一、基础部分

1、golang 中 make 和 new 的区别？（基本必问）

共同点： 给变量分配内存

不同点：

1）作用变量类型不同，new给string,int和数组分配内存，make给切片，map，channel分配内存；

2）返回类型不一样，new返回指向变量的指针，make返回变量本身；

3）new 分配的空间被清零。make 分配空间后，会进行初始化；

\4) 字节的面试官还说了另外一个区别，就是分配的位置，在堆上还是在栈上？这块我比较模糊，大家可以自己探究下，我搜索出来的答案是golang会弱化分配的位置的概念，因为编译的时候会自动内存逃逸处理，懂的大佬帮忙补充下：make、new内存分配是在堆上还是在栈上？
（我个人认为，new内存分配在栈上，make内存分配在堆上）

2、数组和切片的区别 （基本必问）

相同点：

1)只能存储一组相同类型的数据结构

2)都是通过下标来访问，并且有容量长度，长度通过 len 获取，容量通过 cap 获取

区别：

1）数组是定长，访问和复制不能超过数组定义的长度，否则就会下标越界，切片长度和容量可以自动扩容

2）数组是值类型，切片是引用类型，每个切片都引用了一个底层数组，切片本身不能存储任何数据，都是这底层数组存储数据，所以修改切片的时候修改的是底层数组中的数据。切片一旦扩容，指向一个新的底层数组，内存地址也就随之改变

简洁的回答：

1）定义方式不一样 2）初始化方式不一样，数组需要指定大小，大小不改变 3）在函数传递中，数组切片都是值传递。

数组的定义

var a1 [3]int

var a2 […]int{1,2,3}

切片的定义

var a1 []int

var a2 :=make([]int,3,5)

数组的初始化

a1 := […]int{1,2,3}

a2 := [5]int{1,2,3}

切片的初始化

b:= make([]int,3,5)

3、for range 的时候它的地址会发生变化么？

答：在 for a,b := range c 遍历中， a 和 b 在内存中只会存在一份，即之后每次循环时遍历到的数据都是以值覆盖的方式赋给 a 和 b，a，b 的内存地址始终不变。由于有这个特性，for 循环里面如果开协程，不要直接把 a 或者 b 的地址传给协程。解决办法：在每次循环时，创建一个临时变量。

4、go defer，多个 defer 的顺序，defer 在什么时机会修改返回值？

作用：defer延迟函数，释放资源，收尾工作；如释放锁，关闭文件，关闭链接；捕获panic;

避坑指南：defer函数紧跟在资源打开后面，否则defer可能得不到执行，导致内存泄露。

多个 defer 调用顺序是 LIFO（后入先出），defer后的操作可以理解为压入栈中

defer，return，return value（函数返回值） 执行顺序：首先return，其次return value，最后defer。

【Golang】Go语言defer用法大总结(含return返回机制)__奶酪的博客-CSDN博客
blog.csdn.net/Cassie_zkq/article/details/108567205

有名返回值：

```
func b() (i int) {
defer func() {
i++
fmt.Println("defer2:", i)
}()
defer func() {
i++
fmt.Println("defer1:", i)
}()
return i //或者直接写成return
}
func main() {
fmt.Println("return:", b())
}
```

结果：

```
defer1:1
defer2:2
return:2
```

无名返回值：

```
func b() int {
defer func() {
i++
fmt.Println("defer2:", i)
}()
defer func() {
i++
fmt.Println("defer1:", i)
}()
return i
}
func main() {
fmt.Println("return:", b())
}
```

结果：
defer1:1
defer2:2
return:0
无名返回值：return之后的defer修改了实际返回值，但不影响前面的返回结果
有名返回值：return之后的的defer修改了实际返回值，前面的返回结果根据实时的变量值确定

函数返回指针

```
func c() *int {
var i int
defer func() {
i++
fmt.Println("defer2:", i)
}()
defer func() {
i++
fmt.Println("defer1:", i)
}()
return &i
}
func main() {
fmt.Println("return:", *(c()))
}
```

结果：
defer1：1
defer2：2
return：2

defer中有参函数无参函数和方法的区别

```
func test(a int) {//无返回值函数
defer fmt.Println("1、a =", a) //方法
defer func(v int) { fmt.Println("2、a =", v)} (a) //有参函数
defer func() { fmt.Println("3、a =", a)} () //无参函数
a++
}
func main() {
test(1)
}
```

结果为：
3、a=2
2、a=1
1、a=1
原因：defer后面如果跟方法和有参函数，则方法和有参函数会把用到的变量获取当前值并封入defer栈中；而无参函数只把该操作封入栈中，执行时再确定执行时这一刻的变量的值

5、uint 类型溢出问题

超过最大存储值如uint8最大是255

```
var a uint8 =255

var b uint8 =1
```

a+b = 0总之类型溢出会出现难以意料的事

整型的类型

类 型	有无符号	占用存储空间	表数范围	备注
int	有	32位系统4个字节 64位系统8个字节	$-2^{31} \sim 2^{31}-1$ $-2^{63} \sim 2^{63}-1$	
uint	无	32位系统4个字节 64位系统8个字节	$0 \sim 2^{32}-1$ $0 \sim 2^{64}-1$	
rune	有	与int32一样	$-2^{31} \sim 2^{31}-1$	等价 int32，表示一个 Unicode 码
byte	无	与 uint8 等价	$0 \sim 255$	当要存储字符时使用byte

6、能介绍下 rune 类型吗？

相当于int32

golang中的字符串底层实现是通过byte数组的，中文字符在unicode下占2个字节，在utf-8编码下占3个字节，而golang默认编码正好是utf-8

byte 等同于int8，常用来处理ascii字符

rune 等同于int32,常用来处理unicode或utf-8字符

ASCII码、Unicode码、UTF-8的区别：
（1）ASCII码：只包含127个字母，也就是大小写英文字母、数字和一些符号
（2）Unicode码：ASCII编码是1个字节，而Unicode编码通常是2个字节；Unicode码包含中文日文韩文等其他语言文字
（3）UTF-8：解决用Unicode码存储英文时浪费存储空间的问题。（UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间。）

具体参考：三种常见字符编码：ASCII、Unicode和UTF-8

7、 golang 中解析 tag 是怎么实现的？反射原理是什么？(中高级肯定会问，比较难，需要自己多去总结)

参考如下连接

golang中struct关于反射tag_paladinosment的博客-CSDN博客_golang 反射
tagblog.csdn.net/paladinosment/article/details/42570937

```
type User struct {
name string `json:name-field`
age int
}
func main() {
user := &User{"John Doe The Fourth", 20}
field, ok := reflect.TypeOf(user).Elem().FieldByName("name")
if !ok {
panic("Field not found")
}
```

```
fmt.Println(getStructTag(field))
}
func getStructTag(f reflect.StructField) string {
return string(f.Tag)
}
```

