# 成为JavaGC专家

## 成为JavaGC专家Part I -- 深入浅出Java垃圾回收机制 - ImportNew

http://www.importnew.com/1993.html

对于Java开发人员来说，了解垃圾回收机制（GC）有哪些好处呢？首先可以满足作为一名软件工程师的求知欲，其次，深入了解GC如何工作可以帮你写出更好的Java应用。

这仅仅代表我个人的意见，但我坚信一个精通GC的人往往是一个好的Java开发者。如果你对GC的处理过程感兴趣，说明你已经具备较大规模应用的开发经验。如果你曾经想过如何正确的选择GC算法，那意味着你已经完全理解你所开发的应用的特点。当然，我们不能以偏概全，这不能作为评价一个好的开发人员的共通标准。但是，我要说的是，深入理解GC是成为一名伟大的程序员的必经之路。

这是成为JavaGC专家系列文章的第一篇，本篇主要针对GC机制进行介绍，在下一篇中，我们将重点探讨分析GC状态以及来自NHN的GC调优的例子。

本文的目的是以一种简单的方式向你介绍GC机制。我希望这些文章能够帮到你。实际上，我的学生已经在Twitter上发布了[一些很好的关于Java内核的文章](http://www.cubrid.org/blog/tags/Java/)，并且大受欢迎。有兴趣的话，你也可以关注他们。

回到正题，咱们继续谈垃圾回收，在学习GC之前，你首先应该记住一个单词：“stop-the-world”。Stop-the-world会在任何一种GC算法中发生。Stop-the-world意味着 JVM 因为要执行GC而停止了应用程序的执行。当Stop-the-world发生时，除了GC所需的线程以外，所有线程都处于等待状态，直到GC任务完成。GC优化很多时候就是指减少Stop-the-world发生的时间。

**按代的垃圾回收机制**

在Java程序中不能显式地分配和注销内存。有些人把相关的对象设置为null或者调用System.gc()来试图显式地清理内存。设置为null至少没什么坏处，但是调用System.gc()会显著地影响系统性能，必须彻底杜绝（还好，我还没有见到NHN的哪个开发者调用这个方法）。

在Java中，开发人员无法直接在程序代码中清理内存，而是由垃圾回收器自动寻找不必要的垃圾对象，并且清理掉他们。垃圾回收器会在下面两种假设（hypotheses）成立的情况下被创建（称之为假设不如改为推测（suppositions）或者前提（preconditions））。

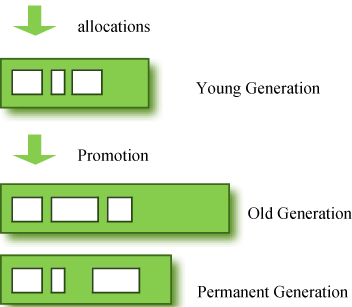
大多数对象会很快变得不可达

只有很少的由老对象（创建时间较长的对象）指向新生对象的引用

这些假设我们称之为弱年代假设（ weak generational hypothesis）。为了强化这一假设，HotSpot虚拟机将其物理上划分为两个–新生代（young generation）和老年代（old generation）。  
新生代（Young generation）: 绝大多数最新被创建的对象会被分配到这里，由于大部分对象在创建后会很快变得不可到达，所以很多对象被创建在新生代，然后消失。对象从这个区域消失的过程我们称之为”minor GC“。

老年代（Old generation）: 对象没有变得不可达，并且从新生代中存活下来，会被拷贝到这里。其所占用的空间要比新生代多。也正由于其相对较大的空间，发生在老年代上的GC要比新生代少得多。对象从老年代中消失的过程，我们称之为”major GC“（或者”full GC“）

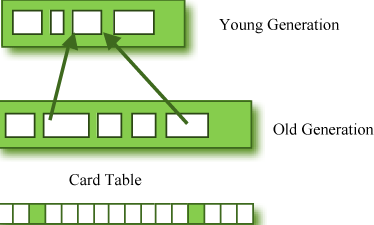
请看下面这个图表。

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-1-GC-Area-Data-Flow.png)

**图1 : GC 空间 & 数据流**

上图中的持久代（ permanent generation ）也被称为方法区（method area）。他用来保存类常量以及字符串常量。因此，这个区域不是用来永久的存储那些从老年代存活下来的对象。这个区域也可能发生GC。并且发生在这个区域上的GC事件也会被算为major GC。

有些人可能会问：  
如果老年代的对象需要引用一个新生代的对象，会发生什么呢？  
为了解决这个问题，老年代中存在一个”card table“，他是一个512 byte大小的块。所有老年代的对象指向新生代对象的引用都会被记录在这个表中。当针对新生代执行GC的时候，只需要查询card table来决定是否可以被收集，而不用查询整个老年代。这个card table由一个write barrier来管理。write barrier给GC带来了很大的性能提升，虽然由此可能带来一些开销，但GC的整体时间被显著的减少。

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure2-Card-Table-Structure.png)

**图 2: Card Table 结构**

**新生代的构成**

为了更好地理解GC，我们现在来学习新生代，新生代是用来保存那些第一次被创建的对象，他可以被分为三个空间

 一个伊甸园空间（Eden ）

 两个幸存者空间（Survivor ）

一共有三个空间，其中包含两个幸存者空间。每个空间的执行顺序如下：

绝大多数刚刚被创建的对象会存放在伊甸园空间。

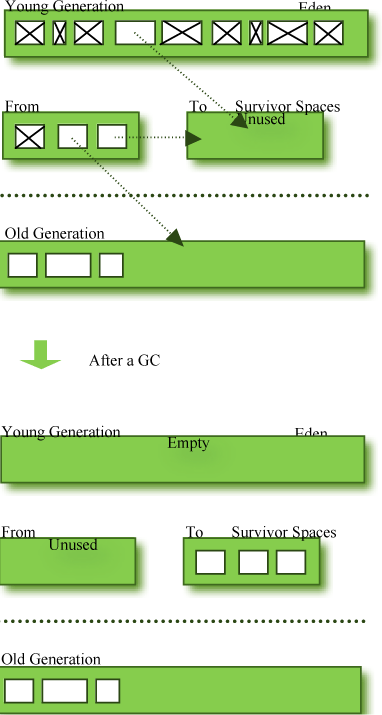
在伊甸园空间执行了第一次GC之后，存活的对象被移动到其中一个幸存者空间。

  此后，在伊甸园空间执行GC之后，存活的对象会被堆积在同一个幸存者空间。

 当一个幸存者空间饱和，还在存活的对象会被移动到另一个幸存者空间。之后会清空已经饱和的那个幸存者空间。

在以上的步骤中重复几次依然存活的对象，就会被移动到老年代。

如果你仔细观察这些步骤就会发现，其中一个幸存者空间必须保持是空的。如果两个幸存者空间都有数据，或者两个空间都是空的，那一定标志着你的系统出现了某种错误。  
通过频繁的minor GC将数据移动到老年代的过程可以用下图来描述：

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-3-Before-After-a-GC.png)

**图 3: GC执行前后对比**

需要注意的是HotSpot虚拟机使用了两种技术来加快内存分配。他们分别是是”**bump-the-pointer**“和“**TLABs（Thread-Local Allocation Buffers）”。**

**Bump-the-pointer**技术跟踪在伊甸园空间创建的最后一个对象。这个对象会被放在伊甸园空间的顶部。如果之后再需要创建对象，只需要检查伊甸园空间是否有足够的剩余空间。如果有足够的空间，对象就会被创建在伊甸园空间，并且被放置在顶部。这样以来，每次创建新的对象时，只需要检查最后被创建的对象。这将极大地加快内存分配速度。但是，如果我们在多线程的情况下，事情将截然不同。如果想要以线程安全的方式以多线程在伊甸园空间存储对象，不可避免的需要加锁，而这将极大地的影响性能。**TLABs** 是HotSpot虚拟机针对这一问题的解决方案。该方案为每一个线程在伊甸园空间分配一块独享的空间，这样每个线程只访问他们自己的TLAB空间，再与bump-the-pointer技术结合可以在不加锁的情况下分配内存。  
以上是针对新生代空间GC技术的简要介绍，你不需要刻意记住我刚刚提到的两种技术。不知道他们不会对你产生什么影响，但是请务必记住在对象刚刚被创建之后，是保存在伊甸园空间的。那些长期存活的对象会经由幸存者空间转存在老年代空间。

**老年代GC处理机制**

老年代空间的GC事件基本上是在空间已满时发生，执行的过程根据GC类型不同而不同，因此，了解不同的GC类型将有助于你理解本节的内容。  
JDK7一共有5种GC类型：

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC  (or “CMS”)
5. Garbage First (G1) GC

其中，Serial GC不应该被用在服务器上。这种GC类型在单核CPU的桌面电脑时代就存在了。使用Serial GC会显著的降低应用的性能指标。  
现在，让我们共同学习每一种GC类型

**1. Serial GC (-XX:+UseSerialGC)**

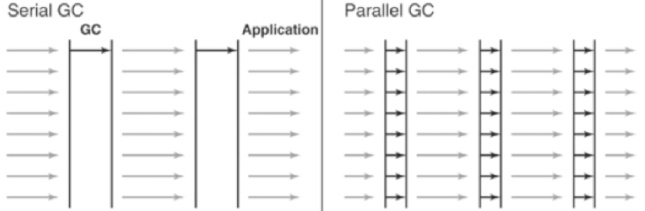
新生代空间的GC方式我们在前面已经介绍过了，在老年代空间中的GC采取称之为”mark-sweep-compact“的算法。

算法的第一步是标记老年代中依然存活对象。（标记）

第二步，从头开始检查堆内存空间，并且只留下依然幸存的对象。（清理）

最后一步，从头开始，顺序地填满堆内存空间，并且将对内存空间分成两部分：一个保存着对象，另一个空着（压缩）。

**2. Parallel GC (-XX:+UseParallelGC)**

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-4-Difference-between-the-Serial-GC-and-Parallel-GC.png)

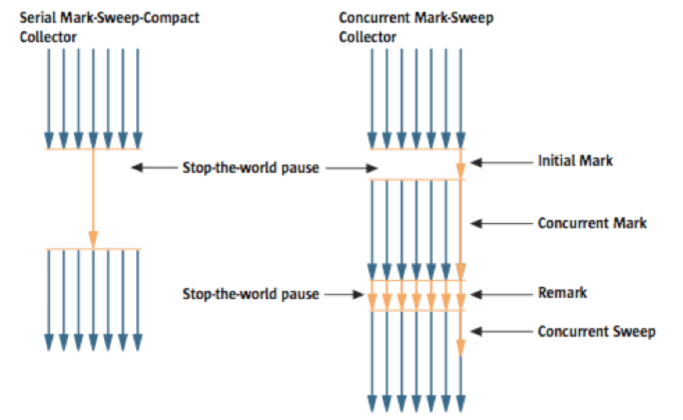
**图 4: Serial GC 与 Parallel GC的区别**

从上图中，你可以轻易地看出serial GC和parallel GC的区别，serial GC只使用一个线程执行GC，而parallel GC使用多个线程，因此parallel GC更高效。这种GC在内存充足以及多核的情况下会很有用，因此我们也称之为”throughput GC“。

**3. Parallel Old GC(-XX:+UseParallelOldGC)**

Parallel Old GC在JDK5之后出现。与parallel GC相比，唯一的区别在于针对老年代的GC算法。Parallel Old GC分为三步：标记-汇总-压缩（mark – summary – compaction）。汇总（summary）步骤与清理（sweep）的不同之处在于，其将依然幸存的对象分发到GC预先处理好的不同区域，算法相对清理来说略微复杂一点。

**4. CMS GC (-XX:+UseConcMarkSweepGC)**

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-5-Serial-GC-CMS-GC.png)

**图 5: Serial GC & CMS GC**

就像你从上图看到的那样, CMS GC比我之前解释的各种算法都要复杂很多。第一步初始化标记（initial mark） 比较简单。这一步骤只是查找那些距离类加载器最近的幸存对象。因此，停顿的时间非常短暂。在之后的并行标记（ concurrent mark ）步骤，所有被幸存对象引用的对象会被确认是否已经被追踪和校验。这一步的不同之处在于，在标记的过程中，其他的线程依然在执行。在重新标记（remark）步骤，会再次检查那些在并行标记步骤中增加或者删除的与幸存对象引用的对象。最后，在并行交换（ concurrent sweep ）步骤，转交垃圾回收过程处理。垃圾回收工作会在其他线程的执行过程中展开。一旦采取了这种GC类型，由GC导致的暂停时间会极其短暂。CMS GC也被称为低延迟GC。它经常被用在那些对于响应时间要求十分苛刻的应用之上。

当然，这种GC类型在拥有stop-the-world时间很短的优点的同时，也有如下缺点：

 它会比其他GC类型占用更多的内存和CPU

 默认情况下不支持压缩步骤

在使用这个GC类型之前你需要慎重考虑。如果因为内存碎片过多而导致压缩任务不得不执行，那么stop-the-world的时间要比其他任何GC类型都长，你需要考虑压缩任务的发生频率以及执行时间。

**5. G1 GC**

最后，我们来学习垃圾回收优先（G1）GC类型。

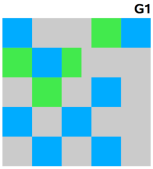
[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-6-Layout-of-G1-GC.png)

图 6:  G1 GC的结构

 如果你想要理解G1，首先你要忘记你所学过的新生代和老年代的概念。正如你在上图所看到的，每个对象被分配到不同的格子，随后GC执行。当一个区域装满之后，对象被分配到另一个区域，并执行GC。这中间不再有从新生代移动到老年代的三个步骤。这个类型是为了替代CMS GC而被创建的，因为CMS GC在长时间持续运作时会产生很多问题。

G1最大的好处是性能，他比我们在上面讨论过的任何一种GC都要快。但是在JDK 6中，他还只是一个早期试用版本。在JDK7之后才由官方正式发布。就我个人看来，NHN在将JDK 7正式投入商用之前需要很长的一段测试期（至少一年）。因此你可能需要再等一段时间。并且，我也听过几次使用了JDK 6中的G1而导致Java虚拟机宕机的事件。请耐心的等到它更稳定吧。

下一次我将讨论GC优化相关的问题，但是在此之前我要先明确一件事情，假如应用中创建的所有对象的大小和类型都是统一的，那么公司使用的WAS的GC参数可以是相同的。但是WAS所创建对象的大小和生命周期根据服务以及硬件的不同而不同。换句话说，不能因为某个应用使用的GC参数“A”，就说明同样的参数也能给其他服务带来最佳的效果。而是要因地制宜，有的放矢。我们需要找到适合每个WAS线程的参数，并且持续的监控和优化每个设备上的WAS实例。这并不是我的一家之谈，而是负责Oracle Java虚拟机研发的工程师在 JavaOne 2010上已经讨论过的。

本文中我们简略的介绍了Java的GC机制，请继续关于我们的后续文章，我们将会讨论如何监控Java GC状态以及优化GC。

另外，我特别推荐一本2011年12月发布的《Java性能》（[Amazon](http://amzn.com/0137142528),也可以通过safari在线阅读），还有在Oracle官网发布的白皮书《Java HotSpotTM虚拟机内存管理》（这本书与Java性能优化不是同一本） 作者Sangmin Lee, NHN公司，性能工程师实验室高级工程师。

**英文原文：**[cubrid](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)**，编译：**[ImportNew](http://www.importnew.com/)**-**[王晓杰](http://www.importnew.com/author/wangxiaojie)

**译文地址：**<http://www.importnew.com/1993.html>

## 成为JavaGC专家Part II — 如何监控Java垃圾回收机制 - ImportNew

http://www.importnew.com/2057.html

本文是成为Java GC专家系列文章的第二篇。在第一篇《[深入浅出Java垃圾回收机制](http://www.importnew.com/1993.html)》中我们学习了不同GC算法的执行过程，GC是如何工作的，什么是新生代和老年代，你应该了解的JDK7中的5种GC类型，以及这5种类型对于应用性能的影响。

在本文中，我将解释**JVM到底是如何执行垃圾回收处理的**。

**什么是GC监控？**

垃圾回收收集监控指的是搞清楚JVM如何执行GC的过程，例如，我们可以查明：

1.        何时一个新生代中的对象被移动到老年代时，所花费的时间。

2.       Stop-the-world 何时发生的，持续了多长时间。

GC监控是为了鉴别JVM是否在高效地执行GC，以及是否有必要进行额外的性能调优。基于以上信息，我们可以修改应用程序或者调整GC算法（GC优化）。

**如何监控GC**

有很多种方法可以监控GC，但其差别仅仅是GC操作通过何种方式展现而已。GC操作是由JVM来完成，而GC监控工具只是将JVM提供的GC信息展现给你，因此，不论你使用何种方式监控GC都将得到相同的结果。所以你也就不必去学习所有的监控GC的方法。但是因为学习每种监控方法不会占用太多时间，了解多一点可以帮助你根据不同的场景选择最为合适的方式。

下面所列的工具以及JVM参数并不适用于所有的HVM供应商。这是因为并没有关于GC信息的强制标准。本文我们将使用HotSpot JVM (Oracle JVM)。因为NHN 一直在使用Oracle (Sun) JVM，所以用它作为示例来解释我们提到的工具和JVM参数更容易些。  
首先，GC监控方法根据访问的接口不同，可以分成CUI 和GUI 两大类。CUI GC监控方法使用一个独立的叫做”jstat”的CUI应用，或者在启动JVM的时候选择JVM参数”verbosegc”。  
GUI GC监控由一个单独的图形化应用来完成，其中三个最常用的应用是”jconsole”, “jvisualvm” 和 “Visual GC”。  
下面我们来详细学习每种方法。

