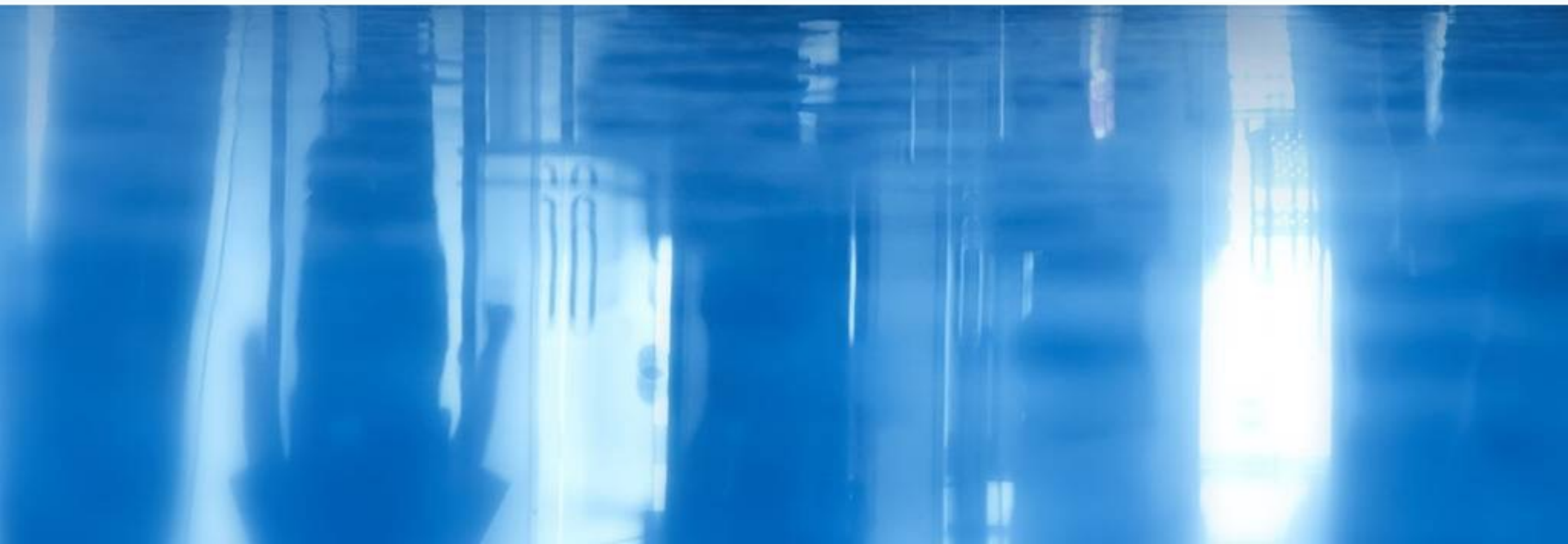


第 8 章

Servlet技术模型



本章内容

- 8.1 Web监听器
- 8.2 Web过滤器
- 8.3 Servlet的多线程问题
- 8.4 Servlet的异步处理

本章内容

- Web应用程序运行过程中可能发生各种事件，如ServletContext事件、会话事件及请求有关的事件等，Web容器采用监听器模型处理这些事件。
- 过滤器用于拦截传入的请求或传出的响应，并监视、修改或以某种方式处理这些通过的数据流。
- 本章主要介绍这两个高级应用，它们是Web事件处理模型和Servlet过滤器。此外，本章还将讨论Servlet多线程问题以及Servlet 3.0的异步处理问题。

8.1 Web监听器

- Web应用程序中的事件主要发生在三个对象上：**ServletContext**、**HttpSession**和**ServletRequest**对象。
- 事件的类型主要包括对象的生命周期事件和属性改变事件。
- 例如，对于**ServletContext**对象，当它初始化和销毁时会发生**ServletContextEvent**事件，当在该对象上添加属性、删除属性或替换属性时会发生**ServletContextAttributeEvent**事件。
- 对于会话对象和请求对象也有类似的事件。为了处理这些事件，**Servlet**容器采用了监听器模型，即需要实现有关的监听器接口。

8.1 Web监听器

- 在Servlet 3.0 API中定义了7个事件类和9个监听器接口，根据监听器所监听事件的类型和范围，可以把它们分为三类：
 - ServletContext事件监听器
 - HttpSession事件监听器
 - ServletRequest事件监听器

8.1.1 监听ServletContext事件

- 在ServletContext对象上可能发生两种事件，对这些事件可使用两个事件监听器接口处理，如表8-1所示。。

表 8-1 ServletContext 事件类与监听器接口

监听对象	事件	监听器接口
ServletContext	ServletContextEvent	ServletContextListener
	ServletContextAttributeEvent	ServletContextAttributeListener

1. 处理ServletContextEvent事件

- 该事件是Web应用程序生命周期事件，当容器对ServletContext对象进行初始化或销毁操作时，将发生ServletContextEvent事件。
- 处理这类事件，需实现ServletContextListener接口，该接口定义如下两方法
 - void contextInitialized (ServletContextEvent sce)
当ServletContext对象初始化时调用。
 - void contextDestroyed (ServletContextEvent sce)
当ServletContext对象销毁时调用。

1. 处理ServletContextEvent事件

- 上述方法的参数是一个ServletContextEvent事件类对象，该类只定义了一个方法，如下所示。

```
public ServletContext  
    getServletContext()
```

- 该方法返回状态发生改变的ServletContext对象

2. 处理ServletContextAttributeEvent事件

- 当ServletContext对象上属性发生改变时，如添加属性、删除属性、替换属性等，将发生ServletContextAttributeEvent事件，处理该类事件，需要实现ServletContextAttributeListener接口。

