代码随想录知识星球精华（最强八股文）第五版（Go篇））

代码随想录知识星球精华（最强八股文）第五版为九份PDF，分别是：

- 代码随想录知识星球八股文概述
- C++篇
- go篇
- Java篇
- 前端篇
- 算法题篇
- 计算机基础篇
- 问答精华篇
- 面经篇

本篇为最强八股文之Go篇。

面试题

精选Go语言面试题50+题，均为常考察内容

Go语言基础

介绍一下Go语言的特点和优势。

- 语法简单
- 支持轻量级线程（goroutine）和通信（channel），高效并发
- 内置垃圾回收

Go 和 Java 对比

1. Java使用广泛，但是Go比Java更适合高并发和轻量级的应用
2. Java通过线程和锁来处理并发， Goroutines和channels是Go语言的并发特性的核心
3. Java是一门功能丰富、面向对象的语言，支持面向对象编程、泛型等高级特性。 Go语言的设计注重简洁和清晰，具有简单的语法和类型系统。它摒弃了一些复杂的特性，强调代码的可读性。
4. Go语言具有垃圾回收机制，开发者无需手动管理内存。Java同样拥有垃圾回收机制，这减轻了开发者的负担，但在一些情况下可能引入一些不可控的暂停。
5. Go适用于构建高性能、高并发的后端服务、网络应用、云服务以及分布式系统。Java广泛应用于大型企业应用、Android应用、大规模分布式系统和企业级应用。

Go string 和 []byte 的区别

如果需要频繁地修改字符串内容，或者处理二进制数据，使用 `[]byte` 更为合适。如果字符串内容基本保持不变，并且主要处理文本数据，那么使用 `string` 更为方便。

1. 不可变性

`string` 是不可变的数据类型，一旦创建就不能被修改。任何修改 `string` 的操作都会产生一个新的 `string`，而原始的 `string` 保持不变。相比之下，`[]byte` 是可变的切片，可以通过索引直接修改切片中的元素。

2. 类型转换

可以在 `string` 和 `[]byte` 之间进行类型转换。使用 `[]byte(s)` 可以将 `string` 转换为 `[]byte`，而使用 `string(b)` 可以将 `[]byte` 转换为 `string`。这个操作会创建新的底层数组，因此在转换后修改其中一个不会影响另一个。

3. 内存分配

- `string` 是一个不可变的视图，底层数据是只读的。`string` 的内存分配和释放由Go运行时管理。
- `[]byte` 是一个可变的切片，底层数据是可以修改的。`[]byte` 的内存管理由程序员负责。

4. Unicode字符

`string` 中的每个元素是一个 Unicode 字符，而 `[]byte` 中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。

make和new的区别

`make`和`new`是两个用于分配内存的内建函数, 在使用场景和返回值类型上有明显的区别。

- `make` 用于创建并初始化切片、映射和通道等引用类型。它返回的是被初始化的非零值（非nil）的引用类型。

```
// 创建并初始化切片
slice := make([]int, 5, 10)

// 创建并初始化映射
myMap := make(map[string]int)

// 创建并初始化通道
ch := make(chan int)
```

- `new` 用于分配值类型的内存，并返回该值类型的指针。它返回的是分配的零值的指针。

```
// 分配整数类型的内存，并返回指针
ptr := new(int)
```

```go
package main

import "fmt"

func main() {
  // 使用 make 创建并初始化切片
  slice := make([]int, 5, 10)
  fmt.Println(slice) // 输出：[0 0 0 0 0]

  // 使用 new 分配整数类型的内存，并返回指针
  ptr := new(int)
  fmt.Println(*ptr) // 输出：0
}
```

总结：

- new只用于分配内存，返回一个指向地址的**指针**。它为每个新类型分配一片内存，初始化为0且返回类型*T的内存地址，它相当于&T{}
- make只可用于**slice,map,channel**的初始化,返回的是**引用**。

数组和切片的区别

1. 数组

- 固定长度，在声明数组时，需要指定数组的长度，且不能更改。
- 值类型，当将一个数组赋值给另一个数组时，会进行值拷贝。这意味着修改一个数组的副本不会影响原始数组。
- 数组的元素在内存中是顺序存储的，分配在一块连续的内存区域

2. 切片

- 切片的长度可以动态调整，而且可以不指定长度。
- 切片是引用类型，当将一个切片赋值给另一个切片时，它们引用的是相同的底层数组。修改一个切片的元素会影响到其他引用该底层数组的切片。
- 切片本身不存储元素，而是引用一个底层数组。切片的底层数组会在需要时进行动态扩展。

```go
// 创建切片
slice1 := make([]int, 3, 5)  // 长度为3，容量为5的切片
slice2 := []int{1, 2, 3}     // 直接初始化切片
slice3 := arr1[:]            // 从数组截取切片
```

切片是如何扩容的

- 切片的扩容容量是按指数增长的。当切片的容量不足时，Go运行时系统会分配一个更大的底层数组，并将原来的元素拷贝到新数组中。新数组的大小通常是原数组的两倍（但并不一定严格遵循2倍关系）
- 在切片扩容时，Go运行时系统会预估未来的元素增长，并提前分配足够的空间。这可以减少频繁的内存分配和拷贝操作。
- 对于小切片，扩容时增加的容量可能相对较小，避免了内存的过度浪费。而对于大切片，扩容时增加的容量可能较多。

首先判断，如果新申请容量大于2倍的旧容量，最终容量就是新申请的容量
否则判断，如果旧切片的长度小于1024，则最终容量就是旧容量的两倍
否则判断，如果旧切片长度大于等于1024，则最终容量从旧容量开始循环,增加原来的 1/4，直到最终容量大于等于新申请的容量
如果最终容量计算值溢出，则最终容量就是新申请容量

扩容前后的Slice是一样的吗

如果扩容后的容量仍然能够容纳新元素，系统会尽量在原地进行扩容，否则会分配一个新的数组，将原有元素复制到新数组中。

go slice的底层实现

切片本身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用 底层数组，设定相关属性将数据读写操作限定在指定的区域内。切片本身是一 个只读对象，其工作机制类似数组指针的一种封装。 主要通过一个结构体来表示，该结构体包含了以下三个字段：

```
type slice struct {
    array unsafe.Pointer // 指向底层数组的指针
    len   int            // 切片的当前长度
    cap   int            // 切片的容量
}
```

Go语言参数传递

对于基本数据类型（如整数、浮点数、布尔值等）和结构体，传递的是值的副本，修改形参的值不会影响实参。

```
package main

import "fmt"

func modifyValue(x int) {
    x = 10
}

func main() {
    a := 5
    modifyValue(a)
    fmt.Println(a) // 输出：5，因为a的值未被修改
}
```

对于切片、映射、通道等引用类型，传递的是引用的副本，修改形参的内容会影响到实参。

```go
package main

import "fmt"

func modifySlice(s []int) {
    s[0] = 10
}

func main() {
    slice := []int{1, 2, 3}
    modifySlice(slice)
    fmt.Println(slice) // 输出：[10 2 3]，因为slice的内容被修改
}
```

Go语言map是有序的还是无序的, 为什么

Go语言中，`map`是一种用于存储键值对的集合类型。它是一种无序的集合，其中每个元素都由一个唯一的键和对应的值组成。当`map`的元素数量达到一定阈值时，Go语言会动态调整`map`的大小。

