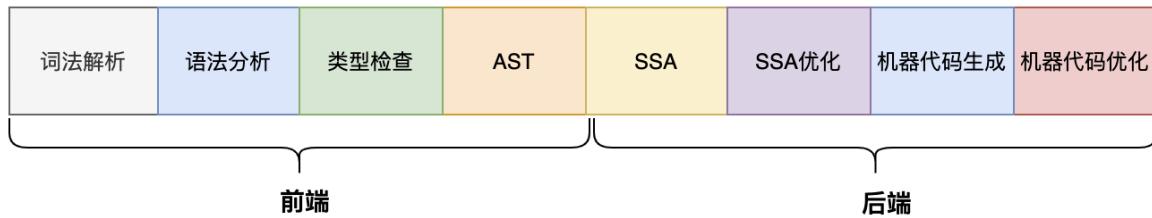


# GO编译流程

## 背景

GO属于静态语言，需要编译才能运行，因为计算机只能识别二进制机器码。GO编译过程分为前端和后端，编译前端主要有词法分析、语法分析、类型检查及语义分析，编译后端主要有中间码生成、代码优化及机器码生成。



## 预备知识

想了解编译过程，我们先了解一下编译过程的一些术语及专业知识。

## CPU架构

GO目前支持386、amd64、arm及arm64等CPU架构，这里简单介绍一下相关架构：

名称	说明
amd64	是一种64位元的电脑处理器架构,别名x86_64
arm	国ARM公司是全球领先的半导体知识产权(IP)提供商。全世界超过95%的智能手机和平板电脑都采用ARM架构
arm64	64位RISC微处理器
i386	是 80386 的那代 CPU 的标准, 主要是支持 32 位的保护模式和实模式两种工作环境
mips	是一种采取精简指令集(RISC)的处理器架构, 是32位大端字节序, 所属公司宣布将放弃继续设计MIPS架构, 全身心投入RISC-V阵营
mipsle	32位小端字节序
mips64	64位大端字节序
mips64le	64位小端字节序

ppc64	是Linux和GCC开源软件社区内常用的，指向目标架构为64位PowerPC和Power Architecture处理
ppc64le	是一个已经推出了纯小端模式，POWER8作为首要目标，OpenPower基金会基础的技术，试图使基于x86的Linux软件的移植工作以最小的工作量进行
riscv64	是一个基于精简指令集(RISC)原则的开源指令集构架(ISA)，简易解释为开源软件运动相对应的一种「开源硬件」
s390x	IBM System z 系列 (zSeries)大型机 (mainframe) 硬件平台，是银行或者大型企业或者科研单位用的
wasm	是一个可移植、体积小、加载快并且兼容 Web 的全新格式

wasm: 编译GOOS=js GOARCH=wasm go build -o main.wasm

## 指令集

不同的系统及硬件指令集有可能不一样，这也是为什么go会针对不同的架构调用相应汇编代码，目前比较常见的指令集arm、x86等。

指令集分为精简指令和复杂指令集：

复杂指令集：通过增加指令的类型减少需要执行指令数。

精简指令集：使用更少的指令类型完成目标的计算任务。

Intel和AMD可以认为是CISC的典型代表。一条复杂的指令实现复杂的功能，对于编译器的要求低，只需一条指令就可以解决问题。ARM必须将数据加载到寄存器才能被指令使用，例如 $a+=b$ ，复杂指令只需要一条指令，简单指令需要先加载再计算。

区别：

指令系统类型	指令
CISC(复杂)	数量多，使用频率差别大，可变长格式
RISC(精简)	数量少，使用频率接近，定长格式，大部分为单周期指令，load/store操作内存

**CISC**对CPU逻辑电路的设计要求高，简化了对编译器的要求，但是带来了CPU成本和功耗的增加。

**RISC**通过多条简单指令拼凑一个复杂功能，对编译器优化要求高，但是功耗低，CPU设计简单，主要用在端侧。

**CISC**指令长度不固定，指令较多，这将导致指令切割复杂，通常需要切割成多个微操作码，然后执行计算。

**RISC**指令长度固定，指令较少，指令码切割简单，这将更容易并行化，执行效率高。

## AST

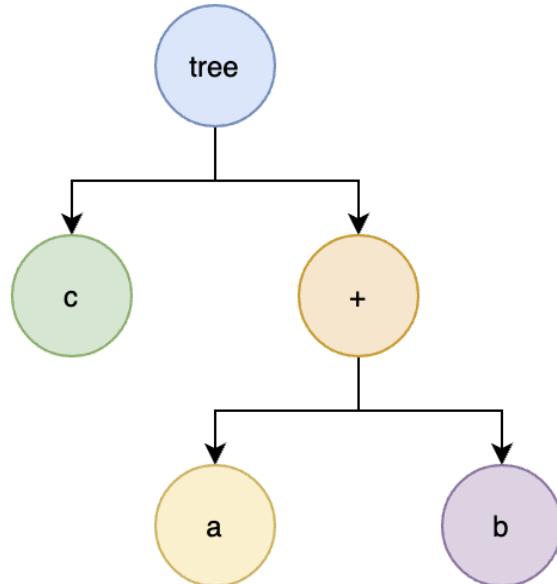
抽象语法树(Abstract Syntax Tree), 是源代码结构的一种抽象表达方式, AST会删除源代码中不重要的一些空格、分号或者括号等相关字符。

例如:c := a + b,这里binaryExpr是指二元表达式

```
- Lhs: [
  - Ident {
    Name: "c"
  }
]

- Rhs: [
  - BinaryExpr {
    Op: "+"
    - X: Ident {
      Name: "a"
    }
    - Y: Ident {
      Name: "b"
    }
  }
]

Tok: ":"
```



## SSA

静态单赋值(Static Single Assignment)是中间代码的特性, 如果中间代码具有静态单赋值特性, 那么每个变量只能会被赋值一次, SSA主要是对代码进行优化的。

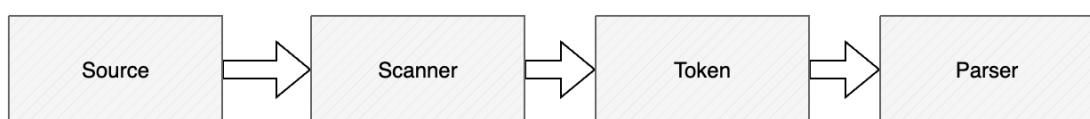
下面代码`x := 1`不会起到任何作用:

```
x := 1
x := 2
y := x
```

## 词法和语法分析

词法分析就是将一个由字符组成的、无法被理解的字符串进行分组, 降低理解字符串的成功。

词法语义执行流程:



首先打开源代码文件，根据b、r、e游标读取字符，通过next()函数扫描获取对应的token。将对应的token进行语义分析，最终得到AST抽象语法树。

## 词法解析

语法分析主要是识别单词对应的token序列。

### Token定义

Go定义的token主要由名称和字面量、操作符、分隔符和关键字组成，下面是token种类：

标识符	操作符	分界符	关键字
-----	-----	-----	-----

**token** 定义如下：

```
const (
    token = iota
    _EOF           // EOF

    _Name          // name 标识符
    _Literal        // literal 字面值

    // 操作符
    _Operator       // op
    _AssignOp      // op=
    ...
    _Star          // *

    // delimiters 分界符
    _Lparen         // (
    _Lbrack         // [
    ...
    _Dot            // .
    _DotDotDot     // ...

    // keywords 关键字
    _Break          // break
    ...
    _Var            // var
)
```

## 源文件读取

主要通过nextch()函数，在缓存区里获取下一个字符。为什么不直接用io/ioutil.ReadAll(), 然后一个字符一个字符处理，非要自己造轮子呢，其实主要原因我们还需要获取当前的行和列信息。

### source数据结构

这里的b指内容开始位置，chw是指读取的字符宽度，e是内容结束位置，r是当前读取的位置。

```
type source struct {
    in     io.Reader
    errh  func(line, col uint, msg string)

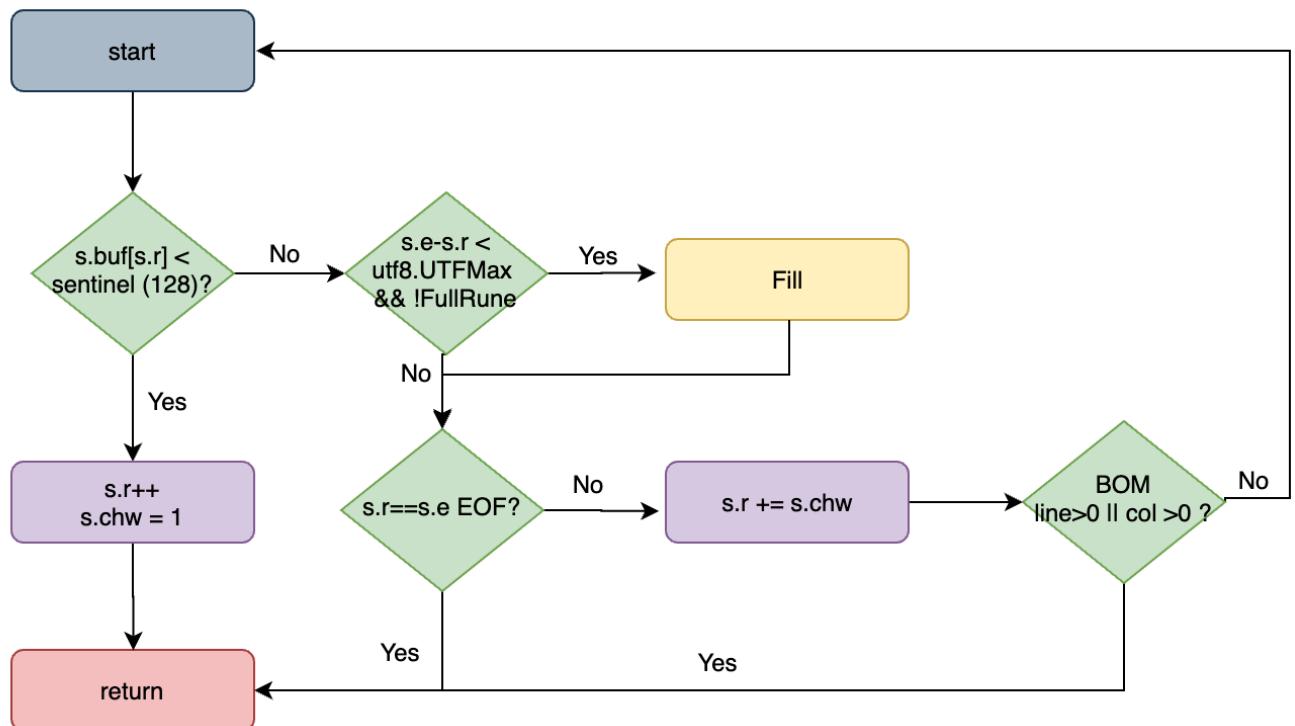
    buf      []byte // 源缓冲区
    ioerr   error   // 挂起的I/O错误，或为零
    b, r, e  int     // 缓冲区索引
    line, col uint   // ch的源位置（基于0）
    ch      rune    // 最近读取的字符
    chw     int     // 字符宽度
}
```

### buf结构定义

```
+----- content in use -----+
|                               |
v                               v
buf [...read...|...segment...|ch|...unread...|s|...free...]
^           ^   ^
|           |   |
b           r-chw  r
                           e
```

Invariant:  $-1 \leq b < r \leq e < \text{len}(\text{buf}) \&& \text{buf}[e] == \text{sentinel}$

## NextCh逻辑



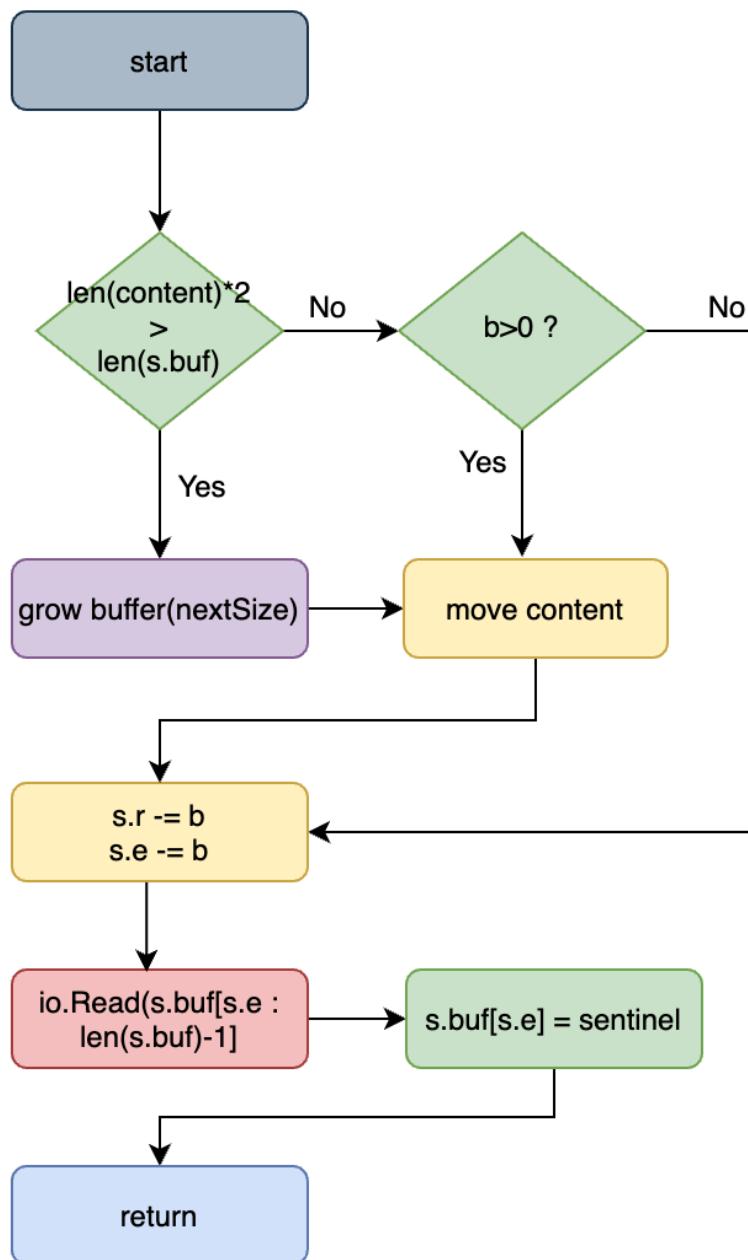
注意这里的sentinel,因为至少一个ASCII字符, s.ch初始值是一个128的ascii码, `const sentinel = utf8.RuneSelf.`

大于sentinel(哨兵), 会进行填充。

`FullRune`报告p中的字节是否以rune的完整UTF-8编码开头。一个无效的编码被认为是一个完整的符文, 因为它会转换为一个宽度为1的错误符文。

BOM表仅允许作为文件中的第一个字符。如果不是返回错误,否则跳到开始处。

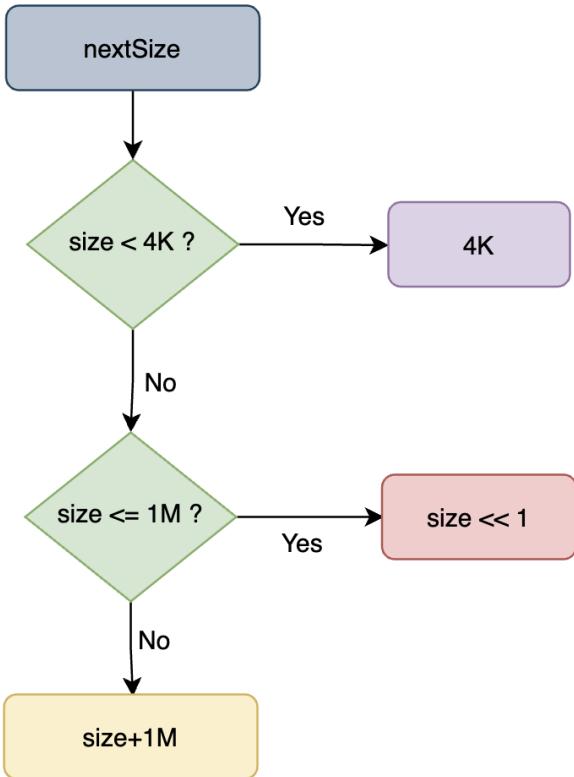
## Fill逻辑



如果当前内容段\*2比buf还要大的话，需要进行buf扩容，这里的b分代表两种意思：1.s.b>=0里b=s.b，否则b=s.r。

这里需要注意的s.b=-1的时候，`func (s *source) stop() { s.b = -1 }`，比如遇到 `for s.ch == ' ' || s.ch == '\t' || s.ch == '\n' && !nlsemi || s.ch == '\r' {`，需要跳过，这时候需要暂停来重新计算段的开始位置。

**nextSize:**



`const min = 4 << 10 // 4K`: 最小缓冲区大小

`const max = 1 << 20 // 1M`: 最大缓冲区大小仍然是原来的两倍

## Token解析

这里要介绍NFA(Non-Deterministic Finite State Automata)不确定的有穷自动机和DFA(Deterministic Finite State Automata)确定的有穷自动机)。

DFA: 一个DFA有有穷个状态 , 主要分为三种状态: 初始状态(initial state): 自动机开始的状态; 终止状态(final state): 一个DFA至少有一个终止状态; 中间状态。

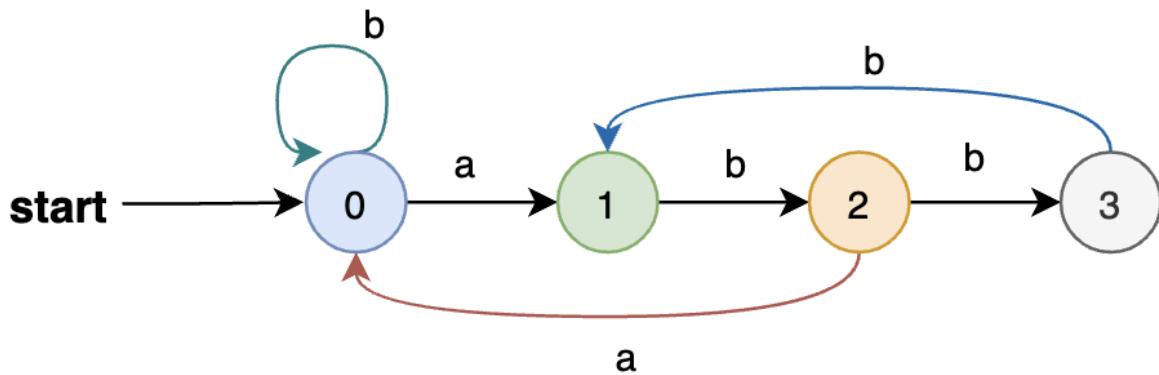
NFA: 对一个输入符号, 有两种或两种以上可能对状态, 所以是不确定的。

## DFA

$$M = (S, \Sigma, \delta, s_0, F)$$

- **S**: 有穷状态集
- $\Sigma$ : 输入字母表, 即输入符号集合(假设 $\epsilon$ 不是 $\Sigma$ 中的元素)
- $\delta$ : 将  $S \times \Sigma$  映射到  $S$  的转换函数  
 $\forall s \in S, a \in \Sigma, \delta(s, a)$  表示从状态  $s$  出发, 沿着标记为  $a$  的边所能到达的状态
- $s_0$ : 开始状态(初始状态),  $s_0 \in S$
- $F$ : 接收状态(或终止状态)集合,  $F \subseteq S$

DFA案例：



转换表

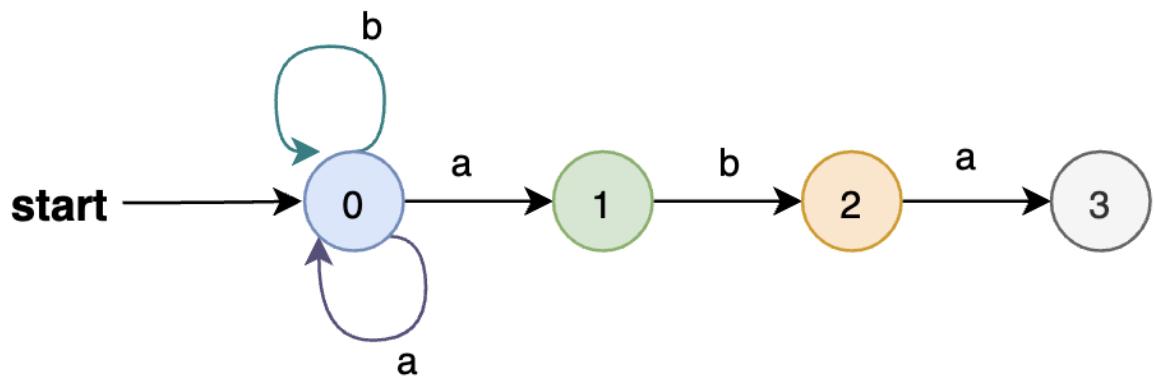
状态	a	b
0	1	0
1	$\emptyset$	2
2	0	3
3	$\emptyset$	1

NFA

$$M = (S, \Sigma, \delta, s_0, F)$$

- **S**: 有穷状态集
- $\Sigma$ : 输入字母表, 即输入符号集合(假设 $\varepsilon$ 不是 $\Sigma$ 中的元素)
- $\delta$ : 将  $S \times \Sigma$  映射到  $2^S$  的转换函数  
 $\forall s \in S, a \in \Sigma, \delta(s, a)$  表示从状态  $s$  出发, 沿着标记为  $a$  的边所能到达的状态 **集合**
- $s_0$ : 开始状态(初始状态),  $s_0 \in S$
- **F**: 接收状态(或终止状态)集合,  $F \subseteq S$

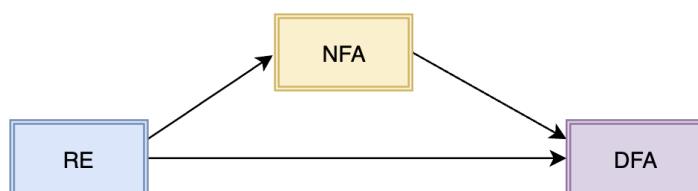
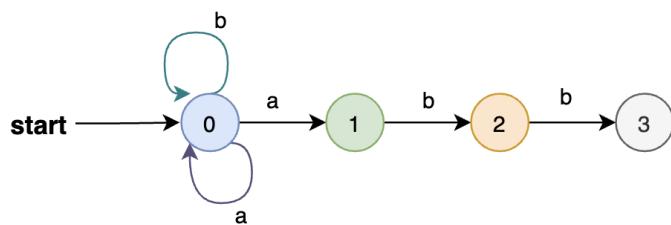
NFA案例：



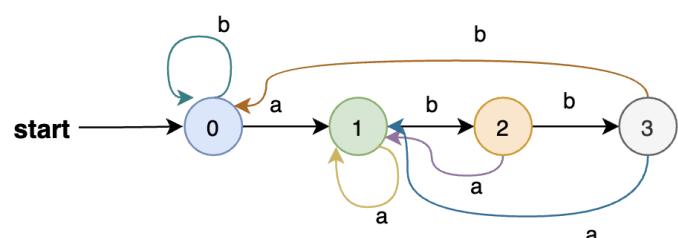
转换表

状态	a	b
0	{0,1}	{0}
1	$\emptyset$	{2}
2	{3}	$\emptyset$
3	$\emptyset$	$\emptyset$

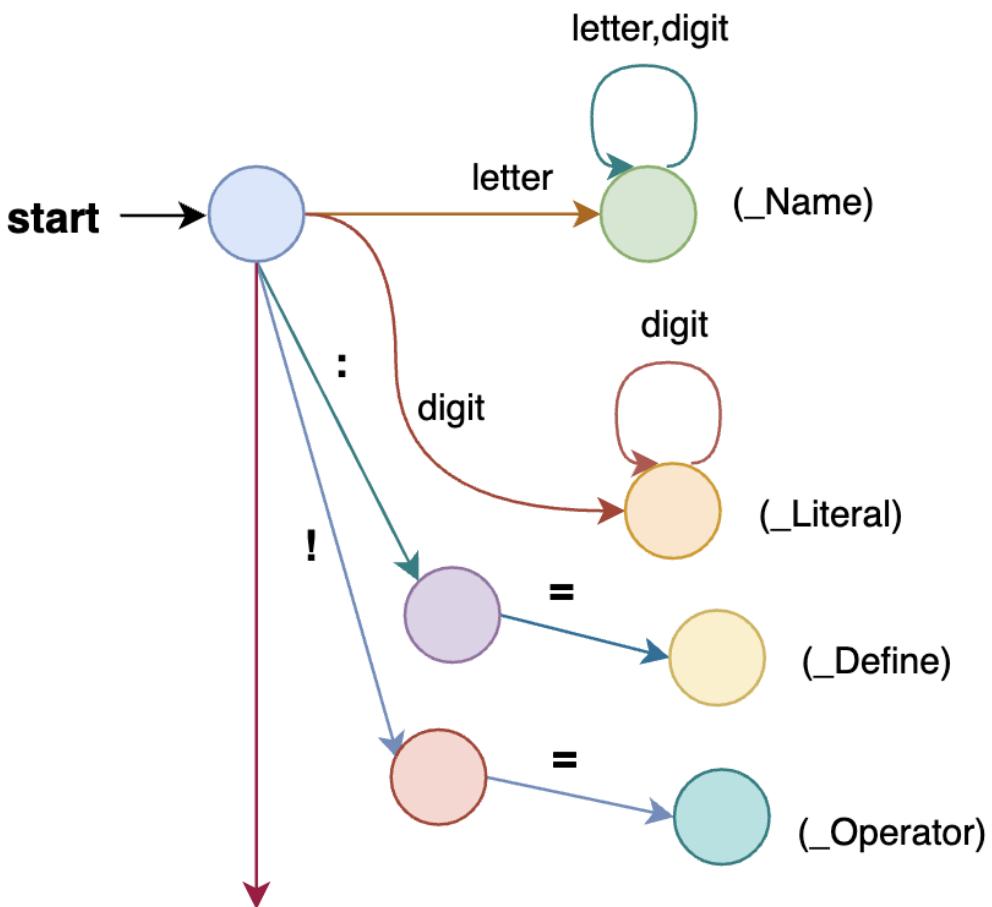
从正则表达式到FA



$$r = (alb)^*abb$$



实现Token的DFA



这里需要注意的就是\*, 默认会token类型是乘法, 后续语法分析会来确认是否是指针。

scanner结构体:

```

type scanner struct {
    source // 源文件
    mode   uint
    nlsemi bool // 如果设置'\n'和EOF转换为';'

    line, col uint // 当前token, 在调用next()后有效
    blank     bool // 行到列是空白的
    tok       token
    lit       string // 字面值, 如果 tok 是 _Name、_Literal 或 _Semi ("分号"、
    "换行符"或"EOF") 则有效; 如果 bad 为真, 则可能格式不正确
    bad       bool // 如果 tok 是 _Literal 则有效, 如果发生语法错误则为 true,
    lit 可能格式错误
    kind      LitKind // 标识符种类, 如果 tok 是 _Literal 则有效
    op        Operator // 操作符, 如果 tok 是 _Operator、_AssignOp 或 _IncOp 则有
    效
}

```

```
    prec      int      // 优先级, 如果 tok 是 _Operator、_AssignOp 或 _IncOp 则有效
}
```

主要负责字符扫描来确认token序列，下面是next()函数：

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

    redo:
        // 跳过空白
        s.stop()
        startLine, startCol := s.pos()
        for s.ch == ' ' || s.ch == '\t' || s.ch == '\n' && !nlsemi || s.ch == '\r' {
            s.nextch()
        }

        // 开始处理tokenk, 记录当前位置
        s.line, s.col = s.pos()
        s.blank = s.line > startLine || startCol == colbase
        s.start()
        .....
        // 处理对应token, 比如字面值, 数字等
        switch s.ch {
        .....
        case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
            s.number(false)

        case "'":
            s.stdString()
        .....
        return
    }
}
```

标识符DFA

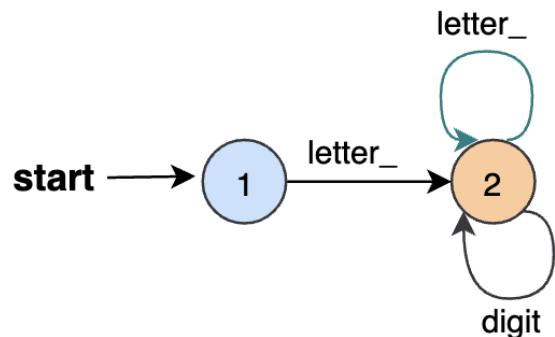
标识符的正则定义：

digit--->0-9

```

letter_ ---->a-zA-Z_
id ---> letter_(letter_|digit)*

```



```

if isLetter(s.ch) || s.ch >= utf8.RuneSelf && s.atIdentChar(true) {
    s.nextch()
    s.ident()
    return
}

func (s *scanner) ident() {
    for isLetter(s.ch) || isDecimal(s.ch) {
        s.nextch()
    }
    ...
    s.lit = string(lit)
    s.tok = _Name
}

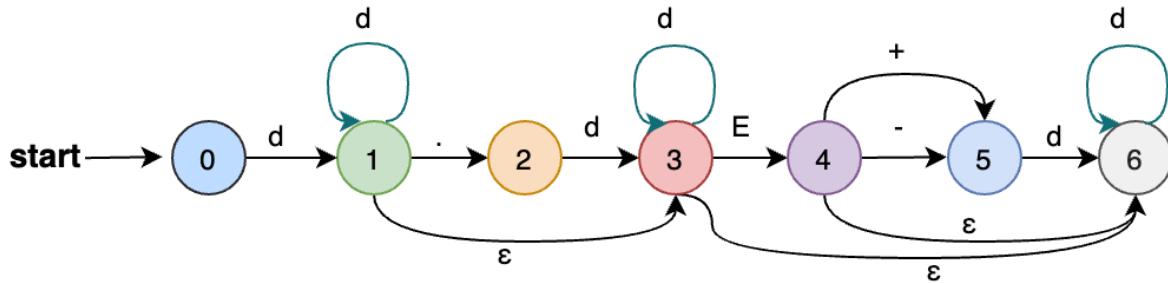
```

识别无符号数

格式 : 123.123E+10

状态	说明
0	初始状态
1	整数部分
2	小数点（左有整数）
3	小数部分
4	字符E
5	指数符号
6	指数数字

DFA:



DFA实现方式可参考Leetcode65题，go并没有采用DFA表驱动法来实现，而是用朴素方式实现过程。

```
func (s *scanner) number(seenPoint bool) {
    ok := true
    kind := IntLit
    base := 10          // number base
    prefix := rune(0)  // one of 0 (decimal), '0' (0-octal), 'x', 'o', or 'b'
    digsep := 0         // bit 0: digit present, bit 1: '_' present
    invalid := -1       // index of invalid digit in literal, or < 0

    // integer part
    if !seenPoint {
        if s.ch == '0' {
            s.nextch()
            switch lower(s.ch) {
            case 'x':
                s.nextch()
                base, prefix = 16, 'x'
            case 'o':
                s.nextch()
                base, prefix = 8, 'o'
            case 'b':
                s.nextch()
                base, prefix = 2, 'b'
            default:
                base, prefix = 8, '0'
                digsep = 1 // leading 0
            }
        }
    }
}
```

```
        }

        .....

    }

// 浮点数

if seenPoint {
    kind = FloatLit
    digsep |= s.digits(base, &invalid)
}

if digsep&1 == 0 && ok {
    s.errorf("%s literal has no digits", baseName(base))
    ok = false
}

// 指数

if e := lower(s.ch); e == 'e' || e == 'p' {
    if ok {
        switch {
            case e == 'e' && prefix != 0 && prefix != '0':
                s.errorf("%q exponent requires decimal mantissa", s.ch)
                ok = false
            case e == 'p' && prefix != 'x':
                s.errorf("%q exponent requires hexadecimal mantissa", s.ch)
                ok = false
        }
    }
    .....
} else if prefix == 'x' && kind == FloatLit && ok {
    s.errorf("hexadecimal mantissa requires a 'p' exponent")
    ok = false
}

// 复数

if s.ch == 'i' {
    kind = ImagLit
    s.nextch()
}

s.setLit(kind, ok) // 设置token
.....
```

```

    s.bad = !ok // correct s.bad
}

```

## 语法分析

编译都是从解析源代码开始，词法分析的作用就是解析源代码文件，将文件中的字符串转换成Token序列，为后续处理和解析提供准备工作。语法分析会检查任何语法错误，如果出错编译过程就会中止。

语法分析模式：

1. 自顶向下：可以被看作找到当前输入流最左推导的过程，对于任意一个输入流，根据当前的输入符号，确定一个生产规则，使用生产规则右侧的符号替代相应的非终结符向下推导。
2. 自底向上：语法分析器从输入流开始，每次都尝试重写最右侧的多个符号，这其实是说解析器会从最简单的符号进行推导，在解析的最后合并成开始符号。

相关术语定义：

名词	说明
终结符集合 T (terminal set)	一个有限集合，其元素称为 终结符
非终结符集合 N (non-terminal set)	一个有限集合，与 T 无公共元素，其元素称为 非终结符
符号集合 V (alphabet)	T 和 N 的并集，其元素称为 符号(symbol)。因此终结符和非终结符都是符号。符号可用字母：A, B, C, X, Y, Z, a, b, c 等表示
符号串(a string of symbols)	一串符号，如 X1 X2 ... Xn 。只有终结符的符号串称为 句子。空串（不含任何符号）也是一个符号串，用 $\epsilon$ 表示。符号串一般用小写字母 u, v, w 表示
产生式(production)	一个描述符号串如何转换的规则。对于上下文本无关语法，其固定形式为：A $\rightarrow$ u，其中 A 为非终结符，u 为一个符号串
产生式集合 P (production set)	一个由有限个产生式组成的集合
展开(expand)	一个动作：将一个产生式 A $\rightarrow$ u 应用到一个含有 A 的符号串 vAw 上，用 u 代替该符号串中的 A，得到一个新的符号串 vuw。
折叠(reduce)	一个动作：将一个产生式 A $\rightarrow$ u 应用到一个含有 u 的符号串 vuw 上，用 A 代替该符号串中的 u，得到一个新的符号串 vAw
起始符号 S (start symbol)	N 中的一个特定的元素
推导(derivate)	一个过程：从一个符号串 u 开始，应用一系列的产生式，展开到另一个的符号串 v。若 v 可以由 u 推导得到，则可写成：u $\Rightarrow$ v
上下文本无关语法 G (context-free grammar, CFG)	一个 4 元组：(T, N, P, S)，其中 T 为终结符集合，N 为非终结符集合，P 为产生式集合，S 为起始符号。一个句子如果能从语法 G 的 S 推导得到，可以直接称此句子由语法 G 推导得到，也可称此句子符合这个语法，或者说此句子属于 G 语言。G 语言(G language) 就是语法 G 推导出来的所有句子的集合

	, 有时也用 $G$ 代表这个集合
解析(parse)	也称为分析, 是一个过程: 给定一个句子 $s$ 和语法 $G$ , 判断 $s$ 是否属于 $G$ , 如果是, 则找出从起始符号推导得到 $s$ 的全过程。推导过程中的任何符号串(包括起始符号和最终的句子)都称为 中间句子(working string)

## 自顶向下

从分析树的顶部(根节点)向底部(叶节点)方向构造分析树。

### 最左推导

最左推导中, 总是选择每个句型的最左非终结符进行替换。

文法	推导过程
$E \rightarrow E + E$	$E \Rightarrow E + E$
$E \rightarrow E * E$	$\Rightarrow id + E$
$E \rightarrow ( E )$	$\Rightarrow id + ( E )$
$E \rightarrow id$	$\Rightarrow id + ( E + E )$
输入	$\Rightarrow id + ( id + E )$
$id + ( id + id )$	$\Rightarrow id + ( id + id )$

### 最右推导

最右推导中, 总是选择每个句型的最右非终结符进行替换。

文法	推导过程
$E \rightarrow E + E$	$E \Rightarrow E + E$
$E \rightarrow E * E$	$\Rightarrow E + ( E )$
$E \rightarrow ( E )$	$\Rightarrow E + ( E + E )$
$E \rightarrow id$	$\Rightarrow E + ( E + id )$
输入	$\Rightarrow E + ( id + id )$
$id + ( id + id )$	$\Rightarrow id + ( id + id )$

自顶向下的语法分析采用最左推导方式:

总是选择每个句型的最左非终结符进行替换

根据输入流中的下一个终结符, 选择最左非终结符的一个候选式

案例：

文法	语法树
$E \rightarrow T E'$	
$E' \rightarrow + T E'   \epsilon$	
$T \rightarrow F T'$	
$T' \rightarrow * F T'   \epsilon$	
$F \rightarrow (E)   id$	
输入	
$id + id * id$	

自顶向下实现方式：

1. 递归下降分析
  - a. 由一组过程组成，每个过程对应一个非终结符
  - b. 从文法开始符号S对应的过程开始，其中递归调用文法中其它非终结符对应的过程。如果S对应的过程体恰好扫描了整个输入串，则成功完成语法分析，在递归过程中可能需要进行回溯。
2. 预测分析
  - a. 预测分析是递归下降分析技术的一个特例，通过在输入中向前看固定个数（通常是一个）符号来选择正确的A-产生式，解析了性能比较低的回溯问题。

问题

1. 同一非终结符的多个候选式存在共同前缀，将导致回溯现象  
文法G:  $S \rightarrow aAd \mid aBe$

2. 左递归文法会使递归下降分析器陷入无限循环

文法	最左	推导过程
$E \rightarrow E + T \mid E - T \mid T$		$E \Rightarrow E + T$
$T \rightarrow T^* F \mid T \mid F \mid F$		$\Rightarrow E + T + T$
$F \rightarrow (E) \mid id$		$\Rightarrow E + T + T$

		推导 ↓	=> ...
输入			
id + id * id			

- 含有  $A \rightarrow A\alpha$  形式产生式的文法称为直接左递归。
- 如果一个文法中有一个非终结符  $A$  使得对某个串  $\alpha$  存在一个推导  $A \Rightarrow^+ A\alpha$ , 那么这个文法就是左递归的。
- 经过两步或两步以上推导产生的左递归称为间接左递归。

### 消除直接左递归

消除过程就是把左递归转换成右递归

文法	推导过程	
$A \rightarrow A\alpha \mid \beta (\alpha \neq \epsilon, \beta \text{不以 } A \text{ 开头})$	最 左 推 导 ↓	$A \Rightarrow A\alpha$
$A \rightarrow \beta A'$		$\Rightarrow A\alpha\alpha$
$A' \rightarrow \alpha A' \mid \epsilon$		$\Rightarrow A\alpha\alpha$
		$\Rightarrow \dots$
输入		$\Rightarrow A\alpha\dots\alpha$
id + id * id		$\Rightarrow \beta\alpha\dots\alpha$

### 消除间接左递归

文法	推导过程	
$S \rightarrow A\alpha \mid b$	最 左 推 导 ↓	$S \Rightarrow A\alpha$
$A \rightarrow A c \mid S d \mid \epsilon$		$\Rightarrow Sd\alpha$
将 $S$ 的定义代入 $A$ -产生式, 得:		$\Rightarrow \dots$
$A \rightarrow A c \mid A\alpha d \mid b d \mid \epsilon$		
消除 $A$ -产生式的直接左递归, 得:		
$A \rightarrow b d A' \mid A'$		
$A' \rightarrow c A' \mid \alpha d A' \mid \epsilon$		

### 提取左公因子

通过改写产生式来推迟决定, 等读入了足够多的输入, 获得足够信息后再做出正确的选择

文法G	文法G'
$S \rightarrow \alpha A d \mid \alpha B e$	$S \rightarrow \alpha S'$
$A \rightarrow c$	$S' \rightarrow A d \mid B e$
$B \rightarrow b$	$A \rightarrow c$
	$B \rightarrow b$

## 自底向上

从分析树的底部(叶节点)向顶部(根节点)方向构造分析树, 自底向上语法采用最左归约方式(反向构造最右推导), 自底向上语法分析的通用构架叫做移入-归约分析(shift-reduce parsing)。

### 移入归约分析

文法	栈	剩余输入	动作
$E \rightarrow E + E$	\$	$id + (id + id) \$$	
$E \rightarrow E * E$	\$ id	$+ (id + id) \$$	移入
$E \rightarrow (E)$	E	$+ (id + id) \$$	归约: $E \rightarrow id$
$E \rightarrow id$	E +	$(id + id) \$$	移入
	E + (	$id + id) \$$	移入
	E + ( id	$+ id) \$$	移入
	E + ( E	$+ id) \$$	归约: $E \rightarrow id$
	E + ( E +	$id) \$$	移入
	E + ( E + id	) \$	移入
	E + ( E + E	) \$	归约: $E \rightarrow id$
	E + ( E	) \$	归约: $E \rightarrow E + E$
	E + ( E )	\$	移入
	E + E	\$	归约: $E \rightarrow (E)$
	E	\$	归约: $E \rightarrow E + E$

## 问题

移入-归约分析中也会存在错误地识别句柄。

文法
$\langle S \rangle \rightarrow \text{var } \langle \text{IDS} \rangle : \langle T \rangle$
$\langle \text{IDS} \rangle \rightarrow i$
$\langle \text{IDS} \rangle \rightarrow \langle \text{IDS} \rangle, i$
$\langle T \rangle \rightarrow \text{real}   \text{int}$
输入
$\text{var } i_A, i_B : \text{real}$

## 文法

### 文法分类体系

- 0型文法 (Type-0 Grammar)
- 1型文法 (Type-1 Grammar)
- 2型文法 (Type-2 Grammar)
- 3型文法 (Type-3 Grammar)

### 0型文法

定义:  $\alpha \rightarrow \beta$

- 无限制文法/短语结构文法(Phrase Structure Grammar, PSG )
- $\forall \alpha \rightarrow \beta \in P, \alpha$ 中至少包含1个非终结符

### 1型文法

定义:  $\alpha \rightarrow \beta$

上下文有关文法(Context-Sensitive Grammar , CSG ), CSG中不包含 $\epsilon$ -产生式

- $\forall \alpha \rightarrow \beta \in P, |\alpha| \leq |\beta|$
- 产生式的一般形式:  $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$  ( $\beta \neq \epsilon$ )

### 2型文法

定义:  $\alpha \rightarrow \beta$

上下文无关文法 (Context-Free Grammar, CFG )

- $\forall \alpha \rightarrow \beta \in P, \alpha \in V_n$
- 产生式的一般形式: $A \rightarrow \beta$

### 3型文法

定义:  $\alpha \rightarrow \beta$

正则文法 (Regular Grammar, RG )

- 右线性(Right Linear)文法:  $A \rightarrow wB$  或  $A \rightarrow w$
- 左线性(Left Linear)文法:  $A \rightarrow Bw$  或  $A \rightarrow w$
- 左线性文法和右线性文法都称为正则文法

## 二义性

如果一个文法可以为某个句子生成多棵分析树，则称这个文法是二义性的

文法:

$\langle S \rangle \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{other}$

LR(0)一般采用优先级和结合性来解决冲突

## 文法分析

### 文法关键函数

#### FOLLOW

非终结符A的后继符号集，可能在某个句型中紧跟在A后边的终结符a的集合，记为  $\text{FOLLOW}(A)$

$\text{FOLLOW}(A) = \{\alpha \mid S \Rightarrow^* \alpha A \alpha \beta, \alpha \in V_T, \alpha, \beta \in (V_T \cup V_N)^*\}$

如果 A 是某个句型的最右符号，则将结束符“\$”添加到  $\text{FOLLOW}(A)$  中

$\text{FOLLOW}(B) = \{\alpha, c\}$

文法G	输入
$S \rightarrow aBC$	
$B \rightarrow bC$	b
$B \rightarrow dB$	d
$B \rightarrow \epsilon$	{d, c}
$C \rightarrow c$	
$C \rightarrow a$	

#### SELECT

产生式  $A \rightarrow \beta$  的可选集是指可以选用该产生式进行推导时对应的输入符号的集合，记为  $\text{SELECT}(A \rightarrow \beta)$

- $\text{SELECT}(A \rightarrow \alpha\beta) = \{\alpha\}$
- $\text{SELECT}(A \rightarrow \epsilon) = \text{FOLLOW}(A)$

#### FIRST

- 串首终结符
  - 串首第一个符号，并且是终结符。简称首终结符
- 给定一个文法符号串  $\alpha$ ,  $\alpha$  的串首终结符集  $\text{FIRST}(\alpha)$  被定义为 可以从  $\alpha$  推导出的所有串首终结符构成的集合。如果  $\alpha \Rightarrow^* \epsilon$ , 那么  $\epsilon$  也在  $\text{FIRST}(\alpha)$  中

- 对于  $\forall \alpha \in (V_T \cup V_N)^*$ ,  $\text{FIRST}(\alpha) = \{ a \mid \alpha \Rightarrow^* a\beta, a \in V_T, \beta \in (V_T \cup V_N)^*\}$ ;
- 如果  $\alpha \Rightarrow^* \epsilon$ , 那么  $\epsilon \in \text{FIRST}(\alpha)$
- 产生式  $A \rightarrow \alpha$  的可选集  $\text{SELECT}$ 
  - 如果  $\epsilon \notin \text{FIRST}(\alpha)$ , 那么  $\text{SELECT}(A \rightarrow \alpha) = \text{FIRST}(\alpha)$
  - $\epsilon \in \text{FIRST}(\alpha)$ , 那么  $\text{SELECT}(A \rightarrow \alpha) = (\text{FIRST}(\alpha) - \{\epsilon\}) \cup \text{FOLLOW}(A)$

## CLOSURE

项目集闭包

$$\text{CLOSURE}(I) = I \cup \{ [B \rightarrow \cdot \gamma, b] \mid [A \rightarrow \alpha \cdot B\beta, a] \in \text{CLOSURE}(I), B \rightarrow \gamma \in P, b \in \text{FIRST}(\beta a)\}$$

## GOTO

$$\text{GOTO}(I, X) = \text{CLOSURE}(\{[A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X\beta, a] \in I\})$$

文法分析模式

## LL(1)

LL(1)文法, 第一个“L”表示从左向右扫描输入, 第二个“L”表示产生最左推导, 其中“1”表示在每一步中只需要向前看一个输入符号来决定语法分析动作

- 文法  $G$  是 LL(1) 的, 当且仅当  $G$  的任意两个具有相同左部的产生式  $A \rightarrow \alpha \mid \beta$  满足下面的条件:
  - 如果  $\alpha$  和  $\beta$  均不能推导出  $\epsilon$ , 则  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
  - $\alpha$  和  $\beta$  至多有一个能推导出  $\epsilon$
  - 如果  $\beta \Rightarrow^* \epsilon$ , 则  $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$ ;
    - 如果  $\alpha \Rightarrow^* \epsilon$ , 则  $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$ ;

## LR

LR文法(Knuth, 1963) 是最大的、可以构造出相应 移入-归约语法分析器的文法类

L: 对输入进行从左到右的扫描

R: 反向构造出一个最右推导序列

## SLR

SLR主要解决LR冲突问题, 引用 FOLLOW集处理冲突

SLR只是简单地考察下一个输入符号  $b$  是否属于与归约 项目  $A \rightarrow \alpha$  相关联的  $\text{FOLLOW}(A)$ , 但  $b \in \text{FOLLOW}(A)$  只是归约  $\alpha$  的一个必要条件, 而非充分条件

## LR(k)

需要向前查看  $k$  个输入符号的 LR 分析,  $k = 0$  和  $k = 1$  这两种情况具有实践意义 当省略( $k$ )时, 表示  $k = 1$ , 对于产生式  $A \rightarrow \alpha$  的归约, 在不同的使用位置,  $A$  会要求不同的后继符号

## LALR

寻找具有相同核心的 LR(1) 项集, 并将这些项集 合并为一个项集。所谓项集的核心就是其第一分量的集合

然后根据合并后得到的项集族构造语法分析表

如果分析表中没有语法分析动作冲突, 给定的文 法就称为 LALR(1) 文法, 就可以根据该分析表 进行语法分析

将一般形式为  $[A \rightarrow \alpha \cdot \beta, a]$  的项称为 LR(1) 项, 其中  $A \rightarrow \alpha\beta$  是一个产生式,  $a$  是一个终结符(这里将 \$ 视为一个特殊的终结符) 它表示在当前状态下,  $A$  后面必须紧跟的终结符, 称为该项的 展望符(lookahead)

## LL(1)分析案例

递归分析法：

文法	SELECT
$\langle \text{PROGRAM} \rangle \rightarrow \text{program } \langle \text{DECLIST} \rangle : \langle \text{TYPE} \rangle ; \langle \text{STLIST} \rangle \text{ end}$	
$\langle \text{DECLIST} \rangle \rightarrow \text{id } \langle \text{DECLISTN} \rangle$	
$\langle \text{DECLISTN} \rangle \rightarrow , \text{id } \langle \text{DECLISTN} \rangle$	
$\langle \text{DECLISTN} \rangle \rightarrow \epsilon$	{:}
$\langle \text{STLIST} \rangle \rightarrow \text{s } \langle \text{STLISTN} \rangle$	
$\langle \text{STLISTN} \rangle \rightarrow ; \text{s } \langle \text{STLISTN} \rangle$	
$\langle \text{STLISTN} \rangle \rightarrow \epsilon$	{end}
$\langle \text{TYPE} \rangle \rightarrow \text{real}$	
$\langle \text{TYPE} \rangle \rightarrow \text{int}$	

```
func Program(Token)
begin
    if Token != "program" then ERROR

    GetNext(Token)
    Declist(Token)

    if Token != ":" then ERROR

    GetNext(Token)
    Type(Token)

