

## Java 6 JVM 参数选项大全 ( 中文版 )

作者 : [Ken Wu](#)

Email: [ken.wug@gmail.com](mailto:ken.wug@gmail.com)

转载本文档请注明原文链

接 <http://kenwublog.com/docs/java6-jvm-options-chinese-edition.htm> !

本文是基于最新的SUN官方文档 [Java SE 6 Hotspot VM Options](#) 编写的译文。主要介绍JVM中的非稳态选项及其使用说明。

为了让读者明白每个选项的含义,作者在原文基础上**补充了大量的资料**。希望这份文档,对正在研究 JVM 参数的朋友有帮助!

另外,考虑到本文档是初稿,如有描述错误,敬请指正。

### 非稳态选项使用说明

-XX:+<option> 启用选项

-XX:-<option> 不启用选项

-XX:<option>=<number> 给选项设置一个数字类型值,可跟单位,例如 32k, 1024m, 2g

-XX:<option>=<string> 给选项设置一个字符串值,例如

-XX:HeapDumpPath=./dump.core

### 行为选项

选项	默认值与限制	描述
-XX:-AllowUserSignalHandlers	限于 Linux 和 Solaris, 默认不启用	允许为 java 进程安装信号处理器。 Java信号处理相关知识, 详见 <a href="http://kenwublog.com/java-asynchronous-notify-based-on-signal">http://kenwublog.com/java-asynchronous-notify-based-on-signal</a>
-XX:-DisableExplicitGC	默认不启用	禁止在运行期显式地调用 System.gc()。 开启该选项后, GC 的触发时机将由 Garbage Collector 全权掌控。 注意:你熟悉的代码里没调用 System.gc(), 不代表你依赖的框架工具没在使用。 例如 RMI 就在多数用户毫不知情的情况下, 显示地调用 GC 来防止自身 OOM。 请仔细权衡禁用带来的影响。
-XX:-RelaxAccessControlCheck	默认不启用	在 Class 校验器中, 放松对访问控制的检查。 作用与 reflection 里的 setAccessible 类似。
-XX:-UseConcMarkSweepGC	默认不启用	启用 CMS 低停顿垃圾收集器。 资料详见: <a href="http://kenwublog.com/docs/CMS_GC.pdf">http://kenwublog.com/docs/CMS_GC.pdf</a>
-XX:-UseParallelGC	-server 时启用 其他情况下, 默认不启用	策略为新生代使用并行清除, 年老代使用单线程 Mark-Sweep-Compact 的垃圾收集器。
-XX:-UseParallelOldGC	默认不启用	策略为老年代和新生代都使用并行清除的垃圾收集器。
-XX:-UseSerialGC	-client 时启用 其他情况下, 默认不启用	使用串行垃圾收集器。
-XX:+UseSplitVerifier	java5 默认不启用 java6 默认启	使用新的 Class 类型校验器。 <b>新 Class 类型校验器有什么特点?</b>

