Go 是一门简单有趣的编程语言，与其他语言一样，在使用时不免会遇到很多坑，不过它们大多不是 Go 本身的设计缺陷。如果你刚从其他语言转到 Go，那这篇文章里的坑多半会踩到。

如果花时间学习官方 doc、wiki、讨论邮件列表、 Rob Pike 的大量文章以及 Go 的源码，会发现这篇文章中的坑是很常见的，新手跳过这些坑，能减少大量调试代码的时间。

# 初级篇

# 1、左大括号 { 不能单独放一行

在其他大多数语言中，{ 的位置你自行决定。Go比较特别，遵守分号注入规则（automatic semicolon injection）：编译器会在每行代码尾部特定分隔符后加;来分隔多条语句，比如会在 ) 后加分号：

# 2、未使用的变量

如果在函数体代码中有未使用的变量，则无法通过编译，不过全局变量声明但不使用是可以的。即使变量声明后为变量赋值，依旧无法通过编译，需在某处使用它：

# 3、未使用的 import

如果你 import一个包，但包中的变量、函数、接口和结构体一个都没有用到的话，将编译失败。可以使用 \_下划线符号作为别名来忽略导入的包，从而避免编译错误，这只会执行 package 的 init()

# 4、简短声明的变量只能在函数内部使用



# 5、使用简短声明来重复声明变量

不能用简短声明方式来单独为一个变量重复声明，:=左侧至少有一个新变量，才允许多变量的重复声明：

*// 错误示例*

func **main**() {

one := 0

one := 1 *// error: no new variables on left side of :=*

}

*// 正确示例*

func **main**() {

one := 0

one, two := 1, 2 *// two 是新变量，允许 one 的重复声明。比如 error 处理经常用同名变量 err*

one, two = two, one *// 交换两个变量值的简写*

}

# 6、不能使用简短声明来设置字段的值

struct 的变量字段不能使用 := 来赋值以使用预定义的变量来避免解决：

*// 错误示例*

type info **struct** {

result **int**

}

func **work**() (**int**, error) {

**return** 3, nil

}

func **main**() {

**var** data info

data.result, err := work() *// error: non-name data.result on left side of :=*

fmt.Printf("info: %+v\n", data)

}

*// 正确示例*

func **main**() {

**var** data info

**var** err error *// err 需要预声明*

data.result, err = work()

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Printf("info: %+v\n", data)

}

# 7、不小心覆盖了变量

对从动态语言转过来的开发者来说，简短声明很好用，这可能会让人误会 := 是一个赋值操作符。如果你在新的代码块中像下边这样误用了 :=，编译不会报错，但是变量不会按你的预期工作：

func **main**() {

x := 1

println(x) *// 1*

{

println(x) *// 1*

x := 2

println(x) *// 2 // 新的 x 变量的作用域只在代码块内部*

}

println(x) *// 1*

}

# 8、显式类型的变量无法使用 nil 来初始化

nil 是 interface、function、pointer、map、slice 和 channel 类型变量的默认初始值。但声明时不指定类型，编译器也无法推断出变量的具体类型。

*// 错误示例*

func **main**() {

**var** x = nil *// error: use of untyped nil*

\_ = x

}

*// 正确示例*

func **main**() {

**var** x **interface**{} = nil

\_ = x

}

# 9.直接使用值为 nil 的 slice、map

允许对值为 nil 的 slice 添加元素，但对值为 nil 的 map添加元素则会造成运行时 panic

*// map 错误示例*

func **main**() {

**var** m map[**string**]**int**

m["one"] = 1 *// error: panic: assignment to entry in nil map*

*// m := make(map[string]int)// map 的正确声明，分配了实际的内存*

}

*// slice 正确示例*

func **main**() {

**var** s []**int**

s = append(s, 1)

}

# 10.map 容量

在创建 map 类型的变量时可以指定容量，但不能像 slice 一样使用 cap() 来检测分配空间的大小：

# 11.string 类型的变量值不能为 nil

对那些喜欢用 nil 初始化字符串的人来说，这就是坑：

*// 错误示例*

func **main**() {

**var** s **string** = nil *// cannot use nil as type string in assignment*

**if** s == nil { *// invalid operation: s == nil (mismatched types string and nil)*

s = "default"

