**一、总结前一天的学习**

从“第三天”的性能测试一节中，我们得知了决定性能测试的几个重要指标，它们是：

ü   吞吐量

ü   Responsetime

ü   Cpuload

ü   MemoryUsage

我 们也在第三天的学习中对Apache做过了一定的优化，使其最优化上述4大核心指标的读数，那么我们的Apache调优了，我们的Tomcat也作些相应 的调整，当完成今的课程后，到时你的“小猫”到时真的会“飞”起来的，所以请用心看完，这篇文章一方面用来向那位曾写过“Tomcat如何承受1000个 用户”的作都的敬，一方面又是这篇原文的一个扩展，因为在把原文的知识用到相关的两个大工程中去后解决了：

1)      承受更大并发用户数

2)      取得了良好的性能与改善（系统平均性能提升达20倍，极端一个交易达80倍）。

另外值的一提的是，我们当时工程里用的“小猫”是跑在32位机下的， 也就是我们的JVM最大受到2GB内存的限制，都已经跑成“飞”了。。。。。。如果在64位机下跑这头“小猫”。。。。。。大家可想而知，会得到什么样的效果呢？下面就请请详细的设置吧！



**二、一切基于JVM（内存）的优化**

**2.1 32位操作系统与64位操作系统中JVM的对比**

我们一般的开发人员，基本用的是都是32位的Windows系统，这就导致了一个严重的问题即：32位windows系统对内存限制，下面先来看一个比较的表格：

|  |  |  |  |
| --- | --- | --- | --- |
| **操作系统** | **操作系统位数** | **内存限制** | **解决办法** |
| **Winxp** | 32 | 4GB | 超级兔子 |
| **Win7** | 32 | 4GB | 可以通过设置/PAE |
| **Win2003** | 32 | 可以突破4GB达16GB | 必需要装win2003 advanced server且要打上sp2补丁 |
| **Win7** | 64 | 无限制 | 机器能插多少内存，系统内存就能支持到多大 |
| **Win2003** | 64 | 无限制 | 机器能插多少内存，系统内存就能支持到多大 |
| **Linux** | 64 | 无限制 | 机器能插多少内存，系统内存就能支持到多大 |
| **Unix** | 64 | 无限制 | 机器能插多少内存，系统内存就能支持到多大 |

上述问题解决后，我们又碰到一个新的问题，32位系统下JVM对内存的限制：不能突破2GB内存，即使你在Win2003 Advanced Server下你的机器装有8GB-16GB的内存，而你的JAVA，只能用到2GB的内存。

其实我一直很想推荐大家使用Linux或者是Mac操作系统的，而且要装64位，因为必竟我们是开发用的不是打游戏用的，而Java源自Unix归于Unix（Linux只是运行在PC上的Unix而己）。

所以很多开发人员运行在win32位系统上更有甚者在生产环境下都会布署win32位的系统，那么这时你的Tomcat要优化，就要讲究点技巧了。**而在64位操作系统上无论是系统内存还是JVM都没有受到2GB这样的限制。**

Tomcat的优化分成两块：

ü   Tomcat启动命令行中的优化参数即JVM优化

ü   Tomcat容器自身参数的优化（这块很像ApacheHttp Server）

这一节先要讲的是Tomcat启动命令行中的优化参数。

Tomcat首先跑在JVM之上的，因为它的启动其实也只是一个java命令行，首先我们需要对这个JAVA的启动命令行进行调优。

**需要注意的是：**

这边讨论的JVM优化是基于Oracle Sun的jdk1.6版有以上，其它JDK或者低版本JDK不适用。

**2.2 Tomcat启动行参数的优化**

Tomcat 的启动参数位于tomcat的安装目录\bin目录下，如果你是Linux操作系统就是catalina.sh文件，如果你是Windows操作系统那么 你需要改动的就是catalina.bat文件。打开该文件，一般该文件头部是一堆的由##包裹着的注释文字，找到注释文字的最后一段如：

|  |
| --- |
| # $Id: catalina.sh 522797 2007-03-27 07:10:29Z fhanik $  # -----------------------------------------------------------------------------    # OS specific support.  $var \_must\_ be set to either true or false. |

敲入一个回车，加入如下的参数

**Linux系统中tomcat的启动参数**

|  |
| --- |
| export JAVA\_OPTS="-server -Xms1400M -Xmx1400M -Xss512k -XX:+AggressiveOpts -XX:+UseBiasedLocking -XX:PermSize=128M -XX:MaxPermSize=256M -XX:+DisableExplicitGC -XX:MaxTenuringThreshold=31 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m  -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -Djava.awt.headless=true " |

