全文本搜索

本章将学习如何使用MySQL的全文本搜索功能进行高级的数据查询和选择。

> 理解全文本搜索

并非所有引擎都支持全文本搜索 MySQL支持几种基本的数据库引擎。并非所有的引擎都支持全文本搜索。两个最常使用的引擎为 MyISAM 和 InnoDB ,前者支持全文本搜索,而后者不支持。这就是为什么虽然本书中创建的多数样例表使用 InnoDB ,而有一个样例表(productnotes 表)却使用 MyISAM 的原因。如果你的应用中需要全文本搜索功能,应该记住这一点。

前面介绍了 LIKE 关键字,它利用通配操作符匹配文本(和部分文本)。使用 LIKE ,能够查找包含特殊值或部分值的行(不管这些值位于列内什么位置)。

还学习了,用基于文本的搜索作为正则表达式匹配列值的更进一步的介绍。使用正则表达式,可以编写查找所需行的非常复杂的匹配模式。

虽然这些搜索机制非常有用,但存在几个重要的限制。

- 性能——通配符和正则表达式匹配通常要求MySQL尝试匹配表中所有行(而且这些搜索极少使用表索引)。因此,由于被搜索行数不断增加,这些搜索可能非常耗时。
- 明确控制——使用通配符和正则表达式匹配,很难(而且并不总是能)明确地控制匹配什么和不匹配什么。例如,指定一个词必须匹配,一个词必须不匹配,而一个词仅在第一个词确实匹配的情况下才可以匹配或者才可以不匹配。
- 智能化的结果——虽然基于通配符和正则表达式的搜索提供了非常灵活的搜索,但它们都不能提供一种智能化的选择结果的方法。例如,一个特殊词的搜索将会返回包含该词的所有行,而不区分包含单个匹配的行和包含多个匹配的行(按照可能是更好的匹配来排列它们)。类似,一个特殊词的搜索将不会找出不包含该词但包含其他相关词的行。

所有这些限制以及更多的限制都可以用全文本搜索来解决。在使用全文本搜索时,MySQL不需要分别查看每个行,不需要分别分析和处理每个词。MySQL创建指定列中各词的一个索引,搜索可以针对这些词进行。这样,MySQL可以快速有效地决定哪些词匹配(哪些行包含它们),哪些词不匹配,它们匹配的频率,等等。

> 使用全文本搜索

为了进行全文本搜索,必须索引被搜索的列,而且要随着数据的改变不断地重新索引。在对表列进行适当设计后,MySQL会自动进行所有的索引和重新索引。

在索引之后, SELECT 可与 Match() 和 Against() 一起使用以实际执行搜索。

1.启用全文本搜索支持

一般在创建表时启用全文本搜索。 `CREATE TABLE` 语句接受 `FULLTEXT` 子句,它给出被索引列的一个逗号分隔的列表。

> 下面的 CREATE 语句演示了 FULLTEXT 子句的使用:

```
MariaDB [test]> create table productnotes1 (note_id int not null auto_increment,prod_id char(10)
not null,note_date datetime not null,note_text text null,primary key(note_id),fulltext(note_text))
engine=myisam;
Query OK, 0 rows affected (0.37 sec)

MariaDB [test]> show table status where name='productnotes1'\G;
*************************** 1. row ***************************
           Name: productnotes1
         Engine: MyISAM
        Version: 10
     Row_format: Dynamic
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 281474976710655
   Index_length: 1024
      Data_free: 0
 Auto_increment: 1
    Create_time: 2016-09-20 15:20:19
    Update_time: 2016-09-20 15:20:19
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

后面将详细考察 `CREATE TABLE` 语句。现在,只需知道这条`CREATE TABLE` 语句定义表 `productnotes1` 并列出它所包含的列即可。这些列中有一个名为 `note_text` 的列,为了进行全文本搜索,MySQL根据子句 `FULLTEXT(note_text)` 的指示对它进行索引。这里的`FULLTEXT` 索引单个列,如果需要也可以指定多个列。

在定义之后,MySQL自动维护该索引。在增加、更新或删除行时,索引随之自动更新。可以在创建表时指定 `FULLTEXT` ,或者在稍后指定(在这种情况下所有已有数据必须立即索引)。

不要在导入数据时使用 **FULLTEXT** 更新索引要花时间,虽然不是很多,但毕竟要花时间。如果正在导入数据到一个新表,此时不应该启用 `FULLTEXT` 索引。应该首先导入所有数据,然后再修改表,定义 `FULLTEXT` 。这样有助于更快地导入数据(而且使索引数据的总时间小于在导入每行时分别进行索引所需的总时间)

2.进行全文本搜索

在索引之后,使用两个函数 Match() 和 Against() 执行全文本搜索,其中 Match() 指定被搜索的列, Against() 指定要使用的搜索表达式。

下面举一个例子:

```
MariaDB [test]> select * from productnotes;
+---------+---------+---------------------+-----------------------------------------------------
-----------------------------------------------------------------------------------------------------
----+
| note_id | prod_id | note_date           | note_text

  |
+---------+---------+---------------------+-----------------------------------------------------
-----------------------------------------------------------------------------------------------------
----+
|     101 | TNT2    | 2005-08-17 00:00:00 | Customer complaint:
Sticks not individually wrapped, too easy to mistakenly detonate all at once.
Recommend individual wrapping.                          |
|     102 | OL1     | 2005-08-18 00:00:00 | Can shipped full, refills not available.
Need to order new can if refill needed.
               |
|     103 | SAFE    | 2005-08-18 00:00:00 | Safe is combination locked, combination not provided
with safe.
This is rarely a problem as safes are typically blown up or dropped by customers.         |
|     104 | FC      | 2005-08-19 00:00:00 | Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait.
                |
|     105 | TNT2    | 2005-08-20 00:00:00 | Included fuses are short and have been known to
detonate too quickly for some customers.
Longer fuses are available (item FU1) and should be recommended. |
|     106 | TNT2    | 2005-08-22 00:00:00 | Matches not included, recommend purchase of matches
or detonator (item DTNTR).
     |
|     107 | SAFE    | 2005-08-23 00:00:00 | Please note that no returns will be accepted if safe
opened using explosives.
    |
|     108 | ANV01   | 2005-08-25 00:00:00 | Multiple customer returns, anvils failing to drop
fast enough or falling backwards on purchaser. Recommend that customer considers using heavier
anvils.  |
|     109 | ANV03   | 2005-09-01 00:00:00 | Item is extremely heavy. Designed for dropping, not
recommended for use with slings, ropes, pulleys, or tightropes.
    |
|     110 | FC      | 2005-09-01 00:00:00 | Customer complaint: rabbit has been able to detect
trap, food apparently less effective now.
     |
|     111 | SLING   | 2005-09-02 00:00:00 | Shipped unassembled, requires common tools (including
oversized hammer).
   |
|     112 | SAFE    | 2005-09-02 00:00:00 | Customer complaint:
Circular hole in safe floor can apparently be easily cut with handsaw.
                                     |
|     113 | ANV01   | 2005-09-05 00:00:00 | Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping,
recommend ANV02 or ANV03 instead.   |
|     114 | SAFE    | 2005-09-07 00:00:00 | Call from individual trapped in safe plummeting to
the ground, suggests an escape hatch be added.
Comment forwarded to vendor.                            |

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

```
-----------------------------------------------------------------------------------------------
----+
14 rows in set (0.00 sec)

MariaDB [test]> select note_text from productnotes where note_text regexp 'rabbit';
+---------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+---------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                      |
+---------------------------------------------------------------------------------------------
----------------------+
2 rows in set (0.00 sec)

MariaDB [test]> select note_text from productnotes where match(note_text) against('rabbit');
+---------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+---------------------------------------------------------------------------------------------
----------------------+
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                      |
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
+---------------------------------------------------------------------------------------------
----------------------+
2 rows in set (0.00 sec)
```

此 SELECT 语句检索单个列 `note_text` 。由于 WHERE 子句,一个全文本搜索被执行。 `Match(note_text)` 指示 MySQL针对指定的列进行搜索, `Against('rabbit')` 指定词 `rabbit` 作为搜索文本。由于有两行包含词 `rabbit` , 这两个行被返回。

使用完整的**Match()**说明 传递给 Match() 的值必须与FULLTEXT() 定义中的相同。如果指定多个列,则必须列出它们(而且次序正确)。

搜索不区分大小写 除非使用 BINARY 方式(本章中没有介绍),否则全文本搜索不区分大小写。

事实是刚才的搜索可以简单地用 LIKE 子句完成,如下所示:

```
MariaDB [test]> select note_text from productnotes where note_text like '%rabbit%';
+----------------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+----------------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                      |
+----------------------------------------------------------------------------------------------------
----------------------+
2 rows in set (0.00 sec)
```

这条 SELECT 语句同样检索出两行,但次序不同(虽然并不总是出现这种情况)。

上述两条 SELECT 语句都不包含 ORDER BY 子句。后者(使用 LIKE)以不特别有用的顺序返回数据。前者(使用全文本搜索)返回以文本匹配的良好程度排序的数据。两个行都包含词 rabbit ,但包含词 rabbit 作为第3个词的行的等级比作为第20个词的行高。这很重要。全文本搜索的一个重要部分就是对结果排序。具有较高等级的行先返回(因为这些行很可能是你真正想要的行)。

为演示排序如何工作,请看以下例子:

```
MariaDB [test]> select note_text,match(note_text) against('rabbit') as rank from productnotes;
+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------+--------------------+
| note_text
                                                              | rank               |
+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------+--------------------+
| Customer complaint:
Sticks not individually wrapped, too easy to mistakenly detonate all at once.
Recommend individual wrapping.                               |                  0 |
| Can shipped full, refills not available.
Need to order new can if refill needed.
                |                  0 |
| Safe is combination locked, combination not provided with safe.
This is rarely a problem as safes are typically blown up or dropped by customers.          |
           0 |
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait.
                 | 1.5905543565750122 |
| Included fuses are short and have been known to detonate too quickly for some customers.
Longer fuses are available (item FU1) and should be recommended. |                  0 |
| Matches not included, recommend purchase of matches or detonator (item DTNTR).
                                                             |                  0 |
| Please note that no returns will be accepted if safe opened using explosives.
                                                             |                  0 |
| Multiple customer returns, anvils failing to drop fast enough or falling backwards on
purchaser. Recommend that customer considers using heavier anvils.  |                  0 |
| Item is extremely heavy. Designed for dropping, not recommended for use with slings, ropes,
pulleys, or tightropes.                                      |                  0 |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                                                             | 1.6408053636550903 |
| Shipped unassembled, requires common tools (including oversized hammer).
                                                             |                  0 |
| Customer complaint:
Circular hole in safe floor can apparently be easily cut with handsaw.
                                      |                  0 |
| Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping,
recommend ANV02 or ANV03 instead.   |                  0 |
| Call from individual trapped in safe plummeting to the ground, suggests an escape hatch be
added.
Comment forwarded to vendor.                                 |                  0 |
+-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------+--------------------+
14 rows in set (0.00 sec)
```

这里,在 `SELECT` 而不是 `WHERE` 子句中使用 `Match()` 和 `Against()` 。这使所有行都被返回(因为没有 WHERE 子句)。 `Match()` 和 `Against()` 用来建立一个计算列(别名为 `rank`),此列包含全文本搜索计算出的等级值。等级由MySQL根据行中词的数目、唯一词的数目、整个索引中词的总数以及包含该词的行的数目计算出来。正如所见,不包含词 `rabbit` 的行等级为0(因此不被前一例子中的 `WHERE` 子句选择)。确实包含词 `rabbit` 的两个行每行都有一个等级值,文本中词靠前的行的等级值比词靠后的行的等级值高。

这个例子有助于说明全文本搜索如何排除行(排除那些等级为0的行),如何排序结果(按等级以降序排序)。

排序多个搜索项 如果指定多个搜索项,则包含多数匹配词的那些行将具有比包含较少词(或仅有一个匹配)的那些行高的等级值。

正如所见,全文本搜索提供了简单 LIKE 搜索不能提供的功能。而且,由于数据是索引的,全文本搜索还相当快。

3.使用查询扩展

查询扩展用来设法放宽所返回的全文本搜索结果的范围。考虑下面的情况。你想找出所有提到 anvils 的注释。只有一个注释包含词 anvils ,但你还想找出可能与你的搜索有关的所有其他行,即使它们不包含词anvils 。

这也是查询扩展的一项任务。在使用查询扩展时,MySQL对数据和索引进行两遍扫描来完成搜索:

- 首先,进行一个基本的全文本搜索,找出与搜索条件匹配的所有行;
- 其次,MySQL检查这些匹配行并选择所有有用的词(我们将会简要地解释MySQL如何断定什么有用,什么无用)。
- 再其次, MySQL再次进行全文本搜索,这次不仅使用原来的条件,而且还使用所有有用的词。

利用查询扩展,能找出可能相关的结果,即使它们并不精确包含所查找的词。

只用于**MySQL**版本**4.1.1**或更高级的版本 查询扩展功能是在MySQL 4.1.1中引入的,因此不能用于之前的版本。

下面举一个例子,首先进行一个简单的全文本搜索,没有查询扩展:

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('anvils');
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
| note_text
                                                           |
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
| Multiple customer returns, anvils failing to drop fast enough or falling backwards on
purchaser. Recommend that customer considers using heavier anvils. |
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
1 row in set (0.00 sec)
```

只有一行包含词 anvils ,因此只返回一行。

下面是相同的搜索,这次使用查询扩展:

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('anvils' with
query expansion);
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| note_text
                                                       |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| Multiple customer returns, anvils failing to drop fast enough or falling backwards on
purchaser. Recommend that customer considers using heavier anvils. |
| Customer complaint:
Sticks not individually wrapped, too easy to mistakenly detonate all at once.
Recommend individual wrapping.                         |
| Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping,
recommend ANV02 or ANV03 instead.  |
| Please note that no returns will be accepted if safe opened using explosives.
                                                       |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                                                        |
| Customer complaint:
Circular hole in safe floor can apparently be easily cut with handsaw.
                                     |
| Matches not included, recommend purchase of matches or detonator (item DTNTR).
                                                        |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
7 rows in set (0.00 sec)
```

这次返回了7行。第一行包含词 `anvils` ,因此等级最高。第二行与 `anvils` 无关,但因为它包含第一行中的两个词(`customer` 和 `recommend`),所以也被检索出来。第3行也包含这两个相同的词,但它们在文本中的位置更靠后且分开得更远,因此也包含这一行,但等级为第三。第三行确实也没有涉及 `anvils` (按它们的产品名)。

正如所见,查询扩展极大地增加了返回的行数,但这样做也增加了你实际上并不想要的行的数目。

行越多越好 表中的行越多(这些行中的文本就越多),使用查询扩展返回的结果越好。

4.布尔文本搜索

MySQL支持全文本搜索的另外一种形式,称为布尔方式(boolean mode)。

以布尔方式,可以提供关于如下内容的细节:

- 要匹配的词;
- 要排斥的词(如果某行包含这个词,则不返回该行,即使它包含其他指定的词也是如此);
- 排列提示(指定某些词比其他词更重要,更重要的词等级更高);
- 表达式分组;
- 另外一些内容。

即使没有 **FULLTEXT** 索引也可以使用 布尔方式不同于迄今为止使用的全文本搜索语法的地方在于,即使没有定义 FULLTEXT 索引,也可以使用它。但这是一种非常缓慢的操作(其性能将随着数据量的增加而降低)。

为演示 IN BOOLEAN MODE 的作用,举一个简单的例子:

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('heavy' in
boolean mode);
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| note_text
                                                       |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| Item is extremely heavy. Designed for dropping, not recommended for use with slings, ropes,
pulleys, or tightropes.                                |
| Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping,
recommend ANV02 or ANV03 instead. |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
2 rows in set (0.00 sec)
```

此全文本搜索检索包含词 heavy 的所有行(有两行)。其中使用了关键字 IN BOOLEAN MODE ,但实际上没有指定布尔操作符,因此,其结果与没有指定布尔方式的结果相同。

IN BOOLEAN MODE 的行为差异 虽然这个例子的结果与没有IN BOOLEAN MODE 的相同,但其行为有一个重要的差别(即使在这个特殊的例子没有表现出来)。

> 为了匹配包含 heavy 但不包含任意以 rope 开始的词的行,可使用以下查询:

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('heavy -rope*'
in boolean mode);
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| note_text
                                                       |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
| Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping,
recommend ANV02 or ANV03 instead. |
+-------------------------------------------------------------------------------------------------
-------------------------------------------------------+
1 row in set (0.00 sec)
```

这次只返回一行。这一次仍然匹配词 `heavy` ,但 `-rope*` 明确地指示 MySQL排除 包含 `rope*` (任何以 `rope` 开始 的词,包 括`ropes`)的行,这就是为什么上一个例子中的第一行被排除的原因。

在**MySQL 4.x**中所需的代码更改 如果你使用的是MySQL4.x,则上面的例子可能不返回任何行。这是 * 操作符处理中的一个错误。为在MySQL 4.x中使用这个例子,使用 -ropes 而不是 -rope* (排除 ropes 而不是排除任何以 rope 开始的词)。

全文本布尔操作符	说明
+	包含,词必须存在
-	排除,词必须不出现
>	包含,而且增加等级值
<	包含,且减少等级值
()	把词组成子表达式(允许这些子表达式作为一个组被包含、排除、排列等)
~	取消一个词的排序值
*	词尾的通配符
""	定义一个短语(与单个词的列表不一样,它匹配整个短语以便包含或排除这个短语)

下面举几个例子,说明某些操作符如何使用:

> 搜索匹配包含词 rabbit 和 bait 的行。

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('+rabbit +bait'
in boolean mode);
+-----------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+-----------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
+-----------------------------------------------------------------------------------------------
----------------------+
1 row in set (0.00 sec)
```

> 搜索匹配包含 rabbit 和 bait 中的至少一个词的行。

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('rabbit bait'
in boolean mode);
+-----------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+-----------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                      |
+-----------------------------------------------------------------------------------------------
----------------------+
2 rows in set (0.00 sec)
```

> 搜索匹配短语 rabbit bait 而不是匹配两个词 rabbit 和 bait 。

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('"rabbit bait"'
in boolean mode);
+-----------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+-----------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
+-----------------------------------------------------------------------------------------------
----------------------+
1 row in set (0.00 sec)
```

> 搜索匹配 rabbit 和 carrot ,增加前者的等级,降低后者的等级。

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('>rabbit
<carrot' in boolean mode);
+-----------------------------------------------------------------------------------------------
----------------------+
| note_text
                     |
+-----------------------------------------------------------------------------------------------
----------------------+
| Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit bait. |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.
                      |
+-----------------------------------------------------------------------------------------------
----------------------+
2 rows in set (0.00 sec)
```

> 搜索匹配词 safe 和 combination ,降低后者的等级。

```
MariaDB [test]> select note_text from productnotes where match(note_text) against('+safe +
(<combination)' in boolean mode);
+-----------------------------------------------------------------------------------------------
-------------------------------------------------+
| note_text
                                                 |
+-----------------------------------------------------------------------------------------------
-------------------------------------------------+
| Safe is combination locked, combination not provided with safe.
This is rarely a problem as safes are typically blown up or dropped by customers. |
+-----------------------------------------------------------------------------------------------
-------------------------------------------------+
1 row in set (0.00 sec)
```

排列而不排序 在布尔方式中,不按等级值降序排序返回的行。

5.全文本搜索的使用说明

在结束本章之前,给出关于全文本搜索的某些重要的说明。

- 在索引全文本数据时,短词被忽略且从索引中排除。短词定义为那些具有3个或3个以下字符的词(如果需要,这个数目可以更改)。
- MySQL带有一个内建的非用词(stopword)列表,这些词在索引全文本数据时总是被忽略。如果需要,可以覆盖这个列表(请参阅MySQL文档以了解如何完成此工作)。
- 许多词出现的频率很高,搜索它们没有用处(返回太多的结果)。因此,MySQL规定了一条50%规则,如果一个词出现在50%以上的行中,则将它作为一个非用词忽略。 50%规则不用于 IN BOOLEANMODE 。
- 如果表中的行数少于3行,则全文本搜索不返回结果(因为每个词或者不出现,或者至少出现在50%的行中)。
- 忽略词中的单引号。例如, don't 索引为 dont 。
- 不具有词分隔符(包括日语和汉语)的语言不能恰当地返回全文本搜索结果。
- 如前所述,仅在 MyISAM 数据库引擎中支持全文本搜索。

没有邻近操作符 邻近搜索是许多全文本搜索支持的一个特性,它能搜索相邻的词(在相同的句子中、相同的段落中或者在特定数目的词的部分中,等等)。MySQL全文本搜索现在还不支持邻近操作符,不过未来的版本有支持这种操作符的计划。

> 小结
>
> > 本章介绍了为什么要使用全文本搜索,以及如何使用MySQL的Match() 和 Against() 函数进行全文本搜索。我们还学习了查询扩展(它能增加找到相关匹配的机会)和如何使用布尔方式进行更细致的查找控制。

DDL语言

> DDL 是数据定义语言的缩写,简单来说,就是对数据库内部的对象进行创建、删除、修改的操作语言。它和 DML 语言的最大区别是 DML 只是对表内部数据的操作,而不涉及到表的定义、结构的修改,更不会涉及到其他对象。 DDL 语句更多的被数据库管理员(DBA)所使用,一般的开发人员很少使用。