    GetNext(Token)
    if Token != ";" then ERROR

    GetNext(Token)
    Stlist(Token)

    if Token != "end" then ERROR

end
```

表驱动的预测分析法：

注意 FOLLOW 集, 如果  $\varepsilon \notin \text{FIRST}(\alpha)$ , 那么  $\text{SELECT}(A \rightarrow \alpha) = \text{FIRST}(\alpha)$ ,  
 $\varepsilon \in \text{FIRST}(\alpha)$ , 那么  $\text{SELECT}(A \rightarrow \alpha) = (\text{FIRST}(\alpha) - \{\varepsilon\}) \cup \text{FOLLOW}(A)$

文法	FIRST	FOLLOW
$E' \rightarrow TE'$	$\text{FIRST}(E) = \{( \text{id} \}$	$\text{FOLLOW}(E) = \{ \$ \}$
$E' \rightarrow +TE' \varepsilon$	$\text{FIRST}(E') = \{ + \varepsilon \}$	$\text{FOLLOW}(E') = \{ \$ \}$
$T \rightarrow FT'$	$\text{FIRST}(T) = \{ ( \text{id} \}$	$\text{FOLLOW}(T) = \{ +\$ \}$
$T' \rightarrow *FT' \varepsilon$	$\text{FIRST}(T') = \{ * \varepsilon \}$	$\text{FOLLOW}(T') = \{ + \$ \}$
$F \rightarrow (E) \text{id}$	$\text{FIRST}(F) = \{ ( \text{id} \}$	$\text{FOLLOW}(F) = \{ *+\$ \}$

文法	SELECT
$E' \rightarrow TE'$	$\text{SELECT}(1) = \{ (\text{id} \}$
$E' \rightarrow +TE'$	$\text{SELECT}(2) = \{ + \}$
$E' \rightarrow \varepsilon$	$\text{SELECT}(3) = \{ \$ \}$
$T \rightarrow FT'$	$\text{SELECT}(4) = \{ (\text{id} \}$
$T' \rightarrow *FT'$	$\text{SELECT}(5) = \{ * \}$
$T' \rightarrow \varepsilon$	$\text{SELECT}(6) = \{ + \$ \}$
$F \rightarrow (E)$	$\text{SELECT}(7) = \{ ( \}$
$F \rightarrow \text{id}$	$\text{SELECT}(8) = \{ \text{id} \}$

非终结符	输入符号					
	id	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

栈	剩余输入	输出
E \$	id+id*id \$	
TE' \$	id+id*id \$	E→TE'
FT'E' \$	id+id*id \$	T→FT'
idT'E' \$	id+id*id \$	F→id
T'E' \$	+id*id \$	
E' \$	+id*id \$	T'→ε
+TE' \$	+id*id \$	E'→+TE'
TE' \$	id*id \$	
FT'E' \$	id*id \$	T→FT'
idT'E' \$	id*id \$	F→id
T'E' \$	*id \$	
*FT'E' \$	*id \$	T'→*FT'
FT'E' \$	id \$	
idT'E' \$	id \$	F→id
T'E' \$	\$	
E'\$	\$	T'→ε
\$	\$	E'→ε

### 递归、非递归的预测分析法

	递归	非递归
程序规模	程序规模较大，不需要转入分析表	主控程序规模较小，需转入分析表较小
直观性	较好	较差
效率	较低	分析时间大约正比于待分析程序的长度

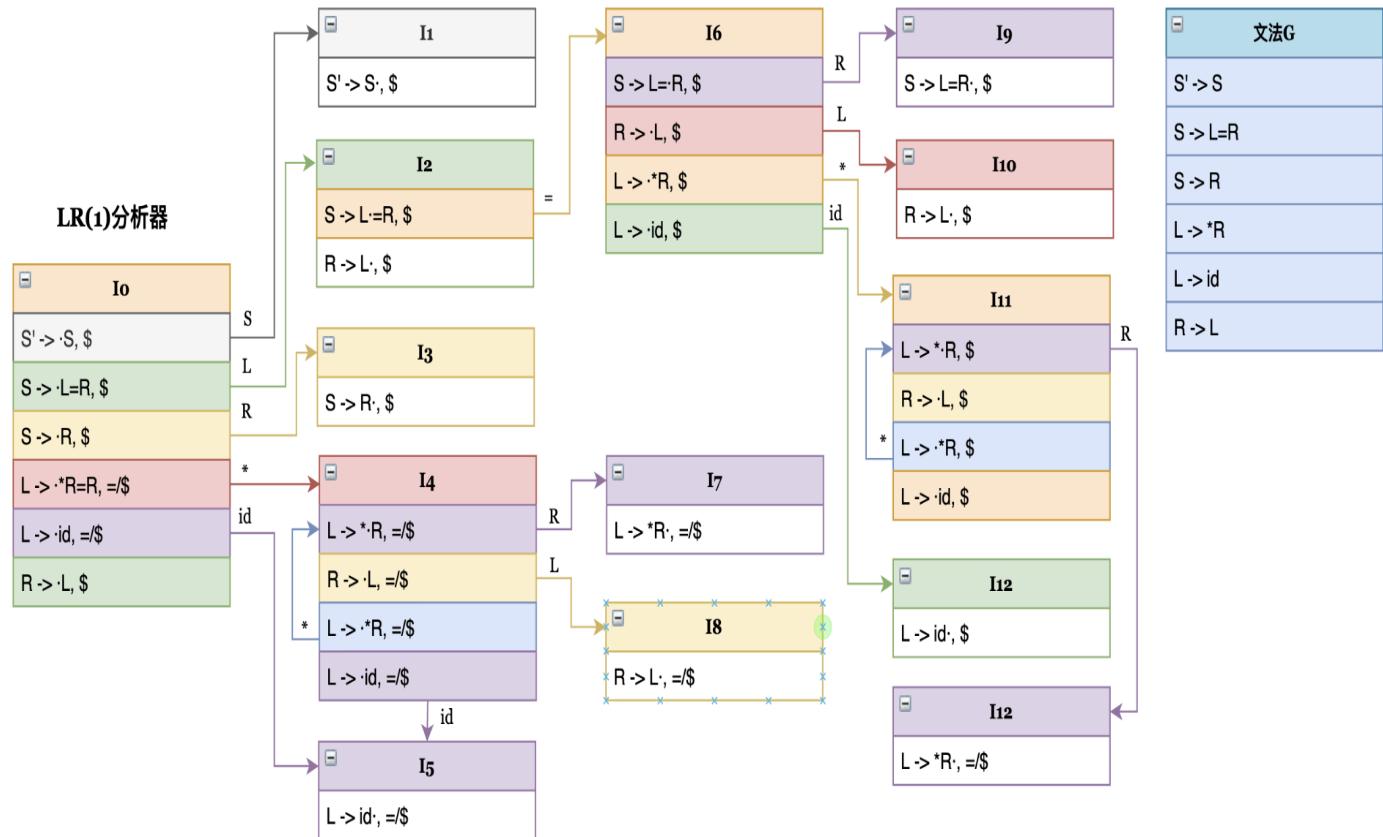
### 预测分析法实现步骤

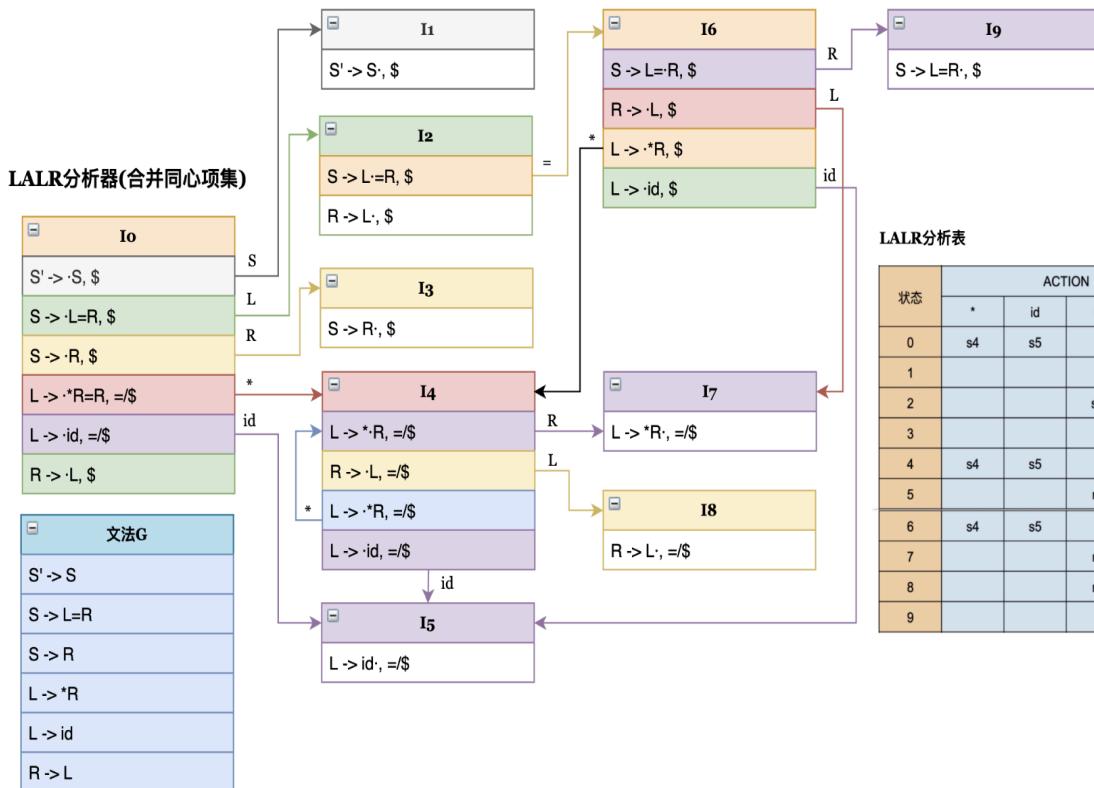
- 构造文法
- 改造文法:消除二义性、消除左递归、消除回溯
- 求每个变量的FIRST集和FOLLOW集, 从而求得每个候选式的SELECT集
- 检查是不是 LL(1) 文法。若是, 构造预测分析表

- 对于递归的预测分析，根据预测分析表为每一个非终结符编写一个过程；对于非递归的预测分析，实现表驱动的预测分析算法

### LALR分析案例

S指移入操作, R是规约操作





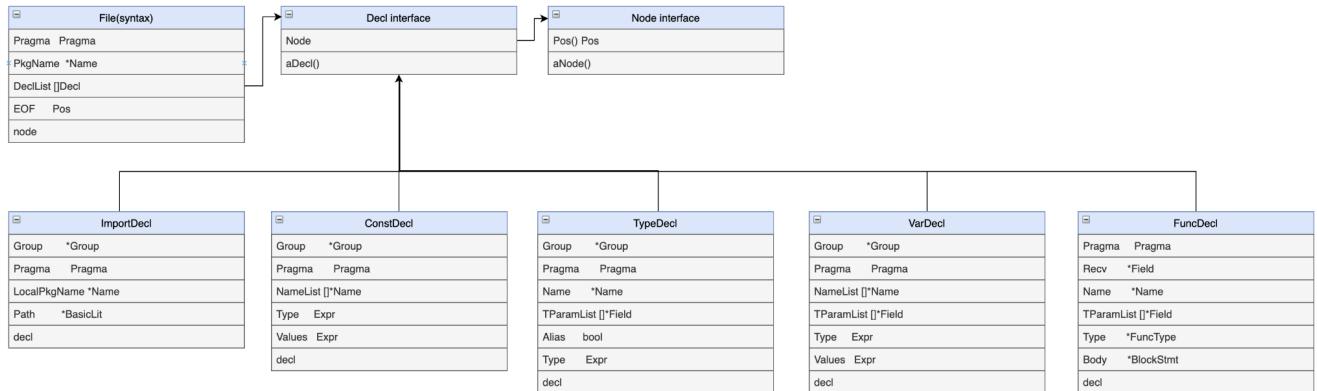
LALR分析表

sn: 将符号a, 状态n压入栈  
sn: 用第n个产生式进行归约

状态	ACTION				GOTO		
	*	id	=	\$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s4	s5				8	9
7			r3	r3			
8			r5	r5			
9				r1			

# Parser

数据结构：



这里需要注释的是Decl是一个接口类型, ImportDecl、ConstDecl、TypeDecl、VarDecl及FuncDecl都是继承此接口的。

文法定义

[

标记一个中括号表达式的开始。要匹配 `[` 请使用 `\[`。

( )

标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符, 请使用 `\(` 和 `\)`。

{

标记限定符表达式的开始。要匹配 `{` 请使用 `\{`。

`SourceFile = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .`

**ImportDecl**  
`ImportSpec = [ "." | PackageName ] ImportPath .`  
`ImportPath = string_lit .`

**TopLevelDecl**

```

ConstSpec = IdentifierList [ [ Type ] "=" ExpressionList ] .
TypeSpec = identifier [ TypeParams ] [ "=" ] Type .
VarSpec = IdentifierList ( Type [ "=" ExpressionList ] | "="
ExpressionList ) .

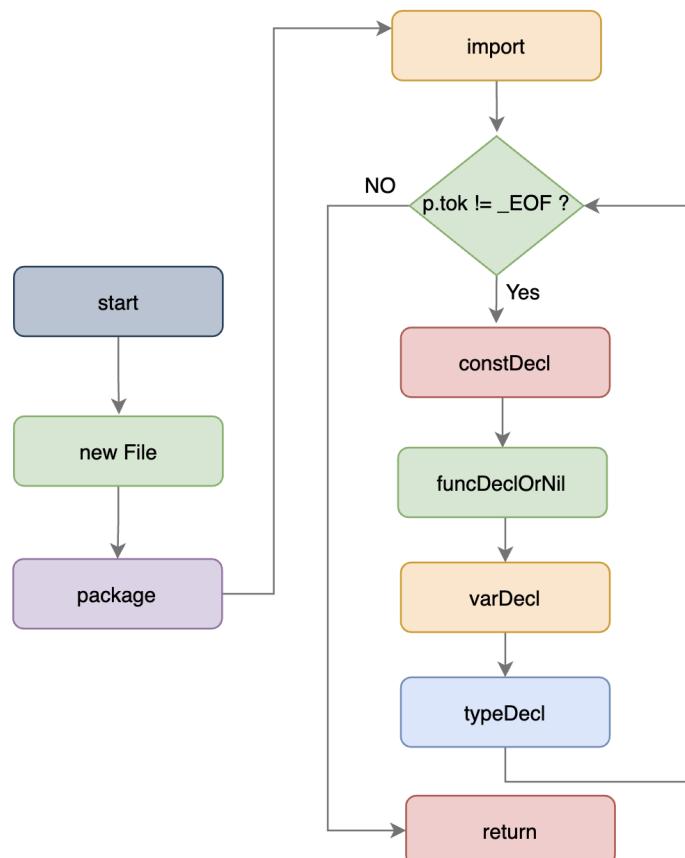
FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature )
.
FunctionName = identifier .
Function      = Signature FunctionBody .

MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
Receiver      = Parameters .

```

parser主要是进行自顶向下的文法分析，比如处理对应token的声明，表达式等。fileOrNil是文件文法入口函数。

整体执行流程



// PackageClause指包名 ImportDecl声明，TopLevelDecl指头部常量、类型、变量及函数声明。  
// SourceFile = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .  
**func** (p \*parser) fileOrNil() \*File {

```

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

// PackageClause
if !p.got(_Package) {
    p.syntaxError("package statement must be first")
    return nil
}

f.Pragma = p.takePragma()
f.PkgName = p.name()
p.want(_Semi)
.....
// { ImportDecl ";" }
for p.got(_Import) {
    f.DeclList = p.appendGroup(f.DeclList, p.importDecl)
    p.want(_Semi)
}

// { TopLevelDecl ";" }
for p.tok != _EOF {
    switch p.tok {
        case _Const:
            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()
            if d := p.funcDeclOrNil(); d != nil {
                f.DeclList = append(f.DeclList, d)
            }

        default:
            .....
    }
}

```

```

        continue
    }
}

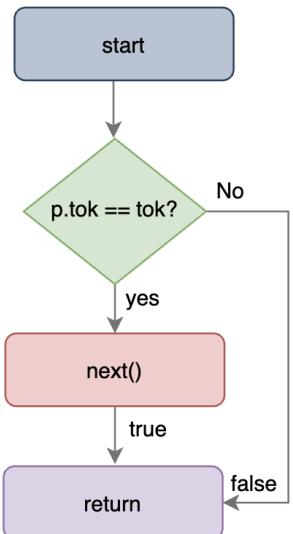
.....
f.Lines = p.line
return f
}

```

## 文法函数

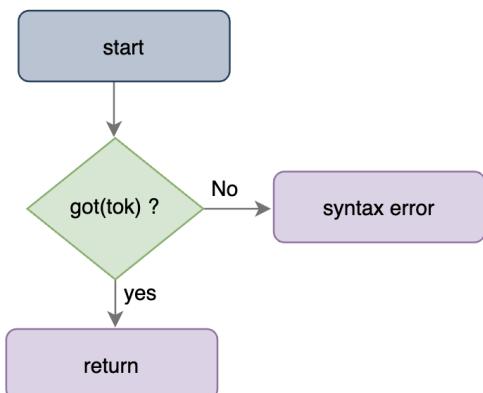
### *got*

`got()`函数是用于快速判断一些语句中的关键字，如果当前解析器的`token`是传入`token`就会直接跳过该`token`并返回`true`。



### *want*

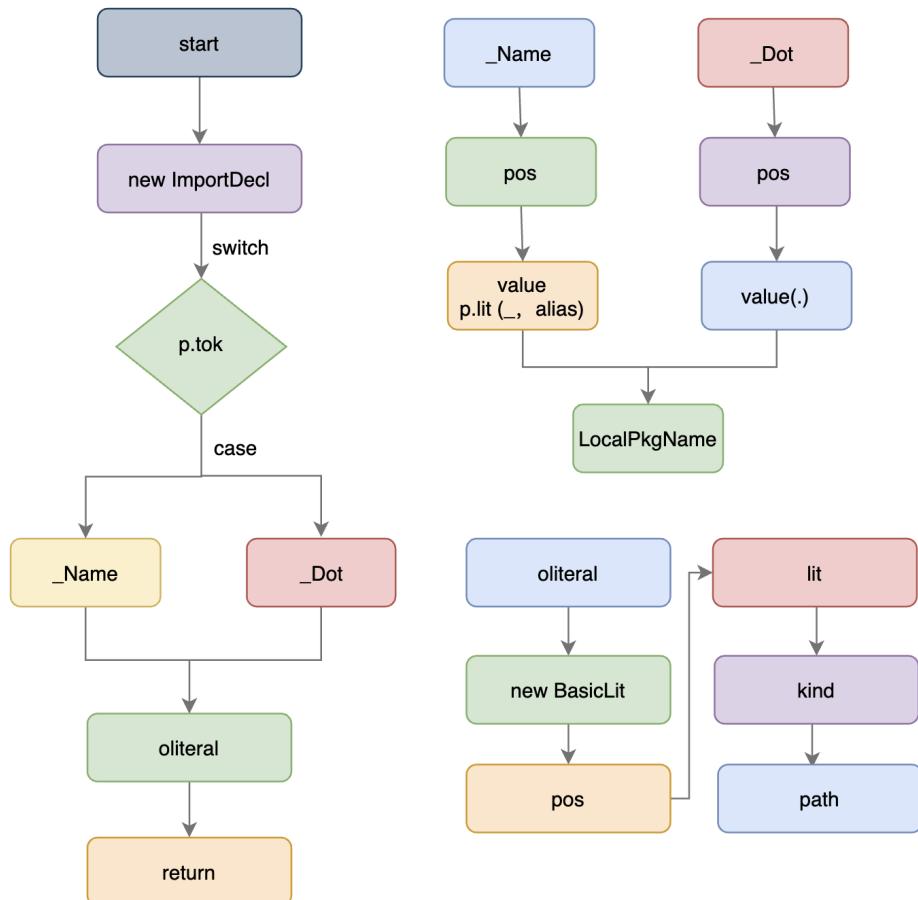
`want()`函数，如果当前`token`不是我们期望的，就会返回语法错误结束这次编译。



*importDecl*

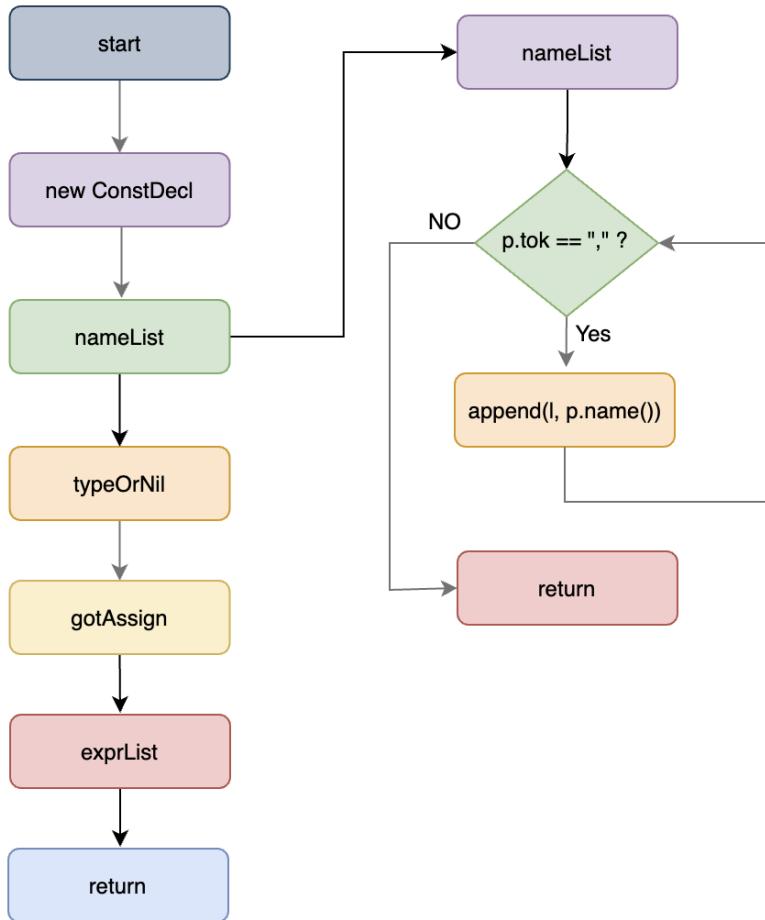
### 文法定义

```
ImportSpec = [ "." | PackageName ] ImportPath .
ImportPath = string_lit .
```



*constDecl*

```
ConstSpec = IdentifierList [ [ Type ] "=" ExpressionList ] .
```

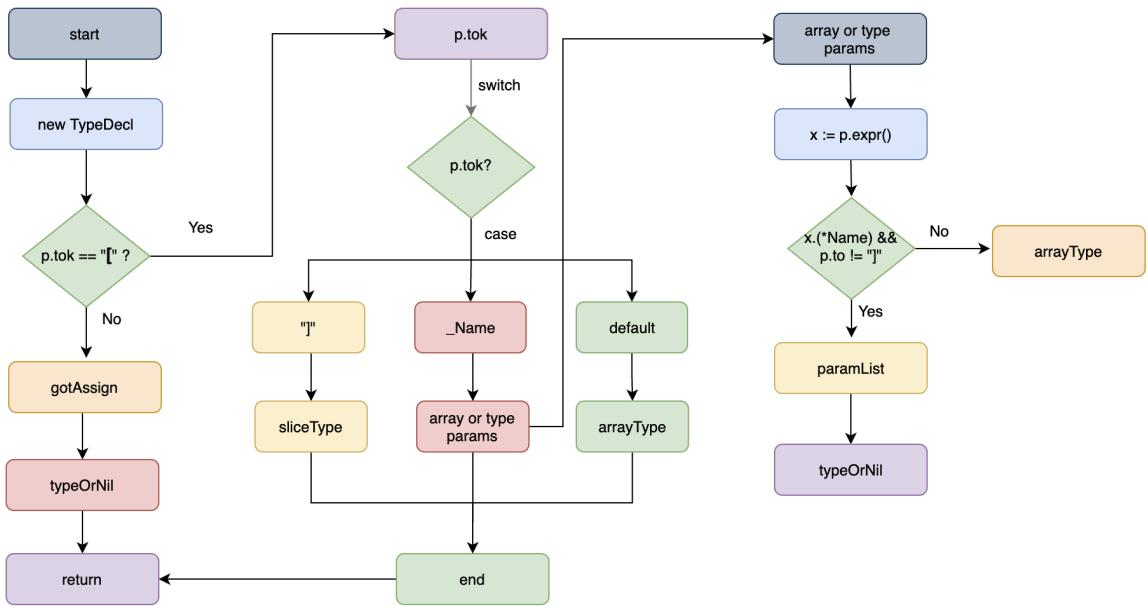


*typeDecl*

TypeSpec = identifier [ TypeParams ] [ "=" ] Type .

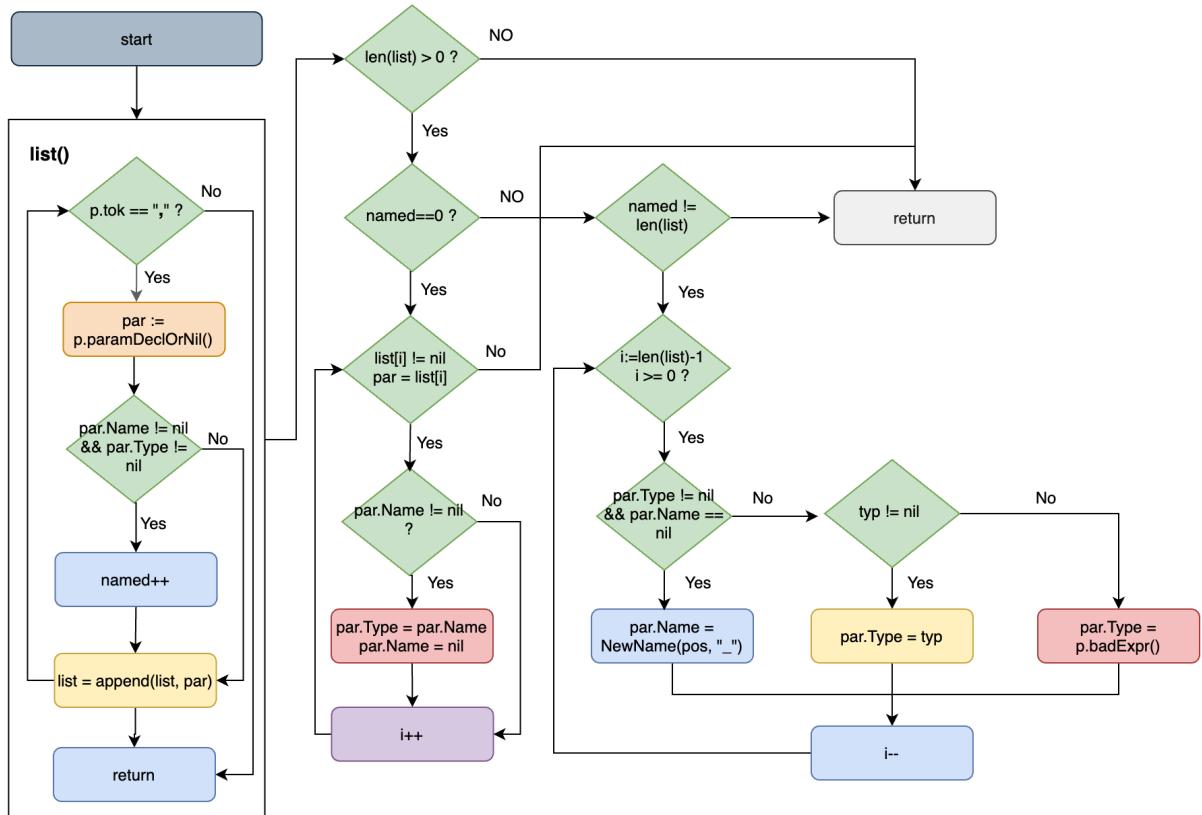
TypeParams:泛型类型参数, 格式:[G, T any]

["] go1.9加入的类型别名 type MyInt2 = int



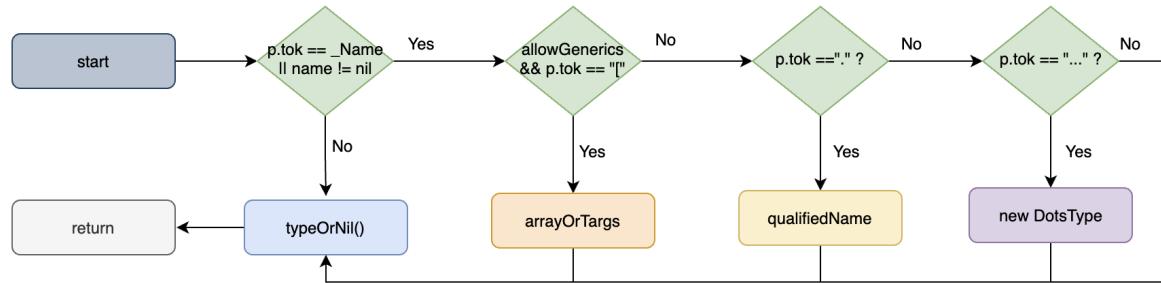
### *paramList*

```
Parameters      = "(" [ ParameterList [ "," ] ] ")" .
ParameterList = ParameterDecl { "," ParameterDecl } .
```

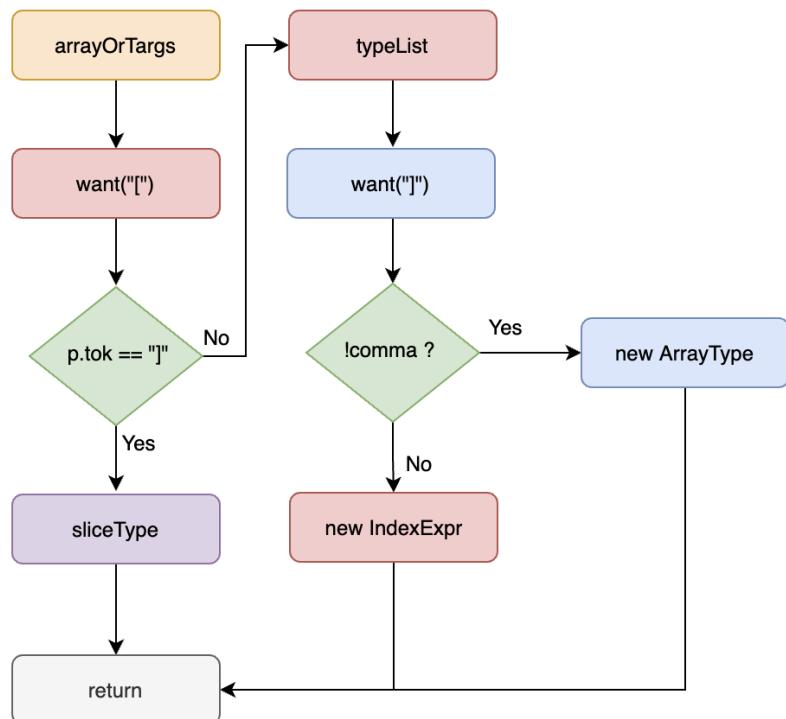


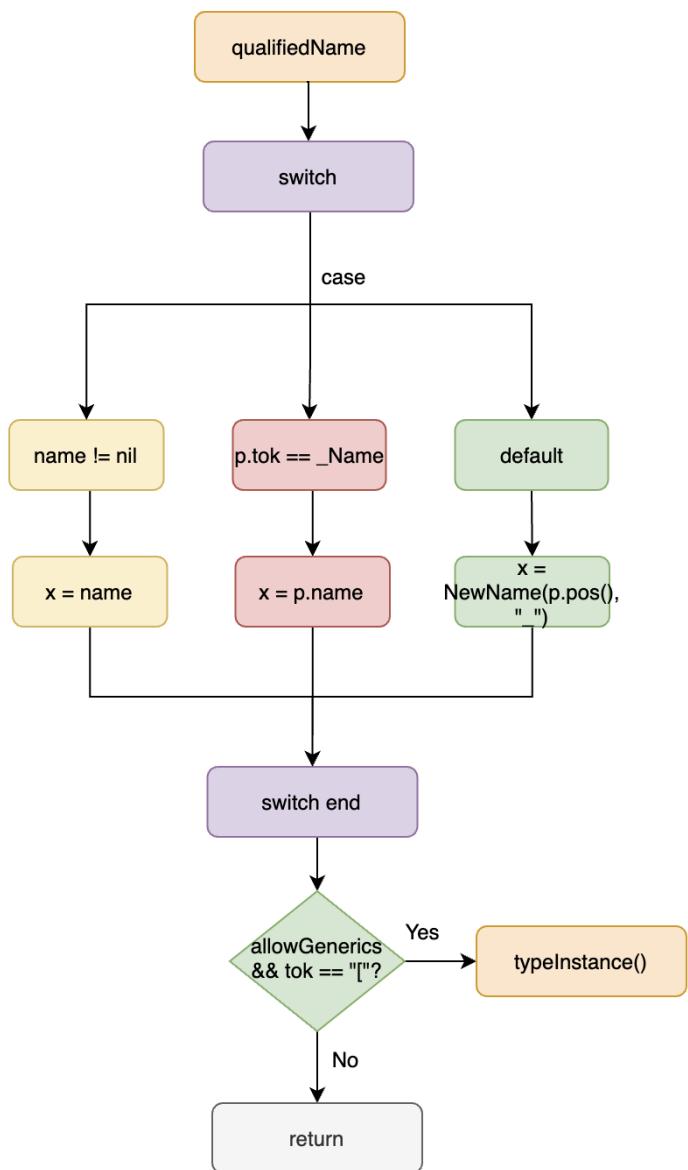
*paramDeclOrNil*

```
ParameterDecl = [ IdentifierList ] [ "..." ] Type .
```



```
type S2[A, B, C any] struct{}
func f10[X, Y, Z any](a S2[ int, bool], b S2[int, bool])
```





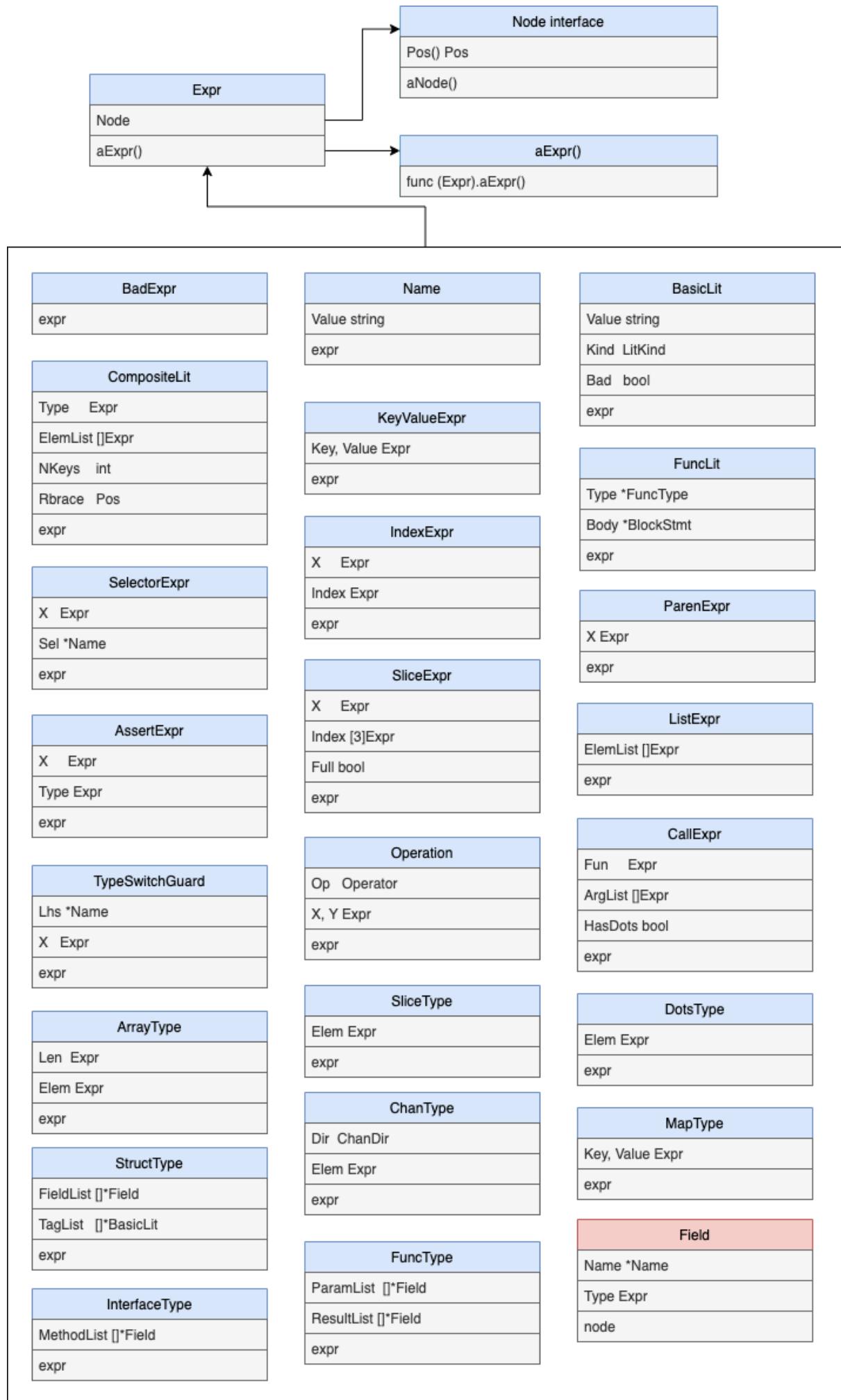
*typeOrNil*

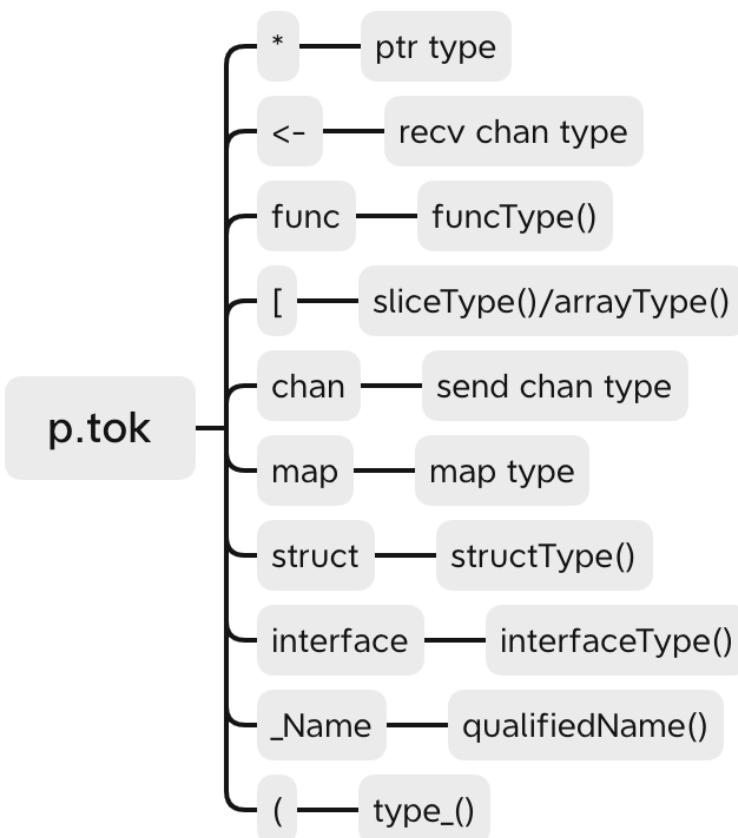
注意type也可以这样写

```
type (
    B int
)
```

```
Type      = TypeName | TypeLit | "(" Type ")"
TypeName = identifier | QualifiedIdent .
TypeLit  = ArrayType | StructType | PointerType | FunctionType |
InterfaceType | SliceType | MapType | Channel_Type .
```

数据结构





### interfaceType

```

InterfaceType = "interface" "{" { ( MethodDecl | EmbeddedElem | TypeList ) ";" } "}" .
TypeList     = "type" Type { "," Type } .

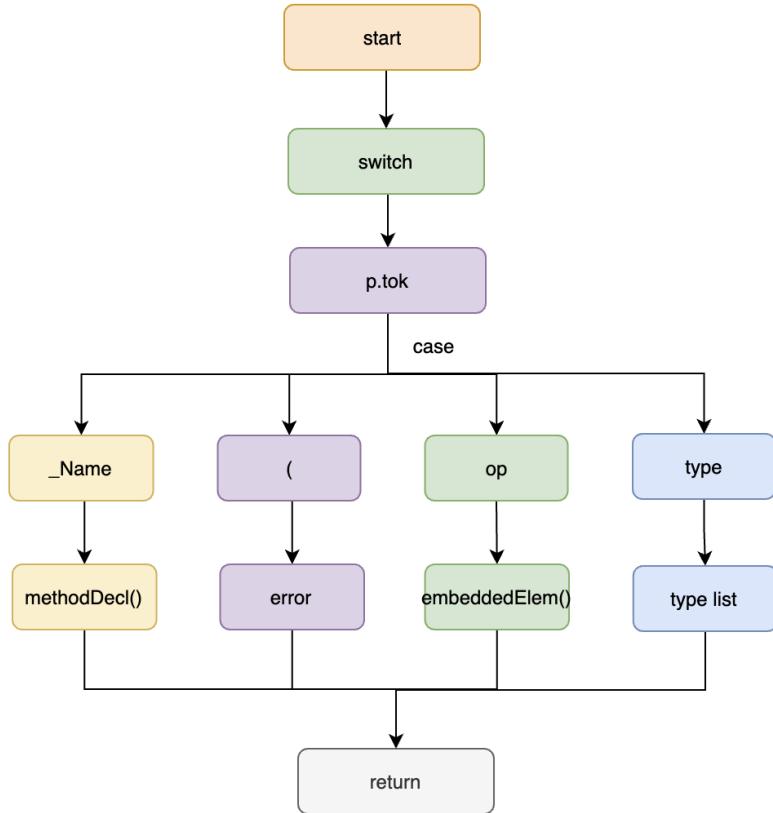
```

```

type SignedInteger interface {
    // 代表只要底层类型都是某个特定类型就可以
    ~int8 | ~int16 | ~int32 | ~int64
}

type SignedInteger1 interface {
    // 约束代表一个type argument可以是int、int8、int16、int32或int64类型
    type string, int8, int16, int32, int64
}

```



*expr*

```

Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr = PrimaryExpr | unary_op UnaryExpr .
PrimaryExpr =
    Operand |
    Conversion |
    PrimaryExpr Selector |
    PrimaryExpr Index |
    PrimaryExpr Slice |
    PrimaryExpr TypeAssertion |
    PrimaryExpr Arguments .

Selector      = ". " identifier .
Index         = "[" Expression "] " .
Slice          = "[" ( [ Expression ] ":" [ Expression ] ) | ( [ Expression ] ":" Expression ":" Expression )
                  "] " .
TypeAssertion = ". " "(" Type ")" .
Arguments     = "(" [ ( ExpressionList | Type [ "," ExpressionList ] )
                  [ "..." ] [ "," ] ] ")" .

```

// 操作符优先级

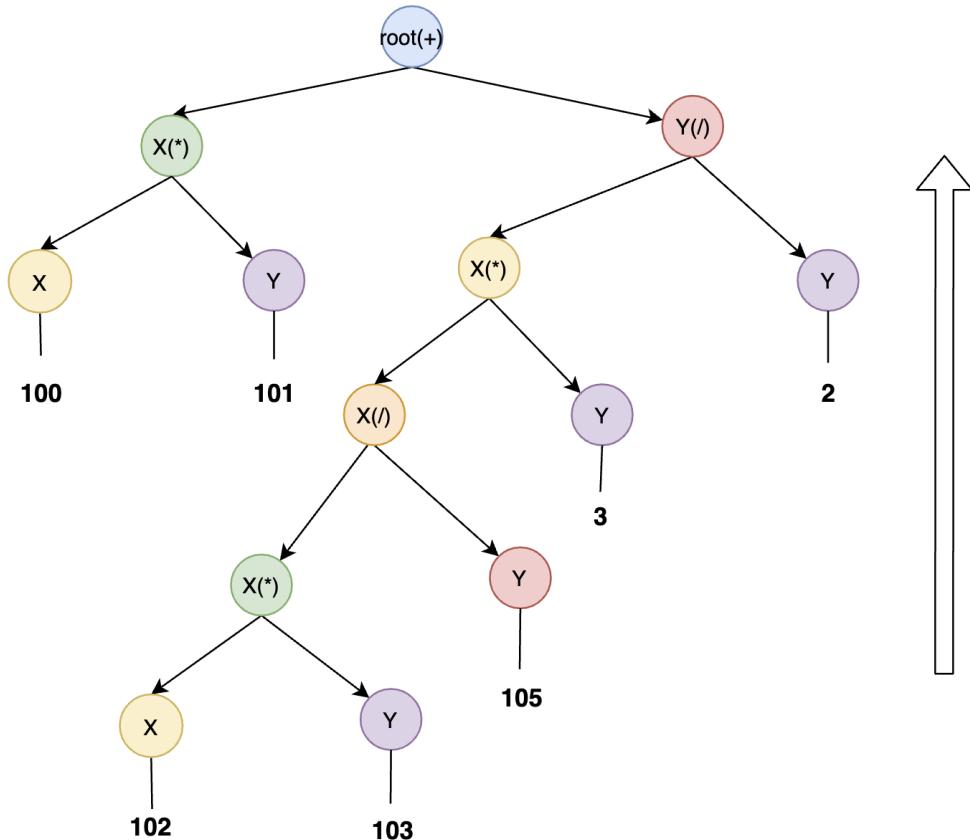
```
const (
    _ = iota
    precOrOr (||)
    precAndAnd (&&)
    precCmp (=)
    precAdd (+)
    precMul (*)
)
```

### binaryExpr

二元操作表达式，这里主要采用自底向上递归算法来实现的。利用操作符的优先级来确定连接的二元表达式。这里的prec是在扫描token的时候处理的。

```
func (p *parser) binaryExpr(prec int) Expr {
    x := p.unaryExpr()
    // 自底向上，进行发散，prec控制优先级，如果prec优先级比之前小，会进行y进行发散
    for (p.tok == _Operator || p.tok == _Star) && p.prec > prec {
        t := new(Operation)
        t.pos = p.pos()
        t.Op = p.op
        tprec := p.prec
        p.next()
        t.X = x // 赋值
        t.Y = p.binaryExpr(tprec)
        x = t // 将上个x值跟和当前x关联起来
    }
    return x
}
```

```
var exp float64 = 100 * 101 + 102 * 103 / 105 * 3 / 2
```



这里介绍另一种方法就是采用栈方式，一般递归都可以转换成栈方式，被称为后缀表达式（又称逆波兰式），主要的思想是将中序表达式转成后缀表达式。

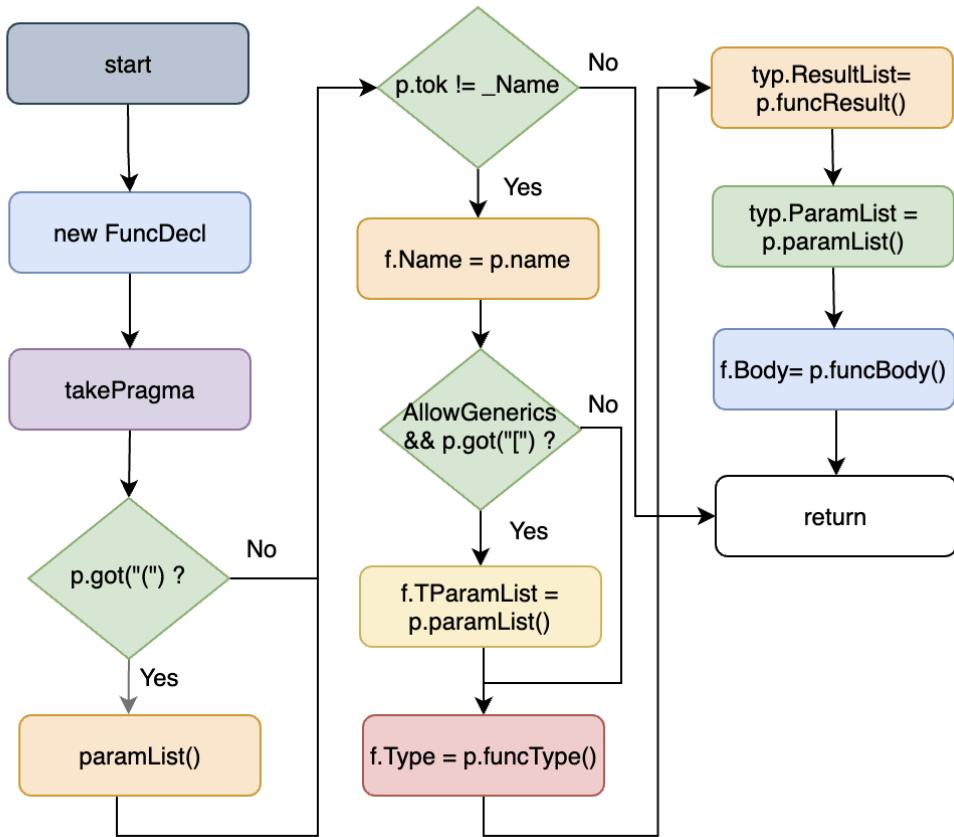
例子：

(1+2)\*(6-7)的后缀表达式就是12+67-\*

首先准备一个栈，从左开始向右遍历逆波兰式的元素。如果取到的元素是操作数，直接入栈，如果是运算符，从栈中弹出2个数进行运算，然后把运算结果入栈当遍历完逆波兰式时，计算结果就保存在栈里了。

*funcDeclOrNil*

