HotSpot JVM内存管理概述

Memory Management in the Java HotSpot™ Virtual Machine

By zhangxsh

sbios@126.com

2013.11.3

目录

前言

本文翻译自 HotSpot JVM 内存管理规范的白皮书：Memory.Management.White.Paper

第一章 介绍

Java 的一个长处就是提供了自动内存管理机制，因此屏蔽了开发人员进行直接内存管理的复杂性。这篇文章提供了一个针对 J2SE 5 内存管理的概览。描述了垃圾收集器(garbage collectors)如何有效的进行内存管理，并给出了一些关于收集器的选择和收集器运行时内存区域大小配置的建议。这篇文章还列出了影响垃圾收集器 (garbage collector)行为最常用的选项，并且提供了很多详细文档的链接，因此本文可以作为参考资料使用。

第二章为初学者介绍自动内存管理的概念。本章会简单的讨论一下自动内存管理和程序员直接内存管理。

第三章整体介绍了垃圾收集的概念、设计决策和性能度量。同时，介绍了基于对象生命周期的分代内存管理机制，这是一种常用的内存组织方式。这种方法已经被证明在减少垃圾收集暂停时间、很宽泛应用范围内的整体消耗方面很有效。剩下的部分是针对 HotSpot JVM (by zhangxsh, HotSpot JVM 是 Sum/Oracle 开发的一种 JVM，其他的还有 BEA/Oracle 的 JRockit，IBM 的 J9 VM 等) 的内容。

第四章介绍了 4 种垃圾收集器，其中一个是在 J2SE 5.0 update6 中增加的，并且描述了分代的内存组织。对于每种垃圾收集器，简要描述他们使用的算法类型和调优参数。

第五章描述了一种在 J2SE 5.0 提供的新技术，它会根据应用系统运行的平台和操作系统自动选择垃圾收集器和堆大小，并根据用户行为动态进行垃圾收集优化。这种技术称为功效学 (ergonomics)。

第六章给出了一些垃圾收集器的选择和配置的推荐配置，同时也提供了一些处理 OutOfMemoryError 错误的建议。第七章简要描述了一些用于垃圾收 集性能 (garbage collection performance)评估的工具。第八章列出了用于控制垃圾收集器的选择和行为最常用的命令行参数。最后，第九章提供了更多涉及这篇文章内容的详细文档。

第二章 直接 VS 自动内存管理

内存管理是这样的一些过程，识别哪些对象不再有用，回收（释放）这些对象使用的内存，使这些内存在随后的分配中可用。在一些编程语言中，内存分配是程序员的责任。这项复杂的任务导致了很多常见的错误，如怪异、错误的程序行为和程序崩溃。结果是，开发人员很大比例的时间都在调试解决这些错误。

在直接内存管理的程序中经常犯的一个错误是悬挂引用（dangling references）。对象使用的空间被回收时，可能还有其他对象被引用着。如果一个对象拥有这样（悬挂）的引用，当它试图访问原始对象时，很可能这块空间已经分配给了新的对象，结果导致了未预期的访问。

另外一个在直接内存管理中常见的错误是内存泄露（space leaks）。内存分配完不再使用后却没有释放就会产生这样的错误。例如，你打算释放一个链表（linked list）使用的空间时犯了一个错误，只回收了链表的第一个对象，其余的对象就不再被引用了，然而这些对象脱离了程序的控制，再也无法使用或恢复。如果产生了足够的泄露，内存将持续消耗，直到再也没有可用的部分。作为替代方案，一种称为垃圾收集（garbage collector）的自动内存管理方法正在被广泛使用，尤其是在现代的面向对象语言中。自动内存管理使得编写出更多抽象的接口、更多稳定代码成为可能。

垃圾收集避免了悬挂引用问题，因为被某处引用的对象永远不会被收集，内存不会被释放。垃圾收集同样解决了上面提到的内存泄露问题，因为不再被引用的内存将自动释放。 （by zhang: 实际上java中依然有“内存泄露”问题，只是这种泄露与上文中提到的传统上的泄露不同。可以理解为对内存的不恰当使用，会导致垃圾收集频繁发生[本应存储对象的没有存储下来]，或OOM错误[本来应释放的内存没有释放]）

第三章 垃圾收集概念

垃圾收集器负责以下几个事情:

1.分配内存 2.确保被引用的对象留在内存中 3.回收执行中代码的引用无法到

达的对象占用的内存（by zhangxsh：强调执行中是为了排除对象互相引用的情况。A、B互相引用，但没有任何执行中代码引用他们，A、B也应被回收）

对象被引用称为活着的（live）。不再被引用的对象称为死掉的（dead），术语叫垃圾（garbage）。发现和释放（或者叫回收（reclaiming））这些垃圾占用的空间叫做垃圾收集（garbage collection）。

垃圾收集可以解决很多内存分配问题，却不能解决所有的内存分配问题。例如，你可以创建对象并无限期的引用着直到没有内存可用。垃圾收集在使用其自身的时间和资源上是一个复杂的任务。

垃圾收集器负责的内存是由精确的算法来组织、分配和回收的，并对开发人员隐藏。空间通常从一个被称为堆（heap）的非常大的内存池分配出来。垃圾收集的时间由垃圾收集器决定。通常，整个堆或堆的一部分被填满或者达到某个阈值的时候，会触发垃圾收集。

满足内存分配的请求是一个困难的任务，这其中包括要从堆中找到一个足够大的内存块。对于大部分的动态内存分配算法，其主要的问题是需要在保持内存分配和回收效率的同时避免内存碎片。

1. 优秀的垃圾收集器的特征

垃圾收集器必须是安全有效的。就是说，使用中的数据永远不能被错误的释放，同时在很少的几个收集周期内垃圾就应该被回收。

垃圾收集器的执行效率也必须很优秀，暂停时间不能太长。暂停的时候，应用系统是不运行的。然而，像大部分的计算机系统那样，这里也必须在时间、空间、频率之间做出权衡。例如，堆很小的时候，垃圾收集的速度很快但堆被填满的速度更快，这样就需要更频繁的垃圾收集。相反的，大的堆填满的速度慢，收集的频率也慢，但花费的时间会比较长。

另一个特征是有限的内存碎片（fragmentation）。当垃圾对象的内存释放后，释放的空间会在各种各样的区域形成小块的空隙以至于可能导致没有一个足够大的连续区域分配给较大的对象。一种减少内存碎片的方法叫做压缩（compaction），在下面垃圾收集器的设计决策部分会讨论到。

可扩展性同样很重要。在多核系统上运行的多线程程序中，内存分配、垃圾收

集都不能成为瓶颈。

2. 设计决策

设计和选择垃圾收集算法时必须做出一系列选择:

串行还是并行

在串行收集中，同一时间只做一件事情。例如，即便有多个cpu可用，却只有一个进行收集工作。当使用并行收集时，垃圾收集任务会分成几个小的部分，这些小的部分在不同的cpu上同时执行。同时执行的操作使得收集速度更快，它的代价是额外的复杂性和可能更多的内存碎片。

