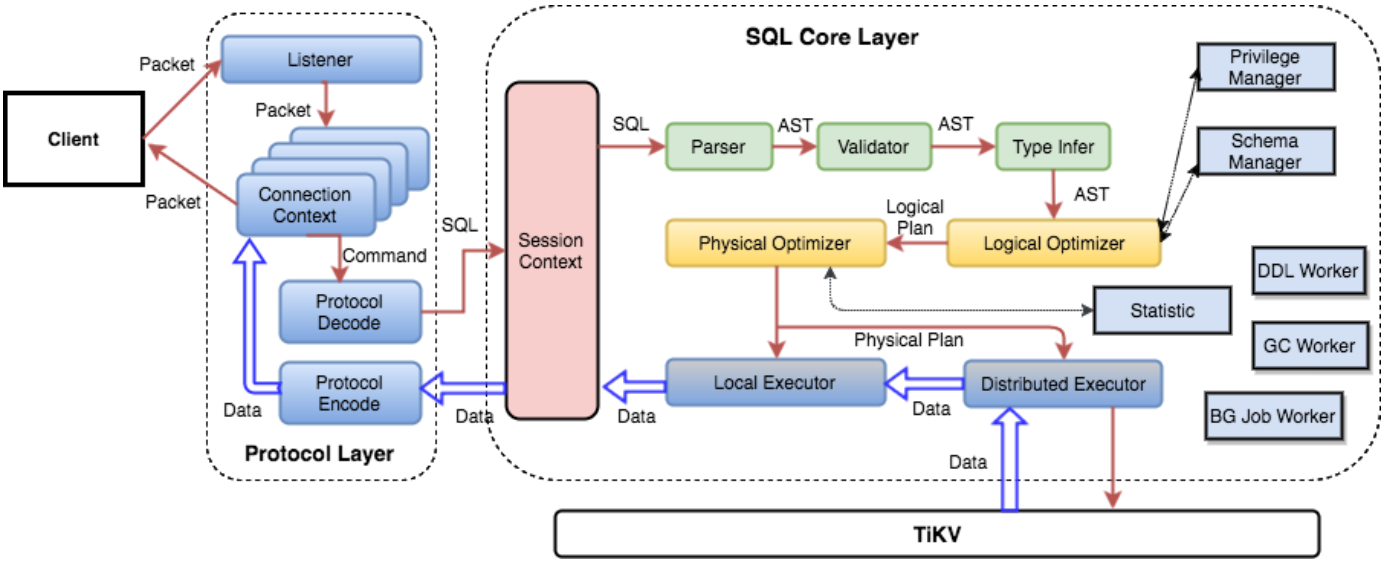
TiDB 源码阅读系列文章（五）TiDB SQL Parser 的实现

马震

2018-03-20

> *本文为 TiDB 源码阅读系列文章的第五篇，主要对 SQL Parser 功能的实现进行了讲解，内容来自社区小伙伴——马震（GitHub ID：mz1999 ）的投稿。*
>
> *TiDB 源码阅读系列文章的撰写初衷，就是希望能与数据库研究者、爱好者进行深入交流，我们欣喜于如此短的时间内就收到了来自社区的反馈。后续，也希望有更多小伙伴加入到与 TiDB 『坦诚相见』的阵列中来。*

PingCAP 发布了 TiDB 的 源码阅读系列文章 ，让我们可以比较系统的去学习了解TiDB的内部实现。最近的一篇 《SQL 的一生》 ，从整体上讲解了一条 SQL 语句的处理流程，从网络上接收数据，MySQL 协议解析和转换，SQL 语法解析，查询计划的制定和优化，查询计划执行，到最后返回结果。



其中， `SQL Parser` 的功能是把 SQL 语句按照 SQL 语法规则进行解析，将文本转换成抽象语法树（ `AST` ），这部分功能需要些背景知识才能比较容易理解，我尝试做下相关知识的介

绍，希望能对读懂这部分代码有点帮助。

TiDB 是使用 goyacc 根据预定义的 SQL 语法规则文件 parser.y 生成 SQL 语法解析器。我们可以在 TiDB 的 Makefile 文件中看到这个过程，先 build `goyacc` 工具，然后使用 `goyacc` 根据 `parser.y` 生成解析器 `parser.go` ：

