第5章 过滤器和监听器

过滤器和监听器是两种特殊的 Servlet。过滤器可以对用户的请求和响应进行过滤，常被用于权限检查和参数编码的统一设置。监听器可以用来对 Web 应用程序进行监听和控制，增强 Web 应用程序的事件处理能力。


视频讲解

5.1 过 滤 器

过滤器(Filter)是在服务器上运行的，位于被请求资源与客户端之间的起过滤功能的程序。它能够对请求和响应进行检查和修改，通常在接收到请求后执行一些通用操作，如过滤敏感词、统一字符编码和实施安全控制等。其工作原理如图 5-1 所示。

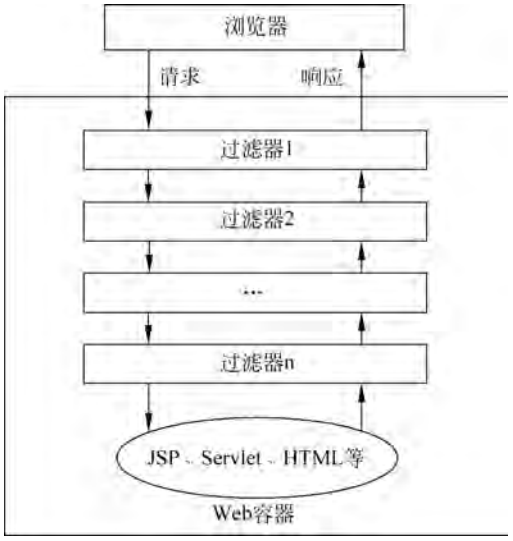

图 5-1 过滤器的工作原理

如图 5-1 所示，客户端向服务器资源发送请求。请求在到达对应资源之前，先由过滤器过滤。过滤器会检查用户请求是否符合过滤规则，如果符合过滤规则，则请求依次通过过滤器链中的过滤器，最终到达请求的资源。同理，资源做出的响应也会依次通过过滤器链最终到达客户端。过滤器根据过滤规则，可以做出如下动作：

(1) 将请求发送给对应的资源(如 JSP、Servlet)。

(2) 用修改后的请求调用相应的资源。

(3) 将请求发送给被请求的资源，修改响应，再将响应发送到客户端。

(4) 禁止调用被请求的资源,将请求重定向到其他资源。

5.1.1 过滤器编程接口

jakarta.servlet 包提供了与过滤器相关的一组接口和类,分别是 Filter 接口、FilterChain 接口、FilterConfig 接口、FilterRegistration 接口、GenericFilter 类和 HttpFilter 类,如表 5-1 所示。

表 5-1 过滤器相关的接口和类

接口、类	说 明
public interface Filter	编写过滤器要实现的接口
public abstract class GenericFilter	Filter 接口的实现类。该类定义了一个通用的、独立于协议的过滤器。要编写在 Web 上使用的 HTTP 过滤器,需继承 HttpFilter 类
public abstract class HttpFilter extends GenericFilter	Filter 接口的实现类,是 GenericFilter 的子类,用于创建适用于 Web 的 HTTP 过滤器
public interface FilterConfig	供 Web 容器在初始化时给过滤器传递信息用
public interface FilterChain	过滤器链(FilterChain)是 Servlet 容器提供给开发人员的一个对象。过滤器使用 FilterChain 对象调用链中的下一个过滤器,如果当前过滤器是链中的最后一个过滤器,则使用 FilterChain 对象调用链末端的资源
public interface FilterRegistration extends Registration	支持过滤器进行深入配置的接口

1. Filter 接口

Filter 接口是编写过滤器需要实现的接口,该接口定义了 init()方法,doFilter()方法和 destroy()方法,具体如表 5-2 所示。

表 5-2 Filter 接口中的方法

方 法 声 明	说 明
void init(FilterConfig filterConfig)	过滤器的初始化方法,Servlet 容器创建过滤器实例后将调用该方法
void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)	用于实现过滤操作。当客户请求满足过滤规则时,Servlet 容器将调用过滤器的 doFilter()方法完成过滤。其中参数 request 和 response 分别代表 Web 服务器或过滤器链中的上一个过滤器传递来的请求和响应,参数 chain 代表当前的过滤器链对象
void destroy()	用于释放被过滤器占用的资源,如关闭 IO 流,关闭数据库等。该方法在 Servlet 容器释放过滤器对象前调用

