

2015年09月07日 • GO by Kyle Quest

Go的50度灰：Golang新开发者要注意的陷阱和常见错误

目录 [–]

1 初级

- 1.1 开大括号不能放在单独的一行
- 1.2 未使用的变量
- 1.3 未使用的Imports
- 1.4 简式的变量声明仅可以在函数内部使用
- 1.5 使用简式声明重复声明变量
- 1.6 偶然的变量隐藏Accidental Variable Shadowing
- 1.7 不使用显式类型，无法使用“nil”来初始化变量
- 1.8 使用“nil” Slices and Maps
- 1.9 Map的容量
- 1.10 字符串不会为nil
- 1.11 Array函数的参数
- 1.12 在Slice和Array使用“range”语句时的出现的不希望得到的值
- 1.13 Slices和Arrays是一维的
- 1.14 访问不存在的Map Keys
- 1.15 Strings无法修改
- 1.16 String和Byte Slice之间的转换
- 1.17 String和索引操作
- 1.18 字符串不总是UTF8文本
- 1.19 字符串的长度
- 1.20 在多行的Slice、Array和Map语句中遗漏逗号
- 1.21 log.Fatal和log.Panic不仅仅是Log
- 1.22 内建的数据结构操作不是同步的
- 1.23 String在“range”语句中的迭代值

- 1.24 对Map使用"for range"语句迭代
- 1.25 "switch"声明中的失效行为
- 1.26 自增和自减
- 1.27 按位NOT操作
- 1.28 操作优先级的差异
- 1.29 未导出的结构体不会被编码
- 1.30 有活动的Goroutines下的应用退出
- 1.31 向无缓存的Channel发送消息, 只要目标接收者准备好就会立即返回
- 1.32 向已关闭的Channel发送会引起Panic
- 1.33 使用"nil" Channels
- 1.34 传值方法的接收者无法修改原有的值

2 中级

- 2.1 关闭HTTP的响应
- 2.2 关闭HTTP的连接
- 2.3 比较Structs, Arrays, Slices, and Maps
- 2.4 从Panic中恢复
- 2.5 在Slice, Array, and Map "range"语句中更新引用元素的值
- 2.6 在Slice中"隐藏"数据
- 2.7 Slice的数据"毁坏"
- 2.8 陈旧的(Stale)Slices
- 2.9 类型声明和方法
- 2.10 从"for switch"和"for select"代码块中跳出
- 2.11 "for"声明中的迭代变量和闭包
- 2.12 Defer函数调用参数的求值
- 2.13 被Defer的函数调用执行
- 2.14 失败的类型断言
- 2.15 阻塞的Goroutine和资源泄露

3 高级

- 3.1 使用指针接收方法的值的实例
- 3.2 更新Map的值
- 3.3 "nil" Interfaces和"nil" Interfaces的值
- 3.4 栈和堆变量

3.5 GOMAXPROCS, 并发, 和并行

3.6 读写操作的重排顺序

3.7 优先调度

原文: [50 Shades of Go: Traps, Gotchas, and Common Mistakes for New Golang Devs](#)

翻译: [Go的50度灰：新Golang开发者要注意的陷阱、技巧和常见错误](#), 译者: 影风LEY

Go是一门简单有趣的语言，但与其他语言类似，它会有一些技巧。。。这些技巧的绝大部分并不是Go的缺陷造成的。如果你以前使用的是其他语言，那么这其中的有些错误就是很自然的陷阱。其它的是由错误的假设和缺少细节造成的。

如果你花时间学习这门语言，阅读官方说明、wiki、邮件列表讨论、大量的优秀博文和Rob Pike的展示，以及源代码，这些技巧中的绝大多数都是显而易见的。尽管不是每个人都是以这种方式开始学习的，但也没关系。如果你是Go语言新人，那么这里的信息将会节约你大量的调试代码的时间。

初级

1/ 开大括号不能放在单独的一行

在大多数其他使用大括号的语言中，你需要选择放置它们的位置。Go的方式不同。你可以为此感谢下自动分号的注入（没有预读）。是的，Go中也是有分号的：-)

失败的例子：

```
1 package main
2
3 import "fmt"
4
5 func main()
6 { //error, can't have the opening brace on a separate line
7     fmt.Println("hello there!")
8 }
```

编译错误：

```
/tmp/sandbox826898458/main.go:6: syntax error: unexpected semicolon or newline before  
{
```

有效的例子：

```
1 package main  
2  
3 import "fmt"  
4  
5 func main() {  
6     fmt.Println("works!")  
7 }
```

2/ 未使用的变量

如果你有未使用的变量，代码将编译失败。当然也有例外。在函数内一定要使用声明的变量，但未使用的全局变量是没问题的。

如果你给未使用的变量分配了一个新的值，代码还是会编译失败。你需要在某个地方使用这个变量，才能让编译器愉快的编译。

Fails:

```
1 package main  
2  
3 var gvar int //not an error  
4  
5 func main() {  
6     var one int //error, unused variable  
7     two := 2 //error, unused variable  
8     var three int //error, even though it's assigned 3 on the  
9     three = 3  
10 }
```

Compile Errors:

/tmp/sandbox473116179/main.go:6: one declared and not used

/tmp/sandbox473116179/main.go:7: two declared and not used

/tmp/sandbox473116179/main.go:8: three declared and not used

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var one int
7     _ = one
8
9     two := 2
10    fmt.Println(two)
11
12    var three int
13    three = 3
14    one = three
15
16    var four int
17    four = four
18 }
```

另一个选择是注释掉或者移除未使用的变量：-)

3/ 未使用的Imports

如果你引入一个包，而没有使用其中的任何函数、接口、结构体或者变量的话，代码将会编译失败。

你可以使用`goimports`来增加引入或者移除未使用的引用：

```
1 | $ go get golang.org/x/tools/cmd/goimports
```

如果你真的需要引入的包，你可以添加一个下划线标记符，`_`，来作为这个包的名字，从而避免编译失败。下滑线标记符用于引入，但不使用。

Fails:

```
1 package main
2
3 import (
4     "fmt"
5     "log"
6     "time"
7 )
8
9 func main() {
10 }
```

Compile Errors:

```
/tmp/sandbox627475386/main.go:4: imported and not used: "fmt"
/tmp/sandbox627475386/main.go:5: imported and not used: "log"
/tmp/sandbox627475386/main.go:6: imported and not used: "time"
```

Works:

```
1 package main
2
3 import (
4     _ "fmt"
5     "log"
6     "time"
7 )
8
9 var _ = log.Println
10
11 func main() {
12     _ = time.Now
13 }
```

另一个选择是移除或者注释掉未使用的imports：-)

4/ 简式的变量声明仅可以在函数内部使用

Fails:

```
1 package main
2
3 myvar := 1 //error
4
5 func main() {
6 }
```

Compile Error:

```
/tmp/sandbox265716165/main.go:3: non-declaration statement outside function body
```

Works:

```
1 package main
2
3 var myvar = 1
4
5 func main() {
6 }
```

5/ 使用简式声明重复声明变量

你不能在一个单独的声明中重复声明一个变量，但在多变量声明中这是允许的，其中至少要有一个新的声明变量。

重复变量需要在相同的代码块内，否则你将得到一个隐藏变量。

Fails:

```
1 package main
2
3 func main() {
4     one := 0
5     one := 1 //error
```

```
6 | }
```

Compile Error:

```
/tmp/sandbox706333626/main.go:5: no new variables on left side of :=
```

Works:

```
1 | package main
2 |
3 | func main() {
4 |     one := 0
5 |     one, two := 1,2
6 |
7 |     one,two = two,one
8 | }
```

6/ 偶然的变量隐藏Accidental Variable Shadowing

短式变量声明的语法如此的方便（尤其对于那些使用过动态语言的开发者而言），很容易让人把它当成一个正常的分配操作。如果你在一个新的代码块中犯了这个错误，将不会出现编译错误，但你的应用将不会做你所期望的事情。

```
1 | package main
2 |
3 | import "fmt"
4 |
5 | func main() {
6 |     x := 1
7 |     fmt.Println(x)    //prints 1
8 |     {
9 |         fmt.Println(x) //prints 1
10 |        x := 2
11 |        fmt.Println(x) //prints 2
12 |    }
13 |     fmt.Println(x)    //prints 1 (bad if you need 2)
14 | }
```


即使对于经验丰富的Go开发者而言，这也是一个非常常见的陷阱。这个坑很容易挖，但又很难发现。

你可以使用 `vet` 命令来发现一些这样的问题。默认情况下，`vet` 不会执行这样的检查，你需要设置 `-shadow` 参数：

```
go tool vet -shadow your_file.go。
```

7/ 不使用显式类型，无法使用“nil”来初始化变量

`nil` 标志符用于表示interface、函数、maps、slices和channels的“零值”。如果你不指定变量的类型，编译器将无法编译你的代码，因为它猜不出具体的类型。

Fails:

```
1 package main
2
3 func main() {
4     var x = nil //error
5
6     _ = x
7 }
```

Compile Error:

```
/tmp/sandbox188239583/main.go:4: use of untyped nil
```

Works:

```
1 package main
2
3 func main() {
4     var x interface{} = nil
5
6     _ = x
7 }
```

8/ 使用“nil” Slices and Maps

在一个 `nil` 的 slice 中添加元素是没问题的，但对一个 map 做同样的事将会生成一个运行时的 panic。

Works:

```
1 package main
2
3 func main() {
4     var s []int
5     s = append(s, 1)
6 }
```

Fails:

```
1 package main
2
3 func main() {
4     var m map[string]int
5     m["one"] = 1 //error
6
7 }
```

9/ Map的容量

你可以在map创建时指定它的容量，但你无法在map上使用 `cap()` 函数。

Fails:

```
1 package main
2
3 func main() {
4     m := make(map[string]int, 99)
5     cap(m) //error
6 }
```

Compile Error:

```
/tmp/sandbox326543983/main.go:5: invalid argument m (type map[string]int) for cap
```

10 / 字符串不会为 `nil`

这对于经常使用 `nil` 分配字符串变量的开发者而言是个需要注意的地方。

Fails:

```
1 package main
2
3 func main() {
4     var x string = nil //error
5
6     if x == nil { //error
7         x = "default"
8     }
9 }
```

Compile Errors:

```
/tmp/sandbox630560459/main.go:4: cannot use nil as type string in assignment
```

```
/tmp/sandbox630560459/main.go:6: invalid operation: x == nil (mismatched types string
and nil)
```

Works:

```
1 package main
2
3 func main() {
4     var x string //defaults to "" (zero value)
5
6     if x == "" {
7         x = "default"
8     }
9 }
```

11 / Array函数的参数

如果你是一个C或则C++开发者，那么数组对你而言就是指针。当你向函数中传递数组时，函数会参照相同的内存区域，这样它们就可以修改原始的数据。Go中的数组是数值，因此当你向函数中传递数组时，函数会得到原始数组数据的一份复制。如果你打算更新数组的数据，这将会是个问题。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := [3]int{1,2,3}
7
8     func(arr [3]int) {
9         arr[0] = 7
10        fmt.Println(arr) //prints [7 2 3]
11    }(x)
12
13    fmt.Println(x) //prints [1 2 3] (not ok if you need [7 2 3])
14 }
```

如果你需要更新原始数组的数据，你可以使用数组指针类型。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := [3]int{1,2,3}
7
8     func(arr *[3]int) {
9         (*arr)[0] = 7
10        fmt.Println(arr) //prints [7 2 3]
11    }(&x)
12
13    fmt.Println(x) //prints [7 2 3]
```

```
14 }  
}
```

另一个选择是使用slice。即使你的函数得到了slice变量的一份拷贝，它依旧会参照原始的数据。

```
1 package main  
2  
3 import "fmt"  
4  
5 func main() {  
6     x := []int{1,2,3}  
7  
8     func(arr []int) {  
9         arr[0] = 7  
10        fmt.Println(arr) //prints [7 2 3]  
11    }(x)  
12  
13    fmt.Println(x) //prints [7 2 3]  
14 }
```

12/ 在Slice和Array使用“range”语句时的出现的不希望得到的值

如果你在其他的语言中使用“for-in”或者“foreach”语句时会发生这种情况。Go中的“range”语法不太一样。它会得到两个值：第一个值是元素的索引，而另一个值是元素的数据。

Bad:

```
1 package main  
2  
3 import "fmt"  
4  
5 func main() {  
6     x := []string{"a","b","c"}  
7  
8     for v := range x {  
9         fmt.Println(v) //prints 0, 1, 2  
10    }
```

```
11 }  
}
```

Good:

```
1 package main  
2  
3 import "fmt"  
4  
5 func main() {  
6     x := []string{"a","b","c"}  
7  
8     for _, v := range x {  
9         fmt.Println(v) //prints a, b, c  
10    }  
11 }
```

13/ Slices和Arrays是一维的

看起来Go好像支持多维的Array和Slice，但不是这样的。尽管可以创建数组的数组或者切片的切片。对于依赖于动态多维数组的数值计算应用而言，Go在性能和复杂度上还相距甚远。

你可以使用纯一维数组、“独立”切片的切片，“共享数据”切片的切片来构建动态的多维数组。

如果你使用纯一维的数组，你需要处理索引、边界检查、当数组需要变大时的内存重新分配。

使用“独立”slice来创建一个动态的多维数组需要两步。首先，你需要创建一个外部的slice。然后，你需要分配每个内部的slice。内部的slice相互之间独立。你可以增加减少它们，而不会影响其他内部的slice。

```
1 package main  
2  
3 func main() {  
4     x := 2  
5     y := 4  
6  
7     table := make([][]int,x)
```

```
8     for i:= range table {
9         table[i] = make([]int,y)
10    }
11 }
```

使用“共享数据”slice的slice来创建一个动态的多维数组需要三步。首先，你需要创建一个用于存放原始数据的数据“容器”。然后，你再创建外部的slice。最后，通过重新切片原始数据slice来初始化各个内部的slice。

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      h, w := 2, 4
7
8      raw := make([]int,h*w)
9      for i := range raw {
10         raw[i] = i
11     }
12     fmt.Println(raw,&raw[4])
13     //prints: [0 1 2 3 4 5 6 7] <ptr_addr_x>
14
15     table := make([][]int,h)
16     for i:= range table {
17         table[i] = raw[i*w:i*w + w]
18     }
19
20     fmt.Println(table,&table[1][0])
21     //prints: [[0 1 2 3] [4 5 6 7]] <ptr_addr_x>
22 }
```

关于多维array和slice已经有了专门申请，但现在看起来这是个低优先级的特性。

14/ 访问不存在的Map Keys

这对于那些希望得到“nil”标示符的开发者而言是个技巧（和其他语言中做的一样）。如果对应的数据类型“零值”是“nil”，那返回的值将会是“nil”，但对于其他的数据类型是不一样的。检测

对应的“零值”可以用于确定map中的记录是否存在，但这并不总是可信（比如，如果在二值的map中“零值”是false，这时你要怎么做）。检测给定map中的记录是否存在的最可信的方法是，通过map的访问操作，检查第二个返回的值。

Bad:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := map[string]string{"one":"a","two":"","three":"c"}
7
8     if v := x["two"]; v == "" { //incorrect
9         fmt.Println("no entry")
10    }
11 }
```

Good:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := map[string]string{"one":"a","two":"","three":"c"}
7
8     if _,ok := x["two"]; !ok {
9         fmt.Println("no entry")
10    }
11 }
```

15/ Strings无法修改

尝试使用索引操作来更新字符串变量中的单个字符将会失败。string是只读的byte slice（和一些额外的属性）。如果你确实需要更新一个字符串，那么使用byte slice，并在需要时把它转

换为string类型。

Fails:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := "text"
7     x[0] = 'T'
8
9     fmt.Println(x)
10 }
```

Compile Error:

```
/tmp/sandbox305565531/main.go:7: cannot assign to x[0]
```

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := "text"
7     xbytes := []byte(x)
8     xbytes[0] = 'T'
9
10    fmt.Println(string(xbytes)) //prints Text
11 }
```

需要注意的是：这并不是在文字string中更新字符的正确方式，因为给定的字符可能会存储在多个byte中。如果你确实需要更新一个文字string，先把它转换为一个rune slice。即使使用rune slice，单个字符也可能会占据多个rune，比如当你的字符有特定的重音符号时就是这种情况。这种复杂又模糊的“字符”本质是Go字符串使用byte序列表示的原因。

16/ String和Byte Slice之间的转换

当你把一个字符串转换为一个 `byte slice`（或者反之）时，你就得到了一个原始数据的完整拷贝。这和其他语言中cast操作不同，也和新的 `slice` 变量指向原始 `byte slice` 使用的相同数组时的重新slice操作不同。

Go在 `[]byte` 到 `string` 和 `string` 到 `[]byte` 的转换中确实使用了一些优化来避免额外的分配（在todo列表中有更多的优化）。

第一个优化避免了当 `[]byte` keys用于在 `map[string]` 集合中查询时的额外分配：`m[string(key)]`。

第二个优化避免了字符串转换为 `[]byte` 后在 `for range` 语句中的额外分配：`for i,v := range []byte(str) {...}`。

17/ String和索引操作

字符串上的索引操作返回一个byte值，而不是一个字符（和其他语言中的做法一样）。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     x := "text"
7     fmt.Println(x[0]) //print 116
8     fmt.Printf("%T",x[0]) //prints uint8
9 }
```

如果你需要访问特定的字符串“字符”（unicode编码的points/runes），使用for range。官方的“unicode/utf8”包和实验中的utf8string包（golang.org/x/exp/utf8string）也可以用。utf8string包中包含了一个很方便的At()方法。把字符串转换为rune的切片也是一个选项。

18/ 字符串不总是UTF8文本

字符串的值不需要是UTF8的文本。它们可以包含任意的字节。只有在string literal使用时，字符串才会是UTF8。即使之后它们可以使用转义序列来包含其他的数据。

为了知道字符串是否是UTF8，你可以使用“unicode/utf8”包中的ValidString()函数。

```
1 package main
2
3 import (
4     "fmt"
5     "unicode/utf8"
6 )
7
8 func main() {
9     data1 := "ABC"
10    fmt.Println(utf8.ValidString(data1)) //prints: true
11
12    data2 := "A\xfeC"
13    fmt.Println(utf8.ValidString(data2)) //prints: false
14 }
```

19/ 字符串的长度

让我们假设你是Python开发者，你有下面这段代码：

```
1 data = u'♥'
2 print(len(data)) #prints: 1
```

当把它转换为Go代码时，你可能会大吃一惊。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     data := "♥"
7     fmt.Println(len(data)) //prints: 3
8 }
```

