## 问题定位

对于Tomcat的处理耗时较长的问题主要有当时的并发量、session数、内存及内存的回收等几个方面造成的。出现问题之后就要进行分析了。   
  
**1.关于Tomcat的session数目**   
这个可以直接从Tomcat的web管理界面去查看即可   
或者借助于第三方工具Lambda Probe来查看，它相对于Tomcat自带的管理稍微多了点功能，但也不多   
  
**2.监视Tomcat的内存使用情况**   
使用JDK自带的jconsole可以比较明了的看到内存的使用情况，线程的状态，当前加载的类的总量等   
JDK自带的jvisualvm可以下载插件（如GC等），可以查看更丰富的信息。如果是分析本地的Tomcat的话，还可以进行内存抽样等，检查每个类的使用情况   
  
**3.打印类的加载情况及对象的回收情况**   
这个可以通过配置JVM的启动参数，打印这些信息（到屏幕（默认也会到catalina.log中）或者文件），具体参数如下：   
-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   
-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   
  
-verbose:class 监视加载的类的情况   
-verbose:gc 在虚拟机发生内存回收时在输出设备显示信息   
-verbose:jni 输出native方法调用的相关情况，一般用于诊断jni调用错误信息   
  
**4.添加JMS远程监控**   
对于部署在局域网内其它机器上的Tomcat，可以打开JMX监控端口，局域网其它机器就可以通过这个端口查看一些常用的参数（但一些比较复杂的功能不支持），同样是在JVM启动参数中配置即可，配置如下：   
-Dcom.sun.management.jmxremote.ssl=false  -Dcom.sun.management.jmxremote.authenticate=false   
-Djava.rmi.server.hostname=192.168.71.38 设置JVM的JMS监控监听的IP地址，主要是为了防止错误的监听成127.0.0.1这个内网地址   
-Dcom.sun.management.jmxremote.port=1090 设置JVM的JMS监控的端口   
-Dcom.sun.management.jmxremote.ssl=false 设置JVM的JMS监控不实用SSL   
-Dcom.sun.management.jmxremote.authenticate=false 设置JVM的JMS监控不需要认证   
  
  
**5.专业点的分析工具有**   
IBM ISA，JProfiler等，具体监控及分析方式去网上搜索即可。

# 集群方案

单个Tomcat的处理性能是有限的，当并发量较大的时候，就需要有部署多套来进行负载均衡了。   
  
集群的关键点有以下几点：   
**1.引入负载端**   
软负载可以使用nginx或者apache来进行，主要是使用一个分发的功能   
参考：   
http://ajita.iteye.com/blog/1715312（nginx负载）   
http://ajita.iteye.com/blog/1717121（apache负载）   
  
**2.共享session处理**   
目前的处理方式有如下几种：   
1).使用Tomcat本身的Session复制功能   
参考http://ajita.iteye.com/blog/1715312（Session复制的配置）   
方案的有点是配置简单，缺点是当集群数量较多时，Session复制的时间会比较长，影响响应的效率   
2).使用第三方来存放共享Session   
目前用的较多的是使用memcached来管理共享Session，借助于memcached-sesson-manager来进行Tomcat的Session管理   
参考http://ajita.iteye.com/blog/1716320（使用MSM管理Tomcat集群session）   
3).使用黏性session的策略   
对于会话要求不太强（不涉及到计费，失败了允许重新请求下等）的场合，同一个用户的session可以由nginx或者apache交给同一个Tomcat来处理，这就是所谓的session sticky策略，目前应用也比较多   
参考：http://ajita.iteye.com/blog/1848665（tomcat session sticky）   
nginx默认不包含session sticky模块，需要重新编译才行（windows下我也不知道怎么重新编译）   
优点是处理效率高多了，缺点是强会话要求的场合不合适   
  
**3.小结**   
以上是实现集群的要点，其中1和2可以组合使用，具体场景具体分析吧~

# JVM优化

Tomcat本身还是运行在JVM上的，通过对JVM参数的调整我们可以使Tomcat拥有更好的性能。针对JVM的优化目前主要在两个方面：   
  
**1.内存调优**   
内存方式的设置是在catalina.sh中，调整一下JAVA\_OPTS变量即可，因为后面的启动参数会把JAVA\_OPTS作为JVM的启动参数来处理。   
  
具体设置如下：   
JAVA\_OPTS="$JAVA\_OPTS -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4"   
  
