附录C 运算符和函数参考

本附录列出所有可在 SQL 语句中使用的 MySQL 函数，还给出构造表达式的运算符。表达式求值常常涉及对表达式中值的类型转换。关于类型转换的环境及 MySQL 用来将值从一种类型转换为另一种类型的规则，可参阅第 2章。

除非另有说明，否则所列出的运算符和函数在 MySQL 中得到支持至少可回溯到 MySQL 3.21.0 版。如果某个运算符或函数的功能与这里介绍的不同，可参阅 MySQL 参考指南中的注释。很可能是在旧的 MySQL 版本中有问题，但后来已经解决了。

运算符和函数的例子以下列形式给出：

```
expression                                    → result
```

其中 expression 说明怎样使用运算符或函数，而 result 给出表达式求值的结果。例如：

```
LOWER("ABC")                                  → "abc"
```

其含意是函数调用 LOWER ("ABC") 得出结果串 ” abc ”。

可利用 mysql 程序试验本附录中所给出的例子。为试验上面的例子，调用 mysql，输入该例子，在其前面放上 SELECT，后面放一个分号，然后按 Enter 键。如下所示：

```
mysql> SELECT LOWER("ABC");
+-------------+
| LOWER("ABC") |
+-------------+
| abc          |
+-------------+
```

MySQL 不需要具有 FROM 子句的 SELECT 语句，这使得以这种方式输入任意表达式对运算符和函数进行试验可以变得很容易（有的数据库系统不允许发布无 FROM 子句的 SELECT 语句，这是一个不好的限制）。

各个例子都包含有不能在其他地方示范的函数的完整表达式。“ 函数汇总 ” 一节就是按这种方式书写的，因为这些函数除了只在特定的表中引用外是没有意义的。

函数名以及为单词形式的操作符（如 BETWEEN）可以由任意的大小写字符来指定。

有的函数参数重复出现，具有固定意义：

- *expr* 代表一个表达式，在不同的上下文中，可能代表数值、串或日期、时间表达式，而且可与常量、表列引用或其他表达式结合。
- *str* 代表一个串，可为字符串、串值表列引用或产生串的表达式。
- *n* 代表一个整数（还可以用字母表中与 n 相近的字母来表示）。
- *x* 代表一个浮点数（还可以用字母表中与 x 相近的字母来表示）。

其他一些参数名较少使用，因此在使用的地方再定义。运算符或函数调用序列的可选部分由方括号（[ ]）表示。

C.1 运算符

运算符用来结合表达式中的项以完成算术、比较、位逻辑或逻辑操作以及模式匹配。

C.1.1 运算符的优先级

运算符优先级的级别不同，这些级别在下面的清单中给出，从最高到最低。相同行上的运算符具有相同的优先级。给定优先级的运算符从左到右求值。较高优先级的运算符在较低优先级的运算符之前求值。

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

一元运算符（一元减、NOT 和 BINARY）较二元运算符更为紧密。即，它们与表达式中紧跟的项结合在一起，而不是与整个表达式结合在一起。如：

```
--2+3                                    → 1
- (2+3)                                  → -5
```

C.1.2 分组运算符

圆括号 (和) 可用来将表达式分组，以及改变运算符的决定表达式中各项求值顺序的缺省优先级。圆括号还可用来使表达式的含义明确，更好理解。如：

```
1 + 2 * 3 / 4                            → 2.50
(((1 + 2) * 3) / 4)                      → 2.25
```

C.1.3 算术运算符

这些运算符进行标准的算术运算。算术运算符对数而不是对串起作用（但是类似于数的串可自动转换这相应的数值值）。涉及 NULL 值的算术运算得出一个 NULL 结果。

■ + 求算术和。

```
2 + 2                                    → 4
3.2 + 4.7                                → 7.9
"43bc" + "21d"                           → 64
"abc" + "def"                            → 0
```

上面最后一个例子说明 “ + ”不像在某些语言中那样可作为串连接运算符。这里，进行算术运算之前，串被转换为数，不是数的串转换为 0。连接串应该用 CONCAT( ) 函数。

■ - 在用于表达式的两项之间时，求两操作数之差。而用在单项之前时，求操作数的相反数（即，使项反号）。

```
10  - 7                                  → 3
- (10  - 7)                              → -3
```

■ * 求操作数之积。

```
2 * 3                                    → 6
2.3 * -4.5                               → -10.3
```

■ / 求操作数之商。零除得 NULL。

```
3 / 1                                    → 3.00
1 / 3                                    → 0.33
1 / 0                                    → NULL
```

■ % 模运算符；求 m 除以 n 的余数。m % n 与 MOD(*m*, *n*) 相同。与除一样，如果除数为零，则结果为 NULL。

```
12 % 4                                   → 0
12 % 5                                   → 2
12 % 0                                   → NULL
```

对于“ + ”、“ - ”和“ * ”，如果两边参数都是整数，则算术运算以 BIGINT 值进行（64 位整数）。这意味着较大值的表达式可能会超出 64 位整数计算的取值范围，其结果不可预料。如：

```
9999999999999999 * 9999999999999999      → -7527149226598858751
99999999999 * 99999999999 * 99999999999  → -1504485813132150785
18014398509481984 * 18014398509481984    → 0
```

对于“ / ”和“ % ”，BIGINT 值仅在结果转换为整数的上下文中使用。

C.1.4 比较运算符

如果比较为真，比较运算符返回 1；如果比较为假，返回 0，可对数或串进行比较，操作数根据下列规则进行转换：

■ 对于“ <=> ”运算符的表达式，当比较中包含 NULL 值时，其结果都为 NULL（除 NULL <=>NULL 为真外，“ <=> ”与“ = ”一样）。

■ 如果两个操作数皆为串，它们作为串按字典顺序比较。串比较按服务器上有效的字符集进行。

■ 如果两个操作数为整数，它们按整数进行数值比较。

■ 不与数进行比较的十六进制常量按二进制串进行比较。

■ 如果一个操作数为 TIMESTAMP 或 DATETIME 值，而另一操作数是一个常量，则此两个操作数按 TIMESTAMP 值进行比较。这样使比较对 ODBC 应用更好。

■ 如果前面任一规则都不适用，那么操作数按浮点数进行比较。注意，这包括串的比较和数的比较。串转换为一个数，如果串与数不类似，其转换结果为 0。例如，“ 14.3 ”转换为 14.3，但“ L4.3 ”转换为 0。

下面的比较说明了这些规则：

```
2 < 12                                   → 1
"2" < "12"                               → 0
"2" < 12                                 → 1
```

第一个比较涉及两个整数，所以进行数值比较。第二个比较涉及两个串，按字典顺序进行比较。第三个比较涉及一个串和一个数，因此它们按浮点值进行比较。

除非比较涉及二进制串，否则串比较是不区分大小写的。如果使用 BINARY 关键字或比较 CHAR BINARY、VARCHAR BINARY 或 BLOB 列的值，则进行区分大小写的比较。

■ = 如果操作数相等，其值为 1，否则为 0。

```
1 = 1                                    → 1
1 = 2                                    → 0
```

```
"abc" = "abc"                    → 1
"abc" = "def"                    → 0
"abc" = "ABC"                    → 1
BINARY "abc" = "ABC"             → 0
BINARY "abc" = "abc"             → 1
"abc" = 0                        → 1
```

“abc”既等于“abc”又等于“ABC”，因为缺省时，串比较是不区分大小写的。利用BINARY运算符，串比较是区分大小写的。“abc”等于0，因为根据比较规则，它要转换为一个数值。而“abc”不类似于数，所以转换为0。

■ <=> NULL-安全（NULL-safe）的等于运算符；在操作数相等时，其值为1，即使操作数为NULL也是这样。它类似于“=”。

```
1 <=> 1                          → 1
1 <=> 2                          → 0
NULL <=> NULL                    → 1
NULL = NULL                      → NULL
```

最后两个例子显示了“=”和“<=>”处理NULL比较的差异。

■ != 或 <> 如果操作数不相等，其值为1，否则其值为0。

```
3.4 != 3.4                       → 0
"abc" <> "ABC"                   → 0
BINARY "abc" <> "ABC"            → 1
"abc" != "def"                   → 1
```

■ < 如果左操作数小于右操作数，其值为1，否则其值为0。

```
3 < 10                           → 1
105.4 < 10e+1                    → 0
"abc" < "ABC"                    → 0
"abc" < "def"                    → 1
```

■ <= 如果左操作数小于等于右操作数，其值为1，否则其值为0。

```
"abc" <= "a"                     → 0
"a" <= "abc"                     → 1
13.5 <= 14                       → 1
(3 * 4) - (6 * 2) <= 0           → 1
```

■ >= 如果左操作数大于等于右操作数，其值为1，否则其值为0。

```
"abc" >= "a"                     → 1
"a" >= "abc"                     → 0
13.5 >= 14                       → 0
(3 * 4) - (6 * 2) >= 0           → 1
```

■ > 如果左操作数大于右操作数，其值为1，否则其值为0。

```
PI() > 3                         → 1
"abc" > "a"                      → 1
SIN(0) > COS(0)                  → 0
```

■ *expr* BETWEEN *min* AND *max* 如果*min*小于等于*expr*且*max*大于等于*expr*，其值为1. 如果操作数*expr*、*min*和*max*的类型都相同，此表达式等价于：

```
expr BETWEEN min AND max
(min <= expr AND expr <= max)
```

如果这些操作数类型不同，将进行类型转换，上面两个表达式可能不等价。 BETWEEN

利用根据 *expr* 的类型所决定的比较进行求值：

- 如果 *expr* 为一个串，则操作数按串进行字典顺序的比较。比较时区分大小写或不区分大小写，由 *expr* 是否二进制串来决定。
- 如果*expr* 为一个整数，则操作数按整数进行比较。
- 如果上两个规则都不真，则操作数按浮点数进行数值比较。

```
"def" BETWEEN "abc" and "ghi"                → 1
"def" BETWEEN "abc" and "def"                → 1
13.3 BETWEEN 10 and 20                       → 1
13.3 BETWEEN 10 and 13                       → 0
2 BETWEEN 2 and 2                            → 1
"B" BETWEEN "A" and "a"                      → 0
BINARY "B" BETWEEN "A" and "a"               → 1
```

