对象引用是怎样严重影响垃圾收集器 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/145/2021_2022__E5_AF_B9_E8_B1_A1_E5_BC_95_E7_c104_145534.htm 如果您认为 Java 游戏开发人员是 Java 编程世界的一级方程式赛车手，那么您就会明白为什么他们会如此地重视程序的性能。 游戏开发人员几乎每天都要面对的性能问题，往往超过了一般程序员考虑问题的范围。哪里可以找到这些特殊的开发人员呢？Java 游戏社区就是一个好去处（参见 参考资料）。 虽然在这个站点可能没有很多关于服务器端的应用，但是我们依然可以从中受益，看看这些“ 惜比特如金 ”的游戏开发人员每天所面对的，我们往往能从中得到宝贵的经验。让我们开始游戏吧！对象泄漏 游戏程序员跟其他程序员一样——他们也需要理解 Java 运行时环境的一些微妙之处，比如垃圾收集。垃圾收集可能是使您感到难于理解的较难的概念之一, 因为它并不能总是毫无遗漏地解决 Java 运行时环境中堆管理的问题。似乎有很多类似这样的讨论，它的开头或结尾写着：“ 我的问题是关于垃圾收集 ”。 假如您正面遭遇内存耗尽（out-of-memory）的错误。于是您使用检测工具想要找到问题所在，但这是徒劳的。您很容易想到另外一个比较可信的原因：这是 Java 虚拟机堆管理的问题,而不会认为这是您自己的程序的缘故。但是，正如 Java 游戏社区的资深专家不止一次地解释的，Java 虚拟机并不存在任何被证实的对象泄漏问题。实践证明，垃圾收集器一般能够精确地判断哪些对象可被收集，并且重新收回它们的内存空间给 Java 虚拟机。所以，如果您遇到了内存耗尽的错误，那么这完全可能是由您的程序造成的，也就

是说您的程序中存在着“ 无意识的对象保留（unintentional object retention）”。 内存泄漏与无意识的对象保留 内存泄漏和无意识的对象保留的区别是什么呢？对于用 Java 语言编写的程序来说，确实没有区别。两者都是指在您的程序中存在一些对象引用，但实际上您并不需要引用这些对象。一个典型的例子是向一个集合中加入一些对象以便以后使用它们，但是您却忘了在使用完以后从集合中删除这些对象。因为集合可以无限制地扩大，并且从来不会变小，所以当您在集合中加入了太多的对象（或者是有很多的对象被集合中的元素所引用）时，您就会因为堆的空间被填满而导致内存耗尽的错误。垃圾收集器不能收集这些您认为已经用完的对象，因为对于垃圾收集器来说，应用程序仍然可以通过这个集合在任何时候访问这些对象，所以这些对象是不可能被当作垃圾的。 对于没有垃圾收集的语言来说，例如 C ，内存泄漏和无意识的对象保留是有区别的。C 程序跟 Java 程序一样，可能产生无意识的对象保留。但是 C 程序中存在真正的内存泄漏，即应用程序无法访问一些对象以至于被这些对象使用的内存无法释放且返还给系统。令人欣慰的是，在 Java 程序中，这种内存泄漏是不可能出现的。所以，我们更喜欢用“ 无意识的对象保留 ”来表示这个令 Java 程序员抓破头皮的内存问题。这样，我们就能区别于其他使用没有垃圾收集语言的程序员。 跟踪被保留的对象 那么当发现了无意识的对象保留该怎么办呢？首先，需要确定哪些对象是被无意保留的，并且需要找到究竟是哪些对象在引用它们。然后必须安排好 应该在哪里释放它们。最容易的方法是使用能够对堆产生快照的检测工具来标识这些对象，比较堆的快照中对象的数目，跟

踪这些对象，找到引用这些对象的对象，然后强制进行垃圾收集。有了这样一个检测器，接下来的工作相对而言就比较简单了: 等待直到系统达到一个稳定的状态，这个状态下大多数新产生的对象都是暂时的，符合被收集的条件；这种状态一般在程序所有的初始化工作都完成了之后。 强制进行一次垃圾收集，并且对此时的堆做一份对象快照。 进行任何可以产生无意地保留的对象的操作。 再强制进行一次垃圾收集，然后对系统堆中的对象做第二次对象快照。 比较两次快照，看看哪些对象的被引用数量比第一次快照时增加了。因为您在快照之前强制进行了垃圾收集，那么剩下的对象都应该是被应用程序所引用的对象，并且通过比较两次快照我们可以准确地找出那些被程序保留的、新产生的对象。 根据您对应用程序本身的理解，并且根据对两次快照的比较，判断出哪些对象是被无意保留的。 跟踪这些对象的引用链，找出究竟是哪些对象在引用这些无意地保留的对象，直到您找到了那个根对象，它就是产生问题的根源。 显式地赋空（nulling）变量 一谈到垃圾收集这个主题，总会涉及到这样一个吸引人的讨论，即显式地赋空变量是否有助于程序的性能。赋空变量是指简单地将 null 值显式地赋值给这个变量，相对于让该变量的引用失去其作用域。清单 1. 局部作用域public static String scopingExample(String string) { StringBuffer sb = new StringBuffer(). sb.append("hello ").append(string). sb.append(", nice to see you!"). return sb.toString(). } 当该方法执行时，运行时栈保留了一个对 StringBuffer 对象的引用，这个对象是在程序的第一行产生的。在这个方法的整个执行期间，栈保存的这个对象引用将会防止该对象被当作垃圾。当这个方法执行

完毕，变量 sb 也就失去了它的作用域，相应地运行时栈就会删除对该 StringBuffer 对象的引用。于是不再有对该 StringBuffer 对象的引用，现在它就可以被当作垃圾收集了。栈删除引用的操作就等于在该方法结束时将 null 值赋给变量 sb。 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com