Golang面试题

1. **Go语言的优势**

拥有着几乎跟C语言一样的运行效率，却比C语言有着更高的开发效率。且有着强大的标准库，自主的内存管理方式和高效的垃圾回收机制，当然最重要的一点是天生支持并发，只需一个go语句就可以开启一个协程，也就是goroutine，goroutine比线程更加的轻量，所占内存更小，goroutine之间切换所消耗的性能也比线程切换要小得多，所以管理起来也更加方便，对Go程序的性能有很大的提高。

1. **内存分配**

Go语言是自主管理内存分配的，优势就是不用每次分配内存都进行系统调用，可以提升性能，缺点就是实现起来比较复杂，全靠go语言内置运行时也就是runtime进行管理。

Go 分配内存的过程，主要由四大组件管理，级别从上到下分别是：mheap、mcentral、mcache和mspan。Go程序在启动时，首先会向操作系统申请一大块内存，并交由mheap结构全局管理。mheap 会将这一大块内存，切分成不同规格的小内存块，我们称之为 mspan。然后将某一种规格交给对应的某种mcentral来管理。而mcentral是全局可见的，我们不可能在每个goroutine来申请内存的时候都加锁约束。解决问题的办法是在本地的mcache上分配。就是在一个Go 程序里，工作线程M和处理器P关联之后才能运行goroutine，每个P都会绑定一个叫 mcache 的本地缓存。然后goroutine可以在这个缓存里申请内存，不需要加锁。一般的内存分配情况就是这样的，如果是申请很小的内存比如小于16个字节会使用mcache上的微型分配器分配。如果超过 32KB 的内存申请，会直接从mheap（堆）上分配对应数量的内存页给程序，每页的大小是8kb。

1. **什么是堆内存和栈内存（内存逃逸）**

堆内存由内存分配器分配由垃圾收集器负责回收。一个程序在运行过程中，只会有一个堆内存（mheap），所以多个goroutine在堆中申请内存需要加锁避免冲突。并且堆内存需要进行垃圾回收，如果有大量的GC操作，将会使程序性能下降的很厉害。所以为了提高程序的性能，应当尽量减少内存在堆上分配。

而栈内存在函数结束后会自动回收，不需要GC操作，性能相对堆内存要好得多。所以我们在编写程序时要尽量避免一些栈内变量逃逸到堆上，而且每一次变量逃逸都需要一次额外的内存分配过程，所以避免栈内变量的逃逸对性能的提高是很有效的。比如避免返回指针类型、引用类型的变量、避免在闭包函数中使用外部变量、避免传入不能确定类型的参数如空接口、避免变量的大小大于32k等等。当不可避免地需要使用到这些变量时，我们可以提前在堆上分配，避免了逃逸后进行二次分配，对性能的提高也是有帮助的。

编码的时候应该注意：全局变量、指针、引用类型都是在堆上分配的，所以对于小变量，传值效率比传指针效率更高（值在栈中分配）。当我们要在函数中使用到大变量时，先将其在堆上分配然后只需在函数内使用到的时候声明一个变量指向它就可以了，避免了内存逃逸，当已经没有任何变量指向它了，那么它就会被GC回收 。

1. **GMP模型（是抢占式的两级线程模型（m:n））**

**G**：goroutine是用户态的轻量级的线程，一个goroutine大概需要4k的内存（运行中再动态扩容），一个Go程序创建几百万个goroutine应该是没有问题的。

**M**：是工作线程，也就是操作系统内核线程，goruntime最多允许创建10000个，超出了就会抛出异常（可以调大）

**P**：包含Go代码等的必要资源和可以调度goroutine的处理器，默认等于CPU的核心数，可以通过GOMAXPROCS调小

**Sched**：调度器的结构，维护M和G的全局队列，以及调度器的一些状态信息

主要的的调度过程就是某个工作线程M先和某个处理器P进行绑定关联形成一个有效的运行环境，然后再选择某个goroutine来运行。

详细的调度过程就是每个处理器P都会维护一个本地的队列用来存放goroutine，全局中还有一个全局队列，当我们使用go语句时就会产生一个goroutine优先放到当前P的本地队列，当本地队列满了之后才会放到全局队列。运行的时候P也会是优先在本地队列的队头取出一个goroutine来运行，当本地队列运行完了之后再取全局队列里的goroutine来运行，当全局队列也运行完了操作系统还会在其它P的队列里取出一半（负载均衡的思想）来给这个P运行，也就是work stealing机制。假设工作线程M1正在绑定处理器P1运行goroutine G1的工作，当这个G1发生io调用进行阻塞时，M1会释放绑定的P1（也就是hand off机制）。这个P1会将G1从本地队列中移除，在线程池中取出一个M2来运行G2，如果线程池里面没有工作线程，他就会自己创建一个。