2. FilterConfig 接口

FilterConfig 接口用于封装过滤器的配置信息。在过滤器初始化时,Servlet 容器将 FilterConfig 对象作为参数传递给过滤器对象的 init()方法完成初始化。FilterConfig 接口中的方法如表 5-3 所示。

表 5-3　FilterConfig 接口中的方法

方 法 声 明	说　　明
String getFilterName()	返回过滤器的名称
ServletContext getServletContext()	返回过滤器对象运行环境中的 ServletContext 对象
String getInitParameter(String name)	返回名为 name 的初始化参数的值
Enumeration < String > getInitParameterNames()	返回过滤器的所有初始化参数名称的枚举

3. FilterChain 接口

FilterChain 接口定义了一个 doFilter()方法，其原型如下：

```
void doFilter ( ServletRequest request, ServletResponse response ) throws IOException,
ServletException
```

这个方法用于调用过滤器链中的下一个过滤器，如果当前过滤器是链上最后一个过滤器，则将请求提交给处理程序或者将响应发送给客户端。

4. HttpFilter 类

一个抽象类，它的子类可以用来创建适用于 Web 站点的 HTTP 过滤器。用于创建过滤器的子类需要覆盖 doFilter()方法。其原型如下：

```
doFilter(jakarta.servlet.http.HttpServletRequest,
jakarta.servlet.http.HttpServletResponse, jakarta.servlet.FilterChain)
```

5.1.2　过滤器生命周期

过滤器生命周期指一个过滤器对象从创建到执行过滤再到销毁的过程。过滤器生命周期可分为创建、过滤、销毁 3 个阶段。表 5-2 的 Filter 接口中的方法就是过滤器生命周期的 3 个方法。

1. 创建阶段

Web 容器启动的时候就会创建过滤器对象，并调用 init()方法，完成对象的初始化。需要注意的是，在一次完整的请求/响应过程中，过滤器对象只会被创建一次，即 init()方法只会被调用一次。

2. 过滤阶段

当客户端向目标资源发出请求时，Web 容器会筛选出符合映射条件的过滤器，并按照类名的先后顺序依次执行过滤器对象的 doFilter()方法。在执行过滤的 doFilter()方法中可以调用 Servlet API 实现对请求、响应的预处理。一般来讲，对请求执行过滤后，还要将其送给过滤器链上的下一个资源。因此，过滤器对象的 doFilter()方法的最后一条语句一般是调用 FilterChain 接口的 doFilter()方法。

3. 销毁阶段

服务器关闭时，Web 容器调用 destroy()方法销毁过滤器对象。

5.1.3　设计过滤器

对于自定义过滤器，有两种实现方案。第一，创建一个类实现 Filter(jakarta.servlet.

Filter)接口。第二,创建一个类继承 HttpFilter(jakarta.servlet.http.HttpFilter)类。对于第一种方案,可以按需重写过滤器生命周期中的 3 个方法,并且请求和响应的类型不局限于 HTTP 请求和 HTTP 响应。对于第二种方案,只需覆盖父类的 doFilter()方法,过滤的请求和响应类型仅限于 HTTP 请求和 HTTP 响应。

【例 5-1】 设计一个简单的过滤器,在控制台输出过滤器的创建信息和请求过滤信息。创建一个名为 chapter5 的动态 Web 项目。在 com.example.filter 包中创建一个过滤器和一个 Servlet,代码如文件 5-1 和文件 5-2 所示。

【文件 5-1】 MyFilter.java

```
1  package com.example.filter;
2
3  //import 部分略
4
5  @WebFilter("/*")
6  public class MyFilter extends HttpFilter {
7
8      public void destroy() {
9          System.out.println("filter destroyed");
10     }
11
12     public void doFilter(ServletRequest request,
13         ServletResponse response, FilterChain chain)
14         throws IOException, ServletException {
15         System.out.println("this is filter");
16         //pass the request along the filter chain
17         chain.doFilter(request, response);
18     }
19
20     public void init(FilterConfig fConfig)
21         throws ServletException {
22         System.out.println("filter initialization");
23     }
24 }
```

【文件 5-2】 MyServlet.java

```
1  package com.example.filter;
2
3  //import 部分此处略
4
5  @WebServlet("/MyServlet")
6  public class MyServlet extends HttpServlet {
7
8      protected void doGet(HttpServletRequest request,
9          HttpServletResponse response)
10         throws ServletException, IOException {
11         PrintWriter out = response.getWriter();
12         System.out.println("this is servlet");
13         out.print("servlet response");
14     }
```

```
15        //此处省略了 doPost()方法
16    }
```

