Hotspot JVM的常用选项

参考：<https://www.zybuluo.com/jewes/note/57352>

Hotspot JVM的常用选项

本文将介绍Hotspot JVM的常用选项。

选项的分类

Hotspot JVM提供以下三大类选项：   
1. 标准选项：这类选项的功能是很稳定的，在后续版本中也不太会发生变化。运行java或者java -help可以看到所有的标准选项。所有的标准选项都是以-开头，比如-version， -server等。   
2. X选项：比如-Xms。这类选项都是以-X开头，可能由于这个原因它们被称为X选项。运行java -X命令可以看到所有的X选项。这类选项的功能还是很稳定，但官方的说法是它们的行为可能会在后续版本中改变，也有可能不在后续版本中提供了。   
3. XX选项：这类选项是属于实验性，主要是给JVM开发者用于开发和调试JVM的，在后续的版本中行为有可能会变化。

XX选项的语法

* 如果是布尔类型的选项，它的格式为-XX:+flag或者-XX:-flag，分别表示开启和关闭该选项。
* 针对非布尔类型的选项，它的格式为-XX:flag=value

在了解这些约定的规范后，我们就可以来看看一些比较常用的选项了。

指定JVM的类型：-server，-client

Hotspot JVM有两种类型，分别是server和client。它们的区别是Server VM的初始堆空间会大一些，默认使用的是并行垃圾回收器。Client VM相对来讲会保守一些，初始堆空间会小一些，使用串行的垃圾回收器，它的目标是为了让JVM的启动速度更快。

JVM在启动的时候会根据硬件和操作系统会自动选择使用Server还是Client类型的JVM。   
- 在32位Windows系统上，不论硬件配置如何，都默认使用Client类型的JVM。   
- 在其他32位操作系统上，如果机器配置有2GB集群以上的内存同时有2个以上的CPU，则默认会使用Server类型的JVM   
- 64位机器上只有Server类型的JVM。也就是说Client类型的JVM只在32位机器上提供。   
- 你也可以使用-server和-client选项来指定JVM的类型，不过只在32位的机器上有效，原因见上面一条。   
详细内容请参见：<http://docs.oracle.com/javase/7/docs/technotes/guides/vm/server-class.html>

指定JIT编译器的模式：-Xint，-Xcomp，-Xmixed

我们知道Java是一种解释型语言，但是随着JIT技术的进步，它能在运行时将Java的字节码编译成本地代码。以下是几个相关的选项：

* -Xint表示禁用JIT，所有字节码都被解释执行，这个模式的速度最慢的。
* -Xcomp表示所有字节码都首先被编译成本地代码，然后再执行。
* -Xmixed，默认模式，让JIT根据程序运行的情况，有选择地将某些代码编译成本地代码。   
  -Xcomp和-Xmixed到底谁的速度快，针对不同的程序可能有不同的结果（me:但是我可以都试试看，看看哪个更优然后选用），基本还是推荐用默认模式。

-version和-showversion

-version就是查看当前机器的java是什么版本，是什么类型的JVM（Server/Client），采用的是什么执行模式。比如，在我的机器上的结果如下：

$ java -version

java version "1.7.0\_71"

Java(TM) SE Runtime Environment (build 1.7.0\_71-b14)

Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)

表示我机器上java是运行在mixed模式下的Server VM。

-showversion的作用是在运行一个程序的时候首先把JVM的版本信息打印出来，这样便于问题诊断。个人建议Server类型的程序都把这个选项打开，这样可以发现一些配置问题，比如程序需要JDK1.7才能运行，而有的机器上装有多个JDK的版本，打开这个选项可以避免使用了错误版本的Java。

查看XX选项的值： -XX:+PrintCommandLineFlags， -XX:+PrintFlagsInitial和-XX:+PrintFlagsFinal

与-showversion类似，-XX:+PrintCommandLineFlags可以让在程序运行前打印出用户手动设置或者JVM自动设置的XX选项，建议加上这个选项以辅助问题诊断。比如在我的机器上，JVM自动给配置了初始的和最大的HeapSize以及其他的一些选项：

