

第十讲：MySQL 语法分析器的设计与实现

知春路遇上八里桥

<2024-06-06 Thu>



- 1 前情提要
- 2 语法分析
- 3 解析树数据结构
- 4 解析流程
- 5 代码分析

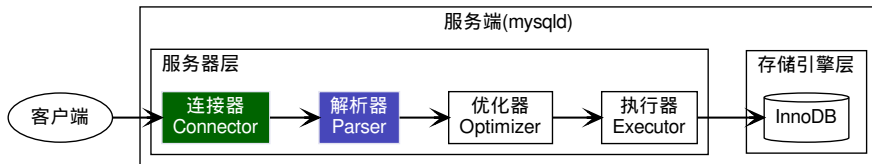


1

前情提要



执行流程



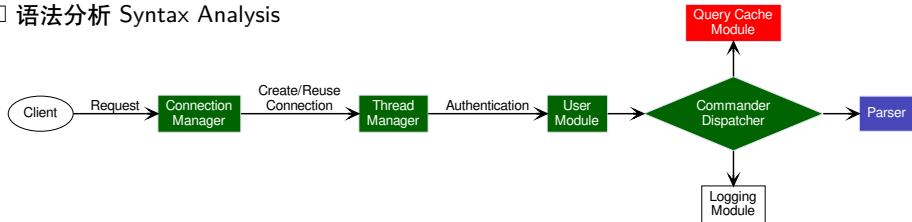
本节内容

- 连接器

- ▶ ☒ 连接管理器 Connection Manager
- ▶ ☒ 线程管理器 Thread Manager
- ▶ ☒ 用户模块 User Module

- 解析器

- ▶ ☒ 网络模块 Net Module
- ▶ ☒ 派发模块 Commander Dispatcher
- ▶ ☒ 词法分析 Lexical Analysis
- ▶ ☐ 语法分析 Syntax Analysis

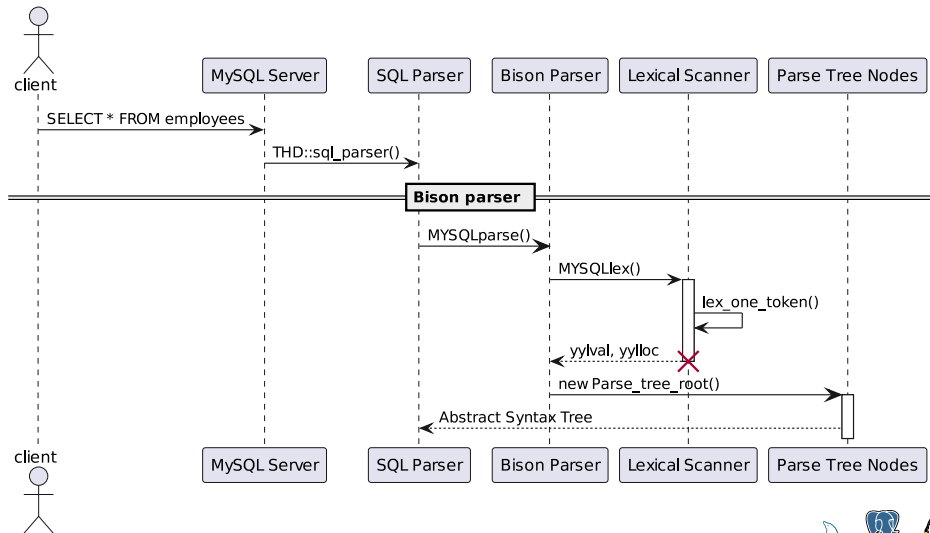


2

语法分析



语法树构建流程



语法分析要点

- ① bison 语法分析器使用 LALR 算法¹，该算法能够自动处理左递归
- ② `dispatch_sql_command()` 开始进入语法解析
- ③ `parse_sql()` 入口函数
- ④ `THD::sql_parser()`
- ⑤ `MYSQLparse()`
 - ▶ 进入编译中间文件 * `build/sql/sql_yacc.cc`
 - ▶ 调用 `yyparse()` 函数，该函数是 bison 生成的
 - ▶ 生成解析树 Parse Tree
- ⑥ `LEX::make_sql_cmd()`
 - ▶ 生成抽象语法树 AST (Abstract Syntax Tree)

