看这一篇就够 - Go 葵花宝典

看这一篇就够 - Go 葵花宝典

一、前言

本文意在对 GoLang 面试题进行归纳总结，也欢迎读者进行评论或者留言分享面试题，希望读者看后能够具备“吊打面试官”的能力。本文中点到为止的知识点望读者可以根据关键词自行查漏补缺。

本文档会长期进行维护更新，建议读者收藏防止迷路。还有就是：知识是温故知新的，常回头看看，定会有收获！

话不多说直入主题！！！

二、内容

基础篇

1. 与其他语言相比，使用 Go 有什么好处？

- 与其他作为学术实验开始的语言不同，Go 代码的设计是务实的。每个功能和语法决策都旨在让程序员的生活更轻松。
- Golang 针对并发进行了优化，并且在规模上运行良好。

2. Go 程序中的包是什么？

包 （pkg） 是 Go 工作区中包含 Go 源文件或其他包的目录。源文件中的每个函数、变量和类型都存储在链接包中。每个 Go 源文件都属于一个包，该包在文件顶部使用以下命令声明：

```
1   package <packagename>
```

您可以使用以下方法导入和导出包以重用导出的函数或类型：

```
1   import <packagename>
```

Golang 的标准包是 fmt，其中包含格式化和打印功能，如 Println（）。

3. Go 支持什么形式的类型转换？将整数转换为浮点数。

Go 支持显式类型转换以满足其严格的类型要求。

```
i := 55 //int
j := 67.8 //float64
sum := i + int(j) //j is converted to int
```

4. 什么是 Goroutine？你如何停止它？

一个 Goroutine 是一个函数或方法执行同时旁边其他任何够程采用了特殊的 Goroutine 线程，这里叫做协程。协程比标准线程更轻量级，大多数 Golang 程序同时使用数千个 Goroutine。

要创建 Goroutine，请 go 在函数声明之前添加关键字。

```
go func(x, y, z)
```

您可以通过向 Goroutine 发送一个信号通道来停止它。Goroutines 只能在被告知检查时响应信号，因此您需要在逻辑位置（例如 for 循环顶部）包含检查。

```
package main

func main() {
    quit := make(chan bool)
    go func() {
        for {
            select {
                case <- quit: return
                default: // ...
            }
        }
    }()
    // ...
    quit <- true
 }
```

5. 如何在运行时检查变量类型？

类型开关（Type Switch）是在运行时检查变量类型的最佳方式。类型开关按类型而不是值来评估变量。每个 Switch 至少包含一个 case 用作条件语句，如果没有一个 case 为真，则执行 default。

6. Go 两个接口之间可以存在什么关系？

- 如果两个接口有相同的方法列表，那么他们就是等价的，可以相互赋值。
- 如果接口 A 的方法列表是接口 B 的方法列表的子集，那么接口 B 可以赋值给接口 A。接口查询是否成功，要在运行期才能够确定。

7. Go 当中同步锁有什么特点？作用是什么

当一个 Goroutine（协程）获得了 Mutex 后，其他 Goroutine（协程）就只能乖乖的等待，除非该 Goroutine 释放了该 Mutex。RWMutex 在读锁占用的情况下，会阻止写，但不阻止读 RWMutex。 在写锁占用情况下，会阻止任何其他 Goroutine（无论读和写）进来，整个锁相当于由该 Goroutine 独占同步锁的作用是保证资源在使用时的独有性，不会因为并发而导致数据错乱，保证系统的稳定性。

8. Go 语言当中 Channel（通道）有什么特点，需要注意什么？

如果给一个 nil 的 channel 发送数据，会造成永远阻塞。

如果从一个 nil 的 channel 中接收数据，也会造成永久阻塞。

给一个已经关闭的 channel 发送数据，会引起 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。

9. Go 语言当中 Channel 缓冲有什么特点？

无缓冲的 channel 是同步的，而有缓冲的 channel 是非同步的。

10. Go 语言中 cap 函数可以作用于哪些内容？

可以作用于的类型有：

- array（数组）
- slice（切片）
- channel（通道）

11. Go Convey 是什么？一般用来做什么？

- goconvey 是一个支持 Golang 的单元测试框架。
- go convey 能够自动监控文件修改并启动测试，并可以将测试结果实时输出到 Web 界面 。
- go convey 提供了丰富的断言简化测试用例的编写。

12. Go 语言当中 new 的作用是什么？

new 创建一个该类型的实例，并且返回指向该实例的指针。

new 函数是内建函数，函数定义：

```
1   func new(Type) *Type
```

- 使用 new 函数来分配空间。
- 传递给 new 函数的是一个类型，而不是一个值 。
- 返回值是指向这个新分配的地址的指针。

13. Go 语言中 make 的作用是什么?

make 的作用是为：slice、map、chan 的初始化然后返回引用。

make 函数是内建函数，函数定义：

```
1   func make(Type, size IntegerType) Type
```

make（T， args）函数的目的和 new（T）不同，仅仅用于创建 slice、map、channel 而且返回类型是实例。

14. golang 中 make 和 new 的区别?

共同点：

- 给变量分配内存。

不同点：

- 作用变量类型不同，new 给 string，int 和数组分配内存，make 给切片，map，channel 分配内存；
- 返回类型不一样，new 返回指向变量的指针，make 返回变量本身；
- new 分配的空间被清零。make 分配空间后，会进行初始化；
- 分配的位置，在堆上还是在栈上？Go 语言会在两个地方给变量分配内存，虽然 Go 也是可以通过 new 来给变量分配内存，但是分配的这块内存，可能在堆上，也可能在栈上。从性能的角度出发，在栈上分配内存和在堆上分配内存，性能差异是非常大的。因此一个变量是在堆上分配内存，还是在栈上分配内存，是需要编译器经过逃逸分析才能得出结论。make 需要对对象进行初始化所以是在堆上。

15. Printf（），Sprintf（），FprintF（） 都是格式化输出，有什么不同?

虽然这三个函数，都是格式化输出，但是输出的目标不一样。

Printf（） 是标准输出，一般是屏幕，也可以重定向。

Sprintf（） 是把格式化字符串输出到指定的字符串中。

Fprintf（） 是把格式化字符串输出到文件中。

16. Go 语言当中数组和切片的区别是什么？

- 数组：

数组固定长度。数组长度是数组类型的一部分，所以[3]int 和[4]int 是两种不同的数组类型数组需要指定大小，不指定也会根据初始化，自动推算出大小，大小不可改变。数组是通过值传递的。

- 切片：

切片可以改变长度。切片是轻量级的数据结构，三个属性，指针，长度，容量，不需要指定大小切片是地址传递（引用传递）可以通过数组来初始化，也可以通过内置函数 make（） 来初始化，初始化的时候 len=cap，然后进行扩容。

17. Go 语言当中值传递和地址传递（引用传递）如何运用？有什么区别？举例说明

值传递只会把参数的值复制一份放进对应的函数，两个变量的地址不同，不可相互修改。

地址传递（引用传递）会将变量本身传入对应的函数，在函数中可以对该变量进行值内容的修改。

18. Go 语言当中数组和切片在传递的时候的区别是什么？

数组是值传递。

切片看上去像是引用传递，但其实是值传递 。

19. Go 语言是如何实现切片扩容的？

```
func main() {
    arr := make([]int, 0)
    for i := 0; i < 2000; i++ {
        fmt.Println("len 为", len(arr), "cap 为", cap(arr))
        arr = append(arr, i)
    }
}

// 我们可以看下结果
依次是 0,1,2,4,8,16,32,64,128,256,512,1024
但到了 1024 之后，就变成了 1024,1280,1696,2304 每次都是扩容了四分之一左右。
```

20. defer 的执行顺序是什么？defer 的作用和特点是什么？

defer 的作用是：

你只需要在调用普通函数或方法前加上关键字 defer，就完成了 defer 所需要的语法。当 defer 语句被执行时，跟在 defer 后面的函数会被延迟执行。直到包含该 defer 语句的函数执行完毕时，

defer 后的函数才会被执行，不论包含 defer 语句的函数是通过 return 正常结束，还是由于 panic 导致的异常结束。你可以在一个函数中执行多条 defer 语句，它们的执行顺序与声明顺序相反。

defer 的常用场景：

- defer 语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、加锁、释放锁。
- 通过 defer 机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资源被释放。
- 释放资源的 defer 应该直接跟在请求资源的语句后。

21. Go 的 defer 底层数据结构?

每个 defer 语句都对应一个_defer 实例，多个实例使用指针连接起来形成一个单链表，保存在 gotoutine 数据结构中，每次插入_defer 实例，均插入到链表的头部，函数结束再一次从头部取出，从而形成后进先出的效果。

22. Golang Slice 的底层实现?

切片是基于数组实现的，它的底层是数组，它自己本身非常小，可以理解为对底层数组的抽象。因为基于数组实现，所以它的底层的内存是连续分配的，效率非常高，还可以通过索引获得数据。

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。

切片对象非常小，是因为它是只有 3 个字段的数据结构：

- 指向底层数组的指针
- 切片的长度
- 切片的容量

23. Golang Slice 的扩容机制，有什么注意点?

Go 中切片扩容的策略是这样的：

- 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。

24. 扩容前后的 Slice 是否相同?

情况一：

原数组还有容量可以扩容（实际容量没有填充完），这种情况下，扩容以后的数组还是指向原来的数组，对一个切片的操作可能影响多个指针指向相同地址的 Slice。

情况二：

原来数组的容量已经达到了最大值，再想扩容，Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append（） 操作。这种情况丝毫不影响原数组。

要复制一个 Slice，最好使用 Copy 函数。

25. Golang 的参数传递、引用类型

Go 语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。因为拷贝的内容有时候是非引用类型（int、string、struct 等这些），这样就在函数中就无法修改原内容数据；有的是引用类型（指针、map、slice、chan 等这 些），这样就可以修改原内容数据。

26. Golang Map 底层实现

Golang 中 map 的底层实现是一个散列表，因此实现 map 的过程实际上就是实现散表的过程。在这个散列表中，主要出现的结构体有两个，一个叫 hmap（a header for a go map），一个叫 bmap（a bucket for a Go map，通常叫其 bucket）。

用链表来解决冲突 ，出现冲突时，不是每一个 key 都申请一个结构通过链表串起来，而是以 bmap 为最小粒度挂载，一个 bmap 可以放 8 个 kv。

在哈希函数的选择上，会在程序启动时，检测 cpu 是否支持 aes，如果支持，则使用 aes hash，否则使用 memhash。每个 map 的底层结构是 hmap，是有若干个结构为 bmap 的 bucket 组成的数组。每个 bucket 底层都采用链表结构。

```
type hmap struct {
    count     int                // 元素个数
    flags     uint8
    B         uint8              // 扩容常量相关字段B是buckets数组的长度的对数 2^B
    noverflow uint16             // 溢出的bucket个数
    hash0     uint32             // hash seed
    buckets    unsafe.Pointer    // buckets 数组指针
    oldbuckets unsafe.Pointer    // 结构扩容的时候用于赋值的buckets数组
    nevacuate  uintptr           // 搬迁进度
    extra *mapextra              // 用于扩容的指针
}
```

27. Golang Map 如何扩容

a. 双倍扩容：扩容采取了一种称为“渐进式”的方式，原有的 key 并不会一次性搬迁完毕，每次最多只会搬迁 2 个 bucket。

b. 等量扩容：重新排列，极端情况下，重新排列也解决不了，map 存储就会蜕变成链表，性能大大降低，此时哈希因子 hash0 的设置，可以降低此类极端场景的发生。

c. 装载因子超过阈值，源码里定义的阈值是 6.5。

d. overflow 的 bucket 数量过多 map 的 bucket 定位和 key 的定位高八位用于定位 bucket，低八位用于定位 key，快速试错后再进行完整对比

28. Golang Map 查找

Go 语言中 map 采用的是哈希查找表，由一个 key 通过哈希函数得到哈希值，64 位系统中就生成一个 64bit 的哈希值，由这个哈希值将 key 对应存到不同的桶 （bucket）中，当有多个哈希映射到相同的的桶中时，使用链表解决哈希冲突。

细节：key 经过 hash 后共 64 位，根据 hmap 中 B 的值，计算它到底要落在哪个桶时，桶的数量为 2^B，如 B=5，那么用 64 位最后 5 位表示第几号桶，在用 hash 值 的高 8 位确定在 bucket 中的存储位置，当前 bmap 中的 bucket 未找到，则查询对应的 overflow bucket，对应位置有数据则对比完整的哈希值，确定是否是要查找的数据。如果当前 map 处于数据搬移状态，则优先从 oldbuckets 查找。

29. Go的原生map中删除元素，内存会自动释放吗?

- 如果删除的元素是值类型，如int，float，bool，string以及数组和struct，map的内存不会自动释放；
- 如果删除的元素是引用类型，如指针，slice，map，chan等，map的内存会自动释放，但释放的内存是子元素应用类型的内存占用；
- 将map设置为nil后，内存被回收；

30. slices能作为map类型的key吗?

