说，每个列值都可以用它的数值列索引或者它的名称来访问。关联索引是区分大小写的，并且给出时必须与查询中使用的列名称的大小写相同。假设发布下面的查询：

```
SELECT last_name, first_name FROM president
```

如果从结果集中获取行，放到一个名为 \$row 的数组中，其数组元素可以这样访问：

```
$row[0]                保存 last_name 值
$row[1]                保存 first_name 值
$row["last_name"]      保存 last_name 值
$row["first_name"]     保存 first_name 值
```

键不受相应列的表名的限制，因此如果选择了不同表中具有相同名称的列，结果会发生名称冲突。在查询选择的列的列表中，最后指定的列优先。要访问隐藏的列，应该使用数值索引，或者编写查询，为该列提供一个别名。

result_type 参数可以为 MYSQL_ASSOC（只返回名称索引值）、MYSQL_NUM（只返回数字索引值）或者 MYSQL_BOTH（返回两种类型索引值）。如果不给出 result_type，其缺省值为 MYSQL_BOTH。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_array ($result))
{
    # print each name twice, once using numeric indices,
    # once using associative (name) indices
    printf ("%s %s<BR>\n", $row[0], $row[1]);
    printf ("%s %s<BR>\n", $row["last_name"], $row["first_name"]);
}
mysql_free_result ($result);
?>
```

■ object

```
mysql_fetch_field (int result_id [, int col_num]);
```

返回结果集中给定列的相关元数据信息，如果没有这样的列，则返回假。如果省略 col_num，则对 mysql_fetch_field() 的后继调用返回结果集后续列的信息。如果不再有剩余的列，则返回值为假。如果指定了 col_num，则其取值范围为 0 到 mysql_num_fields() - 1。在此情形下，mysql_num_fields() 返回给定列的相关信息，如果 col_num 超出范围，返回假。

信息作为具有表 H-1 中所示属性的对象返回。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
```

```

$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
# get column metadata
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("Information for column %d:<BR>\n", $i);
    $meta = mysql_fetch_field ($result);
    if (!$meta)
    {
        print ("No information available<BR>\n");
        continue;
    }
    print ("<PRE>\n");
    printf ("blob:          %s\n", $meta->blob);
    printf ("max_length:    %s\n", $meta->max_length);
    printf ("multiple_key: %s\n", $meta->multiple_key);
    printf ("name:          %s\n", $meta->name);
    printf ("not_null:      %s\n", $meta->not_null);
    printf ("numeric:       %s\n", $meta->numeric);
    printf ("primary_key:   %s\n", $meta->primary_key);
    printf ("table:         %s\n", $meta->table);
    printf ("type:          %s\n", $meta->type);
    printf ("unique_key:    %s\n", $meta->unique_key);
    printf ("unsigned:      %s\n", $meta->unsigned);
    printf ("zerofill:      %s\n", $meta->zerofill);
    print ("</PRE>\n");
}
?>

```

表H-1 mysql_fetch_field() 的属性

属 性	说 明
blob	如果列为 BLOB 类型则值为1, 否则为0
max_length	结果集中最大列值的长度
multiple_key	如果列为非唯一的索引的成分则值为1, 否则为0
name	列名称
not_null	如果列不能包含 NULL 值则为1, 否则为0
numeric	如果列为数值类型则为1, 否则为0
primary_key	如果列为 PRIMARY KEY 的成分则为1, 否则为0
table	包含列的表的名称 (计算列为空)
type	列的类型的名称
unique_key	如果列为 UNIQUE 索引的成分则为1, 否则为0
unsigned	如果列具有 UNSIGNED 属性则为1, 否则为0
zerofill	如果列具有 ZEROFILL 属性则值为1, 否则为0

■ array

mysql_fetch_lengths (int result_id);

返回一个数组, 此数组含有函数 `mysql_fetch_array()`、`mysql_fetch_object()` 或 `mysql_fetch_row()` 最近提取的行中的列值长度。如果没有提取行或者出现错误, 则返回假。

```
<?php
```

```

$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");

