【转】JVM（Java虚拟机）优化大全和案例实战

原文地址：http://blog.csdn.net/kthq/article/details/8618052

堆内存设置

原理

JVM堆内存分为2块：Permanent Space 和 Heap Space。

- Permanent 即 持久代（Permanent Generation），主要存放的是Java类定义信息，与垃圾收集器要收集的Java对象关系不大。
- Heap = { Old + NEW = {Eden, from, to} }，Old 即 年老代（Old Generation），New 即 年轻代（Young Generation）。年老代和年轻代的划分对垃圾收集影响比较大。

年轻代

所有新生成的对象首先都是放在年轻代。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代一般分3个区，1个Eden区，2个Survivor区（from 和 to）。

大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当另一个Survivor区也满了的时候，从前一个Survivor区复制过来的并且此时还存活的对象，将可能被复制到年老代。

2个Survivor区是对称的，没有先后关系，所以同一个Survivor区中可能同时存在从Eden区复制过来对象，和从另一个Survivor区复制过来的对象；而复制到年老区的只有从另一个Survivor区过来的对象。**而且，因为需要交换的原因，Survivor区至少有一个是空的。**特殊的情况下，根据程序需要，Survivor区是可以配置为多个的（多于2个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

针对年轻代的垃圾回收即 Young GC。

年老代

在年轻代中经历了N次（可配置）垃圾回收后仍然存活的对象，就会被复制到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

针对年老代的垃圾回收即 Full GC。

持久代

用于存放静态类型数据，如 Java Class, Method 等。持久代对垃圾回收没有显著影响。但是有些应用可能动态生成或调用一些Class，例如 Hibernate CGLib 等，在这种时候往往需要设置一个比较大的持久代空间来存放这些运行过程中动态增加的类型。

所以，当一组对象生成时，**内存申请过程**如下：

1. JVM会试图为相关Java对象在年轻代的Eden区中初始化一块内存区域。
2. 当Eden区空间足够时，内存申请结束。否则执行下一步。
3. JVM试图释放在Eden区中所有不活跃的对象（Young GC）。释放后若Eden空间仍然不足以放入新对象，JVM则试图将部分Eden区中活跃对象放入Survivor区。
4. Survivor区被用来作为Eden区及年老代的中间交换区域。当年老代空间足够时，Survivor区中存活了一定次数的对象会被移到年老代。
5. 当年老代空间不够时，JVM会在年老代进行完全的垃圾回收（Full GC）。
6. Full GC后，若Survivor区及年老代仍然无法存放从Eden区复制过来的对象，则会导致JVM无法在Eden区为新生成的对象申请内存，即出现“Out of Memory”。

昵称：vincent_hv
园龄：10个月
粉丝：7
关注：1
+加关注

< 2013年10月 >

日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔
1. linux解压zip乱码解决方案
2. 全能系统监控工具dstat
3. 【转】linux sar命令详解
4. 【原】gnome3增加自定义程序快捷方式
5. 【原】Ubuntu13.04安装、卸载Gnome3.8
6. 【原】安装、卸载、查看软件时常用的命令
7. 【原】中文Ubuntu主目录下的文档文件夹改回英文
8. 【原】Ubuntu ATI/Intel双显卡 驱动安装
9. 【原】Ubuntu 12.04 ATI显卡设置双屏显示
10. 【转】Hadoop vs Spark性能对比

随笔分类
Android(8)
Hadoop(2)
Java(20)
JVM(3)
Linux(23)
others(1)
Scala(5)
Spark(20)
数据结构与算法(2)

OOM（“Out of Memory”）异常一般主要有如下2种原因：

1. 年老代溢出，表现为：java.lang.OutOfMemoryError:Javaheapspace
这是最常见的情况，产生的原因可能是：设置的内存参数Xmx过小或程序的内存泄露及使用不当问题。
例如循环上万次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存。还有的时候虽然不会报内存溢出，却会使系统不间断的垃圾回收，也无法处理其它请求。这种情况下除了检查程序、打印堆内存等方法排查，还可以借助一些内存分析工具，比如MAT就很不错。

