JVM实用参数系列

极客学院出版

前言

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，Java 通过使用 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 具备了一次编写，多处运行的特性。JVM 一直是 Java 学习中的重点，也是难点。本教程旨在帮助大家了解 JVM 的结构以及相关参数。JVM 实用参数系列一共包括八篇文章，由浅入深，从编译器、垃圾回收、内存调优等方面介绍 JVM。

适用人群

本教程是 Java 中高级教程，能够帮助开发者对 Java 虚拟机有一个全新的认识。

学习前提

学习本教程前，你需要对 Java 这门语言有了一定的了解。

版本信息

书中演示代码基于以下版本：

语言/框架	版本信息
Java	Java 6

鸣谢：http://ifeve.com/useful-jvm-flags/

目录

1

JVM 类型以及编译器模式

原文地址：https://blog.codecentric.de/en/2012/07/useful-jvm-flags-part-1-jvm-types-and-compiler-modes/

现在的 JVM 运行 Java 程序（和其它的兼容性语言）时在高效性和稳定性方面做的非常出色。自适应内存管理、垃圾收集、及时编译、动态类加载、锁优化——这里仅仅列举了某些场景下会发生的神奇的事情，但他们几乎不会直接与普通的程序员相关。在运行时，JVM 会不断的计算并优化应用或者应用的某些部分。

虽然有了这种程度的自动化（或者说有这么多自动化），但是 JVM 仍然提供了足够多的外部监控和手动调优工具。在有错误或低性能的情况下，JVM 必须能够让专家调试。顺便说一句，除了这些隐藏在引擎中的神奇功能，允许大范围的手动调优也是现代 JVM 的优势之一。有趣的是，一些命令行参数可以在 JVM 启动时传入到 JVM 中。一些 JVM 提供了几百个这样的参数，所以如果没有这方面的知识很容易迷失。这系列博客的目标是着重讲解日常相关的一些参数以及他们的适用场合。我们将专注于 Java6 的 Sun/Oracle HotSpot jvm，大多数情况下，这些参数也会适用于其他一些流行的 JVM 里。

-server and -client

有两种类型的 HotSpot JVM，即“server” 和“client”。服务端的 VM 中的默认为堆提供了一个更大的空间以及一个并行的垃圾收集器，并且在运行时可以更大程度地优化代码。客户端的 VM 更加保守一些（校对注：这里作者指客户端虚拟机有较小的默认堆大小），这样可以缩短 JVM 的启动时间和占用更少的内存。有一个叫”JVM 功效学” 的概念，它会在 JVM 启动的时候根据可用的硬件和操作系统来自动的选择 JVM 的类型。具体的标准可以在这里找到。从标准表中，我们可以看到客户端的 VM 只在 32 位系统中可用。

如果我们不喜欢预选（校对注：指 JVM 自动选择的 JVM 类型）的 JVM，我们可以使用 - server 和 - client 参数来设置使用服务端或客户端的 VM。虽然当初服务端 VM 的目标是长时间运行的服务进程，但是现在看来，在运行独立应用程序时它比客户端 VM 有更出色的性能。当应用的性能非常重要时，我推荐使用 - server 参数来选择服务端 VM。一个常见的问题：在一个 32 位的系统上，HotSpot JDK 可以运行服务端 VM，但是 32 位的 JRE 只能运行客户端 VM。

-version and -showversion

当我们调用 “java” 命令时，我们如何才能知道我们安装的是哪个版本的 Java 和 JVM 类型呢？在同一个系统中安装多个 Java，如果不注意的话有运行错误 JVM 的风险。在不同的 Linux 版本上预装 JVM 这方面，我承认现在已经变的比以前好很多了。幸运的是，我们现在可以使用 - version 参数，它可以打印出正在使用的 JVM 的信息。例如：

```
$ java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Client VM (build 19.1-b02, mixed mode, sharing)
```

输出显示的是 Java 版本号 (1.6.0_24) 和 JRE 确切的 build 号 (1.6.0_24-b07)。我们也可以看到 JVM 的名字 (HotSpot)、类型 (client) 和 build ID（19.1-b02））。除此之外，我们还知道 JVM 以混合模式 (mixed mode) 在运行，这是 HotSpot 默认的运行模式，意味着 JVM 在运行时可以动态的把字节码编译为本地代码。我们也可以看到类数据共享（class data sharing）是开启的，类数据共享（class data sharing）是一种在只读缓存（在 jsa 文件中，”Java Shared Archive”）中存储 JRE 的系统类，被所有 Java 进程的类加载器用来当做共享资源。类数据共享 (Class data sharing) 可能在经常从 jar 文档中读所有的类数据的情况下显示出性能优势。

-version 参数在打印完上述信息后立即终止 JVM。还有一个类似的参数 - showversion 可以用来输出相同的信息，但是 - showversion 紧接着会处理并执行 Java 程序。因此，-showversion 对几乎所有 Java 应用的命令行都是一个有效的补充。你永远不知道你什么时候，突然需要了解一个特定的 Java 应用（崩溃时）使用的 JVM 的一些信息。在启动时添加 -showversion，我们就能保证当我们需要时可以得到这些信息。

-Xint, -Xcomp, 和 -Xmixed

-Xint 和 -Xcomp 参数和我们的日常工作不是很相关，但是我非常有兴趣通过它来了解下 JVM。在解释模式 (interpreted mode) 下，-Xint 标记会强制 JVM 执行所有的字节码，当然这会降低运行速度，通常低 10 倍或更多。-Xcomp 参数与它（-Xint）正好相反，JVM 在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。这听起来不错，因为这完全绕开了缓慢的解释器。然而，很多应用在使用 - Xcomp 也会有一些性能损失，当然这比使用 - Xint 损失的少，原因是 - xcomp 没有让 JVM 启用 JIT 编译器的全部功能。JIT 编译器在运行时创建方法使用文件，然后一步一步的优化每一个方法，有时候会主动的优化应用的行为。这些优化技术，比如，积极的分支预测（optimistic branch prediction），如果不先分析应用就不能有效的使用。另一方面方法只有证明它们与此相关时才会被编译，也就是，在应用中构建某种热点。被调用很少（甚至只有一次）的方法在解释模式下会继续执行，从而减少编译和优化成本。

注意混合模式也有他自己的参数，-Xmixed。最新版本的 HotSpot 的默认模式是混合模式，所以我们不需要特别指定这个标记。我们来用对象填充 HashMap 然后检索它的结果做一个简单的用例。每一个例子，它的运行时间都是很多次运行的平均时间。

```
$ java -server -showversion Benchmark
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Server VM (build 19.1-b02, mixed mode)

Average time: 0.856449 seconds
```

```
$ java -server -showversion -Xcomp Benchmark
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Server VM (build 19.1-b02, compiled mode)
```

```
Average time: 0.950892 seconds
```

```
$ java -server -showversion -Xint Benchmark
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Server VM (build 19.1-b02, interpreted mode)

Average time: 7.622285 seconds
```

当然也有很多使 - Xcomp 表现很好的例子。特别是运行时间长的应用，我强烈建议大家使用 JVM 的默认设置，让 JIT 编译器充分发挥其动态潜力，毕竟 JIT 编译器是组成 JVM 最重要的组件之一。事实上，正是因为 JVM 在这方面的进展才让 Java 不再那么慢。

2

参数分类和即时（JIT）编译器诊断

作者： PATRICK PESCHLOW 原文地址 译者：赵峰 校对：许巧辉

在这个系列的第二部分，我来介绍一下 HotSpot JVM 提供的不同类别的参数。我同样会讨论一些关于 JIT 编译器诊断的有趣参数。

JVM 参数分类

HotSpot JVM 提供了三类参数。第一类包括了标准参数。顾名思义，标准参数中包括功能和输出的参数都是很稳定的，很可能在将来的 JVM 版本中不会改变。你可以用 java 命令（或者是用 java -help）检索出所有标准参数。我们在第一部分中已经见到过一些标准参数，例如：-server。