	用	<p>新 Class 类型校验器，将老的校验步骤拆分成了两步： 1，类型推断。 2，类型校验。</p> <p>新类型校验器通过在 javac 编译时嵌入类型信息到 bytecode 中，省略了类型推断这一步，从而提升了 classloader 的性能。</p> <p>Classload 顺序（供参考） load -&gt; <b>verify</b> -&gt; prepare -&gt; resolve -&gt; init</p> <p><b>关联选项：</b> -XX:+FailOverToOldVerifier</p>
-XX:+FailOverToOldVerifier	Java6 新引入选项，默认启用	<p>如果新的 Class 校验器检查失败，则使用老的校验器。</p> <p><b>为什么会失败？</b> 因为 JDK6 最高向下兼容到 JDK1.2，而 JDK1.2 的 class info 与 JDK6 的 info 存在较大的差异，所以新校验器可能会出现校验失败的情况。</p> <p><b>关联选项：</b> -XX:+UseSplitVerifier</p>
-XX:+HandlePromotionFailure	java5 以前是默认不启用，java6 默认启用	<p>关闭新生代收集担保。</p> <p><b>什么是新生代收集担保？</b> 在一次理想化的 minor gc 中，Eden 和 First Survivor 中的活跃对象会被复制到 Second Survivor。然而，Second Survivor 不一定能容纳下所有从 E 和 F 区 copy 过来的活跃对象。为了确保 minor gc 能够顺利完成，GC 需要在年老代中额外保留一块足以容纳所有活跃对象的内存空间。这个预留操作，就被称之为新生代收集担保（New Generation Guarantee）。如果预留操作无法完成时，仍会触发 major gc(full gc)。</p> <p><b>为什么要关闭新生代收集担保？</b> 因为在年老代中预留的空间大小，是无法精确计算的。为了确保极端情况的发生，GC 参考了最坏情况下的新生代内存占用，即 Eden+First Survivor。这种策略无疑是在浪费年老代内存，从时序角度看，还会提前触发 Full GC。为了避免如上情况的发生，JVM 允许开发者手动关闭新生代收集担保。</p> <p>在开启本选项后，minor gc 将不再提供新生代收集担保，而是在出现 survivor 或年老代不够用时，抛出 promotion failed 异常。</p>
-XX:+UseSpinning	java1.4.2 和 1.5 需要手动启用，java6 默认已启用	<p>启用多线程自旋锁优化。</p> <p><b>自旋锁优化原理</b> 大家知道，Java 的多线程安全是基于 Lock 机制实现的，而 Lock 的性能往往不如人意。原因是，monitorenter 与 monitorexit 这两个控制多线程同步的 bytecode 原语，是 JVM 依赖操作系统互斥(mutex)来实现的。互斥是一种会导致线程挂起，并在较短的时间内又必须重新调度回原线程的，较为消耗资源的操作。</p>

		<p>为了避免进入 OS 互斥,Java6 的开发者们提出了自旋锁优化。</p> <p>自旋锁优化的原理是在线程进入 OS 互斥前,通过 CAS 自旋一定的次数来检测锁的释放。 如果在自旋次数未达到预设值前锁已被释放,则当前线程会立即持有该锁。</p> <p>CAS检测锁的原理详见: <a href="http://kenwublog.com/theory-of-lightweight-locking-upon-cas">http://kenwublog.com/theory-of-lightweight-locking-upon-cas</a></p> <p><b>关联选项:</b> -XX:PreBlockSpin=10</p>
-XX:PreBlockSpin=10	-XX:+UseSpinning 必须先启用,对于 java6 来说已经默认启用了,这里默认自旋 10 次	<p>控制多线程自旋锁优化的自旋次数。(什么是自旋锁优化?见 -XX:+UseSpinning 处的描述)</p> <p><b>关联选项:</b> -XX:+UseSpinning</p>
-XX:+ScavengeBeforeFullGC	默认启用	在 Full GC 前触发一次 Minor GC。
-XX:+UseGCOverheadLimit	默认启用	限制 GC 的运行时间。如果 GC 耗时过长,就抛 OOM。
-XX:+UseTLAB	1.4.2 以前和使用-client 选项时,默认不启用,其余版本默认启用	启用线程本地缓存区 ( Thread Local )。
-XX:+UseThreadPriorities	默认启用	使用本地线程的优先级。
-XX:+UseAltSigs	限于 Solaris, 默认启用	为了防止与其他发送信号的应用程序冲突,允许使用候补信号替代 SIGUSR1 和 SIGUSR2。
-XX:+UseBoundThreads	限于 Solaris, 默认启用	绑定所有的用户线程到内核线程。 减少线程进入饥饿状态 ( 得不到任何 cpu time ) 的次数。
-XX:+UseLWPSynchronization	限于 solaris, 默认启用	使用轻量级进程 ( 内核线程 ) 替换线程同步。
-XX:+MaxFDLimit	限于 Solaris, 默认启用	设置 java 进程可用文件描述符为操作系统允许的最大值。
-XX:+UseVMInterruptibleIO	限于 solaris, 默认启用	在 solaris 中, 允许运行时中断线程。

## 性能选项

选项与默认值	默认值与限制	描述
-XX:+AggressiveOpts	JDK 5 update 6 后引入,但需要手动启用。 JDK6 默认启用。	启用 JVM 开发团队最新的调优成果。例如编译优化,偏向锁,并行年老代收集等。

