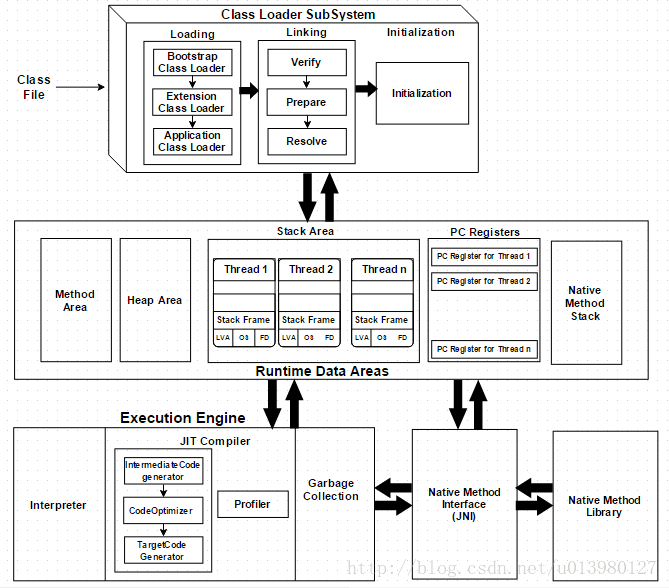
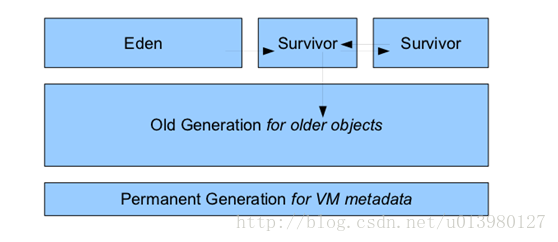
**1. JVM堆内存划分**

这两天看到下面这篇文章的图不错。

[一图读懂JVM架构解析](http://geek.csdn.net/news/detail/131976)



**1.1 JDK7及以前的版本**

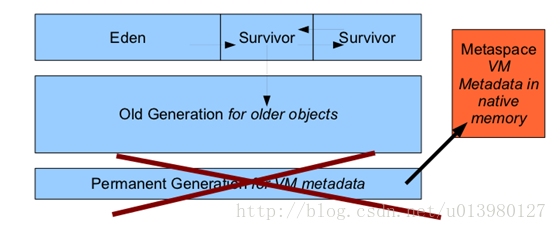


其中最上一层是Nursery内存，一个对象被创建以后首先被放到Nursery中的Eden内   
存中，如果存活期超两个Survivor之后就会被转移到长时内存中(Old Generation)中。

永久内存中存放着对象的方法、变量等元数据信息。通过如果永久内存不够，就会得到如下错误：

Java.lang.OutOfMemoryError: PermGen

## 1.2 JDK8版本



JDK8中把存放元数据中的永久内存从堆内存中移到了本地内存(native memory)中，这样永久内存就不再占用堆内存，它可以通过自动增长来避免JDK7以及前期版本中常见的永久内存错误(java.lang.OutOfMemoryError: PermGen)。

