# Go面试题

## 1、go语言常见面试点

1. 垃圾回收（Garbage Collection）：Go语言自带垃圾回收机制，用于自动管理内存。考察垃圾回收机制的实现原理、优缺点以及如何手动触发垃圾回收等。
2. 并发和并行（Concurrency and Parallelism）：Go语言天生支持并发和并行，考察协程、通道、互斥锁等并发编程的概念和实现。
3. 通道（Channel）：Go语言中的通道是协程之间通信的主要手段，考察通道的创建、关闭、阻塞和非阻塞等。
4. 接口（Interface）：Go语言中的接口是一种抽象类型，用于实现多态和面向对象编程，考察接口的定义、实现、嵌入和类型转换等。
5. 反射（Reflection）：Go语言支持反射，可以在运行时动态获取类型信息和调用对象的方法，考察反射的基本概念、使用方法和性能问题等。
6. 错误处理（Error Handling）：Go语言中的错误处理机制是一种很重要的语言特性，考察错误类型、错误处理函数、错误传递和 panic/recover 等。
7. 包管理（Package Management）：Go语言中的包管理机制非常重要，考察包的导入、命名、初始化和依赖管理等。
8. 标准库（Standard Library）：Go语言的标准库是非常丰富的，包含了大量常用的工具和库，考察标准库中常用的包和函数等。
9. 性能优化（Performance Optimization）：Go语言的性能优势是其一大优点，考察如何进行性能分析和优化，如何利用 Go 语言提供的性能工具等。
10. 数据结构和算法（Data Structures and Algorithms）：Go语言中的数据结构和算法也是面试的重要考察内容，如链表、树、堆、排序和查找等。

## 2、介绍一下垃圾回收算法

Go语言自带垃圾回收机制（Garbage Collection，简称GC），用于自动管理内存。Go语言的垃圾回收机制采用标记-清除算法（Mark and Sweep Algorithm），具有以下特点：

1. 零成本抽象：Go语言的垃圾回收机制对程序员来说是透明的，不需要手动申请或释放内存。
2. 并发标记：Go语言的垃圾回收机制采用并发标记算法，可以与程序并发执行，减少暂停时间。
3. 分代回收：Go语言的垃圾回收机制采用分代回收策略，将对象分为新生代和老年代，分别采用不同的回收策略。
4. 增量回收：Go语言的垃圾回收机制采用增量回收算法，在程序执行过程中逐步回收垃圾，避免长时间暂停。

Go语言的垃圾回收机制会自动监测程序的内存使用情况，并在必要时启动垃圾回收过程。具体来说，当程序中的内存使用量达到一定阈值时，垃圾回收机制就会启动，并扫描程序中的所有对象，标记出所有活跃的对象，然后清除所有未标记的对象。

需要注意的是，Go语言的垃圾回收机制虽然是自动的，但也需要开发者关注其影响因素，如内存使用率、并发度、内存分配模式等，以尽可能减少垃圾回收的成本和影响。

在深入了解Go语言的垃圾回收机制之前，先介绍一下几个与垃圾回收有关的概念：

* 堆（Heap）：堆是一块被分配的动态内存空间，用于存储程序中动态创建的对象。
* 栈（Stack）：栈是一块被分配的静态内存空间，用于存储函数的调用栈和局部变量等。
* 标记（Mark）：在垃圾回收过程中，标记是指将所有活跃的对象打上标记，以便在清除阶段回收未被标记的垃圾对象。
* 清除（Sweep）：在垃圾回收过程中，清除是指回收所有未被标记的垃圾对象，以便重新使用这些内存空间。
* 内存分配器（Allocator）：Go语言中的内存分配器用于分配堆空间，并维护内存池等数据结构，以提高内存分配的效率。

Go语言的垃圾回收机制采用标记-清除算法（Mark and Sweep Algorithm），分为标记和清除两个阶段：

1. 标记阶段：在标记阶段，垃圾回收机制从根对象出发，遍历整个对象图，并标记所有与根对象直接或间接相连的对象，这些被标记的对象都是活跃的对象。标记阶段采用并发标记算法，可以与程序并发执行，避免长时间的暂停。
2. 清除阶段：在清除阶段，垃圾回收机制扫描整个堆空间，将未被标记的对象全部清除，并将这些空闲的内存空间加入到内存池中，以便下次内存分配时使用。清除阶段采用增量回收算法，在程序执行过程中逐步回收垃圾，避免长时间暂停。