**Windows系统中tomcat的启动参数**

|  |
| --- |
| set JAVA\_OPTS=-server -Xms1400M -Xmx1400M -Xss512k -XX:+AggressiveOpts -XX:+UseBiasedLocking -XX:PermSize=128M -XX:MaxPermSize=256M -XX:+DisableExplicitGC -XX:MaxTenuringThreshold=31 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m  -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -Djava.awt.headless=true |

上面参数好多啊，可能有人写到现在都没见一个tomcat的启动命令里加了这么多参数，当然，这些参数只是我机器上的，不一定适合你，尤其是参数后的value（值）是需要根据你自己的实际情况来设置的。

参数解释：

ü   -server

我不管你什么理由，只要你的tomcat是运行在生产环境中的，这个参数必须给我加上

因 为tomcat默认是以一种叫java –client的模式来运行的，server即意味着你的tomcat是以真实的production的模式在运行的，这也就意味着你的tomcat以 server模式运行时将拥有：更大、更高的并发处理能力，更快更强捷的JVM垃圾回收机制，可以获得更多的负载与吞吐量。。。更。。。还有更。。。

Y给我记住啊，如查这个-server都不加，那是要打屁股了。

ü   -Xms–Xmx

即JVM内存设置了，把Xms与Xmx两个值设成一样是最优的做法，有人说Xms为最小值，Xmx为最大值不是挺好的，这样设置还比较人性化，科学化。人性？科学？你个头啊。

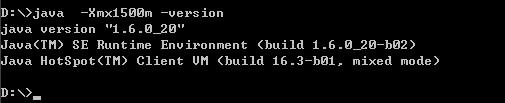
大家想一下这样的场景：

一 个系统随着并发数越来越高，它的内存使用情况逐步上升，上升到最高点不能上升了，开始回落，你们不要认为这个回落就是好事情，由其是大起大落，在内存回落 时它付出的代价是CPU高速开始运转进行垃圾回收，此时严重的甚至会造成你的系统出现“卡壳”就是你在好好的操作，突然网页像死在那边一样几秒甚至十几秒 时间，因为JVM正在进行垃圾回收。

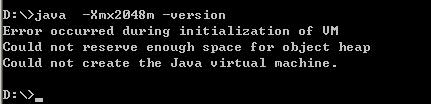
因此一开始我们就把这两个设成一样，使得Tomcat在启动时就为最大化参数充分利用系统的效率，这个道理和jdbcconnection pool里的minpool size与maxpool size的需要设成一个数量是一样的原理。

如何知道我的JVM能够使用最大值啊？拍脑袋？不行！

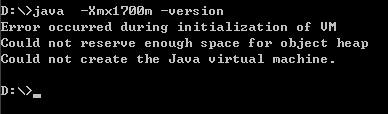
在设这个最大内存即Xmx值时请先打开一个命令行，键入如下的命令：



看，能够正常显示JDK的版本信息，说明，这个值你能够用。不是说32位系统下最高能够使用2GB内存吗？即：2048m，我们不防来试试



可以吗？不可以！不要说2048m呢，我们小一点，试试1700m如何



嘿嘿，连1700m都不可以，更不要说2048m了呢，2048m只是一个理论数值，这样说吧我这边有几台机器，有的机器-Xmx1800都没问题，有的机器最高只能到-Xmx1500m。

因此在设这个-Xms与-Xmx值时一定一定记得先这样测试一下，要不然直接加在tomcat启动命令行中你的tomcat就再也起不来了，要飞是飞不了，直接成了一只瘟猫了。

ü   –Xmn

设置年轻代大小为512m。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。

ü   -Xss

是指设定每个线程的堆栈大小。这个就要依据你的程序，看一个线程 大约需要占用多少内存，可能会有多少线程同时运行等。一般不易设置超过1M，要不然容易出现out ofmemory。

ü   -XX:+AggressiveOpts

作用如其名（aggressive），启用这个参数，则每当JDK版本升级时，你的JVM都会使用最新加入的优化技术（如果有的话）

ü   -XX:+UseBiasedLocking

启用一个优化了的线程锁，我们知道在我们的appserver，每个http请求就是一个线程，有的请求短有的请求长，就会有请求排队的现象，甚至还会出现线程阻塞，这个优化了的线程锁使得你的appserver内对线程处理自动进行最优调配。

ü   -XX:PermSize=128M-XX:MaxPermSize=256M

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；

在数据量的很大的文件导出时，一定要把这两个值设置上，否则会出现内存溢出的错误。

由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。

那么，如果是物理内存4GB，那么64分之一就是64MB，这就是PermSize默认值，也就是永生代内存初始大小；

四分之一是1024MB，这就是MaxPermSize默认大小。

ü   -XX:+DisableExplicitGC

在 程序代码中不允许有显示的调用”System.gc()”。看到过有两个极品工程中每次在DAO操作结束时手动调用System.gc()一下，觉得这样 做好像能够解决它们的out ofmemory问题一样，付出的代价就是系统响应时间严重降低，就和我在关于Xms,Xmx里的解释的原理一样，这样去调用GC导致系统的JVM大起大 落，性能不到什么地方去哟！

