第4章　Filter（过滤器）

学习目标

◆ **了解什么是 Filter**

◆ **能够用 Filter 实现用户自动登录的案例**

◆ **了解什么是装饰设计模式，学会用 Filter 实现统一全站编码和页面静态化技术**

在 Web 开发过程中，为了实现某些特殊的功能，经常需要对请求和响应消息进行处理。例如记录用户访问信息，统计页面访问次数，验证用户身份等。Filter 作为 Servlet2.3 中新增的技术，可以实现用户在访问某个目标资源之前，对访问的请求和响应进行相关处理。接下来，本章将针对 Filter 进行详细地讲解。

4.1　Filter 入门

4.1.1　什么是 Filter

Filter 被称作过滤器或者拦截器，其基本功能就是对 Servlet 容器调用 Servlet 的过程进行拦截，从而在 Servlet 进行响应处理前后实现一些特殊功能。这就好比现实中的污水净化设备，它可以看做一个过滤器，专门用于过滤污水杂质。图 4-1 描述了 Filter 在 Web 应用中的拦截过程，具体如下：

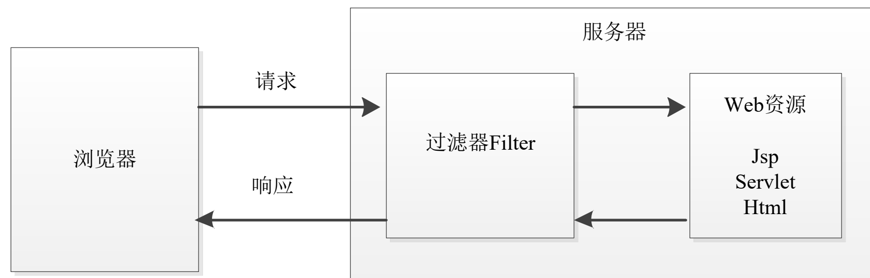

图4-1　Filter 拦截过程

在图 4-1 中，当浏览器访问服务器中的目标资源时，会被 Filter 拦截，在 Filter 中进行预处理操作，然后再将请求转发给目标资源。当服务器接收到这个请求后会对其进行响应，在服务器处理响应的过程中，也需要先将响应结果发送给拦截器，在拦截器中对响应结果进行处理后，才会发送给客户端。

其实，Filter 过滤器就是一个实现了 javax.servlet.Filter 接口的类，在 javax.servlet.Filter 接口中定义了三个方法，具体如表 4-1 所示。

表4-1　Filter 接口中的方法

方法声明	功能描述
init(FilterConfig filterConfig)	init()方法用来初始化过滤器，开发人员可以在 init()方法中完成与构造方法类似的初始化功能，如果初始化代码中要使用到 FilterConfig 对象，那么，这些初始化代码就只能在 Filter 的 init()方法中编写，而不能

	在构造方法中编写
doFilter(ServletRequest request,ServletResponse response,FilterChain chain)	doFilter()方法有多个参数，其中，参数 request 和 response 为 Web 服务器或 Filter 链中的上一个 Filter 传递过来的请求和响应对象；参数 chain 代表当前 Filter 链的对象，在当前 Filter 对象中的 doFilter()方法内部需要调用 FilterChain 对象的 doFilter()方法，才能把请求交付给 Filter 链中的下一个 Filter 或者目标程序去处理。
destroy()	destroy()方法在 Web 服务器卸载 Filter 对象之前被调用，该方法用于释放被 Filter 对象打开的资源，例如关闭数据库和 IO 流。

表 4-1 中的这三个方法都是 Filter 的生命周期方法，其中 init()方法在 Web 应用程序加载的时候调用，destroy()方法在 Web 应用程序卸载的时候调用，这两个方法都只会被调用一次，而 doFilter()方法只要有客户端请求时就会被调用，并且 Filter 所有的工作集中在 doFilter()方法中。

4.1.2　实现第一个 Filter 程序

为了帮助大家快速了解 Filter 的开发过程，接下来，分步骤带大家实现第一个 Filter 程序，具体如下：

（1）在 Eclipse 中创建一个名为 chapter04 的 Web 工程，然后在该工程的 Java Resources/src 目录下创建 cn.itcast.chapter04.filter 包，并在该包下创建一个 MyServlet.java 程序，如例 4-1 所示。

例4-1　　MyServlet.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class MyServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request, HttpServletResponse response)
7              throws ServletException, IOException {
8          response.getWriter().write("Hello MyServlet ");
9      }
10     public void doPost(HttpServletRequest request, HttpServletResponse response)
11             throws ServletException, IOException {
12         doGet(request, response);
13     }
14 }
```

（2）在 web.xml 文件中对 servlet 进行如下配置。

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>cn.itcast.chapter04.filter.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
```

部署 chapter04 工程到 Tomcat 服务器，启动 Tomcat 服务器，在浏览器的地址栏中输入 URL 地址 http://localhost:8080/chapter04/MyServlet，此时，可以看到浏览器成功访问到了 MyServlet 程序，具体如图 4-2 所示。

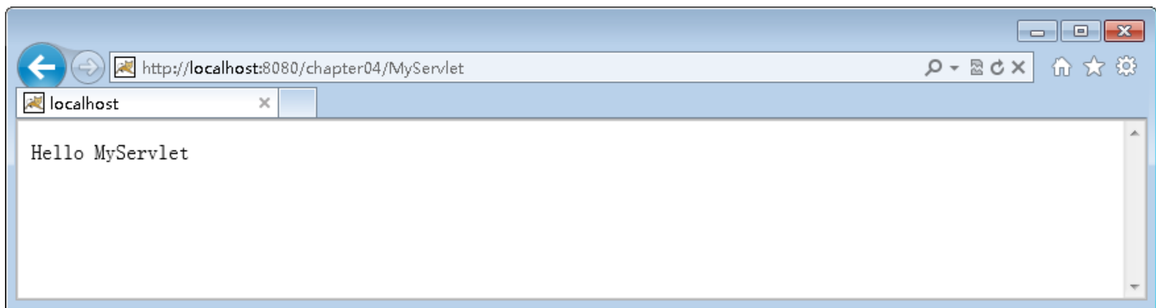

图4-2　运行结果

（3）拦截 MyServlet 程序。在 cn.itcast.chapter04.filter 包中创建过滤器 MyFilter，该类用于拦截 MyServlet 程序，MyFilter 的实现代码如例 4-2 所示。

例4-2　　MyFilter.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.Filter;
4  import javax.servlet.*;
5  public class MyFilter implements Filter {
6      public void destroy() {
7          // 过滤器对象在销毁时自动调用，释放资源
8      }
9      public void doFilter(ServletRequest request, ServletResponse response,
10             FilterChain chain) throws IOException, ServletException {
11         // 用于拦截用户的请求，如果和当前过滤器的拦截路径匹配，该方法会被调用
12         PrintWriter out=response.getWriter();
13         out.write("Hello MyFilter");
14     }
15     public void init(FilterConfig fConfig) throws ServletException {
16         // 过滤器对象在初始化时调用，可以配置一些初始化参数
17     }
18 }
```

过滤器程序与 Servlet 程序类似，同样需要在 web.xml 文件中进行配置，从而设置它所能拦截的资源，具体代码如下：

```
<filter>
  <filter-name>MyFilter</filter-name>
  <filter-class>cn.itcast.chapter04.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>MyFilter</filter-name>
  <url-pattern>/MyServlet</url-pattern>
</filter-mapping>
```

在上述代码中，包含多个元素，这些元素分别具有不同的作用，具体如下：

- <filter>根元素用于注册一个 Filter
- <filter-name>子元素用于设置 Filter 名称
- <filter-class>子元素用于设置 Filter 类的完整名称
- <filter-mapping>根元素用于设置一个过滤器所拦截的资源
- <filter-name>子元素必须与<filter>中的<filter-name>子元素相同
- <url-pattern>子元素用于匹配用户请求的 URL，例如“/MyServlet”，这个 URL 还可以使用通配符“*”来表示，例如“*.do”适用于所有以“.do”开头结尾的 Servlet 路径。

重新启动 Tomcat 服务器，在浏览器的地址栏中输入 URL 地址 http://localhost:8080/chapter04/MyServlet，访问 MyServlet，此时，浏览器窗口显示的结果如图 4-3 所示。

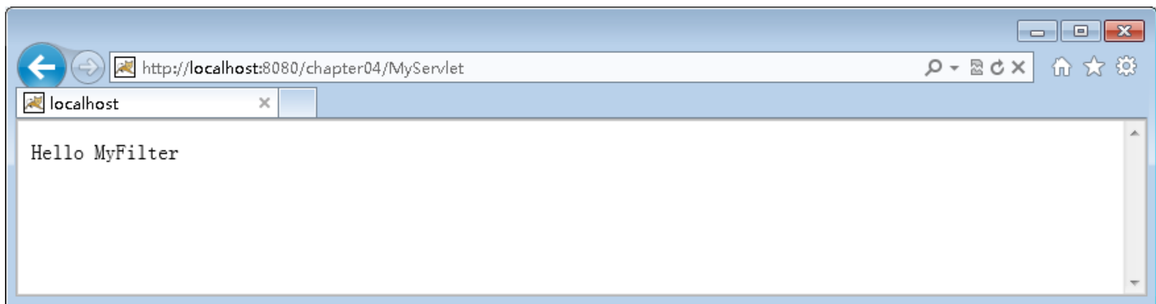

图4-3 MyServlet.java

从图 4-3 可以看出，在使用浏览器访问 MyServlet 时，浏览器窗口中只显示了 MyFilter 的输出信息，并没有显示 MyServlet 的输出信息，说明 MyFilter 成功拦截了 MyServlet 程序。

4.1.3 Filter 映射

通过 4.1.2 小节的学习，了解到 Filter 拦截的资源需要在 web.xml 文件中进行配置，即 Filter 映射。Filter 的映射方式可分为两种，具体如下：

1、 使用通配符“*”拦截用户所有请求

Filter 的<filter-mapping>元素可以配置过滤器所有拦截的资源，如果想让过滤器拦截所有的请求访问，那么需要使用通配符“*”来实现，具体示例如下：

