

讲堂 深入拆解 Java 虚拟机 文章详情

11 | 垃圾回收（上）

2018-08-15 郑雨迪



11 | 垃圾回收（上）

朗读人：郑雨迪 12'11" | 5.59M

你应该听说过这么一句话：免费的其实是最贵的。

Java 虚拟机的自动内存管理，将原本需要由开发人员手动回收的内存，交给垃圾回收器来自动回收。不过既然是自动机制，肯定没法做到像手动回收那般精准高效 [1]，而且还会带来不少与垃圾回收实现相关的问题。

接下来的两篇，我们会深入探索 Java 虚拟机中的垃圾回收器。今天这一篇，我们来回顾一下垃圾回收的基础知识。

引用计数法与可达性分析

垃圾回收，顾名思义，便是将已经分配出去的，但却不再使用的内存回收回来，以便能够再次分配。在 Java 虚拟机的语境下，垃圾指的是死亡的对象所占据的堆空间。这里便涉及了一个关键的问题：如何辨别一个对象是存是亡？

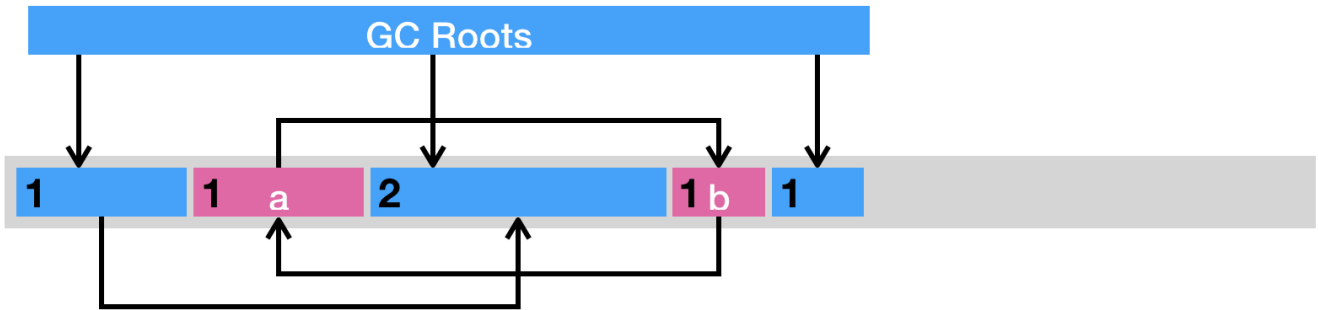
我们先来讲一种古老的辨别方法：引用计数法（reference counting）。它的做法是为每个对象添

加一个引用计数器，用来统计指向该对象的引用个数。一旦某个对象的引用计数器为 0，则说明该对象已经死亡，便可以回收了。

它的具体实现是这样子的：如果有一个引用，被赋值为某一对象，那么将该对象的引用计数器 +1。如果一个指向某一对象的引用，被赋值为其他值，那么将该对象的引用计数器 -1。也就是说，我们需要截获所有的引用更新操作，并且相应地增减目标对象的引用计数器。

除了需要额外的空间来存储计数器，以及繁琐的更新操作，引用计数法还有一个重大的漏洞，那便是无法处理循环引用对象。

举个例子，假设对象 a 与 b 相互引用，除此之外没有其他引用指向 a 或者 b。在这种情况下，a 和 b 实际上已经死了，但由于它们的引用计数器皆不为 0，在引用计数法的心中，这两个对象还活着。因此，这些循环引用对象所占据的空间将不可回收，从而造成了内存泄露。



目前 Java 虚拟机的主流垃圾回收器采取的是可达性分析算法。这个算法的实质在于将一系列 GC Roots 作为初始的存活对象合集（live set），然后从该合集出发，探索所有能够被该集合引用到的对象，并将其加入到该集合中，这个过程我们称之为标记（mark）。最终，未被探索到的对象便是死亡的，是可以回收的。

那么什么是 GC Roots 呢？我们可以暂时理解为由堆外指向堆内的引用，一般而言，GC Roots 包括（但不限于）如下几种：

- 1. Java 方法栈帧中的局部变量；
- 2. 已加载类的静态变量；
- 3. JNI handles；
- 4. 已启动且未停止的 Java 线程。

可达性分析可以解决引用计数法所不能解决的循环引用问题。举例来说，即便对象 a 和 b 相互引用，只要从 GC Roots 出发无法到达 a 或者 b，那么可达性分析便不会将它们加入存活对象合集之中。

