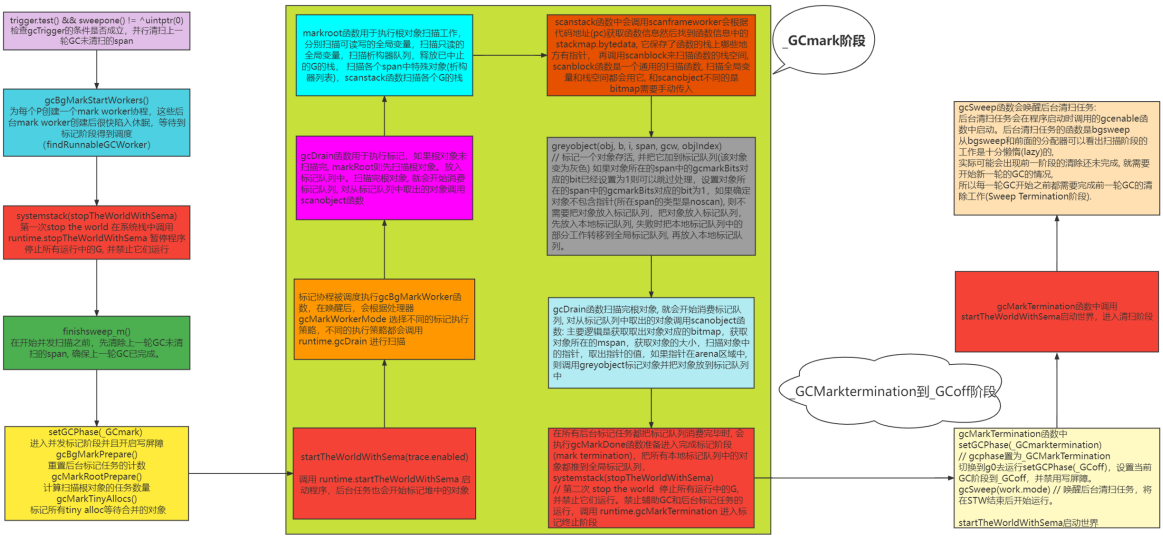
浅析GO语言的GC

我就不从标记-清除，标记-整理，演进过程这样开始讲了。直接从GO语言垃圾收集的实现原理开讲吧。

整体流程

先从整体上看一下垃圾收集的大体流程



接下来会讲解上图的部分函数

入口函数在runtime.gcStart() // mgc.go 571

垃圾收集的多个阶段

1、清理终止阶段

这个阶段会调用trriger.test()函数检查gcTrigger条件是否成立，调用sweepone函数清理上一轮GC没有清扫完的mspan，并在gcBgMarkStartWorkers()函数中创

建好后台标记的协程并让它们陷入休眠，等待唤醒去执行标记工作。执行systemstack(stopTheWorldWithSema) // 停止所有运行中的G。调用finishsweep_m()

清扫上一轮GC未清扫的mspan。

2、标记阶段

```
setGCPhase(_GCmark) // 进入并发标记阶段并且开启写屏障
gcBgMarkPrepare() // 重置后台标记任务的计数
gcMarkRootPrepare() // 计算扫描根对象的任务数量
gcMarkTinyAllocs() // 标记所有tiny alloc等待合并的对象
atomic.Store(&gcBlackenEnabled, 1) // 设置全局变量 runtime.gcBlackenEnabled，用户程序和标记任务可以将对象涂黑。
startTheWorldWithSema(trace.enabled) // 调用 runtime.startTheWorldWithSema 启动程序，后台任务也会开始标记堆中的对象
```

标记协程被调度执行gcBgMarkWorker函数，在唤醒后，会根据处理器 gcMarkWorkerMode 选择不同的标记执行策略，不同的执行策略都会调用

runtime.gcDrain 进行扫描。gcDrain函数用于执行标记，如果根对象未扫描完, markRoot则先扫描根对象。放入标记队列中。扫描完根对象, 就会开始消费标记队

列，对从标记队列中取出的对象调用scanobject函数。markroot函数用于执行根对象扫描工作，分别扫描可读写的全局变量，扫描只读的全局变量，扫描析构器

队列，释放已中止的G的栈， 扫描各个span中特殊对象(析构器列表)，scanstack函数扫描各个G的栈。scanstack函数中会调用scanframeworker会根据代码地址

(pc)获取函数信息然后找到函数信息中的stackmap.bytedata, 它保存了函数的栈上哪些地方有指针， 再调用scanblock来扫描函数的栈空间, scanblock函数是一个

通用的扫描函数, 扫描全局变量和栈空间都会用它, 和scanobject不同的是bitmap需要手动传入。greyobject(obj, b, i, span, gcw, objIndex) 标记一个对象存活, 并

把它加到标记队列(该对象变为灰色) 如果对象所在的span中的gcmarkBits对应的bit已经设置为1则可以跳过处理，设置对象所在的span中的gcmarkBits对应的bit

为1，如果确定对象不包含指针(所在span的类型是noscan), 则不需要把对象放入标记队列，把对象放入标记队列，先放入本地标记队列, 失败时把本地标记队列中

的部分工作转移到全局标记队列, 再放入本地标记队列。gcDrain函数扫描完根对象, 就会开始消费标记队列, 对从标记队列中取出的对象调用scanobject函数: 主要

逻辑是获取取出对象对应的bitmap，获取对象所在的mspan，获取对象的大小，扫描对象中的指针，取出指针的值，如果指针在arena区域中, 则调用greyobject

标记对象并把对象放到标记队列中。在所有后台标记任务都把标记队列消费完毕时, 会执行gcMarkDone函数准备进入完成标记阶段(mark termination)，把所有本

地标记队列中的对象都推到全局标记队列，systemstack(stopTheWorldWithSema) 第二次 stop the world 停止所有运行中的G, 并禁止它们运行。禁止辅助GC和

后台标记任务的运行，调用 runtime.gcMarkTermination 进入标记终止阶段。

3、标记终止阶段

gcMarkTermination函数中setGCPhase(_ GCmarktermination) // gcphase置为_GCMarkTermination

切换到g0去运行setGCPhase(_ GCoff)，设置当前GC阶段到_GCoff，并禁用写屏障。

gcSweep(work.mode) // 唤醒后台清扫任务，将在STW结束后开始运行。

startTheWorldWithSema //启动世界

4、清理阶段

gcSweep函数会唤醒后台清扫任务

后台清扫任务会在程序启动时调用的gcenable函数中启动。后台清扫任务的函数是bgsweep

从bgsweep和前面的分配器可以看出扫描阶段的工作是十分懒惰(lazy)的，实际可能会出现前一阶段的清除还未完成, 就需要开始新一轮的GC的情况。

所以每一轮GC开始之前都需要完成前一轮GC的清除工作(Sweep Termination阶段)。

详细流程分析

触发时机

运行时会通过如下所示的 runtime.gcTrigger.test方法决定是否需要触发垃圾收集，当满足触发垃圾收集的基本条件时 — 允许垃圾收集、程序没有崩溃并且

没有处于垃圾收集循环，该方法会根据三种不同方式触发进行不同的检查。

```
func (t gcTrigger) test() bool {
    if !memstats.enablegc || panicking != 0 || gcphase != _GCoff {
        return false
    }
    switch t.kind {
    case gcTriggerHeap:
        // Non-atomic access to gcController.heapLive for performance. If
        // we are going to trigger on this, this thread just
        // atomically wrote gcController.heapLive anyway and we'll see our
        // own write.
        return gcController.heapLive >= gcController.trigger
    case gcTriggerTime:
        if gcController.gcPercent < 0 {
            return false
        }
        lastgc := int64(atomic.Load64(&memstats.last_gc_nanotime))
        return lastgc != 0 && t.now-lastgc > forcegcperiod
    case gcTriggerCycle:
        // t.n > work.cycles, but accounting for wraparound.
        return int32(t.n-work.cycles) > 0
    }
    return true
}
```

gcTriggerHeap

堆内存的分配达到控制器计算的触发堆大小

在内存管理中，分析了mallocgc这个函数，给对象分配内存需要用到这个函数，这个函数会在向上一级内存管理组件申请mspan之后，将shouldhelpgc置为true

我们来到mallocgc这个函数

```
// Allocate a new maxTinySize block.  来到这里说明当前spanclass=2就是存储大小为16B的块没有合适的空闲内存存放要分配的对象，新分配一个16B的块。
span = c.alloc[tinySpanClass]
v := nextFreeFast(span) // 从当前mspan中找可以存下对象的空闲内存
if v == 0 { // 如果从当前span中没有找到空闲内存，调用nextFree再次判断当前mspan还有可用空间没有，没有就向中心缓存申请新的有可用空间的mspan.
    v, span, shouldhelpgc = c.nextFree(tinySpanClass) // nextFree会将原来满了的mspan归还给mcentral，新分配一个mspan给mcache管理。
} // 来到这里v肯定不为0
x = unsafe.Pointer(v)
(*[2]uint64)(x)[0] = 0
(*[2]uint64)(x)[1] = 0
// See if we need to replace the existing tiny block with the new one
// based on amount of remaining free space.
if !raceenabled && (size < c.tinyoffset || c.tiny == 0) { // 移动tinyalloctor的指针
    // Note: disabled when race detector is on, see comment near end of this function.
    c.tiny = uintptr(x)
    c.tinyoffset = size
}
size = maxTinySize
} else { // 否则按普通的小对象分配 首先获取对象的大小应该使用哪个span类型，也有可能是指针类型的对象。
var sizeclass uint8
if size <= smallSizeMax-8 {
    sizeclass = size_to_class8[divRoundUp(size, smallSizeDiv)]
} else {
    sizeclass = size_to_class128[divRoundUp(size-smallSizeMax, largeSizeDiv)]
}
size = uintptr(class_to_size[sizeclass]) // 确定分配对象的大小以及跨度类 runtime.spanClass；
spc := makeSpanClass(sizeclass, noscan)
span = c.alloc[spc]
v := nextFreeFast(span) // runtime.nextFreeFast 会利用内存管理单元mspan中的 allocCache字段，快速找到该字段为 1 的位数，我们在上面介绍过 1 表示该位对应的内存
if v == 0 {
    v, span, shouldhelpgc = c.nextFree(spc) // 如果我们没有找到空闲的内存，运行时会通过 runtime.mcache.nextFree 找到新的内存管理单元
}
x = unsafe.Pointer(v)
if needzero && span.needzero != 0 {
    memclrNoHeapPointers(unsafe.Pointer(v), size)
```

进入nextFree函数

```
func (c *mcache) nextFree(spc spanClass) (v gclinkptr, s *mspan, shouldhelpgc bool) {
    s = c.alloc[spc]
    shouldhelpgc = false
    freeIndex := s.nextFreeIndex()
    if freeIndex == s.nelems { // 没有空闲空间了 如果span里面所有元素都已分配，则需要获取新的span
        // The span is full.
        if uintptr(s.allocCount) != s.nelems {
            println( args...: "runtime: s.allocCount=", s.allocCount, "s.nelems=", s.nelems)
            throw( s: "s.allocCount != s.nelems && freeIndex == s.nelems")
        }
        c.refill(spc) // 当前这个s(*mspan)指向的mspan所管理的内存中，存不下新的对象了。调用refill去中心缓存中去取mspan
        shouldhelpgc = true // 将需要协助GC标记置为true
        s = c.alloc[spc] // 找到我们新添加的mspan

        freeIndex = s.nextFreeIndex() // 返回mspan中下一个可以存放对象的索引位置。
    }

    if freeIndex >= s.nelems {
        throw( s: "freeIndex is not valid")
    }
    // 可以分配给新对象的内存起始地址
    v = gclinkptr(freeIndex*s.elemsize + s.base())
    s.allocCount++ // 添加已分配的元素计数
    if uintptr(s.allocCount) > s.nelems {
        println( args...: "s.allocCount=", s.allocCount, "s.nelems=", s.nelems)
        throw( s: "s.allocCount > s.nelems")
```

在mallocgc函数后面部分有一段逻辑就是去判断是否要开始gc标记过程的。

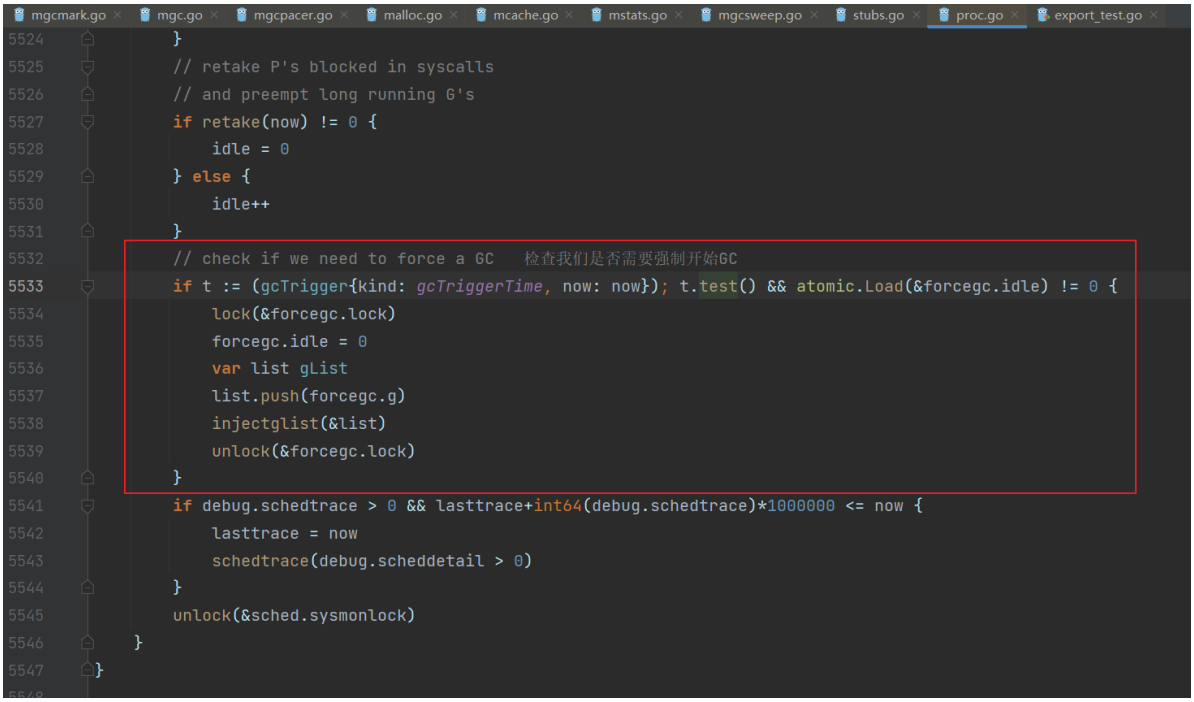
```
            inittrace.bytes += uint64(size)
        }
    }

    if assistG != nil {
        // Account for internal fragmentation in the assist
        // debt now that we know it.
        assistG.gcAssistBytes -= int64(size - dataSize)
    }
    // 调用gcTrigger.test()去看gc认为的活动字节数是否大于等于了触发标记时的堆内存大小，也就是不一定会开始标记，得看是否满足条件。
    if shouldhelpgc {
        if t := (gcTrigger{kind: gcTriggerHeap}); t.test() {
            gcStart(t)
        }
    }
```