（如何查看运行时的调度信息）

1. **内存对齐**
2. **垃圾回收**

Go的垃圾回收采用三色标记清除法结合混合写屏障技术。有手动触发和自动触发两种方式；手动触发需要调用runtime包的GC函数，自动触发可以配置多久触发一次或者内存分配达到多少触发一次。一次完整的垃圾回收会分为四个阶段：

1. 标记准备：需STW然后打开写屏障
2. 标记开始：使用三色标记法标记，和用户程序并发执行
3. 标记终止：STW然后重新扫描触发写屏障的对象，然后在关闭写屏障
4. 清理：把回收的内存归还到堆中，把过多的内存归还给操作系统，清理过程与用户程序并发执行

**然后三色标记法可以分为6个步骤：**

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
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，它自身会被放入黑色集合
5. 重复步骤4，直到灰色集合中无任何对象（用到2个机制：混合写屏障和辅助GC）
6. 收集所有白色对象也就是内存垃圾

**为什么要STW**

STW 的目的是防止 GC 扫描时内存变化引起的混乱，1.18版本采用混合写屏障之后的GC操作几乎不需要STW，只是在标记准备和结束的时候停止大约几十微秒的时间主要是告诉程序开始和结束GC，和一些GC之前的初始化操作。

**混合写屏障**：是让垃圾清理和用户程序可以并发执行的技术，大大减少了 STW 的时间。开启后指针传递时会把指针标记，即本轮不回收，下次 GC 时再确定。

**辅助 GC**：为了防止内存分配过快，协助 GC 做一部分工作。

1. **GC调优**

## 控制内存分配的速度，避免变量重复扩容和内存逃逸（*如少量使用+连接string，slice提前分配足够的内存来降低扩容带来的拷贝）*

* 1. 限制 Goroutine 的数量
  2. 避免map的key过多，导致扫描时间增加
  3. 变量复用，减少对象分配，例如使用 sync.Pool 来复用需要频繁创建的临时对象、和使用全局变量等
  4. *增大 GOGC 的值，降低 GC 的运行频率*
  5. *提高赋值器 mutator 的 CPU 利用率（降低GC的CPU利用率）*

1. *查看GC信息*
   1. *GODEBUG='gctrace=1'; go run main.go*
   2. *go run main.go; go tool trace trace.out*
   3. *debug.ReadGCStats*
   4. *runtime.ReadMemStats：*
2. **defer（v.推迟）**

return有两步：为返回值赋值和返回调用处，defer在其中间执行

defer一般用于成对的操作（如开关文件）、函数收尾和异常捕获

defer函数参数会先在入栈之前求值再入栈，后入的defer先执行

panic之后的defer不会被执行

func F() {

    defer func() {

        if err := recover(); err != nil {

            fmt.Println("捕获异常:", err)

        }

        fmt.Println("b")

    }()

**panic("a")**

**defer fmt.Println("不会被执行")**

}

1. **make和new的区别**
2. 首先是应用场景的不一样

make() 函数专门用来为 slice、map、channel 这三种类型来分配内存并进行初始化的；而 new一般用来给这三类之外的其它类型分配内存，实际上new可以给任意类型分配内存，但是用来给这三类分配内存是不合适的。因为new只会返回一个指针指向分配好的内存，而这三类还有其他不同的字段来管理底层数据结构。

以 slice 类型为例，当创建一个 slice 类型的值时，会创建一个轻量级的[数据结构](https://so.csdn.net/so/search?q=%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/yuntaoren/article/details/_blank)，这个数据结构包含三个字段：一个是指向底层数组的指针、另外两个是这个数组的长度和容量。而用new来创建切片时，切片的长度和容量默认被初始化为0值而不能指定其它值，这样在后面赋值的时候会重新分配内存，降低性能。

1. 传入的参数不同

new只能传入一个类型参数；make可以传入一个、两个、或者三个参数

1. 返回值的类型不一样

new() 函数返回一个指向该参数类型的指针。

make() 函数返回值的类型和它接收的第一个参数类型一样。

1. **数组和切片的区别**

**数组**

数组是值类型长度是固定的，必须在编译时就确定数组长度的大小，而且数组的长度是数组类型的一部分，在运行的过程中也不能动态扩容。且长度无记录，求数组长度时需要遍历，O(n)。而且数组是值类型，大数组传给函数做参数的时候还会生成副本，导致性能降低。所以除非已经知道了固定长度，否则我们很少使用数组

