# 1. 观察者设计模式（监听器）

在学习监听器前，我们先学习观察者设计模式，因为监听器其实就是使用了观察者设计模式。

## 1.1 回顾

我们从之前学习的例子说起。编写GUI程序时遇到的第一个问题就是无法关闭窗体，我们是通过处理windowClosing窗体关闭事件解决的：

|  |
| --- |
| **package** com.demo;  **import** java.awt.\*; **import** java.awt.event.WindowAdapter; **import** java.awt.event.WindowEvent;  **public class** Main {  **public static void** main(String[] args) {  Frame frame = **new** Frame(**"标题"**);  frame.setBounds(500, 200, 400, 300);  frame.setVisible(**true**);  *// 添加事件监听* frame.addWindowListener(**new** WindowAdapter() {  @Override  **public void** windowClosing(WindowEvent e) {  *// 拿到事件源对象，dispose窗体* Frame f = (Frame) e.getSource();  f.dispose();  }  });  } } |

代码中，通过addWindowListener添加对事件的监听，这样当用户点击关闭按钮时，就会执行windowClosing事件处理的代码。

实际上事件的处理就是使用了观察者设计模式，即被观察者（窗体）一旦有了某操作（例如执行了某方法，或这里的被观察者窗体被点击关闭按钮了），就会被观察者（监听器）发现，进而执行对应的操作。

上述处理中有这样几个概念：

（1）frame对象：这是事件源，即发生事件的对象。其是被监听的目标。

（2）WindowListener（默认适配器是WindowAdapter）：这就是监听器，监听器中会对应者事件源中的相关内容。

（3）WindowEvent：这是事件对象，其中封装了事件源等数据，便于调用者拿到使用。因此WindowEvent对象有一个getSource()方法，就是用于拿到事件源对象的。

（4）addWindowListener(WindowListener listener)方法用于注册监听器。如果没有注册的话，是无任何反应的。

## 1.2 观察者设计模式案例

上述都是使用别人设计好的监听器，是比较简单的，为了清晰原理，现在自己编写观察者设计模式的示例代码，这样能更好地理解是如何实现的。

观察者模式，即能实现监听被观察者对象：当被观察者执行了某操作时，如果该操作被监听器注册了，那么监听器中编写的处理方法也将执行。

我们要实现的案例：监听学生的学习和睡觉行为，分别在学习前和睡觉前提供一杯牛奶。我们参照前面事件处理的方法编写代码。

（1）首先定义一个Student类，提供study和sleep方法：

|  |
| --- |
| **package** com.demo;  **public class** Student {  **private** String **name**;   **public** Student(String name) {  **this**.**name** = name;  }   **public void** study() {  System.***out***.println(**name** + **"开始学习"**);  }   **public void** sleep() {  System.***out***.println(**name** + **"开始睡觉"**);  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

（2）编写StudentListener学生监听器接口，声明可监听学习前和睡觉前动作。参照事件处理代码，还要在接口方法中传递一个事件对象StudentEvent。事件对象用于传递事件源，StudentEvent类下面编写。因此这里的StudentListener接口写为：

|  |
| --- |
| **package** com.demo;  **public interface** StudentListener {  *// 监听学习前* **void** beforeStudy(StudentEvent e);  *// 监听睡觉后* **void** beforeSleep(StudentEvent e); } |

（3）编写事件对象StudentEvent，其中封装事件源对象，且这个对象通过构造函数传递进来：

|  |
| --- |
| **package** com.demo;  **public class** StudentEvent {  **private** Object **source**; *// 维护事件源对象* **public** StudentEvent(Object source) {  **this**.**source** = source;  }  *// 提供getSource方法* **public** Object getSource() {  **return this**.**source**;  } } |

（4）最后，还需要改进Student代码，使得Student能够被监听。这里给Student添加一个addStudentListener方法，方法传入StudentListener对象，由该类维护，然后，在每个要监听的行为中判断是否有监听器注册，否则有，就执行监听器中的方法。因此代码如下：

|  |
| --- |
| **package** com.demo;  **public class** Student {  **private** String **name**;  **private** StudentListener **listener**; *// 维护listener* **public** Student(String name) {  **this**.**name** = name;  }   **public void** study() {  **if** (**listener** != **null**) {  **listener**.beforeStudy(**new** StudentEvent(**this**));  }  System.***out***.println(**name** + **"开始学习"**);  }   **public void** sleep() {  **if** (**listener** != **null**) {  **listener**.beforeSleep(**new** StudentEvent(**this**));  }  System.***out***.println(**name** + **"开始睡觉"**);  }   *// 提供addStudentListener方法* **public void** addStudentListener(StudentListener listener) {  **this**.**listener** = listener;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

