操作符优先级

以下列表显示了操作符优先级的由低到高的顺序。排列在同一行的操作符具有相同的优先级。

:=

||, OR, XOR

&&, AND

NOT

BETWEEN, CASE, WHEN, THEN, ELSE

=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN

|

&

<<, >>

-, +

*, /, DIV, %, MOD

^

- (一元减号), ~ (一元比特反转)

!

BINARY, COLLATE

注释：假如 HIGH_NOT_PRECEDENCE SQL 模式被激活，则 NOT 的优先级同 the !操作符相同。

比较函数和操作符

比较运算产生的结果为1(TRUE)、0 (FALSE)或 NULL。这些运算可用于数字和字符串。根据需要，字符串可自动转换为数字，而数字也可自动转换为字符串。

本章中的一些函数 (如 LEAST()和 GREATEST()) 的所得值不包括 1 (TRUE)、 0 (FALSE)和 NULL。然而，其所得值乃是基于按照下述规则运行的比较运算：

MySQL 按照以下规则进行数值比较：

- 若有一个或两个参数为 NULL，除非 NULL-safe <=>等算符，则比较运算的结果为 NULL。
- 若同一个比较运算中的两个参数都是字符串，则按照字符串进行比较。
- 若两个参数均为整数，则按照整数进行比较。
- 十六进制值在不需要作为数字进行比较时，则按照二进制字符串进行处理。
- 假如参数中的一个为 TIMESTAMP 或 DATETIME 列，而其它参数均为常数， 则在进行比较前将常数转为 timestamp。这样做的目的是为了使 ODBC 的进行更加顺利。 注意，这不适合 IN() 中的参数！为了更加可靠，在进行对比时通常使用完整的 datetime/date/time 字符串。
- 在其它情况下，参数作为浮点数进行比较。

在默认状态下，字符串比较不区分大小写，并使用现有字符集(默认为 cp1252 Latin1，同时对英语也适合)。

为了进行比较，可使用 CAST() 函数将某个值转为另外一种类型。 使用 CONVERT() 将字符串值转为不同的字符集。

<=>

NULL-safe equal.这个操作符和=操作符执行相同的比较操作，不过在两个操作码均为 NULL 时，其所得值为1而不为 NULL，而当一个操作码为 NULL 时，其所得值为0而不为 NULL。

```
mysql> SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL;

       -> 1, 1, 0
```

IS boolean_value IS NOT boolean_value

根据一个布尔值来检验一个值，在这里，布尔值可以是 TRUE、FALSE 或 UNKNOWN。

```
mysql> SELECT 1 IS TRUE, 0 IS FALSE, NULL IS UNKNOWN;

       -> 1, 1, 1
```

IS NULL IS NOT NULL

检验一个值是否为 NULL。

```
mysql> SELECT 1 IS NULL, 0 IS NULL, NULL IS NULL;

       -> 0, 0, 1

mysql> SELECT 1 IS NOT NULL, 0 IS NOT NULL, NULL IS NOT NULL;

       -> 1, 1, 0
```

expr BETWEEN min AND max

假如 *expr* 大于或等于 *min* 且 *expr* 小于或等于 *max*,则 BETWEEN 的返回值为1,或是0。若所有参数都是同一类型,则上述关系相当于表达式 (*min* <= *expr* AND *expr* <= *max*)。其它类型的转换根据本章开篇所述规律进行，且适用于3种参数中任意一种。

expr NOT BETWEEN min AND max

这相当于 NOT(*expr* BETWEEN *min* AND *max*)。

COALESCE(value,...)

返回值为列表当中的第一个非 NULL 值，在没有非 NULL 值得情况下返回值为 NULL。

GREATEST(value1,value2,...)

当有2或多个参数时，返回值为最大(最大值的)参数。比较参数所依据的规律同 LEAST() 相同。在没有自变量为 NULL 的情况下，GREATEST() 的返回值为 NULL。

expr IN(value,...)

若 *expr* 为 IN 列表中的任意一个值，则其返回值为 1 ,否则返回值为0。假如所有的值都是常数，则其计算和分类根据 *expr* 的类型进行。这时，使用二分搜索来搜索信息。如 IN 值列表全部由常数组成，则意味着 IN 的速度非常之快。如 *expr* 是一个区分大小写的字符串表达式，则字符串比较也按照区分大小写的方式进行。 IN 列表中所列值的个数仅受限于 max_allowed_packet 值。

expr NOT IN (value,...)

这与 NOT (*expr* IN (*value*,...)) 相同。

ISNULL(expr)

如 *expr* 为 NULL，那么 ISNULL() 的返回值为 1，否则返回值为 0。

INTERVAL(N,N1,N2,N3,...)

假如 *N* < *N1*，则返回值为0；假如 *N* < *N2*等等，则返回值为1；假如 *N* 为 NULL，则返回值为 -1。所有的参数均按照整数处理。为了这个函数的正确运行，必须满足 *N1* < *N2* < *N3* <……< *Nn*。其原因是使用了二分查找(极快速)。

```
mysql> SELECT INTERVAL(23, 1, 15, 17, 30, 44, 200);

       -> 3

mysql> SELECT INTERVAL(10, 1, 10, 100, 1000);

       -> 2

mysql> SELECT INTERVAL(22, 23, 30, 44, 200);

       -> 0
```

```
LEAST(value1,value2,...)
```

在有两个或多个参数的情况下， 返回值为最小（最小值）参数。用一下规则将自变量进行对比：

- 假如返回值被用在一个 INTEGER 语境中，或是所有参数均为整数值，则将其作为整数值进行比较。
- 假如返回值被用在一个 REAL 语境中，或所有参数均为实值，则 将其作为实值进行比较。
- 假如任意一个参数是一个区分大小写的字符串，则将参数按照区分大小写的字符串进行比较。
- 在其它情况下，将参数作为区分大小写的字符串进行比较。

假如任意一个自变量为 NULL，则 LEAST() 的返回值为 NULL。

控制流程函数

```
CASE value

WHEN [compare-value] THEN result

[WHEN [compare-value] THEN result ...]

[ELSE result]

END
```

```
CASE

WHEN [condition] THEN result

[WHEN [condition] THEN result ...]

[ELSE result]

END
```

在第一个方案的返回结果中， *value=compare-value*。而第二个方案的返回结果是第一种情况的真实结果。如果没有匹配的结果值，则返回结果为 ELSE 后的结果，如果没有 ELSE 部分，则返回值为 NULL。

```
mysql> SELECT CASE 1 WHEN 1 THEN 'one'

    ->     WHEN 2 THEN 'two' ELSE 'more' END;

       -> 'one'

mysql> SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END;

       -> 'true'
```

```
mysql> SELECT CASE BINARY 'B'

    ->     WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;

        -> NULL
```

一个 CASE 表达式的默认返回值类型是任何返回值的相容集合类型，但具体情况视其所在语境而定。如果用在字符串语境中，则返回结果味字符串。如果用在数字语境中，则返回结果为十进制值、实值或整数值。

IF(expr1,expr2,expr3)

如果 *expr1* 是 TRUE (*expr1* <> 0 and *expr1* <> NULL)，则 IF() 的返回值为 *expr2*;否则返回值则为 *expr3*。IF() 的返回值为数字值或字符串值，具体情况视其所在语境而定。

```
mysql> SELECT IF(1>2,2,3);

        -> 3

mysql> SELECT IF(1<2,'yes ','no');

        -> 'yes'

mysql> SELECT IF(STRCMP('test','test1'),'no','yes');

        -> 'no'
```

如果 *expr2* 或 *expr3*中只有一个明确是 NULL，则 IF() 函数的结果类型 为非 NULL 表达式的结果类型。

IF() (这一点在其被储存到临时表时很重要)的默认返回值类型按照以下方式计算：

表达式	返回值
expr2 或 *expr3* 返回值为一个字符串。	字符串
expr2 或 *expr3* 返回值为一个浮点值。	浮点
expr2 或 *expr3* 返回值为一个整数。	整数

IFNULL(expr1,expr2)

假如 *expr1* 不为 NULL，则 IFNULL() 的返回值为 *expr1*;否则其返回值为 *expr2*。IFNULL() 的返回值是数字或是字符串，具体情况取决于其所使用的语境 。

NULLIF(expr1,expr2)

如果 *expr1* = *expr2* 成立，那么返回值为 NULL，否则返回值为 *expr1*。这和 CASE WHEN *expr1* = *expr2* THEN NULL ELSE *expr1* END 相同。

字符串函数

ASCII(str)

返回值为字符串 *str* 的最左字符的数值。假如 *str* 为空字符串，则返回值为 0。假如 *str* 为 NULL，则返回值为 NULL。 ASCII() 用于带有从 0到255的数值的字符。

BIN(N)

返回值为 *N*的二进制值的字符串表示，其中 *N*为一个 longlong (BIGINT) 数字。这等同于 CONV(*N*,10,2)。假如 *N* 为 NULL，则返回值为 NULL。

```
mysql> SELECT BIN(12);

       -> '1100'
```

BIT_LENGTH(str)

返回值为二进制的字符串 *str* 长度。

```
mysql> SELECT BIT_LENGTH('text');

       -> 32
```

CHAR(N,... [USING charset])

CHAR() 将每个参数 *N*理解为一个整数，其返回值为一个包含这些整数的代码值所给出的字符的字符串。NULL 值被省略。

```
mysql> SELECT CHAR(77,121,83,81,'76');

       -> 'MySQL'

mysql> SELECT CHAR(77,77.3,'77.3');

       -> 'MMM'
```

大于 255的 CHAR() 参数被转换为多结果字符。 例如，CHAR(256) 相当于 CHAR(1,0)，而 CHAR(256*256) 则相当于 CHAR(1,0,0)：

CHAR_LENGTH(str)

返回值为字符串 *str* 的长度，长度的单位为字符。一个多字节字符算作一个单字符。 *对于一个包含五个二字节字符集*， LENGTH() 返回值为 10，而 CHAR_LENGTH() 的返回值为5。

CHARACTER_LENGTH(str)

CHARACTER_LENGTH() 是 CHAR_LENGTH() 的同义词。

COMPRESS(string_to_compress)

压缩一个字符串。这个函数要求 MySQL 已经用一个诸如 zlib 的压缩库压缩过。 否则，返回值始终是 NULL。UNCOMPRESS() 可将压缩过的字符串进行解压缩。

压缩后的字符串的内容按照以下方式存储：

- 空字符串按照空字符串存储。
- 非空字符串未压缩字符串的四字节长度进行存储(首先为低字节)，后面是压缩字符串。如果字符串以空格结尾，就会在后加一个"."号，以防止当结果值是存储在 CHAR 或 VARCHAR 类型的字段列时，出现自动把结尾空格去掉的现象。(不推荐使用 CHAR 或 VARCHAR 来存储压缩字符串。最好使用一个 BLOB 列代替)。

CONCAT(str1,str2,...)