2. 持久代溢出，表现为：java.lang.OutOfMemoryError:PermGenspace
通常由于持久代设置过小，动态加载了大量Java类而导致溢出，解决办法唯有将参数 -XX:MaxPermSize 调大（一般256m能满足绝大多数应用程序需求）。将部分Java类放到容器共享区（例如Tomcat share lib）去加载的办法也是一个思路，但前提是容器里部署了多个应用，且这些应用有大量的共享类库。

参数说明

- -Xmx3550m：设置JVM**最大堆内存**为3550M。
- -Xms3550m：设置JVM**初始堆内存**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。
- -Xss128k：设置每个线程的栈大小。JDK5.0以后每个线程栈大小为1M，之前每个线程栈大小为256K。应当根据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。需要注意的是：当这个值被设置的较大（例如>2MB）时将会在很大程度上降低系统的性能。
- -Xmn2g：设置**年轻代**大小为2G。在整个堆内存大小确定的情况下，增大年轻代将会减小年老代，反之亦然。此值关系到JVM垃圾回收，对系统性能影响较大，官方推荐配置为整个堆大小的3/8。
- -XX:NewSize=1024m：设置年轻代初始值为1024M。
- -XX:MaxNewSize=1024m：设置年轻代最大值为1024M。
- -XX:PermSize=256m：设置**持久代初始值**为256M。
- -XX:MaxPermSize=256m：设置**持久代最大值**为256M。
- -XX:NewRatio=4：设置年轻代（包括1个Eden和2个Survivor区）与年老代的比值。表示年轻代比年老代为1:4。
- -XX:SurvivorRatio=4：设置**年轻代中Eden区与Survivor区的比值**。表示2个Survivor区（JVM堆内存年轻代中默认有2个大小相等的Survivor区）与1个Eden区的比值为2:4，即1个Survivor区占整个年轻代大小的1/6。
- -XX:MaxTenuringThreshold=7：表示一个对象如果在Survivor区（救助空间）移动了7次还没有被垃圾回收就进入年老代。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，对于需要大量常驻内存的应用，这样做可以提高效率。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，这样可以增加对象在年轻代存活时间，增加对象在年轻代被垃圾回收的概率，减少Full GC的频率，这样做可以在某种程度上提高服务稳定性。

疑问解答

-Xmn，-XX:NewSize/-XX:MaxNewSize，-XX:NewRatio 3组参数都可以影响年轻代的大小，混合使用的情况下，优先级是什么？
如下：

1. 高优先级：-XX:NewSize/-XX:MaxNewSize
2. 中优先级：-Xmn（默认等效 -Xmn=-XX:NewSize=-XX:MaxNewSize=?）
3. 低优先级：-XX:NewRatio

推荐使用-Xmn参数，原因是这个参数简洁，相当于一次设定 NewSize/MaxNewSIze，而且两者相等，适用于生产环境。-Xmn 配合 -Xms/-Xmx，即可将堆内存布局完成。
-Xmn参数是在JDK 1.4 开始支持。

垃圾回收器选择

JVM给出了3种选择：串行收集器、并行收集器、并发收集器。串行收集器只适用于小数据量的情况，所以生产环境的选择主要是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智能判断。

串行收集器

- -XX:+UseSerialGC：设置串行收集器。

并行收集器（吞吐量优先）

- -XX:+UseParallelGC：设置为并行收集器。此配置仅对年轻代有效。即年轻代使用并行收集，而年老代仍

使用串行收集。

- -XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时有多少个线程一起进行垃圾回收。此值建议配置与CPU数目相等。
- -XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0开始支持对年老代并行收集。
- -XX:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间（单位毫秒）。如果无法满足此时间，JVM会自动调整年轻代大小，以满足此时间。
- -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动调整年轻代Eden区大小和Survivor区大小的比例，以达成目标系统规定的最低响应时间或者收集频率等指标。此参数建议在使用并行收集器时，一直打开。

