前言

本文主要讲解三色标记具体工作原理，多标导致的浮动垃圾、漏标的处理方案（读写屏障）等。

基本概念

JVM 中的垃圾回收是基于 标记-复制、标记-清除和标记-整理三种模式的，那么其中最重要的其实是如何标记，像 Serial、Parallel 这类的回收器，无论是单线程标记和多线程标记，其本质采用的是暂停用户线程进行全面标记的算法，这种算法的好处就是标记的很干净，而且实现简单，缺点就是标记时间相对很长，导致 STW 的时间很长；

那么后来就有了并发标记，适用于 CMS 和 G1，并发标记的意思就是可以在不暂停用户线程的情况下对其进行标记，那么实现这种并发标记的算法就是三色标记法，三色标记法最大的特点就是可以异步执行，从而可以以中断时间极少的代价或者完全没有中断来进行整个 GC。

无论使用哪种算法，标记总是必要的一步。这是理算当然的，你不先找到垃圾，怎么进行回收?

基本流程

垃圾回收器的工作流程大体如下：

1.标记出哪些对象是存活的，哪些是垃圾（可回收）；

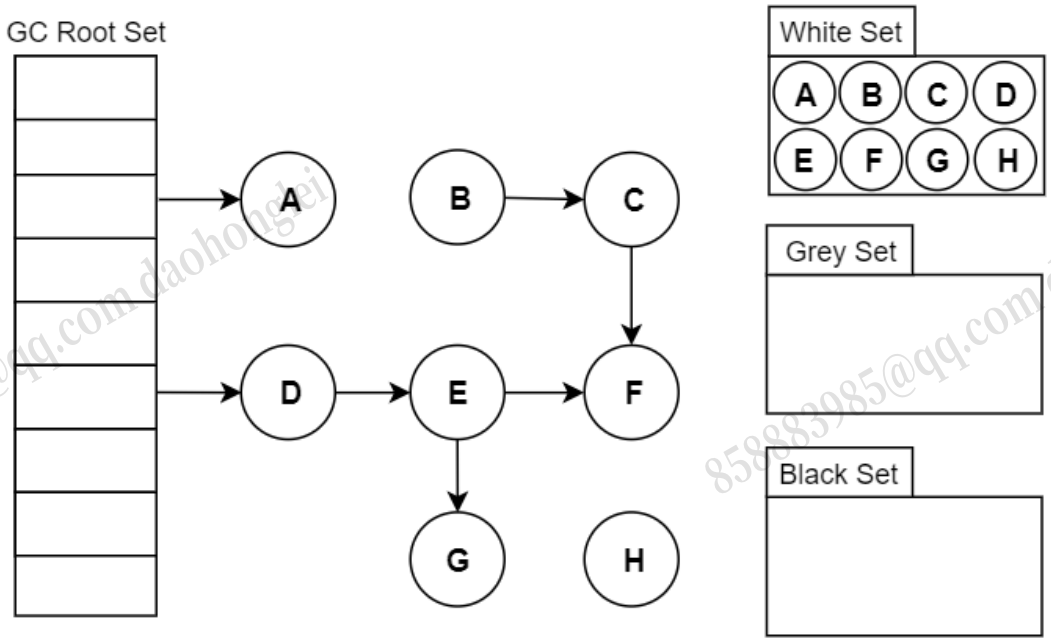
2.进行回收（清除/复制/整理），如果有移动过对象（复制/整理），还需要更新引用。