## 3、介绍一下go的并发和并行

Go语言天生支持并发和并行，是其最重要的特性之一。并发（Concurrency）和并行（Parallelism）虽然类似，但有本质的区别：

* 并发：指多个任务在同一时间段内执行，由于CPU时间片的划分，看起来像是同时执行，但实际上是交替执行的。并发是指多个任务之间有互相制约、互相影响的关系，需要协调、同步和共享资源。
* 并行：指多个任务在同一时刻执行，由于有多个CPU核心同时工作，实际上是真正的同时执行。并行是指多个任务之间没有互相制约、互相影响的关系，可以独立执行，不需要协调、同步和共享资源。

在Go语言中，可以通过协程（Goroutine）和通道（Channel）实现并发和并行，其中：

* 协程：是一种轻量级线程，可以同时执行多个任务，占用的资源很少，创建和销毁的代价很小，通常可以创建成千上万个协程。
* 通道：是一种用于协程之间通信的机制，可以实现数据的传递和同步。通道可以避免协程之间的竞争和死锁等问题。

通过协程和通道的组合，Go语言可以实现各种并发和并行的模式，如并发执行、数据流水线、并行计算等，可以提高程序的执行效率和质量。

需要注意的是，并发和并行的实现都需要考虑到并发安全和性能问题，如协程之间的同步和互斥、通道的缓冲区和阻塞机制、并行计算的任务分配和负载均衡等。在编写并发和并行程序时，需要对这些问题有深入的理解和处理方案。

## 4、介绍下go语言的调度模型

Go语言的调度器是其并发和并行的核心，负责将协程分配到可用的CPU核心上执行，并提供一些调度优化策略，以提高程序的性能和可靠性。

Go语言的调度器采用M:N调度模型，其中M表示操作系统线程（OS Thread），N表示协程（Goroutine）。在M:N调度模型中，Go语言的调度器将协程调度到操作系统线程上执行，以充分利用多核CPU的资源。

具体来说，Go语言的调度器包括三个主要的组件：

1. 调度器（Scheduler）：调度器负责管理和调度协程和操作系统线程的分配和回收，同时负责协程的阻塞和唤醒操作。
2. P（Processor）：P表示调度器的处理器，负责将协程调度到操作系统线程上执行，并提供一些调度优化策略，如工作窃取和自适应调度等。
3. G（Goroutine）：G表示协程，是Go语言并发编程的基本单元，由调度器分配到P上执行。协程可以通过通道和互斥锁等机制进行同步和通信。

Go语言的调度器采用抢占式调度模型，在调度器的帮助下，协程可以随时被中断和重新调度，从而实现任务的快速响应和负载均衡。调度器还提供一些优化策略，如工作窃取和自适应调度等，以优化协程的分配和调度。

需要注意的是，Go语言的调度器和操作系统的线程调度器是两个不同的层次，操作系统线程只是调度器的一种实现方式，而调度器可以根据实际情况调整协程的分配和调度策略。因此，在编写Go程序时，需要充分理解Go语言的调度模型和调度优化策略，以充分利用其并发和并行的优势。

## 5、那GMP呢

GMP是Go语言中的一个重要概念，代表着Go语言的调度模型。GMP是Goroutine、OS Thread、M和P的缩写，它们分别代表了Go语言并发调度中的四个核心概念。

* Goroutine：Goroutine是Go语言的并发执行单元，类似于线程，但是占用的资源更少，创建和销毁的代价也更小。一个Go程序通常包含数千个Goroutine，它们通过通道进行通信和同步。
* OS Thread：OS Thread是操作系统的线程，它是Go语言调度器的执行单元，与Goroutine一一对应。调度器将Goroutine调度到OS Thread上执行，并将它们分配到不同的OS Thread上，以充分利用多核CPU的资源。
* M：M是调度器的执行上下文，它包含了调度器需要的一些信息，如当前线程的状态、堆栈信息、Goroutine队列等。每个OS Thread都会绑定一个M，调度器会根据需要创建和销毁M。
* P：P是处理器的概念，它是调度器的核心组件，负责将Goroutine分配到不同的OS Thread上执行，并提供一些调度优化策略，如工作窃取和自适应调度等。一个P可以执行多个Goroutine，但每个Goroutine只能被一个P执行。

Go语言的调度器采用M:N调度模型，即多个Goroutine被调度到多个OS Thread上执行。在调度器的帮助下，Goroutine可以随时被中断和重新调度，从而实现任务的快速响应和负载均衡。GMP模型是Go语言调度器的核心组成部分，也是Go语言并发编程的基础。

## 6、介绍下你的项目

1. 项目背景：简单介绍您的项目背景，包括项目的目的、使用场景和应用范围等。比如，这是一个基于分布式集群的HPC故障诊断项目，旨在通过实时采集数据和分析算法来帮助用户诊断HPC系统的性能和存储故障。
2. 技术架构：介绍项目的技术架构，包括前端和后端的技术框架、数据采集和存储方式、算法模型和计算资源等。比如，项目采用Go语言编写后端服务，前端使用Vue.js框架，数据采集和存储使用InfluxDB和OpenTSDB，算法模型包括时间序列分析和机器学习等，计算资源基于Kubernetes和Docker的容器技术。
3. 故障诊断：重点介绍项目的核心功能——故障诊断。可以从以下几个方面进行说明：

