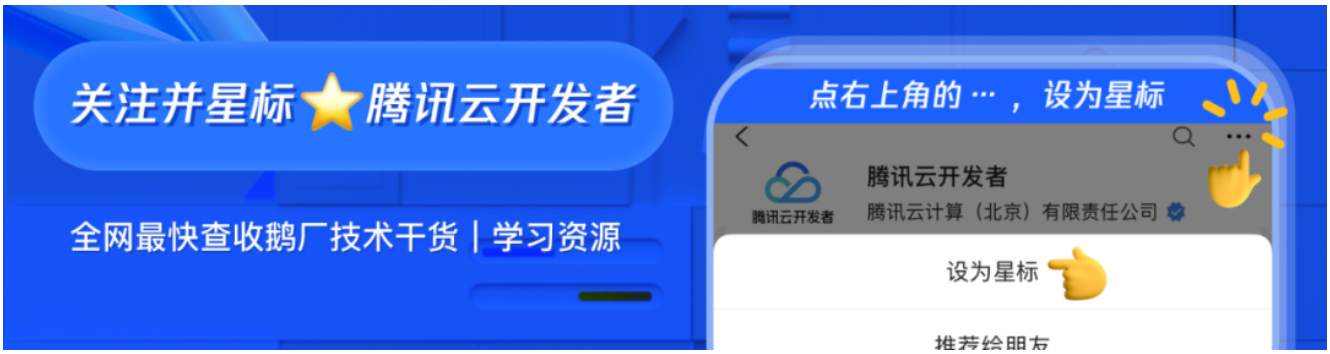
我为什么放弃Go语言?

luozhiyun 腾讯云开发者 2023-06-05 19:00 发表于广东



👉 腾小云导读

你在什么时候会产生“想要放弃用 Go 语言”的念头？也许是在用 Go 开发过程中，接连不断踩坑的时候。本文作者提炼和总结《100 Go Mistakes and How to Avoid Them》里的精华内容，并结合自身的工作经验，盘点了 Go 的常见典型错误，撰写了这篇超全避坑指南。让我们跟随文章，一起重拾用 Go 的信心~

👉 目录

01

注意 shadow 变量

```
var client *http.Client
  if tracing {
    client, err := createClientWithTracing()
    if err != nil {
      return err
    }
    log.Println(client)
  } else {
    client, err := createDefaultClient()
    if err != nil {
      return err
    }
    log.Println(client)
  }
```

在上面这段代码中，声明了一个 client 变量，然后使用 tracing 控制变量的初始化，可能是因为没有声明 err 的缘故，使用的是 := 进行初始化，那么会导致**外层的 client 变量永远是 nil**。这个例子实际上是很容易发生在我们实际的开发中，尤其需要注意。

如果是因为 err 没有初始化的缘故，我们在初始化的时候可以这么做：

```
var client *http.Client
  var err error
  if tracing {
    client, err = createClientWithTracing()
  } else {
```

```
    ...
  }
    if err != nil { // 防止重复代码
        return err
    }
```

或者内层的变量声明换一个变量名字，这样就不容易出错了。

我们也可以使用工具分析代码是否有 shadow，先安装一下工具：

```
go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow
```

然后使用 shadow 命令：

```
go vet -vettool=C:\Users\luozhiyun\go\bin\shadow.exe .\main.go
# command-line-arguments
.\main.go:15:3: declaration of "client" shadows declaration at line 13
.\main.go:21:3: declaration of "client" shadows declaration at line 13
```

02

慎用 init 函数

使用 init 函数之前需要注意下面几件事：

2.1 init 函数会在全局变量之后被执行

init 函数并不是最先被执行的，如果声明了 const 或全局变量，那么 init 函数会在它们之后执行：

```
package main

import "fmt"
```

```
var a = func() int {
  fmt.Println("a")
  return 0
}()

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

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

// output
a
init
main
```

2.2 init 初始化按解析的依赖关系顺序执行

比如 main 包里面有 init 函数，依赖了 redis 包，main 函数执行了 redis 包的 Store 函数，恰好 redis 包里面也有 init 函数，那么执行顺序会是：

还有一种情况，如果是使用 "import _ foo" 这种方式引入的，也是会先调用 foo 包中的 init 函数。

2.3 扰乱单元测试

比如我们在 init 函数中初始了一个全局的变量，但是单测中并不需要，那么实际上会增加单测得复杂度，比如：

```
var db *sql.DB
func init(){
  dataSourceName := os.Getenv("MYSQL_DATA_SOURCE_NAME")
    d, err := sql.Open("mysql", dataSourceName)
    if err != nil {
        log.Panic(err)
    }
    db = d
}
```

在上面这个例子中 init 函数初始化了一个 db 全局变量，那么在单测的时候也会初始化一个这样的变量，但是很多单测其实是很简单的，并不需要依赖这个东西。

03

embed types 优缺点

embed types 指的是我们在 struct 里面定义的匿名的字段，如：

```
type Foo struct {
  Bar
}
type Bar struct {
  Baz int
}
```

那么在上面这个例子中，我们可以通过 Foo.Baz 直接访问到成员变量，当然也可以通过 Foo.Bar.Baz 访问。

这样在很多时候可以增加我们使用的便捷性，如果没有使用 embed types 那么可能需要很多代码，如下：

```
type Logger struct {
        writeCloser io.WriteCloser
}

func (l Logger) Write(p []byte) (int, error) {
        return l.writeCloser.Write(p)
}

func (l Logger) Close() error {
        return l.writeCloser.Close()
}

func main() {
        l := Logger{writeCloser: os.Stdout}
        _, _ = l.Write([]byte("foo"))
        _ = l.Close()
}
```

如果使用了 **embed types 我们的代码可以变得很简洁**：

```
type Logger struct {
        io.WriteCloser
}

func main() {
        l := Logger{WriteCloser: os.Stdout}
        _, _ = l.Write([]byte("foo"))
        _ = l.Close()
}
```

但是同样它也有缺点，**有些字段我们并不想 export，但是 embed types 可能给我们带出去**，例如：

```
type InMem struct {
  sync.Mutex
```

```
  m map[string]int
}

func New() *InMem {
   return &InMem{m: make(map[string]int)}
}
```

Mutex 一般并不想 export， 只想在 InMem 自己的函数中使用，如：

```
func (i *InMem) Get(key string) (int, bool) {
  i.Lock()
  v, contains := i.m[key]
  i.Unlock()
  return v, contains
}
```

但是这么写却可以让拿到 InMem 类型的变量都可以使用它里面的 Lock 方法：

```
m := inmem.New()
m.Lock() // ??
```

04

Functional Options Pattern传递参数

这种方法在很多 Go 开源库都有看到过使用，比如 zap、GRPC 等。

它经常用在需要传递和初始化校验参数列表的时候使用，比如我们现在需要初始化一个 HTTP server，里面可能包含了 port、timeout 等等信息，但是参数列表很多，不能直接写在函数上，并且我们要满足灵活配置的要求，毕竟不是每个 server 都需要很多参数。那么我们可以：