2. 处理ServletContextAttributeEvent事件

- 该接口定义了如下三个方法。
- **public void attributeAdded(ServletContextAttributeEvent sre):** 当在ServletContext对象中添加属性时调用该方法。
- **public void attributeRemoved(ServletContextAttributeEvent sre):** 当从ServletContext对象中删除属性时调用该方法。
- **public void attributeReplaced(ServletContextAttributeEvent sre):** 当在ServletContext对象中替换属性时调用该方法。

2. 处理ServletContextAttributeEvent事件

- 上述方法的参数是ServletContextAttributeEvent类的对象，它是ServletContextEvent类的子类，它定义了下面三个方法。
- `public ServletContext getServletContext()`：返回属性发生改变的ServletContext对象。
- `public String getName()`：返回发生改变的属性名。
- `public Object getValue()`：返回发生改变的属性值对象。
- 注意，当替换属性时，该方法返回的是替换之前的属性值。

2. 处理ServletContextAttributeEvent事件

- 下面程序实现当Web应用启动时就创建一个数据源对象并将它保存在ServletContext对象上，当应用程序销毁时将数据源对象从ServletContext对象上清除，当ServletContext上属性发生改变时登记日志。
- [程序8.1 MyContextListener.java](#)
- 该程序在ServletContextListener接口的contextInitialized()中首先从InitialContext对象中查找数据源对象dataSource并将其存储在ServletContext对象中。
- 在ServletContext的属性修改方法中先通过事件对象的getServletContext()获得上下文对象，然后调用它的log()向日志中写一条消息。

2. 处理ServletContextAttributeEvent事件

- 下面的listenerTest.jsp页面是对监听器的测试，这里使用了监听器对象创建的数据源对象。
- 程序8.2 listenerTest.jsp
- 在该页面中首先通过隐含对象application的getAttribute()得到数据源对象，然后创建ResultSet对象访问数据库。

8.1.2 监听请求事件

- 在ServletRequest对象上可能发生两种事件，对这些事件使用两个事件监听器处理，如表8-2所示。

监听对象	事件	监听器接口
ServletRequest	ServletRequestEvent	ServletRequestListener
	ServletRequestAttributeEvent	ServletRequestAttributeListener

1. 处理ServletRequestEvent事件

- **ServletRequestEvent**是请求对象生命周期事件，当一个请求对象初始化或销毁时将发生该事件，处理该类事件需要使用**ServletRequestListener**接口如下两个方法：
 - **public void requestInitialized**
(**ServletRequestEvent sce**): 当请求对象初始化时调用。
 - **public void requestDestroyed**
(**ServletRequestEvent sce**): 当请求对象销毁时调用。

1. 处理ServletRequestEvent事件

- 上述方法的参数是ServletRequestEvent类对象，该类定义了下面两个方法：
- **public ServletContext getServletContext():**
返回发生该事件的ServletContext对象。
- **public ServletRequest getServletRequest():**
返回发生该事件的ServletRequest对象。

2. 处理ServletRequestAttributeEvent事件

- 在请求对象上添加、删除和替换属性时将发生ServletRequestAttributeEvent事件，处理该类事件需要使用ServletRequestAttributeListener接口，它定义了如下三个方法。
- `public void attributeAdded(ServletRequestAttributeEvent src)`: 当在请求对象中添加属性时调用该方法。
- `public void attributeRemoved`
`(ServletRequestAttributeEvent src)`:
当从请求对象中删除属性时调用该方法。
- `public void attributeReplaced`
`(ServletRequestAttributeEvent src)`:
当在请求对象中替换属性时调用该方法。

2. 处理ServletRequestAttributeEvent事件

- 在上述方法中传递的参数为 **ServletRequestAttributeEvent** 类的对象，该类定义了下面两个方法。
- **public String getName():** 返回在请求对象上添加、删除或替换的属性名。
- **public Object getValue():** 返回在请求对象上添加、删除或替换的属性值。
- 注意，当替换属性时，该方法返回的是替换之前的属性值。

2. 处理ServletRequestAttributeEvent事件

- 下面的MyRequestListener监听器类监听对某个页面的请求并记录自应用程序启动以来被访问的次数。
- [程序8.3 MyRequestListener.java](#)
- 下面是一个测试JSP页面：
- [程序8.4 onlineCount.jsp](#)

8.1.3 监听会话事件

- 在HttpSession对象上可能发生两种事件，对这些事件可使用四个事件监听器处理，这些类和接口如表8-3所示。

表 8-3 HttpSession 事件类与监听器接口

监听对象	事件	监听器接口
<u>HttpSession</u>	<u>HttpSessionEvent</u>	<u>HttpSessionListener</u>
		<u>HttpSessionActivationListener</u>
	<u>HttpSessionBindingEvent</u>	<u>HttpSessionAttributeListener</u>
		<u>HttpSessionBindingListener</u>

1. 处理HttpSessionEvent事件

- HttpSessionEvent事件是会话对象生命周期事件，当一个会话对象被创建和销毁时发生该事件，处理该事件应该使用HttpSessionListener接口，该接口定义了两个方法：
- `public void sessionCreated(HttpSessionEvent se)`: 当会话创建时调用该方法。
- `public void sessionDestroyed(HttpSessionEvent se)`: 当会话销毁时调用该方法。
- 上述方法的参数是一个HttpSessionEvent类对象，该类中只定义了一个`getSession()`，它返回状态发生改变的会话对象，格式如下。

```
public HttpSession getSession()
```

