过滤器 & 监听器

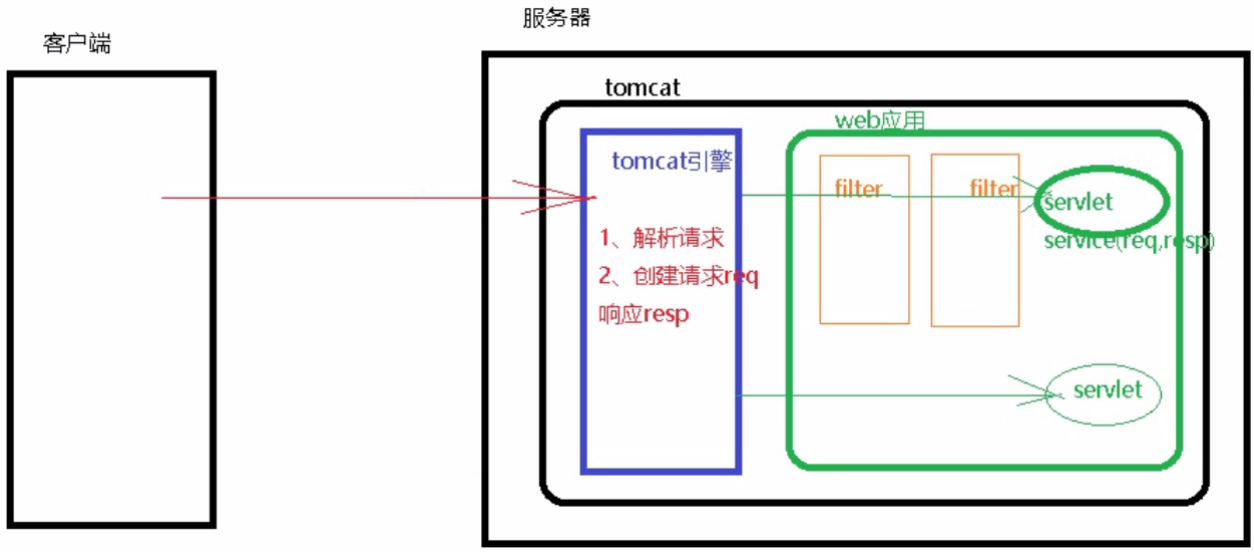
学习目标

1. 能够说出过滤器的作用
2. 能够编写过滤器
3. 能够说出过滤器生命周期相关方法
4. 能够根据过滤路径判断指定的过滤器是否起作用
5. 能够说出什么是过滤器链
6. 能够编写过滤器解决全局乱码

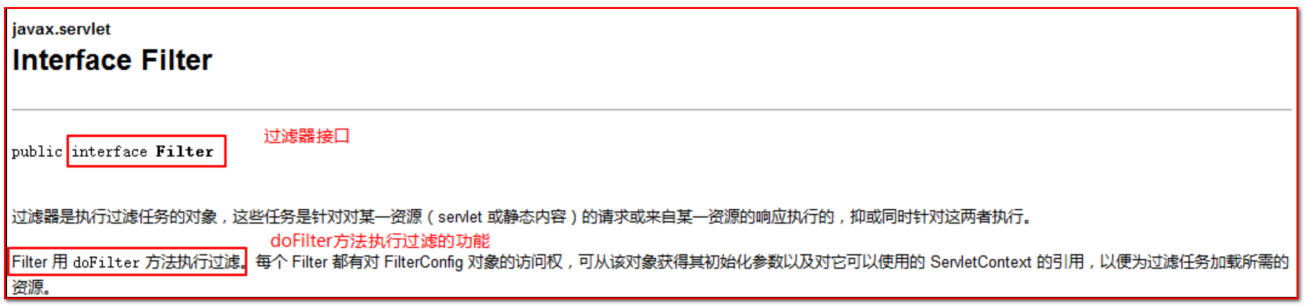
第1章 过滤器Filter

1.1 Filter简介

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



1.2 快速入门



通过API描述可知：

1. 我们创建一个过滤器的话需要实现Filter这个接口

2. doFilter方法执行过滤器的功能

使用步骤

1. 编写一个过滤器的类实现Filter接口
2. 实现接口中尚未实现的方法(着重实现doFilter方法)
3. 在web.xml中（注解）进行配置(主要是配置要对哪些资源进行过滤)
4. 在doFilter方法中书写过滤任务
5. FilterChain.doFilter方法放行

