Go 学习笔记

第 4 版

好好学习　天天向上

前言

在不侵犯作者个人权利的前提下，可自由散播本书。

- 下载：不定期更新，https://github.com/qyuhen/book。
- 联系：qyuhen@hotmail.com

雨痕 二〇一四年末

更新

2012-01-11 开始学习 Go。
2012-01-15 第一版，基于 R60。
2012-03-29 升级到 1.0。
2012-06-15 升级到 1.0.2。
2013-03-26 升级到 1.1。
2013-12-12 第二版，基于 1.2。
2014-05-22 第三版，基于 1.3。
2014-12-20 第四版，基于 1.4。

目录

第一部分 语言

第 1 章 类型

1.1 变量

Go 是静态类型语言，不能在运行期改变变量类型。

使用关键字 var 定义变量，自动初始化为零值。如果提供初始化值，可省略变量类型，由编译器自动推断。

```
var x int
var f float32 = 1.6
var s = "abc"
```

在函数内部，可用更简略的 ":=" 方式定义变量。

```
func main() {
    x := 123        // 注意检查，是定义新局部变量，还是修改全局变量。该方式容易造成错误。
}
```

可一次定义多个变量。

```
var x, y, z int
var s, n = "abc", 123

var (
    a int
    b float32
)

func main() {
    n, s := 0x1234, "Hello, World!"
    println(x, s, n)
}
```

多变量赋值时，先计算所有相关值，然后再从左到右依次赋值。

```
data, i := [3]int{0, 1, 2}, 0
i, data[i] = 2, 100                // (i = 0) -> (i = 2), (data[0] = 100)
```

特殊只写变量 "_"，用于忽略值占位。

```
func test() (int, string) {
    return 1, "abc"
}

func main() {
    _, s := test()
    println(s)
}
```

编译器会将未使用的局部变量当做错误。

```
var s string      // 全局变量没问题。

func main() {
    i := 0        // Error: i declared and not used。(可使用 "_ = i" 规避)
}
```

注意重新赋值与定义新同名变量的区别。

```
s := "abc"
println(&s)

s, y := "hello", 20      // 重新赋值：与前 s 在同一层次的代码块中，且有新的变量被定义。
println(&s, y)           // 通常函数多返回值 err 会被重复使用。

{
    s, z := 1000, 30     // 定义新同名变量：不在同一层次代码块。
    println(&s, z)
}
```

输出：

```
0x2210230f30
0x2210230f30 20
0x2210230f18 30
```

1.2 常量

常量值必须是编译期可确定的数字、字符串、布尔值。

```
const x, y int = 1, 2          // 多常量初始化
const s = "Hello, World!"      // 类型推断

const (                        // 常量组
    a, b      = 10, 100
    c    bool = false
```

```
)

func main() {
    const x = "xxx"             // 未使用局部常量不会引发编译错误。
}
```

不支持 1UL、2LL 这样的类型后缀。

在常量组中，如不提供类型和初始化值，那么视作与上一常量相同。

```
const (
    s   = "abc"
    x               // x = "abc"
)
```

常量值还可以是 len、cap、unsafe.Sizeof 等编译期可确定结果的函数返回值。

```
const (
    a   = "abc"
    b   = len(a)
    c   = unsafe.Sizeof(b)
)
```

如果常量类型足以存储初始化值，那么不会引发溢出错误。

```
const (
    a   byte = 100       // int to byte
    b   int  = 1e20      // float64 to int, overflows
)
```

枚举

关键字 iota 定义常量组中从 0 开始按行计数的自增枚举值。

```
const (
    Sunday = iota     // 0
    Monday            // 1，通常省略后续行表达式。
    Tuesday           // 2
    Wednesday         // 3
    Thursday          // 4
    Friday            // 5
    Saturday          // 6
)
```

```
const (
    _          = iota                  // iota = 0
    KB  int64 = 1 << (10 * iota)       // iota = 1
    MB                                 // 与 KB 表达式相同，但 iota = 2
    GB
    TB
)
```

在同一常量组中，可以提供多个 iota，它们各自增长。

```
const (
    A, B = iota, iota << 10   // 0, 0 << 10
    C, D                      // 1, 1 << 10
)
```

如果 iota 自增被打断，须显式恢复。

```
const (
    A   = iota      // 0
    B               // 1
    C   = "c"       // c
    D               // c，与上一行相同。
    E   = iota      // 4，显式恢复。注意计数包含了 C、D 两行。
    F               // 5
)
```

可通过自定义类型来实现枚举类型限制。

```
type Color int

const (
    Black Color = iota
    Red
    Blue
)

func test(c Color) {}

func main() {
    c := Black
    test(c)

    x := 1
    test(x) // Error: cannot use x (type int) as type Color in function argument
```

```
    test(1) // 常量会被编译器自动转换。
}
```

1.3 基本类型

更明确的数字类型命名，支持 Unicode，支持常用数据结构。

类型	长度	默认值	说明
bool	1	false	
byte	1	0	uint8
rune	4	0	Unicode Code Point, int32
int, uint	4 或 8	0	32 或 64 位
int8, uint8	1	0	-128 ~ 127, 0 ~ 255
int16, uint16	2	0	-32768 ~ 32767, 0 ~ 65535
int32, uint32	4	0	-21亿 ~ 21 亿, 0 ~ 42 亿
int64, uint64	8	0	
float32	4	0.0	
float64	8	0.0	
complex64	8		
complex128	16		
uintptr	4 或 8		足以存储指针的 uint32 或 uint64 整数
array			值类型
struct			值类型
string		""	UTF-8 字符串
slice		nil	引用类型
map		nil	引用类型
channel		nil	引用类型
interface		nil	接口
function		nil	函数

支持八进制、十六进制，以及科学记数法。标准库 math 定义了各数字类型取值范围。

```
a, b, c, d := 071, 0x1F, 1e9, math.MinInt16
```

空指针值 nil，而非 C/C++ NULL。

1.4 引用类型

引用类型包括 slice、map 和 channel。它们有复杂的内部结构，除了申请内存外，还需要初始化相关属性。

内置函数 new 计算类型大小，为其分配零值内存，返回指针。而 make 会被编译器翻译成具体的创建函数，由其分配内存和初始化成员结构，返回对象而非指针。

```
a := []int{0, 0, 0}     // 提供初始化表达式。
a[1] = 10

b := make([]int, 3)     // makeslice
b[1] = 10

c := new([]int)
c[1] = 10               // Error: invalid operation: c[1] (index of type *[]int)
```

有关引用类型具体的内存布局，可参考后续章节。

1.5 类型转换

不支持隐式类型转换，即便是从窄向宽转换也不行。

```
var b byte = 100
// var n int = b        // Error: cannot use b (type byte) as type int in assignment
var n int = int(b)      // 显式转换
```

使用括号避免优先级错误。

```
*Point(p)               // 相当于 *(Point(p))
(*Point)(p)
<-chan int(c)           // 相当于 <-(chan int(c))
(<-chan int)(c)
```

同样不能将其他类型当 bool 值使用。

```
a := 100
if a {                  // Error: non-bool a (type int) used as if condition
    println("true")
}
```

1.6 字符串

字符串是不可变值类型，内部用指针指向 UTF-8 字节数组。

- 默认值是空字符串 ""。
- 用索引号访问某字节，如 s[i]。
- 不能用序号获取字节元素指针，&s[i] 非法。
- 不可变类型，无法修改字节数组。
- 字节数组尾部不包含 NULL。

runtime.h

```
struct String
{
    byte*    str;
    intgo    len;
};
```

使用索引号访问字符 (byte)。

```
s := "abc"
println(s[0] == '\x61', s[1] == 'b', s[2] == 0x63)
```

输出：

```
true true true
```

使用 "`" 定义不做转义处理的原始字符串，支持跨行。

```
s := `a
b\r\n\x00
c`

println(s)
```

输出：

```
a
b\r\n\x00
c
```

连接跨行字符串时，"+" 必须在上一行末尾，否则导致编译错误。

```
s := "Hello, " +
     "World!"

s2 := "Hello, "
    + "World!"    // Error: invalid operation: + untyped string
```

支持用两个索引号返回子串。子串依然指向原字节数组，仅修改了指针和长度属性。

```
s := "Hello, World!"

s1 := s[:5]                  // Hello
s2 := s[7:]                  // World!
s3 := s[1:5]                 // ello
```

单引号字符常量表示 Unicode Code Point，支持 \uFFFF、\U7FFFFFFF、\xFF 格式。对应 rune 类型，UCS-4。

```
func main() {
    fmt.Printf("%T\n", 'a')

    var c1, c2 rune = '\u6211', '们'
    println(c1 == '我', string(c2) == "\xe4\xbb\xac")
}
```

输出：

```
int32                   // rune 是 int32 的别名
true true
```

要修改字符串，可先将其转换成 []rune 或 []byte，完成后再转换为 string。无论哪种转换，都会重新分配内存，并复制字节数组。

```
func main() {
    s := "abcd"
    bs := []byte(s)

    bs[1] = 'B'
    println(string(bs))

    u := "电脑"
```

```
    us := []rune(u)

    us[1] = '话'
    println(string(us))
}
```

输出:

```
aBcd
电话
```

用 for 循环遍历字符串时，也有 byte 和 rune 两种方式。

```
func main() {
    s := "abc汉字"

    for i := 0; i < len(s); i++ {       // byte
        fmt.Printf("%c,", s[i])
    }

    fmt.Println()

    for _, r := range s {               // rune
        fmt.Printf("%c,", r)
    }
}
```

输出:

```
a,b,c,æ,±,,å,-,,
a,b,c,汉,字,
```

1.7 指针

支持指针类型 *T，指针的指针 **T，以及包含包名前缀的 *<package>.T。

- 默认值 nil，没有 NULL 常量。
- 操作符 "&" 取变量地址，"*" 透过指针访问目标对象。
- 不支持指针运算，不支持 "->" 运算符，直接用 "." 访问目标成员。

```
func main() {
    type data struct{ a int }

    var d = data{1234}
    var p *data

    p = &d
```

```
    fmt.Printf("%p, %v\n", p, p.a)   // 直接用指针访问目标对象成员，无须转换。
}
```

输出：

```
0x2101ef018, 1234
```

不能对指针做加减法等运算。

```
x := 1234
p := &x
p++          // Error: invalid operation: p += 1 (mismatched types *int and int)
```

可以在 unsafe.Pointer 和任意类型指针间进行转换。

```
func main() {
    x := 0x12345678

    p := unsafe.Pointer(&x)         // *int -> Pointer
    n := (*[4]byte)(p)              // Pointer -> *[4]byte

    for i := 0; i < len(n); i++ {
        fmt.Printf("%X ", n[i])
    }
}
```

输出：

```
78 56 34 12
```

返回局部变量指针是安全的，编译器会根据需要将其分配在 GC Heap 上。

```
func test() *int {
    x := 100
    return &x            // 在堆上分配 x 内存。但在内联时，也可能直接分配在目标栈。
}
```

将 Pointer 转换成 uintptr，可变相实现指针运算。

```
func main() {
    d := struct {
        s   string
        x   int
    }{"abc", 100}

    p := uintptr(unsafe.Pointer(&d))   // *struct -> Pointer -> uintptr
    p += unsafe.Offsetof(d.x)          // uintptr + offset
```

```
    p2 := unsafe.Pointer(p)            // uintptr -> Pointer
    px := (*int)(p2)                   // Pointer -> *int
    *px = 200                          // d.x = 200

    fmt.Printf("%#v\n", d)
}
```

输出：

```
struct { s string; x int }{s:"abc", x:200}
```

注意：GC 把 uintptr 当成普通整数对象，它无法阻止 "关联" 对象被回收。

1.8 自定义类型

可将类型分为命名和未命名两大类。命名类型包括 bool、int、string 等，而 array、slice、map 等和具体元素类型、长度等有关，属于未命名类型。

具有相同声明的未命名类型被视为同一类型。

- 具有相同基类型的指针。
- 具有相同元素类型和长度的 array。
- 具有相同元素类型的 slice。
- 具有相同键值类型的 map。
- 具有相同元素类型和传送方向的 channel。
- 具有相同字段序列 (字段名、类型、标签、顺序) 的匿名 struct。
- 签名相同 (参数和返回值，不包括参数名称) 的 function。
- 方法集相同 (方法名、方法签名相同，和次序无关) 的 interface。

```
var a struct { x int `a` }
var b struct { x int `ab` }

// cannot use a (type struct { x int "a" }) as type struct { x int "ab" } in assignment
b = a
```

可用 type 在全局或函数内定义新类型。

```
func main() {
    type bigint int64

    var x bigint = 100
    println(x)
}
```

新类型不是原类型的别名，除拥有相同数据存储结构外，它们之间没有任何关系，不会持有原类型任何信息。除非目标类型是未命名类型，否则必须显式转换。

```
x := 1234
var b bigint = bigint(x)          // 必须显式转换，除非是常量。
var b2 int64 = int64(b)

var s myslice = []int{1, 2, 3}    // 未命名类型，隐式转换。
var s2 []int = s
```

第 2 章 表达式

2.1 保留字

语言设计简练，保留字不多。

```
break        default        func         interface      select
case         defer          go           map            struct
chan         else           goto         package        switch
const        fallthrough    if           range          type
continue     for            import       return         var
```

2.2 运算符

全部运算符、分隔符，以及其他符号。

```
+       &       +=       &=       &&       ==       !=       (       )
-       |       -=       |=       ||       <        <=       [       ]
*       ^       *=       ^=       <-       >        >=       {       }
/       <<      /=       <<=      ++       =        :=       ,       ;
%       >>      %=       >>=      --       !        ...      .       :
        &^               &^=
```

运算符结合律全部从左到右。

```
 优先级        运算符                                              说明
------------+-------------------------------------------------+------------------------------
 high        *      /      &      <<     >>     &      &^
             +      -      |      ^
             ==     !=     <      <=     <      >=
             <-                                                  channel
             &&
 low         ||
```

简单位运算演示。

```
0110 &  1011 = 0010       AND      都为 1。
0110 |  1011 = 1111       OR       至少一个为 1。
0110 ^  1011 = 1101       XOR      只能一个为 1。
0110 &^ 1011 = 0100       AND NOT  清除标志位。
```

标志位操作。

```
a := 0
a |= 1 << 2            // 0000100: 在 bit2 设置标志位。
a |= 1 << 6            // 1000100: 在 bit6 设置标志位
a = a &^ (1 << 6)      // 0000100: 清除 bit6 标志位。
```

不支持运算符重载。尤其需要注意，"++"、"--" 是语句而非表达式。

```
n := 0
p := &n

// b := n++           // syntax error
// if n++ == 1 {}     // syntax error
// ++n                // syntax error

n++
*p++                  // (*p)++
```

没有 "~"，取反运算也用 "^"。

```
x := 1
x, ^x                 // 0001, -0010
```

2.3 初始化

初始化复合对象，必须使用类型标签，且左大括号必须在类型尾部。

```
// var a struct { x int } = { 100 }   // syntax error

// var b []int = { 1, 2, 3 }          // syntax error

// c := struct {x int; y string}      // syntax error: unexpected semicolon or newline
// {
// }

var a = struct{ x int }{100}
var b = []int{1, 2, 3}
```

初始化值以 "," 分隔。可以分多行，但最后一行必须以 "," 或 "}" 结尾。

```
a := []int{

```

```
    1,
    2           // Error: need trailing comma before newline in composite literal
}

a := []int{
    1,
    2,          // ok
}

b := []int{
    1,
    2 }         // ok
```

2.4 控制流

2.4.1 IF

很特别的写法：

- 可省略条件表达式括号。
- 支持初始化语句，可定义代码块局部变量。
- 代码块左大括号必须在条件表达式尾部。

```
x := 0

// if x > 10                // Error: missing condition in if statement
// {
// }

if n := "abc"; x > 0 {      // 初始化语句未必就是定义变量，比如 println("init") 也是可以的。
    println(n[2])
} else if x < 0 {           // 注意 else if 和 else 左大括号位置。
    println(n[1])
} else {
    println(n[0])
}
```

不支持三元操作符 "a > b ? a : b"。

2.4.2 For

支持三种循环方式，包括类 while 语法。

```
s := "abc"

for i, n := 0, len(s); i < n; i++ {   // 常见的 for 循环，支持初始化语句。
    println(s[i])
}

n := len(s)
for n > 0 {                           // 替代 while (n > 0) {}
    println(s[n])                     // 替代 for (; n > 0;) {}
    n--
}

for {                                 // 替代 while (true) {}
    println(s)                        // 替代 for (;;) {}
}
```

不要期望编译器能理解你的想法，在初始化语句中计算出全部结果是个好主意。

```
func length(s string) int {
    println("call length.")
    return len(s)
}

func main() {
    s := "abcd"

    for i, n := 0, length(s); i < n; i++ {   // 避免多次调用 length 函数。
        println(i, s[i])
    }
}
```

输出：

```
call length.
0 97
1 98
2 99
3 100
```

2.4.3 Range

类似迭代器操作，返回 (索引, 值) 或 (键, 值)。

```
                    1st value           2nd value
-----------------+-------------------+-----------------+-------------------
 string              index               s[index]           unicode, rune
 array/slice         index               s[index]
 map                 key                 m[key]
 channel             element
```

可忽略不想要的返回值，或用 "_" 这个特殊变量。

```
s := "abc"

for i := range s {                  // 忽略 2nd value，支持 string/array/slice/map。
    println(s[i])
}

for _, c := range s {               // 忽略 index。
    println(c)
}

for range s {                       // 忽略全部返回值，仅迭代。
    ...
}

m := map[string]int{"a": 1, "b": 2}

for k, v := range m {               // 返回 (key, value)。
    println(k, v)
}
```

注意，range 会复制对象。

```
a := [3]int{0, 1, 2}

for i, v := range a {             // index、value 都是从复制品中取出。

    if i == 0 {                   // 在修改前，我们先修改原数组。
        a[1], a[2] = 999, 999
        fmt.Println(a)            // 确认修改有效，输出 [0, 999, 999]。
    }

    a[i] = v + 100                // 使用复制品中取出的 value 修改原数组。
}

fmt.Println(a)                    // 输出 [100, 101, 102]。
```

建议改用引用类型，其底层数据不会被复制。

```
s := []int{1, 2, 3, 4, 5}

for i, v := range s {       // 复制 struct slice { pointer, len, cap }。

    if i == 0 {
        s = s[:3]           // 对 slice 的修改，不会影响 range。
        s[2] = 100          // 对底层数据的修改。
    }

    println(i, v)
}
```

输出：

```
0 1
1 2
2 100
3 4
4 5
```

另外两种引用类型 map、channel 是指针包装，而不像 slice 是 struct。

2.4.4 Switch

分支表达式可以是任意类型，不限于常量。可省略 break，默认自动终止。

```
x := []int{1, 2, 3}
i := 2

switch i {
    case x[1]:
        println("a")
    case 1, 3:
        println("b")
    default:
        println("c")
}
```

输出：

```
a
```

如需要继续下一分支，可使用 fallthrough，但不再判断条件。

```
x := 10
```

```
switch x {
case 10:
    println("a")
    fallthrough
case 0:
    println("b")
}
```

输出：

```
a
b
```

省略条件表达式，可当 if...else if...else 使用。

```
switch {
    case x[1] > 0:
        println("a")
    case x[1] < 0:
        println("b")
    default:
        println("c")
}

switch i := x[2]; {           // 带初始化语句
    case i > 0:
        println("a")
    case i < 0:
        println("b")
    default:
        println("c")
}
```

2.4.5 Goto, Break, Continue

支持在函数内 goto 跳转。标签名区分大小写，未使用标签引发错误。

```
func main() {
    var i int
    for {
        println(i)
        i++
        if i > 2 { goto BREAK }
    }
```

```
BREAK:
    println("break")

EXIT:                   // Error: label EXIT defined and not used
}
```

配合标签，break 和 continue 可在多级嵌套循环中跳出。

```
func main() {
L1:
    for x := 0; x < 3; x++ {
L2:
        for y := 0; y < 5; y++ {
            if y > 2 { continue L2 }
            if x > 1 { break L1 }

            print(x, ":", y, " ")
        }

        println()
    }
}
```

输出：

```
0:0  0:1  0:2
1:0  1:1  1:2
```

附：break 可用于 for、switch、select，而 continue 仅能用于 for 循环。

```
x := 100

switch {
case x >= 0:
    if x == 0 { break }
    println(x)
}
```

第 3 章 函数

3.1 函数定义

不支持 嵌套 (nested)、重载 (overload) 和 默认参数 (default parameter)。

- 无需声明原型。
- 支持不定长变参。
- 支持多返回值。
- 支持命名返回参数。
- 支持匿名函数和闭包。

使用关键字 func 定义函数，左大括号依旧不能另起一行。

```
func test(x, y int, s string) (int, string) {      // 类型相同的相邻参数可合并。
    n := x + y                                     // 多返回值必须用括号。
    return n, fmt.Sprintf(s, n)
}
```

函数是第一类对象，可作为参数传递。建议将复杂签名定义为函数类型，以便于阅读。

```
func test(fn func() int) int {
    return fn()
}

type FormatFunc func(s string, x, y int) string    // 定义函数类型。

func format(fn FormatFunc, s string, x, y int) string {
    return fn(s, x, y)
}

func main() {
    s1 := test(func() int { return 100 })          // 直接将匿名函数当参数。

    s2 := format(func(s string, x, y int) string {
        return fmt.Sprintf(s, x, y)
    }, "%d, %d", 10, 20)

    println(s1, s2)
}
```

有返回值的函数，必须有明确的终止语句，否则会引发编译错误。

3.2 变参

变参本质上就是 slice。只能有一个，且必须是最后一个。

```
func test(s string, n ...int) string {
    var x int
    for _, i := range n {
        x += i
    }

    return fmt.Sprintf(s, x)
}

func main() {
    println(test("sum: %d", 1, 2, 3))
}
```

使用 slice 对象做变参时，必须展开。

```
func main() {
    s := []int{1, 2, 3}
    println(test("sum: %d", s...))
}
```

3.3 返回值

不能用容器对象接收多返回值。只能用多个变量，或 "_" 忽略。

```
func test() (int, int) {
    return 1, 2
}

func main() {
    // s := make([]int, 2)
    // s = test()           // Error: multiple-value test() in single-value context

    x, _ := test()
    println(x)
}
```

多返回值可直接作为其他函数调用实参。

```
func test() (int, int) {
    return 1, 2
}

func add(x, y int) int {
    return x + y
}

func sum(n ...int) int {
    var x int
    for _, i := range n {
        x += i
    }

    return x
}

func main() {
    println(add(test()))
    println(sum(test()))
}
```

命名返回参数可看做与形参类似的局部变量，最后由 return 隐式返回。

```
func add(x, y int) (z int) {
    z = x + y
    return
}

func main() {
    println(add(1, 2))
}
```

命名返回参数可被同名局部变量遮蔽，此时需要显式返回。

```
func add(x, y int) (z int) {
    {                       // 不能在一个级别，引发 "z redeclared in this block" 错误。
        var z = x + y
        // return           // Error: z is shadowed during return
        return z            // 必须显式返回。
    }
}
```

命名返回参数允许 defer 延迟调用通过闭包读取和修改。

```
func add(x, y int) (z int) {
    defer func() {
        z += 100
    }()

    z = x + y
    return
}

func main() {
    println(add(1, 2))     // 输出: 103
}
```

显式 return 返回前，会先修改命名返回参数。

```
func add(x, y int) (z int) {
    defer func() {
        println(z)         // 输出: 203
    }()

    z = x + y
    return z + 200         // 执行顺序: (z = z + 200) -> (call defer) -> (ret)
}

func main() {
    println(add(1, 2))     // 输出: 203
}
```

3.4 匿名函数

匿名函数可赋值给变量，做为结构字段，或者在 channel 里传送。

```
// --- function variable ---

fn := func() { println("Hello, World!") }
fn()

// --- function collection ---

fns := [](func(x int) int){
    func(x int) int { return x + 1 },
    func(x int) int { return x + 2 },
```

```
}

println(fns[0](100))

// --- function as field ---

d := struct {
    fn func() string
}{
    fn: func() string { return "Hello, World!" },
}

println(d.fn())

// --- channel of function ---

fc := make(chan func() string, 2)
fc <- func() string { return "Hello, World!" }
println((<-fc)())
```

闭包复制的是原对象指针，这就很容易解释延迟引用现象。

```
func test() func() {
    x := 100
    fmt.Printf("x (%p) = %d\n", &x, x)

    return func() {
        fmt.Printf("x (%p) = %d\n", &x, x)
    }
}

func main() {
    f := test()
    f()
}
```

输出：

```
x (0x2101ef018) = 100
x (0x2101ef018) = 100
```

在汇编层面，test 实际返回的是 FuncVal 对象，其中包含了匿名函数地址、闭包对象指针。当调用匿名函数时，只需以某个寄存器传递该对象即可。

```
FuncVal { func_address, closure_var_pointer ... }
```

3.5 延迟调用

关键字 defer 用于注册延迟调用。这些调用直到 ret 前才被执行，通常用于释放资源或错误处理。

```
func test() error {
    f, err := os.Create("test.txt")
    if err != nil { return err }

    defer f.Close()                 // 注册调用，而不是注册函数。必须提供参数，哪怕为空。

    f.WriteString("Hello, World!")
    return nil
}
```

多个 defer 注册，按 FILO 次序执行。哪怕函数或某个延迟调用发生错误，这些调用依旧会被执行。

```
func test(x int) {
    defer println("a")
    defer println("b")

    defer func() {
        println(100 / x)            // div0 异常未被捕获，逐步往外传递，最终终止进程。
    }()

    defer println("c")
}

func main() {
    test(0)
}
```

输出：

```
c
b
a
panic: runtime error: integer divide by zero
```

延迟调用参数在注册时求值或复制，可用指针或闭包 "延迟" 读取。

```
func test() {
    x, y := 10, 20

    defer func(i int) {
        println("defer:", i, y)     // y 闭包引用
```

```
    }(x)                            // x 被复制

    x += 10
    y += 100
    println("x =", x, "y =", y)
}
```

输出：

```
x = 20 y = 120
defer: 10 120
```

滥用 defer 可能会导致性能问题，尤其是在一个 "大循环" 里。

```
var lock sync.Mutex

func test() {
    lock.Lock()
    lock.Unlock()
}

func testdefer() {
    lock.Lock()
    defer lock.Unlock()
}

func BenchmarkTest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        test()
    }
}

func BenchmarkTestDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        testdefer()
    }
}
```

输出：

```
BenchmarkTest       50000000          43 ns/op
BenchmarkTestDefer  20000000         128 ns/op
```

3.6 错误处理

没有结构化异常，使用 panic 抛出错误，recover 捕获错误。

```
func test() {
```

```
    defer func() {
        if err := recover(); err != nil {
            println(err.(string))           // 将 interface{} 转型为具体类型。
        }
    }()

    panic("panic error!")
}
```

由于 panic、recover 参数类型为 interface{}，因此可抛出任何类型对象。

```
func panic(v interface{})
func recover() interface{}
```

延迟调用中引发的错误，可被后续延迟调用捕获，但仅最后一个错误可被捕获。

```
func test() {
    defer func() {
        fmt.Println(recover())
    }()

    defer func() {
        panic("defer panic")
    }()

    panic("test panic")
}

func main() {
    test()
}
```

输出：

```
defer panic
```

捕获函数 recover 只有在延迟调用内直接调用才会终止错误，否则总是返回 nil。任何未捕获的错误都会沿调用堆栈向外传递。

```
func test() {
    defer recover()                 // 无效!
    defer fmt.Println(recover())    // 无效!
    defer func() {
        func() {
            println("defer inner")
            recover()               // 无效!
        }()
```

```
    }()

    panic("test panic")
}

func main() {
    test()
}
```

输出：

```
defer inner
<nil>
panic: test panic
```

使用延迟匿名函数或下面这样都是有效的。

```
func except() {
    recover()
}

func test() {
    defer except()
    panic("test panic")
}
```

如果需要保护代码片段，可将代码块重构成匿名函数，如此可确保后续代码被执行。

```
func test(x, y int) {
    var z int

    func() {
        defer func() {
            if recover() != nil { z = 0 }
        }()

        z = x / y
        return
    }()

    println("x / y =", z)
}
```

除用 panic 引发中断性错误外，还可返回 error 类型错误对象来表示函数调用状态。

```
type error interface {
    Error() string
```

```
}
```

标准库 errors.New 和 fmt.Errorf 函数用于创建实现 error 接口的错误对象。通过判断错误对象实例来确定具体错误类型。

```
var ErrDivByZero = errors.New("division by zero")

func div(x, y int) (int, error) {
    if y == 0 { return 0, ErrDivByZero }
    return x / y, nil
}

func main() {
    switch z, err := div(10, 0); err {
    case nil:
        println(z)
    case ErrDivByZero:
        panic(err)
    }
}
```

如何区别使用 panic 和 error 两种方式？惯例是：导致关键流程出现不可修复性错误的使用 panic，其他使用 error。

第 4 章 数据

4.1 Array

和以往认知的数组有很大不同。

- 数组是值类型，赋值和传参会复制整个数组，而不是指针。
- 数组长度必须是常量，且是类型的组成部分。[2]int 和 [3]int 是不同类型。
- 支持 "=="、"!=" 操作符，因为内存总是被初始化过的。
- 指针数组 [n]*T，数组指针 *[n]T。

可用复合语句初始化。

```
a := [3]int{1, 2}               // 未初始化元素值为 0。
b := [...]int{1, 2, 3, 4}       // 通过初始化值确定数组长度。
c := [5]int{2: 100, 4:200}      // 使用索引号初始化元素。

d := [...]struct {
    name string
    age  uint8
}{
    {"user1", 10},              // 可省略元素类型。
    {"user2", 20},              // 别忘了最后一行的逗号。
}
```

支持多维数组。

```
a := [2][3]int{{1, 2, 3}, {4, 5, 6}}
b := [...][2]int{{1, 1}, {2, 2}, {3, 3}}    // 第 2 纬度不能用 "..."。
```

值拷贝行为会造成性能问题，通常会建议使用 slice，或数组指针。

```
func test(x [2]int) {
    fmt.Printf("x: %p\n", &x)
    x[1] = 1000
}

func main() {
    a := [2]int{}
    fmt.Printf("a: %p\n", &a)
```

```
    test(a)
    fmt.Println(a)
}
```

输出：

```
a: 0x2101f9150
x: 0x2101f9170
[0 0]
```

内置函数 len 和 cap 都返回数组长度 (元素数量)。

```
a := [2]int{}
println(len(a), cap(a))   // 2, 2
```

4.2 Slice

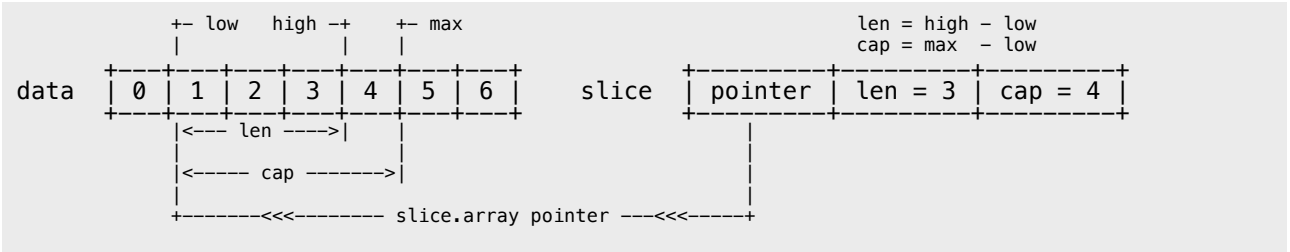
需要说明，slice 并不是数组或数组指针。它通过内部指针和相关属性引用数组片段，以实现变长方案。

runtime.h

```
struct Slice
{                          // must not move anything
    byte*    array;        // actual data
    uintgo   len;          // number of elements
    uintgo   cap;          // allocated number of elements
};
```

- 引用类型。但自身是结构体，值拷贝传递。
- 属性 len 表示可用元素数量，读写操作不能超过该限制。
- 属性 cap 表示最大扩张容量，不能超出数组限制。
- 如果 slice == nil，那么 len、cap 结果都等于 0。

```
data := [...]int{0, 1, 2, 3, 4, 5, 6}
slice := data[1:4:5]                      // [low : high : max]
```



创建表达式使用的是元素索引号，而非数量。

```
data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

expression   slice                  len    cap     comment
-----------+----------------------+------+-------+---------------------
data[:6:8]   [0 1 2 3 4 5]          6      8       省略 low.
data[5:]     [5 6 7 8 9]            5      5       省略 high、max。
data[:3]     [0 1 2]                3      10      省略 low、max。
data[:]      [0 1 2 3 4 5 6 7 8 9]  10     10      全部省略。
```

读写操作实际目标是底层数组，只需注意索引号的差别。

```
data := [...]int{0, 1, 2, 3, 4, 5}

s := data[2:4]
s[0] += 100
s[1] += 200

fmt.Println(s)
fmt.Println(data)
```

输出：

```
[102 203]
[0 1 102 203 4 5]
```

可直接创建 slice 对象，自动分配底层数组。

```
s1 := []int{0, 1, 2, 3, 8: 100}        // 通过初始化表达式构造，可使用索引号。
fmt.Println(s1, len(s1), cap(s1))

s2 := make([]int, 6, 8)                // 使用 make 创建，指定 len 和 cap 值。
fmt.Println(s2, len(s2), cap(s2))

s3 := make([]int, 6)                   // 省略 cap，相当于 cap = len。
fmt.Println(s3, len(s3), cap(s3))
```

输出：

```
[0 1 2 3 0 0 0 0 100]  9  9
[0 0 0 0 0 0]          6  8
[0 0 0 0 0 0]          6  6
```

使用 make 动态创建 slice，避免了数组必须用常量做长度的麻烦。还可用指针直接访问底层数组，退化成普通数组操作。

```
s := []int{0, 1, 2, 3}
```

```
p := &s[2]      // *int, 获取底层数组元素指针。
*p += 100

fmt.Println(s)
```

输出：

```
[0 1 102 3]
```

至于 []T，是指元素类型为 []T 。

```
data := [][]int{
    []int{1, 2, 3},
    []int{100, 200},
    []int{11, 22, 33, 44},
}
```

可直接修改 struct array/slice 成员。

```
d := [5]struct {
    x int
}{}

s := d[:]

d[1].x = 10
s[2].x = 20

fmt.Println(d)
fmt.Printf("%p, %p\n", &d, &d[0])
```

输出：

```
[{0} {10} {20} {0} {0}]
0x20819c180, 0x20819c180
```

4.2.1 reslice

所谓 reslice，是基于已有 slice 创建新 slice 对象，以便在 cap 允许范围内调整属性。

```
s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

s1 := s[2:5]        // [2 3 4]
s2 := s1[2:6:7]     // [4 5 6 7]
s3 := s2[3:6]       // Error
```

```
      +---+---+---+---+---+---+---+---+---+---+
data  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
      +---+---+---+---+---+---+---+---+---+---+
      0       2           5
              +---+---+---+---+---+---+---+---+
s1            | 2 | 3 | 4 |   |   |   |   |   |   len = 3, cap = 8
              +---+---+---+---+---+---+---+---+
              0       2               6   7
                      +---+---+---+---+---+
s2                    | 4 | 5 | 6 | 7 |   |       len = 4, cap = 5
                      +---+---+---+---+---+
                      0           3   4   5
                                  +---+---+---+
s3                                | 7 | 8 | X |   error: slice bounds out of range
                                  +---+---+---+
```

新对象依旧指向原底层数组。

```
s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

s1 := s[2:5]       // [2 3 4]
s1[2] = 100

s2 := s1[2:6]      // [100 5 6 7]
s2[3] = 200

fmt.Println(s)
```

输出：

```
[0 1 2 3 100 5 6 200 8 9]
```

4.2.2 append

向 slice 尾部添加数据，返回新的 slice 对象。

```
s := make([]int, 0, 5)
fmt.Printf("%p\n", &s)

s2 := append(s, 1)
fmt.Printf("%p\n", &s2)

fmt.Println(s, s2)
```

输出：

```
0x210230000
0x210230040
[] [1]
```

简单点说，就是在 array[slice.high] 写数据。

```
data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
s := data[:3]
s2 := append(s, 100, 200)    // 添加多个值。

fmt.Println(data)
fmt.Println(s)
fmt.Println(s2)
```

输出：

```
[0 1 2 100 200 5 6 7 8 9]
[0 1 2]
[0 1 2 100 200]
```

一旦超出原 slice.cap 限制，就会重新分配底层数组，即便原数组并未填满。

```
data := [...]int{0, 1, 2, 3, 4, 10: 0}
s := data[:2:3]

s = append(s, 100, 200)        // 一次 append 两个值，超出 s.cap 限制。

fmt.Println(s, data)           // 重新分配底层数组，与原数组无关。
fmt.Println(&s[0], &data[0])   // 比对底层数组起始指针。
```

输出：

```
[0 1 100 200] [0 1 2 3 4 0 0 0 0 0 0]
0x20819c180 0x20817c0c0
```

从输出结果可以看出，append 后的 s 重新分配了底层数组，并复制数据。如果只追加一个值，则不会超过 s.cap 限制，也就不会重新分配。

通常以 2 倍容量重新分配底层数组。在大批量添加数据时，建议一次性分配足够大的空间，以减少内存分配和数据复制开销。或初始化足够长的 len 属性，改用索引号进行操作。及时释放不再使用的 slice 对象，避免持有过期数组，造成 GC 无法回收。

```
s := make([]int, 0, 1)
c := cap(s)

for i := 0; i < 50; i++ {
    s = append(s, i)
    if n := cap(s); n > c {
        fmt.Printf("cap: %d -> %d\n", c, n)
        c = n
    }
}
```

输出：

```
cap: 1 -> 2
cap: 2 -> 4
```

```
cap: 4 -> 8
cap: 8 -> 16
cap: 16 -> 32
cap: 32 -> 64
```

4.2.3 copy

函数 copy 在两个 slice 间复制数据，复制长度以 len 小的为准。两个 slice 可指向同一底层数组，允许元素区间重叠。

```
data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

s := data[8:]
s2 := data[:5]

copy(s2, s)          // dst:s2, src:s

fmt.Println(s2)
fmt.Println(data)
```

输出：

```
[8 9 2 3 4]
[8 9 2 3 4 5 6 7 8 9]
```

应及时将所需数据 copy 到较小的 slice，以便释放超大号底层数组内存。

4.3 Map

引用类型，哈希表。键必须是支持相等运算符 (==、!=) 类型，比如 number、string、pointer、array、struct，以及对应的 interface。值可以是任意类型，没有限制。

```
m := map[int]struct {
    name string
    age  int
}{
    1:  {"user1", 10},        // 可省略元素类型。
    2:  {"user2", 20},
}

println(m[1].name)
```

预先给 make 函数一个合理元素数量参数，有助于提升性能。因为事先申请一大块内存，可避免后续操作时频繁扩张。

```
m := make(map[string]int, 1000)
```

常见操作：

```
m := map[string]int{
    "a": 1,
}

if v, ok := m["a"]; ok {     // 判断 key 是否存在。
    println(v)
}

println(m["c"])              // 对于不存在的 key，直接返回 \0，不会出错。

m["b"] = 2                   // 新增或修改。

delete(m, "c")               // 删除。如果 key 不存在，不会出错。

println(len(m))              // 获取键值对数量。cap 无效。

for k, v := range m {        // 迭代，可仅返回 key。随机顺序返回，每次都不相同。
    println(k, v)
}
```