JDK8也提供了一个新的设置Matespace内存大小的参数：

-XX:MaxMetaspaceSize=128m

* 1

注意：如果不设置JVM将会根据一定的策略自动增加本地元内存空间。如果你设置的元内存空间过小，你的应用程序可能得到以下错误：

java.lang.OutOfMemoryError: Metadata space

**-XX 参数被称为不稳定参数，此类参数的设置很容易引起JVM性能上的差异。**

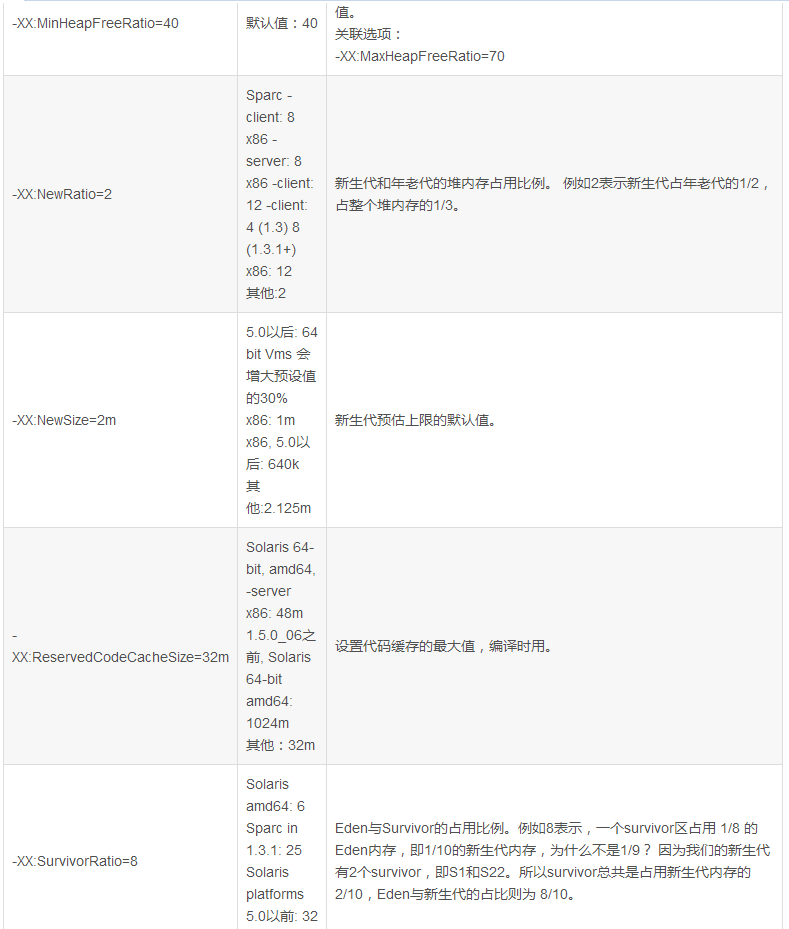


**2.1 行为选项**

| **选项** | **默认值** | **描述** |
| --- | --- | --- |
| -XX:-AllowUserSignalHandlers | 限于Linux和Solaris默认关闭 | 允许为java进程安装信号处理器。 |
| -XX:AltStackSize=16384 | 仅适用于Solaris，从5.0中删除 | 备用信号堆栈大小（以字节为单位） |
| -XX:-DisableExplicitGC | 默认关闭 | 禁止在运行期显式地调用 System.gc()。 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。 |
| -XX:+FailOverToOldVerifier | Java6新引入选项，默认启用 | 如果新的Class校验器检查失败，则使用老的校验器。解决兼容性问题。 关联选项：-XX:+UseSplitVerifier |
| -XX:+HandlePromotionFailure | Java1.5以前默认关闭 Java1.6后默认启用 | 新生代收集担保，于年老代预留内存。 |
| -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:PreBlockSpin | 默认10 | 控制多线程自旋锁优化的自旋次数  前置选项： -XX:+UseSpinning |
| -XX:-RelaxAccessControlCheck | 默认关闭 Java1.6引入 | 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:+ScavengeBeforeFullGC | 默认启用 | 在Full GC前触发一次Minor GC |
| -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:-UseConcMarkSweepGC | 默认关闭 Java1.4引入 | 启用CMS低停顿垃圾收集器。 |
| -XX:+UseGCOverheadLimit | 默认启用 Java1.6引入 | 限制GC的运行时间。如果GC耗时过长，就抛OutOfMemoryError。 |
| -XX:+UseLWPSynchronization | 限于solaris, 默认启用, Java1.4引入 | 使用轻量级进程（内核线程）替换线程同步。 |
| -XX:-UseParallelGC | -server时启用, 其他情况下：默认关闭, Java1.4引入 | 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 | 默认关闭, Java1.5引入 | 为老年代和新生代都使用并行清除的垃圾收集器。开启此选项将自动开启-XX:+UseParallelGC 选项 |
| -XX:-UseSerialGC | -client时启用, 默认关闭, Java1.5引入 | 使用串行垃圾收集器。 |
| -XX:-UseSpinning | Java1.4.2和1.5需要手动启用, Java1.6默认已启用 | 启用多线程自旋锁优化。  关联选项： -XX:PreBlockSpin=10 |
| -XX:+UseTLAB | Java1.4.2以前和使用-client选项时：默认关闭, 其余版本默认启用 | 启用线程本地缓存区(Thread Local) |
| -XX:+UseSplitVerifier | Java1.5默认关闭, Java1.6默认启用 | 使用新的Class类型校验器 。  关联选项:  -XX:+FailOverToOldVerifier |
| -XX:+UseThreadPriorities | 默认启用 | 使用本地线程的优先级。 |
| -XX:+UseVMInterruptibleIO | 限于solaris, 默认启用, Java1.6引入 | 在solaris中，允许运行时中断线程。 |

## 2.2 性能选项









**2.3 调试选项**

| **选项** | **默认值** | **描述** |
| --- | --- | --- |
| -XX:-CITime | 默认启用 | 打印JIT编译器编译耗时。 |
| -XX:ErrorFile=./hs\_err\_pid.log | Java1.6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 默认关闭 | 启用dtrace诊断 |
| -XX:HeapDumpPath=./java\_pid.hprof | 默认是java进程启动位置 | 堆内存快照的存储文件路径。 |
| -XX:-HeapDumpOnOutOfMemoryError | 默认关闭 | 在java.lang.OutOfMemoryError 异常出现时，输出一个dump.core文件，记录当时的堆内存快照（见 -XX:HeapDumpPath 的描述） |
| -XX:OnError=”\;\” | Java1.4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.sh脚本，windows下是.bat批处理。 |
| -XX:OnOutOfMemoryError=”\;\” | Java1.4.2 update 12和Java6时引入 | 当第一次发生java.lang.OutOfMemoryError 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.sh脚本，windows下是.bat批处理。 |
| -XX:-PrintClassHistogram | 默认关闭 | 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。 |
| -XX:-PrintConcurrentLocks | 默认关闭 | 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。 |
| -XX:-PrintCommandLineFlags | Java1.5 引入，默认关闭 | 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 | 默认关闭 | 往stdout打印方法被JIT编译时的信息。 |
| -XX:-PrintGC | 默认关闭 | 开启GC日志打印。 显示结果例如：  [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。 |
| -XX:-PrintGCDetails | Java1.4引入，默认关闭 | 打印GC回收的详细信息。  显示结果例如：  [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。 |
| -XX:-PrintGCTimeStamps | 默认关闭 | 打印GC停顿耗时。  显示结果例如： 2.744: [Full GC (System) 2.744: [Tenured: 0K->2441K(466048K), 0.0598400 secs] 31754K->2441K(518464K), [Perm : 10717K->10717K(16384K)], 0.0599570 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]  该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 jconsole 动态启用。 |
| -XX:-PrintTenuringDistribution | 默认关闭 | 打印对象的存活期限信息。  显示结果例如： [GC Desired survivor size 4653056 bytes, new threshold 32 (max 32) - age 1: 2330640 bytes, 2330640 total - age 2: 9520 bytes, 2340160 total 204009K->21850K(515200K), 0.1563482 secs]  Age1,2表示在第1和2次GC后存活的对象大小。 |
| -XX:-TraceClassLoading | 默认关闭 | 打印class装载信息到stdout。记Loaded状态。  例如： [Loaded java.lang.Object from /opt/taobao/install/jdk1.6.0\_07/jre/lib/rt.jar] |
| -XX:-TraceClassLoadingPreorder | 1.4.2引入，默认关闭 | 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。  例如： [Loading java.lang.Object from /home/confsrv/jdk1.6.0\_14/jre/lib/rt.jar] |
| -XX:-TraceClassResolution | 1.4.2引入，默认关闭 | 打印所有静态类，常量的代码引用位置。用于debug。  例如：  RESOLVE java.util.HashMap java.util.HashMapEntryHashMap.java:209<br>说明HashMap类的209行引用了静态类java.util.HashMapEntryHashMap.java:209<br>说明HashMap类的209行引用了静态类java.util.HashMapEntry |
| -XX:-TraceClassUnloading | 默认关闭 | 打印class的卸载信息到stdout。记Unloaded状态。 |
| -XX:-TraceLoaderConstraints | Java1.6 引入，默认关闭 | 打印class的装载策略变化信息到stdout。 装载策略变化是实现classloader隔离/名称空间一致性的关键技术。 |
| -XX:+PerfSaveDataToFile | 默认启用 | 当java进程因java.lang.OutOfMemoryError 异常或crashed 被强制终止后，生成一个堆快照文件。 |
| -XX:ParallelGCThreads=n | 默认值：随JVM运行平台不同而异 | 配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。 |
| -XX:+UseCompressedOops | 32位默认关闭，64位默认启动 | 使用compressed pointers。这个参数默认在64bit的环境下默认启动，但是如果JVM的内存达到32G后，这个参数就会默认为不启动，因为32G内存后，压缩就没有多大必要了，要管理那么大的内存指针也需要很大的宽度了 |
| -XX:+AlwaysPreTouch | 默认关闭 | 在JVM 初始化时预先对Java堆进行摸底。 |
| -XX:AllocatePrefetchDistance=n | 默认值取决于当前JVM 设置 | 为对象分配设置预取距离。 |
| -XX:InlineSmallCode=n | 默认值取决于当前JVM 设置 | 当编译的代码小于指定的值时,内联编译的代码。 |
| -XX:MaxInlineSize=35 | 默认值：35 | 内联方法的最大字节数。 |
| -XX:FreqInlineSize=n | 默认值取决于当前JVM 设置 | 内联频繁执行的方法的最大字节码大小。 |
| -XX:LoopUnrollLimit=n | 默认值取决于当前JVM 设置 | 代表节点数目小于给定值时打开循环体。 |
| -XX:InitialTenuringThreshold=7 | 默认值：7 | 设置初始的对象在新生代中最大存活次数。 |
| -XX:MaxTenuringThreshold=n | 默认值：15，最大值：15 | 设置对象在新生代中最大的存活次数，最大值15，并行回收机制默认为15，CMS默认为4。 |
| -Xloggc: | 默认关闭 | 输出GC 详细日志信息至指定文件。 |
| -XX:-UseGCLogFileRotation | 默认关闭 | 开启GC 日志文件切分功能，前置选项 -Xloggc |
| -XX:NumberOfGClogFiles=1 | 必须>=1，默认值：1 | 设置切分GC 日志文件数量，文件命名格式：.0, .1, …, .n-1 |
| -XX:GCLogFileSize=8K | 必须>=8K，默认值：8K | GC日志文件切分大小。 |