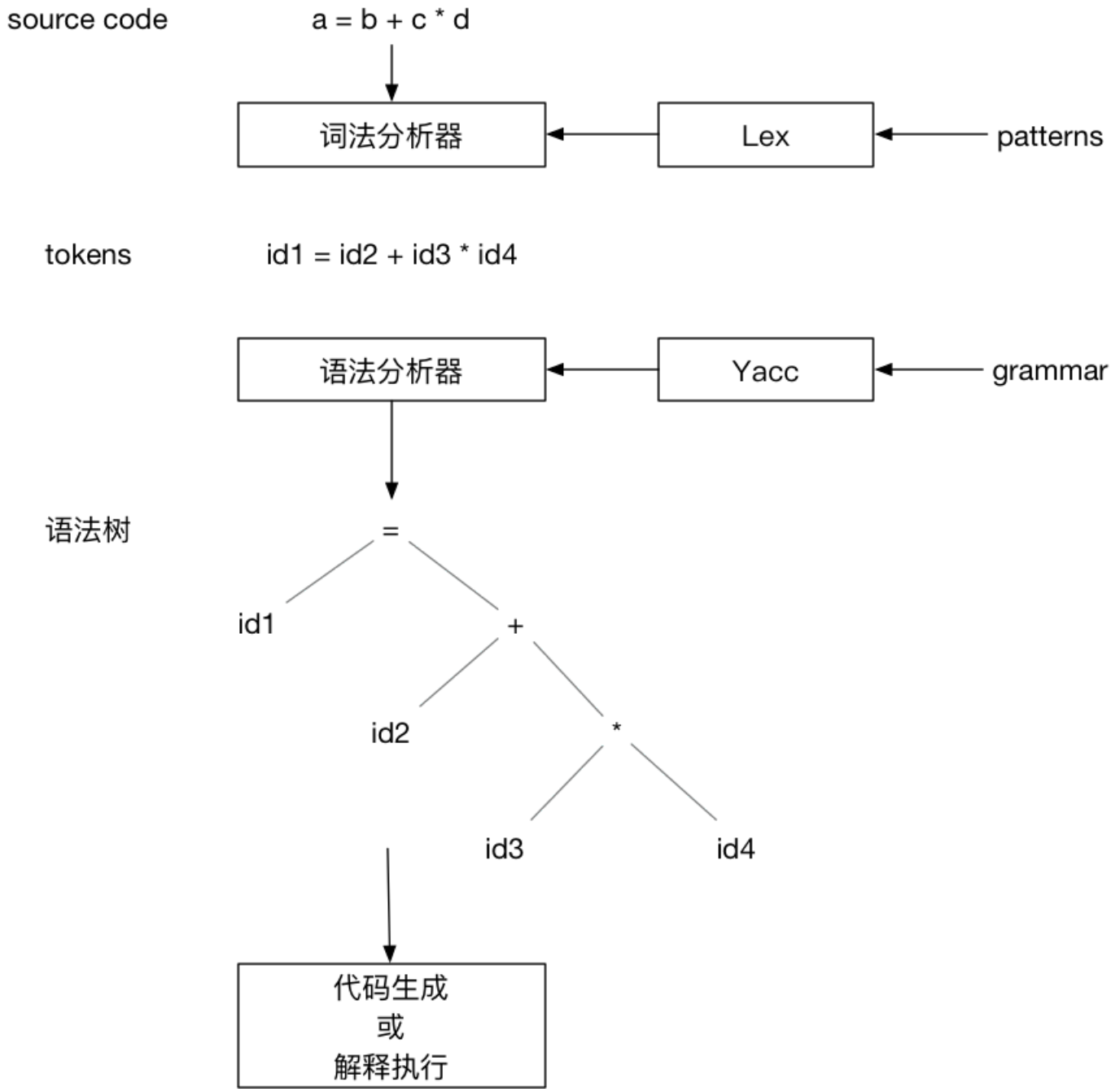
```
goyacc:
    $(GOBUILD) -o bin/goyacc parser/goyacc/main.go

parser: goyacc
    bin/goyacc -o /dev/null parser/parser.y
    bin/goyacc -o parser/parser.go parser/parser.y 2>&1 ...
```

goyacc 是 yacc 的 Golang 版，所以要想看懂语法规则定义文件 parser.y ， 了解解析器是如何工作的，先要对 Lex & Yacc 有些了解。

