Golang高效跳坑

前言

新接手的项目，需要使用 Go 语言开发，在使用的过程中，发现这个语言是"半成品"!

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。

现在分两个部分介绍下常见的问题，关注业务的话可以跳过第一部分。
第一部分(语法篇)是简单说几点关于编译器和静态检查的(你得让你的程序跑起来);
第二部分(业务篇)是关于日常开发的,着重介绍从别的语言改过来不适应的点(个人踩坑了泛型和map)。

语法篇:

1. 未使用的变量,import

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
}
```

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
}
```

2. 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
}
```

3. 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`
```

4. 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)
}
```

5. 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]
}
```

6. 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)    // 0 1 2
    }
}

// 正确示例
func main() {
    x := []string{"a", "b", "c"}
    for _, v := range x {    // 使用 _ 丢弃索引
        fmt.Println(v)
    }
}
```

7. 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")
    }
}
```

8. 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
}
```

9. 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。

10. 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

业务篇

0. 泛型以及工具类

由于项目所用的版本为1.7,所以单独拿出来聊一下泛型.

背景:在业务中遇到操作set的需求,目标是获取两个列表A中有而列表B中无的元素

需求看起来是一个很常规的集合交并集的业务,如果是python直接一行setA-setB就能实现,而java也可以用一些Guava类的工具包实现,但是在go中却不一样:

首先,golang的标准库中没有对set的操作,所以想要操作set要额外方法.
1.使用map来实现set,map中的key为唯一值,这与set的特性一致.
2.在github上已经有了一个成熟的包，名为golang-set，包中提供了线程安全和非线程安全的set函数.
当我使用基础方法1实现int类型的需求后,发现string类型的变量也需要相同的实现,这样就有了很多冗余代码,所以我就想到用引入工具库,当时引用了一个hutool-go的包.导入时发现,包中用到了type T这一1.8的特性.而泛型在golang1.8才引入,也许写工具包的开发也不想写这么多重复代码.

至于泛型是否需要被引入,感兴趣的可以看下作者的回复 Golang作者

1. 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是不一致的。

```
func main() {
    m := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}
    for i := 0; i < 2; i++ {
      for k, v := range m {
        fmt.Println(k, v)
          }
    }
}
```

相同代码执行多次结果不同
result1:

three 3
four 4
one 1
two 2

result2:

one 1
two 2

three 3
four 4

这是因为迭代map时增加的随机的偏移量,所以如果我们希望获得一致的结果需要维护一个顺序的key列表.

```
func main() {

    m := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}

    keys := []string{"one", "two", "three", "four"}
    for i := 0; i < 2; i++ {
        for _, v := range keys {
        fmt.Println(v, m[v])
        }
    }
}
```

2. 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：

```
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:""}
}
```

3. 程序退出时还有 goroutine 在执行

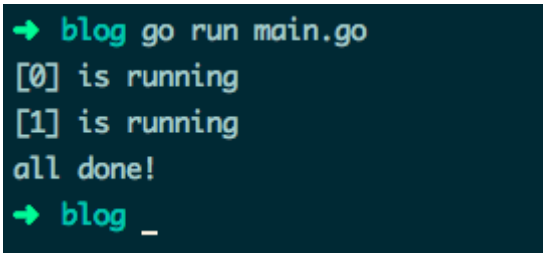
程序默认不等所有 goroutine 都执行完才退出，这点需要特别注意：

```
// 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：



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)
}
```

执行结果：

```
➜ blog go run main.go
[0] is running
[0] is done
[1] is running
[1] is done
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200160ac)
        /usr/local/go/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200160a0)
        /usr/local/go/src/sync/waitgroup.go:131 +0x72
main.main()
        /Users/wuyin/Go/src/blog/main.go:19 +0xe2
exit status 2
➜ blog _
```

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。

```
// 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：

```
➜ blog go run main.go
[1] is running
[1] m => 0
[0] is running
[0] is done
[1] m => 1
[1] is done
all done!
➜ blog _
```

4. 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)
}
```

panic: interface conversion: interface {} is float64, not int

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)
}
```

- 使用 `struct` 类型将你需要的数据映射为数值型

```
// struct 中指定字段类型
func main() {
      var data = []byte(`{"status": 200}`)
      var result struct {
          Status uint64 `json:"status"`
      }

      err := json.NewDecoder(bytes.NewReader(data)).Decode(&result)
      checkError(err)
    fmt.Printf("Result: %+v", result)
}
```

- 可以使用 `struct` 将数值类型映射为 `json.RawMessage` 原生数据类型
  适用于如果 JSON 数据不着急 decode 或 JSON 某个字段的值类型不固定等情况：

```
// 状态名称可能是 int 也可能是 string，指定为 json.RawMessage 类型
func main() {
    records := [][]byte{
        []byte(`{"status":200, "tag":"one"}`),
        []byte(`{"status":"ok", "tag":"two"}`),
    }

    for idx, record := range records {
        var result struct {
            StatusCode uint64
            StatusName string
            Status     json.RawMessage `json:"status"`
            Tag        string          `json:"tag"`
        }

        err := json.NewDecoder(bytes.NewReader(record)).Decode(&result)
        checkError(err)

        var name string
        err = json.Unmarshal(result.Status, &name)
        if err == nil {
            result.StatusName = name
        }

        var code uint64
        err = json.Unmarshal(result.Status, &code)
        if err == nil {
            result.StatusCode = code
        }

        fmt.Printf("[%v] result => %+v\n", idx, result)
    }
}
```