BETWEEN 是在 MySQL 版本 3.21.2 中引入的。

- *expr* IN *(value1, value2, ...)*

```
expr NOT IN (value1,value2,...)
```

如果 *expr* 为列表中某个值，则 IN 的求值为 1，否则为 0。对于 NOT IN，求值正好相反。下面的表达式是等价的：

```
expr NOT IN (value1,value2,...)
NOT (expr IN (value1,value2,...))
```

如果列表中所有值都是常量，MySQL 将对它们进行排序，并利用二分查找对 IN 进行测试，这是非常快的。

```
3 IN (1,2,3,4,5)                             → 1
"d" IN ("a","b","c","d","e")                 → 1
"f" IN ("a","b","c","d","e")                 → 0
3 NOT IN (1,2,3,4,5)                         → 0
"d" NOT IN ("a","b","c","d","e")             → 0
"f" NOT IN ("a","b","c","d","e")             → 1
```

- *expr* IS NULL

```
expr IS NOT NULL
```

如果 *expr* 的值为 NULL，则 IS NULL 求值为 1，否则为 0。IS NOT NULL 正好相反。下面的表达式是等价的：

```
expr IS NOT NULL
NOT (expr IS NULL)
```

IS NULL 和 IS NOT NULL 应该用来确定 *expr* 的值是否为 NULL。因为这时不能使用普通的比较运算符“ = ”和“ ！= ”。

```
NULL IS NULL                                 → 1
0 IS NULL                                    → 0
NULL IS NOT NULL                             → 0
0 IS NOT NULL                                → 1
NOT (0 IS NULL)                              → 1
NOT (NULL IS NULL)                           → 0
NOT NULL IS NULL                             → 1
```

最后一例得出该结果是因为 NOT 比IS 的约束更紧（参阅“ 运算符优先级 ”一节）。

C.1.5 位运算符

位运算符利用 BIGINT 值（64 位整数）执行，它限定了运算的最大取值范围。涉及

NULL值的位运算产生 NULL 结果。

■ | 对操作数进行按位 OR（或）。

```
1 | 1                    → 1
1 | 2                    → 3
1 | 2 | 4 | 8            → 15
1 | 2 | 4 | 8 | 15       → 15
```

■ & 对操作数进行按位 AND（与）。

```
1 & 1                    → 1
1 & 2                    → 0
7 & 5                    → 5
```

■ << 将左操作数左移右操作数所指定的位数。移动负数位结果为零。

```
1 << 2                   → 4
2 << 2                   → 8
1 << 62                  → 4611686018427387904
1 << 63                  → -9223372036854775808
1 << 64                  → 0
```

■ >> 将左操作数右移右操作数所指定的位数。移动负数位结果为零。

```
16 >> 3                  → 2
16 >> 4                  → 1
16 >> 5                  → 0
```

C.1.6 逻辑运算符

逻辑运算符（也称为布尔运算符）测试表达式的真和假。所有逻辑运算在真时返回 1，假时返回 0。逻辑运算解释非零操作数为真，0 操作数为假。NULL 值按运算符说明中所规定的处理。

逻辑运算符希望操作数为数值，因此在运算符执行之前，串操作数将被转换为数。

■ NOT 或 ！ 逻辑否定。如果所跟的操作数为假，则其值为 1，否则其值为 0，而 NOT NULL 求值为 NULL。

```
NOT 0                    → 1
NOT 1                    → 0
NOT NULL                 → NULL
NOT 3                    → 0
NOT NOT 1                → 1
NOT "1"                  → 0
NOT "0"                  → 1
NOT ""                   → 1
NOT "abc"                → 1
```

■ OR 或 || 逻辑 OR。如果任一操作数为真（非零或非 NULL），则其值为 1，否则为 0。

```
0 OR 0                   → 0
0 OR 3                   → 1
4 OR 2                   → 1
1 OR NULL                → 1
```

■ AND 或 && 逻辑 AND。如果两个操作数都为真（非零或非 NULL），则其值为 1，否则为 0。

```
0 AND 0                  → 0
0 AND 3                  → 0
```

```
4 AND 2                                    → 1
1 AND NULL                                 → 0
```

在 MySQL 中，“!”、“||”和“&&”表示逻辑运算，正如它们在 C 中一样。但要特别注意，“||”，不像在某些 SQL 版本中那样进行串的连接。这里连接串应该用 CONCAT( )。

C.1.7 强制运算符

BINARY 使后跟的操作数被当作一个二进制串，从而使得涉及串的比较为区分大小写的。如果后跟的操作数为一个数，将被转换为串的形式：

```
"abc" = "ABC"                              → 1
"abc" = BINARY "ABC"                       → 0
BINARY "abc" = "ABC"                       → 0
"2" < 12                                   → 1
"2" < BINARY 12                            → 0
```

BINARY 产生数到串的转换，因此比较按字典序进行，因为这时两个操作数都是串。

BINARY 在 MySQL 3.23.0 中引入。

C.1.8 模式匹配运算符

MySQL 利用 LIKE 提供了 SQL 的模式匹配，利用 REGEXP 提供了扩展正规表达式模式匹配。除非要匹配的串或模式串为二进制串，否则 SQL 模式匹配是不区分大小写的。扩展正规表达式模式匹配总是区分大小写的。

SQL 模式匹配仅在模式与要匹配的整个串相匹配时才会成功。扩展正规表达式模式匹配仅在模式在串中任意地方找到才会成功。

■ str LIKE pat [ESCAPE ‘ c ’]

```
str NOT LIKE pat [ESCAPE 'c']
```

LIKE 完成一种简单的 SQL 模式匹配，如果模式串 pat 与整个串表达式 str 匹配，则其值为 1。如果不匹配，其值为 0。对于 NOT LIKE，正好相反。下列两个表达式是等价的：

```
str NOT LIKE pat [ESCAPE 'c']
NOT (str LIKE pat [ESCAPE 'c'])
```

如果任一个串为 NULL，则结果为 NULL。

在 SQL 模式中有两个字符具有特殊的含义，被用作通配符，说明如下：

字符	说明
%	匹配除 NULL 外的任意字符序列（包括空串）
_	匹配单个字符

模式可以包含任一通配符，或两个都包含：

```
"catnip" LIKE "cat%"                       → 1
"dogwood" LIKE "%wood"                     → 1
"bird" LIKE "____"                         → 1
"bird" LIKE "___"                          → 0
"dogwood" LIKE "%wo__"                     → 1
```

使用 LIKE 的 SQL 模式匹配的大小写敏感性由被比较的串决定。一般情况下，比较是不区分大小写的。如果有一个串是二进制串，比较将是区分大小写的：

```
"abc" LIKE "ABC"                    → 1
BINARY "abc" LIKE "ABC"             → 0
"abc" LIKE BINARY "ABC"             → 0
```

因为“ % ”与任意字符序列匹配，它甚至能匹配无字符的情况：

```
"" LIKE "%"                         → 1
"cat" LIKE "cat%"                   → 1
```

在 MySQL 中，可对数值表达式使用 LIKE：

```
50 + 50 LIKE "1%"                   → 1
200 LIKE "2__"                      → 1
```

为了按字面意义匹配一个通配符，应该在其前面放置一个转义字符“ \”：

```
"100% pure" LIKE "100%"             → 1
"100% pure" LIKE "100\%"            → 0
"100% pure" LIKE "100\% pure"       → 1
```

如果想使用非“ \ ”的另一转义字符，可利用 ESCAPE 子句进行指定：

```
"100% pure" LIKE "100^%" ESCAPE '^'     → 0
"100% pure" LIKE "100^% pure" ESCAPE '^' → 1
```

■ *str* REGEXP *pat*

```
str NOT REGEXP pat
```

REGEXP 执行扩展正规表达式模式匹配。扩展正规表达式类似于 UNIX 实用程序 grep 和 sed 所用的模式。可使用的模式序列如表 C-1 所示。

表 C-1 扩展正规表达式序列

序 列	说 明
^	在串的开始处匹配
$	在串的结尾处匹配
.	匹配任意单个字符，包括换行符
[...]	匹配出现在方括号中的任意字符
[^...]	匹配不出现在方括号中的任意字符
e*	匹配模式元素 e 的零个或多个实例
e+	匹配模式元素 e 的1个或多个实例
e?	匹配模式元素 e 的零个或多个实例
e1|e2	匹配模式元素 e1 或 e2
e{*m*}	匹配模式元素 e 的m 个实例
e{*m*,}	匹配模式元素 e 的m 个或更多个实例 （
e{,*n*}	匹配模式元素 e 的0 到 n 个实例
e{*m*,*n*}	匹配模式元素 e 的m 到 n 个实例
(...)	将模式元素组成单一元素
other	无特定字符与它们自身相配

如果模式串 pat 与串表达式 str 匹配，则 REGEXP 求值为 1，否则为 0。NOT REGEXP 正好相反。下列两个表达式是等价的：

```
str NOT REGEXP pat
NOT (str REGEXP pat)
```

如果其中有一个串为 NULL，则结果为 NULL。

模式不需要与整个串匹配，只需要在串中某处找出它即可。

[...] 和 [^...] 结构指定字符类。在一个类中，字符的取值范围可用两个端点字符中间加一短划线来表示。如，[a-z] 与任何小写字母匹配，而 [0-9] 与任意数字匹配。为了能够在类中表示字符“] ”，它必须为相应类的第一个字符。为了能够表示字符“ - ”，它必须是相应类的第一个或最后一个字符。而为了能够表示“ ^ ”，它必须不是“ [”之后的第一个字符。

