# Servlet知识点

## Servlet过滤器

### 什么是Servlet过滤器

过滤器是在数据交互之间过滤数据的中间组件，独立于任何平台或者 Servlet 容器。

Servlet过滤器可以应用在客户机和 servlet 之间、servlet 和 servlet 或 JSP 页面之间，以及所包括的每个 JSP 页面之间。  
二、实现一个 Servlet 过滤器

 Servlet过滤器 API 包含javax.servlet包中的 3 个接口，分别是 Filter 、 FilterChain 和 FilterConfig 。  
 实现一个 Servlet 过滤器的确要经历三个步骤。

  1、编写 Servlet 过滤器实现类。  
  2、配置 Servlet 过滤器。把该过滤器添加到 Web 应用程序中（通过在 Web 部署描述符 /web.xml 中声明它）;  
  3、部署 Servlet 过滤器。把过滤器与应用程序一起打包并部署它;

 Servlet容器对部署描述符中声明的每一个过滤器，只创建一个实例（或实例池）。  
 与Servlet类似，容器将在同一个过滤器实例上运行多个线程来同时为多个请求服务，因此，开发过滤器时，也要注意线程安全的问题。

三、编写过滤器实现类

 1、Filter接口：所有的Servlet过滤器类都必须实现javax.servlet.Filter接口  
  a、init(FilterConfig)：  
   这是Servlet过滤器的初始化方法，Servlet容器创建Servlet过滤器实例后将调用这个方法。在这个方法中可以读取web.xml文件中Servlet过滤器的初始化参数。

  b、doFilter(ServletRequest,ServletResponse,FilterChain)：  
   这个方法完成实际的过滤操作，当客户请求访问于过滤器关联的URL时，Servlet容器将先调用过滤器的doFilter方法。FilterChain参数用于访问后续过滤器。

  c、destroy()：  
   Servlet容器在销毁过滤器实例前调用该方法，这个方法中可以释放Servlet过滤器占用的资源。

 2、过滤器实现类创建步骤（过滤器实现类生命周期）：  
  a．实现javax.servlet.Filter接口。  
  b．初始化：实现init方法，读取过滤器的初始化参数。  
  c．过滤：实现doFilter方法，完成对请求或响应的过滤。  
  d．转发或阻塞：调用FilterChain接口对象的doFilter方法，向后续的过滤器传递请求或响应。  
  e．析构：destroy方法销毁过滤器，释放过滤器占用的资源。

四、配置 Servlet 过滤器（在web.xml中配置）

 过滤器通过 web.xml 文件中的两个 XML 标签来声明：   
  1、<filter> : 定义过滤器的名称，并且声明过滤器实现类和 init() 参数。   
   <filter-name> : 指定过滤器的名字;  
   <filter-class> : 指定过滤器类的类名,包括类的路径;  
   <init-param> : 为过滤器实例提供初始化参数，可以有多个;

  2、<filter-mapping> : 将过滤器与 servlet 或 URL 模式相关联。   
   <filter-name> :  指定过滤器的名字,与<filter>中的子元素<filter-name>相对应;  
   <url-pattern> :  指定和过滤器关联的URL，为”/\*”表示所有URL;

  3、<filter-mapping>元素还可以包含0到4个<dispatcher>，指定过滤器对应的请求方式，  
    可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST.

 REQUEST  
     当用户直接访问页面时，Web容器将会调用过滤器。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器就不会被调用。  
   INCLUDE  
   如果目标资源是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用。除此之外，该过滤器不会被调用。  
   FORWARD  
   如果目标资源是通过RequestDispatcher的forward()方法访问时，那么该过滤器将被调用，除此之外，该过滤器不会被调用。  
   ERROR  
     如果目标资源是通过声明式异常处理机制调用时，那么该过滤器将被调用。除此之外，过滤器不会被调用。

 在web.xml中配置Servlet和Servlet过滤器，应该先声明过滤器元素，再声明Servlet元素。  
 两个或更多个过滤器应用到同一个资源，按照它们在配置文件中显示的先后次序调用它们。