5. 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：

```
type data struct {
    num     int
    fp      float32
    complex complex64
    str     string
    char    rune
    yes     bool
```

```
    events  <-chan string
    handler interface{}
    ref     *byte
    raw     [10]byte
}

func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2: ", v1 == v2)    // true
}
```

如果两个结构体中有任意成员是不可比较的，将会造成编译错误。注意数组成员只有在数组元素可比较时候才可比较。

```
type data struct {
    num    int
    checks [10]func() bool        // 无法比较
    doIt   func() bool        // 无法比较
    m      map[string]string    // 无法比较
    bytes  []byte            // 无法比较
}

func main() {
    v1 := data{}
    v2 := data{}

    fmt.Println("v1 == v2: ", v1 == v2)
}
```

invalid operation: v1 == v2 (struct containing [10]func() bool cannot be compared)

Go 提供了一些库函数来比较那些无法使用 `==` 比较的变量，比如使用 "reflect" 包的 `DeepEqual()` ：

```
// 比较相等运算符无法比较的元素
func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))    // true

    m1 := map[string]string{"one": "a", "two": "b"}
    m2 := map[string]string{"two": "b", "one": "a"}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(m1, m2))    // true
```

```
    s1 := []int{1, 2, 3}
    s2 := []int{1, 2, 3}
       // 注意两个 slice 相等，值和顺序必须一致
    fmt.Println("v1 == v2: ", reflect.DeepEqual(s1, s2))    // true
}
```

这种比较方式可能比较慢，根据你的程序需求来使用。`DeepEqual()` 还有其他用法：

```
func main() {
    var b1 []byte = nil
    b2 := []byte{}
    fmt.Println("b1 == b2: ", reflect.DeepEqual(b1, b2))    // false
}
```

注意：

- `DeepEqual()` 并不总适合于比较 slice

```
func main() {
    var str = "one"
    var in interface{} = "one"
    fmt.Println("str == in: ", reflect.DeepEqual(str, in))    // true

    v1 := []string{"one", "two"}
    v2 := []string{"two", "one"}
    fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))    // false

    data := map[string]interface{}{
        "code":  200,
        "value": []string{"one", "two"},
    }
    encoded, _ := json.Marshal(data)
    var decoded map[string]interface{}
    json.Unmarshal(encoded, &decoded)
    fmt.Println("data == decoded: ", reflect.DeepEqual(data, decoded))    // false
}
```

如果要大小写不敏感来比较 byte 或 string 中的英文文本，可以使用 "bytes" 或 "strings" 包的 `ToUpper()` 和 `ToLower()` 函数。比较其他语言的 byte 或 string，应使用 `bytes.EqualFold()` 和 `strings.EqualFold()`

如果 byte slice 中含有验证用户身份的数据（密文哈希、token 等），不应再使用 `reflect.DeepEqual()`、`bytes.Equal()`、`bytes.Compare()`。这三个函数容易对程序造成 timing

attacks，此时应使用 "crypto/subtle" 包中的 `subtle.ConstantTimeCompare()` 等函数

- `reflect.DeepEqual()` 认为空 slice 与 nil slice 并不相等，但注意 `byte.Equal()` 会认为二者相等：

```
func main() {
    var b1 []byte = nil
    b2 := []byte{}

    // b1 与 b2 长度相等、有相同的字节序
    // nil 与 slice 在字节上是相同的
    fmt.Println("b1 == b2: ", bytes.Equal(b1, b2))    // true
}
```

6. 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}
}
```

7. slice 中隐藏的数据

从 slice 中重新切出新 slice 时，新 slice 会引用原 slice 的底层数组。如果跳了这个坑，程序可能会分配大量的临时 slice 来指向原底层数组的部分数据，将导致难以预料的内存使用。

```
func get() []byte {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    return raw[:3]    // 重新分配容量为 10000 的 slice
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 10000 0xc420080000
}
```

可以通过拷贝临时 slice 的数据，而不是重新切片来解决：

```
func get() (res []byte) {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    res = make([]byte, 3)
    copy(res, raw[:3])
    return
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 3 0xc4200160b8
}
```

8. Slice 中数据的误用

举个简单例子，重写文件路径（存储在 slice 中）

分割路径来指向每个不同级的目录，修改第一个目录名再重组子目录名，创建新路径：

```
// 错误使用 slice 的拼接示例
func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    println(sepIndex)

    dir1 := path[:sepIndex]
    dir2 := path[sepIndex+1:]
    println("dir1: ", string(dir1))        // AAAA
    println("dir2: ", string(dir2))        // BBBBBBBBB

    dir1 = append(dir1, "suffix"...)
       println("current path: ", string(path))    // AAAAsuffixBBBB

