**中间件**

## 一、中间件概述

**1 Middleware**

中间件(Middleware)是一种软件，处于系统软件与应用软件之间，它能使处于应用层中的各应用成分之间实现跨网络的协同工作（互操作），这时允许各应用软件之下所涉及的系统结构、操作系统、通信协议、数据库和其它应用服务各不相同。分成数据访问中间件，远程过程调用中间件，消息中间件，应用服务器中间件。

**2 中间件基本框架**

第四层：应用接口（API）层

第三层：中间件服务层（通信，控制，计算，管理，输出，信息）

第二层：应用平台层（Unix、NetWare、NT、VAX、OPENVMS等）

第一层：传输协议层（TCP/IP、IPX/SPX、NetBIOS、OSI等）

## 二、tomcat概述

**1 Tomcat**

Tomcat是一个JSP/Servlet容器。

**2 目录作用**

|---bin：存放各种平台下启动和关闭Tomcat的脚本文件.有个档是catalina.bat,

打开这个windos配置文件,在非注释行加入JDK路径,例如 : SET JAVA\_HOME=C:\j2sdk1.4.2\_06 保存后,就配置好tomcat环境了.

　　|---conf：配置文件

　　|---doc：存放Tomcat文档

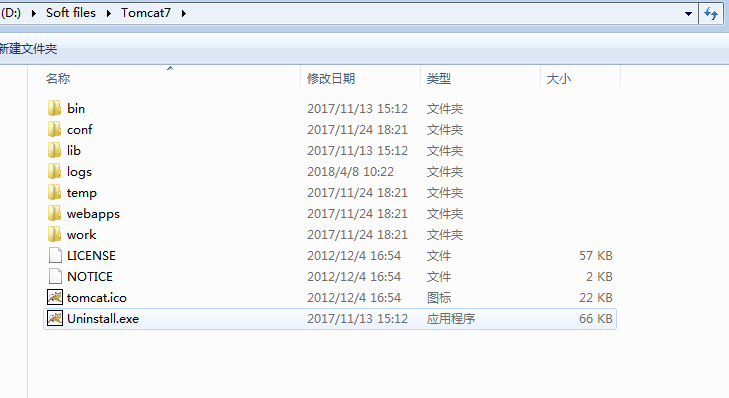
　　|---lib/japser/common：存放Tomcat运行需要的库文件（JARS）

　　|---logs：存放Tomcat执行时的LOG文件

　　|---src：存放Tomcat的源代码

　　|---webapps：Tomcat的主要Web发布目录（包括应用程序示例）

|---work：存放jsp编译后产生的class文件



**3 Tomcat配置文件：**

　server.xml: Tomcat的主配置文件，包含Service, Connector, Engine, Realm, Valve, Hosts主组件的相关配置信息；

　web.xml：遵循Servlet规范标准的配置文件，用于配置servlet，并为所有的Web应用程序提供包括MIME映射等默认配置信息；

　tomcat-user.xml：Realm认证时用到的相关角色、用户和密码等信息；Tomcat自带的manager默认情况下会用到此文件；在Tomcat中添加/删除用户，为用户　　指定角色等将通过编辑此文件实现；

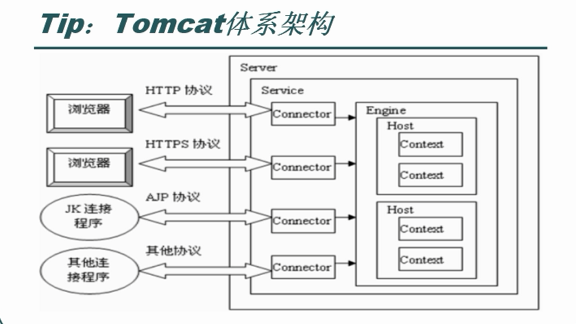
　catalina.policy：Java相关的安全策略配置文件，在系统资源级别上提供访问控制的能力；

　catalina.properties：Tomcat内部package的定义及访问相关控制，也包括对通过类装载器装载的内容的控制；Tomcat在启动时会事先读取此文件的相关设置；

　logging.properties: Tomcat6通过自己内部实现的JAVA日志记录器来记录操作相关的日志，此文件即为日志记录器相关的配置信息，可以用来定义日志记录的组　　件级别以及日志文件的存在位置等；

　context.xml：所有host的默认配置信息；

## 4 架构及常用的组件



**（1）Server组件**

如上面示例文件中定义的：

<Server port=”8005” shutdown=”SHUTDOWN”>

这会让Tomcat6启动一个server实例（即一个JVM），它监听在8005端口以接收shutdown命令，使用 telnet 连接8005 端口可以直接执行 SHUTDOWN 命令来关闭 Tomcat。各Server的定义不能使用同一个端口，这意味着如果在同一个物理机上启动了多个Server实例，必须配置它们使用不同的端口。这个端口的定义用于为管理员提供一个关闭此实例的便捷途径，因此，管理员可以直接telnet至此端口使用SHUTDOWN命令关闭此实例。不过，基于安全角度的考虑，这通常不允许远程进行。

Server的相关属性：

className: 用于实现此Server容器的完全限定类的名称，默认为org.apache.catalina.core.StandardServer；

port: 接收shutdown指令的端口，默认仅允许通过本机访问，默认为8005；

shutdown：发往此Server用于实现关闭tomcat实例的命令字符串，默认为SHUTDOWN；

**（2）Service组件：**

Service主要用于关联一个引擎和与此引擎相关的连接器，每个连接器通过一个特定的端口和协议接收入站请求交将其转发至关联的引擎进行处理。困此，Service要包含一个引擎、一个或多个连接器。

如上面示例中的定义：

<Service name=”Catalina”>

这定义了一个名为Catalina的Service，此名字也会在产生相关的日志信息时记录在日志文件当中。

Service相关的属性：

className： 用于实现service的类名，一般都是org.apache.catalina.core.StandardService。

name：此服务的名称，默认为Catalina；

**（3）Connector组件：**

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；

Tomcat应该考虑工作情形并为相应情形下的请求分别定义好需要的连接器才能正确接收来自于客户端的请求。一个引擎可以有一个或多个连接器，以适应多种请求方式。

定义连接器可以使用多种属性，有些属性也只适用于某特定的连接器类型。一般说来，常见于server.xml中的连接器类型通常有4种：

1. **HTTP连接器 2) SSL连接器 3) AJP 1.3连接器 4) proxy连接器**

如上面示例server.xml中定义的HTTP连接器：

<Connector port=”8080″ protocol=”HTTP/1.1″

maxThreads=”150″ connectionTimeout=”20000″

redirectPort=”8443″/>

定义连接器时可以配置的属性非常多，但通常定义HTTP连接器时必须定义的属性只有“port“，定义AJP连接器时必须定义的属性只有”protocol”，因为默认的协议为HTTP。以下为常用属性的说明：

1) address：指定连接器监听的地址，默认为所有地址，即0.0.0.0； 可以自己指定地，如

2) maxThreads：支持的最大并发连接数，默认为200；

3) port：监听的端口，默认为0；

4) protocol：连接器使用的协议，默认为HTTP/1.1，定义AJP协议时通常为AJP/1.3；

5) redirectPort：如果某连接器支持的协议是HTTP，当接收客户端发来的HTTPS请求时，则转发至此属性定义的端口；

6) connectionTimeout：等待客户端发送请求的超时时间，单位为毫秒，默认为60000，即1分钟；

7) enableLookups：是否通过request.getRemoteHost()进行DNS查询以获取客户端的主机名；默认为true； 进行反解的，可以设置为false