**切片**

切片是引用类型长度是可变的，可以在程序的运行过程中动态扩容。切片是一种轻量的数据结构，包含三个字段：指针、长度和容量。指针指到底层数组可以从这个切片访问的第一个元素，长度是当前切片的元素个数，容量是指从当前切片的第一个元素开始到底层数组最后一个元素的元素个数

1. **切片使用的注意事项**
2. 切片是一个轻量数据结构包括指向底层数组的指针和切片的长度和容量
3. 切片在使用之前需要用make进行初始化，在make之前是nil切片，给它赋值会报错。*所有的nil切片地址都是一样的，输出为0x0。make初始化之后切片内的所有元素都是该类型的零值。当make指定切片的长度为0时，该切片就是一个空切片，空切片指向一个内存地址，但是没有分配内存空间。不管是使用 nil 切片还是空切片，对其调用内置函数 append，len 和 cap 的效果都是一样的。*
4. 扩容：扩容后，新切片指向的数组是一个全新的数组。如果新申请容量大于等于原来的两倍，扩容后的容量等于申请的容量（双数）。如果切片的容量小于 1024 个元素，扩容的时候会翻倍增加容量，如果元素个数超过1024个，扩容后容量为原来的1.25倍。
5. 创建切片的时候建议使用字面量创建，而不是在原数组上创建。在数组上创建切片会共享当前数组，也就是说切片在修改的时候，数组也会被修改，导致一些意料之外的情况。
6. 切片不是线程安全的（在底层结构没有加锁的字段），但在并发执行中不会报错，只是值会被随意更改，不能控制。
7. 深拷贝（修改不影响原切片值）：copy和遍历赋值

浅拷贝（修改会影响原切片值）：slice2=slice1

1. **map底层实现原理**

首先是使用，map是引用类型，它的零值是nil，在添加元素之前必须要使用make进行初始化。它的键的类型必须是可比较的。

Go Map在源码中是一个叫hmap的结构体。其中的buckets字段是一个切片，里面存放着哈希桶。也是结构体，有四个字段，tophash，keys，elements，overflow。Go的Map是将key和value分开存放的，可以使内存更紧凑。tophash存放哈希值的高字节（key的hash值的高8位），keys存放了键，elems存放数据。每个桶只能存放最多8个键值对，超出时就会使用溢出桶，overflow指向下一个溢出桶的位置。当溢出桶的数量大于原桶或平均每个桶里的键值对>=6.5（测试后选择6.5）时就会导致扩容。扩容分两种情况：如果溢出桶过多但是数据不多只会等量扩容，也就是桶的数量不变，只是进行整理。如果是数据太多了就会翻倍扩容，扩容后普通桶的数量为原来的两倍。

1. **map使用的注意事项以及如何保证线程安全**
   1. Go map遍历是无序的，因为每次遍历是从随机的bucket开始的，而且key的位置也在不时改变（不可寻址）。如果想按某顺序，应先对key进行排序，再用key遍历map。
   2. Go map默认是不支持并发安全的。Go官方认为 Go map 更多在典型场景使用，而不应为了小部分情况去加锁，导致更大性能的消耗。同时Go官方提供了并发安全的sync.Map让我们在写并发的时候使用。或者我们也可以在操作map之前先加锁。在“读”多时候适合使用sync.Map，在“写”多的时候适合使用锁和map
   3. map冲突：Go采用链地址法解决冲突，具体就是插入key到map中时，当key定位的桶填满8个元素后，将会创建一个溢出桶，并且将溢出桶插入当前桶所在链表尾部。
2. **channel**

通道本质上是一个环形队列，在源码中是一个结构体，其中的buf字段是一个指针，指向在堆内存上的环形队列，里面存放着通道中的数据。lock字段用来进行每次访问的加锁和解锁，保证并发访问的安全性。sendx和recvx表示下次发送数据的位置和接收的位置。sendq和recvq表示在两端阻塞等待的goroutine队列。

