

1、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")
}
```

案例分析:考察golang的异常处理机制和defer机制。panic触发异常，而defer压栈延迟执行，最终panic会将异常向上抛出。

执行结果:

```
打印后
打印中
打印前
panic: 触发异常

.....堆栈异常信息
```

延伸阅读:

- [关于golang的panic recover异常错误处理](#)
- [Go语言中使用Defer几个场景](#)

2、非原子操作的return和defer联合使用的坑

```
package main

import (
    "fmt"
```

```

)

func main() {
    fmt.Println(doubleScore(0))    //0
    fmt.Println(doubleScore(20.0)) //40
    fmt.Println(doubleScore(50.0)) //50
}

func doubleScore(source float32) (score float32) {
    defer func() {
        if score < 1 || score >= 100 {
            // 将影响返回值
            score = source
        }
    }()
    score = source * 2
    return

    //或者
    //return source * 2
}

```

2、请指出下面代码的问题并说明原因

```

package main

import (
    "fmt"
)

type student struct {
    Name string
    Age  int
}

func pase_student() map[string]*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
    }
    return m
}

func main() {

```

```

students := pase_student()
for k, v := range students {
    fmt.Printf("key=%s,value=%v \n", k, v)
}
}

```

案例解析: Go的迭代变量会复用地址, 也就说迭代变量 `stu` 取地址 `&stu` 始终是一样的地址, 所以循环体中的代码问题在于每个`m`存储的`student`实例地址是一样的。解决方案可以将`map`中的指针声明修改为结构体类型。

而如果改为结构体类型, 则为非引用类型, 每次都是副本,值的拷贝。

故有三种解决方案:

所以解决方案有两个

1. 去除指针, 使用结构体
2. 保留指针, 但在每次迭代时, 都要取出, 使用shadow variable覆盖
3. 取出数组原始的指针值

方案一

```

type student struct {
    Name string
    Age  int
}

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

```

每次的循环迭代变量的地址都是一样,错误就在于每次赋值都是指向的底层地址, 循环变量最终都是执行完都是指向最后一个值。

```

0xc42000a260
0xc42000a260

```

0xc42000a260

key=zhou,value=&{wang 22}

key=li,value=&{wang 22}

key=wang,value=&{wang 22}

方案二

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

方案三

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

3、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:

默认情况runtime.GOMAXPROCS为4，即golang的任务调度器为4核，并发并行执行，goroutine执行是由调度器分配的，顺序是随机的。而第一行代码就将runtime.GOMAXPROCS设置为1，一个任务调度器，goroutine将被放到同一个goroutine任务列表中执行，顺序执行，但是由于第一个中的变量是go 匿名函数引用外部参数值，为闭包形式是延迟加载的，所以打印出的都是10（注意不是9），而第二个goroutine则为匿名函数中传递了参数，匿名函数参数则立即计算并拷贝的，所以每次都是正常执行。而又因为同一个任务列表中goroutine一般是FIFO的，但是go的机制有个特殊点在于runq.next也就是goroutine切换时会最后一个任务放到runq.next优先执行。所以，第二个循环中的goroutine列表中的最后一个goroutine将优先打印，其他的则按顺序执行。因此我们看到的打印输出结果将是

我做了下标记将其区分 m为第二个循环中的

```
m: 9  
i: 10  
i: 10  
i: 10  
i: 10  
i: 10  
i: 10  
i: 10  
i: 10  
i: 10  
m: 0  
m: 1  
m: 2  
m: 3  
m: 4  
m: 5  
m: 6
```

```
m: 7
m: 8
```

不过我用的是go1.8，其他版本未必是这个顺序，和调度器底层实现有关。所以还是随机性最大。

go没有继承，只有组合

下面代码会输出什么？

```
package main

import "fmt"

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:go没有继承，只有组合。

我们看到Teacher中有个匿名字段People，因为该匿名字段可以直接使用当前struct类型直接调用其组合类型包含的函数或字段。我们可以先看下非匿名字段的情况(匿名字段在调用时候较为便利，因为可以省略)。

