当前版本：v1.0

序言：

当你获取到此文档即代表你同意此链接声明的条例，如果不同意请删除此文档。

往后本人翻译的所有文档链接都会分享到这里。所有文档的版本都会按照语义化版本的约定，即【主版本.次版本.修订号】，实际情况不需要修订号。如果文档有错误请指出，发邮件到我邮箱，mailto:806846483@qq.com，邮件主题请按照如下格式：

邮件主体：文件名-版本-页码

邮件正文：指出是什么错，如果让你翻译，你会怎么翻译，如果有参考链接（官方的链接？），请一并附上。如果是实践结果和文档描述不相符合，请描述清楚你的实践过程，我会尽可能再次重试你的实践过程。

另外，需要队友一起翻译技术文档，希望组个翻译社，看完一个文档不够屌，来一起翻译和探讨吧，独乐乐不如众乐乐，希望你有时间并且能坚持下去。有想法的可以加本人微信。

Memory Management in the Java HotSpot™ Virtual Machine

Sun Microsystems
April 2006

目录

1. 介绍

JavaTM 2 平台，**J2SE** 的一个优点就是执行自动内存管理，因此让开发者从复杂的内存管理中解放出来。

这个论文提供 **J2SE 5.0** 发行版中 **HotSpot** 虚拟机（**JVM**） 内存管理的概述。它描述了执行内存管理可用的垃圾搜集器，并且提供一些配置选择器，设定对应搜集器操作内存区域大小的建议。同样它也列出一些影响垃圾搜集器行为常用的选项，并提供文档链接查找更加详细的信息。

章节 2 介绍自动内存管理的相关概念。并且对比讨论自动管理和显式管理内存。章节 3 介绍垃圾按代搜集概念，设计上的选择，性能标准。同样也介绍了常见按代组织内存的方式。实践已经证明对各种各样的应用程序内存进行分代可以有效的减少垃圾搜集器的停顿时间和所有花费。

文档剩余部分提供了关于 **HotSpot JVM** 具体的信息。章节 4 描述了四种可以用的垃圾搜集器，包括 **J2SE5.0** 内新增的搜集器和它们组织可用各年代内存的文档。对于每个搜集器，章节 4 描述了使用的搜集算法和何时选择此搜集器。

章节 5 描述了 **J2SE5.0** 中根据平台和操作系统自动选择搜集器，堆大小，**JVM** 类型，基于用户指定行为动态调整搜集器等信息，这种技术叫做人类工程学。

章节 6 提供配置，选择搜集器相关的建议。同样也给出了一些诊断 **OutOfMemoryError** 相关建议。章节 7 简单的描述可用于评估垃圾搜集器性能的工具，章节 8 列出常用选择垃圾搜集器和调整行为的命令行参数。最后，章节 9 提供更详细信息的文档链接。

2. 显式 vs.自动内存管理

内存管理是释放不再使用的对象的内存，内存可在下次申请空间时使用。在某些程序语言中这是程序员的职责。这项复杂的工作导致许多常见的错误，并且这些错误很容易导致不期望的行为和程序宕机。最终导致开发者花费大量时间调试，修复这些错误。

显式管理内存出现最常见的问题就是悬挂引用问题（指针指向一个已经被 delete 的空间,并且空间再次被新的对象使用）。使用这种引用的结果是未知 的。

另外一个常见的问题就是内存泄漏。内存泄漏的意思就是对象已经不再被使用但是内存一直未释放。比如，你准备释放一个列表占用的内存空间，但是你犯了一个错误把列表的第一个对象释放了，剩余的列表节点对你代码而言此时再也不可达，并且再也不能被使用。如果出现太多的内存泄漏，他们会用完所有可用的内存。

一个管理内存可用的方法就是垃圾搜集器，在当代面向对象语言中经常使用。自动内存管理能够提高接口的抽象性，生成更可靠的代码。
垃圾搜集器能够避免引用悬挂问题，因为被引用的对象不会被回收。垃圾回收器同样能够解决内存泄漏，因为它会自动释放不再引用的内存。

3. 垃圾搜集器概念

垃圾回收器期职责

- 开辟空间
- 保证任何引用可达的对象都在内存内
- 回收不再使用的内存

引用可达的对象叫做存活对象。引用不可达对象叫做死亡对象，术语内叫做垃圾。寻找垃圾，释放空间的过程叫做垃圾搜集。

垃圾搜集器能够解决许多内存开辟问题，但是不能解决所有问题。比如你可以这样做，创建一个对象并且引用它们直到没有内存可用。垃圾搜集器是一个复杂的工作，同样也需要花费时间和资源。

垃圾搜集器组织内存，开辟，释放空间的行为对程序员来说是不可见的。空间通常从一个大的内存池中开辟，这个大的内存池通常叫做堆。

何时垃圾搜集行为由垃圾搜集器决定。比较常见的是在堆满后或者在达到阀值时搜集全部或部分堆空间。

满足一个开辟请求是很困难的任务，它需要在堆中找到一个未用的大小合适空间。大多数动态内存开辟算法主要避免内存碎片问题，保持高效的开辟，释放行为。

垃圾搜集器期望的行为

搜集搜集器必须安全和全面。存活的对象必须不能被释放，不再使用的对象应该在很小的搜集周期后释放。

同样也期望垃圾搜集操作很高效，不会导致应用程序长时间停顿。然而，在大多数计算机系统一样，这通常存在时间，空间，搜集频率直接的权衡。比如，堆小就意味着搜集速度快，但是小堆很容易被填满，因此搜集的频率高。相反，大堆不容易被填满，所以搜集频率低，但是搜集时间长。

垃圾搜集器另外一个标准就是造成有限的碎片。当释放空间时，会出现许多很小的，且不连续的内存块，这样就无法给大对象找到一个合适的内存块。解决内存碎片的一个办法是对空间进行压缩，下面讨论垃圾搜集器设计时会讨论压缩。