返回结果为连接参数产生的字符串。如有任何一个参数为 NULL，则返回值为 NULL。或许有一个或多个参数。 如果所有参数均为非二进制字符串，则结果为非二进制字符串。 如果自变量中含有任一二进制字符串，则结果为一个二进制字符串。一个数字参数被转化为与之相等的二进制字符串格式；若要避免这种情况，可使用显式类型 cast，例如： SELECT CONCAT(CAST(int_col AS CHAR), char_col)

```
mysql> SELECT CONCAT('My', 'S', 'QL');
       -> 'MySQL'
mysql> SELECT CONCAT('My', NULL, 'QL');
       -> NULL
mysql> SELECT CONCAT(14.3);
       -> '14.3'
```

CONCAT_WS(separator,str1,str2,...)

CONCAT_WS() 代表 CONCAT With Separator，是 CONCAT() 的特殊形式。 第一个参数是其它参数的分隔符。分隔符的位置放在要连接的两个字符串之间。分隔符可以是一个字符串，也可以是其它参数。如果分隔符为 NULL，则结果为 NULL。函数会忽略任何分隔符参数后的 NULL 值。

```
mysql> SELECT CONCAT_WS(',','First name','Second name','Last Name');
       -> 'First name,Second name,Last Name'
mysql> SELECT CONCAT_WS(',','First name',NULL,'Last Name');
       -> 'First name,Last Name'
```

CONCAT_WS() 不会忽略任何空字符串。 (然而会忽略所有的 NULL)。

CONV(N,from_base,to_base)

不同数基间转换数字。返回值为数字的 *N*字符串表示，由 *from_base* 基转化为 *to_base* 基。如有任意一个参数为 NULL，则返回值为 NULL。自变量 *N* 被理解为一个整数，但是可以被指定为一个整数或字符串。最小基数为 2，而最大基数则为 36。 If *to_base* 是一个负数，则 *N* 被看作一个带符号数。否则， *N* 被看作无符号数。 CONV() 的运行精确度为 64比特。

```
mysql> SELECT CONV('a',16,2);

       -> '1010'

mysql> SELECT CONV('6E',18,8);

       -> '172'

mysql> SELECT CONV(-17,10,-18);

       -> '-H'

mysql> SELECT CONV(10+'10'+'10'+0xa,10,10);

       -> '40'
```

ELT(N,str1,str2,str3,...)

若 *N* = 1，则返回值为 *str1*，若 *N* = 2，则返回值为 *str2* ，以此类推。 若 *N* 小于1或大于参数的数目，则返回值为 NULL。 ELT() 是 FIELD() 的补数。

```
mysql> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');

       -> 'ej'

mysql> SELECT ELT(4, 'ej', 'Heja', 'hej', 'foo');

      -> 'foo'
```

EXPORT_SET(bits,on,off[,separator[,number_of_bits]])

返回值为一个字符串，其中对于 *bits* 值中的每个位组，可以得到一个 *on* 字符串，而对于每个清零比特位，可以得到一个 *off* 字符串。*bits* 中的比特值按照从右到左的顺序接受检验 (由低位比特到高位比特)。字符串被分隔字符串分开(默认为逗号‘,’)，按照从左到右的顺序被添加到结果中。*number_of_bits* 会给出被检验的二进制位数 (默认为 64)。

```
mysql> SELECT EXPORT_SET(5,'Y','N',',',4);

       -> 'Y,N,Y,N'

mysql> SELECT EXPORT_SET(6,'1','0',',',10);
```

```
      -> '0,1,1,0,0,0,0,0,0,0'
```

FIELD(str,str1,str2,str3,...)

返回值为 *str1*, *str2*, *str3*,……列表中的 *str* 指数。在找不到 *str* 的情况下，返回值为 0。

如果所有对于 FIELD() 的参数均为字符串，则所有参数均按照字符串进行比较。如果所有的参数均为数字，则按照数字进行比较。否则，参数按照双倍进行比较。

如果 *str* 为 NULL，则返回值为0，原因是 NULL 不能同任何值进行同等比较。FIELD() 是 ELT() 的补数。

```
mysql> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo');
      -> 2
mysql> SELECT FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo');
      -> 0
```

FIND_IN_SET(str,strlist)

假如字符串 *str* 在由 *N* 子链组成的字符串列表 *strlist* 中， 则返回值的范围在 1到 *N* 之间 。一个字符串列表就是一个由一些被‘,’符号分开的自链组成的字符串。如果第一个参数是一个常数字符串，而第二个是 type SET 列，则 FIND_IN_SET() 函数被优化，使用比特计算。如果 *str* 不在 *strlist* 或 *strlist* 为空字符串，则返回值为 0。如任意一个参数为 NULL，则返回值为 NULL。 这个函数在第一个参数包含一个逗号(‘,’)时将无法正常运行。

```
mysql> SELECT FIND_IN_SET('b','a,b,c,d');
      -> 2
```

FORMAT(X,D)

将 number *X* 设置为格式 '#,###,###.##',以四舍五入的方式保留到小数点后 *D* 位,而返回结果为一个字符串。

HEX(N_or_S)

如果 *N_OR_S* 是一个数字，则返回一个 十六进制值 *N* 的字符串表示，在这里， *N* 是一个 longlong (BIGINT) 数。这相当于 CONV(N,10,16)。

如果 *N_OR_S* 是一个字符串，则返回值为一个 *N_OR_S* 的十六进制字符串表示， 其中每个 *N_OR_S* 里的每个字符被转化为两个十六进制数字。

```
mysql> SELECT HEX(255);
      -> 'FF'
mysql> SELECT 0x616263;
      -> 'abc'
mysql> SELECT HEX('abc');
```

```
       -> 616263
```

INSERT(str,pos,len,newstr)

返回字符串 *str*,其子字符串起始于 *pos* 位置和长期被字符串 *newstr* 取代的 *len* 字符。 如果 *pos* 超过字符串长度，则返回值为原始字符串。 假如 *len* 的长度大于其它字符串的长度，则从位置 *pos* 开始替换。若任何一个参数为 null，则返回值为 NULL。

```
mysql> SELECT INSERT('Quadratic', 3, 4, 'What');

       -> 'QuWhattic'

mysql> SELECT INSERT('Quadratic', -1, 4, 'What');

       -> 'Quadratic'

mysql> SELECT INSERT('Quadratic', 3, 100, 'What');

       -> 'QuWhat'
```

这个函数支持多字节字元。

INSTR(str,substr)

返回字符串 *str* 中子字符串的第一个出现位置。这和 LOCATE() 的双参数形式相同，除非参数的顺序被颠倒。

```
mysql> SELECT INSTR('foobarbar', 'bar');

       -> 4

mysql> SELECT INSTR('xbar', 'foobar');

       -> 0
```

这个函数支持多字节字元，并且只有当至少有一个参数是二进制字符串时区分大小写。

LCASE(str)

LCASE() 是 LOWER() 的同义词。

LEFT(str,len)

返回从字符串 *str* 开始的 *len* 最左字符。

```
mysql> SELECT LEFT('foobarbar', 5);

       -> 'fooba'
```

LENGTH(str)

返回值为字符串 *str* 的长度，单位为字节。一个多字节字符算作多字节。这意味着 对于一个包含5个2字节字符的字符串， LENGTH() 的返回值为 10,而 CHAR_LENGTH() 的返回值则为5。

```
mysql> SELECT LENGTH('text');

    -> 4
```

LOAD_FILE(file_name)

读取文件并将这一文件按照字符串的格式返回。 文件的位置必须在服务器上,你必须为文件制定路径全名，而且你还必须拥有 FILE 特许权。文件必须可读取，文件容量必须小于 max_allowed_packet 字节。

若文件不存在，或因不满足上述条件而不能被读取， 则函数返回值为 NULL。

```
mysql> UPDATE tbl_name

      SET blob_column=LOAD_FILE('/tmp/picture')

      WHERE id=1;
```

LOCATE(substr,str) , LOCATE(substr,str,pos)

第一个语法返回字符串 *str* 中子字符串 *substr* 的第一个出现位置。第二个语法返回字符串 *str* 中子字符串 *substr* 的第一个出现位置,起始位置在 *pos*。如若 *substr* 不在 *str* 中，则返回值为0。

```
mysql> SELECT LOCATE('bar', 'foobarbar');

    -> 4

mysql> SELECT LOCATE('xbar', 'foobar');

    -> 0

mysql> SELECT LOCATE('bar', 'foobarbar',5);

    -> 7
```

这个函数支持多字节字元，并且只有当至少有一个参数是二进制字符串时区分大小写。

LOWER(str)

返回字符串 *str* 以及所有根据最新的字符集映射表变为小写字母的字符 (默认为 cp1252 Latin1)。

```
mysql> SELECT LOWER('QUADRATICALLY');

    -> 'quadratically'
```

这个函数支持多字节字元。

LPAD(str,len,padstr)

返回字符串 *str*,其左边由字符串 *padstr* 填补到 *len* 字符长度。假如 *str* 的长度大于 *len*,则返回值被缩短至 *len* 字符。

```
mysql> SELECT LPAD('hi',4,'??');
       -> '??hi'
mysql> SELECT LPAD('hi',1,'??');
       -> 'h'
```

LTRIM(str)

返回字符串 *str* ，其引导空格字符被删除。

```
mysql> SELECT LTRIM('  barbar');
       -> 'barbar'
```

这个函数支持多字节字元。

MAKE_SET(bits,str1,str2,...)