- 设置一个**不导出的 struct 叫 options**，用来存放配置参数；

- 创建一个类型 type Option func(options *options) error，用这个类型来作为返回值；

比如我们现在要给 HTTP server 里面设置一个 port 参数，那么我们可以这么声明一个 WithPort 函数，返回 Option 类型的闭包，当这个闭包执行的时候会将 options 的 port 填充进去：

```
type options struct {
        port *int
}

type Option func(options *options) error

func WithPort(port int) Option {
         // 所有的类型校验，赋值，初始化啥的都可以放到这个闭包里面做
        return func(options *options) error {
                if port < 0 {
                        return errors.New("port should be positive")
                }
                options.port = &port
                return nil
        }
}
```

假如我们现在有一个这样的 Option 函数集，除了上面的 port 以外，还可以填充 timeout 等。然后我们可以利用 NewServer 创建我们的 server：

```
func NewServer(addr string, opts ...Option) (*http.Server, error) {
        var options options
        // 遍历所有的 Option
        for _, opt := range opts {
                // 执行闭包
                err := opt(&options)
                if err != nil {
                        return nil, err
                }
        }

        // 接下来可以填充我们的业务逻辑，比如这里设置默认的port 等等
        var port int
        if options.port == nil {
```

```
                port = defaultHTTPPort
        } else {
                if *options.port == 0 {
                        port = randomPort()
                } else {
                        port = *options.port
                }
        }

        // ...
}
```

初始化 server：

```
server, err := httplib.NewServer("localhost",
               httplib.WithPort(8080),
               httplib.WithTimeout(time.Second))
```

这样写的话就比较灵活，如果只想生成一个简单的 server，我们的代码可以变得很简单：

```
server, err := httplib.NewServer("localhost")
```

05

小心八进制整数

比如下面例子：

```
sum := 100 + 010
  fmt.Println(sum)
```

你以为要输出110，其实输出的是 108，因为**在 Go 中以 0 开头的整数表示八进制。**

它经常用在处理 Linux 权限相关的代码上，如下面打开一个文件：

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0644)
```

所以为了可读性，我们在用八进制的时候最好使用 "0o" 的方式表示，比如上面这段代码可以表示为：

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0o644)
```

06

float 的精度问题

在 Go 中浮点数表示方式和其他语言一样，都是通过科学计数法表示，float 在存储中分为三部分：

- 符号位（Sign）：0代表正，1代表为负
- 指数位（Exponent）:用于存储科学计数法中的指数数据，并且采用移位存储
- 尾数部分（Mantissa）： 尾数部分

计算规则我就不在这里展示了，感兴趣的可以自己去查查，我这里说说这种计数法在 Go 里面会有哪些问题。

```
func f1(n int) float64 {
  result := 10_000.
```

```
  for i := 0; i < n; i++ {
    result += 1.0001
  }
  return result
}

func f2(n int) float64 {
  result := 0.
  for i := 0; i < n; i++ {
    result += 1.0001
  }
  return result + 10_000.
}
```

在上面这段代码中，我们简单地做了一下加法：

n	Exact result	f1	f2
10	10010.001	10010.001	10010.001
1k	11000.1	11000.1	11000.1
1m	1.01E+06	1.01E+06	1.01E+06

可以看到 n 越大，误差就越大，并且 f2 的误差是小于 f1的。

对于乘法我们可以做下面的实验：

```
a := 100000.001
b := 1.0001
c := 1.0002

fmt.Println(a * (b + c))
fmt.Println(a*b + a*c)
```

输出：

```
200030.00200030004
200030.0020003
```

正确输出应该是 200030.0020003，所以它们实际上都有一定的误差，但是可以看到**先乘再加精度丢失会更小**。

如果想要准确计算浮点的话，可以尝试 "github.com/shopspring/decimal" 库，换成这个库我们再来计算一下：

```
a := decimal.NewFromFloat(100000.001)
b := decimal.NewFromFloat(1.0001)
c := decimal.NewFromFloat(1.0002)

fmt.Println(a.Mul(b.Add(c))) //200030.0020003
```

07

slice 相关注意点

7.1 区分 slice 的 length 和 capacity

首先让我们初始化一个带有 length 和 capacity 的 slice：

```
s := make([]int, 3, 6)
```

在 make 函数里面，capacity 是可选的参数。上面这段代码我们创建了一个 length 是 3，capacity 是 6 的 slice，那么底层的数据结构是这样的：

slice 的底层实际上指向了一个数组。当然，由于我们的 length 是 3，所以这样设置 s[4] = 0 会 panic 的。需要使用 append 才能添加新元素。

```
panic: runtime error: index out of range [4] with length 3
```

当 appned 超过 cap 大小的时候，slice 会自动帮我们扩容，**在元素数量小于 1024 的时候每次会扩大一倍，当超过了 1024 个元素每次扩大 25%**。

有时候我们会使用 : 操作符从另一个 slice 上面创建一个新切片：

```
s1 := make([]int, 3, 6)
s2 := s1[1:3]
```

实际上这两个 slice 还是指向了底层同样的数组，构如下：

由于指向了同一个数组，那么当我们改变第一个槽位的时候，比如 s1[1]=2，实际上两个 slice 的数据都会发生改变：

但是当我们使用 append 的时候情况会有所不同：

```
s2 = append(s2, 3)

fmt.Println(s1) // [0 2 0]
fmt.Println(s2) // [2 0 3]
```


s1 的 len 并没有被改变，所以看到的还是3元素。

还有一件比较有趣的细节是，如果再接着 append s1 那么第四个元素会被覆盖掉：

```
s1 = append(s1, 4)
  fmt.Println(s1) // [0 2 0 4]
  fmt.Println(s2) // [2 0 4]
```

我们再继续 append s2 直到 s2 发生扩容，这个时候会发现 s2 实际上和 s1 指向的不是同一个数组了：

```
s2 = append(s2, 5, 6, 7)
fmt.Println(s1) //[0 2 0 4]
fmt.Println(s2) //[2 0 4 5 6 7]
```


除了上面这种情况，还有一种情况 append 会产生意想不到的效果：

```
s1 := []int{1, 2, 3}
s2 := s1[1:2]
s3 := append(s2, 10)
```

如果 print 它们应该是这样：

```
s1=[1 2 10], s2=[2], s3=[2 10]
```

7.2 slice 初始化

对于 slice 的初始化实际上有很多种方式：

```
func main() {
        var s []string
        log(1, s)

        s = []string(nil)
        log(2, s)

        s = []string{}
        log(3, s)

        s = make([]string, 0)
        log(4, s)
}

func log(i int, s []string) {
        fmt.Printf("%d: empty=%t\tnil=%t\n", i, len(s) == 0, s == nil)
}
```

输出：

```
1: empty=true   nil=true
2: empty=true   nil=true
3: empty=true   nil=false
4: empty=true   nil=false
```

前两种方式会创建一个 nil 的 slice，后两种会进行初始化，并且这些 slice 的大小都为 0 。

对于 var s []string 这种方式来说，好处就是**不用做任何的内存分配**。比如下面场景可能可以节省一次内存分配：

```
func f() []string {
        var s []string
        if foo() {
                s = append(s, "foo")
        }
        if bar() {
                s = append(s, "bar")
        }
        return s
}
```

