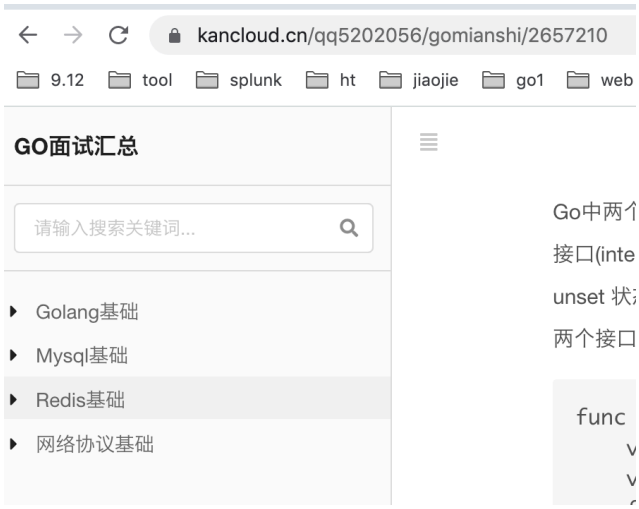
wbt-interview

https://www.kancloud.cn/qq5202056/gomianshi/2657210



https://github.com/interview0506/interview-go

1. https://github.com/interview0506/interview-go/blob/master/question/q007.md
2. https://github.com/interview0506/interview-go/blob/master/question/q008.md

9、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`.

11、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，

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。

2、 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

4、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

5、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。

10、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。

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

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()
}
```

如果不想使用 sync.WaitGroup, 也可以用一个 done channel.

```
package main

import (
 "fmt"
 "math/rand"
)

func main() {
 random := make(chan int)
 done := make(chan bool)

 go func() {
  for {
   num, ok := <-random
   if ok {
    fmt.Println(num)
   } else {
    done <- true
   }
  }
 }()

 go func() {
  defer close(random)

  for i := 0; i < 5; i++ {
   random <- rand.Intn(5)
  }
 }()

 <-done
 close(done)
}
```

题目

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
- 写已经关闭的 chan 会 panic

简单聊聊内存逃逸？

问题

知道golang的内存逃逸吗？什么情况下会发生内存逃逸？

回答

golang程序变量会携带有一组校验数据，用来证明它的整个生命周期是否在运行时完全可知。如果变量通过了这些校验，它就可以在栈上分配

能引起变量逃逸到堆上的典型情况：

- 在方法内把局部变量指针返回 局部变量原本应该在栈中分配，在栈中回收。但是由于返回时被外部引用，因此其生命周期大于栈，则溢出
- 发送指针或带有指针的值到 channel 中。 在编译时，是没有办法知道哪个 goroutine 会在 channel 上接收数据。所以编译器没法知道
- 在一个切片上存储指针或带指针的值。 一个典型的例子就是 []*string 。这会导致切片的内容逃逸。尽管其后面的数组可能是在栈上分
- slice 的背后数组被重新分配了，因为 append 时可能会超出其容量(cap)。 slice 初始化的地方在编译时是可以知道的，它最开始会在栈
- 在 interface 类型上调用方法。 在 interface 类型上调用方法都是动态调度的 —— 方法的真正实现只能在运行时知道。想像一个 io.Read

字符串转成byte数组，会发生内存拷贝吗？

问题

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。

不进行resp.Body.Close()，泄漏是一定的。但是泄漏的goroutine个数就让我迷糊了。由于执行了6遍，每次泄漏一个读和写goroutine，就是
然而执行程序，发现答案是3，出入有点大，为什么呢？

总结

- 所以结论呼之欲出了，虽然执行了 6 次循环，而且每次都没有执行 Body.Close() ,就是因为执行了ioutil.ReadAll()把内容都读出来了，连
  goroutine，所以答案就是3个goroutine。
- 从另外一个角度说，正常情况下我们的代码都会执行 ioutil.ReadAll()，但如果此时忘了 resp.Body.Close()，确实会导致泄漏。但如果你
  和一个写goroutine，这就是为什么代码明明不规范但却看不到明显内存泄漏的原因。
- 那么问题又来了，为什么上面要特意强调是同一个域名呢？改天，回头，以后有空再说吧。
- https://juejin.cn/post/6896993332019822605

- Go语言的GPM调度器是什么？
- Goroutine调度策略
- goroutine调度器概述

https://github.com/interview0506/interview-go/blob/master/mysql/mysql-interview.md

- MySQL数据库经典面试题解析
- MySQL InnoDB MVCC 机制的原理及实现
- 为什么MySQL使用B+树做索引？

所有的defer语句会放入栈中，在入栈的时候会进行相关的值拷贝（也就是下面的"对应的参数会实时解析"）。

defer、return、返回值三者的执行逻辑应该是：
return最先执行，return负责将结果写入返回值中；
接着defer开始执行一些收尾工作；
最后函数携带当前返回值（可能和最初的返回值不相同）退出

（1）无名返回值：
解释：返回值由变量i赋值，相当于返回值=i=0。第二个defer中i++ = 1， 第一个defer中i++ = 2，所以最终i的值是2。但是返回值已经被赋

（2）有名返回值：
解释：这里已经指明了返回值就是i，所以后续对i进行修改都相当于在修改返回值，所以最终函数的返回值是2。

（3）函数返回值为地址
此时的返回值是一个指针（地址），这个指针=&i，相当于指向变量i所在的地址，两个defer语句都对i进行了修改，那么返回值指向的地址的

在panic语句后面的defer语句不被执行
在panic语句前的defer语句会被执行