在golang规范中，可比较的类型都可以作为map key。

不能作为map key 的类型包括：

- slices
- maps
- functions

31. 介绍一下 Channel

Go 语言中，不通过共享内存来通信，而通过通信来实现内存共享。Go 的 CSP（Communicating Sequential Process）并发模型，中文可以叫做通信顺序进程，是通过 goroutine 和 channel 来实现的。

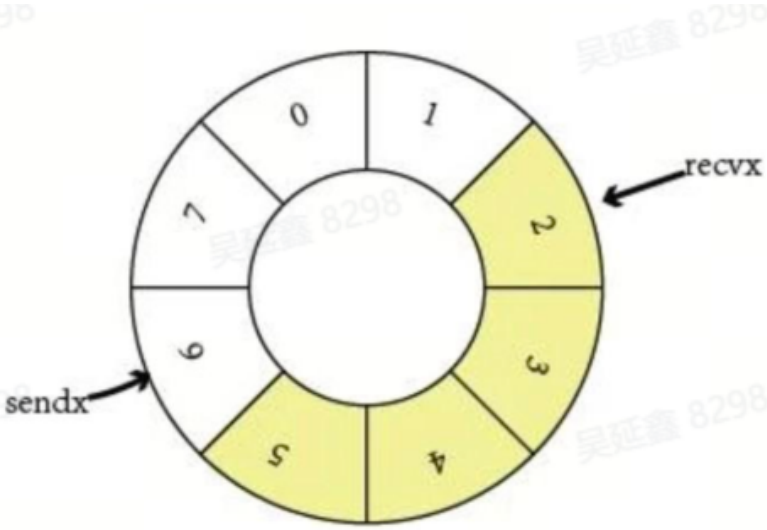
channel 收发遵循先进先出 FIFO 的原则。分为有缓冲区和无缓冲区，channel 中包括 buffer、sendx 和 recvx 收发的位置（ring buffer 记录实现）、sendq、recv。当 channel 因为缓冲区不足而阻塞了队列，则使用双向链表存储。

32. Channel 是线程安全的吗?

channel 是线程安全的，原因是 channel 内部实现了锁的机制。

33. Channel 的 ring buffer 实现

channel 中使用了 ring buffer（环形缓冲区） 来缓存写入的数据。ring buffer 有很多好处，而且非常适合用来实现 FIFO 式的固定长度队列。在 channel 中，ring buffer 的实现如下：



上图展示的是一个缓冲区为 8 的 channel buffer，recvx 指向最早被读取的数据，sendx 指向再次写入时插入的位置。

34. 当Channel通道被close后，读会带来什么问题?

读取没问题，但是写入就会有问题。

```go
package main

import ("fmt")

func main() {
    ch := make(chan int, 2)// 向通道发送数据
    ch <- 1ch <- 2// 关闭通道
    close(ch)// 安全地从通道读取数据
    val, ok := <-chfmt.Println(val, ok) // 输出: 1 true
    val, ok = <-chfmt.Println(val, ok) // 输出: 2 true
    // 当通道中没有数据时，从已关闭的通道读取
    val, ok = <-chfmt.Println(val, ok) // 输出: 0 false
    // 再次尝试向已关闭的通道发送数据将导致 panic
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic:", r)
        }
    }()
```

```
19        ch <- 3 // 这将引发 panic
20    }
```

35. for range 的时候它的地址会发生变化么?

在 for a,b := range c {} 遍历中， a 和 b 在内存中只会存在一份，即之后每次循环时遍历到的数据都是以值覆盖的方式赋给 a 和 b，a，b 的内存地址始终不变。由于有这个特性，for 循环里面如果开协程，不要直接把 a 或者 b 的地址传给协程。解决办法：在每次循环时，创建一个临时变量。

36. golang 中解析 tag 是怎么实现的？反射原理是什么?

Go 中解析的 tag 是通过反射实现的，反射是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力或动态知道给定数据对象的类型和结构，并有机会修改它。反射将接口变量转换成反射对象 Type 和 Value；反射可以通过反射对象 Value 还原成原先的接口变量；反射可以用来修改一个变量的值，前提是这个值可以被修改；tag是啥：结构体支持标记，name string json:name-field 就是 json:name-field 这部分

gorm json yaml gRPC protobuf gin.Bind()都是通过反射来实现的。

```
1   type User struct {
2           name string `json:name-field`
3           age  int
4   }
5   func main() {
6           user := &User{"John Doe The Fourth", 20}
7
8           field, ok := reflect.TypeOf(user).Elem().FieldByName("name")
9           if !ok {
10                  panic("Field not found")
11          }
12          fmt.Println(getStructTag(field))
13  }
14
15  func getStructTag(f reflect.StructField) string {
16          return string(f.Tag)
17  }
```

37. 调用函数传入结构体时，应该传值还是指针?

Go 的函数参数传递都是值传递。所谓值传递：指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。参数传递还有引用传递，所谓引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数

因为 Go 里面的 map，slice，chan 是引用类型。变量区分值类型和引用类型。所谓值类型：变量和变量的值存在同一个位置。所谓引用类型：变量和变量的值是不同的位置，变量的值存储的是对值的引用。但并不是 map，slice，chan 的所有的变量在函数内都能被修改，不同数据类型的底层存储结构和实现可能不太一样，情况也就不一样。

38. 讲讲 Go 的 select 底层数据结构和一些特性？

go 的 select 为 golang 提供了多路 IO 复用机制，和其他 IO 复用一样，用于检测是否有读写事件是否 ready。linux 的系统 IO 模型有 select，poll，epoll，go 的 select 和 linux 系统 select 非常相似。

select 结构组成主要是由 case 语句和执行的函数组成，select 实现的多路复用是：每个线程或者进程都先到注册和接受的 channel（装置）注册，然后阻塞，然后只有一个线程在运输，当注册的线程和进程准备好数据后，装置会根据注册的信息得到相应的数据。

select 的特性：

- select 操作至少要有一个 case 语句，出现读写 nil 的 channel 该分支会忽略，在 nil 的 channel 上操作则会报错。
- select 仅支持管道，而且是单协程操作。
- 每个 case 语句仅能处理一个管道，要么读要么写。
- 多个 case 语句的执行顺序是随机的。
- 存在 default 语句，select 将不会阻塞，但是存在 default 会影响性能。

39. 能介绍下 rune 类型吗？

相当int32

golang中的字符串底层实现是通过byte数组的，中文字符在unicode下占2个字节，在utf-8编码下占3个字节，而golang默认编码正好是utf-8。

byte 等同于int8，常用来处理ascii字符。

rune 等同于int32，常用来处理unicode或utf-8字符。

40. context 结构是什么样的？context 使用场景和用途？

Context通常被称为上下文，在go中，理解为goroutine的运行状态、现场，存在上下层goroutine context的传递，上层goroutine会把context传递给下层goroutine。

Go 的 Context 的数据结构包含 Deadline，Done，Err，Value，Deadline 方法返回一个 time.Time，表示当前 Context 应该结束的时间，ok 则表示有结束时间，Done 方法当 Context 被取消或者超时时候返回的一个 close 的 channel，告诉给 context 相关的函数要停止当前工作然后返回了，Err 表示 context 被取消的原因，Value 方法表示 context 实现共享数据存储的地方，是协程安全的。

主要应用场景：

- 上下文控制；
- 多个 goroutine 之间的数据交互等；
- 超时控制：到某个时间点超时，过多久超时。

场景	介绍
超时处理	通过使用 context 可以方便地设置超时时间，在超时后自动终止协程
终止协程	通过使用 cancel() 方法，协程可以很方便地终止
传递数据	我们可以将数据写入 context，在不同协程间传递数据

41. Go 多返回值怎么实现的?

Go 传参和返回值是通过 FP+offset 实现，并且存储在调用函数的栈帧中。FP 栈底寄存器，指向一个函数栈的顶部；PC 程序计数器，指向下一条执行指令；SB 指向静态数据的基指针，全局符号；SP 栈顶寄存器。

42. Go 语言中不同的类型如何比较是否相等?

像 string，int，float interface 等可以通过 reflect.DeepEqual 和等于号进行比较，像 slice，struct，map 则一般使用 reflect.DeepEqual 来检测是否相等。

43. Go 中 init 函数的特征?

一个包下可以有多个 init 函数，每个文件也可以有多个 init 函数。多个 init 函数按照它们的文件名顺序逐个初始化。应用初始化时初始化工作的顺序是，从被导入的最深层包开始进行初始化，层层递出最后到 main 包。不管包被导入多少次，包内的 init 函数只会执行一次。但包级别变量的初始化先于包内 init 函数的执行。

44. Go 中 uintptr 和 unsafe.Pointer 的区别?

unsafe.Pointer 是通用指针类型，它不能参与计算，任何类型的指针都可以转化成 unsafe.Pointer，unsafe.Pointer 可以转化成任何类型的指针，uintptr 可以转换为 unsafe.Pointer，unsafe.Pointer 可以转换为 uintptr。uintptr 是指针运算的工具，但是它不能持有指针对象（意思就是它跟指针对象不能互相转换），unsafe.Pointer 是指针对象进行运算（也就是 uintptr）的桥梁。

45. 深拷贝和浅拷贝

拷贝最简单的一种形式如下：

```
a := 648
b := a    //把a 拷贝给 b
```

深浅拷贝也和类型有关

类型	详情
引用类型	Slice、Map、Channels、Interfaces、Functions
值类型	String、Array、Int、Struct、Float、Bool

两种类型拷贝效果不同，先说我们比较熟悉的值类型。如什么是拷贝提问里易知， 若是值类型的话，在每一次拷贝后都会新申请一块空间存储值，拷贝后的两个值类型独立不影响。

以引用类型的切片(Slice)为例来讲讲深拷贝和浅拷贝：

类型	例子
深度拷贝	copy(slice1, slice2)
浅拷贝	slice1 = slice2

46. 如果 for range 同时添加数据， for range 会无限执行吗?

不会，在执行 for range 的时候实际遍历的是变量的副本，所以改变遍历的变量是不会有影响的。

47. 单引号，双引号，反引号的区别?

- 单引号，表示byte类型或rune类型，对应 uint8和int32类型，默认是 rune 类型。byte用来强调数据是raw data，而不是数字；而rune用来表示Unicode的code point。
- 双引号，是字符串，实际上是字符数组。可以用索引号访问某字节，也可以用len()函数来获取字符串所占的字节长度。
- 反引号，表示字符串字面量，但不支持任何转义序列。字面量 raw literal string 的意思是，你定义时写的啥样，它就啥样，你有换行，它就换行。你写转义字符，它也就展示转义字符。

48. 什么是数据溢出?