虽然可达性分析的算法本身很简明，但是在实践中还是有不少其他问题需要解决的。

比如说，在多线程环境下，其他线程可能会更新已经访问过的对象中的引用，从而造成误报（将引

用设置为 null) 或者漏报 (将引用设置为未被访问过的对象)。

误报并没有什么伤害, Java 虚拟机至多损失了部分垃圾回收的机会。漏报则比较麻烦, 因为垃圾回收器可能回收事实上仍被引用的对象内存。一旦从原引用访问已经被回收了的对象, 则很有可能会直接导致 Java 虚拟机崩溃。

Stop-the-world 以及安全点

怎么解决这个问题呢? 在 Java 虚拟机里, 传统的垃圾回收算法采用的是一种简单粗暴的方式, 那便是 Stop-the-world, 停止其他非垃圾回收线程的工作, 直到完成垃圾回收。这也就造成了垃圾回收所谓的暂停时间 (GC pause)。

Java 虚拟机中的 Stop-the-world 是通过安全点 (safepoint) 机制来实现的。当 Java 虚拟机收到 Stop-the-world 请求, 它便会等待所有的线程都到达安全点, 才允许请求 Stop-the-world 的线程进行独占的工作。

这篇博客 [2] 还提到了一种比较另类的解释: 安全词。一旦垃圾回收线程喊出了安全词, 其他非垃圾回收线程便会一一停下。

当然, 安全点的初始目的并不是让其他线程停下, 而是找到一个稳定的执行状态。在这个执行状态下, Java 虚拟机的堆栈不会发生变化。这么一来, 垃圾回收器便能够“安全”地执行可达性分析。

举个例子, 当 Java 程序通过 JNI 执行本地代码时, 如果这段代码不访问 Java 对象、调用 Java 方法或者返回至原 Java 方法, 那么 Java 虚拟机的堆栈不会发生改变, 也就代表着这段本地代码可以作为同一个安全点。

只要不离开这个安全点, Java 虚拟机便能够在垃圾回收的同时, 继续运行这段本地代码。

由于本地代码需要通过 JNI 的 API 来完成上述三个操作, 因此 Java 虚拟机仅需在 API 的入口处进行安全点检测 (safepoint poll), 测试是否有其他线程请求停留在安全点里, 便可以在必要的时候挂起当前线程。

除了执行 JNI 本地代码外, Java 线程还有其他几种状态: 解释执行字节码、执行即时编译器生成的机器码和线程阻塞。阻塞的线程由于处于 Java 虚拟机线程调度器的掌控之下, 因此属于安全点。

其他几种状态则是运行状态, 需要虚拟机保证在可预见的时间内进入安全点。否则, 垃圾回收线程可能长期处于等待所有线程进入安全点的状态, 从而变相地提高了垃圾回收的暂停时间。

对于解释执行来说, 字节码与字节码之间皆可作为安全点。Java 虚拟机采取的做法是, 当有安全点请求时, 执行一条字节码便进行一次安全点检测。

执行即时编译器生成的机器码则比较复杂。由于这些代码直接运行在底层硬件之上, 不受 Java 虚拟机掌控, 因此在生成机器码时, 即时编译器需要插入安全点检测, 以避免机器码长时间没有安全点

检测的情况。HotSpot 虚拟机的做法便是在生成代码的方法出口以及非计数循环的循环回边（back-edge）处插入安全点检测。

那么为什么不在每一条机器码或者每一个机器码基本块处插入安全点检测呢？原因主要有两个。

第一，安全点检测本身也有一定的开销。不过 HotSpot 虚拟机已经将机器码中安全点检测简化为一个内存访问操作。在有安全点请求的情况下，Java 虚拟机会将安全点检测访问的内存所在的页设置为不可读，并且定义一个 segfault 处理器，来截获因访问该不可读内存而触发 segfault 的线程，并将它们挂起。

第二，即时编译器生成的机器码打乱了原本栈帧上的对象分布状况。在进入安全点时，机器码还需提供一些额外的信息，来表明哪些寄存器，或者当前栈帧上的哪些内存空间存放着指向对象的引用，以便垃圾回收器能够枚举 GC Roots。

由于这些信息需要不少空间来存储，因此即时编译器会尽量避免过多的安全点检测。

不过，不同的即时编译器插入安全点检测的位置也可能不同。以 Graal 为例，除了上述位置外，它还会在计数循环的循环回边处插入安全点检测。其他的虚拟机也可能选取方法入口而非方法出口来插入安全点检测。

