1. 交替打印数字和字母

问题描述

使用两个 `goroutine` 交替打印序列，一个 `goroutine` 打印数字， 另外一个 `goroutine` 打印字母， 最终效果如下：

```
12AB34CD56EF78GH910IJ1112KL1314MN1516OP1718QR1920ST2122UV2324WX2526YZ2728
```

解题思路

问题很简单，使用 channel 来控制打印的进度。使用两个 channel ，来分别控制数字和字母的打印序列， 数字打印完成后通过 channel 通知字母打印, 字母打印完成后通知数字打印，然后周而复始的工作。

源码参考

```
  letter,number := make(chan bool),make(chan bool)
  wait := sync.WaitGroup{}

  go func() {
    i := 1
    for {
      select {
      case <-number:
        fmt.Print(i)
        i++
        fmt.Print(i)
        i++
        letter <- true
      }
    }
  }()
  wait.Add(1)
  go func(wait *sync.WaitGroup) {
    i := 'A'
    for{
      select {
      case <-letter:
        if i >= 'Z' {
          wait.Done()
          return
        }

        fmt.Print(string(i))
        i++
        fmt.Print(string(i))
        i++
        number <- true
      }
```

```
    }
  }(&wait)
  number<-true
  wait.Wait()
```

源码解析

这里用到了两个`channel`负责通知，letter负责通知打印字母的goroutine来打印字母，number用来通知打印数字的goroutine打印数字。

wait用来等待字母打印完成后退出循环。

2. 判断字符串中字符是否全都不同

问题描述

请实现一个算法，确定一个字符串的所有字符【是否全都不同】。这里我们要求【不允许使用额外的存储结构】。
给定一个string，请返回一个bool值,true代表所有字符全都不同，false代表存在相同的字符。
保证字符串中的字符为【ASCII字符】。字符串的长度小于等于【3000】。

解题思路

这里有几个重点，第一个是`ASCII字符`，`ASCII字符`字符一共有256个，其中128个是常用字符，可以在键盘上输入。128之后的是键盘上无法找到的。

然后是全部不同，也就是字符串中的字符没有重复的，再次，不准使用额外的储存结构，且字符串小于等于3000。

如果允许其他额外储存结构，这个题目很好做。如果不允许的话，可以使用golang内置的方式实现。

源码参考

通过`strings.Count`函数判断：

```
func isUniqueString(s string) bool {
  if strings.Count(s,"") > 3000{
    return  false
  }
  for _,v := range s {
    if v > 127 {
      return false
    }
    if strings.Count(s,string(v)) > 1 {
      return false
    }
  }
  return true
}
```

通过`strings.Index`和`strings.LastIndex`函数判断：

```
func isUniqueString2(s string) bool {
  if strings.Count(s,"") > 3000{
    return  false
  }
  for k,v := range s {
    if v > 127 {
      return false
    }
    if strings.Index(s,string(v)) != k {
      return false
    }
  }
  return true
}
```

源码解析

以上两种方法都可以实现这个算法。

第一个方法使用的是golang内置方法`strings.Count`,可以用来判断在一个字符串中包含的另外一个字符串的数量。

第二个方法使用的是golang内置方法`strings.Index`和`strings.LastIndex`，用来判断指定字符串在另外一个字符串的索引未知，分别是第一次发现位置和最后发现位置。

3. 翻转字符串

问题描述

请实现一个算法，在不使用【额外数据结构和储存空间】的情况下，翻转一个给定的字符串(可以使用单个过程变量)。

给定一个string，请返回一个string，为翻转后的字符串。保证字符串的长度小于等于5000。

解题思路

翻转字符串其实是将一个字符串以中间字符为轴，前后翻转，即将str[len]赋值给str[0],将str[0] 赋值 str[len]。

源码参考

```
func reverString(s string) (string, bool) {
    str := []rune(s)
    l := len(str)
    if l > 5000 {
        return s, false
    }
    for i := 0; i < l/2; i++ {
        str[i], str[l-1-i] = str[l-1-i], str[i]
    }
    return string(str), true
}
```

源码解析

以字符串长度的1/2为轴，前后赋值。

4. 判断两个给定的字符串排序后是否一致

问题描述

给定两个字符串，请编写程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。
这里规定【大小写为不同字符】，且考虑字符串重点空格。给定一个string s1和一个string s2，请返回一个bool，代表两串是否重新排列后可相同。
保证两串的长度都小于等于5000。

解题思路

首先要保证字符串长度小于5000。之后只需要一次循环遍历s1中的字符在s2是否都存在即可。

源码参考

```
func isRegroup(s1,s2 string) bool {
  sl1 := len([]rune(s1))
  sl2 := len([]rune(s2))

  if sl1 > 5000 || sl2 > 5000 || sl1 != sl2{
    return false
  }

  for _,v := range s1 {
    if strings.Count(s1,string(v)) != strings.Count(s2,string(v)) {
      return false
    }
  }
  return true
}
```

源码解析

这里还是使用golang内置方法 `strings.Count` 来判断字符是否一致。

5. 字符串替换问题

问题描述

请编写一个方法，将字符串中的空格全部替换为“%20”。
假定该字符串有足够的空间存放新增的字符，并且知道字符串的真实长度(小于等于1000)，同时保证字符串由【大小写的英文字母组成】。
给定一个string为原始的串，返回替换后的string。

解题思路

两个问题，第一个是只能是英文字母，第二个是替换空格。

源码参考

```
func replaceBlank(s string) (string, bool) {
  if len([]rune(s)) > 1000 {
    return s, false
  }
  for _, v := range s {
    if string(v) != " " && unicode.IsLetter(v) == false {
      return s, false
    }
  }
  return strings.Replace(s, " ", "%20", -1), true
}
```

源码解析

这里使用了golang内置方法`unicode.IsLetter`判断字符是否是字母，之后使用`strings.Replace`来替换空格。

6. 机器人坐标问题

问题描述

有一个机器人，给一串指令，L左转 R右转，F前进一步，B后退一步，问最后机器人的坐标，最开始，机器人位于 0 0，方向为正Y。
可以输入重复指令n： 比如 R2(LF) 这个等于指令 RLFLF。
问最后机器人的坐标是多少?

解题思路

这里的一个难点是解析重复指令。主要指令解析成功，计算坐标就简单了。

源码参考

```
package main
```

```
import (
  "unicode"
)

const (
  Left = iota
  Top
  Right
  Bottom
)

func main() {
  println(move("R2(LF)", 0, 0, Top))
}

func move(cmd string, x0 int, y0 int, z0 int) (x, y, z int) {
  x, y, z = x0, y0, z0
  repeat := 0
  repeatCmd := ""
  for _, s := range cmd {
    switch {
    case unicode.IsNumber(s):
      repeat = repeat*10 + (int(s) - '0')
    case s == ')':
      for i := 0; i < repeat; i++ {
        x, y, z = move(repeatCmd, x, y, z)
      }
      repeat = 0
      repeatCmd = ""
    case repeat > 0 && s != '(' && s != ')':
      repeatCmd = repeatCmd + string(s)
    case s == 'L':
      z = (z + 1) % 4
    case s == 'R':
      z = (z - 1 + 4) % 4
    case s == 'F':
      switch {
      case z == Left || z == Right:
        x = x - z + 1
      case z == Top || z == Bottom:
        y = y - z + 2
      }
    case s == 'B':
      switch {
      case z == Left || z == Right:
        x = x + z - 1
      case z == Top || z == Bottom:
        y = y + z - 2
```

```
      }
    }
  }
  return
}
```

源码解析

这里使用三个值表示机器人当前的状况，分别是：x表示x坐标，y表示y坐标，z表示当前方向。
L、R 命令会改变值z，F、B命令会改变值x、y。
值x、y的改变还受当前的z值影响。

如果是重复指令，那么将重复次数和重复的指令存起来递归调用即可。

7. 下面代码能运行吗？为什么。

```
type Param map[string]interface{}

type Show struct {
  Param
}

func main1() {
  s := new(Show)
  s.Param["RMB"] = 10000
}
```

解析

共发现两个问题：

1. `main` 函数不能加数字。
2. `new` 关键字无法初始化 `Show` 结构体中的 `Param` 属性，所以直接对 `s.Param` 操作会出错。

8. 请说出下面代码存在什么问题。

```
type student struct {
  Name string
}

func zhoujielun(v interface{}) {
  switch msg := v.(type) {
  case *student, student:
    msg.Name
  }
}
```

解析：

golang中有规定，`switch type`的`case T1`，类型列表只有一个，那么`v := m.(type)`中的`v`的类型就是T1类型。

如果是`case T1, T2`，类型列表中有多个，那`v`的类型还是多对应接口的类型，也就是`m`的类型。

所以这里`msg`的类型还是`interface{}`，所以他没有`Name`这个字段，编译阶段就会报错。具体解释见： https://golang.org/ref/spec#Type_switches

