Go

基础

1. 逃逸分析
   1.1 逃逸分析是什么?
   1.2 逃逸分析的作用是什么?
   1.3 逃逸分析是怎么完成的?
   1.4 如何确定一个变量是否发生逃逸?
   1.5 把变量分配到栈上和堆上有什么区别?
2. goroutine
   2.1 为什么无法从父 goroutine 中恢复子 goroutine?
3. 数组和切片
   3.1 数组和切片的区别
   3.2 切片的扩容策略
   3.3 Go 中对 nil 的 Slice 和空 Slice 的处理是一致的吗?
4. map
   4.1 map 的底层实现
   4.2 map 如何扩容
   4.3 什么时候会发生扩容
   4.4 map 什么时候会发生迁移
   4.5 map 查找
   4.6 map 删除
   4.7 为什么遍历map是无序的?
   4.8 如何实现有序遍历map?
   4.9 为什么Go map是非线程安全的?
   4.10 线程安全的map如何实现?
   4.11 Go sync.map 和原生 map 谁的性能好，为什么?
   4.12 为什么 Go map 的负载因子是 6.5?
5. struct
   5.1 Go 的 Struct 能不能比较?

1.4 context 是如何一层一层通知子 context

调度机制

1. 什么是调度机制
2. 什么是调度器
3. 说一下GMP

   首先是GMP的基础。

   其次是GMP的核心。

      两个队列

      两种调度

   最后是 GMP的设计策略

      复用线程

      1）work stealing 机制

      2）hand off 机制

      利用并行

      抢占调度

      全局 G 队列
4. goroutine 有哪几种状态。
5. 什么时候会发生调度
   1. 使用关键字 go
   2. GC
   3. 系统调用
   4. 内存同步访问
6. 特殊的 M0 和 G0

   M0

   G0
7. 队列轮转
8. 一个G由于调度被中断，此后如何恢复?
9. GMP 为什么要有 P?
10. 加了 P 之后会带来什么改变呢?
11. Go Sheduler 的职责是什么
12. GMP 偷取G的时候为什么不用加锁?
13. 系统监控 sysmon 后台监控线程做了什么

基础

1. 逃逸分析

1.1 逃逸分析是什么?

在编译原理中，分析指针动态范围的方法称之为逃逸分析。在Go中的表现是，如果一个对象的指针被多个方法或线程引用时，则称这个指针发生了逃逸。

所以，我认为逃逸分析指的是，通过分析变量的指针作用范围，来决定这个变量是分配在堆上还是栈上。

1.2 逃逸分析的作用是什么?

通过逃逸分析，可以把那些不需要分配在堆上的变量直接分配到栈上，堆上的变量少了，会减轻堆内存分配的开销，同时减少 GC 的压力，提高程序的运行速度。

1.3 逃逸分析是怎么完成的?

首先，Go语言逃逸分析的最基本的原则是：如果一个函数返回对一个变量的引用，那么这个变量就会发生逃逸。

其次是，编译器会分析代码的特征和代码的生命周期，Go 中的变量只有在编译器可以证明函数返回后是否会被引用。

简单地说，编译器会根据变量是否被外部引用来决定是否逃逸。

1. 如果变量在函数外部没有引用，则优先放在栈上。
2. 如果变量在函数外部存在引用，则必定放在堆上。

1.4 如何确定一个变量是否发生逃逸?

可以在编译的时候，启用编译器的额外支持，`-gcflags` 输出编译器的优化细节，其中包含了逃逸分析的变量提示。

1.5 把变量分配到栈上和堆上有什么区别?

堆和栈相比，堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。栈内存分配则会非常快。栈分配内存只需要两个CPU指令：“PUSH”和“RELEASE”，分配和释放；而堆分配内存

首先需要去找到一块大小合适的内存块，之后要通过垃圾回收才能释放。

2. goroutine

2.1 为什么无法从父 goroutine 中恢复子 goroutine?

因为 goroutine 被设计为一个独立的代码执行单元，拥有自己的执行栈，不与其他 goroutine 共享任何数据。也就是说，无法让 gorourine 拥有返回值，也无法让 goroutine 拥有像 pid 那样的唯一 id 标识。

3. 数组和切片

3.1 数组和切片的区别

Go语言中数组是固定长度的，不能动态扩容，在编译期就会确定大小。

切片是一种数据结构，包含一个底层数组的指针，当前切片个数 len 以及切片的最大容量 cap， 描述的是一块数组。

3.2 切片的扩容策略

切片的扩容都是调用growslice方法，不同版本，扩容机制也有细微差距。

Go1.17 版本，切片在扩容时会进行内存对齐，这个和内存分配策略相关。进行内存对齐之后，新 slice 的容量是要 大于等于老 slice 容量的 2倍或者1.25倍。

当新切片需要的容量大于两倍扩容的容量，则直接按照新切片需要的容量扩容，当原 slice 容量小于 1024 的时候，新 slice 容量变成原来的 2 倍；原 slice 容量超过 1024，新 slice 容量变成原来的1.25倍。

在1.18时，又改成不和1024比较了，而是和256比较。

简单地说，就是小切片按照2倍扩容，大切片按照1.25倍扩容。

Go官方注释说这么做的目的是能更平滑的过渡。

3.3 Go 中对 nil 的 Slice 和空 Slice 的处理是一致的吗?

首先 Go 的 JSON 标准库对 nil slice 和 空 slice 的处理是不一致。

- `slice := make([]int,0)` ：slice 不为 nil，但是 slice 没有值，slice 的底层的空间是空的。
- `slice := []int{}` ：slice 的值是 nil，可用于需要返回 slice 的函数，当函数出现异常的时候，保证函数依然会有 nil 的返回值。

4. map

4.1 map 的底层实现

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)。

4.2 map 如何扩容

扩容其实就是一个预分配内容的过程，在 map 中的表现是 预分配 bucket。

1. 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。
2. 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。