```
FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
FunctionName = identifier .
Function     = Signature FunctionBody .
MethodDecl   = "func" Receiver MethodName ( Function | Signature ) .
Receiver     = Parameters .
```



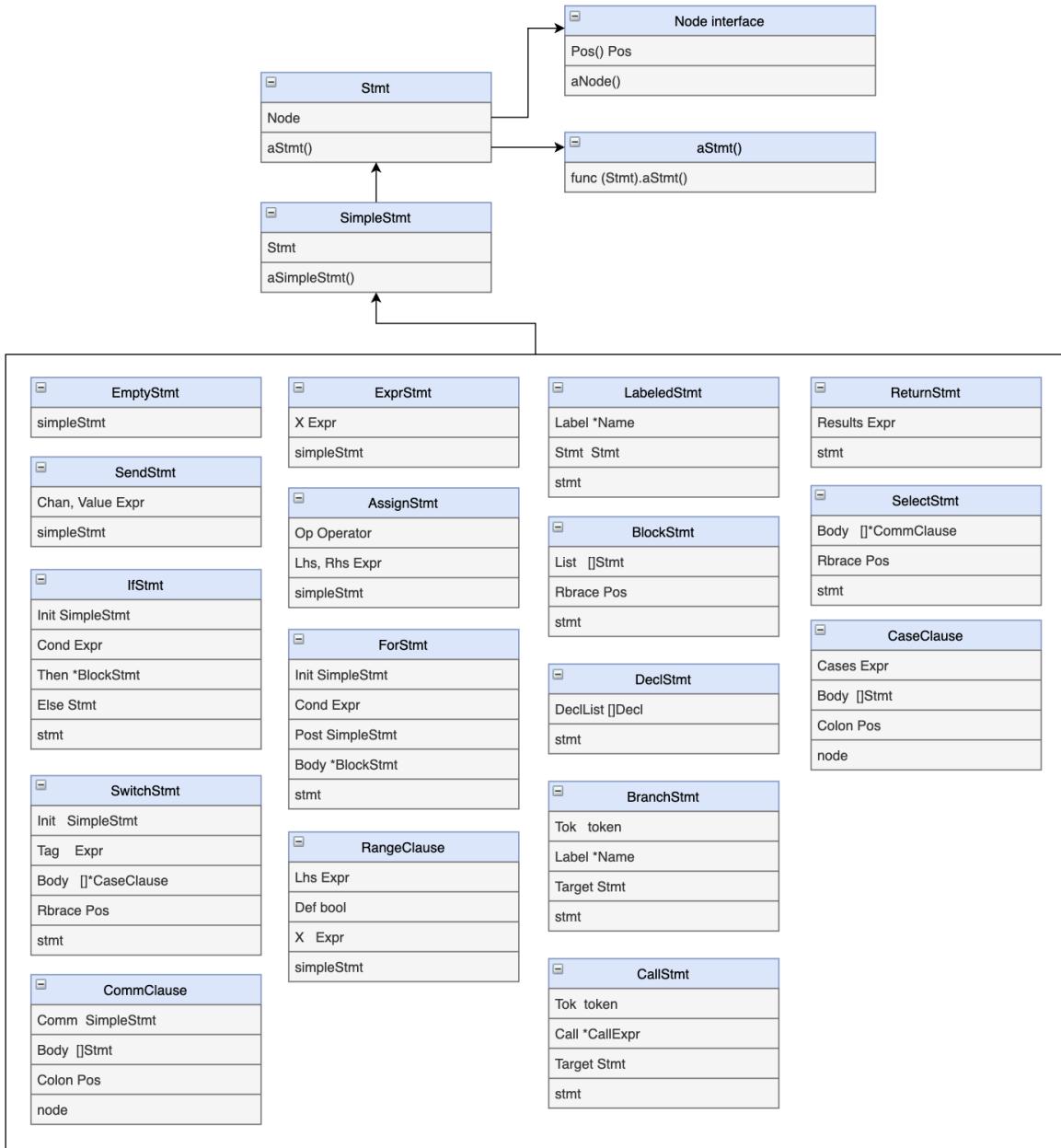
stmt可控制变量的作用域

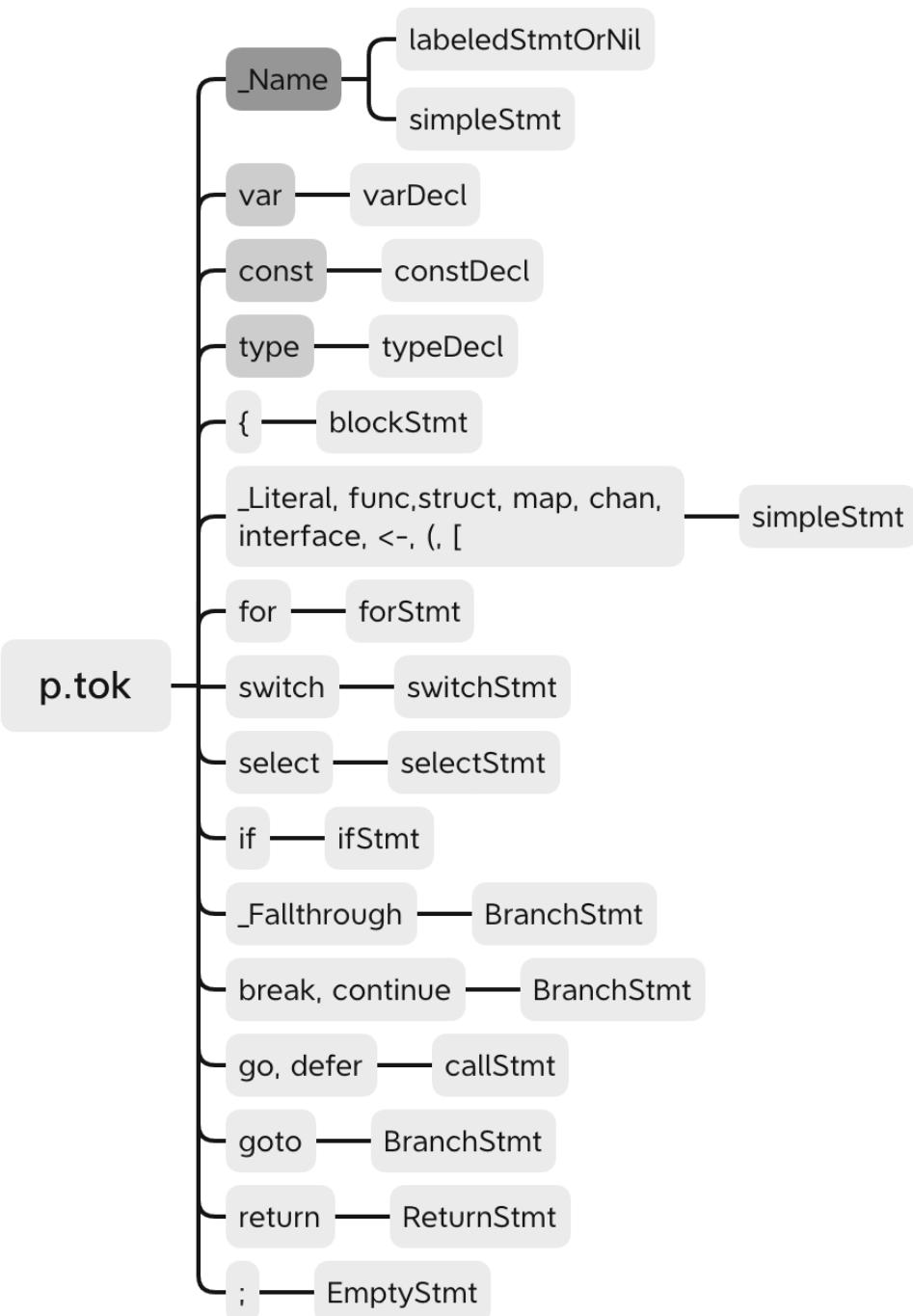
```

StatementList = { Statement ";" } .
Statement =
Declaration | LabeledStmt | SimpleStmt |
GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
DeferStmt .
SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt |
Assignment | ShortVarDecl .

```

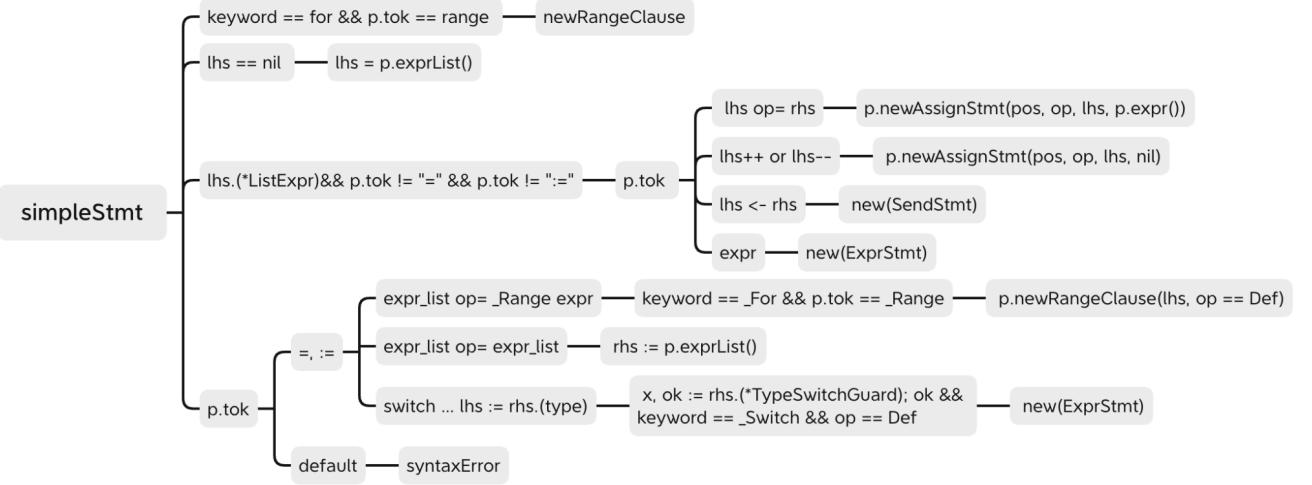
数据结构





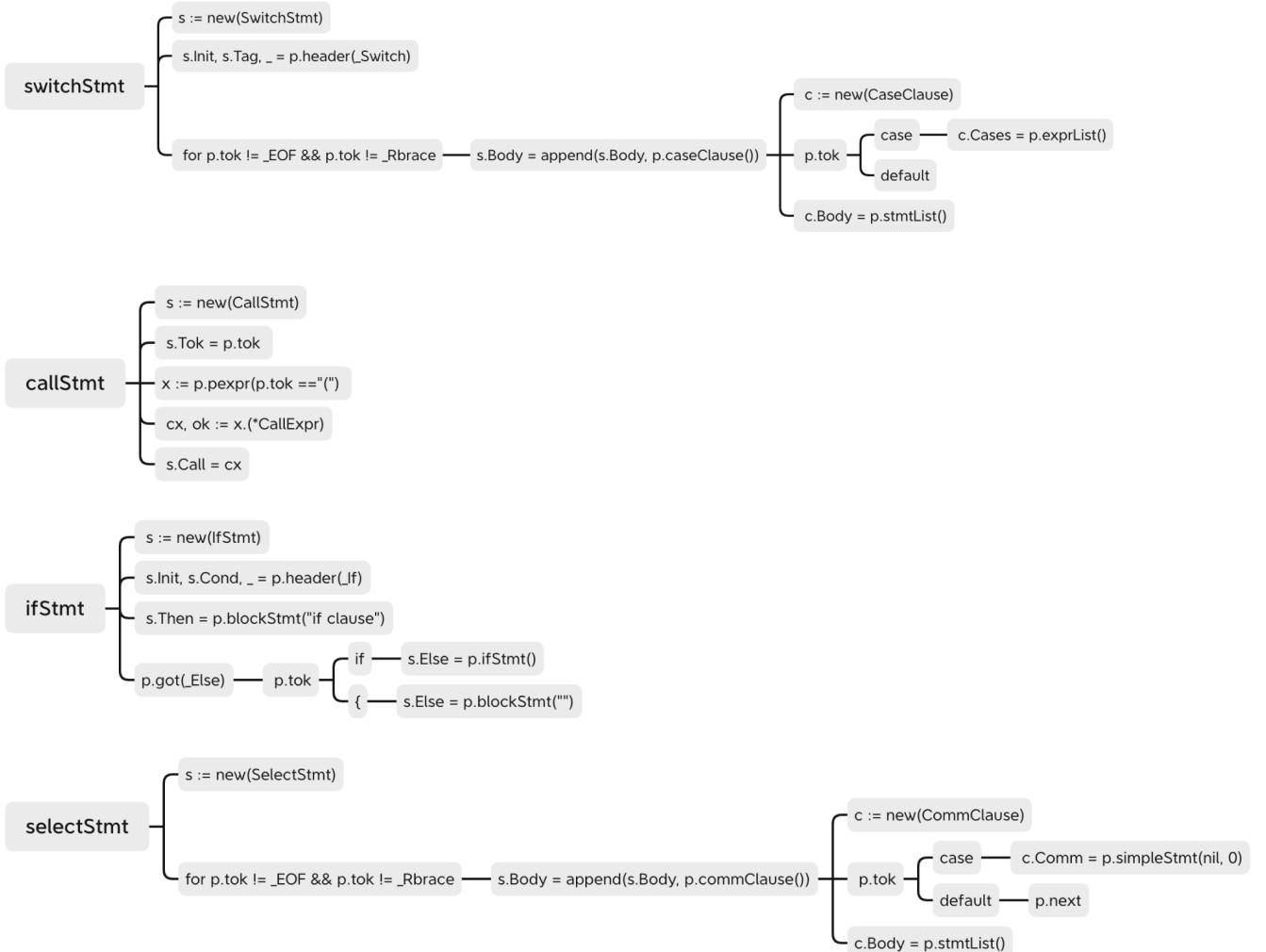
### simpleStmt

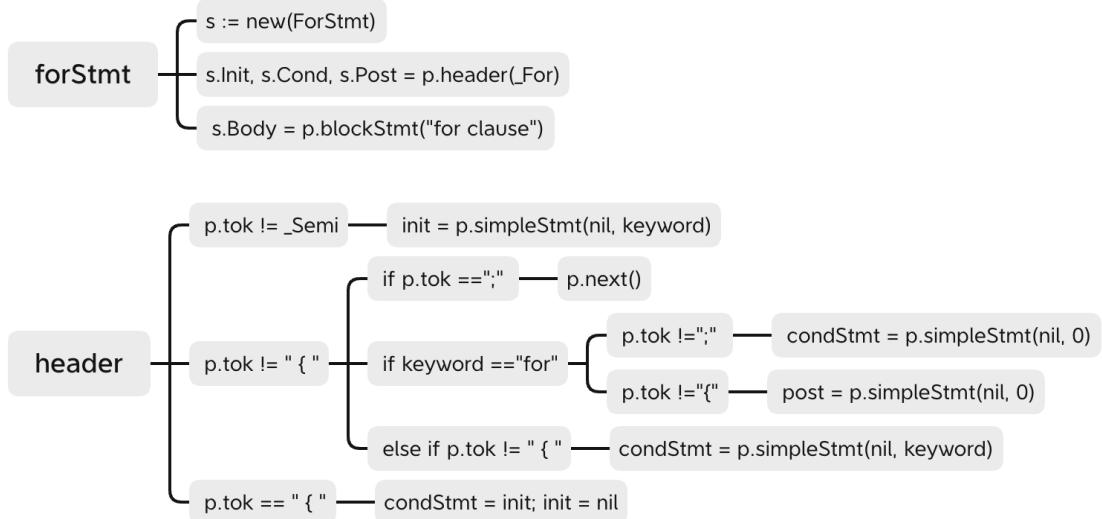
SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl .



## funcStmt

下面图是switchStmt、callStmt、ifStmt、selectStmt、forStmt、header执行流程





## 类型检查

类型术语分为四种：强类型、弱类型、静态类型和动态类型。

强类型在编译期间会有更严格的类型限制，也就是编译器会在编译期发现变量赋值、返回值和函数调用时类型错误，这就是为什么Go会引入泛型。

弱类型出现类型错误时可能会在运行时进行隐式类型转换，在转换时可能会造成错误。

静态类型能够帮助我们在编译期发现程序中类型错误，一般语言会加入静态类型检查。

动态类型是在运行时确认程序的类型，它需要编译时为所有对象加入类型标签等信息，运行时可以使用这些存储类型信息来实现动态派发、向下转型、反射以及其他特性。

Go语言的编译器不仅使用静态类型检查来保证程序运行类型安全，还会在编程期间引入类型相关信息，使用反射来判断参数和变量类型。使用`interface{}`转换成具体类型时会进行动态类型检查，如果无法发生转换就会发生panic。

GO类型检查有两个版本，针对泛型会执行type2目录下的逻辑。

```

if base.Flag.G != 0 {
    // 使用types2进行类型检查，并可能生成IR(中间代码)。
    check2(noders)
    return
}

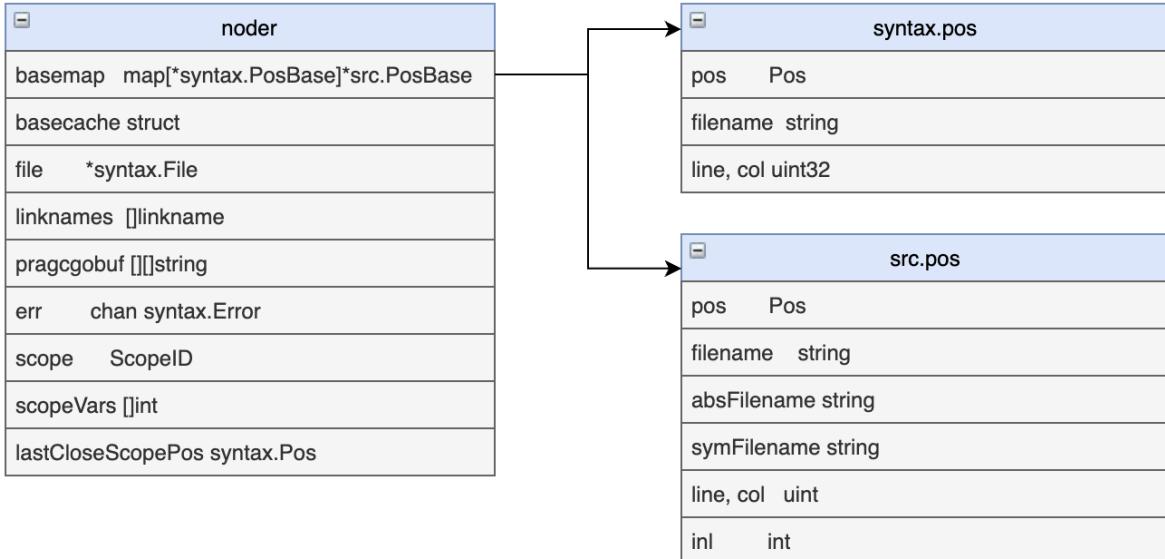
for _, p := range noders {
    p.node()
    p.file = nil // release memory
}

```

## 非泛型

### noder

noder主要生成AST抽象语法树，下面是数据结构：



node函数主要是将当前声明的数据结构转成树结构，也就是node节点，节点数据有操作类型及左右节点等。

```
func (p *noder) node() {
    .....
    // 处理声明列表，将node加到xtop中去
    xtop = append(xtop, p.decls(p.file.DeclList)...)
    .....
}

func (p *noder) decls(decls []syntax.Decl) (l []*Node) {
    var cs constState
    // 循环遍历声明列表，将对应类型进行处理
    for _, decl := range decls {
        p.setlineno(decl)
        switch decl := decl.(type) {
        case *syntax.ImportDecl:
            p.importDecl(decl)

        case *syntax.VarDecl:
            l = append(l, p.varDecl(decl)...)

        case *syntax.ConstDecl:
            l = append(l, p.constDecl(decl, &cs)...)
        }
    }
}
```

```

    case *syntax.TypeDecl:
        l = append(l, p.typeDecl(decl))

    case *syntax.FuncDecl:
        l = append(l, p.funcDecl(decl))

    default:
        panic("unhandled Decl")
    }

}

return
}

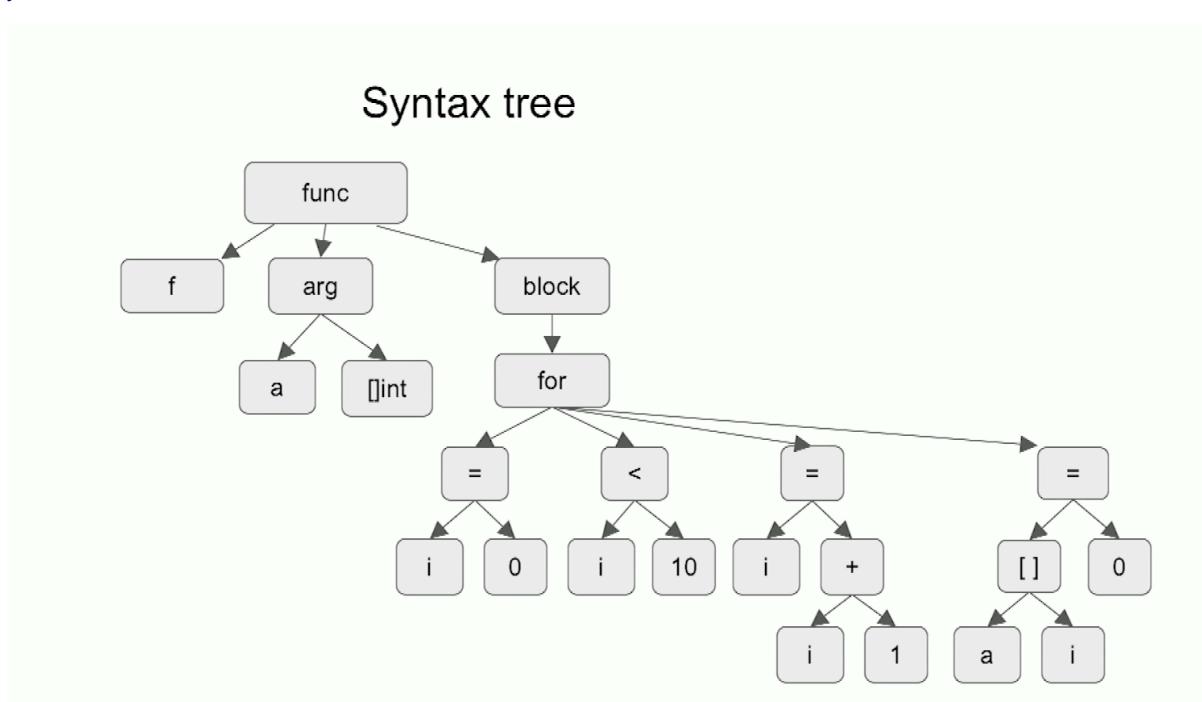
```

举个例子：

```

func f(a []int) {
    for i := 0; i < 10; i++ {
        a[i] = 0
    }
}

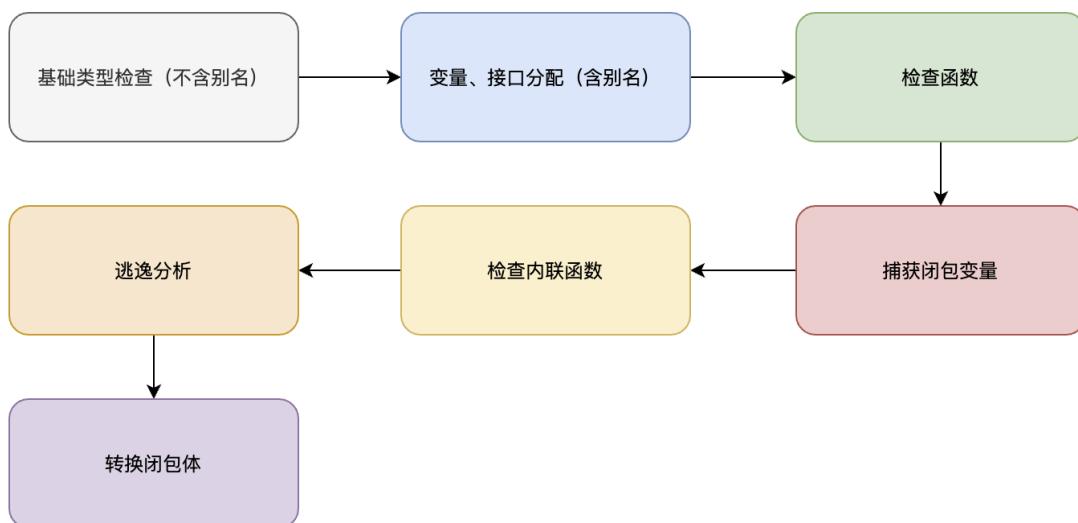
```



执行流程

go分成七个阶段来处理类型检查：

1. 常量、类型以及函数名称和类型检查，收集有关类型的所有信息和方法，但是不包含别名声明。
2. 变量和接口分配，检查别名的声明。
3. 主要是对函数块里数据进行类型检查。
4. 决定如何捕获闭包变量，需要在逃逸分析之前运行，因为值捕获变量不会逃逸。
5. 内联导入实体惰性类型检查。
6. 逃逸分析，需要将堆分配移动到堆栈上，而这又是闭包实现所必需的，它将堆栈变量的地址存储到闭包中。如果闭包没有逃逸，它需要在堆栈上，否则堆栈复制器不会更新它。在逃逸分析中，大值也会移出堆栈；因为大值可能包含指针，所以必须提前发生。
7. 转换闭包体以正确引用捕获的变量。这需要在walk之前进行，因为在walk调用闭包之前必须对闭包进行转换。



```
for i := 0; i < len(xtop); i++ {
    n := xtop[i]
    if op := n.Op; op != ODCL && op != OAS && op != OAS2 && (op != ODCLTYPE || !n.Left.Name.Param.Alias()) {
        xtop[i] = typecheck(n, ctxStmt)
    }
}

timings.Start("fe", "typecheck", "top2")
for i := 0; i < len(xtop); i++ {
    n := xtop[i]
    if op := n.Op; op == ODCL || op == OAS || op == OAS2 || op == ODCLTYPE && n.Left.Name.Param.Alias() {
        xtop[i] = typecheck(n, ctxStmt)
    }
}
```

...

```
initssaconfig()
```

## typecheck

这里介绍一下typecheck1()函数，传入节点Op类型进行不同分支，其中包括加减乘数等操作符、函数、方法调用等 150 多种，节点的种类很多，这里选几个典型案例深入分析。

```
func typecheck1(n *Node, top int) (res *Node) {
    if enableTrace && trace {
        defer tracePrint("typecheck1", n)(&res)
    }

    ok := 0
    switch n.Op {
    default:
        Dump("typecheck", n)
        Fatalf("typecheck %v", n.Op)
    case ONONAME:
        ok |= ctxExpr

    case ONAME:
        ...
    }
    return n
}
```

## OTARRAY

```
case OTARRAY:
    ok |= ctxType
    // 递归处理右节点，也就是切片或数组中元素的类型进行检查
    r := typecheck(n.Right, ctxType)
    if r.Type == nil {
        n.Type = nil
        return n
    }
```

```

var t *types.Type
// 当类型为[]int时, 只需要NewSlice()
if n.Left == nil {
    t = types.NewSlice(r.Type)
} else if n.Left.Op == ODDD { // [...]int类型
    if !n.Diag() {
        n.SetDiag(true)
        yyerror("use of [...] array outside of array literal")
    }
    n.Type = nil
    return n
} else { // [n]int 类型
    n.Left = indexlit(typecheck(n.Left, ctxExpr))
    l := n.Left
    ...
    // 边界检查
    bound := v.U.(*Mpint).Int64()
    if bound < 0 {
        yyerror("array bound must be non-negative")
        n.Type = nil
        return n
    }
    // 初始化数组
    t = types.NewArray(r.Type, bound)
}

```

如果右节点是复合类型, 也就是[...]int类型数组时, 会调用typecheckcomplit()函数进行处理。

```

// Need to handle [...]T arrays specially.

func typecheckcomplit(n *Node) (res *Node) {
    ...
    if n.Right.Op == OTARRAY && n.Right.Left != nil && n.Right.Left.Op
== ODDD {
        n.Right.Right = typecheck(n.Right.Right, ctxType)
        if n.Right.Right.Type == nil {
            n.Type = nil
            return n
        }
        elemType := n.Right.Right.Type
    }
}

```

```

    length := typecheckarraylit(elemType, -1, n.List.Slice(), "array
literal")

    n.Op = OARRAYLIT
    n.Type = types.NewArray(elemType, length)
    n.Right = nil
    return n
}

...
}

```

这三种不同的分支都会更新AST中的内容，数组的长度是类型检查间确定的，`[...]int`这种声明形式是GO为我们提供的语法糖。

## OTMAP

处理hash节点时，编译器会进行key和value类型合法性进行检查。

```

case OTMAP:
    ok |= ctxType
    n.Left = typecheck(n.Left, ctxType)
    n.Right = typecheck(n.Right, ctxType)
    l := n.Left
    r := n.Right
    if l.Type == nil || r.Type == nil {
        n.Type = nil
        return n
    }

    // 创建一个新TMAP结构将hash类型存储到该结构里
    setTypeNode(n, types.NewMap(l.Type, r.Type))
    // 将当前节点加入mapqueue队列，编译器后面会对类型进行再次检查
    mapqueue = append(mapqueue, n)
    n.Left = nil
    n.Right = nil

func NewMap(k, v *Type) *Type {
    t := New(TMAP)
    mt := t.MapType()
    mt.Key = k
}

```

```
    mt.Elem = v
    return t
}
```

## OMAKE

make是go的内置函数，在类型检查阶段之前，无论是创建切片、hash还是channel都是用make关键字，不过在类型检查期间，会被make替换成特定的函数。

make → makechan  
make → makeslice  
make → makemap

```
case OMAKE:
    ok |= ctxExpr
    args := n.List.Slice()

    n.List.Set(nil)
    l := args[0]
    l = typecheck(l, ctxType)
    t := l.Type
    if t == nil {
        n.Type = nil
        return n
    }

    i := 1
    switch t.Etype {
    default:
        yyerror("cannot make type %v", t)
        n.Type = nil
        return n
    }

case TSLICE:
    // 切片长度是否被传入
    if i >= len(args) {
        yyerror("missing len argument to make(%v)", t)
        n.Type = nil
        return n
    }
```

```

    l = args[i]
    i++
    l = typecheck(l, ctxExpr)
    var r *Node
    if i < len(args) {
        r = args[i]
        i++
        r = typecheck(r, ctxExpr)
    }
    ...
    // len 不能大于 cap
    if Isconst(l, CTINT) && r != nil && Isconst(r, CTINT) &&
l.Val().U.(*Mpint).Cmp(r.Val().U.(*Mpint)) > 0 {
        yyerror("len larger than cap in make(%v)", t)
        n.Type = nil
        return n
    }

    n.Left = l
    n.Right = r
    n.Op = OMAKESLICE

    case TMAP:
        if i < len(args) {
            l = args[i]
            i++
            l = typecheck(l, ctxExpr)
            l = defaultlit(l, types.Types[TINT])
            if l.Type == nil {
                n.Type = nil
                return n
            }
            if !checkmake(t, "size", &l) {
                n.Type = nil
                return n
            }
            n.Left = l
        } else {
            n.Left = nodintconst(0)

```

```

        }

        // 将Op改成OMAKEMAP
        n.Op = OMAKEMAP

    case TCHAN:
        l = nil
        // 如果第二个参数不存在，会创建大小为0的Channel
        if i < len(args) {
            l = args[i]
            i++
            l = typecheck(l, ctxExpr)
            l = defaultlit(l, types.Types[TINT])
            if l.Type == nil {
                n.Type = nil
                return n
            }
            if !checkmake(t, "buffer", &l) {
                n.Type = nil
                return n
            }
            n.Left = l
        } else {
            n.Left = nodintconst(0)
        }
        n.Op = OMAKECHAN
    }

    ...
    n.Type = t
}

```

## 泛型

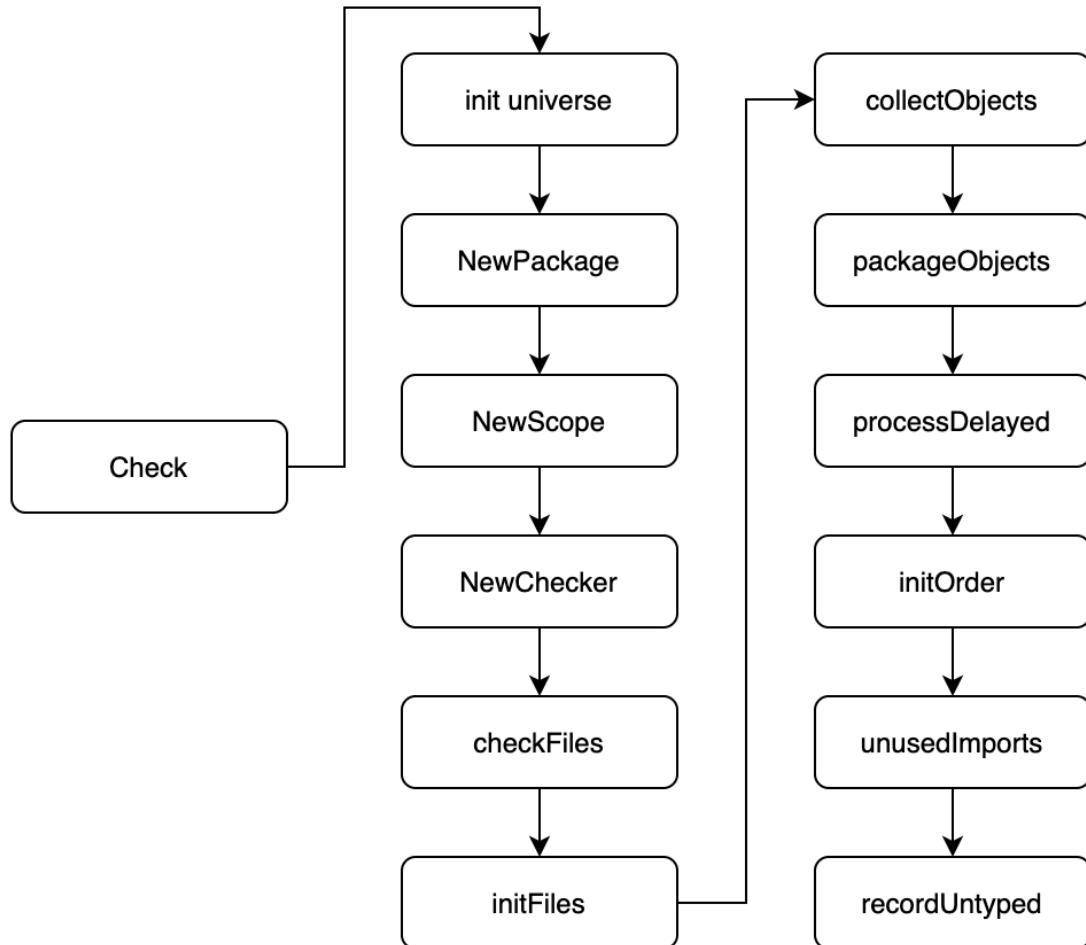
类型主要会涉及到如下几点：

- 类型接口定义，各种类型的结构，比如struct、Interface、Sum等
- 收集对象，比如表达式对应的推断类型等
- 包的加载
- 类型同步及异步检查
- 构建全局变量和全局常量的顺序
- 类型表达式和求值表达式的类型检查

- 类型比较, 比如比较两个结构体是否相等

## 执行流程

经过了词语和语法解析过, 就会调用类型检查服务。这里需要注意全局作用域是通过init()来执行的, 主要初始化一些系统函数和类型。delayed作用是有些函数需要延迟检查才行。



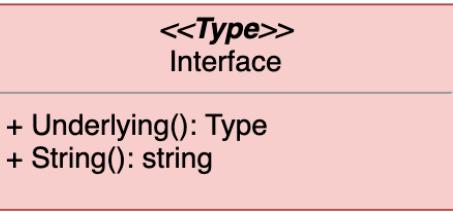
## 类型定义

主要针对go类型定义的结构和操作方法, 为了方便类型检查而设定的。

## 数据结构

## 接口定义

所有类型都需要实现此接口, 这里Underlying()主要是返回实际类型, 只针对Named生效。



## 基础类型

主要定义go基础的类型，这里需要注意的是常量的定义，常用定义如果没有指定类型的话，都会归属于无类型的，这里看一个untypedBool类型的例子：

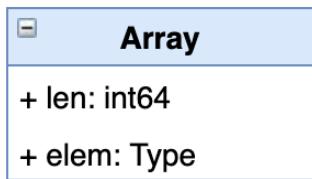
```
package gc
const _ = false
```

isOrdered其实需要类型支持比较的类型才行，所以只有整型、浮点型及字符串。

Basic	<<BasicKind>>	<<BasicInfo>>
+ kind: BasicKind + info: BasicInfo + name: string	<b>&lt;&lt;BasicKind&gt;&gt;</b> int  + Invalid BasicKind = iota + Bool + Int + Int8 + Int16 + Int32 + Int64 + Uint + Uint8 + Uint16 + Uint32 + Uint64 + uintptr + Float32 + Float64 + Complex64 + Complex128 + String  + UnsafePointer + UntypedBool + UntypedInt + UntypedRune + UntypedFloat + UntypedComplex + UntypedString + UntypedNil  + Byte = Uint8 + Rune = Int32	<b>&lt;&lt;BasicInfo&gt;&gt;</b> int  + IsBoolean BasicInfo = 1 << iota + IsInteger + IsUnsigned + IsFloat + IsComplex + IsString + IsUntyped  + IsOrdered = IsInteger   IsFloat   IsString + IsNumeric = IsInteger   IsFloat   IsComplex + IsConstType = IsBoolean   IsNumeric   IsString

## 数组和切片

数组和切片唯一的区别就是是否是定长，elem元素的类型。



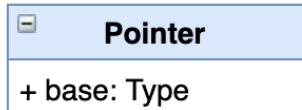
## 结构体

fields存储结构体的元素, tags则是一些tag定义, 比如我们常用的json定义等。



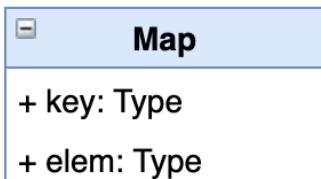
## 指针

指针类型可以是任意的类型, 这里定义成Type。



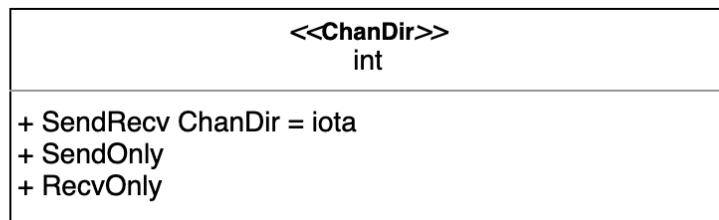
## map

map的key和elem可以是任意类型



## chan

chan里的dir只有三种情况, 发送接收、发送、接收。



type

主要针对已定义的类型，这里的tparams指的是类型约束，targs是类型约束实例化的参数。info字段主要在后期用于类型的循环依赖检查。

Name
+ check: *Checker
+ info: typeInfo
+ obj: *TypeName
+ orig: *Named
+ fromRHS: Type
+ underlying: Type
+ tparams: []*TypeName
+ targs: []Type
+ methods: []*Func

## 函数

函数签名里的主要定义了函数的参数、返回值、是否变长的、作用域、接收方法等。

这里主要介绍一下rparams和tparams参数，下面实例分别介绍了参数的含义：

method = func (T1[A<sub>35</sub>]) m1[C<sub>36</sub> interface{}]() A<sub>35</sub>

rparams = [type A = A<sub>35</sub>]，接收方法的约束类型

tparams = [type C = C<sub>36</sub>]，函数的约束类型

-	<b>Tuple</b>
+ vars:	[]*Var

-	<b>Signature</b>
+ rparams:	[]*TypeName
+ tparams:	[]*TypeName
+ scope:	*Scope
+ recv:	*Var
+ params:	*Tuple
+ results:	*Tuple
+ variadic:	bool

interface

sum定义的一组类型约束集合, embeddeds嵌入的类型, allXXX所有嵌入的集合。

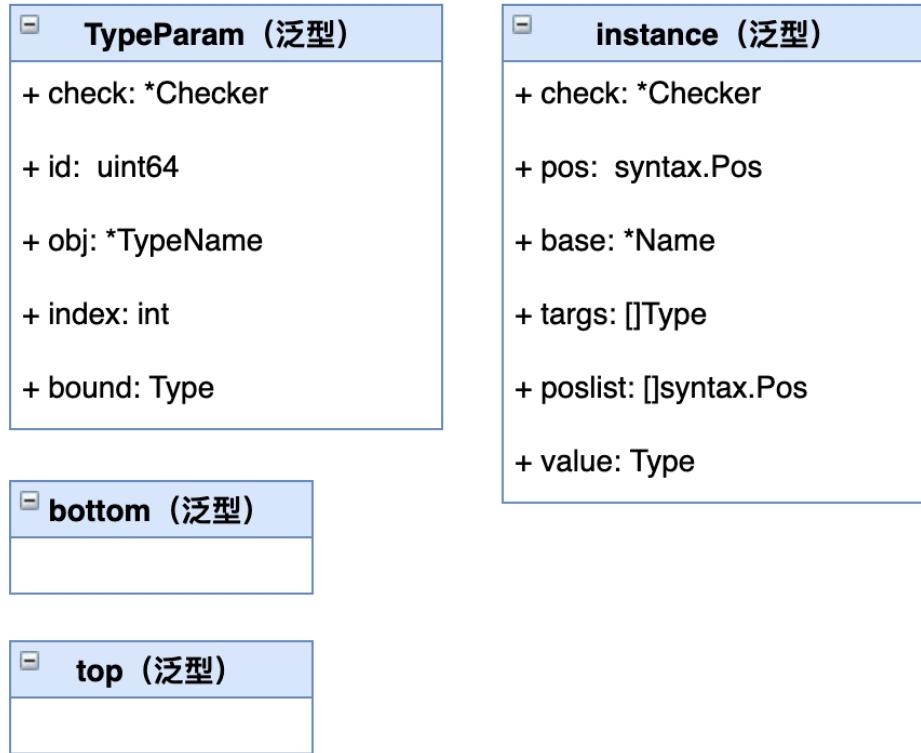
-	<b>Sum</b>
+ types:	[]Type

-	<b>Interface</b>
+ methods:	[]*Func
+ types:	Type (Sum)
+ embeddeds:	[]Type
+ allMethods:	[]*Func
+ allTypes:	Type (Sum)
+ obj:	Object

泛型

id通过nextId返回一个以1单调递增的值, bound指的是类型的边界也就是类型约束的集合, 底层类型还是interface。这里需要注意的是bottom和top, 主要是类型标识, 用于接口类型约束时

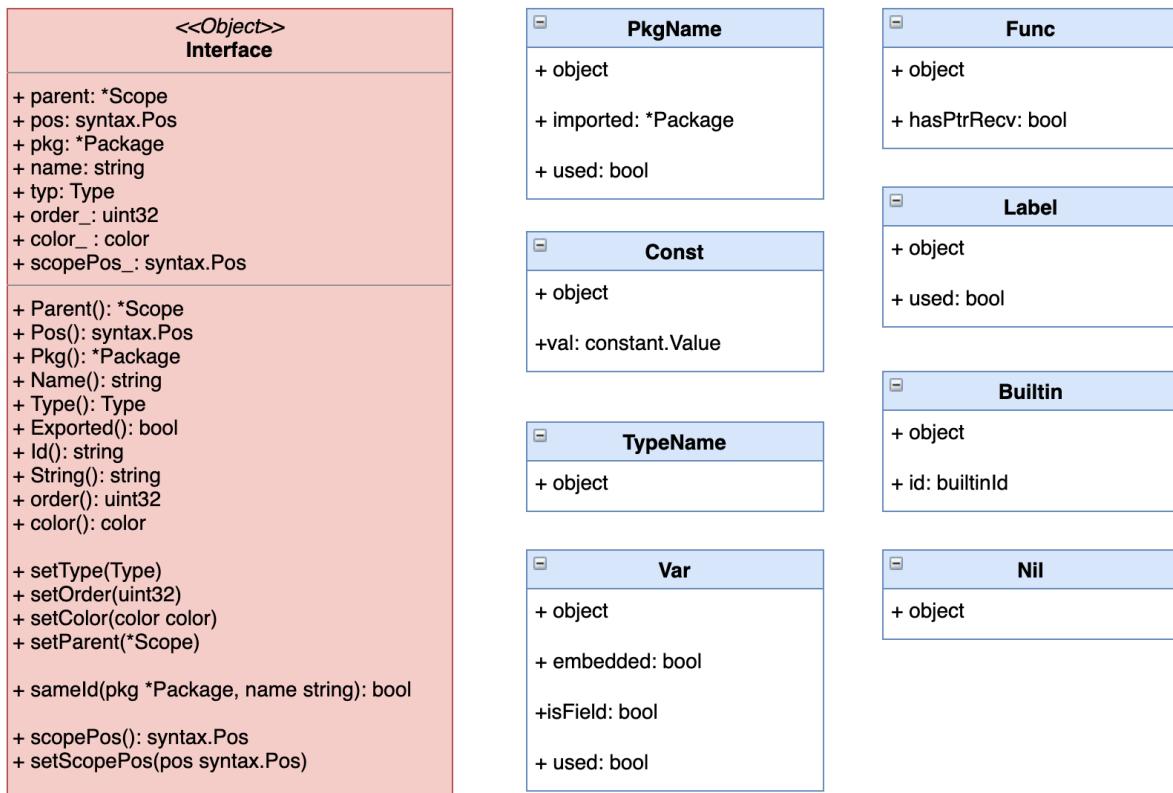
, 嵌套接口里约束的合集, 只有两种情况, 要么是空集, 要么是全集。bottom代表空集, top代表全集。



## 对象定义

一个对象描述了一个命名的语言实体, 例如一个包、常量、类型、变量、函数(包括方法)及标签。所有的对象都包含object嵌入了object对象, object对象实现了一些通用方法, 比如设置类型、排序等。注意我这里把object写入到Object里interface里去了, object是结构体而Object是interface。object结构体实现了Object的所有方法。

## 数据结构



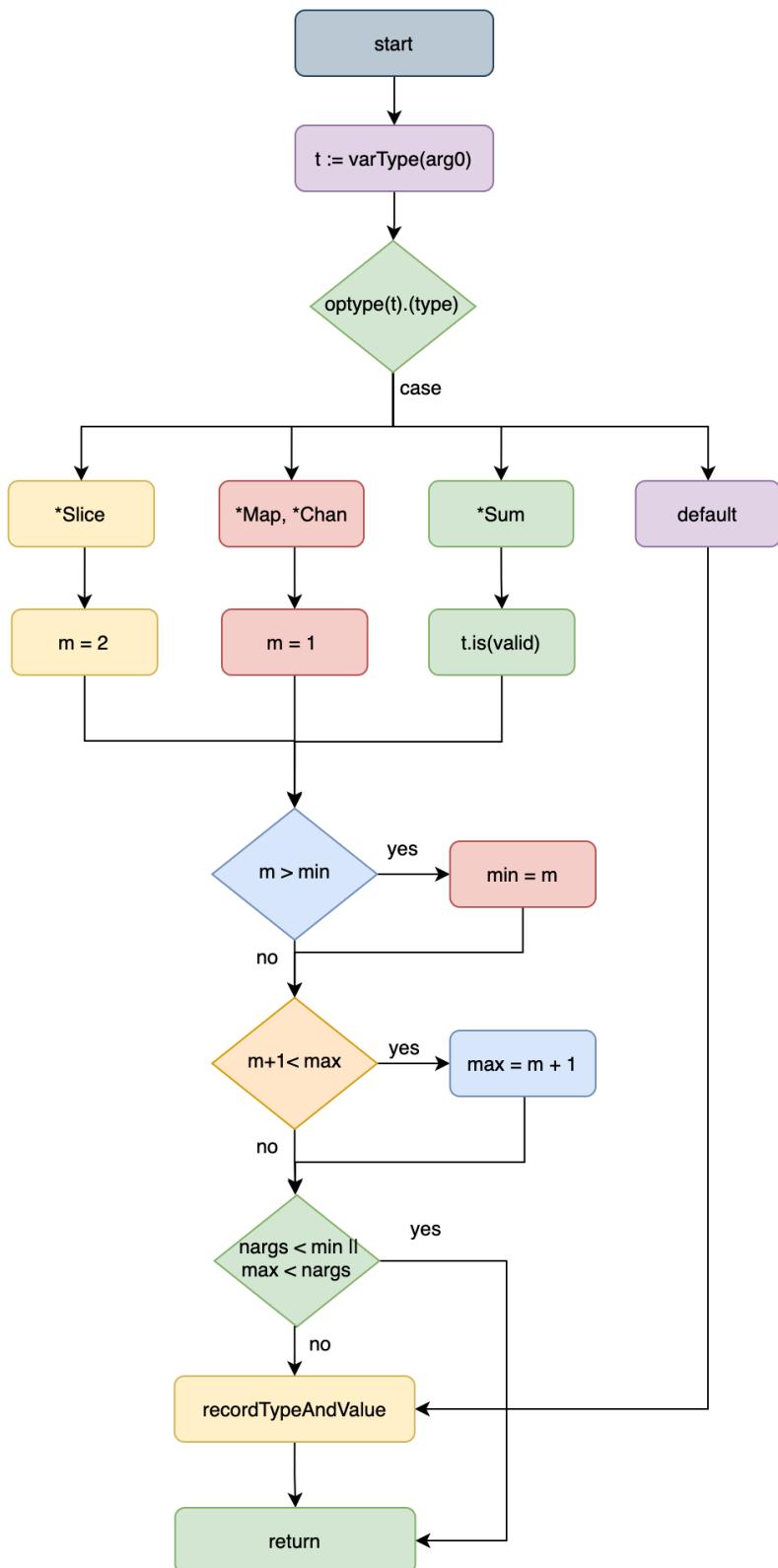
## 内部函数

`builtin` 函数主要创建 go 系统函数的，比如常用 `append`、`cap`、`copy`、`delete` 及 `len` 等。

这里介绍一下 `make` 函数的创建，其它函数区别也不大，这里就不一一介绍了。

```
make
make(T, n)
make(T, n, m)
```

需要检查一下类型是否满足 `slice`、`map` 及 `chan`，这里类型 `Sum(type int | string)` 泛型定义。  
`recordTypeAndValue` 会记录类型和值，也就是函数名和函数的签名 (`m[x] = TypeAndValue{mode, typ, nil}`)。



## Universe

有些系统函数、类型及常量等，需要提前进行初始化并将其注册到全局作用域里。比如我们使用泛型时comparable和any都是在这里提前初始化的。

```
func init() {
    Universe = NewScope(nil, nos, nos, "universe")
    Unsafe = NewPackage("unsafe", "unsafe")
    Unsafe.complete = true

    defPredeclaredTypes()
    defPredeclaredConsts()
    defPredeclaredNil()
    defPredeclaredFuncs()
    defPredeclaredComparable()

    .....

    // "any" 仅在类型参数列表中作为约束可见
    delete(Universe.elems, "any")
}
```

这里介绍一下创建go内置函数，函数定义如下：

```
var predeclaredFuncs = [...]struct {
    name      string
    nargs     int
    variadic  bool
    kind      exprKind
}{

    _Append:  {"append", 1, true, expression},
    _Cap:     {"cap", 1, false, expression},
    _Close:   {"close", 1, false, statement},
    _Complex: {"complex", 2, false, expression},
    _Copy:    {"copy", 2, false, statement},
    _Delete:  {"delete", 2, false, statement},
    _Imag:    {"imag", 1, false, expression},
    .....