并发的（Concurrent）还是停止一切（Stop-the-world）

当执行停止一切（Stop-the-world）的垃圾收集时，应用系统在收集期间完全暂停（suspended）了。另外一种选择是，一个或多个垃圾收集任务可以和应用系统同时并发的执行。通常，一个并发的收集器，大部分工作并发的执行，但仍会有一些短暂的暂停（stop-the-world pauses）。停止一切的垃圾收集比并发收集更简单，因为整个堆都冻结了，在收集期间对象不会改变。它缺点是一些应用程序不喜欢的暂停（paused）。相应的，并发收集的暂停时间更短，但收集器必须格外的小心，执行收集的同时应用系统可能会改变对象的状态，这会增加一些开销。并发收集会影响性能并且需要较大的堆内存。

压缩or不压缩or拷贝

当收集器判定内存中的对象哪些是存活的哪些是垃圾之后，收集器可以压缩（compact）内存，将所有存活的对象放到一起，从而完全的恢复剩余的内存。压缩之后，在第一个空闲位置分配内存将会非常的容易和迅速。可以用一个简单的指针维持下一个可分配对象的位置。相对压缩的收集器，非压缩（non-compacting）的收集器在原地（in-place）释放垃圾对象占用的空间，它不会像压缩的收集器那样移动存活的对象创建一个大的回收区。非压缩的好处是收集完成的很快，缺点是可能有内存碎片。一般来说，从原地释放的内存分配空间比从压缩的堆分配内存更困难些。它必须搜素堆空间找到一个足够大能容纳新对象的连续内存区域。第三种可供选择的是复制（copying）收集器，拷贝（或疏导 evacuates）所有活动的对象到另一个不同的内存区域。它的好处是原来的区域可以直接置空，简单快速的为随后

的内存分配做好准备，缺点是需要额外的空间和时间。

3.性能度量

一些指标用来评估垃圾收集的性能，包括：

吞吐量（Throughput）

一个很长的周期中，除去花费在垃圾收集上的时间占总时间的百分比。

垃圾收集开销（Garbage collection overhead）

与吞吐量相反，这是垃圾收集占总时间的百分比。by zhangxsh：为什么需要两个指标呢？对于并发的垃圾收集算法，垃圾收集的部分任务和应用系统同时运行导致上述两个指标加起来会大于100%）

暂停时间（Pause time）

垃圾收集发生时，应用系统暂停的时间。

收集频率（Frequency of collection）

垃圾收集相对于应用系统运行发生的频率。

占用空间（Footprint）

一种大小的指标，例如堆大小。

反应时间（Promptness）

对象变成垃圾之后到内存可用的时间一个交互式应用需要较低的暂停时间，反之持续的执行时间对于非交互式应用更加重要。一个实时应用程序要求在垃圾收集中的暂停以及收集器的整个周期拥有较少的抖动。运行在个人计算机或嵌入式设备中的应用可能主要关心小的空间占用。

4.分代收集

使用分代（generational collection）收集技术时，内存分为很多代（generations），分离的存储池存储不同年龄的对象。例如，最通用的配置中有两代：一个用于存放年轻的对象，另一存放年老的对象。

不同的代使用不同的算法执行垃圾收集任务，每个算法会基于本代独特的特征进行优化。分代的垃圾收集基于一种被称为弱分代假设（weak generational

hypothesis），它是关于在几种语言（包括java语言）编写的应用程序中观察到的结果：

1、大部分的分配的对象不会被引用（存活）很长时间，这些对象在年轻的时候就死掉了；

2、年老对象引用年轻对象的情况很少出现。

年轻代的收集发生的相对频繁、有效、快速，因为年轻代的空间通常比较小并且有很多的对象都不再被引用。

在年轻代几次收集后仍然生存的对象最终会晋升（promoted）或者被授予（tenured）到年老代。如图 1。年老代一般比年轻代大，并且增长的速度很慢。结果是，年老代的收集很少发生，但是会花费更长的时间才能完成。

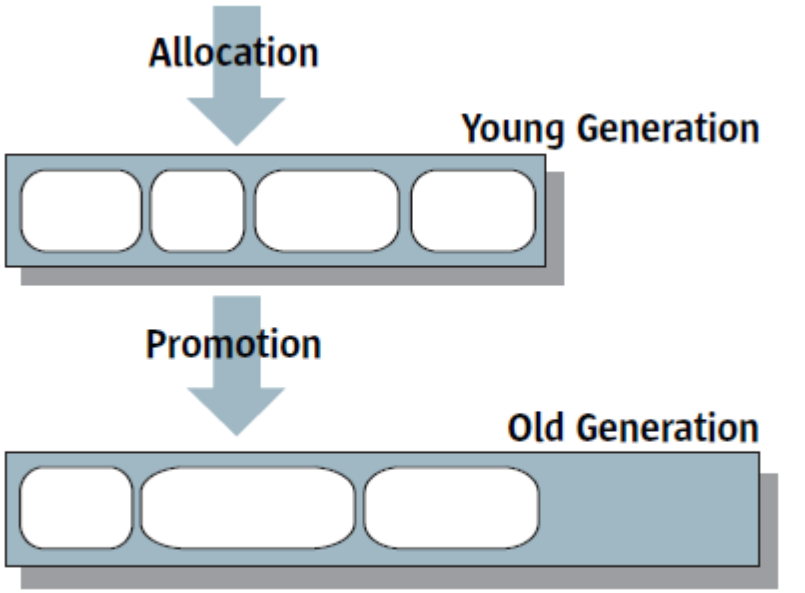

Figure 1. Generational garbage collection

为年轻代设计的收集算法主要关注在速度方面，因为垃圾收集经常发生。另一方面，在空间方面更有效率的算法管理着年老代，因为年老代占据了大部分的堆空间并且年老代的垃圾密度比较低。

第四章 J2SE 5.0 HotSpot JVM 中的垃圾收集

很多公司都有自己的JVM实现，被Oracle收购的sun公司开发的JVM实现名为

HotSpot。这一实现是我们最常用到的。

还有哪些JVM实现呢？比较有名的有Oracle之前收购的BEA公司（就是以前做WebLogic的那家公司）的JRockit，IBM公司的J9VM等。还有个半像不像的Dalvik（Goolge开发的运行在android上那玩意）

当然不知名的还有很多，你可以参考这个列表了解下：

http://en.wikipedia.org/wiki/List_of_Java_virtual_machines

自J2SE 5.0 update 6起，Java HotSpot虚拟机包括四种垃圾收集器。所有的收集器都是代化的。本章描述这些分代和收集的类型，讨论为什么对象的分配通常是快速高效的。然后提供关于每个收集器的详细信息。

1.HotSpot 中的分代

在Java HotSpot 虚拟机中，内存被组织为3个代：年轻代（young generation）、年老代（oldgeneration）和持久代（permanent generation）。大部分对象最初在年轻分配。年老代保存着几次年轻代收集后仍然存活的对象和一些可能直接分配到年老代的大对象。持久带存放着JVM认为可以简化垃圾收集管理的对象，如类和方法以及他们的描述信息。