}

}

*// 正确示例*

func **main**() {

**var** s **string** *// 字符串类型的零值是空串 ""*

**if** s == "" {

s = "default"

}

}

# 12.Array 类型的值作为函数参数

在 C/C++ 中，数组（名）是指针。将数组作为参数传进函数时，相当于传递了数组内存地址的引用，在函数内部会改变该数组的值。

在 Go 中，数组是值。作为参数传进函数时，传递的是数组的原始值拷贝，此时在函数内部是无法更新该数组的：

*// 数组使用值拷贝传参*

func **main**() {

x := [3]**int**{1,2,3}

func(arr [3]**int**) {

arr[0] = 7

fmt.Println(arr) *// [7 2 3]*

}(x)

fmt.Println(x) *// [1 2 3] // 并不是你以为的 [7 2 3]*

}

如果想修改参数数组：

## 直接传递指向这个数组的指针类型：

*// 传址会修改原数据*

func **main**() {

x := [3]**int**{1,2,3}

func(arr \*[3]**int**) {

(\*arr)[0] = 7

fmt.Println(arr) *// &[7 2 3]*

}(&x)

fmt.Println(x) *// [7 2 3]*

}

## 直接使用 slice：即使函数内部得到的是 slice 的值拷贝，但依旧会更新 slice 的原始数据（底层 array）

*// 会修改 slice 的底层 array，从而修改 slice*

func **main**() {

x := []**int**{1, 2, 3}

func(arr []**int**) {

arr[0] = 7

fmt.Println(x) *// [7 2 3]*

}(x)

fmt.Println(x) *// [7 2 3]*

}

# 13.range 遍历 slice 和 array 时混淆了返回值

与其他编程语言中的 for-in 、foreach 遍历语句不同，Go 中的 range 在遍历时会生成 2 个值，第一个是元素索引，第二个是元素的值：

*// 错误示例*

func **main**() {

x := []**string**{"a", "b", "c"}

**for** v := range x {

fmt.Println(v) *// 1 2 3*

}

}

*// 正确示例*

func **main**() {

x := []**string**{"a", "b", "c"}

**for** \_, v := range x { *// 使用 \_ 丢弃索引*

fmt.Println(v)

}

}

# 14.slice 和 array 其实是一维数据

看起来 Go 支持多维的 array 和 slice，可以创建数组的数组、切片的切片，但其实并不是。

对依赖动态计算多维数组值的应用来说，就性能和复杂度而言，用 Go 实现的效果并不理想。

可以使用原始的一维数组、“独立“ 的切片、“共享底层数组”的切片来创建动态的多维数组。

1.使用原始的一维数组：要做好索引检查、溢出检测、以及当数组满时再添加值时要重新做内存分配。

2.使用“独立”的切片分两步：

* 创建外部 slice
  + 对每个内部 slice 进行内存分配

注意内部的 slice 相互独立，使得任一内部 slice 增缩都不会影响到其他的 slice

// 使用各自独立的 6 个 slice 来创建 [2][3] 的动态多维数组

func main() {

x := 2

y := 4

table := make([][]int, x)

for i := range table {

table[i] = make([]int, y)

}

}

1.使用“共享底层数组”的切片

* 创建一个存放原始数据的容器 slice
* 创建其他的 slice
* 切割原始 slice 来初始化其他的 slice

func **main**() {

h, w := 2, 4

raw := make([]**int**, h\*w)

**for** i := range raw {

raw[i] = i

}

*// 初始化原始 slice*

fmt.Println(raw, &raw[4]) *// [0 1 2 3 4 5 6 7] 0xc420012120*

table := make([][]**int**, h)

**for** i := range table {

*// 等间距切割原始 slice，创建动态多维数组 table*

*// 0: raw[0\*4: 0\*4 + 4]*

*// 1: raw[1\*4: 1\*4 + 4]*

table[i] = raw[i\*w : i\*w + w]

