Go面试

Channel&GMP

1、进程、线程、协程区别

进程：是系统进行资源分配和调度的独立单元，拥有独立的地址空间和资源。是具有独立功能的程序上的一次运行活动，进程间通信需要借助操作系统提供的机制。
线程：是进程中的一个执行单元，是CPU调度的基本单位。线程共享进程的资源，但有自己的栈和寄存器，线程间通信开销较低。
协程：是更轻量级的线程，调度由用户控制。协程拥有自己的上下文，切换由程序控制，开销极小，适合高并发场景。

线程和协程：
轻量级-Goroutine初始堆栈空间2KB，且按需增大缩小；线程占用一定的内存和CPU资源，堆栈大小通常是固定的，如2MB
数量限制-可以创建大量的Goroutine；过多的线程可能会导致资源耗尽或性能下降
创建与销毁开销-Goroutine的创建和销毁开销较小；线程有自己的堆栈和上下文信息，创建和销毁则需要更多的资源
调度方式-Goroutine由Go运行时（runtime）进行调度，可以在用户空间中进行，效率更高，调度器使用了GMP；线程的调度是操作系统的内核调度器，在用户模式和内核模式之间切换会有一定的开销
并发模型-Goroutine采用了一种称为M:N的模型，即M个内核态线程可以映射到N个Goroutine/用户态上；线程则采用了一种1:1的模型，即每个线程都映射到一个操作系统线程上
通信与同步-Goroutine之间的通信通过channel，通信共享内存；线程之间的通信可能会涉及到共享内存的同步和互斥机制

2、golang并发优势

- 从调度上看，goroutine远远小于线程，而且是用户空间，采用M:N调度技术，M个内核态线程可以映射到N个Goroutine/用户态。线程切换需要一个完整的上下文切换：即先保存一个线程的状态到内存，再恢复另外一个线程的状态，最后更新调度器的数据结构。
- 从栈空间上，goroutine的栈空间更加动态灵活。线程有固定大小的栈内存差不多2Mb，goroutine生命周期开始只有2Kb，不是固定大小，按需增大缩小

3、go的channel实现原理

channel 是一种内置的数据类型，由运行时系统（runtime）负责管理。它允许两个或多个 goroutine 通过它进行通信，确保并发编程的安全。

内部同步
Go的channel实现依赖于内部同步机制来确保数据的一致性和线程安全。具体来说：
互斥锁：channel内部数据结构和缓冲区通过互斥锁（mutex）或其他同步原语来保护，以确保在多个goroutine同时访问通道时不会出现数据竞争。
条件变量：为了实现发送和接收的阻塞和唤醒机制，Go使用条件变量来管理发送和接收操作的等待状态。当通道的状态满足特定条件（如缓冲区有空间或有数据可读）时，会唤醒等待的goroutine。
channel 收发遵循先进先出 FIFO，分为有缓存和无缓存，**channel 中大致有 buffer(当缓冲区大小部位0 时， 是个 ring buffer)、sendx 和 recvx 收发的位置(ring buffer 记录实现)、sendq、recvq 当前 channel 因为缓冲区不足而阻塞的队列、使用双向链表存储、还有一个 mutex 锁控制并发**

4、channel底层结构/数据结构了解吗

channel由以下数据结构组成：
缓冲区：用于存储通道中的数据。缓冲区的大小在创建通道时确定，对于带缓冲的通道，缓冲区允许在不需要立即同步的情况下存储多个数据项。对于无缓冲的通道，缓冲区为空。
队列：对于带缓冲的通道，内部有一个数据队列来存储待发送和待接收的数据。对于无缓冲的通道，数据队列通常为空，只有在发送和接收操作同时存在时才有数据。
发送和接收队列：用于管理等待发送和接收的goroutine。当通道的缓冲区满时，写操作会阻塞直到有读操作；当缓冲区为空时，读操作会阻塞直到有写操作。

```
type hchan struct {
    qcount   uint           // 循环数组中的元素数量，长度
    dataqsiz uint           // 循环数组的大小，容量
    buf      unsafe.Pointer // 指向底层循环数组的指针（环形缓冲区）
elemsize uint16
    closed   uint32
    elemtype *_type // 元素类型
    sendx    uint   // 下一次写的位置
    recvx    uint   // 下一次读的位置
    recvq    waitq  // 读入元素队列
    sendq    waitq  // 写入元素队列
    lock mutex      // 互斥锁，chan不允许并发读写
}
```

5、已关闭channel读写

- 如果向一个已关闭的channel发送数据，Go运行时会panic。

- 如果尝试从一个已关闭的channel接收数据：
  如果通道中还有数据，读取操作会正常返回数据。
  如果通道中已无数据，读取操作会返回通道的零值

6、有缓冲和无缓冲channel的区别?

无缓冲：每次发送完数据，必须要等到接收到数据后，才会继续下一次数据的发送
有缓冲：发送数据端不必等待数据被接收后才进行下一次发送，可以直接进行

7、channel的特性，从nil的channel中取数据会发生什么

有缓冲，可以单方面读写，缓冲区未满则不阻塞
无缓冲，单方面读写都会阻塞，
nil
给一个 nil channel 发送/接收数据，造成永远阻塞
给一个已经关闭的 channel 发送数据，引起 panic
从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值
无缓冲的channel是同步的，而有缓冲的channel是非同步的

8、怎么优雅地关闭通道

1. 向各个goroutine发通知，令其退出，如shutdown.
2. 等各个goroutine 退出，如: sync.WaitGroup的Done
3. 退出goroutine 之前，确保数据不丢失
   （1.停止生产数据。2.关闭数据channel messages. 3. 消费者goroutine检查判断数据channel messages是否有效，若无效，则退出。）

9、容量为1的channel在什么情况下会堵塞

在一个容量为1的通道中，堵塞通常是由于通道已满（在生产者尝试写入时）或通道为空（在消费者尝试读取时）造成的

10、channel的使用场景

1. **并发通信**：在并发编程中，channel 用于在多个线程或协程之间传递数据，确保安全的通信和数据同步。 日志分析
2. **任务协调**：用于在生产者-消费者模型中协调任务，生产者将任务放入 channel，消费者从 channel 中取出任务处理。 socket
3. **数据流控制**：在数据流处理场景中，channel 用于控制数据的流动和处理，例如在流式数据处理中传递数据块。 数据库分库分表
4. **事件通知**：用于传递事件信号或状态更新，比如在 UI 线程与后台线程之间传递事件通知。 websocket通知前端
5. **资源共享**：协作式多任务处理中，用于不同任务间的资源共享和协作。 日志合并

11、如何避免channel导致的死锁

select-case语句添加default、有缓冲

12、channel在什么情况下会panic?

1.向一个已经关闭的channel发送数据：当尝试向一个已经被关闭的channel发送数据时，会导致运行时panic。
2.关闭一个已经关闭的channel：重复关闭同一个channel也会导致panic。
3.关闭一个nil的channel：尝试关闭一个未被初始化的（nil）channel同样会导致panic。
4.关闭包含阻塞发送操作的channel：如果有一个或多个goroutine正在阻塞等待向一个channel发送数据，而另一个goroutine突然关闭了该channel，这会导致那些阻塞的goroutine尝试向已经关闭的channel发送数据，从而触发panic。
5.从未初始化的 channel 读取数据：如果从一个未初始化或未分配的 channel 读取数据，程序会引发 `panic`。

13、协程间通信

1. 通道（Channel）
   在Go语言中，通道（Channel）是协程间通信的主要机制。通道是一个类型化的管道，用于在不同的协程之间安全地传递数据。

- 声明与初始化：通道通过make函数进行初始化，例如ch := make(chan int)。
- 发送与接收：数据通过<-操作符在通道中发送和接收，例如ch <- value（发送）和value := <-ch（接收）。
- 阻塞：默认情况下，通道的发送和接收操作是阻塞的，直到对方准备好。
- 关闭通道：通过close(ch)关闭通道，表示没有更多的值将被发送到通道中。

2. Context
   context.Context是Go标准库中的一个类型，主要用于控制goroutine的生命周期，但也可以间接用于协程间的通信

- 使用场景：当需要优雅地取消长时间运行的goroutine或传递超时时间时，可以使用Context。
- 取消操作：通过context.WithCancel、context.WithTimeout等函数创建的Context，可以调用返回的cancel函数来取消操作。

3. sync包中的同步原语
   sync包中的同步原语（如sync.WaitGroup、sync.Cond等）主要用于同步协程的执行，间接地用于协程间的通信

- sync.WaitGroup：用于等待一组goroutine完成。通过调用Add方法增加计数，在goroutine结束时调用Done方法减少计数，并在主goroutine中调用Wait方法等待所

有goroutine完成。
- sync.Cond：一个条件变量，用于在多个goroutine之间协调等待/通知操作。

14、怎么实现并发模型?