其实addStudentListener也是使用了策略设计模式。现在我们执行如下程序：

|  |
| --- |
| **package** com.demo;  **public class** Main {  **public static void** main(String[] args) {  Student student = **new** Student(**"张三"**);  *// 添加监听* student.addStudentListener(**new** StudentListener() {  @Override  **public void** beforeStudy(StudentEvent e) {  Student obj = (Student) e.getSource();  System.***out***.println(obj.getName() + **"，给你一杯牛奶"**);  }   @Override  **public void** beforeSleep(StudentEvent e) {  Student obj = (Student) e.getSource();  System.***out***.println(obj.getName() + **"，给你一杯牛奶"**);  }  });  *// 学生执行了行为* student.study();  student.sleep();   } } |

就会发现能够监听学生的行为。当然，如果不通过addStudentListener注册监听，那么调用方法时就像什么都没有发生一样。

# 2. 监听器(Listener)

## 2.1 监听器概述和使用

监听器用来监听服务器中特定对象的创建、销毁或者属性发生变化等的情况。Servlet中提供的监听器一般用来监听服务器维护的对象（例如servletContext、session等），这样能够让开发者在特定的时机执行某些自定义操作。

Servlet规范中一共提供了8个监听器，下面分类进行介绍。

### 2.1.1 生命周期监听器

Servlet提供了三个生命周期监听器，分别是ServletContextListener、HttpSessionListener和ServletRequestListener监听器接口，它们分别用于监听servletContext、session和request对象的创建和销毁。

上述三个接口中都提供了对象创建和销毁事件发生时要调用的方法。开发者可自行写入各个时机需要执行的方法。

编写监听器时，开发者只要自定义类实现监听器接口即可。

例子：监听request对象的创建和销毁。分别在request对象创建和销毁时向控制台输出一行信息。

监听器类代码（使用注解）：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.ServletRequestEvent; **import** javax.servlet.ServletRequestListener; **import** javax.servlet.annotation.WebListener;  **public class** MyRequestListener **implements** ServletRequestListener {  @Override  **public void** requestInitialized(ServletRequestEvent sre) {  *// 通过sre参数能拿到request对象。此时request对象刚创建，是拿不到域对象中值的* String name = (String) sre.getServletRequest().getAttribute(**"name"**);  System.***out***.println(**"获得的name属性："** + name);  System.***out***.println(**"request对象创建"**);  }   @Override  **public void** requestDestroyed(ServletRequestEvent sre) {  *// 这里就能拿到与对象中的值* String name = (String) sre.getServletRequest().getAttribute(**"name"**);  System.***out***.println(**"获得的name属性："** + name);  System.***out***.println(**"request对象销毁..."**);  } } |

上述代码测试了能否成功执行操作并拿到request和域对象。这些事件对象sre等中也提供了getSource()方法用于拿到事件源对象，但是需要强转，为了方便，使用getServletRequest等方法就能直接拿到事件源对象。

当然，为了使监听器生效，需要在web.xml中注册该监听器，只要在listener-class节点中配置该监听器即可。例如：

|  |
| --- |
| <**listener**>  <**listener-class**>com.listener.demo.MyRequestListener</**listener-class**> </**listener**> |

为了简单，也可在类上使用@WebListener注解来配置监听器。

为了成功测试代码，我们在index.jsp中写入以下内容：

|  |
| --- |
| <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%** request.setAttribute(**"name"**, **"张三"**); **%>** 首页 </**body**> </**html**> |

这样，我们每访问一次index.jsp页面，控制台上就会产生如下信息：

|  |
| --- |
| 获得的name属性：null  request对象创建  获得的name属性：张三  request对象销毁... |

因为JSP页面也是一个Servlet，也会产生request对象。

使用ServletContextListener和HttpSessionListener也是类似的步骤，要注意的是servletContext和session对象创建和销毁的时机。

servletContext对象仅会在服务器启动时创建一次，最后在服务器正常关闭时销毁。因此，ServletContextListener的应用场景是完成系统的初始化工作。例如我们以后要学习的Spring框架，可用于管理应用中使用的各种对象（例如DAO、Service对象等）。在Web应用中，Spring框架就是通过配置监听器来完成对象的初始化工作的，Spring的监听器ContextLoaderListener就实现了ServletContextListener接口。Spring使用容器存放应用需要使用的对象，可以把容器理解为一个Map集合，键是对象的唯一名称，值是对象本身。这个以后接触到Spring框架后才有深刻的理解。

