Go 进阶面试题汇总

小白debug (https://xiaobaidebug.top/) 2024年5月19日 大约 40 分钟

golang面试官：for select时，如果通道已经关闭会怎么样？如果只有一个case呢？

答案 (https://golangguide.top/golang/%E6%A0%B8%E5%BF%83%E7%9F%A5%E8%AF%86%E7%82%B9/for%20select%E6%97%B6%EF%BC%8C%E5%A6%82%E6%9E%9C%E9%80%9A%E9%81%93%E5%B7%B2%E7%BB%8F%E5%85%B3%E9%97%AD%E4%BC%9A%E6%80%8E%E4%B9%88%E6%A0%B7%EF%BC%9F%E5%A6%82%E6%9E%9C%E5%8F%AA%E6%9C%89%E4%B8%80%E4%B8%AAcase%E5%91%A2%EF%BC%9F.html)

golang面试题：对已经关闭的的chan进行读写，会怎么样？为什么？

答案 (https://golangguide.top/golang/%E6%A0%B8%E5%BF%83%E7%9F%A5%E8%AF%86%E7%82%B9/golang%E9%9D%A2%E8%AF%95%E9%A2%98%EF%BC%9A%E5%AF%B9%E5%B7%B2%E7%BB%8F%E5%85%B3%E9%97%AD%E7%9A%84%E7%9A%84chan%E8%BF%9B%E8%A1%8C%E8%AF%BB%E5%86%99%EF%BC%8C%E4%BC%9A%E6%80%8E%E4%B9%88%E6%A0%B7%EF%BC%9F%E4%B8%BA%E4%BB%80%E4%B9%88%EF%BC%9F.html)

golang面试题：对未初始化的的chan进行读写，会怎么样？为什么？

答案 (https://golangguide.top/golang/%E6%A0%B8%E5%BF%83%E7%9F%A5%E8%AF%86%E7%82%B9/golang%E9%9D%A2%E8%AF%95%E9%A2%98%EF%BC%9A%E5%AF%B9%E6%9C%AA%E5%88%9D%E5%A7%8B%E5%8C%96%E7%9A%84%E7%9A%8

4chan%E8%BF%9B%E8%A1%8C%E8%AF%BB%E5%86%99%EF%BC%8C%E4%BC%9A%E6%80%8E%E4%B9%88%E6%A0%B7%EF%BC%9F%E4%B8%BA%E4%BB%80%E4%B9%88%EF%BC%9F.html)

golang面试题：能说说uintptr和unsafe.Pointer的区别吗？

https://mp.weixin.qq.com/s/PSkz0zj-vqKzmlKa_b-xAA
(https://mp.weixin.qq.com/s/PSkz0zj-vqKzmlKa_b-xAA)

golang 面试题：reflect（反射包）如何获取字段 tag？为什么 json 包不能导出私有变量的 tag？

答案
(https://golangguide.top/golang/%E6%A0%B8%E5%BF%83%E7%9F%A5%E8%AF%86%E7%82%B9/golang%20%E9%9D%A2%E8%AF%95%E9%A2%98%EF%BC%9Areflect%EF%BC%88%E5%8F%8D%E5%B0%84%E5%8C%85%EF%BC%89%E5%A6%82%E4%BD%95%E8%8E%B7%E5%8F%96%E5%AD%97%E6%AE%B5%20tag%EF%BC%9F%E4%B8%BA%E4%BB%80%E4%9%88%20json%20%E5%8C%85%E4%B8%8D%E8%83%BD%E5%AF%BC%E5%87%BA%E7%A7%81%E6%9C%89%E5%8F%98%E9%87%8F%E7%9A%84%20tag%EF%BC%9F.html)

Golang GC 时会发生什么？

首先我们先来了解下垃圾回收.什么是垃圾回收?

内存管理是程序员开发应用的一大难题。传统的系统级编程语言（主要指C/C++）中，程序开发者必须对内存小心的进行管理操作，控制内存的申请及释放。因为稍有不慎，就可能产生内存泄露问题，这种问题不易发现并且难以定位，一直成为困扰程序开发者的噩梦。如何解决这个头疼的问题呢?

过去一般采用两种办法:

- 内存泄露检测工具。这种工具的原理一般是静态代码扫描，通过扫描程序检测可能出现内存泄露的代码段。然而检测工具难免有疏漏和不足，只能起到辅助作用。
- 智能指针。这是 c++ 中引入的自动内存管理方法，通过拥有自动内存管理功能的指针对象来引用对象，是程序员不用太关注内存的释放，而达到内存自动释放的目的。这种方

法是采用最广泛的做法，但是对程序开发者有一定的学习成本（并非语言层面的原生支持），而且一旦有忘记使用的场景依然无法避免内存泄露。

为了解决这个问题，后来开发出来的几乎所有新语言（java，python，php等等）都引入了语言层面的自动内存管理 – 也就是语言的使用者只用关注内存的申请而不必关心内存的释放，内存释放由虚拟机（virtual machine）或运行时（runtime）来自动进行管理。而这种对不再使用的内存资源进行自动回收的行为就被称为垃圾回收。

常用的垃圾回收的方法:

- 引用计数（reference counting）

这是最简单的一种垃圾回收算法，和之前提到的智能指针异曲同工。对每个对象维护一个引用计数，当引用该对象的对象被销毁或更新时被引用对象的引用计数自动减一，当被引用对象被创建或被赋值给其他对象时引用计数自动加一。当引用计数为0时则立即回收对象

