29 你知道 SQL 分析器的实现原理是什么吗？

更新时间：2020-05-13 18:26:29

> 成功 = 艰苦的劳动 + 正确的方法 + 少谈空话。——爱因斯坦

从对“MySQL 系统架构”的介绍中，我们可以知道，分析器是 MySQL 逻辑架构中的一个重要组件。分析器对于我们的查询语句来说，实现了两大功能：词法分析和语法分析。所以，“研究”分析器其实也就是研究这两大功能点。

1. 初识分析器

分析器是逻辑架构中比较“靠前”的组件，用于搞清楚用户的需求（SQL 语句的目的），所以它在整个 MySQL 系统中发挥着至关重要的作用。在讲解它的两大核心功能之前，我们先来简单看下它的工作过程，以及关键字的定义。

1.1 关键字和非关键字

其实，对于关键字的概念，我们已经多次见到过了。它是 MySQL 服务器内置的一些“单词”，也是 SQL 语法的重要组成部分。我们可以在 MySQL 源码（sql/lex.h）中找到所有关键词的定义：

```
static const SYMBOL symbols[] = {
  /*
   Insert new SQL keywords after that commentary (by alphabetical order):
  */
  { SYM("&&",                AND_AND_SYM)},
  { SYM("<",                 LT)},
  { SYM("<=",                LE)},
  { SYM("<>",                NE)},
  { SYM("!=",                NE)},
  ......
```

MySQL 规定用户对库、表、字段等等的命名不能与关键字冲突，而这些自定义的命名就会被称作是非关键字。最后，需要知道，关键字与非关键字是分析器做词法分析的重要依据。

1.2 分析器的工作过程

当 MySQL 中的 “查询缓存” 没有命中时，查询语句便会进入到分析器中（注意这里组件的工作顺序）。分析器将会对我们的 SQL 语句执行两步操作：

- **词法分析**：SQL 语句其实就是按照 “一定的规则” 编写的字符串，这一步的主要工作就是标记出 “字符串” 中的关键字和非关键字
- **语法分析**：这一步中涵盖了两个重要的工作，第一是对 SQL 语句的语法校验；第二是生成语法树，这也是整个分析器执行过程中最复杂的工作了

SQL 语句解析属于编译器的范畴，它和我们平时使用的编程语言解析并没有本质的区别。同时，正是因为它在逻辑架构中所处的位置，也被称作是 MySQL 服务器的前端。

2. 词法分析

对于 Linux 来说，词法分析一般是通过 Flex 与 Bison 完成的。而对于 MySQL 来说，考虑到效率和灵活性，自己实现了词法分析的部分。下面，我们先去搞清楚词法分析的含义（即它的执行过程），再去简单的看一看它的源码实现。

2.1 词法分析执行过程

词法分析对应的英文是 lexical analysis，通常会被简写为 “lex”。词法分析的核心工作就是把输入转化为一个个有意义的词块，称之为 Token。之后，会再把 Token 划分为关键字和非关键字两类。举个例子，假设我们向词法分析组件传递如下的 SQL 语句：

```
SELECT type, name FROM worker;
```

在经过词法分析组件之后，这条 SQL 语句会被 “分割” 为5个 Token，其中有两个关键字，三个非关键字。如下表所示：

关键字	非关键字	非关键字	关键字	非关键字
SELECT	type	name	FROM	worker

分词的本质是正则表达式的匹配过程（构词规则的识别过程），它所表达的语义是：字符串（SQL 语句）中的单词是什么，代表什么含义。好的，正如你所见，相对来说，词法分析的工作量是比较少的。

2.2 词法分析源码解读

词法分析的核心源码位于 sql/sql_lex.cc 中（注意，需要区分代码的版本），词法分析的主要入口函数是 MYSQLlex。下面，贴出这个函数的部分核心代码：

```
int MYSQLlex(YYSTYPE *yylval, YYLTYPE *yylloc, THD *thd)
{
  // lip 中保存了所有读取的词法信息
  Lex_input_stream *lip= & thd->m_parser_state->m_lip;
  int token;
  ......

  // 通过 lex_one_token 函数得到分析结果，即 Token
  token= lex_one_token(yylval, thd);
  yylloc->cpp.start= lip->get_cpp_tok_start();
  yylloc->raw.start= lip->get_tok_start();

  // 对 Token 进行判断分类
  switch(token) {
  case WITH:
    ......
    token= lex_one_token(yylval, thd);
    ......
  return token;
}
```

