一、基础部分

1. **GO 语言的特点：**
    1. 支持 Goroutine（独立的并非执行单元）和 Channel（协程间通信）；
    2. 内置垃圾回收，无需手动管理内存；
    3. 适合构建高性能，高并发的后端服务、应用、系统；
2. **String 和[]byte 的区别：**
    1. **不可变性：**string 是不可变的，字节数组[]byte 可变。扩展：字节数组的底层是**指针+长度**
    2. **类型转换：**string 与`[]byte` 之间的转换是复制（有内存损耗），可以用 map[string] []byte 建立字符串与[]byte 之间映射，也可 range 来避免内存分配来提高性能。#内存拷贝
    3. **内存分配：**string 是一个不可变的视图，内存分配和释放是由 Go 运行时管理；[]byte 是一个可变的切片，其内存管理由开发者负责；
    4. **Unicode 字符：**string 的每个元素是一个 Unicode 字符；[]byte 的每个元素是一个字节；因此 string 可以包含任意字符，而[]byte 主要用于处理字节数据。
3. **nil 切片和空切片的区别：** `var s1 []int` `s2 := make([]int,0)`
    1. **nil 切片和空切片指向的地址不一样。nil 空切片引用数组指针地址为 0（无指向任何实际地址）**
    2. **空切片的引用数组指针地址是有的，且固定为一个值**

    扩展，切片的数据结构：指针，长度和容量
4. **new()和 make()的区别：**
    1. `new` 的作用是初始化一个指向类型的指针 (*T)；
    2. `make` 的作用是为 `slice,map` 或者 `channel` 初始化`，并且返回引用 T
5. **切片和数组的区别：**（长度，类型，存储）
    1. **数组：**固定长度，值类型（值拷贝），顺序存储；
    2. **切片：**长度动态扩容，引用类型，切片底层是一个结构体，包含长度，容量和底层数组

6. **切片的扩容机制：v1.18 之后**

    1. 期望容量大于当前容量的两倍，使用期望容量
    2. 当前切片的长度小于阈值（默认 256），则将容量翻倍
    3. 当前切片的长度大于等于阈值（默认 256），则每次增加 1.25 倍（具体是：newcap+3*阈值的 25%），直到新容量大于期望容量

7. **拷贝大切片一定比小切片代价大吗？** 答：不会，因为切片底层就三个参数，无非值不一样；

8. **参数传递：**

    1. 对于基本类型（整数、浮点数、布尔值等）和结构体，传递的是值的副本；修改形参不会影响实参；
    2. 针对切片、映射、通道等引用类型，传递的是引用的副本，修改形参会影响实参；

9. **map 的底层实现原理：**go 的 map 实现是基于**散列表**。散列表是一种用于实现字典结构的数据结构，他通过一个哈希函数将键映射到存储桶，**每个存储桶存储一个链表或者红黑树，用于处理哈希冲突。**存储桶的数量是固定的，有 map 的大小和负载因子决定。

10. **map 是有序的还是无序的？**map 底层基于散列表实现的，散列表通过哈希函数将键映射到存储桶；散列表中的存储桶是无序的，不保证元素按照特定顺序存储；

11. **map 不初始化使用会怎么样？** 答：空指针 panic；

12. **map 不初始化长度和初始化长度的区别？** 答： 未初始化的 map，未分配内存且长度为空，添加值会空指针 panic；

13. **map 如何扩容：**

    1. **计算新的存储桶数量：**当 map 的元素数量**达到负载因子的上限时**，会触发扩容。新的存储桶数量通常会是**当前存储桶数量的两倍**；
    2. **分配新的存储桶和散列数组：**创建新的存储桶和散列数组，大小为新的存储桶数量。涉及内存分配；
    3. **重新散列元素：**遍历当前 map 的每个存储桶，将其中的元素重新散列到新的存储桶中。为了保持元素在新存储桶中的顺序；

4. **切换到新的存储桶和散列数组：**将 map 的内部数据结构指向新的存储桶和散列数组。 这个过程是原子的，确保在切换期间不会影响并发访问；

5. **释放旧的存储桶和散列数组：**释放存储空间，避免内存泄漏；

14. **如何按照特定顺序遍历 map：** 将 **map 的键**按**遍历顺序**存储到切片中，根据切片顺序遍历 map；

15. **map 是并发安全的吗，如何并发安全：标准的 map 类型并非是类型安全的，**意味着在多个 goroutine 中并发读写一个 map 可能导致数据竞争和不确定性行为；为了在并发情况使用 map，可以使用 **sync 包中的 sync.Map 类型，是并发安全的映射**；

16. **map 承载多大，大了怎么办？** 答：

    map 的承载能力主要取决于几个因素，包括系统的内存容量和 map 内部的实现机制。

    1. 优化键：使用更好的键，减少 hash 冲突；
    2. 分片：将大 map 分成小 map，分散负载；
    3. 内存释放：定期清理和优化数据；再或者加大内存；
    4. 针对大数据集，可以使用 redis 存储，不仅对数据结构和算法进行了优化，还能持久化存储；

    拓展：**redis 和 map 的区别？**

17. **翻转含有中文、数字、英文字母的字符串：**`[]rune(str)`, 将字符串类型转换成[]rune 再进行翻转。因为 rune 是 int32 的别名，能比 byte 表示更多字符。

18. **Go 错误处理和 Java 的异常处理对比：**

    1. Go 语言用返回值处理错误：函数通常有两个返回值，一个是正常的返回值，一个是 error 类型的值，用于表示可能出现的错误；需要显式的检查错误是否为空来判断该函数执行是否出错；
    2. Go 中的错误是普通的值，是实现了 error 接口的类型；
    3. Go 的错误机制处理在性能上更高效，因为没有引入额外的控制流程（异常栈的构建和查找）
    4. Java 使用异常机制处理错误，当出现错误的时候，通过 throw 关键字抛出异常，然后通过 catch 来捕获并处理异常；