注意事项

过滤器doFilter方法默认拦截请求，如果需要经过过滤器之后，可以继续访问资源，要使用filterChain放行。

代码实现

```
public class MyFilter1 implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        System.out.println("执行MyFilter1的filter......");
        //放行，执行后面的Filter或目标资源
        filterChain.doFilter(servletRequest,servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void destroy() {

    }
}
```

web.xml配置

```
<filter>
    <filter-name>MyFilter1</filter-name>
    <filter-class>com.itheima.filter.MyFilter1</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter1</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

注解方式，不在web.xml中进行配置，使用@webFilter进行配置，如下：

```
@WebFilter(urlPatterns = "/*")
public class MyFilter1 implements Filter {
```

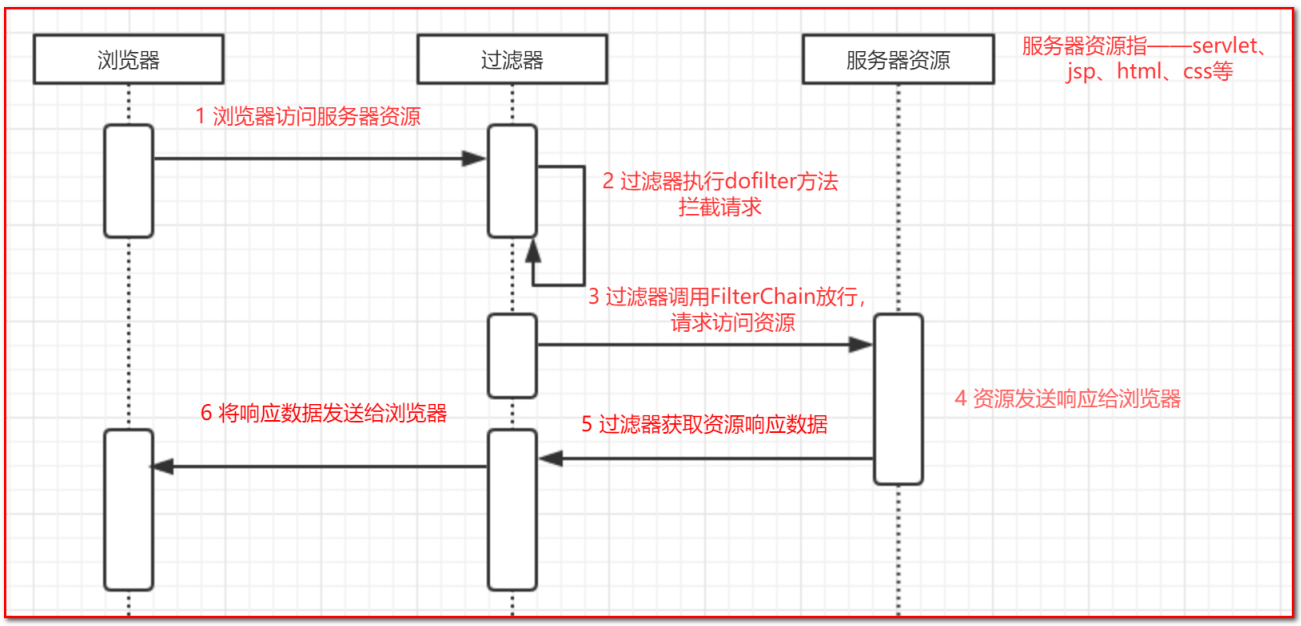
```
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        System.out.println("执行MyFilter1的filter......");
        //放行，执行后面的Filter或目标资源
        filterChain.doFilter(servletRequest,servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void destroy() {