返回一个设定值（一个包含被‘,’号分开的字字符串的字符串），由在 *bits* 组中具有相应的比特的字符串组成。*str1* 对应比特 0， *str2*对应比特1,以此类推。*str1*, *str2*, ...中的 NULL 值不会被添加到结果中。

```
mysql> SELECT MAKE_SET(1,'a','b','c');
       -> 'a'
mysql> SELECT MAKE_SET(1 | 4,'hello','nice','world');
       -> 'hello,world'
mysql> SELECT MAKE_SET(1 | 4,'hello','nice',NULL,'world');
       -> 'hello'
mysql> SELECT MAKE_SET(0,'a','b','c');
       -> ''
```

MID(str,pos,len)

MID(*str*,*pos*,*len*)是 SUBSTRING(*str*,*pos*,*len*)的同义词。

OCT(N)

返回一个 *N*的八进制值的字符串表示，其中 *N* 是一个 longlong (BIGINT) 数。这等同于 CONV(N,10,8)。若 *N* 为 NULL，则返回值为 NULL。

```
mysql> SELECT OCT(12);

       -> '14'
```

OCTET_LENGTH(str)

OCTET_LENGTH() 是 LENGTH() 的同义词。

ORD(str)

若字符串 *str* 的最左字符是一个多字节字符，则返回该字符的代码， 代码的计算通过使用以下公式计算其组成字节的数值而得出：

```
   (1st byte code)

+ (2nd byte code× 256)

+ (3rd byte code× 256²) ...
```

假如最左字符不是一个多字节字符，那么 ORD() 和函数 ASCII() 返回相同的值。

```
mysql> SELECT ORD('2');

       -> 50
```

POSITION(substr IN str)

POSITION(*substr* IN *str*)是 LOCATE(*substr*,*str*)同义词。

QUOTE(str)

引证一个字符串，由此产生一个在 SQL 语句中可用作完全转义数据值的结果。 返回的字符串由单引号标注，每例都带有单引号 (''')、 反斜线符号 ('\')、 ASCII NUL 以及前面有反斜线符号的 Control-Z。如果自变量的值为 NULL，则返回不带单引号的单词 “NULL”。

```
mysql> SELECT QUOTE('Don\'t!');

      -> 'Don\'t!'

mysql> SELECT QUOTE(NULL);

       -> NULL
```

REPEAT(str,count)

返回一个由重复的字符串 *str* 组成的字符串，字符串 *str* 的数目等于 *count* 。 若 *count* <= 0,则返回一个空字符串。若 *str* 或 *count* 为 NULL，则返回 NULL。

```
mysql> SELECT REPEAT('MySQL', 3);

       -> 'MySQLMySQLMySQL'
```

REPLACE(str,from_str,to_str)

返回字符串 *str* 以及所有被字符串 *to_str* 替代的字符串 *from_str* 。

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');

       -> 'WwWwWw.mysql.com'
```

这个函数支持多字节字元。

REVERSE(str)

返回字符串 *str* ，顺序和字符顺序相反。

```
mysql> SELECT REVERSE('abc');

       -> 'cba'
```

这个函数支持多字节字元。

RIGHT(str,len)

从字符串 *str* 开始，返回最右 *len* 字符。

```
mysql> SELECT RIGHT('foobarbar', 4);

       -> 'rbar'
```

这个函数支持多字节字元。

RPAD(str,len,padstr)

返回字符串 *str*,其右边被字符串 *padstr* 填补至 *len* 字符长度。假如字符串 *str* 的长度大于 *len*,则返回值被缩短到与 *len* 字符相同长度。

```
mysql> SELECT RPAD('hi',5,'?');

       -> 'hi???'
```

```
mysql> SELECT RPAD('hi',1,'?');
       -> 'h'
```

这个函数支持多字节字元。

RTRIM(str)

返回字符串 *str* ，结尾空格字符被删去。

```
mysql> SELECT RTRIM('barbar   ');
       -> 'barbar'
```

这个函数支持多字节字元。

SPACE(N)

返回一个由 *N* 间隔符号组成的字符串。

SUBSTRING(str,pos) ,

SUBSTRING(str FROM pos)

SUBSTRING(str,pos,len) ,

SUBSTRING(str FROM pos FOR len)

不带有 *len* 参数的格式从字符串 *str* 返回一个子字符串，起始于位置 *pos*。带有 *len* 参数的格式从字符串 *str* 返回一个长度同 *len* 字符相同的子字符串，起始于位置 *pos*。 使用 FROM 的格式为标准 SQL 语法。也可能对 *pos* 使用一个负值。假若这样，则子字符串的位置起始于字符串结尾的 *pos* 字符，而不是字符串的开头位置。在以下格式的函数中可以对 *pos* 使用一个负值。

```
mysql> SELECT SUBSTRING('Quadratically',5);
       -> 'ratically'

mysql> SELECT SUBSTRING('foobarbar' FROM 4);
       -> 'barbar'

mysql> SELECT SUBSTRING('Quadratically',5,6);
       -> 'ratica'

mysql> SELECT SUBSTRING('Sakila', -3);
       -> 'ila'

mysql> SELECT SUBSTRING('Sakila', -5, 3);
```

```
       -> 'aki'

mysql> SELECT SUBSTRING('Sakila' FROM -4 FOR 2);

       -> 'ki'
```

这个函数支持多字节字元。

注意，如果对 *len* 使用的是一个小于1的值，则结果始终为空字符串。

SUBSTR() 是 SUBSTRING() 的同义词。

SUBSTRING_INDEX(str,delim,count)

在定界符 *delim* 以及 *count* 出现前，从字符串 *str* 返回自字符串。若 *count* 为正值,则返回最终定界符（从左边开始）左边的一切内容。若 *count* 为负值，则返回定界符（从右边开始）右边的一切内容。

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);

       -> 'www.mysql'

mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);

       -> 'mysql.com'
```

这个函数支持多字节字元。

TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str) TRIM(remstr FROM] str)

返回字符串 *str* ， 其中所有 *remstr* 前缀和/或后缀都已被删除。若分类符 BOTH、LEADIN 或 TRAILING 中没有一个是给定的,则假设为 BOTH。 *remstr* 为可选项，在未指定情况下，可删除空格。

```
mysql> SELECT TRIM('  bar   ');

       -> 'bar'

mysql> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');

       -> 'barxxx'

mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');

       -> 'bar'

mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');

       -> 'barx'
```

这个函数支持多字节字元。

UCASE(str)

UCASE()是UPPER()的同义词。

UNCOMPRESS(string_to_uncompress)

对经COMPRESS()函数压缩后的字符串进行解压缩。若参数为压缩值，则结果为 NULL。这个函数要求 MySQL已被诸如zlib之类的压缩库编译过。否则,返回值将始终是 NULL。

```
mysql> SELECT UNCOMPRESS(COMPRESS('any string'));

       -> 'any string'

mysql> SELECT UNCOMPRESS('any string');

       -> NULL
```

UNCOMPRESSED_LENGTH(compressed_string)

返回压缩字符串压缩前的长度。

```
mysql> SELECT UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',30)));

       -> 30
```

UNHEX(str)

执行从HEX(*str*)的反向操作。就是说，它将参数中的每一对十六进制数字理解为一个数字，并将其转化为该数字代表的字符。结果字符以二进制字符串的形式返回。

```
mysql> SELECT UNHEX('4D7953514C');

       -> 'MySQL'

mysql> SELECT 0x4D7953514C;

       -> 'MySQL'

mysql> SELECT UNHEX(HEX('string'));

       -> 'string'

mysql> SELECT HEX(UNHEX('1267'));

       -> '1267'
```

UPPER(str)

返回字符串 *str*， 以及根据最新字符集映射转化为大写字母的字符 (默认为 cp1252 Latin1).

```
mysql> SELECT UPPER('Hej');

       -> 'HEJ'
```

该函数支持多字节字元。

数值函数

若发生错误，所有数学函数会返回 NULL。

DIV

整数除法。 类似于 FLOOR()，然而使用 BIGINT 算法也是可靠的。

```
mysql> SELECT 5 DIV 2;

       -> 2
```

ABS(X)

返回 *x* 的绝对值。

CRC32(expr)