年轻代包括一个伊甸（Eden）区加上俩个小的生还者区（survivor spaces），如图 2。大部分对象直接分配在伊甸区（Eden）。(上面提到的一些大对象可能直接分配在年老代。)最后一次垃圾收集后生存的对象保存在生还者区，在它们在被认为“足够老”以晋升到年老代之前仍有机会死掉。在任意给定的时间，一个生还者区（在图中标记为 From）保存着这些对象，而另一个则是空的，直到下次垃圾收集前也不会使用。

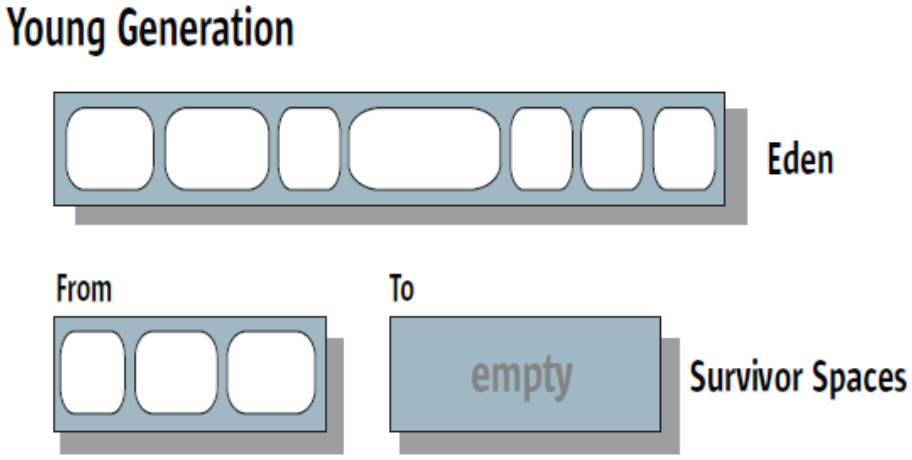

Figure 2. Young generation memory areas

2.垃圾收集的类型

当年轻代填满的时候，将执行一个只在这一代的运行的垃圾收集 -年轻代垃圾收集（young generation collection）（有时候隶属于一个主收集（minor collection））。当年老代或持久代填满的时候，将执行一个典型的全收集（Full collection）（有时候隶属于一个主收集（minor collection））。换句话说，所有的代都会被收集。通常的，使用针对年轻代设计的算法在年轻代先执行，因为这是在年轻代标记垃圾最有效的算法。之后，收集器指定的年老代收集算法（old generation collection algorithm）在年老代和持久代使用。如果有压缩，每个代分别进行。

有时候，年轻代首先收集完后，年老代太满，以至于存不下从年轻代晋升到年老代的对象。在这种情况下，除了CMS收集器外的其他收集器的年轻代收集算法都不运行，代替它的是，年老代的收集算法将用在整个堆上。（年老代收集算法CMS是一个特列，因为它不能用来收集年轻代）

By zhangxsh: 这里有点绕口，一个收集器（collector）可能包括多个算法（algorithm），一些用于年轻代，一些用于年老代。这里的意思是CMS这种收集器外的其他的收集器会在年轻代使用年老代的收集算法（整个堆自然包括年轻代），由于年老代太满的原因。CMS收集器不这样的原因是因为CMS算法不能用在年轻代。

3.快速分配

在下面的垃圾收集器描述中你会看到，在很多情况下都存在一个连续的很大的内存块可用于分配对象。用一种被称为空闲指针（bump-the-pointer）的简单技术在这些块上分配内存是很高效的。就是说，前一次分配的痕迹会保留下来，当有一个新的分配请求时，需要做的仅仅是检查新对象是否能够放的下，如果能，则更新指针引用并初始化对象。

对多线程应用程序来说，分配操作必须是线程安全的。如果使用全局锁保证线程安全，会使得到一代的分配成为瓶颈，并会影响性能。代替它的是HotSpot JVM采用一种称为Thread-Local Allocation Buffers(TLABs)的技术。每个线程从自己拥有的缓存（意思是代中很小的一个部分）中分配从而改善了多线程分配的能力。由于线程只能分配对象到自己的TLAB中，利用空闲指针技术在不需要任何的锁的情况下内存分配依然很快。很少发生的同步只有在线程填满了自己的TLAB后申请一个新的的时候才需要。有几个技术用于最大限度的减少由于使用TLAB技术引起的空间浪费。例如，TLABs浪费的Eden空间被分配器控制在不到1%的水平。结合使用TLABs和使用空闲指针的线性分配（linear allocations）使得每次分配都很有效，只需要大概10条本地指令（native instructions）。

4.串行收集器（Serial Collector）

在串行收集器中，年轻代和年老代的收集都是串行的（只使用一个CPU），并且使用停止一切的模式。就是说收集发生时会挂起应用程序。

串行收集器的年轻代收集图 3 演示了在年轻代使用串行收集器的操作。在伊甸区（Eden）存活的对象除了那些太大而不合适放入的对象都要复制到在图中标记为“To”的最初是空闲状态的生还者区。那些太大的对象直接复制到年老代。另一个使用中的生还者区（标记为 From）中的对象中，较为年轻的对象也复制到标记为 To 的生还者区中，较为年老的则复制到年老代。注意：如果“To”的区域满了，从 Eden 区或 From 区存活的对象就不再复制到 To 了，而是直接晋升而不管这些对象在年轻的的垃圾收集中存活了几次。复制完成后，根据定义，不需要检查，任何留在 Eden 区或 From 区的对象都不是存活的。(这些垃圾对象在图中标记为 X，但真实的收集

器不会检查或标记这些对象。)