**jstat**

jstat 是HotSpot JVM提供的一个监控工具。其他监控工具还有jps 和jstatd。有些时候，你可能需要同时使用三种工具来监控你的应用。jstat 不仅提供GC操作的信息，还提供类装载操作的信息以及运行时编译器操作的信息。本文将只涉及jstat能够提供的信息中与监控GC操作信息相关的功能。  
jstat 被放置在$JDK\_HOME/bin。因此只要java 和 javac能执行，jstat 同样可以执行。  
你可以在命令行环境下执行如下语句。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $> jstat –gc  $<vmid$> 1000    S0C       S1C       S0U    S1U      EC         EU          OC         OU         PC         PU         YGC     YGCT    FGC      FGCT     GCT  3008.0   3072.0    0.0     1511.1   343360.0   46383.0     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   3072.0    0.0     1511.1   343360.0   47530.9     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   3072.0    0.0     1511.1   343360.0   47793.0     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588    $> |

在上图的例子中，实际的数据会按照如下列输出：

|  |  |
| --- | --- |
| 1 | S0C    S1C     S0U     S1U    EC     EU     OC     OU     PC |

vmid (虚拟机 ID)，正如其名字描述的，它是虚拟机的ID，Java应用不论运行在本地还是远程的机器都会拥有自己独立的vmid。运行在本地机器上的vmid称之为lvmid (本地vmid)，通常是PID。如果想得到PID的值你可以使用ps命令或者windows任务管理器，但我们推荐使用jps来获取，因为PID和lvmid有时会不一致。jps 通过Java PS实现，jps命令会返回vmids和main方法的信息，正如ps命令展现PIDS和进程名字那样。  
首先通过jps命令找到你要监控的Java应用的vmid，并把它作为jstat的参数。当几个WAS实例运行在同一台设备上时，如果你只使用jps命令，将只能看到启动（bootstrap）信息。我们建议在这种情况下使用ps -ef | grep java与jps配合使用。  
想要得到GC性能相关的数据需要持续不断地监控，因此在执行jstat时，要规则地输出GC监控的信息。  
例如，执行”jstat –gc 1000″ (或 1s)会每隔一秒展示GC监控数据。”jstat –gc 1000 10″会每隔1秒展现一次，且一共10次。

|  |  |
| --- | --- |
| 参数名称 | 描述 |
| gc | 输出每个堆区域的当前可用空间以及已用空间（伊甸园，幸存者等等），GC执行的总次数，GC操作累计所花费的时间。 |
| gccapactiy | 输出每个堆区域的最小空间限制（ms）/最大空间限制（mx），当前大小，每个区域之上执行GC的次数。（不输出当前已用空间以及GC执行时间）。 |
| gccause | 输出-gcutil提供的信息以及最后一次执行GC的发生原因和当前所执行的GC的发生原因 |
| gcnew | 输出新生代空间的GC性能数据 |
| gcnewcapacity | 输出新生代空间的大小的统计数据。 |
| gcold | 输出老年代空间的GC性能数据。 |
| gcoldcapacity | 输出老年代空间的大小的统计数据。 |
| gcpermcapacity | 输出持久带空间的大小的统计数据。 |
| gcutil | 输出每个堆区域使用占比，以及GC执行的总次数和GC操作所花费的事件。 |

**你可以只关心那些最常用的命令，你会经常用到 -gcutil (或-gccause), -gc and –gccapacity。**

·         -gcutil 被用于检查堆间的使用情况，GC执行的次数以及GC操作所花费的时间。

·         -gccapacity以及其他的参数可以用于检查实际分配内存的大小。

**使用-gc 参数你可以看到如下输出：**

|  |  |
| --- | --- |
| 1  2  3  4 | S0C      S1C    …   GCT  1248.0   896.0  …   1.246  1248.0   896.0  …   1.246  …        …      …   … |

不同的jstat参数输出不同类型的列，如下表所示，根据你使用的”jstat option”会输出不同列的信息。

|  |  |  |
| --- | --- | --- |
| 列 | 说明 | Jstat参数 |
| S0C | 输出Survivor0空间的大小。单位KB。 | -gc -gccapacity -gcnew -gcnewcapacity |
| S1C | 输出Survivor1空间的大小。单位KB。 | -gc -gccapacity -gcnew -gcnewcapacity |
| S0U | 输出Survivor0已用空间的大小。单位KB。 | -gc -gcnew |
| S1U | 输出Survivor1已用空间的大小。单位KB。 | -gc -gcnew |
| EC | 输出Eden空间的大小。单位KB。 | -gc -gccapacity -gcnew -gcnewcapacity |
| EU | 输出Eden已用空间的大小。单位KB。 | -gc -gcnew |
| OC | 输出老年代空间的大小。单位KB。 | -gc -gccapacity -gcold -gcoldcapacity |
| OU | 输出老年代已用空间的大小。单位KB。 | -gc -gcold |
| PC | 输出持久代空间的大小。单位KB。 | -gc -gccapacity -gcold -gcoldcapacity -gcpermcapacity |
| PU | 输出持久代已用空间的大小。单位KB。 | -gc -gcold |
| YGC | 新生代空间GC时间发生的次数。 | -gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| YGCT | 新生代GC处理花费的时间。 | -gc -gcnew -gcutil -gccause |
| FGC | full GC发生的次数。 | -gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| FGCT | full GC操作花费的时间 | -gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| GCT | GC操作花费的总时间。 | -gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| NGCMN | 新生代最小空间容量，单位KB。 | -gccapacity -gcnewcapacity |
| NGCMX | 新生代最大空间容量，单位KB。 | -gccapacity -gcnewcapacity |
| NGC | 新生代当前空间容量，单位KB。 | -gccapacity -gcnewcapacity |
| OGCMN | 老年代最小空间容量，单位KB。 | -gccapacity -gcoldcapacity |
| OGCMX | 老年代最大空间容量，单位KB。 | -gccapacity -gcoldcapacity |
| OGC | 老年代当前空间容量制，单位KB。 | -gccapacity -gcoldcapacity |
| PGCMN | 持久代最小空间容量，单位KB。 | -gccapacity -gcpermcapacity |
| PGCMX | 持久代最大空间容量，单位KB。 | -gccapacity -gcpermcapacity |
| PGC | 持久代当前空间容量，单位KB。 | -gccapacity -gcpermcapacity |
| PC | 持久代当前空间大小，单位KB | -gccapacity -gcpermcapacity |
| PU | 持久代当前已用空间大小，单位KB | -gc -gcold |
| LGCC | 最后一次GC发生的原因 | -gccause |
| GCC | 当前GC发生的原因 | -gccause |
| TT | 老年化阈值。被移动到老年代之前，在新生代空存活的次数。 | -gcnew |
| MTT | 最大老年化阈值。被移动到老年代之前，在新生代空存活的次数。 | -gcnew |
| DSS | 幸存者区所需空间大小，单位KB。 | -gcnew |

jstat 的好处是它可以持续的监控GC操作数据，不论Java应用是运行在本地还是远程，只要有控制台的地方就可以使用。当使用–gcutil 会输出如下信息。在GC优化的时候，你需要特别注意YGC, YGCT, FGC, FGCT 和GCT。

|  |  |
| --- | --- |
| 1  2  3  4 | S0      S1       E        O        P        YGC    YGCT     FGC    FGCT     GCT  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995 |

这些信息很重要，因为它们展示了GC处理到底花费了多少时间。  
在这个例子中，YGC 是217而YGCT 是0.928，这样在简单的计算数据平均数后，你可以知道每次新生代的GC大概需要4ms（0.004秒），而full GC的平均时间为33ms。  
但是，只看数据平均数经常无法分析出真正的GC问题。这是主要是因为GC操作时间严重的偏差（换句话说，假如两次full GC的时间是 67ms，那么其中的一次full GC可能执行了10ms而另一个可能执行了57ms。）为了更好地检测每次GC处理时间，最好使用 –verbosegc来替代数据平均数。

**-verbosegc**

-verbosegc 是在启动一个Java应用时可以指定的JVM参数之一。而jstat 可以监控任何JVM应用，即便它没有制定任何参数。 -verbosegc 需要在启动的时候指定，因此你可能会认为它没有必要（因为jstat可以替代之）。但是， -verbosegc 会以更浅显易懂的方式展现GC发生的结果，因此他对于监控监控GC信息十分有用。

|  |  |  |
| --- | --- | --- |
|  | jstat | -verbosegc |
| 监控对象 | 运行在本机的Java应用可以把日志输出到终端上，或者借助jstatd命令通过网络连接远程的Java应用。 | 只有那些把-verbogc作为启动参数的JVM。 |
| 输出信息 | 堆状态（已用空间，最大限制，GC执行次数/时间，等等） | 执行GC前后新生代和老年代空间大小，GC执行时间。 |
| 输出时间 | Every designated time 每次设定好的时间。 | 每次GC发生的时候。 |
| 何时有用。 | 当你试图观察堆空间变化情况 | 当你试图了解单次GC产生的效果。 |

下面是-verbosegc 的可用参数  
· -XX:+PrintGCDetails  
· -XX:+PrintGCTimeStamps  
· -XX:+PrintHeapAtGC  
· -XX:+PrintGCDateStamps (from JDK 6 update 4)

如果只是用了 -verbosegc 。那么默认会加上 -XX:+PrintGCDetails。 –verbosgc 的附加参数并不是独立的。而是经常组合起来使用。  
使用 –verbosegc后，每次GC发生你都会看到如下格式的结果。

[GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs] <starting occupancy3> -> <ending occupancy3>, <pause time3> secs]

|  |  |
| --- | --- |
| 收集器 | minor gc使用的收集器的名字。 |
| starting occupancy1 | GC执行前新生代空间大小。 |
| ending occupancy1 | GC执行后新生代空间大小。 |
| pause time1 | 因为执行minor GC，Java应用暂停的时间。 |
| starting occupancy3 | GC执行前堆区域总大小 |
| ending occupancy3 | GC执行后堆区域总大小 |
| pause time3 | Java应用由于执行堆空间GC（包括major GC）而停止的时间。 |

这是-verbosegc 输出的minor GC的例子。

|  |  |
| --- | --- |
| 1  2  3  4 | S0    S1     E      O      P        YGC    YGCT    FGC    FGCT     GCT  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995 |

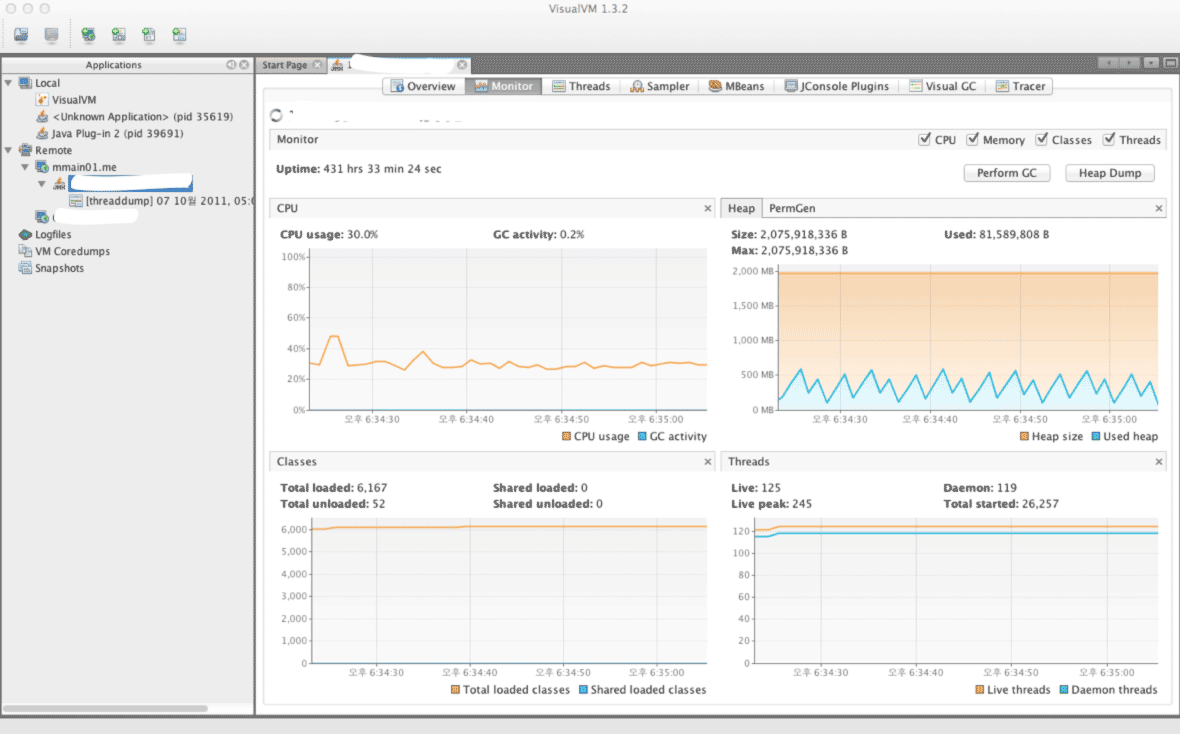
这是 Full GC发生时的例子

|  |  |
| --- | --- |
| 1 | [Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm : 10756K->10756K(12288K)], 0.1762129 secs] [Times: user=0.19 sys=0.00, real=0.19 secs] |

如果使用了 CMS collector，那么如下CMS信息也会被输出。  
由于 –verbosegc 参数在每次GC事件发生的时候都会输出日志，我们可以很轻易地观察到GC操作对于堆空间的影响。

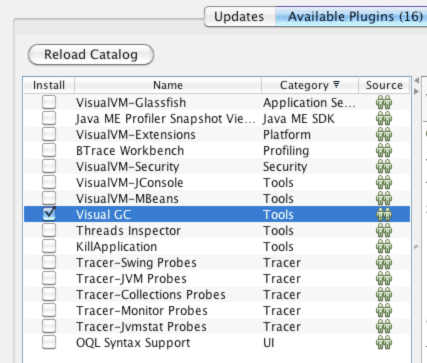
**(Java) VisualVM  + Visual GC**

Java Visual VM是由Oracle JDK提供的图形化的汇总和监控工具。

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-1-VisualVM-Screenshot..png)

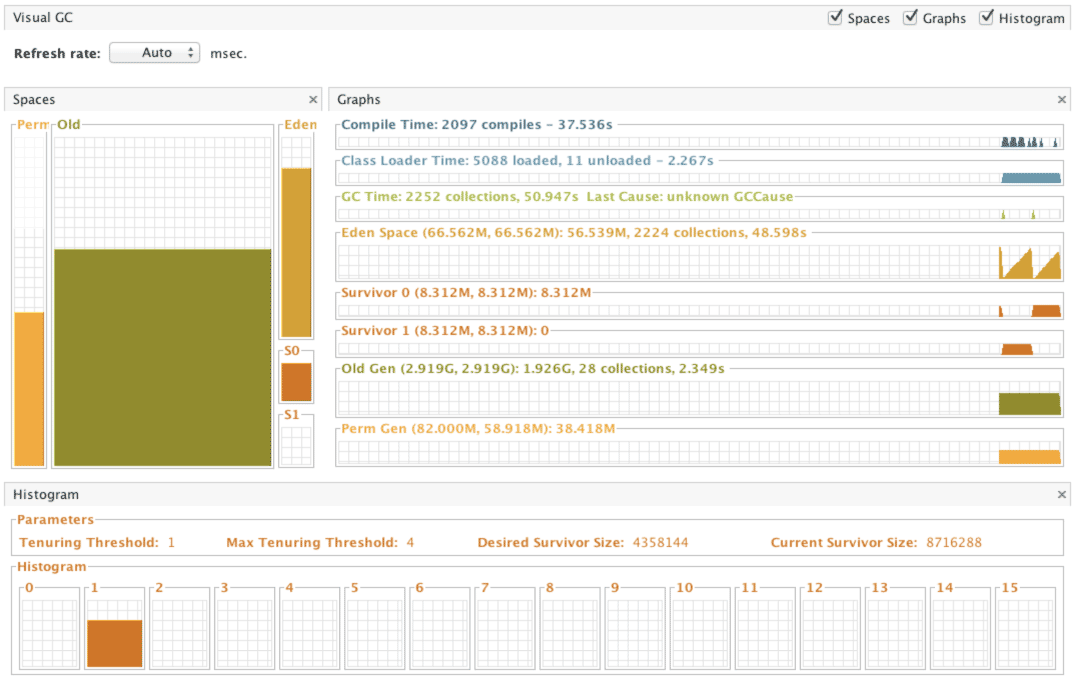
**图1: VisualVM 截图**

除了JDK中自带的版本，你还可以直接从官网下载Visual VM。出于便利性的考虑，JDK中包含的版本被命名为Java VisualVM (jvisualvm),而官网提供的版本被命名为Visual VM (visualvm)。两者的功能基本相同，只有一些细小的差别，例如安装组件的时候。就个人而言，我更喜欢可以从官网下载的Visual VM。

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-2-Viusal-GC-Installation-Screenshot.png)