2. 处理会话属性事件

- 当在会话对象上添加属性、删除属性、替换属性时将发生HttpSessionBindingEvent事件，处理该事件需使用HttpSessionAttributeListener接口，该接口定义了下面三个方法：
- `public void attributeAdded(HttpSessionBindingEvent se):` 当在会话对象上添加属性时调用该方法。
- `public void attributeRemoved(HttpSessionBindingEvent se):` 当从会话对象上删除属性时调用该方法。
- `public void attributeReplaced(HttpSessionBindingEvent se):` 当替换会话对象上的属性时调用该方法。
- **注意：**上述方法的参数是HttpSessionBindingEvent，没有HttpSessionAttributeEvent这个类。

2. 处理会话属性事件

- HttpSessionBindingEvent类中定义了下面三个方法。
- **public HttpSession getSession():** 返回发生改变的会话对象。
- **public String getName():** 返回绑定到会话对象或从会话对象解除绑定的属性名。
- **public Object getValue():** 返回在会话对象上添加、删除或替换的属性值。

2. 处理会话属性事件

- 下面定义的监听器类实现了 `HttpSessionListener` 接口，它用来监视当前所有会话对象。当一个会话对象创建时，将其添加到一个 `ArrayList` 对象中并将其设置为 `ServletContext` 作用域的属性以便其他资源可以访问。当销毁一个会话对象时，从 `ArrayList` 中删除会话。
- [程序8.5 MySessionListener.java](#)
- [程序8.6 sessionDisplay.jsp](#)

3. 处理会话属性绑定事件

- 当一个对象绑定到会话对象或从会话对象中解除绑定时发生HttpSessionBindingEvent事件，应该使用HttpSessionBindingListener接口来处理这类事件，该接口定义的方法有：
- **public void valueBound(HttpSessionBindingEvent event):** 当对象绑定到一个会话上时调用该方法。
- **public void valueUnbound(HttpSessionBindingEvent event):** 当对象从一个会话上解除绑定时调用该方法。

3. 处理会话属性绑定事件

- 下面定义的User类实现了HttpSessionBindingListener接口。当将该类的一个对象绑定到会话对象上时，容器将调用valueBound()，当从会话对象上删除该类的对象时，容器将调用valueUnbound()，这里向日志文件写入有关信息。

程序8.7 User.java

- 程序从HttpSessionBindingEvent对象中检索会话对象，从会话对象中得到ServletContext对象并使用log()登录消息。

3. 处理会话属性绑定事件

- 下面是一个Servlet，它接受登录用户的用户名和口令，然后创建一个User对象并将其绑定到会话对象上。

程序8.8 LoginServlet.java

8.1.4 事件监听器的注册

- 从前面的例子中可看到，我们使用@WebListener注解来注册监听器，这是Servlet 3.0规范增加的功能。
- 事件监听器也可以在DD文件中使用<listener>元素注册。该元素只包含一个<listener-class>元素，用来指定实现了监听器接口的完整的类名。
- 下面代码给出了如何注册MyContextListener和MySessionListener两个监听器。

```
<listener>  
    <listener-class>com.listener.MyContextListener</listener-class>  
</listener>
```

```
<listener>  
    <listener-class>com.listener.MySessionListener</listener-class>  
</listener>
```

8.1.4 事件监听器的注册

- 在web.xml文件中并没有指定哪个监听器类处理哪个事件，这是因为当容器需要处理某种事件时，它能够找到有关的类和方法。容器实例化指定的类并检查类实现的全部接口。对每个相关的接口，它都向各自的监听器列表中添加一个实例。容器按照DD文件中指定的类的顺序将事件传递给监听器。这些类必须存放在WEB-INF\classes目录中或者与其他Servlet类一起打包在JAR文件中。
- **提示：**可以在一个类中实现多个监听器接口。这样，在部署描述文件中就只需要一个<listener>元素。容器就仅创建该类的一个实例并把所有的事件都发送给该实例。

8.2 Web过滤器

- 8.2.1 什么是过滤器
- 8.2.2 过滤器API
- 8.2.3 一个简单的过滤器
- 8.2.4 @WebFilter注解
- 8.2.5 在DD中配置过滤器

8.2.1 什么是过滤器

- **过滤器（Filter）**是Web服务器上的组件，它拦截客户对某个资源的请求和响应，对其进行过滤。图8-3说明了过滤器的一般概念，其中F1是一个过滤器。它显示了请求经过过滤器F1到达Servlet，Servlet产生响应再经过过滤器F1到达客户。这样，过滤器就可以在请求和响应到达目的地之前对它们进行监视。

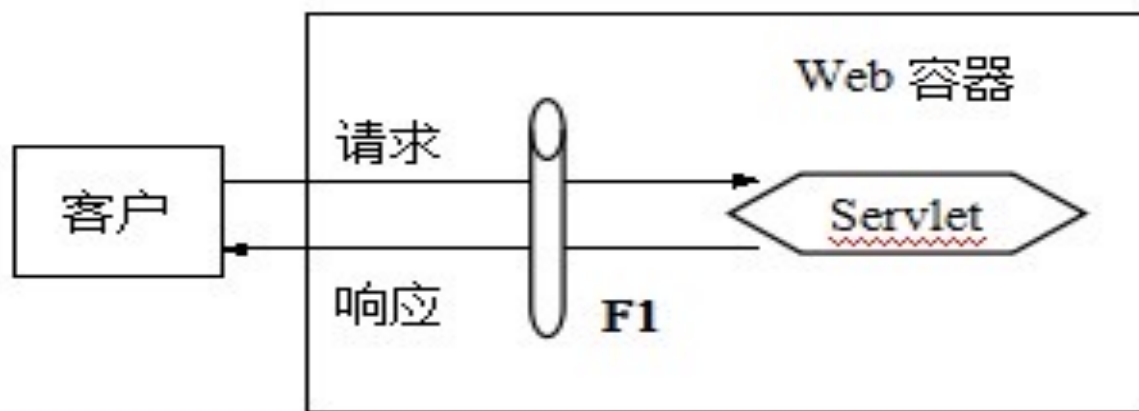


图 8-3 单个的过滤器

8.2.1 什么是过滤器

- 可以在客户和资源之间建立多个过滤器，从而形成**过滤器链**（**filter chain**）。在过滤器链中每个过滤器都对请求处理，然后将请求发送给链中的下一个过滤器（如果它是链中的最后一个，将发送给实际的资源）。类似地，在响应到达客户之前，每个过滤器以相反的顺序对响应处理。图8-4说明了这个过程。
- 这里，请求是按下列顺序处理的：过滤器F1、过滤器F2、过滤器F3，而响应的处理顺序是过滤器F3、过滤器F2、过滤器F1。