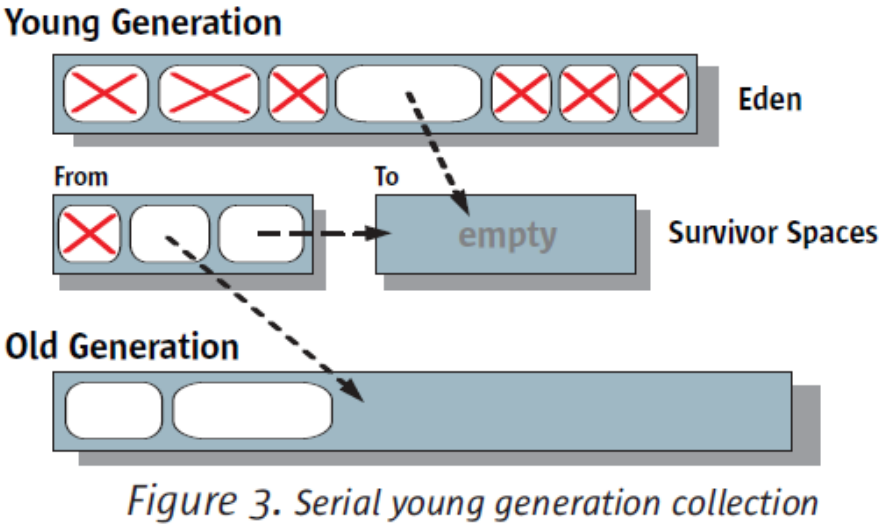

Figure 3. Serial young generation collection

图3 年轻代的串行收集

年轻代的收集完成后，Eden 区和以前使用的生还者区都被置空，只有之前空闲的生还者区保存活着的对象。这时，生还者区交换了角色。如图 4。

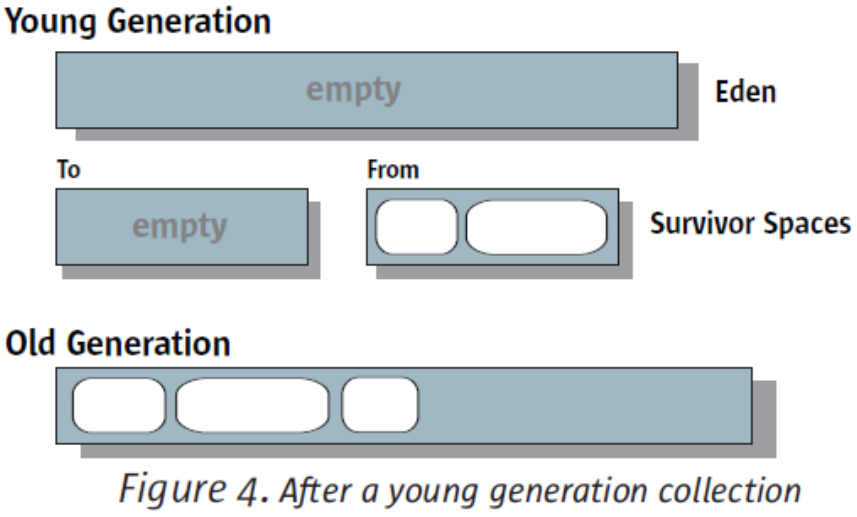

Figure 4. After a young generation collection

图4 年轻代收集完成之后

串行收集器的年老代收集串行收集器在年老代和持久代使用标记-清扫-压缩（mark-sweep-compact）算法。标记阶段，收集器识别哪些对象仍然活着。清扫阶段“扫荡”整个代，识别垃圾。之后，收集器执行平移压缩（sliding compaction），

将存活的对象平移到代的前端（持久代类似），相应的在尾部留下一整块连续的空闲空间。如图 5。压缩后，以后的分配就可以在年老代和持久代使用空闲指针（bump-the-pointer）技术。(by zhangxsh: 上一篇文章的快速分配部分有介绍。)

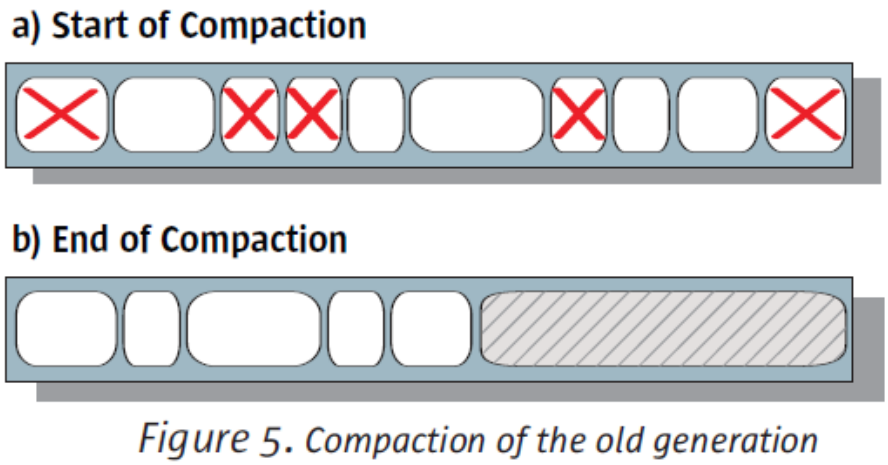

Figure 5. Compaction of the old generation

图 5 年老代的压缩

何时使用串行收集器

串行收集器作为一种选择，在那些运行在客户端环境（client-style machines）的对短暂停没有要求的应用程序用中大量使用。

串行收集器是大多数客户端式（client-style machines）机器上运行的应用程序的选择，这些应用对短暂停没有要求。在今天的硬件上，串行收集器可以有效的管理许多拥有64M堆内存的不平凡的应用程序，并且拥有相对短的在最坏情况下小于半秒的全收集表现。

选择串行收集

在第五章描述中，J2SE 5.0 把串行收集器自动选为非服务器级机器（not server-class machines）的默认垃圾收集器。在其他机器上，串行收集器可以用命令行参数-XX:+UseSerialGC 明确指定。

5.并行收集器（Parallel Collector）

今天，很多java应用程序运行在拥有很大的物理内存的多核机器上。并行收集器（parallel collector），又名吞吐量收集器（throughput collector）被设计出来以便利用多核的优势，而不是空闲很多cpu，只用其中一个做垃圾收集。

并行收集器的年轻代收集

并行收集器使用一个串行收集器使用的年轻代收集算法的并行版本。它仍然是一个 stop-the-world 的拷贝算法，但使用多个 CPU 并行的运行，减少了垃圾收集的开销，因此增加了吞吐量。图 6 说明了年轻代的串行收集器和并行收集器的不同。

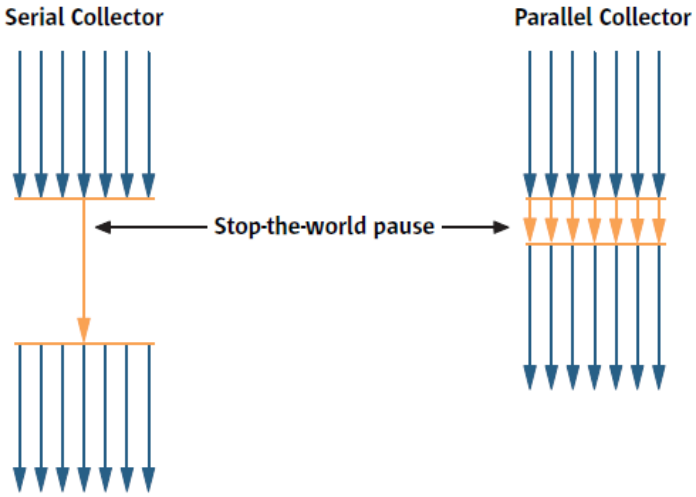

Figure 6. Comparison between serial and parallel young generation collection

图6 比较年轻代的串行和并行收集

并行收集器的年老代收集

并行收集器在年老代使用与串行收集器相同的标记-清扫-压缩（mark-sweepcompact）收集算法。

何时使用并行收集器

在拥有不止一个 CPU 的机器上运行的应用程序可以从并行收集器中获得好处，同时这些应用不能有暂停时间的约束，因为虽然不常见但可能较长的年老代收集仍可能发生。例如，如下应用对于并行收集器通常是合适的：批处理、广告、工资、科学计算等等。

相比并行收集器你可能更愿意考虑并行压缩收集器（parallel compacting collector）（在下面描述），因为它在所有代执行并行收集，而不仅仅是年轻代。

选择并行收集器

J2SE 5.0把并行收集器自动选为服务器级机器（server-class machines）（在第五章定义）的默认垃圾收集器。在其他机器上，并行收集器可以用命令行参数-XX:+UseParallelGC明确指定。