其各项参数如下：   
-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。   
  
-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。   
  
  
**2.垃圾回收策略调优**   
垃圾回收的设置也是在catalina.sh中，调整JAVA\_OPTS变量。   
具体设置如下：   
JAVA\_OPTS="$JAVA\_OPTS -Xmx3550m -Xms3550m -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100"   
具体的垃圾回收策略及相应策略的各项参数如下：   
  
串行收集器（JDK1.5以前主要的回收方式）   
-XX:+UseSerialGC:设置串行收集器   
  
并行收集器（吞吐量优先）   
示例：   
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100   
  
-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。   
-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   
-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。   
-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。   
  
并发收集器（响应时间优先）   
示例：java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC   
-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。   
-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   
  
**3.小结**   
在内存设置中需要做一下权衡   
1)内存越大，一般情况下处理的效率也越高，但同时在做垃圾回收的时候所需要的时间也就越长，在这段时间内的处理效率是必然要受影响的。   
2)在大多数的网络文章中都推荐 Xmx和Xms设置为一致，说是避免频繁的回收，这个在测试的时候没有看到明显的效果，内存的占用情况基本都是锯齿状的效果，所以这个还要根据实际情况来定。

# Server.xml的Connection优化

Tomcat的Connector是Tomcat接收HTTP请求的关键模块，我们可以配置它来指定IO模式，以及处理通过这个Connector接受到的请求的处理线程数以及其它一些常用的HTTP策略。其主要配置参数如下：   
  
**1.指定使用NIO模型来接受HTTP请求**   
protocol="org.apache.coyote.http11.Http11NioProtocol" 指定使用NIO模型来接受HTTP请求。默认是BlockingIO，配置为protocol="HTTP/1.1"   
acceptorThreadCount="2" 使用NIO模型时接收线程的数目   
  
**2.指定使用线程池来处理HTTP请求**   
首先要配置一个线程池来处理请求（与Connector是平级的，多个Connector可以使用同一个线程池来处理请求）   
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"   
maxThreads="1000" minSpareThreads="50" maxIdleTime="600000"/>   
<Connector port="8080"   
executor="tomcatThreadPool" 指定使用的线程池   
  
**3.指定BlockingIO模式下的处理线程数目**   
maxThreads="150"//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。默认值200。可以根据机器的时期性能和内存大小调整，一般可以在400-500。最大可以在800左右。   
minSpareThreads="25"---Tomcat初始化时创建的线程数。默认值4。如果当前没有空闲线程，且没有超过maxThreads，一次性创建的空闲线程数量。Tomcat初始化时创建的线程数量也由此值设置。   
maxSpareThreads="75"--一旦创建的线程超过这个值，Tomcat就会关闭不再需要的socket线程。默认值50。一旦创建的线程超过此数值，Tomcat会关闭不再需要的线程。线程数可以大致上用 “同时在线人数\*每秒用户操作次数\*系统平均操作时间” 来计算。   
acceptCount="100"----指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。默认值10。如果当前可用线程数为0，则将请求放入处理队列中。这个值限定了请求队列的大小，超过这个数值的请求将不予处理。   
connectionTimeout="20000" --网络连接超时，默认值20000，单位：毫秒。设置为0表示永不超时，这样设置有隐患的。通常可设置为30000毫秒。   
  
**4.其它常用设置**   
maxHttpHeaderSize="8192" http请求头信息的最大程度，超过此长度的部分不予处理。一般8K。   
URIEncoding="UTF-8" 指定Tomcat容器的URL编码格式。   
disableUploadTimeout="true" 上传时是否使用超时机制   
enableLookups="false"--是否反查域名，默认值为true。为了提高处理能力，应设置为false   
compression="on"   打开压缩功能   
compressionMinSize="10240" 启用压缩的输出内容大小，默认为2KB   
noCompressionUserAgents="gozilla, traviata"   对于以下的浏览器，不启用压缩   
compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain" 哪些资源类型需要压缩   
  
**5.小结**   
关于Tomcat的Nio和ThreadPool，本身的引入就提高了处理的复杂性，所以对于效率的提高有多少，需要实际验证一下。   
  
**6.配置示例**   
<Connector port="8080"   
redirectPort="8443"   
maxThreads="150"   
minSpareThreads="25"   
maxSpareThreads="75"   
acceptCount="100"   
connectionTimeout="20000"   
protocol="HTTP/1.1"   
  
maxHttpHeaderSize="8192"   
URIEncoding="UTF-8"   
disableUploadTimeout="true"   
enableLookups="false"   
compression="on"   
compressionMinSize="10240"   
noCompressionUserAgents="gozilla, traviata"   
compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain">   
...   
</Connector>