 例子1：单个过滤器配置：容器将其应用于所有接收的请求  
 <filter>      
  <filter-name>FilterName</filter-name>      
  <filter-class></filter-class>    
 </filter>    
 <filter-mapping>      
  <filter-name>FilterName</filter-name>      
  <url-pattern>/\*</url-pattern>    
 </filter-mapping>

  例子2：过滤器应用到特定目录或资源（文件）的配置：此容器只有在接收到对 /mydocs 目录中的资源的请求时才会应用该过滤器。  
   <filter>      
  <filter-name>FilterName</filter-name>      
  <filter-class>packageName.FilterName</filter-class>    
 </filter>    
 <filter-mapping>      
  <filter-name>FilterName</filter-name>      
  <url-pattern>/mydocs/\*</url-pattern>    
 </filter-mapping>

 例子3:定义一个过滤器链：两个或更多个过滤器应用到同一个资源，按照它们在配置文件中显示的先后次序调用它们。  
   <filter>      
  <filter-name>FilterOne</filter-name>      
  <filter-class>packageName.FilterOne</filter-class>    
 </filter>    
 <filter-mapping>      
  <filter-name>FilterOne</filter-name>      
  <url-pattern>/\*</url-pattern>    
 </filter-mapping>

   <filter>      
  <filter-name>FilterTwo</filter-name>      
  <filter-class>packageName.FilterTwo</filter-class>    
 </filter>    
 <filter-mapping>      
  <filter-name>FilterTwo</filter-name>      
  <url-pattern>/mydocs/\*</url-pattern>    
 </filter-mapping>

五、部署 Servlet 过滤器

 只需把过滤器类和其他 Web 组件类包括在一起，把 web.xml 文件（连同过滤器定义和过滤器映射声明）放进 Web 应用程序结构中，servlet 容器将处理之后的其他所有事情。

六、Servlet 过滤器实现注意事项

 1．由于Filter、FilterConfig、FilterChain都是位于javax.servlet包下，并非HTTP包所特有的，  
 所以ServletRequest、ServletResponse在使用前都必须先转换成HttpServletRequest、HttpServletResponse再进行下一步操作。

 2．在web.xml中配置Servlet和Servlet过滤器，应该先声明过滤器元素，再声明Servlet元素。

 3．如果要在Servlet中观察过滤器生成的日志，应该确保在server.xml的localhost对应的<host>元素中配置如下<logger>元素：  
  <Logger className = “org.apache.catalina.logger.FileLogger”  
   directory = “logs”prefix = “localhost\_log.”suffix=”.txt”  
   timestamp = “true”/>

七、过滤器的实现方式

 过滤器实现方式在不保证功能前提下，从性能角度考虑有如下先后顺序：Decorator或Proxy模式；AOP拦截器。

 1，标准Servlet控制器  
  包装了请求和响应对象。  
  web容器来充当过滤管理器（FilterManager）和过滤链（FilterChain）来管理和协调过滤器。  
    
  问题：功能覆盖范围上，也是一个scope，如果功能需求要为某个类的方法实现实现过滤，使用一个Servlet Filter这样过滤器实现。  
   但是它对所有的Servlet请求都进行过滤，这无疑范了杀鸡取卵的错误，会造成系统性能上的损失。

 2，使用Decorator模式来定制过滤器  
  如果过滤器是业务逻辑的一部分，而且在设计时，我们可以确定这些过滤器，进行特定指定的拦截。  
  Decorator模式在点上针对性相当强，特别在这个点上有一系列过滤器需要实现时。  
    
  如果某个过滤功能是很多类都需要的，会形成很多Decorator附加类，造成点形成面的情况，则升级使用AOP拦截器。

  优点：能够动态地为过滤器扩展功能。  
   也可以使用FilterManager和FilterChain过滤器链负责协调和管理过滤处理，这样单独的过滤器就不用和其他过滤器直接通信了。

  问题：不能以一种标准的可移植的方式支持对请求和响应对象的包装，而且不能够修改请求对象。  
   缺乏完善的缓存机制。当过滤器要控制输出流的时候，还必须引入某种形式的缓存机制。

