Go 学习笔记

好好学习，天天向上

前言

暂时不知道写啥……

下载地址：

本书不定期更新，可以到 github.com/qyuhen 下载最新版。

联系方式：

email: qyuhen@hotmail.com
QQ: 1620443

更新记录

2012-01-11 开始学习 Go。
2012-01-15 第一版 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。
2013-06-26 更新 godoc 规范。
2013-07-01 增加 UDP 示例。
2013-07-09 完善反射内容。
2013-07-20 修正部分内容。

目录

第一部分 Go 语言

暂时不知道写啥……

代码测试环境：

- Go 1.1
- OS X 10.8

第 1 章 基础

1.1 变量

使用 var 定义变量，自动初始化为零值 (Zero Value)。

- **bool**: false
- **integers**: 0
- **floats**: 0.0
- **string**: ""
- **pointers**, **functions**, **interfaces**, **slices**, **channels**, **maps**: nil

变量类型总是放在变量名后面，但可以省略 (根据初始化值进行类型推断)。Go 是强类型语言，类型推断只是一种简便的代码语法糖，不同于 Javascript、Python，我们不能修改变量的类型。

```
var a = 1234
var b string = "hello"
var c bool

func main() {
    println(a, b, c)
}
```

在函数内部，甚至可以省略 var 关键字，用 ":=" 这种更短的表达式完成变量类型推断和初始化。

```
a := 1
```

可以一次定义多个变量，并对其赋予初始值。

```
var x, y int                  // 类型相同的多个变量

var (                         // 类型不同的多个变量
    a int
    b bool
)

var c, d int = 1, 2           // 指定类型，多个变量类型相同。
var e, f = 123, "hello"       // 自动推断，多个变量类型按初始值推断。
g, h := 123, "hello"          // 自动推断，多个变量类型按初始值推断。
```

多变量赋值时，将先计算所有相关值，然后 left-to-right 进行变量赋值。

```
sa := []int{1, 2, 3}
i := 0
i, sa[i] = 1, 2               // sets i = 1, sa[0] = 2

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

```
sb[j], j = 2, 1              // sets sb[0] = 2, j = 1
println(sb[j], j)

sc := []int{1, 2, 3}
sc[0], sc[0] = 1, 2          // sets sc[0] = 1, then sc[0] = 2 (so sc[0] = 2 at end)
println(sc[0])
```

和 Python 一样，Go 也有个用来当垃圾桶的特殊变量 "_" (只能写，不能读)。

```
func test() (int, string) {
    return 123, "abc"
}

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

编译器会认为一个未被使用的变量和导入包是个错误。

```
package main

import (
    "fmt"             // Error: imported and not used: fmt
)

func main() {
    var a = 1         // Error: a declared and not used
}
```

1.2 基本类型

Go 对整数进行了更明确的规划，清晰明了。另外一个不同于 C 的地方，就是引用类型这种更加合理的方式代替指针。当然，指针还是有的。

类型	长度	说明
bool	1	true, false。不能把非零值当作 true。
byte	1	uint8
rune	4	int32。存储 Unicode Code Point。
int/uint		与平台有关，在 AMD64/X86-64 平台是 64 位整数。
int8/uint8	1	-128 ~ 127; 0 ~ 255
int16/uint16	2	-32768 ~ 32767; 0 ~ 65535
int32/uint32	4	-21亿 ~ 21亿, 0 ~ 42亿

类型	长度	说明
int64/uint64	8	
float32	4	精确到 7 个小数位
float64	8	精确到 15 个小数位
complex64	8	
complex128	16	
uintptr		足够保存指针的 32 位或 64 位整数
array		值类型 如: [2]int
struct		值类型
string		值类型
slice		引用类型 如: []int
map		引用类型
channel		引用类型
interface		接口类型
function		函数类型

可以用 八进制 (071)、十六进制 (0xFF) 或 科学计数法 (1e2) 对整数类型进行赋值。math 包中包含了 Min/Max 开头的各类型最小、最大值常量。

引用类型 (slice、map、channel) 的默认值为 nil，必须使用 make() 或初始化表达式分配内存，行为类似 C++/C# 的引用。

```
var s []int
var m map[string]int
var c chan int

println(s == nil, m == nil, c == nil)
```

输出:

```
true true true
```

不能隐式将非零值当作 bool true，或零值、nil 当作 bool false。

```
func main() {
    var a = 10

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

```
    if a > 0 { println("True") }
}
```

Go 总是按照值传递 (pass-by-value)，就算是引用和指针本身也是值拷贝。

1.3 类型转换

不支持隐式转换，必须进行显式类型转换 "<type>(expression)"。

```
func main() {
    a := 0x1234
    b := 1234.56
    c := 256

    fmt.Printf("%x\n", uint8(a))     // 截短长度
    fmt.Printf("%d\n", int(b))       // 截断小数
    fmt.Printf("%f\n", float64(c))
}
```

输出:

```
34
1234
256.000000
```

注意下面容易造成误解的例子，转换优先级比这些运算符更高，因此该用括号的时候别嫌麻烦。

```
*Point(p)             // same as *(Point(p))
(*Point)(p)           // p is converted to (*Point)
<-chan int(c)         // same as <-(chan int(c))
(<-chan int)(c)       // c is converted to (<-chan int)
```

1.4 常量

常量必须是编译期能确定的 Number (char/integer/float/complex)、String 和 bool 类型。可以在函数内定义局部常量。

```
const x uint32 = 123
const y = "Hello"
const a, b, c = "meat", 2, "veg"            // 同样支持一次定义多个常量。
const (
    z = false
    a = 123
)
```

不支持 C/C++ 0x12L、0x23LL 这样的类型后缀。

在定义常量组时，如不提供初始化值，则表示与上行常量的类型、值表达式完全相同 (是表达式相同而非结果相同)。

```
func main() {
    const (
        a = "abc"
        b
    )

    println(a, b)
}
```

输出:

```
abc abc
```

常量值还可以是 len()、cap()，unsafe.Sizeof() 常量计算表达式的值。

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

如果目标类型足以存储常量值，不会导致溢出 (overflow)，可不做显式转换。

```
var million int = 1e6        // float syntax OK here
var b byte = 'a'             // int to byte
```

1.4.1 枚举

可以用 iota 生成从 0 开始的自动增长枚举值。按行递增，可以省略后续行的 iota 关键字。

```
const (
    Sunday = iota     // 0
    Monday            // 1
    Tuesday           // 2
    Wednesday         // 3
    Thursday          // 4
    Friday            // 5
    Saturday          // 6
)
```

看一个常用的使用案例，后续行的类型和值表达式与 KB 相同，只有 iota 值发生变化。

```
type ByteSize int64
const (
    _ = iota                                // 忽略
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
)

func main() {
```

```
    println(GB, GB / 1024 / 1024 / 1024)
}
```

输出:

```
1073741824 1
```

可以在同一行使用多个 iota，它们各自增长。

```
const (
    A, B = iota, iota
    C, D
    E, F
)

func main() {
    println(C, D, E, F)
}
```

输出:

```
1 1 2 2
```

如果某行不想递增，可单独提供初始值。不过想要恢复递增，必须再次使用 iota。

```
func main() {
    const (
        a = iota      // 0
        b             // 1
        c             // 2
        d = "ha"      // 独立值
        e             // 没有使用 iota 恢复，所以和 d 值相同
        f = 100
        g             // 和 f 值相同
        h = iota      // 7，恢复 iota couter，继续按行递增。
        i             // 8
    )

    println(a, b, c, d, e, f, g, h, i)
}
```

输出:

```
0 1 2 ha ha 100 100 7 8
```

1.5 字符串

string 是不可变值类型，内部使用指针指向一个 UTF-8 byte 数组。变长编码方式，每个 Unicode 字符可能需要 1 ~ 4 个 byte 存储，不同于 Java/C#/Python 这类语言使用 UTF-16/UTF-32 固定宽度编码。

- 值类型，默认值为空字符串 ""。
- 可以用索引号访问某个字节，如 s[i]。
- 不能通过序号获取字节元素指针，&s[i] 是非法的。

- 不可变类型，无法直接修改字节数组内容。
- 字节数组不包括 NULL (\0) 结尾。

内存结构: src/pkg/runtime/runtime.h

```
146    struct String
147    {
148            byte*  str;
149            int32  len;
150    };
```

单引号字符常量表示一个 Unicode 字符（rune)，支持 \uFFFF、\UFFFFFFFF 和 \xFF 格式字符。

```
func main() {
    b := [10]byte{}
    b[1] = 'a'
    b[0] = '我'

    fmt.Println(b)
}
```

输出:

```
main.go: constant 25105 overflows uint8
make: *** [_go_.6] Error 1
```

使用符号 ` 定义原始是字符串 (Raw String)，不进行转义处理，支持跨行。

```
s := `ab\c`
```

使用 "+" 连接跨行字符串时需要注意，连接符号 "+" 必须在上一行尾部，否则会导致编译错误 (Error: invalid operation: + ideal string)。

```
s := "abc" +
     ", 123"
```

标准库 strings 包提供了 Join() 等常用的各种字符串操作函数。

```
import (
    "strings"
)

func main() {
    a, b, c := "a", "b", "c"
    s := strings.Join([]string{ a, b, c }, "")
    println(s)
}
```

还可以使用 bytes.Buffer 来完成 C# StringBuilder 的活。

```
func main() {
    sb := bytes.Buffer{}

    sb.WriteString("Hello ")
```

```
    sb.WriteString("World")
    sb.WriteString("!")

    fmt.Println(sb.String())
}
```

输出:

```
Hello World!
```

支持切片操作，返回子串 (string)，而非 slice。

```
func main() {
    s := "abcdefg"
    sub := s[1:3]

    fmt.Printf("%T = %v\n", sub, sub)
}
```

输出:

```
string = bc
```

要修改字符串，需要显式转换成 []byte 或 []rune。

```
string(0x1234)        // == "\u1234"
string([]bytes)       // bytes -> string
string([]runes)       // Unicode ints -> UTF-8 string
[]byte(string)        // UTF-8 string -> bytes
[]rune(string)        // UTF-8 string -> Unicode ints
```

转换成 []byte 后，可以修改字节内容，不过还需转换回来。

```
func main() {
    s := "abc"
    var c byte = s[1]               // 按索引号访问

    fmt.Printf("%c, %02x\n", c, c)

    bs := []byte(s)                 // 转换为 bytes，以便修改
    bs[1] = 'B'

    println(string(bs))
}
```

输出:

```
b, 62
aBc
```

看看下面的转换演示。

```
func main() {

    // UTF-8 编码格式存储的字符串
    s := "a:中国人"
    println("utf-8 string: ", s, len(s))
```

```
    /* --- string to bytes ------------------------------------ */

    // 转换成 bytes，以便于修改
    bs := []byte(s)
    bs[0] = 'A'

    for i := 0; i < len(bs); i++ {
        fmt.Printf("%02x ", bs[i])
    }

    // 将 bytes 转换为 string
    println(string(bs))

    /* --- string to unicode ---------------------------------- */

    // 转换成 Unicode
    u := []rune(s)
    println("unicode len =", len(u))

    // 显示 Unicode CodePoint
    for i := 0; i < len(u); i++ {
        fmt.Printf("%04x ", u[i])
    }

    // 按照 unicode 修改
    u[4] = '龙'

    // 将 unicode 转换为 string
    println(string(u))
}
```

输出:

```
utf-8 string:  a:中国人 11

41 3a e4 b8 ad e5 9b bd e4 ba ba A:中国人

unicode len = 5
0061 003a 4e2d 56fd 4eba a:中国龙
```

使用 for 遍历字符串时，需要注意 byte 和 rune 的区别。

```
func main() {
    s := "中国人"

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

    for i, c := range s {
        fmt.Printf("%d = %c\n", i, c)
    }
}
```

输出:

```
ä
```

```
¸
–
å

½
ä
º
º
0 = 中
3 = 国
6 = 人
```

字符串编码方案

字符串编码方案是个即简单又麻烦的话题，这主要涉及到字节 (byte) 和字符 (character) 两个不同的概念。

字节是计算机软件系统的最小存储单位，我们可以用 n 个字节表达一个概念，诸如 4 字节的 32 位整数或着其他什么对象之类的，任何对象最终都是通过字节组合来完成存储的。字符也是一种对象，它和具体的语言有关，比如在中文里，"我" 是一个字符，不能生生将其当作几个字节的拼接。

早期的计算机系统编码方案主要适应英文等字母语言体系，ASCII 编码方案的容量足以表达所有的字符，于是每个字符占用 1 个字节被固定下来，这也造成了一定程度上的误解，将字符和字节等同起来。要知道，这个世界上并非只有英文一种字符，中文等东亚语言体系的字符数量就无法用 ASCII 容纳，人们只好用 2 个或者更多的字节来表达一个字符，于是就有了 GB2312、BIG5 等等种类繁多且互不兼容的编码方案。

随着计算机技术在世界范围内的广泛使用，国际化需求越发重要，以往的字符编码方案已经不适应现代软件开发。于是国际标谁化组织 (ISO) 在参考很多现有方案的基础上统一制定了一种可以容纳世界上所有文字和符号的字符编码方案，这就是 Unicode 编码方案。Unicode 使得我们在一个系统中可以同时处理和显示不同国家的文字。

Unicode 本质上就是通过特定的算法将不同国家不同语言的字符都映射到数字上。比如中文字符 "我" 对应的数字是 25105 (\u6211)。Unicode 字符集 (UCS, Unicode Character Set) 有 UCS-2、UCS-4 两种标准。其中 UCS-2 最多可以表达 U+FFFF 个字符。而 UCS-4 则使用 4 字节编码，首位固定为 0，也就是说最多可以有 2^31 个字符，几乎容纳了全世界所有的字符。

Unicode 只是将字符和数字建立了映射关系，但对于计算机而言，要存储和操作任何数据，都得用字节来表示，这其中还涉及到不同计算机架构的大小端问题 (Big Endian, Little Endian)。于是有了几种将 Unicode 字符数字转换成字节的方法：Unicode 字符集转换格式 (UCS Transformation Format)，缩写 UTF。

目前常用的 UTF 格式有：

- **UTF-8**: 1 到 4 字节不等长方案，西文字符通常只用一个字节，而东亚字符 (CJK) 则需要三到四个字节。
- **UTF-16**: 用 2 字节无符号整数存储 Unicode 字符，与 UCS-2 对应，适合处理中文。
- **UTF-32**: 用 4 字节无符号整数存储 Unicode 字符。与 UCS-4 对应，多数时候有点浪费内存空间。

UTF-8 具有非常良好的平台适应性和交互能力，因此已经成为很多操作系统平台的默认存储编码方案。而 UTF-16 因为等长，浪费空间较少，拥有更好的处理性能，包括 .NET、JVM 等都使用 2 字节的 Unicode Char。

另外，按照大小端划分，UTF 又有 BE 和 LE 两种格式，比如 UTF-16BE、UTF-16LE 等。为了让系统能自动识别出 BE 和 LE，通常在头部添加 BOM 信息，"FE FF" 表示 BE，"FF FE" 表示 LE。后面还会有一两个字符用来表示 UTF-8 或 UTF-32。

1.6 运算符

Go运算符全部是从左到右结合 (总算从 C 泥沼里爬起来了)

优先级	运算符	说明
高	* / % << >> & &^	
·	+ - | ^	
·	== != < <= > >=	
·	<-	channel 运算符
·	&&	
低	||	

位运算符:

```
 6:    0110
11:    1011
----------------
&      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, bit clear。从 a 上清除所有 b 的标志位。
                           11 在 0、1、3 上设置了标志，检查 6 这三个二进制位，如果是 1 就改为 0 即可。
```

演示:

```
func main() {
    a := 0
    a = a | (1 << 3)        // 在 bit3 上设置标志位 (从 bit0 开始算)
    a = a | (1 << 6)        // 在 bit6 上设置标志位
```

```
    println(a)              // a = 72 = 0100 1000

    a = a &^ (1 << 6)       // 清除 bit6 上的标志位，a = 8 = 0000 0100
    println(a)
}
```

不支持运算符重载 (某些内建类型，如 string 做了 "+" 运算符重载)。

在 Go 中 ++、-- 是 statement 而非 expression，只能是后缀。另外，"*p++" 表示 "(*p)++" 而非 "*(p++)" 。

```
func main() {
    i := 0
    p := &i

    //a := i++        // syntax error: unexpected ++, expecting semicolon or newline or }
    i++
    a := i

    //b := (*p)++     // syntax error: unexpected ++, expecting semicolon or newline or }
    *p++
    b := *p

    println(a, b)
}
```

Go 所有的操作符和分隔符都在这了。

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

1.7 指针

Go 保留了指针，*T 表示 T 对应的指针类型。如果包含包名，则应该是 "*<package>.T"。代表指针类型的符号 "*" 总是和类型放在一起，而不是紧挨着变量名。同样支持指针的指针 (**T)。

可以确保下面两个变量都是指针类型，没有 C/C++ 里面那些弯弯绕的注意事项。

```
var a, b *int
```

- 操作符 "&" 取变量地址，用 "*" 透过指针变量间接访问目标对象。
- 默认值是 nil，没有 NULL 常量。
- 不支持指针运算，不支持 "->" 运算符，直接用 "." 选择符操作指针目标对象成员。

- 可以在 unsafe.Pointer 和任意类型指针间进行转换。
- 可以将 unsafe.Pointer 转换为 uintptr，然后做变相指针运算。uintptr 可以转换为整数。

```
type User struct {
    Id int
    Name string
}

func main() {
    i := 100
    var p *int = &i                 // 取地址
    //p++                           // invalid operation: p += 1 (mismatched types *int and int)
    println(*p)                     // 取值

    up := &User{ 1, "Jack" }
    up.Id = 100                     // 直接操作指针对象成员
    fmt.Println(up)

    u2 := *up                       // 拷贝对象
    u2.Name = "Tom"
    fmt.Println(up, u2)
}
```

输出:

```
100
&{100 Jack}
&{100 Jack} {100 Tom}
```

通过 unsafe.Pointer 在不同的指针类型间转换，做到类似 C void* 的用途。

```
const N int = int(unsafe.Sizeof(0))

func main() {
    x := 0x1234

    p := unsafe.Pointer(&x)         // *int -> Pointer
    p2 := (*[N]byte)(p)             // Pointer -> *[4]int，注意 slice 的内存布局和 array 是不同的。
                                    // 数组类型元素长度必须是常量。

    for i, m := 0, len(p2); i < m; i++ {
        fmt.Printf("%02X ", p2[i])
    }
}
```

输出:

```
34 12 00 00
```

将 unsafe.Pointer 转换成 uintptr，变相做指针运算。当然，还得用 Pointer 转换回来才能取值。

```
type User struct {
    Id int
    Name string
}
```

```
func main() {
    p := &User{ 1, "User1" }

    var np uintptr = uintptr(unsafe.Pointer(p)) + unsafe.Offsetof(p.Name)
    var name *string = (*string)(unsafe.Pointer(np))

    println(*name)
}
```

输出:

```
User1
```

对 GC 而言，uintptr 仅是存储地址的整数，并不会构成引用关联。也就是说 uintptr 无法保证目标对象不被回收。即便将 unitptr 转换成 Pointer，也会被当做不安全转换，依旧可能成为野指针。正常情况下的 Pointer 都经由合法类型指针转换而来，自然是安全的，可以确保目标对象不被回收。

可见 uintptr 适合局部环境下的操作，不应该作为外部对象传递。还可以用 ldflags "-u" 链接参数阻止编译 unsafe 代码。

1.8 保留字

Go 的保留字不多，整个语言设计相当简洁。

```
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
```

1.9 控制结构

Go 代码控制结构非常简洁，只有 for、switch、if。

1.9.1 IF

if 语句只需记住:

- 条件表达式没有括号;
- 支持一个初始化表达式 (可以是多变量初始化语句);
- 左大括号必须和条件语句在同一行。

```
func main() {
    a := 10

    if a > 0 {                // 左大括号必须写在这，否则被解释为 "if a > 0;" 导致编译出错。
        a += 100
    } else if a == 0 {        // 注意左大括号位置。
```

```
        a = 0
    } else {                    // 注意左大括号位置。
        a -= 100
    }

    println(a)
}
```

支持单行模式。

```
if a > 0 { a += 100 } else { a -= 100 }
```

初始化语句中定义的都是 block 级别的局部变量，不能在 block 外面使用 (else 分支内有效，但会隐藏 block 外的同名变量)。

```
if err := file.Chmod(0664); err != nil {
    log.Stderr(err)
    return err
}
```

很遗憾，Go 不提供三元操作符 "?:"，也没办法用 Python "and or"，因为不是 bool。

1.9.2 For

for 支持三种形式：

```
for init; condition; post {}
for condition {}
for {}
```

初始化和步进表达式可以是多个值。

```
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
        a[i], a[j] = a[j], a[i]
}
```

每次循环都会重新检查条件表达式，如果达表达式包含函数调用，则可能被执行多次。建议用初始化表达式一次性计算。

```
func main() {
    l := func(s string) int {
        println("get length")
        return len(s)
    }

    ss := "abcd"

    for i := 0; i < l(ss); i++ {
        println(ss[i])
    }
```

```
    println("---------")

    for i, m := 0, l(ss); i < m; i++ {
        println(ss[i])
    }
}
```

输出:

```
get length
97
get length
98
get length
99
get length
100
get length
---------
get length
97
98
99
100
```

1.9.3 Range

for ... range 用于完成迭代器 (iterator) 操作。可用于 string、array、slice、map、channel，每次迭代返回 index:value 、key:value、map key、channel value。同样可以用 "_" 抛弃某些值。迭代右表达式在循环开始前一次性计算，迭代变量每次循环时拷贝赋值。

```
func data() []int {
    fmt.Println("data...")
    return []int{1, 2, 3}
}

func main() {
    for i, x := range data() {
        fmt.Printf("%d: %d, %p\n", i, x, &x)
    }
}
```

输出:

```
data...
0: 1, 0x21015b018
1: 2, 0x21015b018
2: 3, 0x21015b018
```

range 实际上复制了目标对象，因此对 array 的修改不会影响原对象 (除非使用数组指针)。而其他引用类型，可以进行安全的删除操作。

```
func main() {
```

```
    array := [...]int{1, 2, 3}
    for i, x := range array {                           // 迭代变量总是从 array 复制品获取。
        fmt.Printf("array[%d] = %d\n", i, x)

        if i+1 < len(array) {
            array[i+1] += 1                             // 修改的是原对象，而非复制品。
        }
    }

    fmt.Println(array)

    slice := []int{1, 2, 3}
    for i, x := range slice {
        fmt.Printf("slice[%d] = %d\n", i, x)

        if i == 0 { slice[1] = 100 }                    // 引用类型，共享底层数组，修改有效。
        slice = append(slice, x+100)                    // 不会影响 slice 复制品。
    }

    fmt.Println(slice)

    maps := map[string]int{"x": 1, "y": 2, "z": 3}      // 引用类型的复制品，操作会影响原对象。
    for k, v := range maps {
        fmt.Printf("maps[%s] = %d\n", k, v)

        delete(maps, k)                                 // 安全删除元素项。
    }

    fmt.Println(maps)
}
```

输出:

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

slice[0] = 1
slice[1] = 100
slice[2] = 3
[1 100 3 101 200 103]

maps[x] = 1
maps[y] = 2
maps[z] = 3
map[]
```

map 使用随机存储，没有固定迭代次序，因此不建议在迭代时添加元素项，以造成不可预知结果。

```
func main() {
    maps := map[string]int{
        "a": 1, "b": 2, "c": 3, "d": 4, "e": 5,
        "f": 6, "g": 7, "h": 8, "i": 9, "j": 10,
    }
```

```
    for k := range maps {
        fmt.Printf("%s ", k)

        maps[k+k] = 100
        delete(maps, k)
    }

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

输出:

```
$ go run main.go                            // 多执行几次
b d f h j bb a c e g i
map[dd:100 ff:100 hh:100 jj:100 bbbb:100 aa:100 cc:100 ee:100 gg:100 ii:100]

$ go run main.go
a c e g i b d f h j aa cc ee gg ii bb
map[dd:100 ff:100 hh:100 jj:100 aaaa:100 cccc:100 eeee:100 gggg:100 iiii:100 bbbb:100]
```

另外，range string 返回 Unicode 字符，这和直接用 for 循环返回 UTF-8 字符是不同的。

1.9.4 Switch

switch case 表达式可以使用任意类型或表达式。break 不是必须的，自动终止。如希望继续下一 case 处理，则须显式执行 fallthrough。

```
func main() {
    switch a := 5; a {
        case 0, 1:                  // 逗号指定多个分支
            println("a")
        case 100:                   // 什么都不做。不是 fallthrough
        case 5:                     // 多行代码，无需使用 {}。
            println("b")
            fallthrough             // 进入后续 case 处理
        default:
            println("c")
    }
}
```

输出:

```
b
c
```

不指定 switch 条件表达式，或直接为 true 时，可用于替代 if...else if...else...。

```
func main() {
    a := 5
    switch {
        case a > 1: println("a")
        case a > 2: println("b")
        default: println("c")
```

```
    }
}
```

可以使用初始化表达式进一步简化上面的代码。

```
func main() {
    switch a := 5; {
       ...
    }
}
```

1.9.5 Goto, Break, Continue

支持函数内部 goto 跳转， 标签名区分大小写，未经使用的标签也会引发编译错误。
break 和 continue 都可以配合标签，在多级嵌套循环间跳出。
continue 仅对 for{} 循环有效，break 可用于 for{}、switch{}、select{}。

下面例子用 goto 代替 break 就成死循环了。通常建议往后 goto，避免死循环。

```
func main() {

L1:
    for {
        for i := 0; i < 10; i++ {
            if i > 2 {
                break L1
            } else {
                println("L1:", i)
            }
        }
    }

L2:
    for i := 0; i < 5; i++ {
        for {
            println("L2:", i)
            continue L2
        }
    }

    println("over")
}
```

输出:

```
L1: 0
L1: 1
L1: 2
L2: 0
L2: 1
L2: 2
L2: 3
L2: 4
```

```
over
```

1.10 自定义类型

依照类型标识符名称，可以将 Go 类型分为 "命名 (Named)" 和 "未命名 (UnNamed)" 两类。命名类型包括 bool、int、string 等，而未命名类型有 array、slice、map、channel、function 等。

由于 Go 是强类型语言，因此就算底层类型相同的命名类型之间亦不能做隐式转换。但具有相同声明的未命名类型则被视为同一种类型。

相等的未命名类型：

- 具有相同元素类型和长度的 array。
- 具有相同元素类型的 slice。
- 拥有相同的字段序列 (字段名、类型、标签相同，顺序相同) 的匿名 struct。
- 相同基类型的 pointer。
- 拥有完全相同签名 (包括参数和返回值相同，但不包括参数名) 的 function。
- 拥有相同方法集签名 (函数名、参数签名相同，函数排列次序无关) 的 interface。
- 拥有相同 key、value 类型的 map。
- 拥有相同数据类型和传送方向的 channel。

```
func main() {
    var a struct{x, y int}
    var b struct{x, y int}

    a.x = 1
    a.y = 2
    b = a

    fmt.Println(b)
}
```

可以用 type 定义新的类型，以便为其添加方法 (方法定义只能用于同一个包的类型)。

```
type MyInt int
type IntPointer *int

type Data struct {
    x, y int
}

type Tester interface {
    test(int)(string, int)
}

func main() {
    var x = 123
    var p IntPointer = &x;
```

```
    println(*p)

    var d = Data{ 1, 2 }
    println(d.x, d.y)

    p2 := &Data{ y:200 }
    println(p2.x, p2.y)
}
```

type 定义的新类型 (基于其他类型) 并不是一个别名，而是一个全新的类型。除了和底层类型拥有相同的数据结构外，它不会 "继承" 包括方法在内的任何信息，这与 struct 匿名字段嵌入完全不是一码事。如果底层类型是命名类型，那么必须进行显式转换，才能将新类型对象转换为底层类型。

```
type MyInt int
type MyIntSlice []int

func main() {
    var a MyInt = 10

    // var b int = a         // cannot use a (type MyInt) as type int in assignment
    var b int = int(a)

    // var c MyInt = b       // cannot use b (type int) as type MyInt in assignment
    var c MyInt = MyInt(b)

    println(a, b, c)
}
```

而当目标是未命名类型时，无需显式转换。

```
type MyIntSlice []int

func main() {
    var a MyIntSlice = []int{1, 2, 3}
    var b []int = a

    println(a, b)
}
```

1.11 初始化

初始化复合对象 (array、slice、map、struct) 时，必须添加类型标识，否则编译器会把大括号当作代码块处理，导致编译失败。

```
var d1 [2]int = [2]int{ 1 }          // 太繁琐，但不能省略右边的类型。
var d2 = [2]int{ 2 }                 // 在函数外部需要 var。
d3 := [2]int{ 3 }                    // 函数内部这样最简洁。
```

成员初始化表达式可单行或多行。以 "," 分隔多个元素，但多行的最后一个初始化值必须以 "," 或 "}" 结尾，否则将导致语法错误。

```
s := []byte{
    1,
    2,
    4}

m := map[string]int{
    "a": 1,
    "b": 2,                             // 这个逗号是必须的，或以 } 结尾。
}

u := struct {
    name string
    age  int
}{
    "user1",
    20,                                 // 这个逗号是必须的。
}

fmt.Println(s, m, u)
```

1.12 内置函数

内置函数不多，但都很实用。

- **close**: 关闭 channel。
- **len**: 获取 string、array、slice 长度，map key 数量，以及 buffer channel 可用数据数量。
- **cap**: 获取 array 长度，slice 容量，以及 buffer channel 的最大缓冲容量。
- **new**: 通常用于值类型，为指定类型分配初始化过的内存空间，返回指针。
- **make**: 仅用于 slice、map、channel 引用类型，除了初始化内存，还负责设置相关属性。
- **append**: 向 slice 追加 (在其尾部添加) 一个或多个元素。
- **copy**: 在不同 slice 间复制数据。
- **print/println**: 不支持 format，要格式化输出，须使用 fmt 包。
- **complex/real/imag**: 复数处理。
- **panic/recover**: 错误处理。

第 2 章 函数

Go 函数不支持 嵌套 (nested)、重载 (overload) 和 默认参数 (default parameter)，但支持：

- 无需声明原型;
- 不定长度变参;
- 多返回值;
- 命名返回值参数;
- 匿名函数;
- 闭包;

2.1 函数定义

函数使用 func 开头，左大括号不能另起一行 (Go 约定所有的大括号规则都是这样滴，^_^)。

```
// 接收多个参数，无返回值。
func test(a, b int, c string) {
    println(a, b, c)
}

// 单个返回值
func add(a, b int) int {
    return a + b
}

func main() {
    test(1, 2, "abc")        // 1 2 abc
    println(add(1, 2))       // 3
}
```

有返回值的函数必须以明确的终止语句结束执行，否则会引发编译错误。只是这种分析只是语法层面，而不是实际执行流程，所以有点生硬。

- 以 return、panic、goto 结束。
- 条件选择 if..else... 所有分支都能终止。
- 无条件 for 死循环是允许的。
- 没有 break 的 switch、select 得有 default，并确保所有分支都能终止。

2.2 函数类型

可以定义函数类型，函数也可以作为值 (默认值 nil) 被传递。

```
type callback func(s string)                    // 定义函数类型
```

```
func test(a, b int, sum func(int, int) int){        // 接收函数类型参数，也可以直接用 callback 类型。
    println(sum(a, b))
}

func main() {
    var cb callback = func(s string) {              // 函数类型变量。
        println(s)
    }

    cb("Hello, World!")

    test(1, 2, func(a, b int) int { return a + b }) // 这个写 Javascript 的人比较眼熟。
}
```

输出:

```
Hello, World!
3
```

2.3 多返回值、命名返回参数

函数可以象 Python 那样返回多个结果，只是没有 tuple 类型。对于不想要的返回值，可用特殊变量 "_" 忽略。如使用命名返回值参数，则 return 语句可以为空。不为空时，则依旧按顺序返回多个结果。

```
func swap(a, b int) (int, int) {     // 多个返回值
    return b, a
}

func change(a, b int) (x, y int) {   // 命名返回参数
    x = a + 100
    y = b + 100

    return                           // 也可以写成 return x, y
}

func main() {
    a, b := 1, 2
    a, b = swap(a, b)                // 2 1，可以用 _ 接收不想要的返回值
    println(a, b)

    c, d := change(1, 2)
    println(c, d)                    // 101 102
}
```

命名返回参数会被代码块中的同名变量隐藏 (shadowed)，此时就须显式 return 返回结果。

```
func test(a int) (x int) {
    if x := 10; a > 0 {              // 此处的 x 是新定义的同名变量。
        return                       // block 变量 x 隐藏了命名返回值 x。
    }
```

```
    return
}

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

输出:

```
./main.go: x is shadowed during return
```

改用显式 return 返回就没问题了。

```
func test(a int) (x int) {
    if x := 10; a > 0 {
        return x + 1                // 这样就没问题了。
    }

    return                          // 这个不受 if block 影响。
}
```

另外要注意，显式 return 会先修改命名返回值，然后再执行 ret 指令。

```
func test() (x int) {
    defer func() {                  // 执行顺序: x = 100 => call_defer_func => ret
        println(x)
    }()

    return 100
}

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

输出:

```
100
```

多返回值可以直接传递给其他函数做参数。

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

func test(i int, s string) {        // 也可以用变参
    fmt.Println(i, s)
}

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

输出:

```
1 abc
```

2.4 变参

变参本质上就是一个 slice，且必须是最后一个形参。将 slice 传递给变参函数时，注意用 "..." 展开，否则就当作单个参数处理了。和 Python *args 方式相同。

```
func sum(s string, args ...int) {       // 注意语法格式
    var x int
    for _, n := range args {
        x += n
    }
    println(s, x)
}

func main() {
    sum("1 + 2 + 3 =", 1, 2, 3)

    x := []int{0, 1, 2, 3, 4}
    sum("0 + 1 + 2 =", x[:3]...)        // 注意展开
}
```

2.5 匿名函数、闭包

Go 的匿名函数类似 Javascript，远比 Python lambda 好用。对闭包支持良好。

```
/* 闭包支持 */
func closures(x int) (func(int) int) {
    // 返回匿名函数
    return func (y int) int {
        return x + y
    }
}

func main() {
    f := closures(10)        // 匿名函数，引用了 closures.x
    println(f(1))            // 11
    println(f(2))            // 12
}
```

事实上，每次都会创建一个新的匿名函数对象。另外，从输出结果可以观察到闭包的 "延迟" 现象。

```
func main() {
    var fs []func()(int)

    for i := 0; i < 3; i++ {
        fs = append(fs, func() int {
            return i
        })
    }

    for _, f := range fs {
        fmt.Printf("%p = %v\n", f, f())
```

```
    }
}
```

输出：

```
0x42141000 = 3
0x42141040 = 3
0x42141080 = 3
```

闭包指向同一个变量，而不是复制。

```
func test(x int) (f1 func(), f2 func()) {
    fmt.Printf("test: %p = %v\n", &x, x)

    f1 = func() {
        x += 100
        fmt.Printf("f1: %p = %v\n", &x, x)
    }

    f2 = func() {
        fmt.Printf("f2: %p = %v\n", &x, x)
    }

    return
}

func main() {
    f1, f2 := test(100)
    f1()
    f2()
}
```

输出：

```
test: 0x21015a018 = 100
f1  : 0x21015a018 = 200
f2  : 0x21015a018 = 200
```

2.6 Defer

defer 的作用就是向函数注册退出调用 (也就是说必须是函数或方法)，可以看作是：

```
func (...) {
    try {
    }
    finally {
        defer_func2()
        defer_func1()
    }
    return
}
```

多个 defer 定义，按 FILO 的次序执行。就算函数发生严重错误，退出函数依然会被执行。定义时必须提供执行所需参数，包括空参数列表。

```
func test(a, b int) int {
    defer println("defer1:", a, "/", b)

    defer func() {
        println("defer2:", a, b)
    }()                             // 匿名函数，参数列表为空。

    return a / b
}

func main() {
    a := test(10, 2)
    b := test(10, 0)                // 通过输出会发现，就算发生严重 runtime error，defer 依然被执行。

    print(a, b)
}
```

输出:

```
defer2: 10 2
defer1: 10 / 2
defer2: 10 0
defer1: 10 / 0
panic: runtime error: integer divide by zero

[signal 0x8 code=0x7 addr=0x20c9 pc=0x20c9]
```

常用来做资源清理、关闭文件、解锁、记录执行时间等等操作。

```
func WriteData(file *File, mu *Mutex, data ...string) {
    mu.Lock()
    defer mu.Unlock()

    //  write data ...
}
```

既然是退出函数，自然可用于 "篡改" 输出结果 (必须命名返回值，闭包作用)。

```
func test(a, b int) (c int) {
    defer func() {
        c += 100
    }()

    c = a + b
    return
}

func main() {
    x := test(10, 0)
    print(x)
}
```

输出:

```
110
```

用 defer 定义退出调用时，所提供的参数会发生值拷贝。尽管这些函数在退出时才执行，但所使用参数是定义时的拷贝。可以考虑用指针或者闭包代替，以便获得 "最新" 的参数值。

```
func main() {
    x := 10

    defer func(a int) {
        println("a = ", a)
    }(x)

    defer println("print =", x)

    x += 100
    println(x)
}
```

输出:

```
110
print = 10
a =  10
```

闭包引用对象，仅在执行的时候才获取对象。

```
func main() {
    var fs = [4]func(){}

    for i := 0; i < 4; i++ {
        defer println("defer i =", i)                  // defer: 直接获取当前 i 的值
        defer func() { println("defer_closure", i)}() // defer_closure: 在 defer 函数中使用闭包
        fs[i] = func(){ println("closure i = ", i) }  // closure: 仅持有 i 的引用，在函数执行时再获取
    }

    for _, f := range fs { f() }                       // 执行闭包函数
}
```

输出:

```
closure i =  4                                          // 为便于对比，输出结果顺序被整理过。
closure i =  4
closure i =  4
closure i =  4
defer_closure 4
defer_closure 4
defer_closure 4
defer_closure 4
defer i = 3
defer i = 2
defer i = 1
defer i = 0
```

退出调用在 ret 指令之前才发生。也就是说，如果 return 修改命名返回值，那么闭包引用的结果就是修改后的值。

2.7 Panic、Recover

Go 没有 try ... catch ... finally 这种结构化异常处理，而是用 panic 代替 throw/raise 引发错误，然后在 defer 中用 recover 函数捕获错误。

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

如果不使用 recover 捕获，则 panic 沿着调用堆栈向外层传递。recover 仅在 defer 函数中使用才会终止错误，此时函数执行流程已经中断，无法像 catch 那样恢复到后续位置继续执行。

```
func main() {
    // 错误: 无法捕获，recover 只能捕获调用堆栈内层的错误。
    // defer func() {
    //     func() {
    //         fmt.Println(recover())
    //     }()
    // }()

    // 错误: 注意 recover 只能在 defer 函数内部才有效。
    // defer recover()
    // defer fmt.Println(recover())

    // 正确方式
    defer func() {
        if err := recover(); err != nil {
            log.Fatalln(err)
        }
    }()

    panic("abc")
}
```

对局部代码进行错误保护，可以使用匿名函数。

```
func main() {
    fmt.Println("Hello, World!")

    func() {                                // 利用匿名函数形成 function scope。
        defer func() {
            fmt.Println(recover())
        }()

        panic(errors.New("Error!"))
    }()                                     // 直接执行匿名函数。
}
```

虽然 panic 可以抛出任意类型 (interface{}) 的错误对象，但规范中总是使用 error 接口类型。可使用 fmt.Errorf(format, ...)、errors.New(text) 便捷创建错误对象。

除了使用 panic 引发错误外，go 还使用 error 返回错误信息。那么如何区别使用呢?

> The convention in the Go libraries is that even when a package uses panic internally, its external API still presents explicit error return values.

2.8 Call Stack

runtime 包提供了 Caller、Callers 两个函数用来获取调用堆栈信息。

```
func test() {
    fmt.Println(runtime.Caller(1))
}

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

输出:

```
8550 /Users/yuhen/.../main.go 13 true
```

Callers 可以获取完整的调用堆栈列表。

```
func test() {
    ps := make([]uintptr, 10)
    count := runtime.Callers(0, ps)

    for i := 0; i < count; i++ {
        f := runtime.FuncForPC(ps[i])
        fmt.Printf("%d, %s\n", i, f.Name())
    }
}

func main() {
    a := func() {
        test()
    }

    a()
}
```

输出:

```
0, runtime.Callers
1, main.test
2, main._func_001
3, main.main
4, runtime.main
5, runtime.goexit
```

输出调用堆栈信息，可以直接调用 runtime/debug.PrintStack() 函数。

利用 runtime.FuncForPC() 可通过函数指针获取符号信息，比如名称、源文件、行号等。(如果符号表被删除，将导致执行出错)

```
package main

import (
    "fmt"
    "reflect"
    "runtime"
)

func test() {}

func main() {
    p := reflect.ValueOf(test).Pointer()
    f := runtime.FuncForPC(p)

    fmt.Println(f)
    fmt.Printf("%p, %#x\n", test, f.Entry())
}
```

输出:

```
&{main.test  main.go [147 2] 8192 8193 9 8 0 0}
0x2000, 0x2000
```

第 3 章 Array、Slices 和 Maps

3.1 Array

数组定义方式 "[n]<type>" 有点古怪。长度下标 n 必须是编译期正整数常量 (或常量表达式)。 长度是类型的组成部分，也就是说 "[10]int" 和 "[20]int" 是完全不同的两种数组类型。

```
var a [4]int;                          // 所有元素自动被初始化为 0
a[1] = 100;

for i := 0; i < len(a); i++ {
    println(a[i])
}
```

可以用复合语句直接初始化。

```
a := [10]int{ 1, 2, 3, 4 }             // 未提供初始化值的元素为默认值 0
b := [...]int{ 1, 2 }                  // 由初始化列表决定数组长度，不能省略 "..."，否则就成 slice 了。
c := [10]int{ 2:1, 5:100 }             // 按序号初始化元素
```

数组指针类型 *[n]T，指针数组类型 [n]*T。

```
x, y := 1, 2

var p1 *[2]int = &[2]int{x, y}
var p2 [2]*int = [2]*int{&x, &y}

fmt.Printf("%#v\n", p1)
fmt.Printf("%#v\n", p2)
```

输出:

```
&[2]int{1, 2}
[2]*int{(*int)(0x4212f100), (*int)(0x4212f108)}
```

支持 "=="、"!=" 相等操作符 (因为 Go 对象所使用的内存都被初始化为 0，按字节比较是可以的)，但不支持 ">"、"<" 等比较操作符。

```
println([1]string{"a"} == [1]string{"a"})
```

数组是值类型，也就是说会拷贝整个数组内存进行值传递。可用 slice 或指针代替。

```
func test(x *[4]int) {
    for i := 0; i < len(x); i++ {
        println(x[i])                          // 用指针访问数组语法并没有什么不同，比 C 更直观。
    }

    x[3] = 300
}

func main() {
```

```
    x := &[4]int{ 2:100, 1:200 }
    test(x)

    println(x[3])
}
```

可以用 new() 创建数组，返回数组指针。

```
func test(a *[10]int) {
    a[2] = 100                      // 用指针直接操作木有压力。
}

func main() {
    var a = new([10]int)            // 返回指针。
    test(a)

    fmt.Println(a, len(a))
}
```

输出:

```
&[0 0 100 0 0 0 0 0 0 0] 10
```

多维数组和 C 类似，一种数组的数组。

```
func main() {
    var a = [3][2]int{ [...]int{1, 2}, [...]int{3, 4} }
    var b = [3][2]int{ {1, 2}, {3, 4} }

    c := [...][2]int{ {1, 2}, {3, 4}, {5, 6} }             // 第二个维度不能用 "..." 。
    c[1][1] = 100

    fmt.Println(a, "\n", b, "\n", c, len(c), len(c[0]))
}
```

输出:

```
[[1 2] [3 4] [0 0]]
[[1 2] [3 4] [0 0]]
[[1 2] [3 100] [5 6]] 3 2
```

由于元素类型相同，因此可以使用复合字面值初始化数组成员。

```
type User struct {
    Id int
    Name string
}

func main() {
    a := [...]User {
        { 0, "User0" },
        { 1, "User1" },
    }

    b := [...]*User {
        { 0, "User0" },
```

```
        { 1, "User1" },
    }

    fmt.Println(a)
    fmt.Println(b, b[1])
}
```

输出:

```
[{0 User0} {1 User1}]
[0x42122340 0x42122320] &{1 User1}
```

3.2 Slices

作为变长数组的替代方案，slice 具备更灵活的特征。通过相关属性关联到底层数组的局部或全部。

src/pkg/runtime/runtime.h

```
172    struct  Slice
173    {                          // must not move anything
174            byte*  array;      // actual data
175            uint32 len;        // number of elements
176            uint32 cap;        // allocated number of elements
177    };
```

http://blog.golang.org/2011/01/go-slices-usage-and-internals.html

slice 是引用类型，默认值为 nil。可以用内置函数 len() 获取长度，cap() 获取容量。使用索引访问时，不能超出 0 到 len - 1 范围。和值类型 array 不同，引用类型 slice 赋值不会复制底层数组。

用 slice 长时间引用 "超大" 的底层数组，会导致严重的内存浪费。可考虑新建一个小的 slice 对象，然后将所需的数据 copy 过去。

在使用方法上和 Python 类似，可惜不支持负数定义逆向索引。

```
func main() {
    x := [...]int{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

    var s1 []int = x[1:3]     // 内容包括 x[1], x[2]
    fmt.Println(s1)

    s2 := x[4:]               // x[4] ~ x[len - 1]
    fmt.Println(s2)

    s3 := x[:6]               // x[0] ~ x[5]
    fmt.Println(s3)

    s4 := x[:]                // x[0] ~ x[len - 1]
    fmt.Println(s4)
}
```

输出:

```
[1 2]
```

```
[4 5 6 7 8 9]
[0 1 2 3 4 5]
[0 1 2 3 4 5 6 7 8 9]
```

对 slice 的修改就是对底层数组的修改。

```
func main() {
    x := [...]int{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    s := x[1:3]

    s[0] = 100                // 对 slice 的修改实际上是对原数组的修改，注意序号的差异。
    fmt.Println(x)

    x[2] = 200                // 对数组修改会影响 slice。
    fmt.Println(s)
}
```

输出:

```
[0 100 2 3 4 5 6 7 8 9]
[100 200]
```

可以直接创建一个 slice 对象 (内部自动创建底层数组)，而不是从数组开始。

```
func test(s []int) {
    fmt.Println(s)
    s[1] = 100                // 内部总是指向同一个数组。
}

func main() {
    s := []int{ 0, 1, 2 }
    test(s)                   // 引用类型，不用担心复制底层数组。

    fmt.Println(s)
}
```

输出:

```
[0 1 2]
[0 100 2]
```

不能使用 new()，而应该是 make([]T, len, cap)。因为除了分配内存，还需要设置相关的属性。如果忽略 cap 参数，则 cap = len。

```
func main() {
    s1 := make([]int, 10)                   // 相当于 [10]int{...}[:]
    s1[1] = 100
    fmt.Println(s1, len(s1), cap(s1))

    s2 := make([]int, 5, 10)
    s2[4] = 200
    fmt.Println(s2, len(s2), cap(s2))
}
```

输出:

```
[0 100 0 0 0 0 0 0 0 0] 10 10
```

```
[0 0 0 0 200] 5 10
```

3.2.1 reslice

cap 是 slice 非常重要的属性，它表示了 slice 可以容纳的最大元素数量。在初始创建 slice 对象时 **cap = array_length - slice_start_index**，也就是 slice 在数组上的开始位置到数组结尾的元素数量。

在 cap 允许的范围内我们可以 reslice，以便操作后续的数组元素。超出 cap 限制不会导致底层数组重新分配，只会引发 "slice bounds out of range" 错误。

```
func main() {
    a := [...]int{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

    s1 := a[5:7]              // [5 6], len = 2, cap = 5

    s1 = s1[0:4]              // [5, 6, 7, 8], len = 4, cap = 5
                              // 注意 reslice 不能超过 cap 的限制。
                              // 是在 slice 上重新切分，不是 Array，因此序号是以 slice 为准。
}
```

3.2.2 append

可以用 append() 向 slice 尾部添加新元素，这些元素保存到底层数组。append 并不会影响原 slice 的属性，它返回变更后新的 slice 对象。如果超出 cap 限制，则会重新分配底层数组。

```
func main() {
    s1 := make([]int, 3, 6)

    // 添加数据，未超出底层数组容量限制。
    s2 := append(s1, 1, 2, 3)

    // append 不会调整原 slice 属性。
    // s1 == [0 0 0] len:3 cap:6
    fmt.Println(s1, len(s1), cap(s1))

    // 注意 append 是追加，也就是说在 s1 尾部添加。
    // s2 == [0 0 0 1 2 3] len:6 cap:6
    fmt.Println(s2, len(s2), cap(s2))

    // 追加的数据未超出底层数组容量限制。
    // 通过调整 s1，我们可以看到依然使用的是原数组。
    // s1 == [0 0 0 1 2 3] len:6 cap:6
    s1 = s1[:cap(s1)]
    fmt.Println(s1, len(s1), cap(s1))

    // 再次追加数据 (使用了变参)。
    // 原底层数组已经无法容纳新的数据，将重新分配内存，并拷贝原有数据。
    // 我们通过修改数组第一个元素来判断是否指向原数组。
```

```
    // s3 == [100 0 0 1 2 3 4 5 6] len:9 len:12
    s3 := append(s2, []int{ 4, 5, 6 }...)
    s3[0] = 100
    fmt.Println(s3, len(s3), cap(s3))

    // 而原 slice 对象依然指向旧的底层数组对象，所以彻底和 s3 分道扬镳。
    // s1 == [0 0 0 1 2 3] len:6 cap:6
    // s2 == [0 0 0 1 2 3] len:6 cap:6
    fmt.Println(s1, len(s1), cap(s1))
    fmt.Println(s2, len(s2), cap(s2))
}
```

输出:

```
[0 0 0] 3 6
[0 0 0 1 2 3] 6 6
[0 0 0 1 2 3] 6 6
[100 0 0 1 2 3 4 5 6] 9 12
[0 0 0 1 2 3] 6 6
[0 0 0 1 2 3] 6 6
```

因为 append 每次会创建新的 slice 对象，因此要优先考虑用序号操作。

```
func test1(n int) []int {
    datas := make([]int, 0, n)

    for i := 0; i < n; i++ {
        datas = append(datas, i)
    }

    return datas
}

func test2(n int) []int {
    datas := make([]int, n)

    for i := 0; i < n; i++ {
        datas[i] = i
    }

    return datas
}

func main() {
    // datas := test1(10000)
    datas := test2(10000)
    println(len(datas))
}
```

两种写法的性能差异很明显。虽然底层数组相同，但 test2 预先就 "填充" 了全部元素，演变为普通数组操作。test1 append 每次循环都会创建新的 slice 对象，累加起来的消耗并不小。

3.2.3 copy

函数 copy 用于在 slice 间复制数据，可以是指向同一底层数组的两个 slice。复制元素数量受限于 src 和 dst 的 len 值 (两者的最小值)。在同一底层数组的不同 slice 间拷贝时，元素位置可以重叠。

```
func main() {
    s1 := []int{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    s2 := make([]int, 3, 20)

    var n int

    n = copy(s2, s1)                          // n = 3。不同数组上拷贝。s2.len == 3，只能拷 3 个元素。
    fmt.Println(n, s2, len(s2), cap(s2))      // [0 1 2], len:3, cap:20

    s3 := s1[4:6]                             // s3 == [4 5]。s3 和 s1 指向同一个底层数组。
    n = copy(s3, s1[1:5])                     // n = 2。同一数组上拷贝，且存在重叠区域。
    fmt.Println(n, s1, s3)                    // [0 1 2 3 1 2 6 7 8 9] [1 2]
}
```

输出:

```
3 [0 1 2] 3 20
2 [0 1 2 3 1 2 6 7 8 9] [1 2]
```

3.3 Maps

引用类型，类似 Python dict，不保证 Key/Value 存放顺序。Key 必须是支持比较运算符 (==、!=) 的类型。如 number、string、pointer、array、struct、interface (接口实现类型必须支持比较运算符)，不能是 function、map、slice。

map 查找操作比线性搜索快很多，但比起用序号访问 array、slice，大约慢 100x 左右。绝大多数时候，其操作性能要略好于 Python Dict、C++ Map。

```
func test(d map[string]int) {
    d["x"] = 100
}

func main() {
    var d = map[string]int{ "a":1, "b":2 };
    d2 := map[int]string{ 1:"a", 2:"b" };

    test(d)
    fmt.Println(d, d2)

    d3 := make(map[string]string)
    d3["name"] = "Jack"
    fmt.Println(d3, len(d3))
}
```

输出:

```
map[a:1 b:2 x:100] map[1:a 2:b]
map[name:Jack] 1
```

使用 array/struct key 的例子。

```
type User struct {
    Name string
}

func main() {
    a := [2]int{ 0, 1}
    b := [2]int{ 0, 1}
    d := map[[2]int]string { a: "ssss" }
    fmt.Println(d, d[b])

    u := User{ "User1" }
    u2 := u
    d2 := map[User]string { u: "xxxx" }
    fmt.Println(d2, d2[u2])
}
```

输出:

```
map[[0 1]:ssss] ssss
map[{User1}:xxxx] xxxx
```

value 的类型就很自由了，完全可以用匿名结构或者空接口。

```
type User struct {
    Name string
}

func main() {
    i := 100
    d := map[*int]struct{ x, y float64 } { &i: { 1.0, 2.0 } }
    fmt.Println(d, d[&i], d[&i].y)

    d2 := map[string]interface{} { "a": 1, "b": User{ "user1" } }
    fmt.Println(d2, d2["b"].(User).Name)
}
```

输出:

```
map[0x42132018:{1 2}] {1 2} 2
map[a:1 b:{user1}] user1
```

使用 make() 创建 map 时，提供一个合理的初始容量有助于减少后续新增操作的内存分配次数。在需要时，map 会自动扩张容量。

常用的判断和删除操作:

```
func main() {
    var d = map[string]int{ "a":1, "b":2 };

    v, ok := d["b"]                  // key b 存在，v = ["b"], ok = true
    fmt.Println(v, ok)

    v = d["c"]                       // key c 不存在，v = 0 (default)
```

```
    fmt.Println(v)                    // 要判断 key 是否存在，建议用 ok idiom 模式。

    d["c"] = 3                        // 添加或修改
    fmt.Println(d)

    delete(d, "c")                    // 删除。删除不存在的 key，不会引发错误。
    fmt.Println(d)
}
```

输出:

```
2 true
0 false
map[a:1 c:3 b:2]
map[a:1 b:2]
```

迭代器用法:

The Go 1.1 hashmap iteration starts at a random bucket. However, it stores up to 8 key/value pairs in a bucket.

And within a bucket, hashmap iteration always returns the values in the same order.

```
func main() {
    d := map[string]int{ "a":1, "b":2 };

    for k, v := range d {                    // 获取 key, value
        println(k, "=", v)
    }

    for k := range d {                       // 仅获取 key
        println(k, "=", d[k])
    }
}
```

通过 map[key] 返回的只是一个 "临时值拷贝"，修改其自身状态没有任何意义，只能重新 value 赋值或改用指针修改所引用的内存。

```
type User struct {
    Id   int
    Name string
}

func main() {
    users := map[string]User{
        "a": User{1, "user1"},
    }

    fmt.Println(users)

    // Error: cannot assign to users["a"].Name
    //users["a"].Name = "Jack"

    // 重新 value 赋值
    u := users["a"]
    u.Name = "Jack"
```

```
    users["a"] = u

    fmt.Println(users["a"])

    // 改用指针类型
    users2 := map[string]*User{
        "a2": &User{2, "user2"},
    }

    users2["a2"].Name = "Tom"

    fmt.Println(users2["a2"])
}
```

输出:

```
map[a:{1 user1}]
{1 Jack}
&{2 Tom}
```

可以在 for range 迭代时安全删除和插入新的字典项。(不推荐迭代插入操作，详情参考 1.9.3)

```
func main() {
    d := map[string]int { "b":2, "c":3, "e":5 }

    for k, v := range d {
        println(k, v)
        if k == "b" { delete(d, k) }
        if k == "c" { d["a"] = 1 }
    }

    fmt.Println(d)
}
```

输出:

```
c 3
b 2
e 5
map[a:1 c:3 e:5]
```

第 4 章 Structs

看上去和 C struct 非常相似，Go 没有 class，用 struct 来实现面向对象编程模型。

4.1 定义

定义结构很简单，但只有大写字母开头的成员才能在包外被访问 (public，Go 以开头大写字母作为导出符号的标志)。

```
type User struct {
    Id int
    Name string
}

func main() {
    user1 := User{ 1, "Tom" }
    user2 := User{ Name:"Jack" }

    println(user1.Id, user1.Name)
    println(user2.Id, user2.Name)
}
```

输出:

```
1 Tom
0 Jack
```

可以用 "_" 定义 "补位" 字段，支持指向自身的指针类型成员。

```
type Node struct {
    _     int
    Value string
    Next  *Node
}

func main() {
    a := &Node{Value: "a"}
    b := &Node{Value: "b", Next: a}

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

输出:

```
&main.Node{_:0, Value:"a", Next:(*main.Node)(nil)}
&main.Node{_:0, Value:"b", Next:(*main.Node)(0x42120420)}
```

支持匿名结构，可用作成员或直接定义变量。

```
type Person struct {
    name    string
    age     int
```

```
    contact struct {                        // 匿名结构成员
        phone    string
        address  string
        postcode string
    }
}

func main() {
    var d = struct {                        // 直接定义匿名结构变量
        name  string
        age   int
        title string
    }{"user1", 10, "cto"}

    d2 := Person{
        name: "user1",
        age:  10,
    }

    d2.contact.phone = "13500000000"        // 因为没有类型名称，只能如此初始化。或者重复定义那一大坨代码。
    d2.contact.address = "beijing"
    d2.contact.postcode = "100000"

    fmt.Println(d, d2)
}
```

匿名结构还可直接用于 map。

```
func test() {
    d := map[string]struct {
        x string
        y int
    }{
        "a": {"a10", 10},
        "b": {"b20", 20},
    }

    fmt.Println(d)
}
```

4.2 初始化

可以按顺序初始化全部字段，或者用字段名初始化部分字段，没有被初始化的字段为默认值 0。

```
u1 := User{ 1, "Tom" }
u2 := User{ Id:2, Name:"Jack" }
p1 := &User{ 1, "Tom" }                     // 习惯性用法
p2 := &User{}                               // 习惯性用法，相当于 new(User)。
```

允许直接用指针读写成员字段 (Go 没有 "->" 指针运算符)。

```
type User struct {
    Id int
    Name string
}

func main() {
    u := &User{ 100, "Tom" }
    println(u.Id, u.Name)

    u.Id = 200
    println(u.Id, u.Name)
}
```

输出:

```
100 Tom
200 Tom
```

相同类型的结构可以直接拷贝赋值。

```
type User struct {
    Id int
    Name string
}

func main() {
    u := User{ 100, "Tom" }
    println(u.Id, u.Name)

    var u2 *User = new(User)
    *u2 = u
    println(u2.Id, u2.Name)
}
```

支持 "=="、"!=" 操作符，不支持 ">"、"<" 等比较操作符。

```
println(User{"a"} == User{"a"})
```

4.3 匿名字段

将结构体或指针嵌入到另一个结构体，但不提供字段名，这种方式称之为匿名字段。

这看上去很酷， 和大多数 OOP 语言实现继承方式有些类似，但其本质上就是隐式定义了一个以类型名为名称的字段。由于是内容嵌入，所以在内存结构上它们是一个整体。

Go 对 "." 成员运算符做了特殊处理，使得我们可以直接访问和设置匿名字段成员。但在初始化成员时，我们依然要把匿名字段当作一个正常的字段来赋值。

```
type User struct {
    Id int
    Name string
}
```

```
type Manager struct {
    User
    Group string
}

func main() {
    m := Manager{ User{ 1, "Jack"}, "IT" }
    println(m.Id, m.Name, m.Group)

    m.Name = "Tom"
    println(m.Id, m.Name, m.Group)
}
```

输出:

```
1 Jack IT
1 Tom IT
```

就算被嵌套的匿名字段也包含匿名字段，依然可以用 "." 操作符直接访问任意层级的成员。编译器总是从外向里进行查找，直到完成任务或出错。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    User
    Group string
}

type CXO struct {
    Manager
    Title string
}

func main() {
    ceo := CXO{ Manager{ User{ 100, "猪头三" }, "Board of directors" }, "CEO" }
    fmt.Println(ceo, ceo.Id, ceo.Name, ceo.Group, ceo.Title)
}
```

输出:

```
{{{100 猪头三} Board of directors} CEO} 100 猪头三 Board of directors CEO
```

如果嵌入的结构来自其他包，则需要加上包名。可以匿名嵌入任意类型，包括 int、string 等。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    User
    Group string
```

```
    int
    string
}

func main() {
    m := Manager{ User:User{ 1, "Jack" }, Group:"IT", int:100, string:"Hello!" }
    println(m.Id, m.Name, m.Group, m.int, m.string)

    m.Name = "Tom"
    m.int = 200
    m.string = "World!"
    println(m.Id, m.Name, m.Group, m.int, m.string)
}
```

输出:

```
1 Jack IT 100 Hello!
1 Tom IT 200 World!
```

匿名字段可能导致在不同层级上有多个同名字段，如此会导致隐藏行为发生：

- 外部字段会隐藏匿名字段同名成员。
- 如果两个匿名字段存在同名成员，将可能导致字段访问错误。

这只是语法上的隐藏，依然可以通过类型名访问被隐藏的匿名成员。

```
type D1 struct {
    x int
}

type Data struct {
    D1
    x int
}

func main() {
    d := Data { D1{ 10 }, 20 }
    println(d.x, d.D1.x)

    d.x = 200
    d.D1.x = 100
    println(d.x, d.D1.x)
}
```

输出:

```
20 10
200 100
```

如果多个匿名字段在同一层次同名，将导致编译器无法确定目标而出错。

```
type D1 struct {
    x int
}

type D2 struct {
```

```
    x int
}

type Data struct {
    D1
    D2
}

func main() {
    d := Data { D1{ 10 }, D2{ 20 } }
    println(d.x)
}
```

输出:

```
main.go:24: ambiguous DOT reference Data.x
make: *** [_go_.6] Error 1
```

改用匿名字段类型前缀访问，就不会引发编译器错误了。

```
func main() {
    d := Data { D1{ 10 }, D2{ 20 } }
    println(d.D1.x, d.D2.x)
}
```

输出:

```
10 20
```

基于隐藏规则，如果外部类型 Data 也有一个 x int 字段，那么访问 d.x 肯定是没事的。

```
type Data struct {
    D1
    D2
    x int
}

func main() {
    d := Data { D1{ 10 }, D2{ 20 }, 30 }
    println(d.x, d.D1.x, d.D2.x)
}
```

输出:

```
30 10 20
```

需要注意的是：匿名字段嵌入并不是继承，Go 语言根本没有继承概念。

我们不能把下面例子中的 m 直接赋值给 u，而只能是 m.User 进行相同类型的值拷贝。(这种父类变量引用子类对象的行为，称为多态，是 OOP 的重要特征之一)

```
type User struct {
    Id int
    Name string
}

type Manager struct {
```

```
    Group string
    User
}

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

    //var u User = m         // Error: cannot use m (type Manager) as type User in assignment
    var u User = m.User      // value copy

    fmt.Println(m, u)
}
```

试试直接嵌入匿名类型指针。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    *User
}

func main() {
    m := Manager{ User:&User{ 1, "Tom" }, Group:"IT" }     // 注意初始化字段名

    fmt.Println(m, m.Id, m.Name)                           // 就算是指针，访问成员也木有压力。
}
```

输出:

```
{IT 0x42114b40} 1 Tom
```

不能同时嵌入 User 匿名类型和 *User 指针匿名类型，这将引发类似 "duplicate field User" 错误。

4.4 方法

Go 虽然没有 class，但同样支持 method，使用方法和我们习惯的面向对象编程方式相似。更接近 Python Method，包括在函数名前面这个被称为 Receiver 的参数。

Receiver 在多数语言中是隐式传递的 this，在 Python 中通常写作 self。

```
type User struct {
    Id   int
    Name string
}

func (this *User) Test() {
    println(this.Id, this.Name)
}
```

```
func (this User) Test2() {
    println(this.Id, this.Name)
}

func main() {
    u := &User{1, "Jack"}
    u.Test()
    u.Test2()
}
```

方法和函数规则类似，只是在函数名前面多了绑定类型参数 Receiver。

- 只能为相同包中的类型定义方法。
- 如果方法代码不使用 Receiver，那么可以省略变量名。
- Receiver 类型可以是 T 或 *T，我们把 T 称为 BaseType。
- BaseType 不能是指针或接口。
- T 和 *T 的方法都被绑定 (bound) 到 BaseType。
- 就算 Receiver 类型不同，也不能定义同名方法。
- 可以用 value 或 pointer 调用所有绑定的方法，编译器自动进行类型转换。

```
type User struct {
    Name string
}

func (User) Test() {                // 省略 Receiver 的参数名
    println("Test")
}

func (this *User) Test2() {         // 因 method set 的缘故，名字不能相同，就算 Receiver 类型不同也不行。
    println("Test2:", this.Name)    //   否则导致 "method redeclared" 编译错误!
}

func main() {
    u := User{"Tom"}
    p := &u

    u.Test()
    u.Test2()                       // 自动转换，相当于 (&u).Test2()。

    p.Test()                        // 自动转换，相当于 (*p).Test()。
    p.Test2()
}
```

输出:

```
Test
Test2: Tom
Test
Test2: Tom
```

方法不是 struct 的专利，我们给任何非指针和接口类型的本地类型 (同一个包) 添加方法。

```
type MyInt int                          // int 不再当前包中，因此用 MyInt "重新定义"一个类型。
type MySlice []int

func (this MyInt) Test() {
    fmt.Printf("MyInt = %v\n", this)
}

func (this MySlice) Test() {
    fmt.Printf("MySlice = %v\n", this)
}

func main() {
    i := MyInt(10)
    i.Test()

    s := MySlice{ 100, 200 }
    s.Test()
}
```

4.4.1 Method Values vs. Method Expressions

从某种意义上来说，方法是函数的 "语法糖"。当函数与某个特定的类型相绑定 ，那么它就是一个方法。也正因为如此，我们可以将方法 "还原" 成函数。

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

为了区别这两种方式，官方文档中将左边的称为 Method Value，右边则是 Method Expression。

Method Value 是包装后的状态对象，总是与特定的对象实例关联在一起 (类似闭包，拐带私奔)，而 Method Expression 函数将 Receiver 作为第一个显式参数，调用时需额外传递。

```
type User struct {
    Name string
}

func (this User) Test(x int)      {}
func (this *User) Test2(s string) {}

func main() {
    u := User{"Tom"}

    /* === Method Value ====== */

    u.Test(123)
    u.Test2("abc")

    var f1 func(int) = u.Test                       // 签名中没有 Receiver。
    f1(123)

    var f2 func(string) = u.Test2
```

```
    f2("abc")

    /* === Method Expression ====== */

    User.Test(u, 123)                               // 显式提供 Receiver 实参。
    (*User).Test2(&u, "abc")

    var f3 func(User, int) = User.Test              // 签名中显式提供 Receiver 参数。
    f3(u, 123)                                      // 调用也须提供 Receiver 实参。

    var f4 func(*User, string) = (*User).Test2
    f4(&u, "abc")
}
```

将 Method Value 赋值给变量，Receiver 实例立即被复制。也就是说，如果是 Value Receiver，那么和原对象再没啥联系了。

```
type User struct {
    Name string
}

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

func main() {
    u := User{"Tom"}
    f := u.Test                     // 为 f 专门准备了一个 Receiver 复制品。

    u.Name = "xxx"                  // 对 u 的修改不会影响 f 的 Receiver。
    fmt.Println(u)

    f()                             // f 使用的是自带的 Receiver，和 u 没啥关系。
}
```

输出:

```
{xxx}
{Tom}
```

Method Expression 转换必须符合方法集 (Method Set) 规则。

- T 方法集仅拥有 T Receiver 方法。
- *T 方法集则包含全部方法 (T + *T)。

可以将 T.Method 转换为 *T.Method 使用，因为 *T 方法集包含 T.Method 方法。反过来则会导致 "T.Method undefined" 编译错误。

将 T.Method 转换为 *T.Method，函数签名会发生变化，其 Receiver 参数类型变成 *T。传递实参时须使用转换后的类型，最终由编译器在内部还原。

```
type User struct {
    Name string
```

```
}

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

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

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

    // User 方法集中不包含 TestPointer，编译错误！
    //   Error: User.TestPointer undefined (type User has no method TestPointer)
    // User.TestPointer(&u)

    // *User 方法集中包含了 TestValue、TestPointer。
    // 注意调用 TestValue 传递的是 Receiver Pointer 实参。
    (*User).TestValue(&u)
    (*User).TestPointer(&u)

    // 用反射查看签名变化。
    m, _ := reflect.TypeOf(&u).MethodByName("TestValue")
    fmt.Println(m)
}
```

输出:

```
u: 0x210196150, {Tom}
V: 0x210196190, {Tom}
P: 0x210196150, {Tom}
{TestValue  func(*main.User) <func(*main.User) Value> 1}
```

基于以上种种，看到下面这样的代码就没啥可惊讶的了。

```
type User struct { Name string }
func (User) Test()   {}
func (*User) Test2() {}

func main() {
    var u *User
    u.Test2()

    (*User)(nil).Test2()
    (*User).Test2(nil)

    // 不管怎样，编译器调用 func Test(User) 总归要有一个合法的 User Receiver 实例。
    //   runtime error: invalid memory address or nil pointer dereference
    // (*User)(nil).Test()
}
```

4.4.2 Receiver

无论是调用 Method Value 还是 Method Expression，都需要隐式或显示传递 Receiver 实例。如果 Receiver 不用指针类型，那么必然会发生值拷贝行为，就成两个对象了，在方法内部对该对象的修改不会影响到 Caller。

```
func (this User)test() {
    this.Id = 200;
    this.Name = "Tom"
    println(&this, this.Id, this.Name)
}

func main() {
    u := User{ 1, "Jack" }
    u.test()
    println(&u, u.Id, u.Name)
}
```

输出:

```
0x442085f78 200 Tom
0x442085f90 1 Jack
```

不见得使用指针就一定好过传值，因为按照 Go 的内存管理策略，涉及指针和引用的对象会被分配到 GC Heap 上。如果对象很 "小"，显然要比在栈上进行值拷贝 "耗费" 更多。

4.4.3 匿名字段方法

可直接访问匿名字段 (匿名类型或匿名指针类型) 的方法，这种行为类似 "继承"。访问匿名字段方法时，同样有隐藏规则。正是基于这种隐藏规则，我们很容易实现 override 效果。

因为编译器会自动将 receiver 转换为 value 或 pointer。正如下面例子中，Manager.Test receiver 类型可以不是指针。

```
type User struct {
    Id   int
    Name string
}

type Manager struct {
    User
    Group string
}

func (this *User) Test() {
    println("User Test:", this.Id, this.Name)
}

func (this User) ToString() string {
    return fmt.Sprintf("User ToString: [%d] %s", this.Id, this.Name)
}
```

```
func (this Manager) Test() {
    println("Manager Test:", this.Id, this.Name)
}

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

    m.Test()
    println(m.ToString())
}
```

输出:

```
Manager Test: 1 Tom
User ToString: [1] Tom
```

用匿名类型名称可以做到 "基类转型" 的效果。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    User
}

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

    var p *User = &m.User            // 并不是真的多态，只不过是访问其内部的字段而已。
    println(p.Id, p.Name)
}
```

通过下面这个例子，你会发现 Test 的 this 指向 Manager.User 成员指针。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    User                             // 放在第二个位置，offset != 0，使其和外层 Manager 内存地址不同。
}

func (this *User) Test() {
    fmt.Printf("Test this address = %p\n", this)
}

func main() {
    m := &Manager{ User:User{ 1, "Tom" }, Group:"IT" }
```

```
    fmt.Printf("m.address = %p\n", m)
    fmt.Printf("m.User.address = %p\n", &m.User)

    m.Test()
}
```

输出:

```
m.address = 0x421016c0
m.User.address = 0x421016d0
Test this address = 0x421016d0
```

方法总是从 Receiver 类型开始查找目标成员。下面例子中，A.Test() 总是访问 A.x。

```
type A struct {
    x int
}

type B struct {
    A
    x int
}

func (this *A) Test() {
    println("A:", this.x)            // A.x，因为 this *A 类型根本不包括 B 的成员。
}

func (this *B) Test() {
    println("B:", this.x)            // B.x
}

func main() {
    o := B{x: 123}
    o.Test()
    o.A.Test()
}
```

输出:

```
B: 123
A: 0
```

4.5 内存布局

借助于 unsafe 包，我们可以探查一下 struct 的内存布局。

```
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    Id int
```

```
    Name string
}

type Manager struct {
    Group string
    User
}

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

    fmt.Printf("m alignof = %d\n", unsafe.Alignof(m))
    fmt.Printf("m address = %p\n", &m)
    fmt.Printf("m size = %d\n", unsafe.Sizeof(m))

    fmt.Printf("m.Group address = %p\n", &m.Group)
    fmt.Printf("m.Group offset = %d\n", unsafe.Offsetof(m.Group))
    fmt.Printf("m.Group size = %d\n", unsafe.Sizeof(m.Group))

    fmt.Printf("m.User address = %p\n", &m.User)
    fmt.Printf("m.User offset = %d\n", unsafe.Offsetof(m.User))
    fmt.Printf("m.User size = %d\n", unsafe.Sizeof(m.User))

    fmt.Printf("  User.Id address = %p\n", &m.User.Id)
    fmt.Printf("  User.Id offset = %d\n", unsafe.Offsetof(m.User.Id))
    fmt.Printf("  User.Id size = %d\n", unsafe.Sizeof(m.User.Id))

    fmt.Printf("  User.Name address = %p\n", &m.User.Name)
    fmt.Printf("  User.Name offset = %d\n", unsafe.Offsetof(m.User.Name))
    fmt.Printf("  User.Name size = %d\n", unsafe.Sizeof(m.User.Name))
}
```