在文件 5-1 中配置了过滤器生命周期的 3 个方法。为便于观察过滤器的创建和销毁过程，分别在 init()方法和 destroy()方法中增加控制台输出，如第 22 行和第 9 行所示。在执行过滤的 doFilter()方法中，首先向控制台输出字符串“this is filter”(第 15 行)，再将请求传递给过滤器链的下一个资源(第 17 行)。本例中的下一个资源为 MyServlet。MyServlet 在处理请求时首先在控制台输出字符串，再用 out 对象生成响应。在浏览器的地址栏输入“http://localhost:8080/chapter5/MyServlet”后，向 MyServlet 发出请求，可以在浏览器页面看到由 Servlet 生成的响应 servlet response ，如图 5-2 所示。

图 5-2　Servlet 生成的响应

结合过滤器的工作原理可知，在请求到达目标资源 MyServlet 前，由 MyFilter 过滤器执行过滤。执行过滤后，MyFilter 再将请求送给 MyServlet，由 MyServlet 生成客户端响应。可以通过控制台输出了解过滤器从创建到执行过滤的过程。将项目部署到 Tomcat 服务器上。在 Tomcat 启动后，可以看到控制台输出“filter initialization”，如图 5-3 所示，这说明过滤器随着 Tomcat 启动即完成了创建和初始化。随后，通过浏览器给 MyServlet 发送请求，可以看到控制台的输出为“this is filter”和“this is servlet”两条消息。说明请求首先被过滤器过滤，再由过滤器转给 Servlet 处理。

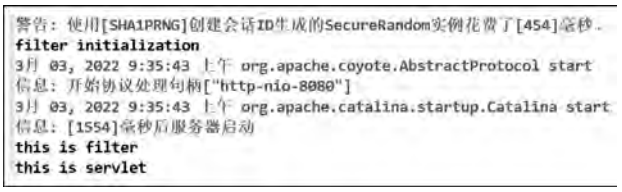

图 5-3　控制台输出

创建过滤器后，还需要进行配置。与 Servlet 的配置类似，过滤器也有两种配置方式：部署描述符和注解。

1. 部署描述符

可以在 web.xml 文件中配置过滤器的基本信息。用于配置过滤器的两个标记分别为<filter/>和<filter-mapping/>。对于例 5-1 中的过滤器 MyFilter 可以进行如下配置。

```
1  <filter>
2      <filter-name>MyFilter</filter-name>
3      <filter-class>com.example.filter.MyFilter</filter-class>
4  </filter>
5  <filter-mapping>
6      <filter-name>MyFilter</filter-name>
7      <url-pattern>/*</url-pattern>
8  </filter-mapping>
```

其中,以下内容需要注意:

(1) <filter>标记用于注册一个过滤器。

(2) <filter-name>标记用于设置 Filter 类的名称。

(3) <filter-class>标记用于设置 Filter 类的全限定名。

(4) <filter-mapping>标记用于设置一个过滤器所过滤的资源。

(5) <url-pattern>标记用于匹配用户请求的 URL,"/*"表示当前项目下的所有资源。即给当前项目根目录下的任何资源发出的请求都会被过滤器过滤。本例中的"/*"也可以替换为 Servlet 的 url-pattern,即"/MyServlet"。

2. 注解

与配置 Servlet 类似,在创建过滤器后,可以使用标注在过滤器类上的@WebFilter 注解来告知 Servlet 容器有哪些过滤器需要被实例化。例 5-1 中,"@WebFilter("/*")"用于匹配用户请求的 URL,"/*"的意思是对项目根目录下任何资源发出的请求都会被过滤器过滤,也可以用表 5-4 中的 urlPatterns 属性指定,如"@WebFilter(urlPatterns={"/*"})"。@WebFilter 注解的常用属性如表 5-4 所示。

表 5-4 @WebFilter 的属性

属性名	类型	说明
filterName	String	用于指定过滤器的名称,默认值是过滤器类的名字
urlPatterns	String[]	指定一组过滤器的 URL 匹配模式
value	String[]	等价于 urlPatterns 属性,与 urlPatterns 属性不能同时使用
servletNames	String[]	指定过滤器应用于哪些 Servlet。其值是@WebServlet 注解的 name 属性的取值
dispatcherTypes	DispatcherType	指定过滤器的转发模式。具体取值包括 ERROR、FORWARD、INCLUDE、REQUEST
initParams	WebInitParam[]	指定过滤器的一组初始化参数