这种方法的优点是实现简单，并且内存的回收很及时。这种算法在内存比较紧张和实时[illegible]较高的系统中使用的比较广泛，如ios cocoa框架，php，python等。

但是简单引用计数算法也有明显的缺点:

1. 频繁更新引用计数降低了性能。

一种简单的解决方法就是编译器将相[illegible]引用计数更新操作合并到一次更新；还有一种方法是针对频繁发生的临时变量引用不进[illegible]数，而是在引用达到0时通过扫描堆栈确认是否还有临时对象引用而决定是否释放。等等还有很多其他方法，具体可以参考这里。

1. 循环引用。

当对象间发生循环引用时引用链中的对象都无法得到释放。最明显的解决办法是避免产生循环引用，如cocoa引入了strong指针和weak指针两种指针类型。或者系统检测循环引用并主动打破循环链。当然这也增加了垃圾回收的复杂度。

- 标记-清除（mark and sweep）

标记-清除（mark and sweep）分为两步，标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为“被引用”；标记完成后进行清除操作，对没有标记过的内存进行回收（回收同时可能伴有碎片整理操作）。这种方法解决了引用计数的不足，但是也有比较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执行，回收是系统响应能力大大降低！当然后续也出现了很多mark&sweep算法的变种（如三色标记法）优化了这个问题。

- 分代搜集（generation）

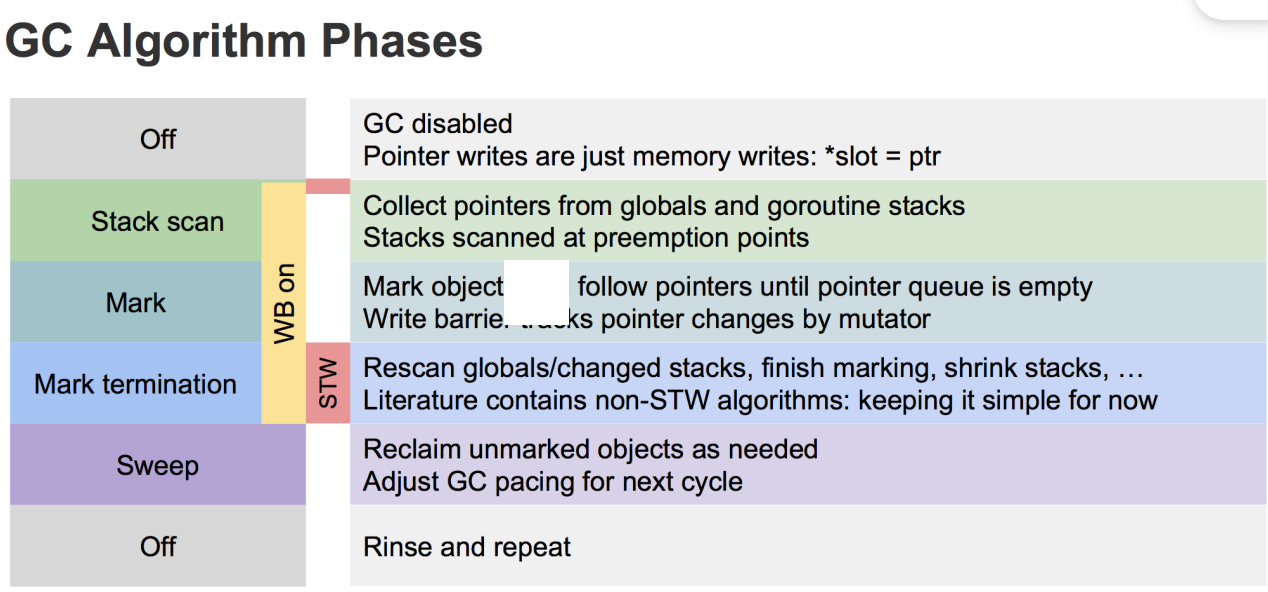
java的jvm 就使用的分代回收的思路。在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为代（generation）的空间。新创建的对象存放在称为新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多），随着垃圾回收的重复执行，生命周期较长的对象会被提升（promotion）到老年代中（这里用到了一个分类的思路，这个是也是科学思考的一个基本思路）。

因此，新生代垃圾回收和老年代垃圾回收两种不同的垃圾回收方式应运而生，分别用于对各自空间中的对象执行垃圾回收。新生代垃圾回收的速度非常快，比老年代快几个数量级，即使新生代垃圾回收的频率更高，执行效率也仍然比老年代垃圾回收强，这是因为大多数对象的生命周期都很短，根本无需提升到老年代。

Golang GC 时会发生什么?

Golang 1.5后，采取的是"非分代的、非移动的、并发的、三色的"标记清除垃圾回收算法。

golang 中的 gc 基本上是标记清除的过程:



img

gc的过程一共分为四个阶段:

1. 栈扫描（开始时STW）
2. 第一次标记（并发）
3. 第二次标记（STW）
4. 清除（并发）

整个进程空间里申请每个对象占据的内存可以视为一个图，初始状态下每个内存对象都是白色标记。

1. 先STW，做一些准备工作，比如 enable write barrier。然后取消STW，将扫描任务作为多个并发的goroutine立即入队给调度器，进而被CPU处理
2. 第一轮先扫描root对象，包括全局指针和 goroutine 栈上的指针，标记为灰色放入队列
3. 第二轮将第一步队列中的对象引用的对象置为灰色加入队列，一个对象引用的所有对象都置灰并加入队列后，这个对象才能置为黑色并从队列之中取出。循环往复，最后队列为空时，整个图剩下的白色内存空间即不可到达的对象，即没有被引用的对象；
4. 第三轮再次STW，将第二轮过程中新增对象申请的内存进行标记（灰色），这里使用了write barrier（写屏障）去记录

Golang gc 优化的核心就是尽量使得 STW(Stop The World) 的时间越来越短。

详细的Golang的GC介绍可以参看[Golang垃圾回收 (https://github.com/KeKe-Li/For-learning-Go-Tutorial/blob/master/src/spec/02.0.md)](https://github.com/KeKe-Li/For-learning-Go-Tutorial/blob/master/src/spec/02.0.md) .