当我们采用了命名字段，再次调用People的方法就必须加上字段名。因为其receiver要求的是

People。

那么我们也知道了，匿名字段下，其receiver其实也是People类型。那么调用的方法自然也是People

的方法。没有继承这么一说。毕竟这个People类型并不知道会被什么类型组合，当然也就无法调用方法时去使用未知的组合者Teacher类型的功能。

```
type Teacher struct {  
    p People  
}  
  
func main() {  
    t := Teacher{}  
    //t.ShowA()  
    t.p.ShowA()  
}
```

调用所谓"继承"或实际上来说是组合的方法B，我们可以采用t.ShowB()即可。其实也是t.p.ShowB()的缩写。

select伪随机

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: 可能会。

select可以在多个chan间等待执行。有三点原则：

- select 中只要有一个case能return，则立刻执行。
- 当如果同一时间有多个case均能return则伪随机方式抽取任意一个执行。
- 如果没有一个case能return则可以执行”default”块。

而在上述代码中运行，满足第二条原则满足，将会随机选择，所以是可能会。

slice追加

请写出以下输出内容

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

slice初始化之后[0,0,0,0,0]，容量也为5，但是可以自动扩容。append函数追加元素，不够则扩容，但不会覆盖原有的元素。所以输出将是[0,0,0,0,0,1,2,3]

锁问题

```
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  
}
```


map非并发安全，对于同一个指针的同一时间读写操作引发资源竞争，需要引入读写锁，否则会报错误信息：“fatal error: concurrent map read and map write”。go1.9中也可以使用并发安全的map来解决。

非缓冲通道的特性

```
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:非缓冲通道阻塞，需要有接守者，否则一直阻塞，将会引起panic

receiver 指针还是struct 还是指针或struct都可

```
package main

import (
    "fmt"
)

type People interface {
    Speak(string) string
}

type Stduent struct{}
```

```

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

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

```

解析:不能通过。因为值Student{}并未实现接口。需要&Student{}

边界检查

```

package main

import "fmt"

func max(numbers ...int) int {
    var largest int
    for _, v := range numbers {
        if v > largest {
            largest = v
        }
    }
    return largest
}

func main() {
    //greatest := max(4, 7, 9, 123, 543, 23, 435, 53, 125)
    //greatest := max(-1, -2) //如果都是负数将是错误的
    greatest := max2(-1, -2)
    fmt.Println(greatest)
}

func max2(numbers ...int) int {
    var largest int
    for i, v := range numbers {
        if v > largest || i == 0 {
            largest = v
        }
    }
}

```

```

    }
    return largest
}

/*
FYI
For your code to also work with only negative numbers such as

greatest := max(-200 -700)

include this as your range statement
for i, v := range numbers {
    if v > largest || i == 0 {
        largest = v
    }
}
}

What does that code do?

The first time through the range loop
the index, i, will be zero
so largest will be set to the first number

Originally largest is set to the zero value for an int, which is zero

Zero would be greater than any negative number

if you only have negative numbers
you need largest to be something less than zero

Thanks to Ricardo G for this code improvement!
*/

```

slice容量

问题:我们知道切片的容量将随着随着长度的增大而自动扩容，那么下面的代码能否正常运行?如果不行，该如何解决?

```

```go
package main

import "fmt"

func main() {

```

```
greeting := make([]string, 3, 5)
// 3 is length - number of elements referred to by the slice
// 5 is capacity - number of elements in the underlying array

greeting[0] = "Good morning!"
greeting[1] = "Bonjour!"
greeting[2] = "buenos dias!"
greeting[3] = "suprabadham"

fmt.Println(greeting[2])
```

```
}
```

### ## 可变参数的使用

```
`append (slice []Type,elem ...Type) `
```

- `...`符号可以解引用slice为可变参数
- 添加类型必须是一致的
- slice是开闭区间

```
```go
package main

import "fmt"

func main() {

    mySlice := []int{1, 2, 3, 4, 5}
    myOtherSlice := []int{6, 7, 8, 9}
    otherSlice := []int{11,12,13}
    //otherStringSlice:=[]string{"hello","world"}
    mySlice = append(mySlice, myOtherSlice...)
    mySlice = append(mySlice,otherSlice...)
    //mySlice = append(mySlice,otherStringSlice...)