 3，使用模板方法模式来定制过滤器。  
  可以和其他方法混用。  
  优点：这种方式是基于标准的过滤器的，基本过滤器作为一个基类，封装了过滤器API的所有细节。  
   专注于预处理和后处理的所有逻辑。  
   基本过滤器声明了每个过滤器要完成的方法，每个过滤器子类来定义这些定义方法。由超类来控制子类的控制流程。  
   
 4，使用AOP框架拦截器。  
  当功能不是针对某个具体类或方法（方法权限除外），而是一系列类，使用动态AOP拦截器，性能损耗也是值得的，而且是必要的。  
    
八、过滤器的应用场合

简洁的说法：  
 1．认证过滤:对用户请求进行统一认证。  
   
 2．登录和审核过滤：对用户的访问请求进行审核和对请求信息进行日志记录。

 3，数据过滤：对用户发送的数据进行过滤，修改或替换。

 4．图像转换过滤 ：转换图像的格式。

 5．数据压缩过滤 ：对请求内容进行解压，对响应内容进行压缩。

 6．加密过滤 ：对请求和响应进行加密处理。

 7．令牌过滤 ：身份验证

 8．资源访问触发事件过滤 ：

 9．XSL/T过滤

 10．Mime-type过滤

复杂的说法：  
在适合使用装饰过滤器模式或者拦截器模式的任何地方，都可以使用过滤器：

加载：对于到达系统的所有请求，过滤器收集诸如浏览器类型、一天中的时间、转发 URL 等相关信息，并对它们进行日志记录。

性能：过滤器在内容通过线路传来并在到达 servlet 和 JSP 页面之前解压缩该内容，然后再取得响应内容，并在将响应内容发送到客户机机器之前将它转换为压缩格式。

安全：过滤器处理身份验证令牌的管理，并适当地限制安全资源的访问，提示用户进行身份验证和/或将他们指引到第三方进行身份验证。  
 过滤器甚至能够管理访问控制列表（Access Control List，ACL），以便除了身份验证之外还提供授权机制。  
 将安全逻辑放在过滤器中，而不是放在 servlet 或者 JSP 页面中，这样提供了巨大的灵活性。  
 在开发期间，过滤器可以关闭（在 web.xml 文件中注释掉）。  
 在生产应用中，过滤器又可以再次启用。此外还可以添加多个过滤器，以便根据需要提高安全、加密和不可拒绝的服务的等级。

会话处理：将 servlet 和 JSP 页面与会话处理代码混杂在一起可能会带来相当大的麻烦。  
 使用过滤器来管理会话可以让 Web 页面集中精力考虑内容显示和委托处理，而不必担心会话管理的细节。

XSLT 转换：不管是使用移动客户端还是使用基于 XML 的 Web 服务，无需把逻辑嵌入应用程序就在 XML 语法之间执行转换的能力都绝对是无价的。

九、MVC 体系结构中的Servlet过滤器

 不管过滤器处于什么位置，过滤器在处理流中的应用都是相同的。过滤器旨在扩充 MVC 体系结构的请求/响应处理流。  
 从 MVC 的观点看，调度器组件（它或者包括在控制器组件中，或者配合控制器组件工作）把请求转发给适当的应用程序组件以进行处理。  
 这使得控制器层成为包括 Servlet 过滤器的最佳位置。通过把过滤器放在控制器组件本身的前面，过滤器可以应用于所有请求，  
 或者通过将它放在控制器/调度器与模型和控制器之间，它可以应用于单独的 Web 组件。

十、应用示例或说明

 1，使用过滤器认证用户：   
 每个过滤器也可以配置初始化参数，可以将不需要过滤的地址配置到这个Filter的配置参数中，  
 过滤时，如果请求地址在配置参数中，则放行，这样就避免了在程序中硬编码。  
 每个Filter中初始化时，都可以得到配置对象，在Filter中配置二个不需要过滤的地址，一个是登陆页面，一个是执行登陆认证的servlet；

 2，登录和审核过滤的示例：使用 servlet 过滤器来控制终端用户对应用程序特性的访问：  
 通过显示基于用户角色的用户界面来控制对应用程序特性的访问。企业用户能够访问特定页面，但个人用户不能访问这样的页面。  
 应用一个过滤器来处理用户请求并返回合适的页面。通过使用过滤器，您能够向一个基于 JSP 的应用程序添加这种类型的访问控制而无需更改现有的代码。   
   