Golang中常用的并发模型有三种：

- 通过channel通知实现并发控制，通过关键字 `go` 来启动一个并发任务，channel 用来在不同的goroutine之间同步、通信
- 通过sync包中的WaitGroup实现并发控制，Goroutine是异步执行的，为防止main函数结束的时候终止Goroutine，用 WaitGroup，通过Add/Done/Wait实现等待
- Context上下文，实现并发控制

15、怎么处理协程运行的超时问题

使用context包: context包提供了超时WithCancel和取消机制WithTimeout。创建一个带有超时限制的`context`，如果协程在指定时间内没有完成，可以通过取消context来中断协程的执行。
使用`time.After`或`time.NewTimer`**: 设置一个超时时间，然后在一个select语句中等待协程完成或超时事件发生。如果超时了，执行相应的逻辑

16、主协程如何等待其他协程完成后再操作

同14
1. **使用 `sync.WaitGroup`**
`sync.WaitGroup` 是一种同步原语，用于等待一组协程完成工作。调用 `Add` 计数协程数量，使用 `Done` 通知 `WaitGroup` 完成，然后使用 `Wait` 方法阻塞主协程，直到计数为0所有协程完成。
2. **使用 Channel 进行同步**
通过创建一个或多个 channel 来实现主协程等待其他协程完成。每个协程在完成任务后发送一个信号到 channel，主协程则从 channel 中接收这些信号以确认所有协程已经完成。
3. **使用 `context.Context`**
`context.Context` 提供了一个机制来控制多个协程的生命周期。你可以使用 `context.WithCancel` 或 `context.WithTimeout` 来创建一个上下文，并在协程中监听上下文的取消信号。主协程可以等待所有协程完成或上下文取消信号。

17、golang协程的调度，聊聊对GMP模型的理解

在GMP模型中，M代表工作线程，是运行Goroutine的实体，P在GMP模型中代表处理器，包含了运行Goroutine的资源，每个P都有一个本地队列，用于存储等待运行的Goroutine。
当需要执行一个Goroutine时，系统首先会尝试找到一个空闲的P和一个与之关联的M。

然后，M从P的本地队列中获取Goroutine并执行。如果P的本地队列为空，M会尝试从全局队列获取Goroutine，或者从其他P的本地队列中“偷取”Goroutine。
这种设计使得Goroutine的调度更加轻量级和高效，减少了线程切换的开销，从而提高了并发性能

18、介绍一下GMP模型，m和p的关系

GMP 模型（Goroutine, Machine, Processor）是 Go 语言的并发调度模型，三种主要组件：goroutines（G）、machines（M）和processors（P）
M 和 P 的关系

- **P 的角色:** Processor（P）负责调度 goroutines。它维护一个本地队列（run queue），其中包含了要执行的 goroutines。P 还负责从队列中取出 goroutines 并将其分配给机器（M）执行。
- **M 的角色:** Machine（M）是实际的系统线程，用于执行 goroutines。每个 M 可以与一个 P 绑定，利用 P 提供的队列来获取和执行 goroutines。
- **绑定关系:** 一个 M 可以与一个 P 绑定，在这种情况下，该 M 从绑定的 P 的队列中获取 goroutines 并执行它们。如果一个 P 没有绑定的 M，则它会创建一个新的 M 进行工作。反之，当 M 执行完任务后，它会回到系统线程池中等待下一个任务。

19、gmp的设计原理和优势

- **Goroutine的轻量性**：Goroutine在创建时只需要很少的内存（初始栈空间通常为2KB），可以在大量的Goroutine之间进行调度。它们的创建和销毁比操作系统线程更高效，因为创建和销毁Goroutine的开销远低于操作系统线程。
- **协作式调度**：Go的运行时系统采用协作式调度方式来管理Goroutine。每个Goroutine会在某些点上进行调度（例如I/O操作、显式的`runtime.Gosched`调用等），这样可以确保调度器能够在适当的时机让其他Goroutine获得执行机会。
- **M:N调度模型**：GMP模型是一种M:N调度模型，其中M个操作系统线程（Machine）可以调度N个Goroutine。这种设计使得Go可以高效地管理大量Goroutine，而不会像传统线程模型那样因为线程数量过多而导致系统资源耗尽。
- **工作窃取**：Processor会维护一个本地的Goroutine队列，当本地队列空闲时，Processor可以从其他Processor的队列中窃取任务。这种工作窃取机制提高了Goroutine的调度效率，避免了负载不均的问题。
  优势:
- **高效的并发执行**：通过轻量级的Goroutine和有效的调度机制，更小的栈空间允许创建大量的实例
- **动态调整**：Go的调度器可以动态调整运行时环境中的Goroutine和Processor数量，以适应不同负载的需求
- **降低上下文切换开销**：与传统线程模型相比，Goroutine的上下文切换开销要小得多，Goroutine的调度和切换是在用户态完成的，避免了内核态和用户态的切换导

致的成本

20、gmp模型调度顺序

- 全局队列与本地队列：新创建的Goroutine首先被放入全局队列中。每个P（Processor）维护一个本地队列，优先从本地队列中获取Goroutine执行。
- 工作窃取：如果P的本地队列为空，它会尝试从全局队列或其他P的本地队列中“窃取”Goroutine来执行，以实现负载均衡。
- 线程与处理器的绑定：M（Machine，即操作系统线程）必须持有P才能执行Goroutine。M从P的本地队列中依次获取Goroutine并执行，形成循环。
- 系统调用处理：当Goroutine进行系统调用时，M会被阻塞，P会转而调度其他M执行其他Goroutine。系统调用完成后，M会尝试重新获取P继续执行Goroutine。
  这种调度顺序确保了Goroutine能够高效地在多个M之间切换执行，充分利用多核CPU资源，同时减少线程切换的开销

21、GMP模型 Work Stealing偷多少

Work Stealing-工作窃取
Go的调度器在进行工作窃取时，通常会窃取**本地队列一半的Goroutine。**

22、抢占式调度

允许操作系统或调度器强制中断正在运行的任务

1. 使用`runtime.Gosched()`来主动让出CPU，让出当前的P（P表示M的本地队列），让出M（M表示系统线程），然后再调度器的合适时机，让其他goroutine运行。
2. 使用`runtime.LockOSThread()`和`runtime.unlockOSThread()`来绑定和解绑一个系统线程，使得在这段时间内，这个goroutine总是运行在同一个系统线程上，从而实现类似于抢占式调度的效果。

23、gmp有了本地队列，为什么还要全局队列，为什么不直接从全局队列拿

全局队列用于存储那些未被任何Processor的本地队列处理的Goroutine。主要作用是：

1. **处理负载不均**：当某些Processor的本地队列变得空闲时，其他Processor可以从全局队列中获取Goroutine，从而平衡负载。
2. **支持工作窃取**：当一个Processor的本地队列空了，它可以从全局队列中窃取Goroutine，确保所有Processor都有任务可做，增加系统的整体吞吐量和响应性。
3. **避免任务遗漏**：全局队列确保了所有待执行的Goroutine都不会被遗漏，即使本地队列的任务处理不均衡。
   原因如下：
4. **性能开销**：本地队列的操作开销更小，能提供更高的性能。
5. **调度效率**：本地队列使得Goroutine的调度更高效。

6. **负载均衡**：全局队列和工作窃取机制结合使用，可以有效地平衡各个Processor的负载，而不需要每个Processor都直接访问全局队列。
   工作窃取策略允许空闲的Processor从全局队列中获取任务，而繁忙的Processor则优先处理自己的本地队列。

24、goroutine的状态机

描述和管理goroutine的生命周期及其状态转换：

1. **Running**：`goroutine`正在运行并执行代码。—goroutine从运行队列中被GMP调度选中
2. **Runnable**：`goroutine`准备好执行，但由于调度器的调度限制，暂时没有被执行。处于这个状态时，它在调度队列中等待。—新的goroutine被创建时
3. **Waiting**：`goroutine`在等待某些条件，如等待锁、等待I/O操作完成等。在这个状态下，它会被挂起，直到条件满足。—等待一个channel操作、锁或定时器
4. **Blocked**：`goroutine`被阻塞，通常由于同步原语（如通道、互斥锁）的竞争。
5. **Dead**：`goroutine`已经完成执行，或者由于某种原因被终止，无法继续运行。
   当一个新的goroutine被创建时，它会被设置为Grunnable状态并加入到运行队列中。当调度器选择一个goroutine执行时，它会将goroutine的状态从Grunnable转换为Grunning。当goroutine执行系统调用或进入阻塞状态时，它的状态会相应地改变

25、GMP模型相比于正常的协程-线程-进程调度，有哪些优缺点?