输出:

```
m alignof = 8
m address = 0x421016c0
m size = 40

m.Group address = 0x421016c0
m.Group offset = 0
m.Group size = 16

m.User address = 0x421016d0
m.User offset = 16
m.User size = 24                            // 内容长度 20，按 8 字节对齐后为 24。
  User.Id address = 0x421016d0
  User.Id offset = 0
  User.Id size = 4
  User.Name address = 0x421016d8
  User.Name offset = 8
  User.Name size = 16
```

4.6 字段标签

可以为字段定义标签，用反射可以读取这些标签。做什么用呢？比如像下面这样：

```
package main

import (
    "reflect"
    "fmt"
)

type User struct {
    Name string "姓名"
    Age  int    "年龄"
}

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

    t := reflect.TypeOf(u)
    v := reflect.ValueOf(u)

    for i := 0; i < t.NumField(); i++ {
        f := t.Field(i)
        fmt.Printf("%s (%s = %v)\n", f.Tag, f.Name, v.Field(i).Interface())
    }
}
```

输出：

```
姓名 (Name = Tom)
年龄 (Age = 23)
```

第 5 章 接口

接口是一个或多个方法签名的集合，任何非接口类型只要拥有与之对应的全部方法实现 (包括相同的名称、参数列表以及返回值。当然除了接口所规定的方法外，它还可以拥有其他的方法)，就表示它 "实现" 了该接口，无需显式在该类型上添加接口声明。此种方式，又被称作 Duck Type。

5.1 接口定义

接口定义看上去和 struct 类似，区别在于：

- 只有方法签名，没有实现。
- 没有数据字段。

习惯性地将接口命名以 er 结尾，而不是以往习惯性地以大写字母 "I" 开头。

```
type Tester interface {
    Test()
}
```

在接口中匿名嵌入另外的接口，这类似我们熟悉的接口继承。

```
type Reader interface {
    Read()
}

type Writer interface {
    Write()
}

type ReadWriter interface {
    Reader
    Writer
}

type ReadWriteTest struct {
}

func (*ReadWriteTest) Read() {
    println("Read")
}

func (*ReadWriteTest) Write() {
    println("Write")
}

func main() {
    t := ReadWriteTest{}

    var rw ReadWriter = &t
```

```
    rw.Read()
    rw.Write()
}
```

接口是可被实例化的类型，而不仅仅是语言上的约束规范。当我们创建接口变量时，将会为其分配内存，并将赋值给它的对象拷贝存储。

可以像下面这样，把接口类型匿名嵌入到 struct 中。

```
type Tester interface {
    Test()
}

type MyTest struct {
}

func (this *MyTest)Test() {
    fmt.Println("Test")
}

type User struct {
    Id int
    Name string
    Tester
}

func main() {
    u := User{ 100, "Jack", nil }
    u.Tester = new(MyTest)

    fmt.Println(u)
    u.Test()
}
```

输出:

```
{100 Jack 0x420e51c0}
Test
```

和结构一样，我们也可以创建匿名接口类型。

```
type User struct {
    Name string
}

func (this *User) String() string {
    return fmt.Sprintf("Name: %s", this.Name)
}

type Data struct {
    s interface {                           // 匿名接口类型成员
        String() string
    }
}
```

```
func main() {
    var i interface {                       // 匿名接口类型变量
        String() string
    } = &User{"Tom"}

    fmt.Println(i.String())

    d := Data{&User{"Jack"}}
    fmt.Println(d.s.String())
}
```

另外一种技巧让函数以接口的面目出现。

```
type Tester interface {
    Do(s string)
}

type DoFunc func(string)

func (f DoFunc) Do(s string) {
    f(s)
}

func main() {
    var d DoFunc = func(s string) { println(s) }
    var t Tester = d
    t.Do("Hello, World!")
}
```

5.2 执行机制

接口对象内部由两部分组成： itab 指针 、data 指针。

src/pkg/runtime/runtime.h

```
151    struct Iface
152    {
153            Itab*  tab;
154            void*  data;
155    };
```

src/pkg/runtime/iface.c

```
32     struct Itab
33     {
34             InterfaceType* inter;
35             Type*  type;
36             Itab*  link;
37             int32  bad;
38             int32  unused;
39             void   (*fun[])(void);
40     };
```

http://research.swtch.com/interfaces
http://groups.google.com/group/golang-nuts/browse_thread/thread/f01465cb6206ad6/86eaf4610512d351?lnk=gst

将对象赋值给接口变量时，会发生值拷贝行为。没错，接口内部存储的是指向这个复制品的指针。而且我们无法修改这个复制品的状态，也无法获取其指针。

```
type User struct {
    Id int
    Name string
}

type Tester interface {
    Test()
}

func (this User) Test() {}

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

    u.Id = 100                  // 显式修改的原对象
    u.Name = "Jack"

    fmt.Println(u, i)           // 虽然接口内的复制品未受到影响

    //i.(User).Name = "Jack"    // 无法操作: cannot assign to i.(User).Name
                                // 要是指针就没问题了。i.(*User).Name = ...，指针指向原始对象。
    fmt.Println(&(i.(User)))    // cannot take the address of i.(User)
}
```

输出:

```
{100 Jack} {1 Tom}
```

只有当接口存储的类型和对象都为 "空" 时，接口才等于 nil。

```
func main() {
    var a interface{} = nil                             // type: nil,  value: ZeroValue
    var b interface{} = (*int)(nil)                     // type: *int, value: nil

    ta, va := reflect.TypeOf(a), reflect.ValueOf(a)
    tb, vb := reflect.TypeOf(b), reflect.ValueOf(b)

    fmt.Println(a == nil, b == nil)                     // true, false
    fmt.Println(b == nil || vb.IsNil())                 // true

    fmt.Println(ta, va.IsValid())                       // <nil>, false
    fmt.Println(tb, vb.IsValid())                       // *int,  true
}
```

再说说接口方法调用机制。

itab 依据 data 类型创建，存储了接口动态调用的元数据信息，其中包括 data 类型所有符合接口签名的方法地址列表。用接口对象调用方法时，就从 itab 中查找所对应的方法，并将 *data (指针) 作为 receiver 参数传递给该方法，完成实际目标方法调用。

编译器构建 itab 时，会区分 T 和 *T 方法集 (method sets)，并从中获取接口实现方法的地址。接口调用不会做 receiver 自动转换，目标方法必须在接口实现方法集中。

- T 仅拥有属于 T 类型的方法集，而 *T 则同时拥有 T + *T 方法集。
- 基于 T 实现方法，表示同时实现了 interface(T) 和 interface(*T) 接口。
- 基于 *T 实现方法，那就只能是对 interface(*T) 实现接口。

Method sets

A type may have a method set associated with it (Interface types, Method declarations). The method set of an interface type is its interface. The method set of any other named type T consists of all methods with receiver type T. The method set of the corresponding pointer type *T is the set of all methods with receiver *T or T (that is, it also contains the method set of T). Any other type has an empty method set. In a method set, each method must have a unique name.

http://golang.org/doc/go_spec.html

为什么使用不同的方法集，究其原因是接口实现时的 value-copy。如果是 pointer-interface，T 和 *T method 确保都能正确执行。而如果是 value-interface，那么 *T 方法是无法保证合法操作这个 readonly-data 的。官方的说法是无法获取其指针。在使用接口时，如果需要修改原对象，记得用指针赋值。

Why do T and *T have different method sets?

From the Go Spec:

The method set of any other named type T consists of all methods with receiver type T. The method set of the corresponding pointer type *T is the set of all methods with receiver *T or T (that is, it also contains the method set of T).

If an interface value contains a pointer *T, a method call can obtain a value by dereferencing the pointer, but if an interface value contains a value T, there is no useful way for a method call to obtain a pointer.

Even in cases where the compiler could take the address of a value to pass to the method, if the method modifies the value the changes will be lost in the caller. As a common example, this code:

var buf bytes.Buffer

io.Copy(buf, os.Stdin)

would copy standard input into a copy of buf, not into buf itself. This is almost never the desired behavior.

http://golang.org/doc/go_faq.html#different_method_sets

```
type User struct {
    Id int
    Name string
}

type Tester interface {
    Test()
}

type Stringer interface{
    String()
}

func (this User) Test() {
    fmt.Println("Test:", this)
}

func (this *User) String() {
    fmt.Printf("String: Id=%d, Name=%s\n", this.Id, this.Name)
}

func main() {
    u := User{ 1, "Tom" }
    p := &u

    // User.Test() 同时实现了 Tester(User) 和 Tester(*User) 接口
    Tester(u).Test()
    Tester(p).Test()

    // *User.String() 仅实现了 Stringer(*User) 接口。下面这个错误就是证明。
    // Error: cannot convert u (type User) to type Stringer:
    //        User does not implement Stringer (String method requires pointer receiver)
    //Stringer(u).String()
    Stringer(p).String()
}
```

输出:

```
Test: {1 Tom}
Test: {1 Tom}
String: Id=1, Name=Tom
```

5.3 匿名字段方法

接口同样支持由 struct 匿名字段实现的方法，因为外层结构 "继承" 了匿名字段的方法集。

匿名字段方法集规则:

- 如果 S 嵌入匿名类型 T，则 S 方法集包含 T 方法集。
- 如果 S 嵌入匿名类型 *T，则 S 方法集包含 *T 的方法集 (T + *T)。
- 如果 S 嵌入匿名类型 T 或 *T，则 *S 方法集包含 *T 的方法集 (T + *T)。

下例中，*User 实现了 Tester 接口，而 Manager 嵌入了 User 匿名字段，是以同样认为 *Manager 实现了 Tester 接口。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    User
}

type Tester interface {
    Test()
}

func (this *User)Test() {
    fmt.Println(this)
}

func main() {
    u := User{ 1, "Tom" }
    m := Manager{ User:User{ 2, "Jack" }, Group:"IT" }

    var i Tester

    i = &u                    // 我们为 *User 定义了 Test()，自然表示实现了该接口。
    i.Test()

    i = &m                    // 匿名字段，同样也拥有 Test()
    i.Test()
}
```

借助于接口，Go 完全可以实现 OOP 所需的操作。

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    User
```

```
}

type Tester interface {
    Test()
}

func (this *User)Test() {            // BaseClass method
    fmt.Println(this)
}

func (this *Manager)Test() {         // Override BaseClass method
    fmt.Println(this)
}

func dosomething(o Tester) {         // 正确调用 *User.Test 和 *Manager.Test，类似多态的功能。
    o.Test()
}

func main() {
    m := Manager{ User:User{ 2, "Jack" }, Group:"IT" }

    dosomething(&m)
    dosomething(&m.User)
}
```

输出:

```
&{IT {2 Jack}}
&{2 Jack}
```

5.4 空接口

任何类型默认都实现了空接口 interface{}，这就是 C void*，Java/C# System.Object。

如果要接收任意类型的参数，可以使用 interface{}。

```
func test1(a int, b interface{}) {}
func test2(a int, b ...interface{}) {}
```

5.5 类型推断

利用类型推断，可以判断接口对象是否是某个具体的接口或类型。

```
type User struct {
    Id int
    Name string
}

type Tester interface {
    Test()
}
```

```
func (this *User)Test() {
    fmt.Println("*User.Test, ", this)
}

func doSomething(i interface{}) {
    // 推断并检查是否实现了 Tester 接口，不会引发错误。
    if o, ok := i.(Tester); ok {
        o.Test()
    }

    // 直接利用推断进行转型。
    // 如果类型推断失败，或者方法不存在都会导致错误。
    i.(*User).Test()
}

func main() {
    u := &User{ 1, "Tom" }
    doSomething(u)
}
```

输出:

```
*User.Test,  &{1 Tom}
*User.Test,  &{1 Tom}
```

而 type switch 则适合处理 interface{} 这种 "无类型" 参数。".(type)" 也仅能用于 switch 语句。

```
type User struct {
    Id int
    Name string
}

func Test(i interface{}) {
    switch v := i.(type) {
        case *User:
            println("User: Name =", v.Name)
        case string:
            println("string = ", v)
        default:
            fmt.Printf("%T: %v\n", v, v)
    }
}

func main() {
    Test(&User{ 1, "Tom" })
    Test("Hello, World!")
    Test(123)
}
```

输出:

```
User: Name = Tom
string =  Hello, World!
int: 123
```

5.6 接口转换

拥有超集的接口可以被转换为子集的接口。比如下面例子中，IManager 拥有 IUser 的全部方法签名，我们可以直接将 im 赋值给 iu。(此例貌似看出 er 作为接口标志的弊端了，还是 I 开头好)

```
type User struct {
    Id int
    Name string
}

type Manager struct {
    Group string
    User
}

type IUser interface {
    Test()
}

type IManager interface {
    Test()
    Test2()
}

func (this *User)Test() { fmt.Println(this) }
func (this *User)Test2() { fmt.Println(this) }

func main() {
    var im IManager = &Manager{"IT", User{1, "Tom"}}
    im.Test()
    im.Test2()

    var iu IUser = im
    iu.Test()
}
```

第 6 章 并发

6.1 Goroutine

作为 Go 语言的一个宣传亮点，goroutine 很吸引人。为更好地理解和适应该它所带来的并发编程变化，我们应该搞清其背后的原理。避开理念和实现细节，会发现官方的默认实现，也只是一个豪华版的 "线程池" 而已。

关键字 go 的本质是：调用相关函数创建一个在内部称为 G (goroutine) 的对象，该对象相当于一个准备扔到线程池中运行的任务。G 除了持有 goroutine 函数地址外，还分配有执行所需的堆栈帧 (stack frame) 内存，另有字段来实现主要寄存器的功能。如此，线程不再持有 "状态"，以实现 "multiplexed thread" 的目的。

不光是 G，还抽象了处理器 P (processor) 和 线程 M (worker thread)。所有的 G 都放在队列中，由一个或多个 M 循环提取执行。M 要执行任务，首先得关联一个处理器 P。P 的数量可通过 GOMAXPROCS() 函数改变，这就是控制并发线程数量的根本手段。

注意，P 仅控制并发数量，不表示进程内只有和 P 数量相等的 M。当某个 M 执行会引发系统调用阻塞 (syscall block) 的 G 时，会暂时释放 P。在此期间，如果列表中还有待运行的 G，那么调度器会创建新的 M 拿着这个 P 去继续执行任务。被阻塞的 M 从系统调用退出时，如果能重新拿到 P，任务自然继续下去。拿不到的话，就只能把未完成的 G 放回队列，让其他活跃的 M 完成后续工作。反正相关状态保存在 G 本身，用哪个 M 完成并不重要。失去 P 的 M 去空闲队列等待召唤，因为调度器会优先使用空闲的 M，而非新建。某些时候，当你发现进程中多了许多线程，大抵就是这样造成。当前官方实现 (go 1.1) 并没有释放多余线程或者限制总数量的手段，只能由开发人员自行解决。比如用几个 goroutine 实现传统意义上的 thread pool，只需将函数指针放到 channel 即可。

M 执行 G 的实现很简单，用汇编指令将 G 相关字段压入寄存器 SP，配合其他字段就能确定执行堆栈帧空间。然后 JMP 到 goroutine 函数地址，开始执行。通常将这种方式称为分段栈。

该实现机制在一定程度上避免了频繁创建和销毁线程的开销，降低并发执行所需成本。而且 G 默认仅需 4~5KB 的栈内存 (按需增减)，相比线程栈也少得多。正因为如此，才可能 "同时运行" 成千上万个并发任务。还有语言层面的便利，也让 goroutine 比 thread 更易用。

```
func main() {
    go func() {
      time.Sleep(3 * time.Second)
      println("go...")
    }()

    println("hi!")
    time.Sleep(5 * time.Second)
```

```
}
```

输出:

```
hi!
go...
```

如果不了解内部执行机制，我们很容易把 goroutine 当做 协程 (coroutine) 这类东西，但它们仅仅是表面上相似。协程 coroutine 通常用同一线程进行切换调度 (yield/resume) 来实现并发执行，而 goroutine 可用多个线程实现多核并行。

"并发 (concurrency)" 和 "并行（parallelism)" 是不同的。在单个 CPU 核上，线程通过时间片或者让出控制权来实现任务切换，达到 "同时" 运行多个任务的目的，这就是所谓的并发。但实际上任何时刻都只有一个任务被执行，其他任务通过某种算法来排队。多核 CPU 可以让同一进程内的 "多个线程" 做到真正意义上的同时运行，它们之间不需要排队 (依然会发生排队，因为线程数量可能超出 CPU 核数量，还有其他的进程等等。这里说的是一个理想状况)，这才是并行。除了多核，并行计算还可能是多台机器上部署运行。

多个 goroutine 运行在同一进程地址空间内，在共享内存数据时通常遵循规则:

Do not communicate by sharing memory. Instead, share memory by communicating。

和官方 6g 编译器基于 multiplexed thread 实现 goroutine 不同，gccgo 为每个 goroutine 准备一个独立线程。相比较之下，6g 版本更加高效和轻便。相关细节可参考附录《源码阅读指南》。

默认情况下，调度器仅使用单线程循环执行 goroutine，也就是说只实现了并发。 想要发挥多核处理器的并行威力，可设置环境变量 GOMAXPROCS 或调用 runtime.GOMAXPROCS(n) 进行调整。范围介于 [1, 256] 之间，通常和 CPU Core 数量 (runtime.NumCPU) 相等。该函数设置新并发线程数量，并返回设置前的值。当 n <= 0 时仅返回当前设置。也可直接使用系统环境变量，避免重新编译。

当 Go 进程启动后，默认有两个 goroutine 执行:

- scavenger: 定时执行垃圾清理。
- main: 程序入口，对应 main() 函数。

还有，进程退出时不会等待 goroutine 完成，需要我们自行处理。

下面的例子，在 MacBook Pro 374 上，启用并行后性能增长非常明显。

```
package main

import ()

func test(c chan bool, n int) {
    x := 0
    for i := 0; i < 100000000; i++ {
        x += i
    }
```

```
    println(n, x)

    if n == 9 {
        c <- true
    }
}
func main() {
    c := make(chan bool)

    for i := 0; i < 10; i++ {
        go test(c, i)
    }

    <-c
}
```

输出:

```
$ go build

$ time GOMAXPROCS=1 ./test

0 4999999950000000
1 4999999950000000
2 4999999950000000
3 4999999950000000
4 4999999950000000
5 4999999950000000
6 4999999950000000
7 4999999950000000
8 4999999950000000
9 4999999950000000

real    0m2.562s
user    0m2.532s
sys     0m0.008s

$ time GOMAXPROCS=2 ./test

0 4999999950000000
1 4999999950000000
2 4999999950000000
6 4999999950000000
3 4999999950000000
7 4999999950000000
4 4999999950000000
8 4999999950000000
5 4999999950000000
9 4999999950000000

real    0m1.684s
user    0m2.535s
sys     0m0.020s
```

6.2 Channel

channel 是类似 pipe 的单/双向数据管道。从设计上确保，在同一时刻，只有一个 goroutine 能从中接收数据。发送和接收都是原子操作，不会中断，只会失败。

6.2.1 阻塞同步

channel 是引用类型，使用 "make(chan <type>)" 创建，<type> 是要传递的数据类型。

channel 使用 "<-" 接收和发送数据。"chan <- data" 发送数据， "data <- chan" 接收数据，可以忽略掉接收结果，纯粹作为同步信号通知。默认情况下，发送和接收都会被阻塞 (block)。

- 发送操作被阻塞，直到接收端准备好接收。
- 接收操作被阻塞，直到发送端准备好发送。

```
func test(c chan int) {
  time.Sleep(3 * time.Second)
  println("go...")

  c <- 1
}

func main() {
    c := make(chan int)
    go test(c)

    println("hi!")
    <- c                      // 阻塞等待退出信号，忽略掉返回的数据。
    println("over!")
}
```

输出:

```
hi!
go...
over!
```

常用的做法是将并发任务和创建 channel 放在一个工厂函数中 (参考 time.Tick 实现)。

```
func task(args ...int) chan int {
    x := 0
    c := make(chan int)

    go func() {
        time.Sleep(2 * time.Second)

        for _, v := range args {
            x += v
        }
```

```
        c <- x
    }()

    return c
}

func main() {
    c := task(1, 2, 3, 4)

    println("do something...")

    i := <- c
    println("task result =", i)
}
```

输出:

```
do something...
task result = 10
```

读取关闭的 channel 不会被阻塞。

```
func main() {
    ch := make(chan bool)
    close(ch)

    v, ok := <-ch
    fmt.Println(v, ok)
}
```

输出:

```
false false
```

读写 nil channel 都会被阻塞。

6.2.2 迭代器

可以用 range 迭代器从 channel 中接收数据，直到 channel close 方才终止循环。

```
func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 20; i++ {
            c <- i
        }

        close(c)              // 如果不关闭，会引发接收端 throw: all goroutines are asleep - deadlock!
    }()

    for v := range c {
        println(v)
    }
}
```

接收方还可以用另一种方式代替迭代器接收数据并判断 channel close 状态。

```
func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 20; i++ {
            c <- i
        }

        close(c)
    }()

    for {
        if v, ok := <- c; ok {
            println(v)
        } else {
            break;
        }
    }
}
```

只有发送端 (另一端正在等待接收) 才能 close，只有接收端才能获得关闭状态。close 调用不是必须的，但如果接收端使用迭代器或者循环接收数据，则必须调用，否则可能导致接收端 throw: all goroutines are asleep - deadlock!。

6.2.3 单向通道

可以将 channel 指定为单向通道。比如 "<-chan int" 仅能接收， "chan<- int" 仅能发送。

```
func recv(c <-chan int, over chan<- bool) {
    for v := range c {
        println(v)
    }

    over <- true
}

func send(c chan<- int) {
    for i := 0; i < 10; i++ {
        c <- i
    }

    close(c)                  // 省略 close，会导致 recv throw: all goroutines are asleep - deadlock!
}

func main() {
    c := make(chan int)       // 双向 channel，可以转换为单向 channel。
    o := make(chan bool)
```

```
    go recv(c, o)
    go send(c)

    <- o
}
```

6.2.4 异步通道

异步 channel，就是给 channel 设定一个 buffer 值，在 buffer 未填满的情况下，不阻塞发送操作。buffer 未读完前，不阻塞接收操作。buffer 是被缓冲数据对象的数量，不是内存大小。

通常情况下，buffered channel 拥有更高的性能。但 buffer 值不是越大越好，过多的 items 可能造成性能瓶颈。可以考虑将多个 item 打包，以减少 buffer 内容的数量。

```
c := make(chan int, 10)
```

buffer 仅仅是 channel 对象的一个内部属性，它的类型依然是 "chan <type>"。

```
func main() {
    c := make(chan int, 10)
    o := make(chan bool)

    go func() {
        time.Sleep(2 * time.Second)
        println("recv:", <- c)
        o <- true
    }()

    c <- 100                        // 未阻塞
    println("send over...")

    <- o
}
```

输出:

```
send over...
recv: 100
```

使用异步 channel 实现 semaphore 同步信号量 (限制并发执行的数量) 的例子。

```
import (
    "fmt"
    "sync"
    "time"
)

var sem = make(chan int, 2)                         // 信号量，并发限制为 2。
var wg = sync.WaitGroup{}                           // 用于等待所有 goroutine 结束。

func worker(i int) {
    sem <- 1                                        // 抢个位子先。如果没有空位，就只能等待。
```

```
    fmt.Println(time.Now().Format("04:05"), i)      // 输出时间，为了看看效果。
    time.Sleep(1 * time.Second)                     // 模拟等待时间，否则很快结束。

    <-sem                                           // 完成工作后，让出空位。
    wg.Done()                                       // 当前 goroutine 结束。
}

func main() {

    for i := 0; i < 10; i++ {                       // 黑心老板一次丢出 10 个工作
        wg.Add(1)                                   // 增加 goroutine 计数。
        go worker(i)
    }

    wg.Wait()                                       // 等待全部 goroutine 结束。
}
```

输出:

```
22:50 0                                             // 注意观察相同时间的并发输出
22:50 1
22:51 2
22:51 3
22:52 4
22:52 5
22:53 6
22:53 7
22:54 8
22:54 9
```

6.2.5 Select

如果有多个 channel 需要监听，可以考虑使用 select，随机处理一个可用的 channel。

还可以定义 case default，如果没有可用 channel，那么不阻塞，直接执行 default block 。如果外面套了循环，就成洪水泛滥了。

```
func main() {
    c1, c2 := make(chan int), make(chan string)
    o := make(chan bool)

    go func() {
        for {
            select {
                case v, ok := <- c1:
                    if !ok { o <- true; break }
                    println("c1 =", v)
                case v, ok := <- c2:
                    if !ok { o <- true; break }
                    println("c2 =", v)
            }
        }
```

```
    }()

    c1 <- 1
    c2 <- "a"
    c2 <- "b"
    c1 <- 2

    close(c1)
    //close(c2)

    <- o
}
```

输出:

```
c1 = 1
c2 = a
c2 = b
c1 = 2
```

select 也可用于发送端，由于其随机选择一个可用的 case，那么折腾同一个 channel 也是可以滴。

```
func main() {
    c := make(chan int)
    o := make(chan bool)

    go func() {
        for v := range c {
            print(v)
        }
        o <- true
    }()

    for i := 0; i < 10; i++ {
        select {                    // 随机发送 0 或 1
            case c <- 0:            // no statement, no fallthrough
            case c <- 1:
        }
    }

    close(c)
    <- o
}
```

如果有其他 goroutine 正在运行，可以用空的 "select {}" 阻塞 main()，避免进程终止。

```
func main() {
    go func() {
        for {
            fmt.Println(time.Now())
            time.Sleep(time.Second)
        }
    }()

    select {}
```

```
}
```

6.2.6 Timeout

time 包还提供了 After、Tick 等函数返回计时器 channel。比如下面的例子中，我们用 After 来处理一个需求：不管有没有数据操作，总之 5 秒后终止循环。

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    tick := time.Tick(1 * time.Second)

    go func() {
        for {
            select {
                case v := <- c:
                    println(v)
                case <- tick:
                    println("tick")
                case <- time.After(5 * time.Second):       // 只用一次，没必要用外部变量。
                    println("timeout")
                    o <- true
                    break
            }
        }
    }()

    <- o
}
```

6.2.7 Multiplexing

channel 本身也可以作为数据发送给其他的 channel。

下面例子中，client 将要处理的数据连同与之通讯的 channel 对象一并打包发给 server。server 使用一个独立的 Request Channel 循环 "监听" 所有的 client 接入请求，并将接收到的数据立即提交给另一个 goroutine 做并发处理，实现接入和处理的并发操作。而处理数据的 goroutine 直接从数据包中获取与之通讯的 channel，返回结果给 client。这和我们熟悉的 socket server 编程模型类似。

```
type Data struct {
    args []int
    ch chan string
}

func server() chan *Data {
    reqs := make(chan *Data)         // 服务器 Request Channel

    go func() {                      // 使用独立的 goroutine 循环处理所有的客户端请求
```

```
      for d := range reqs {          // 循环从 Request Channel 获取客户端请求数据
          go serverProcess(d)        // 将请求的客户端数据交给另外的 goroutine 处理，并立即等待下一个请求。
      }
    }()

    return reqs                      // 返回 Request Channel，用于 close server。
}

func serverProcess(data *Data) {
    x := 0
    for _, i := range data.args {    // 统计用户请求数据
        x += i
    }

    s := fmt.Sprintf("server: %d", x)

    data.ch <- s                     // 使用客户端请求数据包中的 Channel，返回结果给客户端。
}

func main() {
    /* 启动服务器 */
    serverReqs := server()

    /* 客户端向服务器发送请求数据，并等待返回结果 */
    data := &Data{ []int{1, 2, 3}, make(chan string) }
    serverReqs <- data  // 发送请求到服务器
    println(<- data.ch) // 获取服务器返回结果

    /* 关闭服务器 */
    close(serverReqs)
}
```

6.2.8 runtime Goroutine

runtime 包中提供了几个与 goroutine 有关的函数。

Gosched() 类似 yield，让出当前 goroutine 的执行权限。调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。

```
func main() {
    go func() {
        for i := 0; i < 5; i++ {
            println("go1:", i)
            if i == 2 { runtime.Gosched() }
        }
    }()

    go func() {
        println("go2")
    }()

    time.Sleep(3 * time.Second)
```

```
}
```

输出:

```
go1: 0
go1: 1
go1: 2
go2
go1: 3
go1: 4
```

NumCPU() 返回 CPU 核数量，NumGoroutine() 返回正在执行和排队的任务总数。你会发现，总是有几个 goroutine 在偷偷运行 (main 和 GC Heap 管理)。

```
func main() {
    println("start:", runtime.NumGoroutine())

    for i := 0; i < 10; i++ {
        go func(n int) {
            println(n, runtime.NumGoroutine())
        }(i)
    }

    time.Sleep(3 * time.Second)
    println("over:", runtime.NumGoroutine())
}
```

输出:

```
start: 2
0 13
1 12
2 11
3 10
4 9
5 8
6 7
7 6
8 5
9 4
over: 3
```

Goexit() 将终止当前 goroutine (终止整个调用堆栈链，在内层退出)，并确保 defer 函数被调用。

```
func main() {
    go func() {
        defer println("go1 defer...")

        for i := 0; i < 10; i++ {
            println("go1:", i)
            if i == 5 {
                runtime.Goexit()
            }
        }
    }()
```

```
    go func() {
        println("go2")
    }()

    time.Sleep(3 * time.Second)
}
```

输出:

```
go1: 0
go1: 1
go1: 2
go1: 3
go1: 4
go1: 5
go1 defer...
go2
```

第 7 章 程序结构

7.1 源文件

Go 编译器工具对项目目录结构特殊的约定。另外，建议使用 go fmt 格式化源码，以确保编码风格统一。详情参考第 9 章。

7.1.1 格式

编码: 源文件以 UTF-8 编码存储。
结束: 行尾的 ";" 多数情况下可以省略。
注释: 支持 /**/ 和 // 两种注释方式，不能嵌套。
命名: camelCasing 风格，不建议用下划线连接多个单词。

7.1.2 目录结构

Go 不再使用 Makefile，而是改用 "go <command>" 工具集。想要正确运行这些工具，首先得设置好相应的环境参数，而且项目目录结构也有硬性规定。

环境参数设置:

- **GOPATH**: 项目存放目录列表，import 通过该环境变量搜索库静态文件。
  - 必须设置，且不能和 GOROOT (Go 安装目录) 相同。
  - 通常将首个路径作为第三方库的安装目录。
  - 如不被其他项目引用，则无需将项目路径添加到该设置。但也会导致 go install 无法工作。

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

目录结构:

- **bin**: go install 编译安装的可执行文件保存目录。
- **pkg**: go install 编译安装的二进制静态包文件 (.a) 保存目录。
- **src**: 项目源码目录 (每个项目一个子目录)。

```
<projects>
   |_ bin
   |_ pkg
   |_ src
       |_ proj1
       |_ proj2
```

大的项目通常由多个包组成，我们可以创建下级子目录保存。

```
<projects>
   |_ bin
   |_ pkg
   |_ src
       |_ blog
            |_ main.go
            |_ engine
            |_ logic
```

7.2 包

Go 以 package 来组织代码，类似 Python package。

7.2.1 创建包

所有的代码都必须组织在 package 中。

- 在源码的第一行以 "package <name>" 声明包名称 (相当于 namespace)。
- 同一个包可以由多个 .go 源文件组成。
- 包名和其所在目录名可以不同。go build 默认以目录名作为静态包 (*.a) 主文件名。
- 在包内部 (包括多个源文件之间) 可以访问所有成员及其字段。
- 包内仅首字母大写的标识符可以被导出，相当于 public 权限。
- 通常包的全部文件 (源文件和测试文件) 都放在独立目录中，使用 "go install" 编译和安装。
- 可执行程序必须包含 package main，其内部有 main 入口函数。

有关包成员的访问权限设置，Go 采用了和 Python 类似的做法。Python 用下划线标识 private，Go 则用大写首字母来确定 public。这个规则适用于全局变量、全局常量、类型、结构字段、函数、方法等。

- 以无参数无返回值的 main.main() 函数作为程序入口函数；
- 使用 os.Exit(0) 返回终止进程；
- 用 os.Args 获取命令行启动参数。

注：os.Args[0] 返回的是命令行的解析结果，也就是说如果通过系统搜索路径 PATH 查找执行，那么就无法通过它来确定实际执行路径，可用 filepath.Abs(exec.LookPath(os.Args[0]))。

7.2.2 导入包

使用包前，必须使用 import 导入包静态文件路径。

```
        包路径 = 相对路径/静态包主文件名（不包含 .a 扩展名）
```

相对路径是指标准库路径和 GOPATH 环境参数所设置的路径列表。比如在我的电脑编译环境中，import "io/ioutil" 表示导入 "/usr/local/go/pkg/darwin_amd64/io/ioutil.a" 静态库。

包的库文件名和包名可能不同。导入用包静态库文件名 (filename.a)，而调用成员则必须用包名 (package name, namespace)。这和 .NET 差不多，dll 库文件和 namespace 并没有一致关系。

和 Python import 一样，支持多种导入方式。可以设定别名，或者导入包的全部 public 成员。

```
import   "lib/math"         // 正常模式: math.Sin
import M "lib/math"         // 别名模式: M.Sin。是对包设定别名，而不是导入路径。
import . "lib/math"         // 简便模式: Sin。相当于 python 的 from lib.math import *
import _ "lib/math"         // 丢垃圾桶: 这回不怕未使用包导致编译错误了，还可用来调用其初始化方法。
```

使用 "_" 导入的目的，主要是为了执行该包的 init 初始化函数。

对于多级目录结构的项目，我们还可以使用 local 导入方式。比如当前项目下有个 test 包。

```
package main

import (
    "./test"
)

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

但你会发现该模式仅在 "go run" 时有效，build 编译会导致如下错误。

```
$ go build
can't load package: main.go:4:2: local import "./test" in non-local package
```

7.2.3 包初始化

Go 支持初始化函数:

- 每个源文件都可以定义一个或多个初始化函数 func init() {}。
- 所有初始化函数都会在 main() 之前，在单一线程上被调用，仅执行一次。
- 编译器不能保证多个初始化函数的执行次序。
- 初始化函数在当前包所有全局变量初始化 (零或初始化表达式值) 完成后执行。
- 不能在程序代码中直接或间接调用初始化函数。

```
package main

import (
    "strings"
    "time"
)
```

```
func init() {
    println(a)                          // 在全局变量初始化完成后才会执行，否则初始化就有问题了……
    println("init over...")
}

var a string = strings.Join([]string{"a", "b"}, ",")

func init() {
    time.Sleep(2 * time.Second)      // main 必须等待初始化完成后才被执行，包括其他包中的初始化函数。
    println("init2 over...")
}

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

输出:

```
a,b
init over...
init2 over...
Hello, World!
```

可以在初始化函数中执行 goroutine，如果期望 init 等待 goroutine 结束，可以用 channel。

```
package main

import (
    "time"
)

func init() {
    go func() {
        println("goroutine in init...")
        time.Sleep(1 * time.Second)
    }()

    println("init over...")
}

func main() {
    println("Hello, World!")
    time.Sleep(2 * time.Second)             // 等待 goroutine 执行结束后才退出，否则看不到效果了。
}
```

输出:

```
init over...
Hello, World!
goroutine in init...
```

初始化函数只能用来完成初始化该做的活，搞太多事情不符合设计初衷……

7.2.4 包文档

godoc 会自动提取注释生成帮助信息，可以在命令行或 web 下查看。

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

Package:

- 建议用仅包含 package 语句的 doc.go 文件保存大篇幅帮助文档。
- 包文档第一整句作为 package list 说明。整句是指英文句号结尾，无法识别中文符号。

Bug:

- 非 *_test.go 源码中以 BUG 开始的注释，会在页面底部专门的 Bugs 节点显示。

  例如: // BUG(yuhen): memory leak.

Example:

- 通常建议将 Example 测试函数放到独立的 example_<name>_test.go 文件中。
- 成员示例: 将名称写成 Example<FuncOrType> 或 Example<Type>_<Method> 即可。
- 包示例: Example_<name>，name 首字母必须小写。
- 如果 *_test.go 仅有一个 Example 函数，且该函数调用了同文件中的其他成员，那么会显示整个文档，而不仅仅是测试函数自己。

