数据库内核的南天门

——SQL编译器开发

从Parser开始数据库内核开发

内核开发的困难（一）

- 系统性讲PostgreSQL内核开发的文档或者书籍很少
  - 官方的Wiki《**Development information**》

    https://wiki.postgresql.org/wiki/Development_information
  - 《PostgreSQL数据库内核分析》、《PostgreSQL指南内幕探索》等书籍。
- 打开日志debug开关管中窥豹
  - 查看SQL执行的内部过程，在gdb或者日志中查看语法树，代码的执行路径。但是，很难从全局理解一个数据库程序的核心功能是什么。
  - PostgreSQL已经有91万行c代码，30万行注释，没有人能去读取规模这么庞大的源码

内核开发的困难（二）

- 管中窥豹，很容易误入歧途
  - 陷入代码庞大结构以及复杂的细节中，而且这里面涉及进程、内存、网络、存储等方方面面的API
- 不能理清需求
  - 我要做什么，我做的有没有用，会不会被社区接纳

从软件工程的角度审视PostgreSQL

- 需求
  - ？？？
- 设计
  - ？？？
- 编码
  - ？？？
- 测试
  - ？？？

需求

- PostgreSQL 是一个关系型数据库（**抽象概念**）
  - 数据模型是关系模型，用表的集合来表示数据与数据之间的联系，每个表有多个列，每个列有唯一的列名。……

关系模型是数据库软件的最核心也是最基础的概念，E-R模型是对现实中对象-关系的一种抽象，而关系模型则用二维表的形式表示了实体以及其联系。所以一个关系型数据库，就是实体以及联系在软件空间的实现。

