第 20 章　Servlet过滤器和监听器

前一章介绍了 Servlet 的 Web 应用程序开发过程，本章重点讲解 Servlet 的两个特殊的应用：过滤器和监听器功能。了解到 Servlet 在 Web 应用开发过程中怎么补充 JSP 的功能，使得 Web 系统更加的安全和健壮。

本章要点包括以下内容：

- Servlet 过滤器介绍
- Servlet 过滤器的体系结构
- Servlet 过滤器实例
- Servlet 上下文监听器实例
- HTTP 会话监听器实例
- 客户端请求监听器实例

20.1　Servlet过滤器

Servlet API 很久以前就已经运用到了企业 Web 应用开发的过程中（从而实现页面动态效果，虽然现在更多使用了 JSP），而 Servlet 过滤器则是对 J2EE 家族的相对较新的补充。本小节将向读者介绍 Servlet 过滤器的体系结构，定义过滤器的应用，并指导读者完成典型过滤器实现的三步骤。

20.1.1　过滤器的简介

Servlet 实现的过滤器功能是 Web 中的一个小型组件，其拦截来自客户端的请求和响应信息，从而进行查看、提取或者以对客户端和服务器之间交换的数据信息进行某项特定的操作。实现的过滤器通常是用来封装一些辅助性的功能方法，这些过滤器方法可能对真正意义上的客户端请求和响应处理不起决定性作用，但还是非常重要的。可以使用 Servlet 中的过滤器来记录有关客户端请求和响应的数据、处理数据传递过程中所存在的安全性问题以及管理会话属性等等。过滤器是提供了一种面向对象的模块化机制，将公共的过滤器方法封装到可插入到组件中，然后再由一个配置文件来声明这些组件，并动态地进行调用和处理。

Servlet 中的过滤器结合了许多元素，已经使得过滤器成为独特、强大和模块化的 Web 组件。从而概括起来说，Servlet 过滤器是：

- 声明式的：编写好过滤器之后，在动态调用过滤器来处理客户端请求以及响应之前，还需要通过 Web 部署描述符（web.xml）来声明这个过滤器。通过部署描述器，可以允许动态地添加和删除过滤器，并不需改动任何应用程序代码或者 JSP 页面。
- 动态的：通过部署配置文件（web.xml），过滤器在运行时，会动态地由 Servlet 容器调用特定地方法来拦截和处理请求和响应。
- 灵活的：过滤器在 Web 应用处理过程中的使用还是很广泛的，其中涉及到诸如日志记录和安全检测等许多最为公共的辅助任务。过滤器的使用也是很灵活的，因为它可以用来对来自客户端

的直接调用执行预处理和后期等处理，以及处理防火墙后面的 Web 组件间的调度请求。最后，还可以将多个过滤器链接起来以提供更多必需的功能。

- 模块化的：通过将应用程序中的对请求和响应进行的逻辑处理封装到单个类文件（Servlet 文件）中，过滤器中已经定义了可容易地从请求/响应链中添加或删除的模块化单元。
- 可移植的：与 Java 平台中开发的其他许多类文件一样，Servlet 过滤器也是跨平台、跨容器以及可移植的，从而更加能够支持了 Servler 过滤器的模块化和可重用本质。
- 可重用的：由于过滤器实现类的模块化设计，以及通过部署描述器 Web.xml 进行声明的过滤器配置方式，过滤器可以容易地进行添加和删除，以及进行过滤器的可重用性。
- 透明的：允许在对客户端请求和响应编写特定的过滤器，这是为了补充对（而不是以任何方式替代）servlet 或 JSP 页面数据提供一定的额外处理。因而，可以根据需要，在部署描述器 web.xml 添加或删除过滤器，而不会破坏 servlet 或 JSP 页面程序。