Golang 中 Goroutine 如何调度?

goroutine是Golang语言中最经典的设计，也是其魅力所在，goroutine的本质是协程，是实现并行计算的核心。 goroutine使用方式非常的简单，只需使用go关键字即可启动一个协程，并且它是处于异步方式运行，你不需要等它运行完成以后在执行以后的代码。

```text
go func()//通过go关键字启动一个协程      行函数
```

协程:

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。 因此，协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。 线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作执行者则是用户自身程序，goroutine也是协程。

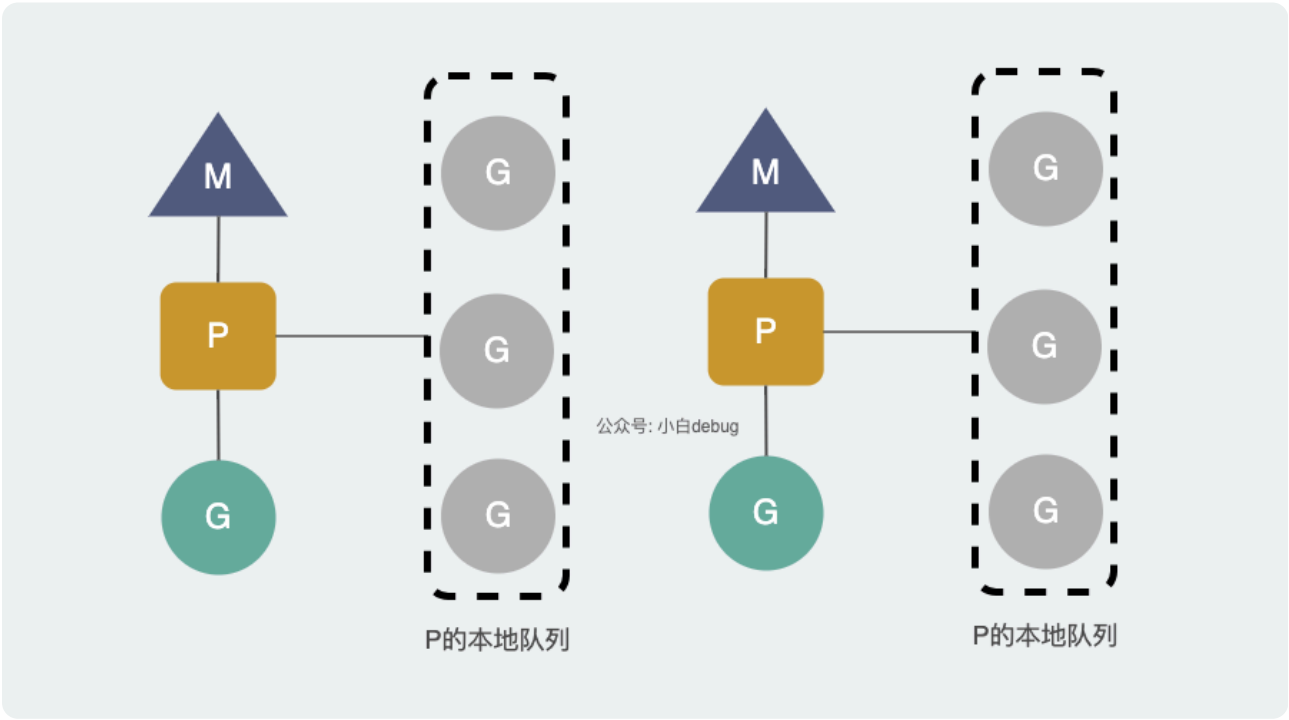
groutine能拥有强大的并发实现是通过GPM调度模型实现.

GMP概念

Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）.

- M:M代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的；M是一个很大的结构，里面维护小对象内存cache（mcache）、当前执行的goroutine、随机数发等等非常多的信息
- G:代表一个goroutine，它有自己的栈，instruction pointer和其他信息（正在等待的channel等等），用于调度。
- P:P全称是Processor，处理器，它的主要用途就是用来执行goroutine的，所以它也维护了一个goroutine队列，里面存储了所有需要它来执行的goroutine
- Sched：代表调度器，它维护有存　　和G的队列以及调度器的一些状态信息等。

调度实现:

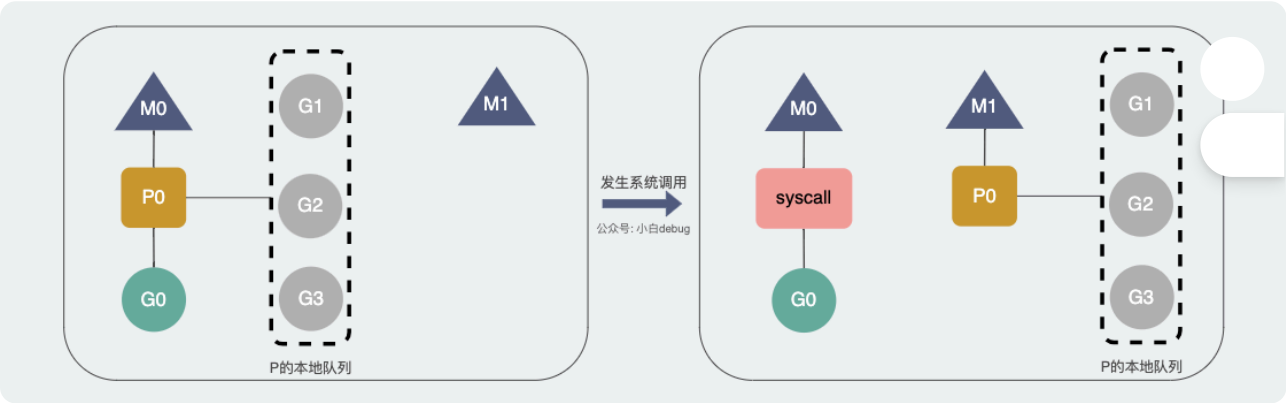


GMP调度示例

从上图中可以看到，有2个物理线程M，两个M都拥有一个处理器P，每一个也都有一个正在运行的goroutine。P的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个goroutine可以同时运行。

图中本地队列里的那些goroutine并没有运行，而是处于ready的就绪态，正在等待被调度。P维护着这个队列（称之为runqueue），Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出（如何决定取哪个goroutine？）一个goroutine执行。

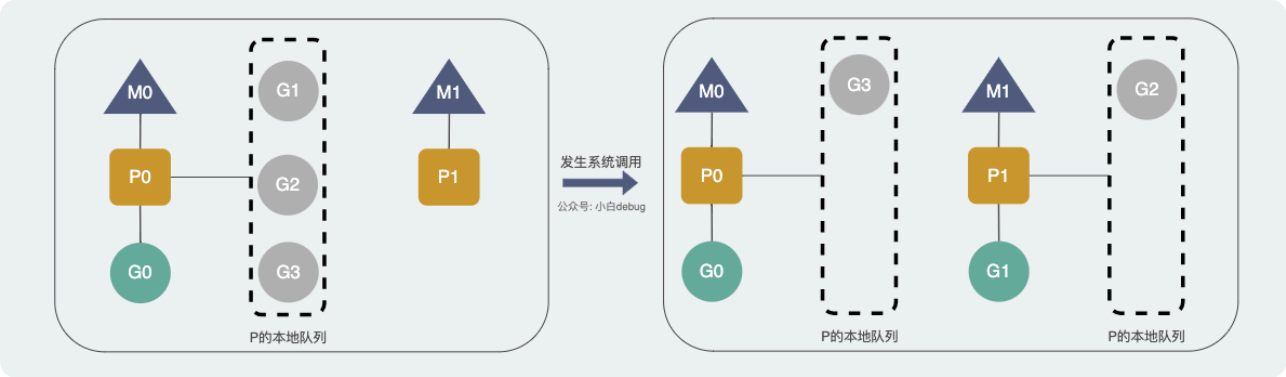
当一个OS线程M0陷入阻塞时，P转而在运行M1，图中的M1可能是正被创建，或者从线程缓存中取出。



发生系统调用

当MO返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS线程那里拿一个P过来， 如果没有拿到的话，它就把goroutine放在一个global runqueue里，然后自己睡眠（放入线程缓存里）。所有的P也会周期性的检查global runqueue并运行其中的goroutine，否则global runqueue上的goroutine永远无法执行。

另一种情况是P所分配的任务G很快就执行完了（分配不均），这就导致了这个处理器P很忙，但是其他的P还有任务，此时如果global runqueue没有任务G了，那么P不得不从其他的P里拿一些G来执行。



重新调度G

通常来说，如果P从其他的P那里要拿任务的话，一般就拿run queue的一半，这就确保了每个OS线程都能充分的使用。

并发编程概念是什么？

并行是指两个或者多个事件在同一时刻发生；并发是指两个或多个事件在同一时间间隔发生。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

并发编程是指在一台处理器上“同时”处理多个任务。并发是在同一实体上的多个事件。多个事件在同一时间间隔发生。并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

下面这段代码有什么错误吗？

```go
func funcMui(x,y int)(sum int,error,{
    return x+y,nil
}
```

解析

> 第二个返回值没有命名,在函数有多个返回值时，只要有一个返回值有命名，
> 其他的也必须命名。如果有多个返回值必须加上括号();
> 如果只有一个返回值且命名也必须加上括号()。
> 这里的第一个返回值有命名 sum，第二个没有命名，所以错误。

下面几段代码能否通过编译，如果能，输出什么?

```go
func main() {
    list := new([]int)
    // 编译错误
    // new([]int) 之后的 list 是一个未设置长度的 *[]int 类型的指针
    // 不能对未设置长度的指针执行 append 操作。
    list = append(list, 1)
    fmt.Println(list)

    s1 := []int{1, 2, 3}
    s2 := []int{4, 5}
    // 编译错误，s2需要展开
    s1 = append(s1, s2)
    fmt.Println(s1)
}
```

下面能否通过编译?

```go
func Test7(t *testing.T) {
    sn1 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}
    sn2 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}
    // true
    if sn1 == sn2 {
        fmt.Println("sn1 == sn2")
    }

    sm1 := struct {
        age int
        m   map[string]string
```

```
    }{age: 11, m: map[string]string{"a": "1"}}
    sm2 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}
    // 编译错误，含有map、slice类型的struct不能进行比较
    if sm1 == sm2 {
        fmt.Println("sm1 == sm2")
    }
}
```

