第2章　用 MySQL 处理数据

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

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

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

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

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

INSERT 语句完成下列操作，这些操作全都涉及数据类型：

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

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

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

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

附录B和附录C提供了 MySQL 列类型、运算和函数的更多信息。

2.1 MySQL 数据类型

MySQL 有几种数据类型，下面分别进行介绍。

1. 数值值

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

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

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

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

数值前可放一个负号“ - ”以表示负值。

2. （字符）串值

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

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

表2-1 串转义序列

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

要在串中包括一个引号，可有如下三种选择：

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

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

■ 如果串是用另外的引号括起来的，则不需要双写相应引号。如：

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

■ 用反斜杠方式表示；这种方法不去管用来将串括起的是单引号还是双引号。如：

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

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

3. 日期和时间值

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

4. NULL 值

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

2.2 MySQL 的列类型

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

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

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

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

2.2.1 列类型概述

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

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

表2-2 数值列类型

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

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

表2-3 串列类型

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

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

表2-4 日期与时间列类型

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

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

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

定义一个列的语法如下：

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

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

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

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

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

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

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

2.2.2 数值列类型

MySQL 的数值列类型有两种：

- 整型。用于无小数部分的数，如 1、43、-3、0 或 -798432。可对正数表示的数据使用整数列，如磅的近似数、英寸的近似数，银河系行星的数目、家族人数或一个盘子里的细菌数等。
- 浮点数。用于可能具有小数部分的数，如 3.14159、-.00273、-4.78、或 39.3E+4。可将浮点数列类型用于有小数点部分或极大、极小的数。可能会表示为浮点数的值有农作物平均产量、距离、钱数（如物品价格或工资）、失业率或股票价格等等。整型值也可以赋予浮点列，这时将它们表示为小数部分为零的浮点值。

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

CREATE TABLE 语句

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

表2-5 数值列类型的取值范围

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

表2-6 数值列类型的存储需求

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

表2-7 M 与 D 对 DECIMAL(M, D) 取值范围的影响

类 型 说 明	取值范围（MySQL < 3.23）	取值范围（MySQL ≥ 3.23）
DECIMAL(4, 1)	-9.9 到 99.9	-999.9 到 9999.9
DECIMAL(5, 1)	-99.9 到 999.9	-9999.9 到 99999.9
DECIMAL(6, 1)	-999.9 到 9999.9	-99999.9 到 999999.9
DECIMAL(6, 2)	-99.99 到 999.99	-9999.99 到 99999.99
DECIMAL(6, 3)	-9.999 到 99.999	-999.999 到 9999.999

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

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

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

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

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

1. 数值列的类型属性

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

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

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

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

如下所示两个属性只用于整数列：

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

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

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

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

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

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

itiny 和 itiny_u 两列都是 TINYINT列，并且都可取256个值，但是itiny的取值范围为 -128 到 127，而itiny_u 的取值范围为 0 到 255。UNSIGNED 对不取负值的列是非常有用的，如存入人口统计或出席人数的列。如果用常规的有符号列来存储这样的值，那么就只利用了该列类型取值范围的一半。通过使列为 UNSIGNED，能有效地成倍增加其取值范围。如果将列用于序列号，且将它设为 UNSIGNED，则可取原双倍的值。

在指定以上属性之后（它们是专门用于数值列的），可以指定通用属性 NULL 或 NOT

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

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

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

2. 使用序列

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

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

(1) MySQL 3.23 以前的版本中的 AUTO_INCREMENT

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

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

  增大计数器会使序列出现空白，但这个特性也有用。例如创建一个具有 AUTO_INCREMENT 列的表，但希望序列从 1000 而不是 1 开始。则可以用后述的两种办法之一达到此目的。一个办法是插入具有明确序列值 1000 的第一个记录，然后通过插入 NULL 到 AUTO_INCREMENT 列来插入后续的记录。另一个办法是插入 AUTO_INCREMENT 列值为 999 的假记录。然后第一个实际插入的记录将得到一个序列号 1000，这时再将假记录删除。
- 如果将一个不合规定的值插入 AUTO_INCREMENT 列，将会出现难以预料的结果。
- 如果删除了在 AUTO_INCREMENT 列中含有最大值的记录，则此值在下一次产生新值时会再次使用。如果删除了表中的所有记录，则所有值都可以重用；相应的序列重新从1开始。
- REPLACE 语句正常起作用。

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

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

(2) MySQL 3.23 版以后的 AUTO_INCREMENT

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

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

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

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

(3) 使用 AUTO_INCREMENT 应该考虑的问题

在使用 AUTO_INCREMENT 列时，应该记住下列要点：

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

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

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

```
DELETE FROM tbl_name
```

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

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

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

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

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

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

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

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

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

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

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

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

打印时，此脚本给出下列输出：

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

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

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

(4) 不用 AUTO_INCREMENT 生成序列

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

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

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

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

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

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

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

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

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

关于将此序列生成方法用于多个计数器的应用，可参阅第 3章。

2.2.3 串列类型

MySQL 提供了几种存放字符数据的串类型。串常常用于如下这样的值：

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

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

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

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

表2-8 串列类型

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

L 以外所需的额外字节为存放该值的长度所需的字节数。MySQL 通过存储值的内容及其长度来处理可变长度的值。这些额外的字节是无符号整数。请注意，可变长类型的最大长度、此类型所需的额外字节数以及占用相同字节数的无符号整数之间的对应关系。例如，MEDIUMBLOB 值可能最多 2^{24} - 1字节长并需要 3 个字节记录其结果。3 个字节的整数类型 MEDIUMINT 的最大无符号值为 2^{24} - 1。这并非偶然。

1. CHAR 和 VARCHAR 列类型

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

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

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

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

除少数情况外，在同一个表中不能混用 CHAR 和 VARCHAR。MySQL 根据情况甚至会将列从一种类型转换为另一种类型。这样做的原因如下：

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

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

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

如果使用 DESCRIBE my_table 查询，则其输出如下：

```
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| c1    | varchar(10) | YES  |     | NULL    |       |
| c2    | varchar(10) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
```

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

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

BLOB 和 TEXT 列类型像 VARCHAR 一样是可变长的，但是它们没有定长的等价类型，因此不能在同一表中与 BLOB 或 TEXT 列一起使用 CHAR 列。这时任何 CHAR 列都将被转换为 VARCHAR 列。

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

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

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

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

查看 DESCRIBE my_table 的输出可看到转换结果如下：

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

2. BLOB 与 TEXT 列类型

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

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

使用 BLOB 和 TEXT 列需要特别注意以下几点：

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

3. ENUM 和 SET 列类型

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

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

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

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

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

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

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

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

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

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

然后，特定的 SET 值将表示顾客实际订购哪些选件，如下所示：

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

空串表示顾客未订购任何选件。这是一个合法的 SET 值。

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

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

对 ENUM 或 SET 列的合法值列表的定义很重要，例如：

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

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

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

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

```
mysql> CREATE TABLE e_table (e ENUM("jane","fred","will","marcia"));
mysql> INSERT INTO e_table
VALUES("jane"),("fred"),("will"),("marcia"),(""),(NULL);
mysql> SELECT e, e+0, e+1, e*3 FROM e_table;
+--------+------+------+------+
| e      | e+0  | e+1  | e*3  |
+--------+------+------+------+
| jane   |    1 |    2 |    3 |
| fred   |    2 |    3 |    6 |
| will   |    3 |    4 |    9 |
| marcia |    4 |    5 |   12 |
|        |    0 |    1 |    0 |
| NULL   | NULL | NULL | NULL |
+--------+------+------+------+
```

可对 ENUM 成员按名或者按编号进行运算，例如：

```
mysql> SELECT e FROM e_table WHERE e="will";
+------+
| e    |
+------+
| will |
+------+
mysql> SELECT e FROM e_table WHERE e=3;
+------+
| e    |
+------+
| will |
+------+
```

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

```
mysql> CREATE TABLE t (e ENUM("a","","b"));
mysql> INSERT INTO t VALUES("a"),(""),("b"),("x");
mysql> SELECT e, e+0 FROM t;
+------+------+
| e    | e+0  |
+------+------+
| a    |    1 |
|      |    2 |
| b    |    3 |
|      |    0 |
+------+------+
```