6.并行压缩收集器（Parallel Compacting Collector）

在J2SE 5.0 update 6中包含了并行压缩收集器（parallel compacting collector）。它与并行收集器的不同是在年老代垃圾收集中使用了一个新的算法。注意：最终，并行压缩收集器会替换并行收集器。

并行压缩收集器的年轻代收集

与并行收集器在年轻代使用的收集算法一致。

并行压缩收集器的年老代收集

在并行压缩收集器中，年老代和持久代使用stop-the-world的、主要是并行的平行压缩算法。包括三个阶段。首先，每个代逻辑的分为固定大小的区域。在标记阶段（marking phase），最初在应用程序代码中能够直接到达的对象集合划分给垃圾收集线程，然后并行的标记所有存活的对象。如果一个对象是活着的，关于这个对象的大小和位置的信息就会更新到它所在区域的数据中。

摘要阶段（summary phase）作用在区域上，而不是对象上。基于之前几次收集后的压缩结果，通常每个代的左侧的一些部分是很稠密的，存放着大部分存活的对象。压缩他们以从这些稠密的区域回收大量的空间是不值得的。所以在摘要阶段的第一件事是检查区域的密度，从左边开始，直到达到一个临界点，从这个点及其右侧的区域回收空间相比压缩的消耗是值得的。这个临界点左侧区域归为稠密前缀（dense prefix），在这些区域中的对象不会移动。临界点右侧的区域则被压缩以消灭所有的死亡空间。摘要阶段计算并存储在每个被压缩区域中存活的对象的第一个字节的位置。注意：摘要阶段一般实现成串行阶段；并行化是可行的，但是对性能来说没有并行化标记和压缩阶段更重要。

在压缩阶段（compaction phase），垃圾收集线程利用摘要数据确定哪些区域需要填满，然后每个线程独立的拷贝数据到对应的区域中。最后产生一个在一端很稠密，另一端只有一个大空闲块的堆空间。

何时使用并行压缩收集器

相比并行收集器，并行压缩收集器对于运行在不止一个cpu的机器上的应用程序拥有更多好处。额外的在年老代并行的操作减少了暂停时间，并且使得并行压缩

收集器相比并行收集器对于有暂停时间要求的应用更合适。并行压缩收集器不适合在大型共享机器（例如SunRays）上运行的应用，在这些机器上一个独立应用不能独占多个CPU太长时间。在这些机器上，应当考虑减少垃圾收集线程（使用命令行参数 -XX:ParallelGCThreads=n）或者选择一个不同的收集器。

选择并行压缩收集器

如果你想使用并行压缩收集器，必须通过命令行参数-XX:+UseParallelOldGC显示的指定。

7.并发标记-清扫收集器（Concurrent Mark-Sweep (CMS) Collector）

对于很多应用程序来说，从始到终的吞吐量并没有更快的响应速度重要。年轻代的垃圾收集一般不会导致长时间的暂停。但是，年老代的收集，尽管比较少，会导致长时间的暂停，特别是堆非常大时。为了解决这个问题，HotSpot JVM包括了一个称为并发标记-清扫收集器（concurrent mark-sweep (CMS) collector）的收集器，也称为低延时收集器（low-latency collector）。

CMS收集器的年轻代收集

与并行收集器一样。

CMS收集器的年老代收集

使用CMS收集器时，大部分的年老代收集工作都与应用系统并行的执行。

CMS收集器的一个收集周期从一个短暂的暂停开始，称为初始标记（initial mark），此时识别初始的应用程序代码直接引用的活动对象集合。然后进入并发标记阶段（concurrent marking phase），收集器从上面那个集合开始标记所有的间接引用的活动对象。由于在标记的同时，应用程序依然在运行并行在不断更新引用字段，所以在并发标记阶段结束时无法担保所有的活动对象都被标记。为了解决这个问题，应用程序会二次暂停，叫做重新标记（remark），重新访问所有在并发标记期间更改过的对象作为最终标记。重新标记的暂停时间通常比初始标记的暂停长很多，因此会使用多线程以提高效率。

在重新标记阶段的最后，保证所有在堆中的活动对象都已经标记，所以随后的并发清扫阶段（concurrent sweep phase）回收所有识别的垃圾。

图 7 说明了串行标记-清扫-压缩收集器与 CMS 收集器在年老代收集上的差别。

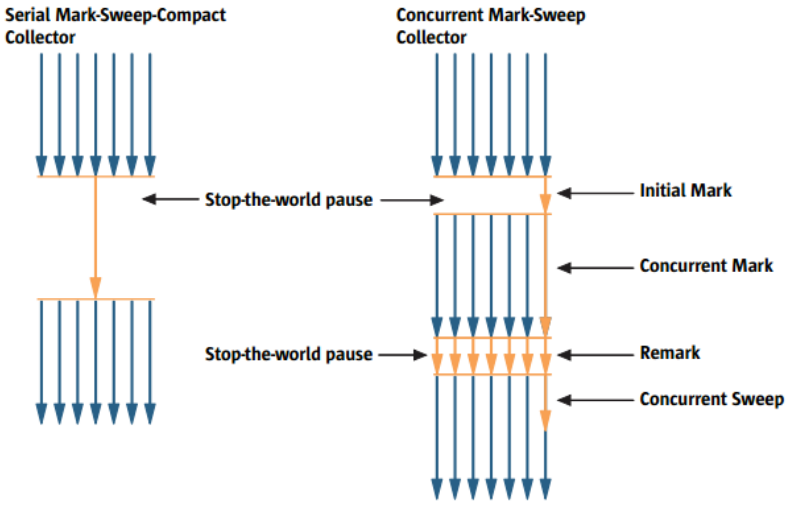

Figure 7. Comparison between serial and CMS old generation collection

图 7比较串行和CMS年老代收集

因为一些任务，如在重新标记阶段重新访问所有对象，收集器增加了大量的工作，同时开销也显著地增加了。对于想要减少暂停时间的收集器来说，这是一个典型需要权衡的场景。

CMS 收集器是仅有的无压缩（non-compacting）的收集器。就是说，当它释放了由垃圾对象占用的空间后，不会移动存活的对象到年老代的一边。参考图 8.

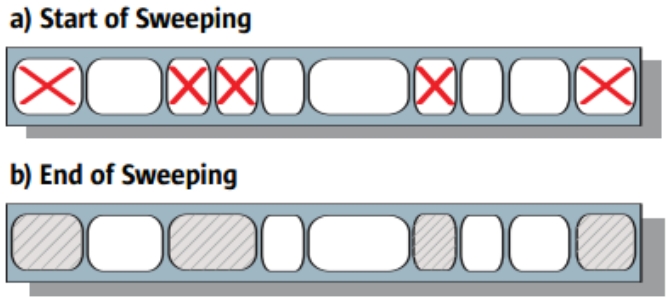

Figure 8. CMS sweeping (but not compacting) of old generation

图8 CMS在年老代的清扫（无压缩）

这节省了时间，但是导致空闲空间不再连续。收集器再也不能用一个简单的指针表示可分配下一个对象的空闲空间。取而代之，现在必须使用一个空闲列表。它创建一些列表，将未分配的内存区域链接到一起，当需要分配对象时，必须搜索相应的列表（基于需要的内存大小）找到一个足够大的区域以容纳该对象，这比使用