而session对象是在服务器第一次调用getSession方法时创建，服务器为每个客户端都维护一个session对象，因此某客户端第一次访问应用时，通过getSession拿不到session对象就会创建属于该客户端的session对象（访问JSP时默认就会调用getSession，因为JSP的page指令中，session属性值默认为true，即默认就会获得session）。session对象何时被销毁呢？session对象将在session过期后销毁（Tomcat默认过期时间是30分钟，可通过配置文件配置），当然调用session.invalidate()方法将主动销毁session对象。

需要注意的是，浏览器关闭时，session对象并未被销毁，因为session的原理是服务器端拿到浏览器的SESSIONID，再查找服务器中是否有匹配的session对象。而浏览器关闭了再打开，浏览器中的SESSIONID这个Cookie过期失效了，因此导致服务器无法拿到之前的session对象，但这只说明之前的session无法再被使用（即失效了），但该对象并不是在此时被销毁，而是等到该session过期后（例如默认的30分钟）被销毁，此时，服务器会为该客户端新创建一个session对象，即此时会话会使用新的session对象，之前的session对象由服务器自行负责销毁。

根据session的这个特点，可使用HttpSessionListener监听器来粗略统计网站的访问量，一种一个session对象就代表着一个客户端。

### 2.1.2 属性监听器

属性监听器用于监听servletContext、session和request中属性的变化，即调用对象的setAttribute()方法时发生的属性（域）的变化情况。它们对应的监听器接口分别是ServletContextAttributeListener、HttpSessionAttributeListener和ServletRequestAttributeListener。

上述三个接口中都提供了增加属性、移除属性和改变属性状态时要调用的方法。开发者可自行写入各个时机需要执行的方法。

属性监听器的使用方法和上述生命周期监听器的使用方法类似。

用HttpSessionAttributeListener监听属性状态变化。代码如下：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.http.HttpSessionAttributeListener; **import** javax.servlet.http.HttpSessionBindingEvent;  **public class** MySessionAttrListener **implements** HttpSessionAttributeListener {   @Override  **public void** attributeAdded(HttpSessionBindingEvent se) {  *// 向session中添加属性触发的事件  // 可直接拿到属性名和属性值* System.***out***.println(**"添加属性名为："** + se.getName());  System.***out***.println(**"添加属性值为："** + se.getValue());  }   @Override  **public void** attributeReplaced(HttpSessionBindingEvent se) {  System.***out***.println(**"重新设置属性名为："** + se.getName());  System.***out***.println(**"重新设置的属性值之前为："** + se.getValue());  *// 通过session拿到现在的值，其中也使用到se.getName()。* System.***out***.println(**"重新设置的属性值现在为："** + se.getSession().getAttribute(se.getName()));   }   @Override  **public void** attributeRemoved(HttpSessionBindingEvent se) {  System.***out***.println(**"移除的属性名为："** + se.getName());  System.***out***.println(**"移除的属性名值："** + se.getValue());  } } |

这里使用web.xml配置监听器：

|  |
| --- |
| <**listener**>  <**listener-class**>com.listener.demo.MySessionAttrListener</**listener-class**> </**listener**> |

在index.jsp中写：

|  |
| --- |
| <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%**  session.setAttribute(**"age"**, **"12"**);  session.setAttribute(**"age"**, **"20"**);  session.removeAttribute(**"age"**); **%>** 首页 </**body**> </**html**> |

这时访问JSP页面，控制台输出的信息为：

|  |
| --- |
| 添加属性名为：age  添加属性值为：12  重新设置属性名为：age  重新设置的属性值之前为：12  重新设置的属性值现在为：20  移除的属性名为：age  移除的属性名值：20 |

### 2.1.3 session相关的感知型监听器

Servlet还提供了Session相关的感知型监听器：

（1）HttpSessionBidingListener：实现该监听器接口的类的实例，能够感知自己何时被加入了session域对象中，和何时被session域对象删除。即能监听本对象被绑定（或解绑）到session上的事件。

（2）HttpSessionActivationListener：实现该接口的类的实例，能够感知自己何时随着session对象被钝化和激活。要求该类还需要实现序列化接口（这是session钝化和激活要求的）。

注意的是，感知型监听器类是不需要进行注册的（即不需要通过web.xml或者@WebListener注册），自动能“感知”到自己的事件，即非服务器对象无需注册“监听”。