 这种基于过滤器的访问控制模型也很灵活，因为把用户角色映射到特定 JSP 的数据存储在一个 XML 文件中。  
 因此，您可以修改映射而不用修改应用程序 — 不需要重新编译或重新部署。

实施：  
     每当用户从一个包含 /controllerservlet模式的 URI 请求资源时就会调用AccessControlFilter的实例  
 当访问控制过滤器被初始化后，从映射文件读取数据，而 AccessControlFilter.doFilter 方法负责处理过滤事务。  
 在获取了用户角色和所请求页面的 URL 之后，doFilter 把这些值与映射数据对比。  
 如果所请求的页面对用户角色是合适的，那么代码将调用 chain.doFilter 方法来调用该页面并继续正常的处理。  
 否则，代码将在 chain.doFilter 之前调用 request.setAttribute，结果是 Controller Servlet 使用户重新进入到登录页面。  
   
相关文件：  
1、web.xml —— Web 部署描述符文件  
 <filter>      
  <filter-name>AccessControlFilter</filter-name>      
  <filter-class>oracle.otnsamples.ibfbs.control.AccessControlFilter</filter-class>        
 </filter>      
 <filter-mapping>       
  <filter-name>AccessControlFilter</filter-name>       
  <url-pattern>/controllerservlet</url-pattern>    
 </filter-mapping>  
      
2、Control.xml —— 描述事件、用户角色和 JSP 联系的xml文件

  <Event>  
     <Name>BUYSTOCK</Name>  
     <Class>oracle.otnsamples.ibfbs.trademanagement.helper.TradeManagementHelper</Class>  
     <Method>buyStock</Method>  
     <Screen>jsps/BuyStock.jsp</Screen>

     <Roles>  
        <Role>USER</Role>  
     </Roles>   
  </Event>   
   ...  
  <Event>  
     <Name>CORPUPLOAD</Name>  
     <Class></Class>  
     <Method></Method>  
     <Screen>jsps/CorporateUpload.jsp</Screen>  
       
  <Roles>  
       <Role>CORP</Role>  
     </Roles>   
  </Event>   
   ...  
  <Event>  
     <Name>CONFIGNEWSUPLOAD</Name>  
     <Class>oracle.otnsamples.ibfbs.admin.helper.AdminHelper</Class>  
     <Method>configNewsUpload</Method>

     <Screen>jsps/UploadData.jsp</Screen>  
     <Roles>  
        <Role>ADMIN</Role>  
     </Roles>   
  </Event>  
   ...  
  <Event>  
     <Name>LOGIN</Name>  
     <Class>oracle.otnsamples.ibfbs.usermanagement.helper.UserManagementHelper</Class>  
     <Method>checkPassword</Method>  
     <Screen>jsps/MyHome.jsp</Screen>

     <Roles>  
        <Role>DEFAULT</Role>  
        <Role>USER</Role>  
        <Role>CORP</Role>  
        <Role>ADMIN</Role>  
     </Roles>   
  </Event>

   
3、AccessControlFilter.java —— 过滤器类

 //读取Control.xml,对用户的权限做检查

 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)                         
  throws IOException, ServletException {   
     
  HttpSession session = ((HttpServletRequest) request).getSession();      
  String eventName = request.getParameter("EVENTNAME");      
  if (eventName != null && urlMap != null ) {        
   String role = (String) session.getAttribute("ROLE");        
   if (role == null) {  
    role = "DEFAULT";        
   }

   URLMapping event = (URLMapping) urlMap.get(eventName);        
     
   if ((event != null) && (event.getRoles() != null) && (event.getRoles().length > 0)) {          
    // New session so not logged in yet. Redirect to login page          
    if (session.isNew())  {          
     request.setAttribute("EVENTNAME", "FIRSTPAGE");   
    }         
    // If invalid access, redirect to login page          
    else  if (!event.isValidRole(role))  {          
     request.setAttribute("EVENTNAME", "LOGINPAGE");   
    }       
   }  
      
  }else {        
   request.setAttribute("EVENTNAME", "FIRSTPAGE");      
  }      
  // The privileges are sufficient to invoke this URL, continue normal      
  // processing of the request      
  chain.doFilter(request, response);    
 }