表 5-4 中,@WebFilter 注解有一个 dispatcherTypes 属性,它可以指定过滤器的转发模式。dispatcherTypes 属性有 4 个常用值。

1) DispatcherType.REQUEST

这个值是 dispatcherTyps 属性的默认值。当过滤器设置 dispatcherTypes 属性值为 DispatcherType.REQUEST 时,过滤器拦截直接 URL 访问资源的请求与响应,包括:在地址栏中直接访问、表单提交、超链接、重定向,对于以其他方式发送的请求与响应不予拦截。

2) DispatcherType.FORWARD

当过滤器设置 dispatcherTypes 属性值为 DispatcherType.FORWARD 时,过滤器拦截以请求转发方式发送的请求,其他的请求不予拦截。

3) DispatcherType.INCLUDE

当过滤器设置 dispatcherTypes 属性值为 DispatcherType.INCLUDE 时,如果用户通过 RequestDispatcher 对象的 include()方法发送请求则被过滤器拦截,对于其他的请求与响应过滤器不予拦截。

4) DispatcherType.ERROR

当过滤器设置 dispatcherTypes 属性值为 DispatcherType.ERROR 时,如果通过声明

式异常处理机制发送请求，则被过滤器拦截，其他的请求与响应过滤器不予拦截。

下面，通过几个例子来进一步说明这几个属性值的作用。

【例 5-2】 创建一个名为 DispatcherTypeFilter 的过滤器，针对特定类型的请求进行过滤(拦截)。过滤器代码如文件 5-3 所示。

【文件 5-3】 DispatcherTypeFilter.java

```
1  package com.example.filter;
2  //import 部分略
3  @WebFilter(urlPatterns = "/filter.jsp",
4  dispatcherTypes = {DispatcherType.ERROR})
5  public class DispatcherTypeFilter extends HttpFilter {
6
7      public void doFilter(ServletRequest request,
8          ServletResponse response, FilterChain chain)
9          throws IOException, ServletException {
10         System.out.print("this is dispatcher filter");
11         // pass the request along the filter chain
12         chain.doFilter(request, response);
13     }
14 }
```

如文件 5-3 所示，第 3～4 行将@WebFilter 注解的 dispatcherTypes 属性设置为 DispatcherType.ERROR。这意味着如果服务器端程序在处理请求过程中抛出异常，则过滤器会被调用。否则，过滤器不会被调用。为验证这个设置能否起作用，需要再创建一个 Servlet 和一个 JSP。

创建一个名为 DispatcherTypeServlet 的 Servlet，并设定该 Servlet 在处理请求过程中会抛出运行时异常。DispatcherTypeServlet 的实现代码如文件 5-4 所示。

【文件 5-4】 DispatcherTypeServlet.java

```
1  package com.example.filter;
2  //import 部分略
3  @WebServlet("/DispatcherTypeServlet")
4  public class DispatcherTypeServlet extends HttpServlet {
5      protected void doGet(HttpServletRequest request,
6          HttpServletResponse response)
7          throws ServletException, IOException {
8          if(true)
9              throw new RuntimeException("hello,exception");
10     }
11     //此处省略了 doPost()方法
12 }
```

创建一个名为 filter 的 JSP，用于输出响应内容。filter.jsp 的代码如文件 5-5 所示。

【文件 5-5】 filter.jsp

```
1  <%@ page contentType = "text/html; charset = UTF - 8" %>
2  <!DOCTYPE html>
3  <html>
```

```
4  <body>
5      This is jsp file.
6  </body>
7  </html>
```


此外,需要在 WEB-INF 文件夹下配置部署描述文件 web.xml,内容如文件 5-6 所示。

【文件 5-6】 web.xml

```
1  <display-name>Chapter5</display-name>
2  <error-page>
3      <error-code>500</error-code>
4      <location>/filter.jsp</location>
5  </error-page>
6  <welcome-file-list>
7      <welcome-file>index.html</welcome-file>
8  </welcome-file-list>
```