```
func deferExit() {
defer func() {
fmt.Println("defer")
}()
os.Exit(0)
}
```

当调用os.Exit()方法退出程序时，defer并不会被执行，上面的defer并不会输出。

++++++++++++++++++++++++
2、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_s

+++++++++

```
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)
}
```

Go的GPM如何调度 https://www.kancloud.c

Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）.

- M: M代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的；M是一个很大的结构，里面维护小对象内存cache（mcach
- G: 代表一个goroutine，它有自己的栈，instruction pointer和其他信息（正在等待的channel等等），用于调度.
- P: P全称是Processor，逻辑处理器，它的主要用途就是用来执行goroutine的，所以它也维护了一个goroutine队列，里面存储了所有需
- Sched：代表调度器，它维护有存储M和G的队列以及调度器的一些状态信息等.

Go中的GPM调度:

新创建的G 会先保存在 P 的本地队列中，如果 P 的本地队列已经满了就会保存在全局的队列中，最终等待被逻辑处理器P执行即可。

在M与P绑定后，M会不断从P的Local队列中无锁地取出G，并切换到G的堆栈执行，当P的Local队列中没有G时，再从Global队列中获取一个

golang中的g0和m0是指goroutine和machine的编号为0的特殊实例。

- g0（也称为system stack）：是Go语言运行时系统中的一个特殊的Goroutine，用于管理整个进程中的系统级别的任务，如垃圾回收、调度
- m0（也称为main
machine）：是Go语言运行时系统中的一个特殊的Machine，主要负责初始化，并在其中执行用户程序的代码。m0可以认为是一个普通的线

总之，g0和m0是Go语言运行时系统中的关键实例，它们分别扮演着系统级别任务管理和用户程序执行的角色。

每一个 M 标配一个 g0，专门用来执行调度相关的代码

M0 是主线程，用来执行 runtime.main -> main.main，M0 也会标配自己的 g0，虽然他只执行 runtime.main
这个答案要更新一下~

如果 main.main 里有阻塞操作的话~ m0 还是会去执行调度逻辑的

M0

1、启动程序后的编号为0的主线程(当前启动一个进程时，进程中会包含一个线程，进程中的第一个线程的编号设置为M0)

如果进程中不开任何线程，可以理解为一个进程就是一个线程。

2、在全局变量runtime.M0中，不需要在heap(堆)上分配。(M0在整个进程中是唯一的，无需在堆上分配)

3、负责执行初始化操作和启动第一个G(M0是负责启动第一个G的，go语言是跑在协程上)

4、启动第一个G之后，M0就和其他的M一样了(负责给其他M进行抢占)

G0

1、每次启动一个M，都会第一个创建的goroutine，就是G0(G0不是整个进程唯一的，而是一个线程中唯一的)

2、G0仅用于负责调度其他的G(M可能会有很多的G，然后G0用来保持调度栈的信息)

当一个M从G1切换到G2，首先应该切换到G0，通过G0把G1干掉，把G2加进来，可以理解G0是一个转换的桥梁

3、G0不指向任何可执行的函数

4、每一个M都会有一个自己的G0

5、在调度或系统调用是会使用M会切换到G0。来进行调度其他的G。

6、M0的G0会放在全局空间

创建一个M就会有一个G0，创建其他的M也会有其他的G0。

首先创建一个M0，M0是全局唯一的。创建一个M就会绑定一个G0，然后初始化goMaxProcs、P的本地队列和全局队列。

M0首先会创建一个第一个G，Main的goroutine，创建完后M0会和G0解绑，执行Main，M0需要找到一个空闲的P去捆绑。

然后将Main放入捆绑的P的本地队列中，然后Main就跟正常的G是一样的了。如果要执行Main就把Main从P的本地队列中

拿过去执行、

可能执行一半时间片(10ms）超时了，Main就依然回到P的本地队列中，再去执行，只到Main函数执行完。或者panic或exit

执行完，Main函数才会被消失。

- 缓存穿透：key对应的数据在数据源并不存在，每次针对此key的请求从缓存获取不到，请求都会到数据源，从而可能压垮数据源。
- 缓存击穿：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数
- 缓存雪崩：当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力