

笔记来源：尚硅谷 JVM 全套教程，百万播放，全网巅峰（宋红康详解 java 虚拟机）

同步更新：https://gitee.com/vectorx/NOTE_JVM

https://codechina.csdn.net/qq_35925558/NOTE_JVM

https://github.com/uxiahnan/NOTE_JVM

[toc]

4. JVM 运行时参数

4.1. JVM 参数选项

官网地址：<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html>

4.1.1. 类型一：标准参数选项

```
> java -help
用法: java [-options] class [args...]
        (执行类)
    或 java [-options] -jar jarfile [args...]
        (执行 jar 文件)
其中选项包括:
    -d32          使用 32 位数据模型 (如果可用)
    -d64          使用 64 位数据模型 (如果可用)
    -server       选择 "server" VM
                  默认 VM 是 server.

    -cp <目录和 zip/jar 文件的类搜索路径>
    -classpath <目录和 zip/jar 文件的类搜索路径>
                  用 ; 分隔的目录, JAR 档案
                  和 ZIP 档案列表, 用于搜索类文件.
    -D<名称>=<值>
                  设置系统属性
    -verbose:[class|gc|jni]
                  启用详细输出
    -version       输出产品版本并退出
    -version:<值>
                  警告: 此功能已过时, 将在
                  未来发行版中删除.
                  需要指定的版本才能运行
    -showversion  输出产品版本并继续
    -jre-restrict-search | -no-jre-restrict-search
                  警告: 此功能已过时, 将在
                  未来发行版中删除.
                  在版本搜索中包括/排除用户专用 JRE
    -? -help      输出此帮助消息
    -X            输出非标准选项的帮助
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
```

```

        按指定的粒度启用断言
-da[:<packagename>...|:<classname>]
-disableassertions[:<packagename>...|:<classname>]
        禁用具有指定粒度的断言
-esa | -enablesystemassertions
        启用系统断言
-dsa | -disablesystemassertions
        禁用系统断言
-agentlib:<libname>[=<选项>]
        加载本机代理库 <libname>, 例如 -agentlib:hprof
        另请参阅 -agentlib:jwp=help 和 -agentlib:hprof=help
-agentpath:<pathname>[=<选项>]
        按完整路径名加载本机代理库
-javaagent:<jarpath>[=<选项>]
        加载 Java 编程语言代理, 请参阅 java.lang.instrument
-splash:<imagepath>
        使用指定的图像显示启动屏幕

```

有关详细信息, 请参阅

<http://www.oracle.com/technetwork/java/javase/documentation/index.html>。

Server 模式和 Client 模式

Hotspot JVM 有两种模式, 分别是 server 和 client, 分别通过 -server 和 -client 模式设置

- 32 位系统上, 默认使用 Client 类型的 JVM。要想使用 Server 模式, 机器配置至少有 2 个以上的 CPU 和 2G 以上的物理内存。client 模式适用于对内存要求较小的桌面应用程序, 默认使用 Serial 串行垃圾收集器
- 64 位系统上, 只支持 server 模式的 JVM, 适用于需要大内存的应用程序, 默认使用并行垃圾收集器

官网地址: <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html>

如何知道系统默认使用的是那种模式呢?

通过 java -version 命令: 可以看到 Server VM 字样, 代表当前系统使用是 Server 模式

```

> java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)

```

4.1.2. 类型二: -X 参数选项

```

> java -X
-Xmixed          混合模式执行 (默认)
-Xint            仅解释模式执行
-Xbootclasspath:<用 ; 分隔的目录和 zip/jar 文件>
                  设置搜索路径以引导类和资源
-Xbootclasspath/a:<用 ; 分隔的目录和 zip/jar 文件>
                  附加在引导类路径末尾