    fmt.Println(mySlice)
}
```

去掉周三

```
package main
```

```
import "fmt"

func main() {

    mySlice := []string{"Monday", "Tuesday"}
    myOtherSlice := []string{"Wednesday", "Thursday", "Friday"}

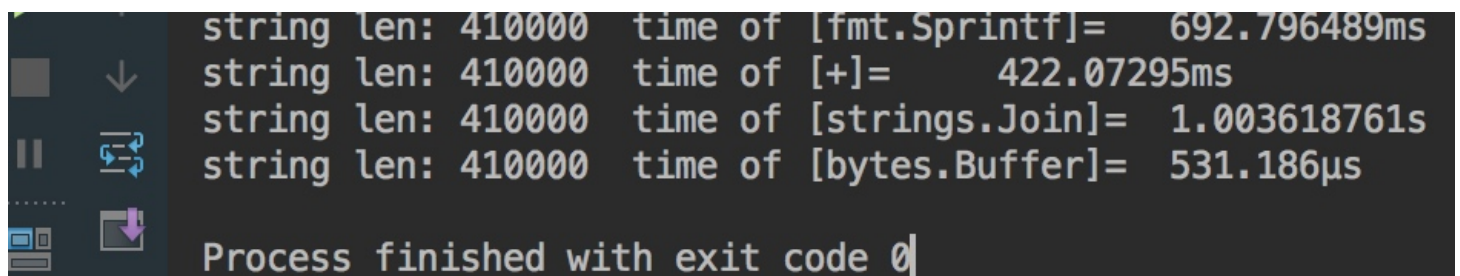
    mySlice = append(mySlice, myOtherSlice...)
    fmt.Println(mySlice)

    mySlice = append(mySlice[:2], mySlice[3:]...)
    fmt.Println(mySlice)

}
```

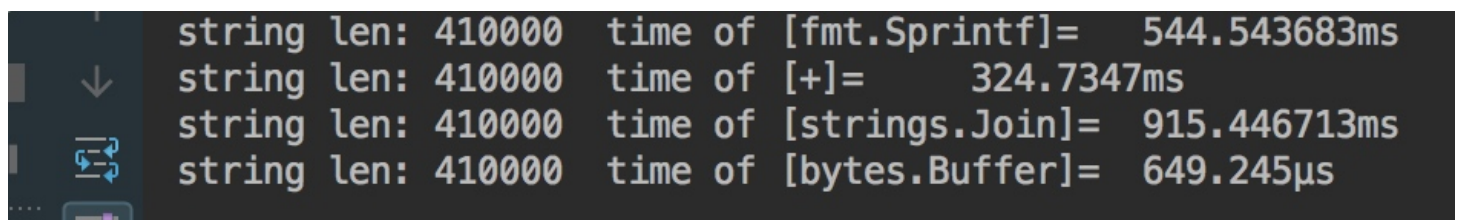
golang中的字符串连接有哪几种?哪种最高效

字符串连接哪一种方式最高效

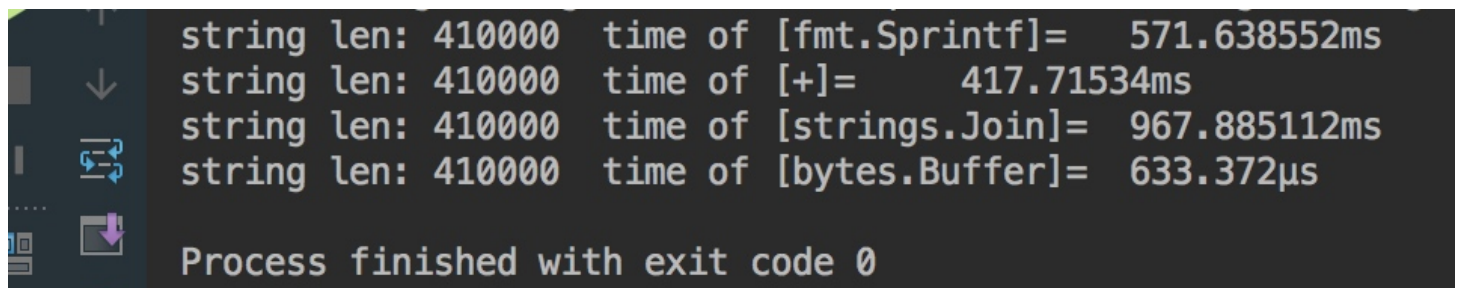