8) acceptCount：设置等待队列的最大长度；通常在tomcat所有处理线程均处于繁忙状态时，新发来的请求将被放置于等待队列中；

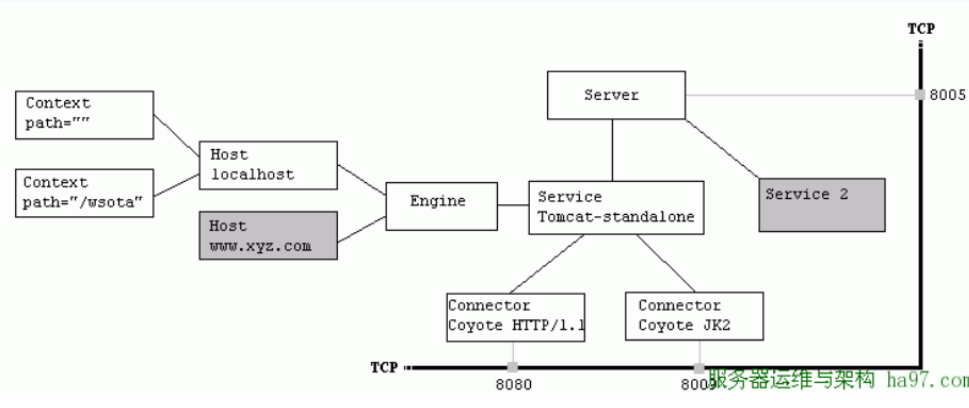
下面是一个定义了多个属性的SSL连接器：

<Connector port=”8443″

maxThreads=”150″ minSpareThreads=”25″ maxSpareThreads=”75″

enableLookups=”false” acceptCount=”100″ debug=”0″ scheme=”https” secure=”true”

clientAuth=”false” sslProtocol=”TLS” />



**（4）Engine组件：**

Engine是Servlet处理器的一个实例，即servlet引擎，默认为定义在server.xml中的Catalina。Engine需要defaultHost属性来为其定义一个接收所有发往非明确定义虚拟主机的请求的host组件。如前面示例中定义的：

<Engine name=”Catalina” defaultHost=”localhost”>

defaultHost：Tomcat支持基于FQDN的虚拟主机，这些虚拟主机可以通过在Engine容器中定义多个不同的Host组件来实现；但如果此引擎的连接器收到一个发往非非明确定义虚拟主机的请求时则需要将此请求发往一个默认的虚拟主机进行处理，因此，在Engine中定义的多个虚拟主机的主机名称中至少要有一个跟defaultHost定义的主机名称同名；

name：Engine组件的名称，用于日志和错误信息记录时区别不同的引擎；

Engine容器中可以包含Realm、Host、Listener和Valve子容器。

**（5）Host组件：**

位于Engine容器中用于接收请求并进行相应处理的主机或虚拟主机，如前面示例中的定义：

<Host name=”localhost” appBase=”webapps”

unpackWARs=”true” autoDeploy=”true”

xmlValidation=”false” xmlNamespaceAware=”false”>

</Host>

常用属性说明：

1) appBase：此Host的webapps目录，即存放非归档的web应用程序的目录或归档后的WAR文件的目录路径；可以使用基于$CATALINA\_HOME的相对路径；

2) autoDeploy：在Tomcat处于运行状态时放置于appBase目录中的应用程序文件是否自动进行deploy；默认为true；

3) unpackWars：在启用此webapps时是否对WAR格式的归档文件先进行展开；默认为true；

**（6）Context组件：**

Context在某些意义上类似于apache中的路径别名，一个Context定义用于标识tomcat实例中的一个Web应用程序；如下面的定义：

<Context path=”” docBase=”/web/webapps”/>

<Context path=”/bbs” docBase=”/web/threads/bbs” reloadable=”true”/>

<Context path=”/chat” docBase=”/web/chat”/>

<Context path=”/darian” docBase=”darian”/>

在tomcat 5.5之前Context体现在/conf/server.xml中的<Host>里的<Context>元素，它由Context接口定义。每个<Context>元素代表了运行在虚拟主机上的单个Web应用。

在tomcat 5.5之后不推荐在server.xml中进行配置，而是在/conf/context.xml中进行独立的配置。

因 为 server.xml 是不可动态重加载的资源，服务器一旦启动了以后，要修改这个文件，就得重启服务器才能重新加载。而 context.xml 文件则不然， tomcat 服务器会定时去扫描这个文件。一旦发现文件被修改（时间戳改变了），就会自动重新加载这个文件，而不需要重启服务器 。

context.xml

<Context path="/kaka" docBase="kaka" debug="0" reloadbale="true" privileged="true">

<WatchedResource>WEB-INF/web.xml</WatchedResource>

<WatchedResource>WEB-INF/kaka.xml</WatchedResource> 监控资源文件，如果web.xml || kaka.xml改变了，则自动重新加载改应用。

<Resource name="jdbc/testSiteds" 表示指定的jndi名称

auth="Container" 表示认证方式，一般为Container

type="javax.sql.DataSource"

maxActive="100" 连接池支持的最大连接数

maxIdle="30" 连接池中最多可空闲maxIdle个连接

maxWait="10000" 连接池中连接用完时,新的请求等待时间,毫秒

username="root" 表示数据库用户名

password="root" 表示数据库用户的密码

driverClassName="com.mysql.jdbc.Driver" 表示JDBC DRIVER

url="jdbc:mysql://localhost:3306/testSite" /> 表示数据库URL地址

</Context>

**context.xml的三个作用范围**

① tomcat server级别：

在/conf/context.xml里配置。（**因为这个contex.xml是每个webapp都会读取的，所以在这个文件里面定义的节点都是全局性的，即每个webapp都会出现**）

② Host级别：（有多个虚拟主机的时候才会用到）

在/conf/Catalina/${hostName}里添加context.xml，继而进行配置

③ web app 级别：（这个context.xml是对应各自特定webapp的，属于webapp内部）

在/conf/Catalina/${hostName}里添加${webAppName}.xml，继而进行配置（这个Context）

**（7）Realm组件：**

一个Realm表示一个安全上下文，它是一个授权访问某个给定Context的用户列表和某用户所允许切换的角色相关定义的列表。因此，Realm就像是一个用户和组相关的数据库。定义Realm时惟一必须要提供的属性是classname，它是Realm的多个不同实现，用于表示此Realm认证的用户及角色等认证信息的存放位置。

JAASRealm：基于Java Authintication and Authorization Service实现用户认证；

JDBCRealm：通过JDBC访问某关系型数据库表实现用户认证；

JNDIRealm：基于JNDI使用目录服务实现认证信息的获取；

MemoryRealm：查找tomcat-user.xml文件实现用户信息的获取；

UserDatabaseRealm：基于UserDatabase文件(通常是tomcat-user.xml)实现用户认证，它实现是一个完全可更新和持久有效的MemoryRealm，因此能够跟标准的MemoryRealm兼容；它通过JNDI实现；