-XX:CompileThreshold=10000	1000	通过 JIT 编译器，将方法编译成机器码的触发阈值，可以理解为调用方法的次数，例如调 1000 次，将方法编译为机器码。
-XX:LargePageSizeInBytes=4m	默认 4m amd64 位：2m	设置堆内存的内存页大小。 调整内存页的方法和性能提升原理，详见 <a href="http://kenwublog.com/tune-large-page-for-jvm-optimization">http://kenwublog.com/tune-large-page-for-jvm-optimization</a>
-XX:MaxHeapFreeRatio=70	70	GC 后，如果发现空闲堆内存占到整个预估上限值的 70%，则收缩预估上限值。  <b>什么是预估上限值？</b> JVM 在启动时，会申请最大值（-Xmx 指定的数值）的地址空间，但其中绝大部分空间不会被立即分配（virtual）。 它们会一直保留着，直到运行过程中，JVM 发现实际占用接近已分配上限值时，才从 virtual 里再分配掉一部分内存。 这里提到的已分配上限值，也可以叫做预估上限值。  引入预估上限值的好处是，可以有效地控制堆的大小。堆越小，GC 效率越高嘛。 注意：预估上限值的大小一定小于或等于最大值。
-XX:MaxNewSize=size	1.3.1 Sparc: 32m 1.3.1 x86: 2.5m	新生代占整个堆内存的最大值。
-XX:MaxPermSize=64m	5.0 以后: 64 bit VMs 会增大预设值的 30% 1.4 amd64 : 96m 1.3.1 -client: 32m  其他默认 64m	Perm（俗称方法区）占整个堆内存的最大值。
-XX:MinHeapFreeRatio=40	40	GC 后，如果发现空闲堆内存占到整个预估上限值的 40%，则增大上限值。 (什么是预估上限值？见 -XX:MaxHeapFreeRatio 处的描述)  <b>关联选项：</b> -XX:MaxHeapFreeRatio=70
-XX:NewRatio=2	Sparc -client: 8 x86	新生代和年老代的堆内存占用比例。 例如 2 表示新生代占年老代的 1/2，占整个堆内存的 1/3。

	<p>-server: 8 x86 -client: 12 -client: 4 (1.3) 8 (1.3.1+) x86: 12</p> <p>其他默认 2</p>	
-XX:NewSize=2.125m	<p>5.0 以后: 64 bit Vms 会增大预设值的 30% x86: 1m x86, 5.0 以后: 640k</p> <p>其他默认 2.125 m</p>	<p>新生代预估上限的默认值。(什么是预估上限值? 见 -XX:MaxHeapFreeRatio 处的描述)</p>
-XX:ReservedCodeCacheSize=32m	<p>Solaris 64-bit, amd64 , -server x86: 48m 1.5.0_06 之前, Solaris 64-bit amd64 : 1024m</p> <p>其他默认 32m</p>	<p>设置代码缓存的最大值, 编译时用。</p>
-XX:SurvivorRatio=8	<p>Solaris amd64 : 6 Sparc in 1.3.1: 25</p>	<p>Eden 与 Survivor 的占用比例。例如 8 表示, 一个 survivor 区占用 1/8 的 Eden 内存, 即 1/10 新生代内存, 为什么不是 1/9? 因为我们的新生代有 2 个 survivor, 即 S1 和 S2。所以 survivor 总共是占用新生代内存的 2/10, Eden 与新生代的占比则为 8/10。</p>

	Solaris platforms 5.0 以前: 32 其他默认 8	
-XX:TargetSurvivorRatio=50	50	实际使用的 survivor 空间大小占比。默认是 50%，最高 90%。
-XX:ThreadStackSize=512	Sparc: 512 Solaris x86: 320 (5.0 以前 256) Sparc 64 bit: 1024 Linux amd64: 1024 (5.0 以前 0) 其他默认 512.	线程堆栈大小
-XX:+UseBiasedLocking	JDK 5 update 6 后引入, 但需要手动启用。 JDK6 默认启用。	启用偏向锁。 偏向锁原理详见 <a href="http://kenwublog.com/theory-of-java-biased-locking">http://kenwublog.com/theory-of-java-biased-locking</a>
-XX:+UseFastAccessorMethods	默认启用	优化原始类型的 getter 方法性能。
-XX:-UseISM	默认启用	启用 solaris 的 ISM。 详见 <a href="#">Intimate Shared Memory</a> .
-XX:+UseLargePages	JDK 5 update 5 后引入, 但需要手动启用。 JDK6 默认启用。	启用大内存分页。 调整内存页的方法和性能提升原理, 详见 <a href="http://kenwublog.com/tune-large-page-for-jvm-optimization">http://kenwublog.com/tune-large-page-for-jvm-optimization</a>
-XX:+UseMPSS	1.4.1 之前: 不启用 其余版本默认启用	启用 solaris 的 MPSS, 不能与 ISM 同时使用。
-XX:+StringCache	默认启用	启用字符串缓存。