```
<filter>
   <filter-name>Filter1</filter-name>
   <filter-class>cn.itcast.chapter04.filter.MyFilter </filter-class>
</filter>
<filter-mapping>
   <filter-name>Filter1</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

2、 拦截不同方式的访问请求

在 web.xml 文件中，一个<filter-mapping>元素用于配置一个 Filter 所负责拦截的资源。<filter-mapping>元素中有一个特殊的子元素<dispatcher>，该元素用于指定过滤器所拦截的资源被 Servlet 容器调用的方式，<dispatcher>元素的值共有四个，具体如下：

- 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，那么该过滤器将不会被调用。

- 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。

为了大家更好地理解上述四个值的作用，接下来以 FORWARD 为例，分步骤演示 Filter 对转发请求的拦截效果，具体如下：

（1）在 chapter04 工程的 cn.itcast.chapter04. filter 包中，创建一个 ServletTest.java 程序，该程序用于将请求转发给 first.jsp 页面，如例 4-3 所示。

例4-3　　ServletTest.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class ServletTest extends HttpServlet {
6      public void doGet(HttpServletRequest request, HttpServletResponse response)
7              throws ServletException, IOException {
8          request.getRequestDispatcher("/first.jsp").forward(request, response);
9      }
10     public void doPost(HttpServletRequest request, HttpServletResponse response)
11             throws ServletException, IOException {
12         doGet(request, response);
13     }
14 }
```

（2）在 chapter04 工程的 WebContent 目录中创建一个 first.jsp 页面，该页面用于输出内容，如例 4-4 所示。

例4-4　　first.jsp

```
1  <%@ page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8"%>
3  <html>
4  <head></head>
5  <body>
6      first.jsp
7  </body>
8  </html>
```

（3）在 chapter04 工程的 cn.itcast.chapter04.filter 包中，创建一个 FilterTest.java 程序，专门用于拦截 first.jsp 页面，如例 4-5 所示。

例4-5　　FilterTest.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
```

```
3  import javax.servlet.*;
4  public class FilterTest implements Filter {
5      public void destroy() {
6          // 过滤器对象在销毁时自动调用，释放资源
7      }
8      public void doFilter(ServletRequest request, ServletResponse response,
9              FilterChain chain) throws IOException, ServletException {
10         // 用于拦截用户的请求，如果和当前过滤器的拦截路径匹配，该方法会被调用
11         PrintWriter out=response.getWriter();
12         out.write("Hello FilterTest");
13     }
14     public void init(FilterConfig fConfig) throws ServletException {
15         // 过滤器对象在初始化时调用，可以配置一些初始化参数
16      }
17 }
```

（4）在 web.xml 文件中，配置 Filter 过滤器，拦截 first.jsp 页面，具体代码如下：

```
<filter>
  <filter-name>FilterTest</filter-name>
  <filter-class>cn.itcast.chapter04.filter.FilterTest</filter-class>
</filter>
<filter-mapping>
  <filter-name>FilterTest</filter-name>
  <url-pattern>/first.jsp</url-pattern>
</filter-mapping>
```

（5）启动 Tomcat 服务器，在浏览器中输入 URL 地址 http://localhost:8080/chapter04/ServletTest 访问 ServletTest，浏览器显示的结果如图 4-4 所示。

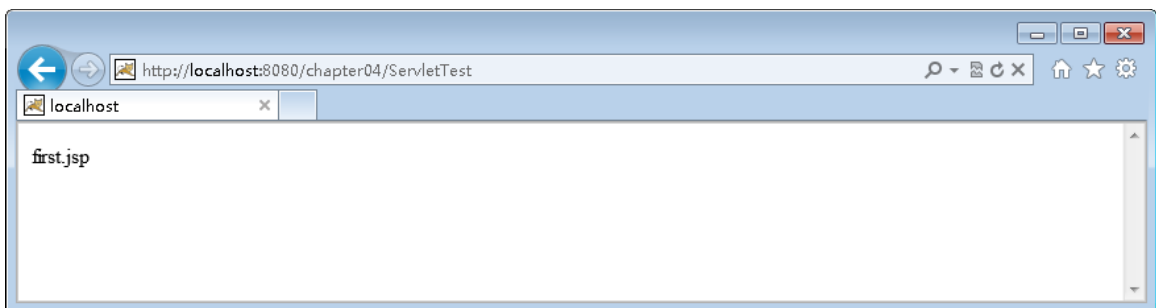

图4-4 运行结果

从图 4-4 中可以看出，浏览器可以正常访问 JSP 页面，说明 FilterTest 没有拦截到 ServletTest 转发的 first.jsp 页面。

（6）为了拦截 ServletTest 通过 forward()方法转发的 first.jsp 页面，需要在 web.xml 文件中的增加一个 <dispatcher>元素，将该元素的值设置为 FORWARD，修改后的 FilterTest 的映射如下所示：

```
<filter>
  <filter-name>FilterTest</filter-name>
  <filter-class>cn.itcast.chapter04.filter.FilterTest</filter-class>
</filter>
<filter-mapping>
```

```
      <filter-name>FilterTest</filter-name>
      <url-pattern>/first.jsp</url-pattern>
      <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
```

（7）启动 Tomcat 服务器，在浏览器的地址栏输入 URL 地址 http://localhost:8080/chapter04/ServletTest 访问 ServletTest，浏览器显示的结果如图 4-5 所示。

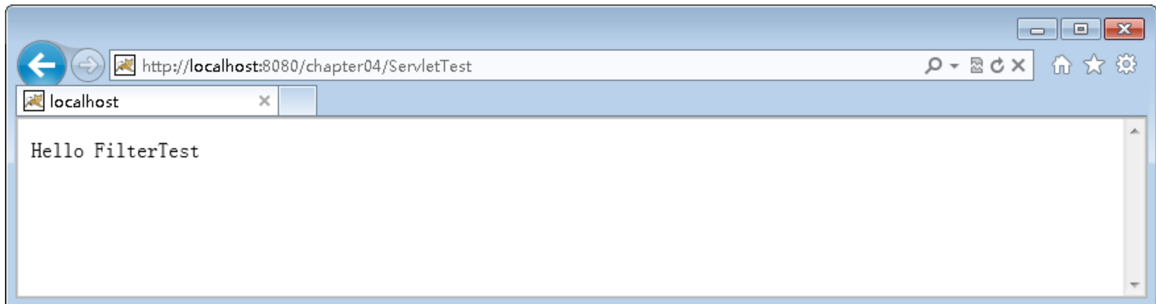

图4-5　运行结果

从图 4-5 中可以看出，浏览器窗口显示的是 FilterTest 中的内容，而 first.jsp 页面的输出内容没有显示。由此可见，ServletTest 中通过 forward()方法转发的 first.jsp 页面被成功拦截了。

4.1.4　Filter 链

在一个 Web 应用程序中可以注册多个 Filter 程序，每个 Filter 程序都可以针对某一个 URL 进行拦截。如果多个 Filter 程序都对同一个 URL 进行拦截，那么这些 Filter 就会组成一个 Filter 链（也叫过滤器链）。Filter 链用 FilterChain 对象来表示，FilterChain 对象中有一个 doFilter()方法，该方法作用就是让 Filter 链上的当前过滤器放行，请求进入下一个 Filter，接下来通过一个图例来描述 Filter 链的拦截过程，如图 4-6 所示。

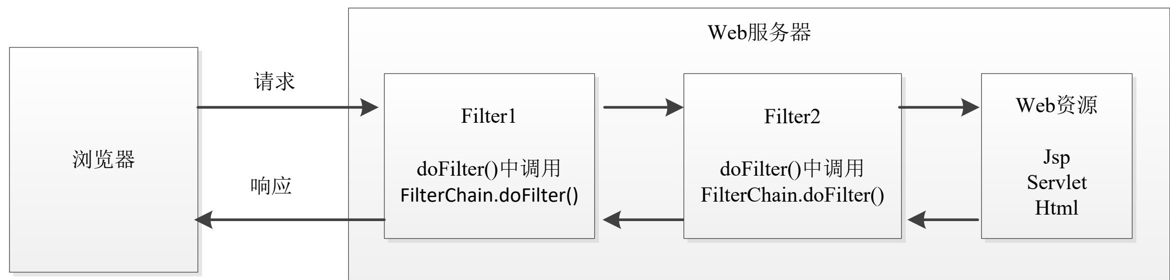

图4-6　Filter 链

在图 4-6 中，当浏览器访问 Web 服务器中的资源时需要经过两个过滤器 Filter1 和 Filter2，首先 Filter1 会对这个请求进行拦截，在 Filter1 过滤器中处理好请求后，通过调用 Filter1 的 doFilter()方法将请求传递给 Filter2，Filter2 将用户请求处理后同样调用 doFilter()方法，最终将请求发送给目标资源。当 Web 服务器对这个请求做出响应时，也会被过滤器拦截，这个拦截顺序与之前相反，最终将响应结果发送给客户端。