GC、RPC、gRPC

1、 golang的GC

是自动的，用于自动释放不再使用的内存，具体实现是：三色标记法 + 混合写屏障。
采用并发标记清除算法，具体实现是：三色标记法 + 混合写屏障，在整个垃圾回收的过程中，不需要启动STW，效率很高。
三色标记算法则用于解决标记-清除算法中的内存碎片化问题，通过将对象标记为白色、黑色或灰色来优化回收过程。
混合写屏障机制通过插入写屏障和删除写屏障的结合，确保对象引用的正确性，同时减少STW（Stop The World）的时间。

工作方式
三色标记法具体过程：
程序刚开始时，所有对象默认标记为白色；从根节点开始遍历，所有可达对象由白色标记为灰色，并将灰色对象放到灰色标记表中；遍历灰色标记表，将灰色对象标记为黑色，并将其由灰色标记表移动到黑色标记表中；最后回收所有白色对象，即垃圾对象
混合写屏障步骤：减少STW的时间
初始标记：将所有栈上的对象标记为黑色，新创建的栈对象也标记为黑色。

并发标记：在标记过程中，新插入的对象和删除的对象都被标记为灰色，以满足三色不变式，确保垃圾回收的正确性。
STW（Stop The World）：
在某些阶段，如标记阶段的开始和结束，可能需要暂停所有goroutine的执行，以确保GC的正确性。这种暂停称为STW。
垃圾回收的`STW`停顿主要发生在标记阶段和清理阶段。标记阶段的频率由`GCTime`控制，而清理阶段在标记阶段之后立即进行

2、Go语言的内存分配和垃圾回收机制?

Go语言内存分配主要通过堆（heap）和栈（stack）来管理，堆用于分配全局对象和动态生成的对象，而栈则用于局部变量和函数调用。堆的内存通过内置函数new和make分配，由gc管理；栈内存由程序控制，调用结束自动释放。
垃圾回收机制方面，Go语言的gc是自动的，采用并发标记清除算法，具体实现是：三色标记法 + 混合写屏障，分为**标记阶段**和**清除阶段**，使用白色、灰色和黑色三种颜色来标记对象，通过扫描根对象集合，逐步将所有可达的对象标记为灰色和黑色，最终回收所有白色对象，即垃圾对象 。

3、gc三色标记和存活对象颜色

GC 的三色标记算法用于管理对象的存活状态，包括：

1. **白色**：表示尚未被标记的对象，即可能是垃圾的对象。
2. **灰色**：表示已经被标记但其引用的对象还未被完全遍历的对象。
3. **黑色**：表示已经完全标记并且其引用的对象也已被遍历的对象。
   过程如下：

- **标记阶段**：从根对象开始，标记所有可达的对象为灰色。
- **遍历阶段**：处理灰色对象，将其引用的对象标记为灰色，然后将灰色对象标记为黑色。
- **清除阶段**：遍历堆，清除所有仍然是白色的对象，这些对象被认为是不再使用的。
  优化垃圾回收，减少停顿时间，并确保内存中的活跃对象被正确保留

4、gc出现的写屏障、gc阶段

1. 写屏障（Write Barrier)

写屏障是在并发环境下处理内存写入操作的机制。主要作用是确保GC能够准确追踪和标记所有活跃对象，尤其是在对象引用更新时。

- 插入写屏障：在标记开始时无需STW，可直接开始，并发进行，但结束时需要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：在GC开始时STW扫描堆栈来记录初始快照，这个过程会保护开始时刻的所有存活对象，但结束时无需STW。

2. GC阶段：
在标记阶段，GC会遍历所有对象，标记出存活的对象；
在清除阶段，GC会回收那些未被标记的对象所占用的内存空间

5、gc的对象是哪的，找根对象从什么位置开始找

GC（垃圾回收）的对象主要是堆内存中的对象
主要包括**全局变量、执行栈和寄存器**中的值，从这些对象开始遍历，逐层跟踪所有可达对象，以确定哪些对象是活动的、哪些是垃圾需要回收。

6、垃圾回收触发时机

1. **内存分配触发**：当Go程序向操作系统申请内存空间，且当前可用内存不足以满足分配请求时，会触发GC来释放不再使用的内存。
2. **内存占用触发**：当Go程序占用的内存达到一定阈值，触发GC减少内存占用。这个阈值可以通过环境变量GOGC来调整，默认为100，即每当内存扩大一倍时启动GC。
3. **定时触发**：调用runtime.GC()函数来手动触发GC，但这种方式会阻塞用户协程，直到垃圾回收流程结束。默认情况下，最长2分钟触发一次GC。
4. **手动触发**：调用runtime.GC()函数来手动触发。

7、rpc通信

在不同计算机系统之间进行远程过程调用，远程方法调用，独立于主进程，崩溃不影响主进程
RPC定义：RPC（Remote Procedure Call）即远程过程调用，是一种允许程序调用另一台计算机上的过程或函数的协议
RPC原理：RPC采用客户机/服务器模式，客户端通过本地调用发送请求到服务器，服务器处理请求后返回结果
RPC优势：RPC简化了分布式系统的开发，提高了系统的可扩展性和可维护性，适用于微服务架构和大型分布式系统。
https://blog.csdn.net/poiuytrewq213/article/details/136348428?spm=1001.2014.3001.5502

8、基于TCP的rpc协议如何实现

https://blog.csdn.net/poiuytrewq213/article/details/136348428?spm=1001.2014.3001.5502
解决粘包：

1. **固定长度消息**
2. **长度前缀**
3. **分隔符**

4. **协议设计**

9、gRPC是如何实现通信的?

gRPC是一个高性能、开源和通用的远程过程调用（RPC）框架，基于 HTTP/2 协议和 Protocol Buffers（protobuf）数据序列化协议，得客户端在调用服务器上的方法时，就像是在调用本地对象一样
调用过程：

- 客户端通过本地调用方式使用服务接口，而gRPC负责将这些调用路由到正确的服务器。
- 服务器处理请求并返回响应。在网络上传输的消息都被protobuf序列化和反序列化

10、Grpc实现过程中经历了哪些层次?

1. **建立连接**：首先，gRPC客户端和服务器端需要建立连接。包括建立TCP连接，其中涉及设置连接参数、解析器获取服务器地址列表、以及平衡器根据服务器地址列表建立TCP连接。
2. **帧交互**：在连接建立后，gRPC服务器端需要发送帧大小、窗口大小等信息给客户端。客户端接收到这些信息后，会更新本地的帧大小、窗口大小等。
3. **请求-响应模型**：gRPC采用请求-响应模型，涉及发送请求头、实际RPC消息、数据帧等多个步骤。请求头包含元数据，如目标服务、方法等。实际RPC消息可能分为多个数据帧传输，取决于消息大小和传输协议限制。
4. **服务端流式RPC与客户端流式RPC**：gRPC支持两种流式RPC，即服务端流式RPC和客户端流式RPC，允许双向的数据传输。
5. **结束通信**：一旦数据传输完成，服务端通过发送带有状态详细信息的Trailers头来结束通信。

11、讲一下protobuf，为什么使用pb不使用json?