创建和操纵表

本章讲授表的创建、更改和删除的基本知识。

> 创建表

MySQL不仅用于表数据操纵,而且还可以用来执行数据库和表的所有操作,包括表本身的创建和处理。

一般有两种创建表的方法:

- 使用具有交互式创建和管理表的工具(如第2章讨论的工具);
- 表也可以直接用MySQL语句操纵。

为了用程序创建表,可使用SQL的 `CREATE TABLE` 语句。值得注意的是,在使用交互式工具时,实际上使用的是MySQL语句。但是,这些语句不是用户编写的,界面工具会自动生成并执行相应的MySQL语句(更改现有表时也是这样)。

1.表创建基础

为利用 CREATE TABLE 创建表,必须给出下列信息:

- 新表的名字,在关键字 CREATE TABLE 之后给出;
- 表列的名字和定义,用逗号分隔。

CREATE TABLE 语句也可能会包括其他关键字或选项,但至少要包括表的名字和列的细节。下面的MySQL语句创建我们所用的 customers 表:

```
MariaDB [test]> create table customers
    -> (
    -> cust_id int not null auto_increment,
    -> cust_name char(50) not null,
    -> cust_address char(50) null,
    -> cust_city char(50) null,
    -> cust_state char(5) null,
    -> cust_zip char(10) null,
    -> cust_country char(50) null,
    -> cust_contact char(50) null,
    -> cust_email char(255) null,
    -> primary key (cust_id)
    -> ) engine=innodb;
```

从上面的例子中可以看到,表名紧跟在 `CREATE TABLE` 关键字后面。实际的表定义(所有列)括在圆括号之中。各列之间用逗号分隔。这个表由9列组成。每列的定义以列名(它在表中必须是唯一的)开始,后跟列的数据类型(关于数据类型的解释,请参阅第1章。此外,附录D列出了MySQL支持的数据类型)。表的主键可以在创建表时用 `PRIMARY KEY` 关键字指定。这里,列 `cust_id` 指定作为主键列。整条语句由右圆括号后的分号结束。(现在先忽略 `ENGINE=InnoDB` 和 `AUTO_INCREMENT` ,后面会对它们进行介绍。)

语句格式化 可回忆一下,以前说过MySQL语句中忽略空格。语句可以在一个长行上输入,也可以分成许多行。它们的作用都相同。这允许你以最适合自己的方式安排语句的格式。前面的 CREATE TABLE 语句就是语句格式化的一个很好的例子,它被安排在多个行上,其中的列定义进行了恰当的缩进,以便阅读和编辑。以何种缩进格式安排SQL语句没有规定,但我强烈推荐采用某种缩进格式。

处理现有的表 在创建新表时,指定的表名必须不存在,否则将出错。如果要防止意外覆盖已有的表,SQL要求首先手工删除该表,然后再重建它,而不是简单地用创建表语句覆盖它。如果你仅想在一个表不存在时创建它,应该在表名后给出 `IF NOT EXISTS` 。这样做不检查已有表的模式是否与你打算创建的表模式相匹配。它只是查看表名是否存在,并且仅在表名不存在时创建它。

```
DROP TABLE IF EXISTS columns_priv;
CREATE TABLE columns_priv (
  Host char(60) COLLATE utf8_bin NOT NULL DEFAULT '',
  Db char(64) COLLATE utf8_bin NOT NULL DEFAULT '',
  User char(16) COLLATE utf8_bin NOT NULL DEFAULT '',
  Table_name char(64) COLLATE utf8_bin NOT NULL DEFAULT '',
  Column_name char(64) COLLATE utf8_bin NOT NULL DEFAULT '',
  Timestamp timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  Column_priv set('Select','Insert','Update','References') CHARACTER SET utf8 NOT NULL DEFAULT
'',
  PRIMARY KEY (Host,Db,User,Table_name,Column_name)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_bin COMMENT='Column privileges';
```

2.使用 NULL 值

前面章节中说过, NULL 值就是没有值或缺值。允许 NULL 值的列也允许在插入行时不给出该列的值。不允许 NULL 值的列不接受该列没有值的行,换句话说,在插入或更新行时,该列必须有值。

每个表列或者是 NULL 列,或者是 NOT NULL 列,这种状态在创建时由表的定义规定。请看下面的例子:

> 创建 orders 表

```
MariaDB [test]> create table orders
    -> (
    -> order_num int not null auto_increment,
    -> order_date datetime not null,
    -> cust_id int not null,
    -> primary key (order_num)
    -> ) engine=innodb;
```

`orders` 包含3个列,分别是订单号、订单日期和客户ID。所有3个列都需要,因此每个列的定义都含有关键字 `NOT NULL` 。这将会阻止插入没有值的列。如果试图插入没有值的列,将返回错误,且插入失败。

> 创建 vendors 表

```
CREATE TABLE vendors (
  vend_id int NOT NULL AUTO_INCREMENT,
  vend_name char(50) NOT NULL,
  vend_address char(50) NULL,
  vend_city char(50) NULL,
  vend_state char(5) NULL,
  vend_zip char(10) NULL,
  vend_country char(50) NULL,
  PRIMARY KEY (vend_id)
) ENGINE=InnoDB;
```

供应商ID和供应商名字列是必需的,因此指定为 `NOT NULL` 。其余5个列全都允许 `NULL` 值,所以不指定 `NOT NULL` 。 `NULL` 为默认设置,如果不指定 `NOT NULL` ,则认为指定的是 `NULL` 。

理解 **NULL** 不要把 NULL 值与空串相混淆。 NULL 值是没有值,它不是空串。如果指定 '' (两个单引号,其间没有字符),这在 NOT NULL 列中是允许的。空串是一个有效的值,它不是无值。 NULL 值用关键字 NULL 而不是空串指定。

3.主键再介绍

正如所述,主键值必须唯一。即,表中的每个行必须具有唯一的主键值。如果主键使用单个列,则它的值必须唯一。如果使用多个列,则这些列的组合值必须唯一。

迄今为止我们看到的 CREATE TABLE 例子都是用单个列作为主键。其中主键用以下的类似的语句定义: `primary key (vend_id)`

为创建由多个列组成的主键,应该以逗号分隔的列表给出各列名,如下所示:

```
CREATE TABLE orderitems (
  order_num int NOT NULL,
  order_item int NOT NULL,
  prod_id char(10) NOT NULL,
  quantity int NOT NULL,
  item_price decimal(8,2) NOT NULL,
  PRIMARY KEY (order_num,order_item),
  ) ENGINE=InnoDB;
```

`orderitems` 表包含orders表中每个订单的细节。每个订单有多项物品,但每个订单任何时候都只有1个第一项物品,1个第二项物品,如此等等。因此,订单号(`order_num` 列)和订单物品(`order_item` 列)的组合是唯一的,从而适合作为主键,其定义为: `RIMARY KEY (order_num,order_item)`

主键可以在创建表时定义(如这里所示),或者在创建表之后定义(本章稍后讨论)。

4.使用 AUTO_INCREMENT

让我们再次考察 `customers` 和 `orders` 表。 `customers` 表中的顾客由列 `cust_id` 唯一标识,每个顾客有一个唯一编号。类似, `orders` 表中的每个订单有一个唯一的订单号,这个订单号存储在列 `order_num` 中。这些编号除它们是唯一的以外没有别的特殊意义。在增加一个新顾客或新订单时,需要一个新的顾客ID或订单号。这些编号可以任意,只要它们是唯一的即可。

显然,使用的最简单的编号是下一个编号,所谓下一个编号是大于当前最大编号的编号。例如,如果 `cust_id` 的最大编号为 `10005` ,则插入表中的下一个顾客可以具有等于 `10006` 的 `cust_id` 。

简单吗?不见得。你怎样确定下一个要使用的值?当然,你可以使用 `SELECT` 语句得出最大的数,然后对它加1。但这样做并不可靠(你需要找出一种办法来保证,在你执行 `SELECT` 和 `INSERT` 两条语句之间没有其他人插入行,对于多用户应用,这种情况是很有可能出现的),而且效率也不高(执行额外的MySQL操作肯定不是理想的办法)。

这就是 `AUTO_INCREMENT` 发挥作用的时候了。请看以下代码行(用来创建 customers 表的 CREATE TABLE 语句的组成部分): `cust_id int NOT NULL AUTO_INCREMENT`

`AUTO_INCREMENT` 告诉MySQL,本列每当增加一行时自动增量。每次执行一个 `INSERT` 操作时,MySQL自动对该列增量(从而才有这个关键字 `AUTO_INCREMENT`),给该列赋予下一个可用的值。这样给每个行分配一个唯一的 `cust_id` ,从而可以用作主键值。

每个表只允许一个 `AUTO_INCREMENT` 列,而且它必须被索引(如,通过使它成为主键)

覆 盖 **AUTO_INCREMENT** 如果一个列被指定为 AUTO_INCREMENT ,则它需要使用特殊的值吗?你可以简单地在 INSERT 语句中指定一个值,只要它是唯一的(至今尚未使用过)即可,该值将被用来替代自动生成的值。后续的增量将开始使用该手工插入的值。

确定 **AUTO_INCREMENT** 值 让MySQL生成(通过自动增量)主键的一个缺点是你不知道这些值都是谁。

考虑这个场景:你正在增加一个新订单。这要求在 `orders` 表中创建一行,然后在 `orderitms` 表中对订购的每项物品创建一行。 `order_num` 在 `orderitem` s 表中与订单细节一起存储。这就是为什么 `orders` 表和 `orderitems` 表为相互关联的表的原因。这显然要求你在插入 `orders` 行之后,插入 `orderitems` 行之前知道生成的 `order_num` 。那么,如何在使用 `AUTO_INCREMENT` 列时获得这个值呢?可使用 `last_insert_id()` 函数获得这个值,如下所示: `select last_insert_id()` 此语句返回最后一个 `AUTO_INCREMENT` 值,然后可以将它用于后续的MySQL语句。

5.指定默认值

如果在插入行时没有给出值,MySQL允许指定此时使用的默认值。默认值用 CREATE TABLE 语句的列定义中的 DEFAULT 关键字指定。

请看下面的例子:

```
CREATE TABLE orderitems (
  order_num int NOT NULL,
  order_item int NOT NULL,
  prod_id char(10) NOT NULL,
  quantity int NOT NULL default 1,
  item_price decimal(8,2) NOT NULL,
  PRIMARY KEY (order_num,order_item),
  ) ENGINE=InnoDB;
```

这条语句创建包含组成订单的各物品的 `orderitems` 表(订单本身存储在 orders 表中)。 `quantity` 列包含订单中每项物品的数量。在此例子中,给该列的描述添加文本 `DEFAULT 1` 指示MySQL,在未给出数量的情况下使用数量 1 。

不允许函数 与大多数DBMS不一样,MySQL不允许使用函数作为默认值,它只支持常量。

使用默认值而不是 **NULL** 值 许多数据库开发人员使用默认值而不是 NULL 列,特别是对用于计算或数据分组的列更是如此。

6.引擎类型

你可能已经注意到,迄今为止使用的 CREATE TABLE 语句全都以ENGINE=InnoDB 语句结束。

与其他DBMS一样, MySQL有一个具体管理和处理数据的内部引擎。

在你使用 CREATE TABLE 语句时,该引擎具体创建表,而在你使用 SELECT语句或进行其他数据库处理时,该引擎在内部处理你的请求。多数时候,此引擎都隐藏在DBMS内,不需要过多关注它。

但MySQL与其他DBMS不一样,它具有多种引擎。它打包多个引擎,这些引擎都隐藏在MySQL服务器内,全都能执行 CREATE TABLE 和 SELECT等命令。

为什么要发行多种引擎呢?

因为它们具有各自不同的功能和特性,为不同的任务选择正确的引擎能获得良好的功能和灵活性。

当然,你完全可以忽略这些数据库引擎。如果省略 ENGINE= 语句,则,多数SQL语句都会默认使用它。但并使用默认引擎(很可能是 MyISAM)不是所有语句都默认使用它,这就是为什么 ENGINE= 语句很重要的原因(也就是为什么本书的样列表中使用两种引擎的原因)。

以下是几个需要知道的引擎:

- InnoDB 是一个可靠的事务处理引擎,它不支持全文本搜索;
- MEMORY 在功能等同于 MyISAM ,但由于数据存储在内存(不是磁盘)中,速度很快(特别适合于临时表);
- MyISAM 是一个性能极高的引擎,它支持全文本搜索,但不支持事务处理。

引擎类型可以混用。除 productnotes 表使用 MyISAM 外,本书中的样例表都使用 InnoDB 。原因是希望支持事务处理(因此,使用 InnoDB),但也需要在 productnotes 中支持全文本搜索(因此,使用 MyISAM)。

外键不能跨引擎 混用引擎类型有一个大缺陷。外键(用于强制实施引用完整性,如第1章所述)不能跨引擎,即使用一个引擎的表不能引用具有使用不同引擎的表的外键。

那么,你应该使用哪个引擎?这有赖于你需要什么样的特性。 MyISAM由于其性能和特性可能是最受欢迎的引擎。但如果你不需要可靠的事务处理,可以使用其他引擎。

> 更新表

为更新表定义,可使用 `ALTER TABLE` 语句。但是,理想状态下,当表中存储数据以后,该表就不应该再被更新。在表的设计过程中需要花费大量时间来考虑,以便后期不对该表进行大的改动。

为了使用 ALTER TABLE 更改表结构,必须给出下面的信息:

- 在 ALTER TABLE 之后给出要更改的表名(该表必须存在,否则将出错);
- 所做更改的列表。

下面的例子给表添加一个列:

> 给 vendors 表增加一个名为 vend_phone 的列

```
MariaDB [test]> alter table vendors add vend_phone char(20);
Query OK, 6 rows affected (0.11 sec)
Records: 6  Duplicates: 0  Warnings: 0

MariaDB [test]> desc vendors;
+--------------+----------+------+-----+---------+----------------+
| Field        | Type     | Null | Key | Default | Extra          |
+--------------+----------+------+-----+---------+----------------+
| vend_id      | int(11)  | NO   | PRI | NULL    | auto_increment |
| vend_name    | char(50) | NO   |     | NULL    |                |
| vend_address | char(50) | YES  |     | NULL    |                |
| vend_city    | char(50) | YES  |     | NULL    |                |
| vend_state   | char(5)  | YES  |     | NULL    |                |
| vend_zip     | char(10) | YES  |     | NULL    |                |
| vend_country | char(50) | YES  |     | NULL    |                |
| vend_phone   | char(20) | YES  |     | NULL    |                |
+--------------+----------+------+-----+---------+----------------+
8 rows in set (0.00 sec)
```

> 删除刚刚添加的列vend_phone

```
MariaDB [test]> alter table vendors drop column vend_phone;
Query OK, 6 rows affected (0.12 sec)
Records: 6  Duplicates: 0  Warnings: 0

MariaDB [test]> desc vendors;
+--------------+----------+------+-----+---------+----------------+
| Field        | Type     | Null | Key | Default | Extra          |
+--------------+----------+------+-----+---------+----------------+
| vend_id      | int(11)  | NO   | PRI | NULL    | auto_increment |
| vend_name    | char(50) | NO   |     | NULL    |                |
| vend_address | char(50) | YES  |     | NULL    |                |
| vend_city    | char(50) | YES  |     | NULL    |                |
| vend_state   | char(5)  | YES  |     | NULL    |                |
| vend_zip     | char(10) | YES  |     | NULL    |                |
| vend_country | char(50) | YES  |     | NULL    |                |
+--------------+----------+------+-----+---------+----------------+
7 rows in set (0.01 sec)
```

> ALTER TABLE 的一种常见用途是定义外键。下面是用来定义讲义中的表所用的外键的代码:

```
MariaDB [test]> alter table orderitems
  -> add constraint fk_orderitems_orders
  -> foreign key (order_num) references orders (order_num);

MariaDB [test]> alter table orderitems
  -> add constraint fk_orderitems_products
  -> foreign key (prod_id) references orders (prod_id);

MariaDB [test]> alter table orders
  -> add constraint fk_orderitems_customers
  -> foreign key (cust_id) references orders (cust_id);

MariaDB [test]> alter table products
  -> add constraint fk_orderitems_vendors
  -> foreign key (vend_id) references orders (vend_id);
```

这里,由于要更改4个不同的表,使用了4条 ALTER TABLE 语句。为了对单个表进行多个更改,可以使用单条 ALTER TABLE 语句,每个更改用逗号分隔。

复杂的表结构更改一般需要手动删除过程,它涉及以下步骤:

- 用新的列布局创建一个新表;
- 使用 INSERT SELECT 语句从旧表复制数据到新表。如果有必要,可使用转换函数和计算字段;
- 检验包含所需数据的新表;
- 重命名旧表(如果确定,可以删除它);
- 用旧表原来的名字重命名新表;
- 根据需要,重新创建触发器、存储过程、索引和外键。

小心使用 **ALTER TABLE** 使用 ALTER TABLE 要极为小心,应该在进行改动前做一个完整的备份(模式和数据的备份)。数据库表的更改不能撤销,如果增加了不需要的列,可能不能删除它们。类似地,如果删除了不应该删除的列,可能会丢失该列中的所有数据。

> 删除表

删除表(删除整个表而不是其内容)非常简单,使用 DROP TABLE 语句即可: `drop table customers2;`

```
ariaDB [test]> create table customer2 as select * from customers;
Query OK, 15 rows affected (0.08 sec)
Records: 15  Duplicates: 0  Warnings: 0

MariaDB [test]> select * from customers;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | The Fudds      | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
|   20001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   20002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   20003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   20004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   20005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   20006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
15 rows in set (0.00 sec)
```

这条语句删除 customers2 表(假设它存在)也不能撤销,执行这条语句将永久删除该表。

> 重命名表

使用 RENAME TABLE 语句可以重命名一个表:

```
MariaDB [test]> rename table customers to customers3;
Query OK, 0 rows affected (0.05 sec)

MariaDB [test]> rename table customers3 to customers;
Query OK, 0 rows affected (0.04 sec)
```

RENAME TABLE 所做的仅是重命名一个表。可以使用下面的语句对多个表重命名:

```
MariaDB [test]> rename table vendors to vendors0,products to products0,orders to orders0;
Query OK, 0 rows affected (0.08 sec)

MariaDB [test]> rename table vendors0 to vendors,products0 to products,orders0 to orders;
Query OK, 0 rows affected (0.06 sec)
```

> 小结
>
> > 本章介绍了几条新SQL语句。 CREATE TABLE 用来创建新表, ALTER TABLE 用来更改表列(或其他诸如约束或索引等对象),而 DROP TABLE 用来完整地删除一个表。这些语句必须小心使用,并且应在做了备份后使用。本章还介绍了数据库引擎、定义主键和外键,以及其他重要的表和列选项。

DML语言

DML(Data Manipulation Language)语句:数据操纵语句,用于添加、删除、更新和查询数据库记录,并检查数据完整性,常用的语句关键字主要包括 insert、delete、udpate 和select 等。

插入数据

本章介绍如何利用SQL的 INSERT 语句将数据插入表中。

> 数据插入

顾名思义, INSERT 是用来插入(或添加)行到数据库表的。插入可以用几种方式使用:

- 插入完整的行;
- 插入行的一部分;
- 插入多行;
- 插入某些查询的结果。

下面将介绍这些内容。

插入及系统安全 可针对每个表或每个用户,利用MySQL的安全机制禁止使用 INSERT 语句。

> 插入完整的行

把数据插入表中的最简单的方法是使用基本的 INSERT 语法,它要求指定表名和被插入到新行中的值。下面举一个例子:

```
MariaDB [test]> insert into customers values(null,'Pep E. Lapew','100 Main Street','Los
Angeles','CA','90046','USA',null,null);
Query OK, 1 row affected (0.02 sec)

MariaDB [test]> select * from customers;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
6 rows in set (0.00 sec)
```

没有输出 INSERT 语句一般不会产生输出。

此例子插入一个新客户到 `customers` 表。存储到每个表列中的数据在 `VALUES` 子句中给出,对每个列必须提供一个值。如果某个列没有值(如上面的 `cust_contact` 和 `cust_email` 列),应该使用 NULL值(假定表允许对该列指定空值)。各个列必须以它们在表定义中出现的次序填充。第一列 `cust_id` 也为 NULL 。这是因为每次插入一个新行时,该列由MySQL自动增量。你不想给出一个值(这是MySQL的工作),又不能省略此列(如前所述,必须给出每个列),所以指定一个 NULL 值(它被MySQL忽略,MySQL在这里插入下一个可用的 `cust_id` 值)。

虽然这种语法很简单,但并不安全,应该尽量避免使用。上面的SQL语句高度依赖于表中列的定义次序,并且还依赖于其次序容易获得的信息。即使可得到这种次序信息,也不能保证下一次表结构变动后各个列保持完全相同的次序。因此,编写依赖于特定列次序的SQL语句是很不安全的。如果这样做,有时难免会出问题。

编写 INSERT 语句的更安全(不过更烦琐)的方法如下:

```
MariaDB [test]> insert into
customers(cust_name,cust_address,cust_city,cust_state,cust_zip,cust_country,cust_contact,cust_ema
il)
  -> values (null,'Pep E. Lapew','100 Main Street','Los Angeles','CA','90046','USA',null,null);
```

此例子完成与前一个 `INSERT` 语句完全相同的工作,但在表名后的括号里明确地给出了列名。在插入行时,MySQL将用 `VALUES` 列表中的相应值填入列表中的对应项。 `VALUES` 中的第一个值对应于第一个指定的列名。第二个值对应于第二个列名,如此等等。

因为提供了列名, `VALUES` 必须以其指定的次序匹配指定的列名,不一定按各个列出现在实际表中的次序。其优点是,即使表的结构改变,此 `INSERT` 语句仍然能正确工作。你会发现 `cust_id` 的 `NULL` 值是不必要的, `cust_id` 列并没有出现在列表中,所以不需要任何值。下面的 `INSERT` 语句填充所有列(与前面的一样),但以一种不同的次序填充。因为给出了列名,所以插入结果仍然正确:

```
MariaDB [test]> insert into
customers(cust_contact,cust_email,cust_address,cust_city,cust_state,cust_zip,cust_country)
  -> values ('Pep E. Lapew',null,null,'100 Main Street','Los Angeles','CA','90046','USA');
```

总是使用列的列表 一般不要使用没有明确给出列的列表的INSERT 语句。使用列的列表能使SQL代码继续发挥作用,即使表结构发生了变化。

仔细地给出值 不管使用哪种 INSERT 语法,都必须给出VALUES 的正确数目。如果不提供列名,则必须给每个表列提供一个值。如果提供列名,则必须对每个列出的列给出一个值。如果不这样,将产生一条错误消息,相应的行插入不成功。

使用这种语法,还可以省略列。这表示可以只给某些列提供值,给其他列不提供值。

(事实上你已经看到过这样的例子:当列名被明确列出时, cust_id 可以省略)

省略列 如果表的定义允许,则可以在 INSERT 操作中省略某些列。省略的列必须满足以下某个条件。

- 该列定义为允许 NULL 值(无值或空值)。
- 在表定义中给出默认值。这表示如果不给出值,将使用默认值。

如果对表中不允许 NULL 值且没有默认值的列不给出值,则MySQL将产生一条错误消息,并且相应的行插入不成功。

提高整体性能 数据库经常被多个客户访问,对处理什么请求以及用什么次序处理进行管理是MySQL的任务。INSERT 操作可能很耗时(特别是有很多索引需要更新时),而且它可能降低等待处理的 SELECT 语句的性能。

如果数据检索是最重要的(通常是这样),则你可以通过在INSERT 和 INTO 之间添加关键字 LOW_PRIORITY ,指示MySQL降低 INSERT 语句的优先级,如下所示:`insert low_priority into`顺便说一下,这也适用于下一章介绍的 UPDATE 和 DELETE 语句。

> 插入多个行

INSERT 可以插入一行到一个表中。但如果你想插入多个行怎么办?

