50 Shades of Go: Traps, Gotchas, and Common Mistakes(Golang 新手可能会踩的 50 个坑)

- Golang 新手可能会踩的 50 个坑

初级篇：1-34

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：

```
// 错误示例
func main()
{
    println("hello world")
}

// 等效于
func main();    // 无函数体
{
    println("hello world")
}

// ./main.go: missing function body
// ./main.go: syntax error: unexpected semicolon or newline before {


// 正确示例
func main() {
    println("hello world")
}
```

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：

```
// 错误示例
var gvar int     // 全局变量，声明不使用也可以

func main() {
    var one int     // error: one declared and not used
    two := 2    // error: two declared and not used
    var three int    // error: three declared and not used
```

```
    three = 3
}


// 正确示例
// 可以直接注释或移除未使用的变量
func main() {
    var one int
    _ = one

    two := 2
    println(two)

    var three int
    one = three

    var four int
    four = four
}
```

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()

```
// 错误示例
import (
    "fmt"    // imported and not used: "fmt"
    "log"    // imported and not used: "log"
    "time"    // imported and not used: "time"
)

func main() {
}


// 正确示例
// 可以使用 goimports 工具来注释或移除未使用到的包
import (
    _ "fmt"
    "log"
    "time"
)

func main() {
    _ = log.Println
    _ = time.Now
}
```

4. 简短声明的变量只能在函数内部使用

```
// 错误示例
myvar := 1    // syntax error: non-declaration statement outside function body
func main() {
}


// 正确示例
var  myvar = 1
func main() {
}
```

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
}
```

这是 Go 开发者常犯的错，而且不易被发现。

可使用 vet 工具来诊断这种变量覆盖，Go 默认不做覆盖检查，添加 -shadow 选项来启用：

```
> go tool vet -shadow main.go
main.go:9: declaration of "x" shadows declaration at main.go:5
```

注意 vet 不会报告全部被覆盖的变量，可以使用 go-nyet 来做进一步的检测：

```
> $GOPATH/bin/go-nyet main.go
main.go:10:3:Shadowing variable `x`
```

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：

```
// 错误示例
func main() {
    m := make(map[string]int, 99)
    println(cap(m))     // error: invalid argument m1 (type map[string]int) for
cap
}
```

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：

1. 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)
    }
}
```

2. 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
}
```

更多关于多维数组的参考

- go-how-is-two-dimensional-arrays-memory-representation
- what-is-a-concise-way-to-create-a-2d-slice-in-go

15. 访问 map 中不存在的 key

和其他编程语言类似，如果访问了 map 中不存在的 key 则希望能返回 nil，比如在 PHP 中：

```
> php -r '$v = ["x"=>1, "y"=>2]; @var_dump($v["z"]);'
NULL
```

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：

```
> php -r '$name="中文"; var_dump($name);'    # "中文" 占用 6 个字节
string(6) "中文"

> php -r '$name="中文"; var_dump($name[0]);' # 把第一个字节当做 Unicode 字符读取，显示 U+FFFD
string(1) "
```

```
> php -r '$name="中文"; var_dump($name[0].$name[1].$name[2]);'
string(3) "中"
```

```
func main() {
    x := "ascii"
    fmt.Println(x[0])        // 97
    fmt.Printf("%T\n", x[0])// uint8
}
```

如果需要使用 for range 迭代访问字符串中的字符（unicode code point / rune），标准库中有 "unicode/utf8" 包来做 UTF8 的相关解码编码。另外 utf8string 也有像 func (s *String) At(i int) rune 等很方便的库函数。

19. 字符串并不都是 UTF8 文本

string 的值不必是 UTF8 文本，可以包含任意的值。只有字符串是文字字面值时才是 UTF8 文本，字串可以通过转义来包含其他数据。

判断字符串是否是 UTF8 文本，可使用 "unicode/utf8" 包中的 ValidString() 函数：