不能保证迭代返回次序，通常是随机结果，具体和版本实现有关。

从 map 中取回的是一个 value 临时复制品，对其成员的修改是没有任何意义的。

```
type user struct{ name string }

m := map[int]user{        // 当 map 因扩张而重新哈希时，各键值项存储位置都会发生改变。 因此，map
    1: {"user1"},         // 被设计成 not addressable。 类似 m[1].name 这种期望透过原 value
}                         // 指针修改成员的行为自然会被禁止。

m[1].name = "Tom"         // Error: cannot assign to m[1].name
```

正确做法是完整替换 value 或使用指针。

```
u := m[1]
u.name = "Tom"
m[1] = u                  // 替换 value。
```

```
m2 := map[int]*user{
    1: &user{"user1"},
}

m2[1].name = "Jack"       // 返回的是指针复制品。透过指针修改原对象是允许的。
```

可以在迭代时安全删除键值。但如果期间有新增操作，那么就不知道会有什么意外了。

```
for i := 0; i < 5; i++ {
    m := map[int]string{
        0:  "a", 1:  "a", 2:  "a", 3:  "a", 4:  "a",
        5:  "a", 6:  "a", 7:  "a", 8:  "a", 9:  "a",
    }

    for k := range m {
        m[k+k] = "x"
        delete(m, k)
    }

    fmt.Println(m)
}
```

输出：

```
map[12:x 16:x 2:x 6:x 10:x 14:x 18:x]
map[12:x 16:x 20:x 28:x 36:x]
map[12:x 16:x 2:x 6:x 10:x 14:x 18:x]
map[12:x 16:x 2:x 6:x 10:x 14:x 18:x]
map[12:x 16:x 20:x 28:x 36:x]
```

4.4 Struct

值类型，赋值和传参会复制全部内容。可用 "_" 定义补位字段，支持指向自身类型的指针成员。

```
type Node struct {
    _    int
    id   int
    data *byte
    next *Node
}

func main() {
    n1 := Node{
        id:   1,
```

```
        data: nil,
    }

    n2 := Node{
        id:   2,
        data: nil,
        next: &n1,
    }
}
```

顺序初始化必须包含全部字段，否则会出错。

```
type User struct {
    name string
    age  int
}

u1 := User{"Tom", 20}
u2 := User{"Tom"}       // Error: too few values in struct initializer
```

支持匿名结构，可用作结构成员或定义变量。

```
type File struct {
    name string
    size int
    attr struct {
        perm  int
        owner int
    }
}

f := File{
    name: "test.txt",
    size: 1025,
    // attr: {0755, 1},   // Error: missing type in composite literal
}

f.attr.owner = 1
f.attr.perm = 0755

var attr = struct {
    perm  int
    owner int
}{2, 0755}

f.attr = attr
```

支持 "=="、"!=" 相等操作符，可用作 map 键类型。

```
type User struct {
    id   int
    name string
}

m := map[User]int{
    User{1, "Tom"}: 100,
}
```

可定义字段标签，用反射读取。标签是类型的组成部分。

```
var u1 struct { name string "username" }
var u2 struct { name string }

u2 = u1   // Error: cannot use u1 (type struct { name string "username" }) as
          //        type struct { name string } in assignment
```

空结构 "节省" 内存，比如用来实现 set 数据结构，或者实现没有 "状态" 只有方法的 "静态类"。

```
var null struct{}

set := make(map[string]struct{})
set["a"] = null
```

4.4.1 匿名字段

匿名字段不过是一种语法糖，从根本上说，就是一个与成员类型同名 (不含包名) 的字段。被匿名嵌入的可以是任何类型，当然也包括指针。

```
type User struct {
    name string
}

type Manager struct {
    User
    title string
}
```

```
m := Manager{
    User:  User{"Tom"},          // 匿名字段的显式字段名，和类型名相同。
    title: "Administrator",
}
```

可以像普通字段那样访问匿名字段成员，编译器从外向内逐级查找所有层次的匿名字段，直到发现目标或出错。

```
type Resource struct {
    id int
}

type User struct {
    Resource
    name string
}

type Manager struct {
    User
    title string
}

var m Manager
m.id = 1
m.name = "Jack"
m.title = "Administrator"
```

外层同名字段会遮蔽嵌入字段成员，相同层次的同名字段也会让编译器无所适从。解决方法是使用显式字段名。

```
type Resource struct {
    id   int
    name string
}

type Classify struct {
    id int
}

type User struct {
    Resource               // Resource.id 与 Classify.id 处于同一层次。
    Classify
    name string            // 遮蔽 Resource.name。
}

u := User{
```

```
    Resource{1, "people"},
    Classify{100},
    "Jack",
}

println(u.name)            // User.name: Jack
println(u.Resource.name)   // people

// println(u.id)           // Error: ambiguous selector u.id
println(u.Classify.id)     // 100
```

不能同时嵌入某一类型和其指针类型，因为它们名字相同。

```
type Resource struct {
    id int
}

type User struct {
    *Resource
    // Resource            // Error: duplicate field Resource
    name string
}

u := User{
    &Resource{1},
    "Administrator",
}

println(u.id)
println(u.Resource.id)
```

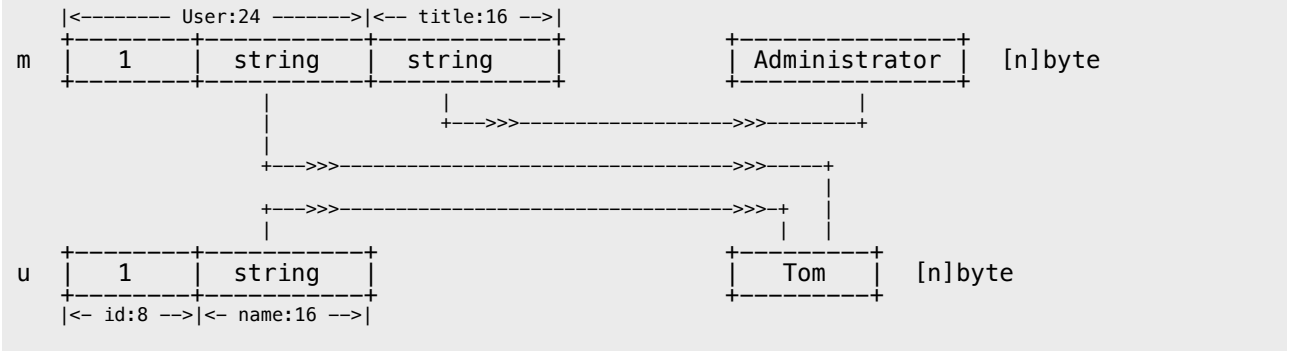
4.4.2 面向对象

面向对象三大特征里，Go 仅支持封装，尽管匿名字段的内存布局和行为类似继承。没有 class 关键字，没有继承、多态等等。

```
type User struct {
    id   int
    name string
}

type Manager struct {
    User
    title string
```

```
}

m := Manager{User{1, "Tom"}, "Administrator"}

// var u User = m     // Error: cannot use m (type Manager) as type User in assignment
                      // 没有继承，自然也不会有多态。
var u User = m.User   // 同类型拷贝。
```

内存布局和 C struct 相同，没有任何附加的 object 信息。



可用 unsafe 包相关函数输出内存地址信息。

```
m      : 0x2102271b0, size: 40, align: 8
m.id   : 0x2102271b0, offset: 0
m.name : 0x2102271b8, offset: 8
m.title: 0x2102271c8, offset: 24
```

第 5 章 方法

5.1 方法定义

方法总是绑定对象实例，并隐式将实例作为第一实参 (receiver)。

- 只能为当前包内命名类型定义方法。
- 参数 receiver 可任意命名。如方法中未曾使用，可省略参数名。
- 参数 receiver 类型可以是 T 或 *T。基类型 T 不能是接口或指针。
- 不支持方法重载，receiver 只是参数签名的组成部分。
- 可用实例 value 或 pointer 调用全部方法，编译器自动转换。

没有构造和析构方法，通常用简单工厂模式返回对象实例。

```
type Queue struct {
    elements []interface{}
}

func NewQueue() *Queue {                        // 创建对象实例。
    return &Queue{make([]interface{}, 10)}
}

func (*Queue) Push(e interface{}) error {       // 省略 receiver 参数名。
    panic("not implemented")
}

// func (Queue) Push(e int) error {             // Error: method redeclared: Queue.Push
//     panic("not implemented")
// }

func (self *Queue) length() int {               // receiver 参数名可以是 self、this 或其他。
    return len(self.elements)
}
```

方法不过是一种特殊的函数，只需将其还原，就知道 receiver T 和 *T 的差别。

```
type Data struct{
    x int
}

func (self Data) ValueTest() {                  // func ValueTest(self Data);
    fmt.Printf("Value: %p\n", &self)
```

```
}

func (self *Data) PointerTest() {             // func PointerTest(self *Data);
    fmt.Printf("Pointer: %p\n", self)
}

func main() {
    d := Data{}
    p := &d
    fmt.Printf("Data: %p\n", p)

    d.ValueTest()       // ValueTest(d)
    d.PointerTest()     // PointerTest(&d)

    p.ValueTest()       // ValueTest(*p)
    p.PointerTest()     // PointerTest(p)
}
```

输出：

```
Data   : 0x2101ef018
Value  : 0x2101ef028
Pointer: 0x2101ef018
Value  : 0x2101ef030
Pointer: 0x2101ef018
```

从 1.4 开始，不再支持多级指针查找方法成员。

```
type X struct{}

func (*X) test() {
    println("X.test")
}

func main() {
    p := &X{}
    p.test()

    // Error: calling method with receiver &p (type **X) requires explicit dereference
    // (&p).test()
}
```

5.2 匿名字段

可以像字段成员那样访问匿名字段方法，编译器负责查找。

```
type User struct {
```

```
    id   int
    name string
}

type Manager struct {
    User
}

func (self *User) ToString() string {                // receiver = &(Manager.User)
    return fmt.Sprintf("User: %p, %v", self, self)
}

func main() {
    m := Manager{User{1, "Tom"}}

    fmt.Printf("Manager: %p\n", &m)
    fmt.Println(m.ToString())
}
```

输出：

```
Manager: 0x2102281b0
User   : 0x2102281b0, &{1 Tom}
```

通过匿名字段，可获得和继承类似的复用能力。依据编译器查找次序，只需在外层定义同名方法，就可以实现 "override"。

```
type User struct {
    id   int
    name string
}

type Manager struct {
    User
    title string
}

func (self *User) ToString() string {
    return fmt.Sprintf("User: %p, %v", self, self)
}

func (self *Manager) ToString() string {
    return fmt.Sprintf("Manager: %p, %v", self, self)
}

func main() {
    m := Manager{User{1, "Tom"}, "Administrator"}

    fmt.Println(m.ToString())
```

```
    fmt.Println(m.User.ToString())
}
```

输出：

```
Manager: 0x2102271b0, &{{1 Tom} Administrator}
User   : 0x2102271b0, &{1 Tom}
```

5.3 方法集

每个类型都有与之关联的方法集，这会影响到接口实现规则。

- 类型 T 方法集包含全部 receiver T 方法。
- 类型 *T 方法集包含全部 receiver T + *T 方法。
- 如类型 S 包含匿名字段 T，则 S 方法集包含 T 方法。
- 如类型 S 包含匿名字段 *T，则 S 方法集包含 T + *T 方法。
- 不管嵌入 T 或 *T，*S 方法集总是包含 T + *T 方法。

用实例 value 和 pointer 调用方法 (含匿名字段) 不受方法集约束，编译器总是查找全部方法，并自动转换 receiver 实参。

5.4 表达式

根据调用者不同，方法分为两种表现形式：

```
instance.method(args...)  --->  <type>.func(instance, args...)
```

前者称为 method value，后者 method expression。

两者都可像普通函数那样赋值和传参，区别在于 method value 绑定实例，而 method expression 则须显式传参。

```
type User struct {
    id   int
    name string
}

func (self *User) Test() {
    fmt.Printf("%p, %v\n", self, self)
}
```

```
func main() {
    u := User{1, "Tom"}
    u.Test()

    mValue := u.Test
    mValue()                            // 隐式传递 receiver

    mExpression := (*User).Test
    mExpression(&u)                     // 显式传递 receiver
}
```

输出：

```
0x210230000, &{1 Tom}
0x210230000, &{1 Tom}
0x210230000, &{1 Tom}
```

需要注意，method value 会复制 receiver。

```
type User struct {
    id   int
    name string
}

func (self User) Test() {
    fmt.Println(self)
}

func main() {
    u := User{1, "Tom"}
    mValue := u.Test           // 立即复制 receiver，因为不是指针类型，不受后续修改影响。

    u.id, u.name = 2, "Jack"
    u.Test()

    mValue()
}
```

输出：

```
{2 Jack}
{1 Tom}
```

在汇编层面，method value 和闭包的实现方式相同，实际返回 FuncVal 类型对象。

```
FuncVal { method_address, receiver_copy }
```

可依据方法集转换 method expression，注意 receiver 类型的差异。

```
type User struct {
    id   int
    name string
}

func (self *User) TestPointer() {
    fmt.Printf("TestPointer: %p, %v\n", self, self)
}

func (self User) TestValue() {
    fmt.Printf("TestValue: %p, %v\n", &self, self)
}

func main() {
    u := User{1, "Tom"}
    fmt.Printf("User: %p, %v\n", &u, u)

    mv := User.TestValue
    mv(u)

    mp := (*User).TestPointer
    mp(&u)

    mp2 := (*User).TestValue    // *User 方法集包含 TestValue。
    mp2(&u)                     // 签名变为 func TestValue(self *User)。
}                               // 实际依然是 receiver value copy。
```

输出：

```
User       : 0x210231000, {1 Tom}
TestValue  : 0x210231060, {1 Tom}
TestPointer: 0x210231000, &{1 Tom}
TestValue  : 0x2102310c0, {1 Tom}
```

将方法 "还原" 成函数，就容易理解下面的代码了。

```
type Data struct{}

func (Data) TestValue()    {}
func (*Data) TestPointer() {}

func main() {
    var p *Data = nil
    p.TestPointer()

    (*Data)(nil).TestPointer()  // method value
    (*Data).TestPointer(nil)    // method expression

    // p.TestValue()            // invalid memory address or nil pointer dereference
```

```
    // (Data)(nil).TestValue()  // cannot convert nil to type Data
    // Data.TestValue(nil)      // cannot use nil as type Data in function argument
}
```

第 6 章 接口

6.1 接口定义

接口是一个或多个方法签名的集合，任何类型的方法集中只要拥有与之对应的全部方法，就表示它 "实现" 了该接口，无须在该类型上显式添加接口声明。

所谓对应方法，是指有相同名称、参数列表 (不包括参数名) 以及返回值。当然，该类型还可以有其他方法。

- 接口命名习惯以 er 结尾，结构体。
- 接口只有方法签名，没有实现。
- 接口没有数据字段。
- 可在接口中嵌入其他接口。
- 类型可实现多个接口。

```
type Stringer interface {
    String() string
}

type Printer interface {
    Stringer                            // 接口嵌入。
    Print()
}

type User struct {
    id   int
    name string
}

func (self *User) String() string {
    return fmt.Sprintf("user %d, %s", self.id, self.name)
}

func (self *User) Print() {
    fmt.Println(self.String())
}

func main() {
    var t Printer = &User{1, "Tom"}   // *User 方法集包含 String、Print。
    t.Print()
}
```

输出：

```
user 1, Tom
```

空接口 interface{} 没有任何方法签名，也就意味着任何类型都实现了空接口。其作用类似面向对象语言中的根对象 object。

```
func Print(v interface{}) {
    fmt.Printf("%T: %v\n", v, v)
}

func main() {
    Print(1)
    Print("Hello, World!")
}
```

输出：

```
int: 1
string: Hello, World!
```

匿名接口可用作变量类型，或结构成员。

```
type Tester struct {
    s interface {
        String() string
    }
}

type User struct {
    id   int
    name string
}

func (self *User) String() string {
    return fmt.Sprintf("user %d, %s", self.id, self.name)
}

func main() {
    t := Tester{&User{1, "Tom"}}
    fmt.Println(t.s.String())
}
```

输出：

```
user 1, Tom
```

6.2 执行机制

接口对象由接口表 (interface table) 指针和数据指针组成。

runtime.h

```
struct Iface
{
    Itab*    tab;
    void*    data;
};

struct Itab
{
    InterfaceType*    inter;
    Type*             type;
    void (*fun[])(void);
};
```

接口表存储元数据信息，包括接口类型、动态类型，以及实现接口的方法指针。无论是反射还是通过接口调用方法，都会用到这些信息。

数据指针持有的是目标对象的只读复制品，复制完整对象或指针。

```
type User struct {
    id   int
    name string
}

func main() {
    u := User{1, "Tom"}
    var i interface{} = u

    u.id = 2
    u.name = "Jack"

    fmt.Printf("%v\n", u)
    fmt.Printf("%v\n", i.(User))
}
```

输出：

```
{2 Jack}
{1 Tom}
```

接口转型返回临时对象，只有使用指针才能修改其状态。

```
type User struct {
    id   int
    name string
}

func main() {
    u := User{1, "Tom"}
    var vi, pi interface{} = u, &u

    // vi.(User).name = "Jack"         // Error: cannot assign to vi.(User).name
    pi.(*User).name = "Jack"

    fmt.Printf("%v\n", vi.(User))
    fmt.Printf("%v\n", pi.(*User))
}
```

输出：

```
{1 Tom}
&{1 Jack}
```

只有 tab 和 data 都为 nil 时，接口才等于 nil。

```
var a interface{} = nil             // tab = nil, data = nil
var b interface{} = (*int)(nil)     // tab 包含 *int 类型信息, data = nil

type iface struct {
    itab, data uintptr
}

ia := *(*iface)(unsafe.Pointer(&a))
ib := *(*iface)(unsafe.Pointer(&b))

fmt.Println(a == nil, ia)
fmt.Println(b == nil, ib, reflect.ValueOf(b).IsNil())
```

输出：

```
true  {0 0}
false {505728 0} true
```

6.3 接口转换

利用类型推断，可判断接口对象是否某个具体的接口或类型。

```
type User struct {
    id   int
    name string
```

```
}

func (self *User) String() string {
    return fmt.Sprintf("%d, %s", self.id, self.name)
}

func main() {
    var o interface{} = &User{1, "Tom"}

    if i, ok := o.(fmt.Stringer); ok {     // ok-idiom
        fmt.Println(i)
    }

    u := o.(*User)
    // u := o.(User)       // panic: interface is *main.User, not main.User
    fmt.Println(u)
}
```

还可用 switch 做批量类型判断，不支持 fallthrough。

```
func main() {
    var o interface{} = &User{1, "Tom"}

    switch v := o.(type) {
    case nil:                                   // o == nil
        fmt.Println("nil")
    case fmt.Stringer:                          // interface
        fmt.Println(v)
    case func() string:                         // func
        fmt.Println(v())
    case *User:                                 // *struct
        fmt.Printf("%d, %s\n", v.id, v.name)
    default:
        fmt.Println("unknown")
    }
}
```

超集接口对象可转换为子集接口，反之出错。

```
type Stringer interface {
    String() string
}

type Printer interface {
    String() string
    Print()
}
```

```
type User struct {
    id   int
    name string
}

func (self *User) String() string {
    return fmt.Sprintf("%d, %v", self.id, self.name)
}

func (self *User) Print() {
    fmt.Println(self.String())
}

func main() {
    var o Printer = &User{1, "Tom"}
    var s Stringer = o
    fmt.Println(s.String())
}
```

6.4 接口技巧

让编译器检查，以确保某个类型实现接口。

```
var _ fmt.Stringer = (*Data)(nil)
```

某些时候，让函数直接 "实现" 接口能省不少事。

```
type Tester interface {
    Do()
}

type FuncDo func()
func (self FuncDo) Do() { self() }

func main() {
    var t Tester = FuncDo(func() { println("Hello, World!") })
    t.Do()
}
```

第 7 章 并发

7.1 Goroutine

Go 在语言层面对并发编程提供支持，一种类似协程，称作 goroutine 的机制。

只需在函数调用语句前添加 go 关键字，就可创建并发执行单元。开发人员无需了解任何执行细节，调度器会自动将其安排到合适的系统线程上执行。goroutine 是一种非常轻量级的实现，可在单个进程里执行成千上万的并发任务。

事实上，入口函数 main 就以 goroutine 运行。另有与之配套的 channel 类型，用以实现 "以通讯来共享内存" 的 CSP 模式。相关实现细节可参考本书第二部分的源码剖析。

```
go func() {
    println("Hello, World!")
}()
```

调度器不能保证多个 goroutine 执行次序，且进程退出时不会等待它们结束。

默认情况下，进程启动后仅允许一个系统线程服务于 goroutine。可使用环境变量或标准库函数 runtime.GOMAXPROCS 修改，让调度器用多个线程实现多核并行，而不仅仅是并发。

```
func sum(id int) {
    var x int64
    for i := 0; i < math.MaxUint32; i++ {
        x += int64(i)
    }

    println(id, x)
}

func main() {
    wg := new(sync.WaitGroup)
    wg.Add(2)

    for i := 0; i < 2; i++ {
        go func(id int) {
            defer wg.Done()
            sum(id)
        }(i)
```

```
    }

    wg.Wait()
}
```

输出：

```
$ go build -o test

$ time -p ./test

0 9223372030412324865
1 9223372030412324865

real   7.70              // 程序开始到结束时间差（非 CPU 时间）
user   7.66              // 用户态所使用 CPU 时间片（多核累加）
sys    0.01              // 内核态所使用 CPU 时间片

$ GOMAXPROCS=2 time -p ./test

0 9223372030412324865
1 9223372030412324865

real   4.18
user   7.61              // 虽然总时间差不多，但由 2 个核并行，real 时间自然少了许多。
sys    0.02
```

调用 runtime.Goexit 将立即终止当前 goroutine 执行，调度器确保所有已注册 defer 延迟调用被执行。

```
func main() {
    wg := new(sync.WaitGroup)
    wg.Add(1)

    go func() {
        defer wg.Done()
        defer println("A.defer")

        func() {
            defer println("B.defer")
            runtime.Goexit()              // 终止当前 goroutine
            println("B")                  // 不会执行
        }()

        println("A")                      // 不会执行
    }()

    wg.Wait()
}
```

输出：

```
B.defer
A.defer
```

和协程 yield 作用类似，Gosched 让出底层线程，将当前 goroutine 暂停，放回队列等待下次被调度执行。

```
func main() {
    wg := new(sync.WaitGroup)
    wg.Add(2)

    go func() {
        defer wg.Done()

        for i := 0; i < 6; i++ {
            println(i)
            if i == 3 { runtime.Gosched() }
        }
    }()

    go func() {
        defer wg.Done()
        println("Hello, World!")
    }()

    wg.Wait()
}
```

输出：

```
$ go run main.go
0
1
2
3
Hello, World!
4
5
```

7.2 Channel

引用类型 channel 是 CSP 模式的具体实现，用于多个 goroutine 通讯。其内部实现了同步，确保并发安全。

默认为同步模式，需要发送和接收配对。否则会被阻塞，直到另一方准备好后被唤醒。

```
func main() {
    data := make(chan int)           // 数据交换队列
    exit := make(chan bool)          // 退出通知

    go func() {
        for d := range data {        // 从队列迭代接收数据，直到 close 。
            fmt.Println(d)
        }

        fmt.Println("recv over.")
        exit <- true                 // 发出退出通知。
    }()

    data <- 1                        // 发送数据。
    data <- 2
    data <- 3
    close(data)                      // 关闭队列。

    fmt.Println("send over.")
    <-exit                           // 等待退出通知。
}
```

输出：

```
1
2
3
send over.
recv over.
```

异步方式通过判断缓冲区来决定是否阻塞。如果缓冲区已满，发送被阻塞；缓冲区为空，接收被阻塞。

通常情况下，异步 channel 可减少排队阻塞，具备更高的效率。但应该考虑使用指针规避大对象拷贝，将多个元素打包，减小缓冲区大小等。

```
func main() {
    data := make(chan int, 3)       // 缓冲区可以存储 3 个元素
    exit := make(chan bool)

    data <- 1                       // 在缓冲区未满前，不会阻塞。
    data <- 2
    data <- 3

    go func() {
        for d := range data {       // 在缓冲区未空前，不会阻塞。
            fmt.Println(d)
```

```
        }

        exit <- true
    }()

    data <- 4                       // 如果缓冲区已满，阻塞。
    data <- 5
    close(data)

    <-exit
}
```

缓冲区是内部属性，并非类型构成要素。

```
var a, b chan int = make(chan int), make(chan int, 3)
```

除用 range 外，还可用 ok-idiom 模式判断 channel 是否关闭。

```
for {
    if d, ok := <-data; ok {
        fmt.Println(d)
    } else {
        break
    }
}
```

向 closed channel 发送数据引发 panic 错误，接收立即返回零值。而 nil channel，无论收发都会被阻塞。

内置函数 len 返回未被读取的缓冲元素数量，cap 返回缓冲区大小。

```
d1 := make(chan int)
d2 := make(chan int, 3)

d2 <- 1

fmt.Println(len(d1), cap(d1))    // 0  0
fmt.Println(len(d2), cap(d2))    // 1  3
```

7.2.1 单向

可以将 channel 隐式转换为单向队列，只收或只发。

```
c := make(chan int, 3)

var send chan<- int = c   // send-only
var recv <-chan int = c   // receive-only

send <- 1
// <-send                 // Error: receive from send-only type chan<- int

<-recv
// recv <- 2              // Error: send to receive-only type <-chan int
```

不能将单向 channel 转换为普通 channel。

```
d := (chan int)(send)     // Error: cannot convert type chan<- int to type chan int
d := (chan int)(recv)     // Error: cannot convert type <-chan int to type chan int
```

7.2.2 选择

如果需要同时处理多个 channel，可使用 select 语句。它随机选择一个可用 channel 做收发操作，或执行 default case。

```
func main() {
    a, b := make(chan int, 3), make(chan int)

    go func() {
        v, ok, s := 0, false, ""

        for {
            select {                          // 随机选择可用 channel，接收数据。
            case v, ok = <-a: s = "a"
            case v, ok = <-b: s = "b"
            }

            if ok {
                fmt.Println(s, v)
            } else {
                os.Exit(0)
            }
        }
    }()

    for i := 0; i < 5; i++ {
        select {                              // 随机选择可用 channel，发送数据。
```

```
        case a <- i:
        case b <- i:
        }
    }

    close(a)
    select {}                                   // 没有可用 channel，阻塞 main goroutine。
}
```

输出：

```
b 3
a 0
a 1
a 2
b 4
```

在循环中使用 select default case 需要小心，避免形成洪水。

7.2.3 模式

用简单工厂模式打包并发任务和 channel。

```
func NewTest() chan int {
    c := make(chan int)
    rand.Seed(time.Now().UnixNano())

    go func() {
        time.Sleep(time.Second)
        c <- rand.Int()
    }()

    return c
}

func main() {
    t := NewTest()
    println(<-t)    // 等待 goroutine 结束返回。
}
```

用 channel 实现信号量 (semaphore)。

```
func main() {
    wg := sync.WaitGroup{}
    wg.Add(3)
```

```
    sem := make(chan int, 1)

    for i := 0; i < 3; i++ {
        go func(id int) {
            defer wg.Done()

            sem <- 1                    // 向 sem 发送数据，阻塞或者成功。

            for x := 0; x < 3; x++ {
                fmt.Println(id, x)
            }

            <-sem                       // 接收数据，使得其他阻塞 goroutine 可以发送数据。
        }(i)
    }

    wg.Wait()
}
```

输出：

```
$ GOMAXPROCS=2 go run main.go
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

用 closed channel 发出退出通知。

```
func main() {
    var wg sync.WaitGroup
    quit := make(chan bool)

    for i := 0; i < 2; i++ {
        wg.Add(1)

        go func(id int) {
            defer wg.Done()

            task := func() {
                println(id, time.Now().Nanosecond())
                time.Sleep(time.Second)
            }
```

```
            for {
                select {
                case <-quit:      // closed channel 不会阻塞，因此可用作退出通知。
                    return
                default:          // 执行正常任务。
                    task()
                }
            }
        }(i)
    }

    time.Sleep(time.Second * 5)   // 让测试 goroutine 运行一会。

    close(quit)                   // 发出退出通知。
    wg.Wait()
}
```

用 select 实现超时 (timeout)。

```
func main() {
    w := make(chan bool)
    c := make(chan int, 2)

    go func() {
        select {
        case v := <-c: fmt.Println(v)
        case <-time.After(time.Second * 3): fmt.Println("timeout.")
        }

        w <- true
    }()

    // c <- 1                     // 注释掉，引发 timeout。
    <-w
}
```

channel 是第一类对象，可传参 (内部实现为指针) 或者作为结构成员。

```
type Request struct {
    data []int
    ret  chan int
}

func NewRequest(data ...int) *Request {
    return &Request{ data, make(chan int, 1) }
}
```

```
func Process(req *Request) {
    x := 0
    for _, i := range req.data {
        x += i
    }

    req.ret <- x
}

func main() {
    req := NewRequest(10, 20, 30)
    Process(req)
    fmt.Println(<-req.ret)
}
```

第 8 章 包

8.1 工作空间

编译工具对源码目录有严格要求，每个工作空间 (workspace) 必须由 bin、pkg、src 三个目录组成。

```
workspace
    |
    +--- bin                                    // go install 安装目录。
    |     |
    |     +--- learn
    |
    +--- pkg                                    // go build 生成静态库 (.a) 存放目录。
    |     |
    |     +--- darwin_amd64
    |               |
    |               +--- mylib.a
    |               |
    |               +--- mylib
    |                       |
    |                       +--- sublib.a
    |
    +--- src                                    // 项目源码目录。
          |
          +--- learn
          |       |
          |       +--- main.go
          |
          +--- mylib
                  |
                  +--- mylib.go
                  |
                  +--- sublib
                          |
                          +--- sublib.go
```

可在 GOPATH 环境变量列表中添加多个工作空间，但不能和 GOROOT 相同。

```
export GOPATH=$HOME/projects/golib:$HOME/projects/go
```

通常 go get 使用第一个工作空间保存下载的第三方库。

8.2 源文件

编码：源码文件必须是 UTF-8 格式，否则会导致编译器出错。
结束：语句以 ";" 结束，多数时候可以省略。
注释：支持 "//"、"/**/" 两种注释方式，不能嵌套。
命名：采用 camelCasing 风格，不建议使用下划线。

8.3 包结构

所有代码都必须组织在 package 中。

- 源文件头部以 "package <name>" 声明包名称。
- 包由同一目录下的多个源码文件组成。
- 包名类似 namespace，与包所在目录名、编译文件名无关。
- 目录名最好不用 main、all、std 这三个保留名称。
- 可执行文件必须包含 package main，入口函数 main。

说明：os.Args 返回命令行参数，os.Exit 终止进程。

要获取正确的可执行文件路径，可用 filepath.Abs(exec.LookPath(os.Args[0]))。

包中成员以名称首字母大小写决定访问权限。

- public: 首字母大写，可被包外访问。
- internal: 首字母小写，仅包内成员可以访问。

该规则适用于全局变量、全局常量、类型、结构字段、函数、方法等。

8.3.1 导入包

使用包成员前，必须先用 import 关键字导入，但不能形成导入循环。

```
import "相对目录/包主文件名"
```

相对目录是指从 <workspace>/pkg/<os_arch> 开始的子目录，以标准库为例：

```
import "fmt"      ->  /usr/local/go/pkg/darwin_amd64/fmt.a
import "os/exec"  ->  /usr/local/go/pkg/darwin_amd64/os/exec.a
```

在导入时，可指定包成员访问方式。比如对包重命名，以避免同名冲突。

```
import     "yuhen/test"      // 默认模式: test.A
import  M  "yuhen/test"      // 包重命名: M.A
import  .  "yuhen/test"      // 简便模式: A
import  _  "yuhen/test"      // 非导入模式: 仅让该包执行初始化函数。
```

未使用的导入包，会被编译器视为错误 (不包括 "import _")。

```
./main.go:4: imported and not used: "fmt"
```

对于当前目录下的子包，除使用默认完整导入路径外，还可使用 local 方式。

```
workspace
    |
    +--- src
          |
          +--- learn
                 |
                 +--- main.go
                 |
                 +--- test
                        |
                        +--- test.go
```

main.go

```
import "learn/test"     // 正常模式
import "./test"         // 本地模式，仅对 go run main.go 有效。
```

8.3.2 自定义路径

可通过 meta 设置为代码库设置自定义路径。

server.go

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, `<meta name="go-import"
                  content="test.com/qyuhen/test git https://github.com/qyuhen/test">`)
}

func main() {
    http.HandleFunc("/qyuhen/test", handler)
    http.ListenAndServe(":80", nil)
}
```

该示例使用自定义域名 test.com 重定向到 github。

```
$ go get -v test.com/qyuhen/test

Fetching https://test.com/qyuhen/test?go-get=1
https fetch failed.
Fetching http://test.com/qyuhen/test?go-get=1
Parsing meta tags from http://test.com/qyuhen/test?go-get=1 (status code 200)
get "test.com/qyuhen/test": found meta tag http://test.com/qyuhen/test?go-get=1
test.com/qyuhen/test (download)
test.com/qyuhen/test
```

如此，该库就有两个有效导入路径，可能会导致存储两个本地副本。为此，可以给库添加专门的 "import comment"。当 go get 下载完成后，会检查本地存储路径和该注释是否一致。

github.com/qyuhen/test/abc.go

```
package test // import "test.com/qyuhen/test"

func Hello() {
    println("Hello, Custom import path!")
}
```

如继续用 github 路径，会导致 go build 失败。

```
$ go get -v github.com/qyuhen/test

github.com/qyuhen/test (download)
package github.com/qyuhen/test
       imports github.com/qyuhen/test
       imports github.com/qyuhen/test: expects import "test.com/qyuhen/test"
```

这就强制包用户使用唯一路径，也便于日后将包迁移到其他位置。

资源：Go 1.4 Custom Import Path Checking

8.3.3 初始化

初始化函数：

- 每个源文件都可以定义一个或多个初始化函数。
- 编译器不保证多个初始化函数执行次序。

- 初始化函数在单一线程被调用，仅执行一次。
- 初始化函数在包所有全局变量初始化后执行。
- 在所有初始化函数结束后才执行 main.main。
- 无法调用初始化函数。

因为无法保证初始化函数执行顺序，因此全局变量应该直接用 var 初始化。

```
var now = time.Now()

func init() {
    fmt.Printf("now: %v\n", now)
}

func init() {
    fmt.Printf("since: %v\n", time.Now().Sub(now))
}
```

可在初始化函数中使用 goroutine，可等待其结束。

```
var now = time.Now()

func main() {
    fmt.Println("main:", int(time.Now().Sub(now).Seconds()))
}

func init() {
    fmt.Println("init:", int(time.Now().Sub(now).Seconds()))
    w := make(chan bool)

    go func() {
        time.Sleep(time.Second * 3)
        w <- true
    }()

    <-w
}
```

输出：

```
init: 0
main: 3
```

不应该滥用初始化函数，仅适合完成当前文件中的相关环境设置。

8.4 文档

扩展工具 godoc 能自动提取注释生成帮助文档。

- 仅和成员相邻 (中间没有空行) 的注释被当做帮助信息。
- 相邻行会合并成同一段落，用空行分隔段落。
- 缩进表示格式化文本，比如示例代码。
- 自动转换 URL 为链接。
- 自动合并多个源码文件中的 package 文档。
- 无法显式 package main 中的成员文档。

8.4.1 Package

- 建议用专门的 doc.go 保存 package 帮助信息。
- 包文档第一整句 (中英文句号结束) 被当做 packages 列表说明。

8.4.2 Example

只要 Example 测试函数名称符合以下规范即可。

```
            格式                                  示例
-----------+-----------------------------------+-----------------------------------------
package     Example,    Example_suffix          Example_test
func        ExampleF,   ExampleF_suffix         ExampleHello
type        ExampleT,   ExampleT_suffix         ExampleUser, ExampleUser_copy
method      ExampleT_M, ExampleT_M_suffix       ExampleUser_ToString
```

说明：使用 suffix 作为示例名称，其首字母必须小写。如果文件中仅有一个 Example 函数，且调用了该文件中的其他成员，那么示例会显示整个文件内容，而不仅仅是测试函数自己。

8.4.3 Bug

非测试源码文件中以 BUG(author) 开始的注释，会在帮助文档 Bugs 节点中显示。

```
// BUG(yuhen): memory leak.
```

第 9 章 进阶

9.1 内存布局

了解对象内存布局，有助于理解值传递、引用传递等概念。

```
+--------+
|   1    |  int
+--------+

+--------+
|  3.14  |  float32
+--------+

+---+---+---+---+
| 1 | 2 | 3 | 4 |   [4]int
+---+---+---+---+
```

string

```
+---------+---------+
| pointer | len = 5 |   s = "hello"
+---------+---------+
    |
    |
    +---+---+---+---+---+
    | h | e | l | l | o |   [5]byte
    +---+---+---+---+---+
        |
    +---------+---------+
    | pointer | len = 2 |   sub = s[1:3]
    +---------+---------+
```

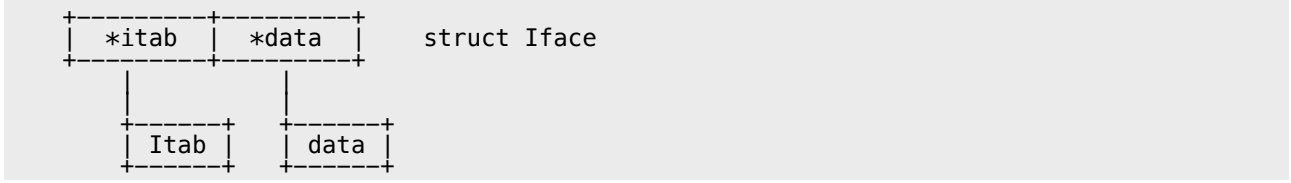
struct

```
+---+---+---+---+---+---+---+---+
| 1 | 2 | 0 | 0 | 3 | 0 | 0 | 0 |  struct { a byte; b byte; c int32 } = { 1, 2, 3 }
+---+---+---+---+---+---+---+---+
  a   b           c


+-----------+-----+
| pointer a |  b  |   struct { a *int; b int }
+-----------+-----+
    |
    |
    +-----+
    | int |
    +-----+
```