Servlet 实现的过滤器是需要通过一个配置文件（web.xml）来灵活声明的模块化以及可重用组件。过滤器是动态地（不需要程序员编写程序进行调用）处理传入的客户端请求和传出的响应，并且也不需要修改任何应用程序代码就可以透明地通过配置文件添加或删除某个过滤器。最后，过滤器也具有可移植性，可以独立于任何平台或者 Servlet 容器，从而允许将这些过滤器容易地部署到大部分的 J2EE 环境中。

在接下来的小节中，将进一步考察 Servlet 过滤器的总体设计，以及如何实现、配置和部署过滤器所涉及的步骤。最后，还将继续探讨 Servlet 过滤器的一些实际应用，简要考察一下模型-视图-控制器（MVC）体系结构中包含的 Servlet 过滤器。

20.1.2　Servlet过滤器的体系结构

其实，过滤器就是一个 Servlet，只是比较特殊而已，可以使用它来对客户端请求以及发出的响应进行部分的过滤操作，多个多虑器一起就可以组成一个过滤器链。

过滤器并不真正意义上对来自客户端的请求作出最终的响应，而是进行部分的过滤工作，对请求数据作检测或者安全性检查等工作。多个特定的过滤器可以一起组成一个过滤器链。

过滤器只有在客户端请求或者发出响应的发生时才会自动进行调用，所有没有必要把这些过滤器字节嵌入到整个 Web 应用系统的框架中，而是通过配置文件来设定。

当然了，大部分情况下，并不需要给 Web 应用程序设置过滤器，因为它不是必需的。如果需要设置过滤器，除了可以设置单个过滤器关联，还可以设置 Web 应用程序与一个过滤器链（多个单过滤器相串联）相关联。

那么整个过滤器的工作过程是如何的呢？下面列举出过程步骤：

（1）首先，通过部署描述器 web.xml 中对过滤器的配置，特定过滤器会捕捉到客户端的请求信息。

（2）过滤器调用内置方法来检查捕获到的请求对象，根据分析结果来决定是把该请求传递给下一个过滤器（如果存在另一个过滤器的话），还是中止该请求并向客户端发出一个响应。如果不存在过滤器链，即仅仅为单个过滤器，则上一个过滤器直接把客户请求转发给 Web 服务器作相应处理。

（3）如果 Web 应用程序关联了过滤器，在客户端请求设法通过各个过滤器被服务器处理时，最终的响应将以相反的顺序通过过滤器链，最后发送给客户端。

过滤器的概念是从 Servlet2.3 才开始引入的，一开始仅仅能够过滤 Web 客户端和客户首次所需访问服务器资源之间的数据信息。当第一个访问的资源（例如某一个 JSP 页面）把客户端请求再转发给另外一个服务器资源（例如另外一个 JSP 页面）时，这时过滤器就不起作用了。到了 Servlet2.4 发布后，这

样的过滤器限制得到了修改。也就是说，现在的过滤器可以作用于 J2EE 环境中的任何请求过程中，可以在客户端和 servlet 以及 JSP 之间，也可以是服务器端的 Servlet 与 Servlet 以及 JSP 页面之间。这大大增加了过滤器在 Web 应用程序中的可适用性。

20.1.3　实现一个Servlet过滤器

在 Web 应用程序中实现一个 Servlet 过滤器需要经历三个步骤：

（1）首先要编写 Servlet 过滤器的实现类程序。

（2）第二步，需要把实现的过滤器添加到 Web 应用程序中，也就是说，需要在 Web 部署描述文件 web.xml 中声明该过滤器。

（3）最后要把相关联的过滤器与应用程序一起打包并部署。

下面将详细介绍其中的每个步骤：

20.1.3.1．编写过滤器的实现类程序

编写过滤器一般需要使用到如下 3 个简单的接口： Filter.java、FilterChain.java 和 FilterConfig.java，这三个接口封装在 java.servlet 包中。从编程的角度来看，过滤器类必须要实现 Filter 接口，然后使用这个过滤器类中的 FilterChain 和 FilterConfig 接口。该过滤器类的一个引用将传递给 FilterChain 对象，以允许过滤器把控制权传递给链中的下一个资源。FilterConfig 对象将由容器提供给过滤器，以允许访问该过滤器的初始化数据。