4.3 什么时候会发生扩容

触发 map 扩容的时机：在向 map 插入新 key 的时候，会进行条件检测，符合下面这 2 个条件，就会触发扩容：

1. 装载因子超过阈值
2. overflow 的 bucket 数量过多

> - 对于条件 1，元素太多，而 bucket 数量太少，很简单：将 B 加 1，bucket 最大数量(2^B)直接变成原来 bucket 数量的 2 倍。于是，就有新老 bucket 了。注意，这时候元素都在老 bucket 里，还没迁移到新的 bucket 来。新 bucket 只是最大数量变为原来最大数量的 2 倍(2^B * 2) 。
> - 对于条件 2，其实元素没那么多，但是 overflow bucket 数特别多，说明很多 bucket 都没装满。解决办法就是开辟一个新 bucket 空间，将老 bucket 中的元素移动到新 bucket，使得同一个 bucket 中的 key 排列地更紧密。这样，原来，在 overflow bucket 中的 key 可以移动到 bucket 中来。结果是节省空间，提高 bucket 利用率，map 的查找和插入效率自然就会提升。

4.4 map 什么时候会发生迁移

map 扩容完毕后，不会马上就进行迁移。而是采取**增量扩容**的方式，当有访问到具体 bukcet 时，才会逐渐的进行迁移（将 oldbucket 迁移到 bucket）。

4.5 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。

4.6 map 删除

删除的逻辑相对比较简单，核心是找到 key 的具体位置。在 bucket 中挨个 cell 寻找。找到对应位置后，对 key 或者 value 进行“清零”操作，将 count 值减 1，将对应位置的 tophash 值置成 Empty

4.7 为什么遍历map是无序的?

遍历的过程，就是按顺序遍历 bucket，同时按顺序遍历 bucket 中的 key，而每个 key 进入 bucket 之前都先进行了哈希散列，所以没办法保证遍历的有序性。

4.8 如何实现有序遍历map?

可以把 map 的 key 全部拿出来进行排序，然后根据 key 去获取 value 。

4.9 为什么Go map是非线程安全的?

因为hash map 的内存是按照2的倍数开辟的，当前面开辟的内存不够的时候，会新开辟一段内存，将原来内存的数据转移到新的内存块中，这个过程是没有加锁的，如果这个时候同时有个读的线程过来获取这块内存数据，就会出现安全问题。

4.10 线程安全的map如何实现?

避免map并发读写panic的方式之一就是加锁，考虑到读写性能，可以使用读写锁提供性能。

4.11 Go sync.map 和原生 map 谁的性能好，为什么?

在基准测试中，在并发安全的情况下sync.Map会比我们常用的map+读写锁更加的快，快了五倍，这是得以于只读read设计，减低锁的粒度。但是利用读写锁的话，我们存储的不是一个简单数据类型，而是一个指针对象，那么用普通map+读写锁能很好地控制锁的粒度，达到更好的操作。

4.12 为什么 Go map 的负载因子是 6.5?

源码里定义的阈值的 6.5 (loadFactorNum/loadFactorDen)，是经过测试后取出的一个比较合理的因子

因为每个 bucket 有 8 个空位，在没有溢出，且所有的桶都装满了的情况下，装载因子算出来的结果是 8。因此当装载因子超过 6.5 时，表明很多 bucket 都快要装满了，查找效率和插入效率都变低了。在这个时候进行扩容是有必要的。

5. struct

5.1 Go 的 Struct 能不能比较?

- 相同 struct 类型的可以比较

- 不同 struct 类型的不可以比较,编译都不过，类型不匹配

5.2 空 struct{} 占用空间么?

可以使用 unsafe.Sizeof 计算出一个数据类型实例需要占用的字节数:

Go 复制代码

```go
package main

import (
  "fmt"
  "unsafe"
)

func main() {
  fmt.Println(unsafe.Sizeof(struct{}{}))  //0
}

```

空结构体 struct{} 实例不占据任何的内存空间。

5.3 空 struct{} 的用途?

因为空结构体不占据内存空间，因此被广泛作为各种场景下的占位符使用。

1. 将 map 作为集合(Set)使用时，可以将值类型定义为空结构体，仅作为占位符使用即可。

2. 不发送数据的信道(channel)

使用 channel 不需要发送任何的数据，只用来通知子协程(goroutine)执行任务，或只用来控制协程并发度。

3. 结构体只包含方法，不包含任何的字段

6. 细枝末节

6.1 Go 中的 rune 和 byte 有什么区别?

在 Go 中字符类型有两种，分别是：

1. byte 类型：字节，是 uint8 的别名类型
2. rune 类型：字符，是 int32 的别名类型

byte 和 rune ，虽然都能表示一个字符，但 byte 只能表示 ASCII 码表中的一个字符（ASCII 码表总共有 256 个字符），数量远远不如 rune 多。

rune 表示的是 Unicode字符中的任一字符，而我们都知道，Unicode 是一个可以表示世界范围内的绝大部分字符的编码，这张表里几乎包含了全世界的所有的字符，当然中文也不在话下。

6.2 golang 中 new 和 make 的区别?

- make 仅用来分配及初始化类型为 slice、map、chan 的数据。
- new 可分配任意类型的数据，根据传入的类型申请一块内存，返回指向这块内存的指针，即类型 *Type。
- make 返回引用，即 Type，new 分配的空间被清零， make 分配空间后，会进行初始。

6.3 go 打印时 %v %+v %#v 的区别?

- `%v` 只输出所有的值；
- `%+v` 先输出字段名字，再输出该字段的值；
- `%#v` 先输出结构体名字值，再输出结构体（字段名字+字段的值）；

6.4 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

1. 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。
2. 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。

6.5 其他语言相比，使用golang有什么好处?