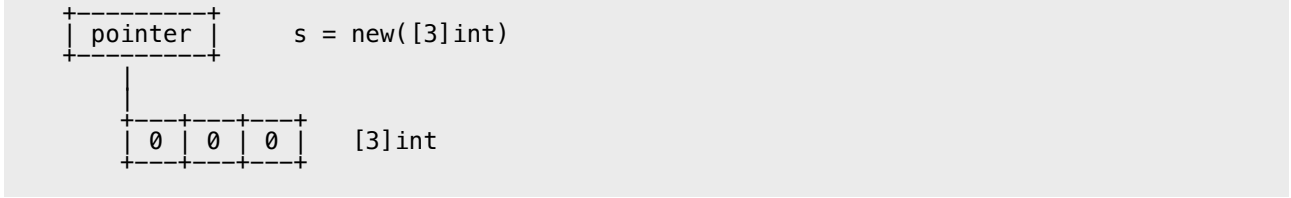
slice

```
+---------+---------+---------+
| pointer | len = 8 | cap = 8 |   x = []int{0, 1, 2, 3, 4, 5, 6, 7 }
+---------+---------+---------+
    |
    |
    +---+---+---+---+---+---+---+---+
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   [8]int
    +---+---+---+---+---+---+---+---+
        |
    +---------+---------+---------+
    | pointer | len = 2 | cap = 5 |  y = x[1:3:6]
    +---------+---------+---------+
```

interface



new



make

9.2 指针陷阱

对象内存分配会受编译参数影响。举个例子，当函数返回对象指针时，必然在堆上分配。可如果该函数被内联，那么这个指针就不会跨栈帧使用，就有可能直接在栈上分配，以实现代码优化目的。因此，是否阻止内联对指针输出结果有很大影响。

允许指针指向对象成员，并确保该对象是可达状态。

除正常指针外，指针还有 unsafe.Pointer 和 uintptr 两种形态。其中 uintptr 被 GC 当做普通整数对象，它不能阻止所 "引用" 对象被回收。

```
type data struct {
    x [1024 * 100]byte
}

func test() uintptr {
    p := &data{}
    return uintptr(unsafe.Pointer(p))
}

func main() {
    const N = 10000
    cache := new([N]uintptr)

    for i := 0; i < N; i++ {
        cache[i] = test()
        time.Sleep(time.Millisecond)
    }
}
```

输出：

```
$ go build -o test && GODEBUG="gctrace=1" ./test

gc607(1): 0+0+0 ms, 0 -> 0 MB 50 -> 45 (3070-3025) objects
gc611(1): 0+0+0 ms, 0 -> 0 MB 50 -> 45 (3090-3045) objects
gc613(1): 0+0+0 ms, 0 -> 0 MB 50 -> 45 (3100-3055) objects
```

合法的 unsafe.Pointer 被当做普通指针对待。

```
func test() unsafe.Pointer {
    p := &data{}
    return unsafe.Pointer(p)
}

func main() {
    const N = 10000
    cache := new([N]unsafe.Pointer)

    for i := 0; i < N; i++ {
        cache[i] = test()
        time.Sleep(time.Millisecond)
    }
}
```

输出：

```
$ go build -o test && GODEBUG="gctrace=1" ./test
```

```
gc12(1): 0+0+0 ms, 199 -> 199 MB 2088 -> 2088 (2095-7) objects
gc13(1): 0+0+0 ms, 399 -> 399 MB 4136 -> 4136 (4143-7) objects
gc14(1): 0+0+0 ms, 799 -> 799 MB 8232 -> 8232 (8239-7) objects
```

指向对象成员的 unsafe.Pointer，同样能确保对象不被回收。

```
type data struct {
    x   [1024 * 100]byte
    y   int
}

func test() unsafe.Pointer {
    d := data{}
    return unsafe.Pointer(&d.y)
}

func main() {
    const N = 10000
    cache := new([N]unsafe.Pointer)

    for i := 0; i < N; i++ {
        cache[i] = test()
        time.Sleep(time.Millisecond)
    }
}
```

输出：

```
$ go build -o test && GODEBUG="gctrace=1" ./test

gc12(1): 0+0+0 ms, 207 -> 207 MB 2088 -> 2088 (2095-7) objects
gc13(1): 1+0+0 ms, 415 -> 415 MB 4136 -> 4136 (4143-7) objects
gc14(1): 3+1+0 ms, 831 -> 831 MB 8232 -> 8232 (8239-7) objects
```

由于可以用 unsafe.Pointer、uintptr 创建 "dangling pointer" 等非法指针，所以在使用时需要特别小心。另外，cgo C.malloc 等函数所返回指针，与 GC 无关。

指针构成的 "循环引用" 加上 runtime.SetFinalizer 会导致内存泄露。

```
type Data struct {
    d   [1024 * 100]byte
    o   *Data
}

func test() {
    var a, b Data
```

```
    a.o = &b
    b.o = &a

    runtime.SetFinalizer(&a, func(d *Data) { fmt.Printf("a %p final.\n", d) })
    runtime.SetFinalizer(&b, func(d *Data) { fmt.Printf("b %p final.\n", d) })
}

func main() {
    for {
        test()
        time.Sleep(time.Millisecond)
    }
}
```

输出：

```
$ go build -gcflags "-N -l" && GODEBUG="gctrace=1" ./test

gc11(1): 2+0+0 ms, 104 -> 104 MB 1127 -> 1127 (1180-53) objects
gc12(1): 4+0+0 ms, 208 -> 208 MB 2151 -> 2151 (2226-75) objects
gc13(1): 8+0+1 ms, 416 -> 416 MB 4198 -> 4198 (4307-109) objects
```

垃圾回收器能正确处理 "指针循环引用"，但无法确定 Finalizer 依赖次序，也就无法调用 Finalizer 函数，这会导致目标对象无法变成不可达状态，其所占用内存无法被回收。

9.3 cgo

通过 cgo，可在 Go 和 C/C++ 代码间相互调用。受 CGO_ENABLED 参数限制。

```
package main

/*
   #include <stdio.h>
   #include <stdlib.h>

   void hello() {
        printf("Hello, World!\n");
   }
*/
import "C"

func main() {
    C.hello()
}
```

调试 cgo 代码是件很麻烦的事，建议单独保存到 .c 文件中。这样可以将其当做独立的 C 程序进行调试。

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

void hello();

#endif
```

test.c

```
#include <stdio.h>
#include "test.h"

void hello() {
    printf("Hello, World!\n");
}

#ifdef __TEST__                      // 避免和 Go bootstrap main 冲突。

int main(int argc, char *argv[]) {
    hello();
    return 0;
}

#endif
```

main.go

```
package main

/*
   #include "test.h"
*/
import "C"

func main() {
    C.hello()
}
```

编译和调试 C，只需在命令行提供宏定义即可。

```
$ gcc -g -D__TEST__ -o test test.c
```

由于 cgo 仅扫描当前目录，如果需要包含其他 C 项目，可在当前目录新建一个 C 文件，然后用 #include 指令将所需的 .h、.c 都包含进来，记得在 CFLAGS 中使用 "-I" 参数指定原路径。某些时候，可能还需指定 "-std" 参数。

9.3.1 Flags

可使用 #cgo 命令定义 CFLAGS、LDFLAGS 等参数，自动合并多个设置。

```
/*
   #cgo CFLAGS: -g
   #cgo CFLAGS: -I./lib -D__VER__=1
   #cgo LDFLAGS: -lpthread

   #include "test.h"
*/
import "C"
```

可设置 GOOS、GOARCH 编译条件，空格表示 OR，逗号 AND，感叹号 NOT。

```
#cgo windows,386 CFLAGS: -I./lib -D__VER__=1
```

9.3.2 DataType

数据类型对应关系。

```
 C                    cgo                   sizeof
--------------------+---------------------+----------------------------------------------
 char                 C.char                1
 signed char          C.schar               1
 unsigned char        C.uchar               1
 short                C.short               2
 unsigned short       C.ushort              2
 int                  C.int                 4
 unsigned int         C.uint                4
 long                 C.long                4 或 8
 unsigned long        C.ulong               4 或 8
 long long            C.longlong            8
 unsinged long long   C.ulonglong           8
 float                C.float               4
 double               C.double              8
 void*                unsafe.Pointer
```

```
 char*                *C.char
 size_t               C.size_t
 NULL                 nil
```

可将 cgo 类型转换为标准 Go 类型。

```
/*
   int add(int x, int y) {
        return x + y;
   }
*/
import "C"

func main() {
    var x C.int = C.add(1, 2)
    var y int = int(x)
    fmt.Println(x, y)
}
```

9.3.3 String

字符串转换函数。

```
/*
   #include <stdio.h>
   #include <stdlib.h>

   void test(char *s) {
       printf("%s\n", s);
   }

   char* cstr() {
       return "abcde";
   }
*/
import "C"

func main() {
    s := "Hello, World!"

    cs := C.CString(s)                  // 该函数在 C heap 分配内存，需要调用 free 释放。
    defer C.free(unsafe.Pointer(cs))    // #include <stdlib.h>

    C.test(cs)
```

```
    cs = C.cstr()

    fmt.Println(C.GoString(cs))
    fmt.Println(C.GoStringN(cs, 2))
    fmt.Println(C.GoBytes(unsafe.Pointer(cs), 2))
}
```

输出：

```
Hello, World!
abcde
ab
[97 98]
```

用 C.malloc/free 分配 C heap 内存。

```
/*
   #include <stdlib.h>
*/
import "C"

func main() {
    m := unsafe.Pointer(C.malloc(4 * 8))
    defer C.free(m)                          // 注释释放内存。

    p := (*[4]int)(m)                        // 转换为数组指针。
    for i := 0; i < 4; i++ {
        p[i] = i + 100
    }

    fmt.Println(p)
}
```

输出：

```
&[100 101 102 103]
```

9.3.4 Struct/Enum/Union

对 struct、enum 支持良好，union 会被转换成字节数组。如果没使用 typedef 定义，那么必须添加 struct_、enum_、union_ 前缀。

struct

```
/*
   #include <stdlib.h>

   struct Data {
       int x;
```

```
   };

   typedef struct {
       int x;
   } DataType;

   struct Data* testData() {
       return malloc(sizeof(struct Data));
   }

   DataType* testDataType() {
       return malloc(sizeof(DataType));
   }
*/
import "C"

func main() {
    var d *C.struct_Data = C.testData()
    defer C.free(unsafe.Pointer(d))

    var dt *C.DataType = C.testDataType()
    defer C.free(unsafe.Pointer(dt))

    d.x = 100
    dt.x = 200

    fmt.Printf("%#v\n", d)
    fmt.Printf("%#v\n", dt)
}
```

输出：

```
&main._Ctype_struct_Data{x:100}
&main._Ctype_DataType{x:200}
```

enum

```
/*
   enum Color { BLACK = 10, RED, BLUE };
   typedef enum { INSERT = 3, DELETE } Mode;
*/
import "C"

func main() {
    var c C.enum_Color = C.RED
    var x uint32 = c
    fmt.Println(c, x)

    var m C.Mode = C.INSERT
    fmt.Println(m)
```

```
}
```

union

```
/*
   #include <stdlib.h>

   union Data {
       char x;
       int y;
   };

   union Data* test() {
       union Data* p = malloc(sizeof(union Data));
       p->x = 100;
       return p;
   }
*/
import "C"

func main() {
    var d *C.union_Data = C.test()
    defer C.free(unsafe.Pointer(d))

    fmt.Println(d)
}
```

输出：

```
&[100 0 0 0]
```

9.3.5 Export

导出 Go 函数给 C 调用，须使用 "//export" 标记。建议在独立头文件中声明函数原型，避免 "duplicate symbol" 错误。

main.go

```
package main

import "fmt"

/*
   #include "test.h"
*/
import "C"

//export hello
```

```
func hello() {
    fmt.Println("Hello, World!\n")
}

func main() {
    C.test()
}
```

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

extern void hello();
void test();

#endif
```

test.c

```
#include <stdio.h>
#include "test.h"

void test() {
    hello();
}
```

9.3.6 Shared Library

在 cgo 中使用 C 共享库。

test.h

```
#ifndef __TEST_HEAD__
#define __TEST_HEAD__

int sum(int x, int y);

#endif
```

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

int sum(int x, int y)
{
```

```
    return x + y + 100;
}
```

编译成 .so 或 .dylib。

```
$ gcc -c -fPIC -o test.o test.c
$ gcc -dynamiclib -o libtest.dylib test.o
```

将共享库和头文件拷贝到 Go 项目目录。

main.go

```
package main

/*
   #cgo CFLAGS: -I.
   #cgo LDFLAGS: -L. -ltest
   #include "test.h"
*/
import "C"

func main() {
    println(C.sum(10, 20))
}
```

输出：

```
$ go build -o test && ./test
130
```

编译成功后可用 ldd 或 otool 查看动态库使用状态。 静态库使用方法类似。

9.4 Reflect

没有运行期类型对象，实例也没有附加字段用来表明身份。只有转换成接口时，才会在其 itab 内部存储与该类型有关的信息，Reflect 所有操作都依赖于此。

9.4.1 Type

以 struct 为例，可获取其全部成员字段信息，包括非导出和匿名字段。

```
type User struct {
    Username string
```

```
}

type Admin struct {
    User
    title string
}

func main() {
    var u Admin
    t := reflect.TypeOf(u)

    for i, n := 0, t.NumField(); i < n; i++ {
        f := t.Field(i)
        fmt.Println(f.Name, f.Type)
    }
}
```

输出：

```
User main.User     // 可进一步递归。
title string
```

如果是指针，应该先使用 Elem 方法获取目标类型，指针本身是没有字段成员的。

```
func main() {
    u := new(Admin)

    t := reflect.TypeOf(u)
    if t.Kind() == reflect.Ptr {
        t = t.Elem()
    }

    for i, n := 0, t.NumField(); i < n; i++ {
        f := t.Field(i)
        fmt.Println(f.Name, f.Type)
    }
}
```

同样，value-interface 和 pointer-interface 也会导致方法集存在差异。

```
type User struct {
}

type Admin struct {
    User
}

func (*User) ToString() {}
```

```
func (Admin) test()     {}

func main() {
    var u Admin

    methods := func(t reflect.Type) {
        for i, n := 0, t.NumMethod(); i < n; i++ {
            m := t.Method(i)
            fmt.Println(m.Name)
        }
    }

    fmt.Println("--- value interface ---")
    methods(reflect.TypeOf(u))

    fmt.Println("--- pointer interface ---")
    methods(reflect.TypeOf(&u))
}
```

输出：

```
--- value interface ---
test
--- pointer interface ---
ToString
test
```

可直接用名称或序号访问字段，包括用多级序号访问嵌入字段成员。

```
type User struct {
    Username string
    age      int
}

type Admin struct {
    User
    title string
}

func main() {
    var u Admin
    t := reflect.TypeOf(u)

    f, _ := t.FieldByName("title")
    fmt.Println(f.Name)

    f, _ = t.FieldByName("User")       // 访问嵌入字段。
    fmt.Println(f.Name)
```

```
    f, _ = t.FieldByName("Username")   // 直接访问嵌入字段成员，会自动深度查找。
    fmt.Println(f.Name)

    f = t.FieldByIndex([]int{0, 1})    // Admin[0] -> User[1] -> age
    fmt.Println(f.Name)
}
```

输出：

```
title
User
Username
age
```

字段标签可实现简单元数据编程，比如标记 ORM Model 属性。

```
type User struct {
    Name string `field:"username" type:"nvarchar(20)"`
    Age  int    `field:"age" type:"tinyint"`
}

func main() {
    var u User

    t := reflect.TypeOf(u)
    f, _ := t.FieldByName("Name")

    fmt.Println(f.Tag)
    fmt.Println(f.Tag.Get("field"))
    fmt.Println(f.Tag.Get("type"))
}
```

输出：

```
field:"username" type:"nvarchar(20)"
username
nvarchar(20)
```

可从基本类型获取所对应复合类型。

```
var (
    Int    = reflect.TypeOf(0)
    String = reflect.TypeOf("")
)

func main() {
    c := reflect.ChanOf(reflect.SendDir, String)
    fmt.Println(c)

    m := reflect.MapOf(String, Int)
```

```
    fmt.Println(m)

    s := reflect.SliceOf(Int)
    fmt.Println(s)

    t := struct{ Name string }{}
    p := reflect.PtrTo(reflect.TypeOf(t))
    fmt.Println(p)
}
```

输出:

```
chan<- string
map[string]int
[]int
*struct { Name string }
```

与之对应，方法 Elem 可返回复合类型的基类型。

```
func main() {
    t := reflect.TypeOf(make(chan int)).Elem()
    fmt.Println(t)
}
```

方法 Implements 判断是否实现了某个具体接口，AssignableTo、ConvertibleTo 用于赋值和转换判断。

```
type Data struct {
}

func (*Data) String() string {
    return ""
}

func main() {
    var d *Data
    t := reflect.TypeOf(d)

    // 没法直接获取接口类型，好在接口本身是个 struct，创建
    // 一个空指针对象，这样传递给 TypeOf 转换成 interface{}
    // 时就有类型信息了。。
    it := reflect.TypeOf((*fmt.Stringer)(nil)).Elem()

    // 为啥不是 t.Implements(fmt.Stringer)，完全可以由编译器生成。
    fmt.Println(t.Implements(it))
}
```

某些时候，获取对齐信息对于内存自动分析是很有用的。

```
type Data struct {
    b   byte
    x   int32
}

func main() {
    var d Data

    t := reflect.TypeOf(d)
    fmt.Println(t.Size(), t.Align())   // sizeof，以及最宽字段的对齐模数。

    f, _ := t.FieldByName("b")
    fmt.Println(f.Type.FieldAlign())   // 字段对齐。
}
```

输出：

```
8  4
1
```

9.4.2 Value

Value 和 Type 使用方法类似，包括使用 Elem 获取指针目标对象。

```
type User struct {
    Username string
    age      int
}

type Admin struct {
    User
    title string
}

func main() {
    u := &Admin{User{"Jack", 23}, "NT"}
    v := reflect.ValueOf(u).Elem()

    fmt.Println(v.FieldByName("title").String())   // 用转换方法获取字段值
    fmt.Println(v.FieldByName("age").Int())        // 直接访问嵌入字段成员
    fmt.Println(v.FieldByIndex([]int{0, 1}).Int()) // 用多级序号访问嵌入字段成员
}
```

输出：

```
NT
```

```
23
23
```

除具体的 Int、String 等转换方法，还可返回 interface{}。只是非导出字段无法使用，需用 CanInterface 判断一下。

```
type User struct {
    Username string
    age      int
}

func main() {
    u := User{"Jack", 23}
    v := reflect.ValueOf(u)

    fmt.Println(v.FieldByName("Username").Interface())
    fmt.Println(v.FieldByName("age").Interface())
}
```

输出：

```
Jack

panic: reflect.Value.Interface: cannot return value obtained from unexported field or
method
```

当然，转换成具体类型不会引发 panic。

```
func main() {
    u := User{"Jack", 23}
    v := reflect.ValueOf(u)

    f := v.FieldByName("age")

    if f.CanInterface() {
        fmt.Println(f.Interface())
    } else {
        fmt.Println(f.Int())
    }
}
```

除 struct，其他复合类型 array、slice、map 取值示例。

```
func main() {
    v := reflect.ValueOf([]int{1, 2, 3})
    for i, n := 0, v.Len(); i < n; i++ {
        fmt.Println(v.Index(i).Int())
```

```
    }

    fmt.Println("--------------------------")

    v = reflect.ValueOf(map[string]int{"a": 1, "b": 2})
    for _, k := range v.MapKeys() {
        fmt.Println(k.String(), v.MapIndex(k).Int())
    }
}
```

输出:

```
1
2
3
--------------------------
a 1
b 2
```

需要注意，Value 某些方法没有遵循 "comma ok" 模式，而是返回 ZeroValue，因此需要用 IsValid 判断一下是否可用。

```
func (v Value) FieldByName(name string) Value {
    v.mustBe(Struct)
    if f, ok := v.typ.FieldByName(name); ok {
        return v.FieldByIndex(f.Index)
    }
    return Value{}
}
```

```
type User struct {
    Username string
    age      int
}

func main() {
    u := User{}
    v := reflect.ValueOf(u)

    f := v.FieldByName("a")
    fmt.Println(f.Kind(), f.IsValid())
}
```

输出:

```
invalid false
```

另外，接口是否为 nil，需要 tab 和 data 都为空。可使用 IsNil 方法判断 data 值。

```
func main() {
    var p *int

    var x interface{} = p
    fmt.Println(x == nil)

    v := reflect.ValueOf(p)
    fmt.Println(v.Kind(), v.IsNil())
}
```

输出：

```
false
ptr true
```

将对象转换为接口，会发生复制行为。该复制品只读，无法被修改。所以要通过接口改变目标对象状态，必须是 pointer-interface。

就算是指针，我们依然没法将这个存储在 data 的指针指向其他对象，只能透过它修改目标对象。因为目标对象并没有被复制，被复制的只是指针。

```
type User struct {
    Username string
    age      int
}

func main() {
    u := User{"Jack", 23}

    v := reflect.ValueOf(u)
    p := reflect.ValueOf(&u)

    fmt.Println(v.CanSet(), v.FieldByName("Username").CanSet())
    fmt.Println(p.CanSet(), p.Elem().FieldByName("Username").CanSet())
}
```

输出：

```
false  false
false  true
```

非导出字段无法直接修改，可改用指针操作。

```
type User struct {
    Username string
    age      int
}

func main() {
```

```
    u := User{"Jack", 23}
    p := reflect.ValueOf(&u).Elem()

    p.FieldByName("Username").SetString("Tom")

    f := p.FieldByName("age")
    fmt.Println(f.CanSet())

    // 判断是否能获取地址。
    if f.CanAddr() {
        age := (*int)(unsafe.Pointer(f.UnsafeAddr()))
        // age := (*int)(unsafe.Pointer(f.Addr().Pointer()))  // 等同
        *age = 88
    }

    // 注意 p 是 Value 类型，需要还原成接口才能转型。
    fmt.Println(u, p.Interface().(User))
}
```

输出:

```
false
{Tom 88} {Tom 88}
```

复合类型修改示例。

```
func main() {
    s := make([]int, 0, 10)
    v := reflect.ValueOf(&s).Elem()

    v.SetLen(2)
    v.Index(0).SetInt(100)
    v.Index(1).SetInt(200)

    fmt.Println(v.Interface(), s)

    v2 := reflect.Append(v, reflect.ValueOf(300))
    v2 = reflect.AppendSlice(v2, reflect.ValueOf([]int{400, 500}))

    fmt.Println(v2.Interface())

    fmt.Println("----------------------")

    m := map[string]int{"a": 1}
    v = reflect.ValueOf(&m).Elem()

    v.SetMapIndex(reflect.ValueOf("a"), reflect.ValueOf(100)) // update
    v.SetMapIndex(reflect.ValueOf("b"), reflect.ValueOf(200)) // add
```

```
    fmt.Println(v.Interface(), m)
}
```

输出:

```
[100 200] [100 200]
[100 200 300 400 500]
----------------------
map[a:100 b:200] map[a:100 b:200]
```

9.4.3 Method

可获取方法参数、返回值类型等信息。

```
type Data struct {
}

func (*Data) Test(x, y int) (int, int) {
    return x + 100, y + 100
}

func (*Data) Sum(s string, x ...int) string {
    c := 0
    for _, n := range x {
        c += n
    }

    return fmt.Sprintf(s, c)
}

func info(m reflect.Method) {
    t := m.Type

    fmt.Println(m.Name)

    for i, n := 0, t.NumIn(); i < n; i++ {
        fmt.Printf("  in[%d] %v\n", i, t.In(i))
    }

    for i, n := 0, t.NumOut(); i < n; i++ {
        fmt.Printf("  out[%d] %v\n", i, t.Out(i))
    }
}

func main() {
    d := new(Data)
    t := reflect.TypeOf(d)
```

```
    test, _ := t.MethodByName("Test")
    info(test)

    sum, _ := t.MethodByName("Sum")
    info(sum)
}
```

输出:

```
Test
  in[0] *main.Data    // receiver
  in[1] int
  in[2] int
  out[0] int
  out[1] int
Sum
  in[0] *main.Data
  in[1] string
  in[2] []int
  out[0] string
```

动态调用方法很简单，按 In 列表准备好所需参数即可 (不包括 receiver)。

```
func main() {
    d := new(Data)
    v := reflect.ValueOf(d)

    exec := func(name string, in []reflect.Value) {
        m := v.MethodByName(name)
        out := m.Call(in)

        for _, v := range out {
            fmt.Println(v.Interface())
        }
    }

    exec("Test", []reflect.Value{
        reflect.ValueOf(1),
        reflect.ValueOf(2),
    })

    fmt.Println("------------------------")

    exec("Sum", []reflect.Value{
        reflect.ValueOf("result = %d"),
        reflect.ValueOf(1),
        reflect.ValueOf(2),
    })
```

```
}
```

输出:

```
101
102
----------------------
result = 3
```

如改用 CallSlice，只需将变参打包成 slice 即可。

```
func main() {
    d := new(Data)
    v := reflect.ValueOf(d)

    m := v.MethodByName("Sum")

    in := []reflect.Value{
        reflect.ValueOf("result = %d"),
        reflect.ValueOf([]int{1, 2}),    // 将变参打包成 slice。
    }

    out := m.CallSlice(in)

    for _, v := range out {
        fmt.Println(v.Interface())
    }
}
```

非导出方法无法调用，甚至无法透过指针操作，因为接口类型信息中没有该方法地址。

9.4.4 Make

利用 Make、New 等函数，可实现近似泛型操作。

```
var (
    Int    = reflect.TypeOf(0)
    String = reflect.TypeOf("")
)

func Make(T reflect.Type, fptr interface{}) {

    // 实际创建 slice 的包装函数。
    swap := func(in []reflect.Value) []reflect.Value {

        // --- 省略算法内容 --- //
```

```
        // 返回和类型匹配的 slice 对象。
        return []reflect.Value{
            reflect.MakeSlice(
                reflect.SliceOf(T),   // slice type
                int(in[0].Int()),     // len
                int(in[1].Int())      // cap
            ),
        }
    }

    // 传入的是函数变量指针，因为我们要将变量指向 swap 函数。
    fn := reflect.ValueOf(fptr).Elem()

    // 获取函数指针类型，生成所需 swap function value。
    v := reflect.MakeFunc(fn.Type(), swap)

    // 修改函数指针实际指向，也就是 swap。
    fn.Set(v)
}

func main() {
    var makeints func(int, int) []int
    var makestrings func(int, int) []string

    // 用相同算法，生成不同类型创建函数。
    Make(Int, &makeints)
    Make(String, &makestrings)

    // 按实际类型使用。
    x := makeints(5, 10)
    fmt.Printf("%#v\n", x)

    s := makestrings(3, 10)
    fmt.Printf("%#v\n", s)
}
```

输出：

```
[]int{0, 0, 0, 0, 0}
[]string{"", "", ""}
```

原理并不复杂。

1. 核心是提供一个 swap 函数，其中利用 reflect.MakeSlice 生成最终 slice 对象，因此需要传入 element type、len、cap 参数。
2. 接下来，利用 MakeFunc 函数生成 swap value，并修改函数变量指向，以达到调用 swap 的目的。

3. 当调用具体类型的函数变量时，实际内部调用的是 swap，相关代码会自动转换参数列表，并将返回结果还原成具体类型返回值。

如此，在共享算法的前提下，无须用 interface{}，无须做类型转换，颇有泛型的效果。

第二部分 源码

基于 Go 1.4，相关文件位于 src/runtime 目录。
文章忽略了 32bit 代码，有兴趣的可自行查看源码文件。
为便于阅读，示例代码做过裁剪。

1. Memory Allocator

Go 内存分配器基于 tcmalloc 模型，这在 malloc.h 头部注释中有明确说明。

```
Memory allocator, based on tcmalloc.
http://goog-perftools.sourceforge.net/doc/tcmalloc.html
```

核心目标很简单：

- 从 mmap 申请大块内存，自主管理，减少系统调用。
- 基于块的内存复用体系，加快内存分配和回收操作。

分配器以页为单位向操作系统申请大块内存。这些大块内存由 n 个地址连续的页组成，并用名为 span 的对象进行管理。

malloc.h

```
PageShift    = 13,
PageSize     = 1<<PageShift,     // 8192 bytes
```

当需要时，span 所管理内存被切分成多个大小相等的小块，每个小块可存储一个对象，故称作 object。

分配器以 32KB 为界，将对象分为大小两种。

malloc.h

```
MaxSmallSize = 32<<10,
```

大对象直接找一个大小合适的 span，这个无需多言。小对象则以 8 的倍数分为不同大小等级 (size class)。比如 class1 为 8 字节，可存储 1 ~ 8 字节大小的对象。

```
NumSizeClasses = 67,
```

当然，实际的对应规则并不是连续和固定的，会根据一些经验和测试结果进行调整，以获得最佳的性能和内存利用率。

malloc.h

```
// Size classes.  Computed and initialized by InitSizes.
//
// SizeToClass(0 <= n <= MaxSmallSize) returns the size class,
```

```
//     1 <= sizeclass < NumSizeClasses, for n.
//     Size class 0 is reserved to mean "not small".
//
// class_to_size[i] = largest size in class i
// class_to_allocnpages[i] = number of pages to allocate when
//     making new objects in class i

int32  runtime·SizeToClass(int32);

extern int32  runtime·class_to_size[NumSizeClasses];
extern int32  runtime·class_to_allocnpages[NumSizeClasses];
extern int8   runtime·size_to_class8[1024/8 + 1];
extern int8   runtime·size_to_class128[(MaxSmallSize-1024)/128 + 1];
```

为了管理好内存，分配器使用三级组件来完成不同操作。

- **heap**: 全局根对象。负责向操作系统申请内存，管理由垃圾回收器收回的空闲 span 内存块。
- **central**: 从 heap 获取空闲 span，并按需要将其切分成 object 块。heap 管理着多个 central 对象，每个 central 负责处理一种等级的内存分配需求。
- **cache**: 运行期，每个 cache 都与某个具体线程相绑定，实现无锁内存分配操作。其内部有个以等级为序号的数组，持有多个切分好的 span 对象。缺少空间时，向等级对应的 central 获取新的 span 即可。

简单描述一下内存分配和回收流程。

分配流程:

- 通过 size class 反查表计算待分配对象等级。
- 从 cache.alloc[sizeclass] 找到等级相同的 span。
- 从 span 切分好的链表中提取可用 object。
- 如 span 没剩余空间，则从 heap.central[sizeclass] 找到对应 central，获取 span。
- 如 central 没可用 span，则向 heap 申请，并切割成所需等级的 object 链表。
- 如 heap 也没有多余 span，那么就向操作系统申请新的内存。

回收流程:

- 垃圾回收器或其他行为引发内存回收操作。
- 将可回收 object 交还给所属 span。
- 将 span 交给对应 central 管理，以便某个 cache 重新获取。

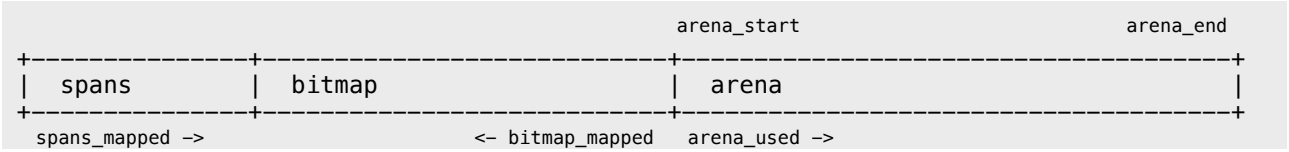
- 如 span 内存全部收回，那么将其返还给 heap，以便被重新切分复用。
- 垃圾回收器定期扫描 heap 所管理的空闲 spans，释放超期不用的物理内存。

从 heap 申请和回收 span 的过程中，分配器会尝试合并地址相邻的 span 块，以形成更大内存块，减少碎片。

1.1 初始化

分配器管理算法依赖连续内存地址。因此，在初始化时，分配器会预留一块巨大的虚拟地址空间。该空间被成三个部分：

- **arena**: 用户内存实际分配范围。
- **bitmap**: 为每个地址提供 4bit 标记位，用于垃圾回收操作。
- **spans**: 记录每个页所对应 span 地址，用于反查和合并操作。



在 64 位系统下，arena 最大容量是 128GB，bitmap 8GB，spans 128MB。这些内存并非一次性分配，而是随着 arena 线性增加，每个区域都有指针标记当前分配位置。

malloc.h

```
struct MHeap
{
    // span lookup
    MSpan** spans;
    uintptr spans_mapped;

    // range of addresses we might see in the heap
    byte *bitmap;
    uintptr bitmap_mapped;
    byte *arena_start;
    byte *arena_used;
    byte *arena_end;
    bool arena_reserved;
};
```

虚拟地址预留操作并非物理内存分配，因此看到 “Hello, World” 消耗上百 GB “内存”，无需大惊小怪。

在运行时初始化时，会调用内存分配器初始化函数。

proc.c

```
void runtime·schedinit(void)
{
    runtime·mallocinit();
}
```

malloc.c

```
void runtime·mallocinit(void)
{
    // 初始化 size class 反查表。
    runtime·InitSizes();

    // 64-bit
    if(sizeof(void*) == 8 && (limit == 0 || limit > (1<<30))) {
        arena_size = MaxMem; // 128GB
        bitmap_size = arena_size / (sizeof(void*)*8/4); // 8GB
        spans_size = arena_size / PageSize * sizeof(runtime·mheap.spans[0]);
        spans_size = ROUND(spans_size, PageSize); // 128MB

        // 尝试从 0xc000000000 开始设置保留地址。
        // 如果失败，则尝试 0x1c000000000 ~ 0x7fc000000000。
        for(i = 0; i <= 0x7f; i++) {
            p = (void*)(i<<40 | 0x00c0ULL<<32);
            p_size = bitmap_size + spans_size + arena_size + PageSize;
            p = runtime·SysReserve(p, p_size, &reserved);
            if(p != nil)
                break;
        }
    }

    // 32-bit
    if (p == nil) {
        // 忽略
    }

    // 按 PageSize 对齐地址。
    // 分配器使用 Address<<PageShift 作为 PageID。
    p1 = (byte*)ROUND((uintptr)p, PageSize);

    // 设定不同区域的起始地址。
    runtime·mheap.spans = (MSpan**)p1;
    runtime·mheap.bitmap = p1 + spans_size;
    runtime·mheap.arena_start = p1 + spans_size + bitmap_size;
    runtime·mheap.arena_used = runtime·mheap.arena_start;
```

```
    runtime·mheap.arena_end = p + p_size;
    runtime·mheap.arena_reserved = reserved;

    // 初始化 heap 和当前 cache。
    runtime·MHeap_Init(&runtime·mheap);
    g->m->mcache = runtime·allocmcache();
}
```

内存地址预留操作通过 mmap PORT_NONE 实现。不过，在 darwin/OSX 中，并未使用 MAP_FIXED 参数，因此未必从 0xc000000000 开始。

mem_darwin.c

```
void* runtime·SysReserve(void *v, uintptr n, bool *reserved)
{
    void *p;

    *reserved = true;
    p = runtime·mmap(v, n, PROT_NONE, MAP_ANON|MAP_PRIVATE, -1, 0);
    if(p < (void*)4096)
        return nil;
    return p;
}
```

分配器根对象 heap 的初始化工作，主要是几个 span 管理链表和 central 数组的创建。

malloc.h

```
MaxMHeapList = 1<<(20 - PageShift), // Maximum page length for fixed-size list in MHeap.

struct MHeap
{
    MSpan free[MaxMHeapList];   // free lists of given length
    MSpan busy[MaxMHeapList];   // busy lists of large objects of given length

    MSpan freelarge;            // free lists length >= MaxMHeapList
    MSpan busylarge;            // busy lists of large objects length >= MaxMHeapList

    struct MHeapCentral {
        MCentral mcentral;
        byte pad[CacheLineSize];
    } central[NumSizeClasses];
};
```

其中，free 和 busy 数组以 span 页数为序号管理多个链表。当 central 有需要时，只需从 free 找到页数合适的链表，从中提取可用 span 即可。busy 记录的自然是已经被使用的 span。

至于 large 链表，用于保存所有超出 free/busy 页数限制的 span。

mheap.c

```
void runtime·MHeap_Init(MHeap *h)
{
    uint32 i;

    // 初始化一些管理类型的固定分配器。
    runtime·FixAlloc_Init(&h->spanalloc, sizeof(MSpan), RecordSpan, ...);
    runtime·FixAlloc_Init(&h->cachealloc, sizeof(MCache), ...);
    runtime·FixAlloc_Init(&h->specialfinalizeralloc, sizeof(SpecialFinalizer), ...);
    runtime·FixAlloc_Init(&h->specialprofilealloc, sizeof(SpecialProfile), ...);

    // 初始化 free/busy 数组。
    for(i=0; i<nelem(h->free); i++) {
        runtime·MSpanList_Init(&h->free[i]);
        runtime·MSpanList_Init(&h->busy[i]);
    }

    // 初始化 large 链表。
    runtime·MSpanList_Init(&h->freelarge);
    runtime·MSpanList_Init(&h->busylarge);

    // 创建所有等级的 central 对象。
    for(i=0; i<nelem(h->central); i++)
        runtime·MCentral_Init(&h->central[i].mcentral, i);
}
```

像 span、cache 这类管理对象，并不从 arena 区域分配，而是使用专门的 FixAlloc 分配器单独管理。其具体实现细节可参考后续章节。

在 span 内部有两个指针，用于将多个对象串成双向链表。

malloc.h

```
struct MSpan
{
    MSpan   *next;      // in a span linked list
    MSpan   *prev;      // in a span linked list

    pageID  start;      // starting page number
```

```
    uintptr npages;     // number of pages in span
    MLink   *freelist;  // list of free objects

    uint8   sizeclass;  // size class
    uint8   state;      // MSpanInUse etc
    uintptr elemsize;   // computed from sizeclass or from npages
};
```

mheap.c

```
void runtime·MSpanList_Init(MSpan *list)
{
    list->state = MSpanListHead;
    list->next = list;
    list->prev = list;
}
```

至于 central，同样是完成两个 span 管理链表的初始化操作。其中 nonempty 链表保存有剩余 object 空间，等待被 cache 获取的 span。而 empty 则保存没有剩余空间或已被 cache 获取的 span。

malloc.h

```
struct MCentral
{
    int32 sizeclass;
    MSpan nonempty; // list of spans with a free object
    MSpan empty;    // list of spans with no free objects (or cached in an MCache)
};
```

mcentral.c

```
void runtime·MCentral_Init(MCentral *c, int32 sizeclass)
{
    c->sizeclass = sizeclass;
    runtime·MSpanList_Init(&c->nonempty);
    runtime·MSpanList_Init(&c->empty);
}
```

最后，用固定分配器创建 cache 对象，并初始化其 alloc 数组。

malloc.h

```
struct MCache
{
    MSpan*  alloc[NumSizeClasses];  // spans to allocate from
};
```

mcache.c

```
// dummy MSpan that contains no free objects.
MSpan runtime·emptymspan;

MCache* runtime·allocmcache(void)
{
    // 使用固定分配器创建 cache 对象。
    c = runtime·FixAlloc_Alloc(&runtime·mheap.cachealloc);

    // 初始化内存。
    runtime·memclr((byte*)c, sizeof(*c));

    // 初始化 alloc 数组，用来保存从 central 获取的不同等级 span 对象。
    for(i = 0; i < NumSizeClasses; i++)
        c->alloc[i] = &runtime·emptymspan;

    return c;
}
```

1.2 分配流程

相关包装函数，最终通过 mallocgc 函数完成内存分配操作。

malloc.go

```
func newobject(typ *_type) unsafe.Pointer {
    return mallocgc(uintptr(typ.size), typ, flags)
}

func newarray(typ *_type, n uintptr) unsafe.Pointer {
    return mallocgc(uintptr(typ.size)*n, typ, flags)
}
```

在分配过程中，需要判断大小对象，还有对小于 16 字节的微小对象做额外处理。

malloc.h

