https://blog.csdn.net/xiajian2010/article/details/17376157

**java jvm 参数 -Xms -Xmx -Xmn -Xss 调优总结**

2013年12月17日 15:17:13 [xiajian2010](https://me.csdn.net/xiajian2010) 阅读数 19639

**缘起**

 今天在对XML-security运行randoop时，出现了一个错误：在编译生成的测试用例时，java.lang.OutOfMemoryError: Java heap space。感觉很新奇，就上网查了一下。涉及点有这样的几点： javac, java.lang.OutOfMemoryError: Java heap space,jvm堆栈的调优。发现了一些有用文章，转载了下。

注：该文章中写成的时间是2006，彼时，金融危机还没有到来，sun还未被收购，jdk还是1.5-1.6之类的；现在，JDK已经更新到1.8(据Oracle官方说，明年发布)，Hotspot垃圾回收的算法也更新为G!算法。但是参数和命令的设置还是不变的，文章中的命令还是可以用。

**堆大小设置**

JVM 中最大堆大小有三方面限制: 1.相关操作系统的数据模型(32-bt还是64-bit)限制; 2.系统的可用虚拟内存限制; 3.系统的可用物理内存限制. 在32位系统下,一般限制在1.5G~2G;64为操作系统对内存无特定限制.  在Windows Server 2003 系统,3.5G物理内存,JDK5.0下测试,最大可设置为1478m.

典型设置:   
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k  
   -Xmx3550m:设置JVM最大可用内存为3550M.  
   -Xms3550m:设置JVM促使内存为3550m.此值可以设置与-Xmx相同,以避免每次垃圾回收完成后JVM重新分配内存.  
   -Xmn2g:设置年轻代大小为2G.

整个堆大小=年轻代大小 + 年老代大小 + 持久代大小.持久代一般固定大小为64m,所以增大年轻代后,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.(注：这里默认的使用的JVM是Sun的Hotspot，而其中使用GC算法就是分代算法。若要了解详情，可以参考一些关于JVM的书： 《[深入理解Java虚拟机](http://book.douban.com/subject/24722612/)》第二版 )  
-Xss128k:设置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.

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0  
   -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=4:设置年轻代中Eden区与Survivor区的大小比值.设置为4,则两个Survivor区与一个Eden区的比值为2:4,一个Survivor区占整个年轻代的1/6  
   -XX:MaxPermSize=16m:设置持久代大小为16m.  
   -XX:MaxTenuringThreshold=0: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.

**回收器选择**

JVM给了三种选择:串行收集器,并行收集器,并发收集器,但是串行收集器只适用于小数据量的情况,所以这里的选择主要针对并行收集器和并发收集器.默认 情况下,JDK5.0以前都是使用串行收集器,如果想使用其他收集器需要在启动时加入相应参数.JDK5.0以后,JVM会根据当前系统配置进行判断.   
吞吐量优先的并行收集器  
如上文所述,并行收集器主要以到达一定的吞吐量为目标,适用于科学技术和后台处理等.  
典型配置:   
java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20  
-XX:+UseParallelGC: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.  
-XX:ParallelGCThreads=20: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.

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC  
-XX:+UseParallelOldGC:配置年老代垃圾收集方式为并行收集.JDK6.0支持对年老代并行收集.

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100  
-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间,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.

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy  
-XX:+UseAdaptiveSizePolicy: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.

响应时间优先的并发收集器  
如上文所述,并发收集器主要是保证系统的响应时间,减少垃圾收集时的停顿时间.适用于应用服务器,电信领域等.  
典型配置:   
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC:设置年老代为并发收集.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:+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.   
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection  
-XX:CMSFullGCsBeforeCompaction: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:+UseCMSCompactAtFullCollection:打开对年老代的压缩.可能会影响性能,但是可以消除碎片

辅助信息  
JVM提供了大量命令行参数,打印信息,供调试使用.主要有以下一些:   
-XX:+PrintGC  
输出形式:[GC 118250K->113543K(130112K), 0.0094143 secs]   
[Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails  
输出形式:[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]   
[GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

