**过滤器4：Ad Hoc验证过滤器**

Tomcat 5和几乎所有符合Servlet 2.4标准的容器都具有广泛的身份验证和授权支持，因此理论上应该几乎不需要实现自己的身份验证过滤器。但实际上，几乎总是有空间在所选资源上应用ad hoc认证过滤器，而不会影响应用程序的其余部分或涉及设置JDBC领域的开销。

您应始终分析手头的问题，看看是否可以通过服务器的本机身份验证支持更好地解决问题。在您需要对所选资源进行简单，临时保护的情况下，AdHocAuthenticate过滤器可能是最佳选择。

此过滤器的操作非常简单。它会在客户端浏览器上触发基本身份验证几乎所有已知的浏览器，包括最早的版本，都支持基本身份验证。它的工作原理如下：

1.客户端尝试访问受保护资源。

2.服务器检查客户端的请求，以确定“授权”标头中是否有任何授权数据。

3.如果未找到授权数据，服务器将发回HTTP状态代码401（未授权访问）和带WWW-authenticate的标头：BASIC realm = <realm>，其中realm是将显示给客户端的文本字符串。

4.客户端弹出登录屏幕，用户应在其中输入用户名和密码。

5.客户端使用简单的base64编码对用户名和密码进行编码，并将两者都发送到服务器。

6.服务器检查客户端请求以确定是否存在任何授权数据，必要时解码base64编码的密码。如果没有授权数据，则返回步骤3。

7.服务器验证密码并允许或拒绝访问。

注意基本身份验证不是很安全，因为base64编码很容易被破译。但是，对于只需要保护资源免受临时访问的应用程序，通常就足够了。有关不同类型身份验证的更多详细信息，请参阅第11章。

AdHocAuthenticate过滤器识别两个密码：一个用于“常规”用户，另一个用于特权“金牌成员”用户。两个密码都配置为过滤器的初始参数。如果用户使用“gold member”密码登录，则会创建一个布尔属性并将其附加到请求。您将在稍后（在管道处理过滤器部分中）了解如何使用此属性。现在，让我们关注此过滤器的身份验证操作。

**AdHocAuthenticateFilter类**

以下是对该类源代码的简要分析：

   package com.apress.projsp20.ch10;

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.Map;

import sun.misc.\*;

无需在此过滤器中包含响应或请求。请注意将用于保存两个只读密码的实例变量：adhocPassword和adhocGoldPassword。这些变量由容器通过init（）方法从web.xml值初始化：

public final class AdHocAuthenticateFilter implements Filter {

private FilterConfig filterCofig = null;

private String adhocPassword = null;

private String adhocGoldPassword = null;

public void doFilter(ServletRequest request, ServletResponse response,FilterChain chain)

throws IOException, ServletException {

if (filterConfig == null) return;

您将请求和响应强制转换为其HTTP servlet版本，以访问和操作与其关联的标头：

    HttpServletRequest myReq = (HttpServletRequest) request;

HttpServletResponse myResp = (HttpServletResponse) response;如果您没有找到授权数据，请执行基本身份验证请求：

       String authString = myReq.getHeader("Authorization");

if (authString == null) {

myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");

myResp.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

return;

} else { // authenticate

如果找到授权数据，则解码用户名和密码。以下子字符串（6）跳过授权标头的常量字符串Basic，到达base64编码的用户名和密码的开头：

        BASE64Decoder decoder = new BASE64Decoder（）;

        String enString = authString.substring（6）;

        String decString = new String（decoder.decodeBuffer（enString））;

        int idx = decString.indexOf（“：”）;

        String uid = decString.substring（0，idx）;

        String pwd = decString.substring（idx + 1）;

您可以调用方法externalGoldAuthenticate（）和externalAuthenticate（）来分别对“gold member”用户和“常规”用户执行实际身份验证。 在生产中，可以通过外部服务器进行身份验证。 成功的身份验证允许访问受保护资源。 此外，“gold member”身份验证将导致将ahem，goldmember布尔属性附加到请求。 验证失败将导致登录对话框在客户端的浏览器上再次弹出：

if (externalGoldAuthenticate(uid,pwd)) {

request.setAttribute("goldmember", new Boolean(true));

