

# Go 学习 笔记

第 3 版

好好学习 天天向上



# 前言

从 2012 年初开始学习和研究 Go 语言，虽不敢说精通，但总体上还是比较熟悉和了解的。这几年踏踏实实读着它的核心源码一路走来，有收获，也有困扰，但还算用得顺手。



就我个人看来，要想了解 Go 的本质，是需要一些 C 的底子。换句话说，抛开 goroutine 等等时髦概念，其本质上还是 Next C 的路子。

这本笔记原则上力求简洁，属于工具书而非教程。目的是在需要的时候，花上很短的时间就可从头到尾复习完所有知识点。对熟练的程序员而言，工具书似乎比教程更重要些。

至于第二部分源码剖析，则属于可选部分，算是写给那些和我一样有好奇心的童鞋看的。

本书在不侵犯作者个人权利的情况下，可自由散播。

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

雨痕 二〇一四年夏初



## 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。
  
- 2014-06-05 增加 Reflect。



# 目录

<b>第一部分 语言</b>	<b>8</b>
<b>第 1 章 类型</b>	<b>9</b>
1.1 变量	9
1.2 常量	10
1.3 基本类型	13
1.4 引用类型	14
1.5 类型转换	14
1.6 字符串	15
1.7 指针	17
1.8 自定义类型	19
<b>第 2 章 表达式</b>	<b>21</b>
2.1 保留字	21
2.2 运算符	21
2.3 初始化	22
2.4 控制流	23
<b>第 3 章 函数</b>	<b>29</b>
3.1 函数定义	29
3.2 变参	30
3.3 返回值	30
3.4 匿名函数	32
3.5 延迟调用	34
3.6 错误处理	35
<b>第 4 章 数据</b>	<b>39</b>
4.1 Array	39
4.2 Slice	40
4.3 Map	44

4.4 Struct	46
<b>第 5 章 方法</b>	<b>52</b>
5.1 方法定义	52
5.2 匿名字段	53
5.3 方法集	54
5.4 表达式	55
<b>第 6 章 接口</b>	<b>58</b>
6.1 接口定义	58
6.2 执行机制	60
6.3 接口转换	61
6.4 接口技巧	63
<b>第 7 章 并发</b>	<b>64</b>
7.1 Goroutine	64
7.2 Channel	66
<b>第 8 章 包</b>	<b>74</b>
8.1 工作空间	74
8.2 源文件	74
8.3 包结构	75
8.4 文档	77
<b>第 9 章 进阶</b>	<b>79</b>
9.1 内存布局	79
9.2 指针陷阱	80
9.3 cgo	83
9.4 Reflect	91
<b>第二部分 源码</b>	<b>106</b>
<b>1. Memory Allocator</b>	<b>107</b>
1.1 初始化	110
1.2 分配流程	113

1.3 释放流程	122
1.4 其他	126
<b>2. Garbage Collector</b>	<b>130</b>
2.1 垃圾回收	130
2.2 内存释放	136
2.3 状态输出	139
<b>3. Goroutine Scheduler</b>	<b>145</b>
3.1 初始化	145
3.2 创建任务	151
3.3 执行任务	154
3.4 连续栈	164
3.5 系统调用	172
3.6 系统监控	177
3.7 状态输出	179
<b>4. Channel</b>	<b>180</b>
4.1 Send	181
4.2 Receive	184
4.3 Select	187
<b>5. Defer</b>	<b>196</b>
<b>第三部分 附录</b>	<b>203</b>
<b>A. 工具</b>	<b>204</b>
1. 工具集	204
2. 条件编译	205
3. 跨平台编译	207
<b>B. 调试</b>	<b>208</b>
1. GDB	208
2. Data Race	208
<b>C. 测试</b>	<b>211</b>

1. Test	211
2. Benchmark	212
3. Example	213
4. Cover	214
5. PProf	215

# 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)    // slice.goc: 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      // 使用 runtime.new 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)
}

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，那怕为空。

    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 `error.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：在包内部使用 `panic`，对外 API 使用 `error` 返回值。

## 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
{
    byte*    array;    // must not move anything
    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。

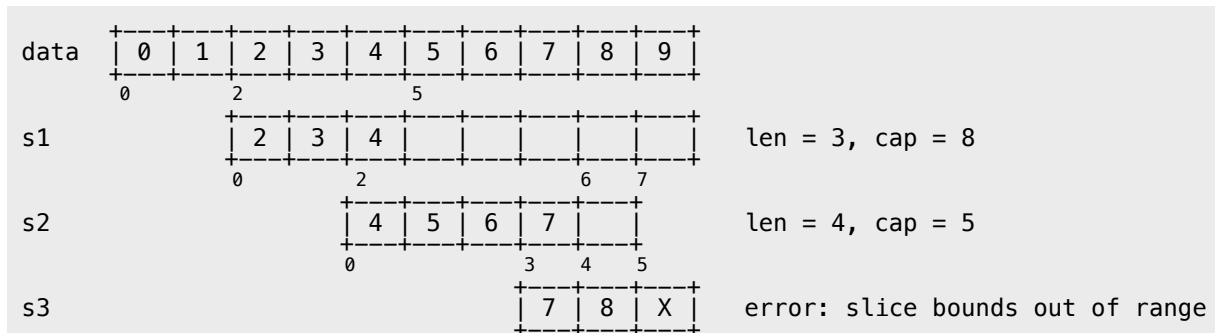
```
data := [][]int{
    []int{1, 2, 3},
    []int{100, 200},
    []int{11, 22, 33, 44},
}
```

### 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
```



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, 5, 6, 7, 8, 9}

s := data[:3:4]
fmt.Println(s)
```

```
s2 := append(s, 100, 200)
fmt.Println(s2)

fmt.Println(data)
fmt.Println(&data[0], &s[0], &s2[0])
```

输出:

```
[0 1 2] // s, s.cap = 4
[0 1 2 100 200] // s2
[0 1 2 3 4 5 6 7 8 9] // data
0x21020e140 0x21020e140 0x210232000 // pointer: data, s->array, s2->array
```

从输出结果可以看出, `s2` 重新分配了底层数组, 并复制数据。如果只追加一个值, 正好没超过 `s.cap` 限制, 那么就不会重新分配数组。

大批量添加数据时, 建议一次性分配 `len` 足够大的 `slice`, 然后用索引号进行操作。还有, 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。

### 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{
    1: {"user1"},
}

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

```

正确做法是完整替换或使用指针。

```

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。

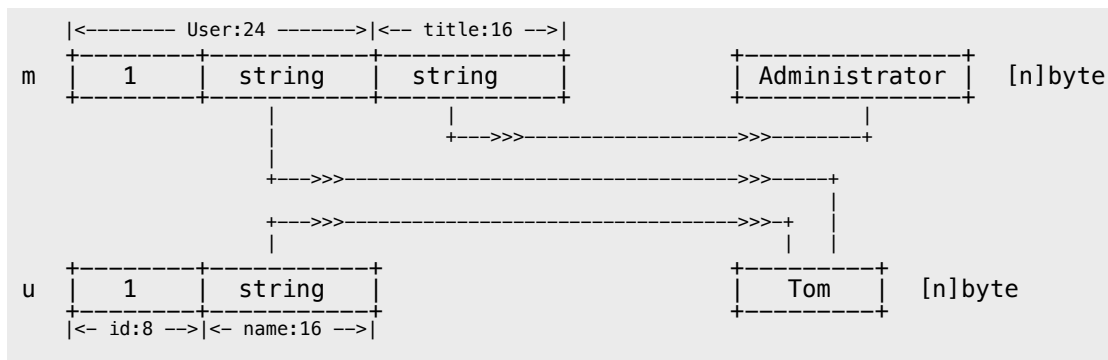
```
type User struct {
    id    int
    name  string
}

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。



可以用 `unsafe` 包中的相关函数输出内存布局信息。

```
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

```

## 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                         // 在缓冲区未空前, 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

如果需要同步处理多个 **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 NewConsumer() chan int {
    data := make(chan int, 3)

    go func() {
        for d := range data {
            fmt.Println(d)
        }

        os.Exit(0)
    }()

    return data
}

func main() {
    data := NewConsumer()

    data <- 1
    data <- 2
    close(data)

    select {}
}

```

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
    }

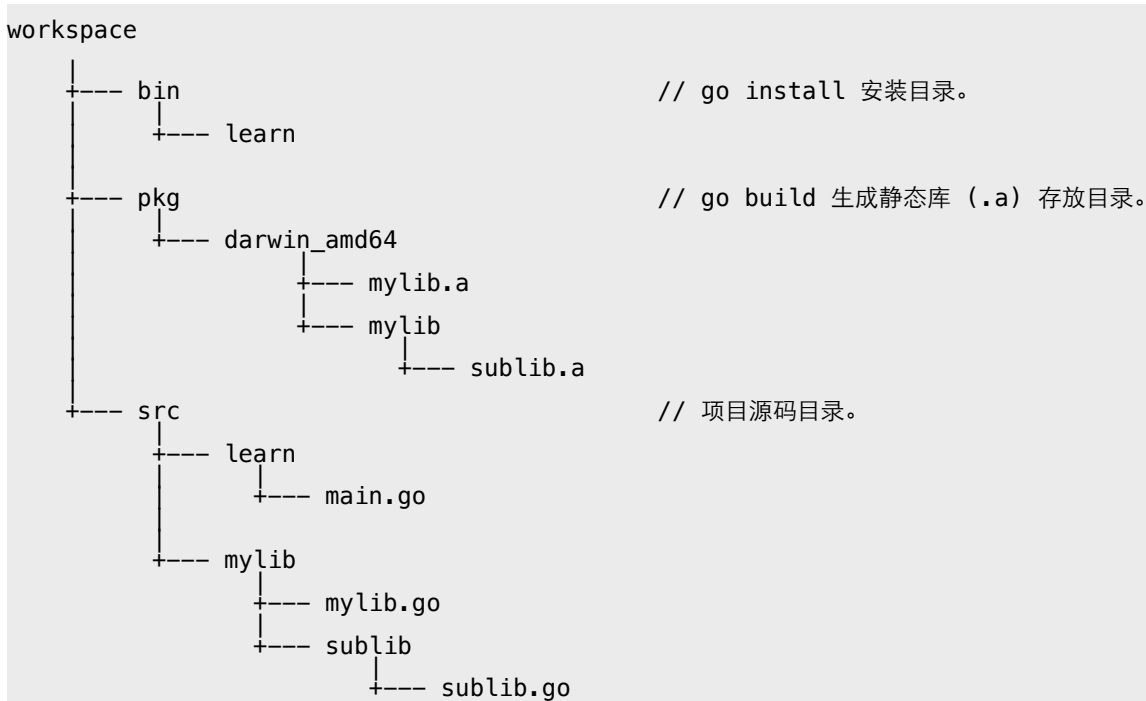
    req.ret <- x
}

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。



可在 `GOPATH` 环境变量列表中添加多个 `workspace`，但不能和 `GOROOT` 相同。

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

通常 `go get` 使用第一个 `workspace` 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 `///</code>、/**/ 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。
- 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。

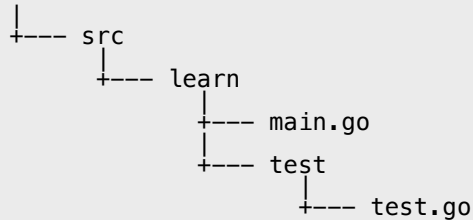
```
import    "yuheng/test"    // 默认模式：test.A
import M  "yuheng/test"    // 包重命名：M.A
import .  "yuheng/test"    // 简便模式：A
import _  "yuheng/test"    // 非导入模式：仅让该包执行初始化函数。
```

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



main.go

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

## 8.2.2 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

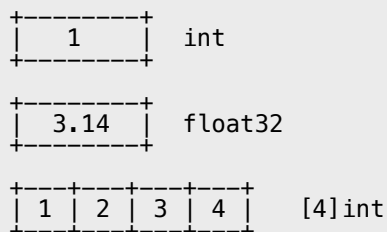
非测试源码文件中以 **BUG(author)** 开始的注释，会在帮助文档 **Bugs** 节点中显示。

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

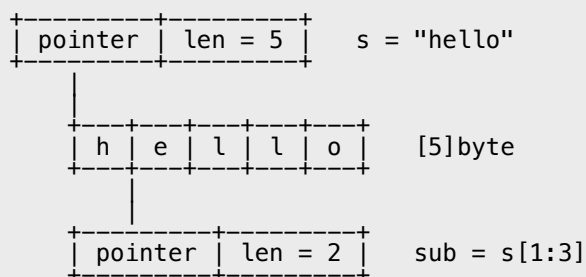
## 第 9 章 进阶

### 9.1 内存布局

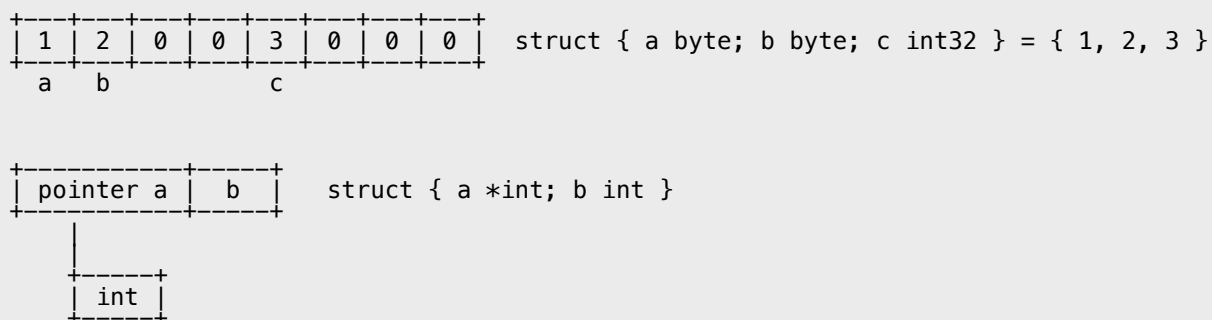
了解对象内存布局，有助于理解值传递、引用传递等概念。



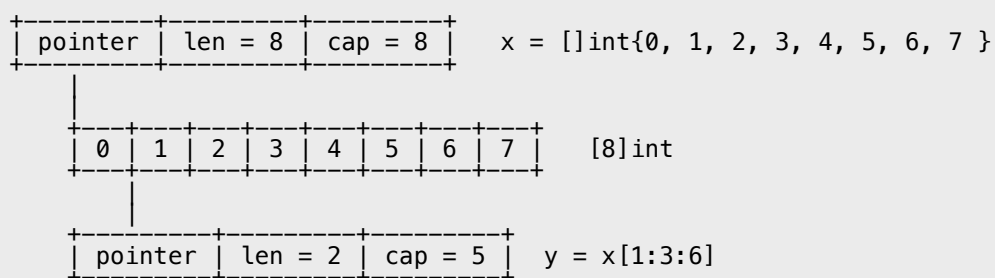
#### string



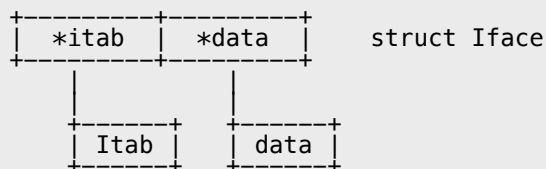
#### struct



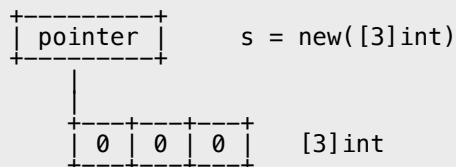
#### slice



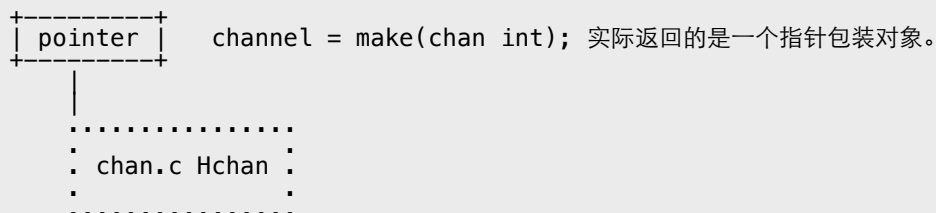
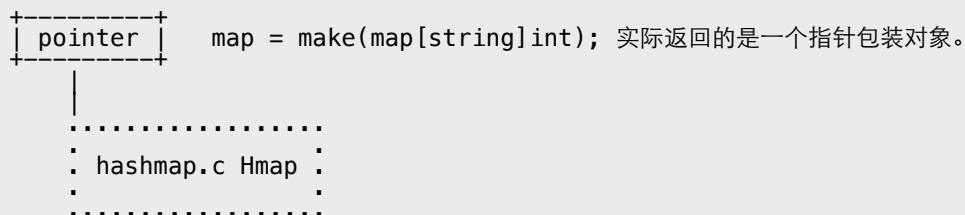
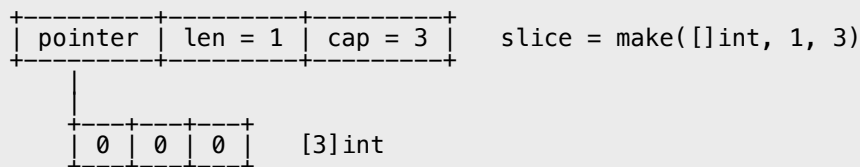
## interface



## 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** 代码很件很麻烦的事，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` 内部存储该与类型有关的信息，`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 1.3，相关文件位于 `src/pkg/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

有关 Go 内存分配器模型，在 `malloc.h` 头部注释中做了明确说明。

## `malloc.h`

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

// The main allocator works in runs of pages.
// Small allocation sizes (up to and including 32 kB) are
// rounded to one of about 100 size classes, each of which
// has its own free list of objects of exactly that size.
// Any free page of memory can be split into a set of objects
// of one size class, which are then managed using free list
// allocators.
//
// The allocator's data structures are:
//
//   FixAlloc: a free-list allocator for fixed-size objects,
//             used to manage storage used by the allocator.
//   MHeap: the malloc heap, managed at page (4096-byte) granularity.
//   MSpan: a run of pages managed by the MHeap.
//   MCentral: a shared free list for a given size class.
//   MCache: a per-thread (in Go, per-P) cache for small objects.
//   MStats: allocation statistics.
//
// Allocating a small object proceeds up a hierarchy of caches:
//
//   1. Round the size up to one of the small size classes
//      and look in the corresponding MCache free list.
//      If the list is not empty, allocate an object from it.
//      This can all be done without acquiring a lock.
//
//   2. If the MCache free list is empty, replenish it by
//      taking a bunch of objects from the MCentral free list.
//      Moving a bunch amortizes the cost of acquiring the MCentral lock.
//
//   3. If the MCentral free list is empty, replenish it by
//      allocating a run of pages from the MHeap and then
//      chopping that memory into a objects of the given size.
//      Allocating many objects amortizes the cost of locking
//      the heap.
//
//   4. If the MHeap is empty or has no page runs large enough,
//      allocate a new group of pages (at least 1MB) from the
//      operating system. Allocating a large run of pages
//      amortizes the cost of talking to the operating system.
```