用于结构化数据的序列化、反序列化和传输，高效地编码和解码，Protobuf数据体积比 JSON 小，解析速度通常比 JSON 更快

1. **紧凑性**：protobuf 使用二进制格式，数据体积比 JSON 小，传输和存储效率更高。
2. **性能**：由于其二进制格式，protobuf 的解析速度通常比 JSON 更快。
3. **类型安全**：protobuf 定义明确的消息类型和字段类型，提供了更强的类型安全。
4. **版本兼容**：protobuf 支持向前和向后兼容，允许在消息格式演进时保持兼容性。

12、golang的http路由实现

net/http提供HTTP 路由功能
使用标准库`net/http`中的`ServeMux`，这是HTTP请求路由器（或者说是多路复用器）。

`ServeMux`会将收到的请求与一组预定义的URL路径进行匹配，并调用`http.Handle` 来将路径与处理函数关联

13、golang程序hang/卡住了可能是什么原因，怎么排查

死循环导致无法GC、死锁
用pprof

14、有什么办法可以获得函数调用的链路

`runtime.Callers`函数可以用来获取当前goroutine的调用栈信息，`runtime.CallersFrames`将程序计数器转换为帧对象。然后，它遍历这些帧对象并打印出每个函数的名称、文件路径和行号，返回`runtime.Frames`实例，可以用来迭代栈帧信息。每个`Frame`包含了文件名、行号和函数名等信息。

Struct&Slice&Map

1、内置类型有哪些

数值、布尔、字符串

2、引用类型的特点

引用类型包括切片（slice）、映射（map）、通道（channel）和接口（interface）
参数传递时，传递的是值的本身，而不是拷贝
引用类型的零值是 `nil`，表示未初始化或没有指向任何内存地址

3、定义两个int的地址一样吗

不一样

4、Go里有哪些数据结构是并发安全的？int类型是并发安全的吗？

- **并发安全的数据结构**：包括 `sync.Mutex`、`sync.RWMutex`、`sync.Map` 和 `chan`。
- **基本类型（如 `int`）**：在并发环境中不是并发安全的，需要使用原子操作或锁来确保安全。

5、Go的struct能否进行比较

对 `struct` 进行比较，取决于结构体中的字段及其类型

- 相同struct类型的可以比较
- 不同struct类型的不可以比较，编译都不过，类型不匹配

6、全局定义两个不同的空结构体，地址是否相同，如果定义两个空结构体分别在不同的结构体中呢

地址通常是不同的。这是因为每个变量在内存中都有其独立的存储位置，即使它们是空结构体也不例外

7、如何判断一个结构体是否实现了某接口?

如果一个类型 T 实现了一个接口中的所有方法，那么我们就可以说类型 T 实现了接口
通过编译时检查来确认结构体是否实现了某个接口
使用类型断言或反射来进行运行时检查，多用于动态类型和调试。
`if _, ok := i.(MyInterface)`
`` `reflect.TypeOf(i).Implements()`方法用于检查`i`是否实现了`MyInterface`接口` ``

8、interface的两种用法？了解interface的底层实现吗?

interface两种用法：
空接口，所有的类型都实现了空接口；
非空接口-有方法的接口，golang的接口是针对类型的，当一个类型要实现接口，就得实现该接口的所有方法
或
作为方法接收者: 接口定义了对象可以实现的方法集。
作为动态类型: 接口类型可以存储任何实现了该接口的具体类型。

interface底层实现：

- `_type`: 指向描述实际类型的指针。
- `data`: 指向指向该类型数据的指针。对于具体类型，`data` 指向对象本身；对于空接口，`data` 为空。
  使得接口可以动态地存储和调用任何类型的方法，实现了 Go 语言的多态性

9、Slice的底层实现

切片的底层实现包含三个关键字段：
1.**指针**：指向底层数组的第一个元素。
2.**长度**：切片中元素的数量。
3.**容量**：从切片的开始可访问的元素总量，即Slice可以容纳的元素最大数量。

```
type slice struct {
        array unsafe.Pointer 指针
        len   int   长度
        cap   int       容量
}
```

这三个字段共同定义了切片的动态特性和边界。切片是引用类型，这意味着当创建或修改切片时，实际上是在操作底层的数组

10、slice的长度、容量、共享和扩容机制

切片的长度和容量：

- **长度（Length）**：切片的长度是指切片中当前包含的元素数量。可以通过内建函数 `len()` 获取。
- **容量（Capacity）**：切片的容量是指可访问的元素总量。可以通过内建函数 `cap()` 获取。
  切片的共享：
  切片与底层数组共享内存。这意味着对切片的修改会影响底层数组，并且从该数组创建的其他切片也会看到这些修改。
  扩容机制：
- 如果当前容量小于 1024，则新容量为原容量的两倍，增长因子2。
- 如果当前容量大于 1024，则新容量增加量为原容量的 1/4，直到达到一定的上限，增长因子1.25。

11、Slice与数组的区别

1. **长度与容量**：数组具有固定长度，声明后不能改变；而slice的长度可变，且拥有一个容量属性，表示slice可以增长到的最大长度。
2. **类型与传递**：数组是值类型，传递时复制整个数组；slice是引用类型，传递时仅复制slice描述值（包含指针、长度和容量），不复制底层数组。
3. **内存共享**：多个slice可以共享同一个底层数组，修改其中一个slice的元素可能会影响到其他slice；而数组之间则是独立的，互不影响。

12、修改底层数组的值，切片的值是否改变

会，切片和数组共享同一个底层数组。当通过切片修改底层数组的值时，所有指向该底层数组的切片都会受到影响，它们的值也会相应地发生改变

13、不同的Slice在复制和传值时，是深拷贝还是浅拷贝？

浅拷贝
对于引用类型的数据，浅拷贝只复制了引用，而不是创建一个新的对象。当其中一个对象修改了共享属性的值时，另一个对象也会受到影响。当一个对象销毁时，内存块会被释放，其他指向的对象销毁时会由于释放一个悬空指针而出错。

14、slice底层原理，如何从slice中删除数据

用`append`函数和切片表达式来创建一个新的切片

15、make一个len为0的切片，获取数据有没有问题

获取长度为0的切片的数据不会报错，但由于切片为空，访问其中任何元素都会导致运行时错误
先检查其长度，避免访问空切片中的元素

16、slice是[]int{1,2}，把它传入一个函数，修改第一项的值为3，函数结束，原来slice值改变了吗

变了，传入函数的切片会修改原来的切片，因为切片是引用类型。在函数中改变切片的内容时，原切片的值也会相应改变。

17、如果函数新建一个list，list=append(slice,3)，调用完这个函数后在函数的外部打印这个原来slice的长度是多少，新的是多少，地址改变了吗，指向的原来的数组呢

如果原切片的底层数组有足够的容量，`append` 操作会在原底层数组中直接扩展切片，并返回一个新切片。
如果容量不足，它会分配一个新的底层数组，将原有元素复制过去，然后在新数组的末尾添加新的元素。

1. **不需要扩展**: 如果 `append` 操作不导致底层数组的扩容（即新元素可以在当前容量内添加），`append` 会在原有的底层数组上进行操作，影响原切片。
2. **需要扩展**: 如果 `append` 需要增加底层数组的容量（即当前容量不足以容纳新元素），Go 会分配一个新的底层数组，将原数组的元素复制到新数组中，然后在新数组中添加新元素。此时，`append` 返回的切片会引用新的底层数组，而原切片仍然引用旧的底层数组。

18、funcA向funcB传slice，funcB中对slice append元素并返回，funcA中打印该slice会受影响吗

- 如果没有发生扩容，`funcB` 修改切片会在 `funcA` 中反映出来。
- 如果发生了扩容，`funcB` 的修改不会影响 `funcA` 中的原切片。

19、map如何顺序读取

`map`是无序的，要顺序读取`map`中的数据，先将`map`的键（key）存储到一个切片（slice）中，对切片进行排序，然后按照这个顺序来读取`map`中的数据。

20、map里面的数据怎么存的，怎么读的，发生哈希冲突后值存在哪

在Go语言中，`map` key-value键值对，存储在哈希桶中，哈希桶内部会根据哈希函数将键分配到不同的桶中。

- **数据存储**：首先通过hash(key)得到下标位置，然后根据下标位置存储对应的value
- **读取数据**：使用哈希函数计算给定键的哈希值，根据哈希值确定该键应该存储在哪个哈希桶中，在确定的哈希桶中查找具体的键。如果找到匹配的键，则返回相应的值。如果未找到，则返回零值。

如果发生哈希冲突（即不同key产生相同的hash），则采用链地址法解决。链地址法则将冲突的value以链表形式存储，并用next指针指向下一个存储位置。

21、map底层实现怎么处理hash碰撞的问题?

