ch01

hello world

```
package main   => 包
import "fmt"       => 引入包
func main() {
fmt.Println （” Hello , 世界” ）
}
```

运行

$ go run helloworld.go

$ go build helloworld.go

命令行参数

os.Args

for 循环

for 是 Go 里面的唯一循环语句

```
for initialization; condition; post {

}

for condition {

}

for {

}
```

ch02

名称

名称的开头是一个字母（Unicode 中的字符即可）或下划线，后面可以跟任意数字和下划线，并区分大小写。

实体第个字母的大小写决定其可见性是否跨包如果名称以大写字母的开头，它是导出的，意味着它对包外是可见和可访问的，可以被自己包之外的其他程序所引用，像 fmt 包中的 Printf 包名本身总是由小写字母组成。

关键字: break default func inte face select case defer go map struct ch an else goto package switch con st fallthrough if range type continue for import return var

常量: true false iota nil

类型: int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr float32 float64 complex128 complex64 bool byte rune string error

函数: make len cap new append copy close delete complex real imag panic recove

声明

变量 var

```
var name type = expression

var i = 3

var i int

var i int = 3
```

- 变量可以通过调用返回多个值的函数进行初始化:

```
var f, err  =  os.Open(name)
```

- 短变量 name := expression //fixme
- Go 不允许存在无用的临时变量，不然会出现编译错误

```
package main
import (
    "fmt"
    "os"
)
func main() {
    s, sep : =" " ,
    for_, arg :  = range os.Args[1:) {
        s += sep +
            sep = ""
    }
    fmt.P intln(s)
}
```

```
package main

import (
    "fmt"
    "log"
    "os"
)

var cmd string = "string"
```

```
func main() {
    cmd, b := 1, 2 // 覆盖了包体外面的额string 类型的string，此处的cmd编程了int类型
    a, b := "sring", 1
    a, c := 1, 2 //会报错，因为a是string类型，而不能复制为int类型。
    _, err := os.Getwd()
    if err != nil {
        log.Fatal("os.Getwe failed: %v", err)
    }
    fmt.Println("some", b, cmd, a)
}
```

常量（const ）

```
const boilingF=212.0
```

类型（type ）

type name underlying-type

```
type Celsius float64
```

函数（func）

```
func (f Fahrenheit) String() string {
    return fmt.Sp intf  ( ”  %g ” ,  f)
}
```

指针

- 函数返回局部变量的地址是非常安全的

new 函数

```
p := new(int) //*int 类型的 , 指向未命名的 int 变量
fmt.Println p) // 输出 "  0"
*p = 2 //把未命名的
fmt.Println(*p) // 输出” 2"
```

变量生命周期

赋值

- 多重赋值

```
x, y = y, x
a[i], a[j] = a[j], a[i]
```

包、文件、导入

- 导入一个没有被引用的包。会触发编译错误

包初始化

```
func init() { /* ...*/ }
```

init 函数不能被调用和被引用，另一方面，它也是普通的函数在每个文件里，当程序启动的时候，init 函数按照它们声明的顺序自动执行。

```
package some

import "fmt"

var a = 1

func init() {
    a = a + 1
    fmt.Println(a)
}

func some() {
    fmt.Println("some")
}

func Print() {
    some()
    init() // some/some.go:18:2: undefined: init Error!!!
}
```

```
package main

import (
    "./some"
)

func main() {
    some.Print()
}
```

作用域

```
if f, err := cs.Open (fname); err != nil {// 编译错误 未使用
    return err
}

f. Stat() / / 编译错误: 未定义
```

```
f. Close() // 编译错误: 未定义
```

第三章基本数据类型

整型

- 如果原始的数值是有符号类型，而且最左边的 bit 位是 1 的话，那么最终结果可能是负的，例如 int8 的例子:

```
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

浮点数

- 提供了两种精度的浮点数，float32 和 float64
- 一个 float32 类型的浮点数可以提供大约 6 个十进制数的精度，而 float64 则可以提供约 15 个十进制数的精度；通常应该优先使用 float64 类型
- 如果一个函数返回的浮点数结果可能失败，最好的做法是用单独的标志报告失败，像这样

```
func compute() (value float64, ok bool) {
    // ...
    if failed {
        return 0, false
    }
    return result, true
}
```

复数

- 提供了两种精度的复数类型: complex64 和 complex128
- 内置的 complex 函数用于构建复数，内建的 real 和 imag 函数分别返回复数的实部和虚部
- Go 已经有很多科学计算的库了。作为一门泛用编程语言，提供一个方便的复数类型还是挺有必要的

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y)                 // "(-5+10i)"
fmt.Println(real(x*y))           // "-5"
fmt.Println(imag(x*y))           // "10"
```

布尔型

- 两种类型 true 和 false

字符串

- 内置的 len 函数可以返回一个字符串中的字节数目（不是 rune 字符数目），索引操作 s[i] 返回第 i 个字节的字节值，i 必须满足 $0 <= i < len(s)$ 条件约束

```
s := "hello, world"
fmt.Println(len(s))     // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')
// TypeOf(s[0]) --> uint8
```

- 如果试图访问超出字符串索引范围的字节将会导致 panic 异常

```
c := s[len(s)] // panic: index out of range
```

- 字符串的值是不可变的：一个字符串包含的字节序列永远不会被改变，当然我们也可以给一个字符串变量分配一个新字符串值。可以像下面这样将一个字符串追加到另一个字符串
- 这并不会导致原始的字符串值被改变，但是变量 s 将因为 += 语句持有一个新的字符串值，但是 t 依然是包含原先的字符串值
- 因为字符串是不可修改的，因此尝试修改字符串内部数据的操作也是被禁止的

```
s := "left foot"
t := s
s += ", right foot"

fmt.Println(s) // "left foot, right foot"
fmt.Println(t) // "left foot"

s[0] = 'L' // compile error: cannot assign to s[0]
```

- 标准库中有四个包对字符串处理尤为重要：bytes、strings、strconv 和 unicode 包。
- strings 包提供了许多如字符串的查询、替换、比较、截断、拆分和合并等功能
- bytes 包也提供了很多类似功能的函数，但是针对和字符串有着相同结构的 []byte 类型
- strconv 包提供了布尔型、整型数、浮点数和对应字符串的相互转换，还提供了双引号转义相关的转换
- unicode 包提供了 IsDigit、IsLetter、IsUpper 和 IsLower 等类似功能，它们用于给字符分类
- path 和 path/filepath 包提供了关于文件路径名更一般的函数操作
- 除了字符串、字符、字节之间的转换，字符串和数值之间的转换也比较常见。由 strconv 包提供这类转换功能

常量

- 常量表达式的值在编译期计算，而不是在运行期
- 每种常量的潜在类型都是基础类型：boolean、string 或数字
- 一个常量的声明语句定义了常量的名字，和变量的声明语法类似，常量的值不可修改

- 如果是批量声明的常量，除了第一个外其它的常量右边的初始化表达式都可以省略，如果省略初始化表达式则表示使用前面常量的初始化表达式写法，对应的常量类型也一样的

```
const pi = 3.14159 // approximately; math.Pi is a better approximation

const (
    a = 1
    b
    c = 2
    d
)

fmt.Println(a, b, c, d) // "1 1 2 2"
```

- 常量声明可以使用 iota 常量生成器初始化，它用于生成一组以相似规则初始化的常量，但是不用每行都写一遍初始化表达式
- 周日将对应 0，周一为 1，如此等等

```
type Weekday int

const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

- 无类型常量，通过延迟明确常量的具体类型，无类型的常量不仅可以提供更高的运算精度，而且可以直接用于更多的表达式而不需要显式的类型转换。例如，例子中的 ZiB 和 YiB 的值已经超出任何 Go 语言中整数类型能表达的范围，但是它们依然是合法的常量，而且像下面的常量表达式依然有效（译注: YiB/ZiB 是在编译期计算出来的，并且结果常量是 1024，是 Go 语言 int 变量能有效表示的）
- 另一个例子，math.Pi 无类型的浮点数常量，可以直接用于任意需要浮点数或复数的地方

```
fmt.Println(YiB/ZiB) // "1024"

var x float32 = math.Pi
var y float64 = math.Pi
var z complex128 = math.Pi
```

第四章复合数据类型

数组

- 数组的每个元素可以通过索引下标来访问，索引下标的范围是从 0 开始到数组长度减 1 的位置。内置的 len 函数将返回数组中元素的个数

```
var a [3]int             // array of 3 integers
fmt.Println(a[0])        // print the first element
fmt.Println(a[len(a)-1]) // print the last element, a[2]

// Print the indices and elements.
for i, v := range a {
    fmt.Printf("%d %d\n", i, v)
}

// Print the elements only.
for _, v := range a {
    fmt.Printf("%d\n", v)
}
```

- 当调用一个函数的时候，函数的每个调用参数将会被赋值给函数内部的参数变量，所以函数参数变量接收的是一个复制的副本，并不是原始调用的变量
- 当然，我们可以显式地传入一个数组指针，那样的话函数通过指针对数组的任何修改都可以直接反馈到调用者。下面的函数用于给 [32]byte 类型的数组清零

```
func zero(ptr *[32]byte) {
    for i := range ptr {
        ptr[i] = 0
    }
}
```

- 数组依然很少用作函数参数；相反，我们一般使用 slice 来替代数组，因为数组依然是僵化的类型，因为数组的类型包含了僵化的长度信息。上面的 zero 函数并不能接收指向 [16]byte 类型数组的指针，而且也没有任何添加或删除数组元素的方法

slice

- Slice（切片）代表变长的序列，序列中每个元素都有相同的类型。一个 slice 类型一般写作 []T，其中 T 代表 slice 中元素的类型；slice 的语法和数组很像，只是没有固定长度而已
- 数组和 slice 之间有着紧密的联系。一个 slice 是一个轻量级的数据结构，提供了访问数组子序列（或者全部）元素的功能，而且 slice 的底层确实引用一个数组对象

```
months := [...]string{1: "January", /* ... */, 12: "December"}

Q2 := months[4:7]
```

```
summer := months[6:9]
fmt.Println(Q2)     // ["April" "May" "June"]
fmt.Println(summer) // ["June" "July" "August"]
```

- 因为 slice 值包含指向第一个 slice 元素的指针，因此向函数传递 slice 将允许在函数内部修改底层数组的元素。换句话说，复制一个 slice 只是对底层的数组创建了一个新的 slice 别名
- 和数组不同的是，slice 之间不能比较，因此我们不能使用 == 操作符来判断两个 slice 是否含有全部相等元素

```
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

- 如果你需要测试一个 slice 是否是空的，使用 len(s) == 0 来判断，而不应该用 s == nil 来判断

map

- 是一个无序的 key/value 对的集合，其中所有的 key 都是不同的，然后通过给定的 key 可以在常数时间复杂度内检索、更新或删除对应的 value
- 在 Go 语言中, 一个 map 就是一个哈希表的引用, map 类型可以写为 map[K]V, 其中 K 和 V 分别对应 key 和 value。map 中所有的 key 都有相同的类型，所有的 value 也有着相同的类型，但是 key 和 value 之间可以是不同的数据类型
- Map 的迭代顺序是不确定的，并且不同的哈希函数实现可能导致不同的遍历顺序
- 和 slice 一样，map 之间也不能进行相等比较

```
ages := make(map[string]int) // mapping from strings to ints

ages := map[string]int{
    "alice":   31,
    "charlie": 34,
}

//和下面的定义等级
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

```
//删除操作
delete(ages, "alice") // remove element ages["alice"]

//遍历
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}

//遍历判断相等
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

结构体

- 结构体是一种聚合的数据类型，是由零个或多个任意类型的值聚合成的实体

```
type Employee struct {
    ID        int
    Name      string
    Address   string
    DoB       time.Time
    Position  string
    Salary    int
    ManagerID int
}

var dilbert Employee

//访问方式
dilbert.Salary -= 5000 // demoted, for writing too few lines of code

position := &dilbert.Position
*position = "Senior " + *position // promoted, for outsourcing to Elbonia
```

- 如果结构体成员名字是以大写字母开头的，那么该成员就是导出的；这是 Go 语言导出规则决定的。一个结构体可能同时包含导出和未导出的成员

- Go 语言有一个特性让我们只声明一个成员对应的数据类型而不指名成员的名字；这类成员就叫匿名成员

```
package main

import (
    "fmt"
)

type point struct {
    X int64
    Y int64
}

type some struct {
    point
    book int64
    X    int64 //注意不会报错
}

func main() {
    var s some
    s.X = 10
    s.Y = 10
    s.book = 10
    s.point.X = 11

    fmt.Println(s.X, s.point.X) // 10, 11
}
```