可以使用多条 INSERT 语句,甚至一次提交它们,每条语句用一个分号结束,或者,只要每条 INSERT 语句中的列名(和次序)相同,可以如下组合各语句:

```
MariaDB [test]> insert into customers
(cust_name,cust_address,cust_city,cust_state,cust_zip,cust_country)
    -> values ('Pep E. LaPew','100 Main Street','Los Angeles','CA','90046','USA'),
    -> ('M. Martian','42 Galaxy Way','New York','NY','11213','USA');
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

MariaDB [test]> select * from customers;
+---------+----------------+--------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address       | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+--------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane     | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane   | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place      | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street   | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10006 | Pep E. Lapew   | 100 Main Street    | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street    | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way      | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+--------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
```

其中单条 INSERT 语句有多组值,每组值用一对圆括号括起来,用逗号分隔。

提高 **INSERT** 的性能 此技术可以提高数据库处理的性能,因为MySQL用单条 INSERT 语句处理多个插入比使用多条 INSERT语句快。

> 插入检索出的数据

INSERT 一般用来给表插入一个指定列值的行。但是, INSERT 还存在另一种形式,可以利用它将一条 SELECT 语句的结果插入表中。这就是所谓的 INSERT SELECT ,顾名思义,它是由一条 INSERT 语句和一条 SELECT语句组成的。

假如你想从另一表中合并客户列表到你的 customers 表。不需要每次读取一行,然后再将它用 INSERT 插入,可以如下进行:

新例子的说明 这个例子把一个名为 `custnew` 的表中的数据导入 `customers` 表中。为了试验这个例子,应该首先创建和填充 `custnew` 表。 `custnew` 表的结构与附录B中描述的 `customers` 表的相同。在填充 `custnew` 时,不应该使用已经在 `customers` 中使用过的 `cust_id` 值(如果主键值重复,后续的 `INSERT` 操作将会失败)或仅省略这列值让MySQL在导入数据的过程中产生新值。

```
MariaDB [test]> create table custnew as select * from customers;
Query OK, 8 rows affected (0.33 sec)
Records: 8  Duplicates: 0  Warnings: 0

MariaDB [test]> select * from custnew;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
8 rows in set (0.00 sec)

MariaDB [test]> desc custnew;
+--------------+-----------+------+-----+---------+-------+
| Field        | Type      | Null | Key | Default | Extra |
+--------------+-----------+------+-----+---------+-------+
| cust_id      | int(11)   | NO   |     | 0       |       |
| cust_name    | char(50)  | NO   |     | NULL    |       |
| cust_address | char(50)  | YES  |     | NULL    |       |
| cust_city    | char(50)  | YES  |     | NULL    |       |
| cust_state   | char(5)   | YES  |     | NULL    |       |
| cust_zip     | char(10)  | YES  |     | NULL    |       |
| cust_country | char(50)  | YES  |     | NULL    |       |
| cust_contact | char(50)  | YES  |     | NULL    |       |
| cust_email   | char(255) | YES  |     | NULL    |       |
+--------------+-----------+------+-----+---------+-------+
9 rows in set (0.00 sec)

MariaDB [test]> desc customers;
+--------------+-----------+------+-----+---------+----------------+
| Field        | Type      | Null | Key | Default | Extra          |
+--------------+-----------+------+-----+---------+----------------+
| cust_id      | int(11)   | NO   | PRI | NULL    | auto_increment |
| cust_name    | char(50)  | NO   |     | NULL    |                |
```

```
| cust_address | char(50)  | YES  |     | NULL    |                |
| cust_city    | char(50)  | YES  |     | NULL    |                |
| cust_state   | char(5)   | YES  |     | NULL    |                |
| cust_zip     | char(10)  | YES  |     | NULL    |                |
| cust_country | char(50)  | YES  |     | NULL    |                |
| cust_contact | char(50)  | YES  |     | NULL    |                |
| cust_email   | char(255) | YES  |     | NULL    |                |
+--------------+-----------+------+-----+---------+----------------+
9 rows in set (0.00 sec)

MariaDB [test]> select * from custnew;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
8 rows in set (0.00 sec)

MariaDB [test]> update custnew set cust_id=20001 where cust_id=10001;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20002 where cust_id=10002;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20003 where cust_id=10003;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20004 where cust_id=10004;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
MariaDB [test]> update custnew set cust_id=20005 where cust_id=10005;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20006 where cust_id=10006;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20007 where cust_id=10007;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> update custnew set cust_id=20008 where cust_id=10008;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> select * from custnew;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   20001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   20002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   20003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   20004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   20005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   20006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
8 rows in set (0.00 sec)

# 将custnew表中的数据导入customers表

MariaDB [test]> insert into customers select * from custnew;
Query OK, 8 rows affected (0.06 sec)
Records: 8  Duplicates: 0  Warnings: 0

MariaDB [test]> select * from customers;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+

| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
```

```
cust_country | cust_contact | cust_email          |
+---------+----------------+--------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
|   20001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   20002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   20003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   20004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   20005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   20006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+--------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
16 rows in set (0.00 sec)
```

这个例子使用 `INSERT SELECT` 从 `custnew` 中将所有数据导入`customers` 。 `SELECT` 语句从 `custnew` 检索出要插入的值,而不是列出它们。 `SELECT` 中列出的每个列对应于 `customers` 表名后所跟的列表中的每个列。这条语句将插入多少行有赖于 `custnew` 表中有多少行。如果这个表为空,则没有行被插入(也不产生错误,因为操作仍然是合法的)。如果这个表确实含有数据,则所有数据将被插入到 `customers` 。

这个例子导入了 cust_id (假设你能够确保 cust_id 的值不重复)。你也可以简单地省略这列(从 INSERT 和 SELECT 中),这样MySQL就会生成新值。

INSERT SELECT 中的列名 为简单起见,这个例子在 INSERT 和 SELECT 语句中使用了相同的列名。但是,不一定要求列名匹配。事实上,MySQL甚至不关心 SELECT 返回的列名。它使用的是列的位置,因此 SELECT 中的第一列(不管其列名)将用来填充表列中指定的第一个列,第二列将用来填充表列中指定的第二个列,如此等等。这对于从使用不同列名的表中导入数据是非常有用的。

INSERT SELECT 中 SELECT 语句可包含 WHERE 子句以过滤插入的数据。

> 小结
>
> > 本章介绍如何将行插入到数据库表。我们学习了使用 INSERT 的几种方法,以及为什么要明确使用列名,学习了如何用 INSERT SELECT 从其他表中导入行。下一章讲述如何使用 UPDATE 和 DELETE 进一步操纵表数据。

更新和删除数据

本章介绍如何利用 UPDATE 和 DELETE 语句进一步操纵表数据。

> 更新数据

为了更新(修改)表中的数据,可使用 UPDATE 语句。可采用两种方式使用 UPDATE :

- 更新表中特定行;
- 更新表中所有行。

下面分别对它们进行介绍。

不要省略 **WHERE** 子句 在使用 UPDATE 时一定要注意细心。因为稍不注意,就会更新表中所有行。在使用这条语句前,请完整地阅读本节。

UPDATE 与安全 可以限制和控制 UPDATE 语句的使用

UPDATE 语句非常容易使用,甚至可以说是太容易使用了。基本的UPDATE 语句由3部分组成,分别是:

- 要更新的表;
- 列名和它们的新值;
- 确定要更新行的过滤条件。

举一个简单例子。

> > 客户`cust_id=10005` 现在有了电子邮件地址`elmer@fudd.com`,因此他的记录需要更新,语句如下:

```
MariaDB [test]> update customers set cust_email = 'elmer@fudd.com' where cust_id = 10005;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> select cust_email from customers where cust_id =10005;
+----------------+
| cust_email     |
+----------------+
| elmer@fudd.com |
+----------------+
1 row in set (0.00 sec)
```

`UPDATE` 语句总是以要更新的表的名字开始。在此例子中,要更新的表的名字为 `customers` 。 `SET` 命令用来将新值赋给被更新的列。如这里所示, `SET` 子句设置 `cust_email` 列为指定的值:`set cust_email = 'elmer@fudd.com'`

`UPDATE`语句以 `WHERE` 子句结束,它告诉MySQL更新哪一行。没有`WHERE` 子句,MySQL将会用这个电子邮件地址更新 `customers` 表中所有行,这不是我们所希望的。

更新多个列的语法稍有不同:

```
MariaDB [test]> update customers set cust_name = 'The Fudds',cust_email = 'elmer@fudd.com' where
cust_id = 10005;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> select * from customers where cust_id =10005;
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+----------------+
| cust_id | cust_name | cust_address     | cust_city | cust_state | cust_zip | cust_country |
cust_contact | cust_email     |
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+----------------+
|   10005 | The Fudds | 4545 53rd Street | Chicago   | IL         | 54545    | USA          | E
Fudd       | elmer@fudd.com |
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+----------------+
1 row in set (0.00 sec)
```

在更新多个列时,只需要使用单个 `SET` 命令,每个“列=值”对之间用逗号分隔(最后一列之后不用逗号)。在此例子中,更新客户 `10005` 的 `cust_name` 和 `cust_email` 列。

在 **UPDATE** 语句中使用子查询 UPDATE 语句中可以使用子查询,使得能用 SELECT 语句检索出的数据更新列数据。

IGNORE 关键字 如果用 UPDATE 语句更新多行,并且在更新这些行中的一行或多行时出一个现错误,则整个 UPDATE 操作被取消(错误发生前更新的所有行被恢复到它们原来的值)。为即使是发生错误,也继续进行更新,可使用 IGNORE 关键字,如下所示: `UPDATE IGNORE customers...`

> 为了删除某个列的值,可设置它为 NULL (假如表定义允许 NULL 值)。

```
MariaDB [test]> update customers set cust_email = null where cust_id = 10005;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [test]> select * from customers where cust_id =10005;
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+------------+
| cust_id | cust_name | cust_address     | cust_city | cust_state | cust_zip | cust_country |
cust_contact | cust_email |
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+------------+
|   10005 | The Fudds | 4545 53rd Street | Chicago   | IL         | 54545    | USA          | E
Fudd       | NULL       |
+---------+-----------+------------------+-----------+------------+----------+--------------+----
----------+------------+
1 row in set (0.00 sec)
```

其中 `NULL` 用来去除 `cust_email` 列中的值。

> 删除数据

为了从一个表中删除(去掉)数据,使用 DELETE 语句。可以两种方式使用 DELETE :

- 从表中删除特定的行;
- 从表中删除所有行。

下面分别对它们进行介绍。

不要省略 **WHERE** 子句 在使用 DELETE 时一定要注意细心。因为稍不注意,就会错误地删除表中所有行。在使用这条语句前,请完整地阅读本节。

DELETE 与安全 可以限制和控制 DELETE 语句的使用

前面说过, UPDATE 非常容易使用,而 DELETE 更容易使用。下面的语句从 customers 表中删除一行:

> 只删除客户 10006

```
MariaDB [test]> delete from customers where cust_id = 10006;
Query OK, 1 row affected (0.04 sec)

MariaDB [test]> select * from customers;
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
| cust_id | cust_name      | cust_address        | cust_city   | cust_state | cust_zip |
cust_country | cust_contact | cust_email          |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   10005 | The Fudds      | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   10007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   10008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
|   20001 | Coyote Inc.    | 200 Maple Lane      | Detroit     | MI         | 44444    | USA
     | Y Lee        | ylee@coyote.com     |
|   20002 | Mouse House    | 333 Fromage Lane    | Columbus    | OH         | 43333    | USA
     | Jerry Mouse  | NULL                |
|   20003 | Wascals        | 1 Sunny Place       | Muncie      | IN         | 42222    | USA
     | Jim Jones    | rabbit@wascally.com |
|   20004 | Yosemite Place | 829 Riverside Drive | Phoenix     | AZ         | 88888    | USA
     | Y Sam        | sam@yosemite.com    |
|   20005 | E Fudd         | 4545 53rd Street    | Chicago     | IL         | 54545    | USA
     | E Fudd       | NULL                |
|   20006 | Pep E. Lapew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20007 | Pep E. LaPew   | 100 Main Street     | Los Angeles | CA         | 90046    | USA
     | NULL         | NULL                |
|   20008 | M. Martian     | 42 Galaxy Way       | New York    | NY         | 11213    | USA
     | NULL         | NULL                |
+---------+----------------+---------------------+-------------+------------+----------+---------
-----+--------------+---------------------+
15 rows in set (0.00 sec)
```

这条语句很容易理解。 `DELETE FROM` 要求指定从中删除数据的表名。 `WHERE` 子句过滤要删除的行。在这个例子中,只删除客户 `10006` 。如果省略 `WHERE` 子句,它将删除表中每个客户。

`DELETE` 不需要列名或通配符。 `DELETE` 删除整行而不是删除列。为了删除指定的列,请使用 `UPDATE` 语句。

删除表的内容而不是表 `DELETE` 语句从表中删除行,甚至是删除表中所有行。但是, `DELETE` 不删除表本身。

更快的删除 如果想从表中删除所有行,不要使用 `DELETE` 。可使用 `TRUNCATE TABLE` 语句,它完成相同的工作,但速度更快(TRUNCATE 实际是删除原来的表并重新创建一个表,而不是逐行删除表中的数据)。

> 更新和删除的指导原则

前一节中使用的 UPDATE 和 DELETE 语句全都具有 WHERE 子句,这样做的理由很充分。如果省略了 WHERE 子句,则 UPDATE 或 DELETE 将被应用到表中所有的行。换句话说,如果执行 UPDATE 而不带 WHERE 子句,则表中每个行都将用新值更新。类似地,如果执行 DELETE 语句而不带 WHERE 子句,表的所有数据都将被删除。

下面是许多SQL程序员使用 UPDATE 或 DELETE 时所遵循的习惯。

- 除非确实打算更新和删除每一行,否则绝对不要使用不带 WHERE子句的 UPDATE 或 DELETE 语句。
- 保证每个表都有主键,尽可能像 WHERE 子句那样使用它(可以指定各主键、多个值或值的范围)。
- 在对 UPDATE 或 DELETE 语句使用 WHERE 子句前,应该先用 SELECT 进行测试,保证它过滤的是正确的记录,以防编写的 WHERE 子句不正确。
- 使用强制实施引用完整性的数据库,这样MySQL将不允许删除具有与其他表相关联的数据的行。

小心使用 MySQL没有撤销(undo)按钮。应该非常小心地使用 UPDATE 和 DELETE ,否则你会发现自己更新或删除了错误的数据。

> 小结
>
> > 我们在本章中学习了如何使用 UPDATE 和 DELETE 语句处理表中的数据。我们学习了这些语句的语法,知道了它们固有的危险性。本章中还讲解了为什么 WHERE 子句对 UPDATE 和 DELETE 语句很重要,并且给出了应该遵循的一些指导原则,以保证数据的安全。

使用视图

本章将介绍视图究竟是什么,它们怎样工作,何时使用它们。我们还将看到如何利用视图简化前面章节中执行的某些SQL操作。

> 视图

需要**MySQL 5** MySQL 5添加了对视图的支持。因此,本章内容适用于MySQL 5及以后的版本。

视图是虚拟的表。与包含数据的表不一样,视图只包含使用时动态检索数据的查询。

理解视图的最好方法是看一个例子。用下面的 SELECT 语句从3个表中检索数据:

```
MariaDB [test]> select cust_name,cust_contact from customers,orders,orderitems
  -> where customers.cust_id=orders.cust_id and orderitems.order_num = orders.order_num
  -> and prod_id='TNT2';
+----------------+--------------+
| cust_name      | cust_contact |
+----------------+--------------+
| Coyote Inc.    | Y Lee        |
| Yosemite Place | Y Sam        |
+----------------+--------------+
2 rows in set (0.00 sec)
```

此查询用来检索订购了某个特定产品的客户。任何需要这个数据的人都必须理解相关表的结构,并且知道如何创建查询和对表进行联结。为了检索其他产品(或多个产品)的相同数据,必须修改最后的 WHERE 子句。

现在,假如可以把整个查询包装成一个名为 productcustomers 的虚拟表,则可以如下轻松地检索出相同的数据:`select cust_name,cust_contact from productcustomers where prod_id = 'TNT2';`

这就是视图的作用。 productcustomers 是一个视图,作为视图,它不包含表中应该有的任何列或数据,它包含的是一个SQL查询(与上面用以正确联结表的相同的查询)。

1.为什么使用视图

我们已经看到了视图应用的一个例子。下面是视图的一些常见应用。

- 重用SQL语句。
- 简化复杂的SQL操作。在编写查询后,可以方便地重用它而不必知道它的基本查询细节。
- 使用表的组成部分而不是整个表。
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

在视图创建之后,可以用与表基本相同的方式利用它们。可以对视图执行 SELECT 操作,过滤和排序数据,将视图联结到其他视图或表,甚至能添加和更新数据(添加和更新数据存在某些限制。关于这个内容稍后还要做进一步的介绍)。

重要的是知道视图仅仅是用来查看存储在别处的数据的一种设施。视图本身不包含数据,因此它们返回的数据是从其他表中检索出来的。在添加或更改这些表中的数据时,视图将返回改变过的数据。

性能问题 因为视图不包含数据,所以每次使用视图时,都必须处理查询执行时所需的任一个检索。如果你用多个联结和过滤创建了复杂的视图或者嵌套了视图,可能会发现性能下降得很厉害。因此,在部署使用了大量视图的应用前,应该进行测试。

2.视图的规则和限制

下面是关于视图创建和使用的一些最常见的规则和限制。

- 与表一样,视图必须唯一命名(不能给视图取与别的视图或表相同的名字)。
- 对于可以创建的视图数目没有限制。
- 为了创建视图,必须具有足够的访问权限。这些限制通常由数据库管理人员授予。
- 视图可以嵌套,即可以利用从其他视图中检索数据的查询来构造一个视图。
- ORDER BY 可以用在视图中,但如果从该视图检索数据 SELECT 中也含有 ORDER BY ,那么该视图中的 ORDER BY 将被覆盖。
- 视图不能索引,也不能有关联的触发器或默认值。
- 视图可以和表一起使用。例如,编写一条联结表和视图的 SELECT语句。

> 使用视图

在理解什么是视图(以及管理它们的规则及约束)后,我们来看一下视图的创建。

- 视图用 CREATE VIEW 语句来创建。
- 使用 SHOW CREATE VIEW viewname ;来查看创建视图的语句。
- 用 DROP 删除视图,其语法为 DROP VIEW viewname;。
- 更新视图时,可以先用DROP再用CREATE,也可以直接用CREATE ORREPLACE VIEW。如果要更新的视图不存在,则第 2 条更新语句会创建一个视图;如果要更新的视图存在,则第 2 条更新语句会替换原有视图。

1.利用视图简化复杂的联结

视图的最常见的应用之一是隐藏复杂的SQL,这通常都会涉及联结。

请看下面的例子:

> 创建一个名为 productcustomers 的视图,它联结三个表,以返回已订购了任意产品的所有客户的列表。如果执行SELECT * FROM productcustomers ,将列出订购了任意产品的客户。

```
MariaDB [test]> create view productcustomers as select cust_name,cust_contact,prod_id from
customers,orders,orderitems where customers.cust_id=orders.cust_id and orderitems.order_num =
orders.order_num ;
Query OK, 0 rows affected (0.06 sec)

MariaDB [test]> select * from productcustomers;
+----------------+--------------+---------+
| cust_name      | cust_contact | prod_id |
+----------------+--------------+---------+
| Coyote Inc.    | Y Lee        | ANV01   |
| Coyote Inc.    | Y Lee        | ANV02   |
| Coyote Inc.    | Y Lee        | TNT2    |
| Coyote Inc.    | Y Lee        | FB      |
| Coyote Inc.    | Y Lee        | FB      |
| Coyote Inc.    | Y Lee        | OL1     |
| Coyote Inc.    | Y Lee        | SLING   |
| Coyote Inc.    | Y Lee        | ANV03   |
| Wascals        | Jim Jones    | JP2000  |
| Yosemite Place | Y Sam        | TNT2    |
| The Fudds      | E Fudd       | FC      |
+----------------+--------------+---------+
11 rows in set (0.00 sec)
```

> 检索订购了产品 TNT2 的客户

```
MariaDB [test]> select cust_name,cust_contact from productcustomers where prod_id='TNT2';
+----------------+--------------+
| cust_name      | cust_contact |
+----------------+--------------+
| Coyote Inc.    | Y Lee        |
| Yosemite Place | Y Sam        |
+----------------+--------------+
2 rows in set (0.00 sec)
```

这条语句通过 WHERE 子句从视图中检索特定数据。在MySQL处理此查询时,它将指定的 WHERE 子句添加到视图查询中的已有WHERE 子句中,以便正确过滤数据。

可以看出,视图极大地简化了复杂SQL语句的使用。利用视图,可一次性编写基础的SQL,然后根据需要多次使用。

创建可重用的视图 创建不受特定数据限制的视图是一种好办法。例如,上面创建的视图返回生产所有产品的客户而不仅仅是 生产TNT2 的客户。扩展视图的范围不仅使得它能被重用,而且甚至更有用。这样做不需要创建和维护多个类似视图。

2.用视图重新格式化检索出的数据

如上所述,视图的另一常见用途是重新格式化检索出的数据。下面的 SELECT 语句在单个组合计算列中返回供应商名和位置:

```
MariaDB [test]> select concat(rtrim(vend_name),' (',rtrim(vend_country),')')
    -> as vend_title
    -> from vendors
    -> order by vend_name;
+--------------------------+
| vend_title               |
+--------------------------+
| ACME (USA)               |
| Anvils R Us (USA)        |
| Furball Inc. (USA)       |
| Jet Set (England)        |
| Jouets Et Ours (France)  |
| LT Supplies (USA)        |
+--------------------------+
6 rows in set (0.00 sec)
```

现在,假如经常需要这个格式的结果。不必在每次需要时执行联结,创建一个视图,每次需要时使用它即可。为把此语句转换为视图,可按如下进行:

```
MariaDB [test]> create view vendorlocation as
    -> select concat(rtrim(vend_name),' (',rtrim(vend_country),')')
    -> as vend_title
    -> from vendors
    -> order by vend_name;
Query OK, 0 rows affected (0.32 sec)
```

这条语句使用与以前的 SELECT 语句相同的查询创建视图。为了检索出以创建所有邮件标签的数据,可如下进行:

```
MariaDB [test]> select * from vendorlocation;
+--------------------------+
| vend_title               |
+--------------------------+
| ACME (USA)               |
| Anvils R Us (USA)        |
| Furball Inc. (USA)       |
| Jet Set (England)        |
| Jouets Et Ours (France)  |
| LT Supplies (USA)        |
+--------------------------+
6 rows in set (0.00 sec)
```

3.用视图过滤不想要的数据

视图对于应用普通的WHERE子句也很有用。

> 定义customeremaillist 视图,它过滤没有电子邮件地址的客户。为此目的,可使用下面的语句:

```
MariaDB [test]> create view customeremaillist as
    -> select cust_id,cust_name,cust_email
    -> from customers
    -> where cust_email is not null;
Query OK, 0 rows affected (0.05 sec)

MariaDB [test]> select * from customeremaillist;
+---------+----------------+---------------------+
| cust_id | cust_name      | cust_email          |
+---------+----------------+---------------------+
|   10001 | Coyote Inc.    | ylee@coyote.com     |
|   10003 | Wascals        | rabbit@wascally.com |
|   10004 | Yosemite Place | sam@yosemite.com    |
|   20001 | Coyote Inc.    | ylee@coyote.com     |
|   20003 | Wascals        | rabbit@wascally.com |
|   20004 | Yosemite Place | sam@yosemite.com    |
+---------+----------------+---------------------+
6 rows in set (0.00 sec)
```

在发送电子邮件到邮件列表时,需要排除没有电子邮件地址的用户。这里的 WHERE 子句过滤了 cust_email 列中具有NULL 值的那些行,使他们不被检索出来。

WHERE 子句与 **WHERE** 子句 如果从视图检索数据时使用了一条WHERE 子句,则两组子句(一组在视图中,另一组是传递给视图的)将自动组合。

4.使用视图与计算字段

视图对于简化计算字段的使用特别有用。

> 检索某个特定订单中的物品,计算每种物品的总价格

```
MariaDB [test]> select prod_id,quantity,item_price,quantity*item_price as expanded_price
    -> from orderitems
    -> where order_num = 20005;
+---------+----------+------------+----------------+
| prod_id | quantity | item_price | expanded_price |
+---------+----------+------------+----------------+
| ANV01   |       10 |       5.99 |          59.90 |
| ANV02   |        3 |       9.99 |          29.97 |
| TNT2    |        5 |      10.00 |          50.00 |
| FB      |        1 |      10.00 |          10.00 |
+---------+----------+------------+----------------+
4 rows in set (0.00 sec)
```

为将其转换为一个视图,如下进行:

```
MariaDB [test]> create view orderitemsexpanded as
    -> select prod_id,quantity,item_price,quantity*item_price as expanded_price
    -> from orderitems;
Query OK, 0 rows affected (0.06 sec)
```

为检索订单 20005 的详细内容(上面的输出),如下进行:

```
MariaDB [test]> create view orderitemsexpanded as
    -> select order_num,prod_id,quantity,item_price,quantity*item_price as expanded_price
    -> from orderitems;
Query OK, 0 rows affected (0.04 sec)

MariaDB [test]> select * from orderitemsexpanded where order_num = 20005;
+-----------+---------+----------+------------+----------------+
| order_num | prod_id | quantity | item_price | expanded_price |
+-----------+---------+----------+------------+----------------+
|     20005 | ANV01   |       10 |       5.99 |          59.90 |
|     20005 | ANV02   |        3 |       9.99 |          29.97 |
|     20005 | TNT2    |        5 |      10.00 |          50.00 |
|     20005 | FB      |        1 |      10.00 |          10.00 |
+-----------+---------+----------+------------+----------------+
4 rows in set (0.00 sec)
```

可以看到,视图非常容易创建,而且很好使用。正确使用,视图可极大地简化复杂的数据处理。

5.更新视图

迄今为止的所有视图都是和 SELECT 语句使用的。然而,视图的数据能否更新?答案视情况而定。

通常,视图是可更新的(即,可以对它们使用 INSERT 、 UPDATE 和DELETE)。更新一个视图将更新其基表(可以回忆一下,视图本身没有数据)。如果你对视图增加或删除行,实际上是对其基表增加或删除行。

但是,并非所有视图都是可更新的。基本上可以说,如果MySQL不能正确地确定被更新的基数据,则不允许更新(包括插入和删除)。这实际上意味着,如果视图定义中有以下操作,则不能进行视图的更新:

- 分组(使用 GROUP BY 和 HAVING);
- 联结;
- 子查询;
- 并;
- 聚集函数(Min() 、 Count() 、 Sum() 等);
- DISTINCT;
- 导出(计算)列。

换句话说,本章许多例子中的视图都是不可更新的。这听上去好像是一个严重的限制,但实际上不是,因为视图主要用于数据检索。

可能的变动 上面列出的限制自MySQL 5以来是正确的。不过,未来的MySQL很可能会取消某些限制。

将视图用于检索 一般,应该将视图用于检索(SELECT 语句)而不用于更新(INSERT 、 UPDATE 和 DELETE)。

> 小结
>
> > 视图为虚拟的表。它们包含的不是数据而是根据需要检索数据的查询。视图提供了一种MySQL的 SELECT 语句层次的封装,可用来简化数据处理以及重新格式化基础数据或保护基础数据。

使用存储过程

本章介绍什么是存储过程,为什么要使用存储过程以及如何使用存储过程,并且介绍创建和使用存储过程的基本语法。

> 存储过程

需要**MySQL 5** MySQL 5添加了对存储过程的支持,因此,本章内容适用于MySQL 5及以后的版本。

迄今为止,使用的大多数SQL语句都是针对一个或多个表的单条语句。并非所有操作都这么简单,经常会有一个完整的操作需要多条语句才能完成。例如,考虑以下的情形。

- 为了处理订单,需要核对以保证库存中有相应的物品。
- 如果库存有物品,这些物品需要预定以便不将它们再卖给别的人,
  并且要减少可用的物品数量以反映正确的库存量。
- 库存中没有的物品需要订购,这需要与供应商进行某种交互。
- 关于哪些物品入库(并且可以立即发货)和哪些物品退订,需要通知相应的客户。

这显然不是一个完整的例子,它甚至超出了本书中所用样例表的范围,但足以帮助表达我们的意思了。执行这个处理需要针对许多表的多条MySQL语句。此外,需要执行的具体语句及其次序也不是固定的,它们可能会(和将)根据哪些物品在库存中哪些不在而变化。

那么,怎样编写此代码?可以单独编写每条语句,并根据结果有条件地执行另外的语句。在每次需要这个处理时(以及每个需要它的应用中)都必须做这些工作。

可以创建存储过程。存储过程简单来说,就是为以后的使用而保存的一条或多条MySQL语句的集合。可将其视为批文件,虽然它们的作用不仅限于批处理。

> 为什么要使用存储过程

既然我们知道了什么是存储过程,那么为什么要使用它们呢?有许多理由,下面列出一些主要的理由。

- 通过把处理封装在容易使用的单元中,简化复杂的操作(正如前面例子所述)。
- 由于不要求反复建立一系列处理步骤,这保证了数据的完整性。如果所有开发人员和应用程序都使用同一(试验和测试)存储过程,则所使用的代码都是相同的。

这一点的延伸就是防止错误。需要执行的步骤越多,出错的可能性就越大。防止错误保证了数据的一致性。

- 简化对变动的管理。如果表名、列名或业务逻辑(或别的内容)有变化,只需要更改存储过程的代码。使用它的人员甚至不需要知道这些变化。

这一点的延伸就是安全性。通过存储过程限制对基础数据的访问减少了数据讹误(无意识的或别的原因所导致的数据讹误)的机会。

- 提高性能。因为使用存储过程比使用单独的SQL语句要快。
- 存在一些只能用在单个请求中的MySQL元素和特性,存储过程可以使用它们来编写功能更强更灵活的代码(在下一章的例子中可以看到。)

换句话说,使用存储过程有3个主要的好处,即简单、安全、高性能。显然,它们都很重要。不过,在将SQL代码转换为存储过程前,也必须知道它的一些缺陷。

- 一般来说,存储过程的编写比基本SQL语句复杂,编写存储过程需要更高的技能,更丰富的经验。
- 你可能没有创建存储过程的安全访问权限。许多数据库管理员限
  制存储过程的创建权限,允许用户使用存储过程,但不允许他们创建存储过程。

尽管有这些缺陷,存储过程还是非常有用的,并且应该尽可能地使用。

不能编写存储过程**?**你依然可以使用 MySQL将编写存储过程的安全和访问与执行存储过程的安全和访问区分开来。这是好事情。即使你不能(或不想)编写自己的存储过程,也仍然可以在适当的时候执行别的存储过程。

> 使用存储过程

使用存储过程需要知道如何执行(运行)它们。存储过程的执行远比其定义更经常遇到,因此,我们将从执行存储过程开始介绍。然后再介绍创建和使用存储过程。

1.执行存储过程

MySQL称存储过程的执行为调用,因此MySQL执行存储过程的语句为 CALL 。 CALL 接受存储过程的名字以及需要传递给它的任意参数。请看以下例子:

> 执行名为 productpricing 的存储过程,它计算并返回产品的最低、最高和平均价格。

```
MariaDB [test]> call productpricing (@pricelow,@pricehigh,@priceaverage);
```

2.创建存储过程

正如所述,编写存储过程并不是微不足道的事情。为让你了解这个过程,请看一个例子。

> 返回产品平均价格的存储过程

```
MariaDB [test]> create procedure productpricing() begin select avg(prod_price) as priceaverage
from products; end;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MariaDB server version for the right syntax to use near '' at line 1
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MariaDB server version for the right syntax to use near 'end' at line 1

MariaDB [test]> delimiter //
MariaDB [test]> create procedure productpricing() begin select avg(prod_price) as priceaverage
from products; end //
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

我们稍后介绍第一条和最后一条语句。此存储过程名为`productpricing` ,用 `CREATE PROCEDURE productpricing()` 语句定义。如果存储过程接受参数,它们将在 `()` 中列举出来。此存储过程没有参数,但后跟的 `()` 仍然需要。 `BEGIN` 和 `END` 语句用来限定存储过程体,过程体本身仅是一个简单的 `SELECT` 语句。

在MySQL处理这段代码时,它创建一个新的存储过程 `product-pricing` 。没有返回数据,因为这段代码并未调用存储过程,这里只是为以后使用而创建它。

mysql 命令行客户机的分隔符 如果你使用的是 mysql 命令行实用程序,应该仔细阅读此说明。

默认的MySQL语句分隔符为 ; (正如你已经在迄今为止所使用的MySQL语句中所看到的那样)。 mysql 命令行实用程序也使用 ; 作为语句分隔符。如果命令行实用程序要解释存储过程自身内的 ; 字符,则它们最终不会成为存储过程的成分,这会使存储过程中的SQL出现句法错误。

解决办法是临时更改命令行实用程序的语句分隔符,如下所示

```
delimiter//
create procedure productpricing()
begin
  select avg(prod_price) as priceaverage
  from products;
end //
delimiter;
```

其中, `DELIMITER //` 告诉命令行实用程序使用`//` 作为新的语句结束分隔符,可以看到标志存储过程结束的 `END` 定义为 `END//` 而不是 `END;` 。这样,存储过程体内的 `;` 仍然保持不动,并且正确地传递给数据库引擎。最后,为恢复为原来的语句分隔符,可使用 `DELIMITER ;` 。除 `\` 符号外,任何字符都可以用作语句分隔符。如果你使用的是 mysql 命令行实用程序,在阅读本章时请记住这里的内容。

那么,如何使用这个存储过程?如下所示:

```
MariaDB [test]> call productpricing();
+--------------+
| priceaverage |
+--------------+
|    16.133571 |
+--------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

`CALL productpricing();` 执行刚创建的存储过程并显示返回的结果。因为存储过程实际上是一种函数,所以存储过程名后需要有 () 符号(即使不传递参数也需要)。

3.删除存储过程

存储过程在创建之后,被保存在服务器上以供使用,直至被删除。删除命令从服务器中删除存储过程。

> 为删除刚创建的存储过程,可使用以下语句:

```
MariaDB [test]> drop procedure productpricing;
Query OK, 0 rows affected (0.00 sec)
```

这条语句删除刚创建的存储过程。请注意没有使用后面的 `()` ,只给出存储过程名。

仅当存在时删除 如果指定的过程不存在,则 `DROP PROCEDURE` 将产生一个错误。当过程存在想删除它时(如果过程不存在也不产生错误)可使用 `DROP PROCEDURE IF EXISTS` 。

```
MariaDB [test]> DROP PROCEDURE IF EXISTS productpricing;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

4.使用参数

productpricing 只是一个简单的存储过程,它简单地显示 SELECT 语句的结果。一般,存储过程并不显示结果,而是把结果返回给你指定的变量。

变量**(variable)** 内存中一个特定的位置,用来临时存储数据。

> 以下是 productpricing 的修改版本(如果不先删除此存储过程,则不能再次创建它):

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure productpricing(out pl decimal (8,2),out ph decimal(8,2),out pa
decimal(8,2)) begin select min(prod_price) into pl from products;select max(prod_price) into ph
from products;select avg(prod_price) into pa from products;  end//
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

此存储过程接受3个参数: pl 存储产品最低价格, ph 存储产品最高价格, pa 存储产品平均价格。每个参数必须具有指定的类型,这里使用十进制值。关键字 OUT 指出相应的参数用来从存储过程传出一个值(返回给调用者)。MySQL支持 IN (传递给存储过程)、 OUT (从存储过程传出,如这里所用)和 INOUT (对存储过程传入和传出)类型的参数。存储过程的代码位于 BEGIN 和 END 语句内,如前所见,它们是一系列SELECT 语句,用来检索值,然后保存到相应的变量(通过指定 INTO 关键字)。

参数的数据类型 存储过程的参数允许的数据类型与表中使用的数据类型相同。附录D列出了这些类型。注意,记录集不是允许的类型,因此,不能通过一个参数返回多个行和列。这就是前面的例子为什么要使用3个参数(和3条 SELECT 语句)的原因。

> 为调用此修改过的存储过程,必须指定3个变量名,如下所示:

```
MariaDB [test]> call productpricing(@pricelow,@pricehigh,@priceaverage);
Query OK, 1 row affected, 1 warning (0.00 sec)

MariaDB [test]> call productpricing(@pl,@ph,@pa);
Query OK, 1 row affected, 1 warning (0.00 sec)
```

由于此存储过程要求3个参数,因此必须正好传递3个参数,不多也不少。所以,这条 CALL 语句给出3个参数。它们是存储过程将保存结果的3个变量的名字。

变量名 所有MySQL变量都必须以 `@` 开始。

在调用时,这条语句并不显示任何数据。它返回以后可以显示(或在其他处理中使用)的变量。

> 为了显示检索出的产品平均价格,可如下进行:

```
MariaDB [test]> select @pl;
+------+
| @pl  |
+------+
| 2.50 |
+------+
1 row in set (0.00 sec)

MariaDB [test]> select @pricelow;
+-----------+
| @pricelow |
+-----------+
|      2.50 |
+-----------+
1 row in set (0.00 sec)
```

> 为了获得3个值,可使用以下语句:

```
MariaDB [test]> select @pl,@ph,@pa;
+------+-------+-------+
| @pl  | @ph   | @pa   |
+------+-------+-------+
| 2.50 | 55.00 | 16.13 |
+------+-------+-------+
1 row in set (0.00 sec)
```

> 下面是另外一个例子,这次使用 IN 和 OUT 参数。 ordertotal 接受订单号并返回该订单的合计:

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure ordertotal(in onumber int,out ototal decimal(8,2)) begin select
sum(item_price*quantity) from orderitems where order_num=onumber into ototal; end//
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

`onumber` 定义为 `IN` ,因为订单号被传入存储过程。 `ototal` 定义为 `OUT` ,因为要从存储过程返回合计。 `SELECT` 语句使用这两个参数, `WHERE` 子句使用 `onumber` 选择正确的行, `INTO` 使用 `ototal` 存储计算出来的合计。

> 为调用这个新存储过程,可使用以下语句:

```
MariaDB [test]> call ordertotal(20005,@total);
Query OK, 1 row affected (0.00 sec)
```

必须给 ordertotal 传递两个参数;第一个参数为订单号,第二个参数为包含计算出来的合计的变量名。

> 为了显示此合计,可如下进行:

```
MariaDB [test]> select @total;
+--------+
| @total |
+--------+
| 149.87 |
+--------+
1 row in set (0.00 sec)
```

@total 已由 ordertotal 的 CALL 语句填写, SELECT 显示它包含的值。

> 为了得到另一个订单的合计显示,需要再次调用存储过程,然后重新显示变量:

```
MariaDB [test]> call ordertotal(20009,@total);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select @total;
+--------+
| @total |
+--------+
|  38.47 |
+--------+
1 row in set (0.00 sec)
```

`into 变量`的位置可以在`from`前面也可以在最后

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure ordertotal0(in onumber int,out xtotal decimal(8,2)) begin select
sum(item_price*quantity) into xtotal from orderitems where order_num=onumber ; end//
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
MariaDB [test]> call ordertotal0;
ERROR 1318 (42000): Incorrect number of arguments for PROCEDURE test.ordertotal0; expected 2, got
0
MariaDB [test]> call ordertotal0(20005,@total);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select @total;
+--------+
| @total |
+--------+
| 149.87 |
+--------+
1 row in set (0.00 sec)
```

5.建立智能存储过程

迄今为止使用的所有存储过程基本上都是封装MySQL简单的 SELECT语句。虽然它们全都是有效的存储过程例子,但它们所能完成的工作你直接用这些被封装的语句就能完成(如果说它们还能带来更多的东西,那就是使事情更复杂)。只有在存储过程内包含业务规则和智能处理时,它们的威力才真正显现出来。

考虑这个场景。你需要获得与以前一样的订单合计,但需要对合计增加营业税,不过只针对某些顾客(或许是你所在州中那些顾客)。那么,你需要做下面几件事情:

- 获得合计(与以前一样);
- 把营业税有条件地添加到合计;
- 返回合计(带或不带税)。

存储过程的完整工作如下:

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure ordertotal(
    -> in onumber int,
    -> in taxable boolean,
    -> out ototal decimal(8,2)) comment 'Obtain order total, optionally adding tax'
    -> begin
    -> declare total decimal(8,2);
    -> declare taxrate int default 6;
    -> select sum(item_price*quantity)
    -> from orderitems
    -> where order_num = onumber
    -> into total;
    -> if taxable then
    -> select total+(total/100*taxrate) into total;
    -> end if;
    -> select total into ototal;
    -> end //
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;


[root@mastera0 ~]# vi procedure.mysql
[root@mastera0 ~]# mysql -uroot -puplooking < procedure.mysql
[root@mastera0 ~]# cat procedure.mysql
use test;
delimiter //
create procedure ordertotal(
in onumber int,
in taxable boolean,
out ototal decimal(8,2)
)

begin
declare total decimal(8,2);
declare taxrate int default 6;

select sum(item_price*quantity)
from orderitems
where order_num = onumber
into total;

if taxable then
select total+(total/100*taxrate) into total;
end if;

select total into ototal;
end //
delimiter ;
```

此存储过程有很大的变动。首先,增加了注释(前面放置 `--`)。在存储过程复杂性增加时,这样做特别重要。添加了另外一个参数 `taxable` ,它是一个布尔值(如果要增加税则为真,否则为假)。在存储过程体中,用 `DECLARE` 语句定义了两个局部变量。 `DECLARE` 要求指定变量名和数据类型,它也支持可选的默认值(这个例子中的 `taxrate` 的默认被设置为 6%)。 `SELECT` 语句已经改变,因此其结果存储到 `total` (局部变量)而不是 `ototal` 。 `IF` 语句检查 `taxable` 是否为真,如果为真,则用另一 `SELECT` 语句增加营业税到局部变量 `total` 。最后,用另一 `SELECT` 语句将 `total` (它增加或许不增加营业税)保存到 `ototal` 。

COMMENT 关键字 本例子中的存储过程在 `CREATE PROCEDURE` 语句中包含了一个 `COMMENT` 值。它不是必需的,但如果给出,将在 `SHOW PROCEDURE STATUS` 的结果中显示。

这显然是一个更高级,功能更强的存储过程。为试验它,请用以下两条语句:

```
MariaDB [test]> call ordertotal(20005,0,@total);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select @total;
+--------+
| @total |
+--------+
| 149.87 |
+--------+
1 row in set (0.00 sec)
```

BOOLEAN 值指定为 1 表示真,指定为 0 表示假(实际上,非零值都考虑为真,只有 0 被视为假)。通过给中间的参数指定 0 或 1 ,可以有条件地将营业税加到订单合计上。

IF 语句 这个例子给出了MySQL的 IF 语句的基本用法。 IF 语句还支持 ELSEIF 和 ELSE 子句(前者还使用 THEN 子句,后者不使用)。在以后章节中我们将会看到 IF 的其他用法(以及其他流控制语句)。

6.检查存储过程

> 为显示用来创建一个存储过程的 CREATE 语句,使用 SHOW CREATE PROCEDURE 语句:

```
MariaDB [test]> show create procedure ordertotal\G;
*************************** 1. row ***************************
           Procedure: ordertotal
            sql_mode:
    Create Procedure: CREATE DEFINER=`root`@`localhost` PROCEDURE `ordertotal`(
in onumber int,
in taxable boolean,
out ototal decimal(8,2))
    COMMENT 'Obtain order total, optionally adding tax'
begin
declare total decimal(8,2);
declare taxrate int default 6;
select sum(item_price*quantity)
from orderitems
where order_num = onumber
into total;
if taxable then
select total+(total/100*taxrate) into total;
end if;
select total into ototal;
end
character_set_client: utf8
collation_connection: utf8_general_ci
  Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

为了获得包括何时、由谁创建等详细信息的存储过程列表,使用 `SHOWPROCEDURE STATUS` 。

限制过程状态结果 `SHOW PROCEDURE STATUS` 列出所有存储过程。为限制其输出,可使用 `LIKE` 指定一个过滤模式,例如:`show procedure status like 'ordertotal';`

> 小结
>
> > 本章介绍了什么是存储过程以及为什么要使用存储过程。我们介绍了存储过程的执行和创建的语法以及使用存储过程的一些方法。下一章我们将继续这个话题。

使用游标

本章将讲授什么是游标以及如何使用游标。

> 游标

需要**MySQL 5** MySQL 5添加了对游标的支持,因此,本章内容适用于MySQL 5及以后的版本。

由前几章可知,MySQL检索操作返回一组称为结果集的行。这组返回的行都是与SQL语句相匹配的行(零行或多行)。使用简单的 SELECT 语句,例如,没有办法得到第一行、下一行或前10行,也不存在每次一行地处理所有行的简单方法(相对于成批地处理它们)。

有时,需要在检索出来的行中前进或后退一行或多行。这就是使用游标的原因。游标(cursor)是一个存储在MySQL服务器上的数据库查询,它不是一条 SELECT 语句,而是被该语句检索出来的结果集。在存储了游标之后,应用程序可以根据需要滚动或浏览其中的数据。

游标主要用于交互式应用,其中用户需要滚动屏幕上的数据,并对数据进行浏览或做出更改。

只能用于存储过程 不像多数DBMS,MySQL游标只能用于存储过程(和函数)。

> 使用游标

使用游标涉及几个明确的步骤。

- 在能够使用游标前,必须声明(定义)它。这个过程实际上没有检索数据,它只是定义要使用的 SELECT 语句。
- 一旦声明后,必须打开游标以供使用。这个过程用前面定义的SELECT 语句把数据实际检索出来。
- 对于填有数据的游标,根据需要取出(检索)各行。
- 在结束游标使用时,必须关闭游标。

在声明游标后,可根据需要频繁地打开和关闭游标。在游标打开后,可根据需要频繁地执行取操作。

1.创建游标

游标用 DECLARE 语句创建。 DECLARE 命名游标,并定义相应的 SELECT 语句,根据需要带 WHERE 和其他子句。

> 定义名为 ordernumbers 的游标,使用了可以检索所有订单的 SELECT 语句。

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure processorders()
    -> begin
    -> declare ordernumbers cursor
    -> for
    -> select order_num from orders;
    -> end //
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

这个存储过程并没有做很多事情, `DECLARE` 语句用来定义和命名游标,这里为 `ordernumbers` 。 存储过程处理完成后,游标就消失(因为它局限于存储过程)。

在定义游标之后,可以打开它。

2.打开和关闭游标

> 游标用 OPEN CURSOR 语句来打开:`open ordernumbers;`

在处理 OPEN 语句时执行查询,存储检索出的数据以供浏览和滚动。

> 游标处理完成后,应当使用如下语句关闭游标:`close ordernumbers;`

CLOSE 释放游标使用的所有内部内存和资源,因此在每个游标不再需要时都应该关闭。

在一个游标关闭后,如果没有重新打开,则不能使用它。但是,使用声明过的游标不需要再次声明,用 OPEN 语句打开它就可以了。

隐含关闭 如果你不明确关闭游标, MySQL将会在到达 END 语句时自动关闭它。

下面是前面例子的修改版本:

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure processorders()
    -> begin
    -> declare ordernumbers cursor
    -> for
    -> select order_num from orders;
    -> open ordernumbers;
    -> close ordernumbers;
    -> end //
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

这个存储过程声明、打开和关闭一个游标。但对检索出的数据什么也没做。

3.使用游标数据

在一个游标被打开后,可以使用 FETCH 语句分别访问它的每一行。FETCH 指定检索什么数据(所需的列),检索出来的数据存储在什么地方。它还向前移动游标中的内部行指针,使下一条 FETCH 语句检索下一行(不重复读取同一行)。

> 第一个例子从游标中检索单个行(第一行):

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure processorders() begin declare o int;declare ordernumbers cursor
for select order_num from orders;
    -> open ordernumbers;
    -> fetch ordernumbers into o;
    -> close ordernumbers;
    -> end //
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

其中 `FETCH` 用来检索当前行的 `order_num` 列(将自动从第一行开始)到一个名为 `o` 的局部声明的变量中。对检索出的数据不做任何处理。

> 在下一个例子中,循环检索数据,从第一行到最后一行:

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure processorders() begin declare done boolean default 0;declare o
int;declare ordernumbers cursor for select order_num from orders;declare continue handler for
sqlstate '02000' set done=1; open ordernumbers;repeat fetch ordernumbers into o;until done end
repeat; close ordernumbers; end//
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

与前一个例子一样,这个例子使用 `FETCH` 检索当前 `order_num` 到声明的名为 `o` 的变量中。但与前一个例子不一样的是,这个例子中的 `FETCH` 是在 `REPEAT` 内,因此它反复执行直到 `done` 为真(由 `UNTIL done END REPEAT;` 规定)。

为使它起作用,用一个 `DEFAULT 0`(假,不结束)定义变量 `done` 。那么, `done` 怎样才能在结束时被设置为真呢?

答案是用以下语句:`declare conitinue handler for sqlstate '02000' set done=1`这条语句定义了一个C`ONTINUE HANDLER` ,它是在条件出现时被执行的代码。这里,它指出当 `SQLSTATE '02000'` 出现时, `SET done=1` 。 `SQLSTATE '02000'` 是一个未找到条件,当 `REPEAT` 由于没有更多的行供循环而不能继续时,出现这个条件。

MySQL的错误代码 关于MySQL 5使用的MySQL错误代码列表,请参阅http://dev.mysql.com/doc/mysql/en/error-handling.html。

DECLARE 语句的次序 DECLARE 语句的发布存在特定的次序。用 DECLARE 语句定义的局部变量必须在定义任意游标或句柄之前定义,而句柄必须在游标之后定义。不遵守此顺序将产生错误消息。

如果调用这个存储过程,它将定义几个变量和一个 `CONTINUE HANDLER` ,定义并打开一个游标,重复读取所有行,然后关闭游标。如果一切正常,你可以在循环内放入任意需要的处理(在 FETCH 语句之后,循环结束之前)。

重复或循环**?** 除这里使用的 `REPEAT` 语句外,MySQL还支持循环语句,它可用来重复执行代码,直到使用 `LEAVE` 语句手动退出为止。通常 `REPEAT` 语句的语法使它更适合于对游标进行循环。

为了把这些内容组织起来,下面给出我们的游标存储过程样例的更进一步修改的版本,这次对取出的数据进行某种实际的处理:

```
MariaDB [test]> delimiter //
MariaDB [test]> create procedure processorders() begin declare done boolean default 0;declare o
int;declare t decimal(8,2);declare ordernumbers cursor for select order_num from orders;declare
continue handler for sqlstate '02000' set done=1; create table if not exists ordertotals
(order_num int,total decimal(8,2)); open ordernumbers;repeat fetch ordernumbers into o;call
ordertotal(o,1,t);insert into ordertotals(order_num,total) values (o,t);until done end repeat;
close ordernumbers; end//
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delimiter ;
```

在这个例子中,我们增加了另一个名为 `t` 的变量(存储每个订单的合计)。此存储过程还在运行中创建了一个新表(如果它不存在的话),名为 `ordertotals` 。这个表将保存存储过程生成的结果。 `FETCH` 像以前一样取每个 `order_num` ,然后用 `CALL` 执行另一个存储过程(我们在前一章中创建)来计算每个订单的带税的合计(结果存储到 `t`)。最后,用 `INSERT` 保存每个订单的订单号和合计。

此存储过程不返回数据,但它能够创建和填充另一个表,可以用一条简单的 SELECT 语句查看该表:

```
MariaDB [test]> call processorders();
Query OK, 1 row affected (0.48 sec)

MariaDB [test]> select * from ordertotals;
+-----------+---------+
| order_num | total   |
+-----------+---------+
|     20005 |  158.86 |
|     20009 |   40.78 |
|     20006 |   58.30 |
|     20007 | 1060.00 |
|     20008 |  132.50 |
|     20008 |  132.50 |
+-----------+---------+
6 rows in set (0.00 sec)
```

这样,我们就得到了存储过程、游标、逐行处理以及存储过程调用其他存储过程的一个完整的工作样例。

> 小结
>
> > 本章介绍了什么是游标以及为什么要使用游标,举了演示基本游标使用的例子,并且讲解了对游标结果进行循环以及逐行处理的技术。

使用触发器

本章学习什么是触发器,为什么要使用触发器以及如何使用触发器。

本章还介绍创建和使用触发器的语法。

> 触发器

需要**MySQL 5** 对触发器的支持是在MySQL 5中增加的。因此,本章内容适用于MySQL 5或之后的版本。

MySQL语句在需要时被执行,存储过程也是如此。但是,如果你想要某条语句(或某些语句)在事件发生时自动执行,怎么办呢?例如:

- 每当增加一个顾客到某个数据库表时,都检查其电话号码格式是否正确,州的缩写是否为大写;
- 每当订购一个产品时,都从库存数量中减去订购的数量;
- 无论何时删除一行,都在某个存档表中保留一个副本。

所有这些例子的共同之处是它们都需要在某个表发生更改时自动处理。这确切地说就是触发器。触发器是MySQL响应以下任意语句而自动执行的一条MySQL语句(或位于 BEGIN 和 END 语句之间的一组语句):

- DELETE ;
- INSERT ;
- UPDATE 。

其他MySQL语句不支持触发器。

> 创建触发器

在创建触发器时,需要给出4条信息:

- 唯一的触发器名;

- 触发器关联的表;
- 触发器应该响应的活动(DELETE 、 INSERT 或 UPDATE);
- 触发器何时执行(处理之前或之后)。

保持每个数据库的触发器名唯一 在MySQL 5中,触发器名必须在每个表中唯一,但不是在每个数据库中唯一。这表示同一数据库中的两个表可具有相同名字的触发器。这在其他每个数据库触发器名必须唯一的DBMS中是不允许的,而且以后的MySQL版本很可能会使命名规则更为严格。因此,现在最好是在数据库范围内使用唯一的触发器名。

触发器用 CREATE TRIGGER 语句创建。下面是一个简单的例子:

```
MariaDB [test]> create trigger triggervendors after insert on vendors for each row select 'hello'
into @args;
Query OK, 0 rows affected (0.35 sec)

MariaDB [test]> insert into vendors values (2007,'xx',null,null,null,null,null);
Query OK, 1 row affected (0.05 sec)

MariaDB [test]> select @args;
+-------+
| @args |
+-------+
| hello |
+-------+
1 row in set (0.00 sec)
```

`CREATE TRIGGER` 用来创建名为 `triggervendors` 的新触发器。触发器可在一个操作发生之前或之后执行,这里给出了 `AFTER INSERT` ,所以此触发器将在 `INSERT` 语句成功执行后执行。这个触发器还指定`FOR EACH ROW` ,因此代码对每个插入行执行。在这个例子中,文本 `hello` 将对每个插入的行显示一次。为了测试这个触发器,使用 INSERT `语句添加一行或多行到` vendors `中,你将看到对每个成功的插入,显示` hello` 消息。

仅支持表 只有表才支持触发器,视图不支持(临时表也不支持)。

触发器按每个表每个事件每次地定义,每个表每个事件每次只允许一个触发器。因此,每个表最多支持6个触发器(每条 INSERT 、 UPDATE和 DELETE 的之前和之后)。单一触发器不能与多个事件或多个表关联,所以,如果你需要一个对 INSERT 和 UPDATE 操作执行的触发器,则应该定义两个触发器。

触发器失败 如果 BEFORE 触发器失败,则MySQL将不执行请求的操作。此外,如果 BEFORE 触发器或语句本身失败, MySQL将不执行 AFTER 触发器(如果有的话)。

> 删除触发器

现在,删除触发器的语法应该很明显了。为了删除一个触发器,可使用 DROP TRIGGER 语句,如下所示:

```
MariaDB [test]> drop trigger triggervendors;
Query OK, 0 rows affected (0.00 sec)
```

触发器不能更新或覆盖。为了修改一个触发器,必须先删除它,然后再重新创建。

> 使用触发器

在有了前面的基础知识后,我们现在来看所支持的每种触发器类型以及它们的差别。

1.INSERT 触发器

INSERT 触发器在 INSERT 语句执行之前或之后执行。需要知道以下几点:

- 在 INSERT 触发器代码内,可引用一个名为 NEW 的虚拟表,访问被插入的行;
- 在 BEFORE INSERT 触发器中, NEW 中的值也可以被更新(允许更改被插入的值);
- 对于 AUTO_INCREMENT 列, NEW 在 INSERT 执行之前包含 0 ,在 INSERT执行之后包含新的自动生成值。

> 下面举一个例子(一个实际有用的例子)。 AUTO_INCREMENT 列具有MySQL自动赋予的值。之前建议了几种确定新生成值的方法,但下面是一种更好的方法:

```
MariaDB [test]> create trigger neworder after insert on orders for each row select new.order_num
into @hh;
Query OK, 0 rows affected (0.37 sec)
```

此代码创建一个名为 `neworder` 的触发器,它按照 `AFTER INSERTON orders` 执行。在插入一个新订单到`orders`表时,MySQL生成一个新订单号并保存到 `order_num` 中。触发器从 `NEW. order_num` 取得这个值并返回它。此触发器必须按照 `AFTER INSERT` 执行,因为在 `BEFOREINSERT` 语句执行之前,新 `order_num` 还没有生成。对于 `orders` 的每次插入使用这个触发器将总是返回新的订单号。

> 为测试这个触发器,试着插入一下新行,如下所示:

```
MariaDB [test]> insert into orders (order_date,cust_id) values (now(),10001);
Query OK, 1 row affected (0.60 sec)

MariaDB [test]> select @hh;
+-------+
| @hh   |
+-------+
| 20010 |
+-------+
1 row in set (0.00 sec)

MariaDB [test]> select * from orders;
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
|     20006 | 2005-09-12 00:00:00 |   10003 |
|     20007 | 2005-09-30 00:00:00 |   10004 |
|     20008 | 2005-10-03 00:00:00 |   10005 |
|     20009 | 2005-10-08 00:00:00 |   10001 |
|     20010 | 2016-09-21 11:08:47 |   10001 |
+-----------+---------------------+---------+
6 rows in set (0.00 sec)
```

orders包含3个列。order_date 和 cust_id必须给出,order_num 由MySQL自动生成,而现在 order_num 还自动被返回。

BEFORE 或 **AFTER ?** 通常,将 BEFORE 用于数据验证和净化(目的是保证插入表中的数据确实是需要的数据)。本提示也适用于 UPDATE 触发器。

2.DELETE 触发器

DELETE 触发器在 DELETE 语句执行之前或之后执行。需要知道以下两点:

- 在 DELETE 触发器代码内,你可以引用一个名为 OLD 的虚拟表,访问被删除的行;
- OLD 中的值全都是只读的,不能更新。

> 下面的例子演示使用 OLD 保存将要被删除的行到一个存档表中:

```
MariaDB [test]> delimiter //

MariaDB [test]> create trigger deleteorder before delete on orders for each row  begin insert
into archive_orders(order_num,order_date,cust_id) values
(OLD.order_num,OLD.order_date,OLD.cust_id); end//
Query OK, 0 rows affected (0.07 sec)

MariaDB [test]> delimiter ;
```

在任意订单被删除前将执行此触发器。它使用一条 `INSERT`语句将 `OLD` 中的值(要被删除的订单)保存到一个名为 `archive_orders` 的存档表中(为实际使用这个例子,你需要用与 `orders` 相同的列创建一个名为 `archive_orders` 的表)。

使用 `BEFORE DELETE` 触发器的优点(相对于 `AFTER DELETE` 触发器来说)为,如果由于某种原因,订单不能存档, `DELETE` 本身将被放弃。

多语句触发器 正如所见,触发器 `deleteorder` 使用 `BEGIN` 和`END` 语句标记触发器体。这在此例子中并不是必需的,不过也没有害处。使用 `BEGIN END` 块的好处是触发器能容纳多条SQL语句(在 `BEGIN END`块 中一条挨着一条)。

3.UPDATE 触发器

UPDATE 触发器在 UPDATE 语句执行之前或之后执行。需要知道以下几点:

- 在 UPDATE 触发器代码中,你可以引用一个名为 OLD 的虚拟表访问以前(UPDATE 语句前)的值,引用一个名为 NEW 的虚拟表访问新更新的值;
- 在 BEFORE UPDATE 触发器中, NEW 中的值可能也被更新(允许更改将要用于 UPDATE 语句中的值);
- OLD 中的值全都是只读的,不能更新。

下面的例子保证州名缩写总是大写(不管 UPDATE 语句中给出的是大写还是小写):

显然,任何数据净化都需要在 UPDATE 语句之前进行,就像这个例子中一样。

> 每次更新一个行时, NEW.vend_state 中的值(将用来更新表行的值)都用 Upper(NEW.vend_state) 替换。

```
MariaDB [test]> create trigger updatevendor before update on vendors for each row set
New.vend_state = Upper（new.vend_state）;
Query OK, 0 rows affected (0.35 sec)
```

4.关于触发器的进一步介绍

在结束本章之前,我们再介绍一些使用触发器时需要记住的重点。

- 与其他DBMS相比,MySQL 5中支持的触发器相当初级。未来的MySQL版本中有一些改进和增强触发器支持的计划。
- 创建触发器可能需要特殊的安全访问权限,但是,触发器的执行是自动的。如果 INSERT 、 UPDATE 或 DELETE 语句能够执行,则相关的触发器也能执行。
- 应该用触发器来保证数据的一致性(大小写、格式等)。在触发器中执行这种类型的处理的优点是它总是进行这种处理,而且是透明地进行,与客户机应用无关。

- 触发器的一种非常有意义的使用是创建审计跟踪。使用触发器,把更改(如果需要,甚至还有之前和之后的状态)记录到另一个表非常容易。
- 遗憾的是,MySQL触发器中不支持 CALL 语句。这表示不能从触发器内调用存储过程。所需的存储过程代码需要复制到触发器内。

> 小结
>
> > 本章介绍了什么是触发器以及为什么要使用触发器,学习了触发器的类型和何时执行它们,列举了几个用于 INSERT 、 DELETE 和 UPDATE 操作的触发器例子。

管理事务处理

本章介绍什么是事务处理以及如何利用 COMMIT 和 ROLLBACK 语句来管理事务处理。

> 事务处理

并非所有引擎都支持事务处理 MySQL支持几种基本的数据库引擎，并非所有引擎都支持明确的事务处理管理。MyISAM 和 InnoDB 是两种最常使用的引擎。前者不支持明确的事务处理管理,而后者支持。这就是为什么本书中使用的样例表被创建来使用 InnoDB 而不是更经常使用的 MyISAM 的原因。如果你的应用中需要事务处理功能,则一定要使用正确的引擎类型。

事务处理(transaction processing)可以用来维护数据库的完整性,它保证成批的MySQL操作要么完全执行,要么完全不执行。

关系数据库设计把数据存储在多个表中,使数据更容易操纵、维护和重用。不用深究如何以及为什么进行关系数据库设计,在某种程度上说,设计良好的数据库模式都是关联的。

前面章中使用的 `orders` 表就是一个很好的例子。订单存储在 `orders` 和 `orderitems` 两个表中: `orders` 存储实际的订单,而 `orderitems` 存储订购的各项物品。这两个表使用称为主键的唯一ID互相关联。这两个表又与包含客户和产品信息的其他表相关联。

给系统添加订单的过程如下。

(1) 检查数据库中是否存在相应的客户(从 customers 表查询),如果不存在,添加他/她。
(2) 检索客户的ID。
(3) 添加一行到 orders 表,把它与客户ID关联。
(4) 检索 orders 表中赋予的新订单ID。
(5) 对于订购的每个物品在 orderitems 表中添加一行,通过检索出来的ID把它与 orders 表关联(以及通过产品ID与 products 表关联)。

现在,假如由于某种数据库故障(如超出磁盘空间、安全限制、表锁等)阻止了这个过程的完成。数据库中的数据会出现什么情况?