} else {

if (!externalAuthenticate(uid,pwd)) {

myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");

myResp.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

return;

} // of if

} // of else

} // of outer else

**过滤器5：请求处理管道中的过滤器**

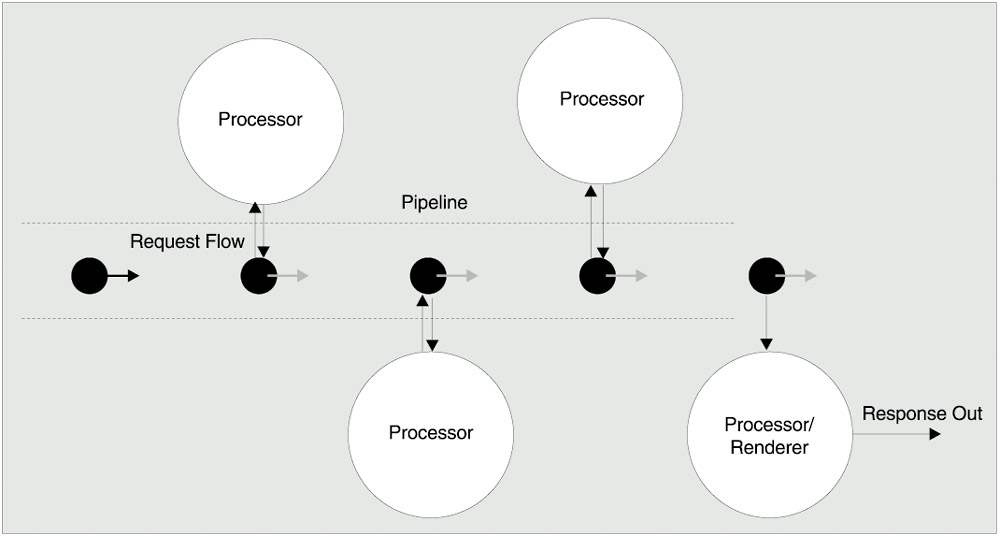
到目前为止，本章的重点是过滤器，它可以控制请求流（停止或让它通过）或生成对请求的响应。由于它们的应用领域可能很广，这些滤波器设计风格仅部分涵盖了滤波器的潜在应用范围。事实上，这些过滤器直接负责响应HTML页面的最终外观的某些部分，这常常使得它们难以组合（以增值组件方式链接）。这是因为，根据定义，它们特定于它们直接生成的页面输出。

管道数据处理模型提供了一种新的过滤器样式，我们的第五个和最后一个过滤器就是例证。此模型的第一个启用程序是servlet <filter-mapping>定义的新<dispatcher>子元素。新元素使过滤器能够参与管道请求处理模型的每个阶段。在这个模型中，过滤器本身就是通过管道传输请求的真正的处理器

**了解管道模型**

流水线数据处理模型有许多不同的名称。 它通常被认为是模型 - 视图 - 控制器（MVC）Web应用程序设计的支持元素，有时它被称为应用程序设计的推送模型，与更传统的拉模型相反。

这个想法很简单，但你必须超越传统的Web应用程序智慧来思考大局。 考虑一下这个模型的例子：



请求进入系统并沿管道穿梭。当它遍历管道时，一系列处理器可以访问请求的内容。每个处理器对请求执行一些任务，然后它附加一些新属性或修改现有属性作为该工作的结果。请求保持完整，直到它到达管道的最后阶段，其中称为渲染器的特殊处理器检查对请求执行的所有工作并生成最终响应（渲染器在MVC范例中也称为视图组件，因为它们本身就是负责最终呈现给用户）。你可以看出为什么这通常被称为“推”模型;数据属性由处理器提取，随请求一起沿管道推送，并仅在最后阶段呈现。