```
func main() {
    str1 := "ABC"
    fmt.Println(utf8.ValidString(str1))    // true

    str2 := "A\xfeC"
    fmt.Println(utf8.ValidString(str2))    // false

    str3 := "A\\xfeC"
    fmt.Println(utf8.ValidString(str3))    // true    // 把转义字符转义成字面值
}
```

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
}
```

参考: normalization

21. 在多行 array、slice、map 语句中缺少 , 号

```
func main() {
    x := []int {
        1,
        2    // syntax error: unexpected newline, expecting comma or }
    }
    y := []int{1,2,}
    z := []int{1,2}
    // ...
}
```

声明语句中 } 折叠到单行后，尾部的 , 不是必需的。

22. log.Fatal 和 log.Panic 不只是 log

log 标准库提供了不同的日志记录等级，与其他语言的日志库不同，Go 的 log 包在调用 Fatal*()、Panic*() 时能做更多日志外的事，如中断程序的执行等：

```
func main() {
    log.Fatal("Fatal level log: log entry")        // 输出信息后，程序终止执行
    log.Println("Nomal level log: log entry")
}
```

23. 对内建数据结构的操作并不是同步的

尽管 Go 本身有大量的特性来支持并发，但并不保证并发的数据安全，用户需自己保证变量等数据以原子操作更新。

goroutine 和 channel 是进行原子操作的好方法，或使用 "sync" 包中的锁。

24. range 迭代 string 得到的值

range 得到的索引是字符值（Unicode point / rune）第一个字节的位置，与其他编程语言不同，这个索引并不直接是字符在字符串中的位置。

注意一个字符可能占多个 rune，比如法文单词 café 中的 é。操作特殊字符可使用norm 包。

for range 迭代会尝试将 string 翻译为 UTF8 文本，对任何无效的码点都直接使用 0XFFFD rune（`&#xfffd;`）UNicode 替代字符来表示。如果 string 中有任何非 UTF8 的数据，应将 string 保存为 byte slice 再进行操作。

```
func main() {
    data := "A\xfe\x02\xff\x04"
    for _, v := range data {
        fmt.Printf("%#x ", v)    // 0x41 0xfffd 0x2 0xfffd 0x4    // 错误
    }

    for _, v := range []byte(data) {
        fmt.Printf("%#x ", v)    // 0x41 0xfe 0x2 0xff 0x4    // 正确
    }
}
```

25. range 迭代 map

如果你希望以特定的顺序（如按 key 排序）来迭代 map，要注意每次迭代都可能产生不一样的结果。

Go 的运行时是有意打乱迭代顺序的，所以你得到的迭代结果可能不一致。但也并不总会打乱，得到连续相同的 5 个迭代结果也是可能的，如：

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

如果你去 Go Playground 重复运行上边的代码，输出是不会变的，只有你更新代码它才会重新编译。重新编译后迭代顺序是被打乱的：

26. switch 中的 fallthrough 语句

switch 语句中的 case 代码块会默认带上 break，但可以使用 fallthrough 来强制执行下一个 case 代码块。

```
func main() {
    isSpace := func(char byte) bool {
        switch char {
        case ' ':    // 空格符会直接 break，返回 false // 和其他语言不一样
        // fallthrough    // 返回 true
        case '\t':
            return true
        }
        return false
    }
    fmt.Println(isSpace('\t'))    // true
    fmt.Println(isSpace(' '))    // false
}
```

不过你可以在 case 代码块末尾使用 fallthrough，强制执行下一个 case 代码块。

也可以改写 case 为多条件判断：

```
func main() {
    isSpace := func(char byte) bool {
        switch char {
        case ' ', '\t':
            return true
        }
        return false
    }
    fmt.Println(isSpace('\t'))    // true
    fmt.Println(isSpace(' '))    // true
}
```

27. 自增和自减运算 很多编程语言都自带前置后置的 ++、-- 运算。但 Go 特立独行，去掉了前置操作，同时 ++、— 只作为运算符而非表达式。

```
// 错误示例
func main() {
    data := []int{1, 2, 3}
    i := 0
    ++i            // syntax error: unexpected ++, expecting }
    fmt.Println(data[i++])    // syntax error: unexpected ++, expecting :
}


// 正确示例
func main() {
    data := []int{1, 2, 3}
    i := 0
    i++
```

```
    fmt.Println(data[i])    // 2
}
```

