# 网址一：

https://blog.csdn.net/chen1415886044/article/details/104613097

## 1. 说说go语言的main函数

(1)、main函数不能带参数。

(2)、main函数不能定义返回值。

(3)、main函数所在的包必须为main包。

(4)、main函数中可以使用flag包来获取和解析命令行参数。

## 2. 在go语言中，new和make的区别？

new函数是内建函数，函数定义为

func new(Type) \*Type

new 的作用是初始化一个指向类型的指针(\*Type )，使用new函数来分配空间。

传递给 new 函数的是一个类型，不是一个值。

返回值是 指向这个新分配的零值的指针。

make函数是内建函数，函数定义为

func make(t Type, size ...IntegerType) Type

make 的作用是为 slice，map 或 chan 初始化并返回引用(Type)。

第一个参数是一个类型，第二个参数是长度。

make(T, args)函数的目的与new(T)不同。它仅仅用于创建 Slice, Map 和 Channel，

并且返回类型是 T（不是T\*）的一个初始化的（不是零值）的实例。

**new与make的区别**

1. 二者都是用来做内存分配的。
2. make只用于slice、map以及channel的初始化，返回的还是这三个引用类型本身；
3. 而new用于类型的内存分配，并且内存对应的值为类型零值，返回的是指向类型的指针。

## 3. 说说go语言中的switch语句？

分支控制语句，根据switch的值来跳转到对应的case去执行。

switch.case 可以处理各种类型，常用来做 接口 interface{} 的判断 (通过variable.(type)). 重

点是会依照 case 的顺序依序执行。

1.无须显式执行break语句，case执行完毕后自动中断;

2.不能出现重复的case常量值

3.只有在case中明确添加fallthrough关键字，才会继续执行紧跟的下一个case。

func convert(val interface{}) {  
 switch t := val.(type) {  
 case int:  
 fmt.Println("val为int类型", t)  
 case string:  
 fmt.Println("val为string类型", t)  
 case float64:  
 fmt.Println("val为float64类型", t)  
 case float32:  
 fmt.Println("val为float32类型", t)  
 case []string:  
 fmt.Println("val为字符串类型的切片", t)  
 default:  
 fmt.Println("val不是上列类型之一")  
 }  
}

## 4. 说说go语言中的for循环？

for循环支持continue和break来控制循环，但是它提供了一个更高级的break，可以选择中断哪一个循环。

for循环不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量 。

for...range

可用for..range完成数据迭代，支持字符串、数组、数组指针、切片、字典、通道类

型，返回索引、键值数据。

注意，range会复制目标数据。受直接影响的是数组，可改用数组指针或切片类型。

## 5.go语言中指针运算有哪些？

(1)、可以通过“&”取指针的地址。

(2)、可以通过“\*”取指针指向的数据。

(3)、区别于C/C++中的指针，Go语言中的指针不能进行偏移和运算，是安全指针。

不要将一个指向数组的指针传递给函数。使用切片。

//调用处 shuzu\_2(&array) 传递指针

func shuzu2(array \*[5]int){

array[0] = 1000

}

//调用处 slice\_shuzu(array[:]) 传递切片 实际上传递引用

func slice\_shuzu(array []int){

array[0] = 1000

}

## 6．说说go语言中的协程？

(1)、协程和线程都可以实现程序的并发执行；

(2)、只需要在函数调用前添加go关键字即可实现go的协程，创建并发任务；

(3)、关键字go并非执行并发任务，而是创建一个并发任务单元；

(4)、通过channel来进行协程间的通信；

## 7．go语言中的引用类型包含哪些？

数组切片、字典(map)、通道（channel）、接口（interface）

## 8. init（）函数和main（）函数

1.这两个函数都是go语言中的保留函数。init用于初始化信息，main用于作为程序的入口

2.这两个函数定义的时候：不能有参数，返回值。只能由go程序自动调用，不能被引用。

3.init（）函数可以定义在任意的包中，可以有多个。main（）函数只能在main包下，并且

只能有一个。

4.执行顺序

A：先执行init（）函数，后执行main（）函数

B：对于同一个go文件中，调用顺序是从上到下的，也就是说，先写的先被执行，后写

的后被执行

C：对于同一个包下，将文件名按照字符串进行排序，之后顺序调用各个文件中 init（）

函数

D：对于不同包下，

如果不存在依赖，按照main包中import的顺序来调用对应包中init（）函数

如果存在依赖，最后被依赖的最先被初始化

导入顺序：main->A->B->C

执行顺序：C->B->A->main

5.存在依赖的包之间，不能循环导入

6.一个包可以被其他多个包import，但是只能被初始化一次。

7.操作，其实是引入该包，而不直接使用包里面的函数，仅仅是调用了该包里的init（）

## 9、说说go语言的同步锁？

(1) 当一个goroutine获得了Mutex后，其他goroutine就只能乖乖的等待，除非该goroutine

释放这个Mutex

(2) RWMutex在读锁占用的情况下，会阻止写，但不阻止读

(3) RWMutex在写锁占用情况下，会阻止任何其他goroutine（无论读和写）进来，整个锁相

当于由该goroutine独占

## **10、说说go语言的关于go vendor？**

(1)、基本思路是将引用的外部包的源代码放在当前工程的vendor目录下面  
(2)、编译go代码会优先从vendor目录先寻找依赖包

(3)、有了vendor目录后，打包当前的工程代码到其他机器的$GOPATH/src下都可以通过编译。

Golang的依赖管理

<https://www.liwenzhou.com/posts/Go/go_dependency/>

如果不同的项目依赖同一个第三方的库的不同版本应该怎么解决？

方式一：vendor

Go语言从v1.5开始开始引入vendor模式，如果项目目录下有vendor目录，那么go工具链会优先使用vendor内的包进行编译、测试等。

godep是一个通过vender模式实现的Go语言的第三方依赖管理工具

godep开发流程

1. 保证程序能够正常编译
2. 执行godep save保存当前项目的所有第三方依赖的版本信息和代码
3. 提交Godeps目录和vender目录到代码库。
4. 如果要更新依赖的版本，可以直接修改Godeps.json文件中的对应项

方式二：go model

go module是Go1.11版本之后官方推出的版本管理工具，并且从Go1.13版本开始，go module将是Go语言默认的依赖管理工具。

## **11、说说go语言的channel特性？**

通道有发送（send）、接收(receive）和关闭（close）三种操作。

通道类型：

单向通道

ch2 := make(chan <- int) // 单向，只写，不能读

ch3 := make(<- chan int) //单向，只读，不能写

双向通道

缓冲通道

ch := make(chan type, capacity)



## **12、说说go语言的select机制？**

同switch相比，二者有个共同特性就是都通过case的方式来处理, 但除此之外几乎完全不同;

·select是go中的控制结构，该机制用来处理异步IO问题。select会监听case语句中

channel的读写操作，当case中channel读写操作为非阻塞状态（即能读写）时，将会触发相应的动作。

·每个 case 必须是一个通信操作，要么是发送要么是接收。

·如果有多个case都可以运行，select会随机公平地选出一个执行，其他不会执行。

·如果没有可运行的case语句，且有default语句，那么就会执行default的动作。

·如果没有可运行的case语句，且没有default语句，select将阻塞，直到某个case通

信可以运行

语法说明：

·每个case都必须是一个通信，要么发送，要么接收

·所有channel表达式都会被求值

·所有被发送的表达式都会被求值

·如果有多个case都可以运行，select会随机公平地选出一个执行。其他不会执

行。

否则：

如果有default子句，则执行该语句。

如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对

channel或值进行求值。

select {

case x := <-ch:

fmt.Println(x)

case ch <- i:

}

Select四种用法

<https://dashen.tech/2020/03/15/Golang%E4%B8%ADselect%E7%9A%84%E5%9B%9B%E5%A4%A7%E7%94%A8%E6%B3%95/>

**Select典型用法**：

<https://blog.csdn.net/zhaominpro/article/details/77570290>

1. 超时控制

//比如在下面的场景中，使用全局resChan来接受response，如果时间超过3S,resChan中还没有数据返回，则第二条case将执行

var resChan = make(chan int)

// do request

func test() {

select {

case data := <-resChan:

doData(data)

case <-time.After(time.Second \* 3):

fmt.Println("request time out")

}

}

func doData(data int) {

//...

}

1. 检查channel是否已满

//在某些情况下是存在不希望channel缓存满了的需求的，可以用如下方法判断

ch := make (chan int, 5)

//...

data：=0

select {

case ch <- data:

default:

//做相应操作，比如丢弃data。视需求而定

}

1. 退出

//主线程（协程）中如下：

var shouldQuit=make(chan struct{})

fun main(){

{

//loop

}

//...out of the loop

select {

case <-c.shouldQuit:

cleanUp()

return

default:

}

//...

}

//再另外一个协程中，如果运行遇到非法操作或不可处理的错误，就向shouldQuit发送数据通知程序停止运行

close(shouldQuit)

举一个有趣的列子：使用channel将异步变为同步

题目如下：

存在以下代码，请问如何实现print函数可以顺序输出1~75，要求不使用锁，只能使用channel

func main() {

for i := 1; i < 75; i++ {

go print(i)

}

time.Sleep(5\*time.Second) //wait all goroutine

}

func print(i int){

}

实现完整代码如下：

package main

import (

"fmt"

"time"

)

var ch = make(chan int)

func main() {

for i := 1; i <= 75; i++ {

go print(i)

ch <- 0

}

time.Sleep(5 \* time.Second) //wait all goroutines

}

func print(i int) {

<-ch

fmt.Println(i)

}

## **13、说说go语言的goconvey框架？**

(1)、goconvey是一个支持golang的单元测试框架

(2)、goconvey能够自动监控文件修改并启动测试，并可以将测试结果实时输出到web界面

(3)、goconvey提供了丰富的断言简化测试用例的编写

对于Golang的程序员来说，至少需要掌握下面四个测试框架：

* GoConvey
* GoStub
* GoMock
* Monkey

## **14、协程，线程，进程的区别？**

* **进程**

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。**每个进程都有自己的独立内存空间，不同进程通过进程间通信来通信**。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

* **线程**

线程是进程的一个实体,**是CPU调度和分派的基本单位**,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

* **协程**

**协程是一种用户态的轻量级线程，协程的调度完全由用户控制**。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

## **15、 Golang的内存模型，为什么小对象多了会造成gc压力。**

通常小对象过多会导致GC三色法消耗过多的GPU。优化思路是，减少对象分配.

## **16、并发编程概念是什么？**

并行是指两个或者多个事件在同一时刻发生；并发是指两个或多个事件在同一时间间隔发生。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

并发编程是指在一台处理器上“同时”处理多个任务。并发是在同一实体上的多个事件。多个事件在同一时间间隔发生。并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

## **17、Log包线程安全吗？**

Golang的标准库提供了log的机制，但是该模块的功能较为简单（看似简单，其实他有他的设计思路）。在输出的位置做了线程安全的保护。

## 18**、主协程如何等其余协程完再操作？**

使用channel进行通信，context,select。

## **19、context包的用途**

在go语言并发编程中，用一个goroutine来处理个任务，而它又会创建多个goroutine来负责不同子任务的场景非常常见，这些场景中往往会需要在API边界之间以及过程之间传递截止时间、取消信号或其它与请求相关的数据，这时候就可以使用context

1.控制并发

1.1。一种是使用sync包下的WaitGroup。

这种方式适合的场景是等待一个goroutine的集合执行完成。

1.2.一种是context 使用的场景是多个goroutine执行同一件事情

2.主动通知停止场景

3.并发超时控制

## **20、slice，len，cap，共享，扩容**

append函数，因为slice底层数据结构是，由数组、len、cap组成，所以，在使用append扩容时，会查看数组后面有没有连续内存快，有就在后面添加，没有就重新生成一个大的素组。

type SliceHeader struct {

Data uintptr

Len int

Cap int

}

### Slice的补充

<https://zcfy.cc/article/how-to-use-slice-capacity-and-length-in-go>

下面的代码输出什么？

vals := make([]int, 5)

for i := 0; i < 5; i++ {

vals = append(vals, i)

}

fmt.Println(vals)

答案是[0 0 0 0 0 0 1 2 3 4] 而不是 [0 1 2 3 4]，因为使用make创建时已经指定了长度，

而append接口是在slice后面追加，容量自动扩充了一倍（从原来的5变成了10）.

