JVM参数

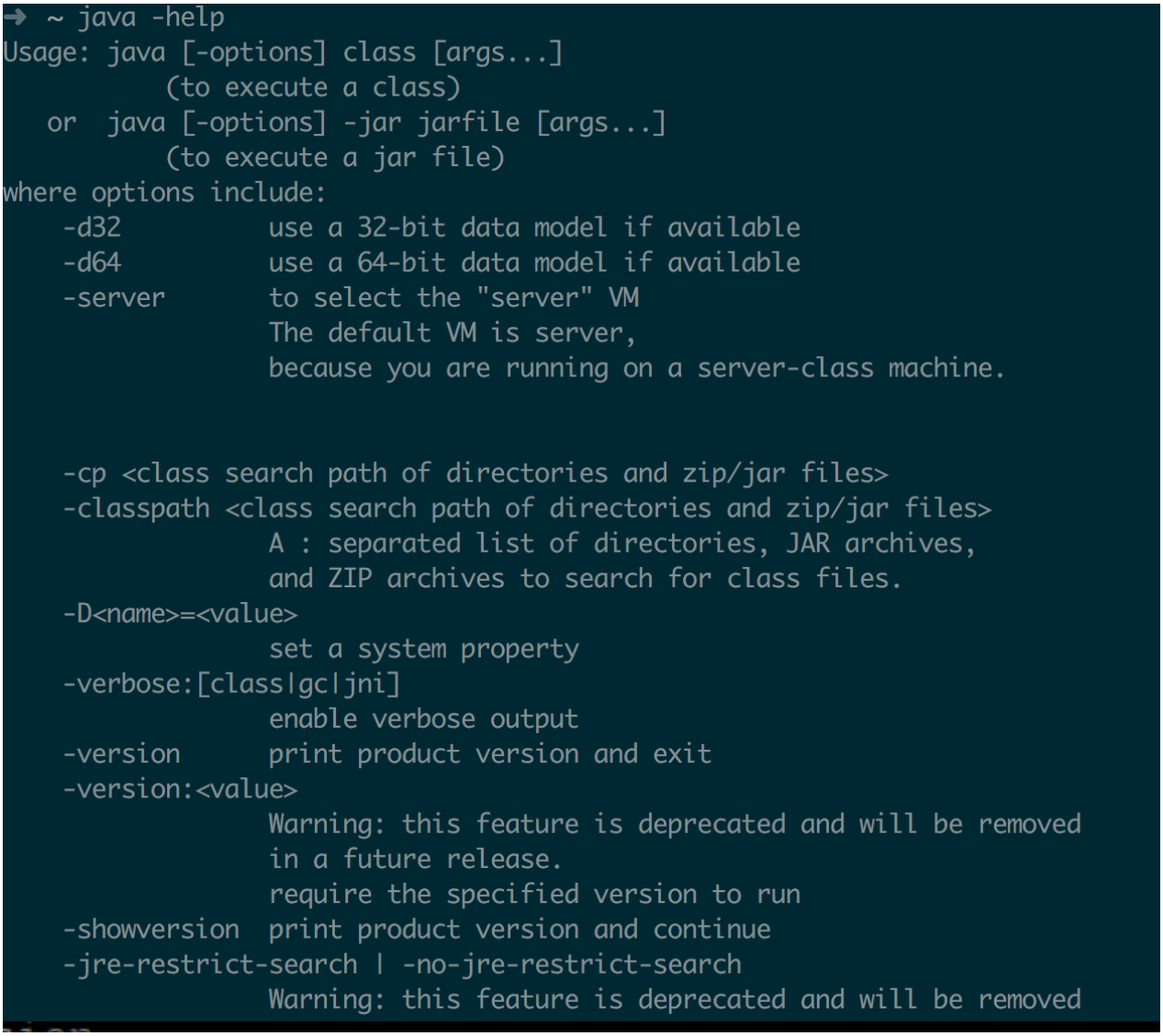
在JVM调整过程中，主要是对JVM参数做的调整，以下我们队JVM主要参数做逐一介绍。JVM参数有很多，其实我们直接使用默认的JVM参数，不去修改都可以满足大多数情况。但是如果你想在有限的硬件资源下，部署的系统达到最大的运行效率，那么进行相关的JVM参数设置是必不可少的。下面我们就来对这些JVM参数进行详细的介绍。

JVM参数主要分为以下三种：标准参数、非标准参数、不稳定参数。

1 标准参数

标准参数，顾名思义，标准参数中包括功能以及输出的结果都是很稳定的，基本上**不会随着JVM版本的变化而变化**。标准参数以-开头，如：java -version、java -jar等，通过java -help可以查询所有的标准参数，