**图 2: Viusal GC 安装截图**

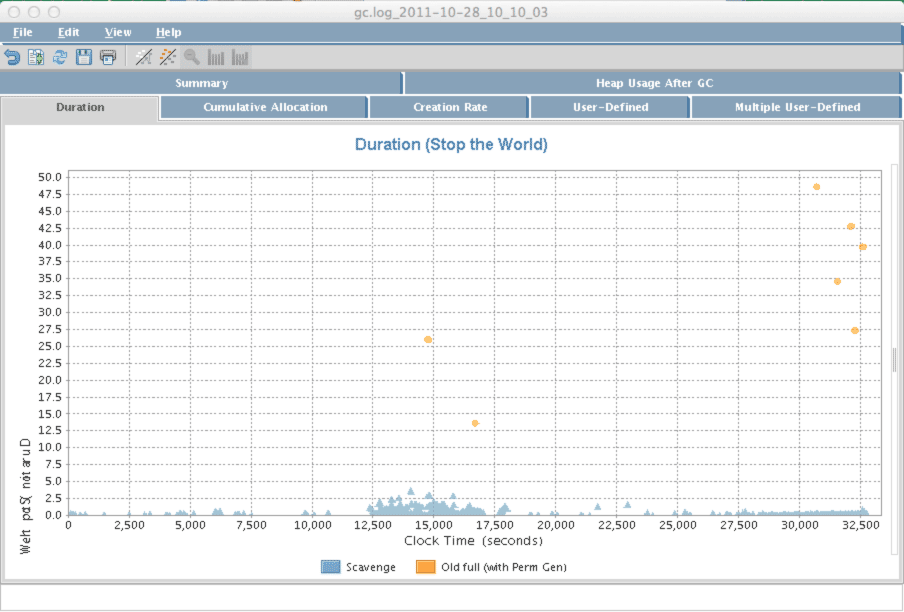
**通过Visual GC，你可以更直观的看到执行jstatd 所得到的信息。**

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-3-Visual-GC-execution-screenshot.png)

**图3: Visual GC 执行截图**

**HPJMeter**

[HPJMeter](https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPJMETER) 可以很方便的分析 -verbosegc 输出的结果，如果Visual GC可以视作jstat的图形化版本，那么HPJMeter就相当于 –verbosgc的图形化版本。当然，GC分析只是HPJMeter提供的众多功能之一，HPJMeter是由惠普开发的性能监控工具，他可以支持HP-UX，Linux以及MS Windows。  
起初，一个成为HPTune 被设计用来图形化的分析-verbosegc.输出的结果。但是，随着HPTune的功能被集成到HPJMeter 3.0版本之后，就没有必要单独下载HPTune了。但运行一个应用时， -verbosegc 的结果会被输出到一个独立的文件中。  
你可以用HPJMeter直接打开这个文件，以便更直观的分析GC性能数据。

[](http://www.importnew.com/wp-content/uploads/2012/12/Figure-4-HPJMeter.png)

 图**4: HPJMeter**

**下次预告**

本文我们主要讲述了如果监控GC操作信息，这将是GC优化的前提。就我个人经验而言，我推荐使用jstat 来监控GC操作，如果你感觉到GC操作的执行时间过长，那就可以使用verbosegc 参数来分析GC。GC优化的大体步骤就是在添加verbosegc 参数后，调整GC参数，分析修改后的结果。在下一篇文章中，我们将通过真实的例子来讲解优化GC的最佳选择。  
作者Sangmin Lee, NHN公司，性能工程师实验室高级工程师。

## 成为Java GC专家（3）—如何优化Java垃圾回收机制 - ImportNew

http://www.importnew.com/3146.html

本文是成为Java GC专家系列文章的第三篇。在第一篇《[成为JavaGC专家Part I — 深入浅出Java垃圾回收机制](http://www.importnew.com/1993.html)》中我们学习了不同GC算法的执行过程，GC是如何工作的，什么是新生代和老年代，你应该了解的JDK7中的5种GC类型，以及这5种类型对于应用性能的影响。

在第二篇《[成为JavaGC专家Part II — 如何监控Java垃圾回收机制](http://www.importnew.com/2057.html)》，我解释了JVM实际上是如何执行垃圾回收的，我们如何监控GC，以及那哪些具可以让我们的工作更快，更高效。在第三篇文章中，我们会基于实际的例子来解释一些优化GC的最佳实践。我认为在阅读本篇文章之前，你已经很好地理解了之前的文章，因此，为了你能够更好地学习本文，如果你还没有读过之前的两篇文章话，请先阅读。

**为什么需要优化GC**

或者说的更确切一些，对于基于Java的服务，是否有必要优化GC？应该说，对于所有的基于Java的服务，并不总是需要进行GC优化，但前提是所运行的基于Java的系统，包含了如下参数或行为：

已经通过 -Xms 和–Xmx 设置了内存大小

包含了 -server 参数

系统中没有超时日志等错误日志

**换句话说，如果你没有设定内存的大小，并且系统充斥着大量的超时日志时，你就需要在你的系统中进行GC优化了。**

但是，你需要时刻铭记一条：GC优化永远是最后一项任务。

想一下进行GC优化的最根本原因，垃圾收集器清除在Java程序中创建的对象，GC执行的次数即需要被垃圾收集器清理的对象个数，与创建对象的数量成正比，因此，首先你应该减少创建对象的数量。

俗话说的好，“冰冻三尺非一日之寒”。我们应该从小事做起，否则日积月累就会很难管理。

我们需要使用StringBuilder 或者StringBuffer 来替代String

应该尽量少的输出日志

但是，我们知道有些情况会让我们束手无策，我们眼睁睁的看着XML以及JSON解析占用了大量的内存。即便我们已经尽可能少的使用String以及尽量少的输出日志，大量的临时内存被用于XML或者JSON解析，例如10-100MB。但是，舍弃XML和JSON是很难的。我们只要知道，他会占用很多内存。

如果应用内存使用量经过几次重复调整之后有所改善，你就可以开始GC优化了。

我为GC优化归纳了两个目的：

一个是将转移到老年代的对象数量降到最少

另一个是减少Full GC的执行时间

**将转移到老年代的对象数量降到最少**

按代的GC机制由Oracle JVM提供，不包括可以在JDK7以及更高版本中使用的G1 GC。换句话说，对象被创建在伊甸园空间，而后转化到幸存者空间，最终剩余的对象被送到老年代。某些比较大的对象会在被创建在伊甸园空间后，直接转移到老年代空间。老年代空间上的GC处理会新生代花费更多的时间。因此，减少被移到老年代对象的数据可以显著地减少Full GC的频率。减少被移到老年代空间的对象的数量，可能被误解为将对象留在新生代。但是，这是不可能的。取而代之，你可以调整新生代空间的大小。

**减少Full GC执行时间**

Full GC的执行时间比Minor GC要长很多。因此，如果Full GC花费了太多的时间（超过1秒），一些连接的部分可能会发生超时错误。

如果你试图通过消减老年代空间来减少Full GC的执行时间，可能会导致OutOfMemoryError 或者 Full GC执行的次数会增加。

与之相反，如果你试图通过增加老年代空间来减少Full GC执行次数，执行时间会增加。

因此，你需要将老年代空间设定为一个“合适”的值。

**影响GC性能的参数**

正如我们在第二篇文章结尾提到的，不要幻想“某个人设定了GC参数后性能得到极大的提高，我们为什么不和他用一样的参数？”，因为不同的Web服务所创建对象的大小和他们的生命周期都不尽相同。

简单来说，如果一个任务的执行条件是A，B，C，D和E，同样的任务执行条件换为A和B，你会觉得哪个更快？从一般人的直觉来看，在A和B条件下执行的任务会更快。

Java GC参数也是相同的道理，设定一些参数不但没有提高GC执行速度，反而可能导致他更慢。GC优化的最基本原则是将不同的GC参数用于2台或者多台服务器，并进行对比，并将那些被证明提高了性能或者减少了GC执行时间的参数应用于服务器。请谨记这一点。

下面这个表格列出了GC参数中与内存大小相关的，可以影响性能的参数。

**表1：GC优化需要考虑的Java参数**

| 定义 | 参数 | 描述 |
| --- | --- | --- |
| 堆内存空间 | -Xms | Heap area size when starting JVM  启动JVM时的堆内存空间。 |
|  | -Xmx | Maximum heap area size  堆内存最大限制 |
| 新生代空间 | -XX:NewRatio | Ratio of New area and Old area  新生代和老年代的占比 |
|  | -XX:NewSize | New area size  新生代空间 |
|  | -XX:SurvivorRatio | Ratio ofEdenarea and Survivor area  伊甸园空间和幸存者空间的占比 |

我在进行GC优化时经常使用-Xms，-Xmx和-XX:NewRatio。-Xms和-Xmx是必须的。你如何设定NewRatio 会对GC性能产生十分显著的影响。有些人可能会问如何设定Perm区域的大小？你可以通过-XX:PermSize 和-XX:MaxPermSize参数来设定，

当OutOfMemoryError 错误发生并且是由于Perm空间不足导致时，另一个可能影响GC性能的参数是GC类型。下表列出了所有可选的GC类型（基于JDK6.0）

**表2：GC类型可选参数**

| 分类 | 参数 | 备考 |
| --- | --- | --- |
| Serial GC | -XX:+UseSerialGC |  |
| Parallel GC | -XX:+UseParallelGC -XX:ParallelGCThreads=value |  |
| Parallel Compacting GC | -XX:+UseParallelOldGC |  |
| CMS GC | -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=value -XX:+UseCMSInitiatingOccupancyOnly |  |
| G1 | -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC | 在JDK6中这两个参数必须同时使用 |

除了G1 GC，可以通过每种类型第一行的参数来切换GC类型。最常用的GC类型是Serial GC。他专门针对客户端系统进行了优化。

影响GC性能的参数有很多，但是上面提到的参数会带来最显著的效果。请牢记，设定过多的参数不一定会减少GC执行时间。

**GC优化过程**

GC优化的过程与大多数性能改善的过程及其类似。下面是我使用的GC优化过程。

**1.监控GC状态**

首先你需要监控GC来检查在系统执行过程中GC的各种状态。请参考前一篇文章中提到的监控方法 [成为JavaGC专家Part II — 如何监控Java垃圾回收机制](http://www.importnew.com/2057.html)。

**2.在分析监控结果后，决定是否进行GC优化**

在检查GC状态的过程中，你应该分析监控结果以便决定是否进行GC优化，如果分析结果表明执行GC的时间只有0.1-0.3秒，那你就没必要浪费时间去进行GC优化。但是，如果GC的执行时间是1-3秒，或者超过10秒，GC将势在必行。

但是，如果你已经为Java分配了10GB的内存，并且不能再减少内存大小，你将无法再对GC进行优化。在进行GC优化之前，你必须想清楚你为什么要分配如此大的内存空间。假如当你分1 GB 或 2 GB内存时出现OutOfMemoryError ，你应该执行堆内存转储（heap dump），并消除隐患。

**注意：**

堆内存转储是一个用来检查Java内存中的对象和数据的文件。该文件可以通过执行JDK中的jmap命令来创建。在创建文件的过程中，Java程序会暂停，因此不要再系统执行过程中创建该文件。

你可以在互联网上搜索堆内存[[s1]](file:///C:\Users\Administrator\Desktop\2013.02.16%20%E6%88%90%E4%B8%BAJava%20GC%E4%B8%93%E5%AE%B6%E7%B3%BB%E5%88%97%E4%B9%8B%E4%B8%89--%E5%A6%82%E4%BD%95%E4%BC%98%E5%8C%96Java%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6%E6%9C%BA%E5%88%B6%20%E5%88%9D%E7%A8%BF%20by%20%E7%8E%8B%E6%99%93%E6%9D%B0.doc#_msocom_1) 转储的详细说明。对于韩国的读者，可以参考我去年发布的书： [The story of troubleshooting for Java developers and system operators](http://book.naver.com/bookdb/book_detail.nhn?bid=6654751) (Sangmin Lee, Hanbit Media, 2011, 416 pages)。

**3. 调整GC类型/内存空间**

如果你已经决定要进行GC优化，那么就要选择GC类型和设定内存空间。在这时，如果你有几台不同服务器，请时刻牢记，检查每一台服务器的GC参数，并进行有针对性的优化。

**4.分析结果**

在调整了GC参数并持续收集24小时之后，开始对结果进行分析，如果你幸运的话，你就找到那些最适合系统的GC参数。反之，你需要通过分析日志来检查内存是如何被分配的。然后你需要通过不断的调整GC类型和内存空间大小一边找到最佳的参数。

**5. 如果结果令人满意，你可以将该参数应用于所有的服务器，并停止GC优化**

有过GC优化结果令人满意，你可以应用于所有的服务器，下面的章节中，我们将看到每个步骤的具体任务。

**监控GC状态及分析结果**

查看运行中的Web Application Server (WAS)的GC状态的最佳方法是通过jstat命令，在第二篇文章[成为JavaGC专家Part II — 如何监控Java垃圾回收机制](http://www.importnew.com/2057.html)中我已经详细解释过jstat命令，因此本篇文章我将重点描述数据部分。

下面这个例子展现了某个JVM在进行GC优化之前的状态。

（很遗憾，这不是一个操作服务器）

|  |  |
| --- | --- |
| 1  2  3  4 | $ jstat -gcutil 21719 1s  S0    S1    E    O    P    YGC    YGCT    FGC    FGCT GCT  48.66 0.00 48.10 49.70 77.45 3428 172.623 3 59.050 231.673  48.66 0.00 48.10 49.70 77.45 3428 172.623 3 59.050 231.673 |

如上表，我们先看一下YGC 和YGCT，计算YGCT/ YGC得到0.050秒（50毫秒）。这意味着新生代空间上的GC操作平均花费50毫秒。在这种情况，你大可不必担心新生代空间上执行的GC操作。  
接下来，我们来看一下FGCT 和FGC。，计算FGCT/ FGC得到19.68秒，这意味着GC的平均执行时间为19.68秒，可能是每次花费19.68秒执行了三次，也可能是其中的两次执行了1秒而另一次执行了58秒。不论哪种情况，都需要进行GC优化。

通过jstat 命令可以很轻易地查看GC状态，但是，分析GC的最佳方式是通过–verbosegc参数来生成日志，在之前的文章中我已经解释了如何分析这些日志，HPJMeter 是我个人最喜欢的用于分析-verbosegc 日志的工具。他很易于使用和分析结果。通过HPJmeter你可以很轻易查看GC执行时间以及GC发生频率。如果GC执行时间满足下面所有的条件，就意味着无需进行GC优化了。

Minor GC执行的很快（小于50ms）

Minor GC执行的并不频繁（大概10秒一次）

Full GC执行的很快（小于1s）

Full GC执行的并不频繁（10分钟一次）

上面提到的数字并不是绝对的；他们根据服务状态的不同而有所区别，某些服务可能满足于Full GC每次0.9秒的速度，但另一些可能不是。因此，针对不同的服务设定不同的值以决定是否进行GC优化。

在查看GC状态的时候有件事你需要特别注意，那就是不要只关注Minor GC 和Full GC的执行时间。还要关注**GC执行的次数**,例如，当新生代空间较小时，Minor GC会过于频繁的执行（有时每秒超过1次）。另外，转移到老年代的对象数增多，则会导致Full GC执行次数增多。因此，别忘了加上–gccapacity参数来查看具体占用了多少空间。

**设定GC类型/内存空间大小**

* **设定GC类型**

OracleJVM有5种GC类型，但是在JDK7之前的版本中，只能在Parallel GC, Parallel Compacting GC 和CMS GC之中选择一个，对于选择哪个没有明确的原则和规则。

这样的话，我们该如何选择呢？强烈建议三者都选，但是，有一点是很明确的：CMS GC比Parallel GCs更快。如果真的如此，那么就选CMS GC了。但是，CMS GC也不总是更快。整体来看，CMS GC模式下的Full GC执行更快，不过，一旦出现并行模式失败，他将比Parallel GC更慢。

**并发模式失败**

我们来详细讲解一下并发模式失败。

Parallel GC 和 CMS GC 最大的不同来自于压缩任务。压缩任务是通过删除已分配内存空间中的空白空间以便压缩内存，清理内存碎片。

在Parallel GC模式下，压缩工作在Full GC执行时进行，这会费很多时间，但是，在执行完Full GC之后，由于能够顺序地分配空间，随后的内存能够被更快的分配。

与之相反的，CMS GC并不进行压缩处理，因此，CMS GC执行的更快。但是，由于没有压缩，在进行磁盘清理之前，内存中会有很多空白空间。这就是说，可能没有足够的空间存储大的对象，例如，虽然老年代空间还有300MB空间，但是一些10MB的对象无法被顺序的存储。在这种情况下，会出现“并行模式失败”警告，并执行压缩处理。在CMS GC模式下，压缩处理的执行时间要比Parallel GCs长很多。另外，这还将导致另外一个问题。关于并发模式失败的详细说明，可以参考Oracle工程师撰写的[Understanding CMS GC Logs](https://blogs.oracle.com/poonam/entry/understanding_cms_gc_logs)。

综上所述，你需要找到最适合你的系统的GC类型。

每个系统都有最适合他的GC类型等着你去寻找，如果你有6台服务器。我建议你每两台设置相同的参数。并添加 –verbosegc参数，分析结果。

* **设定内存空间大小**

下表展示了内存空间大小，GC执行次数以及GC执行时间三者间的关系。

大内存空间

减小GC执行次数

增加GC执行时间

小内存空间

减小GC执行时间

增加GC执行次数

关于如何设置内存空间的大小，没有唯一的标准答案。如果服务器资源足够，而且Full GC也可能在1秒内完成，设置为10GB当然可行。。但绝大多数服务器并不是这样，当内存设为10GB时，可能要花费10~30秒来执行Full GC。当然，执行时间会随对象的大小而改变。

鉴于如此，我们应该如何设定内存空间大小呢？一般来说，我建议为500MB。不过请注意这不是让你将WAS的内存参数设置为–Xms500m 和–Xmx500m。根据优化GC之前的状态，如果Full GC执行之后内存空间剩余300MB，那么最好将内存设置为1GB（300MB（默认程序占用）+ 500MB（老年代最小空间）+200MB（空闲内存））。也就是说你要为老年代额外设置500MB。因此，如果你有三个执行服务器，内存分别设置为1GB，1.5GB，2GB，并且检查结果。