    }
}
```

1.3 过滤器的执行流程

生活案例类比过滤器流程

璐璐从广州开车到长沙，中途途径韶关收费站。如果这座收费站业务通过了，韶关收费站放行，璐璐方可到达长沙。但是如果没有通过韶关收费站的业务，就不能到达长沙目的地。而璐璐中途经过的收费站，就相当于我们java中的过滤器。

图解过滤器的执行流程



1.4 Filter的API详解

1.4.1 filter生命周期及其与生命周期相关的方法

Filter接口有三个方法，并且这个三个都是与Filter的生命相关的方法

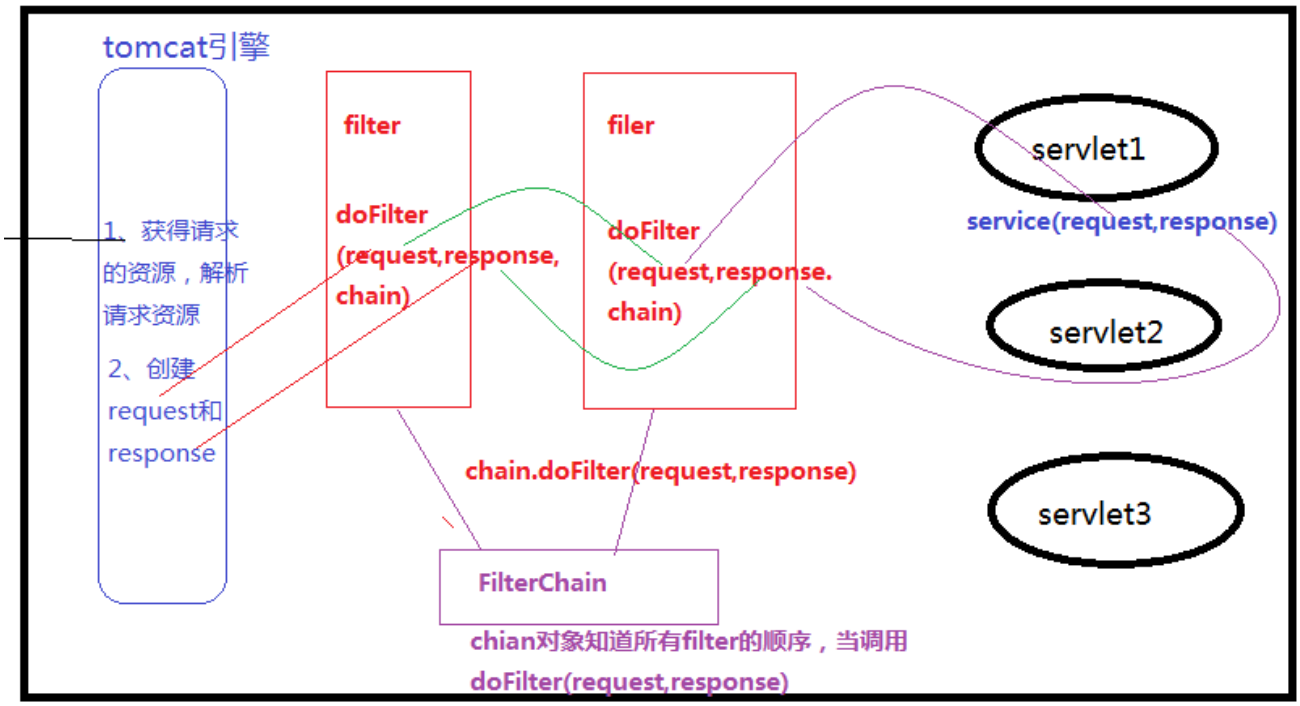
- `init(Filterconfig)`：代表filter对象初始化方法 filter对象创建时执行
- `doFilter(ServletRequest,ServletResponse,FilterCha)`：代表filter执行过滤的核心方法，如果某资源在已经被配置到这个filter进行过滤的话，那么每次访问这个资源都会执行doFilter方法
- `destory()`：代表是filter销毁方法 当filter对象销毁时执行该方法

Filter对象的生命周期:

- Filter何时创建：服务器启动时就创建该filter对象
- Filter何时执行过滤：每当一个请求的路径是满足过滤器的配置路径，那么就会执行一次过滤器的doFilter方法
- Filter何时销毁：服务器关闭时filter销毁

1.4.2 Filter的API详解

- `init(FilterConfig)`:Filter对象创建候，立即执行init方法
  - 参数：FilterConfig是Filter的配置对象，该对象可以获得ServletContext对象
- `destory()`:filter对象销毁前执行destory方法
- `doFilter(ServletRequest,ServletResponse,FilterChain)`:filter执行过滤的核心方法，如果某资源在已经被配置到这个filter进行过滤的话，那么每次访问这个资源都会执行doFilter方法
  - 参数：
    - ServletRequest / ServletResponse：每次在执行doFilter方法时 web容器负责创建一个request和一个response对象作为doFilter的参数传递进来。该request与该response就是在访问目标资源的service方法时的request和response。
    - FilterChain：过滤器链对象，通过该对象的doFilter方法可以放行该请求



代码演示

过滤器doFilter代码如下：