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

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

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

```
mysql> CREATE TABLE s_table (s SET("jane","fred","will","marcia);
mysql> INSERT INTO s_table
VALUES("jane"),("fred"),("will"),("marcia"),(""),(NULL);
mysql> SELECT s, s+0, BIN(s+0) FROM s_table;
+--------+------+----------+
| s      | s+0  | BIN(s+0) |
+--------+------+----------+
| jane   |    1 | 1        |
| fred   |    2 | 10       |
| will   |    4 | 100      |
| marcia |    8 | 1000     |
|        |    0 | 0        |
| NULL   | NULL | NULL     |
+--------+------+----------+
```

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

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

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

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

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

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

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

NULL 值排在其他值前（如果是降序，将排在其他值之后）。

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

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

```
mysql> SELECT CONCAT(e) as e_str FROM e_table ORDER BY e_str;
+--------+
| e_str  |
+--------+
| NULL   |
|        |
| fred   |
| jane   |
| marcia |
| will   |
+--------+
```

4. 串列类型属性

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

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

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

2.2.4 日期和时间列类型

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

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

表2-9 日期和时间列类型

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

表2-10 日期和时间列类型的存储需求

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

表2-11 日期和时间列类型的“零”值

类型说明	零 值
DATE	“0000 - 00 - 00”
TIME	“00:00:00”
DATETIME	“0000- 00 - 00 00:00:00”
TIMESTAMP	00000000000000
YEAR	0000

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

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

1. DATE、TIME 和 DATETIME 列类型

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

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

2. TIMESTAMP 列类型

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

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

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

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

表2-12 TIMESTAMP 显示格式

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

TIMESTAMP 列的显示宽度与存储大小或存储在内部的值无关。TIMESTAMP 值总是以 4 字节存放并按 14 位精度进行计算，与显示宽度无关。为了明白这一点，按如下定义一个表，然后插入一些行，进行检索：

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

该 SELECT 语句的输出如下：

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

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

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

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

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

在以后无论何时更改此记录，都要明确地设置此列为其原有的值。赋予一个明确

的值使时间戳机制失效，因为它阻止了该列的值自动更新：

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

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

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

无论以后何时更新此记录，都不能动该列：

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

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

3. YEAR 列类型

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

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

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

表2-13 日期和时间类型的输入格式

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

4. 日期和时间列类型的属性

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

5. 处理日期和时间列

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

形式。表2-13为每种日期和时间类型所允许的格式。

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

```
"2012-02-03 05:04:09"
"2012-2-03 05:04:09"
"2012-2-3 05:04:09"
"2012-2-3 5:04:09"
"2012-2-3 5:4:09"
"2012-2-3 5:4:9"
```

请注意，有前导零的值根据它们被指定为串或数有不同的解释。串“ 001231”将视为一个六位数字的值并解释为 DATE 的“2000-12-31”和 DATETIME 的“2000-12-31 00:00:00”。而数 001231被认为 1231，这样的解释就有问题了。这种情形最好使用串值，或者如果要使用数值的话，应该用完全限定的值（即， DATE 用 20001231，DATETIME 用 200012310000）。

通常，在 DATE、DATETIME 和 TIMESTAMP 类型之间可以自由地赋值，但是应该记住以下一些限制：

- 如果将 DATETIME 或 TIMESTAMP 值赋给 DATE，则时间部分被删除。
- 如果将 DATE 值赋给 DATETIME 或 TIMESTAMP，结果值的时间部分被设置为零。
- 各种类型具有不同的取值范围。 TIMESTAMP 的取值范围更受限制（1970 到 2037），因此，比方说，不能将 1970 年以前的 DATETIME 值赋给 TIMESTAMP 并得到合理的结果。也不能将 2037 以后的值赋给 TIMESTAMP。

MySQL 提供了许多处理日期和时间值的函数。要了解更详细的信息请参阅附录 C。

6. 歧义年份值的理解

对于所有包括年份部分的日期和时间类型（DATE、DATETIME、TIMESTAMP、YEAR），MySQL将两位数字的年份转换为四位数字的年份。这个转换根据下列规则进行（在 MySQL 4.0 中，这些规则稍有改动，其中 69 将转换为 1969 而不是 2069。这是根据 X/Open UNIX 标准规定的规则作出的改动）：

- 00 到 69 的年份值转换为 2000 到 2069。
- 70 到 99 的年份值转换为 1970 到 1999。

通过将不同的两位数字值赋给一个 YEAR 列然后进行检索，可很容易地看到这些规则的效果。下面是检索程序：

```
mysql> CREATE TABLE y_table (y YEAR);
mysql> INSERT INTO y_table VALUES(68),(69),(99), (00);
mysql> SELECT * FROM y_table;
+------+
| y    |
+------+
| 2068 |
| 1969 |
| 1999 |
| 0000 |
+------+
```

请注意，00 转换为 0000 而不是 2000。这是因为 0 是 YEAR 类型的一个完全合法的值；如果插入一个数值，得到的就是这个结果。要得到 2000，应该插入串“ 0 ”或“ 00 ”。可通过 CONCAT( ) 插入 YEAR 值来保证 MySQL 得到一个串而不是数。CONCAT( ) 函数不管其参数是串或数值，都返回一个串结果。

请记住，将两位数字的年份值转换为四位数字的年份值的规则只产生一种结果。在未给定世纪的情况下，MySQL 没有办法肯定两位数字的年份的含义。如果 MySQL 的转换规则不能得出您所希望的值，解决的方法很简单：即用四位数字输入年份值。

> **MySQL 有千年虫问题吗？**
>
> MySQL 自身是没有 2000 年问题的，因为它在内部是按四位数年份存储日期值的，并且由用户负责提供恰当的日期值。两位数字年份解释的实际问题不是 MySQL 带来的，而是由于有的人想省事，输入歧义数据所引起的问题。如果您愿意冒险，可以继续这样做。在您冒险的时候，MySQL 的猜测规则是可以使用的。但要意识到，很多时候您确实需要输入四位数字的年份。例如，president表列出了 1700 年以来的美国总统，所以在此表中录入出生与死亡日期需要四位的年份值。这些列中的年份值跨了好几个世纪，因此，让 MySQL 从两位数字的年份去猜测是哪个世纪是不可能的。

2.3 选择列的类型

上一节描述了各种可供选择的 MySQL 的列类型及其属性，以及它们可存储的各种值，所占用的存储空间等等。但是在实际创建一个表时怎样决定用哪些类型呢？本节讨论在做出决定前应考虑的各种因素。

最“ 常用 ”的列类型是串类型。可将任何数据存储为串，因为数和日期都可以串的形式表示。但是为什么不将所有列都定义为串从而结束这里的讨论呢？让我们来看一个简单的例子。假定有一些看起来像数的值。可将它们表示为串，但应该这样做吗？这样做会发生什么事？

有一桩事不可避免，那就是可能要使用更多的空间，因为较串来说，数的存储更为有效。我们可能已经注意到，由于数和串处理方式的不同，查询结果也有所不同。例如，数的排序与串的排序就有所不同。数 2 小于数 11，但串“ 2 ”按字典顺序大于“ 11 ”。可用如下数值内容的列来搞清这个问题：

```
SELECT col_name + 0 as num ... ORDER BY num
```

将零加到该列强制得出一个数值，但是这样合理吗？一般可能不合理。将该列作为数而不是串具有几个重要的含义。它对每个列值实施串到数的转换，这是低效的。而且将该列的值转换为计算结果妨碍 MySQL 使用该列上的索引，降低了以后的查询速度。如果这些值一开始就是作为数值存储的，那么这些性能上的降低都不会出现。采用一种表示而不用另一种的简单选择实际上并不简单，它在存储需求、查询效率以及处理性能等方面都会产生重要的影响。

前面的例子说明，在选择列类型时，有以下几个问题需要考虑：

- 列中存储何种类型的值？这是一个显而易见的问题，但必须确定。可将任何类型的值表示为串，尤其当对数值使用更为合适的类型可能得到更好的性能时（日期和时间值

也是这样)。可见，对要处理的值的类型进行评估不一定是件微不足道的事，特别在数据是别人的数据时更是如此。如果正在为其他人建立一个表，搞清列中要存储的值的类型极为重要，必须提足够多的问题以便得到作出决定的充足的信息。