Lex & Yacc 介绍

Lex & Yacc 是用来生成词法分析器和语法分析器的工具，它们的出现简化了编译器的编写。`Lex & Yacc` 分别是由贝尔实验室的 Mike Lesk 和 Stephen C. Johnson 在 1975 年发布。对于 Java 程序员来说，更熟悉的是 ANTLR ， `ANTLR 4` 提供了 `Listener` + `Visitor` 组合接口， 不需要在语法定义中嵌入 `actions` ，使应用代码和语法定义解耦。 `Spark` 的 SQL 解析就是使用了 `ANTLR` 。 `Lex & Yacc` 相对显得有些古老，实现的不是那么优雅，不过我们也不需要非常深入的学习，只要能看懂语法定义文件，了解生成的解析器是如何工作的就够了。我们可以从一个简单的例子开始：



上图描述了使用 `Lex & Yacc` 构建编译器的流程。 `Lex` 根据用户定义的 `patterns` 生成词法分析器。词法分析器读取源代码，根据 `patterns` 将源代码转换成 `tokens` 输出。 `Yacc` 根据用户定义的语法规则生成语法分析器。语法分析器以词法分析器输出的 `tokens` 作为输入，根据语法规则创建出语法树。最后对语法树遍历生成输出结果，结果可以是产生机器代码，或者是边遍历 `AST` 边解释执行。

从上面的流程可以看出，用户需要分别为 `Lex` 提供 `patterns` 的定义，为 `Yacc` 提供语法规则文件， `Lex & Yacc` 根据用户提供的输入文件，生成符合他们需求的词法分析器和语法分析器。这两种配置都是文本文件，并且结构相同：

```
... definitions ...
%%
```

```
... rules ...
%%
... subroutines ...
```

文件内容由 `%%` 分割成三部分，我们重点关注中间规则定义部分。对于上面的例子， `Lex` 的输入文件如下：

```
...
%%
/* 变量 */
[a-z]    {
            yylval = *yytext - 'a';
            return VARIABLE;
         }
/* 整数 */
[0-9]+   {
            yylval = atoi(yytext);
            return INTEGER;
         }
/* 操作符 */
[-+()=/*\n] { return *yytext; }
/* 跳过空格 */
[ \t]    ;
/* 其他格式报错 */
.        yyerror("invalid character");
%%
...
```

上面只列出了规则定义部分，可以看出该规则使用正则表达式定义了变量、整数和操作符等几种 `token` 。例如整数 `token` 的定义如下：

```
[0-9]+  {
            yylval = atoi(yytext);
            return INTEGER;
        }
```

当输入字符串匹配这个正则表达式，大括号内的动作会被执行：将整数值存储在变量 `yylval` 中，并返回 `token` 类型 `INTEGER` 给 `Yacc` 。

再来看看 `Yacc` 语法规则定义文件：

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
...
%%

program:
        program statement '\n'
        |
        ;

statement:
        expr                    { printf("%d\n", $1); }
        | VARIABLE '=' expr     { sym[$1] = $3; }
        ;

expr:
        INTEGER
        | VARIABLE              { $$ = sym[$1]; }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         { $$ = $1 / $3; }
        | '(' expr ')'          { $$ = $2; }
        ;

%%
...
```

第一部分定义了 `token` 类型和运算符的结合性。四种运算符都是左结合，同一行的运算符优先级相同，不同行的运算符，后定义的行具有更高的优先级。

语法规则使用了 `BNF` 定义。 `BNF` 可以用来表达上下文无关（*context-free*）语言，大部分的现代编程语言都可以使用 `BNF` 表示。上面的规则定义了三个**产生式**。**产生式**冒号左边的项（例如 `statement` ）被称为**非终结符**， `INTEGER` 和 `VARIABLE` 被称为**终结符**,它们是由 `Lex` 返回的 `token` 。**终结符**只能出现在**产生式**的右侧。可以使用**产生式**定义的语法生成表达式：