对于 s := []string{} 这种方式来说，它**比较适合初始化一个已知元素的 slice**：

```
s := []string{"foo", "bar", "baz"}
```

如果没有这个需求其实用 var s []string 比较好，反正在使用的适合都是通过 append 添加元素， var s []string 还能节省一次内存分配。

如果我们初始化了一个空的 slice， 那么**最好是使用 len(xxx) == 0 来判断 slice 是不是空的**，如果使用 nil 来判断可能会永远非空的情况，因为对于 s := []string{} 和 s = make([]string, 0) 这两种初始化都是非 nil 的。

对于 []string(nil) 这种初始化的方式，使用场景很少，一种比较方便地使用场景是用它来进行 slice 的 copy：

```
src := []int{0, 1, 2}
dst := append([]int(nil), src...)
```

对于 make 来说，它可以初始化 slice 的 length 和 capacity，如果我们能确定 slice 里面会存放多少元素，**从性能的角度考虑最好使用 make 初始化好**，因为对于一个空的 slice append 元素进去每次达到阈值都需要进行扩容，下面是填充 100 万元素的 benchmark：

```
BenchmarkConvert_EmptySlice-4 22 49739882 ns/op
BenchmarkConvert_GivenCapacity-4 86 13438544 ns/op
BenchmarkConvert_GivenLength-4 91 12800411 ns/op
```

可以看到，如果我们提前填充好 slice 的容量大小，性能是空 slice 的四倍，因为少了扩容时元素复制以及重新申请新数组的开销。

7.3 copy slice

```
src := []int{0, 1, 2}
var dst []int
copy(dst, src)
fmt.Println(dst) // []
```

使用 copy 函数 copy slice 的时候需要注意，上面这种情况实际上会 copy 失败，因为对 slice 来说是由 length 来控制可用数据，copy 并没有复制这个字段，要想 copy 我们可以这么做：

```
src := []int{0, 1, 2}
dst := make([]int, len(src))
copy(dst, src)
fmt.Println(dst) //[0 1 2]
```

除此之外也可以用上面提到的:

```
src := []int{0, 1, 2}
dst := append([]int(nil), src...)
```

7.4 slice capacity内存释放问题

先来看个例子:

```
type Foo struct {
  v []byte
}

func keepFirstTwoElementsOnly(foos []Foo) []Foo {
  return foos[:2]
}

func main() {
  foos := make([]Foo, 1_000)
  printAlloc()

  for i := 0; i < len(foos); i++ {
    foos[i] = Foo{
      v: make([]byte, 1024*1024),
    }
  }
  printAlloc()

  two := keepFirstTwoElementsOnly(foos)
  runtime.GC()
  printAlloc()
  runtime.KeepAlive(two)
}
```

上面这个例子中使用 printAlloc 函数来打印内存占用:

```
func printAlloc() {
  var m runtime.MemStats
  runtime.ReadMemStats(&m)
  fmt.Printf("%d KB\n", m.Alloc/1024)
}
```

上面 foos 初始化了 1000 个容量的 slice，里面 Foo struct 每个都持有 1M 内存的 slice，然后通过 keepFirstTwoElementsOnly 返回持有前两个元素的 Foo 切片，我们的想法是手动执行 GC 之后其他的 998 个 Foo 会被 GC 销毁，但是输出结果如下：

```
387 KB
1024315 KB
1024319 KB
```

实际上并没有，原因就是实际上 keepFirstTwoElementsOnly 返回的 slice 底层持有的数组是和 foos 持有的同一个：

所以我们真的要只返回 slice 的前2个元素的话应该这样做：

```
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
        res := make([]Foo, 2)
        copy(res, foos)
        return res
}
```

不过上面这种方法会初始化一个新的 slice，然后将两个元素 copy 过去。不想进行多余的分配可以这么做：

```
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
        for i := 2; i < len(foos); i++ {
                foos[i].v = nil
        }
        return foos[:2]
}
```

08

注意 range

8.1 copy 的问题

使用 range 的时候如果我们直接修改它返回的数据会不生效，因为返回的数据并不是原始数据：

```
type account struct {
  balance float32
}

  accounts := []account{
    {balance: 100.},
    {balance: 200.},
    {balance: 300.},
  }
  for _, a := range accounts {
    a.balance += 1000
  }
```

如果像上面这么做，那么输出的 accounts 是：

```
[{100} {200} {300}]
```

所以我们想要改变 range 中的数据可以这么做：

```
for i := range accounts {
  accounts[i].balance += 1000
}
```

range slice 的话也会 copy 一份：

```
s := []int{0, 1, 2}
for range s {
  s = append(s, 10)
}
```

这份代码在 range 的时候会 copy 一份，因此只会调用三次 append 后停止。

8.2 指针问题

比方我们想要 range slice 并将返回值存到 map 里面供后面业务使用，类似这样：

```
type Customer struct {
    ID string
    Balance float64
}

test := []Customer{
      {ID: "1", Balance: 10},
      {ID: "2", Balance: -10},
      {ID: "3", Balance: 0},
}

var m map[string]*Customer
for _, customer := range test {
    m[customer.ID] = &customer
}
```

但是这样遍历 map 里面存的并不是我们想要的，你会发现存的 value 都是最后一个：

```
{"1":{"ID":"3","Balance":0},"2":{"ID":"3","Balance":0},"3":{"ID":"3","Ba
```

这是因为当我们使用 range 遍历 slice 的时候，返回的 customer 变量实际上是一个固定的地址：

```
for _, customer := range test {
    fmt.Printf("%p\n", &customer) //我们想要获取这个指针的时候
}
```

输出：

```
0x1400000e240
0x1400000e240
0x1400000e240
```

这是因为迭代器会把数据都放入到 0x1400000e240 这块空间里面：

所以我们可以这样在 range 里面获取指针：

```
for _, customer := range test {
    current := customer // 使用局部变量
    fmt.Printf("%p\n", &current) // 这里获取的指针是 range copy 出来元素的指针
  }
```

或者：

```
for i := range test {
    current := &test[i] // 使用局部变量
    fmt.Printf("%p\n", current)
  }
```

09

注意break作用域

比方说：

```
for i := 0; i < 5; i++ {
      fmt.Printf("%d ", i)

      switch i {
      default:
      case 2:
              break
      }
  }
```

上面这个代码本来想 break 停止遍历，**实际上只是 break 了 switch 作用域**，print 依然会打印：0，1，2，3，4。

正确做法应该是通过 label 的方式 break：

```
loop:
  for i := 0; i < 5; i++ {
    fmt.Printf("%d ", i)
    switch i {
    default:
    case 2:
      break loop
```

```
    }
  }
```

有时候我们会没注意到自己的错误用法，比如下面：

```
for {
    select {
    case <-ch:
      // Do something
    case <-ctx.Done():
      break
    }
  }
```

上面这种写法会导致只跳出了 select，并没有终止 for 循环，正确写法应该这样：

```
loop:
  for {
    select {
    case <-ch:
      // Do something
    case <-ctx.Done():
      break loop
    }
  }
```

