

公告

昵称：crazyYong
园龄：4年11个月
粉丝：52
关注：2

< 2017年10月 >						
日	一	二	三	四	五	六
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

搜索

找找看

谷歌搜索

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签

随笔分类

- (张子秋)从零开始学习jQuery系列(11)
- aws(4)
- docker
- dubbo(1)
- eclipse-myeclipse(1)
- Ehcache(13)
- hibernate学习笔记(35)
- javascript(6)
- javaweb学习总结(32)
- java反射(8)
- java集合(4)
- java加密(10)
- java设计模式(6)
- java数据结构与算法(14)
- java虚拟机(16)
- java杂(23)
- jboss(7)
- JqueryDemo(1)
- Linux鸟哥(12)
- Linux杂(3)
- linux指令(6)
- mangodb(1)
- maven(2)
- mybaties(1)
- MySQL性能调优与架构设计（简朝阳）(22)
- mysql杂(26)
- netty权威指南(16)
- nginx(11)
- rabbitMQ(3)
- redis(20)
- spring(50)
- springsecurity(2)
- struts2.1.6教程(13)
- utils(15)

Tomcat服务器原理详解

【目录】 本文主要讲解**Tomcat**启动和部署**webapp**时的原理和过程， 以及其使用的配置文件的详解。主要有三大部分：

第一部分、Tomcat的简介和启动过程

第二部分、Tomcat部署webapp

第三部分、Tomcat处理一个http请求的过程

【简介】

Tomcat依赖**<CATALINA_HOME>/conf/server.xml**这个配置文件启动**server**（一个**Tomcat**实例，核心就是启动容器**Catalina**）。

Tomcat部署**Webapp**时，依赖**context.xml**和**web.xml**（**<CATALINA_HOME>/conf/**目录下的**context.xml**和**web.xml**在部署任何**webapp**时都会启动，他们定义一些默认行为，而具体每个**webapp**的**META-INF/context.xml** 和 **WEB-INF/web.xml** 则定义了每个**webapp**特定的行为）两个配置文件部署**web**应用。

第一部分、Tomcat的简介和启动过程

一、Tomcat的下载包解压之后的目录



tomcat根目录在tomcat中叫<CATALINA_HOME>

<CATALINA_HOME>/bin： 存放各种平台下启动和关闭Tomcat的脚本文件.其中 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. startup.bat是windows下启动tomcat的文件,shutdown.bat是关闭tomcat的文件。

<CATALINA_HOME>/conf： 存放不同的配置文件（如：server.xml和web.xml）；

server.xml文件:该文件用于配置和server相关的信息，比如tomcat启动的端口号、配置host主机、配置Context

web.xml文件： 部署描述文件，这个web.xml中描述了一些默认的servlet，部署每个webapp时，都会调用这个文件，配置该web应用的默认servlet。

tomcat-users.xml文件： 配置tomcat的用户密码与权限。

context.xml： 定义web应用的默认行为。

<CATALINA_HOME>/lib： 存放Tomcat运行需要的库文件（JARS）；

<CATALINA_HOME>/logs： 存放Tomcat执行时的LOG文件；

<CATALINA_HOME>/temp：

<CATALINA_HOME>/webapps： Tomcat的主要Web发布目录（包括应用程序示例）；

<CATALINA_HOME>/work： 存放jsp编译后产生的class文件；

二、Tomcat启动过程

1、开启Tomcat： 可以在IDE中启动Tomcat的服务器，也可以手动在<CATALINA_HOME>/bin/目录下找到

- web安全(16)
- zookeeper(9)
- 第一本docker笔记(2)
- 服务器(9)
- 构建(1)
- 集成(2)
- 开发工具(1)
- 面试(13)
- 其他(3)
- 系统架构(12)
- 性能优化(2)

随笔档案

- 2017年10月 (17)
- 2017年9月 (31)
- 2017年8月 (13)
- 2017年7月 (17)
- 2017年6月 (3)
- 2017年5月 (14)
- 2017年3月 (4)
- 2016年12月 (1)
- 2016年10月 (8)
- 2016年9月 (2)
- 2016年8月 (7)
- 2016年7月 (18)
- 2016年6月 (13)
- 2016年5月 (7)
- 2016年4月 (1)
- 2016年2月 (15)
- 2016年1月 (2)
- 2015年12月 (1)
- 2015年11月 (5)
- 2015年10月 (19)
- 2015年9月 (5)
- 2015年8月 (10)
- 2015年4月 (16)
- 2015年3月 (53)
- 2015年2月 (4)
- 2015年1月 (50)
- 2014年12月 (52)
- 2014年11月 (55)
- 2014年10月 (19)