第二类是 X 参数，非标准化的参数在将来的版本中可能会改变。所有的这类参数都以 - X 开始，并且可以用 java -X 来检索。注意，不能保证所有参数都可以被检索出来，其中就没有 - Xcomp。

第三类是包含 XX 参数（到目前为止最多的），它们同样不是标准的，甚至很长一段时间内不被列出来（最近，这种情况有改变，我们将在本系列的第三部分中讨论它们）。然而，在实际情况中 X 参数和 XX 参数并没有什么不同。X 参数的功能是十分稳定的，然而很多 XX 参数仍在实验当中（主要是 JVM 的开发者用于 debugging 和调优 JVM 自身的实现）。值的一读的介绍非标准参数的文档 HotSpot JVM documentation，其中明确的指出 XX 参数不应该在不了解的情况下使用。这是真的，并且我认为这个建议同样适用于 X 参数（同样一些标准参数也是）。不管类别是什么，在使用参数之前应该先了解它可能产生的影响。

用一句话来说明 XX 参数的语法。所有的 XX 参数都以”-XX:” 开始，但是随后的语法不同，取决于参数的类型。

对于布尔类型的参数，我们有”+” 或”-“，然后才设置 JVM 选项的实际名称。例如，-XX:+ 用于激活 选项，而 - XX:- 用于注销选项。 对于需要非布尔值的参数，如 string 或者 integer，我们先写参数的名称，后面加上”=”，最后赋值。例如， -XX:= 给 赋值 。 现在让我们来看看 JIT 编译方面的一些 XX 参数。

-XX:+PrintCompilation and -XX:+CITime

当一个 Java 应用运行时，非常容易查看 JIT 编译工作。通过设置 - XX:+PrintCompilation，我们可以简单的输出一些关于从字节码转化成本地代码的编译过程。我们来看一个服务端 VM 运行的例子：

```
$ java -server -XX:+PrintCompilation Benchmark
  1       java.lang.String::hashCode (64 bytes)
  2       java.lang.AbstractStringBuilder::stringSizeOfInt (21 bytes)
  3       java.lang.Integer::getChars (131 bytes)
  4       java.lang.Object::<init> (1 bytes)
---   n   java.lang.System::arraycopy (static)
  5       java.util.HashMap::indexFor (6 bytes)
  6       java.lang.Math::min (11 bytes)
  7       java.lang.String::getChars (66 bytes)
```

```
 8      java.lang.AbstractStringBuilder::append (60 bytes)
 9      java.lang.String::<init> (72 bytes)
 10     java.util.Arrays::copyOfRange (63 bytes)
 11     java.lang.StringBuilder::append (8 bytes)
 12     java.lang.AbstractStringBuilder::<init> (12 bytes)
 13     java.lang.StringBuilder::toString (17 bytes)
 14     java.lang.StringBuilder::<init> (18 bytes)
 15     java.lang.StringBuilder::append (8 bytes)
[...]
 29     java.util.regex.Matcher::reset (83 bytes)
```

每当一个方法被编译，就输出一行 - XX:+PrintCompilation。每行都包含顺序号（唯一的编译任务 ID）和已编译方法的名称和大小。因此，顺序号 1，代表编译 String 类中的 hashCode 方法到原生代码的信息。根据方法的类型和编译任务打印额外的信息。例如，本地的包装方法前方会有”n” 参数，像上面的 System::arraycopy 一样。注意这样的方法不会包含顺序号和方法占用的大小，因为它不需要编译为本地代码。同样可以看到被重复编译的方法，例如 StringBuilder::append 顺序号为 11 和 15。输出在顺序号 29 时停止 ，这表明在这个 Java 应用运行时总共需要编译 29 个方法。

没有官方的文档关于 - XX:+PrintCompilation，但是这个描述是对于此参数比较好的。我推荐更深入学习一下。

JIT 编译器输出帮助我们理解客户端 VM 与服务端 VM 的一些区别。用服务端 VM，我们的应用例子输出了 29 行，同样用客户端 VM，我们会得到 55 行。这看起来可能很怪，因为服务端 VM 应该比客户端 VM 做了 “更多” 的编译。然而，由于它们各自的默认设置，服务端 VM 在判断方法是不是热点和需不需要编译时比客户端 VM 观察方法的时间更长。因此，在使用服务端 VM 时，一些潜在的方法会稍后编译就不奇怪了。

通过另外设置 - XX:+CITime，我们可以在 JVM 关闭时得到各种编译的统计信息。让我们看一下一个特定部分的统计：

```
$ java -server -XX:+CITime Benchmark
[...]
## Accumulated compiler times (for compiled methods only)  Total compilation time   :  0.178 s
   Standard compilation   :  0.129 s, Average : 0.004
   On stack replacement   :  0.049 s, Average : 0.024
[...]
```

总共用了 0.178 s（在 29 个编译任务上）。这些，”on stack replacement” 占用了 0.049 s，即编译的方法目前在堆栈上用去的时间。这种技术并不是简单的实现性能显示，实际上它是非常重要的。没有”on stack replacement”，方法如果要执行很长时间（比如，它们包含了一个长时间运行的循环），它们运行时将不会被它们编译过的副本替换。

再一次，客户端 VM 与服务端 VM 的比较是非常有趣的。客户端 VM 相应的数据表明，即使有 55 个方法被编译了，但这些编译总共用了只有 0.021 s。服务端 VM 做的编译少但是用的时间却比客户端 VM 多。这个原因是，使用服务端 VM 在生成本地代码时执行了更多的优化。

在本系列的第一部分，我们已经学了 - Xint 和 - Xcomp 参数。结合使用 - XX:+PrintCompilation 和 - XX:+CITime，在这两个情况下（校对者注，客户端 VM 与服务端 VM），我们能对 JIT 编译器的行为有更好的了解。使用 - Xint，-XX:+PrintCompilation 在这两种情况下会产生 0 行输出。同样的，使用 - XX:+CITime 时，证实在编译上没有花费时间。现在换用 - Xcomp，输出就完全不同了。在使用客户端 VM 时会产生 726 行输出，然后没有更多的，这是因为每个相关的方法都被编译了。使用服务端 VM，我们甚至能得到 993 行输出，这告诉我们更积极的优化被执行了。同样，JVM 拆机 (JVM teardown) 时打印出的统计显示了两个 VM 的巨大不同。考虑服务端 VM 的运行：

```
$ java -server -Xcomp -XX:+CITime Benchmark
[...]
## Accumulated compiler times (for compiled methods only)  Total compilation time   : 1.567 s
   Standard compilation   : 1.567 s, Average : 0.002
   On stack replacement   : 0.000 s, Average : -1.#IO
[...]
```

使用 - Xcomp 编译用了 1.567 s，这是使用默认设置（即，混合模式）的 10 倍。同样，应用程序的运行速度要比用混合模式的慢。相比较之下，客户端 VM 使用 - Xcomp 编译 726 个方法只用了 0.208 s，甚至低于使用 - Xcomp 的服务端 VM。

补充一点，这里没有”on stack replacement” 发生，因为每一个方法在第一次调用时被编译了。损坏的输出“Average: -1.#IO”（正确的是: 0）再一次表明了，非标准化的输出参数不是非常可靠。

-XX:+UnlockExperimentalVMOptions

有些时候当设置一个特定的 JVM 参数时，JVM 会在输出 “Unrecognized VM option” 后终止。如果发生了这种情况，你应该首先检查你是否输错了参数。然而，如果参数输入是正确的，并且 JVM 并不识别，你或许需要设置 - XX:+UnlockExperimentalVMOptions 来解锁参数。我不是非常清楚这个安全机制的作用，但我猜想这个参数如果不正确使用可能会对 JVM 的稳定性有影响（例如，他们可能会过多的写入 debug 输出的一些日志文件）。

有一些参数只是在 JVM 开发时用，并不实际用于 Java 应用。如果一个参数不能被 -XX:+UnlockExperimentalVMOptions 开启，但是你真的需要使用它，此时你可以尝试使用 debug 版本的 JVM。