```

```

mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
$row_num = 0;
while (mysql_fetch_row ($result))
{
    ++$row_num;
    # get lengths of column values
    printf ("Lengths of values in row %d:<BR>\n", $row_num);
    $len = mysql_fetch_lengths ($result);
    if (!$len)
    {
        print ("No information available<BR>\n");
        break;
    }
    print ("<PRE>\n");
    for ($i = 0; $i < mysql_num_fields ($result); $i++)
        printf ("column %d: %s\n", $i, $len[$i]);
    print ("</PRE>\n");
}
?>

```

■object

```
mysql_fetch_object (int result_id [, int result_type]);
```

作为一个对象返回给定结果集的下一行，如果没有更多的行，则返回假。列值可以作为对象的属性访问。这个属性的名称就是在生成结果集的查询中所选择的列的名称。

result_type 参数可能是 MYSQL_ASSOC（只返回名称索引值）、MYSQL_NUM（只返回数字索引值）或者 MYSQL_BOTH（返回两种类型索引值）。如果省略 result_type，则缺省值为 MYSQL_BOTH（假如数值不是合法的属性名，我不清楚指定 MYSQL_NUM 的作用是什么）。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_object ($result))
    printf ("%s %s<BR>\n", $row->last_name, $row->first_name);
mysql_free_result ($result);
?>

```

■array

```
mysql_fetch_row (int result_id);
```

作为一个数组返回给定结果集的下一行，如果没有更多的行，则返回假。

列值可作为数组元素访问，在 0 到 mysql_num_fields() - 1 范围内使用列索引。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")

```

```

        or die ("Could not select database");
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
        or die ("Query failed");
while ($row = mysql_fetch_row ($result))
    printf ("%s %s<BR>\n", $row[0], $row[1]);
mysql_free_result ($result);

?>

```

■ string

mysql_field_name (int result_id, int col_num);

返回结果集的给定列的名称。

col_num 的范围为 0 到 mysql_num_fields()-1。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
# get column names
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("Name of column %d: ", $i);
    $name = mysql_field_name ($result, $i);
    if (!$name)
        print ("No name available<BR>\n");
    else
        print ("{$name}<BR>\n");
}

?>

```

■ int

mysql_field_seek (int result_id, int col_num);

为随后的 mysql_fetch_field() 调用设置索引。发布没有明确列号的 mysql_fetch_field() 的下次调用, 将返回列 col_num 的信息。如果搜索成功, 返回真, 否则返回假。

col_num 的范围为 0 到 mysql_num_fields() - 1。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
# get column metadata
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("Information for column %d:<BR>\n", $i);
    if (!mysql_field_seek ($result, $i))
    {
        print ("Cannot seek to column<BR>\n");
    }
}

```

```

        continue;
    }
    $meta = mysql_fetch_field ($result, $i);
    if (!$meta)
    {
        print ("No information available<BR>\n");
        continue;
    }
    print ("<PRE>\n");
    printf ("blob:          %s\n", $meta->blob);
    printf ("max_length:    %s\n", $meta->max_length);
    printf ("multiple_key: %s\n", $meta->multiple_key);
    printf ("name:          %s\n", $meta->name);
    printf ("not_null:      %s\n", $meta->not_null);
    printf ("numeric:       %s\n", $meta->numeric);
    printf ("primary_key:   %s\n", $meta->primary_key);
    printf ("table:         %s\n", $meta->table);
    printf ("type:          %s\n", $meta->type);
    printf ("unique_key:    %s\n", $meta->unique_key);
    printf ("unsigned:      %s\n", $meta->unsigned);
    printf ("zerofill:      %s\n", $meta->zerofill);
    print ("</PRE>\n");
}
?>

```

■ string

mysql_field_table (int result_id, int col_num);

返回含有结果集给定列的表名。对于计算列，此名为空。

col_num 的范围为 0 到 mysql_num_fields() - 1。

<?php

```

$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,\n", mysql_field_name ($result, $i));
    printf (" table %s\n", mysql_field_table ($result, $i));
    print "<BR>\n";
}

