

# Go 学习 笔记

第 2 版

好好学习 天天向上



# 前言

转眼用 Go 写程序已经两年了，一路惊喜，一路埋坑。老版笔记也是补丁摺补丁，不堪入目。早有心思重写，可一直被 Go 的不稳定状态所困扰，怕白费功夫。如今这 1.2 算是一个里程碑，终于着手扫尘出新。



新版重新规划了目录结构，将语言和工具分离开来，而且不再包含频繁变化的标准库内容。第二部分由源码剖析等高阶内容代替，以便了解更多底层实现信息。

内容力求简洁，以便在较短时间内完成阅读和复习。如没有大的意外，这个版本会坚持到下一个里程碑。

不定期更新，可到 <https://github.com/qyuheng/book> 下载最新版本。

联系方式：

- qyuheng@hotmail.com

雨痕 二〇一三年末于北京家中



## 更新

- 2012-01-11 开始学习 Go。
- 2012-01-15 完成第一版, 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 开始第二版, 基于 1.2。
- 2014-03-03 完成第一个校对版本。



# 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	12
1.4 引用类型	13
1.5 类型转换	14
1.6 字符串	14
1.7 指针	17
1.8 自定义类型	18
<b>第 2 章 表达式</b>	<b>20</b>
2.1 保留字	20
2.2 运算符	20
2.3 初始化	21
2.4 控制流	22
<b>第 3 章 函数</b>	<b>28</b>
3.1 函数定义	28
3.2 变参	29
3.3 返回值	29
3.4 匿名函数	31
3.5 延迟调用	33
3.6 错误处理	34
<b>第 4 章 数据</b>	<b>37</b>
4.1 Array	37
4.2 Slice	38
4.3 Map	43

4.4 Struct	45
<b>第 5 章 方法</b>	<b>50</b>
5.1 方法定义	50
5.2 匿名字段	51
5.3 方法集	52
5.4 表达式	53
<b>第 6 章 接口</b>	<b>56</b>
6.1 接口定义	56
6.2 执行机制	58
6.3 接口转换	59
6.4 接口技巧	61
<b>第 7 章 并发</b>	<b>62</b>
7.1 Goroutine	62
7.2 Channel	64
<b>第 8 章 包</b>	<b>72</b>
8.1 工作空间	72
8.2 源文件	72
8.3 包结构	73
8.4 文档	75
<b>第 9 章 进阶</b>	<b>77</b>
9.1 内存布局	77
9.2 指针陷阱	78
9.3 cgo	81
<b>第二部分 源码</b>	<b>90</b>
<b>1. Memory Allocator</b>	<b>91</b>
1.1 基本概念	93
1.2 初始化	97
1.3 分配流程	98

1.4 释放流程	106
1.5 其他	109
<b>2. Garbage Collector</b>	<b>113</b>
2.1 垃圾回收	113
2.2 释放内存	116
2.3 状态输出	119
<b>3. Goroutine Scheduler</b>	<b>125</b>
3.1 初始化	127
3.2 创建任务	132
3.3 执行任务	135
3.4 系统调用	142
3.5 系统监控	145
3.6 分段栈	148
3.7 状态输出	151
<b>4. Channel</b>	<b>152</b>
4.1 Channel	152
4.2 Send	153
4.3 Receive	156
<b>第三部分 附录</b>	<b>159</b>
<b>A. 工具</b>	<b>160</b>
1. 工具集	160
2. 条件编译	161
3. 跨平台编译	163
<b>B. 调试</b>	<b>164</b>
1. GDB	164
2. Data Race	164
<b>C. 测试</b>	<b>167</b>
1. Test	167

2. Benchmark	168
3. Example	169
4. Cover	170
5. PProf	171

# 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
)
```

## 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

类型	长度	默认值	说明
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	128		
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.c: runtime·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,±,,â,-,,
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。

6:	0110		
11:	1011		
<hr/>			
&:	0010 = 2	AND	都为 1。
:	1111 = 15	OR	至少一个为 1。

```
^:    1101 = 13    XOR    只能一个为 1。
&^:   0100 = 4     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)++
```

## 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"

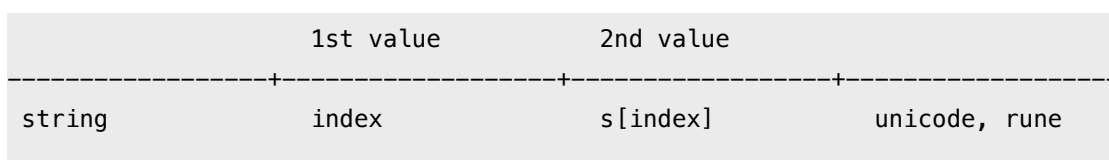
    for i, n := 0, length(s); i < n; i++ {    // 避免多次调用 length 函数。
        println(i, s[i])
    }
}
```

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)。



array/slice	index	s[index]
map	key	m[key]
channel	element	

可忽略不想要的返回值，包括用 "\_" 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。只需将返回对象地址保存到某个寄存器，就可让匿名函数获取相关闭包对象指针。

```
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，那怕是 void。

    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)
    }()

    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

```

### 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调用堆栈传递。

```
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

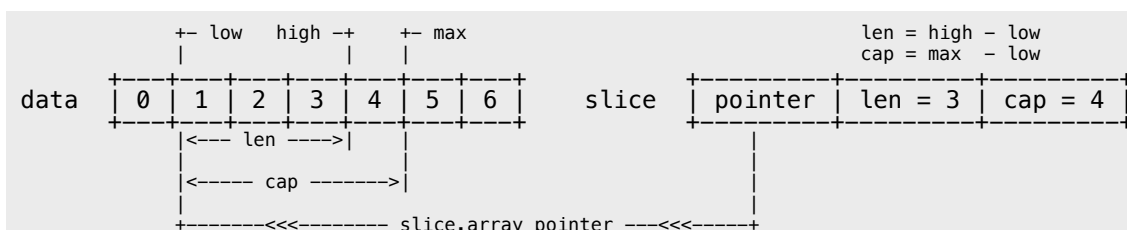
需要说明, `slice` 并不是数组或数组指针。它通过内部指针和相关属性引用数组片段, 以实现变长方案。

`runtime.h`

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

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

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



创建表达式使用的是元素索引号，而非数量。

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

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

读写操作实际目标是底层数组，只需注意索引号的差别。

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

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

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

输出：

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

可直接创建 `slice` 对象，自动分配底层数组。

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

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

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

输出：

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

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

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

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

fmt.Println(s)
```

输出:

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

至于 [][]T，是指元素类型为 []T。

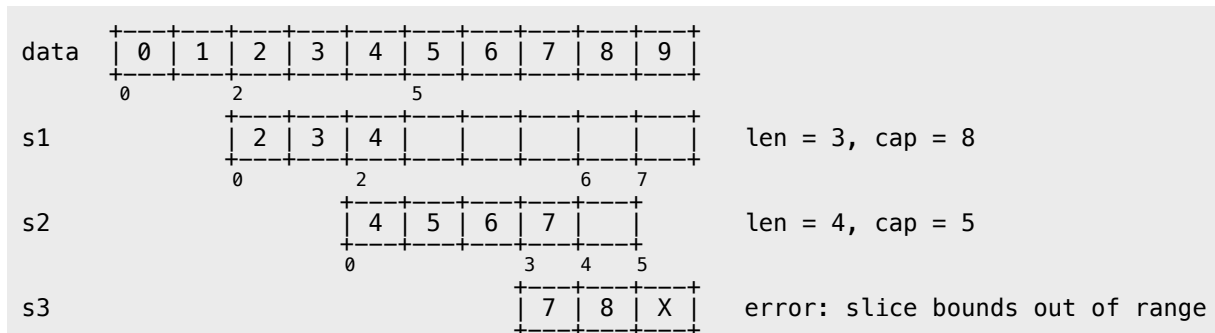
```
data := [][]int{
    []int{1, 2, 3},
    []int{100, 200},
    []int{11, 22, 33, 44},
}
```

### 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。

```
func main() {
    s := make([]int, 0, 5)

    for i := 0; i < cap(s); i++ {
        s = append(s, i)           // append 不会修改原 slice 属性, 而是生成新对象。
    }                             // 此处将新 slice 对象依然保存在原 s 所在内存。

    fmt.Println(s)
}
```

输出:

```
[0 1 2 3 4]
```

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, 应该避免使用 `append`, 因为频繁创建 `slice` 对象会影响性能。可一次性分配 `len` 足够大的 `slice`, 直接用索引号进行操作。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。
    println(k, v)
}
```

不能保证迭代次序和添加时相同，具体结果和不同版本的实现有关。

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。

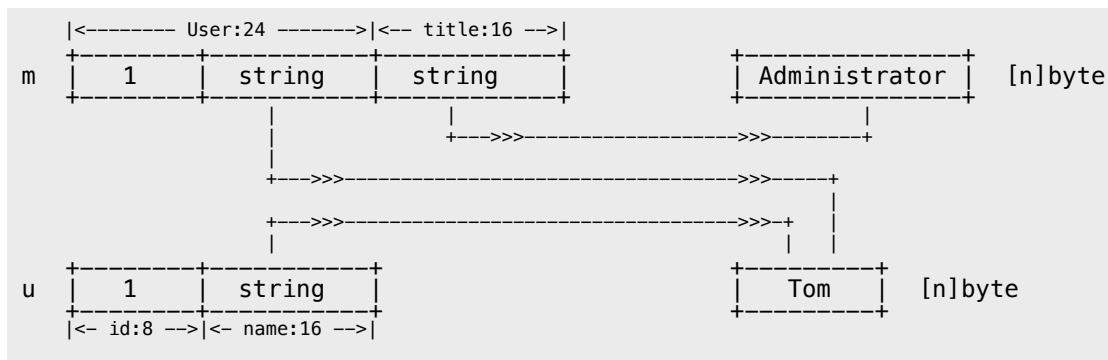
```
type User struct {
    id    int
    name  string
}

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

与协程类似，goroutine 一样使用自定义执行栈来减小内存开销，保存和恢复执行现场。不同的是，goroutine 有一个比较聪明的自主调度器，能合理安排底层系统线程和任务队列，使得成千上万的并发单元可以高效执行。

说明：有关 goroutine 和 scheduler 的实现细节可参考本书附录部分。

在语言层面提供 goroutine 支持，让并发程序编写变得简单。只要在函数调用语句前添加 go 关键字，就可创建并在后台运行 goroutine。

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

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

官方默认实现，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

多个 `goroutine` 运行在同一进程地址空间, 遵循 "以通讯来共享内存" 原则。`channel` 是一个 FIFO Queue, 用于多个 `goroutine` 通讯。其内部实现了同步, 可确保并发安全。



引用类型，使用 **make** 创建，默认为同步模式。同步操作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                       // 在缓冲区未滿前，不会阻塞。
    data <- 2
    data <- 3
```

```

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

    exit <- true
}()

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

<-exit
}

```

缓冲区是内部属性, 并非类型构成要素。

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

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

```

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

```

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

内置函数 `len` 返回未被读取的缓冲元素数量, `cap` 返回缓冲区大小。

```

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

d2 <- 1

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

```

### 7.2.1 单向

可以将 **channel** 隐式转换为单向队列，只收或只发。

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

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

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

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

不能将单向 **channel** 转换为普通 **channel**。

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

### 7.2.2 选择

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

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

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

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

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

```

    }()

    for i := 0; i < 5; i++ {
        select {
            // 随机选择可用 channel, 发送数据。
            case a <- i:
            case b <- i:
            }
    }

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

```

输出:

```

b 3
a 0
a 1
a 2
b 4

```

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

### 7.2.3 模式

用简单工厂模式打包并发任务和 `channel`。

