JVM内存配置详解

基本概念   
堆/Heap   
JVM管理的内存叫堆；在32Bit操作系统上有4G的限制，一般来说Windows下为2G，而Linux 下为3G；64Bit的就没有这个限制。   
JVM初始分配的内存由-Xms指定，默认是物理内存的1/64但小于1G。   
JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4但小于1G。   
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，可以由 -XX:MinHeapFreeRatio=指定。   
默认空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制，可以由 -XX:MaxHeapFreeRatio=指定。   
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，所以上面的两个参数没啥用。   
  
分代/堆模型   
分代是Java垃圾收集的一大亮点，根据对象的生命周期长短，把堆分为3个代：Young，Old和Permanent，根据不同代的特点采用不同的收集算法，可以扬长避短。可参考如下的模型图：   
  
Young(Nursery)：年轻代   
研究表明大部分对象都是朝生暮死，随生随灭的。所以对于年轻代在GC时都采取复制收集算法，具体算法参考下面的描述；   
Young的默认值为4M，随堆内存增大，约为1/15，JVM会根据情况动态管理其大小变化。   
Young里面又分为3 个区域，一个Eden，所有新建对象都会存在于该区，两个Survivor区，用来实施复制算法。   
-XX:NewRatio= 参数可以设置Young与Old的大小比例，-server时默认为1:2，但实际上young启动时远低于这个比率？如果信不过JVM，也可以用 -Xmn硬性规定其大小，有文档推荐设为Heap总大小的1/4。   
-XX:SurvivorRatio= 参数可以设置Eden与Survivor的比例，默认为32。Survivio大了会浪费，小了的话，会使一些年轻对象潜逃到老人区，引起老人区的不安，但这个参数对性能并不太重要。   
  
Old(Tenured)：年老代   
年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法，但标记清理算法其实也不轻松，每次都要遍历区域内所有对象，所以还是没有免费的午餐啊。   
-XX:MaxTenuringThreshold= 设置熬过年轻代多少次收集后移入老人区，CMS中默认为0，熬过第一次GC就转入，可以用-XX:+PrintTenuringDistribution 查看。   
  
Permanent：持久代   
装载Class信息等基础数据，默认64M，如果是类很多很多的服务程序，需要加大其设置 -XX:MaxPermSize=，否则它满了之后会引起fullgc()或Out of Memory。 注意Spring，Hibernate这类喜欢AOP动态生成类的框架需要更多的持久代内存。一般情况下，持久代是不会进行GC的，除非通过 -XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled进行强制设置。   
  
GC的类型   
当每个代满了之后都会自动促发collection，各收集器触发的条件不一样，当然也可以通过一些参数进行强制设定。主要分为两种类型：   
  
Minor Collection：GC用较高的频率对young进行扫描和回收，采用复制算法。   
Major Collection：同时对Young和Old进行内存收集，也叫Full GC；因为成本关系对Old的检查回收频率要比Young低很多，采用标记清除/标记整理算法。可以通过调用代码System.gc()引发major collection，使用-XX:+DisableExplicitGC禁止它，或设为CMS并发 -XX:+ExplicitGCInvokesConcurrent。   
更为具体的阐述如下：   
由于年轻代进进出出的人多而频繁，所以年轻代的GC也就频繁一点，但涉及范围也就年轻代这点弹丸之地内的对象，其特点就是少量，多次，但快速，称之为 Minor Collection。当年轻代的内存使用达到一定的阀值时，Minor Collection就被触发，Eden及某一Survior space（from space）之内存活的的对象被移到另一个空的Survior space（to space）中，然后from space和to space角色对调。当一个对象在两个survivor space之间移动过一定次数（达到预设的阀值）时，它就足够old了，够资格呆在年老代了。当然，如果survivor space比较小不足以容下所有live objects时，部分live objects也会直接晋升到年老代。   
Survior spaces可以看作是Eden和年老代之间的缓冲，通过该缓冲可以检验一个对象生命周期是否足够的长，因为某些对象虽然逃过了一次Minor Collection，并不能说明其生命周期足够长，说不定在下一次Minor Collection之前就挂了。这样一定程度上确保了进入年老代的对象是货真价实的，减少了年老代空间使用的增长速度，也就降低年老代GC的频率。   
当年老代或者永久代的内存使用达到一定阀值时，一次基于所有代的GC就触发了，其特定是涉及范围广（量大），耗费的时间相对较长（较慢），但是频率比较低（次数少），称之为Major Collection(Full Collection)。通常，首先使用针对年轻代的GC算法进行年轻代的GC，然后使用针对年老代的GC算法对年老代和永久代进行GC。  
  