```
func Example_demo() {                       // 包示例
    fmt.Println("demo")
    // Output: demo
}

func ExampleApplication(                    // 类型 Application 示例
    fmt.Println("application")
    // Output: application
}

func ExampleApplication_RunLoop() {         // 方法 Application.RunLoop() 示例
    fmt.Println("application.runloop")
    // Output: application.runloop
}
```

第 8 章 进阶

8.1 运行时

Go 编译生成的可执行文件都内嵌运行时 (runtime)，负责内存分配、垃圾回收、Goroutine 调度、类型管理，以及反射等工作。也正因为如此，使得其文件尺寸相较其他语言更大一些。但由于采用静态链接，无需附加文件 (比如某些动态链接库)，更利于分发。

8.2 内存分配

Go 在 Stack 和 GC Heap 上分配内存，无需手动释放内存。通常情况下，我们无需关心，或者说无法明确知道内存是分配在 Stack 还是 GC Heap 上。

- 简单规则：如果对象 "很大"，或者有获取对象地址的操作，那么就分配到 GC Heap 上。
- 官方文档：正在努力将尽可能多的对象放在 stack frame 上，以提高性能。

> **How do I know whether a variable is allocated on the heap or the stack?**
>
> From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.
>
> The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.
>
> In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

有关内存分配的详细信息，请参考附录的《源码阅读指南》。

除了直接定义变量外，还可用 new 和 make 来分配内存。

- new: 用于值类型，返回一个被初始化的内存块指针。
- make: 用于引用类型 slice、map、channel，分配内存并设置内部属性，返回对象。

当前三种引用类型，除了 slice 返回结构体外，map、channel 都直接返回指针。传递 slice 实参虽然不拷贝底层数组，但自身还是值拷贝传递的，好在最多也就 24 字节。

Go 内存分配器会预留 "巨大" 的 VA 地址空间，也就是说哪怕是最简单的 "Hello, World!" 入门演示进程，其虚拟内存占用也极恐怖。不过，这仅仅是个数字罢了，到不会真的用那么多内存。何况，也没几台机器有上百 GB 的内存 (64bit)。

http://tip.golang.org/doc/faq#Why_does_my_Go_process_use_so_much_virtual_memory

8.3 内存布局

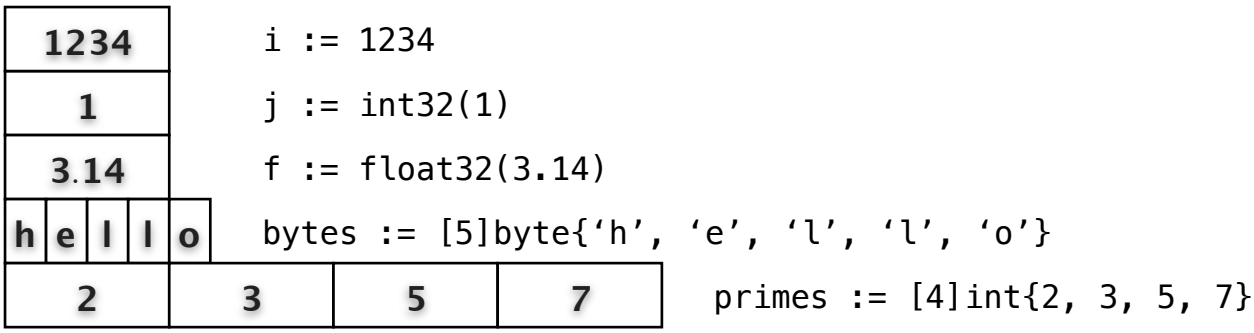
了解对象的内存布局，对于理解值传递、引用类型等概念有极大帮助。

参考资料:

- http://research.swtch.com/2009/11/go-data-structures.html
- http://www.softwareresearch.net/fileadmin/src/docs/teaching/SS10/Sem/Presentation_Aigner_Baumgartner.pdf
- src/pkg/runtime/runtime.h。

8.3.1 基本数据类型

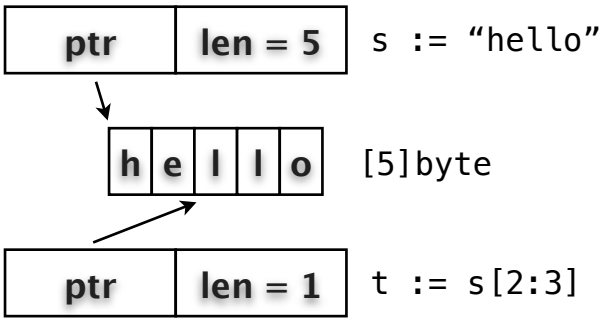
和 C 对应类型的内存结构基本相同。



注意：在不同平台，int 长度会有所区别，另外还有内存对齐行为。

8.3.2 字符串

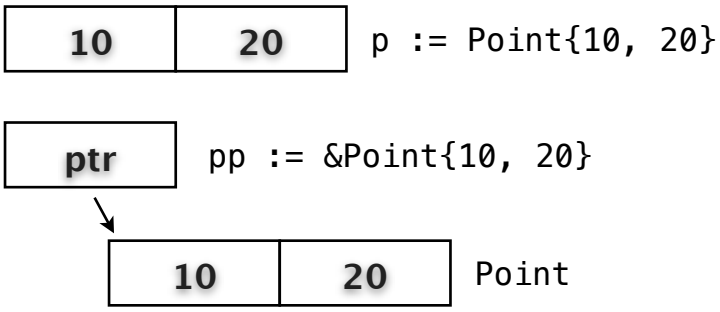
字符串对象存储的是指向一个 UTF-8 字节数组的指针，自带长度信息。 字节数组没有 NULL 结尾，切片操作返回的依然是 string 对象。



8.3.3 结构

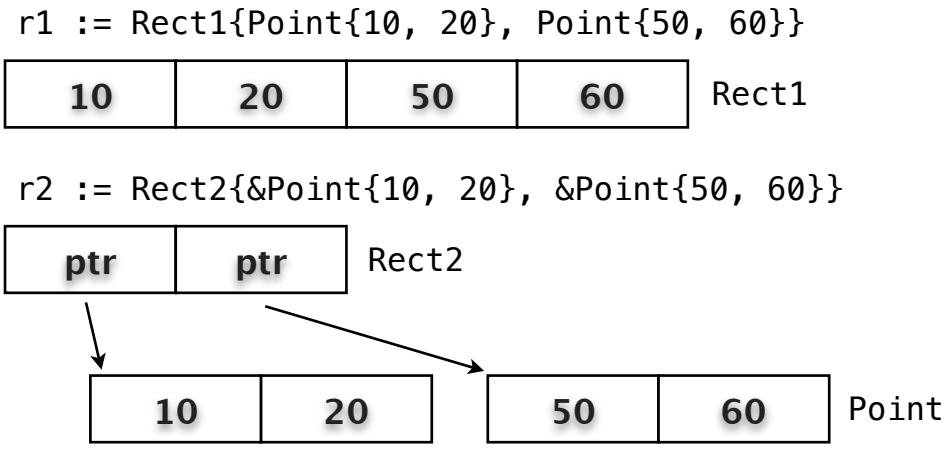
和 C struct 内存布局基本一致。并没有 Python/C# Object 对象头的那些引用计数、类型指针等附加字段。

```
type Point struct { X, Y int }
```



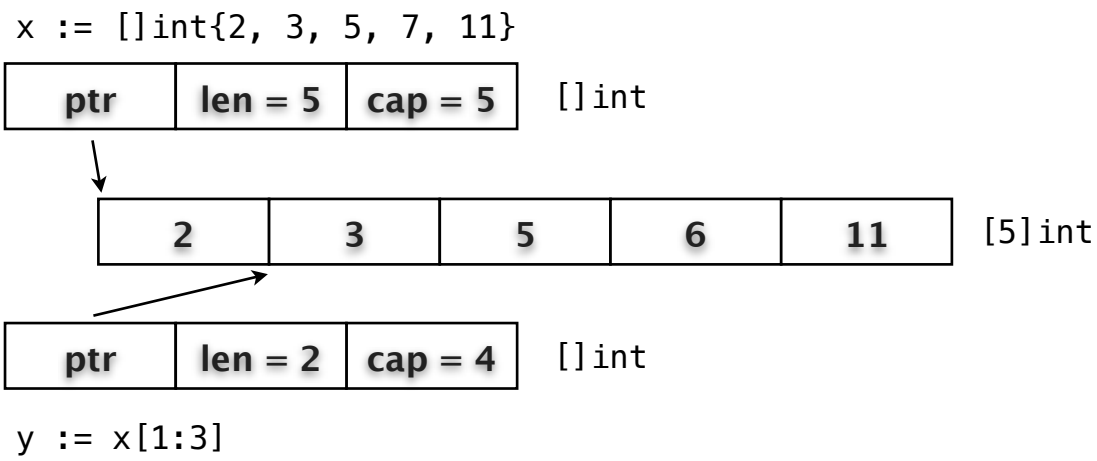
匿名嵌入的结构和原结构是一个整体。如果是嵌入匿名指针，那么存储的就是指针。

```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }
```



8.3.4 Slices

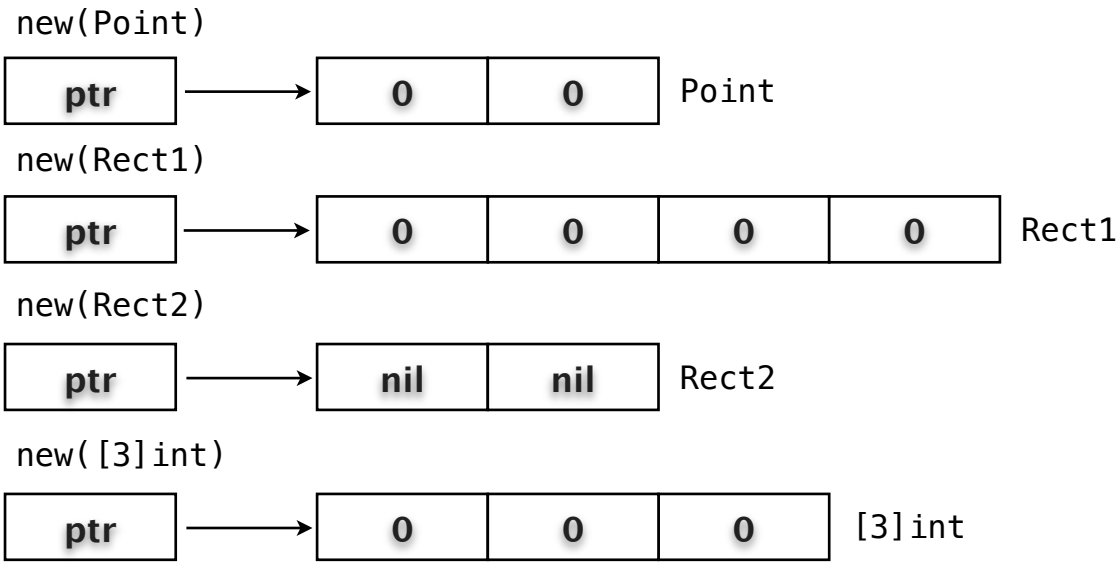
slice 内部指针指向某个底层数组的某个元素。下图示例中，x 和 y 指向同一个底层数组。



8.3.5 new

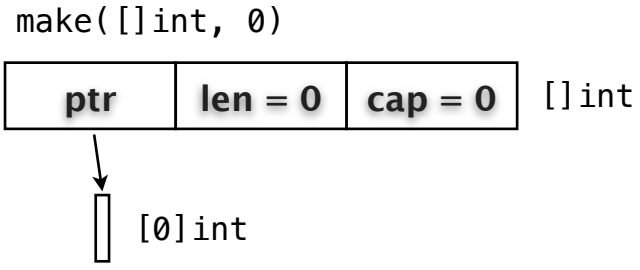
new 返回一段已经被初始化的 GC Heap 内存块指针。

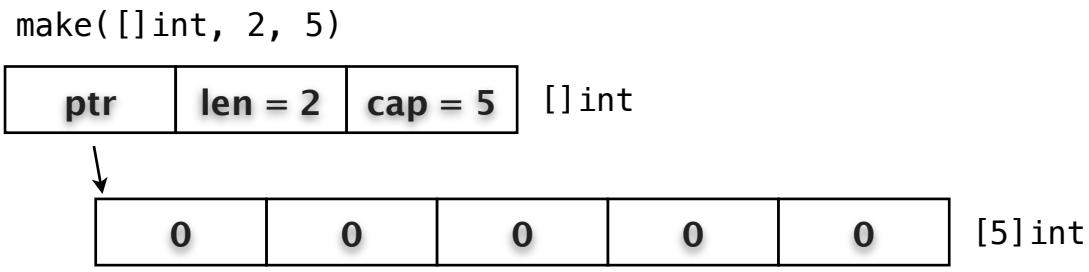
```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }
```



8.3.6 make

make 通常要分配引用和内容两块内存，还得设置一些附加属性。返回目标对象，而非指针。





8.4 反射

反射 (reflection) 可大大提高程序开发的灵活性，借助于反射让静态语言具备更加多样的运行期动态特征。反射让程序具备自省能力，使得 interface{} 有更大发挥余地。

反射使用 TypeOf 和 ValueOf 函数从 interface{} 对象获取实际目标对象的类型和值信息。

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

8.4.1 从接口反射获取 Type、Value

试着利用反射显示一个 "未知" 结构类型的字段、方法等信息。

```
type User struct {
    Id   int
    Name string
    x    float32
}

func (this User) String() {
    println("User:", this.Id, this.Name)
}

func main() {
    d := &User{1, "User1", 1.5}
    t := reflect.TypeOf(d)

    // 想要访问指针对象的 StructField，必须通过 Elem() 获得原始对象。
    if t.Kind() == reflect.Ptr {
        t = t.Elem()
    }

    // 字段，其中包括非导出字段。
    for i := 0; i < t.NumField(); i++ {
        f := t.Field(i)
        fmt.Printf("%-4s %s\n", f.Name, f.Type.Kind())
    }

    // 方法集。这个无需执行 Elem()，但影响方法集结果和签名。
    for i := 0; i < t.NumMethod(); i++ {
```

```
        m := t.Method(i)
        fmt.Println(m.Name, m.Type)
    }
}
```

输出:

```
Id   int
Name string
x    float32

String func(main.User)
```

可用 CanInterface() 判断能否取值，是否非导出字段。取值方法有具体的 Int()、Float()、String() 等，或用 Interface() 返回 interface{}。至于非导出字段，无法用这些方法，只能用指针绕道了。

```
func main() {
    d := &User{1, "User1", 1.5}
    v := reflect.ValueOf(d)

    if v.Kind() == reflect.Ptr {
        v = v.Elem()
    }

    // 取值
    printValue := func(name string) {
        f := v.FieldByName(name)

        if f.CanInterface() {
            switch f.Type().Kind() {
            case reflect.String:
                fmt.Println(f.String())
            default:
                fmt.Println(f.Interface())
            }
        } else if f.CanAddr() {                     // 非导出字段
            p := unsafe.Pointer(f.UnsafeAddr())     // 或用 f.Addr().Pointer()

            switch f.Type().Kind() {
            case reflect.Float32:
                fmt.Println(*(*float32)(p))
            }
        }
    }

    printValue("Id")
    printValue("Name")
    printValue("x")
}
```

输出:

```
1
User1
1.5
```

反射会将匿名嵌入字段当做一个独立字段 (本来就是如此)。如果要获取嵌入类型的字段，必须使用多级索引路径。

```
type User struct {
    Id   int
    Name string
}

type Manager struct {
    User
    title string
}

func main() {
    m := Manager{User{1, "Tom"}, "CXO"}
    t := reflect.TypeOf(m)

    // 序号 0 获取 Manager 的第一个字段，也就是 User。
    fmt.Printf("%+v\n", t.Field(0))

    // 匿名字段实际名字就是类型名。
    f, _ := t.FieldByName("User")
    fmt.Printf("%+v\n", f)

    // 使用 {0, 1} 二级字段索引序号，访问 Manager.User.Name。
    fmt.Printf("%+v\n", t.FieldByIndex([]int{0, 1}))
}
```

输出:

```
{Name:User PkgPath: Type:main.User Tag: Offset:0 Index:[0] Anonymous:true}
{Name:User PkgPath: Type:main.User Tag: Offset:0 Index:[0] Anonymous:true}
{Name:Name PkgPath: Type:string    Tag: Offset:8 Index:[1] Anonymous:false}
```

8.4.2 利用 Value 修改原对象

用反射修改对象的前提是 interface.data settable，也就说是 Pointer-Interface。但修改前要用 Elem() 取得指针指向的实际对象。

用 Kind() 来判断是否是 Pointer，如果不是 Ptr，调用 Elem() 会出错。 还需判断 CanSet() 为真。

```
type User struct {
    Id   int
    Name string
    x    float32
}

func main() {
    d := &User{1, "User1", 1.5}

    v := reflect.ValueOf(d)
    if v.Kind() == reflect.Ptr {
        v = v.Elem()
```

```
    }

    if v.CanSet() {
        // 直接方法设置。
        v.FieldByName("Id").SetInt(2)
        v.FieldByName("Name").SetString("User2")

        // 非导出字段只能用指针赋值。
        p := (*float32)(unsafe.Pointer(v.FieldByName("x").Addr().Pointer()))
        *p = 2.7

        fmt.Println(d)
    }
}
```

输出:

```
&{2 User2 2.7}
```

基本类型操作比较简单。

```
func main() {
    d := 123
    v := reflect.ValueOf(&d)

    if v.Kind() == reflect.Ptr && v.Elem().CanSet() {
        v.Elem().SetInt(456)
    }

    fmt.Println(d)
}
```

输出:

```
456
```

注意 Value 的很多方法和 Type 不同，它并没有遵循 "comma ok" 惯例，而是在失败的时候返回一个 zero Value，比如:

src/pkg/reflect/value.go

```
774    func (v Value) FieldByName(name string) Value {
775            v.mustBe(Struct)
776            if f, ok := v.typ.FieldByName(name); ok {
777                    return v.FieldByIndex(f.Index)
778            }
779            return Value{}
780    }
```

可以用 IsValid() 或 Kind() == Invalid 来验证。

```
v := reflect.Value{}
fmt.Println(v.IsValid(), v.Kind() == reflect.Invalid)
```

输出:

```
false, true
```

这种设计让人很无语，就算每个 Value 结构最多也就 24 字节 (64位)，但也没必要这么闹腾吧？何况 IsValid、Kind 这些方法都是都是 value method，完全寄希望于编译器的内联优化?

8.4.3 调用方法

用反射可以 "动态" 调用方法，就是处理参数稍微麻烦了一些。

```
type User struct {
    Id   int
    Name string
}

func (this *User) Test(x int, s string) string {
    return fmt.Sprintf("user.name: %s; x: %d; s: %s", this.Name, x, s)
}

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

    m := reflect.ValueOf(o).MethodByName("Test")
    args := []reflect.Value{reflect.ValueOf(1234), reflect.ValueOf("Hello")}

    rets := m.Call(args)

    for _, r := range rets {
        switch r.Kind() {
        case reflect.String:
            fmt.Println(r.String())
        }
    }
}
```

输出:

```
user.name: Tom; x: 1234; s: Hello
```

Type 的 NumIn、NumOut、In、Out、 IsVariadic 等方法可获取方法参数和返回值信息。

8.5 数据竞争

Go 1.1 提供了 "-race" 编译参数，让编译器启用数据竞争 (data race) 探测功能，以便及早发现并行代码潜在问题。

编译器会记录所有内存访问代码，如发现非同步共享变量访问 (例如，两个或两个以上的 goroutine 同时访问某个对象，且有写操作)，则输出警告信息。这种竞争探测是运行期行为，不能指望编译时给出提示。通常仅建议在调试或试用版本中使用该功能，以避免额外的性能损失。

下面这段代码中，多个 goroutine 会竞争修改变量 x。让我们看看实际效果如何。

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    x := 10
    f := func() {
        for i := 0; i < 5; i++ {
            x += 1
            println(x)
        }
    }

    for i := 0; i < 2; i++ {
        go f()
    }

    time.Sleep(time.Second * 2)
}
```

输出:

```
$ go build -race && ./test

11
12
13
14
15
==================
WARNING: DATA RACE
Write by goroutine 5:
  main.func·001()
      test/main.go:14 +0x4a
  gosched0()
      /usr/local/go/src/pkg/runtime/proc.c:1218 +0x9f

Previous write by goroutine 4:
  main.func·001()
      test/main.go:14 +0x4a
  gosched0()
      /usr/local/go/src/pkg/runtime/proc.c:1218 +0x9f

Goroutine 5 (running) created at:
  main.main()
      test/main.go:20 +0xcc
  runtime.main()
      /usr/local/go/src/pkg/runtime/proc.c:182 +0x91

Goroutine 4 (finished) created at:
  main.main()
      test/main.go:20 +0xcc
  runtime.main()
      /usr/local/go/src/pkg/runtime/proc.c:182 +0x91
==================
16
17
```

```
18
19
20
Found 1 data race(s)
```

输出结果给出了详细的竞争代码位置，极大方便我们修正代码。看看加锁能不能解决问题。

```
func main() {
    ...

    x := 10
    m := new(sync.Mutex)

    f := func() {
        m.Lock()
        defer m.Unlock()

        for i := 0; i < 3; i++ {
            x += 1
            println(x)
        }
    }

    ...
}
```

输出:

```
$ go build -race && ./test
11
12
13
14
15
16
```

效果不错。还可以用环境变量 GORACE 设置相关输出参数。

- log_path: 记录文件名，自动附加 pid 后缀，默认输出到 stderr。
- exitcode: 发现竞争时进程退出状态码，默认 66。
- strip_path_prefix: 移除记录中的路径前缀，更便于阅读。
- history_size: 调整 goroutine 内存访问记录大小 (32K * 2 ** history_size)，默认 1。

```
$ go build -race

$ GORACE="log_path=./race.log exitcode=100" ./learn
11
12
13
14
15
16
17
```

```
18
19
20

$ echo $?
100

$ cat race.log.24304
==================
WARNING: DATA RACE
Write by goroutine 5:
  main.func·001()
      test/main.go:14 +0x4a
  gosched0()
      /usr/local/go/src/pkg/runtime/proc.c:1218 +0x9f

Previous write by goroutine 4:
  main.func·001()
      test/main.go:14 +0x4a
  gosched0()
      /usr/local/go/src/pkg/runtime/proc.c:1218 +0x9f

Goroutine 5 (running) created at:
  main.main()
      test/main.go:20 +0xcc
  runtime.main()
      /usr/local/go/src/pkg/runtime/proc.c:182 +0x91

Goroutine 4 (running) created at:
  main.main()
      test/main.go:20 +0xcc
  runtime.main()
      /usr/local/go/src/pkg/runtime/proc.c:182 +0x91
==================
Found 1 data race(s)
```

http://golang.org/doc/articles/race_detector.html

8.6 cgo

cgo 让 Go 可以非常简单地融合 C 代码。

C	GO	说明
char	C.char	
signed char	C.schar	
unsigned char	C.uchar	

C	GO	说明
short	C.short	
unsigned short	C.ushort	
int	C.int	注意 C int 在 64 位平台也可能是 4 字节。
unsigned int	C.uint	
long	C.long	
unsigned long	C.ulong	
long long	C.longlong	
unsigned long long	C.ulonglong	
float	C.float	
double	C.double	
void*	unsafe.Pointer	
char*	*C.char	相应的指针写法要注意。
size_t	C.size_t	
NULL	nil	空指针。

建议将 C 代码写在独立的 .c、.h 文件中，然后在 cgo 注释块中 #include，这更利于编写和调试。当然，简单的函数直接写也没问题。

注意 cgo 注释块和 import "C" 之间不能有空行。还可设置 CFLAGS、LDFLAGS 等 gcc 编译参数。

```
/*
#cgo CFLAGS: -O3 -std=gnu99
#cgo LDFLAGS: -lpthread

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

void print_thread() {
    printf("thread: %lu\n", (unsigned long)pthread_self());
}
*/
import "C"                                  // 与上面 C 代码块之间不能有空行，否则出错。

import (
```

```
    "unsafe"
)

func PrintThreadId() {
    C.print_thread()
}

func Test(s string) {
    cs := C.CString(s)                      // 转换后类型: *C.char。
    defer C.free(unsafe.Pointer(cs))        // Go 是强类型，因此 *C.char 要转换成 Pointer。

    C.test(cs)
}
```

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

void test(char *s);

#endif
```

test.c

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

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

C.CString() 使用 C.malloc() 在 C heap 分配内存空间，因此必须调用 C.free() 释放。

```
/*
#cgo CFLAGS: -std=gnu99
#include <stdio.h>
#include <stdlib.h>
*/
import "C"

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

    cs := C.CString(s)                                          // Go string to *C.char
    defer C.free(unsafe.Pointer(cs))                            // #include <stdlib.h>
    fmt.Printf("%p, %p\n", &s, unsafe.Pointer(cs))

    fmt.Printf("%s\n", C.GoString(cs))                          // *C.char to Go string
    fmt.Printf("%s\n", C.GoStringN(cs, 5))                      // 指定长度
    fmt.Printf("%v\n", C.GoBytes(unsafe.Pointer(cs), 5))        // *C.char to []byte
}
```

输出:

```
0xc200037150, 0x42000e0                                          // 显然 cs 不在 mmap 区域
Hello, World!
Hello
[72 101 108 108 111]
```

也可以透过指针转换使用 C malloc/calloc 分配 C heap 内存，但得记得释放。

```
package main

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

import (
    "fmt"
    "unsafe"
)

func main() {
    var nobj, size C.size_t = 10, (C.size_t)(unsafe.Sizeof(0))

    p := C.calloc(nobj, size)                    // void* 对应 unsafe.Pointer。
    defer C.free(p)                              // 必须 free。

    a := (*[10]int)(p)                           // 转换成 Go 数组指针。
    a[5] = 10                                    // Go 里面直接用指针没压力。
    a[7] = 35

    fmt.Println(*a)
}
```

输出:

```
[0 0 0 0 0 10 0 35 0 0]
```

建议用 typedef 定义 C struct 类型标识符，否则 struct data {...} 就必须写成 C.struct_data。

```
/*
#cgo CFLAGS: -std=gnu99
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char* s;
    int x;
} data;

data* test()
{
    data *d = malloc(sizeof(data));                  // 注意别直接在 C 里面返回栈局部变量。
    d->s = "Hello, World!";
    d->x = 100;

    return d;
```

```
}

void test2(data d)
{
    printf("%s, %d\n", d.s, d.x);
}
*/
import "C"

func main() {
    var d *C.data = C.test()
    defer C.free(unsafe.Pointer(d))
    fmt.Printf("%v, %v\n", C.GoString(d.s), d.x)

    d2 := C.data{s: nil, x: 1234}                          // s 是 *C.char 类型。
    C.test2(d2)
}
```

输出:

```
Hello, World!, 100
(null), 1234
```

C enum 使用没啥问题。可 union 无法访问成员，只能当 []byte 用。

```
/*
#cgo CFLAGS: -std=gnu99
#include <stdio.h>
#include <stdlib.h>

enum mode { A = 1, B = 2, C };

typedef union
{
    int iv;
    char* sv;
} data;

void test(enum mode m, data *d)
{
    switch (m) {
        case A: printf("iv = %02x\n", d->iv); break;
        case B: printf("is = %s\n", d->sv); break;
        default: printf("unknown\n"); break;
    }
}
*/
import "C"

func main() {
    var m C.enum_mode = C.A

    d := C.data{}
    fmt.Printf("%T, %#v\n", d, d)
```

```
    d[0] = 1
    d[1] = 2
    d[2] = 3

    C.test(m, &d)
}
```

输出:

```
main._Ctype_data, main._Ctype_data{0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
iv = 30201
```

导出 Go func 给 C 调用，首先得使用 "//export" 标记导出同名函数，其次建议在独立文件中进行声明和调用，否则容易引发 "duplicate symbol" 错误。

main.go

```
/*
#cgo CFLAGS: -std=gnu99
#include <stdio.h>
#include <stdlib.h>
#include "test.h"
*/
import "C"

//export hello
func hello() {
    fmt.Println("Hello, Golang!")
}

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

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

extern void hello();                        // 函数声明，以便 C 调用。
void test();

#endif
```

test.c

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

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

在 C 源码中使用 cgo 类型，必须添加 #include "_cgo_export.h"。

由于 cgo 仅扫描当前目录，如果要包含其他 C 项目，要么将其源文件拷贝到当前目录，要么新建一个 C 源文件将它们全部包含进来 (#inlcude xxx.h; #include xxx.c ...)。不过注意设置 CFLAGS 里面的 "std=gnu99 -I./src/" 等参数。(某些编译错误可能是因为 std=c99 造成的)

例如: https://github.com/gosexy/redis/blob/master/main_c.c

第 9 章 工具

9.1 命令行工具

(1) go build: 编译包

在临时目录下创建包编译后的二进制文件。该命令不会将二进制文件安装到 bin、pkg 目录，但会将包含 main 入口函数的可执行文件拷贝到 src 目录。

go build 会检查并编译所有依赖包。

参数:

- -gcflags: 传递给 6g 编译器的参数。
- -ldflags: 传递给 6l 链接器的参数。
- -work: 查看临时目录。通常在编译完成后被删除。
- -n: 查看但不执行编译命令。
- -x: 查看并执行编译命令。
- -a: 强制编译所有依赖包。
- -v: 查看被编译的包名 (包括依赖包)。
- -p: 并行编译所使用的 CPU 核数量，默认是全部。
- -race: 启用数据竞争检测，适合 Debug 版本使用。

gcflags:

- -N: 禁用优化。
- -l: 禁用内联。
- -m: 输出优化策略信息。
- -B: 禁用边界检查，可以在一定程度提升执行性能。
- -u: 禁用 unsafe 包。

ldflags:

- -s: 删除符号表。
- -w: 删除 DRAWF 调试信息 (不包括符号表)。
- -X: 修改全局字符串变量初始值。如: -X main.x 'abc' -X main.VERSION '0.9.1'
- -H windowsgui: 以 MS Windows GUI 方式编译，可隐藏 Console 窗口。

更多参数可以查看 go tool 6g -h、go tool 6l -h 或 https://golang.org/cmd/ld/。

附: 除自带的 gc 编译器外，还可以选择 gccgo。其汇编质量远远高出 gc，当然在 goroutine 的支持上还是 gc 的策略好一些。编译时必须使用静态链接，否则导致失败。(在 Ubuntu 环境下，可以直接用 apt-get 安装 gccgo)

```
$ go build -compiler gccgo

/usr/bin/ld: cannot find -lgcc_s
collect2: error: ld returned 1 exit status

$ go build -compiler gccgo -gccgoflags '-static-libgcc -O3'       # O3 优化
```

(2) go install: 编译并安装包

和 go build 参数相同，唯一的区别在于将编译结果拷贝到 bin、pkg 目录中。
建议：不要设置 GOBIN 环境变量，否则会优先将编译结果安装到该变量所指定的目录。

(3) go clean: 垃圾清理

清除当前包目录下的编译临时文件。

参数:

- -n: 查看但不执行清理命令。
- -x: 查看并执行清理命令。
- -i: 同时删除 bin、pkg 目录下 go install 安装的二进制文件。
- -r: 同时清理所有依赖包 src 目录下的临时文件。

(4) go get: 下载并安装扩展包

从网络下载并安装第三方扩展包。必须设定 GOPATH 环境变量，且不能和 GOROOT 相同。默认将获取的包文件存储到 GOPATH 第一个有效路径。

参数:

- -d: 下载源码后，并不执行安装命令。
- -u: 检查并下载源码更新文件。
- -x: 查看并执行命令。

go get 支持 go install 的相关参数。

(5) go list: 查看包信息

查看包名、路径、依赖项等等信息。

参数:

- -json: 使用 json 格式输出包的相关信息，包括下面的依赖和导入。
- -f {{.Deps}}: 查看依赖包，包括直接或间接依赖。
- -f {{.Imports}}: 查看导入的包。

(6) gofmt: 调整源码格式

依照官方格式规范格式化源码文件。

参数:

- -w: 格式化结果写回原文件。
- -tabs=false: 不使用 TAB 缩进。
- -tabwidth=4: 缩进长度。
- -comments=false: 去掉注释。慎用，会连 doc 一同去掉。

示例:

```
$ gofmt -tabs=false -tabwidth=4 -w *.go
```

(7) go env: 查看 Go 环境变量

```
$ go env

GOROOT="/usr/local/go"
GOBIN=""
GOARCH="amd64"
GOCHAR="6"
GOOS="darwin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOTOOLDIR="/usr/local/go/pkg/tool/darwin_amd64"
GOGCCFLAGS="-g -O2 -fPIC -m64 -pthread -fno-common"
CGO_ENABLED="1"
```

(8) godoc: 查看包或成员帮助信息

查看某个具体包成员帮助信息。

```
$ godoc fmt Printf

package fmt
    import "fmt"

FUNCTIONS

func Printf(format string, a ...interface{}) (n int, err error)
    Printf formats according to a format specifier and writes to standard
    output. It returns the number of bytes written and any write error
    encountered.
```

查看源码

```
$ godoc -src fmt Printf

// Printf formats according to a format specifier and writes to standard output.
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

启动一个 WebServer，查看官方文档。

```
$ godoc -http=:6060
```

(9) go fix: 修复源码

当新版本的 Go 发布后，可以用该命令修改因为语言规范变更造成的语法错误。

(10) go vet: 源码检查

比如检查源文码中 Printf 等语句的 format 和 args 匹配情况。

```
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("%d, %s\n", "abc")
}
```

输出:

```
$ go vet main.go
main.go:8:2: wrong number of args in Printf call: 2 needed but 1 args
go tool vet: exit status 1
```

9.2 GDB 调试

默认情况下，编译过的二进制文件已经包含了 DWARFv3 调试信息，只要 GDB 7.1 以上版本都可以进行调试。 在 OSX 下，如无法执行调试指令，可尝试用 sudo 方式执行 gdb。

编译参数:

- 不能使用 ldflags "-s -w" 删除调试信息。
- 使用 gcflags "-N -l" 关闭内联优化。

相关函数:

- runtime.Breakpoint()：触发调试器断点。
- runtime/debug.PrintStack()：显示调用堆栈。
- log：适合替代 print 显示调试信息。

GDB 支持：

- source pkg/runtime/runtime-gdb.py。

演示：

```
package main

import (
    "fmt"
    "runtime"
)

func test(s string, x int) (r string) {
    r = fmt.Sprintf("test: %s %d", s, x)
    runtime.Breakpoint()
    return r
}

func main() {
    s := "haha"
    i := 1234
    println(test(s, i))
}
```

```
$ go build -gcflags "-N -l"          // 编译，关闭内联优化。

$ sudo gdb demo                      // 启动 gdb 调试器，手工载入 Go Runtime。
GNU gdb (GDB) 7.5.1
Reading symbols from demo...done.

(gdb) source /usr/local/go/src/pkg/runtime/runtime-gdb.py
Loading Go Runtime support.

(gdb) l main.main                    // 以 <package>.<member> 方式查看源码。
14      func main() {
15          s := "haha"
16          i := 1234
17          println(test(s, i))
18      }

(gdb) l main.go:8                    // 以 <filename.go>:<line> 方式查看源码。
8       func test(s string, x int) (r string) {
9           r = fmt.Sprintf("test: %s %d", s, x)
10          runtime.Breakpoint()
11          return r
12      }
```

```
(gdb) b main.main                    // 以 <package>.<member> 方式设置断点。
Breakpoint 1 at 0x2131: file main.go, line 14.

(gdb) b main.go:17                   // 以 <filename.go>:<line> 方式设置断点。
Breakpoint 2 at 0x2167: file main.go, line 17.

(gdb) info breakpoints               // 查看所有断点。
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000002131 in main.main at main.go:14
2       breakpoint     keep y   0x0000000000002167 in main.main at main.go:17

(gdb) r                              // 启动进程，触发第一个断点。
Starting program: demo
[New Thread 0x1c03 of process 4088]
[Switching to Thread 0x1c03 of process 4088]

Breakpoint 1, main.main () at main.go:14
14      func main() {

(gdb) info goroutines                // 查看 goroutines 信息。
* 1  running runtime.gosched
* 2  syscall runtime.entersyscall

(gdb) goroutine 1 bt                 // 查看指定序号的 goroutine 调用堆栈。
#0  0x000000000000f6c0 in runtime.gosched () at pkg/runtime/proc.c:927
#1  0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244
#2  0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267
#3  0x0000000000000000 in ?? ()

(gdb) goroutine 2 bt                 // 这个是 GC goroutine。
#0  runtime.entersyscall () at pkg/runtime/proc.c:989
#1  0x000000000000d01d in runtime.MHeap_Scavenger () at pkg/runtime/mheap.c:363
#2  0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267
#3  0x0000000000000000 in ?? ()

(gdb) c                              // 继续执行，触发下一个断点。
Continuing.
Breakpoint 2, main.main () at main.go:17
17          println(test(s, i))

(gdb) info goroutines                // 当前 goroutine 序号为 1。
* 1  running runtime.gosched
  2 runnable runtime.gosched

(gdb) goroutine 1 bt                 // 当前 goroutine 调用堆栈。
#0  0x000000000000f6c0 in runtime.gosched () at pkg/runtime/proc.c:927
#1  0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244
#2  0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267
#3  0x0000000000000000 in ?? ()

(gdb) bt                             // 查看当前调用堆栈，可以与当前 goroutine 调用堆栈对比。
#0  main.main () at main.go:17
#1  0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244
#2  0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267
```

```
#3  0x0000000000000000 in ?? ()

(gdb) info frame                    // 堆栈帧信息。
Stack level 0, frame at 0x442139f88:
 rip = 0x2167 in main.main (main.go:17); saved rip 0xe44c
 called by frame at 0x442139fb8
 source language go.
 Arglist at 0x442139f28, args:
 Locals at 0x442139f28, Previous frame's sp is 0x442139f88
 Saved registers:
  rip at 0x442139f80

(gdb) info locals                   // 查看局部变量。
i = 1234
s = "haha"

(gdb) p s                           // 以 Pretty-Print 方式查看变量。
$1 = "haha"

(gdb) p $len(s)                     // 获取对象长度 ($cap)
$2 = 4

(gdb) whatis i                      // 查看对象类型。
type = int

(gdb) c                             // 继续执行，触发 breakpoint() 断点。
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
runtime.breakpoint () at pkg/runtime/asm_amd64.s:81
81          RET

(gdb) n                             // 从 breakpoint() 中出来，执行源码下一行代码。
main.test (s="haha", x=1234, r="test: haha 1234") at main.go:11
11          return r

(gdb) info args                     // 从参数信息中，我们可以看到命名返回参数的值。
s = "haha"
x = 1234
r = "test: haha 1234"

(gdb) x/3xw &r                      // 查看 r 内存数据。(指针 8 + 长度 4)
0x442139f48:   0x42121240     0x00000000     0x0000000f

(gdb) x/15xb 0x42121240             // 查看字符串字节数组
0x42121240:    0x74    0x65    0x73    0x74    0x3a    0x20    0x68    0x61
0x42121248:    0x68    0x61    0x20    0x31    0x32    0x33    0x34

(gdb) c                             // 继续执行，进程结束。
Continuing.
test: haha 1234
[Inferior 1 (process 4088) exited normally]

(gdb) q                             // 退出 GDB。
```