    _Assert: {"assert", 1, false, statement},
    _Trace:  {"trace", 0, true, statement},
}

func defPredeclaredFuncs() {
    for i := range predeclaredFuncs {
        // 获取对应的builtinID
        id := builtinId(i)
```

```

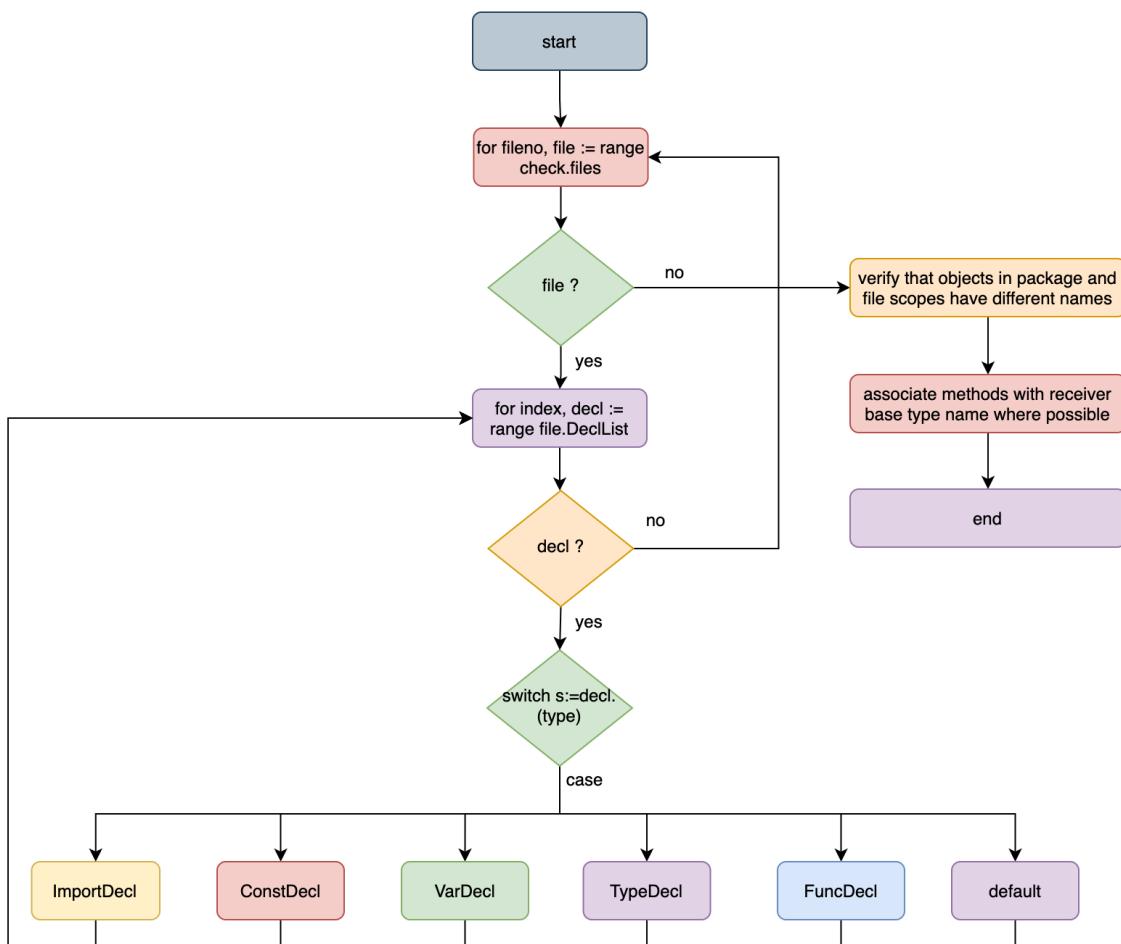
.....
// new builtin类型object
def(newBuiltIn(id))
}
}

```

将循环所有的内置函数，取得对应的builtinId，创建builtin类型的object，然后将对象插入全局作用域里（scope.Insert(obj)）。

## 对象收集

有了对象类型和全局作用域，就可以收集对象了。收集对象主要是针对import、const、var 及 func收集，func body里的内容不会立即收集，主要放到延迟对列进行处理，因为只有顶级对象确认了，才能进行类型检查。这里只介绍一下import对象收集，其它的对象类似，对象收集完还需要验证包和文件范围内的对象是否具有不同的名称，以及方法和接收者进行关联。



## ImportDecl

import包的时候导入函数只是定义了一个接口形式，两个接口分别ImportFromImport，接口的区别就是可以指定目录的方式来支持vendor机制。go是以包的纬度进行编译的，每个包都会编译成一个xxx.a的文件。

```
> fset: *go/token.FileSet {mutex: sync.RWMutex {w: (*sync.Mu
> packages: map[string]*go/types.Package [{"fmt": *{path: "fi
path: "go/parser"

srcDir: "."
lookup: nil
pkg: *go/types.Package nil
err: error nil
rc: io.ReadCloser nil
id: "go/parser"
filename: "/usr/local/go/pkg/darwin_amd64/go/parser.a"
```

所有编译完的包都会存于此结构体里 packages 里。

```
type gcimports struct {
    fset      *token.FileSet
    packages map[string]*types.Package
    lookup   Lookup
}

// import package

path, err := validatedImportPath(s.Path.Value)
.....
imp := check.importPackage(s.Path.Pos(), path, fileDir)
.....
//本地名称覆盖导入的包名称
name := imp.name
if s.LocalPkgName != nil {
    name = s.LocalPkgName.Value
    if path == "C" {
        check.error(s.LocalPkgName, `cannot rename import "C"`)
        continue
    }
}
.....
```

```

pkgName := NewPkgName(s.Pos(), pkg, name, imp)
if s.LocalPkgName != nil {
    //在点导入中，点代表包
    check.recordDef(s.LocalPkgName, pkgName)
} else {
    check.recordImplicit(s, pkgName)
}

//将导入包添加到文件范围里
check.imports = append(check.imports, pkgName)
if name == "." {

    if check.dotImportMap == nil {
        check.dotImportMap = make(map[dotImportKey]*PkgName)
    }

    //将导入的范围与文件范围合并
    for _, obj := range imp.scope.elems {

        if obj.Exported() {

            if alt := fileScope.Insert(obj); alt != nil {

                .....
            }
        }
        .....
    }
} else {
    //在文件范围内声明导入的包对象
    check.declare(fileScope, nil, pkgName, nopos)
}

```

## 对象类型状态

检查obj的声明意味着推断其类型(可能还有常量的值)。对象的类型可能处于三种状态之一，这三种状态由颜色表示：

- 1、类型未知的对象被涂成白色(初始颜色)
- 2、正在推断其类型的对象被涂成灰色
- 3、类型完全推断的对象被涂成黑色

在类型推断期间，对象的颜色从白色变为灰色再变为黑色(预先声明的对象从一开始就被绘制为黑色)。黑色对象(即其类型)只能依赖于(参考)其他黑色对象。白色和灰色对象可能依赖于白色和黑色对象。对灰色对象的依赖性表示一个循环，该循环可能有效，也可能无效。

当对象变成灰色时，它们被推到对象路径(堆栈)上；当它们变黑时，会再次弹出。因此，如果遇到灰色对象(循环)，则它位于对象路径上，并且它所依赖的所有对象都是该路径上的剩余对象。颜色编码使得灰色对象的颜色值指示该对象在对象路径中的索引。

```
//类型已确定，设置成黑色
if obj.color() == white && obj.Type() != nil {
    obj.setColor(black)
    return
}

switch obj.color() {
    case white:
        assert(obj.Type() == nil)
        //除白色和黑色以外的所有颜色值均视为灰色。
        //因为黑色和白色是<灰色的，所以所有>=灰色的值都是灰色的。
        //使用这些值将对象的索引编码到对象路径中。
        obj.setColor(grey + color(check.push(obj)))
        defer func() {
            check.pop().setColor(black)
        }()
}

case black:
    assert(obj.Type() != nil)
    return

default:
    //除白色或黑色以外的颜色值被视为灰色。
    fallthrough

case grey:
    //循环事件处理
    switch obj := obj.(type) {
        case *Const:
            if check.cycle(obj) || obj.typ == nil {
                obj.typ = Typ[Invalid]
            }
        case *Var:
            if check.cycle(obj) || obj.typ == nil {
                obj.typ = Typ[Invalid]
            }
        .....
    default:
        unreachable()
    }
```

```

    assert(obj.Type() != nil)
    return
}

```

循环对象处理条件有两种：

- 1、只涉及常量和变量的循环是无效的
- 2、仅涉及类型(可能还有函数)的循环必须至少具有一个类型定义

```

func (check *Checker) cycle(obj Object) (isCycle bool) {
    //计数循环对象
    assert(obj.color() >= grey)
    start := obj.color() - grey // objPath中obj的索引
    cycle := check.objPath[start:]
    //循环中的(常量或变量)值数
    nval := 0
    //循环中的类型定义数
    ndef := 0
    for _, obj := range cycle {
        switch obj := obj.(type) {
        case *Const, *Var:
            nval++
        case *TypeName:
            var alias bool
            if d := check.objMap[obj]; d != nil {
                //包级对象
                alias = d.tdecl.Aliast
            } else {
                //函数局部对象
                alias = obj.IsAlias()
            }
            if !alias {
                ndef++
            }
        case *Func:

        default:
            unreachable()
        }
    }

    // 只涉及常量和变量的循环是无效的
    if nval == len(cycle) {
        return false
    }

    // 仅涉及类型(可能还有函数)的循环必须至少具有一个类型定义
    // 如果没有类型定义, 我们具有将无限扩展的别名类型名称序列。
    if nval == 0 && ndef > 0 {
        //循环是允许的
        return false // cycle is permitted
    }
}

```

```

    }

    check.cycleError(cycle)

    return true
}

```

## 类型循环引用

这种类型的递归会导致程序的死循环，所以需要提前抛出错误。如果类型是指针的话，就不会有问题的，因为指针的空间是固定的，下面的switch里没有指针类型的case，但是这种语法在go里也没有什么作用。

```

func (check *Checker) typeDecl(obj *TypeName, tdecl *syntax.TypeDecl, def
*Named) {
    assert(obj.typ == nil)

    check.later(func() {
        // 检查循环引用
        check.validType(obj.typ, nil)
    })
    .....
}

func (check *Checker) validType(typ Type, path []Object) TypeInfo {
    switch t := typ.(type) {
    case *Array:
        return check.validType(t.elem, path)

    case *Struct:

    case *Interface:

    case *Named:

    case *instance:

    }
    return valid
}

```

下面是一些循环引用的例子：

```

type (
    T0 int
    T1 /* ERROR cycle */ T1
    T2 *T2
)

```

```

A [10]A

// structs
S0 /* ERROR cycle */ struct{ _ S0 }
S1 /* ERROR cycle */ struct{ S1 }
S2 struct{ _ *S2 }
S3 struct{ *S3 }
)

```

## 作用域

一个标识符的作用域是程序中的一段区域，用于确定该标识符的可见性。当标识符在一段区域中可见时，就可以在该区域内使用此标识符，作用域大概分为如下几种下：

名称
全局作用域(universe)
包作用域
文件作用域
label
函数
块(block)
if
switch
case
for

全局作用域，主要是提前预加载一些系统函数和类型，比如bool,int,new,copy iota等。

openScope

数据结构

```

type Scope struct {
    parent    *Scope
    children []*Scope
    elems     map[string]Object //懒分配
    pos, end syntax.Pos
    comment   string
    isFunc    bool
}

```

自底向上的链表也称为父链, children存取当前作用域的子作用域。elems存放着对象, pos,end存放作用域的范围。

初始化作用域

```

func NewScope(parent *Scope, pos, end syntax.Pos, comment string) *Scope
{
    s := &Scope{parent, nil, nil, pos, end, comment, false}

    //不要将子级添加到 Universe 范围
    if parent != nil && parent != Universe {
        parent.children = append(parent.children, s)
    }
    return s
}

```

根据标识符查询scope

```

func (s *Scope) LookupParent(name string, pos syntax.Pos) (*Scope,
Object) {
    for ; s != nil; s = s.parent {
        if obj := s.elems[name]; obj != nil && (!pos.IsKnown() ||
obj.scopePos().Cmp(pos) <= 0) {
            return s, obj
        }
    }
    return nil, nil
}

```

这里举个例子:

```

func xxx() {
    { var blockVar int64; }
}

```

这里是block作用域, go里面实现还是很简单, 先创建一个scope, 然后处理stmt list, 最后执行defer里的closeScope(), 也就是将scope指向上级作用域。

```
case *syntax.BlockStmt:  
    check.openScope(s, "block")  
    defer check.closeScope()  
    check.stmtList(inner, s.List)  
  
func (check *Checker) closeScope() {  
    check.scope = check.scope.Parent()  
}
```

```
✓ scope: *(*cmd/compile/internal/types2.Scope)(0xc0000079e00)
  ✓ : cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope}(0xc0000079c20),
    > parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope}(0xc0000079c20),
      ✓ children: []*cmd/compile/internal/types2.Scope len: 1, cap: 1, []*cmd/compile/internal/types2.Scope(0xc0000079ec0)
        ✓ [0]: *(*cmd/compile/internal/types2.Scope)(0xc0000079ec0)
          ✓ : cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope}(0xc0000079c80),
            ✓ parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope}(0xc0000079c80),
              ✓ : cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope {parent: *cmd/compile/internal/types2.Scope}(0xc0000079c80),
                > parent: *(*cmd/compile/internal/types2.Scope)(0xc0000079c80)
                > children: []*cmd/compile/internal/types2.Scope len: 1, cap: 1, []*cmd/compile/internal/types2.Scope(0xc0000079c80)
                  elems: map[string]cmd/compile/internal/types2.Object nil
                > pos: cmd/compile/internal/syntax.Pos {base: *cmd/compile/internal/syntax.PosBase {pos: (*cmd/compile/internal/syntax.PosBase)(0xc0000010ae0), line: 1, col: 32}, comment: "function", isFunc: true}
                  children: []*cmd/compile/internal/types2.Scope len: 0, cap: 0, nil
                ✓ elems: map[string]cmd/compile/internal/types2.Object ["blockVar": ...]
                  len(): 1
                  > "blockVar": cmd/compile/internal/types2.Object(*cmd/compile/internal/types2.Var) *{object: (*cmd/compile/internal/types2.Var)(0xc0000010ae0), line: 1, col: 32}
                > pos: cmd/compile/internal/syntax.Pos {base: *cmd/compile/internal/syntax.PosBase}(0xc0000010ae0), line: 1, col: 32
                  > base: *cmd/compile/internal/syntax.PosBase(0xc0000010ae0)
                  line: 1 = 0x1
                  col: 32 = 0x20
                ✓ end: cmd/compile/internal/syntax.Pos {base: *cmd/compile/internal/syntax.PosBase}(0xc0000010ae0), line: 1, col: 32
                  > base: *cmd/compile/internal/syntax.PosBase(0xc0000010ae0)
                  line: 1 = 0x1
                  col: 54 = 0x36
                  comment: "block"
                  isFunc: false
                  elems: map[string]cmd/compile/internal/types2.Object nil
                > pos: cmd/compile/internal/syntax.Pos {base: *cmd/compile/internal/syntax.PosBase}(0xc0000010ae0), line: 1, col: 54
                  > end: cmd/compile/internal/syntax.Pos {base: *cmd/compile/internal/syntax.PosBase}(0xc0000010ae0), line: 1, col: 54
                    comment: "function"
                    isFunc: true
```

## 类型匹配

将一个值赋给一个变量，需要检查类型是否相同。

类型	
Basic	基础类型通过BasicKind进行比较
Array	如果两个数组类型具有相同的元素类型及长度，则它们是相同的
Slice	如果两个切片类型具有相同的元素类型，则它们是相同的
Struct	如果两个结构类型具有相同的字段序列，并且对应的字段具有相同的名称、类型和标记，则它们是相同的。两个嵌入字段被认为具有相同的名称。来自不同包的小写字段名称总是不同的
Pointer	如果两个指针类型具有相同的基类型，则它们是相同的
Tuple	如果两个元组具有相同数量的元素和类型
Signature	如果两个函数类型具有相同数量的参数和结果值，则两个函数类型相同，相应的参数和结果类型相同，或者两个函数都是可变的，或者两者都不是。参数名和结果名不需要匹配。 泛型函数还必须具有匹配的类型参数列表，但参数名称除外。
Sum	如果两个Sum类型包含相同的类型，则它们是相同的
Interface	如果两个接口类型具有相同名称和相同函数类型的相同方法集，则它们是相同的。来自不同包的小写方法名称总是不同的。方法的顺序是不相关的。
Map	如果两个Map类型具有相同的键和值类型，则它们是相同的。
Chan	如果两个通道类型具有相同的值类型和相同的方向，则它们是相同的。

Named	如果两个命名类型的类型名源自相同的类型声明，则它们是相同的。
-------	--------------------------------

## 表达式

go表达式分为如下两种：

```
// 一元表达式
unaryOpPredicates = opPredicates{
    syntax.Add: isNumeric,
    syntax.Sub: isNumeric,
    syntax.Xor: isInteger,
    syntax.Not: isBoolean,
}
```

```
// 二元表达式
binaryOpPredicates = opPredicates{
    syntax.Add: isNumericOrString,
    syntax.Sub: isNumeric,
    syntax.Mul: isNumeric,
    syntax.Div: isNumeric,
    syntax.Rem: isInteger,

    syntax.And:    isInteger,
    syntax.Or:     isInteger,
    syntax.Xor:    isInteger,
    syntax.AndNot: isInteger, // &^ 位清空运行算

    syntax.AndAnd: isBoolean, // &&
    syntax.OrOr:   isBoolean, // ||
}
```

基本算法自顶到底递归检查表达式的类型，如果是常量的话，直接计算其结果。

```
func (check *Checker) expr(x *operand, e syntax.Expr) {
    check.rawExpr(x, e, nil)
    check.exclude(x, 1<<novalue|1<<builtin|1<<typexpr)
    check.singleValue(x)
}

func (check *Checker) rawExpr(x *operand, e syntax.Expr, hint Type)
exprKind {
```

```

        kind := check.exprInternal(x, e, hint)

        check.record(x)

        return kind
    }

// 检查表式式类型
func (check *Checker) exprInternal(x *operand, e syntax.Expr, hint Type)
exprKind {
    x.mode = invalid
    x.typ = Typ[Invalid]

    switch e := e.(type) {
    case nil:
        unreachable()
        .....
    case *syntax.Operation:
        // Y为空代表一元表达式
        if e.Y == nil {
            // unary expression
            //一元表达式; 一元表式
            if e.Op == syntax.Mul {
                // pointer indirection
                //指针间接
                check.exprOrType(x, e.X)
                switch x.mode {
                case invalid:
                    goto Error
                case typexpr:
                    x.typ = &Pointer{base: x.typ}
                default:
                    if typ := asPointer(x.typ); typ != nil {
                        x.mode = variable
                        x.typ = typ.base
                    } else {
                        check.errorf(x, invalidOp+"cannot
indirect %s", x)
                        goto Error
                    }
                }
            }
            break
        }
        check.unary(x, e)
    }
}
```

```

        if x.mode == invalid {
            goto Error
        }
        if e.Op == syntax.Recv {
            x.expr = e
            return statement // receive operations may
appear in statement context
        }
        break
    }

    // binary expression
    check.binary(x, e, e.X, e.Y, e.Op)
    if x.mode == invalid {
        goto Error
    }

    .....
    default:
        panic(fmt.Sprintf("%s: unknown expression type %T",
posFor(e), e))
    }

    x.expr = e
    return expression
}

func (check *Checker) binary(x *operand, e syntax.Expr, lhs, rhs
syntax.Expr, op syntax.Operator) {
    var y operand

    // 递归调用
    check.expr(x, lhs)
    check.expr(&y, rhs)

    .....

    if !check.op(binaryOpPredicates, x, op) {
        x.mode = invalid
        return
    }

    if op == syntax.Div || op == syntax.Rem {
        //检查零除数
        if (x.mode == constant_ || isInteger(x.typ)) && y.mode ==

```

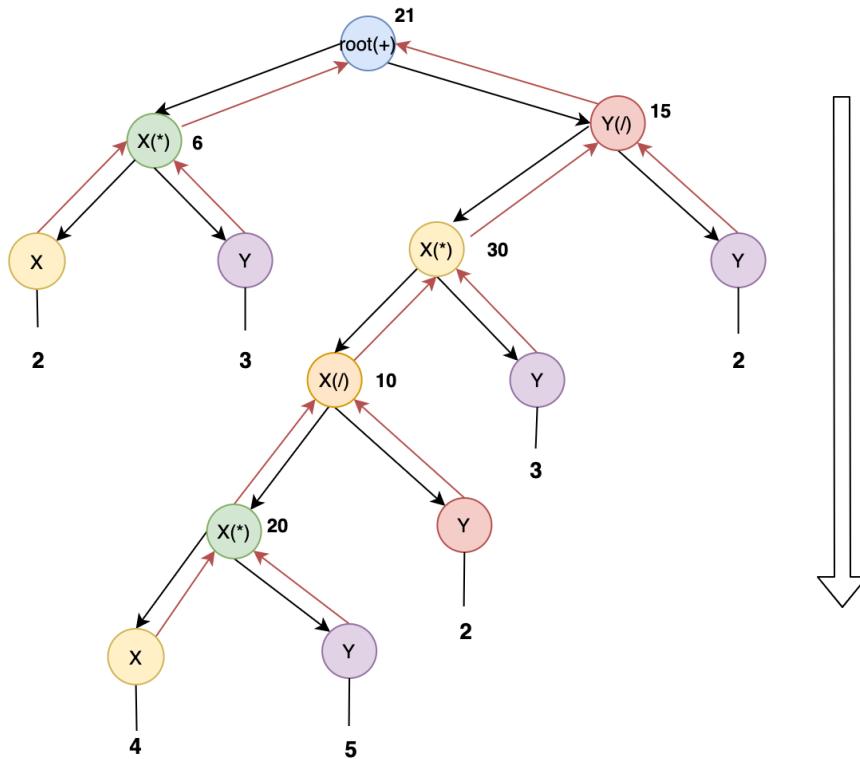
```
constant_ && constant.Sign(y.val) == 0 {
    check.error(&y, invalidOp+"division by zero")
    x.mode = invalid
    return
}
.....
}

if x.mode == constant_ && y.mode == constant_ {
    .....
    // 对整数操作数强制进行整数除法
    tok := op2tok[op]
    if op == syntax.Div && isInteger(x.typ) {
        tok = token.QUO_ASSIGN
    }
    x.val = constant.BinaryOp(x.val, tok, y.val)
    x.expr = e
    check.overflow(x)
    return
}

x.mode = value //操作数是一个计算后的值
// x.typ is unchanged
}
```

这里递归主要采用了自顶向下的方式，先分解，然后解决问题。

```
var exp float64 = 2 * 3+4 * 5 / 2 * 3 / 2
```



## Body Stmt

body里的内容都在延迟队列进行类型检查, 这里介绍一下switchStmt。

```
func (check *Checker) stmt(ctx StmtContext, s syntax.Statement) {
    // statements must end with the same top scope as they started with
    if debug { ... }

    // process collected function literals before scope changes...
    defer check.processDelayed(len(check.delayed))

    inner := ctx && (fallthroughOk || finalSwitchCase)
    switch s := s.(type) {
        case *syntax.EmptyStatement:
            // ignore

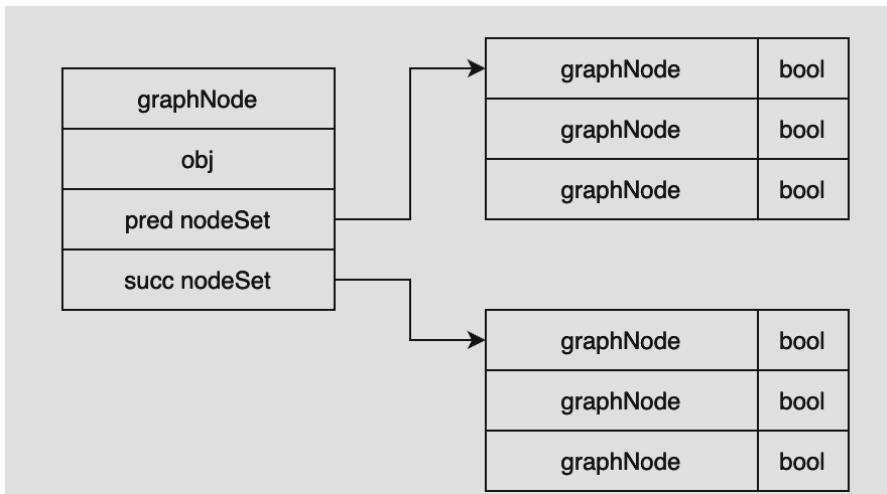
        case *syntax.DeclarationStatement:
        case *syntax.LabeledStatement:
        case *syntax.ExpressionStatement:
        case *syntax.SendStatement:
        case *syntax.AssignmentStatement:
        case *syntax.CallStatement:
        case *syntax.ReturnStatement:
        case *syntax.BranchStatement:
        case *syntax.BlockStatement:
        case *syntax.IfStatement:
        case *syntax.SwitchStatement:
        case *syntax.SelectStatement:
        case *syntax.ForStatement:
        default:
            check.error(s, "invalid statement")
    }
}
```

```
case *syntax.IfStmt:
    // 确定if的作用域
    check.openScope(s, "if")
    defer check.closeScope()

    // 检查if的init条件
    check.simpleStmt(s.Init)
    var x operand
    // if cond
    check.expr(&x, s.Cond)
    if x.mode != invalid && !isBoolean(x.typ) {
        //if 语句中的非布尔条件
        check.error(s.Cond, "non-boolean condition in if statement")
    }
    // 检查 then
    check.stmt(inner, s.Then)
    //解析器产生一个正确的 AST 但如果它被修改了
    //其他地方的 else 分支可能是无效的。再检查一遍。
    switch s.Else.(type) {
        case nil:
            //有效或已报告错误
        case *syntax.IfStmt, *syntax.BlockStmt:
            check.stmt(inner, s.Else)
        default:
            //if 语句中的 else 分支无效
            check.error(s.Else, "invalid else branch in if statement")
    }
```

## 初始化对象顺序

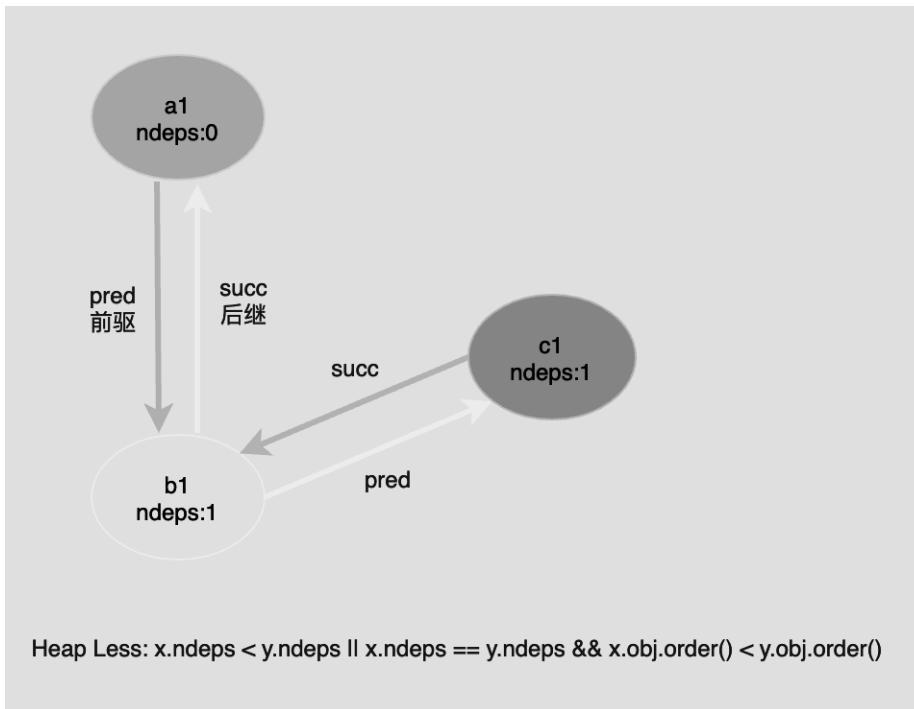
在多个对象有着相互依赖的关系，这里就需要对初始化对象进行排序的，主要的思想先构建对象依赖关系的有向图，然后对每个节点的依赖数目为权重构建最小堆，作为最小优先级队列，目前采用的是go的heap包来实现的，因此队头对象总是依赖对象最少的，所以该队列的遍历顺序就是初始化的顺序。最小优先队列算法这里就不介绍了，具体可以参考算法导论排序算法。



例子：

```
package p
var
(
    a1 int
    b1 = a1 + 10
    c1 = b1 + 20
)
```

对象依赖图：



这里a1的ndeps为0, 直接从堆中弹出, 将所依赖的节点ndeps减1, 也就是a1的前驱b1, a1没有任务依赖。b1的ndeps这里候为0, 直接弹出b1, 最后背出c1。

# IR

编译分为前端和后端，而IR就是用来衔接前后端。源代码翻译成目标机器代码的时候，编译器会构造出一系列中间表示法，go编译器会构造出IR树，该结构更接近源语言层次结构，便于进行静态类型检查之类处理，随后再构造出三地址代码的IR，这种结构与机器指令更加相似，适用于机器相关的任务处理。

这里需要注意的是IR的类型检查目前没有完成，还是用的之前类型检查函数，所以需要将type2的类型转换成type类型。

## IR-Tree 数据结构

IR数据结构主要有三个：

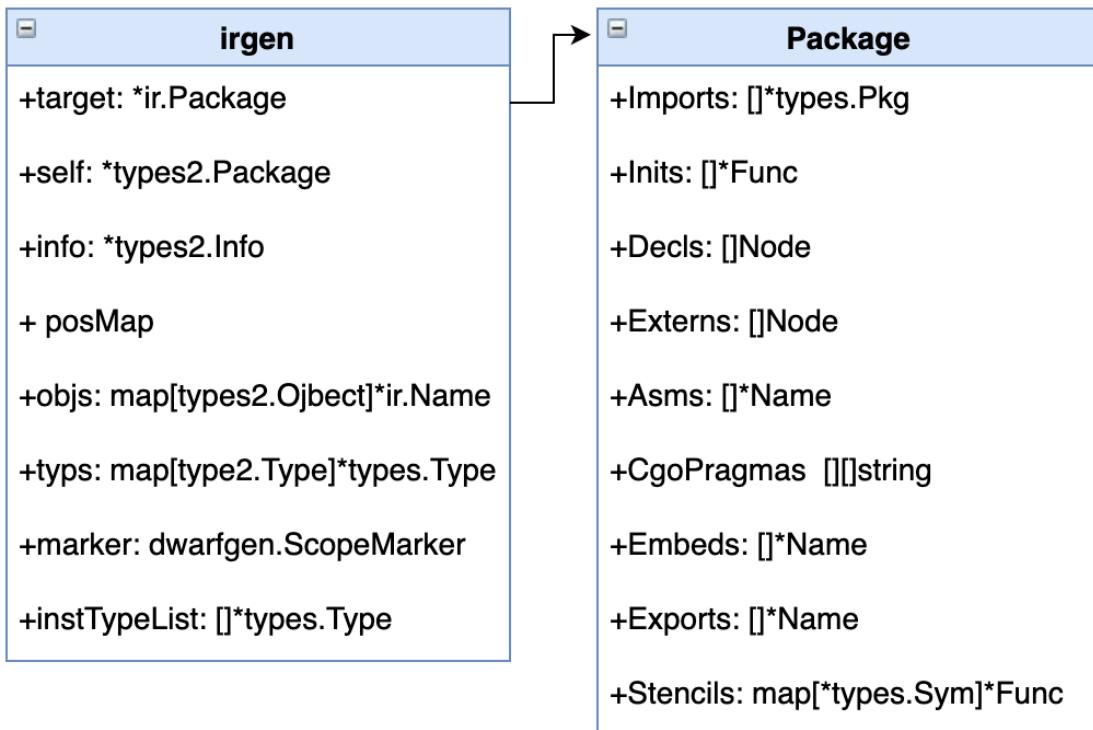
- 1、miniType 存储类型，比如chan、struct、map及array等。
- 2、miniExpr 存储表达式，比如Call、AddString 及 structKey等。
- 3、miniStmt 存储语句的声明，比如Assign、If及For等。

这些节点都会嵌入miniNode结构体，也会实现Node接口，irgen主要存储解析后的IR树。

IR结构体初始化：

```
g := irgen{  
    target: typecheck.Target,  
    self: pkg,  
    info: &info,  
    posMap: m,  
    objs: make(map[types2.Object]*ir.Name),  
    typs: make(map[types2.Type]*types.Type),  
}  
g.generate(noders)
```

## Irgen结构体



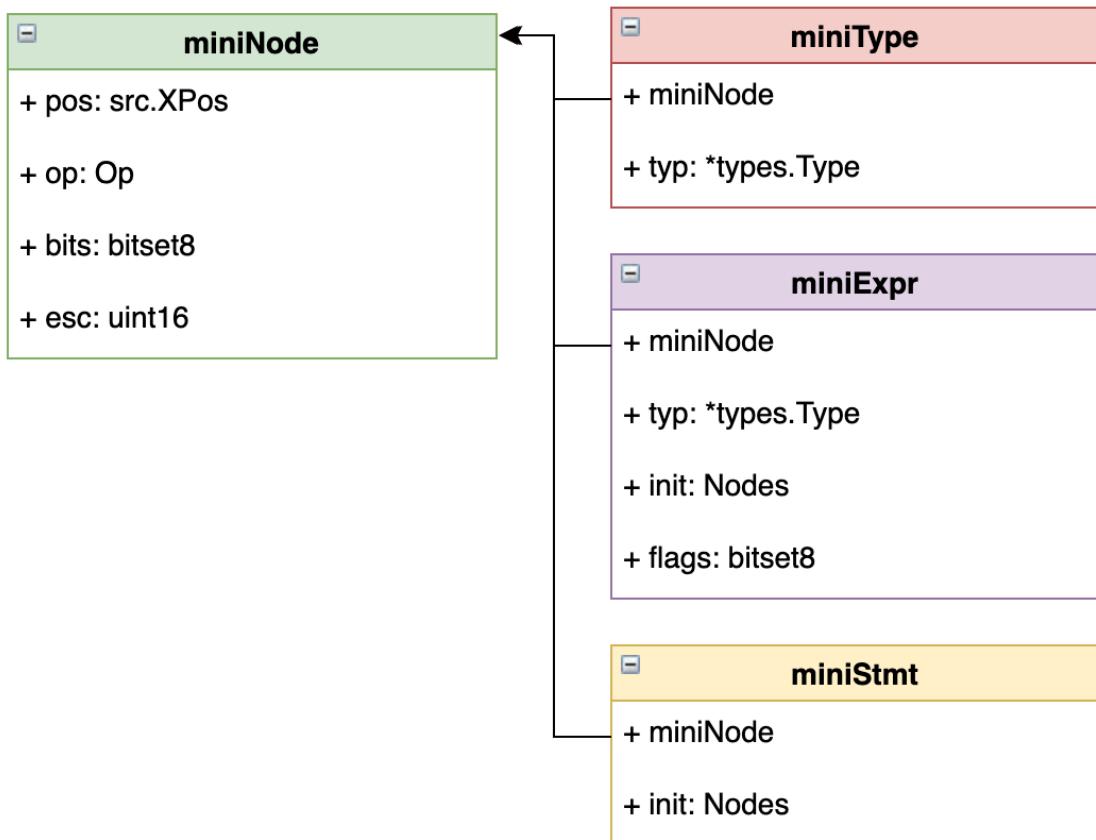
`target`字段存储包信息，包信息包括`imports`和`init`函数，`decls`的节点等。`self`字段指的是`types2`包信息，也就是之前`type2`检查时存储的信息。`objs`和`typs`主要是对`type2`对象和类型的快速索引。`info`字段保存已检查类型包的类型结果。`instTypeList`存储完全实例化的泛型。`marker`主要是对DWARF调试信息的标识符。

## Node Interface

<b>&lt;&lt;Expr&gt;&gt; Interface</b>	<b>&lt;&lt;Node&gt;&gt; Interface</b>
+ Node + isStmt()	+ Format(s fmt.State, verb rune) + Pos() : src.XPos + SetPos(x src.XPos)
<b>&lt;&lt;Ntype&gt;&gt; Interface</b>	+ copy() : Node
+ Node + CanBeNtype()	+ doChildren(func(Node) bool): bool + editChildren(func(Node) Node)
<b>&lt;&lt;Stmt&gt;&gt; Interface</b>	+ Op() : Op + Init(): Nodes
+ Node +isExpr()	+ Type(): *types.Type + SetType(t *types.Type) + Name(): *Name + Sym(): *types.Sym + Val(): constant.Value + SetVal(v constant.Value)
	+ Esc(): uint16 + SetEsc(x uint16) + Diag(): bool + SetDiag(x bool)
	+ Typecheck(): uint8 + SetTypecheck(x uint8) + NonNil(): bool + MarkNonNil()

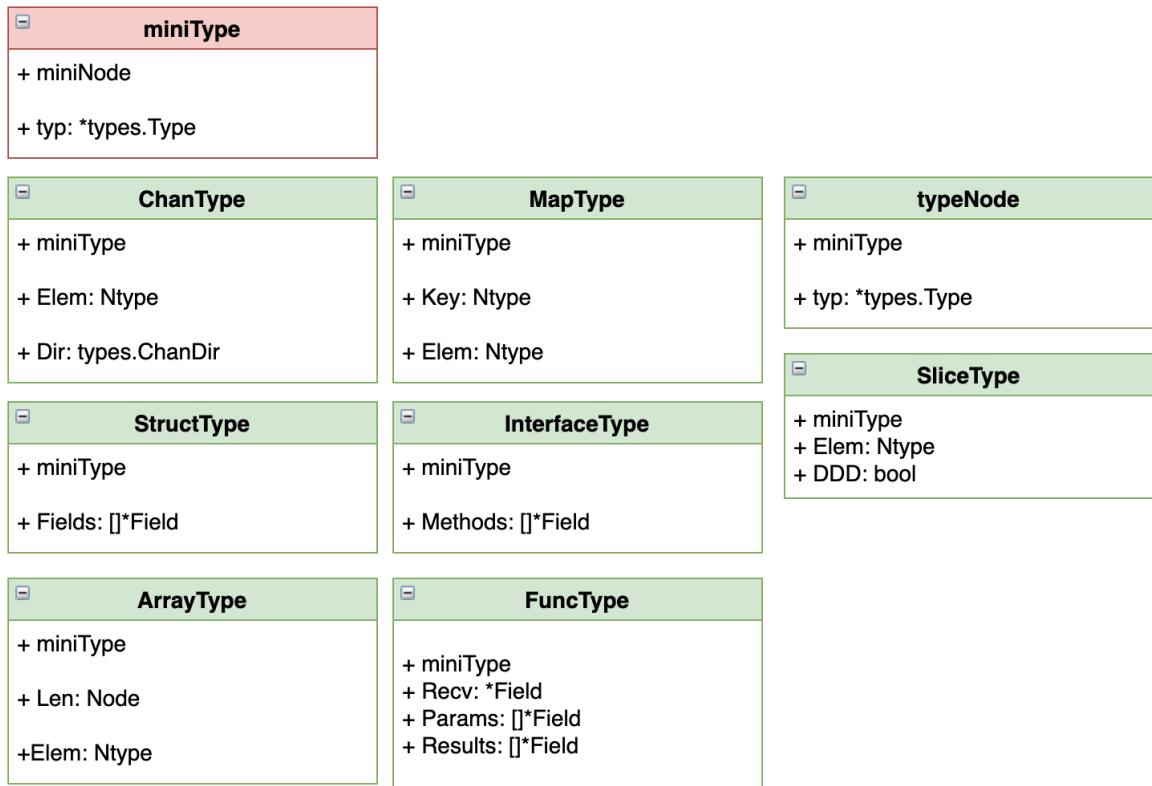
node接口主要实现了对node节点的操作，比如资源位置、副本的节点复制和编辑、节点的操作、逃逸分析及类型检测等。

## miniNode结构体



miniNode结构体会它以8个字节的代价嵌入到另个三个结构体里，形成一个最小节点，并实现了node接口方法。

## miniType 结构体

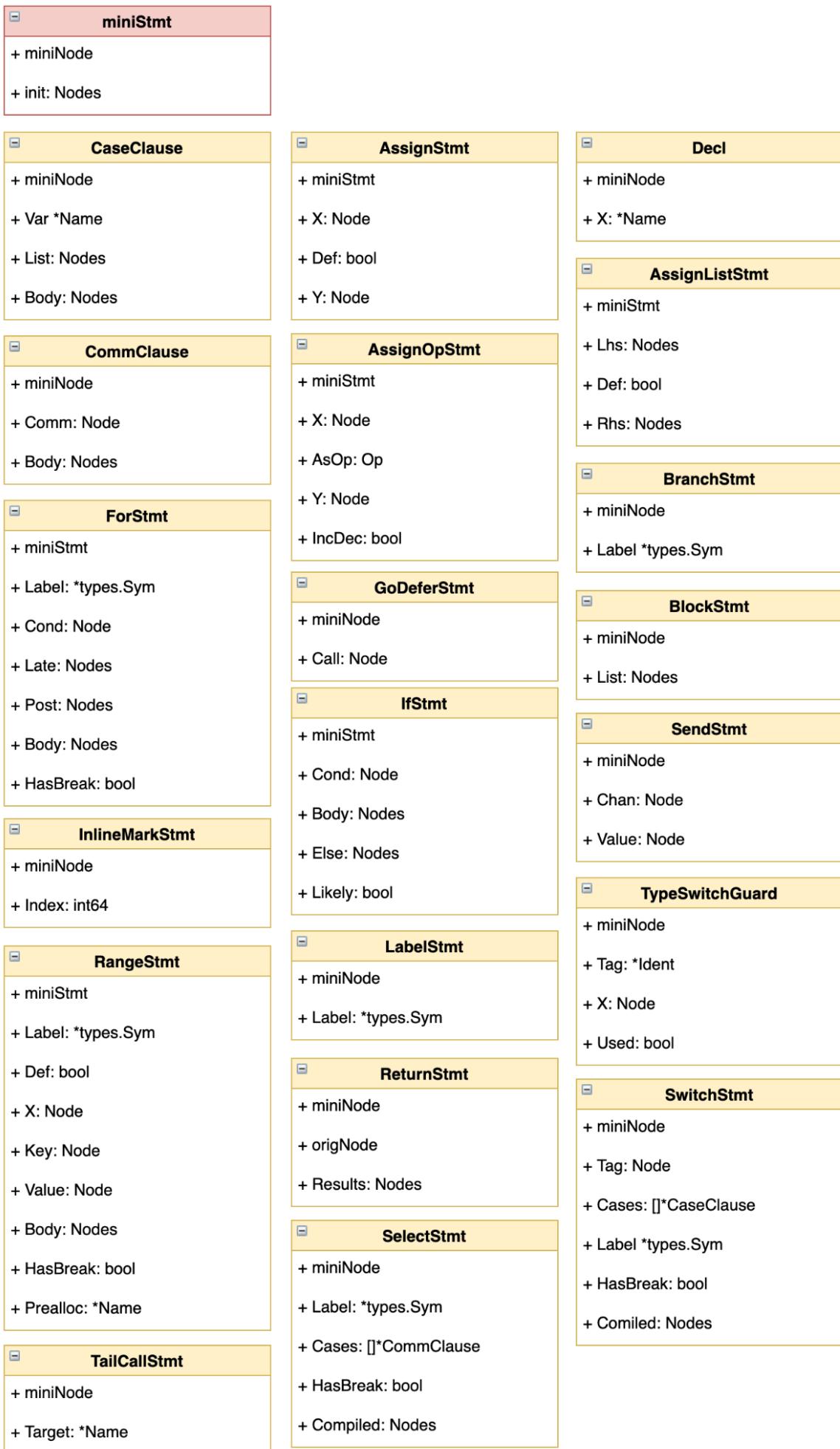


miniType主要是把type2里的类型转成IR类型。

## miniExpr结构体



miniStmt结构体



## Import

主要验证静态库的头信息是否正确，解析包名及import信息。

```
func (g *irgen) generate(noders []*noder) {
    .....
    declLists := make([][]syntax.Decl, len(noders))
Outer:
    for i, p := range noders {
        // 编译指示
        g.pragmaFlags(p.file.Pragma, ir.GoBuildPragma)
        for j, decl := range p.file.DeclList {
            switch decl := decl.(type) {
            case *syntax.ImportDecl:
                g.importDecl(p, decl)
            default:
                declLists[i] = p.file.DeclList[j:]
                continue Outer
            }
        }
    }
}
```

这里importfile函数解析的时候，我们可以看一下bytes.a文件，这里的红色部分，可以解析成ASCII对应数值。

这里import主要分为如下四块，路径检查、绝对路径、打开静态库、检查对象头信息正确性。

checkImportPath

resolveImportPath

openPackage

check object header

这里主要介绍一下检查头信息，以bytes.a静态库为例，下面是库文件的内容，注意红色部分对应的是ASCII值。这里红色的ASCII码对应值分别为：SOH:1,DC2:18,ESC:27。

version := ird.uint64() 获取uint64值，这里的值是1。

```
version := ird.uint64()
if version != iexportVersion {
    base.Errorf("import %q: unknown export format version %d", pkg.Path, version)
    base.ErrorExit()
```

```
}
```

```
func importfile(decl *syntax.ImportDecl) *types.Pkg {
    .....
    if err := checkImportPath(path, false); err != nil {
        base.Errorf("%s", err.Error())
        return nil
    }

    path, err = resolveImportPath(path)
    if err != nil {
        base.Errorf("%s", err)
        return nil
    }
```

```

.....
f, err := openPackage(path)
if err != nil {
    base.Errorf("could not import %q: %v", path, err)
    base.Exit()
}
imp := bio.NewReader(f)
defer imp.Close()
file := f.Name()

// check object header
p, err := imp.ReadString("\n")
if err != nil {
    base.Errorf("import %s: reading input: %v", file, err)
    base.Exit()
}

if p == "!<arch>\n" { // package archive
    //包导出块应该是第一个
    sz := archive.ReadHeader(imp.Reader, "__.PKGDEF")
    .....
}

// 不是 go 目标文件
if !strings.HasPrefix(p, "go object") {
    base.Errorf("import %s: not a go object file: %s", file, p)
    base.Exit()
}
q := objabi.HeaderString()
if p != q {
    base.Errorf("import %s: object is [%s] expected [%s]", file, p, q)
    base.Exit()
}

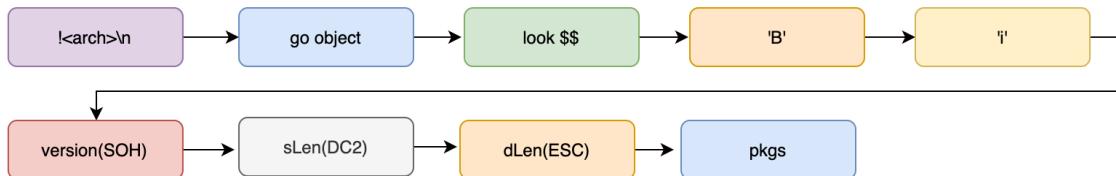
.....
//期望 $$B\n 表示二进制导入格式。

// look for @@
var c byte
for {
    c, err = imp.ReadByte()
    if err != nil {
        break
    }
    if c == '$' {
        c, err = imp.ReadByte()
        if c == '$' || err != nil {
            break
        }
    }
}

//获取$$之后的字符
if err == nil {
    c, _ = imp.ReadByte()
}
.....

```

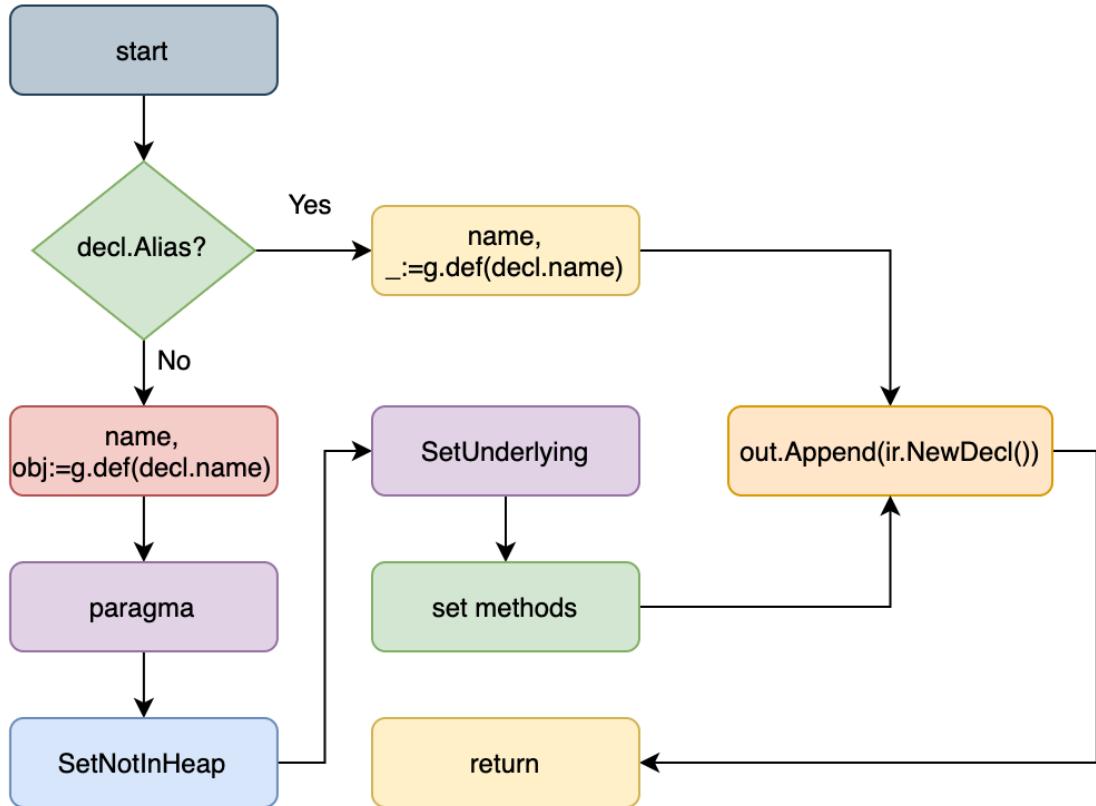
```
        return importpkg  
    }
```