计算循环冗余码校验值并返回一个 32比特无符号值。若参数为 NULL，则结果为 NULL。该参数应为一个字符串，而且在不是字符串的情况下会被作为字符串处理（若有可能）。

```
mysql> SELECT CRC32('MySQL');

       -> 3259397556

mysql> SELECT CRC32('mysql');

       -> 2501908538
```

EXP(X)

返回 e 的 *x* 乘方后的值(自然对数的底)。

```
mysql> SELECT EXP(2);
       -> 7.3890560989307
mysql> SELECT EXP(-2);
       -> 0.13533528323661
mysql> SELECT EXP(0);
       -> 1
```

FLOOR(X)

返回不大于 *x*的最大整数值 。

```
mysql> SELECT FLOOR(1.23);
       -> 1
mysql> SELECT FLOOR(-1.23);
       -> -2
```

注意，返回值会被转化为一个 BIGINT。

LN(X)

返回 *X* 的自然对数,即， *X*相对于基数 e 的对数。

```
mysql> SELECT LN(2);
       -> 0.69314718055995
mysql> SELECT LN(-2);
       -> NULL
```

这个函数同 LOG(*X*)具有相同意义。

LOG(X) LOG(B,X)

若用一个参数调用，这个函数就会返回 *X* 的自然对数。

若用两个参数进行调用，这个函数会返回 *X* 对于任意基数 *B* 的对数。

```
mysql> SELECT LOG(2,65536);
       -> 16
```

```
mysql> SELECT LOG(10,100);
       -> 2
```

LOG(*B*,*X*)就相当于 LOG(*X*) / LOG(*B*)。

LOG2(X)

返回 *X* 的基数为2的对数。

```
mysql> SELECT LOG2(65536);
       -> 16
mysql> SELECT LOG2(-100);
       -> NULL
```

对于查出存储一个数字需要多少个比特，LOG2()非常有效。这个函数相当于表达式 LOG(*X*) / LOG(2)。

LOG10(X)

返回 *X*的基数为10的对数。

```
mysql> SELECT LOG10(2);
       -> 0.30102999566398
mysql> SELECT LOG10(100);
       -> 2
mysql> SELECT LOG10(-100);
       -> NULL
```

LOG10(*X*)相当于 LOG(10,*X*)。

MOD(N,M) , N % M N MOD M

模操作。返回 *N* 被 *M*除后的余数。

```
mysql> SELECT MOD(234, 10);
       -> 4
mysql> SELECT 253 % 7;
       -> 1
```

```
mysql> SELECT MOD(29,9);
       -> 2
mysql> SELECT 29 MOD 9;
       -> 2
```

这个函数支持使用 BIGINT 值。

MOD() 对于带有小数部分的数值也起作用， 它返回除法运算后的精确余数：

```
mysql> SELECT MOD(34.5,3);
       -> 1.5
```

POW(X,Y) , POWER(X,Y)

返回 *X* 的 *Y* 乘方的结果值。

```
mysql> SELECT POW(2,2);
       -> 4
mysql> SELECT POW(2,-2);
       -> 0.25
```

RAND() RAND(N)

返回一个随机浮点值 *v* ，范围在 0到1之间 (即,其范围为 0≤ *v* ≤ 1.0)。若已指定一个整数参数 *N* ，则它被用作种子值，用来产生重复序列。

```
mysql> SELECT RAND();
       -> 0.9233482386203
mysql> SELECT RAND(20);
       -> 0.15888261251047
mysql> SELECT RAND(20);
       -> 0.15888261251047
mysql> SELECT RAND();
       -> 0.63553050033332
mysql> SELECT RAND();
       -> 0.70100469486881
```

```
mysql> SELECT RAND(20);

       -> 0.15888261251047
```

若要在 *i* ≤ *R* ≤ *j* 这个范围得到一个随机整数 *R*，需要用到表达式 FLOOR(*i* + RAND() * (*j*– *i* + 1))。例如， 若要在7到 12的范围（包括7和12）内得到一个随机整数,可使用以下语句：

```
SELECT FLOOR(7 + (RAND() * 6));
```

ROUND(X) ROUND(X,D)

返回参数 *X*,其值接近于最近似的整数。在有两个参数的情况下，返回 *X* ，其值保留到小数点后 *D*位，而第 *D*位的保留方式为四舍五入。若要接保留 *X*值小数点左边的 *D* 位，可将 *D* 设为负值。

```
mysql> SELECT ROUND(-1.23);

       -> -1

mysql> SELECT ROUND(-1.58);

       -> -2

mysql> SELECT ROUND(1.58);

       -> 2
```

```
mysql> SELECT ROUND(1.298, 1);

       -> 1.3

mysql> SELECT ROUND(1.298, 0);

       -> 1

mysql> SELECT ROUND(23.298, -1);

       -> 20
```

返回值的类型同 第一个自变量相同(假设它是一个整数、双精度数或小数)。这意味着对于一个整数参数,结果也是一个整数(无小数部分)。

当第一个参数是十进制常数时，对于准确值参数，ROUND()使用精密数学题库：

- 对于准确值数字，ROUND()使用“四舍五入”或“舍入成最接近的数”的规则:对于一个分数部分为 .5或大于 .5的值，正数则上舍入到邻近的整数值，负数则下舍入临近的整数值。(换言之,其舍入的方向是数轴上远离零的方向)。对于一个分数部分小于.5的值，正数则下舍入下一个整数值，负数则下舍入邻近的整数值，而正数则上舍入邻近的整数值。
- 对于近似值数字，其结果根据 C 库而定。在很多系统中，这意味着 ROUND()的使用遵循“舍入成最接近的偶数”的规则： 一个带有任何小数部分的值会被舍入成最接近的偶数整数。

SIGN(X)

返回参数作为-1、 0或1的符号，该符号取决于 *X* 的值为负、零或正。

```
mysql> SELECT SIGN(-32);
       -> -1
mysql> SELECT SIGN(0);
       -> 0
mysql> SELECT SIGN(234);
       -> 1
```

SQRT(X)

返回非负数 *X* 的二次方根。

```
mysql> SELECT SQRT(4);
       -> 2
mysql> SELECT SQRT(20);
       -> 4.4721359549996
mysql> SELECT SQRT(-16);
       -> NULL
```

TRUNCATE(X,D)

返回被舍去至小数点后 *D*位的数字 *X*。若 *D* 的值为 0,则结果不带有小数点或不带有小数部分。可以将 D 设为负数,若要截去(归零） X 小数点左起第 D 位开始后面所有低位的值.

```
mysql> SELECT TRUNCATE(1.223,1);
       -> 1.2
mysql> SELECT TRUNCATE(1.999,1);
       -> 1.9
mysql> SELECT TRUNCATE(1.999,0);
       -> 1
```

```
mysql> SELECT TRUNCATE(-1.999,1);
       -> -1.9
mysql> SELECT TRUNCATE(122,-2);
      -> 100
mysql> SELECT TRUNCATE(10.28*100,0);
      -> 1028
```

所有数字的舍入方向都接近于零。

日期和时间函数

用于日期值的函数通常会接受时间日期值而忽略时间部分。而用于时间值的函数通常接受时间日期值而忽略日期部分。

返回各自当前日期或时间的函数在每次询问执行开始时计算一次。这意味着在一个单一询问中，对诸如 NOW() 的函数多次访问总是会得到同样的结果(未达到我们的目的，单一询问也包括对存储程序或触发器和被该程序/触发器调用的所有子程序的调用)。这项原则也适用于 CURDATE()、 CURTIME()、 UTC_DATE()、UTC_TIME()、UTC_TIMESTAMP()，以及所有和它们意义相同的函数。

CURRENT_TIMESTAMP()、 CURRENT_TIME()、 CURRENT_DATE()以及 FROM_UNIXTIME()函数返回连接当前时区内的值，这个值可用作 time_zone 系统变量的值。此外， UNIX_TIMESTAMP()假设其参数为一个当前时区的时间日期值。

以下函数的论述中返回值的范围会请求完全日期。 若一个日期为“零” 值，或者是一个诸如'2001-11-00'之类的不完全日期， 提取部分日期值的函数可能会返回 0。 例如， DAYOFMONTH('2001-11-00')会返回0。

ADDDATE(date,INTERVAL expr type) ADDDATE(expr,days)

当被第二个参数的 INTERVAL 格式激活后， ADDDATE()就是 DATE_ADD()的同义词。相关函数 SUBDATE()则是 DATE_SUB()的同义词。对于 INTERVAL 参数上的信息 ，请参见关于 DATE_ADD()的论述。

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
       -> '1998-02-02'
```

ADDTIME(expr,expr2)

ADDTIME()将 *expr2*添加至 *expr* 然后返回结果。 *expr* 是一个时间或时间日期表达式，而 *expr2* 是一个时间表达式。

```
mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999',
    ->             '1 1:1:1.000002');
       -> '1998-01-02 01:01:01.000001'
```

CONVERT_TZ(dt,from_tz,to_tz)

CONVERT_TZ()将时间日期值 *dt* 从 *from_tz* 给出的时区转到 *to_tz* 给出的时区，然后返回结果值。

在从若 *from_tz* 到 UTC 的转化过程中，该值超出 TIMESTAMP 类型的被支持范围，那么转化不会发生。

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
       -> '2004-01-01 13:00:00'
```

注释：若要使用诸如 'MET'或 'Europe/Moscow'之类的指定时间区，首先要设置正确的时区表。

CURDATE()

将当前日期按照'YYYY-MM-DD'或 YYYYMMDD 格式的值返回，具体格式根据函数用在字符串或是数字语境中而定。

```
mysql> SELECT CURDATE();
       -> '1997-12-15'
mysql> SELECT CURDATE() + 0;
       -> 19971215
```

CURRENT_DATE CURRENT_DATE()

CURRENT_DATE 和 CURRENT_DATE()是的同义词.

