go基础学习——垃圾回收

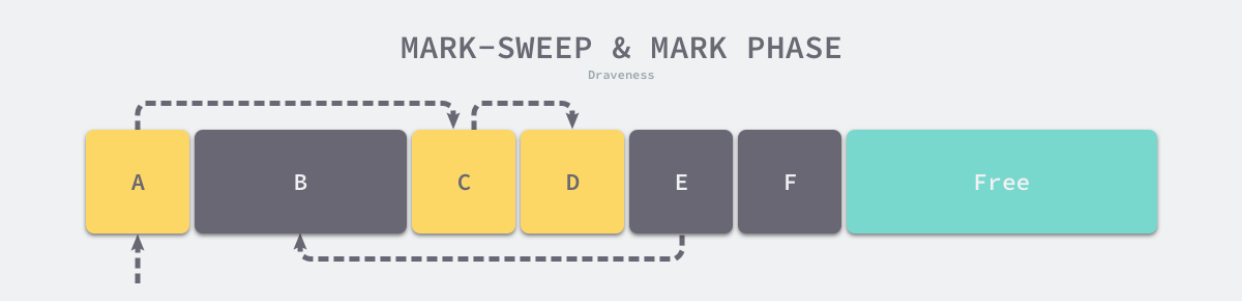
古老的标记清除

垃圾回收当中的标记清除算法是比较常见的算法，其主要分成两个过程：1.标记2.清除

标记过程主要是从根对象出发，遍历所有存活对象并进行标记

清除过程主要是对标记过程中未标记的对象，并将回收的对象加入到链表之中

在标记和清除阶段，都是STW类型。



三种垃圾回收算法比较：

标记清除：
1.执行效率不稳定，如果堆中包含大量对象，而且其中大部分是需要被回收的，这时必须进行大量标记和清除的动作，导致标记和清除两个过程的执行效率都随对象数量增长而降低
2.内存空间的碎片化问题，标记、清除之后会产生大量不连续的内存碎片
标记复制：
1.需要额外的空间留给复制对象，浪费内存空间
2.如果存活对象多，则复制开销大
标记整理：
如果移动存活对象，尤其是在老年代这种每次回收都有大量对象存活区域，移动存活对象并更新所有引用这些对象的地方将会是一种极为负重的操作，而且这种对象移动操作必须全程暂停用户应用程序才能进行。

三色标记+内存屏障

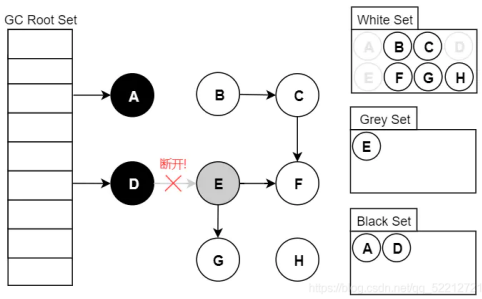
三色标记我最初了解是在《深入了解JVM虚拟机》书中，当中讲了JAVA的垃圾回收机制，其中的G1垃圾收集器就是运用了三色标记和内存屏障技术，当时还不是特别理解，现在进行重新的学习。

三色标记算法将程序中的对象**分为白色、黑色和灰色**三种颜色，其中，白色对象代表了可能被垃圾回收的对象；黑色对象主要有两种，一种是不存在任何引用外部指针的对象，另一种是从根直接可达的对象；灰色对象与黑色对象相比，有指向白色对象的指针，所以垃圾回收时需要遍历灰色对象的子对象。

三色标记过程 1.首先三色标记刚开始并不存在黑色节点，将根结点标记为灰色节点，然后从灰色节点中选取子节点2.从灰色节点中选取子节点后，将原节点变成黑色节点，然后将原节点的子节点全部变成灰色，循环该过程

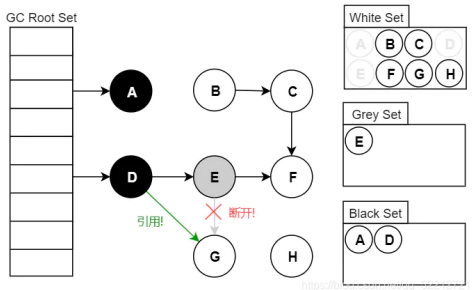
产生问题：因为用户程序可能在标记执行的过程中修改对象的指针，所以三色标记清除算法本身是不可以并发或者增量执行的。如下过程介绍：

假设已经遍历到E（变为灰色了），此时应用执行了 objD.fieldE = null ：



此刻之后，对象E/F/G是“应该”被回收的。然而因为E已经变为灰色了，其仍会被当作存活对象继续遍历下去。最终的结果是：这部分对象仍会被标记为存活，即本轮GC不会回收这部分内存。