#### 要想使结果为[0 1 2 3 4]可以使用如下办法实现

##### a.直接使用索引写入而不是 append

vals := make([]int, 5)

for i := 0; i < 5; i++ {

vals[i] = i

}

fmt.Println(vals)

##### b.使用 0 作为你的长度并指定容量

func keys(m map[string]struct{}) []string {

ret := make([]string, 0, len(m))

for key := range m {

ret = append(ret, key)

}

return ret

}

如果 append 处理它，为什么我们还要担心容量呢？

事实是，在大多数情况下，你不必担心这太多。如果它使你的代码变得更复杂，只

需用 var vals []int 初始化你的切片，然后让 append 函数处理接下来的事。这种情

况下 slice的容量变化为1，2，4，8...，每一次扩大程序都会有不必要的内存分配。

并且还容易造成内存浪费（如果用到了65个容量，程序自动扩充到了128），所以

程序员手动指定容量大小。

## **21、map如何顺序读取**

**map不能顺序读取，是因为他是无序的**，想要有序读取，首先的解决的问题就是，把key变为有序，所以可以把key放入切片，对切片进行排序，遍历切片，通过key取值。

下面是一个顺序读取map例子，将map的key值存储到一个slice数组中，然后对数组进行排序，这样就可以通过slice数组里的值顺序去读map。

package main

import (

"fmt"

"sort"

)

func main() {

/\* 声明一个字符串切片，存储map的key值 \*/

var Name []string

Name = append(Name, "Bob", "Andy", "Clark", "David", "Ella")

/\* 声明索引类型为字符串的map \*/

var Person = make(map[string]string)

Person["Bob"] = "B"

Person["Andy"] = "A"

Person["Clark"] = "C"

Person["David"] = "D"

Person["Ella"] = "E"

fmt.Println("未排序输出:")

for key, value := range Person {

fmt.Println(key, ":", value)

}

/\* 对slice数组进行排序，然后就可以根据key值顺序读取map \*/

sort.Strings(Name)

fmt.Println("排序输出:")

for \_, Key := range Name {

/\* 按顺序从MAP中取值输出 \*/

if Value, ok := Person[Key]; ok {

fmt.Println(Key, ":", Value)

}

}

}

# 网址二：

<https://www.cnblogs.com/yulibostu/articles/12056197.html>

<https://www.cnblogs.com/wpgraceii/p/10528183.html>

## 1. Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量？

Golang中Goroutine 可以通过 Channel 进行安全读写共享变量

## 2. 无缓冲 Chan 的发送和接收是否同步?

ch := make(chan int) 无缓冲的channel由于没有缓冲发送和接收需要同步.

ch := make(chan int, 2) 有缓冲channel不要求发送和接收操作同步.

·channel无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到数据。

·channel有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

## 3. go语言的并发机制以及它所使用的CSP并发模型．

https://www.jianshu.com/p/36e246c6153d

### CSP并发模型

CSP模型是上个世纪七十年代提出的，不同于传统的多线程通过共享内存来通信，CSP讲究的是“以通信的方式来共享内存”。用于描述两个独立的并发实体通过共享的通讯 channel(管道)进行通信的并发模型。 CSP中channel是第一类对象，它不关注发送消息的实体，而关注与发送消息时使用的channel。

### Golang CSP

Golang 就是借用CSP模型的一些概念为之实现并发进行理论支持，其实从实际上出发，go语言并没有，完全实现了CSP模型的所有理论，仅仅是借用了 process和channel这两个概念。process是在go语言上的表现就是 goroutine 是实际并发执行的实体，每个实体之间是通过channel通讯来实现数据共享。

### Channel

Golang中使用 CSP中 channel 这个概念。channel 是被单独创建并且可以在进程之间传递，它的通信模式类似于 boss-worker 模式的，一个实体通过将消息发送到channel 中，然后又监听这个 channel 的实体处理，两个实体之间是匿名的，这个就实现实体中间的解耦，其

中 channel 是同步的一个消息被发送到 channel 中，最终是一定要被另外的实体消费掉的，在实现原理上其实是一个阻塞的消息队列。

### Goroutine

Goroutine 是实际并发执行的实体，它底层是使用协程(coroutine)实现并发，coroutine是一种运行在用户态的用户线程，类似于 greenthread，go底层选择使用coroutine的出发点是因为，它具有以下特点：

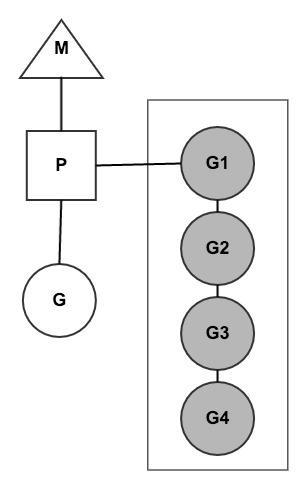
* 用户空间 避免了内核态和用户态的切换导致的成本
* 可以由语言和框架层进行调度
* 更小的栈空间允许创建大量的实例

### Goroutine 调度器

上节已经说了，golang使用goroutine做为最小的执行单位，但是这个执行单位还是在用户空间，实际上最后被处理器执行的还是内核中的线程，用户线程和内核线程的调度方法有：

·N:1 多个用户线程对应一个内核线程 用户级线程模型  
·1:1 一个用户线程对应一个内核线程 内核级线程模型

·M:N 用户线程和内核线程是多对多的对应关系 两级线程模型  
golang 通过为goroutine提供语言层面的调度器，来实现了高效率的M:N线程对应关系

  
图中

* M：系统线程，由操作系统管理goroutine就是跑在M之上的.
* P : Processor，处理器，它的主要用途就是用来执行goroutine的，它维护了一个

goroutine队列，即runqueue。Processor是

让我们从N：1调度到M:N调度的重要部分。

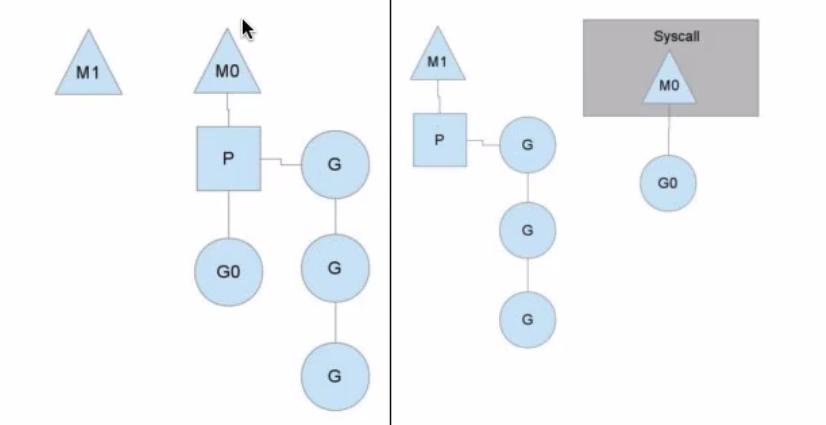
* G : 是待执行的goroutine，它包含了栈空间指令指针
* Gn : 灰色背景的Gn 是已经挂起的goroutine，它们被添加到了执行队列中，然后需要等待网络IO的goroutine，当P通过 epoll查询到特定的fd的时候，会重新调度起对应的，正在挂起的goroutine。

Golang为了调度的公平性，在调度器加入了steal working 算法 ，在一个P自己的执行队列，处理完之后，它会先到全局的执行队列中偷G进行处理，如果没有的话，再会到其他P的执行队列中抢G来进行处理。

#### 线程阻塞

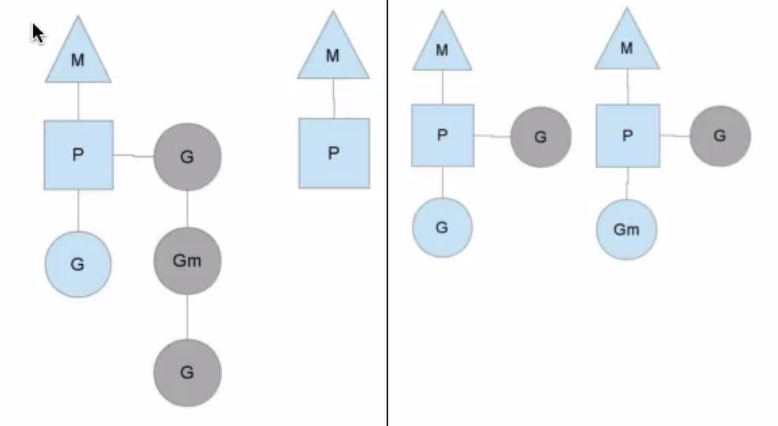
当正在运行的goroutine阻塞的时候，例如进行系统调用，会再创建一个系统线程（M1），

当前的M线程放弃了它的Processor，P转到新的线程中去运行。



#### runqueue执行完成

当其中一个Processor的runqueue为空，没有goroutine可以调度。它会从另外一个上下文偷取一半的goroutine。



#### P的作用

如果将G分配到适当的M上运行，G需要向M申请堆栈内存，资源竞争会造成很多资源的损耗，为了解决这样的问题引入了P，在P对象上预先申请好内存，在G需要运行时先向自己的P申请，不够用或者没有再向M申请。

P解耦了M和G

### 总结

Golang实现了 CSP 并发模型做为并发基础，底层使用goroutine做为并发实体，goroutine非常轻量级可以创建几十万个实体。实体间通过 channel 继续匿名消息传递使之解耦，在语言层面实现了自动调度，这样屏蔽了很多内部细节，对外提供简单的语法关键字，大大简化了并发编程的思维转换和管理线程的复杂性。

## 4. Golang 中常用的并发模型？

Golang 中常用的并发模型有三种:

### 通过channel通知实现并发控制

使用无缓冲通道来实现，发送 goroutine 和接收 gouroutine 必须是同步的，同时准备后，如果没有同时准备好的话，先执行的操作就会阻塞等待，直到另一个相对应的操作准备好为止。这种无缓冲的通道我们也称之为同步通道。

### 通过sync包中的WaitGroup实现并发控制

Goroutine是异步执行的，有的时候为了防止在结束mian函数的时候结束掉Goroutine，所以需要同步等待，这个时候就需要用 WaitGroup了，在 sync 包中，提供了 WaitGroup ，它会等待它收集的所有 goroutine 任务全部完成。在WaitGroup里主要有三个方法:

* Add, 可以添加或减少 goroutine的数量.
* Done, 相当于Add(-1).
* Wait, 执行后会堵塞主线程，直到WaitGroup 里的值减至0.