与实现 Servlet 过滤器的三步模式保持一致，过滤器必须实现如下的三个方法，以便完全实现 Filter 接口：

- ❑ init()：和普通 Servlet 中的 init()方法一样，该方法是在 Servlet 容器实例化过滤器时被调用，主要设计用于使过滤器为处理做准备。该方法接受一个 FilterConfig 类型的对象作为输入。
- ❑ doFilter()：与 Servlet 拥有一个 service()方法（这个方法又调用 doPost()或者 doGet()）来处理请求一样，过滤器拥有单个用于处理请求和响应的方法：doFilter()。这个方法接受三个输入参数：一个 ServletRequest、response 和一个 FilterChain 对象。
- ❑ destroy()：正如您想像的那样，这个方法执行任何清理操作，这些操作可能需要在自动垃圾收集之前进行。

下面创建的 TimeTrackFilter.java 类展示了一个非常简单的过滤器，它跟踪满足一个客户机的 Web 请求所花的大致时间：

```
import javax.servlet.*;
import java.util.*;
import java.io.*;
public class TimeTrackFilter implements Filter {
  private FilterConfig filterConfig = null;
                                                    //初始化过滤器
  public void init(FilterConfig filterConfig)throws ServletException {
    this.filterConfig = filterConfig;
  }
                                                    //执行过滤器功能
    public void doFilter( ServletRequest request,ServletResponse response, FilterChain chain )
                        throws IOException, ServletException {
          Date startTime, endTime;
          double totalTime;
          startTime = new Date();
```

```
            chain.doFilter(request, response);          //把处理发送到下一个过滤器
                                                        // 接下来的语句就是处理请求的过程:计算开始到结束的时间
            endTime = new Date();
            totalTime = endTime.getTime() - startTime.getTime();
            totalTime = totalTime / 1000; //Convert from milliseconds to seconds
            StringWriter sw = new StringWriter();
            PrintWriter writer = new PrintWriter(sw);
            writer.println();
            writer.println("===============");
            writer.println("Total elapsed time is: " + totalTime + " seconds." );
            writer.println("===============");
                                                        // 把结果写到日志当中
            writer.flush();
            filterConfig.getServletContext().
            log(sw.getBuffer().toString());
    }
                                                        //销毁过滤器
        public void destroy() {
            this.filterConfig = null;
        }
}
```

程序说明：

- 初始化：当容器第一次加载该过滤器时，init()方法将被调用。该类在这个方法中包含了一个指向 FilterConfig 对象的引用。我们的过滤器实际上并不需要这样做，因为其中没有使用初始化信息，这里只是出于演示的目的。
- 过滤：过滤器的大多数时间都消耗在这里。doFilter()方法被容器调用，同时传入分别指向这个请求/响应链中的 ServletRequest、ServletResponse 和 FilterChain 对象的引用。然后过滤器就有机会处理请求，将处理任务传递给链中的下一个资源（通过调用 FilterChain 对象引用上的 doFilter()方法），之后在处理控制权返回该过滤器时处理响应。
- 析构：容器紧跟在垃圾收集之前调用 destroy()方法，以便能够执行任何必需的清理代码。

20.1.3.2．在 web.xml 文件中配置 Servlet 过滤器

实现了 Servlet 过滤器后，还需要通过 web.xml 文件中的两个 XML 元素来声明该过滤器。＜filter＞元素定义过滤器的名称，并且声明实现类和 init()参数。＜filter-mapping＞元素将过滤器与 servlet 或 URL 模式相关联。

下面展示在 web.xml 配置文件中声明一个过滤器的方法，把下面这段配置代码插入到 web.xml 文件中的</web-app>标记前面：

```
＜filter＞
＜filter-name＞Page Request Timer＜/filter-name＞
＜filter-class＞TimeTrackFilter＜/filter-class＞
＜/filter＞
＜filter-mapping＞
＜filter-name＞Page Request Timer＜/filter-name＞
＜servlet-name＞Main Servlet＜/servlet-name＞
＜/filter-mapping＞
＜servlet＞
＜servlet-name＞Main Servlet＜/servlet-name＞
```

```
<servlet-class>MainServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Main Servlet</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
```