JSON

- JavaScript 对象表示法（JSON）是一种用于发送和接收结构化信息的标准协议
- JSON 是对 JavaScript 中各种类型的值——字符串、数字、布尔值和对象——Unicode 本文编码
- tag 只在两个地方起作用反射和类型匹配上，其它情况没有作用。
- 两个 struct 相等，需要具备相同的 identical tags。
- reflect 可以获得 tag 内容。

```
package main

import (
    "fmt"
    "reflect"
)
```

```
func main() {
    type S struct {
        F string `species:"gopher" color:"blue"`
    }
    s := S{}
    st := reflect.TypeOf(s)
    field := st.Field(0)
    fmt.Println(field.Tag.Get("color"), field.Tag.Get("species"))
}
```

文本和 HTML 模板

- 需要复杂的打印格式，这时候一般需要将格式化代码分离出来以便更安全地修改。这些功能是由 text/template 和 html/template 等模板包提供的，它们提供了一个将变量值填充到一个文本或 HTML 格式的模板的机制

函数

函数声明

```
func name(parameter-list) (result-list) {
  body
}
```

example

```
func hypot(x,y float64) float64 {
  return math.Sqrt(x*x + y*y)
}
```

同类型可以放在一起

```
func f(i, j, k int, s, t string) {/*...*/}
func f(i int, j int, k int, s string, t string){/*...*/}
```

需要注意点

1. go 语言没有默认参数的概念，也不能指定参数名

   ```
   def some(name=10, kkk=20):
       print(name)

   some(name=11)
   ```

2. 函数类型（具有相同形参列表和返回列表的函数是同种类型）
3. 没有函数体的函数声明，可能是来自于 go 语言之外的语言实现的。

   ```
   package math
   func sin(x float64) float64 // 使用汇编语言实现 （等到 12 章，才会明白）
   ```

递归

多值返回

忽略参数

```
links, _ := findlines(url)
```

传递参数

```
func findLinksLog(url string) ([]string, error) {
  log.Printf("findLinks %s", url)
  return findLinks(url)
}
```

有的函数也接受多返回值函数，作为多参数

```
log.Println(findLinks(url))
// 和下面的调用等价
```

```
links, err := findLinks(url)
log.Println(links, err)
```

裸返回

```
func CountWordsAndImages(url string) (words, images int, err error) {
  resp, err := http.Get(url)
  if err != nil {
    return
  }
  doc, err := html.Parse(resp.Body)
  resp.Body.Close()
  if err != nil {
    err = fmt.Errorf("parsing HTML: %s", err)
    return
  }
  //(这种匿名返回，会让人很难受，让然误解。所以尽量少用这种返回)
  words, images = countWordsAndImages(doc)
  return
}

func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }
```

错误

这里主要讲错误的处理策略

直接向上一层调用者汇报

```
resp, err := http.Get(url)
if err != nil {
  return nil, err
}
```

重试若干次再报错退出

```
func WaitForServer(url string) error {
  const timeout = 1 * time.Minute
  deadline := time.Now().Add(timeout)
  for tries := 0; time.Now().Before(deadline); tries++ {
    _, err := http.Head(url)
    if err == nil {
      return nil // success
    }
    log.Printf("server not responding (%s); retrying...", err)
    time.Sleep(time.Second << uint(tries)) // exponential back-off
  }
  return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

直接终止程序

```
// (In function main.)
if err := WaitForServer(url); err != nil {
  fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
  os.Exit(1)
}
```

应该由主程序来做。不应该由库函数来做。库函数应该报告错误就行了。

```
log.Fatalf // 可以实现日志输出
```

某些情况下，只是记录错误信息，然后继续运行

```
if err := WaitForServer(url); err != nil {
  log.Fatalf("Site is down: %v\n", err)
}
```

直接忽略掉错误

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
  return fmt.Errorf("failed to create temp dir: %v", err)
}
// ...use temp dir...
os.RemoveAll(dir) // 这个函数可能会错误，但是这里忽略了处理。
```

函数变量

```
var f func(int) int
```

注意，函数变量之间不可以比较。所以不能把函数变量作为 map 的 key 值。但是函数类型可以和 nil 比较。

作为参数的函数变量

```
func forEachNode(n *html.Node, pre, post func(n *html.Node) string){
  //body
}
```

匿名函数

```
strings.map(func(r rune) rune {return r+1}, "HAL-9000")
```

闭包的概念

```
func squares() func() int {
  var x int
  return func() int {
    x++
    return x * x
  }
}

func main() {
  f := squares()
  fmt.Println(f()) // "1"
  fmt.Println(f()) // "4"
  fmt.Println(f()) // "9"
  fmt.Println(f()) // "16"
  fb := squares()
  fmt.Println(fb()) // "1"
  fmt.Println(f())  // "25"
}
```

PS : (TonyHuiHUI) 给出可以理解为，squares 的调用后置，并没有清除 squares 的堆栈信息。所以 x 是可以被函数 f 访问到的。当第二次调用 squares 的时候再次创建了一个新的堆栈 fb 就可以访问到新的 x 了

容易出错的地方

wrong

```
var rmdirs []func()
for _, dir := range tempDirs() {
```

```
  os.MkdirAll(dir, 0755)
  rmdirs = append(rmdirs, func() {
    os.RemoveAll(dir) // NOTE: incorrect!
  })
}
```

right

```
var rmdirs []func()
for _, d := range tempDirs() {
  dir := d                // NOTE: necessary!
  os.MkdirAll(dir, 0755) // creates parent directories too
  rmdirs = append(rmdirs, func() {
    os.RemoveAll(dir)
  })
}

// ...do some work...
for _, rmdir := range rmdirs {
  rmdir() // clean up
}
```

变长函数

```
func sum(vals ...int) int {
  total := 0
  for _, val := range vals {
    total += val
  }
  return total
}
```

等价调用

```
fmt.Println(sum(1, 2, 3, 4)) // "10"
// 等价的调用
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

不同类型

```
func f(...int) {}
func g([]int) {}
```

延迟函数

```
func title(url string) error {
  resp, err := http.Get(url)
  if err != nil {
    return err
  }
  // Check Content-Type is HTML (e.g., "text/html; charset=utf-8").
  ct := resp.Header.Get("Content-Type")
  if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
    resp.Body.Close() // 调用了一次
    return fmt.Errorf("%s has type %s, not text/html", url, ct)
  }
  doc, err := html.Parse(resp.Body)
  resp.Body.Close() // 调用了一次
```

```
  if err != nil {
    return fmt.Errorf("parsing %s as HTML: %v", url, err)
  }

  visitNode := func(n *html.Node) {
    if n.Type == html.ElementNode && n.Data == "title" &&
      n.FirstChild != nil {
      fmt.Println(n.FirstChild.Data)
    }
  }
  forEachNode(doc, visitNode, nil)
  return nil
}
```

defer

```
func title(url string) error {
  resp, err := http.Get(url)
  if err != nil {
    return err
  }
  defer resp.Body.Close() // 发生在 return 之后
  ct := resp.Header.Get("Content-Type")
  if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
    return fmt.Errorf("%s has type %s, not text/html", url, ct)
  }
  doc, err := html.Parse(resp.Body)
  if err != nil {
    return fmt.Errorf("parsing %s as HTML: %v", url, err)
  }
  // ...print doc's title element...
  return nil
}
```

注意

1. defer 没有限制使用次数，执行的时候以调用 defer 的顺序倒序执行。
2. defer 语句的求值是在执行 defer 语句的时候执行。
3. defer 的执行在 return 语句之后。

改变返回值结果

```
func double(x int) (result int) {
  defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()// return 后执行打印操作
  return x + x
}
_ = double(4)
// Output:
// "double(4) = 8"
```

```
func triple(x int) (result int) {
  defer func() { result += x }()
  return double(x)
}
fmt.Println(triple(4)) // "12" 改变了返回值
```

文件描述符应用

可能会耗尽文件描述符资源

```
for _, filename := range filenames {
  f, err := os.Open(filename)
```

```
  if err != nil {
    return err
  }
  defer f.Close() // NOTE: risky; could run out of file descriptors
  // ...process f...
}
```

更好的方法

```
for _, filename := range filenames {
  if err := doFile(filename); err != nil {
    return err
  }
}

func doFile(filename string) error {
  f, err := os.Open(filename)
  if err != nil {
    return err
  }
  defer f.Close()
  // ...process f...
}
```

宕机 (panic)

注意

1. 宕机会导致程序退出，只有在十分严重的错误情况下才可以宕机。
2. 当发生宕机时，所有的延迟函数以倒序执行，直到回到 main 函数

```
func main() {
  f(3)
}

func f(x int) {
  fmt.Printf("f(%d)\n", x+0/x) // panics if x == 0
  defer fmt.Printf("defer %d\n", x)
  f(x - 1)
}
```

outputs

```
f(3)
f(2)
f(1)
defer 1
defer 2
defer 3
```

runtime

runtime 包提供了转储栈的方法使程序员可以诊断错误。

gopl.io/ch5/defer2

```
func main() {
  defer printStack()
  f(3)
}

func printStack() {
```

```
  var buf [4096]byte
  n := runtime.Stack(buf[:], false)
  os.Stdout.Write(buf[:n])
}
```

为什么可以打印出栈，因为 go 语言的宕机机制可以让延迟函数的执行在栈清理之前调用

恢复

recover 可以劫持宕机，然后处理之后恢复运行

```
func Parse(input string) (s *Syntax, err error) {
  defer func() {
    if p := recover(); p != nil {
      err = fmt.Errorf("internal error: %v", p) // 这里就会恢复，程序不会退出
    }
  }()
  // ...parser... 假如在这里发生宕机
}
```

```
func soleTitle(doc *html.Node) (title string, err error) {
  type bailout struct{}
  defer func() {
    switch p := recover(); p {
    case nil:
      // no panic
    case bailout{}:
      // "expected" panic
      err = fmt.Errorf("multiple title elements")
    default:
      panic(p) // unexpected panic; carry on panicking
    }
  }() // 定义了一个函数并调用之这是个匿名函数（延迟调用）

  // Bail out of recursion if we find more than one non-empty title.
  forEachNode(doc, func(n *html.Node) {
    if n.Type == html.ElementNode && n.Data == "title" &&
      n.FirstChild != nil {
      if title != "" {
        panic(bailout{}) // 宕机发生地
      }
      title = n.FirstChild.Data
    }
  }, nil)
  if title == "" {
    return "", fmt.Errorf("no title element")
  }
  return title, nil
}
```

方法

6.1 方法声明

方法的声明和普通函数的声明类似，只是在函数名字前面多了一个参数

```
package geometry

import "math"

type Point struct{ X, Y float64 }

// traditional function
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// same thing, but as a method of the Point type
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

- 附加的参数 p 称为方法的接收者
- Go 语言中，接收者不使用特殊名（比如 this 或者 self ），而是我们自己选择接收者名字
- 调用方法的时候，接收者在方法名的前面，这样就和声明保持一致

```
p := Point{1, 2}
q := Point{4, 6}
fmt.PrintIn(Distantce(p, q)) // "5", 函数调用
fmt.PrintIn(p.Distantce(q)) // "5", 函数调用
```

- 因为每一个类型有它自己的命名空间，所以我们能够在其他不同的类型中使用名字 Distance 作为方法名

```
// A Path is a journey connecting the points with straight lines.
type Path []Point

// Distance returns the distance traveled along the path.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

- Go 和许多其他面向对象的语言不同，它可以将方法绑定到任何类型上。可以很方便地为简单的类型（如数字、字符串、slice 、map ，甚至函数等）定义附加的行为。同一个包下的任何类型都可以声明方法，只要它的类型既不是指针类型也不是接口类型。

```
package main

import (
    "fmt"
)

type Greeting func(name string) string

func (p Greeting) Print(v string) {
    fmt.Println(v)
}

func display(name string) string {
    return name
```

```
}

func main() {
    var f Greeting

    f.Print("this is fine")
    //display.Print("This is fine") // 会报错
    f = display
    f.Print("this is fine")
}
```

- 使用方法的第一个好处：命名可以比函数更简短。在包的外部进行调用的时候，方法能够使用更加简短的名字且省略包的名字:

```
import "gopl.io/ch6/geometry"

perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) //"12", 独立函数
fmt.Println(perim.Distance()) //"12"geometry.Path 的方法
```

6.2 指针接收者的方法

由于主调函数会复制每一个实参变量，如果函数需要更新一个变量，或者如果一个实参太大而我们希望避免复制整个实参，因此我们必须使用指针来传递变量的地址。

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
// 注：在真实的程序中，习惯上遵循如果Point 的任何一个方法使用指针接收者，那么所有的 Point 方法都应该使用指针接
```

- 命名类型（Point）与指向它们的指针（*Point）是唯一可以出现在接收者声明处的类型。而且，为防止混淆，不允许本身是指针的类型进行方法声明:

```
type P *int
func (P) f() { /* ... */ } // compile error: invalid receiver type
```

- 通过提供 *Point 能够调用 (*Point).ScaleBy 方法，比如:

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"

// 或者:
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"

// 或者:
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"

// 或者:
p.ScaleBy(2)

// 或者:
pptr.Distance(q)

