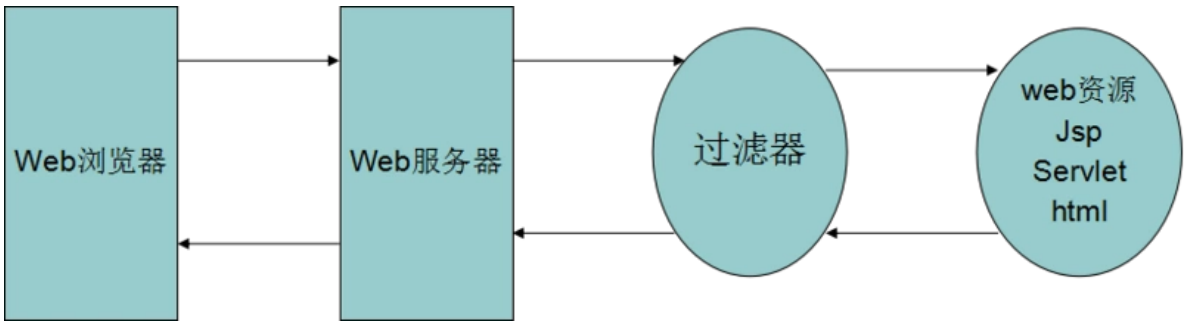
1 什么是过滤器

过滤器是Servlet的高级特性之一，也别把它想得那么高深，只不过是实现Filter接口的Java类罢了！

首先，我们来看看过滤器究竟Web容器的哪处:



从上面的图我们可以发现，当浏览器发送请求给服务器的时候，先执行过滤器，然后才访问Web的资源。服务器响应Response，从Web资源抵达浏览器之前，也会途径过滤器。

我们很容易发现，过滤器可以比喻成一张滤网。我们想想现实中的滤网可以做什么：在泡茶的时候，过滤掉茶叶。那滤网是怎么过滤茶叶的呢？规定大小的网孔，只要网孔比茶叶小，就可以实现过滤了！

引申在Web容器中，过滤器可以做：过滤一些敏感的字符串【规定不能出现敏感字符串】、避免中文乱码【规定Web资源都使用UTF-8编码】、权限验证【规定只有带Session或Cookie的浏览器，才能访问web资源】等等等，过滤器的作用非常大，只要发挥想象就可以有意想不到的效果。

也就是说：当需要限制用户访问某些资源时、在处理请求时提前处理某些资源、服务器响应的内容对其进行处理再返回、我们就是用过滤器来完成的！

2 为什么需要用到过滤器

直接举例子来说明：中文乱码问题。

如果没有用到过滤器：浏览器通过http请求发送数据给Servlet，如果存在中文，就必须指定编码，否则就会乱码！也就是说：如果我每次接受客户端带过来的中文数据，在Serlvet中都要设定编码。这样代码的重复率太高了！

有过滤器的情况就不一样了：只要我在过滤器中指定了编码，可以使全站的Web资源都是使用该编码，并且重用性是非常理想的！

3 过滤器 API

只要Java类实现了Filter接口就可以称为过滤器！Filter接口的方法也十分简单：

```
public interface Filter{
    void init(FilterConfig var1) throw ServletException;
    void doFilter(ServletRequest var1,ServletResponse var2,FilterChain var3)
throw IOException,ServletException;
    void destroy();
}
```

其中init()和destory()方法就不用多说了，~~他俩跟Servlet是一样的。只有在Web服务器加载和销毁的时候被执行，只会被执行一次~~！

值得注意的是doFilter()方法，它有三个参数（ServletRequest,ServletResponse,FilterChain），从前两个参数我们可以发现：过滤器可以完成任何协议的过滤操作！

那FilterChain是什么东西呢？我们看看：

```
public interface FilterChain{
    void doFilter(ServletRequest var1,ServletResponse var2) throw
IOException,ServletException;
}
```

FilterChain是一个接口，里面又定义了doFilter()方法。这究竟是怎么回事啊？

我们可以这样理解：过滤器不单单只有一个，那么我们怎么管理这些过滤器呢？在Java中就使用了链式结构。把所有的过滤器都放在FilterChain里边，如果符合条件，就执行下一个过滤器（如果没有过滤器了，就执行目标资源）。