```
//
// Freeing a small object proceeds up the same hierarchy:
//
// 1. Look up the size class for the object and add it to
//    the MCache free list.
//
// 2. If the MCache free list is too long or the MCache has
//    too much memory, return some to the MCentral free lists.
//
// 3. If all the objects in a given span have returned to
//    the MCentral list, return that span to the page heap.
//
// 4. If the heap has too much memory, return some to the
//    operating system.
//
// TODO(rsc): Step 4 is not implemented.
//
// Allocating and freeing a large object uses the page heap
// directly, bypassing the MCache and MCentral free lists.
```

原理也不算太复杂。

首先，分配器必然是以页 (page) 为单位向操作系统申请大块内存。这些大块内存被称为 span，由多个地址连续的页组成。所有申请的 span 都交由 heap 管理。

## malloc.h

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

其次，分配器以 32KB 为界，将对象分为大小两种。大对象直接从 heap 获取内存，这个无需多言。

## malloc.h

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

为数众多的小对象须进一步划分，以提高内存使用效率。于是又以 8 字节倍数为单位，将小对象分成多个等级 (size class)。例如，L1 为 1 ~ 8 字节，L2 为 9 ~ 16 字节，如此等等。当然，这种划分和计算方式未必是连续和固定的，会根据经验和测试进行调整。

## malloc.h

```
NumSizeClasses = 67,

// 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);
uintptr    runtime·roundupsize(uintptr);
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];
```

注：Go 1.3 不但将 page 增加到 8KB，还将等级调整到 67，其中 0 用于标记大对象。

组件 **central** 从 **heap** 获取 **span**，按照所属等级将大块内存切分成小块。每个小块存储一个相同等级对象，故称作 **object**。**heap** 管理多个 **central** 对象，每个 **central** 对应一种等级。

调度器为每个执行线程绑定一个 **cache**，用于小对象堆内存分配和回收。这么做的好处是无需加锁，大大提升了执行效率。**cache** 从 **central** 获取多个不同等级的 **span**，用以分配 **object** 内存。

一个优秀的内存分配器必然要平衡内存使用，既要用缓存快速分配，又不能有太多闲置造成浪费。**central** 除提供 **object** 储备外，还负责回收空闲空间，在多个 **cache** 间平衡使用。而当发现某 **span** 完整收回所属 **object**，则将其还给 **heap**，交由其他等级 **central** 使用（需重新切分）。如此就形成了两个层面的调度平衡。

最后，**heap** 会尝试合并相邻的空闲 **span**，以形成更大的内存块，减少碎片。

#### 分配流程:

1. 计算待分配对象所属等级。
2. 访问 **cache**，查找相同等级 **span**，从中提取 **object**。
3. 如目标 **span** 没有剩余空间，则向等级相同的 **central** 提取可用 **span**。
4. 如 **central** 没有可用 **span**，则从 **heap** 获取，并切分成 **object** 链表。
5. 当 **heap** 没有足够内存时，向操作系统申请新的 **span** 内存。

#### 释放流程:

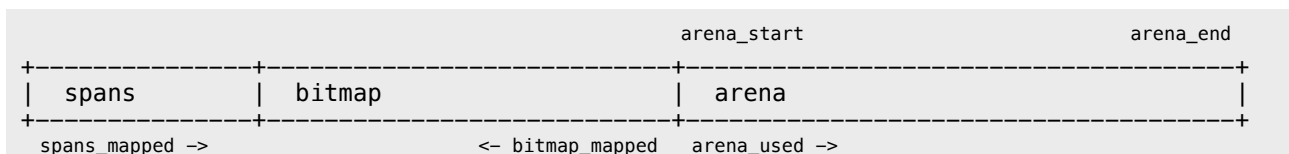
1. 计算待释放对象所属等级。
2. 将其归还给所属的 `cache span`，或添加到等级相同的可回收链表里。
3. 如可回收链表长度超过阈值，则将该链表内的 `object` 全部归还 `central`。
4. 当 `central` 中的 `span` 收回其全部 `object`，那么将该 `span` 还给 `heap`。
5. 垃圾回收器定时扫描 `heap span` 链表，并建议操作系统回收闲置的物理内存。

## 1.1 初始化

函数 `mallocinit` 除初始化 `object` 等级外，最重要的工作是设定保留虚拟地址范围。

预留内存地址包括三个部分：

- **arena**: 实际内存分配地址范围。
- **bitmap**: 为每个地址提供 4bit 标记位，用于垃圾回收等操作。
- **spans**: 记录每个 `page` 所对应的 `span` 地址，用于反查和合并操作。



### malloc.h

```
struct MHeap
{
    MSpan** spans;
    uintptr spans_mapped;
    byte    *bitmap;
    uintptr bitmap_mapped;
    byte    *arena_start;
    byte    *arena_used;    // 下一分配位置。
    byte    *arena_end;
    bool    arena_reserved;
};
```

### malloc.goc

```
void runtime·mallocinit(void)
{
    runtime·InitSizes();

    if(sizeof(void*) == 8 && (limit == 0 || limit > (1<<30))) {
```

```

// 计算各保留区域大小。
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

// 尝试保留从 0xc00000000 开始的内存地址, 如果失败则更换其他地址。
// 重试范围从 0x10c00000000 直到 0x7fc00000000。
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;
}

// 设置保留区域参数。
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。
runtime·MHeap_Init(&runtime·mheap);

// 测试分配器。
runtime·free(runtime·malloc(TinySize));
}

```

附: OSX/Darwin SysReserve 未使用 MAP\_FIXED 参数, 预留内存未必从 0xc00000000 开始。

初始化函数只是调用 mmap 预留虚拟内存地址, 并没有实际内存分配。因此, 就算你看到一个简单 "Hello, World!" 消耗上百 GB 内存, 亦无需担心。

之所以这么做, 是因为内存分配器算法依赖连续地址。当从操作系统分配 span 内存时, 用于管理的 bitmap、spans 同步线性增长。

还有一项任务就是创建 heap 对象, 这是整个内存分配器的根。

## malloc.h

```

struct MHeap
{
    MSpan free[MaxMHeapList]; // 以页数为单位的多个 span 链表。
}

```

```

MSpan freelarge;           // 页数超标的 span 链表。
MSpan busy[MaxMHeapList];  // 以页数为单位管理已分配大对象链表。
MSpan busylarge;           // 页数超标的已分配大对象链表。

// 用 size class 做索引的 central 数组。
struct {
    MCentral;
    byte pad[CacheLineSize];
} central[NumSizeClasses];
};

```

## mheap.c

```

void runtime·MHeap_Init(MHeap *h)
{
    // 初始化 span、cache 等 FixAlloc 分配器。
    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), ...);

    // 初始化 span 管理链表。
    for(i=0; i<nelem(h->free); i++) {
        runtime·MSpanList_Init(&h->free[i]);
        runtime·MSpanList_Init(&h->busy[i]);
    }

    // 初始化 large object 管理链表。
    runtime·MSpanList_Init(&h->freelarge);
    runtime·MSpanList_Init(&h->busylarge);

    // 初始化所有等级 central 对象。
    for(i=0; i<nelem(h->central); i++)
        runtime·MCentral_Init(&h->central[i], i);
}

```

函数初始化全局唯一 heap 对象：

1. 为 MSpan、MCache 等创建 FixAlloc 分配器；
2. 初始化用于 span 和大对象的管理链表；
3. 初始化 central 管理数组。

相比 Go 1.2，MHeap 多了 busy 链表，用于记录大对象内存分配，类似 LOH 的做法。



## 1.2 分配流程

函数 `runtime.new`、`runtime.malloc` 通过调用 `mallocgc` 完成对象堆内存分配。

### malloc.goc

```
void* runtime·mallocgc(uintptr size, uintptr typ, uint32 flag)
{
    c = m->mcache;

    // 小对象
    if(!runtime·debug·efence && size <= MaxSmallSize) {
        // 微小对象 (不包含指针, 小于 16 字节)
        if((flag&(FlagNoScan|FlagNoGC)) == FlagNoScan && size < TinySize) {
            tinysize = c->tinysize;

            // 如果 cache.tiny 还有足够空间。
            if(size <= tinysize) {
                tiny = c->tiny;

                // 指针对齐
                if((size&7) == 0)
                    tiny = (byte*)ROUND((uintptr)tiny, 8);
                else if((size&3) == 0)
                    tiny = (byte*)ROUND((uintptr)tiny, 4);
                else if((size&1) == 0)
                    tiny = (byte*)ROUND((uintptr)tiny, 2);
                size1 = size + (tiny - c->tiny);

                // 直接从 tiny 分配, 并调整 tiny 和 tinysize。
                if(size1 <= tinysize) {
                    v = (MLink*)tiny;
                    c->tiny += size1;
                    c->tinysize -= size1;
                    return v;
                }
            }

            // 如果 tiny 空间不足, 则分配新的 tiny 内存块。
            // 直接从 alloc 获取一块 class size 2 等级的内存。
            s = c->alloc[TinySizeClass];

            // 如果 cache 没有可用剩余空间, 则从 central 获取新的 span/objects。
            if(s->freelist == nil)
                s = runtime·MCache_Refill(c, TinySizeClass);

            // 从 span.freelist 列表提取 object, 并调整链表。
            v = s->freelist;
```

```

    next = v->next;
    s->freelist = next;
    s->ref++;

    // 如果 object 剩余空间大于现有 tiny 内存块, 那么就用大的替换掉。
    if(TinySize-size > tinysize) {
        c->tiny = (byte*)v + size;
        c->tinysize = TinySize - size;
    }

    size = TinySize;
    goto done;
}

// 普通小对象, 计算 size class。
if(size <= 1024-8)
    sizeclass = runtime.size_to_class8[(size+7)>>3];
else
    sizeclass = runtime.size_to_class128[(size-1024+127) >> 7];
size = runtime.class_to_size[sizeclass];

// 从 cache.alloc 链表数组中找到对应等级的 span。
s = c->alloc[sizeclass];
if(s->freelist == nil)
    s = runtime.MCache_Refill(c, sizeclass);

// 从 span.freelist 链表中提取可用的 object, 并调整链表。
v = s->freelist;
next = v->next;
s->freelist = next;
s->ref++;

done:
    c->local_cachealloc += size;
} else {
    // 大对象直接从 heap 分配内存。
    s = largealloc(flag, &size);
    v = (void*)(s->start << PageShift);
}

// 对内存做 GC 标记, 也就是前面提到过的 bitmap 区域。
if(flag & FlagNoGC)
    runtime.marknogc(v);
else if(!(flag & FlagNoScan))
    runtime.markscan(v);

// 检查 GC 回收阈值。
if(!(flag & FlagNoInvokeGC) && mstats.heap_alloc >= mstats.next_gc)

```

```

        runtime.gc(0);

    return v;
}

static MSpan* largealloc(uint32 flag, uintptr *sizep)
{
    // Allocate directly from heap.
    size = *sizep;
    npages = size >> PageShift;
    s = runtime.MHeap_Alloc(&runtime.mheap, npages, 0, 1, !(flag & FlagNoZero));
    return s;
}

```

相比以前版本，MCache 有两个明显的变化：

- 针对微小对象的优化，尝试将其组合到单个内存块，以提升内存利用率。
- 将空闲链表拆成分配和回收两部分，用于分配的 `alloc` 数组保存了从 `central` 获取的所有 `span`，而 `free` 数组则用来保存所有回收的 `object`。

## malloc.h

```

struct MCache
{
    // Allocator cache for tiny objects w/o pointers.
    byte*   tiny;
    uintptr tinysize;

    MSpan*   alloc[NumSizeClasses]; // 从 Central 获取的 Span，用于提取相应等级 object。
    MCacheList free[NumSizeClasses]; // 等待回收的 object 链表。
};

```

将分配和回收分开的好处是，分配操作集中在当前 `span`，当 `free` 链表返还给 `central` 时，就增加了以往 `span` 收回全部 `object` 的几率。

继续跟踪小对象内存分配过程，看看如何从 `central` 获取 `span`。

## mcache.c

```

MSpan* runtime.MCache_Refill(MCache *c, int32 sizeclass)
{
    // 按照 size class 返回 alloc 数组中对应的 span。
    s = c->alloc[sizeclass];

    // 既然要 refill，当前 span 自然没有剩余 object 可用。
    if(s->freelist != nil)

```

```

        runtime·throw("refill on a nonempty span");

// 将该 span 还给 central。
if(s != &emptyspan)
    runtime·MCentral_UncacheSpan(&runtime·mheap.central[sizeclass], s);

// 将 cache.free 链表中的 object 全部交还给 central。
l = &c->free[sizeclass];
if(l->nlist > 0) {
    runtime·MCentral_FreeList(&runtime·mheap.central[sizeclass], l->list);
    l->list = nil;
    l->nlist = 0;
}

// 从 central 获取一个可用的 span。
s = runtime·MCentral_CacheSpan(&runtime·mheap.central[sizeclass]);
c->alloc[sizeclass] = s;

return s;
}

```

在分配阶段提前接入回收操作，是 Go 1.3 的积极信号？

以等级为索引，就可以从 `heap.central` 数组找到目标对象。`central` 有两个 `span` 管理链表，`noempty` 中的 `span` 还有可用 `object` 空间，`empty` 则相反。

## malloc.h

```

struct MCentral
{
    Lock;
    int32 sizeclass;    // 所属等级。
    MSpan nonempty;     // 有剩余空间的 span 链表。
    MSpan empty;        // 没有剩余空间的 span 链表（被 cache 全部拿走，未必已经使用）
    int32 nfree;         // 所有有剩余空间 span.objects 数量。
};

```

## mcentral.c

```

MSpan* runtime·MCentral_CacheSpan(MCentral *c)
{
retry:
    // 从非空 span 链表提取可用 span。
    for(s = c->nonempty.next; s != &c->nonempty; s = s->next) {
        goto hasespan;
    }

    // 没有可用 span，则从 heap 获取。

```

```

    if(!MCentral_Grow(c)) {
        runtime·unlock(c);
        return nil;
    }
    goto retry;
}

havespan:
    // 计算 objects 数量, 并重置 cache.nfree 计数器。
    cap = (s->npages << PageShift) / s->elemsize;
    n = cap - s->ref;
    c->nfree -= n;

    // 将该 span 转移到 empty 链表。
    runtime·MSpanList_Remove(s);
    runtime·MSpanList_InsertBack(&c->empty, s);
    s->incache = true;

    return s;
}

```

从上面函数可以看出, 每次都拿走一整个 **span**。继续跟踪如何从 **heap** 获取 **span**。

## mcentral.c

```

static bool MCentral_Grow(MCentral *c)
{
    // 获取 central 相关信息。
    runtime·MGetSizeClassInfo(c->sizeclass, &size, &npages, &n);

    // 从 heap 获取 span。
    s = runtime·MHeap_Alloc(&runtime·mheap, npages, c->sizeclass, 0, 1);

    // 将 span 切分成 object, 保存到 freelist。
    tailp = &s->freelist;
    p = (byte*)(s->start << PageShift);
    s->limit = p + size*n;
    for(i=0; i<n; i++) {
        v = (MLink*)p;
        *tailp = v;
        tailp = &v->next;
        p += size;
    }
    *tailp = nil;

    // 增加 central 计数器, 并将 span 插入 noempty 链表。
    c->nfree += n;
    runtime·MSpanList_Insert(&c->nonempty, s);
}

```

```
    return true;
}
```

从 heap 拿到 span 后，按所属等级切分成 object 链表。

## malloc.h

```
struct MSpan
{
    MSpan *next;        // 链表
    MSpan *prev;
    PageID start;       // 开始页序号; PageID = pointer >> PageShift
    uintptr npages;     // 包含的页数
    MLink *freelist;    // 可用 object 链表
    uint16 ref;         // 已经使用 object 数量; capacity - freelist.num
    uint8 sizeclass;    // 等级
    bool incache;       // 是否正在被 cache 使用
    byte *limit;        // 结束地址
    MLink *freebuf;     // 等待回收的 object 链表
};
```

接下来的事情，就是看 heap 如何管理 span 了。

## malloc.h

```
struct MHeap
{
    Lock;
    MSpan free[MaxMHeapList]; // 以页数为单位的多个 span 链表。
    MSpan freelarge;         // 页数超标的 span 链表。
    MSpan busy[MaxMHeapList]; // 以页数为单位管理已分配大对象链表。
    MSpan busylarge;         // 页数超标的已分配大对象链表。
};
```

在 heap 里有个以页数为序号的数组 free，其中保存了多个 span 链表。只需用所需页数做索引就可以很方便取到所需 span。如果对应链表没有可用 span，那么就继续找页数更多的链表，直到从 freelarge 链表中取，或者向操作系统申请新内存空间。

成员 busy 和 busylarge 用于记录大对象内存分配。

## mheap.c

```
MSpan* runtime·MHeap_Alloc(MHeap *h, uintptr npage, int32 sizeclass, bool large, ...)
{
    MSpan *s;

    s = MHeap_AllocLocked(h, npage, sizeclass);
```

```

    if(s != nil) {
        // 为大对象分配内存, 记录到 busy[x] 或 busylarge 链表中。
        if(large) {
            if(s->npages < nelem(h->free))
                runtime·MSpanList_InsertBack(&h->busy[s->npages], s);
            else
                runtime·MSpanList_InsertBack(&h->busylarge, s);
        }
    }

    return s;
}

static MSpan* MHeap_AllocLocked(MHeap *h, uintptr npage, int32 sizeclass)
{
    // 从 free 链表数组中查找合适的 span。
    for(n=npage; n < nelem(h->free); n++) {
        if(!runtime·MSpanList_IsEmpty(&h->free[n])) {
            s = h->free[n].next;
            goto HaveSpan;
        }
    }

    // 从更大的链表中查找大小合适的 span。
    if((s = MHeap_AllocLarge(h, npage)) == nil) {
        if(!MHeap_Grow(h, npage))
            return nil;
        if((s = MHeap_AllocLarge(h, npage)) == nil)
            return nil;
    }

HaveSpan:
    // 从链表中移除。
    runtime·MSpanList_Remove(s);

    // 如果 span 大小超出预期。
    if(s->npages > npage) {
        // 截断尾部多余空间, 新建 span。
        t = runtime·FixAlloc_Alloc(&h->spanalloc);
        runtime·MSpan_Init(t, s->start + npage, s->npages - npage);
        s->npages = npage;

        // 在 heap.spans 记录新 span 地址。
        p = t->start;
        p -= ((uintptr)h->arena_start >> PageShift);
        if(p > 0)
            h->spans[p-1] = s;
        h->spans[p] = t;
    }
}

```

```

        h->spans[p+t->npages-1] = t;

        // 将新 span 重新放回 heap 链表。
        MHeap_FreeLocked(h, t);
    }

    // 在 heap.spans 保存 span 地址。
    s->sizeclass = sizeclass;
    p = s->start;
    p -= ((uintptr)h->arena_start >> PageShift);
    for(n=0; n<npage; n++)
        h->spans[p+n] = s;

    return s;
}

```

对于所获取的 **span**，分配器会截取多余部分，重新放回链表。在此期间，还会尝试合并相邻 **span**，以形成更大块可用内存空间，减少碎片。

## mheap.c

```

static void MHeap_FreeLocked(MHeap *h, MSpan *s)
{
    // 将 span 从现有链表中移除。
    runtime·MSpanList_Remove(s);

    p = s->start;
    p -= (uintptr)h->arena_start >> PageShift;