- SQL(Structured Query Language)是一种为关系型数据库管理系统(RDBMS)设计用于管理数据的编程语言

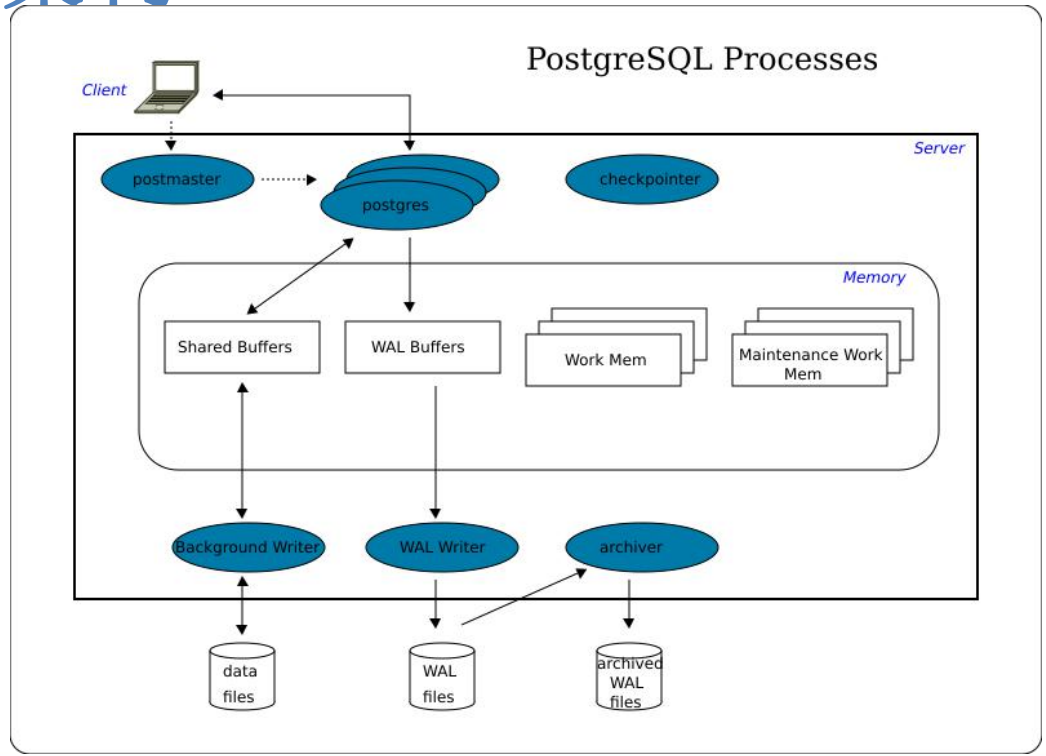
SQL是数据库的功能接口，SQL的能力直接决定了数据库实现的功能

设计

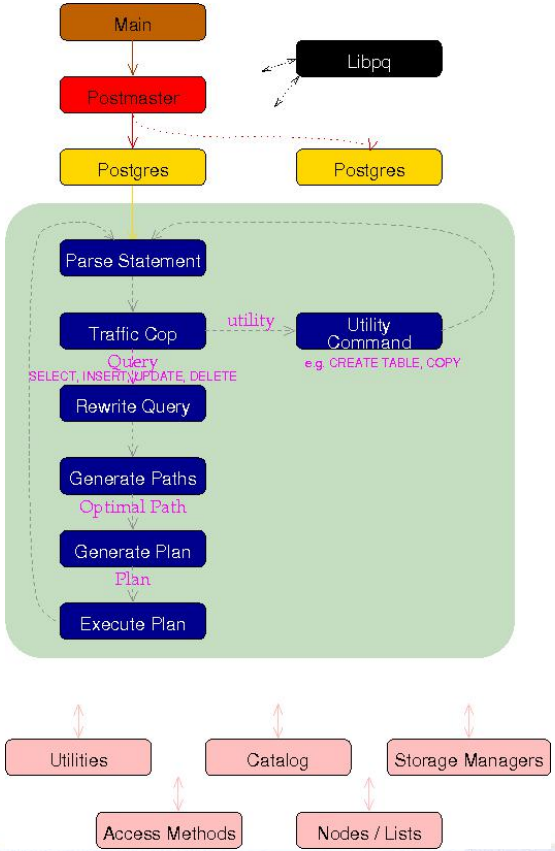
- 数据库的实现文档
  - 《数据库系统概论》
  - 《数据库系统实现》
  - 《Architecture of a Database System》
  - 其他各种出版物以及论文

最核心的需求是二维表在数据库软件中的表示以及二维表的操作接口

实现——程序架构


PostgreSQL Processes
Client
Server
postmaster
postgres
checkpointer
Memory
Shared Buffers
WAL Buffers
Work Mem
Maintenance Work Mem
Background Writer
WAL Writer
archiver
data files
WAL files
archived WAL files

实现——执行逻辑


Main
Libpq
Postmaster
Postgres
Postgres
Parse Statement
Traffic Cop
utility
Utility Command
Query
SELECT, INSERT, UPDATE, DELETE
e.g. CREATE TABLE, COPY
Rewrite Query
Generate Paths
Optimal Path
Generate Plan
Plan
Execute Plan
Utilities
Catalog
Storage Managers
Access Methods
Nodes / Lists

从Parser开始数据库内核开发

- 不管是重新开发还是基于现有数据库修改，一开始很容易就进入了编码阶段，没有理清楚软件要表达的核心的概念，也没有搞清楚软件的基础架构。
- 因为SQL语言是数据库的功能接口，是对关系模型的语言表达，所以对于数据库开发来说，最核心的是定义支持的SQL，解释SQL，执行SQL。
- SQL定义
  - SQL-92，SQL-99,SQL-2016等
  - 自定义SQL

从Parser开始数据库内核开发（续）

- SQL解释
  - SQL编译器，词法分析，语法分析，语义分析
- SQL执行
  - 生成执行计划，执行执行计划

SQL 编译器基础

编译器的基础知识

- 编译原理（龙书、虎书）
  - 只需要理解词法分析，语法分析，语义分析了解一下
- Flex&Bison（O'REILLY）
  - Flex 词法分析器

    官方文档：https://westes.github.io/flex/manual/
  - GNU Bison

    官方文档：https://www.gnu.org/software/bison/manual/bison.html

编译的执行过程

- 查询经过TCP/IP或者Unix socket以数据包的形式传递给backend进程，然后作为一个字符串传递给parser，词法分析器(scan.l)将字符串打散成一个个的token（单词），语法分析器依次读取token并适配对应的语法规则(gram.y)然后生成抽象语法树。

——《**Backend Flowchart** 》https://www.postgresql.org/developer/backend/

- 编写一个完全用Yacc语法分析器的语义动作短语来实现的编译器是有可能的，但是这种编译器很难阅读和维护。…… 为了便于模块化，最好将语法问题(语法分析)和语义问题(类型检查和翻译成机器代码)分开处理。达到此目的的一种方法是由语法分析器生成语法分析树. 技术上，每一个输入单词对应着对应着语法树中的一个叶子节点，每一个语法规则对应着书中的一个内部节点。