我们可以通过 -help 命令来检索出所有标准参数。

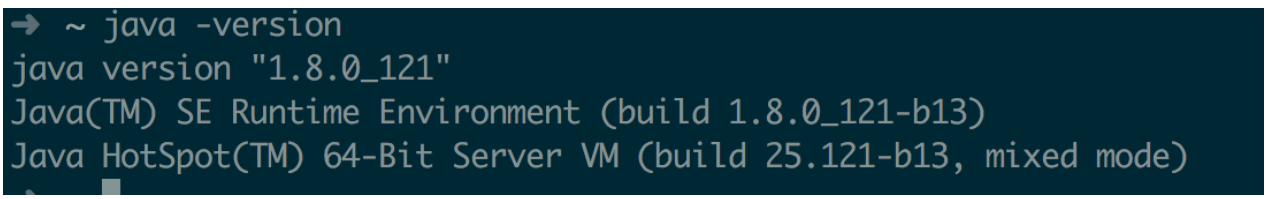


关于这些命令的详细解释，可以参考官网：

https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

-help 也是一个标准参数，再比如使用比较多的 -version也是。

显示Java的版本信息。



2 非标准参数

非标准参数以-X开头，是标准参数的扩展。对应前面讲的标准化参数，这是非标准化参数。表示在将来的JVM版本中可能会发生改变，但是这类以-X开始的参数变化的比较小。

我们可以通过 Java -X 命令来检索所有-X 参数。

```
➜  ~ java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by :>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by :>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by :>
                      prepend in front of bootstrap class path
    -Xdiag            show additional diagnostic messages
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.
    -XshowSettings    show all settings and continue
    -XshowSettings:all
                      show all settings and continue
```

我们可以通过设置非标准参数来配置堆的内存分配，常用的非标准参数有：

-Xmn新生代内存的最大值，包括Eden区和两个Survivor区的总和，写法如：-Xmn1024，-Xmn1024k，-Xmn1024m，-Xmn1g 。

-Xms堆内存的最小值，默认值是总内存/64（且小于1G），默认情况下，当堆中可用内存小于40%(这个值可以用-XX: MinHeapFreeRatio 调整，如-X:MinHeapFreeRatio=30)时，堆内存会开始增加，一直增加到-Xmx的大小。

-Xmx堆内存的最大值，默认值是总内存/64（且小于1G），如果Xms和Xmx都不设置，则两者大小会相同，默认情况下，当堆中可用内存大于70%时，堆内存会开始减少，一直减小到-Xms的大小；

整个堆的大小=年轻代大小+年老代大小，堆的大小不包含持久代大小，如果增大了年轻代，年老代相应就会减小，官方默认的配置为年老代大小/年轻代大小=2/1左右；

建议在开发测试环境可以用Xms和Xmx分别设置最小值最大值，但是在线上生产环境，Xms和Xmx设置的值必须一样，原因与年轻代一样——防止抖动；

-Xss每个线程的栈内存，默认1M，一般来说是不需要改的。

-Xrs减少JVM对操作系统信号的使用。

-Xprof跟踪正运行的程序，并将跟踪数据在标准输出输出；适合于开发环境调试。

-Xnoclassgc关闭针对class的gc功能；因为其阻止内存回收，所以可能会导致OutOfMemoryError错误，慎用；

-Xincgc开启增量gc（默认为关闭）；这有助于减少长时间GC时应用程序出现的停顿；但由于可能和应用程序并发执行，所以会降低CPU对应用的处理能力。

-Xloggc:file与-verbose:gc功能类似，只是将每次GC事件的相关情况记录到一个文件中，文件的位置最好在本地，以避免网络的潜在问题。

3 不稳定参数

这是我们日常开发中接触到最多的参数类型。这也是非标准化参数，相对来说不稳定，随着JVM版本的变化可能会发生变化，主要用于**JVM调优**和debug。

不稳定参数以-XX 开头，此类参数的设置很容易引起JVM 性能上的差异，使JVM存在极大的不稳定性。如果此类参数设置合理将大大提高JVM的性能及稳定性。

不稳定参数分为三类：

性能参数：用于JVM的性能调优和内存分配控制，如内存大小的设置；

行为参数：用于改变JVM的基础行为，如GC的方式和算法的选择；

调试参数：用于监控、打印、输出jvm的信息；

不稳定参数语法规则：

布尔类型参数值：

- -XX:+
- -XX:-

示例：-XX:+UseG1GC：表示启用G1垃圾收集器

数字类型参数值：

- -XX:

示例：-XX:MaxGCPauseMillis=500 ： 表示设置GC的最大停顿时间是500ms

字符串类型参数值：

- -XX:

示例：-XX:HeapDumpPath=./dump.core

4 常用参数

如以下参数示例

```
 —Xms4g –Xmx4g —Xmn1200m —Xss512k –XX:NewRatio=4 –XX:SurvivorRatio=8 –
XX:PermSize=100m –XX:MaxPermSize=256m –XX:MaxTenuringThreshold=15 –
XX:MaxDirectMemorySize=1G –XX:+DisableExplicitGC
```