```
expr -> expr * expr
     -> expr * INTEGER
     -> expr + expr * INTEGER
     -> expr + INTEGER * INTEGER
     -> INTEGER + INTEGER * INTEGER
```

解析表达式是生成表达式的逆向操作，我们需要归约表达式到一个**非终结符**。 `Yacc` 生成的语法分析器使用**自底向上**的归约（*shift-reduce*）方式进行语法解析，同时使用堆栈保存中间状态。还是看例子，表达式 `x + y * z` 的解析过程：

```
1    . x + y * z
2    x . + y * z
3    expr . + y * z
4    expr + . y * z
5    expr + y . * z
6    expr + expr . * z
7    expr + expr * . z
8    expr + expr * z .
9    expr + expr * expr .
10   expr + expr .
11   expr .
12   statement .
13   program  .
```

点（ `.` ）表示当前的读取位置，随着 `.` 从左向右移动，我们将读取的 `token` 压入堆栈，当发现堆栈中的内容匹配了某个**产生式**的右侧，则将匹配的项从堆栈中弹出，将该**产生式**左侧的**非终结符**压入堆栈。这个过程持续进行，直到读取完所有的 `tokens` ，并且只有**启始非终结符**（本例为 `program` ）保留在堆栈中。

产生式右侧的大括号中定义了该规则关联的动作，例如：

```
expr:  expr '*' expr          { $$ = $1 * $3; }
```

我们将堆栈中匹配该**产生式**右侧的项替换为**产生式**左侧的**非终结符**，本例中我们弹出 `expr '*' expr` ，然后把 `expr` 压回堆栈。 我们可以使用 `$position` 的形式访问堆栈中的项，`$1` 引用的是第一项， `$2` 引用的是第二项，以此类推。 `$$` 代表的是归约操作执行后的堆栈顶。本例的动作是将三项从堆栈中弹出，两个表达式相加，结果再压回堆栈顶。

上面例子中语法规则关联的动作，在完成语法解析的同时，也完成了表达式求值。一般我们希望语法解析的结果是一棵抽象语法树（ `AST` ），可以这么定义语法规则关联的动作：

```
...
%%
...
expr:
    INTEGER             { $$ = con($1); }
    | VARIABLE          { $$ = id($1); }
    | expr '+' expr     { $$ = opr('+', 2, $1, $3); }
    | expr '-' expr     { $$ = opr('-', 2, $1, $3); }
    | expr '*' expr     { $$ = opr('*', 2, $1, $3); }
    | expr '/' expr     { $$ = opr('/', 2, $1, $3); }
    | '(' expr ')'      { $$ = $2; }
    ;
%%
nodeType *con(int value) {
    ...
}
nodeType *id(int i) {
    ...
}
nodeType *opr(int oper, int nops, ...) {
    ...
}
```

上面是一个语法规则定义的片段，我们可以看到，每个规则关联的动作不再是求值，而是调用相应的函数，该函数会返回抽象语法树的节点类型 `nodeType` ，然后将这个节点压回堆栈，解析完成时，我们就得到了一颗由 `nodeType` 构成的抽象语法树。对这个语法树进行遍历访问，可以生成机器代码，也可以解释执行。

至此，我们大致了解了 `Lex & Yacc` 的原理。其实还有非常多的细节，例如如何消除语法的歧义，但我们的目的是读懂 TiDB 的代码，掌握这些概念已经够用了。

goyacc 简介

goyacc 是 golang 版的 `Yacc` 。和 `Yacc` 的功能一样， `goyacc` 根据输入的语法规则文件，生成该语法规则的 go 语言版解析器。 `goyacc` 生成的解析器 `yyParse` 要求词法分析器符合下面的接口：

