<https://www.cnblogs.com/emberd/p/5973516.html>

java启动参数共分为三类；

其一是**标准参数**（-），所有的JVM实现都必须实现这些参数的功能，而且向后兼容；  
其二是**非标准参数**（-X），默认jvm实现这些参数的功能，但是并不保证所有jvm实现都满足，且不保证向后兼容；  
其三是**非Stable参数**（-XX），此类参数各个jvm实现会有所不同，将来可能会随时取消，需要慎重使用；

**一．标准参数**

|  |  |
| --- | --- |
| **-client** | 设置jvm使用client模式，特点是启动速度比较快，但运行时性能和内存管理效率不高，通常用于客户端应用程序或者PC应用开发和调试。 |
| **-server** | 设置jvm使server模式，特点是启动速度比较慢，但运行时性能和内存管理效率很高，适用于生产环境。在具有64位能力的jdk环境下将默认启用该模式，而忽略-client参数。 |
| **-jar** | 指定以jar包的形式执行一个应用程序。 要这样执行一个应用程序，必须让jar包的manifest文件中声明初始加载的Main-class，当然那Main-class必须有public static void main(String[] args)方法。 |
| **-agentlib:**libname  **[=options]** | 用于装载本地lib包； 其中 libname为本地代理库文件名，默认搜索路径为环境变量PATH中的路径，options为传给本地库启动时的参数，多个参数之间用逗号分隔。在 Windows平台上jvm搜索本地库名为libname.dll的文件，在linux上jvm搜索本地库名为libname.so的文件，搜索路径环境变量在不同系统上有所不同，比如Solaries上就默认搜索LD\_LIBRARY\_PATH。 比如：-agentlib:hprof用来获取jvm的运行情况，包括CPU、内存、线程等的运行数据，并可输出到指定文件中；windows中搜索路径为JRE\_HOME/bin/hprof.dll。 |
| **-agentpath:**  pathname **[=options]** | 按全路径装载本地库，不再搜索PATH中的路径；其他功能和agentlib相同；更多的信息待续，在后续的JVMTI部分会详述。 |
| **-javaagent:**jarpath  **[=options]** | 指定jvm启动时装入java语言设备代理。 Jarpath 文件中的mainfest文件必须有Agent-Class属性。代理类也必须实现公共的静态public static void premain(String agentArgs, Instrumentation inst)方法（和main方法类似）。当jvm初始化时，将按代理类的说明顺序调用premain方法；具体参见 java.lang.instrument软件包的描述。 |
| **-classpath**classpath  **或**  **-cp**classpath | 告知jvm搜索目录名、jar文档名、zip文档名，之间用分号;分隔；使用-classpath后jvm将不再使用CLASSPATH中的类搜索路径，如果-classpath和CLASSPATH都没有设置，则jvm使用当前路径(.)作为类搜索路径。  jvm搜索类的方式和顺序为：Bootstrap，Extension，User。 Bootstrap中的路径是jvm自带的jar或zip文件，jvm首先搜索这些包文件，用System.getProperty("sun.boot.class.path")可得到搜索路径。 Extension是位于JRE\_HOME/lib/ext目录下的jar文件，jvm在搜索完Bootstrap后就搜索该目录下的jar文件，用System.getProperty("java.ext.dirs")可得到搜索路径。 User搜索顺序为当前路径.、CLASSPATH、-classpath，jvm最后搜索这些目录，用System.getProperty("java.class.path")可得到搜索路径。 |
| **-D**property=value | 设置系统属性名/值对，运行在此jvm之上的应用程序可用System.getProperty("property")得到value的值。 如果value中有空格，则需要用双引号将该值括起来，如-Dname="space string"。 该参数通常用于设置系统级全局变量值，如配置文件路径，以便该属性在程序中任何地方都可访问。 |
| **-verbose  -verbose:class** | 输出jvm载入类的相关信息，当jvm报告说找不到类或者类冲突时可此进行诊断。 |
| **-verbose:gc** | 输出每次GC的相关情况。 |
| **-verbose:jni** | 输出native方法调用的相关情况，一般用于诊断jni调用错误信息。 |
| **-version** | 输出java的版本信息，比如jdk版本、vendor、model。 |

