当面试扯到jvm这一部分的时候，面试官大概率会问你**jvm怎么判断哪些对象应该回收呢?**

这种经典的面试题当然难不住你。

你会脱口而出**引用计数算法**和**可达性分析算法**。

然后你就停下来了吗？难道你不知道你回答了一句话之后，面试官肯定会接着问你能详细说明一下吗？所以，不要停。主动点，面试的时候主动点。你要抓住面试官把话语权交给你的宝贵机会，接着说啊，你得支棱起来

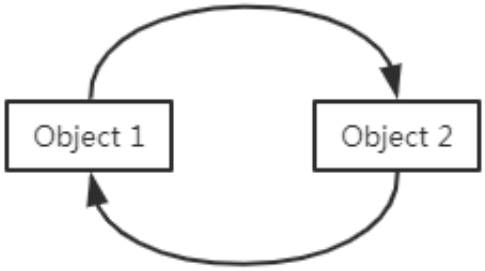
因为**引用计数法的算法**是这样的：在对象中添加一个引用计数器，每当一个地方引用它时，计数器就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可能再被使用的。

但是这样的算法有个问题，是什么呢?

不经意间来一波自问自答。让面试官听的一愣一愣的。

就是不能解决循环依赖的问题。

并拿着自己准备的纸和笔快速的画出下面这样的图：



Object 1和Object 2其实都可以被回收，但是它们之间还有相互引用，所以它们各自的计数器为1，则还是不会被回收。

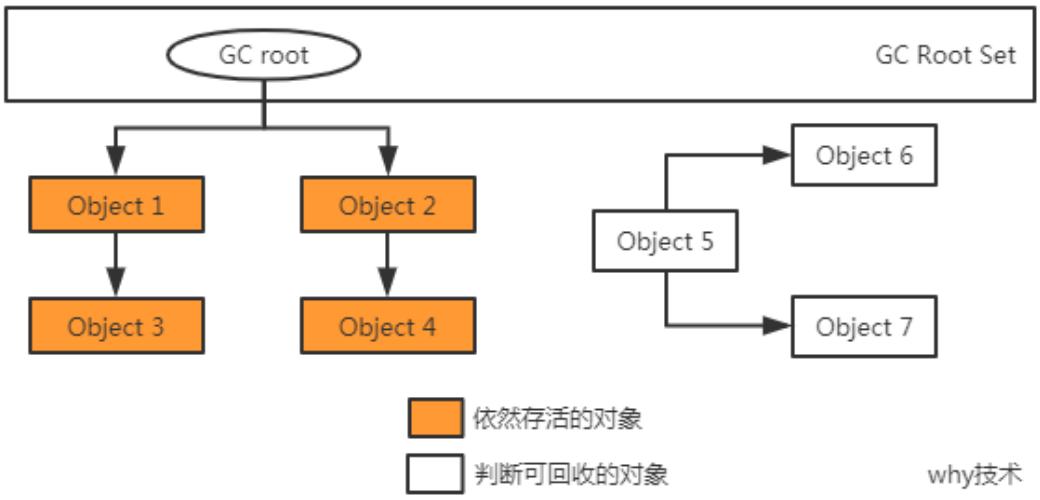
所以，Java虚拟机没有采用引用计数法。它采用的是可达性分析算

法。

可达性分析算法的思路就是通过一系列的“GC Roots”，也就是根对象作为起始节点集合，从根节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为引用链，如果某个对象到GC Roots间没有任何引用链相连。

用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的。所以此对象就是可以被回收的对象。

说这句话的时候再次，快速的纸上画出下面的图：



好了，到这里就可以把话语权交给面试官了。因为到这里，他接下来可以问的点有很多，你不知道他会问什么，比如：

你刚刚谈到了根节点，那你知道哪些对象可以作为根对象吗？

你刚刚谈到了引用，那你知道java里面有哪几种引用吗？

你刚刚谈到了可达性分析算法，那如果在该算法中被判定不可达对象，是不是一定会被回收呢？

谈谈你熟悉的垃圾回收器和他们的工作过程？

.....