9. 写出打印的结果。

```
type People struct {
  name string `json:"name"`
}

func main() {
  js := `{
    "name":"11"
  }`
  var p People
  err := json.Unmarshal([]byte(js), &p)
  if err != nil {
    fmt.Println("err: ", err)
    return
  }
  fmt.Println("people: ", p)
}
```

解析：

按照 golang 的语法，小写开头的方法、属性或 `struct` 是私有的，同样，在`json` 解码或转码的时候也无法上线私有属性的转换。

题目中是无法正常得到`People`的`name`值的。而且，私有属性`name`也不应该加`json`的标签。

10. 下面的代码是有问题的，请说明原因。

```
type People struct {
  Name string
}

func (p *People) String() string {
  return fmt.Sprintf("print: %v", p)
}

func main() {
  p := &People{}
  p.String()
}
```

解析：

在golang中`String() string`方法实际上是实现了`String`的接口的，该接口定义在`fmt/print.go`中：

```
type Stringer interface {
  String() string
}
```

在使用`fmt`包中的打印方法时，如果类型实现了这个接口，会直接调用。而题目中打印`p`的时候会直接调用`p`实现的`String()`方法，然后就产生了循环调用。

11. 请找出下面代码的问题所在。

```
func main() {
  ch := make(chan int, 1000)
  go func() {
    for i := 0; i < 10; i++ {
      ch <- i
    }
  }()
  go func() {
    for {
      a, ok := <-ch
      if !ok {
        fmt.Println("close")
        return
      }
      fmt.Println("a: ", a)
    }
  }()
  close(ch)
  fmt.Println("ok")
  time.Sleep(time.Second * 100)
}
```

解析：

在 golang 中`goroutine`的调度时间是不确定的，在题目中，第一个写`channel`的`goroutine`可能还未调用，或已调用但没有写完时直接`close`管道，可能导致写失败，既然出现`panic`错误。

12. 请说明下面代码书写是否正确。

```
var value int32

func SetValue(delta int32) {
  for {
    v := value
    if atomic.CompareAndSwapInt32(&value, v, (v+delta)) {
      break
    }
  }
}
```

解析：

`atomic.CompareAndSwapInt32` 函数不需要循环调用。

13. 下面的程序运行后为什么会爆异常。

```
type Project struct{}

func (p *Project) deferError() {
  if err := recover(); err != nil {
    fmt.Println("recover: ", err)
  }
}

func (p *Project) exec(msgchan chan interface{}) {
  for msg := range msgchan {
    m := msg.(int)
    fmt.Println("msg: ", m)
  }
}

func (p *Project) run(msgchan chan interface{}) {
  for {
    defer p.deferError()
    go p.exec(msgchan)
    time.Sleep(time.Second * 2)
  }
}

func (p *Project) Main() {
  a := make(chan interface{}, 100)
  go p.run(a)
  go func() {
    for {
      a <- "1"
      time.Sleep(time.Second)
    }
```

```
  }()
  time.Sleep(time.Second * 100000000000000)
}

func main() {
  p := new(Project)
  p.Main()
}
```

解析：

有一下几个问题：

1. `time.Sleep` 的参数数值太大，超过了 `1<<63 - 1` 的限制。
2. `defer p.deferError()` 需要在协程开始出调用，否则无法捕获 `panic`。

14. 请说出下面代码哪里写错了

```
func main() {
  abc := make(chan int, 1000)
  for i := 0; i < 10; i++ {
    abc <- i
  }
  go func() {
    for  a := range abc  {
      fmt.Println("a: ", a)
    }
  }()
  close(abc)
  fmt.Println("close")
  time.Sleep(time.Second * 100)
}
```

解析：

协程可能还未启动，管道就关闭了。

15. 请说出下面代码，执行时为什么会报错

```
type Student struct {
  name string
}

func main() {
  m := map[string]Student{"people": {"zhoujielun"}}
  m["people"].name = "wuyanzu"
}
```

解析：

map的value本身是不可寻址的，因为map中的值会在内存中移动，并且旧的指针地址在map改变时会变得无效。故如果需要修改map值，可以将`map`中的非指针类型`value`，修改为指针类型，比如使用`map[string]*Student`.

16. 请说出下面的代码存在什么问题?

```
type query func(string) string

func exec(name string, vs ...query) string {
  ch := make(chan string)
  fn := func(i int) {
    ch <- vs[i](name)
  }
  for i, _ := range vs {
    go fn(i)
  }
  return <-ch
}

func main() {
  ret := exec("111", func(n string) string {
    return n + "func1"
  }, func(n string) string {
    return n + "func2"
  }, func(n string) string {
    return n + "func3"
  }, func(n string) string {
    return n + "func4"
  })
  fmt.Println(ret)
}
```

解析：

依据4个goroutine的启动后执行效率，很可能打印111func4，但其他的111func*也可能先执行，exec只会返回一条信息。

17. 下面这段代码为什么会卡死?

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
```

```
    var i byte
    go func() {
        for i = 0; i <= 255; i++ {
        }
    }()
    fmt.Println("Dropping mic")
    // Yield execution to force executing other goroutines
    runtime.Gosched()
    runtime.GC()
    fmt.Println("Done")
}
```

解析：

Golang 中，byte 其实被 alias 到 uint8 上了。所以上面的 for 循环会始终成立，因为 i++ 到 i=255 的时候会溢出，i <= 255 一定成立。

也即是， for 循环永远无法退出，所以上面的代码其实可以等价于这样：

```
go func() {
    for {}
}
```

正在被执行的 goroutine 发生以下情况时让出当前 goroutine 的执行权，并调度后面的 goroutine 执行：

- IO 操作
- Channel 阻塞
- system call
- 运行较长时间

如果一个 goroutine 执行时间太长，scheduler 会在其 G 对象上打上一个标志（ preempt），当这个 goroutine 内部发生函数调用的时候，会先主动检查这个标志，如果为 true 则会让出执行权。

main 函数里启动的 goroutine 其实是一个没有 IO 阻塞、没有 Channel 阻塞、没有 system call、没有函数调用的死循环。

也就是，它无法主动让出自己的执行权，即使已经执行很长时间，scheduler 已经标志了 preempt。

而 golang 的 GC 动作是需要所有正在运行 `goroutine` 都停止后进行的。因此，程序会卡在 `runtime.GC()` 等待所有协程退出。

18. 写出下面代码输出内容。

```
package main

import (
  "fmt"
)

func main() {
  defer_call()
```

```
}

func defer_call() {
  defer func() { fmt.Println("打印前") }()
  defer func() { fmt.Println("打印中") }()
  defer func() { fmt.Println("打印后") }()

  panic("触发异常")
}
```

解析：

`defer` 关键字的实现跟go关键字很类似，不同的是它调用的是`runtime.deferproc`而不是`runtime.newproc`。

在`defer`出现的地方，插入了指令`call runtime.deferproc`，然后在函数返回之前的地方，插入指令`call runtime.deferreturn`。

goroutine的控制结构中，有一张表记录`defer`，调用`runtime.deferproc`时会将需要defer的表达式记录在表中，而在调用`runtime.deferreturn`的时候，则会依次从defer表中出栈并执行。

因此，题目最后输出顺序应该是`defer` 定义顺序的倒序。`panic` 错误并不能终止 `defer` 的执行。

19. 以下代码有什么问题，说明原因

```
type student struct {
  Name string
  Age  int
}

func pase_student() {
  m := make(map[string]*student)
  stus := []student{
    {Name: "zhou", Age: 24},
    {Name: "li", Age: 23},
    {Name: "wang", Age: 22},
  }
  for _, stu := range stus {
    m[stu.Name] = &stu
  }
}
```

解析：

golang 的 `for ... range` 语法中，`stu` 变量会被复用，每次循环会将集合中的值复制给这个变量，因此，会导致最后`m`中的`map`中储存的都是`stus`最后一个`student`的值。

20. 下面的代码会输出什么，并说明原因

```
func main() {
```

```
  runtime.GOMAXPROCS(1)
  wg := sync.WaitGroup{}
  wg.Add(20)
  for i := 0; i < 10; i++ {
    go func() {
      fmt.Println("i: ", i)
      wg.Done()
    }()
  }
  for i := 0; i < 10; i++ {
    go func(i int) {
      fmt.Println("i: ", i)
      wg.Done()
    }(i)
  }
  wg.Wait()
}
```

解析：

这个输出结果决定来自于调度器优先调度哪个G。从runtime的源码可以看到，当创建一个G时，会优先放入到下一个调度的`runnext`字段上作为下一次优先调度的G。因此，最先输出的是最后创建的G，也就是9.

```
func newproc(siz int32, fn *funcval) {
  argp := add(unsafe.Pointer(&fn), sys.PtrSize)
  gp := getg()
  pc := getcallerpc()
  systemstack(func() {
    newg := newproc1(fn, argp, siz, gp, pc)

    _p_ := getg().m.p.ptr()
        //新创建的G会调用这个方法来决定如何调度
    runqput(_p_, newg, true)

    if mainStarted {
      wakep()
    }
  })
}
...

  if next {
  retryNext:
    oldnext := _p_.runnext
        //当next是true时总会将新进来的G放入下一次调度字段中
    if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
      goto retryNext
    }
```

```
    if oldnext == 0 {
      return
    }
    // Kick the old runnext out to the regular run queue.
    gp = oldnext.ptr()
  }
```

21. 下面代码会输出什么?