用 dir 命令添加系统库源文件路径，就可以设置相应的源码行号断点。

```
(gdb) dir /usr/local/go/src/pkg/fmt
Source directories searched: /usr/local/go/src/pkg/fmt:$cdir:$cwd

(gdb) b print.go:289                // 如果不使用 dir 添加搜索路径，则必须用全路径。
Breakpoint 1 at 0x1fc49: file /usr/local/go/src/pkg/fmt/print.go, line 289.
```

当然也可以直接用符号名。

```
(gdb) b fmt.Println
Breakpoint 1 at 0x1fc20: file /usr/local/go/src/pkg/fmt/print.go, line 288.
```

创建 .gdbinit 文件，减少输入。

```
$ cat ~/.gdbinit

define goruntime
    source /usr/local/go/src/pkg/runtime/runtime-gdb.py
end
```

更多细节，请参考: http://golang.org/doc/gdb

9.3 条件编译

运行时可通过 runtime 包的相关环境常量进行判断。

- GOOS: windows, darwin, linux, freebsd
- GOARCH: 386, amd64, arm

还可以在源文件头部 (.go, .h, .c, .s 等) 添加编译约束标记，提示编译器检查相关编译变量以确定是否处理该文件，类似 C 编程中的 #if...#else... 指令。

约束标记 "+build" 必须放在源文件头部，用空行与后面的文档和代码分隔开来。 允许有多行标记，用空格、逗号和感叹号来表示 OR、AND、NOT 关系。

例如下面的编译条件：(darwin OR linux) AND (amd64 AND !cgo)

```
// +build darwin linux
// +build amd64,!cgo

package main

import (
    "fmt"
    "go/build"
)
```

```
func main() {
    fmt.Println(build.Default)       // 输出相关编译信息。
}
```

如编译目标平台不符合条件，或者没有关闭 CGO_ENABLED，那么将不处理该文件。

```
$ go build -n
6g ... -I $WORK ./test.go

$ CGO_ENABLED=0 go build -n
6g ... -I $WORK ./main.go ./test.go
```

标记作用于整个文件，不能把不同版本代码放在同一文件中。推荐做法是拆分成多个源文件，改用特定文件名后缀 *_GOOS_GOARCH、*_GOOS、*_GOARCH 代替标记语句。在 Go 源码目录 go/src/pkg/runtime 中，你会发现很多这样的文件 (对所有源码文件有效)。

```
$ ls -l os_*

-rw-r--r--  1 yuhen  admin  14163  3 29 18:23 os_darwin.c
-rw-r--r--  1 yuhen  admin   1353  3 29 18:23 os_darwin.h
-rw-r--r--  1 yuhen  admin   7679  3 29 18:23 os_freebsd.c
-rw-r--r--  1 yuhen  admin    762  3 29 18:23 os_freebsd.h
-rw-r--r--  1 yuhen  admin    603  3 29 18:23 os_freebsd_arm.c
-rw-r--r--  1 yuhen  admin   9478  3 29 18:23 os_linux.c
-rw-r--r--  1 yuhen  admin   1067  3 29 18:23 os_linux.h
-rw-r--r--  1 yuhen  admin    820  3 29 18:23 os_linux_386.c
-rw-r--r--  1 yuhen  admin   2410  3 29 18:23 os_linux_arm.c
-rw-r--r--  1 yuhen  admin   8829  3 29 18:23 os_netbsd.c
-rw-r--r--  1 yuhen  admin    759  3 29 18:23 os_netbsd.h
-rw-r--r--  1 yuhen  admin    529  3 29 18:23 os_netbsd_386.c
-rw-r--r--  1 yuhen  admin    581  3 29 18:23 os_netbsd_amd64.c
-rw-r--r--  1 yuhen  admin    924  3 29 18:23 os_netbsd_arm.c
-rw-r--r--  1 yuhen  admin   7943  3 29 18:23 os_openbsd.c
-rw-r--r--  1 yuhen  admin    631  3 29 18:23 os_openbsd.h
-rw-r--r--  1 yuhen  admin   6050  3 29 18:23 os_plan9.c
-rw-r--r--  1 yuhen  admin   2128  3 29 18:23 os_plan9.h
-rw-r--r--  1 yuhen  admin   2728  3 29 18:23 os_plan9_386.c
-rw-r--r--  1 yuhen  admin   3051  3 29 18:23 os_plan9_amd64.c
-rw-r--r--  1 yuhen  admin  12370  3 29 18:23 os_windows.c
-rw-r--r--  1 yuhen  admin   1152  3 29 18:23 os_windows.h
-rw-r--r--  1 yuhen  admin   2766  3 29 18:23 os_windows_386.c
-rw-r--r--  1 yuhen  admin   3072  3 29 18:23 os_windows_amd64.c
```

如果要忽略某个文件，可以用 ignore。

```
// +build ignore
```

至于自定义条件，需显式通过 go build -tags 参数传递。比如下面通过传递 debug、beta 两个条件参数来控制编译版本。

```
// +build debug,beta
```

```
package main

func init() {
    println("test init.")
}
```

仅传递必要的条件。如果传递 NOT 参数，要对感叹号进行转义。

```
$ go build -tags "debug beta" && ./test
test init.

$ go build -tags "debug \!beta"
```

9.4 跨平台编译

对苦逼的码农而言，在同一个开发环境编译出不同平台所需的二进制可执行文件是非常必要的。

(1) 首先进入 go/src 源码所在目录，执行如下命令创建目标平台所需的包和工具文件。

```
$ cd /usr/local/go/src
$ GOOS=linux GOARCH=amd64 ./make.bash --no-clean
```

如果是 Windows 则修改 GOOS 即可。参数 no-clean 避免清除其他平台文件，还能提高速度。

```
$ GOOS=windows GOARCH=amd64 ./make.bash --no-clean
```

(2) 现在可以编译 Linux 和 Windows 平台所需的执行文件了。

```
$ GOOS=linux GOARCH=amd64 go build
$ GOOS=windows GOARCH=amd64 go build
```

不过该方式暂时不支持 CGO，默认 CGO_ENABLED=0。

http://solovyov.net/en/2012/03/09/cross-compiling-go/

7.5 程序测试

Go 自带了测试框架和工具，很容易完成单元测试和性能测试。

testing 中提供了 T 类型用于单元测试，B 类型用于性能测试。测试代码被放在 *_test.go 文件中，通常和被测试代码属于同一个包，以便访问私有成员。

7.5.1 Test

在测试源文件中，测试函数名称必须是 "Test[^a-z]" 这样的格式。以 "Test" 开头，后面跟上非小写字母开头的字符串。就是用大写字母、下划线或者数字什么的确定前面这个 Test 是个独立单词，好

让 go test 能知道这是个测试函数。像 "Testabc" 这样的，估计丫搞不清是一个测试函数，还是一个普通函数。

测试函数接收一个 *testing.T 类型参数，用于输出信息或中断测试。

main_test.go

```
package main

import "testing"

func TestSum(t *testing.T) {
    if sum(1, 2, 3) != 16 {
        t.Fatal("我就是有意搞错滴，你想怎么地!")
    }
}
```

测试方法:

- Fail: 标记失败，但继续执行当前测试函数。
- FailNow: 失败，立即终止当前测试函数执行。
- Log: 输出错误信息。
- Error: Fail + Log。
- Fatal: FailNow + Log。
- Skip: 跳过当前函数。通常用于未完成的测试用例。

运行 "go test" 开始测试，它会自动搜集所有的测试源文件 (_test.go)，提取全部测试函数。

```
$ go test

--- FAIL: TestSum (0.00 seconds)
       mypkg_test.go:9: 我就是有意搞错滴，你想怎么地!
FAIL
exit status 1
FAIL    mypkg   0.025s
```

参数:

- -v: 显示所有测试函数运行细节。
- -run regex: 指定要执行的的测试函数。

还可以是用 cpuprofile、memprofile 参数生成 pprof 所需结果文件。更多参数信息可参考 "go help testflag"。

7.5.2 Benchmark

性能 (Benchmark) 也是及重要的测试项目。函数以 Benchmark 开头，参数类型是 *testing.B 。

参数 b.N 是测试循环次数，以获得足够的时间基数。为什么不是循环调用 N 次 BenchmarkSum 呢？可能是为了避免循环执行测试函数本身造成过多的性能消耗。

```
func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        sum(1, 2, 3)
    }
}
```

默认情况下，go test 不执行 Benchmark 测试，必须用 "-bench <pattern>" 指定性能测试的函数。Test 和 Benchmark 函数可以放在同一源文件中。

main_test.go

```
package main

import "testing"

func TestSum(t *testing.T) {
    if sum(1, 2, 3) != 16 {
        println("jaha")
        //t.Fatal("我就是有意搞错滴，你想怎么地!")
    }
}

func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        sum(1, 2, 3)
    }
}
```

```
$ go test -v -bench "."

=== RUN  main.TestSum
jaha
--- PASS: main.TestSum (0.00 seconds)
PASS

main.BenchmarkSum     20000000              134 ns/op
```

其他参数:

- benchmem: 输出内存分配统计。
- benchtime: 指定测试时间。
- cpu: 指定 GOMAXPROCS 值。
- timeout: 超时限制。

除了拥有和 T 类似的功能外， B 还有一些和上述参数对应的方法。

7.5.3 Example

Example Test 类似 Python doctest，通过匹配 stdout 输出结果来判断测试成功或失败。在 Example 函数尾部用 "// Output: " 来标注正确输出结果，如果没有则不会执行。

注意：必须用 fmt.Print 系列标准输出函数。

add_test.go

```
package main

import (
    "fmt"
)

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

func ExampleAdd() {
    a := Add(1, 10)
    b := Add(1, 20)
    fmt.Println(a)
    fmt.Println(b)

    // Output:
    // 11
    // 21
}
```

输出:

```
$ go test -v

=== RUN: ExampleAdd
--- PASS: ExampleAdd (7.361us)
PASS
ok      main    0.043s

$ go test -v -run "Example"          // 如果文件中还有其他测试函数，可以用 run 参数指定测试函数。

=== RUN: ExampleAdd
--- PASS: ExampleAdd (7.202us)
PASS
ok      main    0.045s
```

7.5.4 pprof

go tools 集成了 pprof，方便我们对程序进行性能测试，找出瓶颈所在。可使用代码生成测试数据文件，或使用 go test 生成。

```
import (
    "os"
```

```
    "runtime/pprof"
)

func main() {
    // CPU
    w, _ := os.Create("cpu.out")
    defer w.Close()
    pprof.StartCPUProfile(w)
    defer pprof.StopCPUProfile()

    // Memory
    w2, _ := os.Create("mem.out")
    defer w2.Close()
    defer pprof.WriteHeapProfile(w2)

    // code ...
}
```

程序编译执行后，会生成 cpu.out、mem.out 两个结果文件。

pprof 命令行参数：

- text: 输出文本统计结果。
- web: 生成 svg 图形文件，并用浏览器打开。(需安装 graphviz)

```
$ 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
```

pprof 通过统计采样数来衡量性能，大概每秒 100 次的频率。上面输出结果是 443，也就是说程序大约执行了 5 秒。

结果列表每行给出了一个函数的详细性能指标，按列分别对应：

- 函数本地采样数量 (不包括其调用的其他函数)。

- 函数本地采样数量所占百分比。
- 前几个函数 (包括当前函数) 本地采样总和所占百分比。
- 函数 (包括它调用的函数) 采样总数量。
- 函数采样总数量所占百分比。

采样数越大，意味着该函数执行时间越长，也可能就是问题所在。至于 svg 图片，可以观察各函数之间的调用关系，更加直观，还支持图片缩放。

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

不带参数时，直接进入交互模式，可以用 top、top10、web 等指令。

另，测试 HTTP Server 应该选用 net/http/pprof。

9.6 开发工具

习惯 OSX + Sublime Text 2 + GDB 组合。当然，需要安装必要的插件支持。

(1) GoSublime

配合 gocode 可以完成代码提示、语法检查、错误报告、导入管理以及格式化等功能。
可以用 "<cmd> + dot, dot" 快捷键打开命令列表。

(2) Ctags

利用源码或 brew 工具安装最新的 ctags。

```
$ ctags --version

Exuberant Ctags 5.8, Copyright (C) 1996-2009 Darren Hiebert
  Compiled: Mar 10 2013, 22:02:57
  Addresses: <dhiebert@users.sourceforge.net>, http://ctags.sourceforge.net
  Optional compiled features: +wildcards, +regex
```

创建对应支持 Go 的配置文件 ~/.ctags。

```
--langdef=Go
--langmap=Go:.go
--regex-Go=/func([ \t]+\([^)]+\))?[ \t]+([a-zA-Z0-9_]+)/\2/d,func/
--regex-Go=/var[ \t]+([a-zA-Z_][a-zA-Z0-9_]+)/\1/d,var/
--regex-Go=/type[ \t]+([a-zA-Z_][a-zA-Z0-9_]+)/\1/d,type/
```

在 sublime text 2 中安装 ctags 插件，然后就可以在源码间跳转了。

快捷键	说明
<ctrl> + t, <ctrl> + r	重建 .ctags 符号文件。
<ctrl> + t, <ctrl> + t	跳转到定义。
<ctrl> + t, <ctrl> + b	回到前一位置。
<ctrl> + t, <ctrl> + m	返回最后编辑位置。
<alt> + s	当前文件符号列表。
<alt> + <shift> + s	全部文件符号列表。

(3) SublimeGDB

SublimeGDB 可以打开多个监视窗口，查看调试过程中的环境信息。不过我还是喜欢直接用命令行的 gdb。

第二部分 标准库

内容尚未校对，仅供参考。如发现错漏，请告知。

正逐步更新到 go 1.1。

第 10 章 compress

10.1 zlib

先用 Python zlib 库压缩一段文本，保存到文件中。

```
>>> import zlib

>>> s = zlib.compress("Hello, World!")
>>> s
'x\x9c\xf3H\xcd\xc9\xc9\xd7Q\x08\xcf/\xcaIQ\x04\x00\x1f\x9e\x04j'

>>> f = open("test.dat", "w")
>>> f.write(s)
>>> f.close()

>>> !xxd test.dat
0000000: 789c f348 cdc9 c9d7 5108 cf2f ca49 5104  x..H....Q../.IQ.
0000010: 001f 9e04 6a                             ....j
```

使用标准库 compress/zlib 解压缩。

```
import (
    "bytes"
    "compress/zlib"
    "io/ioutil"
)

func main() {
    data, _ := ioutil.ReadFile("test.dat")
    r, _ := zlib.NewReader(bytes.NewBuffer(data))

    bs, _ := ioutil.ReadAll(r)
    println(string(bs))
}
```

输出:

```
Hello, World!
```

试试压缩数据。

```
import (
    "bytes"
    "compress/zlib"
    "fmt"
    "io"
    "io/ioutil"
)

func main() {
    buffer := bytes.NewBuffer(nil)
```

```
    w := zlib.NewWriter(buffer)
    io.WriteString(w, "Hello, World!")
    w.Close()                                       // 必须调用，确保刷新缓存。

    fmt.Printf("% x\n", buffer.Bytes())
    ioutil.WriteFile("test2.dat", buffer.Bytes(), 0644)
}
```

输出:

```
78 9c f2 48 cd c9 c9 d7 51 08 cf 2f ca 49 51 04 04 00 00 ff ff 1f 9e 04 6a
```

再用 Python 还原。

```
>>> s = open("test2.dat", "r").read()
>>> zlib.decompress(s)
'Hello, World!'
```

10.2 zip

zip 格式应该是在不同平台交换压缩数据文件的最佳选择了。

```
import (
    "fmt"
    "archive/zip"
    "os"
    "io/ioutil"
)

func ZipFile() {
    file, _ := os.Create("test.zip")
    defer file.Close()

    writer := zip.NewWriter(file)
    defer writer.Close()

    datas := []struct{ name, body string }{
        {"a.txt", "Hello, World!"},
        {"b.txt", "12345..."},
    }

    // 直接创建压缩项，直接写入内容数据。
    for _, data := range datas {
        w, _ := writer.Create(data.name)
        w.Write([]byte(data.body))
    }
}

func UnZipFile() {
    zr, _ := zip.OpenReader("test.zip")
    defer zr.Close()

    // 迭代压缩包所有内容
```

```
    for _, f := range zr.File {
        // 获取当前压缩项内容
        br, _ := f.Open()
        bs, _ := ioutil.ReadAll(br)

        fmt.Printf("%s: %s\n", f.Name, string(bs))
    }
}

func main() {
    ZipFile()
    UnZipFile()
}
```

输出:

```
a.txt: Hello, World!
b.txt: 123...
```

我们还可以考虑用 CreateHeader 来替代 Create，以便让压缩项有修改时间、权限等信息。

```
func ZipFile() {
    file, _ := os.Create("test.zip")
    defer file.Close()

    writer := zip.NewWriter(file)
    defer writer.Close()

    for _, name := range []string{"a.txt", "b.txt"} {
        info, _ := os.Stat(name)
        header, _ := zip.FileInfoHeader(info)

        w, _ := writer.CreateHeader(header)
        body, _ := ioutil.ReadFile(name)
        w.Write(body)
    }
}
```

第 11 章 container

第 12 章 crypto

12.1 md5

最痛苦的莫过于不同语言的 MD5 结果却不相同。

```
import (
    "crypto/md5"
    "fmt"
)

func main() {
    md5 := md5.New()

    md5.Write([]byte("ab"))
    md5.Write([]byte("cd"))

    fmt.Printf("%x\n", md5.Sum(nil))
}
```

输出:

```
e2fc714c4727ee9395f324cd2e7f331f
```

Python:

```
>>> import hashlib

>>> md5 = hashlib.md5()
>>> md5.update("abcd")
>>> md5.hexdigest()
'e2fc714c4727ee9395f324cd2e7f331f'
```

12.2 sha256

更高哈希安全需求，可以用 SHA256、SHA512。

```
import (
    "crypto/sha256"
    "fmt"
)

func main() {
    sha := sha256.New()

    sha.Write([]byte("ab"))
    sha.Write([]byte("cd"))

    fmt.Printf("%x\n", sha.Sum(nil))
}
```

输出:

```
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
```

Python:

```
>>> import hashlib

>>> sha = hashlib.sha256()
>>> sha.update("abcd")
>>> sha.hexdigest()
'88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589'
```

12.3 hmac

HMAC 在现有的哈希算法基础上，用密钥进行加密。

```
import (
    "crypto/hmac"
    "crypto/sha256"
    "fmt"
)

func main() {
    key := []byte("123456")
    hash := hmac.New(sha256.New, key)

    hash.Write([]byte("abcd"))
    fmt.Printf("%x\n", hash.Sum(nil))
}
```

输出:

```
a65014c0dfa57751a749866e844b6c42266b9b7d54d5c59f7f7067d973f77817
```

Python:

```
>>> import hashlib, hmac

>>> h = hmac.new("123456", digestmod = hashlib.sha256)
>>> h.update("abcd")
>>> h.hexdigest()
'a65014c0dfa57751a749866e844b6c42266b9b7d54d5c59f7f7067d973f77817'
```

12.4 rand

专用于加密算法的随机数生成器，别和 math/rand 搞混了。

```
import (
    "crypto/rand"
    "fmt"
    "math/big"
)

func main() {
```

```
    // 生成 [0, max) 之间的随机数。
    x, _ := rand.Int(rand.Reader, big.NewInt(10))
    println(x.Int64())

    // 生成指定位长度的随机数。
    y, _ := rand.Prime(rand.Reader, 100)
    fmt.Printf("% x\n", y.Bytes())

    // 生成随机 []byte。
    c := make([]byte, 10)
    rand.Read(c)
    fmt.Printf("% x\n", c)
}
```

输出：

```
2
0f 5f fb b6 fa 25 b5 0c 23 67 ed 14 03
81 cd 01 73 f1 0f c1 db d9 48
```

12.5 des

虽说 DES 算法的加密强度有点问题，但应付常规应用是没问题的。注意确保 DES key 长度是 8 字节，TDES key 是 24 字节。原数据长度则必须是 8 的倍数。

```
import (
    "crypto/cipher"
    "crypto/des"
    "fmt"
)

func main() {
    key := []byte("12345678")
    s := "1234567812345678"

    block, _ := des.NewCipher(key)
    enc := cipher.NewCBCEncrypter(block, key[:8])
    dec := cipher.NewCBCDecrypter(block, key[:8])

    eb := make([]byte, len(s))
    db := make([]byte, len(eb))

    enc.CryptBlocks(eb, []byte(s))
    fmt.Println(eb)

    dec.CryptBlocks(db, eb)
    fmt.Println(string(db))
}
```

输出:

```
[61 117 149 169 139 255 128 157 93 107 113 221 4 20 37 108]
1234567812345678
```

这个加密结果和 pyDes 相同。

```
>>> d = pyDes.des("12345678", pyDes.CBC, "12345678")

>>> eb = d.encrypt("1234567812345678")

>>> map(lambda c: ord(c), eb)
[61, 117, 149, 169, 139, 255, 128, 157, 93, 107, 113, 221, 4, 20, 37, 108]

>>> d.decrypt(eb)
'1234567812345678'
```

12.6 rsa

不对称加密通常用来加密对称密钥。

```
import (
    "crypto/rand"
    "crypto/rsa"
    "fmt"
)

func main() {
    prv, _ := rsa.GenerateKey(rand.Reader, 256)
    pub := &prv.PublicKey

    fmt.Printf("%#v\n%#v\n", pub, prv)

    out, _ := rsa.EncryptPKCS1v15(rand.Reader, pub, []byte("Hello"))
    fmt.Printf("%#v\n", out)

    out, _ = rsa.DecryptPKCS1v15(rand.Reader, prv, out)
    fmt.Println(string(out))
}
```

输出:

```
&rsa.PublicKey
{
    N:54526...,
    E:65537
}

&rsa.PrivateKey
{
    PublicKey:rsa.PublicKey{...},
    D:63864...,
    Primes:[]*big.Int
    {
        27782...,
        19626...
    },
}

[]byte{0x50, 0x8a, 0xb, 0xd0, 0x4b, ...}
```

```
Hello
```

用 Python rsa 解密，必须确保 key 相同。

```
>>> import rsa

>>> prv = rsa.PrivateKey(
54526..., 65537,                          # public
63864..., 27782..., 19626...)             # private

>>> d = [0x50, 0x8a, 0xb, 0xd0, 0x4b, ...]
>>> es = "".join(map(chr, d))

>>> rsa.decrypt(es, prv)
'Hello'
```

当然，反过来也是可以的。这回由 Python 生成 key。

```
>>> pub, prv = rsa.newkeys(256)

>>> pub
PublicKey(77382..., 65537)

>>> prv
PrivateKey(
    77382..., 65537,                      # public
    11276..., 61637..., 12554...)         # private

>>> es = rsa.encrypt("world", pub)

>>> map(ord, es)
[161, 72, 236, 77, ..., 160, 147, 208, 67]
```

注意 key 各字段的对应关系。

```
import (
    "crypto/rand"
    "crypto/rsa"
    "fmt"
    "math/big"
)

func fromBase10(base10 string) *big.Int {
    i := new(big.Int)
    i.SetString(base10, 10)
    return i
}

func getKeys() (prv rsa.PrivateKey, pub rsa.PublicKey) {
    pub = rsa.PublicKey{
        N: fromBase10("77382..."),
        E: 65537,
    }
```

```
    prv = rsa.PrivateKey{
        PublicKey: pub,
        D:         fromBase10("11276..."),
        Primes: []*big.Int{
            fromBase10("61637..."),
            fromBase10("12554..."),
        },
    }

    return
}

func main() {
    prv, _ := getKeys()
    out, _ := rsa.DecryptPKCS1v15(rand.Reader, &prv, []byte{161, 72, 236, ..., 67})
    fmt.Println(string(out))
}
```

输出:

```
world
```

还可以利用 rsa 算法做签名验证，以确保数据未被篡改。

```
import (
    "crypto"
    "crypto/md5"
    "crypto/rand"
    "crypto/rsa"
    "fmt"
)

func main() {
    hash := md5.New()
    hash.Write([]byte("hello"))
    data := hash.Sum(nil)

    prv, _ := rsa.GenerateKey(rand.Reader, 1024)
    pub := &prv.PublicKey
    fmt.Printf("key: %#v\n", pub)

    sig, _ := rsa.SignPKCS1v15(rand.Reader, prv, crypto.MD5, data)
    fmt.Printf("sig: %#v\n", sig)

    if rsa.VerifyPKCS1v15(pub, crypto.MD5, data, sig) == nil {
        println("OK")
    } else {
        println("Error")
    }
}
```

输出:

```
key: &rsa.PublicKey{N:81095...1, E:65537}
sig: []byte{0xd, 0x89, 0xe, 0x91, 0x2, 0x7a, ..., 0xa0}
OK
```

试着用 Python rsa 验证签名。

```
>>> pub = rsa.PublicKey(81095..., 65537)
>>> sig = "".join(map(chr, [0xd, 0x89, 0xe, 0x91, 0x2, 0x7a, ..., 0xa0]))
>>> rsa.verify("hello", sig, pub)
```

第 13 章 database

第 14 章 debug

Symbol 除用于调试，还可以做些其他事情，比如弥补反射不足。

通过配置动态调用函数是个常见的需求，但对于 Go 却有点麻烦。因为 Go reflect 依赖 interface 和 method set，这些对普通函数是没用的。Go Package 也不是 .NET Assembly，想要通过函数名找到目标函数，要么用函数表，要么只能打 Symbol 的主意了。

别着急，我们要对付的不是 .text section。函数变量实际指向编译期常量 FuncVal 地址，所以目标应该是 .rodata。不同平台 section 名字有些不同，在 OSX 环境下，名字以 "__" 开头。

```
package main

import (
    "debug/gosym"
    "debug/macho"
    "fmt"
    "io/ioutil"
    "os"
    "unsafe"
)

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

func dynFunc(name string, fn unsafe.Pointer) {
    // 当前可执行文件。
    f, _ := macho.Open(os.Args[0])

    // Go 函数实际上被包装成一个全局的 FuncVal 对象，因此从 rodata 找。
    rodata := f.Section("__rodata")

    // 相关符号信息段。
    symtab := f.Section("__gosymtab")
    pclntab := f.Section("__gopclntab")

    symdata, _ := symtab.Data()
    pclndata, _ := pclntab.Data()

    pcln := gosym.NewLineTable(pclndata, rodata.Addr)
    table, _ := gosym.NewTable(symdata, pcln)

    // FuncVal 符号名会添加一个后缀。
    sym := table.LookupSym(name + ".f")

    // 转换指针，写入 FuncVal 地址。
    *((*uintptr)(fn)) = uintptr(sym.Value)
}

func main() {
```

```
    // 阻止编译器优化掉 add。
    fmt.Fprint(ioutil.Discard, add)

    // 创建所需的函数类型变量。
    var f func(int, int) int

    // 将 f 指向 main.add FuncVal。
    dynFunc("main.add", unsafe.Pointer(&f))

    // 调用。
    println(f(1, 2))
}
```

输出:

```
3
```

整个过程很简单，从 __rodata 找到 main.add.f 符号，然后获取 FuncVal 地址，并将函数变量指向它即可。如果要重复使用，应该缓存相应的对象。当然，通过 Symbol Table 遍历可执行文件内容就更简单了。

注意: 可以用 ldflags "-w" 删除 DWARF 调试信息，但不能用 "-s" strip 掉符号表。

第 15 章 encoding

15.1 json

数据交互必不可少的东西，远比 XML 招人喜欢。

```
import (
    "bytes"
    "encoding/json"
    "fmt"
)

func main() {
    buffer := bytes.NewBuffer(nil)
    encoder := json.NewEncoder(buffer)
    decoder := json.NewDecoder(buffer)

    data := map[string]interface{}{
        "s": "Hello, World!",
        "i": 1234,
        "m": map[string]int{"x": 1, "y": 2},
    }

    encoder.Encode(data)
    fmt.Printf("%v\n", buffer.String())

    d := make(map[string]interface{})
    decoder.Decode(&d)                              // 必须使用指针
    fmt.Printf("%#v\n", d)
}
```

输出:

```
{"i":1234,"m":{"x":1,"y":2},"s":"Hello, World!"}
map[string]interface {}{"i":1234, "m":map[string]interface {}{"y":2, "x":1}, "s":"Hello, World!"}
```

还有简便函数 Marshal、Unmarshal，Indent 则用于格式化输出结果。

```
import (
    "encoding/json"
    "fmt"
)

func main() {
    data := map[string]interface{}{
        "s": "Hello, World!",
        "i": 1234,
        "m": map[string]int{"x": 1, "y": 2},
    }

    d, _ := json.MarshalIndent(data, "", "    ")
    fmt.Printf("%s\n", string(d))
```

```
    d2 := make(map[string]interface{})
    json.Unmarshal(d, &d2)
    fmt.Printf("%#v\n", d2)
}
```

输出:

```
{
    "i": 1234,
    "m": {
        "x": 1,
        "y": 2
    },
    "s": "Hello, World!"
}
map[string]interface {}{"i":1234, "m":map[string]interface {}{"y":2, "x":1}, "s":"Hello, World!"}
```

除了 map，还可使用 struct，只是会忽略非导出字段。支持匿名结构成员，支持匿名嵌入。

```
type P struct{ X, Y int }

type Data struct {
    S   string
    I   int
    T   struct{ A, B string }        // 支持匿名结构
    M   map[string]string            // 支持字典
    M2  map[string]interface{}
    E   []int
    P                                // 支持匿名嵌入
    f   float32                      // 非导出字段被忽略
}

func main() {
    data := Data{
        S:  "Hello, World!",
        I:  1234,
        T:  struct{ A, B string }{"aaa", "bbb"},
        M:  map[string]string{"a": "abc", "n": "123"},
        M2: map[string]interface{}{"n": 899, "s": "hello"},
        E:  []int{1, 2, 3, 4},
        P:  P{100, 200},
        f:  1.23,
    }

    j, _ := json.MarshalIndent(&data, "", "    ")
    fmt.Printf("%s\n", string(j))

    var d Data                       // 无需显式创建内部引用字段，如 M、M2、E。
    json.Unmarshal(j, &d)
    fmt.Printf("%+v\n", d)
}
```

输出:

```
{
    "S": "Hello, World!",
```

```
    "I": 1234,
    "T": {"A": "aaa", "B": "bbb"},
    "M": {"a": "abc", "n": "123"},
    "M2": {"n": 899, "s": "hello"},
    "E": [1, 2, 3, 4],
    "X": 100,
    "Y": 200
}

{
    S:Hello, World!
    I:1234 T:{A:aaa B:bbb}
    M:map[a:abc n:123]
    M2:map[n:899 s:hello]
    E:[1 2 3 4]
    P:{X:100 Y:200}
    f:0
}
```

通过 struct tag 可以设置更多的信息。

```
type Person struct {
    Username string `json:"name"`                   // 显式指定名称
    Age      int    `json:"age,string"`             // 显式指定 JSON 数据类型
    Address  string `json:"address,omitempty"`      // 如果值为空，则忽略该字段。
    Data     []byte `json:"-"`                       // 显式忽略的字段
}

func main() {
    p := &Person{"Tom", 15, "", []byte{1, 2}}
    b, _ := json.Marshal(p)
    fmt.Println(string(b))

    p2 := new(Person)
    json.Unmarshal(b, p2)
    fmt.Println(p2)
}
```

输出:

```
{"name":"Tom","age":"15"}
&{Tom 15  []}
```

第 16 章 expvar

第 17 章 flag

命令行参数处理。

```
import (
    "flag"
)

func main() {
    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    println(*file, *count)
}
```

可用 --help 查看帮助信息。

```
$ ./test --help
Usage of ./main:
  -count=0: process count.
  -file="default.txt": input filename.
```

输出参数看看效果。

```
$ ./test --file "test.txt" --count=12
test.txt 12
```

帮助信息不好看？修改 Usage 这个默认函数就行了。

```
import (
    "flag"
    "fmt"
    "os"
)

func main() {
    flag.Usage = func() {
        fmt.Fprintf(os.Stderr, "MyProgram, version 0.0.0.1\n")
        fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])
        flag.PrintDefaults()
    }

    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    println(*file, *count)
}
```

输出:

```
$ ./test --help
```

```
MyProgram, version 0.0.0.1
Usage of ./main:
  -count=0: process count.
  -file="default.txt": input filename.
```

Args() 获取非 flag 参数列表，通常放在最后。NFlag()、NArg() 返回相关参数数量。

```
import (
    "flag"
    "fmt"
)

func main() {
    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    fmt.Printf("Flag: %d, %s, %d\n", flag.NFlag(), *file, *count)
    fmt.Printf("Args: %d, %v\n", flag.NArg(), flag.Args())
}
```

输出:

```
$ ./test --file="a.txt" a 123
Flag: 1, a.txt, 0
Args: 2, [a 123]
```

还可以用 set 修改 flag 参数值。

```
func main() {
    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    if *count == 0 { flag.Set("count", "100") }

    println(*file, *count)
}
```

输出:

```
$ ./test --file="a.txt"
a.txt 100
```

Visit 用于遍历命令行显式提供的 flag 参数，VisitAll 则遍历所有已定义的 flag 参数。

```
func main() {
    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    println(*file, *count)

    flag.VisitAll(func(f *flag.Flag) {
        fmt.Printf("%s = %v (default: %v)\n", f.Name, f.Value, f.DefValue)
    })
```

```
}
```

输出:

```
$ ./test --file="a.txt"
a.txt 0
count = 0 (default: 0)
file = a.txt (default: default.txt)
```

不想遍历的话，就用 Lookup() 返回指定名称的 flag 参数信息。

```
func main() {
    file := flag.String("file", "default.txt", "input filename.")
    count := flag.Int("count", 0, "process count.")
    flag.Parse()

    println(*file, *count)

    f := flag.Lookup("count")
    fmt.Printf("%s = %v (default: %v)\n", f.Name, f.Value, f.DefValue)
}
```

输出:

```
$ ./test --file="a.txt"
a.txt 0
count = 0 (default: 0)
```

第 18 章 fmt

18.1 print

格式化字符串。

格式	说明
%v	通用格式。+v 输出字段名，#v Go 语法格式。
%T	类型。
%t	布尔: true、false
%b	二进制
%c	字符，如: 0x6211 会显示 '我'。
%d	十进制
%o	八进制
%q	用双引号包含的 Go 字符串。
%x, %X	十六进制。可以是整数、字节数组或字符串。
%U	Unicode 格式，如：U+6211 表示 '我'。
%e, %E	浮点数科学计数法
%f	浮点数
%s	字符串，自动转换字节数组。
%p	指针
+	数字符号，如：+123。
-	左对齐
#	增加数字进制前缀，取消指针前缀。
' ' 或 0	填充字符

接口类型直接输出其内部对象，可以使用该对象允许的控制格式。如果对象 method set 有 Error() 或 String() 方法，会被自动调用。

```
import (
    "errors"
```

```
    "fmt"
)

type Data struct {
    s string
}

func (self *Data) String() string {
    return "Data: " + self.s
}

func main() {
    // 以十六进制输出字节数组。
    fmt.Printf("% x\n", "abcd")                    // 61 62 63 64

    // 输出字节数组，用空格隔开，并添加前缀。
    fmt.Printf("% #x\n", []byte("abcd"))           // 0x61 0x62 0x63 0x64

    // 字符串左对齐。
    fmt.Printf("[%-10s]\n", "abc")                 // [abc       ]

    // 最小宽度 10，包括小数点和小数部分。
    // 小数位 2，以 0 填充。
    fmt.Printf("%010.2f\n", 123.456)               // 0000123.46

    // "*" 表示从参数中读取宽度。
    fmt.Printf("[%0*d]\n", 5, 123)                 // [00123]
    fmt.Printf("[%-*s]\n", 5, "abc")               // [abc  ]

    // 调用 Error()。
    fmt.Printf("%v\n", errors.New("<error>"))      // <error>

    // 调用 String()。
    fmt.Printf("%s\n", &Data{"abc"})               // Data: abc
}
```

注：内置函数 print/println 会尝试将数据转移到 goroutine write buffer，或默认输出到 stderr。通常适用于调试，正式的输出逻辑建议使用 fmt。

18.2 scan

Scan 和 Print 相反，接收输入然后转换成目标类型数据。以换行符或空格来分隔多个输入项，函数返回正确处理的项数量和错误信息。

不同于 Scanln 遇到换行立即终止扫描，Scanf 可以匹配换行符。

```
func main() {
    var s string
    var d int

    fmt.Println(fmt.Scanf("%s\n%d", &s, &d))
```

```
    fmt.Println(s, d)
}
```