上面的这些问题都太常规了，任何一份面经里面都会有这样的几个问题。

而本文要解决的是下面这个稍微不那么常见，但是你答题的过程中一定会提到的点“并发标记”、“浮动垃圾”。

CMS和G1都是有一个并发标记的过程，并发标记要解决什么问题？带来了什么问题？怎么解决这些问题呢？

并发标记要解决什么问题？

刚刚我们谈到的可达性分析算法是需要一个理论上的前提的：**该算法的全过程都需要基于一个能保障一致性的快照中才能够分析，这意味着必须全程冻结用户线程的运行。**

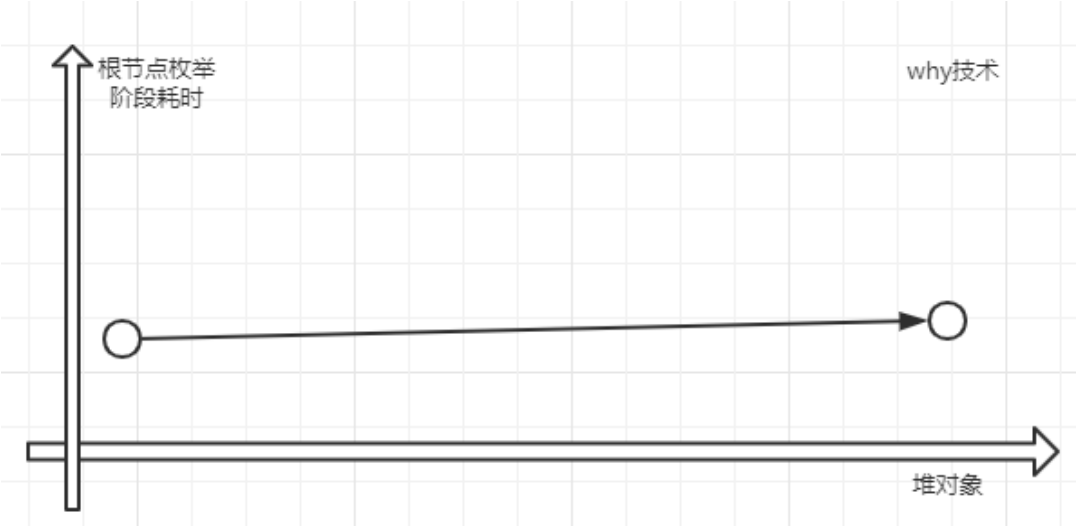
为了不冻结用户线程的运行，那我们就需要让垃圾回收线程和用户线程同时运行。

所有我们来个反证法，先假设不并发标记，即只有垃圾回收线程在运

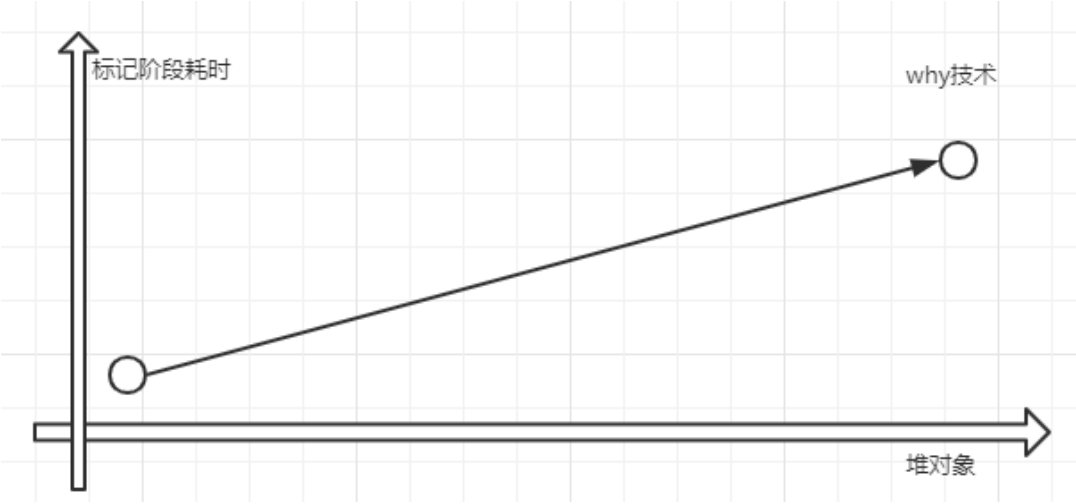
行的流程是怎样的：

第一步是需要找到根节点，也就是我们常说的**根节点枚举。**

而在这个过程中，由于GC Roots是远远少于整个java堆中的全部对象的，而且在OopMap此类优化技巧的加持下，它带来的停顿时间是非常短暂且相对固定的，**可以理解为不会随着堆里面的对象的增加而增加。**大概就是下面这个图的意思：



但是我们做完根节点枚举，只是做完了第一步。**接下来，我们需要从GC Roots往下继续遍历对象图，进行"标记"过程。而这一步的停顿时间必然是随着java堆中的对象增加而增加的。**大概就是下面这个图的意思：



这个逻辑不复杂：堆约大，存储的对象越多，对象图结构越复杂，要

标记更多对象，所以产生的停顿时间也自然就长了。

所有，经过上面的分析，**我们知道了，根节点的枚举阶段是不太耗时的，也不会随着java堆里面存储的对象增加而增加耗时。而"标记"过程的耗时是会随着java堆里面存储的对象增加而增加的。**

"标记"阶段是所有使用可达性分析算法的垃圾回收器都有的阶段。因此我们可以知道，如果能够削减"标记"过程这部分的停顿时间，那么收益将是可观的。

所以并发标记要解决什么问题呢?

就是要消减这一部分的停顿时间。那就是让垃圾回收器和用户线程同时运行，并发工作。也就是我们说的并发标记的阶段。

并发标记带来了什么问题?

在说带来什么问题之前，我们必须得先搞清楚一个问题：

为什么遍历对象图的时候必须在一个能保障一致性的快照中?