Go 中解析的 tag 是通过反射实现的，反射是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力或动态知道给定数据对象的类型和结构，并有机会修改它。反射将接口变量转换成反射对象 Type 和 Value；反射可以通过反射对象 Value 还原成原先的接口变量；反射可以用来修改一个变量的值，前提是这个值可以被修改；tag是啥:结构体支持标记，name string `json:name-field` 就是 `json:name-field` 这部分

gorm json yaml gRPC protobuf gin.Bind()都是通过反射来实现的

8、调用函数传入结构体时，应该传值还是指针？（Golang 都是传值）

Go 的函数参数传递都是值传递。所谓值传递：指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。参数传递还有引用传递，所谓引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数

因为 Go 里面的 map，slice，chan 是引用类型。变量区分值类型和引用类型。所谓值类型：变量和变量的值存在同一个位置。所谓引用类型：变量和变量的值是不同的位置，变量的值存储的是对值的引用。但并不是 map，slice，chan 的所有的变量在函数内都能被修改，不同数据类型的底层存储结构和实现可能不太一样，情况也就不一样。

9、讲讲 Go 的 slice 底层数据结构和一些特性？

答：Go 的 slice 底层数据结构是由一个 array 指针指向底层数组，len 表示切片长度，cap 表示切片容量。slice 的主要实现是扩容。对于 append 向 slice 添加元素时，假如 slice 容量够用，则追加新元素进去，slice.len++，返回原来的 slice。当原容量不够，则 slice 先扩容，扩容之后 slice 得到新的 slice，将元素追加进新的 slice，slice.len++，返回新的 slice。对于切片的扩容规则：当切片比较小时（容量小于 1024），则采用较大的扩容倍速进行扩容（新的扩容会是原来的 2 倍），避免频繁扩容，从而减少内存分配的次数和数据拷贝的代价。当切片较大的时（原来的 slice 的容量大于或者等于 1024），采用较小的扩容倍速（新的扩容将扩大大于或者等于原来 1.25 倍），主要避免空间浪费，网上其实很多总结的是 1.25 倍，那是在不考虑内存对齐的情况下，实际上还要考虑内存对齐，扩容是大于或者等于 1.25 倍。

（关于刚才问的 slice 为什么传到函数内可能被修改，如果 slice 在函数内没有出现扩容，函数外和函数内 slice 变量指向是同一个数组，则函数内复制的 slice 变量值出现更改，函数外这个 slice 变量值也会被修改。如果 slice 在函数内出现扩容，则函数内变量的值会新生成一个数组（也就是新的 slice，而函数外的 slice 指向的还是原来的 slice，则函数内的修改不会影响函数外的 slice。）

10、讲讲 Go 的 select 底层数据结构和一些特性？（难点，没有项目经常可能说不清，面试一般会问你项目中怎么使用select）

答：go 的 select 为 golang 提供了多路 IO 复用机制，和其他 IO 复用一样，用于检测是否有读写事件是否 ready。linux 的系统 IO 模型有 select，poll，epoll，go 的 select 和 linux 系统 select 非常相似。

select 结构组成主要是由 case 语句和执行的函数组成 select 实现的多路复用是：每个线程或者进程都先到注册和接受的 channel（装置）注册，然后阻塞，然后只有一个线程在运输，当注册的线程和进程准备好数据后，装置会根据注册的信息得到相应的数据。

select 的特性

1）select 操作至少要有一个 case 语句，出现读写 nil 的 channel 该分支会忽略，在 nil 的 channel 上操作则会报错。

2）select 仅支持管道，而且是单协程操作。

3）每个 case 语句仅能处理一个管道，要么读要么写。

4）多个 case 语句的执行顺序是随机的。

5）存在 default 语句，select 将不会阻塞，但是存在 default 会影响性能。

11、讲讲 Go 的 defer 底层数据结构和一些特性？

答：每个 defer 语句都对应一个_defer 实例，多个实例使用指针连接起来形成一个单连表，保存在 gotoutine 数据结构中，每次插入_defer 实例，均插入到链表的头部，函数结束再一次从头部取出，从而形成后进先出的效果。

defer 的规则总结：

延迟函数的参数是 defer 语句出现的时候就已经确定了的。

延迟函数执行按照后进先出的顺序执行，即先出现的 defer 最后执行。

延迟函数可能操作主函数的返回值。

申请资源后立即使用 defer 关闭资源是个好习惯。

12、单引号，双引号，反引号的区别？

单引号，表示byte类型或rune类型，对应 uint8和int32类型，默认是 rune 类型。byte用来强调数据是raw data，而不是数字；而rune用来表示Unicode的code point。

双引号，才是字符串，实际上是字符数组。可以用索引号访问某字节，也可以用len()函数来获取字符串所占的字节长度。

反引号，表示不支持任何转义序列的字符串字面量。

二、map相关

1、map 使用注意的点，是否并发安全？

map的类型是map[key]，key类型的ke必须是可比较的，通常情况，会选择内建的基本类型，比如整数、字符串做key的类型。如果要使用struct作为key，要保证struct对象在逻辑上是不可变的。在Go语言中，map[key]函数返回结果可以是一个值，也可以是两个值。map是无序的，如果我们想要保证遍历map时元素有序，可以使用辅助的数据结构，例如orderedmap。

第一，一定要先初始化，否则panic

第二，map类型是容易发生并发访问问题的。不注意就容易发生程序运行时并发读写导致的panic。 Go语言内建的map对象不是线程安全的，并发读写的时候运行时会有检查，遇到并发问题就会导致panic。

2、map 循环是有序的还是无序的？

无序的, map 因扩张而重新哈希时，各键值项存储位置都可能会发生改变，顺序自然也没法保证了，所以官方避免大家依赖顺序，直接打乱处理。就是 for range map 在开始处理循环逻辑的时候，就做了随机播种

3、 map 中删除一个 key，它的内存会释放么？（常问）

如果删除的元素是值类型，如int，float，bool，string以及数组和struct，map的内存不会自动释放

如果删除的元素是引用类型，如指针，chan等，map的内存会自动释放，但释放的内存是子元素应用类型的内存占用

将map设置为nil后，内存被回收。

这个问题还需要大家去搜索下答案，我记得有不一样的说法，谨慎采用本题答案。

4、怎么处理对 map 进行并发访问？有没有其他方案？ 区别是什么？

sync.Map对比原始map:
和原始map+RWLock的实现并发的方式相比，减少了加锁对性能的影响。它做了一些优化:可以无锁访问readmap，而且会优先操作readmap，倘若只操作readmap就可以满足要求，那就不用去操作writemap(dirty)，所以在某些特定场景中它发生锁竞争的频率会远远小于map+RWLock的实现方式

优点:
适合读多写少的场景

缺点:
写多的场景，会导致read map缓存失效，需要加锁，冲突变多，性能急剧下降

方式一、使用内置sync.Map，详细参考

https://mbd.baidu.com/ma/s/7Hwd9yMcmbd.baidu.com/ma/s/7Hwd9yMc

方式二、使用读写锁实现并发安全map

https://mbd.baidu.com/ma/s/qO7b0VQUmbd.baidu.com/ma/s/qO7b0VQU

5、 nil map 和空 map 有何不同？

nil map 未初始化，空map是长度为空（但已初始化已分配内存地址）

1）可以对nil map进行**取值**，但取出来的东西是空(空map也是)：

var m1 map[string]string

fmt.Println(m1["1"])