基本GC收集算法   
复制(copying)：将堆内分成两个相同空间，从根(ThreadLocal的对象，静态对象）开始访问每一个关联的活跃对象，将空间A的活跃对象全部复制到空间B，然后一次性回收整个空间A。   
因为只访问活跃对象，将所有活动对象复制走之后就清空整个空间，不用去访问死对象，所以遍历空间的成本较小，但需要巨大的复制成本和较多的内存。可参考如下的示例图：   
  
标记清除(mark-sweep)：收集器先从根开始访问所有活跃对象，标记为活跃对象。然后再遍历一次整个内存区域，把所有没有标记活跃的对象进行回收处理。该算法遍历整个空间的成本较大暂停时间随空间大小线性增大，而且整理后堆里的碎片很多。可参考如下的示例图：   
  
标记整理(mark-sweep-compact)：综合了上述两者的做法和优点，先标记活跃对象，然后将其合并成较大的内存块。可参考如下的示例图：   
  
GC收集器类型   
古老的串行收集器(Serial Collector)   
-XX:+UseSerialGC：策略为年轻代串行复制，年老代串行标记整理。可参考如下的示例图：   
吞吐量优先的并行收集器(Throughput Collector)   
-XX:+UseParallelGC：这是JDK5 -server的默认值。策略为：   
年轻代：暂停应用程序，多个垃圾收集线程并行的复制收集，线程数默认为CPU个数，CPU很多时，可用 -XX:ParallelGCThreads= 设定线程数。   
年老代：暂停应用程序，与串行收集器一样，单垃圾收集线程标记整理。   
如上可知该收集器需要2+的CPU时才会优于串行收集器，适用于后台处理，科学计算。   
可以使用-XX:MaxGCPauseMillis= 和 -XX:GCTimeRatio 来调整GC的时间。可参考如下的示例图：   
暂停时间优先的并发收集器(Concurrent Low Pause Collector-CMS)   
-XX:+UseConcMarkSweepGC：这是以上两种策略的升级版，策略为：   
年轻代：同样是暂停应用程序，多个垃圾收集线程并行的复制收集。   
年老代：则只有两次短暂停，其他时间应用程序与收集线程并发的清除。   
若要采用标记整理算法，则可以通过设置参数实现；可参考如下的示例图：   
增量并发收集器(Incremental Concurrent-Mark-Sweep/i-CMS)：虽然CMS收集算法在最为耗时的内存区域遍历时采用多线程并发操作，但对于服务器CPU资源不够的情况下，其实对性能是没有提升的，反而会导致系统吞吐量的下降，为了尽量避免这种情况的出现，就有了增量CMS收集算法，就是在并发标记、清理的时候让GC线程、用户线程交叉运行，尽量减少GC线程的全程独占式执行；可参考如下的示例图：   
对于以上的GC收集器的详细设置参数，可以参考 JVM选项的超完整收集《A Collection of JVM Options》，这里就不一一详述了。   
  
并行、并发的区别   
并行(Parallel)与并发(Concurrent)仅一字之差，但体现的意思却完全不同，这可能也是很多同学非常困惑的地方，要想深刻体会这其中的差别，可以多揣摩下上面关于GC收集器的示例图；   
  