**二．非标准参数**

|  |  |
| --- | --- |
| **-Xms**n | 指定jvm堆的初始大小，默认为物理内存的1/64，最小为1M；可以指定单位，比如k、m，若不指定，则默认为字节。 |
| **-Xmx**n | 指定jvm堆的最大值，默认为物理内存的1/4或者1G，最小为2M；单位与-Xms一致。 |
| **-Xmn**n | 指定jvm堆中年轻代的大小 |
| **-Xss**n | 设置单个线程栈的大小，一般默认为512k。 |
| **-Xint** | 设置jvm以解释模式运行，所有的字节码将被直接执行，而不会编译成本地码。 |
| **-Xbatch** | 关闭后台代码编译，强制在前台编译，编译完成之后才能进行代码执行； 默认情况下，jvm在后台进行编译，若没有编译完成，则前台运行代码时以解释模式运行。 |
| **-Xbootclasspath:**  bootclasspath | 让jvm从指定路径（可以是分号分隔的目录、jar、或者zip）中加载bootclass，用来替换jdk的rt.jar；若非必要，一般不会用到； |
| **-Xbootclasspath**  **/a:**path | 将指定路径的所有文件追加到默认bootstrap路径中； |
| **-Xbootclasspath**  **/p:**path | 让jvm优先于bootstrap默认路径加载指定路径的所有文件； |
| **-Xcheck:jni** | 对JNI函数进行附加check；此时jvm将校验传递给JNI函数参数的合法性，在本地代码中遇到非法数据时，jmv将报一个致命错误而终止；使用该参数后将造成性能下降，请慎用。 |
| **-Xfuture** | 让jvm对类文件执行严格的格式检查（默认jvm不进行严格格式检查），以符合类文件格式规范，推荐开发人员使用该参数。 |
| **-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。 若与verbose命令同时出现在命令行中，则以-Xloggc为准。 |
| **-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。 |
| **-Xrs** | 减少jvm对操作系统信号（signals）的使用，该参数从1.3.1开始有效；  从jdk1.3.0开始，jvm允许程序在关闭之前还可以执行一些代码（比如关闭数据库的连接池），即使jvm被突然终止；  jvm 关闭工具通过监控控制台的相关事件而满足以上的功能；更确切的说，通知在关闭工具执行之前，先注册控制台的控制handler，然后对 CTRL\_C\_EVENT, CTRL\_CLOSE\_EVENT,CTRL\_LOGOFF\_EVENT, and CTRL\_SHUTDOWN\_EVENT这几类事件直接返回true。  但如果jvm以服务的形式在后台运行（比如servlet引擎），他能接收CTRL\_LOGOFF\_EVENT事件，但此时并不需要初始化关闭程序；为了避免类似冲突的再次出现，从jdk1.3.1开始提供-Xrs参数；当此参数被设置之后，jvm将不接收控制台的控制handler，也就是说他不监控和处理CTRL\_C\_EVENT, CTRL\_CLOSE\_EVENT, CTRL\_LOGOFF\_EVENT, orCTRL\_SHUTDOWN\_EVENT事件。 |

**三．非Stable参数**

　　前面我们提到用-XX作为前缀的参数列表在jvm中可能是不健壮的，SUN也不推荐使用，后续可能会在没有通知的情况下就直接取消了；但是由于这些参数中的确有很多是对我们很有用的，比如我们经常会见到的-XX:PermSize、-XX:MaxPermSize等等；

下面我们将就Java HotSpot VM中**-XX:**的可配置参数列表进行描述；  
这些参数可以被松散的聚合成三类：  
**行为参数**（Behavioral Options）：用于改变jvm的一些基础行为；  
**性能调优**（Performance Tuning）：用于jvm的性能调优；  
**调试参数**（Debugging Options）：一般用于打开跟踪、打印、输出等jvm参数，用于显示jvm更加详细的信息；

　　由于sun官方文档中对各参数的描述也都非常少（大多只有一句话），而且大多涉及OS层面的东西，很难描述清楚，所以以下是挑选了一些我们开发中可能会用得比较多的配置项，若需要查看所有参数列表，可以点击[HotSpot VM Specific Options](http://java.sun.com/docs/hotspot/VMOptions.html).查看原文；