```
type People struct{}

func (p *People) ShowA() {
  fmt.Println("showA")
  p.ShowB()
}
func (p *People) ShowB() {
  fmt.Println("showB")
}

type Teacher struct {
  People
}

func (t *Teacher) ShowB() {
  fmt.Println("teacher showB")
}

func main() {
  t := Teacher{}
  t.ShowA()
}
```

解析:

输出结果为`showA`、`showB`。golang 语言中没有继承概念，只有组合，也没有虚方法，更没有重载。因此，`*Teacher` 的 `ShowB` 不会覆写被组合的 `People` 的方法。

22. 下面代码会触发异常吗? 请详细说明

```
func main() {
  runtime.GOMAXPROCS(1)
  int_chan := make(chan int, 1)
  string_chan := make(chan string, 1)
  int_chan <- 1
  string_chan <- "hello"
  select {
  case value := <-int_chan:
    fmt.Println(value)
  case value := <-string_chan:
    panic(value)
  }
}
```

解析：

结果是随机执行。golang 在多个`case`可读的时候会公平的选中一个执行。

23. 下面代码输出什么?

```
func calc(index string, a, b int) int {
  ret := a + b
  fmt.Println(index, a, b, ret)
  return ret
}

func main() {
  a := 1
  b := 2
  defer calc("1", a, calc("10", a, b))
  a = 0
  defer calc("2", a, calc("20", a, b))
  b = 1
}
```

解析：

输出结果为：

```
10 1 2 3
20 0 2 2
2 0 2 2
1 1 3 4
```

`defer`在定义的时候会计算好调用函数的参数，所以会优先输出`10`、`20`两个参数。然后根据定义的顺序倒序执行。

24. 请写出以下输入内容

```go
func main() {
  s := make([]int, 5)
  s = append(s, 1, 2, 3)
  fmt.Println(s)
}
```

解析：

输出为 `0 0 0 0 0 1 2 3`。

`make` 在初始化切片时指定了长度，所以追加数据时会从`len(s)` 位置开始填充数据。

25. 下面的代码有什么问题?

```go
type UserAges struct {
  ages map[string]int
  sync.Mutex
}

func (ua *UserAges) Add(name string, age int) {
  ua.Lock()
  defer ua.Unlock()
  ua.ages[name] = age
}

func (ua *UserAges) Get(name string) int {
  if age, ok := ua.ages[name]; ok {
    return age
  }
  return -1
}
```

解析：

在执行 Get方法时可能被thorw。

虽然有使用sync.Mutex做写锁，但是map是并发读写不安全的。map属于引用类型，并发读写时多个协程见是通过指针访问同一个地址，即访问共享变量，此时同时读写资源存在竞争关系。会报错误信息:“fatal error: concurrent map read and map write”。

因此，在 `Get` 中也需要加锁，因为这里只是读，建议使用读写锁 `sync.RWMutex`。

26. 下面的迭代会有什么问题?

```go
func (set *threadSafeSet) Iter() <-chan interface{} {
  ch := make(chan interface{})
  go func() {
    set.RLock()
```

```
    for elem := range set.s {
      ch <- elem
    }

    close(ch)
    set.RUnlock()

  }()
  return ch
}
```

解析：

默认情况下 `make` 初始化的 `channel` 是无缓冲的，也就是在迭代写时会阻塞。

27. 以下代码能编译过去吗？为什么？

```
package main

import (
  "fmt"
)

type People interface {
  Speak(string) string
}

type Student struct{}

func (stu *Student) Speak(think string) (talk string) {
  if think == "bitch" {
    talk = "You are a good boy"
  } else {
    talk = "hi"
  }
  return
}

func main() {
  var peo People = Student{}
  think := "bitch"
  fmt.Println(peo.Speak(think))
}
```

解析：

编译失败，值类型 `Student{}` 未实现接口 `People` 的方法，不能定义为 `People` 类型。

在 golang 语言中，`Student` 和 `*Student` 是两种类型，第一个是表示 `Student` 本身，第二个是指向 `Student` 的指针。

28. 以下代码打印出来什么内容，说出为什么...

```
package main

import (
  "fmt"
)

type People interface {
  Show()
}

type Student struct{}

func (stu *Student) Show() {

}

func live() People {
  var stu *Student
  return stu
}

func main() {
  if live() == nil {
    fmt.Println("AAAAAAA")
  } else {
    fmt.Println("BBBBBBB")
  }
}
```

解析：

跟上一题一样，不同的是`*Student` 的定义后本身没有初始化值，所以 `*Student` 是 `nil`的，但是`*Student` 实现了 `People` 接口，接口不为 `nil`。

29. 在 golang 协程和 channel 配合使用

> 写代码实现两个 goroutine，其中一个产生随机数并写入到 go channel 中，另外一个从 channel 中读取数字并打印到标准输出。最终输出五个随机数。

解析

这是一道很简单的golang基础题目，实现方法也有很多种，一般想答让面试官满意的答案还是有几点注意的地方。

1. `goroutine` 在golang中式非阻塞的
2. `channel` 无缓冲情况下，读写都是阻塞的，且可以用`for`循环来读取数据，当管道关闭后，`for` 退出。

3. golang 中有专用的`select case`语法从管道读取数据。

示例代码如下：

```
func main() {
    out := make(chan int)
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < 5; i++ {
            out <- rand.Intn(5)
        }
        close(out)
    }()
    go func() {
        defer wg.Done()
        for i := range out {
            fmt.Println(i)
        }
    }()
    wg.Wait()
}
```

30. 实现阻塞读且并发安全的 map

GO里面MAP如何实现key不存在 get操作等待 直到key存在或者超时，保证并发安全，且需要实现以下接口：

```
type sp interface {
    Out(key string, val interface{})  //存入key /val，如果该key读取的goroutine挂起，则唤醒。此方法不会阻塞，时刻都可以立即执行并返回
    Rd(key string, timeout time.Duration) interface{}  //读取一个key，如果key不存在阻塞，等待key存在或者超时
}
```

解析：

看到阻塞协程第一个想到的就是`channel`，题目中要求并发安全，那么必须用锁，还要实现多个`goroutine`读的时候如果值不存在则阻塞，直到写入值，那么每个键值需要有一个阻塞`goroutine`的`channel`。

实现如下：

```
type Map struct {
  c   map[string]*entry
  rmx *sync.RWMutex
}
type entry struct {
  ch      chan struct{}
```

```
  value   interface{}
  isExist bool
}

func (m *Map) Out(key string, val interface{}) {
  m.rmx.Lock()
  defer m.rmx.Unlock()
  item, ok := m.c[key]
  if !ok {
    m.c[key] = &entry{
      value: val,
      isExist: true,
    }
    return
  }
  item.value = val
  if !item.isExist {
    if item.ch != nil {
      close(item.ch)
      item.ch = nil
    }
  }
  return
}
```

31. 高并发下的锁与 map 的读写

场景：在一个高并发的web服务器中，要限制IP的频繁访问。现模拟100个IP同时并发访问服务器，每个IP要重复访问1000次。

每个IP三分钟之内只能访问一次。修改以下代码完成该过程，要求能成功输出 success:100

```
package main

import (
  "fmt"
  "time"
)

type Ban struct {
  visitIPs map[string]time.Time
}

func NewBan() *Ban {
  return &Ban{visitIPs: make(map[string]time.Time)}
}
func (o *Ban) visit(ip string) bool {
  if _, ok := o.visitIPs[ip]; ok {
    return true
```

```
  }
  o.visitIPs[ip] = time.Now()
  return false
}
func main() {
  success := 0
  ban := NewBan()
  for i := 0; i < 1000; i++ {
    for j := 0; j < 100; j++ {
      go func() {
        ip := fmt.Sprintf("192.168.1.%d", j)
        if !ban.visit(ip) {
          success++
        }
      }()
    }

  }
  fmt.Println("success:", success)
}
```

解析

该问题主要考察了并发情况下map的读写问题，而给出的初始代码，又存在`for`循环中启动`goroutine`时变量使用问题以及`goroutine`执行滞后问题。

因此，首先要保证启动的`goroutine`得到的参数是正确的，然后保证`map`的并发读写，最后保证三分钟只能访问一次。

多CPU核心下修改`int`的值极端情况下会存在不同步情况，因此需要原子性的修改int值。

下面给出的实例代码，是启动了一个协程每分钟检查一下`map`中的过期`ip`，`for`启动协程时传参。