 4、对请求信息进行日志记录的示例

 public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)throws ServletException, IOException {  
       //把ServletRequest对象构造为HttpServletRequest

   //从请求中提出需要的进行日志记录的信息  
       String url = req.getRequestURI();

       HttpSession so = req.getSession();  
       String canLog = (String)so.getAttribute(url);

       //如果第一次访问该页面，就进行日志处理  
       if (canLog == null) {  
        so.setAttribute(url, "Y");  
        doLog();  
       }  
       chain.doFilter(request,response);  
  }

5、用过滤器来解决客户端和服务器端编码一致，防止中文乱码的问题。

  public class CharacterEncodingFilter implements Filter {  
    protected FilterConfig filterConfig = null;  
    protected String encoding = "";

    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)   
     throws IOException, ServletException {

           if(encoding != null) {  
             servletRequest.setCharacterEncoding(encoding);  
    }  
      filterChain.doFilter(servletRequest, servletResponse);  
    }

    public void destroy() {  
      filterConfig = null;  
      encoding = null;  
   }

       public void init(FilterConfig filterConfig) throws ServletException {  
      this.filterConfig = filterConfig;  
           this.encoding = filterConfig.getInitParameter("encoding");

    }  
  }

 6、使 Browser浏览器 不缓存页面的过滤器

  public class ForceNoCacheFilter implements Filter {

   public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)   
    throws IOException, ServletException {  
   
    ((HttpServletResponse) response).setHeader("Cache-Control","no-cache");  
      ((HttpServletResponse) response).setHeader("Pragma","no-cache");  
      ((HttpServletResponse) response).setDateHeader ("Expires", -1);

      filterChain.doFilter(request, response);  
    }

  }

7、用于检测用户是否登陆的过滤器，如果未登录，则重定向到指的登录页面

  /\*\*  
   \* 用于检测用户是否登陆的过滤器，如果未登录，则重定向到指的登录页面<p>  
   \* 配置参数<p>  
   \* checkSessionKey 需检查的在 Session 中保存的关键字<br/>  
   \* redirectURL 如果用户未登录，则重定向到指定的页面，URL不包括 ContextPath<br/>  
   \* notCheckURLList 不做检查的URL列表，以分号分开，并且 URL 中不包括 ContextPath<br/>  
   \*/  
  public class CheckLoginFilter implements Filter {  
    protected FilterConfig filterConfig = null;  
       private String redirectURL = null;  
    private List notCheckURLList = new ArrayList();  
    private String sessionKey = null;

    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)   
    throws IOException, ServletException {

      HttpServletRequest request = (HttpServletRequest) servletRequest;  
    HttpServletResponse response = (HttpServletResponse) servletResponse;

           HttpSession session = request.getSession();  
        
    if(sessionKey == null) {  
        filterChain.doFilter(request, response);  
        return;  
      }  
      if((!checkRequestURIIntNotFilterList(request)) &&   
      session.getAttribute(sessionKey) == null) {  
        response.sendRedirect(request.getContextPath() + redirectURL);  
        return;  
      }  
      filterChain.doFilter(servletRequest, servletResponse);  
    }

    public void destroy() {  
      notCheckURLList.clear();  
    }

    private boolean checkRequestURIIntNotFilterList(HttpServletRequest request) {

      String uri = request.getServletPath() + (request.getPathInfo() == null ? "" : request.getPathInfo());  
      return notCheckURLList.contains(uri);  
    }

    public void init(FilterConfig filterConfig) throws ServletException {  
      this.filterConfig = filterConfig;  
      redirectURL = filterConfig.getInitParameter("redirectURL");  
           sessionKey = filterConfig.getInitParameter("checkSessionKey");

      String notCheckURLListStr = filterConfig.getInitParameter("notCheckURLList");

      if(notCheckURLListStr != null) {  
        StringTokenizer st = new StringTokenizer(notCheckURLListStr, ";");  
        notCheckURLList.clear();  
        while(st.hasMoreTokens()) {  
          notCheckURLList.add(st.nextToken());  
        }  
      }  
    }  
  }