内建的 `len()` 函数返回byte的数量，而不是像Python中计算好的unicode字符串中字符的数量。

要在Go中得到相同的结果，可以使用“unicode/utf8”包中的 `RuneCountInString()` 函数。

```
1 package main
2
3 import (
4     "fmt"
5     "unicode/utf8"
6 )
7
8 func main() {
9     data := "♥"
10    fmt.Println(utf8.RuneCountInString(data)) //prints: 1
11 }
```

理论上说 `RuneCountInString()` 函数并不返回字符的数量，因为单个字符可能占用多个 rune。

```
1 package main
2
3 import (
4     "fmt"
5     "unicode/utf8"
6 )
7
8 func main() {
9     data := "é"
10    fmt.Println(len(data)) //prints: 3
11    fmt.Println(utf8.RuneCountInString(data)) //prints: 2
12 }
```

20 / 在多行的Slice、Array和Map语句中遗漏逗号

Fails:

```
1 package main
2
3 func main() {
4     x := []int{
5         1,
6         2 //error
7     }
8     _ = x
9 }
```

Compile Errors:

```
/tmp/sandbox367520156/main.go:6: syntax error: need trailing comma before newline in
composite literal /tmp/sandbox367520156/main.go:8: non-declaration statement outside
function body /tmp/sandbox367520156/main.go:9: syntax error: unexpected }
```

Works:

```
1 package main
2
3 func main() {
4     x := []int{
5         1,
6         2,
7     }
8     x = x
9
10    y := []int{3,4,} //no error
11    y = y
12 }
```

当你把声明折叠到单行时，如果你没加末尾的逗号，你将不会得到编译错误。

21/ log.Fatal和log.Panic不仅仅是Log

Logging库一般提供不同的log等级。与这些logging库不同，Go中log包在你调用它的 `Fatal*` `()` 和 `Panic*()` 函数时，可以做的不仅仅是log。当你的应用调用这些函数时，Go也将会终止

应用 :-)

```
1 package main
2
3 import "log"
4
5 func main() {
6     log.Fatalln("Fatal Level: log entry") //app exits here
7     log.Println("Normal Level: log entry")
8 }
```

22 / 内建的数据结构操作不是同步的

即使Go本身有很多特性来支持并发，并发安全的数据集合并不是其中之一 :-)确保数据集合以原子的方式更新是你的职责。Goroutines和channels是实现这些原子操作的推荐方式，但你也可以使用“sync”包，如果它对你的应用有意义的话。

23 / String在“range”语句中的迭代值

索引值（“range”操作返回的第一个值）是返回的第二个值的当前“字符”（unicode编码的point/rune）的第一个byte的索引。它不是当前“字符”的索引，这与其他语言不同。注意真实的字符可能会由多个rune表示。如果你需要处理字符，确保你使用了“norm”包（golang.org/x/text/unicode/norm）。

string变量的 `for range` 语句将会尝试把数据翻译为UTF8文本。对于它无法理解的任何byte序列，它将返回0xfffd runes（即unicode替换字符），而不是真实的数据。如果你任意（非UTF8文本）的数据保存在string变量中，确保把它们转换为byte slice，以得到所有保存的数据。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     data := "A\xfe\x02\xff\x04"
```

```
7     for _,v := range data {
8         fmt.Printf("%#x ",v)
9     }
10    //prints: 0x41 0xfffd 0x2 0xfffd 0x4 (not ok)
11
12    fmt.Println()
13    for _,v := range []byte(data) {
14        fmt.Printf("%#x ",v)
15    }
16    //prints: 0x41 0xfe 0x2 0xff 0x4 (good)
17 }
```

24/ 对Map使用“for range”语句迭代

如果你希望以某个顺序（比如，按key值排序）的方式得到元素，就需要这个技巧。每次的map迭代将会生成不同的结果。Go的runtime有心尝试随机化迭代顺序，但并不总会成功，这样你可能得到一些相同的map迭代结果。所以如果连续看到5个相同的迭代结果，不要惊讶。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     m := map[string]int{"one":1,"two":2,"three":3,"four":4}
7     for k,v := range m {
8         fmt.Println(k,v)
9     }
10 }
```

而且如果你使用Go的游乐场 (<https://play.golang.org/>)，你将总会得到同样的结果，因为除非你修改代码，否则它不会重新编译代码。

25/ "switch"声明中的失效行为

在“switch”声明语句中的“case”语句块在默认情况下会break。这和其他语言中的进入下一个“next”代码块的默认行为不同。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     isSpace := func(ch byte) bool {
7         switch(ch) {
8             case ' ': //error
9             case '\t':
10                 return true
11         }
12         return false
13     }
14
15     fmt.Println(isSpace('\t')) //prints true (ok)
16     fmt.Println(isSpace(' ')) //prints false (not ok)
17 }
```

你可以通过在每个“case”块的结尾使用“fallthrough”，来强制“case”代码块进入。你也可以重写switch语句，来使用“case”块中的表达式列表。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     isSpace := func(ch byte) bool {
7         switch(ch) {
8             case ' ', '\t':
9                 return true
10        }
11        return false
12    }
13
14    fmt.Println(isSpace('\t')) //prints true (ok)
15    fmt.Println(isSpace(' ')) //prints true (ok)
16 }
```


26 / 自增和自减

许多语言都有自增和自减操作。不像其他语言，Go不支持前置版本的操作。你也无法在表达式中使用这两个操作符。

Fails:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     data := []int{1,2,3}
7     i := 0
8     ++i //error
9     fmt.Println(data[i++]) //error
10 }
```

Compile Errors:

```
/tmp/sandbox101231828/main.go:8: syntax error: unexpected ++
```

```
/tmp/sandbox101231828/main.go:9: syntax error: unexpected ++, expecting :
```

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     data := []int{1,2,3}
7     i := 0
8     i++
9     fmt.Println(data[i])
10 }
```

27 / 按位NOT操作

许多语言使用 `~` 作为一元的NOT操作符（即按位补足），但Go为了这个重用了XOR操作符（`^`）。

Fails:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println(~2) //error
7 }
```

Compile Error:

```
/tmp/sandbox965529189/main.go:6: the bitwise complement operator is ^
```

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var d uint8 = 2
7     fmt.Printf("%08b\n", ^d)
8 }
```

Go依旧使用 `^` 作为XOR的操作符，这可能会让一些人迷惑。

如果你愿意，你可以使用一个二元的XOR操作（如，`0x02 XOR 0xff`）来表示一个一元的NOT操作（如，NOT `0x02`）。这可以解释为什么 `^` 被重用来表示一元的NOT操作。

Go也有特殊的‘AND NOT’按位操作（`&^`），这也让NOT操作更加的让人迷惑。这看起来需要特殊的特性/hack来支持 `A AND (NOT B)`，而无需括号。

```
1 package main
2
```

```

3  import "fmt"
4
5  func main() {
6      var a uint8 = 0x82
7      var b uint8 = 0x02
8      fmt.Printf("%08b [A]\n",a)
9      fmt.Printf("%08b [B]\n",b)
10
11     fmt.Printf("%08b (NOT B)\n",^b)
12     fmt.Printf("%08b ^ %08b = %08b [B XOR 0xff]\n",b,0xff,b ^
13
14     fmt.Printf("%08b ^ %08b = %08b [A XOR B]\n",a,b,a ^ b)
15     fmt.Printf("%08b & %08b = %08b [A AND B]\n",a,b,a & b)
16     fmt.Printf("%08b &^%08b = %08b [A 'AND NOT' B]\n",a,b,a &^
17     fmt.Printf("%08b&(^%08b)= %08b [A AND (NOT B)]\n",a,b,a &
18 }

```

28 / 操作优先级的差异

除了“bit clear”操作（`&^`），Go也一个与许多其他语言共享的标准操作符的集合。尽管操作优先级并不总是一样。

```

1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Printf("0x2 & 0x2 + 0x4 -> %#x\n",0x2 & 0x2 + 0x4)
7      //prints: 0x2 & 0x2 + 0x4 -> 0x6
8      //Go:      (0x2 & 0x2) + 0x4
9      //C++:      0x2 & (0x2 + 0x4) -> 0x2
10
11     fmt.Printf("0x2 + 0x2 << 0x1 -> %#x\n",0x2 + 0x2 << 0x1)
12     //prints: 0x2 + 0x2 << 0x1 -> 0x6
13     //Go:      0x2 + (0x2 << 0x1)
14     //C++:      (0x2 + 0x2) << 0x1 -> 0x8
15
16     fmt.Printf("0xf | 0x2 ^ 0x2 -> %#x\n",0xf | 0x2 ^ 0x2)

```

```
17 //prints: 0xf | 0x2 ^ 0x2 -> 0xd
18 //Go:      (0xf | 0x2) ^ 0x2
19 //C++:      0xf | (0x2 ^ 0x2) -> 0xf
20 }
```

29 / 未导出的结构体不会被编码

以小写字母开头的结构体将不会被（json、xml、gob等）编码，因此当你编码这些未导出的结构体时，你将会得到零值。

Fails:

```
1 package main
2
3 import (
4     "fmt"
5     "encoding/json"
6 )
7
8 type MyData struct {
9     One int
10    two string
11 }
12
13 func main() {
14     in := MyData{1,"two"}
15     fmt.Printf("%#v\n",in) //prints main.MyData{One:1, two:"tv
16
17     encoded,_ := json.Marshal(in)
18     fmt.Println(string(encoded)) //prints {"One":1}
19
20
21     var out MyData
22     json.Unmarshal(encoded,&out)
23
24     fmt.Printf("%#v\n",out) //prints main.MyData{One:1, two:"'
25 }
```