28. 按位取反

很多编程语言使用 ~ 作为一元按位取反（NOT）操作符，Go 重用 ^ XOR 操作符来按位取反：

```
// 错误的取反操作
func main() {
    fmt.Println(~2)        // bitwise complement operator is ^
}


// 正确示例
func main() {
    var d uint8 = 2
    fmt.Printf("%08b\n", d)        // 00000010
    fmt.Printf("%08b\n", ^d)    // 11111101
}
```

同时 ^ 也是按位异或（XOR）操作符。

一个操作符能重用两次，是因为一元的 NOT 操作 NOT 0x02，与二元的 XOR 操作 0x22 XOR 0xff 是一致的。

Go 也有特殊的操作符 AND NOT &^ 操作符，不同位才取1。

```
func main() {
    var a uint8 = 0x82
    var b uint8 = 0x02
    fmt.Printf("%08b [A]\n", a)
    fmt.Printf("%08b [B]\n", b)

    fmt.Printf("%08b (NOT B)\n", ^b)
    fmt.Printf("%08b ^ %08b = %08b [B XOR 0xff]\n", b, 0xff, b^0xff)

    fmt.Printf("%08b ^ %08b = %08b [A XOR B]\n", a, b, a^b)
    fmt.Printf("%08b & %08b = %08b [A AND B]\n", a, b, a&b)
    fmt.Printf("%08b &^%08b = %08b [A 'AND NOT' B]\n", a, b, a&^b)
    fmt.Printf("%08b&(^%08b)= %08b [A AND (NOT B)]\n", a, b, a&(^b))
}
```

```
10000010 [A]
00000010 [B]
11111101 (NOT B)
00000010 ^ 11111111 = 11111101 [B XOR 0xff]
10000010 ^ 00000010 = 10000000 [A XOR B]
10000010 & 00000010 = 00000010 [A AND B]
```

```
10000010 &^00000010 = 10000000 [A 'AND NOT' B]
10000010&(^00000010)= 10000000 [A AND (NOT B)]
```

29. 运算符的优先级

除了位清除（bit clear）操作符，Go 也有很多和其他语言一样的位操作符，但优先级另当别论。

30. 不导出的 struct 字段无法被 encode

以小写字母开头的字段成员是无法被外部直接访问的，所以 struct 在进行 json、xml、gob 等格式的 encode 操作时，这些私有字段会被忽略，导出时得到零值：

```go
func main() {
    in := MyData{1, "two"}
    fmt.Printf("%#v\n", in)    // main.MyData{One:1, two:"two"}

    encoded, _ := json.Marshal(in)
    fmt.Println(string(encoded))    // {"One":1}    // 私有字段 two 被忽略了

    var out MyData
    json.Unmarshal(encoded, &out)
    fmt.Printf("%#v\n", out)     // main.MyData{One:1, two:""}
}
```

31. 程序退出时还有 goroutine 在执行

程序默认不等所有 goroutine 都执行完才退出，这点需要特别注意：

```go
// 主程序会直接退出
func main() {
    workerCount := 2
    for i := 0; i < workerCount; i++ {
        go doIt(i)
    }
    time.Sleep(1 * time.Second)
    fmt.Println("all done!")
}

func doIt(workerID int) {
    fmt.Printf("[%v] is running\n", workerID)
    time.Sleep(3 * time.Second)        // 模拟 goroutine 正在执行
    fmt.Printf("[%v] is done\n", workerID)
}
```