```

func 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
    }

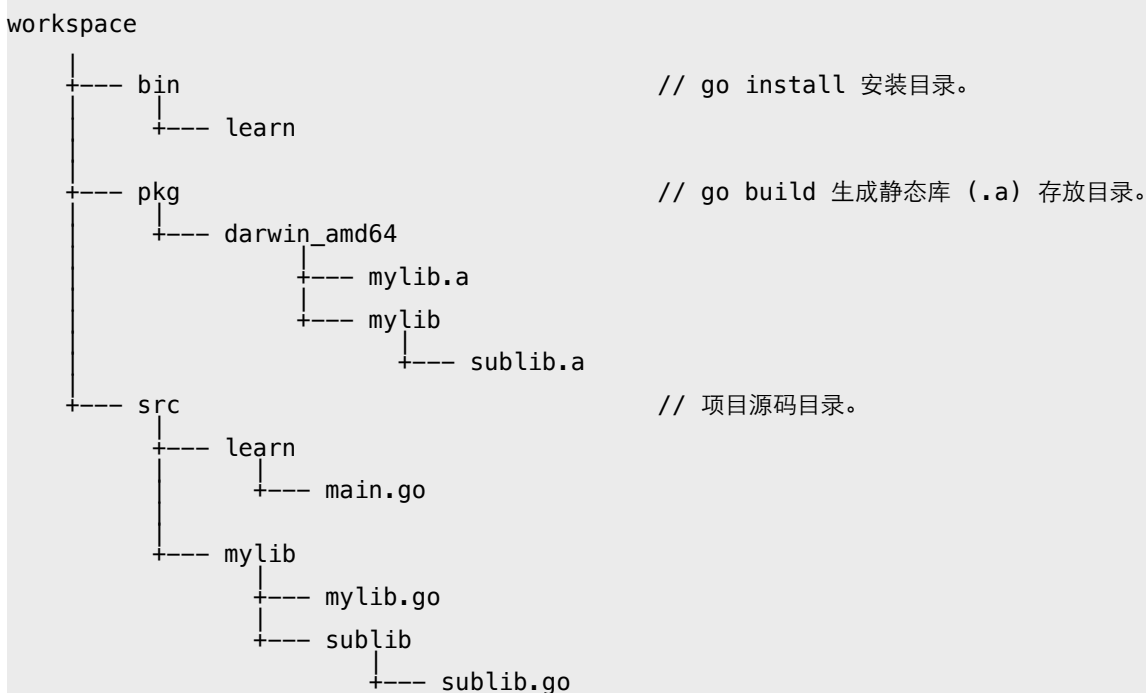
    req.ret <- x
}

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
├── src
│   └── learn
│       ├── main.go
│       └── test
│           └── test.go
```

**main.go**

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

## 8.2.2 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

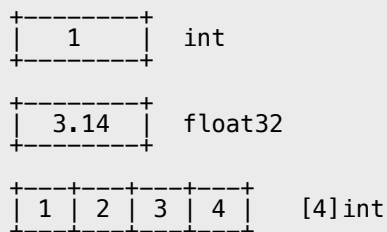
非测试源码文件中以 **BUG(author)** 开始的注释，会在帮助文档 **Bugs** 节点中显示。

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

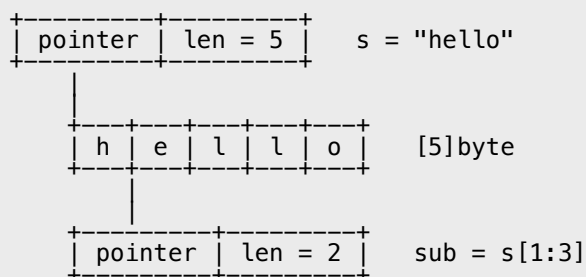
## 第 9 章 进阶

### 9.1 内存布局

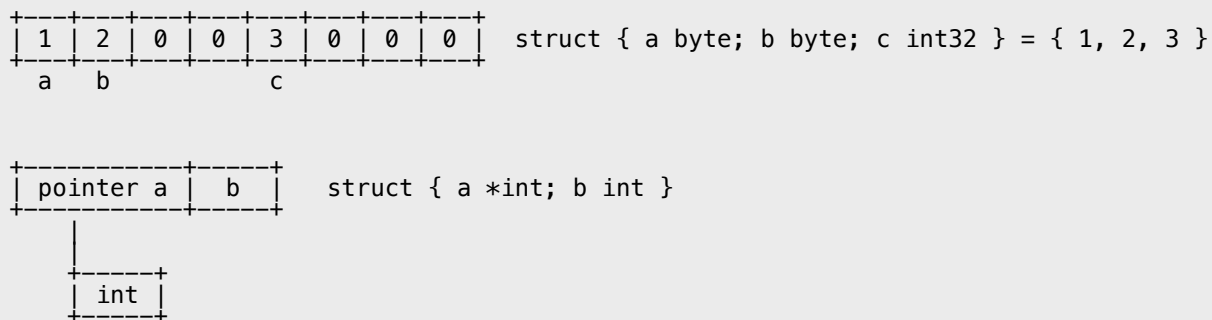
了解对象内存布局，有助于理解值传递、引用传递等概念。



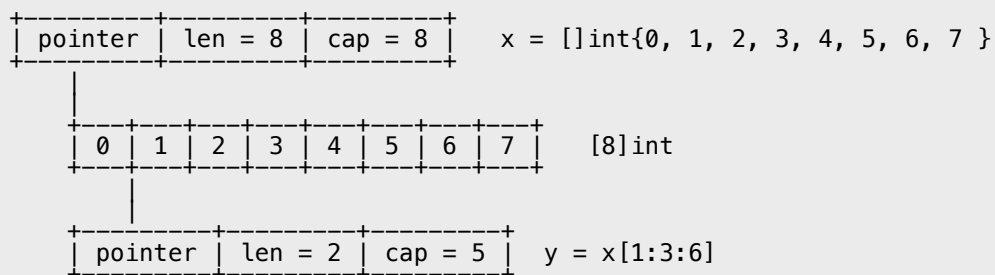
#### string



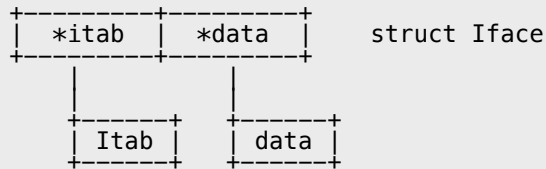
#### struct



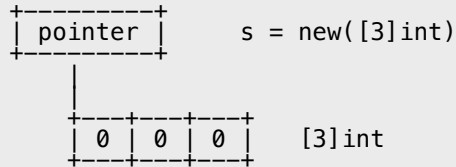
#### slice



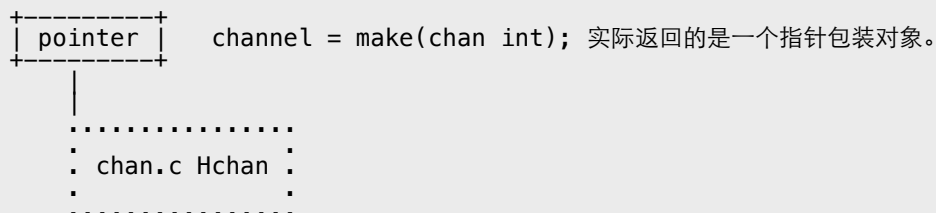
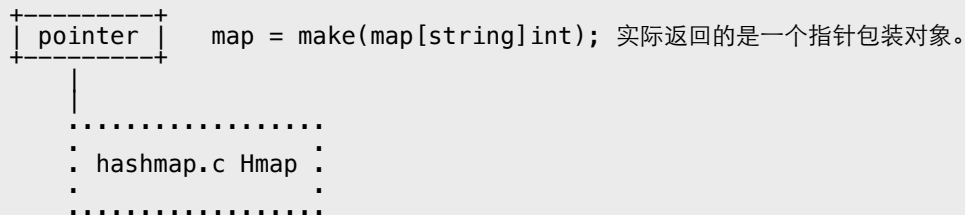
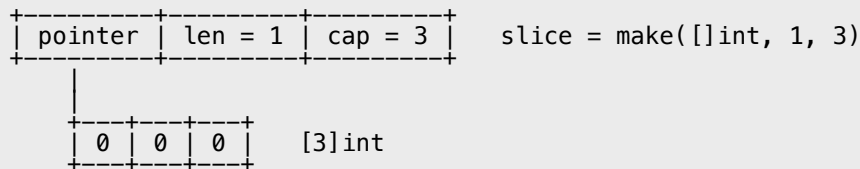
## interface



## 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。

## 第二部分 源码

基于 go 1.2, 相关文件位于 `go/src/pkg/runtime` 目录。为便于阅读, 示例代码做过裁剪。另外, 本文没有提供 32-bit 代码分析, 请自行参考源码文件。

# 1. Memory Allocator

在 `malloc.h` 开头有一大段注释，明确表示 “Memory allocator, based on tcmalloc”，详细描述了内存分配器的相关信息，仔细阅读有助于理解本章内容。

```
// 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-M) 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.
```

举例来描述内存分配器的算法流程，假设每次为对象分配内存都是一次人力资源请求。

作为管理成千上万人力资源的公司 (**heap**)，基于成本考虑，每次招聘不可能是一两个人，通常是一百或者更多。这些人依照实际用工需要，被安排到不同部门 (**central**) 接受职业培训。

当门店 (**cache**) 接到客户请求 (**malloc**)，就从对应部门调人 (**object**) 过来，为其安排任务。在完成任务后，此人并不会立即回到上级部门，而是留在门店。因为按以往经验，该地区出现类似需求的几率很高。如此，可避免去上级部门排队 (**lock**) 申请调人，影响客户服务。

如果门店经理发现某工种滞留人员过多，就会打发其中一部分回培训部，因为其他门店可能更需要这些人。

公司得给每个人发工资，自然不能让人闲着。如某部门闲散人员过多，那么就让他们其中一些人转岗去别的人力紧缺部门，接受新职业培训后，积极完成客户需求。

通过两次平衡，即满足了门店的快速服务，又避免了人力浪费。

另外，董事会定期巡查所有部门，如发现某部门转岗后还有大量闲人，那么就解聘一些，以缩减运营成本。只是这些人的工位依然保留，一旦需要，立即招聘填充，避免影响公司管理架构。

## 1.1 基本概念

内存分配器想要高效，无非是要满足几个条件：首先，是向操作系统申请大块内存，减少系统调用次数；其次，将大块内存分割成多种不同大小规格的小块缓存，用于数量庞大的小对象快速分配和复用；最后，在多个线程之间平衡缓存，即避免锁造成的性能问题，还要充分利用已申请内存。

内存申请和管理以页 (4096 字节) 为单位。每次申请内存，**heap** 都新建一个名为 **span** 的对象将这些页管理起来。这么做是为了减少碎片，尽可能提供更大块内存用于分割。

### malloc.h

```
struct MSpan
{
    MSpan    *next;        // in a span linked list
    MSpan    *prev;        // in a span linked list
    PageID    start;        // starting page number, start = page_address>>PageShift
    uintptr    npages;        // number of pages in span
    MLink    *freelist;    // list of free objects
    uint32    ref;        // number of allocated objects in this span
    int32    sizeclass;    // size class
    uintptr    elemsize;    // computed from sizeclass or from npages
    uint32    state;        // MSpanInUse etc
};
```

另一方面，页对于大多数对象来说太大了。因此，以 8 的倍数为单位，将对象分成不同级别，比如长度 1 ~ 8 字节为 L1，9 ~ 16 为 L2，如此等等。分配器以 32KB 为界，将对象分为大小两种，所有小对象都能对应到 60 个级别中的一种。

将 **span** 的大块内存按需切分成不同级别存储单元，以满足不同小对象的存储需求。举例来说，存储 13 字节小对象，只需找到 L2.**span**，从中获取一个存储单元即可。至于大对象，直接用一个大小合适的 **span** 当存储单元。

在分配器中，级别称为 **sizeclass**，每个切分出来的对象存储单元称为 **object**。

### malloc.h

```
enum
{
    // Computed constant. The definition of MaxSmallSize and the
    // algorithm in msize.c produce some number of different allocation
    // size classes. NumSizeClasses is that number. It's needed here
    // because there are static arrays of this length; when msize runs its
    // size choosing algorithm it double-checks that NumSizeClasses agrees.
```

```

    NumSizeClasses = 61,

    // Tunable constants.
    MaxSmallSize = 32<<10,
}

// Size classes. Computed and initialized by InitSizes.
//
// SizeToClass(0 <= n <= MaxSmallSize) returns the size class,
// 1 <= sizeclass < NumSizeClasses, for n.
// Size class 0 is reserved to mean "not small".
//
// class_to_size[i] = largest size in class i
// class_to_allocnpages[i] = number of pages to allocate when
// making new objects in class i

extern int8 runtime_size_to_class8[1024/8 + 1];
extern int8 runtime_size_to_class128[(MaxSmallSize-1024)/128 + 1];

int32 runtime_SizeToClass(int32);
extern int32 runtime_class_to_size[NumSizeClasses];
extern int32 runtime_class_to_allocnpages[NumSizeClasses];

```

在初始化阶段，分配器向操作系统申请预留了超大范围的虚拟地址，其管理算法依赖连续内存地址做索引因子，必须阻止其他内存申请操作占用这些地址。

所保留地址用途分成三个部分：

- **arena**: 向操作系统申请存储内存的实际地址范围。
- **spans**: 数组指针，记录页所对应 **span** 对象，用于反查和合并。
- **bitmap**: 位图，为内存地址提供 4 bit 属性标记位，用于垃圾回收等。

## malloc.h

```

struct MHeap
{
    // span lookup
    MSpan** spans;           // index = span->start - arena_start>>PageShift
    uintptr spans_mapped;

    // range of addresses we might see in the heap
    byte *bitmap;
    uintptr bitmap_mapped;

    byte *arena_start;
    byte *arena_used;
}

```

```
byte *arena_end;
};
```

作为管理用途的 **spans**、**bitmap**，可随 **arena** 线性增量申请同步扩张，节约内存。

如同每个 CPU 核心都有自己的缓存，内存分配器为每个工作线程绑定一个本地缓存管理对象 **cache**。这样可在无锁状态下快速获取存储单元，减少因为竞争导致排队。

将同一级别的 **object** 单元串成链表，然后用数组管理这些链表。需要时，用 **classsize** 做数组索引就可找到对应链表，提取或归还 **object**。

## malloc.h

```
struct MLink          // 将 object 转型成 MLink，存储下一个 object 指针，构成链表。
{
    MLink *next;
};

// Per-thread (in Go, per-M) cache for small objects.
// No locking needed because it is per-thread (per-M).
struct MCacheList
{
    MLink *list;
    uint32 nlist;
};

struct MCache
{
    MCacheList list[NumSizeClasses];
};
```

当对应链表无法提供可用缓存时，就须向下层机构 **central** 申请。

分配器在 **heap** 里管理着多个 **central** 对象，每个对应一种 **sizeclass**。它从 **heap** 获取 **span**，按 **sizeclass** 切分成 **object**，保存到 **span.freelist** 链表里。切分完成的 **span**，被放到 **noempty** 链表里。

当 **cache** 申请内存时，从 **noempty** 链表拿出一个 **span**，将它的 **freelist** 移交过去就行了。至于这个没有剩余空间的 **span** 会被转移到 **empty** 链表。

## malloc.h

```
struct MCentral
{
```

```

    Lock;
    int32 sizeclass;
    MSpan nonempty;
    MSpan empty;
    int32 nfree;
};

struct MHeap
{
    // central free lists for small size classes.
    struct {
        MCentral;
    } central[NumSizeClasses];
};

```

当 **object** 被归还给 **cache** 时，分配器会检查所对应链表容量。如超出某个阈值，就提取一部分送回 **central**，以分配给其他饥饿线程使用。

通过 **heap.spans** 可反查到 **object** 所对应的 **span**。当 **central** 发现某个 **span** 收回全部 **object** 时，会将其送还给 **heap**，以便分配给其他 **central** 使用。

分配器使用 **central** 来平衡内存分配，让每个 **cache** 仅持有合适数量的 **object**，并回收那些空闲的大块内存，用于切分其他级别 **object**，减少了内存浪费。

整个进程只有一个 **heap**，它管理着内存分配的各个方面。除前面提及的那些，还有两个重要的 **span** 管理链表。

## malloc.h

```

extern MHeap runtime.mheap;

MaxMHeapList = 1<<(20 - PageShift),    // Maximum page length for fixed-size list.