这是因为`map`的实现采用了散列表（hash table）的数据结构。散列表通过哈希函数将键映射到存储桶（bucket）散列表中的存储桶是无序的，它们并不保证元素按照特定顺序存储。

go map的底层实现原理

Go语言中的`map`的底层实现原理主要基于散列表（hash table）。散列表是一种用于实现字典结构的数据结构，它通过一个哈希函数将键映射到存储桶（bucket），每个存储桶存储一个链表或红黑树，用于处理哈希冲突。存储桶的数量是固定的，由`map`的大小和负载因子来确定。

map如何扩容

Go语言中的`map`在元素数量达到一定阈值时，会触发扩容操作，其扩容是自动进行的。

1. **计算新的存储桶数量：**当`map`的元素数量达到负载因子（load factor）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。这个过程会涉及到内存分配。
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。这一步是为了保持元素在新的存储桶中的顺序。
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，以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：**释放旧的存储桶和散列数组的内存空间。这个过程是为了避免内存泄漏。

如何想要按照特定顺序遍历map,怎么做

1. 遍历`map`，将键存储在切片中, 切片是有序的。
2. 使用排序函数对存储键的切片进行排序。
3. 使用排好序的切片，按照顺序遍历`map`。

下面是一个演示：按照键的字母顺序遍历`map`

```go
package main

import (
  "fmt"
  "sort"
)

func main() {
  myMap := map[string]int{
    "apple":  5,
    "banana": 3,
    "orange": 7,
    "grape":  1,
  }

  // 将键存储在切片中
  keys := make([]string, 0, len(myMap))
  for key := range myMap {
    keys = append(keys, key)
  }

  // 对切片进行排序
  sort.Strings(keys)

  // 按照排序后的顺序遍历map
  for _, key := range keys {
    fmt.Printf("%s: %d\n", key, myMap[key])
  }
}
```

go里面的map是并发安全的吗？如何并发安全

Go 中的标准 `map` 类型是非并发安全的，这意味着在多个 Goroutine 中并发读写同一个 `map` 可能导致数据竞争和不确定的行为。为了在并发环境中使用 `map`，Go 提供了 `sync` 包中的 `sync.Map` 类型，它是一种并发安全的映射。

```go
import "sync"

// 创建一个并发安全的 map
var myMap sync.Map

// 在 Goroutine 中使用
go func() {
    // 存入数据
    myMap.Store("key", "value")

    // 从 map 中读取数据
    if value, ok := myMap.Load("key"); ok {
```

```
        // 处理 value
    }
}()
```

Go 的错误处理和 Java 的异常处理对比

1. Go

- Go语言使用返回值来处理错误，函数通常返回两个值，一个是正常的返回值，另一个是`error`类型的值，用于表示可能出现的错误。开发者需要显式地检查错误并进行处理，通过判断返回的`error`值是否为`nil`来确定函数是否执行成功。
- Go中的错误是普通的值，是实现了`error`接口的类型。
- Go的错误处理机制在性能上通常更为高效，因为它不会引入额外的控制流程（异常栈的构建和查找等）

2. Java

- java使用异常机制处理错误。当出现错误时，可以通过`throw`关键字抛出异常，而在调用栈中寻找匹配的`catch`块来捕获并处理异常。
- Java中的异常是对象，是某个类的实例。Java的异常类型必须继承自`Throwable`类或其子类
- 异常处理机制可能在性能上带来一定开销，特别是在抛出和捕获异常的过程中。

```
try {
    // 可能抛出异常的代码
    result = someFunction();
} catch (SomeException e) {
    // 处理异常
}
```

```
result, err := someFunction()
if err != nil {
    // 处理错误
}
```

Go有异常类型吗

Go鼓励使用返回值来处理错误, 在Go中，函数通常会返回两个值，其中一个是函数的正常返回值，另一个是`error`类型的值，表示函数执行是否成功。

```
result, err := someFunction()
if err != nil {
    // 处理错误
}
```

此外，还可以通过使用 `panic` 和 `recover` 关键字来实现类似异常处理的机制。`panic` 用于引发运行时错误，而 `recover` 用于捕获并处理 `panic`

```
func example() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
    panic("Something went wrong!")
}

func main() {
    example()
}
```

介绍一下panic和recover

`panic` 和 `recover` 是用于处理运行时错误和恢复程序执行的两个关键字。但是在一般情况下，Go语言更倾向于使用显式的错误处理，而不是依赖于 `panic` 和 `recover`。

1. panic

- `panic` 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。
- 当程序执行到 `panic` 语句时，它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（如果有的话），然后程序终止。
- `panic` 通常用于表示程序遇到了一些致命错误，例如切片越界、除以零等。

```
func example() {
    panic("Something went wrong!")
}

func main() {
    example()
}
```

2. recover

- `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。
- `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。
- 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。

```
func example() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()

    panic("Something went wrong!")
```

```
}

func main() {
    example()
    fmt.Println("Program continues after panic recovery.")
}
```

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包含 `defer` 语句的函数执行完成之后。通常用于资源释放、锁的释放、日志的记录等。

Q: 执行顺序

`defer`语句是按照后进先出（LIFO）的顺序执行的，即最后一个`defer`语句会最先执行。

Q: 函数参数是在哪个时刻确定的?

`defer`语句中的函数参数在 `defer` 语句被执行时就已经确定了，而不是在函数实际调用时。因此，如果`defer`语句中有函数参数，这些参数的值是在`defer`语句执行时就会被计算并保留。

Q: 对性能有没有影响

`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。但如果在循环中使用了大量的`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之前执行许多次。

Q: **在什么情况下会有问题?**

如果在循环中使用`defer`，并且`defer`中引用了循环变量，由于`defer`语句的延迟执行特性，可能导致循环结束后函数执行时使用的是最后一次循环变量的值。这被称为"defer在循环中的陷阱"。

```
for i := 0; i < 5; i++ {
    defer func() {
        fmt.Println(i)
    }()
}
```

上述代码输出的结果是5个5，而不是0到4。避免这种问题的一种方法是在循环体内部创建一个局部变量，将循环变量的值传递给`defer`中的函数。

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（struct）和接口（interface）来实现面向对象的特性。

1. 结构体是一种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）。

```
type Person struct {
    Name string
    Age  int
}

// 方法
func (p *Person) SayHello() {
    fmt.Println("Hello, my name is", p.Name)
}
```

2. Go语言通过接口来定义对象的行为，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，就被视为实现了该接口。

```
type Speaker interface {
    Speak()
}

type Person struct {
    Name string
}

// Person 实现了 Speaker 接口
func (p *Person) Speak() {
    fmt.Println("Hello, my name is", p.Name)
}
```

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，从而实现对象的复用。
4. 尽管Go语言没有像传统面向对象语言那样的私有成员访问修饰符，但通过首字母大小写来控制成员的可见性，实现了封装的效果。首字母大写的成员是公有的，可以被外部包访问；首字母小写的成员是私有的，只能在定义的包内访问。

Go并发

进程、线程、协程

进程、线程和协程都是并发编程的概念

进程是操作系统分配资源的基本单位，每个进程都有自己的独立内存空间，不同进程之间的数据不能直接共享，通常通过进程间通信（IPC）来进行数据交换，例如管道、消息队列等。

线程是操作系统调度的最小执行单位，同一进程的不同线程共享相同的内存空间，可以直接访问共享数据。

协程是轻量级的用户态线程，由Go调度器进行管理，协程的创建和销毁比线程更为轻量，可以很容易地创建大量的协程。协程之间通过通信来共享数据，而不是通过共享内存。这通过使用通道（channel）等机制来实现。

进程、线程的区别

- 调度:进程是资源管理的基本单位，线程是程序执行的基本单位。
- 切换:线程上下文切换比进程上下文切换要快得多。
- 拥有资源: 进程是拥有资源的一个独立单位，线程不拥有系统资源，但是可以访问隶属于进程的资源。
- 系统开销: 创建或撤销进程时，系统都要为之分配或回收系统资源，如内存空间，I/O 设备等，OS 所付出的开销显著大于在创建或撤销线程时的开 销，进程切换的开销也远大于线程切换的开销。

