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；