10

defer

10.1 注意 defer 的调用时机

有时候我们会像下面一样使用 defer 去关闭一些资源：

```
func readFiles(ch <-chan string) error {
            for path := range ch {
                    file, err := os.Open(path)
                    if err != nil {
                            return err
                    }

                    defer file.Close()

                    // Do something with file
            }
            return nil
}
```

因为defer会在方法结束的时候调用，但是**如果上面的 readFiles 函数永远没有 return，那么 defer 将永远不会被调用**，从而造成内存泄露。并且 defer 写在 for 循环里面，编译器也无法做优化，会影响代码执行性能。

为了避免这种情况，我们可以 wrap 一层：

```
func readFiles(ch <-chan string) error {
      for path := range ch {
          if err := readFile(path); err != nil {
                  return err
          }
      }
      return nil
}

func readFile(path string) error {
      file, err := os.Open(path)
      if err != nil {
              return err
      }

      defer file.Close()

      // Do something with file
      return nil
}
```

10.2 注意 defer 的参数

defer 声明时会先计算确定参数的值。

```
func a() {
    i := 0
    defer notice(i) // 0
    i++
    return
}

func notice(i int) {
  fmt.Println(i)
}
```

在这个例子中，变量 i 在 defer 被调用的时候就已经确定了，而不是在 defer执行的时候，所以上面的语句输出的是 0。

所以我们想要获取这个变量的真实值，应该用引用：

```
func a() {
  i := 0
  defer notice(&i) // 1
  i++
  return
}
```

10.2 defer 下的闭包

```
func a() int {
  i := 0
  defer func() {
    fmt.Println(i + 1) //12
  }()
  i++
  return i+10
```

```
}

func TestA(t *testing.T) {
  fmt.Println(a()) //11
}
```

如果换成闭包的话，**实际上闭包中对变量i是通过**指针传递**的**，所以可以读到真实的值。但是上面的例子中 a 函数返回的是 11 是因为执行顺序是：

```
先计算（i+10）-> （call defer） -> （return）
```

11

string 相关

11.1 迭代带来的问题

在 Go 语言中，字符串是一种基本类型，**默认是通过 utf8 编码的字符序列**，当字符为 ASCII 码时则占用 1 个字节，其他字符根据需要占用 2-4 个字节，比如中文编码通常需要 3 个字节。

那么我们在做 string 迭代的时候可能会产生意想不到的问题：

```
s := "hêllo"
  for i := range s {
    fmt.Printf("position %d: %c\n", i, s[i])
  }
  fmt.Printf("len=%d\n", len(s))
```

输出：

```
position 0: h
position 1: Ã
position 3: l
```

```
position 4: l
position 5: o
len=6
```

上面的输出中发现第二个字符是 Ã，不是 ê，并且位置2的输出"消失"了，这其实就是因为 ê 在 utf8 里面实际上占用 2 个 byte：

s	h	ê	l	l	o
[]byte(s)	68	c3 aa	6c	6c	6f

所以我们在迭代的时候 s[1] 等于 c3 这个 byte 等价 Ã 这个 utf8 值，所以输出的是 hÃllo 而不是 hêllo。

那么根据上面的分析，我们就可以知道在迭代获取字符的时候不能只获取单个 byte，应该使用 range 返回的 value值：

```
s := "hêllo"
  for i, v := range s {
    fmt.Printf("position %d: %c\n", i, v)
  }
```

或者我们可以把 string 转成 rune 数组，在 go 中 rune 代表 Unicode码位，用它可以输出单个字符：

```
s := "hêllo"
  runes := []rune(s)
  for i, _ := range runes {
    fmt.Printf("position %d: %c\n", i, runes[i])
  }
```

输出：

```
position 0: h
position 1: ê
position 2: l
position 3: l
position 4: o
```

11.2 截断带来的问题

在上面我们讲 slice 的时候也提到了，**在对slice使用：操作符进行截断的时候，底层的数组实际上指向同一个**，在 string 里面也需要注意这个问题，比如下面：

```
func (s store) handleLog(log string) error {
            if len(log) < 36 {
                    return errors.New("log is not correctly formatted")
            }
            uuid := log[:36]
            s.store(uuid)
            // Do something
    }
```

这段代码用了：操作符进行截断，但是如果 log 这个对象很大，比如上面的 store 方法把 uuid 一直存在内存里，可能会造成底层的数组一直不释放，从而造成内存泄露。

为了解决这个问题，我们可以先复制一份再处理：

```
func (s store) handleLog(log string) error {
            if len(log) < 36 {
                    return errors.New("log is not correctly formatted")
            }
            uuid := strings.Clone(log[:36]) // copy一份
            s.store(uuid)
            // Do something
    }
```

interface 类型返回的非 nil 问题

假如我们想要继承 error 接口实现一个自己的 MultiError：

```
type MultiError struct {
  errs []string
}

func (m *MultiError) Add(err error) {
  m.errs = append(m.errs, err.Error())
}

func (m *MultiError) Error() string {
  return strings.Join(m.errs, ";")
}
```

然后在使用的时候返回 error，并且想通过 error 是否为 nil 判断是否有错误：

```
func Validate(age int, name string) error {
  var m *MultiError
  if age < 0 {
    m = &MultiError{}
    m.Add(errors.New("age is negative"))
  }
  if name == "" {
    if m == nil {
      m = &MultiError{}
    }
    m.Add(errors.New("name is nil"))
  }

  return m
}

func Test(t *testing.T) {
  if err := Validate(10, "a"); err != nil {
    t.Errorf("invalid")
  }
}
```

实际上 Validate 返回的 err 会总是为**非 nil 的**，也就是上面代码只会输出 invalid:

```
invalid <nil>
```


13

Error

13.1 error wrap

对于 err 的 return 我们一般可以这么处理:

```
err:= xxx()
 if err != nil {
   return err
 }
```

但是这样处理只是简单地将原始的错误抛出去了，无法知道当前处理的这段程序的上下文信息，这个时候我们可能会自定义个 error 结构体，继承 error 接口:

```
err:= xxx()
 if err != nil {
```

```
    return XXError{Err: err}
}
```

然后我们把上下文信息都加到 XXError 中，但是这样虽然可以添加一些上下文信息，但是每次都需要创建一个特定类型的 error 类会变得很麻烦，那么在 1.13 之后，我们可以使用 %w 进行 wrap。

```
if err != nil {
    return fmt.Errorf("xxx failed: %w", err)
}
```

当然除了上面这种做法以外，我们还可以直接 %v 直接格式化我们的错误信息：

```
if err != nil {
    return fmt.Errorf("xxx failed: %v", err)
}
```

这样做的缺点就是我们会丢失这个 err 的类型信息，如果不需要这个类型信息，只是想往上抛打印一些日志当然也无所谓。

13.2 error Is & As

因为我们的 error 可以会被 wrap 好几层，那么使用 == 是可能无法判断我们的 error 究竟是不是我们想要的特定的 error，那么可以用 errors.Is：

```
var BaseErr = errors.New("base error")

func main() {
    err1 := fmt.Errorf("wrap base: %w", BaseErr)
    err2 := fmt.Errorf("wrap err1: %w", err1)
    println(err2 == BaseErr)

    if !errors.Is(err2, BaseErr) {
        panic("err2 is not BaseErr")
    }
```

```
    println("err2 is BaseErr")
}
```