空闲指针（bump-the-pointer）技术在年老代分配对象更加耗时。年轻代的收集也增加了额外的开销，因为大部分的年老代收集是由于年轻代收集导致的对象晋升。

CMS收集器的另外一个缺点是相比其他收集器需要更大的堆内存。由于允许应用程序在标记阶段运行，应用程序可能会继续分配内存，从而可能导致年老代持续增长。此外，虽然收集器保证在标记阶段找到所有活动的对象，但一些对象可能这个阶段变为垃圾，这些垃圾直到下次年老代垃圾收集之前不会回收。一些对象引用着 悬浮垃圾（floating garbage）。

最后，由于没有压缩会导致产生内存碎片。为了处理碎片，CMS收集器跟踪流行的对象大小，估算将来的需求，并可能分裂或加入空闲块以满足需求。

不像其他收集器，CMS收集器不是在年老代满的时候启动的。而是试图启动的足够早以便能在年老的满之前完成收集工作。否则，CMS收集器回到被并行或串行收集器使用的更耗时（time-consuming）的暂停一切的标记-清扫-压缩（stop-the-world mark-sweep-compact）算法。为了避免这种情况，CMS收集器的开始时间是基于之前垃圾收集的耗时以及年老代收集频率的统计结果的。如果年老代的占用率超过初始占用率也会导致CMS收集器启动收集工作。初始占用率的数值由如下命令行参数设置 -XX:CMSInitiatingOccupancyFraction=n，这里的n是年老代大小的百分比。默认值是68。

总之，相比并行收集器，CMS收集器减少年老代的暂停时间 -有时是很可观的 -伴随着年轻代暂停时间的些微延长，吞吐量的一点降低，和额外的堆内存需要。

增量模式（Incremental Mode）

CMS收集器可以用在一种并发阶段是增量的模式下。这种模式通过定期停止并发收集阶段退换后台处理给应用程序来减少长并发阶段的影响。这个工作是这样实现的，收集器将时间分成小块然后在年轻代收集的间隙执行。当收集器运行在非常少的处理器的机器上（例如1或2个），应用程序又非常需要收集器提供的短暂停时，这个特性就非常有用。对于使用此模式的更多信息，请参考在第九章提到的“Tuning Garbage Collection with the 5.0 Java Virtual Machine”。

何时使用CMS收集器

如果你的应用程序需要更短的垃圾收集暂停，并且能承受在应用程序运行期间共享处理器资源给垃圾收集器（由于并发，CMS收集器在收集期间占了一部分本应分配给应用程序的CPU周期），就可以使用CMS收集器。特别的，应用程序持有一大

堆的存活时间又长的数据（一个很大的年老代），同时又运行在2个或以上处理器的机器上，有利于从这个收集器获得好处。例如web服务器。任何应用当有短暂停的需要时都应该考虑CMS收集器。在单处理器上，合适大小年老代的交互式应用程序也可能获得比较好的效果。

选择 CMS 收集器

如果想使用 CMS 收集器，你必须明确的通过这个命令行参数选择-XX:+UseConcMarkSweepGC。如果想运行在增量模式，通过 -XX:+CMSIncrementalMode 选项启用。

第五章 功效学-自动选择和行为优化

在J2SE 5.0，垃圾收集的默认值：垃圾收集器、堆大小以及JVM的类型（客户端还是服务器）都会根据应用运行的硬件平台和操作系统自动选择。相比之前设置命令行参数的方式，自动选择很好的匹配了不同类型的应用系统。

另外，并行收集器增加了一种新的动态优化收集算法。在这种方法中，用户指定渴望的行为，垃圾收集器动态的调整堆区域的大小力图实现所需的行为。依赖于平台的默认选择和垃圾收集器自动调整所需的行为称谓工效学。工效学的目标是提供很好的性能，同时只需要很少的命令行参数优化。

1.自动选择收集器、堆大小和 VM 类型

拥有下面特性的认为是服务器类型的机器

□1、2个或更多的物理CPU，并且

□2、2G或更多的物理内存

这个服务器类型机器的定义适用于除了运行windwos操作系统的32位平台外的所有平台。

一个机器如果不是服务器类型的，默认的设置是：

□1、client类型的JVM

2、串行收集器（serial garbage collector）

□3、初始4M的堆

4、最大64M的堆

在服务器类型的机器上，JVM通常都是server JVM，除非你显示的指定-client参数。运行server JVM的服务器类型的机器默认的收集器是并行收集器（parallel collector），否则默认是串行收集器。

使用平行收集器在服务器类型机器上运行的JVM（client或者server），默认的初始和最大堆如下：

□1、初始堆是1/64的物理内存，最大1GB。（注意：最小初始值是32MB，因为服务器类型的机器至少拥有2GB内存，它的1/64是32MB）

2、最大堆是1/4的物理内存，最大1GB

除此之外，非服务器级的机器使用相同的默认大小(4MB 初始大小，最大 64MB)。默认值通常可以用命令行参数改写。有关的参数在第八章给出。

2.基于行为的并行收集器优化

在J2SE 5.0中，并行收集器加入了一种新的基于应用系统期望垃圾收集行为的优化方法。命令行参数用于指定最大暂停时间和应用程序吞吐量所期望的行为。

最大暂停时间目标

最大暂停时间目标通过如下参数指定：

-XX:MaxGCPauseMillis=n

这应解读为对于平行收集器的一个提示，期望暂停时间是n毫秒或者更少。并行收集器调整堆大小和其他垃圾收集器相关的参数力图保持垃圾收集暂停时间短于n毫秒。这些调整可能导致垃圾收集器降低整体应用的吞吐量，有时期望的暂停时间也实现不了。

最大暂停时间目标分别应用在每个代。特别的，如果目标没达到，会减少代的大小尝试实现目标。默认情况下没有最大暂停时间的设置。

吞吐量目标

吞吐量目标在垃圾收集消耗的时间和非垃圾收集消耗的时间（归为应用时间）方面进行测量。这个目标通过如下参数指定：

-XX:GCTimeRatio=n

垃圾收集时间与应用时间的比例是

$$1/(1+n)$$

例如-XX:GCTimeRatio=19设置了一个垃圾收集占总时间5%的目标。默认的目标是1%（即n=99）。垃圾收集花费的时间是所有代的总和。如果吞吐量目标没有达到，会增加代的大小以便让应用系统在两次垃圾收集之间可以运行的时间更长。大的代需要更长的时间填满。

占用空间目标

如果吞吐量目标和最大暂停时间目标实现了，垃圾收集器会减少堆的大小直到其中之一（总是吞吐量目标）不能实现。然后解决没满足的目标。

目标的优先级

并行收集器优先努力实现最大暂停时间目标。只有当这个目标达到后才解决吞吐量目标。同样的，在前两个目标达成后才会关心占用空间目标。

第六章 推荐

在上一章描述的工效学引导的自动的垃圾收集器、虚拟机、和堆大小的选择对于很大一部分的应用是合理的。因此，选择和配置垃圾收集器的初次推荐是什么都不做！就是说，不用指定特别的垃圾收集器，以及其他。让系统基于应用运行的平台和操作系统自动选择。然后测试你的应用。如果性能是可接受的，也就是拥有足够的吞吐量和足够短的暂停时间，你的事就做完了。你不需要排查故障或者更改垃圾收集设置。