在使用数字类型时如果数据达到最大值，则接下来的数据将会溢出，如 uint 溢出后会从 0 开始，int 溢出后会变为负数。

```
1  package main
2
3  import (
4          "fmt"
5          "math"
6  )
7
8  func main() {
```

```
 9          var n int8 = math.MaxInt8
10          var m uint8 = math.MaxUint8
11
12          n += 2
13          m += 1
14
15          fmt.Println(n) // -127
16          fmt.Println(m) // 0
17  }
18
```

如何避免?

- 正数优先使用 uint, 范围更大
- 添加判断代码判断是否溢出

并发篇

1. Mutex 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;

2. Mutex 正常模式和饥饿模式

- 正常模式（非公平锁）

正常模式下，所有等待锁的 goroutine 按照 FIFO（先进先出）顺序等待。唤醒的 goroutine 不会直接拥有锁，而是会和新请求 goroutine 竞争锁。新请求的 goroutine 更容易抢占：因为它正在 CPU 上执行，所以刚刚唤醒的 goroutine 有很大可能在锁竞争中失败。在这种情况下，这个被唤醒的 goroutine 会加入到等待队列的前面。

- 饥饿模式（公平锁）

为了解决了等待 goroutine 队列的长尾问题，在饥饿模式下，直接由 unlock 把锁交给等待队列中排在第一位的 goroutine （队头），同时，饥饿模式下，新进来的 goroutine 不会参与抢锁也不会进入自旋状态，会直 接进入等待队列的尾部。这样很好的解决了老的 goroutine 一直抢不到锁的场 景。

饥饿模式的触发条件：当一个 goroutine 等待锁时间超过 1 毫秒时，或者当前队列 只剩下一个 goroutine 的时候，Mutex 切换到饥饿模式。

对于两种模式，正常模式下的性能是最好的，goroutine 可以连续多次获取锁，饥饿模式解决了取锁公平的问题，但是性能会下降，这其实是性能和公平的一个平衡模式。

3. Mutex 允许自旋的条件

- 锁已被占用，并且锁不处于饥饿模式。
- 积累的自旋次数小于最大自旋次数（active_spin=4）。
- CPU 核数大于 1。
- 有空闲的 P。
- 当前 Goroutine 所挂载的 P 下，本地待运行队列为空。

PS：这里补充一下 GMP 模型。

4. RWMutex 实现

通过记录 readerCount 读锁的数量来进行控制，当有一个写锁的时候，会将读锁数量设置为负数 1<<30。目的是让新进入的读锁等待之前的写锁释放通知读锁。同样的当有写锁进行抢占时，也会等待之前的读锁都释放完毕，才会开始进行后续的操作。 而等写锁释放完之后，会将值重新加上 1<<30，并通知刚才新进入的读锁（rw.readerSem），两者互相限制。

5. RWMutex 注意事项

- RWMutex 是单写多读锁，该锁可以加多个读锁或者一个写锁。
- 读锁占用的情况下会阻止写，不会阻止读，多个 Goroutine 可以同时获取读锁。
- 写锁会阻止其他 Goroutine（无论读和写）进来，整个锁由该 Goroutine 独占。
- 适用于读多写少的场景。
- RWMutex 类型变量的零值是一个未锁定状态的互斥锁。
- RWMutex 在首次被使用之后就不能再被拷贝。
- RWMutex 的读锁或写锁在未锁定状态，解锁操作都会引发 panic。
- RWMutex 的一个写锁去锁定临界区的共享资源，如果临界区的共享资源已被（读锁或写锁）锁定，这个写锁操作的 goroutine 将被阻塞直到解锁。
- RWMutex 的读锁不要用于递归调用，比较容易产生死锁。
- RWMutex 的锁定状态与特定的 goroutine 没有关联。一个 goroutine 可以 RLock（Lock），另一个 goroutine 可以 RUnlock（Unlock）。
- 写锁被解锁后，所有因操作锁定读锁而被阻塞的 goroutine 会被唤醒，并都可以成功锁定读锁。
- 读锁被解锁后，在没有被其他读锁锁定的前提下，所有因操作锁定写锁而被阻塞的 Goroutine，其中等待时间最长的一个 Goroutine 会被唤醒。

6. Cond 是什么

Cond 实现了一种条件变量，可以使用在多个 Reader 等待共享资源 ready 的场景 （如果只有一读一写，一个锁或者 channel 就搞定了） ，每个 Cond 都会关联一个 Lock（*sync.Mutex or *sync.RWMutex），当修改条件或者调用 Wait 方法时，必须加锁，保护 condition。

7. Broadcast 和 Signal 区别

```
1   func (c *Cond) Broadcast()
```

Broadcast 会唤醒所有等待 c 的 goroutine。调用 Broadcast 的时候，可以加锁，也可以不加锁。

```
1   func (c *Cond) Signal()
```

Signal 只唤醒 1 个等待 c 的 goroutine。调用 Signal 的时候，可以加锁，也可以不加锁。

8. Cond 中 Wait 使用

```
1   func (c *Cond) Wait()
```

Wait（） 会自动释放 c.L 锁，并挂起调用者的 goroutine。之后恢复执行，Wait（） 会在返回时对 c.L 加锁。

除非被 Signal 或者 Broadcast 唤醒，否则 Wait（） 不会返回。

由于 Wait（） 第一次恢复时，C.L 并没有加锁，所以当 Wait 返回时，调用者通常并不能假设条件为真。

取而代之的是，调用者应该在循环中调用 Wait。（简单来说，只要想使用 condition，就必须加锁。）

PS：这里补充下 https://bytetech.info/articles/7354949306505232403

9. WaitGroup 用法

一个 WaitGroup 对象可以等待一组协程结束。

使用方法是：

a. main 协程通过调用 wg.Add（delta int） 设置 worker 协程的个数，然后创建 worker 协程；

b. worker 协程执行结束以后，都要调用 wg.Done（）；

c. main 协程调用 wg.Wait（） 且被 block，直到所有 worker 协程全部执行结束后返回。

10. WaitGroup 实现原理

- WaitGroup 主要维护了 2 个计数器，一个是请求计数器 v，一个是等待计数器 w，二者组成一个 64bit 的值，请求计数器占高 32bit，等待计数器占低 32bit。
- 每次 Add 执行，请求计数器 v 加 1，Done 方法执行，等待计数器减 1，v 为 0 时通过信号量唤醒 Wait（）。

11. 什么是 sync.Once

- Once 可以用来执行且仅仅执行一次动作，常常用于单例对象的初始化场景。
- Once 常常用来初始化单例资源，或者并发访问只需初始化一次的共享资源，或者在测试的时候初始化一次测试资源。
- sync.Once 只暴露了一个方法 Do，你可以多次调用 Do 方法，但是只有第一 次调用 Do 方法时 f 参数才会执行，这里的 f 是一个无参数无返回值的函数。

12. 什么操作叫做原子操作

原子操作即是在程序进行过程中不能被中断的操作，针对某个值的原子操作在进行的过程中，CPU 绝不会再去进行其他的针对该值的操作。为了实现这样的严谨性，原子操作仅会由一个独立的 CPU 指令代表和完成。原子操作是无锁的，常常直接通过 CPU 指令直接实现。 事实上，其它同步技术的实现常常依赖于原子操作。

13. 原子操作和锁的区别

原子操作由底层硬件支持，而锁则由操作系统的调度器实现。 锁应当用来保护一段逻辑，对于一个变量更新的保护。 原子操作通常执行上会更有效率，并且更能利用计算机多核的优势，如果要更新的是一个复合对象，则应当使用 atomic.Value 封装好的实现。

14. 什么是 CAS

CAS 的全称为 Compare And Swap，直译就是比较交换。是一条 CPU 的原子指令，其作用是让 CPU 先进行比较两个值是否相等，然后原子地更新某个位置的值，其实现方式是给予硬件平台的汇编指令，在 intel 的 CPU 中，使用的 cmpxchg 指令，就是说 CAS 是靠硬件实现的，从而在硬件层面提升效率。

简述过程是这样：

假设包含 3 个参数内存位置（V）、预期原值（A）和新值（B）。V 表示要更新变量的值，E 表示预期值，N 表示新值。仅当 V 值等于 E 值时，才会将 V 的值设为 N，如 果 V 值和 E 值不同，则说明已经有其他线程在做更新，则当前线程什么都不做， 最后 CAS 返回当前 V 的真实值。CAS 操作时抱着乐观的态度进行的，它总是认为自己可以成功完成操作。基于这样的原理，CAS 操作即使没有锁，也可以发现其他线程对于当前线程的干扰。

15. sync.Pool 有什么用

对于很多需要重复分配、回收内存的地方，sync.Pool 是一个很好的选择。频繁地分配、回收内存会给 GC 带来一定的负担，严重的时候会引起 CPU 的毛刺 。而 sync.Pool 可以将暂时将不用的对象缓存起来，待下次需要的时候直接使用，不用再次经过内存分配，复用对象的内存，减轻 GC 的压力，提升系统的性能。

16. go语言的并发机制以及它所使用的CSP并发模型

`CSP` 模型是上个世纪七十年代提出的，不同于传统的多线程通过共享内存来通信， `CSP` 讲究的是“以通信的方式来共享内存”。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。 `CSP` 中 `channel` 是第一类对象，它不关注发送消息的实体，而关注于发送消息时使用的 `channel` 。

`Golang` 中 `channel` 是被单独创建并且可以在进程之间传递，它的通信模式类似于 `boss-worker` 模式的，一个实体通过将消息发送到 `channel` 中，然后又监听这个 `channel` 的实体处理，两个实体之间是匿名的，这个就实现实体中间的解耦，其中 `channel` 是同步的一个消息被发送到 `channel` 中，最终是一定要被另外的实体消费掉的，在实现原理上其实类似一个阻塞的消息队列。

17. 怎么限制Goroutine的数量

```
package main

import (
        "fmt"
        "runtime"
        "sync"
        "time"
)
// Pool Goroutine Pool
type Pool struct {
        queue chan int
        wg *sync.WaitGroup
}
// New 新建一个协程池
func NewPool(size int) *Pool{
        if size <=0{
                size = 1
        }
        return &Pool{
                queue:make(chan int,size),
                wg:&sync.WaitGroup{},
        }
}
```

```
24  // Add 新增一个执行
25  func (p *Pool)Add(delta int){
26          // delta为正数就添加
27          for i :=0;i<delta;i++{
28                  p.queue <-1
29          }
30          // delta为负数就减少
31          for i:=0;i>delta;i--{
32                  <-p.queue
33          }
34          p.wg.Add(delta)
35  }
36  // Done 执行完成减一
37  func (p *Pool) Done(){
38          <-p.queue
39          p.wg.Done()
40  }
41  // Wait 等待Goroutine执行完毕
42  func (p *Pool) Wait(){
43          p.wg.Wait()
44  }
45
46  func main(){
47          // 这里限制5个并发
48          pool := NewPool(5)
49          fmt.Println("the NumGoroutine begin is:",runtime.NumGoroutine())
50          for i:=0;i<20;i++{
51                  pool.Add(1)
52                  go func(i int) {
53                          time.Sleep(time.Second)
54                          fmt.Println("the NumGoroutine continue
    is:",runtime.NumGoroutine())
55                          pool.Done()
56                  }(i)
57          }
58          pool.Wait()
59          fmt.Println("the NumGoroutine done is:",runtime.NumGoroutine())
60  }
```

其中，Go的GOMAXPROCS默认值已经设置为CPU的核数，这里允许我们的Go程序充分使用机器的每一个CPU，最大程度的提高我们程序的并发性能。runtime.NumGoroutine函数在被调用后，会返回系统中的处于特定状态的Goroutine的数量。这里的特指是指Grunnable\Gruning\Gsyscall\Gwaition。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。