-XX:+LogCompilation and -XX:+PrintOptoAssembly

如果你在一个场景中发现使用 -XX:+PrintCompilation，不能够给你足够详细的信息，你可以使用 -XX:+LogCompilation 把扩展的编译输出写到 “hotspot.log” 文件中。除了编译方法的很多细节之外，你也可以看到编译

器线程启动的任务。注意 – XX:+LogCompilation 需要使用 – XX:+UnlockExperimentalVMOptions 来解锁。

JVM 甚至允许我们看到从字节码编译生成到本地代码。使用 – XX:+PrintOptoAssembly，由编译器线程生成的本地代码被输出并写到 “hotspot.log” 文件中。使用这个参数要求运行的服务端 VM 是 debug 版本。我们可以研究 – XX:+PrintOptoAssembly 的输出，以至于了解 JVM 实际执行什么样的优化，例如，关于死代码的消除。一个非常有趣的文章提供了一个例子。

关于 XX 参数的更多信息

如果这篇文章勾起了你的兴趣，你可以自己看一下 HotSpot JVM 的 XX 参数。这里是一个很好的起点。

3

打印所有 XX 参数及值

原文地址：https://blog.codecentric.de/en/2012/07/useful-jvm-flags-part-3-printing-all-xx-flags-and-their-values/

本篇文章基于 Java 6（update 21oder 21 之后）版本， HotSpot JVM 提供给了两个新的参数，在 JVM 启动后，在命令行中可以输出所有 XX 参数和值。

`-XX:+PrintFlagsFinal and -XX:+PrintFlagsInitial`

让我们现在就了解一下新参数的输出。以 -client 作为参数的 -XX:+PrintFlagsFinal 的结果是一个按字母排序的 590 个参数表格（注意，每个 release 版本参数的数量会不一样）

```
$ java -client -XX:+PrintFlagsFinal Benchmark
[Global flags]
uintx AdaptivePermSizeWeight              = 20            {product}
uintx AdaptiveSizeDecrementScaleFactor    = 4             {product}
uintx AdaptiveSizeMajorGCDecayTimeScale   = 10            {product}
uintx AdaptiveSizePausePolicy             = 0             {product}[...]
uintx YoungGenerationSizeSupplementDecay  = 8             {product}
uintx YoungPLABSize                       = 4096          {product}
 bool ZeroTLAB                            = false         {product}
 intx hashCode                            = 0             {product}
```

(校对注：你可以尝试在命令行输入上面的命令，亲自实现下)

表格的每一行包括五列，来表示一个 XX 参数。第一列表示参数的数据类型，第二列是名称，第四列为值，第五列是参数的类别。第三列”=” 表示第四列是参数的默认值，而“:=” 表明了参数被用户或者 JVM 赋值了。

注意对于这个例子我只是用了 Benchmark 类，因为这个系列前面的章节也是用的这个类。甚至没有一个主类的情况下你能得到相同的输出，通过运行 java 带另外的参数 -version. 现在让我们检查下 server VM 提供了多少个参数。我们也能指定参数 -XX:+UnlockExperimentalVMOptions 和 -XX:+UnlockDiagnosticVMOptions；来解锁任何额外的隐藏参数。

```
$ java -server -XX:+UnlockExperimentalVMOptions -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal Benchma
```

724 个参数，让我们看一眼那些已经被赋值的参数。

```
$ java -server -XX:+UnlockExperimentalVMOptions -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal Benchma
uintx InitialHeapSize                   := 57505088       {product}
uintx MaxHeapSize                       := 920649728      {product}
uintx ParallelGCThreads                 := 4              {product}
 bool PrintFlagsFinal                   := true           {product}
 bool UseParallelGC                     := true           {product}
```

（校对注：这个命令非常有用）我们仅设置一个自己的参数 -XX:+PrintFlagsFinal。其他参数通过 server VM 基于系统设置的，以便以合适的堆大小和 GC 设置运行。

如果我们只想看下所有 XX 参数的默认值，能够用一个相关的参数，-XX:+PrintFlagsInitial 。 用 -XX:+PrintFlagsInitial, 只是展示了第三列为 “=” 的数据（也包括那些被设置其他值的参数）。

然而，注意当与 - XX:+PrintFlagsFinal 对比的时候，一些参数会丢失，大概因为这些参数是动态创建的。

研究表格的内容是很有意思的，通过比较 client 和 server VM 的行为，很明显了解哪些参数会影响其他的参数。有兴趣的读者，可以看一下这篇不错文章 Inspecting HotSpot JVM Options。这个文章主要解释了第五列的参数类别。

-XX:+PrintCommandLineFlags

让我们看下另外一个参数，事实上这个参数非常有用: -XX:+PrintCommandLineFlags。这个参数让 JVM 打印出那些已经被用户或者 JVM 设置过的详细的 XX 参数的名称和值。

换句话说，它列举出 -XX:+PrintFlagsFinal的结果中第三列有“:=”的参数。以这种方式，我们可以用 - XX:+PrintCommandLineFlags 作为快捷方式来查看修改过的参数。看下面的例子。

```
$ java -server -XX:+PrintCommandLineFlags Benchmark
-XX:InitialHeapSize=57505088 -XX:MaxHeapSize=920081408 -XX:ParallelGCThreads=4 -XX:+PrintCommandLineFlag
```

现在如果我们每次启动 java 程序的时候设置 -XX:+PrintCommandLineFlags 并且输出到日志文件上，这样会记录下我们设置的 JVM 参数对应用程序性能的影响。类似于 -showversion(见 Part1)，我建议 - XX:+PrintCommandLineFlags 这个参数应该总是设置在 JVM 启动的配置项里。因为你从不知道你什么时候会需要这些信息。

奇怪的是在这个例子中，通过 -XX:+PrintCommandLineFlags 列出堆的最大值会比通过 - XX:+PrintFlagsFinal 列举出的相应值小一点。如果谁知道两者之间不同的原因，请告诉我。

4

内存调优

原文地址，译文地址，作者：PATRICK PESCHLOW，译者：郑旭东 校对：梁海舰

理想的情况下，一个 Java 程序使用 JVM 的默认设置也可以运行得很好，所以一般来说，没有必要设置任何 JVM 参数。然而，由于一些性能问题（很不幸的是，这些问题经常出现），一些相关的 JVM 参数知识会是我们工作中得好伙伴。在这篇文章中，我们将介绍一些关于 JVM 内存管理的参数。知道并理解这些参数，将对开发者和运维人员很有帮助。

所有已制定的 HotSpot 内存管理和垃圾回收算法都基于一个相同的堆内存划分：新生代（young generation）里存储着新分配的和较年轻的对象，老年代（old generation）里存储着长寿的对象。在此之外，永久代（permanent generation）存储着那些需要伴随整个 JVM 生命周期的对象，比如，已加载的对象的类定义或者 String 对象内部 Cache。接下来，我们将假设堆内存是按照新生代、老年代和永久代这一经典策略划分的。然而，其他的一些堆内存划分策略也是可行的，一个突出的例子就是新的 G1 垃圾回收器，它模糊了新生代和老年代之间的区别。此外，目前的开发进程似乎表明在未来的 HotSpot JVM 版本中，将不会区分老年代和永久代。

-Xms and -Xmx (or: -XX:InitialHeapSize and -XX:MaxHeapSize)

-Xms 和 - Xmx 可以说是最流行的 JVM 参数，它们可以允许我们指定 JVM 的初始和最大堆内存大小。一般来说，这两个参数的数值单位是 Byte，但同时它们也支持使用速记符号，比如 “k” 或者 “K” 代表 “kilo”，“m” 或者 “M” 代表 “mega”，“g” 或者 “G” 代表 “giga”。举个例子，下面的命令启动了一个初始化堆内存为 128M，最大堆内存为 2G，名叫 “MyApp” 的 Java 应用程序。

```
java -Xms128m -Xmx2g MyApp
```