上面的代码示例声明了一个过滤器（"Page Request Timer"），并把它映射到一个 servlet（"Main Servlet"）。然后为该 servlet 定义了一个映射，以便把每个请求（由通配符指定）都发送到该 servlet。这是控制器组件的典型映射声明。您应该注意这些声明的顺序，因为千万不能背离这些元素的顺序。

20.1.3.3．部署 Servlet 过滤器

Servlet 过滤器的部署非常简单，只需把过滤器的实现类和其他 Web 组件类放置在一起。重点是部署描述器 web.xml 对各类过滤器的配置要正确，然后把配置文件 web.xml 文件（连同过滤器定义和过滤器映射声明）放进 Web 应用程序结构中，servlet 容器将处理之后的其他所有事情。

20.1.4　过滤器的许多应用

过滤器不是真正使用来完成客户端请求的，客户端请求操作最终还是要交付给服务器进行相应处理，过滤器只是做一些辅助性的工作。但是在某些情况下，过滤器是非常有用的，例如安全性检查等。在适合可以使用装饰过滤器模式或者拦截模式的地方都可以使用过滤器。总结起来，过滤器可以使用在如下方面：

- 加载：对所有到达系统的客户端请求，可以使用过滤器来收集诸如浏览器类型、一天中的时间、转发 URL 等相关信息，并把这些信息记录到日志中。
- 性能：通过使用过滤器可以提供处理性能。例如，当客户端数据通过协议传送并在到达 Servlet 和 JSP 页面之前，可以使用过滤器将请求数据打包压缩，然后再取得响应内容。同样地，在将响应内容发送到客户机机器之前，过滤器再将它转换为压缩格式在网络上传输。
- 安全：有时在安全性要求特别高的 Web 应用系统中，过滤器的作用可以得到很好的发挥。过滤器可是使用来对发出请求的客户身份进行验证，以防止非法用户访问特定资源，从而限制某些资源的访问。过滤器除了可以进行客户身份验证之后，还可以管理客户访问列表（Access Control List，ACL），从而可以对客户进行授权管理。使用过滤器来进行安全性管理，而不是让 Servlet 或者 JSP 程序来执行，这是为了提供更高的灵活性，因为过滤器可以随时通过 web.xml 配置文件进行添加和关闭。所以一般情况，这些辅助的工作都可以放置在过滤器中完成。
- 会话处理：将 servlet 和 JSP 页面与会话处理代码混杂在一起可能会带来相当大的麻烦。使用过滤器来管理会话可以让 Web 页面集中精力考虑内容显示和委托处理，而不必担心会话管理的细节。
- XSLT 转换：在 Web 应用程序如果使用到 XML，经常会涉及到 XSLT 转换的问题，这时就可以把转换工作写入到过滤器中，而让服务器端的 Servlet 或者 JSP 真正去处理数据和逻辑。

20.1.5　使过滤器适应MVC体系结构

在下一章节中将会向读者重点介绍模型-视图-控制器（Model-View- Controller，MVC）体系结构，和 Model1 模式相比，MVC 模式的优势非常明显。而且 MVC 现在已经成为了一个非常重要和流行的框架设计方法。下章介绍的 Struts 技术就是使用的 MVC 框架结构。而且使用 Servlet 过滤器技术来管理客

户端请求和服务器端响应之间的处理流，例如页面的定向。

过滤器可以在 MVC 框架中作为一个调度器组件，根据 xml 配置文件可以将客户端请求转发给相应的应用程序组件进行逻辑处理。过滤器也是成为 MVC 框架中的控制层的最佳选择。