# 3. 垃圾收集器

## 3.1 串行回收器

### 3.1.1 新生代串行回收器

**特点**

1. 它仅仅使用单线程进行垃圾回收
2. 它是独占式的垃圾回收
3. 进行垃圾回收时, Java应用程序中的线程都需要暂停(Stop-The-World)
4. 使用复制算法
5. 适合CPU等硬件不是很好的场合

**设置参数**

-XX:+UseSerialGC 指定新生使用新生代串行收集器和老年代串行收集器, 当以client模式运行时, 它是默认的垃圾收集器

* 1

### 3.1.2 老年代串行回收器

**特点**

1. 同新生代串行回收器一样, 单线程, 独占式的垃圾回收器
2. 通常老年代垃圾回收比新生代回收要更长时间, 所以可能会使应用程序停顿较长时间

**设置参数**

-XX:+UseSerialGC 新生代, 老年代都使用串行回收器

-XX:+UseParNewGC 新生代使用ParNew回收器, 老年代使用串行回收器

-XX:+UseParallelGC 新生代使用ParallelGC回收器, 老年代使用串行回收器

* 1
* 2
* 3

## 3.2 并行回收器

### 3.2.1 新生代ParNew回收器

**特点**

1. 将串行回收多线程化
2. 使用复制算法
3. 垃圾回收时, 应用程序仍会暂停, 只不过由于是多线程回收, 在多核CPU上，回收效率会高于串行回收器, 反之在单核CPU， 效率会不如串行回收器

**设置参数**

-XX:+UseParNewGC 新生代使用ParNew回收器, 老年代使用串行回收器

-XX:+UseConcMarkSweepGC 新生代使用ParNew回收器, 老年代使用CMS回收器

-XX:ParallelGCThreads=n 指回ParNew回收器工作时的线程数量, cpu核数小时8时, 其值等于cpu数量, 高于8时,可以使用公式(3+((5\*CPU\_count)/8))

* 1
* 2
* 3

### 3.2.2 新生代ParallelGC回收器

**特点**

1. 同ParNew回收器一样， 不同的地方在于，它非常关注系统的吞吐量(通过参数控制)
2. 使用复制算法
3. 支持自适应的GC调节策略

**设置参数**

-XX:+UseParallelGC　　新生代用ParallelGC回收器, 老年代使用串行回收器

-XX:+UseParallelOldGC　　新生代用ParallelGC回收器, 老年代使用ParallelOldGC回收器系统吞吐量的控制:

-XX:MaxGCPauseMillis=n(单位ms) 　　设置垃圾回收的最大停顿时间,

-XX:GCTimeRatio=n(n在0-100之间)　　设置吞吐量的大小, 假设值为n, 那系统将花费不超过1/(n+1)的时间用于垃圾回收

-XX:+UseAdaptiveSizePolicy　　打开自适应GC策略, 在这种模式下, 新生代的大小, eden,survivior的比例, 晋升老年代的对象年龄等参数会被自动调整,以达到堆大小, 吞吐量, 停顿时间之间的平衡点

* 1
* 2
* 3
* 4
* 5

### 3.2.3 老年代ParallelOldGC回收器

**特点**

1. 同新生代的ParallelGC回收器一样, 是属于老年代的关注吞吐量的多线程并发回收器
2. 使用标记压缩算法

**设置参数**

-XX:+UseParallelOldGC　　新生代用ParallelGC回收器, 老年代使用ParallelOldGC回收器, 是非常关注系统吞吐量的回收器组合, 适合用于对吞吐量要求较高的系统

-XX:ParallelGCThreads=n 　　指回ParNew回收器工作时的线程数量, cpu核数小时8时, 其值等于cpu数量, 高于8时, 可以使用公式(3+((5\*CPU\_count)/8))

* 1
* 2

## 3.3 CMS回收器(Concurrent Mark Sweep，并发标记清除)

### 3.3.1 老年代的并发回收器

**特点**

1. 是并发回收, 非独占式的回收器, 大部分时候应用程序不会停止运行
2. 针对年老代的回收器
3. 使用并发标记清除算法, 因此回收后会有内存碎片, 可以使参数设置进行内存碎片的压缩整理
4. 与ParallelGC和ParallelOldGC不同, CMS主要关注系统停顿时间