$ java -XX:+PrintCommandLineFlags -version

-XX:InitialHeapSize=134217728 -XX:MaxHeapSize=2147483648 -XX:+PrintCommandLineFlags -XX:+UseCompressedOops -XX:+UseParallelGC

java version "1.7.0\_71"

Java(TM) SE Runtime Environment (build 1.7.0\_71-b14)

Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)

相关另外两个选项：-XX:+PrintFlagsInitial表示打印出所有XX选项的默认值，-XX:+PrintFlagsFinal表示打印出XX选项在运行程序时生效的值。

内存大小相关的选项

* -Xms 设置初始堆的大小，也是最小堆的大小，它等价于：-XX:InitialHeapSize
* -Xmx 设置最大堆的大小，它等价于-XX:MaxHeapSize。   
  比如，下面这条命令就是设置堆的初始值为128m，最大值为2g。

java -Xms128m -Xmx2g MyApp

如果堆的初始值和最大值不一样的话，JVM会根据程序的运行情况，自动调整堆的大小，这可能会影响到一些效率。针对服务端程序，一般是把堆的最小值和最大值设置为一样来避免堆扩展和收缩对性能的影响。

* -XX:PermSize 用来设置永久区的初始大小
* -XX:MaxPermSize 用来设置永久区的最大值

永久区是存放类以及常量池的地方，如果程序需要加载的class数量非常多的话，就需要增大永久区的大小。

* -Xss 设置线程栈的大小，线程栈的大小会影响到递归调用的深度，同时也会影响到能同时开启的线程数量。

OutofMemory（OOM）相关的选项

如果程序发生了OOM后，JVM可以配置一些选项来做些善后工作，比如把内存给dump下来，或者自动采取一些别的动作。

* -XX:+HeapDumpOnOutOfMemoryError 表示在内存出现OOM的时候，把Heap转存(Dump)到文件以便后续分析，文件名通常是java\_pid<pid>.hprof，其中pid为该程序的进程号。
* -XX:HeapDumpPath=<path>: 用来指定heap转存文件的存储路径，需要指定的路径下有足够的空间来保存转存文件。
* -XX:OnOutOfMemoryError 用来指定一个可行性程序或者脚本的路径，当发生OOM的时候，去执行这个脚本。

比如，下面的命令可以使得在发生OOM的时候，Heap被转存到文件/tmp/heapdump.hprof，同时执行Home目录中的cleanup.sh文件。

$ java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof -XX:OnOutOfMemoryError ="sh ~/cleanup.sh" MyApp

个人觉得几个选项还是非常有用的，它可以使得你有相关的信息来分析OOM的根源。

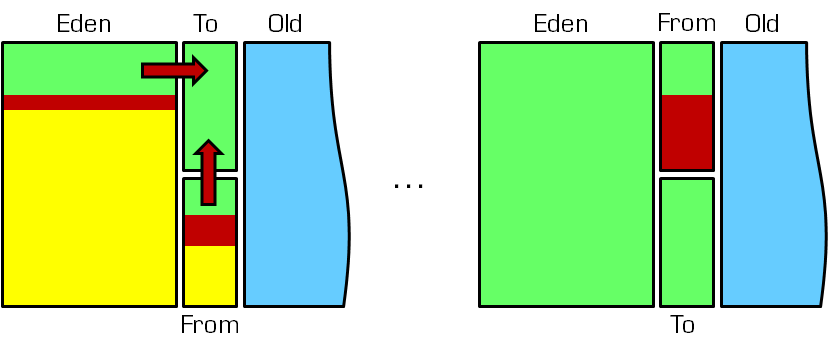
新生代相关的选项

在介绍新生代相关的选项前，先简要介绍下Hotspot VM的Heap分代的背景知识。很多面向对象程序在运行时都具有如下两点特征：   
1. 新创建的对象通常不会存活很长时间，也就是夭折了。   
2. 很少有老对象引用到新对象。

基于这里两点，把新老对象分别放在不同的区域（分别叫做新生代和老生代）可以针对新老对象的特点使用不同的回收算法，同时在回收新对象的时候不用遍历老对象，从而提高垃圾回收的效率。