ü   -XX:+UseParNewGC

对年轻代采用多线程并行回收，这样收得快。

ü   -XX:+UseConcMarkSweepGC

即CMS gc，这一特性只有jdk1.5即后续版本才具有的功能，它使用的是gc估算触发和heap占用触发。

我们知道频频繁的GC会造面JVM的大起大落从而影响到系统的效率，因此使用了CMS GC后可以在GC次数增多的情况下，每次GC的响应时间却很短，比如说使用了CMS GC后经过jprofiler的观察，GC被触发次数非常多，而每次GC耗时仅为几毫秒。

ü   -XX:MaxTenuringThreshold

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率。

这个值的设置是根据本地的jprofiler监控后得到的一个理想的值，不能一概而论原搬照抄。

ü   -XX:+CMSParallelRemarkEnabled

在使用UseParNewGC 的情况下, 尽量减少 mark 的时间

ü   -XX:+UseCMSCompactAtFullCollection

在使用concurrent gc 的情况下, 防止 memoryfragmention, 对live object 进行整理, 使 memory 碎片减少。

ü   -XX:LargePageSizeInBytes

指定 Java heap的分页页面大小

ü   -XX:+UseFastAccessorMethods

get,set 方法转成本地代码

ü   -XX:+UseCMSInitiatingOccupancyOnly

指示只有在 oldgeneration 在使用了初始化的比例后concurrent collector 启动收集

ü   -XX:CMSInitiatingOccupancyFraction=70

CMSInitiatingOccupancyFraction，这个参数设置有很大技巧，基本上满足**(Xmx-Xmn)\*(100- CMSInitiatingOccupancyFraction)/100>=Xmn**就 不会出现promotion failed。在我的应用中Xmx是6000，Xmn是512，那么Xmx-Xmn是5488兆，也就是年老代有5488 兆，CMSInitiatingOccupancyFraction=90说明年老代到90%满的时候开始执行对年老代的并发垃圾回收（CMS），这时还 剩10%的空间是5488\*10%=548兆，所以即使Xmn（也就是年轻代共512兆）里所有对象都搬到年老代里，548兆的空间也足够了，所以只要满 足上面的公式，就不会出现垃圾回收时的promotion failed；

因此这个参数的设置必须与Xmn关联在一起。

ü   -Djava.awt.headless=true

这 个参数一般我们都是放在最后使用的，这全参数的作用是这样的，有时我们会在我们的J2EE工程中使用一些图表工具如：jfreechart，用于在web 网页输出GIF/JPG等流，在winodws环境下，一般我们的app server在输出图形时不会碰到什么问题，但是在linux/unix环境下经常会碰到一个exception导致你在winodws开发环境下图片显 示的好好可是在linux/unix下却显示不出来，因此加上这个参数以免避这样的情况出现。

上述这样的配置，基本上可以达到：

ü   系统响应时间增快

ü   JVM回收速度增快同时又不影响系统的响应率

ü   JVM内存最大化利用

ü   线程阻塞情况最小化

**2.3 Tomcat容器内的优化**

前面我们对Tomcat启动时的命令进行了优化，增加了系统的JVM可使用数、垃圾回收效率与线程阻塞情况、增加了系统响应效率等还有一个很重要的指标，我们没有去做优化，就是吞吐量。

还记得我们在第三天的学习中说的，这个系统本身可以处理1000，你没有优化和配置导致它默认只能处理25。因此下面我们来看Tomcat容器内的优化。

打开tomcat安装目录\conf\server.xml文件，定位到这一行：

|  |
| --- |
| <Connector port="8080" protocol="HTTP/1.1" |

这一行就是我们的tomcat容器性能参数设置的地方，它一般都会有一个默认值，这些默认值是远远不够我们的使用的，我们来看经过更改后的这一段的配置：

|  |
| --- |
| URIEncoding="UTF-8"  minSpareThreads="25" maxSpareThreads="75"            enableLookups="false" disableUploadTimeout="true" connectionTimeout="20000"            acceptCount="300"  maxThreads="300" maxProcessors="1000" minProcessors="5"            useURIValidationHack="false"                                                 compression="on" compressionMinSize="2048"                                                 compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"                 redirectPort="8443"  /> |

ü   URIEncoding=”UTF-8”