```
MaxSmallSize = 32<<10,

TinySize = 16,
TinySizeClass = 2,
```

malloc.go

```
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Pointer {
    // 当前 cache 对象。
```

```
c := gomcache()

var s *mspan
var x unsafe.Pointer

// 判断是否小对象。
if size <= maxSmallSize {
    // 对于小于 16 字节的微小对象，做额外处理。
    if flags&flagNoScan != 0 && size < maxTinySize {
        // 获取当前 cache tiny 块剩余大小。
        tinysize := uintptr(c.tinysize)

        // 如果 tiny 块空间足够...
        if size <= tinysize {
            tiny := unsafe.Pointer(c.tiny)

            // 地址对齐。
            if size&7 == 0 {
                tiny = roundup(tiny, 8)
            } else if size&3 == 0 {
                tiny = roundup(tiny, 4)
            } else if size&1 == 0 {
                tiny = roundup(tiny, 2)
            }

            // 实际大小 = 对象大小 + 对齐所需大小(对齐后地址 - 原地址)。
            size1 := size + (uintptr(tiny) - uintptr(unsafe.Pointer(c.tiny)))

            // 再次判断空间是否足够...
            if size1 <= tinysize {
                // x = 对齐后地址
                x = tiny

                // 调整剩余空间记录。
                c.tiny = (*byte)(add(x, size))
                c.tinysize -= uintptr(size1)
                c.local_tinyallocs++

                return x
            }
        }

        // 如果 tiny 块空间不足，则从 alloc[2] 获取新的 tiny/object 块。
        s = c.alloc[tinySizeClass]
        v := s.freelist

        // 如果该 span 没有可用 object ...
        if v == nil {
```

```
            // 从 central 获取新的 span。
            mp := acquirem()
            mp.scalararg[0] = tinySizeClass
            onM(mcacheRefill_m)
            releasem(mp)

            // 获取 tiny/object 块。
            s = c.alloc[tinySizeClass]
            v = s.freelist
        }

        // 提取 tiny 块后，调整 span.freelist 链表。
        s.freelist = v.next
        s.ref++

        // 初始化 tiny 块内存。
        x = unsafe.Pointer(v)
        (*[2]uint64)(x)[0] = 0
        (*[2]uint64)(x)[1] = 0

        // 如果新 tiny 块剩余空间大于原 tiny 块，那么就换一下。
        if maxTinySize-size > tinysize {
            // 调整剩余位置指针和大小。
            c.tiny = (*byte)(add(x, size))
            c.tinysize = uintptr(maxTinySize - size)
        }
        size = maxTinySize
    } else { // 普通小对象
        var sizeclass int8

        // 计算对应的等级。
        if size <= 1024-8 {
            sizeclass = size_to_class8[(size+7)>>3]
        } else {
            sizeclass = size_to_class128[(size-1024+127)>>7]
        }
        size = uintptr(class_to_size[sizeclass])

        // 从 alloc 数组获取对应的 span。
        s = c.alloc[sizeclass]

        // 从 span 链表提取 object。
        v := s.freelist

        // 如果 span 没有剩余 object，则从 central 获取新的 span。
        if v == nil {
            mp := acquirem()
            mp.scalararg[0] = uintptr(sizeclass)
```

```
                onM(mcacheRefill_m)
                releasem(mp)
                s = c.alloc[sizeclass]
                v = s.freelist
            }

            // 调整 span 链表。
            s.freelist = v.next
            s.ref++

            // 初始化内存。
            x = unsafe.Pointer(v)
            if flags&flagNoZero == 0 {
                v.next = nil
                if size > 2*ptrSize && ((*[2]uintptr)(x))[1] != 0 {
                    memclr(unsafe.Pointer(v), size)
                }
            }
        }
        c.local_cachealloc += intptr(size)
    } else { // 大对象
        mp := acquirem()
        mp.scalararg[0] = uintptr(size)
        mp.scalararg[1] = uintptr(flags)

        // 直接从 heap 分配一个适用的 span。
        // onM 是切换到 M.g0 栈执行函数，相关细节参考后续章节。
        onM(largeAlloc_m)

        s = (*mspan)(mp.ptrarg[0])
        mp.ptrarg[0] = nil
        releasem(mp)
        x = unsafe.Pointer(uintptr(s.start << pageShift))
        size = uintptr(s.elemsize)
    }


    // 在 bitmap 做标记。
    {
        arena_start := uintptr(unsafe.Pointer(mheap_.arena_start))
        off := (uintptr(x) - arena_start) / ptrSize
        xbits := (*uint8)(unsafe.Pointer(arena_start - off/wordsPerBitmapByte - 1))
        shift := (off % wordsPerBitmapByte) * gcBits

        // ...
    }

marked:
```

```
    // 检查分配计数器，以决定是否触发垃圾回收操作。
    if memstats.heap_alloc >= memstats.next_gc {
        gogc(0)
    }

    return x
}
```

函数虽然有点长，但不算太复杂。

- 尽可能将微小对象组合到一个 16 字节的 tiny 块中，据说可提高性能。
- 小对象从 cache.alloc[] 找到等级相同的 span，并从其 freelist 链表获取 object。
- 大对象直接从 heap 分配。

malloc.h

```
struct MCache
{
    // Allocator cache for tiny objects w/o pointers.
    byte*   tiny;
    uintptr tinysize;

    MSpan*  alloc[NumSizeClasses];  // spans to allocate from
};
```

除基本的分配操作外，还需要关注内存不足时的 “扩张” 过程。这需要一点耐心和细心。

首先，当 cache.alloc[] 中对应的 span 没有剩余 object 时，会触发从 central 获取新 span 操作。

malloc.c

```
void runtime·mcacheRefill_m(void)
{
    runtime·MCache_Refill(g->m->mcache, (int32)g->m->scalararg[0]);
}
```

mcache.c

```
MSpan* runtime·MCache_Refill(MCache *c, int32 sizeclass)
{
    MSpan *s;

    // 当前没有剩余空间的 span。
    s = c->alloc[sizeclass];
    if(s->freelist != nil)
```

```
        runtime·throw("refill on a nonempty span");

    // 取消 incache 标记。
    if(s != &runtime·emptymspan)
        s->incache = false;

    // 从 heap.central[] 数组找到对应的 central，并获取新的 span。
    s = runtime·MCentral_CacheSpan(&runtime·mheap.central[sizeclass].mcentral);

    // 保存到 cache.alloc 数组。
    c->alloc[sizeclass] = s;

    return s;
}
```

从 central 新获取的 span 会替代原有对象，被保存到 alloc 数组中。

需要提前说明一点背景知识：从 Go 1.3 开始，垃圾回收算法就有很大变动。其中标记阶段需要执行 StopTheWorld，然后用多线程并发执行标记操作。待标记结束后，立即恢复 StartTheWorld，用单独的 goroutine 执行清理操作。

因此在执行 CacheSpan 时，某些 span 可能还未完成清理。此时主动触发回收操作，有助于提高内存复用率，避免向操作系统过度申请内存。

malloc.h

```
sweep generation:
if sweepgen == h->sweepgen - 2, the span needs sweeping
if sweepgen == h->sweepgen - 1, the span is currently being swept
if sweepgen == h->sweepgen, the span is swept and ready to use
h->sweepgen is incremented by 2 after every GC
```

mcentral.c

```
MSpan* runtime·MCentral_CacheSpan(MCentral *c)
{
    // 当前垃圾回收代龄（随每次回收操作递增）。
    sg = runtime·mheap.sweepgen;
retry:
    // 尝试从 nonempty 链表中获取可用 span。
    for(s = c->nonempty.next; s != &c->nonempty; s = s->next) {
        // 如果 span 标记为等待回收，那么主动执行清理操作。
        if(s->sweepgen == sg-2 && runtime·cas(&s->sweepgen, sg-2, sg-1)) {
            // 将 span 移动到链表尾部。
            runtime·MSpanList_Remove(s);
            runtime·MSpanList_InsertBack(&c->empty, s);
```

```
        // 执行垃圾清理。
        runtime·MSpan_Sweep(s, true);
        goto havespan;
    }

    // 如果正在后台回收，则跳过。
    if(s->sweepgen == sg-1) {
        // the span is being swept by background sweeper, skip
        continue;
    }

    // 可用 span，将其转移到 empty 链表。
    runtime·MSpanList_Remove(s);
    runtime·MSpanList_InsertBack(&c->empty, s);
    goto havespan;
}

// 尝试从 emtpy 链表获取 span，目标是那些等待清理的 span。
for(s = c->empty.next; s != &c->empty; s = s->next) {
    // 如果是等待回收的 span，主动执行回收操作。
    if(s->sweepgen == sg-2 && runtime·cas(&s->sweepgen, sg-2, sg-1)) {
        // 将该 span 移到 empty 链表尾部。
        runtime·MSpanList_Remove(s);
        runtime·MSpanList_InsertBack(&c->empty, s);

        // 执行垃圾清理操作。
        runtime·MSpan_Sweep(s, true);

        // 如果回收后 freelist 链表不为空，表示有可用空间。
        if(s->freelist != nil)
            goto havespan;

        goto retry;
    }

    // 如果正在后台回收，跳过。
    if(s->sweepgen == sg-1) {
        continue;
    }

    // 处理过的 span，其代龄都已经标记为 sg，终止尝试。
    break;
}

// 如果 central 中没有找到可用 span，则向 heap 获取新的 span。
s = MCentral_Grow(c);
if(s == nil)
```

```
        return nil;

    // 将 span 插入到 empty 链表。
    runtime·MSpanList_InsertBack(&c->empty, s);

havespan:
    // 设置待返回 span 的相关属性。
    cap = (s->npages << PageShift) / s->elemsize;
    n = cap - s->ref;

    // 标记被 cache 使用。
    s->incache = true;

    return s;
}
```

相比 Go 1.3，cache 部分又做了很大的改进。代码更加简洁，流程也更加清晰。

而当 central 空间不足时，就需要从 heap 获取新 span 来完成扩张操作。这其中就包括对 span 所管理内存进行切分，形成 object freelist 链表。

mcentral.c

```
static MSpan* MCentral_Grow(MCentral *c)
{
    MLink **tailp, *v;
    byte *p;
    MSpan *s;

    // 计算所需 span 的大小信息。
    npages = runtime·class_to_allocnpages[c->sizeclass];
    size = runtime·class_to_size[c->sizeclass];
    n = (npages << PageShift) / size;

    // 从 heap 获取 span。
    s = runtime·MHeap_Alloc(&runtime·mheap, npages, c->sizeclass, 0, 1);
    if(s == nil)
        return nil;

    // 将 span 所管理的内存切分成 freelist/object 链表。
    tailp = &s->freelist;
    p = (byte*)(s->start << PageShift); // 起始地址。PageID(start) = p >> PageShift
    s->limit = p + size*n;
    for(i=0; i<n; i++) {
        v = (MLink*)p;
        *tailp = v;
        tailp = &v->next;
```

```
        p += size;
    }
    *tailp = nil;

    // 标记。
    runtime·markspan((byte*)(s->start<<PageShift), size, n, ...));

    return s;
}
```

前面在 mallocgc 中提及的大对象分配，也是用的 MHeap_Alloc 函数。

malloc.c

```
void runtime·largeAlloc_m(void)
{
    size = g->m->scalararg[0];
    npages = size >> PageShift;

    s = runtime·MHeap_Alloc(&runtime·mheap, npages, 0, 1, !(flag & FlagNoZero));
    g->m->ptrarg[0] = s;
}
```

mheap.c

```
MSpan* runtime·MHeap_Alloc(MHeap *h, uintptr npage, int32 sizeclass, bool large, ...)
{
    // 判断是否在 g0 栈执行。
    if(g == g->m->g0) {
        s = mheap_alloc(h, npage, sizeclass, large);
    } else {
        ...
    }

    return s;
}

static MSpan* mheap_alloc(MHeap *h, uintptr npage, int32 sizeclass, bool large)
{
    MSpan *s;

    // 如果垃圾回收操作未结束，那么尝试主动收回一些空间，以避免内存过度增长。
    // we need to sweep and reclaim at least n pages.
    if(!h->sweepdone)
        MHeap_Reclaim(h, npage);

    // 返回可用 span。
    s = MHeap_AllocSpanLocked(h, npage);
```

```
    if(s != nil) {
        // 标记代龄等状态。
        runtime·atomicstore(&s->sweepgen, h->sweepgen);
        s->state = MSpanInUse;
        s->freelist = nil;
        s->ref = 0;
        s->sizeclass = sizeclass;
        s->elemsize = (sizeclass==0 ?
            s->npages<<PageShift : runtime·class_to_size[sizeclass]);

        // 如果是大对象...
        if(large) {
            mstats.heap_objects++;
            mstats.heap_alloc += npage<<PageShift;

            // 根据页数，插入到合适的 busy 链表。
            if(s->npages < nelem(h->free))
                runtime·MSpanList_InsertBack(&h->busy[s->npages], s);
            else
                runtime·MSpanList_InsertBack(&h->busylarge, s);
        }
    }

    return s;
}
```

从 heap 获取 span 算法：

- 以页数为序号，从 free[] 提取链表，从中返回可用 span。
- 如链表为空，则继续从页数更大的链表中查找。
- 如 free[] 所有链表均没找到可用 span，就从 freelarge 链表中提取。
- 直到向操作系统申请新的 span 块。

mheap.c

```
static MSpan* MHeap_AllocSpanLocked(MHeap *h, uintptr npage)
{
    uintptr n;
    MSpan *s, *t;
    pageID p;

    // 以页数为序号，从 heap.free[] 中查找链表。
    // 如果当前链表没有可用 span，则从页数更大的链表中提取。
    for(n=npage; n < nelem(h->free); n++) {
        if(!runtime·MSpanList_IsEmpty(&h->free[n])) {
            s = h->free[n].next;
            goto HaveSpan;
```

```
        }
    }

    // 如果 free 所有链表都没找到合适的 span，则尝试更大的 large 链表。
    if((s = MHeap_AllocLarge(h, npage)) == nil) {
        // 还没找到，就只能新申请内存了。
        if(!MHeap_Grow(h, npage))
            return nil;

        // 重新查找合适的 span。
        // 每次向操作系统申请内存最少 1MB/128Pages，而 heap.free 最大下标 127，
        //    因此 FreeSpanLocked 函数会将其放到 freelarge 链表中。
        if((s = MHeap_AllocLarge(h, npage)) == nil)
            return nil;
    }

HaveSpan:
    // 将找到的 span 从 free 链表中移除。
    runtime·MSpanList_Remove(s);

    // 如果该 span 曾释放过物理内存，那么重新映射。
    if(s->npreleased > 0) {
        runtime·SysUsed((void*)(s->start<<PageShift), s->npages<<PageShift);
        mstats.heap_released -= s->npreleased<<PageShift;
        s->npreleased = 0;
    }

    // 如果返回的 span 页数多于需要 ...
    if(s->npages > npage) {
        // 新建一个 span 对象 t，用来管理尾部多余内存空间。
        t = runtime·FixAlloc_Alloc(&h->spanalloc);
        runtime·MSpan_Init(t, s->start + npage, s->npages - npage);

        // 调整实际所需的内存大小。
        s->npages = npage;
        p = t->start;
        p -= ((uintptr)h->arena_start>>PageShift);

        // 在 spans 区域标记 span 指针。
        if(p > 0)
            h->spans[p-1] = s;
        h->spans[p] = t;
        h->spans[p+t->npages-1] = t;

        // 将切出来的多余 span，重新放回 heap 管理链表中。
        MHeap_FreeSpanLocked(h, t, false, false);
        s->state = MSpanFree;
    }
```

```
    // 在 spans 中标记待所有页对应指针。
    p = s->start;
    p -= ((uintptr)h->arena_start>>PageShift);
    for(n=0; n<npage; n++)
        h->spans[p+n] = s;

    return s;
}
```

当找到的 span 大小超出预期时，分配器会执行切割操作，将多余的内存做成新 span 放回 heap 管理链表中。

从 large 里查找 span 的算法被称作 BestFit。很简单，通过循环遍历，找到大小最合适的目标。

mheap.c

```
MHeap_AllocLarge(MHeap *h, uintptr npage)
{
    return BestFit(&h->freelarge, npage, nil);
}

static MSpan* BestFit(MSpan *list, uintptr npage, MSpan *best)
{
    MSpan *s;

    for(s=list->next; s != list; s=s->next) {
        if(s->npages < npage)
            continue;
        if(best == nil
        || s->npages < best->npages
        || (s->npages == best->npages && s->start < best->start))
            best = s;
    }
    return best;
}
```

接着看看将 span 放回 heap 管理链表的 FreeSpanLocked 操作。

mheap.c

```
static void MHeap_FreeSpanLocked(MHeap *h, MSpan *s, bool acctinuse, bool acctidle)
{
    MSpan *t;
    pageID p;
```

```
// 修正状态标记。
s->state = MSpanFree;

// 从当前链表中移除。
runtime·MSpanList_Remove(s);

// 这两个参数会影响垃圾回收的物理内存释放操作。
s->unusedsince = runtime·nanotime();
s->npreleased = 0;

// 实际地址。
p = s->start;
p -= (uintptr)h->arena_start >> PageShift;

// 通过 heap.spans 检查左侧相邻 span。
// 如果左侧相邻 span 也是空闲状态，则合并。
if(p > 0 && (t = h->spans[p-1]) != nil && t->state != MSpanInUse &&
        t->state != MSpanStack) {
    // 修正属性。
    s->start = t->start;
    s->npages += t->npages;
    s->npreleased = t->npreleased; // absorb released pages
    s->needzero |= t->needzero;

    // 新起始地址。
    p -= t->npages;

    // 重新标记 spans。
    h->spans[p] = s;

    // 释放左侧 span 原对象。
    runtime·MSpanList_Remove(t);
    t->state = MSpanDead;
    runtime·FixAlloc_Free(&h->spanalloc, t);
}

// 尝试合并右侧 span。
if((p+s->npages)*sizeof(h->spans[0]) < h->spans_mapped &&
        (t = h->spans[p+s->npages]) != nil &&
        t->state != MSpanInUse && t->state != MSpanStack) {
    s->npages += t->npages;
    s->npreleased += t->npreleased;
    s->needzero |= t->needzero;
    h->spans[p + s->npages - 1] = s;
    runtime·MSpanList_Remove(t);
    t->state = MSpanDead;
    runtime·FixAlloc_Free(&h->spanalloc, t);
}
```

```
    // 根据 span 页数，插入到合适的链表中。
    if(s->npages < nelem(h->free))
        runtime·MSpanList_Insert(&h->free[s->npages], s);
    else
        runtime·MSpanList_Insert(&h->freelarge, s);
}
```

在此，我们看到了 heap.spans 的作用。合并零散内存块，以提供更大复用空间，这有助于减少内存碎片，是内存管理算法的一个重要设计目标。

最后，就是剩下如何向操作系统申请新的内存了。

malloc.h

```
HeapAllocChunk = 1<<20,          // Chunk size for heap growth
```

mheap.c

```
static bool MHeap_Grow(MHeap *h, uintptr npage)
{
    // 每次申请的内存总是 64KB 的倍数，最小 1MB。
    npage = ROUND(npage, (64<<10)/PageSize);
    ask = npage<<PageShift;
    if(ask < HeapAllocChunk)
        ask = HeapAllocChunk;

    // 申请内存。
    v = runtime·MHeap_SysAlloc(h, ask);

    // 创建新的 span 对象进行管理。
    s = runtime·FixAlloc_Alloc(&h->spanalloc);
    runtime·MSpan_Init(s, (uintptr)v>>PageShift, ask>>PageShift);
    p = s->start;
    p -= ((uintptr)h->arena_start>>PageShift);

    // 在 heap.spans 中标记地址。
    h->spans[p] = s;
    h->spans[p + s->npages - 1] = s;

    // 设置状态。
    runtime·atomicstore(&s->sweepgen, h->sweepgen);
    s->state = MSpanInUse;

    // 放回 heap 的管理链表，尝试执行合并操作。
    MHeap_FreeSpanLocked(h, s, false, true);
    return true;
}
```

申请时，需判断目标地址是否在 arena 范围内，且必须从 arena_used 开始。

malloc.c

```
void* runtime·MHeap_SysAlloc(MHeap *h, uintptr n)
{
    // 在 arena 范围内。
    if(n <= h->arena_end - h->arena_used) {
        // 使用 arena_used 地址。
        p = h->arena_used;
        runtime·SysMap(p, n, h->arena_reserved, &mstats.heap_sys);

        // 调整下一次分配位置。
        h->arena_used += n;

        // 同步增加 spans、bitmap 管理内存。
        runtime·MHeap_MapBits(h);
        runtime·MHeap_MapSpans(h);

        return p;
    }

    ...
}
```

mem_linux.c

```
void runtime·SysMap(void *v, uintptr n, bool reserved, uint64 *stat)
{
    p = runtime·mmap(v, n, PROT_READ|PROT_WRITE, MAP_ANON|MAP_FIXED|MAP_PRIVATE, -1, 0);
}
```

mem_darwin.c

```
void runtime·SysMap(void *v, uintptr n, bool reserved, uint64 *stat)
{
    p = runtime·mmap(v, n, PROT_READ|PROT_WRITE, MAP_ANON|MAP_FIXED|MAP_PRIVATE, -1, 0);
}
```

至此，对象内存分配和内存扩展的步骤结束。

1.3 释放流程

垃圾回收器通过调用 MSpan_Sweep 函数完成内存回收操作。

mgc0.c

```
bool runtime·MSpan_Sweep(MSpan *s, bool preserve)
{
    // 当前垃圾回收代龄。
    sweepgen = runtime·mheap.sweepgen;

    arena_start = runtime·mheap.arena_start;

    // 获取 span 相关信息。
    cl = s->sizeclass;
    size = s->elemsize;
    if(cl == 0) {
        // 大对象。
        n = 1;
    } else {
        // 小对象。
        npages = runtime·class_to_allocnpages[cl];
        n = (npages << PageShift) / size;
    }

    res = false;
    nfree = 0;
    end = &head;
    c = g->m->mcache;
    sweepgenset = false;

    // 标记 freelist 里的 object，这些对象未被使用，无需再次检查。
    for(link = s->freelist; link != nil; link = link->next) {
        off = (uintptr*)link - (uintptr*)arena_start;
        bitp = arena_start - off/wordsPerBitmapByte - 1;
        shift = (off % wordsPerBitmapByte) * gcBits;
        *bitp |= bitMarked<<shift;
    }

    // 释放 finalizer、profiler 关联对象。
    specialp = &s->specials;
    special = *specialp;
    while(special != nil) {
        // ...
    }

    // 计算标记位开始位置。
    p = (byte*)(s->start << PageShift);
    off = (uintptr*)p - (uintptr*)arena_start;
    bitp = arena_start - off/wordsPerBitmapByte - 1;
    shift = 0;
    step = size/(PtrSize*wordsPerBitmapByte);
```

```
bitp += step;
if(step == 0) {
    // 8-byte objects.
    bitp++;
    shift = gcBits;
}

// 遍历该 span 所有 object。
for(; n > 0; n--, p += size) {
    // 获取标记位。
    bitp -= step;
    if(step == 0) {
        if(shift != 0)
            bitp--;
        shift = gcBits - shift;
    }

    xbits = *bitp;
    bits = (xbits>>shift) & bitMask;

    // 如果 object 对象标记为可达 (Marked)，则跳过。
    // 包括 freelist 里的未使用对象。
    if((bits&bitMarked) != 0) {
        *bitp &= ~(bitMarked<<shift);
        continue;
    }

    // 重置标记位。
    *bitp = (xbits & ~((bitMarked|(BitsMask<<2))<<shift)) |
            ((uintptr)BitsDead<<(shift+2));

    if(cl == 0) { // 大对象。
        // 清除全部标记位。
        runtime·unmarkspan(p, s->npages<<PageShift);

        // 重置代龄。
        runtime·atomicstore(&s->sweepgen, sweepgen);
        sweepgenset = true;

        if(runtime·debug.efence) {
            // ...
        } else
            // 将大对象所使用的 span 归还给 heap。
            runtime·MHeap_Free(&runtime·mheap, s, 1);

        // 调整 next_gc 阈值。
        runtime·xadd64(&mstats.next_gc,
                       -(uint64)(size * (runtime·gcpercent + 100)/100));
```

```
            res = true;
        } else { // 小对象。
            // 将可回收对象添加到一个链表中。
            end->next = (MLink*)p;
            end = (MLink*)p;
            nfree++;
        }
    }

    // 如可回收小对象数量大于0。
    if(nfree > 0) {
        // 调整 next_gc 阈值。
        runtime·xadd64(&mstats.next_gc,
                       -(uint64)(nfree * size * (runtime·gcpercent + 100)/100));

        // 释放收集的 object 链表。
        res = runtime·MCentral_FreeSpan(&runtime·mheap.central[cl].mcentral, s, nfree,
                                        head.next, end, preserve);
    }

    return res;
}
```

该回收函数在分配流程 CacheSpan 中也曾提及过。

大对象释放很简单，调用 FreeSpanLocked 将 span 重新放回 heap 管理链表即可。

mheap.c

```
void runtime·MHeap_Free(MHeap *h, MSpan *s, int32 acct)
{
    mheap_free(h, s, acct);
}

static void mheap_free(MHeap *h, MSpan *s, int32 acct)
{
    MHeap_FreeSpanLocked(h, s, true, true);
}
```

至于收集的所有小对象，会被追加到 span.freelist 链表。如该 span 收回全部 object，则也将其归还给 heap。

mcentral.c

```
bool runtime·MCentral_FreeSpan(MCentral *c, MSpan *s, int32 n, MLink *start, ...)
{
```

```
    // span 不能是 cache 正在使用的对象。
    if(s->incache)
        runtime·throw("freespan into cached span");

    // 将收集的 object 链表追加到 span.freelist。
    wasempty = s->freelist == nil;
    end->next = s->freelist;
    s->freelist = start;
    s->ref -= n;

    // 将 span 转移到 central.nonempty 链表。
    if(wasempty) {
        runtime·MSpanList_Remove(s);
        runtime·MSpanList_Insert(&c->nonempty, s);
    }

    // 重置回收代龄。
    runtime·atomicstore(&s->sweepgen, runtime·mheap.sweepgen);

    if(s->ref != 0) {
        return false;
    }

    // 如果 span 收回全部 object (span.ref == 0)，从 central 管理链表移除。
    runtime·MSpanList_Remove(s);
    s->needzero = 1;
    s->freelist = nil;

    // 清除标记位。
    runtime·unmarkspan((byte*)(s->start<<PageShift), s->npages<<PageShift);

    // 将 span 交还给 heap。
    runtime·MHeap_Free(&runtime·mheap, s, 0);
    return true;
}
```

释放操作最终结果，仅仅是将可回收对象归还给 span.freelist 或 heap.free 链表，以便后续分配操作复用。至于物理内存释放，则由垃圾回收器的特殊定时操作完成。

1.4 其他

除了用户内存，分配器还需额外的 span、cache 等对象来维持系统运转。这些管理对象所需内存不从 arena 区域分配，不占用与 GC Heap 分配算法有关的内存地址。

系统为每种管理对象初始化一个固定分配器 FixAlloc。

malloc.h

```
struct FixAlloc
{
    uintptr size;                           // 固定分配长度。
    void    (*first)(void *arg, byte *p);   // 关联函数。
    void*   arg;                            // first 函数调用参数。
    MLink*  list;                           // 可复用空间链表。
    byte*   chunk;                          // 后备内存块当前分配指针。
    uint32  nchunk;                         // 后备内存块可用长度。
    uintptr inuse;                          // 后备内存块已使用长度。
};
```

mheap.c

```
void runtime·MHeap_Init(MHeap *h)
{
    runtime·FixAlloc_Init(&h->spanalloc, sizeof(MSpan), RecordSpan, ...);
    runtime·FixAlloc_Init(&h->cachealloc, sizeof(MCache), nil, ...);
    runtime·FixAlloc_Init(&h->specialfinalizeralloc, sizeof(SpecialFinalizer), ...);
    runtime·FixAlloc_Init(&h->specialprofilealloc, sizeof(SpecialProfile), ...);
}
```

FixAlloc 初始化过程很简单。

mfixalloc.c

```
void runtime·FixAlloc_Init(FixAlloc *f, uintptr size,
                           void (*first)(void*, byte*), void *arg, uint64 *stat)
{
    f->size = size;
    f->first = first;
    f->arg = arg;
    f->list = nil;
    f->chunk = nil;
    f->nchunk = 0;
    f->inuse = 0;
    f->stat = stat;
}
```

分配算法和 cache 类似。首先从复用链表提取，如果没找到，就从后备内存块截取。

malloc.h

```
FixAllocChunk = 16<<10,          // Chunk size for FixAlloc
```

mfixalloc.c

```
void* runtime·FixAlloc_Alloc(FixAlloc *f)
{
    void *v;

    // 如果空闲链表不为空，直接从链表提取。
    if(f->list) {
        v = f->list;
        f->list = *(void**)f->list;
        f->inuse += f->size;
        return v;
    }

    // 如果后备内存块空间不足...
    if(f->nchunk < f->size) {
        // 重新申请 16KB 后备内存。
        f->chunk = runtime·persistentalloc(FixAllocChunk, 0, f->stat);
        f->nchunk = FixAllocChunk;
    }

    // 从后备内存块截取。
    v = f->chunk;

    // 执行 first 函数。
    if(f->first)
        f->first(f->arg, v);

    // 调整剩余后备块参数。
    f->chunk += f->size;
    f->nchunk -= f->size;
    f->inuse += f->size;

    return v;
}
```

后备内存块策略有点类似 heap span，申请大块内存以减少系统调用开销。实际上，不同类别的 FixAlloc 会共享一个超大块内存，称之为 persistent。

malloc.go

```
var persistent struct {     // 全局变量，为全部 FixAlloc 提供后备内存块。
    lock mutex
    pos  unsafe.Pointer
    end  unsafe.Pointer
}

func persistentalloc(size, align uintptr, stat *uint64) unsafe.Pointer {
    const (
```

```
        chunk    = 256 << 10
        maxBlock = 64 << 10     // VM reservation granularity is 64K on windows
    )

    // 如果需要 64KB 以上，直接从 mmap 返回。
    if size >= maxBlock {
        return sysAlloc(size, stat)
    }

    // 对齐分配地址。
    persistent.pos = roundup(persistent.pos, align)

    // 如果剩余空间不足 ...
    if uintptr(persistent.pos)+size > uintptr(persistent.end) {
        // 重新从 mmap 申请 256KB 内存，保存到 persistent。
        persistent.pos = sysAlloc(chunk, &memstats.other_sys)
        persistent.end = add(persistent.pos, chunk)
    }

    // 截取内存，调整下次分配地址。
    p := persistent.pos
    persistent.pos = add(persistent.pos, size)

    return p
}
```

mem_linux.c

```
void* runtime·sysAlloc(uintptr n, uint64 *stat)
{
    p = runtime·mmap(nil, n, PROT_READ|PROT_WRITE, MAP_ANON|MAP_PRIVATE, -1, 0);
    return p;
}
```

释放操作仅仅是将对象收回到复用链表。

mfixalloc.c

```
void runtime·FixAlloc_Free(FixAlloc *f, void *p)
{
    f->inuse -= f->size;
    *(void**)p = f->list;
    f->list = p;
}
```

另外，在 FixAlloc 初始化时，还可额外提供一个 first 函数作为参数，比如 spanalloc 中的 RecordSpan。

该函数为 heap.allspans 分配内存，其内存储了所有 span 指针，GC Sweep 和 Heap Dump 操作都会用到这些信息。

mheap.c

```
static void RecordSpan(void *vh, byte *p)
{
    MHeap *h;
    MSpan *s;
    MSpan **all;
    uint32 cap;

    h = vh;
    s = (MSpan*)p;

    // 如果空间不足 ...
    if(h->nspan >= h->nspancap) {
        // 计算新容量。
        cap = 64*1024/sizeof(all[0]);
        if(cap < h->nspancap*3/2)
            cap = h->nspancap*3/2;

        // 分配新空间。
        all = (MSpan**)runtime·sysAlloc(cap*sizeof(all[0]), &mstats.other_sys);
        if(h->allspans) {
            // 将数据拷贝到新分配空间。
            runtime·memmove(all, h->allspans, h->nspancap*sizeof(all[0]));

            // 释放原内存。
            if(h->allspans != runtime·mheap.gcspans)
                runtime·SysFree(h->allspans, h->nspancap*sizeof(all[0]),
                                &mstats.other_sys);
        }

        // 指向新内存空间。
        h->allspans = all;
        h->nspancap = cap;
    }

    // 存储 span 指针。
    h->allspans[h->nspan++] = s;
}
```

2. Garbage Collector

精确垃圾回收，很经典的 Mark-and-Sweep 算法。

当分配 (malloc) 总量超出预设阈值，就会引发垃圾回收。操作前，须暂停用户逻辑执行 (StopTheWorld)，然后启用多个线程执行并行扫描工作，直到标记出所有可回收对象。

从 Go 1.3 开始，默认采用并发内存清理模式。也就是说，标记结束后，立即恢复逻辑执行 (StartTheWorld)。用一个专门的 goroutine 在后台清理内存。这缩短了暂停时间，在一定程度上改善了垃圾回收所引发的问题。

完成清理后，新阈值通常是存活对象所用内存的 2 倍。需要注意的是，清理操作只是调用内存分配器的相关方法，收回不可达对象内存进行复用，并未释放物理内存。

物理内存释放由专门线程定期执行。它检查最后一次垃圾回收时间，如超过 2 分钟，则执行强制回收。还会让操作系统收回闲置超过 5 分钟的 span 物理内存。

2.1 初始化

初始化函数创建并行标记状态对象 markfor，读取 GOGC 环境变量值。

proc.c

```
void runtime·schedinit(void)
{
    runtime·gcinit();
}
```

mgc0.c

```
void runtime·gcinit(void)
{
    runtime·work.markfor = runtime·parforalloc(MaxGcproc);
    runtime·gcpercent = runtime·readgogc();
}

int32 runtime·readgogc(void)
{
    byte *p;

    p = runtime·getenv("GOGC");
```

```
    // 默认值 100。
    if(p == nil || p[0] == '\0')
        return 100;

    // 关闭垃圾回收。
    if(runtime·strcmp(p, (byte*)"off") == 0)
        return -1;

    return runtime·atoi(p);
}
```

2.2 垃圾回收

在内存分配器中提到过，函数 mallocgc 会检查已分配内存是否超过阈值，并以此触发垃圾回收操作。

malloc.go

```
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Pointer {
    if memstats.heap_alloc >= memstats.next_gc {
        gogc(0)
    }
}
```

启动垃圾回收有三种不同方式。

- 0: 检查阈值来决定是否触发回收操作。
- 1: 强制回收。标记完成后，立即恢复用户逻辑执行，在后台并发执行清理操作。
- 2: 强制回收。在完成标记和清理操作前，不恢复用户逻辑执行。

malloc.go

```
func gogc(force int32) {
    // 如果 GOGC < 0，禁用垃圾回收，直接返回。
    if gp := getg(); gp == mp.g0 || mp.locks > 1 || !memstats.enablegc ||
                    panicking != 0 || gcpercent < 0 {
        return
    }

    semacquire(&worldsema, false)

    // 普通回收，会再次检查是否达到回收阈值。
    if force == 0 && memstats.heap_alloc < memstats.next_gc {
        semrelease(&worldsema)
        return
```

```
    }

    // 准备回收 ...
    startTime := nanotime()
    mp = acquirem()
    mp.gcing = 1

    // 停止用户逻辑执行。
    onM(stoptheworld)

    // 清理 sync.Pool 的相关缓存对象，这个后面有专门的剖析章节。
    clearpools()

    // 如果设置环境变量 GODEBUG=gctrace=2，那么会引发两次回收操作。
    n := 1
    if debug.gctrace > 1 {
        n = 2
    }
    for i := 0; i < n; i++ {
        if i > 0 {
            startTime = nanotime()
        }

        // 将 64-bit 开始时间保存到 scalararg 。
        mp.scalararg[0] = uintptr(uint32(startTime)) // low 32 bits
        mp.scalararg[1] = uintptr(startTime >> 32)   // high 32 bits

        // 清理行为标记。
        if force >= 2 {
            mp.scalararg[2] = 1 // eagersweep
        } else {
            mp.scalararg[2] = 0
        }

        // 在 g0 栈执行垃圾回收操作。
        onM(gc_m)
    }

    // 回收结束。
    mp.gcing = 0
    semrelease(&worldsema)

    // 恢复用户逻辑执行。
    onM(startheworld)
}
```

总体逻辑倒不复杂，StopTheWorld -> GC -> StartTheWorld。暂时抛开周边细节，看看垃圾回收流程。

mgc0.c

```
void runtime·gc_m(void)
{
    a.start_time = (uint64)(g->m->scalararg[0]) | ((uint64)(g->m->scalararg[1]) << 32);
    a.eagersweep = g->m->scalararg[2];
    gc(&a);
}

static void gc(struct gc_args *args)
{
    // 如果前次回收的清理操作未完成，那么先把这事结束了。
    while(runtime·sweepone() != -1)
        runtime·sweep.npausesweep++;

    // 为回收操作准备相关环境状态。
    runtime·mheap.gcspans = runtime·mheap.allspans;
    runtime·work.spans = runtime·mheap.allspans;
    runtime·work.nspan = runtime·mheap.nspan;

    runtime·work.nwait = 0;
    runtime·work.ndone = 0;
    runtime·work.nproc = runtime·gcprocs();

    // 初始化并行标记状态对象 markfor。
    //   使用 nproc 个线程执行并行标记任务。
    //   任务总数 = 固定内存段(RootCount) + 当前 goroutine G 的数量。
    //   标记函数 markroot。
    runtime·parforsetup(runtime·work.markfor, runtime·work.nproc,
                        RootCount + runtime·allglen, nil, false, markroot);

    if(runtime·work.nproc > 1) {
        // 重置结束标记。
        runtime·noteclear(&runtime·work.alldone);
        // 唤醒 nproc - 1 个线程准备执行 markroot 函数，因为当前线程也会参与标记工作。
        runtime·helpgc(runtime·work.nproc);
    }

    // 让当前线程也开始执行标记任务。
    gchelperstart();
    runtime·parfordo(runtime·work.markfor);
    scanblock(nil, 0, nil);

    if(runtime·work.nproc > 1)
        // 休眠，等待标记全部结束。
```

```
        runtime·notesleep(&runtime·work.alldone);

    // 收缩 stack 内存。
    runtime·shrinkfinish();

    // 更新所有 cache 统计参数。
    cachestats();

    // 计算上一次回收后 heap_alloc 大小。
    //   当前 next_gc = heap0 + heap0 * (gcpercent/100)
    //   那么 heap0 = next_gc / (1 + gcpercent/100)
    heap0 = mstats.next_gc*100/(runtime·gcpercent+100);

    // 计算下一次 next_gc 阈值。
    // 这个值只是预估，会随着清理操作而改变。
    mstats.next_gc = mstats.heap_alloc+mstats.heap_alloc*runtime·gcpercent/100;
    runtime·atomicstore64(&mstats.last_gc, runtime·unixnanotime());