-XX:+PrintGCTimeStamps -XX:+PrintGC:PrintGCTimeStamps可与上面两个混合使用  
输出形式:11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

-XX:+PrintGCApplicationConcurrentTime:打印每次垃圾回收前,程序未中断的执行时间.可与上面混合使用  
输出形式:Application time: 0.5291524 seconds

-XX:+PrintGCApplicationStoppedTime:打印垃圾回收期间程序暂停的时间.可与上面混合使用  
输出形式:Total time for which application threads were stopped: 0.0468229 seconds

-XX:PrintHeapAtGC:打印GC前后的详细堆栈信息  
输出形式:  
34.702: [GC {Heap before gc invocations=7:  
def new generation total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000)  
eden space 49152K, 99% used [0x1ebd0000, 0x21bce430, 0x21bd0000)  
from space 6144K, 55% used [0x221d0000, 0x22527e10, 0x227d0000)  
to space 6144K, 0% used [0x21bd0000, 0x21bd0000, 0x221d0000)  
tenured generation total 69632K, used 2696K [0x227d0000, 0x26bd0000, 0x26bd0000)  
the space 69632K, 3% used [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)  
compacting perm gen total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
the space 8192K, 35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
ro space 8192K, 66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
rw space 12288K, 46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs] 55264K->6615K(124928K)Heap after gc invocations=8:  
def new generation total 55296K, used 3433K [0x1ebd0000, 0x227d0000, 0x227d0000)  
eden space 49152K, 0% used [0x1ebd0000, 0x1ebd0000, 0x21bd0000)  
from space 6144K, 55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)  
to space 6144K, 0% used [0x221d0000, 0x221d0000, 0x227d0000)  
tenured generation total 69632K, used 3182K [0x227d0000, 0x26bd0000, 0x26bd0000)  
the space 69632K, 4% used [0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)  
compacting perm gen total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
the space 8192K, 35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
ro space 8192K, 66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
rw space 12288K, 46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
}, 0.0757599 secs]

-Xloggc:filename:与上面几个配合使用,把相关日志信息记录到文件以便分析.

**常见配置汇总**

**堆设置**

-Xms:初始堆大小   
-Xmx:最大堆大小   
-XX:NewSize=n:设置年轻代大小   
-XX:NewRatio=n:设置年轻代和年老代的比值.如:为3,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: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:MaxPermSize=n:设置持久代大小

**收集器设置**

-XX:+UseSerialGC:设置串行收集器   
-XX:+UseParallelGC:设置并行收集器   
-XX:+UseParalledlOldGC:设置并行年老代收集器   
-XX:+UseConcMarkSweepGC:设置并发收集器

**垃圾回收统计信息**

-XX:+PrintGC   
-XX:+PrintGCDetails   
-XX:+PrintGCTimeStamps   
-Xloggc:filename

**并行收集器设置**

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数.并行收集线程数.   
-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间   
-XX:GCTimeRatio=n: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)

**并发收集器设置**

-XX:+CMSIncrementalMode:设置为增量模式.适用于单CPU情况.   
-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时,使用的CPU数.并行收集线程数.

**调优总结**

**年轻代大小选择**

响应时间优先的应用:尽可能设大,直到接近系统的最低响应时间限制(根据实际情况选择).在此种情况下,年轻代收集发生的频率也是最小的.同时,减少到达年老代的对象.   
吞吐量优先的应用:尽可能的设置大,可能到达Gbit的程度.因为对响应时间没有要求,垃圾收集可以并行进行,一般适合8CPU以上的应用.

**年老代大小选择**

响应时间优先的应用:年老代使用并发收集器,所以其大小需要小心设置,一般要考虑并发会话率和会话持续时间等一些参数.如果堆设置小了,可以会造成内存碎 片,高回收频率以及应用暂停而使用传统的标记清除方式;如果堆大了,则需要较长的收集时间.最优化的方案,一般需要参考以下数据获得:   
1.并发垃圾收集信息   
2.持久代并发收集次数   
3.传统GC信息   
4.花在年轻代和年老代回收上的时间比例  
5.减少年轻代和年老代花费的时间,一般会提高应用的效率