通过指针变量 p 访问其成员变量 name，有哪几种方式？

A. p.name (http://p.name)

B. (&p).name

C. (*p).name

D. p->name

答案

> AC

关于字符串连接，下面语法正确的是？

A. str := 'abc' + '123'

B. str := "abc" + "123"

C. str := '123' + "abc"

D. fmt.Sprintf("abc%d", 123)

答案

> BD
> golang单引号''中的内容表示单个字符（rune）,反引号``中的内容表示不可转义的字符串

关于iota，下面代码输出什么?

```go
func Test10(t *testing.T) {
    const (
        x = iota
        _
        y
        z = "pi"
        k
        p = iota
        q
    )
    fmt.Println(x, y, z, k, p, q)
}
```

输出

> 0 2 pi pi 5 6

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;

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，而是会和新请求锁的 goroutine 竞争锁的拥有。新请求锁的 goroutine 具有优势：它正在 CPU上执行，而且可能有好几个，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。如果一个等待的 goroutine 超过1ms没有获取锁，那么它将会把锁转变为饥饿模式。

饥饿模式(公平锁)

为了解决了等待 G队列的长尾问题

饥饿模式下，直接由 unlock 把锁交给等待队列中排在第一位的 G(队头)，同时，饥饿模式下，新进来的 G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 g 一直抢不到锁的场景。饥饿模式的触发条件，当一个 G等待锁时间超过1 毫秒时，或者当前队列只剩下一个 g 的时候，Mutex切换到饥饿模式。

总结

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，其实是性能和公平的一个平衡模式。

Mutex 允许自旋的条件

1 锁已被占用，并且锁不处于饥饿模式。

2 积累的自旋次数小于最大自旋次数（active_spin=4）。3 cpu 核数大于1。

4 有空闲的 P。

5 当前 goroutine 所挂载的 P下，本地待运行队列为空。

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写锁之后释放通知读锁。同样的写锁也会等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。

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 Lock 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。

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。

Broadcast 和 Signal 区别

```go
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。

```go
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。

Cond 中 Wait 使用

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

Wait()会自动释放 c.L，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。）

```go
c.L.Lock()

for !condition(){

  c.Wait()

}

... make use of condition ...

c.L.Unlock()
```

WaitGroup 用法

一个 WaitGroup 对象可以等待一组协程结束。使用方法是：

1.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；

2.worker 协程执行结束以后，都要调用 wg.Done()；

3.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。

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，请求计数器减1，v 为0 时通过信号量唤醒 Wait()。

下面代码输出什么？

```
func Test16(t *testing.T) {
    a := [2]int{5, 6}
    b := [3]int{5, 6}
    if a == b {
        fmt.Println("equal")
    } else {
        fmt.Println("not equal")
    }
}
```

A. compilation error

B. equal

C. not equal

答案

> A 编译错误
> 对于数组而言，一个数组是由数组中的值和数组的长度两部分组成的，如果两个数组长度不同，那么两个数组是属于
> 不同类型的，是不能进行比较的

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。

什么操作叫做原子操作

一个或者多个操作在 CPU执行过程中不被中断的特性，称为原子性(atomicity)。这些操作对外表现成一个不可分割的整体，他们要么都执行，要么都不执行，外界不会看到他们只执行到一半的状态。而在现实世界中，CPU 不可能不中断的执行一系列操作，但如果我们在执行多个操作时，能让他们的中间状态对外不可见，那我们就可以宣城他们拥有了“不可分割”的原子性。

在 Go中，一条普通的赋值语句其实不是一个原子操作。列如，在32 位机器上写 int64 类型的变量就会有中间状态，因为他会被　　两次写操作(MOV)——写低32 位和写高32 位。

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。

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。

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。

Goroutine 定义

Goroutine 是一个与其他 goroutines 并行运行在同一地址空间的 Go 函数或方法。一个运行的程序由一个或更多个 goroutine 组成。它与线程、协程、进程等不同。它是一个 goroutine"—— Rob Pike

Goroutines 在同一个用户地址空间里并行独立执行 functions，channels 则用于 goroutines 间的通信和同步访问控制。

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。

给大家丢脸了，用了三年golang，我还是没答对这道内存泄漏题

https://mp.weixin.qq.com/s/-agtdhlW7Yj7S88a0z7KHg (https://mp.weixin.qq.com/s/-agtdhlW7Yj7S88a0z7KHg)

你一定会遇到的内存回收策略导致的疑似内存泄漏的问题

https://colobu.com/2019/08/28/go-memory-leak-i-dont-think-so/ (https://colobu.com/2019/08/28/go-memory-leak-i-dont-think-so/)

GMP里为什么要有P?

https://mp.weixin.qq.com/s/SEE2TUeZQZ7W1BKkmnelAA (https://mp.weixin.qq.com/s/SEE2Tl QZ7W1BKkmnelAA)

go栈扩容和栈缩容，连续栈的缺点

https://segmentfault.com/a/1190000019570427 (https://segmentfault.com/a/1190000019570427)

golang隐藏技能:怎么访问私有成员

https://www.jianshu.com/p/7b3638b47845 (https://www.jianshu.com/p/7b3638b47845)