//

2）不能对nil map进行**赋值**，这样将会抛出一个异常(空map可以赋值)：

var m1 map[string]string

m1["1"] = "1"

panic: assignment to entry in nil map

//
3) 通过fmt打印map时，空map和nil map结果是一样的，都为map[]。所以，这个时候别断定map是空还是nil，而应该通过map == nil来判断。

nil map可取值不可赋值，空map可取值可赋值。取值都为空。

6、map 的数据结构是什么？是怎么实现扩容？

答：golang 中 map 是一个 kv 对集合。底层使用 hash table，用链表来解决冲突 ，出现哈希冲突时，以 bmap 为最小粒度挂载，一个 bmap 可以放 8 个 kv。每个 map 的底层结构是 hmap，是有若干个挂载了 bmap 的 bucket 组成的数组，每个 bucket 底层都采用链表结构来挂载bmap。

map中当我们插入key之后计算所得的哈希值的低B位决定了key落在哪个桶中，哈希值的高8位决定了key落在桶中的哪个位置

hmap 的结构如下：

type hmap struct {
count int // 元素个数
flags uint8
B uint8 // 扩容常量相关字段B是buckets数组的长度的对数(2^B个buckets)
noverflow uint16 // 溢出的bucket个数
hash0 uint32 // hash seed
buckets unsafe.Pointer // buckets 数组指针
oldbuckets unsafe.Pointer // 结构扩容的时候用于赋值的buckets数组
nevacuate uintptr // 搬迁进度
extra *mapextra // 用于扩容的指针
}

map 的容量大小

map 容量最多可容纳 6.5*2^B 个元素，6.5 为装载因子阈值常量。
装载因子=填入表中的元素个数/散列表的长度(也就是桶个数)，（可理解为装载因子等于平均每个桶存的元素个数）
装载因子越大，说明空闲位置越少，冲突越多，散列表的性能会下降。

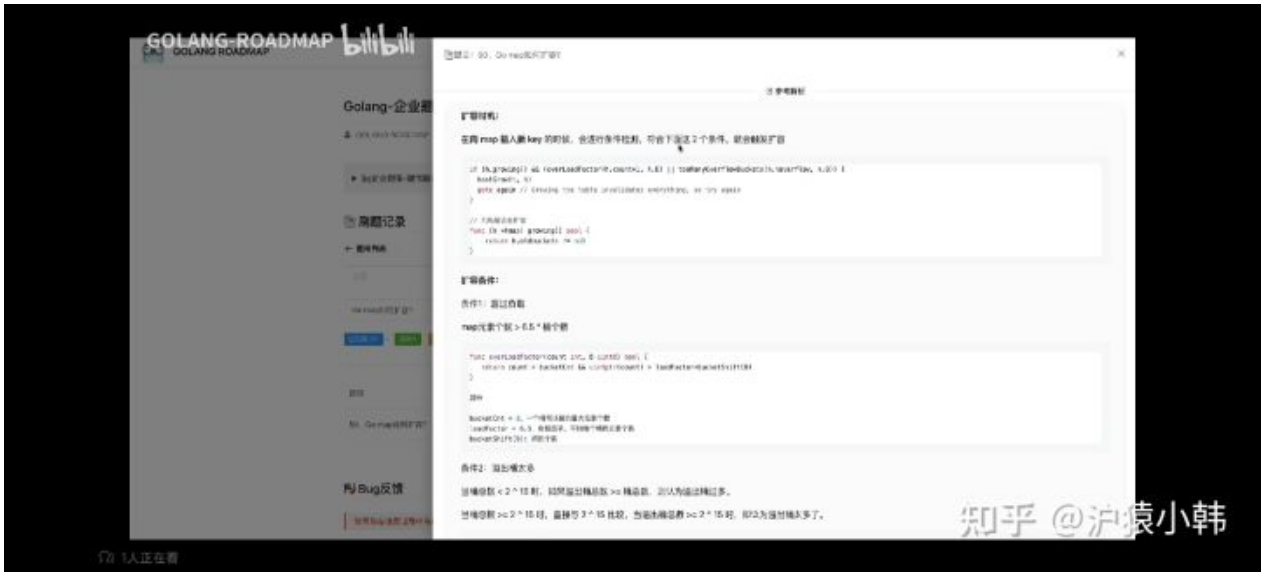
触发 map 扩容的条件

1）装载因子超过阈值。源码里定义的阈值是 6.5(即map元素个数>6.5*桶个数)。

2）溢出桶数量过多。 当桶总数<2^15 时，溢出桶总数>=桶总数或
当桶总数>=2^15 时，溢出桶总数>=2^15,都认为溢出桶过多

扩容机制
1)双倍扩容:针对条件1，新建一个buckets数组，新的buckets数组大小是原来的2倍，然后旧buckets数据搬迁到新的buckets。该方法我们称之为双倍扩容

2)等量扩容:针对条件2，并不扩大容量，buckets数量维持不变，重新做一遍类似双倍扩容的搬迁动作，把松散的键值对重新排列一次，使得同一个bucket中的key排列地更紧密，节省空间，提高 bucket 利用率，进而保证更快的存取。该方法我们称之为等量扩容。



7、slices能作为map类型的key吗？

当时被问的一脸懵逼，其实是这个问题的变种：golang 哪些类型可以作为map key？

答案是：**在golang规范中，可比较的类型都可以作为map key；** 这个问题又延伸到在：golang规范中，哪些数据类型可以比较？

不能作为map key 的类型包括：

- slices
- maps
- functions

详细参考：

golang 哪些类型可以作为map keyblog.csdn.net/lanyang123456/article/details/123765745

三、context相关

1、context 结构是什么样的？context 使用场景和用途？

（难，也常常问你项目中怎么用，光靠记答案很难让面试官满意，反正有各种结合实际的问题）

参考链接：

go context详解 - 卷毛狒狒 - 博客园www.cnblogs.com/juanmaofeifei/p/14439957.html

答：Go 的 Context 的数据结构包含 Deadline，Done，Err，Value，Deadline 方法返回一个 time.Time，表示当前 Context 应该结束的时间，ok 则表示有结束时间，Done 方法当 Context 被取消或者超时时候返回的一个 close 的 channel，告诉给 context 相关的函数要停止当前工作然后返回了，Err 表示 context 被取消的原因，Value 方法表示 context 实现共享数据存储的地方，是协程安全的。context 在业务中是经常被使用的，

