链滴

Go 词法语法分析

作者：someone27889

原文链接：https://ld246.com/article/1598449763573

来源网站：链滴

词法分析

下列一段代码

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello")
}
```

通过lex规则最终解释成

```
PACKAGE  IDENT

IMPORT  QUOTE IDENT QUOTE

IDENT  IDENT LPAREN RPAREN  LBRACE
IDENT DOT IDENT LPAREN QUOTE IDENT QUOTE RPAREN
RBRACE
```

这样将源码翻译成 好多的token(字符) 这一过程可以理解成词法分析

go的词法分析最终生成的Token列表 src/cmd/compile/internal/syntax/tokens.go

```
// Copyright 2016 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package syntax

type token uint

//go:generate stringer -type token -linecomment

const (
    _    token = iota
    _EOF       // EOF

    // names and literals
    _Name    // name
    _Literal // literal

    // operators and operations
    // _Operator is excluding '*' (_Star)
    _Operator // op
    _AssignOp // op=
    _IncOp    // opop
    _Assign   // =
    _Define   // :=
```

```
    _Arrow    // <-
    _Star     // *

    // delimiters
    _Lparen    // (
    _Lbrack    // [
    _Lbrace    // {
    _Rparen    // )
    _Rbrack    // ]
    _Rbrace    // }
    _Comma     // ,
    _Semi      // ;
    _Colon     // :
    _Dot       // .
    _DotDotDot // ...

    // keywords
    _Break       // break
    _Case        // case
    _Chan        // chan
    _Const       // const
    _Continue    // continue
    _Default     // default
    _Defer       // defer
    _Else        // else
    _Fallthrough // fallthrough
    _For         // for
    _Func        // func
    _Go          // go
    _Goto        // goto
    _If          // if
    _Import      // import
    _Interface   // interface
    _Map         // map
    _Package     // package
    _Range       // range
    _Return      // return
    _Select      // select
    _Struct      // struct
    _Switch      // switch
    _Type        // type
    _Var         // var

    // empty line comment to exclude it from .String
    tokenCount //
)

const (
    // for BranchStmt
    Break       = _Break
    Continue    = _Continue
    Fallthrough = _Fallthrough
    Goto        = _Goto
```

```
    // for CallStmt
    Go    = _Go
    Defer = _Defer
)

// Make sure we have at most 64 tokens so we can use them in a set.
const _ uint64 = 1 << (tokenCount - 1)

// contains reports whether tok is in tokset.
func contains(tokset uint64, tok token) bool {
    return tokset&(1<<tok) != 0
}

type LitKind uint8

// TODO(gri) With the 'i' (imaginary) suffix now permitted on integer
//           and floating-point numbers, having a single ImagLit does
//           not represent the literal kind well anymore. Remove it?
const (
    IntLit LitKind = iota
    FloatLit
    ImagLit
    RuneLit
    StringLit
)

type Operator uint

//go:generate stringer -type Operator -linecomment

const (
    _ Operator = iota

    // Def is the : in :=
    Def  // :
    Not  // !
    Recv // <-

    // precOrOr
    OrOr // ||

    // precAndAnd
    AndAnd // &&

    // precCmp
    Eql // ==
    Neq // !=
    Lss // <
    Leq // <=
    Gtr // >
    Geq // >=

    // precAdd
    Add // +
```

```
    Sub // -
    Or  // |
    Xor // ^

    // precMul
    Mul    // *
    Div    // /
    Rem    // %
    And    // &
    AndNot // &^
    Shl    // <<
    Shr    // >>
)

// Operator precedences
const (
    _ = iota
    precOrOr
    precAndAnd
    precCmp
    precAdd
    precMul
)
```

以上便是词法分析需要保留的最终结果

语法分析比较重要的一个结构体 scanner 位于 src/cmd/compile/internal/syntax/scanner.go

```
type scanner struct {
  // 源
    source
  // 模式
    mode   uint
  // 当遇到EOF/换行时 转换成 ;
    nlsemi bool // if set '\n' and EOF translate to ';'

  // 行列
    line, col uint
  // 空行
    blank     bool // line is blank up to col
    tok       token
    lit       string   // valid if tok is _Name, _Literal, or _Semi ("semicolon", "newline", or "EOF");
ay be malformed if bad is true
    bad       bool     // valid if tok is _Literal, true if a syntax error occurred, lit may be malforme

    kind      LitKind  // valid if tok is _Literal
    op        Operator // valid if tok is _Operator, _AssignOp, or _IncOp
    prec      int      // valid if tok is _Operator, _AssignOp, or _IncOp
}
```

scanner 的next函数扫描下一个字符

```
func (s *scanner) next() {
    nlsemi := s.nlsemi
    s.nlsemi = false
```

```
redo:
    // skip white space
    s.stop()
    startLine, startCol := s.pos()
    for s.ch == ' ' || s.ch == '\t' || s.ch == '\n' && !nlsemi || s.ch == '\r' {
        s.nextch()
    }