上面为Java7及以前版本的示例，在Java8中永久代的参数-XX:PermSize和-XX：MaxPermSize已经失效。这在前面章节中已经讲到。

参数解析：

- -Xms4g：初始化堆内存大小为4GB，ms是memory start的简称，等价于-XX:InitialHeapSize。
- -Xmx4g：堆内存最大值为4GB，mx是memory max的简称，等价于-XX:MaxHeapSize。
- -Xmn1200m：设置年轻代大小为1200MB。增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。
- -Xss512k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1MB，以前每个线程堆栈大小为256K。应根据应用线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。
- -XX:NewRatio=4：设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5
- -XX:SurvivorRatio=8：设置年轻代中Eden区与Survivor区的大小比值。设置为8，则两个Survivor区与一个Eden区的比值为2:8，一个Survivor区占整个年轻代的1/10
- -XX:PermSize=100m：初始化永久代大小为100MB。
- -XX:MaxPermSize=256m：设置持久代大小为256MB。
- -XX:MaxTenuringThreshold=15：设置垃圾最大年龄。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:MaxDirectMemorySize=1G：直接内存。报java.lang.OutOfMemoryError: Direct buffer memory异常可以上调这个值。
- -XX:+DisableExplicitGC：禁止运行期显式地调用System.gc()来触发fulll GC。

  注意: Java RMI的定时GC触发机制可通过配置-Dsun.rmi.dgc.server.gcInterval=86400来控制触发的时间。
- -XX:CMSInitiatingOccupancyFraction=60：老年代内存回收阈值，默认值为68。
- -XX:ConcGCThreads=4：CMS垃圾回收器并行线程线，推荐值为CPU核心数。
- -XX:ParallelGCThreads=8：新生代并行收集器的线程数。
- -XX:MaxTenuringThreshold=10：设置垃圾最大年龄。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:CMSMaxAbortablePrecleanTime=500：当abortable-preclean预清理阶段执行达到这个时间时就会结束。

更多详细参数说明请参考官方文档：

https://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

新生代、老生代、永久代的参数，如果不进行指定，虚拟机会自动选择合适的值，同时也会基于系统的开销自动调整。

4.1 -XX:+PrintFlagsFinal

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:+PrintFlagsInitial：查看初始值

-XX:+PrintFlagsFinal：查看最终值（初始值可能被修改掉）

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 -XX:+PrintFlagsFinal
[Global flags]
    uintx AdaptiveSizeDecrementScaleFactor          = 4                                   {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10                                  {product}
    uintx AdaptiveSizePausePolicy                   = 0                                   {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50                                  {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20                                  {product}
    uintx AdaptiveSizePolicyOutputInterval          = 0                                   {product}
    uintx AdaptiveSizePolicyWeight                  = 10                                  {product}
    uintx AdaptiveSizeThroughPutPolicy              = 0                                   {product}
    uintx AdaptiveTimeWeight                        = 25                                  {product}
     bool AdjustConcurrency                         = false                               {product}
     bool AggressiveOpts                            = false                               {product}
     intx AliasLevel                                = 3                                   {C2 product}
     bool AlignVector                               = false                               {C2 product}
     intx AllocateInstancePrefetchLines             = 1                                   {product}
     intx AllocatePrefetchDistance                  = 192                                 {product}
     intx AllocatePrefetchInstr                     = 0                                   {product}
     intx AllocatePrefetchLines                     = 4                                   {product}
     intx AllocatePrefetchStepSize                  = 64                                  {product}
     intx AllocatePrefetchStyle                     = 1                                   {product}
     bool AllowJNIEnvProxy                          = false                               {product}
     bool AllowNonVirtualCalls                      = false                               {product}
     bool AllowParallelDefineClass                  = false                               {product}
     bool AllowUserSignalHandlers                   = false                               {product}
     bool AlwaysActAsServerClassMachine             = false                               {product}
```

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。

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。

```
➜ ~ java -XX:+PrintFlagsInitial
[Global flags]
    uintx AdaptiveSizeDecrementScaleFactor          = 4                                   {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10                                  {product}
    uintx AdaptiveSizePausePolicy                   = 0                                   {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50                                  {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20                                  {product}
    uintx AdaptiveSizePolicyOutputInterval          = 0                                   {product}
    uintx AdaptiveSizePolicyWeight                  = 10                                  {product}
    uintx AdaptiveSizeThroughPutPolicy              = 0                                   {product}
    uintx AdaptiveTimeWeight                        = 25                                  {product}
     bool AdjustConcurrency                         = false                               {product}
     bool AggressiveOpts                            = false                               {product}
     intx AliasLevel                                = 3                                   {C2 product}
     bool AlignVector                               = true                                {C2 product}
     intx AllocateInstancePrefetchLines             = 1                                   {product}
     intx AllocatePrefetchDistance                  = -1                                  {product}
     intx AllocatePrefetchInstr                     = 0                                   {product}
     intx AllocatePrefetchLines                     = 3                                   {product}
     intx AllocatePrefetchStepSize                  = 16                                  {product}
     intx AllocatePrefetchStyle                     = 1                                   {product}
     bool AllowJNIEnvProxy                          = false                               {product}
     bool AllowNonVirtualCalls                      = false                               {product}
```

4.2 -XX:+PrintCommandLineFlags

让我们看下这个参数，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 -XX:+PrintCommandLineFlags
-XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompre
allelGC
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)
where options include:
```