**主要步骤**

1. 初始标记
2. 并发标记
3. 预清理
4. 重新标记
5. 并发清理
6. 并发重置

注:初始标记与重新标记是独占系统资源的，不能与用户线程一起执行，而其它阶段则可以与用户线程一起执行

**设置参数**

-XX:-CMSPrecleaningEnabled　　关闭预清理, 不进行预清理, 默认在并发标记后, 会有一个预清理的操作，可减少停顿时间

-XX:+UseConcMarkSweepGC　　老年代使用CMS回收器, 新生代使用ParNew回收器

-XX:ConcGCThreads=n　　设置并发线程数量,

-XX:ParallelCMSThreads=n　　同上, 设置并发线程数量,

-XX:CMSInitiatingOccupancyFraction=n　　指定老年代回收阀值, 即当老年代内存使用率达到这个值时, 会执行一次CMS回收，默认值为68, 设置技巧: (Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100)>=Xmn

-XX:+UseCMSCompactAtFullCollection　　开启内存碎片的整理, 即当CMS垃圾回收完成后, 进行一次内存碎片整理, 要注意内存碎片的整理并不是并发进行的, 因此可能会引起程序停顿

-XX:CMSFullGCsBeforeCompation=n　　用于指定进行多少次CMS回收后， 再进行一次内存压缩

-XX:+CMSParallelRemarkEnabled　　在使用UseParNewGC 的情况下, 尽量减少 mark 的时间

-XX:+UseCMSInitiatingOccupancyOnly　　表示只有达到阀值时才进行CMS回收

-XX:+CMSConcurrentMTEnabled 当该标志被启用时，并发的CMS阶段将以多线程执行，默认开启

-XX:+CMSIncrementalMode 在增量模式下，CMS 收集器在并发阶段，不会独占整个周期，

而会周期性的暂停，唤醒应用线程。收集器把并发阶段工作，划分为片段，安排在次级(minor) 回收之间运行。这对需要低延迟，运行在少量CPU服务器上的应用很有用。

### 3.3.2 Class的回收(永久区的回收)

**设置参数**

-XX:+CMSClassUnloadingEnabled　　开启回收Perm区的内存, 默认情况下, 是需要触发一次FullGC

-XX:CMSInitiatingPermOccupancyFraction=n　　当永久区占用率达到这个n值时，启动CMS回收, 需上一个参数开启的情况下使用

* 1
* 2

## 3.4 G1回收器(jdk1.7后全新的回收器, 用于取代CMS)

### 3.4.1 概述

**特点**

1. 独特的垃圾回收策略, 属于分代垃圾回收器
2. 使用分区算法, 不要求eden, 年轻代或老年代的空间都连续
3. 并行性: 回收期间, 可由多个线程同时工作, 有效利用多核cpu资源
4. 并发性: 与应用程序可交替执行, 部分工作可以和应用程序同时执行
5. 分代GC: 分代收集器, 同时兼顾年轻代和老年代
6. 空间整理: 回收过程中, 会进行适当对象移动, 减少空间碎片
7. 可预见性: G1可选取部分区域进行回收, 可以缩小回收范围, 减少全局停顿

**主要步骤**

1. 新生代GC
2. 并发标记周期

初始标记新生代GC(此时是并行, 应用程序会暂停止)–>根区域扫描–>并发标记–>重新标记(此时是并行, 应用程序会暂停止)–>独占清理(此时应用程序会暂停止)–>并发清理

1. 混合回收

这个阶段即会执行正常的年轻代gc, 也会选取一些被标记的老年代区域进行回收, 同时处理新生代和年老轻

1. 若需要, 会进行FullGC

混合GC时发生空间不足  
在新生代GC时, survivor区和老年代无法容纳幸存对象时  
以上两者都会导致一次FullGC产生

**设置参数**

-XX:+UseG1GC　　打开G1收集器开关，

-XX:MaxGCPauseMillis=n　　指定目标的最大停顿时间，任何一次停顿时间超过这个值, G1就会尝试调整新生代和老年代的比例, 调整堆大小, 调整晋升年龄

-XX:ParallelGCThreads=n　　用于设置并行回收时, GC的工作线程数量

-XX:InitiatingHeapOccpancyPercent=n　　指定整个堆的使用率达到多少时, 执行一次并发标记周期, 默认45， 过大会导致并发标记周期迟迟不能启动, 增加FullGC的可能, 过小会导致GC频繁, 会导致应用程序性能有所下降

* 1
* 2
* 3
* 4

### 3.4.2 参数选项

| **选项** | **默认值** | **描述** |
| --- | --- | --- |
| -XX:+UseG1GC | 默认关闭 | 使用G1垃圾处理器 |
| -XX:MaxGCPauseMillis=n | 默认值：4294967295 | 设置并行收集最大暂停时间，这是一个理想目标，JVM将尽最大努力来实现它。 |
| -XX:InitiatingHeapOccupancyPercent=n | 默认值：45 | 启动一个并发垃圾收集周期所需要达到的整堆占用比例。这个比例是指整个堆的占用比例而不是某一个代（例如G1）,如果这个值是0则代表‘持续做GC’。默认值是45 |
| -XX:NewRatio=n | 默认值：2 | 设置年轻代和年老代的比值。例如:值为3，则表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4。 |
| -XX:SurvivorRatio=n | 默认值：8 | 年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5 |
| -XX:MaxTenuringThreshold=n | 默认值：15 | 设置垃圾最大存活阀值。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。 |
| -XX:ParallelGCThreads=n | 默认值：随JVM运行平台不同而异 | 配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。 |
| -XX:ConcGCThreads=n | 默认值：随JVM运行平台不同而异 | 并行垃圾收集时，使用的线程数。默认值和JVM运行的平台有关。 |
| -XX:G1ReservePercent=n | 默认值：10 | 设置保留用来做假天花板以减少晋升（新生代对象晋升到老生代）失败可能性的堆数目。 |
| -XX:G1HeapRegionSize=n | 默认值根据堆大小而定 | 使用G1垃圾回收器，java堆被划分成统一大小的区块。这个选项设置每个区块的大小。最小值是1Mb，最大值是32Mb。 |