运行时

1. Goroutine 定义

Golang 在语言级别支持协程，称之为 Goroutine。Golang 标准库提供的所有系统调用操作（包括所有的同步 I/O 操作），都会出让 CPU 给其他 Goroutine。这让 Goroutine 的切换管理不依赖于系统的线程和进程，也不依赖于 CPU 的核心数 量，而是交给 Golang 的运行时统一调度。

2. GMP 指的是什么

G（Goroutine）：我们所说的协程，为用户级的轻量级线程，每个 Goroutine 对象中的 sched 保存着其上下文信息。

M（Machine）：对内核级线程的封装，数量对应真实的 CPU 数（真正干活的对象）。

P（Processor）：即为 G 和 M 的调度对象，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。

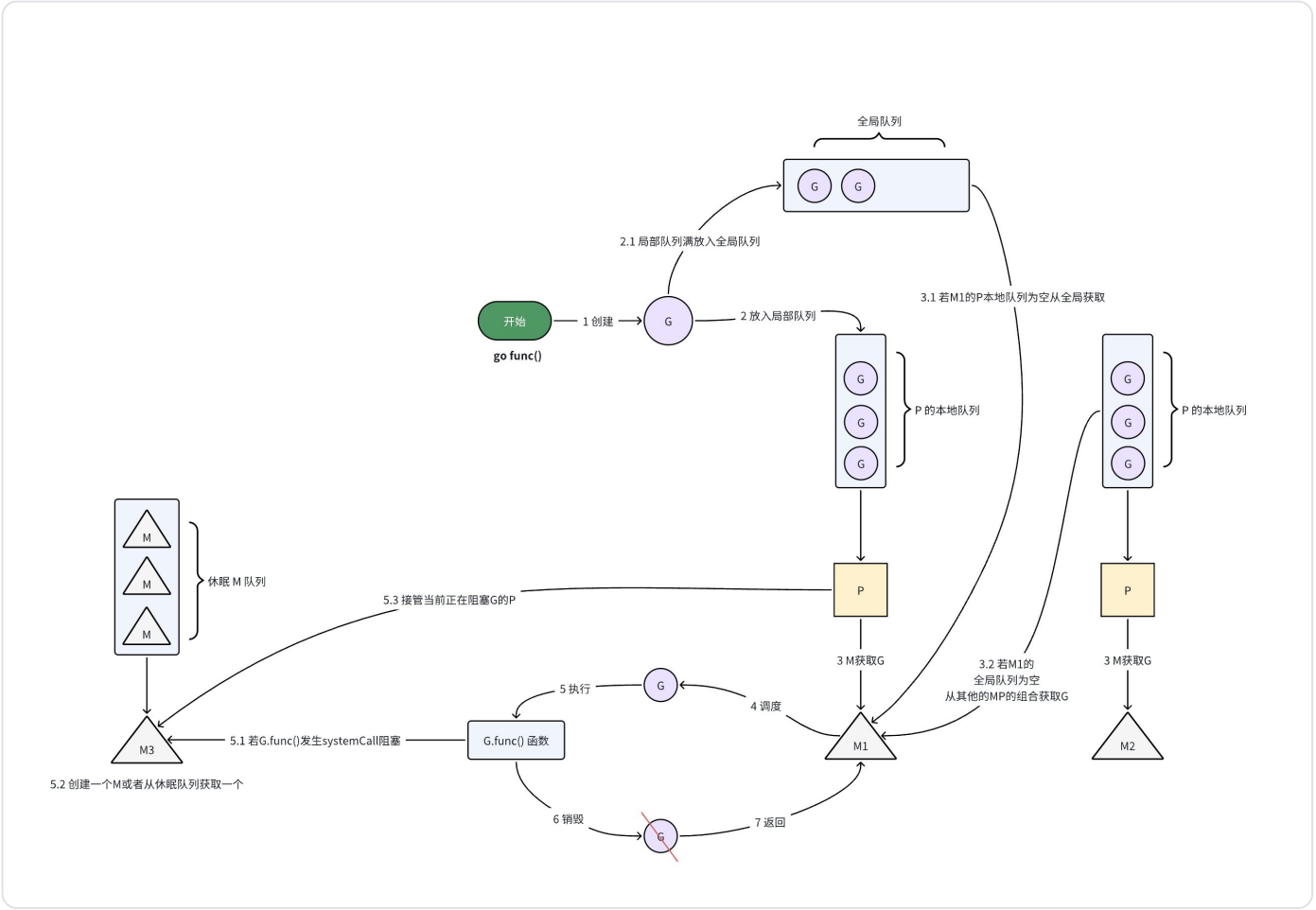
3. 1.0 之前 GM 调度模型

调度器把 G 都分配到 M 上，不同的 G 在不同的 M 并发运行时，都需要向系统申请 资源，比如堆栈内存等，因为资源是全局的，就会因为资源竞争照成很多性能损耗。为了解决这一的问题 go 从 1.1 版本引入，在运行时系统的时候加入 P 对象， 让 P 去管理这个 G 对象，M 想要运行 G，必须绑定 P，才能运行 P 所管理的 G。

GM 调度存在的问题：

- 单一全局互斥锁（Sched.Lock）和集中状态存储。
- Goroutine 传递问题（M 经常在 M 之间传递”可运行”的 goroutine）。
- 每个 M 做内存缓存，导致内存占用过高，数据局部性较差。
- 频繁 syscall 调用，导致严重的线程阻塞/解锁，加剧额外的性能损耗。

4. GMP 调度流程



- 每个 P 有个局部队列，局部队列保存待执行的 goroutine（流程 2），当 M 绑定 的 P 的的局部队列已经满了之后就会把 goroutine 放到全局队列（流程 2.1）。
- 每个 P 和一个 M 绑定，M 是真正的执行 P 中 goroutine 的实体（流程 3），M 从绑定的 P 中的局部队列获取 G 来执行。
- 当 M 绑定的 P 的局部队列为空时，M 会从全局队列获取到本地队列来执行 G （流程 3.1），当从全局队列中没有获取到可执行的 G 时候，M 会从其他 P 的局部队列中偷取 G 来执行（流程 3.2），这种从其他 P 偷的方式称为 work stealing。
- 当 G 因系统调用（syscall）阻塞时会阻塞 M，此时 P 会和 M 解绑即 handoff，并寻找新的 idle 的 M，若没有 idle 的 M 就会新建一个 M（流程 5.2）。
- 当 G 因 channel 或者 networkI/O 阻塞时，不会阻塞 M，M 会寻找其他 runnable 的 G；当阻塞的 G 恢复后会重新进入 runnable 进入 P 队列等待执行（流程 5.3）。

5. GMP 中 work stealing 机制

获取 P 本地队列，当从绑定 P 本地 runq 上找不到可执行的 g，尝试从全局链表中拿，再拿不到从 netpoll 和事件池里拿，最后会从别的 P 里偷任务。P 此时去唤醒一个 M。P 继续执行其它的程序。M 寻找是否有空闲的 P，如果有则将该 G 对象移动到它本身。接下来 M 执行一个调度循环（调用 G 对象->执行->清理线程→ 继续找新的 Goroutine 执行）。

6. GMP 中 hand off 机制

当本线程 M 因为 G 进行的系统调用阻塞时，线程释放绑定的 P，把 P 转移给其他空闲的 M 执行。

细节：当发生上下文切换时，需要对执行现场进行保护，以便下次被调度执行时进行现场恢复。Go 调度器 M 的栈保存在 G 对象上，只需要将 M 所需要的寄存器（SP、PC 等）保存到 G 对象上就可以实现现场保护。当这些寄存器数据被保护起来，就随时可以做上下文切换了，在中断之前把现场保存起来。如果此时 G 任务还没有执行完，M 可以将任务重新丢到 P 的任务队列，等待下一次被调度执行。当再次被调度执行时，M 通过访问 G 的 vdsoSP、vdsoPC 寄存器进行现场恢复 （从上次中断位置继续执行）。

7. 协作式的抢占式调度

在 1.14 版本之前，程序只能依靠 Goroutine 主动让出 CPU 资源才能触发调度。

这种方式存在问题有：

- 某些 Goroutine 可以长时间占用线程，造成其它 Goroutine 的饥饿。
- 垃圾回收需要暂停整个程序（Stop-the-world，STW），最长可能需要几分钟的时间，导致整个程序无法工作。

8. 基于信号的抢占式调度

在任何情况下，Go 运行时并行执行（注意，不是并发）的 goroutines 数量是小于等于 P 的数量的。为了提高系统的性能，P 的数量肯定不是越小越好，所以官方默认值就是 CPU 的核心数，设置的过小的话，如果一个持有 P 的 M， 由 于 P 当前执行的 G 调用了 syscall 而导致 M 被阻塞，那么此时关键点：GO 的调度器是迟钝的，它很可能什么都没做，直到 M 阻塞了相当长时间以后，才会发现有一个 P/M 被 syscall 阻塞了。然后，才会用空闲的 M 来抢这个 P。通过 sysmon 监控实现的抢占式调度，最快在 20us，最慢在 10-20ms 才会发现 有一个 M 持有 P 并阻塞了。操作系统在 1ms 内可以完成很多次线程调度（一般情况 1ms 可以完成几十次线程调度），Go 发起 IO/syscall 的时候执行该 G 的 M 会阻塞然后被 OS 调度走，P 什么也不干，sysmon 最慢要 10-20ms 才能发现这个阻塞，说不定那时候阻塞已经结束了，这样宝贵的 P 资源就这么被阻塞的 M 浪费了。