理论上来讲，GC执行速度应该遵循1GB> 1.5GB> 2GB,因此1GB执行GC速度最快。但是并不说明1GB空间的Full GC会花费1秒而2GB空间会花费2秒。时间取决于服务器的性能和对象的大小。因此，最佳的方式是建立尽可能多的衡量指标来监控他们。

对于内存空间大小，你应该额外设定NewRatio参数。NewRatio参数是新生代和老年代空间的比例，即XX:NewRatio=1意味着新生代与老年代之比为1:1。对于1GB来说就是新生代和老年代各500MB。如果NewRatio为2，意味着新生代老年代之比为1:2，因此该值越大，老年代空间越大，新生代空间越小。

这看似一件不是很重要的事情，但NewRatio参数会显著地影响整个GC的性能。如果新生代空间很小，会用更多的对象被转移到老年代空间，这样导致频繁的Full GC，增加暂停时间。

你可以简单的认为NewRatio 为1是最佳的选择，但是，有时可能设置为2或3更好，我就见过很多这样的例子。

**如何最快的完成GC优化？**

对比性能测试的结果应该是最快地方法，为每一台服务器设置不同的参数并监控他们的状态，强烈建议至少监控1或2天的数据。但是，当你对GC优化是，你要确保每次执行相同的负载。并且请求的比率，例如URL都应该是一致的。不过，即便对于专业测试人员要想精确的控制负载也是很难的，并要花费大量的时间准备。因此，相对来说比较方便和容易的方法是调整才参数，之后花费较长的时间收集结果。

**分析GC优化结果**

在设置了GC参数以及-verbosegc参数之后，通过tail命令确保日志被正确的生成。如果参数设置的不正确或者日志没有生成，你将白白浪费你的时间。如果日志正确的话，持续收集1到2天。随后最好将日志下载到本地PC并用HPJMeter来分析

Full GC 执行时间

Minor GC执行时间

Full GC 执行间隔

Minor GC 执行间隔

Entire Full GC 执行时间

Entire Minor GC 执行时间

Entire GC 执行时间

Full GC e执行时间

Minor GC 执行时间

找到最佳的GC参数是件非常幸运的事情，然而在大多数场合，我们并不会得到幸运之神的眷顾，在进行GC优化时要尽量小心谨慎，想一步完成优化往往会导致OutOfMemoryError 。

**优化示例**

好了，我们一直在纸上谈兵，现在我们看一些实际的GC优化的例子。

**示例1**

下面这个例子针对 **Service S**的优化,对于最近被部署的 Service S，Full GC花费了太长的时间。

请看 jstat –gcutil的执行结果。

|  |  |
| --- | --- |
| 1  2 | S0 S1 E O P YGC YGCT FGC FGCT GCT  12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993 |

最左边的**Perm** 空间对于最初的GC优化不是很重要，这一次YGC参数的值更加有用。

Minor GC和Full GC的平均值如下表所示

**表3：Service S的Minor GC 和Full GC的平均执行时间**

| GC 类型 | GC 执行次数 | GC 执行时间 | 平均 |
| --- | --- | --- | --- |
| Minor GC | 54 | 2.047 | 37 ms |
| Full GC | 5 | 6.946 | 1,389 s |

最重要的是下面两个数据

* 新生代实际使用空间: 212,992 KB
* 老年代实际使用空间: 1,884,160 KB

因此，总的内存空间为2GB,不算Perm空间的话，新生代与老年代之比为1:9。通过**jstat**和-verbosegc 日志进行数据收集，并把三台服务器按照如下方式设置。

* NewRatio=2
* NewRatio=3
* NewRatio=4

一天之后，检查系统的GC日志后发现，在设置了NewRatio参数后很幸运的没有发生Full GC，

**为什么？**

* NewRatio=2: 45 ms
* NewRatio=3: 34 ms
* NewRatio=4: 30 ms

我们看到NewRatio=4 是最佳的参数，虽然它的新生代空间最小，但GC时间确最短。设定这个参数之后，系统没有执行过Full GC。

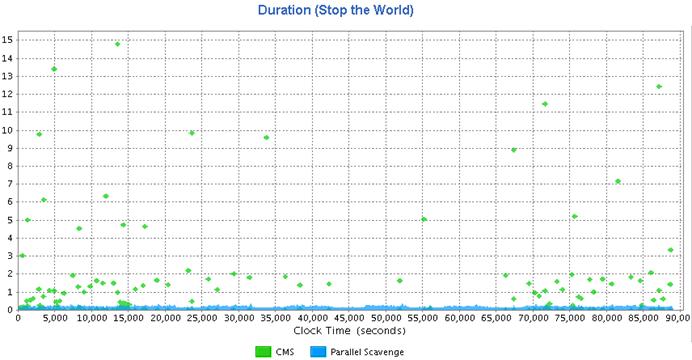
为了说明这个问题，下面是服务之星一段时间后执行jstat –gcutil的结果

|  |  |
| --- | --- |
| 1  2 | S0 S1 E O P YGC YGCT FGC FGCT GCT  8.61 0.00 30.67 24.62 22.38 2424 30.219 0 0.000 30.219 |

你可能会认为因为服务器接受的请求少才导致的GC执行频率下降。实际上，虽然Full GC没有执行，但是Minor GC被执行了 2,424次。

**示例2**

这是一个针对ServiceA的例子，我们通过公司内部的应用性能管理系统（APM）发现JVM暂停了相当长的时间（超过8秒），因此我们进行了GC优化。我们找到了Full GC执行时间过长的原因，并着手解决。

进行GC优化的第一步，就是我们添加了-verbosegc参数，并得到如下结果。  
[](http://www.importnew.com/wp-content/uploads/2013/02/13.jpg)

**图1：进行GC优化之前的STW时间**

如上图所示，由HPJMeter自动生成的图片之一。X坐标表示JVM执行的时间。Y坐标表示每次GC的时间。CMS绿点，表示Full GC结果。Parallel Scavenge蓝点，表示Minor GC结果。

之前我曾经说过CMS GC是最快的，但是上面的的结果显示出于某种原因，它最多花费了15秒。**是什么导致这个结果？**是否想起我之前提过的，CMS在进行内存清理时，会变慢。与此同时，服务的内存被设定为 –Xms1g和–Xmx4g ，且实际分配了4GB内存。

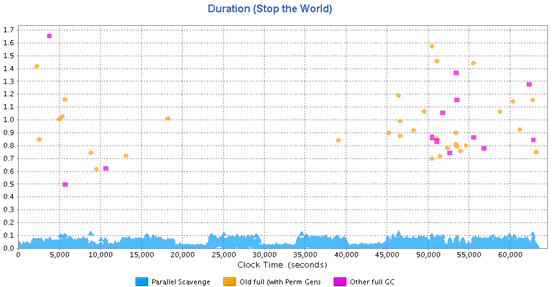
因此，我将GC类型从CMS改为Parallel GC。并且将内存改为2GB，设定NewRatio 为3。几小时之后我使用 jstat –gcutil得到如下结果

|  |  |
| --- | --- |
| 1  2 | S0 S1 E O P YGC YGCT FGC FGCT GCT  0.00 30.48 3.31 26.54 37.01 226 11.131 4 11.758 22.890 |

相对于4GB时的15秒，Full GC变成了平均每次3秒。但是3秒一样比较慢，因此我设计了如下6种场景。

* Case 1: -XX:+UseParallelGC -Xms1536m -Xmx1536m -XX:NewRatio=2
* Case 2: -XX:+UseParallelGC -Xms1536m -Xmx1536m -XX:NewRatio=3
* Case 3: -XX:+UseParallelGC -Xms1g -Xmx1g -XX:NewRatio=3
* Case 4: -XX:+UseParallelOldGC -Xms1536m -Xmx1536m -XX:NewRatio=2
* Case 5: -XX:+UseParallelOldGC -Xms1536m -Xmx1536m -XX:NewRatio=3
* Case 6: -XX:+UseParallelOldGC -Xms1g -Xmx1g -XX:NewRatio=3

**那一个最快呢？**结果显示，内存越小，结果越好。下图展示了Case6的结果。这是GC的性能最好。最长的响应时间只有1.7秒。平均时间在1秒之内。

[](http://www.importnew.com/wp-content/uploads/2013/02/2.png)

**图2：Case6的时间图表**

基于以上结果。我们按照Case6调整了GC参数。但是，这导致了每天晚上都会发生OutOfMemoryError。在这里很难解释具体的原因。简单来说，批处理程序导致了内存泄漏。相关的问题已经被解决。

如果对GC日志只分析很短的时间就贸然对所有服务器进行优化是非常危险的。请时刻牢记，你必须同时分析GC日志和应用程序。

我们回顾了两个关于GC优化的例子，正如我之前提到的，例子中提到的GC参数，可以设置在相同的服务器之上，但前提是他们具有相同的CPU，操作系统，JDK版本以及运行着相同的服务。但是不要直接把我用过的参数用到你的服务至上，它们未必能很好的工作。

**结论**

我凭借经验进行GC优化，而没有执行堆转储并分析内存的详细内容。精确地分析内存可以得到更好的优化效果。但是，这种分析一般适用于内存使用量相对固定的场合。不过，如果服务严重过载并占用的大量的内存，强力建议根据之前的经验进行GC优化。

我已经在一些服务上设置了G1 GC参数，并进行过性能测试。但还没有应用与正式环境，G1 GC参数的速度要快于其他任何GC类型。但是，你必须要升级到JDK7。另外，他的稳定性也暂时没有保障，没人知道是否会出现致命的错误。因此还不到将其正式应用的时候

在未来的某一天，等到JDK7真正稳定了（这不是说他现在不稳定），并且WAS针对JDK7进行优化后，G1 GC最终能够按照预期的那样工作了，我们可能就不需要在进行GC优化了。

想了解GC优化的更多内容，请登录[Slideshare.com](http://www.slideshare.net/) 查看关联资源。强烈推荐[Everything I Ever Learned About JVM Performance Tuning @Twitter](http://www.slideshare.net/aszegedi/everything-i-ever-learned-about-jvm-performance-tuning-twitter)。作者Attila Szegedi，一位Twitter工程师。请花些时间阅读。

By Sangmin Lee, NHN Performance Engineering Lab.

作者Sangmin Lee，就职于NHN性能工程研究院

**英文原文：**[cubrid](http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/)**，编译：**[ImportNew](http://www.importnew.com/)**-**[王晓杰](http://www.importnew.com/author/wangxiaojie)

**译文地址：**<http://www.importnew.com/3146.html>

【如需转载，请在正文中标注并保留原文链接、译文链接和译者等信息，谢谢合作！】

## 成为Java GC专家（4）—Apache的MaxClients参数详解及其在Tomcat执行FullGC时的影响 - ImportNew

http://www.importnew.com/3151.html

这是“成为Java GC专家系列文章”的第四篇。

在第一篇文章 [成为JavaGC专家Part I — 深入浅出Java垃圾回收机制](http://www.importnew.com/1993.html) 中我们学习了不同GC算法的执行过程，GC如何工作，新生代及老年代的基本概念，在JDK7中你应该了解的5种GC类型以及他们的性能如何。

在第二篇文章 [成为JavaGC专家Part II — 如何监控Java垃圾回收机制](http://www.importnew.com/2057.html) 中我们学到了JVM到底是如何执行垃圾回收，我们如何监控GC，以及那些工具可以使得监控过程更高效。

在第三篇文章 [成为Java GC专家系列Part III–如何优化Java垃圾回收机制](http://www.importnew.com/3146.html)中我们通过实际的例子学到了一些可以优化GC的参数。同时我们讲解了如何减少对象被转移到老年代空间，如何缩短Full GC时间，以及如何设置GC类型及内存空间。

在第四篇文章中，我们将阐述Apache中MaxClients 参数的重要性，以及他如何在GC发生时，显著地影响整个系统的性能。我将提供几个例子以方便你理解MaxClients 导致的问题。同时我还会说明如何根据系统的内存情况，设置最佳的MaxClients参数值。

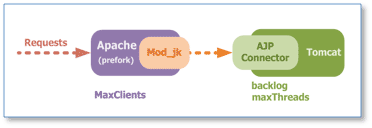
**MaxClients对于系统的影响**

[NHN](http://www.cubrid.org/blog/tags/NHN/) (译者注：NHN是作者工作的公司)服务的执行环境中存在一组[Throttle valve](http://en.wikipedia.org/wiki/Throttle)-type参数（译者注：节流阀参数，用于控制系统负载）。这些参数对于系统来说十分重要。下面我们看一下Apache的 MaxClients 参数在[Full GC](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/) 发生时是如何影响系统的。

大部分开发人员都知道在由于GC发生而导致的”停止世界现象(STW) “（详细请参见[Understanding Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)）。尤其是，NHN的Java开发人员经常会遇到由于GC原因导致的Tomcat报错。由于[Java 虚拟机](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/) (JVM)管理着内存，以Java为基础的程序无法摆脱GC导致的STW现象。假如在某一个时间，当你正在操作你开发的应用时，GC开始执行。即使TTS错误没有发生，你的服务也会给客户展现未预期的503错误。

**服务执行环境**

由于架构本身的特点，相比较而言纵向扩展，Web服务更适合横向扩展（译者注:增加服务器的数量，而不是提高件配置）。因此，总体来讲，物理设备会根据性能要求被配置成1台Apache+n台Tomcat。但是本文假设我们的环境是1台Apache+一台Tomcat同时安装在一台主机行，如下图所示。

[](http://www.importnew.com/wp-content/uploads/2013/02/1.png)

**图1：本文假射的服务执行环境**

仅供参考，本文描述的参数基于Apache 2.2.21 (prefork MPM)，Tomcat 6.0.35，CentOS 4.72 (32-bit)，jdk 1.6.0\_24。

系统可用内存2GB，垃圾收集器使用ParallelOldGC，AdaptiveSizePolicy采用默认的设置true，堆内存空间600M

**STW 和HTTP 503**

让我们假设访问Apache的请求为 200 req/s且有10个httpd进程在运行，另外我们暂时不考虑每个请求的响应时间。在这种前提下，我们假设由于full GC导致的暂停时间为1秒。**当Full GC发生的时候Tomcat会怎样？**

第一件进入你脑海的事情应该是Tomcat会因为full GC而停止响应任何请求。在这种情况下，**当Tomcat暂停相应请求时Apache会发生什么？**

当Tomcat暂停时，请求会以200 req/s的速度不断的涌入Apache。一般来说，在Full GC发生之前，请求的响应可以快速地被10个或更多的httpd进程处理掉。但是，因为Tomcat暂停了，httpd进程会被不停地创建以相应新进请求。直到超过**httpd.conf** 文件中定义 MaxClients 为止。由于默认值为256，Apache不会在乎请求以200 req/s的速度涌入。

这时，**新创建的httpd线程将如何呢？**

Httpd进程通过**mod\_jk** 模块所管理的空闲的AJP连接，将请求转发给Tomcat。如果没有空闲连接，他会申请创建新的连接。但是，因为Tomcat暂停了，创建新连接的请求会被拒绝。因此这些请求会被存储在backlog队列中，数量的多少取决于**server.xml**中关于AJP Connector的设置。一旦请求数量超过backlog队列的空间限制。Apache就会返回拒绝连接错误。并且返回**HTTP 503** 错误给用户。

在这种假设条件下，默认的backlog队列空间是100，而请求到达速度是200 req/s。因此，full GC导致的一秒钟的暂停会使得超过100个请求返回503错误。

这样，当Full GC结束后，backlog队列中存储的内容会被Tomcat接受并在通过工作线程处理，线程的最大数量取决于MaxThreads的值（默认200）。

**MaxClients 与backlog**

在这种情况下，**设定哪个参数可以避免返回给用户503错误呢？**

首先，我们应该知道backlog的值要够大，以至于能够容纳所有因为Full GC导致暂停期间涌入的请求。换句话说太应该不小于200。

那么，**这么设置之后会不会产生新的问题呢？**

让我们假设将backlog设置为200后再重复一下上面的过程。得到的结果比之前更加严重。系统内存使用量一般情况下为50%，但是，在发生Full GC时快速增加到100%，同时导致交换内存空间快速增加，更为严重的是导致Full GC的暂停时间从1秒变成了4秒甚至更多，系统在此期间完全宕机，不能响应任何请求。

在第一种情况下，只有100或更多的请求返回503错误。但是，当我们把backlog调整到200后，超过500个请求会挂起3秒甚至更多地时间无法得到应答

上面这个例子可以很好的说明当你没有完全理解各个设置之间的内在关系时（例如，对于系统的影响），盲目修改系统会导致什么后果。

那么，**为什么会产生这个现象呢？**

问题的根源在于 MaxClients 参数的特性。

将MaxClients 设置为一个很大的值本身没有问题，但最重要的是在设定MaxClients 参数时，你要确保即使等同于MaxClients 数量的httpd进程被同时创建，**内存使用量也不会超过80%**。

系统的内存交换参数一般被设定为60（默认）。因此，当内存使用量超过80%时，就会进行内存交换。

让我们再来看一下为什么这个特性会导致上面那个严重的问题。当请求以200 req/s的速度涌向Tomcat时，Tomcat由于full GC暂停了。此时backlog被设置为200。Apache大约创建100个httpd进程。在这种情况下，一旦内存使用量超过80%，操作系统会激活交换内存区域，并且由于系统认为JVM的老年代中的对象在很长一段时间内未被使用，而将他们移动到交换区域。