——《现代编译原理》4.2抽象语法分析树

语法以及语法树

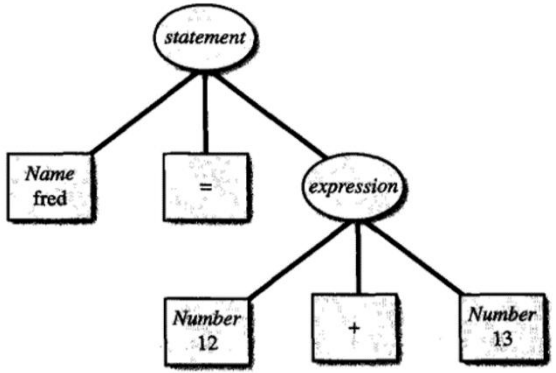
语法

statement : NAME ‘=’ expression

Expression : NUMBER ‘+’ NUMBER

| NUMBER ‘-’ NUMBER

输入:fred=12+13的抽象语法树



标准SQL——一切之始

- 不要去“发明”SQL，标准SQL已经非常成熟，不仅仅是内核开发，任何想了解数据库功能的人都应该阅读SQL标准，或者至少阅读数据库的SQL手册
- 数据库的SQL手册或者SQL标准都用BNF文法表示。BNF文法能严格的定义语法规则，而且它语法简单，表示明确，便于语法分析和编译。
- 对于关系型数据库内核开发来说，标准SQL应该就是产品需求文档，其他的功能是标准SQL功能的扩展
- BNF文法的标准SQL已经非常接近实际的bison中的语法设计，只需要很小的调整就能写在自己的语法分析器中。通过分析SQL语法中的终结符，整理词法分析的需求

PostgreSQL的编译器路径

头文件

```
include/parser
├── analyze.h
├── gramparse.h
├── kwlist.h
├── parse_agg.h
├── parse_clause.h
├── parse_coerce.h
├── parse_collate.h
├── parse_cte.h
├── parse_enr.h
├── parse_expr.h
├── parse_func.h
├── parse_node.h
├── parse_oper.h
├── parse_param.h
├── parse_relation.h
├── parser.h
├── parse_target.h
├── parsetree.h
├── parse_type.h
├── parse_utilcmd.h
├── scanner.h
└── scansup.h

0 directories, 22 files
```

代码文件

```
backend/parser
├── analyze.c
├── check_keywords.pl
├── gram.c
├── gram.h
├── gram.y
├── Makefile
├── parse_agg.c
├── parse_clause.c
├── parse_coerce.c
├── parse_collate.c
├── parse_cte.c
├── parse_enr.c
├── parse_expr.c
├── parse_func.c
├── parse_node.c
├── parse_oper.c
├── parse_param.c
├── parser.c
├── parse_relation.c
├── parse_target.c
├── parse_type.c
├── parse_utilcmd.c
├── README
├── scan.c
├── scan.l
└── scansup.c

0 directories, 26 files
```

语法分析——构建语法树

核心需求

- 建库语法

  CREATE/DROP DATABASE

- 建表语法

  CREATE/DROP TABLE

- 增删改查语法

  INSERT/DELETE/UPDATE/SELECT

- 索引(可选，标准SQL不包含)

  CREATE/DROP INDEX

pg的建表语句

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST } ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

pg中的创建表的语法之一

```
CreateStmt: CREATE OptTemp TABLE IF_P NOT EXISTS qualified_name '(‘
            OptTableElementList ')' OptInherit OptPartitionSpec OptWith
            OnCommitOption OptTableSpace{
                    CreateStmt *n = makeNode(CreateStmt);
                    $7->relpersistence = $2;
                    n->relation = $7;
                    n->tableElts = $9;
                    n->inhRelations = $11;
                    n->partspec = $12;
                    n->ofTypename = NULL;
                    n->constraints = NIL;
                    n->options = $13;
                    n->oncommit = $14;
                    n->tablespacename = $15;
                    n->if_not_exists = true;
                    $$ = (Node *)n;
            }
```