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。

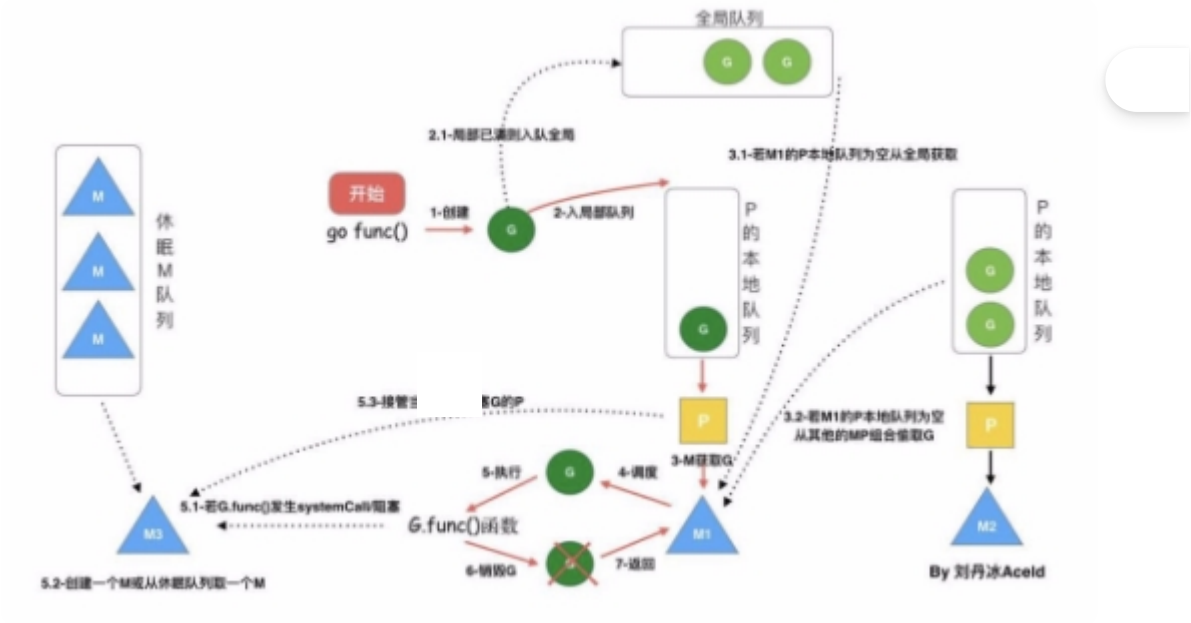
1．单一全局互斥锁(Sched.Lock)和集中状态存储

2．Goroutine 传递问题（M 经常在 M 之间传递"可运行"的 goroutine）

3．每个 M做内存缓存，导致内存占用过高，数据局部性较差

4．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。

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 hand

off，并寻找新的 idle 的 M，若没有 idle 的 M就会新建一个 M(流程5.1)。

- 当 G因 channel 或者 network I/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)

GMP 中 work stealing 机制

存到 P本地队列或者是全局队列。P此时去唤醒一个 M。P继续执行它的执行序。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)。

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。当发生上线文切换时，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)。

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，存在问题

- 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。

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 来强这个 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，宝贵的 P资源就这么被阻塞的 M浪费了。

GMP 调度过程中存在哪些阻塞

- I/O，select
- block on syscall
- channel
- 等待锁
- runtime.Gosched()

sysmon 有什么作用

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 超过10ms的 g，会进行 retake)；
- 收回因 syscall 长时间阻塞的 P；

golang面试题：怎么避免内存逃逸？

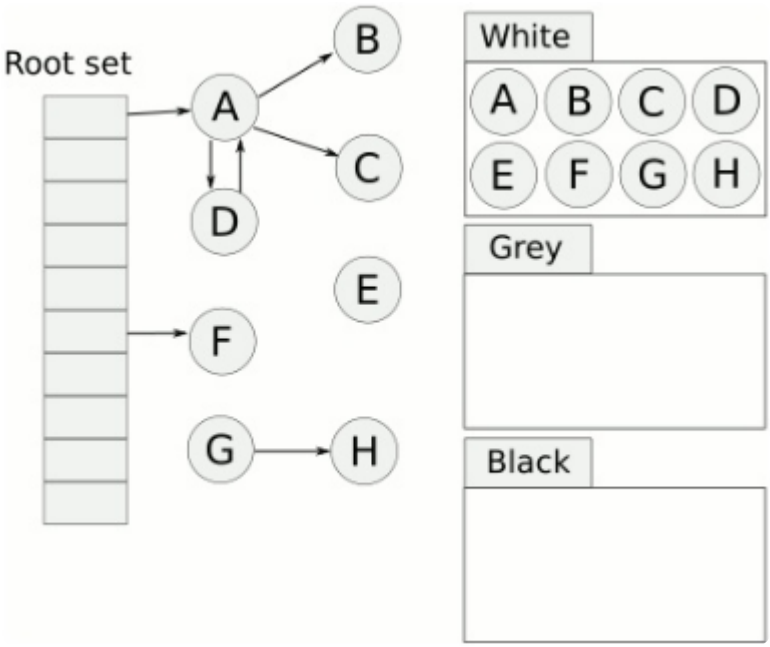
https://mp.weixin.qq.com/s/VzRTHz1JaDUvNRVB_yJa1A
(https://mp.weixin.qq.com/s/VzRTHz1JaDUvNRVB_yJa1A)

golang面试题：简单聊聊内存逃逸？

https://mp.weixin.qq.com/s/wJmztRMB1ZAAlltyMcS0tw
(https://mp.weixin.qq.com/s/wJmztRMB1ZAAlltyMcS0tw)

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，知道灰色集合中没有
- 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

插入写屏障

golang 的回收没有混合屏障之前，一直是插入写屏障，由于栈赋值没有 hook 的原因，所以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，所以后面 golang 是引用混合写屏障解决这个问题。混合写屏障之后，就没有 STW。