管道模型的另一个常见类比是输送带或装配线，其中每个处理器沿着带或线镜像工人（或机器人）。

管道模型的一些非常理想的属性如下：

请求在遍历管道时保持不变，只在附加的属性（有时称为装饰器）上执行工作。

数据管理，业务逻辑和表示逻辑可以干净地分成不同的处理器和渲染器。

处理器可以设计为完全可组合（即以任何顺序链接）或高度专业化（仅适用于由其他处理器创建或修改的特定属性）。

多个渲染器可以组成最终输出（例如通过XSLT将数据转换为XML到HTML）。

请求的状态随着请求通过管道传播。处理器和渲染器根本不跟踪依赖于请求的状态。这使得可以在必要时在多个物理服务器之间复制或穿梭请求及其状态。

通常，使用管道模型设计请求处理逻辑提供以下内容：

清洁，组件化的处理器和易于重复使用的渲染器

应用程序中数据管理，业务逻辑和表示逻辑之间的明确分离

更易于维护的应用程序

更适应不断变化的业务需求的应用程序

可通过容器技术和硬件进步扩展的强大应用程序

性能高度可优化的Web应用程序

最后一点可能不会立即明显，但管道中任何不依赖于其他结果的处理器原则上可以同时执行（例如，可能在两个物理处理器上）和任何具有结果独立的处理器组其他人也可以同时执行。

过去，容器实施的限制阻碍了使用清洁的管道设计模型。 JSP 2.0和Servlet 2.4具有新功能，使创建流水线应用程序的梦想更接近现实。

**将过滤器插入管道**

使用<dispatcher>子元素可以将过滤器插入到请求处理管道中。回想一下，<dispatcher>元素现在允许过滤器拦截请求调度程序的forward（）和include（）调用。这些是Servlet 2.3容器以前无法使用的处理管道中的其他位置，其中过滤器可以为您工作。您已经在最后一章中了解了这种机制是如何工作的，所以让我们在这里看到它的实际应用。

首先，您将重新访问上一章中看到的SimpleFilter类。它只是在日志中写了几行，让你知道它已被调用。现在，您将更改它以处理通过管道传输的请求。

现在只需更改附加属性，而不是直接生成日志输出。修改将查看请求中是否存在名为MsgOut的属性。如果它确实存在，那么代码将附加“：SimpleFilter”;否则，MsgOut将设置为“SimpleFilter”（没有前导冒号）。过滤器（处理器）第一次对请求进行操作时，附加的MsgOut属性将设置为“SimpleFilter”。第二次之后，它将包含“SimpleFilter：SimpleFilter”，并在第三次之后包含“SimpleFilter：SimpleFilter：SimpleFilter”。

最终，MsgOut属性将显示在HTML页面上，因此您可以看到此过滤器在通过管道传输时对特定请求的操作次数。你会在中找到代码：

com.apress.projsp20.ch10.SimpleFilter class:

package com.apress.projsp20.ch10;

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public final class SimpleFilter implements Filter {

private FilterConfig filterConfig = null;

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain)

throws IOException, ServletException {

if (filterConfig == null) return;

filterConfig.getServletContext().log("in SimpleFilter");

Object curVal = request.getAttribute("MsgOut");

if (curVal == null) {

request.setAttribute("MsgOut", new String("SimpleFilter"));

} else {

request.setAttribute("MsgOut", (String) curVal + " :SimpleFilter");

}

chain.doFilter(request, response);

filterConfig.getServletContext().log("leaving SimpleFilter");

}

您还将修改FindProd.jsp文件（管道中的渲染器处理器）以显示新属性。 JSP 2.0的EL非常适合创建渲染器 - 即，从附加属性值呈现HTML的JSP页面。 你将这个修改过的文件命名为Sub.jsp：

<html>

<head></head>

<body>

<h1>You have accessed this page from the ${param.DEPT} department!</h1>

${param.MsgForwarder}<br/>

${requestScope.MsgOut}

</body>

</html>

附加到请求的MsgOut属性在此处使用EL显示。 请注意，还会显示一个名为MsgForwarder的参数。 稍后当您使用请求调度程序的forward（）操作时，您将看到使用此参数。

**默认REQUEST-Only筛选：Servlet 2.3兼容性**

要指定仅对直接来自容器外部的请求执行过滤，可以在过滤器的<filter-mapping>元素中添加以下<dispatcher>元素：

<dispatcher>REQUEST</dispatcher>

例如，您可能在web.xml中具有以下内容：

<filter>

<filter-name>Simple Push Filter</filter-name>

<filter-class>com.apress.projsp20.ch10.SimpleFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>Simple Push Filter</filter-name>