```
!<arch>\n___.PKGDEF      0          0      0     644     12464\n.go object darwin amd64 go1.17.3 X:regabiwrappers,regabig,regabireflect,regabidefer,\nregabiargs\nbuild id "gz9TEABZT6qhCt2c5L2c/PRAFqR9WCrAPu4aMYzhy"\n\n$$B\ni[SOH] DC2[ETX] I[ESC] $GOROOT/src/bytes/buffer.\nbuf[BS] off[BS] lastRead[ACK] readOp[ENQ] Bytes[b] Buffer[ACK] String[ENQ] empty[ETX] Len[ETX] Cap[BS] Truncate[SOH]\net[DE] tryGrowByReslice[ETX] grow[ENQ] Write[BS] err[V] WriteString[SOH] s[BS] ReadFrom[SOH] r[ACK] Reader[STX] i\no[BS] WriteTo[SOH] w[ACK] WriterWriteByte[SOH] c[BS] WriteRune[ETX] Read[ETX] Next[BS] ReadByte[BS] ReadRune[ETX] size\nUnreadRune\nUnreadByte ReadBytes[ENQ] delim[ETX] line[ENQ] readSlice\nReadString[ACK] esc:[ENQ] STX[ETX] ~r[0] esc:[ENQ] <nil>%bytes.Buffer: truncation out of\nrange[SOH] m[BS] bytes.Buffer.Grow: negative
```

## 类型创建

### 类型声明



types2.Object生成ir.Name

```

func (g *irgen) obj(obj types2.Object) *ir.Name {
    .....

    if name, ok := g.objs[obj]; ok {
        return name
    }

    var name *ir.Name
    pos := g.pos(obj)

    class := typecheck.DeclContext
    if obj.Parent() == g.self.Scope() {
        // 前向引用包块声明
        class = ir.PEXTERN
    }

    switch obj := obj.(type) {
    case *types2.Const:
        name = g.objCommon(pos, ir.OLITERAL, g.sym(obj), class, g.typ(obj.Type()))

    case *types2.Func:
        .....
    case *types2.TypeName:
        .....

    case *types2.Var:
        .....
    }
}

```

```

default:
    g.unhandled("object", obj)
}

g.objs[obj] = name
name.GetTypecheck(1)
return name
}

声明语句处理
for _, declList := range declLists {
    g.target.Decls = append(g.target.Decls, g.decls(declList)...)
}

func (g *irgen) decls(decls []syntax.Decl) []ir.Node {
    var res ir.Nodes
    for _, decl := range decls {
        switch decl := decl.(type) {
        case *syntax.ConstDecl:
            g.constDecl(&res, decl)
        case *syntax.FuncDecl:
            g.funcDecl(&res, decl)
        case *syntax.TypeDecl:
            if ir.CurFunc == nil {
                continue // already handled in irgen.generate
            }
            g.typeDecl(&res, decl)
        case *syntax.VarDecl:
            g.varDecl(&res, decl)
        default:
            g.unhandled("declaration", decl)
        }
    }
    return res
}

//处理stmt节点
func (g *irgen) stmt(stmt syntax.Stmt) ir.Node {
    switch stmt := stmt.(type) {
    case nil, *syntax.EmptyStmt:
        return nil
    case *syntax.LabeledStmt:
        return g.labeledStmt(stmt)
    case *syntax.BlockStmt:
        return ir.NewBlockStmt(g.pos(stmt), g.blockStmt(stmt))
    case *syntax.ExprStmt:
        .....
    case *syntax.SendStmt:
        .....
    case *syntax.DeclStmt:
        .....
    case *syntax.AssignStmt:
        .....
    case *syntax.BranchStmt:
        .....
    case *syntax.CallStmt:
        .....
    }
}

```

```

    default:
        g.unhandled("statement", stmt)
        panic("unreachable")
    }
}

func (g *irgen) ifStmt(stmt *syntax.IfStmt) ir.Node {
    init := g.stmt(stmt.Init)
    n := ir.NewIfStmt(g.pos(stmt), g.expr(stmt.Cond), g.blockStmt(stmt.Then), nil)
    if stmt.Else != nil {
        e := g.stmt(stmt.Else)
        if e.Op() == ir.OBLOCK {
            e := e.(*ir.BlockStmt)
            n.Else = e.List
        } else {
            n.Else = []ir.Node{e}
        }
    }
    return g.init(init, n)
}

```

## 泛型函数实例化

泛型函数在go里面相当于定义了一个模版，使用模版生成函数。

举个例子：

```

package main

import "fmt"

type sumAny interface{
    type int, float64
}

func sum[T sumAny](i, j T) T {
    return i + j
}

func main() {
    a := sum[int](1,3)
    b := sum[float64](1.1, 2.1)
    fmt.Println(a, b)
}

```

这里sum使用了泛型，如果使用普通函数的话，可以写成这样

```

func sumInt(i, j int) int {
    return i + j
}

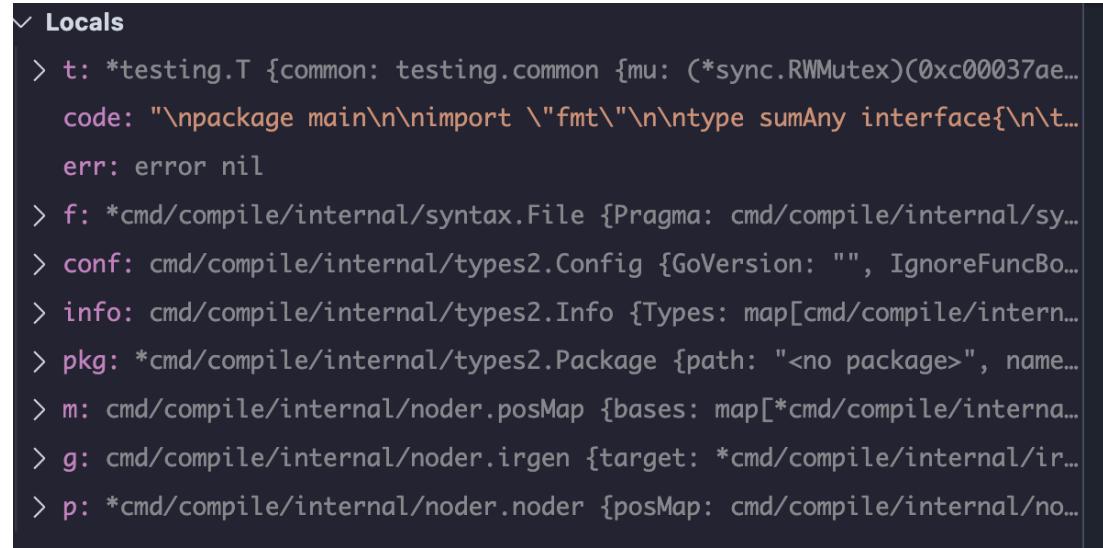
func sumFloat64(i, j float64)float64 {

```

```
    return i + j  
}
```

其实两个函数是等价的，一个代码量少，一个多而已。编译器会生成sum[int]和sum[float64]的函数，只是函数名不同，最后会把IR上的泛型函数声明进行删除。

这里我们使用调试工具看一下p和g变量的结构，也就是我们之前的types和IR树的结构。



```
Locals  
> t: *testing.T {common: testing.common {mu: (*sync.RWMutex)(0xc00037ae...  
code: "\npackage main\n\nimport \"fmt\"\n\ntype sumAny interface{\n\t...  
err: error nil  
> f: *cmd/compile/internal/syntax.File {Pragma: cmd/compile/internal/sy...  
> conf: cmd/compile/internal/types2.Config {GoVersion: "", IgnoreFuncBo...  
> info: cmd/compile/internal/types2.Info {Types: map[cmd/compile/intern...  
> pkg: *cmd/compile/internal/types2.Package {path: "<no package>", name...  
> m: cmd/compile/internal/noder.posMap {bases: map[*cmd/compile/interna...  
> g: cmd/compile/internal/noder.irgen {target: *cmd/compile/internal/ir...  
> p: *cmd/compile/internal/noder.noder {posMap: cmd/compile/internal/no...
```

```
✓ g: cmd/compile/internal/noder.irgen {target: *cmd/compile/internal/ir.Package {Imports: []...}}
  ✓ target: *cmd/compile/internal/ir.Package {Imports: []*cmd/compile/internal/types.Pkg...}
    ✓ : cmd/compile/internal/ir.Package {Imports: []*cmd/compile/internal/types.Pkg len:...}
      ✓ Imports: []*cmd/compile/internal/types.Pkg len: 1, cap: 1, [*(*"cmd/compile/internal/types.Pkg")(0xc00007ba40)]
        > [0]: *(*"cmd/compile/internal/types.Pkg")(0xc00007ba40)
          Inits: []*cmd/compile/internal/ir.Func len: 0, cap: 0, nil
        > Declss: []cmd/compile/internal/ir.Node len: 4, cap: 6, [...,...,...,...]
        > Externs: []cmd/compile/internal/ir.Node len: 1, cap: 1, [...]
        Asms: []*cmd/compile/internal/ir.Name len: 0, cap: 0, nil
        CgoPragmas: [][]string len: 0, cap: 0, nil
        Embeds: []*cmd/compile/internal/ir.Name len: 0, cap: 0, nil
        Exports: []*cmd/compile/internal/ir.Name len: 0, cap: 0, nil
  ✓ Stencils: map[*cmd/compile/internal/types.Sym]*cmd/compile/internal/ir.Func [*{Li...
    len(): 2
    ✓ [key 0]: *cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)}
      ✓ : cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)}
        Linkname: ""
        > Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)
        Name: "sum[int]"
        > Def: cmd/compile/internal/types.Object(*cmd/compile/internal/ir.Name) *{miniE...
          Block: 0
        > Lastlineno: cmd/internal/src.XPos {index: 0, lico: 0}
          flags: 16 = 0x10
      > [val 0]: *cmd/compile/internal/ir.Func {miniNode: (*"cmd/compile/internal/ir.m...}
    ✓ [key 1]: *cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)}
      ✓ : cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)}
        Linkname: ""
        > Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc00007a5a0)
        Name: "sum[float64]"
        > Def: cmd/compile/internal/types.Object(*cmd/compile/internal/ir.Name) *{miniE...
          Block: 0
        > Lastlineno: cmd/internal/src.XPos {index: 0, lico: 0}
          flags: 16 = 0x10
    > [val 1]: *cmd/compile/internal/ir.Func {miniNode: (*"cmd/compile/internal/ir.m...}
```

```
✓ g: cmd/compile/internal/noder.irgen {target: *cmd/compile/internal/ir.Package {Imports: [] *cmd/compile/internal/types.Pkg...}}
  ✓ target: *cmd/compile/internal/ir.Package {Imports: [] *cmd/compile/internal/types.Pkg...}
    ✓ : cmd/compile/internal/ir.Package {Imports: [] *cmd/compile/internal/types.Pkg len: 0, cap: 0, nil}
      ✓ Imports: [] *cmd/compile/internal/types.Pkg len: 1, cap: 1, [*(*cmd/compile/internal/types.Pkg)(0xc00007ba40)]
        > [0]: (*cmd/compile/internal/types.Pkg)(0xc00007ba40)
        Inits: [] *cmd/compile/internal/ir.Func len: 0, cap: 0, nil
      ✓ Decl: [] cmd/compile/internal/ir.Node len: 4, cap: 6, [...,...,...,...]
        > [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Decl) {*miniNode: cmd/compile/internal/ir.miniNode{...}}
        > [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Func) {*miniNode: cmd/compile/internal/ir.miniNode{...}}
        > [2]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Func) {*miniNode: cmd/compile/internal/ir.miniNode{...}}
        > [3]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Func) {*miniNode: cmd/compile/internal/ir.miniNode{...}}
      > Externs: [] cmd/compile/internal/ir.Node len: 1, cap: 1, [...]
      Asms: [] *cmd/compile/internal/ir.Name len: 0, cap: 0, nil
      CgoPragmas: [] [] string len: 0, cap: 0, nil
      ✓ Body: cmd/compile/internal/ir.Nodes len: 3, cap: 4, [...,...,...]
```

```
✓ [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Func) {*miniNode: cmd/compile/internal/ir.miniNode{...}}
  ✓ data: *cmd/compile/internal/ir.Func {*miniNode: cmd/compile/internal/ir.miniNode {pos: (*cmd/internal/ir.XPos)(0xc000158420), op: ODCL...}}
    ✓ : cmd/compile/internal/ir.Func {*miniNode: cmd/compile/internal/ir.miniNode {pos: (*cmd/internal/src.XPos)(0xc000158420), op: ODCL...}}
      > miniNode: cmd/compile/internal/ir.miniNode {pos: (*cmd/internal/src.XPos)(0xc000158420), op: ODCL...}
    ✓ Body: cmd/compile/internal/ir.Nodes len: 3, cap: 4, [...,...,...]
      > [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) {*miniStmt: cmd/compile/internal/ir.miniStmt{...}}
      > [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) {*miniStmt: cmd/compile/internal/ir.miniStmt{...}}
      > [2]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.CallExpr) {*miniExpr: cmd/compile/internal/ir.miniExpr{...}}
```

```

✓ Body: cmd/compile/internal/ir.Nodes len: 3, cap: 4, [..., ..., ...]
✓ [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{miniStmt: cmd/compile/internal/...
✓ data: *cmd/compile/internal/ir.AssignStmt {miniStmt: cmd/compile/internal/ir.miniStmt {miniNode: (*"cmd...
✓ : cmd/compile/internal/ir.AssignStmt {miniStmt: cmd/compile/internal/ir.miniStmt {miniNode: (*"cmd/co...
> miniStmt: cmd/compile/internal/ir.miniStmt {miniNode: (*"cmd/compile/internal/ir.miniNode")(0xc0000b...
> X: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Name) *{miniExpr: (*"cmd/compile/internal/i...
    Def: true
✓ Y: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.CallExpr) *{miniExpr: (*"cmd/compile/internal...
✓ data: *cmd/compile/internal/ir.CallExpr {miniExpr: (*"cmd/compile/internal/ir.miniExpr")(0xc00015cb...
✓ : cmd/compile/internal/ir.CallExpr {miniExpr: (*"cmd/compile/internal/ir.miniExpr")(0xc00015cbd0), ...
    > miniExpr: cmd/compile/internal/ir.miniExpr {miniNode: cmd/compile/internal/ir.miniNode {pos: (*"c...
    > origNode: cmd/compile/internal/ir.origNode {orig: cmd/compile/internal/ir.Node(*cmd/compile/internal...
✓ X: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Name) *{miniExpr: cmd/compile/internal/i...
    ✓ data: *cmd/compile/internal/ir.Name {miniExpr: cmd/compile/internal/ir.miniExpr {miniNode: (*"...
    ✓ : cmd/compile/internal/ir.Name {miniExpr: cmd/compile/internal/ir.miniExpr {miniNode: (*"cmd/c...
        > miniExpr: cmd/compile/internal/ir.miniExpr {miniNode: (*"cmd/compile/internal/ir.miniNode")(0...
            BuiltInOp: OXXX (0) = 0x0
        Class: PFUNC (7) = 0x0
        pragma: 0
        flags: 0 = 0x0
    ✓ sym: *cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*cmd/compile/internal/types.Pkg")...
    ✓ : cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*cmd/compile/internal/types.Pkg")(0x...
        Linkname: ""
        > Pkg: *(*cmd/compile/internal/types.Pkg")(0xc00007a5a0)
        Name: "sum[int]"
        ✓ Def: cmd/compile/internal/types.Object(*cmd/compile/internal/ir.Name) *{miniExpr: cmd/comp...
            Block: 0
            > Lastlineno: cmd/internal/src.XPos {index: 0, lico: 0}
            flags: 16 = 0x10
        > Func: *cmd/compile/internal/ir.Func {miniNode: (*"cmd/compile/internal/ir.miniNode")(0xc00015...
            Offset_: 0
            val: go/constant.Value nil
            Opt: interface {} nil
            Embed: *[]cmd/compile/internal/ir.Embed nil
            PkgName: *cmd/compile/internal/ir.PkgName nil
        > Defn: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Func) *{miniNode: (*"cmd/compile/...
            Curfn: *cmd/compile/internal/ir.Func nil
            Ntype: cmd/compile/internal/ir.Ntype nil
            Heapaddr: *cmd/compile/internal/ir.Name nil
            Innermost: *cmd/compile/internal/ir.Name nil
            Outer: *cmd/compile/internal/ir.Name nil
        > Args: cmd/compile/internal/ir.Nodes len: 2, cap: 2, [..., ...]
            KeepAlive: []*cmd/compile/internal/ir.Name len: 0, cap: 0, nil
            IsDDD: false
            Use: CallUseExpr (1) = 0x0
            NoInline: false
            PreserveClosure: false

```

# 初始化任务

创建初始化任务主要分为四部：

- 1、处理全局变量赋值语句
- 2、处理import语句，按顺序查找出所有依赖包中初始化任务
- 3、创建一个init函数，执行动态赋值语句，并将该函数的函数体设置为所有的动态值语句
- 4、创建.inittask函数，将依次写入 deps 及 fns

## 初始化方式

- 1、静态初始化，一般指常量定义及单一的变量赋值等。
- 2、运行时初始化，一般指字符串连接引用其它变量及获取切片的值等。

```
// 尝试静态初始化
func (s *Schedule) tryStaticInit(nn ir.Node) bool {
    // 只关心简单的"l = r"赋值。多变量/表达式 OAS2 赋值已经
    // 替换为多个简单的 OAS 分配，另一个
    // OAS2* 分配大多需要动态初始化。
    if nn.Op() != ir.OAS {
        return false
    }
    n := nn.(*ir.AssignStmt)
    // 如果是_的话,
    if ir.IsBlank(n.X) && !AnySideEffects(n.Y) {
        // 丢弃
        return true
    }
    lno := ir.SetPos(n)
    defer func() { base.Pos = lno }()
    nam := n.X.(*ir.Name)
    return s.StaticAssign(nam, 0, n.Y, nam.Type())
}

func (s *Schedule) staticcopy(l *ir.Name, loff int64, rn *ir.Name, typ *types.Type) bool {
    .....
    r := rn.Defn.(*ir.AssignStmt).Y
    if r == nil {
        // 没有明确的初始化值。可能归零也可能是外部提供的且未知。
        return false
    }

    for r.Op() == ir.OCONVNOP && !types.Identical(r.Type(), typ) {
        r = r.(*ir.ConvExpr).X
    }

    // 如果操作类型不是如下几种，返回false。
    switch r.Op() {
    case ir.OMETHEXPR:
        .....
    case ir.ONAME:
        .....
    case ir.ONIL:
        return true
    }
}
```

```

case ir.OLITERAL:
    .....
case ir.OADDR:
    .....
case ir.OPTRLIT:
    .....
case ir.OSLICELIT:
    r := r.(*ir.CompLitExpr)
    // copy slice, 静态数据都需要重新复制一下值, 写到ctx.hash里的map里
    staticdata.InitSlice(l, loff, staticdata.GlobalLinksym(s.Temp[s.r]), r.Len)

    .....
case ir.OARRAYLIT, ir.OSTRUCTLIT:
    .....
}

return false
}

```

举个例子

```

package main

// 静态初始化语句
var a int = 100
var b = []string{"Goalng"}

// 动态初始化语句
var b1 = b[0]
var c = a + 100

```

l指定的是当前node节点, 目前是四个, [0]就是var a int = 100, s下的out存的是动态初始化语句。

```
↳ locals
  ↘ l: []cmd/compile/internal/ir.Node len: 4, cap: 4, [*cmd/compile/internal/ir...
    > [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{...
    > [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{...
    > [2]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{...
    > [3]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{...
    ~r1: []cmd/compile/internal/ir.Node len: 0, cap: 0, nil
  ↘ s: cmd/compile/internal/staticinit.Schedule {Out: []cmd/compile/internal/ir...
    ↘ Out: []cmd/compile/internal/ir.Node len: 2, cap: 2, [...,...]
      > [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *...
      > [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *...
    > Plans: map[cmd/compile/internal/ir.Node]*cmd/compile/internal/staticinit...
    > Temps: map[cmd/compile/internal/ir.Node]*cmd/compile/internal/ir.Name [...]
  ↘ o: cmd/compile/internal/pkginit.InitOrder {blocking: map[cmd/compile/intern...
    blocking: map[cmd/compile/internal/ir.Node][]cmd/compile/internal/ir.Node...
    ready: cmd/compile/internal/pkginit.declOrder len: 0, cap: 1, []
  > order: map[cmd/compile/internal/ir.Node]int [...: -1000, ...: -1000, ...:...
```

op:OINDEX指的是var b1 = b[0]里b切片索引, OINDEX操作属于动态初始化语句。

```

    > l: []cmd/compile/internal/ir.Node len: 4, cap: 4, [*cmd/compile/internal/ir.AssignStmt ... 17
    > [0]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{miniStmt: (*"... 17
    > [1]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{miniStmt: (*"... 17
    > [2]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{miniStmt: (*"... 17
    >   data: *cmd/compile/internal/ir.AssignStmt {miniStmt: (*"cmd/compile/internal/ir.mini... 17
    >     : cmd/compile/internal/ir.AssignStmt {miniStmt: (*"cmd/compile/internal/ir.miniStmt... 17
    >       miniStmt: cmd/compile/internal/ir.miniStmt {miniNode: cmd/compile/internal/ir.mini... 17
    >         X: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Name) *{miniExpr: cmd/com... 17
    >           Def: false
    >         Y: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.IndexExpr) *{miniExpr: cm... 17
    >           data: *cmd/compile/internal/ir.IndexExpr {miniExpr: cmd/compile/internal/ir.miniE... 18
    >             : cmd/compile/internal/ir.IndexExpr {miniExpr: cmd/compile/internal/ir.miniExpr... 18
    >               miniExpr: cmd/compile/internal/ir.miniExpr {miniNode: (*"cmd/compile/internal... 18
    >                 miniNode: cmd/compile/internal/ir.miniNode {pos: cmd/internal/src.XPos {index... 18
    >                   pos: cmd/internal/src.XPos {index: 2, lico: 32944}
    >                     op: 0INDEX (66) = 0x0
    >                     bits: 4 = 0x1
    >                     esc: 0 = 0x0
    >                   typ: *(*cmd/compile/internal/types.Type")(0xc0001ee310)
    >                     : cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 16, metho...
    >                       init: cmd/compile/internal/ir.Nodes len: 0, cap: 0, nil
    >                       flags: 0 = 0x0
    >                     X: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.Name) *{miniExpr: (*...
    >                     Index: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.BasicLit) *{miniE...
    >                       Assigned: false
    > [3]: cmd/compile/internal/ir.Node(*cmd/compile/internal/ir.AssignStmt) *{miniStmt: (*"...

```

## 初始化算法

主要是采用heap堆排序来实现依赖关系, 具体可以参数算法导论里的堆排序。

## 数据结构

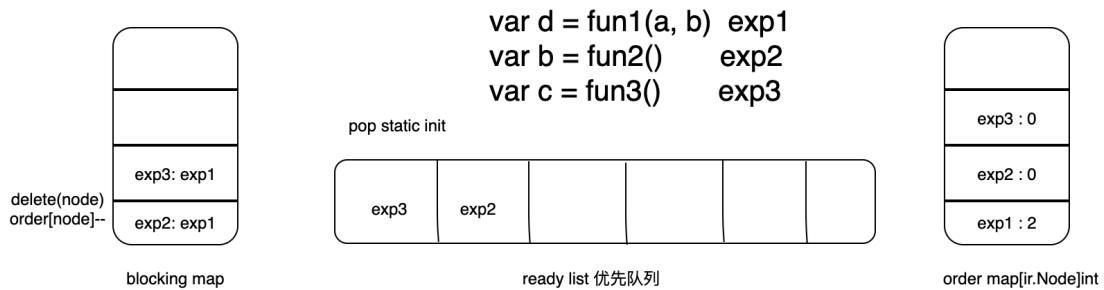
```

const (
    InitNotStarted = iota // 未开始
    InitDone            // 已完成
    InitPending          // 待办
)

type InitOrder struct {
    blocking map[ir.Node][]ir.Node
    // ready 是等待初始化分配的队列
    // 准备好初始化。
    ready declOrder
    order map[ir.Node]int
}

```

}



## 执行步骤

- 1、处理Notstarted的赋值语句，解析依赖列表
- 2、计算order与blocking的数据，使其状态为pending
- 3、pop出ready语句进行staticInit, 找出blocking里的相关的语句，将order为0的pending语句，将其放入ready 队列中，标记为Done，并将其blocking列表中的所有语句order值减一
- 4、重得第三步走到处理完所有pending语句

## 清除无效代码

无效代码指定的是死代码用永远不会执行的代码。

例子：

```
package main

func main() {
    if true {
        goto label
        return
    }

label:
}
```

这个条件永远是true,属于一种无效代码。注意这里的label标签是不能被清除的，不然会引起panic。

```
func stmts(nn *ir.Nodes) {
    var lastLabel = -1
    for i, n := range *nn {
        if n != nil && n.Op() == ir.OLABEL {
            lastLabel = i
        }
    }
    for i, n := range *nn {
```

```

cut := false
if n == nil {
    continue
}
if n.Op() == ir.OIF {
    n := n.(*ir.IfStmt)
    n.Cond = expr(n.Cond)
    if ir.IsConst(n.Cond, constant.Bool) {
        var body ir.Nodes
        if ir.BoolVal(n.Cond) {
            // 清除else节点
            n.Else = ir.Nodes{}
            body = n.Body
        } else {
            // 清除body节点
            n.Body = ir.Nodes{}
            body = n.Else
        }
        // 此逻辑就是为了解析上面例子导致panic。
        if body := body; len(body) != 0 {
            switch body[(len(body) - 1)].Op() {
            case ir.ORETURN, ir.OTAILCALL, ir.OPANIC:
                if i > lastLabel {
                    cut = true
                }
            }
        }
    }
}

if len(n.Init()) != 0 {
    stmts(n.(ir.InitNode).PtrInit())
}
switch n.Op() {
case ir.OBLOCK:
    n := n.(*ir.BlockStmt)
    stmts(&n.List)
case ir.OFOR:
    n := n.(*ir.ForStmt)
    stmts(&n.Body)
case ir.OIF:
    n := n.(*ir.IfStmt)
    stmts(&n.Body)
    stmts(&n.Else)
case ir.ORANGE:
    n := n.(*ir.RangeStmt)
    stmts(&n.Body)
case ir.OSELECT:
    n := n.(*ir.SelectStmt)
    for _, cas := range n.Cases {
        stmts(&cas.Body)
    }
case ir.OSWITCH:
    n := n.(*ir.SwitchStmt)
    for _, cas := range n.Cases {
        stmts(&cas.Body)
    }
}

```

```

        if cut {
            *nn = (*nn)[:i+1]
            break
        }
    }

func expr(n ir.Node) ir.Node {
    //对&& || 的常量的表达式代码，执行死代码消除。
    //产生一个恒定的"if"条件。
    switch n.Op() {
    case ir.OANDAND:
        n := n.(*ir.LogicalExpr)
        n.X = expr(n.X)
        n.Y = expr(n.Y)
        if ir.IsConst(n.X, constant.Bool) {
            if ir.BoolVal(n.X) {
                return n.Y // true && x => x
            } else {
                return n.X // false && x => false
            }
        }
    case ir.OOROR:
        n := n.(*ir.LogicalExpr)
        n.X = expr(n.X)
        n.Y = expr(n.Y)
        if ir.IsConst(n.X, constant.Bool) {
            if ir.BoolVal(n.X) {
                return n.X // true || x => true
            } else {
                return n.Y // false || x => x
            }
        }
    }
    return n
}

```

## 内联函数

内联函数可以提高程序执行效率。如果函数是内联的，编译时会把内联函数的实现替换到每个调用内联函数地方。

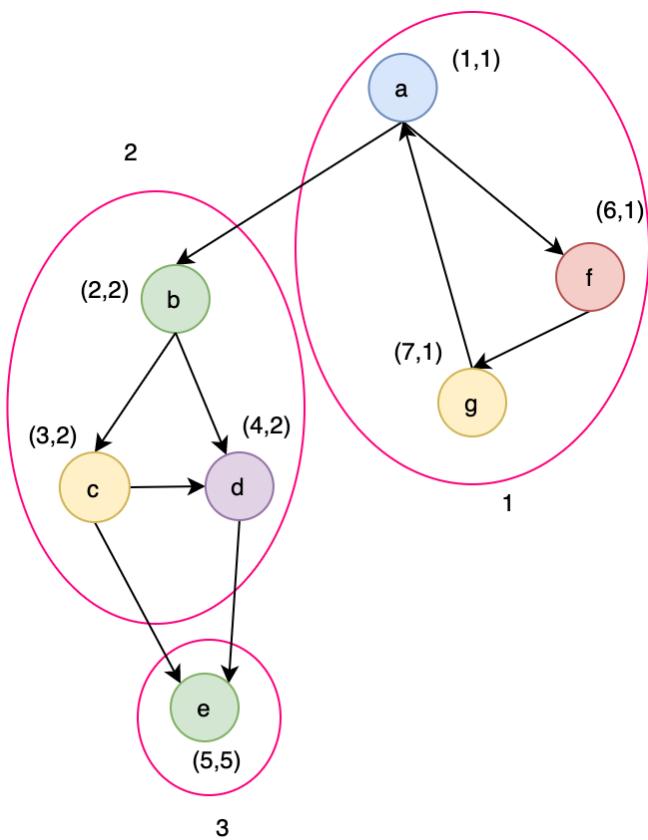
使用内联函数可以解决小函数上下文切分带来的资源占用，但是会影响文件存储大小及内存空间的占用，如果函数体内有循环，那么执行函数代码时间比调用开销大。go提供了一个编译指令`go:noinline`可以使函数不使用内联。

内联函数主要分为三个流程，节点遍历、确认内联函数、创建内联函数节点。

### SCC

`scc`(Strongly connected component)强连通分量 在有向图的数学理论中，如果每个顶点都可以从其他每个顶点到达，则称该图为强连通图，任意有向图的强连通分量形成子图的分

区，子图又形成有向无环图。可以在线性时间(即  $\Theta(V + E)$ ) 内测试图的强连通性，或找到其强连通分量。



上图强连通图有3个，画圈的地方代表一个强连通图。

如何用算法来实现强连通图呢？这里需要介绍一下tarjan算法。

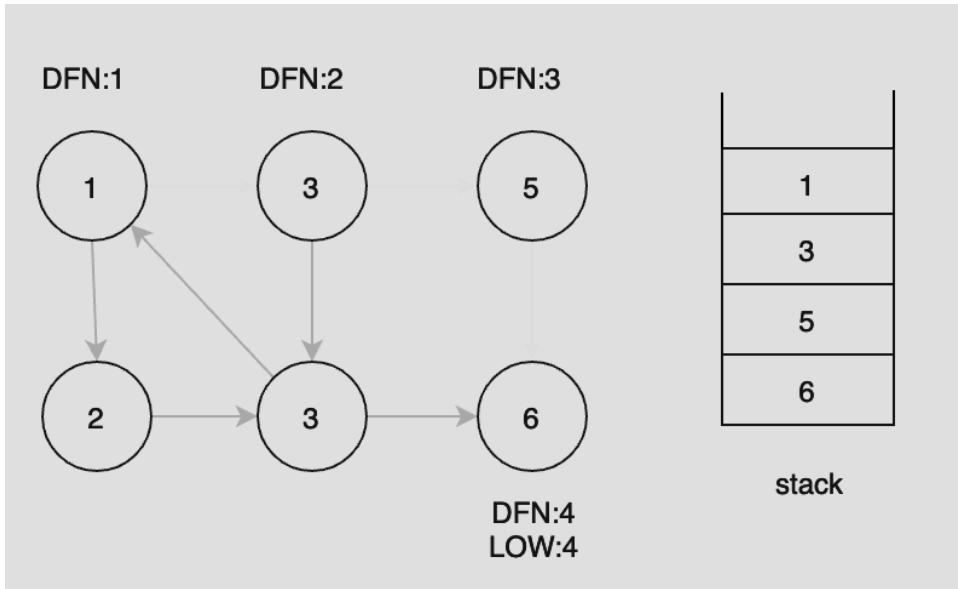
四条边：

树枝边：DFS时经过的边，即DFS搜索树上的边。

前向边：与DFS方向一致，从某个结点指向其某个子孙的边。

后向边：与DFS方向相反，从某个结点指向其某个祖先的边。（返祖边）

横叉边：从某个结点指向搜索树中的另一子树中的某结点的边。



伪代码

```
time = 1 // 当前被访问的时间点
stack
dfn[v] = {0} // 深度优先搜索遍历时结点 u 被搜索的次序。
low[v] = {0} // 在 u 的子树中能够回溯到的最早的已经在栈中的结点
```

```
func tarjan()
for x = 0; x < v; x ++
    if dfn[x] == 0
        DFS(X)

DFS(x)
    stack.push(x)
    dfn[x]=low[x]=time++
    // 回溯
    for y = 0; y < v; y ++
        if g[x][y] { //邻接矩阵存储
            if dfn[y] == 0 // 有其它路线可走
                DFS(y)
                low[x] = min(low[x], low[y])
            else if y in stack
                low[x] = min(low[x], low[y])

        if dfn[x] == low[x]
            stack.pop() //找到强连通分量
```

数据结构

```
type bottomUpVisitor struct {
    analyze func([]*Func, bool) // 逃逸分析
    visitgen uint32
    nodeID map[*Func]uint32
    stack []*Func
}
```

# 内联检查

如何判断一个函数是否有内联，主要分为如下条件：

## 1、基础条件

标记为“go:noinline”，则不需要内联

标记为 "go:norace" 和 -race 编译，则不需要内联

标记为 "go:nochekptr" 和 -d checkptr 编译，则不需要内联

标记为“go:cgo\_unsafe\_args”，则不需要内联

标记为“go:uintptrescapes”，则不需要内联

fn 没有主体(在 Go 之外定义)，则不能内联它

## 2、代价条件

总体函数代价(budget)超过预算，则不需要内联，预算的默认值是80

这里主要介绍一下代价计算规则：

## 数据结构

```
type hairyVisitor struct {
    // 内联预算, 起始值为 80, 函数体内操作都会有对应代价
    // 总体代价超过预算的话就无法内联
    budget int32
    // 原因
    reason string
    // 方法调用的开销, 默认为 57
    extraCallCost int32
    // 用来记录当前函数的局部变量
    usedLocals ir.NameSet
    // 遍历函数并进行复杂度计算
    do func(ir.Node) bool
}

func (v *hairyVisitor) doNode(n ir.Node) bool {
    if n == nil {
        return false
    }
    switch n.Op() {
        //如果 inlinable 调用是可以的，并且我们有 body 的预算。
        case ir.OCALLFUNC: //函数调用
            n := n.(*ir.CallExpr)
            //调用 runtime.getcaller{pc,sp} 的函数不能内联
            //因为 getcaller{pc,sp} 需要一个指向调用者第一个参数的指针。
            //
            //runtime.throw 是一个"廉价调用", 就像普通代码中的恐慌一样。
            if n.X.Op() == ir.ONAME {
                name := n.X.(*ir.Name)
                // 全局函数并且是运行时的包
                if name.Class == ir.PFUNC && types.IsRuntimePkg(name.Sym().Pkg) {
                    fn := name.Sym().Name
                    if fn == "getcallerpc" || fn == "getcallersp" {
                        v.reason = "call to " + fn
                        return true
                    }
                    if fn == "throw" {

```

```

        v.budget -= inlineExtraThrowCost
        break
    }
}

if ir.IsIntrinsicCall(n) {
    //像任何其他节点一样对待。
    break
}

if fn := inlCallee(n.X); fn != nil && fn.Inl != nil {
    // 减去预算成本
    v.budget -= fn.Inl.Cost
    break
}

// 预算成本减去减去内联调用的开销,extraCallCost默认是57
v.budget -= v.extraCallCost

case ir.OCALLMETH:
case ir.OCALL, ir.OCALLINTER:

case ir.OPANIC:

case ir.ORECOVER:

case ir.OCLOSURE:

.....
}

v.budget--

return ir.DoChildren(n, v.do)
}

```

知道被调用函数的内联代价之后，就是遍历SCC(CFG图的强连通分量)，依次判断内联代价是否小于上述中指定的阈值，成立则会进行内联。

## 逃逸分析

逃逸分析主要是确定哪些变量(包括隐式分配，比如new或make的调用及复合类型等)可以在堆上分配。

逃逸的规则：

- 1、指向栈对象的指针不能分配到堆中。
- 2、指向栈上对象的指针，其生命周期不能长于该对象。

我们通过对AST的静态数据流分析来实现逃逸分析。

首先需要构造一个有向加权图，其中顶点(称为位置)表示由语句分配的变量和表达式，边代表变量之间的赋值(权重表示寻址和引用计数)。

通过遍历图寻找可能的赋值路径，如果变量V的地址是存储在堆或其他比它寿命更长的地方，那么V标记为堆分配。

每个分配语句(比如变量声明)或者表达式(比如 "new"或"make")首先映射到一个唯一的"location"。

为每条边加了一个权重, 该权重的计算方式为 : derefs = 引用解析次数 - 取地址次数。

例子:

```
p = &q    // -1  
p = q     // 0  
p = *q    // 1  
p = **q   // 2  
  
p = **&**&q // 2
```

注意&操作符只能应用于取地址表达式, 而表达式&x本身是不可寻址的, 所以derefs不能低于-1, \*操作符应用于引用解析。

每个 Go 语言结构都被降级为这种表示, 通常对流程、路径或上下文不敏感; 并且不区分复合变量中的元素。例如:

```
var x struct { f, g *int }  
var u []*int  
x.f = u[0]
```

// 简单地建模为

```
x = *u
```

也就是说, 我们不区分 x.f 和 x.g 或者 u[0] 和 u[1] 等。但是, 我们确实记录了索引切片所涉及的隐式引用解析。

## 静态数据流

判断是否逃逸, 编译器采用了静态数据流算法进行实现的。

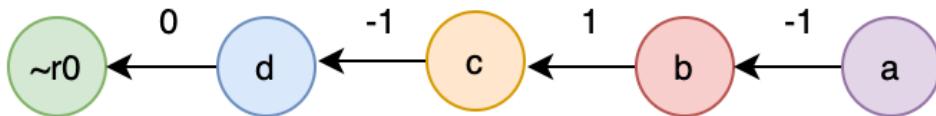
静态数据流是以函数变量和变量之前的赋值操作构建成一个有向加权图, 图的顶点代表变量, 边代表赋值语句, 方向代表数据流向。

例子:

```
type Home struct {  
    Addr string  
}  
  
func GetHome() **Home {  
    a := new(Home)  
    a.Addr = "beijing"  
  
    b := &a  
    c := *b  
    d := &c  
    return d
```

}

创建静态数据流向图



$$\begin{aligned} w(\sim r0, d) &= 0 \\ w(\sim r0, c) &= w(\sim r0, d) + -1 = 0 + -1 = -1 \\ w(\sim r0, b) &= w(\sim r0, c) + 1 = -1 + 1 = 0 \\ w(\sim r0, a) &= w(\sim r0, b) + -1 = -1 \end{aligned}$$

从静态数据流向图 可以看出来c和a是逃逸的。这里参数并没有设计到引用，也就是变量的生命周期问题。

我们也可以直接用工具跑一下分析结果：

```
go tool compile -G=3 -l -m ./a.go
```

输出：

```
./a.go:16:2: moved to heap: c
./a.go:12:10: new(Home) escapes to heap
./a.go:24:13: ... argument does not escape
```

## 数据结构



batch:一个批处理持有共享整个逃逸分析状态, 其中allLocs保存数据流有向图。  
 escape:每个函数都会创建一个escape结构体, 同一个批次共享一个batch。  
 closure:处理闭包相关的逃逸分析。  
 hole:封装了赋值语句中对右侧表达式执行状态。  
 location:可以看成函数里的一个变量, 是数据流的一个顶点, 其中包含隐式(make和new等)和显示的节点。  
 edge:数据流有向图中的一条边, 属性 derefs 表示该边权重, 一条边代表程序中的一个赋值语句。

## 执行流程

Batch批处理函数主要是对函数进行数据流图的构造及逃逸分析。

```

func Batch(fns []*ir.Func, recursive bool) {
    // 操作类型必须是函数
    for _, fn := range fns {
        if fn.Op() != ir.ODCLFUNC {
            base.Fatalf("unexpected node: %v", fn)
        }
    }

    var b batch
    b.heapLoc.escapees = true

    // 从语法树构造数据流图
    for _, fn := range fns {
        if base.Flag.W > 1 {
            s := fmt.Sprintf("\nbefore escape %v", fn)
            ir.Dump(s, fn)
        }
        b.initFunc(fn)
    }
    for _, fn := range fns {
        // 如果函数内部有闭包, 则为真
        // 如果是简单函数或全局变量初始化中的闭包, 则为 false
        if !fn.IsHiddenClosure() {
            // 遍历函数的语法树, 根据赋值语句创建有向图边及权重计算
            b.walkFunc(fn)
        }
    }

    // 处理闭包值或引用, 并捕获它们的自由变量。
    for _, closure := range b.closures {
        b.flowClosure(closure.k, closure.clo)
    }
    b.closures = nil

    // 检查各个变量大小, 大对象直接逃逸到堆上
    for _, loc := range b.allLocs {
        if why := HeapAllocReason(loc.n); why != "" {
            b.flow(b.heapHole().addr(loc.n, why), loc)
        }
    }

    // 对有向权值图各个顶点进行逃逸分析
    b.walkAll()
}

```

```

    // 逃逸分析完成, 在语法树中标记各个变量顶点结果
    b.finish(fns)
}

```

## 创建顶点

顶点的含义指的是函数里的局部变量，为每个顶点创建一个location对象，并加到批处理里的allLocs中去。这里需要注意有些隐式的(new与make)及字面值初始顶点，需要在构造边的时候创建。

```

func (b *batch) initFunc(fn *ir.Func) {
    // 初始化逃逸结构体
    e := b.with(fn)
    if fn.Esc() != escFuncUnknown {
        base.Fatalf("unexpected node: %v", fn)
    }
    fn.SetEsc(escFuncPlanned)
    if base.Flag.LowerM > 3 {
        ir.Dump("escAnalyze", fn)
    }

    // 为局部变量分配位置
    for _, n := range fn.Dcl {
        if n.Op() == ir.ONAME {
            e.newLoc(n, false)
        }
    }

    // 为结果参数初始化 resultIndex
    for i, f := range fn.Type().Results().FieldSlice() {
        e.oldLoc(f.Nname.(*ir.Name)).resultIndex = 1 + i
    }
}

```

## 创建边

创建边的逻辑比点要复杂一些，需要对stmt及expr进行分析，确定边的赋值关系，构造出完整的数据流图。hole保存expr()表达式模拟求值策略，flow()建立两个顶点的有向边。

```

func (b *batch) walkFunc(fn *ir.Func) {
    e := b.with(fn)
    // 设置esc状态符为开始
    fn.SetEsc(escFuncStarted)

    // 识别标记非结构化循环头部的标签
    ir.Visit(fn, func(n ir.Node) {
        switch n.Op() {
        case ir.OLABEL:
            n := n.(*ir.LabelStmt)
            if e.labels == nil {
                e.labels = make(map[*types.Sym]labelState)
            }
        }
    })
}

```

```

        // 非循环
        e.labels[n.Label] = nonlooping

    case ir.OGOTO:
        // If we visited the label before the goto,
        // then this is a looping label.
        // 如果我们在 goto 之前访问了标签,
        // 那么这是一个循环标签。
        n := n.(*ir.BranchStmt)
        if e.labels[n.Label] == nonlooping {
            e.labels[n.Label] = looping
        }
    }

// 处理代码块逻辑
e.block(fn.Body)

// walkFunc函数处理之后还有剩余标签
if len(e.labels) != 0 {
    base.FatalfAt(fn.Pos(), "leftover labels after walkFunc")
}
}

func (e *escape) block(l ir.Nodes) {
    old := e.loopDepth
    e.stmts(l)
    e.loopDepth = old
}

func (e *escape) stmts(l ir.Nodes) {
    for _, n := range l {
        // 遍历所有语句
        e.stmt(n)
    }
}

// 处理单个语句, 这里需要注意的是discard()函数, 有些表达式会产生一个副作用, 需要丢弃它
func (e *escape) stmt(n ir.Node) {
    if n == nil {
        return
    }

    .....

    e.stmts(n.Init())

    switch n.Op() {
    case ir.OFOR, ir.OFORUNTIL:
        n := n.(*ir.ForStmt)
        e.loopDepth++
        // for Init; Cond; Post { Body }
        e.discard(n.Cond)
        e.stmt(n.Post)
        e.block(n.Body)
        e.loopDepth--
    }
}
```

```

        case ir.ORANGE:
        .....
        case ir.OSWITCH:
        .....
        case ir.OSEND:
        .....
    }
}

```

## 逃逸分析

对有向图进行分析的时候，采用了双重队列模式进行双重循环，外层循环主要是遍历 allLocs 里的点，内层循环主要是处理当一个点拥有另一个点的指针时，需要将另外一个点压入到 allLocs 里去，重新进行外层循环。例如：root <- A <- B <- C <- D <- E 中，假如 root 逃逸了，而 root 持有是 D 的指针，这样 D 也需要逃逸，此时需要将 D 压入外层队列。

```

func (b *batch) walkAll() {
    todo := make([]*location, 0, len(b.allLocs)+1)
    enqueue := func(loc *location) {
        if !loc.queued {
            todo = append(todo, loc)
            loc.queued = true
        }
    }

    // 将 allLocs 里的数据入队列，也就是入队所有顶点
    for _, loc := range b.allLocs {
        enqueue(loc)
    }
    // 将堆地址的顶点入队
    enqueue(&b.heapLoc)

    var walkgen uint32
    // 弹出队尾数据，进行逃逸分析
    for len(todo) > 0 {
        root := todo[len(todo)-1]
        todo = todo[:len(todo)-1]
        root.queued = false

        walkgen++
        b.walkOne(root, walkgen, enqueue)
    }
}

func (b *batch) walkOne(root *location, walkgen uint32, enqueue func(*location)) {
    // 数据流图有负边（来自寻址操作），所以使用 Bellman-Ford 算法。
    // 但是，不必担心无限的负循环，因为将中间解引用计数绑定为 0。
    root.walkgen = walkgen
    root.derefs = 0
    root.dst = nil

    todo := []*location{root} // LIFO queue
    for len(todo) > 0 {
        l := todo[len(todo)-1]
        todo = todo[:len(todo)-1]

```

```

drefs := l.drefs

//如果 l.drefs < 0, 则 l 的地址流向root根。
addressOf := drefs < 0
if addressOf {
    //对于像 "root = &l; l = x" 这样的流路径,
    //l 的地址流向根, 但 x 没有。
    //我们通过在 0 处的下限 drefs 来识别这一点。
    drefs = 0
    .....
}

// root的生命周期长于 l
if b.outlives(root, l) {
    // l的值流向root。如果l是一个函数参数, 而root是堆或相应的结果参数, 那么
    //记录该值流, 以便稍后标记该函数。
    if l.isName(ir.PPARAM) {
        .....
        l.leakTo(root, drefs)
    }
}

//如果l的地址流到某个比它更长的地方, 那么l需要被堆分配。
if addressOf && !l.escapes {
    .....
    l.escapes = true
    enqueue(l)
    continue
}
}

for i, edge := range l.edges {
    if edge.src.escapes {
        continue
    }
    d := drefs + edge.drefs
    if edge.src.walkgen != walkgen || edge.src.drefs > d {
        edge.src.walkgen = walkgen
        edge.src.drefs = d
        edge.src.dst = l
        edge.src.dstEdgIdx = i
        todo = append(todo, edge.src)
    }
}
}
}

```

例子：