最终的结果是，GC使用了内存交换空间，暂停时间剧增。因此httpd进程数进一步增加。从而导致上面描述的内存使用量达到100%的情况。

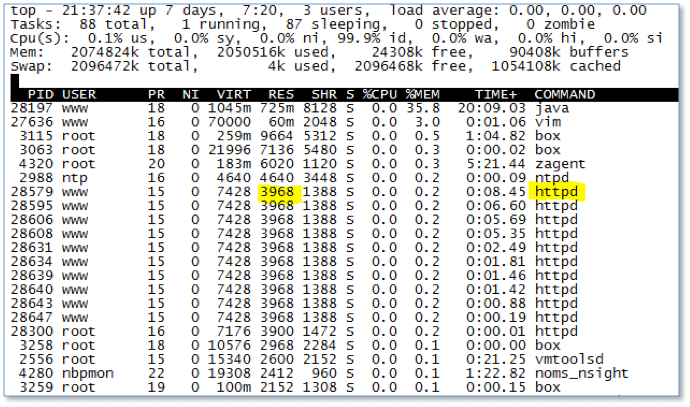
这两个场合的唯一区别就是backlog的值：100 vs.200。**为什么只在200的情况下发生？**

两者不同的原因在于创建的httpd进程的数量。当backlog设置为100时并且Full GC发生时，会创建100个请求的连接并保存在backlog队列中。其他请求得到拒绝连接错误信息并发挥503错误。因此，总的httpd 进程数量仅仅会略高于100。而当backlog被设置为200时，200个请求会创建连接，因此。总的httpd进程数会多于200。这样超过阀值，从而导致内存交换的发生。紧接着，不考虑内存使用量而的设定 MaxClients参数，Full GC导致httpd进程数量暴增，引发内存交换，降低系统性能。

**MaxClients参数的计算公式**

如果系统的内存使2GB，MaxClients 的值在任何情况下都不应该超过内存的80%（1.6GB），以避免由于内存交换导致的性能下降。换句话说。1.6GB的内存应该共享和分配给Apache，Tomcat以及那些默认被安装的代理程序。

让我们假设代理程序被默认安装在系统，并占用了200m内存，对于Tomcat堆内存的-Xmx 被设定为 600m。因此根据top命令的结果，Tomcat会一直占用725m（Perm Gen + Native Heap Area）。最终Apache可以使用700m内存空间。如下所示。

[](http://www.importnew.com/wp-content/uploads/2013/02/21.png)

**图2：测试系统的top截屏**

如上所述，**我们将内存设为700m后MaxClients 应该是多少呢？**

这要取决于加载模块的数量，对于NHN Web服务来说。Apache只是个简单的代理转发，每个httpd线程4m内存（根据top命令的结果）足以（参见图2）。因此。700m内存对应的 MaxClients应该是175。

**总结**

一个健壮的服务配置至少应该能够降低在服务过载时宕机的时间，在合理的范围内成功的应答请求。针对基于Java的Web服务。你必须检查你的服务在Full GC导致的STW时间内能否稳定的响应请求。

为了响应更多的用户请求和应对DDoS攻击，在没有全面考虑系统内存等因素的情况下，贸然地将 MaxClients设置为一个很大的值，那么它将失去作为阀值的功能，而导致系统出现更严重的问题。

本文提到的情况只会持续3-5秒，因此绝大多数传统的监控工具都无法及时的发现。

作者 Dongsoon Choi 高级工程师@Game Service Technical Support Team, NHN Corporation.

**英文原文：**[cubrid](http://www.cubrid.org/blog/dev-platform/maxclients-in-apache-and-its-effect-on-tomcat-during-full-gc/)**，编译：**[ImportNew](http://www.importnew.com/)**-**[王晓杰](http://www.importnew.com/author/wangxiaojie)

**译文地址：**<http://www.importnew.com/3151.html>

【如需转载，请在正文中标注并保留原文链接、译文链接和译者等信息，谢谢合作！】

## 成为Java GC专家（5）—Java性能调优原则 - ImportNew

http://www.importnew.com/13954.html

这是“[成为Java GC专家](http://www.cubrid.org/blog/tags/Garbage%20Collection/)”系列的第五篇文章。在第一篇[深入浅出Java垃圾回收机制](http://www.importnew.com/1993.html)中，我们已经学习了不同的GC算法流程、GC的工作原理、新生代（Young Generation）和老年代（Old Generation）的概念。你应该了解了JDK7中5种GC类型以及各种类型对应用程序的影响。

在第二篇[如何监控Java的垃圾回收](http://www.importnew.com/2057.html)中，阐述了[JVM](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/)是怎样实际执行垃圾回收的，我们怎样去监控GC以及哪些工具能让这个过程更高效。

第三篇[如何如何优化Java垃圾回收机制](http://www.importnew.com/3146.html)中展示了一些基于真实案例的最佳实践。同时讲解了怎样尽量少地将对象放入老年代空间（Old Area），避免频繁地执行完全垃圾回收（Full GC）。还说明了如何设置GC的类型和内存大小。

在第四篇[Apache的MaxClients参数详解及其在Tomcat执行FullGC时的影响](http://www.importnew.com/3151.html)中，解释了MaxClients参数的重要性以及它在垃圾回收过程中对整个系统性能的显著影响。

第五篇文章将讲解Java程序性能调优的原则，尤其是在这个过程中必要的知识以及判断你的程序是否需要调优。还会介绍调优过程中你可能遇到的问题。本文最后会给出一些建议，依据这些你能在对Java程序调优时做出更好的决策。

**概述**

并不是每个程序都需要调优。如果一个程序性能表现和预期一样，你不必付出额外的精力去提高它的性能。然而，在程序调试完成之后，很难马上就满足它的性能需求，于是就有了调优这项工作。无论哪种编程语言，对应用程序进行调优都需要丰富的技术知识并且注意力高度集中。另外，你也不应该用相同的方式对两个程序调优，因为每个程序都有它自己独特的运作方式和不同的资源使用方式。正因如此，调优比写程序需要更多基础知识。例如，你需要熟悉虚拟机、操作系统和计算机架构。而当你面对在这些知识基础上编写的程序时，就能成功地对它进行调优。

有时调优Java程序只需要修改JVM参数，比如GC的参数。但也有些时候需要修改程序代码。无论那种方法，你首先都需要监控执行Java程序的进程。因此本文会讲解下面几个问题：

* **怎样监控Java程序？**
* **应该给JVM设置怎样的参数？**
* **如何确定是否需要修改代码？**

**对Java程序进行调优的必要知识**

Java程序在Java虚拟机中运行。因此为了进行调优，你需要理解JVM的工作流程。我之前有一篇博文[Understanding JVM Internals](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/)，将让你对JVM有深入的了解。

本文中有关JVM运作过程的知识主要关于GC和Hotspot。尽管只有这两方面的知识可能无法对所有的Java程序进行调优，但是这两个因素在大多数情况下都影响着Java程序的性能。

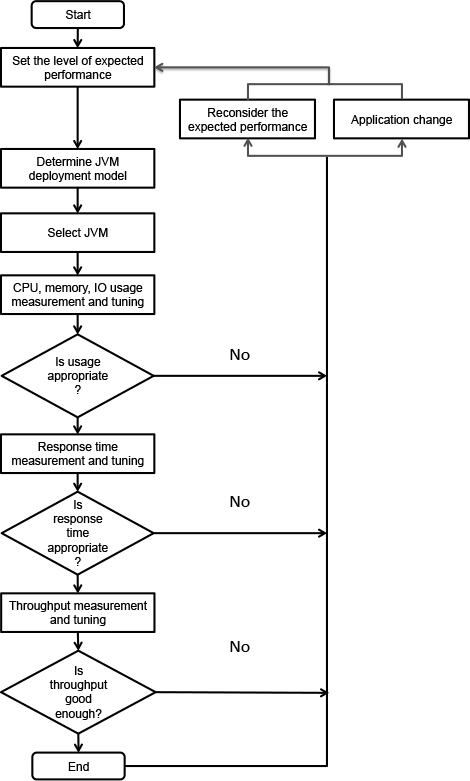
值得注意的是，从操作系统的角度来看，JVM也是一个应用程序进程。为了给JVM创造良好的运行环境，你还需要对操作系统分配资源的过程有所了解。这意味着，想要调优Java程序，除了JVM你也应该理解操作系统或者硬件的工作方式。

需要具有的知识还有Java这门语言本身。另外理解锁和并发、类加载和对象创建都是非常重要的。

当开始调优Java程序时，你应该整合以上各方面的知识来完成工作。

**Java程序性能调优的过程**

图1是一张Java程序性能调优的流程图，摘自由Charlie Hunt和Binu John所著的Java Performance。

[](http://www.importnew.com/13954.html/process-tuning-performance-java-applications)

*图1：Java程序性能调优的过程*

**JVM分布式模型**

**JVM分布式模型**用于决定是在一个JVM还是多个JVM上执行Java程序。你可以根据其有效性、响应能力和可维护性来进行选择。当在多台服务器上运行JVM时，你也可以选择将多个JVM运行于一台服务器或者每台服务器运行一个JVM。例如，对于每台服务器，你可以运行一个使用8GB堆内存的JVM，也可以运行4个使用2GB的JVM。你理应根据处理器内核的个数还有程序的特性来决定这个数量。当优先考虑响应能力时， 使用2GB的堆内存会优于8GB的，原因是这样能在更短的时间内完成Full GC。当然，8GB的堆内存可以降低Full GC的频率。如果你的程序使用了内部缓存，还可以通过增加缓存命中率来提高响应能力。综上所述，选择合适的模型需要考虑应用程序的特性，然后在各种模型中 选定一个能够扬长避短的。

**JVM架构**

选择JVM其实就是决定使用32位还是64位的JVM。在相同的条件下，你最好用32位的。因为32位的JVM比64位性能更好。然而，32位 JVM最大支持的堆内存是4GB（无论在32位操作系统还是64位的上，实际可分配的大小都只有2-3GB）。如果需要更大的堆内存，还是用64位的 JVM比较合适。

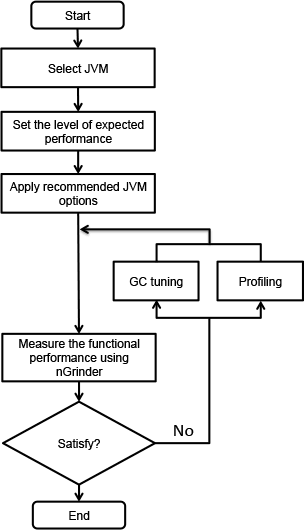
**表1：性能比较（**[数据来源](http://readwrite.com/2011/06/06/cpp-go-java-scala-performance-benchmark)**）**

| **测试基准** | **时间（秒）** | **系数** |
| --- | --- | --- |
| C++ Opt | 23 | 1.0x |
| C++ Dbg | 197 | 8.6x |
| Java 64-bit | 134 | 5.8x |
| Java 32-bit | 290 | 12.6x |
| Java 32-bit GC\* | 106 | 4.6x |
| Java 32-bit SPEC GC\* | 89 | 3.7x |
| Scala | 82 | 3.6x |
| Scala low-level\* | 67 | 2.9x |
| Scala low-level GC\* | 58 | 2.5x |
| Go 6g | 161 | 7.0x |
| Go Pro\* | 126 | 5.5x |

下一步就是运行程序来测试它的性能。这个过程包括GC调优、改变操作系统设置和修改代码。对于这些工作，你可以使用系统监视工具或者性能分析工具。

**注意：**针对响应能力的调优和针对吞吐量的调优可能使用不同的方法。如果经常性地发生[stop-the-word](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)（串行GC暂时中断程序执行），程序的响应能力就会被降低。比如在高吞吐量时执行Full GC。不要忘记，在调优时往往有得有失。这样需要折衷处理的事情不仅发生在响应能力和吞吐量之间。例如使用更多的CPU资源来降低内存的使用，或者不得不忍受响应能力和吞吐量其中一个性能指标的下降。相反的情况同样可能发生，实际的调优应该根据各指标的优先级来执行。

上面**图1**中的流程展示了几乎可用于所有Java程序的性能调优过程，包括Swing应用。然而，对于我们公司[NHN](http://www.cubrid.org/blog/tags/NHN/)用于提供网络服务的服务器端程序来说，这个方法多少有些不合适。下面**图2**中的流程是根据**图1**修改而来，它更简单，也更适合NHN。

[](http://www.importnew.com/13954.html/recommended-procedure-tuning-nhn-java-applications)

*图2：对HNH的Java程序的调优过程*

其中，**Select JVM**表示尽可能使用32位的JVM，除非你需要用64位的JVM来维护一个数GB的缓存。

现在，跟随**图2**中的流程，你会了解到每一步具体的工作。

**JVM参数**

我会主要讲解如何为Web服务端程序设置合适的JVM参数。尽管不一定适合所有的案例，但是**最好的GC算法**是[Concurrent Mark Sweep](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)（CMS垃圾回收），特别是对于Web服务端程序。因为**低延迟**是非常重要的。当然，在使用CMS时，由于新生代空间（New Area）的分配，可能发生较长时间的stop-the-world现象，不过调整新生代空间的大小或者它和整个堆空间的比例可能解决这个问题。

指定新生代空间的大小和指定整个对堆内存的大小同样重要。你最好使用–XX:NewRatio来指定新生代和整个堆的大小比例，或者直接用–XX:NewSize来指定所需的新生代空间。这个配置是非常必要的，因为大部分对象都不会存活很久。在Web程序中，除了缓存数据，其他多数对象都只在HttpRequest到HttpResponse期间创建。这个时间几乎不会超过1秒，表示这些对象的存活时间也不会超过1秒。如果新生代空间不够大，对象会被转移到老年代空间，以便腾出地方给新对象使用。老年代空间（Old Area）垃圾回收的代价是比新生代空间大的多的，因此很需要设置一个充足的新生代空间。

然而，当新生代空间的大小超过一个特定的水平，程序的响应能力会被降低。因为新生代空间的垃圾回收过程，基本上是将数据从一个Survivor Area复制到另外一个（From Space和To Space）。另外，stop-the-world的现象在新生代空间和老年代空间执行垃圾回收时都会发生。如果新生代空间变大，那么Survivor Area的空间也会更大，于是每次复制的数据就更多。基于这样一种特性，我们应该通过指定不同操作系统中HotSpot JVM的NewRatio参数来分配合适大小的新生代空间。

**表2：不同操作系统和配置下NewRatio的默认值**

| **操作系统及参数** | **默认-XX:NewRatio** |
| --- | --- |
| Sparc -server | 2 |
| Sparc -client | 8 |
| x86 -server | 8 |
| x86 -client | 12 |

如果设置了NewRatio，那么整个堆空间的1/(NewRatio +1)就是新生代空间的大小。上表可以看出**Sparc -server**的NewRatio默认值很小，因为相比**x86**的操作系统，Sparc以前更多用于高端应用，这个值就是为它们设置的。但现在x86操作系统的性能有很大提升，使用它们作为服务器已经很普遍了。因此指定NewRatio为2或者3是更好的选择，就和**Sparc -server**上的配置一样。

另外，你还可以通过指定NewSize和MaxNewSize来代替NewRatio。那么新生代空间创建时的大小就是指定的NewSize，随后可以一直增长到MaxNewSize的值。Eden（新创建对象存放的区域）和Survivor Area两个区域会随比例增加。就和你为-Xms（**译者注：**原文是-Xs，应该是笔误）和-Xmx设置相同的值一样，将MaxSize和 MaxNewSize设置为相同的也是一个好选择。

如果同时指定了NewRatio和NewSize，你应该使用更大的那个。于是，当堆空间被创建时，你可以用过下面的表达式计算初始新生代空间的大小：

|  |  |
| --- | --- |
| 1 | min(MaxNewSize, max(NewSize, heap/(NewRatio+1))) |

无论如何，仅通过一次尝试就找到合适的堆空间和新生代空间大小是不可能的。根据我在NHN运行Web服务器的经验，建议使用下面的JVM参数来运行Java程序。监控在这些参数的条件下程序的性能表现之后，你就能够选择更合适的GC算法或者配置。

**表3：推荐的JVM参数**

| **类型** | **参数** |
| --- | --- |
| 运行模式 | -sever |
| 整个堆内存大小 | 为-Xms和-Xmx设置相同的值。 |
| 新生代空间大小 | -XX:NewRatio: 2到4. -XX:NewSize=? –XX:MaxNewSize=?. 使用NewSize代替NewRatio也是可以的。 |
| 持久代空间大小 | -XX:PermSize=256m -XX:MaxPermSize=256m. 设置一个在运行中不会出现问题的值即可，这个参数不影响性能。 |
| GC日志 | -Xloggc:$CATALINA\_BASE/logs/gc.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps. 记录GC日志并不会特别地影响Java程序性能，推荐你尽可能记录日志。 |
| GC算法 | -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75. 一般来说推荐使用这些配置，但是根据程序不同的特性，其他的也有可能更好。 |
| 发生OOM时创建堆内存转储文件 | -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=$CATALINA\_BASE/logs |
| 发生OOM后的操作 | -XX:OnOutOfMemoryError=$CATALINA\_HOME/bin/stop.sh 或 -XX:OnOutOfMemoryError=$CATALINA\_HOME/bin/restart.sh. 记录内存转储文件后，为了管理的需要执行一个合适的操作。 |

**测定程序的性能**

为了得到程序的性能表现，需要以下这些信息：

* **系统吞吐量（TPS、OPS）**：从整体概念上理解程序的性能。
* **每秒请求数（Request Per Second – RPS）**：严格来说，RPS和单纯的响应能力是不同的，但是你可以把它理解为响应能力。通过这个指标，你能够了解到用户需要多长时间才能得到请求的结果。
* **RPS的标准差**：如果可能的话，还有必要包括事件的RPS。一旦出现了偏差，你应该检查GC或者网络系统。