为了让读者更好的学习 Filter 链，接下来，通过一个案例，分步骤演示如何使用 Filter 链拦截 MyServlet 的同一个请求，具体如下：

（1）在 chapter04 工程的 cn.itcast.chapter04.filter 包中新建 MyFilter01 和 MyFilter02，如例 4-6 和 4-7 所示。

例4-6　MyFilter01.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
```

```
3  import javax.servlet.*;
4  public class MyFilter01 implements Filter {
5      public void destroy() {
6          // 过滤器对象在销毁时自动调用，释放资源
7      }
8      public void doFilter(ServletRequest request, ServletResponse response,
9              FilterChain chain) throws IOException, ServletException {
10         // 用于拦截用户的请求，如果和当前过滤器的拦截路径匹配，该方法会被调用
11         PrintWriter out=response.getWriter();
12         out.write("Hello MyFilter01</br>");
13         chain.doFilter(request, response);
14     }
15     public void init(FilterConfig fConfig) throws ServletException {
16         // 过滤器对象在初始化时调用，可以配置一些初始化参数
17     }
18 }
```

例4-7　　MyFilter02.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.Filter;
4  import javax.servlet.*;
5  public class MyFilter02 implements Filter {
6      public void destroy() {
7          // 过滤器对象在销毁时自动调用，释放资源
8      }
9      public void doFilter(ServletRequest request, ServletResponse response,
10             FilterChain chain) throws IOException, ServletException {
11         // 用于拦截用户的请求，如果和当前过滤器的拦截路径匹配，该方法会被调用
12         PrintWriter out=response.getWriter();
13         out.write("MyFilter02 Before</br>");
14         chain.doFilter(request, response);
15         out.write("</br>MyFilter02 After</br>");
16     }
17     public void init(FilterConfig fConfig) throws ServletException {
18         // 过滤器对象在初始化时调用，可以配置一些初始化参数
19     }
20 }
```

（2）在 web.xml 文件中将 MyFilter01 和 MyFilter02 注册在 MyServlet 前面，具体如下所示：

```
<filter>
    <filter-name>MyFilter01</filter-name>
    <filter-class>cn.itcast.chapter04.filter.MyFilter01</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>MyFilter01</filter-name>
```

```
    <url-pattern>/MyServlet</url-pattern>
  </filter-mapping>
  <filter>
    <filter-name>MyFilter02</filter-name>
    <filter-class>cn.itcast.chapter04.filter.MyFilter02</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>MyFilter02</filter-name>
    <url-pattern>/MyServlet</url-pattern>
  </filter-mapping>
  <servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>cn.itcast.chapter04.filter.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
  </servlet-mapping>
```

（3）重新启动 Tomcat 服务器，在浏览器地址栏中输入 http://localhost:8080/chapter04/MyServlet，此时，浏览器窗口中的显示结果如图 4-7 所示。

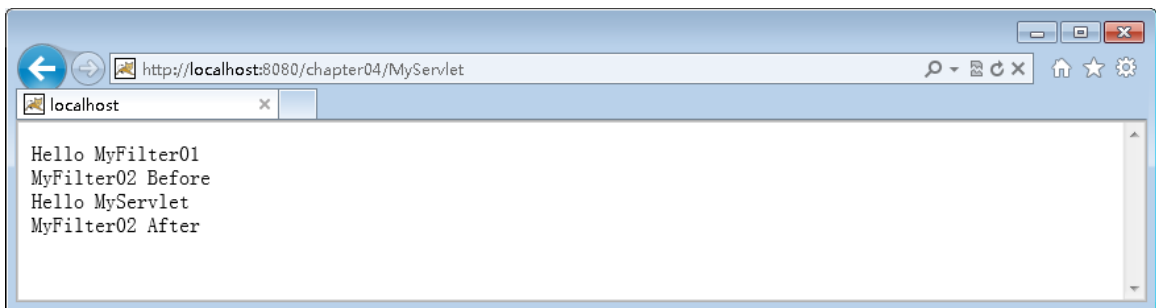

图4-7　运行结果

从图 4-7 中可以看出，MyServlet 首先被 MyFilter01 拦截了，打印出 MyFilter01 中的内容，然后被 MyFilter02 拦截，直到 MyServlet 被 MyFilter02 放行后，浏览器才显示出 MySerlvet 中的输出内容。

需要注意的是，Filter 链中各个 Filter 的拦截顺序与它们在 web.xml 文件中<filter-mapping>元素的映射顺序一致，由于 MyFilter01 的<filter-mapping>元素位于 MyFilter02 的<filter-mapping>元素前面，因此用户的访问请求首先会被 MyFilter01 拦截，然后再被 MyFilter02 拦截。

4.1.5　FilterConfig 接口

为了获取 Filter 程序在 web.xml 文件中的配置信息，Servlet API 提供了一个 FilterConfig 接口，该接口封装了 Filter 程序在 web.xml 中的所有注册信息，并且提供了一系列获取这些配置信息的方法，具体如表 4-2 所示。

表4-2　FilterConfig 接口中的方法

方法声明	功能描述
String getFilterName ()	getFilterName()方法用于返回在 web.xml 文件中为 Filter 所设置

	的名称，也就是返回<filter-name>元素的设置值。
ServletContext getServletContext()	getServletContext()方法用于返回 FilterConfig 对象中所包装的 ServletContext 对象的引用。
String getInitParameter(String name)	getInitParameter(String name)方法用于返回在 web.xml 文件中为 Filter 所设置的某个名称的初始化参数值，如果指定名称的初始化参数不存在，则返回 null。
Enumeration getInitParameterNames()	getInitParameterNames()方法用于返回一个 Enumeration 集合对象，该集合对象中包含在 web.xml 文件中为当前 Filter 设置的所有初始化参数的名称。

表 4-2 列举了 FilterConfig 接口中的一系列方法，为了让读者更好的掌握这些方法，接下来，以 getInitParameter(String name)方法为例，通过一个案例来演示 FilterConfig 接口的作用，具体如例 4-8 所示。

例4-8　　MyFilter03.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  public class MyFilter03 implements Filter {
5      private String characterEncoding;
6      FilterConfig fc;
7      public void destroy() {
8      }
9      public void doFilter(ServletRequest request, ServletResponse response,
10             FilterChain chain) throws IOException, ServletException {
11         // 输出参数信息
12         characterEncoding=fc.getInitParameter("encoding");
13         System.out.println("encoding初始化参数的值为："+characterEncoding);
14         chain.doFilter(request, response);
15     }
16     public void init(FilterConfig fConfig) throws ServletException {
17         // 获取FilterConfig对象
18         this.fc = fConfig;
19     }
20 }
```

接下来在 web.xml 文件中部署过滤器。由于 Filter 链中各个 Filter 的拦截顺序与它们在 web.xml 文件中<filter-mapping>元素的映射顺序一致，因此，为了防止其他 Filter 影响 MyFilter03 的拦截效果，我们将 MyFilter03 注册在 web.xml 文件最前端，具体注册代码如下：

```
<filter>
    <filter-name>MyFilter03</filter-name>
    <filter-class>cn.itcast.chapter04.filter.MyFilter03</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>GBK</param-value>
    </init-param>
</filter>
<filter-mapping>
```