还有其他几个与比较序列有关的更为复杂的特殊构造和等价类可用于字符类中。更详细的信息，请参阅 MySQL 参考指南。

如果在要匹配的串中的任何地方找到相应模式，则扩展正规表达式匹配成功，但可用“ ^ ”和“ $ ”强制模式只在串的开始或结尾处进行匹配。

```
"abcde" REGEXP "b"                    → 1
"abcde" REGEXP "^b"                   → 0
"abcde" REGEXP "b$"                   → 0
"abcde" REGEXP "^a"                   → 1
"abcde" REGEXP "e$"                   → 1
"abcde" REGEXP "^a.*e$"               → 1
```

扩展正规表达式模式匹配是区分大小写的：

```
"abc" REGEXP "ABC"                    → 0
"ABC" REGEXP "ABC"                    → 1
```

■ *str* RLIKE *pat*

```
str NOT RLIKE pat
```

RLIKE 和 NOT RLIKE 为 REGEXP 和 NOT REGEXP 的同义词。

对于串中的转义序列，MySQL 所用的语法类似于 C。例如，‘ \n ’、‘ \t ’ 和 ‘ \\ ’ 分别解释为换行、制表和反斜杠。为了在模式中指定这样的字符，可使用双反斜杠，‘ \\n ’、‘ \\t ’ 和 ‘ \\\\ ’。在对查询进行分析时，去掉一个反斜杠，而在进行模式匹配时，才对其余的转义序列进行解释。

C.2 函数

调用函数以执行计算并返回一个值。函数调用必须在函数名与其后的圆括号之间无空格。

NOW( )	正确
NOW ( )	不正确

如果将多个参数传递给一个函数，则参数必须用逗号分隔。函数参数周围允许有空格：

CONCAT(“ abc ”, “ def ”)	正确
CONCAT(“ abc ”, “ def ”)	也正确

C.2.1 比较函数

■ GREATEST(*expr1, expr2, ...*) 返回最大的参数，其中“ 最大 ”根据下列规则定义：

- ■ 如果在整数环境中调用此函数，或者所有参数都是整数，则各参数按整数进行比较。
- ■ 如果在浮点数环境中调用此函数，或者所有参数都是浮点数，则各参数按浮点数进行比较。
- ■ 如果前两个规则都不真，则各参数按串进行比较。除非有的参数为二进制串，否

则此比较是不区分大小写的。

```
GREATEST(2,3,1)                    → 3
GREATEST(38.5,94.2,-1)             → 94.2
GREATEST("a","ab","abc")           → "abc"
GREATEST(1,3,5)                    → 5
GREATEST("A","b","C")              → "C"
GREATEST(BINARY "A","b","C")       → "b"
```

GREATEST( ) 在 MySQL 3.22.5 版中引入。在更早的版本中，用的是 MAX( )。

■ IF(*expr1*, *expr2*, *expr3*) 如果 *expr*1 为真（非 0 或 NULL），返回 *expr2*，否则返回 expr*3*。IF( ) 根据使用它的上下文返回数或串。

```
IF(1,"true","false")               → "true"
IF(0,"true","false")               → "false"
IF(NULL,"true","false")            → "false"
IF(1.3,"non-zero","zero")          → "non-zero"
IF(0.3,"non-zero","zero")          → "zero"
IF(0.3 != 0,"non-zero","zero")     → "non-zero"
```

expr 是作为整数求值的，最后的三个例子说明，如果不仔细，结果将会搞得你不知所措。1.3 转换为整数值1，因此为真。但 0.3 转换为整数值 0，所以为假。最后的例子示出了使用浮点数的恰当方式：即应该用比较来使用浮点数，因为比较根据测试的结果返回 1 或 0。

■ IFNULL(*expr1*, *expr2*)

如果表达式 *expr1* 为 NULL，则返回 expr*2*；否则返回 expr*1*。IFNULL() 根据使用它的上下文返回一个数或串。

```
IFNULL(NULL,"null")                → "null"
IFNULL("not null","null")          → "not null"
```

■ INTERVAL(*n*, *n1*, *n2*, ...)

如果 n < n*1*，返回 0，如果 *n* < *n2*，返回 1，从此类推。或者如果 n 为 NULL，返回－1。其中值 n*1*, n*2*, ... 必须严格递增（即 n*1* < n*2* < ...），因为使用的是快速二分搜索法。否则，INTERVAL( ) 的结果将不可预测。

```
INTERVAL(1.1,0,1,2)                → 2
INTERVAL(7,1,3,5,7,9)              → 4
```

■ ISNULL(*expr*) 如果表达式 expr 的值为 NULL，返回 1；否则返回 0。

```
ISNULL(NULL)                       → 1
ISNULL(0)                          → 0
ISNULL(1)                          → 0
```

■ LEAST(*expr1*, *expr2*, ...) 返回最小的参数，其中“最小”利用与 GREATEST( ) 函数类似的比较规则来定义。

```
LEAST(2,3,1)                       → 1
LEAST(38.5,94.2,-1)                → -1.0
LEAST("a","ab","abc")              → "a"
```

LEAST( ) 在 MySQL 3.22.5 中引入。在更早的版本中用的是 MIN( )。

■ STRCMP(*str1*, *str2*) 如果 str1 与 str2 相同，返回1，如果不同，返回 0。如果两个参数中有一个为 NULL，则返回 NULL。该比较是区分大小写的。

```
STRCMP("a","a")                    → 0
STRCMP("a","A")                    → 1
```

C.2.2 数值函数

如果出现错误，数值函数返回 NULL。例如，如果将超出取值范围或非法的参数传递给函数，则函数将返回 NULL。

■ ABS(*x*) 返回 *x* 的绝对值。

```
ABS(13.5)                    → 13.5
ABS(-13.5)                   → 13.5
```

■ ACOS(*x*) 返回 *x* 的反余弦值，如果 *x* 不在 -1 到 1 的取值范围内，返回 NULL。

```
ACOS(1)                      → 0.000000
ACOS(0)                      → 1.570796
ACOS(-1)                     → 3.141593
```

ACOS( ) 在 MySQL 3.21.8 中引入。

■ ASIN(*x*) 返回 *x* 的反正弦值，如果 *x* 不在 -1 到 1 的取值范围内，返回 NULL。

```
ASIN(1)                      → 1.570796
ASIN(0)                      → 0.000000
ASIN(-1)                     → -1.570796
```

ASIN( ) 在 MySQL 3.21.8 中引入。

■ ATAN(*x*) 返回 *x* 的反正切值，如果 *x* 不在 -1 到 1 的取值范围内，返回 NULL。

```
ATAN(1)                      → 0.785398
ATAN(0)                      → 0.000000
ATAN(-1)                     → -0.785398
```

ASIN( ) 在 MySQL 3.21.8 中引入。

■ ATAN2(*x*, *y*) 与 ATAN(*x*) 类似，但它根据参数的符号决定返回值的象限。

```
ATAN2(1,1)                   → 0.785398
ATAN2(1,-1)                  → 2.356194
ATAN2(-1,1)                  → -0.785398
ATAN2(-1,-1)                 → -2.356194
```

■ GEILING(*x*) 返回不小于*x* 的最小整数。

```
CEILING(3.8)                 → 4
CEILING(-3.8)                → -3
```

■ COS(*x*) 返回 *x* 的余弦值，*x* 以弧度表示。

```
COS(0)                       → 1.000000
COS(PI())                    → -1.000000
COS(PI()/2)                  → 0.000000
```

COS( ) 在 MySQL 3.21.8 中引入。

■ COT(*x*) 返回 *x* 的余切值，*x* 以弧度表示。

```
COT(PI()/2)                  → 0.00000000
COT(PI()/4)                  → 1.00000000
```

COT( ) 在 MySQL 3.21.16 中引入。

■ DEGREES(*x*) 返回 *x* 的从弧度转换为度的值。

```
DEGREES(PI())                → 180
DEGREES(PI()*2)              → 360
DEGREES(PI()/2)              → 90
```

```
DEGREES(-PI())                          → -180
```

DEGREES( ) 在 MySQL 3.21.16 中引入。

■ EXP(*x*) 返回 *ex*，其中 *e* 为自然对数的底。

```
EXP(1)                                  → 2.718282
EXP(2)                                  → 7.389056
EXP(-1)                                 → 0.367879
1/EXP(1)                                → 0.36787944
```

■ FLOOR(*x*) 返回不大于 *x* 的最大整数（即，截去 *x* 的小数部分）。

```
FLOOR(3.8)                              → 3
FLOOR(-3.8)                             → -4
```

■ LOG(*x*) 返回 *x*的自然对数（底为 *e*）。

```
LOG(0)                                  → NULL
LOG(1)                                  → 0.000000
LOG(2)                                  → 0.693147
LOG(EXP(1))                             → 1.000000
```

可利用 LOG( ) 来计算底为任意值 *b* 的 *x* 的对数为 LOG(*x*)/LOG(*b*)。

```
LOG(100)/LOG(10)                        → 2.00000000
LOG10(100)                              → 2.000000
```

■ LOG10(*x*) 返回底为 10 的 *x* 的对数。

```
LOG10(0)                                → NULL
LOG10(10)                               → 1.000000
LOG10(100)                              → 2.000000
```

■ MOD(*m*, *n*) 与 *m* % *n* 相同，请参阅“ 算术运算符 ”

■ PI( ) 返回π值。

```
PI()                                    → 3.141593
```

PI( ) 在MySQL 3.21.8 中引入。

■ POW(*x*, *y*) 返回 x^y，即x 的 y 次幂。

```
POW(2,3)                                → 8.000000
POW(2,-3)                               → 0.125000
POW(4,.5)                               → 2.000000
POW(16,.25)                             → 2.000000
```