	用	
-XX:AllocatePrefetchLines=1	1	与机器码指令预读相关的一个选项,资料比较少,本文档不做解释。有兴趣的朋友请自行阅读官方 doc。
-XX:AllocatePrefetchStyle=1	1	与机器码指令预读相关的一个选项,资料比较少,本文档不做解释。有兴趣的朋友请自行阅读官方 doc。

## 调试选项

选项与默认值	默认值与限制	描述
-XX:-CITime	1.4 引入。 默认启用	打印 JIT 编译器编译耗时。
-XX:ErrorFile=./hs_err_pid<pid>.log	Java 6 引入。	如果 JVM crashed, 将错误日志输出到指定文件路径。
-XX:-ExtendedDTraceProbes	Java6 引入, 限于 solaris, 默认不启用	启用 <a href="#">dtrace</a> 诊断。
-XX:HeapDumpPath=./java_pid<pid>.hprof	默认是 java 进程启动位置, 即 user.dir	堆内存快照的存储文件路径。 <b>什么是堆内存快照?</b> 当 java 进程因 OOM 或 crash 被 OS 强制终止后, 会生成一个 hprof ( Heap PROFling ) 格式的堆内存快照文件。该文件用于线下调试, 诊断, 查找问题。 文件名一般为 java_<pid>_<date>_<time>_heapDump.hprof 解析快照文件, 可以使用 jhat, eclipse MAT, gdb 等工具。
-XX:-HeapDumpOnOutOfMemoryError	1.4.2 和 5.0 update 7 引入。 默认不启用	在 OOM 时, 输出一个 dump.core 文件, 记录当时的堆内存快照 ( 什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述 )。
-XX:OnError="<cmd args>;<cmd args>"	1.4.2 和 5.0 update 9 引入	当 java 每抛出一个 ERROR 时, 运行指定命令行指令集。指令集是与 OS 环境相关的, 在 linux 下多数是 bash 脚本, windows 下是 dos 批处理。
-XX:OnOutOfMemoryError="<cmd args>;<cmd args>"	1.4.2 和 java6 时引	当第一次发生 OOM 时, 运行指定命令行指令集。指令集是与 OS 环境相关的, 在 linux 下多数是 bash 脚本, windows 下是 dos 批处理。

-XX:-PrintClassHistogram	入 默认 不启 用	在 Windows 下, 按 ctrl-break 或 Linux 下是执行 kill -3( 发送 SIGQUIT 信号 ) 时, 打印 class 柱状图。  Jmap -histo pid 也实现了相同的功能。 详 见 <a href="http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html">http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html</a>
-XX:-PrintConcurrentLocks	默认 不启 用	在 thread dump 的同时, 打印 java.util.concurrent 的锁状态。  Jstack -l pid 也同样实现了同样的功能。 详 见 <a href="http://java.sun.com/javase/6/docs/technotes/tools/share/jstack.html">http://java.sun.com/javase/6/docs/technotes/tools/share/jstack.html</a>
-XX:-PrintCommandLineFlags	5.0 引入, 默认 不启 用	Java 启动时, 往 stdout 打印当前启用的非稳态 jvm options。  例如 : -XX:+UseConcMarkSweepGC -XX:+HeapDumpOnOutOfMemoryError -XX:+DoEscapeAnalysis
-XX:-PrintCompilation	默认 不启 用	往 stdout 打印方法被 JIT 编译时的信息。  例如 : 1        java.lang.String::charAt (33 bytes)
-XX:-PrintGC	默认 不启 用	开启 GC 日志打印。  打印格式例如 : [Full GC 131115K->7482K(1015808K), 0.1633180 secs]  该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。 详 见 <a href="http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump">http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump</a>
-XX:-PrintGCDetails	1.4.0 引入, 默认 不启 用	打印 GC 回收的细节。  打印格式例如 : [Full GC (System) [Tenured: 0K->2394K(466048K), 0.0624140 secs] 30822K->2394K(518464K), [Perm : 10443K->10443K(16384K)], 0.0625410 secs] [Times: user=0.05 sys=0.01, real=0.06 secs]  该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。 详 见 <a href="http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump">http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump</a>
-XX:-PrintGCTimeStamps	默认 不启 用	打印 GC 停顿耗时。  打印格式例如 : <b>2.744:</b> [Full GC (System) 2.744: [Tenured: 0K->2441K(466048K), 0.0598400 secs]