30/ 有活动的Goroutines下的应用退出

应用将不会等待所有的goroutines完成。这对于初学者而言是个很常见的错误。每个人都是以某个程度开始，因此如果犯了初学者的错误也没神马好丢脸的 :-)

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     workerCount := 2
10
11     for i := 0; i < workerCount; i++ {
12         go doit(i)
13     }
14     time.Sleep(1 * time.Second)
15     fmt.Println("all done!")
16 }
17
18 func doit(workerId int) {
19     fmt.Printf("[%v] is running\n", workerId)
20     time.Sleep(3 * time.Second)
21     fmt.Printf("[%v] is done\n", workerId)
22 }
```

你将会看到：

```
1 [0] is running
2 [1] is running
3 all done!
```

一个最常见的解决方法是使用“WaitGroup”变量。它将会让主goroutine等待所有的worker goroutine完成。如果你的应用有长时运行的消息处理循环的worker，你也将需要一个方法向这些goroutine发送信号，让它们退出。你可以给各个worker发送一个“kill”消息。另一个选项是关闭一个所有worker都接收的channel。这是一次向所有goroutine发送信号的简单方式。

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func main() {
9     var wg sync.WaitGroup
10    done := make(chan struct{})
11    workerCount := 2
12
13    for i := 0; i < workerCount; i++ {
14        wg.Add(1)
15        go doit(i,done,wg)
16    }
17
18    close(done)
19    wg.Wait()
20    fmt.Println("all done!")
21 }
22
23 func doit(workerId int,done <-chan struct{},wg sync.WaitGroup) {
24     fmt.Printf("[%v] is running\n",workerId)
25     defer wg.Done()
26     <- done
27     fmt.Printf("[%v] is done\n",workerId)
28 }
```

如果你运行这个应用，你将会看到：

```
1 [0] is running
2 [0] is done
3 [1] is running
4 [1] is done
```

看起来所有的worker在主goroutine退出前都完成了。棒！然而，你也将会看到这个：

```
1 fatal error: all goroutines are asleep - deadlock!
```

这可不太好 :-) 发送了神马？为什么会出现死锁？worker退出了，它们也执行了 `wg.Done()`。应用应该没问题啊。

死锁发生是因为各个worker都得到了原始的“WaitGroup”变量的一个拷贝。当worker执行 `wg.Done()` 时，并没有在主goroutine上的“WaitGroup”变量上生效。

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func main() {
9     var wg sync.WaitGroup
10    done := make(chan struct{})
11    wq := make(chan interface{})
12    workerCount := 2
13
14    for i := 0; i < workerCount; i++ {
15        wg.Add(1)
16        go doit(i,wq,done,&wg)
17    }
18
19    for i := 0; i < workerCount; i++ {
20        wq <- i
21    }
22
23    close(done)
24    wg.Wait()
25    fmt.Println("all done!")
26 }
27
28 func doit(workerId int, wq <-chan interface{},done <-chan struct{
29     fmt.Printf("[%v] is running\n",workerId)
30     defer wg.Done()
31     for {
32         select {
33             case m := <- wq:
```

```
34         fmt.Printf("[%v] m => %v\n",workerId,m)
35     case <- done:
36         fmt.Printf("[%v] is done\n",workerId)
37     return
38 }
39 }
40 }
```

现在它会如预期般工作 :-)

31/ 向无缓存的Channel发送消息，只要目标接收者准备好就会立即返回

发送者将不会被阻塞，除非消息正在被接收者处理。根据你运行代码的机器的不同，接收者的goroutine可能会或者不会有足够的时间，在发送者继续执行前处理消息。

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      ch := make(chan string)
7
8      go func() {
9          for m := range ch {
10             fmt.Println("processed:",m)
11         }
12     }()
13
14     ch <- "cmd.1"
15     ch <- "cmd.2" //won't be processed
16 }
```

32/ 向已关闭的Channel发送会引起Panic

从一个关闭的channel接收是安全的。在接收状态下的`ok`的返回值将被设置为`false`，这意味着没有数据被接收。如果你从一个有缓存的channel接收，你将会首先得到缓存的数据，一旦它为空，返回的`ok`值将变为`false`。

向关闭的channel中发送数据会引起panic。这个行为有文档说明，但对于新的Go开发者的直觉不同，他们可能希望发送行为与接收行为很像。

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     ch := make(chan int)
10    for i := 0; i < 3; i++ {
11        go func(idx int) {
12            ch <- (idx + 1) * 2
13        }(i)
14    }
15
16    //get the first result
17    fmt.Println(<-ch)
18    close(ch) //not ok (you still have other senders)
19    //do other work
20    time.Sleep(2 * time.Second)
21 }
```

根据不同的应用，修复方法也将不同。可能是很小的代码修改，也可能需要修改应用的设计。无论是哪种方法，你都需要确保你的应用不会向关闭的channel中发送数据。

上面那个有bug的例子可以通过使用一个特殊的废弃的channel来向剩余的worker发送不再需要它们的结果的信号来修复。

```
1 package main
2
3 import (
```

```
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     ch := make(chan int)
10    done := make(chan struct{})
11    for i := 0; i < 3; i++ {
12        go func(idx int) {
13            select {
14                case ch <- (idx + 1) * 2: fmt.Println(idx, "sent re
15                case <- done: fmt.Println(idx, "exiting")
16            }
17        }(i)
18    }
19
20    //get first result
21    fmt.Println("result:", <-ch)
22    close(done)
23    //do other work
24    time.Sleep(3 * time.Second)
25 }
```

33/ 使用"nil" Channels

在一个 `nil` 的channel上发送和接收操作会被永久阻塞。这个行为有详细的文档解释，但它对于新的Go开发者而言是个惊喜。

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     var ch chan int
10    for i := 0; i < 3; i++ {
```

```
11         go func(idx int) {
12             ch <- (idx + 1) * 2
13         }(i)
14     }
15
16     //get first result
17     fmt.Println("result:", <-ch)
18     //do other work
19     time.Sleep(2 * time.Second)
20 }
```

如果运行代码你将会看到一个runtime错误:

```
1 fatal error: all goroutines are asleep - deadlock!
```

这个行为可以在 `select` 声明中用于动态开启和关闭 `case` 代码块的方法。

```
1 package main
2
3 import "fmt"
4 import "time"
5
6 func main() {
7     inch := make(chan int)
8     outch := make(chan int)
9
10    go func() {
11        var in <- chan int = inch
12        var out chan <- int
13        var val int
14        for {
15            select {
16                case out <- val:
17                    out = nil
18                    in = inch
19                case val = <- in:
20                    out = outch
21                    in = nil
22            }
23        }
24    }
```

```
24     }()
25
26     go func() {
27         for r := range outch {
28             fmt.Println("result:",r)
29         }
30     }()
31
32     time.Sleep(0)
33     inch <- 1
34     inch <- 2
35     time.Sleep(3 * time.Second)
36 }
```

34/ 传值方法的接收者无法修改原有的值

方法的接收者就像常规的函数参数。如果声明为值，那么你的函数/方法得到的是接收者参数的拷贝。这意味着对接收者所做的修改将不会影响原有的值，除非接收者是一个map或者slice变量，而你更新了集合中的元素，或者你更新的域的接收者是指针。

```
1  package main
2
3  import "fmt"
4
5  type data struct {
6      num int
7      key *string
8      items map[string]bool
9  }
10
11 func (this *data) pmethod() {
12     this.num = 7
13 }
14
15 func (this data) vmethod() {
16     this.num = 8
17     *this.key = "v.key"
18     this.items["vmethod"] = true
19 }
```

```
19 }
20
21 func main() {
22     key := "key.1"
23     d := data{1,&key,make(map[string]bool)}
24
25     fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
26     //prints num=1 key=key.1 items=map[]
27
28     d.pmethod()
29     fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
30     //prints num=7 key=key.1 items=map[]
31
32     d.vmethod()
33     fmt.Printf("num=%v key=%v items=%v\n",d.num,*d.key,d.items)
34     //prints num=7 key=v.key items=map[vmethod:true]
35 }
```

中级

1/ 关闭HTTP的响应

当你使用标准http库发起请求时，你得到一个http的响应变量。如果你不读取响应主体，你依旧需要关闭它。注意对于空的响应你也一定要这么做。对于新的Go开发者而言，这个很容易就会忘掉。

一些新的Go开发者确实尝试关闭响应主体，但他们在错误的地方做。

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6     "io/ioutil"
7 )
8
```

```
9 func main() {
10     resp, err := http.Get("https://api.ipify.org?format=json")
11     defer resp.Body.Close()//not ok
12     if err != nil {
13         fmt.Println(err)
14         return
15     }
16
17     body, err := ioutil.ReadAll(resp.Body)
18     if err != nil {
19         fmt.Println(err)
20         return
21     }
22
23     fmt.Println(string(body))
24 }
```

这段代码对于成功的请求没问题，但如果http的请求失败，`resp`变量可能会是`nil`，这将导致一个`runtime panic`。

最常见的关闭响应主体的方法是在http响应的错误检查后调用`defer`。

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6     "io/ioutil"
7 )
8
9 func main() {
10     resp, err := http.Get("https://api.ipify.org?format=json")
11     if err != nil {
12         fmt.Println(err)
13         return
14     }
15
16     defer resp.Body.Close()//ok, most of the time :-)
17     body, err := ioutil.ReadAll(resp.Body)
18     if err != nil {
```

```
19         fmt.Println(err)
20         return
21     }
22
23     fmt.Println(string(body))
24 }
```