- 采用链地址法
- hash碰撞-不同的键可能会哈希到相同的存储位置
  冲突处理：
  - 当插入一个新键值对时，先计算键的哈希值，然后确定该键应该存储在哪个桶中。
  - 如果该桶已经存在键值对（即发生了哈希冲突），将新键值对插入到桶内部的链表中。
  - 查找键时，会首先定位到正确的桶，然后在桶内部的链表中搜索对应的键。

22、map并发不安全为什么会panic，int并发出错会不会panic，为什么

`map` 的内部结构（如哈希表）在并发读写时可能会被破坏，导致数据不一致或程序崩溃。
`int` 类型是基本数据类型，并不涉及复杂的内部结构。

23、map是并发安全的吗，会出现什么问题

不是。当多个goroutine同时对同一个map进行读写操作时，可能会导致数据竞争（Data Race）问题，程序可能会panic

24、检查数据竞争

数据竞争通常发生在多个goroutine并发读写同一数据时，而没有适当的同步机制来避免冲突
Go提供了内置的数据竞争检测器（Race Detector），可以通过在编译和运行程序时添加`-race`标志来启用
1.**编译时启用数据竞争检测**：go build -race -o myapp 会编译一个带有数据竞争检测功能的程序版本
2.**运行时启用数据竞争检测**：go run -race main.go 会在运行时监视内存访问模式，并报告任何检测到的数据竞争
3.**测试时启用数据竞争检测**：go test -race mypkg 会运行指定的包中的所有测试，并报

告任何数据竞争

你会看到类似于以下的数据竞争报告：

```
==================
WARNING: DATA RACE
Read at 0x00c0000a6000 by goroutine 7:
  main.increment()
      /path/to/your_program.go:10 +0x34

Previous write at 0x00c0000a6000 by goroutine 6:
  main.increment()
      /path/to/your_program.go:10 +0x34
```

解决数据竞争

1. **使用同步机制：** 使用 `sync.Mutex` 或 `sync.RWMutex` 来保护共享数据。
2. **使用 channel：** 利用 channel 进行同步，可以有效地避免数据竞争。
3. **避免共享状态：** 尽量设计无共享状态的并发程序，减少数据竞争的风险。

25、怎么实现并发安全的map（分桶加锁）

1.使用sync.Map：
2.使用互斥锁（sync.Mutex或sync.RWMutex）：
保证了同时只有一个goroutine可以访问共享资源。sync.RWMutex是一个读写互斥锁，它允许多个goroutine同时读取共享资源，但写入操作仍然是互斥的。
3.使用原子操作：
使用原子操作来避免锁的使用。sync/atomic包中的原子函数
4.分桶加锁（bucket-level locking）：
将 `map` 分成多个桶，每个桶有自己的锁，减少锁争用，特别适合读多写少的场景

26、多线程同时读写map中不同的key，一个线程只会读写一个key，会发生什么

竞态
竞态条件是指程序的行为或输出依赖于执行的顺序或时间，当多个goroutine并发访问相同的内存位置且至少有一个goroutine在写入时，就可能发生竞态条件。竞态条件可能导致不可预测的行为，因为不同的goroutine可能以不同的顺序执行，导致每次运行程序时结果都可能不同。

数据竞争是指两个或多个 Goroutine 并发地访问同一块内存位置，并且至少有一个访问是写操作，但没有适当的同步机制来确保对这块内存的访问是安全的。两个 Goroutine 同时读取和写入一个共享变量，而没有使用互斥锁或其他同步机制来保护这个变量。
数据竞争是一种 **竞态条件**，它具体表现为对共享数据的不安全访问，竞态条件可能导致数据竞争（data race）。所有数据竞争都是竞态条件，但不是所有竞态条件都涉及数据竞争。竞态条件可能还包括其他类型的资源竞争和状态不一致问题。

27、怎么用的map，会有并发问题吗？怎么求一个map的长度。sync.Map可以用len来求长度吗？

- map，k-v键值对，通过make初始化，直接赋值，通过key访问数据，delete删除数据
- 会，多个 Goroutine 同时写入，加锁，互斥锁（Mutex）、读写互斥锁（RWMutex）或者 sync.Map
- len
- Range函数中计数

28、普通的map加锁使用与直接用sync.map有什么区别？

- 使用，map需要额外的同步手段，如 `sync.Mutex`，加锁后得解锁；sync.map无需手动管理锁，直接store就行
- 性能，map在高并发写入场景下，需要手动处理锁，可能存在性能瓶颈； sync.map针对高并发读写进行了优化，适用于频繁的并发操作
- 使用场景，map适用于单线程或低并发的场景，简单且直观；sync.map适用于高并发读写的场景，无需过多关注锁的管理

29、sync.Map的底层结构

`sync.Map` 数据结构包括一个 **读缓存** 和一个 **写桶**。读缓存是一个按键组织的快速访问结构，而写桶则包含实际的键值对。

```
// sync.map底层数据结构
type Map struct {
        mu Mutex                           //互斥锁，用于保护dirty map的并发访问
        read atomic.Value                  //只读map，通过原子操作访问，充当高速缓存，用于快速读取数据
        dirty map[interface{}]*entry       //可写map，用于存储最新写入的key-value数据，访问时需要加锁
        misses int                         //计数器，记录从read中读取数据未命中
```

```
的次数，用于决定何时将dirty map的数据同步到read map
}
```

30、sync.map锁，和go的锁有什么区别

- **锁的粒度**: `sync.Map` 的设计通过读缓存和写桶减少了锁的使用，特别是对读操作几乎没有锁的开销。相比之下，使用 `sync.Mutex` 或 `sync.RWMutex` 来保护普通的 map 需要在每次访问时加锁，这会影响性能。
- **性能优化**: `sync.Map` 针对并发读取进行优化，适合高并发读操作的场景，而 `sync.Mutex` 和 `sync.RWMutex` 更适合写操作较多的场景。
- **使用场景**: `sync.Map` 是为了在高并发情况下提供一个高效的解决方案，而 `sync.Mutex` 和 `sync.RWMutex` 通常用于更广泛的同步需求，不仅限于 map。
- 如果写多，map加锁更合适，sync.map更耗时

31、Go如何实现一个单例模式?

1. 使用 `sync.Once` 实现线程安全的单例
2. 使用懒加载和包级变量

32、sync.Once是如何实现的，如何不使用sync.Once实现单例模式?

确保函数只被执行一次，依赖于底层的 `sync` 包中的 `sync.Mutex` 和一些原子操作

- 它包含一个互斥锁（`sync.Mutex`）和一个标志位（通常是一个 `uint32` 类型的 `done` 字段），用于记录函数是否已经被执行过
- `Do` 方法会首先检查标志位，如果函数已经被执行过（标志位不为0），则直接返回。否则，它会获取互斥锁，再次检查标志位，如果仍未被执行，则执行函数 `f`，并将标志位设置为1，以确保下次调用时不会再次执行 `f

```
type Once struct {
        done uint32
        m    Mutex
}
```

- **使用 `sync.Mutex` 实现**
- **使用 `sync/atomic` 实现**

33、new和make的区别

- make 只能用来分配及初始化类型为 slice、map、chan 的数据；new 可以分配任意类型的数据。
- new 分配返回的是指针，即类型 * Type；make 返回引用，即 Type。

- new 只分配内存，并将内存初始化为该类型的零值；make 分配空间后，会进行初始化，对于切片，它会分配底层的数组；对于map，它会创建一个空的哈希表；对于channel，它会创建一个新的channel。

34、var和:=来定义变量有什么不同？new和make初始化变量有什么不同呢?

:=根据右侧推断变量的类型，但只能在函数内部用，局部变量
`var`用于声明变量，并可以指定变量的类型，没有初值会自动初始化为该类型的零值
`var`声明的变量在编译时分配内存

- `new`用于分配内存，并返回指向该内存的指针。它适用于所有数据类型，对于给定的类型T，`new(T)`会分配零值内存的T类型，并返回其地址，即`*T`类型。（除了未初始化的channel、map和slice，因为new只分配内存而不进行初始化，而这些类型在使用前需要初始化）
- `make`用于初始化slice、map和channel。它不仅分配内存，还返回初始化（非零）的内存区域。对于slice，可以指定长度和容量；对于map和channel，可以指定（可选的）初始容量（对于切片，它会分配底层的数组；对于map，它会创建一个空的哈希表；对于channel，它会创建一个新的channel）
  总结:
  `var`用于声明变量，并根据类型自动初始化为零值，而`make`用于创建切片、map和通道，并返回已分配内存的实例
  对于切片、map和channel，不能使用`var`来初始化它们，而必须使用`make`
  var声明的变量可能是在编译时分配的（比如普通变量），而`make`总是在运行时分配内存（比如动态数据结构）
  对于channel来说，`var`声明的channel变量初始值为`nil`，而`make`创建的channel是已分配内存并可以使用的。尝试在`nil` channel上发送或接收数据会导致panic，而使用`make`创建的channel则可以安全地进行通信
  var主要用于变量的声明和初始化，而new主要用于为指针类型分配内存并返回其地址

35、如何实现一个set

使用 `map` 来实现，将map的key作为set的元素，value设置为一个空结构体`struct{}`
代码见最后

36、golang中是怎么设计哈希表的

1. **哈希函数**: Go 的 `map` 使用哈希函数将键映射到桶中。哈希函数的设计旨在减少冲突并提高分布均匀性。
2. **桶结构**: `map` 使用桶（buckets）存储键值对。每个桶内部是一个链表或其他数据结构，用于处理哈希冲突。