可扩展性同样也很重要，对于运行在多核上的多线程程序空间开辟行为和垃圾搜集不应该成为扩展的瓶颈。

设计上的选择

设计和选择垃圾搜集器时，可以做出以下选择：

串行 vs 并行

串行搜集器一次只发生一件事。比如，即使有多个 **CPU**，它也只会使用一个 **CPU** 执行垃圾搜集。当使用并行搜集器时，搜集任务被分解成许多小任务，同时在不同 **CPU** 上执行多个小任务。同时进行操作能够让搜集任务更快结束，但是这个会带来一些内存碎片和其他复杂性。

并发 vs 停止应用程序

停止应用程序类型的垃圾回收器工作时让应用程序完全停止。或者是让多个垃圾搜集任务并发执行，这样就可以和应用程序并行。通常并发垃圾搜集器大多数工作并发执行，但同样也需要让应用程序停止一小会。停止应用程序类型的垃圾回收器实现比并发回收器简单。因为它工作时，堆内容固定不变并且在工作时对象保持不变，缺点就是造成应用程序不希望的停止。相对来说，并发回收器让应用停止时间比停止应用程序类型回收器的时间短，但是它需要考虑回收时应用程序同时更新的对象，这样会影响并发搜集器性能，同样它也较大的堆。

压缩 VS 非压缩 VS 复制

在垃圾回收器确定哪些对象是垃圾那些对象存活时，此时可以压缩内存，移动所有存活的对象然后合并剩余空间。在压缩后，在第一块空闲区域开辟空间更容易也更快。可以用一个简单的指针跟踪下一个可用的区域（**bump the pointer** 和 **TLAB**）。相反，非压缩搜集器直接在对象原始位置释放内存，不移动所有存活对象来生成一个较大的空闲空间。但是非压缩回收器能更快的完毕回收工作，副效应就是带来较多的内存碎片。通常，在释放对象的原始位置开辟需要的空间比在压缩后的空间中开辟要耗时。它需要搜索内存空间直到找到一块连续并且大小合适的内存。第三种方案是是复制型搜集器，它复制存活对象至一块不同的内存区域。好处就是，源内存可以直接当做空闲内，能够更快的用于下次开辟空间，但是副效应就是复制对象需要时间，还需要另外一块内存空间。

测量性能

以下几个基准可用于评估垃圾搜集器性能：

吞吐量-未花费在垃圾搜集上的时间，长时间考虑。

垃圾搜集器花费-吞吐量倒数，花费在垃圾搜集上的时间。

停止时间-发生垃圾回收时应用程序停止时间。

占用空间-可测量的大小，比如堆大小。

速度-对象变成垃圾和这个空间变得可用之间的时间。

交互型程序需要短的停止时间，所有可用于执行的时间比非交互型程序重要。实时程序需要垃圾搜集造成的停止时间和花费在搜集上的时间都是一个很小的上界。运行在小型个人 PC 或者嵌入式系统内的应用程序较为关系可用的内存空间。

分代回收

当使用分代搜集时，内存空间被划分成不同的年代，同样对象放在不同年代的内存空间。比较广泛的分代方法是分为俩个年代：一个年轻代，一个年老代。

搜集不同的年代使用不同搜集 算法。它们依据以下理论。

大多数对象不会长时间被引用。

只有少数年轻对象会变老（还没变老就死了）。

年轻代垃圾回收器发生回收行为频繁，回收行为高效并且快，因为年轻代通常比较小，包含许多不再被引用的对象。

年轻代中的对象经过多次回收行为，如果还活着，就转移到年老代。具体查看图 1。年老代的内存通常大于年轻代，并且增加速度比年轻代慢。

最后年老代的回收行为不频繁，但是需要长时间才能完成。

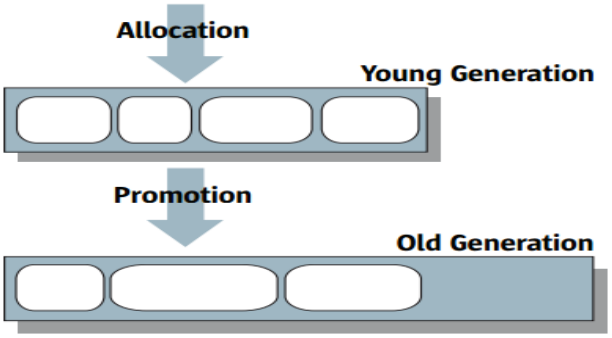

Figure 1. Generational garbage collection

为年轻代选择垃圾回收器主要考虑的因素是速度，因为年轻代搜集行为频繁。另一方面，年老代回收算法在空间使用上更加高效，因为年老代占用,堆大部分内存，而且年老代算法必须在垃圾密度（密度小，对象大，间隔大）很低的情况下工作的很好。

4. J2SE5.0 HotSpot JVM 垃圾搜集器

J2SE5.0 HotSpot JVM 中包含四种垃圾搜集器，并且都是分代搜集器。这个章节描述回收器类型和分代，然后讨论为什么对象开辟空间速度很快，很高效。然后提供更多关于每种回收器的详细信息。

HotSopt 中的分代

HotSpot 虚拟机中的内存划分为年轻代，年老代，持久代。大多数对象在年轻代中开辟空间。年老代中保存经过多次回收后还幸存的对象，当然一些较大的对象可能直接在年老代中开辟空间。持久代保存那些 **JVM** 查找方便的对象(用于 **JVM** 垃圾搜集)，比如对象和方法描述信息，同样也包括类和方法本身。

年轻代由 **Eden** 区域和俩块小的幸存区组成，如图 2 所示。大多数对象在 **Eden** 中开辟空间然后初始化。(正如上面提到少数对象可能直接在年老代开辟空间)幸存区保存年轻代中至少经历过一次回收的对象，但是这些对象在变成年老代之前可能就会死掉。在一个任意的给定的时间点，只使用幸存区中的一个区域保存对象（图中指示的是 **From** 区），幸存区中未保存对象的另外一块区域保存为空闲直到下一次垃圾搜集时使用。