```
type yyLexer interface {
    Lex(lval *yySymType) int
```

```
    Error(e string)
}
```

或者

```
type yyLexerEx interface {
    yyLexer
    // Hook for recording a reduction.
    Reduced(rule, state int, lval *yySymType) (stop bool) // Client should copy
}
```

TiDB 没有使用类似 `Lex` 的工具生成词法分析器，而是纯手工打造，词法分析器对应的代码是 parser/lexer.go ， 它实现了 `goyacc` 要求的接口：

```
...
// Scanner implements the yyLexer interface.
type Scanner struct {
    r   reader
    buf bytes.Buffer

    errs         []error
    stmtStartPos int

    // For scanning such kind of comment: /*! MySQL-specific code */ or /*+ opti
    specialComment specialCommentScanner

    sqlMode mysql.SQLMode
}
// Lex returns a token and store the token value in v.
// Scanner satisfies yyLexer interface.
// 0 and invalid are special token id this function would return:
// return 0 tells parser that scanner meets EOF,
// return invalid tells parser that scanner meets illegal character.
func (s *Scanner) Lex(v *yySymType) int {
    tok, pos, lit := s.scan()
    v.offset = pos.Offset
    v.ident = lit
    ...
}
// Errors returns the errors during a scan.
```

```
func (s *Scanner) Errors() []error {
    return s.errs
}
```

另外 `lexer` 使用了 `字典树` 技术进行 `token` 识别，具体的实现代码在 parser/misc.go

TiDB SQL Parser 的实现

终于到了正题。有了上面的背景知识，对 TiDB 的 `SQL Parser` 模块会相对容易理解一些。TiDB 的词法解析使用的 手写的解析器 （这是出于性能考虑），语法解析采用 `goyacc` 。先看 SQL 语法规则文件 parser.y ， `goyacc` 就是根据这个文件生成SQL语法解析器的。

`parser.y` 有 6500 多行，第一次打开可能会被吓到，其实这个文件仍然符合我们上面介绍过的结构：

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

`parser.y` 第三部分 `subroutines` 是空白没有内容的， 所以我们只需要关注第一部分 `definitions` 和第二部分 `rules` 。

第一部分主要是定义 `token` 的类型、优先级、结合性等。注意 `union` 这个联合体结构体：

```
%union {
    offset int // offset
    item interface{}
    ident string
    expr ast.ExprNode
    statement ast.StmtNode
}
```

该联合体结构体定义了在语法解析过程中被压入堆栈的**项**的属性和类型。

压入堆栈的**项**可能是 `终结符` ，也就是 `token` ，它的类型可以是 `item` 或 `ident` ；

这个**项**也可能是 `非终结符` ，即产生式的左侧，它的类型可以是 `expr` 、 `statement` 、 `item` 或 `ident` 。

`goyacc` 根据这个 `union` 在解析器里生成对应的 `struct` 是：

```
type yySymType struct {
    yys       int
    offset    int // offset
    item      interface{}
    ident     string
    expr      ast.ExprNode
    statement ast.StmtNode
}
```

在语法解析过程中， `非终结符` 会被构造成抽象语法树（ `AST` ）的节点 ast.ExprNode 或 ast.StmtNode 。抽象语法树相关的数据结构都定义在 ast 包中，它们大都实现了 ast.Node 接口：

```
// Node is the basic element of the AST.
// Interfaces embed Node should have 'Node' name suffix.
type Node interface {
    Accept(v Visitor) (node Node, ok bool)
    Text() string
    SetText(text string)
}
```

这个接口有一个 `Accept` 方法，接受 `Visitor` 参数，后续对 `AST` 的处理，主要依赖这个 `Accept` 方法，以 `Visitor` 模式遍历所有的节点以及对 `AST` 做结构转换。

```
// Visitor visits a Node.
type Visitor interface {
    Enter(n Node) (node Node, skipChildren bool)
    Leave(n Node) (node Node, ok bool)
}
```

例如 plan.preprocess 是对 `AST` 做预处理，包括合法性检查以及名字绑定。

union 后面是对 token 和 非终结符 按照类型分别定义：

```
/* 这部分的 token 是 ident 类型 */
%token    <ident>
    ...
    add             "ADD"
    all              "ALL"
    alter             "ALTER"
    analyze             "ANALYZE"
    and             "AND"
    as             "AS"
    asc             "ASC"
    between             "BETWEEN"
    bigIntType         "BIGINT"
    ...