文章分类

javaweb 杂

积分与排名

积分 - 113745

排名 - 2404

最新评论

阅读排行榜

评论排行榜

推荐排行榜

startup.bat并双击，然后程序就会依次执行以下步骤：

(1) 引导（Bootstrap）启动：*调用了org.apache.catalina.startup.Bootstrap.class中的main方法，开始启动Tomcat容器；main方法如下：*



```
public static void main(String args[]) {

    if (daemon == null) {
        daemon = new Bootstrap();//创建了一个引导对象
        try //引导对象初始化，即创建了Catalina容器
            daemon.init();
        } catch (Throwable t) {
            t.printStackTrace();
            return;
        }
    }

    // 根据不同的命令参数执行
    try {
        String command = "start";
        if (args.length > 0) {
            command = args[args.length - 1];
        }

        if (command.equals("startd")) {
            args[args.length - 1] = "start";
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stopd")) {
            args[args.length - 1] = "stop";
            daemon.stop();
        } else if (command.equals("start")) {
            daemon.setAwait(true);
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stop")) {
            daemon.stopServer(args);
        } else {
            log.warn("Bootstrap: command \"" + command + "\" does not exist.");
        }
    } catch (Throwable t) {
        t.printStackTrace();
    }

}

public void init()
    throws Exception
{

    // Set Catalina path
    // 设置catalina_home属性，tomcat启动脚本里有通过-Dcatalina.home设置
    setCatalinaHome();
    // 设置catalina_base属性，运行多实例的时候该目录与catalina_home不同
    setCatalinaBase();

    // 初始化classloader，读取conf/catalina.properties，根据指定的repository创建classloader
    // 有三个classloader 分别是common、catalina、shared，tomcat6中三个相同
    initClassLoaders();
    // 设置当前线程的classloader
    Thread.currentThread().setContextClassLoader(catalinaLoader);
    // 待研究
    SecurityClassLoad.securityClassLoad(catalinaLoader);

    // 以下通过反射调用Catalina中的方法
    // Load our startup class and call its process() method
    if (log.isDebugEnabled())
        log.debug("Loading startup class");
```

```
Class startupClass =
    catalinaLoader.loadClass
        ("org.apache.catalina.startup.Catalina");
Object startupInstance = startupClass.newInstance();

// Set the shared extensions class loader
if (log.isDebugEnabled())
    log.debug("Setting startup class properties");
String methodName = "setParentClassLoader";
Class paramTypes[] = new Class[1];
paramTypes[0] = Class.forName("java.lang.ClassLoader");
Object paramValues[] = new Object[1];
paramValues[0] = sharedLoader;
Method method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);

catalinaDaemon = startupInstance;

}
```

2) 调用Bootstrap中的init () ，创建了Catalina对象（核心容器）：主要进行了以下三步：

- ① Set up the environment variables required by this Tomcat instance
- ② Instantiate the general class loaders that will be used for our running Tomcat instance
- ③ Initialize this Tomcat instance