为了得到更准确的性能表现，你应该等到程序彻底启动完成后再进行测量，因为字节码随后会被HotSpot JIT编译为本地机器码。总体来说，需要在程序加载完指定功能后，用[nGrinder](http://www.nhnopensource.org/ngrinder/)等工具测试至少10分钟。

**切实地调优**

如果nGrinder测试的结果满足了预期，那么你不需要对程序进行性能调优。如果没有达到预期结果，你就应该执行调优来解决问题。接下来会通过实例讲解方法。

**stop-the-world耗时过长**

stop-the-world耗时过长可能是由于GC参数不合理或者代码实现不正确。你可以通过分析工具或堆内存转储文件（Heap dump）来定位问题，比如检查堆内存中对象的类型和数量。如果在其中找到了很多不必要的对象，那么最好去改进代码。如果没有发现创建对象的过程中有特别的问题，那么最好单纯地修改GC参数。

为了适当地调整GC参数，你需要获取一段足够长时间的GC日志，还必须知道哪些情况会导致长时间的stop-the-world。想了解更多关于如何选择合适的GC参数，可以阅读我同事的一篇博文：[How to Monitor Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/)。

**CPU使用率过低**

当系统发生阻塞，吞吐量和CPU使用率都会降低。这可能是由于网络系统或者并发的问题。为了解决这个问题，你可以分析线程转储信息（Thread dump）或者使用分析工具。阅读这篇文章可以获得更多关于线程转储分析的知识：[How to Analyze Java Thread Dumps](http://www.cubrid.org/blog/dev-platform/how-to-analyze-java-thread-dumps/)。

你可以使用商业的分析工具对线程锁进行精确的分析，不过大部分时候，只需使用**JVisualVM**中的CPU分析器，就能获得足够的信息。

**CPU使用率过高**

如果吞吐量很低但是CPU使用率却很高，很可能是低效率代码导致的。这种情况下，你应该使用分析工具定位代码中性能的瓶颈。可使用的工具有：**JVisualVM**、[Eclipse](http://res.importnew.com/eclipse) **TPTP**或者**JProbe**。

**调优方法**

建议你使用如下方法对程序进行调优。

首先，检查性能调优是否必要。测量性能不是一件简单的工作，你也不能保证每次都获得满意的结果。因此如果程序已经满足预期性能需求，不必在调优上增加额外的投入了。

问题只出在一个地方，你要做的就是去解决掉它。二八定律（[Pareto principle](http://en.wikipedia.org/wiki/Pareto_principle)）对性能调优同样适用。这不是说某个模块的低性能一定只源于一个问题，而是强调我们应该在调优时把注意力放在影响最大的那个问题上。在处理好了最重要的之后，你才应该去解决剩下其他的。也就是建议一次只对一个问题进行修复。

另外需要考虑到气球效应（[Balloon effect](http://en.wikipedia.org/wiki/Balloon_effect)），有得必有失。你可以通过使用缓存来提高响应能力，但是当缓存逐渐增大，执行一次Full GC的时间也会更长。一般而言，如果你希望内存使用率比较低，那么吞吐量和响应能力可能都会恶化。因此，要知道什么对自己程序来说最重要的，而哪些又是次要的。

到此为止，你应该已经了解了如何对Java程序进行性能调优。为了介绍性能测定的具体过程，我不得不省略其中一些细节，不过我认为这些也足够应对大多数Java Web服务端程序了。

最后祝调优好运！

原文链接： [dzone](http://java.dzone.com/articles/principles-java-application) 翻译： [ImportNew.com](http://www.importnew.com/)- [蒋 生武](http://www.importnew.com/author/jiangshengwu)  
译文链接： <http://www.importnew.com/13954.html>  
[ **转载请保留原文出处、译者和译文链接。**]

**关于作者：**[蒋 生武](http://www.importnew.com/author/jiangshengwu)

# Java垃圾回收精粹

## Java垃圾回收精粹 — Part1 - ImportNew

http://www.importnew.com/8335.html

## Java垃圾回收精粹 — Part2 - ImportNew

http://www.importnew.com/8343.html

## Java垃圾回收精粹 — Part3 - ImportNew

http://www.importnew.com/8347.html

## Java垃圾回收精粹 — Part4 - ImportNew

http://www.importnew.com/8352.html

# JVM性能优化

## JVM性能优化， Part 1 ―― JVM简介 - ImportNew

http://www.importnew.com/1774.html

## JVM性能优化， Part 2 ―― 编译器 - ImportNew

http://www.importnew.com/2009.html

## JVM性能优化， Part 3 垃圾回收 - ImportNew - ImportNew

http://www.importnew.com/2233.html

## JVM 性能优化, Part 4: C4 垃圾回收 - ImportNew - ImportNew

http://www.importnew.com/2410.html

## gc

### 字段详细含义 | GC日志分析 - - 博客频道 - CSDN.NET

http://blog.csdn.net/huangzhaoyang2009/article/details/11860757

**JVM的GC日志的主要参数包括如下几个：**

-XX:+PrintGC 输出GC日志

-XX:+PrintGCDetails 输出GC的详细日志

-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:../logs/gc.log 日志文件的输出路径

在我做了如下的设置

**Java代码  收藏代码**

1. -XX:+PrintGCDetails -Xloggc:../logs/gc.log -XX:+PrintGCTimeStamps

以后打印出来的日志为：

**Java代码  收藏代码**

1. 0.756: [Full GC (System) 0.756: [CMS: 0K->1696K(204800K), 0.0347096 secs] 11488K->1696K(252608K), [CMS Perm : 10328K->10320K(131072K)], 0.0347949 secs] [Times: user=0.06 sys=0.00, real=0.05 secs]
2. 1.728: [GC 1.728: [ParNew: 38272K->2323K(47808K), 0.0092276 secs] 39968K->4019K(252608K), 0.0093169 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
3. 2.642: [GC 2.643: [ParNew: 40595K->3685K(47808K), 0.0075343 secs] 42291K->5381K(252608K), 0.0075972 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
4. 4.349: [GC 4.349: [ParNew: 41957K->5024K(47808K), 0.0106558 secs] 43653K->6720K(252608K), 0.0107390 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
5. 5.617: [GC 5.617: [ParNew: 43296K->7006K(47808K), 0.0136826 secs] 44992K->8702K(252608K), 0.0137904 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
6. 7.429: [GC 7.429: [ParNew: 45278K->6723K(47808K), 0.0251993 secs] 46974K->10551K(252608K), 0.0252421 secs]

**我们取倒数第二条记录分析一下各个字段都代表了什么含义**

**Java代码  收藏代码**

1. 5.617（时间戳）: [GC（Young GC） 5.617（时间戳）: [ParNew（使用ParNew作为年轻代的垃圾回收期）: 43296K（年轻代垃圾回收前的大小）->7006K（年轻代垃圾回收以后的大小）(47808K)（年轻代的总大小）, 0.0136826 secs（回收时间）] 44992K（堆区垃圾回收前的大小）->8702K（堆区垃圾回收后的大小）(252608K)（堆区总大小）, 0.0137904 secs（回收时间）] [Times: user=0.03（Young GC用户耗时） sys=0.00（Young GC系统耗时）, real=0.02 secs（Young GC实际耗时）]

**我们再对数据做一个简单的分析**

从最后一条GC记录中我们可以看到 Young GC回收了**45278-6723=38555K**的内存

Heap区通过这次回收总共减少了**46974-10551=36423K**的内存。

**38555-36423=2132K**说明通过该次Young GC有2132K的内存被移动到了Old Gen，

我们来验证一下

在最后一次Young GC的回收以前 Old Gen的大小为**8702-7006=1696**

回收以后Old Gen的内存使用为**10551-6723=3828**

Old Gen在该次Young GC以后内存增加了**3828-1696=2132K** 与预计的相符

**重新设置GC日志的输出**

**Java代码  收藏代码**

1. -XX:+PrintGCDetails
2. -XX:+PrintHeapAtGC
3. -XX:+PrintGCDateStamps
4. -XX:+PrintTenuringDistribution
5. -verbose:gc
6. -Xloggc:gc.log

后可以看到进行GC前后的堆内存信息

**Java代码  收藏代码**

1. {Heap before GC invocations=1 (full 0):
2. PSYoungGen      total 152896K, used 131072K [0x00000000f5560000, 0x0000000100000000, 0x0000000100000000)
3. eden space 131072K, 100% used [0x00000000f5560000,0x00000000fd560000,0x00000000fd560000)
4. from space 21824K, 0% used [0x00000000feab0000,0x00000000feab0000,0x0000000100000000)
5. to   space 21824K, 0% used [0x00000000fd560000,0x00000000fd560000,0x00000000feab0000)
6. PSOldGen        total 349568K, used 0K [0x00000000e0000000, 0x00000000f5560000, 0x00000000f5560000)
7. object space 349568K, 0% used [0x00000000e0000000,0x00000000e0000000,0x00000000f5560000)
8. PSPermGen       total 26432K, used 26393K [0x00000000d0000000, 0x00000000d19d0000, 0x00000000e0000000)
9. object space 26432K, 99% used [0x00000000d0000000,0x00000000d19c64a0,0x00000000d19d0000)
10. 2013-05-05T23:16:10.480+0800: 5.228: [GC
11. Desired survivor size 22347776 bytes, **new** threshold 7 (max 15)
12. [PSYoungGen: 131072K->8319K(152896K)] 131072K->8319K(502464K), 0.0176346 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
13. Heap after GC invocations=1 (full 0):
14. PSYoungGen      total 152896K, used 8319K [0x00000000f5560000, 0x0000000100000000, 0x0000000100000000)
15. eden space 131072K, 0% used [0x00000000f5560000,0x00000000f5560000,0x00000000fd560000)
16. from space 21824K, 38% used [0x00000000fd560000,0x00000000fdd7ff78,0x00000000feab0000)
17. to   space 21824K, 0% used [0x00000000feab0000,0x00000000feab0000,0x0000000100000000)
18. PSOldGen        total 349568K, used 0K [0x00000000e0000000, 0x00000000f5560000, 0x00000000f5560000)
19. object space 349568K, 0% used [0x00000000e0000000,0x00000000e0000000,0x00000000f5560000)
20. PSPermGen       total 26432K, used 26393K [0x00000000d0000000, 0x00000000d19d0000, 0x00000000e0000000)
21. object space 26432K, 99% used [0x00000000d0000000,0x00000000d19c64a0,0x00000000d19d0000)
22. }

**Java代码  收藏代码**

1. [0x00000000f5560000,0x00000000f5560000,0x00000000fd560000)

这种形式的日志有两种意义：   
当这种日志出现在generation的详细信息里的时候，三个数字在HotSpot里分别称为low\_boundary、high、high\_boundary。   
low\_boundary: reserved space的最低地址边界；通常也跟“low”相等，这是commited space的最低地址边界   
high: commited space的最高地址边界   
high\_boundary: reserved space的最高地址边界。   
  
[low\_boundary, high\_boundary)范围内的就是reserved space，这个space的大小就是max capacity。   
[low, high)范围内的就是commited space，而这个space的大小就是current capacity（当前容量），简称capacity。   
capacity有可能在一对最小值和最大值之间浮动。最大值就是上面说的max capacity。

转载地址：<http://swcdxd.iteye.com/blog/1859858>

### Minor GC、Major GC和Full GC之间的区别 - ImportNew

http://www.importnew.com/15820.html

### Eclipse Memory Analyzer|诊断Java.lang.OutOfMemoryError（OOM） - ImportNew

http://www.importnew.com/1586.html

在跟踪性能问题时，堆内存是首先应该被监控的最重要的组件之一。一旦堆内存的实际使用量超过其所允许的堆空间，就会产生堆内存压力。而这将导致频繁的全面垃圾回收事件，垃圾回收将窃取CPU周期，轻则导致响应时间延迟，重则导致必须重新启动Java虚拟机才能解决的内存溢出错误。

**内存溢出错误（OOM）**  
当我运行应用时，出现了如下异常：

* java.lang.OutOfMemoryError: GC overhead limit exceeded[7,9]
* java.lang.OutOfMemoryError: Java heap space

第一条信息意味着，出于某种原因，垃圾收集器每次执行都花费了大量时间但只回收了很少量的内存，当我删除了如下代码后：

|  |  |
| --- | --- |
| 1 | System.gc(); |

第一条信息消失了，取而代之的，系统出现了第二条信息。很明显堆内存空间依然存在问题。下面是我调查问题的步骤：

1. 添加下面的Java启动参数
   * -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
     + 系统会生成gc.log文件
   * -XX:+HeapDumpOnOutOfMemoryError
     + 系统会生成堆内存转储文件
2. 分析日志文件:
   * 使用常规的文本编辑器查看gc.log 文件。
   * 使用 [Eclipse](http://res.importnew.com/eclipse) Memory Analyzer 查看 堆内存转储文件 (例如, java\_xxx.hprof)

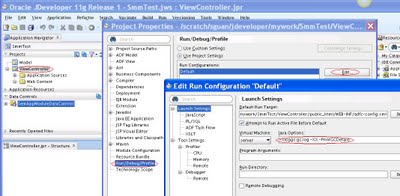
请注意，本文所讨论的所有虚拟机参数都是基于Hotspot虚拟机的。

**Java命令行参数说明:**

* -XX:+PrintGCDetails
  + 打印更多的关于垃圾收集的信息。
* -XX:+PrintGCTimeStamps
  + 打印从HotSpot 虚拟机开始执行直至垃圾收集事件发生所花费的时间（以秒为单位）。
* -Xloggc:gc.log
  + 在每次垃圾收集时打印堆内存以及垃圾收集的信息。

**在JDeveloper中可以按照如下方式设定：**

1. 右键选择你的项目（例如ViewController），显示出菜单
2. 选择Project Properties…
3. 选择Run/Debug/Profile
4. 选择你Run Configuration(例如, Default)
5. 点击Edit按钮
6. 在Java虚拟机参数栏位设定 -Xloggc:gc.log -XX:-PrintGCDetails

[](http://www.importnew.com/wp-content/uploads/2012/12/OOM-1-1.jpg)

运行你的应用并重现内存溢出异常，系统将会生成日志文件gc.log，  
我的是在如下目录：

* …/system11.1.1.5.37.60.13/DefaultDomain

因为我的Web应用是部署在集成的WLS中，并且通过DefaultDomain来执行。  
所以，想要理解gc.log文件的格式，请参考关联阅读[5,15]。

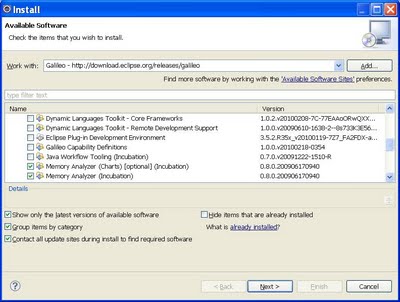
不过，gc.log文件并不能真正的帮到我们，因为他只是简要的打印了堆内存问题，

但并没有指出问题出在哪。  
接下来我要做的是添加如下参数，并重新执行服务。  
**-XX:+HeapDumpOnOutOfMemoryError**  
当服务发生堆内存错误时，会生成java\_pid30835.hprof文件。

**Eclise内存分析器（Eclipse Memory Analyzer）**

堆内存转储文件由HPROF（堆内存和CPU分析工具）生成，堆内存转储文件是2进制格式的，因此必须使用Eclpse Memory Analyzer 来查看。

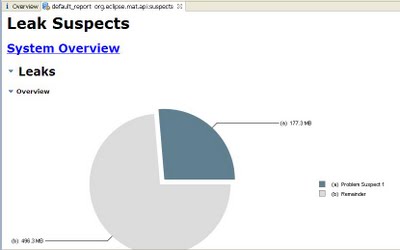
你可以通过Eclipse Update manager 来安装Eclipse MAT，选择”General Purpose Tools “并安装”Memory Analyser (Incubation)”以及”Memory Analyser (Charts)”。

[](http://www.importnew.com/wp-content/uploads/2012/12/OOM-1-2.jpg)

安装之后，双击堆内存转储文件并且选择”Leak Suspects Report”

[](http://www.importnew.com/wp-content/uploads/2012/12/OOM-1-3.jpg)

Eclipse MAT会显示如下图表：

[](http://www.importnew.com/wp-content/uploads/2012/12/OOM-1-4.jpg)

以及问题的嫌疑人：

[](http://www.importnew.com/wp-content/uploads/2012/12/OOM-1-5.jpg)

**调整堆内存空间**

如果你观察到垃圾收集日志文件中有内存溢出错误，那么可以尝试将Java堆内存空间调整为你能够分配给Java虚拟机的物理内存空间的80%，基于具体是老年代空间还是永久代空间发生内存溢出，你可以像这样调整内存空间。

* 针对老年代发生内存溢出
  + increase -Xms and -Xmx
* 针对永久代发生内存溢出
  + increase -XX:PermSize and -XX:MaxPermSize

**参考文献：**

[Eclipse Update Manager](http://www.vogella.de/articles/Eclipse/article.html#updatemanager)

[Eclipse Memory Analyzer](http://www.vogella.de/articles/EclipseMemoryAnalyser/article.html)

[Java Hotspot VM Options](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html)

[Integrated WebLogic Server (WLS)](http://xmlandmore.blogspot.com/2011/04/integrated-weblogic-server-wls.html)

[Diagnosing a Garbage Collection problem](http://java.sun.com/docs/hotspot/gc1.4.2/example.html)

[Frequently Asked Questions about Garbage Collection](http://www.oracle.com/technetwork/java/faq-140837.html)

[GC Overhead Limit Exceeded](http://stackoverflow.com/questions/1393486/what-means-the-error-message-java-lang-outofmemoryerror-gc-overhead-limit-excee)

[HPROF: A Heap/CPU Profiling Tool in J2SE 5.0](http://java.sun.com/developer/technicalArticles/Programming/HPROF.html)

[Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning](http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#par_gc.oom)

Java Performance by Charlie Hunt and Binu John

[Understanding Garbage Collection](http://xmlandmore.blogspot.com/2012/01/understanding-garbage-collection.html)

[Java HotSpot VM Options](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html)

[GCViewer](http://www.tagtraum.com/gcviewer.html) (a free open source tool)

[Understanding Garbage Collector Output of Hotspot VM](http://xmlandmore.blogspot.com/2012/03/understanding-garbage-collector-output.html)

**英文原文：**[xmlandmore](http://xmlandmore.blogspot.kr/2011/05/diagnosing-javalangoutofmemory.html/)**，编译：**[ImportNew](http://www.importnew.com/)**-**[王晓杰](http://www.importnew.com/author/wangxiaojie)

**译文地址：**<http://www.importnew.com/1586.html>

【如需转载，请在正文中标注并保留原文链接、译文链接和译者等信息，谢谢合作！】

### 垃圾回收时-XX:+UseParallelGC -XX:+UseParNewGC两参数 的区别是什么? - 开源中国社区

http://www.oschina.net/question/867417\_87507?sort=time

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。可以同时并行多个垃圾收集线程，但此时用户线程必须停止。

-XX:+UseParNewGC:设置年轻代为多线程收集。可与CMS收集同时使用。在serial基础上实现的多线程收集器。

### EclipseZone - Best JVM settings for Eclipse ...

http://www.eclipsezone.com/eclipse/forums/t20732.html

In my neverending quest to find the best JVM settings for running Eclipse I've been pouring over talks and papers given by Sun engineers and trying different things. Here are the best options that I've found so far for my 2-processor Windows machine running JDK5.0 and Eclipse3.1:   
  
-vmargs -XX:+UseParallelGC   
  
These go on the Eclipse executable's command line. What I'm doing here is taking advantage of the JDK5.0 garbage collection "ergonomics" feature to let the JVM self tune. It's only active with the UseParallelGC collector (also called the *parallel scavenge*collector). Note there is no -XmxNNNm option to set the heap size. According to the doc, any sizing or tweaking options will disable self tuning because it figures we know better than it does (it doesn't know us very well does it?).   
  