```
        <filter-name>MyFilter03</filter-name>
        <url-pattern>/MyServlet</url-pattern>
    </filter-mapping>
```

重新启动 Tomcat 服务器，在浏览器地址栏中输入 http://localhost:8080/chapter04/MyServlet 访问 MyServlet，控制台窗口中显示的结果如图 4-8 所示。

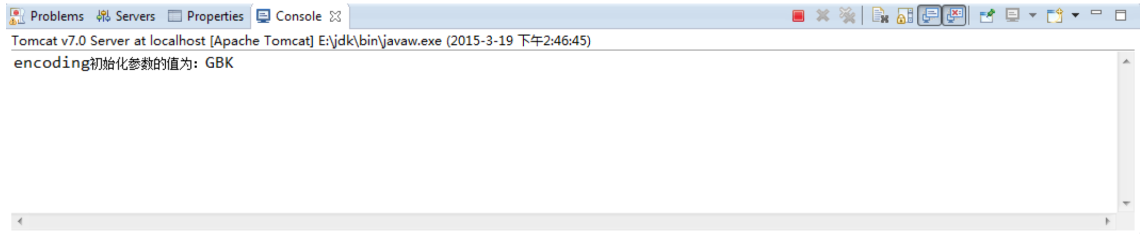

图4-8　控制台窗口

从图 4-8 中可以看出，使用 Filter 获取到了配置文件中的初始化参数。当 Tomcat 服务器启动时，会加载所有的 Web 应用，当加载到 chapter04 这个 Web 应用时，FirstFilter 就会被初始化调用 init()方法，从而可以得到 FilterConfig 对象，然后在 doFilter()方法中通过调用 FilterConfig 对象的 getInitParameter()方法便可以获取在 web.xml 文件中配置的参数信息。

4.2　应用案例—Filter 实现用户自动登录

通过前面的学习，了解到 Cookie 可以实现用户自动登录的功能。当用户第一次访问服务器时，服务器会发送一个包含用户信息的 Cookie。之后，当客户端再次访问服务器时，都会向服务器回送 Cookie，这样，服务器就可以从 Cookie 中获取用户信息，从而实现用户的自动登录功能，具体如图 4-9 所示。

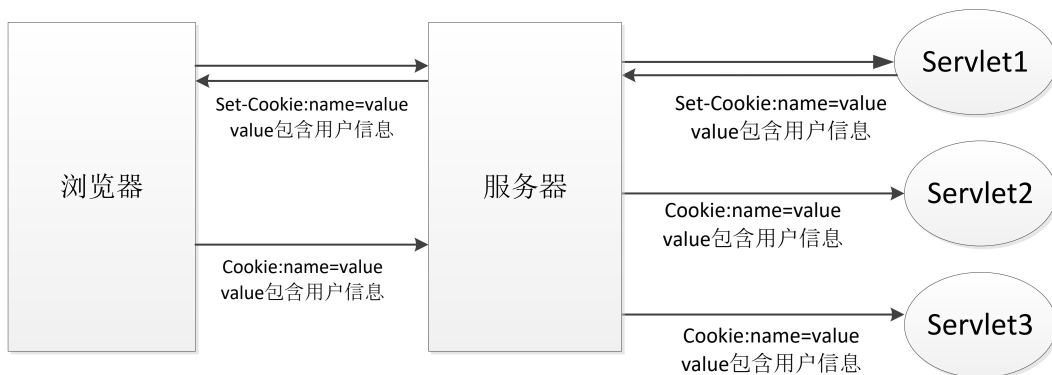

图4-9　Cookie 实现用户登录

从图 4-9 中可以看出，使用 Cookie 实现用户自动登录后，当客户端访问服务器的 Servlet 时，所有的 Servlet 都需要对用户的 Cookie 信息进行校验，这样势必会导致在 Serlvet 程序中书写大量重复的代码。

为了解决上面的问题，可以在 Filter 程序中实现 Cookie 的校验。由于 Filter 是可以对服务器的所有请求进行拦截，因此，一旦请求通过 Filter 程序，就相当于用户信息校验通过，Servlet 程序根据获取到的用户信息，就可以实现自动登录了。接下来，通过一个案例来演示如何使用 Filter 实现用户的自动登录功能，具体步骤如下：

（1） 编写 User.java 程序

在 chapter04 工程创建 cn.itcast.chapter04.entity 包，在该包中编写 User.java 程序，该程序用于封装用户的信息，如例 4-9 所示。

例4-9　User.java

```
1  package cn.itcast.chapter04.entity;
```

```
2  public class User {
3      private String username;
4      private String password;
5      public String getUsername() {
6          return username;
7      }
8      public void setUsername(String username) {
9          this.username = username;
10     }
11     public String getPassword() {
12         return password;
13     }
14     public void setPassword(String password) {
15         this.password = password;
16     }
17 }
```

（2） 编写 login.jsp 页面

在 chapter04 工程的 WebContent 根目录中，编写 login.jsp 页面，该页面用于创建一个用户登录的表单，这个表单需要填写用户名和密码，以及用户自动登录的时间，如例 4-10 所示。

例4-10　login.jsp

```
1  <%@ page language="java" contentType="text/html; charset=utf-8"
2  pageEncoding="utf-8" import="java.util.*"
3  %>
4  <html>
5  <head></head>
6  <center><h3>用户登录</h3></center>
7  <body style="text-align: center;">
8  <form action="${pageContext.request.contextPath }/LoginServlet" method="post">
9  <table border="1" width="600px" cellpadding="0" cellspacing="0" align="center" >
10     <tr>
11         <td height="30" align="center">用户名：</td>
12         <td>  <input type="text" name="username" />${errerMsg }</td>
13     </tr>
14     <tr>
15         <td height="30" align="center">密    码：</td>
16         <td>  <input type="password" name="password" /></td>
17     </tr>
18     <tr>
19         <td height="35" align="center">自动登录时间</td>
20         <td><input type="radio" name="autologin" value="${60*60*24*31 }" />一个月
21             <input type="radio" name="autologin" value="${60*60*24*31*3 }" />三个月
22             <input type="radio" name="autologin" value="${60*60*24*31*6 }" />半年
23             <input type="radio" name="autologin" value="${60*60*24*31*12 }" />一年
24         </td>
```

```
25      </tr>
26      <tr>
27          <td height="30" colspan="2" align="center">
28              <input type="submit" value="登录" />    
29              <input type="reset" value="重置" />
30          </td>
31      </tr>
32 </table>
33 </form>
34 </body>
35 <html>
```

（3）编写 index.jsp 页面

在 chapter04 工程的 WebContent 根目录中，编写 index.jsp 页面，该页面用于显示用户的登录信息，如果没有用户登录，在 index.jsp 页面中就显示一个用户登录的超链接，如果用户已经登录，在 index.jsp 页面中显示登录的用户名，以及一个注销的超链接，如例 4-11 所示。

例4-11　index.jsp

```
1  <%@ page language="java" contentType="text/html; charset=utf-8"
2  pageEncoding="utf-8" import="java.util.*"
3  %>
4  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
5  <html>
6  <head>
7  <title>显示登录的用户信息</title>
8  </head>
9  <body>
10     <br>
11     <center>
12         <h3>欢迎光临</h3>
13     </center>
14     <br>
15     <br>
16     <c:choose>
17         <c:when test="${sessionScope.user==null }">
18             <a href="${pageContext.request.contextPath }/login.jsp">用户登录</a>
19         </c:when>
20         <c:otherwise>
21       欢迎你，${sessionScope.user.username }!
22       <a href="${pageContext.request.contextPath }/LogoutServlet">注销</a>
23         </c:otherwise>
24     </c:choose>
25     <hr>
26 </body>
27 </html>
```

（4）编写 LoginServlet.java 程序

在 chapter04 工程的 cn.itcast.chapter04.servlet 包中，编写 LoginServlet.java 程序，该程序用于处理用户的登录请求，如果输入的用户名和密码正确，则发送一个用户自动登录的 Cookie，并跳转到首页，否则会提示输入的用户名或密码错误，并跳转至登录页面 login.jsp 让用户重新登录，如例 4-12 所示。

例4-12　LoginServlet.java

```
1  package cn.itcast.chapter04.servlet;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import cn.itcast.chapter04.entity.User;
6  public class LoginServlet extends HttpServlet {
7      public void doGet(HttpServletRequest request, HttpServletResponse response)
8              throws ServletException, IOException {
9          // 获得用户名和密码
10         String username = request.getParameter("username");
11         String password = request.getParameter("password");
12         // 检查用户名和密码
13         if ("itcast".equals(username) && "123456".equals(password)) {
14             // 登录成功
15             // 将用户状态 user 对象存入 session 域
16             User user = new User();
17             user.setUsername(username);
18             user.setPassword(password);
19             request.getSession().setAttribute("user", user);
20             // 发送自动登录的 cookie
21             String autoLogin = request.getParameter("autologin");
22             if (autoLogin != null) {
23                 // 注意 cookie 中的密码要加密
24                 Cookie cookie = new Cookie("autologin", username + "-"
25                         + password);
26                 cookie.setMaxAge(Integer.parseInt(autoLogin));
27                 cookie.setPath(request.getContextPath());
28                 response.addCookie(cookie);
29             }
30             // 跳转至首页
31             response.sendRedirect(request.getContextPath()+"/index.jsp");
32         } else {
33             request.setAttribute("errerMsg", "用户名或密码错");
34             request.getRequestDispatcher("/login.jsp").forward(request,
35                     response);
36         }
37     }
38     public void doPost(HttpServletRequest request, HttpServletResponse response)
39             throws ServletException, IOException {
40         doGet(request, response);
```

```
41      }
42 }
```

（5） 编写 LogoutServlet.java 程序

在 chapter04 工程的 cn.itcast.chapter04.servlet 包中，编写 LogoutServlet.java 程序，该程序用于注销用户登录的信息，在这个程序中首先会将 Session 会话中保存的 User 对象删除，然后将自动登录的 Cookie 删除，最后跳转到 index.jsp，如例 4-13 所示。

例4-13　LogoutServlet.java