    // 通过 heap.spans 检查左侧 span 是否可合并。
    if(p > 0 && (t = h->spans[p-1]) != nil && t->state != MSpanInUse) {
        // 合并状态参数。
        s->start = t->start;
        s->npages += t->npages;
        p -= t->npages;
        h->spans[p] = s;

        // 将左侧 span 管理对象从链表移除并释放。
        runtime·MSpanList_Remove(t);
        runtime·FixAlloc_Free(&h->spanalloc, t);
    }

    // 检查右侧 span 是否可合并。
    if((p+s->npages)*sizeof(h->spans[0]) < h->spans_mapped &&
        (t = h->spans[p+s->npages]) != nil && t->state != MSpanInUse) {
        // 合并参数。
        s->npages += t->npages;
        h->spans[p + s->npages - 1] = s;
    }
}

```



```

        // 将右侧 span 管理对象从链表移除并释放。
        runtime·MSpanList_Remove(t);
        runtime·FixAlloc_Free(&h->spanalloc, t);
    }

    // 根据 span 页数, 将其插入到合适的 free 链表中。
    if(s->npages < nelem(h->free))
        runtime·MSpanList_Insert(&h->free[s->npages], s);
    else
        runtime·MSpanList_Insert(&h->freelarge, s);
}

```

不管如何操作, 最终都需要向操作系统申请内存。

## mheap.c

```

static bool MHeap_Grow(MHeap *h, uintptr npage)
{
    // 每次申请的内存是 64KB 的倍数, 最少 1MB。
    npage = (npage+15)&~15;
    ask = npage<<PageShift;
    if(ask < HeapAllocChunk) ask = HeapAllocChunk;

    v = runtime·MHeap_SysAlloc(h, ask);

    // 创建新的 span 保存刚申请的内存块。
    s = runtime·FixAlloc_Alloc(&h->spanalloc);
    runtime·MSpan_Init(s, (uintptr)v>>PageShift, ask>>PageShift);

    // 在 heap.spans 记录该 span 地址。
    p = s->start;
    p -= ((uintptr)h->arena_start>>PageShift);
    h->spans[p] = s;
    h->spans[p + s->npages - 1] = s;

    // 尝试合并相邻 span, 并插入到合适的链表中。
    MHeap_FreeLocked(h, s);
    return true;
}

```

每次申请都须确保在 arena 范围内, 并指定起始内存地址 (arena\_used)。

## malloc.goc

```

void* runtime·MHeap_SysAlloc(MHeap *h, uintptr n)
{
    // 检查是否超出 arena 保留范围。

```

```

    if(n <= h->arena_end - h->arena_used) {
        // 从 arena_used 处申请内存。
        p = h->arena_used;
        runtime·SysMap(p, n, h->arena_reserved, &mstats.heap_sys);

        // 调整下次申请地址。
        h->arena_used += n;

        // 同步为 spans、bitmap 增加内存。
        runtime·MHeap_MapBits(h);
        runtime·MHeap_MapSpans(h);

        return p;
    }
}

```

最后调用 `mmap`、`VirtualAlloc` 完成向操作系统申请内存操作。

#### 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\_windows.c

```

void runtime·SysMap(void *v, uintptr n, bool reserved, uint64 *stat)
{
    p = runtime·stdcall(runtime·VirtualAlloc, 4, v, n, (uintptr)MEM_COMMIT,
        (uintptr)PAGE_READWRITE);
}

```

## 1.3 释放流程

调用 `free` 函数释放 object 内存。

#### malloc.goc

```

void runtime·free(void *v)
{
    // 查找 object 所在 span。
    if(!runtime·mlookup(v, nil, nil, &s)) {
        runtime·throw("free runtime·mlookup");
    }

    size = s->elemsize;
}

```

```

sizeclass = s->sizeclass;

c = m->mcache;

// 大对象
if(sizeclass == 0) {
    // 直接归还给 heap。
    if(runtime·debug·efence)
        runtime·SysFault((void*)(s->start<<PageShift), size);
    else
        runtime·MHeap_Free(&runtime·mheap, s, 1);
} else {
    // 小对象

    // 如果该 object 属于当前 alloc span, 那么就归还到 freelist。
    if(c->alloc[sizeclass] == s) {
        runtime·markfreed(v);
        ((MLink*)v)->next = s->freelist;
        s->freelist = v;
        s->ref--;
    } else {
        // 否则添加到 cache.free 链表。
        runtime·MCache_Free(c, v, sizeclass, size);
    }
}
}
}

```

大对象回归 heap，小对象要检查是否属于当前正在使用的 span。是就复用，不是就交给 free 链表，随后还给 central。

## mcache.c

```

void runtime·MCache_Free(MCache *c, MLink *p, int32 sizeclass, uintptr size)
{
    // 将 object 添加到对应的 free 链表。
    l = &c->free[sizeclass];
    p->next = l->list;
    l->list = p;
    l->nlist++;

    // 如果 free 链表长度超出阈值，则将其归还给 central。
    if(l->nlist >= (runtime·class_to_allocnpages[sizeclass]<<PageShift)/size) {
        runtime·MCentral_FreeList(&runtime·mheap·central[sizeclass], l->list);
        l->list = nil;
        l->nlist = 0;
    }
}

```

函数 `MCentral_FreeList` 将等待回收链表中的 `object` 逐个添加到所属 `span.freebuf` 或 `freelist` 链表，然后检查 `span` 是否可以归还给 `heap`。

### mcentral.c

```
void runtime·MCentral_FreeList(MCentral *c, MLink *start)
{
    // 遍历链表，逐个释放 object。
    for(; start != nil; start = next) {
        next = start->next;
        MCentral_Free(c, start);
    }
}

static void MCentral_Free(MCentral *c, MLink *v)
{
    // 查找该 object 所在 span。(用保留区域的 spans 查找)
    s = runtime·MHeap_Lookup(&runtime·mheap, v);

    // 如果该 span 还在被 cache 使用，那么添加到 span.freebuf。
    if(s->incache) {
        v->next = s->freebuf;
        s->freebuf = v;
        return;
    }

    // 显然该 span 即将非空，转移到 noempty 链表。
    if(s->freelist == nil) {
        runtime·MSpanList_Remove(s);
        runtime·MSpanList_Insert(&c->nonempty, s);
    }

    // 将该 object 添加到 span.freelist。
    v->next = s->freelist;
    s->freelist = v;
    s->ref--;
    c->nfree++;

    // 检查 span.ref 计数器，如果收回全部 object，立即归还给 heap。
    if(s->ref == 0) {
        MCentral_ReturnToHeap(c, s); // unlocks c
    }
}
```

如果 `span` 还在被 `cache` 使用，那么将回收的 `object` 暂存在 `freebuf` 链表中，等交还给 `central` 时，再转移到 `freelist`。

在 `MCache_Refill` 函数获取新 `span` 前, 会调用 `MCentral_UncacheSpan`, 将消耗一空的 `span` 交还给 `central`。

## mcentral.c

```
void runtime·MCentral_UncacheSpan(MCentral *c, MSpan *s)
{
    s->incache = false;

    // 将 freebuf 中的 object 转移到 freelist。
    while((v = s->freebuf) != nil) {
        s->freebuf = v->next;
        v->next = s->freelist;
        s->freelist = v;
        s->ref--;
    }

    // 如果收回全部 object, 将 span 归还给 heap。
    if(s->ref == 0) {
        MCentral_ReturnToHeap(c, s);
        return;
    }

    // 将 span 转移到 central.noempty 链表。
    cap = (s->npages << PageShift) / s->elemsize;
    n = cap - s->ref;
    if(n > 0) {
        c->nfree += n;
        runtime·MSpanList_Remove(s);
        runtime·MSpanList_Insert(&c->nonempty, s);
    }
}
```

最终将 `span` 归还到 `heap` 链表中, 并没有实际释放。物理内存释放是由垃圾回收器的一个特殊定时操作完成的。

## mcentral.c

```
static void MCentral_ReturnToHeap(MCentral *c, MSpan *s)
{
    // 将 span 从 central 链表中移除。
    runtime·MSpanList_Remove(s);

    // 归还给 heap。
    runtime·MHeap_Free(&runtime·mheap, s, 0);
}
```

## mheap.c

```
void runtime·MHeap_Free(MHeap *h, MSpan *s, int32 acct)
{
    // 合并相邻 span, 然后插入到合适的链表中。
    MHeap_FreeLocked(h, s);
}
```

## 1.4 其他

在内存分配过程中, 除普通对象外, 还有些管理对象, 比如 MSpan、MCache 等等。这些管理对象会被缓存, 重复使用。如果也从 heap 分配, 必然会导致所在内存块被长期占用, 造成碎片化, 不利于回收和合并。

为此, 专门为它们额外准备了分配器。因同一种类型长度固定, 故称作 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。
};
```

在 MHeap\_Init 函数中, 专门创建所需的各种 FixAlloc 分配器。

## mheap.c

```
void runtime·MHeap_Init(MHeap *h)
{
    runtime·FixAlloc_Init(&h->spanalloc, sizeof(MSpan), RecordSpan, ...);
    runtime·FixAlloc_Init(&h->cachealloc, sizeof(MCache), nil, ...);
}
```

## 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;
}

```

注意 `Span FixAlloc` 参数 `RecordSpan` 的作用是为 `heap.allspans` 分配内存, 用于存储所有 `span` 指针, `GC Sweep` 和 `Heap Dump` 操作都会用到该数据。

## mheap.c

```

static void RecordSpan(void *vh, byte *p)
{
    MHeap *h;
    MSpan *s;
    MSpan **all;

    h = vh;
    s = (MSpan*)p;

    // 如果现有内存不足。
    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;

        // 重新分配内存。
        all = (MSpan**)runtime.SysAlloc(cap*sizeof(all[0]), &mstats.other_sys);

        // 将原来数据转移过来。
        if(h->allspans) {
            runtime.memmove(all, h->allspans, h->nspancap*sizeof(all[0]));

            // 释放原来的内存。
            if(h->allspans != runtime.mheap.sweepspans)
                runtime.SysFree(h->allspans, h->nspancap*sizeof(all[0]), ...);
        }

        h->allspans = all;
        h->nspancap = cap;
    }

    // 添加新 span。
    h->allspans[h->nspan++] = s;
}

```

回到正题，以 MSpan 为例，看看 FixAlloc 具体如何工作。

## mheap.c

```
static bool MHeap_Grow(MHeap *h, uintptr npage)
{
    // 用 Span FixAlloc 创建新的 Span 对象。
    s = runtime·FixAlloc_Alloc(&h->spanalloc);
    runtime·MSpan_Init(s, (uintptr)v>>PageShift, ask>>PageShift);
}
```

## mfixalloc.c

```
void* runtime·FixAlloc_Alloc(FixAlloc *f)
{
    // 如果空闲链表不为空，直接返回复用空间。
    if(f->list) {
        v = f->list;
        f->list = *(void**)f->list;
        f->inuse += f->size;
        return v;
    }

    // 如果后备内存块可用空间不足。
    if(f->nchunk < f->size) {
        // 重新申请 16KB 内存。
        f->chunk = runtime·persistentalloc(FixAllocChunk, 0, f->stat);
        f->nchunk = FixAllocChunk;
    }

    // 从后备内存块头部提取空间。
    v = f->chunk;

    // 执行 first 函数，比如 RecordSpan。
    if(f->first)
        f->first(f->arg, v);

    // 调整后备内存块的开始位置和可用大小。
    f->chunk += f->size;
    f->nchunk -= f->size;
    f->inuse += f->size;

    return v;
}
```

很简单的一个内存池算法。除一个复用链表外，还有后备大块内存用于切分操作。



**FixAlloc** 并没有使用 **arena** 区域, 而是用 **mmap** 额外申请。另外, 为减少系统调用, 还有个全局 **persistent** 对象, 每次申请 256KB 大块内存用于切分, 总体上和 **MHeap** 体系类似。

## malloc.goc

```
void* runtime·persistentalloc(uintptr size, uintptr align, uint64 *stat)
{
    // 大于 64KB 的直接用 mmap 分配。
    if(size >= PersistentAllocMaxBlock)
        return runtime·SysAlloc(size, stat);

    // 如果 persistent 剩余空间不足, 则重新分配 256KB。
    if(persistent.pos + size > persistent.end) {
        persistent.pos = runtime·SysAlloc(PersistentAllocChunk, &mstats.other_sys);
        persistent.end = persistent.pos + PersistentAllocChunk;
    }

    // 从 persistent 分配内存。
    p = persistent.pos;
    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;
}
```

释放操作也仅仅是将其添加到复用链表。

## mfixalloc.c

```
void runtime·FixAlloc_Free(FixAlloc *f, void *p)
{
    f->inuse -= f->size;
    *(void**)p = f->list;
    f->list = p;
}
```

## 2. Garbage Collector

精确垃圾回收，很经典的 Mark-and-Sweep 算法。

当内存分配 (不包括缓存和未使用的 `span`) 超过预设阈值，就会引发垃圾回收操作。在回收前，需要暂停用户逻辑执行，也就是所谓的 `StopTheWorld`。然后启用多个线程从根对象开始并行扫描，直到标记出所有可回收对象。

1.3 对垃圾回收做了改进。在标记结束后，会立即恢复逻辑执行 (`StartTheWorld`)。创建或唤醒专门的 `goroutine` 去并发完成清理工作，而非以前那样与标记串行。这缩短了暂停时间，一定程度上改善了垃圾回收所引发的问题。

在完成清理后，新的阈值通常是存活对象所使用内存的 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，将内存逐步交还给 `heap`，并未释放物理内存。

在入口函数，运行时专门创建一个 `goroutine` 定期执行与内存回收有关的操作。循环检查最后垃圾回收时间，如超出 2 分钟，则执行强制回收。另建议操作系统收回那些闲置超过 5 分钟的 `span` 物理内存。

### 2.1 垃圾回收

前面提到过，在 `mallocgc` 函数中会检查已分配内存是否超过垃圾回收阈值。

#### malloc.goc

```
void* runtime·mallocgc(uintptr size, uintptr typ, uint32 flag)
{
    if(!(flag & FlagNoInvokeGC) && mstats.heap_alloc >= mstats.next_gc)
        runtime·gc(0);
}
```

#### mgc0.c

```
void runtime·gc(int32 force)
{
    // 检查 GOGC 值。
    if(gcpercent == GcpercentUnknown) {
        // 默认 100。
        if(gcpercent == GcpercentUnknown)
            gcpercent = readgogc();
    }
}
```

```

// GOGC == 0, 禁用垃圾回收。
if(gcpercent < 0)
    return;

// 如果没达到阈值, 那么下次再说。
if(!force && mstats.heap_alloc < mstats.next_gc) {
    return;
}

// 没办法, 先暂停。
runtime·stoptheworld();

// 如果 GODEBUG="gotrace=2", 会引发两次回收。
for(i = 0; i < (runtime·debug.gctrace > 1 ? 2 : 1); i++) {
    runtime·mcall(mgc);
}

// 恢复执行。
runtime·starttheworld();
}

```

在 1.3 的垃圾回收器中, 有两个重要改进:

- 清理操作默认被放到专门的后台 **goroutine** 并发执行。
- 引入代龄机制, 减少对 **heap.allspans** 内存块的扫描操作。

## mgc0.c

```

static void mgc(G *gp)
{
    gc(gp->param);
}

static void gc(struct gc_args *args)
{
    if(work.markfor == nil)
        work.markfor = runtime·parforalloc(MaxGcproc);

    // 非并发清理模式。
    while(runtime·sweepone() != -1)
        gcstats.npausesweep++;

    // 执行并行标记操作。
    work.nproc = runtime·gcprocs();
    runtime·parforsetup(work.markfor, work.nproc, RootCount + runtime·allglen,
        nil, false, markroot);
}

```

```

runtime·parfordo(work.markfor);

// 因为清理过程由后台 goroutine 处理, 只能先预估个值, 现在分配空间的 2 倍。
mstats.next_gc = mstats.heap_alloc+mstats.heap_alloc*gcpercent/100;

// 设置 sweep 对象清理状态。
runtime·mheap.sweepspans = runtime·mheap.allspans;
runtime·mheap.sweepgen += 2; // 增加代龄计数。
runtime·mheap.sweepdone = false;
sweep.spans = runtime·mheap.allspans;
sweep.nspan = runtime·mheap.nspan;
sweep.spanidx = 0;

// 如果是并发清理 (mgc0.c 头部定义的常量),
if(ConcurrentSweep && !args->eagersweep) {
    if(sweep.g == nil)
        // 如果是第一次, 创建专用 goroutine 用于后台并发清理操作。
        sweep.g = runtime·newproc1(&bgsweepv, nil, 0, 0, runtime·gc);
    else if(sweep.parked) {
        // 唤醒 goroutine 清理垃圾。
        sweep.parked = false;
        runtime·ready(sweep.g);
    }
} else {
    // 非并发模式, 立即执行清理操作。
    while(runtime·sweepone() != -1)
        gcstats.npausesweep++;
}

// 收缩所有 goroutine 栈空间。
for(i = 0; i < runtime·allglen; i++)
    runtime·shrinkstack(runtime·allg[i]);
}

```

用于垃圾的清理的 goroutine 被唤醒后会扫描 heap.allspans 列表, 执行清理操作, 完成后休眠直到再次被唤醒。

与常见固定 3 级代龄机制不同, Go 1.3 的代龄会逐步提升。为 span 引入代龄, 可避免每次都扫描全部 span 内存块, 减少操作时间。

## mgc0.c

```

static FuncVal bgsweepv = {bgsweep};

static void bgsweep(void)
{
    for(;;) {

```

```

// 清理垃圾。
while(runtime.sweepone() != -1) {
    runtime.gosched();
}

// 如果期间再次触发过 gc, 那就继续清理。
if(!runtime.mheap.sweepdone) {
    continue;
}

// 全部搞定, 那就暂停该 goroutine。
sweep.parked = true;
runtime.parkunlock(&gclock, "GC sweep wait");
}
}

uintptr runtime.sweepone(void)
{
    // 当前代龄。
    sg = runtime.mheap.sweepgen;

    // 循环处理所有 span。
    for(;;) {
        idx = runtime.xadd(&sweep.spanidx, 1) - 1;

        // 跳出循环, 标记结束。
        if(idx >= sweep.nspan) {
            runtime.mheap.sweepdone = true;
            return -1;
        }

        s = sweep.spans[idx];

        // 如果该 span 未被使用, 提升代龄。
        if(s->state != MSpanInUse) {
            s->sweepgen = sg;
            continue;
        }

        // 检查 span 代龄, 避免过于频繁的扫描。
        // 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
        if(s->sweepgen != sg-2 || !runtime.cas(&s->sweepgen, sg-2, sg-1))
            continue;

        // 清理 span, 返回页数。
    }
}

```

```

    npages = s->npages;
    if(!runtime·MSpan_Sweep(s))
        npages = 0;

    return npages;
}
}

```

清理 `span` 的操作，实际就是调用内存分配器的回收过程。

## mgc0.c

```

bool runtime·MSpan_Sweep(MSpan *s)
{
    MLink head, *end;

    cl = s->sizeclass;
    size = s->elemsize;
    end = &head;

    p = (byte*)(s->start << PageShift);