在主 goroutine 中 Add(delta int) 索要等待goroutine 的数量。 在每一个 goroutine 完成后 Done() 表示这一个goroutine 已经完成，当所有的 goroutine 都完成后，在主 goroutine 中 WaitGroup 返回返回。

### 在Go 1.7 以后引进的强大的Context上下文，实现并发控制

通常,在一些简单场景下使用 channel 和 WaitGroup 已经足够了，但是当面临一些复杂多变的网络并发场景下 channel 和 WaitGroup 显得有些力不从心了。 比如一个网络请求 Request，每个 Request 都需要开启一个 goroutine 做一些事情，这些 goroutine 又可能会开启其他的 goroutine，比如数据库和RPC服务。 所以我们需要一种可以跟踪 goroutine 的方案，才可以达到控制他们的目的，这就是Go语言为我们提供的 Context，称之为上下文非常贴切，它就是goroutine 的上下文。 它是包括一个程序的运行环境、现场和快照等。每个程序要运行时，都需要知道当前程序的运行状态，通常Go 将这些封装在一个 Context 里，再将它传给要执行的 goroutine 。

## 5. JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗？

nil slice：只是声明了slice，却没有给实例化的对象

empty slice： 是指slice不为nil，但是slice没有值，slice的底层的空间是空的

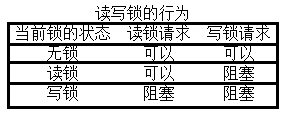
## 6. 互斥锁，读写锁，死锁问题是怎么解决。

* 互斥锁

互斥锁就是互斥变量mutex，用来锁住临界区的.

* 读写锁

通常有些公共数据修改的机会很少，但其读的机会很多。并且在读的过程中会伴随着查找，给这种代码加锁会降低我们的程序效率。读写锁可以解决这个问题。

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/61.jpg)

注意：写独占，读共享，写锁优先级高

* 死锁

一般情况下，如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。

另外一种情况是：若线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。

死锁产生的四个必要条件:

1. 互斥条件：一个资源每次只能被一个进程使用
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。
4. 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

## 7. Data Race问题怎么解决？能不能不加锁解决这个问题？

同步访问共享数据是处理数据竞争的一种有效的方法.golang在1.1之后引入了竞争检测机制，可以使用 go run -race 或者 go build -race来进行静态检测。 其在内部的实现是,开启多个协程执行同一个命令， 并且记录下每个变量的状态.

竞争检测器基于C/C++的ThreadSanitizer 运行时库，该库在Google内部代码基地和Chromium找到许多错误。这个技术在2012年九月集成到Go中，从那时开始，它已经在标准库中检测到42个竞争条件。现在，它已经是我们持续构建过程的一部分，当竞争条件出现时，它会继续捕捉到这些错误。

竞争检测器已经完全集成到Go工具链中，仅仅添加-race标志到命令行就使用了检测器。

$ go test -race mypkg // 测试包

$ go run -race mysrc.go // 编译和运行程序

$ go build -race mycmd // 构建程序

$ go install -race mypkg // 安装程序

要想解决数据竞争的问题可以使用互斥锁sync.Mutex,解决数据竞争(Data race),也可以使用管道解决,使用管道的效率要比互斥锁高.

## 8. 什么是channel，为什么它可以做到线程安全？

Channel是Go中的一个核心类型，可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯(communication),Channel也可以理解是一个先进先出的队列，通过管道进行通信。

Golang的Channel,发送一个数据到Channel 和 从Channel接收一个数据 都是 原子性的。而且Go的设计思想就是:不要通过共享内存来通信，而是通过通信来共享内存，前者就是传统的加锁，后者就是Channel。也就是说，设计Channel的主要目的就是在多任务间传递数据的，这当然是安全的。

## 9. Golang GC 时会发生什么?

首先我们先来了解下垃圾回收.什么是垃圾回收？

内存管理是程序员开发应用的一大难题。传统的系统级编程语言（主要指C/C++）中，程序开发者必须对内存小心的进行管理操作，控制内存的申请及释放。因为稍有不慎，就可能产生内存泄露问题，这种问题不易发现并且难以定位，一直成为困扰程序开发者的噩梦。如何解决这个头疼的问题呢？

过去一般采用两种办法：

* 内存泄露检测工具。这种工具的原理一般是静态代码扫描，通过扫描程序检测可能出现内存泄露的代码段。然而检测工具难免有疏漏和不足，只能起到辅助作用。
* 智能指针。这是 c++ 中引入的自动内存管理方法，通过拥有自动内存管理功能的指针对象来引用对象，是程序员不用太关注内存的释放，而达到内存自动释放的目的。这种方法是采用最广泛的做法，但是对程序开发者有一定的学习成本（并非语言层面的原生支持），而且一旦有忘记使用的场景依然无法避免内存泄露。

为了解决这个问题，后来开发出来的几乎所有新语言（java，python，php等等）都引入了语言层面的自动内存管理 – 也就是语言的使用者只用关注内存的申请而不必关心内存的释放，内存释放由虚拟机（virtual machine）或运行时（runtime）来自动进行管理。而这种对不再使用的内存资源进行自动回收的行为就被称为垃圾回收。

### 常用的垃圾回收的方法:

#### 引用计数（reference counting）

这是最简单的一种垃圾回收算法，和之前提到的智能指针异曲同工。对每个对象维护一个引用计数，当引用该对象的对象被销毁或更新时被引用对象的引用计数自动减一，当被引用对象被创建或被赋值给其他对象时引用计数自动加一。当引用计数为0时则立即回收对象。

这种方法的优点是实现简单，并且内存的回收很及时。这种算法在内存比较紧张和实时性比较高的系统中使用的比较广泛，如ios cocoa框架，php，python等。

但是简单引用计数算法也有明显的缺点：

1. 频繁更新引用计数降低了性能。

一种简单的解决方法就是编译器将相邻的引用计数更新操作合并到一次更新；还有一种方法是针对频繁发生的临时变量引用不进行计数，而是在引用达到0时通过扫描堆栈确认是否还有临时对象引用而决定是否释放。等等还有很多其他方法，具体可以参考这里。

1. 循环引用。

当对象间发生循环引用时引用链中的对象都无法得到释放。最明显的解决办法是避免产生循环引用，如cocoa引入了strong指针和weak指针两种指针类型。或者系统检测循环引用并主动打破循环链。当然这也增加了垃圾回收的复杂度。

#### 标记-清除（mark and sweep）

标记-清除（mark and sweep）分为两步，标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为“被引用”；标记完成后进行清除操作，对没有标记过的内存进行回收（回收同时可能伴有碎片整理操作）。这种方法解决了引用计数的不足，但是也有比较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执行，回收是系统响应能力大大降低！当然后续也出现了很多mark&sweep算法的变种（如三色标记法）优化了这个问题。

#### 分代搜集（generation）

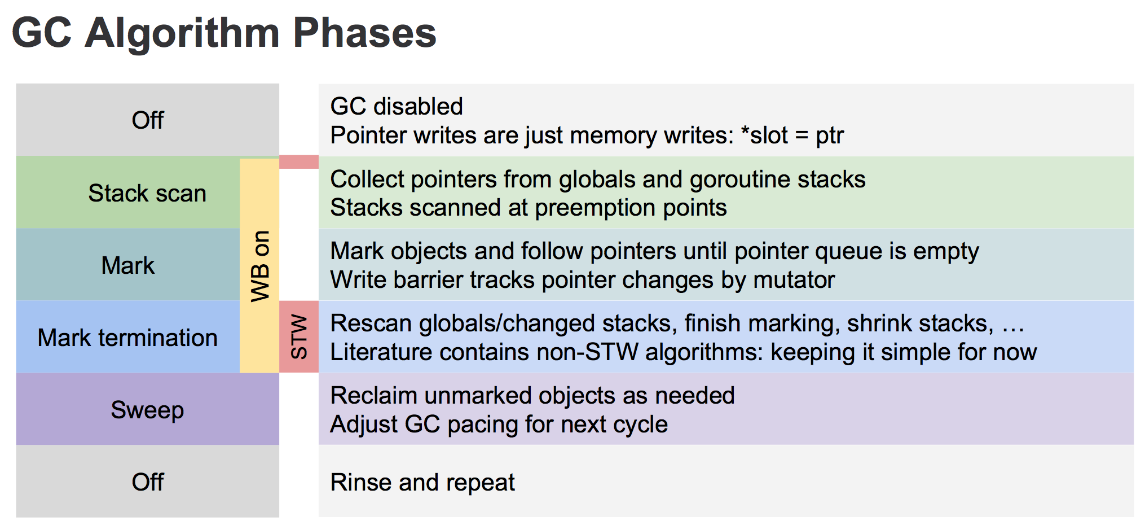
java的jvm 就使用的分代回收的思路。在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为代（generation）的空间。新创建的对象存放在称为新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多），随着垃圾回收的重复执行，生命周期较长的对象会被提升（promotion）到老年代中（这里用到了一个分类的思路，这个是也是科学思考的一个基本思路）。

因此，新生代垃圾回收和老年代垃圾回收两种不同的垃圾回收方式应运而生，分别用于对各自空间中的对象执行垃圾回收。新生代垃圾回收的速度非常快，比老年代快几个数量级，即使新生代垃圾回收的频率更高，执行效率也仍然比老年代垃圾回收强，这是因为大多数对象的生命周期都很短，根本无需提升到老年代。

### Golang GC 时会发生什么?

Golang 1.5后，采取的是“非分代的、非移动的、并发的、三色的”标记清除垃圾回收算法。

golang 中的 gc 基本上是标记清除的过程：

[](https://github.com/KeKe-Li/For-learning-Go-Tutorial/blob/master/src/images/2.jpg)

gc的过程一共分为四个阶段：

1. 栈扫描（开始时STW）
2. 第一次标记（并发）
3. 第二次标记（STW）
4. 清除（并发）

整个进程空间里申请每个对象占据的内存可以视为一个图，初始状态下每个内存对象都是白色标记。

1. 先STW，做一些准备工作，比如 enable write barrier。然后取消STW，将扫描任务作为多个并发的goroutine立即入队给调度器，进而被CPU处理
2. 第一轮先扫描root对象，包括全局指针和 goroutine 栈上的指针，标记为灰色放入队列
3. 第二轮将第一步队列中的对象引用的对象置为灰色加入队列，一个对象引用的所有对象都置灰并加入队列后，这个对象才能置为黑色并从队列之中取出。循环往复，最后队列为空时，整个图剩下的白色内存空间即不可到达的对象，即没有被引用的对象；
4. 第三轮再次STW，将第二轮过程中新增对象申请的内存进行标记（灰色），这里使用了write barrier（写屏障）去记录

Golang gc 优化的核心就是尽量使得 STW(Stop The World) 的时间越来越短。

## 10. Golang 中 Goroutine 如何调度?

goroutine是Golang语言中最经典的设计，也是其魅力所在，goroutine的本质是协程，是实现并行计算的核心。 goroutine使用方式非常的简单，只需使用go关键字即可启动一个协程，并且它是处于异步方式运行，你不需要等它运行完成以后在执行以后的代码。

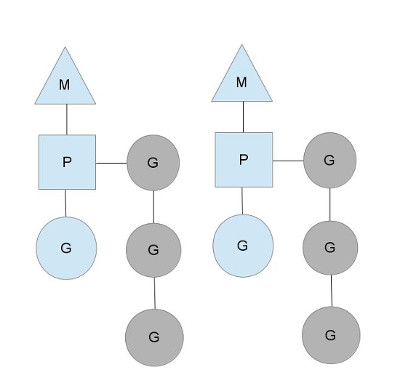
go func()//通过go关键字启动一个协程来运行函数

协程:

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。 因此，协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。 线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作执行者则是用户自身程序，goroutine也是协程。

groutine能拥有强大的并发实现是通过GPM调度模型实现.