吞吐量优先的应用:一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代.原因是,这样可以尽可能回收掉大部分短期对象,减少中期的对象,而年老代尽存放长期存活对象.  
较小堆引起的碎片问题  
因为年老代的并发收集器使用标记,清除算法,所以不会对堆进行压缩.当收集器回收时,他会把相邻的空间进行合并,这样可以分配给较大的对象.但是,当堆空 间较小时,运行一段时间以后,就会出现"碎片",如果并发收集器找不到足够的空间,那么并发收集器将会停止,然后使用传统的标记,清除方式进行回收.如果 出现"碎片",可能需要进行如下配置:   
-XX:+UseCMSCompactAtFullCollection:使用并发收集器时,开启对年老代的压缩.   
-XX:CMSFullGCsBeforeCompaction=0:上面配置开启的情况下,这里设置多少次Full GC后,对年老代进行压缩

在同一个工程下,有两个类,这两个类中只有很少的变动,而最关健的FOR却没有一点变动,可是当我分别运行这两个程序的时候却出现一个很严重的问题,一个程序循环的快,一个循环的慢.这到底是怎么回事呢~???苦苦寻找了半天也没有想到是为什么,因为程序改变的部分根不影响我循环的速度,可是结果却是有很大的差别,一个大约是在一分钟这内就可以循环完,可是另一个却需要六七分钟,这根本就不是一个数据理级的麻.两个完全一样的循环,从代码上根本上是看不出有什么问题.不得以求助同事吧,可是同事看了也感觉很诡异,两个人在那订着代码又看了一个多小时,最后同事让我来个干净点的,关机重启.我到也听话,就顺着同事的意思去了,可就在关机的这个时候他突然说是不是内存的问题,我也空然想到了,还真的有可能是内存的问题,因为快的那个在我之前运行程序之前可给过1G的内存啊,而后来的这个我好像是没有设过内存啊,机器起来了,有了这个想法进去看看吧,结果正中要害,果真是慢的那个没有开内存,程序运行时只不过是JVM默认开的内存.我初步分析是因为内存太小,而我的程序所用内存又正好卡在JVM所开内存边上,不至于溢出.当程序运行时就得花费大部分时间去调用GC去,这样就导致了为什么相同的循环出现两种不同的效率~!  
顺便把内存使用情况的方法也贴出来:

1. public static String getMemUsage() {
2. long free = java.lang.Runtime.getRuntime().freeMemory();
3. long total = java.lang.Runtime.getRuntime().totalMemory();
4. StringBuffer buf = new StringBuffer();
5. buf.append("[Mem: used ").append((total-free)>>20)
6. .append("M free ").append(free>>20)
7. .append("M total ").append(total>>20).append("M]");
8. return buf.toString();
9. }

google一下,大概就说JVM是这样来操作内存:

堆(Heap)和非堆(Non-heap)内存  
按照官方的说法:"Java 虚拟机具有一个堆,堆是运行时数据区域,所有类实例和数组的内存均从此处分配.堆是在 Java 虚拟机启动时创建的.""在JVM中堆之外的内存称为非堆内存(Non-heap memory)".可以看出JVM主要管理两种类型的内存:堆和非堆.简单来说堆就是Java代码可及的内存,是留给开发人员使用的;非堆就是JVM留给自己用的,所以方法区,JVM内部处理或优化所需的内存(如JIT编译后的代码缓存),每个类结构(如运行时常数池,字段和方法数据)以及方法和构造方法的代码都在非堆内存中.

**堆内存分配**

JVM初始分配的内存由-Xms指定,默认是物理内存的1/64;JVM最大分配的内存由-Xmx指定,默认是物理内存的1/4.默认空余堆内存小于40%时,JVM就会增大堆直到-Xmx的最大限制;空余堆内存大于70%时, JVM会减少堆直到-Xms的最小限制.因此服务器一般设置-Xms,-Xmx相等以避免在每次GC 后调整堆的大小.