CREATE TABLE

```
CREATE [ {GLOBAL TEMPORARY | LOCAL TEMPORARY} ] TABLE <Table name>
  <table contents source>
  [ ON COMMIT {PRESERVE ROWS | DELETE ROWS} ]

  <table contents source> ::=
  (<table element list>) |
  OF <UDT name> [ UNDER <supertable name> [ {,<supertable name>}... ] ] [ <table element list> ]

     <table element list> ::=
     <table element> [ {,<table element>}... ]

        <table element> ::=
        <Column definition> |
        <Table Constraint> |
        LIKE <Table name> |
        <Column name> WITH OPTIONS <column option list>

            <Table Constraint> ::=
            [ CONSTRAINT <Constraint name> ]
            Constraint_type
            [ <constraint attributes> ]

            <column option list> ::=
           [ <scope clause> ]
           [ <default clause> ]
           [ <Column Constraint>... ]
           [ COLLATE <Collation name> ]
```

使用bison创建语法

1. 根据BNF文法在bison中创建语法，这时候可以不关心抽象语法树的实现，先解决语法的问题
2. 定义终结符以及其semantic value类型，终结符是存储某种具体数据的类型。
3. 为非终结符构造抽象语法树的数据结构。非终结符通常作为抽象语法树的一个Node。

一般一个非终结符表示一种node，每种node第一个属性就是nodetype(nodetype被设计为枚举类型)，遍历语法树时先识别nodetype再强制转换为对应node类型。在语法树的实现中list(表示多个node) 也是一种node类型。就这样，每次非终结符规约以后作为node加入抽象语法树。

设计语法——定义终结符和非终结符

- 创建关键字、标识符、变量等终结符的token name，定义它们的semantic value类型

```
%define api.value.type union /* Generate YYSTYPE from these types: */
%token <double>  NUM     /* Double precision number. */
%token <symrec*> VAR FUN /* Symbol table pointer: variable/function. */
```

- 创建非终结符，定义类型

```
%type <double> exp input line
```

设计语法——创建语法规则

```
input:
  %empty
| input line;
line:
  '\n'
| exp '\n'   { printf ("%.10g\n", $1); }
| error '\n' { yyerrok;                };
exp:
  NUM
| VAR           { $$ = $1->value.var;          }
| VAR '=' exp     { $$ = $3; $1->value.var = $3;    }
| FUN '(' exp ')'   { $$ = $1->value.fun ($3);      }
| exp '+' exp     { $$ = $1 + $3;              }
| exp '-' exp     { $$ = $1 - $3;              };
```

语法设计——OptTemp

建表语句开始部分：

CREATE [[GLOBAL | LOCAL] { TEMPORARY | TEMP } | UNLOGGED] TABLE [IF NOT EXISTS] table_name

转换成语法：

```
temp_opt : global_opt temp_word
         | %empty
global_opt : GLOBAL
         | LOCAL
         | %empty ;
temp_keyword : TEMPORARY
         | TEMP;
```

语法设计——pg的OptTemp实现

```
OptTemp:
            TEMPORARY                   { $$ = RELPERSISTENCE_TEMP; }
            | TEMP                      { $$ = RELPERSISTENCE_TEMP; }
            | LOCAL TEMPORARY           { $$ = RELPERSISTENCE_TEMP; }
            | LOCAL TEMP                { $$ = RELPERSISTENCE_TEMP; }
            | GLOBAL TEMPORARY          { $$ = RELPERSISTENCE_TEMP; }
            | GLOBAL TEMP               { $$ = RELPERSISTENCE_TEMP; }
            | UNLOGGED                   { $$ = RELPERSISTENCE_UNLOGGED; }
            | /*EMPTY*/                  { $$ = RELPERSISTENCE_PERMANENT; }
;
```

设计语法——解决语法冲突

- Shift(移进)、reduce（规约）

Bison在读到token时，它会将token以及它的semantic value一起压入一个栈(parser stack)，将token入栈的动作被称之为shift(移进)。当最上面的n个token和grouping（非终结符）符合某条语法规则时，会将这n个token和grouping中栈中弹出合并成那条语法规则的grouping，然后将新的grouping入栈，这个动作称之为reduce(归约)

- Lookahead token

Bison实际上并不会马上执行reduce，而是发现可以reduce时，会向前读一个token作为一个”lookahead token”,暂时也不放在栈中。这时候会将栈中的token和grouping尽量的执行更多的reduce直到不能reduce，才会将”lookahead token”压入栈中。

”lookahead token”,存放于yychar，它的semantic value以及location存放于yylval和yylloc

设计语法——解决语法冲突(二)

- 语法冲突的理解
  - 对于相同的输入，在BNF文法中有多条路径
- Shift/reduce冲突
  - 悬空else问题

```
if_stmt:
    "if" expr "then" stmt
    | “if” expr “then” stmt “else” stmt;
```