```
package main

import (
  "context"
  "fmt"
  "sync"
  "sync/atomic"
  "time"
)

type Ban struct {
  visitIPs map[string]time.Time
  lock      sync.Mutex
}

func NewBan(ctx context.Context) *Ban {
  o := &Ban{visitIPs: make(map[string]time.Time)}
```

```
  go func() {
    timer := time.NewTimer(time.Minute * 1)
    for {
      select {
      case <-timer.C:
        o.lock.Lock()
        for k, v := range o.visitIPs {
          if time.Now().Sub(v) >= time.Minute*1 {
            delete(o.visitIPs, k)
          }
        }
        o.lock.Unlock()
        timer.Reset(time.Minute * 1)
      case <-ctx.Done():
        return
      }
    }
  }()
  return o
}
func (o *Ban) visit(ip string) bool {
  o.lock.Lock()
  defer o.lock.Unlock()
  if _, ok := o.visitIPs[ip]; ok {
    return true
  }
  o.visitIPs[ip] = time.Now()
  return false
}
func main() {
  success := int64(0)
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()

  ban := NewBan(ctx)

  wait := &sync.WaitGroup{}

  wait.Add(1000 * 100)
  for i := 0; i < 1000; i++ {
    for j := 0; j < 100; j++ {
      go func(j int) {
        defer wait.Done()
        ip := fmt.Sprintf("192.168.1.%d", j)
        if !ban.visit(ip) {
          atomic.AddInt64(&success, 1)
        }
      }(j)
    }
```

```
  }
  wait.Wait()

  fmt.Println("success:", success)
}
```

32. 写出以下逻辑，要求每秒钟调用一次 proc 并保证程序不退出?

```
package main

func main() {
    go func() {
        // 1 在这里需要你写算法
        // 2 要求每秒钟调用一次proc函数
        // 3 要求程序不能退出
    }()

    select {}
}

func proc() {
    panic("ok")
}
```

解析

题目主要考察了两个知识点：

1. 定时执行执行任务
2. 捕获 panic 错误

题目中要求每秒钟执行一次，首先想到的就是 `time.Ticker` 对象，该函数可每秒钟往 `chan` 中放一个 `Time`,正好符合我们的要求。

在 `golang` 中捕获 `panic` 一般会用到 `recover()` 函数。

```
package main

import (
  "fmt"
  "time"
)

func main() {
  go func() {
    // 1 在这里需要你写算法
    // 2 要求每秒钟调用一次proc函数
```

```
    // 3 要求程序不能退出

    t := time.NewTicker(time.Second * 1)
    for {
      select {
      case <-t.C:
        go func() {
          defer func() {
            if err := recover(); err != nil {
              fmt.Println(err)
            }
          }()
          proc()
        }()
      }
    }
  }()

  select {}
}

func proc() {
  panic("ok")
}
```

33. 为 sync.WaitGroup 中 Wait 函数支持 WaitTimeout 功能.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    wg := sync.WaitGroup{}
    c := make(chan struct{})
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(num int, close <-chan struct{}) {
            defer wg.Done()
            <-close
            fmt.Println(num)
        }(i, c)
    }
```

```
    if WaitTimeout(&wg, time.Second*5) {
        close(c)
        fmt.Println("timeout exit")
    }
    time.Sleep(time.Second * 10)
}

func WaitTimeout(wg *sync.WaitGroup, timeout time.Duration) bool {
    // 要求手写代码
    // 要求sync.WaitGroup支持timeout功能
    // 如果timeout到了超时时间返回true
    // 如果WaitGroup自然结束返回false
}
```

解析

首先 `sync.WaitGroup` 对象的 `Wait` 函数本身是阻塞的，同时，超时用到的`time.Timer` 对象也需要阻塞的读。

同时阻塞的两个对象肯定要每个启动一个协程,每个协程去处理一个阻塞，难点在于怎么知道哪个阻塞先完成。

目前我用的方式是声明一个没有缓冲的`chan`，谁先完成谁优先向管道中写入数据。

```
package main

import (
  "fmt"
  "sync"
  "time"
)

func main() {
  wg := sync.WaitGroup{}
  c := make(chan struct{})
  for i := 0; i < 10; i++ {
    wg.Add(1)
    go func(num int, close <-chan struct{}) {
      defer wg.Done()
      <-close
      fmt.Println(num)
    }(i, c)
  }

  if WaitTimeout(&wg, time.Second*5) {
    close(c)
    fmt.Println("timeout exit")
  }
  time.Sleep(time.Second * 10)
}

func WaitTimeout(wg *sync.WaitGroup, timeout time.Duration) bool {
```

```
  // 要求手写代码
  // 要求sync.WaitGroup支持timeout功能
  // 如果timeout到了超时时间返回true
  // 如果WaitGroup自然结束返回false
  ch := make(chan bool, 1)

  go time.AfterFunc(timeout, func() {
    ch <- true
  })

  go func() {
    wg.Wait()
    ch <- false
  }()

  return <- ch
}
```

34. 写出以下代码出现的问题

```
package main
import (
    "fmt"
)
func main() {
    var x string = nil
    if x == nil {
        x = "default"
    }
    fmt.Println(x)
}
```

golang 中字符串是不能赋值 `nil` 的，也不能跟 `nil` 比较。

35. 写出以下打印内容

```
   package main
   import "fmt"
   const (
       a = iota
       b = iota
   )
   const (
       name = "menglu"
       c    = iota
       d    = iota
```

```
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
}
```

36. 找出下面代码的问题

```
package main
import "fmt"
type query func(string) string

func exec(name string, vs ...query) string {
    ch := make(chan string)
    fn := func(i int) {
        ch <- vs[i](name)
    }
    for i, _ := range vs {
        go fn(i)
    }
    return <-ch
}

func main() {
    ret := exec("111", func(n string) string {
        return n + "func1"
    }, func(n string) string {
        return n + "func2"
    }, func(n string) string {
        return n + "func3"
    }, func(n string) string {
        return n + "func4"
    })
    fmt.Println(ret)
}
```

上面的代码有严重的内存泄漏问题，出错的位置是 `go fn(i)`，实际上代码执行后会启动 4 个协程，但是因为 `ch` 是非缓冲的，只可能有一个协程写入成功。而其他三个协程会一直在后台等待写入。

37. 写出以下打印结果，并解释下为什么这么打印的。

```
package main
import (
    "fmt"
)
func main() {
    str1 := []string{"a", "b", "c"}
    str2 := str1[1:]
    str2[1] = "new"
    fmt.Println(str1)
    str2 = append(str2, "z", "x", "y")
    fmt.Println(str1)
}
```

golang 中的切片底层其实使用的是数组。当使用`str1[1:]`使，`str2`和`str1`底层共享一个数组，这回导致`str2[1] = "new"`语句影响`str1`。

而`append`会导致底层数组扩容，生成新的数组，因此追加数据后的`str2`不会影响`str1`。

但是为什么对`str2`复制后影响的确实`str1`的第三个元素呢？这是因为切片`str2`是从数组的第二个元素开始，`str2`索引为 1 的元素对应的是`str1`索引为 2 的元素。

38. 写出以下打印结果

```
package main

import (
    "fmt"
)

type Student struct {
    Name string
}

func main() {
    fmt.Println(&Student{Name: "menglu"} == &Student{Name: "menglu"})
    fmt.Println(Student{Name: "menglu"} == Student{Name: "menglu"})
}
```

个人理解：指针类型比较的是指针地址，非指针类型比较的是每个属性的值。

39. 写出以下代码的问题

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println([...]string{"1"} == [...]string{"1"})
    fmt.Println([]string{"1"} == []string{"1"})
}
```

数组只能与相同纬度长度以及类型的其他数组比较，切片之间不能直接比较。。

40. 下面代码写法有什么问题?