### 3.4.3 其他GC相关的设置

1. System.gc()
2. (1)禁用System.gc()
3. -XX:+DisableExplicitGC　　禁止程序中调用System.gc()， 加了此参数, 程序若有调用, 返回的空函数调用
4. System.gc()的调用, 会使用FullGC的方式回收整个堆而会忽略CMS或G1等相关回收器
5. (２)System.gc()使用并发回收

　　-XX:+ExplicitGCCinvokesConcurrent 　　使用并发方式处理显示的gc, 即开启后, System.gc()这种显示GC才会并发的回收, (CMS, G1)

* + 1
  + 2
  + 3
  + 4
  + 5

1. 并行GC前额外触发的新生代GC
2. (1)使用并行回收器(UseParallelGC或者UseParallelOldGC)时, 会额外先触发一个新生代GC, 目的是尽可能减少停顿时间
3. (2)若不需要这种特性, 可以使用以下参数去除

　　-XX:-ScavengeBeforeFullGC 　　即去除在FullGC之前的那次新生代GC, 原本默认值为true

* + 1
  + 2
  + 3

1. 对象何时进入老年代
2. (1)当对象首次创建时, 会放在新生代的eden区, 若没有GC的介入，会一直在eden区, GC后，是可能进入survivor区或者年老代
3. (2)当对象年龄达到一定的大小 ,就会离开年轻代， 进入老年代， 对象进入老年代的事件称为晋升, 而对象的年龄是由GC的次数决定的, 每一次GC，若对象没有被回收, 则对象的年龄就会加1, 可以使用以下参数来控制新生代对象的最大年龄：
4. -XX:MaxTenuringThreshold=n　　假设值为n ， 则新生代的对象最多经历n次GC， 就能晋升到老年代, 但这个必不是晋升的必要条件
5. -XX:TargetSurvivorRatio=n　　用于设置Survivor区的目标使用率，即当survivor区GC后使用率超过这个值, 就可能会使用较小的年龄作为晋升年龄
6. (3)除年龄外, 对象体积也会影响对象的晋升的, 若对象体积太大, 新生代无法容纳这个对象, 则这个对象可能就会直接晋升至老年代, 可通过以下参数使用对象直接晋升至老年代的阈值, 单位是byte

　　-XX:PretenureSizeThreshold　　即对象的大小大于此值, 就会绕过新生代, 直接在老年代分配, 此参数只对串行回收器以及ParNew回收有效, 而对ParallelGC回收器无效

1. 在TLAB上分配对象(Thread Local Allocation Buffer, 线程本地分配缓存)
2. (1)TLAB: TLAB是一个线程专用的内存分配区域, 虚拟机为线程分配空间, 针对于体积不大的对象, 会优先使用TLAB, 这个可以加速对象的分配, TLAB是默认开启的, 若要关闭可以使用以下参数关闭
3. -XX:-UseTLAB　　关闭TLAB
4. -XX:+UseTLAB　　开启TLAB, 默认也是开启的
5. -XX:+PrintTLAB　　观察TALB的使用情况
6. -XX:TLABRefillWasteFraction=n　　设置一个比率n， 而refill\_waste的值就是(TLAB\_SIZE/n), 即TLAB空间较小， 大对象无法分配在TLAB，所以会直接分配到堆上，TLAB较小也很容易装满, 因此当TLAB的空间不够分配一个新对象, 就会考虑废弃当前TLAB空间还是直接分配到堆上, 就会使用此参数进行判断, 小于refill\_waste就允许废弃, 而新建TLAB来分配对象，而大于refill\_waste就直接在堆上分配, 默认是64
7. -XX:+ResizeTLAB　　开启TLAB自动调整大小, 默认是开启的, 若要关闭把+号换成-号即可

　　-XX:TLABSize=n　　设置一个TLAB的大小, 前提先关闭TLAB的自动调整

# 4. 性能调优工具

## 4.1 jps(Java Virtual Machine Process Status Tool)

主要用来输出JVM中运行的进程状态信息。语法格式如下：

jps [options] [hostid]

# hostid语法如下：

[protocol:][[//]hostname][:port][/servername]

protocol - 如果protocol及hostname都没有指定，那表示的是与当前环境相关的本地协议，如果指定了hostname却没有指定protocol，那么protocol的默认就是rmi。

hostname - 服务器的IP或者名称，没有指定则表示本机。

port - 远程rmi的端口，如果没有指定则默认为1099。

Servername - 注册到RMI注册中心中的jstatd的名称。

# 如果不指定hostid就默认为当前主机或服务器。

# 命令行参数选项

-q 忽略输出类名、Jar名和传入main方法的参数

-m 输出传入main方法的参数

-l 输出main类或Jar的全限名

-v 输出传入JVM的参数

-V 输出通过标记的文件传递给JVM的参数（.hotspotrc文件，或者是通过参数-XX:Flags=<filename>指定的文件）

-J 用于传递jvm选项到由javac调用的java加载器中，例如，“-J-Xms48m”将把启动内存设置为48M，使用-J选项可以非常方便的向基于Java的开发的底层虚拟机应用程序传递参数。

## 4.2 jstack（Java Stack Trace）

主要用来查看某个Java进程内的线程堆栈信息。语法格式如下：

jstack [option] pid

jstack [option] executable core

jstack [option] [server-id@]remote-hostname-or-ip

# 命令行参数选项说明如下：

-l 长列表，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况

-m mixed mode，不仅会输出Java堆栈信息，还会输出C/C++堆栈信息（比如Native方法）

-F 当’jstack [-l] pid’没有相应的时候强制打印栈信息

-h | -help打印帮助信息

jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在JVM性能调优中使用得非常多。

## 4.3 jmap（Java Memory Map）

jmap用来查看堆内存使用状况，一般结合jhat使用。jmap语法格式如下：

如果运行在64位JVM上，可能需要指定-J-d64命令选项参数。

jmap [option] pid

jmap [option] executable core

jmap [option] [server-id@]remote-hostname-or-ip

# 参数说明

executable 产生core dump的java可执行程序

core 将被打印信息的core dump文件

remote-hostname-or-ip 远程debug服务的主机名或ip

server-id 唯一id,假如一台主机上多个远程debug服务

pid 需要被打印配相信息的java进程id，可以用jps查问

# 基本参数

-dump:[live,]format=b,file=<filename> 使用hprof二进制形式,输出jvm的heap内容到文件=. live子选项是可选的，假如指定live选项,那么只输出活的对象到文件