struct MHeap
{
    Lock;
    MSpan free[MaxMHeapList];    // free lists of given length
    MSpan large;                 // free lists length >= MaxMHeapList
};

```

将页数相同的 **span** 串成链表，放到 **free** 数组，超出页数限制的统统发配到 **large** 链表里。与前面一样，这是为了快速获取大小合适的 **span**。



具体算法：在 **free** 数组找到和需求页数相同的链表，如链表为空，就遍历数组中页数更多的链表。如全部失败，则继续检查 **large**，直至从操作系统申请新的内存为止。

依据上面的算法，所拿到的 **span** 页数可能大于需求预期。为避免浪费，分配器将多余部分截取下来，重新放回链表中。不过在放回之前，会通过 **heap.spans** 检查相邻 **span** 状态，尽可能合并成更大的 **span** 内存块。这么做可减少碎片，提高内存利用率。同样，从操作系统申请内存时，也会尝试合并操作。

## 1.2 初始化

函数 **mallocinit** 除调用 **InitSizes** 初始化 **size\_to\_class** 等几个全局表外，最重要的就是设定保留虚拟地址范围。

### malloc.goc

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

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

        // 在 64-bit 系统中，实际保留地址 256MB spans + 8GB bitmap + 128GB arena
        arena_size = MaxMem;
        bitmap_size = arena_size / (sizeof(void*)*8/4);
        spans_size = arena_size / PageSize * sizeof(runtime·mheap.spans[0]);
        spans_size = ROUND(spans_size, PageSize);

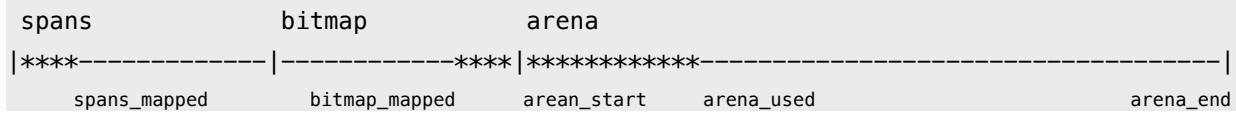
        // 起始地址 0xc000000000，如果失败则换个位置。
        for(i = 0; i <= 0x7f; i++) {
            p = (void*)(i<<40 | 0x00c0ULL<<32);

            // 保留虚拟地址范围，并未实际分配内存。
            p = runtime·SysReserve(p, bitmap_size + spans_size + arena_size);
            if(p != nil) break;
        }
    }

    // 保留区域头部存储 spans、bitmap。仅设置地址，并没有分配内存。
    runtime·mheap.spans = (MSpan**)p;
    runtime·mheap.bitmap = p + spans_size;

    // 确定地址范围和线性分配起始地址 arena_used。每次分配成功后后移该地址。
    runtime·mheap.arena_start = p + spans_size + bitmap_size;
    runtime·mheap.arena_used = runtime·mheap.arena_start;
    runtime·mheap.arena_end = runtime·mheap.arena_start + arena_size;
```

```
// 初始化 heap 全局变量, 为当前线程创建 cache。
runtime·MHeap_Init(&runtime·mheap);
m->mcache = runtime·allocmcache();
}
```



分配器用 **mmap** 向操作系统增量申请内存, 不管预留多大的虚拟地址空间, 也只在实际读写时, 操作系统才会按页分配物理内存。

另外, OSX/darwin **SysReserve** 函数中 **mmap** 没有使用 **MAP\_FIXED** 参数, 因此其保留起始地址未必是 **0xc000000000**。

## mem\_darwin.c

```
void* runtime·SysReserve(void *v, uintptr n)
{
    void *p = runtime·mmap(v, n, PROT_NONE, MAP_ANON|MAP_PRIVATE, -1, 0);
    if(p < (void*)4096) return nil;
    return p;
}
```

## 1.3 分配流程

从分配函数 **malloc**、**new** 开始剖析整个分配流程。

### malloc.goc

```
void* runtime·malloc(uintptr size)
{
    return runtime·mallocgc(size, 0, FlagNoInvokeGC);
}

void runtime·new(Type *typ, uint8 *ret)
{
    ret = runtime·mallocgc(typ->size, ...);
}
```

超过 32KB 的大对象从 **heap** 分配, 小对象直接从 **cache** 对应链表中提取。

### malloc.goc

```

void* runtime·mallocgc(uintptr size, uintptr typ, uint32 flag)
{
    MCache *c;
    MCacheList *l;
    uintptr npages;
    MSpan *s;
    MLink *v;

    c = m->mcache; // m 当前线程。

    // 小对象从 cache 提取 object, 大对象直接从 heap 分配。
    if(size <= MaxSmallSize) {
        // 计算对应 sizeclass。
        if(size <= 1024-8)
            sizeclass = runtime·size_to_class8[(size+7)>>3];
        else
            sizeclass = runtime·size_to_class128[(size-1024+127) >> 7];

        // 用 sizeclass 做索引, 找到对应链表。
        l = &c->list[sizeclass];

        // 如果没有可用 object, 从 central 批发一些过来。
        if(l->list == nil)
            runtime·MCache_Refill(c, sizeclass);

        // 从链表取出第一个可用 object。
        v = l->list;

        // 调整链表状态。
        l->list = v->next;
        l->nlist--;
    } else {
        // 直接从 mheap 获取大小合适的 span。
        npages = size >> PageShift;
        s = runtime·MHeap_Alloc(&runtime·mheap, npages, 0, 1, !(flag & FlagNoZero));
        v = (void*)(s->start << PageShift);
    }

    // 在 heap.bitmap 中标记 object 状态。
    if(!(flag & FlagNoGC))
        runtime·markallocated(v, size, (flag&FlagNoScan) != 0);

    // 检查是否触发垃圾回收。
    if(!(flag & FlagNoInvokeGC) && mstats.heap_alloc >= mstats.next_gc)
        runtime·gc(0);

    return v;
}

```

如果链表为空，则需要从相同 sizeclass 的 central 批量获取的 object。

### mcache.c

```
void runtime·MCache_Refill(MCache *c, int32 sizeclass)
{
    MCacheList *l = &c->list[sizeclass];
    l->nlist = runtime·MCentral_AllocList(&runtime·mheap.central[sizeclass], &l->list);
}
```

直接将找到的 span.freelist 返回给 cache，有别于 1.2 版本之前使用数量表的做法。

### mcentral.c

```
int32 runtime·MCentral_AllocList(MCentral *c, MLink **pfirst)
{
    MSpan *s;
    int32 cap, n;

    // 加锁
    runtime·lock(c);

    // 如果 noempty 链表上没有 span，也就意味着没有剩余 object 可返回。
    if(runtime·MSpanList_IsEmpty(&c->nonempty)) {
        if(!MCentral_Grow(c)) { // 从 heap 获取 heap，切割 object。
            return 0;
        }
    }

    // 从 nonempty 链表提取可用 span。
    s = c->nonempty.next;

    // 计算可用 object 数量。
    cap = (s->npages << PageShift) / s->elemsize;
    n = cap - s->ref;

    // 直接返回整个 span.freelist。
    *pfirst = s->freelist;

    // 调整 span 和 central 状态。
    s->freelist = nil;
    s->ref += n;
    c->nfree -= n;

    // 将 span 转移到 empty 链表。
    runtime·MSpanList_Remove(s);
    runtime·MSpanList_Insert(&c->empty, s);
}
```

```

runtime.unlock(c);
return n;
}

```

扩张 **central** 时，需将获取的 **span** 切割成 **object** 存储到 **span.freelist** 链表。该 **span** 会被添加到 **central.noempty**，表示非空，有 **object** 可供使用。

## mcentral.c

```

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

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

    // 按 sizeclass 切割成 object，保存到 freelist。
    tailp = &s->freelist;
    p = (byte*)(s->start << PageShift); // 起始地址
    for(i=0; i<n; i++) {
        v = (MLink*)p;
        *tailp = v;
        tailp = &v->next;
        p += size;
    }

    // 调整 central 可用 object 数量。
    c->nfree += n;

    // 将 span 添加到 noempty 链表。
    runtime.MSpanList_Insert(&c->nonempty, s);

    return true;
}

```

从 **heap** 获取 **span** 一样要加锁。

## mheap.c

```

MSpan* runtime.MHeap_Alloc(MHeap *h, uintptr npage, int32 sizeclass, int32 acct, int32
zeroed)
{
    runtime.lock(h);

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

```

```
runtime.unlock(h);
return s;
}
```

首先从 **free** 数组中寻找页数相同的 **span** 链表，如链表为空，那么找页数更多的链表。再不行，就从 **large** 里面找。全部失败时，就得向操作系统申请内存了。

如果找到的 **span** 太大，就将多余的部分截取出来，放回链表。在返回 **span** 之前，按页在 **heap.spans** 对应位置填写指针，用于反查。

## mheap.c

```
static MSpan* MHeap_AllocLocked(MHeap *h, uintptr npage, int32 sizeclass)
{
    MSpan *s, *t;
    PageID p;

    // 从 free 数组寻找合适的 span。
    for(n=npage; n < nelem(h->free); n++) {
        // 如果该链表为空，那么继续找页数更多的链表。
        if(!runtime.MSpanList_IsEmpty(&h->free[n])) {
            s = h->free[n].next;
            goto HaveSpan;
        }
    }

    // 如果在 free 所有链表里都没找到，继续从 large 链表中找。
    if((s = MHeap_AllocLarge(h, npage)) == nil) {
        // 全部失败，扩张 heap。
        if(!MHeap_Grow(h, npage)) return nil;

        // 再次从 large 中分配。
        if((s = MHeap_AllocLarge(h, npage)) == nil) return nil;
    }
}
```

### HaveSpan:

```
// 如果返回 span 页数大于期望。
if(s->npages > npage) {
    // 截取 span 尾部多余部分，生成新的 span。
    t = runtime.FixAlloc_Alloc(&h->spanalloc);
    runtime.MSpan_Init(t, s->start + npage, s->npages - npage);
    s->npages = npage;

    // 在 heap.spans 标记截取出来的 span 信息。
    p = t->start;
    if(sizeof(void*) == 8)
        p -= ((uintptr)h->arena_start >> PageShift); // 注意减去 arena_start。
}
```

```

    if(p > 0) h->spans[p-1] = s;                // 标记待返回 span 结束位置。
    h->spans[p] = t;
    h->spans[p+t->npages-1] = t;
    t->state = MSpanInUse;

    // 将截取出来的 span 放回 free/large 链表。该函数会执行合并操作。
    MHeap_FreeLocked(h, t);
}

// 修正待返回 span 属性, 并在 heap.spans 标记信息。
s->sizeclass = sizeclass;
p = s->start;
if(sizeof(void*) == 8) p -= ((uintptr)h->arena_start >> PageShift);

for(n=0; n<npage; n++)    // 在 heap.spans 所有对应位置填写 span 指针。
    h->spans[p+n] = s;    // mheap.c runtime·MHeap_Lookup, MHeap_LookupMaybe

return s;
}

```

将 span 放回链表前, 会尝试合并相邻块。

## mheap.c

```

static void MHeap_FreeLocked(MHeap *h, MSpan *s)
{
    // 修改 span 状态。
    s->state = MSpanFree;

    p = s->start;
    if(sizeof(void*) == 8) p -= (uintptr)h->arena_start >> PageShift;

    // 检查 heap.spans, 合并该 span 前后相邻未使用的 span, 形成更大内存块。
    if(p > 0 && (t = h->spans[p-1]) != nil && t->state != MSpanInUse) {
        // 合并左侧 span。
        s->start = t->start;
        s->npages += t->npages;
        s->nreleased = t->nreleased; // absorb released pages
        p -= t->npages;
        h->spans[p] = s;

        // 释放左侧 span 资源。
        runtime·MSpanList_Remove(t);
        t->state = MSpanDead;
        runtime·FixAlloc_Free(&h->spanalloc, t);
    }
}

```

```

if((p+s->npages)*sizeof(h->spans[0]) < h->spans_mapped &&
    (t = h->spans[p+s->npages]) != nil && t->state != MSpanInUse) {
    // 合并右侧 span。
    s->npages += t->npages;
    s->npreleased += t->npreleased;
    h->spans[p + s->npages - 1] = s;

    // 释放右侧 span 资源。
    runtime·MSpanList_Remove(t);
    t->state = MSpanDead;
    runtime·FixAlloc_Free(&h->spanalloc, t);
}

// 根据页数决定插入 free 还是 large 链表。
if(s->npages < nelem(h->free))
    runtime·MSpanList_Insert(&h->free[s->npages], s);
else
    runtime·MSpanList_Insert(&h->large, s);
}

```

从操作系统申请内存，其大小总是 64KB 的倍数，最少 1MB。

## mheap.c

```

static bool MHeap_Grow(MHeap *h, uintptr npage)
{
    // 最小 1MB，总是 64KB (16 Pages) 的倍数。
    npage = (npage+15)&~15;
    ask = npage<<PageShift;
    if(ask < HeapAllocChunk)           // HeapAllocChunk = 1<<20
        ask = HeapAllocChunk;

    // 从操作系统分配内存。
    v = runtime·MHeap_SysAlloc(h, ask);

    // 创建新 Span 对象，管理新分配内存。
    s = runtime·FixAlloc_Alloc(&h->spanalloc);
    runtime·MSpan_Init(s, (uintptr)v>>PageShift, ask>>PageShift);

    p = s->start;
    if(sizeof(void*) == 8) p -= ((uintptr)h->arena_start>>PageShift);

    // 在 heap.spans 标记 span。
    h->spans[p] = s;
    h->spans[p + s->npages - 1] = s;
    s->state = MSpanInUse;

    // 添加到 free/large 链表，一样会合并相邻未使用的 span。
}

```



```

    MHeap_FreeLocked(h, s);

    return true;
}

```

函数 `MHeap_SysAlloc` 表明 Go 存在内存上限，并不会扩大保留地址范围。不过 64 位系统 128GB 的保留地址空间，暂时够用了。

## malloc.goc

```

void* runtime·MHeap_SysAlloc(MHeap *h, uintptr n)
{
    // 判断是否在保留区域内。
    if(n <= h->arena_end - h->arena_used) {
        // 使用了 arena_used 作为分配地址。
        p = h->arena_used;

        // 在指定地址分配内存。
        runtime·SysMap(p, n, &mstats.heap_sys);

        // 调整下一次分配地址。
        h->arena_used += n;

        // 按需为 heap.spans 等分配内存。
        runtime·MHeap_MapBits(h);
        runtime·MHeap_MapSpans(h);

        return p;
    }

    return p;
}

```

最终通过 `mmap`、`VirtualAlloc` 向操作系统申请内存，注意指定 `arena_used` 作为开始地址。

## mem\_linux.c

```

void runtime·SysMap(void *v, uintptr n, uint64 *stat)
{
    void *p;

    runtime·xadd64(stat, n);

    // On 64-bit, we don't actually have v reserved, so tread carefully.
    if(sizeof(void*) == 8 && (uintptr)v >= 0xfffffffffU) {
        p = mmap_fixed(v, n, PROT_READ|PROT_WRITE, MAP_ANON|MAP_PRIVATE, -1, 0);
    }
}

```

```

    if(p == (void*)ENOMEM)
        runtime·throw("runtime: out of memory");
    if(p != v) {
        runtime·printf("runtime: address space conflict: map(%p) = %p\n", v, p);
        runtime·throw("runtime: address space conflict");
    }

    return;
}

p = runtime·mmap(v, n, PROT_READ|PROT_WRITE, MAP_ANON|MAP_FIXED|MAP_PRIVATE, -1, 0);

if(p == (void*)ENOMEM)
    runtime·throw("runtime: out of memory");
if(p != v)
    runtime·throw("runtime: cannot map pages in arena address space");
}