我们可以想象生活的例子：现在我想在茶杯上能过滤出石头和茶叶出来。石头在一层，茶叶在一层。所以茶杯的过滤装置应该有两层滤网。这个过滤装置就是FilterChain，过滤石头的滤网和过滤茶叶的滤网就是Filter。在石头滤网中，茶叶是属于下一层的，就把茶叶放行，让茶叶的滤网过滤茶叶。过滤完茶叶了，剩下的就是茶（茶就可以比喻成我们的目标资源）

4 快速入门

4.1 简单的过滤器

实现Filter接口的Java类就被称作为过滤器

```
public class FilterDemo1 implements Filter {
    public void destroy() {
    }
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
        //执行这一句，说明放行（让下一个过滤器执行，如果没有过滤器了，就执行执行目标资源）
        chain.doFilter(req, resp);
    }
    public void init(FilterConfig config) throws ServletException {
    }
}
```

4.2 filter部署

过滤器和Servlet是一样的，需要部署到Web服务器上的。

第一种方式：在web.xml文件中配置

filter用于注册过滤器。

```
<filter>
    <filter-name>FilterDemo1</filter-name>
    <filter-class>FilterDemo1</filter-class>
    <init-param>
        <param-name>word_file</param-name>
        <param-value>/WEB-INF/word.txt</param-value>
    </init-param>
</filter>
```

- `<filter-name>` 用于为过滤器指定一个名字，该元素的内容不能为空。
- `<filter-class>` 元素用于指定过滤器的完整的限定类名。
- `<init-param>` 元素用于为过滤器指定初始化参数，它的子元素<param-name>指定参数的名字，`<param-value>` 指定参数的值。在过滤器中，可以使用FilterConfig接口对象来访问初始化参数。

filter-mapping元素用于设置一个Filter 所负责拦截的资源。

一个Filter拦截的资源可通过两种方式来指定：Servlet 名称和资源访问的请求路径

```
<filter-mapping>
    <filter-name>FilterDemo1</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- `<filter-name>` 子元素用于设置filter的注册名称。该值必须是在<filter>元素中声明过的过滤器的名字
- `<url-pattern>` 设置 filter 所拦截的请求路径(过滤器关联的URL样式)
- `<servlet-name>` 指定过滤器所拦截的Servlet名称。
- `<dispatcher>` 指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST。用户可以设置多个 `<dispatcher>` 子元素用来指定 Filter 对资源的多种调用方式进行拦截。

`<dispatcher>` 子元素可以设置的值及其意义：

- 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。

第二种方式：通过注解配置

```
@WebFilter(filterName = "FilterDemo1",urlPatterns = "/*")
```

上面的配置是“/*”，所有的Web资源都需要途径过滤器。

如果想要部分的Web资源进行过滤器过滤则需要指定Web资源的名称即可！

5 过滤器的执行顺序

上面已经说过了，过滤器的doFilter()方法是极其重要的，FilterChain接口是代表着所有的Filter，FilterChain中的doFilter()方法决定着是否放行下一个过滤器执行（如果没有过滤器了，就执行目标资源）。

注意：过滤器之间的执行顺序看在web.xml文件中mapping的先后顺序的，如果放在前面就先执行，放在后面就后执行！如果是通过注解的方式配置，就比较urlPatterns的字符串优先级

6 Filter简单应用

filter的三种典型应用：

1. 可以在filter中根据条件决定是否调用chain.doFilter(request, response)方法，即是否让目标资源执行
2. 在让目标资源执行之前，可以对requestresponse作预处理，再让目标资源执行
3. 在目标资源执行之后，可以捕获目标资源的执行结果，从而实现一些特殊的功能

7 编码过滤器

目的：解决全站的乱码问题。

7.1 开发过滤器

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
    //将request和response强转成http协议的
    HttpServletRequest httpServletRequest = (HttpServletRequest) req;
    HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

    httpServletRequest.setCharacterEncoding("UTF-8");
    httpServletResponse.setCharacterEncoding("UTF-8");
    httpServletResponse.setContentType("text/html;charset=UTF-8");

    chain.doFilter(httpServletRequest, httpServletResponse);
}
```