// 错误:
Point{1, 2}.ScaleBy(2) // compile error: can't take address of Point
```

- 总结：三种合法情况
    1. 实参和形参类型相同

2. 实参类型为 T 而形参为 *T
3. 实参类型为 *T 而形参为 T

指针类型接受者和非指针类型接受者的区别

```
package main

import "fmt"

type Mutatable struct {
    a int
    b int
}

func (m Mutatable) StayTheSame() { // m是传值传到StayTheSame中来的。所以这个函数中m是调用StayTheSame的变量的一
    m.a = 5
    m.b = 7
}

func (m *Mutatable) Mutate() { // m 是调用Mutate的m的一个指针，这是一个巨大的区别
    m.a = 10
    m.b = 14
}

func main() {
    m := &Mutatable{0, 0}
    fmt.Println(m)
    m.StayTheSame()
    fmt.Println(m)
    m.Mutate()
    fmt.Println(m)

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

    n := Mutatable{0, 0}
    fmt.Println(n)
    n.StayTheSame()
    fmt.Println(n)
    n.Mutate()
    fmt.Println(n)
}
```

If you don't need to edit the receiver value, use a value receiver.Value receivers are concurrency safe, while pointer receivers are not concurrency safe.

nil 是一个合法的接收者

示例: net/url **包里** Values **类型定义的一部分**

net/url

```
package url

// Values maps a string key to a list of values.
type Values map[string][]string
// Get returns the first value associated with the given key,
// or "" if there are none.
func (v Values) Get(key string) string {
     if vs := v[key]; len(vs) > 0 {
         return vs[0]
     }
     return ""
```

```
}
// Add adds the value to key.
// It appends to any existing values associated with key.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

gopl.io/ch6/urlvalues

```
m := url.Values{"lang": {"en"}} // direct construction
m.Add("item", "1")
m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q"))    // ""
fmt.Println(m.Get("item")) // "1"      (first value)
fmt.Println(m["item"])     // "[1 2]"  (direct map access)

m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3")         // panic: assignment to entry in nil map
```

6.3 通过结构体内嵌组成类型

gopl.io/ch6/coloredpoint

```
import "image/color"
type Point struct{ X, Y float64 }
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

- 内嵌可以使我们在定义 ColoredPoint 时得到一种句法上的简写形式

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y) // "2"
```

- 我们能够通过类型为 ColoredPoint 的接收者调用内嵌类型 Point 的方法，即使在 ColoredPoint 类型没有声明过这个方法:

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

- 注意: 这里 Point 类型并不能理解为是 ColoredPoint 类型的基类。Distance 有一个形参 Point, q 不是 Point ， 因此虽然 q 有一个内嵌的 Point 字段，但是必须显式地使用它

```
p.Distance(q) // compile error: cannot use q (ColoredPoint) as Point
```

- 匿名字段类型可以是个指向命名类型的指针

```
type ColoredPoint struct {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
```

```
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point                 // p and q now share the same Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point) // "{2 2} {2 2}"
```

- 当编译器处理选择子（比如 p.ScaleBy）的时候，首先，它先查找到直接声明的方法 ScaleBy，之后再从来自 ColoredPoint 的内嵌字段的方法中进行查找，再之后从 Point 和 RGBA 中内嵌字段的方法中进行查找，以此类推
- 方法只能在命名的类型（比如 Point ）和指向它们的指针（*Point ）中声明，但内嵌帮助我们能够在未命名的结构体类型中声明方法。

```
var (
    mu sync.Mutex // guards mapping 包级别变量
    mapping = make(map[string]string) // 包级别变量
)

func Lookup(key string) string {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return v
}


var cache = struct {
         sync.Mutex
         mapping map[string]string
}{mapping: make(map[string]string),} // 匿名结构体变量(ps:struct 没有名字，只使用了一次)

func Lookup(key string) string {
         cache.Lock()
         v := cache.mapping[key]
         cache.Unlock()
         return v
}
```

6.4 方法变量与表达式

方法变量：选择子 **p. Distance** 可以赋予一个方法变量，它是一个函数，把方法（ **Point.Distance** ）绑定到一个接收者 **p** 上

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance        // method value
fmt.Println(distanceFromP(q))      // "5"
var origin Point                   // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", sqrt(5)

scaleP := p.ScaleBy // method value
scaleP(2)           // p becomes (2, 4)
scaleP(3)           //      then (6, 12)
scaleP(10)          //      then (60, 120)
```

方法表达式：和调用一个普通的函数不同，在调用方法的时候必须提供接收者，并且按照选择子的语法进行调用

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance   // method expression
fmt.Println(distance(p, q))  // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
```

```
scale(&p, 2)
fmt.Println(p)            // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"
```

6.5 示例: 位向量

简单示例 | 0000...0000 | 0000...0000 | ... | 0000...0000 |

比如 32 号元素为真，则把上面从右向左的第 32 位二进制位置 1. ### 6.6 封装 #### Go 语言只有一种方式控制命名的可见性: 定义的时候，首字母大写的标识符是可以从包中导出的，而首字母没有大写的则不导出。- 结论 1: 要封装一个对象，必须使用结构体。示例种的 IntSet 类型被声明为结构体但是它只有单个字段:

```
type IntSet struct {
    words []uint64
}
```

而另一种定义则允许其他包内的使用方读取和改变这个 slice

```
type IntSet []uint64
```

- 结论 2: 在 Go 语言中封装的单元是包而不是类型.

```
package test

type Kkk struct {
    private string
    Public  string
}
```

```
package main

import (
    "./test"
    "fmt"
)

type some struct {
    words string
}

func main() {
    var fine = test.Kkk{}
    var good = some{}
    //fmt.Println(fine.private) // 不能访问其他包中结构体中的小写开头变量
    fmt.Println(fine.Public) // 可以访问其他包中结构体中的大写开头变量
    good.words = "long"
    fmt.Println(good.words) // 可以访问本包结构体中的小写开头变量
}
```

- 封装提供了三个优点
  1. 因为使用方不能直接修改对象的变量，所以不需要更多的语句用来检查变量的值。
  2. 隐藏实现细节可以防止使用方依赖的属性发生改变，使得设计者可以更加灵活地改变 API 的实现而不破坏兼容性。
  3. 防止使用者肆意地改变对象内的变量。

```
package log
type Logger struct {
    flags  int
    prefix string
    // ...
}
func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

- 封装并不总是必须的，比如

```
const day = 24 * time.Hour //day is a time.Duration
fmt.Println(day.Seconds()) // "86400"
```

interface

Go 语言的接口的独特之处在于是隐式实现的。对于一个具体的类型，无须申明实现了哪些接口，只需要提供接口所必须的的方法即可。

```
package io

type Reader interface{
    Read(p []byte)(n int, err error)
}

type Closer interface{
    Close() error
}
```

interface 嵌入

```
type ReadCloser interface{
    Reader
    Closer
}
```

指针类型和非指针类型所含有的接口不一定同相

```
 type IntSet struct { /* ... */ }
 func (*IntSet) String() string
 var _ = IntSet{}.String() // compile error: String requires *IntSet receiver

 var s IntSet
 var _ = s.String() // OK: s is a variable and &s has a String method

 var _ fmt.Stringer = &s // OK
 var _ fmt.Stringer = s  // compile error: IntSet lacks String method
```

interface 实现

如果一个具体的类型要实现一个接口，就必须实现接口类型中定义的所有方法。

接口的赋值规则

仅当一个表达式实现了一个接口时，这个表达式才可以赋值给该接口:

```
var M io.Write
w = os.Stdout           // OK : *os.File 有Write 方法
w = new(bytes.Buffer)   // OK: *bytes.Buffer有Write方法
w = time.Second         //  编译错误:  time.Duration缺少Write 方法
```

空接口类型 interface

空接口类型对其实现类型没有任何要求，所以我们可以把任何值赋给空接口类型。

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer )
```

使用 flag.Value 来解析参数

如下一个程序，它实现了睡眠指定时间的功能。通过 -period 命令行标志控制睡眠时长。

```
var period = flag.Duration("period",1*time.Second,"sleep period")
func main() {
    flag.Parse()
```

```
    fmt.Printf ("Sleeping for %v ...",*period)
    time.Sleep(*period)
    fmt.Println()
}

$ ./sleep -period 50ms
Sleeping for 50ms ...
```

对于时间长度的也可以支持自定义类型，只需要满足 flag.Value 接口的类型。

```
package flag
//Value 接口代表了存储在标志内的值
type Value interface {
    String() string
    Set(string) error
}
```

接口值

接口类型的值（接口值），分为两个部分：具体的类型和该类型的值。在 Go 语言中用类型描述符来表述接口值的类型部分。

```
var w io.Write
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

在上述的四行代码中，变量 w 有三不同的值。第一行是变量的初始化，这里将接口的类型和值都设置为 nil 第二行把 *os.File* 类型赋值给 *w*，*w* 的接口值的动态类型变成了 os.File 的类型描述符，动态值设置为了一个指向代表进程的标准输出的 os.File 类型的指针。第三行把 *bytes.Buffer* 类型赋值给 *w*，*w* 的接口值的动态类型变成了 bytes.Buffer 类型描述符，动态值变成了一个指向新分配缓冲区的指针。第四行又把 nil 赋值给了 w。

接口值可以用 = = 和 ! = 操作符来做比较，在比较两个接口值时，如果两个接口值的动态类型一致，但对应的动态值是不可比较的（比如 *slice* ），那么这个比较会以崩溃的方式失败

可以使用 fmt 包的%T 来拿到接口值的的动态类型，这在处理错误和调试时很有帮助:

```
w = os.Stdout
fmt.Printf("%T\n", w) //"*os.File"
```

注意：含有空指针的非空接口

空的接口值（其中不包含任何信息）与仅仅动态值为 **nil** 的接口值是不一样的.

```
const debug = true
func main() {
    var buf *bytes.Buffer  //var buf io.Writer 这样定义没有错
    if debug {
        buf = new(bytes.Buffer) //启用输出收集
    }

    f(buf) //注意:  微妙的错误 当debug为false时，buf是的动态值为nil
    if debug {
        //...使用buf...
    }
}
//如果out 不是nil ，  那么会向其写入输出的数据
func f(out io.Writer) {
//...其他代码...
    if out != nil {
    out.Write([]byte("done!\n") //向一个空接收者写值，崩溃。
    }
}
```

使用 **sort.Interface** 来排序

sort 包提供了针对任意序列根据任意排序函数原地排序的功能。

```
package sort
type Interface interface {
Len() int
Less(i, j int) bool // i, 是序列元素的
Swap(i, j int)
}
```

http. Handler 接口

```
package http
type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}
func ListenAndServe(address string, h Handler) error
```

ListenAndServe 函数需要一个服务器地址，比如 “localhost:8000”，以及一个 Handler 接口的实例（用来接受所有的请求）。

示例：给定一个服务器地址，输入 DB 中的信息

```
func main() {
    db := database {"shoes ":50 ," socks":5}
    log.Fatal(http.ListenAndServe("localhost:8000",db))
}
type dollars Float32
func (d dollars) String() string {return fmt.Sprintf (” $% . 2f” , d) }
type database map[string]dollars
func (db database) ServeHTTP(w http.ResponseWriter,req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
req..URL.Path
req.URL.Query.get("")
. . . . . .
```

error 接口

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

构造 error 最简单的方法就是调用 errors.New

完整的 error 包只有 4 行代码:

```
package errors
func New(text string) error{ return &errorString{text} }
type errorString struct { text string }
func (e *errorString) Error() string { return e.text }
```

通常使用封装函数 fmt.Errorf 来构造 error。

类型断言

形式：x.(T) //x 是一个接口类型的表达式，T 是一个类型

类型断言会检查作为操作数的动态类型是否满足指定的断言类型。

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)  //成功:  f = = os.Stdout
c : = w.(*bytes.Buffer) // 崩溃: 接口持有的是*os. File,不是*bytes.Buffer
```

如果断言出现在需要两个结果的赋值表达式中，那么断言不会在失败时崩溃:

```
var w io.Writer = os.Stdout // luixiao1223: os.Stdout是一个值。 io.Writer是类型。
f , ok : = w.(*os.File)    //成功: ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // 失败: !ok,  b == nil
```

Goroutine

goroutine

1. goroutine 和线程之间再数量上有很大的区别 fixme（9.8 节）
2. main 函数其实就是一个 goroutine，我们把它称作主 goroutine.

```
f()
go f()
```

```
#include <iostream>
// 必须的头文件
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

// 线程的运行函数
void* say_hello(void* args)
{
    cout << "Hello Runoob！" << endl;
    return 0;
}

int main()
{
    // 定义线程的 id 变量，多个变量使用数组
    pthread_t tids[NUM_THREADS];
    for(int i = 0; i < NUM_THREADS; ++i)
    {
        //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
        int ret = pthread_create(&tids[i], NULL, say_hello, NULL);
        if (ret != 0)
        {
            cout << "pthread_create error: error_code=" << ret << endl;
        }
    }
    //等各个线程退出后，进程才结束，否则进程强制结束了，线程可能还没反应过来；
    pthread_exit(NULL);
}
```