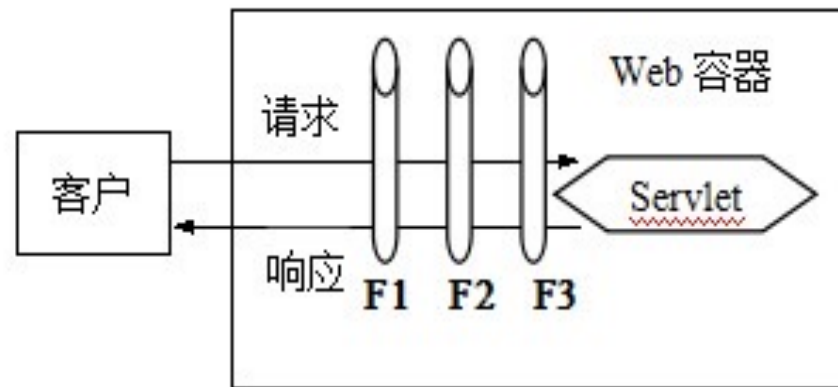


图 8-4 使用多个过滤器

1. 过滤器是如何工作的

- 当容器接收到对某个资源的请求时，它首先检查是否有过滤器与该资源关联。如果有过滤器与该资源关联，容器先把该请求发送给过滤器，而不是直接发送给资源。在过滤器处理完请求后，它将做下面三件事：
 - （1）将请求发送到目标资源。
 - （2）如果有过滤器链，它将把请求（修改过或没有修改过）发送给下一个过滤器。
 - （3）直接产生响应并将其返回给客户。
- 当请求返回到客户时，它将以相反的方向经过同一组过滤器。过滤器链中的每个过滤器都可能修改响应。

2. 过滤器的用途

- Servlet规范中提到的过滤器的一些常见应用包括：

验证过滤器

登录和审计过滤器

数据压缩过滤器

加密过滤器

XSLT过滤器

8.2.2 过滤器API

- 表8-4描述了javax.servlet包中与过滤器有关的三个接口。

表 8-4 过滤器使用的接口

接口	说明
Filter	所有的过滤器都需要实现该接口
<u>FilterConfig</u>	过滤器配置对象。容器提供了该对象，其中包含了该过滤器的初始化参数
<u>FilterChain</u>	过滤器链对象

1. Filter接口

- Filter接口是过滤器API的核心，所有的过滤器都必须实现该接口。该接口声明了三个方法，分别是init()、doFilter()和destroy()，它们是过滤器的生命周期方法。
- **init()**是过滤器初始化方法。在过滤器的生命周期中，init()仅被调用一次。在该方法结束之前，容器并不向过滤器转发请求。该方法的声明格式为：
public void init(FilterConfig filterConfig)
- 参数FilterConfig是过滤器配置对象，通常将FilterConfig参数保存起来以备备用。该方法抛出ServletException异常。

1. Filter接口

- **doFilter()**是实现过滤的方法。如果客户请求的资源与该过滤器关联，容器将调用该方法，格式如下：

```
public void doFilter(ServletRequest request,  
                    ServletResponse response,  
                    FilterChain chain)  
    throws IOException, ServletException;
```

- 该方法执行过滤功能，对请求进行处理或者将请求转发到下一个组件或者直接向客户返回响应。注意，request和response参数被分别声明为ServletRequest和ServletResponse的类型。

1. Filter接口

- 过滤器并不只限于处理HTTP请求。但如果过滤器用在使用HTTP协议的Web应用程序中，这些变量就分别指HttpServletRequest和HttpServletResponse类型的对象。在使用它们之前应把这些参数转换为相应的HTTP类型。
- `destroy()` 是容器在过滤器对象上调用的最后一个方法，声明格式为：

```
public void destroy();
```
- 该方法给过滤器对象一个释放其所获得资源的机会，在结束服务之前执行一些清理工作。

2. FilterConfig接口

- **FilterConfig**对象是过滤器配置对象，通过该对象可以获得过滤器名、过滤器运行的上下文对象以及过滤器的初始化参数。它声明了如下4个方法：
- `public String getFilterName()`
- `public ServletContext getServletContext()`
- `public String getInitParameter(String name)`
- `public Enumeration getInitParameterNames()`
- 容器提供了FilterConfig接口的一个具体实现类，容器创建该类的一个实例、使用初始化参数值对它初始化，然后将它作为一个参数传递给过滤器的`init()`。

3. FilterChain接口

- FilterChain接口只有一个方法，如下所示。

```
public void doFilter(ServletRequest request,  
                    ServletResponse response)  
    throws IOException, ServletException
```
- 在Filter对象的doFilter()中调用该方法使过滤器继续执行，它将控制转到过滤器链的下一个过滤器或实际的资源。
- 容器提供了该接口的一个实现并将它的一个实例作为参数传递给Filter接口的doFilter()。在doFilter()内，可以使用该接口将请求传递给链中的下一个组件，它可能是另一个过滤器或实际的资源。该方法的两个参数将被链中下一个过滤器的doFilter()或Servlet的service()接收。