```
package main
import (
    "fmt"
)
type Student struct {
    Age int
}
func main() {
    kv := map[string]Student{"menglu": {Age: 21}}
    kv["menglu"].Age = 22
    s := []Student{{Age: 21}}
    s[0].Age = 22
    fmt.Println(kv, s)
}
```

golang中的`map`通过`key`获取到的实际上是两个值，第一个是获取到的值，第二个是是否存在该`key`。因此不能直接通过`key`来赋值对象。

41. Mutex

```
package main
import (
  "fmt"
  "sync"
)
var mu sync.Mutex
var chain string
func main() {
  chain = "main"
  A()
  fmt.Println(chain)
}
```

```
func A() {
  mu.Lock()
  defer mu.Unlock()
  chain = chain + " --> A"
  B()
}
func B() {
  chain = chain + " --> B"
  C()
}
func C() {
  mu.Lock()
  defer mu.Unlock()
  chain = chain + " --> C"
}
```

- A: 不能编译
- B: 输出 main --> A --> B --> C
- C: 输出 main
- D: panic

答案：D

会产生死锁`panic`，因为`Mutex`是互斥锁。

42. RWMutex

```
package main
import (
  "fmt"
  "sync"
  "time"
)
var mu sync.RWMutex
var count int
func main() {
  go A()
  time.Sleep(2 * time.Second)
  mu.Lock()
  defer mu.Unlock()
  count++
  fmt.Println(count)
}
func A() {
  mu.RLock()
  defer mu.RUnlock()
  B()
}
func B() {
```

```
  time.Sleep(5 * time.Second)
  C()
}
func C() {
  mu.RLock()
  defer mu.RUnlock()
}
```

- A: 不能编译
- B: 输出 1
- C: 程序hang住
- D: panic

答案：D

会产生死锁`panic`，根据`sync/rwmutex.go`中注释可以知道，读写锁当有一个协程在等待写锁时，其他协程是不能获得读锁的，而在`A`和`C`中同一个调用链中间需要让出读锁，让写锁优先获取，而`A`的读锁又要求`C`调用完成，因此死锁。

43. Waitgroup

```
package main
import (
  "sync"
  "time"
)
func main() {
  var wg sync.WaitGroup
  wg.Add(1)
  go func() {
    time.Sleep(time.Millisecond)
    wg.Done()
    wg.Add(1)
  }()
  wg.Wait()
}
```

- A: 不能编译
- B: 无输出，正常退出
- C: 程序hang住
- D: panic

答案：D

`WaitGroup`在调用`Wait`之后是不能再调用`Add`方法的。

44. 双检查实现单例

```
package doublecheck
```

```
import (
  "sync"
)
type Once struct {
  m    sync.Mutex
  done uint32
}
func (o *Once) Do(f func()) {
  if o.done == 1 {
    return
  }
  o.m.Lock()
  defer o.m.Unlock()
  if o.done == 0 {
    o.done = 1
    f()
  }
}
```

- A: 不能编译
- B: 可以编译，正确实现了单例
- C: 可以编译，有并发问题，f函数可能会被执行多次
- D: 可以编译，但是程序运行会panic

答案：C

在多核CPU中，因为CPU缓存会导致多个核心中变量值不同步。

45. Mutex

```
package main
import (
  "fmt"
  "sync"
)
type MyMutex struct {
  count int
  sync.Mutex
}
func main() {
  var mu MyMutex
  mu.Lock()
  var mu2 = mu
  mu.count++
  mu.Unlock()
  mu2.Lock()
  mu2.count++
  mu2.Unlock()
  fmt.Println(mu.count, mu2.count)
```

```
}
```

- A: 不能编译
- B: 输出 1, 1
- C: 输出 1, 2
- D: panic

答案：D

加锁后复制变量，会将锁的状态也复制，所以`mu1`其实是已经加锁状态，再加锁会死锁。

46. Pool

```
package main
import (
  "bytes"
  "fmt"
  "runtime"
  "sync"
  "time"
)
var pool = sync.Pool{New: func() interface{} { return new(bytes.Buffer) }}
func main() {
  go func() {
    for {
      processRequest(1 << 28) // 256MiB
    }
  }()
  for i := 0; i < 1000; i++ {
    go func() {
      for {
        processRequest(1 << 10) // 1KiB
      }
    }()
  }
  var stats runtime.MemStats
  for i := 0; ; i++ {
    runtime.ReadMemStats(&stats)
    fmt.Printf("Cycle %d: %dB\n", i, stats.Alloc)
    time.Sleep(time.Second)
    runtime.GC()
  }
}
func processRequest(size int) {
  b := pool.Get().(*bytes.Buffer)
  time.Sleep(500 * time.Millisecond)
  b.Grow(size)
  pool.Put(b)
  time.Sleep(1 * time.Millisecond)
```

```
}
```

- A: 不能编译
- B: 可以编译，运行时正常，内存稳定
- C: 可以编译，运行时内存可能暴涨
- D: 可以编译，运行时内存先暴涨，但是过一会会回收掉

答案：C

个人理解，在单核CPU中，内存可能会稳定在`256MB`，如果是多核可能会暴涨。

47. channel

```go
package main
import (
  "fmt"
  "runtime"
  "time"
)
func main() {
  var ch chan int
  go func() {
    ch = make(chan int, 1)
    ch <- 1
  }()
  go func(ch chan int) {
    time.Sleep(time.Second)
    <-ch
  }(ch)
  c := time.Tick(1 * time.Second)
  for range c {
    fmt.Printf("#goroutines: %d\n", runtime.NumGoroutine())
  }
}
```

- A: 不能编译
- B: 一段时间后总是输出 `#goroutines: 1`
- C: 一段时间后总是输出 `#goroutines: 2`
- D: panic

答案：C

因为 `ch` 未初始化，写和读都会阻塞，之后被第一个协程重新赋值，导致写的`ch` 都阻塞。

48. channel

```go
package main
import "fmt"
func main() {
```

```
  var ch chan int
  var count int
  go func() {
    ch <- 1
  }()
  go func() {
    count++
    close(ch)
  }()
  <-ch
  fmt.Println(count)
}
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

答案：D

`ch` 未有被初始化，关闭时会报错。

49. Map

```
package main
import (
  "fmt"
  "sync"
)
func main() {
  var m sync.Map
  m.LoadOrStore("a", 1)
  m.Delete("a")
  fmt.Println(m.Len())
}
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

答案：A

sync.Map`没有`Len`方法。

50. happens before

```
package main
var c = make(chan int)
var a int
func f() {
  a = 1
  <-c
}
func main() {
  go f()
  c <- 0
  print(a)
}
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

答案：B

`c <- 0` 会阻塞依赖于 `f()` 的执行。

51. 对已经关闭的的chan进行读写，会怎么样？为什么？

回答

- 读已经关闭的 chan 能一直读到东西，但是读到的内容根据通道内关闭前是否有元素而不同。
  - 如果 chan 关闭前，buffer 内有元素还未读 , 会正确读到 chan 内的值，且返回的第二个 bool 值（是否读成功）为 true。
  - 如果 chan 关闭前，buffer 内有元素已经被读完，chan 内无值，接下来所有接收的值都会非阻塞直接成功，返回 channel 元素的零值，但是第二个 bool 值一直为 false。
- 写已经关闭的 chan 会 panic

示例

1. 写已经关闭的 chan

```
func main(){
    c := make(chan int,3)
    close(c)
    c <- 1
}
//输出结果
panic: send on closed channel

goroutine 1 [running]
main.main()
...
```

- 注意这个 send on closed channel，待会会提到。

2. 读已经关闭的 chan

```
package main
import "fmt"

func main()  {
    fmt.Println("以下是数值的chan")
    ci:=make(chan int,3)
    ci<-1
    close(ci)
    num,ok := <- ci
    fmt.Printf("读chan的协程结束，num=%v，  ok=%v\n",num,ok)
    num1,ok1 := <-ci
    fmt.Printf("再读chan的协程结束，num=%v，  ok=%v\n",num1,ok1)
    num2,ok2 := <-ci
    fmt.Printf("再再读chan的协程结束，num=%v，  ok=%v\n",num2,ok2)

    fmt.Println("以下是字符串chan")
    cs := make(chan string,3)
    cs <- "aaa"
    close(cs)
    str,ok := <- cs
    fmt.Printf("读chan的协程结束，str=%v，  ok=%v\n",str,ok)
    str1,ok1 := <-cs
    fmt.Printf("再读chan的协程结束，str=%v，  ok=%v\n",str1,ok1)
    str2,ok2 := <-cs
    fmt.Printf("再再读chan的协程结束，str=%v，  ok=%v\n",str2,ok2)

    fmt.Println("以下是结构体chan")
    type MyStruct struct{
        Name string
    }
    cstruct := make(chan MyStruct,3)
    cstruct <- MyStruct{Name: "haha"}
    close(cstruct)
    stru,ok := <- cstruct
    fmt.Printf("读chan的协程结束，stru=%v，  ok=%v\n",stru,ok)
    stru1,ok1 := <-cs
    fmt.Printf("再读chan的协程结束，stru=%v，  ok=%v\n",stru1,ok1)
    stru2,ok2 := <-cs
    fmt.Printf("再再读chan的协程结束，stru=%v，  ok=%v\n",stru2,ok2)
}
```

输出结果

```
以下是数值的chan
读chan的协程结束, num=1,  ok=true
再读chan的协程结束, num=0,  ok=false
再再读chan的协程结束, num=0,  ok=false
以下是字符串chan
读chan的协程结束, str=aaa,  ok=true
再读chan的协程结束, str=,  ok=false
再再读chan的协程结束, str=,  ok=false
以下是结构体chan
读chan的协程结束, stru={haha},  ok=true
再读chan的协程结束, stru=,  ok=false
再再读chan的协程结束, stru=,  ok=false
```

3. 为什么写已经关闭的 `chan` 就会 `panic` 呢?

```
//在 src/runtime/chan.go
func chansend(c *hchan,ep unsafe.Pointer,block bool,callerpc uintptr) bool {
    //省略其他
    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("send on closed channel"))
    }
    //省略其他
}
```

- 当 `c.closed != 0` 则为通道关闭，此时执行写，源码提示直接 `panic`，输出的内容就是上面提到的 `"send on closed channel"`。