如下，main() 主程序不等两个 goroutine 执行完就直接退出了：

常用解决办法：使用 "WaitGroup" 变量，它会让主程序等待所有 goroutine 执行完毕再退出。

如果你的 goroutine 要做消息的循环处理等耗时操作，可以向它们发送一条 kill 消息来关闭它们。或直接关闭一个它们都等待接收数据的 channel：

```
// 等待所有 goroutine 执行完毕
// 进入死锁
func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})

    workerCount := 2
    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doIt(i, done, wg)
    }

    close(done)
    wg.Wait()
    fmt.Println("all done!")
}

func doIt(workerID int, done <-chan struct{}, wg sync.WaitGroup) {
    fmt.Printf("[%v] is running\n", workerID)
    defer wg.Done()
    <-done
    fmt.Printf("[%v] is done\n", workerID)
}
```

31. 程序退出时还有 goroutine 在执行

程序默认不等所有 goroutine 都执行完才退出，这点需要特别注意：

```
// 主程序会直接退出
func main() {
    workerCount := 2
    for i := 0; i < workerCount; i++ {
        go doIt(i)
    }
    time.Sleep(1 * time.Second)
    fmt.Println("all done!")
}

func doIt(workerID int) {
    fmt.Printf("[%v] is running\n", workerID)
    time.Sleep(3 * time.Second)        // 模拟 goroutine 正在执行
    fmt.Printf("[%v] is done\n", workerID)
}
```

如下，main() 主程序不等两个 goroutine 执行完就直接退出了：

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

常用解决办法：使用 "WaitGroup" 变量，它会让主程序等待所有 goroutine 执行完毕再退出。

如果你的 goroutine 要做消息的循环处理等耗时操作，可以向它们发送一条 kill 消息来关闭它们。或直接关闭一个它们都等待接收数据的 channel：

```go
// 等待所有 goroutine 执行完毕
// 进入死锁
func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})

    workerCount := 2
    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doIt(i, done, wg)
    }

    close(done)
    wg.Wait()
    fmt.Println("all done!")
}

func doIt(workerID int, done <-chan struct{}, wg sync.WaitGroup) {
    fmt.Printf("[%v] is running\n", workerID)
    defer wg.Done()
    <-done
    fmt.Printf("[%v] is done\n", workerID)
}
```

执行结果：

```
[1] is running
[1] is done
[0] is running
[0] is done
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc000020120?)
        C:/Program Files/Go/src/runtime/sema.go:62 +0x25
sync.(*WaitGroup).Wait(0xc000068060?)
        C:/Program Files/Go/src/sync/waitgroup.go:116 +0x48
main.main()
```

```
        D:/work/godemo/test.go:20 +0xe6
exit status 2
```

看起来好像 goroutine 都执行完了，然而报错：

> fatal error: all goroutines are asleep - deadlock!

为什么会发生死锁？goroutine 在退出前调用了 wg.Done()，程序应该正常退出的。

原因是 goroutine 得到的 "WaitGroup" 变量是 var wg WaitGroup 的一份拷贝值，即 doIt() 传参只传值。所以哪怕在每个 goroutine 中都调用了 wg.Done()， 主程序中的 wg 变量并不会受到影响。

```go
// 等待所有 goroutine 执行完毕
// 使用传址方式为 WaitGroup 变量传参
// 使用 channel 关闭 goroutine

func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})
    ch := make(chan interface{})

    workerCount := 2
    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doIt(i, ch, done, &wg)    // wg 传指针，doIt() 内部会改变 wg 的值
    }

    for i := 0; i < workerCount; i++ {    // 向 ch 中发送数据，关闭 goroutine
        ch <- i
    }

    close(done)
    wg.Wait()
    close(ch)
    fmt.Println("all done!")
}

func doIt(workerID int, ch <-chan interface{}, done <-chan struct{}, wg
*sync.WaitGroup) {
    fmt.Printf("[%v] is running\n", workerID)
    defer wg.Done()
    for {
        select {
        case m := <-ch:
            fmt.Printf("[%v] m => %v\n", workerID, m)
        case <-done:
            fmt.Printf("[%v] is done\n", workerID)
            return
        }
    }
}
```

运行效果：

```
[1] is running
[1] m => 0
[1] m => 1
[1] is done
[0] is running
[0] is done
all done!
```

