JVM第十七天-并发标记算法原理

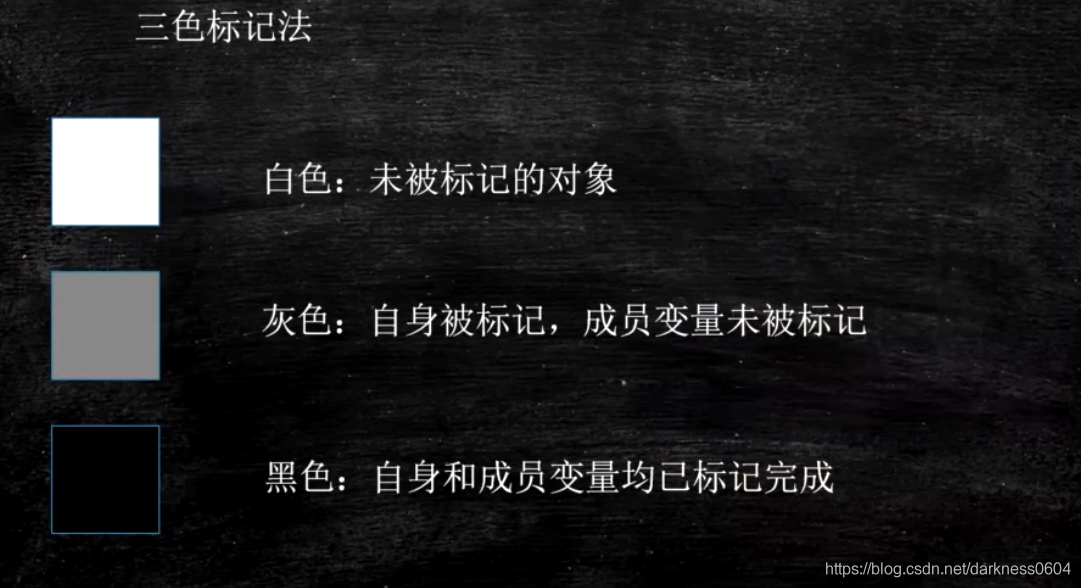
CMS和G1的核心都是并发标记，算法是一样的。

**难点**



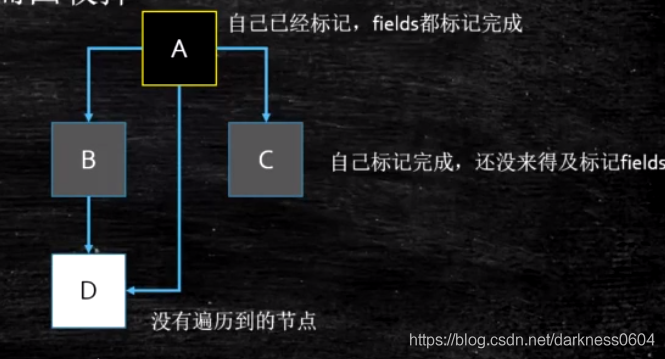
因为是并发， 存在垃圾标记的过程中，新垃圾的产生或者已有垃圾的无效化（不是垃圾了）的问题。