    // 循环处理该 span 上所有 object 块。
    for(; n > 0; n--, p += size, type_data+=type_data_inc) {
        // 检查该 object 在 bitmap 里的标记位，以确定是否可回收。
        off = (uintptr*)p - (uintptr*)arena_start;
        bitp = (uintptr*)arena_start - off/wordsPerBitmapWord - 1;
        shift = off % wordsPerBitmapWord;
        bits = *bitp>>shift;

        if((bits & bitAllocated) == 0)
            continue;

        if((bits & bitMarked) != 0) {
            *bitp &= ~(bitMarked<<shift);
            continue;
        }

        // 根据 size class 分别处理。
        if(cl == 0) {
            // 大对象还给 heap, 调整 next_gc 阈值。
            runtime·MHeap_Free(&runtime·mheap, s, 1);
            runtime·xadd64(&mstats.next_gc, -(uint64)(size * (gcpercent + 100)/100));
            res = true;
        } else {
            // 将可回收的 small object 添加到一个链表。
            end->next = (MLink*)p;
            end = (MLink*)p;
        }
    }
}

```

```

        nfree++;
    }
}

if(nfree > 0) {
    // 调整 next_gc 阈值。
    runtime·xadd64(&mstats.next_gc,
        -(uint64)(nfree * size * (gcpercent + 100)/100));

    // 清理可回收链表。
    res = runtime·MCentral_FreeSpan(&runtime·mheap.central[cl], s, nfree,
        head.next, end);
}

return res;
}

```

## mcentral.c

```

bool runtime·MCentral_FreeSpan(MCentral *c, MSpan *s, int32 n, MLink *start, MLink *end)
{
    // 将该 span 转移到 central.noempty 链表。
    if(s->freelist == nil) {
        runtime·MSpanList_Remove(s);
        runtime·MSpanList_Insert(&c->nonempty, s);
    }

    // 直接将回收链表合并到 span.freelist。
    end->next = s->freelist;
    s->freelist = start;
    s->ref -= n;
    c->nfree += n;

    // 如果还有 object 在使用，那么到此结束。
    if(s->ref != 0) {
        return false;
    }

    // 如果已收回全部 object，将其归还给 heap。
    MCentral_ReturnToHeap(c, s); // unlocks c
    return true;
}

```

本节忽略了标记算法细节，有兴趣的可以自行查阅源码。

## 2.2 内存释放

在进程入口，运行时专门启动一个 **goroutine** 进行强制垃圾回收和物理内存释放。

### proc.c

```
void runtime·main(void)
{
    runtime·newproc1(&scavenger, nil, 0, 0, runtime·main);
}

static FuncVal scavenger = {runtime·MHeap_Scavenger};
```

这个 **goroutine** 会一直循环运行，定时执行相关操作。

### mheap.c

```
void runtime·MHeap_Scavenger(void)
{
    // 1: 如果超过 2 分钟没有做过垃圾回收，强制执行。
    forcegc = 2*60*1e9;

    // 2: 如果某 span 闲置超过 5 分钟，那么将其内存还给操作系统。
    limit = 5*60*1e9;

    // 计算循环间隔，显然是 1 分钟。
    if(forcegc < limit)
        tick = forcegc/2;
    else
        tick = limit/2;

    h = &runtime·mheap;
    for(k=0;; k++) {
        // 暂停 1 分钟。
        runtime·notetsleepg(&note, tick);
        now = runtime·nanotime();

        // 检查距离上次垃圾回收是否已超过 2 分钟。
        if(now - mstats.last_gc > forcegc) {
            // 新建 goroutine，开始强制回收。
            runtime·newproc1(&forcegc·helperv, (byte*)&notep, sizeof(notep), 0,
                            runtime·MHeap_Scavenger);
            now = runtime·nanotime();
        }

        // 检查闲置 span，归还内存。
        scavenge(k, now, limit);
    }
}
```



```
}
```

假设本次回收后已分配内存是 1GB，那么 `next_gc` 阈值就是 2GB，可能导致很长时间无法触发垃圾回收，这显然是个很大的负担。所以，定期执行强制回收是非常有必要的。

## mheap.c

```
static FuncVal forcegchelperv = {(void*)(void))forcegchelper};

static void forcegchelper(Note *note)
{
    runtime.gc(1);
}
```

回收操作上节已经分析过，我们回到物理内存释放操作上来。

## mheap.c

```
static void scavenge(int32 k, uint64 now, uint64 limit)
{
    h = &runtime.mheap;

    // 遍历 heap.free 和 freelarge 链表。
    for(i=0; i < nelem(h->free); i++)
        sumreleased += scavengelist(&h->free[i], now, limit);
    sumreleased += scavengelist(&h->freelarge, now, limit);
}

static uintptr scavengelist(MSpan *list, uint64 now, uint64 limit)
{
    // 遍历链表里的所有 span。
    for(s=list->next; s != list; s=s->next) {
        // 如果该 span 闲置时间超过 5 分钟，而且内存未被释放过。
        if((now - s->unusedsince) > limit && s->npreleased != s->npages) {
            // 标记 span 释放的页数。
            s->npreleased = s->npages;
            // 释放 span 所管理的内存。
            runtime.SysUnused((void*)(s->start << PageShift), s->npages << PageShift);
        }
    }
}
```

遍历 `heap` 所有可分配的 `span` 链表，然后依次检查 `span` 是否闲置过久，是否已经释放过。如果可以，就建议操作系统收回其管理的内存。

## mem\_linux.c

```
void runtime·SysUnused(void *v, uintptr n)
{
    runtime·madvise(v, n, MADV_DONTNEED);
}
```

### mem\_windows.c

```
void runtime·SysUnused(void *v, uintptr n)
{
    r = runtime·stdcall(runtime·VirtualFree, 3, v, n, (uintptr)MEM_DECOMMIT);
}
```

对 Linux 而言，**MADV\_DONTNEED** 实际就是告诉操作系统，这段物理内存暂时不用，可解除映射。但对应的虚拟内存并未释放，下次使用时，由操作系统再次建立物理内存映射。

不过 Windows 有些差别，释放后必须在重新分配。当用 **MHeap\_AllocLocked** 函数从 heap 获取 span 时，会检查 **npreleased** 释放标记，以便重新分配内存。

### mheap.c

```
static MSpan* MHeap_AllocLocked(MHeap *h, uintptr npage, int32 sizeclass)
{
    if(s->npreleased > 0)
        runtime·SysUsed((void*)(s->start<<PageShift), s->npages<<PageShift);
}
```

### mem\_windows.c

```
void runtime·SysUsed(void *v, uintptr n)
{
    r = runtime·stdcall(runtime·VirtualAlloc, 4, v, n, (uintptr)MEM_COMMIT,
        (uintptr)PAGE_READWRITE);
}
```

### mem\_linux.c

```
void runtime·SysUsed(void *v, uintptr n)
{
    USED(v);
    USED(n);
}
```

除了后台自动回收，我们也可以手工调用 **debug/FreeOSMemory** 来释放物理内存。

### mheap.c

```
void runtime/debug·freeOSMemory(void)
```

```
{
    runtime.gc(1);
    runtime.lock(&runtime.mheap);
    scavenge(-1, ~(uintptr)0, 0);    // scavenge(-1, 18446744073709551615, 0)
    runtime.unlock(&runtime.mheap);
}
```

## 2.3 状态输出

使用环境变量 `GODEBUG="gotrace=1"` 可输出 GC 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。

先了解有关的状态对象（使用 `runtime.ReadMemStats` 函数可读取该信息）。

### malloc.h

```
struct MStats
{
    // General statistics.
    uint64  alloc;           // 当前正在使用 object 容量。
    uint64  total_alloc;     // 历史分配总量（包括已被释放的内存）。
    uint64  sys;             // 当前消耗的所有内存总量。
    uint64  malloc;          // 分配操作总次数。
    uint64  nfree;           // 释放操作总次数。

    // Statistics about malloc heap.
    uint64  heap_alloc;      // 正在使用的 object 容量。
    uint64  heap_sys;        // 当前分配的虚拟内存总量（mmap-munmap）。
    uint64  heap_idle;       // 空闲 spans 容量。
    uint64  heap_inuse;      // 正在使用的 spans 容量。
    uint64  heap_released;   // 归还给 OS 的内存容量。
    uint64  heap_objects;    // 当前正在使用的 object 数量。

    // Statistics about garbage collector.
    uint64  next_gc;         // 下次回收阈值。
    uint64  last_gc;         // 上次回收时间。
    uint32  numgc;           // 回收次数。
};
```

在执行垃圾回收操作时，相关统计数据被更新。

### mgc0.c

```
void runtime.updatememstats(GCStats *stats)
{
```

```

// 统计 stack、cache、span 等。
mstats.stacks_inuse = stacks_inuse;
mstats.mcache_inuse = runtime·mheap.cachealloc.inuse;
mstats.mspan_inuse = runtime·mheap.spanalloc.inuse;

// 所有内存开销总和。
mstats.sys = mstats.heap_sys + mstats.stacks_sys + mstats.mspan_sys +
    mstats.mcache_sys + mstats.buckhash_sys + mstats.gc_sys + mstats.other_sys;

// 状态清零。
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 cache 闲置 object 交还给 central。
flushallmcaches();

// 更新 cache 状态。
cachestats();

// 扫描所有 span, 统计正在使用 object 信息。
for(i = 0; i < runtime·mheap.nspan; i++) {
    s = runtime·mheap.allspans[i];
    if(s->state != MSpanInUse)
        continue;

    if(s->sizeclass == 0) { // 大对象
        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;
    }
}

// 按等级分类统计。
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];
    }

    // nfree = 大对象总释放次数 + 各等级小对象释放次数总和
    // nmalloc = 正在使用 object 数量 + 总释放次数
    mstats.nmalloc += mstats.nfree;

    // 累计分配容量 = 正在使用 object 容量 + 大对象释放容量 + 各类小对象释放容量, 历史数据总和。
    mstats.total_alloc = mstats.alloc + runtime·mheap.largefree + smallfree;

    // 当前正在使用 object 容量。
    mstats.heap_alloc = mstats.alloc;

    // 当前正在使用 object 数量。
    mstats.heap_objects = mstats.nmalloc - mstats.nfree;
}

```

要监控的第一种信息来自垃圾回收函数。

## mgc0.c

```

static void gc(struct gc_args *args)
{
    t0 = args->start_time;

    // 从 stoptheworld 开始的各种准备时间 ...

    t1 = runtime·nanotime();

    // 非并发模式清理。
    // 准备并行标记设置。

    t2 = runtime·nanotime();

    // 并行标记。

    t3 = runtime·nanotime();

    // 等待并行任务结束 ...
    // 更新全部 cache 状态。
    // 预估 next_gc。

    heap0 = mstats.next_gc*100/(gcpercent+100);
    mstats.next_gc = mstats.heap_alloc+mstats.heap_alloc*gcpercent/100;

    t4 = runtime·nanotime();

    mstats.last_gc = t4;
}

```

```

mstats.numgc++;

if(runtime.debug.gctrace) {
    heap1 = mstats.heap_alloc;
    runtime.updateMemstats(&stats);

    obj = mstats.nmalloc - mstats.nfree;

    runtime.printf("gc%d(%d): %D+%D+%D+%D us, %D -> %D MB, %D (%D-%D) objects, ...
        mstats.numgc,           // 0: GC 执行次数。
        work.nproc,             // 1: 并行 GC goroutine 数量。
        (t1-t0)/1000,           // 2: 包含 stoptheworld 在内的准备时间。
        (t2-t1)/1000,           // 3: 非并发模式垃圾清理, 准备并行标记所耗时间。
        (t3-t2)/1000,           // 4: 并行标记所耗时间。
        (t4-t3)/1000,           // 5: 等待结束, 更新状态统计等所耗时间。
        heap0>>20,              // 6: 回收前正在使用的 object 内存容量。
        heap1>>20,              // 7: 回收后正在使用的 object 内存容量。
        obj,                     // 8: 回收后正在使用的 object 数量。
        mstats.nmalloc,          // 9: 总计 object 分配次数。
        mstats.nfree,            // A: 总计 object 释放次数。
        sweep.nspan,             // B: 清理需要扫描的 span 数量。
        gcstats.nbgswEEP,        // C: 并发模式清理 span 数量。
        gcstats.npausesweep,     // D: 暂停模式清理 span 数量 (非并发模式)。
        ...);
}
}

```

```

+- 2: 包含 stoptheworld 在内的准备时间。
+- 3: 非并发模式垃圾清理, 准备并行标记所耗时间。
+- 4: 并行标记所耗时间。
+- 5: 等待标记结束, 更新状态等所耗时间。
+- 8: 回收后正在使用的 object 数量。
+- 9: 总计 object 分配次数。
+- A: 总计 object 释放次数。
gc3(1): 3+1+187+0 us, 0 -> 0 MB, 46 (52-6) objects, 21/0/0 sweeps, ...
+- 0: GC 执行次数。
+- 1: 并行标行 goroutine 数量。
+- 6: 回收前正在使用的 object 内存容量。
+- 7: 回收后正在使用的 object 内存容量。
+- B: 清理需要扫描的 span 数量。
+- C: 并发模式清理 span 数量。
+- D: 暂停模式清理 span 数量 (非并发模式)。

```

因 1.3 使用了并发清理模式, 因此输出的统计信息并不准确, 比如回收前后正在使用的 object 容量、数量等。只能通过多次输出结果进行大概评估。

第二种信息则来自物理内存释放函数。

## mheap.c

```
static void scavenge(int32 k, uint64 now, uint64 limit)
```

```
{
    if(runtime·debug·gctrace > 0) {
        if(sumreleased > 0)
            runtime·printf("scvg%d: %D MB released\n", k, (uint64)sumreleased>>20);

        runtime·printf("scvg%d: inuse: %D, idle: %D, sys: %D, released: %D, ...",
            k, // 0: 执行次数。
            mstats.heap_inuse>>20, // 1: 正在被使用的 spans 容量。
            mstats.heap_idle>>20, // 2: 所有闲置 spans 的容量。
            mstats.heap_sys>>20, // 3: 已分配虚拟内存总量。
            mstats.heap_released>>20, // 4: 已释放物理内存总量。
            (mstats.heap_sys - mstats.heap_released)>>20); // 5: 当前实际消耗内存。
    }
}
```

```

+- 0: 执行次数。
+- 2: 所有空闲 span 内存容量。
+- 4: 已释放物理内存总量。
scvg6: inuse: 32, idle: 8, sys: 41, released: 0, consumed: 41 (MB)
+- 3: 已分配虚拟内存总量。
+- 5: 实际消耗的内存 = sys - released
+- 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)。

```

关于物理内存释放总数统计，需要说明一下。在 `scavengelist` 函数，`span` 所释放内存被累加到 `mstats.heap_released`。但该内存会在下次使用时被重新分配，因此需要调整。

## mheap.c

```
static uintptr scavengelist(MSpan *list, uint64 now, uint64 limit)
{
    sumreleased = 0;
    for(s=list->next; s != list; s=s->next) {
        if((now - s->unusedsince) > limit && s->npreleased != s->npages) {
            // 统计要释放的容量。
            released = (s->npages - s->npreleased) << PageShift;

            // 累加到释放总数。
            mstats.heap_released += released;

            sumreleased += released;
            s->npreleased = s->npages;
            runtime·SysUnused((void*)(s->start << PageShift), s->npages << PageShift);
        }
    }
    return sumreleased;
}

static MSpan* MHeap_AllocLocked(MHeap *h, uintptr npage, int32 sizeclass)
{

```

```
// 减去重新分配的内存。如果该 span 没有释放过, 其 npreleased == 0.  
mstats.heap_released -= s->npreleased<<PageShift;  
  
// 重新分配内存。  
if(s->npreleased > 0)  
    runtime·SysUsed((void*)(s->start<<PageShift), s->npages<<PageShift);  
  
// 重置释放标记。  
s->npreleased = 0;  
}
```



## 3. Goroutine Scheduler

调度器是运行时最核心的内容，要搞清工作原理，首先得知道 3 个基本抽象概念：

- **G**: goroutine 对象。每个 go 语句都会创建一个 G 对象。
- **P**: 抽象处理器。如同线程和 CPU 核的关系，执行 G 的前提是要获得一个 P。
- **M**: 系统线程。G 任务函数最终还是要由线程执行。

原理并不复杂，类似协程实现。通过自定义栈保存执行现场，从而将执行状态和执行线程分离。基于 **multiplexed thread**，在单个线程上切换执行多个并发任务。

相比线程，G 默认栈只有 4KB，按需扩张或收缩。这种轻量级设计开销极小，加上并发调度，避免了线程阻塞带来的损耗，轻松创建成千上万的并发任务。

当 M 执行 G 任务时，只需将相关寄存器 (SP) 指向 G 自带执行栈，然后 **JMP** 到任务函数即可。同样，在中途停止任务时，也只须将寄存器值写回 G 相关字段，任何 M 都可据此恢复并完成后续工作。

每个准备工作的 M 都必须绑定一个 P，以获得执行所需的资源环境。

- P 的数量限制了并发任务数量；
- 资源由 P 提供，如此无需为休眠的 M 提供资源设置。
- 数量众多的 G 任务分散到多个 P 队列，减少全局锁排队。

通常，P 的数量是“固定”的（默认 1，最多 256），但 M 则不确定。例如某个 M 被系统调用长时间阻塞，其关联的 P 会被收回，调度器会唤醒或创建新的 M 去执行其他排队任务。失去 P 的 M 则进入休眠状态，直到被调度器重新唤醒。

### 3.1 初始化

在 `proc.c` 注释里提到了 `bootstrap` 过程，其实际代码由汇编实现。

`amd_asm64.s`

```
TEXT _rt0_go(SB),NOSPLIT,$0
```

```
// ... 准备操作 ...
```

```
MOVL    16(SP), AX    // copy argc
```

```

MOVL    AX, 0(SP)
MOVQ    24(SP), AX    // copy argv
MOVQ    AX, 8(SP)
CALL    runtime·args(SB)
CALL    runtime·osinit(SB)
CALL    runtime·hashinit(SB)
CALL    runtime·schedinit(SB)

// create a new goroutine to start program
PUSHQ    $runtime·main·f(SB)    // entry
PUSHQ    $0                    // arg size
ARGSIZE(16)
CALL    runtime·newproc(SB)
ARGSIZE(-1)
POPQ    AX
POPQ    AX

// start this M
CALL    runtime·mstart(SB)

MOVL    $0xf1, 0xf1 // crash
RET

```

看看调度器和几个相关的全局变量。

## proc.c

```

struct Sched {
    M*    midle;        // idle m's waiting for work
    int32  nmidle;       // number of idle m's waiting for work
    int32  nmidlelocked; // number of locked m's waiting for work
    int32  mcount;       // number of m's that have been created
    int32  maxmcount;    // maximum number of m's allowed (or die)

    P*    pidle;        // 空闲 P 链表。
    uint32 npidle;

    // 待运行 G 全局链表
    G*    runqhead;
    G*    runqtail;
    int32  runqsize;

    // 可复用 G 对象链表
    G*    gfree;
};

Sched    runtime·sched;        // 调度器唯一实例。
int32    runtime·gomaxprocs;   // GOMAXPROCS

```