进入gcTrigger.test方法若满足 gcController.heapLive >= gcController.trigger 那么就会执行gcStart开始GC过程。

gcTriggerTime

如果一定时间内没有触发，就会触发新的循环，该触发条件由 runtime.forcegcperiod变量控制，默认为 2 分钟。

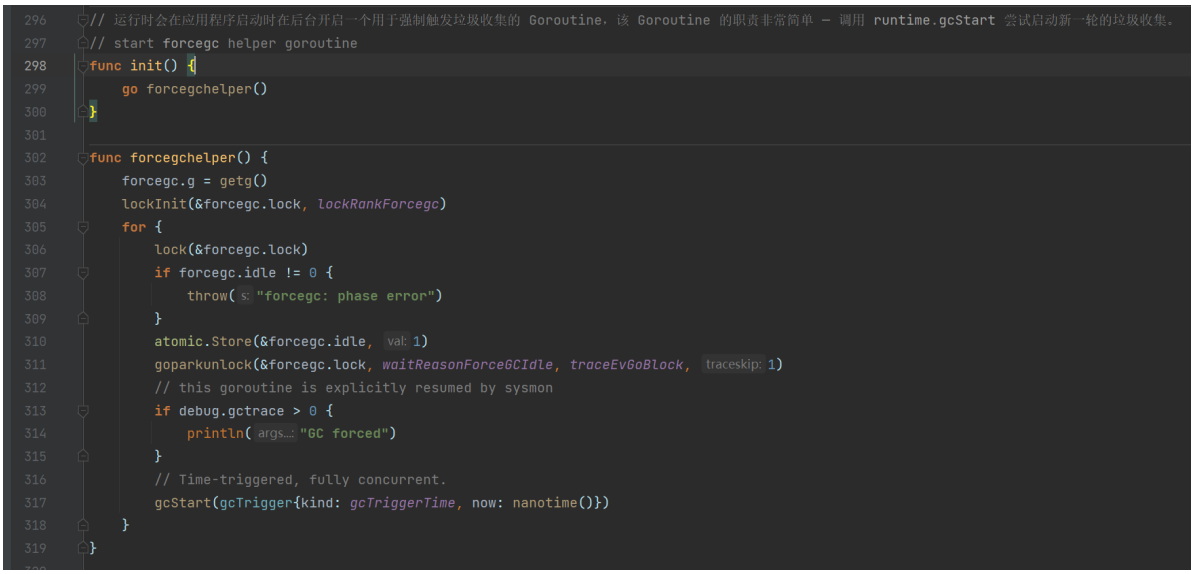
在系统监控线程执行的函数sysmon中有一段检查是否开始GC的逻辑。



再次进入gcTrigger.test()，如果离上一次GC已经过去了2分钟，那么就会强制开始GC。

运行时会在应用程序启动时在后台开启一个用于强制触发垃圾收集的 Goroutine，该 Goroutine 的职责非常简单。

调用 runtime.gcStart尝试启动新一轮的垃圾收集。



为了减少对计算资源的占用，该 Goroutine 会在循环中调用 runtime.goparkunlock主动陷入休眠等待其他 Goroutine 的唤醒，runtime.forcegchelper在

大多数时间都是陷入休眠的，但是它会被系统监控器 runtime.sysmon在满足垃圾收集条件时唤醒。

正如前面sysmon函数中，系统监控在每个循环中都会主动构建一个 runtime.gcTrigger并检查垃圾收集的触发条件是否满足，如果满足条件，系统监控会将

runtime.forcegc状态中持有的 Goroutine 加入全局队列等待调度器的调度。

gcTriggerCycle

要求启动新一轮的GC, 已启动则跳过, 手动触发GC的runtime.GC()会使用这个条件。

来到runtime.GC()

```
405 func GC() {
406     
407     n := atomic.Load(&work.cycles)
408     gcWaitOnMark(n)
409 
410     // We're now in sweep N or later. Trigger GC cycle N+1, which
411     // will first finish sweep N if necessary and then enter sweep
412     // termination N+1.
413     gcStart(gcTrigger{kind: gcTriggerCycle, n: n + 1})
414 
415     // Wait for mark termination N+1 to complete.
416     gcWaitOnMark(n + 1)
417 
418     // Finish sweep N+1 before returning. We do this both to
419     // complete the cycle and because runtime.GC() is often used
420     // as part of tests and benchmarks to get the system into a
421     // relatively stable and isolated state.
422     for atomic.Load(&work.cycles) == n+1 && sweepone() != ^uintptr(0) {
423         sweep.nbgsweep++
424         Gosched()
425     }
```

在正式开始垃圾收集前，运行时需要通过 runtime.gcWaitOnMark 等待上一个循环的标记终止、标记和清除终止阶段完成；

调用 runtime.gcStart 触发新一轮的垃圾收集并通过 runtime.gcWaitOnMark 等待该轮垃圾收集的标记终止阶段正常结束；

持续调用 runtime.sweepone 清理全部待处理的内存管理单元并等待所有的清理工作完成，等待期间会调用 runtime.Gosched让出处理器；

完成本轮垃圾收集的清理工作后，通过 runtime.mProf_PostSweep 将该阶段的堆内存状态快照发布出来，我们可以获取这时的内存状态。

垃圾收集启动

垃圾收集在启动过程一定会调用 runtime.gcStart，虽然该函数的实现比较复杂，但是它的主要职责是修改全局的垃圾收集状态到 _GCmark并做一些准备工

作，我们会分以下几个阶段介绍该函数的实现：

1. 两次调用 runtime.gcTrigger.test 检查是否满足垃圾收集条件，清扫上一轮GC未清扫完的span；
2. 在后台启动用于处理标记任务的工作 Goroutine、暂停程序确定所有内存管理单元都被清理以及其他标记阶段开始前的准备工作；
3. 进入标记阶段、准备后台的标记工作、根对象的标记工作以及微对象、恢复用户程序，进入并发扫描和标记阶段；

处理上一轮GC的遗留工作

```go
func gcStart(trigger gcTrigger) {
    // Since this is called from malloc and malloc is called in
    // the guts of a number of libraries that might be holding
    // locks, don't attempt to start GC in non-preemptible or
    // potentially unstable situations.
    mp := acquirem()
    if gp := getg(); gp == mp.g0 || mp.locks > 1 || mp.preemptoff != "" {
        releasem(mp)
        return
    }
    releasem(mp)
    mp = nil

    // Pick up the remaining unswept/not being swept spans concurrently
    //
    // This shouldn't happen if we're being invoked in background
    // mode since proportional sweep should have just finished
    // sweeping everything, but rounding errors, etc, may leave a
    // few spans unswept. In forced mode, this is necessary since
    // GC can be forced at any point in the sweeping cycle.
    //
    // We check the transition condition continuously here in case
    // this G gets delayed in to the next GC cycle.
    for trigger.test() && sweepone() != ^uintptr(0) { // 并行清扫上一轮GC未清扫的span
        sweep.nbgsweep++
    }

    // Perform GC initialization and the sweep termination
    // transition.
    semacquire(&work.startSema) // 上锁，然后重新检查gcTrigger的条件是否成立，不成立时不触发GC。
    // Re-check transition condition under transition lock.
    if !trigger.test() { // gcTrigger条件不成立，进入if解锁之后直接返回。
        semrelease(&work.startSema)
        return
    }
```

验证垃圾收集条件的同时，该方法还会在循环中不断调用 runtime.sweepone 清理已经被标记的内存单元，完成上一个垃圾收集循环的收尾工作。

创建标记协程并做标记准备工作

调用gcBgMarkStartWorkers()函数，创建标记协程

```go
func gcBgMarkStartWorkers() {
    // Background marking is performed by per-P G's. Ensure that each P has
    // a background GC G.
    //
    // Worker Gs don't exit if gomaxprocs is reduced. If it is raised
    // again, we can reuse the old workers; no need to create new workers.
    for gcBgMarkWorkerCount < gomaxprocs {
        go gcBgMarkWorker() // 创建一个用于标记的协程
        // 启动后等待该任务通知信号量bgMarkReady再继续
        notetsleepg(&work.bgMarkReady, ns: -1)
        noteclear(&work.bgMarkReady)
        // The worker is now guaranteed to be added to the pool before
        // its P's next findRunnableGCWorker.

        gcBgMarkWorkerCount++
    }
}
```

创建之后，很快进入睡眠，等到标记阶段得到调度。

```
1197    for {
1198        // Go to sleep until woken by
1199        // gcController.findRunnableGCWorker.
1200        gopark(func(g *g, nodep unsafe.Pointer) bool { // 标记协程先睡眠，等待被唤醒。
1201            node := (*gcBgMarkWorkerNode)(nodep)
1202
1203            if mp := node.m.ptr(); mp != nil {
1204                // The worker G is no longer running; release
1205                // the M.
1206                //
```

接下来stop the world并做一些准备工作

```
655         traceGCSTWStart( kind: 1)
656     } // 第一次stop the world 在系统栈中调用 runtime.stopTheWorldWithSema 暂停程序
657     systemstack(stopTheWorldWithSema) // 停止所有运行中的G，并禁止它们运行
658     // Finish sweep before we start concurrent scan.
659     systemstack(func() {
660         finishsweep_m() // 清扫上一轮GC未清扫的span，确保上一轮GC已完成。
661     })
662     // 清扫sched.sudogcache和sched.deferpool
663     // clearpools before we start the GC. If we wait they memory will not be
664     // reclaimed until the next GC cycle.
665     clearpools()
666     // 增加GC计数
667     work.cycles++
668
669     gcController.startCycle()
670     work.heapGoal = gcController.heapGoal
671
672     // In STW mode, disable scheduling of user Gs. This may also
673     // disable scheduling of this goroutine, so it may block as
674     // soon as we start the world again.
675     if mode != gcBackgroundMode { // 判断是否并行GC模式
676         schedEnableUser( enable: false)
677     }
678
```

准备工作

```
693     setGCPhase(_GCmark) // 进入并发标记阶段并且开启写屏障
694     // 调用 runtime.gcBgMarkPrepare 初始化后台扫描需要的状态
695     gcBgMarkPrepare() // Must happen before assist enable. // 重置后台标记任务的计数
696     gcMarkRootPrepare() // 调用 runtime.gcMarkRootPrepare函数会计算扫描根对象的任务数量。
697
698     // Mark all active tinyalloc blocks. Since we're
699     // allocating from these, they need to be black like
700     // other allocations. The alternative is to blacken
701     // the tiny block on every allocation from it, which
702     // would slow down the tiny allocator.
703     gcMarkTinyAllocs() // gcMarkTinyAllocs函数会标记所有tiny alloc等待合并的对象
704
705     // At this point all Ps have enabled the write
706     // barrier, thus maintaining the no white to
707     // black invariant. Enable mutator assists to
708     // put back-pressure on fast allocating
709     // mutators. // 启用辅助GC
710     atomic.Store(&gcBlackenEnabled,  val: 1) // 设置全局变量 runtime.gcBlackenEnabled，用户程序和标记任务可以将对象涂黑。
711
712     // Assists and workers can start the moment we start
713     // the world.
714     gcController.markStartTime = now // 记录标记开始的时间
715
716     // In STW mode, we could block the instant systemstack
717     // returns, so make sure we're not preemptible.
718     mp = acquirem()
```

进入标记阶段

启动世界

```
721     systemstack(func() { // StartTheWorld，进入并发标记阶段，
722         now = startTheWorldWithSema(trace.enabled) // 调用 runtime.startTheWorldWithSema 启动程序，后台任务也会开始标记堆中的对象
723         work.pauseNS += now - work.pauseStart
724         work.tMark = now
725         memstats.gcPauseDist.record(now - work.pauseStart)
726     })
727
728     // Release the world sema before Gosched() in STW mode
729     // because we will need to reacquire it later but before
730     // this goroutine becomes runnable again, and we could
731     // self-deadlock otherwise.
732     semrelease(&worldsema)
733     releasem(mp)
734
735     // Make sure we block instead of returning to user code
736     // in STW mode.
737     if mode != gcBackgroundMode {
738         Gosched()
739     }
740
741     semrelease(&work.startSema)
742 }
```

调用 runtime.startTheWorldWithSema 启动程序，后台任务也会开始标记堆中的对象。

接下来就要开始标记的逻辑了，内容很长，首先标记协程被唤醒，继续执行gcBgMarkWorker函数的代码。

根据gcMarkWorkerMode选择不同的标记执行策略

```
1264             casgstatus(gp, _Grunning, _Gwaiting) // 设置G的状态为等待中这样它的栈可以被扫描(两个后台标记任务可以互相扫描对方的栈)
1265             switch pp.gcMarkWorkerMode { // 在唤醒后，我们会根据处理器 gcMarkWorkerMode 选择不同的标记执行策略，不同的执行策略都会调用 runtime.gcDrain
1266             default:                                                                                    //       扫描工作缓冲区 runtime.gcWork
1267                 throw("gcBgMarkWorker: unexpected gcMarkWorkerMode")
1268             case gcMarkWorkerDedicatedMode: // 这个模式下P应该专心执行标记
1269                 gcDrain(&pp.gcw, gcDrainUntilPreempt|gcDrainFlushBgCredit)
1270                 if gp.preempt { // 被抢占时把本地运行队列中的所有G都踢到全局运行队列
1271                     // We were preempted. This is
1272                     // a useful signal to kick
1273                     // everything out of the run
1274                     // queue so it can run
1275                     // somewhere else.
1276                     if drainQ, n := runqdrain(pp); n > 0 {
1277                         lock(&sched.lock)
1278                         globrunqputbatch(&drainQ, int32(n))
1279                         unlock(&sched.lock)
1280                     }
1281                 }
1282                 // Go back to draining, this time
1283                 // without preemption.
1284                 gcDrain(&pp.gcw, gcDrainFlushBgCredit) // 继续执行标记，直到无更多任务，并且需要计算后台的扫描量来减少辅助GC和唤醒等待中的G
1285             case gcMarkWorkerFractionalMode: // 这个模式下P应该适当执行标记
1286                 gcDrain(&pp.gcw, gcDrainFractional|gcDrainUntilPreempt|gcDrainFlushBgCredit)
1287             case gcMarkWorkerIdleMode: // 这个模式下P只在空闲时执行标记
1288                 gcDrain(&pp.gcw, gcDrainIdle|gcDrainUntilPreempt|gcDrainFlushBgCredit)
1289             }
1290             casgstatus(gp, _Gwaiting, _Grunning) // 恢复G的状态到运行中
1291         })
```

用于并发扫描对象的工作协程 Goroutine 总共有三种不同的模式 runtime.gcMarkWorkerMode，这三种不同模式的 Goroutine 在标记对象时使用完全不同的策

略，垃圾收集控制器会按照需要执行不同类型的工作协程。

- gcMarkWorkerDedicatedMode — 处理器专门负责标记对象，不会被调度器抢占；
- gcMarkWorkerFractionalMode — 当垃圾收集的后台 CPU 使用率达不到预期时（默认为 25%），启动该类型的工作协程帮助垃圾收集达到利用率的目标，因