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
}
```

运行效果：

```
Processed: cmd.1
```

33. 向已关闭的 channel 发送数据会造成 panic

从已关闭的 channel 接收数据是安全的：

接收状态值 ok 是 false 时表明 channel 中已没有数据可以接收了。类似的，从有缓冲的 channel 中接收数据，缓存的数据获取完再没有数据可取时，状态值也是 false

向已关闭的 channel 中发送数据会造成 panic：

```
func main() {
    ch := make(chan int)
    for i := 0; i < 3; i++ {
```

```
        go func(idx int) {
            ch <- idx
        }(i)
    }

    fmt.Println(<-ch)         // 输出第一个发送的值
    close(ch)             // 不能关闭，还有其他的 sender
    time.Sleep(2 * time.Second)     // 模拟做其他的操作
}
```

运行结果：

```
0
panic: send on closed channel

goroutine 7 [running]:
main.main.func1(0x1)
        D:/work/godemo/test.go:17 +0x25
created by main.main in goroutine 1
        D:/work/godemo/test.go:16 +0x45
exit status 2
```

针对上边有 bug 的这个例子，可使用一个废弃 channel done 来告诉剩余的 goroutine 无需再向 ch 发送数据。此时 <- done 的结果是 {}：

```
func main() {
    ch := make(chan int)
    done := make(chan struct{})

    for i := 0; i < 3; i++ {
        go func(idx int) {
            select {
            case ch <- (idx + 1) * 2:
                fmt.Println(idx, "Send result")
            case <-done:
                fmt.Println(idx, "Exiting")
            }
        }(i)
    }

    fmt.Println("Result: ", <-ch)
    close(done)
    time.Sleep(3 * time.Second)
}
```

运行效果：

```
Result:  2
2 Exiting
0 Send result
1 Exiting
```

34. 使用了值为 nil 的 channel

在一个值为 nil 的 channel 上发送和接收数据将永久阻塞：

```
func main() {
    var ch chan int // 未初始化，值为 nil
    for i := 0; i < 3; i++ {
        go func(i int) {
            ch <- i
        }(i)
    }

    fmt.Println("Result: ", <-ch)
    time.Sleep(2 * time.Second)
}
```

runtime 死锁错误：

```
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan receive (nil chan)]
```

利用这个死锁的特性，可以用在 select 中动态的打开和关闭 case 语句块：

```
func main() {
    inCh := make(chan int)
    outCh := make(chan int)

    go func() {
        var in <-chan int = inCh
        var out chan<- int
        var val int

        for {
            select {
            case out <- val:
                println("--------")
                out = nil
                in = inCh
            case val = <-in:
                println("++++++++++")
                out = outCh
```

```
                in = nil
            }
        }
    }()

    go func() {
        for r := range outCh {
            fmt.Println("Result: ", r)
        }
    }()

    time.Sleep(0)
    inCh <- 1
    inCh <- 2
    time.Sleep(3 * time.Second)
}
```

运行效果：

```
++++++++++
--------
++++++++++
Result:  1
Result:  2
--------
```

34. 若函数 receiver 传参是传值方式，则无法修改参数的原有值

方法 receiver 的参数与一般函数的参数类似：如果声明为值，那方法体得到的是一份参数的值拷贝，此时对参数的任何修改都不会对原有值产生影响。

除非 receiver 参数是 map 或 slice 类型的变量，并且是以指针方式更新 map 中的字段、slice 中的元素的，才会更新原有值:

```
type data struct {
    num   int
    key   *string
    items map[string]bool
}

func (this *data) pointerFunc() {
    this.num = 7
}

func (this data) valueFunc() {
    this.num = 8
    *this.key = "valueFunc.key"
    this.items["valueFunc"] = true
}
```

```
func main() {
    key := "key1"

    d := data{1, &key, make(map[string]bool)}
    fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)

    d.pointerFunc()    // 修改 num 的值为 7
    fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)

    d.valueFunc()    // 修改 key 和 items 的值
    fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)
}
```

运行结果：

```
num=1  key=key1  items=map[]
num=7  key=key1  items=map[]
num=7  key=valueFunc.key  items=map[valueFunc:true]
```