在实际使用过程中，初始化堆内存的大小通常被视为堆内存大小的下界。然而 JVM 可以在运行时动态的调整堆内存的大小，所以理论上来说我们有可能会看到堆内存的大小小于初始化堆内存的大小。但是即使在非常低的堆内存使用下，我也从来没有遇到过这种情况。这种行为将会方便开发者和系统管理员，因为我们可以通过将 “-Xms” 和 “-Xmx” 设置为相同大小来获得一个固定大小的堆内存。 -Xms 和 - Xmx 实际上是 - XX:InitialHeapSize 和 - XX:MaxHeapSize 的缩写。我们也可以直接使用这两个参数，它们所起得效果是一样的：

```
$ java -XX:InitialHeapSize=128m -XX:MaxHeapSize=2g MyApp
```

需要注意的是，所有 JVM 关于初始 \ 最大堆内存大小的输出都是使用它们的完整名称：“InitialHeapSize” 和 “InitialHeapSize”。所以当你查询一个正在运行的 JVM 的堆内存大小时，如使用 - XX:+PrintCommandLineFlags 参数或者通过 JMX 查询，你应该寻找 “InitialHeapSize” 和 “InitialHeapSize” 标志而不是 “Xms” 和 “Xmx”。

-XX:+HeapDumpOnOutOfMemoryError and -XX:HeapDumpPath

如果我们没法为 - Xmx（最大堆内存）设置一个合适的大小，那么就有可能面临内存溢出（OutOfMemoryError）的风险，这可能是我们使用 JVM 时面临的最可怕的猛兽之一。就同另外一篇关于这个主题的博文说的一

样，导致内存溢出的根本原因需要仔细的定位。通常来说，分析堆内存快照（Heap Dump）是一个很好的定位手段，如果发生内存溢出时没有生成内存快照那就实在是太糟了，特别是对于那种 JVM 已经崩溃或者错误只出现在顺利运行了数小时甚至数天的生产系统上的情况。

幸运的是，我们可以通过设置 - XX:+HeapDumpOnOutOfMemoryError 让 JVM 在发生内存溢出时自动的生成堆内存快照。有了这个参数，当我们不得不面对内存溢出异常的时候会节约大量的时间。默认情况下，堆内存快照会保存在 JVM 的启动目录下名为 java_pid.hprof 的文件里（在这里 就是 JVM 进程的进程号）。也可以通过设置 - XX:HeapDumpPath= 来改变默认的堆内存快照生成路径， 可以是相对或者绝对路径。

虽然这一切听起来很不错，但有一点我们需要牢记。堆内存快照文件有可能很庞大，特别是当内存溢出错误发生的时候。因此，我们推荐将堆内存快照生成路径指定到一个拥有足够磁盘空间的地方。

-XX:OnOutOfMemoryError

当内存溢发生时，我们甚至可以可以执行一些指令，比如发个 E-mail 通知管理员或者执行一些清理工作。通过 - XX:OnOutOfMemoryError 这个参数我们可以做到这一点，这个参数可以接受一串指令和它们的参数。在这里，我们将不会深入它的细节，但我们提供了它的一个例子。在下面的例子中，当内存溢出错误发生的时候，我们会将堆内存快照写到 /tmp/heapdump.hprof 文件并且在 JVM 的运行目录执行脚本 cleanup.sh

```
$ java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof -XX:OnOutOfMemoryError =
```

-XX:PermSize and -XX:MaxPermSize

永久代在堆内存中是一块独立的区域，它包含了所有 JVM 加载的类的对象表示。为了成功运行应用程序，JVM 会加载很多类（因为它们依赖于大量的第三方库，而这又依赖于更多的库并且需要从里面将类加载进来）这就需要增加永久代的大小。我们可以使用 - XX:PermSize 和 - XX:MaxPermSize 来达到这个目的。其中 - XX:MaxPermSize 用于设置永久代大小的最大值，-XX:PermSize 用于设置永久代初始大小。下面是一个简单的例子：

```
$ java -XX:PermSize=128m -XX:MaxPermSize=256m MyApp
```

请注意，这里设置的永久代大小并不会被包括在使用参数 - XX:MaxHeapSize 设置的堆内存大小中。也就是说，通过 - XX:MaxPermSize 设置的永久代内存可能会需要由参数 - XX:MaxHeapSize 设置的堆内存以外的更多的一些堆内存。

-XX:InitialCodeCacheSize and -XX:ReservedCodeCacheSize

JVM 一个有趣的，但往往被忽视的内存区域是 “代码缓存”，它是用来存储已编译方法生成的本地代码。代码缓存确实很少引起性能问题，但是一旦发生其影响可能是毁灭性的。如果代码缓存被占满，JVM 会打印出一条警告消息，并切换到 interpreted-only 模式：JIT 编译器被停用，字节码将不再会被编译成机器码。因此，应用程

序将继续运行，但运行速度会降低一个数量级，直到有人注意到这个问题。就像其他内存区域一样，我们可以自定义代码缓存的大小。相关的参数是 - XX:InitialCodeCacheSize 和 - XX:ReservedCodeCacheSize，它们的参数和上面介绍的参数一样，都是字节值。

-XX:+UseCodeCacheFlushing

如果代码缓存不断增长，例如，因为热部署引起的内存泄漏，那么提高代码的缓存大小只会延缓其发生溢出。为了避免这种情况的发生，我们可以尝试一个有趣的新参数：当代码缓存被填满时让 JVM 放弃一些编译代码。通过使用 - XX:+UseCodeCacheFlushing 这个参数，我们至少可以避免当代码缓存被填满的时候 JVM 切换到 interpreted-only 模式。不过，我仍建议尽快解决代码缓存问题发生的根本原因，如找出内存泄漏并修复它。

5

新生代垃圾回收

原文链接 作者： PATRICK PESCHLOW；译者：严亮

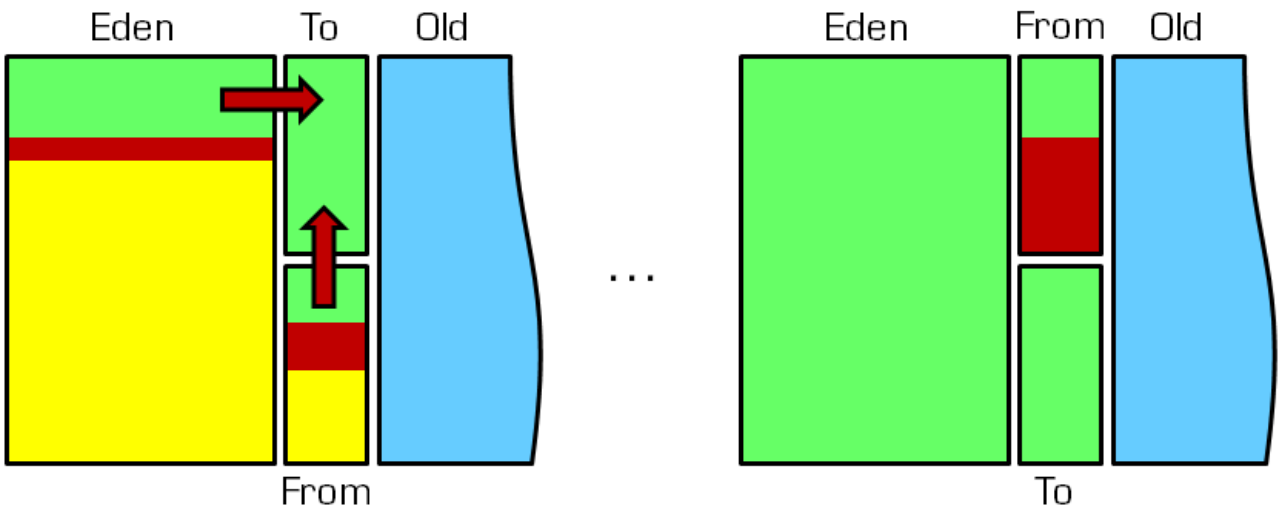
本部分，我们将关注堆 (heap) 中一个主要区域，新生代 (young generation)。首先我们会讨论为什么调整新生代的参数会对应用的性能如此重要，接着我们将学习新生代相关的 JVM 参数。

