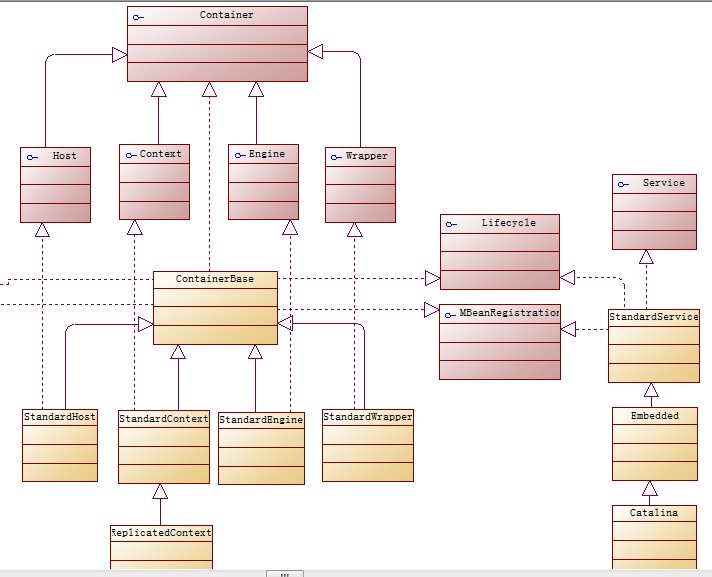
Tomcat 学习

Abstract

主要围绕tomcat 的架构和功能进行总结。

Table of Contents

# Tomcat architecture



## Server (图上并未画出)

Server代表整个容器(container)。它可以包含一个或多个Service，还可以包含一个GlobalNamingResources。

标准的Server接口中没有包括Lifecycle接口，但是在标准实现

org.apache.catalina.core.StandardServer中却实现了Lifecycle这个接口，这使得我们可以为Tomcat 的标准实现设置Listener。一般的方法是在conf/server.xml文件中加入：

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

<Listener className="org.solol.listener.XXXLifecycleListener" />

<Listener className="org.solol.listener.XXXLifecycleListener" />

:

</Server>

## Service

Service中可以包含一个或多个Connector、 Executor，但只有一个Container。不同的Connector可以共享同一个Container。可以看出StandardService. setContainer仅指设置 Engine。在同一个Server中的多个Service之间没有相关性。

Service接口中没有包括Lifecycle接口，但是StandardService中却实现了Lifecycle这个接口，这使得我们可通编写Listener 监听Service 的生命周期。

## Connector

Connector负责接收来自客户端(Client)的请求。比较常见的两个是[HTTP Connector](http://tomcat.apache.org/tomcat-5.5-doc/config/http.html" \o "HTTP Connector)和[AJP Connector](http://tomcat.apache.org/tomcat-5.5-doc/config/ajp.html" \o "AJP Connector)。

## Container

### Engine

负责接收和处理来自它所属的Service中的所有Connector的请求。

Engine 支持继承Container。自己被关联到一个Service。自己属性有：defaultHost、jvmRoute。

### Host

表示一个虚拟主机，并和一个服务器的网络名关联。注意Engine中必须有一个Host的名字和Engine的defaultHost属性匹配 。

### Context

表示在虚拟主机中运行的web应用程序。一个虚拟主机中能够运行多个Context，它们通过各自的Context Path进行相互区分。如果Context Path为""，那么该web应用为该虚拟主机的默认的web应用。

目前可以通过四种方式将Context加入Host:

1. $CATALINA\_HOME/conf/context.xml，其中Context元素中的信息会被所有web应用程序加载
2. $CATALINA\_HOME/conf/[enginename]/[hostname]/context.xml.default，其中Context元素中的信息会被hostname主机下的所有web应用程序加载
3. $CATALINA\_HOME/conf/[enginename]/[hostname]/目录中所有以.xml为扩展名的文件

其中Context元素中的信息会被hostname主机下的所有web应用程序加载

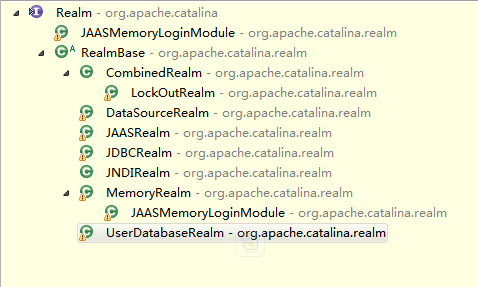
如果通过上面的步骤没有找到，那么最后要从web应用程序的/META-INF/context.xml目录中查找

## GlobalNamingResources

为 Server 定义全局 JNDI 资源。这些资源出现在 Server 的全局 JNDI 资源上下文中。这个上下文和每个 web 应用程序的 JNDI 上下文不同。在全局 JNDI 上下文中定义的资源在每个 web 应用程序的 JNDI 上下文中不可见，但是可以通过 Resource Links 来改变这种可见性。如果您要了解在 Tomcat 中如何使用 JNDI 资源可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/" \l "resource" \o "参考资料)。