```

## mem\_windows.c

```

void runtime·SysMap(void *v, uintptr n, uint64 *stat)
{
    void *p;

    runtime·xadd64(stat, n);
    p = runtime·stdcall(runtime·VirtualAlloc, 4, v, n,
        (uintptr)MEM_COMMIT, (uintptr)PAGE_READWRITE);

    if(p != v)
        runtime·throw("runtime: cannot map pages in arena address space");
}

```

## 1.4 释放流程

调用 `free` 释放 `object`，更确切说是还给 `cache` 或 `heap`。

## malloc.goc

```

void runtime·free(void *v)
{
    // 查找管理该 object 的 span。
    if(!runtime·mlookup(v, nil, nil, &s)) { ... }

    sizeclass = s->sizeclass;
    c = m->mcache;
}

```

```

    if(sizeclass == 0) {
        // 大对象直接归还给 heap。
        runtime·MHeap_Free(&runtime·mheap, s, 1);
    } else {
        // 将 object 归还给 cache。
        runtime·MCache_Free(c, v, sizeclass, size);
    }
}

```

对象释放规则和分配相对应。只需跟踪小对象，反正最终也得回到 **heap**。

## mcache.c

```

void runtime·MCache_Free(MCache *c, void *v, int32 sizeclass, uintptr size)
{
    MCacheList *l;
    MLink *p;

    // 找到对应的链表。
    l = &c->list[sizeclass];

    // 将 object 指针转换成 MLink*, 添加到链表头部。
    p = v;
    p->next = l->list;
    l->list = p;

    // 调整链表和 cache 属性。
    l->nlist++;
    c->local_cachealloc -= size;

    // 如果链表上的 object 数量超出初始值的 2 倍, 则归还一半给 central。
    if(l->nlist >= 2*(runtime·class_to_alloctopages[sizeclass]<<PageShift)/size)
        ReleaseN(l, l->nlist/2, sizeclass);
}

```

内存池本质就是复用，将 **object** 重新放回 **cache** 链表是很自然的。但如果 **cache** 持有太多空闲 **object** 就会造成浪费，可能其他线程 **cache** 更需要这些内存单元。

## mcache.c

```

static void ReleaseN(MCacheList *l, int32 n, int32 sizeclass)
{
    MLink *first, **lp;
    int32 i;

    // 提取头部 n 个 object 组成链表。
    first = l->list;
    lp = &l->list;

```

```

    for(i=0; i<n; i++)
        lp = &(*lp)->next;
    l->list = *lp;
    *lp = nil;
    l->nlist -= n;

    // 将链表归还给 central。
    runtime·MCentral_FreeList(&runtime·mheap.central[sizeclass], first);
}

```

循环链表，依次将 **object** 交回对应的 **span.freelist**。如果发现 **span** 已收回全部内存，那么将其还给 **heap**。

## mcentral.c

```

void runtime·MCentral_FreeList(MCentral *c, MLink *start)
{
    MLink *next;
    runtime·lock(c);

    // 循环链表，依次归还 object。
    for(; start != nil; start = next) {
        next = start->next;
        MCentral_Free(c, start);
    }

    runtime·unlock(c);
}

static void MCentral_Free(MCentral *c, void *v)
{
    MSpan *s;
    MLink *p;

    // 找到管理该 object 的 span。
    s = runtime·MHeap_Lookup(&runtime·mheap, v);

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

    // 将 object 加到 span.freelist 头部。
    p = v;
    p->next = s->freelist;
    s->freelist = p;
    c->nfree++;
}

```

```

// 调整 span 分配计数器。如满血，那么将 span 还给 heap。
if(--s->ref == 0) {
    size = runtime.class_to_size[c->sizeclass];

    // 从 central 链表中移除，然后交还给 heap。
    runtime.MSpanList_Remove(s);
    runtime.MHeap_Free(&runtime.mheap, s, 0);
}
}

```

调用 `MHeap_FreeLocked` 函数，合并相邻 `span`，然后添加到 `heap` 管理链表。释放过程到此为止，内存只是还给 `heap`，将其释放给操作系统是由垃圾回收器完成的。

## mheap.c

```

void runtime.MHeap_Free(MHeap *h, MSpan *s, int32 acct)
{
    MHeap_FreeLocked(h, s);
}

```

## 1.5 其他

内存管理对象中，`cache` 和 `span` 在运行期动态创建。它们引用 `arena` 内存，但自身并不从 `arena` 分配，而是使用 `FixAlloc` 内存分配器。

每个 `FixAlloc` 实例管理和分配一种固定长度的对象，按需使用 `mmap` 申请内存。

## malloc.h

```

// FixAlloc is a simple free-list allocator for fixed size objects. Malloc uses a
// FixAlloc wrapped around SysAlloc to manages its MCache and MSpan objects.
struct FixAlloc
{
    uintptr size;
    void (*first)(void *arg, byte *p); // called first time p is returned
    void* arg;
    MLink* list;
    byte* chunk;
    uint32 nchunk;
    uintptr inuse; // in-use bytes now
};

```

## 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;
}

```

在初始化 heap 时, 创建了 spanalloc 和 cachealloc 两个 FixAlloc 分配器对象。

### mheap.c

```

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

```

用 RecordSpan 的目的是为 heap.allspans 申请内存, 用来存储所有 span 指针。

### mheap.c

```

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

    // 内存不足时, 重新申请, 并将原数据转移过来。
    if(h->nspan >= h->nspancap) {
        cap = 64*1024/sizeof(all[0]);
        if(cap < h->nspancap*3/2) cap = h->nspancap*3/2; // 扩容到 1.5 倍

        all = (MSpan**)runtime·SysAlloc(cap*sizeof(all[0]), &mstats.other_sys);
        if(h->allspans) {
            runtime·memmove(all, h->allspans, h->nspancap*sizeof(all[0]));
            runtime·SysFree(h->allspans, h->nspancap*sizeof(all[0]), &mstats.other_sys);
        }

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

    // 记录 span 指针。
}

```

```

    h->allspans[h->nspan++] = s;
}

```

看看 `FixAlloc_Alloc` 的实际分配过程。

## mheap.c

```

static bool MHeap_Grow(MHeap *h, uintptr npage)
{
    v = runtime·MHeap_SysAlloc(h, ask);
    s = runtime·FixAlloc_Alloc(&h->spanalloc);
    runtime·MSpan_Init(s, (uintptr)v>>PageShift, ask>>PageShift);
}

```

## mfixalloc.c

```

void* runtime·FixAlloc_Alloc(FixAlloc *f)
{
    // 如果链表不为空，直接返回第一个可用块。
    if(f->list) {
        v = f->list;
        f->list = *(void**)f->list;
        f->inuse += f->size;
        return v;
    }

    // 如果链表为空，且剩余空间不足。
    if(f->nchunk < f->size) {
        // 申请一个新的 16KB 内存块，将指针存放到 chunk 字段。
        f->chunk = runtime·persistentalloc(FixAllocChunk, 0, f->stat);

        // 剩余空间长度。
        f->nchunk = FixAllocChunk;
    }

    // 返回可用内存块开始地址。
    v = f->chunk;

    // 调用指定函数，比如 RecordSpan。
    if(f->first)
        f->first(f->arg, v);

    // 调整 chunk 开始位置，减少可用空间长度。
    f->chunk += f->size;
    f->nchunk -= f->size;
    f->inuse += f->size;

    return v;
}

```

从上面代码可以看出，在需要时，`FixAlloc` 会申请一个新的 `chunk` 内存块。以往申请的内存都被切分使用，释放时保存到 `freelist` 链表以供复用。

### `mfixalloc.c`

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

    // 插入到链表头部。
    *(void**)p = f->list;
    f->list = p;
}
```

`FixAlloc` 调用 `mmap` 申请内存。

### `malloc.goc`

```
void* runtime·persistentalloc(uintptr size, uintptr align, uint64 *stat)
{
    runtime·SysAlloc(size, stat);
}
```

### `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;
}
```



## 2. Garbage Collector

Go 使用精确垃圾回收算法，也就是 **Mark-and-Sweep**，没有代龄。

本节内容不会深入到标记和清除的算法细节中，仅注释主要回收过程。如对相关算法感兴趣，可参考相关论文或《深入理解计算机系统 第二版》9.10 章节。

垃圾回收和物理内存释放是分开进行的。垃圾回收尽可能将内存回收到 **heap**，然后由专门的函数定期释放物理内存。

### 2.1 垃圾回收

前面提到过，在分配内存的 `malloc.goc/mallocgc` 函数中会触发垃圾回收。

#### mgc0.c

```
void runtime·gc(int32 force)
{
    // 初始化 gcpercent, 首次从 GOGC 环境变量获取, 默认值 100。
    if(gcpercent == GcpercentUnknown) {    // first time through
        gcpercent = readgogc();
    }

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

    // 非强制回收时, 检查正在使用 object 容量是否超过阈值。
    if(!force && mstats.heap_alloc < mstats.next_gc) { return; }

    // 停止所有不相关的东西, 开始回收!
    a.start_time = runtime·nanotime();
    runtime·stoptheworld();

    // GODEBUG="gotrace=2" 会引发两次回收。
    for(i = 0; i < (runtime·debug.gctrace > 1 ? 2 : 1); i++) {
        // g0 stacks are not scanned, and we don't need to scan gc's internal state.
        // switch to g0, call gc(&a), then switch back
        g->param = &a;
        g->status = Gwaiting;
        g->waitreason = "garbage collection";

        // 调用回收函数。
        runtime·mcall(mgc);
    }
}
```

```

        // record a new start time in case we're going around again
        a.start_time = runtime.nanotime();
    }

    // 恢复执行。
    runtime.starttheworld();

    // 执行 finalizers。
    if(fing == nil)
        fing = runtime.newproc1(&runfinqv, nil, 0, 0, runtime.gc);

    runtime.gosched();
}

```

环境变量 GOGC 可关闭 GC，或调整回收阈值。并发回收受 GOMAXPROC 等限制。

```

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

static void gc(struct gc_args *args)
{
    // 确定并行回收的 goroutine 数量 = min(GOMAXPROCS, cpus, 8)。
    work.nproc = runtime.gcprocs();

    // 添加所有根对象。
    addroots();

    // 设置并行 mark、sweep 属性。
    runtime.parforsetup(work.markfor, work.nproc, work.nroot, nil, false, markroot);
    runtime.parforsetup(work.sweepfor, work.nproc, runtime.mheap.nspan, ..., sweepspan);

    // 开始并行 mark。
    runtime.parfordo(work.markfor);

    // 开始并行 sweep。
    runtime.parfordo(work.sweepfor);

    // 重置统计数据。
    cachestats();

    // 设置下次回收阈值。受 GOGC 影响，默认是回收后内存使用量的 2 倍。
    mstats.next_gc = mstats.heap_alloc+mstats.heap_alloc*gcpercent/100;

    // 更新垃圾回收统计信息。
    mstats.last_gc = t4;
}

```

```

mstats.pause_ns[mstats.numgc%nelem(mstats.pause_ns)] = t4 - t0;
mstats.pause_total_ns += t4 - t0;
mstats.numgc++;
}

```

跳过 markroot, 我们看看 sweepspan 是如何回收内存的。

```

static void sweepspan(ParFor *desc, uint32 idx)
{
    s = runtime·mheap·allspans[idx];
    p = (byte*)(s->start << PageShift);
    cl = s->sizeclass;
    size = s->elemsize;

    if(cl == 0) {
        n = 1;
    } else {
        npages = runtime·class_to_allocnpages[cl];
        n = (npages << PageShift) / size;
    }

    // 遍历 span 所有 object。
    for(; n > 0; n--, p += size, ...) {
        if(cl == 0) {
            // 如果 object 是大对象, 直接归还给 heap。
            runtime·MHeap_Free(&runtime·mheap, s, 1);
        } else {
            switch(compression) {
            case MTypes_Words:
                break;
            case MTypes_Bytes:
                break;
            }

            // 将可回收小对象 object 组成链表。
            end->next = (MLink*)p;
            end = (MLink*)p;
            nfree++;
        }
    }

    if(nfree) {
        // 将可回收 object 链表归还给 span。
        runtime·MCentral_FreeSpan(&runtime·mheap·central[cl], s, nfree, head.next, end);
    }
}

```

直接使用内存分配器释放函数，收回空闲 object，最终将 span 还给 heap。

### mcentral.c

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

    // 将收集的空闲 object 链表添加到 span.freelist。
    end->next = s->freelist;
    s->freelist = start;
    s->ref -= n;
    c->nfree += n;

    // 如果该 span 满血，将其归还给 heap。
    if(s->ref == 0) {
        runtime·MHeap_Free(&runtime·mheap, s, 0);
    }
}
```

## 2.2 释放内存

在进程入口函数，专门新建了一个 goroutine，用于内存释放。

### proc.c

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

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

死循环，做两件事：超时 2 分钟，强制垃圾回收；每隔 1 分钟，释放一次物理内存。

### mheap.c

```
// Release (part of) unused memory to OS.
// Goroutine created at startup.
// Loop forever.
void runtime·MHeap_Scavenger(void)
{
```

```

g->issystem = true;
g->isbackground = true;

forcegc = 2*60*1e9;
limit = 5*60*1e9;

// 确定循环间隔, 从两个条件中选一个小的, 然后取半。按上面两个数字, 也就是 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();

    // 检查上次 GC 到现在是不是超过 2 分钟了。
    if(now - mstats.last_gc > forcegc) {
        // 启动一个新 goroutine, 进行强制垃圾回收。
        runtime·newproc1(&forcegcchelperv, ..., 0, runtime·MHeap_Scavenger);
        now = runtime·nanotime();
    }

    // 释放物理内存。
    scavenge(k, now, limit);
}

static FuncVal forcegcchelperv = {(void*)(void))forcegcchelper};
static void forcegcchelper(Note *note) { runtime·gc(1); }

```

在执行过程中, 长时间达不到 GC 触发条件是很正常的情况。只是这会导致 **cache** 一直持有空闲 **object**, 让 **span** 无法收回全部内存, 自然也就无法释放它管理的物理内存。因此, 就算达不到触发条件, 定期强制回收也是必要的。