下面是一个常见的使用UserDatabase的配置：

<Realm className=”org.apache.catalina.realm.UserDatabaseRealm”

resourceName=”UserDatabase”/>

下面是一个使用JDBC方式获取用户认证信息的配置：

<Realm className=”org.apache.catalina.realm.JDBCRealm” debug=”99″

driverName=”org.gjt.mm.mysql.Driver”

connectionURL=”jdbc:mysql://localhost/authority”

connectionName=”test” connectionPassword=”test”

userTable=”users” userNameCol=”user\_name”

userCredCol=”user\_pass”

userRoleTable=”user\_roles” roleNameCol=”role\_name” />

**（8）Valve组件：**

Valve类似于过滤器，它可以工作于Engine和Host/Context之间、Host和Context之间以及Context和Web应用程序的某资源之间。一个容器内可以建立多个Valve，而且Valve定义的次序也决定了它们生效的次序。Tomcat6中实现了多种不同的Valve：

AccessLogValve：访问日志Valve

ExtendedAccessValve：扩展功能的访问日志Valve

JDBCAccessLogValve：通过JDBC将访问日志信息发送到数据库中；

RequestDumperValve：请求转储Valve；

RemoteAddrValve：基于远程地址的访问控制；

RemoteHostValve：基于远程主机名称的访问控制；

SemaphoreValve：用于控制Tomcat主机上任何容器上的并发访问数量；

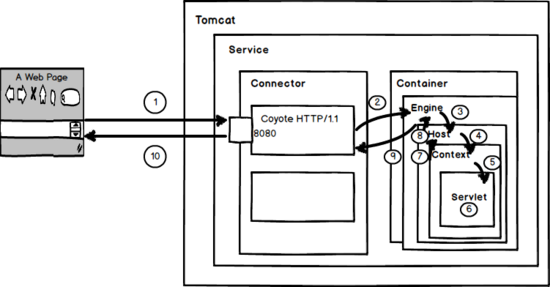
<Context path=”/probe” docBase=”probe”>

<Valve className=”org.apache.catalina.valves.RemoteAddrValve”

allow=”127\.0\.0\.1″/>

</Context>

### 5 请求过程



**假设来自客户的请求为：http://localhost:8080/wsota/wsota\_index.jsp**

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Connector的主要任务是负责接收浏览器的发过来的 tcp 连接请求，创建一个 Request 和 Response 对象分别用于和请求端交换数据，然后会产生一个线程来处理这个请求并把产生的 Request 和 Response 对象传给处理这个请求的线程  
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/wsota/wsota\_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/wsota/wsota\_index.jsp，匹配它所拥有的所有Context  
6) Host匹配到路径为/wsota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=”/wsota”的Context获得请求/wsota\_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

**一个Context对应于一个Web App，每个Web App是由一个或者多个servlet组成的。**

**当一个Web App被初始化的时候，便会为这个webapp创建一个context对象，并把这个context对象注册到指定虚拟主机（host）上，**

**接着，它将用 自己的ClassLoader对象载入部署配置文件web.xml中定义的每个servlet类。它首先载入在$CATALINA\_HOME/conf/web.xml中部署的servlet类，**

**然后载入在自己的Web App根目录下的WEB-INF/web.xml中部署的servlet类。web.xml文件有两部分：servlet类定义和servlet映射定义**

**每个被载入的servlet类都有一个名字，且被填入该Context的映射表(mapping table)中，和某种URL PATTERN对应。当该Context获得请求时，将查询mapping table，找到被请求的servlet，并执行以获得请求回应**

### 6 虚拟主机



# 7 三种运行模式

**1）、bio(blocking I/O)**

阻塞式I/O操作，表示Tomcat使用的是传统的JavaI/O操作(即java.io包及其子包)。

bio模式是三种运行模式中性能最低的一种。**一个线程处理一个请求，缺点：并发量高时，线程数较多**，浪费资源

**可以通过http://localhost:8080/manager/status 查看**

server.xml 文件中的配置

<Connector port="9090" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

**2）、nio(new I/O)**

Java nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看成是non-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能。

**利用 Java 的异步请求 IO 处理，可以通过少量的线程处理大量的请求**

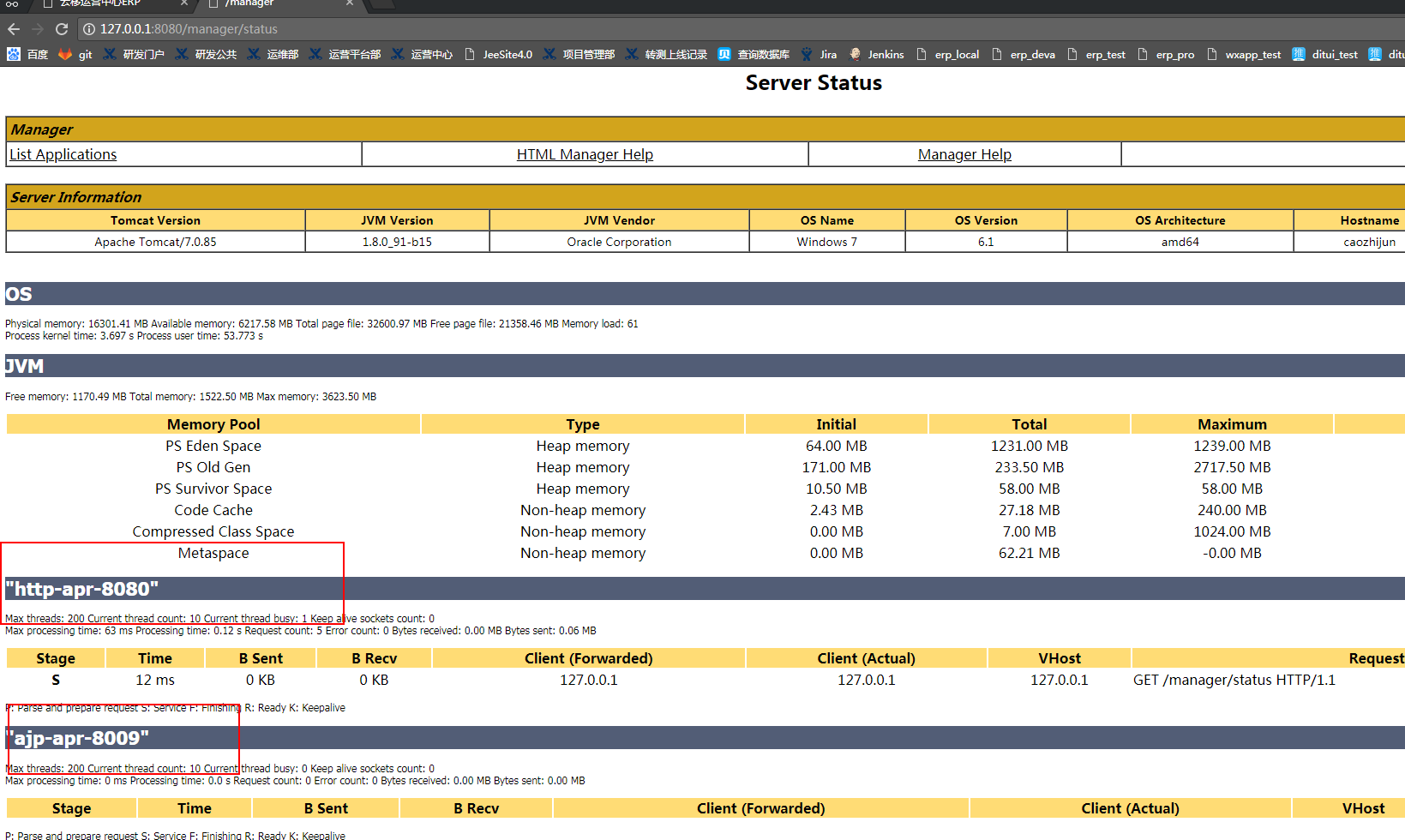
注意： Tomcat8 以上版本在 Linux 系统中，默认使用的就是NIO模式，不需要额外修改 ，Tomcat7必须修改Connector配置来启动

