**过滤器（web）**

# **1 过滤器的概念**

拦截器：（指的范围更广）拦截 检查 放行

过滤器：（拦截器的一种） 拦截 检查 放行

例如： 地铁安检。

在我们实际项目中，我们对访问某些资源的请求需要进条件的筛选、过滤。当这个请求满足特定的情况，才能让这 个请求访问目标资源，如果不满足条件，就不让访问目标资源。

Filter：是Java中预先定义好了的接口，可以过滤不同的内容，具体怎么过滤，需要使用者定义一个实现类，然后实现接口中的过滤方法，在方法中书写过滤的条件。ﬁlter是对客户端访问资源的过滤，符合条件放行，不符合条件不放行。

Filter 是Servlet规范下的一个组件。

# **2 过滤器的编写**

1 实现Filter接口

2 Filter接口中的方法

|  |
| --- |
| 和Filter生命周期有关的三个方法：  public void init(FilterConfig arg0)  public void doFilter(ServletRequest arg0, ServletResponse arg1,FilterChain arg2)  public void destroy() |

3 在web.xml中配置过滤器

|  |
| --- |
| <filter>  <filter-name>配置名字</filter-name>  <filter-class>Filter类全名</filter-class>  </filter>  <filter-mapping>  <filter-name>配置名字</filter-name>  <url-pattern>过滤路径</url-pattern>  </filter-mapping> |

# 3 过滤器的生命周期

过滤器对象创建时，执行init方法一次，反复执行doFilter方法，Filter对象销毁时，执行destroy方法一次。

过滤器什么时候创建？什么时候销毁？

服务器启动时创建，服务器关闭时销毁。

# 4 Filter中的方法

**1 public void init(FilterConfig arg0)**

**init**方法是，Filter对象创建时执行的方法。

该方法的参数 FilterConfig 作用：

|  |
| --- |
| String getFilterName() 获取Filter 的name  String getInitParameter(String name) 获取初始化参数  Enumeration getInitParameterNames() 获取初始化参数的名字  ServletContext getServletContext() 获取ServletContext |

|  |
| --- |
| public class Test2Filter implements Filter{  private ServletContext sc;  public void init(FilterConfig config) throws ServletException {  String name = config.getFilterName();  System.out.println(name+"<<<<<<<<");  String paramValue = config.getInitParameter("initParamName");  System.out.println(paramValue);  sc = config.getServletContext();  }  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  System.out.println("Test2Filter###doFilter###");  chain.doFilter(request, response);  }  public void destroy() {  } |

**2 public void doFilter(ServletRequest arg0, ServletResponse arg1,FilterChain arg2)**

**doFilter**方法是在访问过滤时，反复执行的方法。（起到拦截和放行的作用）

该方法的参数 FilterChain 用来做过滤的放行工作。（所谓放行是指拦截后，进入要请求的资源） arg2.doFilter(arg0,arg1)--->**责任链模式**

|  |
| --- |
| public class TestFilter implements Filter {  public void init(FilterConfig filterConfig) throws ServletException {  System.out.println("TestFilter...Created...");  }  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  System.out.println("TestFilter的doFilter<<<");  **chain.doFilter(request, response); //放行**  }  public void destroy() {  System.out.println("TestFilter...destroy...");  }  } |