5. Java 抛出的异常是对象，是某个类的实例；
6. Java 异常处理机制在性能上带来一定的开销，特别是在抛出和捕获异常的过程中；

19. **Go 有异常类型吗：**

    1. Go 通常使用**返回 error 类型的值**来判断和处理错误；
    2. 当也可以**使用 panic 和 recover 关键词**来实现异常处理的机制；**panic 用于引发运行时的错误，recover 用于捕获和处理 panic；**

20. **介绍一个 Go 的 panic 和 recover**

    panic 和 recover 是处理运行时错误和恢复程序执行的两个关键字。不过 Go 通常使用返回 error 类型的值来显示的进行错误处理，而不是依赖 panic 和 recover;

    panic：

    1. 是一个内建函数，用于引发运行时错误，通常表示程序遇到了不可恢复的错误；
    2. 当执行到 panic 语句时，会立即停止当前函数的执行，并沿着函数调用栈向上搜寻，执行每个被调用函数的 defer 延迟函数，然后程序终止；
    3. panic 通常用于表示程序遇到了一些致命错误，如切片越界，除以零等；

    recover：

    1. recover 是一个内建函数，用于从 panic 引发的运行时错误中进行恢复；
    2. recover 只能在 defer 延迟函数中使用，用于捕获 panic 的值，并防止程序因 panic 而崩溃；
    3. 如果在 defer 函数中调用了 recover，并且程序处于 panic 状态，那么 recover 将返回 panic 的值，并且程序会从 panic 的地方继续执行；

21. **什么是 defer？？有什么作用：**

    defer 用于延迟函数的执行，它会将函数调用推迟到 defer 语句的函数执行完成后。**通常用于资源释放，锁释放，日志记录等；**

执行顺序：后进先出，即最后一个 defer 会先执行；

函数参数在哪个时刻确定：defer 语句中的函数参数，**在 defer 语句被执行时就已经确定**的了，而不是函数实际调用的时候；

对性能的影响：defer 语句的性能影响通常很小，因为它是在函数退出时执行的；但**在循环中使用大量的 defer 语句，可能会导致性能问题**，因为 defer 语句的执行会被延迟到函数退出时，循环可能会在函数退出前执行许多次；

22. **Go 面向对象是怎么实现的：**

    Go 没有类的概念，而是通过结构体和接口来实现面向对象的特性；

    1. 结构体是一个用户定义的数据类型，可以包含字段（成员变量）和方法（成员函数）
    2. 通过接口定义对象的行为，而不是通过明确的继承关系；（一个类型只要实现了接口定义的方法，就视为实现了该接口）
    3. Go 语言通过结构体的组合特性来实现对象的组合。一个结构体可以包含其他结构体作为其字段，从而实现对象的服用；
    4. 尽管 Go 语言没有传统面向对象语言那样的私有成员修饰符，但可以通过首字母大小写来控制成员的可见性，实现封装效果。

23. **sync.map:**

    特点:

    1. **并发安全：**`sync.Map` 通过内部机制确保在多个 goroutine 同时读写时的安全性。
    2. **非类型安全：**`sync.Map` 的键和值都是 `interface{}` 类型，所以在使用时需要进行类型断言。
    3. **适用于高读写比例：**`sync.Map` 的性能在读多写少的场景中表现较好。

    方法:

    1. **`Store(key, value interface{})`：**将键值对**存储**到映射中。如果键已存在，则会更新对应的值
    2. **`Load(key interface{}) (value interface{}, ok bool)`：**根据键从映射中**获取值**。如果键存在，则返回对应的值和 `true`；否则返回 `nil` 和 `false`。

3. **`LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)`**：如果键已存在，则返回已存在的值和 `true`；如果键不存在，则存储并返回新值和 `false`。

4. **`Delete(key interface{})`**：**删除**指定键值对。

5. **`Range(f func(key, value interface{}) bool)`**：**遍历**映射中的所有键值对，直到函数 `f` 返回 `false` 为止。

二、并发：CSP + Goroutine + Channel + WaitGroup

CSP 并发模型

点我超链接: **GPM 调试模型**

传统并发模型： 传统的多线程编程使用 共享内存 的方式来实现，通过锁，条件变量等同步原语来显性地规定**进程的推进顺序**。

但 CSP 模型不同，不是通过共享内存来通信，而是通过通信来共享内存；

CSP 模型： 通信顺序进程，核心思想是：将并发系统视为一组独立的进程（**独立的并发执行单元，可以是进程，线程和协程**），这些多个进程之间通过消息传递来进行通信。Golang 的并发编程受到 CSP 的启发，通过 `Goroutine` 和 `Channel` 实现并发编程。

特点：

1. 独立性：每个进程（Golang 中是协程）都是基本的并发执行单元，每个协程都是独立的，具有自己的执行上下文，
2. 安全通信：通过消息传递进行通信，不涉及共享内存；
3. 简单的同步机制：`Goroutine` 使用 `Channel` 进行通信，避免了锁和其他复杂的同步原语。

Golang 中： 用于启动协程（独立的并发执行单元）； 用于在间同步/通信，`Channel` 可以是**有缓冲**的也可以是**无缓冲**的。

进程，线程和协程的区别

进程（Processes）：在操作系统中，**进程是拥有独立内存空间的执行单元。（通信方式）**进程之间是**完全隔离**的,进程之间通常通过进程间通信（IPC）机制进行通信，如**管道、消息队列、共享内存**等。**（资源占用）**资源开销较大。