¹<https://ustc-compiler-principles.github.io/2023/lab1/Bison/>

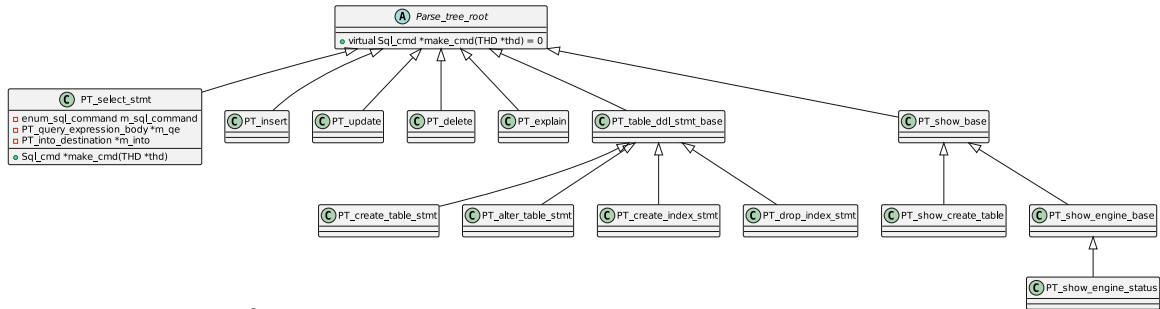


3

解析树数据结构



解析树根节点 Parse_tree_root



- **Parse_tree_root²** 是所有语法树根节点的顶层基类
- 每种类型的 Query 对应的 root 类不同
 - ▶ SELECT 语句 ⇒ **PT_select_stmt** 类
 - ▶ UPDATE 语句 ⇒ **PT_update** 类
 - ▶ DELETE 语句 ⇒ **PT_delete** 类
- 相关类实现见 ★ sql/parse_tree_*

²https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classParse__tree__root.html

SELECT 语句的根节点 PT_select_stmt

① SELECT 语句根节点实现类

► ★ sql/parse_tree_nodes.h

```
1689 class PT_select_stmt : public Parse_tree_root {
1690     typedef Parse_tree_root super;
1691
1692     public:
1693         /**
1694          * @param qe The query expression.
1695          * @param sql_command The type of SQL command.
1696          */
1697         PT_select_stmt(enum_sql_command sql_command, PT_query_expression_body *qe)
1698             : m_sql_command(sql_command),
1699               m_qe(qe),
1700               m_into(nullptr),
1701               m_has_trailing_locking_clauses{false} {}
```

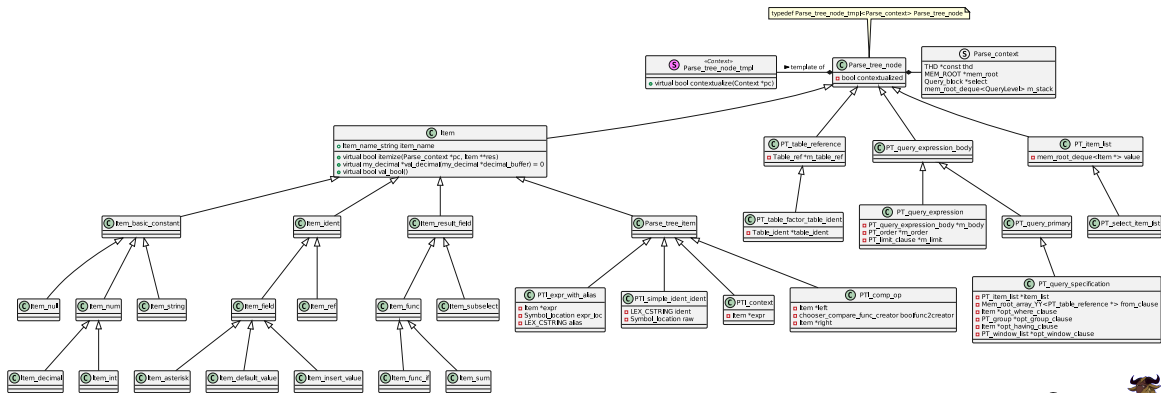
② SELECT 语句的规范见官网³

³<https://dev.mysql.com/doc/refman/8.0/en/select.html>



解析树节点 Parse_tree_node

- `Parse_tree_node`⁴ 是语法树节点的基类
- 最重要的是 `Item` 类，相关类实现见 `sql/item_*`



⁴https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classParse__tree__node__templ.html