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。

4.3 GC日志相关

设置JVM GC格式日志的主要参数包括如下8个：

1. -XX:+PrintGC 输出简要GC日志
2. -XX:+PrintGCDetails 输出详细GC日志
3. -Xloggc:gc.log 输出GC日志到文件
4. -XX:+PrintGCTimeStamps 输出GC的时间戳（以JVM启动到当期的总时长的时间戳形式）
5. -XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2020-04-26T21:53:59.234+0800）
6. -XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息
7. -verbose:gc : 在JDK 8中，`-verbose:gc`是`-XX:+PrintGC`一个别称，日志格式等价与：`-XX:+PrintGC`。不过在JDK 9中 -XX:+PrintGC被标记为deprecated。

-verbose:gc是一个标准的选项，-XX:+PrintGC是一个实验的选项，建议使用-verbose:gc`替代`-XX:+PrintGC

8. -XX:+PrintReferenceGC 打印年轻代各个引用的数量以及时长

开启GC日志

多种方法都能开启GC的日志功能，其中包括：使用-verbose:gc或-XX:+PrintGC这两个标志中的任意一个能创建基本的GC日志（这两个日志标志实际上互为别名，默认情况下的GC日志功能是关闭的）
使用-XX:+PrintGCDetails标志会创建更详细的GC日志

推荐使用-XX:+PrintGCDetails标志（这个标志默认情况下也是关闭的）；通常情况下使用基本的GC日志很难诊断垃圾回收时发生的问题。

开启GC时间提示

除了使用详细的GC日志，我们还推荐使用-XX:+PrintGCTimeStamps或者-XX:+PrintGCDateStamps，便于我们更精确地判断几次GC操作之间的时间。这两个参数之间的差别在于时间戳是相对于0（依据JVM启动的时间）的值，而日期戳（date stamp）是实际的日期字符串。由于日期戳需要进行格式化，所以它的效率可能会受轻微的影响，不过这种操作并不频繁，它造成的影响也很难被我们感知。

指定GC日志路径

默认情况下GC日志直接输出到标准输出，不过使用-Xloggc:filename标志也能修改输出到某个文件。除非显式地使用-PrintGCDetails标志，否则使用-Xloggc会自动地开启基本日志模式。
使用日志循环（Log rotation）标志可以限制保存在GC日志中的数据量；
对于需要长时间运行的服务器而言，这是一个非常有用的标志，否则累积几个月的数据很可能会耗尽服务器的磁盘。

开启日志滚动输出

通过-XX:+UseGCLogfileRotation -XX:NumberOfGCLogfiles=N -XX:GCLogfileSize=N标志可以控制日志文件的循环。

默认情况下，UseGCLogfileRotation标志是关闭的。它负责打开或关闭GC日志滚动记录功能的。
要求必须设置 -Xloggc参数
开启UseGCLogfileRotation标志后，默认的文件数目是0（意味着不作任何限制），默认的日志文件大小是0（同样也是不作任何限制）。

因此，为了让日志循环功能真正生效，我们必须为所有这些标志设定值。
需要注意的是：

- The size of the log file at which point the log will be rotated, must be >= 8K. 设置滚动日志文件的大小，必须大于8k。
  当前写日志文件大小超过该参数值时，日志将写入下一个文件
- 设置滚动日志文件的个数，必须大于等于1
- 必须设置 -Xloggc 参数