}

fmt.Println(table, &table[1][0]) *// [[0 1 2 3] [4 5 6 7]] 0xc420012120*

}

# 15.访问 map 中不存在的 key

Go 则会返回元素对应数据类型的零值，比如 nil、’’ 、false 和 0，取值操作总有值返回，故不能通过取出来的值来判断 key 是不是在 map 中。

检查 key 是否存在可以用 map 直接访问，检查返回的第二个参数即可：

*// 错误的 key 检测方式*

func **main**() {

x := map[string]string{"one": "2", "two": "", "three": "3"}

**if** v := x["two"]; v == "" {

fmt.Println("key two is no entry") *// 键 two 存不存在都会返回的空字符串*

}

}

*// 正确示例*

func main() {

x := map[string]string{"one": "2", "two": "", "three": "3"}

**if** \_, ok := x["two"]; !ok {

fmt.Println("key two is no entry")

}

}

# 16.string 类型的值是常量，不可更改

尝试使用索引遍历字符串，来更新字符串中的个别字符，是不允许的。

string 类型的值是只读的二进制 byte slice，如果真要修改字符串中的字符，将 string 转为 []byte 修改后，再转为 string 即可：

*// 修改字符串的错误示例*

func **main**() {

x := "text"

x[0] = "T" *// error: cannot assign to x[0]*

fmt.Println(x)

}

*// 修改示例*

func **main**() {

x := "text"

xBytes := []**byte**(x)

xBytes[0] = 'T' *// 注意此时的 T 是 rune 类型*

x = **string**(xBytes)

fmt.Println(x) *// Text*

}

注意： 上边的示例并不是更新字符串的正确姿势，因为一个 UTF8 编码的字符可能会占多个字节，比如汉字就需要 3~4个字节来存储，此时更新其中的一个字节是错误的。

更新字串的正确姿势：将 string 转为 rune slice（此时 1 个 rune 可能占多个 byte），直接更新 rune 中的字符

**func** **main**() {

x := "text"

xRunes := []rune(x)

xRunes[0] = '我'

x = string(xRunes)

fmt.Println(x) // 我ext

}

# 17.string 与 byte slice 之间的转换

当进行 string 和 byte slice 相互转换时，参与转换的是拷贝的原始值。这种转换的过程，与其他编程语的强制类型转换操作不同，也和新 slice 与旧 slice 共享底层数组不同。

Go 在 string 与 byte slice 相互转换上优化了两点，避免了额外的内存分配：

* 在 map[string] 中查找 key 时，使用了对应的 []byte，避免做 m[string(key)] 的内存分配
* 使用 for range 迭代 string 转换为 []byte 的迭代：for i,v := range []byte(str) {…}

# 18.string 与索引操作符

对字符串用索引访问返回的不是字符，而是一个 byte 值。

这种处理方式和其他语言一样，比如 PHP 中：

**func** **main**() {

x := "ascii"

fmt.Println(x[0]) // 97

fmt.Printf("%T\n", x[0])// uint8

}

# 20.字符串的长度

在 Python 中：

data = u'♥'

print(len(data)) *# 1*

然而在 Go 中：

**func** **main**() {

char := "♥"

fmt.Println(len(char)) // 3

}

Go 的内建函数 len() 返回的是字符串的 byte 数量，而不是像 Python 中那样是计算 Unicode 字符数。

如果要得到字符串的字符数，可使用 “unicode/utf8” 包中的 RuneCountInString(str string) (n int)