Item 类实现

- bison 解析后会生成包含 Item 节点的树，这里成为解析树
 - ▶ Item 有形如 val_xxx() 方法，val_int() / val_str() 是内置的求值函数
 - ▶ 解析树里会有一部分是 PTI_ 开头的 Item，它们都是继承自 Parse_tree_item
 - ① 在 contextualize 阶段时，会对这些 PTI_item 进行 itemize()
 - ② 将它们从解析树节点转化成真正意义的表达式树节点
- Item 是语法树中的基础对象，表示 Query 中的一个具体的项目
 - ▶ 类定义具体见文件 `* sql/item.h` 的定义如下

```
851  class Item : public Parse_tree_node {
852      typedef Parse_tree_node super;
      :
3559      virtual void allow_array_cast() {}
3560  };
```

- itemize() 是调用 contextualize() 是构造最终的 Item 对象

```
virtual bool itemize(Parse_context *pc, Item **res)
```



典型的 Item 举例

- 常量节点/值节点 `Item_base_constant`
 - ▶ 存储常量值
- 字段节点/列节点 `Item_field`
 - ▶ 存储列字段的相关元信息
- 函数计算节点 `Item_func`
 - ▶ 系统函数，例如 `+`, `-`, `*`, `/`, `>=`, `<>` 等系统提供的基本函数型操作
 - ▶ 常用函数，例如数学函数、加密函数等
- 逻辑计算节点 `Item_cond`
 - ▶ 主要是 `and`, `or`, `not` 等条件操作
 - ▶ 这类函数可以看作是输入值为 1 个或 2 个布尔参数，返回值为布尔的特殊函数
- 聚合函数计算 `Item_sum`
 - ▶ 分为系统聚合函数和 UDF。系统聚合函数包括 `sum`、`count`、`avg`、`max`、`min` 等



4

解析流程



BISON 中的 query_expression 规则

- Query 表达式的求值规则

```
9914 query_expression:
9915     query_expression_body
9916     opt_order_clause
9917     opt_limit_clause
9918     {
9919         $$ = NEW_PTN PT_query_expression($1.body, $2, $3);
9920     }
9921 | with_clause
9922   query_expression_body
9923   opt_order_clause
9924   opt_limit_clause
9925   {
9926       $$= NEW_PTN PT_query_expression($1, $2.body, $3, $4);
9927   }
9928 ;
```



创建语法树节点

- bison 规则的原始代码

```
{  
    $$= NEW_PTN PT_query_expression($1, $2.body, $3, $4);  
}
```

- 自动生成的 sql_yacc.cc 代码, 【添加一些换行符】

```
#line 9918 "/opt/src/mysql-server/sql/sql_yacc.yy"  
{  
    (yyval.query_expression) = NEW_PTN PT_query_expression(  
        (yyvsp[-2].query_expression_body_opt_parens).body,  
        (yyvsp[-1].order),  
        (yyvsp[0].limit_clause));  
}
```

- 代码中用到的宏定义

```
#define NEW_PTN new(YMEM_ROOT)  
#define YMEM_ROOT (YYTHD->mem_root)
```



BISON 语法分析流程

- bison 解析的 LALR 算法可以通过日志简单分析流程，通过 `debug=d,parser_debug` 选项开启

Starting parse	<= 开始解析
Entering state 0	<= 进入解析，初始化解析
Stack now 0	
Reading a token	<= 读取一个 token
Next token is token SELECT_SYM (:)	<= 解析到 <code>select</code>
Shifting token SELECT_SYM (:)	<= 移进 <code>select</code> 符号
Entering state 42	<= 进入下一个状态
...	
Reducing stack by rule 1342 (line 9819):	<= 对表达式进行规约
\$1 = nterm query_expression (:)	
-> \$\$ = nterm select_stmt (:)	
...	<= 此处省略一堆操作
Cleanup: popping token "end of file" (:)	<= 读取到文件结束符
Cleanup: popping nterm start_entry (:)	<= 解析结束后清理工作