我们先来看看一个具体的例子

```
 func main() {
  go spinner(100 * time.Millisecond)
  const n = 45
  fibN := fib(n) // slow
  fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
  for {
    for _, r := range `-\|/` {
```

```
      fmt.Printf("\r%c", r)
      time.Sleep(delay)
    }
  }
}

func fib(x int) int {
  if x < 2 {
    return x
  }
  return fib(x-1) + fib(x-2)
}
```

注意

1. main 函数退出的时候，所有 goroutine 全部暴力退出
2. 没有程序化的方法让一个 goroutine 来终止另一个 goroutine

并发时钟服务器

clock1

```
func main() {
  listener, err := net.Listen("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  for {
    conn, err := listener.Accept()
    if err != nil {
      log.Print(err) // e.g., connection aborted
      continue
    }
    handleConn(conn) // handle one connection at a time
  }
}

func handleConn(c net.Conn) {
  defer c.Close()
  for {
    _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
    if err != nil {
      return // e.g., client disconnected
    }
    time.Sleep(1 * time.Second)
  }
}
//!-
```

当服务器打开后，可以使用

```
nc localhost 8000
```

连接，但是只能允许 1 个客户端连接。

netcat1

```
func main() {
  conn, err := net.Dial("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
```

```
  }
  defer conn.Close()
  mustCopy(os.Stdout, conn)
}
func mustCopy(dst io.Writer, src io.Reader) {
  if _, err := io.Copy(dst, src); err != nil {
    log.Fatal(err)
  }
}
```

clock2

可以接受多个客户端连接

```
func handleConn(c net.Conn) {
  defer c.Close()
  for {
    _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
    if err != nil {
      return // e.g., client disconnected
    }
    time.Sleep(1 * time.Second)
  }
}

func main() {
  listener, err := net.Listen("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  //!+
  for {
    conn, err := listener.Accept()
    if err != nil {
      log.Print(err) // e.g., connection aborted
      continue
    }
    go handleConn(conn) // handle connections concurrently
  }
}
```

并发回声服务器

reverb1

```
func echo(c net.Conn, shout string, delay time.Duration) {
  fmt.Fprintln(c, "\t", strings.ToUpper(shout))
  time.Sleep(delay)
  fmt.Fprintln(c, "\t", shout)
  time.Sleep(delay)
  fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
  input := bufio.NewScanner(c)
  for input.Scan() {
    echo(c, input.Text(), 1*time.Second)
  }
  // NOTE: ignoring potential errors from input.Err()
  c.Close()
```

```
}

func main() {
  l, err := net.Listen("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Print(err) // e.g., connection aborted
      continue
    }
    go handleConn(conn)
  }
}
```

注意所有的返回是按顺序的,

```
some
    SOME
GOOD
    some
    some
    GOOD
    good
    good
```

reverb2

```
func echo(c net.Conn, shout string, delay time.Duration) {
  fmt.Fprintln(c, "\t", strings.ToUpper(shout))
  time.Sleep(delay)
  fmt.Fprintln(c, "\t", shout)
  time.Sleep(delay)
  fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
  input := bufio.NewScanner(c)
  for input.Scan() {
    go echo(c, input.Text(), 1*time.Second)
  }
  // NOTE: ignoring potential errors from input.Err()
  c.Close()
}

func main() {
  l, err := net.Listen("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Print(err) // e.g., connection aborted
      continue
    }
    go handleConn(conn)
  }
}
```

返回是乱序的，因为并发返回

```
some
   SOME
good
   GOOD
   some
   good
   some
   good
```

通道

1. 用来在 goroutine 之间进行通信用的。
2. 通道是引用类型。
3. 通道类型之间可以比较 ==, 通道的 0 值为 nil

```
ch := make(chan int)
ch <- x
x = <- ch
<-ch
close(ch)
```

1. 往一个关闭的通道发送数据，将引发宕机。
2. 从一个关闭的通道获取数据，会不断的获取数据。直到获取完毕所有数据。然后，之后的操作立即返回 nil。
3. 重复关闭一个通道也会引发宕机。

```
ch = make(chan int)     // 无缓冲通道
ch = make(chan int, 0)  // 无缓冲通道
ch = make(chan int, 1)  // 容量为 1 的通道
ch = make(chan init, 3) // 容量为 3 的通道
```

无缓冲通道和容量为 1 的通道是不一样的两种通道。

```
ch = make(chan int, 1)  // 容量为 1 的通道
ch <- 10
fmt.Println("something") // 在 ch<-10 之后立即执行

// goroutine 1
ch = make(chan int, 0)  // 容量为 0 的通道
ch <- 10 // 阻塞住
fmt.Println("something") // 不会立即执行

// goroutine 2
<-ch // 当这句话被执行的时候 goroutine 1 中的阻塞才会被消除。
```

第一个向无缓冲通道里面写入或提取输出的 goroutine 都会阻塞，直到第二个 goroutine 去提出或者写入数据。不同于缓冲为 1 的通道。这也是为什么无缓冲通道可以用来同步两个 goroutine 的原因。

无缓冲通道

1. 无缓冲通道也叫做同步通道。

```
func main() {
  conn, err := net.Dial("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  done := make(chan struct{})
  go func() {
    io.Copy(os.Stdout, conn) // NOTE: ignoring errors
    log.Println("done")
    done <- struct{}{} // signal the main goroutine
  }()
```

```
  mustCopy(conn, os.Stdin)//注意主程序只执行到这一步，就卡住了。因为要读取键盘输入。
  conn.Close()
  <-done // wait for background goroutine to finish
}

func mustCopy(dst io.Writer, src io.Reader) {
  if _, err := io.Copy(dst, src); err != nil {
    log.Fatal(err)
  }
}
```

等待计算结果

```
package main
import (
  "fmt"
  "time"
)
func main() {
  result := 0
  done := make(chan struct{})
  go func() {
    for i := 1; i < 10; i++ {
      result = i + result
      time.Sleep(1000 * time.Millisecond)
      fmt.Println(".....")
    }
    done <- struct{}{}
  }()
  <-done
  fmt.Println(result)
}
```

管道

pipeline1

```
func main() {
  naturals := make(chan int)
  squares := make(chan int)

  // Counter
  go func() {
    for x := 0; ; x++ {
      naturals <- x
    }
  }()

  // Squarer
  go func() {
    for {
      x := <-naturals
      squares <- x * x
    }
  }()

  // Printer (in main goroutine)
  for {
    fmt.Println(<-squares)
  }
}
```

pipeline2

```
func main() {
  naturals := make(chan int)
  squares := make(chan int)

  // Counter
  go func() {
    for x := 0; x < 100; x++ {
      naturals <- x
    }
    close(naturals)
  }()

  // Squarer
  go func() {
    for x := range naturals {
      squares <- x * x
    }
    close(squares)
  }()

  // Printer (in main goroutine)
  for x := range squares {
    fmt.Println(x)
  }
}
```

1. 关闭每一个通道不是必须的。
2. 通道的关闭是通过是否可以访问来确定是否资源回收。而不是是否关闭

单向通道类型

```
func counter(out chan<- int) {
  for x := 0; x < 100; x++ {
    out <- x
  }
  close(out)
}

func squarer(out chan<- int, in <-chan int) {
  for v := range in {
    out <- v * v
  }
  close(out)
}

func printer(in <-chan int) {
  for v := range in {
    fmt.Println(v)
  }
}

func main() {
  naturals := make(chan int)
  squares := make(chan int)

  go counter(naturals)
  go squarer(squares, naturals)
  printer(squares)
}
```

缓冲通道

```
ch := make(chan string, 3)
fmt.Println(cap(ch)) // 3
fmt.Println(len(ch)) // 2 当前有多少个元素在里面
```

1. 注意不能把管道当做队列来用。因为使用队列来用稍有不甚，会导致程序卡住。

```
func mirroredQuery() string {
  responses := make(chan string, 3)
  go func() {responses <- request("asia.gopl.io")} ()
  go func() {responses <- request("europe.gopl.io")} ()
  go func() {responses <- request("americas.gopl.io")} ()
  return <-responses
}

func request(hostname string) (response string) {/* ... */}
```

1. 如果采用无缓冲的 chan，那么慢的两个 goroutine 就会卡住。不会被释放掉，因为这两个 goroutine 阻塞住了，无法运行。
2. 具体是怎么泄露的？fix

并行循环

无并行程序写法

```
func makeThumbnails(filenames []string) {
  for _, f := range filenames {
    if _, err := thumbnail.ImageFile(f); err != nil {
      log.Println(err)
    }
  }
}
```

并行写法

```
func makeThumbnails(filenames []string) {
  for _, f := range filenames {
    go thumbnail.ImageFile(f) // 这个地方存在问题，程序又错误!!!
  }
}
```

上面的程序有两个地方存在问题。

1. f 的问题
2. 程序运行之后瞬间推出。并没有任何函数再等待 goroutine 运行完毕。

```
func makeThumbnails(filenames []string){
  ch := make(chan struct{})
  for _, f : range filenames {
    go func(f string) {
      thumbnail.ImageFile(f)
      ch <-struct{}{}
    }(f)
  }

  for range filenames {
    <-ch
  }
}
```

处理返回值

```
func makeThumbnails(filenames []string) error{
  errors := make(chan error)
  for _, f := range filenames {
```

```
    _, err := thumbnail.ImageFile(f)
    errors <- err
  }(f)

  for range filenames {
    if err := <-errors; err != nil {
      return err // 这里不正确，goroutine 泄露危险
    }
  }

  return nil
}
```

Q: 为什么会造成 goroutine 协奏，goroutine 泄露的原理是什么?

解决方案，是开辟一个和 filenames 一样长的通道。

```
func makeThumbnails5(filenames []string) (thumbfiles []string, err error) {
  type item struct {
    thumbfile string
    err       error
  }
  ch := make(chan item, len(filenames))
  for _, f := range filenames {
    go func(f string) {
      var it item
      it.thumbfile, it.err = thumbnail.ImageFile(f)
      ch <- it
      }(f)
  }
  for range filenames {
    it := <-ch
    if it.err != nil {
      return nil, it.err
    }
    thumbfiles = append(thumbfiles, it.thumbfile)
  }
  return thumbfiles, nil
}
```

一个更好的解决方案

```
func makeThumbnails6(filenames <-chan string) int64 {
  sizes := make(chan int64)
  var wg sync.WaitGroup // number of working goroutines
  for f := range filenames {
    wg.Add(1)
    // worker
    go func(f string) {
      defer wg.Done()
      thumb, err := thumbnail.ImageFile(f)
      if err != nil {
        log.Println(err)
        return }
      info, _ := os.Stat(thumb) // OK to ignore error
      sizes <- info.Size()
    }(f)
  }
  // closer
  go func() {
    wg.Wait()
    close(sizes)
```

```
  }()
  var total int64
  for size := range sizes {
    total += size
  }
  return total
}
```

并发的 Web 爬虫

第一版

```
func crawl(url string) []string {
  fmt.Println(url)
  list, err := links.Extract(url)
  if err != nil {
    log.Print(err)
  }
  return list
}

func main() {
  worklist := make(chan []string)
  // Start with the command-line arguments.
  go func() { worklist <- os.Args[1:] }()
  // Crawl the web concurrently.
  seen := make(map[string]bool)
  for list := range worklist { // 注意，不会结束的原因在这里。因为阻塞住了。当 worklist 中的内容为空时。阻塞住。
    for _, link := range list {
      if !seen[link] {
        seen[link] = true
        go func(link string){
          worklist <- crawl(link)
        }(link)
      }
    }
  }
}
```

问题:

1. 打开太多的链接。并发度太高，导致文件描述符不够用
2. 程序并没有终止。

解决链接太多问题

```
var tokens = make(chan struct{}, 20)
func crawl(url string) []string {
  fmt.Println(url)
  tokens <- struct{}{} // acquire a token
  list, err := links.Extract(url) // 最多只有 20 个 go 线程在获取 url 对应的所有链接。
  <-tokens // release the token
  if err != nil {
    log.Print(err)
  }
  return list
}
```

解决终止问题

```
func main() {
  worklist := make(chan []string)
```

```
  var n int // number of pending sends to worklist
  // Start with the command-line arguments.
  n++
  go func() { worklist <- os.Args[1:] }()
  // Crawl the web concurrently.
  seen := make(map[string]bool)
  for ; n > 0; n-- {
    list := <-worklist
    for _, link := range list {
      if !seen[link] {
        seen[link] = true
        n++
        go func(link string) {
          worklist <- crawl(link)
        }(link)
      }
    }
  }
}
```