■ POWER(*x*, *y*) 这是 POW( ) 的同义词。在 MySQL 3.21.16 中引入。

■ RADIANS(*x*) 返回 *x* 的从角度转换为弧度的值。

```
RADIANS(0)                              → 0
RADIANS(360)                            → 6.28319
RADIANS(-360)                           → -6.28319
```

RADIANS( ) 在 MySQL 3.21.16 中引入。

■ RAND()

```
RAND(n)
```

RAND() 返回一个范围在 0.0 到 1.0 之间的浮点数随机值。RAND(*n*) 功能相同，但利用 *n* 69作为随机数的种子值。所有相同 *n* 值的调用都返回相同的结果，在需要可重复的数字序列时，可利用这个特性。

```
RAND(10)                                → 0.181091
RAND(10)                                → 0.181091
```

```
RAND()                                  → 0.117195
RAND()                                  → 0.358596
RAND(10)                                → 0.181091
```

请注意例子中给出参数和不给参数的那些连续调用所得到的随机数。

■ ROUND(*x*)

```
ROUND(x,d)
ROUND(x)
```

返回 *x* 四舍五入为一个整数的值。ROUND(*x*, *d*) 返回 *x* 四舍五入为一个 *d* 位小数的值。如果 *d* 为 *0*，结果中没有小数点或小数部分。

```
ROUND(15.3)                             → 15
ROUND(15.5)                             → 16
ROUND(-33.27834,2)                      → -33.28
ROUND(1,4)                              → 1.0000
```

■ SIGN(*x*) 根据 x 的值为负、零或正，分别返回 -1、0 或 1。

```
SIGN(15.803)                            → 1
SIGN(0)                                 → 0
SIGN(-99)                               → -1
```

■ SIN(*x*) 返回 *x* 的正弦值，*x* 以弧度表示。

```
SIN(0)                                  → 0.000000
SIN(PI())                               → 0.000000
SIN(PI()/2)                             → 1.000000
```

■ SIN( ) 在 MySQL 3.21.8 中引入。

■ SQRT(*x*) 返回 *x* 的非负平方根。

```
SQRT(625)                               → 25.000000
SQRT(2.25)                              → 1.500000
SQRT(-1)                                → NULL
```

■ TAN(*x*) 返回以弧度表示的 *x* 的正切值。

```
TAN(0)                                  → 0.000000
TAN(PI()/4)                             → 1.000000
```

TAN( ) 在 MySQL 3.21.8 中引入。

■ TRUNCATE(*x*, *d*) 返回将 *x* 截成小数部分为 *d* 位的值。如果 *d* 为 *0*，结果无小数部分。如果 *d* 小于 *x* 中的小数位数，则用零填充所需的宽度。

```
TRUNCATE(1.23,1)                        → 1.2
TRUNCATE(1.23,0)                        → 1
TRUNCATE(1.23,4)                        → 1.2300
```

C.2.3 串函数

本节中多数函数返回串结果。而有的函数，如 LENGTH( )，以串作为参数，但返回数作为结果。对于那些根据串的位置处理串的函数，其第一个（最左）字符的位置为 1而不是 0。

■ ASCII(*str*) 返回串 *str* 最左边字符的 ASCII 码。如果 *str* 为空，则返回 0。如果 str 为 NULL 则返回 NULL。

```
ASCII("abc")                            → 97
ASCII("")                               → 0
```

```
ASCII(NULL)                                    → NULL
```

ASCII( ) 函数是在 MySQL 3.21.2 中引入的。

■ BIN(*n*) 以二进制串的形式返回 n 的值。下列两个例子是等价的：

```
BIN(n)
CONV(n,10,2)
```

详细信息请参阅 CONV( )。

■ CHAR(*n1, n2, ...*) 将各参数作为 ASCII 代码，返回各代码所代表字符连接成的字符串。忽略 NULL 参数。

```
CHAR(65)                                       → "A"
CHAR(97)                                       → "a"
CHAR(89,105,107,101,115,33)                    → "Yikes!"
```

CHAR( ) 函数是在 MySQL 3.21.0 中引入的。

■ CHARACTER_LENGTH(*str*) 此函数是 LINGTH( ) 的同义词。

■ COALESCE(*expr1, expr2, ...*) 返回列表 (*expr1, expr2, ...*) 中的第一个非空元素。

```
COALESCE(NULL,1/0,2,"a",45+97)                 → "2"
```

COALESCE( ) 是在 MySQL 3.23.3 中引入的。

■ CONCAT(*str1, str2, ...*) 返回由其所有参数连接而成的串。如果任一参数为 NULL，则返回 NULL。CONCAT( ) 可接受几个参数。

```
CONCAT("abc","def")                            → "abcdef"
CONCAT("abc")                                  → "abc"
CONCAT("abc",NULL)                             → NULL
CONCAT("Hello",", ","goodbye")                 → "Hello, goodbye"
```

■ CONV(*n, from_base, to_base*) 给出一个以 *from_base* 为基数的数 n，返回一个以 *to_base* 为基数表示的 *n* 的串。如果任一参数为 NULL，则结果为 NULL。*from_base* 与 *to_base* 应该是范围为2到36的整数。N 按 BIGINT 处理（64 位整数），但可能以串来指定，因为基数大于 10 的数可能包括非十进制数（这也是 CONV() 返回一个串的原因。对于基数为 11 到 36，其结果可能包含字符“ A ”到“ Z ”)。如果 n 在基数为 *from_base* 的数中不是一个合法的数，则结果为 0（例如，如果 *from_base* 为 16，而 n 为“ abcdefg ”，则结果为 0，因为“ g ”不是一个合法的十六进制数字)。

n 中的非十进制数字既可用大写也可以用小写表示。结果中的非十进制数字都以大写表示。缺省设置时，*n* 处理为一个无符号数。

将十六进制表示的14转换为二进制如下：

```
CONV("e",16,2)                                 → "1110"
```

将二进制表示的255转换为八进制如下：

```
CONV(11111111,2,8)                             → "377"
CONV("11111111",2,8)                           → "377"
```

CONV( ) 是在 MySQL 3.22.4 中引入的。

■ ELT(*n, str1, str2, ...*) 从串 *str1*、*str2*、... 中返回第*n*个串。如果没有*n*个串，或第*n*个串为 NULL 或*n*为 NULL，则返回 NULL。第二个串的下标为1。ELT( ) 与 FIELD( ) 互为反函数。

```
ELT(3,"a","b","c","d","e")                     → "c"
ELT(0,"a","b","c","d","e")                     → NULL
```

```
ELT(6,"a","b","c","d","e")                    → NULL.
ELT(FIELD("b","a","b","c"),"a","b","c")       → "b"
```

■ EXPORT_SET(*n, on, off, [separator, [bit_count]]*) 返回由串*on*和*off* 组成以串*separator* 分隔的串。*on*用于*n*中置1的每一位，*off* 用于*n*中置 0 的每一位。*bit_count* 表示*n*中测试的最大位数。缺省的*separator*串为逗号，缺省的*bit_count* 为 64。

```
EXPORT_SET(7,"1","-","",5)                    → "+++ --1 "
EXPORT_SET(0xa,"1","0","",6)                  → "010100"
EXPORT_SET(97,"Y","N",",",8)                  → "Y,N,N,N,N,Y,Y,N"
```

EXPORT_SET( ) 在 MySQL 3.23.2 中引入。

■ FIELD(*str, str1, str2, ...*)

在串列表 str*1*、str*2*、... 中查找 *str*，返回其位置。如果无匹配串或 *str* 为 NULL，则返回 0。第一个串的位置为 1。FIELD( ) 与 ELT( ) 互为反函数。

```
FIELD("b","a","b","c")                        → 2
FIELD("d","a","b","c")                        → 0
FIELD(NULL,"a","b","c")                       → 0
FIELD(ELT(2,"a","b","c"),"a","b","c")         → 2
```

■ FIND_IN_SET(*str, str_list)*

str_list 为一个包含以逗号分隔的子串的串（即，它类似一个 SET 值）。FIND_IN_SET( ) 返回 *str* 在*str_list*中的位置。如果*str*未出现在*str_list*中，则返回 0。如果任一参数为 NULL，则返回 NULL。第一个子串的位置为 1。

```
FIND_IN_SET("cow","moose,cow,pig")            → 2
FIND_IN_SET("dog","moose,cow,pig")            → 0
```

FIND_IN_SET( ) 在 MySQL 3.21.22 中引入。

■ FORMAT(*X, D*)

将数 X 转换为如“ nn,nn.nnn ”格式的带有 D 位十进制小数的串。如果 D 为 0，则结果无小数部分。

```
FORMAT(1234.56789,3)                          → "1,234.568"
FORMAT(999999.99,2)                           → "999,999.99"
FORMAT(999999.99,0)                           → "1,000,000"
```

注意最后一个例子中的四舍五入。

■ HEX(*n*)

以十六进制串形式返回 *n* 的值。下面两个表达式是等价的：

```
HEX(n)
CONV(n,10,16)
```

更多的信息请参阅 CONV( ) 的说明。

HEX( ) 是在 MySQL 3.22.4 中引入的。

■ INSERT(*str, pos, len, new_str)* 返回串*str*，其中用串*new_str* 替换从*pos* 开始 *len* 个字符的子串。如果*pos* 超出范围则返回原串，如果任一参数为 NULL，则返回 NULL。

```
INSERT("nighttime",6,4,"fall")                → "nightfall"
INSERT("sunshine",1,3,"rain or ")             → "rain or shine"
INSERT("sunshine",0,3,"rain or ")             → "sunshine"
```

■ INSTR(*str, substr*) INSTR( ) 与LOCATE( )的两参数形式相似，只是参数颠倒了。下面

两个表达式是等价的：

```
INSTR(str,substr)
LOCATE(substr,str)
```

■ LCASE(*str*) 返回串str，其中所有字符转换为小写字符。如果 *str*为NULL，则返回NULL。

```
LCASE("New York, NY")                    → "new york, ny"
LCASE(NULL)                              → NULL
```