/* 这部分的 token 是 item 类型 */
%token    <item>
    /*yy:token "1.%d"    */    floatLit        "floating-point literal"
    /*yy:token "1.%d"    */    decLit           "decimal literal"
    /*yy:token "%d"      */    intLit           "integer literal"
    /*yy:token "%x"      */    hexLit           "hexadecimal literal"
    /*yy:token "%b"      */    bitLit           "bit literal"

    andnot        "&^"
    assignmentEq    ":="
    eq        "="
    ge        ">="
    ...

/* 非终结符按照类型分别定义 */
%type    <expr>
    Expression             "expression"
    BoolPri                "boolean primary expression"
    ExprOrDefault            "expression or default"
    PredicateExpr            "Predicate expression factor"
    SetExpr                "Set variable statement value's expression"
    ...

%type    <statement>
    AdminStmt            "Check table statement or show ddl statement"
    AlterTableStmt            "Alter table statement"
    AlterUserStmt            "Alter user statement"
```

```
    AnalyzeTableStmt         "Analyze table statement"
    BeginTransactionStmt         "BEGIN TRANSACTION statement"
    BinlogStmt             "Binlog base64 statement"
    ...

%type   <item>
    AlterTableOptionListOpt         "alter table option list opt"
    AlterTableSpec             "Alter table specification"
    AlterTableSpecList         "Alter table specification list"
    AnyOrAll             "Any or All for subquery"
    Assignment             "assignment"
    ...

%type    <ident>
    KeyOrIndex         "{KEY|INDEX}"
    ColumnKeywordOpt     "Column keyword or empty"
    PrimaryOpt         "Optional primary keyword"
    NowSym             "CURRENT_TIMESTAMP/LOCALTIME/LOCALTIMESTAMP"
    NowSymFunc         "CURRENT_TIMESTAMP/LOCALTIME/LOCALTIMESTAMP/NOW"
    ...
```

第一部分的最后是对优先级和结合性的定义：

```
...
%precedence sqlCache sqlNoCache
%precedence lowerThanIntervalKeyword
%precedence interval
%precedence lowerThanStringLitToken
%precedence stringLit
...
%right   assignmentEq
%left     pipes or pipesAsOr
%left     xor
%left     andand and
%left     between
...
```

`parser.y` 文件的第二部分是 `SQL` 语法的产生式和每个规则对应的 `aciton` 。SQL语法非常复杂， `parser.y` 的大部分内容都是产生式的定义。

SQL 语法可以参照 MySQL 参考手册的 SQL Statements 部分，例如 SELECT 语法的定义如下：

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
      [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

我们可以在 parser.y 中找到 SELECT 语句的产生式：

```
SelectStmt:
    "SELECT" SelectStmtOpts SelectStmtFieldList OrderByOptional SelectStmtLimit
    { ... }
|   "SELECT" SelectStmtOpts SelectStmtFieldList FromDual WhereClauseOptional Se
    { ... }
|   "SELECT" SelectStmtOpts SelectStmtFieldList "FROM"
    TableRefsClause WhereClauseOptional SelectStmtGroup HavingClause OrderByOpt
    SelectStmtLimit SelectLockOpt
    { ... }
```

产生式 SelectStmt 和 SELECT 语法是对应的。

我省略了大括号中的 `action` ，这部分代码会构建出 `AST` 的 ast.SelectStmt 节点：