另一方面，如果你的应用似乎存在垃圾收集相关的性能问题，最简单的你首先要做的是依据应用程序和平台特性想想默认选择哪个垃圾收集器。或者，先选择一个，然后看看性能是否能接受。

你可以用第七章描述的那些工具测量和分析性能。基于测量结果，考虑修改参数，比如堆大小或者垃圾收集行为。第八章列出了一些常用的命令行参数。注意：最好的性能优化方法是先测量，后调整。使用与你的代码真实运行相关的测试进行测量。并且，防止过度优化，因为应用程序的数据集，硬件，等等 -甚至垃圾收集实现！ -可能会随时间改变。

这一章提供了选择垃圾收集器和设置堆大小的信息。然后提供了一些优化并行

垃圾收集器的建议，然后给出了关于OutOfMemoryError的处理建议。

1.选择一个不同的垃圾收集器

第四章说到对于每个收集器建议使用的场景。第五章描述了什么样的平台默认会自动选择串行或并行收集器。如果你的应用程序或环境特性期望一个相比默认不同的垃圾收集器，通过下面命令行参数明确的指定它:

-XX:+UseSerialGC

-XX:+UseParallelGC

-XX:+UseParallelOldGC

-XX:+UseConcMarkSweepGC

2. 堆大小

第五章说明了默认的初始和最大堆大小。这些大小对于大部分应用来说是何时的，但是如果你正在分析的性能问题（参见第七章）或者是OutOfMemoryError（本章后面会讨论）表明某个代或整个堆的大小存在问题，你可以通过第八章指定的命令行参数调整这些大小。例如，在非服务器类型的机器上，默认的最大堆大小64MB通常太小了，你可以通过-Xmx参数指定一个更大的。除非你遇到了长暂停时间的问题，否则应分配尽可能多的内存到堆。吞吐量与可用内存成比例变化。拥有足够的内存是影响垃圾收集性能最重要的因素。

决定总共分配多少内存给整个堆后你可以考虑每个代的大小。第二重要影响垃圾收集性能是的分配给年轻代的比例。除非你发现过多的年老代收集或暂停时间，应分配足够多的内存给年轻代。然而，如果你使用的是串行收集器，不要分配多余一半的堆内存到年轻代。

当你使用某种并行收集器时，优先指定期望的行为，而不是确切的堆大小值。然后让收集器自动动态的调整堆大小以便实现这些行为，下面会讲到。

3. 并行收集器的优化策略

如果垃圾收集器选择（自动或明确指定）一个并行收集器或并行压缩收集器，

然后指定一个吞吐量目标（参见第五章），通常对于你的应用程序就足够了。不要指定堆的最大值，除非你明确的知道需要的堆大小比默认要大。堆会自动增长或收缩到一定大小以支持选定的吞吐量目标。可以预期，在堆初始化和应用程序行为发生变化时可能发生一些堆大小的震荡。

如果堆大小增长到最大值，在大部分情况下这意味着基于这个最大值吞吐量目标无法达到。设置最大堆内存到接近物理内存并且不会导致交换应用程序的值。再次执行应用程序。如果吞吐量目标依然没有达到，则说明对于在这个平台上可用的内存来讲设置的目标太高了。

如果吞吐量目标可以达到，但是暂停时间太长，选择一个最大暂停时间目标。选择最大暂停时间目标可能意味着吞吐量目标达不到，所以要选择一个可接受的妥协的值。

垃圾收集器在满足互相矛盾的目标时可能导致堆大小来回震荡，直到应用程序达到一个稳定状态。迫使满足吞吐量目标（它需要大堆）与最大暂停时间目标和最小占用目标（这两个目标需要更小的堆）相矛盾。

4.遇到 OutOfMemoryError 时做什么

应用程序终端显示的java.lang.OutOfMemoryError是很多程序员都需要解决的通用问题。这个错误在没有空间分配对象时抛出。意思是，垃圾收集器不能再制造一些空间来容纳一个新对象，并且堆空间不能再扩展。OutOfMemoryError错误不一定意味着内存泄露。它可能是一个简单的配置错误，例如为应用程序配置不足的内存（或默认值没有设置）。

诊断OutOfMemoryError错误的第一步是检查完整的错误信息。在异常信息中，补充信息在“java.lang.OutOfMemoryError”之后提供。这里提供一些常见的可能的补充信息，以及他们是什么意思，针对他们该做些什么。

Java heap space

这表明对象不能在堆中分配。这个问题可能仅仅是一个配置问题。例如，通过命令行参数-Xmx指定的（或者默认的）最大堆内存对于应用来讲是不够的，你就会遇到这个错误。它也可能是一种迹象，不再需要的对象由于应用程序无意的引用着而不能被垃圾收集器回收。HAT工具（参见第七章）可以用来查看所有可引用到的

对象，以及理解这些对象是如何被引用的。另一个可能导致这个错误的原因可能是应用过多的使用终结器（finalizers）以至于执行终结的线程跟不上加入终结队列的速度。jconsole管理工具可以用来监测等待终结的对象数量。

PermGen space

这表明持久代满了。如前所述，这个区域是JVM用来存放元数据信息的。如果应用程序加载了大量的类，则需要增加持久代大小。你可以通过这个命令行参数 -XX:MaxPermSize=n实现，这里的n指大小。

Requested array size exceeds VM limit

这表明应用程序企图分配的数组比堆还大。例如，应用程序尝试分配一个512MB的数组，但是堆最大才256MB，就会出现这个错误。大部分情况下，这个错误要不就是因为堆太小，要不就是因为应用程序计算数组的大小时发生错误引起了一个bug。

一些在第七章描述的工具可以被用于诊断 OutOfMemoryError 问题。最有用的一些工具室 Heap Analysis Tool (HAT)、jconsole 管理工具、附加-histo 参数的 jmap 工具。

第七章 评估垃圾收集性能的工具

各种各样的诊断和监视工具可以用来评估垃圾收集性能。本章简要概述他们中的几个。可以通过第九章中的“Tools and Troubleshooting”链接获得更多的信息。

1. 命令行参数 -XX:+PrintGCDetails

获取垃圾收集初始信息最简单的方法之一是指定这个命令行参数 -XX:+PrintGCDetails。对于每一次收集，形如这样的信息都会在输出的结果中：垃圾收集前后每个不同代活动对象的大小，每个代可用的空间以及垃圾收集消耗的时间。

2. 命令行参数 -XX:+PrintGCTimeStamps

当使用了 -XX:+PrintGCDetails参数后，除了上述输出外，这个参数在每次垃圾收集开始时输出一个时间戳。时间戳有助于帮助你将垃圾收集日志与其他日志时间关联起来。

3.jmap

jmap是一个包含在Solaris™操作系统环境和Linux（但不支持Windows）的JDK中的工具（by zhangxsh：这篇文章说的JDK5时代，在JDK6的Windows版本中已经支持了）。这个工具打印关于一个运行着的JVM或core文件内存相关的统计信息。当没有附加任何命令参数时，它打印加载的共享对象列表，很接近Solaris系统中pmap工具的输出。对于更具体的信息，这些参数可以使用：-heap、-histo和 -permstat。