在Hotspot JVM中，它进一步地将新生代分成了三个区域，一个稍大的区域Eden和两个较小但大小相等的Survivor区域（分别叫做From和To）。一般来讲，新对象首先分配在Eden区域，当Eden区域满的时候，会执行一次Minor GC。MinorGC使用的是标记-拷贝算法（me：也就是标记-复制算法）。垃圾回收器会首先标记Eden和From区域中还存活的对象，然后把它们全部移动到To区域，这样Eden和From区域的空间就可以全部回收了，最后再将指向From和To区域的指针交换一下。

下图展示了MinorGC的流程，绿色区域表示空闲空间，红色表示活动对象，黄色表示可以回收的对象。



简要总结一下，对象在新生代的生命周期是，它首先在Eden区域诞生，如果对象在MinorGC时还存活的话，就移动到Survivor区域。在后续的MinorGC的时候，如果对象还继续存活的话，就在两个Survivor区域将倒腾。那对象什么时候会被移动到老生代呢？有以下条件：   
1. Survivor区域中存活对象占用Survivor空间达到了指定的阈值。   
2. 对象在Survivor空间每倒腾一次其年龄就加1，如果一个对象的年龄达到了一个阈值，也会被移动到老生代。   
3. 大对象会在创建的时候就会被直接放到老生代。

由此可见，新生代的空间大小很重要：如果新生代空间过小，就会导致对象很快就被移动到老生代，从而使得某些原本可以及时回收的对象存活的时间过长，而且老生代回收的代价更大。那相反，如果新生代空间过大，就会使得某些存活时间长的对象在新生代倒腾了很多次，影响到新生代回收垃圾的效率。这就需要根据应用的特点，找到一个合适的值。Hotspot提供了如下一些选项来调节新生代的参数：

* -XX:NewSize和-XX:MaxNewSize分别用来设置新生代的最小和最大值。需要注意的是，新生代是JVM堆的一部分，新生代的空间大小不能大于老生代的大小，因为在极端的情况下，新生代中对象可能会被全部移到老生代，因此-XX:MaxNewSize最大只能设为-Xmx的一半。
* -XX:NewRatio用来设置老生代和新生代大小的比例，比如-XX:NewRatio=2表示1/3的Heap是新生代，2/3的Heap是老生代。使用这个选项的好处是新生代的大小也能随着Heap的变化而变化。
* -XX:SurvivorRatio用来设置新生代中Eden和Survivor空间大小的比例，需要注意的是有两个Survivor。比如-XX:SurvivorRatio=8表示Eden区域在新生代的8/10，两个Survivor分别占1/10（me：jvm的默认比例也是8:1:1）。调节Survivor空间的时候也注意要折中，如果Survivor空间小的话，那么很有可能在一次MinorGC的时候Survivor空间就满了，从而对象就被移到了老生代；如果Survivor空间大的话，那么Eden区域就小了，从而导致MinorGC的发生得更频繁。

总得来说，调节新生代的目标是：

1）避免对象过早地被移到了老生代

2）也要避免需要长期存活的对象在新生代呆的时间过长，这会提高MinorGC发生的频率以及增加单次MinorGC的时间。这需要针对程序的运行情况做一些分析。接下来就介绍了一个参数来分析新生代对象年龄的分布。

-XX:+PrintTenuringDistribution

-XX:+PrintTenuringDistribution让JVM在每次MinorGC后打印出Survivor空间中的对象的年龄分布。比如：

Desired survivor size 75497472 bytes, new threshold 15 (max 15)

- age 1: 19321624 bytes, 19321624 total

- age 2: 79376 bytes, 19401000 total

- age 3: 2904256 bytes, 22305256 total

从第一行中可以看出JVM期望的Survivor空间占用为72M，对象被移到老年代中的年龄阈值为15。其中期望的Survivor空间大小为Survivor空间大小 x -XX:TargetSurvivorRatio的值。

接下来的一行，表示年龄为1的对象约19M，年龄为2的对象约79k，年龄为3的对象约为2.9M，每行后面的数值表示所有小于等于该行年龄的对象的总共大小，比如最后一行就表示所有年龄小于等于3的对象的总共大小为约22M（等于所有年龄对象大小的和）。因为目前Survivor空间中对象的大小22M小于期望Survivor空间的大小72M，所以没有对象会被移到老年代。