4. 为什么读已关闭的 chan 会一直能读到值?

```
func chanrecv(c *hchan,ep unsafe.Pointer,block bool) (selected,received bool) {
    //省略部分逻辑
    lock(&c.lock)
    //当chan被关闭了，而且缓存为空时
    //ep 是指 val,ok := <-c 里的val地址
    if c.closed != 0 && c.qcount == 0 {
        if receenabled {
            raceacquire(c.raceaddr())
        }
        unlock(&c.lock)
        //如果接受之的地址不空，那接收值将获得一个该值类型的零值
        //typedmemclr 会根据类型清理响应的内存
        //这就解释了上面代码为什么关闭的chan 会返回对应类型的零值
        if ep != null {
            typedmemclr(c.elemtype,ep)
```

```
        }
        //返回两个参数 selected,received
        // 第二个采纳数就是 val,ok := <- c 里的 ok
        //也就解释了为什么读关闭的chan会一直返回false
        return true,false
    }
}
```

- `c.closed != 0 && c.qcount == 0` 指通道已经关闭，且缓存为空的情况下（已经读完了之前写到通道里的值）
- 如果接收值的地址 `ep` 不为空
  - 那接收值将获得是一个该类型的零值
  - `typedmemclr` 会根据类型清理相应地址的内存
  - 这就解释了上面代码为什么关闭的 chan 会返回对应类型的零值

52. 下面哪一行代码会 panic，请说明。

```
func main() {
    nil := 123
    fmt.Println(nil)
    var _ map[string]int = nil
}
```

答：第 4 行

解析：

当前作用域中，预定义的 nil 被覆盖，此时 nil 是 int 类型值，不能赋值给 map 类型。

53. 下面代码输出什么?

```
func main() {
    var x int8 = -128
    var y = x/-1
    fmt.Println(y)
}
```

答：-128

解析：

溢出

54. 字符串转成 byte 数组，会发生内存拷贝吗?

回答

字符串转成切片，会产生拷贝。严格来说，只要是发生类型强转都会发生内存拷贝。那么问题来了。

频繁的内存拷贝操作听起来对性能不大友好。有没有什么办法可以在字符串转成切片的时候不用发生拷贝呢?

解释

```
package main

import (
 "fmt"
 "reflect"
 "unsafe"
)

func main() {
 a :="aaa"
 ssh := *(*reflect.StringHeader)(unsafe.Pointer(&a))
 b := *(*[]byte)(unsafe.Pointer(&ssh))
 fmt.Printf("%v",b)
}
```

`StringHeader` 是字符串在go的底层结构。

```
type StringHeader struct {
 Data uintptr
 Len  int
}
```

`SliceHeader` 是切片在go的底层结构。

```
type SliceHeader struct {
 Data uintptr
 Len  int
 Cap  int
}
```

那么如果想要在底层转换二者，只需要把 StringHeader 的地址强转成 SliceHeader 就行。那么go有个很强的包叫 unsafe 。

1. `unsafe.Pointer(&a)` 方法可以得到变量a的地址。
2. `(*reflect.StringHeader)(unsafe.Pointer(&a))` 可以把字符串a转成底层结构的形式。
3. `(*[]byte)(unsafe.Pointer(&ssh))` 可以把ssh底层结构体转成byte的切片的指针。
4. 再通过 `*` 转为指针指向的实际内容。

55. sync.Map 的用法

回答

```
package main

import (
  "fmt"
  "sync"
)

func main(){
  var m sync.Map
  m.Store("address",map[string]string{"province":"江苏","city":"南京"})
        v,_ := m.Load("address")
  fmt.Println(v["province"])
}
```

- A，江苏；
- B，`v["province"]`取值错误；
- C，`m.Store`存储错误；
- D，不知道

解析

`invalid operation: v["province"] (type interface {} does not support indexing)`
因为`func (m *Map) Store(key interface{}, value interface{})`
所以`v`类型是`interface {}`，这里需要一个类型断言

```
fmt.Println(v.(map[string]string)["province"]) //江苏
```

56. 下面代码输出什么?

```
func main() {
    x := []string{"a", "b", "c"}
    for v := range x {
        fmt.Print(v)
    }
}
```

答：012

解析：

注意区别下面代码段：

```
func main() {
    x := []string{"a", "b", "c"}
    for _, v := range x {
        fmt.Print(v)     //输出 abc
    }
}
```

57. 下面这段代码能否编译通过？如果通过，输出什么？

```
type User struct{}
type User1 User
type User2 = User

func (i User) m1() {
    fmt.Println("m1")
}
func (i User) m2() {
    fmt.Println("m2")
}

func main() {
    var i1 User1
    var i2 User2
    i1.m1()
    i2.m2()
}
```

答：不能，报错`i1.m1 undefined (type User1 has no field or method m1)`

解析：

第 2 行代码基于类型 User 创建了新类型 User1，第 3 行代码是创建了 User 的类型别名 User2，注意使用 = 定义类型别名。因为 User2 是别名，完全等价于 User，所以 User2 具有 User 所有的方法。但是 i1.m1() 是不能执行的，因为 User1 没有定义该方法。

58. 关于无缓冲和有冲突的channel，下面说法正确的是?

- A. 无缓冲的channel是默认的缓冲为1的channel；
- B. 无缓冲的channel和有缓冲的channel都是同步的；
- C. 无缓冲的channel和有缓冲的channel都是非同步的；
- D. 无缓冲的channel是同步的，而有缓冲的channel是非同步的；

答：D

59. 下面代码是否能编译通过？如果通过，输出什么？

```
func Foo(x interface{}) {
     if x == nil {
         fmt.Println("empty interface")
         return
     }
     fmt.Println("non-empty interface")
 }
 func main() {
     var x *int = nil
    Foo(x)
}
```

答：non-empty interface

解析：

考点：interface 的内部结构，我们知道接口除了有静态类型，还有动态类型和动态值，当且仅当动态值和动态类型都为 nil 时，接口类型值才为 nil。这里的 x 的动态类型是 `*int`，所以 x 不为 nil。

60. 下面代码输出什么?

```
func main() {
     ch := make(chan int, 100)
     // A
     go func() {
         for i := 0; i < 10; i++ {
             ch <- i
         }
     }()
     // B
    go func() {
        for {
            a, ok := <-ch
            if !ok {
                fmt.Println("close")
                return
            }
            fmt.Println("a: ", a)
        }
    }()
    close(ch)
    fmt.Println("ok")
    time.Sleep(time.Second * 10)
}
```

答：程序抛异常

解析：

先定义下，第一个协程为 A 协程，第二个协程为 B 协程；当 A 协程还没起时，主协程已经将 channel 关闭了，当 A 协程往关闭的 channel 发送数据时会 panic，`panic: send on closed channel`。

61. 关于select机制，下面说法正确的是?

- A. select机制用来处理异步IO问题；
- B. select机制最大的一条限制就是每个case语句里必须是一个IO操作；
- C. golang在语言级别支持select关键字；
- D. select关键字的用法与switch语句非常类似，后面要带判断条件；

答：A B C

62. 下面的代码有什么问题?

```
func Stop(stop <-chan bool) {
    close(stop)
}
```

答：有方向的 channel 不可以被关闭。

63. 下面这段代码存在什么问题?

```
type Param map[string]interface{}

type Show struct {
    *Param
}

func main() {
    s := new(Show)
    s.Param["day"] = 2
}
```

答：存在两个问题

解析：

1. map 需要初始化才能使用；
2. 指针不支持索引。修复代码如下：

```
func main() {
    s := new(Show)
    // 修复代码
    p := make(Param)
    p["day"] = 2
    s.Param = &p
    tmp := *s.Param
    fmt.Println(tmp["day"])
}
```

64. 下面代码编译能通过吗?

```
func main()
{
    fmt.Println("hello world")
}
```

答：编译错误

```
syntax error: unexpected semicolon or newline before {
```

解析：

Go 语言中，大括号不能放在单独的一行。

正确的代码如下：

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

65. 下面这段代码输出什么?

```
var x = []int{2: 2, 3, 0: 1}

func main() {
    fmt.Println(x)
}
```

答：[1 0 2 3]

解析：

字面量初始化切片时候，可以指定索引，没有指定索引的元素会在前一个索引基础之上加一，所以输出`[1 0 2 3]`，而不是`[1 3 2]`。

66. 下面这段代码输出什么?

```
func incr(p *int) int {
    *p++
    return *p
}
func main() {
    v := 1
    incr(&v)
    fmt.Println(v)
}
```

答：2

解析：

知识点：指针。

p 是指针变量，指向变量 v，`*p++`操作的意思是取出变量 v 的值并执行加一操作，所以 v 的最终值是 2。

67. 请指出下面代码的错误?

```
package main

var gvar int

func main() {
    var one int
    two := 2
    var three int
    three = 3

    func(unused string) {
        fmt.Println("Unused arg. No compile error")
    }("what?")
}
```