8.2.3 一个简单的过滤器

- 下面是一个简单的日志过滤器，这个过滤器拦截所有的请求并将请求有关信息记录到日志文件中。程序声明的LogFilter类实现了Filter接口，覆盖了其中的init()、doFilter()和destroy()。
- [程序8.9 LogFilter.java](#)
- 程序在doFilter()中首先将请求对象（request）转换成HttpServletRequest的类型，然后获得当前时间、客户请求的URI和客户地址，并将其写到日志文件中。之后将请求转发到资源，当请求返回到过滤器后再得到当前时间，计算请求资源的时间并写到日志文件中。

8.2.3 一个简单的过滤器

- 要使过滤器起作用必须配置过滤器。对支持Servlet 3.0规范的容器，可以使用注解或DD文件的<filter>元素两种方法配置过滤器。本程序使用的是注解。
- 下面是访问helloworld应用中的addProduct.jsp页面后，在日志中将写入下面信息。

8.2.4 @WebFilter注解

- **@WebFilter**注解用于将一个类声明为过滤器，该注解在部署时被容器处理，容器根据具体的配置将相应的类部署为过滤器。表8-5给出该注解包含的常用元素。

8.2.4 @WebFilter注解

表 8-5 @WebFilter 注解的常用元素

元素名	类 型	说 明
<u>filterName</u>	String	指定过滤器的名称，等价于 web.xml 中的<filter-name>元素。如果没有显式指定，则使用 Filter 的完全限定名作为名称
<u>urlPatterns</u>	String[]	指定一组过滤器的 URL 匹配模式，该元素等价于 web.xml 文件中的<url-pattern>元素
value	String[]	该元素等价于 <u>urlPatterns</u> 元素。两个元素不能同时使用
<u>servletNames</u>	String[]	指定过滤器应用于哪些 <u>Servlet</u> 。取值是 @WebServlet 中 name 属性值，或者是 web.xml 中<servlet-name>的取值
<u>dispatcherTypes</u>	<u>DispatcherType</u>	指定过滤器的转发类型。具体取值包括：ASYNC、ERROR、FORWARD、INCLUDE 和 REQUEST
<u>initParams</u>	<u>WebInitParam</u> []	指定一组过滤器初始化参数，等价于<init-param>元素
<u>asyncSupported</u>	boolean	声明过滤器是否支持异步调用，等价于<async-supported>元素
description	String	指定该过滤器的描述信息，等价于<description>元素
<u>display</u> Name	String	指定该过滤器的显示名称，等价于<display-name>元素

8.2.4 @WebFilter注解

- 上表中所有属性均为可选属性，但是 **value**、**urlPatterns**、**servletNames** 三者必须至少包含一个，且 **value** 和 **urlPatterns** 不能共存，如果同时指定，通常忽略 **value** 的取值。
- 过滤器接口 **Filter** 与 **Servlet** 非常相似，它们具有类似的生命周期行为，区别只是 **Filter** 的 **doFilter()** 中多了一个 **FilterChain** 的参数，通过该参数可以控制是否放行用户请求。像 **Servlet** 一样，**Filter** 也可以具有初始化参数，这些参数可以通过 **@WebFilter** 注解或部署描述文件定义。在过滤器中获得初始化参数使用 **FilterConfig** 实例的 **getInitParameter()**。

8.2.4 @WebFilter注解

- 在实际应用中，使用**Filter**可以更好实现代码复用。例如，一个系统可能包含多个**Servlet**，这些**Servlet**都需要进行一些通用处理，比如权限控制、记录日志等，这将导致多个**Servlet**的**service()**中包含部分相同代码。为解决这种代码重复问题，就可以考虑把这些通用处理提取到**Filter**中完成，这样在**Servlet**中就只剩下针对特定请求相关的处理代码。
- 下面定义一个较为实用的**Filter**，它对用户请求进行过滤，为请求设置编码字符集，从而可以避免为每个**JSP**页面、**Servlet**都设置字符集。该**Filter**还能实现验证用户是否登录，如果用户没有登录，系统直接跳转到登录页面。

8.2.4 @WebFilter注解

- 下面定义一个较为实用的**Filter**，它对用户请求进行过滤，为请求设置编码字符集，从而可以避免为每个**JSP**页面、**Servlet**都设置字符集。该**Filter**还能实现验证用户是否登录，如果用户没有登录，系统直接跳转到登录页面。
- [程序8.10 AuthorityFilter.java](#)

8.2.4 @WebFilter注解

- 该过滤器通过@WebFilter注解的initParams元素指定了三个初始化参数，参数使用@WebInitParam注解指定，每个@WebInitParam指定一个初始化参数。在Filter的doFilter()中通过FilterConfig对象取出参数的值。
- 程序中设置了请求的字符编码，还通过Session对象验证用户是否登录，若没登录将请求直接转发到登录页面，若已登录则转发到请求的资源。

8.2.5 在DD中配置过滤器

- 除了可以通过注解配置过滤器外，还可以使用部署描述文件web.xml配置过滤器类并把请求URL映射到该过滤器上。
- 配置过滤器要用下面两个元素：`<filter>`和`<filter-mapping>`。
- 每个`<filter>`元素向Web应用程序引进一个过滤器，每个`<filter-mapping>`元素将一个过滤器与一组请求URI关联。两个元素都是`<web-app>`的子元素。

1. <filter>元素

- 该元素用来指定过滤器名和过滤器类，下面是<filter>元素的DTD定义：

```
<!ELEMENT filter (description?, display-name?, icon?, filter-name, filter-class, init-param*)>
```