可以通过http://localhost:8080/manager/status 查看

<Connector port="9090" protocol="org.apache.coyote.http11.Http11NioProtocol"

connectionTimeout="20000"

redirectPort="8443" />



**3）、apr(Apache Portable Runtime/Apache可移植运行时)**

安装配置过程相对复杂，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。

Tomcat apr也是在Tomcat上运行高并发应用的首选模式。**从操作系统级别来解决异步的IO问题。**APR是使用原生C语言编写的非堵塞I/O，利用了操作系统的网络连接功能，速度很快。但是需先安装apr和native，若直接启动就支持apr，能大幅度提升性能，不亚于魔兽开局爆高科技兵种，威力强大

Tomcat apr的配置需要以下三个组件的支持：

APR library：APR下载http://apr.apache.org/download.cgi make install

tomcat-native：

OpenSSL libraries

与配置nio运行模式一样，也需要将对应Connector的protocol属性值改为 org.apache.coyote.http11.Http11AprProtocol

server.xml 文件中的配置

<Connector port="8080" protocol="org.apache.coyote.http11.Http11AprProtocol"

URIEncoding="UTF-8"

maxConnections="10000"

maxThreads="2000"

acceptCount="2000"

minSpareThreads="100"

compression="on"

compressionMinSize="2048"

compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"

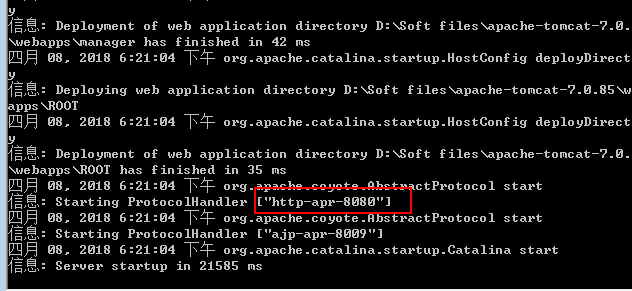
enableLookups="false"

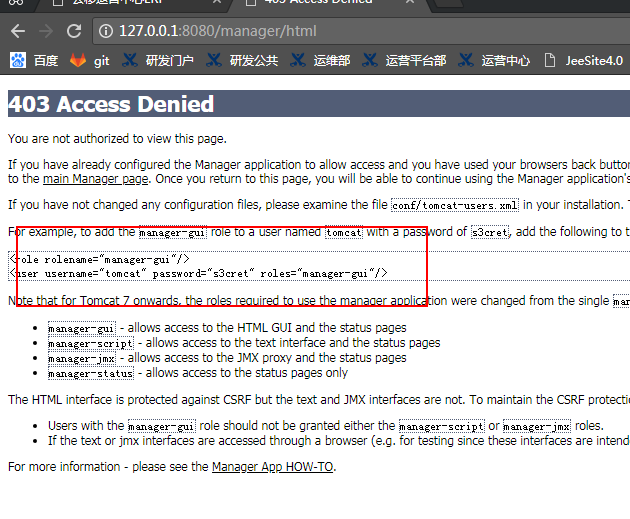
disableUploadTimeout="true"

connectionTimeout="20000"

redirectPort="8443" />

**Tomcat 7.0.85默认了APR的方式，效率最高**





# 9 三种工作模式

（1）独立的Servlet容器，在这种模式下，Tomcat可以作为独立的Java Web服务器，Servlet容器作为构成Web服务器的一部分而存在。独立的Servlet容器是Tomcat的默认模式。

（2）进程内的Servlet容器，Servlet容器分为Web服务器插件和Java容器两部分。Web服务器插件在其它Web服务器内部地址空间打开一个Java虚拟机（JVM），Java容器在此JVM中运行Servlet。如有客户端发出调用Servlet的请求，插件获得对此请求的控制并将它传递（使用JNI通信机制）给Java容器。进程内Servlet容器对于单进程、多线程的服务器非常合适，可以提供较高的运行速度，但是缺乏伸缩性。

（3）进程外的Servlet容器，Servlet容器分为Web服务器插件和Java容器两部分。Web服务器插件在其它Web服务器外部地址空间打开一个JVM，Java容器在此JVM中运行Servlet。如有客户端发出调用Servlet的请求，插件获得对此请求的控制并将它传递（使用IPC通信机制）给Java容器。进程外Servlet容器对于客户请求的响应速度不如进程内容器，但进程外容器具有更好的伸缩性和稳定性。

**10 Tomcat调优**

**1） 服务器资源优化**

服务器所能提供CPU、内存、硬盘的性能对处理能力有决定性影响，所以说服务器性能牛B，Tomcat也不会太差。当然提高服务器的硬件配置，是需要大量RMB的支持的。

所以不到万不得已不会采用这种方式，一般公司会采取下面这种通过优化配置，来提升Tomcat性能的方式。

**2） 运行模式 优化**

bio：默认的模式,性能非常低下,没有经过任何优化处理和支持.

nio：nio(new I/O)，是Java SE 1.4及后续版本提供的一种新的I/O操作方式(即Java.nio包及其子包)。java nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看成是non-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能。

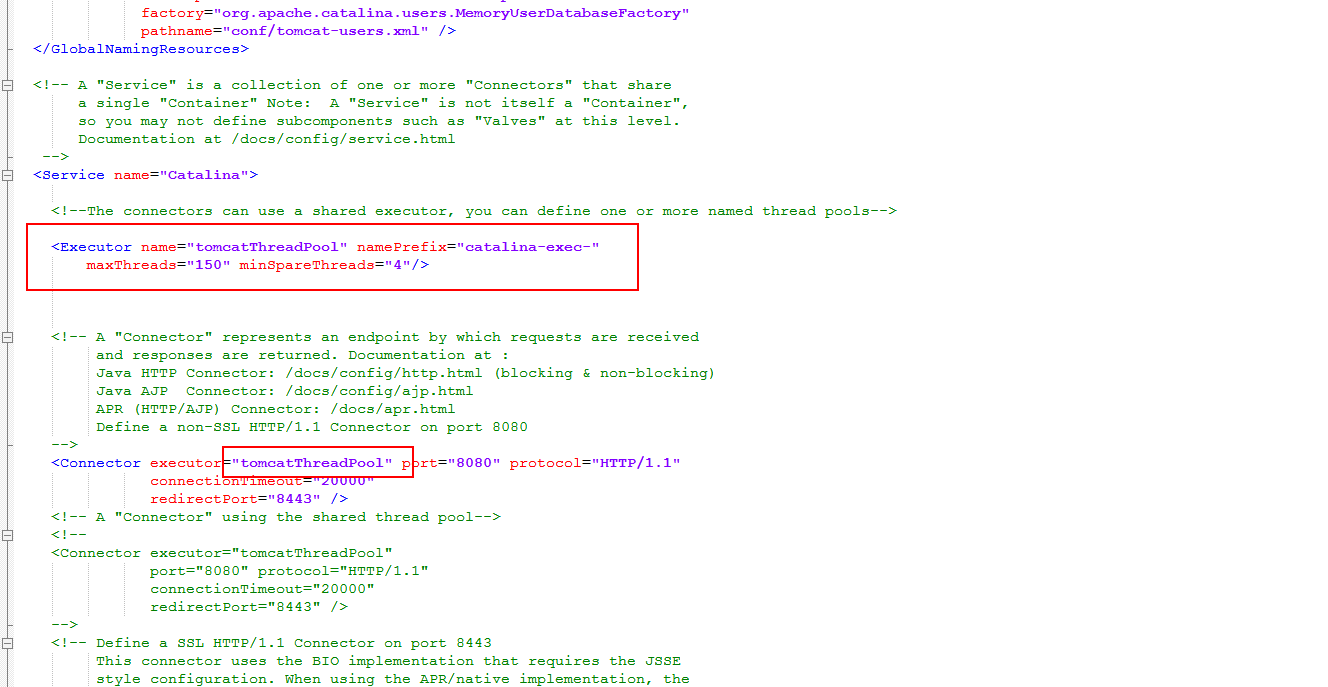
apr：安装起来最困难,但是从操作系统级别来解决异步的IO问题,大幅度的提高性能.