  为它只占用同一个 CPU 的部分资源，所以可以被调度；
- gcMarkWorkerIdleMode — 当处理器没有可以执行的 Goroutine 时，它会运行垃圾收集的标记任务直到被抢占；

runtime.gcControllerState.startCycle会根据全局处理器的个数以及垃圾收集的 CPU 利用率计算出上述的 dedicatedMarkWorkersNeeded和

fractionalUtilizationGoal以决定不同模式的工作协程的数量。

因为后台标记任务的 CPU 利用率为 25%，如果主机是 4 核或者 8 核，那么垃圾收集需要 1 个或者 2 个专门处理相关任务的 Goroutine；不过如果主机是 3 核或者

6 核，因为无法被 4 整除，所以这时需要 0 个或者 1 个专门处理垃圾收集的 Goroutine，运行时需要占用某个 CPU 的部分时间，使用

gcMarkWorkerFractionalMode 模式的协程保证 CPU 的利用率。

根据处理器 gcMarkWorkerMode 选择不同的标记执行策略，不同的执行策略都会调用 runtime.gcDrain 扫描。

gcDrain

gcDrain函数用于执行标记

```
func gcDrain(gcw *gcWork, flags gcDrainFlags) {
    if !writeBarrier.needed {
        throw("gcDrain phase incorrect")
    }

    gp := getg().m.curg
    preemptible := flags&gcDrainUntilPreempt != 0    // 看到抢占标志时是否要返回
    flushBgCredit := flags&gcDrainFlushBgCredit != 0 // 是否计算后台的扫描量来减少辅助GC和唤醒等待中的G
    idle := flags&gcDrainIdle != 0                   // 是否只执行一定量的工作

    initScanWork := gcw.scanWork // 记录初始的已扫描数量

    // checkWork is the scan work before performing the next
    // self-preempt check.
    checkWork := int64(1<<63 - 1)
    var check func() bool
    if flags&(gcDrainIdle|gcDrainFractional) != 0 {
        checkWork = initScanWork + drainCheckThreshold
        if idle {
            check = pollWork
        } else if flags&gcDrainFractional != 0 {
            check = pollFractionalWorkerExit
        }
    }
    // 如果根对象未扫描完，则先扫描根对象。
    // Drain root marking jobs.
    if work.markrootNext < work.markrootJobs { // markrootJobs 是之前计算的根对象任务数
        // Stop if we're preemptible or if someone wants to STW.
        for !(gp.preempt && (preemptible || atomic.Load(&sched.gcwaiting) != 0)) { // 如果标记了preemptible，循环直到被抢占
            job := atomic.Xadd(&work.markrootNext, +1) - 1 // 从根对象扫描队列取出一个值(原子递增)
            if job >= work.markrootJobs {
                break
            }
            markroot(gcw, job)            // 扫描根对象
            if check != nil && check() { // 如果是idle模式并且有其他工作，则返回
```

```
                goto done
            }
        }
    }
    // 根对象已经在标记队列中，消费标记队列  如果标记了preemptible，循环直到被抢占
    // Drain heap marking jobs.
    // Stop if we're preemptible or if someone wants to STW.
    for !(gp.preempt && (preemptible || atomic.Load(&sched.gcwaiting) != 0)) {
        // Try to keep work available on the global queue. We used to
        // check if there were waiting workers, but it's better to
        // just keep work available than to make workers wait. In the
        // worst case, we'll do O(log(_WorkbufSize)) unnecessary
        // balances.
        if work.full == 0 {
            gcw.balance() // 如果全局标记队列为空，把本地标记队列的一部分工作分过去 (如果wbuf2不为空则移动wbuf2过去，否则看wbuf1中是否有4个任务，有就移动一半过去)
        }

        b := gcw.tryGetFast()
        if b == 0 {
            b = gcw.tryGet()
            if b == 0 {
                // Flush the write barrier
                // buffer; this may create
                // more work.
                wbBufFlush(nil, 0)
                b = gcw.tryGet()
            }
        }
        if b == 0 { // 获取不到对象，标记队列已为空，跳出循环
            // Unable to get work.
            break
        }
        scanobject(b, gcw) // gcDrain函数扫描完根对象，就会开始消费标记队列，对从标记队列中取出的对象调用scanobject函数。
        // 如果已经扫描了一定数量的对象(gcCreditSlack的值是2000)
        // Flush background scan work credit to the global
        // account if we've accumulated enough locally so
        // mutator assists can draw on it.
        if gcw.scanWork >= gcCreditSlack {
            atomic.Xaddint64(&gcController.scanWork, gcw.scanWork) // 把扫描的对象数量添加到全局
            if flushBgCredit { // 减少辅助GC的工作量和唤醒等待中的G
                gcFlushBgCredit(gcw.scanWork - initScanWork)
                initScanWork = 0
            }
            checkWork -= gcw.scanWork
            gcw.scanWork = 0

            if checkWork <= 0 {
                checkWork += drainCheckThreshold
                if check != nil && check() {
                    break
                }
            }
        }
    }
```

```
done:
    // Flush remaining scan work credit.
    if gcw.scanWork > 0 { // 把扫描的对象数量添加到全局
        atomic.Xaddint64(&gcController.scanWork, gcw.scanWork)
        if flushBgCredit { // 减少辅助GC的工作量和唤醒等待中的G
            gcFlushBgCredit(gcw.scanWork - initScanWork)
        }
        gcw.scanWork = 0
    }
}
```

1、markroot

markroot函数用于执行根对象扫描工作

```
func markroot(gcw *gcWork, i uint32) {
    // Note: if you add a case here, please also update heapdump.go:dumproots.
    switch {
    case work.baseData <= i && i < work.baseBSS: // 扫描可读写的全局变量
        for _, datap := range activeModules() {
            markrootBlock(datap.data, datap.edata-datap.data,
datap.gcdatamask.bytedata, gcw, int(i-work.baseData))
        }
    // 扫描只读的全局变量
    case work.baseBSS <= i && i < work.baseSpans:
        for _, datap := range activeModules() {
            markrootBlock(datap.bss, datap.ebss-datap.bss,
datap.gcbssmask.bytedata, gcw, int(i-work.baseBSS))
        }
    // 扫描析构器队列
    case i == fixedRootFinalizers:
        for fb := allfin; fb != nil; fb = fb.alllink {
            cnt := uintptr(atomic.Load(&fb.cnt))
            scanblock(uintptr(unsafe.Pointer(&fb.fin[0])),
cnt*unsafe.Sizeof(fb.fin[0]), &finptrmask[0], gcw, nil)
        }
    // 释放已中止的G的栈
    case i == fixedRootFreeGStacks:
        // Switch to the system stack so we can call
        // stackfree.
        systemstack(markrootFreeGStacks)
    // 扫描各个span中特殊对象(析构器列表)
    case work.baseSpans <= i && i < work.baseStacks:
        // mark mspan.specials
        markrootSpans(gcw, int(i-work.baseSpans))
    // 扫描各个G的栈
    default:
        // the rest is scanning goroutine stacks
        var gp *g // 获取需要扫描的G
        if work.baseStacks <= i && i < work.baseEnd {
            // N.B. Atomic read of allglen in gcMarkRootPrepare
            // acts as a barrier to ensure that allgs must be large
            // enough to contain all relevant Gs.
            gp = allgs[i-work.baseStacks]
        } else {
            throw("markroot: bad index")
        }
```

```
        // remember when we've first observed the G blocked
        // needed only to output in traceback
        status := readgstatus(gp) // We are not in a scan state // 记录等待开始的时
间
        if (status == _Gwaiting || status == _Gsyscall) && gp.waitsince == 0 {
            gp.waitsince = work.tstart
        }
        // 切换到g0运行(有可能会扫到自己的栈)
        // scanstack must be done on the system stack in case
        // we're trying to scan our own stack.
        systemstack(func() {
            // If this is a self-scan, put the user G in
            // _Gwaiting to prevent self-deadlock. It may
            // already be in _Gwaiting if this is a mark
            // worker or we're in mark termination.
            userG := getg().m.curg // 判断扫描的栈是否自己的
            selfScan := gp == userG && readgstatus(userG) == _Grunning
            if selfScan { // 如果正在扫描自己的栈则切换状态到等待中防止死锁
                casgstatus(userG, _Grunning, _Gwaiting)
                userG.waitreason = waitReasonGarbageCollectionScan
            }

            // TODO: suspendG blocks (and spins) until gp
            // stops, which may take a while for
            // running goroutines. Consider doing this in
            // two phases where the first is non-blocking:
            // we scan the stacks we can and ask running
            // goroutines to scan themselves; and the
            // second blocks.
            stopped := suspendG(gp) // 阻塞goroutine
            if stopped.dead {
                gp.gcscandone = true
                return
            }
            if gp.gcscandone {
                throw("g already scanned")
            }
            scanstack(gp, gcw) // 扫描G的栈
            gp.gcscandone = true
            resumeG(stopped) // 恢复goroutine

            if selfScan {
                casgstatus(userG, _Gwaiting, _Grunning)
            }
        })
    }
}
```

markroot函数中会对几类根对象进行标记。

我们最主要关心的就是扫描goroutine的栈。

请注意一个关键点在扫描栈时，会阻塞对应的goroutine，停止运行，等扫描完这个goroutine的栈，再继续开始执行，具体原因将三色标记时再分析。

scanstack

扫描栈用的函数是scanstack

```
func scanstack(gp *g, gcw *gcWork) {
    if readgstatus(gp)&_Gscan == 0 {
        print("runtime:scanstack: gp=", gp, ", goid=", gp.goid, ", gp-
>atomicstatus=", hex(readgstatus(gp)), "\n")
        throw("scanstack - bad status")
    }

    switch readgstatus(gp) &^ _Gscan {
    default:
        print("runtime: gp=", gp, ", goid=", gp.goid, ", gp->atomicstatus=",
readgstatus(gp), "\n")
        throw("mark - bad status")
    case _Gdead:
        return
    case _Grunning:
        print("runtime: gp=", gp, ", goid=", gp.goid, ", gp->atomicstatus=",
readgstatus(gp), "\n")
        throw("scanstack: goroutine not stopped")
    case _Grunnable, _Gsyscall, _Gwaiting:
        // ok
    }

    if gp == getg() {
        throw("can't scan our own stack")
    }

    if isShrinkStackSafe(gp) {
        // Shrink the stack if not much of it is being used.
        shrinkstack(gp)
    } else {
        // Otherwise, shrink the stack at the next sync safe point.
        gp.preemptShrink = true
    }

    var state stackScanState
    state.stack = gp.stack

    if stackTraceDebug {
        println("stack trace goroutine", gp.goid)
    }

    if debugScanConservative && gp.asyncSafePoint {
        print("scanning async preempted goroutine ", gp.goid, " stack [",
hex(gp.stack.lo), ",", hex(gp.stack.hi), ")\n")
    }

    // Scan the saved context register. This is effectively a live
    // register that gets moved back and forth between the
    // register and sched.ctxt without a write barrier.
    if gp.sched.ctxt != nil {
        scanblock(uintptr(unsafe.Pointer(&gp.sched.ctxt)), sys.PtrSize,
&oneptrmask[0], gcw, &state)
    }
```

```
    // Scan the stack. Accumulate a list of stack objects.
    scanframe := func(frame *stkframe, unused unsafe.Pointer) bool {
        // scanframeworker会根据代码地址(pc)获取函数信息然后找到函数信息中的
stackmap.bytedata, 它保存了函数的栈上哪些地方有指针。
        scanframeworker(frame, &state, gcw)
        return true
    }


    ......

}
```

scanframeworker

```
func scanframeworker(frame *stkframe, state *stackScanState, gcw *gcWork) {
    if _DebugGC > 1 && frame.continpc != 0 {
        print("scanframe ", funcname(frame.fn), "\n")
    }

    isAsyncPreempt := frame.fn.valid() && frame.fn.funcID == funcID_asyncPreempt
    isDebugCall := frame.fn.valid() && frame.fn.funcID == funcID_debugCallv2
    if state.conservative || isAsyncPreempt || isDebugCall {
        if debugScanConservative {
            println("conservatively scanning function", funcname(frame.fn), "at
PC", hex(frame.continpc))
        }

        // Conservatively scan the frame. Unlike the precise
        // case, this includes the outgoing argument space
        // since we may have stopped while this function was
        // setting up a call.
        //
        // TODO: We could narrow this down if the compiler
        // produced a single map per function of stack slots
        // and registers that ever contain a pointer.
        if frame.varp != 0 {
            size := frame.varp - frame.sp
            if size > 0 {
                scanConservative(frame.sp, size, nil, gcw, state)
            }
        }

        // Scan arguments to this frame.
        if frame.arglen != 0 {
            // TODO: We could pass the entry argument map
            // to narrow this down further.
            scanConservative(frame.argp, frame.arglen, nil, gcw, state)
        }

        if isAsyncPreempt || isDebugCall {
            // This function's frame contained the
            // registers for the asynchronously stopped
            // parent frame. Scan the parent
            // conservatively.
            state.conservative = true
```

```
        } else {
            // We only wanted to scan those two frames
            // conservatively. Clear the flag for future
            // frames.
            state.conservative = false
        }
        return
    }

    locals, args, objs := getStackMap(frame, &state.cache, false)

    // Scan local variables if stack frame has been allocated.
    if locals.n > 0 {
        size := uintptr(locals.n) * sys.PtrSize
        scanblock(frame.varp-size, size, locals.bytedata, gcw, state)
    }

    // Scan arguments.
    if args.n > 0 {
        // 再调用scanblock来扫描函数的栈空间，同时函数的参数也会这样扫描
        scanblock(frame.argp, uintptr(args.n)*sys.PtrSize, args.bytedata, gcw,
state)
    }
```

scanblock

scanblock函数是一个通用的扫描函数，扫描全局变量和栈空间都会用它，和scanobject不同的是bitmap需要手动传入。

```
func scanblock(b0, n0 uintptr, ptrmask *uint8, gcw *gcWork, stk *stackScanState)
{
    // Use local copies of original parameters, so that a stack trace
    // due to one of the throws below shows the original block
    // base and extent.
    b := b0
    n := n0
    // 枚举扫描的地址
    for i := uintptr(0); i < n; {
        // Find bits for the next word.
        // 找到bitmap中对应的byte（内存管理那里讲过就是一个字节的低四位代表分别代表一块8B的内存存储对象是否包含指针)高四位表示是否还要继续扫描
        bits := uint32(*addb(ptrmask, i/(sys.PtrSize*8)))
        if bits == 0 { // 这里也说明下面是每 8 个指针处理一批

            i += sys.PtrSize * 8
            continue
        }
        for j := 0; j < 8 && i < n; j++ { // 枚举byte(8 bit)
            if bits&1 != 0 { // 如果该地址包含指针
                // Same work as in scanobject; see comments there.
                p := *(*uintptr)(unsafe.Pointer(b + i)) // 把对应内存地址里面存储的值取出来
                if p != 0 {
                    if obj, span, objIndex := findObject(p, b, i); obj != 0 { // 找到该对象对应的span和bitmap
                        greyobject(obj, b, i, span, gcw, objIndex) // 标记一个对象存活，并把它加到标记队列(该对象变为灰色)
```

```
                    } else if stk != nil && p >= stk.stack.lo && p <
stk.stack.hi {
                        stk.putPtr(p, false)
                    }
                }
            }
            bits >>= 1 // 处理下一个指针下一个bit
            i += sys.PtrSize
        }
    }
}
```