例1：监听Employee类被session绑定/解绑事件。

只要自定义类（一般是实体类）实现HttpSessionBindingListener即可：

|  |
| --- |
| **package** com.listener.demo;  **import** javax.servlet.http.HttpSessionBindingEvent; **import** javax.servlet.http.HttpSessionBindingListener;  **public class** Employee **implements** HttpSessionBindingListener { **private** String **name**; *// 姓名* **public** Employee(String name) {  **this**.**name** = name;  }   @Override  **public void** valueBound(HttpSessionBindingEvent se) {  *// 该类对象绑定到session触发的事件* System.***out***.println(**"绑定了"** + se.getName());  }   @Override  **public void** valueUnbound(HttpSessionBindingEvent se) {  System.***out***.println(**"解绑了"** + se.getName());   }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

index.jsp代码如下：

|  |
| --- |
| <%@**page import**="**com.listener.demo.Employee**"%> <%@ **page language**="**java**" **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**/>  <**title**>欢迎</**title**> </**head**> <**body**> **<%** session.setAttribute(**"emp"**, **new** Employee(**"张三"**)); *// 可触发绑定事件* session.removeAttribute(**"emp"**); *// 可触发解绑事件* **%>** 首页 </**body**> </**html**> |

运行时就会触发事件。即只要我们的自定义类对象绑定/解绑在了session中，就会触发相应事件。

例2：演示Student类被钝化和激活的。

类只要实现序列化和HttpSessionActivationListener接口即可：

|  |
| --- |
| **package** br.bean;  **import** javax.servlet.http.HttpSessionActivationListener; **import** javax.servlet.http.HttpSessionEvent; **import** java.io.Serializable;  **public class** Student **implements** Serializable, HttpSessionActivationListener {  @Override  **public void** sessionWillPassivate(HttpSessionEvent hse) {  *// 钝化的事件* System.***out***.println(**"Student对象被钝化了"**);  }   @Override  **public void** sessionDidActivate(HttpSessionEvent hse) {  *// 激活的事件* System.***out***.println(**"Student对象被激活了"**);  } } |

然后使用在session中使用一下Student类，以便演示钝化和激活：

|  |
| --- |
| <%@ **page import**="**br.bean.Student**" %> <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> **<%** session.setAttribute(**"name"**, **new** Student()); **%>** </**body**> </**html**> |

此时，使用shutdown命令正常关闭Tomcat，则触发钝化事件，随后，重新启动Tomcat，则会触发激活事件。

# 3. Servlet 3.0新特性

之前我们基本上学习的都是Servlet 2.5规范的内容，在Servlet 3.0中，新增了一些功能，如下小节所述。

## 3.1 使用注解

在之前的学习中我们也提到过可以使用注解的方式配置Servlet、监听器、过滤器等，这里不再赘述。使用注解实际就是Servlet 3.0新增的功能，常用的注解有：

（1）@WebServlet，其中可通过urlPatterns属性（或value属性）配置Servlet映射路径；可通过initParams属性（类型是@WebInitParam注解）配置Servlet的参数等。

（2）@WebFilter注解可配置过滤器；

（3）@WebListener注解可配置监听器。

在Servlet3.0中，Web应用是可以没有web.xml文件的，但一般还是会将注解和XML文件结合使用，因为单用注解会导致硬编码问题，且过滤器无法配置执行顺序。

## 3.2 文件上传

Servlet 3.0中加入了文件上传功能，即实现Servlet 3.0规范的Web服务器“自带”文件上传的功能。Tomcat按照Servlet 3.0规范集成的文件上传功能，实际也是借用了FileUpload组件。如果我们使用Servlet规范的文件上传功能，则无需引入FileUpload组件包。

下面介绍如何使用Servlet 3.0的文件上传功能。

Servlet 3.0中提供了一个Part接口，它代表了文件上传表单中的每一项，Part可以理解为是FileUpload组件中的FileItem。如果要获得文件上传字段的值，则可使用request.getPart(表单中参数名)，如果要获得普通字段的值，还是使用request.getParameter(String name)即可。另外要注意的是，需要在处理文件上传表单的Servlet类上使用@MultipartConfig注解，这样Servet才会知道自己处理的是文件上传表单，上述方法才会有效。

在Part对象中，提供了getName()【获得表单参数名】、getSubmittedFileName()【获得文件名】、getHeader()【获取这一项的头】、getInputStream()【获得上传文件输入流】、write()【写入文件】和delete()【删除临时文件】等方法，供开发者进行使用。基本用法和FileUpload类似。