CURTIME()

将当前时间以'HH:MM:SS'或 HHMMSS 的格式返回， 具体格式根据函数用在字符串或是数字语境中而定。

```
mysql> SELECT CURTIME();
       -> '23:50:26'
mysql> SELECT CURTIME() + 0;
       -> 235026
```

CURRENT_TIME, CURRENT_TIME()

CURRENT_TIME 和 CURRENT_TIME()是 CURTIME()的同义词。

CURRENT_TIMESTAMP, CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP 和 CURRENT_TIMESTAMP() 是 NOW() 的同义词。

DATE(expr)

提取日期或时间日期表达式 *expr* 中的日期部分。

```
mysql> SELECT DATE('2003-12-31 01:02:03');
       -> '2003-12-31'
```

DATEDIFF(expr,expr2)

DATEDIFF() 返回起始时间 *expr* 和结束时间 *expr2*之间的天数。*Expr* 和 *expr2* 为日期或 date-and-time 表达式。计算中只用到这些值的日期部分。

```
mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');
       -> 1
mysql> SELECT DATEDIFF('1997-11-30 23:59:59','1997-12-31');
       -> -31
```

DATE_ADD(date,INTERVAL expr type) DATE_SUB(date,INTERVAL expr type)

这些函数执行日期运算。*date* 是一个 DATETIME 或 DATE 值，用来指定起始时间。 *expr* 是一个表达式，用来指定从起始日期添加或减去的时间间隔值。 *Expr* 是一个字符串;对于负值的时间间隔，它可以以一个 '-' 开头。 *type* 为关键词，它指示了表达式被解释的方式。

关键词 INTERVA 及 *type* 分类符均不区分大小写。

以下表显示了 *type* 和 *expr* 参数的关系：

type 值	预期的 *expr* 格式
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'

HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

MySQL 允许任何 *expr* 格式中的标点分隔符。表中所显示的是建议的 分隔符。若 *date* 参数是一个 DATE 值，而你的计算只会包括 YEAR、MONTH 和 DAY 部分(即,没有时间部分),其结果是一个 DATE 值。否则，结果将是一个 DATETIME 值。

若位于另一端的表达式是一个日期或日期时间值 ， 则 INTERVAL *expr type* 只允许在 +操作符的两端。对于 -操作符， INTERVAL *expr type* 只允许在其右端，原因是从一个时间间隔中提取一个日期或日期时间值是毫无意义的。(见下面的例子)。

```
mysql> SELECT '1997-12-31 23:59:59' + INTERVAL 1 SECOND;

       -> '1998-01-01 00:00:00'

mysql> SELECT INTERVAL 1 DAY + '1997-12-31';

       -> '1998-01-01'

mysql> SELECT '1998-01-01' - INTERVAL 1 SECOND;

       -> '1997-12-31 23:59:59'
```

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',  INTERVAL 1 SECOND);

       -> '1998-01-01 00:00:00'

mysql> SELECT DATE_ADD('1997-12-31 23:59:59', INTERVAL 1 DAY);

       -> '1998-01-01 23:59:59'

mysql> SELECT DATE_ADD('1997-12-31 23:59:59',  INTERVAL '1:1' MINUTE_SECOND);

       -> '1998-01-01 00:01:00'

mysql> SELECT DATE_SUB('1998-01-01 00:00:00',  INTERVAL '1 1:1:1' DAY_SECOND);

       -> '1997-12-30 22:58:59'

mysql> SELECT DATE_ADD('1998-01-01 00:00:00',   INTERVAL '-1 10' DAY_HOUR);

       -> '1997-12-30 14:00:00'

mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);

       -> '1997-12-02'

mysql> SELECT DATE_ADD('1992-12-31 23:59:59.000002',   INTERVAL '1.999999' SECOND_MICROSECOND);
```

```
       -> '1993-01-01 00:00:01.000001'
```

若你指定了一个过于短的时间间隔值 （不包括 *type* 关键词所预期的所有时间间隔部分）， MySQL 假定你已经省去了时间间隔值的最左部分。 例如，你指定了一种类型的 DAY_SECOND， *expr* 的值预期应当具有天、 小时、分钟和秒部分。若你指定了一个类似 '1:10' 的值， MySQL 假定天和小时部分不存在，那么这个值代表分和秒。换言之， '1:10' DAY_SECOND 被解释为相当于 '1:10' MINUTE_SECOND。这相当于 MySQL 将 TIME 值解释为所耗费的时间而不是日时的解释方式。

假如你对一个日期值添加或减去一些含有时间部分的内容，则结果自动转化为一个日期时间值：

```
mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 DAY);

       -> '1999-01-02'

mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR);

       -> '1999-01-01 01:00:00'
```

假如你使用了格式严重错误的日期,则结果为 NULL。假如你添加了 MONTH、YEAR_MONTH 或 YEAR，而结果日期中有一天的日期大于添加的月份的日期最大限度，则这个日期自动被调整为添加月份的最大日期：

```
mysql> SELECT DATE_ADD('1998-01-30', INTERVAL 1 MONTH);

       -> '1998-02-28'
```

DATE_FORMAT(date,format)

根据 *format* 字符串安排 *date* 值的格式。

以下说明符可用在 *format* 字符串中：

说明符	说明
%a	工作日的缩写名称 (Sun..Sat)
%b	月份的缩写名称 (Jan..Dec)
%c	月份，数字形式(0..12)
%D	带有英语后缀的该月日期 (0th,1st,2nd,3rd, ...)
%d	该月日期,数字形式 (00..31)
%e	该月日期,数字形式(0..31)
%f	微秒 (000000..999999)
%H	小时(00..23)
%h	小时(01..12)
%I	小时 (01..12)
%i	分钟,数字形式 (00..59)
%j	一年中的天数 (001..366)
%k	小时 (0..23)
%l	小时 (1..12)
%M	月份名称 (January..December)
%m	月份,数字形式 (00..12)
%p	上午（AM）或下午（ PM）
%r	时间 , 12小时制 (小时 hh:分钟 mm:秒数 ss 后加 AM 或 PM)
%S	秒 (00..59)

%s	秒 (00..59)
%T	时间 , 24小时制 (小时 hh:分钟 mm:秒数 ss)
%U	周 (00..53),其中周日为每周的第一天
%u	周 (00..53),其中周一为每周的第一天
%V	周 (01..53),其中周日为每周的第一天 ;和 %X 同时使用
%v	周 (01..53),其中周一为每周的第一天 ;和 %x 同时使用
%W	工作日名称 (周日..周六)
%w	一周中的每日 (0=周日..6=周六)
%X	该周的年份，其中周日为每周的第一天,数字形式,4位数;和%V 同时使用
%x	该周的年份，其中周一为每周的第一天,数字形式,4位数;和%v 同时使用
%Y	年份,数字形式,4位数
%y	年份,数字形式 (2位数)
%%	'%'文字字符

所有其它字符都被复制到结果中，无需作出解释。

注意， '%'字符要求在格式指定符之前。

月份和日期说明符的范围从零开始，原因是 MySQL 允许存储诸如 '2004-00-00'的不完全日期.

```
mysql>SELECT  DATE_FORMAT('1997-10-04  22:23:00',  '%W  %M  %Y');

       -> 'Saturday October 1997'

mysql>SELECT  DATE_FORMAT('1997-10-04  22:23:00',  '%H:%i:%s');

       -> '22:23:00'

mysql>SELECT  DATE_FORMAT('1997-10-04  22:23:00',  '%D %y %a %d %m %b %j');

       -> '4th 97 Sat 04 10 Oct 277'

mysql>SELECT  DATE_FORMAT('1997-10-04  22:23:00',   '%H %k %I %r %T %S %w');

       -> '22 22 10 10:23:00 PM 22:23:00 00 6'

mysql>SELECT  DATE_FORMAT('1999-01-01',  '%X %V');

       -> '1998 52'
```

DAY(date)

DAY() 和 DAYOFMONTH() 的意义相同。

DAYNAME(date)

返回 *date* 对应的工作日名称。

```
mysql>  SELECT  DAYNAME('1998-02-05');
```

```
       -> '周四'
```

DAYOFMONTH(date)

返回 *date* 对应的该月日期，范围是从 1到31。

```
mysql> SELECT DAYOFMONTH('1998-02-03');
       -> 3
```

DAYOFWEEK(date)

返回 *date* (1 =周日, 2 =周一, ..., 7 =周六)对应的工作日索引。这些索引值符合 ODBC 标准。

```
mysql> SELECT DAYOFWEEK('1998-02-03');
       -> 3
```

DAYOFYEAR(date)

返回 *date* 对应的一年中的天数，范围是从 1到366。

```
mysql> SELECT DAYOFYEAR('1998-02-03');
       -> 34
```

EXTRACT(type FROM date)

EXTRACT()函数所使用的时间间隔类型说明符同 DATE_ADD()或 DATE_SUB()的相同,但它从日期中提取其部分，而不是执行日期运算。