单纯从 JVM 的功能考虑，并不需要新生代，完全可以针对整个堆进行操作。新生代存在的唯一理由是优化垃圾回收 (GC) 的性能。更具体说，把堆划分为新生代和老年代有 2 个好处：简化了新对象的分配 (只在新生代分配内存)， 可以更有效的清除不再需要的对象 (即死对象)(新生代和老年代使用不同的 GC 算法)

通过广泛研究面向对象实现的应用，发现一个共同特点：很多对象的生存时间都很短。同时研究发现，新生对象很少引用生存时间长的对象。结合这 2 个特点，很明显 GC 会频繁访问新生对象，例如在堆中一个单独的区域，称之为新生代。在新生代中，GC 可以快速标记回收” 死对象”，而不需要扫描整个 Heap 中的存活一段时间的” 老对象”。

SUN/Oracle 的 HotSpot JVM 又把新生代进一步划分为 3 个区域：一个相对大点的区域，称为“伊甸园区 (Eden)”；两个相对小点的区域称为“From 幸存区 (survivor)” 和“To 幸存区 (survivor)”。按照规定，新对象会首先分配在 Eden 中 (如果新对象过大，会直接分配在老年代中)。在 GC 中，Eden 中的对象会被移动到 survivor 中，直至对象满足一定的年纪 (定义为熬过 GC 的次数)，会被移动到老年代。

基于大多数新生对象都会在 GC 中被收回的假设。新生代的 GC 使用复制算法。在 GC 前 To 幸存区 (survivor) 保持清空，对象保存在 Eden 和 From 幸存区 (survivor) 中，GC 运行时，Eden 中的幸存对象被复制到 To 幸存区 (survivor)。针对 From 幸存区 (survivor) 中的幸存对象，会考虑对象年龄，如果年龄没达到阀值 (tenuring threshold)，对象会被复制到 To 幸存区 (survivor)。如果达到阀值对象被复制到老年代。复制阶段完成后，Eden 和 From 幸存区中只保存死对象，可以视为清空。如果在复制过程中 To 幸存区被填满了，剩余的对象会被复制到老年代中。最后 From 幸存区和 To 幸存区会调换下名字，在下次 GC 时，To 幸存区会成为 From 幸存区。



https://blog.codecentric.de/files/2011/08/young_gc.png

上图演示 GC 过程，黄色表示死对象，绿色表示剩余空间，红色表示幸存对象

总结一下，对象一般出生在 Eden 区，年轻代 GC 过程中，对象在 2 个幸存区之间移动，如果对象存活到适当的年龄，会被移动到老年代。当对象在老年代死亡时，就需要更高级别的 GC，更重量级的 GC 算法 (复制算法不适用于老年代，因为没有多余的空间用于复制)

现在应该能理解为什么新生代大小非常重要了 (译者，有另外一种说法：新生代大小并不重要，影响 GC 的因素主要是幸存对象的数量)，如果新生代过小，会导致新生对象很快就晋升到老年代中，在老年代中对象很难被回收。如果新生代过大，会发生过多的复制过程。我们需要找到一个合适大小，不幸的是，要想获得一个合适的大小，只能通过不断的测试调优。这就需要 JVM 参数了

-XX:NewSize and -XX:MaxNewSize

就像可以通过参数 (-Xms and -Xmx) 指定堆大小一样，可以通过参数指定新生代大小。设置 XX:MaxNewSize 参数时，应该考虑到新生代只是整个堆的一部分，新生代设置的越大，老年代区域就会减少。一般不允许新生代比老年代还大，因为要考虑 GC 时最坏情况，所有对象都晋升到老年代。(译者: 会发生 OOM 错误) -XX:MaxNewSize 最大可以设置为 - Xmx/2。

考虑性能，一般会通过参数 -XX:NewSize 设置新生代初始大小。如果知道新生代初始分配的对象大小 (经过监控)，这样设置会有帮助，可以节省新生代自动扩展的消耗。

-XX:NewRatio

可以设置新生代和老年代的相对大小。这种方式的优点是新生代大小会随着整个堆大小动态扩展。参数 -XX:NewRatio 设置老年代与新生代的比例。例如 -XX:NewRatio=3 指定老年代 / 新生代为 3/1。 老年代占堆大小的 3/4，新生代占 1/4 。

如果针对新生代，同时定义绝对值和相对值，绝对值将起作用。下面例子：

```
$ java -XX:NewSize=32m -XX:MaxNewSize=512m -XX:NewRatio=3 MyApp
```

以上设置，JVM 会尝试为新生代分配四分之一的堆大小，但不会小于 32MB 或大于 521MB

在设置新生代大小问题上，使用绝对值还是相对值，不存在通用准则 。如果了解应用的内存使用情况， 设置固定大小的堆和新生代更有利，当然也可以设置相对值。如果对应用的内存使用一无所知，正确的做法是不要设置任何参数，如果应用运行良好。很好，我们不用做任何额外动作。如果遇到性能或 OutOfMemoryErrors，在调优之前，首先需要进行一系列有目的的监控测试，缩小问题的根源。

-XX:SurvivorRatio

参数 -XX:SurvivorRatio 与 -XX:NewRatio 类似，作用于新生代内部区域。-XX:SurvivorRatio 指定伊甸园区 (Eden) 与幸存区大小比例。 例如， -XX:SurvivorRatio=10 表示伊甸园区 (Eden) 是 幸存区 To 大小的 10

倍 (也是幸存区 From 的 10 倍)。 所以， 伊甸园区 (Eden) 占新生代大小的 10/12， 幸存区 From 和幸存区 To 每个占新生代的 1/12 。 注意， 两个幸存区永远是一样大的。

设定幸存区大小有什么作用? 假设幸存区相对伊甸园区 (Eden) 太小， 相应新生对象的伊甸园区 (Eden) 永远很大空间， 我们当然希望， 如果这些对象在 GC 时全部被回收， 伊甸园区 (Eden) 被清空， 一切正常。 然而， 如果有一部分对象在 GC 中幸存下来， 幸存区只有很少空间容纳这些对象。 结果大部分幸存对象在一次 GC 后，就会被转移到老年代 ， 这并不是我们希望的。 考虑相反情况， 假设幸存区相对伊甸园区 (Eden) 太大， 当然有足够的空间，容纳 GC 后的幸存对象。 但是过小的伊甸园区 (Eden)， 意味着空间将越快耗尽，增加新生代 GC 次数，这是不可接受的。

总之， 我们希望最小化短命对象晋升到老年代的数量，同时也希望最小化新生代 GC 的次数和持续时间。 我们需要找到针对当前应用的折中方案， 寻找适合方案的起点是 了解当前应用中对象的年龄分布情况。

-XX:+PrintTenuringDistribution

参数 `-XX:+PrintTenuringDistribution` 指定 JVM 在每次新生代 GC 时，输出幸存区中对象的年龄分布。例如:

Desired survivor size 75497472 bytes， new threshold 15 (max 15)

- age 1: 19321624 bytes， 19321624 total
- age 2: 79376 bytes， 19401000 total
- age 3: 2904256 bytes， 22305256 total

第一行说明幸存区 To 大小为 75 MB。 也有关于老年代阀值 (tenuring threshold) 的信息， 老年代阀值，意思是对象从新生代移动到老年代之前，经过几次 GC(即， 对象晋升前的最大年龄)。 上例中， 老年代阀值为 15，最大也是 15。

之后行表示，对于小于老年代阀值的每一个对象年龄，本年龄中对象所占字节 (如果当前年龄没有对象， 这一行会忽略)。 上例中， 一次 GC 后幸存对象大约 19 MB， 两次 GC 后幸存对象大约 79 KB，三次 GC 后幸存对象大约 3 MB 。 每行结尾，显示直到本年龄全部对象大小。 所以， 最后一行的 total 表示幸存区 To 总共被占用 2 2 MB 。 幸存区 To 总大小为 75 MB ， 当前老年代阀值为 15，可以断定在本次 GC 中，没有对象会移动到老年代。现在假设下一次 GC 输出为:

Desired survivor size 75497472 bytes， new threshold 2 (max 15)

- age 1: 68407384 bytes， 68407384 total
- age 2: 12494576 bytes， 80901960 total
- age 3: 79376 bytes， 80981336 total
- age 4: 2904256 bytes， 83885592 total

对比前一次老年代分布。明显的， 年龄 2 和年龄 3 的对象还保持在幸存区中，因为我们看到年龄 3 和 4 的对象大小与前一次年龄 2 和 3 的相同。同时发现幸存区中， 有一部分对象已经被回收， 因为本次年龄 2 的对象大小为 12MB ，而前一次年龄 1 的对象大小为 19 MB。最后可以看到最近的 GC 中，有 68 MB 新对象，从伊甸园区移动到幸存区。

注意， 本次 GC 幸存区占用总大小 84 MB - 大于 75 MB。 结果， JVM 把老年代阀值从 15 降低到 2，在下次 GC 时，一部分对象会强制离开幸存区，这些对象可能会被回收 (如果他们刚好死亡) 或移动到老年代。

-XX:InitialTenuringThreshold， -XX:MaxTenuringThreshold and -XX:TargetSurvivorRatio

参数 -XX:+PrintTenuringDistribution 输出中的部分值可以通过其它参数控制。通过 -XX:InitialTenuringThreshold 和 -XX:MaxTenuringThreshold 可以设定老年代阀值的初始值和最大值。另外， 可以通过参数 -XX:TargetSurvivorRatio 设定幸存区的目标使用率。 例如， -XX:MaxTenuringThreshold=10 -XX:TargetSurvivorRatio=90 设定老年代阀值的上限为 10， 幸存区空间目标使用率为 90%。

有多种方式， 设置新生代行为，没有通用准则。我们必须清楚以下 2 中情况:

1. 如果从年龄分布中发现，有很多对象的年龄持续增长，在到达老年代阀值之前。这表示 -XX:MaxTenuringThreshold 设置过大
2. 如果 -XX:MaxTenuringThreshold 的值大于 1，但是很多对象年龄从未大于 1。应该看下幸存区的目标使用率。如果幸存区使用率从未到达，这表示对象都被 GC 回收，这正是我们想要的。 如果幸存区使用率经常达到，有些年龄超过 1 的对象被移动到老年代中。这种情况，可以尝试调整幸存区大小或目标使用率。

-XX:+NeverTenure and -XX:+AlwaysTenure

最后，我们介绍 2 个颇为少见的参数，对应 2 种极端的新生代 GC 情况。设置参数 -XX:+NeverTenure，对象永远不会晋升到老年代。当我们确定不需要老年代时，可以这样设置。这样设置风险很大， 并且会浪费至少一半的堆内存。相反设置参数 -XX:+AlwaysTenure，表示没有幸存区，所有对象在第一次 GC 时，会晋升到老年代。

没有合理的场景使用这个参数。可以在测试环境中，看下这样设置会发生什么有趣的事。但是并不推荐使用这些参数。

结论 适当的配置新生代非常重要，有相当多的参数可以设置新生代。然而，单独调整新生代，而不考虑老年代是不可能优化成功的。当调整堆和 GC 设置时，我们总是应该同时考虑新生代和老年代。

在本系列的下面 2 部分，我们将讨论 HotSpot JVM 中老年代 GC 策略，我们会学习 “吞吐量 GC 收集器” 和 “并发低延迟 GC 收集器”，也会了解收集器的基本准则，算法和调整参数。

6

吞吐量收集器

原文链接 本文连接 译者：张军 校对：梁海舰

在实践中我们发现对于大多数的应用领域，评估一个垃圾收集 (GC) 算法如何根据如下两个标准：

1. 吞吐量越高算法越好
2. 暂停时间越短算法越好

首先让我们来明确垃圾收集 (GC) 中的两个术语: 吞吐量 (throughput) 和暂停时间 (pause times)。 JVM 在专门的线程 (GC threads) 中执行 GC。 只要 GC 线程是活动的，它们将与应用程序线程 (application threads) 争用当前可用 CPU 的时钟周期。 简单点来说，吞吐量是指应用程序线程用时占程序总用时的比例。 例如，吞吐量 99/100 意味着 100 秒的程序执行时间应用程序线程运行了 99 秒， 而在这一时间段内 GC 线程只运行了 1 秒。

术语” 暂停时间” 是指一个时间段内应用程序线程让与 GC 线程执行而完全暂停。 例如，GC 期间 100 毫秒的暂停时间意味着在这 100 毫秒期间内没有应用程序线程是活动的。 如果说一个正在运行的应用程序有 100 毫秒的 “平均暂停时间”，那么就是说该应用程序所有的暂停时间平均长度为 100 毫秒。 同样，100 毫秒的 “最大暂停时间” 是指该应用程序所有的暂停时间最大不超过 100 毫秒。

#

吞吐量 VS 暂停时间

高吞吐量最好因为这会让应用程序的最终用户感觉只有应用程序线程在做 “生产性” 工作。 直觉上，吞吐量越高程序运行越快。 低暂停时间最好因为从最终用户的角度来看不管是 GC 还是其他原因导致一个应用被挂起始终是不好的。 这取决于应用程序的类型，有时候甚至短暂的 200 毫秒暂停都可能打断终端用户体验。 因此，具有低的最大暂停时间是非常重要的，特别是对于一个交互式应用程序。

不幸的是” 高吞吐量” 和” 低暂停时间” 是一对相互竞争的目标（矛盾）。这样想想看，为了清晰起见简化一下：GC 需要一定的前提条件以便安全地运行。 例如，必须保证应用程序线程在 GC 线程试图确定哪些对象仍然被引用和哪些没有被引用的时候不修改对象的状态。 为此，应用程序在 GC 期间必须停止 (或者仅在 GC 的特定阶段，这取决于所使用的算法)。 然而这会增加额外的线程调度开销：直接开销是上下文切换，间接开销是因为缓存的影响。 加上 JVM 内部安全措施的开销，这意味着 GC 及随之而来的不可忽略的开销，将增加 GC 线程执行实际工作的时间。 因此我们可以通过尽可能少运行 GC 来最大化吞吐量，例如，只有在不可避免的时候进行 GC，来节省所有与它相关的开销。

然而，仅仅偶尔运行 GC 意味着每当 GC 运行时将有许多工作要做，因为在此期间积累在堆中的对象数量很高。 单个 GC 需要花更多时间来完成， 从而导致更高的平均和最大暂停时间。 因此，考虑到低暂停时间，最好频繁地运行 GC 以便更快速地完成。 这反过来又增加了开销并导致吞吐量下降，我们又回到了起点。

综上所述，在设计（或使用）GC 算法时，我们必须确定我们的目标：一个 GC 算法只可能针对两个目标之一（即只专注于最大吞吐量或最小暂停时间），或尝试找到一个二者的折衷。

#

HotSpot 虚拟机上的垃圾收集

该系列的第五部分我们已经讨论过年轻代的垃圾收集器。 对于年老代，HotSpot 虚拟机提供两类垃圾收集算法 (除了新的 G1 垃圾收集算法)，第一类算法试图最大限度地提高吞吐量，而第二类算法试图最小化暂停时间。 今天我们的重点是第一类，” 面向吞吐量” 的垃圾收集算法。

我们希望把重点放在 JVM 配置参数上，所以我只会简要概述 HotSpot 提供的面向吞吐量 (throughput-oriented) 垃圾收集算法。 当年老代中由于缺乏空间导致对象分配失败时会触发垃圾收集器 (事实上，” 分配” 的通常是指从年轻代提升到年老代的对象)。 从所谓的”GC 根”(GC roots) 开始，搜索堆中的可达对象并将其标记为活着的，之后，垃圾收集器将活着的对象移到年老代的一块无碎片 (non-fragmented) 内存块中，并标记剩余的内存空间是空闲的。 也就是说，我们不像复制策略那样移到一个不同的堆区域，像年轻代垃圾收集算法所做的那样。 相反地，我们把所有的对象放在一个堆区域中，从而对该堆区域进行碎片整理。 垃圾收集器使用一个或多个线程来执行垃圾收集。 当使用多个线程时，算法的不同步骤被分解，使得每个收集线程大多时候工作在自己的区域而不干扰其他线程。 在垃圾收集期间，所有的应用程序线程暂停，只有垃圾收集完成之后才会重新开始。 现在让我们来看看跟面向吞吐量垃圾收集算法有关的重要 JVM 配置参数。