    // 目标是 heap.allspans 里的所有 span 对象。
    runtime·mheap.gcspans = runtime·mheap.allspans;
    // GC 使用递增的代龄来表示 span 当前回收状态。
    runtime·mheap.sweepgen += 2;
    runtime·mheap.sweepdone = false;
    runtime·work.spans = runtime·mheap.allspans;
    runtime·work.nspan = runtime·mheap.nspan;
    runtime·sweep.spanidx = 0;

    if(ConcurrentSweep && !args->eagersweep) { // 并发清理
        // 新建或唤醒用于清理操作的 goroutine。
        if(runtime·sweep.g == nil)
            runtime·sweep.g = runtime·newproc1(&bgsweepv, nil, 0, 0, gc);
        else if(runtime·sweep.parked) {
            runtime·sweep.parked = false;
            runtime·ready(runtime·sweep.g); // 唤醒
        }
    } else { // 串行回收
        // 立即执行清理操作。
        while(runtime·sweepone() != -1)
            runtime·sweep.npausesweep++;
    }
}
```

算法的核心是并行回收和是否启用一个 goroutine 来执行清理操作。这个 goroutine 在清理操作结束后被冻结，再次使用前必须唤醒。

如果用专门的 goroutine 执行清理操作，那么 gc 函数不等清理操作结束就立即返回，上级的 gogc 会立即调用 StartTheWorld 恢复用户逻辑执行，这就是并发回收的关键。

我们回过头，看看一些中间环节的实现细节。

在设置并行回收状态对象 markfor 里提到过两个参数：任务总数和标记函数。

mgc0.c

```
enum {
    RootCount   = 5
}
```

任务总数其实是 5 个根内存段 RootData、RootBBS、RootFinalizers、RootSpans、RootFlushCaches，外加所有 goroutine stack 的总和。

mgc0.c

```
static void markroot(ParFor *desc, uint32 i)
{
    switch(i) {
    case RootData:
        ...
        break;

    case RootBss:
        ...
        break;

    case RootFinalizers:
        ...
        break;

    case RootSpans:
        ...
        break;

    case RootFlushCaches:
        flushallmcaches();                  // 清理 cache、stack。
        break;

    default:
        gp = runtime·allg[i - RootCount];
        runtime·shrinkstack(gp);            // 收缩 stack。
        scanstack(gp);
        ...
        break;
    }
}
```

核心算法 scanblock 函数通过扫描内存块，找出存活对象和可回收对象，并在 bitmap 区域进行标记。具体实现细节，本文不做详述，有兴趣可自行阅读源码或相关论文。

那么 parfor 是如何实现并行回收的呢?

这里面有个很大误导。其实 parfor 实现非常简单，仅是一个状态对象，核心是将要执行的多个任务序号平均分配个多个线程。

parfor.c

```
struct ParForThread
{
    // the thread's iteration space [32lsb, 32msb)
    uint64 pos;
};

void runtime·parforsetup(ParFor *desc, uint32 nthr, uint32 n, void *ctx, bool wait,
                         void (*body)(ParFor*, uint32))
{
    uint32 i, begin, end;
    uint64 *pos;

    desc->body = body;  // 任务函数
    desc->nthr = nthr;  // 并发线程数量
    desc->thrseq = 0;
    desc->cnt = n;      // 任务总数

    // 为线程平均分配任务编号。
    //   比如 10 个任务分配给 5 个线程，那么 thr[0] 就是 [0,2)，也就是 0 和 1 这两个任务。
    //   起始和结束编号分别保存在 ParForThread.pos 字段的高低位。
    for(i=0; i<nthr; i++) {
        begin = (uint64)n*i / nthr;
        end = (uint64)n*(i+1) / nthr;
        pos = &desc->thr[i].pos;
        *pos = (uint64)begin | (((uint64)end)<<32);
    }
}
```

现在任务被平均分配，并保存到全局变量 markfor 里。接下来的操作，其实是由被唤醒的线程主动完成，如同当前 GC 主线程主动调用 parfordo 一样。

执行标记任务的多个线程由 helpgc 函数唤醒，其中的关键就是设置 M.helpgc 标记。

proc.c

```
void runtime·helpgc(int32 nproc)
{
    pos = 0;

    // 从 1 开始，因为当前线程也会参与标记任务。
    for(n = 1; n < nproc; n++) {
        // 检查 P 是否被当前线程使用，如果是就跳过。
        if(runtime·allp[pos]->mcache == g->m->mcache)
            pos++;

        // 获取空闲线程。
        mp = mget();

        // 这是关键，线程唤醒后会检查该标记。
        mp->helpgc = n;

        // 为线程分配用户执行的 P.cache。
        mp->mcache = runtime·allp[pos]->mcache;
        pos++;

        // 唤醒线程。
        runtime·notewakeup(&mp->park);
    }
}
```

如果你熟悉线程 M 的工作方式，那么就会知道它通过 stopm 完成休眠操作。

proc.c

```
static void stopm(void)
{
    // 放回空闲队列。
    mput(g->m);
    // 休眠，直到被唤醒。
    runtime·notesleep(&g->m->park);

    // 被唤醒后，清除休眠标记。
    runtime·noteclear(&g->m->park);

    // 检查 helpgc 标记，执行 gchelper 函数。
    if(g->m->helpgc) {
        runtime·gchelper();
        g->m->helpgc = 0;
        g->m->mcache = nil;
        goto retry;
    }
}
```

mgc0.c

```
void runtime·gchelper(void)
{
    gchelperstart();
    runtime·parfordo(runtime·work.markfor);
    scanblock(nil, 0, nil);

    // 检查标记是否全部结束。
    nproc = runtime·work.nproc;
    if(runtime·xadd(&runtime·work.ndone, +1) == nproc-1)
        // 唤醒 GC 主线程。
        runtime·notewakeup(&runtime·work.alldone);

    g->m->traceback = 0;
}
```

最终和 GC 主线程调用过程一致。当 alldone 被唤醒后，GC 主线程恢复后续步骤执行。

至于被线程调用的 parfordo，其实也很简单。

parfor.c

```
void runtime·parfordo(ParFor *desc)
{
    // 每次调用，都会递增 thrseq 值。
    tid = runtime·xadd(&desc->thrseq, 1) - 1;

    // 如果任务线程数量为 1，那么没什么好说的，直接循环执行 body，也就是 markroot。
    if(desc->nthr==1) {
        for(i=0; i<desc->cnt; i++)
            desc->body(desc, i);
        return;
    }

    body = desc->body;

    // 用 tid 作为当前线程的编号，以此提取任务范围值。
    me = &desc->thr[tid];
    mypos = &me->pos;

    for(;;) {
        // 先完成自己的任务。
        for(;;) {
            // 递增当前任务范围的开始编号。
            pos = runtime·xadd64(mypos, 1);
```

```
        // 注意：只有低32位被修改，高32位结束编号不变。
        begin = (uint32)pos-1;
        end = (uint32)(pos>>32);

        // 如果小于结束编号，循环。
        if(begin < end) {
            // 执行 markroot 标记函数。
            body(desc, begin);
            continue;
        }
        break;
    }

    // 尝试从其他线程偷点任务过来，以便尽快完成所有标记操作。
    idle = false;
    for(try=0;; try++) {
        // 如果长时间没有偷到任务，设置结束标记。
        // increment the done counter...
        if(try > desc->nthr*4 && !idle) {
            idle = true;
            runtime·xadd(&desc->done, 1);
        }

        // 如果所有线程都结束，那么退出。
        if(desc->done + !idle == desc->nthr) {
            if(!idle)
                runtime·xadd(&desc->done, 1);
            goto exit;
        }

        // 随机选择一个线程任务。
        victim = runtime·fastrand1() % (desc->nthr-1);
        if(victim >= tid)
            victim++;
        victimpos = &desc->thr[victim].pos;

        for(;;) {
            // 偷取任务。
            pos = runtime·atomicload64(victimpos);
            begin = (uint32)pos;
            end = (uint32)(pos>>32);
            if(begin+1 >= end) {
                begin = end = 0;
                break;
            }
            if(idle) {
                runtime·xadd(&desc->done, -1);
                idle = false;
```

```
                }
                begin2 = begin + (end-begin)/2;
                newpos = (uint64)begin | (uint64)begin2<<32;
                if(runtime·cas64(victimpos, pos, newpos)) {
                    begin = begin2;
                    break;
                }
            }

            // 成功偷到任务...
            if(begin < end) {
                // 添加到自己的任务列表中。
                runtime·atomicstore64(mypos, (uint64)begin | (uint64)end<<32);
                // 返回外层循环，上面的任务处理代码再次被激活。
                break;
            }

            // ...
        }
    }
exit:
    // ...
}
```

每个线程调用 parfordo 的时候，都拿到一个递增的唯一 thrseq 编号，并以此获得事先由 parforsetup 分配好的任务段。接下来，自然是该线程循环执行分配给自己的所有任务，任务编号被传递给 markroot 作为选择目标的判断条件。

在完成自己的任务后，尝试分担其他线程任务，以尽快完成全部任务。这种 steal 算法，在运行时的很多地方都有体现，算是并行开发的一个 “标准” 做法了。

至此，并行标记的所有秘密被揭开，我们继续探究清理操作过程。

不管是并发还是串行清理，最终都是调用 sweepone 函数。

mgc0.c

```
static FuncVal bgsweepv = {runtime·bgsweep};
```

mgc0.go

```
func bgsweep() {
    for {
        for gosweepone() != ^uintptr(0) {
            sweep.nbgsweep++
            Gosched()
```

```
        }

        if !gosweepdone() {
            continue
        }

        // 设置休眠标志。
        sweep.parked = true

        // 休眠当前清理 goroutine。
        goparkunlock(&gclock, "GC sweep wait")
    }
}
```

mgc0.c

```
uintptr runtime·gosweepone(void)
{
    void (*fn)(void);

    fn = sweepone_m;
    runtime·onM(&fn);
    return g->m->scalararg[0];
}

static void sweepone_m(void)
{
    g->m->scalararg[0] = runtime·sweepone();
}
```

清理函数实现很简洁，每次找到一个待清理的 span，然后调用 span_sweep 收回对应的内存，这在内存分配器的释放过程中已经说得很清楚了。

mgc0.c

```
uintptr runtime·sweepone(void)
{
    // 当前代龄，清理前 += 2。
    sg = runtime·mheap.sweepgen;

    // 循环扫描所有 spans。
    for(;;) {
        idx = runtime·xadd(&runtime·sweep.spanidx, 1) - 1;

        // 结束判断。
        if(idx >= runtime·work.nspan) {
            runtime·mheap.sweepdone = true;
            return -1;
```

```
        }

        // 获取 span。
        s = runtime·work.spans[idx];

        // 如果不是正在使用的 span，无需清理。
        if(s->state != MSpanInUse) {
            s->sweepgen = sg;
            continue;
        }

        // 如果不是待清理 span，跳过。
        if(s->sweepgen != sg-2 || !runtime·cas(&s->sweepgen, sg-2, sg-1))
            continue;

        npages = s->npages;

        // 清理。
        if(!runtime·MSpan_Sweep(s, false))
            npages = 0;

        return npages;
    }
}
```

最后剩下的，就是 StopTheWorld 和 StartTheWorld 如何停止和恢复用户逻辑执行。因这会涉及一些 Goroutine Scheduler 知识，您可以暂时跳过，等看完后面的相关章节再回头研究。

proc.c

```
void runtime·stoptheworld(void)
{
    runtime·lock(&runtime·sched.lock);

    // 计数器。
    runtime·sched.stopwait = runtime·gomaxprocs;

    // 设置关键停止标记。
    runtime·atomicstore((uint32*)&runtime·sched.gcwaiting, 1);

    // 在所有运行的 goroutine 上设置抢占标志。
    preemptall();

    // 设置当前 P 的状态。
    g->m->p->status = Pgcstop; // Pgcstop is only diagnostic.
    runtime·sched.stopwait--;
```

```
    // 设置所有处理系统调用 P 的状态。
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        s = p->status;
        if(s == Psyscall && runtime·cas(&p->status, s, Pgcstop))
            runtime·sched.stopwait--;
    }

    // 设置所有空闲 P 状态。
    while(p = pidleget()) {
        p->status = Pgcstop;
        runtime·sched.stopwait--;
    }

    wait = runtime·sched.stopwait > 0;
    runtime·unlock(&runtime·sched.lock);

    // 等待所有 P 停止。
    if(wait) {
        for(;;) {
            // 等待 100us，直到休眠标记被唤醒。
            if(runtime·notetsleep(&runtime·sched.stopnote, 100*1000)) {
                // 清除休眠标记。
                runtime·noteclear(&runtime·sched.stopnote);
                break;
            }
            // 再次发出抢占标记。
            preemptall();
        }
    }
}
```

从代码上来看，StopTheWorld 只是设置了一些标记，包括抢占行为也不过是在在运行的 goroutine 上设置抢占标记。具体这些标记是如何让正在运行的 goroutine 暂停的呢?

如果了解 goroutine 运行机制，必然知道它总是循环执行 schedule 函数，在这个函数头部会检查 gcwaiting 标记，并以此停止当前任务执行。

proc.c

```
static void schedule(void)
{
    // 检查 gcwaiting 标记，停止当前任务执行。
    if(runtime·sched.gcwaiting) {
        gcstopm();
    }
```

```
    ...
}

static void gcstopm(void)
{
    // 释放关联的 P。
    p = releasep();
    runtime·lock(&runtime·sched.lock);
    p->status = Pgcstop;

    // 递减计数器，直到唤醒.
    if(--runtime·sched.stopwait == 0)
        runtime·notewakeup(&runtime·sched.stopnote);

    runtime·unlock(&runtime·sched.lock);
    stopm();
}
```

这样一来，所有正在执行的 goroutine 会被放回队列，相关任务线程也被休眠。至于发出抢占标记，是为了让一直处于忙碌状态的 goroutine 有机会检查停止标记。

反过来，StartTheWorld 就是恢复这些被停止的任务，并唤醒线程继续执行。

proc.c

```
void runtime·starttheworld(void)
{
    ...

    // 重置标记。
    runtime·sched.gcwaiting = 0;

    p1 = nil;

    // 循环所有 P。
    while(p = pidleget()) {
        // 如果该 P 没有任务，那么放回空闲队列。
        // 因为没有任务的 P 被放在列表尾部，故无需继续遍历。
        if(p->runqhead == p->runqtail) {
            pidleput(p);
            break;
        }

        // 关联一个空闲 M 线程。
        p->m = mget();

        // 将准备工作的 P 串成链表。
```

```
        p->link = p1;
        p1 = p;
    }

    // 唤醒 sysmon。
    if(runtime·sched.sysmonwait) {
        runtime·sched.sysmonwait = false;
        runtime·notewakeup(&runtime·sched.sysmonnote);
    }

    runtime·unlock(&runtime·sched.lock);

    // 遍历准备工作的 P。
    while(p1) {
        p = p1;
        p1 = p1->link;

        // 检查并唤醒关联线程 M。
        if(p->m) {
            mp = p->m;
            runtime·notewakeup(&mp->park);
        } else {
            // 如果没有关联线程，新建。
            newm(nil, p);
            add = false;
        }
    }

    // ...
}
```

垃圾回收操作虽然关联很多东西，但我们基本理清了它的运作流程。如同在分配器一章中所说，垃圾回收只是将回收内存，并没有释放空闲的物理内存。

2.3 内存释放

在 main goroutine 入口，运行时使用一个专用线程运行 sysmon 操作。

proc.go

```
// The main goroutine.
func main() {
    ...

    onM(newsysmon)
```

```
    ...

    main_init()
    main_main()
}
```

proc.c

```
void runtime·newsysmon(void)
{
    newm(sysmon, nil);
}
```

在 sysmon 里面会定时启动强制垃圾回收和物理内存释放操作。

proc.c

```
static void sysmon(void)
{
    // 如果超过 2 分钟没有运行 gc，则强制回收。
    forcegcperiod = 2*60*1e9;

    // 如果空闲 span 超过 5 分钟未被使用，则释放其关联物理内存。
    scavengelimit = 5*60*1e9;

    for(;;) {
        runtime·usleep(delay);

        // 启动强制垃圾回收。
        lastgc = runtime·atomicload64(&mstats.last_gc);
        if(lastgc != 0 && unixnow - lastgc > forcegcperiod && ...) {
            runtime·forcegc.idle = 0;
            runtime·forcegc.g->schedlink = nil;

            // 将强制垃圾回收 goroutine 放回任务队列。
            injectglist(runtime·forcegc.g);
        }

        // 启动物理内存释放操作。
        if(lastscavenge + scavengelimit/2 < now) {
            runtime·MHeap_Scavenge(nscavenge, now, scavengelimit);
            lastscavenge = now;
            nscavenge++; // 计数器。
        }
    }
}
```

先说强制垃圾回收操作，这个神秘的 forcegc.g 从何而来?

proc.go

```
// start forcegc helper goroutine
func init() {
    go forcegchelper()
}
```

依照 Go 语言规则，这个 init 初始化函数会被 main goroutine 执行，它创建了一个用来执行强制回收操作的 goroutine。

proc.go

```
func forcegchelper() {
    forcegc.g = getg()
    forcegc.g.issystem = true

    for {
        // 休眠该 goroutine。
        // park 会暂停 goroutine，但不会放回待运行队列。
        goparkunlock(&forcegc.lock, "force gc (idle)")

        // 唤醒后，执行强制垃圾回收。
        gogc(1)
    }
}
```

这个 forcegc.g 会循环执行，每次完成后休眠，直到被 sysmon 重新返回任务队列。

为什么要定期运行强制回收？试想一下，假设回收后已分配内存是 1GB，那么下次回收阈值就是 2GB，这可能导致很长时间无法触发回收操作。这就存在很大的内存浪费，所以强制回收是非常必要的。

接下来看看如何释放物理内存，这是另外一个关注焦点。

heap.c

```
void runtime·MHeap_Scavenge(int32 k, uint64 now, uint64 limit)
{
    h = &runtime·mheap;

    // 保存本次释放的物理内存数量。
    sumreleased = 0;

    // 循环处理 heap.free 里的空闲 span。
    for(i=0; i < nelem(h->free); i++)
```

```
        sumreleased += scavengelist(&h->free[i], now, limit);

    // 处理 heap.freelarge 里的空闲 span。
    sumreleased += scavengelist(&h->freelarge, now, limit);
}
```

释放操作的目标自然是 heap 里的那些空闲 span 内存块。

mheap.c

```
static uintptr scavengelist(MSpan *list, uint64 now, uint64 limit)
{
    sumreleased = 0;

    // 遍历 span 链表。
    for(s=list->next; s != list; s=s->next) {
        // 条件:
        //   未使用时间超过 5 分钟;
        //   已释放物理内存页数不等于 span 总页数 (未释放或部分释放);
        if((now - s->unusedsince) > limit && s->npreleased != s->npages) {
            // 待释放页数。为什么不是全部?
            released = (s->npages - s->npreleased) << PageShift;

            mstats.heap_released += released;
            sumreleased += released;

            // 现在整个 span.npages 都会被释放。
            s->npreleased = s->npages;

            runtime·SysUnused((void*)(s->start << PageShift), s->npages << PageShift);
        }
    }

    return sumreleased;
}
```

至于 npreleased != npages 的问题，先得看看 SysUnused 做了什么。

mem_linux.c

```
void runtime·SysUnused(void *v, uintptr n)
{
    runtime·madvise(v, n, MADV_DONTNEED);
}
```

mem_darwin.c

```
void runtime·SysUnused(void *v, uintptr n)
```

```
{
    // Linux's MADV_DONTNEED is like BSD's MADV_FREE.
    runtime·madvise(v, n, MADV_FREE);
}
```

对 Linux、darwin 等系统而言，MADV_DONTNEED、MADV_FREE 告诉操作系统，这段物理内存暂时不用，可解除 MMU 映射。再次使用时，由操作系统重新建立映射。

注意，尽管物理内存被释放了，但这个 span 管理对象依旧存活，它所占用的虚拟内存并未释放，依然会和左右相邻进行合并。这就是 nprereleased 可能不等于 npages 的关键。

另外，在 Windows 系统下，事情有点特殊，它不支持类似 MADV_DONTNEED 行为。

mem_windows.c

```
void runtime·SysUnused(void *v, uintptr n)
{
    r = runtime·stdcall3(runtime·VirtualFree, (uintptr)v, n, MEM_DECOMMIT);
}
```

显然，VirtualFree 会释放掉 span 管理的虚拟内存。因此，从 heap 获取 span 时需要重新分配内存。

mheap.c

```
static MSpan* MHeap_AllocSpanLocked(MHeap *h, uintptr npage)
{
    if(s->npreleased > 0) {
        runtime·SysUsed((void*)(s->start<<PageShift), s->npages<<PageShift);
        mstats.heap_released -= s->npreleased<<PageShift;
        s->npreleased = 0;
    }
}
```

mem_windows.c

```
void runtime·SysUsed(void *v, uintptr n)
{
    r = runtime·stdcall4(runtime·VirtualAlloc, (uintptr)v, n, MEM_COMMIT,
                         PAGE_READWRITE);
}
```

除了 Windows 系统，其他 Unix-Like 系统的 SysUsed 什么都不做。

mem_linux.c

```
void runtime·SysUsed(void *v, uintptr n)
{
    USED(v);
    USED(n);
}
```

mem_darwin.c

```
void runtime·SysUsed(void *v, uintptr n)
{
    USED(v);
    USED(n);
}
```

除自动回收外，还可手工调用 debug/FreeOSMemory 释放物理内存。

mgc0.go

```
func freeOSMemory() {
    gogc(2)            // force GC and do eager sweep
    onM(scavenge_m)
}
```

mheap.c

```
void runtime·scavenge_m(void)
{
    runtime·MHeap_Scavenge(-1, ~(uintptr)0, 0); // ~(uintptr)0 = 18446744073709551615
}
```

这个调用的参数，now 是比当前实际时间大得多的整数，而 limit 是 0。这意味这所有的空闲 span 都过期，都会被释放物理内存。

2.4 状态输出

与内存和垃圾回收相关的状态对象。

malloc.h

```
struct MStats
{
    // General statistics.
    uint64  alloc;            // 正在使用的 object 容量 (malloc)。
    uint64  total_alloc;      // 历史分配总量，含已释放内存。
    uint64  sys;              // 当前消耗的内存总量，包括 heap、fixalloc 等。
    uint64  nmalloc;          // 分配操作次数。
```

```
    uint64  nfree;            // 释放操作次数。

    // Statistics about malloc heap.
    uint64  heap_alloc;       // 同 alloc，在使用的 object 容量。
    uint64  heap_sys;         // 当前消耗的 heap 内存总量 (mmap-munmap, inuse+idle)。
    uint64  heap_idle;        // 空闲 span 容量。
    uint64  heap_inuse;       // 正在使用 span 容量。
    uint64  heap_released;    // 交还给操作系统的物理内存容量。
    uint64  heap_objects;     // 正在使用的 object 数量。

    // Statistics about garbage collector.
    uint64  next_gc;          // 下次垃圾回收阈值。
    uint64  last_gc;          // 上次垃圾回收结束时间。
    uint32  numgc;            // 垃圾回收次数。
};
```

统计状态更新函数。

mgc0.c

```
void runtime·updatememstats(GCStats *stats)
{
    // 重置状态对象。
    if(stats)
        runtime·memclr((byte*)stats, sizeof(*stats));

    for(mp=runtime·allm; mp; mp=mp->alllink) {
        if(stats) {
            src = (uint64*)&mp->gcstats;
            dst = (uint64*)stats;
            for(i=0; i<sizeof(*stats)/sizeof(uint64); i++)
                dst[i] += src[i];
            runtime·memclr((byte*)&mp->gcstats, sizeof(mp->gcstats));
        }
    }

    // FixAlloc 正在使用内存统计。
    mstats.mcache_inuse = runtime·mheap.cachealloc.inuse;
    mstats.mspan_inuse = runtime·mheap.spanalloc.inuse;

    // 从系统获取的内存总量 (mmap-munmap)。
    mstats.sys = mstats.heap_sys + mstats.stacks_sys + mstats.mspan_sys +
        mstats.mcache_sys + mstats.buckhash_sys + mstats.gc_sys + mstats.other_sys;

    mstats.alloc = 0;
    mstats.total_alloc = 0;
    mstats.nmalloc = 0;
    mstats.nfree = 0;
```

```
for(i = 0; i < nelem(mstats.by_size); i++) {
    mstats.by_size[i].nmalloc = 0;
    mstats.by_size[i].nfree = 0;
}

// 将所有 P.cache.alloc 所持有的 spans 归还给 central。
if(g == g->m->g0)
    flushallmcaches();
else {
    fn = flushallmcaches_m;
    runtime·mcall(&fn);
}

// 更新 cache 统计。
cachestats();

// 统计所有 spans 里正在使用的 object。
for(i = 0; i < runtime·mheap.nspan; i++) {
    s = runtime·mheap.allspans[i];
    if(s->state != MSpanInUse)
        continue;

    // 统计活跃的 object。
    if(s->sizeclass == 0) {
        mstats.nmalloc++;
        mstats.alloc += s->elemsize;
    } else {
        mstats.nmalloc += s->ref;
        mstats.by_size[s->sizeclass].nmalloc += s->ref;
        mstats.alloc += s->ref*s->elemsize;
    }
}

// 按 size class 统计累计分配和释放次数。
smallfree = 0;
mstats.nfree = runtime·mheap.nlargefree;
for(i = 0; i < nelem(mstats.by_size); i++) {
    mstats.nfree += runtime·mheap.nsmallfree[i];
    mstats.by_size[i].nfree = runtime·mheap.nsmallfree[i];
    mstats.by_size[i].nmalloc += runtime·mheap.nsmallfree[i];
    smallfree += runtime·mheap.nsmallfree[i] * runtime·class_to_size[i];
}
mstats.nfree += mstats.tinyallocs;
mstats.nmalloc += mstats.nfree;

// 总分配容量 = 正在使用 object + 已释放容量。
mstats.total_alloc = mstats.alloc + runtime·mheap.largefree + smallfree;
mstats.heap_alloc = mstats.alloc;
```

```
    mstats.heap_objects = mstats.nmalloc - mstats.nfree;
}
```

标准库 runtime.ReadMemStats 函数可刷新并读取该状态数据。

启用环境变量 GODEBUG="gotrace=1" 可输出垃圾回收相关状态信息，这有助于对程序运行状态进行监控，是常见的一种测试手段。

第一类输出信息来自垃圾回收函数。

mgc0.c

```
static void gc(struct gc_args *args)
{
    t0 = args->start_time;

    // 第 1 阶段：包括 stoptheworld、clearpools 在内的初始化时间。

    if(runtime·debug.gctrace)
        t1 = runtime·nanotime();

    // 第 2 阶段：标记前的准备时间。包括完成上次未结束的清理操作，准备并行标记环境等。

    if(runtime·debug.gctrace)
        t2 = runtime·nanotime();

    // 第 3 阶段：并行标记。

    if(runtime·debug.gctrace)
        t3 = runtime·nanotime();

    // 第 4 阶段：收缩栈内存，更新统计信息。

    t4 = runtime·nanotime();

    if(runtime·debug.gctrace) {
        heap1 = mstats.heap_alloc;
        runtime·updatememstats(&stats);
        obj = mstats.nmalloc - mstats.nfree;

        runtime·printf(
                "gc%d(%d):"             // 0, 1
                " %D+%D+%D+%D us,"      // 2, 3, 4, 5
                " %D -> %D MB,"         // 6, 7
                " %D (%D-%D) objects,"  // 8, 9, 10
                " %d goroutines,"       // 11
                " %d/%d/%d sweeps,"     // 12, 13, 14
```

```
                ...,
            mstats.numgc,                // 0: GC 执行次数。
            runtime·work.nproc,          // 1: 并行标记线程数量。
            (t1-t0)/1000,                // 2: 含 StopTheWorld 在内的初始化时间。
            (t2-t1)/1000,                // 3: 并行标记准备时间，包括上次未完成清理任务。
            (t3-t2)/1000,                // 4: 并行标记时间。
            (t4-t3)/1000,                // 5: 收缩栈内存，更新状态等时间。
            heap0>>20,                   // 6: 上次回收后 alloc 容量。
            heap1>>20,                   // 7: 本次回收后 alloc 容量。
            obj,                         // 8: 本次回收后正在使用的 object 数量。
            mstats.nmalloc,              // 9: 总分配次数。
            mstats.nfree,                // 10: 总释放次数。
            runtime·gcount(),            // 11: 待运行 Goroutine 任务数量。
            runtime·work.nspan,          // 12: heap.spans 数量。
            runtime·sweep.nbgsweep,      // 13: 本次并发清理 span 次数。
            runtime·sweep.npausesweep,   // 14: 本次串行清理 span 次数。
            ...
        );
    }
}
```

```
          +- 2: 含 stoptheworld 在内的初始化时间。
          |  +- 3: 并行标记准备时间，包括完成上次未清理任务。
          |  |   +- 4: 并行标记时间。
          |  |   |   +- 5: 收缩栈内存，更新状态等时间。
          |  |   |   |
          |  |   |   |                   +- 8: 本次回收后正在使用的 object 数量。
          |  |   |   |                   |    +- 9: 总分配次数。
          |  |   |   |                   |    |    +- 10: 总释放次数。                        +- 13: 并发清理次数。
          |  |   |   |                   |    |    |                                          |
gc78(2): 6+126+2+17 us, 8->24 MB, 37 (195-158) objects, 3 goroutines, 22/12/0 sweeps
   |  |                      |   |                                  |                 |      |
   |  |                      |   |                                  +- 11: 待运行数量。  |      +- 14: 串行清理。
   |  |                      |   |                                                     |
   |  |                      |   |                                                     +- 12: heap.span 数量。
   |  |                      |   |
   |  |                      |   |
   |  |                      |   +- 7: 本次回收后 alloc 容量。
   |  |                      +- 6: 上次回收后 alloc 容量。
   |  |
   |  +- 1: 并行标记线程数量。
   +- 0: 执行次数。
```

在并发清理模式下，信息输出时，清理工作尚未完成，因此标出的容量信息并不准确，只能通过多次输出结果进行大概评估。

第二类信息来自物理内存释放函数。

mheap.c

```
void runtime·MHeap_Scavenge(int32 k, uint64 now, uint64 limit)
{
    if(runtime·debug.gctrace > 0) {
        // 本次释放的物理内存容量。
        if(sumreleased > 0)
            runtime·printf("scvg%d: %D MB released\n", k, (uint64)sumreleased>>20);
```

```
        runtime·printf(
                "scvg%d: "                                      // 0
                "inuse: %D, "                                   // 1
                "idle: %D, "                                    // 2
                "sys: %D, "                                     // 3
                "released: %D, "                                // 4
                "consumed: %D (MB)\n",                          // 5

            k,                                                  // 0: 释放次数。
            mstats.heap_inuse>>20,                              // 1: 正在使用的 spans 容量。
            mstats.heap_idle>>20,                               // 2: 空闲 spans 容量。
            mstats.heap_sys>>20,                                // 3: 当前 heap 虚拟内存总容量。
            mstats.heap_released>>20,                           // 4: 已释放物理内存总容量。
            (mstats.heap_sys - mstats.heap_released)>>20  // 5: 实际消耗内存容量。
        );
    }
}
```

```
    +- 0: 执行次数.
    |                       +- 2: 所有空闲 span 内存容量。
    |                       |                           +- 4: 已释放物理内存总总量。
    |                       |                           |
scvg6: inuse: 32, idle: 8, sys: 41, released: 0, consumed: 41 (MB)
               |                     |                                |
               |                     |                                +- 5: 实际消耗内存容量。
               |                     +- 3: 已分配虚拟内存总量。
               |
               +- 1: 正在使用的 span 内存容量 (整个 span 容量，而不仅仅是正在使用的 object)。
```

现代操作系统通常会采用机会主义分配策略。内核虽然承诺分配内存，但实际并不会立即分配物理内存。只有在发生读写操作时，内核才会把之前承诺的内存转换为物理内存。而且也不是一次性完成，而是以页的方式逐步分配，按需执行页面请求调度和写入时复制。

所以，相关输出结果更多表示虚拟内存分配值，且和具体操作系统也有很大关系。

3. Goroutine Scheduler

调度器是运行时最核心的内容，其基本理论建立在三种基本对象上。

首先，每次 go 关键词调用都会创建一个 goroutine 对象，代表 G 并发任务。其次，所有 G 任务都由系统线程执行，这些线程被称作 M。

每个 G 对象都有自己的独立栈内存。当 M 执行任务时，从 G 用来保存执行现场的字段中恢复相关寄存器值即可。当 M 需要切换任务时，将寄存器值保存回当前 G 对象，然后从另一 G 对象中恢复，如此实现线程多路复用。

G 初始化栈内存只有几 KB 大小，按需扩张、收缩。这种轻量级设计开销极小，可轻松创建成千上万的并发任务。

除此之外，还有抽象处理器 P，其数量决定了 G 并发任务数量。每个运行 M 都必须获取并绑定一个 P 对象，如同线程必须被调度到某个 CPU Core 才能执行。P 还为 M 提供内存分配器缓存和 G 任务队列等执行资源。

通常情况下，P 数量在初始化时确定，运行时基本固定，但 M 的数量未必和 P 对应。例如，某 M 因系统调用长时间阻塞，其关联 P 就会被运行时收回。然后，调度器会唤醒或新建 M 去执行其他排队任务。失去 P 的 M 被休眠，直到被重新唤醒。

3.1 初始化

由汇编代码实现的 bootstrap 过程。

rt0_linux_amd64.s

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    LEAQ    8(SP), SI                   // argv
    MOVQ    0(SP), DI                   // argc
    MOVQ    $main(SB), AX
    JMP     AX

TEXT main(SB),NOSPLIT,$-8
    MOVQ    $runtime·rt0_go(SB), AX
    JMP     AX
```

要确定这个很简单，随便找个可执行文件，然后反汇编 entry point 即可。

```
(gdb) info files
Local exec file:
       Entry point: 0x437940

(gdb) disass 0x437940
Dump of assembler code for function _rt0_amd64_linux:
```

现在可以确定初始化调用由 rt0_go 汇编完成。

amd_asm64.s

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
    LEAQ    runtime·g0(SB), CX
    LEAQ    runtime·m0(SB), AX

    MOVQ    CX, m_g0(AX)                // save m->g0 = g0
    MOVQ    AX, g_m(CX)                 // save m0 to g0->m

    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)

    // create a new goroutine to start program
    MOVQ    $runtime·main·f(SB), BP     // entry
    PUSHQ   BP
    PUSHQ   $0                          // arg size
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // start this M
    CALL    runtime·mstart(SB)

    MOVL    $0xf1, 0xf1                 // crash
    RET
```

按图索骥，可以看到初始化过程相关的几个函数都做了什么。

runtime.h

```
MaxGomaxprocs = 1<<8,    // The max value of GOMAXPROCS.
```

proc.c

```
void runtime·schedinit(void)
{
    // 设置最大 M 线程数，超出会导致进程崩溃。
```

```
    runtime·sched.maxmcount = 10000;

    // 初始化内存分配器。
    runtime·mallocinit();

    // 获取命令行参数、环境变量。
    runtime·goargs();
    runtime·goenvs();

    // 垃圾回收器初始化。
    runtime·gcinit();

    // 初始化 P。
    procs = 1;
    p = runtime·getenv("GOMAXPROCS");
    if(p != nil && (n = runtime·atoi(p)) > 0) {
        if(n > MaxGomaxprocs)
            n = MaxGomaxprocs;
        procs = n;
    }
    procresize(procs);
}
```

其中内存分配器、垃圾回收器前面都已研究过，此处不多费唇舌。现在需要关心是 procs 这个最关键的 goroutine 并发控制参数。

proc.c

```
SchedT  runtime·sched;                    // 调度器实例。
int32   runtime·gomaxprocs;               // 当前 GOMAXPROCS 值。
P*      runtime·allp[MaxGomaxprocs+1];    // 存储所有的 P 对象，最多 256 个实例。
```

proc.c

```
static void procresize(int32 new)
{
    old = runtime·gomaxprocs;

    // 初始化新 P 对象。
    for(i = 0; i < new; i++) {
        p = runtime·allp[i];

        // 新建 P。
        if(p == nil) {
            p = runtime·newP();
            p->id = i;
            p->status = Pgcstop;
            runtime·atomicstorep(&runtime·allp[i], p);
```

```
    }

    // 创建 P.cache。
    if(p->mcache == nil) {
        if(old==0 && i==0)
            p->mcache = g->m->mcache;  // bootstrap
        else
            p->mcache = runtime·allocmcache();
    }
}

// 将 old P 里面的任务重新分布。
empty = false;
while(!empty) {
    empty = true;

    // 内层 for 循环遍历所有 old P，每次从中取一个 G 任务。
    // 外层 while 循环重复该过程，如此所有先生成的 G 会保存到全局队列的前面，FIFO。
    for(i = 0; i < old; i++) {
        p = runtime·allp[i];

        // 检查 P 的 G 任务队列。
        if(p->runqhead == p->runqtail)
            continue;

        empty = false;

        // 获取尾部最后一个 G。
        p->runqtail--;
        gp = p->runq[p->runqtail%nelem(p->runq)];

        // 将 G 添加到全局任务链表。
        gp->schedlink = runtime·sched.runqhead;
        runtime·sched.runqhead = gp;
        if(runtime·sched.runqtail == nil)
            runtime·sched.runqtail = gp;
        runtime·sched.runqsize++;
    }
}

// 将最多 new * (256/2) 个任务转移到 P 本地队列。
for(i = 1; i < new * nelem(p->runq)/2 && runtime·sched.runqsize > 0; i++) {
    gp = runtime·sched.runqhead;
    runtime·sched.runqhead = gp->schedlink;
    if(runtime·sched.runqhead == nil)
        runtime·sched.runqtail = nil;
    runtime·sched.runqsize--;
    runqput(runtime·allp[i%new], gp);
```

```
    }

    // 如果 new < old，"释放" 掉多余的 P 对象。
    for(i = new; i < old; i++) {
        p = runtime·allp[i];
        runtime·freemcache(p->mcache);
        p->mcache = nil;
        gfpurge(p);
        p->status = Pdead;
        // can't free P itself because it can be referenced by an M in syscall
    }

    // 关联 P 到当前 M。
    p = runtime·allp[0];
    acquirep(p);

    // 将其他 P 放到空闲队列。
    for(i = new-1; i > 0; i--) {
        p = runtime·allp[i];
        p->status = Pidle;
        pidleput(p);
    }

    runtime·atomicstore((uint32*)&runtime·gomaxprocs, new);
}
```

待运行的 G 任务保存在 P 本地队列和全局队列中，因此增加或减少 P 数量都需要重新分布这些任务。还须确保先生成的 G 任务优先放到队列头部，以优先执行。

在完成调度器初始化后，创建新 goroutine 运行 main 函数。

proc.go

```
// The main goroutine.
func main() {
    // 当前 G。
    g := getg()

    // 确定最大栈内存大小。
    if ptrSize == 8 {
        maxstacksize = 1000000000    // 1 GB
    } else {
        maxstacksize = 250000000     // 250 MB
    }

    // 使用单独线程运行 sysmon。
    onM(newsysmon)
```

```
    runtime_init()
    main_init()
    main_main()