假设下一次MinorGC后的输出结果为：

Desired survivor size 75497472 bytes, new threshold 2 (max 15)

- age 1: 68407384 bytes, 68407384 total

- age 2: 12494576 bytes, 80901960 total

- age 3: 79376 bytes, 80981336 total

- age 4: 2904256 bytes, 83885592 total

上次MinorGC后还存活的对象在这次MinorGC年龄都增加了1，可以看到上次年龄为2和3的对象（对应在这次GC后的年龄为3和4）依然存在（大小未变），而一部分上次对象年龄为1的对象在这次GC时被回收了。同时可以看到这次新增了约68M的新对象。这次MinorGC后Survivor区域中对象总的大小为约83M，大于了期望的Survivor空间的大小72M，因此它就把对象移到老年代的年龄的阈值调整为2，在下次MinorGC时一部分对象就会被移到老年代了。

相关的调整选项有：   
- -XX:InitialTenuringThreshold 表示对象被移到老年代的年龄阈值的初始值   
- -XX:MaxTenuringThreshold 表示对象被移到老年代的年龄阈值的最大值   
- -XX:TargetSurvivorRatio 表示MinorGC结束了Survivor区域中占用空间的期望比例。

这些参数的调节没有统一的标准，但是有两点可以借鉴：   
1. 如果Survivor中对象的年龄分布显示很多对象在经历了多次GC最终年龄达到了-XX:MaxTenuringThreshold才被移到老年代，这可能说明-XX:MaxTenuringThreshold设置得过大，也有可能是Survivor的空间过大。   
2. 如果-XX:MaxTenuringThreshold的值大于1，但是很多对象年龄都不大于1，那就得关注一下期望的Survivor空间。如果每次GC后Survivor中对象的大小都没有超过期望的Survivor空间大小，则说明GC工作得很好。反之，则说明可能Survivor空间小了，使得新生成的对象很快就被移到了老年代了。

吞吐量优先收集器的相关选项

衡量JVM垃圾收集器的两个基本指标是吞吐量和停顿时间。吞吐量是指执行用户代码的时间占总的时间的比例，总的时间包括执行用户代码的时间和垃圾回收占用的时间。在垃圾回收的时候执行用户代码的线程必须暂停，这会导致程序暂时失去响应。停顿时间就是衡量垃圾回收时造成的用户线程暂停的时间。这两个指标是在一定程度是相互矛盾的，不可能让一个程序的吞吐量很高的同时停顿时间也短，只能以优先选择一个目标或者折中一下。因此，不同的垃圾回收器会有不同的侧重点。

在Hotspot JVM中，侧重于吞吐量的垃圾回收器是Parallel Scavenge，它的相关选项如下：

* -XX:+UseParallelOldGC 表示新生代和老生代都使用并行回收器，其中的Old表示老生代的意思，而不是旧的意思。
* -XX:ParallelGCThreads=n 表示配置多少个线程来回收垃圾。默认的配置是如果处理器的个数小于8，那么就是处理器的个数；如果处理器大于8，它的值就是3+5N/8。也可以根据程序的需要去设置这个值，比如你的机器有16核，上面有4个Java程序，那么设置将这个值设置为4比较合理，因为JVM不会去探测同一机器上有多少个Java程序。
* -XX:UseAdaptiveSizePolicy 表示是否开启自适应策略，打开这个开关后，JVM自动调节JVM的新生代大小，Eden和Survivor的比例等参数。用户只需要设置期望的吞吐量(-XX:GCTimeRatio)和期望的停顿时间(-XX:MaxGCPauseMillis)。然后，JVM会尽量去向用户期望的方向去优化。

此外，如果机器只有一个核的话，采用并行回收器可能得不偿失，因为多个回收线程会争抢CPU资源，反而造成更大的消耗。这时，就最好采用串行回收器，相关的参数是-XX:+UseSerialGC

CMS收集器

CMS收集器(ConcurrentMarkandSweep)，是一个关注系统停顿时间的收集器。它的主要思想是把收集器分成了不同的阶段，其中某些阶段是可以用户程序并行的，从而减少了整体的系统停顿时间。它主要分成了以下几个阶段：   
- 初始标记 initial mark   
- 并发标记 concurrent mark   
- 重新标记 remark   
- 并发清理 concurrent clean   
- 并发重置 concurrent reset