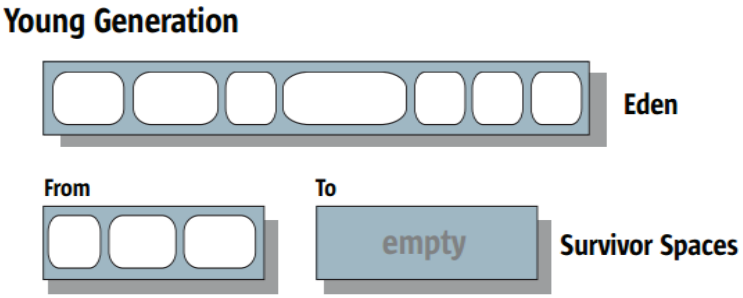

Figure 2. Young generation memory areas

垃圾搜集类型

当年轻代被填满时，一次年轻代搜集在年轻代上执行（有时候也叫做 **minor collection**）。当年老代或者持久代被填满时,执行一次全量搜集（有时候也叫做 **major collection**）。在这种情况下，所有的年代都被搜集。通常年轻代首先被收集，使用

针对年代的算法搜集垃圾，因为这个算法在年轻代中标识垃圾具有很高的效率。然后在年老代和持久代上运行特定的垃圾搜集器。如果有压缩行为，每个代的压缩的行为单独执行。

年轻代先搜集。有时候会造成年轻代提升太多的对象到年老代，导致年老代太过于饱和。在这种情况下，所有的年轻代的搜集器（除了 **CMS 搜集器**）在年轻代上都不运行。相反，年老代垃圾搜集算法在整个堆上使用（**CMS** 年老代算法是比较特殊，因为它不能搜集年轻代）

快速开辟空间

正如你在以下看到的那样，在许多情况下都有一大片连续的内存块用于给对象开辟内存，使用一种比较简单的技术（**bump-the-pointer**）从这样块开辟空间非常高效。保存先前开辟的最后一个对象。当一个新的开辟空间请求需要被满足时，需要检查年代中剩余的空间是否对申请合适，如果合适，更新最后一个对象然后对象初始化。

对于多线程应用程序，开辟空间操作需要多线程安全。如果使用一个全局锁来保证安全，在年代中开辟空间的行为会成为系统瓶颈，然后拉低性能。相反，**HotSpot JVM** 采用一个叫做 **Thread-Local Alloction Buffers**（**TLABs**）的技术。它让每个线程在自己的 **buffer** 里面开辟空间来提升多线程开辟空间的吞吐量(**buffer** 是年代的一小部分)。既然在每个 **TLAB** 里面只有一个线程开辟空间，此时可以使用 **bump-the-pointer** 来快速开辟空间，这样就不需要偶外的锁。仅在很少情况下需要同步实现，比如在线程填满自己的 **TLAB** 的时候需要获取一个新的 **TLAB** 时。**TLAB** 可以多种技术浪费的空间变小。比如，内存分配器切分出 **TLABs** 的浪费小于 **Eden** 的 1%。使用 **TLABs+bump-the-pointer** 技术可以让开辟非常高效，仅需要差不多 10 条原生的指令。

串行搜集器

串行搜集器停止应用程序串行搜集年轻代和年老代 （使用一个 **CPU**）。这样，当垃圾搜集器进行时应用程序执行被强制停止。

年轻代中使用串行搜集器

图 3 指出在年轻代中使用串行搜集器的过程。**Eden** 空间存活的对象被复制到幸存区内（图内标记为 **To** 的那块），那些太大而不适合 **To** 空间的对象不复制到 **To** 空间。这些对象（不适合幸存区的对象）直接被复制到年老代。在 **From** 区域内存活的对象同样被复制到其他区域(图内标记为 **To**)，如果对象太老，直接复制到年老代。(注意：如果 **To** 空间已经满了,从 **Eden** 或者 **From** 来的存活对象会被直接复制到年老代，此时不管此对象在年轻代搜集器下幸存过多少次)。在 **Eden** 或者 **From** 区域内未被复制走的对象都是未活着的对象，它们不需要被再次测试是否存活。(图内标记成 X 的那些对象，注意搜集器不会标记这些对象)

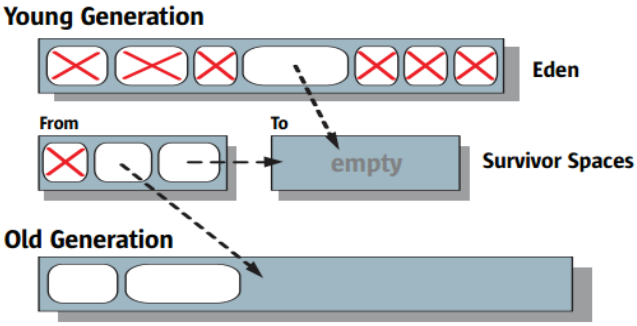

Figure 3. Serial young generation collection

在年轻代搜集器完成后，所有 **Eden** 和幸存区内某一块全是空的（幸存区内另外一块空间保存着存活的对象）。此时，俩个幸存区交换角色。具体查看图 4。

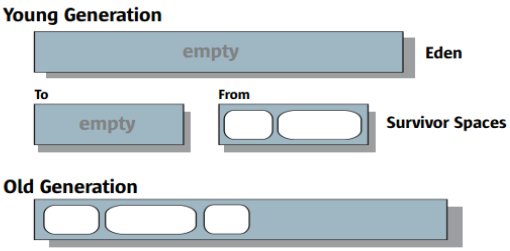

Figure 4. After a young generation collection

年老代使用串行搜集器

在年老代和持久代中串行搜集器使用 **mark-sweep-compact** 算法。在标记阶段，搜集器标示那些存活的对象，在清除阶段，清理区域中所有的垃圾。然后搜集器执行滑动压缩，滑动所有存活对象区至年老代开始地方（在持久代中一样），与其相对的地方都是连续空闲的块。具体查看图 5。压缩使得以使用 **bump-the-pointer** 技术开辟空间很快。

