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中的目录和文件。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、函数名或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起动时需要检查的表。

- 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的东西，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。如果希望能够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))
```

这里的问题是需要在相同的 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：

SHOW DATABASES	列出服务器上的数据库
SHOW TABLES	列出当前数据库中的表
SHOW TABLES FROM db_*name*	列出指定数据库中的表
SHOW COLUMNS FROM tbl_*name*	显示指定表中列的信息
SHOW INDEX FROM tbl_*name*	显示指定表中索引的信息
SHOW TABLE STATUS	显示缺省数据库中表的说明信息
SHOW TABLE STATUS FROM *db_name*	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：

% mysqlshow	列出服务器上的数据库
% mysqlshow *db_name*	列出指定数据库中的表
% mysqlshow *db_name tbl_name*	显示指定表中列的信息
% mysqlshow -- keys *db_name tbl_name*	显示指定表中索引的信息
% mysqlshow --status *db_name*	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 COLUMS 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 *selection_list*	选择哪些列
FROM *table_list*	从何处选择行
WHERE *primary_constraint*	行必须满足什么条件
GROUP BY *grouping_columns*	怎样对结果分组
ORDER BY *sorting_columns*	怎样对结果排序
HAVING *secondary_constraint*	行必须满足的第二条件
LIMIT *count*	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

```
+------+------+
| i1   | c1   |
+------+------+
|    1 | a    |
```

表 t2

```
+------+------+
| i2   | c2   |
+------+------+
|    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 100×200×300=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同连接（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。可用两个方式编写注释。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一个空格（“ -- ”）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，但需要加一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一个注释，但不可能写 5-- 7 这样的表达式，因此，这是一个很有用的探索。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 中一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。实事上，即使 MySQL 具有了子查询，检查用子选择编写的查询也是一件苦差事；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

下面是一个包含一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")
```

可通过将其转换为一个简单的连接，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

另一种常用的子选择查询是查找一个表中有的而另一个表中没有的值。正如以前所看到的那样，“ 那些未给出的值 ”这一类的问题是 LEFT JOIN 可能有用的一个线索。下面的查询包含一个子选择（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

```
+------+------+
| i1   | c1   |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
```

表 t2

```
+------+------+
| i2   | c2   |
+------+------+
|    2 | c    |
|    3 | b    |
|    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一个 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，增加表中相应行的计数值，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。（即使这样做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的学分。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”。