(3) 调用Bootstrap中的load () ：实际上是通过反射调用了catalina的load方法。

①Parse the main configuration file for a Tomcat instance, server.xml, converting each configuration element into the appropriate Tomcat component1。（找到config file (server.xml) ；然后创建digester，解析server.xml，生成各组件对象（Server、Service、Container、Connector等）以及建立相互之间的关系。）

```
// Configure the actions we will be using
digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer",
    "className");
digester.addSetProperties("Server");
digester.addSetNext("Server",
    "setServer",
    "org.apache.catalina.Server");

digester.addObjectCreate("Server/GlobalNamingResources",
    "org.apache.catalina.deploy.NamingResources");
digester.addSetProperties("Server/GlobalNamingResources");
digester.addSetNext("Server/GlobalNamingResources",
    "setGlobalNamingResources",
    "org.apache.catalina.deploy.NamingResources");

digester.addObjectCreate("Server/Listener",
    null, // MUST be specified in the element
    "className");
digester.addSetProperties("Server/Listener");
digester.addSetNext("Server/Listener",
    "addLifecycleListener",
    "org.apache.catalina.LifecycleListener");

digester.addObjectCreate("Server/Service",
    "org.apache.catalina.core.StandardService",
    "className");
digester.addSetProperties("Server/Service");
digester.addSetNext("Server/Service",
    "addService",
    "org.apache.catalina.Service");

digester.addObjectCreate("Server/Service/Listener",
    null, // MUST be specified in the element
```

```
        "className");

    digester.addSetProperties("Server/Service/Listener");
    digester.addSetNext("Server/Service/Listener",
        "addLifecycleListener",
        "org.apache.catalina.LifecycleListener");

    //Executor
    digester.addObjectCreate("Server/Service/Executor",
        "org.apache.catalina.core.StandardThreadExecutor",
        "className");
    digester.addSetProperties("Server/Service/Executor");

    digester.addSetNext("Server/Service/Executor",
        "addExecutor",
        "org.apache.catalina.Executor");

    digester.addRule("Server/Service/Connector",
        new ConnectorCreateRule());
    digester.addRule("Server/Service/Connector",
        new SetAllPropertiesRule(new String[]{"executor"}));
    digester.addSetNext("Server/Service/Connector",
        "addConnector",
        "org.apache.catalina.connector.Connector");

    // ...
```



(4) Start up our outermost Top Level Element—the Server instance。（最后**start()**，同样是在**Bootstrap**中通过反射调用**catalina**对象的**start**方法。接着启动**server.start()**方法：
`((Lifecycle) getServer()).start()`；接着调用**service.start()**方法。接下来是一系列的**container**的**start**，后续在分析（会部署所有的项目）



```
public void start() {

    if (getServer() == null) {
        load();
    }

    if (getServer() == null) {
        log.fatal("Cannot start server. Server instance is not configured.");
        return;
    }

    long t1 = System.nanoTime();

    // Start the new server
    if (getServer() instanceof Lifecycle) {
        try {
            ((Lifecycle) getServer()).start();
        } catch (LifecycleException e) {
            log.error("Catalina.start: ", e);
        }
    }

    long t2 = System.nanoTime();
    if(log.isInfoEnabled())
        log.info("Server startup in " + ((t2 - t1) / 1000000) + " ms");

    try {
        // Register shutdown hook
        if (useShutdownHook) {
            if (shutdownHook == null) {
                shutdownHook = new CatalinaShutdownHook();
            }
            Runtime.getRuntime().addShutdownHook(shutdownHook);

            // If JULI is being used, disable JULI's shutdown hook since
            // shutdown hooks run in parallel and log messages may be lost
```



```

        // if JULI's hook completes before the CatalinaShutdownHook()
        LogManager logManager = LogManager.getLogManager();
        if (logManager instanceof ClassLoaderLogManager) {
            ((ClassLoaderLogManager) logManager).setUseShutdownHook(
                false);
        }
    }
} catch (Throwable t) {
    // This will fail on JDK 1.2. Ignoring, as Tomcat can run
    // fine without the shutdown hook.
}

if (await) {
    await();
    stop();
}

}

}

```

(5) Set up a shutdown hook

A shutdown hook is a standard Thread that encapsulates cleanup actions that should be taken before the Java runtime exits. All shutdown hooks are called by the runtime when the JVM is shutting down. Therefore, the last task that we perform is to install a shutdown hook, as implemented by CatalinaShutdownHook. This hook is registered with the Java Runtime by invoking its addShutdownHook() method:

Runtime.getRuntime().addShutdownHook(),

CatalinaShutdownHook is an inner class of Catalina and so has access to all the data members of Catalina. Its run() method is very simple. It just ensures that stop() is called on this instance of Catalina. This method invokes stop() on the StandardServer instance, which in turn performs a cascaded invocation of stop() on all its child components. Each child does the same for its children, until the entire server has been cleanly stopped.