并行：指多条垃圾收集线程并行，此时用户线程是没有运行的；   
并发：指用户线程与垃圾收集线程并发执行，程序在继续运行，而垃圾收集程序运行于另一个个CPU上。   
并发收集一开始会很短暂的停止一次所有线程来开始初始标记根对象，然后标记线程与应用线程一起并发运行，最后又很短的暂停一次，多线程并行的重新标记之前可能因为并发而漏掉的对象，然后就开始与应用程序并发的清除过程。可见，最长的两个遍历过程都是与应用程序并发执行的，比以前的串行算法改进太多太多了！！！   
串行标记清除是等年老代满了再开始收集的，而并发收集因为要与应用程序一起运行，如果满了才收集，应用程序就无内存可用，所以系统默认68%满的时候就开始收集。内存已设得较大，吃内存又没有这么快的时候，可以用 -XX:CMSInitiatingOccupancyFraction=恰当增大该比率。   
  
年轻代的痛   
由于对年轻代的复制收集，依然必须停止所有应用程序线程，原理如此，只能靠多CPU，多收集线程并发来提高收集速度，但除非你的 Server独占整台服务器，否则如果服务器上本身还有很多其他线程时，切换起来速度就..... 所以，搞到最后，暂停时间的瓶颈就落在了年轻代的复制算法上。   
因此Young的大小设置挺重要的，大点就不用频繁GC，而且增大GC的间隔后，可以让多点对象自己死掉而不用复制了。但Young增大时，GC造成的停顿时间攀升得非常恐怖，据某人的测试结果显示：默认8M的Young，只需要几毫秒的时间，64M就升到90毫秒，而升到256M时，就要到300毫秒了，峰值还会攀到恐怖的800ms。谁叫复制算法，要等Young满了才开始收集，开始收集就要停止所有线程呢。   
  
参考资料   
主要参考：JDK5.0垃圾收集优化之--Don't Pause   
官方指南：Tuning Garbage Collection with the 5.0 Java Virtual Machine   
  
　　Sun HotSpot 1.4.1 JVM堆大小的调整   
　　   
　　Sun HotSpot 1.4.1使用分代收集器，它把堆分为三个主要的域：新域、旧域以及永久域。Jvm生成的所有新对象放在新域中。一旦对象经历了一定数量的垃圾收集循环后，便获得使用期并进入旧域。在永久域中jvm则存储class和method对象。就配置而言，永久域是一个独立域并且不认为是堆的一部分。   
　　   
　　下面介绍如何控制这些域的大小。可使用-Xms和-Xmx 控制整个堆的原始大小或最大值。   
　　   
　　下面的命令是把初始大小设置为128M：   
　　   
　　java –Xms128m   
　　   
　　–Xmx256m为控制新域的大小，可使用 -XX:NewRatio设置新域在堆中所占的比例。   
　　   
　　下面的命令把整个堆设置成128m，新域比率设置成3，即新域与旧域比例为 1：3，新域为堆的1/4或32M：   
　　   
　　java –Xms128m –Xmx128m   
　　   
　　 –XX:NewRatio =3可使用-XX:NewSize和-XX:MaxNewsize设置新域的初始值和最大值。   
　　   
　　下面的命令把新域的初始值和最大值设置成64m:   
　　   
　　java –Xms256m –Xmx256m –Xmn64m   
　　   
　　永久域默认大小为4m。运行程序时，jvm会调整永久域的大小以满足需要。每次调整时，jvm会对堆进行一次完全的垃圾收集。   
　　   
　　使用-XX:MaXPerSize标志来增加永久域搭大小。在WebLogic Server应用程序加载较多类时，经常需要增加永久域的最大值。当jvm加载类时，永久域中的对象急剧增加，从而使jvm不断调整永久域大小。为了避免调整，可使用-XX:PerSize标志设置初始值。   
　　   
　　下面把永久域初始值设置成32m，最大值设置成64m。   
　　   
　　java -Xms512m -Xmx512m -Xmn128m -XX:PermSize=32m -XX:MaxPermSize=64m   
　　   
　　默认状态下，HotSpot在新域中使用复制收集器。该域一般分为三个部分。第一部分为Eden，用于生成新的对象。另两部分称为救助空间，当 Eden布满时，收集器停止应用程序，把所有可到达对象复制到当前的from救助空间，一旦当前的from救助空间布满，收集器则把可到达对象复制到当前的to救助空间。From和to救助空间互换角色。维持活动的对象将在救助空间不断复制，直到它们获得使用期并转入旧域。使用 -XX:SurvivorRatio可控制新域子空间的大小。   
　　   
　　同NewRation一样，SurvivorRation规定某救助域与Eden空间的比值。比如，以下命令把新域设置成64m，Eden占32m，每个救助域各占16m：   
　　   
　　java -Xms256m -Xmx256m -Xmn64m -XX:SurvivorRation =2   
　　   
　　如前所述，默认状态下 HotSpot对新域使用复制收集器，对旧域使用标记－清除－压缩收集器。在新域中使用复制收集器有很多意义，因为应用程序生成的大部分对象是短寿命的。理想状态下，所有过渡对象在移出Eden空间时将被收集。假如能够这样的话，并且移出Eden空间的对象是长寿命的，那么理论上可以立即把它们移进旧域，避免在救助空间反复复制。但是，应用程序不能适合这种理想状态，因为它们有一小部分中长寿命的对象。最好是保持这些中长寿命的对象并放在新域中，因为复制小部分的对象总比压缩旧域廉价。为控制新域中对象的复制，可用-XX:TargetSurvivorRatio控制救助空间的比例（该值是设置救助空间的使用比例。如救助空间位1M，该值50表示可用500K）。该值是一个百分比，默认值是50。当较大的堆栈使用较低的sruvivorratio时，应增加该值到80至90，以更好利用救助空间。用-XX:maxtenuring threshold可控制上限。   
　　   
　　为放置所有的复制全部发生以及希望对象从eden扩展到旧域，可以把MaxTenuring Threshold设置成0。设置完成后，实际上就不再使用救助空间了，因此应把SurvivorRatio设成最大值以最大化Eden空间，设置如下：   
　　   
　　java … -XX:MaxTenuringThreshold=0 –XX:SurvivorRatio＝50000 …   
  