- 与其他作为学术实验的语言不同，golang代码的设计是实务的。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。

- 由于单一的标准代码格式，golang通常被认为比其他语言更具有可读性。
- 具有自动垃圾回收机制。

7. chan

7.1 Channel 是同步的还是异步的?

Channel 是异步进行的, channel 存在 3 种状态：

- nil，未初始化的状态，只进行了声明，或者手动赋值为 nil
- active，正常的 channel，可读或者可写
- closed，已关闭，千万不要误认为关闭 channel 后，channel 的值是 nil

7.2 channel 为什么它可以做到线程安全?

Channel 可以理解是一个先进先出的队列，通过管道进行通信, 发送一个数据到 Channel 和从 Channel 接收一个数据都是原子性的。不要通过共享内存来通信，而是通过通信来共享内存，前者就是传统的加锁，后者就是 Channel。设计 Channel 的主要目的就是在多任务间传递数据的，本身就是安全的。

7.3 chan 的数据结构

chan 的底层主要是一个 `hchan` 的结构体，包含几个方面：

1. 环形队列的表示
2. 队列的发送或者接收的索引位置
3. 等待读或者写的 goroutine 队列
4. chan 的元数据，包括元素类型以及 chan 是否关闭。

```go
type hchan struct {
	qcount   uint  // 队列中的总元素个数
	dataqsiz uint  // 环形队列大小，即可存放元素的个数
	buf      unsafe.Pointer // 环形队列指针
	elemsize uint16  //每个元素的大小
	closed   uint32  //标识关闭状态
	elemtype *_type // 元素类型
	sendx    uint   // 发送索引，元素写入时存放到队列中的位置

	recvx    uint   // 接收索引，元素从队列的该位置读出
	recvq    waitq  // 等待读消息的goroutine队列
	sendq    waitq  // 等待写消息的goroutine队列
	lock mutex  //互斥锁，chan不允许并发读写
}

```

7.4 chan 的读写

向 channel 写数据:

1. 若等待接收队列 recvq 不为空，则缓冲区中无数据或无缓冲区，将直接从 recvq 取出 G ，并把数据写入，最后把该 G 唤醒，结束发送过程。
2. 若缓冲区中有空余位置，则将数据写入缓冲区，结束发送过程。
3. 若缓冲区中没有空余位置，则将发送数据写入 G，将当前 G 加入 sendq ，进入睡眠，等待被读 goroutine 唤醒。

从 channel 读数据

1. 若等待发送队列 sendq 不为空，且没有缓冲区，直接从 sendq 中取出 G ，把 G 中数据读出，最后把 G 唤醒，结束读取过程。
2. 如果等待发送队列 sendq 不为空，说明缓冲区已满，从缓冲区中首部读出数据，把 G 中数据写入缓冲区尾部，把 G 唤醒，结束读取过程。
3. 如果缓冲区中有数据，则从缓冲区取出数据，结束读取过程。
4. 将当前 goroutine 加入 recvq ，进入睡眠，等待被写 goroutine 唤醒。

关闭 channel

1.关闭 channel 时会将 recvq 中的 G 全部唤醒，本该写入 G 的数据位置为 nil。将 sendq 中的 G 全部唤醒，但是这些 G 会 panic。

7.5 chan 中 panic 出现的场景有：

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

7.6 chan 堵塞的场景

1. 读写一个 `nil` channel。

7.7 无缓冲 Chan 的发送和接收是否同步？

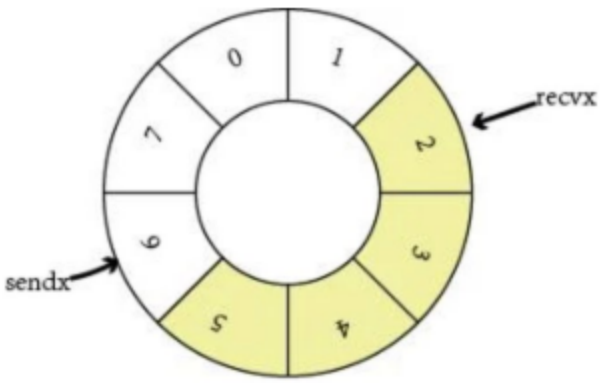
channel 无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到数据；channel 有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

所以无缓冲是发送和接收是同步的。

7.8 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。

7.9 简单介绍一下 chan 的内部实现?

channel 内部维护了两个 goroutine 队列，一个是待发送数据的 goroutine 队列，另一个是待读取数据的 goroutine 队列。

每当对 channel 的读写操作超过了可缓冲的 goroutine 数量，那么当前的 goroutine 就会被挂到对应的队列上，直到有其他 goroutine 执行了与之相反的读写操作，将它重新唤起。

8. defer

8.1 defer 的作用是什么

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。

8.2 defer 的常用场景：

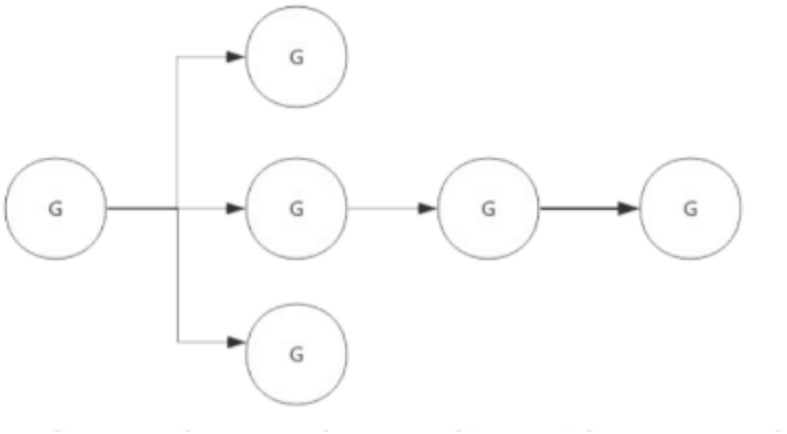
- defer 语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、 加锁、释放锁。
- 通过 defer 机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资 源被释放。
- 释放资源的 defer 应该直接跟在请求资源的语句后。