线程（Threads）：**（管理方式）**线程是操作系统管理的轻量级执行单元，独立调度的基本单元，内核级线程，属于同一个进程；**（资源占用）**每个线程有自己的堆栈，占用较多的内存资源；**（可扩展性）**线程的创建受操作系统和硬件的限制；**（通信方式）**线程之间共享进程的内存空间。线程之间的通信通过共享内存和同步原语（如锁、条件变量）进行。

协程（Coroutines）：**（管理方式）**协程是用户态的轻量级线程，由编程语言运行时库管理，也可称为用户级线程；**（资源占用）**每个协程只需要很小的栈空间，几 kb。**（可扩展性）**数以万计的 goroutine 可以在单个或少数几个线程上运行；**（通信方式）**协程之间通过 channel 进行通信，避免了共享内存的复杂性。

Goroutine 是什么

回答: 定义 + 特点

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

如何控制 goroutine 的生命周期

1. 用关键字 `go` 启动 goroutine 启动
2. 主程序通过 `WaitGroup` 等待子协程结束 or 通过 `channel` 通知退出

通过 `WaitGroup` 等待子协程结束	通过 `channel` 通知退出

func main {
var wg sync.WaitGroup
wg.Add(1)
go func(){
defer wg.Done()
// do something
}
wg.wait()
}
	func main {
	quit := make(chan bool)
	go func(){
	// do something
	quit <- true
	}
	<- quit
	}

1. 通过 `context` 包实现超时控制、取消和传递参数

Channel 是什么

点我超链接: 通关 Channel

Channel 是 Go 语言中的一种并发原语，用于在 goroutine 之间**传递数据**和**同步执行**。Channel 实际上是一种特殊类型的数据结构，可以将其想象成一个管道，通过它可以发送和接收数据，实现 goroutine 之间的通信和同步。

消息传递：适用于实现发布-订阅模型或通过消息进行事件通知的场景。

多路复用：使用 `select` 语句，可以在多个 Channel 操作中选择一个**非阻塞**的执行，实现多路复用。

Channel 的特点包括：

1. Channel 是**类型安全**的，可以确保发送和接收的数据类型一致。
2. Channel 是**阻塞**的，当发送或接收操作没有被满足时，会阻塞当前 goroutine，直到满足条件。
3. Channel 是**有缓存**的，可以指定缓存区大小，当缓存区已满时发送操作会被阻塞，当缓存区为空时接收操作会被阻塞。
4. Channel 是**可以关闭**的，可以使用 `close()`函数关闭 Channel，关闭后的 Channel 不能再进行发送操作，但可以进行接收操作。

Channel 的线程安全性

Channel 是线程安全的，可以放心在多个 goroutine 中同时使用 channel 进行数据传递而无需额外的同步机制。

同步机制：Channel 的发送和接收操作是原子操作。Go 运行时确保在同一时刻只有一个 goroutine 在发送或接收数据，从而避免了竞态条件。

阻塞和非阻塞操作：

1. **无缓冲（unbuffered）：**发送和接收操作都是同步的，即发送操作会阻塞直到有另一个 goroutine 执行接收操作，反之亦然。（同步模式）
2. **有缓冲（buffered）：**当缓冲区**未满**时，**发送操作**不会阻塞；当缓冲区**非空**时，**接收操作**不会阻塞。只有在缓冲区**满或空**时，操作才会阻塞。（异步模式）

多 goroutine 并发访问：Channel 设计为可以安全地在多个 goroutine 中同时使用。多个 goroutine 可以同时尝试向同一个 channel 发送或接收数据，Go 运行时会处理这些并发操作以确保线程安全。

从一个关闭的 channel 仍然能读出数据吗

可以，Channel 的关闭指的是关闭发送操作；已在 Channel 的数据依然可以被读取直到被完全读空；可以通过读取的布尔值判断是否读取有效的值，如果为空则说明 Channel 已完全关闭，且通道中没有数据。

操作 Channel 导致 Panic 的情况

1. 向一个**关闭**的 channel 进行写操作；2. **关闭**一个 nil 的 channel；3. **重复关闭**一个 channel。

关闭 Channel 的原则：**不要从一个 receiver 侧关闭 channel，也不要在有多个 sender 时，关闭 channel。**

☆ 如何优雅的关闭 Channel ☆

点我超链接

WaitGroup 是什么？

点我超链接: Channel 和 WaitGroup

WaitGroup（等待组）是 Go 语言中用于管理并发的机制，提供了一种等待一组 Goroutine 完成执行后，再继续执行的方式。

WaitGroup 在 Go 标准库中的 sync 包中的，**它提供了三个基本方法：`Add()`、`Done()`和 `Wait()`。**

`Add()`用于添加需要等待的 Goroutines 的数量。 `Done()`由每个 Goroutine 在完成其工作时调用，以递减等待组的内部计数器。 `Wait()`用于阻塞 Goroutine 的执行，直到所有 Goroutines 都调用了 `Done()`。

通过使用 `Add()`增加等待组的计数器，每个 Goroutine 都会发出需要等待的信号。当 Goroutine 完成其工作时，它调用 `Done()`来减少等待组的计数器。然后，主 Goroutine 或其他 Goroutine 可以调用 `Wait()`来阻塞，直到等待组的计数器达到零。

什么是互斥锁(mutex)

互斥锁是一种用于控制对共享资源访问的同步控制, 它确保在任意时刻只有一个线程能够访问共享资源, 从而避免多个线程同时对资源进行写操作导致的数据竞争和不一致性.

在并发编程中, 多个线程(或 gorouitne)可能同时访问共享数据, 如果不进行同步控制,可能导致: **竞态条件和数据不一致性**.