    // 终止进程。
    exit(0)
}
```

3.2 创建任务

编译器会将每条 go func 语句编译成 newproc 函数调用，创建 G 对象。

反编译一个简单的示例。

test.go

```
package main

import ()

func main() {
    go println("Hello, World!")
}
```

```
(gdb) disass main.main
Dump of assembler code for function main.main:
   0x000000000000202f <+47>:    lea    rcx,[rip+0xff582]   # 0x1015b8 <main.print.1.f>
   0x0000000000002036 <+54>:    push   rcx
   0x0000000000002037 <+55>:    push   0x10
   0x0000000000002039 <+57>:    call   0x2e880 <runtime.newproc>
```

先熟悉 G 里面几个常见的字段成员。

runtime.h

```
struct  Stack
{
    uintptr lo;    // 栈内存开始地址。
    uintptr hi;    // 结束地址。
};

struct  Gobuf
{
    uintptr sp;    // 对应 SP 寄存器。
```

```
    uintptr pc;    // IP/PC 寄存器。
    void*   ctxt;
    uintreg ret;
    uintptr lr;    // ARM LR 寄存器。
};

struct  G
{
    Stack   stack;              // 自定义栈。
    uintptr stackguard0;        // 栈溢出检查边界。
    Gobuf   sched;              // 执行现场。
    G*      schedlink;          // 链表。
};
```

跟踪 newproc 的调用过程，最终目标是 newproc1。

proc.c

```
G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    siz = narg + nret;
    siz = (siz+7) & ~7; // 8 字节对齐

    // 当前 P。
    p = g->m->p;

    // 获取可复用的空闲 G 对象，或新建。
    if((newg = gfget(p)) == nil) {
        newg = runtime·malg(StackMin);
        runtime·casgstatus(newg, Gidle, Gdead);

        // 添加到 allg 全局变量。
        runtime·allgadd(newg);
    }

    // 将参数和返回值入栈。
    sp = (byte*)newg->stack.hi;
    sp -= 4*sizeof(uintreg);
    sp -= siz;
    runtime·memmove(sp, argp, narg);

    // thechar 5 代表 ARM，在 arch_xxx.h 中定义。
    // 因为 ARM 需要额外保存 Caller's LR 寄存器值。
    if(thechar == '5') {
        // caller's LR
        sp -= sizeof(void*);
        *(void**)sp = nil;
    }
```

```
    // 在 sched 里保存执行现场参数。
    runtime·memclr((byte*)&newg->sched, sizeof newg->sched);
    newg->sched.sp = (uintptr)sp;
    newg->sched.pc = (uintptr)runtime·goexit + PCQuantum;
    newg->sched.g = newg;

    // 这个调用很关键，不过我们在后面详说。
    runtime·gostartcallfn(&newg->sched, fn);

    newg->gopc = (uintptr)callerpc;
    runtime·casgstatus(newg, Gdead, Grunnable);

    // 将生成的 G 对象放到 P 本地队列或全局队列。
    runqput(p, newg);

    // 如果有空闲 P，且没有处于自旋状态的 M ...
    if(runtime·atomicload(&runtime·sched.npidle) != 0 &&
       runtime·atomicload(&runtime·sched.nmspinning) == 0 &&
       fn->fn != runtime·main)
        // 唤醒一个休眠的 M，或新建。
        wakep();

    return newg;
}
```

提取可复用 G 对象，将参数、返回值入栈，设置执行现场的寄存器值。最后，放到待运行队列等待被 M 执行。

P 使用 gfree 链表存储可复用 G 对象，这很好理解。除本地复用链表外，还有一个全局复用链表。当某 P 本地链表过长时，就转移一部分到全局链表，以供其他 P 使用。

runtime.h

```
struct  SchedT
{
    // Global cache of dead G's. (任务结束，复用对象)
    G*      gfree;
    int32   ngfree;
};

struct P
{
    // Available G's (status == Gdead)
    G*      gfree;
    int32   gfreecnt;
};
```

proc.c

```
static G* gfget(P *p)
{
    G *gp;
    void (*fn)(G*);

retry:
    // 从 P 本地链表获取一个可复用 G 对象。
    gp = p->gfree;

    // 如果为空，转向全局链表。
    if(gp == nil && runtime·sched.gfree) {
        // 从全局链表提取一些复用对象到本地，直到填满 32 个。
        while(p->gfreecnt < 32 && runtime·sched.gfree != nil) {
            p->gfreecnt++;
            gp = runtime·sched.gfree;
            runtime·sched.gfree = gp->schedlink;
            runtime·sched.ngfree--;
            gp->schedlink = p->gfree;
            p->gfree = gp;
        }

        // 填充后再从本地链表获取。
        goto retry;
    }

    // 如果找到可复用 G 对象。
    if(gp) {
        // 调整本地链表。
        p->gfree = gp->schedlink;
        p->gfreecnt--;

        // 检查自定义栈。
        if(gp->stack.lo == 0) {
            // 重新分配栈内存。
            if(g == g->m->g0) {
                gp->stack = runtime·stackalloc(FixedStack);
            } else {
                g->m->scalararg[0] = FixedStack;
                g->m->ptrarg[0] = gp;
                fn = mstackalloc;
                runtime·mcall(&fn);
                g->m->ptrarg[0] = nil;
            }

            // 设置栈顶。
            gp->stackguard0 = gp->stack.lo + StackGuard;
```

```
        }
    }

    return gp;
}
```

暂时不去理会自定义栈，后面有专门的章节说明这个问题。

没有可复用对象时，新建。

proc.c

```
G* runtime·malg(int32 stacksize)
{
    G *newg;
    void (*fn)(G*);

    // 新建 G 对象。
    newg = allocg();

    // 分配自定义栈内存。
    if(stacksize >= 0) {
        stacksize = runtime·round2(StackSystem + stacksize);
        if(g == g->m->g0) {
            newg->stack = runtime·stackalloc(stacksize);
        } else {
            g->m->scalararg[0] = stacksize;
            g->m->ptrarg[0] = newg;
            fn = mstackalloc;
            runtime·mcall(&fn);
            g->m->ptrarg[0] = nil;
        }
        newg->stackguard0 = newg->stack.lo + StackGuard;
        newg->stackguard1 = ~(uintptr)0;
    }
    return newg;
}

static G* allocg(void)
{
    return runtime·newG();
}
```

proc.go

```
func newG() *g {
    return new(g)
}
```

新建 G 对象被添加到全局变量 allg。

proc.c

```
Slice   runtime·allgs;       // Go Slice。
G**     runtime·allg;        // 当前所有 G 对象，包括完成任务，等待复用的。
uintptr runtime·allglen;     // 数量。
```

proc.go

```
func allgadd(gp *g) {
    allgs = append(allgs, gp)
    allg = &allgs[0]
    allglen = uintptr(len(allgs))
}
```

所有参数设置好后，G 对象所代表的并发任务被放入待运行队列。

runtime.h

```
struct  SchedT
{
    // Global runnable queue. (待运行任务)
    G*      runqhead;
    G*      runqtail;
    int32   runqsize;
};

struct P
{
    // Queue of runnable goroutines. (用数组实现的环状队列)
    uint32  runqhead;
    uint32  runqtail;
    G*      runq[256];
};
```

proc.c

```
static void runqput(P *p, G *gp)
{
    uint32 h, t;

retry:
    // 很典型的数组环状队列实现。
    // 累加 head、tail 位置计数器，然后取模获取实际存储索引。
    h = runtime·atomicload(&p->runqhead);
    t = p->runqtail;
    if(t - h < nelem(p->runq)) {
```

```
        p->runq[t%nelem(p->runq)] = gp;
        runtime·atomicstore(&p->runqtail, t+1);
        return;
    }

    // 如果本地队列已满，则放入全局待运行队列。
    if(runqputslow(p, gp, h, t))
        return;

    goto retry;
}

static bool runqputslow(P *p, G *gp, uint32 h, uint32 t)
{
    // 从本地队列提取一半待运行 G 任务。
    n = t-h;
    n = n/2;
    for(i=0; i<n; i++)
        batch[i] = p->runq[(h+i)%nelem(p->runq)];

    // 调整本地队列位置。
    if(!runtime·cas(&p->runqhead, h, h+n))
        return false;

    // 添加当前 G。
    batch[n] = gp;

    // 链表结构。
    for(i=0; i<n; i++)
        batch[i]->schedlink = batch[i+1];

    // 将这一批 G 放到全局队列。
    globrunqputbatch(batch[0], batch[n], n+1);
    return true;
}

static void globrunqputbatch(G *ghead, G *gtail, int32 n)
{
    // 直接将链表附加到全局链表尾部。
    gtail->schedlink = nil;
    if(runtime·sched.runqtail)
        runtime·sched.runqtail->schedlink = ghead;
    else
        runtime·sched.runqhead = ghead;
    runtime·sched.runqtail = gtail;
    runtime·sched.runqsize += n;
}
```

两个队列采用了不同的设计。本地队列长度固定，用数组自然是效率最高。而全局队列长度未知，只能用链表实现。

调度器在很多地方都采用两级队列设计，本地队列是为了当前线程无锁获取资源，而全局队列则是为了在多个 P/M 间进行平衡。当 P 管理的对象数量过多时就会上交一部分到全局，反过来，就从全局提取一批到本地。总之，最终目的是为了更好地复用内存，更快地完成任务执行。

3.3 任务线程

不管语言层面如何抽象，所有 G 任务总归要由线程执行，每个系统线程对应一个 M。

runtime.h

```
struct  M
{
    G*      g0;                     // 运行时管理栈。
    void    (*mstartfn)(void);      // 启动函数，比如执行 sysmon。

    G*      curg;                   // 当前运行的 G。
    P*      p;                      // 当前关联的 P。
    P*      nextp;                  // 临时存放获取的 P，用于后续任务。

    Note    park;                   // 休眠标记。

    M*      alllink;                // 全局 allm 链表。
};
```

先了解 M 的创建过程。

proc.c

```
static void newm(void(*fn)(void), P *p)
{
    // 创建 M 对象。
    mp = runtime·allocm(p);

    // 设置待绑定 P 和启动函数。
    mp->nextp = p;
    mp->mstartfn = fn;

    // 创建系统线程。
    runtime·newosproc(mp, (byte*)mp->g0->stack.hi);
}
```

```
M* runtime·allocm(P *p)
{
    mp = runtime·newM();

    // 初始化。
    mcommoninit(mp);

    // 创建一个 G，用于初始化 g0 栈。
    if(runtime·iscgo || Solaris || Windows || Plan9)
        mp->g0 = runtime·malg(-1);
    else
        mp->g0 = runtime·malg(8192);
    mp->g0->m = mp;

    return mp;
}
```

调度器会检查 M 总数，如超出限制会导致进程崩溃。默认 10000，多数时候无需关心，也可调用 debug/SetMaxThreads 修改。

proc.c

```
static void mcommoninit(M *mp)
{
    // 增加计数器，设置 ID。
    mp->id = runtime·sched.mcount++;

    // 检查系统当前 M 总数，如果超出限制，引发进程崩溃。
    checkmcount();

    // 添加到全局链表。
    mp->alllink = runtime·allm;
    runtime·atomicstorep(&runtime·allm, mp);
}

static void checkmcount(void)
{
    if(runtime·sched.mcount > runtime·sched.maxmcount){
        runtime·printf("runtime: program exceeds %d-thread limit\n",
                       runtime·sched.maxmcount);
        runtime·throw("thread exhaustion");
    }
}
```

最关键的是 newosproc 创建系统线程。

os_linux.c

```
void runtime·newosproc(M *mp, void *stk)
{
    flags = CLONE_VM    /* share memory */
        | CLONE_FS      /* share cwd, etc */
        | CLONE_FILES   /* share fd table */
        | CLONE_SIGHAND /* share sig handler table */
        | CLONE_THREAD; /* revisit - okay for now */

    ret = runtime·clone(flags, stk, mp, mp->g0, runtime·mstart);
}
```

os_darwin.c

```
void runtime·newosproc(M *mp, void *stk)
{
    errno = runtime·bsdthread_create(stk, mp, mp->g0, runtime·mstart);
}
```

我们看到了线程函数 mstart，这是后面要跟踪的目标。

M 有个很神秘的 g0 成员，它被传递给 newosproc 作为线程栈内存，用来执行运行时管理指令，以避免在 G 用户栈上切换上下文。

假如 M 线程直接使用 G 栈，那么就不能在执行管理操作时将它放回队列，也不能转交给其他 M 执行，那会导致多个线程共用栈内存。同样不能执行用户栈的扩张或收缩操作。因此，在执行管理指令前，必须将线程栈切换到 g0。

在前面章节中时常出现的 onM、mcall 就是用 g0 来执行管理命令。

runtime.h

```
struct  M
{
    uintptr scalararg[4];        // scalar argument/return for mcall
    void*   ptrarg[4];           // pointer argument/return for mcall
};
```

asm_amd64.s

```
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI                    // DI 保存要运行的管理函数指针。

    // 保存当前 G 执行现场。
    get_tls(CX)
    MOVQ    g(CX), AX                       // save state in g->sched
```

```
    MOVQ    0(SP), BX                       // caller's PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX)
    LEAQ    fn+0(FP), BX                    // caller's SP
    MOVQ    BX, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)

    // 切换到 g0 栈，执行管理函数。
    MOVQ    g(CX), BX                       // g
    MOVQ    g_m(BX), BX                     // g.m
    MOVQ    m_g0(BX), SI                    // m.g0
    MOVQ    SI, g(CX)                       // g = m->g0
    MOVQ    (g_sched+gobuf_sp)(SI), SP      // sp = m->g0->sched.sp
    PUSHQ   AX
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    CALL    DI                              // fn arg
    RET
```

在创建 G 时，调度器会调用 wakep 唤醒 M 执行任务。

proc.c

```
static void wakep(void)
{
    startm(nil, true);
}

static void startm(P *p, bool spinning)
{
    M *mp;
    void (*fn)(void);

    // 获取空闲 P。如果没有，直接返回。
    if(p == nil) {
        p = pidleget();
        if(p == nil) {
            return;
        }
    }

    // 获取空闲 M， 或新建。
    mp = mget();
    if(mp == nil) {
        fn = nil;
        newm(fn, p);
        return;
    }
```

```
    // 临时保存待用 P。
    mp->nextp = p;

    // 唤醒。
    runtime·notewakeup(&mp->park);
}

static M* mget(void)
{
    // 从空闲列表获取 M。
    if((mp = runtime·sched.midle) != nil){
        runtime·sched.midle = mp->schedlink;
        runtime·sched.nmidle--;
    }
    return mp;
}
```

当 M 线程找不到后续待运行 G 任务，或因某种原因被剥夺关联 P 时，会休眠线程，并被保存到 sched.midle 空闲链表中，直到被重新获取、唤醒。

proc.c

```
static void stopm(void)
{
    ...
retry:
    // 添加到空闲链表。
    mput(g->m);

    // 休眠线程，直到被唤醒后继续执行。
    runtime·notesleep(&g->m->park);

    // 被唤醒后，清除休眠标志。
    runtime·noteclear(&g->m->park);

    // 处理 GC 任务 (这个因为 StopTheWorld，并不需要 P)。
    if(g->m->helpgc) {
        runtime·gchelper();
        g->m->helpgc = 0;
        g->m->mcache = nil;
        goto retry;
    }

    // 既然被唤醒，必然获取了可用 P，关联。
    acquirep(g->m->nextp);
    g->m->nextp = nil;
}
```

```
static void mput(M *mp)
{
    // 添加到空闲链表。
    mp->schedlink = runtime·sched.midle;
    runtime·sched.midle = mp;
    runtime·sched.nmidle++;
}

static void acquirep(P *p)
{
    g->m->mcache = p->mcache;
    g->m->p = p;
    p->m = g->m;
    p->status = Prunning;
}
```

休眠操作通过 futex 实现，这是一种快速用户区互斥实现。该锁定在用户空间用原子指令完成，只在结果不一致时才进入系统内核，有非常高的执行效率。

lock_futex.go

```
func notesleep(n *note) {
    for atomicload(key32(&n.key)) == 0 {
        futexsleep(key32(&n.key), 0, -1)   // 休眠直到被唤醒 (timeout = -1)。
    }                                      // 唤醒后，n.key = 1，终止循环。
}
```

os_linux.c

```
void runtime·futexsleep(uint32 *addr, uint32 val, int64 ns)
{
    Timespec ts;

    // 不超时。
    if(ns < 0) {
        runtime·futex(addr, FUTEX_WAIT, val, nil, nil, 0);
        return;
    }

    ts.tv_nsec = 0;
    ts.tv_sec = runtime·timediv(ns, 1000000000LL, (int32*)&ts.tv_nsec);
    runtime·futex(addr, FUTEX_WAIT, val, &ts, nil, 0);
}
```

唤醒操作会修改标记值，成功后调用 noteclear 重置状态。

lock_futex.go

```
func notewakeup(n *note) {
    old := xchg(key32(&n.key), 1)
    futexwakeup(key32(&n.key), 1)
}

func noteclear(n *note) {
    n.key = 0
}
```

os_linux.c

```
void runtime·futexwakeup(uint32 *addr, uint32 cnt)
{
    ret = runtime·futex(addr, FUTEX_WAKE, cnt, nil, nil, 0);
}
```

3.4 任务执行

线程函数 mstart 让 M 进入调度器核心循环，它不停从 P 本地队列、全局队列查找并执行待运行 G 任务。期间，会处理一下垃圾回收等额外操作，完成后继续回来执行任务。

proc.c

```
static void mstart(void)
{
    // 执行启动函数。
    if(g->m->mstartfn)
        g->m->mstartfn();

    if(g->m->helpgc) {
        // 如果正在垃圾回收，休眠线程。
        g->m->helpgc = 0;
        stopm();
    } else if(g->m != &runtime·m0) {
        // 关联 P。
        acquirep(g->m->nextp);
        g->m->nextp = nil;
    }

    // 执行调度函数。
    schedule();
}
```

核心循环过程: schedule -> execute -> G.func -> goexit 。

proc.c

```
static void schedule(void)
{
    gp = nil;

    // 当前 P 任务执行次数计数器。
    tick = g->m->p->schedtick;

    // 每隔 61 次，就从全局队列提取一个任务，以确保公平。
    // This is a fancy way to say tick%61==0,
    if(tick - (((uint64)tick*0x4325c53fu)>>36)*61 == 0 && runtime·sched.runqsize > 0) {
        gp = globrunqget(g->m->p, 1);  // 仅返回一个 G，不转移。
    }

    // 从本地队列提取任务。
    if(gp == nil) {
        gp = runqget(g->m->p);
    }

    // 从其他地方查找任务。
    if(gp == nil) {
        gp = findrunnable();  // blocks until work is available
    }

    // 执行任务。
    execute(gp);
}
```

全局队列存储了超出 P 本地数量限制的待运行任务，是所有 P/M 的后备资源。

proc.c

```
static G* globrunqget(P *p, int32 max)
{
    G *gp, *gp1;
    int32 n;

    if(runtime·sched.runqsize == 0)
        return nil;

    // 确定要转移的任务数。
    n = runtime·sched.runqsize/runtime·gomaxprocs+1;
    if(n > runtime·sched.runqsize)
        n = runtime·sched.runqsize;
    if(max > 0 && n > max)
        n = max;
    if(n > nelem(p->runq)/2)
```

```
        n = nelem(p->runq)/2;

    runtime·sched.runqsize -= n;
    if(runtime·sched.runqsize == 0)
        runtime·sched.runqtail = nil;

    // 将第一个任务返回。
    gp = runtime·sched.runqhead;
    runtime·sched.runqhead = gp->schedlink;
    n--;

    // 转移一批任务到本地队列。
    while(n--) {
        gp1 = runtime·sched.runqhead;
        runtime·sched.runqhead = gp1->schedlink;
        runqput(p, gp1);
    }

    return gp;
}
```

本地队列优先为线程提供无锁任务获取。

proc.c

```
static G* runqget(P *p)
{
    G *gp;
    uint32 t, h;

    // 从数组循环队列返回任务。
    for(;;) {
        h = runtime·atomicload(&p->runqhead);
        t = p->runqtail;
        if(t == h)
            return nil;
        gp = p->runq[h%nelem(p->runq)];
        if(runtime·cas(&p->runqhead, h, h+1))
            return gp;
    }
}
```

如果本地和全局队列中都没找到可用任务，调度器就会费尽心思检查各个角落。包括网络任务，甚至是从其他 P 队列中偷一些过来。

proc.c

```
static G* findrunnable(void)
```

```
{
top:
    // 本地队列。
    gp = runqget(g->m->p);
    if(gp)
        return gp;

    // 全局队列。
    if(runtime·sched.runqsize) {
        gp = globrunqget(g->m->p, 0); // 转移一批到本地。
        if(gp)
            return gp;
    }

    // 网络任务。
    gp = runtime·netpoll(false);  // non-blocking
    if(gp) {
        injectglist(gp->schedlink);  // 插入全局队列。
        runtime·casgstatus(gp, Gwaiting, Grunnable);
        return gp;
    }

    // 从其他 P 偷一些任务。
    for(i = 0; i < 2*runtime·gomaxprocs; i++) {
        p = runtime·allp[runtime·fastrand1()%runtime·gomaxprocs]; // 随机选择。
        if(p == g->m->p)      // 当前 P。
            gp = runqget(p);
        else                  // 其他 P。
            gp = runqsteal(g->m->p, p);
        if(gp)
            return gp;
    }
stop:
    // 再次检查全局队列。
    if(runtime·sched.runqsize) {
        gp = globrunqget(g->m->p, 0);
        return gp;
    }

    // 解除 P 绑定，返回空闲列表。
    p = releasep();
    pidleput(p);

    // 循环检查其他 P 任务列表，如果有未完成任务，那么跳转到 top 重试，以便偷一些过来。
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p && p->runqhead != p->runqtail) {
            // 重新关联 P。
```

```
            p = pidleget();
            if(p) {
                acquirep(p);
                goto top;
            }
            break;
        }
    }

    // 再次检查网络任务。
    if(runtime·xchg64(&runtime·sched.lastpoll, 0) != 0) {
        gp = runtime·netpoll(true);  // block until new work is available
        runtime·atomicstore64(&runtime·sched.lastpoll, runtime·nanotime());
        if(gp) {
            p = pidleget();
            if(p) {
                // 重新关联 P。
                acquirep(p);
                // 将其他任务添加到全局队列。
                injectglist(gp->schedlink);

                runtime·casgstatus(gp, Gwaiting, Grunnable);
                return gp;
            }

            // 如果没有可用 P，添加到全局队列。
            injectglist(gp);
        }
    }

    // 如果什么任务都没拿到，休眠当前线程。
    stopm();
    goto top;
}
```

这种偷窃行为就是官方文档所提及的 work-stealing 算法。

proc.c

```
static G* runqsteal(P *p, P *p2)
{
    G *gp;
    G *batch[nelem(p->runq)/2];

    // 将 P2 一半任务转移到 batch。
    n = runqgrab(p2, batch);
    if(n == 0)
        return nil;
```

```
    // 返回一个任务。
    n--;
    gp = batch[n];
    if(n == 0)
        return gp;

    // 将剩余任务添加到 P 队列。
    h = runtime·atomicload(&p->runqhead);
    t = p->runqtail;
    for(i=0; i<n; i++, t++)
        p->runq[t%nelem(p->runq)] = batch[i];
    runtime·atomicstore(&p->runqtail, t);

    return gp;
}
```

等拿到 G 任务，接下来就交由 execute 负责执行。

proc.c

```
static void execute(G *gp)
{
    // 修改状态。
    runtime·casgstatus(gp, Grunnable, Grunning);
    gp->waitsince = 0;
    gp->preempt = false;
    gp->stackguard0 = gp->stack.lo + StackGuard;
    g->m->p->schedtick++;  // 执行计数器。
    g->m->curg = gp;
    gp->m = g->m;

    runtime·gogo(&gp->sched);
}
```

asm_amd64.s

```
TEXT runtime·gogo(SB), NOSPLIT, $0-8
    MOVQ    buf+0(FP), BX       // gobuf
    MOVQ    gobuf_g(BX), DX
    MOVQ    0(DX), CX           // make sure g != nil
    get_tls(CX)
    MOVQ    DX, g(CX)
    MOVQ    gobuf_sp(BX), SP    // 从 G.sched 恢复执行现场。
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_pc(BX), BX    // G.sched.pc 指向 goroutine 任务函数。
    JMP     BX                  // 执行该函数。
```

汇编函数从 sched 恢复现场，将寄存器指向 G 用户栈，然后跳转到任务函数开始执行。

这里有个问题，汇编函数并没有保存 execute 返回现场，也就是说等任务函数结束后，执行绪不会回到 execute。那 goexit 如何执行？如何继续循环调用？

要解释清这个问题，需要埋一个在创建任务时留下的坑：gostartcallfn。

proc.c

```
G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    ...

    newg->sched.sp = (uintptr)sp;
    newg->sched.pc = (uintptr)runtime·goexit + PCQuantum;
    newg->sched.g = newg;

    runtime·gostartcallfn(&newg->sched, fn);

    ...

    return newg;
}
```

stack.c

```
void runtime·gostartcallfn(Gobuf *gobuf, FuncVal *fv)
{
    runtime·gostartcall(gobuf, fn, fv);
}
```

sys_x86.c

```
void runtime·gostartcall(Gobuf *gobuf, void (*fn)(void), void *ctxt)
{
    sp = (uintptr*)gobuf->sp;

    // 将 pc，也就是 goexit 地址入栈。
    *--sp = (uintptr)gobuf->pc;
    gobuf->sp = (uintptr)sp;

    // 将 pc 指向 fn。
    gobuf->pc = (uintptr)fn;
    gobuf->ctxt = ctxt;
}
```

很有意思，在 gostartcall 中，提前将 goexit 地址压入 G 栈。

汇编函数 gogo 是 long jmp，也就是说当任务函数执行结束时，其尾部 RET 指令从栈上弹给 PC 寄存器的是 goexit 地址，这就是秘密所在。

asm_amd64.s

```
TEXT runtime·goexit(SB),NOSPLIT,$0-0
    BYTE    $0x90                // NOP
    CALL    runtime·goexit1(SB)  // does not return
```

proc.c

```
void runtime·goexit1(void)
{
    fn = goexit0;
    runtime·mcall(&fn);
}

static void goexit0(G *gp)
{
    runtime·casgstatus(gp, Grunning, Gdead);

    // 将 G 放回 P.gfree 复用链表。
    gfput(g->m->p, gp);

    schedule();
}
```

任务结束，当前 G 对象被放回复用链表，而线程则继续 schedule 循环往复。

proc.c

```
static void gfput(P *p, G *gp)
{
    stksize = gp->stack.hi - gp->stack.lo;

    // 如果不是默认栈，释放。
    if(stksize != FixedStack) {
        runtime·stackfree(gp->stack);
        gp->stack.lo = 0;
        gp->stack.hi = 0;
        gp->stackguard0 = 0;
    }

    // 添加到复用链表。
    gp->schedlink = p->gfree;
    p->gfree = gp;
```

```
    p->gfreecnt++;

    // 如果 P 复用链表过长 ...
    if(p->gfreecnt >= 64) {
        // 将超出的复用对象转移到全局链表。
        while(p->gfreecnt >= 32) {
            p->gfreecnt--;
            gp = p->gfree;
            p->gfree = gp->schedlink;
            gp->schedlink = runtime·sched.gfree;
            runtime·sched.gfree = gp;
            runtime·sched.ngfree++;
        }
    }
}
```

另外，在 schedule 里还提到 lockedg，这表示该 G 任务只能交由指定 M 执行，且该 M 在绑定解除前会被休眠，不再执行其他任务。当 M 遇到 lockedg，需要将 P 和 G 都交给绑定 M 去执行。

proc.c

```
static void schedule(void)
{
    // 如果当前 M 被绑定给某个 G，那么交出 P，休眠。
    // 直到被某个拿到 locked G 的 M 唤醒。
    if(g->m->lockedg) {
        stoplockedm();
        execute(g->m->lockedg);  // Never returns.
    }

    ...

    // 如果当前 G 被绑定到某个 M，那么将 P 和 G 都交给对方。
    // 唤醒绑定 M，自己回空闲队列。
    if(gp->lockedm) {
        startlockedm(gp);
        goto top;          // 唤醒后回到头部重新获取任务。
    }
}

static void startlockedm(G *gp)
{
    // 获取 G 绑定的 M。
    mp = gp->lockedm;

    // 将当前 P 交给对方，并唤醒。
    p = releasep();
```

```
    mp->nextp = p;
    runtime·notewakeup(&mp->park);

    // 休眠当前 M。
    stopm();
}

static void stoplockedm(void)
{
    // 上交 P，唤醒其他 M 执行任务。
    if(g->m->p) {
        p = releasep();
        handoffp(p);
    }

    // 休眠，直到被拿到 locked G 的 M 唤醒。
    runtime·notesleep(&g->m->park);
    runtime·noteclear(&g->m->park);

    // 绑定对方交过来的 P。
    acquirep(g->m->nextp);
    g->m->nextp = nil;
}

static void goexit0(G *gp)
{
    // 解除绑定。
    gp->lockedm = nil;
    g->m->lockedg = nil;

    schedule();
}
```

在 cgo 里就用 lockOSThread 锁定线程。

proc.c

```
static void lockOSThread(void)
{
    g->m->lockedg = g;
    g->lockedm = g->m;
}

static void unlockOSThread(void)
{
    g->m->lockedg = nil;
    g->lockedm = nil;
}
```

cgocall.go

```
func cgocall(fn, arg unsafe.Pointer) {
    cgocall_errno(fn, arg)
}

func cgocall_errno(fn, arg unsafe.Pointer) int32 {
    /*
     * Lock g to m to ensure we stay on the same stack if we do a
     * cgo callback. Add entry to defer stack in case of panic.
     */
    lockOSThread()
    mp := getg().m
    mp.ncgocall++
    mp.ncgo++
    defer endcgo(mp)

    /*
     * Announce we are entering a system call
     * so that the scheduler knows to create another
     * M to run goroutines while we are in the
     * foreign code.
     */
    entersyscall()
    errno := asmcgocall_errno(fn, arg)  // 切换到 g0 stack 执行。
    exitsyscall()

    return errno
}

func endcgo(mp *m) {
    unlockOSThread()
}
```

调度器提供了两种方式暂时中断 G 任务执行。

proc.go

```
func Gosched() {
    mcall(gosched_m)   // mcall 保存执行现场到 G.sched，然后切换到 g0 栈。
}
```

proc.c

```
void runtime·gosched_m(G *gp)
{
    // 将状态从正在运行调整为可运行。
    runtime·casgstatus(gp, Grunning, Grunnable);
```

```
    // 将 G 重新放回全局队列。
    globrunqput(gp);

    // 当前 M 继续查找并执行其他任务。
    schedule();
}
```

与 gosched 不同，gopark 并不会将 G 放回待运行队列。

proc.go

```
func gopark(unlockf unsafe.Pointer, lock unsafe.Pointer, reason string) {
    mcall(park_m)
}
```

proc.c

```
void runtime·park_m(G *gp)
{
    // 修改状态为等待。
    runtime·casgstatus(gp, Grunning, Gwaiting);

    // 当前 M 继续获取并执行其他任务。
    schedule();
}
```

直到显式调用 goready 将该 G 重新放回队列。

proc.go

```
func goready(gp *g) {
    onM(ready_m)
}
```

proc.c

```
void runtime·ready_m(void)
{
    runtime·ready(gp);
}

void runtime·ready(G *gp)
{
    // 修改状态为可运行。
    runtime·casgstatus(gp, Gwaiting, Grunnable);

    // 将 G 重新放回本地待运行队列。
    runqput(g->m->p, gp);
```

```
    // 唤醒某个 M 执行任务。
    if(runtime·atomicload(&runtime·sched.npidle) != 0 &&
       runtime·atomicload(&runtime·sched.nmspinning) == 0)
        wakep();
}
```

3.5 连续栈

连续栈的地位被正式确定下来，Go 1.4 已经移除了分段栈代码。

相比较分段栈，连续栈结构更简单，实现算法也更加简洁。除栈开始、结束地址外，只需维护一个用于溢出检查的指针即可。

runtime.h

```
struct  Stack
{
    uintptr lo;
    uintptr hi;
};

struct  G
{
    // stack describes the actual stack memory: [stack.lo, stack.hi).
    // stackguard0 is the stack pointer compared in the Go stack growth prologue.
    // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a
    // preemption.
    Stack   stack;
    uintptr stackguard0;
}
```

结构示意图:

```
lo                stackguard0                                                 hi
+---------------+-----------------------------------------------------------+
|  StackGuard   |                                                           |
+---------------+-----------------------------------------------------------+
                                                                  <-- SP
```

在 stack.h 头部栈结构布局说明中有对 StackGuard 的详细解释。

stack.h

```
/*
The per-goroutine g->stackguard is set to point StackGuard bytes
```

```
above the bottom of the stack.  Each function compares its stack
pointer against g->stackguard to check for overflow.  To cut one
instruction from the check sequence for functions with tiny frames,
the stack is allowed to protrude StackSmall bytes below the stack
guard.  Functions with large frames don't bother with the check and
always call morestack.
*/

StackGuard = 512 + StackSystem
```

在经过几个版本的反复调整后，栈默认大小又回到 2048 字节，这是个利好消息。

stack.h

```
StackMin = 2048,
```

proc.c

```
G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    if((newg = gfget(p)) == nil) {
        newg = runtime·malg(StackMin);
    }
}
```

和内存分配器的的做法类似，调度器会在 cache 上缓存 stack 对象。

malloc.h

```
// Number of orders that get caching.  Order 0 is FixedStack
// and each successive order is twice as large.
NumStackOrders = 3,
```

runtime.h

```
struct StackFreeList
{
    MLink   *list;    // linked list of free stacks
    uintptr size;     // total size of stacks in list
};

struct MCache
{
    StackFreeList stackcache[NumStackOrders];
};
```

被缓存的 stack 依据大小分成 3 种 order，这与 FixedStack 值有很大关系。

stack.h

```
#ifdef GOOS_windows
    StackSystem = 512 * sizeof(uintptr),
#else
    #ifdef GOOS_plan9
        StackSystem = 512,
    #else
        StackSystem = 0,
    #endif  // Plan 9
#endif      // Windows

// The minimum stack size to allocate.
// The hackery here rounds FixedStack0 up to a power of 2.
FixedStack0 = StackMin + StackSystem,
FixedStack1 = FixedStack0 - 1,
FixedStack2 = FixedStack1 | (FixedStack1 >> 1),
FixedStack3 = FixedStack2 | (FixedStack2 >> 2),
FixedStack4 = FixedStack3 | (FixedStack3 >> 4),
FixedStack5 = FixedStack4 | (FixedStack4 >> 8),
FixedStack6 = FixedStack5 | (FixedStack5 >> 16),
FixedStack  = FixedStack6 + 1,
```

对 Linux、darwin 等系统而言，FixedStack 值和 StackMin 相同。 因此被缓存 stack 大小分别是：

```
FiexedStack           = 2048
FixedStack << order 0 = 2048
FixedStack << order 1 = 4096
FixedStack << order 2 = 8192
```

在确定相关结构和参数后，看看 stack 具体的分配过程。

malloc.h

```
StackCacheSize = 32*1024,    // Per-P, per order stack segment cache size.
```

stack.c

```
Stack runtime·stackalloc(uint32 n)
{
    // 从复用链表分配，或直接从 heap.span 分配。
    if(StackCache && n < FixedStack << NumStackOrders && n < StackCacheSize) {
        // 计算对应的 stack order。
        order = 0;
        n2 = n;
        while(n2 > FixedStack) {
            order++;
```

```
            n2 >>= 1;
        }

        c = g->m->mcache;
        if(c == nil || g->m->gcing || g->m->helpgc) {
            // 从全局缓存分配。
            x = poolalloc(order);
        } else {
            // 从本地 cache 分配。
            x = c->stackcache[order].list;

            // 如果本地没有复用 stack，则从全局缓存转移一批过来。
            if(x == nil) {
                stackcacherefill(c, order);
                x = c->stackcache[order].list;
            }

            // 调整链表。
            c->stackcache[order].list = x->next;
            c->stackcache[order].size -= n;
        }
        v = (byte*)x;
    } else {
        // 直接从 heap.spans 分配。
        s = runtime·MHeap_AllocStack(&runtime·mheap, ROUND(n, PageSize) >> PageShift);
        v = (byte*)(s->start<<PageShift);
    }

    return (Stack){(uintptr)v, (uintptr)v+n};
}
```

整个过程和从 cache 分配 object 如出一辙。而在释放时，调度器会主动将过多的复用对象从本地转移到全局缓存。

stack.c

```
void runtime·stackfree(Stack stk)
{
    n = stk.hi - stk.lo;
    v = (void*)stk.lo;

    if(StackCache && n < FixedStack << NumStackOrders && n < StackCacheSize) {
        // 计算 order。
        order = 0;
        n2 = n;
        while(n2 > FixedStack) {
            order++;
            n2 >>= 1;
```

```
        }

        x = (MLink*)v;
        c = g->m->mcache;
        if(c == nil || g->m->gcing || g->m->helpgc) {
            // 归还给全局缓存。
            poolfree(x, order);
        } else {
            // 如果本地缓存超出容量限制，则归还一批给全局缓存。
            if(c->stackcache[order].size >= StackCacheSize)
                stackcacherelease(c, order);

            // 添加到 cache 本地链表。
            x->next = c->stackcache[order].list;
            c->stackcache[order].list = x;
            c->stackcache[order].size += n;
        }
    } else {
        // 归还给 heap。
        s = runtime·MHeap_Lookup(&runtime·mheap, v);
        runtime·MHeap_FreeStack(&runtime·mheap, s);
    }
}
```

全局缓存池基本上就是对 span 链表的操作，类似做法在内存分配器章节早已见过。

stack.c

```
MSpan runtime·stackpool[NumStackOrders];   // 全局缓存。

void runtime·stackinit(void)
{
    for(i = 0; i < NumStackOrders; i++)
        runtime·MSpanList_Init(&runtime·stackpool[i]);
}

static MLink* poolalloc(uint8 order)
{
    MSpan *list;
    MSpan *s;
    MLink *x;

    list = &runtime·stackpool[order];
    s = list->next;
    if(s == list) {
        // 如果没有 stack 可用，则从 heap 获取一个 span。
        s = runtime·MHeap_AllocStack(&runtime·mheap, StackCacheSize >> PageShift);
```

```
        // 切分。
        for(i = 0; i < StackCacheSize; i += FixedStack << order) {
            x = (MLink*)((s->start << PageShift) + i);
            x->next = s->freelist;
            s->freelist = x;
        }

        // 插入链表。
        runtime·MSpanList_Insert(list, s);
    }

    x = s->freelist;
    s->freelist = x->next;
    s->ref++;
    if(s->freelist == nil) {
        // all stacks in s are allocated.
        runtime·MSpanList_Remove(s);
    }
    return x;
}

static void poolfree(MLink *x, uint8 order)
{
    MSpan *s;

    s = runtime·MHeap_Lookup(&runtime·mheap, x);
    if(s->freelist == nil) {
        // 有对象归还，自然要重新放回复用链表中。
        runtime·MSpanList_Insert(&runtime·stackpool[order], s);
    }

    x->next = s->freelist;
    s->freelist = x;
    s->ref--;

    // 如果该 span 内存被全部收回，还给 heap。
    if(s->ref == 0) {
        runtime·MSpanList_Remove(s);
        s->freelist = nil;
        runtime·MHeap_FreeStack(&runtime·mheap, s);
    }
}
```

相关的批量转移和归还操作也没什么值得深究的。

stack.c

```
static void stackcacherefill(MCache *c, uint8 order)
```

```
{
    MLink *x, *list;
    uintptr size;

    // Grab some stacks from the global cache.
    // Grab half of the allowed capacity (to prevent thrashing).
    list = nil;
    size = 0;

    while(size < StackCacheSize/2) {
        x = poolalloc(order);
        x->next = list;
        list = x;
        size += FixedStack << order;
    }

    c->stackcache[order].list = list;
    c->stackcache[order].size = size;
}

static void stackcacherelease(MCache *c, uint8 order)
{
    MLink *x, *y;
    uintptr size;

    x = c->stackcache[order].list;
    size = c->stackcache[order].size;

    while(size > StackCacheSize/2) {
        y = x->next;
        poolfree(x, order);
        x = y;
        size -= FixedStack << order;
    }

    c->stackcache[order].list = x;
    c->stackcache[order].size = size;
}
```

连续栈的调整行为是由编译器偷偷完成的。反汇编可执行文件，你会看到编译器会在函数头部插入 morestack 调用，这是运行时检查栈内存的关键。

```
(gdb) disass
Dump of assembler code for function main.main:
   0x000000000000207f <+15>:     call   0x2ddf0 <runtime.morestack_noctxt>
```

asm_amd64.s

```
TEXT runtime·morestack_noctxt(SB),NOSPLIT,$0
    MOVL    $0, DX
    JMP     runtime·morestack(SB)

TEXT runtime·morestack(SB),NOSPLIT,$0-0
    MOVQ    (g_sched+gobuf_sp)(BP), SP
    CALL    runtime·newstack(SB)
```

当需要调整栈大小时，会调用 newstack 完成连续栈的重新分配。

stack.c

```
// Called from runtime·morestack when more stack is needed.
// Allocate larger stack and relocate to new stack.
// Stack growth is multiplicative, for constant amortized cost.
void runtime·newstack(void)
{
    gp = g->m->curg;

    // 修改状态。
    runtime·casgstatus(gp, Grunning, Gwaiting);
    gp->waitreason = runtime·gostringnocopy((byte*)"stack growth");

    // 调整执行现场参数。
    runtime·rewindmorestack(&gp->sched);

    // 新栈需要 2 倍空间。
    oldsize = gp->stack.hi - gp->stack.lo;
    newsize = oldsize * 2;

    // 分配新 stack，并将数据拷贝到新站。
    copystack(gp, newsize);

    // 恢复状态，继续执行。
    runtime·casgstatus(gp, Gwaiting, Grunning);
    runtime·gogo(&gp->sched);
}
```

与分段栈相比，连续栈核心算法 copystack 非常简单易懂。

stack.c

```
static void copystack(G *gp, uintptr newsize)
{
    old = gp->stack;
    used = old.hi - gp->sched.sp;

    // 创建新栈。
```

```
    new = runtime·stackalloc(newsize);

    // ... 一些栈内容调整操作 ...