通过使用过滤器作为 MVC 框架中的控制层，负责页面的转发以及安全检测等工作，这样仅仅需要通过 xml 配置文件就可以统一对页面转换进行管理。使得应用程序的结构清晰，易于后期程序的维护和管理。

MVC 体系结构已经非常流行，关于这部分的知识将在下面重点讲解。

20.2　Servlet事件监听器

在 Servlet 技术中已经定义了一些事件，并且我们可以针对这些事件来编写相关的事件监听器，从而对事件做出相应处理。Servlet 事件主要有三类：Servlet 上下文事件、会话事件与请求事件。下面具体讲解这三类事件的监听器实现：

（1）对 Servlet 上下文进行监听

可以监听 ServletContext 对象的创建和删除以及属性的添加、删除和修改等操作，该监听器需要使用到如下两个接口类：

- ServletContextAttributeListener：监听对 ServletContext 属性的操作，比如增加/删除/修改
- ServletContextListener：监听 ServletContext, 当创建 ServletContext 时，激发 contextInitialized(ServletContextEvent sce) 方法；当销毁 ServletContext 时，激发 contextDestroyed(ServletContextEvent sce)方法。

（2）监听 Http 会话

可以监听 Http 会话活动情况、Http 会话中属性设置情况，也可以监听 Http 会话的 active、paasivate 情况等，该监听器需要使用到如下多个接口类：

- HttpSessionListener：监听 HttpSession 的操作。当创建一个 Session 时，激发 session Created(SessionEvent se)方法；当销毁一个 Session 时，激发 sessionDestroyed (HttpSessionEvent se) 方法。
- HttpSessionActivationListener：用于监听 Http 会话 active、passivate 情况。
- HttpSessionAttributeListener：监听 HttpSession 中的属性的操作。当在 Session 增加一个属性时，激发 attributeAdded(HttpSessionBindingEvent se) 方法；当在 Session 删除一个属性时，激发 attributeRemoved(HttpSessionBindingEvent se)方法；当在 Session 属性被重新设置时，激发 attributeReplaced(HttpSessionBindingEvent se) 方法。

（3）对客户端请求进行监听

对客户端的请求进行监听是在 Servlet2.4 规范中新添加的一项技术，使用的接口类如下：

- ServletRequestListener 接口类
- ServletRequestAttrubuteListener 接口类

20.2.1　Servlet上下文监听器实例

范例：下面编写一个实例，使它能够对 ServletContext 以及属性进行监听。由以上介绍可以，该类需要实现 ServletContextAttributeListener 和 ServletContextListener 接口类，其详细代码如下：

```
package servlet;
```

```
import java.io.FileOutputStream;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
public class MyServletContextListener
                implements ServletContextListener,ServletContextAttributeListener{
    private ServletContext context = null;                    //定义一个 ServletContext 对象变量，赋为 null
    public void contextInitialized(ServletContextEvent s) {
        // TODO 该方法实现了 ServletContextListener 接口定义的方法，对 ServletContext 进行初始化
        this.context = s.getServletContext();                 //初始化一个 ServletContext 对象
        print("ServletContext 初始化.....");                  //打印出该方法的操作信息
    }
    public void contextDestroyed(ServletContextEvent s) {
        // TODO 该方法实现了 ServletContextListener 接口类定义方法，用于释放 ServletContext 对象
        this.context = null;
        print("ServletContext 被释放.....");
    }
    public void attributeAdded(ServletContextAttributeEvent sa) {
        // TODO 当上下文添加属性时，将调用该方法。这里只是将添加的属性信息打印出来
        print("增加 ServletContext 对象的一个属性：attributeAdded('"+sa.getName()+"','"+sa.getValue()+"')");
    }
    public void attributeRemoved(ServletContextAttributeEvent sa) {
        // TODO 当把 ServletContext 中的某个属性删除时，调用该方法
        print("删除 ServletContext 对象的某一个属性：attributeRemoved('
            "+sa.getName()+"','')");
    }
    public void attributeReplaced(ServletContextAttributeEvent sa) {
        // TODO 当上下文进行属性值更新时，将调用该方法
        print("更改 ServletContext 对象的某一个属性：attributeReplaced('
            "+sa.getName()+"','"+sa.getValue()+"')");
    }
    private void print(String message){
        // 调用该方法在 txt 文件中打印出 message 字符串信息
        PrintWriter out = null;
        try{
            out = new PrintWriter(new FileOutputStream("D:\\output.txt",true));
            out.println(new java.util.Date().toLocaleString()+" ContextListener: "+message);
            out.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

程序说明：该监听器类实现了 ServletContextAttributeListener 和 ServletContextListener 两个接口类中的五个方法：

- contextInitialized(ServletContextEvent s)方法用来初始化 ServletContext 对象。
- contextDestroyed(ServletContextEvent s)方法在上下文中删除某个属性的时候会调用。
- attributeAdded(ServletContextAttributeEvent sa)方法在上下文中添加一个新的属性时调用。
- attributeReplaced(ServletContextAttributeEvent sa)方法在更新属性的时候调用
- attributeRemoved(ServletContextAttributeEvent sa)方法在上下文中删除某个属性的时候会被调用。

在使用这个监听器之前还需要在 Web 模块中的 web.xml 配置文件中进行声明，代码如下：