- 列值有特定的取值范围吗？如果它们是整数，它们总是非负值吗？如果这样，可采用 UNSIGNED 类型。如果它们是串，总能从定长值集中选出它们吗？如果这样， ENUM 或 SET 是很合适的类型。

  在类型的取值范围与所用的存储量之间存在折衷。需有一个多“大”的类型？对于数，如果其取值范围有限，可以选择较小的类型，对取值范围几乎无限的数，应该选择较大的类型。对于串，可以使它们短也可以使它们长，但如果希望存储的值只含不到 10 个字符，就不应该选用 CHAR(255)。
- 性能与效率问题是什么？有些类型比另外一些类型的处理效率高。数值运算一般比串的运算快。短串比长串运行更快，而且磁盘消耗更小。定长类型比可变长类型的性能更好。
- 希望对值进行什么样的比较？对于串，其比较可以是区分大小写的，也可以不区分大小写。其选择也会影响排序，因为它是基于比较的。
- 计划对列进行索引吗？如果计划对列进行索引，那么将会影响您对列类型的选择，因为有的 MySQL 版本不允许对某些类型进行索引，例如不能对 BLOB 和 TEXT 类型进行索引。而且有的 MySQL 版本要求定义索引列为 NOT NULL 的，这使您不能使用 NULL 值。

现在让我们来更详细地考虑这些问题。这里要指出的是：在创建表时，希望作出尽可能好的列类型选择，但如果所作的选择其实际并不是最佳的，这也不会带来多大的问题。可用 ALTER TABLE 将原来选择的类型转换为更好的类型。在发现数据所含的值比原设想的大时，可像将 SMALLINT 更换成 MEDIUMINT 那样简单地对类型进行更换。有时这种更换也可能很复杂，例如将 CHAR 类型更换成具有特定值集的 ENUM 类型。在 MySQL 3.23 及以后的版本中，可使用 PROCEDURE ANALYSE( ) 来获得表列的信息，诸如最小值和最大值以及推荐的覆盖列中值的取值范围的最佳类型。这有助于确定使用更小的类型，从而改进涉及该表的查询的性能，并减少存储该表所需的空间量。

2.3.1 列中存储何种类型的值

在决定列的类型时，首先应该考虑该列的值类型，因为这对于所选择的类型来说具有最为明显的意义。通常，在数值列中存储数，在串列中存储串，在日期和时间列中存储日期和时间。如果数值有小数部分，那么应该用浮点列类型而不是整数类型，如此等等。有时也存在例外，不可一概而论。主要是为了有意义地选择类型，应该理解所用数据的特性。如果您打算存储自己的数据，大概对如何存储它们会有自己很好的想法。但是，如果其他人请您为他们建一个表，决定列类型有时会很困难。这不像处理自己的数据那么容易。应该充分地提问，搞清表实际应该包含何种类型的值。

如果有人告诉您，某列需要记录“降雨量”。那是一个数吗？或者它“主要”是一个数值，即，一般是但不总是编码成一个数吗？例如，在看电视新闻时，气象预报一般包括降雨量。有时是一个数（如“ 0.25 ”英寸的雨量），但是有时是“微量 (trace) ”降雨，意思是“雨根本

就不大”。这对气象预报很合适，但在数据库中怎样存储？有可能需要将“微量”量化为一个数，以便能用数值列类型来记录降雨量，或许需要使用串，以便可以记录“微量”这个词。或者可以提出某种更为复杂的安排，使用一个数值列和一个串列，如果填充一个列就让另一个列为 NULL。很明显，可能的话，应该避免最后这种选择；最后这种选择使表难于理解，使查询更为困难。

我们一般尽量以数值形式存储所有的行，而且只为了显示的需要才对它们进行转换。例如，如果小于 0.01 英寸的非零降雨量被视为微量，那么可以如下选择列值：

```
SELECT IF(precip>0 AND precip<.01,"trace",precip) FROM ...
```

对于金钱的计算，需要处理元和分部分。这似乎像浮点值，但 FLOAT 和 DOUBLE 容易出现舍入错误，除了只需要大致精确的记录外，这些类型可能不适合。因为人们对自己的钱都是很敏感的，最好是用一种能提供完善的精确性的类型，例如：

- 将钱表示为 DECIMAL(M, 2) 类型，选择 M 为适合于所需取值范围的最大宽度。这给出具有两位小数精度的浮点值。DECIMAL 的优点是将值表示为一个串，而且不容易出现舍入错误。不利之处是串运算比内部存储为数的值上的运算效率差。
- 可在内部用整数类型来表示所有的钱值。其优点是内部用整数来计算，这样会非常快。不利之处是在输入或输出时需要利用乘或除 100 对值进行转换。

有些数据显然是数值的，但必须决定是使用浮点类型还是使用整数类型。应该搞清楚所用的单位是什么以及需要什么样的精度。整个单元的精度都够吗？或者需要表示小数的单元吗？这将有助于您在整数列和浮点数列之间进行区分。例如，如果您正表示权重，那么如果记录的值为英磅，可以使用一个整形列。如果希望记录小数部分，就应该使用浮点列。在有的情况下，甚至会使用多个字段，例如：如果希望根据磅和盎司记录权重，则可以使用多个列。

高度（height）是另外一种数值类型，有如下几种表示方法：

- 诸如“6 英尺 2 英寸”可表示为“6-2”这样一个串。这种形式具有容易察看和理解的优点（当然比“74 英寸更好理解”），但是这种值很难用于数学运算，如求和或取平均值。
- 一个数值字段表示英尺，另一个数值字段表示英寸。这样的表示进行数值运算相对容易，但两个字段比一个字段难于使用。
- 只用一个表示英寸的数值段。这是数据库最容易处理的方式，但是这种方式意义最不明确。不过要记住，不一定要用与您惯常使用的那种格式来表示值。可以用 MySQL 的函数将值转换为看上去意义明显的值。因此，最后这种表示方法可能是表示高度的最好方法。

如果需要存储日期信息，需要包括时间吗？即，它们永远都需要包括时间吗？MySQL 不提供具有可选时间部分的日期类型：DATE 可不包含时间，而 DATETIME 必须包含时间。如果时间确实是可选的，那么可用一个 DATE 列记录日期，一个 TIME 列记录时间。允许 TIME 列为 NULL 并解释为“无时间”：

```
CREATE TABLE my_table
(
  date DATE NOT NULL,
  time TIME NULL
)
```

在用基于日期信息的主-细目关系连接两个表时，决定是否需要时间值特别重要。

假如您正在进行一项研究，包括一些对进入您的办公室的人进行测试的题目。在一个标准的初步测试集之后，您可能会在同一天进行几个额外的测试，测试的选择视初步测试结果而定。您可能会利用一个主-细目关系来表示这些信息，其中题目的标识信息和标准的初步测试存储在一个主记录中，而其他测试保存为辅助细目表的行。然后基于题目 ID 与进行测试的日期将这两个表连接到一起。

在这种情况下必须回答的问题是，是否可以只用日期，或者是否需要既使用日期又使用时间。这个问题依赖于一个题目是否可以在同一天投入测试过程不止一次。如果是这样，那么应该记录时间（比方说，记录测试过程开始的时间），或者用 DATETIME 列，或者分别用 DATE 和 TIME 列（两者都必须填写）。如果一个题目一天测试了两次，没有时间值就不能将该题目的细目记录与适当的主记录进行关联。

我曾经听过有人声称“我不需要时间；我从不在同一天把一道题测试两次”。有时他们是对的，但是我也看到过这些人后来在录入同一天测试多次的题目的数据后，反过来考虑怎样防止细目记录与错误的主记录相混。很抱歉，这时已经太迟了！

有时可以在表中增加 TIME 列来处理这个问题，不幸的是，除非有某些独立的数据源，如原书面记录，否则很难整理现有记录。此外，没办法消除细目记录的歧义，以便将它们关联到合适的主记录上。即使有独立的信息源，这样做也是非常乱的，很可能使已经编写来利用表的应用程序出问题。最好是向表的拥有者说明问题并保证在创建他们的表之前进行很好的描述。