输出：

```
false
err2 is BaseErr
```

在上面，我们通过 errors.Is 就可以判断出 err2 里面包含了 BaseErr 错误。errors.Is 里面会递归调用 Unwrap 方法拆包装，然后挨个使用 == 判断是否和指定类型的 error 相等。

errors.As 主要用来做类型判断，原因也是和上面一样，error 被 wrap 之后我们通过 err.(type) 无法直接判断，errors.As 会用 Unwrap 方法拆包装，然后挨个判断类型。使用如下：

```
type TypicalErr struct {
    e string
}

func (t TypicalErr) Error() string {
    return t.e
}

func main() {
    err := TypicalErr{"typical error"}
    err1 := fmt.Errorf("wrap err: %w", err)
    err2 := fmt.Errorf("wrap err1: %w", err1)
    var e TypicalErr
    if !errors.As(err2, &e) {
        panic("TypicalErr is not on the chain of err2")
    }
    println("TypicalErr is on the chain of err2")
    println(err == e)
}
```

输出：

```
TypicalErr is on the chain of err2
true
```

13.3 处理 defer 中的 error

比如下面代码，我们如果在调用 Close 的时候报错是没有处理的：

```
func getBalance(db *sql.DB, clientID string) (
            float32, error) {
            rows, err := db.Query(query, clientID)
            if err != nil {
                    return 0, err
            }
            defer rows.Close()

            // Use rows
    }
```

那么也许我们可以在 defer 中打印一些 log，但是无法 return，defer 不接受一个 err 类型的返回值：

```
defer func() {
            err := rows.Close()
            if err != nil {
                    log.Printf("failed to close rows: %v", err)
            }
            return err //无法通过编译
    }()
```

那么我们可能想通过默认 err 返回值的方式将 defer 的 error 也返回了：

```
func getBalance(db *sql.DB, clientID string) (balance float32, err error)
            rows, err = db.Query(query, clientID)
            if err != nil {
```

```
                return 0, err
        }
        defer func() {
                err = rows.Close()
        }()

        // Use rows
}
```

上面代码看起来没问题，那么假如 Query 的时候和 Close 的时候同时发生异常呢？其中有一个 error 会被覆盖，那么我们可以根据自己的需求选择一个打印日志，另一个 error 返回：

```
defer func() {
        closeErr := rows.Close()
        if err != nil {
                if closeErr != nil {
                        log.Printf("failed to close rows: %v", err)
                }
                return
        }
        err = closeErr
}()
```

14

happens before 保证

创建 goroutine 发生先于 goroutine 执行，所以下面这段代码先读一个变量，然后在 goroutine 中写变量不会发生 data race 问题：

```
i := 0
go func() {
        i++
}()
```

goroutine 退出没有任何 happen before保证，例如下面代码会有 data race：

```
i := 0
    go func() {
            i++
    }()
    fmt.Println(i)
```

channel 操作中 send 操作是 happens before receive 操作：

```
var c = make(chan int, 10)
var a string

func f() {
  a = "hello, world"
  c <- 0
}

func main() {
  go f()
  <-c
  print(a)
}
```

上面执行顺序应该是：

```
variable change -》channel send -》channel receive -》variable read
```

上面能够保证一定输出 "hello, world"。

close channel 是 happens before receive 操作，所以下面这个例子中也不会有 data race 问题：

```
i := 0
    ch := make(chan struct{})
    go func() {
```

```
            <-ch
            fmt.Println(i)
    }()
    i++
    close(ch)
```

在无缓冲的 channel 中 receive 操作是 happens before send 操作的，例如：

```
var c = make(chan int)
var a string

func f() {
  a = "hello, world"
  <-c
}

func main() {
  go f()
  c <- 0
  print(a)
}
```

这里同样能保证输出 hello, world。

15

Context Values

在 context 里面我们可以通过 key value 的形式传递一些信息：

context.WithValue 是从 parentCtx 创建，所以创建出来的 ctx 既包含了父类的上下文信息，也包含了当前新加的上下文。

```
fmt.Println(ctx.Value("key"))
```

使用的时候可以直接通过 Value 函数输出。那么其实就可以想到，如果 key 相同的话后面的值会覆盖前面的值的，所以在写 key 的时候可以自**定义一个非导出的类型作为 key 来保证唯一**：

```
package provider

    type key string

    const myCustomKey key = "key"

    func f(ctx context.Context) {
            ctx = context.WithValue(ctx, myCustomKey, "foo")
            // ...
    }
```

16

应多关注 goroutine 何时停止

很多同学觉得 goroutine 比较轻量，认为可以随意地启动 goroutine 去执行任何而不会有很大的性能损耗。这个观点基本没错，但是如果在 goroutine 启动之后因为代码问题导致它一直占用，没有停止，数量多了之后**可能会造成内存泄漏**。

比如下面的例子：

```
ch := foo()
    go func() {
            for v := range ch {
                    // ...
            }
    }()
```

如果在该 goroutine 中的 channel 一直没有关闭，那么这个 goroutine 就不会结束，会一直挂着占用一部分内存。

还有一种情况是我们的主进程已经停止运行了，但是 goroutine 里面的任务还没结束就被主进程杀掉了，那么这样也可能造成我们的任务执行出问题，比如资源没有释放，抑或是数据

还没处理完等等，如下：

```
func main() {
            newWatcher()

            // Run the application
    }

    type watcher struct { /* Some resources */ }

    func newWatcher() {
            w := watcher{}
            go w.watch()
    }
```

上面这段代码就可能出现主进程已经执行 over 了，但是 watch 函数还没跑完的情况，那么其实可以通过设置 stop 函数，让主进程执行完之后执行 stop 函数即可：

```
func main() {
            w := newWatcher()
            defer w.close()

            // Run the application
    }

    func newWatcher() watcher {
            w := watcher{}
            go w.watch()
            return w
    }

    func (w watcher) close() {
            // Close the resources
    }
```

17

Channel

17.1 select & channel

select 和 channel 搭配起来往往有意想不到的效果，比如下面：

```
for {
            select {
            case v := <-messageCh:
                    fmt.Println(v)
            case <-disconnectCh:
                    fmt.Println("disconnection, return")
                    return
            }
    }
```

上面代码中接受了 messageCh 和 disconnectCh 两个 channel 的数据，如果我们想先接受 messageCh 的数组再接受 disconnectCh 的数据，那么上面代码会产生bug ，如：

```
for i := 0; i < 10; i++ {
            messageCh <- i
    }
    disconnectCh <- struct{}{}
```

我们想要上面的 select 先输出完 messageCh 里面的数据，然后再 return，实际上可能会输出：

```
0
1
2
3
4
disconnection, return
```

这是因为 select 不像 switch 会依次匹配 case 分支，select 会**随机**执行下面的 case 分支，所以想要做到先消费 messageCh channel 数据，如果**只有单个 goroutine** 生产数据可以这样做：

- 使用无缓冲的 messageCh channel，这样在发送数据的时候会一直等待，直到数据被消费了才会往下走，相当于是个同步模型了（无缓冲的 channel 是 receive happens before send）；
- 在 select 里面使用单个channel，比如面的 demo 中我们可以定义一种特殊的 tag 来结束 channel，当读到这个特殊的 tag 的时候 return，这样就没必要用两个 channel 了。