```
<listener>
  <listener-class>servlet.MyServletContextListener</listener-class>
</listener>
```

接下来就编写 JSP 程序来操作 ServletContext 的属性，看监听器程序做出什么反应，该 JSP 的一段代码如下：

```
<%
  out.println("Test ServletContextListener");
  application.setAttribute("userid","zzb");          //添加一个属性
  application.setAttribute("userid","zzb2");         //替换掉已经添加的属性
  application.removeAttribute("userid");             //删除该属性
%>
```

代码说明：当第一次添加属性 userid 时，监听器调用 contextInitialized(ServletContextEvent s)初始化监听方法和 attributeAdded(ServletContextAttributeEvent sa)添加属性监听方法。

可以查看 D 根目录下的 output.txt 文件内容，如下：

```
2006-7-12 14:07:50 ContextListener: ServletContext 初始化.....
2006-7-12 14:13:55 ContextListener: 增加 ServletContext 对象的一个属性：attributeAdded('userid','zzb')
2006-7-12 14:13:55 ContextListener: 更改 ServletContext 对象的某一个属性：attributeReplaced('userid','zzb2')
2006-7-12 14:13:55 ContextListener: 删除 ServletContext 对象的某一个属性：attributeRemoved('userid')
```

该 log 文件记录了监听器所做的动作。

20.2.2　Http会话监听器实例

通过上一个监听器实例，读者应该对监听器的实现过程有所了解，这一小节将要介绍基于 Http 会话的监听器。

首先创建监听器类 MySessionListener.java，其源代码如下：

```
package servlet;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.http.HttpSessionActivationListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
public class MySessionListener
        implements HttpSessionActivationListener,HttpSessionAttributeListener,
                HttpSessionListener,ServletContextListener{
```