语言类库

1. Context

1.1 context 的用途

Context（上下文）是 Golang 应用开发常用的并发控制技术，它可以控制一组呈树状结构的 goroutine，每个 goroutine 拥有相同的上下文。Context 是并发安全的，主要是用于控制多个协程之间的协作、取消操作。



1.2 context 的数据结构

Context 只定义了接口，凡是实现该接口的类都可称为是一种 context。

Go 复制代码

```go
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}

```

- 「Deadline」 方法：可以获取设置的截止时间，返回值 deadline 是截止时间，到了这个时间，Context 会自动发起取消请求，返回值 ok 表示是否设置了截止时间。
- 「Done」 方法：返回一个只读的 channel ，类型为 struct{}。如果这个 chan 可以读取，说明已经发出了取消信号，可以做清理操作，然后退出协程，释放资源。
- 「Err」 方法：返回 Context 被取消的原因。
- 「Value」 方法：获取 Context 上绑定的值，是一个键值对，通过 key 来获取对应的值。

1.3 context 的使用场景以及注意事项

Go 里的 context 有 cancelCtx 、timerCtx、valueCtx。它们分别是用来通知取消、通知超时、存储 key – value 值。context 的 注意事项如下：

- context 的 Done() 方法往往需要配合 select {} 使用，以监听退出。
- 尽量通过函数参数来暴露 context，不要在自定义结构体里包含它。
- WithValue 类型的 context 应该尽量存储一些全局的 data，而不要存储一些可有可无的局部 data。
- context 是并发安全的。
- 一旦 context 执行取消动作，所有派生的 context 都会触发取消。

1.4 context 是如何一层一层通知子 context

当 `ctx, cancel := context.WithCancel(父Context)` 时，会将当前的 ctx 挂到父 context 下，然后开个 goroutine 协程去监控父 context 的 channel 事件，一旦有 channel 通知，则自身也会触发自己的 channel 去通知它的子 context， 关键代码如下

go Go 复制代码

```go
go func() {
    select {
    case <-parent.Done():
    child.cancel(false, parent.Err())
    case <-child.Done():
    }
}()
```

调度机制

1. 什么是调度机制

Go 程序的执行有用户程序和运行时两个层面，它们通过函数调用来实现内存管理、channel 通信、Go routine 运行。用户程序进行的所有系统调用，都会被 Runtime 拦截，方便后续的调度以及垃圾回收相关的工作。

而调度机制则是调度器（ `Go Scheduler` ）调度 Goroutine 到内核线程运行时所采用的一种策略。

它的具体实现，就是我们常说的GMP模型。

2. 什么是调度器

调度器在Go 源码中的标识是 `Go scheduler` 。它是运行时最重要的一部分。Runtime 维护所有的 Goroutine ， 并通过scheduler 进行调度。goroutine 与 threads 是独立，但是 goroutine 要依赖 threads 才能运行。

3. 说一下GMP

首先是GMP的基础。

- G 代表的是一个 Goroutine 是 Go 运行时调度的基本单位， 包含 Goroutine 的栈信息，也就是 routine 的运行时上下文，以及 pc 指向当前运行到的指令位置。
- M 代表内核线程，包含正在运行的 Go routine 相关信息。
- P 代表一个虚拟的处理器 Processor，它维护一个处于 Runnable 状态的 goroutine 队列，称为本地队列。

其次是GMP的核心。

也就是两个队列以及两种调度。

两个队列

在整个 Go 调度器的生命周期中，存在着两个非常重要的队列：

- 全局队列（Global Runnable Queue）：全局只有一个
- 本地队列（Local Runnable Queue）：每个 P 都会维护一个本地队列

当你执行 go func() 创建一个 goroutine 时，会优选将该协程放入到当前 P 的本地队列中等待被 P 选中执行。但是，如果当前 P 的本地队列任务太多了，已经存放不下了，那么这个 goroutine 就只能放入到全局队列中。

> 这两种队列使用环形链表实现，最多可以存储 256 个待执行任务。

两种调度

一个协程得以运行，需要同时满足以下两个条件：

1. P 已经和某个线程进行绑定，这样才能参考操作系统的调度获得 CPU 时间

2. P 已经从队列中（可以是本地队列，也可以是全局队列，甚至是从其他 P 的队列）取到该协程

第一个条件就是 **操作系统调度**，而第二个其实就是 **Go 里的调度器**。

最后是 GMP的设计策略

复用线程

利用线程池复用线程，避免频繁的创建、销毁线程，减少资源浪费。

1）work stealing 机制

当本线程无可运行的 G 时，尝试从其他线程绑定的 P 偷取 G，而不是销毁线程。

2）hand off 机制

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

利用并行

GOMAXPROCS 设置 P 的数量，最多有 GOMAXPROCS 个线程分布在多个 CPU 上同时运行。
GOMAXPROCS 也限制了并发的程度，比如 GOMAXPROCS = 核数/2，则最多利用了一半的 CPU 核进行并行。

抢占调度

在 Go 中，一个 goroutine 最多占用 CPU 10ms，防止其他 goroutine 被饿死，这是协作式抢占调度。

全局 G 队列

当创建一个新的G之后优先加入本地队列，如果本地队列满了，会将本地队列的G移动到全局队列里面，当M执行work stealing从其他P偷不到G时，它可以从全局G队列获取G。