有时具有一些不完整的数据，这会干扰列类型的选择。如果进行家谱研究，需要记录出生日期和死亡日期，有时会发现所能搜集到的数据中只是某人出生或死亡的年份，但没有确切的日期。如果使用 DATE 列，除非有完整的日期值，否则不能输入日期。如果希望能够记录所具有的任何信息，即使不完整也保存，那么可能必须保存独立的年、月、日字段。这样就可以输入所具有的日期成员并将没有的部分设为 NULL。在 MySQL 3.23 及以后的版本中，还允许 DATE 的日为 0 或者月和日部分为 0。这样“模糊”的日期可用来表示不完整的日期值。

2.3.2 列值有特定的取值范围吗

如果已经决定从通用类别上选择一种列类型，那么考虑想要表示的值的取值范围会有助于将您的选择缩减到该类别中特定的类型上。假如希望存储整数值。这些整数值的取值范围为 0 到 1000，那么可以使用从 SMALLINT 到 BIGINT 的所有类型。如果这些整数值的取值范围最多为 2 000 000，则不能使用 SMALLINT，其选择范围从 MEDIUMINT 到 BIGINT。需要从这个可能的选择范围中选取一种类型。

当然，可以简单地为想要存储的值选择最大的类型（如上述例子中选择 BIGINT）。但是，一般应该为所要存储的值选择足以存储它的最小的类型。这样做，可以最小化表占用的存储量，得到最好的性能，因为通常较小列的处理比较大列的快。

如果不知道所要表示的值的取值范围，那么必须进行猜测或使用 BIGINT 以应付最坏的情况。（请注意，如果进行猜测时使用了一个太小的类型，工作不会白做；以后可以利用

ALTER TABLE 来将此列改为更大一些的类型。）

在第1章中，我们为学分保存方案创建了一个 score 表，它有一个记录测验和测试学分的 score 列。为了讨论简单起见，创建该表时使用了 INT 类型，但现在可以看出，如果学分在 0 到 100 的取值范围内，更好的选择应该是 TINYINT UNSIGNED，因为所用的存储空间较小。

数据的取值范围还影响列类型的属性。如果该数据从不为负，可使用 UNSIGNED 属性；否则就不能用它。

串类型没有数值列那样的“取值范围”，但它们有长度，需要知道该串可使用的列最大长度。如果串短于256个字符，可使用 CHAR、VARCHAR、TINYTEXT 或 TINYBLOB 等类型。如果想要更长的串，可使用 TEXT 或 BLOB 类型，而 CHAR 和 VARCHAR 不再是选项。

对于用来表示某个固定值集合的串列，可以考虑使用 ENUM 或 SET 列类型。它们可能是很好的选项，因为它们在内部是用数来表示的。这两个类型上的运算是数值化的，因此，比其他的串类型效率更高。它们还比其他串类型紧凑、节省空间。

在描述必须处理的值的范围时，最好的术语是“总是”和“决不”（如“总是小于 1000”或“决不为负”），因为它们能更准确地约束列类型的选择。但在未确证之前，要慎用这两个术语。特别是与其他人谈他们的数据，而他们开始乱用这两个术语时要注意。在有人说“总是”或“决不”时，一定要搞清他们说的确实是这个含义。有时人们说自己的数据总是有某种特定的性质，而其真正的含义是“几乎总是”。

例如，假如您为某些人设计一个表，而他们告诉您，“我们的测试学分总是 0 到 100”。根据这个描述，您选择了 TINYINT 类型并使它为 UNSIGNED 的，因为值总是非负的。然而，您发现编码录入数据库的人有时用－1 来表示“学生因病缺席”。呀，他们没告诉您这事。可能可以用 NULL 来表示 －1，但如果不能，必须记录－1，这样就不能用 UNSIGNED 列了（只好用 ALTER TABLE 来补救！）。

有时关于这些情形的讨论可通过提一些简单的问题来简化，如问：曾经有过例外吗？如果曾经有过例外情况，即使是只有一次，也必须考虑。您会发现，和您讨论数据库设计的人总是认为，如果例外不经常发生，那么就没什么关系。然而在创建数据库时，就不能这样想了。需要提的问题并不是例外出现有多频繁，而是有没有例外？如果有，必须考虑进去。

2.3.3 性能与效率问题

列类型的选择会在几个方面影响查询性能。如果记住下几节讨论的一般准则，将能够选出有助于 MySQL 有效处理表的列类型。

1. 数值与串的运算

数值运算一般比串运算更快。例如比较运算，可在单一运算中对数进行比较。而串运算涉及几个逐字节的比较，如果串更长的话，这种比较还要多。

如果串列的值数目有限，应该利用 ENUM 或 SET 类型来获得数值运算的优越性。这两种类型在内部是用数表示的，可更为有效地进行处理。

例如替换串的表示。有时可用数来表示串值以改进其性能。例如，为了用点分四位数

（dotted-quad）表示法来表示 IP 号，如 192.168.0.4，可以使用串。但是也可以通过用四字节的 UNSIGNED 类型的每个字节存储四位数的每个部分，将 IP 号转换为整数形式。这即可以节省空间又可加快查找速度。但另一方面，将 IP 号表示为 INT 值会使诸如查找某个子网的号码这样的模式匹配难于完成。因此，不能只考虑空间问题；必须根据利用这些值做什么来决定哪种表示更适合。

2. 更小的类型与更大的类型

更小的类型比更大的类型处理要快得多。首先，它们占用的空间较小，且涉及的磁盘活动开销也少。对于串，其处理时间与串长度直接相关。

一般情况下，较小的表处理更快，因为查询处理需要的磁盘 I/O 少。对于定长类型的列，应该选择最小的类型，只要能存储所需范围的值即可。例如，如果 MEDIUMINT 够用，就不要选择 BIGINT。如果只需要 FLOAT 精度，就不应该选择 DOUBLE。对于可变长类型，也仍然能够节省空间。一个 BLOB 类型的值用 2 字节记录值的长度，而一个 LONGBLOB 则用 4 字节记录其值的长度。如果存储的值长度永远不会超过 64KB，使用 BLOB 将使每个值节省 2 字节（当然，对于 TEXT 类型也可以做类似的考虑）。

3. 定长与可变长类型

定长类型一般比可变长类型处理得更快：

- 对于可变长列，由于记录大小不同，在其上进行许多删除和更改将会使表中的碎片更多。需要定期运行 OPTIMIZE TABLE 以保持性能。而定长列就没有这个问题。
- 在出现表崩溃时，定长列的表易于重新构造，因为每个记录的开始位置是确定的。可变长列就没有这种便利。这不是一个与查询处理有关的性能问题，但它必定能加快表的修复过程。

如果表中有可变长的列，将它们转换为定长列能够改进性能，因为定长记录易于处理。在试图这样做之前，应该考虑下列问题：

- 使用定长列涉及某种折衷。它们更快，但占用的空间更多。 CHAR(n) 类型列的每个值总要占用 n 个字节（即使空串也是如此），因为在表中存储时，值的长度不够将在右边补空格。而 VARCHAR(N) 类型的列所占空间较少，因为只给它们分配存储每个值所需要的空间，每个值再加一个字节用于记录其长度。因此，如果在 CHAR 和 VARCHAR 列之间进行选择，需要对时间与空间作出折衷。如果速度是主要关心的因素，则利用 CHAR 列来取得定长列的性能优势。如果空间是关键，应该使用 VARCHAR 列。
- 不能只转换一个可变长列；必须对它们全部进行转换。而且必须使用一个 ALTER TABLE 语句同时全部转换，否则转换将不起作用。
- 有时不能使用定长类型，即使想这样做也不行。例如对于比 255 字符长的串，没有定长类型。

4. 可索引类型

索引能加快查询速度，因此，应该选择可索引的类型。

5. NULL 与 NOT NULL 类型

如果定义一列为 NOT NULL，其处理更快，因为 MySQL 在查询处理中不必检查该列的值弄清它是否为 NULL，表中每行还能节省一位。

避免列中有 NULL 可以使查询更简单，因为不需要将 NULL 作为一种特殊情形来考虑。

通常，查询越简单，处理就越快。

所给出的性能准则有时是互相矛盾的。例如，根据 MySQL 能对行定位这一方面来说，包含 CHAR 列的定长行比包含 VARCHAR 列的可变长行处理快。但另一方面，它也将占用更多的空间，因此，会导致更多的磁盘活动。从这个观点来看， VARCHAR 可能会更快。作为一个经验规则，可假定定长列能改善性能，即使它占用更多的空间也如此。对于某个特殊的关键应用，可能会希望以定长和可变长两种方式实现一个表，并进行某些测试以决定哪种方式对您的特定应用来说更快。