输出:

```
abc                         // 输入
123                         // 输入
2 <nil>
abc 123
```

Scan 则根据需要的项数量将换行符当做空格处理。多个连续空白当做单个空格处理。

```
fmt.Scan(&s, &d)
fmt.Sscan("abc 123", &s, &d)
fmt.Sscan("abc\n123", &s, &d)
```

除了某些未实现 (%p, %T, #, +) 的，大部分控制符含义与 Print 相同。

```
func main() {
    var s string
    var d int
    var f float32

    // 支持前缀
    fmt.Sscan("abc 0x123", &s, &d)
    fmt.Printf("%s %#x\n", s, d)                    // abc 0x123

    // 最大长度控制。如果有多余字符会影响后续参数
    fmt.Sscanf("abcd", "%2s", &s)
    fmt.Printf("%s\n", s)                           // ab

    // 只能提供长度，无法控制小数位。
    fmt.Sscanf("123.456", "%5f", &f)
    fmt.Printf("%f\n", f)                           // 123.400002
}
```

Scan 对于有固定格式的文件和字符串操作比较方便，但缺乏容错能力。可以考虑全部读取后，用正则表达式做进一步处理。

第 19 章 go

利用 AST 可以做一些辅助开发工具。比如自行定义一些类似 "// +build" 这类的特殊注释。

```
import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
)

func main() {
    src := `
package main

type data struct{}

// +aop:checkargs
func (*data) test(a int, b string) map[string]int {
    return nil
}

// comment2
func main() {
    println("Hello, World!")
}
`
    fset := token.NewFileSet()
    f, err := parser.ParseFile(fset, "", src, parser.ParseComments)
    if err != nil {
        panic(err)
    }

    ast.Inspect(f, func(n ast.Node) bool {
        if f, ok := n.(*ast.FuncDecl); ok {
            fmt.Printf("name: %s\n", f.Name)
            fmt.Printf("params: %+v\n", f.Type.Params)
            fmt.Printf("result: %+v\n", f.Type.Results)
            fmt.Printf("line: %d\n", fset.Position(f.Pos()).Line)
            fmt.Printf("doc: %s\n", f.Doc.Text())
        }

        return true
    })
}
```

输出:

```
name: test
params: &{Opening:71 List:[0x210257580 0x2102575c0] Closing:87}
result: &{Opening:0 List:[0x210257640] Closing:0}
line: 7
doc: +aop:checkargs
```

```
name: main
params: &{Opening:145 List:[] Closing:146}
result: <nil>
line: 12
doc: comment2
```

通过遍历找到所有函数和方法，接下来就可以利用 Doc 里面的特殊标记，比如 "aop" 进行所需的处理。当然，这需要想象力。

第 20 章 hash

标准库提供了多种常用数据校验算法。

- CRC32: 检错能力强，开销小，不能发现的错误几率低于 0.0047%。
- Adler32: 速度更快。但可靠性略差，尤其是对较短数据的处理存在问题。

```
import (
    "hash/adler32"
    "hash/crc32"
    "strings"
)

func main() {
    println(crc32.ChecksumIEEE([]byte("abcd")))

    s := strings.Repeat("abcd", 100)
    println(adler32.Checksum([]byte(s)))
}
```

输出:

```
3984772369
2430114281
```

Python:

```
>>> zlib.crc32("abcd") & 0xffffffff
3984772369

>>> zlib.adler32("abcd" * 100) & 0xffffffff
2430114281
```

第 21 章 html

第 22 章 image

第 23 章 index

后缀数组 (suffix array) 搜索效率极高，应用于很多复杂的字符串处理算法，最常见的就是全文检索了。为了测试和常用搜索算法的性能差异，先用标准库 bytes 写两个正反向的顺序搜索算法，并确保结果一致。

```
import (
    "bytes"
    "fmt"
    "index/suffixarray"
)

// 正向顺序搜索
func search(data, sep []byte) []int {
    ret := make([]int, 0, 300)
    pos, length := 0, len(sep)
    for {
        if n := bytes.Index(data, sep); n >= 0 {
            ret = append(ret, pos+n)
            data = data[n+length:]
            pos += (n + length)
        } else {
            break
        }
    }
    return ret
}

// 逆向顺序搜索
func search2(data, sep []byte) []int {
    ret := make([]int, 0, 300)

    for {
        if n := bytes.LastIndex(data, sep); n >= 0 {
            ret = append(ret, n)
            data = data[:n]
        } else {
            break
        }
    }
    return ret
}

func main() {
    data := []byte{2, 1, 2, 3, 2, 1, 2, 4, 4, 2, 1}
    sep := []byte{2, 1}

    fmt.Printf("%v\n", search(data, sep))
    fmt.Printf("%v\n", search2(data, sep))
    fmt.Printf("%v\n", suffixarray.New(data).Lookup(sep, -1))
}
```

输出:

```
[0 4 9]
[9 4 0]
[9 0 4]
```

后缀数组搜索算法的结果非顺序输出，但这不影响测试。接下来，随便保存一个标准库帮助网页做素材，搜索最常见的 "func"，看看性能测试结果。

main_test.go

```
import (
    "index/suffixarray"
    "io/ioutil"
    "os"
    "testing"
)

var data, sep []byte
var index *suffixarray.Index

func init() {
    f, _ := os.Open("test.html")
    data, _ = ioutil.ReadAll(f)
    sep = []byte("func")
    index = suffixarray.New(data)
}

func BenchmarkSuffixArray(b *testing.B) {
    n := 0
    for i := 0; i < b.N; i++ {
        n = len(index.Lookup(sep, -1))
    }

    b.Log(n)
}

func BenchmarkBytesIndex(b *testing.B) {
    n := 0
    for i := 0; i < b.N; i++ {
        n = len(search(data, sep))
    }

    b.Log(n)
}

func BenchmarkBytesIndex2(b *testing.B) {
    n := 0
    for i := 0; i < b.N; i++ {
        n = len(search2(data, sep))
    }

    b.Log(n)
}
```

输出：

```
$ go test -v -bench "."

BenchmarkSuffixArray   1000000             2934 ns/op
--- BENCH: BenchmarkSuffixArray
       main_test.go:25: 244

BenchmarkBytesIndex      10000           142591 ns/op
--- BENCH: BenchmarkBytesIndex
       main_test.go:33: 244

BenchmarkBytesIndex2     10000           228976 ns/op
--- BENCH: BenchmarkBytesIndex2
       main_test.go:41: 244

ok     learn  6.788s
```

逆向顺序搜索算法虽然简单一点，但性能更差。

虽然顺序算法算不上最优，但与后缀数组相比，如此大的性能差异就已经说明问题了。当然，后缀数组也不是没有缺点，比如会使用更多的内存，还得做预处理 (切分和排序) 操作，并不适合用于简单字符串搜索。

第 24 章 io

24.1 interface

标准库 io 包定义了 IO 操作所需的常用接口。

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type Seeker interface {
    Seek(offset int64, whence int) (ret int64, err error)    // 0: Begin; 1: Current; 2: End
}

type Closer interface {
    Close() error
}
```

除了相关接口，还有几个实用的类型。

- LimitedReader: 限制最多可读取数据的数量。
- MulitReader: 串联多个 Reader 供连续读取数据。
- TeeReader: 将读取的数据同步写入另外一个 Writer 中。
- SectionReader: 同时指定开始和结束位置的片段读取器。
- MultiWriter: 同时将数据写入多个 Writer 中。

常用类型转换:

- bytes: NewReader([]byte)、NewBuffer([]byte)、NewBufferString(string)。
- strings: NewReader(string)

注意：

- Read 方法会同时返回尾数据和 EOF。在处理错误前，应该先检查数据。
- Close 方法不会将缓存数据写入存储设置，需调用 Flush 或 Sync 方法。
- ReadAt、WriteAt 操作不会影响底层对象的 offset 值。
- 标准输入输出 os.Stdin、Stdout、Stderr 已经包装成 os.File。
- 空设备 /dev/null，建议使用 os.DevNull 或 ioutil.Discard。

标准库 io、ioutil 包中另有很多便捷函数，可简化编码。

24.2 text

通常使用 os.Open / Create 打开或创建文件，然后用 bufio 进行缓冲 IO 操作，以提高性能。下面是一个按行读取 UTF-8 格式文本文件的例子。

- Open: 只读。
- OpenFile: 通过 flag 和 perm 控制打开模式。如：OpenFile("test.txt", os.O_RDWR, 0600)
- Create: 以读写方式创建文件，已有文件会被 Truncate(0)。

```
func main() {
    f, _ := os.Open("test.txt")
    defer f.Close()

    r := bufio.NewReaderSize(f, 4096)

    for {
        line, isPrefix, err := r.ReadLine()

        if isPrefix {
            print(string(line))             // 如果是一行的前部分，则不换行。(行数据超出缓冲区大小)
        } else if len(line) > 0 {
            println(string(line))
        }

        if err == io.EOF { break }
    }
}
```

利用 fmt 包中的函数可以非常方便地读写格式化字符串。

24.3 binary

在 Unix-like 系统下，并不区分文本文件和二进制文件。如果我们希望读写二进制文件，则必须将相关类型转换成 []byte。

下面例子中，使用 encoding/binary 在 Number 和 []byte 间进行转换。当然，还可以使用序列化等手段一次性转换一个结构体。

```
import (
    "encoding/binary"
    "fmt"
    "log"
    "os"
)

func checkError(err interface{}) {
    if err != nil { log.Fatal(err) }
}
```

```
func write() {
    f, err := os.Create("test.dat")
    checkError(err)

    defer func() {
        f.Sync()                                                    // 必须显式调用, Close 不会这事。
        f.Close()
    }()

    var i int32 = 0x1234                                            // 必须是具体长度的类型
    checkError(binary.Write(f, binary.LittleEndian, i))

    var d float32 = 0.1234
    checkError(binary.Write(f, binary.LittleEndian, d))

    var s string = "Hello, 中国!"
    checkError(binary.Write(f, binary.LittleEndian, int32(len(s))))   // 先写入字符串长度
    _, err = f.WriteString(s)                                       // 写入字符串内容
    checkError(err)
}

func read() {
    f, err := os.Open("test.dat")
    checkError(err)
    defer f.Close()

    var i int32
    checkError(binary.Read(f, binary.LittleEndian, &i))

    var d float32
    checkError(binary.Read(f, binary.LittleEndian, &d))

    var l int32
    checkError(binary.Read(f, binary.LittleEndian, &l))
    s := make([]byte, l)
    _, err = f.Read(s)

    fmt.Printf("%#x; %f; %s;\n", i, d, string(s))
}

func main() {
    write()
    read()
}
```

读取字符串除长度参数外，也可用结束标记。比如 '\n'，然后用 bufio.Reader.ReadString() 读取。

24.4 pipe

某些时候，需要在多个任务间进行数据流通讯。io.Pipe 提供读写管道。

```
func main() {
    wait := make(chan bool)
    r, w := io.Pipe()

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Fprintf(w, "data%d\n", i)
        }

        w.Close()
    }()

    go func() {
        for {
            var s string
            _, err := fmt.Fscanln(r, &s)
            if err == io.EOF { break }
            println(s)
        }

        wait <- true
    }()

    <-wait
}
```

24.5 encoding

除了 UTF-8 编码，我们通常还需要读写 GB、UTF-16 等其他编码格式文件。标准库并没有内置字符集转换包，须借助第三方扩展。

```
$ go get code.google.com/p/mahonia/
```

先用 Python 准备一个 gb18030 的文件样本。

```
>>> import sys, codecs

>>> reload(sys)
>>> sys.setdefaultencoding("utf-8")

>>> f = codecs.open("test.txt", "w", "gb18030")
>>> f.write("中国人")
>>> f.close()

>>> !xxd test.txt
0000000: d6d0 b9fa c8cb                           ......
```

导入 mahonia 包，创建 gb18030 Decoder 进行编码转换。

```
import (
    "bufio"
```

```
    "code.google.com/p/mahonia"
    "log"
    "os"
)

func checkError(err interface{}) {
    if err != nil { log.Fatal(err) }
}

func main() {
    f, err := os.Open("test.txt")
    checkError(err)
    defer f.Close()

    decoder := mahonia.NewDecoder("gb18030")
    r := bufio.NewReader(decoder.NewReader(f))

    line, _, err := r.ReadLine()
    checkError(err)
    println(string(line))
}
```

除了 mahonia，还可以使用 go-charset，只是没这个方便罢了。

24.6 buffer

除了文件，缓冲区是另一个最常用的 IO 操作了 —— bytes.Buffer。

```
import (
    "bytes"
    "encoding/binary"
    "fmt"
)

func main() {
    buffer := bytes.NewBuffer(nil)

    // 写入整数
    var i int32 = 0x1234
    binary.Write(buffer, binary.LittleEndian, i)

    // 写入字符串
    buffer.WriteString("abc")

    fmt.Printf("% x\n", buffer.Bytes())
}
```

输出:

```
34 12 00 00 61 62 63
```

默认情况下 (没有提供初始值)，Buffer 使用结构体自带的一个 [64]byte 数组作为存储空间。当超出限制时，另创建一个 2x 的空间，并复制未读取的数据。

当缓冲区的数据被完全读取后，会将写入位置重置到底层数据的开始处。因此只要读写操作平衡，就无需担心内存会持续增长。

UnreadByte 和 UnreadRune 方法的作用是忽略刚才的读操作，将内部 offset 恢复到读操作以前。

```
import (
    "bytes"
    "io"
)

func main() {
    buffer := bytes.NewBuffer([]byte{0, 1, 2, 3, 4})

    c, _ := buffer.ReadByte()
    buffer.UnreadByte()
    println(c)

    for {
        c, err := buffer.ReadByte()
        if err == io.EOF { break }
        println(c)
    }
}
```

输出:

```
0
0
1
2
3
4
```

24.7 temp

使用临时文件是常用的编程需求。

- os.TempDir: 返回系统临时目录。
- ioutil.TempDir: 在指定目录或系统临时目录下创建指定前缀名称的临时目录。
- ioutil.TempFile: 在指定目录或系统临时目录下创建指定前缀名称的可读写文件对象。

ioutil.TempDir 和 TempFile 可以确保不同的进程或者每次调用都唯一临时名称，但创建者必须负责删除这些临时目录和文件。当不指定目标目录时，表示使用系统临时目录。

```
import (
    "fmt"
    "io/ioutil"
    "os"
)
```

```
func main() {
    f, _ := ioutil.TempFile("", "abc")

    defer func() {
        f.Close()
        os.Remove(f.Name())
    }()

    fmt.Println(f.Name())
}
```

输出:

```
/var/folders/zb/xvwdb1dj4rv450lk847s_z200000gn/T/abc471645390
```

24.8 path

标准库提供了两个路径包，其中 path 专用于 Unix-Like 环境，建议使用 filepath。

filepath.Abs 的作用是获取完整的绝对路径， 可以用 IsAbs 判断是否绝对路径。
类似 "$HOME/Documents" 这样包含环境变量的地址，则用 os.ExpandEnv 展开。只是没有处理 "~/.profile" 这类路径的函数，十分让人奇怪。

```
import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    fmt.Println(filepath.Abs("./demo.txt"))
    fmt.Println(os.ExpandEnv("$HOME/demo.txt"))
    fmt.Println(os.ExpandEnv("$HOME" + "~/demo.txt"[1:]))  // 判断 "~/" 前缀，替换。本处简单处理。

    fmt.Println(filepath.IsAbs("/etc/a.conf"), filepath.IsAbs("./demo.txt"))
}
```

输出:

```
/Users/yuhen/Documents/go/src/test/demo.txt <nil>
/Users/yuhen/demo.txt
/Users/yuhen/demo.txt
true false
```

Dir 用于返回目录名，Base 则返回文件全名，Ext 获取扩展名。

```
p, _ := filepath.Abs("./demo.txt")

println(p)
println(filepath.Dir(p))
println(filepath.Base(p))
println(filepath.Ext(p))
```

输出:

```
/Users/yuhen/Documents/go/src/test/demo.txt
```

```
/Users/yuhen/Documents/go/src/test
demo.txt
.txt
```

Clean 则用于将路径名清理干净，这在拼接多级路径时很有用。

```
p := "..//test/../test/./demo.txt"
println(filepath.Clean(p))
```

输出:

```
../test/demo.txt
```

Rel 和 Abs 相反，用于获取相对路径。

```
p, _ := filepath.Abs("./a/b/c/demo.txt")
d, _ := filepath.Abs("./a")
r, _ := filepath.Rel(d, p)

println(p)
println(r)
```

输出:

```
/Users/yuhen/Documents/go/src/test/a/b/c/demo.txt
b/c/demo.txt
```

Split 将路径分割成 "路径" 和 "文件名" 两部分，而 Join 则其合并成为一个完整路径。

```
p1, _ := filepath.Abs("./demo.txt")
dir, name := filepath.Split(p1)
println(dir, ",", name)

p2 := filepath.Join(dir, name)
println(p2)
```

输出:

```
/Users/yuhen/Documents/go/src/test/ , demo.txt
/Users/yuhen/Documents/go/src/test/demo.txt
```

不要误解 SplitList，它的实际用途是分解由特定符号分隔的多个路径。比如在 *nix 系统中用 ":" 分隔多个 PATH 路径。分隔符由 os.PathListSeparator 常量定义。

```
path := os.Getenv("PATH")
println(path)
fmt.Printf("%#v\n", filepath.SplitList(path))
```

输出:

```
/opt/local/bin:/opt/local/sbin:/usr/local:/usr/local/bin:/usr/local/sbin:/usr/bin:/bin:/usr/sbin
[]string{"/opt/local/bin", "/opt/local/sbin", "/usr/local", "/usr/local/bin", "/usr/local/sbin", "/
usr/bin", "/bin", "/usr/sbin"}
```

Match 和 Glob 都是通配符判断，语法规则简单：

- *: 任意 n 个字符。
- ?: 0 或 1 个字符。

- []: 匹配括号内的任意字符。
- [^] 或 [!]: 不匹配括号内的任意字符。
- {,}: 匹配多组通配字符中的一个。

Glob 用于匹配当前目录下的文件，而 Match 则匹配字符串。如果需要使用特定语法字符，使用 "\" 转义。注意匹配的文件名不能包含路径，可以用 Base 函数获取文件名。

```
m, _ := filepath.Glob("m*.go")
fmt.Println(m)

m2, _ := filepath.Match("*.go", "main.go")
fmt.Println(m2)
```

输出:

```
[main.go]
true
```

Walk 用于遍历多级路径下的所有文件。包括 "." 在内的所有目录和文件都会触发 walkFunc，当该函数返回有效错误 (err != nil) 时，遍历立即终止，Walk 会返回该错误。

```
err := filepath.Walk(".", func(path string, info os.FileInfo, err error) error {
    if !info.IsDir() {
        fmt.Printf("%s, %d\n", path, info.Size())
    }

    return nil
})

fmt.Println(err)
```

输出:

```
code/evaltest.go, 1234
code/mgotest.go, 1244
main, 1327120
main.go, 261
<nil>
```

但如果是为了忽略某些子目录时，可以在 walkFunc 内直接返回 SkipDir。

```
err := filepath.Walk(".", func(path string, info os.FileInfo, err error) error {
    if info.IsDir() && strings.Contains(path, "code") {
        return filepath.SkipDir
    }

    fmt.Printf("%s, %d\n", path, info.Size())
    return nil
})

fmt.Println(err)
```

输出:

```
., 170
main, 1327120
```

```
main.go, 331
<nil>
```

EvalSymlinks 获取符号链接的真实目标文件路径。

```
p, _ := filepath.EvalSymlinks("/usr/local/s.sh")
println(p)
```

输出:

```
/users/yuhen/Documents/OSX/config/s.sh
```

FromSlash 和 ToSlash 的作用是在不同平台上转换路径表达方式，比如将 Windows "\a\b.txt" 转换为 *nix 路径 "/a/b.txt"，或反之。

第 25 章 log

轻量级的日志解决方案。使用 Mutex 同步，支持并发调用。

- Print: 显示信息。
- Fatal: Print + os.Exit(1)。
- Panic: Print + panic。

默认将信息输出到 stderr，可以使用 SetOutput() 将其保存到文件。

```
import (
    "log"
    "os"
)

func init() {
    f, _ := os.Create("test.log")

    log.SetOutput(f)
    log.SetPrefix("[TEST]")
    log.SetFlags(log.LstdFlags | log.Lshortfile)
}

func main() {
    log.Panicln("Error ?")
}
```

输出:

```
$ go run main.go
panic: Error ?

goroutine 1 [running]:
log.Panicln(0x22101aef78, 0x1, 0x1)
       /usr/local/go/src/pkg/log/log.go:321 +0xaa
main.main()
       /Users/yuhen/test/main.go:17 +0x9d

goroutine 2 [runnable]:
exit status 2

$ cat test.log
[TEST]2013/04/08 21:56:05 main.go:17: Error ?
```

SetPrefix 设置头标记，用来分组多个 Logger 输出。SetFlags 则用来设置时间、文件名等参数。

第 26 章 math

26.1 rand

伪随机数生成器。如果 seed 相同，那么生成的随机数相同。除全局函数，还可创建独立对象。

```
import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    // 通常用 UnixNano 做 Seed。
    r := rand.New(rand.NewSource(time.Now().UnixNano()))

    fmt.Printf("%d\n", r.Int())
    fmt.Printf("%f\n", r.Float32())

    // 生成 [0, n) 随机列表。
    fmt.Println(r.Perm(10))
}
```

输出：

```
1763195000996166836
0.440542
[8 3 0 6 5 7 4 2 1 9]
```

26.2 big

超大整数。提供按位和字节操作方法，以及其他各种运算。

可直接当位图使用。

```
import (
    "fmt"
    "math/big"
)

func main() {
    i := big.NewInt(0)

    // 1<<1000
    i.SetBit(i, 1000, 1)
    println(i.Bit(1000))

    fmt.Printf("%s, % x\n", i, i.Bytes())
}
```

支持按常用进制输出。

```
func main() {
    i := big.NewInt(0)

    i.SetBytes([]byte{0x1, 0x2, 0x3, 0x4})

    fmt.Printf("%#x\n", i)
    fmt.Printf("%# x\n", i.Bytes())
    fmt.Printf("%b\n", i)
}
```

第 27 章 net

27.1 tcp

虽然 net 包提供了具体的 TCPListener、UDPConn 等类型，但我们通常直接用 Listen 和 Dial 工厂函数，通过相关参数它们会自动创建合适的目标类型对象。基于接口编程也是基本的设计原则。

```
import (
    "bufio"
    "fmt"
    "io"
    "log"
    "net"
    "runtime"
)

func server() {
    listen, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatalln(err)
    }

    for {
        conn, _ := listen.Accept()

        go func() {
            defer conn.Close()
            rw := bufio.NewReadWriter(bufio.NewReader(conn), bufio.NewWriter(conn))

            for {
                s, err := rw.ReadString('\n')
                if err == io.EOF {
                    fmt.Println("[server] client close...")
                    return
                }

                println("[server]", s)
                fmt.Fprintf(rw, "server: %s\n", s)
                rw.Flush()
            }
        }()
    }
}

func client() {
    conn, _ := net.Dial("tcp", "localhost:8080")
    defer conn.Close()

    rw := bufio.NewReadWriter(bufio.NewReader(conn), bufio.NewWriter(conn))
    fmt.Fprintln(rw, "Hello, World!")
    rw.Flush()
```

```
    s, _ := rw.ReadString('\n')
    fmt.Println("[client]", s)
}

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    go server()
    go client()

    select{}
}
```

使用 goroutine 大大简化了 server 编码模式，而且远比 threading 要高效。

27.2 udp

net.Listen 不接受 UDP 协议，另外也无需 Accept 操作。

```
import (
    "fmt"
    "io"
    "log"
    "net"
    "runtime"
)

func server() {
    addr, _ := net.ResolveUDPAddr("udp", ":9999")
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        log.Fatalln(err)
    }

    defer conn.Close()

    for {
        buffer := make([]byte, 1024)
        n, addr, err := conn.ReadFromUDP(buffer)

        if err != nil {
            fmt.Println(err)
        }

        fmt.Println(string(buffer[:n]), addr)
    }
}

func client() {
    addr, _ := net.ResolveUDPAddr("udp", ":9999")
    conn, _ := net.DialUDP("udp", nil, addr)
```

```
    defer conn.Close()

    for i := 0; i < 10; i++ {
        io.WriteString(conn, "Hi!")
    }
}

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    go server()
    go client()

    select {}
}
```

27.3 http

27.3.1 server

使用 net/http 包开发 WebServer，需要两个基本元素：

- Server: 监听，并为每个客户端接入创建一个 goroutine。
- Handler: ServeHTTP 接口方法，通过 response、request 参数处理请求信息和返回结果。

```
import (
    "io"
    "net/http"
    "runtime"
    "time"
)

type myHandler struct{}

func (*myHandler) ServeHTTP(response http.ResponseWriter, request *http.Request) {
    http.SetCookie(response, &http.Cookie{Name: "a", Value: "xxx"})
    http.SetCookie(response, &http.Cookie{Name: "b", Value: "yyy"})
    response.Header().Add("Server", "go/1.0")
    io.WriteString(response, "Hello, World!")
}

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    server := &http.Server{
        Addr:        ":8080",
        Handler:     &myHandler{},
        ReadTimeout: time.Second * 5,       // 建议设置，避免端口意外未释放，无法接入新的连接。
    }
```

```
    server.ListenAndServe()
}
```

输出:

```
$ http get http://localhost:8080
HTTP/1.1 200 OK
Date: Mon, 07 May 2012 12:44:45 GMT
Server: go/1.0
Set-Cookie: a=xxx
Set-Cookie: b=yyy
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8

Hello, World!
```

利用包中提供的函数，可以简化相关代码。

```
func main() {
    http.ListenAndServe(":8080", &myHandler{})
}
```

可以用 NotFound、Error、Redirect 来处理 "特殊" 的返回结果。

```
func (*myHandler) ServeHTTP(response http.ResponseWriter, request *http.Request) {
    http.NotFound(response, request)
}
```

输出:

```
$ http get http://localhost:8080
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
Date: Mon, 07 May 2012 12:52:01 GMT
Transfer-Encoding: chunked

404 page not found
```

包中自带了一些常用的 Handler，不过最重要的就是组合其他 Handler，然后通过 url Route 进行匹配调用的 ServerMux。

采取路径前缀最长匹配策略，不支持正则。比如优先匹配 /a/b/，然后才是 /a/ 前缀。

```
import (
    "io"
    "net/http"
)

type myHandler struct{}

func (*myHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "server: "+r.URL.String())
}

func main() {
    mux := http.NewServeMux()
```

```
    // 支持 /a/、/a/b 路径
    mux.Handle("/a/", &myHandler{})

    // 静态文件。/static/filename 经 StripPrefix 将前缀移除后，FileServer 即可访问指定的文件。
    mux.Handle("/static/", http.StripPrefix("/static/",
        http.FileServer(http.Dir("/go/src/pkg/net/http"))))

    // 使用函数做为 Handler。
    mux.HandleFunc("/b", func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "bbb")
    })

    // 路径跳转
    mux.Handle("/c", http.RedirectHandler("/b", http.StatusFound))

    http.ListenAndServe(":8080", mux)
}
```

输出:

```
$ http get http://localhost:8080/a/
HTTP/1.1 200 OK
Date: Mon, 07 May 2012 13:24:39 GMT
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8

server: /a/

$ http get http://localhost:8080/a/b
HTTP/1.1 200 OK
Date: Mon, 07 May 2012 13:24:41 GMT
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8

server: /a/b

$ http get http://localhost:8080/b
HTTP/1.1 200 OK
Date: Mon, 07 May 2012 13:24:49 GMT
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8

bbb

$ http get http://localhost:8080/c
HTTP/1.1 302 Found
Date: Tue, 08 May 2012 06:52:17 GMT
Location: /b
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8

<a href="/b">Found</a>.

$ http get http://localhost:8080/static/server.go
```

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 35840
Content-Type: text/plain; charset=utf-8
Date: Mon, 07 May 2012 13:24:59 GMT
Last-Modified: Thu, 26 Apr 2012 20:37:39 GMT

// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.
// HTTP server.  See RFC 2616.
```

利用包中提供的默认 DefaultServeMux 变量，我们可以进一步简化上述代码。

```
import (
    "io"
    . "net/http"
)

type myHandler struct{}

func (*myHandler) ServeHTTP(w ResponseWriter, r *Request) {
    io.WriteString(w, "server: "+r.URL.String())
}

func main() {
    Handle("/a/", &myHandler{})
    Handle("/static/", StripPrefix("/static/", FileServer(Dir("/usr/local/go/src/pkg/net/http"))))
    HandleFunc("/b", func(w ResponseWriter, r *Request) {
        io.WriteString(w, "bbb")
    })

    ListenAndServe(":8080", nil)
}
```

还可以用 Hijacker 将 ResponseWriter 转换为 net.Conn 和 bufio.ReadWriter，以便我们更 "精确" 控制读写过程。具体使用方法请参考官方文档。

27.3.2 https

如需要启用 HTTPS 安全协议，可用 openssl 创建所需的 X.509 证书。

```
$ openssl genrsa -out key.pem 1024/2038                              // 生成 key.pem
$ openssl req -new -x509 -key key.pem -out cert.pem -days 1095    // 生成 cert.pem
```

改用 TLS 版本的函数启动 WebServer。

```
http.ListenAndServeTLS(":10443", "cert.pem", "key.pem", nil)
```

客户端方面，因为生成的是 "Bad Certificate"，因此要指定一个特殊的参数，否则无法正常访问。

```
import (
    "crypto/tls"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "net/http"
)

func Server() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "Hello, World!")
    })

    if err := http.ListenAndServeTLS(":10443", "cert.pem", "key.pem", nil); err != nil {
        log.Fatalln(err)
    }
}

func Test() {
    client := &http.Client{
        Transport: &http.Transport{
            TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
        },
    }

    if res, err := client.Get("https://localhost:10443"); err != nil {
        log.Fatalln(err)
    } else {
        defer res.Body.Close()
        rs, _ := ioutil.ReadAll(res.Body)
        fmt.Println(string(rs))
    }
}
```

27.3.3 keep-alive

http.Client 默认使用 DefaultTransport，会打开 keep-alive 以便重复使用 Connection。但某些时候会导致内存泄露。可以采取以下方式关闭 keep-alive。

(1) 自定义 Transport，将 DisableKeepAlives 设为 true。

```
func main() {
    for {
        client := &http.Client{
            Transport: &http.Transport{DisableKeepAlives: true},  // 关闭 Keep-Alive
        }

        res, _ := client.Get("https://localhost:8080/")
        rs, _ := ioutil.ReadAll(res.Body)
        fmt.Println(string(rs))
```

```
        res.Body.Close()
    }
}
```

(2) 自定义 Request，将 Close 设为 true。

```
func main() {
    for {
        client := new(http.Client)

        req, _ := http.NewRequest("GET", "https://localhost:8080/", nil)
        req.Close = true                                          // 关闭 Keep-Alive

        res, _ := client.Do(req)                                  // Do 接受 Request 参数。
        rs, _ := ioutil.ReadAll(res.Body)
        fmt.Println(string(rs))
        res.Body.Close()
    }
}
```

27.4 url

可能是为了避免形成依赖关系，http.Request 并没有直接提供访问 URI 参数的方法，所需操作需要额外使用 url 包。

分解函数 Parse() 和 ParseRequestURI() 的区别在于前者能正确处理 #fragment 信息，后者认为浏览器已经将该信息处理掉，其结果就是下面这样：

```
func main() {
    s := "http://localhost:8080/a.html?x=1#bottom"
    u, _ := url.Parse(s)
    u2, _ := url.ParseRequestURI(s)

    fmt.Printf("%#v\n%#v\n", u, u2)
}
```

输出:

```
{Scheme:"http", Host:"localhost:8080", Path:"/a.html", RawQuery:"x=1", Fragment:"bottom", ...}
{Scheme:"http", Host:"localhost:8080", Path:"/a.html", RawQuery:"x=1#bottom", Fragment:"", ...}
```

相关方法都能自动完成编码处理，也可调用 QueryEscape()、 QueryUnescape() 自行处理。

```
func main() {
    u, _ := url.Parse("http://localhost:8080/%E6%B5%8B%E8%AF%95.html?x=%E6%95%B0%E6%8D%AE")
    fmt.Printf("#v\n%#v\n", u, u.Query())
}
```

输出:

```
&url.URL{
    Scheme:"http",
    Path:"/测试.html",
    RawQuery:"x=%E6%95%B0%E6%8D%AE",
```

```
    ...
}

url.Values{
    "x":[]string{"数据"}
}
```

URL.Query() 方法分解请求参数，结果保存 Values 字典中，支持参数中存在多个同名 Key。

```
func main() {
    u, _ := url.Parse("/a.html?x=1&y=2&x=abc")
    fmt.Printf("%#v\n", u.Query())
}
```

输出:

```
{
    "x":[]string{"1", "abc"},
    "y":[]string{"2"}
}
```

要自行构建一个 URL 也很容易。

```
func main() {
    v := url.Values{
        "a": {"1", "xyz"},
        "b": {"我们"},
    }

    u := url.URL{
        Scheme:   "https",
        Host:     "www.test.com:9000",
        Path:     "/测试.html",
        RawQuery: v.Encode(),
    }

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

输出:

```
{
    Scheme:"https",
    Host:"www.test.com:9000",
    Path:"/测试.html",
    RawQuery:"a=1&a=xyz&b=%E6%88%91%E4%BB%AC",
    ...
}

"https://www.test.com:9000/%E6%B5%8B%E8%AF%95.html?a=1&a=xyz&b=%E6%88%91%E4%BB%AC"
```

27.5 rpc

默认的 rpc 服务基于 encoding/gob 编码协议。

服务函数签名有严格限制，第一个函数参数用于传递调用参数，通常使用复合类型指针 (比如 *struct)，第二参数是返回值指针。可以用 RegisterName 指定非默认服务名称。

```
import (
    "fmt"
    "log"
    "net"
    "net/rpc"
)

type MyService struct {
}

func (*MyService) Add(args []int, reply *int) error {
    *reply = args[0] * args[1]
    return nil
}

func Server() {
    l, err := net.Listen("tcp", ":8181")
    if err != nil {
        log.Fatalln(err)
    }

    server := rpc.NewServer()
    server.RegisterName("my", new(MyService))
    go server.Accept(l)
}

func Client() {
    client, _ := rpc.Dial("tcp", "127.0.0.1:8181")

    x := 0
    if err := client.Call("my.Add", []int{2, 3}, &x); err != nil {
        log.Fatalln(err)
    }

    fmt.Println(x)
}

func main() {
    Server()
    Client()
}
```

rpc.Server.Accept() 为每个接入的连接创建默认 gobServerCodec 类型进行消息编码。当然，我们可以用其他的编码替换，比如 net/rpc/jsonprc。

```
func Server() {
    server := rpc.NewServer()
    server.RegisterName("my", new(MyService))
```

```
    go func() {
        l, _ := net.Listen("tcp", ":8181")

        for {
            conn, _ := l.Accept()
            go server.ServeCodec(jsonrpc.NewServerCodec(conn))
        }
    }()
}

func Client() {
    client, _ := jsonrpc.Dial("tcp", "127.0.0.1:8181")
    defer client.Close()

    x := 0
    if err := client.Call("my.Add", []int{2, 3}, &x); err != nil {
        log.Fatalln(err)
    }

    fmt.Println(x)
}
```

Client 另提供了一个 Go()，用于异步返回结果。

```
func Client() {
    client, _ := jsonrpc.Dial("tcp", "127.0.0.1:8181")
    defer client.Close()

    x := 0
    rep := client.Go("my.Add", []int{2, 3}, &x, nil)

    rcall := <-rep.Done
    if rep.Error != nil {
        log.Fatalln(rep.Error)
    }

    fmt.Println(x)
    fmt.Println("  Call :", rcall, rcall == rep)
    fmt.Println("  Reply:", *(rcall.Reply.(*int)))
}
```

输出:

```
6
  Call : &{my.Add [2 3] 0xf840088668 <nil> 0xf8400d5000} true
  Reply: 6
```

Go() 返回一个 struct Call，其中的 channel 可以显式通过参数传递，或者由函数自动创建。只需通过 Call.Done 即可知道函数异步调用是否完成。channel 返回的和 Go() 返回的是同一对象。

显式传递 Buffer Channel 的好处是，多次调用服务，然后一次性处理结果。通常情况下，异步架构具备更好的性能。

```
make(chan *Call, 10)
```

下面是使用 http 的例子。

```
func Server() {
    server := rpc.NewServer()
    server.RegisterName("my", new(MyService))

    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        server.ServeHTTP(w, r)
    })

    go http.ListenAndServe(":8181", mux)
}

func Client() {
    client, _ := rpc.DialHTTP("tcp", "127.0.0.1:8181")

    x := 0
    if err := client.Call("my.Add", []int{2, 3}, &x); err != nil {
        log.Fatalln(err)
    }

    fmt.Println(x)
}
```

和 http 包一样，rpc 也提供了操作 DefaultServer 的简便函数。

第 28 章 os

系统编程自然少不了和操作系统打交道。

28.1 system

相关函数使用比较简单，仅挑几个 "异类" 说明一下。

- MkdirAll: 创建多级子目录。
- RemoveAll: 删除多级子目录。

检查文件或目录是否存在，与我们以往先判断后操作相反。直接进行操作，然后根据错误对象，来判断原因。

- IsExist: 根据返回错误对象，判断是否因为 "已存在" 而导致错误发生。
- IsNotExist: 根据返回错误对象，判断是否因为目标 "不存在" 而导致错误发生。
- IsPermission: 根据返回对象，判断是否因为权限不够而导致错误发生。