4. goroutine 有哪几种状态。

- Waiting，等待某件事发生，比如说 io 或者系统调用。
- Runnable，就绪状态，只要获得M就可以运行。

- Executing，运行状态，goroutine 正在 M 上运行。

5. 什么时候会发生调度

1. 使用关键字 go

创建一个新的 goroutine，Go Scheduler 可能会考虑调度。

2. GC

由于进行 GC 的 goroutine 也需要在M上运行， 因此肯定会发生调度。

3. 系统调用

当 goroutine 因为系统调用堵塞时，会被调走，同时新的 Goroutine 会被调度到当前 M。

4. 内存同步访问

`atomic,mutex,channel` 等操作使得Goroutine 堵塞时，也会被调度走。

6. 特殊的 M0 和 G0

M0

M0是启动程序后的编号为0的主线程，这个M对应的实例会在全局变量rutime.m0中，不需要在heap上分配，M0负责执行初始化操作和启动第一个G，在之后M0就和其他的M一样了。

G0

G0是每次启动一个M都会第一个创建的goroutine，G0仅用于负责调度G，G0不指向任何可执行的函数，每个M都会有一个自己的G0，在调度或系统调用时会使用G0的栈空间，全局变量的G0是M0的G0。

G0 栈的作用就是为运行 Runtime 代码提供“环境”的。

7. 队列轮转

可见每个P维护着一个包含G的队列，不考虑G进入系统调用或IO操作的情况下，P周期性的将G调度到M中执行，执行一小段时间，将上下文保存下来，然后将G放到队列尾部，然后从队列中重新取出一个G进行调度。

除了每个P维护的G队列以外，还有一个全局的队列，每个P会周期性地查看全局队列中是否有G待运行并将其调度到M中执行，全局队列中G的来源，主要有从系统调用中恢复的G。之所以P会周期性地查看全局队列，也是为了防止全局队列中的G被饿死。

> m 在本地队列调度达到 61 次之后，如果全局队列有 routine， 那么就会从全局队列中获取一个 Goroutine 运行。

8. 一个G由于调度被中断，此后如何恢复?

中断的时候将寄存器里的栈信息，保存到自己的G对象里面（堆区）。当再次轮到自己执行时，将自己保存的栈信息复制到寄存器里面，这样就接着上次之后运行了。

9. GMP 为什么要有 P?

在 Go v1.1 之前，实际上 GMP确实是没有 P 的，所有的 M 线程都要从 全局队列中获取 G 来执行任务，为了避免冲突，从全局队列中获取 G 的时候，要先获取一把大锁。

当一个程序的并发量比较小的时候，影响还不大，而当程序的并发量非常大的时候，这个全局队列会成为性能的瓶颈。

除此之外 ，若直接把 G 从全局队列分配给 M，那么当 G 中发生系统调用或者其他阻塞性的操作时，M 会有一段时间处于挂起的状态，此时又没有新创建的线程来代替该线程继续从队列中取出其他 G 来运行，从效率上其实会打折扣。

10. 加了 P 之后会带来什么改变呢?

- 每个 P 有自己的本地队列，大幅度的减轻了对全局队列的直接依赖，所带来的效果就是锁竞争的减少。而 GM 模型的性能开销大头就是锁竞争。
- 当一个 M 中 运行的 G 发生阻塞性操作时，P 会重新选择一个 M，若没有 M 就新创建一个 M 来继续从 P

本地队列中取 G 来执行，提高运行效率。

- 每个 P 相对的平衡上，在 GMP 模型中也实现了 Work Stealing 算法，如果 P 的本地队列为空，则会从全局队列或其他 P 的本地队列中窃取可运行的 G 来运行，减少空转，提高了资源利用率。

11. Go Sheduler 的职责是什么

将所有处于 Runnable 的 Goroutine 均匀调度到在P上运行的M。

12. GMP 偷取G的时候为什么不用加锁?

P 从本地队列取 G 的这个操作，是一个 CAS 操作，它具有原子性，是由硬件直接支持的，不需要并发的竞争关系。

13. 系统监控 sysmon 后台监控线程做了什么

在 `runtime.main()` 函数中，执行 `runtime_init()` 前，会启动一个 `sysmon` 的监控线程，执行后台监控任务。

> sysmon 执行一个无限循环，一开始每次循环休眠 20us，之后（1 ms 后）每次休眠时间倍增，最终每一轮都会休眠 10ms。

sysmon 中会进行 netpool（获取 fd 事件）、retake（抢占）、forcegc（按时间强制执行 gc），scavenge heap（释放自由列表中多余的项减少内存占用）等处理。

14. M 如何找工作

> 工作线程 M 费尽心机也要找到一个可运行的 goroutine，这是它的工作和职责，不达目的，绝不罢体，这种锲而不舍的精神值得每个人学习。

共经历三个过程：先从本地队列找，定期会从全局队列找，最后实在没办法，就去别的 P 偷。

内存分配管理

垃圾回收机制

1. 三色标记原理

三色标记算法将程序中的对象分成白色、灰色和黑色三类：

- 白色对象 — 潜在的垃圾，其内存可能会被垃圾收集器回收；
- 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；
- 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；