```
@WebFilter(filterName = "FilterDemo",urlPatterns = "/*")
public class FilterDemo1 implements Filter{

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.print("过滤器初始化了");
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        System.out.print("每次请求的路径被过滤器拦截的时候，都会执行过滤器的doFilter方法");
        //放行当前请求
        filterChain.doFilter(servletRequest,servletResponse);
    }

    @Override
    public void destroy() {
        System.out.println("服务器停止的时候销毁过滤器，执行过滤器的desotory方法");
    }
}
```

servlet资源代码如下：

```
@WebServlet(name = "DemoServlet",urlPatterns = "/DemoServlet")
public class DemoServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        doGet(request,response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.setContentType("text/html;charset=utf-8");
        response.getWriter().write("DemoServlet给出响应");
    }
}
```

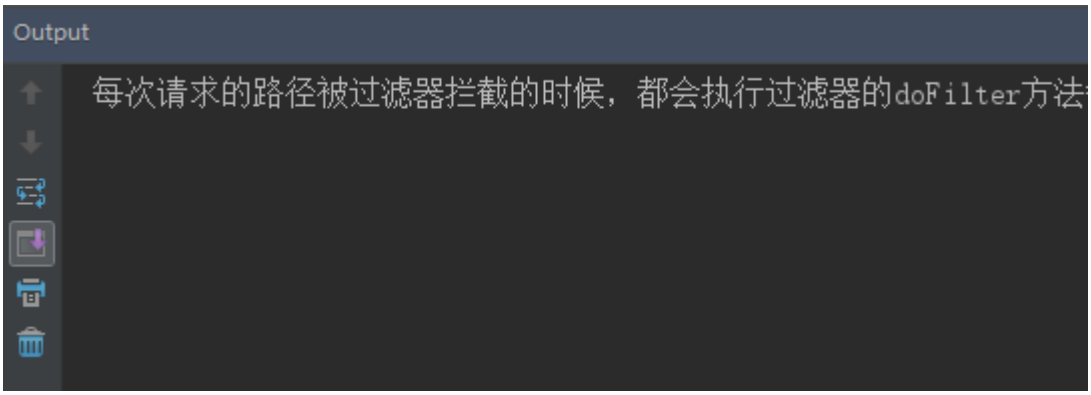
运行结果

在浏览器多次输入`http://localhost:8080/day20/DemoServlet`访问servlet资源

或者多次输入`http://localhost:8080/day20`访问首页index.jsp资源

浏览器能够访问到目标资源。

控制台结果如下：



1.5 Filter配置

1.5.1 映射路径

- xml方式配置：

```
<filter>
    <filter-name>MyFilter1</filter-name>
    <filter-class>com.itheima.filter.MyFilter1</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter1</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 注解配置：

```
@WebFilter(urlPatterns = "/*")
public class MyFilter1 implements Filter{
  //...
}
```

不管是注解配置还是xml配置，配置虚拟路径的语法都是一致的，如下：

1. 完全匹配

   ```
   语法：/sertvle1
   ```

   只有访问的地址是servlet1时，才执行该过滤器

2. 目录匹配

   ```
   语法：/aaa/bbb/* ，经常使用
   ```

   当访问的目标资源 的地址是/aaa/bbb/任何资源 都执行该过滤器

3. 扩展名匹配

```
语法：*.abc  *.jsp
```

当访问的目标资源的扩展名是 abc、jsp时 才执行该过滤器

> 代码演示 映射路径

1.5.2 拦截方式

有了上面学习的映射路径，我们可以控制过滤器过滤指定的内容，但是我们在访问资源的时候，并不是每次都是之间访问，有时是以转发的方式访问的，这就需要我们要让过滤器可以区分不同的访问资源的方式，有不同的拦截方式。