* 数据采集：介绍项目如何实时采集HPC应用运行时的相关指标数据，包括CPU使用率、内存占用、磁盘读写等。可以说明数据采集的方式、采集的频率和数据存储的方式等。
* 数据分析：介绍项目如何分析采集的数据，从而判断HPC系统是否出现性能劣化和存储故障。可以介绍分析算法的原理和实现方式，包括时间序列分析和机器学习等。可以用具体的数据和实例来说明算法的效果和精度。
* 结果展示：介绍项目如何将分析结果展示给用户，包括可视化的图表、表格和报告等。可以说明展示方式的灵活性和定制性，以满足不同用户的需求。

1. 项目成果：介绍项目的成果和应用效果，可以从以下几个方面进行说明：

* 故障定位：说明项目如何帮助用户快速定位故障，并提供相应的解决方案。可以介绍实际应用的效果和用户的反馈。
* 性能优化：说明项目如何帮助用户优化HPC应用的性能，提高作业的效率和可靠性。可以用实际应用的数据和实例来说明性能优化的效果和影响。

1. 改进计划：介绍项目的改进计划和未来发展方向，包括优化算法模型、提高数据采集和存储的效率、增强可扩展性和可靠性等。可以说明您对项目的思考

## 7、Go语言主要特征

1.自动立即回收。

2.更丰富的内置类型。

3.函数多返回值。

4.错误处理。

5.匿名函数和闭包。

6.类型和接口。

7.并发编程。

8.反射。

9.语言交互性

## 8、init函数有什么特征和main函数有什么区别？

go语言中init函数用于包(package)的初始化，该函数是go语言的一个重要特性。

有下面的特征：

1 init函数是用于程序执行前做包的初始化的函数，比如初始化包里的变量等

2 每个包可以拥有多个init函数

3 包的每个源文件也可以拥有多个init函数

4 同一个包中多个init函数的执行顺序go语言没有明确的定义(说明)

5 不同包的init函数按照包导入的依赖关系决定该初始化函数的执行顺序

6 init函数不能被其他函数调用，而是在main函数执行之前，自动被调用

Main函数：

Go语言程序的默认入口函数(主函数)：func **main**()

函数体用｛｝一对括号包裹。

func **main**(){

*//函数体*

}

异同：

相同点：

两个函数在定义时不能有任何的参数和返回值，且Go程序自动调用。

不同点：

init可以应用于任意包中，且可以重复定义多个。

main函数只能用于main包中，且只能定义一个。

两个函数的执行顺序：

对同一个go文件的init()调用顺序是从上到下的。

对同一个package中不同文件是按文件名字符串比较“从小到大”顺序调用各文件中的init()函数。

对于不同的package，如果不相互依赖的话，按照main包中”先import的后调用”的顺序调用其包中的init()，如果package存在依赖，则先调用最早被依赖的package中的init()，最后调用main函数。

如果init函数中使用了println()或者print()你会发现在执行过程中这两个不会按照你想象中的顺序执行。这两个函数官方只推荐在测试环境中使用，对于正式环境不要使用。

## 9、go中有哪些常用命令？

go env用于打印Go语言的环境信息。

go run命令可以编译并运行命令源码文件。

go get可以根据要求和实际情况从互联网上下载或更新指定的代码包及其依赖包，并对它们进行编译和安装。

go build命令用于编译我们指定的源码文件或代码包以及它们的依赖包。

go install用于编译并安装指定的代码包及它们的依赖包。

go clean命令会删除掉执行其它命令时产生的一些文件和目录。

go doc命令可以打印附于Go语言程序实体上的文档。我们可以通过把程序实体的标识符作为该命令的参数来达到查看其文档的目的。

go test命令用于对Go语言编写的程序进行测试。

go list命令的作用是列出指定的代码包的信息。

go fix会把指定代码包的所有Go语言源码文件中的旧版本代码修正为新版本的代码。

go vet是一个用于检查Go语言源码中静态错误的简单工具。

go tool pprof命令来交互式的访问概要文件的内容。

## 10、new和make有什么区别？

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。

## 11、 垃圾回收

**垃圾回收、三色标记原理**

垃圾回收就是对程序中不再使用的内存资源进行自动回收的操作。

##### 1.1 常见的垃圾回收算法：

* 引用计数：每个对象维护一个引用计数，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 +1；如果这个对象被销毁，则计数 -1 ，当计数为 0 时，回收该对象。
  + 优点：对象可以很快被回收，不会出现内存耗尽或到达阀值才回收。
  + 缺点：不能很好的处理循环引用
* 标记-清除：从根变量开始遍历所有引用的对象，引用的对象标记“被引用”，没有被标记的则进行回收。
  + 优点：解决了引用计数的缺点。
  + 缺点：需要 STW（stop the world），暂时停止程序运行。
* 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。
  + 优点：回收性能好
  + 缺点：算法复杂