```

```

-Xbootclasspath/p:<用 ; 分隔的目录和 zip/jar 文件>
                        置于引导类路径之前
-Xdiag                  显示附加诊断消息
-Xnoclassgc             禁用类垃圾收集
-Xincgc                 启用增量垃圾收集
-Xloggc:<file>          将 GC 状态记录在文件中（带时间戳）
-Xbatch                 禁用后台编译
-Xms<size>              设置初始 Java 堆大小
-Xmx<size>              设置最大 Java 堆大小
-Xss<size>              设置 Java 线程堆栈大小
-Xprof                  输出 cpu 配置文件数据
-Xfuture                启用最严格的检查，预期将来的默认值
-Xrs                    减少 Java/VM 对操作系统信号的使用（请参阅文档）
-Xcheck:jni             对 JNI 函数执行其他检查
-Xshare:off             不尝试使用共享类数据
-Xshare:auto            在可能的情况下使用共享类数据（默认）
-Xshare:on              要求使用共享类数据，否则将失败。
-XshowSettings          显示所有设置并继续
-XshowSettings:all      显示所有设置并继续
                        显示所有设置并继续
-XshowSettings:vm       显示所有与 vm 相关的设置并继续
-XshowSettings:properties 显示所有属性设置并继续
                        显示所有属性设置并继续
-XshowSettings:locale   显示所有与区域设置相关的设置并继续

```

-X 选项是非标准选项，如有更改，恕不另行通知。

如何知道 JVM 默认使用的是混合模式呢？

同样地，通过 java -version 命令：可以看到 mixed mode 字样，代表当前系统使用的是混合模式

4.1.3. 类型三：-XX 参数选项

Boolean 类型格式

```

-XX:+<option>  启用option属性
-XX:-<option>  禁用option属性

```

非 Boolean 类型格式

```

-XX:<option>=<number>  设置option数值，可以带单位如k/K/m/M/g/G
-XX:<option>=<string>  设置option字符值

```

4.2. 添加 JVM 参数选项

eclipse 和 idea 中配置不必多说，在 Run Configurations 中 VM Options 中配置即可，大同小异

运行 jar 包

```
java -Xms100m -Xmx100m -XX:+PrintGCDetails -XX:+PrintGCDateStamps -  
XX:+PrintGCTimeStamps -jar demo.jar
```

Tomcat 运行 war 包

```
# linux下catalina.sh添加  
JAVA_OPTS="-Xms512M -Xmx1024M"  
# windows下catalina.bat添加  
set "JAVA_OPTS=-Xms512M -Xmx1024M"
```

程序运行中

```
# 设置Boolean类型参数  
jinfo -flag [+|-]<name> <pid>  
# 设置非Boolean类型参数  
jinfo -flag <name>=<value> <pid>
```

4.3. 常用的 JVM 参数选项

4.3.1. 打印设置的 XX 选项及值

```
-XX:+PrintCommandLineFlags 程序运行时JVM默认设置或用户手动设置的XX选项  
-XX:+PrintFlagsInitial 打印所有XX选项的默认值  
-XX:+PrintFlagsFinal 打印所有XX选项的实际值  
-XX:+PrintVMOptions 打印JVM的参数
```

4.3.2. 堆、栈、方法区等内存大小设置

```
# 栈  
-Xss128k <==> -XX:ThreadStackSize=128k 设置线程栈的大小为128K  
  
# 堆  
-Xms2048m <==> -XX:InitialHeapSize=2048m 设置JVM初始堆内存为2048M  
-Xmx2048m <==> -XX:MaxHeapSize=2048m 设置JVM最大堆内存为2048M  
-Xmn2g <==> -XX:NewSize=2g -XX:MaxNewSize=2g 设置年轻代大小为2G  
-XX:SurvivorRatio=8 设置Eden区与Survivor区的比值，默认为8  
-XX:NewRatio=2 设置老年代与年轻代的比例，默认为2  
-XX:+UseAdaptiveSizePolicy 设置大小比例自适应，默认开启  
-XX:PretenureSizeThreshold=1024 设置让大于此阈值的对象直接分配在老年代，只对Serial、ParNew收集器有效  
-XX:MaxTenuringThreshold=15 设置新生代晋升老年代的年龄限制，默认为15
```

```
-XX:TargetSurvivorRatio 设置MinorGC结束后Survivor区占用空间的期望比例

# 方法区
-XX:MetaspaceSize / -XX:PermSize=256m 设置元空间/永久代初始值为256M
-XX:MaxMetaspaceSize / -XX:MaxPermSize=256m 设置元空间/永久代最大值为256M
-XX:+UseCompressedOops 使用压缩对象
-XX:+UseCompressedClassPointers 使用压缩类指针
-XX:CompressedClassSpaceSize 设置Klass Metaspace的大小, 默认1G

# 直接内存
-XX:MaxDirectMemorySize 指定DirectMemory容量, 默认等于Java堆最大值
```