如果故障发生在添加了客户之后, orders 表添加之前,不会有什么问题。某些客户没有订单是完全合法的。在重新执行此过程时,所插入的客户记录将被检索和使用。可以有效地从出故障的地方开始执行此过程。

但是,如果故障发生在 orders 行添加之后, orderitems 行添加之前,怎么办呢?现在,数据库中有一个空订单。

更糟的是,如果系统在添加 orderitems 行之中出现故障。结果是数据库中存在不完整的订单,而且你还不知道。

如何解决这种问题?这里就需要使用事务处理了。事务处理是一种机制,用来管理必须成批执行的MySQL操作,以保证数据库不包含不完整的操作结果。利用事务处理,可以保证一组操作不会中途停止,它们或者作为整体执行,或者完全不执行(除非明确指示)。如果没有错误发生,整组语句提交给(写到)数据库表。如果发生错误,则进行回退(撤销)以恢复数据库到某个已知且安全的状态。

因此,请看相同的例子,这次我们说明过程如何工作。

- 检查数据库中是否存在相应的客户,如果不存在,添加他/她。
- 提交客户信息。
- 检索客户的ID。
- 添加一行到 orders 表。
- 如果在添加行到 orders 表时出现故障,回退。
- 检索 orders 表中赋予的新订单ID。
- 对于订购的每项物品,添加新行到 orderitems 表。
- 如果在添加新行到 orderitems 时出现故障,回退所有添加的orderitems 行和 orders 行。
- 提交订单信息。

在使用事务和事务处理时,有几个关键词汇反复出现。下面是关于事务处理需要知道的几个术语:

- 事务(transaction)指一组SQL语句;
- 回退(rollback)指撤销指定SQL语句的过程;
- 提交(commit)指将未存储的SQL语句结果写入数据库表;
- 保留点(savepoint)指事务处理中设置的临时占位符(place-holder),你可以对它发布回退(与回退整个事务处理不同)。

> 控制事务处理

既然我们已经知道了什么是事务处理,下面讨论事务处理的管理中所涉及的问题。

管理事务处理的关键在于将SQL语句组分解为逻辑块,并明确规定数据何时应该回退,何时不应该回退。

MySQL使用下面的语句来标识事务的开始:`start transaction;`

1.使用 ROLLBACK

MySQL的 ROLLBACK 命令用来回退(撤销)MySQL语句,请看下面的语句:

```
MariaDB [test]> select * from ordertotals;
+-----------+---------+
| order_num | total   |
+-----------+---------+
|     20005 |  158.86 |
|     20009 |   40.78 |
|     20006 |   58.30 |
|     20007 | 1060.00 |
|     20008 |  132.50 |
|     20008 |  132.50 |
+-----------+---------+
6 rows in set (0.00 sec)

MariaDB [test]> start transaction;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delete from ordertotals;
Query OK, 6 rows affected (0.00 sec)

MariaDB [test]> select * from ordertotals;
Empty set (0.00 sec)

MariaDB [test]> rollback;
Query OK, 0 rows affected (0.55 sec)

MariaDB [test]> select * from ordertotals;
+-----------+---------+
| order_num | total   |
+-----------+---------+
|     20005 |  158.86 |
|     20009 |   40.78 |
|     20006 |   58.30 |
|     20007 | 1060.00 |
|     20008 |  132.50 |
|     20008 |  132.50 |
+-----------+---------+
6 rows in set (0.00 sec)
```

这个例子从显示 ordertotals 表的内容开始。首先执行一条 SELECT 以显示该表不为空。然后开始一个事务处理,用一条 DELETE 语句删除 ordertotals 中的所有行。另一条SELECT 语句验证 ordertotals 确实为空。这时用一条 ROLLBACK 语句回退START TRANSACTION 之后的所有语句,最后一条 SELECT 语句显示该表不为空。

显然, ROLLBACK 只能在一个事务处理内使用(在执行一条 STARTTRANSACTION 命令之后)。

哪些语句可以回退**?** 事务处理用来管理 INSERT 、 UPDATE 和DELETE 语句。你不能回退 SELECT 语句。(这样做也没有什么意义。)你不能回退 CREATE 或 DROP 操作。事务处理块中可以使用这两条语句,但如果你执行回退,它们不会被撤销。

2.使用 COMMIT

一般的MySQL语句都是直接针对数据库表执行和编写的。这就是所谓的隐含提交(implicit commit),即提交(写或保存)操作是自动进行的。

但是,在事务处理块中,提交不会隐含地进行。为进行明确的提交,使用 COMMIT 语句,如下所示:

```
MariaDB [test]> start transaction;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delete from orderitems where order_num=20010;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delete from orders where order_num=20010;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> commit;
Query OK, 0 rows affected (0.00 sec)
```

在这个例子中,从系统中完全删除订单 `20010` 。因为涉及更新两个数据库表 `orders` 和 `orderItems` ,所以使用事务处理块来保证订单不被部分删除。最后的 COMMIT 语句仅在不出错时写出更改。如果第一条 DELETE 起作用,但第二条失败,则 DELETE 不会提交(实际上,它是被自动撤销的)。

隐含事务关闭 当 `COMMIT` 或 `ROLLBACK` 语句执行后,事务会自隐含事务关闭动关闭(将来的更改会隐含提交)。

3.使用保留点

简单的 `ROLLBACK` 和 `COMMIT` 语句就可以写入或撤销整个事务处理。但是,只是对简单的事务处理才能这样做,更复杂的事务处理可能需要部分提交或回退。

例如,前面描述的添加订单的过程为一个事务处理。如果发生错误,只需要返回到添加 orders 行之前即可,不需要回退到 customers 表(如果存在的话)。

为了支持回退部分事务处理,必须能在事务处理块中合适的位置放置占位符。这样,如果需要回退,可以回退到某个占位符。

这些占位符称为保留点。为了创建占位符,可如下使用 SAVEPOINT语句:`savepoint delete1;`

每个保留点都取标识它的唯一名字,以便在回退时,MySQL知道要回退到何处。为了回退到本例给出的保留点,可如下进行:`rollback to delete1;`

保留点越多越好 可以在MySQL代码中设置任意多的保留点,越多越好。为什么呢?因为保留点越多,你就越能按自己的意愿灵活地进行回退。

释放保留点 保留点在事务处理完成(执行一条 ROLLBACK 或COMMIT)后自动释放。自MySQL 5以来,也可以用 RELEASESAVEPOINT 明确地释放保留点。

4.更改默认的提交行为

正如所述,默认的MySQL行为是自动提交所有更改。换句话说,任何时候你执行一条MySQL语句,该语句实际上都是针对表执行的,而且所做的更改立即生效。为指示MySQL不自动提交更改,需要使用以下语句:`set autocommit=0;`

`autocommit` 标志决定是否自动提交更改,不管有没有 `COMMIT` 语句。设置 `autocommit` 为 `0` (假)指示MySQL不自动提交更改(直到 `autocommit` 被设置为 `1` 真为止)。

标志为连接专用 `autocommit` 标志是针对每个连接而不是服务器的。

> 查看autocommit的值

```
MariaDB [test]> show variables like '%autocommit%';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

MariaDB [test]> select @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
1 row in set (0.00 sec)
```

> 课堂练习

```
MariaDB [test]> create database db1;
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> create table db1.t1 (id int primary key );
Query OK, 0 rows affected (0.06 sec)

MariaDB [test]> insert into db1.t1 values (1);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)

MariaDB [test]> start transaction;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> delete from db1.t1;
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> savepoint delete1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> insert into db1.t1 values (10);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> savepoint insert1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
+----+
| id |
+----+
| 10 |
+----+
1 row in set (0.00 sec)

MariaDB [test]> rollback to delete1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
Empty set (0.00 sec)

MariaDB [test]> rollback to insert1;
ERROR 1305 (42000): SAVEPOINT insert1 does not exist
MariaDB [test]> rollback to insert1;
ERROR 1305 (42000): SAVEPOINT insert1 does not exist
MariaDB [test]> insert into db1.t1 values (20);
Query OK, 1 row affected (0.00 sec)
```

```
MariaDB [test]> select * from db1.t1;
+----+
| id |
+----+
| 20 |
+----+
1 row in set (0.00 sec)

MariaDB [test]> savepoint in1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> insert into db1.t1 values (30);
Query OK, 1 row affected (0.00 sec)

MariaDB [test]> savepoint in2;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
+----+
| id |
+----+
| 20 |
| 30 |
+----+
2 rows in set (0.00 sec)

MariaDB [test]> rollback to in1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
+----+
| id |
+----+
| 20 |
+----+
1 row in set (0.00 sec)

MariaDB [test]> rollback to in2;
ERROR 1305 (42000): SAVEPOINT in2 does not exist
MariaDB [test]> rollback to delete1;
Query OK, 0 rows affected (0.00 sec)

MariaDB [test]> select * from db1.t1;
Empty set (0.00 sec)
```

注意，当退回到某个保留点时，该保留点之后的保留点就会消失。

> 小结
>
> > 本章介绍了事务处理是必须完整执行的SQL语句块。我们学习了如何使用 COMMIT 和 ROLLBACK 语句对何时写数据,何时撤销进行明确的管理。还学习了如何使用保留点对回退操作提供更强大的控制。

全球化和本地化

本章介绍MySQL处理不同字符集和语言的基础知识。

> 字符集和校对顺序

数据库表被用来存储和检索数据。不同的语言和字符集需要以不同的方式存储和检索。因此,MySQL需要适应不同的字符集(不同的字和字符),适应不同的排序和检索数据的方法。

在讨论多种语言和字符集时,将会遇到以下重要术语:

- 字符集为字母和符号的集合;
- 编码为某个字符集成员的内部表示;
- 校对为规定字符如何比较的指令。

校对为什么重要 排序英文正文很容易,对吗?或许不。考虑词APE、apex和Apple。它们处于正确的排序顺序吗?这有赖于你是否想区分大小写。使用区分大小写的校对顺序,这些词有一种排序方式,使用不区分大小写的校对顺序有另外一种排序方式。这不仅影响排序(如用 ORDER BY 排序数据),还影响搜索(例如,寻找apple的WHERE子句是否能找到APPLE)。在使用诸如法文à或德文ö这样的字符时,情况更复杂,在使用不基于拉丁文的字符集(日文、希伯来文、俄文等)时,情况更为复杂。

在MySQL的正常数据库活动(SELECT 、 INSERT 等)中,不需要操心太多的东西。使用何种字符集和校对的决定在服务器、数据库和表级进行。

> 使用字符集和校对顺序

> > MySQL支持众多的字符集。为查看所支持的字符集完整列表,使用以下语句:

```
MariaDB [test]> show character set;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| utf8mb4  | UTF-8 Unicode               | utf8mb4_general_ci  |      4 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| utf16    | UTF-16 Unicode              | utf16_general_ci    |      4 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| utf32    | UTF-32 Unicode              | utf32_general_ci    |      4 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
| cp932    | SJIS for Windows Japanese   | cp932_japanese_ci   |      2 |
| eucjpms  | UJIS for Windows Japanese   | eucjpms_japanese_ci |      3 |
+----------+-----------------------------+---------------------+--------+
39 rows in set (0.00 sec)
```

这条语句显示所有可用的字符集以及每个字符集的描述和默认校对。

> 为了查看所支持校对的完整列表,使用以下语句:

```
MariaDB [test]> show collation;
+----------------------------+----------+-----+---------+----------+---------+
| Collation                  | Charset  | Id  | Default | Compiled | Sortlen |
+----------------------------+----------+-----+---------+----------+---------+
| big5_chinese_ci            | big5     |   1 | Yes     | Yes      |       1 |
| big5_bin                   | big5     |  84 |         | Yes      |       1 |
| dec8_swedish_ci            | dec8     |   3 | Yes     | Yes      |       1 |
| dec8_bin                   | dec8     |  69 |         | Yes      |       1 |
| cp850_general_ci           | cp850    |   4 | Yes     | Yes      |       1 |
| cp850_bin                  | cp850    |  80 |         | Yes      |       1 |
| hp8_english_ci             | hp8      |   6 | Yes     | Yes      |       1 |
| hp8_bin                    | hp8      |  72 |         | Yes      |       1 |
| koi8r_general_ci           | koi8r    |   7 | Yes     | Yes      |       1 |
| koi8r_bin                  | koi8r    |  74 |         | Yes      |       1 |
| latin1_german1_ci          | latin1   |   5 |         | Yes      |       1 |
| latin1_swedish_ci          | latin1   |   8 | Yes     | Yes      |       1 |
| latin1_danish_ci           | latin1   |  15 |         | Yes      |       1 |
| latin1_german2_ci          | latin1   |  31 |         | Yes      |       2 |
| latin1_bin                 | latin1   |  47 |         | Yes      |       1 |
| latin1_general_ci          | latin1   |  48 |         | Yes      |       1 |
| latin1_general_cs          | latin1   |  49 |         | Yes      |       1 |
| latin1_spanish_ci          | latin1   |  94 |         | Yes      |       1 |
| latin2_czech_cs            | latin2   |   2 |         | Yes      |       4 |
| latin2_general_ci          | latin2   |   9 | Yes     | Yes      |       1 |
| latin2_hungarian_ci        | latin2   |  21 |         | Yes      |       1 |
| latin2_croatian_ci         | latin2   |  27 |         | Yes      |       1 |
| latin2_bin                 | latin2   |  77 |         | Yes      |       1 |
| swe7_swedish_ci            | swe7     |  10 | Yes     | Yes      |       1 |
| swe7_bin                   | swe7     |  82 |         | Yes      |       1 |
| ascii_general_ci           | ascii    |  11 | Yes     | Yes      |       1 |
| ascii_bin                  | ascii    |  65 |         | Yes      |       1 |
| ujis_japanese_ci           | ujis     |  12 | Yes     | Yes      |       1 |
| ujis_bin                   | ujis     |  91 |         | Yes      |       1 |
| sjis_japanese_ci           | sjis     |  13 | Yes     | Yes      |       1 |
| sjis_bin                   | sjis     |  88 |         | Yes      |       1 |
| hebrew_general_ci          | hebrew   |  16 | Yes     | Yes      |       1 |
| hebrew_bin                 | hebrew   |  71 |         | Yes      |       1 |
| tis620_thai_ci             | tis620   |  18 | Yes     | Yes      |       4 |
| tis620_bin                 | tis620   |  89 |         | Yes      |       1 |
| euckr_korean_ci            | euckr    |  19 | Yes     | Yes      |       1 |
| euckr_bin                  | euckr    |  85 |         | Yes      |       1 |
| koi8u_general_ci           | koi8u    |  22 | Yes     | Yes      |       1 |
| koi8u_bin                  | koi8u    |  75 |         | Yes      |       1 |
| gb2312_chinese_ci          | gb2312   |  24 | Yes     | Yes      |       1 |
| gb2312_bin                 | gb2312   |  86 |         | Yes      |       1 |
| greek_general_ci           | greek    |  25 | Yes     | Yes      |       1 |
| greek_bin                  | greek    |  70 |         | Yes      |       1 |
| cp1250_general_ci          | cp1250   |  26 | Yes     | Yes      |       1 |
| cp1250_czech_cs            | cp1250   |  34 |         | Yes      |       2 |
| cp1250_croatian_ci         | cp1250   |  44 |         | Yes      |       1 |
| cp1250_bin                 | cp1250   |  66 |         | Yes      |       1 |
| cp1250_polish_ci           | cp1250   |  99 |         | Yes      |       1 |
| gbk_chinese_ci             | gbk      |  28 | Yes     | Yes      |       1 |
```

gbk_bin	gbk	87		Yes	1
latin5_turkish_ci	latin5	30	Yes	Yes	1
latin5_bin	latin5	78		Yes	1
armscii8_general_ci	armscii8	32	Yes	Yes	1
armscii8_bin	armscii8	64		Yes	1
utf8_general_ci	utf8	33	Yes	Yes	1
utf8_bin	utf8	83		Yes	1
utf8_unicode_ci	utf8	192		Yes	8
utf8_icelandic_ci	utf8	193		Yes	8
utf8_latvian_ci	utf8	194		Yes	8
utf8_romanian_ci	utf8	195		Yes	8
utf8_slovenian_ci	utf8	196		Yes	8
utf8_polish_ci	utf8	197		Yes	8
utf8_estonian_ci	utf8	198		Yes	8
utf8_spanish_ci	utf8	199		Yes	8
utf8_swedish_ci	utf8	200		Yes	8
utf8_turkish_ci	utf8	201		Yes	8
utf8_czech_ci	utf8	202		Yes	8
utf8_danish_ci	utf8	203		Yes	8
utf8_lithuanian_ci	utf8	204		Yes	8
utf8_slovak_ci	utf8	205		Yes	8
utf8_spanish2_ci	utf8	206		Yes	8
utf8_roman_ci	utf8	207		Yes	8
utf8_persian_ci	utf8	208		Yes	8
utf8_esperanto_ci	utf8	209		Yes	8
utf8_hungarian_ci	utf8	210		Yes	8
utf8_sinhala_ci	utf8	211		Yes	8
utf8_croatian_ci	utf8	213		Yes	8
utf8_general_mysql500_ci	utf8	223		Yes	1
ucs2_general_ci	ucs2	35	Yes	Yes	1
ucs2_bin	ucs2	90		Yes	1
ucs2_unicode_ci	ucs2	128		Yes	8
ucs2_icelandic_ci	ucs2	129		Yes	8
ucs2_latvian_ci	ucs2	130		Yes	8
ucs2_romanian_ci	ucs2	131		Yes	8
ucs2_slovenian_ci	ucs2	132		Yes	8
ucs2_polish_ci	ucs2	133		Yes	8
ucs2_estonian_ci	ucs2	134		Yes	8
ucs2_spanish_ci	ucs2	135		Yes	8
ucs2_swedish_ci	ucs2	136		Yes	8
ucs2_turkish_ci	ucs2	137		Yes	8
ucs2_czech_ci	ucs2	138		Yes	8
ucs2_danish_ci	ucs2	139		Yes	8
ucs2_lithuanian_ci	ucs2	140		Yes	8
ucs2_slovak_ci	ucs2	141		Yes	8
ucs2_spanish2_ci	ucs2	142		Yes	8
ucs2_roman_ci	ucs2	143		Yes	8
ucs2_persian_ci	ucs2	144		Yes	8
ucs2_esperanto_ci	ucs2	145		Yes	8
ucs2_hungarian_ci	ucs2	146		Yes	8
ucs2_sinhala_ci	ucs2	147		Yes	8
ucs2_croatian_ci	ucs2	149		Yes	8