If you want to see what the garbage collector is doing you could add more printing options like this (all on one line) to dump some logging information to a file that you can view or process later:   
  
-vmargs -Xloggc:c:tempe.txt -XX:+PrintGCDetails -XX:+PrintGCTimeStamps   
-XX:+PrintTenuringDistribution -verbose:gc -XX:+UseParallelGC   
  
  
Here are some references if you want to do your own research: 

[Java Performance Tuning From A Garbage Collection Perspective](http://web.princeton.edu/sites/isapps/jasig/2004summerWestminster/Presentations/java-sig2004_garbagecollection.pdf)

[NYC JavaSIG ? August 2004 JavaOne Overview](http://www.javasig.com/Archive/lectures/JavaSIG-JavaONE2004Overview.pdf)

[Performance Documentation for the Java HotSpot VM](http://java.sun.com/docs/hotspot/index.html)

[Garbage Collector Ergonomics](http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html)

[A Collection of JVM Options](http://blogs.sun.com/roller/resources/watt/jvm-options-list.html)

[Using jconsole](http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html)

If you find something better, drop me a line. Enjoy!

17 replies so far ([http://www.eclipsezone.com/images/std/icons/write_16x16.gif](http://www.eclipsezone.com/forums/post!reply.jspa?threadID=20732) [Post your own](http://www.eclipsezone.com/forums/post!reply.jspa?threadID=20732))

### [原]如何加快Eclipse的启动速度？ - 推酷

http://www.tuicool.com/articles/a6FRju

原文  [http://blog.csdn.net/lanxuezaipiao/article/details/42261871](http://blog.csdn.net/lanxuezaipiao/article/details/42261871?utm_source=tuicool)

主题 [Eclipse](http://www.tuicool.com/topics/11070068)[数据库](http://www.tuicool.com/topics/11000064)

很多人感觉自己的elipse启动比较慢，其实并不是因为装的插件太多或者是导入的项目有点大，而是因为参数的设置不合理导致的。可以在eclipse.ini里面添加  
-Xloggc:gc.log看看启动的日志。下面简单的说一下通过优化一下配置来节约eclipse的启动时间，文中的数据是我本机的环境，我本机内存为3.16G。  
第一次优化：把-Xms（初始化堆大小）-Xmx（JVM最大堆大小）设置为512m，避免频繁GC(垃圾回收)。如下所示：

重启eclipse的时候Full GC的次数为8次，minitor GC的次数为3

不知道大家是不是跟我一样，打开Eclipse都是龟速，每次点击打开后，我都会跑出去倒杯水再回来。那么有没有什么办法可以加快启动速度呢？我的思路主要有以下几个：

* 卸载不会再用或不常用的一堆插件。
* 别把所有的项目都放在一个workspace里面，可以多建几个workspace。比如最近一段时间要着手的一个或几个项目放在一个workspace里。
* 通过修改Eclipse的配置文件（ eclipse.ini ）来加快Eclipse的启动速度。

前面两个相信大部分人都能想到，做起来也很简单， **本文主要介绍第三种方式，即通过修改配置文件来加速。**

我的环境：

Eclipse版本：Eclipse LUNA

JDK版本：jdk 1.8.0

本机内存：6G

首先需要在 eclipse.ini 里面添加 -Xloggc:gc.log 配置来查看启动的日志，然后通过分析日志来调优参数。

我原先的vmargs参数配置如下（后面添加了几个打印日志选项）

-Xms40m

-Xmx256m

-verbose:gc

-XX:+PrintGCDateStamps

-XX:+PrintGCDetails

-Xloggc:gc.log

启动Eclipse成功后，配置目录下会多处一个gc.log的日志文件，内容如下：

2014-12-01T17:17:24.102+0800: 1.011: [GC (Allocation Failure) [PSYoungGen: 10240K->1528K(11776K)] 10240K->2573K(39424K), 0.0357842 secs] [Times: user=0.00 sys=0.00, real=0.04 secs]

2014-12-01T17:17:24.695+0800: 1.604: [GC (Allocation Failure) [PSYoungGen: 11768K->1528K(11776K)] 12813K->6418K(39424K), 0.0163152 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

2014-12-01T17:17:24.933+0800: 1.843: [GC (Allocation Failure) [PSYoungGen: 11768K->1528K(11776K)] 16658K->10668K(39424K), 0.0209557 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

2014-12-01T17:17:25.766+0800: 2.676: [GC (Allocation Failure) [PSYoungGen: 11768K->1528K(22016K)] 20908K->12994K(49664K), 0.0152186 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

2014-12-01T17:17:27.385+0800: 4.294: [GC (Allocation Failure) [PSYoungGen: 22008K->1521K(22016K)] 33474K->19971K(49664K), 0.0243423 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

2014-12-01T17:17:28.515+0800: 5.424: [GC (Metadata GC Threshold) [PSYoungGen: 14910K->6286K(50176K)] 33360K->24736K(77824K), 0.0408780 secs] [Times: user=0.09 sys=0.00, real=0.04 secs]

2014-12-01T17:17:28.556+0800: 5.466: [ Full GC (Metadata GC Threshold) [PSYoungGen: 6286K->0K(50176K)] [ParOldGen: 18450K->21574K(46080K)] 24736K->21574K(96256K), [Metaspace: 19172K->19172K(1069056K)], 0.3825136 secs] [Times: user=0.75 sys=0.00, real=0.38 secs]

2014-12-01T17:17:33.134+0800: 10.044: [GC (Allocation Failure) [PSYoungGen: 40960K->9212K(50176K)] 62534K->34014K(96256K), 0.0848276 secs] [Times: user=0.13 sys=0.00, real=0.09 secs]

2014-12-01T17:17:35.882+0800: 12.792: [GC (Metadata GC Threshold) [PSYoungGen: 37650K->12260K(61952K)] 62451K->41391K(108032K), 0.0443515 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]

2014-12-01T17:17:35.926+0800: 12.837: [ Full GC (Metadata GC Threshold) [PSYoungGen: 12260K->0K(61952K)] [ParOldGen: 29131K->31402K(68096K)] 41391K->31402K(130048K), [Metaspace: 32296K->32296K(1079296K)], 0.2952576 secs] [Times: user=0.45 sys=0.00, real=0.30 secs]

2014-12-01T17:17:40.580+0800: 17.490: [GC (Allocation Failure) [PSYoungGen: 49664K->12627K(65536K)] 81066K->44038K(133632K), 0.0400038 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]

2014-12-01T17:17:43.820+0800: 20.730: [GC (Allocation Failure) [PSYoungGen: 62291K->17399K(63488K)] 93702K->51113K(131584K), 0.0402957 secs] [Times: user=0.02 sys=0.02, real=0.04 secs]

2014-12-01T17:17:44.362+0800: 21.271: [GC (Metadata GC Threshold) [PSYoungGen: 25530K->13577K(59904K)] 59244K->47299K(128000K), 0.0291300 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

2014-12-01T17:17:44.391+0800: 21.301: [ Full GC (Metadata GC Threshold) [PSYoungGen: 13577K->0K(59904K)] [ParOldGen: 33721K->39688K(88576K)] 47299K->39688K(148480K), [Metaspace: 53933K->53933K(1099776K)], 0.4561979 secs] [Times: user=0.67 sys=0.02, real=0.46 secs]

2014-12-01T17:17:50.364+0800: 27.274: [GC (Allocation Failure) [PSYoungGen: 46080K->11833K(66048K)] 85768K->51530K(154624K), 0.0260724 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

2014-12-01T17:17:52.379+0800: 29.289: [GC (Allocation Failure) [PSYoungGen: 56889K->14031K(66048K)] 96586K->53735K(154624K), 0.0301987 secs] [Times: user=0.06 sys=0.00, real=0.03 secs]

2014-12-01T17:17:54.831+0800: 31.742: [GC (Allocation Failure) [PSYoungGen: 59087K->16928K(62976K)] 98791K->56641K(151552K), 0.0651508 secs] [Times: user=0.14 sys=0.00, real=0.07 secs]

2014-12-01T17:18:12.857+0800: 49.768: [GC (Allocation Failure) [PSYoungGen: 58912K->17695K(59904K)] 98625K->57415K(148480K), 0.0486718 secs] [Times: user=0.09 sys=0.00, real=0.05 secs]

可以发现Full GC的次数为3次，minior GC的次数为15次。进一步分析发现：

* 触发Full GC和部分GC的原因是“ **Metadata GC Threshold** ”，没用过java 8的人可能对这个不熟悉，因为Metaspace是java 8的新特性，取代了大家非常熟悉的PermGen，具体了解可以看这篇文章： [*Java 8新特性探究（九）跟OOM：Permgen说再见吧*](http://my.oschina.net/benhaile/blog/214159) 。   
  上面的意思是说Metaspace的大小设置的太小了，造成频繁GC，这个可以通过 -XX:MetaspaceSize 参数设置。   
  注意：关于Metaspace的配置不只是 -XX:MetaspaceSize 这一个参数，还有其它的，主要有下面这些：
  + -XX:InitialBootClassLoaderMetaspaceSize=32M ：to increase the boot class loader Metaspace;
  + -XX:MinMetaspaceFreeRatio=50 ：to make Metaspaces grow more agressively;
  + -XX:MaxMetaspaceFreeRatio=80 ：to reduce the chance of Metaspaces shrinking;
  + -XX:MinMetaspaceExpansion=4M ：the minumum size by which a Metaspace is exanded;
  + - XX:MaxMetaspaceExpansion=16M ：the maximum size to expand a Metaspace by without Full GC.
* 大部分的Minor GC都出现了AF（Allocation Failure）错误，AF 并不表示有什么代码发生了错误，而只是无法从堆中分配足够的空间。这也能理解，上面Eclipse的默认配置中Xms（初始化堆大小）只有40m，-Xmx（最大堆大小）只有256m，这里加大这两个参数的配置即可。

经过上述分析，我修改的配置参数如下：

-Xms512m

-Xmx1024m

-XX:MetaspaceSize=128M

-verbose:gc

-XX:+PrintGCDateStamps

-XX:+PrintGCDetails

-Xloggc:gc.log

此时重启Eclipse后发现速度稍快了点，打印日志如下：

2014-12-01T18:14:11.134+0800: 7.991: [GC (Allocation Failure) [PSYoungGen: 131584K->21493K(153088K)] 131584K->35355K(502784K), 0.1561477 secs] [Times: user=0.30 sys=0.02, real=0.16 secs]2014-12-01T18:14:18.805+0800: 15.662: [GC (Allocation Failure) [PSYoungGen: 153077K->21494K(284672K)] 166939K->67517K(634368K), 0.1496186 secs] [Times: user=0.16 sys=0.05, real=0.15 secs]

发现Full GC消失了，最后剩下2次Minor GC。不过这2次AF问题我是调不掉了，不管修改什么参数，最终还是会出现这2次GC，相比之前已经很好了，就到此为止吧。如果您有更好的设置方法（比如能够消除最后的两次GC），或者纠正文中错误的方法，请留言告知。

参考资料

### garbage collection - Java GC (Allocation Failure) - Stack Overflow

http://stackoverflow.com/questions/28342736/java-gc-allocation-failure

"Allocation Failure" is a cause of GC cycle to kick.

"Allocation Failure" means what no more space left in Eden to allocate object. So, it is normal cause of young GC.

Older JVM were not printing GC cause for minor GC cycles.

"Allocation Failure" is almost only possible cause for minor GC. Another reason for minor GC to kick could be CMS remark phase (if +XX:+ScavengeBeforeRemark is enabled).

### eclipse gc 配置

在eclipse安装目录上生成gc.log

提高开发效率之tomcat免重启(随意更改java代码) - 企业应用 - Java - ... - 文章写作网

http://www.articleswriting.net/article/12041147922/

eclipse.ini 文件里面最后加入：（这里面要把原有的部分去掉一些）

-vmargs

-Xms128M

-Xmx512M

-XX:PermSize=64M

-XX:MaxPermSize=512M

-XX:+PrintGCTimeStamps

-XX:+PrintGCDetails

-XX:+TraceClassLoading

-XX:+TraceClassUnloading

-XX:+PrintGCTimeStamps

-XX:+PrintGCDetails

-verbose:gc

-Xloggc:gc.log

### JVM实用参数（八）GC日志 | 并发编程网 - ifeve.com

http://ifeve.com/useful-jvm-flags-part-8-gc-logging/

JVM实用参数（八）GC日志

原文地址：<https://blog.codecentric.de/en/2014/01/useful-jvm-flags-part-8-gc-logging/>

作者：[PATRICK PESCHLOW](https://blog.codecentric.de/en/author/patrick-peschlow/)，译者：Greenster 校对：梁海舰

本系列的最后一部分是有关垃圾收集（GC）日志的JVM参数。GC日志是一个很重要的工具，它准确记录了每一次的GC的执行时间和执行结果，通过分析GC日志可以优化堆设置和GC设置，或者改进应用程序的对象分配模式。

-XX:+PrintGC

参数-XX:+PrintGC（或者-verbose:gc）开启了简单GC日志模式，为每一次新生代（young generation）的GC和每一次的Full GC打印一行信息。下面举例说明：

|  |  |
| --- | --- |
| 1 | [GC 246656K->243120K(376320K), 0.0929090 secs] |
| 2 | [Full GC 243120K->241951K(629760K), 1.5589690 secs] | |

每行开始首先是GC的类型（可以是“GC”或者“Full GC”），然后是在GC之前和GC之后已使用的堆空间，再然后是当前的堆容量，最后是GC持续的时间（以秒计）。

第一行的意思就是GC将已使用的堆空间从246656K减少到243120K，当前的堆容量（译者注：GC发生时）是376320K，GC持续的时间是0.0929090秒。

简单模式的GC日志格式是与GC算法无关的，日志也没有提供太多的信息。在上面的例子中，我们甚至无法从日志中判断是否GC将一些对象从young generation移到了old generation。所以详细模式的GC日志更有用一些。

-XX:PrintGCDetails

如果不是使用-XX:+PrintGC，而是-XX:PrintGCDetails，就开启了详细GC日志模式。在这种模式下，日志格式和所使用的GC算法有关。我们首先看一下使用Throughput垃圾收集器在young generation中生成的日志。为了便于阅读这里将一行日志分为多行并使用缩进。

|  |  |
| --- | --- |
| 1 | [GC |
| 2 | [PSYoungGen: 142816K->10752K(142848K)] 246648K->243136K(375296K), 0.0935090 secs | |

|  |  |
| --- | --- |
| 3 | ] |
| 4 | [Times: user=0.55 sys=0.10, real=0.09 secs] | |

我们可以很容易发现：这是一次在young generation中的GC，它将已使用的堆空间从246648K减少到了243136K，用时0.0935090秒。此外我们还可以得到更多的信息：所使用的垃圾收集器（即PSYoungGen）、young generation的大小和使用情况（在这个例子中“PSYoungGen”垃圾收集器将young generation所使用的堆空间从142816K减少到10752K）。

既然我们已经知道了young generation的大小，所以很容易判定发生了GC，因为young generation无法分配更多的对象空间：已经使用了142848K中的142816K。我们可以进一步得出结论，多数从young generation移除的对象仍然在堆空间中，只是被移到了old generation：通过对比绿色的和蓝色的部分可以发现即使young generation几乎被完全清空（从142816K减少到10752K），但是所占用的堆空间仍然基本相同（从246648K到243136K）。