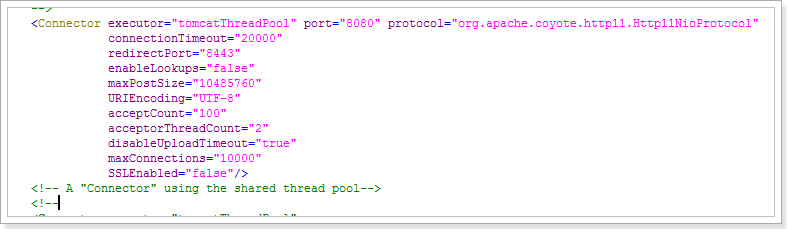
启动NIO模式，修改server.xml里的Connector节点,

修改protocol为org.apache.coyote.http11.Http11NioProtocol

**3） 执行器线程池优化**

在tomcat中每一个用户请求都是一个线程，所以可以使用线程池提高性能。





**11 Executor重要参数说明：**

name：共享线程池的名字。这是Connector为了共享线程池要引用的名字，该名字必须唯一。

namePrefix:在JVM上，每个运行线程都可以有一个name 字符串。这一属性为线程池中每个线程的name字符串设置了一个前缀，Tomcat将把线程号追加到这一前缀的后面。默认值：tomcat-exec-；

maxThreads：该线程池可以容纳的最大线程数。默认值：200；

maxIdleTime：在Tomcat关闭一个空闲线程之前，允许空闲线程持续的时间(以毫秒为单位)。只有当前活跃的线程数大于minSpareThread的值，才会关闭空闲线程。默认值：60000(一分钟)。

minSpareThreads：Tomcat应该始终打开的最小不活跃线程数。默认值：25。

threadPriority：线程的等级。默认是Thread.NORM\_PRIORITY

**12 Connector重要参数说明：**

executor：表示使用该参数值对应的线程池；

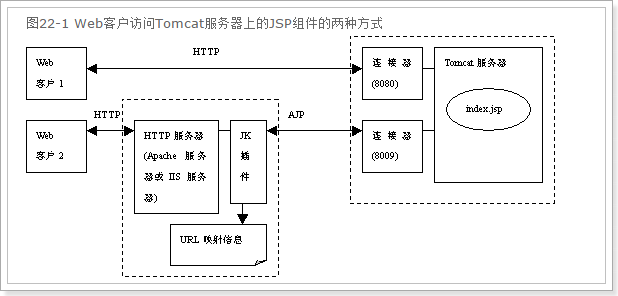
minProcessors：服务器启动时创建的处理请求的线程数；

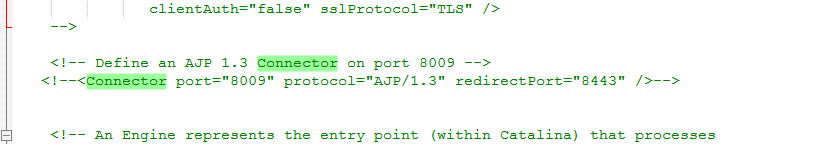
maxProcessors：最大可以创建的处理请求的线程数；

acceptCount：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。

**14 禁用AJP连接器**

AJPv13协议是面向包的；WEB服务器会尝试维护一个永久TCP连接到servlet容器，并且在多个请求和响应周期过程会重用连接。我们一般是使用Nginx+tomcat的架构，所以用不着AJP协议，所以把AJP连接器禁用。在管理页面看不到AJP了。





**15 jvm内存优化**

Tomcat内存优化主要是对 tomcat 启动参数优化，我们可以在 tomcat 的启动脚本 catalina.sh 中设置 java\_OPTS 参数。

　　JAVA\_OPTS参数说明

　　-server 启用jdk 的 server 版；

　　-Xms java虚拟机初始化时的最小内存；

　　-Xmx java虚拟机可使用的最大内存；

　　-XX: PermSize 内存永久保留区域

　　-XX:MaxPermSize 内存最大永久保留区域

　　服务器参数配置

　　现公司服务器内存一般都可以加到最大2G ，所以可以采取以下配置：

　　JAVA\_OPTS=’-Xms1024m -Xmx2048m -XX: PermSize=256M -XX:MaxNewSize=256m -XX:MaxPermSize=256m’