    // token start
    s.line, s.col = s.pos()
    s.blank = s.line > startLine || startCol == colbase
    s.start()
    if isLetter(s.ch) || s.ch >= utf8.RuneSelf && s.atIdentChar(true) {
        s.nextch()
        s.ident()
        return
    }

    switch s.ch {
    case -1:
        if nlsemi {
            s.lit = "EOF"
            s.tok = _Semi
            break
        }
        s.tok = _EOF

    case '\n':
        s.nextch()
        s.lit = "newline"
        s.tok = _Semi

    case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
        s.number(false)

    case '"':
        s.stdString()

    case '`':
        s.rawString()

    case '\'':
        s.rune()

    case '(':
        s.nextch()
        s.tok = _Lparen

    case '[':
        s.nextch()
        s.tok = _Lbrack

...
```

根据扫描到的 不同的 字符 翻译成不同的 token 并且存储到scanner的 缓冲区内(tok)中,实际类型是 0 234.... iota正数类型

词法解析的过程都是惰性的，只有在上层的解析器需要时才会调用 next 获取最新的 Token

语法分析

- 词法分析的 输出 是 语法分析的 输入 但是 词法分析和语法分析是同时进行的

主要结构体,里面存入了文件,scanner语法分析器,和一些错误行数信息

```
type parser struct {
    file  *PosBase
    errh  ErrorHandler
    mode  Mode
    pragh PragmaHandler
    // parser 中存入了 语法分析的scanner
    scanner

    base   *PosBase // current position base
    first  error    // first error encountered
    errcnt int      // number of errors encountered
    pragma Pragma   // pragmas

    fnest  int    // function nesting level (for error handling)
    xnest  int    // expression nesting level (for complit ambiguity resolution)
    indent []byte // tracing support
}
```

通过fileOrNil来同时进行词法语法转译

```
func (p *parser) fileOrNil() *File {
    if trace {
        defer p.trace("file")()
    }

    f := new(File)
    f.pos = p.pos()

    // PackageClause
    // 检测 是否存有 _Package token对应的为 语法需要分析的 package
    if !p.got(_Package) {
        p.syntaxError("package statement must be first")
        return nil
    }
    // 存入参数和 package name
    f.Pragma = p.takePragma()
    f.PkgName = p.name()
    // 然后用 ; 分隔开Package
    p.want(_Semi)

    // don't bother continuing if package clause has errors
    if p.first != nil {
```

```
        return nil
    }
    // 词法分析的 import 语法分析成 ImportDecl
    // { ImportDecl ";" }
    for p.got(_Import) {
        f.DeclList = p.appendGroup(f.DeclList, p.importDecl)
        p.want(_Semi)
    }
    // 顶层代码: var const type 分别转成 varDecl constDecl typeDecl
    // 顶层代码都是通过 appendGroup来进行分析
    // { TopLevelDecl ";" }
    for p.tok != _EOF {
        switch p.tok {
        case _Const:
            // 语法分析词法分析同时进行的标志，由于结构体中存有scanner，这里调用的是scanner.ne
t来进行下一个字符的词法分析,分析出的_Import等token继续流入这里进行语法分析
            p.next()
            f.DeclList = p.appendGroup(f.DeclList, p.constDecl)

        case _Type:
            p.next()
            f.DeclList = p.appendGroup(f.DeclList, p.typeDecl)

        case _Var:
            p.next()
            f.DeclList = p.appendGroup(f.DeclList, p.varDecl)

        case _Func:
            p.next()
            // 函数略特殊
            // 函数会转成 Funcdel 存有属性类型参数函数体的 语法分析结构
            // 函数的主体其实就是一个 Stmt 数组，Stmt 是一个接口，实现该接口的类型其实也非常多
总共有 14 种不同类型的 Stmt 实现： 例如:IfStmt ForStmt SwitchStmt等
            if d := p.funcDeclOrNil(); d != nil {
                f.DeclList = append(f.DeclList, d)
            }

        default:
            if p.tok == _Lbrace && len(f.DeclList) > 0 && isEmptyFuncDecl(f.DeclList[len(f.DeclList
-1]) {
                // opening { of function declaration on next line
                p.syntaxError("unexpected semicolon or newline before {")
            } else {
                p.syntaxError("non-declaration statement outside function body")
            }
            p.advance(_Const, _Type, _Var, _Func)
            continue
        }

        // Reset p.pragma BEFORE advancing to the next token (consuming ';')
        // since comments before may set pragmas for the next function decl.
        p.clearPragma()

        if p.tok != _EOF && !p.got(_Semi) {
```

```
            p.syntaxError("after top level declaration")
            p.advance(_Const, _Type, _Var, _Func)
        }
    }
    // p.tok == _EOF

    p.clearPragma()
    f.Lines = p.line

    return f
}
```

最后解析为

```
ConstDecl = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec = IdentifierList [ [ Type ] "=" ExpressionList ] .

TypeDecl  = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec  = AliasDecl | TypeDef .
AliasDecl = identifier "=" Type .
TypeDef   = identifier Type .

VarDecl = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList ) .

FunctionDecl = "func" FunctionName Signature [ FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .

MethodDecl = "func" Receiver MethodName Signature [ FunctionBody ] .
Receiver   = Parameters .

Block = "{" StatementList "}" .
StatementList = { Statement ";" } .

Statement =
    Declaration | LabeledStmt | SimpleStmt |
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
    DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarD
cl .
```

对应的类型形成语法树

```
"json.go": SourceFile {
    PackageName: "json",
    ImportDecl: []Import{
        "io",
    },
    TopLevelDecl: ...
```

```
}
```

生成语法树后就是 使用语法树 生成中间代码

参考资料

draveness