2.3.4 希望对值进行什么样的比较

根据定义串的方式，可以使串类型以区分大小写或不区分大小写的方式进行比较和排序。表2-14 示出不区分大小写的每个类型及其等价的区分大小写类型。根据列定义中给不给出关键字 BINARY，有的类型（CHAR、VARCHAR）是二进制编码或非二进制编码的。其他类型（BLOB、TEXT）的“二进制化”隐含在类型名中。

表2-14 串类型是否区分大小写

非二进制类型（不区分大小写）	二进制类型（区分大小写）
CHAR(M)	CHAR(M) BINARY
VARCHAR(M)	VARCHAR(M) BINARY
TINYTEXT	TINYBLOB
TEXT	BLOB
MEDIUMTEXT	MEDIUMBLOB
LONGTEXT	LONGBLOB

请注意，二进制（区分大小写）类型仅在比较和排序行为上不同于相应的非二进制（不区分大小写）类型。任意串类型都可以包含任意种类的数据。特别是， TEXT 类型尽管在列类型名中称为“ TEXT（文本）”，但它可以很好地存储二进制数据。

如果希望使用一个在比较时既区分大小写，又可不区分大小写的列。可在希望进行区分大小写的比较时，利用 BINARY 关键字强制串作为二进制串值。例如，如果 my_col 为一个 CHAR 列，可按不同的方式对其进行比较：

```
my_col = “ ABC ”          不区分大小写
BINARY my_col = “ ABC ”   区分大小写
my_col = BINARY “ ABC ”   区分大小写
```

如果有一个希望以非字典顺序存储的串值，可考虑使用 ENUM 列。ENUM 值的排序是根据列定义中所列出枚举值的顺序进行的，因此可以使这些值以任意想要的次序排序。

2.3.5 计划对列进行索引吗

使用索引可更有效地处理查询。索引的选择是第 4 章中的一个主题，但一般原则是将 WHERE 子句中用来选择行的列用于索引。

如果您要对某列进行索引或将该列包含在多列索引中，则在类型的选择上可能会有限定。在早于 3.23.2 版的 MySQL 发行版中，索引列必须定义为 NOT NULL，并且不能对 BLOB 或 TEXT 类型进行索引。这些限制在 MySQL 3.23.2 版中都撤消了，但如果您正使用一个更早的版本，不能或不愿升级，那么必须遵从这些约束。不过在下列情形中可以绕过它们：

- 如果可以指定某个值作为专用的值，那么能够将其作为与 NULL 相同的东西对待。对于 DATE 列，可以指定“ 0000 - 00 - 00 ”表示“ 无日期 ”。在串列中，可以指定空串代表“ 缺值 ”。在数值列中，如果该列一般只存储非负值，则可使用 -1。
- 不能对 BLOB 或 TEXT 类型进行索引，但如果串不超过 255 它符，可使用等价的 VARCHAR 列类型并对其进行索引。可将 VARCHAR(255) BINARY 用于 BLOB 值，将 VARCHAR(255) 用于 TEXT 值。

2.3.6 列类型选择问题的相互关联程度

不要以为列类型的选择是相互独立的。例如，数值的取值范围与存储大小有关；在增大取值的范围时，需要更多的存储空间，这会影响性能。另外，考虑选择使用 AUTO_INCREMENT 来创建一个存放唯一序列号的列有何含义。这个选择有几个结果，它们涉及列的类型、索引和 NULL 的使用，现列出如下：

- AUTO_INCREMENT 是一个应该只用于整数类型的列属性。它将您的选择限定在 TINYINT 到 BIGINT 之上。
- AUTO_INCREMENT 列应该进行索引，从而当前最大的序列号可以很快就确定，不用对表进行全部扫描。此外，为了防止序列号被重用，索引号必须是唯一的。这表示必须将列定义为 PRIMARY KEY 或定义为 UNIQUE 索引。
- 如果所用的 MySQL 版本早于 3.23.2，则索引列不能包含 NULL 值，因此，必须定义列为 NOT NULL。

所有这一切表示，不能像如下这样只定义一个 AUTO_INCREMENT 列：

```
my_col arbitrary_type AUTO_INCREMENT
```

应该按如下定义：

```
my_col integer_type AUTO_INCREMENT NOT NULL PRIMARY KEY
```

或如下定义：

```
my_col integer_type AUTO_INCREMENT NOT NULL, UNIQUE(my_col)
```

使用 AUTO_INCREMENT 得到的另一个结果是，由于它是用来生成一个正值序列的，因此，最好将 AUTO_INCREMENT 列定义为 UNSIGNED：

```
my_col integer_type UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY
my_col integer_type UNSIGNED AUTO_INCREMENT NOT NULL, UNIQUE(my_col)
```

2.4 表达式求值和类型转换

MySQL 允许编写包括常量、函数调用和表列引用的表达式。这些值可利用不同类型的运算符进行组合，诸如算术运算符或比较运算符。表达式的项可用圆括号来分组。

表达式在 SELECT 语句的列选择列表和 WHERE 子句中出现得最为频繁，如下所示：

```
SELECT
    CONCAT(last_name, ", ", first_name),
    (TO_DAYS(death) - TO_DAYS(birth) / 365)
FROM president
WHERE
    birth > "1900-1-1" AND DEATH IS NOT NULL
```

所选择的每列给出了一个表达式，如 WHERE 子句中所示的那样。表达式也出现在

DELETE 和 UPDATE 语句的 WHERE 子句中，以及出现在 INSERT 语句的 VALUES( ) 子句中。

在 MySQL 遇到一个表达式时，它对其求值得出结果。例如， (4 * 3)/(4 - 2) 求值得 6。表达式求值可能涉及类型转换。例如， MySQL 在数 960821 用于需要日期值的环境时，将其转换为日期“ 1996-08-21 ”。

本节讨论怎样编写 MySQL 的表达式，以及在表达式求值中 MySQL 所使用的类型转换规则。每个 MySQL 的运算符都介绍过了，但 MySQL 有那么多的函数，我们只接触过几个。每个运算符和函数的进一步介绍可参阅附录 C。

2.4.1 撰写表达式

表达式可以只是一个简单的常量，如：

```
0                    数值常量
“ abc ”              串常量
```

表达式可以进行函数调用。有的函数需要参数（圆括号中有值），而有的不需要。多个参数应该用逗号分隔。在调用一个函数时，参数旁边可以有空格，但在函数名与圆括号间不能有空格。下面是一些函数例子：

```
NOW( )                          无参数函数
STRCMP(“ abc ”, “ def ” )       有两个参数的函数
STRCMP( “ abc ”, “ def ” )      参数旁边有空格是合法的
STRCMP (“ abc ”, “ def ” )      函数名后跟空格是不合法的
```

如果函数名后有一个空格， MySQL 的分析程序可能会将函数名解释为一个列名（函数名不是保留字，如果需要的话，可将它们用作列名）。其结果是出现一个语法错误。

表达式中可使用表列。最简单的情形是，当某个列所属的表在上下文中是明确的，则可简单地给出列名对该列进行引用。下面的每个 SELECT 语句中惟一地出了一个表名，因此，列的引用无歧义：

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

如果使用哪个表的列不明确，可在列名前加上表名。如果使用哪个数据库中的表也不明确的话，可在表名前加上数据库名。如果只是希望意思更明显，也可以在无歧义的上下文中利用这种更为具体的表示形式，如：

```
SELECT
    president.last_name, president.first_name,
    member.last_name, member.first_name
FROM president, member
WHERE president.last_name = member.last_name

SELECT samp_db.student.name FROM samp_db.student
```

总之，可以组合所有这些值以得到更为复杂的表达式。

1. 运算符的类型

MySQL 有几种类型的运算符，可用来连接表达式的项。算术运算符，如表 2-15 所示，一般包括加、减、乘、除以及模运算符。在两个操作数都是整数时，“ + ”、“ - ”和“ * ”算术运算用 BIGINT（64 位）整数值来完成。而在结果预期为一个整数时，“ / ”和“ % ”也是用