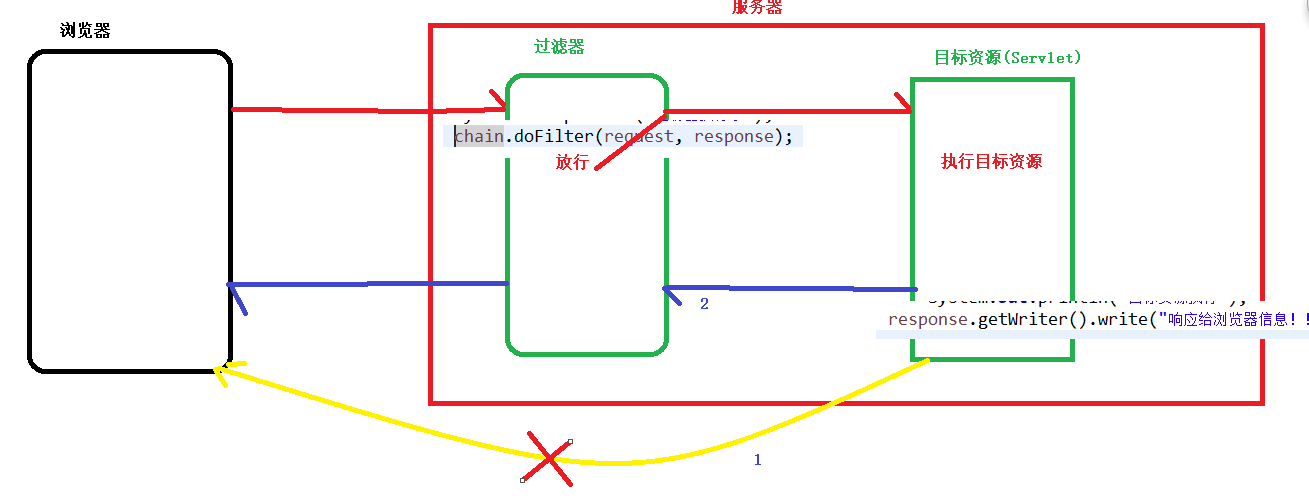
# 5 过滤器的执行流程

## 5.1单过滤器执行流程（拦截器）

平常生活中我们坐地铁或者火车，登车的时候需要先经过安检，安检通过了才能检票，检票通过了才能登车。安检和检票都相当于我们今天学习的过滤器。

创建一个目标资源Servlet：

* 单过滤器的执行流程



**目标资源：访问某个资源被过滤器拦截，这个资源就称之为目标资源**

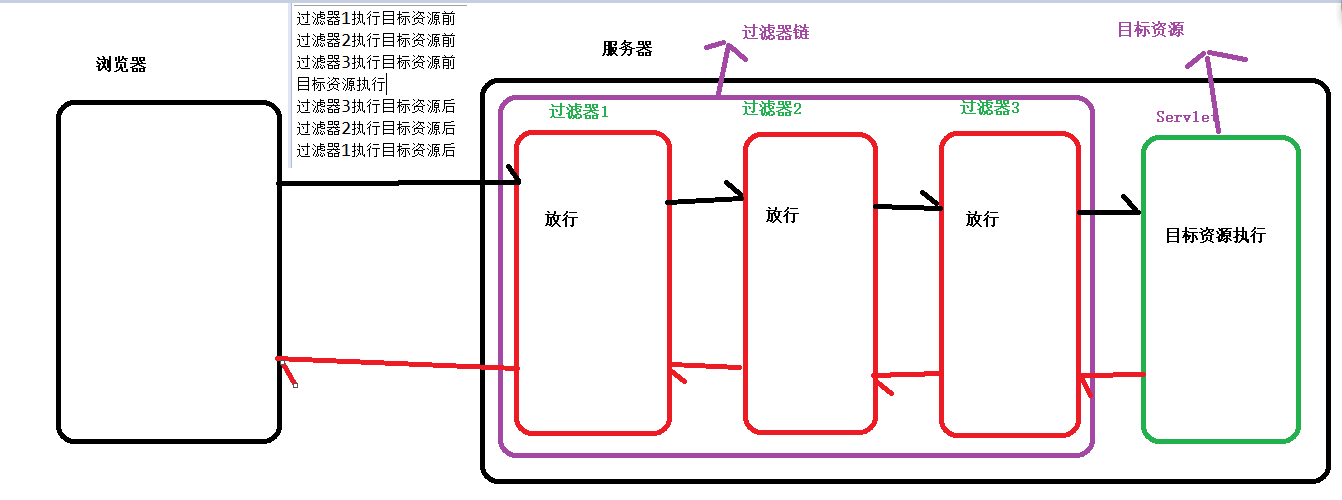
#### 5.2过滤器链的执行流程

过滤器链：

再次回到我们生活中坐地铁或者火车的例子 , 登车的时候需要先经过安检，安检通过了才能检票，检票通过了才能登车 , 其中任意一个没有通过都是登不了车的 , 并且这两个过滤是有顺序。