-finalizerinfo 打印正等候回收的对象的信息

-heap 打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况

-histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”\*”. 如果live子参数加上后,只统计活的对象数量

-permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来

-F 强迫.在pid没有相应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效

-h | -help 打印辅助信息

-J 传递参数给jmap启动的jvm

## 4.4 jhat（Java Heap Analysis Tool）

Jhat用于对JAVA heap进行离线分析的工具，它可以对不同虚拟机中导出的heap信息文件进行分析

# jmap -dump:format=b,file=dumpFileName pid

jmap -dump:format=b,file=/tmp/dump.dat 21711

jhat -port 9998 /tmp/dump.dat

# 注意如果Dump文件太大，可能需要加上-J-Xmx512m这种参数指定最大堆内存

# 在浏览器中输入主机地址:9998查看

另外，可以使用Eclipse插件MAT(Memory Analyzer Tool)对dump文件进行分析。

## 4.5 jstat（Java Virtual Machine Statistics Monitoring Tool）

Jstat用于监控基于HotSpot的JVM，对其堆的使用情况进行实时的命令行的统计，使用jstat我们可以对指定的JVM做如下监控：

* 类的加载及卸载情况
* 查看新生代、老生代及持久代的容量及使用情况
* 查看新生代、老生代及持久代的垃圾收集情况，包括垃圾回收的次数及垃圾回收所占用的时间
* 查看新生代中Eden区及Survior区中容量及分配情况等

语法：

jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]

generalOption - 单个的常用的命令行选项，如-help, -options, 或 -version。

outputOptions -一个或多个输出选项，由单个的statOption选项组成，可以和-t, -h, and -J等选项配合使用。

statOption 根据jstat统计的维度不同，可以使用如下表中的选项进行不同维度的统计，不同的操作系统支持的选项可能会不一样，可以通过-options选项，查看不同操作系统所支持选项

-h n 用于指定每隔几行就输出列头，如果不指定，默认是只在第一行出现列头。

-J javaOption 用于将给定的javaOption传给java应用程序加载器，例如，“-J-Xms48m”将把启动内存设置为48M

-t n 用于在输出内容的第一列显示时间戳，这个时间戳代表的时JVM开始启动到现在的时间

vmid VM的进程号，即当前运行的java进程号

interval 间隔时间，单位可以是秒或者毫秒，通过指定s或ms确定，默认单位为毫秒

count 打印次数，如果缺省则打印无数次

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

统计维度与输出

1. class 用于查看类加载情况的统计

| **列名** | **说明** |
| --- | --- |
| Loaded | 加载了的类的数量 |
| Bytes | 加载了的类的大小，单为Kb |
| Unloaded | 卸载了的类的数量 |
| Bytes | 卸载了的类的大小，单为Kb |
| Time | 花在类的加载及卸载的时间 |

2. compiler 用于查看HotSpot中即时编译器编译情况的统计

| **列名** | **说明** |
| --- | --- |
| Compiled | 编译任务执行的次数 |
| Failed | 编译任务执行失败的次数 |
| Invalid | 编译任务非法执行的次数 |
| Time | 执行编译花费的时间 |
| FailedType | 最后一次编译失败的编译类型 |
| FailedMethod | 最后一次编译失败的类名及方法名 |

3. gc 用于查看JVM中堆的垃圾收集情况的统计

| **列名** | **说明** |
| --- | --- |
| S0C | 新生代中Survivor space中S0当前容量的大小（KB） |
| S1C | 新生代中Survivor space中S1当前容量的大小（KB） |
| S0U | 新生代中Survivor space中S0容量使用的大小（KB） |
| S1U | 新生代中Survivor space中S1容量使用的大小（KB） |
| EC | Eden space当前容量的大小（KB） |
| EU | Eden space容量使用的大小（KB） |
| OC | Old space当前容量的大小（KB） |
| OU | Old space使用容量的大小（KB） |
| PC | Permanent space当前容量的大小（KB） |
| PU | Permanent space使用容量的大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| YGCT | 从应用程序启动到采样时 Young GC 所用的时间(秒) |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |
| FGCT | 从应用程序启动到采样时 Full GC 所用的时间(秒) |
| GCT | T从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC |

4. gccapacity 用于查看新生代、老生代及持久代的存储容量情况

| **列名** | **说明** |
| --- | --- |
| NGCMN | 新生代的最小容量大小（KB） |
| NGCMX | 新生代的最大容量大小（KB） |
| NGC | 当前新生代的容量大小（KB） |
| S0C | 当前新生代中survivor space 0的容量大小（KB） |
| S1C | 当前新生代中survivor space 1的容量大小（KB） |
| EC | Eden space当前容量的大小（KB） |
| OGCMN | 老生代的最小容量大小（KB） |
| OGCMX | 老生代的最大容量大小（KB） |
| OGC | 当前老生代的容量大小（KB） |
| OC | 当前老生代的空间容量大小（KB） |
| PGCMN | 持久代的最小容量大小（KB） |
| PGCMX | 持久代的最大容量大小（KB） |
| PGC | 当前持久代的容量大小（KB） |
| PC | 当前持久代的空间容量大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |

5. gccause 用于查看垃圾收集的统计情况（这个和-gcutil选项一样），如果有发生垃圾收集，它还会显示最后一次及当前正在发生垃圾收集的原因。

| **列名** | **说明** |
| --- | --- |
| LGCC | 最后一次垃圾收集的原因，可能为“unknown GCCause”、“System.gc()”等 |
| GCC | 当前垃圾收集的原因 |