```
type SelectStmt struct {
    dmlNode
    resultSetNode

    // SelectStmtOpts wraps around select hints and switches.
    *SelectStmtOpts
    // Distinct represents whether the select has distinct option.
    Distinct bool
    // From is the from clause of the query.
    From *TableRefsClause
    // Where is the where clause in select statement.
    Where ExprNode
    // Fields is the select expression list.
    Fields *FieldList
    // GroupBy is the group by expression list.
    GroupBy *GroupByClause
    // Having is the having condition.
    Having *HavingClause
    // OrderBy is the ordering expression list.
    OrderBy *OrderByClause
    // Limit is the limit clause.
    Limit *Limit
    // LockTp is the lock type
    LockTp SelectLockType
    // TableHints represents the level Optimizer Hint
    TableHints []*TableOptimizerHint
}
```

可以看出， `ast.SelectStmt` 结构体内包含的内容和 `SELECT` 语法也是一一对应的。

其他的产生式也都是根据对应的 `SQL` 语法来编写的。从 `parser.y` 的注释看到，这个文件最初是用 工具 从 `BNF` 转化生成的，从头手写这个规则文件，工作量会非常大。

完成了语法规则文件 `parser.y` 的定义，就可以使用 `goyacc` 生成语法解析器：

```
bin/goyacc -o parser/parser.go parser/parser.y 2>&1
```

TiDB 对 `lexer` 和 `parser.go` 进行了封装，对外提供 parser.yy_parser 进行 SQL 语句的解析：

```
// Parse parses a query string to raw ast.StmtNode.
func (parser *Parser) Parse(sql, charset, collation string) ([]ast.StmtNode, er
    ...
}
```

最后，我写了一个简单的例子，使用 TiDB 的 `SQL Parser` 进行 SQL 语法解析，构建出 `AST` ，然后利用 `visitor` 遍历 `AST` ：

```
package main

import (
    "fmt"
    "github.com/pingcap/parser"
    "github.com/pingcap/parser/ast"
    _ "github.com/pingcap/tidb/types/parser_driver"
)
type visitor struct{}

func (v *visitor) Enter(in ast.Node) (out ast.Node, skipChildren bool) {
    fmt.Printf("%T\n", in)
    return in, false
}

func (v *visitor) Leave(in ast.Node) (out ast.Node, ok bool) {
    return in, true
}

func main() {
    p := parser.New()

    sql := "SELECT /*+ TIDB_SMJ(employees) */ emp_no, first_name, last_name " +
        "FROM employees USE INDEX (last_name) " +
        "where last_name='Aamodt' and gender='F' and birth_date > '1960-01-01'"
    stmtNodes, _, err := p.Parse(sql, "", "")

    if err != nil {
        fmt.Printf("parse error:\n%v\n%s", err, sql)
        return
    }
```

```
        for _, stmtNode := range stmtNodes {
            v := visitor{}
            stmtNode.Accept(&v)
        }
    }
```

我实现的 `visitor` 什么也没干，只是输出了节点的类型。 这段代码的运行结果如下，依次输出遍历过程中遇到的节点类型：

```
*ast.SelectStmt
*ast.TableOptimizerHint
*ast.TableRefsClause
*ast.Join
*ast.TableSource
*ast.TableName
*ast.BinaryOperationExpr
*ast.BinaryOperationExpr
*ast.BinaryOperationExpr
*ast.ColumnNameExpr
*ast.ColumnName
*ast.ValueExpr
*ast.BinaryOperationExpr
*ast.ColumnNameExpr
*ast.ColumnName
*ast.ValueExpr
*ast.BinaryOperationExpr
*ast.ColumnNameExpr
*ast.ColumnName
*ast.ValueExpr
*ast.FieldList
*ast.SelectField
*ast.ColumnNameExpr
*ast.ColumnName
*ast.SelectField
*ast.ColumnNameExpr
*ast.ColumnName
*ast.SelectField
*ast.ColumnNameExpr
*ast.ColumnName
```

了解了 TiDB `SQL Parser` 的实现，我们就有可能实现 TiDB 当前不支持的语法，例如添加内置函数，也为我们学习查询计划以及优化打下了基础。希望这篇文章对你能有所帮助。

> *作者介绍：马震，金蝶天燕架构师，曾负责中间件、大数据平台的研发，今年转向了 NewSQL 领域，关注 OLTP/AP 融合，目前在推动金蝶下一代 ERP 引入 TiDB 作为数据库*

> *存储服务。*

> *点击查看更多 TiDB 源码阅读系列文章*