在浏览器的地址栏输入“http://localhost:8080/chapter5/DispatcherTypeServlet”给本例中的 Servlet 发请求,Servlet 收到请求后会抛出一个运行时异常(文件 5-4 的第 8～9 行),而根据部署描述文件 web.xml 的设置,当服务器端抛出运行时异常,即出现 500 错误(Internal Server Error)时,Servlet 容器会将请求转发给 filter.jsp(文件 5-6 的第 2～5 行)。而过滤器 DispatcherTypeFilter 的转发模式为 DispatcherType.ERROR(文件 5-3 的第 3～4 行),这个请求就会被过滤器过滤,进而产生相应的输出。浏览器端的输出如图 5-4 所示,同时在控制台可见过滤器的输出,如图 5-5 所示。

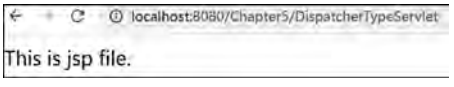

图 5-4 浏览器端的输出

```
        at org.apache.tomcat.util.net.SocketProcessorBase.run
(SocketProcessorBase.java:49)
        at org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1191)
        at org.apache.tomcat.util.threads.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:659)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.ru
(TaskThread.java:61)
        at java.base/java.lang.Thread.run(Thread.java:834)

this is dispatcher filter
```

图 5-5 控制台的输出

如果通过浏览器的地址栏直接给 filter.jsp 发送请求,由于 filter.jsp 代码在运行中不会抛出异常,即这个请求不符合过滤器的过滤规则,因此过滤器不会被调用。这样,只能在浏览器页面上看到如图 5-4 的 JSP 的输出。读者不妨分析下,如果将文件 5-4 第 8 行的条件改为永假条件,会有什么样的输出? 此外,如果将文件 5-4 的 Servlet 中的代码改为将请求转发给 JSP,即:

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
```

```
        request.getRequestDispatcher("filter.jsp")
        .forward(request, response);
    }
```

如将过滤器的转发模式设置为 DispatcherType.FORWARD。再次给 Servlet 发请求，得到的结果同图 5-4 和图 5-5。读者可自行分析其原理。

5.1.4 过滤器应用案例

【例 5-3】 创建一个名为 IPFilter 的过滤器，实现禁止地址为 127.0.0.1 的主机访问 filter.jsp 的功能。

分析：根据过滤器的特性，某主机给本系统资源发送的请求首先到达过滤器。此时，可以通过过滤器获取远程主机的 IP 地址，与系统中设定的 IP 地址黑名单比对。如果该主机的 IP 地址在黑名单中，则拦截该主机的请求；否则将请求传递给过滤器链的下一个资源。IPFilter 的代码如文件 5-7 所示。

【文件 5-7】 IPFilter.java

```
1  package com.example.filter;
2  //import 部分此处略
3
4  @WebFilter(urlPatterns = "/filter.jsp",initParams = {@WebInitParam(
5  name = "forbid",value = "127.0.0.1")})
6  public class IPFilter extends HttpFilter {
7      private String address;
8
9      public void doFilter(ServletRequest request,
10         ServletResponse response, FilterChain chain)
11         throws IOException, ServletException {
12         PrintWriter out = response.getWriter();
13         if(address.equals(request.getRemoteAddr()))
14             out.print("Sorry, you are forbidden to visit");
15       else
16             chain.doFilter(request, response);
17     }
18     public void init(FilterConfig config) {
19         address = config.getInitParameter("forbid");
20     }
21 }
```

如文件 5-7 所示，过滤器 IPFilter 利用@WebFilter 标签的 initParams 属性设置了过滤器的初始化参数 forbid，并将其初始值设定为 127.0.0.1。随后，利用过滤器的 init()方法读取初始化参数。启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://127.0.0.1:8080/chapter5/filter.jsp，可见利用本机地址访问 filter.jsp 时，请求被过滤器拦截，浏览器显示结果如图 5-6 所示。

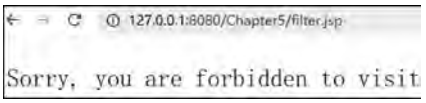

图 5-6 禁止特定 IP 访问系统的浏览器显示结果

【例 5-4】 防止中文乱码。

在填写表单数据时,难免会需要输入中文,如姓名、公司名称等。可以利用过滤器的特性,在表单提交的请求到达 RequestParamServlet 前,对请求中的字符设置编码规则。并在 RequestParamServlet 响应到达客户端前设置响应消息中的字符编码规则。

创建一个名为 CharFilter 的过滤器,用于解决例 3-5 中填写表单数据时遇到的中文乱码问题,其中请求和响应消息的字符编码均设置为 UTF-8。CharFilter 的代码如文件 5-8 所示。

【文件 5-8】 CharFilter.java

