

Go底层探索(一):编译器

刘庆辉 猿码记 2023-01-15 03:00 Posted on 北京

收录于合集

#Go 101 #Go进阶 14

1. 介绍

@注: 以下内容来自本人学习《Go语言底层原理剖析》书中的摘要信息。另外这本书中使用的 Go 是老版本, 我使用的版本是 Go1.18 有时候源码路径可能会不一样

编译器是一个大型且复杂的系统, 一个好的编译器会很好地结合形式语言理论、算法、人工智能、系统设计、计算机体系结构及编程语言理论。

Go 语言的编译器遵循了主流编译器采用的经典策略及相似的处理流程和优化规则 (例如经典的递归下降的语法解析、抽象语法树的构建)。另外, Go 语言编译器有一些特殊的设计, 例如内存的逃逸等;

1.1 为什么要了解Go语言编译器?

通过了解 Go 语言编译器, 不仅可以了解大部分高级语言编译器的一般性流程与规则, [也能指导我们写出更加优秀的程序。](#)

1.2 三阶段编译器

在经典的编译原理中, 一般将编译器分为**编译器前端**、**优化器**和**编译器后端**。这种编译器被称为三阶段编译器 (`three-phase compiler`) 。

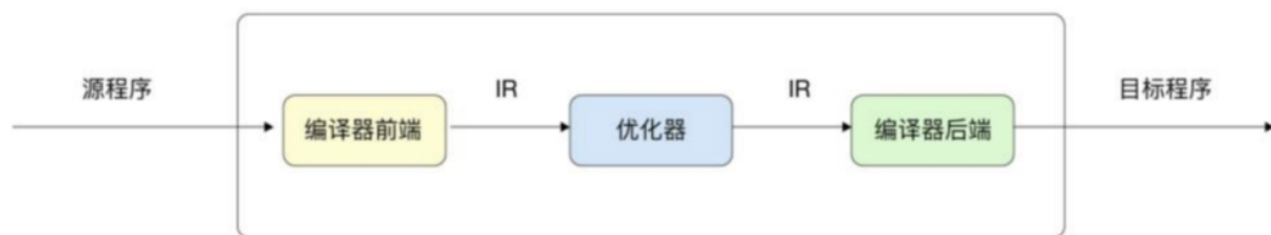


图1-1 三阶段编译器

猿码记

- **编译器前端**: 专注于理解源程序、扫描解析源程序并进行精准的语义表达;
- **优化器(中间阶段)**: 编译器会使用多个 **IR** 阶段、多种数据结构表示代码，并在中间阶段对代码进行多次优化。例如:识别冗余代码、识别内存逃逸等。**编译器的中间阶段离不开编译器前端记录的细节.**
- **编译器后端**: 专注于生成特定目标机器上的程序，这种程序可能是可执行文件，也可能是需要进一步处理的中间形态 **obj** 文件、汇编语言等.

编译器优化并不是一个非常明确的概念。优化的主要目的一般是降低程序资源的消耗，比较常见的是降低内存与 **CPU** 的使用率。但在很多时候，这些目标可能是相互冲突的，对一个目标的优化可能降低另一个目标的效率。

1.3 Go编译器阶段

Go 语言编译器一般缩写为小写的 **gc** (go compiler)，需要和大写的 **GC**（垃圾回收）进行区分。Go 语言编译器的执行流程可细化为多个阶段，包括词法解析、语法解析、抽象语法树构建、类型检查、变量捕获、函数内联、逃逸分析、闭包重写、遍历函数、SSA生成、机器码生成。



图1-2 Go语言编译器执行流程

猿码记

和 Go 语言编译器有关的代码主要位于 `src/cmd/compile/internal` 目录下，在后面分析中给出的文件路径均默认位于该目录中。

2. 词法解析

在词法解析阶段，Go 语言编译器会扫描输入的 Go 源文件，并将其符号（`token`）化。例如将表达式 `a := b + c(12)` 符号化之后的情形，如下图所示：