3. **动态扩容**: 当 `map` 的负载因子超过阈值时，会触发动态扩容。Go 的 `map` 通过重新哈希并将键值对分配到新的、更大的桶数组中来实现扩容。
4. **读写分离**: Go 的 `map` 实现通过优化读写操作的并发性能。对于频繁读操作的 `map`，读操作通常是高效的，因为 Go 对 `map` 的读操作是线程安全的，但写操作需要锁定。
5. **垃圾回收**: Go 的垃圾回收机制负责清理不再使用的 `map` 元素，以管理内存使用。

内存安全&锁

1、内存逃逸

指变量的存储位置：是存储在栈上（局部存储，函数调用结束后自动释放），还是逃逸到堆上（全局存储，需要手动管理内存）
是指**程序中的某些变量或数据的生命周期超出了其原始作用域，当变量逃逸到函数外部或持续存在于堆上时，导致内存分配的开销。**Go 编译器会进行逃逸分析，以确定哪些变量需要在堆上分配内存。（主要原因包括变量的生命周期超出其作用域、大对象的分配、闭包引用、接口动态分配以及切片和map操作等）
内存逃逸发生在以下几种情况：

1. **闭包（Closure）**：当一个函数返回一个闭包，而闭包内引用了函数内的变量时，这些变量必须逃逸到堆上，因为返回的闭包可能会在函数调用结束后继续使用这些变量。
2. **指针传递**：如果一个函数返回一个指针，或者将指针作为参数传递给其他函数，这可能导致原本在栈上的变量逃逸到堆上，以保证其生命周期能够延续到函数外部。
3. **数据结构的引用**：当你将一个大数据结构（例如切片、映射、通道等）传递给函数时，Go编译器可能会决定将其分配在堆上，以便在函数调用完成后仍然可以访问它们。
   检测内存逃逸： `go build -gcflags=-m`
   减少内存逃逸：
4. **尽量减少闭包的使用**：在可能的情况下，避免使用闭包或尽量减少闭包内部的变量引用，以减少变量逃逸的情况。
5. **传值而非传指针**：如果传递给函数的变量很小，可以考虑直接传递值而不是指针，这样可以避免堆分配。
6. **优化数据结构的使用**：如果一个数据结构非常大且频繁传递，考虑重新设计数据结构或使用不同的设计模式来减少内存逃逸。
7. **使用内存分析工具**：利用Go的内存分析工具（如`pprof`）来分析内存使用情况，找出内存逃逸的瓶颈，并优化相关代码。
   https://blog.csdn.net/weixin_45925028/article/details/134175541

2、Go语言内存泄漏的排查思路

1.使用pprof工具进行性能分析：通过go tool pprof命令，可以生成内存使用的报告，从而发现内存泄漏的线索。
2.使用runtime包进行调试：定期打印内存使用情况，判断是否存在内存泄漏。
3.使用第三方库进行内存泄漏检测：`go-memstats` 包、监控工具（如Prometheus与Grafana）
4.代码审查和静态分析：注意检查全局变量、缓存、长生命周期的对象等。
5.测试

3、聊聊goroutine泄漏？怎么避免和排查goroutine泄漏的问题

当大量goroutine被创建却不会释放时(即发生了goroutine泄露)，会消耗大量内存，造成内存泄露。
goroutine里有在堆上申请空间的操作，则这部分堆内存也不能被垃圾回收器回收（new关键字或者使用&对现有类型的变量取址来在堆上申请空间）：

1. **忘记关闭通道（channel）**：当goroutine持续监听一个未关闭的通道时，它将永久运行。
2. **无限循环**：在goroutine中，如果存在无条件的无限循环，该goroutine将永远不会退出。
3. **依赖外部条件**：goroutine可能依赖于某个外部条件来决定何时退出，如果这个条件永远不满足，goroutine也会泄漏。
   避免：
4. **使用**`sync.WaitGroup`：确保在所有goroutine完成后，主程序能正确等待它们结束。
5. **避免无限循环**：在循环中添加退出条件，或者使用`context.Context`来取消goroutine。
6. **关闭通道**：当不再需要goroutine时，及时关闭通道以通知goroutine退出。
   排查：
   使用pprof检测泄漏
   使用`debug`包避免泄漏，检查和清理goroutine

4、golang内存管理，为什么要这么设计

Go语言的内存管理不仅包括垃圾回收机制，还涉及到内存分配、逃逸分析、内存监控等多个方面，几个关键特性：

1. 垃圾回收机制
   垃圾回收机制方面，Go语言的gc是自动的，采用并发标记清除算法，具体实现是：三色标记法 + 混合写屏障，分为**标记阶段**和**清除阶段**，使用白色、灰色和黑色三种颜色来标记对象，通过扫描根对象集合，逐步将所有可达的对象标记为灰色和黑色，最终回收所有白色对象，即垃圾对象 。

2. 内存分配器
   Golang的内存分配器借鉴了TCMalloc的设计思想，其核心思想是使用多级缓存并将对象根据大小分类，按照类别实施不同的分配策略。主要组件包括：

- **mspan**：堆上内存管理的基本单元，由一连串的页构成（每页8KB），根据内存大小将mspan分成了67个等级。
- **mcache**：每个线程有一个独立的堆内存缓存，从mcache上申请内存不需要加锁。
- **mcentral**：所有线程共享的缓存，当mcache内存不足时，会从mcentral中申请内存。
- **mheap**：全局唯一的堆内存区域，当mcentral上mspan不足时或对象大于32KB时，会从mheap上申请内存。
  内存分配时，分配器会根据对象的大小和类型从合适的mspan中分配内存。

3. 逃逸分析
   在Golang中，一个变量分配在堆上还是栈上，主要依靠逃逸分析。逃逸分析是编译器在编译时分析代码，决定哪些变量的分配应该在栈上，哪些应该在堆上。栈内存由编译器管理，分配和释放速度快，但空间有限；堆内存由程序管理，空间较大，但分配和释放相对较慢。
4. 内存监控
   Golang提供了pprof工具来进行内存使用的监控，pprof会动态抓取程序的内存使用情况，并生成对应的报告。在pprof报告中，可以查看程序内部的内存分配情况、内存使用模式、内存泄露等细节

提升性能、减少内存碎片、简化开发者工作、支持高并发

5、golang内存管理和操作系统内存管理之间的关系

操作系统负责底层的物理内存和虚拟内存管理，而Go的运行时系统（runtime）则在操作系统提供的虚拟内存空间中进行内存管理，包括内存分配、回收和垃圾回收。
Go语言的内存管理机制建立在操作系统的内存管理基础上：
Go的内存分配器采用了与tcmalloc库相同的实现，是一个带内存池的分配器，底层直接调用操作系统的内存管理函数
Go为每个系统线程分配一个本地的MCache，并定期做垃圾回收，与操作系统内存管理协同工作。
Go的内存管理系统在多线程下的稳定性和效率问题，与操作系统相似

6、go的栈溢出到堆空间的情况有哪些

1. **递归调用**: 当函数递归深度过大时，栈空间可能不够用，Go 的运行时会将部分数据从栈移到堆上，尤其是当递归调用的深度动态增长时。
2. **大局部变量**: 如果函数中有大对象或大数组，超出了栈的限制，Go 会将这些大对象移到堆上。

3. **闭包**: 当一个函数闭包捕获了外部函数的局部变量时，这些变量可能被分配在堆上，尤其是在闭包的生命周期超出当前函数调用时。
4. **长时间运行的 goroutine**: 对于长期运行的 goroutine 或者 goroutine 中使用大量栈空间的场景，栈空间的扩展可能导致部分栈数据被移动到堆上。
   栈溢出发生在递归或过多的函数调用导致栈空间耗尽时，而内存泄漏是指程序无法释放已分配的内存。栈溢出也与内存逃逸不同，内存逃逸是指原本应该在栈上分配的变量被转移到堆上，从而增加了内存占用。

7、go的panic是什么？怎么捕获处理panic?

运行时错误机制，当遇到无法恢复的错误情况时，程序会触发panic，导致程序崩溃 输出错误信息和调用栈
要捕获并处理panic，可以使用`defer`关键字和`recover`函数。`defer`注册一个延迟调用的函数，该函数会在包含它的函数执行完毕后执行。`recover`函数则用于在defer注册的函数中恢复panic，阻止程序崩溃，并返回panic的值。

8、一个协程中发生了panic，对其他协程有什么影响?其他协程能够捕获panic吗?

其他协程不会受到直接影响，也不能捕获这个 `panic`

9、a和b两个线程，a里面有defer recover，a里面新开了一个b，b没写defer recover，b发生了panic，ab两个线程会发生

- 如果 `a` 的 `defer` 和 `recover` 成功捕获了 `b` 的 `panic`，则 `a` 继续执行，而 `b` 会终止。
- 如果 `recover` 没有处理到 `b` 的 `panic`（例如，`panic` 没有传递到 `a`，或 `recover` 不在正确的作用域），则 `a` 也会因未处理的 `panic` 终止。

10、defer是做什么用的，如果一个程序里面有多个defer会怎么样?