```

M    runtime.m0;
G    runtime.g0;           // idle goroutine for m0
M*   runtime.allm;
G**  runtime.allg;

```

调度器的初始化过程并不复杂，调用相关部件的初始化方法，初始化环境和全局参数，而其中最需要关注的就是 **P**。

## proc.c

```

// The bootstrap sequence is:
//
// call osinit
// call schedinit
// make & queue new G
// call runtime.mstart
//
// The new G calls runtime.main.
void runtime.schedinit(void)
{
    // 最大线程数量限制。
    runtime.sched.maxmcount = 10000;

    // 符号表初始化。
    runtime.syntabinit();

    // 内存分配器初始化。
    runtime.mallocinit();

    // 初始化当前 M。
    mcommoninit(m);

    // 相关参数。
    runtime.goargs();
    runtime.goenvs();
    runtime.parsedebgvars();
    runtime.sched.lastpoll = runtime.nanotime();

    // 设置 GOMAXPROCS，默认值 1，最大值 256。
    procs = 1;
    p = runtime.getenv("GOMAXPROCS");
    if(p != nil && (n = runtime.atoi(p)) > 0) {
        if(n > MaxGomaxprocs)
            n = MaxGomaxprocs;
        procs = n;
    }
}

```

```
// 初始化 P。
runtime·allp = runtime·malloc((MaxGomaxprocs+1)*sizeof(runtime·allp[0]));
proccresize(procs);
}
```

为全局 P 指针数组分配内存 (一次性分配了 256 个指针)，然后按 GOMAXPROCS 数量初始化 P 实例。

## runtime.h

```
struct P
{
    uint32  status;          // one of Pidle/Prunning/...
    P*      link;
    M*      m;               // back-link to associated M (nil if idle)
    MCache* mcache;

    // 待运行 G 队列 (循环数组)
    uint32  runqhead;
    uint32  runqtail;
    G*      runq[256];

    // 可复用 G 对象链表 (status == Gdead)
    G*      gfree;
    int32   gfreecnt;
};
```

调整 P 的数量，需要将现有 P 本地任务队列进行转移，然后再平均给新建的 P 对象。

## proc.c

```
static void proccresize(int32 new)
{
    old = runtime·gomaxprocs;

    // 不能超出 256。
    if(old < 0 || old > MaxGomaxprocs || new <= 0 || new > MaxGomaxprocs)
        runtime·throw("proccresize: invalid arg");

    // 创建并初始化所需的 P。
    for(i = 0; i < new; i++) {
        p = runtime·allp[i];
        if(p == nil) {
            p = (P*)runtime·mallocgc(sizeof(*p), 0, FlagNoInvokeGC);
            p->id = i;
            p->status = Pgcstop;
            runtime·atomicstorep(&runtime·allp[i], p);
        }
    }
}
```

```

// 为 P 设置 MCache。
if(p->mcache == nil) {
    if(old==0 && i==0)
        p->mcache = m->mcache; // bootstrap
    else
        p->mcache = runtime·allocmcache();
}
}

// 由于 P 数量调整, 需重新分布 P 里面的 G 队列。
empty = false;
while(!empty) {
    empty = true;

    // 内层 for 循环遍历所有 old P, 依次从每个 P 里提取一个 G 放到全局队列。
    // 然后通过外层 while 循环多次调用 for 循环, 确保所有 G 都被转移到全局队列。
    // 这么做的目的还是希望各 P 里面先创建的 G 会优先进入全局队列, 以更早被执行。
    for(i = 0; i < old; i++) {
        p = runtime·allp[i];

        // 链表为空。
        if(p->runqhead == p->runqtail)
            continue;

        empty = false;

        // 从 P 队列尾部提取 G。
        p->runqtail--;
        gp = p->runq[p->runqtail%nelem(p->runq)];

        // 放到全局 G 队列 (链表)。
        gp->schedlink = runtime·sched.runqhead;
        runtime·sched.runqhead = gp;
        if(runtime·sched.runqtail == nil)
            runtime·sched.runqtail = gp;
        runtime·sched.runqsize++;
    }
}

// 将全局队列中的 "部分" G 重新分布到 P 本地队列中。
// 每个 P 队列里最多填充一半。
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 P 数量少于 old, "释放" 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); // 将 P 持有的可复用 G 对象转移到全局链表。
        p->status = Pdead;
        // can't free P itself because it can be referenced by an M in syscall
    }

    // 为当前 M 关联 P。
    p = runtime·allp[0];
    acquirep(p);

    // 将全部 P 放到空闲链表 (allp[0] 已被使用)。
    for(i = new-1; i > 0; i--) {
        p = runtime·allp[i];
        p->status = Pidle;
        pidleput(p);
    }
}

```

G 结构很复杂, 涉及到栈、抢占调用等很多参数, 后面会详细说明。

## runtime.h

```

struct G
{
    G*  schedlink;
};

```

在完成调度器初始化后, bootstrap 创建 main goroutine 运行 runtime·main。

## proc.c

```

// The main goroutine.
void runtime·main(void)
{
    // 栈大小限制。
    if(sizeof(void*) == 8)
        runtime·maxstacksize = 1000000000;
    else
        runtime·maxstacksize = 250000000;

    // 使用独立线程执行系统监控。
}

```

```

newm(sysmon, nil);

// 创建 goroutine 运行垃圾回收器定时操作。
runtime.newproc1(&scavenger, nil, 0, 0, runtime.main);

// 执行用户代码的 init 和 main 函数。
main.init();
main.main();

// 进程结束。
runtime.exit(0);
}

```

## 3.2 创建任务

编译器将每个 go 语句编译成 newproc1 函数调用，创建 G 对象。

### proc.c

```

G* runtime.newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    byte *sp;
    G *newg;
    P *p;

    // 从当前 P 获取可复用的 G 对象。
    p = m->p;
    if((newg = gfget(p)) != nil) {
        // ...
    } else {
        // 新建 G 对象。
        newg = runtime.malg(StackMin);

        // 全局变量 allg 保存所有 G 对象指针。
        allgadd(newg);
    }

    // 函数参数入栈。
    sp = (byte*)newg->stackbase;
    sp -= siz;
    runtime.memmove(sp, argp, narg);

    // sched 用于保存执行现场，比如寄存器值。
    runtime.memclr((byte*)&newg->sched, sizeof newg->sched);
    newg->sched.sp = (uintptr)sp;
    newg->sched.pc = (uintptr)runtime.goexit;
}

```

```

newg->sched.g = newg;
runtime.gostartcallfn(&newg->sched, fn);

// 设置 G 状态。
newg->gopc = (uintptr)callerpc;
newg->status = Grunnable;

// 将 G 保存到 P 本地队列。
// 如果本地队列已满，则放到全局队列。
runqput(p, newg);

// 尝试唤醒某个 M 开始执行任务。(有空闲 P，且没有处于自旋等待的 M)
if(runtime.atomicload(&runtime.sched.npidle) != 0 &&
    runtime.atomicload(&runtime.sched.nmspinning) == 0 &&
    fn->fn != runtime.main) // TODO: fast atomic
    wakep();

return newg;
}

static void wakep(void)
{
    startm(nil, true);
}

```

调度器对 G 对象做了缓存复用处理。如果 P 本地可复用队列为空，则从全局队列提取一些到本地。实在没有的情况下，才会新建 G 对象。

## proc.c

```

static G* gfget(P *p)
{
    retry:
        // 本地可复用 G 队列。
        gp = p->gfree;

        // 如果本地队列为空，则从全局队列找。
        if(gp == nil && runtime.sched.gfree) {
            // 提取可复用 G 到本地队列（本地队列最多有 32 个可复用对象）。
            while(p->gfreesent < 32 && runtime.sched.gfree) {
                p->gfreesent++;
                gp = runtime.sched.gfree;
                runtime.sched.gfree = gp->schedlink;
                gp->schedlink = p->gfree;
                p->gfree = gp;
            }

            goto retry;
        }
}

```



```

}

// 从本地队列提取。
if(gp) {
    p->gfree = gp->schedlink;
    p->gfreecnt--;

    // 初始化栈空间。
    if(gp->stack0 == 0) {
        // Stack was deallocated in gput. Allocate a new one.
    }
}

return gp;
}

```

G.sched 字段保存了执行现场所需的环境参数。

#### runtime.h

```

struct G
{
    Gobuf    sched;
    uintptr  gopc;    // 执行该 go 语句的函数指针。
};

struct Gobuf
{
    uintptr  sp;      // 相当于 SP 寄存器
    uintptr  pc;      // 相当于 IP/PC 寄存器
    G*       g;
    void*    ctxt;
    uintreg  ret;
    uintptr  lr;
};

```

在 newproc1 初始化 G.sched 过程中，有个关键的 gostartcallfn 调用。

#### proc.c

```

void runtime·gostartcallfn(Gobuf *gobuf, FuncVal *fv)
{
    runtime·gostartcall(gobuf, fv->fn, fv);
}

```

#### sys\_x86.c

```

void runtime·gostartcall(Gobuf *gobuf, void (*fn)(void), void *ctxt)

```

```

{
    sp = (uintptr*)gobuf->sp;

    // 在初始化时, pc 指向 runtime·goexit。
    // 将 runtime·goexit 地址入栈。
    *--sp = (uintptr)gobuf->pc;

    // 重新调整 sp 位置。
    gobuf->sp = (uintptr)sp;

    // 将 pc 从 runtime·goexit 改为实际要执行的 go 函数。
    gobuf->pc = (uintptr)fn;

    gobuf->ctxt = ctxt;
}

```

当 M 执行完 G 任务后, 必须调用 `goexit` 清理现场。将 `goexit` 地址提前入栈, 任务函数结束时, 其 `RET` 指令会将该地址 `POP` 到 `IP` 寄存器, 从而完成对 `goexit` 函数的调用。

注: 汇编 `CALL` 指令会将当前 `IP` 入栈, 调用完成后再由 `RET` 指令 `POP IP` 恢复到调用位置。

### 3.3 执行任务

要执行 `goroutine`, 必须创建或唤醒某个处于休眠状态的 M。

```

runtime.h
struct M
{
    G*      g0;                // 用于执行调度器指令。
    void    (*mstartfn)(void); // 启动函数。
    G*      curg;              // 准备执行的 G 任务。
    P*      p;                 // 关联 P。
    P*      nextp;             // 关联 P 临时存放位置。
    Note    park;              // 休眠标记。
    MCache* mcache;           // 所关联 P.cache。
};

```

每个 M 都有个 `g0` 栈, 用于执行调度器相关指令 (和 `goroutine` 函数执行栈分开)。

让 M 工作的前提是有闲置的 P 可用, P 的数量限制了同时并发 M 的数量。没有关联到 P 的 M 不会被销毁, 而是进入休眠状态, 等待下次被唤醒。

**proc.c**

```
static void startm(P *p, bool spinning)
{
    M *mp;
    void (*fn)(void);

    // 获取空闲 P。
    if(p == nil) {
        p = pidleget();
        if(p == nil) {
            return;
        }
    }

    // 从空闲 M 队列获取。
    mp = mget();
    if(mp == nil) {
        // 创建 M, 注意 fn = nil。
        fn = nil;
        newm(fn, p);
        return;
    }

    // 关联 P。
    mp->nextp = p;

    // 唤醒休眠的 M。
    // 调度器调用 notesleep 让 M 休眠, notewakeup 唤醒。
    runtime·notewakeup(&mp->park);
}
```

M 是对系统线程的一种抽象，其最大数量和 P 没有对应关系。

**proc.c**

```
static void newm(void(*fn)(void), P *p)
{
    // 创建并初始化 M。
    mp = runtime·allocm(p);
    mp->nextp = p;
    mp->mstartfn = fn;

    // 创建线程。
    runtime·newosproc(mp, (byte*)mp->g0->stackbase);
}
```

初始化过程并不复杂，其中有个最大数量检查 (可用 `runtime/debug.SetMaxThreads` 修改)，超出会导致进程崩溃退出。

## proc.c

```
M* runtime·allocm(P *p)
{
    mp = runtime·cnew(mtype);

    // 初始化。
    mcommoninit(mp);

    // 创建 g0 栈。
    if(runtime·iscgo || Solaris || Windows)
        mp->g0 = runtime·malg(-1);
    else
        mp->g0 = runtime·malg(8192);

    return mp;
}

static void mcommoninit(M *mp)
{
    mp->id = runtime·sched.mcount++;

    // 检查是否超出最大 M 数量限制。
    checkmcount();

    runtime·mpreinit(mp);

    // 将 M 添加到 allm 链表。
    mp->alllink = runtime·allm;
    runtime·atomicstorep(&runtime·allm, mp);
}

static void checkmcount(void)
{
    // 在 schedinit 函数中, maxmcount 默认值 10000。
    if(runtime·sched.mcount > runtime·sched.maxmcount) {
        runtime·printf("runtime: program exceeds %d-thread limit\n",
                       runtime·sched.maxmcount);
        runtime·throw("thread exhaustion");
    }
}
```

回到 M 执行上来，通过系统调用创建线程，执行函数 `runtime·mstart`。

## 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);
}
```

线程函数首先检查是否有 `mstartfn`，比如在 `runtime·main` 里 `newm(sysmon, nil)` 就是如此执行的。绑定 `P` 的相关环境后，正式进入调度器核心 `schedule` 循环。

## proc.c

```
void runtime·mstart(void)
{
    // 如果提供了 startfn 函数，那么就先执行。
    if(m->mstartfn)
        m->mstartfn();

    if(m->helpgc) {
        m->helpgc = 0;
        stopm();
    } else if(m != &runtime·m0) {
        // 绑定 P。
        acquirep(m->nextp);
        m->nextp = nil;
    }

    // 调度函数，会一直循环。
    schedule();

    // TODO(brainman): This point is never reached, because scheduler
    // does not release os threads at the moment. But once this path
    // is enabled, we must remove our seh here.
}

static void acquirep(P *p)
{
    m->mcache = p->mcache;
    m->p = p;
    p->m = m;
    p->status = Prunning;
}
```

调度函数首先从相关队列提取待运行的 G。

#### proc.c

```
// One round of scheduler: find a runnable goroutine and execute it.
// Never returns.
static void schedule(void)
{
top:
    gp = nil;

    // 定期检查全局队列，以确保其中的 G 不会等待太长时间。
    tick = m->p->schedtick;
    if(tick - (((uint64)tick*0x4325c53fu)>>36)*61 == 0 && runtime.sched.runqsize > 0) {
        gp = globrunqget(m->p, 1); // 返回一个可用 G。
    }

    // 从本地队列提取。
    if(gp == nil) {
        gp = runqget(m->p);
    }

    // 从其他队列查找。
    if(gp == nil) {
        gp = findrunnable(); // blocks until work is available
    }

    // 如果 G 被锁定到某个 M，那么就将 G 连同 P 一起移交给它。
    if(gp->lockedm) {
        // Hands off own p to the locked m,
        // then blocks waiting for a new p.
        startlockedm(gp);
        goto top;
    }

    execute(gp);
}
```

官方文档所提及 **work-stealing** 算法，就在 **findrunnable** 函数里。该函数会尝试各种途径获取可用 G，直到休眠为止。

#### proc.c

```
static G* findrunnable(void)
{
top:
    // 本地队列
```

```

gp = runqget(m->p);
if(gp)
    return gp;

// 全局队列
if(runtime.sched.runqsize) {
    // 转移一批 G 到本地队列, 并返回第一个。
    gp = globrunqget(m->p, 0);
    if(gp)
        return gp;
}

// poll network
gp = runtime.netpoll(false); // non-blocking
if(gp) {
    injectglist(gp->schedlink);
    gp->status = Grunnable;
    return gp;
}

// 随机从其他某个 P 队列 "偷" 一半过来。
for(i = 0; i < 2*runtime.gomaxprocs; i++) {
    p = runtime.allp[runtime.fastrand1()%runtime.gomaxprocs];
    if(p == m->p)
        gp = runqget(p);
    else
        gp = runqsteal(m->p, p);
    if(gp)
        return gp;
}
stop:
// 再次检查全局队列。
if(runtime.sched.runqsize) {
    gp = globrunqget(m->p, 0);
    return gp;
}

// 如果还是没有 G 任务, 释放 P。
p = releasep();
pidleput(p);

// 看看能不能换个有任务的 P 继续工作。
for(i = 0; i < runtime.gomaxprocs; i++) {
    p = runtime.allp[i];
    if(p && p->runqhead != p->runqtail) {
        p = pidleget();
        if(p) {
            acquirep(p);

```

```

        goto top;
    }
    break;
}
}

// 看看有没有新的 network epoll 任务出现, 不过还得关联到一个 P 才行。
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) {
            acquirep(p);
            injectglist(gp->schedlink);
            gp->status = Grunnable;
            return gp;
        }
        injectglist(gp);
    }
}

// 什么任务都没有的话, 进入休眠。
// 注意线程虽然休眠, 但线程执行绪依然存在, 一旦被唤醒, 会在此处继续执行。
// 至于 P, 无需担心, startm 函数总是拿到可用 P 后才会唤醒 M。
stopm();
goto top;
}

```

在准备好 G 后, 开始执行 `execute` 函数。

## proc.c

```

// Schedules gp to run on the current M.
// Never returns.
static void execute(G *gp)
{
    // 设置相关状态。
    gp->status = Grunning;
    gp->waitsince = 0;
    gp->preempt = false;
    gp->stackguard0 = gp->stackguard;
    m->p->schedtick++;
    m->curg = gp;
    gp->m = m;

    // 调用汇编代码, 执行 goroutine。
    runtime·gogo(&gp->sched);
}

```



```
}

```

## asm\_amd64.s

```
// void gogo(Gobuf*)
// restore state from Gobuf; longjmp
TEXT runtime·gogo(SB), NOSPLIT, $0-8
    MOVQ    8(SP), 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.sp 恢复 SP 寄存器
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    $0, gobuf_sp(BX)    // clear to help garbage collector
    MOVQ    $0, gobuf_ret(BX)
    MOVQ    $0, gobuf_ctxt(BX)
    MOVQ    gobuf_pc(BX), BX    // 将 G.sched.pc, 也就是 goroutine 函数地址压入 BX。
    JMP     BX                  // 跳转到该函数, 开始执行。
```

这段汇编指令并不复杂, 核心就是从 `G.sched` 里面恢复执行现场。

注意, `gogo` 没有保存 `execute` 的现场, 没有将 `IP` 入栈。前面我们已经提到过, `G` 预先将 `goexit` 地址入栈。因此, 当 `goroutine` 函数执行完毕, 其尾部的 `RET` 指令 `POP IP`, 实际跳转到 `goexit` 函数。

## proc.c

```
runtime·goexit(void)
{
    runtime·mcall(goexit0);
}

static void goexit0(G *gp)
{
    gp->status = Gdead;
    gp->m = nil;
    gp->lockedm = nil;
    gp->paniconfault = 0;
    m->curg = nil;
    m->lockedg = nil;
    m->locked = 0;

    // 释放栈空间, 并将 G 放回本地可复用链表。
    runtime·unwindstack(gp, nil);
    gfput(m->p, gp);
}
```

```

    // 重新进入调度函数。
    schedule();
}

static void gfree(P *p, G *gp)
{
    // 保存到 P 本地可复用链表。
    gp->schedlink = p->gfree;
    p->gfree = gp;
    p->gfreecnt++;

    // 如果链表过长, 就将多余部分转移到全局链表。
    if(p->gfreecnt >= 64) {
        while(p->gfreecnt >= 32) {
            p->gfreecnt--;
            gp = p->gfree;
            p->gfree = gp->schedlink;
            gp->schedlink = runtime.sched.gfree;
            runtime.sched.gfree = gp;
        }
    }
}