三色标记法分三个阶段：标记阶段、着色阶段和回收阶段。

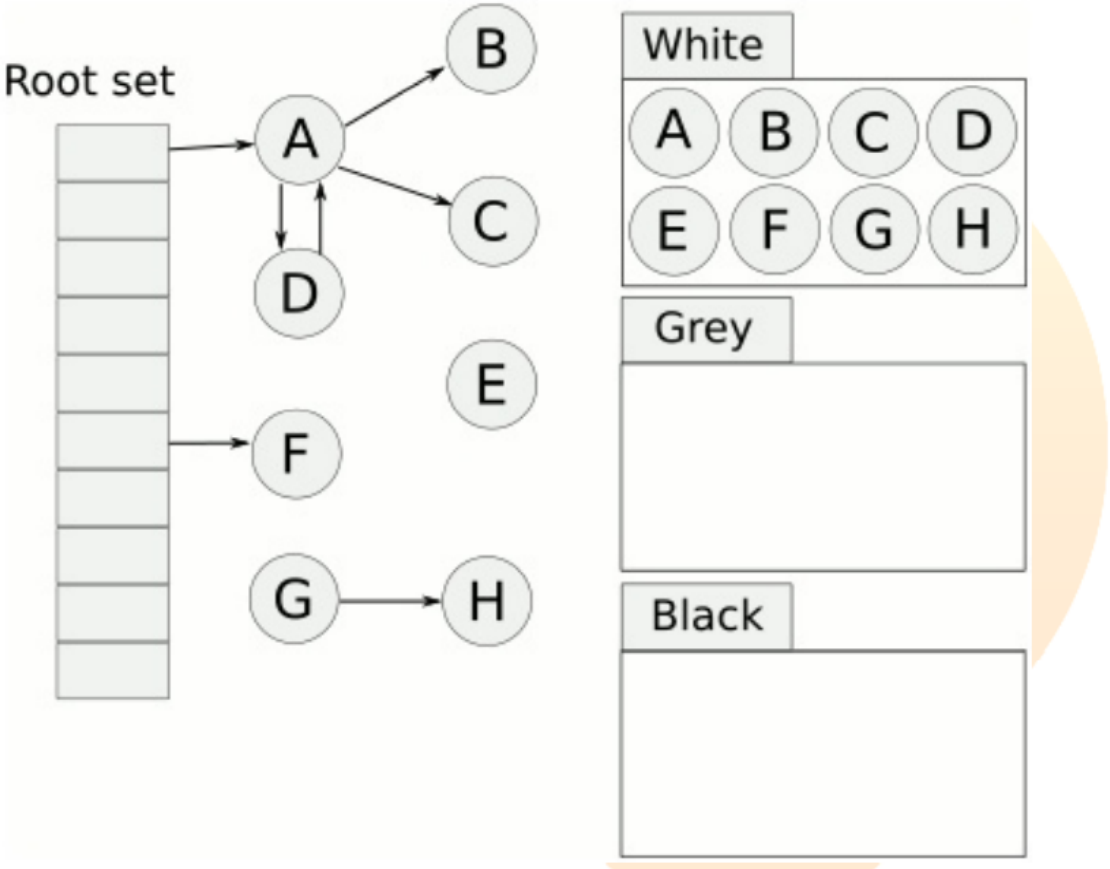
标记阶段

在 GC 工作开始时，Go 语言随即为每个处理机 P 配置一个 `mark worker` 线程 G 来帮助标记内存，它们负责为进入 STW 后的清理做前期工作。

一旦根程序（Root）被加入待处理队列 `work pool`，标记周期便正式开始，这些线程 G 开始遍历 span 并标记内存块 object。

着色阶段

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：



- 1. 首先把所有的对象都放到白色的集合中。
- 2. 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。
- 3. 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
- 4. 循环步骤 3，直到灰色集合中没有对象
- 5. 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

> 当三色的标记清除的标记阶段结束之后，应用程序的堆中就不存在任何的灰色对象，我们只能看到黑色的存活对象以及白色的垃圾对象，垃圾收集器可以回收这些白色的垃圾

回收阶段

2. 什么是STW

STW 在 Go 中指的是 Stop The World 到 Start The World 这段时间。

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

3. 为什么需要屏障技术

因为用户程序可能在标记执行的过程中修改对象的指针，所以三色标记清除算法本身是不可以并发或者增量执行的，它仍然需要 STW，所以后面新增了屏障技术，并发或者增量地标记对象。

> 造成引用对象丢失的条件:
>
> 一个黑色的节点 A 新增了指向白色节点 C 的引用，并且白色节点 C 没有除了 A 之外的其他灰色节点的引用，或者存在但是在 GC 过程中被删除了。以上两个条件需要同时满足：满足条件 1 时说明节点 A 已扫描完毕，A 指向 C 的引用无法再被扫描到；满足条件 2 时说明白色节点 C 无其他灰色节点的引用了，即扫描结束后会被忽略 。
>
> 写屏障破坏两个条件其一即可
>
> - 破坏条件 1：Dijistra 写屏障
>
> 满足强三色不变性：黑色节点不允许引用白色节点当黑色节点新增了白色节点的引用时，将对应的白色节点改为灰色
>
> - 破坏条件 2：Yuasa 写屏障
>
> 满足弱三色不变性：黑色节点允许引用白色节点，但是该白色节点有其他灰色节点间接的引用（确保不会被遗漏）当白色节点被删除了一个引用时，悲观地认为它一定会被一个黑色节点新增引用，所以将它置为灰色