协程和线程的区别

- 线程和进程都是同步机制，而协程是异步机制。
- 线程是抢占式，而协程是非抢占式的。需要用户释放使用权切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。
- 一个线程可以有多个协程，一个进程也可以有多个协程。
- 协程不被操作系统内核管理，而完全是由程序控制。线程是被分割的CPU资源，协程是组织好的代码流程，线程是协程的资源。但协程不会直接使用线程，协程直接利用的是执行器关联任意线程或线程池。
- 协程能保留上一次调用时的状态。

并行和并发的区别

- 并发就是在一段时间内，多个任务都会被处理;但在某一时刻，只有一个任务 在执行。单核处理器可以做到并发。比如有两个进程 A 和 B，A 运行一个时间片之后，切换到 B，B 运行一个时间片之后又切换到 A。因为切换速度足够快，所以宏观上表现为在一段时间内能同时运行多个程序。
- 并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。

Go语言并发模型

Go语言的并发模型建立在goroutine和channel之上。其设计理念是**共享数据通过通信而不是通过共享来实现**

- Goroutines 是Go中的轻量级线程，由Go运行时（runtime）管理。与传统线程相比，goroutines的创建和销毁开销很小。程序可以同时运行多个goroutines，它们共享相同的地址空间。
- Goroutines之间的通信通过channel（通道）实现。通道提供了一种安全、同步的方式，用于在goroutines之间传递数据。使用通道可以避免多个goroutines同时访问共享数据而导致竞态条件的问题。
- 多路复用： `select` 语句允许在多个通道操作中选择一个执行。这种方式可以有效地处理多个通道的并发操作，避免了阻塞。
- 互斥锁和条件变量
  - Go提供了`sync`包，其中包括 `Mutex`（互斥锁）等同步原语，用于在多个goroutines之间进行互斥访问共享资源。
  - `sync` 包还提供了 `Cond`（条件变量），用于在goroutines之间建立更复杂的同步。
- 原子操作： Go提供了`sync/atomic`包，其中包括一系列原子性操作，用于在不使用锁的情况下进行安全的并发操作。

什么是go runtine

goroutine（协程）是一种轻量级的线程，由Go运行时（runtime）管理，一个典型的 Go 程序可能会同时运行成千上万个 goroutine，Goroutines 使得程序可以并发执行，而无需显式地创建和管理线程。通过关键字 `go` 可以启动一个新的 goroutine，例如：`go someFunction()`。

每个 goroutine 都有自己的独立栈空间，这使得它们之间的数据不容易互相干扰。与传统的多线程编程相比，使用 goroutines 不需要开发者显式地进行线程的创建、销毁和同步。Go 运行时会自动处理这些事务。

如何控制 goroutine 的生命周期

1. 启动

使用关键字 `go` 可以启动一个新的 goroutine。

```
go func() {
    // goroutine 的代码逻辑
}()
```

2. 等待结束

希望主程序等待某个 goroutine 执行完毕后再继续执行。可以使用 `sync.WaitGroup` 来实现等待。

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup

    wg.Add(1) // 添加一个等待的 goroutine

    go func() {
        defer wg.Done() // goroutine 完成时调用 Done 减少计数
        // goroutine 的代码逻辑
        fmt.Println("Goroutine executing...")
    }()

    // 等待所有 goroutine 完成
    wg.Wait()

    fmt.Println("Main goroutine exiting.")
}
```

3. 使用通道（channel）来通知 goroutine 退出

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    quit := make(chan bool)

    go func() {
        defer fmt.Println("Goroutine exiting...")
        // goroutine 的代码逻辑
        time.Sleep(time.Second * 2)
        quit <- true // 发送退出通知
    }()

    // 主 goroutine 等待退出通知
    <-quit
    fmt.Println("Main goroutine exiting.")
}
```

4. 使用context包

Go 标准库中的 `context` 可以实现超时控制、取消、传递参数等功能。

Go语言中的Channel是什么, 有哪些用途，如何处理阻塞

Channel（通道）是用于在goroutines之间进行通信的一种机制。通道提供了一种并发安全的方式来进行goroutines之间的通信。通过通道，可以避免在多个goroutines之间共享内存而引发的竞态条件问题，因为通道的读写是原子性的。

用途

- **数据传递：** 主要用于在goroutines之间传递数据，确保数据的安全传递和同步。
- **同步执行：** 通过Channel可以实现在不同goroutines之间的同步执行，确保某个goroutine在另一个goroutine完成某个操作之前等待。
- **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。

如何处理阻塞

1. 缓冲通道，在创建通道时指定缓冲区大小，即创建一个缓冲通道。当缓冲区未满时，发送数据不会阻塞。当缓冲区未空时，接收数据不会阻塞。
2. `select`语句用于处理多个通道操作，可以用于避免阻塞。
3. 使用`time.After`创建一个定时器，可以在超时后执行特定的操作，避免永久阻塞。
4. `select`语句中使用`default`分支，可以在所有通道都阻塞的情况下执行非阻塞的操作。

什么是互斥锁（mutex）？在什么情况下会用到它们?

互斥锁是一种用于控制对共享资源访问的同步机制。它确保在任意时刻只有一个线程能够访问共享资源，从而避免了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（或者Goroutines）可能同时访问共享的数据，如果不进行同步控制，可能导致以下问题：

- **竞态条件（Race Condition）：** 多个线程同时修改共享资源，导致最终结果依赖于执行时机，可能引发不确定的行为。
- **数据不一致性：** 多个线程同时读写共享资源，可能导致数据不一致，破坏了程序的正确性。

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时，其他线程需要等待该线程释放锁后才能获得锁。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，互斥锁通常使用`sync`包中的`Mutex`类型来实现。以下是一个简单的示例：

```
package main

import (
  "fmt"
  "sync"
)

var counter int
var mutex sync.Mutex

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

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

func main() {
  var wg sync.WaitGroup
  for i := 0; i < 1000; i++ {
    wg.Add(1)
    go increment(&wg)
  }
  wg.Wait()

  fmt.Println("Counter:", counter)
}
```

`mutex.Lock()`用于加锁，`mutex.Unlock()`用于解锁。这确保了`counter`的并发访问是安全的，避免了竞态条件。

需要注意的是，在使用互斥锁时，要确保在临界区内的代码执行时间较短，以减小锁的持有时间，从而提高程序的并发性能。过长的锁持有时间可能导致其他线程被阻塞，降低并发性。

Mutex有几种模式

mutex有两种模式：**normal** 和 **starvation**

1. 正常模式

在正常模式中，锁的获取是非公平的，即等待锁的 Goroutine 不保证按照先来先服务（FIFO）的顺序获得锁。新到来的 Goroutine 有可能在等待时间较长的 Goroutine 之前获得锁。

2. 饥饿模式：

在饥饿模式中，系统保证等待锁的 Goroutine 按照一定的公平原则获得锁，避免某些 Goroutine 长时间无法获取锁的情况。

Mutex有几种状态

- mutexLocked — 表示互斥锁的锁定状态;
- mutexWoken — 表示从正常模式被从唤醒;
- mutexStarving — 当前的互斥锁进入饥饿状态;
- waitersCount — 当前互斥锁上等待的 Goroutine 个数;

无缓冲的 channel 和有缓冲的 channel 的区别?