-heap用于获取如下信息：垃圾收集器的名称、具体算法的细节（例如并行垃圾收集的线程数）、堆的配置信息、堆的使用信息的摘要。

-histo用于获取关于类的堆的直方图。对于每个类，打印其在堆中实例的个数，这些对象消耗的内存总量的字节数和类的全限定名。直方图对于了解堆是如何使用的非常有用。

对于动态产生和加载大量类的应用（例如JSP、web容器）来说配置持久代的大小非常重要。如果应用加载了“太多”的类将发生OutOfMemoryError错误。jmap的 -permstat参数可以用来获取持久代对象的统计信息。

4.jstat

jstat 使用 HotSpot JVM 内建的方式提供运行中的应用的性能和资源消耗信息。但诊断性能问题时可以使用这个工具，特别是当问题与堆大小和垃圾收集相关时。它的许多选项能够打印垃圾收集行为和性能的统计数据和各代的用量。

5. HPROF:堆分析器

HPROF是JDK5.0提供的一个简单的分析器代理。它是一个使用JVM TI接口链接到JVM的动态链接库。它使用ASCII和二进制格式输出特征信息到文件或socket。这

些信息未来可以用前端工具分析。

HPROF能够呈现CPU使用率、堆分配统计和锁争用特征。另外，它可以输出完整的heap dump并且报告JVM中所有线程和锁的状态。当分析性能、锁争用、内存泄露等问题时，HPROF非常有用。参见第九章HPROF文档的链接。

6.HAT:堆分析工具（Heap Analysis Tool）

堆分析工具(HAT)帮助分析无意识的对象持有（unintentional object retention)。这个术语是用来描述一个对象不再需要，但由于存在某个活动对象到它的引用路径仍然存活。HAT 基于 HPROF 产生的堆快照提供了一个非常方便的方法浏览对象拓扑。这个工具支持很多查询，包括“显示所有从根集合到这个对象的引用路径”。参见第九章 HAT 文档的链接。

第八章 有关垃圾收集的关键参数

一些命令行参数可以用来选择垃圾收集器，指定堆或代的大小，修改垃圾收集行为，获取垃圾收集统计数据。本章给出一些最常用的参数。有关各种各样参数更多完整的列表和详细信息可以参见第九章。注意：指定的数字可以以“m”或“M”结尾表示兆字节，以“k”或“K”结尾表示千字节，以“g”或“G”结尾表示千兆字节。

垃圾收集器的选择

选项	选择的垃圾收集器
-XX:+UseSerialGC	Serial (串行收集器)
-XX:+UseParallelGC	Parallel (并行收集器)
-XX:+UseParallelOldGC	Parallel compacting (并行压缩收集器)
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep (CMS)

垃圾收集器统计

选项	描述
-XX:+PrintGC	每次垃圾收集时输出基本信息。
-XX:+PrintGCDetails	每次垃圾收集时输出更详细的信息。
-XX:+PrintGCTimeStamps	在每次垃圾收集事件开始输出时间戳。与-XX:+PrintGC 或-XX:+PrintGCDetails 一同使用。

堆和代的大小

选项	默认值	描述
-Xmsn	参见第五章	堆初始大小，以字节为单位。
-Xmxn	参见第五章	堆最大值，以字节为单位。
-XX:MinHeapFreeRatio=minimum and -XX:MaxHeapFreeRatio=maximum	40（最小），70（最大）	堆空闲空间与总大小的比值范围。应用在每个代上。例如，如果最小值为 30 并且某个代的空闲百分比低于 30%时，这个代会扩展以保持 30%的空闲空间。相似的，如果最大值是 60 并且空闲百分比超过 60%，代会收缩以保持只有 60%的空闲空间。
-XX:NewSize=n	依赖平台	年轻代的初始大小，以字节为单位。
-XX:NewRatio=n	客户端 2，服务器 8	年轻代和年老代的比例。例如，如果 n 是 3，比例是 1:3，Eden 和生还者区的加和是年轻代和年老代大小总和的 1/4。
-XX:SurvivorRatio=n	32	每个生还者区与 Eden 区的比例。例如，如果 n 是 7，每个生还者区是年轻代的 1/9（不是 1/8，因为有 2 个生还者区）。
-XX:MaxPermSize=n	依赖平台	持久代的最大值。

并行和并行压缩收集器的选项

选项	默认值	描述
-XX:ParallelGCThreads=n	CPU 的个数	垃圾收集的线程数。
-XX:MaxGCPauseMillis=n	无	期望垃圾收集器的暂停时间小于等于 n。
-XX:GCTimeRatio=n	99	设置垃圾收集时间占总时间的目标为 1/(1+n)。

CMS 收集器的选项

选项	默认值	描述
-XX:+CMSIncrementalMode	禁用	启用增量模式。并行阶段是增量的，并行阶段周期性暂停以释放处理器给应用程序。
-XX:+CMSIncrementalPacing	禁用	是否允许 CMS 处理器基于应用程序行为自动放弃之前的处理器。
-XX:ParallelGCThreads=n	CPU 的个数	年轻代并行收集和年老代并行收集部分的线程数。

第九章 更多参考信息

HotSpot Garbage Collection and Performance Tuning

- garbage Collection in the Java HotSpot Virtual Machine (http://www.devx.com/Java/Article/21977)
- Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine (http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

Ergonomics

- Server -Class Machine Detection (http://java.sun.com/j2se/1.5.0/docs/guide/vm/server -class.html)
- Garbage Collector Ergonomics (http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc -ergonomics.html)

- Ergonomics in the 5.0 Java™ Virtual Machine

(http://java.sun.com/docs/hotspot/gc5.0/ergo5.html)

Options

- Java™ HotSpot VM Options

(http://java.sun.com/docs/hotspot/VMOptions.html)

- Solaris and Linux options

(http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html)

- Windows options

(http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html)

Tools and Troubleshooting

- Java™ 2 Platform, Standard Edition 5.0 Trouble -Shooting and

Diagnostic Guide

(http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf)

- HPROF: A Heap/CPU Profiling Tool in J2SE 5.0

(http://java.sun.com/developer/technicalArticles/Programming/HPROF.html)

- Hat: Heap Analysis Tool

(https://hat.dev.java.net/)

Finalization

- Finalization, threads, and the Java technology -based memory model

(http://devresource.hp.com/drc/resources/jmemmodel/index.jsp)

- How to Handle Java Finalization's Memory -Retention Issues

(http://www.devx.com/Java/Article/30192)

Miscellaneous

- J2SE 5.0 Release Notes

(http://java.sun.com/j2se/1.5.0/relnotes.html)

- Java™ Virtual Machines

(http://java.sun.com/j2se/1.5.0/docs/guide/vm/index.html)

- Sun Java™ Real -Time System (Java RTS)

(http://java.sun.com/j2se/realtime/index.jsp)

- General book on garbage collection: Garbage Collection: Algorithms for Automatic Dynamic Memory Managementby Richard Jones and Rafael Lins, John Wiley & Sons, 1996.