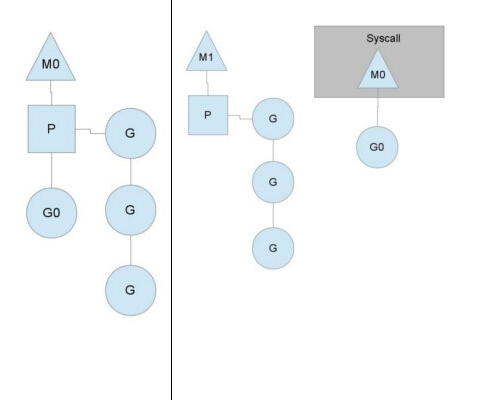
调度实现:

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/65.jpg)

从上图中可以看到，有2个物理线程M，每一个M都拥有一个处理器P，每一个也都有一个正在运行的goroutine。P的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个goroutine可以同时运行。

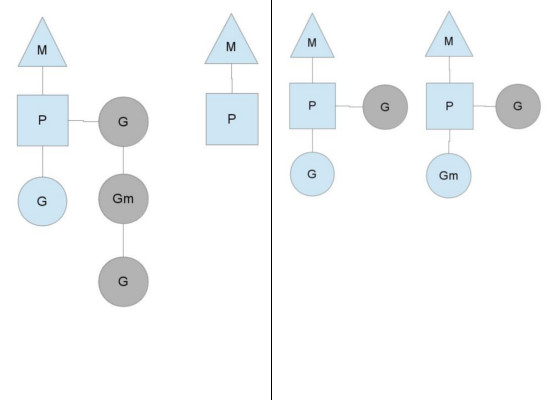
图中灰色的那些goroutine并没有运行，而是出于ready的就绪态，正在等待被调度。P维护着这个队列（称之为runqueue），Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出（如何决定取哪个goroutine？）一个goroutine执行。

当一个OS线程M0陷入阻塞时，P转而在运行M1，图中的M1可能是正被创建，或者从线程缓存中取出。

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/60.jpg)

当MO返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS线程那里拿一个P过来， 如果没有拿到的话，它就把goroutine放在一个global runqueue里，然后自己睡眠（放入线程缓存里）。所有的P也会周期性的检查global runqueue并运行其中的goroutine，否则global runqueue上的goroutine永远无法执行。

另一种情况是P所分配的任务G很快就执行完了（分配不均），这就导致了这个处理器P很忙，但是其他的P还有任务，此时如果global runqueue没有任务G了，那么P不得不从其他的P里拿一些G来执行。

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/64.jpg)

通常来说，如果P从其他的P那里要拿任务的话，一般就拿run queue的一半，这就确保了每个OS线程都能充分的使用。

## 11. 负载均衡原理是什么?

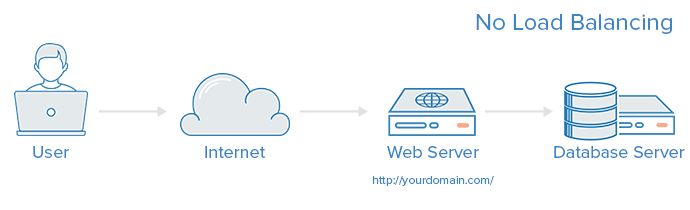
负载均衡Load Balance）是高可用网络基础架构的关键组件，通常用于将工作负载分布到多个服务器来提高网站、应用、数据库或其他服务的性能和可靠性。负载均衡，其核心就是网络流量分发，分很多维度。

负载均衡（Load Balance）通常是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

负载均衡是建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

### 通过一个例子详细介绍:

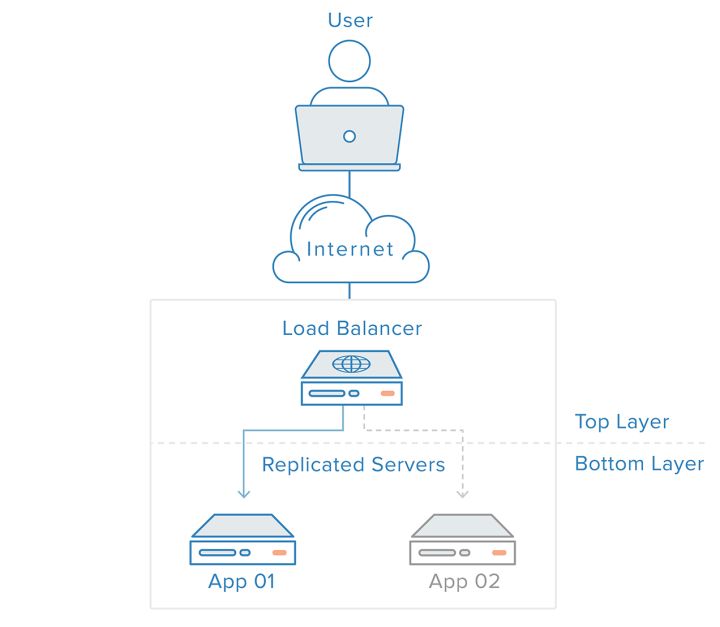
* 没有负载均衡 web 架构

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/66.jpg)

在这里用户是直连到 web 服务器，如果这个服务器宕机了，那么用户自然也就没办法访问了。 另外，如果同时有很多用户试图访问服务器，超过了其能处理的极限，就会出现加载速度缓慢或根本无法连接的情况。

而通过在后端引入一个负载均衡器和至少一个额外的 web 服务器，可以缓解这个故障。 通常情况下，所有的后端服务器会保证提供相同的内容，以便用户无论哪个服务器响应，都能收到一致的内容。

* 有负载均衡 web 架构

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/67.jpg)

用户访问负载均衡器，再由负载均衡器将请求转发给后端服务器。在这种情况下，单点故障现在转移到负载均衡器上了。 这里又可以通过引入第二个负载均衡器来缓解。

### 那么负载均衡器的工作方式是什么样的呢,负载均衡器又可以处理什么样的请求？

负载均衡器的管理员能主要为下面四种主要类型的请求设置转发规则：

* HTTP (七层)
* HTTPS (七层)
* TCP (四层)
* UDP (四层)

负载均衡器如何选择要转发的后端服务器？

负载均衡器一般根据两个因素来决定要将请求转发到哪个服务器。首先，确保所选择的服务器能够对请求做出响应，然后根据预先配置的规则从健康服务器池（healthy pool）中进行选择。

因为，负载均衡器应当只选择能正常做出响应的后端服务器，因此就需要有一种判断后端服务器是否健康的方法。为了监视后台服务器的运行状况，运行状态检查服务会定期尝试使用转发规则定义的协议和端口去连接后端服务器。 如果，服务器无法通过健康检查，就会从池中剔除，保证流量不会被转发到该服务器，直到其再次通过健康检查为止。

负载均衡算法

负载均衡算法决定了后端的哪些健康服务器会被选中。 其中常用的算法包括：

* Round Robin（轮询）：为第一个请求选择列表中的第一个服务器，然后按顺序向下移动列表直到结尾，然后循环。
* Least Connections（最小连接）：优先选择连接数最少的服务器，在普遍会话较长的情况下推荐使用。
* Source：根据请求源的 IP 的散列（hash）来选择要转发的服务器。这种方式可以一定程度上保证特定用户能连接到相同的服务器。

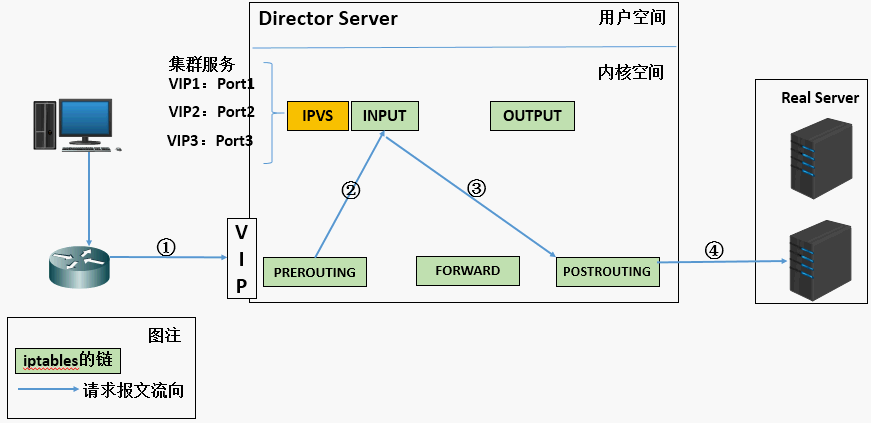
如果你的应用需要处理状态而要求用户能连接到和之前相同的服务器。可以通过 Source 算法基于客户端的 IP 信息创建关联，或者使用粘性会话（sticky sessions）。

除此之外，想要解决负载均衡器的单点故障问题，可以将第二个负载均衡器连接到第一个上，从而形成一个集群。

## 12. LVS相关 了解.

LVS是 Linux Virtual Server 的简称，也就是Linux虚拟服务器。这是一个由章文嵩博士发起的一个开源项目，它的官方网站是[LinuxVirtualServer](http://www.linuxvirtualserver.org/)现在 LVS 已经是 Linux 内核标准的一部分。使用 LVS 可以达到的技术目标是：通过 LVS 达到的负载均衡技术和 Linux 操作系统实现一个高性能高可用的 Linux 服务器集群，它具有良好的可靠性、可扩展性和可操作性。 从而以低廉的成本实现最优的性能。LVS 是一个实现负载均衡集群的开源软件项目，LVS架构从逻辑上可分为调度层、Server集群层和共享存储。

### LVS的基本工作原理:

[](https://github.com/KeKe-Li/golang-interview-questions/blob/master/src/images/68.jpg)

1. 当用户向负载均衡调度器（Director Server）发起请求，调度器将请求发往至内核空间
2. PREROUTING链首先会接收到用户请求，判断目标IP确定是本机IP，将数据包发往INPUT链
3. IPVS是工作在INPUT链上的，当用户请求到达INPUT时，IPVS会将用户请求和自己已定义好的集群服务进行比对，如果用户请求的就是定义的集群服务，那么此时IPVS会强行修改数据包里的目标IP地址及端口，并将新的数据包发往POSTROUTING链
4. POSTROUTING链接收数据包后发现目标IP地址刚好是自己的后端服务器，那么此时通过选路，将数据包最终发送给后端的服务器

### LVS的组成:

LVS 由2部分程序组成，包括 ipvs 和 ipvsadm。

1. ipvs(ip virtual server)：一段代码工作在内核空间，叫ipvs，是真正生效实现调度的代码。
2. ipvsadm：另外一段是工作在用户空间，叫ipvsadm，负责为ipvs内核框架编写规则，定义谁是集群服务，而谁是后端真实的服务器(Real Server)

详细的LVS的介绍可以参考[LVS详解](https://www.cnblogs.com/liqing1009/p/8763045.html).