```
string len: 410000 time of [fmt.Sprintf]= 692.796489ms
string len: 410000 time of [+=] = 422.07295ms
string len: 410000 time of [strings.Join]= 1.003618761s
string len: 410000 time of [bytes.Buffer]= 531.186µs

Process finished with exit code 0
```



```
string len: 410000 time of [fmt.Sprintf]= 544.543683ms
string len: 410000 time of [+=] = 324.7347ms
string len: 410000 time of [strings.Join]= 915.446713ms
string len: 410000 time of [bytes.Buffer]= 649.245µs
```



```
string len: 410000 time of [fmt.Sprintf]= 571.638552ms
string len: 410000 time of [+=] = 417.71534ms
string len: 410000 time of [strings.Join]= 967.885112ms
string len: 410000 time of [bytes.Buffer]= 633.372µs

Process finished with exit code 0
```

strings.Join 最慢

fmt.Sprintf 和 string + 差不多

bytes.Buffer又比上者快约500倍

请使用两种方式将下列代码改进

```
ch := make(chan int)
ch <- 100
fmt.Println(<-ch)
```

```
//fatal error: all goroutines are asleep - deadlock!
```

分析：

原因是：

ch <- 100, 是 unbuffered channel , 它会block, 直到有人把它发送的消息取走。因此, 第6行的语句永远无法执行, 造成死锁

go判断死锁的代码位于：

src/pkg/runtime/proc.c

改进方式1:

```
ch := make(chan int, 1)
ch <- 100
fmt.Println(<-ch)
```

改进方式2:

```
ch := make(chan int)
go func(){
    ch <- 100
}()
fmt.Println(<-ch)
```

流水线问题

用goroutine打印4个文件内容

```
func main(){
    var wg sync.WaitGroup
    exit := make(chan bool)
```

```

ch := make(chan int,4)

for i:=0; i <cap(ch); i++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done()
        fmt.Println(id,"->",i+1)
    }(i)
    close(exit)
}

wg.Wait()
<-exit
}

```

[如何理解默认阻塞的channel](#)

[go deadlock](#)

[golang 实现一种环形队列，及周期任务](#)

[扛住100亿次请求？我们来试一试](#)

[进一步认识golang并发](#)

[golang每日一练](#)

[multiply thread](#)

[golang流水线问题](#)

<http://www.sunaloe.cn/about>

<http://fuxiaohei.me/2017/4/22/gopherchina-2017.html>

interface{} 另一个特殊场景就是空接口，对应的代码就是需要类型推断：

```

go
func do(v interface{}){
    switch t := v.(type){
    case int:
        fmt.Printf("int - %d",t)
    case error:
        fmt.Printf("error - %s",t.Error())
    default:
        fmt.Printf("interface - %v",t)
    }
}

```

```
}
```

不到万不得已不要这么写代码。否则需要推断类型的 case 越来越多，代码可维护性瞬间下降。

闭包引用问题

下列代码输出什么？

```
package main

import (
    "fmt"
    "os"
)

func main() {
    for i := 0; i < 3; i++ {
        go func() {
            fmt.Println(i)
        }()
    }
    os.Stdin.Read(make([]byte, 1))
}
```

上述原因在于main执行太快，而goroutine都没来得及执行，i已经变味了3

那么我们可以采取让main执行慢一些，生成的3个goroutine将会有看能正常执行

```
```go
```

```
package main
```

```
import (
```

```
 "fmt"
```

```
 "time"
```

```
)
```

```
func main() {
```

```
 for i := 0; i < 3; i++ {
```

```
 go func() {
```

```
 fmt.Println(i)
```

```
 }()
```



```
time.Sleep(1 * time.Second)
}
}
```

```
```go
    for i := 0; i < 3; i++ {
        go func(v int) {
            fmt.Println(v)
        }(i)
        time.Sleep(1 * time.Second)
    }
}
```