```

收尾工作包括处理栈空间, 将 **G** 放到可复用链表。然后再次跳转到 **schedule** 函数, 形成调度循环。

再看 **M** 休眠操作是如何实现的。

## proc.c

```

static void stopm(void)
{
    // 将 M 放到闲置队列。
    mput(m);

    // 通过系统调用让 M 休眠。(线程暂停执行, 但未被销毁, 执行绪依然保留)
    runtime.notesleep(&m->park);

    // 被唤醒后, 重置标记。(唤醒函数会修改该标记)
    runtime.noteclear(&m->park);

    // 关联 P, 继续从休眠位置开始执行。(既然被唤醒, 调度器必然为其准备好了 P 对象)
    acquirep(m->nextp);
    m->nextp = nil;
}

```

线程休眠基本上是通过用户空间的互斥方式实现。

### lock\_futex.c

```
void runtime·notesleep(Note *n)
{
    while(runtime·atomicload((uint32*)&n->key) == 0) {
        runtime·futexsleep((uint32*)&n->key, 0, -1);
    }
}
```

### os\_linux.c

```
void runtime·futexsleep(uint32 *addr, uint32 val, int64 ns)
{
    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);
}
```

函数 `notesleep` 传递的 `ns` 参数是 `-1`，在 `futexsleep` 里意味着永不超时。因此，在不唤醒的情况下，`M` 所对应的线程会一直沉睡。

调用 `notewakeup` 解除休眠，唤醒 `M` 继续执行。

### lock\_futex.c

```
void runtime·notewakeup(Note *n)
{
    // 修改标记位值。
    old = runtime·xchg((uint32*)&n->key, 1);

    // 解除 umtx_sleep 休眠。
    runtime·futexwakeup((uint32*)&n->key, 1);
}
```

### os\_linux.c

```
void runtime·futexwakeup(uint32 *addr, uint32 cnt)
{
    ret = runtime·futex(addr, FUTEX_WAKE, cnt, nil, nil, 0);
    if(ret >= 0) return;
}
```

另外，在 `schedule` 里还提到了 `lockedm`，这是一种很特别的情况，它表示当前 `G` 只能由特定的 `M` 执行。

#### proc.c

```
static void startlockedm(G *gp)
{
    // G 锁定的目标 M。
    mp = gp->lockedm;

    // 解除当前 M 的 P 绑定。
    p = releasep();

    // 将 P 移交给目标 M，并唤醒。
    mp->nextp = p;
    runtime·notewakeup(&mp->park);

    // 当前 M 开始休眠。
    stopm();
}
```

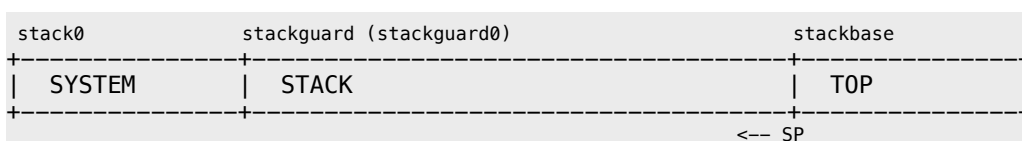
### 3.4 连续栈

在 1.3 里，默认采用连续栈，但保留了分段栈算法，可用环境变量切换。

#### runtime.h

```
struct G
{
    uintptr stackguard;    // 执行栈顶。
    uintptr stackguard0;   // 与 stackguard 相同，通常用于设置 StackPreempt 抢占标记。
    uintptr stackbase;     // 执行栈底。
    uintptr stack0;        // 所分配栈起始位置。
    uintptr stacksize;     // 栈大小（包括所有分段）。
};
```

栈结构依然是分段栈模式，只是连续栈在需要时会分配更大空间，拷贝所有数据。



在创建 `G` 对象时，为其分配栈空间，默认大小 4096。

**proc.c**

```

G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    if((newg = gfget(p)) != nil) {
        ...
    } else {
        newg = runtime·malg(StackMin); // 默认大小 4096。
        allgadd(newg);
    }

    ...

    return newg;
}

G* runtime·malg(int32 stacksize)
{
    newg = runtime·malloc(sizeof(G));

    if(stacksize >= 0) {
        stacksize = runtime·round2(StackSystem + stacksize);
        if(g == m->g0) {
            stk = runtime·stackalloc(newg, stacksize);
        } else {
            newg->stacksize = stacksize;
            g->param = newg;
            runtime·mcall(mstackalloc);
            stk = g->param;
            g->param = nil;
        }

        // 设置栈相关参数。
        newg->stack0 = (uintptr)stk;
        newg->stackguard = (uintptr)stk + StackGuard;
        newg->stackguard0 = newg->stackguard;
        newg->stackbase = (uintptr)stk + stacksize - sizeof(Stktop);
    }

    return newg;
}

```

系统为默认固定大小的栈提供了缓存策略，所需内存直接从 `mmap` 分配而非 `heap`。这也容易理解，这些缓存对象生命周期持续到进程结束，完全没必要和 GC 打交道。

栈缓存由 `M` 管理，在创建 `G` 的时候分配给它。

## runtime.h

```

struct M
{
    int32    stackinuse;           // 已使用数量
    uint32   stackcachepos;       // 可用索引序号
    uint32   stackcachecnt;       // 可用数量
    void*    stackcache[StackCacheSize]; // 缓存数组
};

```

## stack.c

```

void* runtime·stackalloc(G *gp, uint32 n)
{
    // 累加栈分配空间计数。
    gp->stacksize += n;

    malloced = true;

    // 缓存固定大小的栈空间。
    if(n == FixedStack || m->mallocing) {
        // 如果缓存为空, 那么从全局缓存提取或重新分配。
        if(m->stackcachecnt == 0)
            stackcacherefill();

        // 从本地缓存数组中获取栈空间。
        pos = m->stackcachepos;
        pos = (pos - 1) % StackCacheSize;
        v = m->stackcache[pos];
        m->stackcachepos = pos;
        m->stackcachecnt--;
        m->stackinuse++;

        // malloced 标记用于判断缓存和分配。
        malloced = false;
    } else
        // 从 gc heap 单独分配空间。
        v = runtime·malloccg(n, 0, FlagNoProfiling|FlagNoGC|FlagNoZero|FlagNoInvokeGC);

    // 重置 TOP 区域。
    top = (Stktop*)((byte*)v+n-sizeof(Stktop));
    runtime·memclr((byte*)top, sizeof(*top));

    top->malloced = malloced;
    return v;
}

static void stackcacherefill(void)

```

```
{
    n = (StackCacheNode*)runtime·SysAlloc(FixedStack*StackCacheBatch, ...);
}
```

在函数头部，编译器都会插入 `morestack` 调用，用于检查是否需要扩张栈空间。

### asm\_amd64.s

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    // Call newstack on m->g0's stack.
    MOVQ    m_g0(BX), BP
    MOVQ    BP, g(CX)
    MOVQ    (g_sched+gobuf_sp)(BP), SP
    CALL    runtime·newstack(SB)
    MOVQ    $0, 0x1003    // crash if newstack returns
    RET
```

在有些臃肿的 `newstack` 函数里，我们可以同时看到连续栈和分段栈算法。

### stack.c

```
void runtime·newstack(void)
{
    // gp->status is usually Grunning, but it could be Gsyscall if a stack overflow
    // happens during a function call inside entersyscall.

    // 保存当前状态，用于恢复。
    gp = m->curg;
    oldstatus = gp->status;

    framesize = m->moreframesize;
    argsize = m->moreargsize;

    gp->status = Gwaiting;
    gp->waitreason = "stack growth";

    // 抢占式调度。
    if(gp->stackguard0 == (uintptr)StackPreempt) {
        runtime·gosched0(gp);    // never return
    }

    /* --- 连续栈 ----- */

    // 由 GOCOPYSTACK 控制，默认 true (proc.c runtime·schedinit)。
    if(runtime·copystack) {
        nframes = copyabletopsegment(gp);
        if(nframes != -1) {
            // 计算当前栈大小。

```

```

    oldstk = (byte*)gp->stackguard - StackGuard;
    oldbase = (byte*)gp->stackbase + sizeof(Stktop);
    oldsize = oldbase - oldstk;

    // 新栈是原来的 2 倍。
    newsize = oldsize * 2;

    // 分配新栈空间, 并拷贝数据。
    copystack(gp, nframes, newsize);

    // 恢复状态。
    gp->status = oldstatus;

    // 用 sched 现场恢复执行。
    runtime·gogo(&gp->sched);
}
}

/* ---- 分段栈 ----- */

// 计算新栈帧所需大小。
framesize += argsize;
framesize += StackExtra;    // room for more functions, Stktop.
if(framesize < StackMin)
    framesize = StackMin;
framesize += StackSystem;
framesize = runtime·round2(framesize);

// 分配新栈帧空间。
stk = runtime·stackalloc(gp, framesize);

// 在新栈帧 top 区域记录原栈帧相关信息, 形成链表结构。
top = (Stktop*)(stk+framesize-sizeof(*top));
top->stackbase = gp->stackbase;
top->stackguard = gp->stackguard;

// 将 G 栈信息指向新栈帧空间。
gp->stackbase = (uintptr)top;
gp->stackguard = (uintptr)stk + StackGuard;
gp->stackguard0 = gp->stackguard;

// 将参数复制到新栈帧。
sp = (uintptr)top;
if(argsize > 0) {
    sp -= argsize;
    dst = (uintptr*)sp;
    dstend = dst + argsize/sizeof(*dst);
    src = (uintptr*)top->argp;

```



```

        while(dst < dstend)
            *dst++ = *src++;
    }
}

```

当需要扩张时，连续栈会重新分配栈内存，然后将数据转移过去，并释放现有栈空间。

## stack.c

```

static void copystack(G *gp, uintptr nframes, uintptr newsize)
{
    // 确定旧栈参数。
    oldstk = (byte*)gp->stackguard - StackGuard;
    oldbase = (byte*)gp->stackbase + sizeof(Stktop);
    oldsize = oldbase - oldstk;
    used = oldbase - (byte*)gp->sched.sp;
    oldtop = (Stktop*)gp->stackbase;

    // 分配新栈。
    newstk = runtime.stackalloc(gp, newsize);
    newbase = newstk + newsize;
    newtop = (Stktop*)(newbase - sizeof(Stktop));
    malloced = newtop->malloced;

    // 将旧站数据转移到新栈。
    runtime.memmove(newbase - used, oldbase - used, used);
    newtop->malloced = malloced; // 分配标记

    // 修改 G 参数，指向新栈。
    gp->stackbase = (uintptr)newtop;
    gp->stackguard = (uintptr)newstk + StackGuard;
    gp->stackguard0 = (uintptr)newstk + StackGuard;
    if(gp->stack0 == (uintptr)oldstk)
        gp->stack0 = (uintptr)newstk;
    gp->sched.sp = (uintptr)(newbase - used);

    // 释放旧栈内存。
    runtime.stackfree(gp, oldstk, oldtop);
}

```

当 `goexit` 清理 G 任务现场时，会调用 `unwindstack` 释放额外分配的栈空间。

## panic.c

```

void runtime.unwindstack(G *gp, byte *sp)
{
    // 通过 TOP 链表，检查并释放分段栈空间（不包括第一个分段）。
    while((top = (Stktop*)gp->stackbase) != 0 && top->stackbase != 0) {

```

```

    // 当前分段信息。
    stk = (byte*)gp->stackguard - StackGuard;
    if(stk <= sp && sp < (byte*)gp->stackbase)
        break;

    // 指向前一分段。
    gp->stackbase = top->stackbase;
    gp->stackguard = top->stackguard;
    gp->stackguard0 = gp->stackguard;

    // 释放当前分段。
    runtime·stackfree(gp, stk, top);
}
}

```

## stack.c

```

void runtime·stackfree(G *gp, void *v, Stktop *top)
{
    n = (uintptr)(top+1) - (uintptr)v;
    gp->stacksize -= n;

    // 如果是重新分配的栈, 释放内存!
    if(top->malloced) {
        runtime·free(v);
        return;
    }

    // 如果缓存已满, 则转移一部分到全局缓存。
    if(m->stackcachecnt == StackCacheSize)
        stackcacherelease();

    // 将栈空间归还给本地缓存。
    pos = m->stackcachepos;
    m->stackcache[pos] = v;
    m->stackcachepos = (pos + 1) % StackCacheSize;
    m->stackcachecnt++;
    m->stackinuse--;
}

```

我们注意到, `unwindstack` 并没有处理第一分段, 也就是说不管是连续栈还是分段栈, 默认的栈空间并没有被收回。

对于分段栈而言, 第一分段会一直跟随 `G` 对象进行复用。而如果是连续栈, 会存在因空间扩张被重新分配的问题。在 `goexit` 将 `G` 放回空闲链表时, 会专门处理这个状况。

**proc.c**

```

static void gfpout(P *p, G *gp)
{
    top = (Stktop*)gp->stackbase;

    // 如果是重新分配的栈，那么就释放掉。
    if(top->malloved) {
        runtime·stackfree(gp, (void*)gp->stack0, top);
        gp->stack0 = 0;
        gp->stackguard = 0;
        gp->stackguard0 = 0;
        gp->stackbase = 0;
    }
}

static G* gfget(P *p)
{
    if(gp) {
        // 如果没有栈空间，则重新分配。
        if(gp->stack0 == 0) {
            // Stack was deallocated in gfpout. Allocate a new one.
            if(g == m->g0) {
                stk = runtime·stackalloc(gp, FixedStack);
            } else {
                gp->stacksize = FixedStack;
                g->param = gp;
                runtime·mcall(mstackalloc);
                stk = g->param;
                g->param = nil;
            }
            gp->stack0 = (uintptr)stk;
            gp->stackbase = (uintptr)stk + FixedStack - sizeof(Stktop);
            gp->stackguard = (uintptr)stk + StackGuard;
            gp->stackguard0 = gp->stackguard;
        }
    }

    return gp;
}

```

在垃圾回收过程中，还会尝试收缩连续栈空间。

**mgc0.c**

```

static void gc(struct gc_args *args)
{
    // Shrink a stack if not much of it is being used.
    for(i = 0; i < runtime·allglen; i++)

```

```
runtime·shrinkstack(runtime·allg[i]);
}
```

### 3.5 系统调用

为配合调度器工作，专门使用 `entersyscall`、`entersyscallblock` 函数对 `syscall`、`cgo` 等操作进行包装，以便在阻塞时能释放 `P` 让其他任务得以运行。

#### src/syscall/asm\_linux\_amd64.s

```
TEXT    ·Syscall(SB),NOSPLIT,$0-64
    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, $0xffffffffffff001
    JLS ok
    MOVQ    $-1, 40(SP) // r1
    MOVQ    $0, 48(SP) // r2
    NEGQ    AX
    MOVQ    AX, 56(SP) // errno
    CALL    runtime·exitsyscall(SB)
    RET
```

#### cgo-call.c

```
void runtime·cgocall(void (*fn)(void*), void *arg)
{
    runtime·entersyscall();
    runtime·asmcgocall(fn, arg);
    runtime·exitsyscall();

    endcgo();
}
```

函数 `entersyscall` 所包装系统调用，不会释放所关联 `P`，但会因超时被 `sysmon` 没收。而 `entersyscallblock` 则是长时间阻塞模式，主动释放 `P`。

两个函数都会将执行现场保存到 `G.sched`，就算 `P` 被没收，依旧可被其他 `M` 恢复执行。

**proc.c**

```

void entersyscallblock(int32 dummy)
{
    // 将执行现场保存到 G.sched。
    save(runtime·getcallerpc(&dummy), runtime·getcallersp(&dummy));

    // 设置 syscall 状态
    g->syscallsp = g->sched.sp;
    g->syscallpc = g->sched.pc;
    g->syscallstack = g->stackbase;
    g->syscallguard = g->stackguard;
    g->status = Gsyscall;

    // 释放所关联的 P。
    p = releasep();

    // 尝试让该 P 执行其他任务。
    handoffp(p);

    save(runtime·getcallerpc(&dummy), runtime·getcallersp(&dummy));
}

static void save(void *pc, uintptr sp)
{
    g->sched.pc = (uintptr)pc;
    g->sched.sp = sp;
    g->sched.lr = 0;
    g->sched.ret = 0;
    g->sched.ctxt = 0;
    g->sched.g = g;
}

static P* releasep(void)
{
    m->p = nil;
    m->mcache = nil;
    p->m = nil;
    p->status = Pidle;
    return p;
}

```

对于所释放 P，函数 `handoffp` 会进行各种 `startm` 尝试，直到放回闲置队列。

**proc.c**

```

static void handoffp(P *p)
{
    // 如果 P 本地队列还有待执行任务，唤醒或创建 M 执行。

```

```

    if(p->runqhead != p->runqtail || runtime·sched·runqsize) {
        startm(p, false);
        return;
    }

    // 检查全局队列是否有待运行任务。
    if(runtime·sched·runqsize) {
        startm(p, false);
        return;
    }

    ...