## 13. 微服务架构是什么样子的?

通常传统的项目体积庞大，需求、设计、开发、测试、部署流程固定。新功能需要在原项目上做修改。

但是微服务可以看做是对大项目的拆分，是在快速迭代更新上线的需求下产生的。新的功能模块会发布成新的服务组件，与其他已发布的服务组件一同协作。 服务内部有多个生产者和消费者，通常以http rest的方式调用，服务总体以一个（或几个）服务的形式呈现给客户使用。

微服务架构是一种思想对微服务架构我们没有一个明确的定义，但简单来说微服务架构是：

采用一组服务的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻量级交互机制来通信，例如 RPC、HTTP 等，服务可独立扩展伸缩，每个服务定义了明确的边界，不同的服务甚至可以采用不同的编程语言来实现，由独立的团队来维护。

Golang的微服务框架[kit](https://gokit.io/)中有详细的微服务的例子,可以参考学习.

微服务架构设计包括：

1. 服务熔断降级限流机制 熔断降级的概念(Rate Limiter 限流器,Circuit breaker 断路器).
2. 框架调用方式解耦方式 Kit 或 Istio 或 Micro 服务发现(consul zookeeper kubeneters etcd ) RPC调用框架.
3. 链路监控,zipkin和prometheus.
4. 多级缓存.
5. 网关 (kong gateway).
6. Docker部署管理 Kubenetters.
7. 自动集成部署 CI/CD 实践.
8. 自动扩容机制规则.
9. 压测 优化.
10. Trasport 数据传输(序列化和反序列化).
11. Logging 日志.
12. Metrics 指针对每个请求信息的仪表盘化.

微服务架构介绍详细的可以参考:

* [Microservice Architectures](http://www.pst.ifi.lmu.de/Lehre/wise-14-15/mse/microservice-architectures.pdf)

## 14. 分布式锁实现原理，用过吗？

## 15. Etcd怎么实现分布式锁?

首先思考下Etcd是什么？可能很多人第一反应可能是一个键值存储仓库，却没有重视官方定义的后半句，用于配置共享和服务发现。

A highly-available key value store for shared configuration and service discovery.

实际上，etcd 作为一个受到 ZooKeeper 与 doozer 启发而催生的项目，除了拥有与之类似的功能外，更专注于以下四点。

* 简单：基于 HTTP+JSON 的 API 让你用 curl 就可以轻松使用。
* 安全：可选 SSL 客户认证机制。
* 快速：每个实例每秒支持一千次写操作。
* 可信：使用 Raft 算法充分实现了分布式。

但是这里我们主要讲述Etcd如何实现分布式锁?

## 16. Redis的数据结构有哪些，以及实现场景?

Redis的数据结构有五种:

### string 字符串

字符串类型是Redis中最为基础的数据存储类型，该类型可以接受任何格式的数据，如JPEG

图像数据或Json对象描述信息等。

在Redis中字符串类型的Value最多可以容纳的数据长度是512M。

·如果设置的键不存在则为添加，如果设置的键已经存在则修改。

使用 Strings 类型，可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受 Redis 的定时持久化（可以选择 RDB 模式或者 AOF 模式），操作日志及 Replication 等功能。

除了提供与 Memcached 一样的 get、set、incr、decr 等操作外，Redis 还提供了下面一些操作：

1. LEN niushuai：O(1)获取字符串长度.
2. APPEND niushuai redis：往字符串 append 内容，而且采用智能分配内存（每次2倍）.
3. 设置和获取字符串的某一段内容.
4. 设置及获取字符串的某一位（bit）.
5. 批量设置一系列字符串的内容.
6. 原子计数器.
7. GETSET 命令的妙用，请于清空旧值的同时设置一个新值，配合原子计数器使用.

### Hash 字典

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象

在 Memcached 中，我们经常将一些结构化的信息打包成 hashmap，在客户端序列化后存储为一个字符串的值（一般是 JSON 格式），比如用户的昵称、年龄、性别、积分等。这时候在需要修改其中某一项时，通常需要将字符串（JSON）取出来，然后进行反序列化，修改某一项的值，再序列化成字符串（JSON）存储回去。简单修改一个属性就干这么多事情，消耗必定是很大的，也不适用于一些可能并发操作的场合（比如两个并发的操作都需要修改积分）。而 Redis 的 Hash 结构可以使你像在数据库中 Update 一个属性一样只修改某一项属性值。

Hash可以用来存储、读取、修改用户属性。

### List 列表

列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）

或者尾部（右边）。

List本质是个链表，List的元素是有序的，元素的值可以重复.

Redis 还提供了操作 List 中某一段元素的 API，你可以直接查询，删除 List 中某一段的元素。

List 列表应用:

1. 微博 TimeLine.
2. 消息队列.

### Set集合

Set 就是一个集合，集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Set 数据结构，可以存储一些集合性的数据。比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。因为 Redis 非常人性化的为集合提供了求交集、并集、差集等操作，那么就可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

Set 集合应用:

1. 共同好友、二度好友
2. 利用唯一性，可以统计访问网站的所有独立 IP.
3. 好友推荐的时候，根据 tag 求交集，大于某个 threshold 就可以推荐.

### Sorted Set有序集合

和Sets相比，Sorted Sets是将 Set 中的元素增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，比如一个存储全班同学成绩的 Sorted Sets，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用 Sorted Sets 来做带权重的队列，比如普通消息的 score 为1，重要消息的 score 为2，然后工作线程可以选择按 score 的倒序来获取工作任务。让重要的任务优先执行。

Sorted Set有序集合应用:

1.带有权重的元素，比如一个游戏的用户得分排行榜.

2.比较复杂的数据结构，一般用到的场景不算太多.

### Redis 其他功能使用场景:

* 订阅-发布系统

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

* 事务——Transactions

谁说 NoSQL 都不支持事务，虽然 Redis 的 Transactions 提供的并不是严格的 ACID 的事务（比如一串用 EXEC 提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个 Transactions 还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间有会有其它客户端命令插进来执行）。Redis 还提供了一个 Watch 功能，你可以对一个 key 进行 Watch，然后再执行 Transactions，在这过程中，如果这个 Watched 的值进行了修改，那么这个 Transactions 会发现并拒绝执行。

## 17. Mysql高可用方案有哪些?

Mysql高可用方案包括:

1. 主从复制方案

这是MySQL自身提供的一种高可用解决方案，数据同步方法采用的是MySQL replication技术。MySQL replication就是从服务器到主服务器拉取二进制日志文件，然后再将日志文件解析成相应的SQL在从服务器上重新执行一遍主服务器的操作，通过这种方式保证数据的一致性。为了达到更高的可用性，在实际的应用环境中，一般都是采用MySQL replication技术配合高可用集群软件keepalived来实现自动failover，这种方式可以实现95.000%的SLA。

1. MMM/MHA高可用方案

MMM提供了MySQL主主复制配置的监控、故障转移和管理的一套可伸缩的脚本套件。在MMM高可用方案中，典型的应用是双主多从架构，通过MySQL replication技术可以实现两个服务器互为主从，且在任何时候只有一个节点可以被写入，避免了多点写入的数据冲突。同时，当可写的主节点故障时，MMM套件可以立刻监控到，然后将服务自动切换到另一个主节点，继续提供服务，从而实现MySQL的高可用。

1. Heartbeat/SAN高可用方案

在这个方案中，处理failover的方式是高可用集群软件Heartbeat，它监控和管理各个节点间连接的网络，并监控集群服务，当节点出现故障或者服务不可用时，自动在其他节点启动集群服务。在数据共享方面，通过SAN（Storage Area Network）存储来共享数据，这种方案可以实现99.990%的SLA。

1. Heartbeat/DRBD高可用方案

这个方案处理failover的方式上依旧采用Heartbeat，不同的是，在数据共享方面，采用了基于块级别的数据同步软件DRBD来实现。DRBD是一个用软件实现的、无共享的、服务器之间镜像块设备内容的存储复制解决方案。和SAN网络不同，它并不共享存储，而是通过服务器之间的网络复制数据。

1. NDB CLUSTER高可用方案

国内用NDB集群的公司非常少，貌似有些银行有用。NDB集群不需要依赖第三方组件，全部都使用官方组件，能保证数据的一致性，某个数据节点挂掉，其他数据节点依然可以提供服务，管理节点需要做冗余以防挂掉。缺点是：管理和配置都很复杂，而且某些SQL语句例如join语句需要避免。

## 18. Go语言的栈空间管理是怎么样的?

### 线程栈介绍

在我们开始讨论Go语言的栈之前，让我们来看看C语言是怎么管理栈的。

当你在C语言里启动一个线程的时候，标准库（standard library）会负责分配一块内存来用作线程的栈空间。它首先分配一块内存，告诉内核它的地址，然后让内核来控制线程的运行。如果这块分配的内存空间不够大的话，问题就变得复杂起来了。

为了避免递归导致的占空间耗尽这个问题，我们可以调整标准库分配给栈的内存空间的大小。但是增大这个参数会导致所有的线程都占用那么多的栈空间，即使这些函数并不需要递归调用。在这种情况下，虽然你的程序没有用到分配的栈，它还是会耗尽所有的内存。

另外一个解决办法是给每个线程分配不同大小的栈。这样你就需要给每个线程配置栈的大小，从而使得创建线程变得更加麻烦。想要决定一个线程会使用多少内存通常是非常困难的。

### Go语言的解决办法

Go语言的运行环境（runtime）尝试在goroutine需要的时候动态地分配栈空间，而不是给每个goroutine分配固定大小的内存空间。这样就避免了需要程序员来决定栈的大小。Go的开发小组正尝试从一种解决方案切换到另外一种解决方案。接下来将会讨论老的解决方案和它的缺点，然后介绍新的方案以及选择它的原因。

### 分块式的栈（Segmented stacks）

分块式的栈是最初Go语言组织栈的方式。当创建一个goroutine的时候，它会分配一个8KB的内存空间来给goroutine的栈使用。

我们最感兴趣的是当这8KB的栈空间被用完的时候。为了处理这种情况，每个Go函数的开头都有一小段检测代码。这段代码会检查我们是否已经用完了分配的栈空间。如果是的话，它会调用morestack函数。morestack函数分配一块新的内存作为栈空间，并且在这块栈空间的底部填入各种信息（包括之前的那块栈地址）。在分配了这块新的栈空间之后，它会重试刚才造成栈空间不足的函数。这个过程叫做栈分裂（stack split）。当经过栈分裂之后，栈结构如下图所示。

在新分配的栈底部，还插入了一个叫做lessstack的函数指针。这个函数还没有被调用。这样设置是为了从刚才造成栈空间不足的那个函数返回时做准备的。当我们从那个函数返回时，它会跳转到lessstack。lessstack函数会查看在栈底部存放的数据结构里的信息，然后调整栈指针（stack pointer）。这样就完成了从新的栈块到老的栈块的跳转。接下来，新分配的这个块栈空间就可以被释放掉了。

#### 分块式的栈的问题

分块式的栈让我们能够按照需求来扩展和收缩栈的大小。程序员不需要花精力去估计goroutine会用到多大的栈。创建一个新的goroutine的开销也不大。当程序员不知道栈会扩展到多少大时，它也能很好的处理这种情况。

这一直是之前Go语言管理栈的的方法。但这个方法有一个问题。缩减栈空间是一个开销相对较大的操作。如果在一个循环里有栈分裂，那么它的开销就变得不可忽略了。一个函数会扩展，然后分裂栈。当它返回的时候又会释放之前分配的内存块。如果这些都发生在一个循环里的话，代价是相当大的。

这就是所谓的**热分裂问题（hot split problem）**。它是Go语言开发者选择新的栈管理方法的主要原因。新的方法叫做**栈复制法（stack copying）**。