- 从上面定义可以看到，每个过滤器都需要一个<filter-name>元素和一个<filter-class>元素。其他元素是可选的。下面代码说明了<filter>元素的使用。

1. <filter>元素

<filter>

<!--指定过滤器名和过滤器类-->

<filter-name>validatorFilter</filter-name>

<filter-class>filter.ValidatorFilter</filter-class>

<init-param>

 <param-name>locale</param-name>

 <param-value>USA</param-value>

</init-param>

</filter>

1. <filter>元素

- 这里定义了一个名为**validatorFilter**的过滤器，同时为该过滤器定义了一个名为**locale**的初始化参数。这样，在应用程序启动时容器将创建一个**filter.ValidatorFilter**类的实例。在初始化阶段，过滤器将调用**FilterConfig**对象的**getParameterValue("locale")**检索**locale**参数的值。

2. <filter-mapping>元素

- 该元素的作用定义过滤器映射，<filter-mapping>元素的DTD定义如下：

```
<!ELEMENT filter-mapping (filter-name, (url-pattern | servlet-name),dispatcher)>
```
- <filter-name>元素是在<filter>元素中定义的过滤器名，<url-pattern>用来将过滤器应用到一组通过URI标识的请求，<servlet-name>用来将过滤器应用到通过该名标识的Servlet提供服务的所有请求。在使用<servlet-name>情况下，模式匹配遵循与Servlet映射同样的规则。

2. <filter-mapping>元素

下面代码说明了<filter-mapping>元素的使用:

```
<filter-mapping>
```

```
    <filter-name>validatorFilter</filter-name>
```

```
    <url-pattern>*.jsp</url-pattern>
```

```
</filter-mapping>
```

```
<filter-mapping>
```

```
    <filter-name>validatorFilter</filter-name>
```

```
    <servlet-name>reportServlet</servlet-name>
```

```
</filter-mapping>
```

2. <filter-mapping>元素

- 上面的第一个映射将**validatorFilter**与所有的请求URL后缀为.jsp的请求相关联。第二个映射将**validatorFilter**与所有对名为**reportServlet**的Servlet的请求相关联。这里使用的Servlet名必须是部署描述文件中使用<servlet>元素定义的一个Servlet。

3. 配置过滤器链

- 在某些情况下，对一个请求可能需要应用多个过滤器，这样的过滤器链可以使用多个<filter-mapping>元素配置。
- 当容器接收到一个请求，它将查找所有与请求URI匹配的过滤器映射的URL模式，这是过滤器链中的第一组过滤器。
- 接下来，它将查找与请求URI匹配的Servlet名，这是过滤器链中的第二组过滤器。在这两组过滤器中，过滤器的顺序是它们在DD文件中的顺序。

3. 配置过滤器链

- 为了理解这个过程，考虑下面对过滤器和Servlet映射代码。

```
<servlet-mapping>
```

```
    <servlet-name>FrontController</servlet-name>
```

```
    <url-pattern>*.do</url-pattern>
```

```
</servlet-mapping>
```

```
<filter-mapping>
```

```
    <filter-name>perfFilter</filter-name>
```

```
    <servlet-name>FrontController</servlet-name>
```

```
</filter-mapping>
```

3. 配置过滤器链

```
<filter-mapping>
```

```
    <filter-name>auditFilter</filter-name>
```

```
    <url-pattern>*.do</url-pattern>
```

```
</filter-mapping>
```

```
<filter-mapping>
```

```
    <filter-name>transformFilter</filter-name>
```

```
    <url-pattern>*.do</url-pattern>
```

```
</filter-mapping>
```

- 如果一个请求URI为/admin/addCustomer.do，将以下的顺序应用过滤器： auditFilter、transformFilter、perfFilter。

4. 为转发的请求配置过滤器

- 从Servlet 2.4开始，过滤器还可以应用在从组件内部转发的请求上，这包括使用RequestDispatcher的include()和forward()转发的请求以及对错误处理调用的资源的请求。
- 要为转发的请求配置过滤器，可以使用<filter-mapping>元素的子元素<dispatcher>实现，该元素的取值包括下面4个：REQUEST、INCLUDE、FORWARD和ERROR。

4. 为转发的请求配置过滤器

- REQUEST表示过滤器应用在直接来自客户的请求上。
- INCLUDE表示过滤器应用在与调用 **RequestDispatcher** 的 `include()` 匹配的请求。
- FORWARD表示过滤器应用在与调用 **RequestDispatcher** 的 `forward()` 匹配的请求。
- ERROR表示过滤器应用于由在发生错误而引起转发的请求上。

4. 为转发的请求配置过滤器

- 在<filter-mapping>元素中可以使用多个<dispatcher>元素使过滤器应用在多种情况下，例如：

```
<filter-mapping>
```

```
    <filter-name>auditFilter</filter-name>
```

```
    <url-pattern>*.do</url-pattern>
```

```
    <dispatcher>INCLUDE</dispatcher>
```

```
    <dispatcher>FORWARD</dispatcher>
```

```
</filter-mapping>
```

- 上述过滤器映射将只应用在从内部转发的且其URL与*.do匹配的请求上，任何直接来自客户的请求，即使其URL与*.do匹配也将不应用auditFilter过滤器。

8.3 Servlet的多线程问题

- 在Web应用程序中，一个Servlet在一个时刻可能被多个用户同时访问。这时Web容器将为每个用户创建一个线程。如果Servlet不涉及共享资源的问题，不必关心多线程问题。但如果Servlet需要共享资源，需要保证Servlet是线程安全的。
- 下面首先来看一个非线程安全的Servlet。该Servlet从客户接受两个整数，然后计算它们的和或差。
- [程序8.11 CalculatorServlet.java](#)