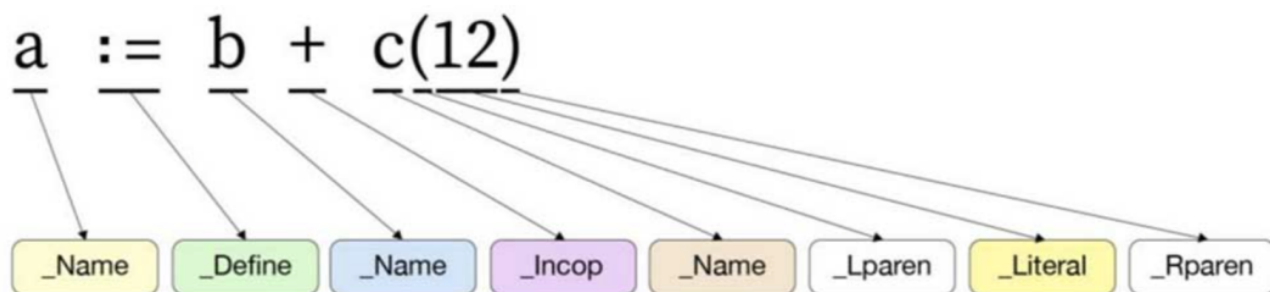
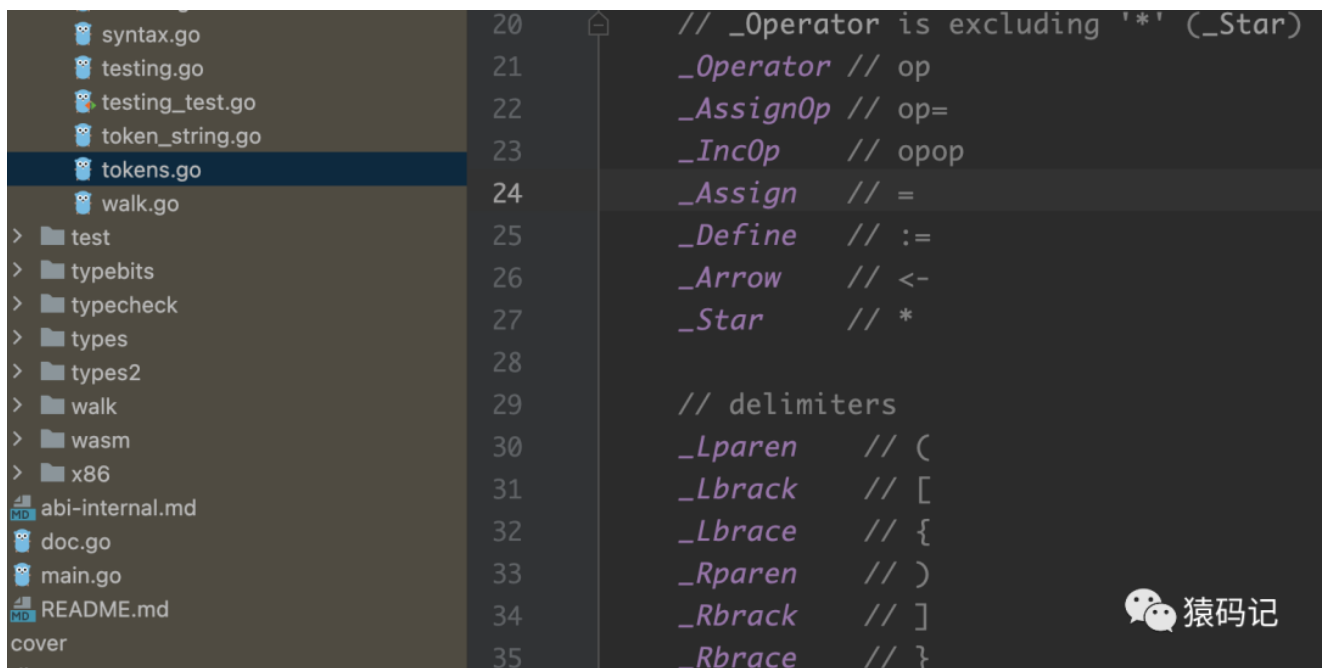


图1-3 Go语言编译器词法解析示例

从上图可以看出：

- `+` 操作符会被转换为 `_IncOp` ；
- 赋值符号 `:=` 会被转换为 `_Define` ；
- 变量名 `a`、`b`、`c` 会被转换为 `_Name` ；
- ...