```
mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');
      -> 1999
mysql> SELECT EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03');
      -> 199907
mysql> SELECT EXTRACT(DAY_MINUTE FROM '1999-07-02 01:02:03');
      -> 20102
mysql> SELECT EXTRACT(MICROSECOND     FROM '2003-01-02 10:30:00.00123');
       -> 123
```

GET_FORMAT(DATE|TIME|DATETIME, 'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL')

返回一个格式字符串。这个函数在同 DATE_FORMAT() 及 STR_TO_DATE() 函数结合时很有用。

第一个参数的3个可能值和第二个参数的5个可能值产生 15个可能格式字符串 (对于使用的说明符，请参见 DATE_FORMAT() 函数说明表)。

函数调用	结果
GET_FORMAT(DATE,'USA')	'%m.%d.%Y'
GET_FORMAT(DATE,'JIS')	'%Y-%m-%d'
GET_FORMAT(DATE,'ISO')	'%Y-%m-%d'
GET_FORMAT(DATE,'EUR')	'%d.%m.%Y'
GET_FORMAT(DATE,'INTERNAL')	'%Y%m%d'
GET_FORMAT(DATETIME,'USA')	'%Y-%m-%d- %H.%i.%s'
GET_FORMAT(DATETIME,'JIS')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'ISO')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'EUR')	'%Y-%m-%d- %H.%i.%s'
GET_FORMAT(DATETIME,'INTERNAL')	'%Y%m%d%H%i%s'
GET_FORMAT(TIME,'USA')	'%h:%i:%s %p'
GET_FORMAT(TIME,'JIS')	'%H:%i:%s'
GET_FORMAT(TIME,'ISO')	'%H:%i:%s'
GET_FORMAT(TIME,'EUR')	'%H.%i.%S'
GET_FORMAT(TIME,'INTERNAL')	'%H%i%s'

ISO 格式为 ISO 9075,而非 ISO 8601.

也可以使用 TIMESTAMP,这时 GET_FORMAT() 的返回值和 DATETIME 相同。

```
mysql> SELECT DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR'));

       -> '03.10.2003'

mysql> SELECT STR_TO_DATE('10.31.2003',GET_FORMAT(DATE,'USA'));

       -> '2003-10-31'
```

HOUR(time)

返回 *time* 对应的小时数。对于日时值的返回值范围是从 0到 23。

```
mysql> SELECT HOUR('10:05:03');

       -> 10
```

然而， TIME 值的范围实际上非常大,所以 HOUR 可以返回大于23的值。

```
mysql> SELECT HOUR('272:59:59');

       -> 272
```

LAST_DAY(date)

获取一个日期或日期时间值，返回该月最后一天对应的值。若参数无效，则返回 NULL。

```
mysql> SELECT LAST_DAY('2003-02-05');

       -> '2003-02-28'

mysql> SELECT LAST_DAY('2004-02-05');

       -> '2004-02-29'

mysql> SELECT LAST_DAY('2004-01-01 01:01:01');

       -> '2004-01-31'

mysql> SELECT LAST_DAY('2003-03-32');

       -> NULL
```

LOCALTIME, LOCALTIME()

LOCALTIME 及 LOCALTIME() 和 NOW() 具有相同意义。

LOCALTIMESTAMP, LOCALTIMESTAMP()

LOCALTIMESTAMP 和 LOCALTIMESTAMP() 和 NOW() 具有相同意义。

MAKEDATE(year,dayofyear)

给出年份值和一年中的天数值，返回一个日期。*dayofyear* 必须大于 0，否则结果为 NULL。

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);

       -> '2001-01-31', '2001-02-01'

mysql> SELECT MAKEDATE(2001,365), MAKEDATE(2004,365);

       -> '2001-12-31', '2004-12-30'

mysql> SELECT MAKEDATE(2001,0);
```

```
       -> NULL
```

MAKETIME(hour,minute,second)

返回由 *hour*、 *minute* 和 *second* 参数计算得出的时间值。

```
mysql> SELECT MAKETIME(12,15,30);
       -> '12:15:30'
```

MICROSECOND(expr)

从时间或日期时间表达式 *expr* 返回微秒值，其数字范围从 0到 999999。

```
mysql> SELECT MICROSECOND('12:00:00.123456');
       -> 123456
mysql> SELECT MICROSECOND('1997-12-31 23:59:59.000010');
       -> 10
```

MINUTE(time)

返回 *time* 对应的分钟数,范围是从 0到 59。

```
mysql> SELECT MINUTE('98-02-03 10:05:03');
       -> 5
```

MONTH(date)

返回 *date* 对应的月份，范围时从 1到 12。

```
mysql> SELECT MONTH('1998-02-03');
       -> 2
```

MONTHNAME(date)

返回 *date* 对应月份的全名。

```
mysql> SELECT MONTHNAME('1998-02-05');
       -> 'February '
```

NOW()

返回当前日期和时间值，其格式为 'YYYY-MM-DD HH:MM:SS'或 YYYYMMDDHHMMSS， 具体格式取决于该函数是否用在字符串中或数字语境中。

```
mysql> SELECT NOW();
       -> '1997-12-15 23:50:26'
mysql> SELECT NOW() + 0;
       -> 19971215235026
```

在一个存储程序或触发器内， NOW() 返回一个常数时间，该常数指示了该程序或触发语句开始执行的时间。这同 SYSDATE() 的运行有所不同。

PERIOD_ADD(P,N)

添加 *N* 个月至周期 *P* (格式为 YYMM 或 YYYYMM)，返回值的格式为 YYYYMM。注意周期参数 *P* *不是*日期值。

```
mysql> SELECT PERIOD_ADD(9801,2);
       -> 199803
```

PERIOD_DIFF(P1,P2)

返回周期 *P1*和 *P2* 之间的月份数。*P1* 和 *P2* 的格式应该为 YYMM 或 YYYYMM。注意周期参数 *P1*和 *P2* *不是*日期值。

```
mysql> SELECT PERIOD_DIFF(9802,199703);
       -> 11
```

QUARTER(date)

返回 *date* 对应的一年中的季度值，范围是从 1到 4。

```
mysql> SELECT QUARTER('98-04-01');
       -> 2
```

SECOND(time)

返回 *time* 对应的秒数,范围是从 0到59。

```
mysql> SELECT SECOND('10:05:03');
       -> 3
```

SEC_TO_TIME(seconds)

返回被转化为小时、 分钟和秒数的 *seconds* 参数值,其格式为 'HH:MM:SS' 或 HHMMSS，具体格式根据该函数是否用在字符串或数字语境中而定。

```
mysql> SELECT SEC_TO_TIME(2378);
       -> '00:39:38'
mysql> SELECT SEC_TO_TIME(2378) + 0;
       -> 3938
```

STR_TO_DATE(str,format)

这是 DATE_FORMAT() 函数的倒转。它获取一个字符串 *str* 和一个格式字符串 *format*。若格式字符串包含日期和时间部分，则 STR_TO_DATE() 返回一个 DATETIME 值， 若该字符串只包含日期部分或时间部分，则返回一个 DATE 或 TIME 值。

str 所包含的日期、时间或日期时间值应该在 *format* 指示的格式中被给定。对于可用在 *format* 中的说明符，请参见 DATE_FORMAT() 函数说明表。 所有其它的字符被逐字获取，因此不会被解释。若 *str* 包含一个非法日期、时间或日期时间值，则 STR_TO_DATE() 返回 NULL。同时，一个非法值会引起警告。

对日期值部分的范围检查在11.3.1节，“DATETIME、DATE 和 TIMESTAMP 类型”有详细说明。其意义是,例如,只要具体日期部分的范围时从 1到 31之间，则允许一个日期中的具体日期部分大于一个月中天数值。并且，允许“零”日期或带有0值部分的日期。

```
mysql> SELECT STR_TO_DATE('00/00/0000', '%m/%d/%Y');
       -> '0000-00-00'
mysql> SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y');
      -> '2004-04-31'
```

SUBDATE(date,INTERVAL expr type) SUBDATE(expr,days)

当被第二个参数的 INTERVAL 型式调用时， SUBDATE() 和 DATE_SUB() 的意义相同。对于有关 INTERVAL 参数的信息， 见有关 DATE_ADD() 的讨论。

```
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
       -> '1997-12-02'
mysql> SELECT SUBDATE('1998-01-02', INTERVAL 31 DAY);
       -> '1997-12-02'
```

第二个形式允许对 *days* 使用整数值。在这些情况下，它被算作由日期或日期时间表达式 *expr* 中提取的天数。

```
mysql> SELECT SUBDATE('1998-01-02 12:00:00', 31);
       -> '1997-12-02 12:00:00'
```

注意不能使用格式 "%X%V" 来将一个 year-week 字符串转化为一个日期，原因是当一个星期跨越一个月份界限时，一个年和星期的组合不能标示一个唯一的年和月份。若要将 year-week 转化为一个日期，则也应指定具体工作日：

```
mysql> select str_to_date('200442 Monday', '%X%V %W');

-> 2004-10-18
```

SUBTIME(expr,expr2)

SUBTIME() 从 *expr* 中提取 *expr2* ，然后返回结果。*expr* 是一个时间或日期时间表达式，而 *xpr2* 是一个时间表达式。

```
mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');

       -> '1997-12-30 22:58:58.999997'

mysql> SELECT SUBTIME('01:00:00.999999', '02:00:00.999998');

       -> '-00:59:59.999999'
```

SYSDATE()

返回当前日期和时间值，格式为'YYYY-MM-DD HH:MM:SS'或 YYYYMMDDHHMMSS， 具体格式根据函数是否用在字符串或数字语境而定。

在一个存储程序或触发器中， SYSDATE() 返回其执行的时间，而非存储成都或触发语句开始执行的时间。这个 NOW() 的运作有所不同。

TIME(expr)