4. 什么是屏障技术

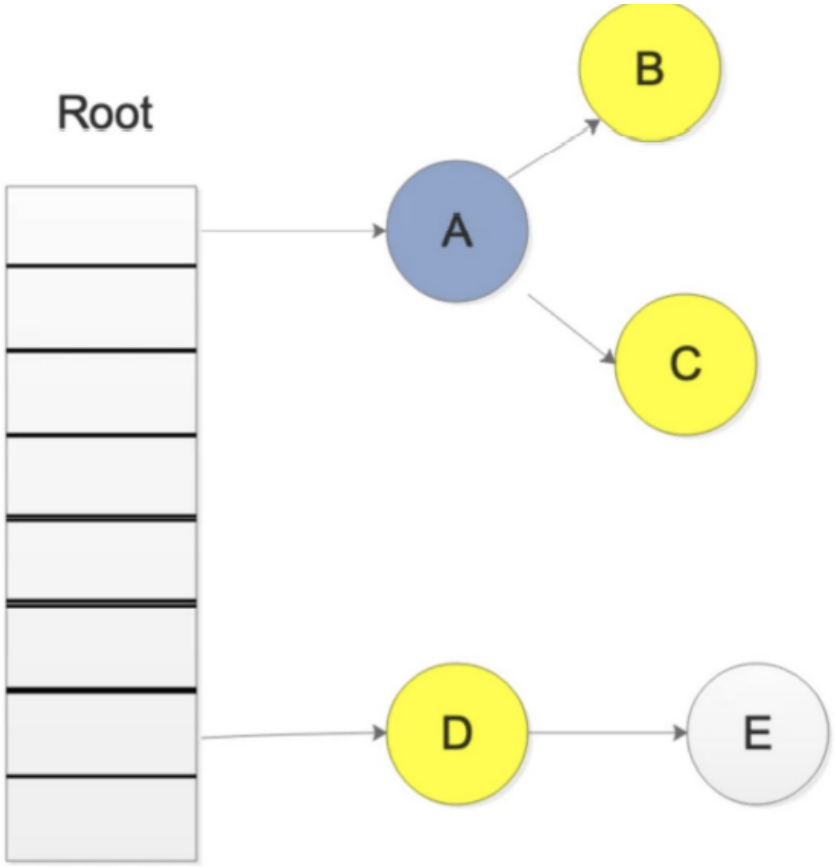
垃圾收集中的屏障技术更像是一个钩子方法，它是在用户程序读取对象、创建新对象以及更新对象指针时执行的一段代码，根据操作类型的不同，我们可以将它们分成读屏障（Read barrier）和写屏障（Write barrier）两种，因为读屏障需要在读操作中加入代码片段，对用户程序的性能影响很大，所以编程语言往往都会采用写屏障保证三色不变性。

4. 什么是三色不变性

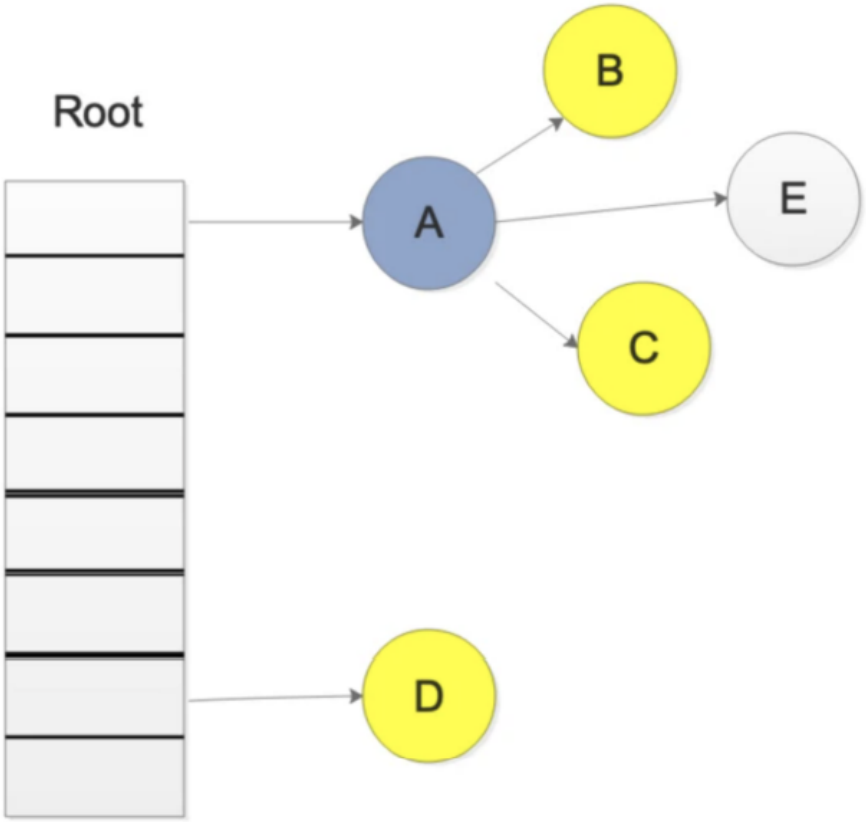
- 强三色不变性 — 黑色对象不会指向白色对象，只会指向灰色对象或者黑色对象；
- 弱三色不变性 — 黑色对象指向的白色对象必须包含一条从灰色对象经由多个白色对象的可达路径。

5. 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



- 这样就会导致 E 对象就扫描不到了，而被误认为白色对象，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。

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

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。

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，原子状态切换）。

6. GC 的触发条件?

- 主动触发(手动触发)，通过调用 runtime.GC 来触发 GC，此调用阻塞式地等待当前 GC 运行完毕。
- 被动触发，分为两种方式：
  - 使用系统监控，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,每次内存分配时检查当前内存分配量是否已达到阈值（环境变量 GOGC）：默认 100%，即当内存扩大一倍时启用 GC。

性能相关

1. 为什么要进行内存对齐?

因为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。这么设计的目的，是减少 CPU 访问内存的次数，加大 CPU 访问内存的吞吐量。比如同样读取 8 个字节的数据，一次读取 4 个字节那么只需要读取 2 次。

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。

总而言之，就是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。

并发编程

sync

Mutex

互斥锁的加锁过程比较复杂，它涉及自旋、信号量以及调度等概念：

- 如果互斥锁处于初始化状态，会通过置位 mutexLocked 加锁；
- 如果互斥锁处于 mutexLocked 状态并且在普通模式下工作，会进入自旋，执行 30 次 PAUSE 指令消耗 CPU 时间等待锁的释放；
- 如果当前 Goroutine 等待锁的时间超过了 1ms，互斥锁就会切换到饥饿模式；
- 互斥锁在正常情况下会通过 runtime.sync_runtime_SemacquireMutex 将尝试获取锁的 Goroutine 切换至休眠状态，等待锁的持有者唤醒；
- 如果当前 Goroutine 是互斥锁上的最后一个等待的协程或者等待的时间小于 1ms，那么它会将互斥锁切换回正常模式；