同样，我们java代码中，有时需要对同一个请求，进行多次不同业务的过滤，所以我们java代码中需要多个过滤器。只有所有的过滤器都对请求进行了放行，请求才能访问到目标资源，只要多个过滤器中只有一个过滤器不放行 请求，那么这个请求都不能够访问到目标资源。多个过滤器组成的一个整体我们称为过滤器链。而且，过滤器链中的过滤器是一个一个的执行的，一个过滤器执行完毕之后，会执行下一个过滤器，后面没有过滤器了，才会访问到目标资源。只要其中一个过滤器没有放行，那么这个过滤器后面的过滤器也都不会执行了。

**先执行的都是过滤器，然后才会涉及到目标资源的执行**



多个过滤器的执行顺序：

跟在web.xml配置的过滤器顺序有关

设计模式： **责任链模式**

23种设计模式： 单例模式、工厂模式（简单工厂、抽象工厂、工厂方法）

比较器(Comparator) ： 策略模式

Arrays.sort(aa,new Comparator())

IO: 转换流： InputStreamReader() OutputStreamWriter()

适配器模式 BufferedReader(new InputStreamReader(System.in))

BufferdReader ： 装饰者模式 ----> 增强某个类的功能

# 6 关于<filter-mapping>配置

<filter-mapping>

<filter-name>TestFilter</filter-name>

<url-pattern></url-pattern>

</filter-mapping>

## 6.1 <url-pattern>配置

1 /具体路径 /TestServlet 只有访问TestServlet才会经过这个过滤器

2 /\* 过滤所有请求

3 \*.do 过滤所有结尾是 .do的请求

4 当请求经过多个过滤器时，过滤器的执行先后顺序和过滤器在web.xml中的配置顺序有关。

|  |
| --- |
|  |

## 6.2 其他配置

<filter-mapping>

<filter-name>

<url-pattern>

<servlet-name> ---> 指定具体Servlet（只过滤这个Servlet）

<dispatcher> ---> 配置过滤器的过滤方式

REQUEST 默认 拦截请求

FORWARD 拦截转发（只拦截转发）

INCLUDE 拦截包含

ERROR 拦截错误页面

</dispatcher>

</filter-mapping>

# 7 案例

## 7.1 编码过滤器（处理POST请求）

|  |
| --- |
| public class EncodingFilter implements Filter {  public void destroy() {  }  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  chain.doFilter(request, response);  }  public void init(FilterConfig config) throws ServletException {  }  } |

|  |
| --- |
| <filter>  <filter-name>EncodingFilter</filter-name>  <filter-class>com.offcn.web.filter.demo1.EncodingFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>EncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

## 7.2 登录过滤器

|  |
| --- |
| public class LoginFilter implements Filter {  public void init(FilterConfig config) throws ServletException {  }  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  HttpServletResponse resp = (HttpServletResponse) response;  HttpSession session = req.getSession();  User user = (User) session.getAttribute("user");  String uri = req.getRequestURI();  if (uri.contains("login.jsp") || uri.endsWith("LoginServlet.do")) {  chain.doFilter(request, response);  } else {  if (user == null) { // 没登录过  resp.sendRedirect("login.jsp?error=pleaselogin");  } else {  chain.doFilter(request, response);  }  }  }  public void destroy() {  }  } |