如果有多个 goroutine 生产数据，那么可以这样：

```
for {
   select {
    case v := <-messageCh:
        fmt.Println(v)
    case <-disconnectCh:
        for {
           select {
                case v := <-messageCh:
                        fmt.Println(v)
                default:
                        fmt.Println("disconnection, return")
                    return
                        }
                    }
            }
    }
```

在读取 disconnectCh 的时候里面再套一个循环读取 messageCh，读完了之后会调用 default 分支进行 return。

17.2 不要使用 nil channel

使用 nil channel 进行收发数据的时候会永远阻塞，例如发送数据：

```
var ch chan int
ch <- 0 //block
```

接收数据：

```
var ch chan int
<-ch //block
```

17.3 Channel 的 close 问题

channel 在 close 之后仍然可以接收数据的，例如：

```
ch1 := make(chan int, 1)
  close(ch1)
  for {
    v := <-ch1
    fmt.Println(v)
  }
```

这段代码会一直 print 0。这会导致什么问题呢？比如我们想要将两个 channel 的数据汇集到另一个 channel 中：

```
func merge(ch1, ch2 <-chan int) <-chan int {
        ch := make(chan int, 1)
        go func() {
          for {
            select {
            case v:= <-ch1:
              ch <- v
            case v:= <-ch2:
              ch <- v
            }
          }
          close(ch) // 永远运行不到
        }()
        return ch
}
```

由于 channel 被 close 了还可以接收到数据，所以上面代码中，即使 ch1 和 ch2 都被 close 了，也是运行不到 close(ch) 这段代码，并且还一直将 0 推入到 ch channel 中。所以为了感知到 channel 被关闭了，我们应该使用 channel 返回的两个参数：

```
    v, open := <-ch1
    fmt.Print(v, open) //open返回false 表示没有被关闭
```

那么回到我们上面的例子中，就可以这样做：

```
func merge(ch1, ch2 <-chan int) <-chan int {
  ch := make(chan int, 1)
  ch1Closed := false
  ch2Closed := false

  go func() {
    for {
      select {
      case v, open := <-ch1:
        if !open { // 如果已经关闭
          ch1Closed = true //标记为true
          break
        }
        ch <- v
      case v, open := <-ch2:
        if !open { // 如果已经关闭
          ch2Closed = true//标记为true
          break
        }
        ch <- v
      }

      if ch1Closed && ch2Closed {//都关闭了
        close(ch)//关闭ch
        return
      }
    }
  }()
  return ch
}
```

通过两个标记以及返回的 open 变量就可以判断 channel 是否被关闭了，如果都关闭了，那么执行 close(ch)。

18

string format 带来的 dead lock

如果类型定义了 String() 方法，它会被用在 fmt.Printf() 中生成默认的输出：等同于使用格式化描述符 %v 产生的输出。还有 fmt.Print() 和 fmt.Println() 也会自动使用 String() 方法。

那么我们看看下面的例子：

```
type Customer struct {
  mutex sync.RWMutex
  id string
  age int
}

func (c *Customer) UpdateAge(age int) error {
  c.mutex.Lock()
  defer c.mutex.Unlock()

  if age < 0 {
    return fmt.Errorf("age should be positive for customer %v", c)
  }

  c.age = age
  return nil
}

func (c *Customer) String() string {
  fmt.Println("enter string method")
  c.mutex.RLock()
  defer c.mutex.RUnlock()
  return fmt.Sprintf("id %s, age %d", c.id, c.age)
}
```

这个例子中，如果调用 UpdateAge 方法 age 小于0会调用 fmt.Errorf，格式化输出，这个时候 String() 方法里面也进行了加锁，那么这样会造成死锁。

```
mutex.Lock -> check age -> Format error -> call String() -> mutex.RLock
```

解决方法也很简单，一个是缩小锁的范围，在 check age 之后再加锁，另一种方法是 Format error 的时候不要 Format 整个结构体，可以改成 Format id 就行了。

19

错误使用 sync.WaitGroup

sync.WaitGroup 通常用在并发中等待 goroutines 任务完成，用 Add 方法添加计数器，当任务完成后需要调用 Done 方法让计数器减一。等待的线程会调用 Wait 方法等待，直到 sync.WaitGroup 内计数器为零。

需要注意的是 Add 方法是怎么使用的，如下：

```
wg := sync.WaitGroup{}
    var v uint64

    for i := 0; i < 3; i++ {
            go func() {
                    wg.Add(1)
                    atomic.AddUint64(&v, 1)
                    wg.Done()
            }()
    }

    wg.Wait()
    fmt.Println(v)
```

这样使用可能会导致 v 不一定等于3，因为在 for 循环里面创建的 3 个 goroutines 不一定比外面的主线程先执行，从而导致在调用 Add 方法之前可能 Wait 方法就执行了，并且恰好 sync.WaitGroup 里面计数器是零，然后就通过了。

正确的做法应该是在创建 goroutines 之前就将要创建多少个 goroutines 通过 Add 方法添加进去。

20

不要拷贝 sync 类型

sync 包里面提供一些并发操作的类型，如 mutex、condition、wait gorup 等等，这些类型都不应该被拷贝之后使用。

有时候我们在使用的时候拷贝是很隐秘的，比如下面：

```
type Counter struct {
  mu sync.Mutex
  counters map[string]int
}

func (c Counter) Increment(name string) {
  c.mu.Lock()
  defer c.mu.Unlock()
  c.counters[name]++
}

func NewCounter() Counter {
  return Counter{counters: map[string]int{}}
}

func main() {
  counter := NewCounter()
  go counter.Increment("aa")
  go counter.Increment("bb")
}
```

receiver 是一个值类型，所以调用 Increment 方法的时候实际上拷贝了一份 Counter 里面的变量。这里我们可以将 receiver 改成一个指针，或者将 sync.Mutex 变量改成指针类型。

所以如果：

- receiver 是值类型；
- 函数参数是 sync 包类型；
- 函数参数的结构体里面包含了 sync 包类型；

遇到这种情况需要注意检查一下，我们可以借用 go vet 来检测，比如上面如果并发调用了就可以检测出来：

```
» go vet . bear@BEARLUO-MB7
# github.com/cch123/gogctuner/main
./main.go:53:9: Increment passes lock by value: github.com/cch123/gogctur
```

21

time.After 内存泄露

我们用一个简单的例子模拟一下：

```
package main

import (
    "fmt"
    "time"
)
//define a channel
var chs chan int

func Get() {
    for {
        select {
            case v := <- chs:
                fmt.Printf("print:%v\n", v)
            case <- time.After(3 * time.Minute):
                fmt.Printf("time.After:%v", time.Now().Unix())
        }
    }
}

func Put() {
    var i = 0
    for {
        i++
        chs <- i
    }
}

func main() {
    chs = make(chan int, 100)
```

```
    go Put()
    Get()
}
```

逻辑很简单就是先往 channel 里面存数据，然后不停地使用 for select case 语法从 channel 里面取数据，为了防止长时间取不到数据，所以在上面加了 time.After 定时器，这里只是简单打印一下。