开启语句

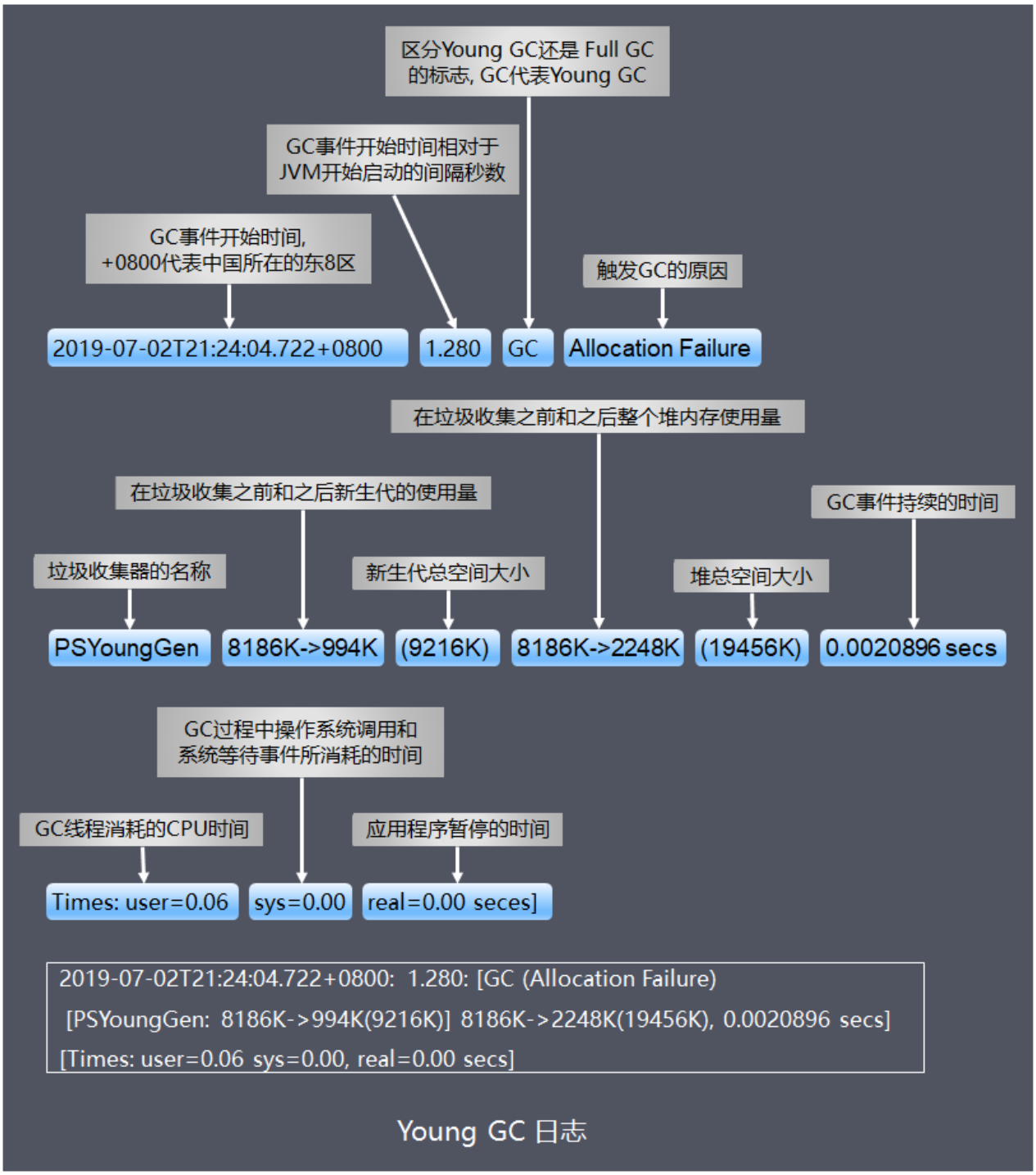
```
-XX:+PrintGCDetails  -XX:+PrintGCDateStamps
-Xloggc:/home/hadoop/gc.log
-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=10  -XX:GCLogFileSize=512k
```