当else作为”lookahead token”时，这时候栈中的token既可以使用规则1规约，同时也可以用规则2移进，两者都合法，冲突发生时bison会选择默认移进。

设计语法——解决语法冲突(三)

- reduce/reduce冲突
  - 对于相同的输入，有多个规则可以进行reduce的时候

```
sequence:
      %empty        { printf ("empty sequence\n"); }
    | maybeword
    | sequence word  { printf ("added word %s\n", $2); };
maybeword:
      %empty    { printf ("empty maybeword\n"); }
    | word      { printf ("single word %s\n", $1); };
```

读到一个word这时候有多条路径到达sequence，

设计语法——解决语法冲突(四)

- Shift/reduce冲突有时候无法避免，但是reduce/reduce冲突基本上都是语法设计的问题。
- 对于shift/reduce冲突，在确认语法设计确实没有问题的时候，用优先级以及结合性或者虚拟token，不建议在重新设计语法上花费太多时间，因为即使重新设计的语法可能引入其他冲突。
- Reduce/reduce冲突一定要解决
- 警惕语法中所有右边可能为空的规则，能合并的尽量合并在一起
- 解决完冲突问题，剩下构建语法树的问题就是填充语法树的编程问题

调用语法分析器

- 入口函数yyparse()

yyparse的作用:调用yyparse()的时候会执行语法解析，读入token，执行规则的action，在遇到结束符号或者不可恢复的语法错误以后返回结果。

yyparse的默认函数定义：int yyparse (void)

在规则的action最后返回宏YYACCEPT或者YYABORT能理解结束语法分析

可以通过声明%parse-param来为yyparse定义参数例如：

%parse-param {int *nastiness} {int *randomness}

可以在调用yyparse时，传入两个参数

int nastiness, randomness;

value = yyparse (&nastiness, &randomness);

可以通过这种方式将抽象语法树的指针传入语法分析器，语法分析就是把语法树填充的过程

调用语法分析器

- yylex()是词法分析器的启动函数，每次调用yylex()词法分析器都会匹配一段文本并且返回token type。下一次继续调用返回下一个token type。
- Bison并不实现这个函数，但是yyparse会调用这个函数。所以要么自己实现，要么使用词法分析器的实现。如果yylex不在bison的语法文件中实现，必须在编译bison的时候用”-d”参数生成token type的宏的头文件，然后yylex函数引用这些宏。
- yylex的返回值正常情况下为正值，为负或者零表示输入已经结束。
- 通常yylex()没有参数，主要通过全局变量yylval和yylloc与程序其它部分交互。但是，我们让词法分析器可重入以后，这两个参数就不可用。

yylex()与Reentrant（可重入）

- 常规的flex和bison不支持重入，一次只能分析一个输入流，因为它们之间使用静态结构记录分析过程，并且它们与调用程序之间的通信也依赖静态结构。”纯”或者”可重入”代码能够将静态结构替换为参数传递给词法分析器和语法分析器。这样，可以它们在多线程中使用，或者同时打开多个词法分析器，或者递归的词法分析器。
- 如果bison声明了%define api.pure full以后成为纯的可重入的词法分析器，全局变量yylval、yylloc不可用，这时候这两个变量成了由yylex传递过去的两个指针参数。这样就在词法分析器中将值赋予这两个指针，然后在调用完yylex以后从两个指针中取回值。
- 传递给yyllex的参数由%lex-param声明

  %lex-param {scanner_mode *mode}

- 如果声明了%define api.pure full’和 %locations，yylex会两连个默认参数：YYSTYPE *lvalp, YYLTYPE *llocp。

语法分析器总结

- 确定要解释的BNF文法
- 编写语法规则
- 定义终结符token和非终结符grouping
- 修改语法或者指定优先级以及结合性来解决语法冲突
- 实现规则的action代码，生成语法树
- 定义yyparse的参数，将语法树通过指针传递出去
- 定义yylex，确定词法分析器

词法分析——识别token

词法分析器-flex

- 词法分析器scanner
  - A scanner is a program which recognizes lexical patterns in text.
  - 词法分析器就是在文本中识别出指定词法模式（正则表达式）的单词
- Scan.l
  - 词法分析的定义文件，定义了lexical patterns的正则表达式以及识别到以后要执行的代码
  - Flex会根据scan.l生成scan.c的代码文件，也可以生成scan.h的头文件，文件名也可指定