## Realm

Realm组件在Engine、Host和Context中都有支持。 Realm 是一个"数据库"存储着用户名、密码和角色信息。可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/#resource)，Tomcat提供了多个实现：



Realm类似于Unix里面的group。在Unix中，一个group对应着系统的一定资源，某个group不能访问不属于它的资源。Tomcat用Realm来对不同的应用（类似系统资源）赋给不同的用户（类似group）。没有权限的用户则不能访问这个应用。

可以在任何Catalina容器（Engine，Host或者Context）中嵌套Realm元素。另外，Engine或者Host的Realm会自动被低层次的容器继承，除非被明确覆盖。

Tomcat提供多种Realm，

1、JDBCRealm，这个Realm将用户信息存在数据库里，通过JDBC获得用户信息来进行验证。

2、JNDIRealm，用户信息存在基于LDAP的服务器里，通过JNDI获取用户信息。

3、MemoryRealm，用户信息存在一个xml文件里面，上面讲的manager应用验证用户时即使用此种Realm。通过Realm我们可以方便地对访问某个应用的客户进行验证。

## Loader

Loader是web应用程序的类装载器。必须有一个类装载器按照Servlet Specification的要求从如下的位置装载类：

从web应用程序的/WEB-INF/classes目录装载

从web应用程序的/WEB-INF/lib目录中的jar文件中装载

从Catalina中装载对于所有web应用可见的资源

Loader元素应该嵌入到Context元素中，如果没有设置那么会自动创建一个默认的Loader。如果想更深入的了解Catalina实现的Loader可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/" \l "resource" \o "参考资料)。

而标准实现org.apache.catalina.loader.WebappLoader

## Manager

Manager组件只在Context中有支持。 是session管理器(session manager)，负责session的创建和维护。

Manager元素应该嵌入到Context元素中，如果没有设置那么会自动创建一个默认的Manager。

Manager 支持className和distributable两个公共属性，而标准实现 org.apache.catalina.session.StandardManager和 org.apache.catalina.session.PersistentManager还可能支持一些扩展属性。

## Access Logs

Access Logs特性在Engine、Host和Context中都有支持。

* Engine中的Access Logs记录所有Engine处理的请求的访问日志
* Host中的Access Logs记录所有Host处理的请求的访问日志
* Context中的Access Logs记录所有Context处理的请求的访问日志

一般的配置方法是在conf/server.xml文件的相关元素中加入：

<Engine ...>

...

<Valve className="org.apache.catalina.valves.AccessLogValve"

prefix="catalina\_access\_log." suffix=".txt" pattern="common"/>

</Engine>

上面的<Engine>，在Host中要被换成<Host>，在Context中要被换成<Context>。

Access Logs特性的实现是通过Tomcat的Value框架来完成的。如果您要了解这种技术的细节可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/" \l "resource" \o "参考资料)。

如果您要了解Access Log Valve设置的更多信息，可以查阅[这里](http://tomcat.apache.org/tomcat-5.5-doc/config/valve.html" \l "Access%20Log%20Valve" \o "Access Log Valve)。

## Lifecycle Listeners

Lifecycle Listeners特性在Engine、Host和Context中都有支持。这个特性使得我们可以方便的进行生命周期的管理。

值得一提的是在Tomcat的标准实现中Server和Service也支持生命周期的管理，但是在官方文档中没有显著的说明，所以没有在图中体现出来。细节可以查阅Server和Service部分。

* Engine中的Lifecycle Listeners监听该Engine的生命周期事件(Eifecycle Event)
* Host中的Lifecycle Listeners监听该Host的生命周期事件(Eifecycle Event)
* Context中的Lifecycle Listeners监听该Context的生命周期事件(Eifecycle Event)

一般的配置方法是在conf/server.xml文件的相关元素中加入：

<Engine ...>...

<Listener className="com.mycompany.mypackage.MyListener" ... >

...

</Engine>

上面的<Engine>，在Host中要被换成<Host>，在Context中要被换成<Context>。

另外，可以通过<Listener>元素为listener添加属性。

## Request Filters

Request Filters特性在Engine、Host和Context中都有支持。

* Engine中的Request Filters过滤所有Engine处理的请求
* Host中的Request Filters过滤所有Host处理的请求
* Context中的Request Filters过滤所有Context处理的请求

一般的配置方法是在conf/server.xml文件的相关元素中加入：

<Engine ...>

...

<Valve className="org.apache.catalina.valves.RemoteHostValve"

allow="\*.mycompany.com,www.yourcompany.com"/>

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

deny="192.168.1.\*"/>

...