本文着重来看下标记的部分。

三色标记

基本算法

要找出存活对象，根据可达性分析，从 GC Roots 开始进行遍历访问，可达的则为存活对象：



最终结果：A/D/E/F/G 可达

我们把遍历对象图过程中遇到的对象，按“是否访问过”这个条件标记成以下三种颜色：

白色：尚未被 GC 访问过的对象，如果全部标记已完成依旧为白色的，称为不可达对象，既垃圾对象。

黑色：本对象已经被 GC 访问过，且本对象的子引用对象也已经被访问过了。

灰色：本对象已访问过，但是本对象的子引用对象还没有被访问过，全部访问完会变成黑色，属于中间态。

标记过程

1.在 GC 并发标记刚开始时，所以对象均为白色集合。

2.将所有 GCRoots 直接引用的对象标记为灰色集合。

3.判断若灰色集合中的对象不存在子引用，则将其放入黑色集合，若存在子引用对象，则

将其所有的子引用对象放入灰色集合，当前对象放入黑色集合。

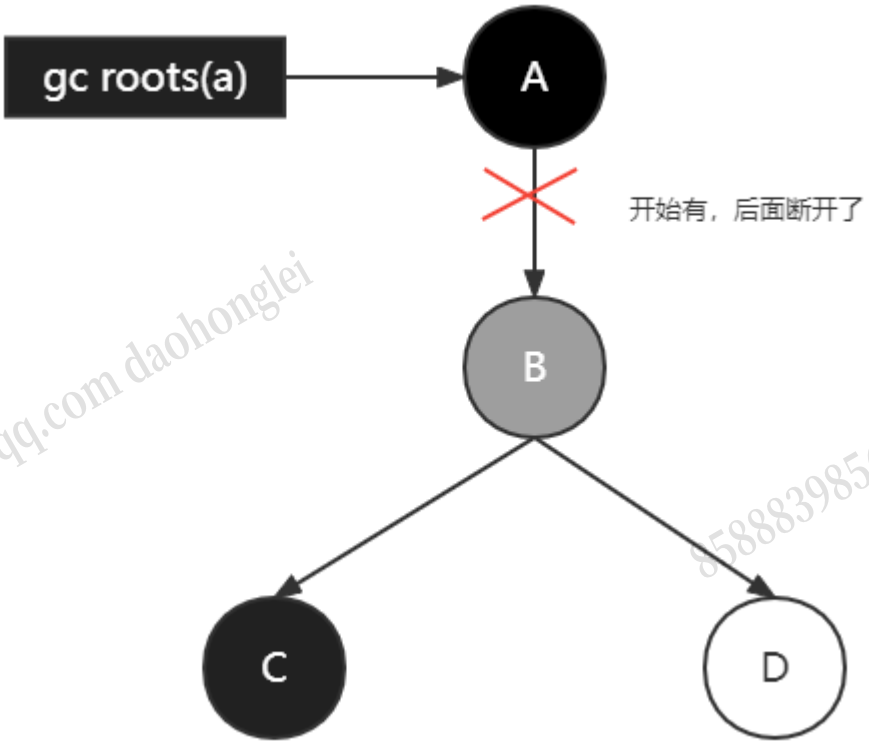
4.按照步骤三，以此类推，直至灰色集合中的所有对象变成黑色后，本轮标记完成，且当前白色集合内的对象称为不可达对象，既垃圾对象。

多标与漏标

问题：由于此过程是在和用户线程并发运行的情况下，对象的引用处于随时可变的情况下，那么就会造成多标和漏标的问题。

多标与浮动垃圾

如图：开始有 A->B 的引用，但此时应用执行 objA.fieldB = null，将引用断开



此刻之后，对象 B/C/D 是“应该”被回收的。然而因为 B 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。

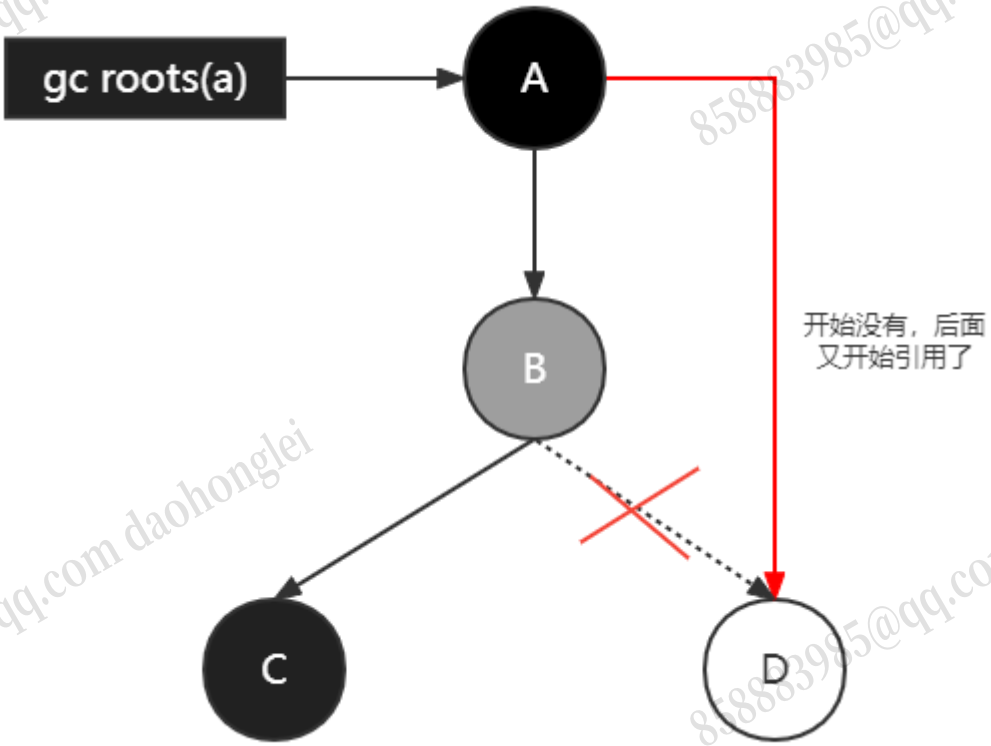
漏标

如图：开始没有 A->D 的引用，但此时应用执行如下代码

```
var D = objB.fieldD;

objB.fieldD = null;  // 灰色 B 断开引用 白色 D

objA.fieldD = D;  // 黑色 A 引用 白色 D
```