词法分析例子

```
%{
  /* need this for the call to atof() below */
  #include <math.h>
  %}

  DIGIT    [0-9]
  ID       [a-z][a-z0-9]*
  %%
  {DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                  atoi( yytext ) );
             }
  {DIGIT}+"."{DIGIT}*       {
            printf( "A float: %s (%g)\n", yytext,
                  atof( yytext ) );
            }
```

```
 if|then|begin|end|procedure|function       {
            printf( "A keyword: %s\n", yytext );
            }
 {ID}       printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^{}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%
```

Token

- 词法分析器被作为语法分析器输入的时候，作用就是使用正则表达式识别出语法分析器中的单个终结符，被称之为token
- Token包括两部分，token type以及semantic value。
- Token type用大写的类似c语言的标识符来表示，它表示一种类型的终结符，比如INTEGER, IDENTIFIER, IF ,RETURN或‘,’。词法分析器返回给语法分析器的也只是token type，token type是是枚举类型中的一个的整数值。
- Semantic value表示token的具体值。Semantic value也可以定义类型。

词法分析——关键字列表

- 在任何语言中总有定义关键字，词法分析的一个作用就是识别出关键字，一般每个关键字是一个token type，而其他都是标识符；当然，也可以为每个关键字设置词法模式(正则表达式)，分别识别
- 关键字是一个预定义的列表，词法分析器在识别到一个单词以后，需要去列表中查询是否是关键字，如果是，返回关键字的token type，否则一般返回一个标识符
- 还有一种自定义的必须唯一的符号，比如表名，索引名，这些一般在语义分析阶段通过符号表来解决，不过对于数据库来说，会直接查询数据库的元数据

词法分析——pg的关键字查找

```
ident_start             [A-Za-z\200-\377_]
ident_cont              [A-Za-z\200-\377_0-9\$]
identifier              {ident_start}{ident_cont}*
{identifier}            {
                        const ScanKeyword *keyword;
                        char              *ident;
                        SET_YYLLOC();
                        /* Is it a keyword? */
                        keyword = ScanKeywordLookup(yytext,yyextra->keywords,
                                yyextra->num_keywords);
                        if (keyword != NULL){
                                yylval->keyword = keyword->name;
                                return keyword->value;}

                        // No.  Convert the identifier to lower case, and truncate if necessary.
                        ident = downcase_truncate_identifier(yytext, yyleng, true);
                        yylval->str = ident;
                        return IDENT;
                        }
```

词法分析——起始状态(Start Conditions)

- Flex提供了一种机制可以在某种情况下被激活的匹配规则。以‘<sc>’开头的语法模式只有在词法分析器正处于‘<sc>’的状态下才会有效。词法分析器默认工作在INITIAL状态。
- 起始状态的切换能够让词法分析器“开小差去干别的活”，比如对于c语言代码，可以切入头文件，获取一些预定义的函数名或者类型，处理完了再切回代码文件。对于SQL来说，最常见的就是识别单引号引起来的字符串常量以及双引号引起来的大小写不敏感标识符
- 单引号或者双引号引起来的识别一般要经过多个词法模式的匹配才能确定识别成功或者失败。在这个过程中，匹配到的单词需要用全局变量存起来，如果失败则报错，成功的话有可能读入了更多了的字符，这时候要用调用yyless()将多余的字符退回给词法分析器，而且这时候要结束当前状态，切回之前的状态
- 起始状态可以让不同的词法分析器集成在一起。Pg通过改变flex默认的函数名和变量定义了自己的词法分析器

词法分析——pg的起始状态

* <xb> bit string literal
* <xc> extended C-style comments
* <xd> delimited identifiers (double-quoted identifiers)
* <xh> hexadecimal numeric string
* <xq> standard quoted strings
* <xe> extended quoted strings (support backslash escape sequences)
* <xdolq> foo quoted strings
* <xui> quoted identifier with Unicode escapes
* <xuiend> end of a quoted identifier with Unicode escapes, UESCAPE can follow
* <xus> quoted string with Unicode escapes
* <xusend> end of a quoted string with Unicode escapes, UESCAPE can follow
* <xeu> Unicode surrogate pair in extended quoted string