为了说明这个问题，我们就要引入"三色标记"大法了。**注意："三色标记"也是jvm的一个考点哦。**

什么是"三色标记"？《深入理解Java虚拟机(第三版)》中是这样描述的：

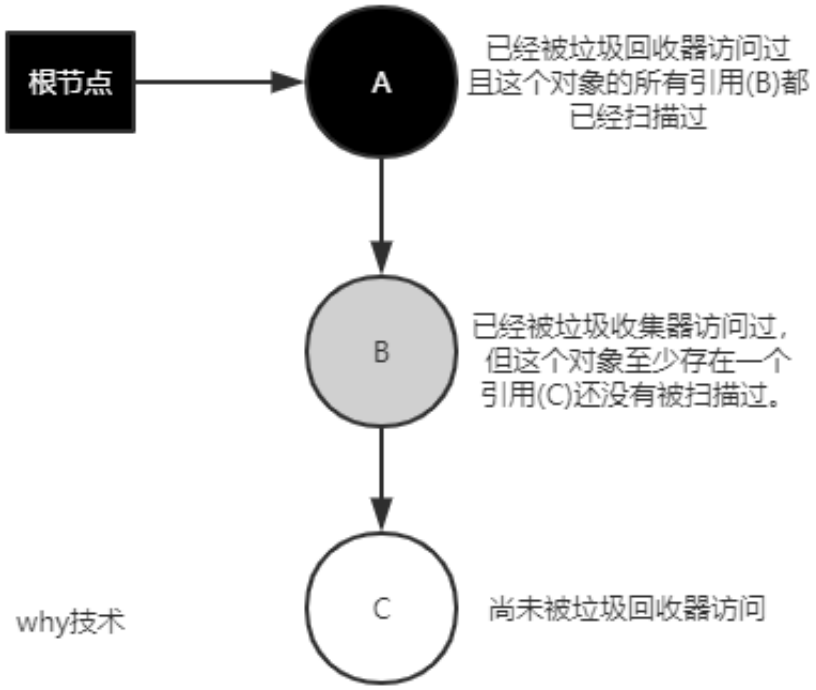
在遍历对象图的过程中，把访问都的对象**按照"是否访问过"这个条件**标记成以下三种颜色：

白色：表示对象尚未被垃圾回收器访问过。显然，在可达性分析刚刚开始的阶段，所有的对象都是白色的，若在分析结束的阶段，仍然是白色的对象，即代表不可达。

黑色：表示对象已经被垃圾回收器访问过，且这个对象的所有引用都已经扫描过。黑色的对象代表已经扫描过，它是安全存活的，如果有其它的对象引用指向了黑色对象，无须重新扫描一遍。黑色对象不可能直接（不经过灰色对象）指向某个白色对象。

灰色：表示对象已经被垃圾回收器访问过，但这个对象至少存在一个引用还没有被扫描过。

读完上面描述，再品一品下面的图：



可以看到，灰色对象是黑色对象与白色对象之间的中间态。当标记过程结束后，只会有黑色和白色的对象，而白色的对象就是需要被回收的对象。

在可达性分析的扫描过程中，如果只有垃圾回收线程在工作，那肯定不会有任何问题。

但是垃圾回收器和用户线程同时运行呢？这个时候就有点意思了。

垃圾回收器在对象图上面标记颜色，而同时用户线程在修改引用关系，引用关系修改了，那么对象图就变化了，这样就有可能出现两种后果：

一种是把原本消亡的对象错误的标记为存活，这不是好事，但是其实

是可以容忍的，只不过产生了一点逃过本次回收的浮动垃圾而已，下次清理就可以。

一种是把原本存活的对象错误的标记为已消亡，这就是非常严重的后果了，一个程序还需要使用的对象被回收了，那程序肯定会因此发生错误。

当面试官问你：**为什么会产生浮动垃圾**的时候，你就可以用上面的话来回答。

但是大概率情况下面试官应该更加关心第二种情况。

他可能会问：**你刚刚说的第二种情况，"把原本存活的对象错误的标记为已消亡"能具体的说明一下吗？怎么消亡的？垃圾回收器是怎么解决这个问题的?**

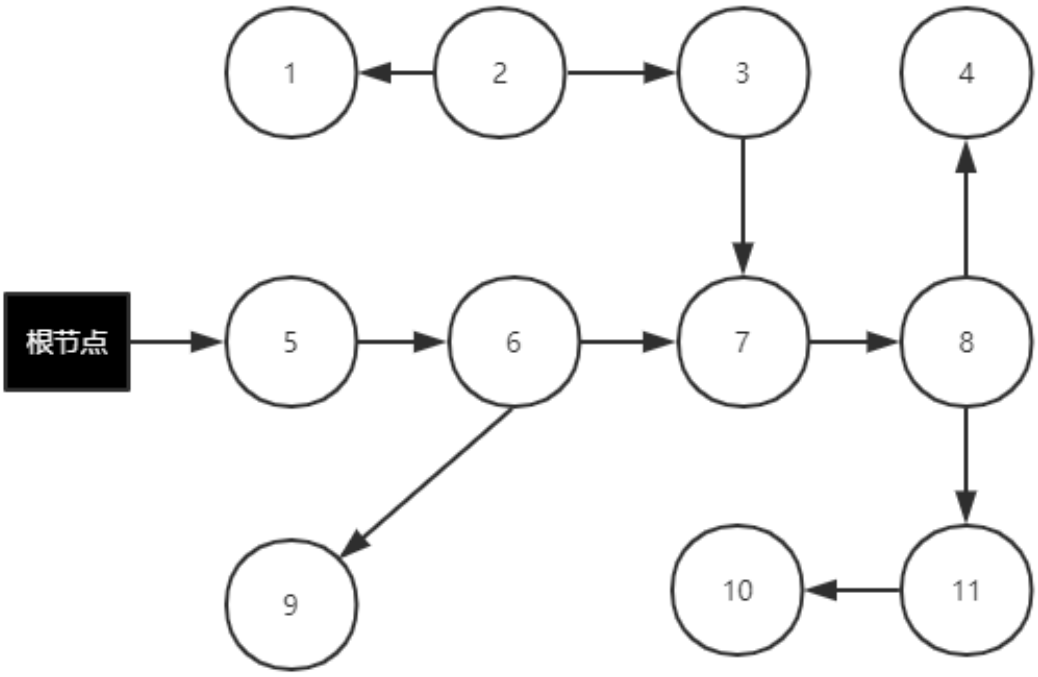
所以接下来，我们主要分析一下并发标记的过程中"对象消失"的问题。具体"对象"是怎么没了的。