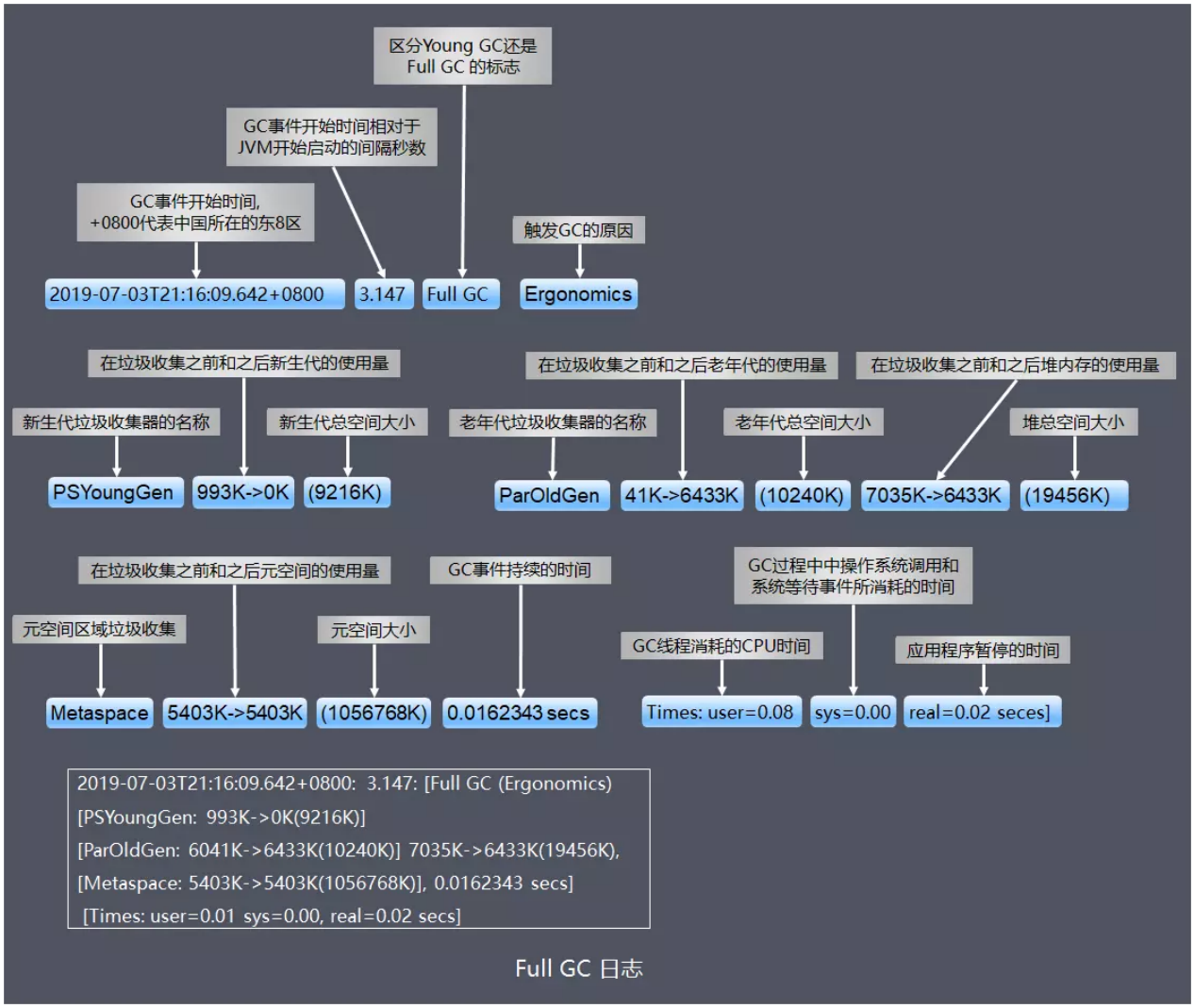
其他有用参数

-XX:+PrintGCApplicationStoppedTime
打印GC造成应用暂停的时间

-XX:+PrintTenuringDistribution
在每次新生代 young GC时,输出幸存区中对象的年龄分布

日志含义


区分Young GC还是 Full GC
的标志, GC代表Young GC
GC事件开始时间相对于
JVM开始启动的间隔秒数
GC事件开始时间,
+0800代表中国所在的东8区
触发GC的原因
2019-07-02T21:24:04.722+0800
1.280
GC
Allocation Failure
在垃圾收集之前和之后整个堆内存使用量
在垃圾收集之前和之后新生代的使用量
GC事件持续的时间
垃圾收集器的名称
新生代总空间大小
堆总空间大小
PSYoungGen
8186K->994K
(9216K)
8186K->2248K
(19456K)
0.0020896 secs
GC过程中操作系统调用和
系统等待事件所消耗的时间
GC线程消耗的CPU时间
应用程序暂停的时间
Times: user=0.06
sys=0.00
real=0.00 seces]
2019-07-02T21:24:04.722+0800: 1.280: [GC (Allocation Failure)
[PSYoungGen: 8186K->994K(9216K)] 8186K->2248K(19456K), 0.0020896 secs]
[Times: user=0.06 sys=0.00, real=0.00 secs]
Young GC 日志



4.4 -XX:CMSFullGCsBeforeCompaction

CMSFullGCsBeforeCompaction 说的是，在上一次CMS并发GC执行过后，到底还要再执行多少次full GC才会做压缩。默认是0，也就是在默认配置下每次CMS GC顶不住了而要转入full GC的时候都会做压缩。 如果把CMSFullGCsBeforeCompaction配置为10，就会让上面说的第一个条件变成每隔10次真正的full GC才做一次压缩（而不是每10次CMS并发GC就做一次压缩，目前VM里没有这样的参数）。这会使full GC更少做压缩，也就更容易使CMS的old gen受碎片化问题的困扰。 本来这个参数就是用来配置降低full GC压缩的频率，以期减少某些full GC的暂停时间。CMS回退到full GC时用的算法是mark-sweep-compact，但compaction是可选的，不做的话碎片化会严重些但这次full GC的暂停时间会短些；这是个取舍。

```
-XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction=10
```

两个参数必须同时使用才能生效。

4.5 -XX:HeapDumpPath

堆内存出现OOM的概率是所有内存耗尽异常中最高的，出错时的堆内信息对解决问题非常有帮助，所以给JVM设置这个参数(**-XX:+HeapDumpOnOutOfMemoryError**)，让JVM遇到OOM异常时能输出堆内信息，并通过（**-XX:+HeapDumpPath**）参数设置堆内存溢出快照输出的文件地址，这对于特别是对相隔数月才出现的OOM异常尤为重要。

这两个参数通常配套使用：

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=./
```

4.6 -XX:OnOutOfMemoryError