9. M 和 P 的数量问题?

p默认cpu内核数。

M与P的数量没有绝对关系，一个M阻塞，P就会去创建或者切换另一个M，所以，即使P的默认数量是1，也有可能会创建很多个M出来。

10. GMP 调度过程中存在哪些阻塞

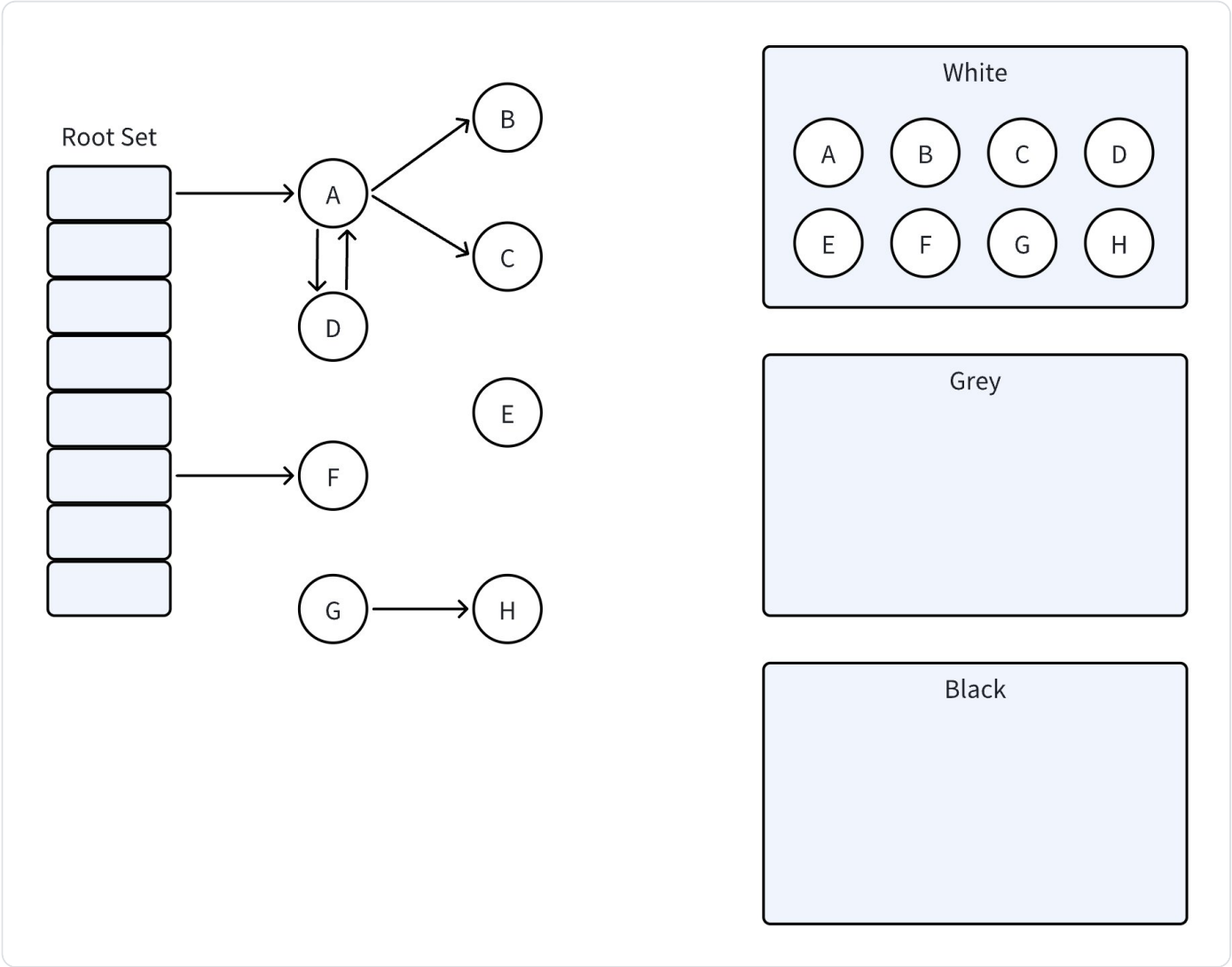
Sysmon 也叫监控线程，变动的周期性检查，好处：

- 释放闲置超过 5 分钟的 span 物理内存；
- 如果超过 2 分钟没有垃圾回收，强制执行；
- 将长时间未处理的 netpoll 添加到全局队列；

- 向长时间运行的 G 任务发出抢占调度（超过 10ms 的 g，会进行 retake）；
- 收回因 syscall 长时间阻塞的 P；

11. 三色标记原理

我们首先看一张图，大概就会对三色标记法有一个大致的了解：



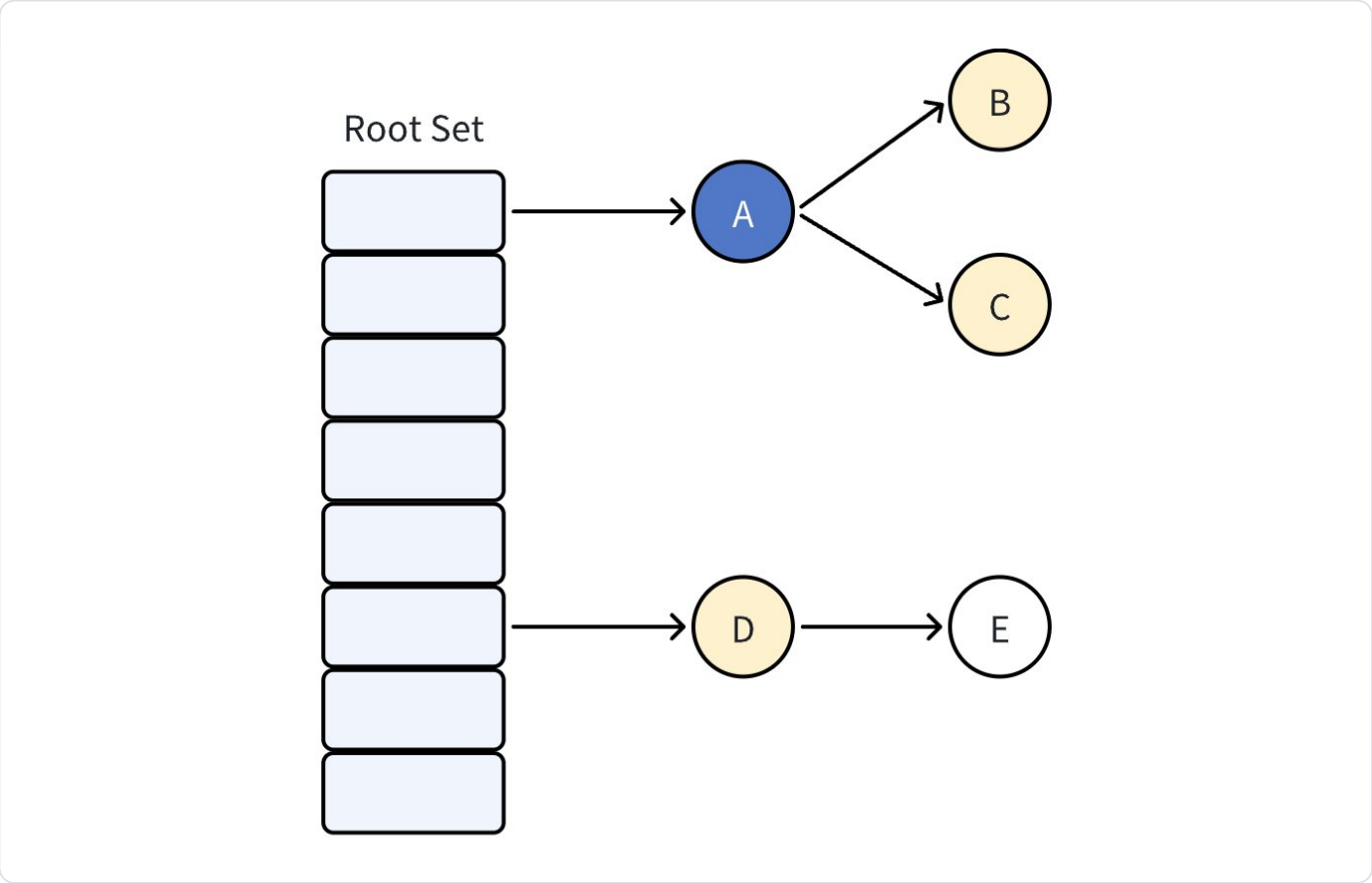
原理：
首先把所有的对象都放到白色的集合中。

- 从根节点开始遍历对象，遍历到的白色对象从白色集合中放到灰色集合中。
- 遍历灰色集合中的对象，把灰色对象引用的白色集合的对象放入到灰色集合中，同时把遍历过的灰色集合中的对象放到黑色的集合中。
- 循环步骤 3，直到灰色集合中没有对象。
- 步骤 4 结束后，白色集合中的对象就是不可达对象，也就是垃圾，进行回收。

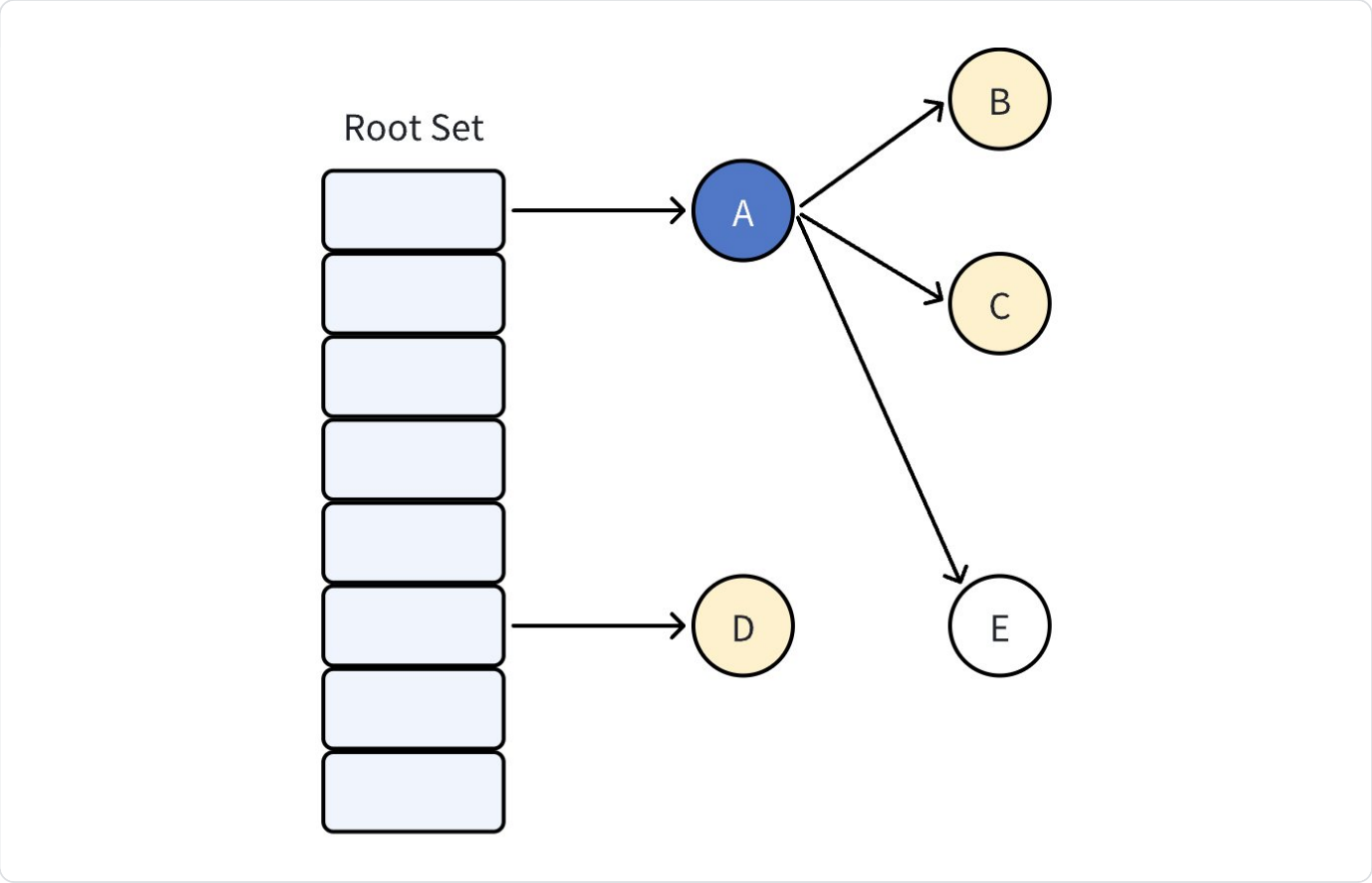
12. 写屏障

Go 在进行三色标记的时候并没有 STW，也就是说，此时的对象还是可以进行修改。

那么我们考虑一下，下面的情况：



我们在进行三色标记中扫描灰色集合中，扫描到了对象 A，并标记了对象 A 的 所有引用，这时候，开始扫描对象 D 的引用，而此时，另一个 goroutine 修改了 D->E 的引用，变成了如下图所示：



这样会不会导致 E 对象就扫描不到了，而被误认为为白色对象，也就是垃圾写屏障就是为了解决这样的问题，引入写屏障后，在上述步骤后，E 会被认为是存活的，即使后面 E 被 A 对象抛弃，E 会被在下一轮的 GC 中进行回收，这一轮 GC 中是不会对对象 E 进行回收的。

13. 插入写屏障

Go GC 在混合写屏障之前，一直是插入写屏障，由于栈赋值没有 hook 的原因，栈中没有启用写屏障，所以有 STW。

Golang 的解决方法是：只是需要在结束时启动 STW 来重新扫描栈。这个自然就会导致整个进程的赋值器卡顿。

14. 删除写屏障

Golang 没有这一步，Golang 的内存写屏障是由插入写屏障到混合写屏障过渡的。简单介绍一下，一个对象即使被删除了最后一个指向它的指针也依旧可以活过这一轮，在下一轮 GC 中才被清理掉。

15. 混合写屏障

- 混合写屏障继承了插入写屏障的优点，起始无需 STW 打快照，直接并发扫 描垃圾即可；
- 混合写屏障继承了删除写屏障的优点，赋值器是黑色赋值器，GC 期间，任何在栈上创建的新对象，均为黑色。扫描过一次就不需要扫描了，这样就消除了插入写屏障时期最后 STW 的重新扫描栈；
- 混合写屏障扫描精度继承了删除写屏障，比插入写屏障更低，随着带来的是 GC 过程全程无 STW；
- 混合写屏障扫描栈虽然没有 STW，但是扫描某一个具体的栈的时候，还是要停止这个 goroutine 赋值器的工作（针对一个 goroutine 栈来说，是暂停扫的，要么全灰，要么全黑哈，原子状态切换）。

16. GC 触发时机

> 有关GC详细内容可以查看笔者另一篇文章：
> https://bytetech.info/articles/7360889177946701876

- 主动触发：调用 runtime.GC；
- 被动触发：使用系统监控，该触发条件由 runtime.forcegcperiod 变量控制，默认为 2 分钟。当超过两分钟没有产生任何 GC 时，强制触发 GC。 使用步调（Pacing）算法，其核心思想是控制内存增长的比例。如 Go 的 GC 是一种比例 GC，下一次 GC 结束时的堆大小和上一次 GC 存活堆大小成比例。

17. go gc 是怎么实现的?

