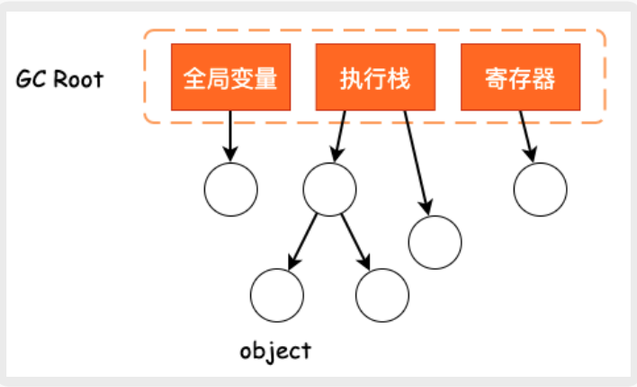
GC

概述

Go 所实现的 GC 方式为标记-清除，没有使用引用计数以及分代回收，并且也没有对回收的对象进行内存整理，因为 Go 使用的是基于 tcmalloc 的内存分配算法，对对象进行整理不会带来实质性的性能提升



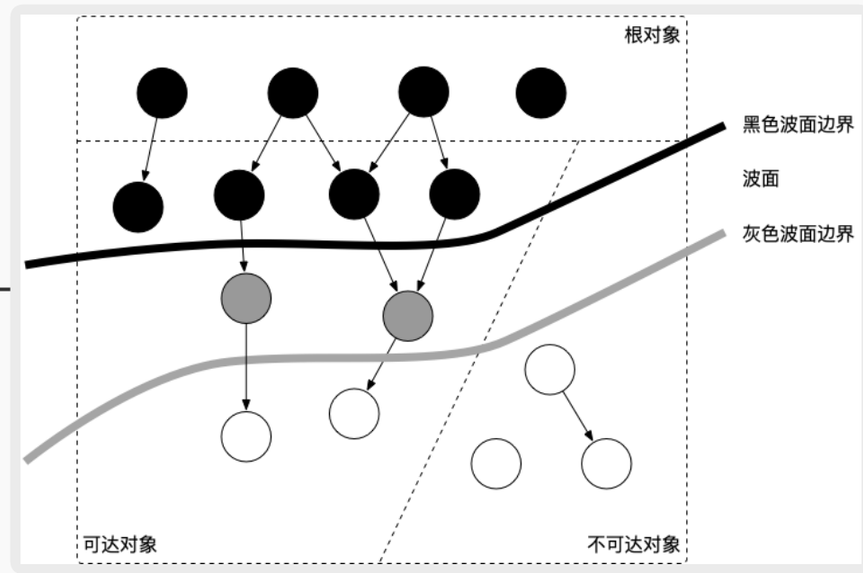
- 标记-清除简单的来说就是从 GC Root 开始，逐一地向下进行遍历，从 GC Root 到当前对象存在可达路径时，认为对象存活。不存在可达路径时，那么该对象应该被标记为可回收，后续对该对象所在堆内存进行回收
- GC Root 主要包含了编译器确定的全局变量，goroutine 执行栈（重点扫描区域）以及寄存器中所保存的指针

Go version 1.3

- 在 1.3 版本之前，Go 使用最简单的标记-清除方式，也就是当需要进行 GC 时，停止所有 goroutine 的运行，开始进行标记，标记完了以后对需要回收的对象进行清理，事情做完了以后再回复所有 goroutine 的运行
- 停止所有 goroutine 运行有一个简称，Stop The World，STW。很明显，这种方式的 STW 时间将会很长，可能达到几百毫秒，在此期间其它 goroutine 都不可运行