  
  
-Xmx4000M -Xms4000M -Xmn600M -XX:PermSize=64M -XX:MaxPermSize=128M -Xss256K -XX:+DisableExplicitGC -XX:SurvivorRatio=1 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:LargePageSizeInBytes=128M -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=80 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log   
  
  
  
jdk1.4.2 JVM官方地址：http://java.sun.com/j2se/1.4.2/docs/guide/vm/index.html   
标准和非标注参数(for windows)：http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/java.html   
非 stable参数：http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp   
  
中文地址：   
http://blog.csdn.net/sfdev/archive/2008/01/23/2062042.aspx                              
  
  
  
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.查看原文；   
  
首先来介绍行为参数：   
  
  
  
  
参数及其默认值 描述   
-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: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   
  
我们在日常性能调优中基本上都会用到以上黑体的这几个属性；   
  
  
  
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="<cmd args>;<cmd args>" 出现致命ERROR之后运行自定义命令   
-XX:OnOutOfMemoryError="<cmd args>;<cmd args>" 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   
  
-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。   
  
-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。   
  
-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。   
  
/usr/local/jdk/bin/java -Dresin.home=/usr/local/resin -server -Xms1800M -Xmx1800M -Xmn300M -Xss512K -XX:PermSize=300M -XX:MaxPermSize=300M -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=5 -XX:GCTimeRatio=19 -Xnoclassgc -XX:+DisableExplicitGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:-CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log   
  
堆大小设置   
  
JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制。我在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JVM参数设置：   
  
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。   
  
-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JVM参数配置：   
  
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JVM参数配置：   
  
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JVM参数配置汇总   
  
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。   
  
四、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：   
  
并发垃圾收集信息   
  
持久代并发收集次数   
  
传统GC信息   
  
花在年轻代和年老代回收上的时间比例   
  
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JVM参数配置：   
  
-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  
  
1: heap size   
  
a: -Xmx<n>                        
  
指定 jvm 的最大 heap 大小 , 如 :-Xmx=2g   
  
b: -Xms<n>                        
  
指定 jvm 的最小 heap 大小 , 如 :-Xms=2g ， 高并发应用， 建议和-Xmx一样， 防止因为内存收缩／突然增大带来的性能影响。   
  
c: -Xmn<n>                        
  
指定 jvm 中 New Generation 的大小 , 如 :-Xmn256m。 这个参数很影响性能， 如果你的程序需要比较多的临时内存， 建议设置到512M， 如果用的少， 尽量降低这个数值， 一般来说128／256足以使用了。   
  