<url-pattern>/jsp/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

</filter-mapping>

现在，如果您打开URL http：// localhost：8080 / filters2 / jsp / Sub.jsp？DEPT = Accounting，您将看到如下页面：



请注意页面上的MsgOut属性的EL呈现 - 您知道过滤器已被调用一次。 这种行为就是在Servlet 2.4标准之前使用过滤器支持的所有行为。 如果出于兼容性原因未指定任何<dispatcher>子元素，那么它也是Tomcat 5的默认行为。

**INCLUDE-Only Filtering**

使用Tomcat 5和Servlet 2.4，您可以指定您的过滤器仅适用于包含的请求。 通过将web.xml更改为此来尝试此操作：

<filter-mapping>

<filter-name>Simple Push Filter</filter-name>

<url-pattern>/jsp/\*</url-pattern>

<dispatcher>INCLUDE</dispatcher>

</filter-mapping>

重启Tomcat并尝试通过以下URL直接访问Sub.jsp：

<http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting>

请注意，MsgOut属性不存在，表示结果页面尚未通过SimpleFilter。

通过在<dispatcher>子元素中指定INCLUDE，您说过滤器应该仅映射到包含的请求。 要查看此操作，请查看jsp / Master.jsp文件：

<html>

<head></head>

<body>

<h1>First Inclusion</h1>

<jsp:include page="/jsp/Sub.jsp" flush="true">

<jsp:param name="DEPT" value="Accounting"/>

</jsp:include>

<hr/>

<h1>Second Inclusion</h1>

<jsp:include page="/jsp/Sub.jsp" flush="true">

<jsp:param name="DEPT" value="Engineering"/>

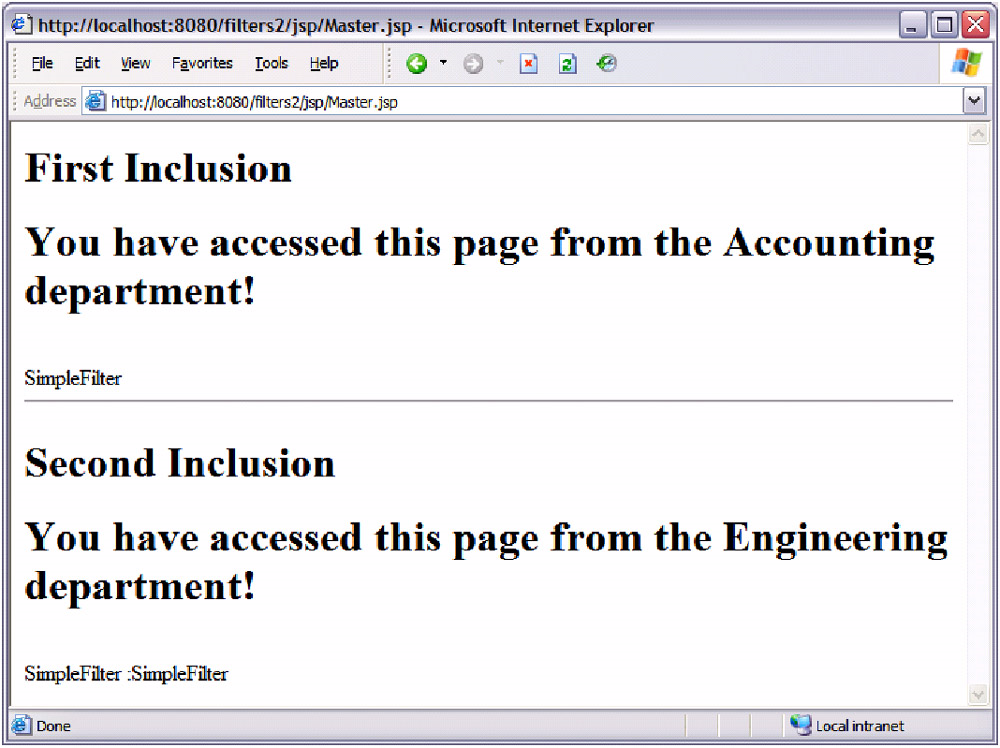
</jsp:include>

</body>

</html>

此JSP文件对请求调度程序进行两次调用，每次都包含您一直使用的/jsp/Sub.jsp。 这两个include（）动作的输出合并为输出响应。 请注意，您还要为每个包含提供不同的DEPT参数。