**很多项目中，有些jsp 和servlet 在不登录的情况下也可以访问。**

**不要用判断路径 放行的方式（因为会有很多路径需要放行）**

**如何完成？**

**要被过滤的jsp 放在文件夹中 例如 用户管理需要登录才能访问**

**创建一个文件夹 user ---| \*\*\*jsp**

**\*\*\*.jsp**

**...**

**对于用户模块的Servlet**

**配置 Servlet的mapping时**

**<url-pattern>/user/...Servlet**

**配置登录过滤器：**

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

# 8 监听器的概念

**监听器**是一个对象，用来监听另一个对象的变化。

例如：汽车上，安装一个报警器。

监听其他对象的对象叫做监听器。

被监听的对象叫做事件源。

报警器就是一个监听器。

汽车就是事件源。

触发 踹一脚汽车。

**web监听器（**3类，8个**）**

Listener 是JavaEE 中Servlet规范下的一个组件

用来监听ServletRequest HttpSession ServletContext这个三个域对象。

# 9 第一类：监听域对象的生命周期

## 9.1 ServletContextListener

监听ServletContext的生命周期

ServletContext 服务器启动时创建，服务器关闭时销毁。

void contextInitialized(ServletContextEvent arg0)

**ServletContext创建时调用的方法。**

void contextDestroyed(ServletContextEvent arg0)

**ServletContext销毁时调用的方法。**

|  |
| --- |
| public class MyServletContextListener implements ServletContextListener {  /\*  \* 监听ServletContext的生命周期（创建和销毁）  \* \*/    public void contextInitialized(ServletContextEvent arg0) {  //ServletContext创建时调用的方法    //arg0.getServletContext(); 获取到事件源    System.out.println("ServletContext 出生了....");  public void contextDestroyed(ServletContextEvent arg0) {  //ServletContext销毁时调用的方法  System.out.println("ServletContext 升天了....");  }  } |

## 9.2 HttpSessionListener

|  |
| --- |
| public class MySessionListener implements HttpSessionListener {  public void sessionCreated(HttpSessionEvent arg0) {    //arg0.getSession(); 获取事件源    System.out.println("Session创建了<<<<<");  }  public void sessionDestroyed(HttpSessionEvent arg0) {    System.out.println("Session销毁<<<<<");  }  } |

## 9.3 ServletRequestListener

|  |
| --- |
| public class MyRequestListener implements ServletRequestListener {  public void requestInitialized(ServletRequestEvent arg0) {  System.out.println("request创建了###");  }    public void requestDestroyed(ServletRequestEvent arg0) {  System.out.println("request销毁了###");  }  } |

**案例1： 任务调度（使用 Timer和TimerTask）**

|  |
| --- |
| public class TimerListener implements ServletContextListener{  public void contextInitialized(ServletContextEvent arg0) {  Timer timer = new Timer();  timer.schedule(new TimerTask() {  @Override  public void run() {  System.out.println("发邮件....");  }  }, 5000, 2000);  }  public void contextDestroyed(ServletContextEvent arg0) {  // TODO Auto-generated method stub  }  } |

**案例2： 统计在线人数（ServletContextListener HttpSessionListener）**

|  |
| --- |
| public class OnlineScListener implements ServletContextListener {  public void contextInitialized(ServletContextEvent arg0) {  ServletContext sc = arg0.getServletContext();  sc.setAttribute("count", 0);  }  public void contextDestroyed(ServletContextEvent arg0) {  }  } |
| public class OnlineSessionListener implements HttpSessionListener {  public void sessionCreated(HttpSessionEvent arg0) {  ServletContext sc = arg0.getSession().getServletContext();  int count = (Integer)sc.getAttribute("count");  //count++;  sc.setAttribute("count", ++count);  }  public void sessionDestroyed(HttpSessionEvent arg0) {  ServletContext sc = arg0.getSession().getServletContext();  int count = (Integer)sc.getAttribute("count");  sc.setAttribute("count", --count);  }  } |

|  |
| --- |
| <body>  <h1>当前在线人数${count }</h1>  </body> |

# 10 第二类：监听值变化

## 10.1 ServletContextAttributeListener

|  |
| --- |
| public class MyScAttrListener implements ServletContextAttributeListener {  public void attributeAdded(ServletContextAttributeEvent arg0) {  System.out.println("向SC中添加了一个值...");  }  public void attributeRemoved(ServletContextAttributeEvent arg0) {  System.out.println("从SC中移除一个值<<<");  }  public void attributeReplaced(ServletContextAttributeEvent arg0) {    System.out.println("替换了SC中的一个值###");  }  } |