释放物理内存操作, 会扫描 **heap.free** 和 **large** 链表, 查找长期闲置的 **span**。

## mheap.c

```

static void scavenge(int32 k, uint64 now, uint64 limit)
{
    h = &runtime.mheap;
    sumreleased = 0;

```

```

// 检查 heap.free 所有链表里的 span。
for(i=0; i < nelem(h->free); i++)
    sumreleased += scavengelist(&h->free[i], now, limit);

// 检查 heap.large 链表里所有的 span。
sumreleased += scavengelist(&h->large, now, limit);
}

static uintptr scavengelist(MSpan *list, uint64 now, uint64 limit)
{
    if(runtime·MSpanList_IsEmpty(list)) return 0;

    // 遍历链表上所有 span。
    for(s=list->next; s != list; s=s->next) {
        // 检查未使用时间是否超过 5 分钟, npreleased != npages 表示未全部释放。
        if((now - s->unusedsince) > limit && s->npreleased != s->npages) {
            mstats.heap_released += released;
            sumreleased += released;

            // 标记该 span 物理内存被释放, 重新使用时内核会重新分配。
            // mheap.c 142 MHeap_AllocLocked, 基本上是对 Windows VirtualAlloc 的措施。
            s->npreleased = s->npages;

            // 释放该 span 所管理的内存。
            runtime·SysUnused((void*)(s->start << PageShift), s->npages << PageShift);
        }
    }

    return sumreleased;
}

```

所谓释放, 仅仅是通知操作系统, 可以解除物理内存和该段虚拟地址的映射。

内存分配器并没有放弃这段虚拟内存, 在下次重新使用该 span 时, 操作系统会重新为其分配物理内存。因此, 无须销毁这些 span 对象。

### mem\_linux.c

```

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

```

### mem\_darwin.c

```

void runtime·SysUnused(void *v, uintptr n)
{
    // Linux's MADV_DONTNEED is like BSD's MADV_FREE.
}

```

```
runtime·madvise(v, n, MADV_FREE);
}
```

## mem\_windows.c

```
void runtime·SysUnused(void *v, uintptr n)
{
    void *r = runtime·stdcall(runtime·VirtualFree, 3, v, n, (uintptr)MEM_DECOMMIT);
    if(r == nil)
        runtime·throw("runtime: failed to decommit pages");
}
```

## 2.3 状态输出

使用环境变量 `GODEBUG="gctrace=1"` 可输出 GC 相关信息，有助于对程序运行状态进行监控。

在此之前，需了解与统计相关的状态对象和函数。

## malloc.h

```
struct MStats
{
    // General statistics.
    uint64    alloc;           // 当前正在使用 object 容量。
    uint64    total_alloc;     // 自启动以来累计全部分配容量，包括已经被释放的内存。
    uint64    sys;            // 包括 heap_sys、stack、span、cache 在内所有内存总和。
    uint64    nmalloc;        // object 分配总次数。
    uint64    nfree;          // object 释放总次数。

    // Statistics about malloc heap.
    uint64    heap_alloc;     // 等于 alloc。
    uint64    heap_sys;       // mmap - munmap, 不等于 RSS。
    uint64    heap_idle;      // heap 空闲 span 总容量。bytes
    uint64    heap_inuse;     // 正在使用的 span 总容量。bytes
    uint64    heap_released;  // 归还给 OS 的内存容量。bytes
    uint64    heap_objects;   // 当前正在使用的 object 数量。

    // Statistics about garbage collector.
    uint64    next_gc;        // 下次 GC 阈值。
    uint64    last_gc;        // 最后一次 GC 时间。
    uint32    numgc;          // GC 次数。
    bool      enablegc;
    bool      debuggc;

    // 按 sizeclass 分类统计数据。
}
```

```

struct {
    uint32 size;
    uint64 nmalloc;
    uint64 nfree;
} by_size[NumSizeClasses];
};

```

## mgc0.c

```

static void updatememstats(GCStats *stats)
{
    // 重置状态对象。
    if(stats) runtime·memclr((byte*)stats, sizeof(*stats));

    // 统计 stack、cache、sys 等内存容量。
    mstats.sys = mstats.heap_sys + mstats.stacks_sys + mstats.mspan_sys +
        mstats.mcache_sys + mstats.buckhash_sys + mstats.gc_sys + mstats.other_sys;

    // 重置 mstats 相关属性。
    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 链表中空闲的 object 全部归还给 central。
    for(pp=runtime·allp; p=*pp; pp++) {
        c = p->mcache;
        if(c==nil) continue;
        runtime·MCache_ReleaseAll(c);
    }

    // 刷新所有 cache 统计信息。
    cachestats();

    // 遍历所有 span, 统计正在使用 object 数量。此时, cache 上已经没有空闲 object 了。
    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, 整个 span 当一个 object 用。
            mstats.nmalloc++;
            mstats.alloc += s->elemsize;
        } else {
            // 通过 span.ref 计数器就可知道分配出去的 object 数量。

```



```

        mstats.nmalloc += s->ref; // 累加总数
        mstats.by_size[s->sizeclass].nmalloc += s->ref; // 按 sizeclass 累加
        mstats.alloc += s->ref*s->elemsize; // 总分配内存容量
    }
}

smallfree = 0;

// 释放大对象的次数。
mstats.nfree = runtime.mheap.nlargefree;

// 统计小对象释放次数。
for(i = 0; i < nelem(mstats.by_size); i++) {
    // 累加总释放次数。
    mstats.nfree += runtime.mheap.nsmallfree[i];

    // 按 sizeclass 分类累加。
    mstats.by_size[i].nfree = runtime.mheap.nsmallfree[i];

    // 分配次数 = 正在使用未释放 object 数量 + 释放次数。
    mstats.by_size[i].nmalloc += runtime.mheap.nsmallfree[i];

    // 累加小对象总释放容量。
    smallfree += runtime.mheap.nsmallfree[i] * runtime.class_to_size[i];
}

// 分配次数 = 正在使用未释放 object 数量 + 释放次数。
mstats.nmalloc += mstats.nfree;

// 累计分配内存容量 = 正在使用 object 容量 + 历史释放容量。不等于当前 heap 管理的内存容量。
mstats.total_alloc = mstats.alloc + runtime.mheap.largefree + smallfree;

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

// 当前还在使用 object 数量 = 分配次数 - 释放次数。
mstats.heap_objects = mstats.nmalloc - mstats.nfree;
}

```

第一种信息来自垃圾回收函数。

## mgc0.c

```

static void gc(struct gc_args *args)
{
    // ... 各种准备 ....
    t0 = args->start_time; // 这个时间来自 stoptheworld 之前。 runtime.gc mgc0.c:2005

```

```

if(runtime·debug·gctrace) {
    updatememstats(nil);
    heap0 = mstats.heap_alloc;
    obj0 = mstats.nmalloc - mstats.nfree;
}

t1 = runtime·nanotime();
// ... mark ...
t2 = runtime·nanotime();
// ... sweep ...
t3 = runtime·nanotime();
// ... cleanup, next_gc ...
t4 = runtime·nanotime();

mstats.last_gc = t4;
mstats.numgc++;

if(runtime·debug·gctrace) {
    updatememstats(&stats);
    heap1 = mstats.heap_alloc;
    obj1 = mstats.nmalloc - mstats.nfree;

    runtime·printf("gc%d(%d): %D+%D+%D ms, %D -> %D MB %D -> %D (%D-%D) objects,"
        " %D(%D) handoff, %D(%D) steal, %D/%D/%D yields\n",
        mstats.numgc,           // GC 次数
        work.nproc,             // 并行 GC goroutine 数量
        (t2-t1)/1000000,        // mark 耗时 ms
        (t3-t2)/1000000,        // sweep 耗时 ms
        (t1-t0+t4-t3)/1000000,  // cleanup: 准备和结束清理耗时, 包括 stoptheworld.
        heap0>>20,              // 回收前正在使用的内存 MB
        heap1>>20,              // 回收后正在使用的内存 MB
        obj0,                   // 回收前正在使用的 object 数量
        obj1,                   // 回收后正在使用的 object 数量
        mstats.nmalloc,         // 总计 object 分配次数
        mstats.nfree,           // 总计 object 释放次数
    )
}
}

```

+- 垃圾回收次数  
 +- 并发回收 goroutine 数量  
 gc443(1): 0+0+0 ms, 400 -> 160 MB 47 -> 44 (706-662) objects, ...  
 +- 回收前正在使用的 object 数量 (活跃对象, 非缓存 object)  
 +- 回收后正在使用的 object 数量  
 +- 总计释放 object 次数  
 +- 总计分配 object 次数  
 +- 回收后正在使用内存  
 +- 回收前正在使用内存 (仅是正在使用的 object, 不包括空闲 span 等)  
 +- mark + sweep + cleanup 耗时 (cleanup 包括 stoptheworld 时间和结束清理时间)

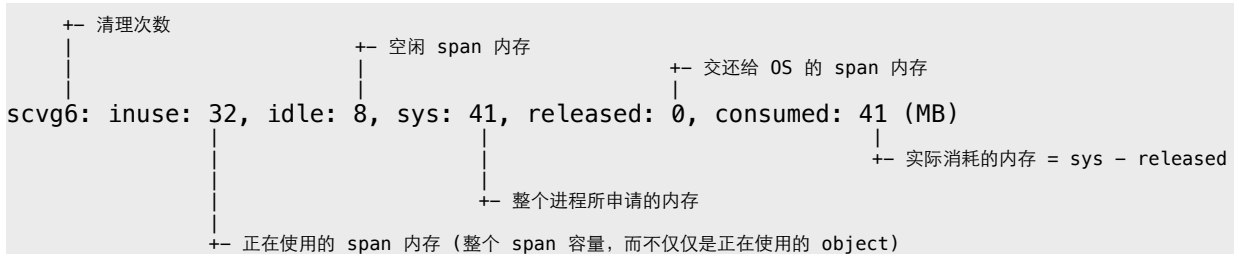
另一种来自物理内存释放函数。

## mheap.c

```
static void scavenge(int32 k, uint64 now, uint64 limit)
{
    sumreleased = 归还给 OS 的内存;

    if(runtime·debug·gctrace > 0) {
        if(sumreleased > 0)
            printf("scvg%d: %D MB released",
                k, // 执行 scavenge 的次数。
                (uint64)sumreleased>>20 // 本次归还给 OS 的内存 MB。
            );

        printf("scvg%d: inuse: %D, idle: %D, sys: %D, released: %D, consumed: %D (MB)",
            k, // 执行 scavenge 的次数。
            mstats.heap_inuse>>20, // 正在使用的 span 内存
            mstats.heap_idle>>20, // 空闲 span 内存
            mstats.heap_sys>>20, // 乱七八糟内存总量
            mstats.heap_released>>20, // 交还给 OS 的内存量
            (mstats.heap_sys - mstats.heap_released)>>20); // 当前实际消耗
    }
}
```



当 `span` 物理内存被释放，释放页数保存在 `npreleased`。累加释放容量到 `released`，表示当前整个进程交还给操作系统的物理内存总量。而当该 `span` 再次被启用时，内核会补齐所释放内存，就应该从 `released` 减掉。

## 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) {
            // (页数 - 以前释放页数)<<4096 = 本次释放页数。MHeap_FreeLocked 合并 span 造成的。
            released = (s->npages - s->npreleased) << PageShift;
            sumreleased += released;
        }
    }
}
```

```

        // 累加释放计数。
        mstats.heap_released += released;

        // 标记释放页数。
        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)
{
    mstats.heap_idle -= s->npages<<PageShift;

    // 如果该 span 以前被释放过, 那么 s-> npreleased > 0, 应该减去。
    mstats.heap_released -= s->npreleased<<PageShift;

    // 重新启用后, OS 会重新为其分配内存, 重置释放标记。
    s->npreleased = 0;
}

```

### 3. Goroutine Scheduler

在 `proc.c` 头部提到调度器几个相关抽象概念。

```
// Goroutine scheduler
// The scheduler's job is to distribute ready-to-run goroutines over worker threads.
//
// The main concepts are:
// G - goroutine.
// M - worker thread, or machine.
// P - processor, a resource that is required to execute Go code.
//     M must have an associated P to execute Go code, however it can be
//     blocked or in a syscall w/o an associated P.
//
// Design doc at http://golang.org/s/go11sched.
```

网上有大牛用地鼠运砖的例子来描述调度器工作流程，说得极好。我也东施效颦，讲讲我理解的调度器。

作为老板 (scheduler)，其目的自然是让地鼠 (M) 在最短时间内，用最快速度将砖 (G) 从仓库运到窑里。除了老板，还有一只地位特殊的地鼠 `sysmon`，专职监工。

刚开始，老板让唯一的马仔，名叫 `M0` 的地鼠通知监工上班，然后组装好所有小车 (P)，因为地鼠必须有车才能工作。当然，除非老板大发慈悲，否则车就这些。好了，现在我们知道基本的工作单位是：地鼠(M) + 小车(P) + 砖(G)。

地鼠 `M0` 工作后，陆陆续续有很多新的砖被丢到仓库里。这家伙心里不平衡，就跟老板说：还有好多小车闲着，你多招几个地鼠来工作吧，否则会耽误公司业务。

新地鼠们来了，大家齐心协力将砖运往窑里。当某只地鼠发现分给自己的那拨运完了，就看看仓库里是否有没分配的砖，如果有就划拉一部分给自己。实在没有，就从别的地鼠那偷一点过来。好地鼠都是活雷锋，为了尽快完成老板交代的任务，可谓尽心尽力。

说起来，还有两种特殊的砖头 (syscall、locked)。作为好奇心严重的小地鼠，看到第一种砖头的时候，多半会停下来 (block)，拿到手上把玩。结果么，车被监工拿走分给其他围观的小地鼠，或者自己主动将车送给酱油众。砖头瞅清楚了，车也没了，工作无法继续。小地鼠四下找找，看有没有闲下来的小车，有就继续工作，没有就只能把手上的砖头放回仓库，等其他有车的伙计运走。另一种

砖头上刻了某只地鼠的名字，表明要专鼠专送，否则会倒大霉。小地鼠没办法，只好将砖连同车一起交给那个家伙，自己失业回宿舍休息，等下次有机会再上岗。

小地鼠们一次一块，在仓库和砖窑间奔波。如果所有的砖都运完了，除了倒霉的 M0，其他地鼠就把车放到空地上，回宿舍睡觉。没办法，老板不放鼠啊。等有了新砖头，自然会通知它们开工的。

运砖工作看着简单，但幺蛾子也不少。比如说，某地鼠可能会看某砖不顺眼，就把砖扔回仓库，去运下一块 (gosched)。还有就是砖忒重，可怜的小地鼠快累死了也没送到窑里，还会耽误吃饭。幸好，善良的监工会通知 (preempt) 它，等吃完饭再接着送。

从某种意义上说，goroutine 本质还是线程池。只是调度处理做得更好，更高效和智能。

要想高效，首先得 multiplexed thread，频繁创建和销毁线程肯定是行不通的。其次，得让线程在阻塞时自动切换，去执行其他任务。具体的做法是将执行状态从线程分离出来，用名为 G 的结构体为线程提供执行现场。

每条 go 语句都会生成一个 G 对象，它包含 8KB 栈内存，可按需扩张。执行时只需修改 SP 寄存器，使其指向 G 所提供的栈，然后 JMP 到目标函数即可。如需要切换其他任务，只需将相关寄存器内容写回 G 对应字段。因为 G 保存了执行现场，任何线程都可以恢复执行该任务。

线程被抽象成 M 对象，如果不给它指定任务，那么就会从本地和全局队列中查找等待运行的 G 对象，如此循环往复。M 还有一个名为 g0 的栈，专门用于执行调度器指令。

如同系统线程需要调度到 CPU 核心才能执行一样，M 也必须关联一个抽象处理器 P 才能工作。P 的数量决定了 goroutine 并发数。默认值 1，但可用环境变量或相关函数修改。

调度器用 P 将 M 和 G 关联起来，大大提升了执行性能。首先，不用为休眠的 M 提供对象缓存，只需在执行时访问 P 缓存即可，减少了内存浪费。其次，将大量的 G 分散到多个 P 本地队列中，减少了全局锁造成的排队。

相对而言，P 的数量是固定的，M 则根据需要新建或复用，两者数量并不一致。例如，某 M 因系统调用时间过长而被收回 P，正好又没有其他休眠 M 可用，调度器就会新建一个来执行任务。最多只能有 256 个 P，但却可创建成千上万的 M。

在之前版本，死循环函数可以一直霸占着 **M**。如果正好  $P = 1$ ，那么其他任务就没机会运行，甚至 **GC** 都会因 **stoptheworld** 时间过长而被卡住。为此，专门增加了抢占式调度，通过设定抢占标志，让 **M** 主动切换。

### 3.1 初始化

从 **bootstrap** 开始。流程注释写在 **proc.c** 里，但实际代码是由汇编实现的。

#### proc.c