对于无缓冲区channel：

发送的数据如果没有被接收方接收，那么**发送方阻塞；**如果一直接收不到发送方的数据，**接收方阻塞**；

有缓冲的channel：

发送方在缓冲区满的时候阻塞，接收方不阻塞；接收方在缓冲区为空的时候阻塞，发送方不阻塞。

Go什么时候发生阻塞？阻塞时调度器会怎么做。

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

其它回答：

- **channel阻塞**：当goroutine读写channel发生阻塞时，会调用gopark函数，该G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就会脱离当前M，此时P会更新自己的状态为Psyscall，M与G相互绑定，进行系统调用。结束以后，若该P状态还是Psyscall，则直接关联该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上运行的时间超过10ms时候，或者P处于Psyscall状态过长等情况就会调用retake函数，触发新的调度。
- **主动让出**：由于是协作式调度，该G会主动让出当前的P（通过GoSched），更新状态为Grunnable，该P会调度队列中的G运行。

goroutine什么情况会发生内存泄漏？如何避免。

在Go中内存泄露分为暂时性内存泄露和永久性内存泄露。

暂时性内存泄露

- 获取长字符串中的一段导致长字符串未释放
- 获取长slice中的一段导致长slice未释放
- 在长slice新建slice导致泄漏

string相比切片少了一个容量的cap字段，可以把string当成一个只读的切片类型。获取长string或者切片中的一段内容，由于新生成的对象和老的string或者切片共用一个内存空间，会导致老的string和切片资源暂时得不到释放，造成短暂的内存泄漏

go的垃圾回收机制了解吗?

Go垃圾回收

Go1.3之前采用**标记清除法**， Go1.3之后采用**三色标记法**，Go1.8采用**三色标记法+混合写屏障**。

1. 标记清除法

初始版本的Go语言使用了一个基于标记-清扫（Mark-Sweep）算法的垃圾回收器。

- 在标记清除算法中，首先从根对象（如全局变量、栈中的引用等）出发，标记所有可达对象。这一过程通常使用深度优先搜索或广度优先搜索进行。标记的方式通常是将对象的标记位从未标记改为已标记。所有的可达对象都被标记为“活动”或“存活”。
- 在清扫阶段，遍历整个堆内存，将未被标记的对象视为垃圾，即不再被引用。所有未被标记的对象都将被回收，它们的内存将被释放，以便后续的内存分配。
- 标记清除算法执行完清扫阶段后，可能会产生内存碎片，即一些被回收的内存空间可能是不连续的。为了解决这个问题，一些实现中可能会进行内存碎片整理。
- 标记清除算法的主要优势是能够回收不再使用的内存，但它也有一些缺点。其中一个主要的缺点是清扫阶段可能会引起一定程度的停顿，因为在这个阶段需要遍历整个堆内存。另外，由于标记清除算法只关注“存活”和“垃圾”两种状态，不涉及内存分配的具体位置，可能导致内存碎片的产生。

2. 三色标记法

- 三色标记：将对象分为三种颜色：白色、灰色、和黑色。初始时，所有对象都被标记为白色，表示它们都是未被访问的垃圾对象。
- 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。

3. 三色标记法+混合写屏障

这种方法有一个缺陷，如果对象的引用被用户修改了，那么之前的标记就无效了。因此Go采用了**写屏障技术**，当对象新增或者更新会将其着色为灰色。

一次完整的GC分为四个阶段：

1. 准备标记（需要STW），开启写屏障。
2. 开始标记
3. 标记结束（STW），关闭写屏障
4. 清理（并发）

基于插入写屏障和删除写屏障在结束时需要STW来重新扫描栈，带来性能瓶颈。**混合写屏障**分为以下四步：

1. GC开始时，将栈上的全部对象标记为黑色（不需要二次扫描，无需STW）；
2. GC期间，任何栈上创建的新对象均为黑色
3. 被删除引用的对象标记为灰色
4. 被添加引用的对象标记为灰色

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。

Go语言中GC的流程是什么

Go1.14 版本以 STW 为界限，可以将 GC 划分为五个阶段:

- GCMark 标记准备阶段，为并发标记做准备工作，启动写屏障
- STWGCMark 扫描标记阶段，与赋值器并发执行，写屏障开启并发
- 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
- 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
- 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 tool trace 等工具

- 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。
- 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。
- 需要时，增大 GOGC 的值，降低 GC 的运行频率。

GMP（重要）

GMP 协程调度模型详解

GMP调度

GMP 指的是 Go 的运行时系统（Runtime）中的三个关键组件：Goroutine、M（Machine）、P（Processor）。

1. **Goroutine：**
    - Goroutine 是 Go 语言中的轻量级线程，它由 Go 运行时管理。Goroutines 是并发执行的基本单位，相比于传统的线程，它们更轻量，消耗更少的资源，并由运行时系统调度。在 Go 中，你可以创建成千上万个 Goroutine，并且它们可以非常高效地运行。
2. **M（Machine）：**
    - M 表示调度器的线程，它负责将 Goroutines 映射到真正的操作系统线程上。在运行时系统中，有一个

全局的 M 列表，每个 M 负责调度 Goroutines。当一个 Goroutine 需要执行时，它会被分配给一个 M，并在该 M 的线程上运行。M 的数量可以根据系统的负载动态调整。

3. **P（Processor）：**
    - P 表示处理器，它是用于执行 Goroutines 的上下文。P 可以看作是调度上下文，它保存了 Goroutines 的执行状态、调度队列等信息。P 的数量也是可以动态调整的，它不是直接与物理处理器核心对应的，而是与运行时系统中的 Goroutines 数目和负载情况有关。

GMP 模型的工作原理如下：

- 当一个 Goroutine 被创建时，它会被放入一个 P 的本地队列。
- 当 P 的本地队列满了，或者某个 Goroutine 长时间没有被调度执行时，P 会尝试从全局队列中获取 Goroutine。
- 如果全局队列也为空，P 会从其他 P 的本地队列中偷取一些 Goroutines，以保证尽可能多地利用所有的处理器。
- M 的数量决定了同时并发执行的 Goroutine 数目。如果某个 M 阻塞（比如在系统调用中），它的工作会被其他 M 接管。

Go 中的内存逃逸现象是什么?

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。

存逃逸的情况可能发生在以下几种情况：

1、 当在函数内部创建一个局部变量，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。

```
func createPointer() *int {
    x := 42
    return &x // x 的内存逃逸
}
```

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。

```
func sendData(ch chan<- *int) {
    x := 42
    ch <- &x // x 的内存逃逸
}
```

3、如果在函数内部使用 `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。

```
func createWithNew() *int {
    x := new(int) // x 的内存逃逸
    return x
}
```

CAP 理论，为什么不能同时满足

CAP 理论是分布式系统设计中的三个基本属性，它们分别是一致性（Consistency）、可用性（Availability）、和分区容错性（Partition Tolerance）。CAP 理论由计算机科学家 Eric Brewer 在2000年提出。

1. **一致性（Consistency）：**
    - 一致性要求系统在所有节点上的数据是一致的。即，如果在一个节点上修改了数据，那么其他节点应该立即看到这个修改。这意味着在任何时刻，不同节点上的数据应该保持一致。
2. **可用性（Availability）：**
    - 可用性要求系统能够对用户的请求做出响应，即使在出现节点故障的情况下仍然保持可用。可用性意味着系统在出现故障时仍然能够提供服务，尽管可能是部分服务。
3. **分区容错性（Partition Tolerance）：**
    - 分区容错性是指系统在面对网络分区的情况下仍能够正常工作。即，当节点之间的网络出现故障或无法通信时，系统仍能够保持一致性和可用性。

CAP 理论提出的是在分布式系统中这三个属性不能同时被满足。这是由于在分布式系统中，网络的不确定性和延迟会导致无法同时满足一致性、可用性和分区容错性。

Go Web

你有使用过哪些Go的Web框架？介绍一下它们。

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

- 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。
- 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。这使得实现日志记录、身份验证、错误处理等功能变得非常简单。
- 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，可以方便地构建HTTP响应。