#

-XX:+UseSerialGC

我们使用该标志来激活串行垃圾收集器，例如单线程面向吞吐量垃圾收集器。 无论年轻代还是年老代都将只有一个线程执行垃圾收集。 该标志被推荐用于只有单个可用处理器核心的 JVM。 在这种情况下，使用多个垃圾收集线程甚至会适得其反，因为这些线程将争用 CPU 资源，造成同步开销，却从未真正并行运行。

#

-XX:+UseParallelGC

有了这个标志，我们告诉 JVM 使用多线程并行执行年轻代垃圾收集。 在我看来，Java 6 中不应该使用该标志因为 - XX:+UseParallelOldGC 显然更合适。 需要注意的是 Java 7 中该情况改变了一点 (详见本概述)，就是 - XX:+UseParallelGC 能达到 - XX:+UseParallelOldGC 一样的效果。

#

-XX:+UseParallelOldGC

该标志的命名有点不巧，因为” 老” 听起来像” 过时”。然而，” 老” 实际上是指年老代，这也解释了为什么 - XX:+UseParallelOldGC 要优于 - XX:+UseParallelGC：除了激活年轻代并行垃圾收集，也激活了年老代并行垃圾收集。当期望高吞吐量，并且 JVM 有两个或更多可用处理器核心时，我建议使用该标志。作为旁注，HotSpot 的并行面向吞吐量垃圾收集算法通常称为” 吞吐量收集器”，因为它们旨在通过并行执行来提高吞吐量。

#

-XX:ParallelGCThreads

通过 - XX:ParallelGCThreads= 我们可以指定并行垃圾收集的线程数量。 例如，-XX:ParallelGCThreads=6 表示每次并行垃圾收集将有 6 个线程执行。 如果不明确设置该标志，虚拟机将使用基于可用 (虚拟) 处理器数量计算的默认值。 决定因素是由 Java Runtime。availableProcessors() 方法的返回值 N，如果 N<=8，并行垃圾收集器将使用 N 个垃圾收集线程，如果 N>8 个可用处理器，垃圾收集线程数量应为 3+5N/8。 当 JVM 独占地使用系统和处理器时使用默认设置更有意义。 但是，如果有多个 JVM(或其他耗 CPU 的系统) 在同一台机器上运行，我们应该使用 - XX:ParallelGCThreads 来减少垃圾收集线程数到一个适当的值。 例如，如果 4 个以服务器方式运行的 JVM 同时跑在在一个具有 16 核处理器的机器上，设置 - XX:ParallelGCThreads=4 是明智的，它能使不同 JVM 的垃圾收集器不会相互干扰。

#

-XX:-UseAdaptiveSizePolicy

吞吐量垃圾收集器提供了一个有趣的 (但常见，至少在现代 JVM 上) 机制以提高垃圾收集配置的用户友好性。 这种机制被看做是 HotSpot 在 Java 5 中引入的” 人体工程学” 概念的一部分。 通过人体工程学，垃圾收集器能将堆大小动态变动像 GC 设置一样应用到不同的堆区域，只要有证据表明这些变动将能提高 GC 性能。“提高 GC 性能” 的确切含义可以由用户通过 - XX:GCTimeRatio 和 - XX:MaxGCPauseMillis(见下文) 标记来指定。 重要的是要知道人体工程学是默认激活的。 这很好，因为自适应行为是 JVM 最大优势之一。 不过，有时我们需要非常清楚对于特定应用什么样的设置是最合适的，在这些情况下，我们可能不希望 JVM 混乱我们的设置。 每当我们发现处于这种情况时，我们可以考虑通过 - XX:-UseAdaptiveSizePolicy 停用一些人体工程学。

#

-XX:GCTimeRatio

通过 - XX:GCTimeRatio= 我们告诉 JVM 吞吐量要达到的目标值。 更准确地说，-XX:GCTimeRatio=N 指定目标应用程序线程的执行时间 (与总的程序执行时间) 达到 N/(N+1) 的目标比值。 例如，通过 - XX:GCTimeRatio=9 我们要求应用程序线程在整个执行时间中至少 9/10 是活动的 (因此，GC 线程占用其余 1/10)。 基于运行时的测量，JVM 将会尝试修改堆和 GC 设置以期达到目标吞吐量。 -XX:GCTimeRatio 的默认值是 99，也就是说，应用程序线程应该运行至少 99% 的总执行时间。

#

-XX:MaxGCPauseMillis

通过 - XX:GCTimeRatio=<value> 告诉 JVM 最大暂停时间的目标值 (以毫秒为单位)。 在运行时，吞吐量收集器计算在暂停期间观察到的统计数据 (加权平均和标准偏差)。 如果统计表明正在经历的暂停其时间存在超过目标值的风险时，JVM 会修改堆和 GC 设置以降低它们。 需要注意的是，年轻代和年老代垃圾收集的统计数据是分开计算的，还要注意，默认情况下，最大暂停时间没有被设置。 如果最大暂停时间和最小吞吐量同时设置了目标值，实现最大暂停时间目标具有更高的优先级。 当然，无法保证 JVM 将一定能达到任一目标，即使它会努力去做。 最后，一切都取决于手头应用程序的行为。

当设置最大暂停时间目标时，我们应注意不要选择太小的值。 正如我们现在所知道的，为了保持低暂停时间，JVM 需要增加 GC 次数，那样可能会严重影响可达到的吞吐量。 这就是为什么对于要求低暂停时间作为主要目标的应用程序 (大多数是 Web 应用程序)，我会建议不要使用吞吐量收集器，而是选择 CMS 收集器。 CMS 收集器是本系列下一部分的主题。

7

CMS 收集器

原文连接 本文连接 译者： iDestiny 校对：梁海舰

HotSpot JVM 的并发标记清理收集器 (CMS 收集器) 的主要目标就是：低应用停顿时间。该目标对于大多数交互式应用很重要，比如 web 应用。在我们看一下有关 JVM 的参数之前, 让我们简要回顾 CMS 收集器的操作和使用它时可能出现的主要挑战。

就像吞吐量收集器 (参见本系列的第 6 部分),CMS 收集器处理老年代的对象, 然而其操作要复杂得多。吞吐量收集器总是暂停应用程序线程，并且可能是相当长的一段时间，然而这能够使该算法安全地忽略应用程序。相比之下，CMS 收集器被设计成在大多数时间能与应用程序线程并行执行，仅仅会有一点 (短暂的) 停顿时间。GC 与应用程序并行的缺点就是，可能会出现各种同步和数据不一致的问题。为了实现安全且正确的并发执行，CMS 收集器的 GC 周期被分为了好几个连续的阶段。

#

CMS 收集器的过程

CMS 收集器的 GC 周期由 6 个阶段组成。其中 4 个阶段 (名字以 Concurrent 开始的) 与实际的应用程序是并发执行的，而其他 2 个阶段需要暂停应用程序线程。

1. 初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。
2. 并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。
3. 并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。
4. 重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。
5. 并发清理：所有不再被应用的对象将从堆里清除掉。
6. 并发重置：收集器做一些收尾的工作，以便下一次 GC 周期能有一个干净的状态。

一个常见的误解是, CMS 收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使 “stop-the-world” 阶段相对于并发阶段的时间很短。

应该指出，尽管 CMS 收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过 “stop-the-world” 方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻带的垃圾回收时间通常是相当短的。