互斥锁通过在临界区(对共享资源的访问区域)中使用锁来解决这些问题, 基本上,当一个线程获得了互斥锁是,其他线程需要等该线程释放后才能获得锁. 这确保了在任一时刻只有一个线程能够进入临界区.

在 go 语言中,互斥锁使用 sync 包中的 Mutex 类型来实现.

```
var counter int
var mutex sync.Mutex

func increment(wg *sync.WaitGroup){
      defer wg.Done()

      mutex.Lock() // 互斥锁加锁
      counter++
      murex.Unlock() //互斥锁解锁
}

func main(){
      var wg sync.WaitGroup
      for i:=0;i<100;i++{
            wg.Add(1)
            go increment(&wg)
      }
      wg.wait()
}
```

Mutex 有几种模式 和 状态

Mutex 有两种模式: 正常模式 和 饥饿模式

四种状态: mutexLocked: 表示互斥锁的锁定状态, mutexWoken: 表示从正常模式被唤醒, mutexStarving: 当前互斥锁进入饥饿状态, waitersCount: 当前互斥锁上等待的 Goroutine 个数

讲讲 GMP 是什么？

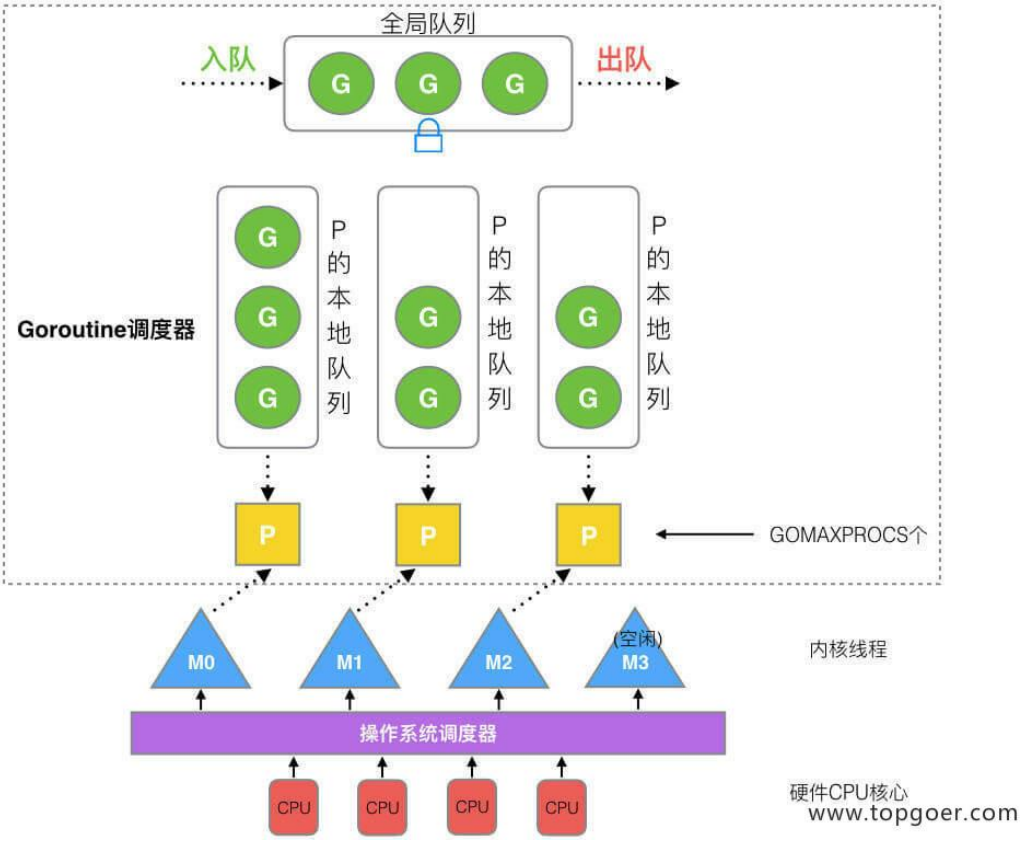
点我超链接

GMP 指的是 G、M 和 P 三个概念的组合，用于描述 Go 运行时的调度模型。这个模型主要用于高效管理和调度并发执行的 Goroutines。

G：Goroutine，goLang 中的协程，是用户态线程；每个 Goroutine 拥有自己的堆栈和执行上下文；每个 Goroutine 占用的内存非常小，几 kb；

M：操作系统线程，用于执行 Goroutine；

P：执行上下文，包含 Goroutines 队列和其他运行时资源。P 负责将 Goroutines 分配给 M 来执行。



img

全局队列（Global Queue）：存放等待运行的 G。
P 的本地队列：同全局队列类似，存放的也是等待运行的 G，存的数量有限，不超过 256 个。新建 G'时，G'优先加入到 P 的本地队列，如果队列满了，则会把本地队列中一半的 G 移动到全局队列。
P 列表：所有的 P 都在程序启动时创建，并保存在数组中，最多有 GOMAXPROCS(可配置) 个。
M：线程想运行任务就得获取 P，从 P 的本地队列获取 G，P 队列为空时，M 也会尝试从全局队列拿一批 G 放到 P 的本地队列，或从其他 P 的本地队列偷一半放到自己 P 的本地队列。M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。

三者关系:

- 每一个运行的 M 都必须绑定一个 P,线程 M 创建后会去检查并执行 G(goroutine)对象
- 每一个 P 保存着一个协程 G 的队列
- 除了每个 P 自身保存的 G 的队列外,调度器还拥有一个全局的 G 队列

- M 从队列中提取 G,并执行
- P 的个数就是 GOMAXPROCS(最大 256),启动时固定的,一般不修改
- M 的个数和 P 的个数不一定一样多(会有休眠的 M 或 P 不绑定 M) (最大 10000)
- P 是用一个全局数组(255)来保存的,并且维护着一个全局的 P 空闲链表

GMP 模型为什么需要 P?