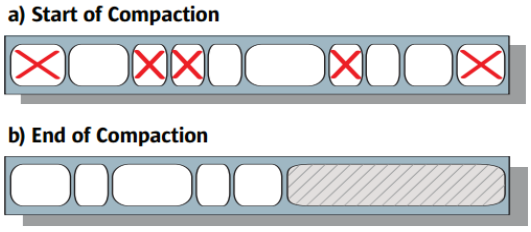

Figure 5. Compaction of the old generation

何时使用串行搜集器

运行在 **client-style** 的虚拟机适合使用串行垃圾搜集器，并且应用程序不需要低停止的时间。在当前的硬件环境下，串行搜集器能够高效的管理许多堆空间为 64MB 的应用程序，并且全量搜集最坏情况下的停止时间小于半秒。

使用串行搜集器

在 **J2SE5.0** 的发行版中，非服务器类机器上默认选择串行垃圾搜集器，正如章节 5 内描述的那样，可以通过指定 **-XX:+UseSerialGC** 命令行参数来显式指定。

并行搜集器

当今许多 **Java** 应用程序运行大内存和多核 **CPU** 的机器上。并行搜集器，也叫做吞吐量搜集器能够更好的利用可用的 **CPU**，而不是让在垃圾搜集器工作时让其他 **CPU** 闲置。

使用并行搜集器搜集年轻代

在年轻代上使用并行搜集器就像串行搜集器的并行版本。它同样是让应用程序停止，复制对象的搜集器，使用多个 **CPU** 并行搜集年轻代，减少垃圾搜集耗费，提供应用程序吞吐量。图 6 展示出并行搜集器搜集年轻代和串行搜集器搜集年轻代的不同。

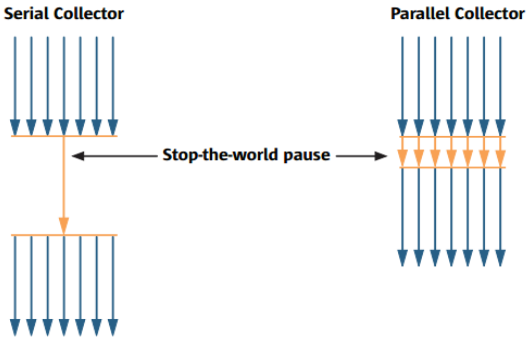

Figure 6. Comparison between serial and parallel young generation collection

使用并行搜集器搜集年老代

并行搜集器和串行搜集器一样使用串行的 **mark-sweep-compact** 算法搜集年老代。

何时使用并行搜集器

运行在多核 **CPU** 上并且能够接手频率低但长停止时间(搜集年老代会停止应用程序)的应用程序能够从并行搜集器上获益。比如，以下环境适合使用并行搜集器，批处理程序，订单类型系统，科学计算等。

你可能想用并行压缩搜集器而不是并行搜集器，因为并行压缩搜集器对所有代的搜集都是并行。

使用并行搜集器

在 **J2SE5.0** 发行版中，当是服务器类型的机器时自动选择使用并行搜集器。在其他类型机器时，可用使用命令行参数 **-XX:+UseParalleGC** 显式使用并行搜集器。

并行压缩搜集器

在 **J2SE5.0 update 6** 中介绍了并行压缩搜集器。与并行搜集器不同的是它使用新的搜集算法搜集年老代。注意：并行压缩搜集器将取代并行搜集器。

在年轻代上使用并行压缩搜集器

年轻代上使用并行压缩搜集器和使用并行搜集器时一样。

在年老代使用并行压缩搜集器

在年老代和持久代搜集时会停止应用程序，进行并行压缩。搜集过程分为三个阶段。首先，每个年代在逻辑上被分为固定大小的区域。在 **marking phase** 时，应用程序直接可达的初始存活对象集合（垃圾回收时使用的根对象）被划分在垃圾回收线程数目内。然后所有存活的对象被并行标记。标记为存活的对象的大小和位置信息都会被更新。

在 **summary phase** 时，对每个区域执行总结操作。由于先前已经进行过压缩行为，每个区域的左端比较密集，包含大多数存活的对象。从如果密集区域中回收对象后几乎不用再次压缩它们。所以，总结阶段的第一件事就是测试区域的密集程度，从最左边的对象开始直到某个对象的右边区域需要被回收，此时这个对象的右边需要被压缩。这个对象的左边叫做浓密的前缀，浓密前缀内的对象不需要被移动。这个对象右边空间内的对象需要清理，并且空间被压缩。对每个压缩后的区域，总结阶段计

算，存储新区存活对象的第一个字节。注意：当前实现，总结阶段实现成串行，并行是可能的，但是它对性能的影响没有标记和压缩阶段那么重要。

在 **compaction phase** 时，垃圾搜集器线程使用总结阶段的数据标识需要被填满的区域，线程能够独立的复制对象到区域内。这样就能导致堆在某一端是浓密的，并且有一大块空闲块在另一端。

何时使用并行压缩搜集器

和并行搜集器一样，并行压缩搜集器适合在机器有多个 **CPU** 时使用。另外，在年老代中并行导致并行压缩搜集器在应用程序对停止时间约束很严格时比并行搜集器更加合适。但是在大量共享的机器（比如 **SunRays**）上使用并行压缩搜集器并不合适。在这种机器上，考虑减少垃圾搜集线程数目（**-XX： ParallelGCThreads=n**）或者选择其他不同的搜集器。