其主要的应用 ：

1：上下文控制，2：多个 goroutine 之间的数据交互等，3：超时控制：到某个时间点超时，过多久超时。

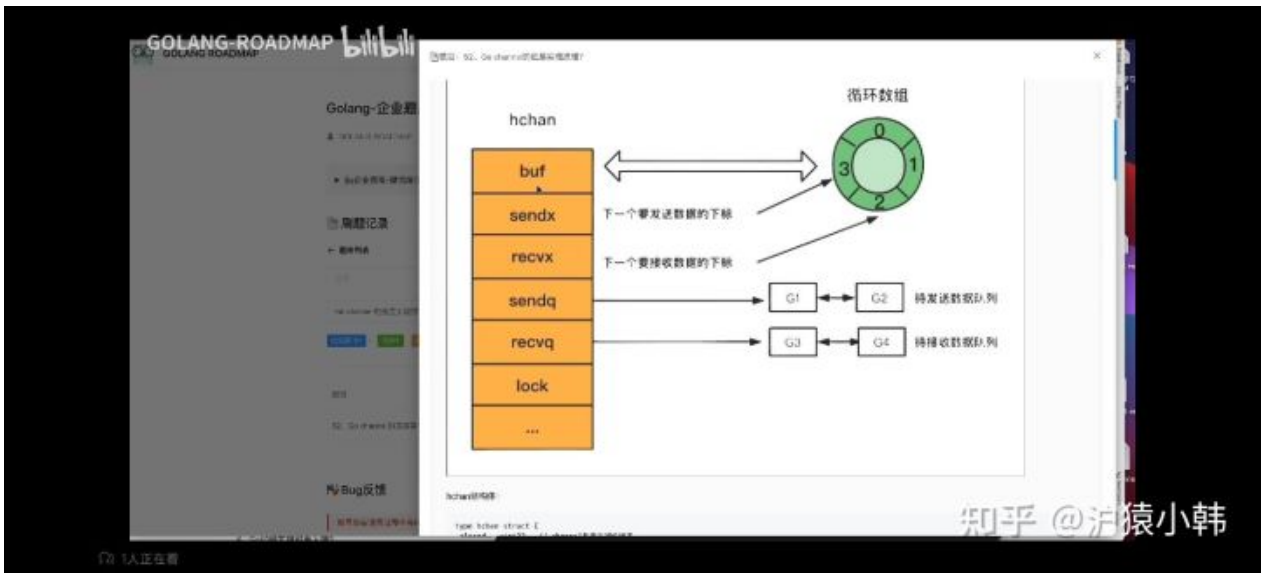
四、channel相关

1、channel 是否线程安全？锁用在什么地方？

为什么设计成线程安全?
不同协程通过channel进行通信，本身的使用场景就是多线程，为了保证数据的一致性，必须实现线程安全。

如何实现线程安全的?
channel的底层实现中，hchan结构体中采用Mutex锁来保证数据读写安全。在对循环数组buf中的数据进行入队和出队操作时，必须先获取互斥锁，才能操作channel数据。

2、go channel 的底层实现原理 （数据结构）



buf(循环数组)、
sendx(循环数组中下一个要发送数据的下标)、
recvx(循环数组中下一个要接受数据的下标)、
sendq(写等待的goroutine队列)、
recvq(读等待的goroutine队列)、
lock(锁)

3、nil、关闭的 channel、有数据的 channel，再进行读、写、关闭会怎么样？（各类变种题型，重要）

还要去了解一下单向channel,如只读或者只写通道常见的异常问题，这块还需要大家自己总结总结，有懂得大佬也可以评论发送答案。

4、向 channel 发送数据和从 channel 读数据的流程是什么样的？

发送流程：

接收流程：

这个没啥好说的，底层原理，1、2、3描述出来，保证面试官满意。具体的文字描述下面一题有，channel的概念多且复杂，脑海中有个总分的概念，否则你说的再多，面试官也抓不住你说的重点，等于白说。问题5已经为大家总结好了。

5、讲讲 Go 的 chan 底层数据结构和主要使用场景

答：channel 的数据结构包含 qccount 当前队列中剩余元素个数，dataqsiz 环形队列长度，即可以存放的元素个数，buf 环形队列指针，elemsize 每个元素的大小，closed 标识关闭状态，elemtype 元素类型，sendx 队列下表，指示元素写入时存放到队列中的位置，recv 队列下表，指示元素从队列的该位置读出。recvq 等待读消息的 goroutine 队列，sendq 等待写消息的 goroutine 队列，lock 互斥锁，chan 不允许并发读写。

无缓冲和有缓冲区别： 管道没有缓冲区，从管道读数据会阻塞，直到有协程向管道中写入数据。同样，向管道写入数据也会阻塞，直到有协程从管道读取数据。管道有缓冲区但缓冲区没有数据，从管道读取数据也会阻塞，直到协程写入数据，如果管道满了，写数据也会阻塞，直到协程从缓冲区读取数据。

channel 的一些特点

1）、读写 nil 管道(未初始化管道)会永久阻塞
2）、关闭的管道读数据仍然可以读数据
3）、往关闭的管道写数据会 panic

4）、关闭为 nil 的管道 panic
5）、关闭已经关闭的管道 panic

向 channel 写数据的流程：
如果等待接收队列 recvq 不为空 :
说明缓冲区中没有数据或者没有缓冲区，此时直接从 recvq 取出 G,并把数据写入，最后把该 G 唤醒，结束发送过程；
如果等待接收队列为空 :
1）如果缓冲区中有空余位置，将数据写入缓冲区，结束发送过程；
2）如果缓冲区中没有空余位置或没有缓冲区，将待发送数据写入 G，将当前 G 加入 sendq，进入睡眠，等待被读 goroutine 唤醒；

向 channel 读数据的流程：
如果等待发送队列 sendq 不为空：
1）没有缓冲区，直接从 sendq 中取出 G，把 G 中数据读出，最后把 G 唤醒，结束读取过程；
2）有缓冲区且缓冲区一定已满，从缓冲区中首部读出数据，把 G 中数据写入缓冲区尾部，把 G 唤醒，结束读取过程；
如果等待发送队列为空：
1）如果缓冲区中有数据：则从缓冲区取出数据，结束读取过程；
2）如果缓冲区中没数据或没有缓冲区：将当前 goroutine 加入 recvq，进入睡眠，等待被写 goroutine 唤醒；

使用场景： 消息传递、消息过滤，信号广播，事件订阅与广播，请求、响应转发，任务分发，结果汇总，并发控制，限流，同步与异步

五、GMP相关

1、什么是 GMP？（必问）

答：G 代表着 goroutine，P 代表着上下文处理器，M 代表 thread 线程，在 GPM 模型，有一个全局队列（Global Queue）：存放等待运行的 G，还有一个 P 的本地队列：也是存放等待运行的 G，但数量有限，不超过 256 个。GPM 的调度流程从 go func()开始创建一个 goroutine，新建的 goroutine 优先保存在 P 的本地队列中，如果 P 的本地队列已经满了，则会保存到全局队列中。M 会从 P 的队列中取一个可执行状态的 G 来执行，如果 P 的本地队列为空，就会从其他的 MP 组合偷取一个可执行的 G 来执行，当 M 执行某一个 G 时候发生系统调用或者阻塞，M 阻塞，如果这个时候 G 在执行，runtime 会把这个线程 M 从 P 中摘除，然后创建一个新的操作系统线程来服务于这个 P，当 M 系统调用结束时，这个 G 会尝试获取一个空闲的 P 来执行，并放入到这个 P 的本地队列，如果这个线程 M 变成休眠状态，加入到空闲线程中，然后整个 G 就会被放入到全局队列中。