使用类CatalinaShutdownHook实现，它继承Thread，run中进行清理

```

// Register shutdown hook
if (useShutdownHook) {
    if (shutdownHook == null) {
        shutdownHook = new CatalinaShutdownHook();
    }
    Runtime.getRuntime().addShutdownHook(shutdownHook);
    // If JULI is being used, disable JULI's shutdown hook since
    // shutdown hooks run in parallel and log messages may be lost
    // if JULI's hook completes before the CatalinaShutdownHook()
    LogManager logManager = LogManager.getLogManager();
    if (logManager instanceof ClassLoaderLogManager) {
        ((ClassLoaderLogManager) logManager).setUseShutdownHook(
            false);
    }
}
}

```

三、server.xml配置简介：

下面讲述这个文件中的基本配置信息，更具体的配置信息请参考tomcat的文档：

元素名	属性	解释
server	port	指定一个端口，这个端口负责监听关闭tomcat的请求
	shutdown	指定向端口发送的命令字符串
service	name	指定 service 的名字
Connector(表示客户端和service之间的连接)	port	指定服务器端要创建的端口号，并在这个端口监听来自客户端的请求
	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
	enableLookups	如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机，DNS查询，而是返回其ip地址
	redirectPort	指定服务器正在处理http请求时收到了一个SSL传输请求后重定向的端口号
	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
	connectionTimeout	指定超时时间数(以毫秒为单位)
Engine(表示指定service中的请求处理机，接收和处理来自Connector的请求)	defaultHost	指定缺省的处理请求的主机名，它至少与其中的一个host元素的name属性值是一样的
Context(表示一个web应用程序，通常为WAR文件，关于WAR的具体信息见servlet规范)	docBase	应用程序的路径或者是WAR文件存放的路径
	path	表示此web应用程序的url的前缀，这样请求的url为http://localhost:8080/path/****
	reloadable	这个属性非常重要，如果为true，则tomcat会自动检测应用程序的/WEB-INF/lib 和/WEB-INF/classes 载新的应用程序，我们可以在不重启tomcat的情况下改变应用程序
host(表示一个虚拟主机)	name	指定主机名
	appBase	应用程序基本目录，即存放应用程序的目录
	unpackWARs	如果为true，则tomcat会自动将WAR文件解压，否则不解压，直接从WAR文件中运行应用程序
Logger(表示日志，调试和错误信息)	className	指定logger使用的类名，此类必须实现org.apache.catalina.Logger 接口
	prefix	指定log文件的前缀
	suffix	指定log文件的后缀
	timestamp	如果为true，则log文件名中要加入时间，如下例:localhost_log_2001-10-04.txt
Realm(表示存放用户名，密码及role的数据库)	className	指定Realm使用的类名，此类必须实现org.apache.catalina.Realm接口
Valve(功能与Logger差不多，其prefix和suffix属性解释和Logger 中的一样)	className	指定Valve使用的类名，如用org.apache.catalina.valves.AccessLogValve类可以记录应用程序的访问信息
	directory	指定log文件存放的位置
	pattern	有两个值，common方式记录远程主机名或ip地址，用户名，日期，第一行请求的字符串，HTTP方法。combined方式比common方式记录的值更多

四、web.xml配置简介：

1、默认(欢迎)文件的设置

在tomcat4\conf\web.xml中， <welcome-file-list> 与IIS中的默认文件意思相同。

```
<welcome-file-list>
<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

2、报错文件的设置

```
<error-page>
<error-code>404</error-code>
<location>/notFileFound.jsp</location>
</error-page>
<error-page>
<exception-type>java.lang.NullPointerException</exception-type>
<location>/null.jsp</location>
</error-page>
```

如果某文件资源没有找到，服务器要报404错误，按上述配置则会调用\webapps\ROOT\notFileFound.jsp。

如果执行的某个JSP文件产生NullPointException ，则会调用\webapps\ROOT\null.jsp

3、会话超时的设置

设置session 的过期时间，单位是分钟；

```
<session-config>
<session-timeout>30</session-timeout>
</session-config>
```

4、过滤器的设置