这部分本应该回收 但是 没有回收到的内存，被称之为“浮动垃圾”。浮动垃圾并不会影响应用程序的正确性，只是需要等到下一轮垃圾回收中才被清除。
这种还好，还是可以清除的，但下面这种：



此时切回GC线程继续跑，因为E已经没有对G的引用了，所以不会将G放到灰色集合；尽管因为D重新引用了G，但因为D已经是黑色了，不会再重新做遍历处理。
最终导致的结果是：G会一直停留在白色集合中，最后被当作垃圾进行清除。这直接影响到了应用程序的正确性，是不可接受的。

为了解决该问题，各语言分别使用了不同的手段：

在Java当中：

CMS：写屏障+增量更新

当对象D的成员变量的引用发生变化时（objD.fieldG = G;），我们可以利用写屏障，将D新的成员变量引用对象G记录下来：
【当有新引用插入进来时，记录下新的引用对象】
这种做法的思路是：不要求保留原始快照，而是针对新增的引用，将其记录下来等待遍历，即增量更新（Incremental Update）。

G1:写屏障+SATB

当对象E的成员变量的引用发生变化时（objE.fieldG = null;），我们可以利用写屏障，将E原来成员变量的引用对象G记录下来：
【当原来成员变量的引用发生变化之前，记录下原来的引用对象】
这种做法的思路是：尝试保留开始时的对象图，即原始快照（Snapshot At The Beginning，SATB），当某个时刻 的GC Roots确定后，当时的对象图就已经确定了。
比如 当时 D是引用着G的，那后续的标记也应该是按照这个时刻的对象图走（D引用着G）。如果期间发生变化，则可以记录起来，保证标记依然按照原本的视图来。
值得一提的是，扫描所有GC Roots 这个操作（即初始标记）通常是需要STW的，否则有可能永远都扫不完，因为并发期间可能增加新的GC Roots。

ZGC：读屏障

而在go当中，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

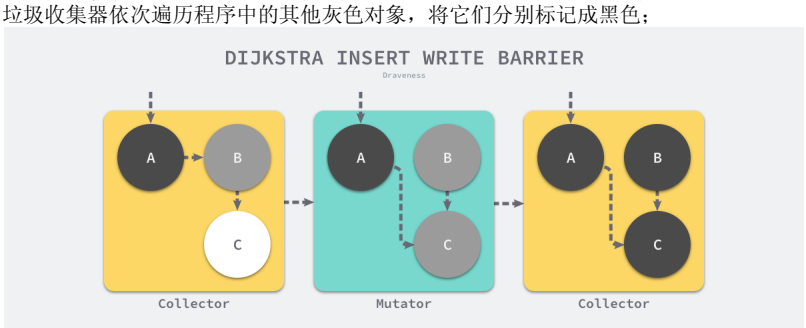
三色不变性

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

插入写屏障和删除写屏障

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：

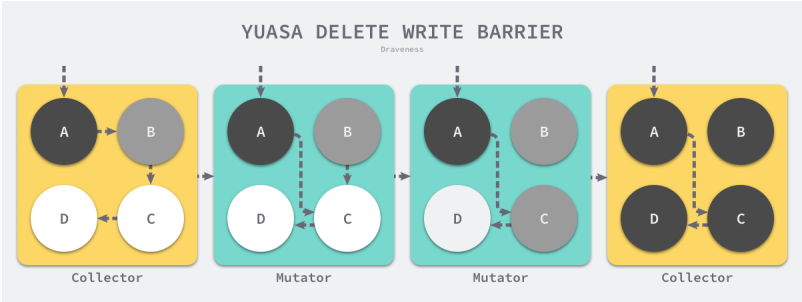
1. 垃圾收集器将根对象指向 A 对象标记成黑色并将 A 对象指向的对象 B 标记成灰色；
2. 用户程序修改 A 对象的指针，将原本指向 B 对象的指针指向 C 对象，这时触发写屏障将 C 对象标记成灰色；
3. 垃圾收集器依次遍历程序中的其他灰色对象，将它们分别标记成黑色；



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；



go中的垃圾回收流程

Sweep Termination: 对未清扫的span进行清扫, 只有上一轮的GC的清扫工作完成才可以开始新一轮的GC

Mark: 扫描所有根对象, 和根对象可以到达的所有对象, 标记它们不被回收

Mark Termination: 完成标记工作, 重新扫描部分根对象(要求STW)

Sweep: 按标记结果清扫span