- 该程序将计算结果存放在变量`result`中，它根据用户在页面中单击的是“相加”按钮或“相减”按钮决定求和还是求差。
- 注意，`result`被声明为一个成员变量。为了演示多个用户请求时出现的问题，程序中调用`Thread`类的`sleep()`在计算出`result`后睡眠一段时间（假设2秒种），睡眠的时间通过`Servlet`初始化参数`sleepTime`得到。`getNumber()`实现字符串到`int`数据的转换。最后输出计算的结果。

- 下面是一个JSP页面，其中的表单包含两个文本框用来接受两个整数，两个提交按钮，一个做加法、一个做减法。
- [程序8.12 calculator.jsp](#)
- 下面测试该Servlet的执行。打开两个浏览器窗口，每个窗口都载入calculator.jsp页面，在两个页面的文本框中都输入100和50，如图8-5所示。



图 8-5 在两个窗口打开 calculator.jsp 页面

- 然后单击第一个页面中的“相加”按钮，在2秒钟内单击第二个页面的“相减”按钮，得到运行结果如图8-6和图8-7所示。

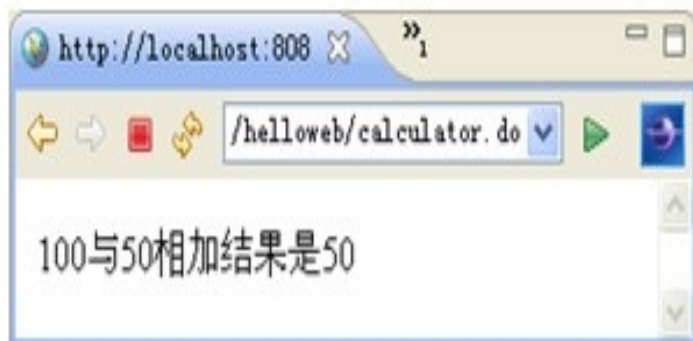


图 8-6 单击“相加”按钮的结果



图 8-7 单击“相减”按钮的结果

- 从运行结果可以看到，其中一个结果是正确的（第二个页面），另一个结果是错误的。该Servlet的执行过程如下：当两个用户同时访问该Servlet时，服务器创建两个线程来提供服务。当第一个用户提交表单后，它执行其所在线程的doPost()，计算100与50的和并将结果150存放在result变量中，然后在输出前睡眠2秒钟。在这个时间内，当第二个窗口提交时，它将计算100与50的差，将结果50写到result变量中，此时第一个线程的计算结果被覆盖，当第一个线程恢复执行后输出结果也为50。

- 出现这种错误的原因是在Servlet中使用成员变量result来保存请求计算结果，成员变量在多个线程（请求）中只有一份拷贝，而这里的result应该是请求的专有数据。解决这个问题的是用方法的局部变量来保存请求的专有数据。这样，进入方法的每个线程都有自己的一份方法变量的拷贝，任何线程都不会修改其他线程的局部变量。

- 除了上述这种简单情况外，**Servlet**还经常要共享外部资源，如使用一个数据库连接对象。如果将连接对象声明为**Servlet**的成员变量，则当多个并发的请求在同一个连接上写入数据时，数据库将产生错误的数
据，因此通常不用成员变量来保存请求的
专有数据。

- 下面是编写线程安全的Servlet的一些建议：

（1）用方法的局部变量保存请求中的专有数据。对方法中定义的局部变量，进入方法的每个线程都有自己的一份方法变量拷贝。如果要在不同的请求之间共享数据，应该使用会话来共享这类数据。

（2）只用Servlet的成员变量来存放那些不会改变的数据。有些数据在Servlet生命周期中不发生变化，通常是在初始化时确定的，这些数据可以使用成员变量保存。如，数据库连接名称、其他资源的路径等。在上述例子中sleepTime的值是在初始化时设定的并在Servlet的生命期内不发生改变，所以可以把它定义为一个成员变量。

(3) 对可能被请求修改的成员变量同步（使用synchronized关键字）。有时数据成员变量或者环境属性可能被请求修改。当访问这些数据时应该对它们同步，以避免多个线程同时修改这些数据。

(4) 如果Servlet访问外部资源，那么需要对这些资源同步。例如，假设Servlet要从文件中读写数据。当一个线程读写一个文件时，其他线程也可能正在读写这个文件。文件访问本身不是线程安全的，所以必须编写同步代码访问这些资源。

- 在编写线程安全的Servlet时，下面两种方法是不应该使用的：

（1）在Servlet API中提供了一个SingleThreadModel接口，实现这个接口的Servlet在被多个客户请求时一个时刻只有一个线程运行。这个接口已被标记不推荐使用。

（2）对doGet()或doPost()同步。如果必须在Servlet中使用同步代码，应尽量在最小的代码块范围上进行同步。同步代码越少，Servlet执行效率越高。

。

8.4 Servlet的异步处理

- Servlet 3.0之前，Servlet的执行过程大致如下：Web容器接收到用户对某个Servlet请求之后启动一个线程，在该线程中对请求的数据进行预处理；接着，调用业务接口的某些方法，以完成业务处理；最后，根据处理的结果提交或转发响应，Servlet线程结束。其中第二步的业务处理通常是最耗时的，如访问数据库操作，跨网络调用等。此时，Servlet线程一直处于阻塞状态，直到业务执行完毕。在此过程中，线程资源一直被占用而得不到释放，对于并发用户较多的应用，这有可能造成性能的瓶颈。