```
<filter>
<filter-name>FilterSource</filter-name>
<filter-class>project4. FilterSource </filter-class>
</filter>
<filter-mapping>
<filter-name>FilterSource</filter-name>
<url-pattern>/WwwServlet</url-pattern>
(<url-pattern>/haha/*</url-pattern>)
</filter-mapping>
```

过滤：

- 1) 身份验证的过滤Authentication Filters
- 2) 日志和审核的过滤Logging and Auditing Filters

- 3) 图片转化的过滤Image conversion Filters
- 4) 数据压缩的过滤Data compression Filters
- 5) 加密过滤Encryption Filters
- 6) Tokenizing Filters
- 7) 资源访问事件触发的过滤Filters that trigger resource access events XSL/T 过滤XSL/T filters
- 9) 内容类型的过滤Mime-type chain Filter 注意监听器的顺序，如：先安全过滤，然后资源，然后内容类型等，这个顺序可以自己定。

五、管理

- 1、用户配置
 - 在进行具体Tomcat管理之前，先给tomcat添加一个用户，使这个用户有权限来进行管理。
 - 打开conf目录下的tomcat-users.xml文件，在相应的位置添加下面一行：
<user name="user" password="user" roles="standard,manager"/>
 - 然后重起tomcat，在浏览器中输入http://localhost:8080/manager/，会弹出对话框，输入上面的用户名和密码即可。
- 2、应用程序列表
 - 在浏览器中输入http://localhost:8080/manager/list，浏览器将会显示如下的信息：
OK - Listed applications for virtual host localhost
/ex:running:1
/examples:running:1
/webdav:running:0
/tomcat-docs:running:0
/manager:running:0
/:running:0
 - 上面显示的信息分别为：应用程序的路径、当前状态、连接这个程序的session数
- 3、重新装载应用程序
 - 在浏览器中输入 http://localhost:8080/manager/reload?path=/examples，浏览器显示如下：
OK - Reloaded application at context path /examples
 - 这表示example应用程序装载成功，如果我们将server.xml的Context元素的reloadable属性设为true，则没必要利用这种方式重新装载应用程序，因为tomcat会自动装载。
- 4、显示session信息
 - 在浏览器中输入http://localhost:8080/manager/sessions?path=/examples，浏览器显示如下：
OK - Session information for application at context path /examples Default maximum session inactive interval 30 minutes
- 5、启动和关闭应用程序
 - 在浏览器中输入http://localhost:8080/manager/start?path=/examples和http://localhost:8080/manager/stop?path=/examples分别启动和关闭examples应用程序。

六 、 Tomcat Server的组成部分

1.1 – Server

A Server element represents the entire Catalina servlet container. (Singleton)

1.2 – Service

A Service element represents the combination of one or more Connector components that share a single Engine