点我超链接

如果不需要 P，那么 M 线程就会直接从全局队列中获取 G 来执行；

1. 调度效率：但是全局队列只有一个，而 M 线程有多个，多线程访问同一资源需要加锁保证互斥/同步，所以会造成激烈的锁竞争；
2. 线程复用困难：此外，当 M 线程运行的 G 发生系统调用或者阻塞的时候，M 线程会被挂起，而不会从全局队列中获取新的 G 运行，运行效率不高；
3. 上下文切换：由于 G 的数量级会远大于 M，操作系统在处理 G 的时候，可能会导致需要频繁的切换上下文和性能瓶颈；
4. 无法控制并发级别，导致线程数量过多，系统性能下降；（P 可通过 GOMAXPROCS 设置）

所以，P 的出现可以解决以上问题：

1. 资源管理和调度：每个 P 负责维护各自的本地队列，能够有效的管理和调度 G；
2. 线程复用：当一个 G 阻塞 M 时，该 M 对应的 P 会寻找其他空闲的 M 来执行，（若没有空闲的，创建新 M 来执行）；
3. 减少上下文切换开销：由于每个 P 自己的 G 队列，M 对 G 的切换主要在同一个 P 中进行，减少了跨 P 切换的开销，提升效率；
4. 限制并发级别：通过 runtime.GOMAXPROCS 设置 P 的数量，可以限制程序的最大并发级别，防止过多的 G 导致系统资源耗尽。

有关 P,M 个数的问题

P 和 M 何时会被创建

GMP 中的抢占式调度

点我超链接： 有具体流程

1. **起源/问题：**在 1.2 版本之前，Go 的调度器是不支持抢占式调度的，程序只能依靠 Goroutine 主动让出 CPU 资源才能切换调度，这会导致两个问题：

    1. 一旦 Goroutine 出现死锁，那么 G 将永久占用对应的 P 和 M，同一个 P 中的其他 G 将不会被调用，出现”饿死“的情况；

    2. 当只有一个 P（GOMAXPROCS=1）的时候，整个 Go 程序的其他 G 都将”饿死“；

    为了解决以上两个问题，Go 分别实现了**基于协作的抢占式调度** 和 **基于信号的抢占式调度**

2. **基于协作的抢占式调度：** G 在执行过程中主动与调度器合作，在合适的时机主动让出 CPU 资源。这种方式依赖于任务在合适的时机自愿让出控制权，以便调度器可以切换到其他任务。

3. **基于信号的抢占式调度：**由调度器主动控制任务切换**，不管协程有没有意愿主动让出 CPU 运行权，只要某个协程执行时间过长（>10ms），就会发送信号强行夺取 CPU 运行权。

Go 什么时候发生阻塞? 阻塞时调度器如何做

- 用于**原子、互斥量或通道操作**导致 G 阻塞，调度器将把**当前阻塞的 G 从本地运行队列 LRQ** 换出，并重新调度其它 G;

- 由于**网络请求和 IO** 导致的阻塞，Go 提供了**网络轮询器** (Netpoller）来处理，后台用 epoll 等技术实现 IO 多路复用。

其他:

- **channel 阻塞:** 当 G 读写 Channel 发生阻塞时, 会调用 gopark 函数,该 G 会脱离当前的 M 和 P,调度器将新的 G 放入当前 M;

- **系统调用:** 当某个 G 由于系统调用陷入内核态, 该 P 会脱离当前 M, 此时 P 会更新自己的状态为 Psycall, M 与 G 互相绑定,进行系统调用. G 系统调用结束后,若该 P 状态还是 Psycall, 则直接关联 M 和 G, 否则使用闲置的处理器处理该 G

- **系统监控:** 当某个 G 在 P 上运行时间超过 10ms, 或者 P 处于 Psycall 状态过长等情况会调用 retake 函数,触发新的调度

- **主动让出:** 由于时协作式调度, 该 G 会主动让出当前的 P,

goroutine 在什么情况下会被挂起呢？

goroutine 被挂起,即调度器重新发起调度更换 P 来执行时:

在 channel 堵塞的时候; 在垃圾回收的时候； sleep 休眠; 锁等待; 抢占； I/O 阻塞

select 实现机制

1. **锁定 scase 中所有 channel**
2. **按照随机顺序检查 scase 中的 channel 是否 ready**
    1. 如果 case 可读, 读取 channel 的数据
    2. 如果 case 可写, 写入 channel
    3. 如果都没有准备好, 就直接返回
3. **所有 case 都没有准备好,而且没有 default**
    1. 将当前的 goroutine 加入到所有 channel 的等待队列
    2. 将当前协程转入阻塞,等待被唤醒
4. **唤醒之后,返回 channel 对于的 case index**

select 总结

1. select 语句中除了 default 之外,每个 case 操作一个 channel, 要么读要么写
2. 除了 default 之外, 各个 case 执行顺序是随机的;
3. 如果 select 中没有 default, 会阻塞等待任意 case;
4. 读操作要判断成功读取,关闭的 channel 也可以读取;

使用场景

超时处理	非阻塞通道操作	多通道等待

`select {
case m := -c:
 	
fmt.Println("Received message:", m)
case <-time.After(3 * time.Second):
fmt.Println("Timeout")
}
	select {
	case msg := -messages:
	
	fmt.Println("Received message", msg)
	default:
	fmt.Println("No message received")
	}
		select {
		case msg1 := -c1:
		
		fmt.Println("Received from c1:", msg1)
		case msg2 := -c2:
		
		fmt.Println("Received from c2:", msg2)
		}

Go 协程的通信方式

使用 channel 进行控制子 goroutine 的机制可以总结为：

循环监听一个 channel,在循环中可以放 select 来监听 channel 达到通知子 goroutine 的效果，再配合 Waitgroup,主进程可以等待所有协程结束后再自己退出;这样就通过 channel 实现了优雅控制 goroutine 并发的开始和结束