答：变量 one、two 和 three 声明未使用

解析：

知识点：**未使用变量**。

如果有未使用的变量代码将编译失败。但也有例外，函数中声明的变量必须要使用，但可以有未使用的全局变量。函数的参数未使用也是可以的。

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

修复代码：

```
func main() {
    var one int
```

```
    _ = one

    two := 2
    fmt.Println(two)

    var three int
    three = 3
    one = three

    var four int
    four = four
}
```

另一个选择是注释掉或者移除未使用的变量 。

68. 下面代码输出什么?

```
type ConfigOne struct {
    Daemon string
}

func (c *ConfigOne) String() string {
    return fmt.Sprintf("print: %v", c)
}

func main() {
    c := &ConfigOne{}
    c.String()
}
```

答：运行时错误

解析：

如果类型实现 String() 方法，当格式化输出时会自动使用 String() 方法。上面这段代码是在该类型的 String() 方法内使用格式化输出，导致递归调用，最后抛错。

```
runtime: goroutine stack exceeds 1000000000-byte limit
fatal error: stack overflow
```

69. 下面代码输出什么?

```
func main() {
    var a = []int{1, 2, 3, 4, 5}
    var r = make([]int, 0)

    for i, v := range a {
        if i == 0 {
```

```
            a = append(a, 6, 7)
        }

        r = append(r, v)
    }

    fmt.Println(r)
}
```

答：[1 2 3 4 5]

解析：

a 在 for range 过程中增加了两个元素，len 由 5 增加到 7，但 for range 时会使用 a 的副本 a' 参与循环，副本的 len 依旧是 5，因此 for range 只会循环 5 次，也就只获取 a 对应的底层数组的前 5 个元素。

70. 下面的代码有什么问题?

```
import (
    "fmt"
    "log"
    "time"
)
func main() {
}
```

答：导入的包没有被使用

解析：

如果引入一个包，但是未使用其中如何函数、接口、结构体或变量的话，代码将编译失败。

如果你真的需要引入包，可以使用下划线操作符，`_`，来作为这个包的名字，从而避免失败。下划线操作符用于引入，但不使用。

我们还可以注释或者移除未使用的包。

修复代码：

```
import (
    _ "fmt"
    "log"
    "time"
)
var _ = log.Println
func main() {
    _ = time.Now
}
```

71. 下面代码输出什么?

```
func main() {
    x := interface{}(nil)
    y := (*int)(nil)
    a := y == x
    b := y == nil
    _, c := x.(interface{})
    println(a, b, c)
}
```

- A. true true true
- B. false true true
- C. true true true
- D. false true false

答：D

解析：

知识点：类型断言。

类型断言语法：i.(Type)，其中 i 是接口，Type 是类型或接口。编译时会自动检测 i 的动态类型与 Type 是否一致。但是，如果动态类型不存在，则断言总是失败。

72. 下面代码有几处错误的地方？请说明原因。

```
func main() {
    var s []int
    s = append(s,1)

    var m map[string]int
    m["one"] = 1
}
```

答：有 1 处错误

解析：

有 1 处错误，不能对 nil 的 map 直接赋值，需要使用 make() 初始化。但可以使用 append() 函数对为 nil 的 slice 增加元素。

修复代码：

```
func main() {
    var m map[string]int
    m = make(map[string]int)
    m["one"] = 1
}
```

73. 下面代码有什么问题?

```
func main() {
    m := make(map[string]int,2)
    cap(m)
}
```

答：使用 cap() 获取 map 的容量

解析：

1. 使用 make 创建 map 变量时可以指定第二个参数，不过会被忽略。
2. cap() 函数适用于数组、数组指针、slice 和 channel，不适用于 map，可以使用 len() 返回 map 的元素个数。

74. 下面的代码有什么问题?

```
func main() {
    var x = nil
    _ = x
}
```

解析：

nil 用于表示 interface、函数、maps、slices 和 channels 的“零值”。如果不指定变量的类型，编译器猜不出变量的具体类型，导致编译错误。

修复代码：

```
func main() {
    var x interface{} = nil
    _ = x
}
```

75. 下面代码能编译通过吗?

```
type info struct {
    result int
}

func work() (int,error) {
    return 13,nil
}

func main() {
    var data info

    data.result, err := work()
    fmt.Printf("info: %+v\n",data)
}
```

答：编译失败

```
non-name data.result on left side of :=
```

解析：

不能使用短变量声明设置结构体字段值，修复代码：

```
func main() {
    var data info

    var err error
    data.result, err = work() //ok
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(data)
}
```

76. 下面代码有什么错误?

```
func main() {
    one := 0
    one := 1
}
```

答：变量重复声明

解析：

不能在单独的声明中重复声明一个变量，但在多变量声明的时候是可以的，但必须保证至少有一个变量是新声明的。

修复代码：

```
func main() {
    one := 0
    one, two := 1,2
    one,two = two,one
}
```

77. 下面代码有什么问题?

```
func main() {
    x := []int{
        1,
        2
    }
    _ = x
}
```

答：编译错误

解析：

第四行代码没有逗号。用字面量初始化数组、slice 和 map 时，最好是在每个元素后面加上逗号，即使是声明在一行或者多行都不会出错。

修复代码：

```
func main() {
    x := []int{    // 多行
        1,
        2,
    }
    x = x

    y := []int{3,4,} // 一行 no error
    y = y
}
```

78. 下面代码输出什么?

```
func test(x byte)  {
    fmt.Println(x)
}

func main() {
    var a byte = 0x11
    var b uint8 = a
    var c uint8 = a + b
    test(c)
}
```

答：34

解析：

与 rune 是 int32 的别名一样，byte 是 uint8 的别名，别名类型无序转换，可直接转换。

79. 下面的代码有什么问题?

```
func main() {
    const x = 123
    const y = 1.23
    fmt.Println(x)
}
```

答：编译可以通过

解析：

知识点：常量。

常量是一个简单值的标识符，在程序运行时，不会被修改的量。不像变量，常量未使用是能编译通过的。

80. 下面代码输出什么?

```
const (
    x uint16 = 120
    y
    s = "abc"
    z
)

func main() {
    fmt.Printf("%T %v\n", y, y)
    fmt.Printf("%T %v\n", z, z)
}
```

答：

```
uint16 120
string abc
```

解析：

常量组中如不指定类型和初始化值，则与上一行非空常量右值相同

81. 下面代码有什么问题?

```
func main() {
    var x string = nil

    if x == nil {
        x = "default"
    }
}
```

答：将 nil 分配给 string 类型的变量

解析：

修复代码：

```
func main() {
    var x string //defaults to "" (zero value)

    if x == "" {
        x = "default"
    }
}
```

82. 下面的代码有什么问题?

```
func main() {
    data := []int{1,2,3}
    i := 0
    ++i
    fmt.Println(data[i++])
}
```

解析：

对于自增、自减，需要注意：

- 自增、自减不在是运算符，只能作为独立语句，而不是表达式；
- 不像其他语言，Go 语言中不支持 ++i 和 --i 操作；

表达式通常是求值代码，可作为右值或参数使用。而语句表示完成一个任务，比如 if、for 语句等。表达式可作为语句使用，但语句不能当做表达式。

修复代码：

```
func main() {
    data := []int{1,2,3}
    i := 0
    i++
    fmt.Println(data[i])
}
```

83. 下面代码最后一行输出什么？请说明原因。

```
func main() {
    x := 1
    fmt.Println(x)
    {
        fmt.Println(x)
        i,x := 2,2
        fmt.Println(i,x)
    }
    fmt.Println(x)  // print ?
}
```

答：输出`1`

解析：

知识点：变量隐藏。

使用变量简短声明符号 := 时，如果符号左边有多个变量，只需要保证至少有一个变量是新声明的，并对已定义的变量尽进行赋值操作。但如果出现作用域之后，就会导致变量隐藏的问题，就像这个例子一样。

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

84. 下面代码有什么问题?

```
type foo struct {
    bar int
}

func main() {
    var f foo
    f.bar, tmp := 1, 2
}
```

答：编译错误

```
non-name f.bar on left side of :=
```

解析：

`:=` 操作符不能用于结构体字段赋值。

85. 下面的代码输出什么?

```
func main() {
    fmt.Println(~2)
}
```

答：编译错误

```
invalid character U+007E '~'
```

解析：

很多语言都是采用 `~` 作为按位取反运算符，Go 里面采用的是 `^` 。按位取反之后返回一个每个 bit 位都取反的数，对于有符号的整数来说，是按照补码进行取反操作的（快速计算方法：对数 a 取反，结果为 -(a+1) ），对于无符号整数来说就是按位取反。例如：

```
func main() {
    var a int8 = 3
    var b uint8 = 3
    var c int8 = -3

    fmt.Printf("^%b=%b %d\n", a, ^a, ^a) // ^11=-100 -4
    fmt.Printf("^%b=%b %d\n", b, ^b, ^b) // ^11=11111100 252
    fmt.Printf("^%b=%b %d\n", c, ^c, ^c) // ^-11=10 2
}
```