Service是这样一个集合：它由一个或者多个Connector组成，以及一个Engine，负责处理所有Connector所获得的客户请求

1.3 – Connector

一个Connector将在某个指定端口上侦听客户请求，并将获得的请求交给Engine来处理，从Engine处获得回应并返回客户
TOMCAT有两个典型的Connector，一个直接侦听来自browser的http请求，一个侦听来自其它WebServer的请求
Coyote Http/1.1 Connector 在端口8080处侦听来自客户browser的http请求
Coyote JK2 Connector 在端口8009处侦听来自其它WebServer(Apache)的servlet/jsp代理请求

1.4 – Engine

The Engine element represents the entire request processing machinery associated with a particular Service

It receives and processes all requests from one or more Connectors and returns the completed response to the Connector for ultimate transmission back to the client

Engine下可以配置多个虚拟主机Virtual Host，每个虚拟主机都有一个域名

当Engine获得一个请求时，它把该请求匹配到某个Host上，然后把该请求交给该Host来处理

Engine有一个默认虚拟主机，当请求无法匹配到任何一个Host上的时候，将交给该默认Host来处理

1.5 – Host

代表一个Virtual Host，虚拟主机，每个虚拟主机和某个网络域名Domain Name相匹配

每个虚拟主机下都可以部署(deploy)一个或者多个Web App，每个Web App对应于一个Context，有一个Context path

当Host获得一个请求时，将把该请求匹配到某个Context上，然后把该请求交给该Context来处理

匹配的方法是“最长匹配”，所以一个path==“”的Context将成为该Host的默认Context

所有无法和其它Context的路径名匹配的请求都将最终和该默认Context匹配

1.6 – Context

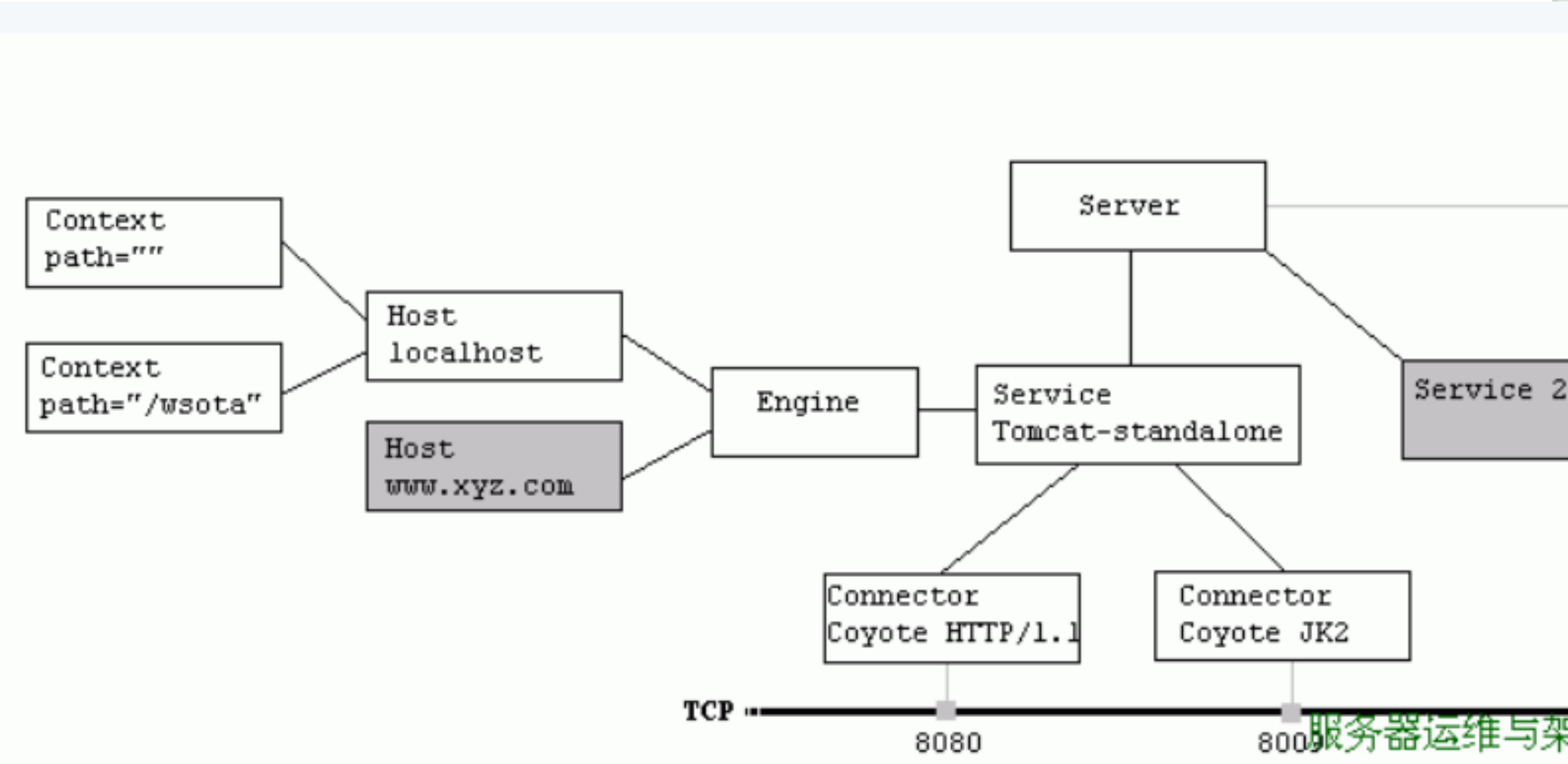
一个Context对应于一个Web Application，一个Web Application由一个或者多个Servlet组成

Context在创建的时候将根据配置文件\$CATALINA_HOME/conf/web.xml和\$WEBAPP_HOME/WEB-INF/web.xml载入Servlet类

当Context获得请求时，将在自己的映射表(mapping table)中寻找相匹配的Servlet类

如果找到，则执行该类，获得请求的回应，并返回

2 – Tomcat Server的结构图



【Tomcat的启动过程】Tomcat 先根据/conf/server.xml 下的配置启动Server，再加载Service，对于与Engine相匹配的Host，每个Host 下面都有一个或多个Context。

注意：Context 既可配置在server.xml 下，也可配置成一单独的文件，放在conf\Catalina\localhost 下，简称应用配置文件。

Web Application 对应一个Context，每个Web Application 由一个或多个Servlet 组成。当一个Web Application 被初始化的时候，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 Application 根目录下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 路径对应。当该Context 获得请求时，将查询mapping table，找到被请求的Servlet，并执行以获得请求响应。

所以，对于Tomcat 来说，主要就是以下这几个文件：conf 下的server.xml、web.xml，以及项目下的web.xml，加载就是读取这些配置文件。

3 – 配置文件\$CATALINA_HOME/conf/server.xml的说明

该文件描述了如何启动Tomcat Server