提取一个时间或日期时间表达式的时间部分，并将其以字符串形式返回。

```
mysql> SELECT TIME('2003-12-31 01:02:03');

       -> '01:02:03'

mysql> SELECT TIME('2003-12-31 01:02:03.000123');

      -> '01:02:03.000123'
```

TIMEDIFF(expr,expr2)

TIMEDIFF() 返回起始时间 *expr* 和结束时间 *expr2* 之间的时间。 *expr* 和 *expr2* 为时间或 date-and-time 表达式,两个的类型必须一样。

```
mysql> SELECT TIMEDIFF('2000:01:01 00:00:00',   '2000:01:01 00:00:00.000001');

       -> '-00:00:00.000001'

mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',  '1997-12-30 01:01:01.000002');
```

```
       -> '46:58:57.999999'
```

TIMESTAMP(expr) , TIMESTAMP(expr,expr2)

对于一个单参数,该函数将日期或日期时间表达式 *expr* 作为日期时间值返回.对于两个参数,它将时间表达式 *expr2* 添加到日期或日期时间表达式 *expr* 中，将 theresult 作为日期时间值返回。

```
mysql> SELECT TIMESTAMP('2003-12-31');

       -> '2003-12-31 00:00:00'

mysql> SELECT TIMESTAMP('2003-12-31 12:00:00','12:00:00');

       -> '2004-01-01 00:00:00'
```

TIMESTAMPADD(interval,int_expr,datetime_expr)

将整型表达式 *int_expr* 添加到日期或日期时间表达式 *datetime_expr* 中。 *int_expr* 的单位被时间间隔参数给定，该参数必须是以下值的其中一个：FRAC_SECOND、SECOND、 MINUTE、 HOUR、 DAY、 WEEK、 MONTH、 QUARTER 或 YEAR。

可使用所显示的关键词指定 *Interval* 值，或使用 SQL_TSI_ 前缀。例如， DAY 或 SQL_TSI_DAY 都是正确的。

```
mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');

       -> '2003-01-02 00:01:00'

mysql> SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');

       -> '2003-01-09'
```

TIMESTAMPDIFF(interval,datetime_expr1,datetime_expr2)

返回日期或日期时间表达式 *datetime_expr1* 和 *datetime_expr2*the 之间的整数差。其结果的单位由 *interval* 参数给出。 *interval* 的法定值同 TIMESTAMPADD() 函数说明中所列出的相同。

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');

       -> 3

mysql> SELECT TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01');

       -> -1
```

TIME_FORMAT(time,format)

其使用和 DATE_FORMAT() 函数相同,然而 *format* 字符串可能仅会包含处理小时、分钟和秒的格式说明符。其它说明符产生一个 NULL 值或0。

若 *time* value 包含一个大于23的小时部分，则 %H 和 %k 小时格式说明符会产生一个大于0..23的通常范围的值。另一个小时格式说明符产生小时值模数12。

```
mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
       -> '100 100 04 04 4'
```

TIME_TO_SEC(time)

返回已转化为秒的 *time* 参数。

```
mysql> SELECT TIME_TO_SEC('22:23:00');
       -> 80580
mysql> SELECT TIME_TO_SEC('00:39:38');
       -> 2378
```

TO_DAYS(date)

给定一个日期 *date*,返回一个天数 (从年份0开始的天数)。

```
mysql> SELECT TO_DAYS(950501);
       -> 728779
mysql> SELECT TO_DAYS('1997-10-07');
       -> 729669
```

TO_DAYS() 不用于阳历出现(1582) 前的值，原因是当日历改变时，遗失的日期不会被考虑在内。

请记住， MySQL 使用11.3 节，“日期和时间类型”中的规则将日期中的二位数年份值转化为四位。例如， '1997-10-07'和 '97-10-07'被视为同样的日期:

```
mysql> SELECT TO_DAYS('1997-10-07'), TO_DAYS('97-10-07');
       -> 729669, 729669
```

对于1582年之前的日期(或许在其它地区为下一年),该函数的结果实不可靠的。

UTC_DATE, UTC_DATE()

返回当前 UTC 日期值，其格式为 'YYYY-MM-DD'或 YYYYMMDD，具体格式取决于函数是否用在字符串或数字语境中。

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
       -> '2003-08-14', 20030814
```

UTC_TIME, UTC_TIME()

返回当前 UTC 值，其格式为 'HH:MM:SS'或 HHMMSS，具体格式根据该函数是否用在字符串或数字语境而定。

```
mysql> SELECT UTC_TIME(), UTC_TIME() + 0;

       -> '18:07:53', 180753
```

UTC_TIMESTAMP, UTC_TIMESTAMP()

返回当前 UTC 日期及时间值，格式为 'YYYY-MM-DD HH:MM:SS'或 YYYYMMDDHHMMSS，具体格式根据该函数是否用在字符串或数字语境而定。

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;

       -> '2003-08-14 18:08:04', 20030814180804
```

WEEK(date[,mode])

该函数返回 *date* 对应的星期数。WEEK() 的双参数形式允许你指定该星期是否起始于周日或周一， 以及返回值的范围是否为从0到53或从1到53。若 *mode* 参数被省略，则使用 default_week_format 系统自变量的值。

以下表说明了 *mode* 参数的工作过程：d

	第一天		
Mode	工作日	范围	Week 1为第一周 ...
0	周日	0-53	本年度中有一个周日
1	周一	0-53	本年度中有3天以上
2	周日	1-53	本年度中有一个周日
3	周一	1-53	本年度中有3天以上
4	周日	0-53	本年度中有3天以上
5	周一	0-53	本年度中有一个周一
6	周日	1-53	本年度中有3天以上
7	周一	1-53	本年度中有一个周一

```
mysql> SELECT WEEK('1998-02-20');

       -> 7

mysql> SELECT WEEK('1998-02-20',0);

       -> 7

mysql> SELECT WEEK('1998-02-20',1);

       -> 8

mysql> SELECT WEEK('1998-12-31',1);

       -> 53
```

注意，假如有一个日期位于前一年的最后一周， 若你不使用2、3、6或7作为 *mode* 参数选择，则 MySQL 返回 0：

```
mysql> SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0);

       -> 2000, 0
```

有人或许会提出意见，认为 MySQL 对于 WEEK() 函数应该返回 52，原因是给定的日期实际上发生在1999年的第52周。我们决定返回0作为代替的原因是我们希望该函数能返回"给定年份的星期数"。这使得 WEEK() 函数在同其它从日期中抽取日期部分的函数结合时的使用更加可靠。

假如你更希望所计算的关于年份的结果包括给定日期所在周的第一天，则应使用 0、2、5或 7作为 *mode* 参数选择。

```
mysql> SELECT WEEK('2000-01-01',2);

       -> 52
```

作为选择，可使用 YEARWEEK() 函数：

```
mysql> SELECT YEARWEEK('2000-01-01');

       -> 199952

mysql> SELECT MID(YEARWEEK('2000-01-01'),5,2);

       -> '52'
```

WEEKDAY(date)

返回 *date* (0 =周一, 1 =周二, ... 6 =周日)对应的工作日索引 weekday index for

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');

       -> 1

mysql> SELECT WEEKDAY('1997-11-05');

       -> 2
```

WEEKOFYEAR(date)

将该日期的阳历周以数字形式返回，范围是从1到53。它是一个兼容度函数，相当于 WEEK(*date*,3)。

```
mysql> SELECT WEEKOFYEAR('1998-02-20');

       -> 8
```

YEAR(date)

返回 *date* 对应的年份,范围是从1000到9999。

```
mysql> SELECT YEAR('98-02-03');
       -> 1998
```

YEARWEEK(date), YEARWEEK(date,start)

返回一个日期对应的年或周。*start* 参数的工作同 *start* 参数对 WEEK() 的工作相同。结果中的年份可以和该年的第一周和最后一周对应的日期参数有所不同。

```
mysql> SELECT YEARWEEK('1987-01-01');
       -> 198653
```

注意，周数和 WEEK() 函数队可选参数0或 1可能会返回的(0) w 有所不同，原因是此时 WEEK() 返回给定年份的语境中的周。

```
       -> '1997-10-07'
```

Cast 函数和操作符

BINARY

BINARY 操作符将后面的字符串抛给一个二进制字符串。这是一种简单的方式来促使逐字节而不是逐字符的进行列比较。这使得比较区分大小写，即使该列不被定义为 BINARY 或 BLOB。BINARY 也会产生结尾空白，从而更加显眼。

```
mysql> SELECT 'a' = 'A';
       -> 1
mysql> SELECT BINARY 'a' = 'A';
       -> 0
mysql> SELECT 'a' = 'a ';
       -> 1
mysql> SELECT BINARY 'a' = 'a ';
       -> 0
```

BINARY 影响整个比较；它可以在任何操作数前被给定，而产生相同的结果。

BINARY *str* 是 CAST(*str* AS BINARY) 的缩略形式。

注意，在一些语境中，假如你将一个编入索引的列派给 BINARY, MySQL 将不能有效使用这个索引。

假如你想要将一个 BLOB 值或其它二进制字符串进行区分大小写的比较，你可利用二进制字符串没有字符集这一事实实现这个目的，这样就不会有文书夹的概念。为执行一个区分大小写的比较，可使用 CONVERT() 函数将一个字符串值转化为一个不区分大小写的字符集。其结果为一个非二进制字符串，因此 LIKE 操作也不会区分大小写：

SELECT 'A' LIKE CONVERT(*blob_col* USING latin1) FROM *tbl_name*;