实际上，这些 `token` 就是用 `iota` 声明的整数常量，定义在 `syntax/tokens.go` 文件中。如下图所示：



```
20 // _Operator is excluding '*' (_Star)
21 _Operator // op
22 _AssignOp // op=
23 _IncOp    // opop
24 _Assign   // =
25 _Define   // :=
26 _Arrow    // <-
27 _Star     // *
28
29 // delimiters
30 _Lparen    // (
31 _Lbrack    // [
32 _Lbrace    // {
33 _Rparen    // )
34 _Rbrack    // ]
35 _Rbrace    // }
```

2.1 总结归纳

- 符号化保留了 Go 语言中定义的符号，可以识别出错误的拼写。
- 字符串被转换为整数后，在后续的阶段中能够被更加高效地处理。

3. 语法解析

词法解析阶段结束后，需要根据 Go 语言中指定的语法对符号化后的 Go 文件进行解析。

Go 语言采用了标准的自上而下的递归下降 [Top-Down Recursive-Descent] 算法，以简单高效的方式完成无须回溯的语法扫描。

如下图示例:

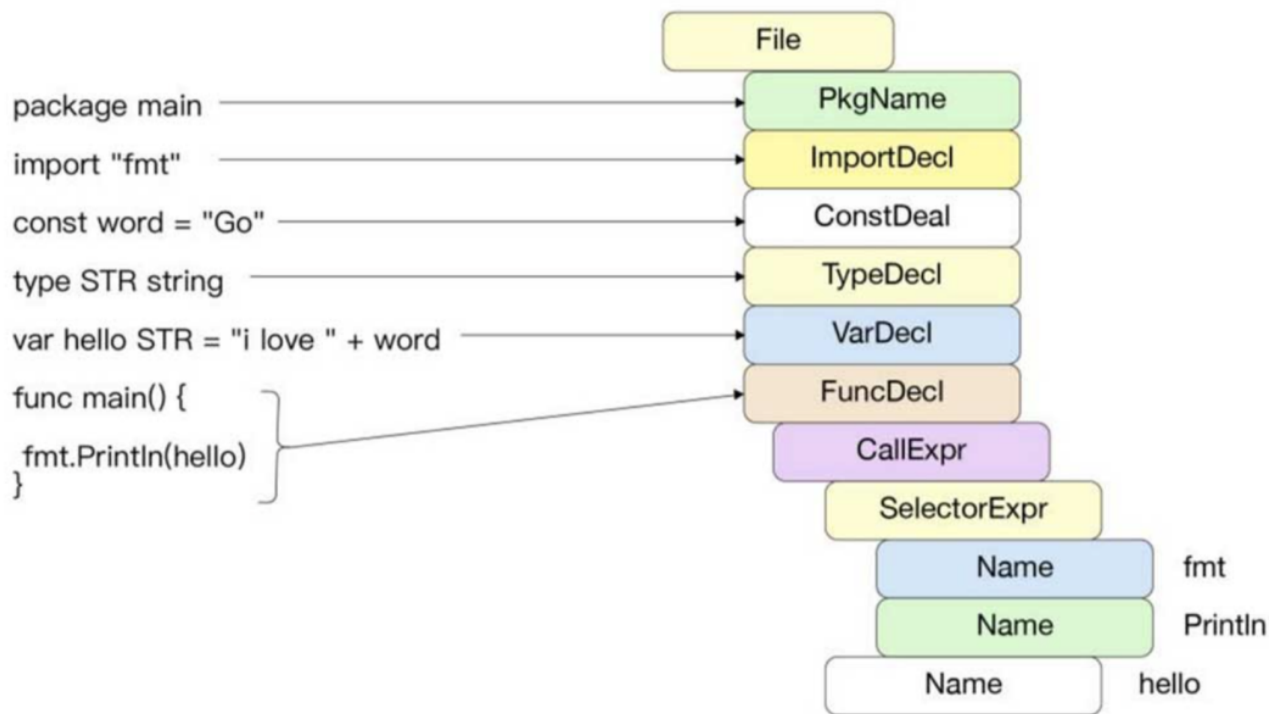


图1-4 Go语言编译器对文件进行语法解析的示意图  猿码记

源文件中的每一种声明都有对应的语法，采用对应的语法进行解析，能够较快地解析并识别可能出现的语法错误。

3.1 总结归纳

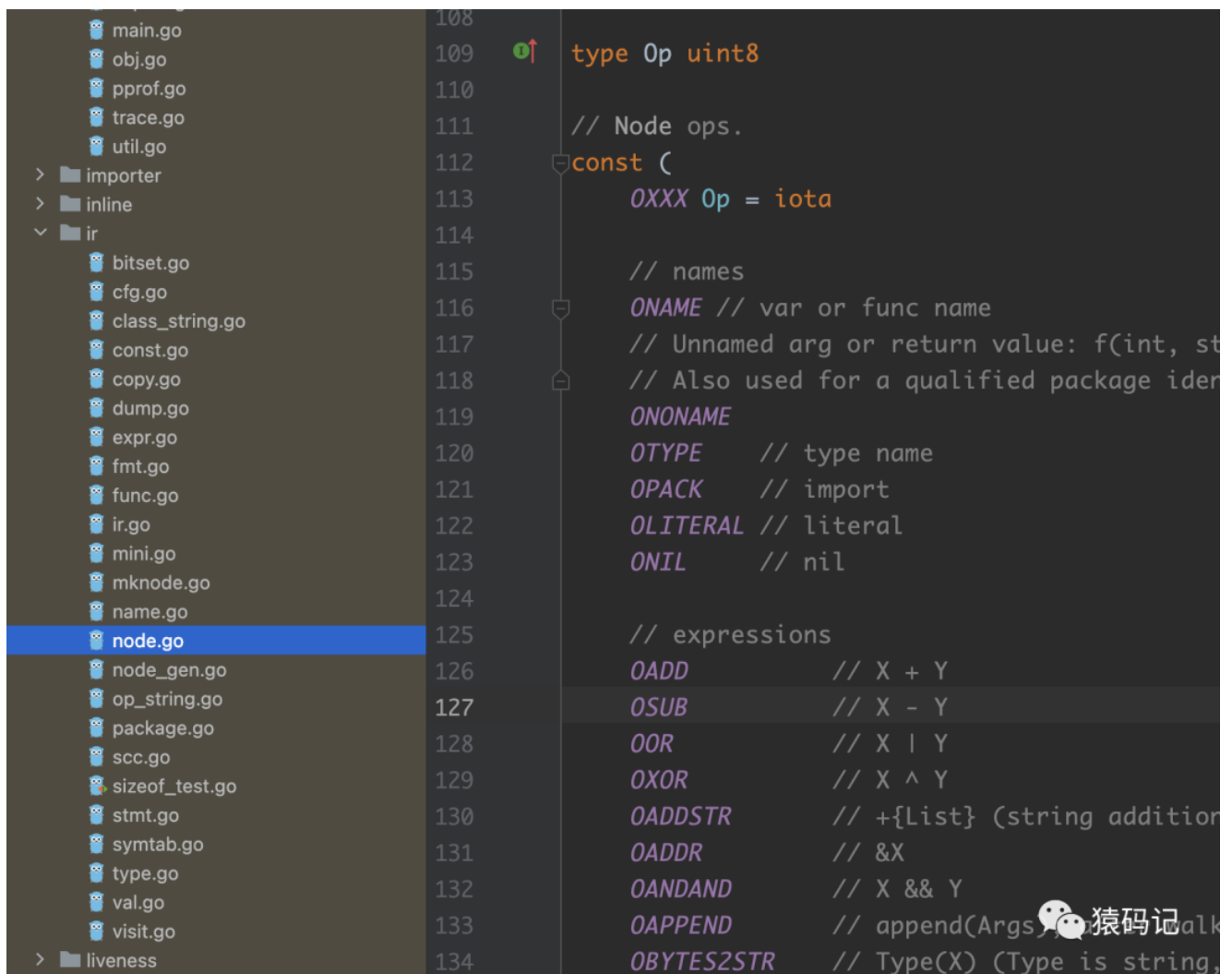
- 语法解析阶段的目的: 解析并识别可能出现的语法错误.

4. 抽象语法树构建

编译器前端必须构建程序的中间表示形式，以便在编译器中间阶段及后端使用。抽象语法树 [Abstract Syntax Tree, AST] 是一种常见的树状结构的中间态。

在 Go 语言源文件中的任何一种 `import`、`type`、`const`、`func` 声明都是一个根节点，在根节点下包含当前声明的子节点。

通过使用 `decls` 函数，将源文件中的所有声明语句转为节点数组。每个节点都包含了当前节点属性的 `Op` 字段，以 0 开头。与词法解析阶段中的 `token` 相同的是，`Op` 字段也是一个整数。不同的是，每个 `Op` 字段都包含了语义信息，如下图：



```
108
109 type Op uint8
110
111 // Node ops.
112 const (
113     OXXX Op = iota
114
115     // names
116     ONAME // var or func name
117     // Unnamed arg or return value: f(int, st
118     // Also used for a qualified package ider
119     ONONAME
120     OTYPE // type name
121     OPACK // import
122     OLITERAL // literal
123     ONIL // nil
124
125     // expressions
126     OADD // X + Y
127     OSUB // X - Y
128     OOR // X | Y
129     OXOR // X ^ Y
130     OADDSTR // +{List} (string addition
131     OADDR // &X
132     OANDAND // X && Y
133     OAPPEND // append(Args)
134     OBYTES2STR // Type(X) (Type is string,
```