互斥锁的解锁过程与之相比就比较简单，其代码行数不多、逻辑清晰，也比较容易理解：

- 当互斥锁已经被解锁时，调用 sync.Mutex.Unlock 会直接抛出异常；
- 当互斥锁处于饥饿模式时，将锁的所有权交给队列中的下一个等待者，等待者会负责设置 mutexLocked 标志位；

- 当互斥锁处于普通模式时，如果没有 Goroutine 等待锁的释放或者已经有被唤醒的 Goroutine 获得了锁，会直接返回；在其他情况下会通过 sync.runtime_Semrelease 唤醒对应的 Goroutine；

RWMutex

虽然读写互斥锁 sync.RWMutex 提供的功能比较复杂，但是因为它建立在 sync.Mutex 上，所以实现会简单很多。我们总结一下读锁和写锁的关系：

- 调用 sync.RWMutex.Lock 尝试获取写锁时；
  - 每次 sync.RWMutex.RUnlock 都会将 readerCount 其减一，当它归零时该 Goroutine 会获得写锁；
  - 将 readerCount 减少 rwmutexMaxReaders 个数以阻塞后续的读操作；
- 调用 sync.RWMutex.Unlock 释放写锁时，会先通知所有的读操作，然后才会释放持有的互斥锁；

读写互斥锁在互斥锁之上提供了额外的更细粒度的控制，能够在读操作远远多于写操作时提升性能。

WaitGroup

通过对 sync.WaitGroup 的分析和研究，我们能够得出以下结论：

- sync.WaitGroup 必须在 sync.WaitGroup.Wait 方法返回之后才能被重新使用；
- sync.WaitGroup.Done 只是对 sync.WaitGroup.Add 方法的简单封装，我们可以向 sync.WaitGroup.Add 方法传入任意负数（需要保证计数器非负）快速将计数器归零以唤醒等待的 Goroutine；
- 可以同时有多个 Goroutine 等待当前 sync.WaitGroup 计数器的归零，这些 Goroutine 会被同时唤醒；

Once

Go 语言标准库中 sync.Once 可以保证在 Go 程序运行期间的某段代码只会执行一次。在运行如下所示的代码时，我们会看到如下所示的运行结果：

Go 复制代码

```go
func main() {
    o := &sync.Once{}
    for i := 0; i < 10; i++ {
        o.Do(func() {
            fmt.Println("only once")
        })
    }
}

$ go run main.go
only once
```

结构体

每一个 sync.Once 结构体中都只包含一个用于标识代码块是否执行过的 done 以及一个互斥锁 sync.Mutex：

Go 复制代码

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

Go

接口

sync.Once.Do 是 sync.Once 结构体对外唯一暴露的方法，该方法会接收一个入参为空的函数：

- 如果传入的函数已经执行过，会直接返回；
- 如果传入的函数没有执行过，会调用 sync.Once.doSlow 执行传入的函数：

```go
func (o *Once) Do(f func()) {
    if atomic.LoadUint32(&o.done) == 0 {
        o.doSlow(f)
    }
}

func (o *Once) doSlow(f func()) {
    o.m.Lock()
    defer o.m.Unlock()
    if o.done == 0 {
        defer atomic.StoreUint32(&o.done, 1)
        f()
    }
}
```

1. 为当前 Goroutine 获取互斥锁；
2. 执行传入的无入参函数；
3. 运行延迟函数调用，将成员变量 done 更新成 1；

sync.Once 会通过成员变量 done 确保函数不会执行第二次。

小结

作为用于保证函数执行次数的 sync.Once 结构体，它使用互斥锁和 sync/atomic 包提供的方法实现了某个函数在程序运行期间只能执行一次的语义。在使用该结构体时，我们也需要注意以下的问题：

- sync.Once.Do 方法中传入的函数只会被执行一次，哪怕函数中发生了 panic；
- 两次调用 sync.Once.Do 方法传入不同的函数只会执行第一次调传入的函数；

select

我们简单总结一下 select 结构的执行过程与实现原理，首先在编译期间，Go 语言会对 select 语句进行优化，它会根据 select 中 case 的不同选择不同的优化路径：

1. 空的 select 语句会被转换成调用 runtime.block 直接挂起当前 Goroutine；
2. 如果 select 语句中只包含一个 case，编译器会将其转换成 if ch == nil { block }; n; 表达式；

    - 首先判断操作的 Channel 是不是空的；
    - 然后执行 case 结构中的内容；
3. 如果 select 语句中只包含两个 case 并且其中一个是 default，那么会使用 runtime.selectnbrecv 和 runtime.selectnbsend 非阻塞地执行收发操作；
4. 在默认情况下会通过 runtime.selectgo 获取执行 case 的索引，并通过多个 if 语句执行对应 case 中的代码；

在编译器已经对 select 语句进行优化之后，Go 语言会在运行时执行编译期间展开的 runtime.selectgo 函数，该函数会按照以下的流程执行：

1. 随机生成一个遍历的轮询顺序 pollOrder 并根据 Channel 地址生成锁定顺序 lockOrder；
2. 根据 pollOrder 遍历所有的 case 查看是否有可以立刻处理的 Channel；
    a. 如果存在，直接获取 case 对应的索引并返回；
    b. 如果不存在，创建 runtime.sudog 结构体，将当前 Goroutine 加入到所有相关 Channel 的收发队列，并调用 runtime.gopark 挂起当前 Goroutine 等待调度器的唤醒；
3. 当调度器唤醒当前 Goroutine 时，会再次按照 lockOrder 遍历所有的 case，从中查找需要被处理的 runtime.sudog 对应的索引；

select 关键字是 Go 语言特有的控制结构，它的实现原理比较复杂，需要编译器和运行时函数的通力合作。