这里借助《深入理解Java虚拟机(第三版)》的示例，但是第三版的示例的描述写的不是特别容易理解，我就尽我所能的描述的清楚一些，下面会结合动图，分析标记的三种情况：

正常标记

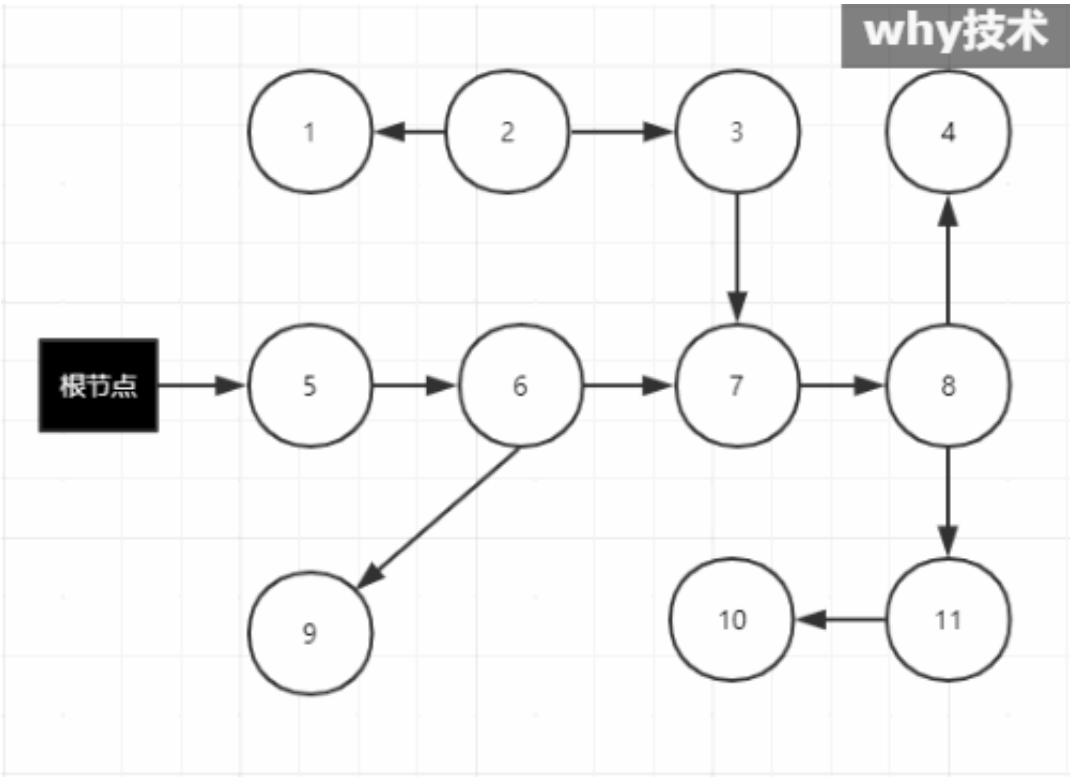
我们先看一下一次正常的标记过程：

首先是初始状态，很简单，只有GC Roots是黑色的。同时需要注意下面的图片的箭头方向，代表的是有向的，比如其中的一条引用链是：**根节点->5->6->7->8->11->10**