　　配置完成后可重启Tomcat ，通过以下命令进行查看配置是否生效：

　　首先查看Tomcat 进程号：

　　sudo lsof -i:9027

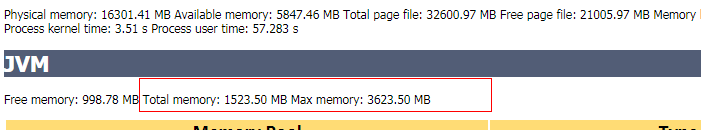
　　我们可以看到Tomcat 进程号是 12222 。

　　查看是否配置生效：

　　sudo jmap – heap 12222

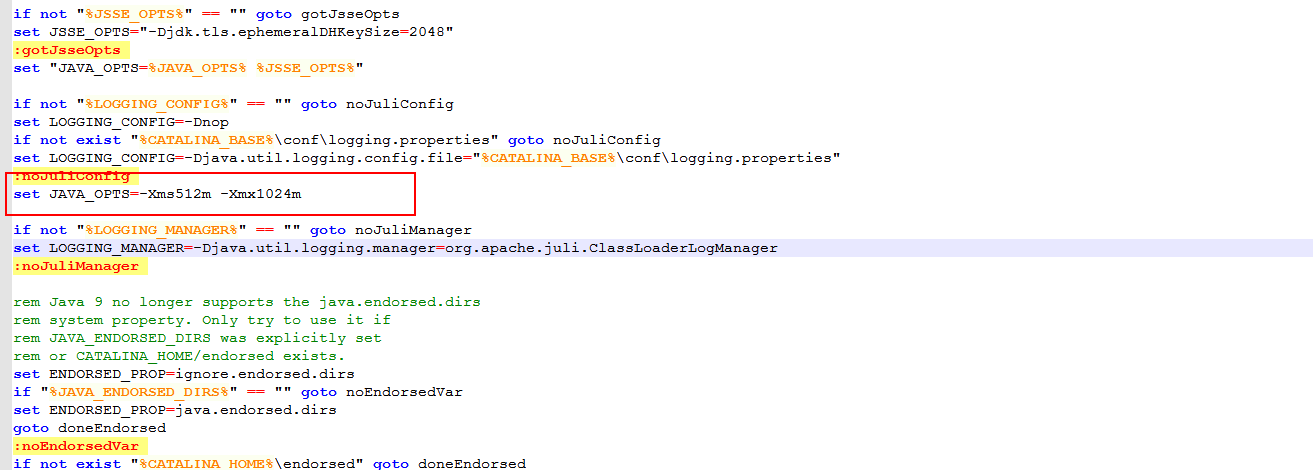
　　我们可以看到MaxHeapSize 等参数已经生效。

**修改之前：**

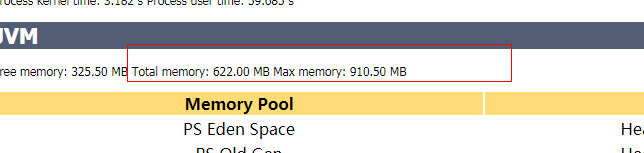


**D:\Soft files\apache-tomcat-7.0.85-01\bin\catalina.bat**

修改中：



**修改之后：**



## 三、Jboss

**1 JBoss完全实现了J2EE的服务栈：**

EJB (Enterprise JavaBeans)

JMS (Java Message Service)

JTS/JTA (Java Transaction Service / Java Transaction API)

Servlet and JSP (JavaServer Pages)

JNDI (Java Naming and Directory Interface)

它还提供一些高级特性，比如集群、JMX、Web Service。它还整合了IIOP（Internet Inter-ORB Protocol）。

因为JBoss代码遵循LGPL许可，你可以在任何商业应用中免费使用它，而不用支付费用。

**2 JBoss中的部署过程非常的简单、直接。**在每一个配置中，JBoss不断的扫描一个特殊的目录的变化server/config-name/deploy此目录一般被称为“部署目录”。

你可以把下列文件拷贝到此目录下：

任何jar库（其中的类将被自动添加到JBoss的classpath中）

EJB JAR

WAR (Web Appliction aRrchive)

EAR (Enterprise Application aRchive)

**3 JBoss的群集的目的是提供：负载平衡（Load Balance）；高可用（High Availablity）**

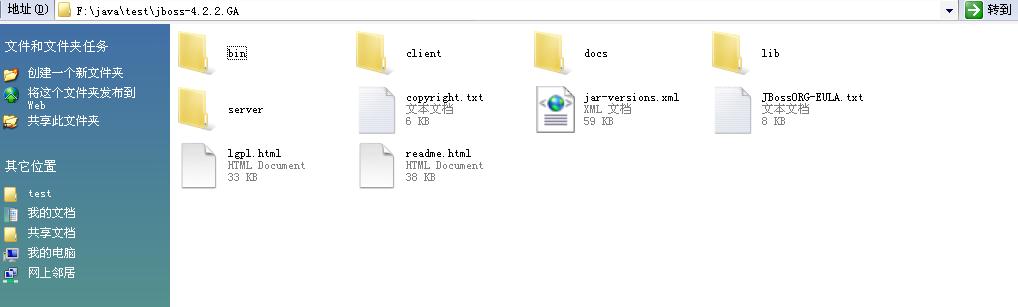
而域的目的则是将多台服务器组成一个服务器组（Server Group），并为一个服务器组内的多台主机（Host）提供：

单点集中配置（通过一个域控制器，即Domain Controller，实现组内主机的统一配置）

单点统一部署，通过域控制器将项目一次部署至组内全部主机。

**简单来讲，群集的目标是让多台服务器分摊压力，当一台或多台服务器当机时，服务可以继续保持运转；而域的目标则是提供集中配置和管理多台服务器的能力。**

**4 Jboss4目录说明**



bin目录：主要包含run.jar、shutdowm.jar等文件，用于启动、停止服务器脚本

client目录：主要包含与客户端相关的文件（客户端与JBoss通信所需的的Java库（JARs））

docs目录：主要包含JBoss服务器的文档以及examples文件夹

lib目录：主要包含些JAR，JBoss启动时加载，且被所有JBoss配置共享（不要把你的库放在这里）

server目录：主要包含与服务器有关的配置文件。每个配置必须放在不同的子目录。子目录的名字config-name表示配置的名字。JBoss的server目录可以包含任意数目的目录，每个表示一个不同的JBoss配置。

**JBoss包含3个默认的配置：minimial，default和all**

server/all目录：JBoss的完全配置，启动所有服务，包括集群和IIOP。

server/default目录：JBoss的默认配置。在没有在JBoss命令航中指定配置名称时使用。

server/default/conf目录：JBoss的配置文件。

server/default/data目录：JBoss的数据库文件。比如，嵌入的数据库，或者JBossMQ。

server/default/deploy目录：JBoss的热部署目录。放到这里的任何文件或目录会被JBoss自动部署。EJB、WAR、EAR，甚至服务。

server/default/lib目录：一些JAR，JBoss在启动特定配置时加载他们。

server/minimal目录：server/config-name/conf目录包含JBoss的配置文件。

jacorb.properties JBoss IIOP 配置

jbossmq-state.xml JBossMQ（JMS的实现）用户配置

jboss-service.xml 启动时运行的JBoss的服务的定义（class loader，JNDI，deployer等）

log4j.xml Log4J的配置

login-config.xml JBoss安全配置（JBossSX）

standardjaws.xml JBoss的遗留CMP 1.1引擎的默认配置。包含对各种数据库的JDBC-to-SQL的映射信息，默认CMP设置，日志配置等。

standardjbosscmp-jdbc.xml 除了被用于JBoss的CMP 2.0引擎之外，与standardjaws.xml相同

**5 Jboss7目录说明**







**6 修改访问端口**

默认是0，端口是根据设置的基准值+偏移量得到的，JBoss默认的访问的端口是8080，默认的偏移量是0，所以访问的端口是8080，

修改红色部分，修改偏移量，这样就可以得到新的访问端口，在浏览器访问http://192.168.0.120:8180

在<JBOSS\_HOME> /standalone/configuration/standalone.xml文件中

<socket-binding-group name="standard-sockets" default-interface="public" port-offset="${jboss.socket.binding.port-offset:100}

也可以采用下面这种方法，这是指定一个ip，只有这一个ip能进行访问

<interface name="public">

<inet-address value="${jboss.bind.address:192.168.0.120}" />

</interface>

**7 启动过程**

设置环境变量 JBOSS\_CLASSPATH (可以自己加上安全管理器和xml解析器)

a) %JBOSS\_CLASSPATH%;

b) %JAVA\_HOME%/lib/tools.jar;

c) run.jar

d) ../lib/crimson.jar；

设置启动参数JAXP(xml解析器和相应工厂)

启动

设置配置信息

读取配置文件JBoss.properties，保存在系统属性中(System.properties)

设置缺省属性jboss.home和java.security.auth.login.config

创建MBeanServer.的实例:

把配置文件和补丁文件所在的目录指定给特定的远程类加载器Mlet

加载保存配置文件（mlet会自动在配置文件目录中查找）

初始化并启动MBean

配置服务ConfigurationService

加载配置文件

保存配置

服务控制ServiceControl

初始化服务程序（init方法）

启动服务程序（start方法）

**8 Tomcat和jboss比较**

（1）tomcat

Tomcat 服务器是一个免费的开放源代码的Web 应用服务器，它是Apache 软件基金会（Apache Software Foundation）的Jakarta 项目中的一个核心项目，由Apache、Sun 和其他一些公司及个人共同开发而成。由于有了Sun 的参与和支持，最新的Servlet 和JSP 规范总是能在Tomcat 中得到体现，Tomcat 5 支持最新的Servlet 2.4 和JSP 2.0 规范。因为Tomcat 技术先进、性能稳定，而且免费，因而深受Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的Web 应用服务器。

Tomcat 很受广大程序员的喜欢，因为它运行时占用的系统资源小，扩展性好，支持负载平衡与邮件服务等开发应用系统常用的功能；而且它还在不断的改进和完善中，任何一个感兴趣的程序员都可以更改它或在其中加入新的功能。

Tomcat 是一个小型的轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好Apache 服务器，可利用它响应对HTML 页面的访问请求。实际上Tomcat 部分是Apache 服务器的扩展，但它是独立运行的，所以当你运行tomcat 时，它实际上作为一个与Apache 独立的进程单独运行的。

（2）Jboss

JBoss是全世界开发者共同努力的成果，一个基于J2EE的开放源代码的应用服务器。 因为JBoss代码遵循LGPL许可，你可以在任何商业应用中免费使用它，而不用支付费用。JBoss支持EJB 1.1和EJB 2.0 EJB3.0的规范，它是一个管理EJB的容器和服务器。类似于Sun's J2SDK Enterprise Edition（J2EE），JBoss的目标是一个源代码开放的J2EE环境。但是JBoss核心服务仅是提供EJB服务器。JBoss不包括serverlers/JSP page 的WEB容器，当然可以和Tomcat或Jetty绑定使用。

JBoss还具有如下六大优点：

1、JBoss是免费的，开放源代码J2EE的实现，它通过LGPL许可证进行发布。

2、JBoss需要的内存和硬盘空间比较小。

3、安装非常简单。先解压缩JBss打包文件再配置一些环境变量就可以了。

4、JBoss能够"热部署"，部署BEAN只是简单拷贝BEAN的JAR文件到部署路径下就可以了。如果没有加载就加载它；如果已经加载了就卸载掉，然后LOAD这个新的。

5、JBoss与Web服务器在同一个Java虚拟机中运行，Servlet调用EJB不经过网络，从而大大提高运行效率，提升安全性能。

6、用户可以直接实施J2EE-EAR，而不是以前分别实施EJB-JAR和Web-WAR，非常方便。

JBoss的安装和配置可以直接拷贝使用，但是要改动 %JBoss-HOME%\bin\run.bat里JAVA-HOME的设置，改成本机JDK的目录。运行run.bat来启动JBoss。

## 四、Jetty 知识

**1 Jetty 是一个开源的servlet容器，**它为基于Java的web内容，例如JSP和servlet提供运行环境。Jetty是使用**Java语言编写**的，它的API以一组JAR包的形式发布。开发人员可以将Jetty容器实例化成一个对象，可以迅速为一些独立运行（stand-alone）的Java应用提供网络和web连接。可以  **嵌入式**Servlet容器。它极度轻量级、高便携性、功能强大、灵活和扩展性好，而且支持各种技术如SPDY、WebSocket、OSGi、JMX、JNDI和JAAS。Jetty对于开发来说非常便利，也已经广泛地应用于生产环境。如何安装和配置Jetty服务器，主要讲述Jetty的独立运行模式（**Standalone**）。随后介绍它的配置选项，以及模块化的架构。

Jetty有**独立运行模式、嵌入运行模式和Jetty Maven插件**操作模式。独立运行模式为准。

**2 部署环境**

两个软件：

1）Java 8u60

http://download.Oracle.com/otn-pub/java/jdk/8u60-b27/jdk-8u60-windows-x64.exe

2）Jetty 9.3.2

http://eclipse.org/downloads/download.php?file=/jetty/stable-9/dist/jetty-distribution-9.3.2.v20150730.zip&r=1

早些的版本也可以用，但应该尽可能使用最新版本。

Jetty是一个跨平台的软件，因此它的二进制发布包可适用于多个平台。

**3 部署运行**

下载并解压Jetty，进入Jetty的解压目录，执行命令：**java -jar start.jar**

输出如下：

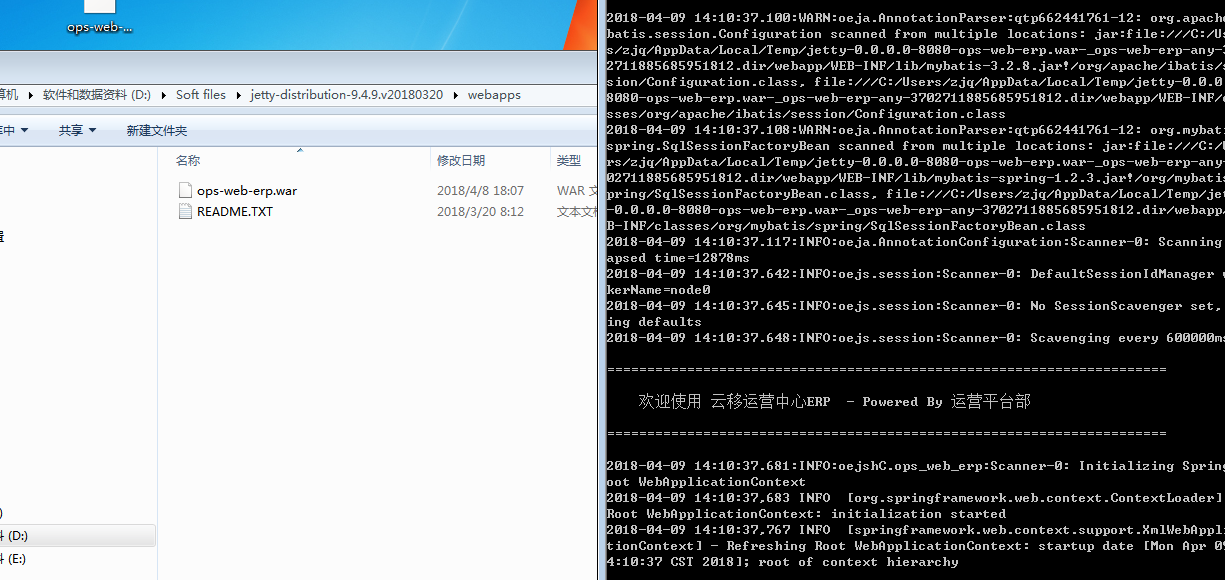


**4 用浏览器访问地址：http://localhost:8080/，可以看到**



**5 运行Jetty Web应用**

在Jetty应用服务器中部署war项目很简单，只需把项目war包放入Jetty的webapps子目录即可。你都无需重启Jetty，Jetty会自动随时监听webapps目录，自动完成新项目的部署。