关于 G,P,M 的个数问题，G 的个数理论上是无限制的，但是受内存限制，P 的数量一般建议是逻辑 CPU 数量的 2 倍，M 的数据默认启动的时候是 10000，内核很难支持这么多线程数，所以整个限制客户忽略，M 一般不做设置，设置好 P，M 一般都是要大于 P。

2、进程、线程、协程有什么区别？（必问）

进程：是应用程序的启动实例，每个进程都有独立的内存空间，不同的进程通过进程间的通信方式来通信。

线程：从属于进程，每个进程至少包含一个线程，线程是 CPU 调度的基本单位，多个线程之间可以共享进程的资源并通过共享内存等线程间的通信方式来通信。

协程：为轻量级线程，与线程相比，协程不受操作系统的调度，协程的调度器由用户应用程序提供，协程调度器按照调度策略把协程调度到线程中运行

3、抢占式调度是如何抢占的？

基于协作式抢占

基于信号量抢占

就像操作系统要负责线程的调度一样，Go的runtime要负责goroutine的调度。现代操作系统调度线程都是抢占式的，我们不能依赖用户代码主动让出CPU，或者因为IO、锁等待而让出，这样会造成调度的不公平。基于经典的时间片算法，当线程的时间片用完

之后，会被时钟中断给打断，调度器会将当前线程的执行上下文进行保存，然后恢复下一个线程的上下文，分配新的时间片令其开始执行。这种抢占对于线程本身是无感知的，系统底层支持，不需要开发人员特殊处理。

基于时间片的抢占式调度有个明显的优点，能够避免CPU资源持续被少数线程占用，从而使其他线程长时间处于饥饿状态。goroutine的调度器也用到了时间片算法，但是和操作系统的线程调度还是有些区别的，因为整个Go程序都是运行在用户态的，所以不能像操作系统那样利用时钟中断来打断运行中的goroutine。也得益于完全在用户态实现，goroutine的调度切换更加轻量。

上面这两段文字只是对调度的一个概括，具体的协作式调度、信号量调度大家还需要去详细了解，这偏底层了，大厂或者中高级开发会问。（字节就问了）

4、M 和 P 的数量问题？

p默认cpu内核数

M与P的数量没有绝对关系，一个M阻塞，P就会去创建或者切换另一个M，所以，即使P的默认数量是1，也有可能会创建很多个M出来

【Go语言调度模型G、M、P的数量多少合适？】

详细参考这篇文章

https://mbd.baidu.com/ma/s/ZMOKQATrmbd.baidu.com/ma/s/ZMOKQATr

GMP数量这一块，结论很好记，没用项目经验的话，问了项目中怎么用可能容易卡壳。

六、锁相关

1、除了 mutex 以外还有那些方式安全读写共享变量？

* 将共享变量的读写放到一个 goroutine 中，其它 goroutine 通过 channel 进行读写操作。

* 可以用个数为 1 的信号量（semaphore）实现互斥

* 通过 Mutex 锁实现

2、Go 如何实现原子操作？

答：原子操作就是不可中断的操作，外界是看不到原子操作的中间状态，要么看到原子操作已经完成，要么看到原子操作已经结束。在某个值的原子操作执行的过程中，CPU 绝对不会再去执行其他针对该值的操作，那么其他操作也是原子操作。

Go 语言的标准库代码包 sync/atomic 提供了常见的原子操作:
·增减(Add 为前缀的函数)
·读取（Load 为前缀的函数）
·写入（Store 为前缀的函数）
·比较并交换(CompareAndSwap 为前缀的函数)
·交换(Swap 为前缀的函数)
(atomic操作的对象是地址，需要把变量的地址作为参数传递给方法)

原子操作与互斥锁的区别
1)、原子操作由底层硬件支持，而锁是基于原子操作+信号量完成的。若实现相同的功能，前者通常会更有效率
2)、原子操作是单个指令的互斥操作;互斥锁/读写锁是一种数据结构，可以完成临界区(多个指令)的互斥操作，扩大原子操作的范围
3)、原子操作是无锁操作，属于乐观锁;说起锁的时候，一般属于悲观锁

3、Mutex 是悲观锁还是乐观锁？悲观锁、乐观锁是什么？

悲观锁

悲观锁：当要对数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制，在修改数据之前先锁定，再修改的方式被称之为悲观并发控制【Pessimistic Concurrency Control，缩写“PCC”，又名“悲观锁”】。

乐观锁

乐观锁是相对悲观锁而言的，乐观锁假设数据一般情况不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果冲突，则返回给用户异常信息，让用户决定如何去做。乐观锁适用于读多写少的场景，这样可以提高程序的吞吐量

4、Mutex 有几种模式？

原文链接：https://blog.csdn.net/weixin_52690231/article/details/125267207

1）正常模式

1. 一个尝试加锁失败的goroutine会先自旋几次（此时仍占用CPU资源，不会阻塞挂起），尝试通过原子操作获得锁，若几次自旋之后仍不能获得锁，则会按照先入先出FIFO的顺序进入等待队列排队等候。
2. 但是当锁被释放，第一个等待者被唤醒后并不会直接拥有锁，而是需要和处于自旋阶段的尚未排队等待的goroutine竞争。这种情况下后来者更有优势，一方面，它们正在CPU上运行，自然比刚被唤醒的goroutine更有优势，另一方面处于自旋状态的goroutine可以有很多，而被唤醒的goroutine每次只有一个，所以被唤醒的goroutine有很大概率拿不到锁。这种情况下被唤醒的goroutine会被重新插入到队列的头部，而不是尾部。

当一个goroutine加锁等待时间超过了1ms后，它会把当前Mutex从正常模式切换至“饥饿模式”。

2）饥饿模式

在饥饿模式下，Mutex 的拥有者将直接把锁交给队列最前面的 waiter。新来的 goroutine 不会尝试获取锁，即使锁处于Unlock状态，它也不会去抢，也不会自旋，它会乖乖地加入到等待队列的尾部。
如果拥有 Mutex 的 waiter 发现下面两种情况的其中之一，它就会把这个 Mutex 转换成正常模式:
1. 此 waiter 已经是队列中的最后一个 waiter 了，没有其它的等待锁的 goroutine 了；
2. 此 waiter 的等待时间小于 1 毫秒。