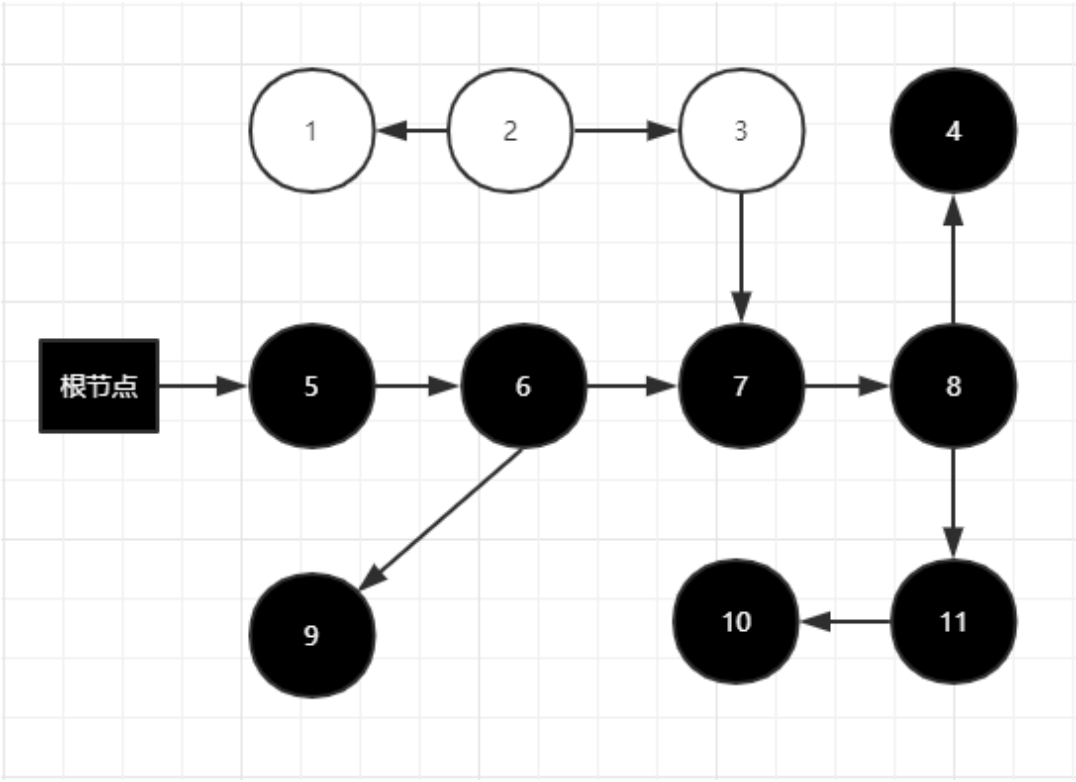


在扫描的过程中，变化是这样的：

内心OS：为了做下面的这些动图、为了把动图里面的每张图截的大小一个像素都不差，鬼知道我做的多辛苦，做瞎我的钛合金狗眼。



你看上面的动图，灰色对象始终是介于黑色和白色之间的。当扫描顺利完成后，对象图就变成了这个样子：



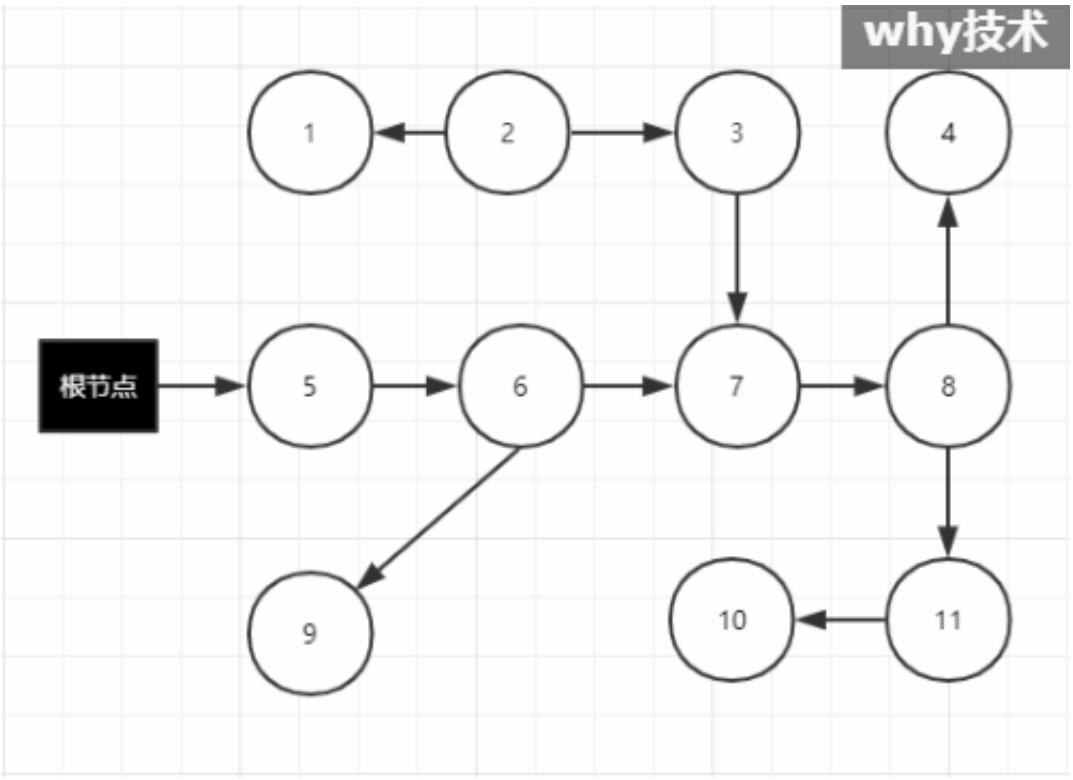
此时，黑色对象是存活的对象，白色对象是消亡了，可以回收的对象。

记住，上面演示的是一切都是那么美好的正常情况。