```

?>

■ string

mysql_field_type (int result_id, int col_num);

返回结果集给定列的类型名。类型名在附录 B 中列出。

col_num 的范围为 0 到 mysql_num_fields() - 1。

<?php

```

$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db");
$query = "SELECT * FROM president";

```

```

$result = mysql_query ($query)
    or die ("Query failed");
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,\n", mysql_field_name ($result, $i));
    printf (" type %s\n", mysql_field_type ($result, $i));
    print "<BR>\n";
}
?>

```

■ string

mysql_field_flags (int result_id, int col_num);

作为字符串返回结果集中给定列的相关元数据信息，如果出现错误，则返回假。这个字符串由以空格分开的词组成，说明哪个列的标记值为真。对于假的标记，在字符串中给出相应的词。表 H-2 列出了可能在字符串中给出的词。

col_num 的范围为0到 mysql_num_fields() - 1。

```

<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db");
$query = "SELECT * FROM member";
$result = mysql_query ($query)
    or die ("Query failed");
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,\n", mysql_field_name ($result, $i));
    printf (" flags %s\n", mysql_field_flags ($result, $i));
    print "<BR>\n";
}
?>

```

表H-2 mysql_field_flags() 的值

| 属 性 | 说 明 |
|----------------|-----------------------|
| auto_increment | 列具有 AUTO_INCREMENT 属性 |
| binary | 列具有 BINARY 属性 |
| blob | 列为 BLOB 类型 |
| enum | 列为 ENUM 类型 |
| multiple_key | 列为非惟一索引的成分 |
| not_null | 列不能包含 NULL 值 |
| primary_key | 列为 PRIMARY_KEY 的成分 |
| timestamp | 列为 TIMESTAMP 类型 |
| unique_key | 列 UNIQUE 索引的成分 |
| unsigned | 列具有 UNSIGNED 属性 |
| zerofill | 列具有 ZEROFILL 属性 |

■ int

mysql_field_len (int result_id, int col_num);

返回结果集给定列中值可能的最大长度。

col_num 的范围为0到 mysql_num_fields()-1。

col_num 的范围为0到 mysql_num_fields()-1。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db");
$query = "SELECT * FROM member";
$result = mysql_query ($query)
    or die ("Query failed");
for ($i = 0; $i < mysql_num_fields ($result); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,\n", mysql_field_name ($result, $i));
    printf (" len %d\n", mysql_field_len ($result, $i));
    print "<BR>\n";
}
?>
```

■ int

mysql_free_result (int result_id);

释放与给定结果集相关的任何资源。在脚本终止时，会自动释放结果集，但是你也可以在生成许多结果集的脚本中显式地调用这个函数。例如，下面的脚本将占用相当数量的内存：

```
<?php
$link = mysql_connect ("localhost", "paul", "secret");
mysql_select_db ("samp_db");
for ($i = 0; $i < 10000; $i++)
{
    $result = mysql_query ("SELECT * from president");
}
?>
```

在调用 mysql_query() 之后加入一个 mysql_free_result() 调用，可将结果集内存的使用量减少到几乎没有：

```
<?php
$link = mysql_connect ("localhost", "paul", "secret");
mysql_select_db ("samp_db");
for ($i = 0; $i < 10000; $i++)
{
    $result = mysql_query ("SELECT * from president");
    mysql_free_result ($result);
}
?>
```

■ int

mysql_insert_id ([int link_id]);

在给定的连接中，返回由最近执行的查询生成的 AUTO_INCREMENT 值。在连接期间，如果不生成这样的值，则返回零。一般在期望产生新值的查询之后应该立即调用 mysql_insert_id()；如果在那个查询和要使用这个值的地方之间有其他的查询干扰，则 mysql_insert_id() 的值将被干扰查询设置为 0。

请注意，mysql_insert_id() 的性能与 SQL 函数中 LAST_INSERT_ID() 的性能不同。mysql_insert_id() 在客户机中维护，并且每次查询都要设置，LAST_INSERT_ID() 在服务器

中维护，而且从一个查询到另一个查询都保持一个值。

由 `mysql_insert_id()` 返回的值是连接专用的，不受其他连接中 `AUTO_INCREMENT` 活动的影响。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "INSERT INTO member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2002-6-3')";
$result = mysql_query ($query)
    or die ("Query failed");
printf ("membership number for new member: %d\n", mysql_insert_id());
?>
```