Go 的 GC 回收有三次演进过程：

- Go V1.3 之前普通标记清除（mark and sweep）方法，整体过程需要启动 STW，效率极低。
- Go V1.5 三色标记法，堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描一次栈(需要 STW)，效率普通。
- Go V1.8 三色标记法，混合写屏障机制：栈空间不启动（全部标记成黑色），堆空间启用写屏障，整个过程不要 STW，效率高。

Go1.3 之前的版本所谓标记清除是先启动 STW 暂停，然后执行标记，再执行数据回收，最后停止 STW。Go1.3 版本标记清除做了点优化，流程是：先启动 STW 暂停，然后执行标记，停止 STW，最后再执行数据回收。

Go1.5 三色标记主要是插入屏障和删除屏障，写入屏障的流程：

程序开始，全部标记为白色，1）所有的对象放到白色集合，2）遍历一次根节点，得到灰色节点，3）遍历灰色节点，将可达的对象，从白色标记灰色，遍历之后的灰色标记成黑色，4）由于并发特性，此刻外界向在堆中的对象发生添加对象，以及在栈中的对象添加对象，在堆中的对象会触发插入屏障机制，栈中的对象不触发，5）由于堆中对象插入屏障，则会把堆中黑色对象添加的白色对象改成灰色，栈中的黑色对象添加的白色对象依然是白色，6）循环第 5 步，直到没有灰色节点，7）在准备回收白色前，重新遍历扫描一次栈空间，加上 STW 暂停保护栈，防止外界干扰（有新的白色会被添加成黑色）在 STW 中，将栈中的对象一次三色标记，直到没有灰色，8）停止 STW，清除白色。至于删除写屏障，则是遍历灰色节点的时候出现可达的节点被删除，这个时候触发删除写屏障，这个可达的被删除的节点也是灰色，等循环三色标记之后，直到没有灰色节点，然后清理白色，删除写屏障会造成一个对象即使被删除了最后一个指向它的指针也依旧可以活过这一轮，在下一轮 GC 中被清理掉。

GoV1.8 混合写屏障规则是：

1）GC 开始将栈上的对象全部扫描并标记为黑色(之后不再进行第二次重复扫描，无需 STW)，2）GC 期间，任何在栈上创建的新对象，均为黑色。3）被删除的对象标记为灰色。4）被添加的对象标记为灰色。

18. Go 语言中 GC 的流程是什么?

Go 1.14 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。

19. 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。

20. 知道 golang 的内存逃逸吗？什么情况下会发生内存逃逸？

- 本该分配到栈上的变量，跑到了堆上，这就导致了内存逃逸。
- 栈是高地址到低地址，栈上的变量，函数结束后变量会跟着回收掉，不会有额外性能的开销。
- 变量从栈逃逸到堆上，如果要回收掉，需要进行 gc，那么 gc 一定会带来额外的性能开销。编程语言不断优化 gc 算法，主要目的都是为了减少 gc 带来的额外性能开销，变量一旦逃逸会导致性能开销变大。

内存逃逸的情况如下：

a. 方法内返回局部变量指针。

b. 向 channel 发送指针数据。

c. 在闭包中引用包外的值。

d. 在 slice 或 map 中存储指针。

e. 切片（扩容后）长度太大。

f. 在 interface 类型上调用方法。

21. 谈谈内存泄漏，什么情况下内存会泄漏？怎么定位排查内存泄漏问题？

go 中的内存泄漏一般都是 goroutine 泄漏，就是 goroutine 没有被关闭，或者没有添加超时控制，让 goroutine 一只处于阻塞状态，不能被 GC。

内存泄露有下面一些情况：

- 如果 goroutine 在执行时被阻塞而无法退出，就会导致 goroutine 的内存泄漏，一个 goroutine 的最低栈大小为 2KB，在高并发的场景下，对内存的消耗也是非常恐怖的。
- 互斥锁未释放或者造成死锁会造成内存泄漏。
- time.Ticker 是每隔指定的时间就会向通道内写数据。作为循环触发器，必须调用 stop 方法才会停止，从而被 GC 掉，否则会一直占用内存空间。
- 字符串的截取引发临时性的内存泄漏

```
1   func main() {
2           var str0 = "12345678901234567890"
3           str1 := str0[:10]
4   }
```

- 切片截取引起子切片内存泄漏

```
func main() {
        var s0 = []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
        s1 := s0[:3]
}
```

- 函数数组传参引发内存泄漏【如果我们在函数传参的时候用到了数组传参，且这个数组够大（我们假设数组大小为 100 万，64 位机上消耗的内存约为 800w 字节，即 8MB 内存），或者该函数短时间内被调用 N 次，那么可想而知，会消耗大量内存，对性能产生极大的影响，如果短时间内分配大量内存，而又来不及 GC，那么就会产生临时性的内存泄漏，对于高并发场景相当可怕。】

排查方式：

一般通过 pprof 是 Go 的性能分析工具，在程序运行过程中，可以记录程序的运行信息，可以是 CPU 使用情况、内存使用情况、goroutine 运行情况等，当需要性能调优或者定位 Bug 时候，这些记录的信息是相当重要。

22. Channel 分配在栈上还是堆上？哪些对象分配在堆上，哪些对象分配在栈上？

Channel 被设计用来实现协程间通信的组件，其作用域和生命周期不可能仅限于某个函数内部，所以 golang 直接将其分配在堆上。准确地说，你并不需要知道。Golang 中的变量只要被引用就一直会存活，存储在堆上还是栈上由内部实现决定而和具体的语法没有关系。

知道变量的存储位置确实和效率编程有关系。如果可能，Golang 编译器会将函数的局部变量分配到函数栈帧（stack frame）上。然而，如果编译器不能确保变量在函数 return 之后不再被引用，编译器就会将变量分配到堆上。而且，如果一个局部变量非常大，那么它也应该被分配到堆上而不是栈上。

当前情况下，如果一个变量被取地址，那么它就有可能被分配到堆上，然而，还要对这些变量做逃逸分析，如果函数 return 之后，变量不再被引用，则将其分配到栈上。

23. 对已经关闭的chan进行读写，会怎么样？为什么？

- 读**已经关闭**的 `chan` 能一直读到东西，但是读到的内容根据通道内 `关闭前` 是否有元素而不同。
  - 如果 `chan` 关闭前， `buffer` 内有元素**还未读**，会正确读到 `chan` 内的值，且返回的第二个bool值（是否读成功）为 `true` 。
  - 如果 `chan` 关闭前， `buffer` 内有元素**已经被读完**， `chan` 内无值，接下来所有接收的值都会非阻塞直接成功，返回 `channel` 元素的**零值**，但是第二个 `bool` 值一直为 `false` 。
- 写**已经关闭**的 `chan` 会 `panic` 。

24. 对未初始化的chan进行读写，会怎样？为什么？

- 对于写的情况
  - 未初始化的 `chan` 此时是等于 `nil`，当它不能阻塞的情况下，直接返回 `false`，表示写 `chan` 失败。
  - 当 `chan` 能阻塞的情况下，则直接阻塞 `gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)`，然后调用 `throw(s string)` 抛出错误，其中 `waitReasonChanSendNilChan` 就是刚刚提到的报错 `"chan send (nil chan)"`。
- 对于读的情况
  - 未初始化的 `chan` 此时是等于 `nil`，当它不能阻塞的情况下，直接返回 `false`，表示读 `chan` 失败。
  - 当 `chan` 能阻塞的情况下，则直接阻塞 `gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)`,然后调用 `throw(s string)` 抛出错误，其中 `waitReasonChanReceiveNilChan` 就是刚刚提到的报错 `"chan receive (nil chan)"`。

25. 说说uintptr和unsafe.Pointer的区别

- unsafe.Pointer只是单纯的通用指针类型，用于转换不同类型指针，它不可以参与指针运算；
- 而uintptr是用于指针运算的，GC 不把 uintptr 当指针，也就是说 uintptr 无法持有对象，uintptr 类型的目标会被回收；
- unsafe.Pointer 可以和 普通指针 进行相互转换；
- unsafe.Pointer 可以和 uintptr 进行相互转换。

举例：

```
1  package main
2
3  import (
4          "fmt"
5          "unsafe"
6  )
7
8  type W struct {
9          b int32
10         c int64
11 }
12
13 func main() {
14         var w *W = new(W)
```

```
15          //这时w的变量打印出来都是默认值0，0
16          fmt.Println(w.b,w.c)
17
18          //现在我们通过指针运算给b变量赋值为10
19          b := unsafe.Pointer(uintptr(unsafe.Pointer(w)) +
    unsafe.Offsetof(w.b))
20          *((*int)(b)) = 10
21          //此时结果就变成了10，0
22          fmt.Println(w.b,w.c)
23  }
```

- `uintptr(unsafe.Pointer(w))` 获取了 `w` 的指针 起始值 。
- `unsafe.Offsetof(w.b)` 获取 `b` 变量的 偏移量 。
- 两个 相加 就得到了 `b` 的 地址值 ，将通用指针 `Pointer` 转换成具体指针 `((*int)(b))` ，通过 `*` 符号取值，然后赋值。 `*((*int)(b))` 相当于把 `(*int)(b)` 转换成 `int` 了，最后对变量重新赋值成 `10` ，这样指针运算就完成了。

26. 介绍一下大对象小对象，为什么小对象多了会造成 gc 压力?

小于等于 32k 的对象就是小对象，其它都是大对象。一般小对象通过 mspan 分配内存；大对象则直接由 mheap 分配内存。通常小对象过多会导致 GC 三色法消耗过多的 CPU。优化思路是，减少对象分配。

小对象：如果申请小对象时，发现当前内存空间不存在空闲跨度时，将会需要调用 nextFree 方法获取新的可用的对象，可能会触发 GC 行为。

大对象：如果申请大于 32k 以上的大对象时，可能会触发 GC 行为。

27. Go语言的栈空间管理是怎么样的?

Go语言的运行环境（runtime）会在goroutine需要的时候动态地分配栈空间，而不是给每个goroutine分配固定大小的内存空间。这样就避免了需要程序员来决定栈的大小。

分块式的栈是最初 `Go` 语言组织栈的方式。当创建一个 `goroutine` 的时候，它会分配一个 `8KB` 的内存空间来给 `goroutine` 的栈使用。我们可能会考虑当这 `8KB` 的栈空间被用完的时候该怎么办?