5、goroutine 的自旋占用资源如何解决

自旋锁是指当一个线程在获取锁的时候，如果锁已经被其他线程获取，那么该线程将循环等待，然后不断地判断是否能够被成功获取，直到获取到锁才会退出循环。

自旋的条件如下：

1）还没自旋超过 4 次，

2）多核处理器，

3）GOMAXPROCS > 1，

4）p 上本地 goroutine 队列为空。

mutex 会让当前的 goroutine 去空转 CPU，在空转完后再次调用 CAS 方法去尝试性的占有锁资源，直到不满足自旋条件，则最终会加入到等待队列里。

七、并发相关

1、怎么控制并发数？

第一，有缓冲通道

根据通道中没有数据时读取操作陷入阻塞和通道已满时继续写入操作陷入阻塞的特性，正好实现控制并发数量。

```
func main() {
count := 10 // 最大支持并发
sum := 100 // 任务总数
wg := sync.WaitGroup{} //控制主协程等待所有子协程执行完之后再退出。
c := make(chan struct{}, count) // 控制任务并发的chan
defer close(c)
for i:=0; i<sum;i++{
wg.Add(1)
c <- struct{}{} // 作用类似于waitgroup.Add(1)
go func(j int) {
defer wg.Done()
fmt.Println(j)
<- c // 执行完毕，释放资源
}(i)
}
wg.Wait()
}
```

第二，三方库实现的协程池

panjf2000/ants（比较火）

Jeffail/tunny

```
import (
"log"
"time"
"github.com/Jeffail/tunny"
)
func main() {
pool := tunny.NewFunc(10,
func(i interface{}) interface{} {
log.Println(i)
time.Sleep(time.Second)
return nil
})
defer pool.Close()
for i := 0; i < 500; i++ {
go pool.Process(i)
}
time.Sleep(time.Second * 4)
}
```

2、多个 goroutine 对同一个 map 写会 panic，异常是否可以用 defer 捕获？

可以捕获异常，但是只能捕获一次，Go语言，可以使用多值返回来返回错误。不要用异常代替错误，更不要用来控制流程。在极个别的情况下，才使用Go中引入的Exception处理：defer, panic, recover Go中，对异常处理的原则是：多用error包，少用panic

```
defer func() { if err := recover(); err != nil { // 打印异常，关闭资源，退出此函数 fmt.Println(err) } }()
```

3、如何优雅的实现一个 goroutine 池

（百度、手写代码，本人面传音控股被问道：请求数大于消费能力怎么设计协程池）

这一块能啃下来，offer满天飞，这应该是保证高并发系统稳定性、高可用的核心部分之一。

建议参考：

Golang学习篇--协程池_Word哥的博客-CSDN博客_golang协程池blog.csdn.net/finghting321/article/details/106492915/

八、GC相关

1、常见的GC算法

1. 引用计数：为每个对象维护一个引用计数，当引用该对象的对象销毁时，引用计数-1，当对象引用计数为 0 时回收该对象。
   • 代表语言：Python、PHP、Swift
   • 优点：对象回收快，不会出现内存耗尽或达到某个阈值时才回收。
   • 缺点：不能很好的处理循环引用，而实时维护引用计数也是有损耗的。
2. 分代收集：按照对象生命周期长短划分不同的代空间，生命周期长的放入老年代，短的放入新生代，不同代有不同的回收算法和回收频率.
   • 代表语言：Java
   • 优点：回收性能好
   • 缺点：算法复杂
3. 标记-清除：从根变量开始遍历所有引用的对象，标记引用的对象，没有被标记的进行回收。
   • 代表语言：Golang（三色标记法)
   • 优点：解决了引用计数的缺点。
   • 缺点：需要 STW，暂时停掉程序运行。

2、go gc 是怎么实现的？（必问）

(1）GC机制随着golang版本变化如何变化的？
Go 的 GC 回收有三次演进过程，
Go V1.3 之前普通标记清除（mark and sweep）方法:先启动 STW 暂停，然后执行标记，再执行数据回收，最后停止 STW，效率极低。
Go V1.3 版本标记清除做了点优化，流程是：先启动 STW 暂停，然后执行标记，停止 STW，最后再执行数据回收。
Go V1.5 三色标记法,堆空间启动写屏障,栈空间不启动。全部扫描之后,需要重新扫描一次栈(需要 STW),标记栈上引用的白色对象的存活,效率普通。
Go V1.8 三色标记法，混合写屏障机制：栈空间不启动（栈上的可达对象和并发时新创建和删除的对象全部标记成黑色），堆空间启用写屏障，整个过程对于栈的处理不用 STW，效率高。

(2)三色标记法的流程？

```
1. 创建：白、灰、黑三个集合
2. 将所有对象放入白色集合中
3. 遍历所有root对象，把遍历到的对象从白色集合放入灰色集合(这里放入灰色集合的都是根节点的对象)
4. 遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，自身标记为黑色
5. 重复步骤4，直到灰色中无任何对象
6. 收集所有白色对象（垃圾）
```

详细流程:
1.标记准备(Mark Setup): 打开写屏障(Write Barrier)需STW(stop the world)
2.标记开始(Marking): 使用三色标记法并发标记，与用户程序并发执行
3.标记终止(Mark Termination): 对触发写屏障的对象进行重新扫描标记，关闭写屏障(Write Barrier)，需 STW (stop the world)
4.清理(Sweeping): 将需要回收的内存归还到堆中，将过多的内存归还给操作系统，与用户程序并发执行

即：
Mark Setup(要stw):

```
1. stop the world 暂停程序执行
2. 启动标记工作携程(markworkergoroutine)，用于第二阶段
3. 启动写屏障
4. 将root对象放入标记队列(放入标记队列里的就是灰色)
5. start the world 取消程序暂停
```

Marking(不用stw):

```
1.从标记队列里取出对象，标记为黑色
2.然后检测是否指向了另一个对象，如果有，将另一个对象放入队列(灰色)
3.重复上述过程，直到灰色对象标记队列为空
4.在扫描过程中，用户程序如果新创建了对象或者修改了对象,就会触发写屏障，将对象放入单独的marking队列，也就是标记为灰色
```

Mark Termination(要stw):

```
1.stop the world暂停程序执行
2.rescan:将marking阶段修改的对象触发写屏障产生的队列里的对象取出，标记为黑色:然后检测是否指向了其他对象，如果有，将另一个对象放入标记
3.关闭写屏障
4. start the world取消程序暂停
```

Sweep(不用stw):

```
1.清除白色的对象
```

(3)插入屏障、删除屏障，混合写屏障

插入写屏障:
对象被引用时触发的机制（只在堆内存中生效）：赋值器这一行为通知给并发执行的回收器，被引用的对象标记为灰色
缺点：结束时需要STW来重新扫描栈，标记栈上引用的白色对象的存活

删除写屏障:
对象被删除时触发的机制（只在堆内存中生效）：当一个白色对象被另外一个对象解除引用时，将该被引用对象标记为灰色（白色对象被保护）
缺点：产生内存冗余，如果上述该白色对象没有被别的对象引用，相当于还是垃圾，但是这一轮垃圾回收并没有处理掉他。

混合写屏障：
1）GC 开始将栈上的对象全部扫描并标记为黑色(之后不再进行第二次重复扫描，无需 STW)，
2）GC 期间，任何在栈上创建的新对象，均为黑色。
3）堆上被删除的对象标记为灰色。
4）堆上被添加的对象标记为灰色。