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

func worker(id int, stop <-chan bool, wg *sync.WaitGroup) {
	defer wg.Done() // 通知 WaitGroup 当前 goroutine 完成
	for {
		select {
		case <-stop: // 接收到停止信号
			fmt.Printf("Worker %d stopping\n", id)
			return
		default: // 没有接收到停止信号，继续工作
			fmt.Printf("Worker %d is working\n", id)
			time.Sleep(time.Second) // 模拟工作
		}
	}
}

func main() {
	var wg sync.WaitGroup
	stop := make(chan bool) // 创建一个 channel 用于发送停止信号

	for i := 1; i <= 3; i++ { // 启动三个 worker
		wg.Add(1)
		go worker(i, stop, &wg)
	}

	time.Sleep(5 * time.Second) // 主程序休眠一段时间，然后通知所有 worker 停止
	close(stop) // 关闭 channel，所有监听这个 channel 的 goroutine 都会接收到信号

	wg.Wait() // 阻塞，等待所有 worker 停止
	fmt.Println("All workers stopped")
}
```

三、内存

什么是内存溢出现象

内存溢出是指程序在运行过程中**消耗了比系统或环境能提供的更多的内存**，导致程序崩溃或性能显著下降。比如：

1. **无限循环或递归:** 如果代码中出现了无限循环或递归，没有合适的终止条件，会导致内存不断增长，最终造成溢出。
2. **创建过大的数据结构:** 如果程序中创建了过大的数组、切片、字典或其他数据结构，而系统内存不足以支撑该数据结构的大小，会引起内存溢出。
3. **频繁的 I/O 或网络请求:** 如果程序频繁地进行 I/O 操作或网络请求，并且没有控制并发的数量，可能会在短时间内分配大量内存，导致溢出。

检测方法：

1. 使用 pprof 观察内存分配情况，查看是否存在某些数据结构或代码路径消耗了大量内存。
2. 观察系统日志中的“out of memory”错误。
3. 使用 Go 自带的 go tool trace 和 go tool pprof 工具。

什么是内存泄漏现象

内存泄漏是指**一些不再需要的内存无法被垃圾回收（GC）回收**，导致程序内存使用量持续增加，最终可能引发内存溢出。

1. 有对象或数据结构被引用但不再使用（例如，将大对象存储在全局变量或闭包中）。
2. 使用缓存在不释放的切片、映射等结构中存储了大量数据。
3. 在某些数据结构中有过多未清理的元素，导致 GC 无法回收这些内存。

检测方法：

1. 使用 pprof 查看内存分配图，观察是否有未释放的内存。
2. 监控垃圾回收（GC）日志，GC 频率异常高且内存占用未显著下降。
3. 使用第三方监控工具（如 Prometheus + Grafana）来可视化程序的内存占用情况。

什么是内存逃逸现象

内存逃逸(Memory Escape）是指**一个变量在函数内部创建，但在函数结束后仍然被其他部分引用，导致变量的生命周期超出了函数的范围**，从而使得该变量的内存需要在堆上分配**而不是在栈上分配**。

1. **当在函数内部创建了一个局部变量**,然后返回该变量的指针, 而该指针**被函数外部的代码引用**时, 这个局部变量会发生内存逃逸;

2. **当将局部变量通过 channel 或 goroutine 传递给其他部分**, 而这些部分**可能在原始函数退出后访问这个变量时**,也会导致内存逃逸;

3. **在函数内部使用 new 或 make 分配的变量**, 即使返回的是指针, **但这个指针可能被外部持有**,从而导致变量在堆上分配而不是在栈上分配;

检测方法： 在编译代码时使用 -gcflags="-m" 标志，比如 go build -gcflags="-m" main.go，编译器会输出哪些变量发生了逃逸。编译器会在输出中标记 moves to heap 的变量，指示这些变量发生了逃逸。

类型	描述	检测方法	预防措施
内存溢出	程序内存使用超出系统限制，导致崩溃	pprof、系统日志	控制数据大小、并发数量、分批处理数据
内存泄漏	未被使用的内存无法被 GC 回收，内存占用持续增加	pprof、监控工具、GC 日志	释放不再使用的引用、控制缓存大小
内存逃逸	栈内变量分配到堆上，增加 GC 负担，影响性能	-gcflags="-m" 编译输出	避免不必要的指针、闭包、interface{} 的使用

垃圾回收(GC)机制