- dispatcher：访问的方式
  - REQUEST：默认值，代表直接访问某个资源时执行filter
  - FORWARD：转发时才执行filter

```
格式：
@WebFilter(urlPatterns = "/*",dispatcherTypes = {DispatcherType.REQUEST,DispatcherType.FORWARD})
public class MyFilter1 implements Filter {
    //...
}
```

> 注意：只有访问的路径 与访问的方式 都匹配时 那么该filter才会执行

代码演示：

1. 创建ForwardServlet，转发到index.jsp去，代码如下

```
@WebFilter(filterName = "MethodFilter",urlPatterns = "/index.jsp",dispatcherTypes =
DispatcherType.FORWARD)
public class ForwardServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        doGet(request,response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        System.out.println("ForwardServlet执行了，下面开始转发请求到index.jsp去");
        request.getRequestDispatcher("/index.jsp").forward(request,response);
    }
}
```

2. 在MethodFilter的dispatcherTypes中以数组的形式添加两个拦截方式

```
@WebFilter(filterName = "MethodFilter",dispatcherTypes =
{DispatcherType.FORWARD,DispatcherType.REQUEST},urlPatterns = "/*")
public class MethodFilter implements Filter {
    public void destroy() {

    }
```

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
ServletException, IOException {
        System.out.println("+++++++++MethodFilter过滤器执行了++++++++++");
        chain.doFilter(req, resp);
    }