（屏障的作用：避免程序运行过程中，变量被误回收；减少STW的时间）

3、go 是 gc 算法是怎么实现的？（得物，出现频率低）

```
func GC() {
n := atomic.Load(&work.cycles)
gcWaitOnMark(n)
gcStart(gcTrigger{kind: gcTriggerCycle, n: n + 1})
gcWaitOnMark(n + 1)
for atomic.Load(&work.cycles) == n+1 && sweepone() != ^uintptr(0) {
sweep.nbgsweep++
Gosched()
```

```
}
for atomic.Load(&work.cycles) == n+1 && atomic.Load(&mheap_.sweepers) != 0 {
Gosched()
}
mp := acquirem()
cycle := atomic.Load(&work.cycles)
if cycle == n+1 || (gcphase == _GCmark && cycle == n+2) {
mProf_PostSweep()
}
releasem(mp)
}
```

底层原理了，可能大厂，中高级才会问，参考：

Golang GC算法解读_suchy_sz的博客-CSDN博客_go的gc算法blog.csdn.net/shudaqi2010/article/details/90025192

4、GC 中 stw 时机，各个阶段是如何解决的？（百度）

底层原理，自行百度一下，我等渣渣简历都过不了BAT，字节，虾皮，特使拉以及一些国Q还能收到面试邀约。

1）在开始新的一轮 GC 周期前，需要调用 gcWaitOnMark 方法上一轮 GC 的标记结束（含扫描终止、标记、或标记终止等）。

2）开始新的一轮 GC 周期，调用 gcStart 方法触发 GC 行为，开始扫描标记阶段。

3）需要调用 gcWaitOnMark 方法等待，直到当前 GC 周期的扫描、标记、标记终止完成。

4）需要调用 sweepone 方法，扫描未扫除的堆跨度，并持续扫除，保证清理完成。在等待扫除完毕前的阻塞时间，会调用 Gosched 让出。

5）在本轮 GC 已经基本完成后，会调用 mProf_PostSweep 方法。以此记录最后一次标记终止时的堆配置文件快照。

6）结束，释放 M。

5、GC 的触发时机？

初级必问，分为系统触发和主动触发。

1）gcTriggerHeap：当所分配的堆大小达到阈值时，将会触发（默认值为100%，当前堆内存占用是上次GC结束后占用内存的2倍时，触发GC）。

2）gcTriggerTime：当距离上一个 GC 周期的时间超过一定时间时，将会触发(时间周期以runtime.forcegcperiod 变量为准，默认 2 分钟)。

3）gcTriggerCycle：如果没有开启 GC，则启动 GC。

4）手动触发的 runtime.GC 方法。

九、内存相关

1、谈谈内存泄露，什么情况下内存会泄露？怎么定位排查内存泄漏问题？

答：go 中的内存泄漏一般都是 goroutine 泄漏，就是 goroutine 没有被关闭，或者没有添加超时控制，让 goroutine 一只处于阻塞状态，不能被 GC。

内存泄露有下面一些情况

永久性内存泄漏：
1）如果 goroutine 在执行时被阻塞而无法退出，就会导致 goroutine 的内存泄漏，一个 goroutine 的最低栈大小为 2KB，在高并发的场景下，对内存的消耗也是非常恐怖的。

2）互斥锁未释放或者造成死锁会造成内存泄漏

3）time.Ticker未关闭导致泄漏。作为循环触发器，必须调用 stop 方法才会停止，从而被 GC 掉，否则会一直占用内存空间。

暂时性内存泄漏：
4）字符串的截取引发临时性的内存泄漏

func main() { var str0 = "123456789012345678901" str1 := str0[:10] }

5）切片截取引起子切片内存泄漏

func main() { var s0 = []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} s1 := s0[:3] }

6）函数数组传参引发内存泄漏
【如果我们在函数传参的时候用到了数组传参，且这个数组够大（我们假设数组大小为 100 万，64 位机上消耗的内存约为 800w 字节，即 8MB 内存），或者该函数短时间内被调用 N 次，那么可想而知，会消耗大量内存，对性能产生极大的影响，如果短时间内分配大量内存，而又来不及 GC，那么就会产生临时性的内存泄漏，对于高并发场景相当可怕。】

排查方式：

一般通过 pprof 是 Go 的性能分析工具，在程序运行过程中，可以记录程序的运行信息，可以是 CPU 使用情况、内存使用情况、goroutine 运行情况等，当需要性能调优或者定位 Bug 时候，这些记录的信息是相当重要。

当然你能说说具体的分析指标更加分咯，有的面试官就喜欢他问什么，你简洁的回答什么，不喜欢巴拉巴拉详细解释一通，比如虾P面试官，不过他考察的内容特别多，可能是为了节约时间。

2、知道 golang 的内存逃逸吗？什么情况下会发生内存逃逸？（必问）

答：
1)本该分配到栈上的变量，跑到了堆上，这就导致了内存逃逸。
2)栈上的变量，函数结束后变量会跟着回收掉，不会有额外性能的开销。
3)变量从栈逃逸到堆上，如果要回收掉，需要进行 gc，会带来额外的性能开销。

内存逃逸的情况如下：

1)指针逃逸 : 如果一个函数结束之后外部还有引用，那么必定被分配在堆中(即方法返回局部变量指针)

2)栈空间不足逃逸 : 在分配时不能确定栈空间大小，不知道到底要开辟的空间是否超出限制也会发生逃逸,直接分配到堆中

3)动态类型逃逸 : interface类型可以代表任意类型，编译器不知道参数会是什么类型，只有运行时才知道，因此只能分配到堆上。

4)切片由于append操作，导致容量变大，就会被分配在堆上面

5)发送指针或带有指针的值到 channel 中，因为不知道什么时候被释放，就会被分配到堆上（下同）

6)在 slice 或 map 中存储指针

3、请简述 Go 是如何分配内存的？

mspan 内存管理的基础单元; (是一个双向链表，每个span包含npages个页)（这里的页指go自己封装的内存页）
mcache 线程缓存;
mcentral 中心缓存; （全局的）
mheap 页堆

分配流程:

1. 首先通过计算使用的大小规格
2. 然后使用mcache中对应大小规格的块(mspan)分配，或直接用页堆分配。
3. 如果mcache 中没有可用的块，则向mcentral申请,如果mcentral 中没有可用的块，则向mheap申请，并根据算法找到最合适的mspan 。
4. 如果申请到的mspan超出申请大小，将会根据需求进行切分，以返回用户所需的页数。剩余的页构成一个新的mspan放回mheap 的空闲列表。
5. 如果mheap中没有可用span，则向操作系统申请一系列新的页(最小1MB)

4、Channel 分配在栈上还是堆上？哪些对象分配在堆上，哪些对象分配在栈上？

Channel 被设计用来实现协程间通信的组件，其作用域和生命周期不可能仅限于某个函数内部，所以 golang 直接将其分配在堆上

准确地说，你并不需要知道。Golang 中的变量只要被引用就一直会存活，存储在堆上还是栈上由内部实现决定而和具体的语法没有关系。

知道变量的存储位置确实和效率编程有关系。如果可能，Golang 编译器会将函数的局部变量分配到函数栈帧（stack frame）上。然而，如果编译器不能确保变量在函数 return 之后不再被引用，编译器就会将变量分配到堆上。而且，如果一个局部变量非常大，那么它也应该被分配到堆上而不是栈上。