```
1  package cn.itcast.chapter04.servlet;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class LogoutServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request, HttpServletResponse response)
7              throws ServletException, IOException {
8          // 用户注销
9          request.getSession().removeAttribute("user");// 从 session 中移除 user
10         // 从客户端删除自动登录的 cookie
11         Cookie cookie = new Cookie("autologin", "msg");
12         cookie.setPath(request.getContextPath());
13         cookie.setMaxAge(0);
14         response.addCookie(cookie);
15         response.sendRedirect(request.getContextPath()+"/index.jsp");
16     }
17     public void doPost(HttpServletRequest request, HttpServletResponse response)
18             throws ServletException, IOException {
19         doGet(request, response);
20     }
21 }
```

（6） 编写 AutoLoginFilter.java 过滤器程序

在 chapter04 工程的 cn.itcast.chapter04.filter 包中，编写 AutoLoginFilter.java 程序，该程序用于拦截用户登录的访问请求，判断请求中是否包含用户自动登录的 Cookie，如果包含则获取 Cookie 中的用户名和密码，并验证用户名和密码是否正确，如果正确，则将用户的登录信息封装到 User 对象存入 Session 域中，完成用户自动登录，如例 4-14 所示。

例4-14　AutoLoginFilter.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import cn.itcast.chapter04.entity.User;
6  public class AutoLoginFilter implements Filter {
7      public void init(FilterConfig filterConfig) throws ServletException {
8      }
9      public void doFilter(ServletRequest req, ServletResponse response,
10             FilterChain chain) throws IOException, ServletException {
```

```
11          HttpServletRequest request = (HttpServletRequest) req;
12          // 获得一个名为 autologin 的 cookie
13          Cookie[] cookies = request.getCookies();
14          String autologin = null;
15          for (int i = 0; cookies != null && i < cookies.length; i++) {
16              if ("autologin".equals(cookies[i].getName())) {
17                  // 找到了指定的 cookie
18                  autologin = cookies[i].getValue();
19                  break;
20              }
21          }
22          if (autologin != null) {
23              // 做自动登录
24              String[] parts = autologin.split("-");
25              String username = parts[0];
26              String password = parts[1];
27              // 检查用户名和密码
28              if ("itcast".equals(username)&& ("123456").equals(password)) {
29                  // 登录成功,将用户状态 user 对象存入 session 域
30                  User user = new User();
31                  user.setUsername(username);
32                  user.setPassword(password);
33                  request.getSession().setAttribute("user", user);
34              }
35          }
36          // 放行
37          chain.doFilter(request, response);
38      }
39      public void destroy() {
40      }
41 }
```

在 web.xml 文件中，配置 AutoLoginFilter 过滤器，由于要拦截用户访问资源的所有请求，因此将<filter-mapping>元素拦截的路径设置为/*，具体代码如下：

```
<filter>
  <filter-name>AutoLoginFilter</filter-name>
  <filter-class> cn.itcast.chapter04.filter.AutoLoginFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>AutoLoginFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

（7） 访问 login.jsp 页面

重新启动 Web 服务，打开 IE 浏览器，在地址栏中输入 http://localhost:8080/chapter04/login.jsp

此时，浏览器窗口中会显示一个用户登录的表单，在这个表单中输入用户名 itcast、密码 123456，并选择用户自动登录的时间，如图 4-10 所示。

图4-10 运行结果

（8）实现用户登录

点击图 4-10 中的登录按钮，便可完成用户自动登录，此时，在浏览器窗口中会显示登录的用户名，如图 4-11 所示。

图4-11 运行结果

从图 4-11 可以看出，用户已经登录成功了，此时再开启一个 IE 浏览器，在地址栏中直接输 http://localhost:8080/chapter04/index.jsp 仍可以看到用户的登录信息，因此可以说明完成了用户自动登录的功能。

（9）注销用户。

点击图 4-11 中的注销超链接，就可以注销当前的用户，然后显示 index.jsp 页面，如图 4-12 所示。

图4-12 运行结果

4.3 Filter 高级应用

通过前面的学习，了解到 Filter 过滤器可以获取到代表用户请求和响应的 request、response 对象。可是如果想对 request 和 response 对象中的任何信息进行修改，则需要通过包装类来实现。在 Servlet API 中，提供了 HttpServletRequestWrapper 和 HttpServletResponseWrapper 两个类，它们分别是 request 和 response 对象的包装类。接下来，本节将围绕 Filter 程序中包装类的使用进行详细地讲解。

4.3.1 装饰设计模式

HttpServletRequestWrapper 和 HttpServletResponseWrapper 作为 request 和 response 对象的包装类，它们都采用了装饰设计模式。所谓装饰设计模式，指的是通过包装类的方式，动态增强某个类的功能。为了大家更好的理解装饰设计模式，接下来，先来简单介绍一下装饰设计模式的特点，具体如下：

- 包装类要和被包装对象实现同样的接口。
- 包装类持有一个被包装对象，例如在 HttpServletRequestWrapper 定义的构造方法中，需要传递一个 HttpServletRequest 类型的参数。
- 包装类在实现接口的过程中，对于不需要包装的方法原封不动地调用被包装对象的方法来实现，对于需要包装的方法自己实现。

了解了装饰设计模式的特点，接下来，通过一个案例来演示如何实现装饰设计模式，在 chapter04 工程的 cn.itcast.chapter04. decorator 包中，编写 PhoneDemo.java 程序，具体如例 4-15 所示。

例4-15 PhoneDemo.java

```
1  package cn.itcast.chapter04.decorator;
2  /**
3   * 手机
4   */
5  interface Phone{
6      // 手机的功能
7      void action();
8  }
9  /**
10  * 非智能手机
11  */
12 class Non_SmartPhone implements Phone{
13     // 非智能机具有打电话的功能
14     public void action() {
15         System.out.println("可以打电话");
16     }
17 }
18 /**
19  * 智能手机
20  */
21 class SmartPhone implements Phone{
22     private Phone nonSmartPhone;
23     public SmartPhone(Phone nonSmartPhone){
```

```
24          this.nonSmartPhone = nonSmartPhone;
25      }
26      //智能机拥有打电话和玩愤怒的小鸟的功能
27      public void action() {
28          nonSmartPhone.action();
29          System.out.println("可以玩愤怒的小鸟"); // 在非智能机的基础上，实现功能的增强
30      }
31 }
32 public class PhoneDemo {
33      public static void main(String[] args) {
34          Phone nPhone = new Non_SmartPhone();
35          System.out.println("--------------手机装饰前--------------");
36          nPhone.action();
37          Phone smartPhone = new SmartPhone(nPhone);
38          System.out.println("--------------手机装饰后--------------");
39          smartPhone.action();
40      }
41 }
```

在例 4-15 中，Non_SmartPhone 类表示非智能手机，它是属于被包装类，SmartPhone 类表示智能手机，它是 Non_SmartPhone 类的包装类，Non_SmartPhone 类和 SmartPhone 类实现了相同的接口 Phone。第 22 行代码用于在 SmartPhone 类中持有被包装类 Non_SmartPhone 的对象，第 29 行代码用于在被包装类的基础上，实现功能的增强。

程序的运行结果如图 4-13 所示。

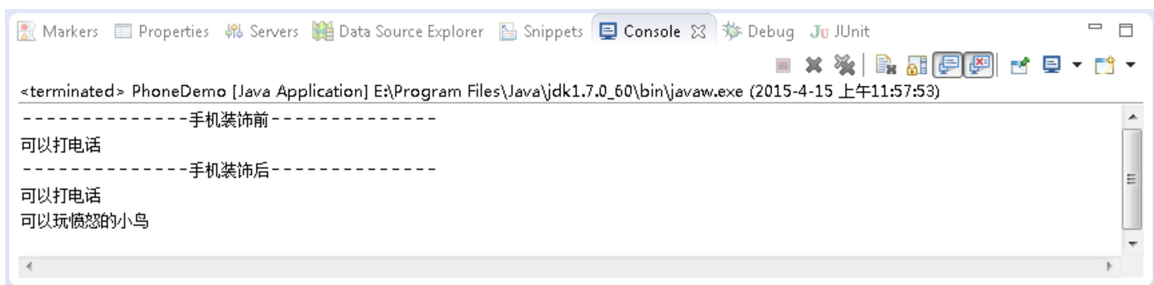

图4-13　运行结果

从图 4-13 中可以看出，SmartPhone 对 Non_SmartPhone 类进行包装后，SmartPhone 类型的对象不仅具有了“打电话”功能，还具有了“玩游戏”的功能。

4.3.2 Filter 实现统一全站编码

在 Web 开发中，经常会遇到中文乱码问题，按照前面所学的知识，解决乱码的通常做法都是在 Servlet 程序中设置编码方式，但是，如果多个 Servlet 程序都需要设置编码方式，势必会书写大量重复的代码。

为了解决上面的问题，我们可以在 Filter 中对获取到的请求和响应消息进行编码，从而统一全站的编码方式。接下来，分步骤讲解如何使用 Filter 实现统一全站的编码，具体如下：

（1）编写 form.jsp 页面

在 chapter04 工程的 WebContent 根目录中，编写一个 form.jsp 页面，用于提交用户登录的表单信息，如例 4-16 所示。

例4-16　form.jsp

```
1  <%@ page language="java" contentType="text/html; charset=utf-8"
2  pageEncoding="utf-8" import="java.util.*"
3  %>
4  
5  <html>
6  <head></head>
7  <center>
8      <h3>用户登录</h3>
9  </center>
10 <body style="text-align: center;">
11     <a href="<%=request.getContextPath()%>/CharacterServlet?name=
12               传智播客&password=123456">点击超链接登录</a>
13     <form action="<%=request.getContextPath()%>/CharacterServlet"
14         method="post">
15         <table border="1" width="600px" cellpadding="0" cellspacing="0"
16             align="center">
17             <tr>
18                 <td height="30" align="center">用户名：</td>
19                 <td> <input type="text" name="name" /></td>
20             </tr>
21             <tr>
22                 <td height="30" align="center">密   码：</td>
23                 <td> <input type="password" name="password" /></td>
24             </tr>
25             <tr>
26                 <td height="30" colspan="2" align="center"><input type="submit"
27                     value="登录" />     <input type="reset"
28                     value="重置" /></td>
29             </tr>
30         </table>
31     </form>
32 </body>
33 <html>
```

（2）编写 CharacterServlet.java 程序

在 chapter04 工程的 cn.itcast.chapter04.servlet 包中，编写一个 CharacterServlet.java 程序，该程序用于获取用户输入的请求参数，并将参数输出到控制台，如例 4-17 所示。

例4-17　CharacterServlet.java