```
ServletContext context = null;
 int users = 0;
 public void sessionWillPassivate(HttpSessionEvent arg0) {
        // 监听 Http 会话的 passivate 情况
        print("sessionWillPassivate("+arg0.getSession().getId()+")");
 }
 public void sessionDidActivate(HttpSessionEvent arg0) {
        // 监听 Http 会话的 active 情况
        print("sessionDidActivate("+arg0.getSession().getId()+")");
 }
 public void attributeAdded(HttpSessionBindingEvent arg0) {
        // 监听 Http 会话中的属性添加
        print("attributeAdded('"+arg0.getSession().getId()+"','"+arg0.getName()+"','"+arg0.getValue()+"')");
 }
 public void attributeRemoved(HttpSessionBindingEvent arg0) {
        // 监听 Http 会话中属性删除
        print("attributeRemoved('"+arg0.getSession().getId()+"','"+arg0.getName()+"','"+arg0.getValue()+"')");
 }
 public void attributeReplaced(HttpSessionBindingEvent arg0) {
        // 监听 Http 会话中的属性更改操作
        print("attributeReplaced('"+arg0.getSession().getId()+"','"+arg0.getName()+"','"+arg0.getValue()+"')");
 }
 public void sessionCreated(HttpSessionEvent arg0) {
        // Http 会话的创建监听
        users++;                                                    //创建一个会话，把 users 变量加一
        print("sessionCreated('"+arg0.getSession().getId()+"'),目前拥有"+users+"个用户");
        context.setAttribute("users",new Integer(users));       //把会话数设置到 ServletContext 的属性 users 中
 }
 public void sessionDestroyed(HttpSessionEvent arg0) {
        // Http 会话的释放监听
        users--;                                                    //释放一个会话，把 users 变量减一
        print("sessionDestroyed('"+arg0.getSession().getId()+"'),目前拥有"+users+"个用户");
        context.setAttribute("users",new Integer(users));       //把会话数设置到 ServletContext 的属性 users 中
 }
 public void contextInitialized(ServletContextEvent arg0) {
        // 该方法实现了 ServletContextListener 接口定义的方法，对 ServletContext 进行初始化
        this.context = arg0.getServletContext();                    //初始化 ServletContext 对象
        print("ServletContext 初始化.....");                        //打印出该方法的操作信息
 }
 public void contextDestroyed(ServletContextEvent arg0) {
        // 监听 Servlet 上下文被释放
        this.context = null;                                        //释放 ServletContext 对象
        print("ServletContext 被释放.....");                        //打印出该方法的操作信息
 }
 private void print(String message){
        // 调用该方法在 txt 文件中打印出 message 字符串信息
        PrintWriter out = null;
        try{
              out = new PrintWriter(new FileOutputStream("d:\\output.txt",true));
              out.println(new java.util.Date().toLocaleString()+" SessionListener: "+message);
```

```
                out.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
}
```

程序说明：

（1）该程序实现了 HttpSessionListener 接口类中的两个方法：

- sessionCreated(HttpSessionEvent arg0)方法进行 Http 会话创建的监听，如果 Http 会话被创建将调用该方法。
- sessionDestroyed(HttpSessionEvent arg0)方法对 Http 会话销毁进行监听，如果某个 Http 会话被释放将调用该方法。

（2）实现 HttpSessionActivationListener 接口类中的如下两个方法：

- sessionDidActivate(HttpSessionEvent arg0)方法对 Http 会话处于 active 情况进行监听。
- sessionWillPassivate(HttpSessionEvent arg0)方法对 Http 会话处于 passivate 情况进行监听。

（3）实现 HttpSessionAttributeListener 接口类中的如下三种方法：

- attributeAdded(HttpSessionBindingEvent arg0)方法对 Http 会话中属性添加进行监听。
- attributeReplaced(HttpSessionBindingEvent arg0)方法对 Http 会话中属性修改进行监听。
- attributeRemoved(HttpSessionBindingEvent arg0)方法对 Http 会话中属性删除进行监听。

（4）另外，该程序中还实现了 ServletContextListener 接口类，该类的实现方法已经在上一小节中所有介绍。

同样需要在 web.xml 配置文件进行该监听器的声明，配置方法如上一节介绍。该监听器实现了在线会话人数的统计，当一个会话创建的时候，users 变量将加一；当销毁一个会话对象的时候，users 变量将减一。

下面创建测试的 JSP 页面程序，代码如下：

```
<%
  out.println("Test SessionListener");
  session.setAttribute("username","zzb1");          //在 Http 会话中设置一个用户 username 属性
  session.setAttribute("username","zzb2");          //修改之上添加的 username 属性
  session.removeAttribute("username");              //删除创建的 username 属性
  session.invalidate();                             //passivate Http 会话
%>
```

执行效果可以查看 output.txt 文档内容，如下：