一个替代方案

解决连接数过多，没有解决终止问题？

```
func main() {
  worklist := make(chan []string)  // lists of URLs, may have duplicates
  unseenLinks := make(chan string) // de-duplicated URLs
  // Add command-line arguments to worklist.
  go func() { worklist <- os.Args[1:] }()
  // Create 20 crawler goroutines to fetch each unseen link.
  for i := 0; i < 20; i++ {
    go func() {
      for link := range unseenLinks {
        foundLinks := crawl(link)
        go func() { worklist <- foundLinks }()
      }
    }()
  }
  // The main goroutine de-duplicates worklist items
  // and sends the unseen ones to the crawlers.
  seen := make(map[string]bool)
  for list := range worklist {
    for _, link := range list {
      if !seen[link] {
        seen[link] = true
        unseenLinks <- link
      }
    }
  }
}
```

多路 select

倒计时程序

倒计时发射程序

```
func main() {
  fmt.Println("Commencing countdown.")
  tick := time.Tick(1 * time.Second)
  for countdown := 10; countdown > 0; countdown-- {
```

```
    fmt.Println(countdown)
    <-tick
  }
  launch()
}
```

取消发射程序

```
abort := make(chan struct{})
go func() {
  os.Stdin.Read(make([]byte, 1)) // read a single byte
  abort <- struct{}{}
}()
```

如何同时接收并处理 tick 和 abort 消息？select

```
select {
     case <-ch1:
         // ...
     case x := <-ch2:
         // ...use x...
     case ch3 <- y:
         // ...
     default:
  // ...
}
```

最终实现

```
func main() {
  // ...create abort channel...
  fmt.Println("Commencing countdown.  Press return to abort.")
  select {
  case <-time.After(10 * time.Second):
    // Do nothing.
  case <-abort:
    fmt.Println("Launch aborted!")
    return
    }
  launch()
}
```

注意事项

多个 **chan** 同时有数据的时候怎么办？

随机选择一个执行

goroutine 泄露

```
func main() {
  // ...create abort channel...
  fmt.Println("Commencing countdown.  Press return to abort.")
  tick := time.Tick(1 * time.Second)// 行为类似创建一个 goroutine 不断的向 tick 发送事件
  for countdown := 10; countdown > 0; countdown-- {
    fmt.Println(countdown)
    select {
    case <-tick:
      // Do nothing.
    case <-abort:
      fmt.Println("Launch aborted!")
      return // 返回之后，tick 没有对象去取事件，但是却有一个 goroutine 再不断的发事件给 tick. 所以 goroutine 泄露
    }
  }
```

```
  launch()
}
```

记得关闭它

```
ticker := time.NewTicker(1 * time.Second)
<-ticker.C // receive from the ticker's channel
ticker.Stop() // cause the ticker's goroutine to terminate (FIXME)
```

非阻塞信道

```
select {
case <-abort:
  fmt.Printf("Launch aborted!\n")
  return
default:
  // do nothing
}
```

nil chan

往 nil chan 发送或者接受数据都是永久阻塞。

```
nil <- "some" // 发送
<- nil        // 接受
```

并发目录遍历

第一版

```
// walkDir recursively walks the file tree rooted at dir
// and sends the size of each found file on fileSizes.
func walkDir(dir string, fileSizes chan<- int64) {
  for _, entry := range dirents(dir) {
    if entry.IsDir() {
      subdir := filepath.Join(dir, entry.Name())
      walkDir(subdir, fileSizes)
    }else {
      fileSizes <- entry.Size()
    }
  }
}
// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
  entries, err := ioutil.ReadDir(dir)
  if err != nil {
    fmt.Fprintf(os.Stderr, "du1: %v\n", err)
    return nil
  }
  return entries
}
```

```
func main() {
  // Determine the initial directories.
  flag.Parse()
  roots := flag.Args()
  if len(roots) == 0 {
    roots = []string{"."}
  }
  // Traverse the file tree.
  fileSizes := make(chan int64)
  go func() {
```

```
    for _, root := range roots {
      walkDir(root, fileSizes)
    }
    close(fileSizes)
  }()
  // Print the results.
  var nfiles, nbytes int64
  for size := range fileSizes {
    nfiles++
    nbytes += size
  }
  printDiskUsage(nfiles, nbytes)
}

func printDiskUsage(nfiles, nbytes int64) {
  fmt.Printf("%d files  %.1f GB\n", nfiles, float64(nbytes)/1e9)
}
```

改进版

每隔一段时间输出信息

参数 v 控制是否输出 filesize，每隔一段时间输出

```
var verbose = flag.Bool("v", false, "show verbose progress messages")
func main() {
  // ...start background goroutine...
  // Print the results periodically.
  var tick <-chan time.Time
  if *verbose {
    tick = time.Tick(500 * time.Millisecond)
  }
  var nfiles, nbytes int64
loop:
  for {
    select {
    case size, ok := <-fileSizes:
      if !ok {
        break loop // fileSizes was closed
      }
      nfiles++
      nbytes += size
    case <-tick: // 有么有 goroutine 泄露?
      printDiskUsage(nfiles, nbytes)
    }
  }
  printDiskUsage(nfiles, nbytes) // final totals
}
```

充分利用好并发

```
func main() {
  // ...determine roots...
  // Traverse each root of the file tree in parallel.
  fileSizes := make(chan int64)
  var n sync.WaitGroup
  for _, root := range roots {
    n.Add(1)
    go walkDir(root, &n, fileSizes)
  }
  go func() {
```

```
    n.Wait()
    close(fileSizes)
  }()
  // ...select loop... 上一个例子的信息输出循环
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
  defer n.Done()
  for _, entry := range dirents(dir) {
    if entry.IsDir() {
      n.Add(1)
      subdir := filepath.Join(dir, entry.Name())
      go walkDir(subdir, n, fileSizes)
    } else {
      fileSizes <- entry.Size()
    }
  }
}
```

限制上一个例子的并发量

```
// sema is a counting semaphore for limiting concurrency in dirents.
var sema = make(chan struct{}, 20)
// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
  sema <- struct{}{}        // acquire token
  defer func() { <-sema }() // release token
  // ...
}
```

取消

```
var done = make(chan struct{})
func cancelled() bool {
  select {
  case <-done:
    return true
  default:
    return false
  }
}

// Cancel traversal when input is detected.
go func() {
  os.Stdin.Read(make([]byte, 1)) // read a single byte
  close(done)
}()

for {
  select {
  case <-done: // 为什么多个地方有这个语句？因为在一个关闭的通道上取东西，直接返回
    // Drain fileSizes to allow existing goroutines to finish.
    for range fileSizes {
      // Do nothing.
    }
    return
  case size, ok := <-fileSizes:
    // ...
  }
}
```

goroutine 各自分别退出自己。

```
func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
  defer n.Done()
  if cancelled() {
    return
  }
  for _, entry := range dirents(dir) {
    // ...
  }
}
func dirents(dir string) []os.FileInfo {
  select {
  case sema <- struct{}{}: // acquire token
  case <-done:
    return nil // cancelled
  }
  defer func() { <-sema }() // release token
  // ...read directory...
}
```

重要的测试技巧

Q：其实，你自己是很难判断是否把所有的 goroutine 都关闭并推出了。那么如何在 main 函数返回的时候，确定还有没有 goroutine 再运行呢？

A：在 main 函数退出之前，调用一个 panic，看运行时堆栈信息。如果信息只有 main 函数的堆栈则说明所有 goroutine 都退出了。否则，就是说有的 goroutine 没有退出。

聊天服务器

```
func main() {
  listener, err := net.Listen("tcp", "localhost:8000")
  if err != nil {
    log.Fatal(err)
  }
  go broadcaster() // 广播消息
  for {
    conn, err := listener.Accept()
    if err != nil {
      log.Print(err)
      continue
    }
    go handleConn(conn) // 管理客户端发来的连接
  }
}

type client chan<- string // an outgoing message channel
var (
  entering = make(chan client) // 这个 entering chan 将输入输出什么？应该是 chan<- string
  leaving  = make(chan client)
  messages = make(chan string) // all incoming client messages
)
func broadcaster() {
  clients := make(map[client]bool) // all connected clients (chan 是可 hash 的么？)
  for {
    select {
    case msg := <-messages:
      // Broadcast incoming message to all
      // clients' outgoing message channels.
      for cli := range clients {
```

```
        cli <- msg
      }
    case cli := <-entering:
      clients[cli] = true
    case cli := <-leaving:
      delete(clients, cli)
      close(cli)
    }
  }
}

func handleConn(conn net.Conn) {
  ch := make(chan string) // outgoing client messages
  go clientWriter(conn, ch)
  who := conn.RemoteAddr().String()
  ch <- "You are " + who
  messages <- who + " has arrived"
  // 把 ch 本身放到 entering 里面, 不是从 ch 中取出 string 放入 entering. 因为 entering 的接收类型为 chan<- string
  entering <- ch
  input := bufio.NewScanner(conn)
  for input.Scan() {
    messages <- who + ": " + input.Text()
  }
  // NOTE: ignoring potential errors from input.Err()
  leaving <- ch
  messages <- who + " has left"
  conn.Close()
}
func clientWriter(conn net.Conn, ch <-chan string) {
  for msg := range ch { // 会泄露么?
    fmt.Fprintln(conn, msg) // NOTE: ignoring network errors
  }
}
```

9. 使用共享变量实现并发

9.1 竞态

并发: 一个程序只有一个 **goroutine** 时，程序串行进行；有多个 **goroutine** 时，里面的事件并发进行

并发安全: 函数在并发调用时能正常工作

竞态: 多个 **goroutine** 按某些交错顺序执行时，程序无法给出正确结果

举例: 银行账户

```
package bank
var balance int
func Deposit(amount int) { balance = balance + amount }
func Balance() int  {return balance }
```

```
// Alice:
go func () {
    bank.Deposit(200) // Al
    fmt.Println("=", bank. Balance()) // A2
}()
// Bob :
go bank.Deposit(100) // B
```

- 程序中的这种状况是竞态中的一种，称为数据竞态 (data race)。数据竞态发生于两个 goroutine 并发读写同一个变量并且至少其中一个是写入时
- Go 语言很少有 “未定义行为” 的问题

```
var x []int
go func() { x = make([]int, 10) }()
go func() { x = make([]int, 1000000) }()
x[999999) = 1 // 注意: 未定义行为，可能造成内存异常
```

避免数据竞态的三种方法

- 不要修改变量

```
var icons = make(map[string]image.Image)
func loadlcon(name string) image.Image
// 注意: 并发不安全
func Icon(name string) image.Image {
    icon, ok := icons[name]
    if !ok {
        icon = loadIcon(name)
        icons[name] = icon
    }
    return icon
}

var icons = map[string]image.Image{
    "spades.png": load Icon("spades.png" ),
    "hearts.png": load Icon  ( "hearts.png" ) ,
    "diamonds.png": load Icon("diamonds.png"),
    "clubs.png" : loadlcon("clubs.png" ),
}
// 并发安全
func Icon(name string) image.Image { return icons[name] }
```

- 避免从多个 goroutine 访问同一个变量

```
// Package bank provides a concurrency-safe bank with one account.
package bank
```

```
var deposits = make(chan int) // send amount to deposit
var balances = make(chan int) // receive balance

func Deposit(amount int) { deposits <- amount }
func Balance() int       { return <-balances }

func teller() {
    var balance int // balance 被限制在 teller goroutine 中
    for {
        select {
        case amount := <-deposits:
            balance += amount
        case balances <- balance:
        }
    }
}

func init() {
    go teller() // start the monitor goroutine
}
```

- 允许多个 goroutine 访问同一个变量，但在同一时间只有一个 goroutine 可以访问。这种方法称为互斥机制。

9.2 互斥锁: sync.Mutex

一个计数上限为 1 的信号量称为二进制信号量（**binary semaphore**）

sync 包有一个单独的 **Mutex** 类型来支持互斥锁模式。它的 **Lock** 方法用于获取令牌（**token**，此过程也称为上锁），**Unlock** 方法用于释放令牌:

```
// Package bank provides a concurrency-safe single-account bank.
package bank

//!+
import "sync"

var (
    mu      sync.Mutex // guards balance
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}
```

- 在 Lock 和 Unlock 之间的代码，可以自由地读取和修改共享变量，这一部分称为临界区域
- 这种函数、互斥锁、变量的组合方式称为监控（monitor）模式
- **Go** 语言的 **defer** 语句: 通过延迟执行 Unlock 就可以把临界区域隐式扩展到当前函数的结尾，避免了必须在一个或者多个远离 Lock 的位置插入一条 Unlock 语句

```
func Balance() int {
    mu.Lock()
    defer mu.Unlock()
```

```
    return balance
}
```

- 另一个问题：下面的代码 Deposit 会通过调用 mu.Lock() 来尝试再次获取互斥锁，但由于互斥锁是不能再入的（无法对一个已经上锁的五斥量再上锁），因此这会导致死锁，Withdraw 会一直被卡住

```
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    Deposit(-amount)
    if Balance() < 0 {
        Deposit (amount)
        return false //余额不足
    }
    return true
}
```

解决方法:

```
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    deposit(-amount)
    if Balance() < 0 {
        deposit (amount)
        return false //余额不足
    }
    return true
}

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    deposit (amount)
}
```