《Go语言底层原理剖析》这本书中使用的 Go 是老版本，我使用的版本是 Go.18，所以源码路径会有所出入。

还是以 `a :=b+c` (12) 为例，该赋值语句最终会变为如图1-6所示的抽象语法树。节点之间具有从上到下的层次结构和依赖关系。

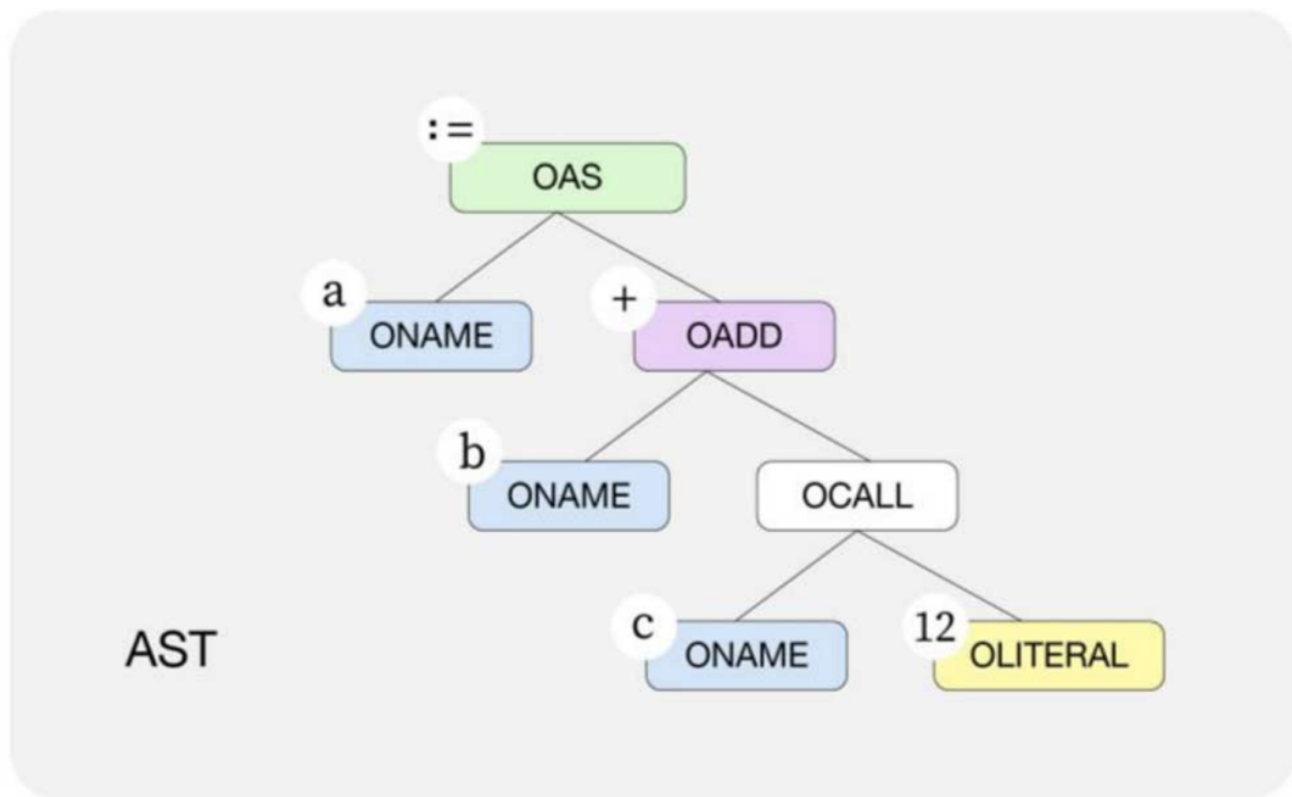


图1-6 抽象语法树

猿码记

4.1 归纳总结

- 了解 Go 程序从源码到抽象树生成的逻辑步骤；

5. 类型检查

完成抽象语法树的初步构建后，就进入类型检查阶段，遍历节点树并决定节点的类型。节点的类型判断有以下两种情况：

- 明确指定的类型：在语法中明确指定，例如 `var a int`。
- 需要通过编译器类型推断得到的类型：例如，`a := 1` 中的变量 `a` 与常量 `1` 都未直接声明类型，编译器会自动推断出节点常量 `1` 的类型为 `TINT`，并自动推断出 `a` 的类型为 `TINT`。