说一下 Gin 的拦截器的原理

在 Gin 中，拦截器通常称为中间件（Middleware）。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或后处理逻辑。Gin 的中间件机制基于 Go 的函数闭包和`gin.Context`的特性。

Gin 的中间件是通过在路由定义中添加中间件函数来实现的，这些中间件函数会在请求到达路由处理函数之前被执行。

1. 中间件函数

中间件是一个函数，它接受一个 `gin.Context` 对象作为参数，并执行一些逻辑。中间件可以在处理函数之前或之后修改请求或响应。

```
func MyMiddleware(c *gin.Context) {
    // 在处理函数之前执行的逻辑
    fmt.Println("Middleware: Before handling request")

    // 执行下一个中间件或处理函数
    c.Next()

    // 在处理函数之后执行的逻辑
    fmt.Println("Middleware: After handling request")
}
```

2. **注册中间件：**

在 Gin 中，通过 `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，可以对整个路由组或单个路由生效。

```
r := gin.New()

// 注册中间件
r.Use(MyMiddleware)

// 定义路由
r.GET("/hello", func(c *gin.Context) {
    c.JSON(200, gin.H{"message": "Hello, Gin!"})
})
```

上述例子中的 `MyMiddleware` 就是一个简单的中间件，它会在处理 `/hello` 路由的请求之前和之后输出一些信息。

3. **中间件链：**

可以通过在 `Use` 方法中添加多个中间件函数，形成中间件链。中间件链中的中间件按照添加的顺序依次执行。

```
r := gin.New()

// 中间件链
r.Use(Middleware1, Middleware2, Middleware3)

// 定义路由
r.GET("/hello", func(c *gin.Context) {
    c.JSON(200, gin.H{"message": "Hello, Gin!"})
})
```

在上述例子中，`Middleware1`、`Middleware2`、`Middleware3` 将会按照它们添加的顺序执行。

4. **中间件的执行顺序：**

中间件的执行顺序非常重要，因为它们可能会相互影响。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()`，后续中间件和处理函数将不会被执行。

```go
func MyMiddleware(c *gin.Context) {
    fmt.Println("Middleware: Before handling request")

    // 如果不调用 c.Next()，后续中间件和处理函数将不会执行
    // c.Next()

    fmt.Println("Middleware: After handling request")
}
```

说一下 Gin 的路由怎么实现的

路由的实现是通过 `gin.Engine` 类型来管理的，而该类型实现了 `http.Handler` 接口，因此可以直接用作 `http.ListenAndServe` 的参数。Gin 的路由包括基本的路由、参数路由、组路由等

介绍一下Go中的context包的作用

`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， `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` ，也可以使用 `WithDeadline` 、 `WithTimeout` 、 `WithCancel` 或 `WithValue` 创建的修改副本替换它，听起来有点绕，其实总结起就是一句话： **`context` 的作用就是在不同的 `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。**

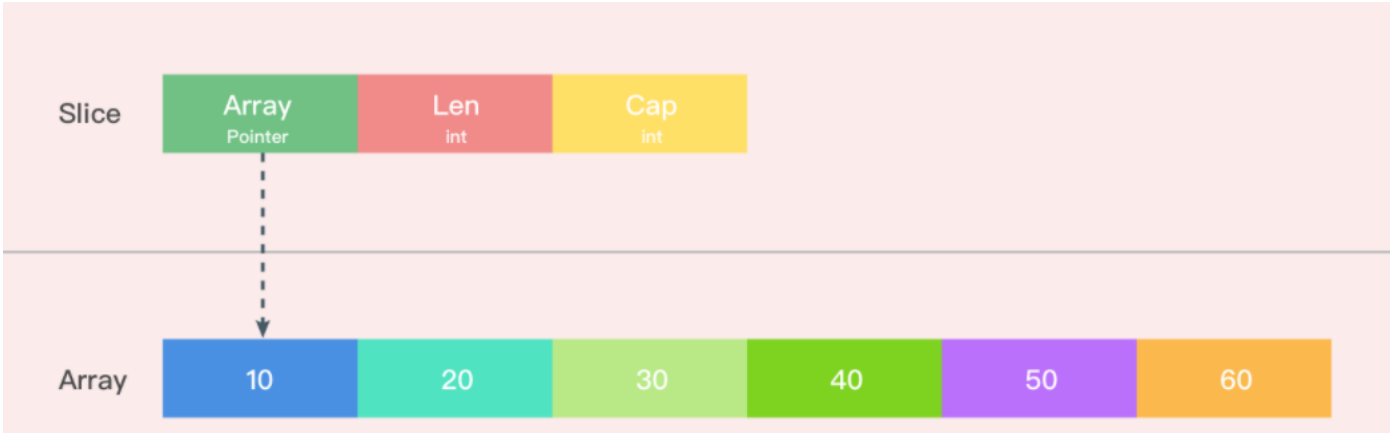
关于context原理，可以参看：小白也能看懂的context包详解：从入门到精通

第四版题目

简述slice的底层原理，和数组的区别是什么

slice内部的是通过指针引用一个底层数组，是对数组一个连续片段的引用，另外还有长度和容量两个变量

其数据结构如下：



Slice是可变长度的数组，其长度是基于底层数组的长度的，如果底层数组的长度不足以满足需求，可以进行扩容，扩容其实就是另外开一个数组把当前切片的内容copy过去， 扩容策略简单来说是以下规则：

如果新申请的容量大于两倍的旧容量，那么最终容量就是新申请的容量大小

如果不大于两倍旧容量，则判断旧容量是否小于1024，如果是则最终容量是旧容量的两倍

如果旧容量大于等于1024，就从旧容量大小开始每次增加现有旧容量的四分之一，直到满足需求为止

slice和数组的区别

- 声明时数组要指定长度或者是写 ...， 而slice方括号中为空
- 数组声明后可以不进行初始化，其内元素已经为默认零值，而slice需要初始化才能使用，不初始化时为空
- 函数传参时参数为数组是传的一个数组的copy，改变形参不会对原数组产生影响；而参数为slice时因为slice本身结构中包含了数组指针，因此改变slice形参可以改变底层的数组，同时在传slice时要注意slice的扩容问题

channel的发送和接收操作有哪些基本特性?

1. **对同一个通道，发送操作之间是互斥的，接收操作之间也是互斥的。**

即同一时刻，go的runtime只会执行对同一个通道的任意个发送操作中的某一个，直到这个发送的元素值被完全复制进该通道之后，其他发送操作才可能被执行。接收操作类似。

2. **发送操作和接收操作中对元素值的处理都是不可分割的**

即发送操作和接收操作都是原子的。例如接收操作时元素值从通道移动到外界，这个移动操作包含了两步，第一步是生成正在通道中的这个元素值的副本，并准备给到接收方，第二步是删除在通道中的这个元素值，这两个操作会一起完成。类似于innodb的事务机制。

这样既是为了保证通道中元素值的完整性，也是为了保证通道操作的唯一性

3. **发送操作在完全完成之前会被阻塞，接收操作也是。**

类似于接收操作，发送操作也是包括了复制元素值和放置副本到通道内部两个步骤。在这两个步骤完全完成之前，发起这个操作的那句代码会一直阻塞在那里，在通道完成发送操作之后，runtime系统会通知这句代码所在的goroutine，解除阻塞，以使它去争取继续运行代码的机会。如此阻塞代码也是为了实现操作的互斥和元素值的完整。