注册一个延迟调用的函数，函数执行结束时延迟执行代码块
逆序执行

11、recover和panic的实现原理

- 程序执行过程中遇到无法继续运行的错误时，如数组越界访问或空指针引用，Go 运行时（runtime）会触发 `panic`。
- `recover` 是一个内置函数，用于捕获并处理 `panic`
  实现原理：
- 编译过程将panic和recover转换成gopanic和gorecover函数，defer转成deferproc（，在defer末尾追加deferreturn指令）
- 运行中遇到gopanic，从当前goroutine取出defer链表并调用reflectcall

- 如果调用过程遇到gorecover函数，直接将当前panic.recovered标记成true并返回panic参数，恢复正常流程
  （gopanic从defer结构体中取出程序计数器pc和栈指针sp并调用recovery恢复，根据pc和sp跳转deferproc函数，返回值不为0跳转deferreturn并恢复正常流程）
- 没有遇到会依次遍历所有defer结构，最后调用fatalpanic中止，打印panic并返回错误码

12、哪些情况下会panic？所有的panic都可以recover吗？recover函数的使用过程，是怎么捕获的

- 运行时错误、调用panic函数、无效类型转换、递归调用导致栈溢出、并发竞争条件
- 不是，一旦 `panic` 到达了主函数（`main` 函数）的顶层，或者在一个没有defer函数的函数中发生，或者在一个已经被defer函数捕获并重新panic的函数中发生，程序就会崩溃并输出panic的信息。
- defer中调用recover

13、Go锁的类型

`sync.Mutex`、 `sync.RWMutex`、 `sync.WaitGroup`、 `sync.Once`、 `sync.Cond`
`sync.Cond` 是条件变量，用于在某些条件发生时通知其他 goroutine。

14、快速定位死锁

1. **使用 `go run -race`**：Go 的竞态检测器，在运行程序时使用 `-race` 标志，可以检测到潜在的死锁和其他竞态条件。
2. **启用 Go 的调试信息**：
   - 在程序中，使用 `runtime` 包中的 `runtime.Stack` 来打印堆栈信息，有助于识别阻塞的 goroutine。
   - 在调试时，可以通过 `runtime/pprof` 包生成运行时剖析文件，分析锁的持有和等待情况。
3. **查看堆栈跟踪**：使用 `runtime/pprof` 包中
   的 `pprof.Lookup("goroutine").WriteTo` 方法生成当前 goroutine 的堆栈跟踪，显示哪些 goroutine 在等待锁
4. **使用日志记录**：添加日志记录，标记每次锁的获取和释放，这样可以追踪锁的使用情况。
5. **分析死锁图**

15、sync的读写锁，先加读锁，能不能加上写锁，能不能再加上读锁

先加读锁后，不能直接加写锁，但可以再加读锁

- **它不能直接加上写锁。**原因是读写锁的设计允许多个读操作并发，但写操作是互斥的。如果允许一个已经持有读锁的线程再加上写锁，就会破坏这种互斥性，导致数据不一致的问题
- **它可以再加上读锁**，这被称为锁重入。锁重入是指线程在已经持有锁的情况下再次请求该锁，这是允许的，以避免死锁情况的发生。在读锁的情况下，重入计数会记录在ThreadLocalHoldCounter线程本地变量中
  sync.RWMutex
- **读锁与写锁的互斥**：读锁与写锁是互斥的，读锁可以并发持有，但在持有读锁时无法获取写锁，持有写锁时无法获取读锁。
- **写锁的独占性**：写锁会阻止其他 goroutine 获取读锁或写锁，确保写操作的独占性。
- **读锁的共享性**：读锁允许多个 goroutine 同时持有，但不能与写锁共存。

16、sync.mutex的底层实现

```
type Mutex struct {
        state int32
        sema  uint32
}
```

state用来表示当前互斥锁处于的状态，sema用于控制锁状态的信号量

- `state`：一个`int32`类型的字段，用于表示锁的状态和等待锁的goroutine数量。通过位操作，它可以记录锁是否被持有、是否有goroutine被唤醒、锁是否处于饥饿状态等信息。
- `sema`：一个信号量，用于阻塞和唤醒goroutine。当goroutine尝试获取锁而锁已被其他goroutine持有时，当前goroutine会被阻塞在信号量上，直到锁被释放

17、`sync.Mutex`的实现中包括两种模式

`sync.Mutex`的实现中包括两种模式：正常模式和饥饿模式。

- **正常模式**：在正常模式下，所有等待锁的goroutine按照先进先出（FIFO）的顺序等待。当锁被释放时，新到达的goroutine和等待队列中的goroutine会竞争锁的拥有权。通常，新到达的goroutine更容易获取锁，因为它正在CPU上执行。竞争失败的goroutine会被放到等待队列的前面，以便下次竞争。
- **饥饿模式**：当某个goroutine等待锁的时间超过一定时间（默认为1毫秒）后，`sync.Mutex`会切换到饥饿模式。在饥饿模式下，新到达的goroutine不会尝试获取锁，而是直接加入等待队列的尾部。锁的所有权会从释放锁的goroutine直接交给等待队列中的第一个goroutine，从而避免饥饿现象。

18、atomic里cas方法，它里面的实现有没有加锁，原子操作和go的锁有什么区别

在 Go 的 `sync/atomic` 包中实现原子操作，用于无锁地比较并交换值。
CAS 方法本身不涉及加锁，而是通过硬件支持的原子指令实现，这些原子指令确保了操作的原子性而无需额外的锁定机制

- **原子操作**：提供基本的原子性操作，通常用于简单的同步场景，如计数器的增加或标志的设置。它们对性能影响较小，但只适用于简单的同步需求。
- **Go 锁（如 `sync.Mutex`）**：提供更复杂的同步机制，支持多线程之间的互斥访问。虽然性能开销较高，但能够处理复杂的并发场景，如读写锁、条件变量等。

19、用户从客户端访问一个页面，webserver如何主动的给这个页面推送一个通知

涉及到实时通信技术
websocket：在单个长连接上进行全双工通讯的协议。客户端和服务器之间一旦建立WebSocket连接，两者就可以随时发送数据。

20、实现一个web的server，如何设置这个server返回的response的类型，比如说是一个图片一个视频一个json

HTTP响应头中的`Content-Type`字段：

- `text/html`：HTML格式
- `application/json`：JSON格式
- `image/jpeg`：JPEG图片格式
- `image/png`：PNG图片格式
- `video/mp4`：MP4视频格式

21、select语句的用途

用于同时处理多个通道（channel）的发送与接收操作

22、select中如果没有default会出现什么情况？case中的通道被关闭了会出现什么情况?

select会阻塞，直到有case可以执行
在接收操作中，如果通道关闭，`select` 会正常处理并能检测到通道的关闭；在发送操作中，向关闭的通道发送数据会导致 `panic`

23、waitgroup的原理

WaitGroup用于等待一组协程的完成：
1.定义一个sync.WaitGroup变量，`WaitGroup` 内部维护一个计数器
2.在每个要等待的协程开始执行之前，调用WaitGroup的Add方法，参数为1，表示要等待一个协程。
3.在每个协程完成时，调用WaitGroup的Done方法，表示该协程已完成。
4.在主协程中，调用WaitGroup的Wait方法，这将阻塞主协程，直到所有添加的协程都调用了Done方法。

24、Go语言中context常用场景，及实现细节

context包是用来管理Goroutine的生命周期、传递上下文信息和控制取消操作的机制。`context` 包的常用场景包括：

1. **请求取消**：在超时或请求取消时终止 goroutine。使用 `context.WithCancel` 创建一个可以取消的上下文，将取消函数传递给 goroutine，以便在请求取消时中止处理。
2. **超时处理**：设置操作超时以避免阻塞。使用 context.WithTimeout 创建一个具有超时的上下文，在上下文超时后，操作将自动取消。
3. **数据传递**：在请求范围内传递数据。使用 context.WithValue 将数据存储在上下文中，在处理过程中通过 ctx.Value 检索数据。
4. **并发任务管理**：确保所有并发任务在需要时能够被正确取消。使用 `context.WithCancel` 创建上下文，并传递给所有任务，在需要取消时调用 `cancel` 函数。
   每个上下文对象都有一个 `Done` 方法，当上下文被取消时，所有监听这个通道的 goroutine 都会收到通知。