9.3 读写互斥锁：sync.RWMutex

```
var mu sync.RWMutex
var balance int

func Balance() int {
    mu.RLock()
    defer mu.RUnlock()
    return balance
}
```

- Rlock 仅可用于在临界区域内对共享变量元写操作的情形。一般来讲，我们不应当假定那些逻辑上只读的函数和方法不会更新一些变量
- 因为复杂的内部簿记工作，仅在绝大部分 goroutine 都在获取读锁并且锁竞争比较激烈时（即，goroutine 一般都需要等待后才能获到锁），RWMutex 才有优势

9.4 内存同步

```
var x, y int
go func() {
    x = 1 // A1
    fmt.Print ("y:", y, " ") // A2
}()
go func() {
    y = 1 // A1
    fmt.Print ("x:", x, " ") // A2
}()
```

可能的结果: y:0 x:1 x:0 y:1 x:1 y:1 y:1 x:1 x:0 y:0 y:0 x:0

- goroutine 是串行一致的（sequentially consistent），但在缺乏使用通道或者互斥量来显式同步的情况下，并不能保证所有的 goroutine 看到的事件顺序都是一致的

9.5 延迟初始化: sync.Once

```
var icons map[string]image.Image

func loadIcons() {
    icons = map[string]image.Image{
        "spades.png": loadIcon("spades.png" ),
        "hearts.png": load Icon  ( "hearts.png" ) ,
        "diamonds.png": load Icon("diamonds.png"),
        "clubs.png" : loadIcon("clubs.png" ),
    }
}

// 注意: 并发不安全
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons()
    }
    return icon[name]
}
```

语句可能被如下重排:

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["spades.png"] = load Icon("spades.png" ),
    icons["hearts.png"] = load Icon  ( "hearts.png" ) ,
    icons["diamonds.png"] = load Icon("diamonds.png"),
    icons["clubs.png"] = load Icon("clubs.png" ),
}
```

解决方法:

```
// 注意: 并发安全
func Icon(name string) image.Image {
    mu.Lock()
    defer mu.Unlock()
    if icons == nil {
        loadIcons()
    }
    return icon[name]
}
```

使两个 goroutine 可以并发访问 icons:

```
// 注意: 并发安全
func Icon(name string) image.Image {
    mu.RLock()
    if icons != nil {
        icon := icons[name]
        mu.RUnlock()
        return icon
    }
    mu.RUnlock()

    mu.Lock()
    if icons == nil {
        loadIcons()
    }
```

```
    icon := icons[name]
    mu.Unlock()
    return icon
}
```

- sync.Once 提供了这种一次性初始化问题的解决方案
- Once 包含一个布尔变量和一个互斥量，布尔变量记录初始化是否已经完成，互斥量则负责保护这个布尔变量和客户端的数据结构

```
var loadIconsOnce sync.Once
var icons map[string]image.Image

// 并发安全
func Icon(name string) image.Image {
    loadiconsOnce.Do(loadicons)
    return icons[name]
}
```

9.6 竞态检测器

简单地把 - **race** 命令行参数加到 **go build** 、**go run** 、**go test** 命令里边即可使用竞态检测器功能

-它会让编译器为你的应用或测试构建一个修改后的版本，它使用额外的手法高效记录在执行时对共享变量的所有访问，以及读写这些变量的 goroutine 标识，以及所有的同步事件

9.7 示例

9.8 goroutine 与线程

本质上两者差别是量变

三个区分点:

1. 可增长的栈
2. goroutine 调度：Go 包含自己的一个调度器，由特定 Go 语言结构出发
3. GOMAXPROCS：Go 调度器使用 GOMAXPROCS 参数来确定需要使用多少个 OS 线程来同时执行 Go 代码

包和 go 工具

包路径

1. 全局唯一性

2. 应该以互联网域名作为开始

```
import (
"fmt"
"math/rand"
"encoding/json"
"golang.org/x/net/html"
"github.com/go-sql-driver/mysql"
)
```

包声明

```
package xxx
```

包导入

```
import "fmt"
import "os"

import (
    "fmt"
    "os"
)
```

重名导入

```
import (
    "crypto/rand"
    mrand "math/rand" // 通过指定一个不同的名称mrand就避免了冲突
    )
```

每个导入声明从当前包向导入的包建立一个依赖如果这些依赖形成一个循环，go build 工具会报错

空导入

```
import _ "image/png" //注册 PNG 解码器
```

如果导人的包的名字没有在文件中引用，就会产生一个编译错误。但是，有时候，我们必须导人一个包，这仅仅是为了利用其副作用: 对包级别的变量执行初始化表达式求值，并执行它的 init 函数。

go 工具

go 工具（go tool），它用来下载、查询、格式化、构建、测试以及安装 Go 代码包。

工作空间的组织

在安装 go 工具时，通过 GOPATH 环境变量来指定工作空间的根。而在工作空间下有三个子目录，他们分别代表着: src: 源文件目录，每一个包放在一个目录中。pkg: 构建工具存储编译后的包的位置。bin: 放置可执行程序。

包的下载

go get 命令 go get -u 可以指定下载最新的版本的包

包的构建

go build 命令

如果包的名字是 main，则 go build 命令调用链接器在当前目录中创建可执行程序，可执行程序的名字取自包的导入路径的最后一段。

go run 命令可以将包的构建和运行合并起来

```
$ go run quoteags.go one "two three" four\ five
["one" "two three" "four five"]
```

第一个不是以 go 文件结尾的参数会作为 Go 可执行程序的参数列表的开始

go install 令和 go build 非常相似，区别是它会保存每一个包的编译代码和命令，而不是把它们丢弃, 而是保存在 $GOPATH/pkg 目录中。这样于没有改变的包和命令不需要重新编译，从而使后续的构建更加快速。

构建标签

在包的声明之前的注释

```
//+build linux darwin  表明go build只会在构建Linux 或者Mac OS X 系统应用的时候才会对它进行编译
//+build ignore  任何时候都不要编译这个文件:
```

包的文档化

```
// Fprintf 根据格式说明符格式化并写入w
// 返回写入的字节数及可能遇到的错误
func Fprintf(w io.Writer,format string,a ... interface{}) (int,error)
```

包声明的前面的文档注释被认为是整个包的文档注释，且只有一个。如果包的注释比较长可以使用一个注释文件，通常叫做 doc.go

go doc 工具输出在命令行上指定的内容的声明和整个文档注释.

godoc 工具它提供相互链接的 HTM 页面服务，进而提供不少于 go doc 命令的信息

```
$ godoc -http :8000
```

内部包

这是包用来封装 Go 程序最重要的机制。没有导出的标识符只能在同一个包内访问，导出的标识符可以在世界任何地方访问。

go build 工具会特殊对待导人路径中包含路径片段 internal 的情况，这些包叫内部包。

```
net/http
net/http/internal/chunked（内部包）
net/http/httputil
net/url
```

net/http/internal/chunked (内部包) 可以从 net/http 或者 net/http/httputil 导入，但是不能从 net/url 导入。然而，net/url 可以导入 net/http/httputil。

包的查询

go list 工具上报可用包的信息。判断一个包是否存在于工作空间中，如果存在输出它的导人路径。

可使用通配符..., 用来匹配包的导入路径中的任意字串。

```
$ go list ... //枚举一个go工作空间的所有包
$ go list  gopl.io/ch3/... //列举指定子树中的所有包
$ go list ...xml... //一个具体的主题
```

go list 命令获取每个包的完整元数据，而不仅仅是导人路径，并且提供各种对于用户或者其他工具可访问的格式。

-json 标记使 go list 以 JSON 格式输出每个包的完整记录

```
$ go list -json hash
```

-f 标记可以让用户通过 text/template 包提供的模板语言定制输出格式。

testing

go 测试工具

测试是 go 的一个子命令

```
go test ## go run
```

一个 go 的 test 文件主要文件名为"*$_{\text{test}}$.go", 它可以包含三种函数

1. 以 Test 为开头的函数测试要么，通过不通过。
2. 以 Benchmark 开头的函数会产生一个运行时间统计
3. 以 Example 开头的函数是一个样例展示函数

*go test 的机制是去搜索"$_{\textbf{test}}$**.go**" 的文件，并用一个临时 **main** 函数包裹它并运行里面测试函数。

测试函数

测试函数

先来看看简单的示例。

注意: 测试函数必须以 Test 开始

```
func TestName(t *testing.T) { // ...
}
```

先来看个例子，回文的例子 word.go

```
package word
func IsPalindrome(s string) bool {
  for i := range s {
    if s[i] != s[len(s)-1-i] {
      return false
    }
  }
  return true
}
```

我们如何来测试这个回文函数?，创建一个 word$_{\text{test}}$.go

```
package word
import "testing"
func TestPalindrome(t *testing.T) {
  if !IsPalindrome("detartrated") {
    t.Error(`IsPalindrome("detartrated") = false`)
  }
  if !IsPalindrome("kayak") {
    t.Error(`IsPalindrome("kayak") = false`)
  } }
func TestNonPalindrome(t *testing.T) {
  if IsPalindrome("palindrome") {
    t.Error(`IsPalindrome("palindrome") = true`)
  }
}
```

如果正确会是如下

```
$ go test
ok   gopl.io/ch11/word1  0.008s
```

错误是如下

```
$ go test
--- FAIL: TestFrenchPalindrome (0.00s)
word_test.go:28: IsPalindrome("été") = false
--- FAIL: TestCanalPalindrome (0.00s)
word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
FAIL    gopl.io/ch11/word1  0.014s
```

可以输出更为详细的信息 (verbose)

```
$ go test -v
```

用正则表达式运行其中一部分函数

```
$ go test -v -run="French|Canal"
```

把测试用例放到一个表里面挨个测试

```
func TestIsPalindrome(t *testing.T) {
  var tests = []struct {
    input string
    want bool
  }{
    {"", true},
    {"a", true},
    {"aa", true},
    {"ab", false},
    {"kayak", true},
    {"detartrated", true},
    {"A man, a plan, a canal: Panama", true},
    {"Evil I did dwell; lewd did I live.", true},
    {"Able was I ere I saw Elba", true},
    {"été", true},
    {"Et se resservir, ivresse reste.", true},
    {"palindrome", false}, // non-palindrome
    {"desserts", false},   // semi-palindrome
  }
  for _, test := range tests {
    if got := IsPalindrome(test.input); got != test.want {
      t.Errorf("IsPalindrome(%q) = %v", test.input, got)
    }
  }
}
```

注意:

1. 如果测试 t.Errorf 并不会引发 panic 所以如果你想中断，自己写 t.Fatalf 来做。
2. 测试 t.Errorf 应尽量输出有用的信息

随机测试

随机函数要注意保留该有的随机种子。方便问题复现

用一个更为清晰简单但是正确的方法实现功能

写一个函数还产生测试数据

gen->[(test_1,result_1),(test_2,result_2],(test_3, result_3),...,(test_n,result_n))

```
//!+random
// randomPalindrome returns a palindrome whose length and contents
```

```
// are derived from the pseudo-random number generator rng.
func randomPalindrome(rng *rand.Rand) string {
  n := rng.Intn(25) // random length up to 24
  runes := make([]rune, n)
  for i := 0; i < (n+1)/2; i++ {
    r := rune(rng.Intn(0x1000)) // random rune up to '\u0999'
    runes[i] = r
    runes[n-1-i] = r
  }
  return string(runes)
}

func TestRandomPalindromes(t *testing.T) {
  // Initialize a pseudo-random number generator.
  seed := time.Now().UTC().UnixNano()
  t.Logf("Random seed: %d", seed)
  rng := rand.New(rand.NewSource(seed))

  for i := 0; i < 1000; i++ {
    p := randomPalindrome(rng)
    if !IsPalindrome(p) {
      t.Errorf("IsPalindrome(%q) = false", p)
    }
  }
}
```

测试命令 (运用了白盒测试)

测试带命令行的程序

命令行函数

```
var (
  n = flag.Bool("n", false, "omit trailing newline")
  s = flag.String("s", " ", "separator")
)

var out io.Writer = os.Stdout // modified during testing

func main() {
  flag.Parse()
  if err := echo(!*n, *s, flag.Args()); err != nil {
    fmt.Fprintf(os.Stderr, "echo: %v\n", err)
    os.Exit(1)
  }
}

func echo(newline bool, sep string, args []string) error {
  fmt.Fprint(out, strings.Join(args, sep))
  if newline {
    fmt.Fprintln(out)
  }
  return nil
}
```

如何测试这个函数?