```
1  import java.io.IOException;
2
3  //import 部分此处略
4
5  @WebFilter("/RequestParamServlet")
6  public class CharFilter extends HttpFilter {
7
8    public void doFilter(jakarta.servlet.ServletRequest request,
9  jakarta.servlet.ServletResponse response, FilterChain chain)
10 throws IOException, ServletException {
11      request.setCharacterEncoding("UTF-8");
12      response.setCharacterEncoding("UTF-8");
13      chain.doFilter(request, response);
14   }
15 }
```

5.2 监 听 器

5.2.1 监听器概述

在 Web 应用程序设计中,经常需要对某些事件进行监听,以便及时做出处理。对于桌面应用程序而言,鼠标单击或双击、键盘上的键被按下等都是事件。类似地,对于 Web 应用程序来说,session 对象的创建、请求域中某个属性的移除等都是事件。为此,Servlet 规范提供了监听器(Listener),专门用于监听 Servlet 事件。监听器技术涉及几个重要的概念,分别如下。

(1) 事件:对于 Web 应用程序而言,ServletContext 对象、HttpSession 对象和 ServletRequest 对象的状态改变可称为 Servlet 事件。如 HttpSession 对象的创建,ServletRequest 对象中属性的增加或移除都是事件。

(2) 监听器:负责监听事件是否发生。它是一个实现了一个或多个 Servlet 事件监听接口的类。它在 Web 应用程序部署时被注册到 Web 容器中并被实例化。

(3) 事件处理器:监听器的方法。当事件发生的时候,监听器会监听到事件的发生,并触发相应的处理器用以处理事件。

5.2.2 监听器编程接口

Web 应用程序中的监听器就是一段实现了特定接口的 Java 程序,专门用于监听 Web 应用程序中 ServletContext、HttpSession 和 ServletRequest 等域对象的状态变化,包括这些对象的创建、销毁及域对象属性的修改。

根据监听对象的不同，监听器可以划分为以下 3 种。

(1) ServletContext 监听器：用于监听 ServletContext 对象。

(2) HttpSession 监听器：用于监听 HttpSession 对象。

(3) ServletRequest 监听器：用于监听 ServletRequest 对象。

这 3 种监听器共包含 9 个监听器接口和 7 个监听事件类，如表 5-5 所示。

表 5-5　监听器接口与事件类

监听对象	监听器接口	监听事件类	说　明
ServletContext	ServletContextListener	ServletContextEvent	用于监听 ServletContext 对象的创建和销毁
	ServletContextAttribute-Listener	ServletContextAttribute-Event	用于监听 ServletContext 对象中属性的变更
HttpSession	HttpSessionListener	HttpSessionEvent	用于监听 HttpSession 对象的创建和销毁
	HttpSessionActivation-Listener		用于监听 HttpSession 对象被钝化和激活的事件
	HttpSessionAttribute-Listener	HttpSessionBinding-Event	用于监听 HttpSession 对象中属性的变更
	HttpSessionBinding-Listener		用于监听 HttpSession 对象中 JavaBean 对象的绑定和解绑事件
	HttpSessionId Listener	HttpSessionEvent	用于监听 HttpSession 对象的 id 的变更
ServletRequest	ServletRequestListener	ServletRequestEvent	用于监听 ServletRequest 对象的创建和销毁
	ServletRequestAttribute-Listener	SevletRequestAttribute-Event	用于监听 ServletRequest 对象中属性的变更

1. 监听 ServletContext 对象

用于监听 ServletContext 对象的接口有两个。其中，ServletContextListener 接口用于监听 ServletContext 对象的创建和销毁，ServletContextAttributeListener 用于监听 ServletContext 对象中属性的变化(添加、移除)。常用的监听方法如表 5-6 所示。

表 5-6　SevletContext 监听器的接口和常用方法

接口名称	接口方法	事　件
ServletContextListener	void contextInitialized(ServletContext-Event sce)	创建 ServletContext 对象
	void contextDestroyed(ServletContext-Event sce)	销毁 ServletContext 对象
ServletContextAttributeListener	void attributeAdded(ServletContext-AttributeEvent event)	添加属性
	void attributeRemoved(ServletContext-AttributeEvent event)	删除属性
	void attributeReplaced(ServletContext-AttributeEvent event)	修改属性

2. 监听 HttpSession 对象

用于监听 HttpSession 对象的接口有 5 个。这些接口对应的常用方法如表 5-7 所示。

表 5-7 HttpSession 监听器的接口和常用方法

接口名称	接口方法	事件
HttpSessionListener	void sessionCreated(HttpSessionEvent se)	创建 HttpSession 对象
	void sessionDestroyed(HttpSessionEvent se)	销毁 HttpSession 对象
HttpSessionActivation-Listener	void sessionWillPassivate(HttpSessionEvent se)	HttpSession 对象被激活
	void sessionDidActivate(HttpSessionEvent se)	HttpSession 对象被钝化
HttpSessionAttribute-Listener	void attributeAdded(HttpSessionBindingEvent event)	增加属性
	void attributeRemoved(HttpSessionBindingEvent event)	移除属性
	void attributeReplaced(HttpSessionBindingEvent event)	修改属性
HttpSessionBinding-Listener	void valueBound(HttpSessionBindingEvent event)	HttpSession 对象绑定
	void valueUnbound(HttpSessionBindingEvent event)	HttpSession 对象解绑
HttpSessionIdListener	void sessionIdChanged(HttpSessionEvent event, String oldSessionId)	session-id 变化

说明：激活(Activate)和钝化(Passivate)是 Web 容器为了更好地利用系统资源或者进行服务器负载均衡而对特定对象采取的措施。会话对象的钝化指的是暂时将会话对象通过序列化的方法存储到硬盘上,而激活与钝化相反,指的是把硬盘上存储的会话对象重新加载到 Web 容器中。

3. 监听 ServletRequest 对象

用于监听 ServletRequest 对象的接口有两个,这些接口对应的常用方法如表 5-8 所示。

表 5-8 ServletRequest 监听器的接口和常用方法

接口名称	接口方法	事件
ServletRequest-Listener	void requestInitialized(ServletRequestEvent sre)	创建 ServletRequest 对象
	void requestDestroyed(ServletRequestEvent sre)	销毁 ServletRequest 对象
ServletRequest-AttributeListener	void attributeAdded(ServletRequestAttributeEvent srae)	添加属性
	void attributeRemoved(ServletRequestAttributeEvent srae)	移除属性
	void attributeReplaced(ServletRequestAttributeEvent srae)	修改属性

5.2.3 监听器应用案例

设计一个监听器一般需要如下步骤。

(1) 实现合适的接口：监听器需要根据监听对象的不同,实现表 5-5 中的某个监听接口。

(2) 设计事件处理器：根据所选择的监听器接口,实现该接口中的相关方法。

(3) 配置监听器：既可以在部署描述文件 web.xml 文件中配置,又可以利用注解@WebListener 完成监听器配置。

(4) 提供任何需要的初始化参数。

【例 5-5】 设计监听器,监听 ServletContext、HttpSession 和 ServletRequest 域对象的生命周期事件。

为实现这个目标,就要设计监听器类来实现针对这些域对象的监听器接口。可以设计一个类,来实现 3 个接口,从而使这个类具有针对 3 个域对象的事件监听的功能。监听器 MyListener 代码如文件 5-9 所示。

【文件 5-9】 MyListener.java