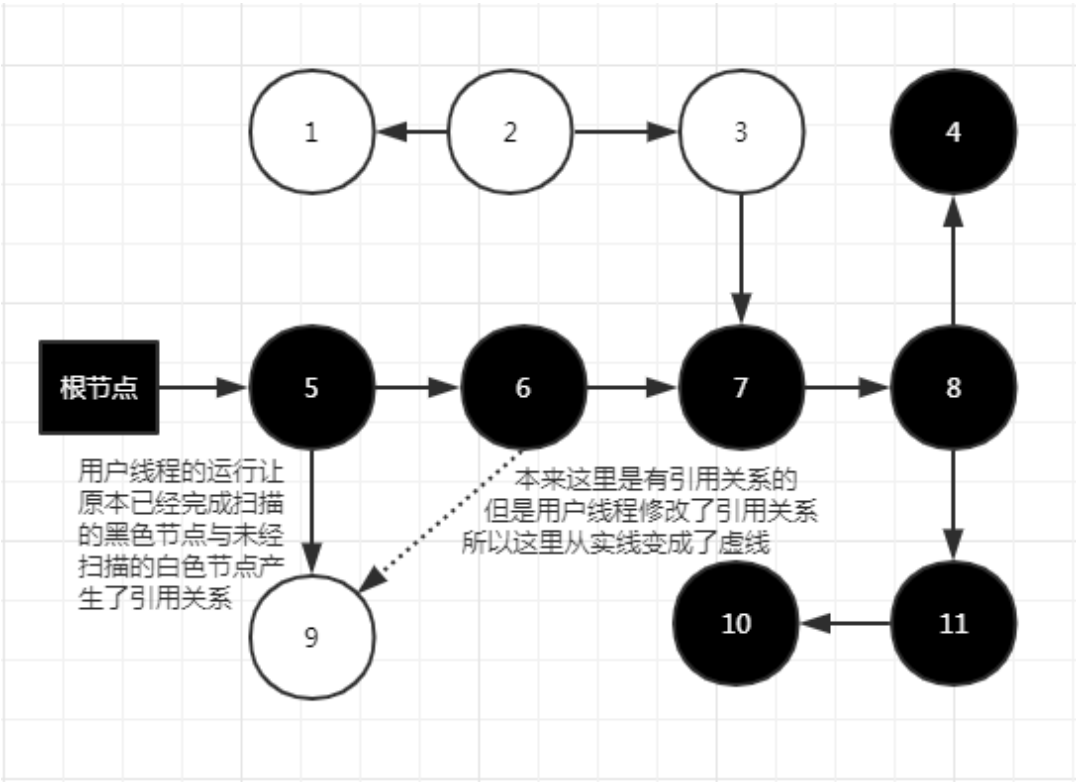
对象消失的情况一

接下来，我们看看对象消失的情况：

如果用户线程在标记的时候，修改了引用关系，就会出现下面的情况：



当扫描完成后，对象图就变成了这个样子：

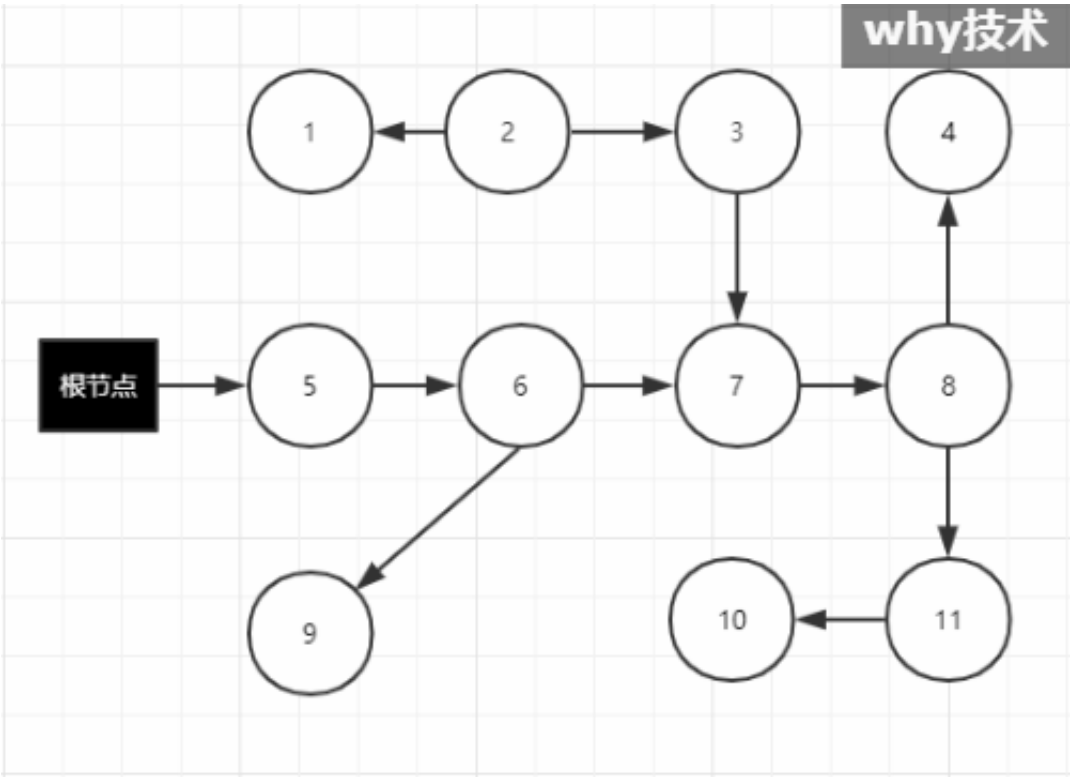


这时，我们和之前分析的正常扫描结束的对象图对比，就能清楚的看到，扫描完成后，原本还在被对象5引用的对象9，由于是白色对象，所以根据三色标记原则，对象9会被当成垃圾回收。