从 MYSQLlex 的代码可以看出，它是通过 lex_one_token 函数得到了分析结果，并将这个结果赋值给变量 token。lex_one_token 代码同样位于 sql/sql_lex.cc 中，我们来看一看它的执行过程：

```
static int lex_one_token(YYSTYPE *yylval, THD *thd)
{
  uchar c= 0;
  bool comment_closed;
  int tokval, result_state;
  uint length;
  enum my_lex_states state;
  Lex_input_stream *lip= & thd->m_parser_state->m_lip;
  const CHARSET_INFO *cs= thd->charset();
  // 保存了词法分析状态机中的各种状态
  const my_lex_states *state_map= cs->state_maps->main_map;
  const uchar *ident_map= cs->ident_map;

  lip->yylval=yylval;   // The global state

  lip->start_token();
  state=lip->next_state;
  lip->next_state=MY_LEX_OPERATOR_OR_IDENT;

  // 循环体中根据每次 state 变量的取值来决定经过哪一个分支，另外 lip 中也会保存下一个状态的信息
  for (;;)
  {
    switch (state) {
    case MY_LEX_OPERATOR_OR_IDENT: // Next is operator or keyword
    // 初始 state 状态
    case MY_LEX_START:   // Start of token
      // Skip starting whitespace
      while(state_map[c= lip->yyPeek()] == MY_LEX_SKIP)
      ......
  }
}
```

state_map 在初始化时首先将所有的 ASCII 表中的字符所属状态填满，接着会对当前词法分析的状态进行判定来决定返回什么类型的 Token。这个判定的主要工作对应到 for 循环，循环体中会根据每次 state 变量的取值来决定经过哪一个分支，另外 lip 中也会保存下一个状态的信息。

3. 语法分析

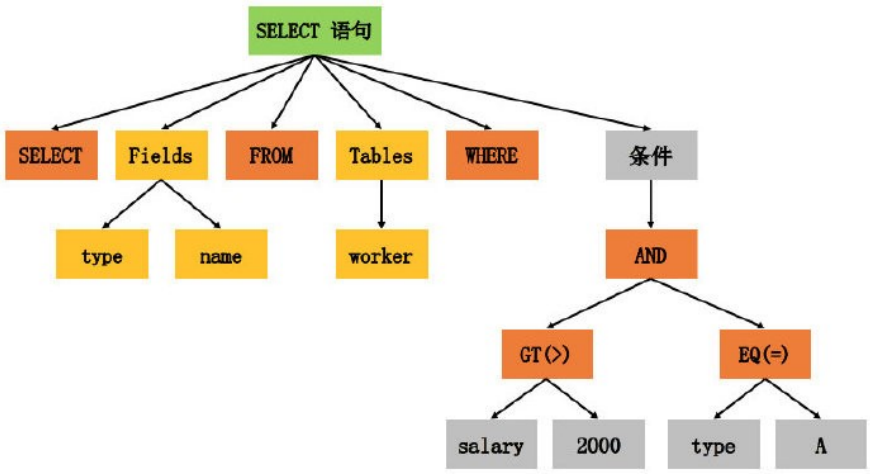
根据词法分析的结果，语法分析器会根据语法规则，判断输入的 SQL 语句是否满足 MySQL 规定的语法，之后生成一棵语法树。所以，语法分析是词法分析的下游。

3.1 生成语法树

语法分析对应的英文是 syntax analysis，简单的说，它就是确定词法分析中的 Token 是如何彼此关联的。MySQL 中的语法分析使用的是 Bison，Bison 会根据 MySQL 定义的语法规则进行语法解析。而语法解析实际上就是生成语法树的过程，这个过程也是最复杂、最精彩的部分。下面，我们先来看一条示例 SQL 语句：

```
SELECT type, name FROM worker WHERE salary > 2000 AND type = 'A';
```

如果把它作为语法分析的输入（实际上是词法分析的输出），可以得到如下图所示的语法树：



可以看出，语法树中的各个“树节点”其实就是 SQL 语句中的各个 Token，而不同的颜色则标记了不同的 Token 类型。顺利生成语法树之后，SQL 分析器的工作也就基本完成了。

3.2 语法树生成源码解读

生成语法树相关的知识点其实属于编译器的范畴，这对于大多数人而言，是比较陌生的。使用 Bison 来构建语法树几乎成了“业界标准”，但是也正是由于它的功能单一，只适用于一些特定的问题。所以，相对来说，在工程领域，它的知名度并不高。