使得tomcat可以解析含有中文名的文件的url，真方便，不像apache里还有搞个mod\_encoding，还要手工编译

ü   maxSpareThreads

maxSpareThreads 的意思就是如果空闲状态的线程数多于设置的数目，则将这些线程中止，减少这个池中的线程总数。

ü   minSpareThreads

最小备用线程数，tomcat启动时的初始化的线程数。

ü   enableLookups

这个功效和Apache中的HostnameLookups一样，设为关闭。

ü   connectionTimeout

connectionTimeout为网络连接超时时间毫秒数。

ü   maxThreads

maxThreads Tomcat使用线程来处理接收的每个请求。这个值表示Tomcat可创建的最大的线程数，即最大并发数。

ü   acceptCount

acceptCount是当线程数达到maxThreads后，后续请求会被放入一个等待队列，这个acceptCount是这个队列的大小，如果这个队列也满了，就直接refuse connection

ü   maxProcessors与minProcessors

在 Java中线程是程序运行时的路径，是在一个程序中与其它控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出CPU最大利用率的高效程序，使空闲时间保持最低，从而接受更多的请求。

通常Windows是1000个左右，Linux是2000个左右。

ü   useURIValidationHack

我们来看一下tomcat中的一段源码：

|  |
| --- |
| security          if (connector.getUseURIValidationHack()) {              String uri = validate(request.getRequestURI());              if (uri == null) {                  res.setStatus(400);                  res.setMessage("Invalid URI");                  throw new IOException("Invalid URI");              } else {                  req.requestURI().setString(uri);                  // Redoing the URI decoding                  req.decodedURI().duplicate(req.requestURI());                  req.getURLDecoder().convert(req.decodedURI(), true);              }          } |

可以看到如果把useURIValidationHack设成"false"，可以减少它对一些url的不必要的检查从而减省开销。

ü   enableLookups="false"

为了消除DNS查询对性能的影响我们可以关闭DNS查询，方式是修改server.xml文件中的enableLookups参数值。

ü   disableUploadTimeout

类似于Apache中的keeyalive一样

ü   给Tomcat配置gzip压缩(HTTP压缩)功能

|  |
| --- |
| compression="on" compressionMinSize="2048"  compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain" |

HTTP 压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对 于普通的浏览过程HTML,CSS,Javascript , Text ，它可以节省40%左右的流量。更为重要的是，它可以对动态生成的，包括CGI、PHP , JSP , ASP , Servlet,SHTML等输出的网页也能进行压缩，压缩效率惊人。

1)compression="on" 打开压缩功能

2)compressionMinSize="2048" 启用压缩的输出内容大小，这里面默认为2KB

3)noCompressionUserAgents="gozilla, traviata" 对于以下的浏览器，不启用压缩

4)compressableMimeType="text/html,text/xml"　压缩类型

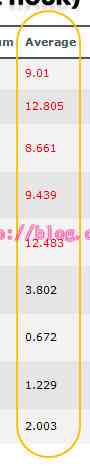
最后不要忘了把8443端口的地方也加上同样的配置，因为如果我们走https协议的话，我们将会用到8443端口这个段的配置，对吧？

|  |
| --- |
| URIEncoding="UTF-8"  minSpareThreads="25" maxSpareThreads="75"            enableLookups="false" disableUploadTimeout="true" connectionTimeout="20000"            acceptCount="300"  maxThreads="300" maxProcessors="1000" minProcessors="5"            useURIValidationHack="false"                      compression="on" compressionMinSize="2048"                      compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"                  SSLEnabled="true"             scheme="https" secure="true"             clientAuth="false" sslProtocol="TLS"             keystoreFile="d:/tomcat2/conf/shnlap93.jks" keystorePass="aaaaaa"        /> |

好了，所有的Tomcat优化的地方都加上了。结合第三天中的Apache的性能优化，我们这个架构可以“飞奔”起来了，当然这边把有提及任何关于数据库优化的步骤，但仅凭这两步，我们的系统已经有了很大的提升。

举个真实的例子：上一个项目，经过4轮performance testing，第一轮进行了问题的定位，第二轮就是进行了apache+tomcat/weblogic的优化，第三轮是做集群优化，第四轮是sql与codes的优化。

在到达第二轮时，我们的性能已经提升了多少倍呢？我们来看一个loaderrunner的截图吧：



左边第一列是第一轮没有经过任何调优的压力测试报告。

右边这一列是经过了apache优化，tomcat优化后得到的压力测试报告。

大家看看，这就提高了多少倍？这还只是在没有改动代码的情况下得到的改善，现在明白了好好的调优一

个apache和tomcat其实是多么的重要了？如果加上后面的代码、SQL的调优、数据库的调优。。。。。。所以我在上一个工程中有单笔交易性能（无论是吞吐量、响应时间）提高了80倍这样的极端例子的存在。