这样就出现了对象消失的情况。

对象消息的情况二

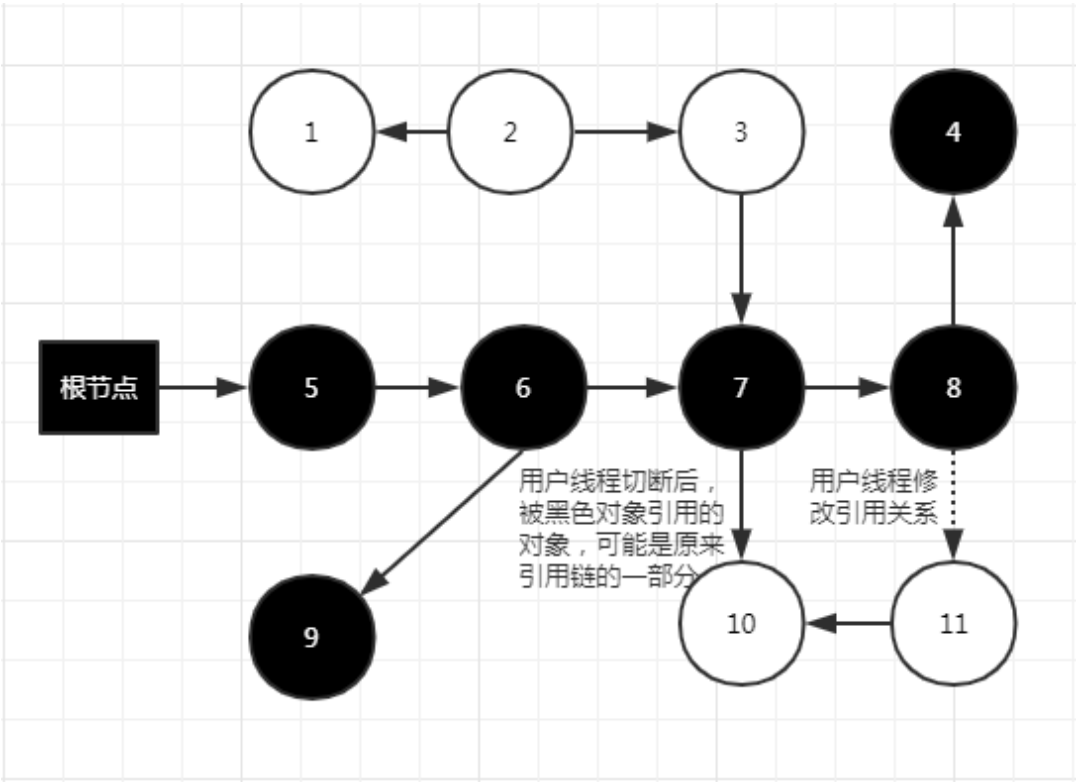
下面再给各位看看另外一种"对象消失"的现象：



上面演示的是用户线程切断引用后重新被黑色对象引用的对象就是原来引用链的一部分。

对象7和对象10本来就是原引用链（根节点->5->6->7->8->11->10）的一部分。修改后的引用链变成了（根节点->5->6->7->10）。

当扫描完成后，对象图就变成了这个样子：



由于黑色对象不会重新扫描，这将导致扫描结束后对象10和对象11都会回收了。他们都是被修改之前的原来的引用链的一部分。

所以，回到最开始的疑问：并发标记带来了什么问题?

经过我们上面三种情况(一种正常情况，两种"对象丢失"的情况)的动图分析，和扫描完成后的最终对象图进行分析对比，我们知道了，**并发标记除了会产生浮动垃圾，还会出现"对象消失"的问题。**

怎么解决"对象消失"问题呢?