第一次测试

Servlet1中向浏览器回应中文数据，没有出现乱码。

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.getWriter().write("看完博客点赞！");
}
```

分析

上面的过滤器是不完善的，因为浏览器用get方式提交给服务器的中文数据，单单靠上面的过滤器是无法完成的！

我们之前解决get方式的乱码问题是这样的：使用request获取传递过来的数据，经过ISO 8859-1反编码获取得到不是乱码的数据（传到Servlet上的数据已经被ISO 8859-1编码过了，反编码就可以获取原来的数据），再用UTF-8编码，得到中文数据！

在Servlet获取浏览器以GET方式提交过来的中文是乱码的根本原因是：getParameter()方法是以ISO 8859-1的编码来获取浏览器传递过来的数据的，得到的是乱码。

既然知道了根本原因，那也好办了：过滤器传递的request对象，使用getParameter()方法的时候，获取得到的是正常的中文数据。

也就是说，sun公司为我们提供的request对象是不够用的，因为sun公司提供的request对象使用getParameter()获取get方式提交过来的数据是乱码，于是我们要增强request对象（使得getParameter()获取得到的是中文）！

7.2 增强request对象

增强request对象，我们要使用包装设计模式!

包装设计模式的五个步骤：

1. 实现与被增强对象相同的接口
2. 定义一个变量记住被增强对象
3. 定义一个构造器，接收被增强对象
4. 覆盖需要增强的方法
5. 对于不想增强的方法，直接调用被增强对象（目标对象）的方法

sun公司也知道我们可能对request对象的方法不满意，于是提供了HttpServletRequestWrapper类给我们实现（如果实现HttpServletRequest接口的话，要实现太多的方法了！）

```
class MyRequest extends HttpServletRequestWrapper {
    private HttpServletRequest request;
    public MyRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }
    @Override
    public String getParameter(String name) {
        String value = this.request.getParameter(name);
        if (value == null) {
            return null;
        }
        //如果不是get方法的，直接返回就行了
        if (!this.request.getMethod().equalsIgnoreCase("get")) {
            return value;
        }
        try {
            //进来了就说明是get方法，把乱码的数据
            value = new String(value.getBytes("ISO8859-1"),
this.request.getCharacterEncoding());
            return value ;
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
            throw new RuntimeException("不支持该编码");
        }
    }
}
```

将被增强的request对象传递给目标资源，那么目标资源使用request调用getParameter()方法的时候，获取得到的就是中文数据，而不是乱码了！

```
//将request和response强转成http协议的
HttpServletRequest httpServletRequest = (HttpServletRequest) req;
HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

httpServletRequest.setCharacterEncoding("UTF-8");
httpServletResponse.setCharacterEncoding("UTF-8");
httpServletResponse.setContentType("text/html;charset=UTF-8");

MyRequest myRequest = new MyRequest(httpServletRequest);

//传递给目标资源的request是被增强后的。
chain.doFilter(myRequest, httpServletResponse);
```

第二次测试

使用get方式传递中文数据给服务器，能够正常显示。

8 敏感词的过滤器

如果用户输入了敏感词，我们要将这些不文明用于屏蔽掉，替换成符号！

要实现这样的功能也很简单，用户输入的敏感词肯定是在getParameter()获取的，我们在getParameter()得到这些数据的时候，判断有没有敏感词汇，如果有就替换掉就好了！简单来说：也是要增强request对象.

8.1 增强request对象

```
class MyDirtyRequest extends HttpServletRequestWrapper {
    HttpServletRequest request;
    //定义一堆敏感词汇
    private List<String> list = Arrays.asList("傻b", "尼玛", "操蛋");
    public MyDirtyRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }
    @Override
    public String getParameter(String name) {
        String value = this.request.getParameter(name);
        if (value == null) {
            return null;
        }
        //遍历list集合，看看获取得到的数据有没有敏感词汇
        for (String s : list) {
            if (s.equals(value)) {
                value = "*****";
            }
        }
        return value ;
    }
}
```

8.2 开发过滤器

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
    //将request和response强转成http协议的
    HttpServletRequest httpServletRequest = (HttpServletRequest) req;
    HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

    MyDirtyRequest dirtyRequest = new MyDirtyRequest(httpServletRequest);

    //传送给目标资源的是被增强后的request对象
    chain.doFilter(dirtyRequest, httpServletResponse);
}
```