d: -XX:PermSize=<n>   
  
指定 jvm 中 Perm Generation 的最小值 , 如 :-XX:PermSize=32m。 这个参数需要看你的实际情况，。 可以通过jmap 命令看看到底需要多少。   
  
e: -XX:MaxPermSize=<n>            
  
指定 Perm Generation 的最大值 , 如 :-XX:MaxPermSize=64m   
  
f: -Xss<n>                        
  
指定线程桟大小 , 如 :-Xss128k， 一般来说，webx框架下的应用需要256K。 如果你的程序有大规模的递归行为， 请考虑设置到512K／1M。 这个需要全面的测试才能知道。 不过， 256K已经很大了。 这个参数对性能的影响比较大的。   
  
g: -XX:NewRatio=<n>   
  
指定 jvm 中 Old Generation heap size 与 New Generation 的比例 , 在使用 CMS GC 的情况下此参数失效 , 如 :-XX:NewRatio=2   
  
h: -XX:SurvivorRatio=<n>   
  
指定 New Generation 中 Eden Space 与一个 Survivor Space 的 heap size 比例 ,-XX:SurvivorRatio=8, 那么在总共 New Generation 为 10m 的情况下 ,Eden Space 为 8m   
  
i: -XX:MinHeapFreeRatio=<n>   
  
指定 jvm heap 在使用率小于 n 的情况下 ,heap 进行收缩 ,Xmx==Xms 的情况下无效 , 如 :-XX:MinHeapFreeRatio=30   
  
j: -XX:MaxHeapFreeRatio=<n>   
  
指定 jvm heap 在使用率大于 n 的情况下 ,heap 进行扩张 ,Xmx==Xms 的情况下无效 , 如 :-XX:MaxHeapFreeRatio=70   
  
k: -XX:LargePageSizeInBytes=<n>   
  
指定 Java heap 的分页页面大小 , 如 :-XX:LargePageSizeInBytes=128m   
  
2: garbage collector   
  
a: -XX:+UseParallelGC   
  
指定在 New Generation 使用 parallel collector, 并行收集 , 暂停 app threads, 同时启动多个垃圾回收 thread, 不能和 CMS gc 一起使用 . 系统吨吐量优先 , 但是会有较长长时间的 app pause, 后台系统任务可以使用此 gc   
  
b: -XX:ParallelGCThreads=<n>   
  
指定 parallel collection 时启动的 thread 个数 , 默认是物理 processor 的个数 ,   
  
c: -XX:+UseParallelOldGC   
  
指定在 Old Generation 使用 parallel collector   
  
d: -XX:+UseParNewGC   
  
指定在 New Generation 使用 parallel collector, 是 UseParallelGC 的 gc 的升级版本 , 有更好的性能或者优点 , 可以和 CMS gc 一起使用   
  
e: -XX:+CMSParallelRemarkEnabled   
  
在使用 UseParNewGC 的情况下 , 尽量减少 mark 的时间   
  
f: -XX:+UseConcMarkSweepGC   
  
指定在 Old Generation 使用 concurrent cmark sweep gc,gc thread 和 app thread 并行 ( 在 init-mark 和 remark 时 pause app thread). app pause 时间较短 , 适合交互性强的系统 , 如 web server   
  
g: -XX:+UseCMSCompactAtFullCollection   
  
在使用 concurrent gc 的情况下 , 防止 memory fragmention, 对 live object 进行整理 , 使 memory 碎片减少   
  
h: -XX:CMSInitiatingOccupancyFraction=<n>   
  
指示在 old generation 在使用了 n% 的比例后 , 启动 concurrent collector, 默认值是 68, 如 :-XX:CMSInitiatingOccupancyFraction=70   
  