greyobject

greyobject用于标记一个对象存活,，并把它加到标记队列(该对象变为灰色)

```
func greyobject(obj, base, off uintptr, span *mspan, gcw *gcWork, objIndex
uintptr) {
    // obj should be start of allocation, and so must be at least pointer-
aligned.
    if obj&(sys.PtrSize-1) != 0 {
        throw("greyobject: obj not pointer-aligned")
    }
    mbits := span.markBitsForIndex(objIndex)

    if useCheckmark {
        if setCheckmark(obj, base, off, mbits) {
            // Already marked.
            return
        }
    } else {
        if debug.gccheckmark > 0 && span.isFree(objIndex) {
            print("runtime: marking free object ", hex(obj), " found at *(",
hex(base), "+", hex(off), ")\n")
            gcDumpObject("base", base, off)
            gcDumpObject("obj", obj, ^uintptr(0))
            getg().m.traceback = 2
            throw("marking free object")
        }
        // 如果对象所在的span中的gcmarkBits对应的bit已经设置为1则可以跳过处理
        // If marked we have nothing to do.
        if mbits.isMarked() {
            return
        }
        mbits.setMarked() // 设置对象所在的span中的gcmarkBits对应的bit为1

        // Mark span.
        arena, pageIdx, pageMask := pageIndexOf(span.base())
        if arena.pageMarks[pageIdx]&pageMask == 0 {
            atomic.Or8(&arena.pageMarks[pageIdx], pageMask)
        }
        // 如果确定对象不包含指针(所在span的类型是noscan)，则不需要把对象放入标记队列
        // If this is a noscan object, fast-track it to black
        // instead of greying it.
        if span.spanclass.noscan() {
            gcw.bytesMarked += uint64(span.elemsize)
            return
```

```
        }
    }
    // 把对象放入标记队列，先放入本地标记队列，失败时把本地标记队列中的部分工作转移到全局标记队列，再放入本地标记队列。
    // Queue the obj for scanning. The PREFETCH(obj) logic has been removed but
    // seems like a nice optimization that can be added back in.
    // There needs to be time between the PREFETCH and the use.
    // Previously we put the obj in an 8 element buffer that is drained at a rate
    // to give the PREFETCH time to do its work.
    // Use of PREFETCHNTA might be more appropriate than PREFETCH
    if !gcw.putFast(obj) {
        gcw.put(obj)
    }
}
```

让我们把视线关注回gcDrain函数中，当调用markroot将根对象标记完成之后，继续往下执行，此时工作队列中就有灰色对象了，从标记工作队列中取出对象调用

scanobject函数。

gcDrain函数后半部分逻辑

```
    // 根对象已经在标记队列中，消费标记队列  如果标记了preemptible，循环直到被抢占
    // Drain heap marking jobs.
    // Stop if we're preemptible or if someone wants to STW.
    for !(gp.preempt && (preemptible || atomic.Load(&sched.gcwaiting) != 0)) {
        // Try to keep work available on the global queue. We used to
        // check if there were waiting workers, but it's better to
        // just keep work available than to make workers wait. In the
        // worst case, we'll do O(log(_WorkbufSize)) unnecessary
        // balances.
        if work.full == 0 {
            // 如果全局标记队列为空，把本地标记队列的一部分工作分过去 (如果wbuf2不为空则移动wbuf2过去,否则看wbuf1中是否有4个任务，有就移动一半过去)
            gcw.balance()
        }

        b := gcw.tryGetFast()
        if b == 0 {
            b = gcw.tryGet()
            if b == 0 {
                // Flush the write barrier
                // buffer; this may create
                // more work.
                wbBufFlush(nil, 0) // 这个函数会将写屏障缓冲中的对象标记，并加入到标记工作队列中去。
                b = gcw.tryGet()
            }
        }
        if b == 0 { // 获取不到对象，标记队列已为空，跳出循环
            // Unable to get work.
            break
        }
        // gcDrain函数扫描完根对象，就会开始消费标记队列，对从标记队列中取出的对象调用scanobject函数。
        scanobject(b, gcw)
```

```
        // 如果已经扫描了一定数量的对象(gcCreditSlack的值是2000)
        // Flush background scan work credit to the global
        // account if we've accumulated enough locally so
        // mutator assists can draw on it.
        if gcw.scanWork >= gcCreditSlack {
            atomic.Xaddint64(&gcController.scanWork, gcw.scanWork) // 把扫描的对象数量添加到全局
            if flushBgCredit { // 减少辅助GC的工作量和唤醒等待中的G
                gcFlushBgCredit(gcw.scanWork - initScanWork)
                initScanWork = 0
            }
            checkWork -= gcw.scanWork
            gcw.scanWork = 0

            if checkWork <= 0 {
                checkWork += drainCheckThreshold
                if check != nil && check() {
                    break
                }
            }
        }
    }

done:
    // Flush remaining scan work credit.
    if gcw.scanWork > 0 { // 把扫描的对象数量添加到全局
        atomic.Xaddint64(&gcController.scanWork, gcw.scanWork)
        if flushBgCredit { // 减少辅助GC的工作量和唤醒等待中的G
            gcFlushBgCredit(gcw.scanWork - initScanWork)
        }
        gcw.scanWork = 0
    }
}
```

2、scanobject

```
func scanobject(b uintptr, gcw *gcWork) {
    // Find the bits for b and the size of the object at b.
    //
    // b is either the beginning of an object, in which case this
    // is the size of the object to scan, or it points to an
    // oblet, in which case we compute the size to scan below.
    hbits := heapBitsForAddr(b) // 获取对象对应的bitmap
    s := spanOfUnchecked(b) // 获取对象所在的span
    n := s.elemsize // 获取对象的大小
    if n == 0 {
        throw("scanobject n == 0")
    }
    // 对象大小过大时(maxObletBytes是128KB)需要分割扫描 每次最多只扫描128KB
    if n > maxObletBytes {
        // Large object. Break into oblets for better
        // parallelism and lower latency.
        if b == s.base() {
            // It's possible this is a noscan object (not
            // from greyobject, but from other code
            // paths), in which case we must *not* enqueue
            // oblets since their bitmaps will be
```

```
            // uninitialized.
            if s.spanclass.noscan() {
                // Bypass the whole scan.
                gcw.bytesMarked += uint64(n)
                return
            }

            // Enqueue the other oblets to scan later.
            // Some oblets may be in b's scalar tail, but
            // these will be marked as "no more pointers",
            // so we'll drop out immediately when we go to
            // scan those.
            for oblet := b + maxObletBytes; oblet < s.base()+s.elemsize; oblet
+= maxObletBytes {
                if !gcw.putFast(oblet) {
                    gcw.put(oblet)
                }
            }
        }

        // Compute the size of the oblet. Since this object
        // must be a large object, s.base() is the beginning
        // of the object.
        n = s.base() + s.elemsize - b
        if n > maxObletBytes {
            n = maxObletBytes
        }
    }
    // 扫描对象中的指针
    var i uintptr
    for i = 0; i < n; i, hbits = i+sys.PtrSize, hbits.next() {
        // Load bits once. See CL 22712 and issue 16973 for discussion.
        bits := hbits.bits()
        if bits&bitScan == 0 { // 是否继续扫描
            break // no more pointers in this object
        }
        if bits&bitPointer == 0 { // 判断这个对象中是否包含指针
            continue // not a pointer
        }
        // 取出指针的值
        // Work here is duplicated in scanblock and above.
        // If you make changes here, make changes there too.
        obj := *(*uintptr)(unsafe.Pointer(b + i))

        // At this point we have extracted the next potential pointer.
        // Quickly filter out nil and pointers back to the current object.
        if obj != 0 && obj-b >= n {
            // Test if obj points into the Go heap and, if so,
            // mark the object.
            //
            // Note that it's possible for findObject to
            // fail if obj points to a just-allocated heap
            // object because of a race with growing the
            // heap. In this case, we know the object was
            // just allocated and hence will be marked by
            // allocation itself.
            // 如果指针在arena区域中，则调用greyobject标记对象并把对象放到标记队列中。
            if obj, span, objIndex := findObject(obj, b, i); obj != 0 {
```

```
                greyobject(obj, b, i, span, gcw, objIndex) // 为找到的活跃对象上色，
并放入工作队列。
            }
        }
    }
    gcw.bytesMarked += uint64(n)
    gcw.scanWork += int64(i)
}
```

在所有后台标记任务都把标记队列消费完毕时, 会执行 gcMarkDone 函数准备进入完成标记阶段(mark termination)

```
func gcMarkDone() {
    // Ensure only one thread is running the ragged barrier at a
    // time.
    semacquire(&work.markDoneSema)

top:
    // Re-check transition condition under transition lock.
    //
    // It's critical that this checks the global work queues are
    // empty before performing the ragged barrier. Otherwise,
    // there could be global work that a P could take after the P
    // has passed the ragged barrier.
    if !(gcphase == _GCmark && work.nwait == work.nproc &&
!gcMarkWorkAvailable(nil)) {
        semrelease(&work.markDoneSema)
        return
    }

    // forEachP needs worldsema to execute, and we'll need it to
    // stop the world later, so acquire worldsema now.
    semacquire(&worldsema)

    // Flush all local buffers and collect flushedWork flags.
    gcMarkDoneFlushed = 0
    systemstack(func() { // 把所有本地标记队列中的对象都推到全局标记队列
        gp := getg().m.curg
        // Mark the user stack as preemptible so that it may be scanned.
        // Otherwise, our attempt to force all P's to a safepoint could
        // result in a deadlock as we attempt to preempt a worker that's
        // trying to preempt us (e.g. for a stack scan).
        casgstatus(gp, _Grunning, _Gwaiting)
        forEachP(func(_p_ *p) {
            // Flush the write barrier buffer, since this may add
            // work to the gcWork.
            wbBufFlush1(_p_)

            // Flush the gcWork, since this may create global work
            // and set the flushedWork flag.
            //
            // TODO(austin): Break up these workbufs to
            // better distribute work.
            _p_.gcw.dispose()
            // Collect the flushedWork flag.
            if _p_.gcw.flushedWork {
                atomic.Xadd(&gcMarkDoneFlushed, 1)
```

```
                _p_.gcw.flushedWork = false
            }
        })
        casgstatus(gp, _Gwaiting, _Grunning)
    })

    if gcMarkDoneFlushed != 0 {
        // More grey objects were discovered since the
        // previous termination check, so there may be more
        // work to do. Keep going. It's possible the
        // transition condition became true again during the
        // ragged barrier, so re-check it.
        semrelease(&worldsema)
        goto top
    }

    // There was no global work, no local work, and no Ps
    // communicated work since we took markDoneSema. Therefore
    // there are no grey objects and no more objects can be
    // shaded. Transition to mark termination.
    now := nanotime()
    work.tMarkTerm = now
    work.pauseStart = now
    getg().m.preemptoff = "gcing"
    if trace.enabled {
        traceGCSTWStart(0)
    }
    // 第二次 stop the world
    // 停止所有运行中的G, 并禁止它们运行
    systemstack(stopTheWorldWithSema)
    // The gcphase is _GCmark, it will transition to _GCmarktermination
    // below. The important thing is that the wb remains active until
    // all marking is complete. This includes writes made by the GC.

    // There is sometimes work left over when we enter mark termination due
    // to write barriers performed after the completion barrier above.
    // Detect this and resume concurrent mark. This is obviously
    // unfortunate.
    //
    // See issue #27993 for details.
    //
    // Switch to the system stack to call wbBufFlush1, though in this case
    // it doesn't matter because we're non-preemptible anyway.
    restart := false
    systemstack(func() {
        for _, p := range allp {
            wbBufFlush1(p)
            if !p.gcw.empty() {
                restart = true
                break
            }
        }
    })
    if restart {
        getg().m.preemptoff = ""
        systemstack(func() {
            now := startTheWorldWithSema(true)
            work.pauseNS += now - work.pauseStart
```

```
            memstats.gcPauseDist.record(now - work.pauseStart)
        })
        semrelease(&worldsema)
        goto top
    }

    // Disable assists and background workers. We must do
    // this before waking blocked assists.
    atomic.Store(&gcBlackenEnabled, 0) // 禁止辅助GC和后台标记任务的运行

    // Wake all blocked assists. These will run when we
    // start the world again.
    gcWakeAllAssists() // 唤醒所有因为辅助GC而休眠的G

    // Likewise, release the transition lock. Blocked
    // workers and assists will run when we start the
    // world again.
    semrelease(&work.markDoneSema)

    // In STW mode, re-enable user goroutines. These will be
    // queued to run after we start the world.
    schedEnableUser(true)

    // endCycle depends on all gcWork cache stats being flushed.
    // The termination algorithm above ensured that up to
    // allocations since the ragged barrier.
    nextTriggerRatio := gcController.endCycle(work.userForced) // 计算下一次触发gc
需要的heap大小

    // Perform mark termination. This will restart the world.
    gcMarkTermination(nextTriggerRatio) // 调用 runtime.gcMarkTermination 进入标记
终止阶段 // 进入完成标记阶段，会重新启动世界
}
```