```
for i := 0; i < 3; i++ {
    v := i
    go func() {
        fmt.Println(v)
    }()
    time.Sleep(1 * time.Second)
}
```

redis实现优先级

redis实现优先级

迭代channel类型数据的注意

为避免死锁，迭代channel的时候一定要注意注意对channel的关闭。

```
func main() {
    c := make(chan bool)
    go func() {
        fmt.Println("test")
        c <- true
        close(c) // 需要加上，否则panic
    }()

    for v := range c {
```

```
        fmt.Println(v)
    }
}
```

cap(map)无法编译通过

```
package main

import "fmt"

func main() {
    m := make(map[int]string, 3)
    m[1] = "hello"
    m[2] = "world"
    m[3] = "test"
    m[4] = "test1"
    m[5] = "test2"
    fmt.Println(m)

    fmt.Println(len(m), cap(m))//invalid argument m (type map[int]string) for cap
}
```

map key不存在时的处理

先面代码有什么问题?该如何修改

```
go
package main
import "fmt"
func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}
    if v := x["two"]; v == "" {
        fmt.Println("no entry")
    }
}
```

解析：这对于那些希望得到“nil”标示符的开发者而言是个技巧（和其他语言中做的一样）。如果

对应的数据类型的“零值”是“nil”，那返回的值将会是“nil”，但对于其他的数据类型是不一样的。检测对应的“零值”可以用于确定map中的记录是否存在，但这并不总是可信（比如，如果在二值的map中“零值”是false，这时你要怎么做）。检测给定map中的记录是否存在的最可信的方法是，通过map的访问操作，检查第二个返回的值。

string 字符串修改

下列代码能否正确运行？有无不适用的场景？

```
package main

import "fmt"

func main() {
    x := "text"
    xbytes := []byte(x)
    xbytes[0] = 'T'
    fmt.Println(string(xbytes))
}
```

解析:能正常运行，但是不适用非单字节的字符。golang string本身并不可修改。而rune(int32别名)其实就是字符类型 byte(int8)字节。字符串由字节序列组成，由于每个字符占有的字节数并不固定，比如中文字符。所以需要将字符串转为字符数组类型，根据索引修改对应的字符，而不是字节。

string的索引操作是byte，所以想要获取特定的字符，推荐使用for range

utf8与go

理论上说 `RuneCountInString()` 函数并不返回字符的数量，因为单个字符可能占用多个rune。

```
package main
import (
    "fmt"
    "unicode/utf8"
)

func main() {
    data := "é"
```

```
fmt.Println(len(data)) //prints: 3
fmt.Println(utf8.RuneCountInString(data)) //prints: 2
}
```

go的静态变量

```
func reverse1(s []int) []int {
    length := len(s)
    // s2 := []int{} // s2 := [length]int{} //non-constant array bound length
    h

    for i := 0; i < length; i++ {
        j := length - i - 1
        s[j] = s[i]
    }
    return s
}
```

[Golang面试题解析](#)

[golang面试题解析](#)

[goroutine 的调度顺序是随机的](#)

[golang开发者的50度灰](#)

没参加过GO语言的面试，用了2年多快3年go语言，主要用来写游戏服务器框架，如果要我出面试题，大概会从以下几个方面考虑：1. Data Race问题怎么解决？能不能不加锁解决这个问题？2. 使用goroutine以及channel设计TCP链接的消息收发，以及消息处理。3. 使用go语言，编写并行计算的快速排序算法。然后结合项目需求再提一些问题，比如数据库，网络协议，加密解密。考go语言细节的话建议买本比较流行的go语言书，从里面挑几条重要的语法问题出一下，以免招不到人。另外就是go语言中比较特别的自带test和benchmark，这是我喜欢go语言的重要原因之一，我一直觉得软件测试是保证软件质量的重要一环，运用好这中测试方法可以极大的提高软件质量和开发效率，面试的时候也能问问这方面情况。

[go调度详解](#)