```
func main() {
    if err := os.Mkdir("a", 0755); os.IsExist(err) {
        println("Directory a exists! Mkdir Error!")
    }

    if file, err := os.Open("./a/test.txt"); os.IsNotExist(err) {
        println("File not exists! Can't Open ...")
    } else if os.IsPermission(err) {
        println("Permission Denied!")
    } else {
        file.Close()
    }
}
```

Stat/Lstat 用于获取文件信息，但 Lstat 只读取符号链接文件自身。

```
import (
    "os"
    "time"
)

func main() {
    name := "./a.txt"
    os.Chmod(name, 0600)

    info, _ := os.Stat(name)
    os.Chtimes(name, time.Now(), info.ModTime())
}
```

在操作链接文件时，需要注意符号链接。

```
func main() {
    os.Link("./a.txt", "./b.txt")

    sa, _ := os.Stat("./a.txt")
    sb, _ := os.Stat("./b.txt")
    fmt.Println(os.SameFile(sa, sb))

    os.Symlink("./a.txt", "./c.txt")
    fmt.Println(os.Readlink("./c.txt"))
}
```

os.Exit 会立即终止进程，不会调用 defer 函数。

```
import (
    "os"
)

func main() {
    defer println("exit...")         // 这个不会被调用...
    os.Exit(-1)
}
```

28.2 environ

Environ 返回全部的环境变量 (key=value)，Getenv 获得单个变量值。

```
import (
    "fmt"
    "os"
)

func main() {
    envs := os.Environ()
    for _, s := range envs {
        fmt.Println(s)
    }

    fmt.Printf("$HOME = %s\n", os.Getenv("HOME"))
}
```

输出:

```
HOME=/Users/yuhen
GOROOT=/usr/local/go
... ...
$HOME = /Users/yuhen
```

还有两个类似模板的简便函数。

```
fmt.Println(os.Expand("$HOME; $GOROOT", func(s string) string {
    return s + " = " + os.Getenv(s)
}))
```

```
fmt.Println(os.ExpandEnv("HOME = $HOME; GOROOT = $GOROOT"))
```

输出:

```
HOME = /Users/yuhen; GOROOT = /usr/local/go
HOME = /Users/yuhen; GOROOT = /usr/local/go
```

ExpandEnv 用来展开环境变量更方便，但 Expand 完全可以作为一个轻便的字符串模板引擎。

28.3 process

使用 Process 可以创建进程以执行其他程序，不过我们常用的是其封装版本 os/exec.Cmd。

```
import (
    "fmt"
    "log"
    "os/exec"
    "io/ioutil"
)

func logErr(err interface{}) {
    if err != nil {
        log.Fatalln(err)
    }
}

func main() {
    cmd := exec.Command("ls", "-lh", "/usr/local/go/bin")
    stdout, _ := cmd.StdoutPipe()

    logErr(cmd.Start())

    bs, _ := ioutil.ReadAll(stdout)         // 必须在 Wait 之前，因为之后进程状态已经被彻底清除。
    fmt.Println(string(bs))

    logErr(cmd.Wait())
}
```

使用方法很简单，Command 创建 Cmd 对象，然后用 Start 方法启动，用 Wait 等待进程结束。在启动前，我们可以挂接 StdoutPipe 等管道，或者将 cmd.Stdout 指向 os.Stdout。

注意：在 Unix-like 体系下，主进程需要获取子进程的退出状态，否则会导致僵尸进程，除非主进程先咯屁，由 init 完成状态检查工作。

```
func main() {
    cmd := exec.Command("ls", "-lh", "/usr/local/go/bin")
    logErr(cmd.Start())

    fmt.Println(cmd.Process.Pid)
    time.Sleep(time.Minute)
}
```

输出:

```
2095
```

执行上面的代码后，另启动一个终端，你会找到一个僵尸进程。

```
$ ps aux | grep 2095 | grep -v grep
yuhen           2095   0.0  0.0        0      0 s000  Z+    8:35下午   0:00.00 (ls)
```

加上 Wait，这个子进程会及时释放掉遗留的状态信息，彻底消散。

另有 Run 方法，包装了 Start + Wait 这个过程。而 Output 方法则会调用 Run，然后返回 stdout 数据。用来改造前面的例子，会使代码更加简洁。CombinedOutput 和 Output 的区别是同时返回 stdout、stderr 信息。

```
func main() {
    bs, err := exec.Command("ls", "-lh", "/usr/local/go/bin").Output()
    logErr(err)

    fmt.Println(string(bs))
}
```

透过 Cmd.Process 我们可以给子进程发送信号，获取 PID。而由 Wait 提取的 ProcessStat 则包括了退出信息等。

```
import (
    "fmt"
    "log"
    "os/exec"
)

func logErr(err interface{}) {
    if err != nil {
        log.Fatalln(err)
    }
}

func main() {
    cmd := exec.Command("ls", "-lh", "/usr/local/go")
    logErr(cmd.Run())

    fmt.Println(cmd.Path)
    fmt.Println(cmd.ProcessState.Exited())
    fmt.Println(cmd.ProcessState.Success())
    fmt.Println(cmd.ProcessState.SystemTime())
}
```

输出:

```
/bin/ls
true
true
2.471ms
```

Process 有个特殊的用途，就是直接通过 FindProcess 挂到某个已经运行的程序上。如下面的例子，我们先用 exec.Cmd 执行 ps 命令获取 top 进程 pid，然后用 Process 发送终止信号给它。

```
import (
    "log"
    "os"
    "os/exec"
    "strings"
    "strconv"
)

func main() {
    cmd := exec.Command("sh", "-c", "ps aux | grep top | grep -v grep")

    bs, err := cmd.Output()
    if err != nil { log.Fatalln(err) }

    ss := strings.Fields(string(bs))
    pid, _ := strconv.Atoi(ss[1])

    proc, _ := os.FindProcess(pid)
    proc.Kill()
    proc.Wait()
}
```

28.4 signal

信号处理方法非常简单：创建 channel，接收 os.signal 数据，然后作出相应的动作即可。

```
import (
    "fmt"
    "os"
    "os/signal"
)

func main() {
    fmt.Println(os.Getpid())

    sig := make(chan os.Signal)
    signal.Notify(sig)

    for s := range sig {
        fmt.Println(s)
    }
}
```

输出:

```
2615
```

我们另起一个终端，看看信号测试效果。

```
$ kill -s INT 2615
```

```
$ kill -s TERM 2615
$ kill -s USR1 2615
$ kill -s USR2 2615
$ kill -s KILL 2615
```

输出:

```
2615
^Cinterrupt
interrupt
terminated
user defined signal 1
user defined signal 2
Killed: 9
```

如果仅捕获特定的信号，可以将其作为参数传递给 Notify 方法。比如下面的例子中，通过忽略 syscall.SIGINT 或 os.Interrupt 信号，避免 Ctrl + C 中断。在接收到退出信号时，可以使用自定义退出逻辑。

```
func main() {
    sig := make(chan os.Signal)
    signal.Notify(sig, os.Interrupt, syscall.SIGTERM)

    for {
        switch <-sig {
        case os.Interrupt:
            println("Interrupt")
        case syscall.SIGTERM:
            println("Terminated")
            os.Exit(0)
        }
    }
}
```

可以用 signal 代替 channel 阻塞 main 函数退出。

```
func waitExit() {
    sig := make(chan os.Signal)
    signal.Notify(sig, os.Interrupt, syscall.SIGTERM)

    for {
        switch <-sig {
        case os.Interrupt, syscall.SIGTERM:
             return                                  // 或 os.Exit(0)
        }
    }
}

func main() {
    ...
    waitExit()
}
```

28.5 user

在 Unix-like 环境下编程，免不了受帐号权限设置限制。
Current 返回当前帐号，Lookup 可按用户名或 uid 进行查找。

```
import (
    "fmt"
    "os/user"
)

func main() {
    u, _ := user.Current()
    fmt.Printf("%#v\n", u)
}
```

输出:

```
&user.User{Uid:"501", Gid:"20", Username:"yuhen", Name:"Q.yuhen", HomeDir:"/Users/yuhen"}
```

第 29 章 runtime

第 30 章 sort

第 31 章 strings

31.1 strconv

Append 系列函数将整数等转换为字符串后，添加到现有的字节数组中。

```
import (
    "strconv"
    "fmt"
)

func main() {
    bs := make([]byte, 0, 100)
    bs = strconv.AppendInt(bs, 4567, 10)
    bs = strconv.AppendBool(bs, false)
    bs = strconv.AppendQuote(bs, "abcxx")
    bs = strconv.AppendQuoteRune(bs, '我')

    fmt.Println(string(bs))
}
```

输出:

```
4567false"abcxx"'我'
```

Format 和 Parse 在字符串和整数类型间进行转换，需要提供进制参数。

```
fmt.Println(strconv.FormatInt(1234, 2))
fmt.Println(strconv.FormatInt(1234, 8))
fmt.Println(strconv.FormatInt(1234, 10))
fmt.Println(strconv.FormatInt(1234, 16))

fmt.Println(strconv.ParseInt("10011010010", 2, 0))
fmt.Println(strconv.ParseInt("2322", 8, 0))
fmt.Println(strconv.ParseInt("1234", 10, 0))
fmt.Println(strconv.ParseInt("4d2", 16, 0))
```

Parse 函数另需要提供 bitSize 参数，用于检查转换结果是否超出我们预期的限制。比如 bitSize = 8，那么转换结果就不应该超出 8 位整数限制，否则转换出错。(0 int; 8 int8; 16 int16; 32 int32; 64 int64)

```
fmt.Println(strconv.ParseInt("127", 10, 8))
fmt.Println(strconv.ParseInt("128", 10, 8))

fmt.Println(strconv.ParseUint("255", 10, 8))
fmt.Println(strconv.ParseUint("256", 10, 8))
```

输出:

```
127 <nil>
127 strconv.ParseInt: parsing "128": value out of range
255 <nil>
18446744073709551615 strconv.ParseUint: parsing "256": value out of range
```

Atoi 和 Itoa 是相关函数的简便封装版本。

31.2 strings

忽略大小写判断字符串是否相同可以用 EqualFold。

```
strings.EqualFold("Abc", "abc")
```

Fields 用于分割字符串，忽略多余的空格，提取有效字段。

```
fmt.Printf("%#v\n", strings.Fields("   name\tsex age       address"))
```

输出:

```
[]string{"name", "sex", "age", "address"}
```

没有 Startswith、Endswith，其对应函数是 HasPrefix 和 HasSuffix。

```
strings.HasPrefix("Abcde", "Ab")
strings.HasSuffix("Abcde", "cde")
```

SplitAfter 和 Split 的区别在于，前者会保留分隔符。

```
fmt.Printf("%#v\n", strings.Split("a,b,c,d", ","))
fmt.Printf("%#v\n", strings.SplitAfter("a,b,c,d", ","))
```

输出:

```
[]string{"a", "b", "c", "d"}
[]string{"a,", "b,", "c,", "d"}
```

31.3 template

Go 提供了 string 和 html 两套模板引擎，使用方法大同小异。

31.3.1 基本语法

基本语法构成很简单。

- 标记: {{ ... }}
- 变量: $name
- 成员: "." 访问 Key、Field、Method 成员。
- 迭代: {{range}}...{{end}}
- 判断: {{if}}...{{else}}...{{end}}
- 上下文: {{with}}...{{end}}

默认情况下，"$" 总是指向 Execute() 传入的上下文对象，"." 指向当前上下文对象。

```
import (
    "log"
    "os"
    "text/template"
)

func tmpl(text string, data interface{}) {
    t, _ := template.New("test").Parse(text)
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatalln(err)
    }
}

func main() {
    t := `
        Data:
            {{.}}
            {{$}}

        Var、Key:
            {{/* 注意变量使用 := 赋值 */}}
            {{$name := .name}} Data.Name = {{$name}}

        Range:
            Map  : {{range $key, $value := $}} {{$key}} = {{$value}}; {{end}}

            {{/* 此处的 . 代表当前上下文，也就是 item */}}
            Slice: {{range .ints}} {{.}}, {{end}}

            {{/* 索引序号 */}}
            Slice: {{range $index, $value := .ints}} [{{$index}}] = {{$value}}, {{end}}

            {{/* 用内置函数 index 访问指定序号的元素 */}}
            Index: ints[2] = {{index .ints 2}}

        With:
            {{/* with 定义块上下文，可以减少前缀输入 */}}
            {{with .meta}} {{.X}} {{end}}

        IF:
            {{/* 0、false、nil，len = 0 的 string、array、slice、map，或者不存在的成员都是 FALSE */}}
            {{if .ints}} Y {{else}} N {{end}}
            {{with .SomeOne}} Y {{else}} N {{end}}
    `

    d := map[string]interface{}{
        "name": "Q.yuhen",
        "ints": []int{1, 2, 3, 4},
        "meta": struct{ X int }{100},
    }

    tmpl(t, d)
}
```

输出:

```
Data:
    map[ints:[1 2 3 4] meta:{100} name:Q.yuhen]
    map[ints:[1 2 3 4] meta:{100} name:Q.yuhen]

Var、Key:
    Data.Name = Q.yuhen

Range:
    Map  :  ints = [1 2 3 4];  meta = {100};  name = Q.yuhen;
    Slice:  1,  2,  3,  4,
    Slice:  [0] = 1,  [1] = 2,  [2] = 3,  [3] = 4,
    Index: ints[2] = 3

With:
    100

IF:
    Y
    N
```

31.3.2 函数调用

在模版中调用函数，不需要括号和逗号分隔参数。

```
func main() {
    t := `
        {{ $x := len $ }} Length = {{$x}}  {{/* 用变量存储函数返回值 */}}
        {{ len $ | printf "Length = %d"}}  {{/* 用管道将返回值传递给下一个函数 */}}
    `

    tmpl(t, []int{0, 1, 2, 3})
}
```

除了内置函数，还可以直接调用数据对象方法和额外注入的函数。外部函数须提前注入。

```
type User struct {
    Id   int
    Name string
}

func (this User) ToString(s string) string {
    return s + " = " + this.Name
}

func FormatUser(f string, u *User) string {
    return fmt.Sprintf(f, u)
}

func main() {
    tmpl := template.New("test")
    tmpl.Funcs(template.FuncMap{"FormatUser": FormatUser})
```

```
    tmpl.Parse(`
        Method  : {{ .ToString "Username = " }}
        Function: {{ FormatUser "%#v" $ }}       {{/* 多个参数之间，无需逗号分隔 */}}
    `)

    tmpl.Execute(os.Stdout, &User{1, "User1"})
}
```

输出:

```
Method  : Username = User1
Function: &main.User{Id:1, Name:"User1"}
```

31.3.3 嵌套模板

模板可以包含多个子模板，以便形成嵌套输出。子模板要访问数据对象，则必须通过 {{template}} 参数传递。

```
func main() {
    root := `
        T1: {
            {{template "T1" .}}       {{/* 输出子模板内容，注意传递参数给子模板 */}}
        }

        T2: {
            {{template "T2" .age}}    {{/* 输出子模板内容，注意传递参数给子模板 */}}
        }
    `

    t1 := `
        Name: {{.name}}; Age: {{.age}}
    `

    t2 := `
        Age: {{.}}
    `

    tmpl, _ := template.New("root").Parse(root)
    tmpl.New("T1").Parse(t1)
    tmpl.New("T2").Parse(t2)

    d := map[string]interface{}{"name": "User1", "age": 18}
    tmpl.Execute(os.Stdout, d)
}
```

输出:

```
T1: {
    Name: User1; Age: 18
}

T2: {
    Age: 18
}
```

也可直接父模版中定义子模板，以实现复用和细分。

```
func main() {
    s := `
        {{define "T1"}}
            Name: {{.name}}; Age: {{.age}}
        {{end}}

        {{define "T2"}}
            Age: {{.}}
        {{end}}

        T1 {
            {{template "T1" .}}
        }

        T2 {
            {{template "T2" .age}}
        }
    `

    tmpl, _ := template.New("root").Parse(s)

    d := map[string]interface{}{"name": "User1", "age": 18}
    tmpl.Execute(os.Stdout, d)

    // 查看所有模版，包括父模版。
    for _, sub := range tmpl.Templates() {
        println(sub.Name())
    }
}
```

输出:

```
T1 {
    Name: User1; Age: 18
}

T2 {
    Age: 18
}

root
T1
T2
```

另有 ParseFiles 和 ParseGlob 函数用于载入模板文件。默认使用文件名作为模板名，并将第一参数作为根模板。

root.txt

```
Root:
    T1 {{template "t1.txt" .}}
    T2 {{template "t2.txt" .age}}
```

t1.txt

```
Name:{{.name}}; Age:{{.age}}
```

t2.txt

```
Age:{{.}}
```

```
func main() {
    tmpl, _ := template.ParseFiles("root.txt", "t1.txt", "t2.txt")

    d := map[string]interface{}{"name": "User1", "age": 18}
    tmpl.Execute(os.Stdout, d)
}
```

输出:

```
Root:
    T1 Name:User1; Age:18
    T2 Age:18
```

还可以直接在模板文件中使用 {{define}} 定义模板名以替代 "丑陋" 的文件名。

root.txt

```
{{define "Root"}}
Root:
    T1 {{template "T1" .}}
    T2 {{template "T2" .age}}
{{end}}
```

t1.txt

```
{{define "T1"}}
    Name:{{.name}}; Age:{{.age}}
{{end}}
```

t2.txt

```
{{define "T2"}}
    Age:{{.}}
{{end}}
```

不过这样一样，只能换用 ExecuteTemplate 了。

```
// t.Execute(os.Stdout, d)
t.ExecuteTemplate(os.Stdout, "Root", d)
```

31.4 regexp

Go 的正则表达式语法规则和 Python 基本相同，只是功能要弱许多，感觉是个半成品。
这个库的设计真的很难接受。

31.4.1. 基本语法

转义: . ^ $ * + ?{ } [ ] \ | ( )	
定义:	
\d	数字，相当于 [0-9]。
\D	非数字字符，相当于 [^0-9]。
\s	空白字符，相当于 [\t\r\n\f\v]。
\S	非空白字符。
\w	字母或数字，相当于 [0-9a-zA-Z]。
\W	非字母或数字。
.	任意字符。
|	或。
^	非，或者开始位置标记。
$	结束位置标记。
\b	单词边界。
\B	非单词边界。
重复:	
*	0 或任意多个字符。添加 ? 后缀避免贪婪匹配。
?	0 或一个字符。
+	1 或多个字符。
{n}	n 个字符。
{n,}	最少 n 个字符。
{,m}	最多 m 个字符。
{n, m}	n 到 m 个字符。
编译: 可以直接在表达式前部添加 "(?iLmsux)" 标志	
s	单行。
i	忽略大小写。

L	让 \w 匹配本地字符，对中文支持不好。
m	多行。
x	忽略多余的空白字符。
u	Unicode。

31.4.2 match

Regexp 对象提供了对 []byte 和 string 的不同操作，使用方法相同。

```
reg, _ := regexp.Compile(`\d{2,}`)
s := "ab0c1234d567x"

fmt.Println(reg.MatchString(s))
```

输出:

```
true
```

也可以直接使用简便函数。

```
m, _ := regexp.MatchString(`\d{2,}`, "ab0c1234d567x")
```

31.4.3 find

Find 返回首个匹配，FindAll 可指定最大匹配数量 (-1 全部)，匹配失败返回 "" 或 nil。

```
reg, _ := regexp.Compile(`\d{2,}`)
s := "ab0c1234d567x"

fmt.Printf("%#v\n", reg.FindString(s))
fmt.Printf("%#v\n", reg.FindAllString(s, -1))
```

输出:

```
"1234"
[]string{"1234", "567"}
```

FindIndex 返回匹配的 [开始, 结束] 位置。

```
if loc := reg.FindStringIndex(s); loc != nil {
    fmt.Printf("%#v, %#v\n", loc, s[loc[0]:loc[1]])
}

if locs := reg.FindAllStringIndex(s, -1); locs != nil {
    for _, loc := range locs {
        fmt.Printf("%#v %#v; ", loc, s[loc[0]:loc[1]])
    }
    fmt.Println()
}
```

输出：

```
[]int{4, 8}, "1234"
[]int{4, 8} "1234"; []int{9, 12} "567";
```

31.4.4 group

组操作不太方便，不知为什么没用 map 保存组结果，只能通过 SubexpNames 检索所有组，包括命名组。

```
reg, _ := regexp.Compile(`(?P<digit>\d{2,})(?P<letter>[a-z]+)`)
s := "ab0c1234d567x"

// 组序号，组名，匹配值
names := reg.SubexpNames()
for i, v := range reg.FindStringSubmatch(s) {
    fmt.Printf("%d, %s, %s\n", i, names[i], v)
}

// 多个匹配
for _, m := range reg.FindAllStringSubmatch(s, -1) {
    fmt.Printf("%v; ", m)
}
fmt.Println()

// 古里古怪的 [起始,结束] 位置信息。
fmt.Println(reg.FindStringSubmatchIndex(s))
fmt.Println(reg.FindAllStringSubmatchIndex(s, -1))
```

输出:

```
0, , 1234d                                  // $0
1, digit, 1234                              // $1
2, letter, d                                // $2

[1234d 1234 d]; [567x 567 x];

[4 9 4 8 8 9]
[[4 9 4 8 8 9] [9 13 9 12 12 13]]
```

除了 (?P<name>) 命名组和 (?:) 非捕获组外，貌似不支持反向引用和位置声明等用法。

```
reg, _ := regexp.Compile(`(?:\d{2,})(?P<letter>[a-z]+)`)   // 不会捕获第一个组，但依然包含在 $0 中。
s := "ab0c1234d567x"

fmt.Printf("%#v\n", reg.SubexpNames())
fmt.Printf("%#v\n", reg.FindAllStringSubmatch(s, -1))
```

输出:

```
[]string{"", "letter"}
[][]string{[]string{"1234d", "d"}, []string{"567x", "x"}}
```

31.4.5 replace

支持包含组引用的通用替换，函数替换，以及不解析组引用的 Literal 替换。

```
reg, _ := regexp.Compile(`(\d{2,})(?P<letter>[a-z]+)`)
s := "ab0c1234d567x"

fmt.Println("Submatch:", reg.FindAllStringSubmatch(s, -1))

// 使用组名或序号替换。组序号可以写成 ${n}，$$ 转义。
fmt.Println(reg.ReplaceAllString(s, " $letter$1 "))

// 不解析 $ 符号，原文替换。
fmt.Println(reg.ReplaceAllLiteralString(s, " $ "))

// 使用替换函数。
fmt.Println(reg.ReplaceAllStringFunc(s, func(x string) string { return " (" + x + ") " }))
```

输出:

```
Submatch: [[1234d 1234 d] [567x 567 x]]
ab0c d1234  x567
ab0c $  $
ab0c (1234d)  (567x)
```

31.4.6 expand

使用正则从字符串中提取匹配组，替换模板中的标记，并将结果追加到 dst 尾部。

```
ExpandString(dst []byte, template string, src string, match []int) []byte
```

match 参数是 FindSubmatchIndex 返回的 [起时, 结束] 位置信息。

```
reg, _ := regexp.Compile(`(\d{2,})(?P<letter>[a-z]+)`)
s := "ab0c1234d567x"

r := reg.ExpandString([]byte("abc"), "($1; $letter)", s, reg.FindStringSubmatchIndex(s))
fmt.Println(string(r))
```

输出:

```
abc(1234; d)
```

第 32 章 sync

虽说并发和同步是一对搭档，但强烈建议用 channel，尽可能避免使用 Lock。

32.1 lock

下面的例子中，不同的 goroutine 交错运行。

```
import (
    "fmt"
    "time"
    "strconv"
)

func main() {
    f := func(name string) {
        for i := 0; i < 3; i++ {
            fmt.Printf("[%s] %d\n", name, i)
        }
    }

    for i := 0; i < 2; i++ {
        go f("go" + strconv.Itoa(i))
    }

    time.Sleep(time.Second * 3)
}
```

输出:

```
[go0] 0
[go1] 0
[go0] 1
[go1] 1
[go0] 2
[go1] 2
```

借助于 sync.Mutex 这个 Locker，我们可以简单地实现排队。

```
import (
    "fmt"
    "time"
    "strconv"
    "sync"
)

func main() {
    m := new(sync.Mutex)

    f := func(name string) {
        m.Lock()
        defer m.Unlock()
```

```
        for i := 0; i < 3; i++ {
            fmt.Printf("[%s] %d\n", name, i)
        }
    }

    for i := 0; i < 2; i++ {
        go f("go" + strconv.Itoa(i))
    }

    time.Sleep(time.Second * 3)
}
```

输出:

```
[go0] 0
[go0] 1
[go0] 2
[go1] 0
[go1] 1
[go1] 2
```

注意：不支持递归锁。因为多个 goroutine 可能运行在同一线程，因此基于线程实现的递归锁是无法使用的。

另一个 Locker 实现是 RWMutex，通常称为读写锁。当写操作获得锁时，所有读操作锁被阻塞。但所有读操作锁之间不会阻塞互斥，完全是并发行为，也就是说读锁不过是检查写锁状态而已。同样，想获得写锁的前提是所有读锁被释放。

```
import (
    "fmt"
    "time"
    "sync"
)

func main() {
    m := new(sync.RWMutex)

    r := func(x int) {
        fmt.Printf("[%d] RLock...\n", x)

        // 等待写锁释放。
        m.RLock()
        defer m.RUnlock()

        for n := 0; n < 2; n++ {
            fmt.Printf("[%d] %d\n", x, n)
            time.Sleep(time.Second * 1)
        }
    }

    // 写锁。
    m.Lock()
```

```
    // 启动多个读 goroutine。
    for i := 0; i < 3; i++ {
        go r(i)
    }

    time.Sleep(time.Second * 1)

    // 释放写锁，使得多个读锁可以进行。
    fmt.Println("Unlock...")
    m.Unlock()

    // 再次请求写锁，必须等到所有读锁释放。
    m.Lock()
    fmt.Println("Lock...")
}
```

输出：

```
[0] RLock...
[1] RLock...
[2] RLock...
Unlock...
[0] 0
[1] 0
[2] 0
[0] 1
[1] 1
[2] 1
Lock...
```

32.2 cond

Cond 在 Locker 的基础上增加了一种 "通知" 机制。其内部通过一个计数器和信号量来实现通知和广播的效果。

- 当调用 Wait 时，内部计数器增加，然后阻塞，请求系统信号量。
- 当调用 Signal 时，计数器 -1，释放信号量，使某个 Wait goroutine 解除阻塞。
- 当调用 Broadcast 时，释放和计数器等量的信号量，使得所有 Wait goroutine 解除阻塞。

那为什么还需要外部传入一个 Locker 呢？同时用 cond.L 对 Wait 和 Signal/Broadcast 进行保护，可以确保：

- 在发送信号的同时，不会有新的 goroutine 进入 Wait。
- 在 Wait 逻辑未完成前，不会有新的事件发生。

需要注意的是，在调用 Signal、Broadcast 前，应该确保目标进入 Wait 阻塞状态。

对 Wait 使用 L.Lock 是必须的 (该函数内部调用 L.Lock，不保护会导致出错)。Wait 内部在请求信号量进入阻塞前，会释放保护锁，使得其他代码可以获得保护锁 (比如用来做事件通知操作)。在获得信号量后，会自动调用 L.Lock 再次获取保护锁。

```
import (
    "fmt"
    "time"
    "sync"
)

func main() {
    locker := new(sync.Mutex)
    cod := sync.NewCond(locker)

    for i := 0; i < 2; i++ {
        go func(x int) {
            cod.L.Lock()
            cod.Wait()
            cod.L.Unlock()

            for n := 0; n < 3; n++ {
                fmt.Printf("[%d] %d...\n", x, n)
            }
        }(i)
    }

    // 等待全部进入 Wait 状态
    time.Sleep(time.Second * 1)

    cod.L.Lock()
    cod.Broadcast()
    cod.L.Unlock()

    time.Sleep(time.Second * 5)
}
```

输出:

```
[0] 0...
[1] 0...
[1] 1...
[0] 1...
[1] 2...
[0] 2...
```

我们可以调整 Wait 的 Lock 保护范围，使得接收到广播的 goroutine 顺序执行。

```
func main() {
    locker := new(sync.Mutex)
    cod := sync.NewCond(locker)

    for i := 0; i < 2; i++ {
        go func(x int) {
            cod.L.Lock()
            cod.Wait()
            defer cod.L.Unlock()   // !!!!

            for n := 0; n < 3; n++ {
```

```
                fmt.Printf("[%d] %d...\n", x, n)
            }
        }(i)
    }

    time.Sleep(time.Second * 1)

    cod.L.Lock()
    cod.Broadcast()
    cod.L.Unlock()

    time.Sleep(time.Second * 5)
}
```

输出：

```
[0] 0...
[0] 1...
[0] 2...
[1] 0...
[1] 1...
[1] 2...
```

上面的例子中，尽管对 Broadcast 的保护不是必须的，但依然建议这么做。老实说，这个 Cond 用起来很别扭，远没有 Python 中的库好用、自然。完全可以把这个外部 Lock 交给开发人员自行处理，反正 Cond 内部对计数器已经做了同步保护。

32.3 once

Once.Do 可以确保目标仅会被执行一次。用来做并发环境初始化挺好的。

```
import (
    "fmt"
    "time"
    "sync"
)

func main() {
    once := new(sync.Once)

    for i := 0; i < 3; i++ {
        go func(x int) {
            once.Do(func() {
                fmt.Printf("once %d\n", x)
            })

            fmt.Printf("%d ...\n", x)
        }(i)
    }

    time.Sleep(time.Second * 3)
}
```

输出:

```
once 0
0 ...
1 ...
2 ...
```

32.4 wait

Go 进程并不会等待所有 goroutine 退出，因此要实现一个完整的并发逻辑，必须让主进程等待。通常的做法是用一个 channel 等待结束通知，但多数时候，我们都无法确定哪个 goroutine 是最后完成的。只好用丑陋的 time.Sleep。

现在有了 WaitGroup，这个问题就比较好处理了。

```
import (
    "fmt"
    "sync"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    wg := sync.WaitGroup{}

    for i := 0; i < 3; i++ {
        wg.Add(1)

        go func(x, n int) {
            for y := 0; y < n; y++ {
                fmt.Printf("[%d]: %d\n", x, y)
            }

            wg.Done()
        }(i, rand.Intn(5))
    }

    wg.Wait()
}
```

输出:

```
[0]: 0
[1]: 0
[2]: 0
[0]: 1
[2]: 1
[2]: 2
[2]: 3
```

32.5 atomic

sync/atomic 包可以确保对目标数据进行原子操作，但包中也提到这样一句话。

> These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package. Share memory by communicating; don't communicate by sharing memory.

第 33 章 syscall

33.1 fork

在 Unix-like 下编程，没有 fork 自然是不行滴。

```
import (
    "os"
    "runtime"
    "syscall"
    "time"
)

func fork() (int, syscall.Errno) {
    r1, r2, err := syscall.RawSyscall(syscall.SYS_FORK, 0, 0, 0)
    if err != 0 { return 0, err }

    if runtime.GOOS == "darwin" && r2 == 1 { r1 = 0 }
    return int(r1), 0
}

func main() {
    println(os.Getpid())

    pid, _ := fork()

    if pid == 0 {
        println("child:", os.Getpid(), os.Getppid())
        time.Sleep(time.Second * 5)
        os.Exit(-1)
    } else {
        println("parent:", os.Getpid(), pid)

        p, _ := os.FindProcess(pid)
        status, _ := p.Wait()
        println("parent: child exit", status.Success())
    }
}
```

输出:

```
1496
parent: 1496 1498
child: 1498 1496
parent: child exit false
```

os.FindProcess Wait 自然没有 waitpid(-1,,,) 等待所有子进程方便，可问题是 syscall 里面只有一个在 BSD/Darwin 下正常工作的 Wait4 函数。也不知道 Go 开发人员是怎么想的。

33.2 daemon

除了使用命令行工具 nohup、daemon 外，自带 Daemon 功能貌似更好一些。

```
import (
    "log"
    "os"
    "runtime"
    "syscall"
    "time"
)

func fork() (int, syscall.Errno) {
    r1, r2, err := syscall.RawSyscall(syscall.SYS_FORK, 0, 0, 0)
    if err != 0 { return 0, err }

    if runtime.GOOS == "darwin" && r2 == 1 { r1 = 0 }
    return int(r1), 0
}

func daemon() {
    pid, err := fork()
    if err != 0 { log.Fatalln("Daemon Error!") }

    if pid != 0 { os.Exit(0) }

    syscall.Umask(0)
    syscall.Setsid()
    os.Chdir("/")

    f, _ := os.Open("/dev/null")
    devnull := f.Fd()

    syscall.Dup2(int(devnull), int(os.Stdin.Fd()))
    syscall.Dup2(int(devnull), int(os.Stdout.Fd()))
    syscall.Dup2(int(devnull), int(os.Stderr.Fd()))
}

func main() {
    println(os.Getpid())

    daemon()

    for {
        time.Sleep(time.Second * 1)
    }
}
```

第 34 章 time

对于 Go time format 格式定义，实在有点不习惯。

34.1 datetime

可以用 RFC3339 格式解析常见时间字符串，或者直接以数字创建。

```
t, _ := time.Parse("2006-01-02 15:04:05 Z0700", "2012-06-09 00:08:32 +0800")     // 格式化解析
fmt.Println(t)

fmt.Println(t.Format("2006-01-02 15:04:05"))                                      // 格式化输出
```

输出:

```
2012-06-09 00:08:32 +0800 CST
2012-06-09 00:08:32
```

RFC3339 格式里的 magic 数字含义，可以参考 pkg/time/format.go 里头部 const 定义。

```
stdLongMonth      = iota + stdNeedDate  // "January"
stdMonth                                // "Jan"
stdNumMonth                             // "1"
stdZeroMonth                            // "01"
stdLongWeekDay                          // "Monday"
stdWeekDay                              // "Mon"
stdDay                                  // "2"
stdUnderDay                             // "_2"
stdZeroDay                              // "02"
stdHour           = iota + stdNeedClock // "15"
stdHour12                               // "3"
stdZeroHour12                           // "03"
stdMinute                               // "4"
stdZeroMinute                           // "04"
stdSecond                               // "5"
stdZeroSecond                           // "05"
stdLongYear       = iota + stdNeedDate  // "2006"
stdYear                                 // "06"
stdPM             = iota + stdNeedClock // "PM"
stdpm                                   // "pm"
stdTZ             = iota                // "MST"
stdISO8601TZ                            // "Z0700"  // prints Z for UTC
stdISO8601ColonTZ                       // "Z07:00" // prints Z for UTC
stdNumTZ                                // "-0700"  // always numeric
stdNumShortTZ                           // "-07"    // always numeric
stdNumColonTZ                           // "-07:00" // always numeric
stdFracSecond0                          // ".0", ".00", ... , trailing zeros included
stdFracSecond9                          // ".9", ".99", ..., trailing zeros omitted
```

获取本地时区注意 "Local" 大小写，否则出错。Time.Zone 方法可获取时区及时间差。另提供了相关方法，可以简单在 UTC 和本地时区间转换，还可以用 In 方法转换为任意时区时间。

```
loc, _ := time.LoadLocation("Local")
lt := time.Date(2012, 6, 9, 0, 8, 32, 0, loc)

ut := lt.UTC()
lt2 := ut.Local()

fmt.Println(lt)
fmt.Println(ut)
fmt.Println(lt2)
```

输出:

```
2012-06-09 00:08:32 +0800 CST
2012-06-08 16:08:32 +0000 UTC
2012-06-09 00:08:32 +0800 CST
```

身在北京，要早 8 小时迎接 2012-06-09 这天。

除了使用 Year、Month、Day、Hour、Minute、Second、Date、Clock 这些方法获取相关字段外，我们常用的做法是将其转换为 Unix Timestamp。

```
loc, _ := time.LoadLocation("Local")
t := time.Date(2012, 6, 9, 0, 8, 32, 0, loc)

utSencond := t.Unix()
utNanoSecond := t.UnixNano()
t2 := time.Unix(utSencond, 0)

fmt.Println(utSencond)
fmt.Println(utNanoSecond)
fmt.Println(t2)
```

输出:

```
1339171712
1339171712000000000
2012-06-09 00:08:32 +0800 CST
```

Unix 返回 UTC 时间，在 Python 中，可以用 datetime.fromtimestamp 或 time.localtime 转换为本地时间。

Sub 用于计算时间差，Add 和 AddDate 则是调整时间 (支持负数)。至于 Before、After、Equal 自然是比较两个时间的先后或者相等了。

```
loc, _ := time.LoadLocation("Local")

t1 := time.Date(2012, 6, 9, 0, 8, 32, 0, loc)
t2 := time.Date(2012, 6, 9, 0, 9, 32, 0, loc)

d := t2.Sub(t1)
fmt.Println(d)

fmt.Println(t1.Add(d))
fmt.Println(t2.AddDate(-1, 0, 0))
```

```
fmt.Println(t1.Before(t2))
fmt.Println(t2.After(t1))
fmt.Println(t1.Add(d).Equal(t2))
```

输出:

```
1m0s
2012-06-09 00:09:32 +0800 CST
2011-06-09 00:09:32 +0800 CST
true
true
true
```

34.2 duration

Duration 是一个 64 位整数，用以记录一段以纳秒为单位的时长。

```
t := time.Now()
fmt.Println(t)

var d time.Duration = time.Hour * 1 + time.Second * 10
t2 := t.Add(d)
fmt.Println(t2)
```

输出:

```
2012-06-09 00:54:23.989535 +0800 CST
2012-06-09 01:54:33.989535 +0800 CST
```

另提供了相关方法，用不同的计量单位来表示 Duration。

```
var d time.Duration = time.Hour * 1 + time.Second * 10

fmt.Println(d.Hours())               // 总计多少小时
fmt.Println(d.Seconds())             // 总计多少秒
```

输出:

```
1.0027777777777778
3610
```

可以用更自然一点的方法来表达: 时 h、分 m、秒 s、毫秒 ms、微秒 us、纳秒 ns。

```
d, _ := time.ParseDuration("1h2m3s4ms5us6ns")

fmt.Println(d)
fmt.Println(d.Nanoseconds())
```

输出:

```
1h2m3.004005006s
3723004005006
```

要知道从某个时候到现在共过去多少时间，不妨试试 Since。

```
t1 := time.Now()
time.Sleep(time.Second * 2)
```

```
fmt.Println(time.Since(t1))
```

输出:

```
2.001021s
```

Duration.String() 输出样式实在别扭，又没找到 layout 控制，只好自己写一个。

```
func DurationFormat(d time.Duration) string {
    ts := d.Nanoseconds() / (1000 * 1000 * 1000)
    h, v := ts / 3600, ts % 3600
    m, s := v / 60, v % 60
    return fmt.Sprintf("%02d:%02d:%02d", h, m, s)
}
```

34.3 timer

Timer 和 Ticker 使用一个 channel 来触发计时器事件。

```
t := time.NewTimer(time.Second * 1)
fmt.Println(<-t.C)
```

Timer 事件仅被触发一次，在触发前可以调用 Stop 方法取消事件。或者用 AfterFunc 替代 NewTimer 创建计时器，在事件触发时执行特定函数。Ticker 则一直运作，直到 Stop 被调用。

其实最常用是 After 和 Tick 这两个工厂函数，它们创建计时器，并直接返回 Channel。

```
a := time.After(time.Second * 10)
t := time.Tick(time.Second * 1)

for {
    select {
    case <-a:
        fmt.Println("timeout")
        os.Exit(1)
    case v, _ := <-t:
        fmt.Println(v)
    }
}
```

输出:

```
2012-06-09 12:40:39.306083 +0800 CST
2012-06-09 12:40:40.306652 +0800 CST
2012-06-09 12:40:41.306612 +0800 CST
2012-06-09 12:40:42.30664 +0800 CST
2012-06-09 12:40:43.306576 +0800 CST
2012-06-09 12:40:44.306556 +0800 CST
2012-06-09 12:40:45.307022 +0800 CST
2012-06-09 12:40:46.30666 +0800 CST
2012-06-09 12:40:47.306417 +0800 CST
timeout
```

Tick 才是我们习惯的计时器类型。注意每次调用 After 和 Tick 都会创建新的对象和 Channel，如果时间参数为 0，则直接返回 nil。

附录

A. Go 源码阅读指南

基于官方 Golang 1.1 Beta (2013/04) 版本。

1. 内存分配

使用基于 TCMalloc 的内存分配器，架构上也基本照搬。

相关信息可查看 src/pkg/runtime/malloc.h 头部注释。

相关源码文件 malloc.goc、mheap.c、mcentral.c、mem_linux.c 等。

1.1 内存块粒度

- **page**: 内存块最小单位，默认 4KB。
- **span**: 多个地址连续的 page 构成一个 span，以提供更大容量的内存块。
- **object**: 存储单一对象的内存块，按照大小和对齐方式，有多种规格 (class)。

class 规格从 1 到 60，其对应的容量大小存储在 class_to_size 表中。

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
// class_to_transfercount[i] = number of objects to move when
//	taking a bunch of objects out of the central lists
//	and putting them in the thread free list.
```

```
int32   runtime·SizeToClass(int32);
extern  int32   runtime·class_to_size[NumSizeClasses];
```

当 class = 60 时，maxsize = 32768。故以 32KB 为界，将对象分为大小对象。malloc.goc / mallocgc() 中，小对象从 cache 中分配，大对象直接从 heap 分配。

使用 runtime.SizeToClass() 函数可以计算对象所对应的 class。

与 class 有关的还有两个表：

- class_to_allocnpages: 分配该 class 规格对象所需的 page 数量，以便确定目标 span 大小。
- class_to_transfercount: 当 cache 空间不够时，从 central 批量获取内存块的数量。

1.2 Heap: 提供 span 粒度内存块

malloc.h

```
PageShift    = 12
MaxMHeapList = 1<<(20 - PageShift),  // Maximum page length for fixed-size list in MHeap.

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

struct MSpan
{
        MSpan  *next;         // in a span linked list
        MSpan  *prev;         // in a span linked list

        MLink  *freelist;     // list of free objects
        uint32  ref;          // number of allocated objects in this span
        int32   sizeclass;    // size class
};
```

- 所有内存都由 heap 从 OS 申请。不足时进行扩张，每次最少 1MB。
- free 数组管理多个链表，每个链表由同一大小 (page 数量相同) 的 span 构成。
- 只需按 free[pages] 即可访问对应大小规格的链表来获取 span。
- 如果 span pages 数量超出 free 限制，则放到 large 链表中。

申请内存：

(1) 根据所需 pages 数量从 free 对应链表获取。如没有合适的，则寻找更大尺寸的 span。
(2) 如果所有 free 链表都不满足，就去 large 里找 "大小合适，地址靠前" 的 span。
(3) 再不行的话，就让 heap 扩张内存 (新的 large span)，然后再从 large 里获取。
(4) 返回 span 前，会将多余 pages 切分组成新的 span，放到合适的 free 链表。

释放内存：

(1) 如果发现相邻的 span 也未使用 (位图)，合并成更大的 span。

(2) 根据处理后的 span 大小，插入到合适的 free 链表，或者超限放到 large。

(3) 未实现将过多的 span 还给 OS 功能，目前由 scavenge() 完成物理内存释放 (参见 GC)。

1.3 Central: 提供 object 粒度内存块

malloc.h

```
// Central list of free objects of a given size.
struct MCentral
{
        int32 sizeclass;
        MSpan nonempty;
        MSpan empty;
};

struct MHeap
{
        union {
                MCentral;
                byte pad[CacheLineSize];
        } central[NumSizeClasses];
};

struct MSpan
{
        MSpan  *next;         // in a span linked list
        MSpan  *prev;         // in a span linked list

        MLink  *freelist;     // list of free objects
        uint32 ref;           // number of allocated objects in this span
        int32  sizeclass;     // size class
};
```

- 将从 heap 获取的 span 按 class "切分" 成 object 粒度，保存在 span.freelist。
- 获取 span 前，需查 class_to_allocnpages 表确定所需 span 大小。
- heap.central[n] 管理多个不同 class 规格的 central 对象。
- 有两个链表，分别管理没有剩余空间 (empty) 和 还有剩余空间 (noempty) span。

分配内存：

(1) 按所需大小，从 heap.central[n] 找 central，然后从 noempty.span.freelist 提取。

(2) 提取后，如果 span.freelist 为空，表示没有剩余空间，将其移到 central.empty 链表。

(3) 每分出一个 object，那么该 span.ref++，释放 object 时 span.ref--。

释放内存：

(1) 找到所释放 object 对应的 span。

(2) 将 object "添加" 到 span.freelist。

(3) 如果需要，将 span 移到 central.noempty 链表。

(4) 如果 span 完全释放 (span.ref == 0)，则将其归还给 heap。

1.4 Cache: 线程对象池

结构定义: malloc.h

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

- 线程从该处申请内存时，无需加锁，可获得最佳效率。
- cache 为线程私有，central 则是多线程共享，需要使用自旋锁。
- cache 只是为线程提供内存分配，与对象多线程同步无关。
- cache.list[n] 管理多个不同 class 规格的 object freelist。

分配内存:

(1) 计算 object class，从 cache.list[n] 中获取对应的 cachelist，提取 object 内存块。

(2) 如果 cachelist 为空，就从对应大小的 central.noempty.span.freelist 批量取些过来。

(3) 批量提取数量从 class_to_transfercount 查表获得。

释放内存:

(1) 找到对应 class 的 cachelist，将 object 添加到该链表。

(2) 如果 cachelist 太长 (256) 或占用内存太多 (2MB)，则返还部分 object 给 central。

1.5 Malloc: 为对象分配内存

malloc.goc

```
void* runtime·malloc(uintptr size)
{
	return runtime·mallocgc(size, 0, 0, 1);
}

// Allocate an object of at least size bytes.
// Small objects are allocated from the per-thread cache's free lists.
// Large objects (> 32 kB) are allocated straight from the heap.
void* runtime·mallocgc(uintptr size, uint32 flag, int32 dogc, int32 zeroed)
{
	// 当前线程 MCache
	c = m->mcache;
```

```
        // 判断对象大小是否超过 32KB。
        // malloc.h: MaxSmallSize = 32<<10
        if(size <= MaxSmallSize) {
                // 计算 object 对应的 class 级别。
                sizeclass = runtime·SizeToClass(size);

                // 查表获取该 class 级别对应的 object 大小。
                size = runtime·class_to_size[sizeclass];

                // 从 MCache 分配内存。
                v = runtime·MCache_Alloc(c, sizeclass, size, zeroed);
        } else {
                // 计算所需页数。
                npages = size >> PageShift;

                // 直接从 MHeap 分配内存。
                s = runtime·MHeap_Alloc(runtime·mheap, npages, 0, 1, zeroed);
                v = (void*)(s->start << PageShift);
        }

        return v;
}
```

2. 垃圾回收

2.1 触发垃圾回收

为对象分配内存时可能会引发垃圾回收行为。

malloc.goc

```
void* runtime·mallocgc(uintptr size, uint32 flag, int32 dogc, int32 zeroed)
{
        ... ...

        // 检查 dogc 标记和 next_gc 阀值。(为大多数类型分配内存时，dogc = 1)
        if(dogc && mstats.heap_alloc >= mstats.next_gc)
                runtime·gc(0);
}
```

垃圾回收实际过程:

```
mallocgc() -> runtime.gc() -> gc() -> runfinqv() -> runtime.free() -> MCache -> MCentral -> MHeap 回收
```

runtime.gc() 会检查 gopercent (环境变量 GOGC)，如果 < 0，则放弃回收。

mgc0.c

```
void runtime·gc(int32 force)
{
        // 默认 gcpercent = 100
        if(gcpercent == GcpercentUnknown) {  // first time through
```

```
                gcpercent = readgogc();
        }

        if(gcpercent < 0) return;

        // Run gc on a bigger stack to eliminate
        gcv.fn = (void*)gc;
        reflect·call(&gcv, (byte*)&ap, sizeof(ap));
}

#define GcpercentUnknown (-2)

// Initialized from $GOGC.  GOGC=off means no gc.
//
// Next gc is after we've allocated an extra amount of
// memory proportional to the amount already in use.
// If gcpercent=100 and we're using 4M, we'll gc again
// when we get to 8M.  This keeps the gc cost in linear
// proportion to the allocation cost.  Adjusting gcpercent
// just changes the linear constant (and also the amount of
// extra memory used).
static int32 gcpercent = GcpercentUnknown;

static int32 readgogc(void)
{
        byte *p;

        p = runtime·getenv("GOGC");
        if(p == nil || p[0] == '\0') return 100;
        if(runtime·strcmp(p, (byte*)"off") == 0) return -1;
        return runtime·atoi(p);
}
```

gc() 会检查 heap_alloc >= next_gc，以此决定是否开始回收。

mgc0.c

```
static void gc(struct gc_args *args)
{
        // 检查强制回收标记和内存分配阀值。
        if(!args->force && mstats.heap_alloc < mstats.next_gc) {
                return;
        }

        // 准备回收...
        runtime·stoptheworld();

        // 一堆状态统计...

        // 获取并行 gc 所需的 goroutine 数量。
        work.nproc = runtime·gcprocs();

        // 开始 mark、sweep ...
        runtime·parfordo(work.markfor);
```

```
        runtime·parfordo(work.sweepfor);

        // 计算下次垃圾回收阀值。
        mstats.next_gc = mstats.heap_alloc+mstats.heap_alloc*gcpercent/100;

        if(finq != nil) {
                // 创建 goroutine 开始清理内存 ...
                if(fing == nil)
                        fing = runtime·newproc1(&runfinqv, nil, 0, 0, runtime·gc);
        }

        // 恢复 ...
        runtime·starttheworld();

        // give the queued finalizers, if any, a chance to run
        if(finq != nil) runtime·gosched();
}

static FuncVal runfinqv = {runfinq};

static void runfinq(void)
{
        for(;;) {
                for(; ...) {
                        for(i=0; ...) {
                                if(framecap < framesz) {
                                        runtime·free(frame);
                                }
                        }
                }
                runtime·gc(1); // trigger another gc to clean up the finalized objects, if possible
        }
}
```

malloc.goc

```
// Free the object whose base pointer is v.
void runtime·free(void *v)
{
        // Find size class for v.
        sizeclass = s->sizeclass;
        c = m->mcache;

        if(sizeclass == 0) {
                // Large object.
                runtime·MHeap_Free(runtime·mheap, s, 1);
        } else {
                // Small object.
                size = runtime·class_to_size[sizeclass];
                runtime·MCache_Free(c, v, sizeclass, size);
        }
}
```

回收结束，会计算下次阀值: next_gc = heap_alloc + heap_alloc * gcpercent / 100。默认是当前所分配内存的 2 倍，不过强制回收操作无视该阀值。所回收内存逐步合并到上层内存管理单元，终点是 heap，并没有释放给操作系统。

也可调用标准库 runtime/GC() 函数主动引发回收。

2.2 并行垃圾回收

并行 GC goroutine 数量由 gcprocs() 函数决定。

proc.c

```
int32 runtime·gcprocs(void)
{
	int32 n;

	// Figure out how many CPUs to use during GC.
	// Limited by gomaxprocs, number of actual CPUs, and MaxGcproc.
	runtime·lock(&runtime·sched);
	n = runtime·gomaxprocs;
	if(n > runtime·ncpu)
		n = runtime·ncpu;
	if(n > MaxGcproc)
		n = MaxGcproc;
	if(n > runtime·sched.nmidle+1) // one M is currently running
		n = runtime·sched.nmidle+1;
	runtime·unlock(&runtime·sched);
	return n;
}
```

2.3 强制回收和物理释放

mheap.c

```
// Release (part of) unused memory to OS.
// Goroutine created at startup.
// Loop forever.
void runtime·MHeap_Scavenger(void)
{
	// 关键指标1：超过 2 分钟没有内存回收，那么强制执行一次。
	// If we go two minutes without a garbage collection, force one to run.
	forcegc = 2*60*1e9;

	// 关键指标2: 如果一个 span 超过 5 分钟都未使用，那么将它还给操作系统。
	// If a span goes unused for 5 minutes after a garbage collection,
	// we hand it back to the operating system.
	limit = 5*60*1e9;

	// 计算每次循环休眠时间，2 个指标谁大用谁，时间是 1/2 。
	// Make wake-up period small enough for the sampling to be correct.
	if(forcegc < limit)
```

```
                tick = forcegc/2;
        else
                tick = limit/2;

        h = runtime·mheap;
        for(k=0;; k++) {
                // 休眠
                runtime·notetsleep(&note, tick);

                // 计算当前时间和上次垃圾回收时间差
                now = runtime·nanotime();
                if(now - mstats.last_gc > forcegc) {
                        // 创建 goroutine 执行 forcegchelperv。
                        runtime·newproc1(&forcegchelperv, ...);
                }

                // 执行 scavenge，强制释放超时不用的 span。
                sumreleased = scavenge(now, limit);
        }
}

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

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

需要关心一下 scavenge()，这才是真正释放物理内存的所在。

mheap.c

```
static uintptr scavenge(uint64 now, uint64 limit)
{
        h = runtime·mheap;
        sumreleased = 0;

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

        // 处理 heap.large 链表。
        sumreleased += scavengelist(&h->large, now, limit);
        return sumreleased;
}

static uintptr scavengelist(MSpan *list, uint64 now, uint64 limit)
{
        // 如果是 central.empty，那么表示该链表上的 span 都在使用，跳过。
        if(runtime·MSpanList_IsEmpty(list))
                return 0;

        sumreleased = 0;
```

```
        // 遍历链表上所有 span。
        for(s=list->next; s != list; s=s->next) {
                // 如果 当前时间 - 未使用开始时间 > 5分钟
                if((now - s->unusedsince) > limit) {
                        // 这个才是释放物理内存的关键。
                        runtime·SysUnused((void*)(s->start << PageShift), s->npages << PageShift);
                }
        }
        return sumreleased;
}
```

mem_linux.c

```
void runtime·SysUnused(void *v, uintptr n)
{
        // 建议内核释放该内存，也就是交欢给 OS。
        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);
}
```

可调用标准库 runtime/debug.FreeOSMemory() 让强制回收、释放内存。

2.4 GOGCTRACE

使用环境变量 GOGCTRACE，可以输出垃圾回收处理信息。

输出信息含义如下:

mheap.c: MHeap_Scavenger()

```
scvg9: inuse: 0, idle: 4, sys: 5, released: 4, consumed: 0 (MB)
回收次数:
正在使用的内存 (MB)
空闲内存
从系统总共申请的内存
本次释放内存
剩余使用内存
```

mgc0.c: gc()

```
gc334(1): 0+0+0 ms, 4 -> 1 MB 40 -> 37 (1369-1332) objects, ...
回收次数 (并行 gc goroutine 数量)
mark+sweep+cleanup 耗费时间
```

回收前后 Heap 已分配且正在使用的内存大小

回收前后剩余对象数量 (已分配内存块 - 空闲内存块 数量)

累计执行分配和释放操作次数

3. Goroutine

3.1 相关概念

- G: Goroutine
- M: Worker Thread
- P: Processor

P 抽象处理器，数量总是和 GOMAXPROCS 相同。M 执行时必须获取 P，以此控制并发执行。

3.2 GOMAXPROCS

GOMAXPROCS 最大值 256 (1<<8)，默认为 1 (proc.c 122, schedinit)。

runtime.GOMAXPROCS(n) 设置 n 个并发线程，同时返回设置前的值。

runtime.GOMAXPROCS(0) 返回当前设置值。

gomaxprocsfunc() 调整 newprocs 值，然后调用 starttheworld() -> procresize() 完成 P 数量调整。

进程启动，会有两个默认 goroutine:

- main: 执行 main.main() 入口函数。
- scavenger: 执行 MHeap_Scavenger 用于强制垃圾回收。

proc.c

```
// The main goroutine.
void runtime·main(void)
{
        runtime·newproc1(&scavenger, nil, 0, 0, runtime·main);
        main·init();
        main·main();
        runtime·exit(0);
}
```

3.3 G 和 M 创建流程

proc.c

```
G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 nret, void *callerpc)
{
    byte *sp;
```

```
    G *newg;

    // 创建新的 G (如果 free-list 有重用对象，那么直接拿一个)
    if((newg = gfget(m->p)) != nil) {
    } else {
        newg = runtime·malg(StackMin);
    }

    // G 通过 runtime.malloc() 分配内存，G.stack 使用 mallocgc() 分配，都是 MCache。

    newg->sched.sp = (uintptr)sp;                   // 栈顶
    newg->sched.pc = (byte*)runtime·goexit;         // 退出函数
    newg->sched.g = newg;
    newg->fnstart = fn;                             // goroutine func
    newg->gopc = (uintptr)callerpc;                 // 调用 newproc1 的函数地址
    newg->status = Grunnable;                       // 可运行状态

    // 放入可运行队列
    runqput(m->p, newg);

    // 调度 M
    if(runtime·atomicload(&runtime·sched.npidle) != 0 ...
        wakep();

    return newg;
}

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

// 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)
{
    M *mp;
    void (*fn)(void);

    // 从 idle-list 获取一个 P，如果没有的话，直接退出。
    if(p == nil) {
        p = pidleget();
        if(p == nil) return;
    }

    // 有可用的 P，获取一个空闲的 M。
    mp = mget();

    // 如果没有空闲 M，则新建。
    if(mp == nil) {
        newm(fn, p);
        return;
    }
}
```

```
// 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)
{
    M *mp;
    mp = runtime·allocm(p);
    mp->nextp = p;
    mp->mstartfn = fn;

    runtime·newosproc(mp, (byte*)mp->g0->stackbase);
}

// os_darwin.c
void runtime·newosproc(M *mp, void *stk)
{
    // 创建线程 Thread(mstart)。
    errno = runtime·bsdthread_create(stk, mp, mp->g0, runtime·mstart);
}

// os_linux.c
void runtime·newosproc(M *mp, void *stk)
{
    // 创建线程 Thread(mstart)。sys_linux_386.s 261
    ret = runtime·clone(flags, stk, mp, mp->g0, runtime·mstart);
}
```

3.4 M 执行流程

```
// proc.c 455: Called to start an M.
void runtime·mstart(void)
{
    schedule();
}

// proc.c 1121: One round of scheduler: find a runnable goroutine and execute it. Never returns.
static void schedule(void)
{
    G *gp;

    // 从可运行队列获取一个 G。没有的话，就从其他 P 的队列，或者全局队列中找。
    gp = runqget(m->p);
    if (gp == nil) gp = findrunnable();

    execute(gp);
}

// proc.c 958: Schedules gp to run on the current M. Never returns.
static void execute(G *gp)
{
    // 修改状态
    gp->status = Grunning;
    m->p->tick++;
    m->curg = gp;
```

```
    gp->m = m;

    if(gp->sched.pc == (byte*)runtime·goexit)
        runtime·gogocallfn(&gp->sched, gp->fnstart);       // fnstart 就是 goroutine func。

    runtime·gogo(&gp->sched, 0);                           // gogo() call goexit()。
}

// asm_386.s
// void gogocallfn(Gobuf*, FuncVal*); restore state from Gobuf but then call fn.
// (call fn, returning to state in Gobuf)
TEXT runtime·gogocallfn(SB), 7, $0
    MOVL    8(SP), DX                       // fnstart，也就是 goroutine func
    MOVL    4(SP), BX                       // gobuf, G.sched
    MOVL    gobuf_g(BX), DI
    get_tls(CX)
    MOVL    DI, g(CX)
    MOVL    0(DI), CX
    MOVL    gobuf_sp(BX), SP                // 恢复 SP 寄存器，确定堆栈帧。
    MOVL    gobuf_pc(BX), BX
    PUSHL   BX
    MOVL    0(DX), BX                       // 将 fnstart 地址放到 BX 寄存器。
    JMP BX                                  // 跳转并执行 fnstart 函数代码。
    POPL    BX

// asm_386.s
// void gogo(Gobuf*, uintptr); restore state from Gobuf; longjmp
TEXT runtime·gogo(SB), 7, $0
    MOVL    8(SP), AX                       // return 2nd arg
    MOVL    4(SP), BX                       // gobuf, G.sched
    MOVL    gobuf_g(BX), DX
    MOVL    0(DX), CX
    get_tls(CX)
    MOVL    DX, g(CX)
    MOVL    gobuf_sp(BX), SP                // 恢复 SP 寄存器
    MOVL    gobuf_pc(BX), BX                // 将 sched.pc，也就是 runtime.goexit 地址放到 BX。
    JMP BX                                  // 执行 goexit


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

// proc.c 1222: runtime·goexit continuation on g0.
static void goexit0(G *gp)
{
    // 调整状态
    gp->status = Gdead;
    gp->m = nil;
    gp->lockedm = nil;
    m->curg = nil;
    m->lockedg = nil;
```

```
    m->locked = 0;

    // 释放多余的栈内存
    runtime·unwindstack(gp, nil);

    // 将 G 放到 free-list 供复用。
    gfput(m->p, gp);

    // 回到 schedule，重新循环。
    schedule();
}
```

3.5 系统调用

当某个 M 因为系统调用发生阻塞 (thread block) 时，会调用 handoffp() 将 P 放入 idle-list。

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

    // 保存现场
    // Leave SP around for gc and traceback.
    g->sched.sp = (uintptr)runtime·getcallersp(&dummy);
    g->sched.pc = runtime·getcallerpc(&dummy);
    … ...

    // 修改状态
    g->status = Gsyscall;

    // 将 P 从当前 M 上分离。P.status = Pidle
    p = releasep();

    // 将 P 放到 idle list。
    handoffp(p);
}
```

从系统调用退出 exitsyscall0()，尝试获取一个 P，如果失败就将 G 放入全局运行队列，同时将 M 放入空闲队列。

```
static void exitsyscall0(G *gp)
{
    P *p;

    // 修改状态
    gp->status = Grunnable;
    gp->m = nil;
    m->curg = nil;

    // 获取 P
    p = pidleget();
```

```
    // 没有空闲 P，将 G 放入全局队列。
    if(p == nil) globrunqput(gp);

    // 能获取 P，继续执行该 G。
    if(p) execute(gp);                // 后续代码不会被执行。

    // 没有 P，停止 M，并放入 idle-list。
    stopm();
}
```

执行 Gosched 也会将 G 放入全局队列，然后去获取新的任务执行。
因为 G 执行状态被保存在自身的 sched 字段，所以现场和 M 是分离的。

B. Go 函数调用反汇编

用一个简单例子，看看官方 gc/6g 编译器所生成汇编质量。(2013/04, go 1.1 beta)

```
package main

import ()

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

func main() {
    x, y := 1, 2
    z := add(x, y)
    println(z)
}
```

直接用 go build 看默认优化效果。

```
(gdb) set disassembly-flavor intel                                  // 习惯 INTEL 格式

(gdb) b main.main
Breakpoint 1 at 0x2000: file main.go, line 10.

(gdb) r
Starting program: test
[New Thread 0x1c03 of process 2856]

Breakpoint 1, main.main () at main.go:10
10      func main() {
(gdb) disass
Dump of assembler code for function main.main:
=> 0x0000000000002000 <+0>:   mov    rcx,QWORD PTR gs:0x8a0        // G
   0x0000000000002009 <+9>:   cmp    rsp,QWORD PTR [rcx]           // 和 rsp 比较，
   0x000000000000200c <+12>:  ja     0x2013 <main.main+19>         // 以确定是否需要调整 G stack 空间。
   0x000000000000200e <+14>:  call   0x19620 <runtime.morestack00>
   0x0000000000002013 <+19>:  sub    rsp,0x8                       // 调整 rsp 位置，设定栈顶。
   0x0000000000002017 <+23>:  mov    rbx,0x1                       // 优先使用寄存器。
   0x000000000000201e <+30>:  mov    rax,0x2                       // 优化有限，gcc 会直接合并成 0x3。
   0x0000000000002025 <+37>:  add    rbx,rax                       // 加法结果保存在 rbx。
   0x0000000000002028 <+40>:  mov    QWORD PTR [rsp],rbx           // 准备 print 参数，从栈顶开始。
   0x000000000000202c <+44>:  call   0xf120 <runtime.printint>
   0x0000000000002031 <+49>:  call   0xf2c0 <runtime.printnl>
   0x0000000000002036 <+54>:  add    rsp,0x8
   0x000000000000203a <+58>:  ret
End of assembler dump.
```

函数 add 内联，省去了函数调用的额外开销。但代码指令优化不够，相比 GCC 还差很多。为了查看 gc/6g 如何安排参数和返回值，编译一个非优化版本。

```
$ go build -gcflags "-N -l"
```

```
(gdb) b main.main
Breakpoint 1 at 0x2030: main.go, line 10.

(gdb) b main.add
Breakpoint 2 at 0x2000: main.go, line 5.

(gdb) r
Starting program: test
[New Thread 0x1c03 of process 2904]

Breakpoint 1, main.main () at main.go:10
10      func main() {
(gdb) disass
Dump of assembler code for function main.main:
=> 0x0000000000002030 <+0>:   mov    rcx,QWORD PTR gs:0x8a0
   0x0000000000002039 <+9>:   cmp    rsp,QWORD PTR [rcx]
   0x000000000000203c <+12>:  ja     0x2043 <main.main+19>
   0x000000000000203e <+14>:  call   0x196c0 <runtime.morestack00>
   0x0000000000002043 <+19>:  sub    rsp,0x38                      // 栈顶
   0x0000000000002047 <+23>:  mov    QWORD PTR [rsp+0x18],0x1      // 局部变量 x
   0x0000000000002050 <+32>:  mov    QWORD PTR [rsp+0x28],0x2      // 局部变量 y
   0x0000000000002059 <+41>:  mov    rbx,QWORD PTR [rsp+0x18]      // 复制实参 x
   0x000000000000205e <+46>:  mov    QWORD PTR [rsp],rbx           // 实参 x 在 [rsp]
   0x0000000000002062 <+50>:  mov    rbx,QWORD PTR [rsp+0x28]      // 复制实参 y
   0x0000000000002067 <+55>:  mov    QWORD PTR [rsp+0x8],rbx       // 实参 y 在 [rsp+0x8]
   0x000000000000206c <+60>:  call   0x2000 <main.add>             // 调用 add 函数
   0x0000000000002071 <+65>:  mov    rbx,QWORD PTR [rsp+0x10]      // add 返回结果在 [rsp+0x10]
   0x0000000000002076 <+70>:  mov    QWORD PTR [rsp+0x30],rbx      // 将结果复制到其他地方
   0x000000000000207b <+75>:  mov    rbx,QWORD PTR [rsp+0x30]
   0x0000000000002080 <+80>:  mov    QWORD PTR [rsp+0x20],rbx      // 最终倒腾到 [rsp+0x20]
   0x0000000000002085 <+85>:  mov    rbx,QWORD PTR [rsp+0x20]
   0x000000000000208a <+90>:  mov    QWORD PTR [rsp],rbx           // println 实参也是放在 [rsp]
   0x000000000000208e <+94>:  call   0xf1c0 <runtime.printint>
   0x0000000000002093 <+99>:  call   0xf360 <runtime.printnl>
   0x0000000000002098 <+104>: add    rsp,0x38
   0x000000000000209c <+108>: ret
End of assembler dump.
```

在 Caller StackFrame 准备参数和函数返回值存储空间，从 [rsp] 开始依次是: arg1, arg2, ... ret。

继续看看 add 函数。

```
(gdb) disass
Dump of assembler code for function main.add:
=> 0x0000000000002000 <+0>:   sub    rsp,0x8                       // 当前栈帧长度 0x8
   0x0000000000002004 <+4>:   mov    rbx,QWORD PTR [rsp+0x10]      // rsp+0x10 是 main 栈顶，形参 x。
   0x0000000000002009 <+9>:   mov    rbp,QWORD PTR [rsp+0x18]      // 形参 y。
   0x000000000000200e <+14>:  add    rbx,rbp                       // 加法。
   0x0000000000002011 <+17>:  mov    QWORD PTR [rsp],rbx
   0x0000000000002015 <+21>:  mov    rbx,QWORD PTR [rsp]
```

```
   0x0000000000002019 <+25>:  mov    QWORD PTR [rsp+0x20],rbx      // 结果最终保存到 main ret 位置。
   0x000000000000201e <+30>:  add    rsp,0x8
   0x0000000000002022 <+34>:  ret
End of assembler dump.
```

没什么惊奇，从 main 栈帧空间取形参，结果也保存到原本就准备好的地方。

我们在 add 函数中并没有看到调整 G stack 空间的指令。可一旦 goroutine，事情就不一样了。改一下代码。

```
package main

import (
    "time"
)

func test(x, y int) {
    z := x + y
    println(z)
}

func main() {
    x, y := 1, 2
    go test(x, y)

    time.Sleep(time.Hour)
}
```

默认优化编译。

```
(gdb) disass
Dump of assembler code for function main.main:
=> 0x0000000000002040 <+0>:   mov    rcx,QWORD PTR gs:0x8a0
   0x0000000000002049 <+9>:   cmp    rsp,QWORD PTR [rcx]
   0x000000000000204c <+12>:  ja     0x2053 <main.main+19>
   0x000000000000204e <+14>:  call   0x1bd10 <runtime.morestack00>
   0x0000000000002053 <+19>:  sub    rsp,0x10
   0x0000000000002057 <+23>:  mov    rdx,0x1
   0x000000000000205e <+30>:  mov    rax,0x2
   0x0000000000002065 <+37>:  mov    QWORD PTR [rsp],rdx           // 实参 x
   0x0000000000002069 <+41>:  mov    rbx,rax
   0x000000000000206c <+44>:  mov    QWORD PTR [rsp+0x8],rax       // 实参 y
   0x0000000000002071 <+49>:  push   0x37d98                       // 这是 test 函数地址?
   0x0000000000002076 <+54>:  push   0x10
   0x0000000000002078 <+56>:  call   0x139b0 <runtime.newproc>
   0x000000000000207d <+61>:  pop    rcx
   0x000000000000207e <+62>:  pop    rcx
   0x000000000000207f <+63>:  movabs rbx,0x34630b8a000
   0x0000000000002089 <+73>:  mov    QWORD PTR [rsp],rbx
   0x000000000000208d <+77>:  call   0x1b3d0 <time.Sleep>
   0x0000000000002092 <+82>:  add    rsp,0x10
   0x0000000000002096 <+86>:  ret
End of assembler dump.
```

先看看 newproc 的函数签名。

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.
// Cannot split the stack because it assumes that the arguments
// are available sequentially after &fn; they would not be
// copied if a stack split occurred.  It's OK for this to call
// functions that split the stack.
void runtime·newproc(int32 siz, FuncVal* fn, ...)
```

runtime.h

```
struct FuncVal
{
	void   (*fn)(void);
	// variable-size, fn-specific data here
};
```

两个参数，按照参数排列顺序，0x37d98 应该是一个 FuncVal 对象，0x10 是 siz。

```
(gdb) x/1xg 0x37d98
0x37d98 <main.test·f>:0x0000000000002000

(gdb) x/4i 0x0000000000002000                                   // 果然是 test 地址。
   0x2000 <main.test>:mov    rcx,QWORD PTR gs:0x8a0
   0x2009 <main.test+9>:     cmp    rsp,QWORD PTR [rcx]
   0x200c <main.test+12>:    ja     0x2013 <main.test+19>
   0x200e <main.test+14>:    call   0x1bda0 <runtime.morestack16>
```

回到 main 函数。

```
Dump of assembler code for function main.main:
=> 0x0000000000002040 <+0>:   mov    rcx,QWORD PTR gs:0x8a0
   0x0000000000002049 <+9>:   cmp    rsp,QWORD PTR [rcx]
   0x000000000000204c <+12>:  ja     0x2053 <main.main+19>
   0x000000000000204e <+14>:  call   0x1bd10 <runtime.morestack00>
   0x0000000000002053 <+19>:  sub    rsp,0x10
   0x0000000000002057 <+23>:  mov    rdx,0x1
   0x000000000000205e <+30>:  mov    rax,0x2
   0x0000000000002065 <+37>:  mov    QWORD PTR [rsp],rdx           // 实参 x
   0x0000000000002069 <+41>:  mov    rbx,rax
   0x000000000000206c <+44>:  mov    QWORD PTR [rsp+0x8],rax       // 实参 y
   0x0000000000002071 <+49>:  push   0x37d98                       // FuncVal(test)
   0x0000000000002076 <+54>:  push   0x10                          // size
   0x0000000000002078 <+56>:  call   0x139b0 <runtime.newproc>     // 创建 G
   0x000000000000207d <+61>:  pop    rcx
   0x000000000000207e <+62>:  pop    rcx
   0x000000000000207f <+63>:  movabs rbx,0x34630b8a000
   0x0000000000002089 <+73>:  mov    QWORD PTR [rsp],rbx
   0x000000000000208d <+77>:  call   0x1b3d0 <time.Sleep>
```

```
   0x0000000000002092 <+82>:  add    rsp,0x10
   0x0000000000002096 <+86>:  ret
End of assembler dump.
```

和前面《源码阅读指南》中的分析一致，相应的参数顺序：size, test_address, arg1, arg2 ...

```
(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     0x2013 <main.test+19>
   0x000000000000200e <+14>:  call   0x1bda0 <runtime.morestack16>
   0x0000000000002013 <+19>:  sub    rsp,0x8
   0x0000000000002017 <+23>:  mov    rbx,QWORD PTR [rsp+0x10]      // size, test_address 已经被
   0x000000000000201c <+28>:  mov    rbp,QWORD PTR [rsp+0x18]      //   newproc pop 掉了，所以这个
   0x0000000000002021 <+33>:  add    rbx,rbp                       //   地址是没问题的。
   0x0000000000002024 <+36>:  mov    QWORD PTR [rsp],rbx
   0x0000000000002028 <+40>:  call   0x10e00 <runtime.printint>
   0x000000000000202d <+45>:  call   0x10fe0 <runtime.printnl>
   0x0000000000002032 <+50>:  add    rsp,0x8
   0x0000000000002036 <+54>:  ret
End of assembler dump.
```

在 test 函数出现 morestack 的身影，一切都释然了。morestack 代码可以直接看 asm_386.s、asm_amd64.s。

个人建议发行版本要去掉符号表和调试信息，具体编译参数请参考前面工具章节。