词法分析——pg的字符串识别

```
quote                   '
quotestop               {quote}{whitespace}*
quotecontinue           {quote}{whitespace_with_newline}{quote}
quotefail               {quote}{whitespace}*"-“
xqstart                 {quote}
xqdouble                {quote}{quote}
xqinside                [^’]+
{xqstart}               {
        yyextra->warn_on_first_escape = true;
        yyextra->saw_non_ascii = false;
        SET_YYLLOC();
        if (yyextra->standard_conforming_strings)
                BEGIN(xq);
        else
                BEGIN(xe);
        startlit();
}
```

词法分析——pg的字符串识别(续)

```
<xq,xus>{xqinside} {
        addlit(yytext, yyleng, yyscanner);
        }
<xq,xe>{quotestop} |<xq,xe>{quotefail} {
        yyless(1);
        BEGIN(INITIAL);
        /*
         * check that the data remains valid if it might have been
         * made invalid by unescaping any chars.
         */
        if (yyextra->saw_non_ascii)
                pg_verifymbstr(yyextra->literalbuf,yyextra->literallen,
                               false);
        yylval->str = litbufdup(yyscanner);
        return SCONST;
}
```

词法分析——pg的字符串识别(续)

```
static void addlit(char *ytext, int yleng, core_yyscan_t yyscanner)
{
        /* enlarge buffer if needed */
        if ((yyextra->literallen + yleng) >= yyextra->literalalloc)
        {
                do
                {
                        yyextra->literalalloc *= 2;
                } while ((yyextra->literallen + yleng) >= yyextra-
>literalalloc);
                yyextra->literalbuf = (char *) repalloc(yyextra-
>literalbuf,
                                                        yyextra->literalalloc);
        }
        /* append new data */
        memcpy(yyextra->literalbuf + yyextra->literallen, ytext, yleng);
        yyextra->literallen += yleng;
}
```

词法分析——pg的字符串识别(续)

```
static char *litbufdup(core_yyscan_t yyscanner)
{
        int     llen = yyextra->literallen;
        char    *new = palloc(llen + 1);
        memcpy(new, yyextra->literalbuf, llen);
        new[llen] = '\0';
        return new;
}
```

词法分析——全局变量yylval

- 这是语法分析器bison定义的一个变量，它的类型是YYSTYPE，也是由bison定义
- Token的semantic value被存储在yylval中。这意味着flex将token的具体值存储在里面，然后bison在识别到token type以后，会从yylval中获取token的值
- Yylval的类型可以有多种定义方式，但是一般会在语法分析文件开头定义为union，pg的gram.y也是这么定义的。

Yylval的类型定义参考bison文档：
https://www.gnu.org/software/bison/manual/bison.html#Multiple-Types

词法分析——可重入词法分析器

- 设置选项%option reentrant以及在词法分析器中传入用户自定义数据结构

```
%option reentrant
%option extra-type="struct stat *"
%%
__filesize__ printf( "%ld", yyextra->st_size);
__lastmod__ printf( "%ld", yyextra->st_mtime);
%%
```

```
void scan_file( char* filename )
{
        yyscan_t scanner;
        struct stat st;
        FILE *in;
        in = fopen( filename, "r" );
        stat( filename, &st );
        yylex_init_extra( st, &scanner);
        yyset_in( in, scanner );
        yylex( scanner );
        yylex_destroy( scanner );
        fclose( in );
}
```

pg的YY_EXTRA_TYPE:core_yy_extra_type

Data Fields

char *	scanbuf
Size	scanbuflen
const ScanKeywordList *	keywordlist
const uint16 *	keyword_tokens
int	backslash_quote
bool	escape_string_warning
bool	standard_conforming_strings
char *	literalbuf
int	literallen
int	literalalloc
int	xcdepth
char *	dolqstart
int32	utf16_first_part
bool	warn_on_first_escape
bool	saw_non_ascii

词法分析——总结

- 了解词法分析的职责，抓住核心功能
- 词法模式
- Token的token type和semantic value
- 关键字列表
- Start condition
- Reentrant

总结

从Parser开始数据库内核开发（总结）

- 数据库软件业务问题的抽象——关系模型
- 从编译器纵览数据库执行过程

  字符串->token->语法规则->抽象语法树->语义分析->优化重写->执行计划

- 不止SQL parser
  - 阅读其它语法标准，了解语言整体设计，不止于调用API或者阅读源码
  - 接触底层硬件语言c，汇编，更加了解语言的实现细节
  - 将行业知识抽象成数据或者语言标准，创建自己的数据或者程序语言

THANKS