并发收集器（响应时间优先）

- -XX:+UseConcMarkSweepGC：即CMS收集，设置年老代为并发收集。CMS收集是JDK1.4后期版本开始引入的新GC算法。它的主要适合场景是对响应时间的重要性需求大于对吞吐量的需求，能够承受垃圾回收线程和应用线程共享CPU资源，并且应用中存在比较多的长生命周期对象。CMS收集的目标是尽量减少应用的暂停时间，减少Full GC发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代内存。
- -XX:+UseParNewGC：设置年轻代为并发收集。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，所以无需再设置此参数。
- -XX:CMSFullGCsBeforeCompaction=0：由于并发收集器不对内存空间进行压缩和整理，所以运行一段时间并行收集以后会产生内存碎片，内存使用效率降低。此参数设置运行0次Full GC后对内存空间进行压缩和整理，即每次Full GC后立刻开始压缩和整理内存。
- -XX:+UseCMSCompactAtFullCollection：打开内存空间的压缩和整理，在Full GC后执行。可能会影响性能，但可以消除内存碎片。
- -XX:+CMSIncrementalMode：设置为增量收集模式。一般适用于单CPU情况。
- -XX:CMSInitiatingOccupancyFraction=70：表示年老代内存空间使用到70%时就开始执行CMS收集，以确保年老代有足够的空间接纳来自年轻代的对象，避免Full GC的发生。

其它垃圾回收参数

- -XX:+ScavengeBeforeFullGC：年轻代GC优于Full GC执行。
- -XX:-DisableExplicitGC：不响应 System.gc() 代码。
- -XX:+UseThreadPriorities：启用本地线程优先级API。即使 java.lang.Thread.setPriority() 生效，不启用则无效。
- -XX:SoftRefLRUPolicyMSPerMB=0：软引用对象在最后一次被访问后能存活0毫秒（JVM默认为1000毫秒）。
- -XX:TargetSurvivorRatio=90：允许90%的Survivor区被占用（JVM默认为50%）。提高对于Survivor区的使用率。

辅助信息参数设置

- -XX:-CITime：打印消耗在JIT编译的时间。
- -XX:ErrorFile=./hs_err_pid.log：保存错误日志或数据到指定文件中。
- **-XX:HeapDumpPath=./java_pid.hprof：指定Dump堆内存时的路径。**
- **-XX:-HeapDumpOnOutOfMemoryError：当首次遭遇内存溢出时Dump出此时的堆内存。**
- -XX:OnError=";"：出现致命ERROR后运行自定义命令。
- -XX:OnOutOfMemoryError=";"：当首次遭遇内存溢出时执行自定义命令。
- -XX:-PrintClassHistogram：按下 Ctrl+Break 后打印堆内存中类实例的柱状信息，同JDK的 jmap -histo 命令。
- -XX:-PrintConcurrentLocks：按下 Ctrl+Break 后打印线程栈中并发锁的相关信息，同JDK的 jstack -l 命令。
- -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:-PrintGCDetails：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。

关于参数名称等

- 标准参数（-），所有JVM都必须支持这些参数的功能，而且向后兼容；例如：
    - **-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开发调试；在32位环境下直接运行Java程序默认启用该模式。
    - **-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环境下默认启用该模式。
- 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；
- 非稳定参数（-XX），此类参数各个JVM实现会有所不同，将来可能会不被支持，需要慎重使用；

JVM服务参数调优实战

大型网站服务器案例

承受海量访问的动态Web应用

服务器配置：8 CPU, 8G MEM, JDK 1.6.X

参数方案：

-server -Xmx3550m -Xms3550m -Xmn1256m -Xss128k -XX:SurvivorRatio=6 -XX:MaxPermSize=256m -XX:ParallelGCThreads=8 -XX:MaxTenuringThreshold=0 -XX:+UseConcMarkSweepGC