```
<!------->
<!-- 启动Server 在端口8005处等待关闭命令 如果接受到"SHUTDOWN"字符串则关闭服务器 -->
<Server port="8005" shutdown="SHUTDOWN" debug="0">
<!-- Listener ??? 目前没有看到这里 -->
<Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" debug="0"/>
<Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" debug="0"/>
<!-- Global JNDI resources ??? 目前没有看到这里，先略去 -->
<GlobalNamingResources>
    . . . . .
</GlobalNamingResources>
<!-- Tomcat的Standalone Service Service是一组Connector的集合 它们共用一个Engine来处理所有Connector收到的请求 -->
<Service name="Tomcat-Standalone">
<!-- Coyote HTTP/1.1 Connector className : 该Connector的实现类是
org.apache.coyote.tomcat4.CoyoteConnector port :
在端口号8080处侦听来自客户browser的HTTP1.1请求 minProcessors : 该Connector先创建5个线程等待客户请求，
每个请求由一个线程负责 maxProcessors : 当现有的线程不够服务客户请求时，若线程总数不足75个，则创建新线程来处理请求
acceptCount : 当现有线程已经达到最大数75时，为客户请求排队 当队列中请求数超过100时，后来的请求返回Connection
refused
错误 redirectport : 当客户请求是https时，把该请求转发到端口8443去 其它属性略 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
    port="8080"
    minProcessors="5" maxProcessors="75" acceptCount="100"
    enableLookups="true"
    redirectPort="8443"
    debug="0"
    connectionTimeout="20000"
    useURIVValidationHack="false"
    disableUploadTimeout="true" />
<!-- Engine用来处理Connector收到的Http请求 它将匹配请求和自己的虚拟主机，
并把请求转交给对应的Host来处理默认虚拟主机是localhost -->
<Engine name="Standalone" defaultHost="localhost" debug="0">
<!-- 日志类，目前没有看到，略去先 -->
<Logger className="org.apache.catalina.logger.FileLogger" .../>
<!-- Realm，目前没有看到，略去先 -->
<Realm className="org.apache.catalina.realm.UserDatabaseRealm" .../>
<!-- 虚拟主机localhost appBase : 该虚拟主机的根目录是webapps/ 它将匹配请求和
自己的Context的路径，并把请求转交给对应的Context来处理 -->
<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true" autoDeploy="true">
<!-- 日志类，目前没有看到，略去先 -->
<Logger className="org.apache.catalina.logger.FileLogger" .../>
<!-- Context，对应于一个Web App path : 该Context的路径名是"", 故该Context是该Host的
默认Context docBase : 该Context的根目录是webapps/mycontext/ -->
<Context path="" docBase="mycontext" debug="0"/>
<!-- 另外一个Context，路径名是/wsota -->
<Context path="/wsota" docBase="wsotaProject" debug="0"/>
</Host>
</Engine>
</Service>
</Server>
<!------->
```





第二部分、Tomcat部署webapp

1、Context的部署配置文件web.xml的说明

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
- 分析一下所有的Context共享的web.xml文件，在其中定义的servlet被所有的Web App载入