首先来介绍**行为参数**：

|  |  |
| --- | --- |
| -XX:-DisableExplicitGC | 禁止调用System.gc()；但jvm的gc仍然有效 |
| -XX:+MaxFDLimit | 最大化文件描述符的数量限制 |
| -XX:+ScavengeBeforeFullGC | 新生代GC优先于Full GC执行 |
| -XX:+UseGCOverheadLimit | 在抛出OOM之前限制jvm耗费在GC上的时间比例 |
| **-XX:-UseConcMarkSweepGC** | **对老生代采用并发标记交换算法进行GC** |
| **-XX:-UseParallelGC** | **启用并行GC** |
| -XX:-UseParallelOldGC | 对Full GC启用并行，当-XX:-UseParallelGC启用时该项自动启用 |
| **-XX:-UseSerialGC** | **启用串行GC** |
| -XX:+UseThreadPriorities | 启用本地线程优先级 |

上面表格中黑体的三个参数代表着jvm中GC执行的三种方式，即**串行、并行、并发**；  
串行**（SerialGC）**是jvm的默认GC方式，一般适用于小型应用和单处理器，算法比较简单，GC效率也较高，但可能会给应用带来停顿；  
并行**（ParallelGC）**是指GC运行时，对应用程序运行没有影响，GC和app两者的线程在并发执行，这样可以最大限度不影响app的运行；  
并发**（ConcMarkSweepGC）**是指多个线程并发执行GC，一般适用于多处理器系统中，可以提高GC的效率，但算法复杂，系统消耗较大；

**性能调优**参数列表：

|  |  |
| --- | --- |
| 参数及其默认值 | 描述 |
| -XX:LargePageSizeInBytes=4m | 设置用于Java堆的大页面尺寸 |
| -XX:MaxHeapFreeRatio=70 | GC后java堆中空闲量占的最大比例 |
| **-XX:MaxNewSize=size** | **新生成对象能占用内存的最大值** |
| **-XX:MaxPermSize=64m** | **老生代对象能占用内存的最大值** |
| -XX:MinHeapFreeRatio=40 | GC后java堆中空闲量占的最小比例 |
| -XX:NewRatio=2 | 新生代内存容量与老生代内存容量的比例 |
| -XX:SurvivorRatio | 新生代中survivor区和eden区的比例 |
| **-XX:NewSize=2.125m** | **新生代对象生成时占用内存的默认值** |
| -XX:ReservedCodeCacheSize=32m | 保留代码占用的内存容量 |
| -XX:ThreadStackSize=512 | 设置线程栈大小，若为0则使用系统默认值 |
| -XX:+UseLargePages | 使用大页面内存 |

**调试参数**列表：

|  |  |
| --- | --- |
| 参数及其默认值 | 描述 |
| -XX:-CITime | 打印消耗在JIT编译的时间 |
| -XX:ErrorFile=./hs\_err\_pid<pid>.log | 保存错误日志或者数据到文件中 |
| -XX:-ExtendedDTraceProbes | 开启solaris特有的dtrace探针 |
| **-XX:HeapDumpPath=./java\_pid<pid>.hprof** | **指定导出堆信息时的路径或文件名** |
| **-XX:-HeapDumpOnOutOfMemoryError** | **当首次遭遇OOM时导出此时堆中相关信息** |
| -XX:OnError="" | 出现致命ERROR之后运行自定义命令 |
| -XX:OnOutOfMemoryError="" | 当首次遭遇OOM时执行自定义命令 |
| -XX:-PrintClassHistogram | 遇到Ctrl-Break后打印类实例的柱状信息，与jmap -histo功能相同 |
| **-XX:-PrintConcurrentLocks** | **遇到Ctrl-Break后打印并发锁的相关信息，与jstack -l功能相同** |
| -XX:-PrintCommandLineFlags | 打印在命令行中出现过的标记 |
| -XX:-PrintCompilation | 当一个方法被编译时打印相关信息 |
| -XX:-PrintGC | 每次GC时打印相关信息 |
| -XX:-PrintGC Details | 每次GC时打印详细信息 |
| -XX:-PrintGCTimeStamps | 打印每次GC的时间戳 |
| -XX:-TraceClassLoading | 跟踪类的加载信息 |
| -XX:-TraceClassLoadingPreorder | 跟踪被引用到的所有类的加载信息 |
| -XX:-TraceClassResolution | 跟踪常量池 |
| -XX:-TraceClassUnloading | 跟踪类的卸载信息 |
| -XX:-TraceLoaderConstraints | 跟踪类加载器约束的相关信息 |

分类: [学习总结](http://www.cnblogs.com/emberd/category/647576.html)