6. gcnew 用于查看新生代垃圾收集的情况

| **列名** | **说明** |
| --- | --- |
| S0C | 当前新生代中survivor space 0的容量大小（KB） |
| S1C | 当前新生代中survivor space 1的容量大小（KB） |
| S0U | S0已经使用的大小（KB） |
| S1U | S1已经使用的大小（KB） |
| TT | Tenuring threshold，要了解这个参数，我们需要了解一点Java内存对象的结构，在Sun JVM中，（除了数组之外的）对象都有两个机器字（words）的头部。第一个字中包含这个对象的标示哈希码以及其他一些类似锁状态和等标识信息，第二个字中包含一个指向对象的类的引用，其中第二个字节就会被垃圾收集算法使用到。  在新生代中做垃圾收集的时候，每次复制一个对象后，将增加这个对象的收集计数，当一个对象在新生代中被复制了一定次数后，该算法即判定该对象是长周期的对象，把他移动到老生代，这个阈值叫着tenuring threshold。这个阈值用于表示某个/些在执行批定次数youngGC后还活着的对象，即使此时新生的的Survior没有满，也同样被认为是长周期对象，将会被移到老生代中。 |
| MTT | Maximum tenuring threshold，用于表示TT的最大值。 |
| DSS | Desired survivor size (KB).可以参与这里：<http://blog.csdn.net/yangjun2/article/details/6542357> |
| EC | Eden space当前容量的大小（KB） |
| EU | Eden space已经使用的大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| YGCT | 从应用程序启动到采样时 Young GC 所用的时间(单位秒) |

7. gcnewcapacity 用于查看新生代的存储容量情况

| **列名** | **说明** |
| --- | --- |
| NGCMN | 新生代的最小容量大小（KB） |
| NGCMX | 新生代的最大容量大小（KB） |
| NGC | 当前新生代的容量大小（KB） |
| S0CMX | 新生代中SO的最大容量大小（KB） |
| S0C | 当前新生代中SO的容量大小（KB） |
| S1CMX | 新生代中S1的最大容量大小（KB） |
| S1C | 当前新生代中S1的容量大小（KB） |
| ECMX | 新生代中Eden的最大容量大小（KB） |
| EC | 当前新生代中Eden的容量大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |

8. gcold 用于查看老生代及持久代发生GC的情况

| **列名** | **说明** |
| --- | --- |
| PC | 当前持久代容量的大小（KB） |
| PU | 持久代使用容量的大小（KB） |
| OC | 当前老年代容量的大小（KB） |
| OU | 老年代使用容量的大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |
| FGCT | 从应用程序启动到采样时 Full GC 所用的时间(单位秒) |
| GCT | 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC |

9. gcoldcapacity 用于查看老生代的容量

| **列名** | **说明** |
| --- | --- |
| OGCMN | 老生代的最小容量大小（KB） |
| OGCMX | 老生代的最大容量大小（KB） |
| OGC | 当前老生代的容量大小（KB） |
| OC | 当前新生代的空间容量大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |
| FGCT | 从应用程序启动到采样时 Full GC 所用的时间(单位秒) |
| GCT | 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC |

10. gcpermcapacity 用于查看持久代的容量

| **列名** | **说明** |
| --- | --- |
| PGCMN | 持久代的最小容量大小（KB） |
| PGCMX | 持久代的最大容量大小（KB） |
| PGC | 当前持久代的容量大小（KB） |
| PC | 当前持久代的空间容量大小（KB） |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| FGC | Full GC次数 |
| FGCT | 从应用程序启动到采样时 Full GC 所用的时间(单位秒) |
| GCT | 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC |

11. gcutil 用于查看新生代、老生代及持代垃圾收集的情况

| **列名** | **说明** |
| --- | --- |
| S0 | Heap上的 Survivor space 0 区已使用空间的百分比 |
| S1 | Heap上的 Survivor space 1 区已使用空间的百分比 |
| E | Heap上的 Eden space 区已使用空间的百分比 |
| O | Heap上的 Old space 区已使用空间的百分比 |
| P | Perm space 区已使用空间的百分比 |
| YGC | 从应用程序启动到采样时发生 Young GC 的次数 |
| YGCT | 从应用程序启动到采样时 Young GC 所用的时间(单位秒) |
| FGC | 从应用程序启动到采样时发生 Full GC 的次数 |
| FGCT | 从应用程序启动到采样时 Full GC 所用的时间(单位秒) |
| GCT | 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC |

12. printcompilation HotSpot编译方法的统计

| **列名** | **说明** |
| --- | --- |
| Compiled | 编译任务执行的次数 |
| Size | 方法的字节码所占的字节数 |
| Type | 编译类型 |
| Method | 指定确定被编译方法的类名及方法名，类名中使名“/”而不是“.”做为命名分隔符，方法名是被指定的类中的方法，这两个字段的格式是由HotSpot中的“-XX:+PrintComplation”选项确定的。 |

示例

jstat -gc 1618 250 4

* 1

## 4.6 jvisualvm（Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool）

jvisualvm同jconsole都是一个基于图形化界面的、可以查看本地及远程的JAVA GUI监控工具，Jvisualvm同jconsole的使用方式一样，直接在命令行打入Jvisualvm即可启动，不过Jvisualvm相比，界面更美观一些，数据更实时。

## 4.7 jconsole

一个java GUI监视工具，可以以图表化的形式显示各种数据。并可通过远程连接监视远程的服务器VM。用java写的GUI程序，用来监控VM，并可监控远程的VM，非常易用，而且功能非常强。命令行里打 jconsole，选则进程就可以了。

需要注意的就是在运行jconsole之前，必须要先设置环境变量DISPLAY，否则会报错误，Linux下设置环境变量如下：   
export DISPLAY=:0.0

## 4.8 jprofile

JProfiler是一个需要商业授权的全功能的Java剖析工具（profiler），专用于分析J2SE和J2EE应用程序。

它把CPU、执行绪和内存的剖析组合在一个强大的应用中。JProfiler可提供许多IDE整合和应用服务器整合用途。JProfiler直觉式的GUI让你可以找到效能瓶颈、抓出内存漏失(memory leaks)、并解决执行绪的问题。它让你得以对heap walker作资源回收器的root analysis，可以轻易找出内存漏失；heap快照（snapshot）模式让未被参照（reference）的对象、稍微被参照的对象、或在终结（finalization）队列的对象都会被移除；整合精灵以便剖析浏览器的Java外挂功能。

## 4.9 jca

Java线程分析工具，专业的线程分析工具兼容sun/oracle JDK dump线程堆，图形化显示线程概括信息，非常容易的定位问题。   
jca是一个类工具 启动方法：

java -jar jca433.jar

* 1

## 4.10 jinfo命令(Java Configuration Info)

jinfo可以输出并修改运行时的java 进程的opts。用处比较简单，用于输出JAVA系统参数及命令行参数。用法是jinfo -opt pid 如：查看2788的MaxPerm大小可以用 jinfo -flag MaxPermSize 2788。