```
1  package cn.itcast.chapter04.servlet;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class CharacterServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request, HttpServletResponse response)
7              throws ServletException, IOException {
```

```
8           System.out.println(request.getParameter("name"));
9           System.out.println(request.getParameter("password"));
10      }
11      public void doPost(HttpServletRequest request, HttpServletResponse response)
12              throws ServletException, IOException {
13          doGet(request, response);
14      }
15 }
```

（3）编写 CharacterFilter.java 过滤器

CharacterFilter 类用于拦截用户的请求访问，实现统一全站编码的功能。但是，由于请求方式 post 和 get 解决乱码方式的不同，post 方式的请求参数存放在消息体中，可以通过 setCharacterEncoding()方法进行设置，而 get 方式的请求参数存放在消息头中，必须得通过获取 URI 参数才能进行设置。如果每次单独对 get 请求方式进行处理，势必会很麻烦，为此，可以通过 HttpServletRequestWrapper 类对 HttpServletRequest 进行包装，通过重写 getParameter()的方式来设置 get 方式提交参数的编码，CharacterFilter 类的实现代码如例 4-18 所示。

例4-18　CharacterFilter.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class CharacterFilter implements Filter {
6      public void init(FilterConfig filterConfig) throws ServletException {
7      }
8      public void doFilter(ServletRequest req, ServletResponse resp,
9              FilterChain chain) throws IOException, ServletException {
10         HttpServletRequest request = (HttpServletRequest) req;
11         HttpServletResponse response = (HttpServletResponse) resp;
12         // 拦截所有的请求 解决全站中文乱码
13         // 指定 request 和 response 的编码
14         request.setCharacterEncoding("utf-8"); // 只对消息体有效
15         response.setContentType("text/html;charset=utf-8");
16         // 在放行时 应该给目标资源一个 request 对象 让目标资源调用
17         // getParameter 时调到我们写的 getParameter
18         // 对 request 进行包装
19         CharacterRequest characterRequest = new CharacterRequest(request);
20         chain.doFilter(characterRequest, response);
21     }
22     public void destroy() {
23     }
24 }
25 // 针对 request 对象进行包装
26 // 继承 默认包装类 HttpServletRequestWrapper
27 class CharacterRequest extends HttpServletRequestWrapper {
28     public CharacterRequest(HttpServletRequest request) {
```

```
29          super(request);
30      }
31      // 子类继承父类一定会覆写一些方法，此处用于重写 getParamter()方法
32      public String getParameter(String name) {
33          // 调用 被包装对象的 getParameter()方法 获得请求参数
34          String value = super.getParameter(name);
35          if (value == null)
36              return null;
37          // 判断请求方式
38          String method = super.getMethod();
39          if ("get".equalsIgnoreCase(method)) {
40              try {
41                  value = new String(value.getBytes("iso-8859-1"), "utf-8");
42              } catch (UnsupportedEncodingException e) {
43                  throw new RuntimeException(e);
44              }
45          }
46          // 解决乱码后返回结果
47          return value;
48      }
49 }
```

在 web.xml 文件中，配置 CharacterFilter 过滤器，由于要拦截用户访问资源的所有请求，因此将<filter-mapping>元素拦截的路径设置为/*，具体代码如下：

```
<filter>
    <filter-name>CharacterFilter</filter-name>
    <filter-class>cn.itcast.chapter04.filter.CharacterFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CharacterFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

（4）访问 form.jsp 页面

重新启动 Web 服务器，打开 IE 浏览器，在地址栏中输入 http://localhost:8080/chapter04/form.jsp 此时，浏览器窗口中会显示一个用户登录的表单，在这个表单中输入用户名传智播客、密码 123456，如图 4-14 所示。

图4-14　运行结果

点击图 4-14 界面中的登录按钮来提交表单，此时，控制台窗口显示的结果如图 4-15 所示。

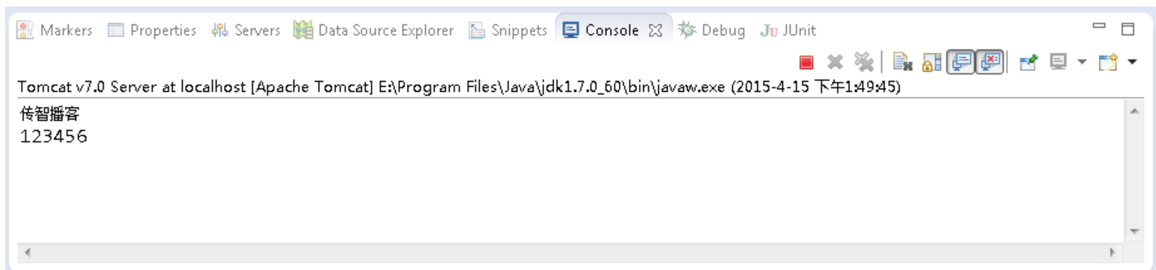

图4-15　控制台窗口

从图 4-15 可以看出，在 form.jsp 表单中输入的用户名和密码都正确的显示在控制台窗口，而且中文的用户名也没有出现乱码问题。需要注意的是，form.jsp 表单的提交方式是 post，因此可以说明使用 post 提交表单可以解决中文乱码问题，表单的提交方式还有一种是 get，接下来就验证 get 方式提交表单的乱码问题能否解决。

（5）使用超连接访问 form.jsp 页面

点击图 4-14 界面中的“点击超链接登录”提交表单，这种提交方式相当于 get 方式提交表单，此时，控制台窗口显示的结果如图 4-16 所示。

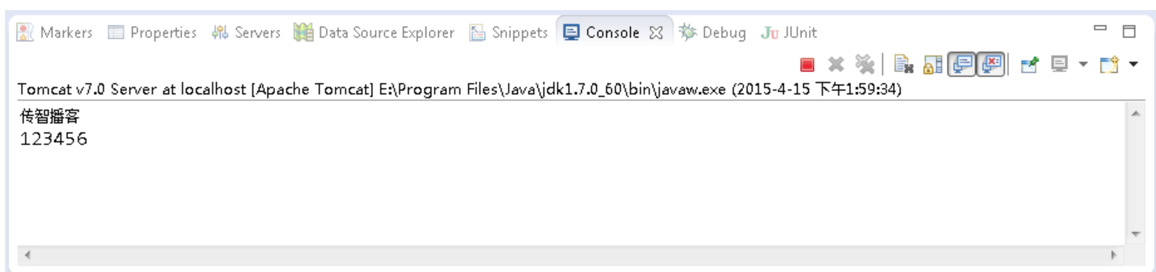

图4-16　控制台窗口

从图 4-16 可以看出，get 方式提交表单与 post 方式提交表单的效果是一样的，同样不会出现乱码问题。因此，可以说明使用 Filter 过滤器可以方便的完成统一全站编码的功能。

需要注意的是，针对 response 对象，Servlet API 也提供了对应的包装类 HttpServletResponseWrapper，它的用法与 HttpServletRequestWrapper 类似，在此不再进行举例介绍了。

4.3.3 Filter 实现页面静态化

在实际开发中，有时为了提高程序性能、减轻数据库访问压力以及对搜索引擎的优化，可以使用 Filter 实现动态页面静态化。页面静态化就是先于用户获取资源或数据库数据进而通过静态化处理，生成静态页

面，所有人都访问这一个静态页面，而静态化处理的页面的访问速度要比动态页面快的多，因此程序性能会有大大的提升。接下来通过一张图来简单描述页面静态化的过程，如图 4-17 所示。

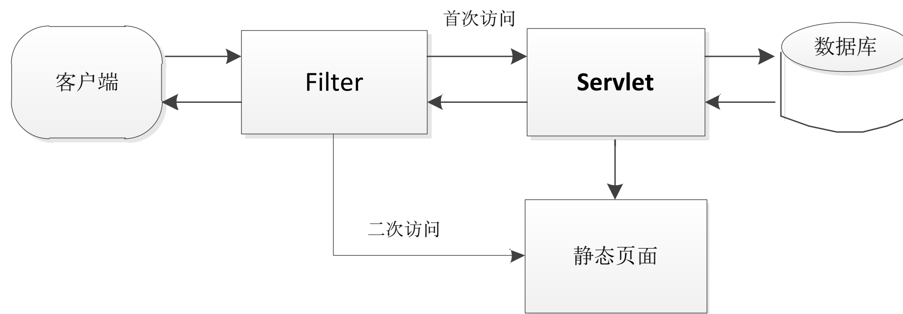

图4-17　页面静态化

图 4-17 中，当客户端首次访问页面时，Filter 会自定义 response 输出缓存 HTML 源码。当客户端第二次访问页面时，就会直接访问静态页面，这样避免访问数据库。

接下来通过一个显示图书分类的案例，分步骤讲解如何使用 Filter 实现页面静态化，具体如下：

（1）　首先写一个完整的查询图书的功能，测试成功后，再通过编写过滤器将动态的图书信息页面静态化。在 MySQL 中创建一个数据库 chapter04，在该数据库中创建数据表 t_book 并插入数据，其中，bname 为图书名称、price 为图书价格、category 为图书分类，具体 SQL 语句如下：