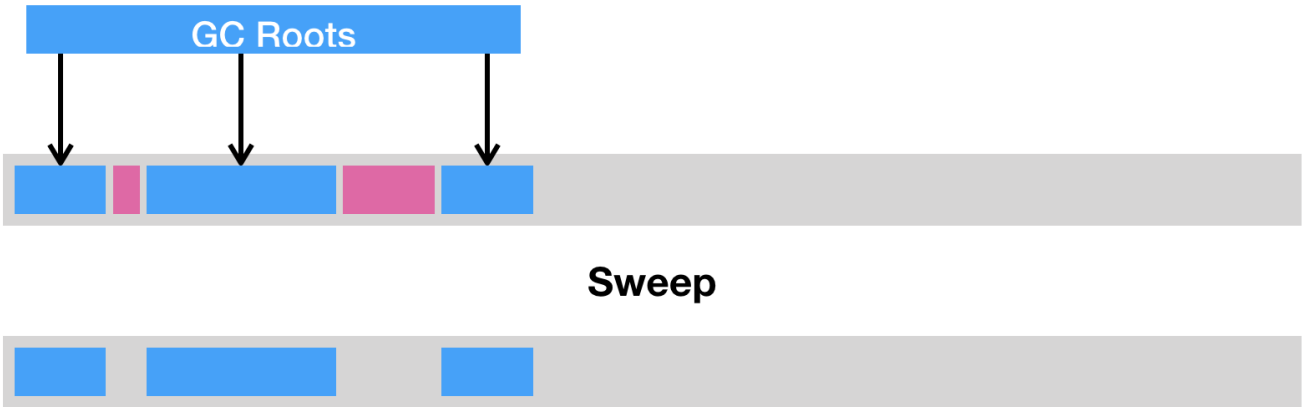
不管如何，其目的都是在可接受的性能开销以及内存开销之内，避免机器码长时间不进入安全点的情况，间接地减少垃圾回收的暂停时间。

除了垃圾回收之外，Java 虚拟机其他一些对堆栈内容的一致性有要求的操作也会用到安全点这一机制。我会在涉及的时候再进行具体的讲解。

垃圾回收的三种方式

当标记完所有的存活对象时，我们便可以进行死亡对象的回收工作了。主流的基础回收方式可分为三种。

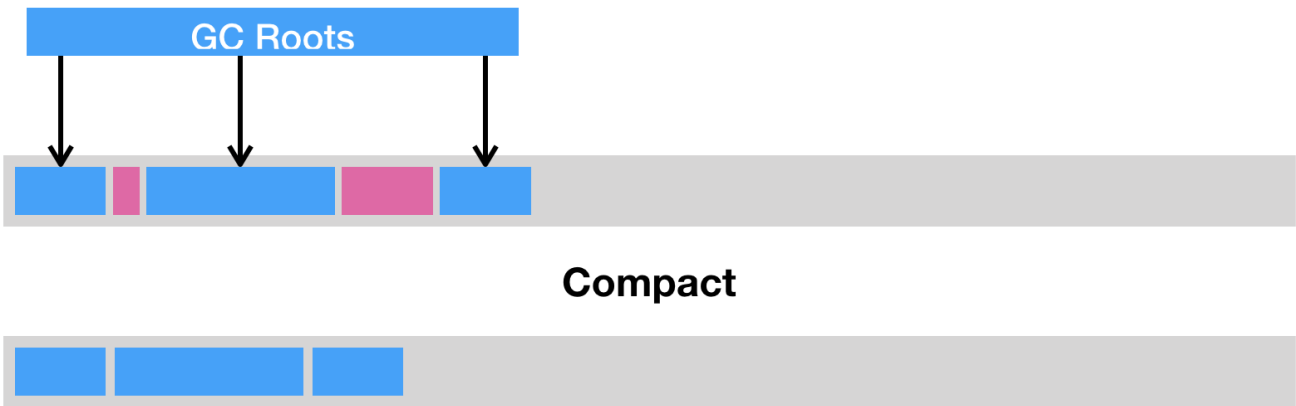
第一种是清除（sweep），即把死亡对象所占据的内存标记为空闲内存，并记录在一个空闲列表（free list）之中。当需要新建对象时，内存管理模块便会从该空闲列表中寻找空闲内存，并划分给新建的对象。