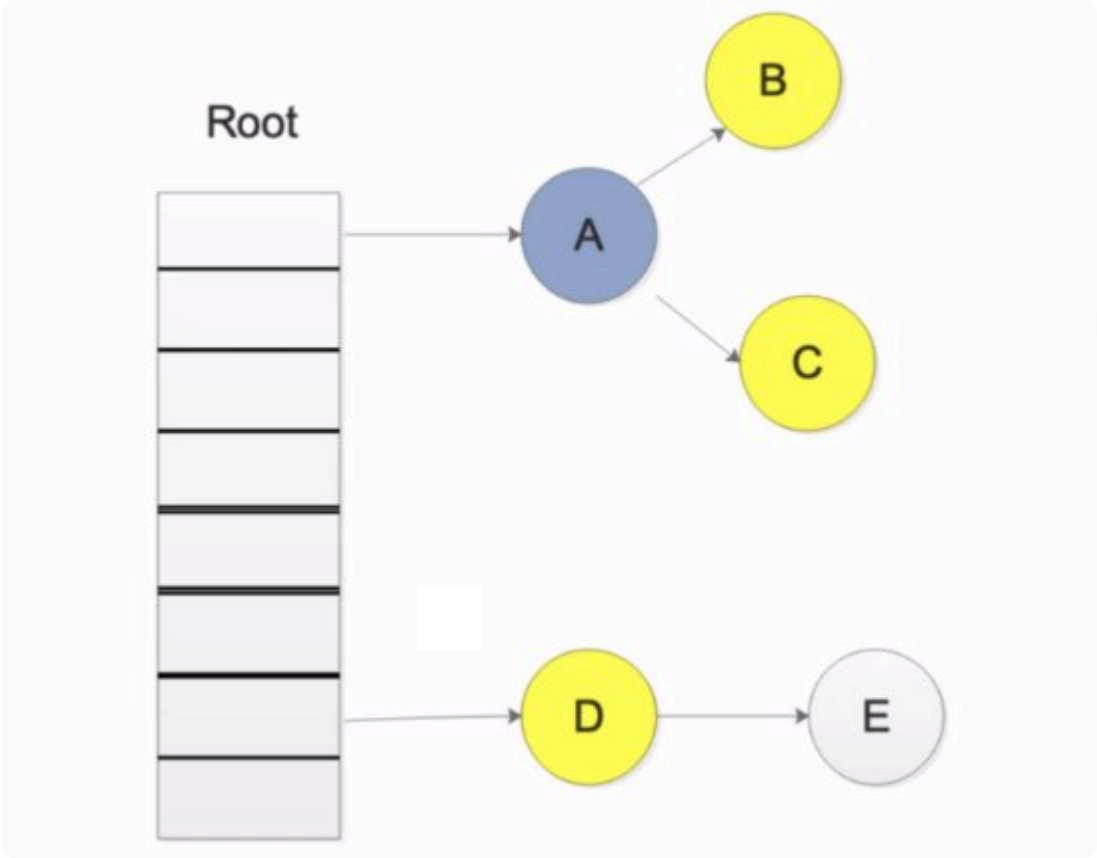
删除写屏障

goalng 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 中被清理掉。

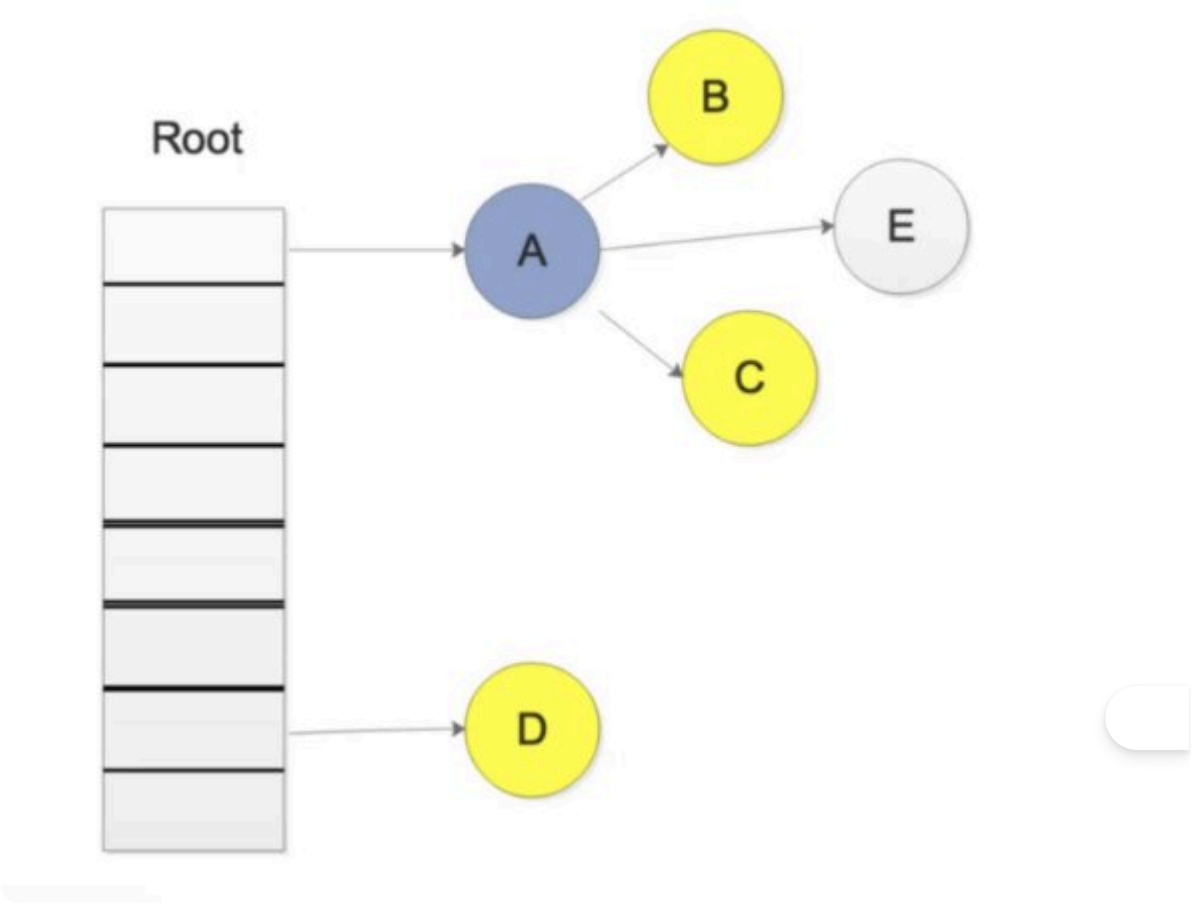
写屏障

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 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。

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 赋值器的工作的哈（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）。

GC 触发时机

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.由 GOGC 控制,默认100 即2 倍的关系,200 就是3 倍,

当 Go新创建的对象所占用的内存大小，除以上次 GC结束后保留下来的对象占用内存大小。

Go 语言中 GC 的流程是什么？

当前版本的 Go 以 STW 为界限，可以 GC 划分为五个阶段：阶段说明赋值器状态 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 STWGCoff内存清扫阶段，将需要回收的内存归还到堆中，写屏障关闭并发 GCoff内存归还阶段，将过多的内存归还给操作系统，写屏障关闭并发。

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。

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。

`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，厂 '' 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。

Goroutine和Channel的作用分别是什么？

进程是内存资源管理和cpu调度的执行单元。为了有效利用多核处理器的优势，将进程进一步细分，允许一个进程里存在多个线程，这多个线程还是共享同一片内存空间，但cpu调度的最小单元变成了线程。

那协程又是什么呢，以及与线程的差异性??

协程，可以看作是轻量级的线程。但与线程不同的是，线程的切换是由操作系统控制的，而协程的切换则是由用户控制的。

最早支持协程的程序语言应该是lisp方言scheme里的continuation（续延），续延允许scheme保存任意函数调用的现场，保存起来并重新执行。Lua,C#,python等语言也有自己的协程实现。

Go中的goroutinue就是协程,可以实现并行，多个协程可以在多个处理器同时跑。而协程同一时刻只能在一个处理器上跑（可以把宿主语言想象成单线程的就好了）。 然而,多个goroutine之间的通信是通过channel，而协程的通信是通过yield和resume()操作。

goroutine非常简单，只需要在函数的调用前面加关键字go即可，例如:

```text
go elegance()
```

我们也可以启动5个goroutines分别打印索引。

```text
func main() {