■ int

mysql_num_fields (int result_id);

返回给定结果集中的列数。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
printf ("Number of columns: %d\n", mysql_num_fields ($result));
?>
```

■ int

mysql_num_rows (int result_id);

返回给定结果集中的行数。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT * FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
printf ("Number of rows: %d\n", mysql_num_rows ($result));
?>
```

■ int

mysql_result (int result_id, int row, mixed field);

从给定结果集的行中返回一个值。列由参数 `field` 标识，可能是数字列索引，或者是在查询中指定的列名称索引。

这个函数速度慢，使用 `mysql_fetch_array()`、`mysql_fetch_object()` 或者 `mysql_fetch_row()` 会更合适一些。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
```

```
mysql_select_db ("samp_db")
    or die ("Could not select database");
$query = "SELECT last_name, first_name FROM president";
$result = mysql_query ($query)
    or die ("Query failed");
for ($i = 0; $i < mysql_num_rows ($result); $i++)
{
    for ($j = 0; $j < mysql_num_fields ($result); $j++)
    {
        if ($j > 0)
            print (" ");
        print (mysql_result ($result, $i, $j));
    }
    print "<BR>\n";
}
mysql_free_result ($result);
?>
```

■ string

mysql_tablename (int result_id, int row_num);

给定一个由 mysql_list_dbs() 或者 mysql_list_tables() 返回的结果集标识符和一个行索引 row_num, 返回在结果集的给定列中存储的名称。

行索引的范围为0到 mysql_num_rows()-1。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
$result = mysql_list_tables ("samp_db")
    or die ("Query failed");
print ("samp_db tables:<BR>\n");
for ($i = 0; $i < mysql_num_rows ($result); $i++)
    printf ("%s<BR>\n", mysql_tablename ($result, $i));
?>
```

H.2.5 数据库例程

本节的例程允许创建和删除数据库。它们也允许为当前的服务器会话选择一个缺省的数据库。

mysql_create_db (string db_name [, int link_id]);

■ int

告诉由 link_id 标识的 MySQL 服务器用给定的名称来创建数据库。如果数据库创建成功, 则返回真; 如果出现错误, 则返回假。必须在数据库有创建它的 CREATE 权限。

可能利用 mysql_query() 较利用 mysql_create_db() 发布 CREATE DATABASE 语句更为适合。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
if (mysql_create_db ("my_db"))
    print ("Database created successfully\n");
else
    printf ("Error creating database: %s\n", mysql_error ());
?>
```

■ int

```
mysql_drop_db (string db_name [, int link_id]);
```

告诉由 link_id 标识的 MySQL 服务器用给定的名称来删除数据库。如果数据库删除成功，则返回真；如果出现错误，则返回假。必须有对数据库进行删除的 DROP 权限。

要小心这个函数；如果删除数据库，它就不存在了，且不能恢复。

使用 mysql_query() 较使用 mysql_drop_db() 发布 DROP DATABASE 语句更为适合。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
if (mysql_drop_db ("my_db"))
    print ("Database dropped successfully\n");
else
    printf ("Error dropping database: %s\n", mysql_error ());
?>
```

■ int

```
mysql_select_db (string db_name [, int link_id]);
```

选择给定的数据库使其对给定的连接为缺省数据库。成功时返回真，如果出现错误，则返回假。

如果没有指定连接，则使用当前的连接。如果当前没有连接，则 mysql_select_db() 试图打开一个连接，就像无参数地调用 mysql_connect() 一样。如果这种连接企图失败，则 mysql_select_db() 失败。

```
<?php
$link = mysql_pconnect ("pit-viper.snake.net", "paul", "secret")
    or die ("Could not connect");
print ("Connected successfully");
mysql_select_db ("samp_db")
    or die ("Could not select database");
?>
```