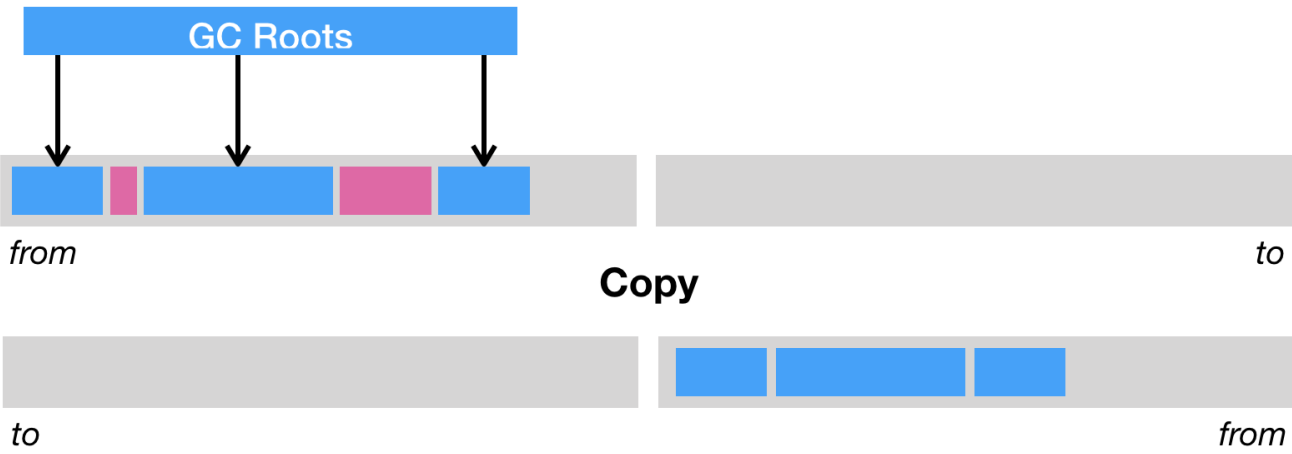
清除这种回收方式的原理及其简单，但是有两个缺点。一是会造成内存碎片。由于 Java 虚拟机的堆中对象必须是连续分布的，因此可能出现总空闲内存足够，但是无法分配的极端情况。

另一个则是分配效率较低。如果是一块连续的内存空间，那么我们可以通过指针加法（pointer bumping）来做分配。而对于空闲列表，Java 虚拟机则需要逐个访问列表中的项，来查找能够放入新建对象的空闲内存。

第二种是压缩（compact），即把存活的对象聚集到内存区域的起始位置，从而留下一段连续的内存空间。这种做法能够解决内存碎片化的问题，但代价是压缩算法的性能开销。



第三种则是复制（copy），即把内存区域分为两等分，分别用两个指针 from 和 to 来维护，并且只是用 from 指针指向的内存区域来分配内存。当发生垃圾回收时，便把存活的对象复制到 to 指针指向的内存区域中，并且交换 from 指针和 to 指针的内容。复制这种回收方式同样能够解决内存碎片化的问题，但是它的缺点也极其明显，即堆空间的使用效率极其低下。



当然，现代的垃圾回收器往往会综合上述几种回收方式，综合它们优点的同时规避它们的缺点。在下一篇中我们会详细介绍 Java 虚拟机中垃圾回收算法的具体实现。

总结与实践

今天我介绍了垃圾回收的一些基础知识。

Java 虚拟机中的垃圾回收器采用可达性分析来探索所有存活的对象。它从一系列 GC Roots 出发，边标记边探索所有被引用的对象。

为了防止在标记过程中堆栈的状态发生改变，Java 虚拟机采取安全点机制来实现 Stop-the-world 操作，暂停其他非垃圾回收线程。

回收死亡对象的内存共有三种方式，分别为：会造成内存碎片的清除、性能开销较大的压缩、以及堆使用效率较低的复制。

今天的实践环节，你可以体验一下无安全点检测的计数循环带来的长暂停。你可以分别测单独跑 foo 方法或者 bar 方法的时间，然后与合起来跑的时间比较一下。

```
// time java SafepointTest
/ 你还可以使用如下几个选项
// -XX:+PrintGC
// -XX:+PrintGCApplicationStoppedTime
// -XX:+PrintSafepointStatistics
// -XX:+UseCountedLoopSafepoints

public class SafepointTest {
    static double sum = 0;

    public static void foo() {
```



```
    for (int i = 0; i < 0x77777777; i++) {  
        sum += Math.sqrt(i);  
    }  
}  
  
public static void bar() {  
    for (int i = 0; i < 50_000_000; i++) {  
        new Object().hashCode();  
    }  
}  
  
public static void main(String[] args) {  
    new Thread(SafepointTest::foo).start();  
    new Thread(SafepointTest::bar).start();  
}  
}
```