    for i:=1;i<5;i++ {
        go func(i int) {
            fmt.Println(i)
        }(i)
    }
    // 停歇5s，保证打印全部结束
    time.Sleep(5*time.Second)
}
```

在分析goroutine执行的随机性和并发性，启动了5个goroutine，再加上main函数的主goroutine，总共有6个goroutines。由于goroutine类似于"守护线程"，异步执行的,如果主goroutine不等待片刻，可能程序就没有输出打印了。

在Golang中channel则是goroutinues之间进行通信的渠道。

可以把channel形象比喻为工厂里的传送带,一头的生产者goroutine往传输带放东西,另一头的消费者goroutinue则从输送带取东西。channel实际上是一个有类型的消息队列,遵循先进先出的特点。

1. channel的操作符号

ch <- data 表示data被发送给channel ch；

data <- ch 表示从channel ch取一个值，然后赋给data。

1. 阻塞式channel

channel默认是没有缓冲区的，也就是说，通信是阻塞的。send操作必须等到有消费者accept才算完成。

应用示例:

```text
func main() {
    ch1 := make(chan int)
    go pump(ch1) // pump hangs
    fmt.Println(<-ch1) // prints only 1
}

func pump(ch chan int) {
    for i:= 1; ; i++ {
        ch <- i
    }
}
```

在函数pump()里的channel在接受到第一个元素后就被阻塞了，直到主goroutinue取走了数据。最终channel阻塞在接受第二个元素，程序只打印 1。

没有缓冲(buffer)的channel只能容纳一个元素，而带有缓冲(buffer)channel则可以非阻塞容纳N个元素。发送数据到缓冲(buffer) channel不会被阻塞，除非channel已满；同样的，从缓冲(buffer) channel取数据也不会被阻塞，除非channel空了。

怎么查看Goroutine的数量?

GOMAXPROCS中控制的是未被阻塞的所有Goroutine,可以被Multiplex到多少个线程上运行,通过GOMAXPROCS可以查看Goroutine的数量。

相关推荐

- 还没有offer?专注【突击面试】的【go后端开发】训练营了解一下
- golang进阶面试题八股文合集(https://golangguide.top/golang/%E9%9D%A2%E8%AF%95%E9%A2%98/2.Go%E8%BF%9B%E9%98%B6.html)

- golang常用标准库第三方库大全 (https://golangguide.top/golang/%E5%B8%B8%E7%94%A8%E5%8C%85%E5%A4%A7%E5%85%A8.html)
- golang学习路线 (https://golangguide.top/golang/%E5%AD%A6%E4%B9%A0%E8%B7%AF%E7%BA%BF.html)

上次编辑于: 2024/5/26 07:49:28
贡献者: xiaobai-tech