9 压缩资源过滤器

按照过滤器的执行顺序：执行完目标资源，过滤器后面的代码还会执行。所以，我们在过滤器中可以获取执行完目标资源后的response对象！

我们知道sun公司提供的response对象调用write()方法，是直接把数据返回给浏览器的。**我们要想实现压缩的功能，**write()方法就不能直接把数据写到浏览器上！

这和上面是类似的，过滤器传递给目标资源的response对象就需要被我们增强，使得目标资源调用writer()方法的时候不把数据直接写到浏览器上！

9.1 增强response对象

response对象可能会使用PrintWriter或者ServletOutputStream对象来调用writer()方法的，所以我们增强response对象的时候，需要把getOutputSteam和getWriter()重写

```
class MyResponse extends HttpServletResponseWrapper{
    HttpServletResponse response;
    public MyResponse(HttpServletResponse response) {
        super(response);
        this.response = response;
    }
    @Override
    public ServletOutputStream getOutputStream() throws IOException {
        return super.getOutputStream();
    }
    @Override
    public PrintWriter getWriter() throws IOException {
        return super.getWriter();
    }
}
```

接下来，ServletOutputSteam要调用writer()方法，使得它不会把数据写到浏览器上。这又要我们增强一遍了！

```
/*增强ServletOutputSteam，让writer方法不把数据直接返回给浏览器*/
class MyServletOutputStream extends ServletOutputStream{
    private ByteArrayOutputStream byteArrayOutputStream;
    public MyServletOutputStream(ByteArrayOutputStream byteArrayOutputStream) {
        this.byteArrayOutputStream = byteArrayOutputStream;
    }
    //当调用write()方法的时候，其实是把数据写byteArrayOutputSteam上
    @Override
    public void write(int b) throws IOException {
        this.byteArrayOutputStream.write(b);
    }
}
```

PrintWriter对象就好办了，它本来就是一个包装类，看它的构造方法，我们直接可以把ByteArrayOutputSteam传递给PrintWriter上。

```
@Override
public PrintWriter getWriter() throws IOException {
    printWriter = new PrintWriter(new OutputStreamWriter(byteArrayOutputStream,
this.response.getCharacterEncoding()));
    return printWriter;
}
```

我们把数据都写在了ByteArrayOutputSteam上了，应该提供方法给外界过去缓存中的数据！

```
public byte[] getBuffer() {
    try {
        //防止数据在缓存中，要刷新一下！
        if (printWriter != null) {
            printWriter.close();
        }
        if (byteArrayOutputStream != null) {
            byteArrayOutputStream.flush();
            return byteArrayOutputStream.toByteArray();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

9.2 开发过滤器

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) resp;
    MyResponse myResponse = new MyResponse(response);
    //把被增强的response对象传递进去，目标资源调用write()方法的时候就不会直接把数据写在浏览
器上了
    chain.doFilter(request, myResponse);
    //得到目标资源想要返回给浏览器的数据
    byte[] bytes = myResponse.getBuffer();
    //输出原来的大小
```

```
    System.out.println("压缩前："+bytes.length);
    //使用GZIP来压缩资源，再返回给浏览器
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    GZIPOutputStream gzipOutputStream = new
GZIPOutputStream(byteArrayOutputStream);
    gzipOutputStream.write(bytes);
    gzipOutputStream.flush();
    //得到压缩后的数据
    byte[] gzip = byteArrayOutputStream.toByteArray();
    System.out.println("压缩后：" + gzip.length);
    //还要设置头，告诉浏览器，这是压缩数据！
    response.setHeader("content-encoding", "gzip");
    response.setContentLength(gzip.length);
    response.getOutputStream().write(gzip);
}
```