大多数情况下，当你的http响应失败时，`resp` 变量将为 `nil`，而 `err` 变量将是 `non-nil`。然而，当你得到一个重定向的错误时，两个变量都将是 `non-nil`。这意味着你最后依然会内存泄露。

通过在http响应错误处理中添加一个关闭 `non-nil` 响应主体的的调用来修复这个问题。另一个方法是使用一个 `defer` 调用来关闭所有失败和成功的请求的响应主体。

```
1  package main
2
3  import (
4      "fmt"
5      "net/http"
6      "io/ioutil"
7  )
8
9  func main() {
10     resp, err := http.Get("https://api.ipify.org?format=json")
11     if resp != nil {
12         defer resp.Body.Close()
13     }
14
15     if err != nil {
16         fmt.Println(err)
17         return
18     }
19
20     body, err := ioutil.ReadAll(resp.Body)
21     if err != nil {
22         fmt.Println(err)
23         return
24     }
25 }
```

```
26     fmt.Println(string(body))
27 }
```

`resp.Body.Close()`的原始实现也会读取并丢弃剩余的响应主体数据。这确保了http的连接在keepalive http连接行为开启的情况下，可以被另一个请求复用。最新的http客户端的行为是不同的。现在读取并丢弃剩余的响应数据是你的职责。如果你不这么做，http的连接可能会关闭，而无法被重用。这个小技巧应该会写在Go 1.5的文档中。

如果http连接的重用对你的应用很重要，你可能需要在响应处理逻辑的后面添加像下面的代码：

```
1  _, err = io.Copy(ioutil.Discard, resp.Body)
```

如果你不立即读取整个响应将是必要的，这可能在处理json API响应时会发生：

```
1  json.NewDecoder(resp.Body).Decode(&data)
```

2/ 关闭HTTP的连接

一些HTTP服务器保持会保持一段时间的网络连接（根据HTTP 1.1的说明和服务端端的“keep-alive”配置）。默认情况下，标准http库只在目标HTTP服务器要求关闭时才会关闭网络连接。这意味着你的应用在某些条件下消耗完sockets/file的描述符。

你可以通过设置请求变量中的`Close`域的值为`true`，来让http库在请求完成时关闭连接。

另一个选项是添加一个`Connection`的请求头，并设置为`close`。目标HTTP服务器应该也会响应一个`Connection: close`的头。当http库看到这个响应头时，它也将关闭连接。

```
1  package main
2
3  import (
4      "fmt"
5      "net/http"
6      "io/ioutil"
7  )
8
```



```
9 func main() {
10     req, err := http.NewRequest("GET", "http://golang.org", nil)
11     if err != nil {
12         fmt.Println(err)
13         return
14     }
15
16     req.Close = true
17     //or do this:
18     //req.Header.Add("Connection", "close")
19
20     resp, err := http.DefaultClient.Do(req)
21     if resp != nil {
22         defer resp.Body.Close()
23     }
24
25     if err != nil {
26         fmt.Println(err)
27         return
28     }
29
30     body, err := ioutil.ReadAll(resp.Body)
31     if err != nil {
32         fmt.Println(err)
33         return
34     }
35
36     fmt.Println(len(string(body)))
37 }
```

你也可以取消http的全局连接复用。你将需要为此创建一个自定义的http传输配置。

```
1 package main
2
3 import (
4     "fmt"
5     "net/http"
6     "io/ioutil"
7 )
8
```

```
9  func main() {
10      tr := &http.Transport{DisableKeepAlives: true}
11      client := &http.Client{Transport: tr}
12
13      resp, err := client.Get("http://golang.org")
14      if resp != nil {
15          defer resp.Body.Close()
16      }
17
18      if err != nil {
19          fmt.Println(err)
20          return
21      }
22
23      fmt.Println(resp.StatusCode)
24
25      body, err := ioutil.ReadAll(resp.Body)
26      if err != nil {
27          fmt.Println(err)
28          return
29      }
30
31      fmt.Println(len(string(body)))
32  }
```

如果你向同一个HTTP服务器发送大量的请求，那么把保持网络连接的打开是没问题的。然而，如果你的应用在短时间内向大量不同的HTTP服务器发送一两个请求，那么在引用收到响应后立刻关闭网络连接是一个好主意。增加打开文件的限制数可能也是个好主意。当然，正确的选择源自于应用。

3/ 比较Structs, Arrays, Slices, and Maps

如果结构体中的各个元素都可以用你可以使用等号来比较的话，那就可以使用相号, ==, 来比较结构体变量。

```
1  package main
2
3  import "fmt"
```

```

4
5  type data struct {
6      num int
7      fp float32
8      complex complex64
9      str string
10     char rune
11     yes bool
12     events <-chan string
13     handler interface{}
14     ref *byte
15     raw [10]byte
16 }
17
18 func main() {
19     v1 := data{}
20     v2 := data{}
21     fmt.Println("v1 == v2:", v1 == v2) //prints: v1 == v2: true
22 }

```

如果结构体中的元素无法比较，那使用等号将导致编译错误。注意数组仅在它们的数据元素可比较的情况下才可以比较。

```

1  package main
2
3  import "fmt"
4
5  type data struct {
6      num int //ok
7      checks [10]func() bool //not comparable
8      doit func() bool //not comparable
9      m map[string] string //not comparable
10     bytes []byte //not comparable
11 }
12
13 func main() {
14     v1 := data{}
15     v2 := data{}
16     fmt.Println("v1 == v2:", v1 == v2)
17 }

```

Go确实提供了一些助手函数，用于比较那些无法使用等号比较的变量。

最常用的方法是使用reflect包中的`DeepEqual()`函数。

```
1 package main
2
3 import (
4     "fmt"
5     "reflect"
6 )
7
8 type data struct {
9     num int //ok
10    checks [10]func() bool //not comparable
11    doit func() bool //not comparable
12    m map[string] string //not comparable
13    bytes []byte //not comparable
14 }
15
16 func main() {
17     v1 := data{}
18     v2 := data{}
19     fmt.Println("v1 == v2:", reflect.DeepEqual(v1, v2)) //prints
20
21     m1 := map[string]string{"one": "a", "two": "b"}
22     m2 := map[string]string{"two": "b", "one": "a"}
23     fmt.Println("m1 == m2:", reflect.DeepEqual(m1, m2)) //print
24
25     s1 := []int{1, 2, 3}
26     s2 := []int{1, 2, 3}
27     fmt.Println("s1 == s2:", reflect.DeepEqual(s1, s2)) //print
28 }
```

除了很慢（这个可能会也可能不会影响你的应用），`DeepEqual()`也有其他自身的技巧。

```
1 package main
2
3 import (
4     "fmt"
```

```
5         "reflect"
6     )
7
8     func main() {
9         var b1 []byte = nil
10        b2 := []byte{}
11        fmt.Println("b1 == b2:", reflect.DeepEqual(b1, b2)) //print
12    }
```

`DeepEqual()` 不会认为空的 `slice` 与“nil”的 `slice` 相等。这个行为与你使用 `bytes.Equal()` 函数的行为不同。 `bytes.Equal()` 认为“nil”和空的slice是相等的。

```
1     package main
2
3     import (
4         "fmt"
5         "bytes"
6     )
7
8     func main() {
9         var b1 []byte = nil
10        b2 := []byte{}
11        fmt.Println("b1 == b2:", bytes.Equal(b1, b2)) //prints: b1
12    }
```

`DeepEqual()` 在比较slice时并不总是完美的。

```
1     package main
2
3     import (
4         "fmt"
5         "reflect"
6         "encoding/json"
7     )
8
9     func main() {
10        var str string = "one"
11        var in interface{} = "one"
12        fmt.Println("str == in:", str == in, reflect.DeepEqual(str,
13        //prints: str == in: true true
```

```
14
15     v1 := []string{"one", "two"}
16     v2 := []interface{}{"one", "two"}
17     fmt.Println("v1 == v2:", reflect.DeepEqual(v1, v2))
18     //prints: v1 == v2: false (not ok)
19
20     data := map[string]interface{}{
21         "code": 200,
22         "value": []string{"one", "two"},
23     }
24     encoded, _ := json.Marshal(data)
25     var decoded map[string]interface{}
26     json.Unmarshal(encoded, &decoded)
27     fmt.Println("data == decoded:", reflect.DeepEqual(data, decoded))
28     //prints: data == decoded: false (not ok)
29 }
```

如果你的 `byte slice`（或者字符串）中包含文字数据，而当你要不区分大小写形式的值时（在使用 `==`，`bytes.Equal()`，或者 `bytes.Compare()`），你可能会尝试使用“bytes”和“string”包中的 `ToUpper()` 或者 `ToLower()` 函数。对于英语文本，这么做是没问题的，但对于许多其他的语言来说就不行了。这时应该使用 `strings.EqualFold()` 和 `bytes.EqualFold()`。

如果你的byte slice中包含需要验证用户数据的隐私信息（比如，加密哈希、tokens等），不要使用 `reflect.DeepEqual()`、`bytes.Equal()`，或者 `bytes.Compare()`，因为这些函数将会让你的应用易于被定时攻击。为了避免泄露时间信息，使用'crypto/subtle'包中的函数（即，`subtle.ConstantTimeCompare()`）。

4/ 从Panic中恢复

`recover()` 函数可以用于获取/拦截 `panic`。仅当在一个 `defer` 函数中被完成时，调用 `recover()` 将会完成这个小技巧。

Incorrect:

```
1 package main
2
```

```
3  import "fmt"
4
5  func main() {
6      recover() //doesn't do anything
7      panic("not good")
8      recover() //won't be executed :)
9      fmt.Println("ok")
10 }
```