```
1  package com.example.listener;
2  //import 部分此处略
3
4  @WebListener
5  public class MyListener implements ServletContextListener,
6    HttpSessionListener, ServletRequestListener {
7      public void contextInitialized(ServletContextEvent sce) {
8          System.out.println("ServletContext 对象被创建");
9      }
10     public void contextDestroyed(ServletContextEvent sce) {
11         System.out.println("ServletContext 对象被销毁");
12     }
13     public void sessionCreated(HttpSessionEvent se) {
14         System.out.println("HttpSession 对象被创建");
15     }
16     public void sessionDestroyed(HttpSessionEvent se) {
17         System.out.println("HttpSession 对象被销毁");
18     }
19     public void requestInitialized(ServletRequestEvent sre) {
20         System.out.println("ServletRequest 对象被创建");
21     }
22     public void requestDestroyed(ServletRequestEvent sre) {
23         System.out.println("ServletRequest 对象被销毁");
24     }
25 }
```

文件 5-9 中,@WebListener 的作用是配置监听器(第 4 行)。ServletContext 对象与当前的 Web 应用程序对应。因为已在 Tomcat 服务器上部署了 chapter5 项目,当服务器启动时,Tomcat 服务器会自动加载 chapter5 项目,并创建与其对应的 ServletContext 对象。由于 chapter5 项目中配置了 MyListener 监听器,并且该监听器实现了 ServletContextListener 接口,当 Tomcat 创建 ServletContext 对象时就会调用 MyListener 类的 contextInitialized()方法作为事件处理器,输出“ServletContext 对象被创建”这行信息(第 7～9 行)。要观察 ServletContext 对象的销毁信息,可以将 Tomcat 服务器关闭。Tomcat 服务器关闭前,会销毁 ServletContext 对象,同时 contextDestroyed()方法被调用(第 10～12 行),在控制台可见 ServletContext 对象被销毁的信息,如图 5-7 所示。

为了查看 HttpSessionListener 和 ServletRequestListener 的运行效果,可以在 chapter5 项目中编写一个名为 listener.jsp 的文件,内容如文件 5-10 所示。

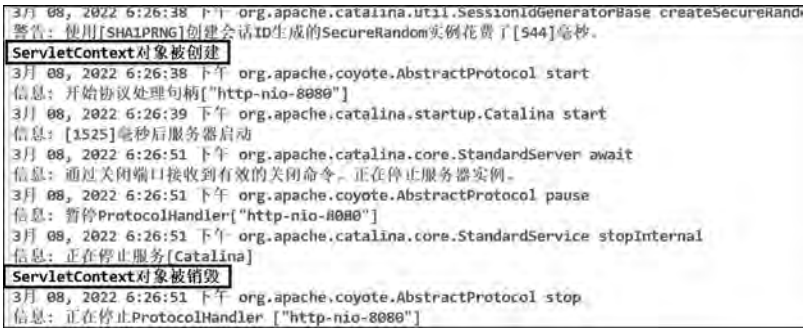

图 5-7　ServletContext 对象的创建和销毁

【文件 5-10】　listener. jsp