**6 Jetty目录剖析**

bin：可执行脚本文件

demo- base：

etc：Jetty模块定义的XML配置文件的目录

lib：Jetty依赖的库文件

logs：Jetty的日志目录

modules：Jetty的模块

resources：外部资源配置文件的目录

webapps：项目WAR文件的目录

还需要关心根目录下的一个文件：start.d（Wondows系统是start.ini文件），它定义了Jetty的活动模块。

**7 基本配置**

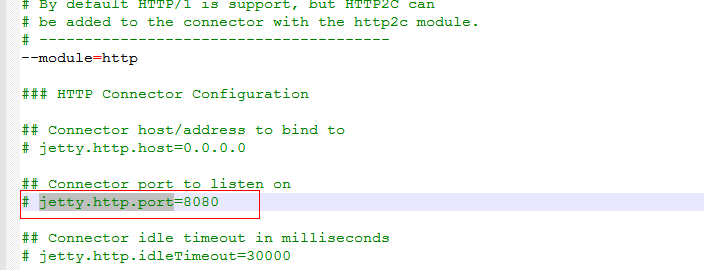
1）、修改Jetty的端口

Jetty默认使用8080端口，要让它使用其他端口（如7070），那么编辑start.d（Wondows系统是start.ini文件），找到jetty.http.port行，修改为：

## Connector port to listen on

jetty.http.port=7070

保存并退出，再重启Jetty。

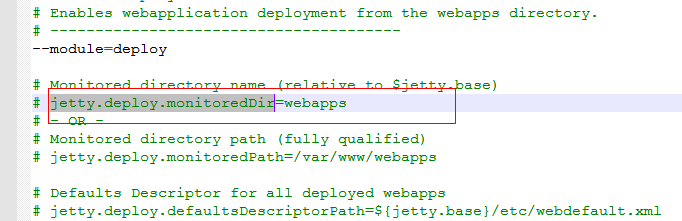


2）、修改webapps目录

Jetty下的webapps是默认的Web项目的部署目录，如果想修改此目录，可修改start.d配置文件（start.ini），移除以下行的注释符号“#”

# jetty.deploy.monitoredDir=webapps

并把内容修改到你指定的目录。保存并退出，再重启Jetty。



**8 Jetty的模块化架构**

Jetty运行于模块化的架构之上，这意味着Jetty的功能是以模块的方式运行的，比如HTTP、HTTPS、SSL、日志logging、JMX、JNDI、WebSocket等模块。常用的模块如HTTP、JSP和WebSocket模块都是默认就激活的，而其他如HTTPS、JMX等模块则需要手动激活。

1）、单个模块的剖析

Jetty的modules子目录列出了所有的模块，这些模块是扩展名为.mod的文件，它声明了要被激活的JAR文件（在Jetty的lib子目录下）和XML配置文件（在Jetty的etc子目录下），以及其他要作为模块被激活的资源。

比如，可以查看modules子目录的logging.mod文件的内容，可以看到，它声明了配置文件是etc/jetty-logging.xml，所需的JAR包在lib/logging处，另外logs目录是必须的。

[ xml]

etc/jetty-logging.xml

[files]

logs/

[lib]

lib/logging/\*\*.jar

resources/

2）、通过命令行激活模块

激活Jetty的模块有两种方式。

第一种方式是通过命令行激活：

java -jar start.jar --add-to-startd=logging

上面的命令会在Jetty目录下创建logging.ini文件，相关的配置可以在此文件中查到。配置日志后，可以再次启动Jetty，并可以查看到日志模块是激活了的。

3）、通过配置文件start.ini激活模块

第二种方式是通过配置文件start.ini激活模块

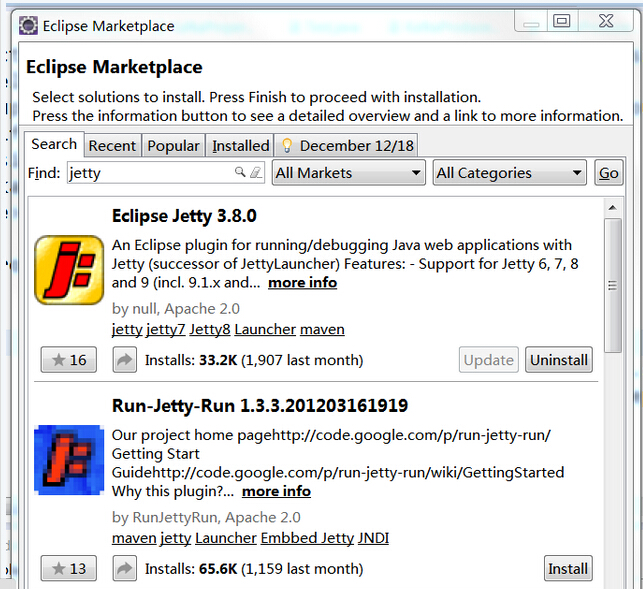
--module=logging

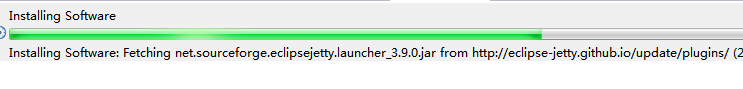
这种方式和前一种相似，且更常用。

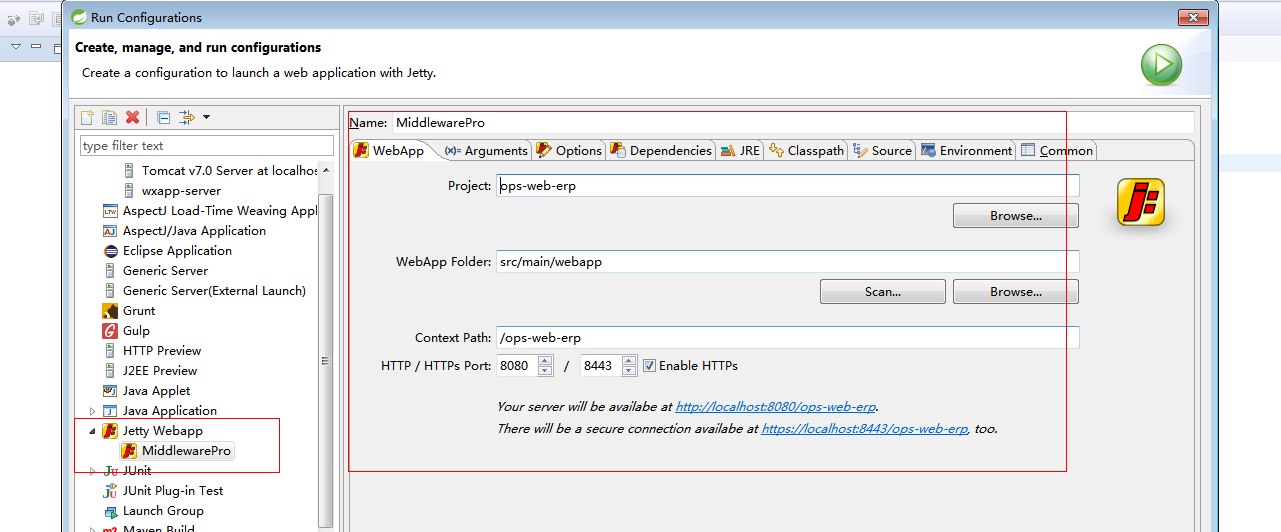
4）、配置模块

正如上面提到的，mod文件声明了相关的XML配置文件，在Jetty的etc子目录下，可以通过这些配置文件来配置模块。比如日志模块声明了相关的配置文件是jetty-logging.xml，可以通过修改此配置文件来调整日志。

**9 eclipse和 jetty**







报错加载类找不到