■ LEFT(*str, len*) 返回串*str* 中最左边的 *len* 个字符，如果没有 *len* 那么多个字符，则返回整个串。如果 *str* 为 NULL，则返回 NULL。如果*len* 为 NULL，或小于1，则返回空串。

```
LEFT("my left foot", 2)                  → "my"
LEFT(NULL,10)                            → NULL
LEFT("abc",NULL)                         → ""
LEFT("abc",0)                            → ""
```

■ LENGTH(*str*) 返回串 str 的长度。

```
LENGTH("abc")                            → 3
LENGTH("")                               → 0
LENGTH(NULL)                             → NULL
```

■ LOCATE(*substr, str*)

LOCATE (*substr, str, pos*)

LOCATE()的两参数形式返回串*substr*在*str*中第一次出现的位置，如果*substr* 未出现在 str 中，则返回 0。如果任一参数为 NULL，则返回 NULL。如果给出位置参数*pos*，LOCATE( ) 从pos 开始查找 *substr*。查找是区分大小写的。

```
LOCATE("b","abc")                        → 2
LOCATE("b","ABC")                        → 0
```

■ LOWER(*str*) 此函数为 LCASE( ) 的同义词。

■ LPAD(*str, len, pad_str*) 返回一个串，此串由串*str*左边加上数个*pad_str*串使其长度达到*len*个字符。如果*str*的长度已经是*len*，则返回 *str*。

```
LPAD("abc",12,"def")                     → "defdefdefabc"
LPAD("abc",10,".")                       → ".......abc"
LPAD("abc",2,".")                        → "abc"
```

LPAD( ) 是在 MySQL 3.22.2 中引入的。

■ LTRIM(*str*) 返回删除了左边空格的 str 串，如果 str 为 NULL，则返回 NULL。

```
LTRIM("  abc  ")                         → "abc  "
```

■ MAKE_SET(*n, bit0_str, bit1_str, ...*) 基于整数*n*和串*bit0_str*、*bit1_str*、...，构造 SET 值（由逗号分隔的子串组成的一个串）。对于*n*值中置1的每位，相应的串包括在结果中（如果位 0 置 1，结果将包括 *bit0_str*，如此等等）。如果*n*为 0，则结果为空串。如果 n 为 NULL，则结果为 NULL。若串列表中任一参数为 NULL，则构造结果串时将其忽略。

```
MAKE_SET(8,"a","b","c","d","e")          → "d"
MAKE_SET(7,"a","b","c","d","e")          → "a,b,c"
MAKE_SET(2+16,"a","b","c","d","e")       → "b,e"
MAKE_SET(2|16,"a","b","c","d","e")       → "b,e"
```

```
MAKE_SET(-1,"a","b","c","d","e")                    → "a,b,c,d,e"
```

最后一个例子选择了每个串，因为－1 使所有位都置 1。

MAKE_SET( ) 是在 MySQL 3.22.2 中引入的。

■ MID(*str, pos, len*) 返回串*str*中从*pos*开始长度为*len*的子串。如果任一参数为 NULL，则返回 NULL。

```
MID("what a dull example",8,4)                      → "dull"
```

■ OCT(*n*) 以八进制形式的串返回 *n* 的值。下面列两个表达式是等价的：

```
OCT(n)
CONV(n,10,8)
```

详细信息请参阅 CON( ) 的介绍。

OCT( ) 是在 MySQL 3.22.4 中引入的。

■ OCTET_LENGTH(*str*) 此函数为 LENGTH( ) 的同义词。

■ POSITION(*substr* IN *str*) 此函数与 LOCATE( ) 的两参数形式类似。下面的表达式是等价的：

```
POSITION(substr IN str)
LOCATE(substr,str)
```

■ REPEAT(*str, n*) 返回由串*str*重复*n*次所得的串。如果 *n* 不是正数或任一参数为 NULL，则返回空串。

```
REPEAT("x",10)                                      → "xxxxxxxxxx"
REPEAT("abc",3)                                     → "abcabcabc"
```

REPEAT( ) 是在 MySQL 3.21.10 中引入的。

■ REPLACE(*str, from_str, to_str*) 返回一个串，此串为将 *str* 中所有子串 *from_str* 替换成子串 *to_str* 所得的串。如果任一参数为 NULL，则返回 NULL。如果 *to_str* 为空，则结果是删除 *from_str*。如果 *from_str* 为空，则 REPLACE( ) 返回*str* 不变。

```
REPLACE("abracadabra","a","oh")                     → "ohbrohcohdohbroh"
REPLACE("abracadabra","a","")                       → "brcdbr"
REPLACE("abracadabra","","x")                       → "abracadabra"
```

■ REVERSE(*str*) 返回一个颠倒串*str* 中所有字符得到的串。如果 *str* 为 NULL，则返回 NULL。

```
REVERSE("abracadabra")                              → "arbadacarba"
REVERSE("Madam, I'm Adam")                          → "madA m'I ,madaM"
```

REVERSE（） 是在 MySQL 3.21.19 中引入的。

■ RIGHT(*str, len*) 返回串*str*中最右边长度为*len*个字符的子串。如果没有 *len*个字符，则返回整个串。如果*str*为NULL，则返回NULL。如果*len*为NULL 或小于1，则返回空串。

```
RIGHT("rightmost",4)                                → "most"
```

■ RPAD(*str, len, pad_str*) 返回由串*str*右边补上数个*pad_str*串以达到*len*个字符长的串。如果*str*已包含*len*个字符，则返回*str*。

```
RPAD("abc",12,"def")                                → "abcdefdefdef"
RPAD("abc",10,".")                                  → "abc......."
RPAD("abc",2,".")                                   → "abc"
```

RPAD( ) 是在 MySQL 3.22.2 中引入的。

■ RTRIM(*str*) 返回删除了右边空格的 *str* 串，如果 *str* 为空，则返回空。

```
RTRIM("  abc  ")                    → "  abc"
```

■ SOUNDEX(*str*) 返回由串 *str* 计算出来的声音串。忽略*str* 中的非字母数字字符。超出范围“ A ”到“ Z ”以外的字符作为元音处理。

```
SOUNDEX("Cow")                      → "C000"
SOUNDEX("Cowl")                     → "C400"
SOUNDEX("Howl")                     → "H400"
SOUNDEX("Hello")                    → "H400"
```

■ SPACE(*n*) 返回由*n*个空格组成的串，如果 *n* 非负，则为空集，如果 *n* 为 NULL，则为 NULL。

```
SPACE(6)                            → "      "
SPACE(0)                            → ""
SPACE(NULL)                         → NULL
```

SPACE( ) 是在 MySQL 3.21.16 中引入。

■ SUBSTRING(*str, pos*)

```
SUBSTRING(str,pos,len)
SUBSTRING(str FROMpos)
SUBSTRING(str FROM pos FOR len)
```

返回串*str* 中从*pos* 开始的一个子串。如果给出 *len*参数，则返回该长度的一个子串，否则返回*str*中以*pos*开始的右边的子串。

```
SUBSTRING("abcdef",3)               → "cdef"
SUBSTRING("abcdef",3,2)             → "cd"
```

下列表达式是等价的：

```
SUBSTRING(str,pos,len)
SUBSTRING(str FROM pos FOR len)
MID(str,pos,len)
```

■ SUBSTRING_INDEX(*str, delim, n*) 返回串 *str*中的一个子串。如果 *n*为正数，SUBSTRING_INDEX( ) 查找 *delim* 分隔串的第*n*次出现，然后返回该分隔符左边的所有字符。如果*n*为负，则从串*str*的右边向左计数，查找*delim*的第*n*次出现，然后返回该分隔符右边的所有字符。如果在 *str*中未找到*delim*，则返回整个串。如果任意参数为 NULL，则返回 NULL。

```
SUBSTRING_INDEX("jar.jar","j",-2)   → "ar.jar"
SUBSTRING_INDEX(USER(),"@",1)       → "paul"
```

SUBSTRING_INDEX( ) 是在 MySQL 3.21.15 中引入的。

■ TRIM([[LEADING | TRAILING | BOTH] [*trim_str*] FROM] *str*) 从串*str*中去掉前导的和/或后跟的串 trim_str。如果指定 LEADING，则去掉前导的 *trim_str*。如果指定 TRAILING，则去掉后跟的trim_str。如果指定 BOTH，则去掉前导的和后跟的 *trim_str*。如果 LEADING、TRAILING 或 BOTH 都不给出，缺省为 BOTH。如果不给出 trim_str，则 TRIM( ) 去掉空格。

```
TRIM("^" FROM "^^^XYZ^^")           → "XYZ"
TRIM(LEADING "^" FROM "^^^XYZ^^")   → "XYZ^^"
TRIM(TRAILING "^" FROM "^^^XYZ^^")  → "^^^XYZ"
TRIM(BOTH "^" FROM "^^^XYZ^^")      → "XYZ"
```

```
TRIM(BOTH FROM "   ABC   ")          → "ABC"
TRIM("   ABC   ")                    → "ABC"
```

TRIM( ) 是在 MySQL 3.21.12 中引入的。

■ UCASE(*str*) 将串 *str* 的所有字符转换为大写。如果 str 为 NULL，则返回 NULL。

```
UCASE("New York, NY")                → "NEW YORK, NY"
UCASE(NULL)                          → NULL
```