- 解析算法参考 bison 文档说明⁵

⁵https://www.gnu.org/software/bison/manual/html_node/Algorithm.html



查看 BISON 解析后的语法树

- THD::sql_parser() 代码, 具体见 ★ sql/sql_class.cc

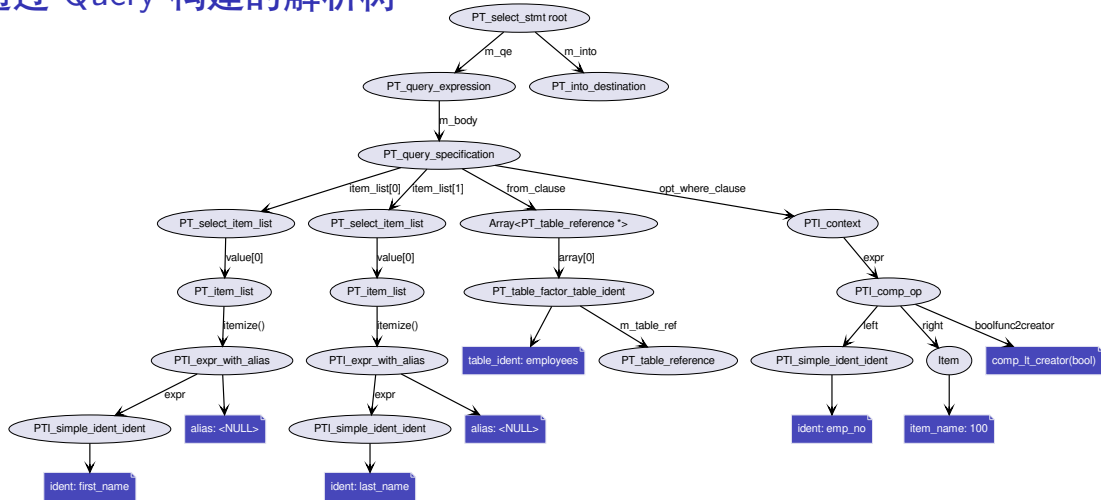
```
3058 Parse_tree_root *root = nullptr;  
3059 if (MYSQLparse(this, &root) || is_error()) {
```

- bison 解析过程在 THD::sql_parser() 中发生, 解析完成生成 root 变量

```
(gdb) b THD::sql_parser()  
(gdb) p *root  
$4 = {_vptr.Parse_tree_root = 0x55555e1fca50 <vtable for PT_delete+16>}  
(gdb) i vtbl root  
vtable for 'Parse_tree_root' @ 0x55555e1fca50 (subobject @ 0x7fff30110600):  
[0]: 0x5555594ac070 <PT_delete::~~PT_delete()>  
[1]: 0x5555594ac0a2 <PT_delete::~~PT_delete()>  
[2]: 0x55555948e196 <PT_delete::make_cmd(THD*)>  
(gdb)
```