BIGINT（64 位）整数值来完成的。应该认识到，如果某个运算涉及更大的值，如结果超过 64 位，其结果不可预料。

表2-15 算术运算符

运 算 符	语 法	说 明
+	a + b	加；操作数之和
-	a - b	减；操作数之差
-	-a	一元减号；操作数取负
*	a * b	乘；操作数之积
/	a / b	除；操作数之商
%	a % b	模；操作数除后的余数

逻辑运算符如表2-16所示，对表达式进行估计以确定其为真（非零）或假（零）。MySQL 包含有 C 风格的“ && ”、“ || ”和“ ! ”运算符，可替换 AND、OR 和 NOT。要特别注意“ || ”运算符，ANSI SQL 指定“ || ”作为串连接符，但在 MySQL 中，它表示一个逻辑或运算。如果执行下面的查询，则返回数 0：

```
SELECT "abc" || "def"                    → 0
```

MySQL 为进行运算，将“ abc ”和“ def ”转换为整数，且两者都转换为 0, 0与 0进行或运算，结果为 0。在 MySQL 中，必须用 CONCAT(“ abc ”，“ def ”) 来完成串的连接。

表2-16 逻辑运算符

运 算 符	语 法	说 明
AND, &&	a AND B, a && b	逻辑与；如果两操作数为真，结果为真
OR, ||	a OR B, a || b	逻辑或；任一操作数为真，结果为真
NOT, !	NOT a, !a	逻辑非；如果操作数为假，结果为真

位运算符如表2-17 所示，完成按位“ 与 ”和“ 或 ”，其中结果的每一位按两个操作数的对应位的逻辑 AND 或 OR 求值。还可以进行位的左移或右移。位运算用 BIGINT（64 位）整数值进行。

表2-17 位运算

运 算 符	语 法	说 明
&	a & b	按位 AND（与）；如果两个操作数的对应位为 1，则该结果位为 1
|	a | b	按位 OR（或）；如果两操作数的对应位中有一位为 1，则该结果位为 1
<<	a << b	将 a 左移 b 个二进制位
>>	a >> b	将 a 右移 b 个二进制位

比较运算符如表 2-18 所示，其中包括测试相对大小或数和串的顺序的运算符，以及完成模式匹配和测试 NULL 值的运算符。“ <=> ”运算符是 MySQL 特有的，在 MySQL 3.23版本中引入。

表2-18 比较运算符

运 算 符	语 法	说 明
=	a = b	如果两操作数相等，为真
!=, <>	a != b, a <> b	如果两操作数不等，为真
<	a < b	如果 a 小于 b，为真
<=	a <= b	如果 a 小于等于 b，为真

(续)

运 算 符	语 法	说 明
>=	a >= b	如果 a 大于等于 b，为真
>	a > b	如果 a 大于 b，为真
IN	a IN (b1, b2, ...)	如果 a 为 b1, b2, ...中任意一个，为真
BETWEEN	a BETWEEN b AND c	如果 a 在值 b 和 c 之间包括等于 b 和 c，为真
LIKE	a LIKE b	SQL 模式匹配；如果 a 与 b 匹配，为真
NOT LIKE	a NOT LIKE b	SQL 模式匹配；如果 a 与 b 不匹配，为真
REGEXP	a REGEXP b	扩展正规表达式匹配；如果 a 与 b 匹配，为真
NOT REGEXP	a NOT REGEXP b	扩展正规表达式匹配；如果 a 与 b 不匹配，为真
<=>	a <=> b	如果两操作数相同（即使为 NULL），为真
IS NULL	a IS NULL	如果操作数为 NULL，为真
IS NOT NULL	a IS NOT NULL	如果操作数不为 NULL，为真

自 MySQL 3.23版本起，可使用 BINARY 运行符，此运算符可用来将一个串转换为一个二进制串，这个串在比较中是区分大小写的。下列的第一个比较是不区分大小写的，但第二个和第三个比较是区分大小写的：

```
"abc" = "Abc"                → 1
BINARY "abc" = "Abc"         → 0
"abc" = BINARY "Abc"         → 0
```

没有相应的 NOT BINARY 计算。如果希望使一个列既能在区分大小写又能在不区分大小写的环境中使用，则应该利用不区分大小写的列并对希望区分大小写的比较使用 BINARY。

对于利用二进制串类型（CHAR BINARY、VARCHAR BINARY 和 BLOB 类型）定义的列，其比较总是区分大小写的。为了对这样的列类型实现不区分大小写的比较，可利用 UPPER( ) 或 LOWER( ) 来转换成相同的大小写：

```
UPPER(col_name) < UPPER("Smith")
LOWER(col_name) < LOWER("Smith")
```

对于不区分大小写的串比较，有可能把多个字符认为是相等的，这取决于所用的字符集。例如“ e ”和“ é ”对于比较和排序操作可能是相同的。二进制（区分大小写）比较利用字符的 ASCII 值来完成。

模式匹配允许查找值而不必给出精确的直接值。 MySQL 利用 LIKE 运算符和通配符“ % ”（匹配任意的字符序列）和“ _ ”（匹配任意单个字符），提供 SQL 的模式匹配。MySQL 还基于类似于诸如 grep、sed 和 vi 等 UNIX 程序中所用的 REGEXP 运算符和扩展正规表达式，提供模式匹配。为了完成模式匹配，必须使用这些模式匹配运算符中的某一个；不能使用“ = ”。为了进行相反的模式匹配，可使用 NOT LIKE 或 NOT REGEXP。

除了使用的模式运算符和模式字符不同外，这两种模式匹配还在以下重要的方面存在差异：

- 除非至少有一个操作数为二进制串，否则 LIKE 是不区分大小写的。REGEXP 是区分大小写的。（在 MySQL 3.23.4 以后的版本中，除非至少有一个操作数是二进制串，否则 REGEXP 是不区分大小写的。）
- 仅当整个串匹配， SQL 才是模式匹配的。仅当相应的模式在串中某一处出现，正规表达式才匹配。

用于 LIKE 运算符的模式可以包括“ % ”和“ _ ”通配符。例如，模式“ Frank% ”与任何以“ Frank ”起头的串匹配：

```
"Franklin" LIKE "Frank%"                    → 1
"Frankfurter" LIKE "Frank%"                 → 1
```

通配符“ % ”与任何串匹配，其中包括与空字符序列匹配，因此“ Frank% ”与“ Frank ”匹配：

```
"Frank" LIKE "Frank%"                       → 1
```

这也表示模式“ % ”与任何串匹配，其中包括与空串匹配。但是，“ % ”不与 NULL 匹配。事实上，具有 NULL 操作数的任何模式匹配都将失败：

```
"Frank" LIKE NULL                           → NULL
NULL LIKE "Frank%"                          → NULL
```

MySQL 的 LIKE 运算符是不区分大小写的，除非它至少有一个操作数是二进制串。因此，缺省时“ Frank% ”与串“ Frankly ”和“ frankly ”匹配，但在二进制比较中，它只与其中之一匹配：

```
"Frankly" LIKE "Frank%"                     → 1
"frankly" LIKE "Frank%"                     → 1
BINARY "Frankly" LIKE "Frank%"              → 1
BINARY "frankly" LIKE "Frank%"              → 0
```

这不同于 ANSI SQL 的 LIKE 运算符，它是区分大小写的。

通配符可在模式中任何地方给出。“ %bert ”与“ Englebert ”、“ Bert ”和“ Albert ”匹配。“ %bert% ”也与所有这些串匹配，而且还与如像“ Berthold ”、“ Bertram ”、和“ Alberta ”这样的串匹配。

LIKE 所允许的另一个通配符是“ _ ”，它与单个字符匹配。“ __ ”与三个字符的串匹配。“ c_t ”与“ cat ”、“ cut ”甚至“ c_t ”匹配（因为“ _ ”与自身匹配）。

为了关掉“ % ”或“ _ ”的特殊含义，与这些字符的直接实例相匹配，需要在它们前面放置一个斜杠（“ \% ”或“ _ ”），如：

```
"abc" LIKE "a%c"                            → 1
"abc" LIKE "a\%c"                           → 0
"a%c" LIKE "a\%c"                           → 1
```

MySQL 的另一种形式的模式匹配使用了正规表达式。运算符为 REGEXP 而不是 LIKE（RLIKE 为 REGEXP 的同义词）。最常用的正规表达式模式字符如下：