然后我没用 pprof 看一下内存占用：

```
$ go tool pprof -http=:8081 http://localhost:6060/debug/pprof/heap
```

发现不一会儿 Timer 的内存占用很高了。这是因为在计时器触发之前，垃圾收集器不会回收 Timer，但是在循环里面每次都调用 time.After都会实例化一个一个新的定时器，并且这个定时器会在激活之后才会被清除。

为了避免这种情况我们可以使用 下面代码：

```
func Get() {
    delay := time.NewTimer(3 * time.Minute)
```

```
    defer delay.Stop()

    for {
        delay.Reset(3 * time.Minute)

        select {
            case v := <- chs:
                fmt.Printf("print:%v\n", v)
            case <- delay.C:
                fmt.Printf("time.After:%v", time.Now().Unix())
        }
    }
}
```

22

HTTP body 忘记 Close 导致的泄露

```
type handler struct {
        client http.Client
        url string
}

func (h handler) getBody() (string, error) {
        resp, err := h.client.Get(h.url)
        if err != nil {
                return "", err
        }

        body, err := io.ReadAll(resp.Body)
        if err != nil {
                return "", err
        }

        return string(body), nil
}
```

上面这段代码看起来没什么问题，但是 resp 是 *http.Response 类型，里面包含了 Body io.ReadCloser 对象，它是一个 io 类，必须要正确关闭，否则是会产生资源泄露的。一般我们可以这么做：

```
defer func() {
        err := resp.Body.Close()
        if err != nil {
                log.Printf("failed to close response: %v\n", err)
        }
}()
```

23

Cache line

目前在计算机中，主要有两大存储器 SRAM 和 DRAM。主存储器是由 DRAM 实现的，也就是我们常说的内存，在 CPU 里通常会有 L1、L2、L3 这样三层高速缓存是用 SRAM 实现的。

当从内存中取单元到 cache 中时，会一次取一个 cacheline 大小的内存区域到 cache 中，然后存进相应的 cacheline 中，所以当你读取一个变量的时候，可能会把它相邻的变量也读取到 CPU 的缓存中(如果正好在一个 cacheline 中)，因为有很大的几率你会继续访问相邻的变量，这样 CPU 利用缓存就可以加速对内存的访问。

cacheline 大小通常有 32 bit，64 bit， 128 bit。拿我电脑的 64 bit 举例：

```
cat /sys/devices/system/cpu/cpu1/cache/index0/coherency_line_size
64
```

我们设置两个函数，一个 index 加2，一个 index 加8：

```
func sum2(s []int64) int64 {
  var total int64
  for i := 0; i < len(s); i += 2 {
    total += s[i]
  }
  return total
}

func sum8(s []int64) int64 {
  var total int64
  for i := 0; i < len(s); i += 8 {
    total += s[i]
  }
  return total
}
```

这看起来 sum8 处理的元素比 sum2 少四倍，那么性能应该也快四倍左右，书上说只快了10%，但是我没测出来这个数据，无所谓了大家知道因为 cacheline 的存在，并且数据在 L1 缓存里面性能很高就行了。

然后再看看 slice 类型的结构体和结构体里包含 slice：

```
type Foo struct {
        a int64
        b int64
}

func sumFoo(foos []Foo) int64 {
        var total int64
        for i := 0; i < len(foos); i++ {
                total += foos[i].a
        }
```

```
        return total
}
```

Foo 里面包含了两个字段 a 和 b， sumFoo 会遍历 Foo slice 将所有 a 字段加起来返回。

```
type Bar struct {
        a []int64
        b []int64
}

func sumBar(bar Bar) int64 {
        var total int64
        for i := 0; i < len(bar.a); i++ {
                total += bar.a[i]
        }
        return total
}
```

Bar 里面是包含了 a，b 两个 slice，sumBar 会将 Bar 里面的 a 的元素和相加返回。我们同样用两个 benchmark 测试一下：

```
func Benchmark_sumBar(b *testing.B) {
  s := Bar{
    a: make([]int64, 16),
    b: make([]int64, 16),
  }

  b.RunParallel(func(pb *testing.PB) {
    for pb.Next() {
      sumBar(s)
    }
  })
}

func Benchmark_sumFoo(b *testing.B) {
  s := make([]Foo, 16)

  b.RunParallel(func(pb *testing.PB) {
    for pb.Next() {
      sumFoo(s)
    }
```

```
    })
}
```

测试结果：

```
# go test -gcflags "-N -l" -bench .
Benchmark_sumBar-16 249029368 4.855 ns/op
Benchmark_sumFoo-16 238571205 5.056 ns/op
```

sumBar 会比 sumFoo 快一点的。这是因为对于 sumFoo 来说要读完整个数据才行，而对于 sumBar 来说只需要读前16 bytes 读入到 cache line：

24

关于 False Sharing 造成的性能问题

False Sharing 是由于多线程对于同一片内存进行并行读写操作的时候会造成内存缓存失效，而反复将数据载入缓存所造成的性能问题。

因为现在 CPU 的缓存都是分级的，对于 L1 缓存来说是每个 Core 所独享的，那么就有可能面临缓存数据失效的问题。

如果同一片数据被多个 Core 同时加载，那么它就是共享状态在共享状态下想要修改数据要先向所有的其他 CPU 核心广播一个请求，要求先把其他 CPU 核心里面的 cache ，都变成无效的状态，然后再更新当前 cache 里面的数据。

CPU 核心里面的 cache 变成无效之后就不能使用了，需要重新加载，因为不同级别的缓存的速度是差异很大的，所以这其实性能影响还蛮大的，我们写个测试看看。

```
type MyAtomic interface {
  IncreaseAllEles()
}

type Pad struct {
  a uint64
  _p1 [15]uint64
  b uint64
  _p2 [15]uint64
  c uint64
  _p3 [15]uint64
}

func (myatomic *Pad) IncreaseAllEles() {
  atomic.AddUint64(&myatomic.a, 1)
  atomic.AddUint64(&myatomic.b, 1)
  atomic.AddUint64(&myatomic.c, 1)
}

type NoPad struct {
  a uint64
  b uint64
  c uint64
}

func (myatomic *NoPad) IncreaseAllEles() {
  atomic.AddUint64(&myatomic.a, 1)
  atomic.AddUint64(&myatomic.b, 1)
  atomic.AddUint64(&myatomic.c, 1)
}
```

这里我定义了两个结构体 Pad 和 NoPad。然后我们定义一个 benchmark 进行多线程测试：

```
func testAtomicIncrease(myatomic MyAtomic) {
  paraNum := 1000
  addTimes := 1000
  var wg sync.WaitGroup
```

```
  wg.Add(paraNum)
  for i := 0; i < paraNum; i++ {
    go func() {
      for j := 0; j < addTimes; j++ {
        myatomic.IncreaseAllEles()
      }
      wg.Done()
    }()
  }
  wg.Wait()

}
func BenchmarkNoPad(b *testing.B) {
  myatomic := &NoPad{}
  b.ResetTimer()
  testAtomicIncrease(myatomic)
}

func BenchmarkPad(b *testing.B) {
  myatomic := &Pad{}
  b.ResetTimer()
  testAtomicIncrease(myatomic)
}
```