4.3.3. OutOfMemory 相关的选项

```
-XX:+HeapDumpOnOutOfMemoryError 内存出现OOM时生成Heap转储文件, 两者互斥
-XX:+HeapDumpBeforeFullGC 出现FullGC时生成Heap转储文件, 两者互斥
-XX:HeapDumpPath=<path> 指定heap转储文件的存储路径, 默认当前目录
-XX:OnOutOfMemoryError=<path> 指定可行性程序或脚本的路径, 当发生OOM时执行脚本
```

4.3.4. 垃圾收集器相关选项

首先需了解垃圾收集器之间的搭配使用关系

- 红色虚线表示在 jdk8 时被 Deprecate, jdk9 时被删除
- 绿色虚线表示在 jdk14 时被 Deprecate
- 绿色虚框表示在 jdk9 时被 Deprecate, jdk14 时被删除

image-20210506182458663

```
# Serial回收器
-XX:+UseSerialGC 年轻代使用Serial GC, 老年代使用Serial Old GC
# ParNew回收器
-XX:+UseParNewGC 年轻代使用ParNew GC
-XX:ParallelGCThreads 设置年轻代并行收集器的线程数。
    一般地, 最好与CPU数量相等, 以避免过多的线程数影响垃圾收集性能。
```

```
$$ ParallelGCThreads = \begin{cases} CPU\_Count & \& \text{ (CPU\_Count <= 8) } \\ 3 + (5 * CPU\_Count / 8) & \& \text{ (CPU\_Count > 8) } \end{cases} $$
```

```
# Parallel回收器
-XX:+UseParallelGC 年轻代使用 Parallel Scavenge GC, 互相激活
-XX:+UseParallelOldGC 老年代使用 Parallel Old GC, 互相激活
-XX:ParallelGCThreads
-XX:MaxGCPauseMillis 设置垃圾收集器最大停顿时间 (即STW的时间), 单位是毫秒。
    为了尽可能地把停顿时间控制在MaxGCPauseMills以内, 收集器在工作时会调整Java堆大小或者其他一些参数。
    对于用户来讲, 停顿时间越短体验越好; 但是服务器端注重高并发, 整体的吞吐量。
```

所以服务器端适合Parallel，进行控制。该参数使用需谨慎。

-XX:GCTimeRatio 垃圾收集时间占总时间的比例 ($1 / (N + 1)$)，用于衡量吞吐量大小
取值范围 (0,100)，默认值99，也就是垃圾回收时间不超过1%。

与前一个-XX: MaxGCPauseMillis参数有一定矛盾性。暂停时间越长，Radio参数就容易超过设定的比例。

-XX:+UseAdaptiveSizePolicy 设置Parallel Scavenge收集器具有自适应调节策略。

在这种模式下，年轻代的大小、Eden和Survivor的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。

在手动调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量 (GCTimeRatio) 和停顿时间 (MaxGCPauseMills)，让虚拟机自己完成调优工作。