在类型检查阶段，会对一些类型做特别的语法或语义检查，例如：

- 引用的结构体字段是否是大写可导出的？
- 数组的访问是否超过了其长度？
- 数组的索引是不是正整数？

除此之外，在类型检查阶段还会进行其他工作。例如：计算编译时常量、将标识符与声明绑定等；

6. 变量捕获

类型检查阶段完成后，Go 语言编译器将对抽象语法树进行分析及重构，从而完成一系列优化。

变量捕获主要是针对闭包场景而言的，由于闭包函数中可能引用闭包外的变量，因此变量捕获需要明确在闭包中通过值引用或地址引用的方式来捕获变量。

6.1 举个例子

```
func main() {  
    a := 1  
    b := 2  
    // 使用闭包  
    go func() {  
        fmt.Println(a, b)  
    }()  
    a = 99  
}
```

上面例子中，在闭包内引入了闭包外的 `a`、`b` 变量，由于变量 `a` 在闭包之后又进行了其他赋值操作，因此在闭包中，`a`、`b` 变量的引用方式会有所不同。通过如下方式查看当前程序闭包变量捕获的情况：

```
$ go tool compile -m=2 main.go | grep capturing  
main.go:8:2: main capturing by ref: a (addr=false assign=true width=8)  
main.go:9:2: main capturing by value: b (addr=false assign=false width=8)
```

上面输出说明：

- `by ref: a` : `a` 采取 `ref` 引用传递的方式,
- `by value: b` : `b` 采取了值传递的方式。
- `assign=true` : 代表变量 `a` 在闭包完成后又进行了赋值操作。

7. 函数内联

函数内联指将较小的函数直接组合进调用者的函数。这是现代编译器优化的一种核心技术。

7.1 优点

函数内联的优势在于,可以减少函数调用带来的开销。对于 Go 语言来说,函数调用的成本在于:参数与返回值栈复制、较小的栈寄存器开销以及函数序言部分的检查栈扩容 (Go 语言中的栈是可以动态扩容的);

7.2 性能对比

下面通过写一段程序,来对比函数内联和不内联的性能;

```
package tests

import "testing"

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

// 使用了函数内联

func BenchmarkUseOnline(b *testing.B) {
    var a = 10
    for i := 0; i < b.N; i++ {
        // 进行大小计算
        max(a, i)
    }
}
```

```
//go:noinline

func maxNotOnline(a, b int) int {
    if a > b {
        return a
    }
    return b
}

// 使用了函数内联

func BenchmarkNotUseOnline(b *testing.B) {
    var a = 10
    for i := 0; i < b.N; i++ {
        // 进行大小计算
        maxNotOnline(a, i)
    }
}
```

运行测试:

```
$ go test -bench=. tests/func_test.go
BenchmarkUseOnline-12      1000000000      0.2577 ns/op
BenchmarkNotUseOnline-12  774045170      1.558 ns/op
```

从上面运行结果可以看出使用了函数内联的方法，比不使用的快了近三倍；

`go: noinline` : 代表当前函数是禁止进行函数内联优化的。

7.3 不使用内联

Go 语言编译器会计算函数内联花费的成本，只有执行相对简单的函数时才会内联。函数内联的核心逻辑位于 `gc/inl.go` 中。以下情况都不会使用函数内联：

- 当函数内部有 `for`、`range`、`go`、`select` 等语句时，该函数不会被内联，
- 当函数执行过于复杂（例如太多的语句或者函数为**递归函数**）时，也不会执行内联。
- 如果函数前的注释中有 `go: noinline` 标识，则该函数不会执行内联

如果希望程序中所有的函数都不执行内联操作，那么可以添加编译器选项 `-l` ,如下：

```
# go build
$ go build -gcflags="-l" main.go
# go tool命令
$ go tool compile -l main.go
```

7.4 不内联原因

在调试时，可以使用 `go tool compile -m=2` 来打印调试信息，并输出不可以内联的原因,如下代码：

```
package tests

import (
    "testing"
)
// 使用递归
func fib(i int) int {
    if i < 2 {
        return i
    }
    return fib(i-1) + fib(i-2)
}
func TestRun(t *testing.T) {
    i := 10
    fib(i)
}
```

打印调试信息：

```
$ go tool compile -m=2 tests/funcLine_test.go
tests/funcLine_test.go:8:6: cannot inline fib: recursive
```

```
tests/funcLine_test.go:14:6: can inline TestRun with cost 65 as: func(*testing.T) { i := 10; fib(10) }
tests/funcLine_test.go:14:14: t does not escape
```