使用并行压缩搜集器

如果你想使用并行压缩搜集器，你必须使用命令行参数**-XX:+UseParallelOldGC。**

并发标记扫除（CMS）搜集器

对于许多应用程序，吞吐量没有响应速度重要。搜集器年轻代可能不会造成太长的停止时间。但是年老代搜集会，虽然它发生的不频繁，特别是在有大堆时。为了解决这个问题，**HotSpot JVM** 包含了一个并发标记扫除（**CMS**）搜集器，有时也叫做低延时搜集器。

年轻代上使用 CMS 搜集器

在年轻代上使用 **CMS** 搜集器和使用并行搜集器一样。

年老代上使用 CMS 搜集器

CMS 搜集器搜集年老代时大多数时候和正在执行的应用程序并发执行。
CMS 搜集器搜集周期开始于一个很小停止时间，叫做 **initial mark**，标识从应用程序代码直接可达的存活对象作为初始集合。然后进入 **concurrent marking phase**，从初始结合开始遍历标记所有存活的对象。在标记阶段正在执行的应用程序会更新引用之间的关系，所以在并发标记阶段不保证所有存活对象都被标记。为了解决这个，应用程序再次被停止一小会，此时叫做 **remark**，重新遍历所有在并发标记阶段被更改过的对象。因为重新标记阶段对象比初始化标记阶段更多，多个线程并行标记会更加高效。

在重新标记阶段结束后，堆中所有存活对象都已经保证被标记，所以，下一个阶段 **concurrent sweep phase** 清理所有垃圾。图 7 展示出在年老代使用串行 **mark-sweep-compact** 搜集器和 **CMS** 搜集器的不同。

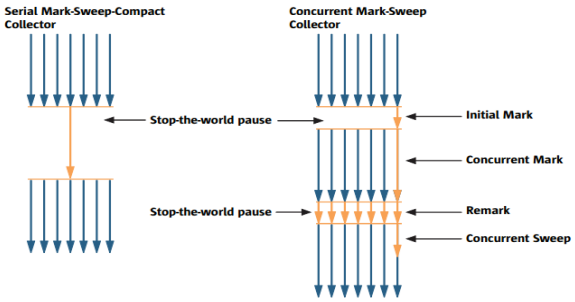

Figure 7. Comparison between serial and CMS old generation collection

因为有些任务会增加垃圾搜集器必须要做的工作，比如在重新标记阶段重新遍历对象，这样同样增加了搜集器的花费。对于大多数企图减少停止时间的搜集器来说，这是一个典型的权衡。

CMS 搜集器是仅有不进行压缩的搜集器。这样，在释放死亡对象占用的空间后，它不移动存活对象到年老代的另一端。请看图 8

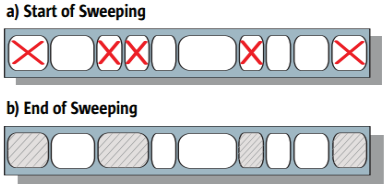

Figure 8. CMS sweeping (but not compacting) of old generation

这样做会节省时间，同样也会造成空间不连续，这样搜集器开辟空间时再也使用不了简单指针指向下一次开辟对象使用的空间。相反，它需要一个空闲区域列表。空闲区域列表连接所有可用的区域，当对象需要开辟空间时，在列表上寻找一个 合适的区域存放该对象，与 **bump-the-pointer** 技术来说，使用空闲列表开辟空间更加耗时。这样同样隐含的增加了年轻代搜集时对象提升的几率。

CMS 搜集器的另一个缺点就是它需要大的内存空间。在标记阶段允许应用程序运行，此时应用程序会继续开辟空间，因此可能潜在的让年老代增长。另外，在标记阶段虽然保证所有存活的对象都能标记到，可能某些对象在标记阶段变成垃圾，因此它们将不会被回收直到下一次年老代回收时。这些垃圾叫做 **floating garbage**。

最后，没有压缩会造成内存碎片。为了解决内存碎片，**CMS** 搜集器跟踪已经开辟的对象大小，预估下一次的请求，可能分裂或者合并空闲块来满足申请。

不像其他搜集器那样，**CMS** 搜集器并不是在年老代变满后才开始搜集。相反，它会在在年老大变满之前尽早的进行垃圾搜集。换句话说，**CMS** 搜集器比 **stop-the-world mark-sweep-compact** 的并行，串行搜集器花费的时间更多。为了避免这个，CMS 搜集器根据先前统计的回收行为发生的频率和搜集的时间来决定开始时间。在年老代占用量超过一定初始占用量时 **CMS** 搜集器会开始工作，初始占用量的值通过**-XX:CMSInitiatingOccupancyFaction=n** 来指定，n 是年老代的百分比，默认值是 68。

总结来说，与并行搜集器相比，**CMS** 搜集器减少搜集年老代停止时间-有时候比较戏剧-提高年轻代搜集停止时间，有时减少吞吐量，需要大的堆。

增量模型

CMS 搜集器能够在增量模式下，此时并发阶段以增量的方式结束。在这种模型下在长并发阶段可以间歇的停下并发阶段（让

出 CPU 给应用程序）来减少影响。年轻代搜集之间的时间块被垃圾搜集器划分为更小的时间块用于调度搜集器。在小数目的处理器上使用 CMS 搜集器，并且在应用程序需要低的停顿时间，这种特色就非常有用(比如..1 个或者 2 个 CPU)。
关于这种模型的更多信息参考章节 9，Tuning Garbage Collection with the 5.0 Java

何时使用 CMS 搜集器

需要低延时的程序并且尽可能和搜集器一起分享处理器资源（因为是并发进行垃圾搜集，搜集周期会获取 CPU 时间片）的应用适合使用 CMS 搜集器.典型的例子,应用程序有相当大量长久存活的对象（很大的年老代），机器有俩个或者更多的处理器。比如，web 服务器，在需要低停顿时间的应用都应该考虑 CMS 垃圾搜集器。同样,它也可能使用于有适中大小的年老代，单核处理器的交互型应用。