    public void init(FilterConfig config) throws ServletException {

    }

}
```

3. 浏览器再次访问请求：`http://localhost:8080/day20/ForwardServlet`,在转发前后都执行了过滤。

```
+++++++++MethodFilter过滤器执行了++++++++++
ForwardServlet执行了，下面开始转发请求到index.jsp去
+++++++++MethodFilter过滤器执行了++++++++++
```

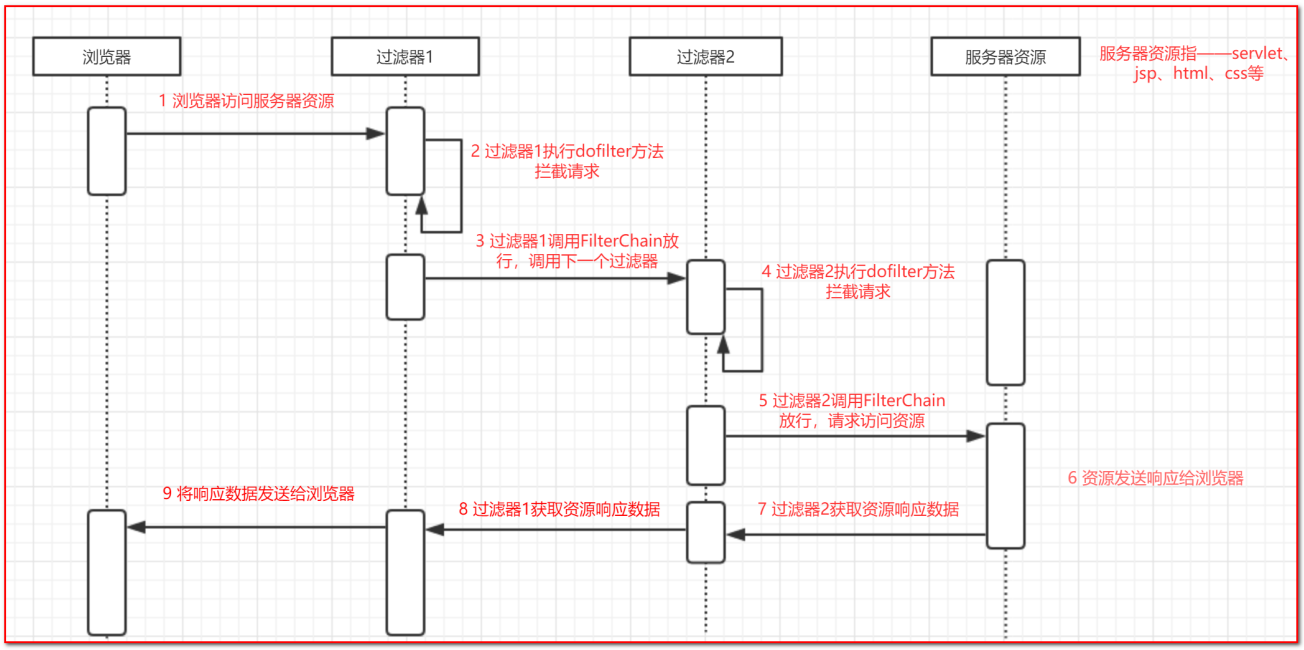
1.6 过滤器链

什么是过滤器链

璐璐从广州开车到长沙，途径韶关收费站，株洲收费站。韶关收费站收的是路过韶关的费用，株洲收费站收的是路过株洲的费用。如果这二座收费站业务通过了，方可到达长沙，只要其中一个收费站的业务没有通过，那么就不能到达长沙。而且，只有通过了韶关收费站，才能到达株洲收费站。

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。

过滤器链的执行流程



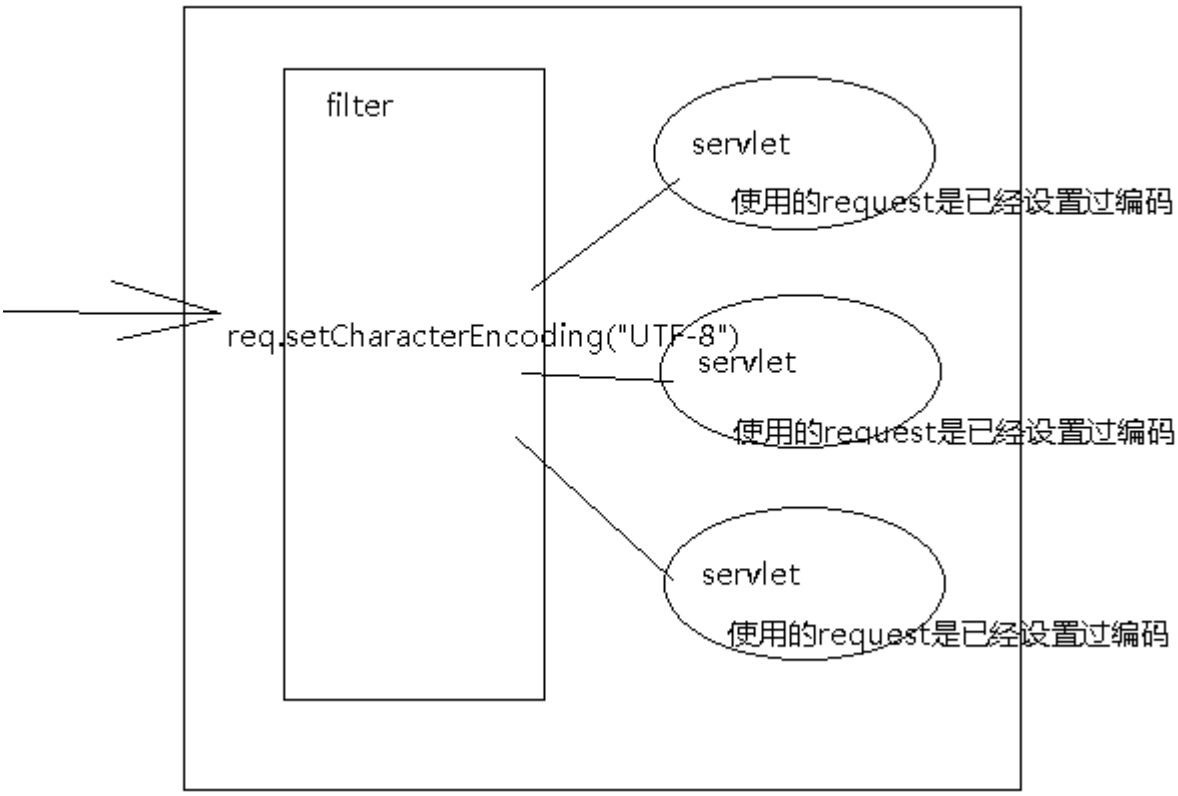
提示：

- web.xml配置的Filter的执行顺序取决于filter-mapping的顺序
- 注解配置的filter的执行顺序取决于filter的类名称的字母排序

1.7 过滤器的应用场景

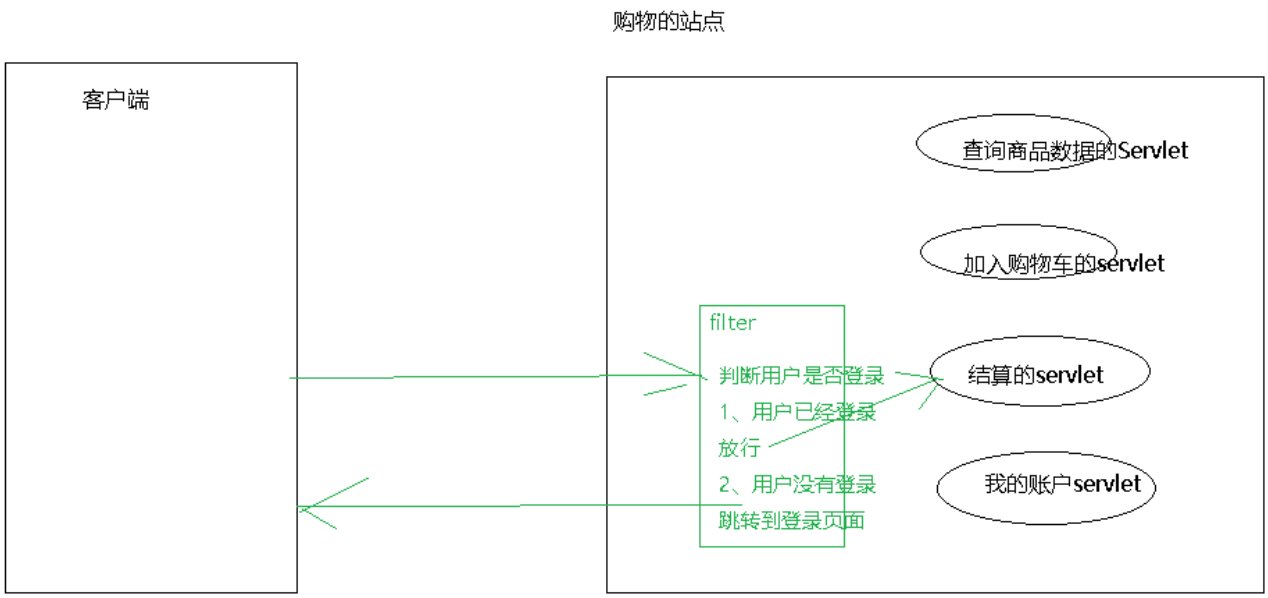
处理全站中文乱码

浏览器发出的任何请求，通过过滤器统一处理中文乱码。



登录权限检查

当客户端浏览器发出一个请求，这个请求在访问到正真的目标资源之前，我们需要进行登录权限的检查。如果已经登录，我们就让这个请求通过，访问资源；如果没有登录，我们不能让请求访问目标资源。这样的操作需要在访问具体资源进行条件的过滤，我们可以使用过滤器来完成。