[1] <https://media.giphy.com/media/EZ8QO0myvsSk/giphy.gif>

[2] <http://psy-lob-saw.blogspot.com/2015/12/safepoints.html>



版权归极客邦科技所有，未经许可不得转载

精选留言



非常感谢，此篇可用通俗易懂来形容，其他同学问的问题也很棒！

小结：

1:垃圾回收-核心工作就是回收垃圾，哪关键点回来了。什么是垃圾？这个垃圾需要分类嘛？怎么定位垃圾？怎么回收垃圾？回收垃圾的方法都有哪些？他们都有什么优缺点？另外，就是我们为什么要学习垃圾回收？

2:站在JVM的视角来看

垃圾-就是无用对象所占用的堆内存空间

貌似不需要垃圾分类，识别垃圾并回收就行

定位垃圾，是垃圾回收的关键点

晚安💎💎，明天继续写

2018-08-16



茶底

14

老师下一期能讲一下g1算法吗。讲深一点💎💎

2018-08-15



黑崽

1

第二，即时编译器生成的机器码打乱了原本栈帧上的对象分布状况。没明白这个原因。第一个原因中解释，只要去访问一个内存地址就可以知道是不是要暂停了，那我只有判断完暂停以后再去恢复寄存器中状态不就可以了？反正只有一次，这个打乱不打乱有什么区别呢？

2018-08-19



godtrue

1

疑问

1:JVM的stop-the-world机制非常不友好，有哪些解决之道？原理是什么？

2:压测时出现频繁的gc容易理解，但是有时出现毛刺是因为什么呢？

3:fullgc有卡顿，对性能很不利，怎么避免呢？

2018-08-17



godtrue

1

非常感谢，此篇可用通俗易懂来形容，其他同学问的问题也很棒！

小结：

1:垃圾回收-核心工作就是回收垃圾，哪关键点回来了。什么是垃圾？这个垃圾需要分类嘛？怎么定位垃圾？怎么回收垃圾？回收垃圾的方法都有哪些？他们都有什么优缺点？另外，就是我们为什么要学习垃圾回收？

2:站在JVM的视角来看

垃圾-就是无用对象所占用的堆内存空间

垃圾分类-貌似不需要垃圾分类，识别垃圾并回收就行

定位垃圾-是垃圾回收的关键点，无用的对象占用的堆空间即是垃圾，那就需要先定位无用的对象，这里的无用是不再使用的意思，咋判断呢？文中介绍了两种方法，计数法和标记法（祥看原文）核心在于能定位出无用的对象，后出现的方法往往比早出现的更好一点，这里也一样，标记法能解决计数法，解决不了的循环引用不能回收的问题，但是也存在其他的问题，误报和漏报的问题，误报浪费点垃圾回收的机会浪费点空间，漏报在多线程并发工作时可能会死JVM的，所以，比较严重，所以，JVM采用了简单粗暴的stop-the-world的方式来对待，所以，老年代的回收有卡顿的现象

怎么回收垃圾-定位出垃圾，回收就是一个简单的事情了，当然也非常关键，把要回收的堆内存空间标记为可继续使用就行，下次有新对象能在此空间创建就行

回收垃圾的方法-文中介绍了三种，清除、压缩、复制

清除法-简单，但易产生碎片，可能总空间够但分配不了的问题

压缩法-能解决清除法的问题，但是复杂且耗性能

复制法-折衷一些，但是空间利用率低，总之，各有千秋

为什么要学-这个最容易，因为面试需要、装逼需要、升职加薪需要、人类天生好奇、还有免于被鄙视及可以鄙视其他人

2018-08-17



风动静泉

1

有个内存泄露相关的问题想请教一下老师。

基本描述:

一个单线程的程序中，在其run方法中有局部变量（map list等类型），也使用了该线程类ThreadDemo的全局变量（map），线程中会去执行ftp任务（同步）。

问题:

如果ftp任务由于某种原因阻塞了，调用ftp任务的线程ThreadDemo中的变量（局部变量和全局变量）会变成不可达状态吗？还是说此时线程也阻塞，会导致内存泄露？