调优说明：

- -Xmx 与 -Xms 相同以避免JVM反复重新申请内存。-Xmx 的大小约等于系统内存大小的一半，即充分利用系统资源，又给予系统安全运行的空间。
- -Xmn1256m 设置年轻代大小为1256MB。此值对系统性能影响较大，Sun官方推荐配置年轻代大小为整个堆的3/8。
- -Xss128k 设置较小的线程栈以支持创建更多的线程，支持海量访问，并提升系统性能。
- -XX:SurvivorRatio=6 设置年轻代中Eden区与Survivor区的比值。系统默认是8，根据经验设置为6，则2个Survivor区与1个Eden区的比值为2:6，一个Survivor区占整个年轻代的1/8。
- -XX:ParallelGCThreads=8 配置并行收集器的线程数，即同时8个线程一起进行垃圾回收。此值一般配置为与CPU数目相等。
- -XX:MaxTenuringThreshold=0 设置垃圾最大年龄（在年轻代的存活次数）。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。根据被海量访问的动态Web应用之特点，其内存要么被缓存起来以减少直接访问DB，要么被快速回收以支持高并发海量请求，因此其内存对象在年轻代存活多次意义不大，可以直接进入年老代，根据实际应用效果，在这里设置此值为0。
- -XX:+UseConcMarkSweepGC 设置年老代为并发收集。CMS（ConcMarkSweepGC）收集的目标是尽量减少应用的暂停时间，减少Full GC发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代内存，适用于应用中存在比较多的长生命周期对象的情况。

内部集成构建服务器案例

高性能数据处理的工具应用

服务器配置：1 CPU, 4G MEM, JDK 1.6.X

参数方案：

-server -XX:PermSize=196m -XX:MaxPermSize=196m -Xmn320m -Xms768m -Xmx1024m

调优说明：

- -XX:PermSize=196m -XX:MaxPermSize=196m 根据集成构建的特点，大规模的系统编译可能需要加载大量的Java类到内存中，所以预先分配好大量的持久代内存是高效和必要的。
- -Xmn320m 遵循年轻代大小为整个堆的3/8原则。
- -Xms768m -Xmx1024m 根据系统大致能够承受的堆内存大小设置即可。

在64位服务器上运行应用程序，构建执行时，用 jmap -heap 11540 命令观察JVM堆内存状况如下：

```
Attaching to process ID 11540, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.12-b01

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 1073741824 (1024.0MB)
   NewSize          = 335544320 (320.0MB)
   MaxNewSize       = 335544320 (320.0MB)
   OldSize          = 5439488 (5.1875MB)
   NewRatio         = 2
   SurvivorRatio    = 8
   PermSize         = 205520896 (196.0MB)
   MaxPermSize      = 205520896 (196.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 255852544 (244.0MB)
   used     = 101395504 (96.69828796386719MB)
   free     = 154457040 (147.3017120361328MB)
   39.63044588683081% used
From Space:
   capacity = 34144256 (32.5625MB)
   used     = 33993968 (32.41917419433594MB)
   free     = 150288 (0.1433258056640625MB)
   99.55984397492803% used
To Space:
   capacity = 39845888 (38.0MB)
   used     = 0 (0.0MB)
   free     = 39845888 (38.0MB)
   0.0% used
PS Old Generation
   capacity = 469762048 (448.0MB)
   used     = 44347696 (42.29325866699219MB)
   free     = 425414352 (405.7067413330078MB)
   9.440459523882184% used
PS Perm Generation
   capacity = 205520896 (196.0MB)
   used     = 85169496 (81.22396087646484MB)
   free     = 120351400 (114.77603912353516MB)
   41.440796365543285% used
```

结果是比较健康的。

分类: JVM

« 上一篇：【转】JVM参数设置、分析
» 下一篇：【转】弹性分布式数据集：一种基于内存的集群计算的容错性抽象方法

posted @ 2013-09-21 17:45 vincent_hv 阅读(175) 评论(0) 编辑 收藏