‘ . ’与任意单个字符匹配：

```
"abc" REGEXP "a.c"                          → 1
```

‘ [...] ’与方括号中任意字符匹配。可列出由短划线‘ - ’分隔的范围端点指定一个字符范围。为了否定这种区间的意义（即与未列出的任何字符匹配），指定‘ ^ ’作为该区间的第一个字符即可：

```
"abc" REGEXP "[a-z]"                        → 1
"abc" REGEXP "[^a-z]"                       → 0
```

‘ * ’表示“ 与其前面字符的任意数目的字符匹配 ”，因此，如‘ x* ’与任意数目的‘ x ’字符匹配，例如：

```
"abcdef" REGEXP "a.*f"                      → 1
"abc" REGEXP "[0-9]*abc"                    → 1
"abc" REGEXP "[0-9][0-9]*"                  → 0
```

“ 任意数目 ”包括 0个实例，这也就是为什么第二个表达式匹配成功的原因。‘ ^pat ’和‘ pat$ ’固定了一种模式匹配，从而模式 pat 只在它出现在串的前头时匹配，而‘ ^pat$ ’只在

pat 匹配整个串时匹配，例如：

```
"abc" REGEXP "b"                    → 1
"abc" REGEXP "^b"                   → 0
"abc" REGEXP "b$"                   → 0
"abc" REGEXP "^abc$"                → 1
"abcd" REGEXP "^abc$"               → 0
```

REGEXP 模式可从某个表列中取出，虽然如果该列包含几个不同的值时，这样做比常量模式慢。每当列值更改时，必须对模式进行检查并转换成内部形式。

MySQL 的正规表达式匹配还有一些特殊的模式字符。要了解更详细信息请参阅附录 C。

2. 运算符的优先级

当求一个表达式的值时，首先查看运算符以决定运算的先后次序。有的运算符具有较高的优先级；例如，乘和除比加和减的优先级更高。下面的两个表达式是等价的，因为“ * ”和“ / ”先于“ + ”和“ - ”计算：

```
1 + 2 * 3 - 4 / 5                   → 6.2
1 + 6 - .8                          → 6.2
```

下面列出了运算符的优先级，从高到低。列在同一行中的运算符具有相同的优先级。优先级较高的运算符在优先级较低的运算符之前求值。

```
BINARY
NOT  !
- (unary minus)
*  /  %
+  -
<<  >>
&
|
<  <=  =  <=>  !=  <>  >=  >  IN  IS  LIKE  REGEXP  RLIKE
BETWEEN
AND  &&
OR  ||
```

可用圆括号来忽略运算符的优先级并改变表达式的求值顺序，如：

```
1 + 2 * 3 - 4 / 5                   → 6.2
(1 + 2) * (3 - 4) / 5               → -0.6
```

3. 表达式中的 NULL 值

请注意，在表达式中使用 NULL 值时，其结果有可能出现意外。下列准则将有助于避免出问题。

如果将 NULL 作为算术运算或位运算符的一个操作数，其结果为 NULL：

```
1 + NULL                            → NULL
1 | NULL                            → NULL
```

如果将 NULL 用于逻辑运算符，NULL 被认为是假：

```
1 AND NULL                          → 0
1 OR NULL                           → 1
0 AND NULL                          → 0
0 OR NULL                           → 0
```

NULL 作为任意比较运算符的操作数，除 <=>、IS NULL 和 IS NOT NULL 运算符（它们是专门扩展来处理 NULL 值的）外，将产生一个 NULL 结果。如：

```
1 = NULL                            → NULL
NULL = NULL                         → NULL
1 <=> NULL                          → 0
```

```
NULL <=> NULL                    → 1
1 IS NULL                        → 0
NULL IS NULL                     → 1
```

如果给函数一个 NULL 参数，除了那些处理 NULL 参数的函数外，一般返回一个 NULL 结果。例如，IFNULL( ) 能够处理 NULL 参数并适当地返回真或假。STRCMP( ) 期望一个非 NULL 的参数；如果它发现传给它的是一个 NULL 参数，则返回 NULL 而不是真或假。

在排序操作中，NULL 值被归到一起。在升序排序中，NULL 将排在所有非 NULL 值之前（包括空串），而在降序排序中，NULL 将排在所有非 NULL 值之后。

2.4.2 类型转换

MySQL 根据所执行的操作类型，自动地进行大量的类型转换，任何时候，只要将一个类型的值用于需要另一类型值的场合，就会进行这种转换。下面是需要进行类型转换的原因：

- 操作数转换为适合于某种运算符求值的类型。
- 函数参数转换为函数所需的类型。
- 转换某个值以便赋给一个具有不同类型的表列。

下列表达式涉及类型转换。它由加运算符“ + ”和两个操作数 1 和“ 2 ”组成：

```
1 + "2"
```

其中操作数的类型不同，一个是数，另一个是串，因此，MySQL 对其中之一进行转换以便使它们两个具有相同的类型。但是应该转换哪一个呢？因为，“ + ”是一个数值运算符，所以 MySQL 希望操作数为数，因此，将串“ 2 ”转换为数 2。然后求此表达式的值得出 3。再举一例。CONCAT( ) 函数连接串产生一个更长的串作为结果。为了完成此工作，它将参数解释为串，而不管参数实际是何类型。如果传递给 CONCAT( ) 几个数，则它将把它们转换成串，然后返回这些串的连接，如：

```
CONCAT(1,2,3)                    → "123"
```

如果作为表达式的组成部分调用 CONCAT( )，可能会进行进一步的类型转换。考察下列表达式及其结果：

```
REPEAT('X',CONCAT(1,2,3)/10)     → "XXXXXXXXXXX"
```

CONCAT(1, 2, 3) 产生串“ 123 ”。表达式 “ 123 ”/10 转换为 123/10，因为除是一个算术运算符。这个表达式的结果的浮点形式为 12.3，但 REPEAT( ) 需要整数的重复计数值，所以进行整除得 12。然后，REPEAT(‘ X ’, 12) 产生一个含有12个‘ X ’ 字符的结果串。

一般原则是，MySQL 尽量将值转换为表达式所需要的类型，尽量避免由于值的类型不对而导致错误。根据上下文，MySQL 将在三种通用类型（数、串或日期与时间）之间进行值的转换。但是，值不能总是可以从一种类型转为另一种类型。如果被转换值不是给定类型的合法值，则此转换失败。将如 “ abc ” 这样不像数的东西转换为数，则结果为 0。将不像日期或时间的东西转换为日期或时间类型结果为该类型的“ 零 ”值。例如，将串 “ abc ” 转换为日期结果为“ 零 ”日期“ 0000-00-00 ”。而任何值都可以处理为串，因此，一般将某个值转换为串不会产生问题。

MySQL 也进行一些微小的类型转换。如果在整型环境中使用一个浮点值，此值将被转换，转换时进行四舍五入。也可以进行相反的工作；一个整数用作浮点数也不会有问题。

除非其内容显示表示一个数，否则十六进制常数一般作为串处理。在串上下文中，每对

十六进制数字转换为一个字符，其结果作为串。下面是一些转换的样例：

```
0x61                                → "a"
0x61 + 0                            → 97
CONCAT(0x61)                        → "a"
CONCAT(0x61 + 0)                    → "97"
```

相同的解释原理也应用到比较上；除非与其比较的是一个数，否则十六进制常量按串对待，例如：

```
10 = 0x0a                           → 1
10 = 0x09                           → 0
"\n" = 0x0a                         → 1
"\n" = 0x0a + 0                     → 0
("\n" = 0x0a) + 0                   → 1
```

某些运算符可将操作数强制转换为它们所要的类型，而不管操作数是什么类型。例如，算术运算符需要数，并按此对操作数进行转换，参考如下运算：

```
3 + 4                               → 7
"3" + 4                             → 7
"3" + "4"                           → 7
```

MySQL 不对整个串进行寻找一个数的查找；它只查看串的起始处。如果一个串不以数作为前导部分，其转换结果为 0。

```
"1973-2-4" + 0                      → 1973
"12:14:01" + 0                      → 12
"23-skidoo" + 0                     → 23
"-23-skidoo" + 0                    → -23
"carbon-14" + 0                     → 0
```

请注意，MySQL 的串到数的转换规则自 3.23版以后已经改变了。在该版本以前，类似于数的串被转换为四舍五入的整数值。自 3.23 版后，它们转换为浮点值，例如：

```
"-428.9" + 0                        → -429 (MySQL < 3.23)
"-428.9" + 0                        → -428.9 (MySQL ≥ 3.23)
```