1.8 案例：解决全站乱码

案例需求

浏览器发出的任何请求，通过过滤器统一处理中文乱码。

案例效果

分别以get方式和post方式提交中文，servlet中不做中文乱码处理，直接获取参数，得到的参数不存在中文乱码问题。

案例分析

1.创建一个过滤器。

2.因为对所有的请求进行乱码的过滤，所以过滤器的过滤路径配置为/*

3.针对post请求处理乱码

实现步骤

1. 创建一个form.jsp表单，用于测试过滤器解决乱码的功能：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <form action="${pageContext.request.contextPath }/encoding" method="post">
        <input type="text" name="name"/>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

2. 创建一个用于接收表单请求的EncodingServlet：

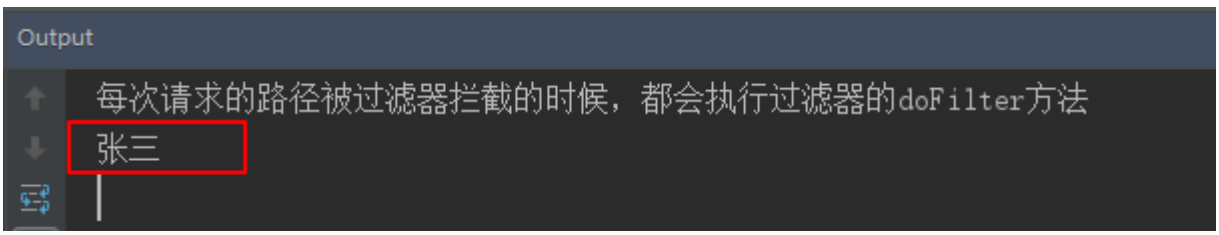
```
@WebServlet(name = "EncodingServlet" , urlPatterns = "/encoding")
public class EncodingServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        doGet(request,response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        //获取表单提交的参数
        String name = request.getParameter("name");
        System.out.println(name);
    }
}
```

3. 创建EncodingFilter用于乱码处理的过滤器