## 10.2 HttpSessionAttributeListener

|  |
| --- |
| public class MySessionAttrListener implements HttpSessionAttributeListener {  public void attributeAdded(HttpSessionBindingEvent arg0) {  System.out.println("向Session中添加了一个值");  }  public void attributeRemoved(HttpSessionBindingEvent arg0) {  System.out.println("从Session中移除一个值");  }  public void attributeReplaced(HttpSessionBindingEvent arg0) {  System.out.println("替换Session中的值");  }  } |

## 10.3 ServletRequestAttributeListener

|  |
| --- |
| public class MyRequestAttributeListener implements ServletRequestAttributeListener {  public void attributeAdded(ServletRequestAttributeEvent arg0) {  System.out.println("request添加");  }  public void attributeRemoved(ServletRequestAttributeEvent arg0) {  System.out.println("request移除");  }  public void attributeReplaced(ServletRequestAttributeEvent arg0) {  System.out.println("request替换");  }  } |

**监听器需要web.xml中配置**

|  |
| --- |
| <listener>  <listener-class>com.offcn.web.listener.MyServletContextListener</listener-class>  </listener>  <listener>  <listener-class>com.offcn.web.listener.MySessionListener</listener-class>  </listener>  <listener>  <listener-class>com.offcn.web.listener.MyRequestListener</listener-class>  </listener>      <listener>  <listener-class>com.offcn.web.listener.TimerListener</listener-class>  </listener>      <listener>  <listener-class>com.offcn.web.listener.onlinecount.OnlineScListener</listener-class>  </listener>    <listener>  <listener-class>com.offcn.web.listener.onlinecount.OnlineSessionListener</listener-class>  </listener>      <listener>  <listener-class>com.offcn.web.listener.attr.MyScAttrListener</listener-class>  </listener>      <listener>  <listener-class>com.offcn.web.listener.attr.MySessionAttrListener</listener-class>  </listener>    <listener>  <listener-class>com.offcn.web.listener.attr.MyRequestAttributeListener</listener-class>  </listener> |

# 11 第三类：监听HttpSession中的对象的（JavaBean）

前两类监听器是作用在 ServletContext HttpSession ServletRequest上的，第三类监听器是作用在JavaBean上的。

这类监听器不需要在web.xml中配置。

## 11.1 HttpSessionBindingListener

|  |
| --- |
| public class User implements HttpSessionBindingListener {  private int id;  private String name;  public User(){    }    public User(int id, String name) {  super();  this.id = id;  this.name = name;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public void valueBound(HttpSessionBindingEvent arg0) {  System.out.println("对象绑定到了Session中");  }  public void valueUnbound(HttpSessionBindingEvent arg0) {  System.out.println("对象从Session中移除");  }  } |

|  |
| --- |
| <%  User user = new User(1,"aaa");  session.setAttribute("user", user);  session.removeAttribute("user");    %> |

## 11.2 HttpSessionActivationListener

**Session的钝化和活化**

放在Session中的 没有实现Serilizable接口的对象，在Session钝化时，不会被序列化到磁盘上。

|  |
| --- |
| public class Person implements Serializable,HttpSessionActivationListener {  private int id;  private String name;  public Person(){    }  public Person(int id, String name) {  super();  this.id = id;  this.name = name;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public void sessionDidActivate(HttpSessionEvent arg0) {  //活化  System.out.println("对象被活化......");  }  public void sessionWillPassivate(HttpSessionEvent arg0) {  //钝化  System.out.println("对象被钝化.......");  } |

|  |
| --- |
| <%  Person p = new Person(1,"zhangsan");  session.setAttribute("p", p);  %> |

|  |
| --- |
| 在 META-INF/context.xml  在context.xml中配置  <Context>  <!--  maxIdleSwap:1 session如果1分钟没有使用 就序列化.  directory:序列化后 文件所保存的路径.  -->  <Manager className="org.apache.catalina.session.PersistentManager" maxIdleSwap="1">  <Store className="org.apache.catalina.session.FileStore" directory="offcn"/>  </Manager>  </Context> |
| <body>  <h1>${sessionScope.p.name }</h1>  </body> |