```
// The bootstrap sequence is:
//
//   call osinit
//   call schedinit
//   make & queue new G
//   call runtime·mstart
//
// The new G calls runtime·main.

struct Sched {
    uint64    goidgen;

    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;        // idle P's
    uint32     npidle;
    uint32     nm spinning;

    // Global runnable queue.
    G*         runqhead;
    G*         runqtail;
    int32      runqsize;

    // Global cache of dead G's.
    Lock       gflock;
    G*         gfree;
};

Sched runtime·sched;
M      runtime·m0;        // main thread
G      runtime·g0;        // idle goroutine for m0
P**    runtime·allp;      // all P, runtime.h
```

```

M*    runtime·allm;        // all M
G*    runtime·allg;        // all G
G*    runtime·lastg;

```

## asm\_amd64.s

```

TEXT _rt0_go(SB),NOSPLIT,$0

    // 为当前主线程 m0 设置 g0。
    get_tls(BX)
    LEAQ    runtime·g0(SB), CX
    MOVQ    CX, g(BX)
    LEAQ    runtime·m0(SB), AX
    MOVQ    AX, m(BX)

    MOVQ    CX, m_g0(AX) // save m->g0 = g0

    // 初始化
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·hashinit(SB)
    CALL    runtime·schedinit(SB)

    // 创建 main goroutine。
    PUSHQ    $runtime·main·f(SB) // entry
    PUSHQ    $0                  // arg size
    CALL    runtime·newproc(SB)

    // 开始执行。
    CALL    runtime·mstart(SB)

    RET

```

调度器初始化最重要的任务是创建 P。

## proc.c

```

void runtime·schedinit(void)
{
    // 默认最大线程数量限制。如果调度器发现 M 数量超出该值，直接让进程崩溃。
    runtime·sched·maxmcount = 10000;

    // 初始化内存分配器。
    runtime·mallocinit();

    // 初始化当前 M，也就是 m0。
    mcommoninit(m);

    // 分解命令行参数和环境变量。

```



```

runtime·goargs();
runtime·goenvs();
runtime·parsedebgvars();

// 默认 GOMAXPROCS = 1, 最多不能超过 256。
procs = 1;
p = runtime·getenv("GOMAXPROCS");
if(p != nil && (n = runtime·atoi(p)) > 0) {
    if(n > MaxGomaxprocs) n = MaxGomaxprocs;
    procs = n;
}

// P 指针数组。
runtime·allp = runtime·malloc((MaxGomaxprocs+1)*sizeof(runtime·allp[0]));

// 调整 P 数量。
proccresize(procs);
}

```

调整数量除按需创建和释放 P 对象外, 还会导致原 G 队列被重新分配。

## runtime.h

```

struct P
{
    int32    id;
    uint32   status;           // one of Pidle/Prunning/...
    P*       link;
    M*       m;               // back-link to associated M (nil if idle)
    MCache*  mcache;

    // Queue of runnable goroutines.
    G**      runq;
    int32    runqhead;
    int32    runqtail;
    int32    runqsize;

    // Available G's (status == Gdead)
    G*       gfree;
    int32    gfreecnt;
};

```

## proc.c

```

// Change number of processors. The world is stopped, sched is locked.
static void proccresize(int32 new)
{
    // 当前值。
    old = runtime·gomaxprocs;

```

```

// 初始化 P。
for(i = 0; i < new; i++) {
    p = runtime·allp[i];

    // 新建 P。
    if(p == nil) {
        p = (P*)runtime·mallocgc(sizeof(*p), 0, FlagNoInvokeGC);
        p->id = i;
        p->status = Pgcstop;
        runtime·atomicstorep(&runtime·allp[i], p);
    }

    // 创建内存分配器 cache。
    if(p->mcache == nil) {
        if(old==0 && i==0)
            p->mcache = m->mcache; // bootstrap
        else
            p->mcache = runtime·allocmcache();
    }

    // 新建 G 队列。
    if(p->runq == nil) {
        p->runqsize = 128;
        p->runq = (G**)runtime·mallocgc(p->runqsize*sizeof(G*), 0, FlagNoInvokeGC);
    }
}

// 将原来 P 上所有待运行 G 转移到全局队列。
for(i = 0; i < old; i++) {
    p = runtime·allp[i];
    while(gp = runqget(p))
        globrunqput(gp);
}

// 将 G 从全局队列重新分配到 P。
for(i = 1; runtime·sched.runqhead; i++) {
    gp = runtime·sched.runqhead;
    runtime·sched.runqhead = gp->schedlink;
    runqput(runtime·allp[i%new], gp);
}
runtime·sched.runqtail = nil;
runtime·sched.runqsize = 0;

// 如果是减小 GOMAXPROCS, 那么就释放掉多余的 P。
for(i = new; i < old; i++) {
    p = runtime·allp[i];
    runtime·freemcache(p->mcache);
}

```

```

    p->mcache = nil;
    gfpurge(p);
    p->status = Pdead;
    // can't free P itself because it can be referenced by an M in syscall
}

// 为当前 M 关联 P。
acquirep(p);

// 重置其他 P 状态, 放入空闲队列。
for(i = new-1; i > 0; i--) {
    p = runtime·allp[i];
    p->status = Pidle;
    pidleput(p);
}

// 设置新的 GOMAXPROCS 值。
runtime·atomicstore((uint32*)&runtime·gomaxprocs, new);
}

```

完成初始化后, 开始执行 `main goroutine`。在 `main.main` 之前, 还分别创建了专职系统监控的 `M(sysmon)` 和用于垃圾回收的 `G(scavenger)`。

## proc.c

```

// The main goroutine.
void runtime·main(void)
{
    // Max stack size is 1 GB on 64-bit, 250 MB on 32-bit.
    if(sizeof(void*) == 8)
        runtime·maxstacksize = 1000000000;
    else
        runtime·maxstacksize = 250000000;

    // 专门创建一个线程, 运行调度器监控。
    newm(sysmon, nil);

    if(m != &runtime·m0)
        runtime·throw("runtime·main not on m0");

    // 创建 goroutine 运行垃圾回收。
    runtime·newproc1(&scavenger, nil, 0, 0, runtime·main);

    // 执行用户代码 main.init 初始化函数。
    main·init();

    // 执行用户代码入口 main.main。
    main·main();
}

```

```
// 退出，终止进程。
runtime·exit(0);
}
```

## 3.2 创建任务

编译器将 `go` 语句翻译成 `runtime·newproc` 函数调用，创建 `G` 对象。

### runtime.h

```
struct G
{
    uintptr stackguard0; // can be set to StackPreempt as opposed to stackguard
    uintptr stackbase;
    Gobuf    sched;
    uintptr stackguard; // same as stackguard0, but not set to StackPreempt
    uintptr stack0;
    uintptr stacksize;
    G*      alllink;      // on allg
    void*    param;        // passed parameter on wakeup
    int16    status;
    int64    goid;
    int8*    waitreason;   // if status==Gwaiting
    G*      schedlink;
    bool     preempt;      // preemption signal, duplicates stackguard0 = StackPreempt
    M*       m;            // for debuggers, but offset not hard-coded
    M*       lockedm;
    uintptr  gopc;         // pc of go statement that created this goroutine
};
```

### proc.c

```
// Create a new g running fn with siz bytes of arguments.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to this.
void runtime·newproc(int32 siz, FuncVal* fn, ...)
{
    runtime·newproc1(fn, argp, siz, 0, runtime·getcallerpc(&siz));
}
```

和内存分配器一样，调度器对 `G` 做了复用缓存处理。`gfgget` 首先从 `P.gfree` 获取可复用 `G`，如果失败，就从 `sched.gfree` 转移一批过来（填满 32 个坑）。实在没有，才新建 `G` 实例。

```

// Create a new g running fn with narg bytes of arguments starting
// at argp and returning nret bytes of results. callerpc is the
// address of the go statement that created this. The new g is put
// on the queue of g's waiting to run.
G* runtime.newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    siz = narg + nret;
    siz = (siz+7) & ~7;

    // 尝试从 p.gfree、runtime.sched.gfree 链表拿出一个 G 复用。
    if((newg = gfget(m->p)) != nil) {
        ...
    } else {
        // 新建 G, stacksize = 8KB。
        newg = runtime.malg(StackMin);

        // 放到全局双向链表。
        runtime.lock(&runtime.sched);
        if(runtime.lastg == nil)
            runtime.allg = newg;
        else
            runtime.lastg->alllink = newg;
        runtime.lastg = newg;
        runtime.unlock(&runtime.sched);
    }

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

    // 初始化 G.sched, 该结构用于保存执行现场数据。
    runtime.memclr((byte*)&newg->sched, sizeof newg->sched);
    newg->sched.sp = (uintptr)sp;                // 类似 SP 寄存器。
    newg->sched.pc = (uintptr)runtime.goexit;    // 类似 PC 寄存器。
    newg->sched.g = newg;

    // 非常关键的一次调用。坑爹啊! 这是执行完 G, 调用 goexit 的关键。
    // stack.c runtime.gostartcallfn -> sys_x86.c runtime.gostartcall
    runtime.gostartcallfn(&newg->sched, fn);

    // 将 G 放入当前 M->P 队列尾部。
    newg->gopc = (uintptr)callerpc;                // 创建 G 的函数指针。
    newg->status = Grunnable;
    runqput(m->p, newg);

    // 尝试唤醒 M 开始执行。(有空闲 P, 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)
        wakep();

    return newg;
}

```

获取 G 以后，将函数实参入栈，最后尝试唤醒某个 M 开始执行。

```

// Tries to add one more P to execute G's.
// Called when a G is made runnable (newproc, ready).
static void wakep(void)
{
    // be conservative about spinning threads
    if(!runtime·cas(&runtime·sched.nmspinning, 0, 1)) return;
    startm(nil, true);
}

```

在 M 执行完 G 任务后，需要调用 runtime·goexit 清理现场，重新循环。

通过 runtime·gostartcallfn -> sys\_x86.c runtime·gostartcall，将 goexit 地址预先入栈，等 G 函数执行结束，其 ret 指令会将该地址 pop 到 IP/PC 寄存器，完成调用。

## runtime.h

```

struct Gobuf
{
    uintptr    sp;
    uintptr    pc;
    G*         g;
    uintptr    ret;
    void*      ctxt;
    uintptr    lr;
};

```

## sys\_x86.c

```

void runtime·gostartcall(Gobuf *gobuf, void (*fn)(void), void *ctxt)
{
    uintptr *sp;

    sp = (uintptr*)gobuf->sp;

    // 此时 pc 指向 goexit，将其地址入栈。
    *--sp = (uintptr)gobuf->pc;

    // 调整栈，尤其是 pc 指向 goroutine 函数。
}

```

```

    gobuf->sp = (uintptr)sp;
    gobuf->pc = (uintptr)fn;
    gobuf->ctxt = ctxt;
}

```

### 3.3 执行任务

M 除了使用 G stack 外，还有自己的 g0 stack，专门执行调度操作。

#### runtime.h

```

struct M
{
    G*      g0;                // goroutine with scheduling stack
    void    (*mstartfn)(void);
    G*      curg;              // current running goroutine
    P*      p;                 // attached P for executing Go code
    P*      nextp;
    int32   id;
    bool    spinning;
    Note    park;
    M*      alllink;           // on allm
    M*      schedlink;
    MCache* mcache;
    G*      lockedg;
    uintptr createstack[32];    // Stack that created this thread.
    uint32   locked;           // tracking for LockOSThread
    M*      nextwaitm;         // next M waiting for lock
};

```

开始工作前，必须获得一个空闲 P。

```

// Schedules some M to run the p (creates an M if necessary).
// If p==nil, tries to get an idle P, if no idle P's returns false.
static void startm(P *p, bool spinning)
{
    // 获取空闲 P。
    if(p == nil) {
        p = pidleget();

        // 如果没有可用 P，获取 M 是没有意义的。
        if(p == nil) {
            return;
        }
    }
}

```

```

// 从 runtime.sched.midle 队列提取空闲 M。
mp = mget();

// 如果没有可用 M, 新建。
if(mp == nil) {
    newm(fn, p);
    return;
}
}

```

新建的 M 对象关联一个系统线程, 在此会检查 M 总数是否超出阈值。

### proc.c

```

// Create a new m. It will start off with a call to fn, or else the scheduler.
static void newm(void(*fn)(void), P *p)
{
    mp = runtime.allocm(p);

    mp->nextp = p;
    mp->mstartfn = fn;

    // 创建系统线程。
    runtime.newosproc(mp, (byte*)mp->g0->stackbase);
}

// Allocate a new m unassociated with any thread.
M* runtime.allocm(P *p)
{
    mp = runtime.cnew(mtype);
    mcommoninit(mp);