2018-08-16



彩色的沙漠

1

@正是那朵玫瑰老师有几个不明白的地方，误报和漏报不太明白：

- 1、假设A引用开始指向A1对象：A----->A1,按老师说的误报就是将引用A指向null：A----->null，那么此时A1对象不是没有引用了，不就可以垃圾回收了么，为什么会错过垃圾回收的机会呢？
- 2、漏报，是将A引用指向一个未被访问的对象假设对象为B：A----->B,此时A引用原来指向的对象应该没有引用了吧，为什么会垃圾回收器可能会回收事实上仍被引用的对象呢？

2018-08-15

作者回复

这里指的是，GC已经标记完成，然后其他线程进行修改的情况(也是并发GC所要解决的问题)。

当GC标记完成，还未开始回收时，你更新了其中一个引用，使之指向null，那么原来指向的对象本可以被回收的。

如果指向一个新的对象，这个对象可没有被标记为不能回收，垃圾回收器就直接给回收掉了

老师我也有和@正是那朵玫瑰一样的问题，看了老师的讲解，还是不太明白。GC标记完成，那GC标记的是引用还是具体的堆空间对象。如果标记的具体的堆空间对象，并不会造成GC并发问题，误报和漏报，改变的是引用关系。请老师解答，谢谢！

2018-08-15



王贺

猜一下，作业里面的应该是Math.log(10)

2018-08-15

作者回复

厉害！这都看得出来！是Math.log10(i)，不过这个时间太长了，我新版本的代码是sqrt。然后不知怎么给回滚到这个log10，而且还是乱的

2018-08-15



no13bus

昨天看书正好看到这章节，真的不错

2018-08-15



明天更美好

总算可以听的懂了

2018-08-15



茶底

老师下一期能讲一下g1算法吗。讲深一点💎💎

2018-08-15



code-artist

“最终，未被探索到的对象便是死亡的，是可以回收的。”，已经通过GC_Roots探索到所有的存活对象集。是不是整个堆空间减去存活对象集占用的空间就是空闲堆空间了？这样就不用具体去追究死亡对象的地址空间了？

2018-08-17

1

1

1

1

0

0



WolvesLeader

很是不明白，我的理解有没有stop the word 是和垃圾回收器有关的，看完之后怎么觉得您的意思是，不管什么垃圾回收器都会出现stop the word

2018-08-17



仰望

“举例来说，即便对象 a 和 b 相互引用，只要从 GC Roots 出发无法到达 a 或者 b，”
这个如何判断的？

2018-08-17



魔都浪子

能做一些源码分析吗，比如G1

2018-08-16



维维

请教老师，可达性分析a与b互为引用，这不是死循环了吗，为什么能被回收的，这里不太明白。

2018-08-16



涛哥

老师，特别想了解下G1的原理和机制

2018-08-15



正是那朵玫瑰

感谢老师解答，菜鸟实在还是不太明白[捂脸]，一个对象不可达了，gc才会标记为可以回收，老师所说的：“你更新了其中一个引用”，“你”指的是gc线程还是工作线程，引用指的是哪里的引用，GC都已经标记完成，怎么会有引用被更新？

2018-08-15

作者回复

工作线程。

你已经看出来这里的问题啦。如果工作线程还在继续工作，更新引用，那么就会引发你能想到的这些问题。传统的做法就是接下来讲的Safepoint，禁止工作线程继续更新引用。新的做法是通过一系列算法保证不会出现漏报的情况，从而支持并发回收。

2018-08-15



有铭

代码跑不起来啊，wrMpth.log10是什么

2018-08-15

作者回复

多谢指出，我更新一下

2018-08-15

0

0

0

0

0

0

0



正是那朵玫瑰

老师有几个不明白的地方，误报和漏报不太明白：

- 1、假设A引用开始指向A1对象：A----->A1,按老师说的误报就是将引用A指向null：A----->null，那么此时A1对象不是没有引用了，不就可以垃圾回收了么，为什么会错过垃圾回收的机会呢？
- 2、漏报，是将A引用指向一个未被访问的对象假设对象为B：A----->B,此时A引用原来指向的对象应该没有引用了吧，为什么会垃圾回收器可能会回收事实上仍被引用的对象呢？

2018-08-15

作者回复

这里指的是，GC已经标记完成，然后其他线程进行修改的情况(也是并发GC所要解决的问题)。

当GC标记完成，还未开始回收时，你更新了其中一个引用，使之指向null，那么原来指向的对象本可以被回收的。

如果指向一个新的对象，这个对象可没有被标记为不能回收，垃圾回收器就直接给回收掉了

2018-08-15