25、是否了解反射

Go 的反射机制允许程序在运行时检查和修改类型和值

- `reflect.Type`：表示一个 Go 类型。可以用来获取有关类型的信息，比如类型名称、字段信息等。
- `reflect.Value`：表示一个值，可以用来获取和修改值的具体内容。
  获取类型和值：通过 `reflect.TypeOf` 和 `reflect.ValueOf` 函数，可以分别获取对象的类型和对应的值
  修改值：CanSet()、SetInt()
  使用反射操作结构体：Field()
  反射和接口

其他

1、分布式事务

2、gorm常见hook

1.BeforeSave：在保存模型前调用。无论是创建新记录还是更新现有记录，BeforeSave 都会被触发。**用于数据验证或预处理。**
2.BeforeCreate：在创建模型前调用。当调用 Create 方法时，BeforeCreate 会在模型被插入数据库之前执行。**用于设置自动增长的字段（如 UUID）、检查数据的完整性等。**
3.AfterCreate：在创建模型后调用。模型被成功插入数据库后，会触发 AfterCreate。用于记录日志、更新关联数据等。
4.BeforeUpdate：在更新模型前调用。当调用 Update 或 Save方法时，BeforeUpdate 会在模型被更新前执行。**用于检查更新的数据是否合法。**
5.AfterUpdate：在更新模型后调用。模型被成功更新后，会触发 AfterUpdate。类似于 AfterCreate，**用于记录日志或更新关联数据。**
6.BeforeDelete：在删除模型前调用。当调用 Delete 方法时，BeforeDelete 会在模型从数据库中删除前执行。用于检查删除操作是否安全，或者执行一些清理工作。
7.AfterDelete：在删除模型后调用。模型被成功删除后，会触发 AfterDelete。用于记录日志或执行一些后续操作。
8.AfterFind：在查询记录后调用。当使用 `Find` 方法加载数据时，AfterFind 会在数据被加载到模型实例后执行。用于对查询结果进行额外的处理验证。

3、用过gorm，如果一张上百万的数据的表，要新建一个字段的索引，如何保证线上的服务尽量少的被影响

1. **在低峰时段操作**
2. **使用在线DDL**：如果数据库支持在线DDL（Data Definition Language）操作，可以使用此功能来创建索引，可以在不阻塞读写操作的情况下进行。
3. **分批处理**

4、gin怎么实现记录所有的响应日志

使用Gin自带的Logger中间件

5、Gin框架相比标准包，解决了哪些问题？

路由匹配能力：标准包只支持精确匹配，而Gin提供了更灵活的路由匹配方式，包括通配符、路径参数等，满足复杂路由需求。
请求处理流程：标准包需要手动处理请求和响应的多个环节，如读取数据、反序列化、设置响应头等，而Gin通过中间件机制简化了这些流程，提高了开发效率。
性能优化：Gin使用高效的Radix树作为路由算法基础，优化了中间件处理机制，相比标准包在处理大量请求时具有更高的性能。
功能扩展性：Gin支持丰富的中间件和插件，可以轻松扩展自定义功能，如日志记录、权限控制等，而标准包的功能相对固定，扩展性较差。

错误处理：Gin提供了更为完善的错误处理机制，能够捕获并恢复panic，保证服务的稳定性，而标准包在错误处理方面较为简单。

6、Go程序启动时发生什么

1. **编译和链接**：源代码首先被编译成中间对象文件，然后链接成一个可执行文件，包括链接运行时库，如垃圾回收器、goroutine调度器等。
2. **运行时初始化**：执行程序时，Go运行时首先初始化，包括内存分配、goroutine调度器初始化、垃圾回收器启动等。
3. **执行main函数**：完成初始化后，运行时调用main函数，开始执行程序逻辑。
4. **并发调度和系统调用处理**：如果程序中使用了goroutines，运行时将在必要时创建额外的系统线程，并在这些线程间调度goroutines的执行。程序执行过程中的系统调用由运行时处理。

7、面向对象三大概念

封装、继承和多态。

1. **封装**：将对象的状态和行为封装在一起，隐藏对象的内部实现细节，只通过对象的方法来控制和操作数据。这有助于保护对象的内部状态，提高代码的安全性和可维护性。
2. **继承**：允许一个类（子类）继承另一个类（父类）的属性和方法，通过重用父类的代码来减少代码的重复性，并实现代码的扩展和维护。
3. **多态**：允许使用同一操作符或方法调用来操作不同的数据类型，提高代码的灵活性。多态包括编译时多态（方法重载）和运行时多态（方法重写）。

8、go和java的多态区别

Java通过继承实现多态，而Go通过接口实现多态的效果

9、从面向对象的角度聊聊java和golang的区别

Java面向对象主要封装继承多态
面向对象中的 “封装” 指的是可以隐藏对象的内部属性和实现细节，仅对外提供公开接口调用，go里封装是指属性访问权限，通过首字母大小写来控制；
面向对象中的 “继承” 指的是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，go继承通过结构体（struct）和方法（method）来模拟面向对象的特性；
面向对象中的 “多态” 指的同一个行为具有多种不同表现形式或形态的能力，go多态是通过接口来实现的

10、go和c++指针的大小

在Go和C++中是相同的，因为指针的本质是用来存储变量内存地址的变量，其大小取决于系统体系结构而非所指向的数据类型
指针的大小与平台的体系结构密切相关。在32位系统中，指针的大小通常为4字节（32位）；而在64位系统中，指针的大小通常为8字节（64位）

11、相比于C语言，Go语言开发有哪些改进？

内存管理：Go语言引入了垃圾回收机制，自动管理内存，解决了C语言中手动管理内存带来的复杂性和错误风险。
语法简洁性：Go语言的语法更加简洁现代，减少了代码的复杂性，提高了开发效率。
并发编程：Go语言内置了强大的并发编程支持，通过goroutine和channel可以轻松实现高效的并发程序，而C语言则需要手动管理线程和同步。
包体系：Go语言引入了包体系，方便代码结构的控制和第三方资源的引用，而C语言没有包的概念。
开发效率：Go语言提供了丰富的标准库和工具，使得开发者能够更快地构建应用程序，减少了从零开始编写代码的需要。

12、在函数参数传递一个非指针的互斥锁会发生什么事情？为什么会发生？

1.**互斥锁副本的产生**
2.**互斥效果不共享**
3.**死锁和竞争条件的风险**
原因：

- **Go语言的值传递特性**：Go语言中的函数参数默认是按值传递的，这意味着当你传递一个变量给函数时，实际上传递的是这个变量的一个副本。互斥锁（如`sync.Mutex`）作为结构体类型，在Go中也是按值传递的。
- **互斥锁的状态不可共享**：互斥锁的状态（如是否已锁定）是封装在互斥锁变量内部的。由于按值传递，函数接收到的只是互斥锁状态的一个副本，而不是原始状态本身。

13、gin框架特点

路由支持restful风格
中间件支持Logger日志记录、身份验证、请求解析
支持多种数据格式，比如json、yaml
支持参数绑定和校验
Viper获取配置
基于`jwt`和`casbin`实现的权限管理

14、其他go框架

Beego还采用了流行的MVC架构，有利于应用程序的高效组织和管理，支持RESTful服务，内置了ORM模块，提高了开发效率和安全性
Echo极简的API设计，反射和接口的动态调用等技术，提供了更高的性能。

- Gin和Echo是轻量级框架，适用于构建小型、高性能的应用程序；Beego和Iris是更全面的框架，适用于构建中大型应用，但对于性能要求较高的场景，可以优先选择Iris；Revel是一个全栈式的框架，适合快速开发，但需要注意其对项目结构的侵入式要求。

如何实现一个set

```
package main

import "fmt"

// Set 结构体
type Set map[string]struct{}

// NewSet 创建一个新的 Set 实例
func NewSet() Set {
    return make(Set)
}

// Add 向 Set 中添加元素
func (s Set) Add(element string) {
    s[element] = struct{}{}
}

// Remove 从 Set 中移除元素
func (s Set) Remove(element string) {
    delete(s, element)
}

// Contains 检查 Set 是否包含元素
func (s Set) Contains(element string) bool {
    _, exists := s[element]
    return exists
}

// Size 获取 Set 的大小
func (s Set) Size() int {
```

```
    return len(s)
}

func main() {
    s := NewSet()
    s.Add("apple")
    s.Add("banana")

    fmt.Println(s.Contains("apple"))  // 输出: true
    fmt.Println(s.Contains("orange")) // 输出: false

    s.Remove("apple")
    fmt.Println(s.Contains("apple"))  // 输出: false

    fmt.Println("Size:", s.Size()) // 输出: Size: 1
}
```