标记终止阶段

gcMarkTermination

```
func gcMarkTermination(nextTriggerRatio float64) {
    // Start marktermination (write barrier remains enabled for now).
    setGCPhase(_GCmarktermination) // gcphase置为_GCMarkTermination

    work.heap1 = gcController.heapLive
    startTime := nanotime()

    mp := acquirem()
    mp.preemptoff = "gcing"
    _g_ := getg()
    _g_.m.traceback = 2
    gp := _g_.m.curg
    casgstatus(gp, _Grunning, _Gwaiting)
    gp.waitreason = waitReasonGarbageCollection

    // Run gc on the g0 stack. We do this so that the g stack
    // we're currently running on will no longer change. Cuts
    // the root set down a bit (g0 stacks are not scanned, and
    // we don't need to scan gc's internal state).  We also
```

```
    // need to switch to g0 so we can shrink the stack.
    systemstack(func() {
        gcMark(startTime)
        // Must return immediately.
        // The outer function's stack may have moved
        // during gcMark (it shrinks stacks, including the
        // outer function's stack), so we must not refer
        // to any of its variables. Return back to the
        // non-system stack to pick up the new addresses
        // before continuing.
    })
    // 重新切换到g0运行
    systemstack(func() {
        work.heap2 = work.bytesMarked
        if debug.gccheckmark > 0 { // 如果启用了checkmark则执行检查，检查是否所有可到达
的对象都有标记。
            // Run a full non-parallel, stop-the-world
            // mark using checkmark bits, to check that we
            // didn't forget to mark anything during the
            // concurrent mark process.
            startCheckmarks()
            gcResetMarkState()
            gcw := &getg().m.p.ptr().gcw
            gcDrain(gcw, 0)
            wbBufFlush1(getg().m.p.ptr())
            gcw.dispose()
            endCheckmarks()
        }

        // marking is complete so we can turn the write barrier off
        setGCPhase(_GCoff) // 设置当前GC阶段到_GCoff，并禁用写屏障。
        gcSweep(work.mode) // 唤醒后台清扫任务，将在STW结束后开始运行。
    })
    // 设置G的状态为运行中
    _g_.m.traceback = 0
    casgstatus(gp, _Gwaiting, _Grunning)

    if trace.enabled {
        traceGCDone()
    }

    // all done
    mp.preemptoff = ""

    if gcphase != _GCoff {
        throw("gc done but gcphase != _GCoff")
    }

    // Record heapGoal and heap_inuse for scavenger.
    gcController.lastHeapGoal = gcController.heapGoal
    memstats.last_heap_inuse = memstats.heap_inuse

    // Update GC trigger and pacing for the next cycle.
    gcController.commit(nextTriggerRatio) // 更新下一次触发gc需要的heap大小
(gc_trigger)
    // 更新用时记录
    // Update timing memstats
    now := nanotime()
```

```
    sec, nsec, _ := time_now()
    unixNow := sec*1e9 + int64(nsec)
    work.pauseNS += now - work.pauseStart
    work.tEnd = now
    memstats.gcPauseDist.record(now - work.pauseStart)
    atomic.Store64(&memstats.last_gc_unix, uint64(unixNow)) // must be Unix time
to make sense to user
    atomic.Store64(&memstats.last_gc_nanotime, uint64(now)) // monotonic time
for us
    memstats.pause_ns[memstats.numgc%uint32(len(memstats.pause_ns))] =
uint64(work.pauseNS)
    memstats.pause_end[memstats.numgc%uint32(len(memstats.pause_end))] =
uint64(unixNow)
    memstats.pause_total_ns += uint64(work.pauseNS)
    // 更新所用cpu记录
    // Update work.totaltime.
    sweepTermCpu := int64(work.stwprocs) * (work.tMark - work.tSweepTerm)
    // We report idle marking time below, but omit it from the
    // overall utilization here since it's "free".
    markCpu := gcController.assistTime + gcController.dedicatedMarkTime +
gcController.fractionalMarkTime
    markTermCpu := int64(work.stwprocs) * (work.tEnd - work.tMarkTerm)
    cycleCpu := sweepTermCpu + markCpu + markTermCpu
    work.totaltime += cycleCpu

    // Compute overall GC CPU utilization.
    totalCpu := sched.totaltime + (now-sched.procresizetime)*int64(gomaxprocs)
    memstats.gc_cpu_fraction = float64(work.totaltime) / float64(totalCpu)
    // 重置清扫状态
    // Reset sweep state.
    sweep.nbgsweep = 0
    sweep.npausesweep = 0
    // 统计强制开始GC的次数
    if work.userForced {
        memstats.numforcedgc++
    }
    // 统计执行GC的次数然后唤醒等待清扫的G
    // Bump GC cycle count and wake goroutines waiting on sweep.
    lock(&work.sweepWaiters.lock)
    memstats.numgc++
    injectglist(&work.sweepWaiters.list)
    unlock(&work.sweepWaiters.lock)

    // Finish the current heap profiling cycle and start a new
    // heap profiling cycle. We do this before starting the world
    // so events don't leak into the wrong cycle.
    mProf_NextCycle() // 性能统计用

    // There may be stale spans in mcaches that need to be swept.
    // Those aren't tracked in any sweep lists, so we need to
    // count them against sweep completion until we ensure all
    // those spans have been forced out.
    sl := newSweepLocker()
    sl.blockCompletion()
    // 重新启动世界
    systemstack(func() { startTheWorldWithSema(true) }) // Start The World，进入清
扫阶段
```

```
    // Flush the heap profile so we can start a new cycle next GC.
    // This is relatively expensive, so we don't do it with the
    // world stopped.
    mProf_Flush() // 性能统计用

    // Prepare workbufs for freeing by the sweeper. We do this
    // asynchronously because it can take non-trivial time.
    prepareFreeWorkbufs() // 移动标记队列使用的缓冲区到自由列表，使得它们可以被回收

    // Free stack spans. This must be done between GC cycles.
    systemstack(freeStackSpans) // 释放未使用的栈

    // Ensure all mcaches are flushed. Each P will flush its own
    // mcache before allocating, but idle Ps may not. Since this
    // is necessary to sweep all spans, we need to ensure all
    // mcaches are flushed before we start the next GC cycle.
    systemstack(func() {
        forEachP(func(_p_ *p) {
            _p_.mcache.prepareForSweep()
        })
    })
    // Now that we've swept stale spans in mcaches, they don't
    // count against unswept spans.
    sl.dispose()

    // Print gctrace before dropping worldsema. As soon as we drop
    // worldsema another cycle could start and smash the stats
    // we're trying to print.
    if debug.gctrace > 0 {
        util := int(memstats.gc_cpu_fraction * 100)

        var sbuf [24]byte
        printlock()
        print("gc ", memstats.numgc,
            " @", string(itoaDiv(sbuf[:], uint64(work.tSweepTerm-
runtimeInitTime)/1e6, 3)), "s ",
            util, "%: ")
        prev := work.tSweepTerm
        for i, ns := range []int64{work.tMark, work.tMarkTerm, work.tEnd} {
            if i != 0 {
                print("+")
            }
            print(string(fmtNSAsMS(sbuf[:], uint64(ns-prev))))
            prev = ns
        }
        print(" ms clock, ")
        for i, ns := range []int64{sweepTermCpu, gcController.assistTime,
gcController.dedicatedMarkTime + gcController.fractionalMarkTime,
gcController.idleMarkTime, markTermCpu} {
            if i == 2 || i == 3 {
                // Separate mark time components with /.
                print("/")
            } else if i != 0 {
                print("+")
            }
            print(string(fmtNSAsMS(sbuf[:], uint64(ns))))
        }
        print(" ms cpu, ",
```

```
            work.heap0>>20, "->", work.heap1>>20, "->", work.heap2>>20, " MB, ",
            work.heapGoal>>20, " MB goal, ",
            work.maxprocs, " P")
        if work.userForced {
            print(" (forced)")
        }
        print("\n")
        printunlock()
    }

    semrelease(&worldsema)
    semrelease(&gcsema)
    // Careful: another GC cycle may start now.

    releasem(mp) // 重新允许当前的G被抢占
    mp = nil
    // 如果是并行GC，让当前M继续运行(会回到gcBgMarkWorker然后休眠)，如果不是并行GC，则让当前M开始调度
    // now that gc is done, kick off finalizer thread if needed
    if !concurrentSweep {
        // give the queued finalizers, if any, a chance to run
        Gosched()
    }
}
```

setGCPhase(_GCoff)

```
func setGCPhase(x uint32) { // setGCPhase函数会修改表示当前GC阶段的全局变量和是否开启写屏障的全局变量
    atomic.Store(&gcphase, x)
    writeBarrier.needed = gcphase == _GCmark || gcphase == _GCmarktermination
    writeBarrier.enabled = writeBarrier.needed || writeBarrier.cgo
}
```

清除阶段

gcSweep

gcSweep 函数会唤醒后台清扫任务。

```
func gcSweep(mode gcMode) {
    assertWorldStopped()

    if gcphase != _GCoff {
        throw("gcSweep being done but phase is not GCoff")
    }

    lock(&mheap_.lock)
    mheap_.sweepgen += 2
    mheap_.sweepDrained = 0
    mheap_.pagesSwept = 0
    mheap_.sweepArenas = mheap_.allArenas
    mheap_.reclaimIndex = 0
    mheap_.reclaimCredit = 0
    unlock(&mheap_.lock)

    sweep.centralIndex.clear()
```

```
    if !_ConcurrentSweep || mode == gcForceBlockMode {
        // Special case synchronous sweep.
        // Record that no proportional sweeping has to happen.
        lock(&mheap_.lock)
        mheap_.sweepPagesPerByte = 0
        unlock(&mheap_.lock)
        // Sweep all spans eagerly.
        for sweepone() != ^uintptr(0) {
            sweep.npausesweep++
        }
        // Free workbufs eagerly.
        prepareFreeWorkbufs()
        for freeSomeWbufs(false) {
        }
        // All "free" events for this mark/sweep cycle have
        // now happened, so we can make this profile cycle
        // available immediately.
        mProf_NextCycle()
        mProf_Flush()
        return
    }
    // 唤醒后台清扫任务
    // Background sweep.
    lock(&sweep.lock)
    if sweep.parked {
        sweep.parked = false
        ready(sweep.g, 0, true)
    }
    unlock(&sweep.lock)
}
```

后台清扫任务会在程序启动时 runtime.main 函数中调用的 gcenable 函数启动。

```
func gcenable() {
    // Kick off sweeping and scavenging.
    gcenable_setup = make(chan int, 2)
    go bgsweep()
    go bgscavenge()
    <-gcenable_setup
    <-gcenable_setup
    gcenable_setup = nil
    memstats.enablegc = true // now that runtime is initialized, GC is okay
}
```

后台清扫任务执行的函数是bgsweep

```
func bgsweep() {
    sweep.g = getg()

    lockInit(&sweep.lock, lockRankSweep)
    lock(&sweep.lock) // 等待唤醒
    sweep.parked = true
    gcenable_setup <- 1
    goparkunlock(&sweep.lock, waitReasonGCSweepWait, traceEvGoBlock, 1)
    // 循环清扫
```

```
    for {
        for sweepone() != ^uintptr(0) { // 清扫一个span, 然后进入调度(一次只做少量工作)
            sweep.nbgsweep++
            Gosched()
        }
        for freeSomewbufs(true) { // 释放一些未使用的标记队列缓冲区到heap
            Gosched()
        }
        lock(&sweep.lock) // 如果清扫未完成则继续循环
        if !isSweepDone() {
            // This can happen if a GC runs between
            // gosweepone returning ^0 above
            // and the lock being acquired.
            unlock(&sweep.lock)
            continue
        }
        sweep.parked = true // 否则让后台清扫任务进入休眠, 当前M继续调度
        goparkunlock(&sweep.lock, waitReasonGCSweepWait, traceEvGoBlock, 1)
    }
}
```