##### 1.2 三色标记法

* 初始状态下所有对象都是白色的。
* 从根节点开始遍历所有对象，把遍历到的对象变成灰色对象
* 遍历灰色对象，将灰色对象引用的对象也变成灰色对象，然后将遍历过的灰色对象变成黑色对象。
* 循环步骤3，直到灰色对象全部变黑色。
* 通过写屏障(write-barrier)检测对象有变化，重复以上操作
* 收集所有白色对象（垃圾）。

##### 1.3 STW（Stop The World）

* 为了避免在 GC 的过程中，对象之间的引用关系发生新的变更，使得GC的结果发生错误（如GC过程中新增了一个引用，但是由于未扫描到该引用导致将被引用的对象清除了），停止所有正在运行的协程。
* STW对性能有一些影响，Golang目前已经可以做到1ms以下的STW。

##### 1.4 写屏障(Write Barrier)

* 为了避免GC的过程中新修改的引用关系到GC的结果发生错误，我们需要进行STW。但是STW会影响程序的性能，所以我们要通过写屏障技术尽可能地缩短STW的时间。

造成引用对象丢失的条件:

一个黑色的节点A新增了指向白色节点C的引用，并且白色节点C没有除了A之外的其他灰色节点的引用，或者存在但是在GC过程中被删除了。以上两个条件需要同时满足：满足条件1时说明节点A已扫描完毕，A指向C的引用无法再被扫描到；满足条件2时说明白色节点C无其他灰色节点的引用了，即扫描结束后会被忽略 。

写屏障破坏两个条件其一即可

* 破坏条件1：Dijistra写屏障

满足强三色不变性：黑色节点不允许引用白色节点 当黑色节点新增了白色节点的引用时，将对应的白色节点改为灰色

* 破坏条件2：Yuasa写屏障

满足弱三色不变性：黑色节点允许引用白色节点，但是该白色节点有其他灰色节点间接的引用（确保不会被遗漏） 当白色节点被删除了一个引用时，悲观地认为它一定会被一个黑色节点新增引用，所以将它置为灰色

## 12、GPM 调度 和 CSP 模型

### 12.1 CSP 模型？

CSP 模型是“以通信的方式来共享内存”，不同于传统的多线程通过共享内存来通信。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。

### 12.2 GPM 分别是什么、分别有多少数量？

* G（Goroutine）：即Go协程，每个go关键字都会创建一个协程。
* M（Machine）：工作线程，在Go中称为Machine，数量对应真实的CPU数（真正干活的对象）。
* P（Processor）：处理器（Go中定义的一个摡念，非CPU），包含运行Go代码的必要资源，用来调度 G 和 M 之间的关联关系，其数量可通过 GOMAXPROCS() 来设置，默认为核心数。

M必须拥有P才可以执行G中的代码，P含有一个包含多个G的队列，P可以调度G交由M执行。

### 12.3 Goroutine调度策略

* 队列轮转：P 会周期性的将G调度到M中执行，执行一段时间后，保存上下文，将G放到队列尾部，然后从队列中再取出一个G进行调度。除此之外，P还会周期性的查看全局队列是否有G等待调度到M中执行。
* 系统调用：当G0即将进入系统调用时，M0将释放P，进而某个空闲的M1获取P，继续执行P队列中剩下的G。M1的来源有可能是M的缓存池，也可能是新建的。
* 当G0系统调用结束后，如果有空闲的P，则获取一个P，继续执行G0。如果没有，则将G0放入全局队列，等待被其他的P调度。然后M0将进入缓存池睡眠。

## 13、channel什么时候会发生panic

* 关闭值为 nil 的 channel
* 关闭已经关闭的 channel
* 向已经关闭的 channel 中写数据

## 14、说说逃逸分析

「逃逸分析」就是程序运行时内存的分配位置(栈或堆)，是由编译器来确定的。堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。

逃逸场景：

1. 变量类型不确定
2. 暴露给外部指针
3. 变量所占内存较大
4. **变量大小不确定**

## 15、channel什么时候会死锁？

1、当一个channel中没有数据，而直接读取时，会发生死锁：

2、写入数据超过channel的容量，也会造成死锁：