#

挑战

当我们在真实的应用中使用 CMS 收集器时，我们会面临两个主要的挑战，可能需要进行调优：

1. 堆碎片
2. 对象分配率高

堆碎片是有可能的，不像吞吐量收集器，CMS 收集器并没有任何碎片整理的机制。因此，应用程序有可能出现这样的情形，即使总的堆大小远没有耗尽，但却不能分配对象——仅仅是因为没有足够连续的空间完全容纳对象。当这种事发生后，并发算法不会帮上任何忙，因此，万不得已 JVM 会触发 Full GC。回想一下，Full GC 将运行吞吐量收集器的算法，从而解决碎片问题——但却暂停了应用程序线程。因此尽管 CMS 收集器带来完全的并发性，但仍然有可能发生长时间的 “stop-the-world” 的风险。这是 “设计”，而不能避免的——我们只能通过调优收集器来它的可能性。想要 100% 保证避免”stop-the-world”，对于交互式应用是有问题的。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。从某种程度上说，老年代将没有足够的可用空间来容纳一个从年轻代提升过来的对象。这种情况被称为 “并发模式失败”，并且 JVM 会执行堆碎片整理：触发 Full GC。

当这些情形之一出现在实践中时 (经常会出现在生产系统中)，经常被证实是老年代有大量不必要的对象。一个可行的办法就是增加年轻代的堆大小，以防止年轻代短生命的对象提前进入老年代。另一个办法就似乎利用分析器，快照运行系统的堆转储，并且分析过度的对象分配，找出这些对象，最终减少这些对象的申请。

下面我看看大多数与 CMS 收集器调优相关的 JVM 标志参数。

-XX：+UseConcMarkSweepGC

该标志首先是激活 CMS 收集器。默认 HotSpot JVM 使用的是并行收集器。

-XX：UseParNewGC

当使用 CMS 收集器时，该标志激活年轻代使用多线程并行执行垃圾回收。这令人很惊讶，我们不能简单在并行收集器中重用 - XX：UserParNewGC 标志，因为概念上年轻代用的算法是一样的。然而，对于 CMS 收集器，年轻代 GC 算法和老年代 GC 算法是不同的，因此年轻代 GC 有两种不同的实现，并且是两个不同的标志。

注意最新的 JVM 版本，当使用 - XX：+UseConcMarkSweepGC 时，-XX：UseParNewGC 会自动开启。因此，如果年轻代的并行 GC 不想开启，可以通过设置 - XX：-UseParNewGC 来关掉。

-XX：+CMSConcurrentMTEnabled

当该标志被启用时，并发的 CMS 阶段将以多线程执行 (因此，多个 GC 线程会与所有的应用程序线程并行工作)。该标志已经默认开启，如果顺序执行更好，这取决于所使用的硬件，多线程执行可以通过 - XX：-CMSConcurremntMTEnabled 禁用。

-XX：ConcGCThreads

标志 - XX：ConcGCThreads=(早期 JVM 版本也叫 -XX:ParallelCMSThreads) 定义并发 CMS 过程运行时的线程数。比如 value=4 意味着 CMS 周期的所有阶段都以 4 个线程来执行。尽管更多的线程会加快并发 CMS 过程，但其也会带来额外的同步开销。因此，对于特定的应用程序，应该通过测试来判断增加 CMS 线程数是否真的能够带来性能的提升。

如果还标志未设置，JVM 会根据并行收集器中的 - XX：ParallelGCThreads 参数的值来计算出默认的并行 CMS 线程数。该公式是 ConcGCThreads = (ParallelGCThreads + 3)/4。因此，对于 CMS 收集器， -XX:ParallelGCThreads标志不仅影响“stop-the-world”垃圾收集阶段，还影响并发阶段。

总之，有不少方法可以配置 CMS 收集器的多线程执行。正是由于这个原因, 建议第一次运行 CMS 收集器时使用其默认设置, 然后如果需要调优再进行测试。只有在生产系统中测量 (或类生产测试系统) 发现应用程序的暂停时间的目标没有达到 , 就可以通过这些标志应该进行 GC 调优。

-XX:CMSInitiatingOccupancyFraction

当堆满之后，并行收集器便开始进行垃圾收集，例如，当没有足够的空间来容纳新分配或提升的对象。对于 CMS 收集器，长时间等待是不可取的，因为在并发垃圾收集期间应用持续在运行 (并且分配对象)。因此，为了在应用程序使用完内存之前完成垃圾收集周期，CMS 收集器要比并行收集器更先启动。

因为不同的应用会有不同对象分配模式，JVM 会收集实际的对象分配 (和释放) 的运行时数据，并且分析这些数据，来决定什么时候启动一次 CMS 垃圾收集周期。为了引导这一过程， JVM 会在一开始执行 CMS 周期前作一些线索查找。该线索由 -XX:CMSInitiatingOccupancyFraction=来设置，该值代表老年代堆空间的使用率。比如，value=75意味着第一次CMS垃圾收集会在老年代被占用75%时被触发。通常CMSInitiatingOccupancyFraction的默认值为68(之前很长时间的经历来决定的)。

-XX：+UseCMSInitiatingOccupancyOnly

我们用 -XX+UseCMSInitiatingOccupancyOnly 标志来命令 JVM 不基于运行时收集的数据来启动 CMS 垃圾收集周期。而是，当该标志被开启时，JVM 通过 CMSInitiatingOccupancyFraction 的值进行每一次 CMS 收集，而不仅仅是第一次。然而，请记住大多数情况下，JVM 比我们自己能作出更好的垃圾收集决策。因此，只有当我们充足的理由 (比如测试) 并且对应用程序产生的对象的生命周期有深刻的认知时，才应该使用该标志。

-XX:+CMSClassUnloadingEnabled

相对于并行收集器，CMS 收集器默认不会对永久代进行垃圾回收。如果希望对永久代进行垃圾回收，可用设置标志 -XX:+CMSClassUnloadingEnabled。在早期 JVM 版本中，要求设置额外的标志 - XX:+CMSPermGenSweepingEnabled。注意，即使没有设置这个标志，一旦永久代耗尽空间也会尝试进行垃圾回收，但是收集不会是并行的，而再一次进行 Full GC。

-XX:+CMSIncrementalMode

该标志将开启 CMS 收集器的增量模式。增量模式经常暂停 CMS 过程，以便对应用程序线程作出完全的让步。因此，收集器将花更长的时间完成整个收集周期。因此，只有通过测试后发现正常 CMS 周期对应用程序线程干扰太大时，才应该使用增量模式。由于现代服务器有足够的处理器来适应并发的垃圾收集，所以这种情况发生得很少。

-XX:+ExplicitGCInvokesConcurrent and -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

如今, 被广泛接受的最佳实践是避免显式地调用 GC(所谓的 “系统 GC”)，即在应用程序中调用 system.gc()。然而，这个建议是不管使用的 GC 算法的，值得一提的是，当使用 CMS 收集器时，系统 GC 将是一件很不幸的事，因为它默认会触发一次 Full GC。幸运的是，有一种方式可以改变默认设置。标志 - XX:+ExplicitGCInvokesConcurrent 命令 JVM 无论什么时候调用系统 GC，都执行 CMS GC，而不是 Full GC。第二个标志 -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses 保证当有系统 GC 调用时，永久代也被包括进 CMS 垃圾回收的范围内。因此，通过使用这些标志，我们可以防止出现意料之外的”stop-the-world” 的系统 GC。

-XX:+DisableExplicitGC

然而在这个问题上… 这是一个很好提到 - XX:+ DisableExplicitGC 标志的机会，该标志将告诉 JVM 完全忽略系统的 GC 调用 (不管使用的收集器是什么类型)。对于我而言，该标志属于默认的标志集合中，可以安全地定义在每个 JVM 上运行，而不需要进一步思考。

8

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，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：

```
[GC 246656K->243120K(376320K), 0.0929090 secs]
[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。

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

```
[Full GC
   [PSYoungGen: 10752K->9707K(142848K)]
   [ParOldGen: 232384K->232244K(485888K)] 243136K->241951K(628736K)
   [PSPermGen: 3162K->3161K(21504K)], 1.5265450 secs
]
```

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：

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

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