有个 bug, 在低版本(1.5.09 and early)的 jvm 上出现 , http://bugs.sun.com/bugdatabase/view\_bug.do?bug\_id=6486089   
  
i: -XX:+UseCMSInitiatingOccupancyOnly   
  
指示只有在 old generation 在使用了初始化的比例后 concurrent collector 启动收集   
  
3:others   
  
a: -XX:MaxTenuringThreshold=<n>   
  
指定一个 object 在经历了 n 次 young gc 后转移到 old generation 区 , 在 linux64 的 java6 下默认值是 15, 此参数对于 throughput collector 无效 , 如 :-XX:MaxTenuringThreshold=31   
  
b: -XX:+DisableExplicitGC   
  
禁止 java 程序中的 full gc, 如 System.gc() 的调用. 最好加上么， 防止程序在代码里误用了。对性能造成冲击。   
  
c: -XX:+UseFastAccessorMethods   
  
get,set 方法转成本地代码   
  
d: -XX:+PrintGCDetails   
  
打应垃圾收集的情况如 :   
  
[GC 15610.466: [ParNew: 229689K->20221K(235968K), 0.0194460 secs] 1159829K->953935K(2070976K), 0.0196420 secs]   
  
e: -XX:+PrintGCTimeStamps   
  
打应垃圾收集的时间情况 , 如 :   
  
[Times: user=0.09 sys=0.00, real=0.02 secs]   
  
f: -XX:+PrintGCApplicationStoppedTime   
  
打应垃圾收集时 , 系统的停顿时间 , 如 :   
  
Total time for which application threads were stopped: 0.0225920 seconds   
  
4: a web server product sample and process   
  
JAVA\_OPTS=" -server -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 "   
  
最初的时候我们用 UseParallelGC 和 UseParallelOldGC,heap 开了 3G,NewRatio 设成 1. 这样的配置下 young gc 发生频率约 12,3 妙一次 , 平均每次花费 80ms 左右 ,full gc 发生的频率极低 , 每次消耗 1s 左右 . 从所有 gc 消耗系统时间看 , 系统使用率还是满高的 , 但是不论是 young gc 还是 old gc,applicaton thread pause 的时间比较长 , 不合适 web 应用 . 我们也调小 New Generation 的 , 但是这样会使 full gc 时间加长 .   
  
后来我们就用 CMS gc(-XX:+UseConcMarkSweepGC), 当时的总 heap 还是 3g, 新生代 1.5g 后 , 观察不是很理想 , 改为 jvm heap 为 2g 新生代设置 -Xmn1g, 在这样的情况下 young gc 发生的频率变成 ,7,8 妙一次 , 平均每次时间 40~50 毫秒左右 ,CMS gc 很少发生 , 每次时间在 init-mark 和 remark(two steps stop all app thread) 总共平均花费 80~90ms 左右 .   
  
在这里我们曾经 New Generation 调大到 1400m, 总共 2g 的 jvm heap, 平均每次 ygc 花费时间 60~70ms 左右 ,CMS gc 的 init-mark 和 remark 之和平均在 50ms 左右 , 这里我们意识到错误的方向 , 或者说 CMS 的作用 , 所以进行了修改   
  
最后我们调小 New Generation 为 256m,young gc 2,3 秒发生一次 , 平均停顿时间在 25 毫秒左右 ,CMS gc 的 init-mark 和 remark 之和平均在 50ms 左右 , 这样使系统比较平滑 , 经压力测试 , 这个配置下系统性能是比较高的   
  
在使用 CMS gc 的时候他有两种触发 gc 的方式 :gc 估算触发和 heap 占用触发 . 我们的 1.5.0.09 环境下有次 old 区 heap 占用再 30% 左右 , 她就频繁 gc, 个人感觉系统估算触发这种方式不靠谱 , 还是用 heap 使用比率触发比较稳妥 .   
  
这些数据都来自 64 位测试机 , 过程中的数据都是我在 jboss log 找的 , 当时没有记下来 , 可能存在一点点偏差 , 但不会很大 , 基本过程就是这样 .   
  
5: 总结   
  
web server 作为交互性要求较高的应用 , 我们应该使用 Parallel+CMS,UseParNewGC 这个在 jdk6 -server 上是默认的 ,new generation gc, 新生代不能太大 , 这样每次 pause 会短一些 .CMS mark-sweep generation 可以大一些 , 可以根据 pause time 实际情况控制