扩展

1. **发送操作和接收操作在什么时候可能会被长时间阻塞?**

对于缓冲通道来说，如果通道已满，则对它的所有发送操作都将被阻塞，直到通道中有元素值被接收走，此时通道会通知阻塞队列的首个goroutine，通知顺序是公平的

相对的，如果通道已空，那么对它的所有接收操作会被阻塞，直到通道中有新的元素值出现。

对于非缓冲通道，无论是发送还是接收操作，一开始执行就会被阻塞，直到配对的操作也开始执行，才会继续。可以说非缓冲通道就是在用同步的方式传递数据。且用非缓冲通道时，数据并不会用通道作中转。

如果错误操作也会造成长时间阻塞，最典型的就是对值为nil（即未初始化）的通道进行操作。

2. **发送操作和接收操作什么时候会引发panic?**

对一个已经关闭的通道做发送操作会引发panic，但对已经关闭的通道可以进行接收操作

对一个已经关闭的通道进行关闭操作会引发panic。

defer底层原理

1、每次defer语句在执行的时候，都会将函数进行"压栈"，函数参数会被拷贝下来。当外层函数退出时，defer函数会按照定义的顺序逆序执行 。如果defer执行的函数为nil，那么会在最终调用函数中产生panic。

2、为什么defer要按照定义的顺序逆序执行

后面定义的函数可能会依赖前面的资源，所以要先执行。如果前面先执行，释放掉这个依赖，那后面的函数就找不到它的依赖了。

3、defer函数定义时，对外部变量的引用方式有两种

分别是函数参数以及作为闭包引用。

在作为函数参数的时候，在defer定义时就把值传递给defer，并被缓存起来。

如果是作为闭包引用，则会在defer真正调用的时候，根据整个上下文去确定当前的值。

4、defer后面的语句在执行的时候，函数调用的参数会被保存起来，也就是复制一份。

在真正执行的时候，实际上用到的是复制的变量，也就是说，如果这个变量是一个"值类型"，那他就和定义的时候是一致的，如果是一个"引用"，那么就可能和定义的时候的值不一致

defer配合recover

recover(异常捕获)可以让程序在引发panic的时候不会崩溃退出。

在引发panic的时候，panic会停掉当前正在执行的程序，但是，在这之前，它会有序的执行完当前goroutine的defer列表中的语句。

所以我们通常在defer里面挂一个recover，防止程序直接挂掉，类似于try...catch，但绝对不能像try...catch这样使用，因为panic的作用不是为了抓异常。recover函数只在defer的上下文中才有效，如果直接调用recover，会返回nil

interface常见问题：

接口就是一种约定

接口分为侵入式和非侵入式，类必须明确表示自己实现了某个接口

侵入式和非侵入式的区别

1、侵入式：

你的代码里已经嵌入了别的代码，这些代码可能是你引入过的框架，也可能是你通过接口继承得来的，比如：java中的继承，必须显示的表明我要继承那个接口，这样你就可以拥有侵入代码的一些功能。所以我们就称这段代码是侵入式代码。

优点：

通过侵入代码与你的代码结合可以更好的利用侵入代码提供给的功能。

缺点：

框架外代码就不能使用了，不利于代码复用。依赖太多重构代码太痛苦了。

2、非侵入式（没有依赖，自主研发）：

正好与侵入式相反，你的代码没有引入别的包或框架，完完全全是自主开发。比如go中的接口，不需要显示的继承接口，只需要实现接口的所有方法就叫实现了该接口，即便该接口删掉了，也不会影响我，所有go语言的接口数非侵入式接口；再如Python所崇尚的鸭子类型。

优点：

代码可复用，方便移植。非侵入式也体现了代码的设计原则：高内聚，低耦合

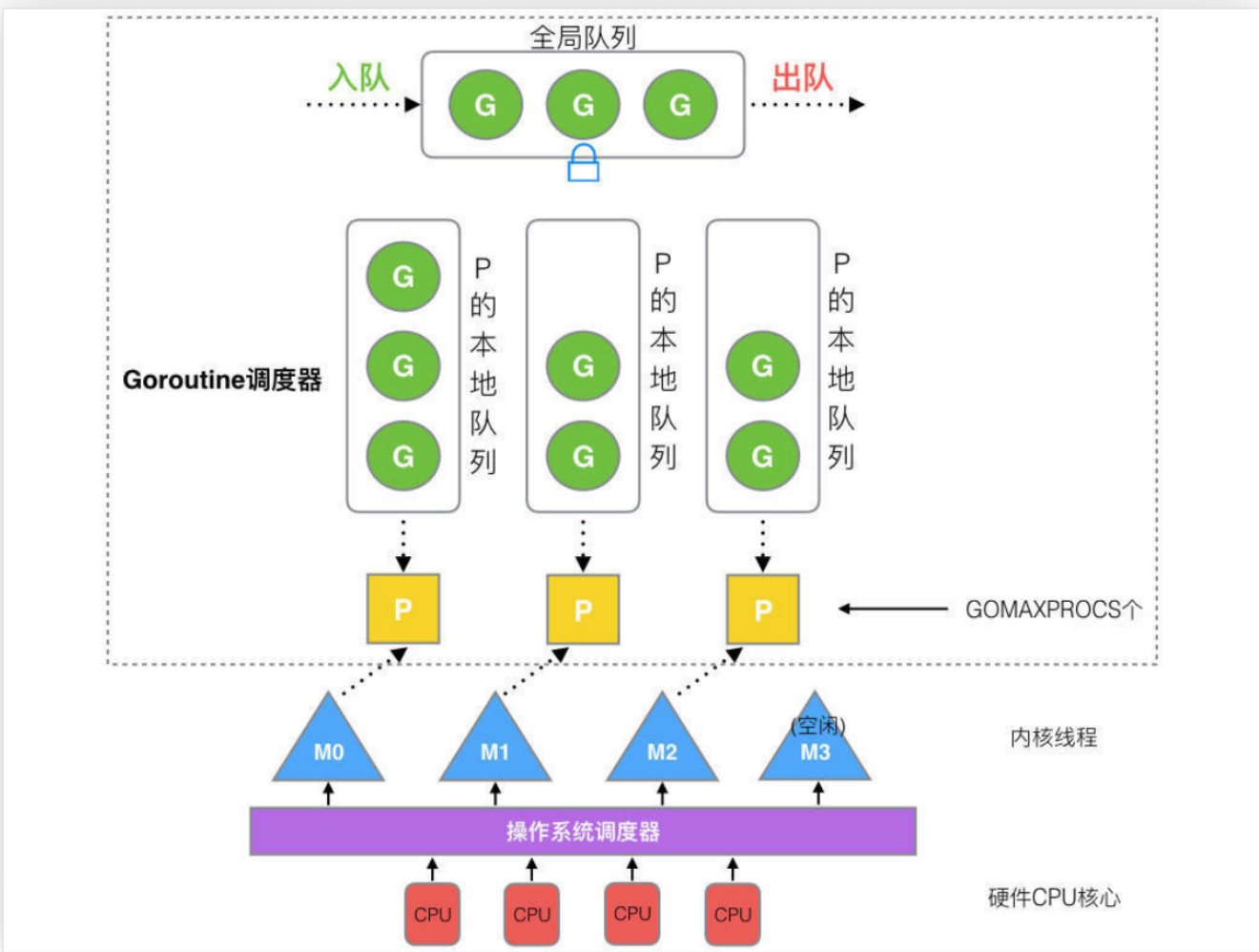
缺点：

无法复用框架提供的代码和功能

简述GMP模型和它的优点

GMP模型中：

- G表示goroutine
- M表示Thread
- P表示Processor：Processor包含了运行goroutine所需要的资源，如果线程想要运行goroutine则必须要先获取processor，P中还包含了可运行的G队列，其中线程是执行goroutine的实体，processor的功能是把goroutine调度到工作线程上去执行