■ UPPER(*str*) 此函数为 UCASE( ) 的同义词。

C.2.4 日期和时间函数

日期和时间函数取各种形式的参数。一般情况下，需要一个 DATE参数的函数也接受DATETIME或 TIMESTAMP 参数并且忽略值的时间部分。

■ ADDATE(*date*, INTERVAL *expr interval*) 此函数为 DATE_ADD( ) 的同义词。

■ CURDATE( ) 以“ *YYYY-MM-DD* ”格式返回一个串作为当前时间，根据使用环境不同，也可以返回 *YYYYMMDD* 格式的数值作为当前日期。

```
CURDATE()                            → "1999-08-10"
CURDATE() + 0                        → 19990810
```

合法的日期和时间值

如果未向日期和时间函数提供合法的日期和时间值，就不可能得到正确的结果。因此应该对参数进行仔细检查。

■ CURRENT_DATE 此函数是 CURDATE( ) 的同义词，注意没有圆括号。

■ CURRENT_TIME 此函数为 CURTIME( ) 的同义词，注意没有圆括号。

■ CURRENT_TIMESTAMP 此函数为 NOW( ) 的同义词，注意没有圆括号。

■ CURTIME() 以“ *hh:mm:ss* ”格式的串返回当前时间，根据上下文不同，也可以 *hhmmss* 格式的数返回当前时间。

```
CURTIME()                            → "16:41:01"
CURTIME() + 0                        → 164101
```

CURTIME( ) 是在 MySQL 3.21.12 中引入的。

■ DATE_ADD(*date*, INTERVAL e*xpr interval*) 给日期或日期和时间值 *date* 加上一个时间间隔，然后返回其结果。 *expr* 指定要增加（如果 *expr* 以 ‘ - ’ 号开始，则为减去）到*date*的时间值，而*interval*指定 *expr* 的类型。如果 *date* 为 DATE 值，则结果为 DATE 值，在结果的计算中不涉及与时间有关的值，否则，结果是一个 DATETIME 的值。如果 *date* 不是一个合法的日期，则返回 NULL。

```
DATE_ADD("1999-12-01",INTERVAL 1 YEAR)          → "2000-12-01"
DATE_ADD("1999-12-01",INTERVAL 60 DAY)          → "2000-01-30"
DATE_ADD("1999-12-01",INTERVAL -3 MONTH)        → "1999-09-01"
DATE_ADD("1999-12-01 08:30:00",INTERVAL 12 HOUR)  → "1999-12-01 20:30:00"
```

表C-2给出了允许的 interval 值、其意义以及每种间隔类型的格式。关键字 INTERVAL 和 interval 说明符可以大写或小写字符给出。

增加到日期的表达式 *expr* 可用数或串的形式指定，如果包含非数字字符，则必须以串的形式给出。分隔字符可以任意，有可能为任一标点符号：

```
DATE_ADD("1999-12-01",INTERVAL "2:3" YEAR_MONTH)  → "2002-03-01"
DATE_ADD("1999-12-01",INTERVAL "2-3" YEAR_MONTH)  → "2002-03-01"
```

*expr*值的各部分根据 interval说明符所期望的部分从右至左进行匹配。例如，HOUR_SECOND 期望的格式为“*hh:mm:ss*”。*expr* 值为“15:21”解释为“00:15:21”而不是“15:21:00”。

```
DATE_ADD("1999-12-01 12:00:00",INTERVAL "15:21" HOUR_SECOND)
                                                  → "1999-12-01 12:15:21"
```

表C-2 DATE_ADD( ) 间隔类型

类 型	说 明	值的格式
SECOND	秒	*ss*
MINUTE	分	*mm*
HOUR	时	*hh*
DAY	日	*DD*
MONTH	月	*MM*
YEAR	年	*YY*
MINUTE_SECOND	分和秒	*“mm:ss”*
HOUR_MINUTE	时和分	*“hh:mm”*
HOUR_SECOND	时、分、秒	*“hh:mm:ss”*
DAY_HOUR	天和小时	*“DD hh”*
DAY_MINUTE	天、小时和分	*“DD hh:mm”*
DAY_SECOND	天、小时、分和秒	*“DD hh:mm:ss”*
YEAR_MONTH	年和月	*“YY-MM”*

如果 interval 为 YEAR、MONTH 或 YEAR_MONTH 并且结果中天的部分大于结果月份中的天数，则天数设置为该月中最大的天数。

```
DATE_ADD("1999-12-31",INTERVAL 2 MONTH)          → "2000-02-29"
```

DATE_ADD 是在 MySQL 3.22.4 中引入的。

■ DATE_FORMAT(*date, format*) 根据格式串 format 格式化日期或日期和时间值 date，返回结果串。可用DATE_FORMAT( ) 来格式化 DATE 或 DATETIME 值，以便得到所希望的格式。

```
DATE_FORMAT("1999-12-01","%M %e, %Y")          → "December 1, 1999"
DATE_FORMAT("1999-12-01","The %D of %M")       → "The 1st of December"
```

表C-3给出了可用于格式串中的说明符。

自MySQL 3.23.0 以来，在格式代码前要求“%”字符。在更早的版本中，允许“%”出现，但它是可选的。出现在格式串中但未列在表中的字符被逐字地拷贝到结果串中。

如果对 DATE 值引用时间说明符，则该值的时间部分处理为“00:00:00”。

```
DATE_FORMAT("1999-12-01","%i")                 → "00"
```

DATE_FORMAT( ) 是在 MySQL 3.21.14 中引入的。

■ DATE_SUB(*date*, INTERVAL *expr interval*) 以与 DATE_ADD() 相同的方式求出日期数值，只是此函数从 date 中减去 expr 而非加上 expr。详细请参阅 DATE_ADD( )。

```
DATE_SUB("1999-12-01",INTERVAL 1 MONTH)          → "1999-11-01"
DATE_SUB("1999-12-01",INTERVAL "13-2" YEAR_MONTH) → "1986-10-01"
DATE_SUB("1999-12-01 04:53:12",INTERVAL "13-2" MINUTE_SECOND)
```

```
                                                    → "1999-12-01 04:40:10"
DATE_SUB("1999-12-01 04:53:12",INTERVAL "13-2" HOUR_MINUTE)
                                                    → "1999-11-30 15:51:12"
```

DATE_SUB( ) 是在 MySQL 3.22.4 中引入的。

表C-3 DATE_FORMAT( ) 格式说明符

说 明 符	说 明
%S, %s	两位数字形式的秒（00,01, . . ., 59）
%i	两位数字形式的分（00,01, . . ., 59）
%H	两位数字形式的小时，24 小时（00,01, . . ., 23）
%h, %I	两位数字形式的小时，12 小时（01,02, . . ., 12）
%k	数字形式的小时，24 小时（0,1, . . ., 23）
%l	数字形式的小时，12 小时（1, 2, . . ., 12）
%T	24 小时的时间形式（hh:mm:ss）
%r	12 小时的时间形式（hh:mm:ss AM 或 hh:mm:ss PM）
%p	AM 或 PM
%W	一周中每一天的名称（Sunday, Monday, . . ., Saturday）
%a	一周中每一天名称的缩写（Sun, Mon, . . ., Sat）
%d	两位数字表示月中的天数（00, 01, . . ., 31）
%e	数字形式表示月中的天数（1, 2， . . ., 31）
%D	英文后缀表示月中的天数（1st, 2nd, 3rd, . . .）
%w	以数字形式表示周中的天数（0=Sunday, 1=Monday, . . ., 6=Saturday）
%j	以三位数字表示年中的天数（001, 002, . . ., 366）
%U	周（0, 1, 52），其中 Sunday 为周中的第一天
%u	周（0, 1, 52），其中 Monday 为周中的第一天
%M	月名（January, February, . . ., December）
%b	缩写的月名（January, February, . . ., December）
%m	两位数字表示的月份（01, 02, . . ., 12）
%c	数字表示的月份（1, 2, . . ., 12）
%Y	四位数字表示的年份
%y	两位数字表示的年份
%%	直接值“ % ”

■ DAYNAME(*date*) 返回包含日期值 *date* 的周日名串。

```
DAYNAME("1999-12-01")                → "Wednesday"
DAYNAME("1900-12-01")                → "Saturday"
```

DAYNAME( ) 是在 MySQL 3.21.23 中引入的。

■ DAYOFMONTH(*date*) 返回日期值 *date* 在月中的天数的数值，范围从 1 到 31。

```
DAYOFMONTH("1999-12-01")             → 1
DAYOFMONTH("1999-12-25")             → 25
```

DAYMONTH( ) 是在 MySQL 3.21.22 中引入的。

■ DAYOFWEEK(*date*) 返回日期值 *date* 在每周中的天数的数值，按 ODBC 标准星期天为1，星期二为 2，. . .，星期六为 7。可参阅 WEEKDAY( ) 函数。

```
DAYOFWEEK("1999-12-05")              → 1
DAYNAME("1999-12-05")                → "Sunday"
DAYOFWEEK("1999-12-11")              → 7
DAYNAME("1999-12-11")                → "Saturday"
```

DAYWEEK( ) 是在 MySQL 3.21.15 中引入的。

■ DAYOFYEAR(*date*) 返回日期值 *date* 在年中的天数的数值，范围从 1 到 366。

```
DAYOFYEAR("1999-12-01")          → 335
DAYOFYEAR("2000-12-31")          → 366
```

DAYOFYEAR( ) 是在 MySQL 3.21.22 中引入的。

■ EXTRACT(*interval* FROM *datetime*) 返回由 *interval* 表示的日期和时间值 *datetime* 的组成部分，*interval* 可以是 DATE_ADD( ) 所允许的任一间隔说明符。

```
EXTRACT(YEAR FROM "1999-12-01 13:42:19")          → 1999
EXTRACT(MONTH FROM "1999-12-01 13:42:19")         → 12
EXTRACT(DAY FROM "1999-12-01 13:42:19")           → 1
EXTRACT(HOUR_MINUTE FROM "1999-12-01 13:42:19")   → 1342
EXTRACT(SECOND FROM "1999-12-01 13:42:19")        → 19
```