结果可以看到快了 40% 左右：

```
BenchmarkNoPad
BenchmarkNoPad-10       1000000000           0.1360 ns/op
BenchmarkPad
BenchmarkPad-10         1000000000           0.08887 ns/op
```

如果没有 pad 话，变量数据都会在一条 cache line 里面，这样如果其中一个线程修改了数据会导致另一个线程的 cache line 无效，需要重新加载：

加了 padding 之后数据都不在同一个 cache line 上了，即使发生了修改 invalid 不是同一行数据也不需要重新加载。

25

内存对齐

简而言之，现在的 CPU 访问内存的时候是一次性访问多个 bytes，比如64位架构一次访问8bytes ，该处理器只能从地址为8的倍数的内存开始读取数据，所以要求数据在存放的时候首地址的值是8的倍数存放，这就是所谓的内存对齐。

比如下面的例子中因为内存对齐的存在，所以下面的例子中 b 这个字段只能在后面另外找地址为8的倍数地址开始存放：

除此之外还有一个零大小字段对齐的问题，如果结构体或数组类型不包含大小大于零的字段或元素，那么它的大小就为0。比如x [0]int8,空结构体struct{} 。当它作为字段时不需要对齐，但是作为结构体最后一个字段时需要对齐。我们拿空结构体来举个例子：

```
type M struct {
    m int64
    x struct{}
}

type N struct {
    x struct{}
    n int64
```

```
}

func main() {
    m := M{}
    n := N{}
    fmt.Printf("as final field size:%d\nnot as final field size:%d\n", ur
}
```

输出:

```
as final field size:16
not as final field size:8
```

当然，我们不可能手动去调整内存对齐，我们可以通过使用工具 fieldalignment:

```
$ go install golang.org/x/tools/go/analysis/passes/fieldalignment/cmd/fie

$ fieldalignment -fix .\main\my.go
main\my.go:13:9: struct of size 24 could be 16
```

26

逃逸分析

Go 是通过在编译器里做逃逸分析（escape analysis）来决定一个对象放栈上还是放堆上，不逃逸的对象放栈上，可能逃逸的放堆上。对于 Go 来说，我们可以通过下面指令来看变量是否逃逸:

```
go run -gcflags '-m -l' main.go
```

- -m 会打印出逃逸分析的优化策略，实际上最多总共可以用 4 个 -m，但是信息量较大，一般用 1 个就可以了。
- -l 会禁用函数内联，在这里禁用掉内联能更好的观察逃逸情况，减少干扰。

26.1 指针逃逸

在函数中创建了一个对象，返回了这个对象的指针。这种情况下，函数虽然退出了，但是因为指针的存在，对象的内存不能随着函数结束而回收，因此只能分配在堆上。

```
type Demo struct {
  name string
}

func createDemo(name string) *Demo {
  d := new(Demo) // 局部变量 d 逃逸到堆
  d.name = name
  return d
}

func main() {
  demo := createDemo("demo")
  fmt.Println(demo)
}
```

我们检测一下：

```
go run -gcflags '-m -l'  .\main\main.go
# command-line-arguments
main\main.go:12:17: leaking param: name
main\main.go:13:10: new(Demo) escapes to heap
main\main.go:20:13: ... argument does not escape
&{demo}
```

26.2 interface{}/any 动态类型逃逸

因为编译期间很难确定其参数的具体类型，也会发生逃逸，例如这样：

```
func createDemo(name string) any {
  d := new(Demo) // 局部变量 d 逃逸到堆
  d.name = name
  return d
}
```

26.3 切片长度或容量没指定逃逸

如果使用局部切片时，已知切片的长度或容量，请使用常量或数值字面量来定义，否则也会逃逸：

```
func main() {
    number := 10
    s1 := make([]int, 0, number)
    for i := 0; i < number; i++ {
        s1 = append(s1, i)
    }
    s2 := make([]int, 0, 10)
    for i := 0; i < 10; i++ {
        s2 = append(s2, i)
    }
}
```

输出一下：

```
go run -gcflags '-m -l'  main.go

./main.go:65:12: make([]int, 0, number) escapes to heap
./main.go:69:12: make([]int, 0, 10) does not escape
```

26.4 闭包

例如下面：Increase() 返回值是一个闭包函数，该闭包函数访问了外部变量 n，那变量 n 将会一直存在，直到 in 被销毁。很显然，变量 n 占用的内存不能随着函数 Increase() 的退出而回收，因此将会逃逸到堆上。

```
func Increase() func() int {
  n := 0
  return func() int {
    n++
    return n
  }
}
```

```
func main() {
  in := Increase()
  fmt.Println(in()) // 1
  fmt.Println(in()) // 2
}
```

输出：

```
go run -gcflags '-m -l'  main.go

./main.go:64:5: moved to heap: n
./main.go:65:12: func literal escapes to heap
```

27

byte slice 和 string 的转换优化

直接通过强转 string(bytes) 或者 []byte(str) 会带来数据的复制，性能不佳，所以在追求极致性能场景使用 unsafe 包的方式直接进行转换来提升性能：

```
// toBytes performs unholy acts to avoid allocations
func toBytes(s string) []byte {
  return *(*[]byte)(unsafe.Pointer(&s))
}
// toString performs unholy acts to avoid allocations
func toString(b []byte) string {
  return *(*string)(unsafe.Pointer(&b))
}
```

在 Go 1.12 中，增加了几个方法 String、StringData、Slice 和 SliceData ,用来做这种性能转换。

28

容器中的 GOMAXPROCS

自 Go 1.5 开始， Go 的 GOMAXPROCS 默认值已经设置为 CPU 的核数，但是在 Docker 或 k8s 容器中 runtime.GOMAXPROCS() 获取的是 宿主机的 CPU 核数 。这样会导致 P 值设置过大，导致生成线程过多，会增加上

下文切换的负担，导致严重的上下文切换，浪费 CPU。

所以可以使用 uber 的 automaxprocs 库，大致原理是读取 CGroup 值识别容器的 CPU quota，计算得到实际核心数，并自动设置 GOMAXPROCS 线程数量。

```
import _ "go.uber.org/automaxprocs"

func main() {
  // Your application logic here
}
```

29

总结

以上就是本篇文章对《100 Go Mistakes How to Avoid Them》书中内容的技术总结，也是一些在日常使用 Go 在工作中容易忽视掉的问题。内容量较大，常见错误和技巧也很多，可以反复阅读，感兴趣的开发者可以收藏下来慢慢研究。

参考：

https://go.dev/ref/mem

https://colobu.com/2019/01/24/cacheline-affects-performance-in-go/

https://teivah.medium.com/go-and-cpu-caches-af5d32cc5592

https://geektutu.com/post/hpg-escape-analysis.html

https://github.com/uber-go/automaxprocs

https://gfw.go101.org/article/unsafe.html

-End-

原创作者 | 罗志鐨

技术责编 | 吴连火

使用Go语言时还有什么易错点？欢迎在评论区分享。我们将选取1则最有意义的分享，送出腾讯云开发者-文化衫1件（见下图）。6月12日中午12点开奖。