```
-
XX:OnOutOfMemoryError="/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Cont
ents/Home/binjconsole"
```

表示发生OOM后，运行jconsole程序。这里可以不用加“”，因为jconsole.exe路径Program Files含有空格。

利用这个参数，我们可以在系统OOM后，自定义一个脚本，可以用来发送邮件告警信息，可以用来重启系统等等。

4.7 XX:InitialCodeCacheSize

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。

4.8 -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。

4.9 其他实用参数

参数名称	含义	默认值	
-Xms	初始堆大小	物理内存的1/64(<1GB)	默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制.
-Xmx	最大堆大小	物理内存的1/4(<1GB)	默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制
-Xmn	年轻代大小(1.4or lator)		**注意**：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小. 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8

-XX:NewSize	设置年轻代大小(for 1.3/1.4)		
-XX:MaxNewSize	年轻代最大值(for 1.3/1.4)		
-XX:PermSize	设置持久代(perm gen)初始值	物理内存的1/64	
-XX:MaxPermSize	设置持久代最大值	物理内存的1/4	
-Xss	每个线程的堆栈大小		JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。（校长） 和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"" -Xss is translated in a VM flag named ThreadStackSize" 一般设置这个值就可以了。
-XX:ThreadStackSize	Thread Stack Size		(0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.]
-XX:NewRatio	年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)		-XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5 Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。
-XX:SurvivorRatio	Eden区与Survivor区的大小比值		设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10
-XX:LargePageSizeInBytes	内存页的大小不可设置过大， 会影响Perm的大小		=128m
-XX:+UseFastAccessorMethods	原始类型的快速优化		
-XX:+DisableExplicitGC	关闭System.gc()		这个参数需要严格的测试
-XX:MaxTenuringThreshold	垃圾最大年龄		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率 该参数只有在串行GC时才有效.
-XX:+AggressiveOpts	加快编译		
-XX:+UseBiasedLocking	锁机制的性能改善		
-Xnoclassgc	禁用垃圾回收		
-XX:SoftRefLRUPolicyMSPerMB	每兆堆空闲空间中SoftReference的存活时间	1s	softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap
-XX:PretenureSizeThreshold	对象超过多大是直接在旧生代分配	0	单位字节 新生代采用Parallel Scavenge GC时无效 另一种直接在旧生代分配的情况是大的数组对象,且数组中无外部引用对象.
-XX:TLABWasteTargetPercent	TLAB占eden区的百分比	1%	

-XX:+*CollectGen0First*	FullGC时是否先YGC	false	

并行收集器相关参数

-XX:+UseParallelGC	**Full GC采用parallel MSC (此项待验证)**		**选择垃圾收集器为并行收集器.此配置仅对年轻代有效.即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集.(此项待验证)**
-XX:+UseParNewGC	设置年轻代为并行收集		可与CMS收集同时使用 JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值
-XX:ParallelGCThreads	并行收集器的线程数		此值最好配置与处理器数目相等 同样适用于CMS
-XX:+UseParallelOldGC	年老代垃圾收集方式为并行收集(Parallel Compacting)		这个是JAVA 6出现的参数选项
-XX:MaxGCPauseMillis	每次年轻代垃圾回收的最长时间(最大暂停时间)		如果无法满足此时间,JVM会自动调整年轻代大小,以满足此值.
-XX:+UseAdaptiveSizePolicy	自动选择年轻代区大小和相应的Survivor区比例		设置此选项后,并行收集器会自动选择年轻代区大小和相应的Survivor区比例,以达到目标系统规定的最低相应时间或者收集频率等,此值建议使用并行收集器时,一直打开.
-XX:GCTimeRatio	设置垃圾回收时间占程序运行时间的百分比		公式为1/(1+n)
-XX:+*ScavengeBeforeFullGC*	Full GC前调用YGC	true	Do young generation GC prior

CMS参数设置

-XX:+UseConcMarkSweepGC	**使用CMS内存收集**		**测试中配置这个以后,-XX:NewRatio=4的配置失效了,原因不明.所以,此时年轻代大小最好用-Xmn设置.???**
-XX:+AggressiveHeap			试图是使用大量的物理内存 长时间大内存使用的优化，能检查计算资源（内存， 处理器数量） 至少需要256MB内存 大量的CPU / 内存， （在1.4.1在4CPU的机器上已经显示有提升）
-XX:CMSFullGCsBeforeCompaction	多少次后进行内存压缩		由于并发收集器不对内存空间进行压缩,整理,所以运行一段时间以后会产生"碎片",使得运行效率降低.此值设置运行多少次GC以后对内存空间进行压缩,整理.
-XX:+CMSParallelRemarkEnabled	降低标记停顿		
-XX+UseCMSCompactAtFullCollection	在FULL GC的时候， 对年老代的压缩		CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。 可能会影响性能,但是可以消除碎片
-XX:+UseCMSInitiatingOccupancyOnly	使用手动定义初始化定义开始CMS收集		禁止hostspot自行触发CMS GC
-XX:CMSInitiatingOccupancyFraction=70	使用cms作为垃圾回收 使用70%后开始CMS收集	92	为了保证不出现promotion failed(见下面介绍)错误,该值的设置需要满足以下公式**CMSInitiatingOccupancyFraction计算公式**
-XX:CMSInitiatingPermOccupancyFraction	设置Perm Gen使用到达多少比率时触发	92	
-XX:+CMSIncrementalMode	设置为增量模式		用于单CPU情况
-XX:+CMSClassUnloadingEnabled			

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()。 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:+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。 为什么会失败？ 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。 关联选项： -XX:+UseSplitVerifier
-	Java1.5以前默认关闭Java1.6	关闭新生代收集担保。 **什么是新生代收集担保？** 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)。 ** 为什么要关闭新生代收集担保？ ** 因为在