当在编译时加入 `-m=2` 标志时，可以打印出函数的内联调试信息。可以看出 `fib` 函数为递归函数，所以不能被内联。

8. 逃逸分析

逃逸分析是 Go 语言中重要的优化阶段，用于标识变量内存应该被分配在栈区还是堆区。

在传统的 C 或 C++ 语言中，开发者经常会犯的错误是函数返回了一个栈上的对象指针，在函数执行完成，栈被销毁后，继续访问被销毁栈上的对象指针，导致出现问题。

Go 语言能够通过编译时的逃逸分析识别这种问题，自动将该变量放置到堆区，并借助 Go 运行时的垃圾回收机制自动释放内存。编译器会尽可能地将变量放置到栈中，因为栈中的对象随着函数调用结束会被自动销毁，减轻运行时分配和垃圾回收的负担。

8.1 分配原则

在 Go 语言中，开发者模糊了栈区与堆区的差别，不管是字符串、数组字面量，还是通过 `new`、`make` 标识符创建的对象，都既可能被分配到栈中，也可能被分配到堆中。分配时，遵循以下两个原则：

- 原则1：指向栈上对象的指针不能被存储到堆中
- 原则2：指向栈上对象的指针不能超过该栈对象的生命周期

举个例子：

```
// 全局变量
var a *int

func TestVarEscape(t *testing.T) {
    // 局部变量
```

```
b := 1
// 引用变量b地址
a = &b
}
```

运行测试:

```
$ go tool compile -m=2 tests/var_test.go
tests/var_test.go:10:6: can inline TestVarEscape with cost 9 as: func(*testing.T) { b := 1; a = &b
tests/var_test.go:12:2: b escapes to heap:
tests/var_test.go:12:2:   flow: {heap} = &b:
tests/var_test.go:12:2:   from &b (address-of) at tests/var_test.go:14:6
tests/var_test.go:12:2:   from a = &b (assign) at tests/var_test.go:14:4
tests/var_test.go:10:20: t does not escape
tests/var_test.go:12:2: moved to heap: b # 变量b最终被分配到堆内存
```

在上例中，变量 `a` 为全局变量，是一个指针。在函数中，全局变量 `a` 引用了局部变量 `b` 的地址。

如果变量 `b` 被分配到栈中，那么最终程序将违背原则2，因此变量 `b` 最终将被分配到堆中。

9.闭包重写

在前面的阶段，编译器完成了闭包变量的捕获用于决定是通过指针引用还是值引用的方式传递外部变量。在完成逃逸分析后，下一个优化的阶段为闭包重写，闭包重写分为以下两种情况:

- **闭包定义后被立即调用:** 这种情况下,闭包只能被调用一次,可以将闭包转换为普通函数的调用形式。
- **包定义后不被立即调用:** 同一个闭包可能被调用多次，这时需要创建闭包对象。

9.1 重写示例

下面示例展示的是闭包函数，重写后的样子。

```
// 闭包定义后被立即调用
func todo() {
    a := 1
    func() {
        fmt.Println(a)
        a = 2
    }()
}
```

闭包重写后:

```
func todo() {
    a := 1
    func1(&a)
    fmt.Println("aa:", a)
}
func func1(a *int) {
    fmt.Println(*a)
    *a = 2
}
```

10.遍历函数

闭包重写后，需要遍历函数。在该阶段会识别出声明但是并未被使用的变量，遍历函数中的声明和表达式，将某些代表操作的节点转换为运行时的具体函数执行。

例如: 获取 `map` 中的值会被转换为运行时 `mapaccess2_fast64` 函数:

```
v,ok := m["foo"]
// 转化为
tmp,ok := runtime.mapaccess2_fast64(typeOf(m),m,"foo")
v := *tmp
```

字符串变量的拼接会被转换为调用运行时 `concatstrings` 函数。对于 `new` 操作，如果变量发生了逃逸，那么最终会调用运行时 `newobject` 函数将变量分配到堆区。`for...range` 语句会重写为更简单的 `for` 语句形式。

11. SSA生成

遍历函数后，编译器会将抽象语法树转换为下一个重要的中间表示形态，称为 `SSA (Static Single Assignment, 静态单赋值)`。`SSA` 被大多数现代的编译器（包括 `GCC`和`LLVM`）使用,在 `Go 1.7` 中被正式引入并替换了之前的编译器后端，用于最终生成更有效的机器码。