```
@WebFilter(filterName = "EncodingFilter",urlPatterns = "/*")
```

```
public class EncodingFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
ServletException, IOException {
       //首先
        HttpServletRequest request = (HttpServletRequest)req;
        resp.setContentType("text/html;charset=utf-8");
        //获取请求的方式
        String method = request.getMethod();
       if("post".equalsIgnoreCase(method)){
            //如果是post请求
            request.setCharacterEncoding("utf-8");
            chain.doFilter(request, resp);
            //结束当前方法
            return;
        }
        chain.doFilter(req, resp);
    }

    public void init(FilterConfig config) throws ServletException {

    }
}
```

4. 浏览器访问form.jsp页面，表单提交方式为post，表单输入中文
5. 点击提交按钮，查看控制台，post乱码问题得到解决



6. 将表单的提交方式换成get，提交表单，查看控制台没有乱码问题。

第2章 监听器Listener

2.1 什么是监听器

监听器就是监听某个对象的的状态变化的组件 监听器的相关概念： **事件源**：被监听的对象 ----- 主要三个域对象 request session servletContext **监听器**：监听事件源对象 事件源对象的状态的变化都会触发监听器 ---- 6+2 **响应行为**：监听器监听到事件源的状态变化时 所涉及的功能代码 ---- 程序员编写代码

2.2 监听器有哪些

第一维度：按照被监听的对象划分：ServletRequest域 HttpSession域 ServletContext域 第二维度：安装监听的内容分：监听域对象的创建与销毁的监听域对象的属性变化的

	ServletContext域	HttpSession域	ServletRequest域
域对象的创建与销毁	ServletContextListener	HttpSessionListener	ServlerRequestListener
域对象内的属性的变化	ServletContextAttributeListener	HttpSessionAttributeListener	ServlerRequestAttributeListener

2.3 ServletContextListener

ServletContextListener：用于监听ServletContext域的创建与销毁的监听器

- 应用场景：服务器启动时，加载配置信息
- ServletContext域的生命周期
  - 何时创建：服务器启动创建
  - 何时销毁：服务器关闭销毁
  - 作用范围：整个web应用
- 监听器的编写步骤：
  1. 编写一个监听器类去实现监听器接口
  2. 覆盖监听器的方法
  3. 需要在web.xml（注解）中进行配置

代码实现：

```
public class MyServletContextListener implements ServletContextListener{

    @Override
    public void contextInitialized(ServletContextEvent sce) {
             System.out.println("ServletContext对象创建");
             ServletContext context = sce.getServletContext();
             System.out.println(context);
             Object source = sce.getSource();
             System.out.println(source);
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("ServletContext对象销毁");
    }

}
```

web.xml配置

```
<listener>
  <listener-class>com.itheima.contextfilter.MyServletContextListener
  </listener-class>
</listener>
```

注解方式：

```
@WebListener
public class MyServletContextListener implements ServletContextListener{
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("ServletContext对象创建的了.....");
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("ServletContext对象销毁的了.....");
    }
}
```