Golang 使用的垃圾回收机制是**三色标记法**，三色标记法是**标记-清除法**的一种增强版本。

暂停程序（Stop the world，STW）

随着用户程序申请越来越多的内存，系统中的垃圾也逐渐增多；当程序的内存占用达到一定阈值时，整个应用程序就会全部暂停，垃圾收集器会扫描已经分配的所有对象并回收不再使用的内存空间，当这个过程结束后，用户程序才可以继续执行，

标记清除（Mark-Sweep）

1. 原始的标记清楚法分为两个步骤：

    1. 标记。先 STP(Stop The World)，暂停整个程序的全部运行线程，将被引用的对象打上标记

    2. 清除没有被打标机的对象，即回收内存资源，然后恢复运行线程。

    这样做有个很大的问题就是要通过 STW 保证 GC 期间标记对象的状态不能变化，整个程序都要暂停掉，在外部看来程序就会卡顿。

三色标记法

实现三色标记法的变种以缩短 STW 的时间。三色标记算法将程序中的对象分成白色、黑色和灰色三类 4：

- 白色对象 — 潜在的垃圾，其内存可能会被垃圾收集器回收；
- 黑色对象 — 活跃的对象，包括不存在任何引用外部指针的对象以及从根对象可达的对象；
- 灰色对象 — 活跃的对象，因为存在指向白色对象的外部指针，垃圾收集器会扫描这些对象的子对象；

根搜索：垃圾回收从根对象开始搜索，根对象包括全局变量、栈上的对象以及其他一些持有对象引用的地方。所有根对象被标记为灰色，表示它们是待处理的对象。

标记阶段：从灰色对象开始，垃圾回收器遍历对象的引用关系，将其引用的对象标记为灰色，然后将该对象标记为黑色。这个过程一直进行，直到所有可达对象都被标记为黑色。

并发标记：在标记阶段，垃圾回收器采用并发标记的方式，与程序的执行同时进行。这意味着程序的执行不会因为垃圾回收而停顿，从而减小了对程序性能的影响。

清扫阶段：在标记完成后，垃圾回收器会扫描堆中的所有对象，将未被标记的对象回收(释放其内存)。这些未被标记的对象被认为是不可达的垃圾。

内存返还：垃圾回收完成后，系统中的内存得以回收并用于新的对象分配。

GC 触发：垃圾回收的触发条件通常是在分配新对象时，如果达到一定的内存分配阈值，就会触发垃圾回收。

另外，一些特定的事件（如系统调用、网络阻塞等）也可能触发垃圾回收。

三色标记法+ 混合写屏障

写屏障技术: 当对象新增或者更新会将其着色为**灰色**。

总而言之就是, 确保**黑色对象不能引用白色对象**，这个改进直接使得 GC 时间从 2s 降低到 2us。

GC 的流程是什么?

G01.14 版本以 STW 为界限，可以将 GC 划分为五个阶段：

1. GCMark **标记准备阶段**，为并发标记做准备工作，**启动写屏障**
2. STWGCMark **扫描标记阶段**，与赋值器并发执行，写屏障开启并发
3. GCMarkTermination **标记终止阶段**，保证一个周期内标记任务完成，**停止写屏障**
4. GCoff **内存清扫**阶段，将需要回收的内存**归还到堆**中，写屏障关闭
5. GCoff **内存归还**阶段，将过多的内存**归还给操作系统**，写屏障关闭。

GC 如何调优

通过 `go tool pprof` 和 `go tooltrace` 等工具

1. 控制内存分配的速度，**限制 Goroutine 的数量**，从而提高赋值器对 CPU 的利用率。
2. **减少并复用内存**，例如使用 sync.Pool 来复用需要频繁创建临时对象，例如提前分配足够的内存来降低多余的拷贝。
3. 需要时，**增大 GOGC 的值**，降低 GC 的运行频率。

四、GORM

如何在 GORM 中使用原生 SQL 查询？

读操作：使用 Raw 方法来执行原生 SQL 查询。Raw 方法接受一个 SQL 查询字符串和可选的参数列表，并返回一个 *gorm.DB 对象

写操作：还可以使用 Exec 方法来执行不需要返回值的 SQL 查询

GORM 的 Preload 方法和 Joins 方法有什么区别？在什么情况下使用哪种方法更好？

Preload 方法是在查询时预加载关联数据，而 Joins 方法是通过 SQL JOIN 语句连接多个表查询数据。

Preload 方法适用于关联表较少、数据量不大的情况；而 Joins 方法适用于关联表较多、数据量较大的情况。

如何进行事务管理？/ 事务处理有哪些注意点？

GORM 的事务管理使用 Begin、Commit 和 Rollback 方法实现。

首先启动事务时一定要做 **错误判断**。

建议在 **启动事务** 之后马上写 **defer 方法**。

在 **defer 方法** 内对 **err** 进行判断，如果 **全局** 中有 **err!=nil** 就回滚 (全局中 **err** 都为 **nil** 才能 **提交事务**)

在 **提交事务** 之后我们可以定义一个 **钩子函数 afterCommit**，来统一处理事务提交后的 **逻辑**。