1. **channel注意事项**
   1. 通道属于引用类型，是多个gorotine之间传递数据和同步的重要工具，在使用之前需要用make进行初始化，make之后通道的缓冲容量就被固定了，如果还没有初始化就向通道发送或者接收元素的话会导致当前goroutine永久阻塞。
   2. 在make的时候可以用第二个参数指定通道的缓冲容量，如果不指定那么就是一个非缓冲通道。缓冲通道有以下机制可以保证并发安全：1.每次只能有一个goroutine可以向通道中发送数据，有多个goroutine需排队发送，如果通道已经满了，试图发送数据的gorotine就会阻塞，直到可以发送数据。2.每次也只能有一个goroutine可以从通道中取数据（随机一个取），取的时候通道为空的话则当前gorotine会阻塞直到可以成功接收数据。非缓冲通道有一些区别：向非缓冲通道中发送或接收数据，当前goroutine都会阻塞，直到有别的goroutine在通道的另一端和它进行握手交换数据。
   3. channel会发生死锁的场景：
      1. 非缓冲通道只写不读
      2. 非缓冲通道读在写后面
      3. 缓存channel写入超过缓冲区数量
      4. 没初始化就读或者写
      5. 多个协程互相等待
   4. 注意避免goroutine泄露：

多个gorotine排队向无缓冲通道发送数据，后面发送的goroutine因为没有其他gorotine来接收而被永久阻塞，这种情况称为goroutine泄露。和垃圾变量不同，泄露的goroutine并不会被自动回收，所以我们应该保证每一个goroutine都能正常退出。还有一些别到的情况比如死锁等导致goroutine资源一直无法释放。

**排查Goroutine泄露**

* + 1. 单个函数：调用 runtime.NumGoroutine 方法来打印 执行代码前后Goroutine 的运行数量，进行比较
    2. 生产/测试环境：使用PProf实时监测Goroutine的数量：

**-import** \_ "net/http/pprof"

**-程序中开启HTTP监听服务** http.ListenAndServe("localhost:6060", nil)

**-执行命令：go tool pprof** -http=:1248 http://127.0.0.1:6060/debug/pprof/goroutine

1. **Goroutine的底层原理**

goroutine是用户态的轻量级的线程（协程），由Go runtime管理而不是操作系统。在源码中是一个结构体，有一个goid字段用来标识（类似进程的唯一Pid）、一个schedule字段是结构体类型用来在切换的时候保存当前goroutine的上下文、一个stack字段也是结构体，存放栈的上下界内存地址。

1. **Golang控制同步的方式**

**主要有六种：**

1. **channel**

channel的存取数据的机制可以用来控制goroutine的同步，且channel是协程安全的不会有数据冲突，比锁好用多了，但是不太适应比较复杂的场景，比如控制多个goroutine的同步问题，而且channel存在死锁问题。（比如存取位置不对，一直阻塞。）

1. **sync.Mutex**
2. **sync.WaitGroup**

有三个函数Add( ), Done( ), Wait( ) （先统一Add再并发Done最后Wait）

1. **sync.Context主要配合select使用**
2. **sync.Pool和sync.Once用的比较少**
3. **context**

context主要用于控制goroutine的同步以及超时控制、取消方法的调用等。特别是在子goroutine之下还有子goroutine的情况下使用起来就很方便。

context的底层结构就是一个叫Context的接口，其中有四个函数，Deadline，Done，Err和Value。Deadline返回一个time包的Time类型，表示当前Context应该结束的时间；Done主要是告诉context相关的函数要停止工作了；Err表示context被取消的原因；Value是context实现共享数据存储的地方，是协程安全的。

context包还有5个函数，Background和TODO用来返回一个Context；另外三个分别是返回带有取消函数和带有截止日期以及带有超时时间段的上下文和一个取消函数。

**具体的使用在标准库的context包中**

1. **Go Module**

Go modules 是 Go 语言的依赖解决方案，是GOPATH的使用模式的进化。主要是解决了GOPATH没有版本控制的概念。现在的gomudle默认都是开启的，只需要在程序运行之前使用go mod init 项目名 生成go.mod文件进行初始化就可以用了。

go mod init 生成 go.mod 文件  
go mod download 下载 go.mod 文件中指明的所有依赖  
go mod tidy 整理现有的依赖  
go mod graph 查看现有的依赖结构  
go mod edit 编辑 go.mod 文件  
go mod vendor 导出项目所有的依赖到vendor目录  
go mod verify 校验一个模块是否被篡改过  
go mod why 查看为什么需要依赖某模块

1. **Go Workspace**

在之前是先将某个模块上传到github然后再需要用到的时候再import下来。而使用Workspace之后能够在本地项目的go.work文件中，通过设置一系列的模块本地路径，再将路径下的模块组成一个当前的工作区。

1. **值类型和引用类型**

值类型通常会在栈上分配：

包括所有的整型int和浮点型float、布尔型、字符串、数组和结构体

值类型存储的就是数据

引用类型通常会在堆上分配：

包括指针、切片、map、通道channel、接口

引用类型存储的是指向底层数据结构的地址，切片、map、通道使用var声明之后必须使用make函数对其进行分配。make之前值都为nil，给它赋值时会导致panic