```

type Home struct {
    addr string
}

func home() (r1 **Home) {
    a := new(Home)

```

```
a.addr = "beijing"
```

```
b := &a
```

```
c := *b
```

```
d := b
```

```
e := d
```

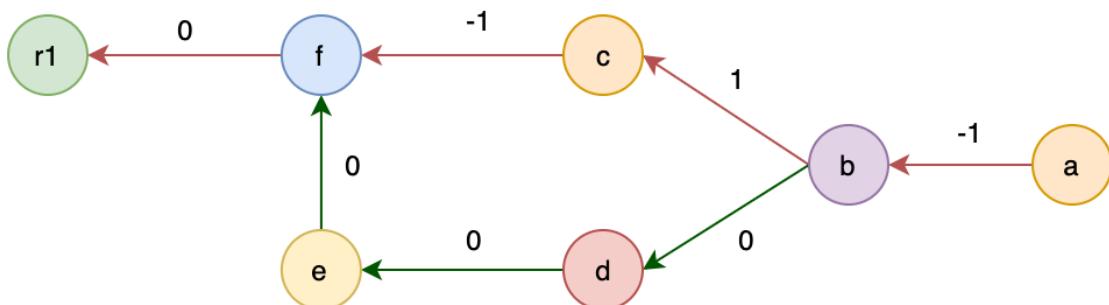
```
f := &c
```

```
f = e
```

```
return f
```

```
}
```

```
./a.go:12:2: moved to heap: a  
./a.go:17:2: moved to heap: c  
./a.go:12:10: new(Home) escapes to heap  
./a.go:34:13: ... argument does not escape
```



这里分成2个路径，红色箭头代表路径1，绿色箭头代表路径2。

从路径1和2可以分析出a和c进行了逃逸，具体计算规则可以根据之前的逃逸条件求得。

## SSA

SSA指的是静态单赋值(Static Single Assignment)，主要用途来自于它如何通过简化变量的属性来，同时简化和改进各种编译器优化的结果。我们先看一下GO 1.5 compiler流程：



GO1.5是采用90年代作为plan9编译器的，这是整个GO项目编译器的基础，以便它可以编译GO而不是C。然后在go1.5编译器代码库用C翻译为go,因此开发go1.5时，需要映射运行时函数之类的事情。

## 场景

下面代码是编译器生成汇编代码示例：

1. **MOVQ AX, BX  
SHLQ \$0x3, BX  
MOVQ BX,0x10(SP)  
CALL runtime.memmove(SB)**

为什么要执行MOVQ AX, BX？为什么编译器要这样做？为什么不是3个指令还是四个指令？

2. **IMULQ \$0x10,R8**

为什么不：

**SHLQ \$0x4, R8**

乘法指令硬件是需要多个周期才能运行完指令，是相当昂贵的，为什么不用移位来执行此操作？

0x10等于10进制是16，16的二进制是10000，只需要左移位4位。

3. **MOVQ R8, 0x20(CX)  
MOVQ 0X20(CX), R9**

为什么不：

**MOVQ R8, 0X20(CX)**

**MOVQ R8, R9**

编译器正在做一些愚蠢的事情，它正在将一个寄存器写入内存，然后将该值直接从内存中再次读取到另一个寄存器。

4. **LEAQ 0x10(SP), BX  
MOVQ BX, SI**

为什么不：

**LEAQ 0x10(SP), SI**

5. **ANDL R8, BX  
CMPL \$0x0, BX**

为什么不：

**ANDL R8, BX**

这是一个不需要比较的例子。

6. **MOVQ AX, CX  
MOVQ CX, R9**

为什么不：

**MOVQ AX, R9**

我们把数据移到一个临时寄存器，为什么不直接移动呢？

7. **XORL BP, BP**

**CMPQ BP, AX**

**JNE**

为什么不：

**TESTQ AX, AX**

**JNE**

为什么不用两个操作数进行逻辑与运，只会改变标志位不会改变操作数，在进行判断是否等于0。

例子：

<b>x := 1</b>	<b>SSA形式</b>	<b>y1 := 1</b>
<b>y := 2</b>	<b>—————&gt;</b>	<b>y2 := 2</b>
<b>x := y</b>		<b>x1 := y2</b>

可以看到第一个赋值不是必需的，并且第三行中使用的y的值来自y的第二个赋值。程序必须执行到达定义分析以确定这一点，但如果该程序采用SSA，则两者都是显而易见的。

## 发展历史

如何优化使编译器做得更好使代码体积减少20%，速度快10%呢？SSA就是解析此类问题的，编译器将AST树转换成基于SSA的表示，可以实现许多在当前编译器中很难实现的优化。

2015-02-10: SSA提案已经提交给golang开发社区。

2015-03-01: 创建SSA dev分支。

2016-03-01: SSA dev分支合并到master分支。

2016-08-15: GO 1.7开始支持SSA, 针对amd64架构。

2017-02-16: GO1.8 SSA支持架构(386、amd64,p32、arm、arm64、mips、mips64、ppc64、s390x)。

生产环境大数据报告：

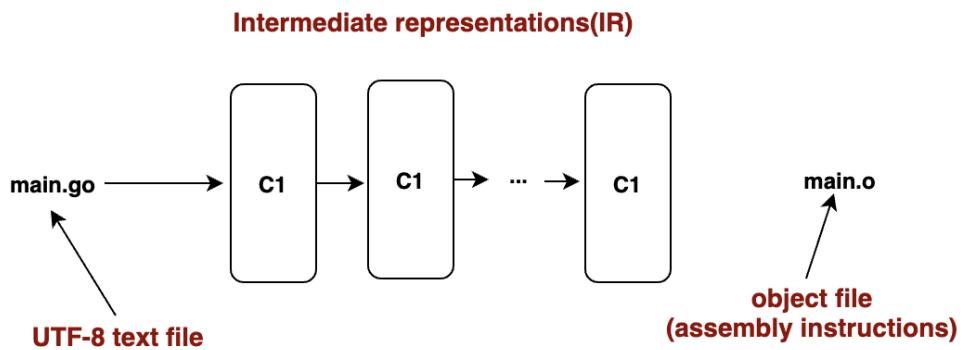
大数据工作负载-提高15%

凸包算法(凸包问题可以描述为：给定一个点集P, 求最小点集S, 使得S构成的形状能包含这些点集)-改善14-24%

哈希函数-提高39%

音频处理(arm)-提高48%

这里说明一下凸包算法优化场景：凸包函数执行大量的浮点运算，比1.5有很大改善。



编译器读取utf-8 GO源文件，通过中间组件进行处理，最后得到可执行指令。

我们将代码生成替换成SSA，主要原因是从头开始重新编写整个编译器是一项非常艰巨的工作，一次取一个片段要容易得多，我们只是想做更好的机器代码，只需要替换最后一部分。

## 静态单赋值

静态单赋值代表着每个变量只有一个赋值，通过一个使用变量重命名的过程将它们转换成静态单赋值，如下：

a = 1		a1 = 1
b = 2		b1 = 2
c = a * b	----->	c1 = a1 * b1
d = c * 5		d1 = c1 * 5

下面代码是带有分支静态单赋值：

x = 1  if a {  x = 10  }	----->	x1 = 1  if a {  x2 = 10  }
--	--------	--

`fmt.Println(x)`

`fmt.Println(x?)`

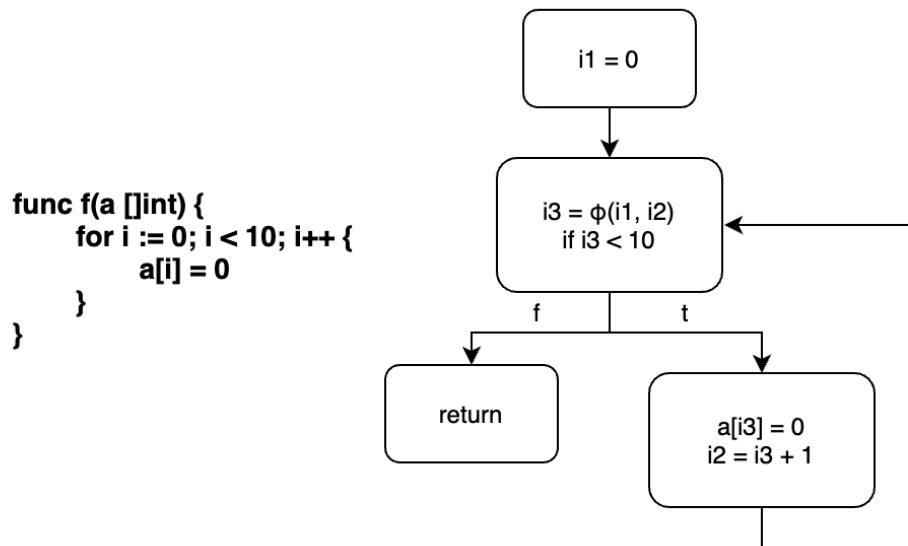
打印x需要b的条件，我们没办法输入固定数字。所以SSA中有一个 $\phi$ 函数：

`x3 = φ(x1, x2)`

`fmt.Println(x3)`

它表示两个不同值的集合

**ssa**中间表示法：



ssa为以下各项提供快速、准确的优化算法：

- 1、消除通用子表达式
- 2、无用代码消除
- 3、死存储消除
- 4、nil检查消除
- 5、边界检查消除
- 6、寄存器分配
- 7、指令调度
- 8、.....

优化算法

消除通用子表达式

$y = x + 9$		$y = x + 9$
...	----->	...
$z = x + 9$		$z = y$

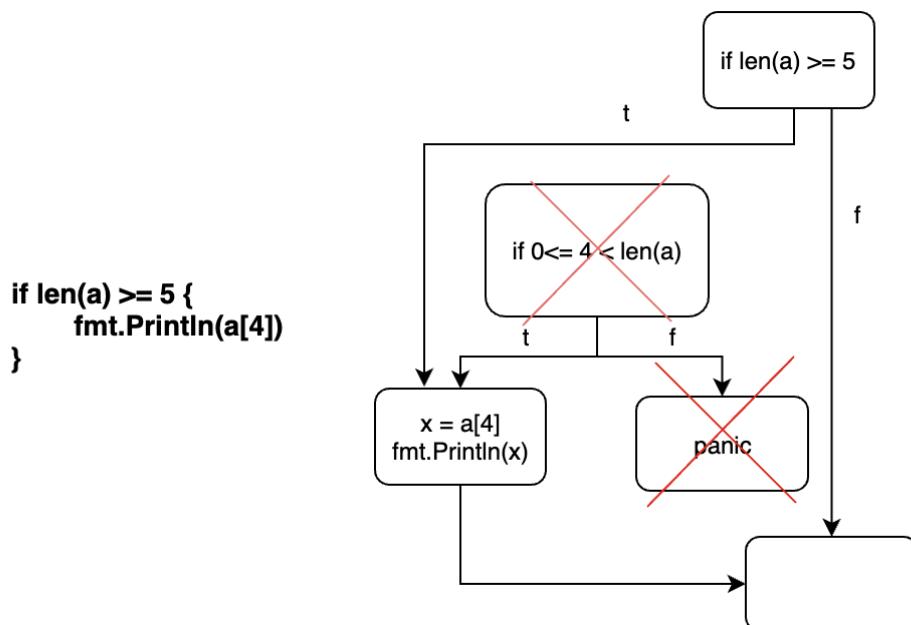
死存储消除：

$*p = 0$		$*p = \theta$
...	----->	
$*p = 1$		$*p = 1$

消除条件：

- 1、p没有改变
- 2、没有控制流使用\*p=1值
- 3、中间没有对\*p的读取

边界检查消除



重写规则

可以使用ssa定义重写规则做更多的优化

$y = x - x$		$y = 0$
(Sub64 x x)	----->	(Const64 [0])
$y = 4 * 3$		$y = 12$
(Mul64 (Const64 [c]) (Const64[d]))	----->	(Const64 [ c * d ])
$y = x * 16$	----->	$y = x << 4$

(Mul64x (Const64 [c])) && isPowerOfTwo(c)		(Lsh64x64 x (Const64 [log2(c)]))
z = x == y	----->	w = x != y w = !z
(Not(Eq64 x y))		(Neq64 x y )

## 中间表示分类

- 1、图IR, 将编译器的知识编码在图中, 通过图中的对象来表述: 结点、边及列表树。
- 2、线性IR, 类似某些抽象机上的伪代码, 迭代遍历简单的线性操作序列。
- 3、混合IR, 结合了图IR和线性IR的要素, 为的是获取两者的优势而避免其弱点。一种常见的混合表示使用底层的线性IR来表示无循环代码的块, 使用图来表示这些块之间的控制流。

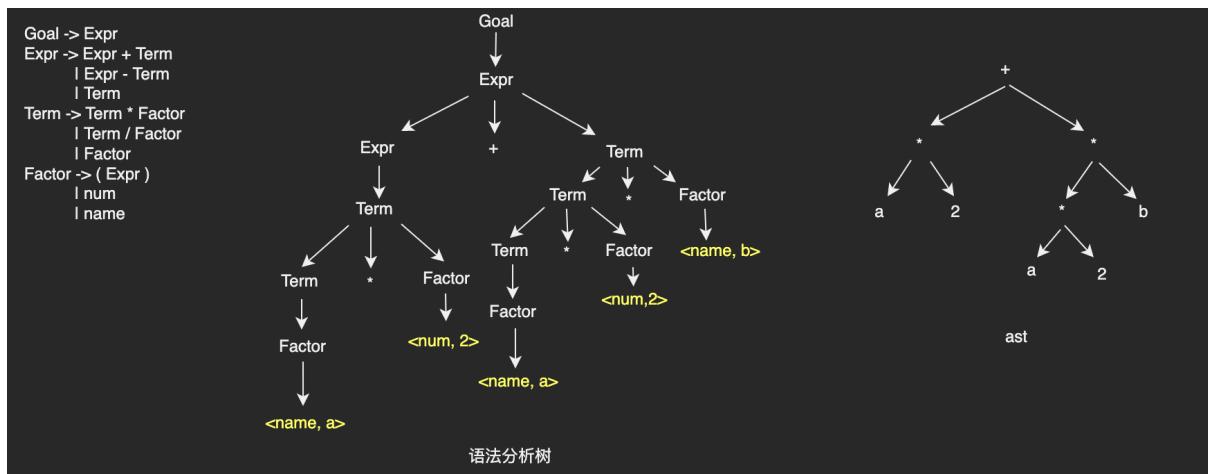
## 图IR使用场景

1、语法分析树

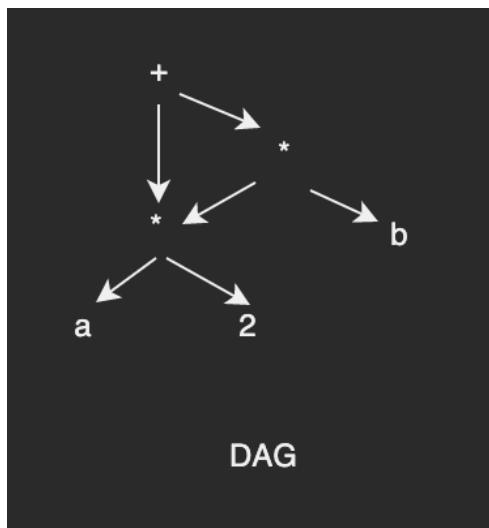
2、AST抽象语法树3、有向非循环图(DAG)

DAG是具有共享机制的一种AST。相同子树只实例化一次, 它可能有多个父结点。在实际系统中使用DAG有两个原因。如果内存约束限制了编译器能够处理的程序大小, 使用DAG可能有助于减少内存占用。第二个原历系统使用DAG是为了暴露出冗余之处, 主要在能够生成更好的编译后代码。

案例:  $a * 2 + a * 2 * b$



DAG图:



## 线性IR

汇编语言程序是一种线性代码，它包含一个指令序列，其中各个指令按出现顺序执行（或按与该顺序相一致的某种顺序执行）。

### 编译器中线性IR：

- 1、单地址，代码模拟了累加器机器和堆栈机的行为。这种代码暴露了机器对隐式名字的使用，因此编译器能够相应地调整代码，由此得出的代码相当紧凑。
- 2、二地址代码模拟了具有破坏性操作的机器。随着内存的限制变得不那么重要，这种代码废而不用了，而三地址代码可以显式模拟破坏性操作。破坏性操作：一种操作，总是用操作的结果重新定义其中一个操作数。
- 3、三地址代码模拟的是这样一种机器，其中大多数操作有两个操作数并生成一个结果。RISC体系结构的崛起，使这种代码流行起来。

## 单地址应用场景

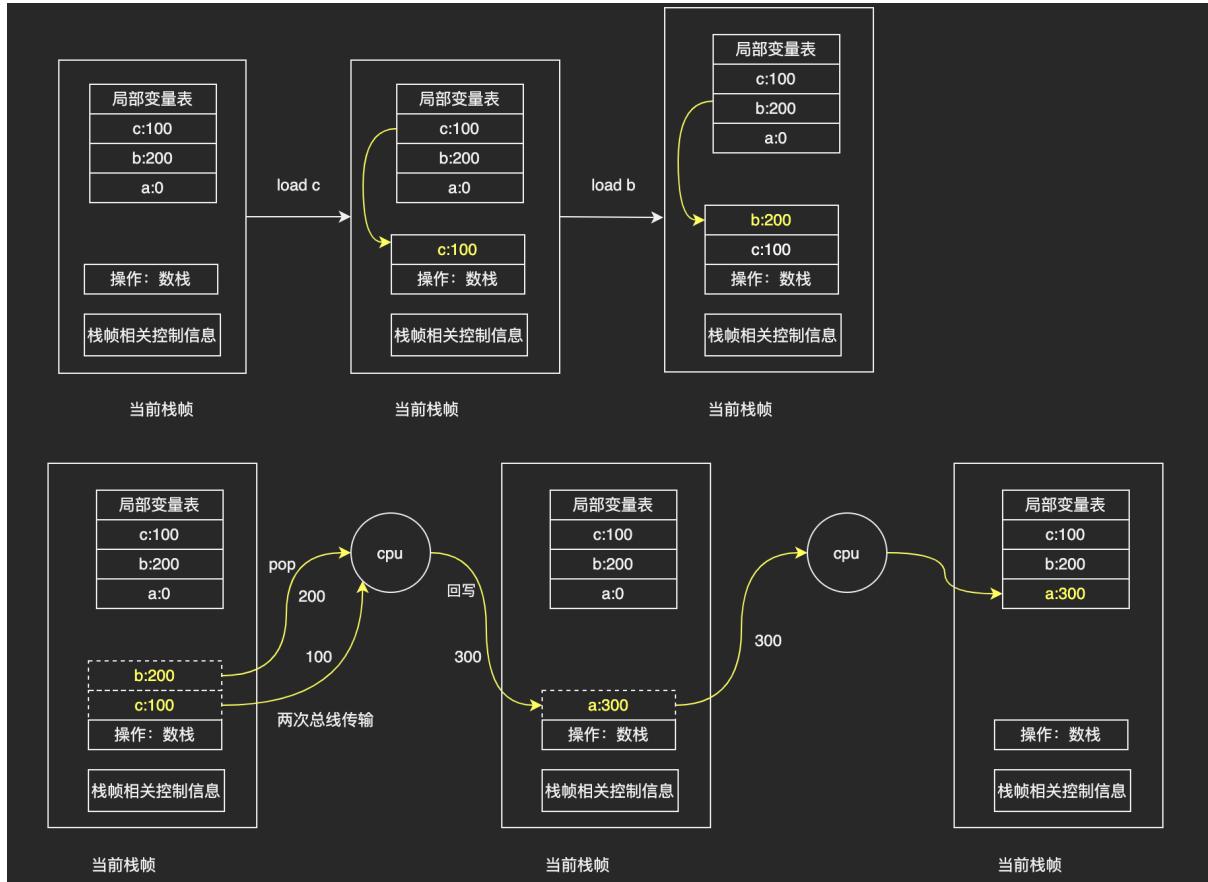
### 虚拟机实现方式：

- 1、基于栈
- 2、基于虚拟寄存器

无论使用哪种机制，都要实现如下几点：

- 1、取指令，其中指令来源于内存
- 2、译码，决定指令类型（执行何种操作），另外译码的过程要包括从内存中取操作数
- 3、执行，指令译码后，被虚拟机执行（其实最终都会借助于物理机资源）
- 4、存储计算结果

### 基于栈执行流程：

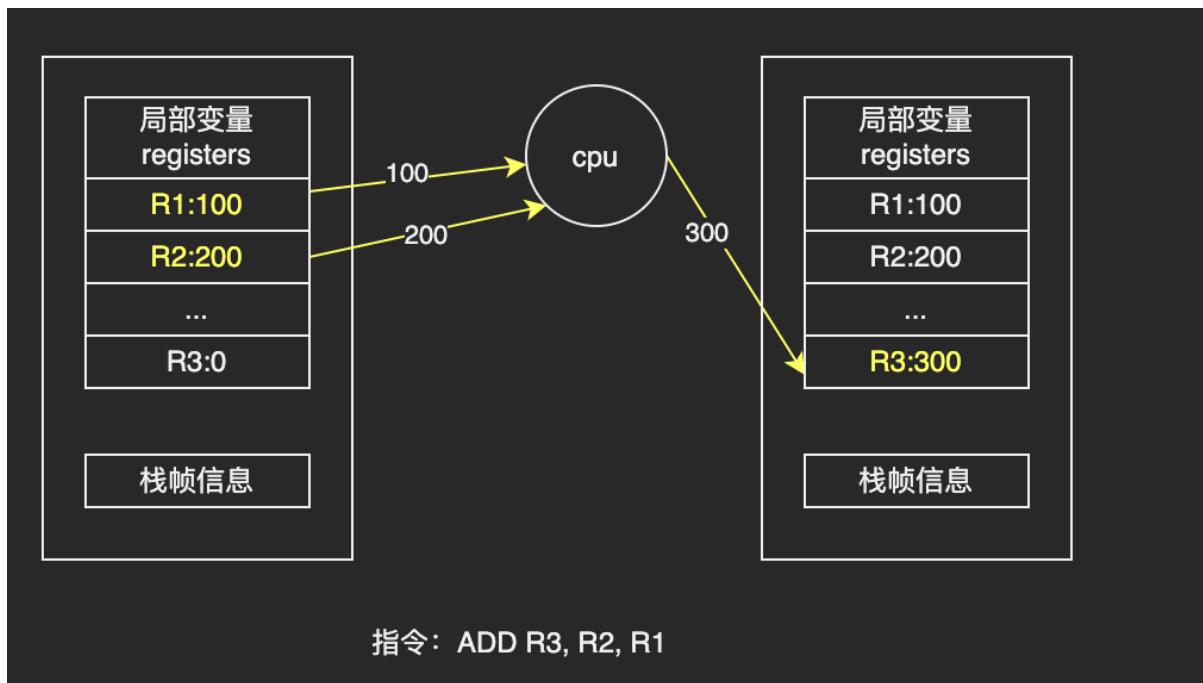


基于栈的虚拟机有一个操作数栈的概念，虚拟机在进行真正的运算时都是直接与操作数栈（operand stack）进行交互，这样做的直接好处就是虚拟机可以无视具体的物理架构，但缺点也显而易见，就是速度慢，因为无论什么操作都要通过操作数栈这一结构（不过值得一提的是实现时通过栈顶缓存（top-of-stack caching）可以大幅降低基于栈的解释器的数据移动开销，可以让这部分开销跟基于寄存器的在同等水平）。

#### 基于寄存器执行流程：

基于寄存器的虚拟机中没有操作数栈的概念，但是有很多虚拟寄存器，一般情况下这些寄存器（操作数）都是别名，需要执行引擎对这些寄存器（操作数）的解析，找出操作数的具体位置，然后取出操作数进行运算。

新的虚拟机也用栈分配活动记录，寄存器就在该活动记录中。当进入Lua程序的函数体时，函数从栈中分配一个足以容纳该函数所有寄存器的活动记录。函数的所有局部变量都各占据一个寄存器。因此，存取局部变量是相当高效的。



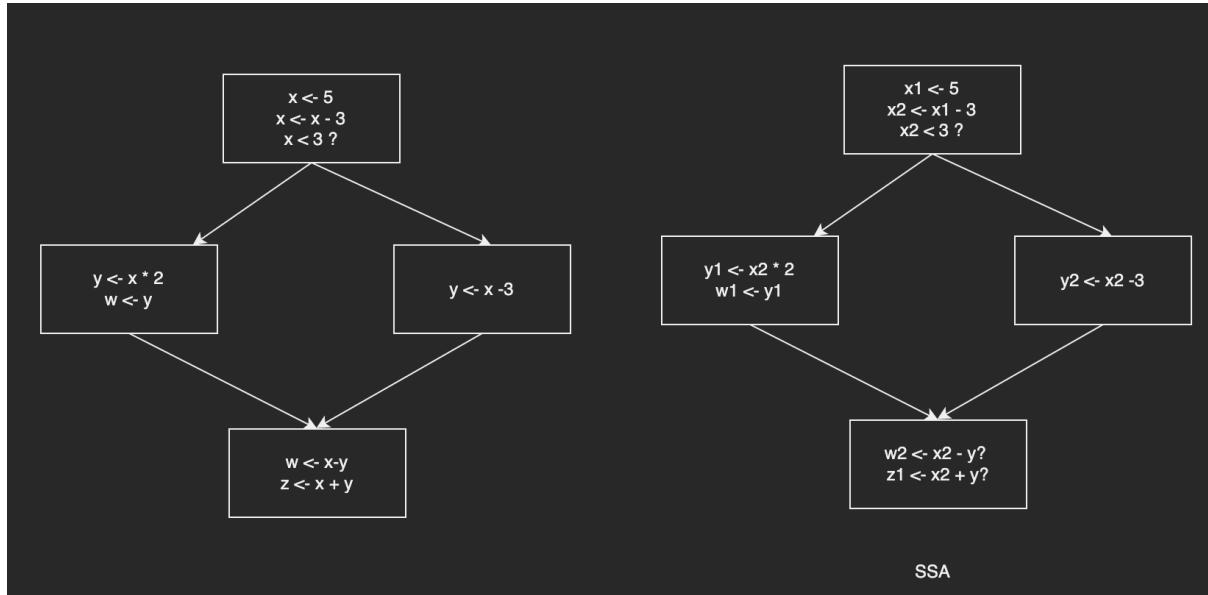
### 栈式虚拟机 VS 寄存器式虚拟机

- 1、指令条数：栈式虚拟机多
- 2、代码尺寸：栈式虚拟机
- 3、移植性：栈式虚拟机移植性更好
- 4、指令优化：寄存器式虚拟机更能优化

三地址码(Three address code, 经常被缩写为TAC或3AC)，一种中间语言，编译器使用它来改进代码转换效率。每个三地址码指令，都可以被分解为一个四元组(4-tuple)：(运算符，操作数1，操作数2，结果)。因为每个陈述都包含了至多三个(如：goto语句，仅含一个变量)变量，所以它被称为三地址码。

### 图IR

将普通代码转换为SSA形式主要是用新变量替换每个赋值的目标，并将变量的每个用法替换为到达该点的变量『版本』。



这样，就很清楚看到变量每次使用所值的定义，除了一种情况：底部块中y的两个使用都可以指y1或y2，这取决于控制流彩的路径。

要解决此问题，会在最后一个块中插入一个特殊语句，称为 $\Phi$ (phi)函数。该语句将通过『选择』y1或y2生成y的新定义，称为y3，具体取决于过去的控制流程。

### 优势边界计算最小SSA

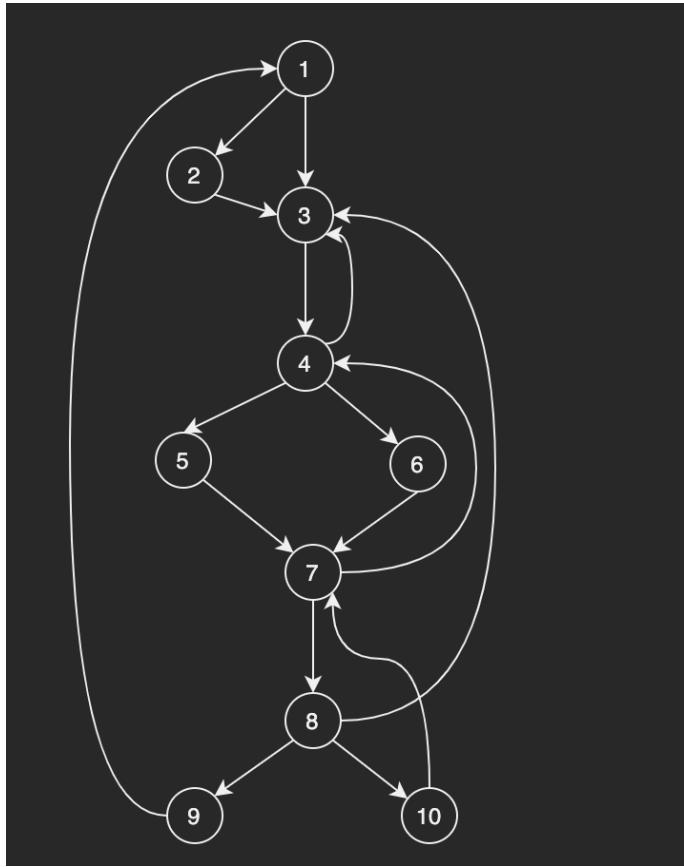
首先，我们需要理解支配者(dominator)的概念：如果不是先通过A就不可能到达B，则节点A严格支配控制流图中的不同节点B。这很有用，因为如果我们到达B，我们就知道A中的任何代码都已运行。如果A严格支配B或A = B，我们说A支配B(B由A支配)。

### 支配节点(dominator)

支配节点分为三种：

- 1、普通支配节点
- 2、严格支配节点
- 4、直接支配节点

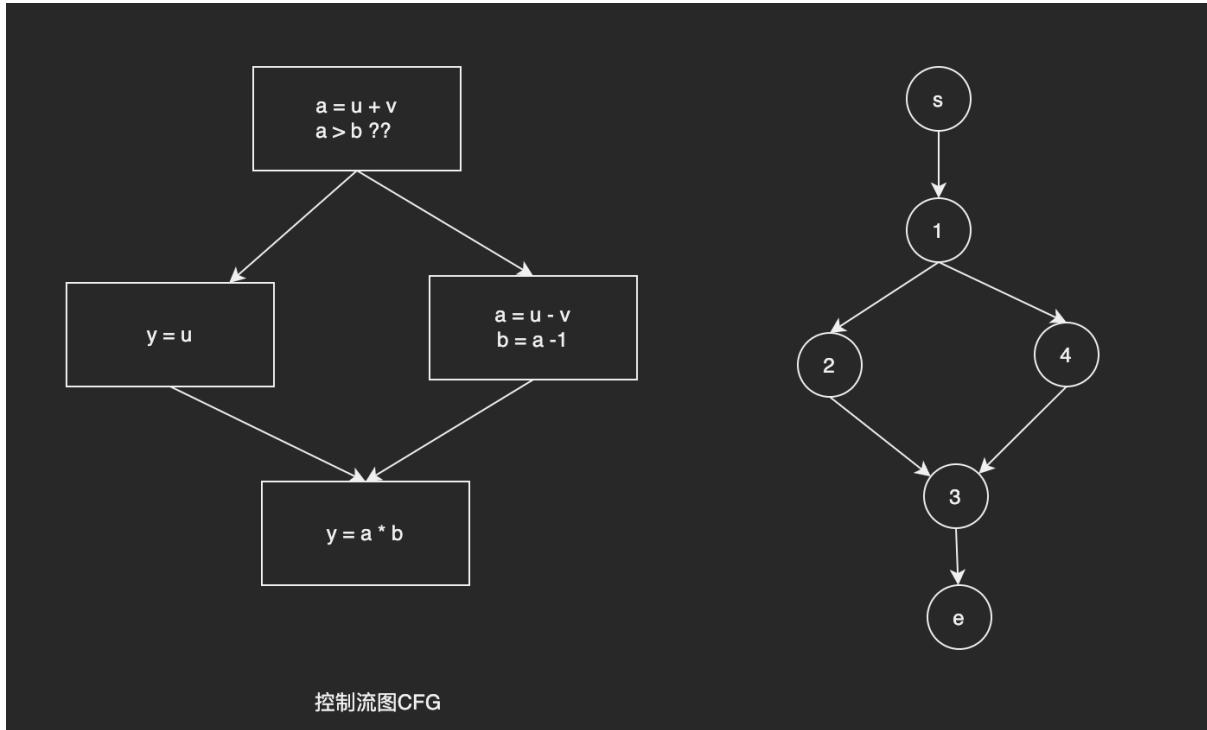
支配节点(dominator)流图中，每一条从入口节点到节点n的路径都经过节点d，则d支配n，记 $dom\ n$ ，每个节点都支配自己。下图中，节点1支配流图所有节点，节点2只支配它自己，因为节点3有多个条件，节点3支配除节点1和节点2以外所有节点。



严格支配节点 (strictly dominator) 在上述支配节点中, 若 $d \neq n$ 并且 $d \text{ dom } n$ , 则 $d$ 严格支配 $n$ , 记 $d \text{ sdom } n$ 。上图中  $1 \text{ sdom } 2, 1 \text{ sdom } 3$ 。

支配节点树 (dominator tree)

使用支配节点树来表示流图的支配信息



如上图1支配自己,2,3,4和e,  $\text{dom}(3) = \{s, 1, 3\}$ , 意味着我们可以找到所有这些中最接近的节点支配节点3, 除了自身。直接支配节点(immediate dominator):最近节点是1,  $\text{idom}(3)=1$ 。

例1:



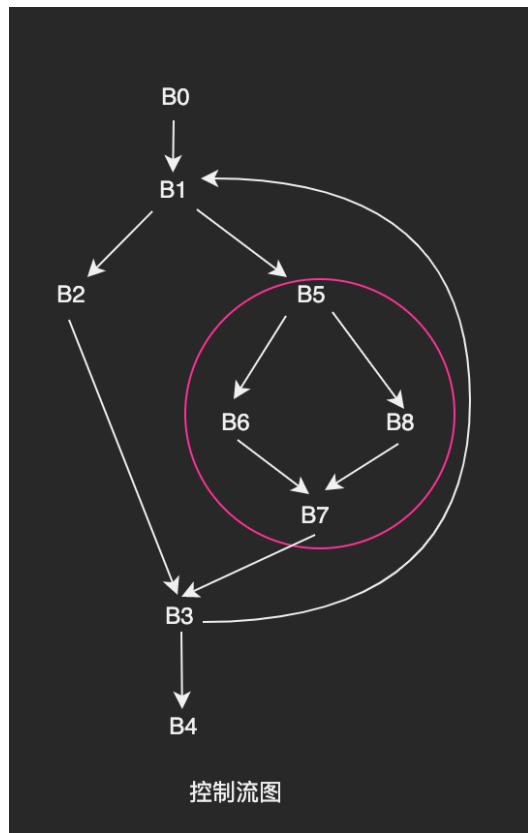
$$\text{dom}(7) = \{0, 1, 2, 6, 7\}$$

$$\text{dom}(4) = \{0, 1, 3, 4\}$$

优势边界(Dominance Frontier)

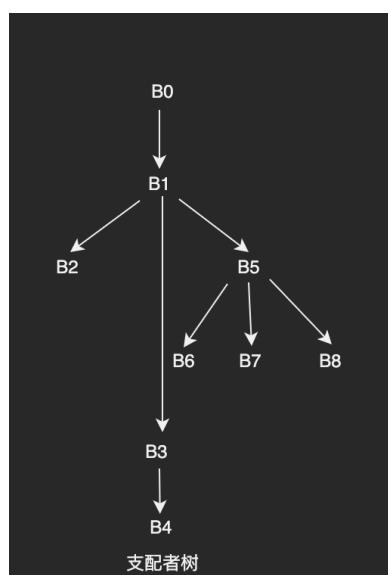
节点x的Dominance Frontier是所有符合下面条件的节点w的集合: x是w的前驱支配节点, 但不是w的严格支配节点。

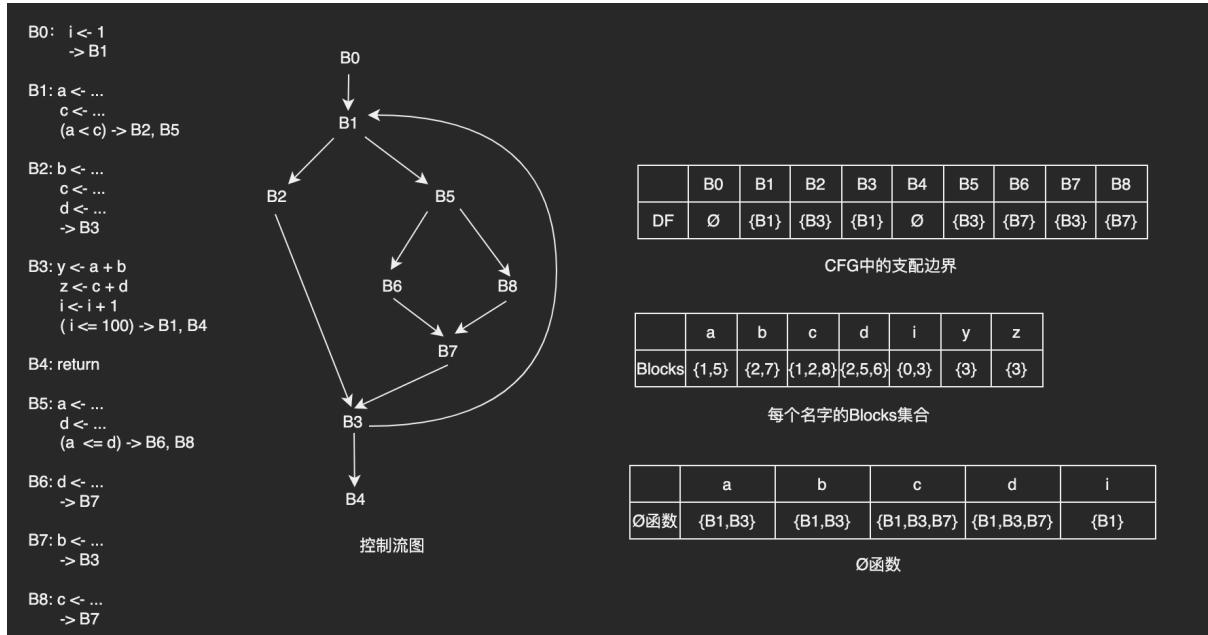
本质上，它是支配节点和非支配节点直接的“分界线”。



$$\begin{aligned} \text{Dom}(5) &= \{5, 6, 7, 8\} \\ \text{DF}(5) &= \{3\} \end{aligned}$$

Ø函数生成





将算法的处理限于全局名字集合，使其避免了对基本程序块B1中的x和y插入“死亡”的Ø函数。但局部名字和全局名字之间的区分不足以避免所有的『死亡』Ø函数。例如，B1中b的Ø函数不是活动的，因为在使用b值之前重新定义了b。为避免插入这些Ø函数，编译器可以构建LiveOut集合，并在插入Ø函数内层循环中增加一个对变量活动性的条件判断，这一改动使用算法可以产生剪枝静态单赋值形式。

不同风格的静态单赋值形式：

1、最小静态单赋值形式(minimal SSA)在任何汇合点处插入一个Ø函数，只要对应于同一原始名字的两个不同定义会合。这将插入符合静态单赋值形式定义、数目最少的Ø函数。但其中一些Ø函数可能是死亡的，定义并没有规定值在会合时一定是活动的。

2、剪枝静态单赋值形式(pruned SSA)向Ø函数插入算法添加一个活动性判断，以避免添加死亡的Ø函数。构造过程必须计算LiveOut集合，因此构建剪枝静态单赋值形式的代价高于构建最小静态单赋值形式。

3、半剪枝静态单赋值形式(semipruned SSA)是最小静态单赋值形式和剪枝静态单赋值形式之间的一种折中。在插入Ø函数之前，算法先删除非跨越基本程序块边界活动的任何名字。这可以缩减名字空间并减少Ø函数的数目，而又没有计算LiveOut集合的开销。

## 执行流程

入口main函数会对中间代码生成进行处理，下面是主函数的执行流程：

```
func Main(archInit func(*Arch)) {
    ...
    // 初始初始化SSA生成配置
    initssaconfig()
    ...
}
```

```

timings.Start("be", "compilefuncs")
fcount = 0
for i := 0; i < len(xtop); i++ {
    n := xtop[i]
    if n.Op == ODCLFUNC {
        funccompile(n)
        fcount++
    }
}
timings.AddEvent(fcount, "funcs")

// 编译函数
compileFunctions()
}

```

## 模板生成中间代码

在ssa/gen/目录下运行go run \*.go, 就会生成对应../rewriteARM64.go文件, 这里会生成对应各种平台架构规则。

比如执行一个两个常量做加法运算, 可以定义如下规则:

(ADD x (MOVDconst [c])) => (ADDconst [c] x)

生成对应用op操作, 生成对应的中间代码

```

func rewriteValueARM64_OpARM64ADD(v *Value) bool {
    v_1 := v.Args[1]
    v_0 := v.Args[0]
    b := v.Block
    typ := &b.Func.Config.Types
    // match: (ADD x (MOVDconst [c]))
    // result: (ADDconst [c] x)
    for {
        for _i0 := 0; _i0 <= 1; _i0, v_0, v_1 = _i0+1, v_1, v_0 {
            x := v_0
            if v_1.Op != OpARM64MOVDconst {
                continue
            }
            c := auxIntToInt64(v_1.AuxInt)
            v.reset(OpARM64ADDconst)
    }
}

```

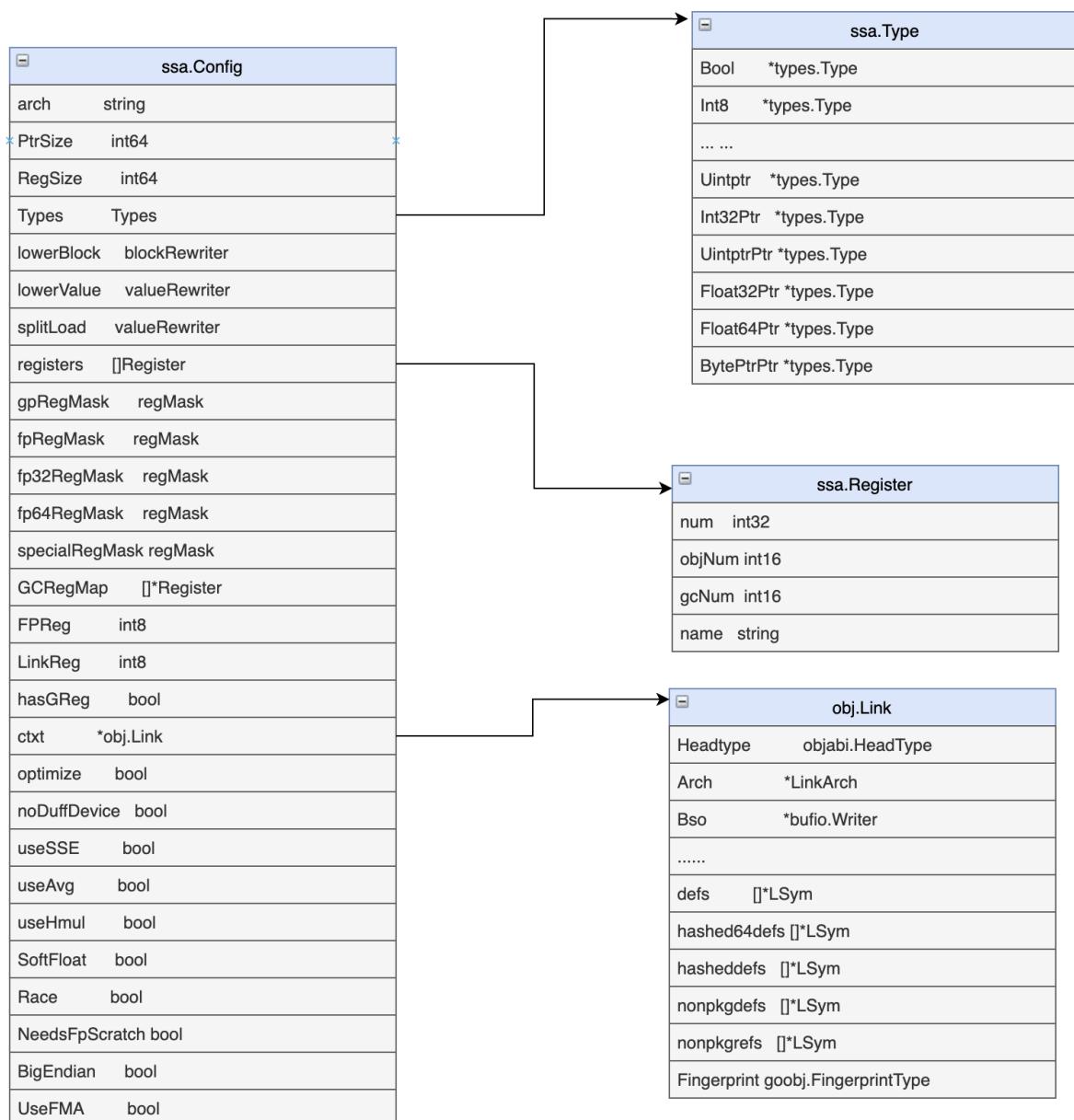
```

        v.AuxInt = int64ToInt(c)
        v.AddArg(x)
        return true
    }
    break
}
}

```

## 配置初始化

### 数据结构



主要会缓存可能用到的类型指针、SSA配置及调用的运行时函数。

```

func initssaconfig() {
    types_ := ssa.NewTypes()

    // 是否使用软件模拟float
    if thearch.SoftFloat {
        softfloatInit()
    }

    // 初始化指针,主要用来优化获取效率
    _ = types.NewPtr(types.Types[TINTER])           // *interface{}
    _ = types.NewPtr(types.NewPtr(types.Types[TSTRING])) // **string
    _ = types.NewPtr(types.NewSlice(types.Types[TINTER])) // *[]interface{}

    _ = types.NewPtr(types.NewPtr(types.Bytetype))     // **byte
    ...

    // ssa config 根据当前CPU架构来初始配置
    ssaConfig = ssa.NewConfig(thearch.LinkArch.Name, *types_, Ctxt,
Debug.N == 0)

    ssaConfig.SoftFloat = thearch.SoftFloat
    ssaConfig.Race = flag_race
    ssaCaches = make([]ssa.Cache, nBackendWorkers)

    // 运行时函数
    assertE2I = sysfunc("assertE2I")
    assertE2I2 = sysfunc("assertE2I2")
    assertI2I = sysfunc("assertI2I")
    assertI2I2 = sysfunc("assertI2I2")
    deferproc = sysfunc("deferproc")
    deferprocStack = sysfunc("deferprocStack")
    ...
}

```

## Walk

walk将语法树中节点一些元素进行替换，替换为真正运行时函数。

```

func walkexpr(n *Node, init *Nodes) *Node {
    if n == nil {
        return n
    }
}

```

```

    }

    ...

opswitch:
    switch n.Op {
        default:
            Dump("walk", n)
            Fatalf("walkexpr: switch 1 unknown op %s", n)

        case ONONAME, OEMPTY, OGETG, ONEWOBJ:
        case OPANIC:
            n = mkcall("gopanic", nil, init, n.Left)

        case ORECOVER:
            n = mkcall("gorecover", n.Type, init, nod(OADDR, nodfp, nil))

        ...
    }
}

```

常见替换函数：

原语法	运行时函数
m[i]	mapaccess
	mapassign
delete(m, k)	mapdelete
<-c	chanrecv1
	chanrecv2
make	makechan makechan64
	makemap makemap64
	makeslice makeslice64
	block

select	selectgo selectsetpc
	selectnbsend selectnbrecv
panic	gopanic
new	newobject
recover	gorecover

## Compile SSA

walk函数处理过后，抽象语法树就不会改变了，compileSSA函数会将抽象语法树转换成中间代码。

可以通过GOSSAFUNC=hello go build hello.go 生成ssa.html文件。

The screenshot shows the SSA compiler interface. On the left, under 'sources' and 'AST', the code for 'buildssa-enter' and 'buildssa-body' is displayed. On the right, under 'start', the generated SSA code is shown, including assembly-like instructions like v1 (?), v2 (?), and v3 (?). The interface has a dark mode toggle and a help link.

