使用技巧：JavaEE性能问题解决手册 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/144/2021_2022__E4_BD_BF_E7_94_A8_E6_8A_80_E5_c104_144792.htm 这篇文章，是PRO JAVA EE 5 Performance Management and Optimization 的一个章节，作者Steven Haines分享了他在调优企业级JAVA应用时所遇到的常见问题。 Java EE(Java企业开发平台)应用程序，无论应用程序服务器如何部署，所面对的一系列问题大致相同。作为一个JAVAEE问题解决专家，我曾经面对过众多的环境同时也写了不少常见问题的观察报告。在这方面，我觉得我很象一个汽车修理工人:你告诉修理工人发动机有声音，他就会询问你一系列的问题，帮你回忆发动机运行的情形。从这些信息中，他寻找到可能引起问题的原因。 众多解决问题的方法思路基本相同，第一天我同要解决问题的客户接触，接触的时候，我会寻找已经出现的问题以及造成的负面的影响。了解应用程序的体系结构和问题表现出的症状，这些工作很够很大程度上提高我解决问题的几率。在这一节，我分享我在这个领域遇过的常见问题和他们的症状。希望这篇文章能成为你JAVAEE的故障检测手册。 内存溢出错误 最常见的折磨着企业级应用程序的错误是让人恐惧的outofmemoryError(内存溢出错误) 这个错误引起下面这些典型的症状: ----应用服务器崩溃 ----性能下降 ----一个看起来好像无法结束的死循环在重复不断的执行垃圾收集，它会导致程序停止运行，并且经常导致应用服务器崩溃 不管症状是什么，如果你想让程序恢复正常运行，你一般都需要重新启动应用服务器。 引发out-of-memory 错误的原因 在你打算解决out-of-memory 错

误之前，首先了解为什么会引发这个错误对你有很大的帮助。如果JVM里运行的程序, 它的内存堆和持久存储区域的都满了，这个时候程序还想创建对象实例的话，垃圾收集器就会启动，试图释放足够的内存来创建这个对象。这个时候如果垃圾收集器没有能力释放出足够的内存，它就会抛出OutOfMemoryError内存溢出错误。 Out-of-memory错误一般是JAVA内存泄漏引起的。回忆上面所讨论的内容，内存泄漏的原因是一个对象虽然不被使用了，但是依然还有对象引用他。当一个对象不再被使用时，但是依然有一个或多个对象引用这个对象，因此垃圾收集器就不会释放它所占据的内存。这块内存就被占用了，堆中也就少了块可用的空间。在WEB REQUESTS中这种类型的的内存泄漏很典型，一两个内存对象的泄漏可能不会导致程序服务器的崩溃，但是10000或者20000个就可能会导致这个恶果。而且，大多数这些泄漏的对象并不是象DOUBLE或者INTEGER这样的简单对象，而可能是存在于堆中一系列相关的对象。例如，你可能在不经意间引用了一个Person对象，但是这个对象包含一个Profile对象，此对象还包含了许多拥有一系列数据的PerformanceReview对象。这样不只是丢失了那个Person对象所占据的100 bytes的内存，你丢失了这一系列相关对象所占据的内存空间，可能是高达500KB甚至更多。 为了寻找这个问题的真正根源，你需要判断是内存泄漏还是以OutOfMemoryError形式出现的其他一些故障。我使用以下2种方法来判断: ----深入分析内存数据 ----观察堆的增长方式 不同JVM(JAVA虚拟机)的调整程序的运作方式是不相同的，例如SUN和IBM的JVM，但都有相同的的地方。 SUN JVM的

内存管理方式 SUN的JVM是类似人类家族，也就是在一个地方创建对象，在它长期占据空间之前给它多次死亡的机会。SUN JVM会划分为: 1 年轻的一代(Young generation)，包括EDEN和2个幸存者空间(出发地和目的地the From space and the To space) 2 老一代(Old generation) 3 永久的一代(Permanent generation)图1 解释了SUN 堆的家族和空间的详细分类对象在EDEN出生就是被创建，当EDEN满了的时候，垃圾收集器就把所有在EDEN中的对象扫描一次，把所有有效的对象拷贝到第一个幸存者空间，同时把无效的对象所占用的空间释放。当EDEN再次变满了的时候，就启动移动程序把EDEN中有效的对象拷贝到第二个幸存者空间，同时，也将第一个幸存者空间中的有效对象拷贝到第二个幸存者空间。如果填充到第二个生存者空间中的有效对象被第一个生存者空间或EDEN中的对象引用，那么这些对象就是长期存在的(也就是说，他们被拷贝到老一代)。若垃圾收集器依据这种小幅度的调整收集(minor collection)不能找出足够的空间，就是象这样的拷贝收集(copy collection)，就运行大幅度的收集，就是让所有的东西停止(stop-the-world collection)。运行这个大幅度的调整收集时，垃圾收集器就停止所有在堆中运行的线程并执行清除动作(mark-and-sweep collection)，把新一代空间释放空并准备重启程序。 图2和图3展示的是了小幅度收集如何运行图2。对象在EDEN被创建一直到这个空间变满。图3。处理的顺序十分重要:垃圾收集器首先扫描EDEN和生存者空间，这就保证了占据空间的对象有足够的机会证明自己是有效的。图4展示了一个小幅度调整是如何运行的图4:当垃圾收集器释放所有的无效的对象并把有效的对象移动到一个更紧凑整齐的新

空间，它将EDEN和生存者空间清空。 以上就是SUN实现的垃圾收集器机制，你可以看出在老一代中的对象会被大幅度调整器收集清除。长生命周期的对象的清除花费的代价很高，因此如果你希望生命周期短的对象在占据空间前及时的死亡，就需要一个主垃圾收集器去回收他们的内存。 上面所讲解的东西是为了更好的帮助我们识别出内存泄漏。当JAVA中的一个对象包含了一个并不想要的一个指向其他对象的引用的时候，内存就会泄漏，这个引用阻止了垃圾收集器去回收它所占据的内存。采用这种机制的SUN 虚拟机，对象不会被丢弃，而是利用自己特有的方法把他们从乐园和幸存者空间移动到老一代地区。因此，在一个基于多用户的WEB环境，如果许多请求造成了泄漏，你就会发现老一代的增长。 图5显示了那些潜在可能造成泄漏的对象:主收集器收集后遗留下来占据空间的对象会越来越多。不是所有的占据空间的对象都造成内存泄漏，但是造成内存泄漏的对象最终都占据者空间。如果内存泄漏的确存在，这些造成泄漏的对象就会不断的占据空间，直至造成内存溢出。 因此，我们需要去跟踪垃圾收集器在处理老一代中的运行:每次垃圾收集器大幅度收集运行时，有多少内存被释放?老一代内容是不是按一定的原理来增长? 图5。阴影表示在经过大幅度的收集后幸存下来的对象，这些对象是潜在可能引发内存泄漏的对象 一部分这些相关的信息是可以通过跟踪API得到，更详细的信息通过详细的垃圾收集器的日志也可以看到。和所有的跟踪技术一样，日值记录详细的程度影响着JVM的性能，你想得到的信息越详细，付出的代价也就越高。为了能够判断内存是否泄漏，我使用了能够显示辈分之间所有的不同的较权威的技术来显示

他们的区别，并以此来得到结果。SUN 的日志报告提供的信息比这个详细的程度超过5%，我的很多客户都一直使用那些设置来保证他们管理和调整垃圾收集器。下面的这个设置能够给你提供足够的分析数据: 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