```
CREATE DATABASE chapter04;
USE chapter04;
CREATE TABLE t_book(
  bid CHAR(32) PRIMARY KEY,
  bname VARCHAR(100),
  price NUMERIC(10,2),
  category INT
);

INSERT INTO t_book VALUES('b1', 'JavaSE_1', 10, 1);
INSERT INTO t_book VALUES('b2', 'JavaSE_2', 15, 1);
INSERT INTO t_book VALUES('b3', 'JavaSE_3', 20, 1);
INSERT INTO t_book VALUES('b4', 'JavaSE_4', 25, 1);

INSERT INTO t_book VALUES('b5', 'JavaEE_1', 30, 2);
INSERT INTO t_book VALUES('b6', 'JavaEE_2', 35, 2);
INSERT INTO t_book VALUES('b7', 'JavaEE_3', 40, 2);

INSERT INTO t_book VALUES('b8', 'Java_framework_1', 45, 3);
INSERT INTO t_book VALUES('b9', 'Java_framework_2', 50, 3);
```

查询 t_book 表中的数据，查询结果如下：

```
mysql> SELECT * FROM t_book;
+-----+------------------+-------+----------+
| bid | bname            | price | category |
+-----+------------------+-------+----------+
| b1  | JavaSE_1         | 10.00 |        1 |
| b2  | JavaSE_2         | 15.00 |        1 |
```

```
| b3  | JavaSE_3           | 20.00 |        1 |
| b4  | JavaSE_4           | 25.00 |        1 |
| b5  | JavaEE_1           | 30.00 |        2 |
| b6  | JavaEE_2           | 35.00 |        2 |
| b7  | JavaEE_3           | 40.00 |        2 |
| b8  | Java_framework_1 | 45.00 |        3 |
| b9  | Java_framework_2 | 50.00 |        3 |
+-----+------------------+-------+----------+
9 rows in set (0.04 sec)
```

（2） 将所需的连接数据库包导入到 chapter04 项目中的 lib 文件夹下，创建包 cn.itcast.domain 并在包中编写一个 Book 类，具体代码如例 4-19 所示。

例4-19　Book.java

```
1  package cn.itcast.domain;
2  public class Book {
3      private String bid;
4      private String bname;
5      private double price;
6      private int category;
7      public String getBid() {
8          return bid;
9      }
10     public void setBid(String bid) {
11         this.bid = bid;
12     }
13     public String getBname() {
14         return bname;
15     }
16     public void setBname(String bname) {
17         this.bname = bname;
18     }
19     public double getPrice() {
20         return price;
21     }
22     public void setPrice(double price) {
23         this.price = price;
24     }
25     public int getCategory() {
26         return category;
27     }
28     public void setCategory(int category) {
29         this.category = category;
30     }
31     @Override
32     public String toString() {
```

```
33          return "Book [bid=" + bid + ", bname=" + bname + ", price=" + price
34                  + ", category=" + category + "]";
35      }
36 }
```

（3）　创建包 cn.itcast.dao 并在包中编写 BookDao 类，在本案例中会涉及到两个查询方法，一个是查询所有图书，一个按分类查询图书，，具体代码如例 4-20 所示。

例4-20　BookDao.java

```
1  package cn.itcast.dao;
2  import java.sql.SQLException;
3  import java.util.List;
4  import org.apache.commons.dbutils.QueryRunner;
5  import org.apache.commons.dbutils.handlers.BeanListHandler;
6  import cn.itcast.domain.Book;
7  import cn.itcast.jdbc.utils.JDBCUtils;//这个 JDBCUtils 与 chapter03 中的一样
8  public class BookDao {
9      private QueryRunner qr = new QueryRunner(JDBCUtils.getDataSource());
10     //查询所有
11     public List<Book> findAll() {
12         try {
13             String sql = "select * from t_book";
14             return qr.query(sql, new BeanListHandler<Book>(Book.class));
15         } catch (SQLException e) {
16             throw new RuntimeException(e);
17         }
18     }
19     // 按分类查询
20     public List<Book> findByCategory(int category) {
21         try {
22             String sql = "select * from t_book where category=?";
23             return qr.query(sql, new BeanListHandler<Book>(Book.class), category);
24         } catch (SQLException e) {
25             throw new RuntimeException(e);
26         }
27     }
28 }
```

（4）　在包 cn.itcast.chapter04.serrvlet 下创建一个 BookServlet。该 Servlet 要获取 category 参数，如果这个参数存在，说明是按分类查询，如果不存在，表示查询所有。然后将查询出的数据存成 List 并保存到 request 中转发到页面，具体代码如例 4-21 所示。

例4-21　BookServlet.java

```
1  package cn.itcast.chapter04.servlet;
2  import java.io.IOException;
3  import java.util.List;
4  import javax.servlet.ServletException;
5  import javax.servlet.http.HttpServlet;
```

```
6  import javax.servlet.http.HttpServletRequest;
7  import javax.servlet.http.HttpServletResponse;
8  import cn.itcast.dao.BookDao;
9  import cn.itcast.domain.Book;
10 public class BookServlet extends HttpServlet {
11     public void doGet(HttpServletRequest request, HttpServletResponse response)
12             throws ServletException, IOException {
13         String param = request.getParameter("category");//获取 category 参数
14         BookDao dao = new BookDao();
15         List<Book> bookList = null;
16         // 如果 category 参数不存在，表示查询所有
17         if(param == null || param.trim().isEmpty()) {
18             bookList = dao.findAll();
19         } else {
20             int category = Integer.parseInt(param);    //把参数转换成 int 类型
21             // 按分类查询图书
22             bookList = dao.findByCategory(category);
23         }
24         // 把图书保存到 request 中
25         request.setAttribute("bookList", bookList);
26         request.getRequestDispatcher("/show.jsp").forward(request, response);
27     }
28 }
```

（5）在 WebContent 根目录中，编写 index_book.jsp 页面，用于显示图书分类，具体如例 4-22 所示。

例4-22　index_book.jsp

```
1  <%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
2  <%
3  String path = request.getContextPath();
4  String basePath = request.getScheme()+"://"+request.getServerName()+
5  ":"+request.getServerPort()+path+"/";
6  %>
7  
8  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
9  <html>
10   <head>
11     <base href="<%=basePath%>">
12 
13     <title>My JSP 'index_book.jsp' starting page</title>
14      <meta http-equiv="pragma" content="no-cache">
15      <meta http-equiv="cache-control" content="no-cache">
16      <meta http-equiv="expires" content="0">
17      <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
18      <meta http-equiv="description" content="This is my page">
19   </head>
```

```
20  <body>
21    <a href="<%=request.getContextPath()%>/BookServlet">全部图书</a><br/>
22    <a href="<%=request.getContextPath()%>/BookServlet?category=1">JavaSE 分类</a><br/>
23    <a href="<%=request.getContextPath()%>/BookServlet?category=2">JavaEE 分类</a><br/>
24 <a href="<%=request.getContextPath()%>/BookServlet?category=3">Java 框架分类</a><br/>
25  </body>
26 </html>
```

（6）在 WebContent 根目录中，编写一个 show.jsp 页面，用于显示每类图书信息，具体代码如例 4-23 所示。

例4-23　show.jsp

```
1  <%@page import="cn.itcast.domain.Book"%>
2  <%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
3  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
4  <html>
5  <head>
6  <title>My JSP 'show.jsp' starting page</title>
7  <meta http-equiv="pragma" content="no-cache">
8  <meta http-equiv="cache-control" content="no-cache">
9  <meta http-equiv="expires" content="0">
10 <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
11 <meta http-equiv="description" content="This is my page">
12 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
13 <!--
14     <link rel="stylesheet" type="text/css" href="styles.css">
15     -->
16 </head>
17 <body>
18     <table border="1" align="center" width="50%">
19         <tr>
20             <th>图书名称</th>
21             <th>图书单价</th>
22             <th>图书分类</th>
23         </tr>
24         <%
25             List<Book> list = (List) request.getAttribute("bookList");
26             for (Book b : list) {
27         %>
28         <tr>
29             <td><%=b.getBname()%></td>
30             <td><%=b.getPrice()%></td>
31             <td>
32                 <%  if (b.getCategory() == 1) {%>
33                     <p style="color: red;">JavaSE 分类</p>
34                 <%} else if (b.getCategory() == 2) {%>
```