为了处理这种情况，每个 `Go` 函数的开头都有一小段检测代码。这段代码会检查我们是否已经用完了分配的栈空间。如果是的话，它会调用 `morestack` 函数。 `morestack` 函数分配一块新的内存作为栈空间，并且在这块栈空间的底部填入各种信息（包括之前的那块栈地址）。在分配了这块新的栈空间之后，它会重试刚才造成栈空间不足的函数。这个过程叫做栈分裂（ `stack split` ）。

在新分配的栈底部，还插入了一个叫做 `lessstack` 的函数指针。这个函数还没有被调用。这样设置是为了从刚才造成栈空间不足的那个函数返回时做准备的。当我们从那个函数返回时，它会跳

转到 `lessstack`。 `lessstack` 函数会查看在栈底部存放的数据结构里的信息，然后调整栈指针（`stack pointer`）。这样就完成了从新的栈块到老的栈块的跳转。接下来，新分配的这个块栈空间就可以被释放掉了。

分块式的栈让我们能够按照需求来扩展和收缩栈的大小。 Go开发者不需要花精力去估计goroutine会用到多大的栈。创建一个新的goroutine的开销也不大。当 Go开发者不知道栈会扩展到多少大时，它也能很好的处理这种情况。

这一直是之前 `Go` 语言管理栈的的方法。但这个方法有一个问题。缩减栈空间是一个开销相对较大的操作。如果在一个循环里有栈分裂，那么它的开销就变得不可忽略了。一个函数会扩展，然后分裂栈。当它返回的时候又会释放之前分配的内存块。如果这些都发生在一个循环里的话，代价是相当大的。 这就是所谓的热分裂问题（`hot split problem`）。它是Go语言开发者选择新的栈管理方法的主要原因。新的方法叫做栈复制法（`stack copying`）。

栈复制法一开始和分块式的栈很像。当 `goroutine` 运行并用完栈空间的时候，与之前的方法一样，栈溢出检查会被触发。但是，不像之前的方法那样分配一个新的内存块并链接到老的栈内存块，新的方法会分配一个两倍大的内存块并把老的内存块内容复制到新的内存块里。这样做意味着当栈缩减回之前大小时，我们不需要做任何事情。栈的缩减没有任何代价。而且，当栈再次扩展时，运行环境也不需要再做任何事。它可以重用之前分配的空间。

栈的复制听起来很容易，但实际操作并非那么简单。存储在栈上的变量的地址可能已经被使用到。也就是说程序使用到了一些指向栈的指针。当移动栈的时候，所有指向栈里内容的指针都会变得无效。然而，指向栈内容的指针自身也必定是保存在栈上的。这是为了保证内存安全的必要条件。否则一个程序就有可能访问一段已经无效的栈空间了。

因为垃圾回收的需要，我们必须知道栈的哪些部分是被用作指针了。当我们移动栈的时候，我们可以更新栈里的指针让它们指向新的地址。所有相关的指针都会被更新。我们使用了垃圾回收的信息来复制栈，但并不是任何使用栈的函数都有这些信息。因为很大一部分运行环境是用 `C` 语言写的，很多被调用的运行环境里的函数并没有指针的信息，所以也就不能够被复制了。当遇到这种情况时，我们只能退回到分块式的栈并支付相应的开销。

这也是为什么现在运行环境的开发者正在用 `Go` 语言重写运行环境的大部分代码。无法用 `Go` 语言重写的部分（比如调度器的核心代码和垃圾回收器）会在特殊的栈上运行。这个特殊栈的大小由运行环境的开发者设置。

这些改变除了使栈复制成为可能，它也允许我们在将来实现并行垃圾回收。

另外一种不同的栈处理方式就是在虚拟内存中分配大内存段。由于物理内存只是在真正使用时才会被分配，因此看起来好似你可以分配一个大内存段并让操 作系统处理它。下面是这种方法的一些问题

首先， `32` 位系统只能支持 `4G` 字节虚拟内存，并且应用只能用到其中的 `3G` 空间。由于同时运行百万 `goroutines` 的情况并不少见，因此你很可 能用光虚拟内存，即便我们假设每个 `goroutine` 的 `stack` 只有8K。

第二，然而我们可以在 `64` 位系统中分配大内存，它依赖于过量内存使用。所谓过量使用是指当你分配的内存大小超出物理内存大小时，依赖操作系统保证 在需要时能够分配出物理内存。然而，允

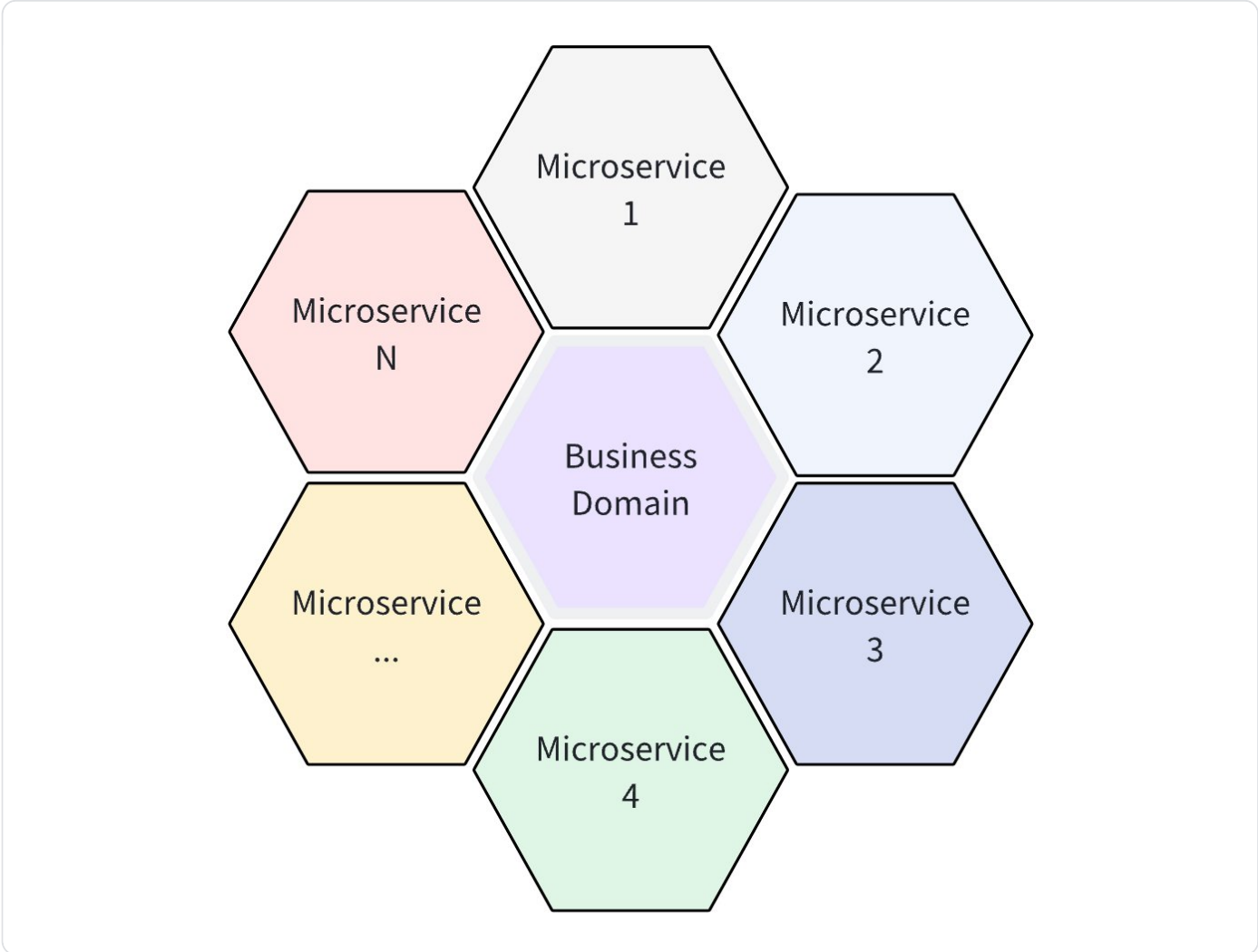
许过量使用可能会导致一些风险。由于一些进程分配了超出机器物理内存大小的内存，如果这些进程使用更多内存 时，操作系统将不得不为它们补充分配内存。这会导致操作系统将一些内存段放入磁盘缓存，这常常会增加不可预测的处理延迟。正是考虑到这个原因，一 些新系统关闭了对过量使用的支持。

微服务

1. 对微服务有何了解?

微服务，又称微服务架构，是一种架构风格，它将应用程序构建为以业务领域为模型的小型自治服务集合。

通俗地说，你必须看到蜜蜂如何通过对齐六角形蜡细胞来构建它们的蜂窝状物。他们最初从使用各种材料的小部分开始，并继续从中构建一个大型蜂箱。 这些细胞形成图案，产生坚固的结构，将蜂窝的特定部分固定在一起。 这里，每个细胞独立于另一个细胞，但它也与其他细胞相关。这意味着对一个细胞的损害不会损害其他细胞，因此，蜜蜂可以在不影响完整蜂箱的情况下重建这些细胞。



请参考上图。这里，每个六边形形状代表单独的服务组件。与蜜蜂的工作类似，每个敏捷团队都使用可用的框架和所选的技术堆栈构建单独的服务组件。 就像在蜂箱中一样，每个服务组件形成一个强大的微服务架构，以提供更好的可扩展性。此外，敏捷团队可以单独处理每个服务组件的问题，而对整个应用程序没有影响或影响最小。

2. 说说微服务架构的优势

优势	说明
独立开发	所有微服务都可以根据各自的功能轻松开发。
独立部署	根据他们所提供的服务，可以在任何应用中单独部署。
故障隔离	即使应用中的一个服务不起作用，系统仍然继续运行。
混合技术栈	可以用不同的语言和技术来构建同一应用程序的不同服务。
粒度缩放	各个组件可根据需要进行扩展，无需将所有组件融合到一起。

3. 微服务有哪些特点?

- 解耦
  - 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展。
- 组件化
  - 微服务被视为可以轻松更换和升级的独立组件。
- 业务能力
  - 微服务非常简单，专注于单一功能。
- 自治
  - 开发人员和团队可以彼此独立工作，从而提高速度。
- 持续交付
  - 通过软件创建，测试和批准的系统自动化，允许频繁发布软件。
- 责任
  - 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品。
- 分散治理
  - 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题。
- 敏捷
  - 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃。

4. 微服务架构如何运作?

微服务架构具有以下组件：

- 客户端

        - 来自不同设备的不同用户发送请求。
    - 身份提供商
        - 验证用户或客户身份并颁发安全令牌。
    - API 网关
        - 处理客户端请求。
    - 静态内容
        - 容纳系统的所有内容。
    - 管理
        - 在节点上平衡服务并识别故障。
    - 服务发现
        - 查找微服务之间通信路径的指南。
    - 网络
        - 代理服务器及其数据中心的分布式网络。
    - 远程服务
        - 启用驻留在 IT 设备网络上的远程访问信息。

5. 微服务架构的优缺点是什么？

优点	缺点
自由使用不同的技术	增加故障排除挑战
每个微服务都侧重于单一功能	由于远程调用而增加延迟
支持单个可部署单元	增加了配置和其他操作的工作量
允许经常发布软件	难以保持交易安全
确保每项服务的安全性	艰难地跨越各种便捷跟踪数据
多个服务是并行开发和部署的	难以在服务之间进行编码

6. SOA 和微服务架构之间的主要区别是什么？

SOA	微服务
遵循“尽可能多的共享”架构方法	遵循“尽可能少分享”架构方法

重要性在于“业务功能”重用	重要性在于“有界背景”的概念
它们有共同的治理和标准	它们专注于人们的合作和其他选择的自由
使用企业服务总线（ESB）进行通信	简单的消息系统
它们支持多种消息协议	它们使用轻量级协议，如 HTTP/REST 等
多线程，有更多的开销来处理 I/O	单线程，通常使用 Event Loop 功能进行非锁定 I/O 处理
最大化应用程序服务可重用性	专注于解耦
传统的关系数据库更常用	现代关系数据库更常用
系统的变化需要修改整体	系统的变化是创造一种新的服务
DevOps/Continuous Delivery 正在变得流 行，但还不是主流	专注于 DevOps/持续交付

PS：这里还需要补充 DDD（领域驱动设计）。

7. 什么是 REST / RESTful 以及它的用途是什么?