ucs2_general_mysql500_ci	ucs2	159		Yes	1

cp866_general_ci	cp866	36	Yes	Yes	1
cp866_bin	cp866	68		Yes	1
keybcs2_general_ci	keybcs2	37	Yes	Yes	1
keybcs2_bin	keybcs2	73		Yes	1
macce_general_ci	macce	38	Yes	Yes	1
macce_bin	macce	43		Yes	1
macroman_general_ci	macroman	39	Yes	Yes	1
macroman_bin	macroman	53		Yes	1
cp852_general_ci	cp852	40	Yes	Yes	1
cp852_bin	cp852	81		Yes	1
latin7_estonian_cs	latin7	20		Yes	1
latin7_general_ci	latin7	41	Yes	Yes	1
latin7_general_cs	latin7	42		Yes	1
latin7_bin	latin7	79		Yes	1
utf8mb4_general_ci	utf8mb4	45	Yes	Yes	1
utf8mb4_bin	utf8mb4	46		Yes	1
utf8mb4_unicode_ci	utf8mb4	224		Yes	8
utf8mb4_icelandic_ci	utf8mb4	225		Yes	8
utf8mb4_latvian_ci	utf8mb4	226		Yes	8
utf8mb4_romanian_ci	utf8mb4	227		Yes	8
utf8mb4_slovenian_ci	utf8mb4	228		Yes	8
utf8mb4_polish_ci	utf8mb4	229		Yes	8
utf8mb4_estonian_ci	utf8mb4	230		Yes	8
utf8mb4_spanish_ci	utf8mb4	231		Yes	8
utf8mb4_swedish_ci	utf8mb4	232		Yes	8
utf8mb4_turkish_ci	utf8mb4	233		Yes	8
utf8mb4_czech_ci	utf8mb4	234		Yes	8
utf8mb4_danish_ci	utf8mb4	235		Yes	8
utf8mb4_lithuanian_ci	utf8mb4	236		Yes	8
utf8mb4_slovak_ci	utf8mb4	237		Yes	8
utf8mb4_spanish2_ci	utf8mb4	238		Yes	8
utf8mb4_roman_ci	utf8mb4	239		Yes	8
utf8mb4_persian_ci	utf8mb4	240		Yes	8
utf8mb4_esperanto_ci	utf8mb4	241		Yes	8
utf8mb4_hungarian_ci	utf8mb4	242		Yes	8
utf8mb4_sinhala_ci	utf8mb4	243		Yes	8
utf8mb4_croatian_ci	utf8mb4	245		Yes	8
cp1251_bulgarian_ci	cp1251	14		Yes	1
cp1251_ukrainian_ci	cp1251	23		Yes	1
cp1251_bin	cp1251	50		Yes	1
cp1251_general_ci	cp1251	51	Yes	Yes	1
cp1251_general_cs	cp1251	52		Yes	1
utf16_general_ci	utf16	54	Yes	Yes	1
utf16_bin	utf16	55		Yes	1
utf16_unicode_ci	utf16	101		Yes	8
utf16_icelandic_ci	utf16	102		Yes	8
utf16_latvian_ci	utf16	103		Yes	8
utf16_romanian_ci	utf16	104		Yes	8
utf16_slovenian_ci	utf16	105		Yes	8
utf16_polish_ci	utf16	106		Yes	8
utf16_estonian_ci	utf16	107		Yes	8
utf16_spanish_ci	utf16	108		Yes	8
utf16_swedish_ci	utf16	109		Yes	8

```
| utf16_turkish_ci         | utf16    | 110 |         | Yes      |       8 |
| utf16_czech_ci           | utf16    | 111 |         | Yes      |       8 |
| utf16_danish_ci          | utf16    | 112 |         | Yes      |       8 |
| utf16_lithuanian_ci      | utf16    | 113 |         | Yes      |       8 |
| utf16_slovak_ci          | utf16    | 114 |         | Yes      |       8 |
| utf16_spanish2_ci        | utf16    | 115 |         | Yes      |       8 |
| utf16_roman_ci           | utf16    | 116 |         | Yes      |       8 |
| utf16_persian_ci         | utf16    | 117 |         | Yes      |       8 |
| utf16_esperanto_ci       | utf16    | 118 |         | Yes      |       8 |
| utf16_hungarian_ci       | utf16    | 119 |         | Yes      |       8 |
| utf16_sinhala_ci         | utf16    | 120 |         | Yes      |       8 |
| utf16_croatian_ci        | utf16    | 215 |         | Yes      |       8 |
| cp1256_general_ci        | cp1256   |  57 | Yes     | Yes      |       1 |
| cp1256_bin               | cp1256   |  67 |         | Yes      |       1 |
| cp1257_lithuanian_ci     | cp1257   |  29 |         | Yes      |       1 |
| cp1257_bin               | cp1257   |  58 |         | Yes      |       1 |
| cp1257_general_ci        | cp1257   |  59 | Yes     | Yes      |       1 |
| utf32_general_ci         | utf32    |  60 | Yes     | Yes      |       1 |
| utf32_bin                | utf32    |  61 |         | Yes      |       1 |
| utf32_unicode_ci         | utf32    | 160 |         | Yes      |       8 |
| utf32_icelandic_ci       | utf32    | 161 |         | Yes      |       8 |
| utf32_latvian_ci         | utf32    | 162 |         | Yes      |       8 |
| utf32_romanian_ci        | utf32    | 163 |         | Yes      |       8 |
| utf32_slovenian_ci       | utf32    | 164 |         | Yes      |       8 |
| utf32_polish_ci          | utf32    | 165 |         | Yes      |       8 |
| utf32_estonian_ci        | utf32    | 166 |         | Yes      |       8 |
| utf32_spanish_ci         | utf32    | 167 |         | Yes      |       8 |
| utf32_swedish_ci         | utf32    | 168 |         | Yes      |       8 |
| utf32_turkish_ci         | utf32    | 169 |         | Yes      |       8 |
| utf32_czech_ci           | utf32    | 170 |         | Yes      |       8 |
| utf32_danish_ci          | utf32    | 171 |         | Yes      |       8 |
| utf32_lithuanian_ci      | utf32    | 172 |         | Yes      |       8 |
| utf32_slovak_ci          | utf32    | 173 |         | Yes      |       8 |
| utf32_spanish2_ci        | utf32    | 174 |         | Yes      |       8 |
| utf32_roman_ci           | utf32    | 175 |         | Yes      |       8 |
| utf32_persian_ci         | utf32    | 176 |         | Yes      |       8 |
| utf32_esperanto_ci       | utf32    | 177 |         | Yes      |       8 |
| utf32_hungarian_ci       | utf32    | 178 |         | Yes      |       8 |
| utf32_sinhala_ci         | utf32    | 179 |         | Yes      |       8 |
| utf32_croatian_ci        | utf32    | 214 |         | Yes      |       8 |
| binary                   | binary   |  63 | Yes     | Yes      |       1 |
| geostd8_general_ci       | geostd8  |  92 | Yes     | Yes      |       1 |
| geostd8_bin              | geostd8  |  93 |         | Yes      |       1 |
| cp932_japanese_ci        | cp932    |  95 | Yes     | Yes      |       1 |
| cp932_bin                | cp932    |  96 |         | Yes      |       1 |
| eucjpms_japanese_ci      | eucjpms  |  97 | Yes     | Yes      |       1 |
| eucjpms_bin              | eucjpms  |  98 |         | Yes      |       1 |
+--------------------------+----------+-----+---------+----------+---------+
202 rows in set (0.00 sec)

MariaDB [test]> show collation like '%latin1%';
+-------------------+---------+----+---------+----------+---------+

| Collation         | Charset | Id | Default | Compiled | Sortlen |
```

```
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         | Yes      |       1 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       1 |
| latin1_danish_ci  | latin1  | 15 |         | Yes      |       1 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       1 |
| latin1_general_ci | latin1  | 48 |         | Yes      |       1 |
| latin1_general_cs | latin1  | 49 |         | Yes      |       1 |
| latin1_spanish_ci | latin1  | 94 |         | Yes      |       1 |
+-------------------+---------+----+---------+----------+---------+
8 rows in set (0.00 sec)
```

此语句显示所有可用的校对,以及它们适用的字符集。可以看到有的字符集具有不止一种校对。例如, `latin1` 对不同的欧洲语言有几种校对,而且许多校对出现两次,一次区分大小写(由 `_cs` 表示),一次不区分大小写(由 `_ci` 表示)。

通常系统管理在安装时定义一个默认的字符集和校对。此外,也可以在创建数据库时,指定默认的字符集和校对。

> 为了确定所用的字符集和校对,可以使用以下语句:

```
MariaDB [test]> show variables like 'character%';
+--------------------------+----------------------------+
| Variable_name            | Value                      |
+--------------------------+----------------------------+
| character_set_client     | utf8                       |
| character_set_connection | utf8                       |
| character_set_database   | latin1                     |
| character_set_filesystem | binary                     |
| character_set_results    | utf8                       |
| character_set_server     | latin1                     |
| character_set_system     | utf8                       |
| character_sets_dir       | /usr/share/mysql/charsets/ |
+--------------------------+----------------------------+
8 rows in set (0.00 sec)

MariaDB [test]> show variables like 'collation%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | utf8_general_ci   |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+-------------------+
3 rows in set (0.00 sec)
```

实际上,字符集很少是服务器范围(甚至数据库范围)的设置。不同的表,甚至不同的列都可能需要不同的字符集,而且两者都可以在创建表时指定。

> 为了给表指定字符集和校对,可使用带子句的 CREATE TABLE :

```
MariaDB [test]> create table mytable (columnn1 int,colunmn2 varchar(10)) default character set
hebrew collate hebrew_general_ci;
Query OK, 0 rows affected (0.49 sec)
```

此语句创建一个包含两列的表,并且指定一个字符集和一个校对顺序。

这个例子中指定了 CHARACTER SET 和 COLLATE 两者。一般,MySQL如下确定使用什么样的字符集和校对。

- 如果指定 `CHARACTER SET` 和 `COLLATE` 两者,则使用这些值。
- 如果只指定 `CHARACTER SET` ,则使用此字符集及其默认的校对(如 `SHOW CHARACTER SET` 的结果中所示)。
- 如果既不指定 `CHARACTER SET` ,也不指定 `COLLATE` ,则使用数据库默认。

> 除了能指定字符集和校对的表范围外,MySQL还允许对每个列设置它们,如下所示:

```
MariaDB [test]> create table mytable1 (coln1 int,coln2 varchar(10),coln3 varchar(10) character
set latin1 collate latin1_general_ci) default character set hebrew collate hebrew_general_ci;
Query OK, 0 rows affected (0.36 sec)
```

这里对整个表以及一个特定的列指定了 `CHARACTER SET` 和 `COLLATE` 。

如前所述,校对在对用 `ORDER BY` 子句检索出来的数据排序时起重要的作用。如果你需要用与创建表时不同的校对顺序排序特定的 `SELECT` 语句,可以在 `SELECT` 语句自身中进行:

```
MariaDB [test]> create table db1.t2 (name varchar(10) character set latin1 collate
latin1_general_ci);
Query OK, 0 rows affected (0.06 sec)

MariaDB [test]> insert into db1.t2 values ('a'),('b'),('A'),('B');
Query OK, 4 rows affected (0.07 sec)
Records: 4  Duplicates: 0  Warnings: 0

MariaDB [test]> select * from db1.t2;
+------+
| name |
+------+
| a    |
| b    |
| A    |
| B    |
+------+
4 rows in set (0.00 sec)

MariaDB [test]> select * from db1.t2 order by name collate latin1_general_cs;
+------+
| name |
+------+
| A    |
| a    |
| B    |
| b    |
+------+
4 rows in set (0.00 sec)

MariaDB [test]> select * from db1.t2 order by name;
+------+
| name |
+------+
| a    |
| A    |
| b    |
| B    |
+------+
4 rows in set (0.00 sec)
```

此 SELECT 使用 COLLATE 指定一个备用的校对顺序(在这个例子中,为区分大小写的校对)。这显然将会影响到结果排序的次序。

临时区分大小写 上面的 SELECT 语句演示了在通常不区分大小写的表上进行区分大小写搜索的一种技术。当然,反过来也是可以的。

ELECT 的其他 **COLLATE** 子句 除了这里看到的在 `ORDER BY` 子句 中使用以外, `COLLATE` 还可以用于 `GROUP BY` 、 `HAVING` 、聚集函数、别名等。

最后,值得注意的是,如果绝对需要,串可以在字符集之间进行转换。为此,使用 Cast() 或 Convert ()函数。

> 小结

> 本章中,我们学习了字符集和校对的基础知识,还学习了如何对特定的表和列定义字符集和校对,如何在需要时使用备用的校对。

DCL语言

- DCL(Data Control Language)语句:数据控制语句,用于控制不同数据段直接的许可和访问级别的语句。这些语句定义了数据库、表、字段、用户的访问权限和安全级别。主要的语句关键字包括 grant、revoke 等。

安全管理

数据库服务器通常包含关键的数据,确保这些数据的安全和完整需要利用访问控制。本章将学习MySQL的访问控制和用户管理。

> 访问控制

MySQL服务器的安全基础是:用户应该对他们需要的数据具有适当的访问权,既不能多也不能少。换句话说,用户不能对过多的数据具有过多的访问权。

考虑以下内容:

- 多数用户只需要对表进行读和写,但少数用户甚至需要能创建和删除表;
- 某些用户需要读表,但可能不需要更新表;
- 你可能想允许用户添加数据,但不允许他们删除数据;
- 某些用户(管理员)可能需要处理用户账号的权限,但多数用户不需要;
- 你可能想让用户通过存储过程访问数据,但不允许他们直接访问数据;
- 你可能想根据用户登录的地点限制对某些功能的访问。

这些都只是例子,但有助于说明一个重要的事实,即你需要给用户提供他们所需的访问权,且仅提供他们所需的访问权。这就是所谓的访问控制,管理访问控制需要创建和管理用户账号。

使用**MySQL Administrator** MySQL Administrator提供了一个图形用户界面,可用来管理用户及账号权限。MySQL Administrator在内部利用本章介绍的语句,使你能交互地、方便地管理访问控制。

我们知道,为了执行数据库操作,需要登录MySQL。MySQL创建一个名为 `root` 的用户账号,它对整个MySQL服务器具有完全的控制。你可能已经在本书各章的学习中使用 root 进行过登录,在对非现实的数据库试验MySQL时,这样做很好。不过在现实世界的日常工作中,决不能使用 `root` 。应该创建一系列的账号,有的用于管理,有的供用户使用,有的供开发人员使用,等等。

防止无意的错误 重要的是注意到,访问控制的目的不仅仅是防止用户的恶意企图。数据梦魇更为常见的是无意识错误的结果,如错打MySQL语句,在不合适的数据库中操作或其他一些用户错误。通过保证用户不能执行他们不应该执行的语句,访问控制有助于避免这些情况的发生。

不要使用 **root** 应该严肃对待 `root` 登录的使用。仅在绝对需要时使用它(或许在你不能登录其他管理账号时使用)。不应该在日常的MySQL操作中使用 `root` 。

> 管理用户

MySQL用户账号和信息存储在名为 `mysql` 的MySQL数据库中。一般不需要直接访问 `mysql` 数据库和表(你稍后会明白这一点),但有时需要直接访问。需要直接访问它的时机之一是在需要获得所有用户账号列表时。为此,可使用以下代码:

```
MariaDB [test]> use mysql;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [mysql]> select user from user;
+------+
| user |
+------+
| root |
| root |
| root |
|      |
| root |
|      |
| root |
+------+
7 rows in set (0.00 sec)
```

mysql 数据库有一个名为 `user` 的表,它包含所有用户账号。 user表有一个名为 `user` 的列,它存储用户登录名。新安装的服务器可能只有一个用户(如这里所示),过去建立的服务器可能具有很多用户。

用多个客户机进行试验 试验对用户账号和权限进行更改的最好办法是打开多个数据库客户机(如 mysql 命令行实用程序的多个副本),一个作为管理登录,其他作为被测试的用户登录。

1.创建用户账号

> 为了创建一个新用户账号,使用 CREATE USER 语句,如下所示:

```
MariaDB [mysql]> create user superman identified by 'p@$$w0rd';
Query OK, 0 rows affected (0.00 sec)
```

CREATE USER 创建一个新用户账号。在创建用户账号时不一定需要口令,不过这个例子用 `IDENTIFIED BY 'p@$$wOrd'` 给出了一个口令。

如果你再次列出用户账号,将会在输出中看到新账号。指定散列口令 `IDENTIFIED BY` 指定的口令为纯文本, MySQL 将在保存到 `user` 表之前对其进行加密。为了作为散列值指定口令,使用 `IDENTIFIED BY PASSWORD` 。

使用 **GRANT** 或 **INSERT** GRANT 语句(稍后介绍)也可以创建用户账号,但一般来说 `CREATE USER` 是最清楚和最简单的句子。此外,也可以通过直接插入行到 `user` 表来增加用户,不过为安全起见,一般不建议这样做。MySQL用来存储用户账号信息的表(以及表模式等)极为重要,对它们的任何毁坏都可能严重地伤害到MySQL服务器。因此,相对于直接处理来说,最好是用标记和函数来处理这些表。

> 为重新命名一个用户账号,使用 RENAME USER 语句,如下所示:

```
MariaDB [mysql]> rename user superman@'%' to batman;
Query OK, 0 rows affected (0.00 sec)
MariaDB [mysql]> rename user batman@'%' to superman@'%';
Query OK, 0 rows affected (0.00 sec)
```

MySQL 5之前 仅MySQL 5或之后的版本支持 `RENAME USER` 。为了在以前的MySQL中重命名一个用户,可使用 `UPDATE` 直接更新 `user` 表。

2.删除用户账号

> 为了删除一个用户账号(以及相关的权限),使用 DROP USER 语句,如下所示:

```
MariaDB [mysql]> drop user superman;
Query OK, 0 rows affected (0.00 sec)
```

MySQL 5之前 自MySQL 5以来, DROP USER 删除用户账号和所有相关的账号权限。在MySQL 5以前, DROP USER 只能用来删除用户账号,不能删除相关的权限。因此,如果使用旧版本的MySQL,需要先用 REVOKE 删除与账号相关的权限,然后再用 DROP USER 删除账号。

3.设置访问权限

在创建用户账号后,必须接着分配访问权限。新创建的用户账号没有访问权限。它们能登录MySQL,但不能看到数据,不能执行任何数据库操作。

> 为看到赋予用户账号的权限,使用 SHOW GRANTS FOR ,如下所示:

```
MariaDB [mysql]> create user 'wonderwoman'@'172.25.0.12';
Query OK, 0 rows affected (0.00 sec)

MariaDB [mysql]> show grants for 'wonderwoman'@'172.25.0.12';
+-----------------------------------------------------+
| Grants for wonderwoman@172.25.0.12                  |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'wonderwoman'@'172.25.0.12' |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

输出结果显示用户 `wonderwoman` 有一个权限 `USAGE ON *.*` 。`USAGE` 表示根本没有权限(我知道,这不很直观),所以,此结果表示在任意数据库和任意表上对任何东西没有权限。

用户定义为 `user@host` MySQL的权限用用户名和主机名结合定义。如果不指定主机名,则使用默认的主机名 `%` (授予用户访问权限而不管主机名)。

为设置权限,使用 GRANT 语句。 GRANT 要求你至少给出以下信息:

- 要授予的权限;
- 被授予访问权限的数据库或表;
- 用户名。

> 以下例子给出 GRANT 的用法:

```
MariaDB [mysql]> grant select on db1.* to 'wonderwoman'@'172.25.0.12';
Query OK, 0 rows affected (0.00 sec)

MariaDB [mysql]> show grants for 'wonderwoman'@'172.25.0.12';
+-----------------------------------------------------------+
| Grants for wonderwoman@172.25.0.12                        |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO 'wonderwoman'@'172.25.0.12'         |
| GRANT SELECT ON `db1`.* TO 'wonderwoman'@'172.25.0.12' |
+-----------------------------------------------------------+
2 rows in set (0.00 sec)
```

此 `GRANT` 允许用户在 `db1.*` (db1 数据库的所有表)上使用 `SELECT`。通过只授予 `SELECT` 访问权限,用户 `wonderwoman` 对 `db1` 数据库中的所有数据具有只读访问权限。

每个 `GRANT` 添加(或更新)用户的一个权限。MySQL读取所有授权,并根据它们确定权限。

> `GRANT` 的反操作为 `REVOKE` ,用它来撤销特定的权限。下面举一个例子:

```
MariaDB [mysql]> revoke select on db1.* from 'wonderwoman'@'172.25.0.12';
Query OK, 0 rows affected (0.00 sec)

MariaDB [mysql]> show grants for 'wonderwoman'@'172.25.0.12';
+-----------------------------------------------------+
| Grants for wonderwoman@172.25.0.12                  |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'wonderwoman'@'172.25.0.12' |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

这条 `REVOKE` 语句取消刚赋予用户 `wonderwoman` 的 `SELECT` 访问权限。被撤销的访问权限必须存在,否则会出错。

GRANT 和 REVOKE 可在几个层次上控制访问权限:

- 整个服务器,使用 GRANT ALL 和 REVOKE ALL;
- 整个数据库,使用 ON database.*;
- 特定的表,使用 ON database.table;
- 特定的列;
- 特定的存储过程。

下表列出可以授予或撤销的每个权限。

权限	说明
ALL	除GRANT OPTION外的所有权限
ALTER	使用ALTER TABLE
ALTER ROUTINE	使用ALTER PROCEDURE和DROP PROCEDURE
CREATE	使用CREATE TABLE
CREATE ROUTINE	使用CREATE PROCEDURE
CREATE TEMPORARY TABLES	使用CREATE TEMPORARY TABLE
CREATE USER	使用CREATE USER、 DROP USER、 RENAME USER和REVOKE ALL PRIVILEGES
CREATE VIEW	使用CREATE VIEW
DELETE	使用DELETE
DROP	使用DROP TABLE
EXECUTE	使用CALL和存储过程
FILE	使用SELECT INTO OUTFILE和LOAD DATA INFILE
GRANT OPTION	使用GRANT和REVOKE
INDEX	使用CREATE INDEX和DROP INDEX
INSERT	使用INSERT
LOCK TABLES	使用LOCK TABLES
PROCESS	使用SHOW FULL PROCESSLIST
RELOAD	使用FLUSH
REPLICATION CLIENT	服务器位置的访问
REPLICATION SLAVE	由复制从属使用
SELECT	使用SELECT
SHOW DATABASES	使用SHOW DATABASES
SHOW VIEW	使用SHOW CREATE VIEW
SHUTDOWN	使用mysqladmin shutdown(用来关闭MySQL)
SUPER	使用CHANGE MASTER、KILL、LOGS、PURGE、MASTER和SET GLOBAL。还允许mysqladmin调试登录
UPDATE	使用UPDATE
USAGE	无权限权限