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // uffixBBBB

    println("new path: ", string(path))    // AAAAsuffix/uffixBBBB    // 错误结果
}
```

拼接的结果不是正确的 `AAAAsuffix/BBBBBBBBB`，因为 dir1、 dir2 两个 slice 引用的数据都是 `path` 的底层数组，第 13 行修改 `dir1` 同时也修改了 `path`，也导致了 `dir2` 的修改

解决方法：

- 重新分配新的 slice 并拷贝你需要的数据
- 使用完整的 slice 表达式：`input[low:high:max]`，容量便调整为 max - low

```
// 使用 full slice expression
func main() {

    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    dir1 := path[:sepIndex:sepIndex]        // 此时 cap(dir1) 指定为4， 而不是先前的
16
    dir2 := path[sepIndex+1:]
    dir1 = append(dir1, "suffix"...)

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // BBBBBBBBB
    println("new path: ", string(path))    // AAAAsuffix/BBBBBBBBB
}
```

第 6 行中第三个参数是用来控制 dir1 的新容量，再往 dir1 中 append 超额元素时，将分配新的 buffer 来保存。而不是覆盖原来的 path 底层数组

9. 旧 slice

当你从一个已存在的 slice 创建新 slice 时，二者的数据指向相同的底层数组。如果你的程序使用这个特性，那需要注意 "旧"（stale） slice 问题。

某些情况下，向一个 slice 中追加元素而它指向的底层数组容量不足时，将会重新分配一个新数组来存储数据。而其他 slice 还指向原来的旧底层数组。

```
// 超过容量将重新分配数组来拷贝值、重新存储
func main() {
    s1 := []int{1, 2, 3}
    fmt.Println(len(s1), cap(s1), s1)    // 3 3 [1 2 3 ]

    s2 := s1[1:]
    fmt.Println(len(s2), cap(s2), s2)    // 2 2 [2 3]

    for i := range s2 {
        s2[i] += 20
    }
    // 此时的 s1 与 s2 是指向同一个底层数组的
    fmt.Println(s1)        // [1 22 23]
    fmt.Println(s2)        // [22 23]

    s2 = append(s2, 4)    // 向容量为 2 的 s2 中再追加元素，此时将分配新数组来存

    for i := range s2 {
        s2[i] += 10
    }
    fmt.Println(s1)        // [1 22 23]    // 此时的 s1 不再更新，为旧数据
    fmt.Println(s2)        // [32 33 14]
}
```

10. for 语句中的迭代变量与闭包函数

for 语句中的迭代变量在每次迭代中都会重用，即 for 中创建的闭包函数接收到的参数始终是同一个变量，在 goroutine 开始执行时都会得到同一个迭代值：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
```

```
        go func() {
            fmt.Println(v)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 three three three
}
```

最简单的解决方法：无需修改 goroutine 函数，在 for 内部使用局部变量保存迭代值，再传参：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        vCopy := v
        go func() {
            fmt.Println(vCopy)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

另一个解决方法：直接将当前的迭代值以参数形式传递给匿名函数：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        go func(in string) {
            fmt.Println(in)
        }(v)
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

11. 更新 map 字段的值

如果 map 一个字段的值是 struct 类型，则无法直接更新该 struct 的单个字段：

```
// 无法直接更新 struct 的字段值
type data struct {
    name string
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    m["x"].name = "Jerry"
}
```

cannot assign to struct field m["x"].name in map

因为 map 中的元素是不可寻址的。需区分开的是，slice 的元素可寻址：

```
type data struct {
    name string
}

func main() {
    s := []data{{"Tom"}}
    s[0].name = "Jerry"
    fmt.Println(s)    // [{Jerry}]
}
```

注意：不久前 gccgo 编译器可更新 map struct 元素的字段值，不过很快便修复了，官方认为是 Go1.3 的潜在特性，无需及时实现，依旧在 todo list 中。

更新 map 中 struct 元素的字段值，有 2 个方法：

- 使用局部变量

```
// 提取整个 struct 到局部变量中，修改字段值后再整个赋值
type data struct {
    name string
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    r := m["x"]
    r.name = "Jerry"
```

```
    m["x"] = r
    fmt.Println(m)    // map[x:{Jerry}]
}
```

- 使用指向元素的 map 指针

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }

    m["x"].name = "Jerry"    // 直接修改 m["x"] 中的字段
    fmt.Println(m["x"])    // &{Jerry}
}
```

但是要注意下边这种误用：

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }
    m["z"].name = "what???"
    fmt.Println(m["x"])
}
```

panic: runtime error: invalid memory address or nil pointer dereference

总结

推荐上手go前系统性地学习一下go的集合相关操作，避免踩坑。

感谢原作者 Kyle Quest 总结的这篇博客，让我受益匪浅。

GitHub refer: Introduction-to-Golang