```
<!------->

<web-app>

<!-- 概述： 该文件是所有的WEB APP共用的部署配置文件， 每当一个WEB APP
被DEPLOY，该文件都将先被处理，然后才是WEB APP自己的/WEB-INF/web.xml -->

<!-- +-----+ -->

<!-- | servlet类定义部分 | -->

<!-- +-----+ -->

<!-- DefaultServlet
当用户的HTTP请求无法匹配任何一个servlet的时候，该servlet被执行
URL PATTERN MAPPING : / -->

<servlet>
<servlet-name>default</servlet-name>
<servlet-class>
org.apache.catalina.servlets.DefaultServlet
</servlet-class>
<init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
</init-param>
<init-param>
    <param-name>listings</param-name>
    <param-value>>true</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>

<!-- InvokerServlet
处理一个WEB APP中的匿名servlet 当一个servlet被编写并编译放入
/WEB-INF/classes/中，却没有在/WEB-INF/web.xml中定义的时候
该servlet被调用，把匿名servlet映射成/servlet/ClassName的形式
URL PATTERN MAPPING : /servlet/* -->

<servlet>
    <servlet-name>invoker</servlet-name>
    <servlet-class>org.apache.catalina.servlets.InvokerServlet </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- JspServlet
当请求的是一个JSP页面的时候 (*.jsp) 该servlet被调用
它是一个JSP编译器，将请求的JSP页面编译成为servlet再执行
URL PATTERN MAPPING : *.jsp -->

<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>logVerbosityLevel</param-name>
        <param-value>WARNING</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>

<!-- +-----+ -->

<!-- | servlet映射定义部分 | -->

<!-- +-----+ -->

<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jsp</url-pattern>
</servlet-mapping>

<!-- +-----+ -->

<!-- | 其它部分，略去先 | -->
```

```
<!-- +-----+-->
... ..
</web-app>
<!------->
```



2、Context.xml和Context节点说明：


(1) 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**

- ① path：即要建立的**虚拟目录**，,注意是/kaka，访问**Web应用的 上下文根**，如 **http://localhost:8080/kaka/******。这个属性必须是**唯一的，对应一个webapp**。
 - ② docBase：为应用程序的路径或WAR文件存放的路径，可以是绝对路径，也可是相对路径，**相对路径是相对于 <Host >**
 - ③ reloadable：如果这个属性设为true，Tomcat服务器在运行状态下会监视在WEB-INF/classes和Web-INF /lib目录CLASS文件的改变，如果监视到有class文件被更新，服务器自动重新加载Web应用，这样我们可以在不重起tomcat的情况下改变应 用程序
- 一个Host元素中嵌套任意多的Context元素。每个Context的路径必须是惟一的，由path属性定义。另外，你必须定义一个path=""的context，这个Context称为该虚拟主机的**缺省web应用**，用来处理那些不能匹配任何Context的Context路径的请求。

(2) 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 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>
```



(3) 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）

第三部分、Tomcat处理一个http请求的过程

1– Tomcat Server处理一个http请求的过程

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

（1-1）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

参考：<http://docs.huihoo.com/apache/tomcat/heavyz/01-startup.html>

from:<http://www.cnblogs.com/mo-wang/p/3705147.html>

分类：[服务器](#)

好文要顶

关注我

收藏该文

[crazyYong](#)
关注 - 2
粉丝 - 52

+加关注

1

0

推荐

反对

« 上一篇：[Tomcat 系统架构与设计模式，第 2 部分：设计模式分析](#)
» 下一篇：[论MySQL何时使用索引，何时不使用索引](#)

posted @ 2015-08-06 00:07 [crazyYong](#) 阅读(3672) 评论(1) 编辑 收藏

评论列表

#1楼

2016-07-31 15:11

城固如春

讲的很好~学习了

支持(0) 反对(0)

刷新评论

刷新页面

返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

- 【推荐】50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【域名】腾讯云 新注册用户域名抢购1元起
- 【推荐】报表开发有捷径：快速设计轻松集成，数据可视化和交互



最新IT新闻:

- 王健林的新目标
 - 背靠阿里又远离阿里，钉钉如何成了马云口中的"惊喜"
 - 东芝股东大会批准出售闪存芯片子公司 2万亿日元卖给贝恩财团
 - 大量iPhone 8入市以旧换新：惨成iPhone X过渡机
 - 微软撤销针对美国司法部的诉讼：因政府已推出新的政策
- » 更多新闻...



最新知识库文章:

- 实用VPC虚拟私有云设计原则
 - 如何阅读计算机科学类的书
 - Google 及其云智慧
 - 做到这一点，你也可以成为优秀的程序员
 - 写给立志做码农的大学生
- » 更多知识库文章...