```
func compileSSA(fn *Node, worker int) {
    // 生成具有SSA特性中间代码，会经过几十次迭代
    f := buildssa(fn, worker)
```

```

    ...
    pp := newProgs(fn, worker)
    defer pp.Free()
    genssa(f, pp)
    ...
    pp.Flush()
}

func buildssa(fn *Node, worker int) *ssa.Func {
    name := fn.funcname()
    ...
    s.curfn = fn

    s.f = ssa.NewFunc(&fe)
    s.config = ssaConfig
    s.f.Type = fn.Type
    s.f.Config = ssaConfig
    s.f.Cache = &ssaCaches[worker]
    s.f.Cache.Reset()
    s.f.Name = name
    s.f.DebugTest = s.f.DebugHashMatch("GOSSAHASH")
    s.f.PrintOrHtmlSSA = printssa
    ...
    // 这里对应就是上面图需要处理的节点
    s.stmtList(fn.Func.Enter)
    s.stmtList(fn.Nbody)

    // 插入phi函数
    s.insertPhis()

    ...
    // Main call to ssa package to compile function
    ssa.Compile(s.f)
    return s.f
}

```

stmt函数会根据节点操作符的不同将当前AST节点转换对应的中间代码：

```
func (s *state) stmt(n *Node) {
```

```

...
switch n.Op {
case OCALLMETH, OCALLINTER:
    s.call(n, callNormal)
    if n.Op == OCALLFUNC && n.Left.Op == ONAME && n.Left.Class() ==
PFUNC {
        if fn := n.Left.Sym.Name; compiling_runtime && fn == "throw"
        ||
        n.Left.Sym.Pkg == Runtimepkg && (fn == "throwinit" || fn
== "gopanic" || fn == "panicwrap" || fn == "block" || fn ==
"panicmakeslicelen" || fn == "panicmakesliceap") {
            m := s.mem()
            b := s.endBlock()
            b.Kind = ssa.BlockExit
            b.SetControl(m)
        }
    }
    // 生成对应函数SSA节点
    s.call(n.Left, callDefer)
}
case OGO:
    s.call(n.Left, callGo)
...
}
}

func (s *state) callResult(n *Node, k callKind) *ssa.Value {
    return s.call(n, k, false)
}

func (s *state) call(n *Node, k callKind) *ssa.Value {
    ...
    var call *ssa.Value
    switch {
    case k == callDefer:
        // new ssa defer节点函数
        call = s.newValue1A(ssa.OpStaticCall, types.TypeMem, deferproc,
s.mem())
    case k == callGo:

```

```

        call = s.newValue1A(ssa.OpStaticCall, types.TypeMem, newproc,
s.mem())
    case sym != nil:
        call = s.newValue1A(ssa.OpStaticCall, types.TypeMem,
sym.Linksym(), s.mem())
        ..
    }
    ...
}

```

```

// 插入phi函数
func (s *state) insertPhis() {
    // 块的数量小于等于500的时候, 使用简单的phi生成函数
    if len(s.f.Blocks) <= smallBlocks {
        sps := simplePhiState{s: s, f: s.f, defvars: s.defvars}
        // 如果一个块当前不包含变量的定义, 我们递归它的前驱中寻找定义。如果该
        // 块只有一个前驱, 只需递归查询它的定义。
        //否则, 我们收集所有前驱的定义并构造一个φ函数, 将它们连接成一个新值。这
        //个φ 函数被记录为这个基本块中的当前定义
        sps.insertPhis()
        return
    }
    ps := phiState{s: s, f: s.f, defvars: s.defvars}
    // 这个生成函数就是上面讲的通过支配树及优势边界生成的
    ps.insertPhis()
}

```

```

// 编译器优化转换函数列表
var passes = [...]pass{
    // TODO: combine phielim and copyelim into a single pass?
    {name: "number lines", fn: numberLines, required: true},
    {name: "early phielim", fn: phielim},
    {name: "early copyelim", fn: copyelim},
    {name: "early deadcode", fn: deadcode}, // 删除无用代码
    ...
    {name: "stackframe", fn: stackframe, required: true},
    {name: "trim", fn: trim}, // remove empty blocks
}

```

```

func Compile(f *Func) {
    if f.Log() {

```

```

f.Logf("compiling %s\n", f.Name)

}

phaseName := "init"

// 多轮转换
for _, p := range passes {
    f.pass = &p
    p.fn(f)
}

phaseName = ""
}

```

## 例子

file:///Users/shen/go/src/me/compileSSA/ssa.html

sources	AST	
<pre> 3 func Sum(i, j int) int { 4 5     c := 10 + i + j 6     if c &gt; 10 { 7         c = 20 8     } 9     return c 10 } </pre>	<p>before insert phis</p> <pre> <b>start</b>  b1: v1 (?) = InitMem &lt;mem&gt; v2 (?) = SP &lt;uintptr&gt; v3 (?) = SB &lt;uintptr&gt; v4 (?) = LocalAddr &lt;*int&gt; {i} v2 v1 v5 (?) = LocalAddr &lt;*int&gt; {j} v2 v1 v6 (?) = LocalAddr &lt;*int&gt; {~r2} v2 v1 v7 (3) = Arg &lt;int&gt; {i} (i[int]) v8 (3) = Arg &lt;int&gt; {j} (j[int]) v9 (?) = GetClosurePtr &lt;*uint8&gt; v10 (?) = Const64 &lt;int&gt; [0] v11 (?) = Const64 &lt;int&gt; [10] v12 (5) = Add64 &lt;int&gt; v11 v7 v13 (5) = Add64 &lt;int&gt; v12 v8 (c[int]) v14 (6) = Less64 &lt;bool&gt; v11 v13 v15 (?) = Const64 &lt;int&gt; [20] (c[int]) If v14 → b3 b2 (6)  b2: ← b1 b3 v16 (9) = Phi &lt;int&gt; v13 v15 (c[int]) v18 (9) = Copy &lt;mem&gt; v1 v17 (9) = MakeResult &lt;int,mem&gt; v16 v18 Ret v17 (+9)  b3: ← b1 Plain → b2 (9)  name i[int]: v7 name j[int]: v8 name c[int]: v13 v15 v16 </pre>	<p>-</p> <p>-</p>

这里返回值c通过ssa生成过后就没办法确定其值，所以就会生成一个phi函数。

The screenshot shows a debugger interface with two main panes: 'sources' on the left and 'AST' on the right.

**Sources:**

```
/Users/shen/go/src/me/compileSSA/1.go
3 func Sum(i, j int) int {
4
5     c := 10 + i + j
6     if c > 10 {
7         c = 20
8     }
9     return c
10 }
```

**AST:**

before insert phis

**start**

b1:

```
v1 (?) = InitMem <mem>
v2 (?) = SP <uintptr>
v3 (?) = SB <uintptr>
v4 (?) = LocalAddr <*int> {i} v2 v1
v5 (?) = LocalAddr <*int> {j} v2 v1
v6 (?) = LocalAddr <*int> {~r2} v2 v1
v7 (?) = Arg <int> {i} (i[int])
v8 (?) = Arg <int> {j} (j[int])
v9 (?) = GetClosurePtr <*uint8>
v10 (?) = Const64 <int> [0]
v11 (?) = Const64 <int> [10]
v12 (5) = Add64 <int> v11 v7
v13 (5) = Add64 <int> v12 v8 (c[int])
v14 (6) = Less64 <bool> v11 v13
v15 (?) = Const64 <int> [20] (c[int])
If v14 → b3 b2 (6)
```

b2: ← b1 b3

```
v16 (9) = Phi <int> v13 v15 (c[int])
v18 (9) = Copy <mem> v1
v17 (9) = MakeResult <int,mem> v16 v18
Ret v17 (+9)
```

b3: ← b1

Plain → b2 (9)

name i[int]: v7  
name j[int]: v8  
name c[int]: v13 v15 v16

**v14 (6) = Less64 <bool> v11 v13**

v14 SSA生成时调试代码如下：

```

✓ [13]: *(cmd/compile/internal/ssa.Value)(0xc000780620)
  ✓ : cmd/compile/internal/ssa.Value {ID: 14, Op: OpLess64 (2350), Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 1, method.
    ID: 14
    Op: OpLess64 (2350)
  > Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 1, methods: (*cmd/compile/internal/types.Fields)(0xc0003e4ef8), allMe
    AuxInt: 0
    Aux: cmd/compile/internal/ssa.Aux nil
  ✓ Args: []*cmd/compile/internal/ssa.Value len: 2, cap: 3, [*(*cmd/compile/internal/ssa.Value)(0xc0007804d0), *(*cmd/compile/internal/ssa.Value)"
  ✓ [0]: *(cmd/compile/internal/ssa.Value)(0xc0007804d0)
    ✓ : cmd/compile/internal/ssa.Value {ID: 11, Op: OpConst64 (2425), Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, m
      ID: 11
      Op: OpConst64 (2425)
    > Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, methods: (*cmd/compile/internal/types.Fields)(0xc0003e4fd8), a
      AuxInt: 10
      Aux: cmd/compile/internal/ssa.Aux nil
      Args: []*cmd/compile/internal/ssa.Value len: 0, cap: 3, []
    > Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*cmd/internal/src.XPos)(0xc0007b6c34), Kind: BlockIf (119), Likely: BranchUnknown (0
    > Pos: cmd/internal/src.XPos {index: 0, lico: 0}
      Uses: 2
      OnWasmStack: false
      InCache: true
    > argstorage: [3]*cmd/compile/internal/ssa.Value [*nil,*nil,*nil]
  ✓ [1]: *(cmd/compile/internal/ssa.Value)(0xc0007805b0)
    ✓ : cmd/compile/internal/ssa.Value {ID: 13, Op: OpAdd64 (2220), Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, me
      ID: 13
      Op: OpAdd64 (2220)
    > Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, methods: (*cmd/compile/internal/types.Fields)(0xc0003e4fd8), a
      AuxInt: 0
      Aux: cmd/compile/internal/ssa.Aux nil
    > Args: []*cmd/compile/internal/ssa.Value len: 2, cap: 3, [*(*cmd/compile/internal/ssa.Value)(0xc000780540), *(*cmd/compile/internal/ssa.Value"
    > Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*cmd/internal/src.XPos)(0xc0007b6c34), Kind: BlockIf (119), Likely: BranchUnknown (0
    > Pos: cmd/internal/src.XPos {index: 2, lico: 28896}
      Uses: 1
      OnWasmStack: false
      InCache: false
    > argstorage: [3]*cmd/compile/internal/ssa.Value [*(*cmd/compile/internal/ssa.Value)(0xc000780540), *(*cmd/compile/internal/ssa.Value)(0xc
  > Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*cmd/internal/src.XPos)(0xc0007b6c34), Kind: BlockIf (119), Likely: BranchUnknown (0), F
  > Pos: cmd/internal/src.XPos {index: 2, lico: 32880}
    Uses: 1

```

## 中间代码优化

优化函数列表：

名称	说明
numberLines	记录源文件和ssa文件行号的对应关系及其实行的统计
phielim	优化phi函数重复参数
copyelim	从 f 中移除所有对 OpCopy 值的使用。需要后续的死代码传递才能实际删除副本
deadcode	删除生成的死代码
opt	优化一些通用规则，比如

	rewriteBlockgeneric, rewriteValuegeneric
zcse	对具有零参数的值的函数进行公共子表达式消除
cse	执行公共子表达式消除
phiopt	根据前面的 if 消除 boolean Phis
nilcheckelim	消除不必要的 nil 检查
prove	删除了冗余的 BlockIf 分支, 这些分支可以从之前的主要比较中推断出来
decomposeBuiltIn	分解将复合内置类型上的 phi 操作转换为简单类型上的 phi 操作, 然后调用重写规则来分解这些类型上的其他操作
expandCalls	将 LE(后期扩展)调用转换为更面向平台 ABI 的低级形式, 就像它们接收值 args 一样
softfloat	使用softfloat进行编译
elimDeadAutosGeneric	死代码自动删除
branchelim	消除分支生成CondSelect 指令
writebarrier	在必要时为存储操作(Store、Move、Zero)插入写屏障。它将存储操作重写为分支和运行时调用
insertLoopReschedChecks	在循环中插入重新安排的检查
addressingModes	将地址计算结合到内存操作中, 可以执行复杂的寻址模式
elimUnreadAutos	删除存储(以及关联的操作 VarDef 和 VarKill )到从未读取过的汽车
tightenTupleSelectors	确保元组选择器(Select0、Select1 和 SelectN ops)与其元组生成器位于同一块中。该函数还确保没有重复的元组选择器
tighten	将值移动到更靠近使用它们的块。这可以减少所需的寄存器溢出量, 如果它还没有创建更多的实时值。一个 Value 可以移动到支配所有使用它的块的任何块。
critical	拆分临界边(那些从具有多个外边的块到具有多个内边的块的边)。Regalloc 想要一个无临界边的 CFG, 以便它可以实现 phi 值。
likelyadjust	分支预测下一步要执行动作, 比如{"default", "call", "ret", "exit"}

layout	对 f 中的基本块进行排序，以最小化控制流指令为目标。此阶段返回后，f.Blocks 的顺序很重要，并且是这些块将出现在汇编输出中的顺序。
schedule	安排每个块中的值。此阶段返回后，b.Values 的顺序很重要，并且是这些值在汇编输出中出现的顺序。现在它使用优先级队列生成一个合理有效的调度
flagalloc	在所有生成标志的指令中分配标志寄存器。如果需要溢出/恢复标志值，则重新计算标志值。
regalloc	对 f 执行寄存器分配。它将 f.RegAlloc 设置为结果分配
loopRotate	将开始时带有检查循环条件的循环转换为结束时带有检查循环条件的循环。这有助于循环避免额外的不必要的跳跃。
stackframe	回调到前端以分配帧偏移量
trim	移除空的块

## 优化方法

- 删除公共子表达式
- 复制传播
- 删除无用代码
- 常量合并
- 代码移动
- 强度削弱
- 删除归纳变量

## 删除无用代码

1、复制传播：在复制语句之后尽可能地用代替，也就是找到原始值

b = c a[1] = d a[2] = b	b = c a[1] = t5 a[2] = <b>c</b>
-------------------------------	---------------------------------------

## 2. 基于基本块的DAG删除无用代码

从一个DAG上删除所有没有附加活跃变量的根节点(即没有父节点的节点)。重复应用这样的处理过程就可以从DAG中消除所有对应于无用代码的节点。

代码

```
func deadcode(f *Func) {

    // 找到可达块
    reachable := ReachableBlocks(f)

    // 删除死代码, 针对死代码边
    for _, b := range f.Blocks {
        if reachable[b.ID] {
            continue
        }
        for i := 0; i < len(b.Succs); {
            e := b.Succs[i]
            if reachable[e.b.ID] {
                b.removeEdge(i)
            } else {
                i++
            }
        }
    }

    // 删除死代码, 活代码边
    for _, b := range f.Blocks {
        // 不可达的跳出循环
        if !reachable[b.ID] {
            continue
        }
        // 块不是第一个跳出循环
        if b.Kind != BlockFirst {
            continue
        }
        // 块是根结点, 并可达的
        b.removeEdge(1)
        b.Kind = BlockPlain
        b.Likely = BranchUnknown
    }

    // 复制传播
    copyelim(f)

    // 查找活跃变量
}
```

```
live, order := liveValues(f, reachable)
defer f.retDeadcodeLive(live)
defer f.retDeadcodeLiveOrderStmts(order)

// 初始化一个稀疏集，映射死代码的和重复的条目
s := f.newSparseSet(f.NumValues())
defer f.retSparseSet(s)
i := 0
for _, name := range f.Names {
    .....
}
clearNames := f.Names[i:]
for j := range clearNames {
    clearNames[j] = nil
}
f.Names = f.Names[:i]
.....
//任何未能匹配活跃变量的边界都可以移动到块末尾
pendingLines.foreachEntry(func(j int32, l uint, bi int32) {
    b := f.Blocks[bi]
    if b.Pos.Line() == l && b.Pos.FileIndex() == j {
        b.Pos = b.Pos.WithIsStmt()
    }
})

//从块的值列表中删除死值。
for _, b := range f.Blocks {
    i := 0
    for _, v := range b.Values {
        if live[v.ID] {
            b.Values[i] = v
            i++
        } else {
            f.freeValue(v)
        }
    }
    b.truncateValues(i)
}

//从 WBLoads 列表中移除死块。
i = 0
for _, b := range f.WBLoads {
    if reachable[b.ID] {
        f.WBLoads[i] = b
    }
}
```

```

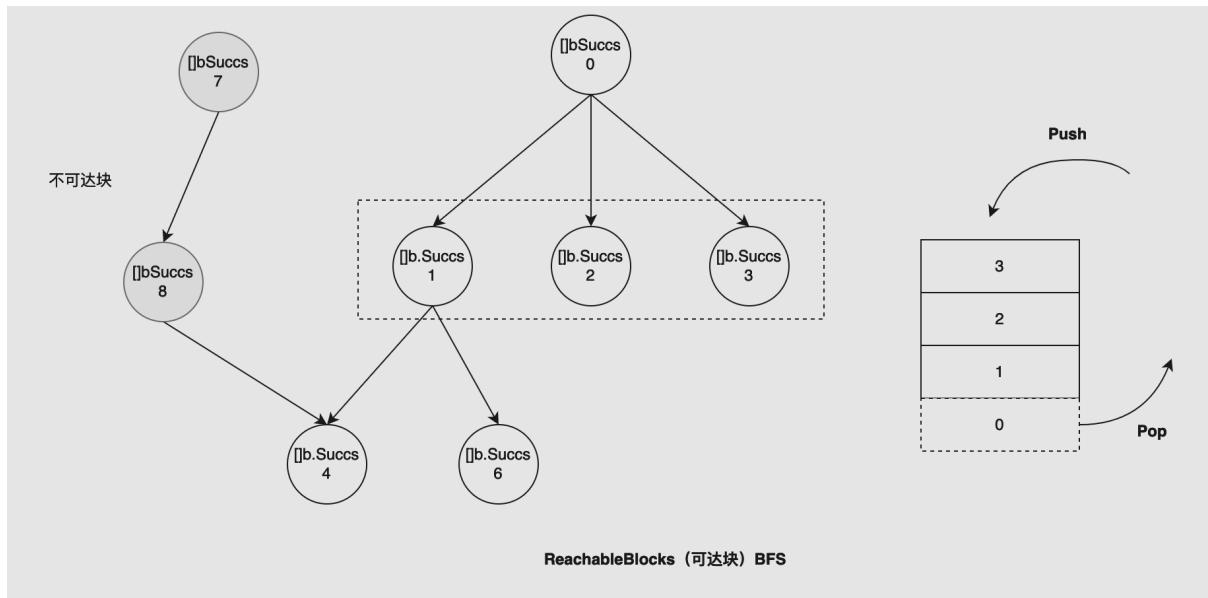
        i++
    }
}
clearWBLoads := f.WBLoads[i:]
for j := range clearWBLoads {
    clearWBLoads[j] = nil
}
f.WBLoads = f.WBLoads[:i]

//删除不可达块。将死块返回给分配器。
i = 0
for _, b := range f.Blocks {
    if reachable[b.ID] {
        f.Blocks[i] = b
        i++
    } else {
        if len(b.Values) > 0 {
            b.Fatalf("live values in unreachable block %v:
%v", b, b.Values)
        }
        f.freeBlock(b)
    }
}

//零值赋值, 帮助GC
tail := f.Blocks[i:]
for j := range tail {
    tail[j] = nil
}
f.Blocks = f.Blocks[:i]
}

```

可达块查找主要使用BFS搜索算法来实现，将不可达块进行删除



## 公共子表达式消除

1、根据两个值等价定义规则，进行分区

- equivalent(v, w):
- v.op == w.op
- v.type == w.type
- v.aux == w.aux
- v.auxint == w.auxint
- len(v.args) == len(w.args)
- v.block == w.block if v.op == OpPhi
- equivalent(v.args[i], w.args[i]) for i in 0..len(v.args)-1

2、将等价类拆分为具有非等价参数点，进一步细化，直接找不到分裂点

3、构建支配树

4、找到v 和 w 在同一个等价类中并且 v 支配 w

5、替换Arg和controlValues

代码：

```
//cse 对函数执行公共子表达式消除。
func cse(f *Func) {
    //使用上述条件的子集进行初始组分区。
    a := make([]*Value, 0, f.NumValues())
    if f.auxmap == nil {
        f.auxmap = auxmap{}
    }
    for _, b := range f.Blocks {
```

```

        for _, v := range b.Values {
            if v.Type.IsMemory() {
                //内存值永远不会 cse
                continue
            }
            if f.auxmap[v.Aux] == 0 {
                f.auxmap[v.Aux] = int32(len(f.auxmap)) + 1
            }
            a = append(a, v)
        }
    }
partition := partitionValues(a, f.auxmap)

// 值相等类, 用于ID映射的
valueEqClass := make([]ID, f.NumValues())
for _, b := range f.Blocks {
    for _, v := range b.Values {
        valueEqClass[v.ID] = -v.ID
    }
}
var pNum ID = 1
for _, e := range partition {
    for _, v := range e {
        valueEqClass[v.ID] = pNum
    }
    pNum++
}

```

//将等价类拆分为具有非等价参数点, 进一步细化, 直接找不到分裂点

```

var splitPoints []int
// 可重用 partitionBy regClass 以减少分配
byArgClass := new(partitionByArgClass)
for {
    changed := false
    .....
}

```

// 构建支配树

```
sdom := f.Sdom()
```

//计算我们想要做的替换。我们用 v 代替 w

//如果 v 和 w 在同一个等价类中并且 v 支配 w。

```
rewrite := make([]*Value, f.NumValues())
byDom := new(partitionByDom)
```

.....

```

rewrites := int64(0)

//应用替换
for _, b := range f.Blocks {
    for _, v := range b.Values {
        for i, w := range v.Args {
            if x := rewrite[w.ID]; x != nil {
                .....
                v.SetArg(i, x)
                rewrites++
            }
        }
    }
    for i, v := range b.ControlValues() {
        if x := rewrite[v.ID]; x != nil {
            b.ReplaceControl(i, x)
        }
    }
}
}

```

我们看一下分区和相等值对象数据存储：

```

func Sum(i, j int) int{

    c := 10 + i + j
    if c > 10 {
        c = 20
    }
    return c
}

```

partition主要是根据操作类型及参数长度等进行分区的，这里会将  $10+i$  和  $i+j$  放到同一个分区里。

```
✓ partition: []cmd/compile/internal/ssa.eqclass len: 1, cap: 1,..  
✓ [0]: cmd/compile/internal/ssa.eqclass len: 2, cap: 19, [*(*...  
✓ [0]: (*cmd/compile/internal/ssa.Value)(0xc0003ea5b0)  
✓ : cmd/compile/internal/ssa.Value {ID: 13, Op: OpAdd64 (22...  
  ID: 13  
  Op: OpAdd64 (2220)  
> Type: *cmd/compile/internal/types.Type {Extra: interface...  
  AuxInt: 0  
  Aux: cmd/compile/internal/ssa.Aux nil  
> Args: []*cmd/compile/internal/ssa.Value len: 2, cap: 3, ..  
> Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*"c...  
> Pos: cmd/internal/src.XPos {index: 2, lico: 28897}  
  Uses: 2  
  OnWasmStack: false  
  InCache: false  
> argstorage: [3]*cmd/compile/internal/ssa.Value [*(*cmd/..  
✓ [1]: (*cmd/compile/internal/ssa.Value)(0xc0003ea7e0)  
✓ : cmd/compile/internal/ssa.Value {ID: 18, Op: OpAdd64 (2...  
  ID: 18  
  Op: OpAdd64 (2220)  
> Type: *cmd/compile/internal/types.Type {Extra: interface...  
  AuxInt: 0  
  Aux: cmd/compile/internal/ssa.Aux nil  
> Args: []*cmd/compile/internal/ssa.Value len: 2, cap: 3, ..  
> Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*"c...  
> Pos: cmd/internal/src.XPos {index: 2, lico: 28896}  
  Uses: 1  
  OnWasmStack: false  
  InCache: false  
> argstorage: [3]*cmd/compile/internal/ssa.Value [*(*cmd/..
```

valueEqClass 对应值对象相同特性对应一个ID，这里 13 和 18 对应同一个组，给他分配一个相同 ID。

```
✓ valueEqClass: []cmd/compile/internal/ssa.ID len: 19, cap:  
[0]: 0  
[1]: -1  
[2]: 0  
[3]: 0  
[4]: 0  
[5]: 0  
[6]: 0  
[7]: -7  
[8]: -8  
[9]: 0  
[10]: 0  
[11]: -11  
[12]: 0  
[13]: 1  
[14]: -14  
[15]: -15  
[16]: -16  
[17]: -17  
[18]: 1
```

## 常量合并

将带有常量的表达式进行常量合并进行，GO列举了200种转换规则，这里介绍几个规则的实现方式。

```
// 根据当前Value的操作类型进行匹配
func rewriteValuegeneric(v *Value) bool {
    switch v.Op {
        case OpAdd16:
            return rewriteValuegeneric_OpAdd16(v)
        case OpAdd32:
            return rewriteValuegeneric_OpAdd32(v)
        .....
        case OpCom64:
            return rewriteValuegeneric_OpCom64(v)
        case OpCom8:
            return rewriteValuegeneric_OpCom8(v)
        .....
    }

    func rewriteValuegeneric_OpAdd16(v *Value) bool {
        v_1 := v.Args[1]
        v_0 := v.Args[0]
        b := v.Block
        // match: (Add16 (Const16 [c]) (Const16 [d]))  c+d
        // result: (Const16 [c+d])
        for {
            for _i0 := 0; _i0 <= 1; _i0, v_0, v_1 = _i0+1, v_1, v_0 {
                if v_0.Op != OpConst16 {
                    continue
                }
                c := auxIntToInt16(v_0.AuxInt)
                if v_1.Op != OpConst16 {
                    continue
                }
                d := auxIntToInt16(v_1.AuxInt)
                v.reset(OpConst16)
                v.AuxInt = int16ToAuxInt(c + d)
                return true
            }
            break
        }
        // match: (Add16 <t> (Mul16 x y) (Mul16 x z))
        // result: (Mul16 x (Add16 <t> y z))
        for {
            t := v.Type
            for _i0 := 0; _i0 <= 1; _i0, v_0, v_1 = _i0+1, v_1, v_0 {
                if v_0.Op != OpMul16 {
                    continue
                }
                _ = v_0.Args[1]
                v_0_0 := v_0.Args[0]
```

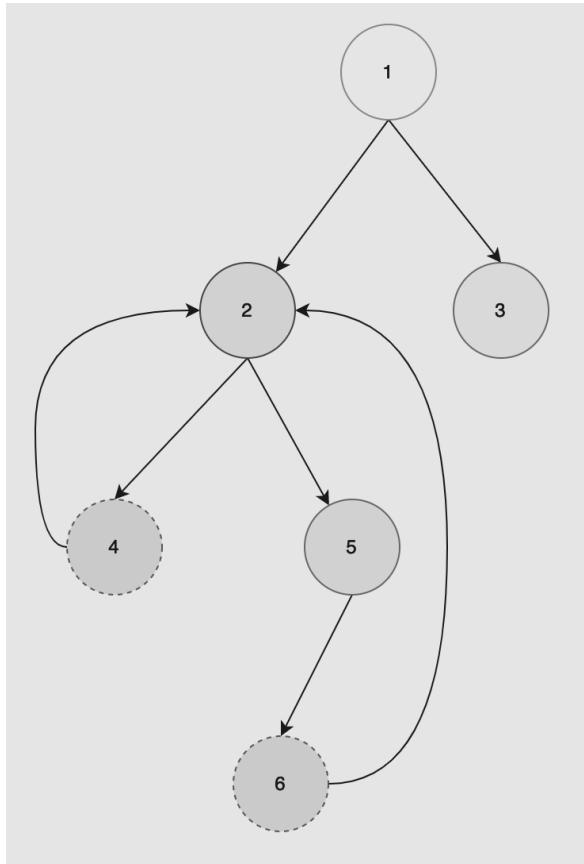
```

v_0_1 := v_0.Args[1]
for _i1 := 0; _i1 <= 1; _i1, v_0_0, v_0_1 = _i1+1, v_0_1, v_0_0 {
    x := v_0_0
    y := v_0_1
    if v_1.Op != OpMul16 {
        continue
    }
    _ = v_1.Args[1]
    v_1_0 := v_1.Args[0]
    v_1_1 := v_1.Args[1]
    for _i2 := 0; _i2 <= 1; _i2, v_1_0, v_1_1 = _i2+1, v_1_1, v_1_0 {
        if x != v_1_0 {
            continue
        }
        z := v_1_1
        v.reset(OpMul16)
        v0 := b.NewValue0(v.Pos, OpAdd16, t)
        v0.AddArg2(y, z)
        v.AddArg2(x, v0)
        return true
    }
}
break
}
}

```

## 代码移动

代码移动需要预先计算出最近的公共祖先，一般使用LCA算法，下面图中4和6节点的最近公共祖先就是2。



如何快速找到最近公共祖先节点，这里引入ST(Sparse Table Algorithm) 算法，中文一般管 Sparse Table 称作稀疏表。原理是基于二进制的倍增、动态规划思想。

### 1. 通过欧拉回路的值来进行最近公共祖先节点

[https://en.wikipedia.org/wiki/Euler\\_tour\\_technique#/media/File:Stirling\\_permutation\\_Euler\\_tour.svg](https://en.wikipedia.org/wiki/Euler_tour_technique#/media/File:Stirling_permutation_Euler_tour.svg)

```

func makeLCArange(f *Func) *lcaRange {
    // 获取支配树
    dom := f.Idom()

    // Build tree
    blocks := make([]lcaRangeBlock, f.NumBlocks())
    for _, b := range f.Blocks {
        blocks[b.ID].b = b
        if dom[b.ID] == nil {
            continue // 块入口或不可达
        }
        // 获取直接支配点, 用作父节点
        parent := dom[b.ID].ID
        blocks[b.ID].parent = parent
        blocks[b.ID].sibling = blocks[parent].firstChild
        blocks[parent].firstChild = b.ID
    }
}

```

```

//计算欧拉回路的排序。
//每个可达块将在回路中出现#children+1 次。
tour := make([]ID, 0, f.NumBlocks()*2-1)
type queueEntry struct {
    bid ID // 块ID 继续工作
    cid ID // 我们已经在处理的孩子(0 = 还没有开始)
}
q := []queueEntry{{f.Entry.ID, 0}}
for len(q) > 0 {

    // 队列里弹出一个
    n := len(q) - 1
    bid := q[n].bid
    cid := q[n].cid
    q = q[:n]
}

```

//将块添加到回路中。  
 /\*\*Euler Tour Tree(欧拉回路树, 后文简称 ETT)是一种可以解决 动态树 问题的数据结构。ETT

将动态树的操作转换成了其 DFS 序列上的区间操作, 再用其他数据结构来维护序列的区间操作, 从而维护动态树的操作。例如,

ETT 将动态树的加边操作转换成了多个序列拆分操作和序列合并操作, 如果能维护序列拆分操作和序列合并操作, 就能维护动态树的加边操作。

```

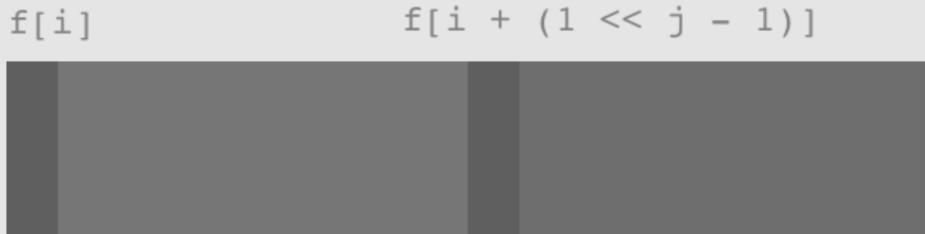
**/
blocks[bid].pos = int32(len(tour))
tour = append(tour, bid)

//继续下一个子边(如果有的话 )。
if cid == 0 {
    //这是我们第一次访问 b, 设置它的深度。
    blocks[bid].depth = blocks[blocks[bid].parent].depth + 1
    //然后探索它的第一个孩子。
    cid = blocks[bid].firstChild
} else {
    //我们之前已经处理过, 探索下一个孩子。
    cid = blocks[cid].sibling
}
if cid != 0 {
    q = append(q, queueEntry{bid, cid}, queueEntry{cid, 0})
}
}
.....
```

## 1. 预处理

ST 算法原理上还是动态规划，我们用  $a(1 \dots n)$  表示待查询的一组数，设  $f(i, j)$  表示从  $a(i)$  到  $a(i + 2^j - 1)$  这个范围内的最大值。也就是说  $a[i]$  为起点，连续  $2^j$  个数的最大值。由于元素个数为  $2^j$  个，所以从中间平均分成两部分，每一部分元素个数刚好为  $2^{j-1}$  个，也就是说，把  $f(i, j)$  划分成  $f(i, j - 1)$  和  $f(i + 2^{j-1}, j - 1)$ 。

我画个图来描述一下这个场景：



整个区间的最大值一定是左右两部分最大值的较大值，满足动态规划的最优化原理（子状态影响父状态）。很显而易见的状态转移方程：

$$f(i, j) = \max(f(i, j - 1), f(i + 2^{j-1}, j - 1))$$

```
func makeLCArange(f *Func) *lcaRange {
    .....
    // 计算快速范围-最小查询数据结构
    rangeMin := make([][]ID, 0, bits.Len64(uint64(len(tour))))
    rangeMin = append(rangeMin, tour)
    // fa[i][j]是指节点 i 的 2^j 级的祖先的编号
    // 主要做一些预处理工作, 采用倍增算法
    for logS, s := 1, 2; s < len(tour); logS, s = logS+1, s*2 {
        r := make([][]ID, len(tour)-s+1)
        for i := 0; i < len(tour)-s+1; i++ {
            bid := rangeMin[logS-1][i]
            bid2 := rangeMin[logS-1][i+s/2]
            if blocks[bid2].depth < blocks[bid].depth {
                bid = bid2
            }
            r[i] = bid
        }
        rangeMin = append(rangeMin, r)
    }

    return &lcaRange{blocks: blocks, rangeMin: rangeMin}
}
```

## 查询

我们继续思考区间最大值问题，假设我要查询  $[l, r]$  这个区间，那么我们如果找到两个子区间，他们的并集精确覆盖到  $[l, r]$  是不是就满足要求了

```
f[i, i + (1 << (x + 1) - 1)]
```



```
f[j - 1 << (1 << (x + 1) + 1), j]
```

两个集合的并集，恰好是  $[i, j]$

```
func (lca *lcaRange) find(a, b *Block) *Block {
    if a == b {
        return a
    }

    //找到 a 和 b 欧拉回路的位置。
    p1 := lca.blocks[a.ID].pos
    p2 := lca.blocks[b.ID].pos
    if p1 > p2 {
        p1, p2 = p2, p1
    }
}
```

// 最低的共同祖先是从p1到p2的最小深度块。我们已经预先计算了遍历的2次幂子序列的  
最小深度块，将正确的两个预先计算的值结合起来得到答案。

```
logS := uint(log64(int64(p2 - p1)))
bid1 := lca.rangeMin[logS][p1]
bid2 := lca.rangeMin[logS][p2-1<<logS+1]
if lca.blocks[bid1].depth < lca.blocks[bid2].depth {
    return lca.blocks[bid1].b
}
return lca.blocks[bid2].b
}
```

## 活跃变量分析

活跃变量定义：对于变量x和程序点p，如果在流图中沿着p开始的某条路径会引用变量x在p点的值，则称变量x在点p是活跃(live)的。否则称变量x在点p不活跃(dead)

活跃变量信息的主要用途：

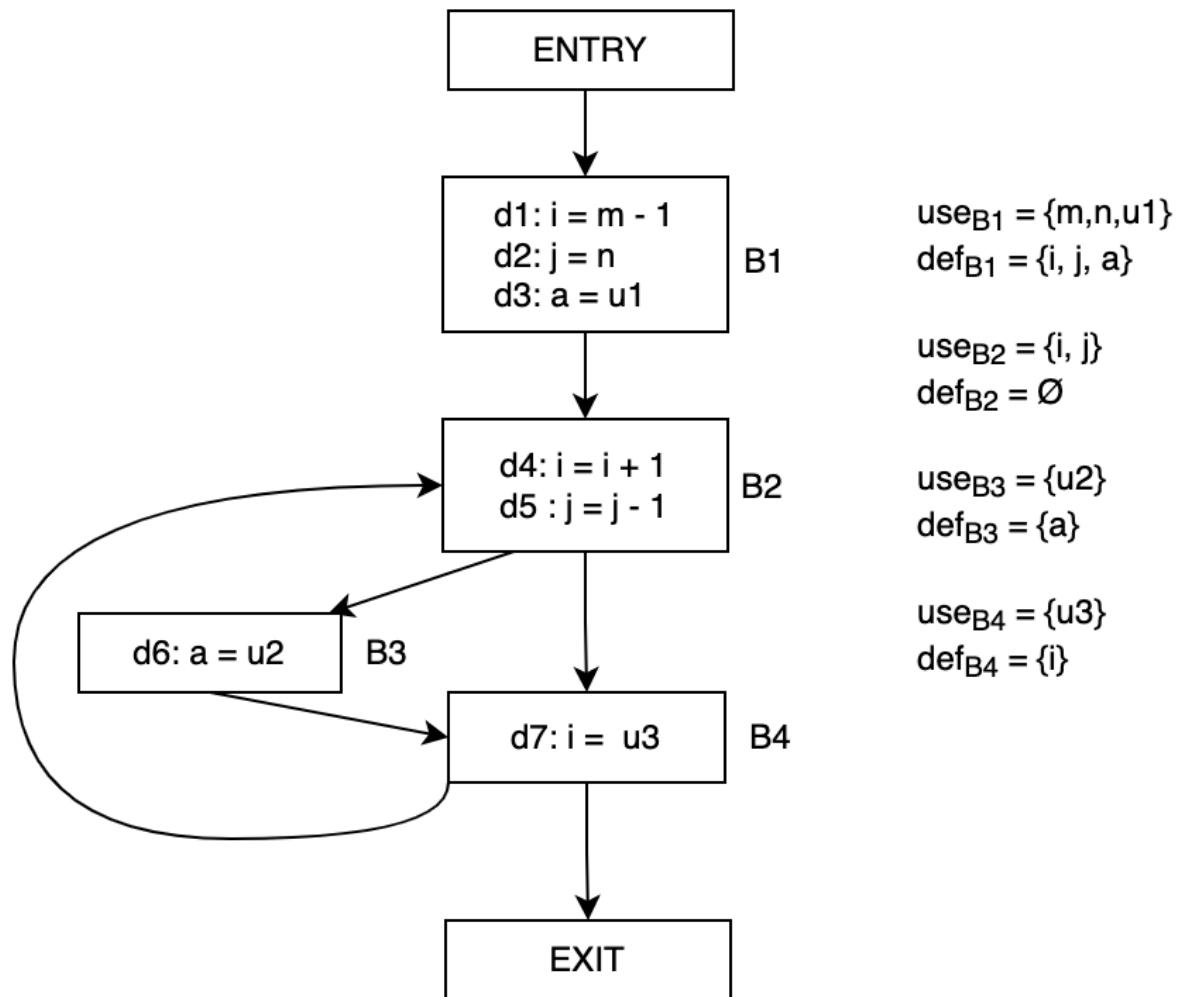
1. 删除无用赋值，也就是说定义了变量，后面没有使用，相当于不活跃的
2. 为基本块分配寄存器

- 如果所有寄存器都被占用了，并且还需要申请一个寄存器，则应该考虑使用已经存放的死亡值的寄存器，因为这个值不需要保存到内存
  - 如果一个值在基本块结尾处是死的就不必在结尾处保存这个值

## 活跃变量计算流程：

**defB**: 在基本块B中定值, 但是定值前在B中没有被引用的变量集合

`useB`: 在基本块B中引用，但是引用前在B中没有被定值的变量集合



## 伪代码

IN[EXIT] =  $\emptyset$

for 除EXIT之外每个基本块B

$$\text{IN[B]} = \emptyset$$

for 某个IN值发生了改变

for 除EXIT之外的每个基本块B

$OUT[B] = \cup S$ 是B的一个后继  $IN[S]$

$$IN[B] = useB \cup (OUT[B] - defB)$$

活跃变量也是通过传递函数计算得到的，这里S是B的一个后继，比如上图B3的后继是B4，所以计算B3的OUT[B]时需要将B4的IN[B]集合并到B3来，注意计算活跃变量是一个逆向数据流，从EXIT开始的。

	OUT[B] <sup>1</sup>	IN[B] <sup>1</sup>	OUT[B] <sup>2</sup>	IN[B] <sup>2</sup>	OUT[B] <sup>3</sup>	IN[B] <sup>3</sup>
B4		u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3	j,u2,u3
B3	u3	u2,u3	j,u2,u3	j,u2,u3		j,u2,u3
B2	u2,u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3		i,j,u2,u3
B1	i,j,u2,u3	m,n,u1,u2,u3	i,j,u2,u3	m,n,u1,u2,u3		m,n,u1,u2,u3

第一轮IN值发生了改变 第二轮IN值发生了改变

第二轮IN值没有改变，  
跟上一轮一样

上图灰色的背景就是OUT[B]，比如B1里活跃变量只有i,j,u2,u3。

每个块对应的活跃变量列表，如下：

OUT[B]	B1	B2	B3	B4
a	X	X	X	X
i	✓	✓	✓	✓
j	✓	✓	X	✓
m	X	X	X	X
n	X	X	X	X
u1	X	X	X	X
u2	✓	✓	✓	✓
u3	✓	✓	✓	✓

//求解活跃变量数据流方程

```

func (lv *liveness) solve() {
    // 这些临时位向量的存在是为了避免循环内的连续内存分配和释放。
    nvars := int32(len(lv.vars))
    newlivein := bitvec.New(nvars)
    newliveout := bitvec.New(nvars)

    // 返回后序遍历块
    po := lv.f.Postorder()

    // 活跃变量是一个逆向数据流工程，返向遍历
    for change := true; change; {
        change = false
        for _, b := range po {
            be := lv.blockEffects(b)

            newliveout.Clear()
            switch b.Kind {
            case ssa.BlockRet:
                for _, pos := range lv.cache.retuevar {
                    newliveout.Set(pos)
                }
            case ssa.BlockRetJmp:
                for _, pos := range lv.cache.tailuevar {
                    newliveout.Set(pos)
                }
            case ssa.BlockExit:
                //恐慌退出
            default:

                // 块的输出计算
                // 计算公式
                // out[b] = \bigcup_{s \in succ[b]} in[s]
                newliveout.Copy(lv.blockEffects(b.Succs[0].Block()).livein)
                for _, succ := range b.Succs[1:] {
                    newliveout.Or(newliveout, lv.blockEffects(succ.Block()).livein)
                }
            }

            if !be.liveout.Eq(newliveout) {
                change = true
                be.liveout.Copy(newliveout)
            }

            // 块输入计算
            // 计算公式
            // in[b] = uevar[b] \cup (out[b] \setminus varkill[b])
            newlivein.AndNot(be.liveout, be.varkill)
            be.livein.Or(newlivein, be.uevar)
        }
    }
}

```

# 机器码

GO语言编译的最后一个阶段就是根据SSA中间代码生成机器码，这里机器码是在目标CPU架构上能够运行的二进制代码，机器码生成过程其实是对SSA中间代码的降级过程，在SSA中间代码降级的过程中，编译器将一些值重写成了目标CPU架构的特定值，降级过程处理了所有机器特定的重写规则并对代码进行了优化。

## 寄存器分配

分配寄存器有两种方法：

1. 图染色寄存器分配算法
2. 线性扫描寄存器分配算法

无论是图染色还是线性扫描算法，都涉及到活跃变量分析。假设a、b是程序中的两个变量，如果任何一个程序点a、b中最多只有一个变量是活跃的，那么a、b可以共用一个寄存器。

## 线性扫描

线性扫描的重点是需要记录当前指令哪些地方在使用，当物理寄存器不够的情况下，就需要做溢出操作，选一个周期较长的寄存器将其进行存储，这里还需要对当前寄存器变量下一个使用位置进行记录。

例子：

1. load @x -> v1
2. load @y -> v2
3. load @z -> v3
4. mult v1, v3 -> v4
5. add v4, v2 -> v5
6. add v5, v1 -> v6
7. store v6 -> z

空闲链表	P1	虚拟寄存器	下一次使用虚拟寄存器	虚拟寄存器	物理寄存器	用户
p1,p2		v1	4, 6	v1	p1	4,6(指令4和指令6使用了v1虚拟寄存器)
p2		v5	6	v2	X	5
X		v6	7	v3	p2 , X	4
p2	P2	虚拟寄存器	下一次使用虚拟寄存器	v4	p2 , X	5
X		v2	5	v5	p1, X	6
p1,p2		X		v6	p1	7
p1,p2		v4	5			
		X				

空闲链表主要存当前可用的寄存器资源，此例子有两个物理寄存器P1和P2，虚拟寄存器有六个从v1到v6。

初始化的时候，空闲链表里有P1和P2，`load @x->v1`，将从空闲链表里弹出一个p1分配给v1，然后计算出v1被引用的地方，这里代码第四行和第六行分别使用了v1，将其记录一下。

此时空闲队列里只剩p2物理寄存器，执行`load @y->v2`，将p2从空闲对列中移除，v2下次使用的虚拟寄存器是5。

执行`load @z->v3`的时候，发现空闲队列里没有可用的物理寄存器，这里就会产生寄存器溢出。需要在v1和v2中选择一个下次使用周期较长的虚拟寄存器，对其进行store。这里v2下次使用是5，而v1下次使用是4，所以将v2的物理寄存器进行存储，也就是p2寄存器值存储到y里，以此类推。

经过线性扫描完成过后，最后就形成了如下：

1. `load @x -> p1`
2. `load @y -> p2`
3. `store p2 -> @y dead code` 这行属于无用存储
4. `load @z -> p2`
5. `mult p1, p2 -> p2`
6. `store p1 -> @x`
7. `load @y -> p1`
8. `add p2, p1 -> p1`
9. `load @x -> p2`
10. `add p1, p2 -> p1`
11. `store p1 -> @z`

假设我们无法对指令重新排序，当然我们重新排序寄存器，可以最大限度减少所需的加载和存储数量。

但是在编译器中，不会一次完成所有的事情，不会同时进行寄存器分配和指令调度，大多数生产编译器是分开进行处理的。一般分为一个指令调度阶段和一个寄存器分配阶段。

在寄存器分配阶段，算法通常不考虑重新排序指令，如果考虑重新排序指令，它将变得非常复杂。完成调度是在寄存器分配之前和之后完成的，实际情况这种算法并不是最优的，当我们不清楚选择哪个寄存器做为牺牲器时，一个虚拟寄存器作为牺牲品来获取物理寄存器。线性扫描获取的物理寄存器，主要是因为它将来会用得最久。但这并不总是最好选择，因为有时最好的选择可能是选择一个自加载以为未被修改的寄存器。因此，如果有一个自加载后未被修改的寄存器，这意味着它是干净的，这意味着不需要把它存储在内存中。

## 寄存器分配流程

GO寄存器分配目前采用的线性扫描，主要通过支配树及活跃变量分析，其基本思想还是跟上面一样。

### 资源初始化

首先GO会对寄存器资源进行初始化，常见的SP、SB及G寄存器进行检查，找到我们可以使用哪些寄存器，不同的平台使用的寄存器也会有些不同，比如在arm中使用动态link的时候就需要用到R9寄存器。线性扫描寄存器分配会受到块出现顺序的影响，这里主要是采用了拓扑排序的思想进行块的排序。就跟学习一门技术一样，先学哪个，然后学哪个，才能掌握此门技术。针对活跃变量分析，主要记录活跃变量到下次使用此变量的距离以及每个块末尾的期望寄存器赋值，方便后续的寄存器的溢出操作。最后会初始化支配树，也是对后续寄存溢出时找到此前节点的孩子或者兄弟节点。

### 数据结构

### 寄存器申请

从可以的寄存器集里随机获取一个未使用的寄存器，如果没有可用的寄存器就会做溢出操作，根据距离来计算需要溢出的寄存器，如果有空闲的寄存器，在释放之前进行复制操作。

这里的pickReg操作涉及到一个德布鲁因序列算法，该算法能够快速找到二进制最后1的位置，具体参见：[德布鲁因序列](#)。