```
tx, err := g.DB().Begin()  // 启动事务
if err != nil {
   return errors.New("启动事务失败")
}
```

```go
defer func() {
   if err != nil {
      tx.Rollback()  // 回滚
   } else {
      tx.Commit()    // 事务完成
      //定义钩子函数
      afterCommmit()
   }
}()
```

五、Go Web

Gin 框架

Gin 是一个用于构建 Web 应用的 API 轻量级的 Go 语言框架。拥有高性能和简洁的 API 设计；

1. Gin 提供了灵活而简单的路由机制，支持参数和通配符；通过 Gin，可以轻松定义路由并处理不同的 HTTP 请求方法；
2. Gin 支持中间件，可以在请求到达处理程序之前或之后执行额外的逻辑，可以实现日子记录，身份验证，错误处理等功能变得非常简单；
3. Gin 提供了简便的方法来处理 JSON 和 XML 数据，通过 c.Json 和 c.xml 等方法，可以方便的构建 HTTP 响应；

Gin 拦截器的原理

在 Gin 中，拦截器通常被称为中间件。中间件允许在请求到达处理函数之前或之后执行一些预处理和后处理逻辑。

Gin 的中间件机制是基于 Go 的函数闭包和 gin.Context 特性。

1. 实现中间件函数： 该函数接受一个 gin.Context 对象作为参数，并执行一些逻辑。
2. 注册中间件：使用 Use 方法注册中间件，在路由定义中使用 Use 方法添加中间件函数，对整个路由组或单个路由生效；
3. 中间件链： 可以添加多个中间件，形成中间件链；中间件的执行根据添加的顺序依次执行；
4. 中间件的执行顺序： 在执行完一个中间件的逻辑后，通过 c.Next()将控制权传递给下一个中间件或处理函数，如果中间件没有调用 c.Next()，后续中间件没有调用 c.Next()，后续中间件和处理函数将不会被执行；

Gin 的路由是怎么实现的

路由的实现是通过 gin.Engine 类型来管理的，该类型实现了 http.Handler 接口,因此可以执行用作 http.ListenAndServe 的参数.

Gin 的路由包括基本路由、参数路由和组路由;

介绍一下 Go 中 Context 的作用

1. context 可以用来在 goroutine 之间传递上下文信息, 相同的 context 可以传递给运行在不同 goroutine 中的函数,上下文对于多个 goroutine 同时使用是安全的;

2. context 包定义了上下文类型,可以使用 background、TODO 创建一个上下文, 在函数调用链之间传播 context;

总结: context 的作用是在不同的 goroutine 之间同步请求特定的数据,取消信号以及处理请求的截止日期;

Gin 框架常用库

1. go-jwt: 用于 Gin 框架的 JWT 中间件

2. go-cors: 用于处理跨域请求的中间件

3. zap: 用于记录日志;

4. validateap: 用于参数校验;

Gin 框架如何实现跨域？

Gin 框架跨域问题

1.写一个 **中间件** 来配置 **跨域**。

2.使用官网提供的 **插件**: **github.com/gin-contrib/cors**。

其他组件

介绍一下 grpc？

GRPC 是一种高性能、开源的远程过程调用（RPC）框架，由 Google 开发并基于 HTTP/2 协议实现。它允许在不同的计算机之间进行跨语言和跨平台的通信，使得构建分布式系统变得更加简单和高效。

GRPC 使用 Protocol Buffers（简称 Protobuf）作为默认的序列化机制，而不是使用 JSON。

gRPC 的主要特点

1. **高性能：gRPC 使用 HTTP/2 来提供多路复用、头部压缩和双向流**，因此它比传统的 HTTP/1.1 更高效。
2. **跨语言支持：**gRPC **支持多种编程语言**，如 C++, Java, Python, Go 等。这使得它在多语言项目中非常实用。
3. **简单的定义服务：**使用 Protocol Buffers 作为接口定义语言（IDL），可以**简单地定义服务和消息结构**。
4. **流式传输：**支持四种不同类型的服务方法，包括**单向和双向流式传输**，适用于各种实时通信需求。
5. **自动生成代码：**根据 Protocol Buffers 定义的服务，**gRPC 可以自动生成客户端和服务器端的代码**，大大减少了手工编码的工作量。

使用 gRPC 的基本步骤

1. **定义服务：**
   - 使用 `.proto` 文件来定义服务和消息。例如：

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

1. **生成代码：**使用 `protoc` 编译器根据 `.proto` 文件生成客户端和服务器端的代码。
2. **实现服务逻辑：**在服务器端实现定义好的服务逻辑。
3. **启动服务器：**启动 gRPC 服务器并监听特定的端口。
4. **调用服务：**在客户端调用服务，发送请求并处理响应。

gRPC 和 REST 的比较

1. **性能：**gRPC 使用 HTTP/2，性能优于基于 HTTP/1.1 的 REST。

2. **数据格式**：gRPC 使用 Protocol Buffers，而 REST 通常使用 JSON。**前者更高效**但更复杂。
3. **服务定义**：gRPC 使用 `.proto` 文件定义服务，REST 通常使用 OpenAPI 或 Swagger。
4. **流式传输**：gRPC 原生支持双向流，而 REST 通常不支持。

protobuf/json 区别与优势？

Protobuf 是一种轻量级的数据交换格式，具有以下优势：

1. **效率高**：Protobuf 使用**二进制编码**，相比于文本格式的 JSON，它的**编码和解码速度更快，传输的数据量更小**，节省了带宽和存储空间。
2. **可读性好**：虽然 Protobuf 是二进制格式，但它的**定义文件是可读的**，易于理解和维护。相比之下，JSON 是一种文本格式，可读性较好，但在大型数据结构的情况下，Protobuf 的定义文件更加清晰和简洁。
3. **跨语言支持**：Protobuf **支持多种编程语言**，包括 Java、C++、Python 等，这使得在不同语言之间进行通信变得更加方便。
4. **版本兼容性**：Protobuf **支持向后和向前兼容的数据格式演化**，这意味着可以在不破坏现有客户端和服务端的情况下，对数据结构进行扩展和修改。

相比之下，JSON 是一种常用的文本格式，具有以下特点：

1. 可读性好：JSON 使用文本格式，易于阅读和理解，对于调试和开发过程中的数据交换非常方便。
2. 平台无关性：JSON 是一种独立于编程语言的数据格式，几乎所有的编程语言都支持 JSON 的解析和生成。
3. 灵活性：JSON 支持动态的数据结构，可以轻松地添加、删除和修改字段，适用于一些需要频繁变动的数据。

总的来说，GRPC 使用 Protobuf 作为默认的序列化机制，相比于 JSON，Protobuf 在**性能、可读性和跨语言支持**方面具有优势。然而，选择使用 GRPC 还是 JSON 取决于具体的应用场景和需求。

参考

面经

https://www.nowcoder.com/discuss/353154773744558080

Go 程序员面试笔试宝典：https://golang.design/go-questions/

http://interview.wzcu.com/Golang/%E5%9F%BA%E7%A1%80.html

高性能

https://geektutu.com/post/hpg-escape-analysis.html

高并发

https://github.com/IBBD/IBBD.github.io/blob/master/golang/golang-high-concurrence.md

https://github.com/xiaobaiTech/golangFamily?tab=readme-ov-file