至此整个GC流程大概分析完了。

关键技术

让我们来看看GO语言GC过程中用到的三色标记以及屏障技术

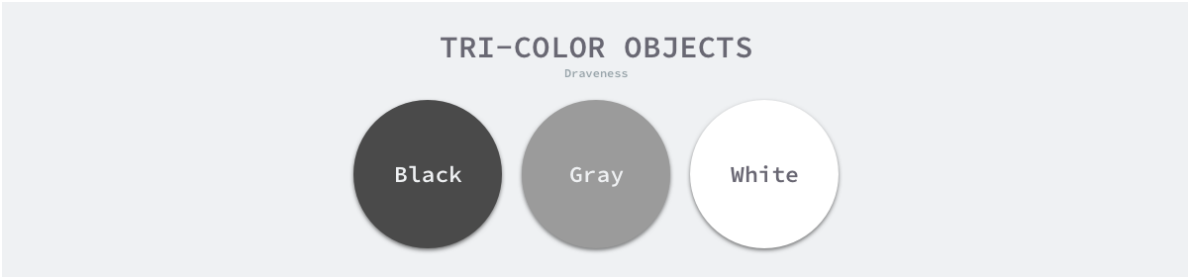
直接搬运德莱文大神的吧

三色抽象

为了解决原始标记清除算法带来的长时间 STW，多数现代的追踪式垃圾收集器都会实现三色标记算法的变种以缩短 STW 的时间。三色标记算法将程序中的对象分

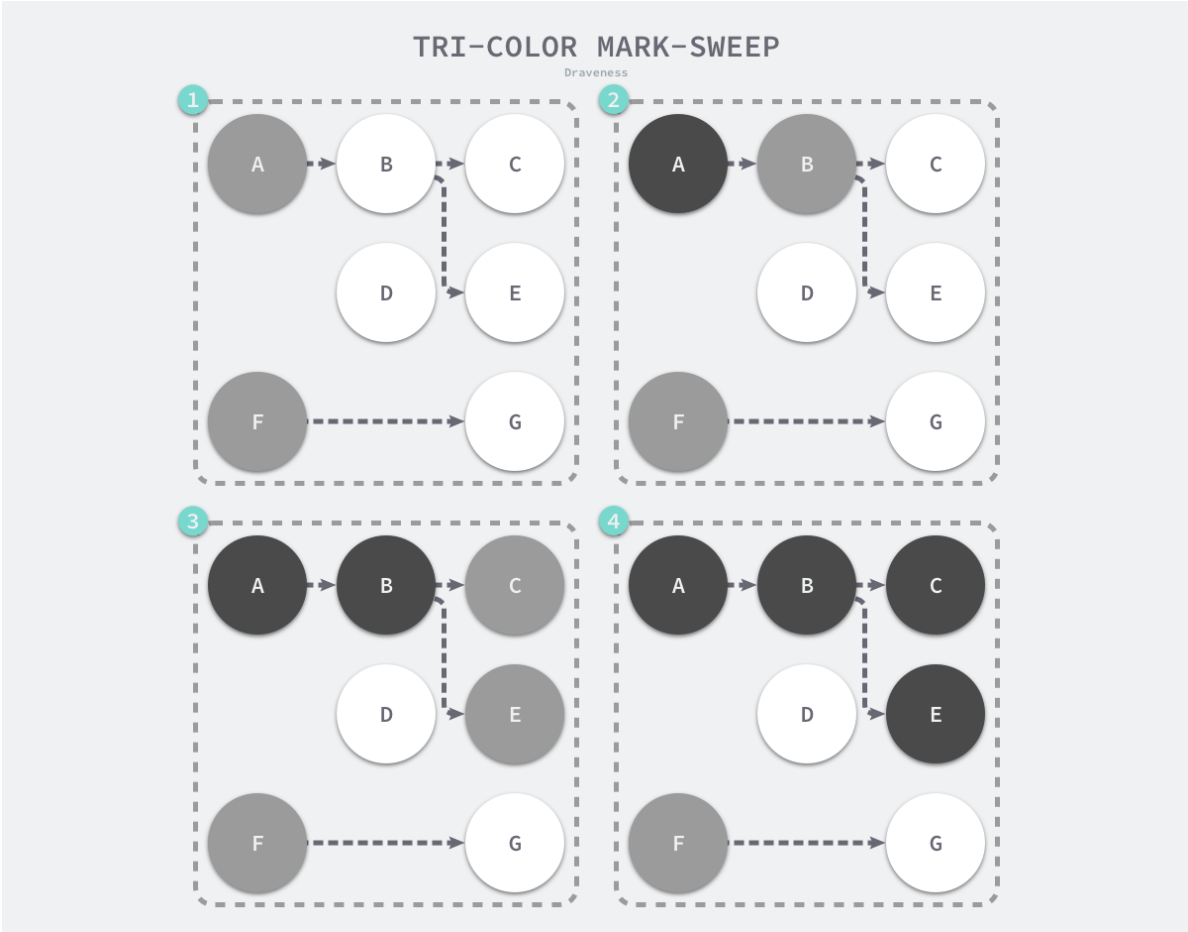
成白色、黑色和灰色三类。

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



在垃圾收集器开始工作时，程序中不存在任何的黑色对象，垃圾收集的根对象会被标记成灰色，垃圾收集器只会从灰色对象集合中取出对象开始扫描，当灰色集合

中不存在任何对象时，标记阶段就会结束。

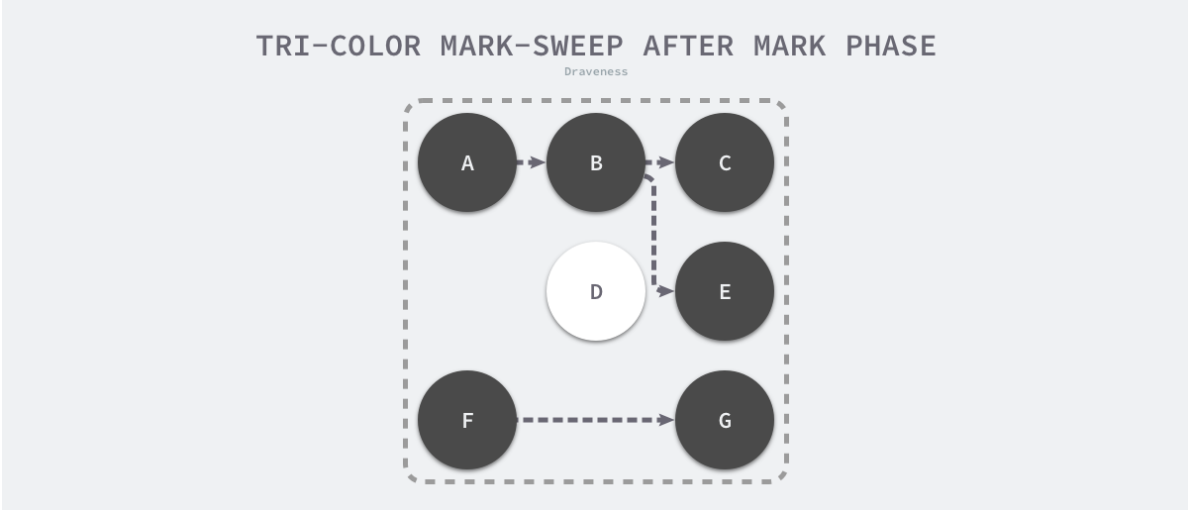


三色标记垃圾收集器的工作原理很简单，我们可以将其归纳成以下几个步骤：

1. 从灰色对象的集合中选择一个灰色对象并将其标记成黑色；
2. 将黑色对象指向的所有对象都标记成灰色，保证该对象和被该对象引用的对象都不会被回收；
3. 重复上述两个步骤直到对象图中不存在灰色对象；

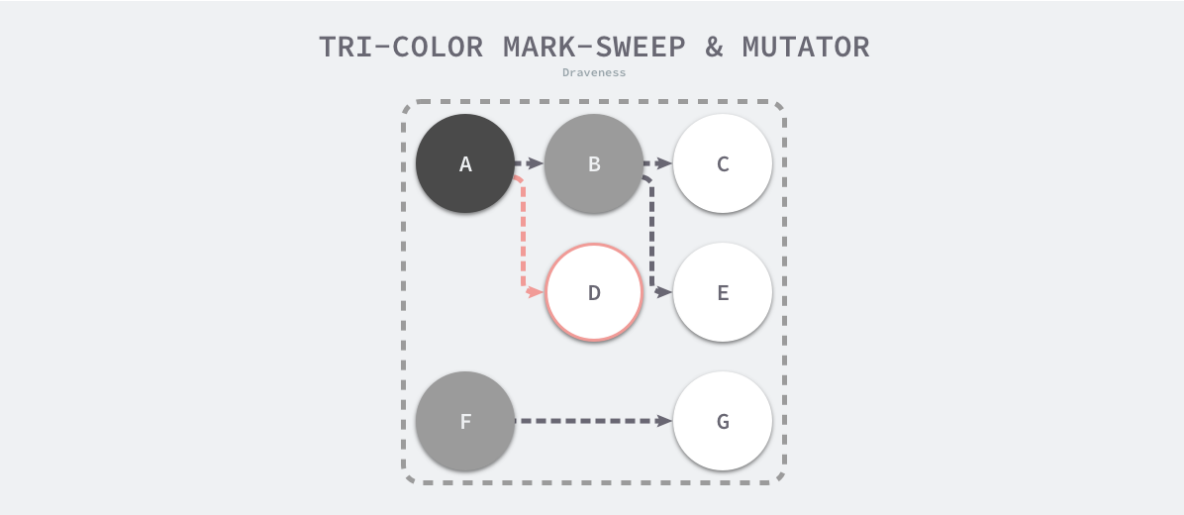
当三色的标记清除的标记阶段结束之后，应用程序的堆中就不存在任何的灰色对象，我们只能看到黑色的存活对象以及白色的垃圾对象，垃圾收集器可以回收这些

白色的垃圾，下面是使用三色标记垃圾收集器执行标记后的堆内存，堆中只有对象 D 为待回收的垃圾：



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，在如下所示的三色标记过

程中，用户程序建立了从 A 对象到 D 对象的引用，但是因为程序中已经不存在灰色对象了，所以 D 对象会被垃圾收集器错误地回收。



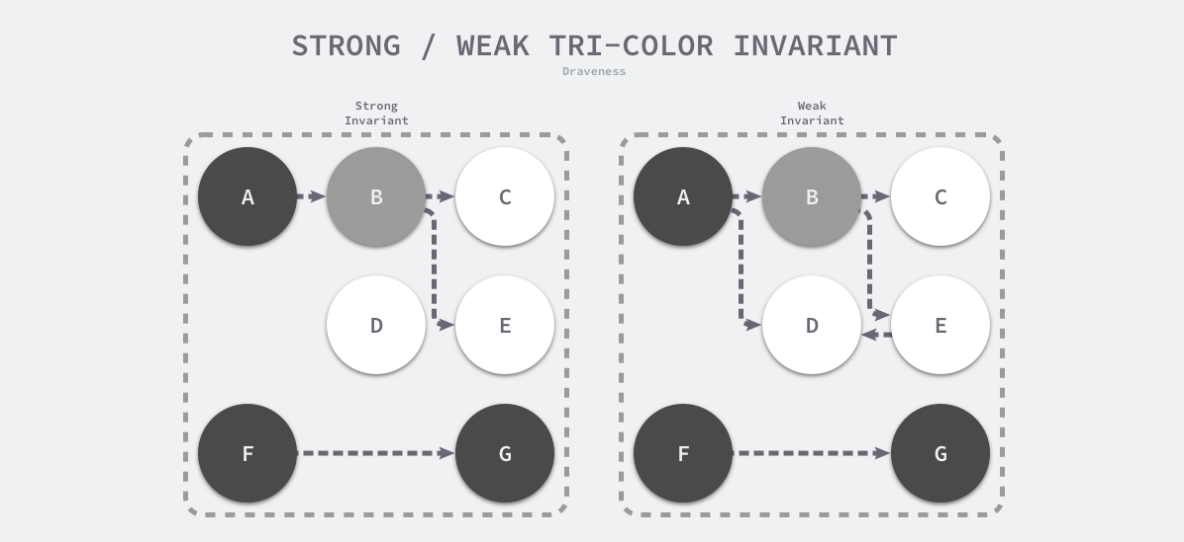
屏障技术

内存屏障技术是一种屏障指令，它可以让 CPU 或者编译器在执行内存相关操作时遵循特定的约束，目前多数的现代处理器都会乱序执行指令以最大化性能，但是

该技术能够保证内存操作的顺序性，在内存屏障前执行的操作一定会先于内存屏障后执行的操作。

想要在并发或者增量的标记算法中保证正确性，我们需要达成以下两种三色不变性（Tri-color invariant）中的一种：

- 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。



上图分别展示了遵循强三色不变性和弱三色不变性的堆内存，遵循上述两个不变性中的任意一个，我们都能保证垃圾收集算法的正确性，而屏障技术就是在并发或

者增量标记过程中保证三色不变性的重要技术。

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。

我们在这里想要介绍的是 Go 语言中使用的两种写屏障技术，分别是 Dijkstra 提出的插入写屏障和 Yuasa 提出的删除写屏障，这里会分析它们如何保证三色不

变性和垃圾收集器的正确性。

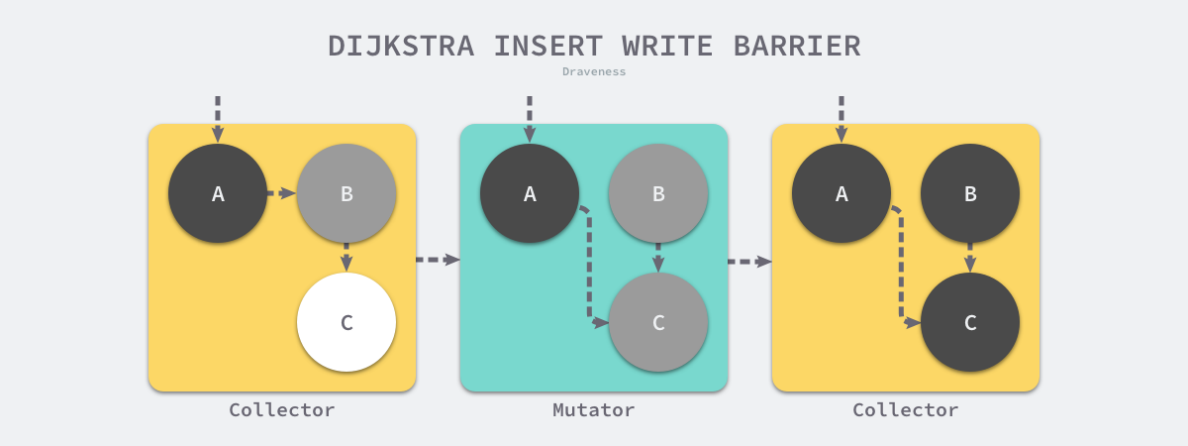
插入写屏障

Dijkstra 在 1978 年提出了插入写屏障，通过如下所示的写屏障，用户程序和垃圾收集器可以在交替工作的情况下保证程序执行的正确性：

```
writePointer(slot, ptr):
    shade(ptr)
    *slot = ptr
```

上述插入写屏障的伪代码非常好理解，每当执行类似 *slot = ptr 的表达式时，我们会执行上述写屏障通过 shade 函数尝试改变指针的颜色。如果 ptr 指针是

白色的，那么该函数会将该对象设置成灰色，其他情况则保持不变。



假设我们在应用程序中使用 Dijkstra 提出的插入写屏障，在一个垃圾收集器和用户程序交替运行的场景中会出现如上图所示的标记过程：

1. 垃圾收集器将根对象指向 A 对象标记成黑色并将 A 对象指向的对象 B 标记成灰色；
2. 用户程序修改 A 对象的指针，将原本指向 B 对象的指针指向 C 对象，这时触发写屏障将 C 对象标记成灰色；
3. 垃圾收集器依次遍历程序中的其他灰色对象，将它们分别标记成黑色；

Dijkstra 的插入写屏障是一种相对保守的屏障技术，它会将**有存活可能的对象都标记成灰色**以满足强三色不变性。在如上所示的垃圾收集过程中，实际上不再存活

的 B 对象最后没有被回收；而如果我们在第二和第三步之间将指向 C 对象的指针改回指向 B，垃圾收集器仍然认为 C 对象是存活的，这些被错误标记的垃圾对象

只有在下一个循环才会被回收。

插入式的 Dijkstra 写屏障虽然实现非常简单并且也能保证强三色不变性，但是它也有明显的缺点。因为栈上的对象在垃圾收集中也会被认为是根对象，所以为了保

证内存的安全，Dijkstra 必须为栈上的对象增加写屏障或者在标记阶段完成重新对栈上的对象进行扫描，这两种方法各有各的缺点，前者会大幅度增加写入指针的

额外开销，后者重新扫描栈对象时需要暂停程序，垃圾收集算法的设计者需要在这两者之间做出权衡。

删除写屏障

Yuasa 在 1990 年的论文 Real-time garbage collection on general-purpose machines 中提出了删除写屏障，因为一旦该写屏障开始工作，它会保证开启写屏障

时堆上所有对象的可达，所以也被称作快照垃圾收集（Snapshot GC）

该算法会使用如下所示的写屏障保证增量或者并发执行垃圾收集时程序的正确性：

```
writePointer(slot, ptr)
    shade(*slot)
    *slot = ptr
```

上述代码会在老对象的引用被删除时，将白色的老对象涂成灰色，这样删除写屏障就可以保证弱三色不变性，老对象引用的下游对象一定可以被灰色对象引用。

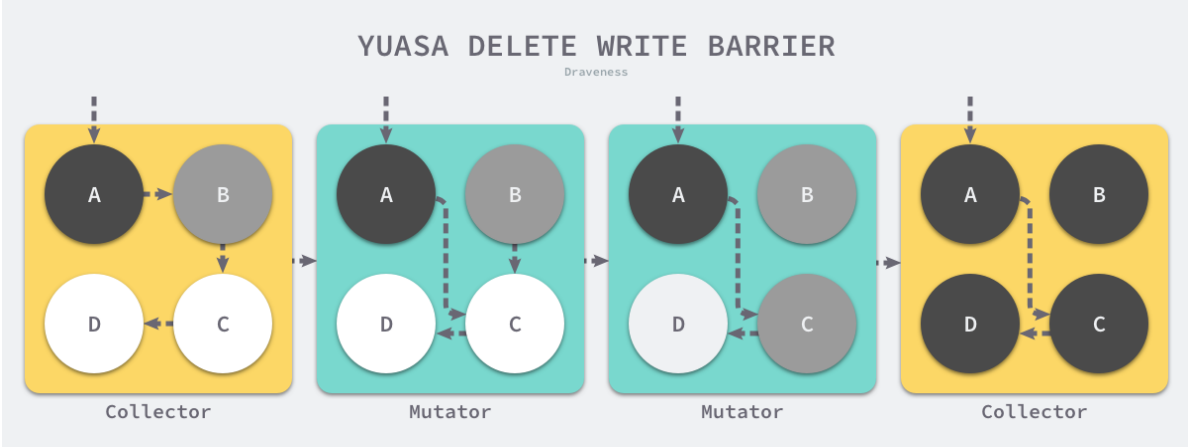


图 7-29 Yuasa 删除写屏障

假设我们在应用程序中使用 Yuasa 提出的删除写屏障，在一个垃圾收集器和用户程序交替运行的场景中会出现如上图所示的标记过程：

1. 垃圾收集器将根对象指向 A 对象标记成黑色并将 A 对象指向的对象 B 标记成灰色；
2. 用户程序将 A 对象原本指向 B 的指针指向 C，触发删除写屏障，但是因为 B 对象已经是灰色的，所以不做改变；
3. **用户程序将 B 对象原本指向 C 的指针删除，触发删除写屏障，白色的 C 对象被涂成灰色**；
4. 垃圾收集器依次遍历程序中的其他灰色对象，将它们分别标记成黑色；

