**1、Tomcat的缺省端口是多少，怎么修改？**

**1）找到Tomcat目录下的conf文件夹**

**2）进入conf文件夹里面找到server.xml文件**

**3）打开server.xml文件**

**4）在server.xml文件里面找到下列信息**

**<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443" uriEncoding="utf-8"/>**

**port="8080"改成你想要的端口**

**2、tomcat 有哪几种Connector 运行模式(优化)？**

**bio：传统的Java I/O操作，同步且阻塞IO。**

**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。**

**nio：JDK1.4开始支持，同步阻塞或同步非阻塞IO。**

**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**

**aio(nio.2)：JDK7开始支持，异步非阻塞IO。**

**apr：Tomcat将以JNI的形式调用Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地 提高Tomcat对静态文件的处理性能。**

**<!--**

**<Connector connectionTimeout="20000" port="8000" protocol="HTTP/1.1" redirectPort="8443" uriEncoding="utf-8"/>**

**-->**

**<!-- protocol 启用 nio模式，(tomcat8默认使用的是nio)(apr模式利用系统级异步io) -->**

**<!-- minProcessors最小空闲连接线程数-->**

**<!-- maxProcessors最大连接线程数-->**

**<!-- acceptCount允许的最大连接数，应大于等于maxProcessors-->**

**<!-- enableLookups 如果为true,requst.getRemoteHost会执行DNS查找，反向解析ip对应域名或主机名-->**

**<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"**

**connectionTimeout="20000"**

**redirectPort="8443**

**maxThreads=“500”**

**minSpareThreads=“100”**

**maxSpareThreads=“200”**

**acceptCount="200"**

**enableLookups="false"**

**/>**

**其他配置**

**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**

**3、Tomcat有几种部署方式？**

**1）直接把Web项目放在webapps下，Tomcat会自动将其部署**

**2）在server.xml文件上配置<Context>节点，设置相关的属性即可**

**3）通过Catalina来进行配置:进入到conf\Catalina\localhost文件下，创建一个xml文件，该文件的名字就是站点的名字。**

**编写XML的方式来进行设置。**

**4、tomcat容器是如何创建servlet类实例？用到了什么原理？**

**当容器启动时，会读取在webapps目录下所有的web应用中的web.xml文件，然后对xml文件进行解析，**

**并读取servlet注册信息。然后，将每个应用中注册的servlet类都进行加载，并通过反射的方式实例化。**

**（有时候也是在第一次请求时实例化）在servlet注册时加上如果为正数，则在一开始就实例化，**

**如果不写或为负数，则第一次请求实例化。**

**5.tomcat 如何优化？**

**1、优化连接配置.这里以tomcat7的参数配置为例，需要修改conf/server.xml文件，修改连接数，关闭客户端dns查询。**

**参数解释：**

**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**

**maxSpareThreads : 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。**

**minSpareThreads : 最小备用线程数，tomcat启动时的初始化的线程数。**

**enableLookups : 这个功效和Apache中的HostnameLookups一样，设为关闭。**

**connectionTimeout : connectionTimeout为网络连接超时时间毫秒数。**

**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。**

**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**

**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。**

**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。**

**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。**

**disableUploadTimeout ：类似于Apache中的keeyalive一样**

**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？**

**<!–enable tomcat ssl–>**

**<Connector port=”8443″ protocol=”HTTP/1.1″**

**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。**

**6.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。**

**7.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**

**8.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**

**8.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**

**9.专业点的分析工具有**

**IBM ISA，JProfiler、probe 等，具体监控及分析方式去网上搜索即可**

**10.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 ；**

**11.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**

**12.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**

**13.Tomcat一个请求的完整过程**

**Ng:(nginx)**

**upstream yy\_001{**

**server 10.99.99.99:8080;**

**server 10.99.99.100:8080;**

**hash $\*\*;**

**healthcheck\_enabled;**

**healthcheck\_delay 3000;**

**healthcheck\_timeout 1000;**

**healthcheck\_failcount 2;**

**healthcheck\_send 'GET /healthcheck.html HTTP/1.0' 'Host: wo.com' 'Connection: close';**

**}**

**server {**

**include base.conf;**

**server\_name wo.de.tian;**

**...**

**location /yy/ {**

**proxy\_pass http://yy\_001;**

**}**

**首先 dns 解析 wo.de.tian机器，一般是ng服务器ip地址**

**然后 ng根据server的配置，寻找路径为 yy/的机器列表，ip和端口**

**最后 选择其中一台机器进行访问—->下面为详细过程**

**1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得**

**2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应**

**3) Engine获得请求localhost/yy/index.jsp，匹配它所拥有的所有虚拟主机Host**

**4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）**

**5) localhost Host获得请求/yy/index.jsp，匹配它所拥有的所有Context**

**6) Host匹配到路径为/yy的Context（如果匹配不到就把该请求交给路径名为”“的Context去处理）**

**7) path=”/yy”的Context获得请求/index.jsp，在它的mapping table中寻找对应的servlet**

**8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类**

**9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法**

**10)Context把执行完了之后的HttpServletResponse对象返回给Host**

**11)Host把HttpServletResponse对象返回给Engine**

**12)Engine把HttpServletResponse对象返回给Connector**

**13)Connector把HttpServletResponse对象返回给客户browser**

**14.Tomcat工作模式？**

**Tomcat是一个JSP/Servlet容器。其作为Servlet容器，有三种工作模式：独立的Servlet容器、进程内的Servlet容器和进程外的Servlet容器。**

**进入Tomcat的请求可以根据Tomcat的工作模式分为如下两类：**

**Tomcat作为应用程序服务器：请求来自于前端的web服务器，这可能是Apache, IIS, Nginx等；**

**Tomcat作为独立服务器：请求来自于web浏览器；**