凡是名字以并发开头的阶段都是可以和用户线程并行的，其他阶段也是要暂停用户程序线程。   
CMS虽然能减少系统的停顿时间，但是它也有其缺点：   
1. 从它的名字可以看出，它是一个标记-清除收集器，也就说运行了一段时间后，内存会产生碎片，从而导致无法找到连续空间来分配大对象。   
2. CMS收集器在运行过程中会占用一些内存，同时系统还在运行，如果系统产生新对象的速度比CMS清理的速度快的话，会导致CMS运行失败。

当上面的任何一种情况发生的时候，JVM就会触发一次Full GC，会导致JVM停顿较长时间。

它的相关选项如下：   
- -XX:+UseConcMarkSweepGC 表示老年代开启CMS收集器，而新生代默认会使用并行收集器。   
- -XX:ConcGCThreads 指定用多少个线程来执行CMS的并发阶段。   
- -XX:CMSInitiatingOccupancyFraction 指定在老生代用掉多少内存后开始进行垃圾回收。与吞吐量优先的回收器不同的是，吞吐量优先的回收器在老生代内存用尽了以后才开始进行收集，这对CMS来讲是不行的，因为吞吐量优先的垃圾回收器运行的时候会停止所有用户线程，所以不会产生新的对象，而CMS运行的时候，用户线程还有可能产生新的对象，所以不能等到内存用光后才开始运行。比如-XX:CMSInitiatingOccupancyFraction=75表示老生代用掉75%后开始回收垃圾。默认值是68。   
- -XX:+ExplicitGCInvokesConcurrent 如果在代码里面显式调用System.gc()，那么它还是会执行Full GC从而导致用户线程被暂停。采用这个选项使得显式触发GC的时候还是使用CMS收集器。   
- -XX:+DisableExplicitGC 一个相关的选项，这个选项是禁止显式调用GC

GC日志相关的选项

分析GC问题不可避免地要查看GC日志，下面是一些GC日志相关的选项：

* -XX:+PrintGC，等同于-verbose:gc 表示打开简化的GC日志，相关输出如下：

[GC 425355K->351685K(506816K), 0.2175300 secs]   
[Full GC 500561K->456058K(506816K), 0.6421920 secs]

其中以GC开头的行表示发生了一次Minor GC，后面的数字表示收集前后Heap空间的占用量，圆括号里面表示Heap大小，最后的数字表示用了多少时间。比如：上面的例子中，表示在这次GC新生代空间占用从425355K降到了351685K，总的新生代空间为506816K，这次GC耗时0.22秒。   
通过这个选项只能看到一些基本信息，而且所有收集器的输出在这个模式下都是一样的。

* -XX:+PrintGCDetails 这个选项会打印出更多的GC日志，不同的收集器产生的日志会不一样。因此，在后续的文章中再介绍不同收集器的日志格式。
* -XX:+PrintGCTimeStamps and -XX:+PrintGCDateStamps 这两个选项把GC的时间戳显示在GC的日志中。其中，-XX:+PrintGCTimeStamps打印GC发生的时间相对于JVM启动的时间，-XX:+PrintGCDateStamps表示打印出GC发生的具体时间。

比如，以下是-XX:+PrintGCTimeStamps的输出

0,185: [GC 66048K->53077K(251392K), 0,0977580 secs]

0,323: [GC 119125K->114661K(317440K), 0,1448850 secs]

0,603: [GC 246757K->243133K(375296K), 0,2860800 secs]

以下是两个都打开后的输出

2014-12-26T17:52:38.613-0800: 3.395: [GC 139776K->58339K(506816K), 0.1442900 secs]

* -Xloggc:<file> 表示把GC日志写入到一个文件中去，而不是打印到标准输出中。

需要注意的是：这些和GC日志相关的选项可以在JVM已经启动后再开启，可以通过jinfo这个工具去设置。具体可以参见jinfo的帮助文件。这样就可以在需要诊断问题的时候再开启GC日志。