10 HTML转义过滤器

只要把getParameter()获取得到的数据转义一遍，就可以完成功能了。

10.1 增强request

```
class MyHtmlRequest extends HttpServletRequestWrapper{
    private HttpServletRequest request;
    public MyHtmlRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }
    @Override
    public String getParameter(String name) {
        String value = this.request.getParameter(name);
        return this.Filter(value);
    }
    public String Filter(String message) {
        if (message == null)
            return (null);
        char content[] = new char[message.length()];
        message.getChars(0, message.length(), content, 0);
        StringBuffer result = new StringBuffer(content.length + 50);
        for (int i = 0; i < content.length; i++) {
            switch (content[i]) {
                case '<':
                    result.append("&lt;");
                    break;
                case '>':
                    result.append("&gt;");
                    break;
                case '&':
                    result.append("&amp;");
                    break;
                case '"':
                    result.append("&quot;");
                    break;
                default:
                    result.append(content[i]);
```

```
            }
        }
        return (result.toString());
    }
}
```

10.2 开发过滤器

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) resp;
    MyHtmlRequest myHtmlRequest = new MyHtmlRequest(request);

    //传入的是被增强的request!
    chain.doFilter(myHtmlRequest, response);
}
```

10.3 测试

jsp代码：

```
<form action="${pageContext.request.contextPath}/Servlet1" method="post">
    <input type="hidden" name="username" value="<h1>你好i好<h1>">
    <input type="submit" value="提交">
</form>
```

Servlet代码：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String value = request.getParameter("username");
    response.getWriter().write(value);
}
```

11 缓存数据到内存

在前面我们已经做过了，让浏览器不缓存数据【验证码的图片是不应该缓存的】。

现在我们要做的是：缓存数据到内存中【如果某个资源重复使用，不轻易变化，应该缓存到内存中】

这个和压缩数据的Filter非常类似的，因为让数据不直接输出给浏览器，把数据用一个容器（ByteArrayOutputSteam）存起来。如果已经有缓存了，就取缓存的。没有缓存就执行目标资源！

11.1 增强response对象

```
class MyResponse extends HttpServletResponseWrapper {
    private ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
    private PrintWriter printWriter ;
    private HttpServletResponse response;
    public MyResponse(HttpServletResponse response) {
```

```
        super(response);
        this.response = response;
    }
    @Override
    public ServletOutputStream getOutputStream() throws IOException {
        //这个的ServletOutputSteam对象调用write()方法的时候，把数据是写在
byteArrayOutputSteam上
        return new MyServletOutputStream(byteArrayOutputStream);
    }
    @Override
    public PrintWriter getWriter() throws IOException {
        printWriter = new PrintWriter(new
OutputStreamWriter(byteArrayOutputStream,
this.response.getCharacterEncoding()));
        return printWriter;
    }
    public byte[] getBuffer() {
        try {
            //防止数据在缓存中，要刷新一下!
            if (printWriter != null) {
                printWriter.close();
            }
            if (byteArrayOutputStream != null) {
                byteArrayOutputStream.flush();
                return byteArrayOutputStream.toByteArray();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
//增强ServletOutputSteam，让writer方法不把数据直接返回给浏览器
class MyServletOutputStream extends ServletOutputStream {
    private ByteArrayOutputStream byteArrayOutputStream;
    public MyServletOutputStream(ByteArrayOutputStream byteArrayOutputStream) {
        this.byteArrayOutputStream = byteArrayOutputStream;
    }
    //当调用write()方法的时候，其实是把数据写byteArrayOutputSteam上
    @Override
    public void write(int b) throws IOException {
        this.byteArrayOutputStream.write(b);
    }
}
```

11.2 开发过滤器

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
    //定义一个Map集合，key为页面的地址，value为内存的缓存
    Map<String, byte[]> map = new HashMap<>();
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) resp;
    //得到客户端想要请求的资源
    String uri = request.getRequestURI();
    byte[] bytes = map.get(uri);
```

```
    //如果有缓存，直接返回给浏览器就行了，就不用执行目标资源了
    if (bytes != null) {
        response.getOutputStream().write(bytes);
        return ;
    }
    //如果没有缓存，就让目标执行
    MyResponse myResponse = new MyResponse(response);
    chain.doFilter(request, myResponse);
    //得到目标资源想要发送给浏览器的数据
    byte[] b = myResponse.getBuffer();
    //把数据存到集合中
    map.put(uri, b);
    //把数据返回给浏览器
    response.getOutputStream().write(b);
}
```