EXTRACT( ) 可用于 DATE 值，这时若提取时间将得到 NULL。

EXTRACT( ) 是在 MySQL 3.23.0 中引入的。

■ FROM_DAYS(*n*) 给此函数一个表示自 0 年以来的日期数的数值（一般由调用 TO_DAYS( ) 得到），返回相应的日期。

```
TO_DAYS("1999-12-01")            → 730454
FROM_DAYS(730454 + 3)            → "1999-12-04"
```

FROM_DAYS( ) 只打算用于公历（1582 年以来）所包含的日期。

■ FROM_UNIXTIME(*unix_timestamp*)

FROM_UNIXTIME(*unix_timestamp, format*)

给此函数一个 UNIX 时间戳值 *unix_timestamp*（如由 UNIX_TIMESTAMP( ) 所返回的值），返回一个以“*YYYY-MM-DD hh:mm:ss* ”格式表示的日期和时间值，或者根据使用环境不同返回一个 *YYYYMMDDhhmmss* 格式的数。如果给出 format 参数，则返回值被格式化为一个串，这与 DATE_FORMAT( ) 函数所做的一样。

```
FROM_UNIXTIME(934340541)         → "1999-08-10 22:02:21"
FROM_UNIXTIME(944028000)         → "1999-12-01 00:00:00"
FROM_UNIXTIME(944028000,"%Y")    → "1999"
```

FROM_UNIXTIME( ) 是在 MySQL 3.21.5 中引入的。而两个参数的形式是在 MySQL 3.21.8 中引入的。

■ HOUR(*time*) 返回时间值 *time* 中小时的值，范围为 0 到 23。

```
HOUR("12:31:58")                 → 12
```

HOUR( ) 是在 MySQL 3.21.22 中引入的。

■ MINUTE(*time*) 返回时间值 *time* 中分的值，范围为 0 到 59。

```
MINUTE("12:31:58")               → 31
```

MINUTE( ) 是在 MySQL 3.21.22 中引入的。

■ MONTH(*date*) 返回日期值 *date* 中月份的值，范围为 1 到 12。

```
MONTH("1999-12-01")              → 12
```

MONTH( ) 是在 MySQL 3.21.22 中引入的。

■ MONTHNAME(*date*) 返回包含日期值 *date* 中的月份名的串。

```
MONTHNAME("1999-12-01")          → "December"
```

MONTHNAME( ) 是在 MySQL 3.21.23 中引入的。

■ NOW() 返回“*YYYY-MM-DD hh:mm:ss*”格式的当前日期和时间串，或者根据不同的上下文返回 *YYYYMMDDhhmmss* 格式的当前日期和时间的数值。

```
NOW()                                    → "1999-08-10 18:51:43"
NOW() + 0                                → 19990810185143
```

■ PERIOD_ADD(*period, n*) 给时期值 *period* 加上 *n* 个月，返回其结果。返回值的格式为 *YYYYMM*。而 *period* 参数可能是 *YYYYMM* 或 *YYMM*。两者都不是日期值。

```
PERIOD_ADD(199902,12)                    → 200002
PERIOD_ADD(9902,-3)                      → 199811
```

■ PERIOD_DIFF(*period1, period2*) 求周期值参数之差，返回相差的月份数。两参数的格式可以是 *YYYYMM* 或 *YYMM*。两者都不是日期值。

```
PERIOD_DIFF(200002,199902)               → 12
PERIOD_DIFF(199811,9902)                 → -3
```

■ QUARTER(*date*) 返回日期值 *date* 中当年的季度值，范围为 1 到 4。

```
QUARTER("1999-12-01")                    → 4
QUARTER("2000-01-01")                    → 1
```

QUARTER( ) 是在 MySQL 3.21.22 中引入的。

■ SECOND(*time*) 返回时间值*time* 中的秒数，范围从 0 到 59。

```
SECOND("12:31:58")                       → 58
SECOND(123158)                           → 58
```

SECOND( ) 是在 MySQL 3.21.22 中引入的。

■ SEC_TO_TIME(*seconds*) 给出秒数 *seconds*，返回“*hh:mm:ss*”格式的时间串，或根据上下文不同返回 *hhmmss* 格式的数。

```
SEC_TO_TIME(29834)                       → "08:17:14"
SEC_TO_TIME(29834) + 0                   → 81714
```

SEC_TO_TIME( ) 是在 MySQL 3.21.5 中引入的。

■ SUBDATE(*date*, INTERVAL *expr interval*) 此函数为 DATE_SUB( ) 的同义词。

■ SYSDATE( ) 此函数为 NOW( ) 的同义词。

■ TIME_FORMAT(*time, format*) 根据格式串 *format* 格式化时间值*time*，返回结果串。这里的格式化串与 DATE_FORMAT( ) 函数所用的相同，但使用的只是那些与时间有关的说明符。其他说明产生 NULL 值或 0。

```
TIME_FORMAT("12:31:58","%H %i")          → "12 31"
TIME_FORMAT(123158,"%H %i")              → "12 31"
```

TIME_FORMAT( ) 是在 MySQL 3.21.3 中引入的。

■ TIME_TO_SEC(*time*) 给出一个表示所用去的时间值 time，给出相应的秒数。

```
TIME_TO_SEC("08:17:14")                  → 29834
```

TIME_TO_SEC( ) 是在 MySQL 3.21.16 中引入的。

■ TO_DAYS(*date*) 返回将日期值 *date*转换为自 0 年以来的天数值。此值可传递给 FROM_DAYS( ) 转换为一个日期值。

```
TO_DAYS("1999-12-01")                    → 730454
FROM_DAYS(730454 - 365)                  → "1998-12-01"
```

TODAYS( ) 只打算用于公历（1582 年以来）所包含的日期。

■ UNIX_TIMESTAMP( )

UNIX_TIMESTAMP(*date*)

无参数调用时，返回自 UNIX 时间点（“ 1970-01-01 00:00:00 ” GMT）以来的秒数。在用一个日期值参数 *date* 调用时，返回 UNIX 时间点到该日期间的秒数，日期 *date* 可以几种方式给出，这些方式分别为： DATE 或 DATETIME 串、TIMESTAMP 值，或本地时间的 *YYYYMMDD* 或 *YYMMDD* 格式的数。

```
UNIX_TIMESTAMP()                         → 934341073
UNIX_TIMESTAMP("1999-12-01")             → 944028000
UNIX_TIMESTAMP(991201)                   → 944028000
```

■ WEEK(*date*)

WEEK(*date, first*)

在用单个参数调用时，返回日期值 *date* 在当年中的星期数，其范围从 0 到 52。假定一周从星期天开始。在用两个参数调用时，返回相同的值，但 *first* 参数指出了一周从哪天开始。如果 first 为 0，则一周从星期天开始。如果 *first* 为 1，则一周从星期一开始。

```
WEEK("1999-12-05")                       → 49
WEEK("1999-12-05",0)                     → 49
WEEK("1999-12-05",1)                     → 48
```

WEEK( ) 是在 MySQL 3.21.22 中引入的。其两参数的形式是在 MySQL 3.22.1 中引入的。

■ WEEKDAY(*date*) 返回日期值 *date* 为星期几的数值。星期几的数值从星期一的 0 到星期天的 6，可参阅 DAYOFWEEK( ) 函数。

```
WEEKDAY("1999-12-05")                    → 6
DAYNAME("1999-12-05")                    → "Sunday"
WEEKDAY("1999-12-13")                    → 0
DAYNAME("1999-12-13")                    → "Monday"
```

■ YEAR(*date*) 返回日期值 *date* 中的年份，其范围从 1000 到 9999。

```
YEAR("1999-12-01")                       → 1999
```

YEAR( ) 是在 MySQL 3.21.22 中引入的。

C.2.5 汇总函数

汇总函数（summary function）也称为聚合函数（aggregate function）。它们根据一组值求出一个值。但结果值只根据选定行中非 NULL 值进行计算（也有例外，那就是 COUNT(*) 对所有的行进行计数）。汇总函数可用来汇总所有的列。或者在由其他列或列组合中的明确的值分组时，汇总多个值。请参阅第 1 章中的“ 生成汇总 ”一节。

对于该节中的例子，假定有一个表 my_table，含有一个整数列 my_col，该列有六行，其值分别为 1、3、5、7、9 和 NULL。

■ AVG(*expr*) 返回所选择行中所有非 NULL 值的 expr 的平均值。

```
SELECT AVG(my_col) FROM my_table         → 5.0000
SELECT AVG(my_col)*2 FROM my_table       → 10.0000
SELECT AVG(my_col*2) FROM my_table       → 10.0000
```

■ BIT_AND(*expr*) 返回所选择行中所有非 NULL 值的 expr 的按位 AND 值。

```
SELECT BIT_AND(my_col) FROM my_table          → 1
```

BIT_AND( ) 是在 MySQL 3.21.11 中引入的。

■ BIT_OR(*expr*) 返回所选择行中所有非 NULL 值的 expr 的按位 OR 值。

```
SELECT BIT_OR(my_col) FROM my_table           → 15
```

BIT_OR( ) 是在 MySQL 3.21.11 中引入的。

■ COUNT(*expr*) 对于除“ * ”以外的任何参数，返回所选择集合中非 NULL 值的数的计数。对于参数“ * ”，返回选择集合中所有行的计数，不管行中是否包含 NULL 值。

```
SELECT COUNT(my_col) FROM my_table            → 5
SELECT COUNT(*) FROM my_table                 → 6
```

无 WHERE 子句的 COUNT(*) 是优化的，能非常快地返回 FROM 子句中指定的表中的记录数。在指定不止一个表时， COUNT(*) 返回各表中行数之积：

```
SELECT COUNT(*) FROM my_table AS m1, my_table AS m2
                                              → 36
```