USAGE	无访问权限

使用 GRANT 和 REVOKE ,再结合上表中列出的权限,你能对用户可以就你的宝贵数据做什么事情和不能做什么事情具有完全的控制。

未来的授权 在使用 GRANT 和 REVOKE 时,用户账号必须存在,但对所涉及的对象没有这个要求。这允许管理员在创建数据库和表之前设计和实现安全措施。

这样做的副作用是,当某个数据库或表被删除时(用 DROP 语句),相关的访问权限仍然存在。而且,如果将来重新创建该数据库或表,这些权限仍然起作用。

简化多次授权 可通过列出各权限并用逗号分隔,将多条GRANT 语句串在一起,如下所示:`grant select,insert on db1.* to 'wonderwoman'@'172.25.0.12';`

4.更改口令

> 为了更改用户口令,可使用 SET PASSWORD 语句。新口令必须如下加密:

```
MariaDB [mysql]> create user bforta;
Query OK, 0 rows affected (0.00 sec)

MariaDB [mysql]> set password for bforta = password('uplooking');
Query OK, 0 rows affected (0.00 sec)
```

SET PASSWORD 更新用户口令。新口令必须传递到 Password() 函数进行加密。

SET PASSWORD 还可以用来设置你自己的口令:`set password = password('uplooking');`

在不指定用户名时, SET PASSWORD 更新当前登录用户的口令。

> 小结
>
> 本章学习了通过赋予用户特殊的权限进行访问控制和保护MySQL服务器。

其他

MySQL语句的语法

> 通过帮助可以查看mysql的语法

```
MariaDB [mysql]> help;

General information about MariaDB can be found at
http://mariadb.org

List of all MySQL commands:
Note that all text commands must be first on line and end with ';'
?         (\?) Synonym for `help'.
clear     (\c) Clear the current input statement.
connect   (\r) Reconnect to the server. Optional arguments are db and host.
delimiter (\d) Set statement delimiter.
edit      (\e) Edit command with $EDITOR.
ego       (\G) Send command to mysql server, display result vertically.
exit      (\q) Exit mysql. Same as quit.
go        (\g) Send command to mysql server.
help      (\h) Display this help.
nopager   (\n) Disable pager, print to stdout.
notee     (\t) Don't write into outfile.
pager     (\P) Set PAGER [to_pager]. Print the query results via PAGER.
print     (\p) Print current command.
prompt    (\R) Change your mysql prompt.
quit      (\q) Quit mysql.
rehash    (#) Rebuild completion hash.
rehash    (#) Rebuild completion hash.
source    (\.) Execute an SQL script file. Takes a file name as an argument.
status    (\s) Get status information from the server.
system    (\!) Execute a system shell command.
tee       (\T) Set outfile [to_outfile]. Append everything into given outfile.
use       (\u) Use another database. Takes database name as argument.
charset   (\C) Switch to another charset. Might be needed for processing binlog with multi-byte
charsets.
warnings  (\W) Show warnings after every statement.
nowarning (\w) Don't show warnings after every statement.

For server side help, type 'help contents'
```

> 查看`create`和`create database`的帮助

```
MariaDB [mysql]> help create;
Many help items for your request exist.
To make a more specific request, please type 'help <item>',
where <item> is one of the following
topics:
   CREATE DATABASE
   CREATE EVENT
   CREATE FUNCTION
   CREATE FUNCTION UDF
   CREATE INDEX
   CREATE PROCEDURE
   CREATE SERVER
   CREATE TABLE
   CREATE TABLESPACE
   CREATE TRIGGER
   CREATE USER
   CREATE VIEW
   SHOW
   SHOW CREATE DATABASE
   SHOW CREATE EVENT
   SHOW CREATE FUNCTION
   SHOW CREATE PROCEDURE
   SHOW CREATE TABLE
   SPATIAL

MariaDB [mysql]> help create database;
Name: 'CREATE DATABASE'
Description:
Syntax:
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification] ...

create_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name

CREATE DATABASE creates a database with the given name. To use this
statement, you need the CREATE privilege for the database. CREATE
SCHEMA is a synonym for CREATE DATABASE.

URL: http://dev.mysql.com/doc/refman/5.5/en/create-database.html
```

下表罗列出常见的DDL、DML、DCL、DQL的用法：

```
DDL     数据库定义语言   create\drop\alter

      create database [dbname];
      drop database [dbname];
      create table [tbname] (col1 type,col2 type,col3....);
      drop table [tbname];

DML     数据库操作语言   insert into\delete from\update
      insert into [tbname] set col1=value,col2=value,col3=value;
      insert into [tbname] values (1,'booboo'),(2,'batman'),(3,'superman');
      insert into [tbname] (name,id) values ();
      delete from [tbname] where id=1 and name='booboo';
      update [tbname] set col='superman' where id=2;

DCL     数据库控制语言   grant revoke
      认证 + 授权
      grant all on *.* to booboo@'172.25.0.11' identified by 'uplooking';
        all 权限
        *.* 库.表
      flush privileges; 刷新授权表
      revoke [权限] on [库].[表] from booboo@'172.25.0.11';


DQL     数据库查询语言   show databases;
      use mysql;
      show tables;
      desc mysql.user;
      select * from db1.t1;
```

MySQL数据类型

数据类型是定义列中可以存储什么数据以及该数据实际怎样存储的基本规则。

数据类型用于以下目的。

- 数据类型允许限制可存储在列中的数据。例如,数值数据类型列只能接受数值。
- 数据类型允许在内部更有效地存储数据。可以用一种比文本串更简洁的格式存储数值和日期时间值。
- 数据类型允许变换排序顺序。如果所有数据都作为串处理,则1位于10之前,而10又位于2之前(串以字典顺序排序,从左边开始比较,一次一个字符)。作为数值数据类型,数值才能正确排序。

在设计表时,应该特别重视所用的数据类型。使用错误的数据类型可能会严重地影响应用程序的功能和性能。更改包含数据的列不是一件小事(而且这样做可能会导致数据丢失)。

本章虽然不是关于数据类型及其如何使用的一个完整的教材,但介绍了MySQL主要的数据类型和用途。

> 串数据类型

最常用的数据类型是串数据类型。它们存储串,如名字、地址、电话号码、邮政编码等。有两种基本的串类型,分别为定长串和变长串。

- 定长串接受长度固定的字符串,其长度是在创建表时指定的。例如,名字列可允许30个字符,而社会安全号列允许11个字符(允许的字符数目中包括两个破折号)。定长列不允许多于指定的字符数目。它们分配的存储空间与指定的一样多。因此,如果串 Ben 存储到30个字符的名字字段,则存储的是30个字符, CHAR 属于定长串类型。
- 变长串存储可变长度的文本。有些变长数据类型具有最大的定长,而有些则是完全变长的。不管是哪种,只有指

定的数据得到保存(额外的数据不保存) TEXT 属于变长串类型。

既然变长数据类型这样灵活,为什么还要使用定长数据类型?回答是因为性能。MySQL处理定长列远比处理变长列快得多。此外,MySQL不允许对变长列(或一个列的可变部分)进行索引。这也会极大地影响性能。

串数据类型	说明
CHAR	1~255个字符的定长串。它的长度必须在创建时指定,否则MySQL假定为CHAR(1)
ENUM	接受最多64 K个串组成的一个预定义集合的某个串
LONGTEXT	与TEXT相同,但最大长度为4 GB
MEDIUMTEXT	与TEXT相同,但最大长度为16 K
SET	接受最多64个串组成的一个预定义集合的零个或多个串
TEXT	最大长度为64 K的变长文本
TINYTEXT	与TEXT相同,但最大长度为255字节
VARCHAR	长度可变,最多不超过255字节。如果在创建时指定为VARCHAR(n),则可存储0到n个字符的变长串(其中n≤255)

使用引号 不管使用何种形式的串数据类型,串值都必须括在引号内(通常单引号更好)。

当数值不是数值时 你可能会认为电话号码和邮政编码应该存储在数值字段中(数值字段只存储数值数据),但是,这样做却是不可取的。如果在数值字段中存储邮政编码01234,则保存的将是数值1234,实际上丢失了一位数字。

需要遵守的基本规则是:如果数值是计算(求和、平均等)中使用的数值,则应该存储在数值数据类型列中。如果作为字符串(可能只包含数字)使用,则应该保存在串数据类型列中。

> 数值数据类型

数值数据类型存储数值。MySQL支持多种数值数据类型,每种存储的数值具有不同的取值范围。显然,支持的取值范围越大,所需存储空间越多。此外,有的数值数据类型支持使用十进制小数点(和小数),而有的则只支持整数。下表列出了常用的MySQL数值数据类型。

有符号或无符号 所有数值数据类型(除 `BIT` 和 `BOOLEAN` 外)都可以有符号或无符号。有符号数值列可以存储正或负的数值,无符号数值列只能存储正数。默认情况为有符号,但如果你知道自己不需要存储负值,可以使用 `UNSIGNED` 关键字,这样做将允许你存储两倍大小的值。

数值数据类型	说明
BIT	位字段,1~64位。(在MySQL 5之前,BIT在功能上等价于TINYINT
BIGINT	整数值,支持□9223372036854775808~9223372036854775807(如果是UNSIGNED,为0~18446744073709551615)的数
BOOLEAN(或BOOL)	布尔标志,或者为0或者为1,主要用于开/关(on/off)标志
DECIMAL(或DEC)	精度可变的浮点值
DOUBLE	双精度浮点值
FLOAT	单精度浮点值
INT(或INTEGER)	整数值,支持□2147483648~2147483647 (如果是UNSIGNED,为0~4294967295)的数
MEDIUMINT	整数值,支持□8388608~8388607(如果是UNSIGNED,为0~16777215)的数
REAL	4字节的浮点值
SMALLINT	整数值,支持32768~32767(如果是UNSIGNED,为0~65535)的数
TINYINT	整数值,支持128~127(如果为UNSIGNED,为0~255)的数

不使用引号 与串不一样,数值不应该括在引号内。

存储货币数据类型 MySQL中没有专门存储货币的数据类型,一般情况下使用 DECIMAL(8, 2)

> 日期和时间数据类型

MySQL使用专门的数据类型来存储日期和时间值。见下表。

日期和时间数据类型	说明
DATE	表示1000-01-01~9999-12-31的日期,格式为YYYY-MM-DD
DATETIME	DATE和TIME的组合
TIMESTAMP	功能和DATETIME相同(但范围较小)
TIME	格式为HH:MM:SS
YEAR	用2位数字表示,范围是70(1970年)~69(2069年),用4位数字表示,范围是1901年~2155年

> 二进制数据类型

二进制数据类型可存储任何数据(甚至包括二进制信息),如图像、多媒体、字处理文档等(见下表）。

二进制数据类型	说明
BLOB	Blob最大长度为64 KB
MEDIUMBLOB	Blob最大长度为16 MB
LONGBLOB	Blob最大长度为4 GB
TINYBLOB	Blob最大长度为255字节

实战项目

> 实战项目1:熟悉SQL语句
> 根据下面的表格练习sql语句（书店的库存数据库）

ID	BOOKNAME	WRITER	BOOKDATE	PRICE	AMOUNT
1	Live with Linux	Tube	2007-1-25	75.00	50
2	Linux inside	Kevin	2008-2-15	83.00	50
3	L.A.M.P	Tom	2008-2-5	82.50	50
4	My way	Jam	2007-12-3	45.25	130
5	Open your heart	July	2007-3-18	35.00	20
6	Pro C	Todd	2007-8-25	85.00	25
7	Thinking in C	John	2006-6-13	65.00	30

在该表格中,编号、书名、作者、进货日期、价格、数量就是不同的列,而每一行保存的则是具体的数据。

建立一个名字为 bookshop 的库,在这个库中建立一张关于书籍库存名字为 reserve 的表,表的结构如图所示。

```
[root@serverg ~]# mysql -uroot -p
Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 5.5.41-MariaDB MariaDB Server
Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
#显示库
MariaDB [(none)]> show databases;
+--------------------+
| Database
|
+--------------------+
| information_schema |
| mysql
|
| performance_schema |
| test
|
+--------------------+
4 rows in set (0.00 sec)
#建库
MariaDB [(none)]> create database db1;
Query OK, 1 row affected (0.00 sec)
#使用库
MariaDB [(none)]> use db1;
Database changed
#删除库
MariaDB [db1]> drop database db1;
Query OK, 0 rows affected (0.02 sec)
MariaDB [(none)]> create database booboo;
Query OK, 1 row affected (0.00 sec)
#使用库
MariaDB [(none)]> use booboo;
Database changed
#建表
MariaDB [booboo]> create table bookshop (
-> id int primary key,
列名为 id 整数型 主建
-> bookname varchar(50) not null,
列名为 bookname 可变长度字符类型最长为 50 字节 不为
空
-> writer varchar(50),
列名为 writer 可变长度字符类型最长为 50 字节
-> bookdate date,
列名为 bookdate 日期型
-> price float,
列名为 price 浮点型
-> amount int
列名为 amount 整数型
-> );
Query OK, 0 rows affected (0.05 sec)

#显示表
```

```
MariaDB [booboo]> show tables ;
+------------------+
| Tables_in_booboo |
+------------------+
| bookshop
|
+------------------+
1 row in set (0.00 sec)
#插入表数据
MariaDB [booboo]> insert into bookshop values (1,"Live with Linux","Tube","2007-1-25",75.00,50);
Query OK, 1 row affected (0.01 sec)
MariaDB [booboo]> insert into bookshop values (2,"Linux inside","Kevin","2008-2-15",83.00,50)
-> ;
Query OK, 1 row affected (0.06 sec)
MariaDB [booboo]> insert into bookshop values
-> (3,"L.A.M.P","Tom","2008-2-5",82.5,50),
-> (4,"My way","Jam","2007-12-3",45.25,130),
-> (5,"Open your heart","July","2007-3-8",35,20),
-> (6,"Pro C","Todd","2007-8-25",85,25),
-> (7,"Thinking in C","John","2006-6-13",65,30);
Query OK, 5 rows affected (0.03 sec)
Records: 5 Duplicates: 0 Warnings: 0
#显示表数据
MariaDB [booboo]> select * from bookshop;
+----+-----------------+--------+------------+-------+--------+
| id | bookname
| writer | bookdate | price | amount |
+----+-----------------+--------+------------+-------+--------+
| 1 | Live with Linux | Tube | 2007-01-25 | 75 | 50 |
| 2 | Linux inside | Kevin | 2008-02-15 | 83 | 50 |
| 3 | L.A.M.P
| Tom | 2008-02-05 | 82.5 | 50 |
| 4 | My way
| Jam | 2007-12-03 | 45.25 | 130 |
| 5 | Open your heart | July | 2007-03-08 | 35 | 20 |
| 6 | Pro C
| Todd | 2007-08-25 | 85 | 25 |
| 7 | Thinking in C | John | 2006-06-13 | 65 | 30 |
+----+-----------------+--------+------------+-------+--------+
7 rows in set (0.00 sec)
#查询表数据
MariaDB [booboo]> select bookname,writer from bookshop where id=4;
+----------+--------+
| bookname | writer |
+----------+--------+
| My way | Jam |
+----------+--------+
1 row in set (0.00 sec)
MariaDB [booboo]> select bookname,writer from bookshop where id=4 or id=5;
+-----------------+--------+
| bookname
| writer |

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

```
| My way
| Jam |
| Open your heart | July |
+-----------------+--------+
2 rows in set (0.00 sec)
#修改表数据
MariaDB [booboo]> update bookshop set
-> writer="booboo"
-> where id=1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1 Changed: 1 Warnings: 0
#删除表数据
MariaDB [booboo]> delete from bookshop where id=7;
Query OK, 1 row affected (0.02 sec)
MariaDB [booboo]> select * from bookshop;
+----+-----------------+--------+------------+-------+--------+
| id | bookname
| writer | bookdate | price | amount |
+----+-----------------+--------+------------+-------+--------+
| 1 | Live with Linux | booboo | 2007-01-25 | 75 | 50 |
| 2 | Linux inside | Kevin | 2008-02-15 | 83 | 50 |
| 3 | L.A.M.P
| Tom | 2008-02-05 | 82.5 | 50 |
| 4 | My way
| Jam | 2007-12-03 | 45.25 | 130 |
| 5 | Open your heart | July | 2007-03-08 | 35 | 20 |
| 6 | Pro C
| Todd | 2007-08-25 | 85 | 25 |
+----+-----------------+--------+------------+-------+--------+
6 rows in set (0.00 sec)
```

> 实战项目2:熟悉mysql.user表

1. 查看mysql库中user表中的host,user,password列的值；
2. 删除mysql库中的user表中，user列为空或者password列为空的行；

```
MariaDB [mysql]> select host,user,password from mysql.user;
+----------------------+------+-------------------------------------------+
| host                 | user | password                                  |
+----------------------+------+-------------------------------------------+
| localhost            | root | *6FF883623B8639D08083FF411D20E6856EB7D2BF |
| mastera0.example.com | root |                                           |
| 127.0.0.1            | root |                                           |
| ::1                  | root |                                           |
| localhost            |      |                                           |
| mastera0.example.com |      |                                           |
+----------------------+------+-------------------------------------------+
6 rows in set (0.00 sec)

MariaDB [mysql]> delete from mysql.user where user=' ' or password=' ';
Query OK, 5 rows affected (0.00 sec)

MariaDB [mysql]> select host,user,password from mysql.user;
+-----------+------+-------------------------------------------+
| host      | user | password                                  |
+-----------+------+-------------------------------------------+
| localhost | root | *6FF883623B8639D08083FF411D20E6856EB7D2BF |
+-----------+------+-------------------------------------------+
1 row in set (0.00 sec)
```

> 实战项目3：完成数据库用户权限操作项目
>
> > 要求:添加授权用户测试客户机优先使用的哪个密码(X为学生机号)

1. user1@'172.25.X.%' uplooking
2. user1@'172.25.X.12' uplooking123

```
# mastera:

MariaDB [mysql]> grant all on *.* to user1@"172.25.0.%" identified by "uplooking";
Query OK, 0 rows affected (0.00 sec)
MariaDB [mysql]> grant all on *.* to user1@"172.25.0.12" identified by "uplooking123";
Query OK, 0 rows affected (0.00 sec)
MariaDB [mysql]> flush privileges;
Query OK, 0 rows affected (0.01 sec)
MariaDB [mysql]> \q
Bye

# masterb:

# 第一次输入密码为: uplooking
[root@serverb ~]# mysql -uuser1 -h172.25.0.11 -p'uplooking'
ERROR 1045 (28000): Access denied for user 'user1'@'mastera0.example.com' (using password:YES)
#第二次输入密码为:uplooking123
[root@serverb ~]# mysql -uuser1 -h172.25.0.11 -p'uplooking123'
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 7
Server version: 5.5.41-MariaDB MariaDB Server
Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MariaDB [(none)]>
```

实验结论:uplooking2 密码进去了 授权越精确越优先

> 实战项目4: 破解 MariaDB 5.5 的 root 密码

1. 停止服务 `systemctl stop mariadb`
2. 跳过授权表启动服务 `mysqld_safe --skip-grant-tables &`
3. 修改root密码 `update mysql.user set password=password('uplooking') where user='root';`
4. 停止跳过授权表启动服务 `kill -9`
5. 启动服务 `systemctl start mariadb`

```
# rhel7 mariadb5.5
[root@serverg ~]# systemctl stop mariadb
[root@serverg ~]# mysqld_safe --skip-grant-tables &
[1] 3078
[root@serverg ~]# 160304 18:36:15 mysqld_safe Logging to '/var/log/mariadb/mariadb.log'.
160304 18:36:15 mysqld_safe Starting mysqld daemon with databases from /var/lib/mysql
[root@serverg ~]# mysql -uxxx
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 1
Server version: 5.5.41-MariaDB MariaDB Server
Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MariaDB [(none)]> use mysql;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
MariaDB [mysql]> update user set password=password("redhat") where user="root" and
host="localhost";
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
MariaDB [mysql]> \q
Bye
[root@serverg ~]# ps -ef |grep mysql
mysql
3221
1 0 18:36 ?
00:00:00 /usr/libexec/mysqld --basedir=/usr
--datadir=/var/lib/mysql --plugin-dir=/usr/lib64/mysql/plugin --user=mysql --skip-grant-tables --
log-
error=/var/log/mariadb/mariadb.log
--pid-file=/var/run/mariadb/mariadb.pid
--socket=/var/lib/mysql/mysql.sock
root
3287 3256 0 18:40 pts/0 00:00:00 grep --color=auto mysql
[root@serverg ~]# kill -9 3221
[root@serverg ~]# systemctl start mariadb
[root@serverg ~]# mysql -uroot -predhat
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 5.5.41-MariaDB MariaDB Server
Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MariaDB [(none)]>
```

总结

本章要求掌握sql语句的基本用法，包括`create database`,`create table`,`drop database`,`drop table`,`insert into`,`update`,`delete from`,`grant`,`revoke`;其他sql语句作为拓展。