### 栈复制法（stack copying）

栈复制法一开始和分块式的栈很像。当goroutine运行并用完栈空间的时候，与之前的方法一样，栈溢出检查会被触发。但是，不像之前的方法那样分配一个新的内存块并链接到老的栈内存块，新的方法会分配一个两倍大的内存块并把老的内存块内容复制到新的内存块里。这样做意味着当栈缩减回之前大小时，我们不需要做任何事情。栈的缩减没有任何代价。而且，当栈再次扩展时，运行环境也不需要再做任何事。它可以重用之前分配的空间。

#### 栈是如何被复制的？

栈的复制听起来很容易，但实际操作并非那么简单。存储在栈上的变量的地址可能已经被使用到。也就是说程序使用到了一些指向栈的指针。当移动栈的时候，所有指向栈里内容的指针都会变得无效。幸运的是，指向栈内容的指针自身也必定是保存在栈上的。这是为了保证内存安全的必要条件。否则一个程序就有可能访问一段已经无效的栈空间了。

因为垃圾回收的需要，我们必须知道栈的哪些部分是被用作指针了。当我们移动栈的时候，我们可以更新栈里的指针让它们指向新的地址。所有相关的指针都会被更新。我们使用了垃圾回收的信息来复制栈，但并不是任何使用栈的函数都有这些信息。因为很大一部分运行环境是用C语言写的，很多被调用的运行环境里的函数并没有指针的信息，所以也就不能够被复制了。当遇到这种情况时，我们只能退回到分块式的栈并支付相应的开销。（注：这部分信息有点过时了，但还是值得一读！）

这也是为什么现在运行环境的开发者正在用Go语言重写运行环境的大部分代码。无法用Go语言重写的部分（比如调度器的核心代码和垃圾回收器）会在特殊的栈上运行。这个特殊栈的大小由运行环境的开发者设置。

这些改变除了使栈复制成为可能，它也允许我们在将来实现并行垃圾回收。

### 再说一下虚拟内存

还有一种处理栈空间的办法是分配很大一块虚拟内存。因为只有在内存地址被访问到的时候才会真正分配物理内存，似乎我们可以简单地分配一块很大的虚拟内存然后让操作系统来完成剩下的工作。但是这个方法有几个问题。

首先，32位的系统只有4GB的虚拟内存，而通常只有其中的3GB可以被应用程序使用。创建上百万的goroutine也不是不常见，这时你很可能会用完所有的虚拟内存（即使我们假设栈只用到8KB的空间）。

其次，即使我们可以在64位的系统里分配大量的虚拟内存，它依赖过量使用（overcommitting）内存。过量使用是指我们分配比实际物理内存空间更多的虚拟内存，并且依赖操作系统来确保能够分配到需要的物理内存。但是过量使用虚拟内存是存在一定风险的。因为一个进程真的使用了比实际物理内存更大的内存空间时，它需要开始为新的需求腾出可用的物理空间。它通常会把一块内存里的内容保存到磁盘上。这样会导致延迟不可预测。因为这个原因，我们通常不在系统里过量使用内存。

## 19. Goroutine和Channel的作用分别是什么?

进程是内存资源管理和cpu调度的执行单元。为了有效利用多核处理器的优势，将进程进一步细分，允许一个进程里存在多个线程，这多个线程还是共享同一片内存空间，但cpu调度的最小单元变成了线程。

那协程又是什么呢，以及与线程的差异性??

协程，可以看作是轻量级的线程。但与线程不同的是，线程的切换是由操作系统控制的，而协程的切换则是由用户控制的。

Go中的goroutinue就是协程

多个goroutine之间的通信是通过channel

## 20. 怎么查看Goroutine的数量?

GOMAXPROCS中控制的是未被阻塞的所有Goroutine,可以被Multiplex到多少个线程上运行,通过GOMAXPROCS可以查看Goroutine的数量。

https://juejin.im/post/6844903662553137165

## 21. 怎么限制Goroutine的数量.

在Golang中，Goroutine虽然很好，但是数量太多了，往往会带来很多麻烦，比如耗尽系统资源导致程序崩溃，或者CPU使用率过高导致系统忙不过来。所以我们可以限制下Goroutine的数量,这样就需要在每一次执行go之前判断goroutine的数量，如果数量超了，就要阻塞go的执行。第一时间想到的就是使用通道。每次执行的go之前向通道写入值，直到通道满的时候就阻塞了，

使用带缓冲的通道实现，通道缓冲大小就是限制的goroutine的数量

## 22. 滑动窗口的概念以及应用?

## 23. 怎么做弹性扩缩容，原理是什么?

## 24. 让你设计一个web框架，你要怎么设计，说一下步骤.

## 25. 说一下中间件原理.

中间件（middleware）是基础软件的一大类，属于可复用软件的范畴。中间件处于操作系统软件与用户的应用软件的中间。中间件在操作系统、网络和数据库之上，应用软件的下层，总的作用是为处于自己上层的应用软件提供运行与开发的环境，帮助用户灵活、高效地开发和集成复杂的应用软件  IDC的定义是：中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。

中间件解决的问题是：

在中间件产生以前，应用软件直接使用操作系统、网络协议和数据库等开发，这些都是计算机最底层的东西，越底层越复杂，开发者不得不面临许多很棘手的问题，如操作系统的多样性，繁杂的网络程序设计、管理，复杂多变的网络环境，数据分散处理带来的不一致性问题、性能和效率、安全，等等。这些与用户的业务没有直接关系，但又必须解决，耗费了大量有限的时间和精力。于是，有人提出能不能将应用软件所要面临的共性问题进行提炼、抽象，在操作系统之上再形成一个可复用的部分，供成千上万的应用软件重复使用。这一技术思想最终构成了中间件这类的软件。中间件屏蔽了底层操作系统的复杂性，使程序开发人员面对一个简单而统一的开发环境，减少程序设计的复杂性，将注意力集中在自己的业务上，不必再为程序在不同系统软件上的移植而重复工作，从而大大减少了技术上的负担。

## 26. 怎么设计orm，让你写,你会怎么写?

## 27. 用过原生的http包吗？

## 28. 一个非常大的数组，让其中两个数想加等于1000怎么算?

## 29. 各个系统出问题怎么监控报警.

## 30. 常用测试工具，压测工具，方法?

goconvey,vegeta

## 31. 复杂的单元测试怎么测试，比如有外部接口mysql接口的情况

## 32. redis集群，哨兵，持久化，事务

## 33. mysql和redis区别是什么？

## 34. 高可用软件是什么？

## 35. 怎么搞一个并发服务程序？

## 36. 讲解一下你做过的项目，然后找问题问实现细节。

## 37. mysql事务说下。

## 38. 怎么做一个自动化配置平台系统？

## 39. grpc遵循什么协议？

## 40. grpc内部原理是什么？

## 41. http2的特点是什么，与http1.1的对比。

## 42. go struct能不能比较

·同一个struct的2个实例能不能比较 ==  !=

如果struct类型中成员变量不带不能比较的类型就可以比较，如果带有不能比较的类型就不能比较

·两个不同的struct的实例能不能比较 ==  !=

可以强制转换，强转之后可以比较（如果成员变量含有不可比较成员变量，即使可以强制转换，也不可以比较）

### 补充1：golang官网对于所有类型是否可以比较的说明

https://golang.org/ref/spec#Comparison\_operators

等号运算符==和！=适用于可比较的操作数。 排序运算符<，<=，>和> =适用于已排序的操作数。 这些术语和比较结果定义如下：布尔值是可比较的。两个布尔值相等true或相等 false。

* 整数值是通常可比较且有序的。
* 浮点值是可比较的且有序，如IEEE-754标准所定义。
* 复数值是可比较的。如果和 u，v则两个复数值和相等。 real(u) == real(v)imag(u) == imag(v)
* 字符串值是可比较的且按字节顺序排序。
* 指针值是可比较的。如果两个指针值指向同一个变量，或者两个指针都具有value，则它们相等nil。指向不同[零大小](https://golang.org/ref/spec#Size_and_alignment_guarantees)变量的指针可以相等或可以不相等。
* 通道值是可比较的。如果两个通道值是通过相同的调用创建的，[make](https://golang.org/ref/spec#Making_slices_maps_and_channels) 或者两个通道都具有value，则它们是相等的 nil。
* 接口值是可比较的。如果两个接口值具有[相同的](https://golang.org/ref/spec#Type_identity)动态类型和相等的动态值，或者两个接口值都具有value，则它们是相等的nil。
* 值x非接口类型X和一个值t接口类型T是可比较的，当类型的值X是可比较的和 X器具T。如果t动态类型等于，X 并且t动态值等于，则它们相等x。
* 如果结构的所有字段都是可比较的，则它们的值是可比较的。如果两个结构值对应的非[空白](https://golang.org/ref/spec#Blank_identifier)字段相等，则它们相等。
* 如果数组元素类型的值可比较，则数组值可比较。如果两个数组的对应元素相等，则它们相等。

如果动态类型相同的两个接口值不具有可比性，则将它们进行比较会导致[运行时恐慌](https://golang.org/ref/spec#Run_time_panics)。此行为不仅适用于直接接口值比较，而且适用于将接口值或结构的数组与接口值字段进行比较。

切片，map和函数值不可比较。但是，在特殊情况下，可以将切片，映射或函数值与预先声明的标识符进行比较nil。指针，通道和接口值to的nil 比较也是允许的，并且遵循上面的一般规则。

### 补充2：“==”比较 规则

#### https://www.jianshu.com/p/a982807819fa

#### golang变量类型

 1，基本类型

* 整型，包括int，uint，int8，uint8，int16，uint16，int32，uint32，int64，uint64，byte，rune，uintptr等
* 浮点型，包括float32，float64
* 复数类型，包括complex64，complex128
* 字符串类型，string
* 布尔型，bool

 2，复合类型

* 数组
* struct结构体

 3，引用类型

* slice
* map
* channel
* pointer or 引用类型

 4，接口类型  
io.Reader, io.Writer,error等

#### 一，基本类型的变量比较

比较的两个变量类型必须相等。而且，golang没有隐式类型转换，比较的两个变量必须类型完全一样，类型别名也不行。如果要比较，先做类型转换再比较。

* 类型完全不一样的，不能比较
* 类型再定义关系，不能比较，可以强转比较
* 类型别名关系，可以比较

Float重点说明：float类型比较

同C/C++语言中一样，需要定义一个精度值来衡量

判断是否相等

If a-b < 精度(0.000000001)

则认为相等

判断谁大

If math.max(a,b) == a

则a大

应用

type Floater struct {

Accuracy float64 //精度,最大为小数点后15位

}

//是否相等

func (f Floater) IsEqual(a, b float64) bool {

return math.Abs(a-b) < f.Accuracy

}

//0为相等 1为a大于b -1为a小于b

func (f Floater) Bccomp(a, b float64) int8 {

if math.Abs(a-b) < f.Accuracy {

return 0

}

if math.Max(a, b) == a {

return 1

} else {

return -1

}

}

func main() {

f := Floater{Accuracy: 0.000000000001}

var a float64 = 1.0000000002

var b float64 = 1.0000000001

fmt.Println(f.Bccomp(a, b)) //1

fmt.Println(f.Bccomp(b, a)) //-1

fmt.Println(f.Bccomp(a, a)) //0

}

#### 二，复合类型的变量比较

复合类型是逐个字段，逐个元素比较的。需要注意的是，array 或者struct中每个元素必须要是可比较的，如果某个array的元素 or struct的成员不能比较(比如是后面介绍的slice，map等)，则此复合类型也不能比较。