```
>> go run echo.go -n=false -s="'" some some some
>> some'some'some
```

测试代码

```
func TestEcho(t *testing.T) {
  var tests = []struct {
    newline bool
    sep     string
    args    []string
    want    string
  }{
    {true, "", []string{}, "\n"},
    {false, "", []string{}, ""},
    {true, "\t", []string{"one", "two", "three"}, "one\ttwo\tthree\n"},
    {true, ",", []string{"a", "b", "c"}, "a,b,c\n"},
    {false, ":", []string{"1", "2", "3"}, "1:2:3"},
  }

  for _, test := range tests {
    descr := fmt.Sprintf("echo(%v, %q, %q)",
      test.newline, test.sep, test.args)

    out = new(bytes.Buffer) // captured output 这里修改了包体变量 out 原本为 stdout 现在为一个 buffer
    if err := echo(test.newline, test.sep, test.args); err != nil {
      t.Errorf("%s failed: %v", descr, err)
      continue
    }
    got := out.(*bytes.Buffer).String()
    if got != test.want {
      t.Errorf("%s = %q, want %q", descr, got, test.want)
    }
  }
}
```

NOTE:

1. 在测试的代码里面不要调用 log.Fatal 或者 os.Exit，因为这两个调用会阻止跟踪的过程，这两个函数的调用可以认为是 main 函数的特权。

白盒测试

1. 回文测试是黑盒测试
2. echo 测试是白盒测试（因为我们修改 echo 的变量 out，所以实际上我们是知道 echo 程序的代码结构的）

邮件空间预警

```
var usage = make(map[string]int64)

func bytesInUse(username string) int64 { return usage[username] }

// Email sender configuration.
// NOTE: never put passwords in source code!
const sender = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"

const template = `Warning: you are using %d bytes of storage,
%d%% of your quota.`

func CheckQuota(username string) {
  used := bytesInUse(username)
  const quota = 1000000000 // 1GB
  percent := 100 * used / quota
  if percent < 90 {
```

```
    return // OK
  }
  msg := fmt.Sprintf(template, used, percent)
  auth := smtp.PlainAuth("", sender, password, hostname)
  err := smtp.SendMail(hostname+":587", auth, sender,
    []string{username}, []byte(msg))
  if err != nil {
    log.Printf("smtp.SendMail(%s) failed: %s", username, err)
  }
}
```

函数功能：当用户的使用空间炒股 90percent 的时候，发送一封邮件给用户。

为了测试这个功能是否正确。我们改变源码。把 notifyUser 抽象出来做一个发送邮件的函数

```
var usage = make(map[string]int64)

func bytesInUse(username string) int64 { return usage[username] }

// E-mail sender configuration.
// NOTE: never put passwords in source code!
const sender = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"

const template = `Warning: you are using %d bytes of storage,
%d%% of your quota.`

//!+factored
var notifyUser = func(username, msg string) {
  auth := smtp.PlainAuth("", sender, password, hostname)
  err := smtp.SendMail(hostname+":587", auth, sender,
    []string{username}, []byte(msg))
  if err != nil {
    log.Printf("smtp.SendMail(%s) failed: %s", username, err)
  }
}

func CheckQuota(username string) {
  used := bytesInUse(username)
  const quota = 1000000000 // 1GB
  percent := 100 * used / quota
  if percent < 90 {
    return // OK
  }
  msg := fmt.Sprintf(template, used, percent)
  notifyUser(username, msg)
}
```

测试 90 功能是否正确，替换掉原来的 notifyUser，不做实际发送操作。

```
func TestCheckQuotaNotifiesUser(t *testing.T) {
  var notifiedUser, notifiedMsg string
  notifyUser = func(user, msg string) {
    notifiedUser, notifiedMsg = user, msg
  }

  const user = "joe@example.org"
  usage[user] = 980000000 // simulate a 980MB-used condition

  CheckQuota(user)
```

```
  if notifiedUser == "" && notifiedMsg == "" {
    t.Fatalf("notifyUser not called")
  }
  if notifiedUser != user {
    t.Errorf("wrong user (%s) notified, want %s",
      notifiedUser, user)
  }
  const wantSubstring = "98% of your quota"
  if !strings.Contains(notifiedMsg, wantSubstring) {
    t.Errorf("unexpected notification message <<%s>>, "+
      "want substring %q", notifiedMsg, wantSubstring)
  }
}

//!-test
```

恢复现场

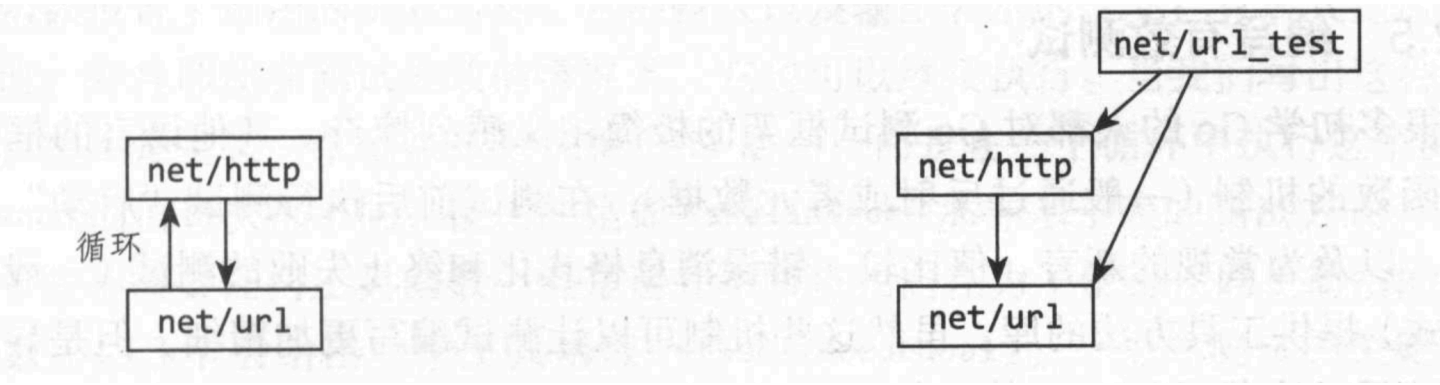
```
func TestCheckQuotaNotifiesUser(t *testing.T) {
  // Save and restore original notifyUser.
  saved := notifyUser
  defer func() { notifyUser = saved }()

  // Install the test's fake notifyUser.
  var notifiedUser, notifiedMsg string
  notifyUser = func(user, msg string) {
    notifiedUser, notifiedMsg = user, msg
  }
  // ...rest of test...
}
```

总结一下白盒测试的过程

1. 替换掉原本的函数
2. 测试
3. 回复现场

外部测试包

有一个 **url** 的测试函数需要用到 **http**。那么需要在 **net/url** 包中声明这个测试函数会导致包循环引用，如图中向上的箭头所示，但是 **10.1** 节讲过，**Go** 规范禁止循环引用.



包内测试

和代码写相同的包名

包外测试

比如写

```
package url_test //和 url 不是一个包，但是可以引入 url 和 http 包来测试 url 包
```

如何判断一个包的文件种类

1. 一般文件

   ```
   $ go list -f={{.GoFiles}} fmt
   [doc.go format.go print.go scan.go]
   ```

2. 包内测试文件

   ```
   $ go list -f={{.TestGoFiles}} fmt
   [export_test.go]
   ```

3. 包外测试

   ```
   $ go list -f={{.XTestGoFiles}} fmt
   [fmt_test.go scan_test.go stringer_test.go]
   ```

如何让包外测试访问到包内的变量?

专门写一个 $\text{export}_{\text{test}}$.go 来暴露包内测试的变量。这样外部测试包就可以访问包内的变量了。

编写有效测试

1. go 的测试非常简单，go 的逻辑是，让写代码的人来维护代码。所以代码和测试代码差不多。
2. 不要在测试里面 panic，这样是无效的测试。因为没有为维护者提供信息。

避免脆弱测试

测试如果不稳定，那么程序编写人员会非常难受。

覆盖率

```
go test -v -run=Coverage gopl.io/ch7/eval
go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
-covermode=count ## 可以显示代码被执行的次数
```

覆盖率不是总会百分之 100 执行，比如有的代码可能永远不会执行到

benchmark 函数

1. 函数名要以 Benchmark 开头
2. *testing.T 作为函数参数

```
import "testing"
func BenchmarkIsPalindrome(b *testing.B) {
  for i := 0; i < b.N; i++ {
    IsPalindrome("A man, a plan, a canal: Panama")
  }
}
```

```
go test -bench=.
go test -bench=. -benchmem ## 可以查看内存情况。
```

最后会产生报告，其中 N 是测试命令自己会根据程序运行情况调整大小的。最后可以从报告中看出来大小。

性能剖析

1. 不要过早优化。
2. 如何优化关键代码？使用工具剖析，不要使用靠自己的直觉。因为关键代码很有可能和你的直觉不一样。

三中性能剖析工具

```
$ go test -cpuprofile=cpu.out     ## cpu
$ go test -blockprofile=block.out ## 阻塞
$ go test -memprofile=mem.out     ## 内存
```

运行时内存剖析

1. 长时间运行的程序分析，无法如上面那样仅仅启动分析就行。
2. 可以使用 runtime API 来启动

性能剖析工具

如果运行时产生了数据，那么就可以用工具来分析

```
go tool pprof
```

更过的关于性能剖析的方法，请参考阅读Profiling Go Programs【博客】

Example 函数

```
func ExampleIsPalindrome() {
  fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
  fmt.Println(IsPalindrome("palindrome"))
  // Output:
  // true
  // false
}
```

1. 示例作用
2. 如果包含 Output 测试函数回去运行它，并检测输出是否一致。
3. 展示在 web-based 的应用中。

反射

- Go 语言提供了一种机制，在编译时不知道类型的情况下，可更新变量、在运行时查看值、调用方法以及直接对它们的布局进行操作，这种机制称为反射（reflection）

12.1 为什么使用反射

- 有时我们需要写一个函数有能力统一处理各种值类型的函数，而这些类型可能无法共享同一个接口，也可能布局未知，也有可能这个类型在我们设计函数时还不存在
- 例子：fmt.Printf 中的格式化逻辑，它可以输出任意类型的任意值，甚至是用户自定义的一个类型

```
func Sprint(x interface{}) string {
    type stringe interface{
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
        return x
    case int:
        return strconv.Itoa(x)
    // 对 int16、uint32 等类型做类似处理
    case bool:
        if x{
            return "true"
        }
        return "false"
    default:
        // arry, chan, func, map, pointer, slice, struct
        return "???"
    }
}
```

问题：如何处理类似 []float64、map[string][]string 的其他类型、以及自己命名的类型，比如 url.Values? - 当我们无法透视一个未知类型的布局、代码无法继续时时，就需要反射了

12.2 reflect.Type 和 reflect.Value

- reflect 包定义了两个重要的类型：Type 和 Value

reflect.Type

- reflect.TypeOf 函数接受任何的 interface{} 参数，并且把接口中的动态类型以 reflect.Type 形式返回。

```
t := reflect.TypeOf(3)  // 一个 reflect.Type
fmt.Println(t.String()) // "int"
fmt.Println(t)          // "int"
```

- reflect.Type 满足 fmt.Stringer, fmt.Printf 提供了一个简写方式%T,内部实现就使用了 reflect.TypeOf: `fmt.Printf("%T\n", e) // "int"`

reflect.Value

- reflect.ValueOf 函数接受任意的 interface{} 并将接口的动态值以 reflect.Value 的形式返回

```
v := reflect.ValueOf(3)  // 一个reflect.Value
fmt.Println(v)           // "3"
fmt.Printf("%v\n", v)    // "3"
fmt.Println(v. String()) // 注意: "<int Value>"
```

- reflect.Value 也满足 fmt.Stringer, fmt 包的%v 功能会对 reflect.Value 进行特殊处理
- 调用 Value 的 Type 方法会把它的类型以 reflect.Type 方式返回:

```
t := v.Type()           // 一个reflect.Type
fmt.Println(t.String()) // "int"
```

- reflect.ValueOf 的逆操作是 reflect.Value.Interface 方法。它返回一个 interface{} 接口值，与 reflect.Value 包含同一个具体值。

```
v := reflect.ValueOf(3) // a reflect.Value
x := v.Interface()      // an interface{}
i := x.(int)            // an int x.(int)是一个类型断言
fmt.Printf("%d\n", i)   // "3"
```

- 格式化函数的第二次尝试

```
package format

import (
    "reflect"
    "strconv"
)

// Any formats any value as a string.
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}

// formatAtom formats a value without inspecting its internal structure.
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...floating-point and complex cases omitted for brevity...
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}
```

该函数把每个值当做一个没有内部结构且不可分割的物体。对于聚合类型（结构体和数组）以及接口，它只输出了值的类型；对于引用类型（通道、函数、指针、slice 和 map），它输出了类型和以十六进制表示的引用地址

```
    var x int64 = 1
    var d time.Duration = 1 * time.Nanosecond
    fmt.Println(format.Any(x))                  // "1"
    fmt.Println(format.Any(d))                  // "1"
    fmt.Println(format.Any([]int64{x}))         // "[]int64 0x8202b87b0"
    fmt.Println(format.Any([]time.Duration{d})) // "[]time.Duration 0x8202b87e0"
```

12.5 使用 reflect.Value 来设置值

- reflect.Value 分为可寻址的和不可寻址的两类