当前情况下，如果一个变量被取地址，那么它就有可能被分配到堆上,然而，还要对这些变量做逃逸分析，如果函数 return 之后，变量不再被引用，则将其分配到栈上。

5、介绍一下大对象小对象，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 CPU。优化思路是，减少对象分配。

微对象：小于16B的对象 （依次尝试微型分配器、线程缓存、中心缓存、堆 分配内存）
小对象：16B≤对象大小≤32KB 的对象 （依次尝试线程缓存、中心缓存、堆 分配内存）
大对象：32KB≤对象大小 的对象（直接分配到堆上）

6、内存对齐机制

什么是内存对齐:

为了能让CPU可以更快的存取到各个字段，Go编译器通过在结构体的各个字段之间填充一些空白已达到对齐的目的。所谓的数据对齐，是指内存地址是所存储数据大小(按字节为单位）的整数倍，以便CPU可以一次将该数据从内存中读取出来。

优点: 1.提高可移植性。2..提高内存的访问效率。

缺点: 浪费内存空间,实际上是空间换时间。

十、其他问题

1、Go 多返回值怎么实现的？

答：Go 传参和返回值是通过 FP+offset 实现，并且存储在调用函数的栈帧中。FP 栈底寄存器，指向一个函数栈的顶部;PC 程序计数器，指向下一条执行指令;SB 指向静态数据的基指针，全局符号;SP 栈顶寄存器。

2、讲讲 Go 中主协程如何等待其余协程退出？

答：（1）Go 的 sync.WaitGroup 是等待一组协程结束，sync.WaitGroup 只有 3 个方法，Add()是添加协程计数，Done()减去一个计数，Wait()阻塞直到所有的任务完成。
（2）Go 里面还能通过有缓冲的 channel 实现其阻塞等待一组协程结束，这个不能保证一组 goroutine 按照顺序执行，可以并发执

行协程。
（3）Go 里面能通过无缓冲的 channel 实现其阻塞等待一组协程结束，这个能保证一组 goroutine 按照顺序执行，但是不能并发执行。

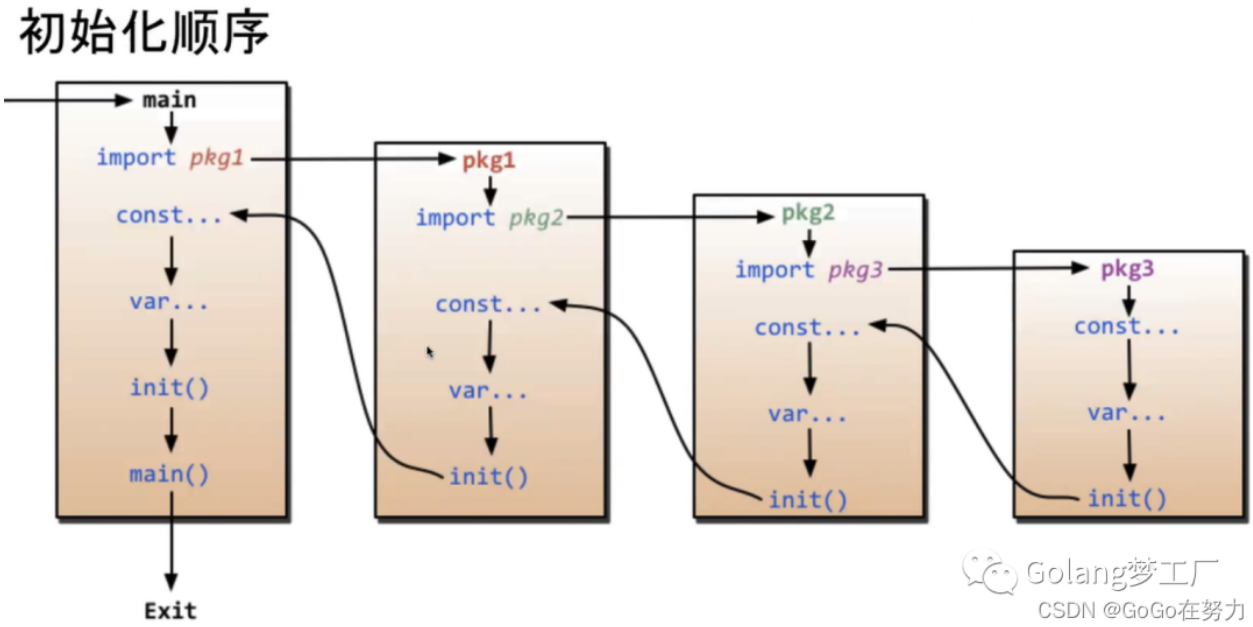
3、Go 语言中不同的类型如何比较是否相等？

答：像 string，int，float interface 等可以通过等于号或 reflect.DeepEqual 进行比较，
像 slice，struct，map 则一般使用 reflect.DeepEqual 来检测是否深度相等。

4、Go 中 init 函数的特征?

答：
（1）每个包按照import包、按照const常量、var包变量、init函数、main函数的序自动执行；
（2）init函数没有输入参数、返回值；init函数不能被其他函数调用；
（3）每个包可以有多个init函数；包的每个源文件也可以有多个init函数，这点比较特殊；
（4）同一个包的init执行顺序按照它们的文件名顺序逐个初始化；不同包的init函数按照包导入的依赖关系决定执行顺序。

init函数的主要用途：初始化不能使用初始化表达式初始化的变量



5、Go 中 uintptr 和 unsafe.Pointer 的区别？

答：unsafe.Pointer 是通用指针类型，它不能参与计算，任何类型的指针都可以转化成 unsafe.Pointer，unsafe.Pointer 可以转化成任何类型的指针，uintptr 可以转换为 unsafe.Pointer，unsafe.Pointer 可以转换为 uintptr。uintptr 是指针运算的工具，但是它不能持有指针对象（意思就是它跟指针对象不能互相转换），unsafe.Pointer 是指针对象进行运算（也就是 uintptr）的桥梁。

6、golang共享内存（互斥锁）方法实现发送多个get请求

待补充

7、从数组中取一个相同大小的slice有成本吗？

或者这么问：从切片中取一个相同大小的数组有成本吗？

这是爱立信的二面题目，这个问题我至今还没搞懂，不知道从什么切入点去分析，欢迎指教。

PS：爱立信面试都要英文自我介绍，以及问答，如果英文回答不上来，会直接切换成中文。

8、PHP能实现并发处理事务吗？

多进程：pcntl扩展

php pcntl用法-PHP问题-PHP中文网www.php.cn/php-ask-473095.html

多线程：

1）swoole（常用，生态完善）

2）pthread扩展（不常用）

为什么php多线程没人用?23 赞同 · 18 评论回答

参考并致谢

1、可可酱 可可酱：Golang常见面试题

2、Bel_Ami同学 golang 面试题(从基础到高级)

3、知乎:沪猿小韩