使用 CMS 垃圾搜集器

如果你想使用 CMS 搜集，你必须显式的指定命令行参数 -XX:+UseConcMarkSweepGC。如果你想搜集器运行在增量模型下，使用-XX:+CMSIncrementalMode 选项。

5. 人类工程学-自动选择和调整行为

在 J2SE5.0 中,垃圾搜集器,堆大小,HotSpot 虚拟机(客户端还是服务端模型)都是基于应用正在运行的平台和操作系统默认选择。这些自动选择也适合不同类型应用的需要，所需的命令行参数也比以前版本更少。

另外，并行垃圾搜集器可以动态的调整行为。通过这个方法，用户指定期待的行为，垃圾搜集器动态调整堆区域的大小来满足用户的需要的行为。这种基于平台默认选择和垃圾搜集器自动调整的行为叫做人类工程学。人类工程学的目的就是最小化命令行参数，而且提供好的性能。

自动选择搜集器,堆大小,虚拟

满足以下的机器定义为服务器类型机器

2 或者更多的硬件处理器。

2G 或者更大物理内存。

这个定义适用于所有平台，除了 32 位 Windows 的系统。

对于不是服务器类型的机器,JVM 默认类型，垃圾搜集器，堆大小是

客户端类型 JVM

串行垃圾搜集器

初始堆大小 4MB

最大堆大小 64MB

在服务器类型机器上，JVM 总是服务器类型 JVM，除非你显式指定-client 命令行参数。服务器类型机器运行服务器 JVM，默认垃圾搜集器是并行搜集器。如果 JVM 是客户端类型 JVM，默认搜集器就是串行搜集器。

在服务器类型的机器上运行使用并行垃圾搜集器的 JVM（无论是 client 还是 server），堆的默认值和初始值都是

初始堆大小为物理内存的 64 分之一，最大 1GB（注意，最小初始化堆大小为 32MB，因为服务器类型机器定义为有至少 2GB,2GB * 1/64=32MB）

最大堆内存为物理内存的四分之一，最大 1GB。

要不然就使用和非服务器类型机器相同的默认值（最小初始化堆 4MB,最大堆 64MB）。默认值总是能够通过指定命令行参数覆盖,相关选项查看章节 8。

基于行为调整并行搜集器

在 **J2SE5.0** 的发行版中，并行搜集器的新调整方法被加入，基于应用程序在垃圾搜集器方面所需的行为。使用命令行选指定期望的最大停顿时间和应用程序吞吐量。

最大停顿时间目标

最大停顿时间通过**-XX:MaxGCPauseMillis=n** 来指定。这等于暗示并行搜集器应该停顿 n 毫秒或者更少的时间。并行搜集器将调整堆大小和其他垃圾搜集相关参数来满足停顿时间小于等于 n 毫秒。这种调整可能导致垃圾搜集器减少应用程序吞吐量或者在有些时候期望的停顿时间不能达到。
最大停顿时间在每个年代中单独使用。明确来说，如果目标没有达到，年代会变得更小来以此来达到目标。最大停顿时间没有默认值。

吞吐量目标

吞吐量目标是一个测量术语,它指的是花费在垃圾搜集器上时间和未花费在垃圾搜集器上的时间（应用程序时间）。这个目标通过**-XX:GCTimeRation=n** 来指定，垃圾回收时间和应用程序时间百分比为 1 / （1 + n ）
比如，**-XX:GCTimeRatio=19** 表示 5%的时间用于垃圾搜集。默认的值为 1%（n=99）。这个时间是用在所有年代上的总时间。如果吞吐量目标没有达到。年代的大小将增长，以此尽量提升应用程序可用的时间。大容量的年代需要花费更多的时间去填满。

可用空间

如果吞吐量目标和最大停顿时间目标已经达到，搜集器减少堆的大小直到某个目标未达到（通常是吞吐量目标）。然后再解决未达到的目标。

目标优先级

并行垃圾搜集首先满足最大停顿时间目标。然后考虑吞吐量目标。然后是可用空间目标。

6. 建议

先前描述的人类工程学导致自动选择垃圾收集器，虚拟机类型，堆大小，这已经能够满足绝大部分应用的需求。因此，第一点建议就是，不修改虚拟机的任何配置！不指定特定的垃圾收集器类型，让 **JVM** 根据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，吞吐量，停顿时间在可接受范围，配置结束，你压根不用调整垃圾收集器选项。
另一方面来说，如果应用程序的性能问题垃圾回收相关，首先你需要考虑的是默认选择的垃圾回收器（根据平台和操作系统）是否适合。如果不合适，显式指定你认为合适的垃圾回收器，然后看性能是否在可接受范围。

你可以使用第七章节描述的工具来测量，分析性能。你可以根据结果修改选项，比如控制堆大小和垃圾回收器行为，一些常见

的选项将在第八章中共描述。请注意：调整性能最好的方法是先进行测量，然后再调整。使用与你代码实际相关的用例进行测量。同样，不要做过度的优化，因为应用数据集，硬件，垃圾回收器的实现都在不断的发展。

以下一章节提供选择垃圾回收器，指定堆大小的相关信息。然后提供调整并行垃圾搜集器的建议和处理 **OutOfMemoryErrors** 的相关建议。

何时选择不同的垃圾回收器

章节 4 介绍了各种搜集器和使用的环境。章节 5 描述了在何种平台会默认自动选择串行，并行搜集器。如果你的应用程序或者环境标准需要和默认搜集器不同的搜集器。通过以下参数选择显式指定垃圾搜集器。