■ MAX(*expr*) 返回所选择的行中所有非 NULL 值的 *expr* 的最大值。MAX( ) 也可以用于串，此时它返回字典序的最大值。

■ MIN(*expr*) 返回所选择的行中所有非 NULL 值的 *expr* 的最小值。MIN( ) 也可以用于串，此时它返回字典序的最小值。

```
SELECT MAX(my_col) FROM my_table              → 9
SELECT MIN(my_col) FROM my_table              → 1
```

■ STD(*expr*) 返回所选择行中所有非 NULL 值的 *expr* 的标准偏差。

```
SELECT STD(my_col) FROM my_table              → 2.8284
```

■ STDDEV(*expr*) 此函数为 STD( ) 的同义词。

■ SUM(*expr*) 返回所选择行中所有非 NULL 值的 *expr* 之和。

```
SELECT SUM(my_col) FROM my_table              → 25
```

C.2.6 杂项函数

本节中函数不属于上述所有类别。

■ BENCHMARK(*count, expr*) 重复求表达式 *expr* 的值 *count* 次。BENCHMARK() 是一个有些不寻常的函数，它是为在客户机程序中使用而设计的。它返回的值总是 0，因此返回的这个值没什么用途。其价值在于 mysql 显示查询结果后打印的占用时间：

```
mysql> SELECT BENCHMARK(100000,PASSWORD("secret"));
+------------------------------------+
| BENCHMARK(100000,PASSWORD("secret")) |
+------------------------------------+
|                                  0 |
+------------------------------------+
1 row in set (1.63 sec)
```

这里的时间只是服务器对表达式求值有多快的一个大致的估计，因为它表示的是客户机上的背景时间，而不是服务器上的 CPU 时间。这个时间受诸如服务器上的负载、此查询到达

时服务器是否处于可运行状态或换出等因素的影响。可以多次执行以观察代表性的值是什么。

BENCHMARK( ) 是在 MySQL 3.22.15 中引入的。

■ BIT_COUNT(*n*) 返回参数中置1的位数，返回值按 BIGINT 值对待（64 位整数）。

```
BIT_COUNT(0)                    → 0
BIT_COUNT(1)                    → 1
BIT_COUNT(2)                    → 1
BIT_COUNT(7)                    → 3
BIT_COUNT(-1)                   → 64
```

■ DATABASE() 返回当前数据库名串，如果不存在当前数据库，则返回空串。

```
DATABASE()                      → "samp_db"
```

■ DECODE(*str, password*) 给出一个作为调用 ENCODE( ) 所得结果的加密串 str，利用口令 password 串对其进行解密，并返回此结果串。

```
DECODE(ENCODE("secret","scramble"),"scramble")    → "secret"
```

■ ENCODE(*str, password*) 利用口令串 *password* 对串*str* 进行加密，返回二进制串形式的结果。此串可用 DECODE( ) 进行解码。因为加密后的结果是一个二进制串，因此若希望将它存储起来，应该利用 BLOB 列类型的列。

■ ENCRYPT(*str*)

ENCRYPT(*str, salt*)

对串 str 进行加密并返回结果串。这是一种不可逆的加密。如果给出 *salt* 参数，它应该是一个两字符的串。通过指定 *salt* 值，串的加密结果每次都相同。如果不给出 *salt* 参数，对 ENCRYPT( ) 的调用随时间不同产生不同的结果。

```
ENCRYPT("secret","AB")          → "ABS5SGh1EL6bk"
ENCRYPT("secret","AB")          → "ABS5SGh1EL6bk"
ENCRYPT("secret")               → "z1oPSN18WRwFA"
ENCRYPT("secret")               → "12FJgqDtVOg7Q"
```

ENCRYPT( ) 利用了 UNIX 的 crypt( ) 系统调用，并在两个方面依赖于它。首先，如果 crypt( ) 在您的系统上不可用，ENCRYPT( ) 总是返回 NULL。其次，ENCRYPT( ) 服从于 crypt( ) 在其上运行的系统的处理方式。特别是，crypt( ) 在某些系统上只处理串的前 8 个字符。

ENCRYPT( ) 是在 MySQL 3.21.12 中引入的。自 MySQL 3.22 以来，salt 可长于两个字符。

■ GET_LOCK(*str, timeout*) GET_LOCK( ) 与 RELEASE_LOCK( ) 联合使用以完成询问（协同）锁定。可利用这两个函数来编写基于一个协议锁名的状态进行协作的应用程序。

利用串 str 指定的锁名与 timeout 秒的超时值对 GET_LOCK( ) 进行调用。如果在规定的时间内成功获得相应的锁，则返回 1；如果由于超时使获得锁的企图失败，则返回 0；如果出错，则返回 NULL。timeout 值决定企图获得锁需要等待多长时间，而不是决定锁的持续时间。在获得锁后，该锁保持有效直到释放。

下面的调用请求一个名为“ Nellie ”的锁，对其等待 10 秒：

GET_LOCK(“ Nellie ”, 10)

此锁仅对该串自身起作用。它不锁定数据库、表或表中的任何行或列。换句话说就是，这个锁不阻止其他客户机对数据库表执行操作，这也就是 GET_LOCK( ) 锁定只是询问锁的原

因，它允许其他协同客户机决定该锁是否起作用。

对某个名称具有锁的客户机将阻塞其他客户机锁定该名称的企图（或多线程客户机内的其他线程的企图，这个多线程客户机维持对服务器的多个连接）。假定客户机 1锁定了串“ Nellie ”。如果客户机2企图锁定相同的串，它将阻塞直到客户机 1释放该锁或超时为止。在前一种情形中，客户机2将成功获得该锁，而后一种情况将会失败。

因为两个客户机不能同时锁定一个给定的串，所以在一个名称上有约定的数个应用程序可利用该名称的锁状态作为一个指示器，以指示何时执行与该名称有关的操作能保证安全。例如，可以基于某个表中的某行的唯一键值构造一个锁名，使得能够协同锁定该行。

通过用锁名调用 RELEASE_LOCK( ) 直接释放锁：

```
RELEASE_LOCK(“ Nellie ”)
```

如果成功地释放该锁， RELEASE_LOCK( ) 返回1；如果该锁为其他连接所占有（只能释放自己所拥有的锁），返回0；如果不存在这样的锁，返回 NULL。

客户机连接所占有的锁在该连接终止时，或者如果该客户机发布另一 GET_LOCK( ) 调用时，会自动释放（一个客户机连接一次只能锁定一个串）。上述的后一种情形中，所占用的锁在获得新锁前释放，即使新锁名与旧锁名相同也是如此。

GET_LOCK(str, 0) 可用作一个简单的查询，不用等待就可确定 str 上的锁是否有效（如果 str 当前未被锁定，这当然会锁住此串，不过要记住恰当地调用 RELEASE_LOCK( ) 来释放它）。

GET_LOCK( ) 是在 MySQL 3.21.7 中引入的。

■ LAST_INSERT_ID( )

```
LAST_INSERT_ID(expr)
```

无参数时，返回最近在当前服务器会话中生成的 AUTO_INCREMENT 值，如果未曾生成过这样的值，则返回 0。在有参数的调用时，可将此函数用于 UPDATE 语句。其结果以与自动生成的值相同的方式处理，这对于生成序列是非常有用的。

更详细的内容请参阅第 2 章。对于这两种形式的 LAST_INSERT_ID( )，其值都是由服务器在每次连接的基础上维持的，而且将不被其他客户机更改，即使那些导致建立新自动生成值的客户机也不能更改。

带参数的 LAST_INSERT_ID( ) 的形式是在 MySQL 3.22.9 中引入的。

■ LOAD_FILE(*file_name*) 此函数读文件 *file_name* 并将其内容作为一个串返回。文件必须位于服务器上，必须以绝对（完全）路径名指定，而且必须是完全可读的，以保证不用试图读取一个受保护的文件。因为该文件必须在服务器上，所以您必须具有 file 权限。如果上述任一条件不满足， LOAD_FILE( ) 将返回 NULL。

LOAD_FILE( ) 是在 MySQL 3.23.0 中引入的。

■ MD5(*str*) 基于 RSA DATA Security, Inc. MD5 Message_Digest 算法，计算串*str*的校验和。返回一个由 32 位十六进制数字组成的串。

```
MD5("secret")                    → "5ebe2294ecd0e0f08eab7690d2a6ee69"
```

MD5( ) 是在 MySQL 3.23.2 中引入的。

■ PASSWORD(*str*) 给出一个串*str*，计算并返回 MySQL 授权表中所用形式的加密口令串。这是一种不可逆的加密。

```
PASSWORD("secret")                    → "428567f408994404"
```

注意，PASSWORD() 不采用与 UNIX 加密用户账号口令所用的加密算法相同的算法。如果需要那种加密，可使用 ENCRYPT( )。

- RELEASE_LOCK(*str*) RELEASE_LOCK( ) 与 GET_LOCK( ) 联合使用。详细内容请参阅 GET_LOCK( ) 的描述。

RELEASE_LOCK( ) 是在 MySQL 3.21.7 中引入的。

- SESSION_USER( ) 此函数为 USER( ) 的同义词。
- SYSTEM_USER( ) 此函数为 USER( ) 的同义词。
- USER( ) 返回一个表示当前客户机用户名的串。自 MySQL 3.22.1 以来，此串的形式为user@host，其中 user 为用户名，而 host 为客户机通过其连接到服务器的主机名。

```
USER()                                → "paul@localhost"
SUBSTRING_INDEX(USER(),"@",1)         → "paul"
SUBSTRING_INDEX(USER(),"@",-1)        → "localhost"
```

- VERSION( ) 返回描述服务器版本的串，例如，“ 3.22.25.log ”。可能看到的跟在服务器版本号后的后缀为 -log（登录打开）、-debug（服务器正运行调试模式）或 -demo（服务器正按演示模式运行）。

```
VERSION()                             → "3.23.1-alpha-log"
```

VERSION( ) 是在 MySQL 3.21.13 中引入的。