## 4.11 Jdb命令(The Java Debugger)

用来对core文件和正在运行的Java进程进行实时地调试，里面包含了丰富的命令帮助您进行调试。

## 4.12 Jstatd命令(Java Statistics Monitoring Daemon)

jstatd是一个基于RMI（Remove Method Invocation）的服务程序，它用于监控基于HotSpot的JVM中资源的创建及销毁，并且提供了一个远程接口允许远程的监控工具连接到本地的JVM执行命令。

jstatd是基于RMI的，所以在运行jstatd的服务器上必须存在RMI注册中心，如果没有通过选项”-p port”指定要连接的端口，jstatd会尝试连接RMI注册中心的默认端口。后面会谈到如何连接到一个默认的RMI内部注册中心，如何禁止默认的RMI内部注册中心的创建，以及如何启动一个外部注册中心。

参数选项

-nr 如果RMI注册中心没有找到，不会创建一个内部的RMI注册中心。

-p port RMI注册中心的端口号，默认为1099。

-n rminame 默认为JStatRemoteHost；如果同一台主机上同时运行了多个jstatd服务，rminame可以用于唯一确定一个jstatd服务；这里需要注意一下，如果开启了这个选项，那么监控客户端远程连接时，必须同时指定hostid及vmid，才可以唯一确定要连接的服务，这个可以参看jps章节中列出远程服务器上Java进程的示例。

-J 用于传递jvm选项到由javac调用的java加载器中，例如，“-J-Xms48m”将把启动内存设置为48M，使用-J选项可以非常方便的向基于Java的开发的底层虚拟机应用程序传递参数。

安全性

jstatd服务只能监视具有适当的本地访问权限的JVM，因此jstatd进程与被监控的JVM必须运行在相同的用户权限中。但是有一些特殊的用户权限，如基于UNIX（TM）为系统的root用户，它有权限访问系统中所有JVM的资源，如果jstatd进程运行在这种权限中，那么它可以监视系统中的所有JVM，但是这也带来了额外的安全问题。

示例

# 使用内部RMI注册中心（默认端口是1099），启动jstatd

jstatd -J-Djava.security.policy=jstatd.all.policy

# 使用外部的RMI注册中心，启动jstatd

rmiregistry&jstatd -J-Djava.security.policy=all.policy

# 外部的RMI注册中心来启动jstatd，此注册中心的端口为2020

rmiregistry 2020&jstatd -J-Djava.security.policy=all.policy -p 2020

# 外部的RMI注册中心来启动jstatd，此注册中心的端口为2020，并且绑定到RMI注册中心的名为AlternateJstatdServerName

rmiregistry 2020&jstatd -J-Djava.security.policy=all.policy -p 2020 -n AlternateJstatdServerName

# 禁止内部RMI注册中心的创建

jstatd -J-Djava.security.policy=all.policy -nr

# 开启RMI日记记录，jstatd运行在开启了日志记录功能的RMI注册中

jstatd -J-Djava.security.policy=all.policy -J-Djava.rmi.server.logCalls=true

## 4.13 httpwatch

网页数据分析工具，对客户端到服务器端的请求，响应数据有效的监控分析。

# 5. 生产环境示例

## 5.1 Tomcat7

**catalina.sh（只运行一个Tomcat）**

# 8G内存

JAVA\_OPTS="

-Dfile.encoding=UTF-8

-server

-Djava.awt.headless=true

-Xms6144m

-Xmx6144m

-XX:NewSize=1024m

-XX:MaxNewSize=2048m

-XX:PermSize=512m

-XX:MaxPermSize=512m

-XX:MaxTenuringThreshold=15

-XX:NewRatio=2

-XX:+AggressiveOpts

-XX:+UseBiasedLocking

-XX:+UseConcMarkSweepGC

-XX:+UseParNewGC

-XX:+CMSParallelRemarkEnabled

-XX:LargePageSizeInBytes=128m

-XX:+UseFastAccessorMethods

-XX:+UseCMSInitiatingOccupancyOnly

-XX:+DisableExplicitGC"

# 16G内存（垃圾回收略）

JAVA\_OPTS="

-Dfile.encoding=UTF-8

-server

-Xms13312m

-Xmx13312m

-XX:NewSize=3072m

-XX:MaxNewSize=4096m

-XX:PermSize=512m

-XX:MaxPermSize=512m

-XX:MaxTenuringThreshold=10

-XX:NewRatio=2

-XX:+DisableExplicitGC"

# 32G内存（垃圾回收略）

JAVA\_OPTS="

-Dfile.encoding=UTF-8

-server

-Xms29696m

-Xmx29696m

-XX:NewSize=6144m

-XX:MaxNewSize=9216m

-XX:PermSize=1024m

-XX:MaxPermSize=1024m

-XX:MaxTenuringThreshold=10

-XX:NewRatio=2

-XX:+DisableExplicitGC"

## 5.2 Hadoop

建议Hadoop进程的GC参数加上如下选项，很多商业版都默认加上了，这对JDK性能提升有很大帮助：

-XX:CMSInitiatingOccupancyFraction=70

-XX:+HeapDumpOnOutOfMemoryError

-XX:+UseConcMarkSweepGC

-XX:-CMSConcurrentMTEnabled

-XX:+CMSIncrementalMode

-Djava.net.preferIPv4Stack=true

# 6. 参考

[JDK8中JVM堆内存划分](http://blog.csdn.net/jia20003/article/details/50703944)

[JDK8引进的JVM参数变化记录](http://blog.csdn.net/jiangguilong2000/article/details/47975457)

[JVM 不稳定参数](http://286.iteye.com/blog/1924947)

[java 8 JVM性能优化](http://www.tuicool.com/articles/juiIbiA)

[Java性能优化攻略详解](http://blog.csdn.net/hj7jay/article/details/51451624)

[Jvm垃圾回收器详细](http://blog.csdn.net/lghuntfor/article/details/51052737)

[JVM性能调优监控工具jps、jstack、jmap、jhat、jstat、hprof使用详解](https://my.oschina.net/feichexia/blog/196575)

[jps命令(Java Virtual Machine Process Status Tool)](http://blog.csdn.net/fenglibing/article/details/6411932)

[Tomcat 8 安装和配置、优化](https://github.com/judasn/Linux-Tutorial/blob/master/Tomcat-Install-And-Settings.md)