最终导致的结果是：D 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。

读写屏障 （类似 AOP 思想）

不难分析，漏标只有同时满足以下两个条件时才会发生：

条件一：灰色对象 断开了 白色对象的引用（直接或间接的引用）；即灰色对象的原来成员变量的引用发生了变化。

条件二：黑色对象 重新引用了 该白色对象；即黑色对象 成员变量增加了 新的引用。

从代码的角度看：

```
var D = objB.fieldD; //1.读
objB.fieldD = null;  // 2.写
objA.fieldD = D;  // 3.写
```

1.读取 对象 B 的成员变量 fieldD 的引用值，即对象 D；

2.对象 B 往其成员变量 fieldD，写入 null 值。

3.对象 A 往其成员变量 fieldD，写入 对象 D；

我们只要在上面这三步中的任意一步中做一些“手脚”，将对象 G 记录起来，然后作为灰色对象再进行遍历即可。比如放到一个特定的集合，等初始的 GC Roots 遍历完（并发标记），该集合的对象 遍历即可（重新标记）。

重新标记通常是需要 STW 的，因为应用程序一直在跑的话，该集合可能会一直增加新的对象，导致永远都跑不完。当然，并发标记期间也可以将该集合中的大部分先跑了，从而缩短重新标记 STW 的时间，这个是优化问题了。

1.写屏障用于拦截第二和第三步；而读屏障则是拦截第一步。

2.它们的拦截的目的很简单：就是在读写前后，将对象 D 给记录下来。

写屏障

给某个对象的成员变量赋值时，其底层代码大概长这样：

```
/**
```

```
* @param field 某对象的成员变量，如 D.fieldG
* @param new_value 新值，如 null
*/
void oop_field_store(oop* field, oop new_value) {
    *field = new_value; // 赋值操作
}
```

所谓的写屏障，其实就是指在赋值操作前后，加入一些处理（可以参考 AOP 的概念）：

```
void oop_field_store(oop* field, oop new_value) {
    pre_write_barrier(field); // 写屏障-写前操作
    *field = new_value;
    post_write_barrier(field, value);  // 写屏障-写后操作
}
```

(1) 写屏障 + SATB

当对象 B 的成员变量的引用发生变化时（objB.fieldD = null;），我们可以利用写屏障，将 B 原来成员变量的引用对象 D 记录下来（保存到一个集合中）：

```
void pre_write_barrier(oop* field) {
    oop old_value = *field; // 获取旧值
    remark_set.add(old_value); // 记录 原来的引用对象
}
```

【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。

重新标记阶段，再将集合中的对象重新标记为黑色，那么此时就不会回收这些对象了，但是会产生浮动垃圾（比如 objA.fieldB = null 此时也会记录该操作，B 对象不会被回收）

SATB 破坏了条件一：【灰色对象断开了白色对象的引用】，从而保证了不会漏标。

应用

垃圾回收器 G1 默认机制，优点是效率较高，但会产生浮动垃圾

(2) 写屏障 + 增量更新

当对象 A 的成员变量的引用发生变化时（objA.fieldD = D;），我们可以利用写屏障，将 A 新的成员变量引用对象 D 记录下来（保存到一个集合中）：

```
void post_write_barrier(oop* field, oop new_value) {
 if($gc_phase == GC_CONCURRENT_MARK && !isMarkd(field)) {
    remark_set.add(new_value); // 记录新引用的对象
 }
}
```

【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）。

重新标记阶段，再将集合中的对象重新扫描标记一遍，那么此时就获取了最终的对象图，也不会产生浮动垃圾。

增量更新破坏了条件二：【黑色对象 重新引用了 该白色对象】，从而保证了不会漏标。

应用

垃圾回收器 CMS 默认机制，优点不会产生浮动垃圾，但效率较低。

读屏障

```
oop oop_field_load(oop* field) {
    pre_load_barrier(field); // 读屏障-读取前操作
    return *field;
}
```

读屏障是直接针对第一步：var D = objB.fieldD;;，当读取成员变量时，一律记录下来：

```
void pre_load_barrier(oop* field, oop old_value) {
  if($gc_phase == GC_CONCURRENT_MARK && !isMarkd(field)) {
      oop old_value = *field;
      remark_set.add(old_value); // 记录读取到的对象
  }
}
```

这种做法是保守的，但也是安全的。因为条件二中【黑色对象 重新引用了 该白色对象】，重新引用的前提是：得获取到该白色对象，此时已经读屏障就发挥作用了。

为什么 G1 用 SATB？CMS 用增量更新?

我的理解：

SATB 相对增量更新效率会高（当然 SATB 可能造成更多的浮动浮动垃圾），因为不需要在重新标记阶段深度扫描被删除引用对象。CMS 对增量引用的根对象会做深度扫描，G1 因为很多对象都位于不同的 region，CMS 就一块老年代区域，重新扫描的话 G1 代价

会比 CMS 高，所以 G1 选择 SATB 不深度扫描对象，只是简单的标记，等到下一轮 GC 再做深度扫描。