GMP模型中有如下几种结构：

- 全局队列：存放等待运行的G
- P的本地队列：每个P所拥有的队列，也是存放等待运行的G，有最大个数的限制，新创建一个G优先加入到P的本地队列中，如果本地队列已满就放入全局队列中顺便将本地队列中一半的P放到全局队列
- P：所有的P都在程序启动时即创建，并保存在一个数组中，最多有GOMAXPROCS个
- M：线程想要运行G就得获取P，优先从P的本地队列中获取，如果P的本地队列为空，就会尝试从全局队列中拿一批G放入本地队列，或者从其他的P中偷一半放到自己P的本地队列中

GMP模型的优点

- 复用线程：避免频繁的创建销毁线程
    - work stealing策略：当本线程绑定的P本地队列中无可运行的G时，会尝试从其他P那里偷G来运行，而不是销毁本线程
    - hand off机制：当本线程因为执行某个G发生系统调用而阻塞时，会将绑定的P释放，将P转移给其他空闲的M去执行
- 多核并行：GOMAXPROCS设置P的数量，因此最多可有这么多个线程分布在多个cpu上同时执行
- 抢占：在其他协程中要等待一个协程主动让出cpu才会让下一个协程执行，而go中一个goroutine最多占用cpu10ms，防止其他goroutine被饿死
- 全局队列：当work stealing策略失效时，会从全局队列中获取G来执行

goroutine与线程的区别

1、使用方面：

（1）goroutine比线程更加轻量级，可以轻松创建十万、百万，不用担心资源问题

（2）goroutine与channel搭配使用，能够更加方便的实现高并发

2、实现方面：

（1）从资源上讲

1. 线程栈的内存大小一般固定为2MB
2. goroutine栈内存是可变的，初始的时候一般为2KB，最大可以扩大到1GB

（2）从调度上讲

1. 线程的调度由OS的内核完成
2. goroutine调度由自身的调度器完成

goroutine与线程的联系：

（1）多个goroutine绑定在同一个线程上面，按照一定的调度算法执行

goroutine调度机制

三个基本概念：MPG

1、M

代表一个线程，所有的G(goroutine)任务最终都会在M上执行

2、P（Processor）

1. 代表一个处理器，每个运行的M都必须绑定一个P。P的个数是GOMAXPOCS，最大为256，在程序启动时固定，一般不去修改。
2. GOMAXPOCS默认值是当前电脑的核心数，单核CPU就只能设置为1，如果设置>1，在GOMAXPOCS函数中也会被修改为1。
3. M和P的个数不一定一样多，M>=P，每一个P都会保存本地的G任务队列，另外还有一个全局的G任务队列。G任务队列可以认为线程池中的线程队列。

3、G（Goroutine）

1. 代表一个goroutine对象，每次go调用的时候都会创建一个G对象

goroutine调度流程

带了张图，便于理解

1、启动一个goroutine

也就是创建一个G对象，然后加入到本地队列或者全局队列中

2、查找是否有空闲的P

如果没有就直接返回
如果有，就用系统API创建一个M（线程）

3、由这个刚创建的M循环执行能找到的G任务

4、G任务执行的循序

- 先从本地队列找，本地没有找到
- 就从全局队列找，如果还没有找到
- 就去其他P中找

5、所有的G任务的执行是按照go的调用顺序执行的

6、如果一个系统调用或者G任务执行的时间太长，就会一直占用这个线程

（1）在启动的时候，会专门创建一个线程sysmon，用来监控和管理，在内部挨个循环

（2）sysmon主要执行任务（中断G任务）

1. 记录所有P的G任务并用schedtick变量计数，该变量在每执行一个G任务之后递增
2. 如果schedtick一直没有递增，说明这个P一直在执行同一个任务
3. 如果持续超过10ms，就在这个G任务的栈信息加一个标记
4. G任务在执行的时候，会检查这个标记，然后中断自己，把自己添加到队列的末尾，执行下一个G

（3）G任务的恢复

1. 中断的时候将寄存器中栈的信息保存到自己G对象里面
2. 当两次轮到自己执行的时候，将自己保存的栈信息复制到寄存器里面，这样就可以接着上一次运行

goroutine是**按照抢占式进行调度**的，一个goroutine最多执行10ms就会换下一个

goroutine在什么情况下会被挂起呢?

goroutine被挂起也就是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;
- IO阻塞;

主goroutine与其他goroutine有什么不同

类似于一个进程总会有一个主线程，每一个独立的go程序在运行时也总会有一个主goroutine，主goroutine是自动启用而不需要手动操作的，每条go语句（启用一个goroutine的语句）一般都会携带一个函数调用，这个调用的函数被称为go函数，而主goroutine的go函数就是作为程序入口的main函数

从main函数的角度来理解主goroutine，则主goroutine就是程序运行的主程序，其他goroutine执行的程序是被异步调用的，同时主goroutine的结束也就意味着整个进程的结束。

扩展

```go
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                go func() {
                        fmt.Println(i)
                }()
        }
}
```

问：这段代码执行后会输出什么？

这段代码在主goroutine中进行了十次go语句调用，也就是启用了十个goroutine（因为执行一条go语句时go的runtime总是会优先从存放有空闲G的队列中获取一个G，没有空闲的情况下才会创建新的G，所以叫启用goroutine更合适）来打印出i的值。

需要注意的是go函数真正被执行的时间一定总是滞后于所属的go语句被执行的时间的（因为GMP调度器需要按先进先出的顺序来执行G），而go语句本身执行完毕后如果不加干涉则不会等待其go函数的执行，会立刻去执行后面的语句，所以for循环会很快的执行完，此时的那些go函数很可能还没有执行，此时i的值已经为10

另外当for语句执行完毕后主goroutine便结束了，则那些还未被执行的go函数将不会继续执行，即不会输出内容

所以对于以上代码，绝大多数情况下不会有任何输出，也可能会输出乱序的0到9或是10个10

GC（垃圾回收）原理 1.5版本

三色标记法

1、概念

（1）白色：代表最终需要清理的对象内存块
（2）灰色：待处理的内存块
（3）黑色：活跃的内存块

2、流程

（1）起初将所有对象都置为白色

（2）扫描出所有的可达（可以搜寻到的）对象，也就是还在使用的，不需要清理的对象，标记为灰色，放入待处理队列

（3）从队列中提取灰色对象，将其引用对象标记为灰色放入队列，将自身标记为黑色

（4）有专有的锁监视对象内存修改

（5）在完成全部的扫描和标记工作之后，剩余的只有黑色和白色，分别代表活跃对象与回收对象

（6）清理所有的白色对象

简述Go的垃圾回收机制

go目前使用的垃圾回收机制是三色标记法配合写屏障和辅助GC

三色标记法是对标记回收算法的改进:

1. **初始阶段**所有对象都是**白色**
2. 从root根出发**扫描根对象**,将它们引用到的对象都标记为**灰色**,其中**root区域主要是当前程序运行到的栈和全局数据区域,是实时使用到的内存**
3. 将**灰色对象标记为黑色**,分析该灰色对象是否引用了其他对象,如果有引用其他对象,就将引用的对象标记为灰色
4. **不断分析灰色对象,直到灰色对象队列为空,此时白色对象即为垃圾,进行回收**

在内存管理中,allocBits记录了每块内存的分配情况,而gcmarkBits记录了每块内存的回收标记情况,在标记阶段会对每块内存进行标记,有对象引用的标记为1,没有的标记为0,结束标记后,将allocBits指向gcmarkBits,则有标记的才是存活的内存块,这样就完成了内存回收

进行垃圾回收需要进行STW,如果STW时间过长对于应用执行来说是灾难性的,因此为了缩短STW的时间,golang引入了**写屏障和辅助GC**

写屏障是让GC和应用程序并发执行的手段,可以有效减少STW的时间

辅助GC是为了防止GC过程中内存分配的速度过快,因此会在GC过程中让mutator线程并发执行,协助GC执行一部分回收工作

GC触发机制有:

1. **内存分配量达到阈值**:每次内存分配前都会检查当前内存分配量是否达到阈值,如果达到则触发GC, 阈值=上次GC时的内存分配量 * 内存增长率
2. **定时触发GC**:默认情况下两分钟触发一次GC,可由runtime中的参数声明
3. **手动触发GC**:可以在代码中通过使用runtime.GC()来手动触发

select实现机制

1、锁定scase中所有channel

2、按照随机顺序检测scase中的channel是否ready

（1）如果case可读，读取channel中的数据
（2）如果case可写，写入channel
（3）如果都没准备好，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对应的case index

5、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

协程优势及其通信方式

协程相当于是用户态的线程

进程切换消耗资源很大，且进程间通信较复杂，于是有了线程

每个线程都有自己的堆栈和寄存器，并共享所属进程内的其他资源，因此可以方便地实现线程切换和通信，但是也由于多个线程共享地址空间，任何一个线程出错时，同进程内的所有进程都会崩溃

但是线程也难以实现高并发，因为：

1. **线程消耗的内存还是很多**，在linux系统中高达8MB，同时为了解决线程申请堆内存时互相竞争的问题，每个线程预先在这个空间内预分配了64MB作为堆内存池，因此没有足够的内存去开启十几万的线程实现并发
2. **线程切换耗时**：线程的切换是由内核控制的，因此当线程繁多的时候，线程间的切换会消耗cpu很多的计算能力

而**协程使本来由内核实现的切换工作，交给了用户态的代码完成**

通常创建协程时，会从进程的堆上分配一段内存作为协程的栈，线程的栈有8M，而协程的栈只有几十K

每个协程都有独立的栈，在go语言中，运行时系统会帮助自动创建和销毁系统线程，而对应的用户级协程是架设在线程之上的，用户级协程的创建销毁和调度都完全由程序实现和处理，而不用经过操作系统去做，速度会很快，很容易控制和灵活

因此总结下来，协程的优势有：

1. **节省cpu,避免系统内核级的线程频繁切换造成的cpu资源浪费**
2. **节约内存**
3. **稳定性**
4. **开发效率,协程是合作式的,可以方便地将一些操作异步化**

Go中协程的通信方式

可以通过共享内存(变量)加锁的方式来进行通信,但是维护成本较高

官方推荐**通过channel进行通信**:

channel的主要功能是:

1. **作为队列存储数据**
2. **阻塞和唤醒goroutine**

select:

select搭配channel使用,其机制是监听多个channel,每一个case都是一个事件,一旦某个事件就绪(chan没有堵塞),就会从这些就绪事件中随机选择一个去执行,default用于所有chan都堵塞的情况执行

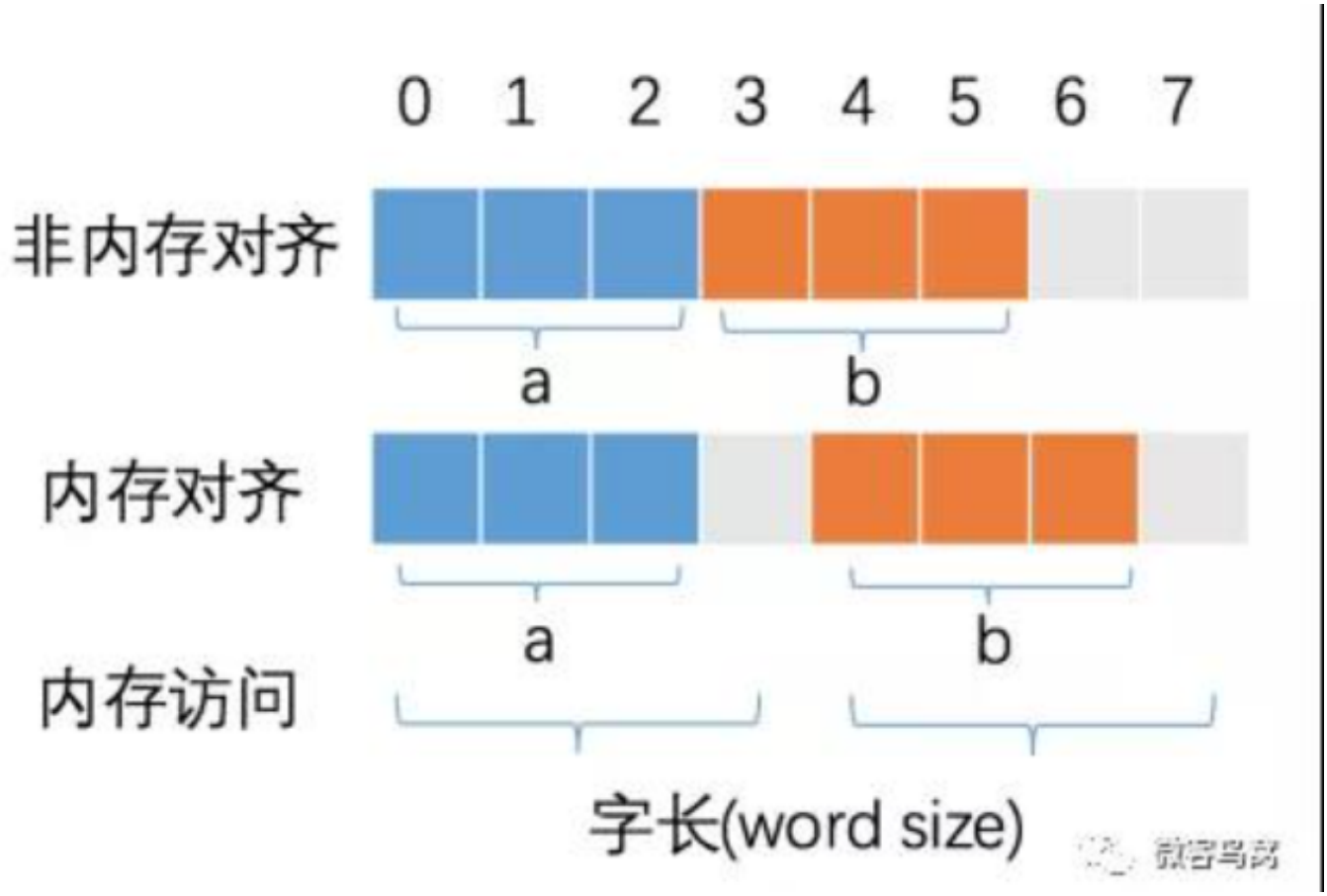
使用channel进行控制子goroutine的机制可以总结为:

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的内存对齐

CPU 访问内存时，并不是逐个字节访问，而是以字长（word size）为单位访问。比如 32 位的 CPU ，字长为 4 字节，那么 CPU 访问内存的单位也是 4 字节。

CPU 始终以字长访问内存，如果不进行内存对齐，很可能增加 CPU 访问内存的次数，例如：



变量 a、b 各占据 3 字节的空间，内存对齐后，a、b 占据 4 字节空间，CPU 读取 b 变量的值只需要进行一次内存访问。如果不进行内存对齐，CPU 读取 b 变量的值需要进行 2 次内存访问。第一次访问得到 b 变量的第 1 个字节，第二次访问得到 b 变量的后两个字节。

也可以看到，内存对齐对实现变量的原子性操作也是有好处的，每次内存访问是原子的，如果变量的大小不超过字长，那么内存对齐后，对该变量的访问就是原子的，这个特性在并发场景下至关重要。

简言之：合理的内存对齐可以提高内存读写的性能，并且便于实现变量操作的原子性。