在 `SSA` 生成阶段，每个变量在声明之前都需要被定义，并且，**每个变量只会被赋值一次**。

11.1 SSA阶段作用

`SSA` 生成阶段是编译器进行后续优化的保证，例如**常量传播 (Constant Propagation)**、**无效代码清除、消除冗余、强度降低 (Strength Reduction)** 等。

在 `SSA` 阶段，编译器先执行与特定指令集无关的优化，再执行与特定指令集有关的优化，并最终生成与特定指令集有关的指令和寄存器分配方式。

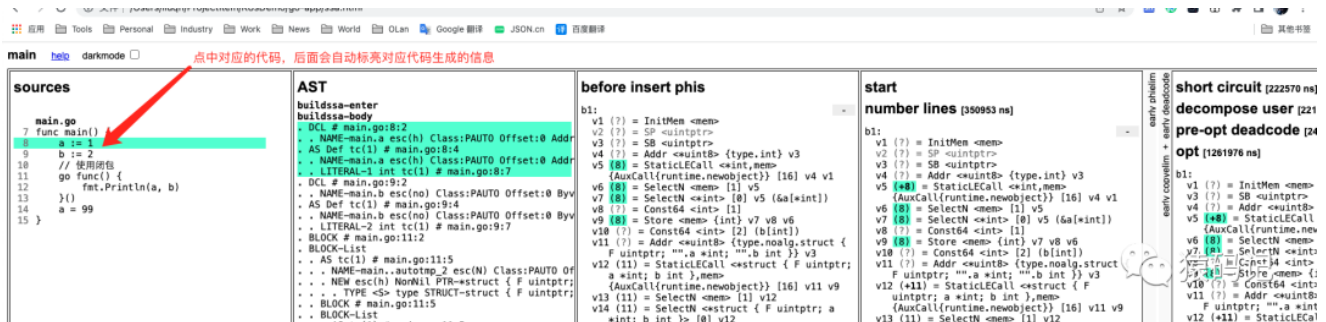
- **`SSA lower`阶段之后**：开始执行与特定指令集有关的重写与优化。
- **`genssa`阶段**：编译器会生成与单个指令对应的 `Prog` 结构。

11.2 怎么生成SSA

`Go` 语言提供了强有力的工具查看 `SSA` 初始及其后续优化阶段生成的代码片段，可以通过在编译时指定 `GOSSAFUNC=main` 实现,使用如下：

```
$ GOSSAFUNC=main go tool compile main.go
dumped SSA to /Users/liuqh/ProjectItem/K8sDemo/go-app/ssa.html
```

打开ssa.html



上述图片展示了 SSA的初始阶段、优化阶段、最终阶段的代码片段

12. 机器码生成(汇编器)

在 SSA 后，编译器将调用与特定指令集有关的 汇编器 (Assembler) 生成 obj 文件，obj 文件作为 链接器 (Linker) 的输入，生成二进制可执行文件。

汇编和链接是编译器后端与特定指令集有关的阶段。由于历史原因，Go 语言的汇编器基于了不太常见的 plan9 汇编器的输入形式。需要注意的是，输入汇编器中的汇编指令不是机器码的表现形式，其仍然是人类可读的底层抽象。

12.1 源程序转汇编代码

```
package main

import "fmt"

func main() {

    fmt.Println("hello word")

}
```

转成汇编代码:


```
$ go tool compile -S main.go

"".main STEXT size=103 args=0x0 locals=0x40 funcid=0x0 align=0x0
    0x0000 00000 (main.go:5)      TEXT    "".main(SB), ABIInternal, $64-0
    0x0000 00000 (main.go:5)      CMPQ    SP, 16(R14)
    0x0004 00004 (main.go:5)      PCDATA  $0, $-2
    0x0004 00004 (main.go:5)      JLS     92
    0x0006 00006 (main.go:5)      PCDATA  $0, $-1
    0x0006 00006 (main.go:5)      SUBQ    $64, SP
    0x000a 00010 (main.go:5)      MOVQ    BP, 56(SP)
    0x000f 00015 (main.go:5)      LEAQ    56(SP), BP
    ....
```



微信搜一搜
猿码记

3分钟前点击
了阅读原文

戳“阅读原文”我们一起进步

收录于合集 #Go 101

上一篇

Uber Go规范(二): Slice、Struct、Map

下一篇

Go底层探索(二):字符串

Read more

People who liked this content also liked

Python常用库(二):数学计算

猿码记



Redis从理论到实战：用Redis解决缓存穿透、缓存击穿问题（提供解决方案）