Works:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      defer func() {
7          fmt.Println("recovered:", recover())
8      }()
9
10     panic("not good")
11 }
```

`recover()` 的调用仅当它在 `defer` 函数中被直接调用时才有效。

Fails:

```
1  package main
2
3  import "fmt"
4
5  func doRecover() {
6      fmt.Println("recovered =>", recover()) //prints: recovered
7  }
8
9  func main() {
10     defer func() {
11         doRecover() //panic is not recovered
12     }()
13 }
```

```
14     panic("not good")
15 }
```

5/ 在Slice, Array, and Map "range"语句中更新引用元素的值

在“range”语句中生成的数据的值是真实集合元素的拷贝。它们不是原有元素的引用。

这意味着更新这些值将不会修改原来的数据。同时也意味着使用这些值的地址将不会得到原有数据的指针。

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      data := []int{1,2,3}
7      for _,v := range data {
8          v *= 10 //original item is not changed
9      }
10
11     fmt.Println("data:",data) //prints data: [1 2 3]
12 }
```

如果你需要更新原有集合中的数据，使用索引操作符来获得数据。

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      data := []int{1,2,3}
7      for i,_ := range data {
8          data[i] *= 10
9      }
10
11     fmt.Println("data:",data) //prints data: [10 20 30]
12 }
```


如果你的集合保存的是指针，那规则会稍有不同。

如果要更新原有记录指向的数据，你依然需要使用索引操作，但你可以使用for range语句中的第二个值来更新存储在目标位置的数据。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     data := []*struct{num int} { {1},{2},{3} }
7
8     for _,v := range data {
9         v.num *= 10
10    }
11
12    fmt.Println(data[0],data[1],data[2]) //prints {10} {20}
13 }
```

6/ 在Slice中"隐藏"数据

当你重新划分一个slice时，新的slice将引用原有slice的数组。如果你忘了这个行为的话，在你的应用分配大量临时的slice用于创建新的slice来引用原有数据的一小部分时，会导致难以预期的内存使用。

```
1 package main
2
3 import "fmt"
4
5 func get() []byte {
6     raw := make([]byte,10000)
7     fmt.Println(len(raw),cap(raw),&raw[0]) //prints: 10000 10000
8     return raw[:3]
9 }
10
11 func main() {
12     data := get()
13     fmt.Println(len(data),cap(data),&data[0]) //prints: 3 10000
```

```
14 }  
}
```

为了避免这个陷阱，你需要从临时的slice中拷贝数据（而不是重新划分slice）。

```
1 package main  
2  
3 import "fmt"  
4  
5 func get() []byte {  
6     raw := make([]byte,10000)  
7     fmt.Println(len(raw),cap(raw),&raw[0]) //prints: 10000 10000 <nil>  
8     res := make([]byte,3)  
9     copy(res,raw[:3])  
10    return res  
11 }  
12  
13 func main() {  
14     data := get()  
15     fmt.Println(len(data),cap(data),&data[0]) //prints: 3 3 <nil>  
16 }
```

7/ Slice的数据“毁坏”

比如说你需要重新一个路径（在slice中保存）。你通过修改第一个文件夹的名字，然后把名字合并来创建新的路劲，来重新划分指向各个文件夹的路径。

```
1 package main  
2  
3 import (  
4     "fmt"  
5     "bytes"  
6 )  
7  
8 func main() {  
9     path := []byte("AAAA/BBBBBBBBBB")  
10    sepIndex := bytes.IndexByte(path,'/')  
11    dir1 := path[:sepIndex]  
12    dir2 := path[sepIndex+1:]
```

```

13     fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAA/
14     fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBB/
15
16     dir1 = append(dir1,"suffix"... )
17     path = bytes.Join([] []byte{dir1,dir2},[]byte{'/'})
18
19     fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAA/
20     fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => uff/
21
22     fmt.Println("new path =>",string(path))
23 }

```

结果与你想象的不一样。与"AAAAsuffix/BBBBBBBBBB"相反，你将会得到"AAAAsuffix/uffixBBBB"。这个情况的发生是因为两个文件夹的slice都潜在的引用了同一个原始的路径slice。这意味着原始路径也被修改了。根据你的应用，这也许会是个问题。

通过分配新的slice并拷贝需要的数据，你可以修复这个问题。另一个选择是使用完整的slice表达式。

```

1  package main
2
3  import (
4      "fmt"
5      "bytes"
6  )
7
8  func main() {
9      path := []byte("AAAA/BBBBBBBBBB")
10     sepIndex := bytes.IndexByte(path,'/')
11     dir1 := path[:sepIndex:sepIndex] //full slice expression
12     dir2 := path[sepIndex+1:]
13     fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAA/
14     fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBB/
15
16     dir1 = append(dir1,"suffix"... )
17     path = bytes.Join([] []byte{dir1,dir2},[]byte{'/'})
18
19     fmt.Println("dir1 =>",string(dir1)) //prints: dir1 => AAA/
20     fmt.Println("dir2 =>",string(dir2)) //prints: dir2 => BBB/

```

```
21     fmt.Println("new path =>",string(path))
22 }
23
```

完整的slice表达式中的额外参数可以控制新的slice的容量。现在在那个slice后添加元素将会触发一个新的buffer分配，而不是覆盖第二个slice中的数据。

8 / 陈旧的(Stale)Slices

多个slice可以引用同一个数据。比如，当你从一个已有的slice创建一个新的slice时，这就会发生。如果你的应用功能需要这种行为，那么你将需要关注下“走味的”slice。

在某些情况下，在一个slice中添加新的数据，在原有数组无法保持更多新的数据时，将导致分配一个新的数组。而现在其他的slice还指向老的数组（和老的数据）。

```
1  import "fmt"
2
3  func main() {
4      s1 := []int{1,2,3}
5      fmt.Println(len(s1),cap(s1),s1) //prints 3 3 [1 2 3]
6
7      s2 := s1[1:]
8      fmt.Println(len(s2),cap(s2),s2) //prints 2 2 [2 3]
9
10     for i := range s2 { s2[i] += 20 }
11
12     //still referencing the same array
13     fmt.Println(s1) //prints [1 22 23]
14     fmt.Println(s2) //prints [22 23]
15
16     s2 = append(s2,4)
17
18     for i := range s2 { s2[i] += 10 }
19
20     //s1 is now "stale"
21     fmt.Println(s1) //prints [1 22 23]
22     fmt.Println(s2) //prints [32 33 14]
23 }
```

9/ 类型声明和方法

当你通过把一个现有（非interface）的类型定义为一个新的类型时，新的类型不会继承现有类型的方法。

Fails:

```
1 package main
2
3 import "sync"
4
5 type myMutex sync.Mutex
6
7 func main() {
8     var mtx myMutex
9     mtx.Lock() //error
10    mtx.Unlock() //error
11 }
```

Compile Errors:

```
/tmp/sandbox106401185/main.go:9: mtx.Lock undefined (type myMutex has no field or
method Lock) /tmp/sandbox106401185/main.go:10: mtx.Unlock undefined (type myMutex
has no field or method Unlock)
```

如果你确实需要原有类型的方法，你可以定义一个新的struct类型，用匿名方式把原有类型嵌入其中。

Works:

```
1 package main
2
3 import "sync"
4
5 type myLocker struct {
6     sync.Mutex
```

```
7     }  
8  
9     func main() {  
10         var lock myLocker  
11         lock.Lock() //ok  
12         lock.Unlock() //ok  
13     }
```

interface类型的声明也会保留它们的方法集合。

Works:

```
1     package main  
2  
3     import "sync"  
4  
5     type myLocker sync.Locker  
6  
7     func main() {  
8         var lock myLocker = new(sync.Mutex)  
9         lock.Lock() //ok  
10        lock.Unlock() //ok  
11    }
```

10 / 从"for switch"和"for select"代码块中跳出

没有标签的“break”声明只能从内部的switch/select代码块中跳出来。如果无法使用“return”声明的话，那就为外部循环定义一个标签是另一个好的选择。

```
1     package main  
2  
3     import "fmt"  
4  
5     func main() {  
6         loop:  
7             for {  
8                 switch {  
9                     case true:  
10                        fmt.Println("breaking out...")  
11                    }12            }13    }
```

```
11         break loop
12     }
13 }
14
15 fmt.Println("out!")
16 }
```

"goto"声明也可以完成这个功能。。。

11 / "for"声明中的迭代变量和闭包

这在Go中是个很常见的技巧。for语句中的迭代变量在每次迭代时被重新使用。这就意味着你在for循环中创建的闭包（即函数字面量）将会引用同一个变量（而在那些goroutine开始执行时就会得到那个变量的值）。

Incorrect:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     data := []string{"one", "two", "three"}
10
11     for _, v := range data {
12         go func() {
13             fmt.Println(v)
14         }()
15     }
16
17     time.Sleep(3 * time.Second)
18     //goroutines print: three, three, three
19 }
```

最简单的解决方法（不需要修改goroutine）是，在for循环代码块内把当前迭代的变量值保存到一个局部变量中。

Works:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     data := []string{"one", "two", "three"}
10
11     for _, v := range data {
12         vcopy := v //
13         go func() {
14             fmt.Println(vcopy)
15         }()
16     }
17
18     time.Sleep(3 * time.Second)
19     //goroutines print: one, two, three
20 }
```