##### 1，数组类型变量比较

* 数组的长度是类型的一部分，如果数组长度不同，无法比较
* 逐个元素比较类型和值。每个对应元素的比较遵循基本类型变量的比较规则。跟struct一样，如果item是不可比较的类型，则array也不能做比较。

##### 2，struct类型变量比较

该类型可以使用两个变量做 “==” 或 “！=” 比较，如果成员类型中包含了不可以比较的类型，使用该种比较方式程序编译时会报错，一般使用reflect中的DeepEqual接口比较。

·同一个struct的2个实例能不能比较 ==  !=

如果struct类型中成员变量不带不能比较的类型就可以比较，如果带有不能比较的类型就不能比较

·两个不同的struct的实例能不能比较 ==  !=

可以强制转换，强转之后可以比较（如果成员变量含有不可比较成员变量，即使可以强制转换，也不可以比较）

#### 三，引用类型的变量比较

##### 1，普通的变量引用类型&val和channel的比较规则

引用类型变量存储的是某个变量的内存地址。所以引用类型变量的比较，判断的是这两个引用类型存储的是不是同一个变量。

* 如果是同一个变量，则内存地址肯定也一样，则引用类型变量相等，用"=="判断为true
* 如果不是同一个变量，则内存地址肯定不一样，"=="结果为false

##### 2，slice这种引用类型的比较

slice类型不可比较，只能与零值nil做比较。

##### 3， map类型的比较

map类型和slice一样，不能比较，只能与nil做比较。

#### 四，interface{}类型变量的比较

接口类型的变量，包含该接口变量存储的值和值的类型两部分组成，分别称为接口的动态类型和动态值。只有动态类型和动态值都相同时，两个接口变量才相同:

而且接口的动态类型必须要是可比较的，如果不能比较(比如slice，map)，则运行时会报panic。因为编译器在编译时无法获取接口的动态类型，所以编译能通过，但是运行时直接panic:

#### 五，函数类型的比较

golang的func作为一等公民，也是一种类型，而且不可比较

#### 六，slice和map的特殊比较

map和slice是不可比较类型，使用下面的特殊方法来对slice和map做比较

##### 1，[]byte类型的变量，使用工具包byte提供的函数就可以做比较

s1 := []byte{'f', 'o', 'o'}

s2 := []byte{'f', 'o', 'o'}

fmt.Println(bytes.Equal(s1, s2)) // true

s2 = []byte{'b', 'a', 'r'}

fmt.Println(bytes.Equal(s1, s2)) // false

##### 2，使用反射

reflect.DeepEqual函数可以用来比较两个任意类型的变量

对map类型做比较：

m1 := map[string]int{"foo": 1, "bar": 2}

m2 := map[string]int{"foo": 1, "bar": 2}

// fmt.Println(m1 == m2) // map can only be compared to nil

fmt.Println(reflect.DeepEqual(m1, m2)) // true  
  
对slice类型做比较：

s := []string{"foo"}

fmt.Println(reflect.DeepEqual(s, []string{"foo"})) // true

fmt.Println(reflect.DeepEqual(s, []string{"bar"})) // false

对struct类型做比较：

type T struct {

name string

Age int

}

t := T{"foo", 10}

fmt.Println(reflect.DeepEqual(t, T{"bar", 20})) // false

fmt.Println(reflect.DeepEqual(t, T{"bar", 10})) // false

fmt.Println(reflect.DeepEqual(t, T{"foo", 10})) // true

##### 3. 使用google的cmp包

用例：

import (

"fmt"

"github.com/google/go-cmp/cmp"

)

type T struct {

Name string

Age int

City string

}

func main() {

x := T{"Michał", 99, "London"}

y := T{"Adam", 88, "London"}

if diff := cmp.Diff(x, y); diff != "" {

fmt.Println(diff)

}

}

结果：

main.T{

- Name: "Michał",

+ Name: "Adam",

- Age: 99,

+ Age: 88,

City: "London",

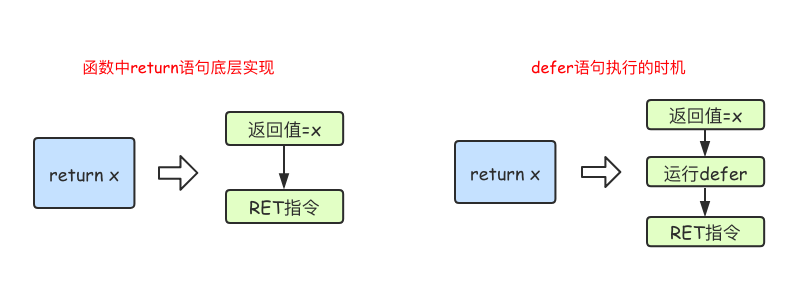
}

#### 总结：

* 1，复合类型，只有每个元素(成员)可比较，而且类型和值都相等时，两个复合元素才相等
* 2，slice，map不可比较，但是可以用reflect或者cmp包来比较
* 3，func作为golnag的一等公民，也是一个类型，也不能比较。
* 4，引用类型的比较是看指向的是不是同一个变量
* 5，类型再定义(type A string)不可比较，是两种不同的类型
* 6，类型别名(type A = string)可比较，是同一种类型。

## 43. go defer（for defer）

Defer执行时机



三条使用规则

规则一 当defer被声明时，其参数就会被实时解析

规则二 defer执行顺序为先进后出

规则三 defer可以读取有名返回值

## 44. select可以用于什么?

Go的select主要是处理多个channel的操作.

## 45.context包的用途是什么?

## 46. client如何实现长连接?

## 47. 实现set

根据go中map的keys的无序性和唯一性，可以将其作为set

## 48.实现消息队列（多生产者，多消费者）

根据Goroutine和channel的读写可以实现消息队列，、

https://blog.csdn.net/phpduang/article/details/80143626

## 49. 大文件排序

https://blog.csdn.net/michellechouu/article/details/47002393

## 50. 基本排序，哪些是稳定的

选择排序、快速排序、希尔排序、堆排序不是稳定的排序算法，

冒泡排序、插入排序、归并排序和基数排序是稳定的排序算法

## 51. Http get跟head

get:获取由Request-URI标识的任何信息(以实体的形式)，如果Request-URI引用某个数据处理过程，则应该以它产生的数据作为在响应中的实体，而不是该过程的源代码文本，除非该过程碰巧输出该文本。

head: 除了服务器不能在响应中返回消息体，HEAD方法与GET相同。用来获取暗示实体的元信息，而不需要传输实体本身。常用于测试超文本链接的有效性、可用性和最近的修改。

## 52.Http 401,403

401 Unauthorized： 该HTTP状态码表示认证错误，它是为了认证设计的，而不是为了授权设计的。收到401响应，表示请求没有被认证—压根没有认证或者认证不正确—但是请重新认证和重试。（一般在响应头部包含一个*WWW-Authenticate*来描述如何认证）。通常由web服务器返回，而不是web应用。从性质上来说是临时的东西。（服务器要求客户端重试）

403 Forbidden：该HTTP状态码是关于授权方面的。从性质上来说是永久的东西，和应用的业务逻辑相关联。它比401更具体，更实际。收到403响应表示服务器完成认证过程，但是客户端请求没有权限去访问要求的资源。

总的来说，401 Unauthorized响应应该用来表示缺失或错误的认证；403 Forbidden响应应该在这之后用，当用户被认证后，但用户没有被授权在特定资源上执行操作。

## 53.Http keep-alive

## 54.Http能不能一次连接多次请求，不等后端返回

## 55.TCP 和 UDP 有什么区别,适用场景

* TCP 是面向连接的，UDP 是面向无连接的；故 TCP 需要建立连接和断开连接，UDP 不需要。
* TCP 是流协议，UDP 是数据包协议；故 TCP 数据没有大小限制，UDP 数据报有大小限制（UDP 协议本身限制、数据链路层的 MTU、缓存区大小）。
* TCP 是可靠协议，UDP 是不可靠协议；故 TCP 会处理数据丢包重发以及乱序等情况，UDP 则不会处理。

### UDP 的特点及使用场景:

UDP 不提供复杂的控制机制，利用 IP 提供面向无连接的通信服务，随时都可以发送数据，处理简单且高效，经常用于以下场景：

包总量较小的通信（DNS、SNMP）

视频、音频等多媒体通信（即时通信）

广播通信

### TCP 的特点及使用场景:

相对于 UDP，TCP 实现了数据传输过程中的各种控制，可以进行丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

在对可靠性要求较高的情况下，可以使用 TCP，即不考虑 UDP 的时候，都可以选择 TCP。

## 56.Golang 里的逃逸分析是什么？怎么避免内存逃逸？

## 57.配置中心如何保证一致性？

## 58.Golang 的GC触发时机是什么?

## 59.Redis 里数据结构的实现熟悉吗?

## 60.Etcd的Raft一致性算法原理?

## 61.微服务概念.

## 62.SLB原理.

## 63.分布式一直性原则.

如何保证服务宕机造成的分布式服务节点处理问题?

## 65.服务发现怎么实现的.

## Golang面经

https://www.nowcoder.com/discuss/145338

# 网址三

<https://www.jishuchi.com/read/go-interview/3435>

## 1. 在go语言中，Printf()、Sprintf()、Fprintf()函数的区别用法是什么？

都是把格式好的字符串输出，只是输出的目标不一样：

Printf()，是把格式字符串输出到标准输出（一般是屏幕，可以重定向）。

Sprintf()，是把格式字符串输出到指定字符串中，所以参数比printf多一个char\*。那就是目标字符串地址。

Fprintf()， 是把格式字符串输出到指定文件设备中，所以参数笔printf多一个文件指针FILE\*。主要用于文件操作。Fprintf()是格式化输出到一个stream，通常是到文件。

## 2．说说go语言中，数组与切片的区别？

(1). 数组

1.数组是具有固定长度且拥有零个或者多个相同数据类型元素的序列。

2.数组的长度是数组类型的一部分，所以[3]int 和 [4]int 是两种不同的数组类型。数组需要指定大小，不指定也会根据初始化的自动推算出大小，不可改变 ;

3.数组是值传递;

数组是内置(build-in)类型,是一组同类型数据的集合，它是值类型，通过从0开始的下标索引访问元素值。在初始化后长度是固定的，无法修改其长度。当作为方法的参数传入时将复制一份数组而不是引用同一指针。数组的长度也是其类型的一部分，通过内置函数len(array)获取其长度。

(2). 切片

1.切片表示一个拥有相同类型元素的可变长度的序列。 切片是一种轻量级的数据结构，它有三个属性：指针、长度和容量。

2.切片不需要指定大小;

3.切片是地址传递;