```
1  <html>
2  <body>
3        This is listener
4  </body>
5  </html>
```

为了尽快看到 HttpSession 对象的创建和销毁过程,可以在项目的 web. xml 文件中设置 session 的超时时间为 1min,内容如下:

```
<session-config>
    <session-timeout>1</session-timeout>
</session-config>
```

启动 Tomcat 服务器,在浏览器的地址栏输入“http://localhost:8080/chapter5/listener. jsp”,观察控制台输出。当浏览器第一次访问项目中的动态资源(JSP、Servlet)时,会创建 HttpSession 对象,Tomcat 服务器会调用监听器的 sessionCreated()方法作为创建 HttpSession 对象事件的事件处理器(第 13～15 行)。并且,由于浏览器发送请求,会自动创建 HttpServletRequest 对象,Tomcat 服务器会调用 requestInitialized()方法作为创建请求对象事件的处理器(第 19～21 行)。当 listener. jsp 对该请求做出响应后,请求随即被销毁,控制台就会输出“ServletRequest 对象被销毁”的消息(第 22～24 行),如图 5-8 所示。

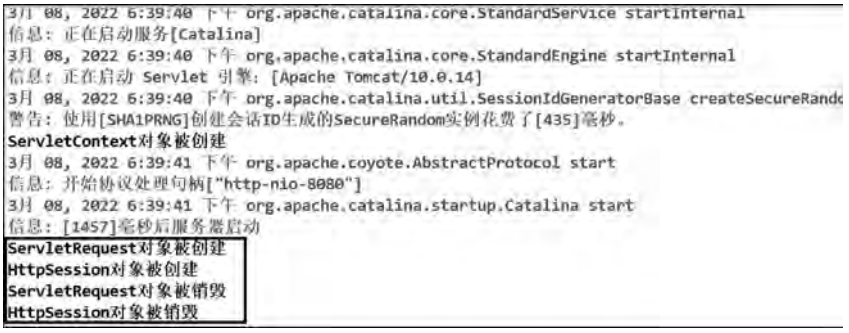

图 5-8　控制台输出

如果关闭访问 listener. jsp 文件的浏览器页面或保持浏览器不刷新。与之对应的 HttpSession 对象会在 1min 后被销毁。控制台显示结果如图 5-8 所示。