8.4.1 概 述

- **Servlet 3.0**增加了异步处理支持，**Servlet**的执行过程调整如下：**Web**容器接收到用户对某个**Servlet**请求之后启动一个线程，在该线程中对请求的数据进行预处理；接着，**Servlet**线程将请求转交给一个异步线程来执行业务处理，**Servlet**线程本身返回至容器，此时**Servlet**还没有生成响应数据。异步线程处理完业务以后，可以直接生成响应数据（异步线程拥有**ServletRequest**和**ServletResponse**对象的引用），或者将请求转发给其他**Servlet**或**JSP**页面。这样，**Servlet**线程不再是一直处于阻塞状态以等待业务逻辑的处理，而是启动异步线程之后可以立即返回。

- 异步线程处理可应用于Servlet和过滤器两种组件，由于异步处理的工作模式和普通工作模式在实现上有着本质的区别，因此默认情况下，Servlet和过滤器并没有开启异步处理特性。
- Servlet 3.0的异步处理是通过AsyncContext类来实现的，Servlet可以通过ServletRequest的如下两个方法创建AsyncContext对象，开启异步调用。
- **public AsyncContext startAsync():** 开始异步调用并返回AsyncContext对象，其中包含最初的请求和响应对象。

- `public AsyncContext startAsync(ServletRequest request, ServletResponse response)`: 开启异步调用，并传递经过包装的请求和响应对象。
- **AsyncContext**表示异步处理的上下文，该类提供了一些工具方法，可完成启动后台线程、转发请求、设置异步调用的超时时长、获取`request`和`response`对象等功能。

8.4.2 异步调用Servlet的开发

- 编写异步Servlet和过滤器很简单。如果一个任务需要花费较长时间完成，就应该通过异步Servlet实现。在异步Servlet中一般需要执行下面操作：
- 调用ServletRequest对象的startAsync()，该方法返回AsyncContext对象，它是异步处理的上下文对象。
- 调用AsyncContext对象的setTimeout()，传递一个毫秒时间设置容器等待指定任务完成的时间。如果没有设置超时时间，容器将使用默认时间。在指定的时间内任务不能完成将抛出异常。
- 调用AsyncContext对象的start()，为其传递一个要用异步线程执行的Runnable对象。

- 下面是一个简单的模拟异步处理的Servlet。
- [程序8.13 AsyncDemoServlet.java](#)
- 该类中创建了一个AsyncContext对象，并通过该对象以异步的方式启动了一个后台线程。该线程执行体模拟调用耗时的业务方法，下面的Executor类就是线程体类。
- [程序8.14 Executor.java](#)
- 从运行结果可以看到，主线程结束时异步线程还没有结束，主线程已经返回给容器。
- 对于希望启用异步调用的Servlet而言，开发者必须显式指定开启异步调用，有两种方式指定异步调用：
 - 为@WebServlet注解指定asyncSupported=true。

- 在web.xml文件的<servlet>元素中增加<async-supported>子元素。
- Servlet 3.0为<servlet>和<filter>标签增加了<async-supported>子标签，该标签的默认取值为false，要启用异步处理支持，则将其设为true即可。以Servlet为例，其配置如下所示：

```
<servlet>  
    <servlet-name>AsyncDemoServlet</servlet-name>  
    <servlet-class>com.demo.AsyncDemoServlet  
        </servlet-class>  
    <async-supported>true</async-supported>  
</servlet>
```

8.4.3 实现AsyncListener接口

- 在Servlet 3.0中增加了一个AsyncListener接口来处理异步操作事件。当Servlet启用异步调用时发生AsyncEvent事件。要处理这类事件，需实现AsyncListener接口，该接口定义了如下4个方法：
- `public void onStartAsync (AsyncEvent event)`: 当异步调用开始时触发该方法。
- `public void onComplete (AsyncEvent event)`: 当异步调用完成时触发该方法。
- `public void onError (AsyncEvent event)`: 当异步调用出错时触发该方法。
- `public void onTimeout (AsyncEvent event)`: 当异步调用超时触发该方法。

- 上述方法的参数是AsyncEvent类的对象，通过该对象的getAsyncContext()、getSuppliedRequest()和getSuppliedResponse()可分别返回AsyncContext对象、ServletRequest对象及ServletResponse对象等。
- 下面的MyAsyncListener类实现了AsyncListener接口，当发生异步操作事件时可由它处理。

程序8.15 MyAsyncListener.java

- **注意：** 与其他Web监听器不同，AsyncListener监听器不需要使用@WebListener注册或在web.xml中注册，但需要使用AsyncContext对象的addListener()进行手工注册，该方法格式如下：
public void addListener(AsyncListener listener)
- 下面的AsyncListenerServlet是一个异步Servlet，它使用了上述监听器。

程序8.16 AsyncListenerServlet.java

8.5 小 结

- 在Web应用程序运行过程中会发生某些事件，为了处理这些事件，容器也采用了事件监听器模型。根据事件的类型和范围，可以把事件监听器分为三类：**ServletContext**事件监听器、**HttpSession**事件监听器和**ServletRequest**事件监听器。
- 对Web应用来说，过滤器是Web服务器上的组件，它们对客户和资源之间的请求和响应进行过滤。可以定义多种类型的过滤器，如验证过滤器、审计过滤器、数据压缩过滤器、加密过滤器等。

8.5 小 结

- 在编写有多个用户同时访问的**Servlet**时，一定要保证**Servlet**是线程安全的。一般不使用成员变量共享请求的专有数据。如果必须要在多个请求之间共享数据，则应当对这些数据同步，并且尽量在最小的代码块范围上进行同步。
- **Servlet 3.0**新增的**Servlet**和过滤器的异步处理线程的目的是提高系统的性能。它通过**AsyncContext**对象的**startAsync()**和**start()**开始和启动一个异步执行的线程，使当前线程立即返回，使耗时的操作在一个异步线程中执行，从而提高系统的性能。