```
35                      <p style="color: blue;">JavaEE 分类</p>
36                  <%} else {%>
37                      <p style="color: green;">Java 框架分类</p>
38                  <%}%>
39              </td>
40          </tr>
41          <%}%>
42      </table>
43 </body>
44 </html>
```

页面完成后，先验证查看图书分类的功能是否完成，重新启动 Web 服务器，打开 IE 浏览器，在地址栏中输入 http://localhost:8080/chapter04/index_book.jsp，查询结果如图 4-18。

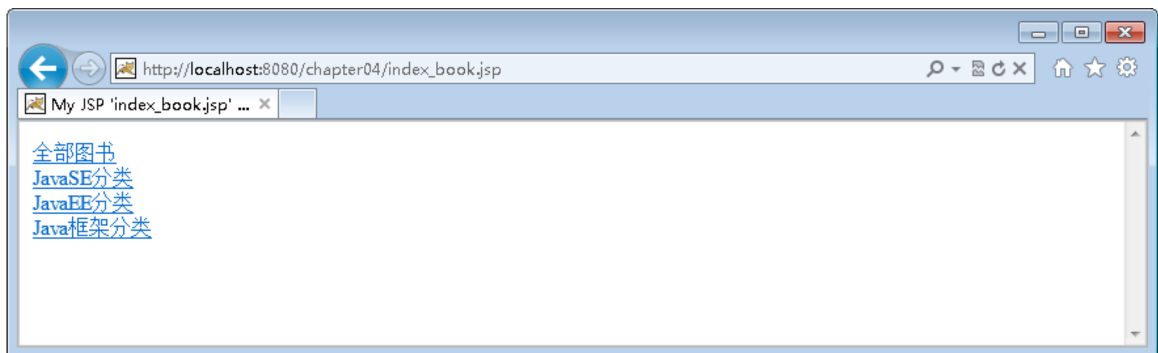

图4-18 查询结果

点击查看 JavaSE 分类，查询结果如图 4-19。

图书名称	图书单价	图书分类
JavaSE_1	10.0	JavaSE分类
JavaSE_2	15.0	JavaSE分类
JavaSE_3	20.0	JavaSE分类
JavaSE_4	25.0	JavaSE分类

图4-19 查询结果

从图 4-19 中的 URL 地址可以看出，本次查询是通过 BookServlet 查询数据库所得结果。接下来，将编写过滤器来实现页面静态化。

（7）在包 cn.itcast.chapter04.filter 下编写 StaticResponse 类，该类为自定义的 Response，重写了 Writer() 方法，使其向指定的文件输出。具体代码如例 4-24 所示。

例4-24 StaticResponse.java

```
1  package cn.itcast.chapter04.filter;
1  import java.io.FileNotFoundException;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.io.UnsupportedEncodingException;
5  import javax.servlet.http.HttpServletResponse;
```

```
6  import javax.servlet.http.HttpServletResponseWrapper;
7  public class StaticResponse extends HttpServletResponseWrapper {
8      private HttpServletResponse response;
9      private PrintWriter pw;
10     // 其中 staticPath 为静态页面的路径
11     public StaticResponse(HttpServletResponse response, String staticPath)
12             throws FileNotFoundException, UnsupportedEncodingException {
13         super(response);
14         this.response = response;
15         // pw 与指定文件绑定在一起，当使用 pw 输出时，就是向指定的文件输出
16         pw = new PrintWriter(staticPath, "utf-8");
17     }
18     // 当 show.jsp 输出页面中的内容时，使用的就是 getWriter()获取的流对象。
19     public PrintWriter getWriter() throws IOException {
20         return pw;
21     }
22     // 关闭方法在过滤器中调用，可以刷新缓冲区。
23     public void close() {
24         pw.close();
25     }
26 }
```

（8）在 cn.itcast.chapter04.filter 包下编写过滤器 StaticFilter 类，该类实现了判断是否存在静态页面，如果存在静态页面，则重定向到静态页面，如果静态页面不存在，那么生成静态页面，其具体代码如例 4-25 所示。

例4-25　StaticFilter.java

```
1  package cn.itcast.chapter04.filter;
2  import java.io.IOException;
3  import java.util.HashMap;
4  import java.util.Map;
5  import javax.servlet.Filter;
6  import javax.servlet.FilterChain;
7  import javax.servlet.FilterConfig;
8  import javax.servlet.ServletContext;
9  import javax.servlet.ServletException;
10 import javax.servlet.ServletRequest;
11 import javax.servlet.ServletResponse;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14 public class StaticFilter implements Filter {
15     private FilterConfig config;
16     public void destroy() {
17     }
18     public void doFilter(ServletRequest request, ServletResponse response,
19             FilterChain chain) throws IOException, ServletException {
```

```
20          // 把 request 和 response 都强转成 http 的
21          HttpServletRequest req = (HttpServletRequest) request;
22          HttpServletResponse res = (HttpServletResponse) response;
23          // 1. 获取 Map
24          // 获取 ServletContext
25          ServletContext sc = config.getServletContext();
26          Map<String, String> staticMap = (Map<String, String>) sc
27                  .getAttribute("static_map");
28          if (staticMap == null) {
29              staticMap = new HashMap<String, String>();
30              sc.setAttribute("static_map", staticMap);
31          }
32          // 2. 通过访问路径获取对应的静态页面
33          // 生成 key: book_前缀，后缀为 category 的值
34          String category = request.getParameter("category");
35          String key = "book_" + category;// 可能有: book_null、book_1、book_2、book_3
36          // 查看这个路径对应的静态页面是否存在
37          if (staticMap.containsKey(key)) {// 如果静态页面已经存在
38              String staticPath = staticMap.get(key);// 获取静态页面路径
39              // 重定向到静态页面
40              res.sendRedirect(req.getContextPath() + "/html/" + staticPath);
41              return;
42          }
43          // 如果静态页面不存在
44          // 3. 生成静态页面
45          // 创建静态页面的路径
46          String staticPath = key + ".html";
47          // 获取真实路径
48          String realPath = sc.getRealPath("/html/" + staticPath);
49          // 创建自定义 Response
50          StaticResponse sr = new StaticResponse(res, realPath);
51          // 放行
52          chain.doFilter(request, sr);
53          // 保存静态页面到 map 中
54          staticMap.put(key, staticPath);
55          // 4. 重定向到静态页面
56          res.sendRedirect(req.getContextPath() + "/html/" + staticPath);
57      }
58      public void init(FilterConfig fConfig) throws ServletException {
59          this.config = fConfig;
60      }
61 }
```

编写完 Filter 后，要在 web.xml 中配置 StaticFilter，配置如下：

```
<filter>
```

```
    <filter-name>StaticFilter</filter-name>
    <filter-class>cn.itcast.chapter04.filter.StaticFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>StaticFilter</filter-name>
    <servlet-name>BookServlet</servlet-name>
  </filter-mapping>
```

需要注意的是，在运行程序前，要在 WebContent 根目录下创建 html 文件夹，以便存放生成的静态页面。重新启动 Web 服务器，打开 IE 浏览器，在地址栏中输入 http://localhost:8080/chapter04/index_book.jsp，查询结果如图 4-20 所示。

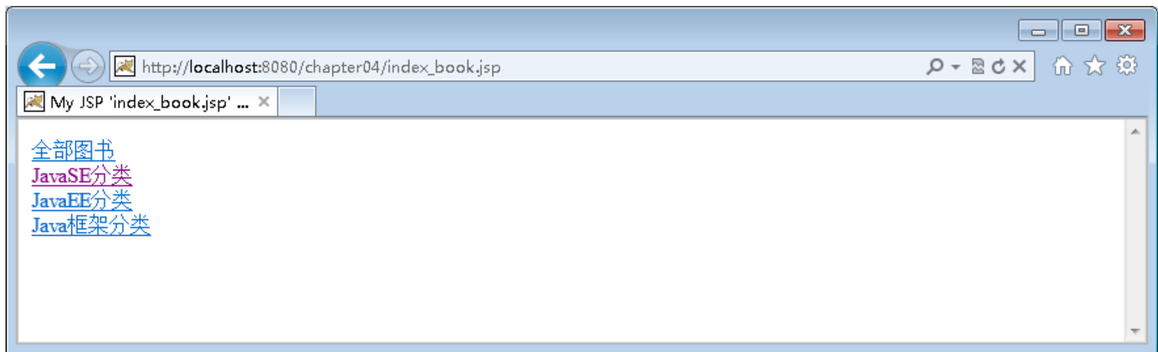

图4-20 查询结果

这时查看 JavaSE 分类，查询结果如图 4-21 所示。

图书名称	图书单价	图书分类
JavaSE_1	10.0	JavaSE分类
JavaSE_2	15.0	JavaSE分类
JavaSE_3	20.0	JavaSE分类
JavaSE_4	25.0	JavaSE分类

图4-21 查询结果

在图 4-21 中，可以看到浏览器的 URL 路径变为访问静态页面了，说明完成了动态页面静态化的功能。需要注意的是，页面静态化虽然可以提高程序的性能，但是却脱离了查询数据库，当数据库发生变化时，用户无法获得最新的数据。这时可以在对数据库进行操作时添加一个标记，告诉程序什么时候需要查询数据库，此功能在这里就不详细讲解了，有兴趣的同学可以自行研究。

4.4 本章小结

本章主要讲解了过滤器的开发与应用，首先讲解了什么是 Filter，然后讲解了如何开发一个 Filter 程序以及开发 Filter 程序时需要注意的细节，最后讲解了 Filter 在实际开发中的具体应用。通过本章的学习，读者能够掌握 Filter 的概念以及创建和部署的过程，并且能够掌握 Filter 的具体应用。