    // 放回闲置队列。
    pidleput(p);
}

```

从系统调用退出时，必须检查关联 P 是否有效，能否获取空闲的 P。如果关联失败，则将 G 放回全局队列，等待其他 M 恢复并完成余下任务。

## proc.c

```

void runtime·exitsyscall(void)
{
    // 完成系统调用。
    if(exitsyscallfast()) {
        return;
    }

    // 尝试获取 P 完成任务。
    runtime·mcall(exitsyscall0);
}

static bool exitsyscallfast(void)
{
    // 检查 P 是否还在（如果 P 被 sysmon 拿走，status = Pidle）。
    if(m->p && m->p->status == Psyscall && runtime·cas(&m->p->status, Psyscall,
        Prunning)) {
        m->mcache = m->p->mcache;
        m->p->m = m;
        return true;
    }

    // 尝试获取闲置的 P。
    m->p = nil;
    if(runtime·sched·pidle) {
        p = pidleget();
        if(p) {

```

```

        acquirep(p);
        return true;
    }
}

return false;
}

static void exitsyscall0(G *gp)
{
    gp->status = Grunnable;
    gp->m = nil;
    m->curg = nil;

    // 尝试获取空闲的 P。
    p = pidleget();

    // 失败, 将 G 放回全局队列。
    if(p == nil)
        globrunqput(gp);

    // 成功, 关联 P, 完成后续工作。
    if(p) {
        acquirep(p);
        execute(gp); // Never returns.
    }

    // 检查是否有 locked G 需要处理。
    if(m->lockedg) {
        stoplockedm();
        execute(gp); // Never returns.
    }

    // 休眠 M, 直到唤醒后继续执行任务调度。
    stopm();
    schedule(); // Never returns.
}

```

可主动调用 `runtime/Gosched` 中断当前任务, 将其放回全局队列, 等待下次被其他 M 执行。

## proc.c

```

void runtime·gosched(void)
{
    runtime·mcall(runtime·gosched0);
}

```

```

void runtime·gosched0(G *gp)
{
    gp->status = Grunnable;
    gp->m = nil;
    m->curg = nil;

    // 将当前 G 放回全局队列。
    globrunqput(gp);

    // M 继续处理其他任务。
    schedule();
}

```

与 `gosched` 类似的还有 `park` 函数，区别在于 `park` 没有将 `G` 放回队列，一直处于暂停状态（注意不是 `M` 被暂停）。

#### proc.c

```

void runtime·park(bool(*unlockf)(G*, void*), void *lock, int8 *reason)
{
    m->waitlock = lock;
    m->waitunlockf = unlockf;
    g->waitreason = reason;
    runtime·mcall(park0);
}

static void park0(G *gp)
{
    gp->status = Gwaiting;
    gp->m = nil;
    m->curg = nil;

    // 当前 M 去执行其他任务，该函数不会返回。
    schedule();
}

```

只有调用 `ready` 函数，才能将其放回队列，得以继续执行。这种设计被用于并发垃圾清理和 `channel` 实现。

#### proc.c

```

void runtime·ready(G *gp)
{
    gp->status = Grunnable;

    // 放回本地队列。
    runqput(m->p, gp);
}

```



}

### 3.6 系统监控

系统监控线程 `sysmon` 完成如下三个工作:

- 将长时间没有处理的 `netpoll` 结果添加到全局队列。
- 收回长时间处于 `Psyscall` 状态的 `P`。
- 向长时间运行的 `G` 发出抢占调度通知。

系统监控积极回收 `P`, 让处于等待状态的 `G` 任务得以运行, 这是一种公平策略。

`proc.c`

```
static void sysmon(void)
{
    for(;;) {
        // 暂停。
        runtime·usleep(delay);

        // 处理网络操作结果。
        lastpoll = runtime·atomicload64(&runtime·sched·lastpoll);
        now = runtime·nanotime();
        if(lastpoll != 0 && lastpoll + 10*1000*1000 < now) {
            runtime·cas64(&runtime·sched·lastpoll, lastpoll, now);

            // 获取多个结果 (链表)。
            gp = runtime·netpoll(false); // non-blocking

            if(gp) {
                // 添加到全局队列。
                injectglist(gp);
            }
        }

        // 1. 收回长时间处于系统调用阻塞的 P。
        // 2. 向长时间运行的 G 任务发出抢占通知。
        if(retake(now))
            idle = 0;
        else
            idle++;
    }
}

static uint32 retake(int64 now)
```

```

{
    // 检查所有的 P。
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        s = p->status;

        if(s == Psyscall) {
            // 没收长时间处于系统调用阻塞的 P（有其他待运行任务）。
            if(p->runqhead == p->runqtail &&
                runtime·atomicload(&runtime·sched·nm spinning) +
                runtime·atomicload(&runtime·sched·npidle) > 0 &&
                pd->syscallwhen + 10*1000*1000 > now)
                continue;

            // 通过修改 P 状态，让 exitsyscallfast 中的判断失效。
            if(runtime·cas(&p->status, s, Pidle)) {
                handoffp(p);
            }
        } else if(s == Prunning) {
            // 向长时间运行的 G 发出抢占通知。
            if(pd->schedwhen + 10*1000*1000 > now)
                continue;
            preemptone(p);
        }
    }
    return n;
}

```

所谓抢占通知，不过是在 G 上设置标志而已。

## proc.c

```

static bool preemptone(P *p)
{
    mp = p->m;
    gp = mp->curg;

    gp->preempt = true;
    gp->stackguard0 = StackPreempt;
    return true;
}

```

编译器在非内联函数头部插入 `morestack` 调用，这是引发抢占调度的关键。

## asm\_amd64.s

```

TEXT runtime·morestack(SB),NOSPLIT,$0-0
    CALL    runtime·newstack(SB)

```

RET

## stack.c

```
void runtime·newstack(void)
{
    // 检查抢占标志。
    if(gp->stackguard0 == (uintptr)StackPreempt) {
        if(oldstatus != Grunning || m->locks || m->mallocing || m->gcing ||
            m->p->status != Prunning) {
            gp->stackguard0 = gp->stackguard;
            gp->status = oldstatus;
            runtime·gogo(&gp->sched);    // never return
        }

        gp->status = oldstatus;
        runtime·gosched0(gp);    // 暂停当前任务，切换其他任务执行。never return
    }
}
```

换句话说，只有内部包含非内联函数调用才有机会触发抢占式调度。另外，发出抢占调度的不仅仅是 `sysmon`，在很多地方都有类似操作。

### 3.7 状态输出

使用用环境变量 `CODEBUG="schedtrace=xxx"` 输出调度器跟踪信息。

```
$ GOMAXPROCS=2 GODEBUG="schedtrace=1000" ./test
```

The diagram illustrates the mapping of kernel parameters to their meanings in the 'top' command output. It shows the following parameters and their corresponding meanings:

- `gomaxprocs=2`: P 总数。
- `idleprocs=2`: 空闲 P 数量。
- `threads=5`: M 总数。
- `idlethreads=2`: 空闲 M 数量。
- `runqueue=0`: 全局队列任务数量。
- `[0 0]`: 各 P 本地队列。

Additionally, the `SCHED 1003ms` output is annotated with `+-` and a vertical dashed line, indicating it represents `跟踪耗时。`

更详细的信息,需指定 "scheddetail=1"。

```
$ GOMAXPROCS=2 GODEBUG="schedtrace=1000,scheddetail=1" ./test
```

具体代码请参考 `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。

架构设计上，类似 FIFO 队列，多个 G 排队等待收发操作。如果是同步模式，就从排队的链表中获取一个能直接与之交换数据的对象；异步模式则围绕着数据缓冲区空位排队。

实际上，channel 并没有想象的那么神秘，一样是基于锁、链表、环状队列的实现。核心就是如果没有匹配的通讯对象或空槽，就让 G 休眠 (park)，然后在条件满足时唤醒。

SudoG 是对 G 的包装，WaitQ 则是 SudoG 排队链表。

chan.h

```
struct SudoG
{
    G*      g;
    uint32* selectdone;
    SudoG*  link;
    int64   releasetime;
    byte*   elem;        // data element
};

struct WaitQ
{
    SudoG*  first;
    SudoG*  last;
};
```

Hchan 除发送和接收排队链表外，还用 qcount、sendx、recvx 构建缓冲槽环状队列。

chan.h

```
struct Hchan
{
    uintgo  qcount;           // 已使用槽数量。
    uintgo  dataqsiz;         // 缓冲区槽数量，同步模式为 0。
    uint16  elemsize;         // 数据类型长度。
    bool    closed;           // 关闭标记。
    Type*   elemtype;         // 数据类型。
    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。
    Lock;
```

```
};
```

创建 `channel` 时，需要给出缓冲槽的数量，同步模式为 0。

## chan.goc

```
static Hchan* makechan(ChanType *t, int64 hint)
{
    Hchan *c;
    Type *elem;

    elem = t->elem;

    // 数据类型长度不能超过 64KB。
    if(elem->size >= (1<<16))
        runtime.throw("makechan: invalid channel element type");

    // 为 channel 和缓冲槽分配内存。
    c = (Hchan*)runtime.mallocgc(sizeof(*c) + hint*elem->size,
                                   (uintptr)t | TypeInfo_Chan, 0);
    c->elemsize = elem->size;
    c->elemtype = elem;
    c->dataqsiz = hint;

    return c;
}
```

## 4.1 Send

同步模式需从接收链表里找到接收者，然后交换数据。而异步模式，则要看是否有空数据槽可用。当无法获取接收者或空槽时，将当前 `G` 包装成 `SudoG`，加入等待发送链表，直到被唤醒。

参数 `eq` 为待发送数据，`block` 是否为阻塞模式。

## chan.goc

```
static bool chansend(ChanType *t, Hchan *c, byte *ep, bool block, void *pc)
{
    // 向 nil channel 发送数据，阻塞。
    if(c == nil) {
        USED(t);
        if(!block)
            return false;
        runtime.park(nil, nil, "chan send (nil chan)");
    }
}
```

```

        return false; // not reached
    }

    // 加锁。
    runtime.lock(c);

    // 向 closed channel 发送数据, 出错。
    if(c->closed)
        goto closed;

    // 如果缓冲槽大于 0, 异步模式。
    if(c->dataqsiz > 0)
        goto asynch;

    // 从链表里找一个等待接收的 SudoG。
    sg = dequeue(&c->recvq);
    if(sg != nil) {
        // 解除锁定。
        runtime.unlock(c);

        gp = sg->g;
        gp->param = sg;

        // 将数据拷贝给接收方。
        if(sg->elem != nil)
            c->elemtype->alg->copy(c->elemsize, sg->elem, ep);

        // 唤醒接收 G。
        runtime.ready(gp);
        return true;
    }

    if(!block) {
        runtime.unlock(c);
        return false;
    }

    // 将当前 G 封装为 SudoG。
    msg.elem = ep;
    msg.g = g;
    msg.selectdone = nil;
    g->param = nil;

    // 添加到等待发送链表, 阻塞直到被唤醒。
    enqueue(&c->sendq, &msg);
    runtime.parkunlock(c, "chan send");

    // 被意外唤醒, 出错 (通常是 channel closed。如被接收者唤醒, g->param != nil)。

```

```

if(g->param == nil) {
    runtime·lock(c);
    if(!c->closed)
        runtime·throw("chansend: spurious wakeup");
    goto closed;
}

```

```

return true;

```

```

asynch:

```

```

    if(c->closed)
        goto closed;

```

```

// 如果缓冲槽已满。

```

```

if(c->qcount >= c->dataqsiz) {
    if(!block) {
        runtime·unlock(c);
        return false;
    }

```

```

// 包装当前 G，加入等待发送队列，阻塞。

```

```

mysg.g = g;
mysg.elem = nil;
mysg.selectdone = nil;
enqueue(&c->sendq, &mysg);
runtime·parkunlock(c, "chan send");

```

```

// 被唤醒后，重试。

```

```

runtime·lock(c);
goto asynch;

```

```

}

```

```

// 如果缓冲槽有空位，将数据拷贝到缓冲区。

```

```

c->elemtype->alg->copy(c->elemsize, chanbuf(c, c->sendx), ep);

```

```

// 调整缓冲槽队列索引。

```

```

if(++c->sendx == c->dataqsiz)
    c->sendx = 0;
c->qcount++;

```

```

// 缓冲槽已经有数据了，尝试从接收队列中唤醒一个 SudoG。

```

```

sg = dequeue(&c->recvq);

```

```

if(sg != nil) {

```

```

    gp = sg->g;
    runtime·unlock(c);
    runtime·ready(gp);

```

```

} else

```

```

    runtime·unlock(c);

```

```

    return true;

closed:
    runtime.unlock(c);
    runtime.panicstring("send on closed channel");
    return false; // not reached
}

```

虽然将同步和异步放在一个函数里，但不算复杂。

- 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: 如有空位，拷贝数据到缓冲区。否则排队、阻塞。

## 4.2 Receive

和接收操作类似，规则如下：

- 从 nil channel 接收数据，阻塞。
- 从 closed channel 接收数据，直接返回。
- 同步接收: 如有发送者，交换数据。否则排队、阻塞。
- 异步接收: 如缓冲区不为空，拷贝数据。否则排队、阻塞。

参数 `ep` 保存接收到的数据 (`SudoG.elem = ep`)，`received` 表示接收是否成功。

### chan.goc

```

static bool chanrecv(ChanType *t, Hchan* c, byte *ep, bool block, bool *received)
{
    // 从 nil channel 接收数据，阻塞。
    if(c == nil) {
        USED(t);
        if(!block)
            return false;
        runtime.park(nil, nil, "chan receive (nil chan)");
        return false; // not reached
    }

    // 加锁。
    runtime.lock(c);

```



```

// 异步接收。
if(c->dataqsiz > 0)
    goto asynch;

// 从 closed channel 接收数据, 直接返回。
if(c->closed)
    goto closed;

// 找一个等待发送的 SudoG。
sg = dequeue(&c->sendq);
if(sg != nil) {
    runtime.unlock(c);

    // 拷贝数据。
    if(ep != nil)
        c->elemtype->alg->copy(c->elemsize, ep, sg->elem);

    // 唤醒发送方。
    gp = sg->g;
    gp->param = sg;
    runtime.ready(gp);

    // 成功接收标记。
    if(received != nil)
        *received = true;

    return true;
}

if(!block) {
    runtime.unlock(c);
    return false;
}

// 将当前 G 包装成 SudoG, 加入等待接收链表, 阻塞。
mysg.elem = ep;
mysg.g = g;
mysg.selectdone = nil;
g->param = nil;
enqueue(&c->recvq, &mysg);
runtime.parkunlock(c, "chan receive");

// 意外唤醒。(通常是 channel closed。如被发送方唤醒, g->param != nil)
if(g->param == nil) {
    runtime.lock(c);
    if(!c->closed)
        runtime.throw("chanrecv: spurious wakeup");
}

```

```

        goto closed;
    }

    // 被发送方唤醒, 成功接收标记。
    if(received != nil)
        *received = true;

    return true;

```

asynch:

```

    // 如缓冲槽没可用数据。
    if(c->qcount <= 0) {
        if(c->closed)
            goto closed;

        if(!block) {
            runtime·unlock(c);
            if(received != nil)
                *received = false;
            return false;
        }

        // 包装当前 G, 加入等待链表, 阻塞。
        mysg.g = g;
        mysg.elem = nil;
        mysg.selectdone = nil;
        enqueue(&c->recvq, &mysg);
        runtime·parkunlock(c, "chan receive");

        // 被唤醒, 重试。
        runtime·lock(c);
        goto asynch;
    }

    // 从缓冲槽拷贝数据。
    if(ep != nil)
        c->elemtype->alg->copy(c->elemsize, ep, chanbuf(c, c->recvx));
    c->elemtype->alg->copy(c->elemsize, chanbuf(c, c->recvx), nil);

    // 调整缓冲槽队列索引。
    if(++c->recvx == c->dataqsiz)
        c->recvx = 0;
    c->qcount--;

    // 有新空位, 尝试唤醒一个处于等待状态的异步发送者。
    sg = dequeue(&c->sendq);
    if(sg != nil) {
        gp = sg->g;

```

```

        runtime·unlock(c);
        runtime·ready(gp);
    } else
        runtime·unlock(c);

    if(received != nil)
        *received = true;

    return true;

closed:
    if(ep != nil)
        c->elemtype->alg->copy(c->elemsize, ep, nil);

    if(received != nil)
        *received = false;

    runtime·unlock(c);
    return true;
}

```

## 4.3 Select

Channel Select 设计是个让人有些不好评价的东西。

每个 case 语句都被编译器转换为 Scase 对象注册到 Select.scase 数组，而 Select 对象则维护着整个 select 代码块逻辑，这其中有两个很重要的排序规则。

- pollorder: 对顺序索引号洗牌。遍历时，返回随机序号。
- lockorder: 将地址相同 channel 排列到相邻位置，避免对同一 channel 重复锁定。

### chan.h

```

struct Scase
{
    SudoG    sg;           // must be first member (cast to Scase)
    Hchan*   chan;         // chan
    byte*    pc;           // return pc
    uint16   kind;
    uint16   so;           // vararg of selected bool
    bool*    receivedp;    // pointer to received bool (recv2)
};

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)
};

```

创建函数会为相关成员数组分配内存。

## chan.goc

```

static Select* newselect(int32 size)
{
    int32 n;
    Select *sel;

    n = 0;
    if(size > 1)
        n = size-1;

    // 一次性分配数组, 其中包括 scase、lockorder、pollorder 数组。
    sel = runtime·mal(sizeof(*sel) +
        n*sizeof(sel->scase[0]) +
        size*sizeof(sel->lockorder[0]) +
        size*sizeof(sel->pollorder[0]));

    sel->tcase = size;
    sel->ncase = 0;
    sel->lockorder = (void*)(sel->scase + size);
    sel->pollorder = (void*)(sel->lockorder + size);

    return sel;
}

```

内存布局:

```

+-----+-----+-----+-----+
| select | scase array | lockorder array | pollorder array |
+-----+-----+-----+-----+

```

注册函数包括 send、receive、default 三种 case。

## chan.goc

```

static void selectsend(Select *sel, Hchan *c, void *pc, void *elem, int32 so)
{
    int32 i;
    Scase *cas;

```

```

    i = sel->ncase;
    sel->ncase = i+1;
    cas = &sel->scase[i];

    cas->pc = pc;
    cas->chan = c;
    cas->so = so;
    cas->kind = CaseSend;
    cas->sg.elem = elem;
}

static void selectrecv(Select *sel, Hchan *c, void *pc, void *elem, bool *received,
int32 so)
{
    int32 i;
    Scase *cas;

    i = sel->ncase;
    sel->ncase = i+1;
    cas = &sel->scase[i];

    cas->pc = pc;
    cas->chan = c;
    cas->so = so;
    cas->kind = CaseRecv;
    cas->sg.elem = elem;
    cas->receivedp = received;
}

static void selectdefault(Select *sel, void *callerpc, int32 so)
{
    int32 i;
    Scase *cas;

    i = sel->ncase;
    sel->ncase = i+1;
    cas = &sel->scase[i];

    cas->pc = callerpc;
    cas->chan = nil;
    cas->so = so;
    cas->kind = CaseDefault;
}

```

注册过程很简单，ncase 表示注册序号，不能超过 tcase 限制。关键还是看 select 如何随机选择可用 channel case。

## chan.goc

```

static void* selectgo(Select **selp)
{
    sel = *selp;

    // 按顺序往 pollorder 填充序号。
    for(i=0; i<sel->ncase; i++)
        sel->pollorder[i] = i;

    // 对 pollorder 洗牌, 这样 pollorder[x] 返回的是一个“随机”序号。
    for(i=1; i<sel->ncase; i++) {
        o = sel->pollorder[i];
        j = runtime·fastrand1()%(i+1);
        sel->pollorder[i] = sel->pollorder[j];
        sel->pollorder[j] = o;
    }

    // 对 lockorder 进行排序, 确保相同地址的 channel 处于相邻位置。
    // 需要对所有 case channel 进行加锁, 以遍历它们的状态。
    // 不同 case 可能使用同一 channel, 如此可避免 sellock 对同一个 channel 重复加锁。
    for(i=0; i<sel->ncase; i++) {
        j = i;
        c = sel->scase[j].chan;
        while(j > 0 && sel->lockorder[k=(j-1)/2] < c) {
            sel->lockorder[j] = sel->lockorder[k];
            j = k;
        }
        sel->lockorder[j] = c;
    }
    for(i=sel->ncase; i-->0; ) {
        c = sel->lockorder[i];
        sel->lockorder[i] = sel->lockorder[0];
        j = 0;
        for(;;) {
            k = j*2+1;
            if(k >= i)
                break;
            if(k+1 < i && sel->lockorder[k] < sel->lockorder[k+1])
                k++;
            if(c < sel->lockorder[k]) {
                sel->lockorder[j] = sel->lockorder[k];
                j = k;
                continue;
            }
            break;
        }
        sel->lockorder[j] = c;
    }
}

```