CMS回收器

-XX:+UseConcMarkSweepGC 年轻代使用CMS GC。

开启该参数后会自动将-XX: +UseParNewGC打开。即：ParNew (Young区) + CMS (Old区) + Serial Old的组合

-XX:CMSInitiatingOccupanyFraction 设置堆内存使用率的阈值，一旦达到该阈值，便开始进行回收。JDK5及以前版本的默认值为68，DK6及以上版本默认值为92%。

如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低CMS的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。

反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。

因此通过该选项便可以有效降低Full GC的执行次数。

-XX:+UseCMSInitiatingOccupancyOnly 是否动态可调，使CMS一直按CMSInitiatingOccupancyFraction设定的值启动

-XX:+UseCMSCompactAtFullCollection 用于指定在执行完Full GC后对内存空间进行压缩整理

以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间变得更长了。

-XX:CMSFullGCsBeforeCompaction 设置在执行多少次Full GC后对内存空间进行压缩整理。

-XX:ParallelCMSThreads 设置CMS的线程数量。

CMS 默认启动的线程数是 $(ParallelGCThreads + 3) / 4$ ，ParallelGCThreads 是年轻代并行收集器的线程数。

当CPU 资源比较紧张时，受到CMS收集器线程的影响，应用程序的性能在垃圾回收阶段可能会非常糟糕。

-XX:ConcGCThreads 设置并发垃圾收集的线程数，默认该值是基于ParallelGCThreads计算出来的

-XX:+CMSScavengeBeforeRemark 强制hotspot在cms remark阶段之前做一次minor gc，用于提高remark阶段的速度

-XX:+CMSClassUnloadingEnable 如果有的话，启用回收Perm 区 (JDK8之前)

-XX:+CMSParallelInitialEnabled 用于开启CMS initial-mark阶段采用多线程的方式进行标记
用于提高标记速度，在Java8开始已经默认开启

-XX:+CMSParallelRemarkEnabled 用户开启CMS remark阶段采用多线程的方式进行重新标记，默认开启

-XX:+ExplicitGCInvokesConcurrent

-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

这两个参数用户指定hotspot虚拟在执行System.gc()时使用CMS周期

-XX:+CMSPrecleaningEnabled 指定CMS是否需要Pre cleaning阶段

G1回收器

-XX:+UseG1GC 手动指定使用G1收集器执行内存回收任务。

-XX:G1HeapRegionSize 设置每个Region的大小。

值是2的幂，范围是1MB到32MB之间，目标是根据最小的Java堆大小划分出约2048个区域。默认是堆内存的1/2000。

- XX:MaxGCPauseMillis 设置期望达到的最大GC停顿时间指标（JVM会尽力实现，但不保证达到）。默认值是200ms
- XX:ParallelGCThread 设置STW时GC线程数的值。最多设置为8
- XX:ConcGCThreads 设置并发标记的线程数。将n设置为并行垃圾回收线程数（ParallelGCThreads）的1/4左右。
- XX:InitiatingHeapOccupancyPercent 设置触发并发GC周期的Java堆占用率阈值。超过此值，就触发GC。默认值是45。
- XX:G1NewSizePercent 新生代占用整个堆内存的最小百分比（默认5%）
- XX:G1MaxNewSizePercent 新生代占用整个堆内存的最大百分比（默认60%）
- XX:G1ReservePercent=10 保留内存区域，防止 to space（Survivor中的to区）溢出

如何选择垃圾回收器？


- 优先让 JVM 自适应，调整堆的大小
- 串行收集器：内存小于 100M；单核、单机程序，并且没有停顿时间的要求
- 并行收集器：多 CPU、高吞吐量、允许停顿时间超过 1 秒
- 并发收集器：多 CPU、追求低停顿时间、快速响应（比如延迟不能超过 1 秒，如互联网应用）
- 官方推荐 G1，性能高。现在互联网的项目，基本都是使用 G1