    // 拷贝栈数据。
    runtime·memmove((byte*)new.hi - used, (byte*)old.hi - used, used);

    // 切换到新栈。
    gp->stack = new;
    gp->stackguard0 = new.lo + StackGuard;
    gp->sched.sp = new.hi - used;

    // 释放旧栈。
    if(newsize > old.hi-old.lo) {
        // 扩张，立即释放。
        runtime·stackfree(old);
    } else {
        // 收缩操作有点复杂，因为原栈上的某些数据可能对垃圾回收器有用。
        // 放到一个临时队列，等待垃圾回收器处理。
        *(Stack*)old.lo = stackfreequeue;
        stackfreequeue = old;
    }
}
```

垃圾回收器会清理所有缓存，释放掉临时存储的 stack，并收缩栈内存。

mgc0.c

```
static void gc(struct gc_args *args)
{
    runtime·shrinkfinish();
}

static void markroot(ParFor *desc, uint32 i)
{
    switch(i) {
    case RootData:
        ...
    case RootFlushCaches:
        flushallmcaches();                  // 清理 cache。
        break;
    default:
        gp = runtime·allg[i - RootCount];
        runtime·shrinkstack(gp);            // 收缩栈内存。
        break;
    }
}
```

```
static void flushallmcaches(void)
{
    // Flush MCache's to MCentral.
    for(pp=runtime·allp; p=*pp; pp++) {
        c = p->mcache;
        runtime·MCache_ReleaseAll(c);
        runtime·stackcache_clear(c);        // 释放 cache 里缓存的 stack。
    }
}
```

stack.c

```
static Stack stackfreequeue;

// 清理临时 stack。
void runtime·shrinkfinish(void)
{
    s = stackfreequeue;
    stackfreequeue = (Stack){0,0};

    while(s.lo != 0) {
        t = *(Stack*)s.lo;
        runtime·stackfree(s);
        s = t;
    }
}

// 收缩栈内存。
void runtime·shrinkstack(G *gp)
{
    oldsize = gp->stack.hi - gp->stack.lo;
    newsize = oldsize / 2;
    if(newsize < FixedStack)
        return;                    // don't shrink below the minimum-sized stack

    used = gp->stack.hi - gp->sched.sp;
    if(used >= oldsize / 4)
        return;                    // still using at least 1/4 of the segment.

    copystack(gp, newsize);
}

// 释放所有 cache 持有的 stack 缓存。
void runtime·stackcache_clear(MCache *c)
{
    for(order = 0; order < NumStackOrders; order++) {
        x = c->stackcache[order].list;
        while(x != nil) {
            y = x->next;
```

```
            poolfree(x, order);
            x = y;
        }
        c->stackcache[order].list = nil;
        c->stackcache[order].size = 0;
    }
}
```

调度器完成 G 任务后，会将其放回复用列表，并释放掉额外分配的栈内存。

proc.c

```
static void gfput(P *p, G *gp)
{
    stksize = gp->stack.hi - gp->stack.lo;

    // 如果不是默认栈，释放。
    if(stksize != FixedStack) {
        runtime·stackfree(gp->stack);
        gp->stack.lo = 0;
        gp->stack.hi = 0;
        gp->stackguard0 = 0;
    }
}
```

还有，在减少 P 数量时，会释放不再使用的关联 cache，这也会引发 stack 清理操作。

proc.c

```
static void procresize(int32 new)
{
    // free unused P's
    for(i = new; i < old; i++) {
        p = runtime·allp[i];
        runtime·freemcache(p->mcache);
    }
}
```

mcache.c

```
static void freemcache(MCache *c)
{
    runtime·stackcache_clear(c);
}
```

官方一直在宣传连续栈的好处，但实际性能表现和具体场景有关，并非处处适宜。另外，缓存对象的确可以提升性能，但过多的缓存对象放在复用链表中，却成为浪费和负担。兴许以后的版本会有更好的表现。

3.6 系统调用

为支持并发调度，专门对 syscall、cgo 等操作进行包装，以便在长时间阻塞时能切换执行其他任务。

src/syscall/asm_linux_amd64.s

```
TEXT    ·Syscall(SB),NOSPLIT,$0-56
    CALL    runtime·entersyscall(SB)
    MOVQ    16(SP), DI
    MOVQ    24(SP), SI
    MOVQ    32(SP), DX
    MOVQ    $0, R10
    MOVQ    $0, R8
    MOVQ    $0, R9
    MOVQ    8(SP), AX                // syscall entry
    SYSCALL
    CMPQ    AX, $0xfffffffffffff001
    JLS ok
    MOVQ    $-1, 40(SP)              // r1
    MOVQ    $0, 48(SP)               // r2
    NEGQ    AX
    MOVQ    AX, 56(SP)               // errno
    CALL    runtime·exitsyscall(SB)
    RET
ok:
    MOVQ    AX, 40(SP)               // r1
    MOVQ    DX, 48(SP)               // r2
    MOVQ    $0, 56(SP)               // errno
    CALL    runtime·exitsyscall(SB)
    RET
```

cgocall.go

```
func cgocall(fn, arg unsafe.Pointer) {
    cgocall_errno(fn, arg)
}

func cgocall_errno(fn, arg unsafe.Pointer) int32 {
    entersyscall()
    errno := asmcgocall_errno(fn, arg)
    exitsyscall()
```

```
    return errno
}
```

进入系统调用前保存执行现场，这是任务切换的关键。

proc.c

```
void ·entersyscall(int32 dummy)
{
    runtime·reentersyscall((uintptr)runtime·getcallerpc(&dummy),
                           runtime·getcallersp(&dummy));
}

void runtime·reentersyscall(uintptr pc, uintptr sp)
{
    // 保存现场。
    save(pc, sp);

    g->syscallsp = sp;
    g->syscallpc = pc;
    runtime·casgstatus(g, Grunning, Gsyscall);

    // 唤醒 sysmon 线程。
    if(runtime·atomicload(&runtime·sched.sysmonwait)) {
        fn = entersyscall_sysmon;
        runtime·onM(&fn);
        save(pc, sp);
    }

    // 解除任务关联引用。
    g->m->mcache = nil;
    g->m->p->m = nil;
    runtime·atomicstore(&g->m->p->status, Psyscall);
}

static void save(uintptr pc, uintptr sp)
{
    g->sched.pc = pc;
    g->sched.sp = sp;
    g->sched.lr = 0;
    g->sched.ret = 0;
    g->sched.ctxt = 0;
    g->sched.g = g;
}
```

必须确保 sysmon 线程运行，如此才能在长时间阻塞时，回收其关联 P 执行其他任务。

proc.c

```
static void entersyscall_sysmon(void)
{
    // 唤醒 sysmon M。
    if(runtime·atomicload(&runtime·sched.sysmonwait)) {
        runtime·atomicstore(&runtime·sched.sysmonwait, 0);
        runtime·notewakeup(&runtime·sched.sysmonnote);
    }
}
```

另有 entersyscallblock 会主动释放 P，用于执行可确定的长时间阻塞调用。

proc.c

```
void ·entersyscallblock(int32 dummy)
{
    save((uintptr)runtime·getcallerpc(&dummy), runtime·getcallersp(&dummy));
    g->syscallsp = g->sched.sp;
    g->syscallpc = g->sched.pc;
    runtime·casgstatus(g, Grunning, Gsyscall);

    // 释放关联 P。
    fn = entersyscallblock_handoff;
    runtime·onM(&fn);

    save((uintptr)runtime·getcallerpc(&dummy), runtime·getcallersp(&dummy));
}

static void entersyscallblock_handoff(void)
{
    // 释放 P，让其执行其他任务。
    handoffp(releasep());
}
```

从系统调用退出时，优先检查关联 P 是否还在。

proc.c

```
void ·exitsyscall(int32 dummy)
{
    // 如果能关联 P。
    if(exitsyscallfast()) {
        runtime·casgstatus(g, Gsyscall, Grunning);
        return;
    }
```

```
    fn = exitsyscall0;
    runtime·mcall(&fn);
}

static bool exitsyscallfast(void)
{
    // 如果关联 P 扔在，尝试重新关联。
    if(g->m->p && g->m->p->status == Psyscall &&
       runtime·cas(&g->m->p->status, Psyscall, Prunning)) {
        g->m->mcache = g->m->p->mcache;
        g->m->p->m = g->m;
        return true;
    }

    // 尝试关联空闲 P。
    g->m->p = nil;
    if(runtime·sched.pidle) {
        fn = exitsyscallfast_pidle;
        runtime·onM(&fn);
        if(g->m->scalararg[0]) {
            g->m->scalararg[0] = 0;
            return true;
        }
    }
    return false;
}

static void exitsyscallfast_pidle(void)
{
    p = pidleget();
    if(p) {
        acquirep(p);
    }
}
```

如快速退出失败，且无法获取可用 P，那只能将当前 G 任务放回待运行队列。

proc.c

```
static void exitsyscall0(G *gp)
{
    runtime·casgstatus(gp, Gsyscall, Grunnable);

    // 获取空闲 P。
    p = pidleget();

    // 如获取 P 失败，将当前 G 放回全局队列。
    if(p == nil)
```

```
        globrunqput(gp);

    // 关联 P，继续执行。
    if(p) {
        acquirep(p);
        execute(gp);   // Never returns.
    }

    // 关联失败，休眠当前 M。
    stopm();
    schedule();        // Never returns.
}
```

注：以 Raw 开头的函数不使用包装模式。

3.7 系统监控

调度器使用专门线程跑系统监控，主动完成那些长时间没有触发的事件。

- 将长时间没有处理的 netpoll 结果添加到任务队列。
- 收回因 syscall 长时间阻塞的 P。
- 向长时间运行的 G 任务发出抢占调度通知。
- 如果超过 2 分钟没有运行垃圾回收，那么强制启动。
- 释放那些闲置超过 5 分钟的 span 物理内存。

proc.go

```
// The main goroutine.
func main() {
    onM(newsysmon)
}
```

proc.c

```
void runtime·newsysmon(void)
{
    // 启动独立线程运行 sysmon。
    newm(sysmon, nil);
}
```

监控函数 sysmon 循环运行所有检查任务。

proc.c

```
static void sysmon(void)
{
    // If we go two minutes without a garbage collection, force one to run.
    forcegcperiod = 2*60*1e9;

    // If a heap span goes unused for 5 minutes after a garbage collection,
    // we hand it back to the operating system.
    scavengelimit = 5*60*1e9;

    // Make wake-up period small enough for the sampling to be correct.
    maxsleep = forcegcperiod/2;
    if(scavengelimit < forcegcperiod)
        maxsleep = scavengelimit/2;

    for(;;) {
        if(idle == 0)           // start with 20us sleep...
            delay = 20;
        else if(idle > 50)      // start doubling the sleep after 1ms...
            delay *= 2;
        if(delay > 10*1000)     // up to 10ms
            delay = 10*1000;

        // 根据 idle 调整循环暂停时间。
        runtime·usleep(delay);

        // 如垃圾回收启动，休眠 sysmon 线程。
        if(runtime·debug.schedtrace <= 0 && (runtime·sched.gcwaiting ...)) {
            if(runtime·atomicload(&runtime·sched.gcwaiting) || ...) {
                // 设置标志，休眠一段时间。
                runtime·atomicstore(&runtime·sched.sysmonwait, 1);
                runtime·notetsleep(&runtime·sched.sysmonnote, maxsleep);

                // 唤醒后清除等待标志。
                runtime·atomicstore(&runtime·sched.sysmonwait, 0);
                runtime·noteclear(&runtime·sched.sysmonnote);

                idle = 0;
                delay = 20;
            }
        }

        // 如超过 10ms 没处理 netpoll，立即获取，并添加到任务队列。
        lastpoll = runtime·atomicload64(&runtime·sched.lastpoll);
        if(lastpoll != 0 && lastpoll + 10*1000*1000 < now) {
            runtime·cas64(&runtime·sched.lastpoll, lastpoll, now);
            gp = runtime·netpoll(false);  // non-blocking
            if(gp) {
                injectglist(gp);
```

```
            }
        }

        // 收回因系统调用长时间阻塞的 P。
        // 向长时间运行的 G 任务发出抢占调度通知。
        if(retake(now))
            idle = 0;
        else
            idle++;

        // 如超过 2 分钟未做垃圾回收，强制启动。
        lastgc = runtime·atomicload64(&mstats.last_gc);
        if(lastgc != 0 && unixnow - lastgc > forcegcperiod && ...) {
            // 将 forcegc.G 放回任务队列，使其运行。
            injectglist(runtime·forcegc.g);
        }

        // 释放长时间闲置 span 物理内存。
        if(lastscavenge + scavengelimit/2 < now) {
            runtime·MHeap_Scavenge(nscavenge, now, scavengelimit);
            lastscavenge = now;
        }
    }
}
```

forcegc 和 scavenge 前面都已说过。retake 使用计数器判断 syscall 或 G 任务的运行时间。

proc.c

```
struct Pdesc
{
    uint32  schedtick;       // scheduler execute 执行次数。
    int64   schedwhen;
    uint32  syscalltick;     // syscall 执行次数，在 exitsyscall 结束前递增。
    int64   syscallwhen;
};

static Pdesc pdesc[MaxGomaxprocs];

static uint32 retake(int64 now)
{
    n = 0;

    // 循环检查所有 P。
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p==nil) continue;
```

```
        pd = &pdesc[i];

        s = p->status;
        if(s == Psyscall) {
            // 如果和 pdesc 中的计数不等，表示启动了新 syscall，刷新计数器。
            // 再次 retake 时，如计数依然相等，表示依然阻塞在上次 syscall 中，
            // 时间起码超过一次 sysmon sleep (最少 20us)。
            t = p->syscalltick;
            if(pd->syscalltick != t) {
                pd->syscalltick = t;
                pd->syscallwhen = now;
                continue;
            }

            // 如 P 没有其他任务，且没超过 10ms，跳过。
            if(p->runqhead == p->runqtail &&
               runtime·atomicload(&runtime·sched.nmspinning) +
               runtime·atomicload(&runtime·sched.npidle) > 0 &&
               pd->syscallwhen + 10*1000*1000 > now)
                 continue;

            // 收回被 syscall 阻塞的 P，用于执行其他任务。
            if(runtime·cas(&p->status, s, Pidle)) {
                n++;
                handoffp(p);
            }
        } else if(s == Prunning) {
            // 计数不等，表示启动新 G 任务执行，刷新计数器。
            // 再次 retake 时，如计数依然相等，表示该任务执行时间超过一次 sysmon sleep 间隔。
            t = p->schedtick;
            if(pd->schedtick != t) {
                pd->schedtick = t;
                pd->schedwhen = now;
                continue;
            }

            // 检查超时 (10ms)。
            if(pd->schedwhen + 10*1000*1000 > now)
                continue;

            // 设置抢占调度标记。
            preemptone(p);
        }
    }
    return n;
}
```

前面说过 entersyscall 会保存现场，解除引用，因此 sysmon 可以安全拿回 P。调度器会积极尝试让这个 P 跑起来，这是它的责任。

proc.c

```
static void handoffp(P *p)
{
    // if it has local work, start it straight away
    if(p->runqhead != p->runqtail || runtime·sched.runqsize) {
        startm(p, false);
        return;
    }

    // no local work, check that there are no spinning/idle M's,
    // otherwise our help is not required
    if(runtime·atomicload(&runtime·sched.nmspinning) +
       runtime·atomicload(&runtime·sched.npidle) == 0 &&
       runtime·cas(&runtime·sched.nmspinning, 0, 1)){
        startm(p, true);
        return;
    }

    // gc
    if(runtime·sched.gcwaiting) {
        p->status = Pgcstop;
        if(--runtime·sched.stopwait == 0)
            runtime·notewakeup(&runtime·sched.stopnote);
        return;
    }

    if(runtime·sched.runqsize) {
        startm(p, false);
        return;
    }

    // If this is the last running P and nobody is polling network,
    // need to wakeup another M to poll network.
    if(runtime·sched.npidle == runtime·gomaxprocs-1 &&
       runtime·atomicload64(&runtime·sched.lastpoll) != 0) {
        startm(p, false);
        return;
    }

    pidleput(p);
}
```

至于抢占调度通知不过是在 G 栈上设置一个标志。类似操作，在很多地方都能看到。

proc.c

```
// Tell the goroutine running on processor P to stop.
static bool preemptone(P *p)
{
    mp = p->m;
    if(mp == nil || mp == g->m)
        return false;

    gp = mp->curg;
    if(gp == nil || gp == mp->g0)
        return false;

    gp->preempt = true;
    // Every call in a go routine checks for stack overflow by
    // comparing the current stack pointer to gp->stackguard0.
    // Setting gp->stackguard0 to StackPreempt folds
    // preemption into the normal stack overflow check.
    gp->stackguard0 = StackPreempt;
    return true;
}
```

实际的调度行为由编译器插入到函数头部的 morestack 引发。

asm_amd64.s

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    MOVQ    (g_sched+gobuf_sp)(BP), SP
    CALL    runtime·newstack(SB)
```

stack.c

```
void runtime·newstack(void)
{
    if(gp->stackguard0 == (uintptr)StackPreempt) {
        // Act like goroutine called runtime.Gosched.
        runtime·casgstatus(gp, Gwaiting, Grunning);
        runtime·gosched_m(gp);  // never return
    }
}
```

可见抢占调度的前提是执行其他非内联函数。如果任务跑没有函数调用的无限循环，那么 M/P 就会被一直霸占，最惨的是 GOMAXPROCS = 1，其他任务都会饿死。

3.8 状态输出

使用用环境变量 GODEBUG="schedtrace=xxx" 输出调度器跟踪信息。

```
$ GOMAXPROCS=2 GODEBUG="schedtrace=1000" ./test


  +- 跟踪总耗时.
  |                          +- 空闲 P 数量。
  |                          |                            +- 处于自旋状态的 M 数量。
  |                          |                            |                                  + 全局及各 P 本地任务数量。
  |                          |                            |                                  |
5023ms: gomaxprocs=2 idleprocs=0 threads=3 spinningthreads=0 idlethreads=0 runqueue=12 [44 44]
              |                          |                                |
              |                          |                                +- 空闲 M 数量。
              |                          +- M 总数。
              +- P 总数。
```

更详细的信息,需指定 "scheddetail=1"。

```
$ GOMAXPROCS=2 GODEBUG="schedtrace=1000,scheddetail=1" ./test

SCHED 1002ms: gomaxprocs=2 idleprocs=0 threads=3 idlethreads=0 runqueue=0 ...
  P0: status=1 schedtick=4 syscalltick=3 m=0 runqsize=51 gfreecnt=0
  P1: status=1 schedtick=5 syscalltick=0 m=2 runqsize=50 gfreecnt=0
  M2: p=1 curg=10 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 ...
  M1: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=1 dying=0 helpgc=0 ...
  M0: p=0 curg=9 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 ...
  G1: status=4(sleep) m=-1 lockedm=-1
  G2: status=1() m=-1 lockedm=-1
  G3: status=1() m=-1 lockedm=-1
```

相关代码请参考 proc.c/runtime·schedtrace 函数。

4. Channel

Channel 是 Go 实现 CSP 模型的关键，鼓励用通讯来实现共享。

在具体实现上，类似 FIFO 队列，多个 G 排队等待收发操作。同步模式，从排队链表中获取一个能与之交换数据的对象；异步模式，围绕数据缓冲区空位排队。

4.1 初始化

先了解几个基本的数据类型。

SudoG 对 G 进行包装，而 WaitQ 则是 SudoG 排队链表。

runtime.h

```
struct  SudoG
{
    G*      g;
    uint32* selectdone;
    SudoG*  next;
    SudoG*  prev;
    void*   elem;          // 发送或接收的数据。
    int64   releasetime;
    int32   nrelease;      // -1 for acquire
    SudoG*  waitlink;      // G.waiting list
};
```

chan.h

```
struct  WaitQ
{
    SudoG*  first;
    SudoG*  last;
};
```

每个 channel 除发送和接收排队链表外，还由一个环状数据缓冲槽队列。

chan.h

```
struct  Hchan
{
    uintgo  qcount;          // 缓冲数据项数量。
    uintgo  dataqsiz;        // 缓冲槽数量。
    byte*   buf;             // 缓冲区指针。
```

```
    uint16  elemsize;       // 数据项长度。
    uint32  closed;         // 关闭标记。
    Type*   elemtype;       // 数据项类型。
    uintgo  sendx;          // 发送索引。
    uintgo  recvx;          // 接收索引。
    WaitQ   recvq;          // 等待接收 G 排队链表。
    WaitQ   sendq;          // 等待发送 G 排队链表。
    Mutex   lock;
};
```

创建 channel 对象时，需要指定缓冲槽数量，同步模式为 0。

chan.go

```
const (
    hchanSize = unsafe.Sizeof(hchan{}) +
                uintptr(-int(unsafe.Sizeof(hchan{}))&(maxAlign-1))
)


func makechan(t *chantype, size int64) *hchan {
    elem := t.elem

    // 数据项长度不能超过 64KB。
    if elem.size >= 1<<16 {
        gothrow("makechan: invalid channel element type")
    }

    var c *hchan
    if elem.kind&kindNoPointers != 0 || size == 0 {
        // 一次性分配 channel 和缓冲区内存。
        c = (*hchan)(mallocgc(hchanSize+uintptr(size)*uintptr(elem.size), ...))

        // 调整缓冲区指针。
        if size > 0 && elem.size != 0 {
            c.buf = (*uint8)(add(unsafe.Pointer(c), hchanSize))
        } else {
            c.buf = (*uint8)(unsafe.Pointer(c))
        }
    } else {
        // 如果数据项是指针，单独分配一个指针数组作为缓冲区。
        c = new(hchan)
        c.buf = (*uint8)(newarray(elem, uintptr(size)))
    }

    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)
```

```
    return c
}
```

4.2 收发数据

同步和异步实现算法有很大差异。但不知什么原因，运行时开发人员硬将这些塞到同一个函数里。为阅读方便，我们将其拆开说明。

同步收发操作的关键，是从排队链表里找到一个合作者。找到，直接私下交换数据；找不到，把自己打包成 SudoG，放到排队链表里，然后休眠，直到被另一方唤醒。

参数 ep 表示待发送或接收数据内存指针。

chan.go

```
func chansend(t *chantype, c *hchan, ep unsafe.Pointer, ...) bool {
    lock(&c.lock)

    // --- 同步模式 ---------------------------------------------------

    if c.dataqsiz == 0 {
        // 从接收排队链表查找接收者。
        sg := c.recvq.dequeue()
        if sg != nil {
            // 找到合作者以后，就不再需要 channel 参与。
            unlock(&c.lock)

            recvg := sg.g

            // 将数据拷贝给接收者。
            if sg.elem != nil {
                memmove(unsafe.Pointer(sg.elem), ep, uintptr(c.elemsize))
                sg.elem = nil
            }

            // 将准备唤醒的接收者 SudoG 保存到目标 G.param。
            // 同步方式的参与双方通过检查该参数来确定是被另一方唤醒。
            // 另外，channel select 用该参数获知可用的 SudoG。
            // 异步方式仅关心缓冲槽，并不需要有另一方配合，因此无需填写该参数。
            recvg.param = unsafe.Pointer(sg)

            // 唤醒接收者。
            // 注意，此时接收者已经离开排队链表，而且数据已经完成拷贝。
            goready(recvg)
```

```
            return true
        }

        // 如果找不到接收者，那么打包成 SudoG。
        gp := getg()
        mysg := acquireSudog()
        mysg.releasetime = 0
        mysg.elem = ep
        mysg.waitlink = nil
        gp.waiting = mysg
        mysg.g = gp
        mysg.selectdone = nil
        gp.param = nil

        // 放到发送者排队链表、阻塞。
        c.sendq.enqueue(mysg)
        goparkunlock(&c.lock, "chan send")

        // 被唤醒。检查 param 参数，确定是被接收者而不是 close 唤醒。
        // 被唤醒前，数据已经被接收者拷贝完成。
        gp.waiting = nil
        if gp.param == nil {
            if c.closed == 0 {
                gothrow("chansend: spurious wakeup")
            }
            panic("send on closed channel")
        }
        gp.param = nil

        releaseSudog(mysg)
        return true
    }

    return true
}
```

同步接收和同步发送流程基本一致。

chan.go

```
func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer, ...) (selected, received bool) {
    lock(&c.lock)

    // --- 同步模式 ---------------------------------------------------

    if c.dataqsiz == 0 {
        // 从发送排队链表找出一个发送者。
```

```
    sg := c.sendq.dequeue()
    if sg != nil {
        // 撇开 channel，私下交易。
        unlock(&c.lock)

        // 拷贝数据。
        if ep != nil {
            memmove(ep, sg.elem, uintptr(c.elemsize))
        }
        sg.elem = nil
        gp := sg.g

        // 设置唤醒检查参数，唤醒发送者。
        // 唤醒前，数据已经完成拷贝。
        gp.param = unsafe.Pointer(sg)
        goready(gp)

        selected = true
        received = true
        return
    }

    // 如果没有发送者，打包成 SudoG。
    gp := getg()
    mysg := acquireSudog()
    mysg.releasetime = 0
    mysg.elem = ep
    mysg.waitlink = nil
    gp.waiting = mysg
    mysg.g = gp
    mysg.selectdone = nil
    gp.param = nil

    // 放到接收排队链表、阻塞。
    c.recvq.enqueue(mysg)
    goparkunlock(&c.lock, "chan receive")

    // 检查是否被发送者唤醒，以确定数据可用。close 唤醒肯定不会有数据。
    // 唤醒前，发送者已经将数据拷贝到 ep。
    haveData := gp.param != nil
    gp.param = nil
    releaseSudog(mysg)

    if haveData {
        selected = true
        received = true
        return
    }
```

```
        lock(&c.lock)
        return recvclosed(c, ep)
    }
}
```

简单点说，要么主动找到合作方，要么去排队等着被动唤醒，这是一个双向过程。

SudoG 会被缓存到 cache，这个没什么需要特别说明的。

malloc.h

```
struct MCache
{
    SudoG*  sudogcache;
};
```

proc.go

```
func acquireSudog() *sudog {
    c := gomcache()
    s := c.sudogcache
    if s != nil {
        c.sudogcache = s.next
        s.next = nil
        return s
    }

    mp := acquirem()
    p := new(sudog)
    releasem(mp)
    return p
}

func releaseSudog(s *sudog) {
    gp := getg()
    c := gomcache()
    s.next = c.sudogcache
    c.sudogcache = s
}
```

同步关键是查找合作方，而异步关键则是缓冲区空槽。channel 使用 qcount、sendx、recvx 来维护一个环状缓冲队列。

chan.go

```
func chansend(t *chantype, c *hchan, ep unsafe.Pointer, ...) bool {
```

```
    lock(&c.lock)

    // --- 异步模式 ----------------------------------------------------

    // 如果缓冲区没有空槽。
    for c.qcount >= c.dataqsiz {
        // 打包成 SudoG。
        gp := getg()
        mysg := acquireSudog()
        mysg.releasetime = 0
        mysg.g = gp
        mysg.elem = nil
        mysg.selectdone = nil

        // 放入发送队列，休眠。
        c.sendq.enqueue(mysg)
        goparkunlock(&c.lock, "chan send")

        // 被唤醒，释放 SudoG。
        // 被唤醒前，SudoG 已经被某个接收者弹出排队链表。
        releaseSudog(mysg)

        // ... 循环重试 ...
    }

    // 有空槽，直接将数据拷贝到缓冲区。
    memmove(chanbuf(c, c.sendx), ep, uintptr(c.elemsize))

    // 调整缓冲区参数。
    c.sendx++
    if c.sendx == c.dataqsiz {
        c.sendx = 0
    }
    c.qcount++

    // 发送操作完成。
    // 缓冲区有了可用数据，尝试将某个接收者从排队链表弹出，唤醒它处理数据。
    sg := c.recvq.dequeue()
    if sg != nil {
        recvg := sg.g
        unlock(&c.lock)
        goready(recvg)
    }

    return true
}
```

将数据拷贝到缓冲区，然后尝试去唤醒某个接收者处理。同样，接收者如果找不到可用的缓存数据，会将自己放到排队链表，等待某个发送者写入数据后唤醒。

chan.go

```
func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer, ...) (selected, received bool) {
    lock(&c.lock)

    // --- 异步模式 ---------------------------------------------------

    // 如果没有可用缓存数据。
    for c.qcount <= 0 {
        // 打包成 SudoG.
        gp := getg()
        mysg := acquireSudog()
        mysg.releasetime = 0
        mysg.elem = nil
        mysg.g = gp
        mysg.selectdone = nil

        // 放到接收排队链表、阻塞。
        c.recvq.enqueue(mysg)
        goparkunlock(&c.lock, "chan receive")

        // 被唤醒。
        // 被唤醒前，SudoG 已经被弹出排队链表。
        releaseSudog(mysg)
        lock(&c.lock)

        // ... 循环重试 ...
    }

    // 如果有可用数据项，直接拷贝到 ep。
    if ep != nil {
        memmove(ep, chanbuf(c, c.recvx), uintptr(c.elemsize))
    }

    // 清除缓冲槽，调整缓冲区参数。
    memclr(chanbuf(c, c.recvx), uintptr(c.elemsize))
    c.recvx++
    if c.recvx == c.dataqsiz {
        c.recvx = 0
    }
    c.qcount--

    // 接收完成，表示有空槽，尝试唤醒某个发送者。
    sg := c.sendq.dequeue()
    if sg != nil {
```

```
        gp := sg.g
        unlock(&c.lock)
        goready(gp)
    }

    selected = true
    received = true
    return
}
```

对于 nil channel，总是阻塞。而当用 close 关闭 channel 时，会唤醒所有排队者，让它们处理完已有的操作，比如已排队等待发送的数据，或已经写入缓冲区的数据。

chan.go

```
func closechan(c *hchan) {
    // 不要 close nil channel。
    if c == nil {
        panic("close of nil channel")
    }

    // 不要多次 close channal。
    if c.closed != 0 {
        panic("close of closed channel")
    }

    // 关闭标志。
    c.closed = 1

    // 唤醒所有排队的接收者。
    for {
        sg := c.recvq.dequeue()
        if sg == nil {
            break
        }
        gp := sg.g
        sg.elem = nil
        gp.param = nil    // 如果是同步方式，表明是 closechan 唤醒。

        goready(gp)
    }

    // 唤醒所有排队的发送者。
    for {
        sg := c.sendq.dequeue()
        if sg == nil {
            break
        }
```

```
        gp := sg.g
        sg.elem = nil
        gp.param = nil

        goready(gp)
    }
}
```

总结规则如下：

发送：

- 向 nil channel 发送数据，阻塞。
- 向 closed channel 发送数据，出错。
- 同步发送: 如有接收者，交换数据。否则排队、阻塞。
- 异步发送: 如有空槽，拷贝数据到缓冲区。否则排队、阻塞。

接收：

- 从 nil channel 接收数据，阻塞。
- 从 closed channel 接收数据，返回已有数据或零值。
- 同步接收: 如有发送者，交换数据。否则排队、阻塞。
- 异步接收: 如有缓冲项，拷贝数据。否则排队、阻塞。

4.3 选择模式

编译器会将所有 case 转换为 Scase 对象，注册到 select.scase，自然也包括 default。

chan.h

```
struct  Scase
{
    void*   elem;           // data element
    Hchan*  chan;           // chan
    uintptr pc;             // return pc
    uint16  kind;
    uint16  so;             // vararg of selected bool
    bool*   receivedp;      // pointer to received bool (recv2)
    int64   releasetime;
};

struct  Select
```

```
{
    uint16  tcase;          // total count of scase[]
    uint16  ncase;          // currently filled scase[]
    uint16* pollorder;      // case poll order
    Hchan** lockorder;      // channel lock order
    Scase   scase[1];       // one per case (in order of appearance)
};
```

因为 case 语句数量是确定的，因此在初始化时，会一次性分配所需的全部内存。

select.go

```
func newselect(sel *_select, selsize int64, size int32) {
    // 确认内存长度。
    if selsize != int64(selectsize(uintptr(size))) {
        gothrow("bad select size")
    }

    sel.tcase = uint16(size)
    sel.ncase = 0

    // 确认起始地址。
    sel.lockorder = (**hchan)(add(unsafe.Pointer(&sel.scase),
                                  uintptr(size)*unsafe.Sizeof(_select{}.scase[0])))
    sel.pollorder = (*uint16)(add(unsafe.Pointer(sel.lockorder),
                                  uintptr(size)*unsafe.Sizeof(*_select{}.lockorder)))
}

// 内存组成 select + scase + lockerorder + pollorder。
func selectsize(size uintptr) uintptr {
    selsize := unsafe.Sizeof(_select{}) +
        (size-1)*unsafe.Sizeof(_select{}.scase[0]) +  // Select 已经有一个 scase[1]
        size*unsafe.Sizeof(*_select{}.lockorder) +
        size*unsafe.Sizeof(*_select{}.pollorder)
    return round(selsize, _Int64Align)
}
```

内存布局:

```
+----------+---------------+-------------------+--------------------+
|  select  |  scase array  |  lockorder array  |  pollorder array   |
+----------+---------------+-------------------+--------------------+
```

后两成员是算法需要使用的排序表。pollorder 保存乱序后的 scase 序号，如此遍历时就形成了随机选择。而 lockorder 对 case channel 地址排序，当多处使用同一 channel 时，可避免重复加锁。

注册函数并没多少玄机，无非是通过 ncase 确定注册位置。依据 case channel 操作方式，分为 send、recv、default 三种类型。

select.go

```
func selectsend(sel *_select, c *hchan, elem unsafe.Pointer) (selected bool) {
    if c != nil {
        selectsendImpl(sel, c, getcallerpc(unsafe.Pointer(&sel)), elem,
                       uintptr(unsafe.Pointer(&selected))-uintptr(unsafe.Pointer(&sel)))
    }
    return
}

func selectsendImpl(sel *_select, c *hchan, pc uintptr, elem unsafe.Pointer, so uintptr)
{
    // 当前注册位置。
    i := sel.ncase

    // 判断是否超出限制。
    if i >= sel.tcase {
        gothrow("selectsend: too many cases")
    }

    // 下一注册位置。
    sel.ncase = i + 1

    // 通过当前注册位置，获取 scase 指针。
    cas := (*scase)(add(unsafe.Pointer(&sel.scase),
                        uintptr(i)*unsafe.Sizeof(sel.scase[0])))

    cas.pc = pc
    cas._chan = c
    cas.so = uint16(so)
    cas.kind = _CaseSend
    cas.elem = elem
}
```