XX:+HandlePromotionFailure	后默认启用	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。 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survior或年老代不够用时，抛出promotion failed异常。
-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=10	默认值：10	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) 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	使用串行垃圾收集器。

	入	
-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。 **自旋锁优化原理** 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。 为了避免进入OS互斥，Java6的开发者们提出了自旋锁优化。 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。 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 。 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。 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。 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。

G1 垃圾收集选项

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	Number of threads concurrent garbage collectors will use. The default value varies with the platform on which the JVM is running.
-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。

性能选项：

选项	默认值	描述
-XX:+AggressiveOpts	Java1.5 引入默认关闭 Java1.6后默认开启	开启编译器性能优化。
-XX:CompileThreshold=10000	默认值：1000	通过JIT编译器，将方法编译成机器码的触发阀值，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。 [-client: 1,500]
-XX:LargePageSizeInBytes=4m	默认值：4mamd64位：2m	设置堆内存的内存最大值。
		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。 **什么是预估上限值?** JVM在启动时，会申请最大值（-Xmx指定的

-XX:MaxHeapFreeRatio=70	默认值：70	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。从Java1.4开始, MaxNewSize成为NewRatio的一个函数
-XX:MaxPermSize=64m	Java1.5以后:：64 bit VMs会增大预设值的30%1.4 amd64:：96m1.3.1 -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	默认值：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) 关联选项： -XX:MaxHeapFreeRatio=70
-XX:NewRatio=2	Sparc -client: 8 x86 -server: 8 x86 -client: 12 -client: 4 (1.3) 8 (1.3.1+) x86: 12 其他:2	新生代和年老代的堆内存占用比例。 例如2例如2表示新生代占年老代的1/2，占整个堆内存的1/3。
-XX:NewSize=2m	5.0以后: 64 bit Vms 会增大预设值的30%x86: 1mx86, 5.0以后: 640k 其他:2.125m	新生代预估上限的默认值。
-XX:ReservedCodeCacheSize=32m	Solaris 64-bit, amd64, -server x86: 48m 1.5.0_06之前, Solaris 64-bit amd64: 1024m 其他：32m	设置代码缓存的最大值，编译时用。
	Solaris amd64: 6	

-XX:SurvivorRatio=8	Sparc in 1.3.1: 25 Solaris platforms 5.0以前: 32 其他：8	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的新生代内存，为什么不是1/9？ 因为我们的新生代有2个survivor，即S1和S22。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。
-XX:TargetSurvivorRatio=50	默认值：50	实际使用的survivor空间大小占比。默认是47%，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：512.	线程堆栈大小。
-XX:+UseBiasedLocking	Java1.5 update 6后引入默认关闭。 Java1.6默认启用。	启用偏向锁。实例详解
-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。 Intimate Shared Memory.
-XX:+UseLargePages	Java1.5 update 5后引入默认关闭Java1.6 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Java Support for Large Memory Pages 关联选项： -XX:LargePageSizeInBytes=4m
-XX:+UseMPSS	Java1.4.1 之前默认关闭其他版本默认启用	启用solaris的MPSS，不能与ISM同时使用。
-XX:+UseStringCache	默认开启	缓存常用字符串。
-XX:AllocatePrefetchLines=1	默认值：1	在使用JIT生成的预读取指令分配对象后读取的缓存行数。如果上次分配的对象是一个实例则默认值是1，如果是一个数组则是3
-XX:AllocatePrefetchStyle=1	默认值：1	预读取指令的生成代码风格 0- 无预读取指令生成 1-在每次分配后执行预读取命令 2-当预读取指令执行后使用TLAB()分配水印指针来找回入口
-XX:+UseCompressedStrings	Java1.6 update 21 引入	其中，对于不需要16位字符的字符串，可以使用byte[] 而非char[]。对于许多应用，这可以节省内存，但速度较慢（5%-10%）
-XX:+OptimizeStringConcat	Java1.6 update 20 引入	在可能的情况下优化字符串连接操作。

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。 **什么是堆内存快照？** 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 javaheapDump.hprof 解析快照文件，可以使用jhat, eclipse MAT，gdb等工具。
-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.HashMap$Entry HashMap.java:209 说明HashMap类的209行引用了静态类 java.util.HashMap$Entry
-XX:-TraceClassUnloading	默认关闭	打印class的卸载信息到stdout。记Unloaded状态。
		打印class的装载策略变化信息到stdout。 例如： [Adding new constraint for name: java/lang/String, loader[0]: sun/misc/Launcher$ExtClassLoader, loader[1]:] [Setting

-XX:-TraceLoaderConstraints	Java1.6引入，默认关闭	class object in existing constraint for name: [Ljava/lang/Object; and loader sun/misc/Launcher$ExtClassLoader] [Updating constraint for name org/xml/sax/InputSource, loader , by setting class object] [Extending constraint for name java/lang/Object by 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。
-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。