MySQL 语法分析的源码位于 sql/sql_yacc.yy 中，下面，我们先来看一看核心的解析过程：

```
// SELECT 语句的解析入口
select:
        select_init
        {
          $$= NEW_PTN PT_select($1, SQLCOM_SELECT);
        }
      ;

// 首先找到 SELECT 关键字
select_init:
        SELECT_SYM select_part2 opt_union_clause
        {
          $$= NEW_PTN PT_select_init2($1, $2, $3);
        }
      | '(' select_paren ')' union_opt
        {
```

```
        {
          $$= NEW_PTN PT_select_init_parenthesis($2, $4);
        }
      ;
......

select_part2:
        // 解析 SELECT 的列名
        select_options_and_item_list
        // 解析 ORDER BY 子句
        opt_order_clause
        // 解析 LIMIT 子句
        opt_limit_clause
        // 解析 SELECT 语句中的锁
        opt_select_lock_type
        {
          $$= NEW_PTN PT_select_part2($1, NULL, NULL, NULL, NULL, NULL,
                                      $2, $3, NULL, NULL, $4);
        }
      | select_options_and_item_list into opt_select_lock_type
        {
          $$= NEW_PTN PT_select_part2($1, $2, NULL, NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, $3);
        }
      // 解析 “各种” 子句
      | select_options_and_item_list  /* #1 */
        opt_into                      /* #2 */
        from_clause                   /* #3 */
        opt_where_clause              /* #4 */
        opt_group_clause              /* #5 */
        opt_having_clause             /* #6 */
        opt_order_clause              /* #7 */
        opt_limit_clause              /* #8 */
        opt_procedure_analyse_clause  /* #9 */
        opt_into                      /* #10 */
        opt_select_lock_type          /* #11 */
        ......
      ;
......

// FROM 子句，解析表名
from_clause:
        FROM table_reference_list { $$= $2; }
      ;
opt_from_clause:
        /* empty */ { $$= NULL; }
      | from_clause
      ;

// 解析表的联合查询
table_reference_list:
        join_table_list
        {
          $$= NEW_PTN PT_table_reference_list($1);
        }
      | DUAL_SYM { $$= NULL; }
        /* oracle compatibility: oracle always requires FROM clause,
           and DUAL is system table without fields.
           Is "SELECT 1 FROM DUAL" any better than "SELECT 1" ?
        Hmmm :) */
      ;

// 解析 WHERE 子句（条件表达式）
opt_where_clause:
        /* empty */  { $$= NULL; }
      | WHERE expr
        {
          $$= new PTI_context<CTX_WHERE>(@$, $2);
        }
      ;
```

通过 Bison 完成语法解析之后，会将解析结果生成的数据结构保存在 struct LEX 中。LEX 结构体定义在 sql/sql_lex.h 中，源码如下所示：

```
struct LEX: public Query_tables_list
{
  friend bool lex_start(THD *thd);

  SELECT_LEX_UNIT *unit;                ///< Outer-most query expression
  /// @todo: select_lex can be replaced with unit->first-select()
  SELECT_LEX *select_lex;               ///< First query block
  SELECT_LEX *all_selects_list;         ///< List of all query blocks
private:
  /* current SELECT_LEX in parsing */
  SELECT_LEX *m_current_select;
  // 存储 SELECT 关键字之后的列名
  List<Item> item_list;
  // 存储查询的数据表名称
  SQL_I_List<TABLE_LIST>  table_list;
  // 存储查询条件
  Item *where;
}
```

词法分析从 SQL 语句中得到了 Token，并对 Token 进行了分类；语法分析则把这些分类的 Token 按照一定的规则组织成了语法树，并最终保存到 LEX 数据结构中。之后，LEX 会继续向下传递到优化器，优化器再去根据这里的数据，生成执行计划。

4. 总结

SQL 分析器的实现（词法分析和语法分析）是极为复杂的，但幸运的是：大多数时候，你只需要知道这里面都做了些什么 ？能够分析出 SQL 语句中的每一个 Token，以及对它们进行正确的分类。当然，如果能够正确的将这些 Token 组织（画）成一棵语法树，就再好不过了。最后，我不建议你去直接读 MySQL 的实现源码，那确实是一件非常难的事，且并不一定能够指导你的工程应用。

5. 问题

给你一条 SQL 查询语句，你能总结出分析器的工作过程吗 ？

以你工作中遇到的 SQL 查询语句为例，尝试对它进行 Token 的拆分，并组织成语法树 ？

6. 参考资料

《高性能 MySQL（第三版）》

LR parser

Flex (lexical analyser generator)

Understanding SQL Query Parsing

SQL Parser

```
}
```

28 带你认识MySQL的系统架构

30 SQL 查询优化器的实现原理又是什么样的呢？