```
x := 2                   // 值类型变量？
```

```
a := reflect.ValueOf(2)     // 2   int   no
b := reflect.ValueOf(x)     // 2   int   no
c := reflect.ValueOf(&x)    // &x *int   no
d := c.Elem()               // 2   int   yes (x)
```

前三个都是保存的副本，不可寻址，所以可以通过 reflect.ValueOf(&x).Elem() 来获得任意变量 x 可寻址的 Value 值

- 可以通过变量的 CanAddr 方法来询问 reflect.Value 变量是否可寻址: `fmt.Println(x.CanAddr())`
- 从一个可寻址的 reflect. Value() 获取变量需要三步。首先，调用 Addr()，返回一个 Value，其中包含一个指向变量的指针，接下来，在这个 Value 上调用 Interface()，会返回一个包含这个指针的 Interface{} 值。最后，如果我们知道变量的类型，我们可以使用类型断言来把接口内容转换为一个普通指针。之后就可以通过这个指针来更新变量了

```
x := 2
d := reflect.ValueOf(&x).Elem()     // d 代表变量 x
p := d.Addr().Interface().(*int)    // px := &x
*px = 3                             // x = 3
fmt.Println(x)                      // "3"
```

- 还可以直接通过可寻址的 reflect.Value 来更新变量,不用通过指针,而是直接调用 reflect.Value.Set 方法:`d.Set(reflect.ValueOf(4))` 以及一些特化的 Set 变种 SetInt 、SetUint 、SetString 、SetFloat 等: `d.SetInt(3)`

注: 1. 平常由编译器来检查的那些可赋值性条件，在这种情况下则是在运行时由 Set 方法来检查 2. 变量和值类型需一致 3. 再不可寻址的 reflect.Value 上调用会崩溃 4. 反射可以读取到未导出的结构字段的值，但是不能更新它们，更新前可以用 CanSet 方法判断

12.9 注意事项

慎用反射的三个原因:

1. 基于反射的代码是很脆弱的。能导致编译器报告类型错误的每种写法，在反射中都有一个对应的误用方法。编译器在编译时就能向你报告这个错误，而反射错误则要等到执行时才以崩溃的方式来报告
2. 类型其实也算是某种形式的文档，而反射的相关操作则无法做静态类型检查，所以大量使用反射的代码是很难理解的
3. 基于反射的函数会比为特定类型优化的函数慢一两个数量级

低级编程接口

go 的安全保护机制分为动态和静态检查两部分。

1. 静态如类型检查
2. 动态如数组越界

另外在 go 中无法获得 goroutine 所在的实际线程 id，因为 go 会在不同的时候调度不同的线程给 goroutine 去运行。也不能获取指针对应的内存地址。因为在不同的时候资源回收之后，动态内存管理可能挪动数据位置，实际指针地址会发生变动。

unsafe package

为什么要使用这些低级的编程接口

1. 效率
2. 和其它语言库交互
3. 实现不能由纯 go 语言实现的函数

unsafe package

1. 是由编译器实现的。
2. 暴露了 go 的内存结构。

unsafe.Sizeof, Alignof, and Offsetof

Sizeof

1. Sizeof 只报告数据结构里面的固定部分大小。比如字符串的指针和长度。而字符串具体占用了多大空间则不会报告。
2. Sizeof 报告的大小，最小是里面固定字段的大小总和。会多余是因为字节对齐。

Type	Size
`bool`	1 byte
`intN`, `uintN`, `floatN`, `complexN`	*N* / 8 bytes (for example, `float64` is 8 bytes)
`int`, `uint`, `uintptr`	1 word
`*T`	1 word
`string`	2 words (data, len)
`[]T`	3 words (data, len, cap)
`map`	1 word
`func`	1 word
`chan`	1 word
`interface`	2 words (type, value)

1. 编译器不一定按照你定义字节的顺序来分配内存或对齐。目前的编译器还不会帮你调整更好的顺序。

```
                                                // 64-bit    32-bit
    struct{ bool; float64; int16 }              // 3 words   4 words
    struct{ float64; int16; bool }              // 2 words   3 words
    struct{ bool; int16; float64 }              // 2 words   3 words
```

Alignof

参数需要的对齐字节数

Offsetof

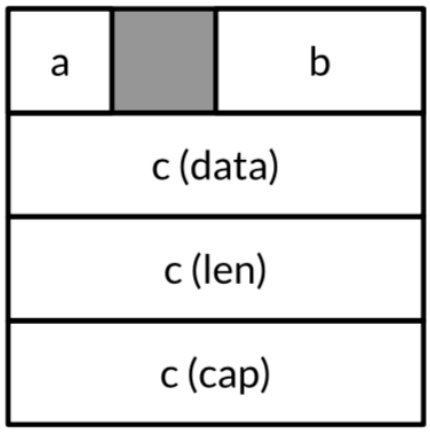
参数的偏移地址

总结

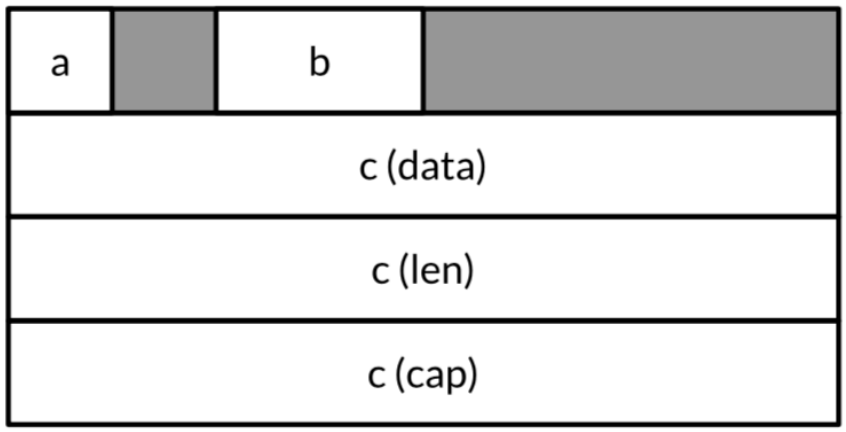
我们来看一个具体的例子。

```
var x struct {
  a bool
  b int16
  c []int
}
```

它的内存结构图



(32-bit)



(64-bit)

调用三个函数的结果

```
//Typical 32-bit platform:
Sizeof(x)   = 16  Alignof(x)   = 4
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 12  Alignof(x.c) = 4  Offsetof(x.c) = 4
//Typical 64-bit platform:
Sizeof(x)   = 32  Alignof(x)   = 8
Sizeof(x.a) = 1   Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 24  Alignof(x.c) = 8  Offsetof(x.c) = 8
```

Pointer

1. 常规指针如 * T, 可以转换为 unsafe.Pointer, 但是反之则不一定合法。因为你可能讲一个 uint16 转给一个 uint64. 这是不合法的。

```
//通过把浮点 64 位转为 uint64. 来看浮点的内存结构
package math
func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }
fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

1. unsafe.Pointer 也可以转换为 uintptr.

   - 这个 uintptr 只是一个数字，指向一个内存地址，这是危险的。因为 go 的资源回收，可能会移动变量那么 uintptr 指向的位置可能变成一个非法位置。unsafe.Pointer 却不是这样的，go 的内存管理会位置 unsafe.Pointer 始终指向正确的位置。
   - 有的 uintptr 值，对应的是不合法的内存值。

   正确代码

   ```
   gopl.io/ch13/unsafeptr
   var x struct {
     a bool
     b int16
     c []int }
   // equivalent to pb := &x.b
   pb := (*int16)(unsafe.Pointer(
     uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
   *pb = 42
   fmt.Println(x.b) // "42"
   ```

   错误代码

   ```
   // NOTE: subtly incorrect!
   tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
   pb := (*int16)(unsafe.Pointer(tmp))
   *pb = 42
   ```

   ```
   //错误
   pT := uintptr(unsafe.Pointer(new(T))) // NOTE: w
   // 资源回收了但是 pT 还指向一个错误的位置
   ```

2. call 外部库的时候。应该立即将 uintptr 转换为 unsafe.Pointer

   ```
   package reflect
   func (Value) Pointer() uintptr
   func (Value) UnsafeAddr() uintptr
   func (Value) InterfaceData() [2]uintptr // (index 1)
   ```

3. goroutine 的 stack 会增长，所以会发生移动。所以内存地址是会变动的。

Example:Deep Equivalence

在 reflect 模块中，有一个 deep equivalence. 他可以深度比较两个对象的是否相同。对于 nil 和空的字典比较是不等。比如

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"
var c, d map[string]int = nil, make(map[string]int)
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

我们来自己实现一个 deep equal 但稍微不同的是上面的比较我们得出相等

```
// Package equal provides a deep equivalence relation for arbitrary values.
package equal

import (
  "reflect"
  "unsafe"
)
```

```
//!+
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
  if !x.IsValid() || !y.IsValid() {
    return x.IsValid() == y.IsValid()
  }
  if x.Type() != y.Type() {
    return false
  }

  // ...cycle check omitted (shown later)...

  //!-
  //!+cyclecheck
  // cycle check
  if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
      return true // identical references
    }
    c := comparison{xptr, yptr, x.Type()}
    if seen[c] {
      return true // already seen
    }
    seen[c] = true
  }
  //!-cyclecheck
  //!+
  switch x.Kind() {
  case reflect.Bool:
    return x.Bool() == y.Bool()

  case reflect.String:
    return x.String() == y.String()

  // ...numeric cases omitted for brevity...

  //!-
  case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32,
    reflect.Int64:
    return x.Int() == y.Int()

  case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32,
    reflect.Uint64, reflect.Uintptr:
    return x.Uint() == y.Uint()

  case reflect.Float32, reflect.Float64:
    return x.Float() == y.Float()

  case reflect.Complex64, reflect.Complex128:
    return x.Complex() == y.Complex()
  //!+
  case reflect.Chan, reflect.UnsafePointer, reflect.Func:
    return x.Pointer() == y.Pointer()

  case reflect.Ptr, reflect.Interface:
    return equal(x.Elem(), y.Elem(), seen)

  case reflect.Array, reflect.Slice:
```

```
    if x.Len() != y.Len() {
      return false
    }
    for i := 0; i < x.Len(); i++ {
      if !equal(x.Index(i), y.Index(i), seen) {
        return false
      }
    }
    return true

  // ...struct and map cases omitted for brevity...
  //!-
  case reflect.Struct:
    for i, n := 0, x.NumField(); i < n; i++ {
      if !equal(x.Field(i), y.Field(i), seen) {
        return false
      }
    }
    return true

  case reflect.Map:
    if x.Len() != y.Len() {
      return false
    }
    for _, k := range x.MapKeys() {
      if !equal(x.MapIndex(k), y.MapIndex(k), seen) {
        return false
      }
    }
    return true
    //!+
  }
  panic("unreachable")
}

//!-

//!+comparison
// Equal reports whether x and y are deeply equal.
//!-comparison
//
// Map keys are always compared with ==, not deeply.
// (This matters for keys containing pointers or interfaces.)
//!+comparison
func Equal(x, y interface{}) bool {
  seen := make(map[comparison]bool)
  return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
  x, y unsafe.Pointer
  t    reflect.Type
}

//!-comparison
```

Calling C Code with cgo

有两个包可以用来处理调用不同代码的一个是 cgo，一个是 SWIG（更复杂）

调用 C

1. 如果简单，可以考虑用 go 实现
2. 如果不太复杂，可以考虑用子进程的方法调用
3. 复杂，和灵活的调用使用 cgo

import C

1. C 不是一个包，而是一个预处理命令。运行在编译之前。

```
// Package bzip provides a writer that uses bzip2 compression (bzip.org).
package bzip
/*
     #cgo CFLAGS: -I/usr/include
     #cgo LDFLAGS: -L/usr/lib -lbz2
     #include <bzlib.h>
     int bz2compress(bz_stream *s, int action,
                     char *in, unsigned *inlen, char *out, unsigned *outlen);
 */
import "C"

import ( "io"
  "unsafe" )
type writer struct {
  w      io.Writer // underlying output stream
  stream *C.bz_stream
  outbuf [64 * 1024]byte
}
// NewWriter returns a writer for bzip2-compressed streams.
func NewWriter(out io.Writer) io.WriteCloser {
  const (
    blockSize  = 9
    verbosity = 0
    workFactor = 30
  )
  w := &writer{w: out, stream: new(C.bz_stream)}
  C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
  return w
}
```

工作原理

1. cgo 会生成一个临时的包
2. cgo 会调用 C 编译器去编译处理
3. 最后这些可用的部分会放到 C 中。调用者只要调用这个 C 就可以了。
4. 可以是用 #cgo，在注释中引入编译选项。

更多

1. 当然也可以把 go 的东西编译为静态库或者动态库，供 C 调用。
2. cgo 的内容非常多，更多参考cgo
3. cap 是容量的，和 len 的不同在于。len 是长度，是现有 data 的长度。cap 是最多可以容纳的极限。

Another Word of Caution

unsafe 模块比 reflect 模块更不建议使用。使用的时候要更为慎重