    // 创建 M.g0
    // In case of cgo, pthread_create will make us a stack.
    // Windows will layout sched stack on OS stack.
    if(runtime.iscgo || 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 M 总数量(默认10000), 超出会引发进程崩溃。

```



```

    checkmcount();

    runtime·mpreinit(mp);

    // Add to runtime·allm so garbage collector doesn't free m
    // when it is just in a register or thread-local storage.
    mp->alllink = runtime·allm;
    runtime·atomicstorep(&runtime·allm, mp);
}

```

继续执行的秘密在 `newosproc` 函数里, `runtime·mstart`。

### os\_linux.c

```

void runtime·newosproc(M *mp, void *stk)
{
    ret = runtime·clone(flags, stk, mp, mp->g0, runtime·mstart);
}

```

M 有两种执行方式: 像普通线程那样执行函数, 如 `newm(sysmon)`; 循环 `schedule`, 从队列中获取 G。

### proc.c

```

// Called to start an M.
void runtime·mstart(void)
{
    if(g != m->g0)
        runtime·throw("bad runtime·mstart");

    // 如果 newm 提供了执行函数, 那么直接执行。
    // 执行完成后, 这个 M 会加入到 G 的抢夺行列。
    if(m->mstartfn)
        m->mstartfn();

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

    // 调度器核心执行函数。
    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.
}

// Associate p and the current m.
static void acquirep(P *p)
{
    m->mcache = p->mcache;    // 设置小对象分配缓存。
    m->p = p;
    p->m = m;
    p->status = Prunning;
}

```

除了 P 本地队列，还会定期检查全局队列，住在这里的多半是 netpoll 和 syscall 遗民。

```

// 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，除返回一个外，还尝试转移一批到本地队列。
    tick = m->p->schedtick;
    if(tick - (((uint64)tick*0x4325c53fu)>>36)*61 == 0 && runtime·sched·runqsize > 0) {
        gp = globrunqget(m->p, 1);
    }

    // 从当前 P.runq 中提取 G。
    if(gp == nil) {
        gp = runqget(m->p);
    }

    // 如果为空，那么就从其他队列中查找。
    if(gp == nil) {
        gp = findrunnable(); // blocks until work is available
    }

    // 如果发现这个 G 被锁定到某个 M 上，那么将 P 转交给那个锁定 M 继续执行。
    if(gp->lockedm) {
        // Hands off own p to the locked m,
        // then blocks waiting for a new p.
        startlockedm(gp);
        goto top;
    }

    // 执行 G 任务。
    execute(gp);
}

```

```
}

```

官方文档提及的 **work-stealing** 算法，就在 **findrunnable** 里。

```
// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from global queue, poll network.
static G* findrunnable(void)
{
    G *gp;
    P *p;

top:
    // 尝试从当前 P 队列获取。
    gp = runqget(m->p);
    if(gp) return gp;

    // 尝试从全局队列获取一批 G，默认数量 sched.runqsize/gomaxprocs+1。
    if(runtime·sched.runqsize) {
        gp = globrunqget(m->p, 0); // 返回第一个 G，其他的都转移到当前 P 的队列中。
        if(gp) return gp;
    }

    // ... 省略 netpoll ...

    // 随机从其他 P 里获取。
    for(i = 0; i < 2*runtime·gomaxprocs; i++) {
        p = runtime·allp[runtime·fastrand1()%runtime·gomaxprocs];

        if(p == m->p) // 如果是自己，gp == nil，继续循环。
            gp = runqget(p);
        else
            gp = runqsteal(m->p, p); // 偷一半到自己的 P 队列，并返回一个。

        if(gp) return gp;
    }
stop:
    // 再次尝试全局队列。
    if(runtime·sched.runqsize) {
        gp = globrunqget(m->p, 0);
        return gp;
    }

    // 还是找不到，释放 P。
    p = releasep();
    pidleput(p);

    // 再次检查所有的 P 队列。

```

```

    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p && p->runqhead != p->runqtail) {
            p = pidleget();
            if(p) {
                acquirep(p); // 将 P 拿回来。
                goto top;    // 再来一次。
            }
            break;
        }
    }

    // 折腾半天, 还是什么收获都没有, 只好 stop, 将自己放入 sched.middle 队列。
    stopm();
    goto top;
}

```

如果发现 G 锁定了别的 M, 那么就将 P 和 G 交给它, 自己回空闲队列睡觉。

```

// Schedules the locked m to run the locked gp.
static void startlockedm(G *gp)
{
    mp = gp->lockedm;

    // 将 P 释放, 交给 mp。
    p = releasep();
    mp->nextp = p;

    // 唤醒 pm。
    runtime·notewakeup(&mp->park);

    // 休眠。
    stopm();
}

```

回到任务执行流程, schedule -> execute -> gogo, 这段汇编代码才是关键。

```

// Schedules gp to run on the current M.
// Never returns.
static void execute(G *gp)
{
    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        // 用 G.sched.sp 设定 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 函数指针放到 BX。
    JMP     BX                      // 跳转, 执行 goroutine 函数。
```

兜兜转转，总算 JMP 到 goroutine 函数。前面提到过，该函数 ret 指令使得预先压入的 goexit 地址 pop 到 PC/IP 寄存器，完成清理工作。

```
// Finishes execution of the current goroutine.
void runtime·goexit(void)
{
    runtime·mcall(goexit0);
}

static void goexit0(G *gp)
{
    gp->status = Gdead;
    m->curg = nil;

    // 释放 G 额外分配的栈帧。
    runtime·unwindstack(gp, nil);

    // 将 G 放回 P.gfree。
    // 如果空闲数量过多，就移交一部分给 sched.gfree，以便其他 P 获取。
    gput(m->p, gp);

    // 再次调用 schedule 循环。
    schedule();
}
```

在 goexit0 尾部，再次调用 schedule，开始下一轮工作。

### 3.4 系统调用

用标准库包装系统调用，插入 `entersyscall` 或 `entersyscallblock`，以配合调度器工作。两者区别是：`entersyscall` 的 `P` 可能被 `sysmon` 拿走，`entersyscallblock` 则是主动将 `P` 交给其他 `M` 去执行任务。

#### proc.c

```
// The same as runtime.entersyscall(), but with a hint that the syscall is blocking.
void entersyscallblock(int32 dummy)
{
    p = releasep();
    handoffp(p);

    g->stackguard0 = StackPreempt; // 可抢占标志
}
```

转移 `P` 函数 `handoffp` 会进行各种 `startm` 尝试，直到把 `P` 放回 `sched.pidle`。

```
// Hands off P from syscall or locked M.
static void handoffp(P *p)
{
    // if it has local work, start it straight away
    if(p->runqhead != p->runqtail || runtime.sched.runqsize) {
        startm(p, false);
        return;
    }

    // no local work, check that there are no spinning/idle M's,
    // otherwise our help is not required
    if(runtime.atomicload(&runtime.sched.nmspinning) +
        runtime.atomicload(&runtime.sched.npidle) == 0 && // TODO: fast atomic
        runtime.cas(&runtime.sched.nmspinning, 0, 1)) {
        startm(p, true);
        return;
    }

    if(runtime.sched.runqsize) {
        runtime.unlock(&runtime.sched);
        startm(p, false);
        return;
    }

    // If this is the last running P and nobody is polling network,
    // need to wakeup another M to poll network.
    if(runtime.sched.npidle == runtime.gomaxprocs-1 &&
        runtime.atomicload64(&runtime.sched.lastpoll) != 0) {
```

```

        runtime.unlock(&runtime.sched);
        startm(p, false);
        return;
    }

    pidleput(p);
}

```

从系统调用退出，必须检查关联 **P** 是否有效，以确定能否继续工作。在获取 **P** 失败后，将 **G** 放回全局队列，由其他 **M** 完成该任务。至于这个失去工作的 **M**，将回到空闲队列，直到下次被唤醒。

```

void runtime·exitsyscall(void)
{
    if(exitsyscallfast()) {
        // There's a cpu for us, so we can run.
        return;
    }

    // Call the scheduler.
    runtime·mcall(exitsyscall0);
}

static bool exitsyscallfast(void)
{
    // 检查 Psyscall 状态，如果被 sysmon 拿走，状态值会被修改。
    if(m->p && m->p->status == Psyscall &&
        runtime·cas(&m->p->status, Psyscall, Prunning)) {
        return true;
    }

    // 获取其他空闲 P。
    m->p = nil;
    if(runtime·sched.pidle) {
        p = pidleget();
        if(p) {
            acquirep(p);
            return true;
        }
    }

    return false;
}

// runtime·exitsyscall slow path on g0.
// Failed to acquire P, enqueue gp as runnable.
static void exitsyscall0(G *gp)

```

```

{
    // 获取空闲 P。
    p = pidleget();

    // 获取失败，将 G 放到全局队列。
    if(p == nil)
        globrunqput(gp);

    // 获取成功，关联 P，继续执行。
    if(p) {
        acquirep(p);
        execute(gp); // Never returns.
    }

    // 失败，停止。
    stopm();
}

```

可主动调用 `runtime.Gosched` 中断任务。当前 G 被放回全局队列，M 执行其他任务。

#### proc.c

```

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

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

    // 让出 M。
    gp->m = nil;
    m->curg = nil;

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

    // 让 M 执行其他任务。
    schedule();
}

```

与之类似的 `runtime·park` 虽然也会中断当前任务，却没有被放回等待队列，而是一直阻塞到被专门唤醒。这个设计被用于 `channel` 实现。

#### proc.c

```

// Puts the current goroutine into a waiting state and unlocks the lock.

```



```
// The goroutine can be made runnable again by calling runtime·ready(gp).
void runtime·park(void(*unlockf)(Lock*), Lock *lock, int8 *reason)
{
    runtime·mcall(park0);
}

// runtime·park continuation on g0.
static void park0(G *gp)
{
    // 注意状态是 Gwaiting。
    gp->status = Gwaiting;

    // 让出 M, 让它执行其他任务。
    gp->m = nil;
    m->curg = nil;
    schedule();
}

// Mark gp ready to run.
void runtime·ready(G *gp)
{
    // 修改状态为 Grunnable, 重新放回本地队列。
    gp->status = Grunnable;
    runqput(m->p, gp);
}
```

### 3.5 系统监控

系统监控线程 `sysmon` 主要起三个作用：

- 将 `netpoll` 结果放到全局队列。
- 收回长时间处于 `Psyscall` 状态的 `P`。
- 向长时间运行的 `G` 发出抢占通知。

#### proc.c

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

        // poll network if not polled for more than 10ms
        if(lastpoll != 0 && lastpoll + 10*1000*1000 < now) {
            gp = runtime·netpoll(false); // non-blocking
            if(gp) {
```

```

        injectglist(gp);
    }
}

// retake P's blocked in syscalls
// and preempt long running G's
if(retake(now))
    idle = 0;
else
    idle++;
}
}

static uint32 retake(int64 now)
{
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p==nil) continue;
        s = p->status;

        if(s == Psyscall) {
            // Retake P from syscall if it's there for more than 1 sysmon tick (20us).
            if(runtime·cas(&p->status, s, Pidle)) {
                handoffp(p);
            }
        } else if(s == Prunning) {
            // Preempt G if it's running for more than 10ms.
            if(pd->schedwhen + 10*1000*1000 > now) continue;

            preemptone(p); // 发出抢占通知
        }
    }
}

// Tell the goroutine running on processor P to stop.
// Returns true if preemption request was issued.
static bool preemptone(P *p)
{
    M *mp = p->m;
    G *gp = mp->curg;

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

    return true;
}

```

所谓抢占通知，不过是设置 G 的几个成员。那么抢占行为何时发生呢？

编译器会在每个非内联函数头部插入 "runtime.morestack" 汇编指令，用于判断和扩大执行栈。猫腻就在这里。

```
(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   0x227a0 <runtime.morestack00>
    0x0000000000002013 <+19>:   jmp     0x2000 <main.test>
    0x0000000000002015 <+21>:   sub     rsp,0x50
```

跟踪 morestack，最终在 runtime.newstack 一大堆代码里找到切换操作。

### asm\_amd64.s

```
TEXT runtime.morestack00(SB),NOSPLIT,$0
    get_tls(CX)
    MOVQ    m(CX), BX
    MOVQ    $0, AX
    MOVQ    AX, m_moreframesize(BX)
    MOVQ    $runtime.morestack(SB), AX
    JMP     AX

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

### stack.c

```
void runtime.newstack(void)
{
    gp = m->curg;

    // 检查抢占标志
    if(gp->stackguard0 == (uintptr)StackPreempt) {
        runtime.gosched0(gp);    // never return
    }
}
```

正如官方所言，1.2 抢占调度还很原始，只有在调用非内联函数时才有可能引发抢占调度。另外，发出抢占通知的并不仅仅是 sysmon，调度器在很多地方都会设置抢占标志。

### 3.6 分段栈

M 使用 G 自带栈内存，而非线程堆栈。执行过程中，`runtime.newstack` 会做出判断，按需增加新的栈帧。这种方式通常被称作分段栈。1.2 将初始栈帧从 4KB 提高到 8KB，据测试可提升执行性能。

看看创建 G 对象时，如何设置栈信息。

#### proc.c

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

    if(stacksize >= 0) {
        if(g == m->g0) {
            // running on scheduler stack already.
            stk = runtime·stackalloc(StackSystem + stacksize);
        } else {
            // have to call stackalloc on scheduler stack.
            g->param = (void*)(StackSystem + stacksize);
            runtime·mcall(mstackalloc);
            stk = g->param;
            g->param = nil;
        }

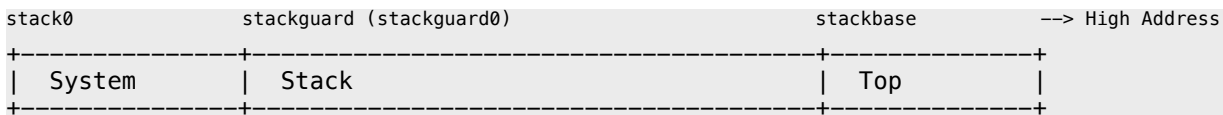
        newg->stacksize = StackSystem + stacksize;
        newg->stack0 = (uintptr)stk;
        newg->stackguard = (uintptr)stk + StackGuard;
        newg->stackguard0 = newg->stackguard;
        newg->stackbase = (uintptr)stk + StackSystem + stacksize - sizeof(Stktop);
        runtime·memclr((byte*)newg->stackbase, sizeof(Stktop));
    }

    return newg;
}
```