```
func (s *regAllocState) allocReg(mask regMask, v *Value) register {
    .....
    //如果可用，选择一个未使用的寄存器。
    if mask&^s.used != 0 {
        return pickReg(mask &^ s.used)
    }
}
```

//找到一个要溢出的寄存器。我们溢出包含下一次使用的值的寄存器，该值的下一次使用尽可能在最远距离使用。

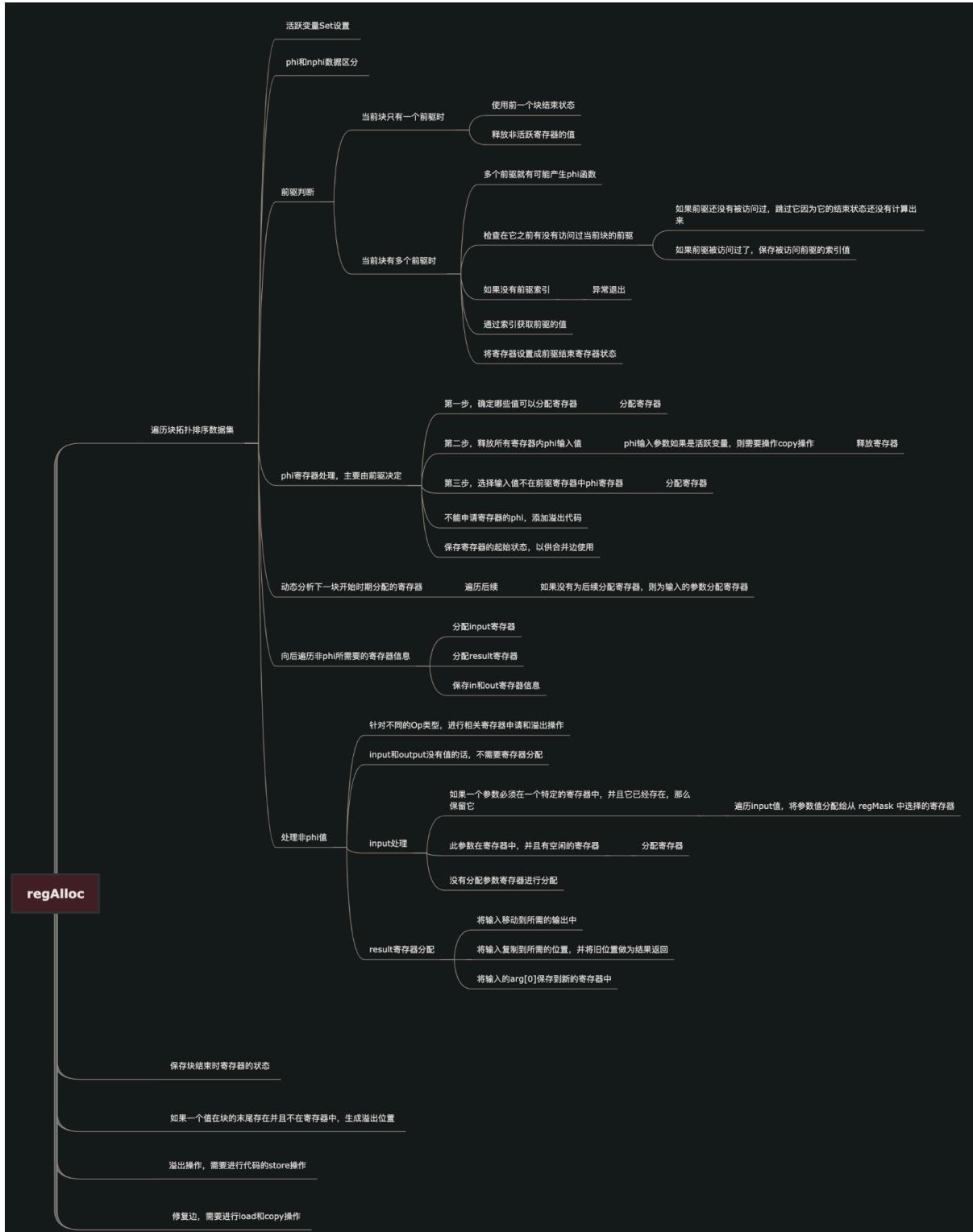
```

var r register
maxuse := int32(-1)
for t := register(0); t < s.numRegs; t++ {
    // 判断寄存器是否使用
    if mask>>t&1 == 0 {
        continue
    }
    v := s.regs[t].v
    if n := s.values[v.ID].uses.dist; n > maxuse {
        //v 的下一次使用比任何值都更远, 最佳候选者
        r = t
        maxuse = n
    }
}
.....
//如果有空闲寄存器, 尝试在free之前移动它。
v2 := s.regs[r].v
m := s.compatRegs(v2.Type) && s.used && s.tmpused && (regMask(1) << r)
if m != 0 && !s.values[v2.ID].rematerializeable && countRegs(s.values[v2.ID].regs) == 1 {
    r2 := pickReg(m)
    c := s.curBlock.NewValue1(v2.Pos, OpCopy, v2.Type, s.regs[r].c)
    s.copies[c] = false
    if s.f.pass.debug > regDebug {
        fmt.Printf("copy %s to %s : %s\n", v2, c, &s.registers[r2])
    }
    s.setOrig(c, v2)
    s.assignReg(r2, v2, c)
}
s.freeReg(r)
return r
}

```

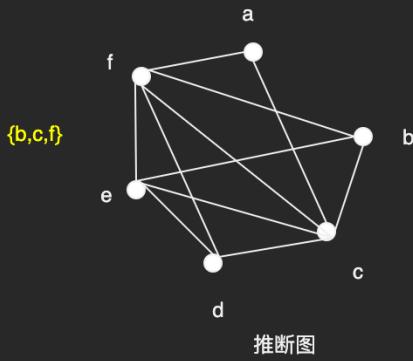
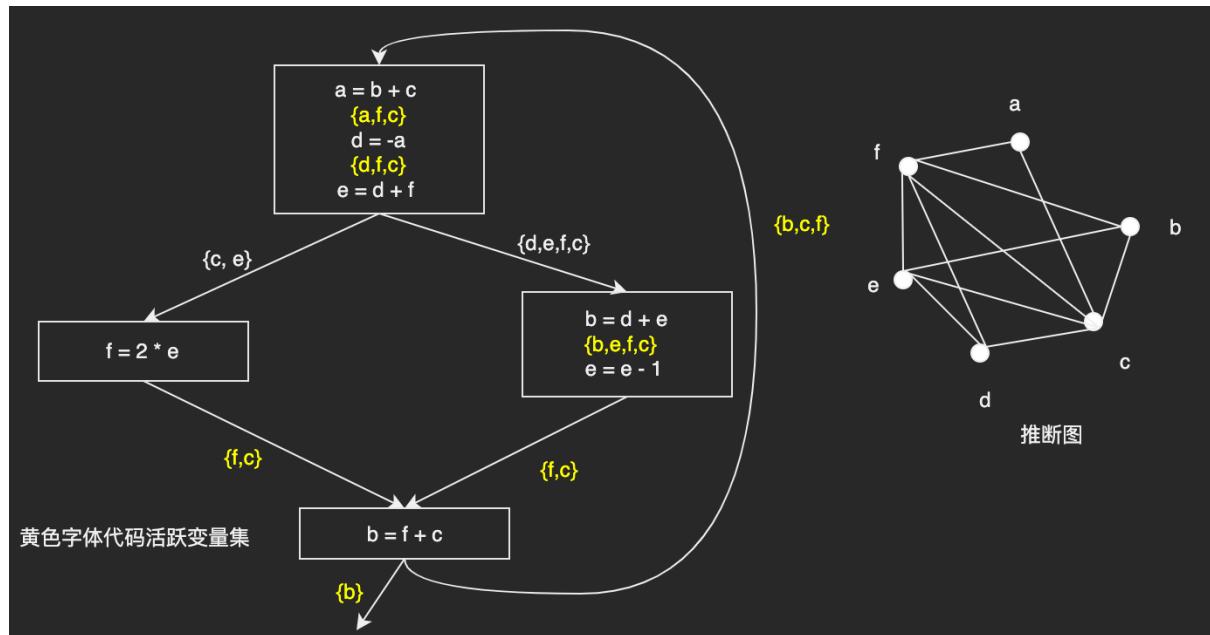
在申请寄存器之前还需要处理phi函数和非phi函数的处理, 然后决定我们需要溢出哪些寄存器, 最后修复一下溢出的边, 也就是我们在什么时候进行load和store。

下面是regAlloc的基本流程:

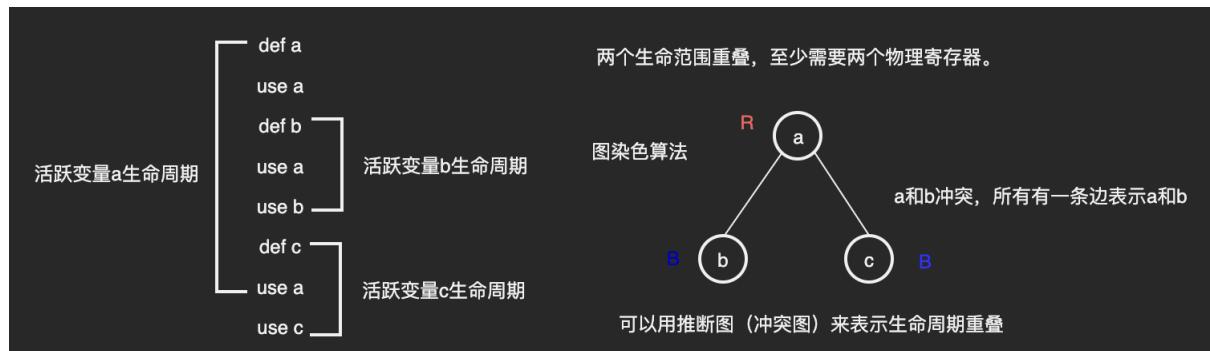


## 图染色方法

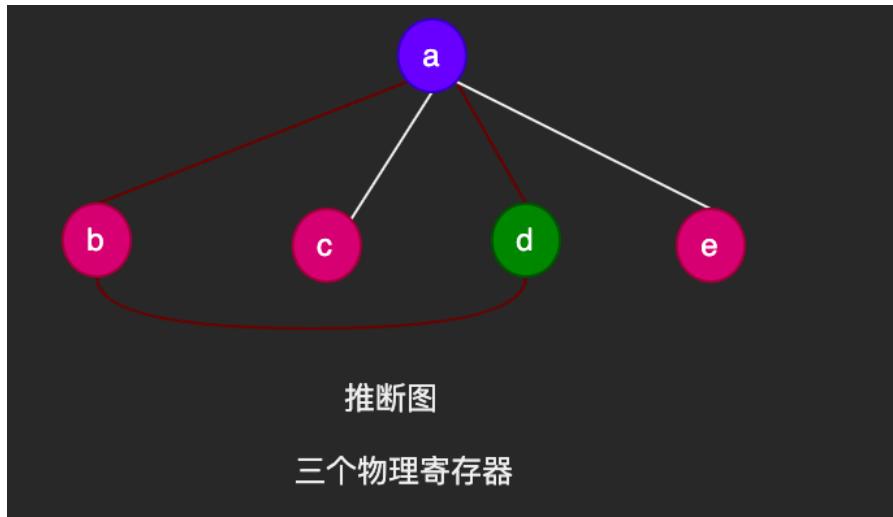
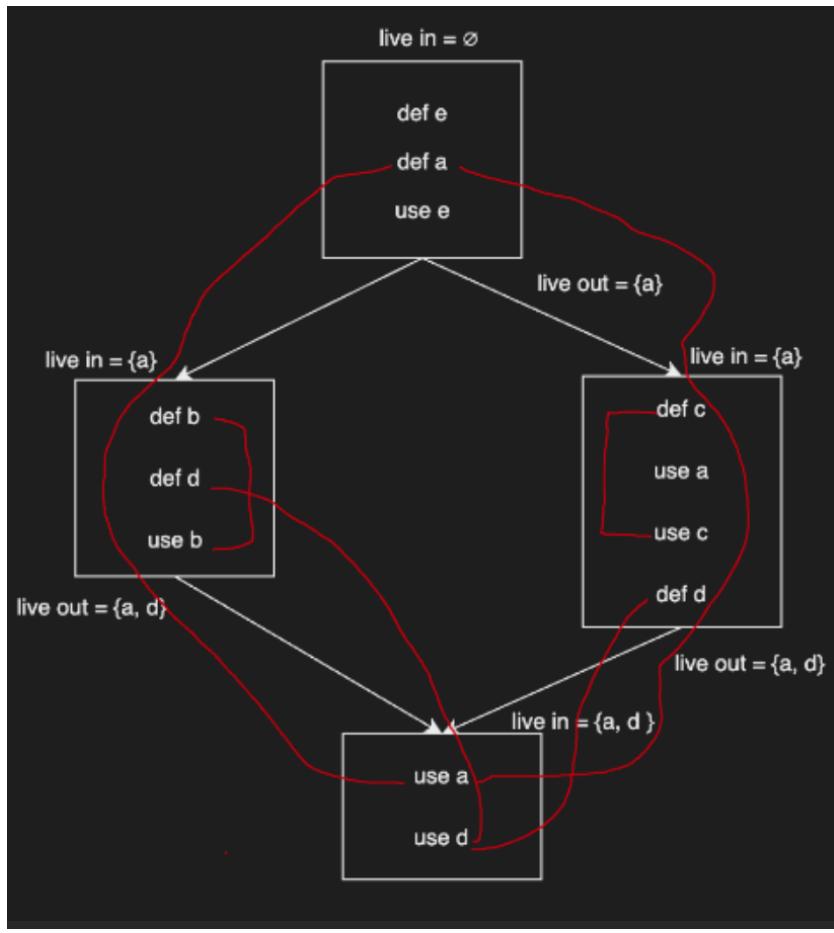
图染色寄存器分配算法，将活跃变量分析的结果构成一个无向图，该图的每个顶点代表一个变量，每一条边代表变量之间的活跃关系，例如某个程序点活跃变量分析结果为:{a,f,c}则图中a,f,c用边连接起来，构成一个无向图，这个图叫做推断图(interference graph)



## 块分析



案例：

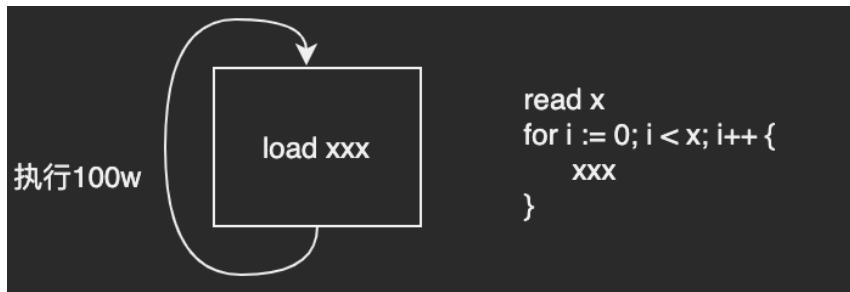


### 溢出活跃范围

意味着在每个定义之后插入一个store, 每次使用之前插入一个load。

溢出具有大量定义和使用的寄存器或具有较少定义和使用的寄存器是否有意义？更少的更有意义，因为定义和使用较少的那个将需要较少的加载和存储。目标是尽量减少将插入到代码中的加载和存储数量，所以更愿意溢出defs和uses较少的寄存器。但是defs和uses的数量并不是唯一的考虑因素，并不是所有基本块都会被执行相同的次数，因为一些基本块可能在循环内，一些基本块可能在外部循环。因此，如果寄存器在深层嵌套循环内的基本块中仅使用一次，那么load会被执行很多次，所以将不得不考虑循环的执行频率。这一切都是在编译时

间完成的，所以编译器并不知道每个基本块要执行多少次，不是所有的基本块都有循环次数计数。

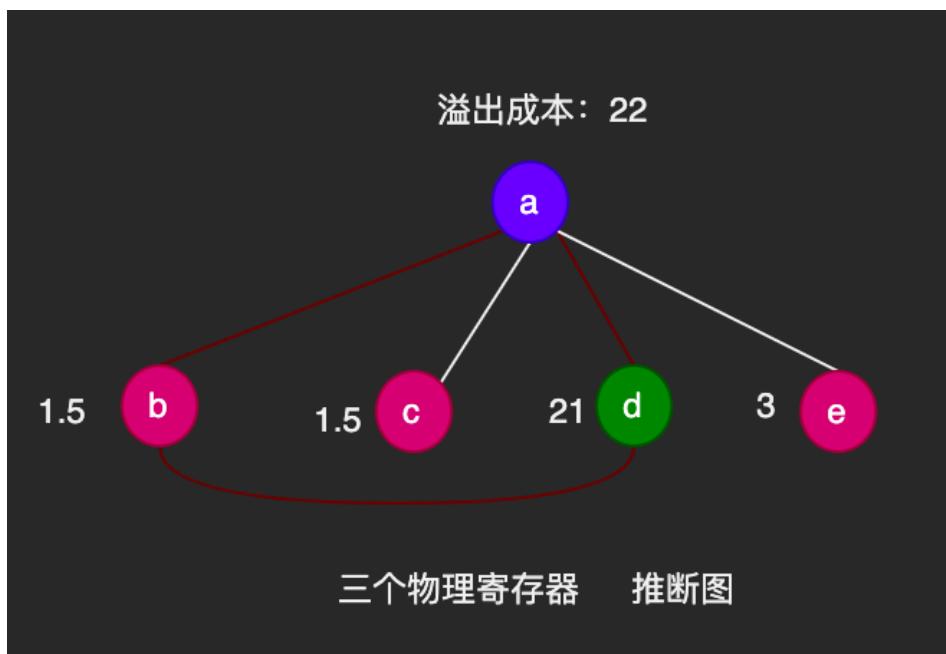


本地寄存器分配编译时间是线性的，但是运行速度非常差，因为生成的代码将具有过多的 load和store。在基本块中使用之前必须加载所有内容，导致load和store过多。

### 基于成本溢出算法

#### 泄露成本计算

```
load : 2
store: 1
def: defs * cost store * exec freq(执行块的频率)
use: uses * cost load * exec freq
```



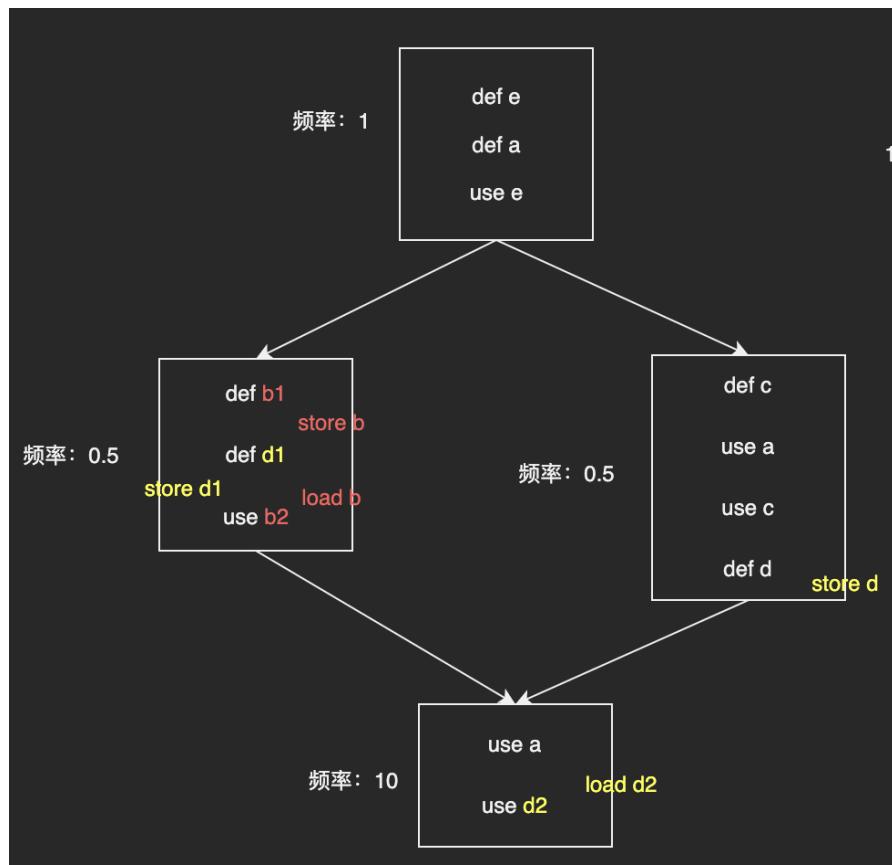
$$a \text{ 溢出成本} = 1 * 1 * 1 + 1 * 2 + 0.5 + 1 * 2 * 10 = 22$$

$$b \text{ 溢出成本} = 1 * 1 * 0.5 + 1 * 2 * 0.5 = 1.5$$

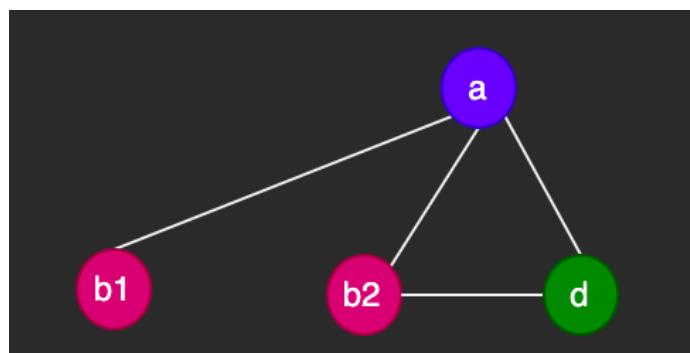
$$c \text{ 溢出成本} = 1.5$$

$$d \text{ 溢出成本} = 1 * 1 * 0.5 + 1 * 1 * 0.5 + 1 * 2 * 10 = 21$$

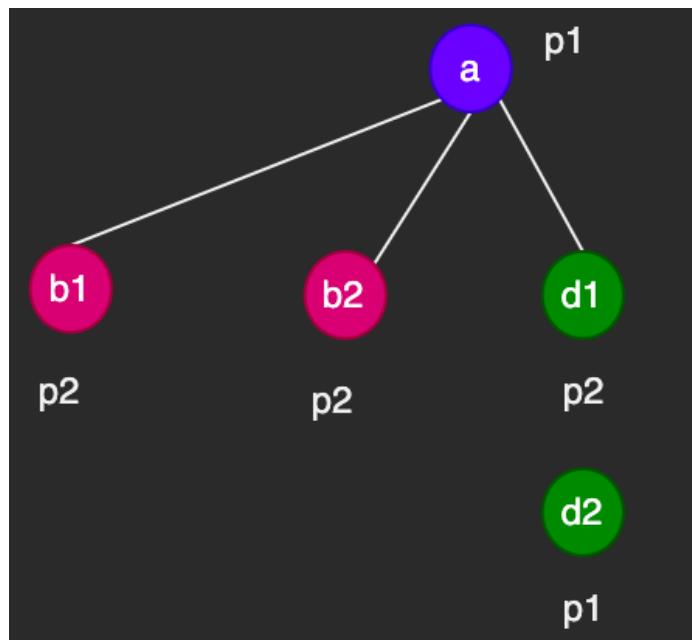
$$e \text{ 溢出成本} = 1 * 1 * 1 + 1 * 2 * 1 = 3$$



1、移出最小度的节点, c和e直接移出, a,b,d里b的溢出成本最少, 移出b, 这里用红色表示插入的代码。



2、溢出最便宜的寄存器d, 发现些图没有冲突了



指令调度

指令调度一般要进行两次，一次在寄存器分配之前，另一次在寄存器分配之后。

在寄存器分配后进行指令调度的明显优势？

n个指令，但是寄存器分配过后，我们将有 $n+m$ 个指令，其中m是溢出的。

## 为什么编译器会对指令重新排序？

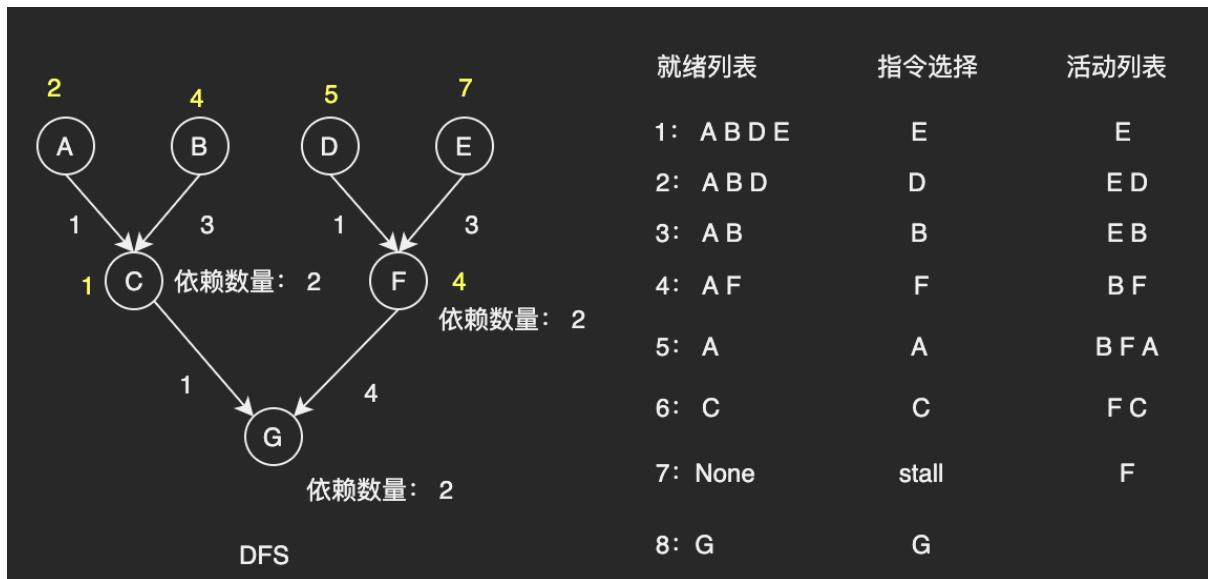
如果后续指令需要来自先前的指令，比如目标寄存器，如果在写入之后有一个读取，那么将导致停顿，这将导致延迟。

指令调度	原指令执行顺序	数据依赖图 (DDG)
A: loadi 5 -> r1	1:A	
B: load x -> r2	2:B	
C: add r1, r2 -> r3	3: stall	
D: loadi 7 -> r4	4: stall	
E: load y -> r5	5: C	
F: mult r4, r5 -> r6	6: D	
G: add r3, r6 -> r7	7: E	
load 延迟 = 3个周期	8: stall	$C: 2 + 3 = 5$
mult 延迟 = 4个周期	9: stall	$F: 7 + 3 = 10$
其它为1个周期	10: F	$G: 10 + 4 = 14$
单核处理器	11: stall 12: stall 13: stall 14: G	

这个指令流需要14个周期才行执行完，但是很明显，我们可以通过重新排序这些指令来最大程度地减少停顿次数。

list scheduling 列表调度算法，并不是一个特定的算法。跟踪就绪列表，选择就绪的条件必须是指令的关键路径距离，是数据依赖图中该指令与离开指令之间最长路径。

## 数据依赖图(DDG)

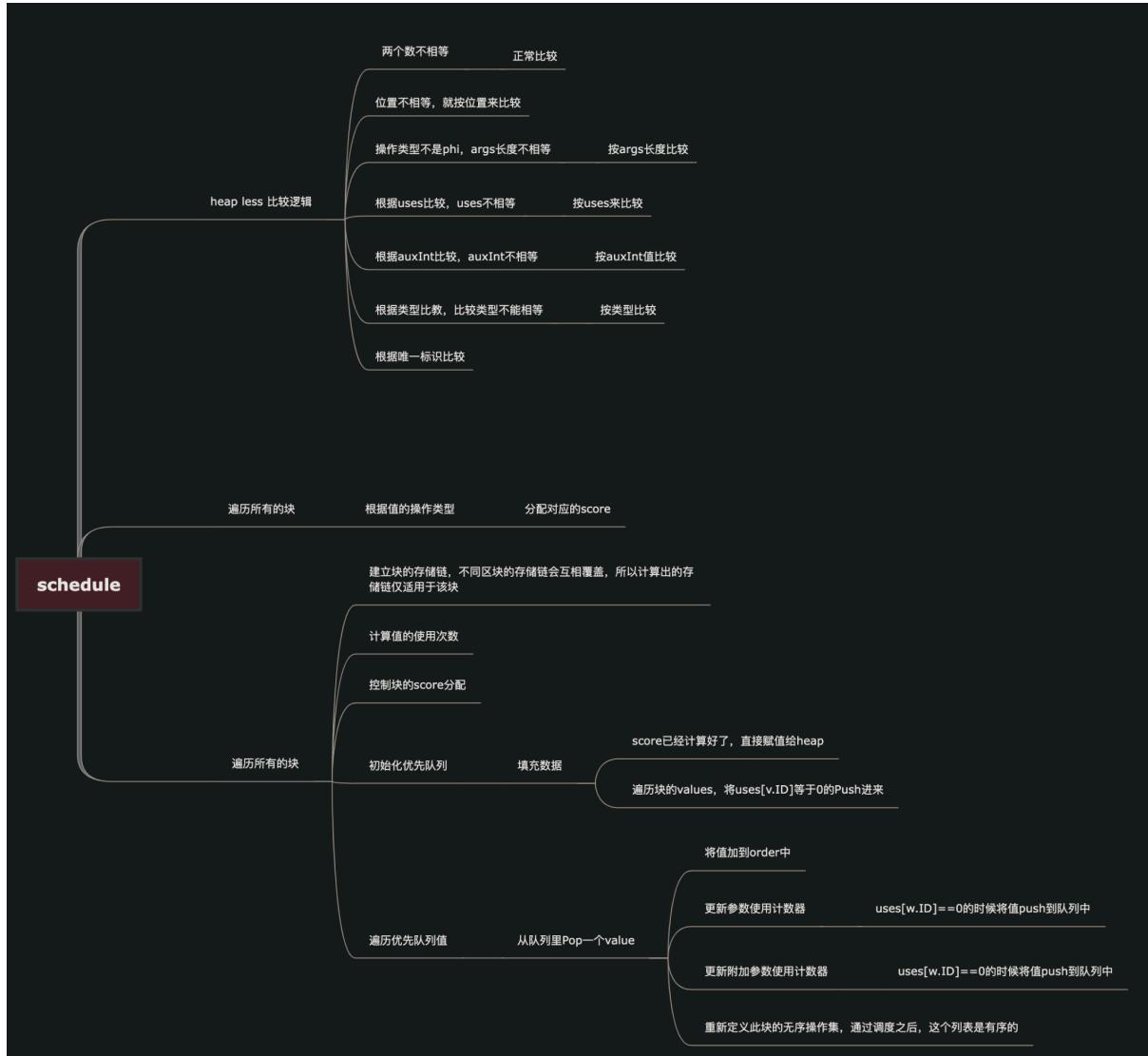


从上图的执行过程来看，指令调度完成后只需要8个周期。

黄色字体指的是路径长度，依赖数量就是入度，这里A、B、D、E依赖数量为0，活动列表值需要等待周期结束就可以删除，比如E在活动列表的周期是3，这时候就需要等其周期已满，到达第四行的时候进行删除。

## 优先队列

GO采用的数据结构是优先队列，指令执行的周期当成score。每次都从队列里弹出一个消耗最小的指令进行运算。



## 堆栈分配

堆栈主要为一些OpArg和干扰的值分配LocalSlot, 标识符已经在前期分配好了, 通过活跃变量分析及干扰图的构建进行处理, 其中活跃变量和干扰图存储采用了稀疏算法。

### Slot数据类型

```

type LocalSlot struct {
    N *ir.Name // 代表堆栈位置的 ONAME *ir.Name。
    Type *types.Type // 类型
    Off int64 // 偏移量

    SplitOf *LocalSlot // slot 是 SplitOf 的分解
    SplitOffset int64 // 分解的偏移量。
}
    
```

堆栈布局分为如下三种：

1、优化被禁用，`s` 在堆栈上并完整地表示。

[ -----s string ----] { N: s, Type: string, Off: 0 }

2、`s` 没有被分解，但是 SSA 单独操作它的各个部分，所以每个指向单个堆栈槽的字段都有一个 LocalSlot。

[ -----s string ----] { N: s, Type: \*uint8, Off: 0 }, {N: s, Type: int, Off: 8}

3、`s` 被分解，它的每个字段都在自己的堆栈槽中，并且有自己的 LocalSLot。

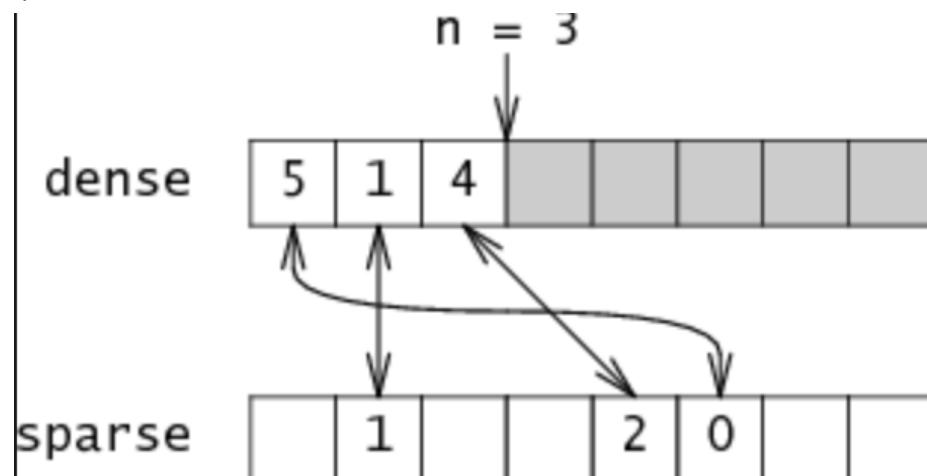
[ ptr \*uint8 ] [ len int] { N: ptr, Type: \*uint8, Off: 0, SplitOf: parent, SplitOffset: 0},

{ N: len, Type: int, Off: 0, SplitOf: parent, SplitOffset: 8}

parent = &{N: s, Type: string}

### 稀疏算法

稀疏算法主要是能在O(1)的时间内查询到指定的值，这里活跃变量和干扰图存储都使用此算法。采用了两个数组来实现对数值的映射，一个是密集数组另一个是稀疏数组，具体的对应关系如下：



当你查询4的时候，只需要先在sparse[4]找到对应的索引，比如是这里是2，然后dense[2]就得到4的值了。

### 干扰图算法

干扰图是一个图结构，每个变量是一个结点，结点的连线表示这两个变量是冲突的。也就是它们不能使用同一个寄存器。冲突的定义如果一个值已定义，而另一个值处于活动状态，则两个值会发生干扰。

### stackalloc整体流程



## 生成汇编

go会生成一种前端代码，主要保存了一些源操作数及目标操作数等信息，根据不同的CPU架构处理对应的Op操作翻译，在这之前已经生成SSA中间代码了，这里只是转换成prog对象，并对一些源和目标操作数的类型、偏移量及PC指令等处理。

```
//初始化对应构架的生成函数
func Init(arch *ssagen.ArchInfo) {
    arch.LinkArch = &x86.Linkamd64
    arch.REGSP = x86.REGSP
    arch.MAXWIDTH = 1 << 50

    arch.ZeroRange = zerorange
    arch.Ginsnop = ginsnop
    arch.Ginsnopdefer = ginsnop

    arch.SSAMarkMoves = ssaMarkMoves
    arch.SSAGenValue = ssaGenValue
    arch.SSAGenBlock = ssaGenBlock
    arch.LoadRegResults = loadRegResults
    arch.SpillArgReg = spillArgReg
}

// prog 数据结构
type Prog struct {
    Ctxt *Link // 链接器上下文
    Link *Prog // 链表中的下一个 Prog
    From Addr // 第一个源操作数
    RestArgs []AddrPos // 可以打包任何不适合 {Prog.From, Prog.To} 的操作数
    To Addr // 目标操作数(第二个是下面的 RegTo2)
    Pool *Prog // 常量池入口, 用于arm、arm64后端
    Forwd *Prog // 对于 x86 后端
    Rel *Prog // 对于 x86 后端
}
```

```

Pc    int64  // 对于后端或汇编器:虚拟或实际程序计数器, 取决于阶段性
Pos    src.XPos // 该指令的源位置
Spadj  int32  // 指令对堆栈指针的影响(递增或递减量)
As     As     // 汇编操作码
Reg    int16  // 第二个源操作数
RegTo2 int16  // 第二个目标操作数
Mark   uint16 // 特定于 arch 的项目的位掩码
Optab  uint16 // 特定于架构的操作码索引
Scond   uint8 // 描述指令后缀的位
Back   uint8  // 对于 x86 后端:向后分支状态
Ft     uint8  // 对于 x86 后端:Prog.From 的类型索引
Tt     uint8  // 对于 x86 后端:输入 Prog.To 的索引
Isize  uint8  /对于 x86 后端:指令的大小(以字节为单位)
}

```

```

// 将Op对应操作, 构建Prog
func ssaGenValue(s *ssagen.State, v *ssa.Value) {
    switch v.Op {
    case ssa.OpAMD64VFMADD231SD:
        p := s.Prog(v.Op.Asm())
        p.From = obj.Addr{Type: obj.TYPE_REG, Reg: v.Args[2].Reg()}
        p.To = obj.Addr{Type: obj.TYPE_REG, Reg: v.Reg()}
        p.SetFrom3Reg(v.Args[1].Reg())
    case ssa.OpAMD64ADDQ, ssa.OpAMD64ADDL:
        .....
    }
}

```

举个简单的例子，整体的执行流程：

**c := 10 + i**

根据SSA生成的中间代码进行Op匹配，这里是AMD64的架构，命中OpAMD64ADDQconst，args是标识是i。

```
> s: *cmd/compile/internal/ssagen.State {ABI: ABIInternal (1), pp: *cmd/compile/in...
✓ v: *cmd/compile/internal/ssa.Value {ID: 12, Op: OpAMD64ADDQconst (278), Type: *c...
  ✓ : cmd/compile/internal/ssa.Value {ID: 12, Op: OpAMD64ADDQconst (278), Type: *cm...
    ID: 12
    Op: OpAMD64ADDQconst (278)
  > Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, met...
    AuxInt: 10
  Aux: cmd/compile/internal/ssa.Aux nil
  > Args: []*cmd/compile/internal/ssa.Value len: 1, cap: 3, [*(*"cmd/compile/internal...
  > Block: *cmd/compile/internal/ssa.Block {ID: 1, Pos: (*"cmd/internal/src.XPos")...
  > Pos: cmd/internal/src.XPos {index: 2, lico: 28832}
  Uses: 1
  OnWasmStack: false
  InCache: false
  ✓ argstorage: [3]*cmd/compile/internal/ssa.Value [*(*"cmd/compile/internal/ssa...
    ✓ [0]: *(*"cmd/compile/internal/ssa.Value")(0xc000700310)
      ✓ : cmd/compile/internal/ssa.Value {ID: 7, Op: OpArgIntReg (2432), Type: *cmd...
        ID: 7
        Op: OpArgIntReg (2432)
      > Type: *cmd/compile/internal/types.Type {Extra: interface {} nil, Width: 8, ...
        AuxInt: 0
      ✓ Aux: cmd/compile/internal/ssa.Aux(*cmd/compile/internal/ssa_AUX_NAME_OFFSET)...
        ✓ data: *cmd/compile/internal/ssa.AuxNameOffset {Name: *(*"cmd/compile/inte...
          ✓ : cmd/compile/internal/ssa.AuxNameOffset {Name: *(*"cmd/compile/internal...
            ✓ Name: *(*"cmd/compile/internal/ir.Name")(0xc000440750)
              ✓ : cmd/compile/internal/ir.Name {miniExpr: cmd/compile/internal/ir.mini...
                > miniExpr: cmd/compile/internal/ir_miniExpr {miniNode: (*"cmd/compile...
                  BuiltinOp: OXXX (0) = 0x0
                  Class: PPARAM (4) = 0x0
                  pragma: 0
                  flags: 0 = 0x0
                ✓ sym: *cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/com...
                  ✓ : cmd/compile/internal/types.Sym {Linkname: "", Pkg: *(*"cmd/compil...
                    Linkname: ""
                  > Pkg: *(*"cmd/compile/internal/types.Pkg")(0xc0000925a0)
                    Name: "i"
                    Def: cmd/compile/internal/types.Object nil
                    Block: 0
                  > Lastlineno: cmd/internal/src.XPos {index: 0, lico: 0}
                    flags: 0 = 0x0
                  Func: *cmd/compile/internal/ir.Func nil
                  Offset_: 0
                  val: go/constant.Value nil
```

## prog结构体初始化赋值

```
> v: *cmd/compile/internal/ssa.Value {ID: 12, Op: 0pAMD64ADDQconst (278), Type: *cmd/compile/internal/types.Type {Extra: 0, Name: "", Kind: CONST}, Value: 0x0, Loc: 0xc000166800, Reg: 2064, Index: 0, Scale: 1, Type: TYPE_CONST (4), Name: NAME_NONE (0), Class: 0, Offset: 0}
  r: 2064
  a: 2064

  < p: *cmd/internal/obj.Prog {Ctx: *cmd/internal/obj.Link {Headtype: Hdarwin (1), Arch: *(*cmd/internal/obj.LinkArch)(&LinkArch{Headtype: Hdarwin (1), Arch: *(*cmd/internal/obj.LinkArch)(0x21752c0)}), Debugasm: "", Sym: nil, Val: interface {}}, From: cmd/internal/obj.Addr {Reg: 0, Index: 0, Scale: 0, Type: TYPE_CONST (4), Name: NAME_NONE (0), Class: 0, Offset: 0}, To: cmd/internal/obj.Addr {Reg: 2064, Index: 0, Scale: 0, Type: TYPE_REG (7), Name: NAME_NONE (0), Class: 0, Offset: 0}}
    < From: cmd/internal/obj.Addr {Reg: 0, Index: 0, Scale: 0, Type: TYPE_CONST (4), Name: NAME_NONE (0), Class: 0, Offset: 0}
      Reg: 0
      Index: 0
      Scale: 0
      Type: TYPE_CONST (4) = 0x0
      Name: NAME_NONE (0)
      Class: 0
      Offset: 10
      Sym: *cmd/internal/obj.LSym nil
      Val: interface {} nil
      RestArgs: []cmd/internal/obj.AddrPos len: 0, cap: 0, nil
    < To: cmd/internal/obj.Addr {Reg: 2064, Index: 0, Scale: 0, Type: TYPE_REG (7), Name: NAME_NONE (0), Class: 0, Offset: 0}
      Reg: 2064
      Index: 0
      Scale: 0
      Type: TYPE_REG (7) = 0x0
      Name: NAME_NONE (0)
      Class: 0
      Offset: 0
      Sym: *cmd/internal/obj.LSym nil
      Val: interface {} nil
      Pool: *cmd/internal/obj.Prog nil
      Forwd: *cmd/internal/obj.Prog nil
      Rel: *cmd/internal/obj.Prog nil
      Pc: 4
    < Pos: cmd/internal/src.XPos {index: 2, lico: 28833}
      Spadj: 0
      As: cmd/internal/obj/x86.AADDQ (6172)
      Reg: 0
      RegTo2: 0
      Mark: 0 = 0x0
      Optab: 0 = 0x0
      Scond: 0 = 0x0
      Back: 0 = 0x0
```

## 最后生成对应的汇编代码

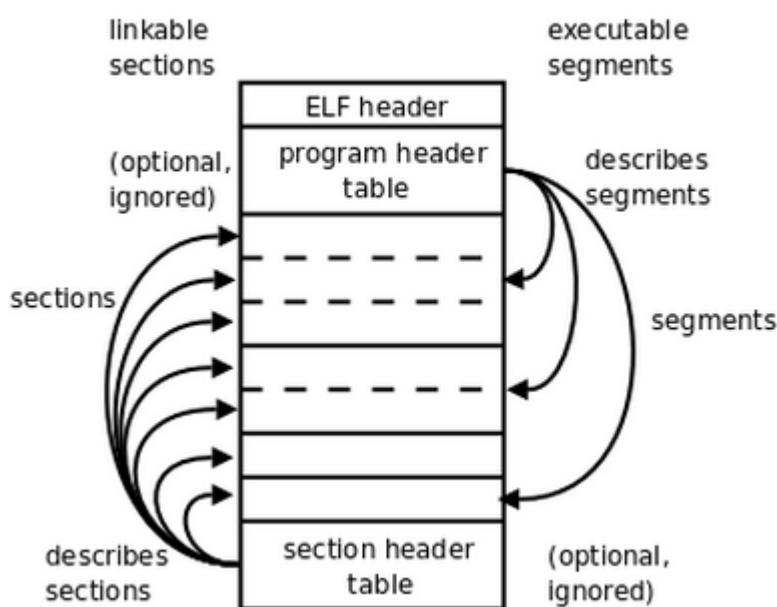
```
ssa-gen:00000 (<unknown line number>) TEXT    """.Sum(SB), ABIInternal
ssa-gen:00001 (<unknown line number>) FUNCDATA    $0, glocals·33cdecccbe80329f1fdbee7f5874cb(SB)
ssa-gen:00002 (<unknown line number>) FUNCDATA    $1, glocals·33cdecccbe80329f1fdbee7f5874cb(SB)
ssa-gen:00003 (<unknown line number>) FUNCDATA    $5, """.Sum.arginfo1(SB)
ssa-gen:00004 (<unknown line number>) ADDQ    $10, AX
ssa-gen:00005 (<unknown line number>) RET
ssa-gen:00006 (<unknown line number>) END
DISG
```

## ELF

ELF 的全称是 Executable and Linking Format, 即“可执行可连接格式”，通俗来说，就是二进制程序。ELF 规定了这二进制程序的组织规范，所有以这规范组织的文件都叫 ELF 文件。

ELF 文件类型	示例
可重定位文件(relocatable file)	GO 编译的 xxx.o 文件
共享目标文件(shared object file)	动态库 xxxx.so, 以及使用动态链接的 bin
可执行文件(executable file)	静态链接的文件 xxxx.a
core 文件	例如 Linux 上的 coredump 文件

首先，ELF 文件格式提供了两种视图，分别是链接视图和执行视图。



链接视图是以节(section)为单位，执行视图是以段(segment)为单位。链接视图就是在链接时用到的视图，而执行视图则是在执行时用到的视图。

名称	说明
ELF header	描述整个文件的组织
Program Header Table	描述文件中的各种segments, 用来告诉系统如何创建进程映像的
sections	sections是从链接的角度来描述elf文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序(所以很多加固手段删除了section header table)
segments	segments是从运行的角度来描述elf文件，segments与sections是包含的关系，一个segment包含若干个section
Section Header Table	包含了文件各个segction的属性信息，我们都将结合例子来解释

我们先来分析一下GO的ELF文件内容：

```
readelf -h ./main
```

```
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
        ELF64
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x44f4d0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 456 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 7
  Size of section headers: 64 (bytes)
  Number of section headers: 23
  Section header string table index: 3
```

Magic前几位置7f 45 4c 46是魔数，后面对应Class、Data、Version、OS/ABI及ABI Version，其余的字段主要是一些目标文件类型及体系结构类型及header对应的偏移量等。

```
readelf -l ./main
```

```
Elf file type is EXEC (Executable file)
Entry point 0x44f4d0
There are 7 program headers, starting at offset 64

Program Headers:
Type          Offset        VirtAddr       PhysAddr
              FileSiz      MemSiz         Flags  Align
PHDR          0x0000000000000040 0x0000000000400040 0x0000000000400040
              0x0000000000000188 0x0000000000000188 R      0x1000
NOTE          0x0000000000000f9c 0x0000000000400f9c 0x0000000000400f9c
              0x0000000000000064 0x0000000000000064 R      0x4
LOAD          0x0000000000000000 0x0000000000400000 0x0000000000400000
              0x000000000008213b 0x000000000008213b R E    0x1000
LOAD          0x0000000000083000 0x0000000000483000 0x0000000000483000
              0x00000000000905b6 0x00000000000905b6 R      0x1000
LOAD          0x00000000000114000 0x0000000000514000 0x0000000000514000
              0x0000000000013700 0x000000000032438 RW    0x1000
GNU_STACK     0x0000000000000000 0x0000000000000000 0x0000000000000000
              0x0000000000000000 0x0000000000000000 RW    0x8
LOOS+0x5041580 0x0000000000000000 0x0000000000000000 0x0000000000000000
              0x0000000000000000 0x0000000000000000 0x0000000000000000 0x8

Section to Segment mapping:
Segment Sections...
 00
 01  .note.go.buildid
 02  .text .note.go.buildid
 03  .rodata .typelink .itablink .gosymtab .gopclntab
 04  .noptrrdata .data .bss .noptrbss
 05
 06
```

segment是sections的集合，多个section组成一个segment。

这些字段对应是一些成员对应内存中的虚拟地址，从文件头到该段第一个字节的偏移量以及内存对齐。

PHDR:program header table本身的信息

LOAD:loadable segment

NOTE:一些辅助信息

GNU\_STACK:只是一个ELF程序头，它告诉系统当ELF加载到内存中时如何控制堆栈

## 生成机器码

不同平台架构机器码会有些不同，不过生成逻辑基本是相同的，我们这里主要介绍一下arm32平台。查看机器码对应的格式可以参考些文档《ARM Architecture Reference Manual》。

## 条件操作码信息

<b>Code</b>	<b>Suffix</b>	<b>Flags</b>	<b>Meaning</b>
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

*Table 4-2: Condition code summary*

## 指令类型信息

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2(operand1 is ignored)
BIC	1110	operand1 AND NOT operand2(Bit clear)
MVN	1111	NOT operand2(operand1 is ignored)

Mov机器码的格式

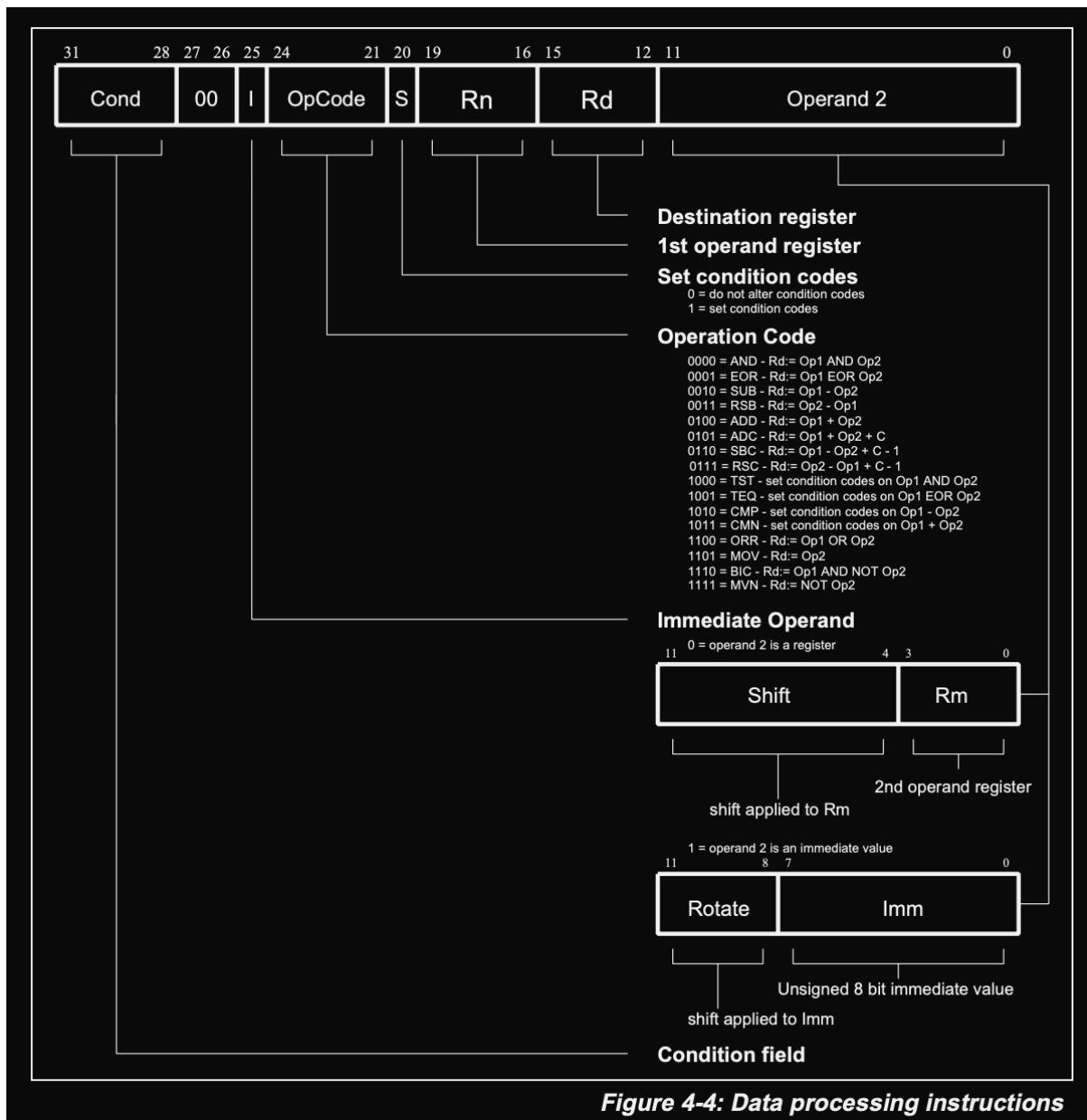


Figure 4-4: Data processing instructions

## 指令转换

Mov r0, r1 对应机器码: 0xe1a00001= 0b 1110 00 0 1101 0 0000 0000 00000000000001

Moveq r0, 0xff 对应机器码: 0x03a000ff=0b 0000 00 1 1101 0 0000 0000 000011111111

Mov r0, r1 => 1110 00 0 1101 0 0000 0000 00000000000001

0b	1110	00	0	1101	0	0000	0000	000000 000001
二进制	条件码为无条件	保留位	I位表示我们源操作数	opcode位, 指令类型	S位表示是否影响	代表Rn, 第一操作数	目的操作数, 0000代	源操作数是寄存器名

		是什么类型, 0 :表示寄存器 1:立即数		cpsr(控制寄存器)	寄存器	表是R0寄存器	字是R0。 如果I位是0:源操作数寄存器, 1:立即数,,前面四位是移位。所以表示的数最大0xff
--	--	--------------------------	--	-------------	-----	---------	--

## GO转换逻辑

```
//建立optab对应翻译表
var optab = []Optab{
    /* struct Optab:
       OPCODE, from, prog->reg, to, type, size, param, flag, extra data size, optional suffix */
    {obj.ATEXT, C_ADDR, C_NONE, C_TEXTSIZE, 0, 0, 0, 0, 0, 0},
    {AADD, C_REG, C_REG, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AADD, C_REG, C_NONE, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AAND, C_REG, C_REG, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AAND, C_REG, C_NONE, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AORR, C_REG, C_REG, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AORR, C_REG, C_NONE, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AMOVW, C_REG, C_NONE, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {AMVN, C_REG, C_NONE, C_REG, 1, 4, 0, 0, 0, C_SBIT},
    {ACMP, C_REG, C_REG, C_NONE, 1, 4, 0, 0, 0, 0},
    {AADD, C_RCON, C_REG, C_REG, 2, 4, 0, 0, 0, C_SBIT},
    {AADD, C_RCON, C_NONE, C_REG, 2, 4, 0, 0, 0, C_SBIT},
    {AAND, C_RCON, C_REG, C_REG, 2, 4, 0, 0, 0, C_SBIT},
    {AAND, C_RCON, C_NONE, C_REG, 2, 4, 0, 0, 0, C_SBIT},
    {AORR, C_RCON, C_REG, C_REG, 2, 4, 0, 0, 0, C_SBIT},
    .....
}
```

通过汇编对应的操作查找对应optab信息，主要逻辑在span5函数里，并对bp和pc的偏移量进行计算。

```
func (pp *Progs) Flush() {
    plist := &obj.Plist{Firstpc: pp.Text, Curfn: pp.CurFunc}
    obj.Flushplist(base.Ctxt, plist, pp.NewProg, base.Ctxt.Pkgpath)
}
```

```
//将函数转换为机器码
func Flushplist(ctx *Link, plist *Plist, newprog ProgAlloc, myimportpath string) {
    .....
```

```

for _, s := range text {
    mkfwd(s)
    if ctxt.Arch.ErrorCheck != nil {
        ctxt.Arch.ErrorCheck(ctxt, s)
    }
    linkpatch(ctxt, s, newprog)
    ctxt.Arch.Preprocess(ctxt, s, newprog)
    ctxt.Arch.Assemble(ctxt, s, newprog)
    if ctxt.Errors > 0 {
        continue
    }
    linkpeln(ctxt, s)
    if myimportpath != "" {
        ctxt.populateDWARF(plist.Curfn, s, myimportpath)
    }
}

.....
}

// 将对象写入文件中
func WriteObjFile(ctxt *Link, b *bio.Writer) {

debugAsmEmit(ctxt)

genFuncInfoSyms(ctxt)

w := writer{
    Writer: goobj.NewWriter(b),
    ctxt: ctxt,
    pkgpath: objabi.PathToPrefix(ctxt.Pkgpath),
}
start := b.Offset()
w.init()

.....
}

// Header
h := goobj.Header{
    Magic: goobj.Magic,
    Fingerprint: ctxt.Fingerprint,
    Flags: flags,
}
h.Write(w.Writer)

.....
}

// Symbol definitions
h.Offsets[goobj.BlkSymdef] = w.Offset()
for _, s := range ctxt.defs {

```

```
// 二进制方式写入  
w.Sym(s)  
}  
  
.....  
}
```

## 总结

本文只是浅略的介绍了GO编译器，也给出了多个方案来实现编译策略，从数据结构到算法，前端到后端，基本覆盖了整个编译的流程，GO项目成员并没有提供更多的文档介绍实现流程，文中可能不是非常正常。其中GO编译器好多算法的实现和策略都有待优化，比如最常见的文法分析、指令调度及中间代码优化等，并不是最优解。