**非堆内存分配**

JVM使用-XX:PermSize设置非堆内存初始值,默认是物理内存的1/64;由XX:MaxPermSize设置最大非堆内存的大小,默认是物理内存的1/4.  
JVM内存限制(最大值)  
首先JVM内存首先受限于实际的最大物理内存,假设物理内存无限大的话,JVM内存的最大值跟操作系统有很大的关系.简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制,这个限制一般是 2GB-3GB(一般来说Windows系统下为1.5G-2G,Linux系统下为2G-3G),而64bit以上的处理器就不会有限制了  
JVM内存的调优  
1. Heap设定与垃圾回收Java Heap分为3个区,Young,Old和Permanent.Young保存刚实例化的对象.当该区被填满时,GC会将对象移到Old区.Permanent区则负责保存反射对象,本文不讨论该区.JVM的Heap分配可以使用-X参数设定,  
-Xms  
初始Heap大小  
-Xmx  
java heap最大值  
-Xmn  
young generation的heap大小  
JVM有2个GC线程.第一个线程负责回收Heap的Young区.第二个线程在Heap不足时,遍历Heap,将Young 区升级为Older区.Older区的大小等于-Xmx减去-Xmn,不能将-Xms的值设的过大,因为第二个线程被迫运行会降低JVM的性能.  
为什么一些程序频繁发生GC?有如下原因:

* 程序内调用了System.gc()或Runtime.gc().
* 一些中间件软件调用自己的GC方法,此时需要设置参数禁止这些GC.
* Java的Heap太小,一般默认的Heap值都很小.
* 频繁实例化对象,Release对象.此时尽量保存并重用对象,例如使用StringBuffer()和String().

如果你发现每次GC后,Heap的剩余空间会是总空间的50%,这表示你的Heap处于健康状态.许多Server端的Java程序每次GC后最好能有65%的剩余空间.经验之谈:  
1．Server端JVM最好将-Xms和-Xmx设为相同值.为了优化GC,最好让-Xmn值约等于-Xmx的1/3[2].  
2．一个GUI程序最好是每10到20秒间运行一次GC,每次在半秒之内完成[2].  
注意:  
1．增加Heap的大小虽然会降低GC的频率,但也增加了每次GC的时间.并且GC运行时,所有的用户线程将暂停,也就是GC期间,Java应用程序不做任何工作.  
2．Heap大小并不决定进程的内存使用量.进程的内存使用量要大于-Xmx定义的值,因为Java为其他任务分配内存,例如每个线程的Stack等.  
2．Stack的设定  
每个线程都有他自己的Stack.  
-Xss  
每个线程的Stack大小  
Stack的大小限制着线程的数量.如果Stack过大就好导致内存溢漏.-Xss参数决定Stack大小,例如-Xss1024K.如果Stack太小,也会导致Stack溢漏.  
3．硬件环境  
硬件环境也影响GC的效率,例如机器的种类,内存,swap空间,和CPU的数量.  
如果你的程序需要频繁创建很多transient对象,会导致JVM频繁GC.这种情况你可以增加机器的内存,来减少Swap空间的使用[2].  
4．4种GC  
第一种为单线程GC,也是默认的GC.,该GC适用于单CPU机器.  
第二种为Throughput GC,是多线程的GC,适用于多CPU,使用大量线程的程序.第二种GC与第一种GC相似,不同在于GC在收集Young区是多线程的,但在Old区和第一种一样,仍然采用单线程.-XX:+UseParallelGC参数启动该GC.  
第三种为Concurrent Low Pause GC,类似于第一种,适用于多CPU,并要求缩短因GC造成程序停滞的时间.这种GC可以在Old区的回收同时,运行应用程序.-XX:+UseConcMarkSweepGC参数启动该GC.  
第四种为Incremental Low Pause GC,适用于要求缩短因GC造成程序停滞的时间.这种GC可以在Young区回收的同时,回收一部分Old区对象.-Xincgc参数启动该GC.  
4种GC的具体描述参见[3].  
参考文章:  
1. JVM Tuning. <http://www.caucho.com/resin-3.0/performance/jvm-tuning.xtp#garbage-collection>  
2. Performance tuning Java: Tuning steps  
<http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,1604,00.html>  
3. Tuning Garbage Collection with the 1.4.2 JavaTM Virtual Machine .  
<http://java.sun.com/docs/hotspot/gc1.4.2/>