现在导航到http：// localhost：8080 / filters2 / jsp / Master.jsp，结果页面应与以下内容匹配：



请注意MsgOut属性值 - 它反映了SimpleFilter的操作。 在第二次包含之后，第一次包含Sub.jsp和“SimpleFilter：SimpleFilter”之后，该属性为“SimpleFilter”。 该请求已由SimpleFilter操作两次，每次包含在Master.jsp中。

**FORWARD-Only Filtering**

要使过滤器仅对转发的请求起作用，请按如下所示修改web.xml文件：

<filter-mapping>

<filter-name>Simple Push Filter</filter-name>

<url-pattern>/jsp/\*</url-pattern>

<dispatcher>FORWARD</dispatcher>

</filter-mapping>

由于在<disptacher>元素中指定了FORWARD，因此只有转发的请求才会传递给过滤器。 要查看此操作，请查看jsp / Forwarder.jsp文件：

<jsp:forward page="/jsp/Sub.jsp">

<jsp:param name="DEPT" value="Accounting"/>

<jsp:param name="MsgForwarder" value="Forwarded from forwarder.jsp"/>

</jsp:forward>

此JSP页面只是将请求转发给Sub.jsp，并设置DEPT和MsgForwarder参数。 这两个参数都使用EL显示在Sub.jsp中。

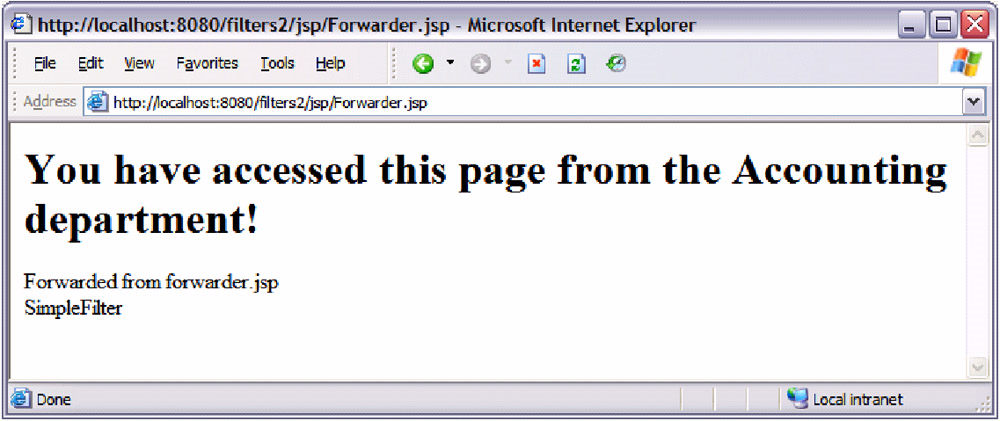
重新加载应用程序或重新启动Tomcat。 现在依次尝试以下网址：

http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting

http://localhost:8080/filters2/jsp/Master.jsp

http://localhost:8080/filters2/jsp/Forwarder.jsp

对于前两个URL，请注意不再过滤直接请求和包含的请求。 第三个URL应该产生类似于这样的页面：



您可以从此实验中看到，只过滤了转发的请求。

**Combining Dispatcher Actions**

当然，您可以使用多个<dispatcher>元素来指示要应用的过滤器的多个位置。 例如：

<filter-mapping>

<filter-name>Simple Push Filter</filter-name>

<url-pattern>/jsp/\*</url-pattern>

<dispatcher>FORWARD</dispatcher>

<dispatcher>INCLUDE</dispatcher>

<dispatcher>REQUEST</dispatcher>

</filter-mapping>

现在，尝试前三个网址，您将看到以下内容：

URL 1：单独请求时，过滤器在Sub.jsp上处于活动状态。

URL 2：过滤器在Master.jsp PLUS中处于活动状态，每个包含的Sub.jsps。

URL 3：过滤器在Forwarder.jsp上处于活动状态PLUS转发的Sub.jsp。