有一个大佬叫Wilson，他在1994年在理论上证明了，**当且仅当以下两个条件同时满足时**，会产生"对象消失"的问题，原来应该是黑色的对象被误标为了白色：

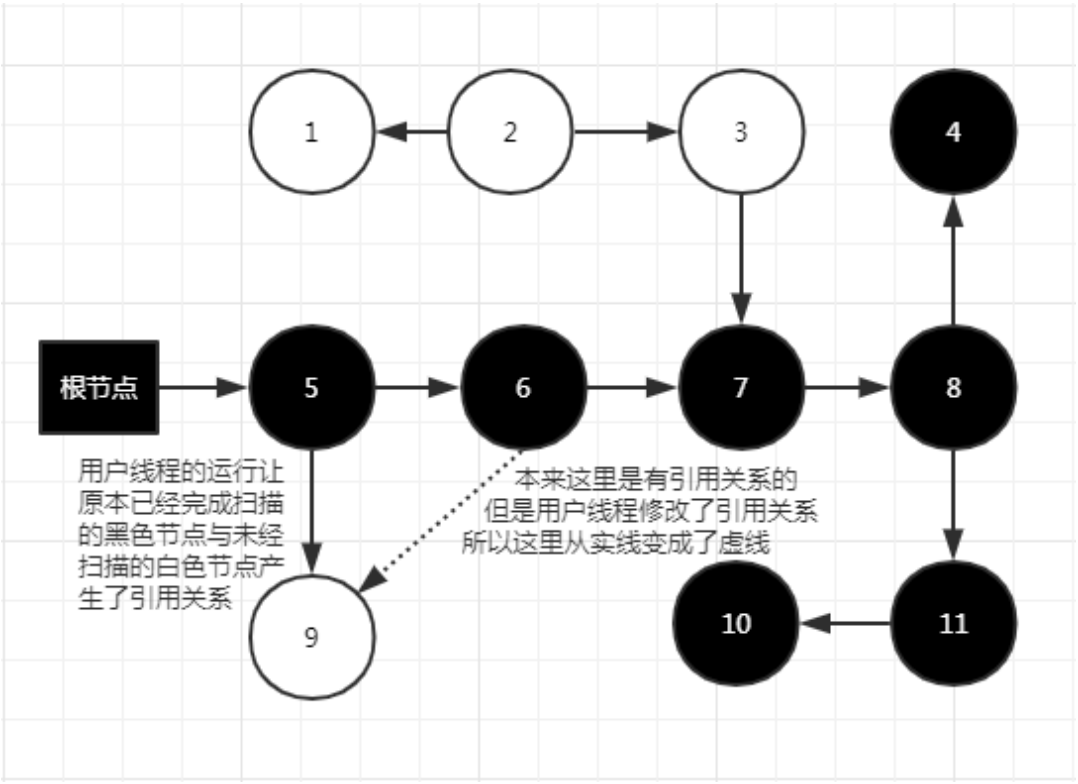
条件一：赋值器插入了一条或者多条从黑色对象到白色对象的新引用。

条件二：赋值器删除了全部从灰色对象到该白色对象的直接或间接引用。

你在结合我们上面出现过的图捋一捋上面的这两个条件，是不是当且仅当的关系：

黑色对象5到白色对象9之间的引用是新建的，对应条件一。

黑色对象6到白色对象9之间的引用被删除了，对应条件二。



由于两个条件之间是当且仅当的关系。所以，我们要解决并发标记时对象消失的问题，只需要破坏两个条件中的任意一个就行。

于是产生了两种解决方案：**增量更新（Incremental Update）和原始快照（Snapshot At The Beginning，SATB）。**

在HotSpot虚拟机中，**CMS是基于增量更新来做并发标记的，G1则采用的是原始快照的方式。**

什么是增量更新呢?

增量更新要破坏的是第一个条件（赋值器插入了一条或者多条从黑色对象到白色对象的新引用），当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。

可以简化的理解为：黑色对象一旦插入了指向白色对象的引用之后，它就变回了灰色对象。

下面的图就是一次并发扫描结束之后，记录了黑色对象5新指向了白色对象9：

这样对象9又被扫描成为了黑色。也就不会被回收，所以不会出现对象

消失的情况。

什么是原始快照呢?

原始快照要破坏的是第二个条件（赋值器删除了全部从灰色对象到该白色对象的直接或间接引用），当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次。

这个可以简化理解为：无论引用关系删除与否，都会按照刚刚开始扫描那一刻的对象图快照开进行搜索。

需要注意的是，上面的介绍中无论是对引用关系记录的插入还是删除，**虚拟机的记录操作都是通过写屏障实现的**。写屏障也是一个重要的知识点，但是不是本文重点，就不进行详细介绍了。

只是补充两点：

1.这里的写屏障和我们常说的为了解决并发乱序执行问题的"内存屏障"不是一码事，需要区分开来。

2.写屏障可以看作虚拟机层面对"引用类型字段赋值"这个动作的AOP切面，在引用对象赋值时会产生一个环形通知，供程序执行额外的动作，也就是说赋值的前后都在写屏障的覆盖范畴内。在赋值前的部分的写屏障叫做写前屏障(Pre-Write Barrier)，在赋值后的则叫作写后屏障(Post-Write Barrier)。

所以，经过简单的推导我们可以知道：

增量更新用的是写后屏障(Post-Write Barrier)，记录了所有新增的引用关系。

原始快照用的是写前屏障(Pre-Write Barrier)，将所有即将被删除的引用关系的旧引用记录下来。

最后说一句(求关注)

最近有很多读者在找我修改简历、咨询工作的相关事情了，我就知道马上又要开始春招了。

其实我也不是很有资格给你们修改简历，也不是一个技术很牛逼的人，只是把我知道的分享出来了而已，不仅能让我巩固知识，还是倒

逼我进行知识输入，在此之外还能对你有一点点帮助，那就是我文章的全部价值所在。

另外如果你正在经历春招或者社招，有兴趣的可以阅读一下我之前的这篇文章，看看是否有一点点帮助：

《面试了15位来自985/211高校的2020届研究生之后的思考》

才疏学浅，难免会有纰漏，如果你发现了错误的地方，还请你留言给我指出来，我对其加以修改。

如果你觉得文章还不错，你的转发、分享、赞赏、点赞、留言就是对我最大的鼓励。

感谢您的阅读，**我坚持原创**，十分欢迎并感谢您的关注。