通过 Query 构建的解析树



`select first_name, last_name from employees where emp_no < 100;`

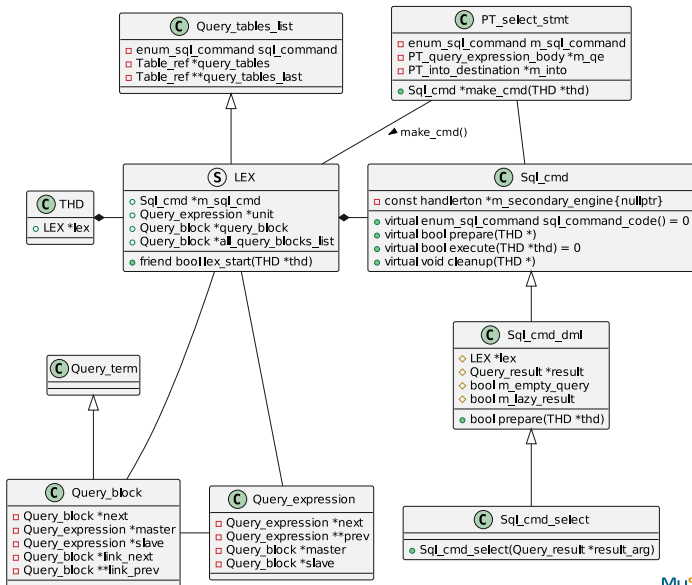


5

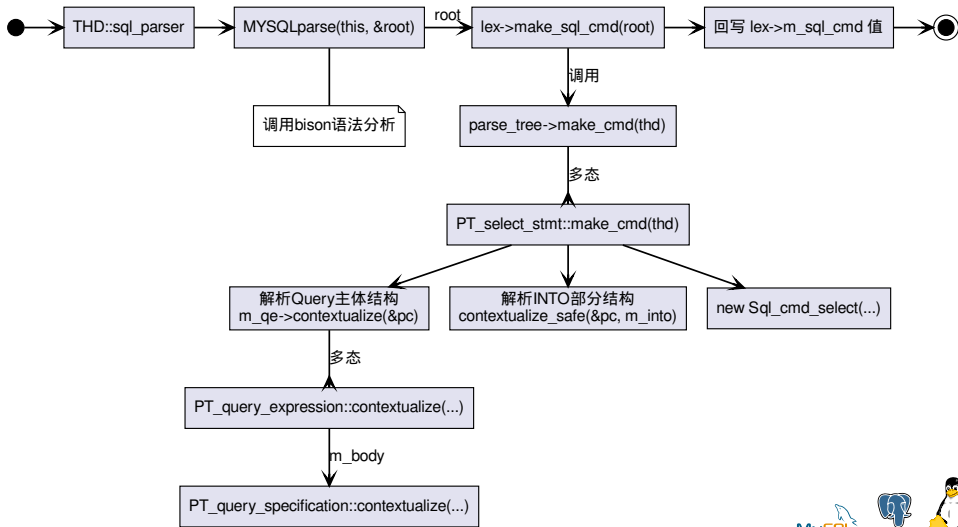
代码分析



LEX 数据结构



THD::sql_parser()



make_cmd

- SELECT 语句根节点创建 AST

- ▶ 代码节选自 `sql/parse_tree_nodes.cc` 的 1689 行

```
Sql_cmd *PT_select_stmt::make_cmd(THD *thd) {  
    if (m_qe->contextualize(&pc)) { // 处理 query_expression  
        return nullptr;  
    }  
    if (contextualize_safe(&pc, m_into)) { // 处理 INTO 子句  
        return nullptr;  
    }  
    // 调用 placement new 构造对象  
    if (thd->lex->sql_command == SQLCOM_SELECT)  
        return new (thd->mem_root) Sql_cmd_select(thd->lex->result);  
    else // (thd->lex->sql_command == SQLCOM_DO)  
        return new (thd->mem_root) Sql_cmd_do(nullptr);  
}
```

- `make_cmd()` 函数可以将代码上下文化, `placement new`⁶ 出 `Sql_cmd` 对象⁷

⁶在已经被分配但尚未处理的 (raw) 内存中构造对象

⁷https://dev.mysql.com/doc/dev/mysql-server/8.0.37/classSql__cmd.html



Query_expression 和 Query_block

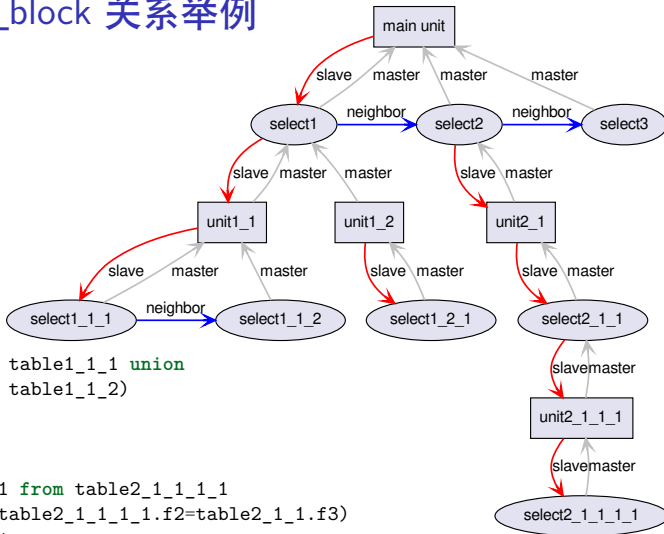
- Query_expression 表示查询表达式
- Query_block 表示查询块
- 查询表达式包含一个或多个查询块
 - ▶ 多个表示我们有 UNION 查询
- 这两个类都有 master、slave、next 和 prev 四个字段。
 - ▶ 对于 Query_block 类
 - ① master 和 slave 指向 Query_expression 类型的对象
 - ▶ 对于 Query_expression 类:
 - ① 它们指向 Query_block
 - ② master 是指向外部节点的指针
 - ③ slave 是指向第一个内部节点的指针
 - ④ neighbors 是同一级别上的两个 Query_block 或 Query_expression 对象
- 参考 ★ sql/sql_lex.h 中 Query_expression 定义前的注释

623 `class Query_expression {`



Query_expression 和 Query_block 关系举例

```
select *  
  from table1  
  where table1.field IN (select * from table1_1_1 union  
                        select * from table1_1_2)  
  
union  
select *  
  from table2  
  where table2.field=(select (select f1 from table2_1_1_1_1  
                            where table2_1_1_1_1.f2=table2_1_1.f3)  
                    from table2_1_1  
                    where table2_1_1.f1=table2.f2)  
  
union  
select * from table3;
```



结束