Representational State Transfer（REST）/ RESTful Web 服务是一种帮助计算机系统通过 Internet 进行通信的架构风格。这使得微服务更容易理解和实现。 微服务可以使用或不使用 RESTful API 实现，但使用 RESTful API 构建松散耦合的微服务总是更容易。

8. 分布式锁实现原理

在分析分布式锁的三种实现方式之前，先了解一下分布式锁应该具备哪些条件：

a. 在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；

b. 高可用的获取锁与释放锁；

c. 高性能的获取锁与释放锁；

d. 具备可重入特性；

e. 具备锁失效机制，防止死锁；

f. 具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

分布式的CAP理论告诉我们“任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。”所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

通常分布式锁以单独的服务方式实现，目前比较常用的分布式锁实现有三种：

- 基于数据库实现分布式锁。
- 基于缓存（redis，memcached，tair）实现分布式锁。
- 基于Zookeeper实现分布式锁。
- 基于ETCD实现分布式锁。

9. Mysql高可用方案有哪些

a. 主从复制方案

b. MMM/MHA高可用方案

c. Heartbeat/SAN高可用方案

d. Heartbeat/DRBD高可用方案

e. NDB CLUSTER高可用方案

算法

1. 使用 Channel 交替打印数字和字母

```
12AB34CD56EF78GH910IJ1112KL1314MN1516OP1718QR1920ST2122UV2324WX2526YZ2728
```

解题思路：

为了实现交替打印数字和字母，我们可以使用两个 `goroutine`，一个负责打印数字，另一个负责打印字母。我们可以使用 `channel` 来进行两个 `goroutine` 的同步。

具体的解题思路如下：

a. 创建一个 `channel`，用于数字 `goroutine` 和字母 `goroutine` 之间的通信。

b. 创建一个 `goroutine`，用于打印数字。在该 `goroutine` 中，通过一个循环不断从 `channel` 中接收数字，并打印出来。每次打印完数字后，再向 `channel` 中发送一个信号，表示数字已经打印完毕。

c. 创建一个 `goroutine`，用于打印字母。在该 `goroutine` 中，通过一个循环不断从 `channel` 中接收字母，并打印出来。每次打印完字母后，再向 `channel` 中发送一个信号，表示字母已经打印完毕。

d. 在主 `goroutine` 中，先向 `channel` 中发送一个数字信号，表示可以开始打印数字了。然后进入一个循环，每次从 `channel` 中接收一个信号，表示数字已经打印完毕，然后向 `channel` 中发送一个字母信号，表示可以开始打印字母了。如此循环，直到打印完所有数字和字母。

```
```

```go
package main

import (
        "fmt"
        "sync"
)

func printNumber(ch chan bool, wg *sync.WaitGroup) {
        defer wg.Done()
        for i := 1; i <= 28; i += 2 {
                <-ch // 等待信号，表示可以开始打印数字
                fmt.Print(i)
                fmt.Print(i + 1)
                ch <- true // 发送信号，表示数字已经打印完毕
        }
}

func printLetter(ch chan bool, wg *sync.WaitGroup) {
        defer wg.Done()
        for i := 'A'; i <= 'Z'; i += 2 {
                <-ch // 等待信号，表示可以开始打印字母
                fmt.Print(string(i))
                fmt.Print(string(i + 1))
                ch <- true // 发送信号，表示字母已经打印完毕
        }
}

func waitGroupWithTimeout(wg *sync.WaitGroup, timeout time.Duration) bool {
    ch := make(chan bool)
    go func() {
       defer close(ch)
       wg.Wait()
    }()
    select {
    case <-ch:
       return false
    case <-time.After(timeout):
       return true
    }
}

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

        var wg sync.WaitGroup
        wg.Add(2)

```

```
48          go printNumber(ch, &wg)
49          go printLetter(ch, &wg)
50
51          ch <- true // 发送一个数字信号，表示可以开始打印数字
52
53          waitGroupWithTimeout(&wg, 1*time.Second)
54          fmt.Println()
55  }
```

2. 翻转字符串

请实现一个算法，在不使用【额外数据结构和储存空间】的情况下，翻转一个给定的字符串(可以使用单个过程变量)。

给定一个string，请返回一个string，为翻转后的字符串。保证字符串的长度小于等于16。

解题思路：

一种常见的解题思路是使用双指针法。

定义两个指针，一个指向字符串的开头，一个指向字符串的末尾。然后交换两个指针所指向的字符，然后两个指针向中间移动，重复这个过程，直到两个指针相遇。

```
1   func reverseString(s string) string {
2       str := []rune(s)
3       left := 0
4       right := len(str) - 1
5
6       for left < right {
7           str[left], str[right] = str[right], str[left]
8           left++
9           right--
10      }
11
12      return string(str)
13  }
```

多种解题思路

除了使用双指针法，还可以使用递归法来翻转字符串。

```
1   func reverseString(s string) string {
2       str := []rune(s)
3       return string(reverse(str, 0, len(str)-1))
4   }
```

```
5
6   func reverse(str []rune, left, right int) []rune {
7       if left >= right {
8           return str
9       }
10
11      str[left], str[right] = str[right], str[left]
12      return reverse(str, left+1, right-1)
13  }
```

3. 判断字符串中字符是否全都不同

请实现一个算法，确定一个字符串的所有字符【是否全都不同】。这里我们要求【不允许使用额外的存储结构】。 给定一个string，请返回一个bool值，true代表所有字符全都不同，false代表存在相同的字符。保证字符串中的字符为【ASCII字符】。字符串的长度小于等于【3000】。

解题思路：

这里有几个重点，第一个是 ASCII字符， ASCII字符 字符一共有256个，其中128个是常用字符，可以在键盘上输入。128之后的是键盘上无法找到的。

然后是全部不同，也就是字符串中的字符没有重复的，再次不准使用额外的储存结构，且字符串小于等于3000。

如果允许其他额外储存结构，这个题目很好做。如果不允许的话，可以使用golang内置的方式实现。

```
1   func isUniqueString(s string) bool {
2       if strings.Count(s, "") > 3000{
3           return false
4       }
5       for _,v := range s {
6           if v > 127 {
7               return false
8           }
9           if strings.Count(s, string(v)) > 1 {
10              return false
11          }
12      }
13      return true
14  }
```

这里使用strings.Count()，可以用来判断在一个字符串中包含的另外一个字符串的数量。

```go
func isUniqueString(s string) bool {
    if strings.Count(s, "") > 3000 {
        return false
    }
    for k, v := range s {
        if v > 127 {
            return false
        }
        if strings.Index(s, string(v)) != k {
            return false
        }
    }
    return true
}
```

这里使用strings.Index() 或者 strings.LastIndex() ，用来判断指定字符串在另外一个字符串的索引位置，分别是第一次发现位置和最后发现位置。

4. 机器人坐标问题

有一个机器人，给一串指令，L左转，R右转，F前进一步，B后退一步，问最后机器人的坐标，最开始，机器人位于 (0,0)，方向为正Y。 可以输入重复指令n，比如 R2(LF) 这个等于指令 RLFLF。问最后机器人的坐标是多少?

解题思路：

这里的一个难点是解析重复指令。主要指令部分解析成功，计算坐标就简单了。

```go
package main
import (
    "unicode"
)

const (
    Left = iota
    Top
    Right
    Bottom
)

func main() {
    println(move("R2(LF)", 0, 0, Top))
}

// x: x坐标; y: y坐标; z: 方向;
```

```go
// 这里需要注意方向是相对于机器人的前后左右，不是世界坐标的前后左右，所以方向是相对的变化的。
func move(cmd string, x0 int, y0 int, z0 int) (x, y, z int) {
    x, y, z = x0, y0, z0
    repeat := 0
    repeatCmd := ""
    for _, s := range cmd {
        switch {
            case unicode.IsNumber(s):
                // int(2)=50 - '0'=2 unicode转为10进制数字
                repeat = repeat*10 + (int(s) - '0')
            case s == ')':
                for i := 0; i < repeat; i++ {
                    // 对重复的命令递归操作
                    x, y, z = move(repeatCmd, x, y, z)
                }
                repeat = 0
                repeatCmd = ""
            // 遇到需要执行重复命令的将重复命令记录下，这里2(LF)，由于先判断了2，repeat=2
            case repeat > 0 && s != '(':
                repeatCmd = repeatCmd + string(s)
            case s == 'L':
                z = (z + 1) % 4
            case s == 'R':
                z = (z - 1 + 4) % 4
            case s == 'F':
                switch {
                    // 方向是左或者右时，前进会改变x的大小
                    case z == Left || z == Right:
                        x = x - z + 1
                    // 方向是向前或者向后时，前进会改变y的大小
                    case z == Top || z == Bottom:
                        y = y - z + 2
                }
            case s == 'B':
                switch {
                    case z == Left || z == Right:
                        x = x + z - 1
                    case z == Top || z == Bottom:
                        y = y + z - 2
                }
        }
    }
    return
}
```

这里使用三个值表示机器人当前的状况，分别是：x表示x坐标，y表示y坐标，z表示当前方向。 L、R 命令会改变值z，F、B命令会改变值x、y 。 值x、y的改变还受当前的z值影响。

如果是重复指令，那么将重复次数和重复的指令存起来递归调用即可。

5. 反转链表

```go
package main

import "fmt"

type ListNode struct {
    Val  intNext *ListNode
}

// ReverseList 反转链表
func ReverseList(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        nextTemp := curr.Next // 临时保存下一个节点
        curr.Next = prev      // 反转当前节点
        prev = curr           // 前移prev指针
        curr = nextTemp       // 前移curr指针
    }
    return prev
}

func main() {
    // 创建一个简单的链表 1 -> 2 -> 3
    head := &ListNode{
        Val: 1,
        Next: &ListNode{Val: 2, Next: &ListNode{Val: 3}},
    }

    // 反转链表
    reversedHead := ReverseList(head)// 打印
    printList(reversedHead)
}

func printList(head *ListNode) {
    for head != nil {
        fmt.Println(head.Val)
        if head.Next != nil {
            fmt.Print(" -> ")
        }
```

```
        head = head.Next
    }
    fmt.Println()
}
```

6. 生产者、消费者模型

```
package main

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

const MaxItems = 5 // 生产的最大项目数

// 生产者函数，向channel写入数据
func producer(ch chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < MaxItems; i++ {
        ch <- i // 生产一个项目
        fmt.Println("Produced", i)
        time.Sleep(time.Second)
    }
}

// 消费者函数，从channel读取数据
func consumer(ch <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for item := range ch {
        fmt.Println("Consumed", item)
        time.Sleep(time.Second) // 模拟耗时操作
    }
}

func main() {
    var wg sync.WaitGroup
    ch := make(chan int, MaxItems)
    // 启动生产者 goroutine
    wg.Add(1)
    go producer(ch, &wg)

    wg.Add(1)
    go consumer(ch, &wg)
```

```
39
40      wg.Wait()
41      close(ch)
42  }
```