-XX:+UseSerialGC
-XX:+UseParallelGC
-XX:+UseParalleOldGC
-XX:+UseConcMarkSweepGC

堆大小

章节 5 介绍了初始化和最大堆大小。默认配置对于大多数应用来说或许已经合适，但是如果你分析得出性能问题或者 **OutOfMemoryError** 问题（稍后我们会讨论这个问题）某个代大小或整个堆大小相关，你可以通过命令行参数修改堆大小(参数详见章节 8)。比如，非服务类型的 **JVM** 默认最大堆大小为 64MB,这个大小通常太小，你可以通过**-Xmx** 选项修改最大堆大选项。除非你的问题在于长时间的停顿时间，尽可能分配大的堆内存。吞吐量与可用内存成正比，足够可用的内存是影响垃圾回收器性能的最重要因素。

在决定你尽可能给的可用内存总大小后，然后你可以考虑不同代的大小。第二个影响垃圾搜集器性能的因素是年轻代的比例。给年轻代足够多的内存除非你发现过多的年老代搜集或者停顿时间。然而,当你选择使用串行搜集器，分配给年轻代的内存不要超过总堆内存大小的一半。

当你使用并行搜集器中的一个时，最好指定期望的行为而不是指定精确的堆大小。让搜集器自动动态的修改堆大小来满足你需要的行为，以下讲描述这个策略。

并行搜集器的调整策略

如果垃圾搜集器选择(显示或者自动选择)为并行搜集器或者并行压缩搜集器，预先指定足够你应用程序的吞吐量。不要指定堆内存最大值除非你知道期望的堆内存比默认最大堆内存还大。堆大小将会增长或收缩直到满足选择的吞吐量目标。堆大小初始调整和更改堆大小时波动对程序来说是可接受的。
如果堆大小已经增长到最大，大多数时候意味者吞吐量目标已经不能达到。将最大值设置为最接近平台最大物理内存的值，并且不会引起应用程序内存页交换。重新执行程序。如果吞吐量还是未能达到,说明应用程序时间目标对于平台可用内存来说太高了。

如果吞吐量目标能够达到，但是停顿时间太长，选择最大停顿时间。选择最大停顿时间目标意味者吞吐量目标可能不能满足，所以，根据你的应用程序选择一个折中的值。

堆大小会波动，因为垃圾搜集器会尝试满足竞争的目标，即使应用已经达到一个稳定的状态。满足吞吐量目标（需要大的内存）和最大停顿时间（需要小的内存）和最小使用空间（需要小的内存）在相互竞争着。

出现 OutIfMemoryError 我们能做些什么

许多开发者面临的一个常见问题是应用因为 **java.lang.OutOfMemoryError** 而终止。抛出这个错误是因为没有足够的内存为对象开辟内存。因此，垃圾搜集器找不到任何可用内存来存放新生成的对象，堆也不可能再次进行扩展。**OutOfMemoryError** 并不是一定意味着内存泄漏。也可能是因为配置问题，比如指定的堆大小对于应用来说太小。

诊断 **OutOfMemoryError** 的第一步是检查全部的错误信息。在异常信息内 **java.lang.OutOfMemoryError** 后包含着更多的信息。下面是一些常见的附加信息。

Java heap space

这表明不能在堆中开辟新的对象。这个问题可能是调整配置问题。通过**-Xmx** 指定一个不够的内存大小（或者是默认的内存大小）可能会得到这个错误。同样这也表明对象已经不再需要但是无法被回收，因为应用程序无意识的引用着它们。**HAT** 工具（章节 7）可以用于查看可达到的对象然后分析是那个引用在让它存活。此错误的另一个潜在来源是应用程序过分的调用终结器导致调用终结器的线程无法跟上终结对象进入队列的速度。**jconsole** 管理工具可以用于监视准备终结对象的数目。

PermGen space

PermGen space 表明持久代已经饱和，正如先前描述的那样，**JVM** 使用持久代保存元数据。如果一个应用程序加载大量类，那么持久代需要增加。你可以通过指定命令行参数**-XX:MaxPermSize=n** 来控制持久代大小。

Requested array size exceeds VM limit

Requested array size exceeds VM limit 表明应用程序企图开辟比堆还大的数组。比如，应用程序尝试申请 512MB 大小的数组，但是最大堆大小只有 256MB。那么就会抛出这个错误。大多数时候这个问题可能是堆太小也有可能是应用程序 bug 导致。

第七章描述的工具可以用于诊断 **OutOfMemoryError** 问题。分析这个问题最有用的工具是堆分析工具（**HAT**），**jconsole** 管理工具，**jmap** 工具使用**-histo** 选项。

7. 评估垃圾搜集器性能相关的工具

有各种各样诊断和监控工具能够用于评估垃圾回收器性能。这个章节提供一些关于它们简单的概述。更多信息查看 Tools and Troubleshooting 的链接（在第 9 章）。

-XX:+PrintGCDetails 命令行选项

获取垃圾回收器初始化信息最简单的方式是指定命令行参数**-XX:+PrintGCDetails**。对于每个搜集器，这个选项会打印出在垃圾回收前后各个年代存活对象的数目。各个年代可用的空间和回收花费的时间。

-XX:+PrintGCTimeStamp 命令行选项

这个选项会打印出每次回收的开始时间。除了**-XX:+PrintGCDetails** 选项输出的信息之外。时间戳能够帮助你把垃圾搜集日志和其他事件日志关联起来。

jmap

jamp 是一个命令选项工具，它包含在 **Solaris** 和 **Linux** 操作系统环境的 **JDK** 中。它从正在运行的 **JVM** 或者 **core** 文件中统计内存相关数据并输出。如果没有带任何命令行选项，它打印出已加载的共享对象，这个和 **Solaris pmap** 工具的输出类似。可以用通过**-heap， -histo， -permstat** 选项查看更多信息。