上述过程中的第三步触发了 Yuasa 删除写屏障的着色，因为用户程序删除了 B 指向 C 对象的指针，所以 C 和 D 两个对象会分别违反强三色不变性和弱三色不变性：

- 强三色不变性 — 黑色的 A 对象直接指向白色的 C 对象；
- 弱三色不变性 — 垃圾收集器无法从某个灰色对象出发，经过几个连续的白色对象访问白色的 C 和 D 两个对象；

Yuasa 删除写屏障通过对 C 对象的着色，保证了 C 对象和下游的 D 对象能够在这一次垃圾收集的循环中存活，避免发生悬挂指针以保证用户程序的正确性。

终于搬运完了，舒坦！！！

混合写屏障

GO语言再GO 1.5中实现了基于三色标记清除算法，是使用迪杰斯特拉提出的插入写屏障来解决漏标的问题。

运行时并没有在所有的垃圾收集根对象上开启插入写屏障。因为应用程序可能包含成百上千的 Goroutine，而垃圾收集的根对象一般包括全局变量和栈对象，如果

运行时需要在几百个 Goroutine 的栈上都开启写屏障，会带来巨大的额外开销，所以 Go 团队在实现上选择了在标记阶段完成时**暂停程序、将所有栈对象标记为灰**

色并重新扫描，在活跃 Goroutine 非常多的程序中，重新扫描的过程需要占用 10 ~ 100ms 的时间。

大概就是说go1.7之前，在并发标记期间，栈上的黑对象直接指向了一个白对象，由于不会触发写屏障，那为了这个白对象不被清除，就需要在标记完成时STW，

重新标记栈对象。

在说混合写屏障之前，再来看看删除写屏障。

Yuasa-style 屏障

伪代码：

```
writePointer(slot, ptr)
    shade(*slot)
    *slot = ptr
```

总结：

1. 删除写屏障也叫基于快照的写屏障方案，必须在起始时，STW 扫描整个栈（注意了，是所有的 goroutine 栈），保证**所有堆上在用的对象**都处于灰色保护下，

   保证的是弱三色不变式；

2. 由于起始快照的原因，起始也是执行 STW，删除写屏障不适用于栈特别大的场景，栈越大，STW 扫描时间越长，对于现代服务器上的程序来说，栈地址空间

   都很大，所以删除写屏障都不适用，一般适用于很小的栈内存，比如嵌入式，物联网的一些程序；

3. 并且删除写屏障会导致扫描进度（波面）的后退，所以扫描精度不如插入写屏障；

GO 在混合写屏障之前并没有单独使用删除写屏障，Golang 的内存写屏障是由插入写屏障到混合写屏障过渡的。不过，虽然 Golang 从来没有直接使用删除写屏

障，但是混合写屏障却用到了删除写屏障的思路。

标准的删除写屏障就是开始GC之前必须STW，将所有根对象都标记为黑色，那么就会使所有可达对象都在灰色的保护下(根黑，下一级在堆上的全灰)，之后利用删

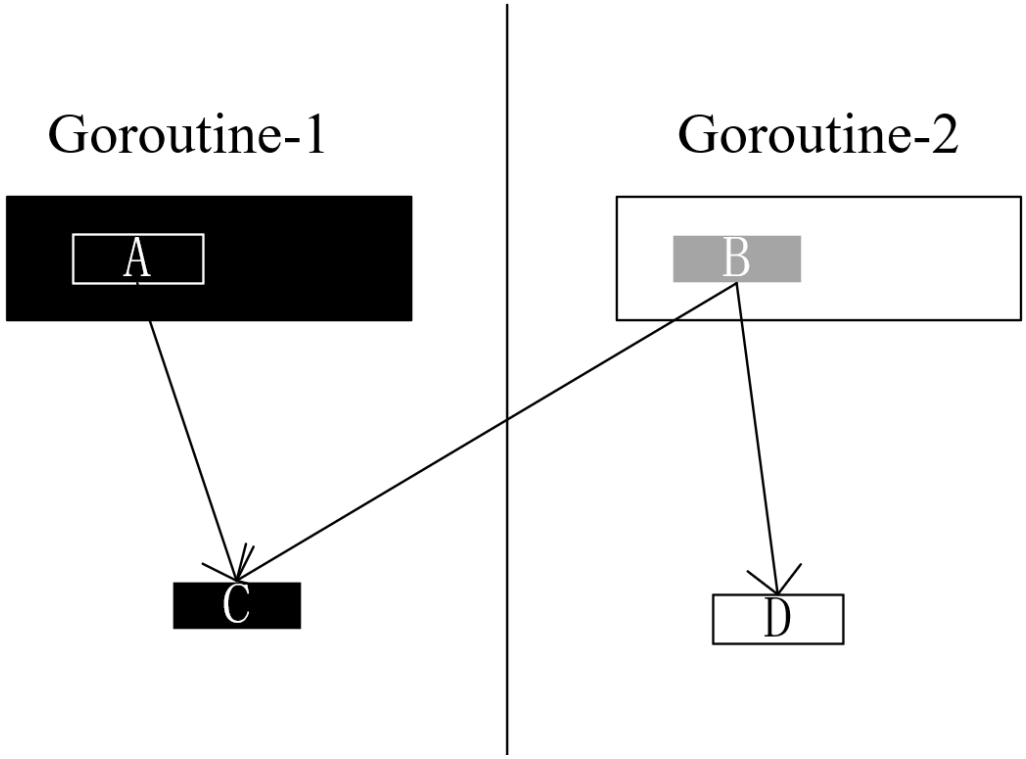
除写屏障捕捉内存写操作，确保弱三色不变式不被破坏，就可以保证垃圾回收的正确性。

思考问题：标准的删除写屏障在扫描栈上对象时，STW时间很长，那么我们是不是可以不用暂停所有的Goroutine，只暂停需要扫描栈的goroutine，也就不用全

局的STW了，这个时候再加上操作堆上对象时会添加的删除写屏障，是否能够满足不漏标的要求呢?

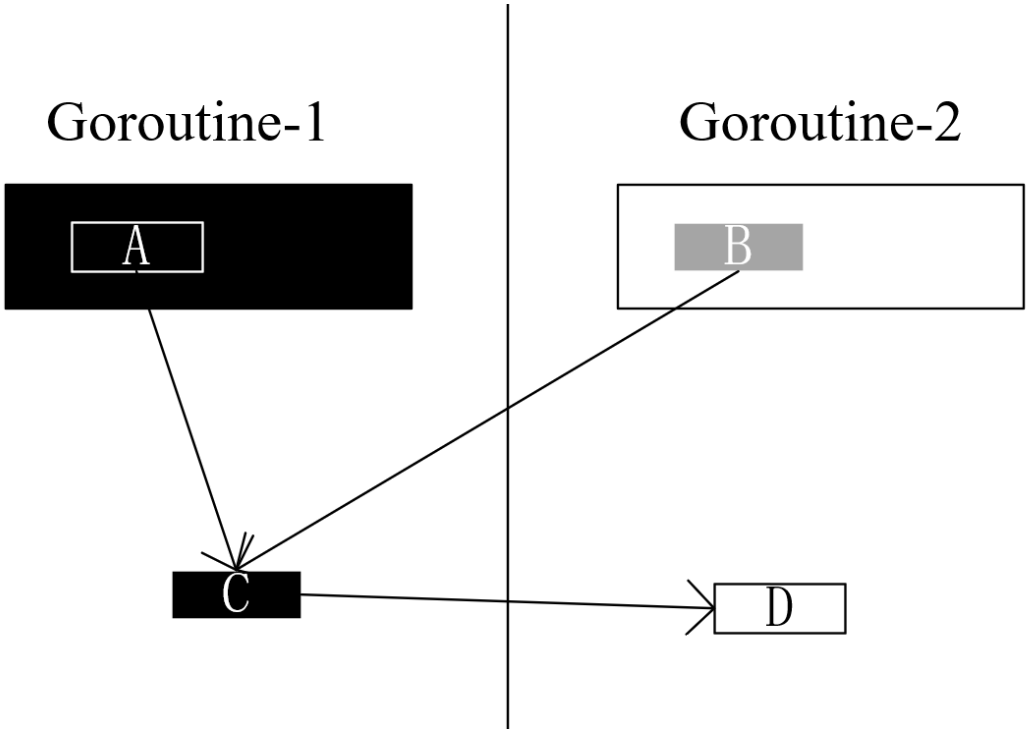
其实这样是不行的，也会出现漏标的情况！举例如下：

1. A 是 g1 栈的一个对象，g1栈已经扫描完了，并且 C 也是扫黑了的对象；
2. B 是 g2 栈的对象，指向了 C 和 D，g2 完全还没扫描，B 是一个灰色对象，D 是白色对象



步骤一：g2 进行赋值变更，把 C 指向 D 对象，这个时候黑色的 C 就指向了白色的 D（由于是删除屏障，这里是不会触发hook的）；

步骤二：把 B 指向 C 的引用删除，由于是栈对象操作，不会触发删除写屏障；



步骤三：清理，因为 C 已经是黑色对象了，所以不会再扫描，所以 D 就会被错误的清理掉。

解决办法有如下：

- 栈上对象也 hook，所有对象赋值（插入，删除）都 hook（这个就不实际了）所有的插入，删除如果都 hook ，那么一定都不会有问题，虽然本轮精度很差，

  但是下轮GC可以回收了。但是还是那句话，栈，寄存器的赋值 hook 是不现实的。

- STW把所有栈上对象都扫黑，使得整个堆上的在用对象都处于灰色保护；整栈扫黑，那么在用的堆上的对象是一定处于灰色堆对象的保护下的，之后配合堆对

  象删除写屏障就能保证在用对象不丢失。

- **加入插入写屏障的逻辑，C 指向 D 的时候，把 D 置灰，这样扫描也没问题。这样就能去掉起始 STW 扫描，从而可以并发，一个一个栈扫描。**

Go语言在1.8版本引入的混合写屏障就是如此，在开始标记时先将goroutine对应的栈标记为黑色。但是这个过程不是像标准的删除写屏障一样，暂停所有的g

oroutine，然后再开始标记栈上对象，而是暂停一个goroutine，去标记它的栈。在这期间其他goroutine是可以运行的，比如上面两个图g2改变了对象的引用关

系，此时只依靠删除写屏障，仍然会出现漏标的问题。那么再引入插入写屏障，就可以解决问题。

混合写屏障

golang 1.5 之后已经实现了插入写屏障，但是由于栈对象赋值无法 hook 的原因，导致扫描完之后还有一次 STW 重新扫描栈的整机停顿，混合写屏障就是解决这

个问题的。

论文里的伪代码：

```
writePointer(slot, ptr):
    shade(*slot)
    if current stack is grey:
        shade(ptr)
    *slot = ptr
```

golang 实际实现的伪代码：

```
writePointer(slot, ptr):
    shade(*slot)
    shade(ptr)
    *slot = ptr
```

在看了论文里的伪代码之后，我不明白为什么会有 if current stack is grey 这个判断，后来试着想了一下，不知道对不对。

第一个shade(*slot)就是删除写屏障，第二个shade(ptr)是插入写屏障。

其实标准的删除写屏障策略是首先STW然后标记所有栈对象，之后堆上的对象都是灰色保护 (grey-protected) 了，只需要删除写屏障就能够保证不漏标，但是GO

从始至终是没有采用过这种办法的，那混合写屏障加入插入写屏障的原因是什么呢？原因就在于Go语言实现三色标记时，不是全局STW然后再标记栈对象，而

是标记哪个goroutine的栈上对象时就暂停哪个 goroutine 前面的代码分析中也给出了代码逻辑，在标记协程标记暂停goroutine栈上的对象时，其他goroutine是

可以在运行的，比如发生思考问题中的情况，此时判断g2栈对象是灰对象，添加写屏障将D置为灰色，就不会发生漏标的情况。

我认为插入写屏障是解决在所有栈对象没标记完时，可能发生引用改变导致漏标这种情况的，而当所有栈对象已经标记为黑色了，只需要删除写屏障就可以保证不

漏标了，go还保证在gc 标记期间新创建的对象都是黑色的。

而go官方的实现中没有if current stack is grey 进行判断，删除写屏障和插入写屏障都会在引用改变时添加。也就是是说，如果在垃圾回收阶段，只要是堆上的一

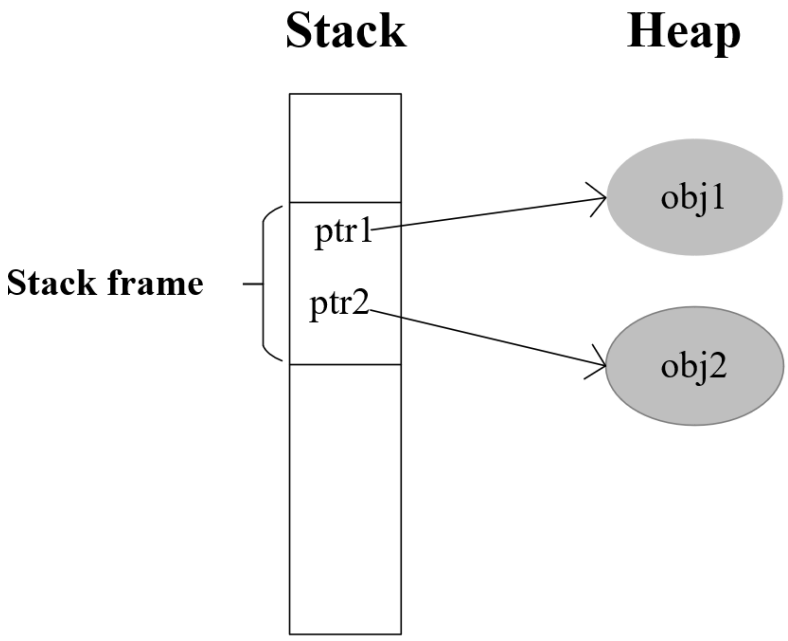
个赋值 * slot = ptr 那么都会被 hook 住，然后把旧值（*slot）指向的对象，和新值（ptr）指向的对象都置灰（投到扫描队列）。我暂时没能够分析出他们为