		<p>31754K-&gt;2441K(518464K), [Perm : 10717K-&gt;10717K(16384K)], 0.0599570 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]</p> <p>该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。 详见 <a href="http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump">http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap_Dump</a></p>
-XX:-PrintTenuringDistribution	默认不启用	<p>打印对象的存活期限信息。</p> <p>打印格式例如： [GC Desired survivor size 4653056 bytes, new threshold 32 (max 32) - age 1: 2330640 bytes, 2330640 total - age 2: 9520 bytes, 2340160 total 204009K-&gt;21850K(515200K), 0.1563482 secs]</p> <p>Age1 2 表示在第 1 和 2 次 GC 后存活的对象大小。</p>
-XX:-TraceClassLoading	默认不启用	<p>打印 class 装载信息到 stdout。记 Loaded 状态。</p> <p>例如： [Loaded java.lang.Object from /opt/taobao/install/jdk1.6.0_07/jre/lib/rt.jar]</p>
-XX:-TraceClassLoadingPreorder	1.4.2 引入，默认不启用	<p>按 class 的引用/依赖顺序打印类装载信息到 stdout。不同于 TraceClassLoading，本选项只记 Loading 状态。</p> <p>例如： [Loading java.lang.Object from /home/confsrv/jdk1.6.0_14/jre/lib/rt.jar]</p>
-XX:-TraceClassResolution	1.4.2 引入，默认不启用	<p>打印所有静态类，常量的代码引用位置。用于 debug。</p> <p>例如： RESOLVE java.util.HashMap java.util.HashMap\$Entry HashMap.java:209</p> <p>说明 HashMap 类的 209 行引用了静态类 java.util.HashMap\$Entry</p>
-XX:-TraceClassUnloading	默认不启用	<p>打印 class 的卸载信息到 stdout。记 Unloaded 状态。</p>
-XX:-TraceLoaderConstraints	Java6 引入，默认不启用	<p>打印 class 的装载策略变化信息到 stdout。</p> <p>例如： [Adding new constraint for name: java/lang/String, loader[0]: sun/misc/Launcher\$ExtClassLoader, loader[1]: &lt;bootloader&gt; ] [Setting class object in existing constraint for name: [Ljava/lang/Object; and loader sun/misc/Launcher\$ExtClassLoader ] [Updating constraint for name org/xml/sax/InputStream, loader &lt;bootloader&gt;, by setting class object ] [Extending constraint for name java/lang/Object by</p>

		adding loader[15]: sun/reflect/DelegatingClassLoader ]  装载策略变化是实现 classloader 隔离/名称空间一致性的关键技术。 对此感兴趣的朋友，详见 <a href="http://kenwublog.com/docs/Dynamic+Class+Loading+in+the+Java+Virtual+Machine.pdf">http://kenwublog.com/docs/Dynamic+Class+Loading+in+the+Java+Virtual+Machine.pdf</a> 中的 constraint rules一章。
-XX:+PerfSaveDataToFile	默认 启用	当 java 进程因 OOM 或 crashed 被强制终止后，生成一个堆快照文件(什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述)。

### 作者敬告

完善的单元测试 功能回归测试 和性能基准测试可以减少因调整非稳态 JVM 选项带来的风险。

### 参考资料

Java6 性能调优白皮书

[http://java.sun.com/performance/reference/whitepapers/6\\_performance.html](http://java.sun.com/performance/reference/whitepapers/6_performance.html)

Java6 GC 调优指南

[http://java.sun.com/javase/technologies/hotspot/gc/gc\\_tuning\\_6.html](http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html)

更为全面的 options 列表

<http://blogs.sun.com/watt/resource/jvm-options-list.html>