另一个解决方法是把当前的迭代变量作为匿名goroutine的参数。

Works:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     data := []string{"one", "two", "three"}
```



```
10     for _,v := range data {
11         go func(in string) {
12             fmt.Println(in)
13         }(v)
14     }
15
16     time.Sleep(3 * time.Second)
17     //goroutines print: one, two, three
18 }
19
```

下面这个陷阱稍微复杂一些的版本。

Incorrect:

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  type field struct {
9      name string
10 }
11
12 func (p *field) print() {
13     fmt.Println(p.name)
14 }
15
16 func main() {
17     data := []field{ {"one"}, {"two"}, {"three"} }
18
19     for _,v := range data {
20         go v.print()
21     }
22
23     time.Sleep(3 * time.Second)
24     //goroutines print: three, three, three
25 }
```

Works:

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 type field struct {
9     name string
10 }
11
12 func (p *field) print() {
13     fmt.Println(p.name)
14 }
15
16 func main() {
17     data := []field{ {"one"}, {"two"}, {"three"} }
18
19     for _,v := range data {
20         v := v
21         go v.print()
22     }
23
24     time.Sleep(3 * time.Second)
25     //goroutines print: one, two, three
26 }
```

在运行这段代码时你认为会看到什么结果？（原因是什么？）

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 type field struct {
9     name string
```

```
10 }
11
12 func (p *field) print() {
13     fmt.Println(p.name)
14 }
15
16 func main() {
17     data := []*field{ {"one"}, {"two"}, {"three"} }
18
19     for _, v := range data {
20         go v.print()
21     }
22
23     time.Sleep(3 * time.Second)
24 }
```

12/ Defer函数调用参数的求值

被 `defer` 的函数的参数会在 `defer` 声明时求值（而不是在函数实际执行时）。

Arguments for a deferred function call are evaluated when the defer statement is evaluated (not when the function is actually executing).

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var i int = 1
7
8     defer fmt.Println("result =>", func() int { return i * 2 })()
9     i++
10    //prints: result => 2 (not ok if you expected 4)
11 }
```

13/ 被Defer的函数调用执行

被defer的调用会在包含的函数的末尾执行，而不是包含代码块的末尾。对于Go新手而言，一个很常犯的错误就是无法区分被defer的代码执行规则和变量作用规则。如果你有一个长时运行的函数，而函数内有一个for循环试图在每次迭代时都defer资源清理调用，那就会出现问

题。

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6     "path/filepath"
7 )
8
9 func main() {
10     if len(os.Args) != 2 {
11         os.Exit(-1)
12     }
13
14     start, err := os.Stat(os.Args[1])
15     if err != nil || !start.IsDir(){
16         os.Exit(-1)
17     }
18
19     var targets []string
20     filepath.Walk(os.Args[1], func(fpath string, fi os.FileInf
21         if err != nil {
22             return err
23         }
24
25         if !fi.Mode().IsRegular() {
26             return nil
27         }
28
29         targets = append(targets, fpath)
30         return nil
31     })
32
33     for _, target := range targets {
34         f, err := os.Open(target)
```

```
35         if err != nil {
36             fmt.Println("bad target:",target,"error:",err) //f
37             break
38         }
39         defer f.Close() //will not be closed at the end of thi
40         //do something with the file...
41     }
42 }
```

解决这个问题一个方法是把代码块写成一个函数。

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6      "path/filepath"
7  )
8
9  func main() {
10     if len(os.Args) != 2 {
11         os.Exit(-1)
12     }
13
14     start, err := os.Stat(os.Args[1])
15     if err != nil || !start.IsDir(){
16         os.Exit(-1)
17     }
18
19     var targets []string
20     filepath.Walk(os.Args[1], func(fpath string, fi os.FileInf
21         if err != nil {
22             return err
23         }
24
25         if !fi.Mode().IsRegular() {
26             return nil
27         }
28
29         targets = append(targets,fpath)
```

```
30         return nil
31     })
32
33     for _,target := range targets {
34         func() {
35             f, err := os.Open(target)
36             if err != nil {
37                 fmt.Println("bad target:",target,"error:",err)
38                 return
39             }
40             defer f.Close() //ok
41             //do something with the file...
42         }()
43     }
44 }
```

另一个方法是去掉 `defer` 语句 :-)

14/ 失败的类型断言

失败的类型断言返回断言声明中使用的目标类型的“零值”。这在与隐藏变量混合时，会发生未知情况。

Incorrect:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var data interface{} = "great"
7
8     if data, ok := data.(int); ok {
9         fmt.Println("[is an int] value =>",data)
10    } else {
11        fmt.Println("[not an int] value =>",data)
12        //prints: [not an int] value => 0 (not "great")
13    }
14 }
```

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var data interface{} = "great"
7
8     if res, ok := data.(int); ok {
9         fmt.Println("[is an int] value =>", res)
10    } else {
11        fmt.Println("[not an int] value =>", data)
12        //prints: [not an int] value => great (as expected)
13    }
14 }
```

15/ 阻塞的Goroutine和资源泄露

Rob Pike在2012年的Google I/O大会上所做的“Go Concurrency Patterns”的演讲上，说道过几种基础的并发模式。从一组目标中获取第一个结果就是其中之一。

```
1 func First(query string, replicas ...Search) Result {
2     c := make(chan Result)
3     searchReplica := func(i int) { c <- replicas[i](query) }
4     for i := range replicas {
5         go searchReplica(i)
6     }
7     return <-c
8 }
```

这个函数在每次搜索重复时都会起一个goroutine。每个goroutine把它的搜索结果发送到结果的channel中。结果channel的第一个值被返回。

那其他goroutine的结果会怎样呢？还有那些goroutine自身呢？

在 `First()` 函数中的结果channel是没缓存的。这意味着只有第一个goroutine返回。其他的goroutine会困在尝试发送结果的过程中。这意味着，如果你有不止一个的重复时，每个调用将会泄露资源。

为了避免泄露，你需要确保所有的goroutine退出。一个不错的方法是使用一个有足够保存所有缓存结果的channel。

```
1 func First(query string, replicas ...Search) Result {
2     c := make(chan Result, len(replicas))
3     searchReplica := func(i int) { c <- replicas[i](query) }
4     for i := range replicas {
5         go searchReplica(i)
6     }
7     return <-c
8 }
```

另一个不错的解决方法是使用一个有default情况的select语句和一个保存一个缓存结果的channel。default情况保证了即使当结果channel无法收到消息的情况下，goroutine也不会堵塞。

```
1 func First(query string, replicas ...Search) Result {
2     c := make(chan Result, 1)
3     searchReplica := func(i int) {
4         select {
5             case c <- replicas[i](query):
6             default:
7             }
8     }
9     for i := range replicas {
10        go searchReplica(i)
11    }
12    return <-c
13 }
```

你也可以使用特殊的取消channel来终止workers。

```
1 func First(query string, replicas ...Search) Result {
2     c := make(chan Result)
```



```
3     done := make(chan struct{})
4     defer close(done)
5     searchReplica := func(i int) {
6         select {
7             case c <- replicas[i](query):
8             case <- done:
9             }
10    }
11    for i := range replicas {
12        go searchReplica(i)
13    }
14
15    return <-c
16 }
```

为何在演讲中会包含这些bug？Rob Pike仅仅是不想把演示复杂化。这么作是合理的，但对于Go新手而言，可能会直接使用代码，而不去思考它可能有问题。

高级

1/ 使用指针接收方法的值的实例

只要值是可取址的，那在这个值上调用指针接收方法是没问题的。换句话说，在某些情况下，你不需要在有一个接收值的方法版本。

然而并不是所有的变量是可取址的。Map的元素就不是。通过interface引用的变量也不是。

```
1 package main
2
3 import "fmt"
4
5 type data struct {
6     name string
7 }
8
9 func (p *data) print() {
10    fmt.Println("name:", p.name)
```

```
11 }
12
13 type printer interface {
14     print()
15 }
16
17 func main() {
18     d1 := data{"one"}
19     d1.print() //ok
20
21     var in printer = data{"two"} //error
22     in.print()
23
24     m := map[string]data {"x":data{"three"}}
25     m["x"].print() //error
26 }
```

Compile Errors:

```
/tmp/sandbox017696142/main.go:21: cannot use data literal (type data) as type printer in
assignment: data does not implement printer (print method has pointer receiver)
/tmp/sandbox017696142/main.go:25: cannot call pointer method on m["x"]
/tmp/sandbox017696142/main.go:25: cannot take the address of m["x"]
```

2/ 更新Map的值

如果你有一个struct值的map，你无法更新单个的struct值。

Fails:

```
1 package main
2
3 type data struct {
4     name string
5 }
6
7 func main() {
8     m := map[string]data {"x":{"one"}}
```

```
9      m["x"].name = "two" //error
10  }
```

Compile Error:

```
/tmp/sandbox380452744/main.go:9: cannot assign to m["x"].name
```

这个操作无效是因为map元素是无法取址的。

而让Go新手更加困惑的是slice元素是可以取址的。

```
1  package main
2
3  import "fmt"
4
5  type data struct {
6      name string
7  }
8
9  func main() {
10     s := []data one
11     s[0].name = "two" //ok
12     fmt.Println(s)    //prints: [{two}]
13 }
```

注意在不久之前，使用编译器之一（gccgo）是可以更新map的元素值的，但这一行为很快就被修复了 :)它也被认为是Go 1.3的潜在特性。在那时还不是要急需支持的，但依旧在todo list中。

第一个有效的方法是使用一个临时变量。