逻辑和位运算符比算术运算符要求更为严格。它们不仅希望操作数为数，而且还要求是整数。这表示一个浮点数，如 .3，不被视为真，虽然它是非零的；这是因为在转换为整数时，.3已经转换为0了。在下面的表达式中，除非各操作数有一个至少为 1 的值，否则各操作数不被认为是真。

```
0.3 OR .04                          → 0
1.3 OR .04                          → 1
0.3 AND .04                         → 0
1.3 AND .04                         → 0
1.3 AND 1.04                        → 1
```

这种转换也出现在 IF( ) 函数中，此函数要求第一个参数为整数。为了恰当地对浮点值进行测试，最好是利用明确的比较。否则，小于 1 的值将被认为是假，例如：

```
IF(1.3,"non-zero","zero")           → "non-zero"
IF(0.3,"non-zero","zero")           → "zero"
IF(0.3>0,"non-zero","zero")         → "non-zero"
```

模式匹配运算符要求对串进行处理。这表示可将 MySQL 的模式匹配运算符用于数，因为 MySQL 会在试图进行的匹配中将它们转换成串。例如：

```
12345 LIKE "1%"                     → 1
12345 REGEXP "1.*5"                 → 1
```

大小比较运算符（“ < ”、“ <= ”、“ = ”等等）是上下文相关的；即，它们根据操作数的类

型求值。下面的表达式从数值上对操作数进行比较，因为操作符两边都是数。

```
2 < 11                                    → 1
```

下面的表达式涉及串比较，因为其两边的操作数都是串：

```
"2" < "11"                                → 0
```

在下面的比较中，类型是混合的，因此，MySQL 按数比较它们。结果是两个表达式都为真：

```
"2" < 11                                  → 1
2 < "11"                                  → 1
```

在各个比较中，MySQL 根据下列规则对操作数进行转换：

- 除了“ <=> ”运算符外，涉及 NULL 值的比较其值为 NULL（除 NULL <=> NULL 为真外，“ <=> ”与“ = ”相同）。
- 如果两个操作数都是串，则按串进行字典顺序的比较。串比较利用服务器上有效的字符集进行。
- 如果两个操作数都为整数，则按整数进行数的比较。
- 不与数进行比较的十六进制常量按二进制串进行比较。
- 如果其中有一个操作数为 TIMESTAMP 或 DATETIME 值而另一个为常量，则按 TIMESTAMP 值进行比较。这样做将使比较对 ODBC 应用更好。
- 否则，两个操作数将按浮点值进行数的比较。注意，这包括一个串与一个数进行比较的情况。其中串被转换为数，如果该串转换后不是一个数，则结果为 0。例如，”14.3 ”转换为 14.3，但“ L4.3 ”转换为 0。

1. 日期与时间的解释规则

MySQL 按表达式的环境将串和数自由地转换为日期和时间值，反之亦然。日期和时间值在数值上下文中转换为数；数在日期或时间上下文中转换为日期或时间。在将一个值赋予一个日期或时间列时，或在函数需要一个日期或时间值时，进行转换为日期或时间值的转换。

如果表 my_table 含有一个 DATE 列 date_col，下列语句是等价的：

```
INSERT INTO my_table SET date_col = "1997-04-13"
INSERT INTO my_table SET date_col = "19970413"
INSERT INTO my_table SET date_col = 19970413
```

TO_DAYS( ) 函数的参数在下面三个表达中为相同的值：

```
TO_DAYS("1997-04-10")                     → 729489
TO_DAYS("19970410")                       → 729489
TO_DAYS(19970410)                         → 729489
```

2. 测试并强制进行类型转换

为了了解表达式中类型转换是怎样进行的，用 mysql 程序发布一条对表达式求值的 SELECT 语句如下：

```
mysql> SELECT 0x41, 0x41 + 0;
+------+----------+
| 0x41 | 0x41 + 0 |
+------+----------+
| A    |       65 |
+------+----------+
```

正如您所想像的那样，笔者在撰写本章时，做了不少这种比较。

测试表达式的求值对于诸如 DELETE 或 UPDATE 这种修改记录的语句极为重要，因为需

要保证只涉及所需涉及的行。检查表达式的一个办法是，预先执行一条具有准备用于 DELETE 或 UPDATE 语句的相同 WHERE 子句，以验证该子句选择的行是正确的。假如表 my_table 具有一个含有下列值的 CHAR 列 char_col：

```
"abc"
"def"
"00"
"ghi"
"jkl"
"00"
"mno"
```

对于这些值，下列查询的作用是什么？

```
DELETE FROM my_table WHERE char_col = 00
```

原来的打算大概是想删除包含值 “00” 的那两行。但实际作用是删除了所有的行。之所以这样是由于 MySQL 的比较规则在起作用。char_col 为一个串列，但 00 没有用引号括起来，因此，它被作为数对待了。按 MySQL 的比较规则，涉及一个串与一个数的比较按两个数的比较来求值。随着 DELETE 查询的执行，char_col 的每个值被转换为 0，”00 ” 也被转换为 0，因此，所有不类似数的串都转换成 0。从而，对于每一行，WHERE 子句都为真，因此，DELETE 语句清空了该表。显然，这是一种在执行 DELETE 前，应该用 SELECT 语句对 WHERE 子句进行测试的情况，这样将会示出表达式所选择的行太多了。如下所示：

```
mysql> SELECT char_col FROM my_table WHERE char_col = 00;
+----------+
| char_col |
+----------+
| "abc"    |
| "def"    |
| "00"     |
| "ghi"    |
| "jkl"    |
| "00"     |
| "mno"    |
+----------+
```

如果不能肯定某个值的使用方式，可以利用 MySQL 的表达式求值机制将该值强制转换为特定的类型：

- 增加 + 0 或 + 0.0 到某项上以强制转换到一个数值：

```
0x65                    → "e"
0x65 + 0                → 101
0x65 + 0.0              → 101.0
```

- 利用 CONCAT( ) 将值转换为串：

```
14                      → 14
CONCAT(14)              → "14"
```

- 利用 ASCII( ) 得到字符的 ASCII 值：

```
"A"                     → "A"
ASCII("A")              → 65
```

- 利用 DATE_ADD( ) 强制转换串或数为日期：

```
19990101                → 19990101
DATE_ADD(19990101, INTERVAL 0 DAY)  → "1999-01-01"
"19990101"              → "19990101"
```

```
DATE_ADD("19990101", INTERVAL 0 DAY)          → "1999-01-01"
```

3. 超范围值或非法值的转换

超范围值或非法值的转换的基本原则为：无用输入，无用输出。如果不在存储日期前对其进行验证，那么可能会得到不喜欢的东西。下面给出一些 MySQL 处理超范围值或不合适值的一般原则，这些内容曾经在前面介绍过：

- 对于数值或TIME列，超出合法范围的值被剪裁为相应取值范围的最接近的数值并作为结果值存储。
- 对于非 ENUM 或 SET 的串列，太长的串被截为适合该列存储的最大长度的串。ENUM 或 SET 列的赋值依赖于定义列时给出的合法值。如果赋予 ENUM 列一个未作为枚举成员给出的值，将会赋予一个错误成员（即，对应于零值成员的空串）。如果赋予 SET 列一个包含未作为集合成员给出的子串的值，那么，那些未作为集合成员给出的子串将被删除，并将剩余成员构成的值赋给该列。
- 对于日期或时间列，非法值被转换为该类型适当的“零”值（参阅表 2-11）。对于非 TIME 的日期和时间列，超出取值范围的值可转换为“零”值、NULL或某种其他的值（换句话说，结果是不可预料的）。

这些转换都将作为 ALTER TABLE、LOAD DATA、UPDATE 和多行 INSERT 语句的警告信息报告。在 mysql 客户机中，这些信息显示在查询报告的状态行上。在编程语言中，可通过某些其他手段取得这个信息。如果使用的是 MySQL C API，那么可调用 mysql_info( ) 函数来获得这个信息。对于 Perl DBI API，可利用数据库连接的 mysql_info 属性。所提供的这个信息是警告信息的次数计数。为了知道更改了哪些行，可发布一条 SELECT ... INTO OUTFILE 查询，并将结果与原始行进行比较。