什么不采用第一种实现来减少开销。

另外还要说一点，在标记栈上对象，我们都在说把栈上对象标记为黑色，实际上根本就没有这个过程看以下源代码

```
func scanblock(b0, n0 uintptr, ptrmask *uint8, gcw *gcWork, stk *stackScanState)
{
    // Use local copies of original parameters, so that a stack trace
    // due to one of the throws below shows the original block
    // base and extent.
    b := b0 // 扫描开始的位置
    n := n0 // 扫描结束的位置
    // 枚举扫描的地址
    for i := uintptr(0); i < n; {
        // Find bits for the next word.
        // 每个 bit 标识一个 8 字节，8个 bit （1个字节）标识 64 个字节这里计算到合适的
bits
        bits := uint32(*addb(ptrmask, i/(sys.PtrSize*8)))
        if bits == 0 { // 如果整个 bits == 0，那么说明这 8 个 8 字节都没有指针引用，可以
直接跳到下一轮。
            i += sys.PtrSize * 8
            continue
        }
        for j := 0; j < 8 && i < n; j++ { // bits 非0，说明内部有指针引用，就必须一个个
扫描查看。
            if bits&1 != 0 { // 如果该地址包含指针
                // Same work as in scanobject; see comments there.
                p := *(*uintptr)(unsafe.Pointer(b + i)) // 把对应内存地址里面存储的值
取出来
                if p != 0 {
                    // 找到指针指向的对象对应的起始地址，mspan以及在mspan中的索引。
                    if obj, span, objIndex := findObject(p, b, i); obj != 0 {
                        greyobject(obj, b, i, span, gcw, objIndex) // 标记一个对象
存活，并把它加到标记队列(该对象变为灰色)
                    } else if stk != nil && p >= stk.stack.lo && p <
stk.stack.hi {
                        stk.putPtr(p, false)
                    }
                }
            }
            bits >>= 1 // 处理下一个指针下一个bit
            i += sys.PtrSize
        }
    }
}
```

可以看到当栈帧上有指针对象时，是将它们所指向的对象标记为灰色。只是这种效果等同于将栈上对象标记为黑色。



写屏障代码触发点

重点:

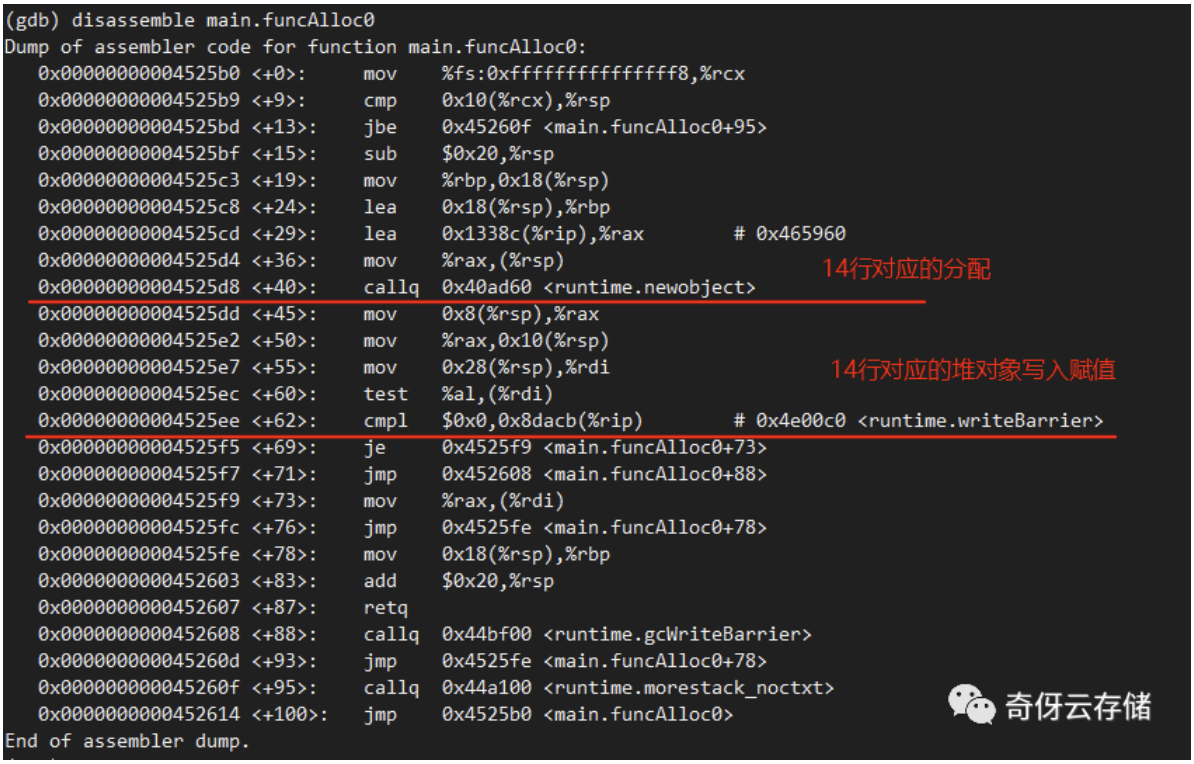
1. 写屏障的代码在编译期间生成好，之后不会再变化；
2. 堆上对象赋值才会生成写屏障；
3. 哪些对象分配在栈上，哪些分配在堆上？也是编译期间由编译器决定，这个过程叫做“逃逸分析”；

举例:

main.funcAlloc0

```
func funcAlloc0 (a *Tstruct) {
    a.base = new(BaseStruct)    // new 一个BaseStruct结构体，赋值给 a.base 字段
}
```

反汇编就能看到编译出的代码，如下:



在合适的位置，编译器给你程序添加的指令函数 runtime·gcWriteBarrier，这个就是入口，每次堆对象赋值，如果开启了垃圾回收开关，都会去里面转一圈。

实现伪代码如下：

```
if runtime.writeBarrier.enabled {
    runtime.gcWriteBarrier(ptr, val)
} else {
    *ptr = val
}
```

注意参数 gcWriteBarrier 传值在 golang 里是个特例，这里用的是寄存器（ golang 的惯例是用栈来传递的，但是考虑性能原因，这里必须用寄存器了）

runtme·gcWriteBarrier

这个函数是个纯汇编的函数，golang 按照不同的 cpu 指令集实现的，路径可以去看 src/runtime/asm_amd64.s这个文件。

```
TEXT runtime·gcWriteBarrier<ABIInternal>(SB),NOSPLIT,$112
    // Save the registers clobbered by the fast path. This is slightly
    // faster than having the caller spill these.
    MOVQ    R12, 96(SP)
    MOVQ    R13, 104(SP)
    // TODO: Consider passing g.m.p in as an argument so they can be shared
    // across a sequence of write barriers.
#ifdef GOEXPERIMENT_regabig
    MOVQ    g_m(R14), R13
#else
    get_tls(R13)
    MOVQ    g(R13), R13
    MOVQ    g_m(R13), R13
#endif
    MOVQ    m_p(R13), R13
    MOVQ    (p_wbBuf+wbBuf_next)(R13), R12
    // Increment wbBuf.next position.
```

```
    LEAQ    16(R12), R12
    MOVQ    R12, (p_wbBuf+wbBuf_next)(R13)
    CMPQ    R12, (p_wbBuf+wbBuf_end)(R13) // 检查p的wbBuf(写屏障缓冲区)是否满?
    // Record the write. 记录写入
    MOVQ    AX, -16(R12)    // Record value 记录新值
    // Note: This turns bad pointer writes into bad
    // pointer reads, which could be confusing. We could avoid
    // reading from obviously bad pointers, which would
    // take care of the vast majority of these. We could
    // patch this up in the signal handler, or use XCHG to
    // combine the read and the write.
    MOVQ    (DI), R13
    MOVQ    R13, -8(R12)    // Record *slot 记录旧值
    // Is the buffer full? (flags set in CMPQ above)
    JEQ flush
ret: // 赋值: *slot = val 本来就是要做的事
    MOVQ    96(SP), R12
    MOVQ    104(SP), R13
    // Do the write.
    MOVQ    AX, (DI)
    RET

flush:
    // Save all general purpose registers since these could be
    // clobbered by wbBufFlush and were not saved by the caller.
    // It is possible for wbBufFlush to clobber other registers
    // (e.g., SSE registers), but the compiler takes care of saving
    // those in the caller if necessary. This strikes a balance
    // with registers that are likely to be used.
    //
    // We don't have type information for these, but all code under
    // here is NOSPLIT, so nothing will observe these.
    //
    // TODO: We could strike a different balance; e.g., saving X0
    // and not saving GP registers that are less likely to be used.
    MOVQ    DI, 0(SP)   // Also first argument to wbBufFlush
    MOVQ    AX, 8(SP)   // Also second argument to wbBufFlush
    MOVQ    BX, 16(SP)
    MOVQ    CX, 24(SP)
    MOVQ    DX, 32(SP)
    // DI already saved
    MOVQ    SI, 40(SP)
    MOVQ    BP, 48(SP)
    MOVQ    R8, 56(SP)
    MOVQ    R9, 64(SP)
    MOVQ    R10, 72(SP)
    MOVQ    R11, 80(SP)
    // R12 already saved
    // R13 already saved
    // R14 is g
    MOVQ    R15, 88(SP)

    // This takes arguments DI and AX
    CALL    runtime·wbBufFlush(SB) //  写屏障缓冲队列满了，统一处理，这个其实是一个批量优
化手段

    MOVQ    0(SP), DI
    MOVQ    8(SP), AX
```

```
    MOVQ    16(SP), BX
    MOVQ    24(SP), CX
    MOVQ    32(SP), DX
    MOVQ    40(SP), SI
    MOVQ    48(SP), BP
    MOVQ    56(SP), R8
    MOVQ    64(SP), R9
    MOVQ    72(SP), R10
    MOVQ    80(SP), R11
    MOVQ    88(SP), R15
    JMP ret
```

runtime·gcWriteBarrier函数干啥的，这个函数其实只干两件事：

1. 执行写请求（原本就要做的事情）
2. 处理 GC 相关的逻辑（投队列，置灰色保护）

那为什么上面 gcWriteBarrier 这个函数怎么复杂?

其实是做的一个优化处理，每次触发写屏障的时候（ hook ），我们当然可以直接shade（ptr），但是我们知道，毕竟这段写屏障的代码是比业务多出来的，这些

都是开销，我们能快就快，每次这样做太零散，我们可以攒一批，一批队列满了，一批去入队，置灰色。这样效率更高。一般情况下，只需要简单入队就行了，

buf 满了之后，才 flush 去批量置灰，这样写屏障对业务的影响就更小了，wbBuf 就是这个队列的实现。相当于就是消费写屏障缓冲中的消息。

wbBufFlush

这个函数 wbBufFlush 是 golang 实现的，在文件 src/runtime/mwbbuf.go 里。本质上是封装调用 wbBufFlush1 ，这个函数才是 hook 写操作想要做的事情，

这个函数做两个事情：

1. 批量循环处理 buf 队列里的值；
2. shade（这个值）；

```
func wbBufFlush1(_p_ *p) {
    // Get the buffered pointers.
    start := uintptr(unsafe.Pointer(&_p_.wbBuf.buf[0]))
    n := (_p_.wbBuf.next - start) / unsafe.Sizeof(_p_.wbBuf.buf[0])
    ptrs := _p_.wbBuf.buf[:n]

    // Poison the buffer to make extra sure nothing is enqueued
    // while we're processing the buffer.
    _p_.wbBuf.next = 0

    if useCheckmark {
        // Slow path for checkmark mode.
        for _, ptr := range ptrs {
            shade(ptr)
        }
        _p_.wbBuf.reset()
        return
    }

    // Mark all of the pointers in the buffer and record only the
    // pointers we greyed. We use the buffer itself to temporarily
```

```
    // record greyed pointers.
    //
    // TODO: Should scanobject/scanblock just stuff pointers into
    // the wbBuf? Then this would become the sole greying path.
    //
    // TODO: We could avoid shading any of the "new" pointers in
    // the buffer if the stack has been shaded, or even avoid
    // putting them in the buffer at all (which would double its
    // capacity). This is slightly complicated with the buffer; we
    // could track whether any un-shaded goroutine has used the
    // buffer, or just track globally whether there are any
    // un-shaded stacks and flush after each stack scan.
    gcw := &_p_.gcw // 拿到p的工作队列
    pos := 0
    for _, ptr := range ptrs { // ptrs是存储着写屏障缓冲区数据的切片，切片的内容就是赋值前后新旧两个对象的地址吧(我猜的)
        if ptr < minLegalPointer {
            // nil pointers are very common, especially
            // for the "old" values. Filter out these and
            // other "obvious" non-heap pointers ASAP.
            //
            // TODO: Should we filter out nils in the fast
            // path to reduce the rate of flushes?
            continue
        }
        obj, span, objIndex := findObject(ptr, 0, 0) // 根据地址找到对象基地址，对象所在mspan，对象在mspan中的索引(这个函数功能这么骚的吗？)
        if obj == 0 {
            continue
        }
        // TODO: Consider making two passes where the first
        // just prefetches the mark bits.
        mbits := span.markBitsForIndex(objIndex) // 从mspan的gcmarkBits位图找到对象的标记位
        if mbits.isMarked() { // 如果对象已经标记过了
            continue
        }
        mbits.setMarked() // 标记此对象在gcmarkBits位图对应位置

        // Mark span.
        arena, pageIdx, pageMask := pageIndexOf(span.base())
        if arena.pageMarks[pageIdx]&pageMask == 0 {
            atomic.Or8(&arena.pageMarks[pageIdx], pageMask)
        }

        if span.spanclass.noscan() {
            gcw.bytesMarked += uint64(span.elemsize)
            continue
        }
        ptrs[pos] = obj
        pos++
    }

    // Enqueue the greyed objects.
    gcw.putBatch(ptrs[:pos]) // 将对象放入标记工作队列中

    _p_.wbBuf.reset()
}
```

对象置灰

说了这么久“置灰色”，那么到底写屏障是怎么置灰色的？实现如何。其实本质下就下面一行代码，在文件 src/runtime/mwbbuf.go 的函数 wbBufFlush1：

```
// Enqueue the greyed objects.
gcw.putBatch(ptrs[:pos])
```

在 golang 里面，到底什么样的是灰色对象？

1. 只要在扫描队列中的对象，就是灰色的。

其实，白，灰，黑 三色这个是我们认为抽象出来的概念，也就是所谓的三色标记法，那么这个概念落到实处，又是怎么样的实现。

golang 内部对象并没有保存颜色的属性，三色只是对他们的状态的描述，是通过一个队列 + 掩码位图来实现的：

- 白色对象：对象所在 span 的 gcmarkBits 中对应的 bit 为 0，不在队列；
- 灰色对象：对象所在 span 的 gcmarkBits 中对应的 bit 为 1，且对象在扫描队列中；
- 黑色对象：对象所在 span 的 gcmarkBits 中对应的 bit 为 1，且对象已经从扫描队列中处理并摘除掉；

暂时只想写到这里了，以后再慢慢优化，甚至纠错。下面开始总结。

总结

Go 现在的GC机制和Java中的CMS垃圾收集器很像，Java中的CMS垃圾收集器分为四个阶段：初始标记，并发标记，重新标记，并发清除。

会在初始标记以及重新标记阶段STW。CMS第一次STW的原因是如果不 STW , 在程序运行的过程中，会有不断的局部变量出现，这样我们就不知道要标记到什么时

候才算结束了。第二次STW的原因是为防止漏标重新扫描young gen，regieter，global object，以及mod union table。

参考

Golang源码探索(三) GC的实现原理

golang 混合写屏障原理深入剖析，这篇文章给你梳理的明明白白！！！

粗线条话GC（三）

Go语言的设计与实现