若要使用一个不同的字符集,替换其在上述语句中的 latin1 名。

CONVERT() 一般可用于比较出现在不同字符集中的字符串。

CAST(expr AS type), CONVERT(expr,type) , CONVERT(expr USING transcoding_name)

CAST() 和 CONVERT() 函数可用来获取一个类型的值，并产生另一个类型的值。

这个*类型* 可以是以下值其中的 一个：

- BINARY[(*N*)]
- CHAR[(*N*)]
- DATE
- DATETIME
- DECIMAL
- SIGNED [INTEGER]
- TIME
- UNSIGNED [INTEGER]

BINARY 产生一个二进制字符串。关于它怎样影响比较结果的说明见本章中 BINARY 操作符项。

假如给定了随意长度 *N*，则 BINARY[*N*] 使 cast 使用该参数的不多于 *N* 个字节。同样的， CHAR[*N*] 会使 cast 使用该参数的不多于 *N* 个字符。

CAST() and CONVERT(... USING ...)是标准 SQL 语法。CONVERT() 的非 USING 格式是 ofis ODBC 语法。

带有 USING 的 CONVERT() 被用来在不同的字符集之间转化数据。在 MySQL 中,自动译码名和相应的字符集名称相同。例如。 这个语句将服务器的默认字符集中的字符串 'abc' 转化为 utf8 字符集中相应的字符串：

SELECT CONVERT('abc' USING utf8);

当你想要在一个 CREATE ... SELECT 语句中创建一个特殊类型的列，则 cast 函数会很有用：

CREATE TABLE new_table SELECT CAST('2000-01-01' AS DATE);

该函数也用于 ENUM 列按词法顺序的排序。通常 ENUM 列的排序在使用内部数值时发生。将这些值按照词法顺序派给 CHAR 结果：

SELECT *enum_col* FROM *tbl_name* ORDER BY CAST(*enum_col* AS CHAR);

CAST(*str* AS BINARY) 和 BINARY *str* 相同。 CAST(*expr* AS CHAR) 将表达式视为一个带有默认字符集的字符串。

若用于一个诸如 CONCAT('Date: ',CAST(NOW() AS DATE)) 这样的比较复杂的表达式的一部分，CAST() 也会改变结果。

你不应在不同的格式中使用 CAST() 来析取数据，但可以使用诸如 LEFT() 或 EXTRACT() 的样的字符串函数来代替。

若要在数值语境中将一个字符串派给一个数值,通常情况下，除了将字符串值作为数字使用外，你不需要做任何事：

```
mysql> SELECT 1+'1';
      -> 2
```

若要在一个字符串语境中使用一个数字，该数字会被自动转化为一个 BINARY 字符串。

```
mysql> SELECT CONCAT('hello you ',2);

       -> 'hello you 2'
```

MySQL 支持带符号和无符号的 64 比特值的运算。若你正在使用数字操作符 （如 +）而其中一个操作数为无符号整数，则结果为无符号。可使用 SIGNED 和 UNSIGNED cast 操作符来覆盖它。将运算分别派给带符号或无符号 64 比特整数。

```
mysql> SELECT CAST(1-2 AS UNSIGNED)

       -> 18446744073709551615

mysql> SELECT CAST(CAST(1-2 AS UNSIGNED) AS SIGNED);

       -> -1
```

注意，假如任意一个操作数为一个浮点值，则结果为一个浮点值， 且不会受到上述规则影响 (关于这一点， DECIMAL 列值被视为浮点值)。

```
mysql> SELECT CAST(1 AS UNSIGNED) - 2.0;

       -> -1.0
```

若你在一个算术运算中使用了一个字符串，它会被转化为一个浮点数。

其他函数

FOUND_ROWS()

A SELECT 语句可能包括一个 LIMIT 子句，用来限制服务器返回客户端的行数。在有些情况下，需要不用再次运行该语句而得知在没有 LIMIT 时到底该语句返回了多少行。为了知道这个行数,包括在 SELECT 语句中选择 SQL_CALC_FOUND_ROWS，随后调用 FOUND_ROWS()：

```
mysql> SELECT SQL_CALC_FOUND_ROWS * FROM tbl_name

    -> WHERE id > 100 LIMIT 10;

mysql> SELECT FOUND_ROWS();
```

第二个 SELECT 返回一个数字，指示了在没有 LIMIT 子句的情况下，第一个 SELECT 返回了多少行 (若上述的 SELECT 语句不包括 SQL_CALC_FOUND_ROWS 选项,则使用 LIMIT 和不使用时，FOUND_ROWS() 可能会返回不同的结果)。

通过 FOUND_ROWS() 的有效行数是瞬时的，并且不用于越过 SELECT SQL_CALC_FOUND_ROWS 语句后面的语句。若你需要稍候参阅这个值，那么将其保存：

```
mysql> SELECT SQL_CALC_FOUND_ROWS * FROM ... ;

mysql> SET @rows = FOUND_ROWS();
```

假如你正在使用 SELECT SQL_CALC_FOUND_ROWS， MySQL 必须计算出在全部结果集合中有所少行。然而， 这比不用 LIMIT 而再次运行问询要快，原因是结

果集合不需要被送至客户端。

SQL_CALC_FOUND_ROWS 和 FOUND_ROWS() 在当你希望限制一个问询返回的行数时很有用，同时还能不需要再次运行问询而确定全部结果集合中的行数。一个例子就是提供页式显示的 Web 脚本，该显示包含显示搜索结果其它部分的页的连接。使用 FOUND_ROWS() 使你确定剩下的结果需要多少其它的页。

SQL_CALC_FOUND_ROWS 和 FOUND_ROWS() 的应用对于 UNION 问询比对于简单 SELECT 语句更为复杂，原因是在 UNION 中，LIMIT 可能会出现在多个位置。它可能适用于 UNION 中的个人 SELECT 语句，或是总体上 到 UNION 结果的全程。

SQL_CALC_FOUND_ROWS 对于 UNION 的意向是它应该不需要全程 LIMIT 而返回应返回的行数。SQL_CALC_FOUND_ROWS 和 UNION 一同使用的条件是：

- SQL_CALC_FOUND_ROWS 关键词必须出现在 UNION 的第一个 SELECT 中。
- FOUND_ROWS() 的值只有在使用 UNION ALL 时才是精确的。若使用不带 ALL 的 UNION，则会发生两次删除， 而 FOUND_ROWS() 的指只需近似的。
- 假若 UNION 中没有出现 LIMIT，则 SQL_CALC_FOUND_ROWS 被忽略，返回临时表中的创建的用来处理 UNION 的行数。

LAST_INSERT_ID() LAST_INSERT_ID(expr)

自动返回*最后*一个 INSERT 或 UPDATE 问询为 AUTO_INCREMENT 列设置的*第一个*发生的值。

```
mysql> SELECT LAST_INSERT_ID();

       -> 195
```

产生的 ID *每次连接后*保存在服务器中。这意味着函数向一个给定客户端返回的值是该客户端产生对影响 AUTO_INCREMENT 列的最新语句第一个 AUTO_INCREMENT 值的。这个值不能被其它客户端影响，即使它们产生它们自己的 AUTO_INCREMENT 值。这个行为保证了你能够找回自己的 ID 而不用担心其它客户端的活动，而且不需要加锁或处理。

假如你使用一个非“magic”值来更新某一行的 AUTO_INCREMENT 列，则 LAST_INSERT_ID() 的值不会变化(换言之,一个不是 NULL 也不是 0的值)。

重点:假如你使用单 INSERT 语句插入多个行， LAST_INSERT_ID() *只*返回插入的*第一*行产生的值。其原因是这使依靠其它服务器复制同样的 INSERT 语句变得简单。

若给出作为到 LAST_INSERT_ID() 的参数 *expr* ，则参数的值被函数返回，并作为被 LAST_INSERT_ID() 返回的下一个值而被记忆。这可用于模拟序列：

- 创建一个表，用来控制顺序计数器并使其初始化：

```
mysql> CREATE TABLE sequence (id INT NOT NULL);
mysql> INSERT INTO sequence VALUES (0);
```

- 使用该表产生这样的序列数 ：

```
mysql> UPDATE sequence SET id=LAST_INSERT_ID(id+1);
mysql> SELECT LAST_INSERT_ID();
```

UPDATE 语句会增加顺序计数器并引发向 LAST_INSERT_ID() 的下一次调用，用来返回升级后的值。

你可以不用调用 LAST_INSERT_ID() 而产生序列，但这样使用这个函数的效用在于 ID 值被保存在服务器中，作为自动产生的值。它适用于多个用户，原因是多个用户均可使用 UPDATE 语句并用 SELECT 语句(或 mysql_insert_id()),得到他们自己的序列值，而不会影响其它产生他们自己的序列值的客户端或被其它产生他们自己的序列值的客户端所影响。

注意， mysql_insert_id() 仅会在 INSERT 和 UPDATE 语句后面被升级， 因此你不能在执行了其它诸如 SELECT 或 SET 这样的 SQL 语句后使用 C API 函数来找回 LAST_INSERT_ID(*expr*) 对应的值。

ROW_COUNT()

ROW_COUNT() 返回被前面语句升级的、插入的或删除的行数。 这个行数和 mysql 客户端显示的行数及 mysql_affected_rows() C API 函数返回的值相同。

DEFAULT(col_name)

返回一个表列的默认值。若该列没有默认值则会产生错误。

```
mysql> UPDATE t SET i = DEFAULT(i)+1 WHERE id < 100;
```

UUID()

返回一个通用唯一标识符 (UUID)

Fish 2010