**func** **main**() {

char := "♥"

fmt.Println(utf8.RuneCountInString(char)) // 1

}

注意： RuneCountInString 并不总是返回我们看到的字符数，因为有的字符会占用 2 个 rune：

**func** **main**() {

char := "é"

fmt.Println(len(char)) // 3

fmt.Println(utf8.RuneCountInString(char)) // 2

fmt.Println("cafe\u0301") // café // 法文的 cafe，实际上是两个 rune 的组合

}

# 32.向无缓冲的 channel 发送数据，只要 receiver 准备好了就会立刻返回

只有在数据被 receiver 处理时，sender 才会阻塞。因运行环境而异，在 sender 发送完数据后，receiver 的 goroutine 可能没有足够的时间处理下一个数据。如：

func **main**() {

ch := make(chan **string**)

go **func**() {

**for** m := range ch {

fmt.Println("Processed:", m)

time.Sleep(1 \* time.Second) *// 模拟需要长时间运行的操作*

}

}()

ch <- "cmd.1"

ch <- "cmd.2" *// 不会被接收处理*

}

# 中级篇

# 36.关闭 HTTP 的响应体

使用 HTTP 标准库发起请求、获取响应时，即使你不从响应中读取任何数据或响应为空，都需要手动关闭响应体。新手很容易忘记手动关闭，或者写在了错误的位置：

正确姿势：

*// 正确示例*

func **main**() {

resp, err := http.Get("http://www.baidu.com")

*// 关闭 resp.Body 的正确姿势*

**if** resp != nil {

defer resp.Body.Close()

}

checkError(err)

defer resp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)

checkError(err)

fmt.Println(string(body))

}

resp.Body.Close() 早先版本的实现是读取响应体的数据之后丢弃，保证了 keep-alive 的 HTTP 连接能重用处理不止一个请求。但 Go 的最新版本将读取并丢弃数据的任务交给了用户，如果你不处理，HTTP 连接可能会直接关闭而非重用，参考在 Go 1.5 版本文档。

如果程序大量重用 HTTP 长连接，你可能要在处理响应的逻辑代码中加入：

\_, err = io.Copy(ioutil.Discard, resp.Body) *// 手动丢弃读取完毕的数据*

如果你需要完整读取响应，上边的代码是需要写的。比如在解码 API 的 JSON 响应数据：

**json**.NewDecoder(**resp**.Body).Decode(&**data**)

# 37.关闭 HTTP 连接

一些支持 HTTP1.1 或 HTTP1.0 配置了 connection: keep-alive 选项的服务器会保持一段时间的长连接。但标准库 “net/http” 的连接默认只在服务器主动要求关闭时才断开，所以你的程序可能会消耗完 socket 描述符。解决办法有 2 个，请求结束后：

* 直接设置请求变量的 Close 字段值为 true，每次请求结束后就会主动关闭连接。
* 设置 Header 请求头部选项 Connection: close，然后服务器返回的响应头部也会有这个选项，此时 HTTP 标准库会主动断开连接。
* *// 主动关闭连接*
* func **main**() {
* req, err := http.NewRequest("GET", "http://golang.org", nil)
* checkError(err)
* req.Close = true
* *//req.Header.Add("Connection", "close") // 等效的关闭方式*
* resp, err := http.DefaultClient.Do(req)
* **if** resp != nil {
* defer resp.Body.Close()
* }
* checkError(err)
* body, err := ioutil.ReadAll(resp.Body)
* checkError(err)
* fmt.Println(string(body))
* }

你可以创建一个自定义配置的 HTTP transport 客户端，用来取消 HTTP 全局的复用连接：

func **main**() {

tr := http.Transport{DisableKeepAlives: true}

client := http.Client{Transport: &tr}

resp, err := client.Get("https://golang.google.cn/")

**if** resp != nil {

defer resp.Body.Close()

}

checkError(err)

fmt.Println(resp.StatusCode) *// 200*

body, err := ioutil.ReadAll(resp.Body)

checkError(err)

fmt.Println(len(string(body)))

}

根据需求选择使用场景：

* 若你的程序要向同一服务器发大量请求，使用默认的保持长连接。
* 若你的程序要连接大量的服务器，且每台服务器只请求一两次，那收到请求后直接关闭连接。或增加最大文件打开数 fs.file-max 的值。

# 38.将 JSON 中的数字解码为 interface 类型

在 encode/decode JSON 数据时，Go 默认会将数值当做 float64 处理，比如下边的代码会造成 panic：

func **main**() {

**var** data = []**byte**(`{"status": 200}`)

**var** result map[**string**]**interface**{}

**if** err := json.Unmarshal(data, &result); err != nil {

log.Fatalln(err)