</Engine/>

上面的<Engine>，在Host中要被换成<Host>，在Context中要被换成<Context>。

Request Filters特性的实现是通过Tomcat的Value框架来完成的。如果您要了解这种技术的细节可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/" \l "resource" \o "参考资料)。

如果您要了解Remote Address Filter设置的更多信息，可以查阅[这里](http://tomcat.apache.org/tomcat-5.5-doc/config/valve.html" \l "Remote%20Address%20Filter" \o "Remote Address Filter)。

如果您要了解Remote Host Filter设置的更多信息，可以查阅[这里](http://tomcat.apache.org/tomcat-5.5-doc/config/valve.html" \l "Remote%20Host%20Filter" \o "Remote Host Filter)。

## Single Sign On

Single Sign On特性只在Host中都有支持。 在一些时候，特别是在Portal环境下，可能会希望当用户访问一个虚拟主机下的多个web应用时只登陆一次，即所谓的单点登陆。

一般的配置方法是在conf/server.xml文件的相关元素中加入：

<Host name="localhost" ...>

...

<Valve className="org.apache.catalina.authenticator.SingleSignOn" debug="0"/>

...

</Host>

Single Sign On操作遵循下面的规则：

* 该虚拟主机下的所有Web应用程序必须共享同一个Realm。在实践中通常通过为Host或(对应的Engine)嵌入Realm而不是为每一个Context嵌入相同的Realm来实现。
* 在用户访问该虚拟主机下的所有Web应用程序中的非保护资源时不需要验证。
* 在用户访问该虚拟主机下的所有Web应用程序中的保护资源时需要验证。
* 一旦被验证，就会对该虚拟主机下的所有Web应用程序有效，而不用每个应用程序单独验证。
* 如果用户从一个Web应用中退出，那么所有Web应用程序中的session都会失效。
* 使用SSO需要客户环境支持cookies。

Single Sign On特性的实现是通过Tomcat的Value框架来完成的。如果您要了解这种技术的细节可以查阅[参考资料](http://www.solol.org/technologic/java/j-tomcatarch/" \l "resource" \o "参考资料)。

如果您要了解Single Sign On设置的更多信息，可以查阅[这里](http://tomcat.apache.org/tomcat-5.5-doc/config/valve.html" \l "Single%20Sign%20On%20Valve" \o "Single Sign On Valve)。

# Tomcat [launch](app:ds:launch)

org.apache.catalina.startup.Bootstrap Main method, used for testing only.

org.apache.catalina.startup.Catalina

# Coyote

coyote是tomcat的Connector框架的名字，简单说就是coyote来处理底层的socket，并将http请求、响应等字节流层面的东西，包装成Request和Response两个类（这两个类是tomcat定义的，而非servlet中的ServletRequest和ServletResponse），供容器使用； 所以，coyote本质上是为tomcat的容器提供了对底层socket连接数据的封装，以Request类的形式，让容器能够访问到底层的数据。

tomcat默认支持的协议有http1.1和ajp，

JIOEndpoint，还有AprEndpoint、NioEndpoint

# SSI

SSI 全称（ Server Side Includes ）是在 HTML 静态页面中使用的指令。

Tomcat 服务器的 SSI 支持的实现方式和 Apache 里面的 SSI 指令一样。Tomcat实现使用了一个Servlet和一个filter：

org.apache.calalina.ssi.SSIFilter

org.apache.catalina.ssi.SSIServlet

路径解析里面都要添加映射规则“ \*.shtml ”，而且 Filter 配置的时候需要在映射规则里面添加能够执行的 mime type ， contentType 的初始化参数。

在Tomcat 里面默认是禁用了 SSI 的。

# Tomcat utils

Tomcat的连接池、线程池等直接和socket，由org.apache.tomcat.util.net包的类去完成。

同时，为了能让我们编写的servlet能够得到ServletRequest，tomcat使用了facade模式，将比较底层、低级的Request包装成为ServletRequest（这一过程通常发生在Wrapper容器一级），这也是为很多人津津乐道的tomcat对设计模式的一个巧妙的运用，具体过程将会在以后讨论。

# Jasper 2 JSP Engine

编译JSP文件，可以也在ant 中做为一个target使用。

# *Appendix*

<http://blog.csdn.net/wangchengsi/article/details/2973012>

http://blog.csdn.net/wangchengsi/article/details/2995536

使用 CAS 在 Tomcat 中实现单点登录

<http://www.ibm.com/developerworks/cn/opensource/os-cn-cas/>

Apache HTTP Server 与 Tomcat 的三种连接方式介绍(JK ,http\_proxy ,ajp\_proxy)

<http://www.ibm.com/developerworks/cn/opensource/os-lo-apache-tomcat/>

Tomcat中SSI的配置

[*http://edu.codepub.com/2009/0619/6664.php*](http://edu.codepub.com/2009/0619/6664.php)

[图解 Tomcat 体系结构](http://www.blogjava.net/nokiaguy/archive/2008/10/12/233914.html)

*http://www.blogjava.net/nokiaguy/archive/2008/10/12/233914.html*