特别说明：

- 没有最好的收集器，更没有万能的收集器
- 调优永远是针对特定场景、特定需求，不存在一劳永逸的收集器

4.3.5. GC 日志相关选项

- XX:+PrintGC <==> -verbose:gc 打印简要日志信息
- XX:+PrintGCDetails 打印详细日志信息
- XX:+PrintGCTimeStamps 打印程序启动到GC发生的时间，搭配-XX:+PrintGCDetails使用
- XX:+PrintGCDateStamps 打印GC发生时的时间戳，搭配-XX:+PrintGCDetails使用
- XX:+PrintHeapAtGC 打印GC前后的堆信息，如下图
- Xloggc:<file> 输出GC日志指定路径下的文件中

 image-20210506195156935

- XX:+TraceClassLoading 监控类的加载
- XX:+PrintGCApplicationStoppedTime 打印GC时线程的停顿时间
- XX:+PrintGCApplicationConcurrentTime 打印垃圾收集之前应用未中断的执行时间
- XX:+PrintReferenceGC 打印回收了多少种不同引用类型的引用
- XX:+PrintTenuringDistribution 打印JVM在每次MinorGC后当前使用的Survivor中对象的年龄分布
- XX:+UseGCLogFileRotation 启用GC日志文件的自动转储
- XX:NumberOfGCLogFiles=1 设置GC日志文件的循环数目
- XX:GCLogFileSize=1M 设置GC日志文件的大小

4.3.6. 其他参数

```
-XX:+DisableExplicitGC 禁用hotspot执行System.gc(), 默认禁用
-XX:ReservedCodeCacheSize=<n>[g|m|k]、-XX:InitialCodeCacheSize=<n>[g|m|k] 指定代码缓存的大小
-XX:+UseCodeCacheFlushing 放弃一些被编译的代码, 避免代码缓存被占满时JVM切换到interpreted-only的情况
-XX:+DoEscapeAnalysis 开启逃逸分析
-XX:+UseBiasedLocking 开启偏向锁
-XX:+UseLargePages 开启使用大页面
-XX:+PrintTLAB 打印TLAB的使用情况
-XX:TLABSize 设置TLAB大小
```

4.4. 通过 Java 代码获取 JVM 参数

Java 提供了 `java.lang.management` 包用于监视和管理 Java 虚拟机和 Java 运行时中的其他组件, 它允许本地或远程监控和管理运行的 Java 虚拟机。其中 `ManagementFactory` 类较为常用, 另外 `Runtime` 类可获取内存、CPU 核数等相关的数据。通过使用这些 api, 可以监控应用服务器的堆内存使用情况, 设置一些阈值进行报警等处理。

```
public class MemoryMonitor {
    public static void main(String[] args) {
        MemoryMXBean memorymbean = ManagementFactory.getMemoryMXBean();
        MemoryUsage usage = memorymbean.getHeapMemoryUsage();
        System.out.println("INIT HEAP: " + usage.getInit() / 1024 / 1024 + "m");
        System.out.println("MAX HEAP: " + usage.getMax() / 1024 / 1024 + "m");
        System.out.println("USE HEAP: " + usage.getUsed() / 1024 / 1024 + "m");
        System.out.println("\nFull Information:");
        System.out.println("Heap Memory Usage: " +
memorymbean.getHeapMemoryUsage());
        System.out.println("Non-Heap Memory Usage: " +
memorymbean.getNonHeapMemoryUsage());

        System.out.println("=====通过java来获取相关系统状态
===== ");
        System.out.println("当前堆内存大小totalMemory " + (int)
Runtime.getRuntime().totalMemory() / 1024 / 1024 + "m");// 当前堆内存大小
        System.out.println("空闲堆内存大小freeMemory " + (int)
Runtime.getRuntime().freeMemory() / 1024 / 1024 + "m");// 空闲堆内存大小
        System.out.println("最大可用总堆内存maxMemory " +
Runtime.getRuntime().maxMemory() / 1024 / 1024 + "m");// 最大可用总堆内存大小

    }
}
```