详细日志的“Times”部分包含了GC所使用的CPU时间信息，分别为操作系统的用户空间和系统空间所使用的时间。同时，它显示了GC运行的“真实”时间（0.09秒是0.0929090秒的近似值）。如果CPU时间（译者注：0.55秒+0.10秒）明显多于”真实“时间（译者注：0.09秒），我们可以得出结论：GC使用了多线程运行。这样的话CPU时间就是所有GC线程所花费的CPU时间的总和。实际上我们的例子中的垃圾收集器使用了8个线程。

接下来看一下Full GC的输出日志

|  |  |
| --- | --- |
| 1 | [Full GC |
| 2 | [PSYoungGen: 10752K->9707K(142848K)] | |

|  |  |  |
| --- | --- | --- |
| 3 | [ParOldGen: 232384K->232244K(485888K)] 243136K->241951K(628736K) | |
| 4 | [PSPermGen: 3162K->3161K(21504K)], 1.5265450 secs |

|  |  |
| --- | --- |
| 5 | ] |

除了关于young generation的详细信息，日志也提供了old generation和permanent generation的详细信息。对于这三个generations，一样也可以看到所使用的垃圾收集器、堆空间的大小、GC前后的堆使用情况。需要注意的是显示堆空间的大小等于young generation和old generation各自堆空间的和。以上面为例，堆空间总共占用了241951K，其中9707K在young generation，232244K在old generation。Full GC持续了大约1.53秒，用户空间的CPU执行时间为10.96秒，说明GC使用了多线程（和之前一样8个线程）。

对不同generation详细的日志可以让我们分析GC的原因，如果某个generation的日志显示在GC之前，堆空间几乎被占满，那么很有可能就是这个generation触发了GC。但是在上面的例子中，三个generation中的任何一个都不是这样的，在这种情况下是什么原因触发了GC呢。对于Throughput垃圾收集器，在某一个generation被过度使用之前，GC ergonomics（参考本系列第6节）决定要启动GC。

Full GC也可以通过显式的请求而触发，可以是通过应用程序，或者是一个外部的JVM接口。这样触发的GC可以很容易在日志里分辨出来，因为输出的日志是以“Full GC(System)”开头的，而不是“Full GC”。

对于Serial垃圾收集器，详细的GC日志和Throughput垃圾收集器是非常相似的。唯一的区别是不同的generation日志可能使用了不同的GC算法（例如：old generation的日志可能以Tenured开头，而不是ParOldGen）。使用垃圾收集器作为一行日志的开头可以方便我们从日志就判断出JVM的GC设置。

对于CMS垃圾收集器，young generation的详细日志也和Throughput垃圾收集器非常相似，但是old generation的日志却不是这样。对于CMS垃圾收集器，在old generation中的GC是在不同的时间片内与应用程序同时运行的。GC日志自然也和Full GC的日志不同。而且在不同时间片的日志夹杂着在此期间young generation的GC日志。但是了解了上面介绍的GC日志的基本元素，也不难理解在不同时间片内的日志。只是在解释GC运行时间时要特别注意，由于大多数时间片内的GC都是和应用程序同时运行的，所以和那种独占式的GC相比，GC的持续时间更长一些并不说明一定有问题。

正如我们在第7节中所了解的，即使CMS垃圾收集器没有完成一个CMS周期，Full GC也可能会发生。如果发生了GC，在日志中会包含触发Full GC的原因，例如众所周知的”concurrent mode failure“。

为了避免过于冗长，我这里就不详细说明CMS垃圾收集器的日志了。另外，CMS垃圾收集器的作者做了详细的说明（在这里），强烈建议阅读。

-XX:+PrintGCTimeStamps和-XX:+PrintGCDateStamps

使用-XX:+PrintGCTimeStamps可以将时间和日期也加到GC日志中。表示自JVM启动至今的时间戳会被添加到每一行中。例子如下：

|  |  |
| --- | --- |
| 1 | 0.185: [GC 66048K->53077K(251392K), 0.0977580 secs] |
| 2 | 0.323: [GC 119125K->114661K(317440K), 0.1448850 secs] | |

|  |  |
| --- | --- |
| 3 | 0.603: [GC 246757K->243133K(375296K), 0.2860800 secs] |

如果指定了-XX:+PrintGCDateStamps，每一行就添加上了绝对的日期和时间。

|  |  |
| --- | --- |
| 1 | 2014-01-03T12:08:38.102-0100: [GC 66048K->53077K(251392K), 0.0959470 secs] |
| 2 | 2014-01-03T12:08:38.239-0100: [GC 119125K->114661K(317440K), 0.1421720 secs] |

|  |  |
| --- | --- |
| 3 | 2014-01-03T12:08:38.513-0100: [GC 246757K->243133K(375296K), 0.2761000 secs] |

如果需要也可以同时使用两个参数。推荐同时使用这两个参数，因为这样在关联不同来源的GC日志时很有帮助。

-Xloggc

缺省的GC日志时输出到终端的，使用-Xloggc:也可以输出到指定的文件。需要注意这个参数隐式的设置了参数-XX:+PrintGC和-XX:+PrintGCTimeStamps，但为了以防在新版本的JVM中有任何变化，我仍建议显示的设置这些参数。

可管理的JVM参数

一个常常被讨论的问题是在生产环境中GC日志是否应该开启。因为它所产生的开销通常都非常有限，因此我的答案是需要开启。但并不一定在启动JVM时就必须指定GC日志参数。

HotSpot JVM有一类特别的参数叫做可管理的参数。对于这些参数，可以在运行时修改他们的值。我们这里所讨论的所有参数以及以“PrintGC”开头的参数都是可管理的参数。这样在任何时候我们都可以开启或是关闭GC日志。比如我们可以使用JDK自带的jinfo工具来设置这些参数，或者是通过JMX客户端调用HotSpotDiagnostic MXBean的setVMOption方法来设置这些参数。

***原创文章，转载请注明：****转载自*[*并发编程网 – ifeve.com*](http://ifeve.com/)

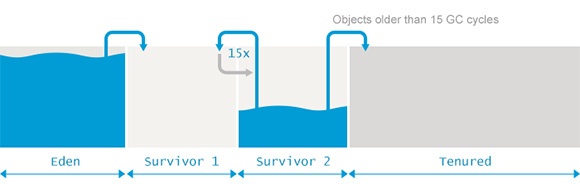
***本文链接地址:***[*JVM实用参数（八）GC日志*](http://ifeve.com/useful-jvm-flags-part-8-gc-logging/)

### Full GC是否真的存在 - Java译站

http://www.deepinmind.com/gc/2015/03/03/minor-gc-vs-major-gc-vs-full-gc.html

在Plumbr这和GC暂停检测打交道的这段日子里，我查阅了与这个主题相关的大量文章，书籍以及资料。在这当中，我经常会对新生代GC, 年老代GC以及Full GC的事件的使用（滥用）感到困惑。于是便有了这篇文章，希望能够清除一些困惑。

本文需要读者对JVM内建的GC相关的常用原理有一定的了解。像eden区，Survivor区以及年老区空间的划分，分代假设（generational hypothesis）以及不同的GC算法就不在本文的讨论范围之内了。



新生代GC（Minor GC）

新生代垃圾的回收被称作**Minor GC**。这个定义非常清晰，理解起来也不会有什么歧义。不过当处理新生代GC事件时，还是有一些有意思的东西值得注意的：

* 只要JVM无法为新创建的对象分配空间，就肯定会触发新生代GC，比方说Eden区满了。因此对象创建得越频繁，新生代GC肯定也更频繁。
* 一旦内存池满了，它的所有内容就会被拷贝走，指针又将重新归零。因此和经典的标记(Mark)，清除(Sweep)，整理(Compact)的过程不同的是，Eden区和Survivor区的清理只涉及到标记和拷贝。在它们中是不会出现碎片的。写指针始终在当前使用区的顶部。
* 在一次新生代GC事件中，通常不涉及到年老代。年老代到年轻代的引用被认为是GC的根对象。而在标记阶段中，从年轻代到年老代的引用则会被忽略掉。
* **和通常所理解的不一样的是，所有的新生代GC都会触发“stop-the-world”暂停**，这会中断应用程序的线程。对绝大多数应用而言，暂停的时间是可以忽略不计的。如果Eden区中的大多数对象都是垃圾对象并且永远不会被拷贝到Survivor区/年老代中的话，这么做是合理的。如果恰好相反的话，那么绝大多数的新生对象都不应该被回收，新生代GC的暂停时间就会变得相对较长了。

现在来看新生代GC还是很清晰的——**每一次新生代GC都会对年轻代进行垃圾清除**。

年老代GC（ajor GC）与Full GC

你会发现关于这两种GC其实并没有明确的定义。JVM规范或者垃圾回收相关的论文中都没有提及。不过从直觉来说，根据新生代GC(Minor GC)清理的是年轻代空间的认识来看，不难得出以下推论（这里应当从英文出发来理解，Minor, Major与Full GC，翻译过来的名称已经带有明显的释义了）：

* **Major GC**清理的是年老代的空间。
* **Full GC**清理的是整个堆——包括年轻代与年老代空间

不幸的是这么理解会有一点复杂与困惑。首先——许多年老代GC其实是由新生代GC触发的，因此在很多情况下两者无法孤立来看待。另一方面——许多现代的垃圾回收器会对年老代进行部分清理，因此，使用“清理”这个术语则显得有点牵强。

那么问题就来了，先别再纠结某次GC到底是年老代GC还是Full GC了，**你应该关注的是这次GC是否中断了应用线程还是能够和应用线程并发地执行**。

即便是在JVM的官方工具中，也存在着这一困扰。通过一个例子来说明应该更容易理解一些。我们用两款工具来跟踪某个运行着CMS回收器的JVM，来比较下它们的输出有什么不同：

首先通过[jstat](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html)的输出来查看下GC的信息：

my-precious: me$ jstat -gc -t 4235 1s

Time S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT

5.7 34048.0 34048.0 0.0 34048.0 272640.0 194699.7 1756416.0 181419.9 18304.0 17865.1 2688.0 2497.6 3 0.275 0 0.000 0.275

6.7 34048.0 34048.0 34048.0 0.0 272640.0 247555.4 1756416.0 263447.9 18816.0 18123.3 2688.0 2523.1 4 0.359 0 0.000 0.359

7.7 34048.0 34048.0 0.0 34048.0 272640.0 257729.3 1756416.0 345109.8 19072.0 18396.6 2688.0 2550.3 5 0.451 0 0.000 0.451

8.7 34048.0 34048.0 34048.0 34048.0 272640.0 272640.0 1756416.0 444982.5 19456.0 18681.3 2816.0 2575.8 7 0.550 0 0.000 0.550

9.7 34048.0 34048.0 34046.7 0.0 272640.0 16777.0 1756416.0 587906.3 20096.0 19235.1 2944.0 2631.8 8 0.720 0 0.000 0.720

10.7 34048.0 34048.0 0.0 34046.2 272640.0 80171.6 1756416.0 664913.4 20352.0 19495.9 2944.0 2657.4 9 0.810 0 0.000 0.810

11.7 34048.0 34048.0 34048.0 0.0 272640.0 129480.8 1756416.0 745100.2 20608.0 19704.5 2944.0 2678.4 10 0.896 0 0.000 0.896

12.7 34048.0 34048.0 0.0 34046.6 272640.0 164070.7 1756416.0 822073.7 20992.0 19937.1 3072.0 2702.8 11 0.978 0 0.000 0.978

13.7 34048.0 34048.0 34048.0 0.0 272640.0 211949.9 1756416.0 897364.4 21248.0 20179.6 3072.0 2728.1 12 1.087 1 0.004 1.091

14.7 34048.0 34048.0 0.0 34047.1 272640.0 245801.5 1756416.0 597362.6 21504.0 20390.6 3072.0 2750.3 13 1.183 2 0.050 1.233

15.7 34048.0 34048.0 0.0 34048.0 272640.0 21474.1 1756416.0 757347.0 22012.0 20792.0 3200.0 2791.0 15 1.336 2 0.050 1.386

16.7 34048.0 34048.0 34047.0 0.0 272640.0 48378.0 1756416.0 838594.4 22268.0 21003.5 3200.0 2813.2 16 1.433 2 0.050 1.484

这段输出是从JVM启动后第17秒开始截取的。从中可以看出，在经过了12次新生代GC后出现了两次Full GC，共耗时50ms。通过GUI的工具也可以获取到同样的信息，比如说[jsonsole](http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html)或者是[jvisualvm](http://download.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html)。

在接受这一结论前，我们再来看下同样是这次JVM启动后所输出的GC日志。很明显-XX:+PrintGCDetails给我们讲述的是一段截然不同却更为详尽的故事：

java -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC eu.plumbr.demo.GarbageProducer

3.157: [GC (Allocation Failure) 3.157: [ParNew: 272640K->34048K(306688K), 0.0844702 secs] 272640K->69574K(2063104K), 0.0845560 secs] [Times: user=0.23 sys=0.03, real=0.09 secs]

4.092: [GC (Allocation Failure) 4.092: [ParNew: 306688K->34048K(306688K), 0.1013723 secs] 342214K->136584K(2063104K), 0.1014307 secs] [Times: user=0.25 sys=0.05, real=0.10 secs]

... cut **for** brevity ...

11.292: [GC (Allocation Failure) 11.292: [ParNew: 306686K->34048K(306688K), 0.0857219 secs] 971599K->779148K(2063104K), 0.0857875 secs] [Times: user=0.26 sys=0.04, real=0.09 secs]

12.140: [GC (Allocation Failure) 12.140: [ParNew: 306688K->34046K(306688K), 0.0821774 secs] 1051788K->856120K(2063104K), 0.0822400 secs] [Times: user=0.25 sys=0.03, real=0.08 secs]

12.989: [GC (Allocation Failure) 12.989: [ParNew: 306686K->34048K(306688K), 0.1086667 secs] 1128760K->931412K(2063104K), 0.1087416 secs] [Times: user=0.24 sys=0.04, real=0.11 secs]

13.098: [GC (CMS Initial Mark) [1 CMS-initial-mark: 897364K(1756416K)] 936667K(2063104K), 0.0041705 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]

13.102: [CMS-concurrent-mark-start]

13.341: [CMS-concurrent-mark: 0.238/0.238 secs] [Times: user=0.36 sys=0.01, real=0.24 secs]

13.341: [CMS-concurrent-preclean-start]

13.350: [CMS-concurrent-preclean: 0.009/0.009 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

13.350: [CMS-concurrent-abortable-preclean-start]

13.878: [GC (Allocation Failure) 13.878: [ParNew: 306688K->34047K(306688K), 0.0960456 secs] 1204052K->1010638K(2063104K), 0.0961542 secs] [Times: user=0.29 sys=0.04, real=0.09 secs]

14.366: [CMS-concurrent-abortable-preclean: 0.917/1.016 secs] [Times: user=2.22 sys=0.07, real=1.01 secs]

14.366: [GC (CMS Final Remark) [YG occupancy: 182593 K (306688 K)]14.366: [Rescan (parallel) , 0.0291598 secs]14.395: [weak refs processing, 0.0000232 secs]14.395: [**class** **unloading**, 0.0117661 secs]14.407: [scrub symbol table, 0.0015323 secs]14.409: [scrub string table, 0.0003221 secs][1 CMS-remark: 976591K(1756416K)] 1159184K(2063104K), 0.0462010 secs] [Times: user=0.14 sys=0.00, real=0.05 secs]

14.412: [CMS-concurrent-sweep-start]

14.633: [CMS-concurrent-sweep: 0.221/0.221 secs] [Times: user=0.37 sys=0.00, real=0.22 secs]

14.633: [CMS-concurrent-reset-start]

14.636: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

从以上能够看出，在运行了12次新生代GC后的确出现了一些“不太寻常”的事情。但并不是执行了两次Full GC，这个“不寻常”的事情其实只是在年老代中执行了一次包含了数个阶段的GC而已：

* 初始标记阶段，从0.0041705 秒或者说4ms的时候开始。这一阶段是一次“stop-the-world”事件，所有的应用线程都被暂停以便进行初始标记。
* 并发地执行标记和预清理(Preclean)的阶段。这是和应用线程一起并发执行的。
* 最终标记阶段，从0.0462010秒或者说46毫秒的时候开始。这一阶段也同样是“stop-the-world”的。
* 并发地进行清除操作。正如名字所说的，这一阶段也无需中断应用线程，可以并发地执行。

因此我们从实际的GC日志中所看到的是这样——其实没有什么两次所谓的Full GC,只有一次清理年老代空间的Major GC而已。

如果你再想想jstat所输出的结果再下结论的话，就能得出正确的结论了。它明确地列出了两次stop-the-world事件，总耗时50ms，这是影响到活跃线程的总的延迟时间。不过如果你想依据这来优化吞吐量的话，你可能就被误导了——jstat只列出了两次stop-the-world的初始标记和最终标记的阶段，它把并发执行的那部分工作给隐藏掉了。

结论

基于上述事实来看，最好就是不要再考虑什么新生代GC，年老代GC，Full GC。你应该做的是监控应用的时延和吞吐量，并把GC事件与之结合来看。从这些事件中，你便能得知某个GC事件是否中断了应用程序或者只是并发地执行而已。

原创文章转载请注明出处：[Full GC是否真的存在](http://www.deepinmind.com/gc/2015/03/03/minor-gc-vs-major-gc-vs-full-gc.html)

[英文原文链接](https://plumbr.eu/blog/minor-gc-vs-major-gc-vs-full-gc?utm_source=feedly&utm_reader=feedly&utm_medium=rss&utm_campaign=rss)