```
1  package main
2
3  import "fmt"
4
5  type data struct {
6      name string
7  }
8
```

```
9 func main() {
10     m := map[string]data {"x":{"one"}}
11     r := m["x"]
12     r.name = "two"
13     m["x"] = r
14     fmt.Printf("%v",m) //prints: map[x:{two}]
15 }
```

另一个有效的方法是使用指针的map。

```
1 package main
2
3 import "fmt"
4
5 type data struct {
6     name string
7 }
8
9 func main() {
10     m := map[string]*data {"x":{"one"}}
11     m["x"].name = "two" //ok
12     fmt.Println(m["x"]) //prints: &{two}
13 }
```

顺便说下，当你运行下面的代码时会发生什么？

```
1 package main
2
3 type data struct {
4     name string
5 }
6
7 func main() {
8     m := map[string]*data {"x":{"one"}}
9     m["z"].name = "what?" //???
10 }
```

3/ "nil" Interfaces和"nil" Interfaces的值

这在Go中是第二最常见的技巧，因为interface虽然看起来像指针，但并不是指针。interface变量仅在类型和值为“nil”时才为“nil”。

interface的类型和值会根据用于创建对应interface变量的类型和值的变化而变化。当你检查一个interface变量是否等于“nil”时，这就会导致未预期的行为。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var data *byte
7     var in interface{}
8
9     fmt.Println(data,data == nil) //prints: <nil> true
10    fmt.Println(in,in == nil)      //prints: <nil> true
11
12    in = data
13    fmt.Println(in,in == nil)      //prints: <nil> false
14    //'data' is 'nil', but 'in' is not 'nil'
15 }
```

当你的函数返回interface时，小心这个陷阱。

Incorrect:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     doit := func(arg int) interface{} {
7         var result *struct{} = nil
8
9         if(arg > 0) {
10             result = &struct{}{}
11         }
12
13         return result
14     }
```

```
14     }
15
16     if res := doit(-1); res != nil {
17         fmt.Println("good result:",res) //prints: good result:
18         //'res' is not 'nil', but its value is 'nil'
19     }
20 }
```

Works:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     doit := func(arg int) interface{} {
7         var result *struct{} = nil
8
9         if(arg > 0) {
10             result = &struct{}{}
11         } else {
12             ret
```

4/ 栈和堆变量

你并不总是知道变量是分配到栈还是堆上。在C++中，使用new创建的变量总是在堆上。在Go中，即使是使用 `new()` 或者 `make()` 函数来分配，变量的位置还是由编译器决定。编译器根据变量的大小和“泄露分析”的结果来决定其位置。这也意味着在局部变量上返回引用是没问题的，而这在C或者C++这样的语言中是不行的。

如果你想知道变量分配的位置，在“go build”或“go run”上传入“-m” gc标志（即，go run -gcflags -m app.go）。

5/ GOMAXPROCS, 并发, 和并行

默认情况下，Go仅使用一个执行上下文/OS线程（在当前的版本）。这个数量可以通过设置 `GOMAXPROCS` 来提高。

一个常见的误解是，`GOMAXPROCS` 表示了CPU的数量，Go将使用这个数量来运行goroutine。而 `runtime.GOMAXPROCS()` 函数的文档让人更加的迷茫。`GOMAXPROCS` 变量描述 (<https://golang.org/pkg/runtime/>) 所讨论OS线程的内容比较好。

你可以设置 `GOMAXPROCS` 的数量大于CPU的数量。`GOMAXPROCS` 的最大值是256。

```
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9     fmt.Println(runtime.GOMAXPROCS(-1)) //prints: 1
10    fmt.Println(runtime.NumCPU())        //prints: 1 (on play.g
11    runtime.GOMAXPROCS(20)
12    fmt.Println(runtime.GOMAXPROCS(-1)) //prints: 20
13    runtime.GOMAXPROCS(300)
14    fmt.Println(runtime.GOMAXPROCS(-1)) //prints: 256
15 }
```

6/ 读写操作的重排顺序

Go可能会对某些操作进行重新排序，但它能保证在一个goroutine内的所有行为顺序是不变的。然而，它并不保证多goroutine的执行顺序。

```
1 package main
2
3 import (
4     "runtime"
5     "time"
6 )
```

```
7  var _ = runtime.GOMAXPROCS(3)
8
9  var a, b int
10
11 func u1() {
12     a = 1
13     b = 2
14 }
15
16 func u2() {
17     a = 3
18     b = 4
19 }
20
21 func p() {
22     println(a)
23     println(b)
24 }
25
26 func main() {
27     go u1()
28     go u2()
29     go p()
30     time.Sleep(1 * time.Second)
31 }
32
```

如果你多运行几次上面的代码，你可能会发现a和b变量有多个不同的组合：

```
1  1
2  2
3
4  3
5  4
6
7  0
8  2
9
10 0
11 0
```



```
12 | 1
13 | 4
14 |
```

a和**b**最有趣的组合式是"02"。这表明**b**在**a**之前更新了。

如果你需要在多goroutine内放置读写顺序的变化，你将需要使用channel，或者使用"sync"包构建合适的结构体。

7/ 优先调度

有可能会出现这种情况，一个无耻的goroutine阻止其他goroutine运行。当你有一个不让调度器运行的for循环时，这就会发生。

```
1 | package main
2 |
3 | import "fmt"
4 |
5 | func main() {
6 |     done := false
7 |
8 |     go func(){
9 |         done = true
10 |     }()
11 |
12 |     for !done {
13 |     }
14 |     fmt.Println("done!")
15 | }
```

for循环并不需要是空的。只要它包含了不会触发调度执行的代码，就会发生这种问题。

调度器会在GC、“go”声明、阻塞channel操作、阻塞系统调用和lock操作后运行。它也会在非内联函数调用后执行。

```
1 | package main
2 |
3 | import "fmt"
```

```
4
5 func main() {
6     done := false
7
8     go func(){
9         done = true
10    }()
11
12    for !done {
13        fmt.Println("not done!") //not inlined
14    }
15    fmt.Println("done!")
16 }
```

要想知道你在for循环中调用的函数是否是内联的，你可以在“go build”或“go run”时传入“-m” gc标志（如， `go build -gcflags -m`）。

另一个选择是显式的唤起调度器。你可以使用“runtime”包中的 `Gosched()` 函数。

```
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9     done := false
10
11    go func(){
12        done = true
13    }()
14
15    for !done {
16        runtime.Gosched()
17    }
18    fmt.Println("done!")
19 }
```

如果你看到了这里，并想留下评论或者想法，你可以在这个[Reddit讨论](https://www.reddit.com/r/golang/comments/4jz8qj/gotchas-and-common-mistakes-in-go-golang/)里随意留言。

评论

NEWER

如何编写Go代码

OLDER

使用LinkedHashMap实现LRU缓存

网友跟贴

9人参与

抵制低俗，文明上网，登录发帖



160361473 | 退出

发表跟贴

最新



有态度网友06GrGB [鸟窝四川省网友]

2017-03-17 14:25:16

有态度网友06GrFu [鸟窝北京市网友]

1

关于“优先调度”，在我的测试环境里，已经不会出现楼主所呈现的问题了。
done := false
go func(){
done = true
}()
for !done {
}
fmt.Println(done) //此时是true，这就说明了并没有阻止其他goroutine的执行
fmt.Println(done!)

试试加上 runtime.GOMAXPROCS(1)。
As of Go 1.5, the default value of GOMAXPROCS is the number of CPUs (whatever

your operating system considers to be a CPU) visible to the program at startup.

0 0 | 分享 | 回复



有态度网友06GrFu [鸟窝北京市网友]

2016-08-12 10:30:59

楼主总结的哪些坑，还真是不错，谢谢楼主

0 0 | 分享 | 回复



有态度网友06GrFu [鸟窝北京市网友]

2016-08-12 10:30:07

有态度网友06GrFu [鸟窝北京市网友]

1

关于“优先调度”，在我的测试环境里，已经不会出现楼主所呈现的问题了。

```
done := false
```

```
go func(){
```

```
done = true
```

```
}()
```

```
for !done {
```

```
}
```

```
fmt.Println(done) //此时是true，这就说明了并没有阻止其他goroutine的执行
```

```
fmt.Println(done!)
```

我的测试环境是go 1.6

0 0 | 分享 | 回复



有态度网友06GrFp [鸟窝广东省深圳市网友]

2016-08-04 03:45:17

吴冉波 [鸟窝北京市网友]

1

偶然的变量隐藏，遇到更恶心的是这种多返回的时候：

```
err := errors.New(origin)
```

```
if true {
```

```
_, err = doSomethingFailTow() // 这里用:=是编译不过的
```

```
fmt.Println(err)
```

```
}
```

```
if true {
```

```
innervar, err := doSomethingFail() // 坑在此
```

```
fmt.Println(err)
```

```
}
```

```
fmt.Println(err)
```

找bug内伤不止一次

吴冉波 [鸟窝北京市网友]

2

```
func main() {  
    one := 0  
    if true {  
        one, two := 1, 2  
        one, two = two, one  
    }  
    fmt.Println(one)  
}
```

结果是0。真希望是编译器bug，能给修了。。。

if语句中声明了一个新的变量one，屏蔽了外面的one。

0 0 | 分享 | 回复



吴冉波 [鸟窝北京市网友]

2015-09-28 11:40:21

前几条坑，需要的不是记住，而是有个好vim插件。

0 0 | 分享 | 回复

没有更多跟贴了