#### stack.c

```
void* runtime·stackalloc(uint32 n)
{
    return runtime·malloccg(n, 0, FlagNoProfiling|FlagNoGC|FlagNoZero|FlagNoInvokeGC);
}
```

栈结构有些复杂，下面是简易示意图。



其中 `stackbase`、`stackguard` 指针作用类似 BP、SP 寄存器，`Top` 段保存前栈帧指针，`System` 段用于信号处理等系统用途。

当需要扩张 G 栈内存时，会调用 `morestack -> newstack` 创建新栈帧。

## stack.c

```
// Called from runtime.newstackcall or from runtime.morestack when a new
// stack segment is needed. Allocate a new stack big enough for
// m->moreframesize bytes, copy m->moreargsize bytes to the new frame,
// and then act as though runtime.lessstack called the function at
// m->morepc.
void runtime.newstack(void)
{
    framesize = m->moreframesize;
    argsize = m->moreargsize;
    gp = m->curg;
    oldstatus = gp->status;
    sp = gp->sched.sp;

    // 栈溢出检查。
    if(sp < gp->stackguard - StackGuard) {
        runtime.throw("runtime: split stack overflow");
    }

    if(newstackcall && m->morebuf.sp - sizeof(Stktop) - argsize - 32 > gp->stackguard) {
        // special case: called from runtime.newstackcall (framesize==1)
        // to call code with an arbitrary argument size,
        // and we have enough space on the current stack.
        // the new Stktop* is necessary to unwind, but
        // we don't need to create a new segment.
        top = (Stktop*)(m->morebuf.sp - sizeof(*top));
        stk = (byte*)gp->stackguard - StackGuard;
        free = 0;
    } else {
        // allocate new segment.

        // 计算实际需要扩张的栈内存大小。
        framesize += argsize;
        framesize += StackExtra;    // room for more functions, Stktop.
        if(framesize < StackMin) framesize = StackMin;
        framesize += StackSystem;
```

```

// 所有栈帧。
gp->stacksize += framesize;

// 不能超出最大栈内存限制。
if(gp->stacksize > runtime·maxstacksize) {
    runtime·throw("stack overflow");
}

// 为新栈帧分配内存。
stk = runtime·stackalloc(framesize);
top = (Stktop*)(stk+framesize-sizeof(*top));
free = framesize;
}

// 在新栈帧 top 区记录原栈帧相关指针。
top->stackbase = gp->stackbase;
top->stackguard = gp->stackguard;

// 在新栈帧 top 区域记录扩张信息。
top->gobuf = m->morebuf;
top->argp = m->moreargp;
top->argsize = argsize;
top->free = free;

// 清除 M 上的扩张信息。
m->moreargp = nil;
m->morebuf.pc = (uintptr)nil;
m->morebuf.lr = (uintptr)nil;
m->morebuf.sp = (uintptr)nil;

// 修改 G 栈信息, 指向新栈帧。
gp->stackbase = (uintptr)top;
gp->stackguard = (uintptr)stk + StackGuard;
gp->stackguard0 = gp->stackguard;
}

```

任务执行结束, goexit 用 unwindstack 释放所有新增的栈帧。

## panic.c

```

// Free stack frames until we hit the last one
// or until we find the one that contains the sp.
void runtime·unwindstack(G *gp, byte *sp)
{
    // 因为初始栈帧 top 区域是空的, 因此这个循环只是对后分配的栈帧做处理。
    while((top = (Stktop*)gp->stackbase) != 0 && top->stackbase != 0) {
        stk = (byte*)gp->stackguard - StackGuard;
    }
}

```

```
// 检查参数 sp 是否在当前栈帧内，如果是，终止释放。
if(stk <= sp && sp < (byte*)gp->stackbase) break;

// 释放当前栈帧前，修改 G 参数，使其指向前一栈帧。
gp->stackbase = top->stackbase;
gp->stackguard = top->stackguard;
gp->stackguard0 = gp->stackguard;

// 根据当前栈帧 top 内的大小信息，释放该栈帧。
if(top->free != 0) {
    gp->stacksize -= top->free;
    runtime.stackfree(stk, top->free);
}
}
```

### 3.7 状态输出

使用环境变量 `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 `SCHED 1003ms` output. The output string is: `gomaxprocs=2 idleprocs=2 threads=5 idlethreads=2 runqueue=0 [0 0]`. Vertical dashed lines connect each parameter to its meaning:

- `gomaxprocs=2` is linked to `GOMAXPROCS, P 总数` (Total number of P, GOMAXPROCS).
- `idleprocs=2` is linked to `空闲 P 数量` (Number of idle P).
- `threads=5` is linked to `M 总数` (Total number of M).
- `idlethreads=2` is linked to `空闲 M 数量` (Number of idle M).
- `runqueue=0` is linked to `各 P.G 本地队列` (Local queue for each P.G).
- The `[0 0]` part is linked to `G 全局队列` (Global queue for G).
- The `1003ms` part is linked to `跟踪耗时` (Tracking time).

更详细的信息，需指定 "scheddetail=1"。

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

```
SCHED 1008ms: gomaxprocs=2 idleprocs=2 threads=5 idlethreads=2 runqueue=0 gcwaiting=0
nmiddlelocked=1 nmspinning=0 stopwait=0 sysmonwait=0
```

```
P0: status=0 schedtick=1281 syscalltick=863 m=-1 runqsize=0/128 gfreecnt=1
P1: status=0 schedtick=1301 syscalltick=860 m=-1 runqsize=0/128 gfreecnt=1
M2: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=0 ... spinning=0 lockedg=-1
M1: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=1 ... spinning=0 lockedg=-1
M0: p=-1 curg=4 mallocing=0 throwing=0 gcing=0 locks=2 ... spinning=0 lockedg=-1
G1: status=4(sleep) m=-1 lockedm=-1
G2: status=3() m=3 lockedm=-1
G4: status=3(stack split) m=0 lockedm=-1
```

## 4. Channel

如果没有 channel, goroutine 会失色不少。

### 4.1 Channel

简单点说, channel 就是 FIFO 队列, 多个 goroutine 排队进行收发操作。同步模式, 从排队中寻找一个能直接与之交换数据的对象; 异步模式, 则是围绕着缓冲区空位排队。

SudoG 封装了 G 和 elem 指针, WaitQ 则是排队链表。

chan.c

```
struct SudoG
{
    G*      g;           // g and selgen constitute
    uint32  selgen;      // a weak pointer to g
    SudoG*  link;
    int64   releasetime;
    byte*   elem;        // data element
};

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

所有等待发送和接收的 G 都保存在 sendq、recvq 链表; dataqsiz 是缓冲区容量 (元素数量), sendx、recvq、qcount 共同维护基于数组的环状队列状态。

chan.c

```
// The garbage collector is assuming that Hchan can only contain pointers into the stack
// and cannot contain pointers into the heap.
struct Hchan
{
    uintgo  qcount;      // total data in the q
    uintgo  dataqsiz;    // size of the circular q
    uint16  elemsize;
    uint16  pad;         // ensures proper alignment of the buffer that follows Hchan
    bool    closed;
    Alg*    elemalg;     // interface for element type
    uintgo  sendx;       // send index
```



```

uintgo  recvx;    // receive index
WaitQ   recvq;    // list of recv waiters
WaitQ   sendq;    // list of send waiters
Lock;
};

```

创建 **channel** 对象时，会检查元素类型长度，一次性分配包括缓冲区在内的全部内存。

## chan.c

```

Hchan* runtime·makechan_c(ChanType *t, int64 hint)
{
    elem = t->elem;

    // 元素类型不能大于 64KB。
    if(elem->size >= (1<<16))
        runtime·throw("makechan: invalid channel element type");

    // 为 channel 和缓冲 slot 分配内存。
    c = (Hchan*)runtime·mallocgc(sizeof(*c) + hint*elem->size, ..., 0);
    c->elemsize = elem->size;
    c->elemalg = elem->alg;    // 该类型接口，比如复制数据等方法。
    c->dataqsiz = hint;

    return c;
}

```

## 4.2 Send

加锁，确保在同一时刻只有一个操作。想想也很正常，否则这个底层状态实在不好维护。

在同一个函数里实现了同步和异步版本，参数 **pres** 用来接收操作是否成功。

## chan.c

```

/*
 * generic single channel send/recv
 * if the bool pointer is nil, then the full exchange will occur. if pres is not nil,
 * then the protocol will not sleep but return if it could not complete.
 */
void runtime·chansend(ChanType *t, Hchan *c, byte *ep, bool *pres, void *pc)
{
    // 如果 channel == nil
    if(c == nil) {
        // 想要立即获取状态，返回 false。
        if(pres != nil) {

```

```

        *pres = false;
        return;
    }

    // park 会释放 M, 但不会把 G 放回队列。如果没有人唤醒(ready)它, 那就一直阻塞。
    runtime·park(nil, nil, "chan send (nil chan)");
    return; // not reached
}

// 这把锁的粒度是不是太大了?
runtime·lock(c);

// 如果向 closed channel 发数据, 直接 panic。
if(c->closed) goto closed;

// 如果是 buffered channel, 那么执行异步版本。
if(c->dataqsiz > 0) goto asynch;

// 从 recvq 里抓一个 G 出来。
sg = dequeue(&c->recvq);
if(sg != nil) {
    // 有人接收, 就私下一对一解决, 没 channel 什么事了, 释放锁。
    runtime·unlock(c);

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

    // 利用接口方法, 拷贝数据。
    if(sg->elem != nil)
        c->elemalg->copy(c->elemsize, sg->elem, ep);

    // 唤醒这个因 park 而处于 Waiting 状态的接收人。
    runtime·ready(gp);

    return;
}

// 将当前 G 打包放到 sendq 里面排队去。
mysg.elem = ep;
mysg.g = g;
mysg.selgen = NOSELGEN;
g->param = nil;
enqueue(&c->sendq, &mysg);

// 释放 M, 将当前 G 改成 Waiting, 阻塞到有人唤醒为止。
runtime·park(runtime·unlock, c, "chan send");
return;

```

```

async:
    if(c->closed) goto closed;

    // 如果缓冲 slots 已满, 打包放到 sendq 里, 等待被唤醒。
    if(c->qcount >= c->dataqsiz) {
        msg.g = g;
        msg.elem = nil;
        msg.selgen = NOSELGEN;
        enqueue(&c->sendq, &msg);
        runtime·park(runtime·unlock, c, "chan send");

        runtime·lock(c);

        // 被唤醒, 尝试再次向缓冲区发送数据。
        goto async;
    }

    // 将数据拷贝到 slots。
    c->elemalg->copy(c->elemsize, chanbuf(c, c->sendx), ep);
    if(++c->sendx == c->dataqsiz) c->sendx = 0;
    c->qcount++;

    // 现在 slots 有数据了, 如果 recvg 正好有人, 就唤醒它接收数据, 没有就算了。
    sg = dequeue(&c->recvg);
    if(sg != nil) {
        gp = sg->g;
        runtime·unlock(c);
        runtime·ready(gp);
    } else
        runtime·unlock(c);

    return;

closed:
    runtime·unlock(c);
    runtime·panicstring("send on closed channel");
}

```

不算复杂, 总结一下:

- 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, 直接将数据拷贝给它, 否则排队、阻塞。
- 异步发送: 如缓冲区未满, 将数据拷贝到缓冲区, 否则排队、阻塞。

## 4.3 Receive

接收和发送过程类似，规则如下：

- 从 nil channel 接收数据，阻塞。
- 从 closed channel 接受数据，直接返回。
- 同步接收：如有发送者，直接从它那接收数据，否则排队、阻塞。
- 异步接收：如缓冲区不为空，从缓冲区拷贝数据，否则排队、阻塞。

参数 `eq` 保存接收的数据，`selected` 是否有数据提供，`received` 是否接收成功。

### chan.c

```
void runtime·chanrecv(ChanType *t, Hchan* c, byte *ep, bool *selected, bool *received)
{
    // 如果是 nil channel, 阻塞。
    if(c == nil) {
        runtime·park(nil, nil, "chan receive (nil chan)");
        return; // not reached
    }

    // 和发送使用同一把锁。
    runtime·lock(c);

    // 异步版本。
    if(c->dataqsiz > 0) goto asynch;

    // 从 closed channel 拿不到任何东西，直接返回，不会 panic。
    if(c->closed) goto closed;

    // 从 sendq 找一个等着发送数据的家伙。
    sg = dequeue(&c->sendq);
    if(sg != nil) {
        // 一样是抛开 channel 私下交易。
        runtime·unlock(c);

        // 拷贝数据。
        if(ep != nil) c->elemalg->copy(c->elemsize, ep, sg->elem);
        gp = sg->g;
        gp->param = sg;

        // 唤醒发送者，让它解除阻塞。
        runtime·ready(gp);

        return;
    }
}
```

```

// 没找到发送者, 将自己打包放到 recvq, 等待。
mysg.elem = ep;
mysg.g = g;
mysg.selgen = NOSELGEN;
g->param = nil;
enqueue(&c->recvq, &mysg);
runtime·park(runtime·unlock, c, "chan receive");

return;

```

asynch:

```

// 如果缓冲区没有数据。
if(c->qcount <= 0) {
    if(c->closed) goto closed;

    // 将自己打包放到 recvq, 等待。
    mysg.g = g;
    mysg.elem = nil;
    mysg.selgen = NOSELGEN;
    enqueue(&c->recvq, &mysg);
    runtime·park(runtime·unlock, c, "chan receive");

    runtime·lock(c);

    // 被唤醒, 再次尝试从缓冲区接收数据。
    goto asynch;
}

// 从缓冲区拷贝数据。
if(ep != nil) c->elemalg->copy(c->elemsize, ep, chanbuf(c, c->recvx));

// 将缓冲区该位置清零。
c->elemalg->copy(c->elemsize, chanbuf(c, c->recvx), nil);
if(++c->recvx == c->dataqsiz) c->recvx = 0;
c->qcount--;

// 现在缓冲区有空位了, 如果 sendq 有人排队, 尝试唤醒, 没有也无所谓。
sg = dequeue(&c->sendq);
if(sg != nil) {
    gp = sg->g;
    runtime·unlock(c);
    runtime·ready(gp);
} else
    runtime·unlock(c);

return;

```

```
closed:
    runtime.unlock(c);
}
```

## 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 <code>bin</code> 、 <code>pkg</code> 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。
-blockprofilerate n	超出该参数设置时间的阻塞才被记录。单位：纳秒
-cpuprofile cpu.out	
-memprofile mem.out	内存分配。
-memprofilerate 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%      443 100.0% net.sotypeToNet
|      |
|      +- 当前函数及其所调用函数所占百分比
|      |
|      +- 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。