}

fmt.Printf("%T\n", result["status"]) *// float64*

**var** status = result["status"].(**int**) *// 类型断言错误*

fmt.Println("Status value: ", status)

}

如果你尝试 decode 的 JSON 字段是整型，你可以：

* **将 int 值转为 float 统一使用**
* **将 decode 后需要的 float 值转为 int 使用**
* *// 将 decode 的值转为 int 使用*
* func **main**() {
* **var** data = []**byte**(`{"status": 200}`)
* **var** result map[**string**]**interface**{}
* **if** err := json.Unmarshal(data, &result); err != nil {
* log.Fatalln(err)
* }
* **var** status = uint64(result["status"].(float64))
* fmt.Println("Status value: ", status)
* }
* **使用 Decoder 类型来 decode JSON 数据，明确表示字段的值类型**
* *// 指定字段类型*
* func **main**() {
* **var** data = []**byte**(`{"status": 200}`)
* **var** result map[**string**]**interface**{}
* **var** decoder = json.NewDecoder(bytes.NewReader(data))
* decoder.UseNumber()
* **if** err := decoder.Decode(&result); err != nil {
* log.Fatalln(err)
* }
* **var** status, \_ = result["status"].(json.Number).Int64()
* fmt.Println("Status value: ", status)
* }
* *// 你可以使用 string 来存储数值数据，在 decode 时再决定按 int 还是 float 使用*
* *// 将数据转为 decode 为 string*
* func **main**() {
* **var** data = []**byte**({"status": 200})
* **var** result map[**string**]**interface**{}
* **var** decoder = json.NewDecoder(bytes.NewReader(data))
* decoder.UseNumber()
* **if** err := decoder.Decode(&result); err != nil {
* log.Fatalln(err)
* }
* **var** status uint64
* err := json.Unmarshal([]**byte**(result["status"].(json.Number).String()), &status);
* checkError(err)
* fmt.Println("Status value: ", status)
* }

# 39.struct、array、slice 和 map 的值比较

可以使用相等运算符 == 来比较结构体变量，前提是两个结构体的成员都是可比较的类型：

# 40.从 panic 中恢复

在一个 defer 延迟执行的函数中调用 recover() ，它便能捕捉 / 中断 panic

*// 错误的 recover 调用示例*

func **main**() {

recover() *// 什么都不会捕捉*

panic("not good") *// 发生 panic，主程序退出*

recover() *// 不会被执行*

println("ok")

}

*// 正确的 recover 调用示例*

func **main**() {

defer **func**() {

fmt.Println("recovered: ", recover())

}()

panic("not good")

}

从上边可以看出，recover() 仅在 defer 执行的函数中调用才会生效。

*// 错误的调用示例*

func **main**() {

defer **func**() {

doRecover()

}()

panic("not good")

}

func **doRecover**() {

fmt.Println("recobered: ", recover())

}

# 41.在 range 迭代 slice、array、map 时通过更新引用来更新元素

在 range 迭代中，得到的值其实是元素的一份值拷贝，更新拷贝并不会更改原来的元素，即是拷贝的地址并不是原有元素的地址：

func **main**() {

data := []**int**{1, 2, 3}

**for** \_, v := range data {

v \*= 10 *// data 中原有元素是不会被修改的*

}

fmt.Println("data: ", data) *// data: [1 2 3]*

}

如果要修改原有元素的值，应该使用索引直接访问：

func **main**() {

data := []**int**{1, 2, 3}

**for** i, v := range data {

data[i] = v \* 10

}

fmt.Println("data: ", data) *// data: [10 20 30]*

}

如果你的集合保存的是指向值的指针，需稍作修改。依旧需要使用索引访问元素，不过可以使用 range 出来的元素直接更新原有值：

func **main**() {

data := []\***struct**{ num **int** }{{1}, {2}, {3},}

**for** \_, v := range data {

v.num \*= 10 *// 直接使用指针更新*

}

fmt.Println(data[0], data[1], data[2]) *// &{10} &{20} &{30}*

}