```

}

// 按 lockorder 顺序对所有 channel 加锁。
// select 必须获得所有 case channel 可操作状态。
sellock(sel);

loop:
// pass 1 - 遍历检查所有 case。
dfl = nil;
for(i=0; i<sel->ncase; i++) {
    // 使用 pollorder 随机返回某个 case。
    o = sel->pollorder[i];
    cas = &sel->scase[o];
    c = cas->chan;

    // 依据 kind, 进行对应读写操作。
    switch(cas->kind) {
    case CaseRecv:
        if(c->dataqsiz > 0) {
            if(c->qcount > 0)
                goto asyncrecv;
        } else {
            sg = dequeue(&c->sendq);
            if(sg != nil)
                goto syncrecv;
        }
        if(c->closed)
            goto rclose;
        break;

    case CaseSend:
        if(c->closed)
            goto sclose;
        if(c->dataqsiz > 0) {
            if(c->qcount < c->dataqsiz)
                goto asyncsend;
        } else {
            sg = dequeue(&c->recvq);
            if(sg != nil)
                goto syncsend;
        }
        break;

    case CaseDefault:
        dfl = cas;
        break;
    }
}
}

```

```

// 如果没有可用 channel, 那么尝试执行 case default。
if(dfl != nil) {
    selunlock(sel);
    cas = dfl;
    goto retc;
}

// pass 2 - 没有可用 channel 和 default,
//           将当前 SudoG 保存到所有 channel 队列, 以便在某个 channel 可操作时被唤醒。
done = 0;
for(i=0; i<sel->ncase; i++) {
    o = sel->pollorder[i];
    cas = &sel->scase[o];
    c = cas->chan;

    // 将 case.sg 换成当前 G。
    // 当被 channel 唤醒时, sg 会作为 g->param 传回。
    // 因为 sg 是 case 第一个字段, 地址相同, 因此可以转换回 Scase 对象。
    sg = &cas->sg;
    sg->g = g;
    sg->selectdone = &done;

    switch(cas->kind) {
    case CaseRecv:
        enqueue(&c->recvq, sg);
        break;

    case CaseSend:
        enqueue(&c->sendq, sg);
        break;
    }
}

// 当前 G 休眠, 等待唤醒。
g->param = nil;
runtime·park(sel·parkcommit, sel, "select");

// 被唤醒后, 再次加锁。
sellock(sel);
sg = g->param;

// pass 3 - 将 G 从其他未命中的 channel 等待队列中清除。
for(i=0; i<sel->ncase; i++) {
    cas = &sel->scase[i];
    if(cas != (Scase*)sg) {
        c = cas->chan;
    }
}

```



```

        if(cas->kind == CaseSend)
            dequeueg(&c->sendq);
        else
            dequeueg(&c->recvq);
    }
}

// 意外唤醒, 重试 (如果被 channel 唤醒, g->param != nil)。
if(sg == nil)
    goto loop;

// 被 channel 唤醒, 将 sg 转换回 Scase 对象。
cas = (Scase*)sg;
c = cas->chan;

selunlock(sel);
goto retc;

asyncrecv:
    // can receive from buffer

asyncsend:
    // can send to buffer

syncrecv:
    // can receive from sleeping sender (sg)

rclose:
    // read at end of closed channel

syncsend:
    // can send to sleeping receiver (sg)

retc:
    // return pc corresponding to chosen case.
    // Set boolean passed during select creation
    // (at offset selp + cas->so) to true.
    // If cas->so == 0, this is a reflect-driven select and we
    // don't need to update the boolean.
    pc = cas->pc;
    if(cas->so > 0) {
        as = (byte*)selp + cas->so;
        *as = true;
    }

    // 释放 Select。
    runtime·free(sel);
    return pc;

```

```

sclose:
    // send on closed channel
    selunlock(sel);
    runtime·panicstring("send on closed channel");
    return nil; // not reached
}

```

虽然删除了很多代码，但整体流程还是很清楚的。

1. 对 `pollorder` 洗牌，构成 "随机" 选择条件。
2. 对 `lockorder` 排序，避免 `sellock` 对同一 `channel` 重复锁定。
3. 遍历 `case` 查找可用 `channel`。因为通过 `pollorder` 返回，所以就是随机选择。
4. 遍历时对单个 `channel` 的操作和前面两节内容相同，此处不做赘述。
5. 如所有 `channel` 都不可用，就尝试执行 `case default`。
6. 如没有 `default`，就将所有 `case.sg.g` 换成当前 `G`，放到所属 `channel` 排队链表中。
7. 一旦某个 `channel` 可用，就会唤醒当前 `G`，自然就可以找到可用 `case`。

在遍历前获取所有 `channel` 锁，确保并发安全，这个代价确实不小。

## chan.goc

```

static void sellock(Select *sel)
{
    Hchan *c, *c0;

    c = nil;
    for(i=0; i<sel->ncase; i++) {
        c0 = sel->lockorder[i]; // channel

        // 如果前一个 channel 和当前地址相同，那么就表示已经加过锁了。
        if(c0 && c0 != c) {
            c = sel->lockorder[i];
            runtime·lock(c);
        }
    }
}

```

另外，如果在 `select` 语句外面套上 `for` 循环，那就意味着每次都要创建 `Select` 对象，完成注册、洗牌、排序、选择和释放等一大堆操作。

```

func main() {
    a, b := make(chan bool), make(chan bool)
}

```

```

    for {
        select {
            case <-a:
            case <-b:
            }
        }
    }
}

```

汇编结果:

```
$ go build -gcflags "-S"
```

```
--- prog list "main" ---
```

```

0023 MOVL    $2,(SP)
0024 PCDATA  $0,$16
0025 CALL    ,runtime.newselect+0(SB)    // 创建 Select
0034 CALL    ,runtime.selectrecv+0(SB)   // 注册
0046 CALL    ,runtime.selectrecv+0(SB)   // 注册
0055 CALL    ,runtime.selectgo+0(SB)     // 执行 (goto rect: runtime.free(sel))
0057 JMP     ,23                         // 循环跳转
0058 RET     ,

```

## 5. Defer

通过一个简单示例，看看 defer 内部执行过程。

```
package main

import "sync"

var lock sync.Mutex

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

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

用反汇编观察编译器如何处理 defer 调用。

```
(gdb) disass
Dump of assembler code for function main.test:
=> 0x0000000000002000 <+0>:      mov     rcx,QWORD PTR gs:0x8a0
    0x0000000000002009 <+9>:      cmp     rsp,QWORD PTR [rcx]
    0x000000000000200c <+12>:     ja      0x2015 <main.test+21>
    0x000000000000200e <+14>:     call   0x21db0 <runtime.morestack00_noctxt>
    0x0000000000002013 <+19>:     jmp     0x2000 <main.test>
    0x0000000000002015 <+21>:     sub     rsp,0x8
    0x0000000000002019 <+25>:     lea     rbx,ds:0x65d28
    0x0000000000002021 <+33>:     mov     QWORD PTR [rsp],rbx
    0x0000000000002025 <+37>:     call   0x23470 <sync.(*Mutex).Lock>
    0x000000000000202a <+42>:     lea     rbx,ds:0x65d28
    0x0000000000002032 <+50>:     mov     QWORD PTR [rsp],rbx
    0x0000000000002036 <+54>:     mov     ecx,0x40680
    0x000000000000203b <+59>:     push    rcx
    0x000000000000203c <+60>:     push    0x8
    0x000000000000203e <+62>:     call   0xeaa0 <runtime.deferproc>
    0x0000000000002043 <+67>:     pop     rcx
    0x0000000000002044 <+68>:     pop     rcx
    0x0000000000002045 <+69>:     test    rax,rax
    0x0000000000002048 <+72>:     jne     0x2055 <main.test+85>
    0x000000000000204a <+74>:     nop
    0x000000000000204b <+75>:     call   0xeb10 <runtime.deferreturn>
    0x0000000000002050 <+80>:     add     rsp,0x8
    0x0000000000002054 <+84>:     ret
```

```

0x00000000000002055 <+85>:    nop
0x00000000000002056 <+86>:    call    0xeb10 <runtime.deferreturn>
0x0000000000000205b <+91>:    add     rsp,0x8
0x0000000000000205f <+95>:    ret

```

End of assembler dump.

不算太复杂，核心就是 `deferproc` 和 `deferreturn` 两个函数。

## panic.c

```

uintptr runtime·deferproc(int32 siz, FuncVal *fn, ...)
{
    Defer *d;

    // 新建 Defer 对象。
    d = newdefer(siz);

    // 保存 fn 等参数。
    d->fn = fn;
    d->pc = runtime·getcalle rpc(&siz);

    // 保存第一个参数地址。这个很重要，因为这个地址记录了所在栈帧（main.test）的信息。
    if(thechar == '5')
        d->argp = (byte*)&fn+2; // skip caller's saved link register
    else
        d->argp = (byte*)&fn+1;

    // 将所有参数拷贝到新建的 Defer 对象里。
    runtime·memmove(d->args, d->argp, d->siz);

    return 0;
}

```

和以往一样，`Defer` 对象会通过可复用链表缓存。

## runtime.h

```

struct P
{
    Defer* deferpool[5]; // pool of available Defer structs of different sizes
};

```

## panic.c

```

static Defer* newdefer(int32 siz)
{
    d = nil;
    sc = DEFERCLASS(siz);
}

```

```

// 从 P.deferpool 链接数组提取可复用对象。
if(sc < nelem(p->deferpool)) {
    p = m->p;
    d = p->deferpool[sc];
    if(d)
        p->deferpool[sc] = d->link;
}

// 如果链表为空或 size 超出, 则新建。
if(d == nil) {
    // deferpool is empty or just a big defer
    total = TOTALSIZE(siz);
    d = runtime·malloc(total);
}

d->siz = siz;
d->special = 0;

// 新建的 Defer 被添加到 G.defer 链表。
d->link = g->defer;
g->defer = d;

return d;
}

```

所有 `defer` 定义都被转换成 `Defer` 对象保存在 `G.defer` 链表中。编译器会在函数尾部插入 `"call deferreturn"` 指令, 确保在退出前调用这些 `Defer` 对象。

编译器在这里玩了个技巧, 用汇编指令重复执行 `"call deferreturn"`, 以便执行当前函数里多个 `defer` 定义。

## panic.c

```

void runtime·deferreturn(uintptr arg0)
{
    Defer *d;
    byte *argp;
    FuncVal *fn;

    // 从 G.defer 获取 Defer 对象。
    d = g->defer;
    if(d == nil)
        return;

    // 第一个参数地址。
    argp = (byte*)&arg0;

```

```

// 在 deferproc 里提过, d.argp 实际保存了 defer 所在函数堆栈帧的地址,
// 那么匹配这两个地址, 就可以知道是否属于同一个函数栈帧。如果不同, 那么这
// 个 Defer 对象必然不属于当前函数, 以此中断 call deferreturn 循环。
if(d->argp != argp)
    return;

// 将 Defer 参数入栈。
runtime.memmove(argp, d->args, d->siz);
fn = d->fn;

// 释放 Defer 对象。
g->defer = d->link;
freedefer(d);

// 执行 defer.fn 函数。
// 两个参数分别是 fn 和 arg0 地址。
runtime.jumpdefer(fn, argp);
}

```

汇编函数 `jumpdefer` 虽然只有几行, 但是很有意思。

### asm\_amd64.s

```

// void jumpdefer(fn, sp);
// called from deferreturn.
// 1. pop the caller
// 2. sub 5 bytes from the callers return
// 3. jmp to the argument
TEXT runtime.jumpdefer(SB), NOSPLIT, $0-16
    MOVQ    8(SP), DX    // fn, 也就是实际要执行的 defer 函数。
    MOVQ    16(SP), BX   // argp, 也就是 callreturn 参数 arg0 地址。
    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

```

第三行 `"-8(BX)"` 是什么意思?

`BX` 保存着 `arg0` 的地址, 该参数在 `"call deferreturn"` 前保存在栈顶, 而 `call` 指令又会 `"PUSH IP"` (`main.test 0x2050`), 所以这个结果就是将 `0x2050` 入栈。紧接着减去 `"call deferreturn"` 指令长度 5, 正好就是 `0x204b`。

```

0x000000000000204b <+75>: call    0xeb10 <runtime.deferreturn>
0x0000000000002050 <+80>: add     rsp,0x8

```

这个地址最终由 `deferreturn RET` 指令 "POP IP", 形成循环调用。配合对 `defer.argp` 所在栈帧地址检查, 就可以完成当前函数多个 `defer` 定义的调用。

至于 `Defer` 对象的释放过程, 无需多言。

## panic.c

```
static void freedefer(Defer *d)
{
    int32 sc;
    P *p;

    if(d->special)
        return;
    sc = DEFERCLASS(d->siz);
    if(sc < nelem(p->deferpool)) {
        p = m->p;
        d->link = p->deferpool[sc];
        p->deferpool[sc] = d;
        // No need to wipe out pointers in argp/pc/fn/args,
        // because we empty the pool before GC.
    } else
        runtime.free(d);
}
```

回过头想想, 一个 `defer` 完整过程要处理缓存对象, 参数拷贝, 以及多次函数调用。显然比直接函数调用慢得多。

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

(gdb) disass

Dump of assembler code for function main.test:

```
0x0000000000002015 <+21>:    sub    rsp,0x8
0x0000000000002019 <+25>:    lea    rbx,ds:0x65d28
0x0000000000002021 <+33>:    mov    QWORD PTR [rsp],rbx
0x0000000000002025 <+37>:    call   0x23450 <sync.(*Mutex).Lock>
0x000000000000202a <+42>:    lea    rbx,ds:0x65d28
0x0000000000002032 <+50>:    mov    QWORD PTR [rsp],rbx
0x0000000000002036 <+54>:    call   0x23560 <sync.(*Mutex).Unlock>
0x000000000000203b <+59>:    add    rsp,0x8
0x000000000000203f <+63>:    ret
```

End of assembler dump.



用 **benchmark** 测试, 性能差异在 2x 以上。如果这个函数被大循环多次调用, 其结果就非常明显了。因此, 合理使用 **defer** 也是算法优化的一部分。

```
var lock sync.Mutex

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

func test2() {
    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++ {
        test2()
    }
}
```

BenchmarkTest	50000000	43.6 ns/op
BenchmarkTestDefer	20000000	102 ns/op

还有一种情况就是用户调用 `runtime/Goexit` 终止 goroutine 执行。

## panic.c

```
void runtime·Goexit(void)
{
    rundefer();
    runtime·goexit();
}

static void rundefer(void)
{
    Defer *d;

    while((d = g->defer) != nil) {
        g->defer = d->link;
        reflect·call(d->fn, (byte*)d->args, d->siz, d->siz);
    }
}
```

```
    freedefers(d);  
  }  
}
```

这个时候只需依次处理掉 `G.defer` 链表里所有 `Defer` 对象即可，无需判断所属堆栈帧。

## 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。	
-ldflags	传递给 5l/6l/8l 链接器的参数。	
-work	查看编译临时目录。	
-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。	

更多参数:

```
go help build
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 workspace 目录。

参数	说明	示例
-d	仅下载，不执行安装命令。	
-t	下载执行测试所需的依赖包。	
-u	更新包，包括其依赖包。	
-x	查看并执行命令。	

## 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
pkg/runtime

... ..

---
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
```

注意：跨平台编译不支持 `CGO`，默认 `CGO_ENABLED=0`。

## 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/pkg/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。

### 1. Test

使用 `testing.T` 相关方法决定测试状态。

#### testing.T

方法	说明	其他
<code>Fail</code>	标记失败, 但继续执行该测试函数。	
<code>FailNow</code>	失败, 立即停止当前测试函数。	
<code>Log</code>	输出信息。仅在失败或 <code>-v</code> 参数时输出。	<code>Logf</code>
<code>SkipNow</code>	跳过当前测试函数。	<code>Skipf = SkipNow + Logf</code>
<code>Error</code>	<code>Fail + Log</code>	<code>Errorf</code>
<code>Fatal</code>	<code>FailNow + Log</code>	<code>Fatalf</code>

#### 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
<code>-c</code>	仅编译，不执行测试。	
<code>-v</code>	显示所有测试函数执行细节。	
<code>-run regex</code>	执行指定的测试函数。（正则表达式）	
<code>-parallel n</code>	并发执行测试函数。（默认：GOMAXPROCS）	
<code>-timeout t</code>	单个测试超时时间。	<code>-timeout 2m30s</code>

```

$ 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

```

## 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
<code>-bench regex</code>	执行指定性能测试函数。(正则表达式)	
<code>-benchmem</code>	输出内存统计信息。	
<code>-benchtime t</code>	设置每个性能测试运行时间。	<code>-benchtime 1m30s</code>
<code>-cpu</code>	设置并发测试。默认 <code>GOMAXPROCS</code> 。	<code>-cpu 1,2,4</code>

```
$ go test -v -bench .

=== RUN TestSum
--- PASS: TestSum (2.00 seconds)
=== RUN TestTimeout
--- PASS: TestTimeout (5.00 seconds)
PASS

BenchmarkSum      1000000000    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。

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。

```
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)

    // code ...
}
```

go test

参数	说明
-blockprofile block.out	goroutine 阻塞。
-blockprofrate n	超出该参数设置时间的阻塞才被记录。单位：纳秒
-cpuprofile cpu.out	
-memprofile mem.out	内存分配。
-memprofrate n	超出该参数设置的内存分配才被记录。单位：字节，默认 512KB。

输出分析结果。

```
$ go tool pprof -text test cpu.out
```

Total: 443 samples

440	99.3%	99.3%	443	100.0%	net.sotypeToNet
1	0.2%	99.5%	1	0.2%	net.(*UDPConn).WriteToUDP
1	0.2%	99.8%	1	0.2%	runtime.FixAlloc_Free
1	0.2%	100.0%	1	0.2%	strconv.ParseInt
0	0.0%	100.0%	443	100.0%	net.unixSocket

0	0.0%	100.0%	53	12.0%	runtime.InitSizes
0	0.0%	100.0%	321	72.5%	runtime.cmalloc
0	0.0%	100.0%	1	0.2%	runtime.convT2E
0	0.0%	100.0%	1	0.2%	runtime.gc
0	0.0%	100.0%	1	0.2%	runtime.printeface
0	0.0%	100.0%	136	30.7%	syscall.Kevent
0	0.0%	100.0%	97	21.9%	syscall.Read
0	0.0%	100.0%	251	56.7%	syscall.Syscall
0	0.0%	100.0%	136	30.7%	syscall.Syscall6
0	0.0%	100.0%	154	34.8%	syscall.Write
0	0.0%	100.0%	136	30.7%	syscall.kevent

共计 443 次采样，大约每秒 100 次，耗时 5 秒左右。各列数据含义：

```

+- 当前函数采样次数（不包括它调用的其他函数）
|
|  +- 当前函数采样所占百分比
|  |
|  | 440 99.3% 99.3%
|  |
|  |  +- 当前函数及其所调用函数所占百分比
|  |  |
|  |  | 443 100.0% net.sotypeToNet
|  |  |
|  |  |  +- 当前函数及其所调用函数采样次数
|  |  |  |
|  |  |  +- 列表前几行（含当前行）累计所占百分比。

```

还可使用 `web` 参数在浏览器输出图形。(需安装 `graphviz`)

```
$ go tool pprof -web test cpu.out
```

内存记录文件输出格式类似。如不使用 `text/web` 参数，将进入交互命令模式。