切片可以通过数组来初始化，也可以通过内置函数make()初始化 .初始化时len=cap,在追加元素时如果容量cap不足时将按len的2倍扩容；

## 3.解释以下命令的作用？

go env: #用于查看go的环境变量

go run: #用于编译并运行go源码文件

go build: #用于编译源码文件、代码包、依赖包

go get: #用于动态获取远程代码包

go install: #用于编译go文件，并将编译结构安装到bin、pkg目录

go clean: #用于清理工作目录，删除编译和安装遗留的目标文件

go version: #用于查看go的版本信息

## 4.go语言中没有隐藏的this指针，这句话是什么意思？

方法施加的对象显式传递，没有被隐藏起来

golang的面向对象表达更直观，对于面向过程只是换了一种语法形式来表达

方法施加的对象不需要非得是指针，也不用非得叫this

## 5. go语言触发异常的场景有哪些？

A. 空指针解析

B. 下标越界

C. 除数为0

D. 调用panic函数

## 6. 说说go语言的beego框架？

A. beego是一个golang实现的轻量级HTTP框架

B. beego可以通过注释路由、正则路由等多种方式完成url路由注入

C. 可以使用bee new工具生成空工程，然后使用bee run命令自动热编译

## 7. 说说go语言的goconvey框架？

A. goconvey是一个支持golang的单元测试框架

B. goconvey能够自动监控文件修改并启动测试，并可以将测试结果实时输出到web界面

C. goconvey提供了丰富的断言简化测试用例的编写

## 8. go语言中，GoStub的作用是什么？

A. GoStub可以对全局变量打桩

B. GoStub可以对函数打桩

C. GoStub不可以对类的成员方法打桩

D. GoStub可以打动态桩，比如对一个函数打桩后，多次调用该函数会有不同的行为

## 9.进程线程协程之间的区别

进程切换需要的资源很最大，效率很低 线程切换需要的资源一般，效率一般 协程切换任务资源很小，效率高 多进程、多线程根据cpu核数不一样可能是并行的 也可能是并发的。

协程的本质就是使用当前进程在不同的函数代码中切换执行，可以理解为并行。 协程是一个用户层面的概念，不同协程的模型实现可能是单线程，也可能是多线程。

进程拥有自己独立的堆和栈，既不共享堆，亦不共享栈，进程由操作系统调度。

线程拥有自己独立的栈和共享的堆，共享堆，不共享栈，线程亦由操作系统调度(标准线程是这样的)。

协程和线程一样共享堆，不共享栈，协程由程序员在协程的代码里显示调度。

一个应用程序一般对应一个进程，一个进程一般有一个主线程，还有若干个辅助线程，线程之间是平行运行的，在线程里面可以开启协程，让程序在特定的时间内运行。

协程和线程的区别是：协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。

## 10. 以下代码有什么问题，说明原因。

1. type student struct {
2. Name string Age int
3. }
4. func pase\_student() {
5. m := make(map[string] \* student)
6. stus := [] student {
7. {
8. Name: "zhou",
9. Age: 24
10. }, {
11. Name: "li",
12. Age: 23
13. }, {
14. Name: "wang",
15. Age: 22
16. },
17. }
18. for \_, stu := range stus {
19. m[stu.Name] = &stu
20. }
21. }

考点：**foreach**

解答：这样的写法初学者经常会遇到的，很危险！与Java的foreach一样，都是使用副本的方式。所以m[stu.Name]=&stu实际上一致指向同一个指针，最终该指针的值为遍历的最后一个struct的值拷贝。就像想修改切片元素的属性：

1. for \_, stu := range stus {
2. stu.Age = stu.Age + 10
3. }

也是不可行的。大家可以试试打印出来：

1. func pase\_student() {
2. m := make(map[string] \* student)
3. stus := [] student {
4. {
5. Name: "zhou",
6. Age: 24
7. }, {
8. Name: "li",
9. Age: 23
10. }, {
11. Name: "wang",
12. Age: 22
13. },
14. }
15. // 错误写法
16. for \_, stu := range stus {
17. m[stu.Name] = & stu
18. }
19. for k, v := range m {
20. println(k, "=>", v.Name)
21. }
22. // 正确
23. for i := 0; i < len(stus); i++ {
24. m[stus[i].Name] = & stus[i]
25. }
26. for k, v := range m {
27. println(k, "=>", v.Name)
28. }
29. }

## 11. 下面的代码会输出什么，并说明原因

https://juejin.im/post/6844903662553137165

Func main(){

runtime.GOMAXPROCS(1)

wg := sync.WaitGroup {}

wg.Add(20)

for i := 0; i < 10; i++ {

go func() {

fmt.Println("A: ", i)

wg.Done()

}()

}

for i := 0; i < 10; i++ {

go func(i int) {

fmt.Println("B: ", i)

wg.Done()

}(i)

}

wg.Wait()

}

考点：**go执行的随机性和闭包**

解答：谁也不知道执行后打印的顺序是什么样的，所以只能说是随机数字。但是A:均为输出10，B:从0~9输出(顺序不定)。第一个go func中i是外部for的一个变量，地址不变化。遍历完成后，最终i=10。故go func执行时，i的值始终是10。

第二个go func中i是函数参数，与外部for中的i完全是两个变量。尾部(i)将发生值拷贝，go func内部指向值拷贝地址。

## 12. 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()

}

考点：go的组合继承

解答：这是Golang的组合模式，可以实现OOP的继承。被组合的类型People所包含的方法虽然升级成了外部类型Teacher这个组合类型的方法（一定要是匿名字段），但它们的方法(ShowA())调用时接受者并没有发生变化。此时People类型并不知道自己会被什么类型组合，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。

showAshowB

## 13. 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)

}

}

考点：select随机性

解答： select会随机选择一个可用通用做收发操作。所以代码是有b能触发异常，也有可能不会。单个chan如果无缓冲时，将会阻塞。但结合 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。

## 14. 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

}

考点：defer执行顺序

解答：这道题类似第1题需要注意到defer执行顺序和值传递 index:1肯定是最后执行的，但是index:1的第三个参数是一个函数，所以最先被调用calc("10",1,2)==>10,1,2,3

执行index:2时,与之前一样，需要先调用calc("20",0,2)==>20,0,2,2

执行到b=1时候开始调用，index:2==>calc("2",0,2)==>2,0,2,2

最后执行index:1==>calc("1",1,3)==>1,1,3,4

10 1 2 3

20 0 2 2

2 0 2 2

1 1 3 4

## 15. 请写出以下输入内容

1. func main() {
2. s := make([] int, 0)
3. s = append(s, 1, 2, 3)
4. fmt.Println(s)
5. }

考点：make默认值和append

解答： make初始化是由默认值的哦，此处默认值为0

输出[0 0 0 0 0 1 2 3]

大家试试改为:

1. s := make([] int, 0)

2. s = append(s, 1, 2, 3)

3. fmt.Println(s) //[1 2 3]

## 16. 下面的代码有什么问题?

1. type UserAges struct {
2. ages map[string]
3. int sync.Mutex
4. }
5. func(ua \* UserAges) Add(name string, age int) {
6. ua.Lock()
7. defer ua.Unlock()
8. ua.ages[name] = age
9. }
10. func(ua \* UserAges) Get(name string) int {
11. if age, ok := ua.ages[name]; ok {
12. return age
13. }
14. return -1
15. }

考点：**map线程安全**

解答：可能会出现fatal error: concurrent map read and map write. 修改一下看看效果

1. func(ua \* UserAges) Get(name string) int {
2. ua.Lock()
3. defer ua.Unlock()
4. if age, ok := ua.ages[name]; ok {
5. return age
6. }
7. return -1
8. }

## 17. 下面的迭代会有什么问题？

1. func(set \* threadSafeSet) Iter() <-chan interface {} {
2. ch := make(chan interface {})
3. go func() {
4. set.RLock()
5. for elem := range set.s {
6. ch <-elem
7. }
8. close(ch)
9. set.RUnlock()
10. }()
11. return ch
12. }

考点：**chan缓存池**

解答：看到这道题，我也在猜想出题者的意图在哪里。 chan?sync.RWMutex?go?chan缓存池?迭代? 所以只能再读一次题目，就从迭代入手看看。既然是迭代就会要求set.s全部可以遍历一次。但是chan是为缓存的，那就代表这写入一次就会阻塞。我们把代码恢复为可以运行的方式，看看效果

1. package main
2. import(
3. "sync"
4. "fmt"
5. )
6. //下面的迭代会有什么问题？
7. type threadSafeSet struct {
8. sync.RWMutex
9. s []interface {}
10. }
11. func(set \* threadSafeSet) Iter() <-chan interface {} {
12. // ch := make(chan interface{})
13. // 解除注释看看！
14. ch := make(chan interface {}, len(set.s))
15. go func() {
16. set.RLock()
17. for elem, value := range set.s {
18. ch <-elem
19. println("Iter:", elem, value)
20. }
21. close(ch)
22. set.RUnlock()
23. }()
24. return ch
25. }
26. func main() {
27. th := threadSafeSet {
28. s :[]interface {} {
29. "1", "2"
30. },
31. }
32. v := <-th.Iter()
33. fmt.Sprintf("%s%v", "ch", v)
34. }

## 18. 以下代码能编译过去吗？为什么？

1. package main
2. import (
3. "fmt"
4. )
5. type People interface {
6. Speak(string) string
7. }
8. type Stduent struct {}
9. func(stu \* Stduent) Speak(think string)(talk string) {
10. if think == "bitch" {
11. talk = "You are a good boy"
12. } else {
13. talk = "hi"
14. }
15. return
16. }
17. func main() {
18. var peo People = Stduent {}
19. think := "bitch"
20. fmt.Println(peo.Speak(think))
21. }

考点：**golang的方法集**

解答：编译不通过！做错了！？说明你对golang的方法集还有一些疑问。一句话：golang的方法集仅仅影响接口实现和方法表达式转化，与通过实例或者指针调用方法无关。

## 19. 以下代码打印出来什么内容，说出为什么。

1. package main
2. import(
3. "fmt"
4. )
5. type People interface {
6. Show()
7. }
8. type Student struct {}
9. func(stu \* Student) Show() {}
10. func live() People {
11. var stu \* Student
12. return stu
13. }
14. func main() {
15. if live() == nil {
16. fmt.Println("AAAAAAA")
17. } else {
18. fmt.Println("BBBBBBB")
19. }
20. }

考点：**interface内部结构**

解答：很经典的题！这个考点是很多人忽略的interface内部结构。 go中的接口分为两种一种是空的接口类似这样：

1. var in interface{}

另一种如题目：

1. type People interface {
2. Show()
3. }

他们的底层结构如下：

1. type eface struct {
2. //空接口
3. \_type \*\_type //类型信息
4. data unsafe.Pointer //指向数据的指针(go语言中特殊的指针类型unsafe.Pointer类似于c语言中的void\*)
5. }
6. type iface struct {
7. //带有方法的接口
8. tab \*itab //存储type信息还有结构实现方法的集合
9. data unsafe.Pointer //指向数据的指针(go语言中特殊的指针类型unsafe.Pointer类似于c语言中的void\*)}
10. type \_type struct {
11. size uintptr //类型大小
12. ptrdata uintptr //前缀持有所有指针的内存大小
13. hash uint32 //数据hash值
14. tflag tflag align uint8 //对齐
15. fieldalign uint8 //嵌入结构体时的对齐
16. kind uint8 //kind 有些枚举值kind等于0是无效的
17. alg \*typeAlg //函数指针数组，类型实现的所有方法
18. gcdata \*byte str nameOff ptrToThis typeOff
19. }
20. type itab struct {
21. inter \*interfacetype //接口类型
22. \_type \*\_type //结构类型
23. link \*itab bad int32 inhash int32 fun [1]uintptr //可变大小 方法集合
24. }

可以看出iface比eface 中间多了一层itab结构。 itab 存储\_type信息和[]fun方法集，从上面的结构我们就可得出，因为data指向了nil 并不代表interface 是nil，所以返回值并不为空，这里的fun(方法集)定义了接口的接收规则，在编译的过程中需要验证是否实现接口结果：

1. BBBBBBB