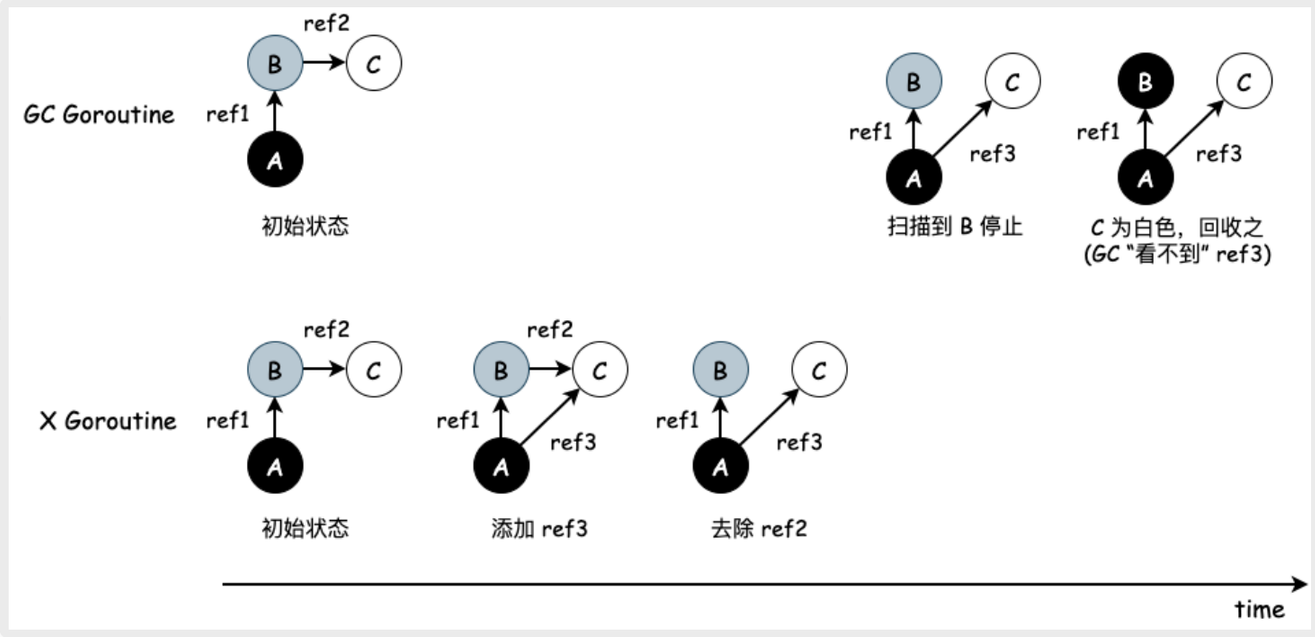
Go version 1.5

- 在 1.5 版本中，Go 引入了三色标记+屏障机制来优化 STW 所持续的时间
- 三色标记
  - 逻辑颜色定义
    - 白色（可能死亡）：在回收开始阶段，所有对象均为白色，当回收结束后，白色对象均不可达
    - 灰色（正在扫描）：已被回收器访问到的对象，但回收器需要对其中的一个或多个指针进行扫描，因为他们可能还指向白色对象
    - 黑色（确认存活）：已被回收器访问到的对象，其中所有字段都已被扫描，黑色对象中任何一个指针都不可能直接指向白色对象



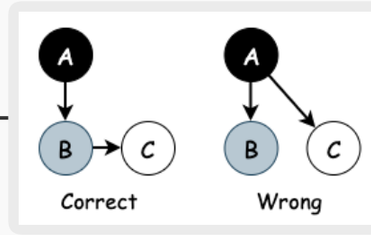
在标记开始后，黑色波面边界以及灰色波面边界会不断往前推进，直到所有可达的灰色对象都变为黑色对象为止

  - 标记过程
    1. 在初始化状态时，所有的对象均为白色。GC 将从根节点开始遍历，将遍历到的节点置为灰色
    2. 然后，将灰色节点所引用的白色对象置为灰色，并将原有灰色对象置为黑色。就像上图一样，黑色边界和灰色边界是逐层推进的，而不是一下递归到底
    3. 最后，回收所有白色对象，一次 GC 结束
- 为什么三色标记需要 STW
  - 因为 Go GC 是和其他 goroutine 并发执行的，所以依然可能存在并发的竞态条件，也就是说，即使是使用三色标记-清除，依然需要 STW

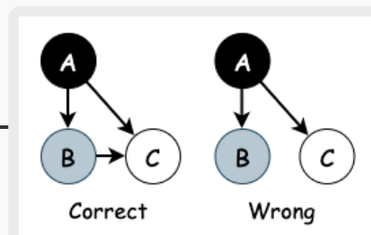


  - 简而言之，当 GC 扫描到 B 对象时，ref2 已经被干掉，无法通过 B 找到白色对象 C，但是另一个 goroutine 在已经被标记为黑色的 A 对象中添加了一个对象 ref3，而 ref3 是不会被 GC 所扫描到的，因为 A 已经处理了。所以，C 对象将会作为白色对象被清理，但实际上这个清理是错误的
  - 我们来简单分析一下在没有 STW 的情况下为什么会出现错误清理的情况
    1. 一个白色的对象挂在了黑色对象上
    2. 灰色对象在 GC 时与其白色对象的连接关系丢失

    所以，我们需要一个机制来保证这两种条件不会发送，那么我们就可以大胆的执行并发 GC 了
  - 屏障机制
    - "强-弱"三色不变式
      - 强三色不变式
        - 从三色标记的过程中我们可以看到，黑色对象下面挂的一定是灰色对象，灰色对象下面挂的一定是白色对象，强三色不变使规定: 不允许出现黑色对象下面挂白色对象



      - 弱三色不变式
        - 黑色对象可以引用白色对象，但前提是该白色对象的上游链路中存在灰色对象



    - 插入屏障 (满足强三色不变式)
      - 在 GC 进行时，若 A 对象引用 B 对象，那么 B 对象需要被标记为灰色。那么此时就不会存在黑色对象引用白色对象的情况了，因为白色对象已经变成了灰色对象。插入屏障仅在堆内存中生效，栈内存仍然需要使用 STW 进行二次扫描
    - 删除屏障 (满足弱三色不变式)
      - 当某 A 对象需要被删除时，将其标记为灰色，可以防止白色对象丢失的情况
      - 但是同时也可能造成对象在本轮 GC 中未被回收，需要在下一轮 GC 才能被回收的情况

Go version 1.8

- 插入屏障和写入屏障的短板
  - 插入屏障：仅在堆内存中生效，因此栈内存在一轮扫描以后需要对栈空间进行 STW，以确保不会"误杀"白色对象
  - 删除屏障：回收精度会受到影响，GC 开始以后，所有被删除的对象都会被标记为灰色，那么这些原本需要被回收的对象只能在下次 GC 开始时才会被回收
- 因此，1.8 版本中新增了混合写屏障，主要是针对于栈空间的 GC 而进行的优化
- 混合屏障规则：GC 开始时将栈上的对象进行扫描，并将可达对象全部标记为黑色。对于栈上新增的对象，若当前栈未扫描完毕，则将其置为灰色。若当前栈已经扫描完毕，则将新增的对象置为黑色

总结

Go 的 GC 最早期使用的回收算法是标记-清除算法，该算法需要在执行期间需要暂停应用程序(STW)，无法满足并发程序的实时性。后面 Go 的 GC 转为使用三色标记清除算法，并通过混合写屏障技术保证了 Go 并发执行 GC 时内存中对象的三色一致性（这里的并发指的是 GC 和应用程序的 goroutine 能同时执行）。

一次完整的垃圾回收会分为四个阶段，分别是标记准备、标记、结束标记以及清理。在标记准备和标记结束阶段会需要 STW，标记阶段会减少程序的性能，而清理阶段是不会对程序有影响的。