-heap 选项取回的信息包含，垃圾搜集器名字，详细的算法描述信（比如并行垃圾搜集器使用多少个线程）。堆配置信息，堆使用概要信息。

-**histo** 选项用于取回堆中类的直方图。它打印出每个类的实列数目，使用的总内存（总的字节数目），全类命。这个直方图可以用于理解堆内存是怎么被使用的。

对于动态生成类和加载大量类的应用（比如 **Java Server Pages** 和 **web** 容器），配置持久代是很重要的事。如果应用程序加载太多的类，就会抛出 **OutOfMemoryError** 错误。**jmap** 的**-permstat** 选项能够用于获取持久代对象的统计信息。

jstat

jstat 工具使用 **HotSpot JVM** 内置的指令来提供正在运行应用的性能信息和资源消耗信息。这个工具能用于诊断性能问题，通常和堆大小和垃圾回收器相关。各种各样的选项能够用于打印出关于垃圾收集器行为和各个年代使用情况的统计信息。

HPROF:Heap Profiler

HPROF 是 **JDK5.0** 的一个简单代理库。它是一个动态链接库，它使用 **JVM** 和 **Java Virtual Machine Tools Interface(JVM TI)**接口。它输出分析到文件或者 **socket** 通道，可以是字节形式也可以是 **ASCII** 码形式。这些信息将来可以用分析工具进一步处理。

HRPOF 能够显示 **CPU** 使用情况，堆统计信息，监视器竞争信息。另外，它能够输出完整的堆 dump 信息来展示出 **JVM** 中所有监视器，线程的状。**HRPOF** 分析性能,锁争用,内存枯竭,其他问题非常有用。具体查看章节 9 **HPROF** 相关文档。

HAT:堆分析工具

堆分析工具(**HAT**)帮助调试无意识的保留对象。这个术语表示不再需要的一个对象无法被回收因为可以从某个存活的对象通过某些路径达到此对象。**HAT** 使用 **HPROF** 文件，提供一个方便的页面展示对象的拓扑图通过。这个工具允许多种查询，比如。展示所有从 **root** 对象到此对象的所有引用路径。具体查看章节 9 的链接。

8. 和垃圾回收相关的关键选项

可以使用许多命令行参数来选择垃圾回收器，指定堆或者每个年代的大小，修改垃圾回收器行为，取回垃圾回收器的统计数据。这个章节展示一些最常用的选项。更多选项的详细信息查看章节 9。注意，当你指定数目的结尾是 m 或者 M 表示兆字节，k 或者 K 表示千字节，g 或者 G 表示 吉字节。

选择垃圾回收器

选项	选择的垃圾搜集器
-XX:+UseSerialGC	Serial
-XX:+UseParallelGC	Parallel
-XX:+UseParallelGC	Parallel compacting
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep(CMS)

垃圾搜集器统计信息

选项	描述
-XX:+PrintGC	输出每次垃圾回收时的基本信息
-XX:+PrintGCDetails	输出每次垃圾回收时更多的详细信息
-XX:+PrintGCTimeStamps	输出每次垃圾回收时的开始时间，和 -XX:+PrintGC 或 -XX:+PrintGCDetails 一起使用展示何时开始垃圾回收.

堆和年代大小

选项	默认值	描述
-Xmsn	查看第五章	堆的初始化大小
-Xmxn	查看第五章	最大堆大小
-XX:MinHeapFreeRatio=minimum **-XX:MaxHeapFreeRatio=maximum**	40（最小） 70（最大）	空闲内存空间和总堆大小的比例，这个参数在每个代上使用。比如，最小值是 30。某个年代的空闲比例小于 30，然后这个年代的内存空间就会扩张，直到空闲空间/总队空间=30%，通常，如果最大值是 60，如果某个年代空闲空间超过百分之 60，那么内存就会收缩直到空闲空间等于比等于百分之 60。
-XX:NewSize=n	根据平台不同而不同	年轻代初始化字节数。
-XX:NewRation=n	客户端类型 JVM 是 2 服务端类型 JVM 是 8	年轻代和年老代的比例，比如，n=3，年轻代：年老代=1：3，年轻代占总堆的四分之一。
-XX:SurvivorRation=n	32	幸存去和 edge 区的比例，比如，n=7，

		表示幸存区占年轻代的九分之一（不是八分之一，幸存去有俩个）
-XX:MaxPermSize=n	根据平台不同而不同	持久代最大大小。

并行和并行压缩搜集器可用的选项

选项	默认值	描述
-XX:ParallelGCThreads=n	CPU 的数目	搜集垃圾时使用的 CPU 数目
-XX:MaxGCPauseMillis=n	没有默认值	搜集器让应用停止的时间小于或者等于 n 毫秒
-XX:GCTimeRation=n	99	花费在垃圾搜集上的总时间，1 / （n+1）

CMS 搜集器可用的选项

选项	默认值	描述
-XX:+CMSIncrementalMode	关闭(J2SE8 中已经标记为废弃)	是否允许并发阶段以增量方式结束,这样会间歇停止垃圾搜集器过程以此来让出 CPU 给应用程序。
-XX:+CMSIncrementalPacing	关闭	是否允许在放弃 CPU 之前根据应用程序行为自动控制 CMS 工作量。
-XX:ParallelGCThreads=n	CPU 数目	在年轻代搜集时使用的线程数目和年老代并行部分使用的线程数目。

9. 更多信息

HotSpot Garbage Collection and Performance Tuning

- Garbage Collection in the Java HotSpot Virtual Machine
- Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine

Ergonomics

- Server-Class Machine Detection
- Garbage Collector Ergonomics
- Ergonomics in the 5.0 Java™ Virtual Machine

Options

- Java™ HotSpot VM Options
- Solaris and Linux options
- Windows options

Tools and Troubleshooting

- Java™ 2 Platform, Standard Edition 5.0 Trouble-Shooting and Diagnostic Guide
- HPROF: A Heap/CPU Profiling Tool in J2SE 5.0
- Hat: Heap Analysis Tool

Finalization

- Finalization, threads, and the Java technology-based memory model
- How to Handle Java Finalization's Memory-Retention Issues

Miscellaneous

- J2SE 5.0 Release Notes
- Java™ Virtual Machines
- Sun Java™ Real-Time System (Java RTS)

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