**核心算法 三色标记法**

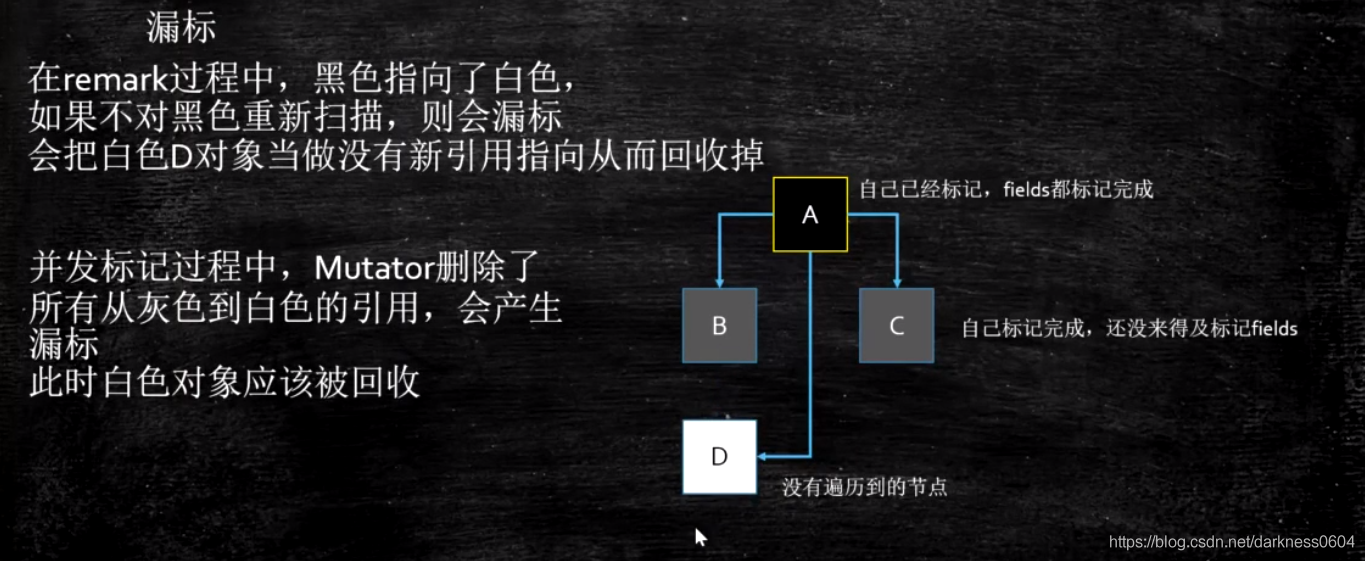




**漏标问题**



黑色对象A指向了白色对象D



与此同时，灰色对象B取消了对白色对象D的引用指向。

此时，D被视为垃圾对象，但实际被A引用，可A已经被标记成黑色了，不会二次标记了，也就造成了D的漏标。

**如何解决漏标问题**

发生漏标是在两个基础上：1、A对D的新增引用 。 2、B对D的引用遗弃。

只要打破其中一个就可以不漏标。

1、关注引用的增加，当A指向D的时候，将这条引用的src，也就是A置为灰色，下次扫描（remark）的时候就可以通过A的重新扫描标记到D了了，就不会发生漏标了，CMS就是使用的这种方法。

2、关注引用的删除，每当发生删除时，会把被删除的这个引用推到GC的堆栈里，当下次并发标记扫描的时候，

会从这个堆栈里检测一下堆栈里的这些引用通过查看各自的rememberedset中是否有新的引用重新指向了自身，

如果有的话，将不把这个对象当作是垃圾对象。G1就是使用的这种方式

一、**CMS**采用的是增量更新(Incremental update)(post-write barrier)，致力于第一个条件的打破，记录所有新增的引用关系：

只要在写屏障(post-write barrier)里发现要有一个白对象的引用被赋值到一个黑对象 的字段里，那就把这个黑对象变成灰色的，即插入的时候记录下来。

哪怕删除所有灰对象到该白对象的引用，remark 阶段重新以这些引用关系的src为根再扫描一遍就不会漏标了。

（Incremental update的write barrier会拦截所有新插入的引用关系，并且按需要记录新的引用关系。常见实现会判断例如：a.foo = b 那么a是否是黑色对象而b是否是白色对象。也有完全不做过滤的变种。CMS具体使用的write barrier是无条件的，跟HotSpot VM除G1外的其它GC的card marking基本一样。）

二、**G1**中，使用的是STAB(snapshot-at-the-beginning)(pre-write barrier)的方式，致力于第二个条件的打破。

采用最保守的做法，把变更前的引用对象记录在**satb\_mark\_queue**中下来，当作是存活对象，让其活过这一周期。当下一次并发标记的时候，会依次处理**satb\_mark\_queue**中的对象，确保这部分对象在本轮GC是存活的。

[NextTAMS,top]指针之间的对象是并发标记期间新增对象，也在这一个周期里隐式存活。

因此 G1 的 SATB 会产生更多的浮动垃圾。

但是换来的好处就是：不需要像 CMS 那样 remark，再走一遍 root trace 这种相当耗时的流程。

**CMS**的incremental update设计使得它在remark阶段必须重新扫描所有线程栈和整个young gen作为root。

**G1**的SATB设计在remark阶段则只需要扫描标记剩下的satb\_mark\_queue。

**存放变更前的引用对象的地方之satb\_mark\_queue**

satb\_mark\_queue就是STAB解决方案中，存放那些在并发标记过程中发生了变动的对象的地方。

以SATB write barrier为例，每个Java线程有一个独立的、定长的SATBMarkQueue，mutator在barrier里只把old\_value压入该队列中。

一个队列满了之后，它就会被加到全局的SATB队列集合SATBMarkQueueSet里等待处理，然后给对应的Java线程换一个新的、干净的队列继续执行下去。

并发标记（concurrent marker）会定期检查全局SATB队列集合的大小。当全局集合中队列数量超过一定阈值后，concurrent marker就会处理集合里的所有队列：把队列里记录的每个oop都标记上，并将其引用字段压到标记栈（marking stack）上等后面做进一步标记。

**G1为什么使用SATB的方式解决漏标问题？**

分析一下，当灰色B>白色D引用消失时，引用D会被push到堆栈（satb\_mark\_queue），下次并发标记扫描时是可以拿到这个引用D的，由于有RSet的存在，不需要扫描整个堆去查找指向白色D的引用，效率比较高

SATB配合RSet,浑然天成，使得G1变得很快的原因之一。

另外，如果使用第一种，将A置灰的话，其实会有一个重新标记一遍之前A标过的所有节点，比较耗费时间，作为追求效率的G1来说，自然不用它。

**顺便一提-更新Rset之dirty\_card\_queue**

我们知道，Rset存放了其他region对此region对象的引用，那这个是什么时候放的呢？

这个其实是借助了barrier：每次向引用类型字段赋值都要经过很多步骤来更新RSet的话开销实在太大，而实际G1的实现是类似：

跟SATB marking queue类似，每个Java线程有一个dirty card queue，然后有一个全局的DirtyCardQueueSe

实际更新RSet的动作就交由多个ConcurrentG1RefineThread并发完成。

每当全局队列集合超过一定阈值后，ConcurrentG1RefineThread就会取出若干个队列，遍历每个队列记录的card并将card加到对应的region的RSet里去。

参考文章：

关于incremental update与SATB的一点理解

<https://hllvm-group.iteye.com/group/topic/44529>

G1垃圾收集器之SATB

<https://blog.csdn.net/f191501223/article/details/84726719>

JVM-G1算法和数据结构那些事

<https://segmentfault.com/a/1190000021394215?utm_source=tag-newest>

R大详解G1算法原理

<https://hllvm-group.iteye.com/group/topic/44381#post-272188>