```
2006-7-12 15:24:47 SessionListener: ServletContext 初始化.....
2006-7-12 15:25:47 SessionListener: sessionCreated('720C4654847ECE025DF6A8AF09117212'),目前拥有 1 个用户
2006-7-12 15:25:47 SessionListener:
attributeAdded('720C4654847ECE025DF6A8AF09117212','username','zzb1')
2006-7-12 15:25:47 SessionListener:
attributeReplaced('720C4654847ECE025DF6A8AF09117212','username','zzb1')
2006-7-12 15:25:47 SessionListener:
attributeRemoved('720C4654847ECE025DF6A8AF09117212','username','zzb2')
2006-7-12 15:25:47 SessionListener: sessionDestroyed('720C4654847ECE025DF6A8AF09117212'),目前拥有 0 个用户
```

20.2.3　客户端请求监听器实例

对客户端请求进行监听的技术是在 Servlet2.4 版本之后才出现的。一旦监听程序能够获得客户端请求，就可以对所有客户端请求进行统一处理。例如一个 Web 程序，如果在本机访问，就可以不登录，如果是远程访问，即需要登录。这是通过监听客户端请求，从请求中获取到客户端地址，并通过地址来做出相应的处理。

实现客户端请求监听的程序代码如下：

```
package servlet;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import javax.servlet.ServletRequest;
import javax.servlet.ServletRequestAttributeEvent;
import javax.servlet.ServletRequestAttributeListener;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
public class MyRequestListener implements
                ServletRequestListener,ServletRequestAttributeListener{
    public void requestDestroyed(ServletRequestEvent arg0) {
        // 对销毁客户端请求进行监听
        print("reqeust destroyed");
    }
    public void requestInitialized(ServletRequestEvent arg0) {
        // 对实现客户端请求进行监听
        print("Request init");
        ServletRequest sr = arg0.getServletRequest();              //初始化客户端请求
        print(sr.getRemoteAddr());                                 //获得请求客户端的地址
    }
    public void attributeAdded(ServletRequestAttributeEvent arg0) {
        // 对属性的添加进行监听
        print("attributeAdded('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    public void attributeRemoved(ServletRequestAttributeEvent arg0) {
        // 对属性的删除进行监听
        print("attributeRemoved('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    public void attributeReplaced(ServletRequestAttributeEvent arg0) {
        // 对属性的更新进行监听
        print("attributeReplaced('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    private void print(String message){
        // 调用该方法在 txt 文件中打印出 message 字符串信息
        PrintWriter out = null;
        try{
            out = new PrintWriter(new FileOutputStream("d:\\output.txt",true));
            out.println(new java.util.Date().toLocaleString()+" RequestListener: "+message);
            out.close();
        }
        catch(Exception e)
```

```
            {
                e.printStackTrace();
            }
        }
}
```

程序说明：该监听器类实现了 ServletRequestListener 和 ServletRequestAttributeListener 两接口类，ServletRequestListener 接口类中定义的两个方法对客户端请求的创建和销毁进行监听；ServletRequestAttributeListener 接口类对请求中的属性添加、修改和删除进行监听。

下面编写 JSP 程序进行测试，代码如下：

```
<%
   out.println("Test RequestListener");
   request.setAttribute("username","zzb1");          //在请求中设置一个用户 username 属性
   request.setAttribute("username","zzb2");          //修改之上添加的 username 属性
   request.removeAttribute("username");              //删除创建的 username 属性
%>
```

注意：Weblogic Server8.1 **还没有能支持** Servlet2.4 **规范，所以也不能支持客户端请求的监听。**

20.3　本章小结

虽然过滤器才出现几年时间，但它们本身已作为一个关键组件嵌入到了所有敏捷的、面向对象的 J2EE Web 应用程序中。本文首先向读者介绍了 Servlet 过滤器的使用。讨论了实现过滤器所涉及的精确步骤，以及如何在 Web 应用程序中声明过滤器，然后与应用程序一起部署它。本文还阐述了 Servlet 的监听器的实现，以及一些最普遍应用，其中客户端请求监听是从 Servlet2.4 开发才加入到规范中的。