另外需要注意的是，如果作为二元运算符，^ 表示按位异或，即：对应位相同为 0，相异为 1。例如：

```
func main() {
    var a int8 = 3
    var c int8 = 5

    fmt.Printf("a: %08b\n",a)
    fmt.Printf("c: %08b\n",c)
    fmt.Printf("a^c: %08b\n",a ^ c)
}
```

给大家重点介绍下这个操作符 &^，按位置零，例如：z = x &^ y，表示如果 y 中的 bit 位为 1，则 z 对应 bit 位为 0，否则 z 对应 bit 位等于 x 中相应的 bit 位的值。

不知道大家发现没有，我们还可以这样理解或操作符 | ，表达式 z = x | y，如果 y 中的 bit 位为 1，则 z 对应 bit 位为 1，否则 z 对应 bit 位等于 x 中相应的 bit 位的值，与 &^ 完全相反。

```
var x uint8 = 214
var y uint8 = 92
fmt.Printf("x: %08b\n",x)
fmt.Printf("y: %08b\n",y)
fmt.Printf("x | y: %08b\n",x | y)
fmt.Printf("x &^ y: %08b\n",x &^ y)
```

输出：

```
x: 11010110
y: 01011100
x | y: 11011110
x &^ y: 10000010
```

86. 下面代码输出什么?

```
func main() {
    var ch chan int
    select {
    case v, ok := <-ch:
        println(v, ok)
    default:
        println("default")
    }
}
```

答：default

解析：

ch 为 nil，读写都会阻塞。

87. 下面这段代码输出什么?

```
type People struct {
    name string `json:"name"`
}

func main() {
    js := `{
        "name":"seekload"
    }`
    var p People
    err := json.Unmarshal([]byte(js), &p)
    if err != nil {
        fmt.Println("err: ", err)
        return
    }
    fmt.Println(p)
}
```

答：输出 {}

解析：

知识点：结构体访问控制，因为 name 首字母是小写，导致其他包不能访问，所以输出为空结构体。

修复代码：

```
type People struct {
    Name string `json:"name"`
}
```

88. 下面这段代码输出什么？

```
type T struct {
    ls []int
}

func foo(t T) {
    t.ls[0] = 100
}

func main() {
    var t = T{
        ls: []int{1, 2, 3},
    }

    foo(t)
    fmt.Println(t.ls[0])
}
```

- A. 1
- B. 100
- C. compilation error

答：输出 B

解析：

调用 foo() 函数时虽然是传值，但 foo() 函数中，字段 ls 依旧可以看成是指向底层数组的指针。

89. 下面代码输出什么？

```
func main() {
    isMatch := func(i int) bool {
        switch(i) {
        case 1:
        case 2:
            return true
        }
        return false
    }

    fmt.Println(isMatch(1))
    fmt.Println(isMatch(2))
}
```

答：false true

解析：

Go 语言的 switch 语句虽然没有"break"，但如果 case 完成程序会默认 break，可以在 case 语句后面加上关键字 fallthrough，这样就会接着走下一个 case 语句（不用匹配后续条件表达式）。或者，利用 case 可以匹配多个值的特性。

修复代码：

```
func main() {
    isMatch := func(i int) bool {
        switch(i) {
        case 1:
            fallthrough
        case 2:
            return true
        }
        return false
    }

    fmt.Println(isMatch(1))     // true
    fmt.Println(isMatch(2))     // true

    match := func(i int) bool {
        switch(i) {
        case 1,2:
            return true
        }
        return false
    }

    fmt.Println(match(1))       // true
    fmt.Println(match(2))       // true
}
```

90. 下面的代码能否正确输出?

```
func main() {
    var fn1 = func() {}
    var fn2 = func() {}

    if fn1 != fn2 {
        println("fn1 not equal fn2")
    }
}
```

答：编译错误

```
invalid operation: fn1 != fn2 (func can only be compared to nil)
```

解析：

函数只能与 nil 比较。

91. 下面代码输出什么?

```
type T struct {
    n int
}

func main() {
    m := make(map[int]T)
    m[0].n = 1
    fmt.Println(m[0].n)
}
```

- A. 1
- B. compilation error

答：B

```
cannot assign to struct field m[0].n in map
```

解析：

map[key]struct 中 struct 是不可寻址的，所以无法直接赋值。

修复代码：

```
type T struct {
    n int
}

func main() {
    m := make(map[int]T)

    t := T{1}
    m[0] = t
    fmt.Println(m[0].n)
}
```

92. 下面的代码有什么问题?

```
type X struct {}

func (x *X) test()  {
    println(x)
}

func main() {

    var a *X
    a.test()

    X{}.test()
}
```

答：X{} 是不可寻址的，不能直接调用方法

解析：

知识点：在方法中，指针类型的接收者必须是合法指针（包括 nil）,或能获取实例地址。

修复代码：

```
func main() {

    var a *X
    a.test()    // 相当于 test(nil)

    var x = X{}
    x.test()
}
```

93. 下面代码有什么不规范的地方吗?

```
func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}

    if v := x["two"]; v == "" {
        fmt.Println("no entry")
    }
}
```

解析：

检查 map 是否含有某一元素，直接判断元素的值并不是一种合适的方式。最可靠的操作是使用访问 map 时返回的第二个值。

修复代码如下：

```
func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}

    if _,ok := x["two"]; !ok {
        fmt.Println("no entry")
    }
}
```

94. 关于 channel 下面描述正确的是?

- A. 向已关闭的通道发送数据会引发 panic；
- B. 从已关闭的缓冲通道接收数据，返回已缓冲数据或者零值；
- C. 无论接收还是接收，nil 通道都会阻塞；

答：A B C

95. 下面的代码有几处问题？请详细说明。

```
type T struct {
    n int
}

func (t *T) Set(n int) {
    t.n = n
}

func getT() T {
    return T{}
}

func main() {
    getT().Set(1)
}
```

答：有两处问题

解析：

- 1.直接返回的 T{} 不可寻址；
- 2.不可寻址的结构体不能调用带结构体指针接收者的方法；

修复代码：

```
type T struct {
    n int
}

func (t *T) Set(n int) {
    t.n = n
}

func getT() T {
    return T{}
}

func main() {
    t := getT()
    t.Set(2)
    fmt.Println(t.n)
}
```

96. 下面的代码有什么问题?

```
type N int

func (n N) value(){
    n++
    fmt.Printf("v:%p,%v\n",&n,n)
}

func (n *N) pointer(){
    *n++
    fmt.Printf("v:%p,%v\n",n,*n)
}


func main() {

    var a N = 25

    p := &a
    p1 := &p
```

```
    p1.value()
    p1.pointer()
}
```

答：编译错误

```
calling method value with receiver p1 (type **N) requires explicit dereference
calling method pointer with receiver p1 (type **N) requires explicit dereference
```

解析：

不能使用多级指针调用方法。

97. 下面的代码输出什么？

```
type N int

func (n N) test(){
    fmt.Println(n)
}

func main()  {
    var n N = 10
    fmt.Println(n)

    n++
    f1 := N.test
    f1(n)

    n++
    f2 := (*N).test
    f2(&n)
}
```

答：10 11 12

解析：

知识点：方法表达式。

通过类型引用的方法表达式会被还原成普通函数样式，接收者是第一个参数，调用时显示传参。类型可以是 T 或 *T，只要目标方法存在于该类型的方法集中就可以。

还可以直接使用方法表达式调用：

```
func main()  {
    var n N = 10

    fmt.Println(n)

    n++
    N.test(n)

    n++
    (*N).test(&n)
}
```

98. 关于 channel 下面描述正确的是?

- A. close() 可以用于只接收通道；
- B. 单向通道可以转换为双向通道；
- C. 不能在单向通道上做逆向操作（例如：只发送通道用于接收）；

答：C

99. 下面的代码有什么问题?

```
type T struct {
    n int
}

func getT() T {
    return T{}
}

func main() {
    getT().n = 1
}
```

答：编译错误

```
cannot assign to getT().n
```

解析：

直接返回的 T{} 无法寻址，不可直接赋值。

修复代码：

```
type T struct {
    n int
}
```

```
func getT() T {
    return T{}
}

func main() {
    t := getT()
    p := &t.n    // <=> p = &(t.n)
    *p = 1
    fmt.Println(t.n)
}
```

100. 下面的代码有什么问题?

```
package main

import "fmt"

func main() {
    s := make([]int, 3, 9)
    fmt.Println(len(s))
    s2 := s[4:8]
    fmt.Println(len(s2))
}
```

答：代码没问题，输出 3 4

解析：

从一个基础切片派生出的子切片的长度可能大于基础切片的长度。假设基础切片是 baseSlice，使用操作符 [low,high]，有如下规则：0 <= low <= high <= cap(baseSlice)，只要上述满足这个关系，下标 low 和 high 都可以大于 len(baseSlice)。

注： 以上资料源自网络