选择算法的实现有些复杂，又是一个超长函数，充斥大量的 goto。

select.go

```
func selectgo(sel *_select) {
    pc, offset := selectgoImpl(sel)
}

func selectgoImpl(sel *_select) (uintptr, uint16) {
    scaseslice := sliceStruct{unsafe.Pointer(&sel.scase), int(sel.ncase), ...}
    scases := *(*[]scase)(unsafe.Pointer(&scaseslice))
```

```
    // 填充 pollorder，然后洗牌形成乱序。
    pollorder := *(*[]uint16)(unsafe.Pointer(&pollslice))
    ...

    // 将 case channel 按地址排序。
    lockorder := *(*[]*hchan)(unsafe.Pointer(&lockslice))
    ...

    // 锁定全部 channel。
    sellock(sel)

loop:

    // 1: 查找已准备好的 case。

    for i := 0; i < int(sel.ncase); i++ {
        // 使用 pollorder 返回 case，这就是 select 随机选择的关键。
        cas = &scases[pollorder[i]]
        c = cas._chan

        switch cas.kind {
        case _CaseRecv:
            if c.dataqsiz > 0 {           // 异步
                if c.qcount > 0 {         // 有缓冲数据
                    goto asyncrecv
                }
            } else {                      // 同步
                sg = c.sendq.dequeue()
                if sg != nil {            // 有接收者
                    goto syncrecv
                }
            }
            if c.closed != 0 {            // 关闭
                goto rclose
            }

        case _CaseSend:
            if c.closed != 0 {
                goto sclose
            }
            if c.dataqsiz > 0 {
                if c.qcount < c.dataqsiz {
                    goto asyncsend
                }
            } else {
                sg = c.recvq.dequeue()
                if sg != nil {
```

```
                goto syncsend
            }
        }

    case _CaseDefault:
        dfl = cas
    }
}

// 如没有准备好的 case，尝试执行 default。
if dfl != nil {
    selunlock(sel)
    cas = dfl
    goto retc
}


// 2: 如果没有任何准备好的 case ...
//    打包 SudoG，放到所有 channel 排队链表，等待唤醒。

gp = getg()
done = 0
for i := 0; i < int(sel.ncase); i++ {
    cas = &scases[pollorder[i]]
    c = cas._chan

    // 创建 SudoG。
    sg := acquireSudog()
    sg.g = gp
    sg.elem = cas.elem
    sg.waitlink = gp.waiting
    gp.waiting = sg           // 全部 SudoG 链表。

    // 将 SudoG 放到 channel 排队链表。
    switch cas.kind {
    case _CaseRecv:
        c.recvq.enqueue(sg)

    case _CaseSend:
        c.sendq.enqueue(sg)
    }
}

// 休眠，等待唤醒。
gp.param = nil
gopark(unsafe.Pointer(funcPC(selparkcommit)), unsafe.Pointer(sel), "select")

// 因所有 channel SudoG 都使用当前 G，所以可被任何 channel 操作唤醒。
```

```
    sellock(sel)
    sg = (*sudog)(gp.param)   // 同步时指向被唤醒的 SudoG，异步为 nil。


    // 3: 找出被唤醒的 case channel。

    cas = nil
    sglist = gp.waiting

    // 遍历检查被唤醒的 SudoG 是否是第 2 步创建的。
    // 注意，sglist 和 pollorder 顺序一致。
    for i := int(sel.ncase) - 1; i >= 0; i-- {
        k = &scases[pollorder[i]]

        // 如果属于 ...
        if sg == sglist {
            // 同步唤醒。
            cas = k
        } else {
            // 不属于，取消排队。
            c = k._chan
            if k.kind == _CaseSend {
                c.sendq.dequeueSudoG(sglist)
            } else {
                c.recvq.dequeueSudoG(sglist)
            }
        }

        sgnext = sglist.waitlink
        sglist.waitlink = nil
        releaseSudog(sglist)
        sglist = sgnext
    }

    // 异步唤醒，回到第 1 步处理。
    if cas == nil {
        goto loop
    }

    // 同步方式下，在被唤醒前，数据已经完成交换，直接结束即可。
    selunlock(sel)
    goto retc


// 下面这些代码在前一节已经说过，此处忽略。
asyncrecv:
asyncsend:
syncrecv:
```

```
rclose:
syncsend:

retc:
    return cas.pc, cas.so

sclose:
    // send on closed channel
    selunlock(sel)
    panic("send on closed channel")
}
```

简化后的流程看上去清爽多了。

- 遍历检查可用 case，因使用 pollorder，所以过程就是随机的。
- 如没有 channel 准备好，那么尝试执行 default。
- 如都不可用，则打包 SudoG 放到所有 channel 排队。
- 当被任何 channel 操作唤醒时，遍历找到被激活的 case。

每次操作都对所有 channel 加锁，是个不小的代价。

select.go

```
func sellock(sel *_select) {
    var c *hchan
    for _, c0 := range lockorder {
        // 如果和前一 channel 地址相同，则无需加锁。
        if c0 != nil && c0 != c {
            c = c0
            lock(&c.lock)
        }
    }
}

func selunlock(sel *_select) {
    n := int(sel.ncase)
    r := 0

    // 因为 default case 的 channel 为 nil，排序后总是在 lockorder[0]，跳过。
    if n > 0 && lockorder[0] == nil {
        r = 1
    }

    for i := n - 1; i >= r; i-- {
        c := lockorder[i]
        if i > 0 && c == lockorder[i-1] {
```

```
            continue      // will unlock it on the next iteration
        }
        unlock(&c.lock)
    }
}
```

如果在 select 语句外套上循环，那就意味着每次循环都要创建对象，完成注册、洗牌、排序、选择等一大堆操作。

5. Defer

反编译简单示例，看看 defer 的真实面目。

```
package main

import ()

func main() {
    x := 0x100
    defer println(x)
}
```

```
(gdb) disas main.main
Dump of assembler code for function main.main:
   0x0000000000002016 <+22>:    sub    rsp,0x8
   0x000000000000201a <+26>:    mov    rcx,0x100
   0x0000000000002021 <+33>:    mov    QWORD PTR [rsp],rcx
   0x0000000000002025 <+37>:    lea    rcx,[rip+0x4a5fc]   # 0x4c628 <main.print.1.f>
   0x000000000000202c <+44>:    push   rcx
   0x000000000000202d <+45>:    push   0x8
   0x000000000000202f <+47>:    call   0xad80 <runtime.deferproc>
   0x0000000000002034 <+52>:    pop    rcx
   0x0000000000002035 <+53>:    pop    rcx
   0x0000000000002036 <+54>:    test   rax,rax
   0x0000000000002039 <+57>:    jne    0x2046 <main.main+70>
   0x000000000000203b <+59>:    nop
   0x000000000000203c <+60>:    call   0xb490 <runtime.deferreturn>
   0x0000000000002041 <+65>:    add    rsp,0x8
   0x0000000000002045 <+69>:    ret
```

不算太复杂，编译器将其处理成 deferproc 和 deferreturn 两个函数。

panic.go

```
func deferproc(siz int32, fn *funcval) {
    // 编译器依次将 args、fn、siz 入栈。
    // 通过 fn 在栈上的地址，确认 args。
    argp := uintptr(unsafe.Pointer(&fn))
    argp += unsafe.Sizeof(fn)

    mp := acquirem()
    mp.scalararg[0] = uintptr(siz)
    mp.ptrarg[0] = unsafe.Pointer(fn)
    mp.scalararg[1] = argp
    mp.scalararg[2] = getcallerpc(unsafe.Pointer(&siz))
```

```
    onM(deferproc_m)
    releasem(mp)
}
```

panic.c

```
void runtime·deferproc_m(void)
{
    siz = g->m->scalararg[0];
    fn = g->m->ptrarg[0];
    argp = g->m->scalararg[1];
    callerpc = g->m->scalararg[2];

    d = runtime·newdefer(siz);
    d->fn = fn;
    d->pc = callerpc;
    d->argp = argp;

    // 将参数拷贝到 defer.argp。
    runtime·memmove(d+1, (void*)argp, siz);
}
```

依照 Defer 结构和内存对齐，指针运算 "d+1" 就是 argp，只是这写法真的好吗？

runtime.h

```
struct Defer
{
    int32       siz;
    bool        started;
    uintptr     argp;      // where args were copied from
    uintptr     pc;
    FuncVal*    fn;
    Panic*      panic;     // panic that is running defer
    Defer*      link;
};
```

和以往一样，Defer 会被复用。

runtime.h

```
struct P
{
    Defer*  deferpool[5];
};
```

panic.go

```
func newdefer(siz int32) *_defer {
```

```
    var d *_defer

    // 按 16 字节对齐后计算长度索引。
    sc := deferclass(uintptr(siz))
    mp := acquirem()

    // 复用 P.deferpool[5] 里的 Defer 对象。
    if sc < uintptr(len(p{}.deferpool)) {
        pp := mp.p
        d = pp.deferpool[sc]
        if d != nil {
            pp.deferpool[sc] = d.link
        }
    }

    // 超出长度限制的，直接分配。
    if d == nil {
        // Allocate new defer+args.
        total := goroundupsize(totaldefersize(uintptr(siz)))
        d = (*_defer)(mallocgc(total, deferType, 0))
    }

    d.siz = siz
    gp := mp.curg

    // 添加到链表。
    d.link = gp._defer
    gp._defer = d

    releasem(mp)
    return d
}
```

所有链表被保存到 G.defer 链表。

runtime.h

```
struct  G
{
    Defer*  defer;
};
```

在函数退出前，deferreturn 完成 Defer 调用。

panic.go

```
func deferreturn(arg0 uintptr) {
    gp := getg()
```

```
    // 从链表提取一个 Defer。
    d := gp._defer
    if d == nil {
        return
    }

    // 调用 deferproc 后，会 pop 掉 siz、fn，那么 arg0 就是 argp。
    // 如果地址不等，显然就属于无效调用。
    argp := uintptr(unsafe.Pointer(&arg0))
    if d.argp != argp {
        return
    }

    mp := acquirem()

    // 复制参数。
    // 很无语，这又出了一个 deferArgs，和前面的 d+1 一个意思。
    // 这真的是同一个人写的代码?
    memmove(unsafe.Pointer(argp), deferArgs(d), uintptr(d.siz))

    fn := d.fn
    d.fn = nil
    gp._defer = d.link
    freedefer(d)
    releasem(mp)

    // 执行 defer 函数。
    jmpdefer(fn, argp)
}
```

汇编函数 jmpdefer 很有意思。

asm_amd64.s

```
// void jmpdefer(fn, sp);
// called from deferreturn.
// 1. pop the caller
// 2. sub 5 bytes from the callers return
// 3. jmp to the argument
TEXT runtime·jmpdefer(SB), NOSPLIT, $0-16
    MOVQ    fv+0(FP), DX    // fn
    MOVQ    argp+8(FP), BX  // caller sp
    LEAQ    -8(BX), SP      // caller sp after CALL
    SUBQ    $5, (SP)        // return to CALL again
    MOVQ    0(DX), BX
    JMP     BX              // but first run the deferred function
```

简单点说就是找出 "call deferreturn" 时入栈的 PC 寄存器地址。

```
   0x000000000000203c <+60>:    call   0xb490 <runtime.deferreturn>
   0x0000000000002041 <+65>:    add    rsp,0x8
```

因 PC 寄存器指向下一条指令，那么栈上值应该就是 0x2041，减去 call 指令长度 5，结果就是 0x203c。将此地址入栈，等 jmpdefer 结束，deferreturn RET 指令所恢复的 PC 寄存器值就又回到了 "call deferreturn"。配合对 argp 地址检查，就实现了函数内多个 Defer 的调用。

如果调用 Goexit 终止 goroutine，那么直接循环调用链上的所有 Defer 即可。

panic.go

```
func Goexit() {
    // Run all deferred functions for the current goroutine.
    gp := getg()
    for {
        d := gp._defer
        if d == nil {
            break
        }
        if d.started {
            d.fn = nil
            gp._defer = d.link
            freedefer(d)
            continue
        }
        d.started = true
        reflectcall(unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz), uint32(d.siz))

        d._panic = nil
        d.fn = nil
        gp._defer = d.link
        freedefer(d)
    }
    goexit()
}
```

回过头想想，一个完整 defer 过程要处理缓存对象，参数拷贝，以及多次函数调用，显然要比直接函数调用慢得多。

```
var lock sync.Mutex

func test() {
```

```
    lock.Lock()
    lock.Unlock()
}

func testdefer() {
    lock.Lock()
    defer lock.Unlock()
}

func BenchmarkTest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        test()
    }
}

func BenchmarkTestDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        testdefer()
    }
}
```

```
BenchmarkTest        30000000     43.5 ns/op
BenchmarkTestDefer   10000000      211 ns/op
```

这对于 CPU 密集型算法有很大影响，需区别对待。

6. Finalizer

Finalizer 用途类似析构函数，在关联对象被回收时执行。

malloc.go

```
func SetFinalizer(obj interface{}, finalizer interface{}) {
    // object 类型信息。
    e := (*eface)(unsafe.Pointer(&obj))
    etyp := e._type
    ot := (*ptrtype)(unsafe.Pointer(etyp))

    // 忽略 nil 对象。
    _, base, _ := findObject(e.data)
    if base == nil {
        if e.data == unsafe.Pointer(&zerobase) {
            return
        }
    }

    // finalizer 函数类型信息。
    f := (*eface)(unsafe.Pointer(&finalizer))
    ftyp := f._type

    // 如果 finalizer 为 nil，清除。
    if ftyp == nil {
        // switch to M stack and remove finalizer
        mp := acquirem()
        mp.ptrarg[0] = e.data
        onM(removeFinalizer_m)
        releasem(mp)
        return
    }

    // 确定 finalizer goroutine 启动。
    // 所有可执行的 finalizer 都由该 goroutine 执行。
    createfing()

    // 添加 finalizer 记录。
    mp := acquirem()
    mp.ptrarg[0] = f.data
    mp.ptrarg[1] = e.data
    mp.scalararg[0] = nret
    mp.ptrarg[2] = unsafe.Pointer(fint)
    mp.ptrarg[3] = unsafe.Pointer(ot)
    onM(setFinalizer_m)
    releasem(mp)
}
```

malloc.c

```
void runtime·setFinalizer_m(void)
{
    fn = g->m->ptrarg[0];
    arg = g->m->ptrarg[1];
    nret = g->m->scalararg[0];
    fint = g->m->ptrarg[2];
    ot = g->m->ptrarg[3];

    g->m->scalararg[0] = runtime·addfinalizer(arg, fn, nret, fint, ot);
}
```

相关信息打包成 SpecialFinalizer 对象，添加到关联对象所在 span.specials 链表。

malloc.h

```
struct Special
{
    Special*    next;     // linked list in span
    uint16      offset;   // span offset of object
    byte        kind;     // kind of Special
};

struct SpecialFinalizer
{
    Special     special;
    FuncVal*    fn;
    uintptr     nret;
    Type*       fint;
    PtrType*    ot;
};

struct MSpan
{
    Special     *specials;  // linked list of special records sorted by offset.
};
```

mheap.c

```
bool runtime·addfinalizer(void *p, FuncVal *f, uintptr nret, Type *fint, PtrType *ot)
{
    SpecialFinalizer *s;

    // 创建 finalizer special 对象。
    s = runtime·FixAlloc_Alloc(&runtime·mheap.specialfinalizeralloc);
    s->special.kind = KindSpecialFinalizer;
    s->fn = f;
```

```
    s->nret = nret;
    s->fint = fint;
    s->ot = ot;

    // 添加到待执行队列。
    // 虽然传递 s->special，但因地址相同，可转换回 SpecialFinalizer。
    if(addspecial(p, &s->special))
        return true;

    // 添加失败，表示已存在，放弃。
    runtime·FixAlloc_Free(&runtime·mheap.specialfinalizeralloc, s);
    return false;
}

static bool addspecial(void *p, Special *s)
{
    // 查找 p 所在 span。
    span = runtime·MHeap_LookupMaybe(&runtime·mheap, p);

    // 确保该 span 已经完成垃圾清理。
    runtime·MSpan_EnsureSwept(span);

    offset = (uintptr)p - (span->start << PageShift);
    kind = s->kind;

    // 使用 offset、kind 检查该 special 是否已经存在。
    t = &span->specials;
    while((x = *t) != nil) {
        if(offset == x->offset && kind == x->kind) {
            return false; // already exists
        }
        if(offset < x->offset || (offset == x->offset && kind < x->kind))
            break;
        t = &x->next;
    }

    // 添加到 span.specials 链表。
    s->offset = offset;
    s->next = x;
    *t = s;

    return true;
}
```

当执行垃圾清理操作时，会检查 span.specials 链表。如果关联对象可以被回收，那么就将 finalizer 放到执行队列。

mgc0.c

```
bool runtime·MSpan_Sweep(MSpan *s, bool preserve)
{
    specialp = &s->specials;
    special = *specialp;
    while(special != nil) {
        // 通过 bitmap 检查 finalizer 关联对象状态。
        p = (byte*)(s->start << PageShift) + special->offset/size*size;
        off = (uintptr*)p - (uintptr*)arena_start;
        bitp = arena_start - off/wordsPerBitmapByte - 1;
        shift = (off % wordsPerBitmapByte) * gcBits;
        bits = (*bitp>>shift) & bitMask;

        // 如果是不可达对象。
        if((bits&bitMarked) == 0) {
            // 对象地址。
            p = (byte*)(s->start << PageShift) + special->offset;
            y = special;

            // 从链表移除。
            special = special->next;
            *specialp = special;

            // 将 finalizer 添加到执行队列，释放 special。
            if(!runtime·freespecial(y, p, size, false)) {
                // 将关联对象标记为可达状态。
                *bitp |= bitMarked << shift;
            }
        } else {
            // 存活对象，保持 special record。
            specialp = &special->next;
            special = *specialp;
        }
    }
}
```

注意，finalizer 会导致关联对象重新变成可达状态，也就是说不会被清理操作回收。这是为了保证在 finalizer 函数内能安全访问关联对象。待下次回收时，finalizer 已不存在，关联对象就可被正常收回。

mheap.c

```
bool runtime·freespecial(Special *s, void *p, uintptr size, bool freed)
{
    SpecialFinalizer *sf;

    switch(s->kind) {
```

```
    case KindSpecialFinalizer:
        // 转换回 SpecialFinalizer，放到执行队列。
        sf = (SpecialFinalizer*)s;
        runtime·queuefinalizer(p, sf->fn, sf->nret, sf->fint, sf->ot);

        // Special 已经从 span.specials 移除，回收。
        runtime·FixAlloc_Free(&runtime·mheap.specialfinalizeralloc, sf);
        return false; // don't free p until finalizer is done
    }
}
```

执行队列 FinBlock 保存多个待执行的 FianlizerSpecial，而全局变量 finq 用链表管理多个 FinBlock。

FinBlock 和其他类型一样，被缓存、复用。

malloc.h

```
struct FinBlock
{
    FinBlock  *alllink;
    FinBlock  *next;
    int32     cnt;
    int32     cap;
    Finalizer fin[1];
};
```

mgc0.c

```
FinBlock*   runtime·finq;   // list of finalizers that are to be executed
FinBlock*   runtime·finc;   // cache of free blocks
```

mgc0.c

```
void runtime·queuefinalizer(byte *p, FuncVal *fn, uintptr nret, Type *fint, PtrType *ot)
{
    // 检查是否需要新建 FinBlock。
    if(runtime·finq == nil || runtime·finq->cnt == runtime·finq->cap) {
        // 如果复用链表为空，新建。
        if(runtime·finc == nil) {
            runtime·finc = runtime·persistentalloc(FinBlockSize, 0, &mstats.gc_sys);
            runtime·finc->cap = (FinBlockSize - sizeof(FinBlock)) / sizeof(Finalizer)+1;
            runtime·finc->alllink = runtime·allfin;
            runtime·allfin = runtime·finc;
        }
        block = runtime·finc;
        runtime·finc = block->next;
        block->next = runtime·finq;
```

```
        runtime·finq = block;
    }

    // 添加到 FinBlock 队列。
    f = &runtime·finq->fin[runtime·finq->cnt];
    runtime·finq->cnt++;
    f->fn = fn;
    f->nret = nret;
    f->fint = fint;
    f->ot = ot;
    f->arg = p;

    // 有了新任务，设置唤醒标记。
    runtime·fingwake = true;
}
```

在准备好执行队列后，由专门的 goroutine fing 完成最终执行操作。

mgc0.c

```
G*  runtime·fing;   // goroutine that runs finalizers
```

malloc.go

```
var fingCreate uint32

func createfing() {
    // 仅执行一次。
    if fingCreate == 0 && cas(&fingCreate, 0, 1) {
        go runfinq()
    }
}

func runfinq() {
    for {
        // 置换全局队列。
        fb := finq
        finq = nil

        // 如果队列为空，休眠。
        if fb == nil {
            gp := getg()
            fing = gp
            fingwait = true      // 休眠标记。
            gp.issystem = true
            goparkunlock(&finlock, "finalizer wait")
            gp.issystem = false
            continue
        }
```

```
        // 循环处理所有 FinBlock。
        for fb != nil {
            // 循环处理 FinBlock 队列里所有 FinalizerSpecial。
            for i := int32(0); i < fb.cnt; i++ {
                // 执行 finalizer 函数。
                f := (*finalizer)(add(unsafe.Pointer(&fb.fin),
                                      uintptr(i)*unsafe.Sizeof(finalizer{})))
                reflectcall(unsafe.Pointer(f.fn), frame, uint32(framesz),
                            uint32(framesz))

                // 解除引用。
                f.fn = nil
                f.arg = nil
                f.ot = nil
            }

            fb.cnt = 0
            next := fb.next

            // 将当前 FinBlock 放回复用链表。
            fb.next = finc
            finc = fb

            fb = next
        }
    }
}
```

如执行队列为空，fing 会被休眠。然后在 M 查找任务时，尝试唤醒。

proc.c

```
// Finds a runnable goroutine to execute.
static G* findrunnable(void)
{
    // 如果 fing 被休眠，且唤醒标记为真，那么执行。
    if(runtime·fingwait && runtime·fingwake && (gp = runtime·wakefing()) != nil)
        runtime·ready(gp);
}
```

mgc0.c

```
G* runtime·wakefing(void)
{
    G *res;

    // 如正在休眠，且唤醒标记为真，返回 fing。
    if(runtime·fingwait && runtime·fingwake) {
```

```
        runtime·fingwait = false;    // 取消相关标记。
        runtime·fingwake = false;
        res = runtime·fing;          // 返回 fing。
    }
    return res;
}
```

第三部分 附录

A. 工具

1. 工具集

1.1 go build

参数	说明	示例
-gcflags	传递给 5g/6g/8g 编译器的参数。 (5:arm, 6:x86-64, 8:x86)	
-ldflags	传递给 5l/6l/8l 链接器的参数。	
-work	查看编译临时目录。	
-race	允许数据竞争检测 (仅支持 amd64)。	
-n	查看但不执行编译命令。	
-x	查看并执行编译命令。	
-a	强制重新编译所有依赖包。	
-v	查看被编译的包名，包括依赖包。	
-p n	并行编译所使用 CPU core 数量。默认全部。	
-o	输出文件名。	

gcflags

参数	说明	示例
-B	禁用边界检查。	
-N	禁用优化。	
-l	禁用函数内联。	
-u	禁用 unsafe 代码。	
-m	输出优化信息。	
-S	输出汇编代码。	

ldflags

参数	说明	示例
-w	禁用 DRAWF 调试信息，但不包括符号表。	
-s	禁用符号表。	
-X	修改字符串符号值。	-X main.VER '0.99' -X main.S 'abc'
-H	链接文件类型，其中包括 windowsgui。	cmd/ld/doc.go

更多参数:

```
go tool 6g -h  或 https://golang.org/cmd/gc/
go tool 6l -h  或 https://golang.org/cmd/ld/
```

1.2 go install

和 go build 参数相同，将生成文件拷贝到 bin、pkg 目录。优先使用 GOBIN 环境变量所指定目录。

1.3 go clean

参数	说明	示例
-n	查看但不执行清理命令。	
-x	查看并执行清理命令。	
-i	删除 bin、pkg 目录下的二进制文件。	
-r	清理所有依赖包临时文件。	

1.4 go get

下载并安装扩展包。默认保存到 GOPATH 指定的第一个工作空间。

参数	说明	示例
-d	仅下载，不执行安装命令。	
-t	下载测试所需的依赖包。	
-u	更新包，包括其依赖包。	
-v	查看并执行命令。	

1.5 go tool objdump

反汇编可执行文件。

```
$ go tool objdump -s "main\.\w+" test
$ go tool objdump -s "main\.main" test
```

2. 条件编译

通过 runtime.GOOS/GOARCH 判断，或使用编译约束标记。

```
// +build darwin linux
```

```
                                <--- 必须有空行，以区别包文档。
package main
```

在源文件 (.go, .h, .c, .s 等) 头部添加 "+build" 注释，指示编译器检查相关环境变量。多个约束标记会合并处理。其中空格表示 OR，逗号 AND，感叹号 NOT。

```
// +build darwin linux         --> 合并结果 (darwin OR linux) AND (amd64 AND (NOT cgo))
// +build amd64,!cgo
```

如果 GOOS、GOARCH 条件不符合，则编译器会会忽略该文件。

还可使用文件名来表示编译约束，比如 test_darwin_amd64.go。使用文件名拆分多个不同平台源文件，更利于维护。

```
$ ls -l /usr/local/go/src/pkg/runtime

-rw-r--r--@  1 yuhen  admin   11545 11 29 05:38 os_darwin.c
-rw-r--r--@  1 yuhen  admin    1382 11 29 05:38 os_darwin.h
-rw-r--r--@  1 yuhen  admin    6990 11 29 05:38 os_freebsd.c
-rw-r--r--@  1 yuhen  admin     791 11 29 05:38 os_freebsd.h
-rw-r--r--@  1 yuhen  admin     644 11 29 05:38 os_freebsd_arm.c
-rw-r--r--@  1 yuhen  admin    8624 11 29 05:38 os_linux.c
-rw-r--r--@  1 yuhen  admin    1067 11 29 05:38 os_linux.h
-rw-r--r--@  1 yuhen  admin     861 11 29 05:38 os_linux_386.c
-rw-r--r--@  1 yuhen  admin    2418 11 29 05:38 os_linux_arm.c
```

支持：*_GOOS、*_GOARCH、*_GOOS_GOARCH、*_GOARCH_GOOS 格式。

可忽略某个文件，或指定编译器版本号。更多信息参考标准库 go/build 文档。

```
// +build ignore
// +build go1.2             <--- 最低需要 go 1.2 编译。
```

自定义约束条件，需使用 "go build -tags" 参数。

test.go

```
// +build beta,debug

package main

func init() {
    println("test.go init")
```

```
}
```

输出：

```
$ go build -tags "debug beta" && ./test
test.go init

$ go build -tags "debug" && ./test
$ go build -tags "debug \!cgo" && ./test
```

3. 跨平台编译

首先得生成与平台相关的工具和标准库。

```
$ cd /usr/local/go/src

$ GOOS=linux GOARCH=amd64 ./make.bash --no-clean

# Building C bootstrap tool.
cmd/dist

# Building compilers and Go bootstrap tool for host, darwin/amd64.

cmd/6l
cmd/6a
cmd/6c
cmd/6g
...
---
Installed Go for linux/amd64 in /usr/local/go
Installed commands in /usr/local/go/bin
```

说明：参数 no-clean 避免清除其他平台文件。

然后回到项目所在目录，设定 GOOS、GOARCH 环境变量即可编译目标平台文件。

```
$ GOOS=linux GOARCH=amd64 go build -o test

$ file test
learn: ELF 64-bit LSB executable, x86-64, version 1 (SYSV)

$ uname -a
Darwin Kernel Version 12.5.0: RELEASE_X86_64 x86_64
```

4. 预处理

简单点说，go generate 扫描源代码文件，找出所有 "//go:generate" 注释，提取并执行预处理命令。

- 命令必须放在 .go 文件。
- 每个文件里可以有多个 generate 指令。
- 必须显式用 go generate 执行。
- 命令行支持环境变量。
- 按文件名顺序依次提取执行。
- 串行执行，出错终止。
- 必须以 "//go:generate" 开头，双斜线后没有空格。

不属于 build 组成部分，设计目标是提供给包开发者使用，因为包用户可能不具备命令执行环境。

```
//go:generate ls -l
//go:generate du
```

```
  参数          说明                                          示例
-----------+-------------------------------------+---------------------------------
 -x           显示并执行命令。
 -n           显示但不执行命令。
 -v           输出处理的包和源文件名。
```

还可定义别名。须提前定义，仅在当前文件内有效。

```
//go:generate -command YACC go tool yacc
//go:generate YACC -o test.go -p parse test.y
```

可用条件编译，让 go build 忽略包含 generate 的文件。

```
// +build generate

$ go generate -tags generate
```

资源：Design Document Generating code

B. 调试

1. GDB

默认情况下，编译的二进制文件已包含 DWARFv3 调试信息，只要 GDB 7.1 以上版本都可以调试。

相关选项：

- 调试: 禁用内联和优化 -gcflags "-N -l"。
- 发布: 删除调试信息和符号表 -ldflags "-w -s"。

除了使用 GDB 的断点命令外，还可以使用 runtime.Breakpoint 函数触发中断。另外，runtime/debug.PrintStack 可用来输出调用堆栈信息。

某些时候，需要手工载入 Go Runtime support (runtime-gdb.py)。

.gdbinit

```
define goruntime
    source /usr/local/go/src/runtime/runtime-gdb.py
end

set disassembly-flavor intel
set print pretty on
dir /usr/local/go/src/pkg/runtime
```

说明：OSX 环境下，可能需要以 sudo 方式启动 gdb。

2. Data Race

数据竞争 (data race) 是并发程序里不太容易发现的错误，且很难捕获和恢复错误现场。Go 运行时内置了竞争检测，允许我们使用编译器参数打开这个功能。它会记录和监测运行时内存访问状态，发出非同步访问警告信息。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)
```

```
    x := 100

    go func() {
        defer wg.Done()

        for {
            x += 1
        }
    }()

    go func() {
        defer wg.Done()
        for {
            x += 100
        }
    }()

    wg.Wait()
}
```

输出：

```
$ GOMAXPROCS=2 go run -race main.go

==================
WARNING: DATA RACE
Write by goroutine 4:
  main.func·002()
      main.go:25 +0x59

Previous write by goroutine 3:
  main.func·001()
      main.go:18 +0x59

Goroutine 4 (running) created at:
  main.main()
      main.go:27 +0x16f

Goroutine 3 (running) created at:
  main.main()
      main.go:20 +0x100
==================
```

数据竞争检测会严重影响性能，不建议在生产环境中使用。

```
func main() {
    x := 100

    for i := 0; i < 10000; i++ {
```

```
        x += 1
    }

    fmt.Println(x)
}
```

输出：

```
$ go build && time ./test

10100

real   0m0.060s
user   0m0.001s
sys    0m0.003s

$ go build -race  && time ./test

10100

real   0m1.025s
user   0m0.003s
sys    0m0.009s
```

通常作为非性能测试项启用。

```
$ go test -race
```

C. 测试

自带代码测试、性能测试、覆盖率测试框架。

- 测试代码必须保存在 *_test.go 文件。
- 测试函数命名符合 TestName 格式，Name 以大写字母开头。

注: 不要将代码放在名为 main 的目录下，这会导致 go test "cannot import main" 错误。

1. Test

使用 testing.T 相关方法决定测试状态。

testing.T

方法	说明	其他
Fail	标记失败，但继续执行该测试函数。	
FailNow	失败，立即停止当前测试函数。	
Log	输出信息。仅在失败或 -v 参数时输出。	Logf
SkipNow	跳过当前测试函数。	Skipf = SkipNow + Logf
Error	Fail + Log	Errorf
Fatal	FailNow + Log	Fatalf

main_test.go

```
package main

import (
    "testing"
    "time"
)

func sum(n ...int) int {
    var c int
    for _, i := range n {
        c += i
    }

    return c
}

func TestSum(t *testing.T) {
    time.Sleep(time.Second * 2)
```

```
    if sum(1, 2, 3) != 6 {
        t.Fatal("sum error!")
    }
}

func TestTimeout(t *testing.T) {
    time.Sleep(time.Second * 5)
}
```

默认 go test 执行所有单元测试函数，支持 go build 参数。

```
  参数             说明                                          示例
---------------+---------------------------------------+---------------------------------
 -c              仅编译，不执行测试。
 -v              显示所有测试函数执行细节。
 -run regex      执行指定的测试函数。(正则表达式)
 -parallel n     并发执行测试函数。 (默认: GOMAXPROCS)
 -timeout t      单个测试超时时间。                          -timeout 2m30s
```

```
$ go test -v -timeout 3s

=== RUN TestSum
--- PASS: TestSum (2.00 seconds)
=== RUN TestTimeout
panic: test timed out after 3s
FAIL   test   3.044s


$ go test -v -run "(?i)sum"

=== RUN TestSum
--- PASS: TestSum (2.00 seconds)
PASS
ok     test   2.044s
```

可重写 TestMain 函数，处理一些 setup/teardown 操作。

```
func TestMain(m *testing.M) {
    println("setup")
    code := m.Run()
    println("teardown")
    os.Exit(code)
}

func TestA(t *testing.T) {}
func TestB(t *testing.T) {}
```

```
func BenchmarkC(b *testing.B) {}
```

输出：

```
$ go test -v -test.bench .

setup
=== RUN TestA
--- PASS: TestA (0.00s)
=== RUN TestB
--- PASS: TestB (0.00s)
PASS
BenchmarkC   2000000000            0.00 ns/op
teardown
ok     test   0.028s
```

2. Benchmark

性能测试需要运行足够多的次数才能计算单次执行平均时间。

```
func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        if sum(1, 2, 3) != 6 {
            b.Fatal("sum")
        }
    }
}
```

默认情况下，go test 不会执行性能测试函数，须使用 "-bench" 参数。

go test

```
  参数             说明                                      示例
---------------+-----------------------------------------+--------------------------------
 -bench regex     执行指定性能测试函数。(正则表达式)
 -benchmem        输出内存统计信息。
 -benchtime t     设置每个性能测试运行时间。                  -benchtime 1m30s
 -cpu             设置并发测试。默认 GOMAXPROCS。             -cpu 1,2,4
```

```
$ go test -v -bench .

=== RUN TestSum
--- PASS: TestSum (2.00 seconds)
=== RUN TestTimeout
--- PASS: TestTimeout (5.00 seconds)
PASS
```

```
BenchmarkSum    100000000    11.0 ns/op

ok     test   8.358s


$ go test -bench . -benchmem -cpu 1,2,4 -benchtime 30s

BenchmarkSum       5000000000    11.1 ns/op    0 B/op    0 allocs/op
BenchmarkSum-2     5000000000    11.4 ns/op    0 B/op    0 allocs/op
BenchmarkSum-4     5000000000    11.3 ns/op    0 B/op    0 allocs/op

ok     test   193.246s
```

3. Example

与 testing.T 类似，区别在于通过捕获 stdout 输出来判断测试结果。

```
func ExampleSum() {
    fmt.Println(sum(1, 2, 3))
    fmt.Println(sum(10, 20, 30))
    // Output:
    // 6
    // 60
}
```

不能使用内置函数 print/println，它们默认输出到 stderr。

```
$ go test -v

=== RUN: ExampleSum
--- PASS: ExampleSum (8.058us)
PASS

ok     test   0.271s
```

Example 代码可输出到文档，详情参考包文档章节。

4. Cover

除显示代码覆盖率百分比外，还可输出详细分析记录文件。

go test

```
  参数             说明
---------------+-------------------------------------------------------------------------
 -cover          允许覆盖分析。
 -covermode      代码分析模式。 (set: 是否执行; count: 执行次数; atomic: 次数, 并发支持)
 -coverprofile   输出结果文件。
```

```
$ go test -cover -coverprofile=cover.out -covermode=count

PASS
coverage: 80.0% of statements
ok     test   0.043s


$ go tool cover -func=cover.out

test.go:    Sum             100.0%
test.go:    Add             0.0%
total:      (statements)    80.0%
```

用浏览器输出结果，能查看更详细直观的信息。包括用不同颜色标记覆盖、运行次数等。

```
$ go tool cover -html=cover.out
```

说明：将鼠标移到代码块，可以看到具体的执行次数。

5. PProf

监控程序执行，找出性能瓶颈。

```
import (
    "os"
    "runtime/pprof"
)

func main() {
    // CPU
    cpu, _ := os.Create("cpu.out")
    defer cpu.Close()
    pprof.StartCPUProfile(cpu)
    defer pprof.StopCPUProfile()

    // Memory
    mem, _ := os.Create("mem.out")
```

```
    defer mem.Close()
    defer pprof.WriteHeapProfile(mem)
}
```

除调用 runtime/pprof 相关函数外，还可直接用测试命令输出所需记录文件。

go test

```
  参数                          说明
--------------------------+------------------------------------------------------------
 -blockprofile block.out     goroutine 阻塞。
 -blockprofilerate n         超出该参数设置时间的阻塞才被记录。单位：纳秒
 -cpuprofile cpu.out         CPU。
 -memprofile mem.out         内存分配。
 -memprofilerate n           超出该参数（bytes）设置的内存分配才被记录。默认 512KB（mprof.go）
```

以 net/http 包为演示，先生成记录文件。

```
$ go test -v -test.bench "." -cpuprofile cpu.out -memprofile mem.out net/http
```

进入交互式查看模式。

```
$ go tool pprof http.test mem.out

(pprof) top5
2597.58kB of 2597.58kB total (  100%)
Dropped 421 nodes (cum <= 12.99kB)
Showing top 5 nodes out of 28 (cum >= 1536.60kB)

      flat  flat%   sum%        cum   cum%
 1024.04kB 39.42% 39.42%  1024.04kB 39.42%  encoding/asn1.parsePrintableString
  548.84kB 21.13% 60.55%   548.84kB 21.13%  mime.setExtensionType
  512.56kB 19.73% 80.28%  1536.60kB 59.16%  crypto/x509.parseCertificate
  512.14kB 19.72%   100%   512.14kB 19.72%  mcommoninit
         0     0%   100%  1536.60kB 59.16%  crypto/tls.(*Conn).Handshake
```

- flat: 仅当前函数，不包括其调用的其他函数。
- sum: 列表前几行所占百分比总和。
- cum: 当前函数完整调用堆栈。

默认输出 inuse_space，可在命令行指定其他值，包括排序方式。

```
$ go tool pprof -alloc_space -cum http.test mem.out
```

可输出函数调用的列表统计信息。

```
(pprof) peek parseCertificate

2597.58kB of 2597.58kB total (  100%)
Dropped 421 nodes (cum <= 12.99kB)
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                          1536.60kB   100% |   crypto/...ParseCertificate
  512.56kB 19.73% 19.73%  1536.60kB 59.16%                | crypto/x509.parseCertificate
                                          1024.04kB   100% |   encoding/asn1.Unmarshal
----------------------------------------------------------+-------------
```

或者是更详细的源码模式。

```
(pprof) list parseCertificate
Total: 2.54MB
ROUTINE ======================== crypto/x509.parseCertificate in crypto/x509/x509.go
  512.56kB     1.50MB (flat, cum) 59.16% of Total

         .          .    851:func parseCertificate(in *certificate) (...) {
  512.56kB   512.56kB    852:    out := new(Certificate)
         .          .    853:    out.Raw = in.Raw

         .          .    880:            return nil, err
         .          .    881:    }
         .        1MB    882:    if _, err := asn1.Unmarshal(..., &issuer); err != nil {
         .          .    883:            return nil, err
         .          .    884:    }
         .          .    885:
```

除交互模式外，还可直接输出统计结果。

```
$ go tool pprof -text http.test mem.out

2597.58kB of 2597.58kB total (  100%)
Dropped 421 nodes (cum <= 12.99kB)

      flat  flat%   sum%        cum   cum%
 1024.04kB 39.42% 39.42%  1024.04kB 39.42%  encoding/asn1.parsePrintableString
  548.84kB 21.13% 60.55%   548.84kB 21.13%  mime.setExtensionType
  512.56kB 19.73% 80.28%  1536.60kB 59.16%  crypto/x509.parseCertificate
  512.14kB 19.72%   100%   512.14kB 19.72%  mcommoninit
         0     0%   100%  1536.60kB 59.16%  crypto/tls.(*Conn).Handshake
         0     0%   100%  1536.60kB 59.16%  crypto/tls.(*Conn).clientHandshake
```

输出图形文件。

```
$ go tool pprof -web http.test mem.out
```

还可用 net/http/pprof 实时查看 runtime profiling 信息。

```
package main

import (
    _ "net/http/pprof"

    "net/http"
    "time"
)

func main() {
    go http.ListenAndServe("localhost:6060", nil)

    for {
        time.Sleep(time.Second)
    }
}
```

在浏览器中查看 http://localhost:6060/debug/pprof/。

附: 自定义统计数据，可用 expvar 导出，用浏览器访问 /debug/vars 查看。