示例代码：

|  |
| --- |
| **package** br.action;  **import** javax.servlet.ServletException; **import** javax.servlet.annotation.MultipartConfig; **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** javax.servlet.http.Part; **import** java.io.IOException; **import** java.util.Collection; **import** java.util.Iterator;  *// 可以用MultipartConfig注解的属性控制上传文件的大小 // maxFileSize控制单个上传文件的大小，这里控制为4M。 // maxRequestSize控制整个请求上传数据的大小，这里控制为5M。* @MultipartConfig(maxFileSize = 1024 \* 1024 \* 4, maxRequestSize = 1024 \* 1024 \* 5) @WebServlet(**"/fileUploadServlet"**) **public class** FileUploadServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**); *// 设置请求编码，解决上传文件名乱码。* String name = req.getParameter(**"name"**); *// 获得普通参数值* System.***out***.println(name);  Collection<Part> parts = **null**;  **try** {  *// 如果一个表单参数要上传多个文件（例如上传多个参数为photo的照片），则不能用getPart(name)，需要使用getParts()再遍历* parts = req.getParts();  } **catch** (Exception e) {  *// 前面限制了文件上传大小，这里要通过异常判断出错类型。实际上出错类型还是我们之前在FileUpload中使用的类型  // 但是这里无法捕获FileUploadBase$FileSizeLimitExceededException这些异常类型，因此只能查看异常message中是否包含这些信息* **if** (e.getMessage().contains(**"FileUploadBase$FileSizeLimitExceededException"**)) {  *// 说明是单个文件上传大小超过限制* resp.getWriter().write(**"单个文件上传大小不能超过4M"**);  } **else if** (e.getMessage().contains(**"FileUploadBase$SizeLimitExceededException"**)) {  *// 说明是上传请求数据总大小超过限制* resp.getWriter().write(**"总文件上传大小不能超过5M"**);  } **else** {  e.printStackTrace();  }  }  **if** (parts != **null**) {  *// 遍历parts* **for** (Iterator<Part> it = parts.iterator(); it.hasNext(); ) {  Part part = it.next();  System.***out***.println(part.getName()); *// 获得字段参数名* String fileName = part.getSubmittedFileName(); *// 获得上传文件名* System.***out***.println(fileName);  **if** (fileName != **null** && !**""**.equals(fileName)) {  part.write(**"d:/"** + fileName); *// 写入硬盘* }  part.delete(); *// 确保删除。Tomcat的API中说，Tomcat会自动删除临时文件，但为了确保删除，可以手动调用delete方法。* }  *// 遍历结束* }  } } |

上述代码中处理了上传文件大小限制、乱码等问题，可做参考。

至于@MultipartConfig这些注解的功能是Tomcat服务器通过反射去完成的，开发者会使用即可。

## 3.3 Cookie的HttpOnly属性

普通的Cookie是可以被浏览器JavaScript脚本读取的，例如：

|  |
| --- |
| console.log(document.cookie); |

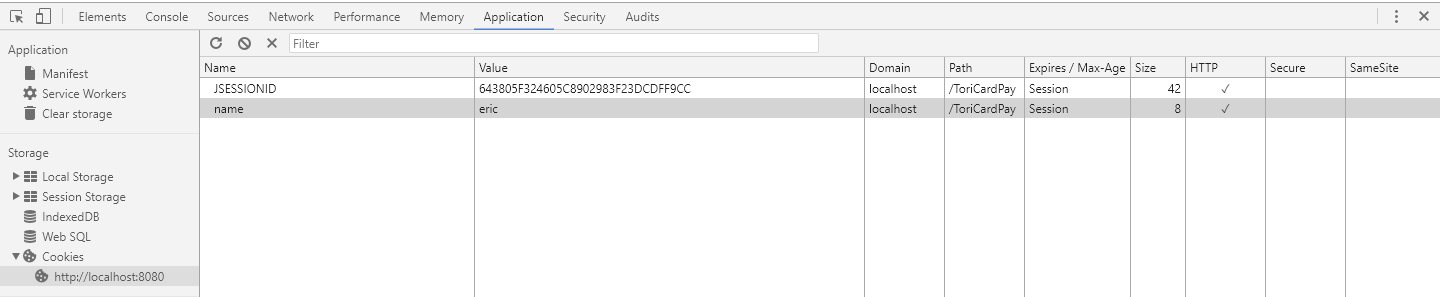
执行上面的语句就会把当前网站的所有Cookie信息列出来。其实允许脚本访问cookie是不安全的，例如JS脚本注入攻击能够窃取用户cookie，进而伪装成用户访问网站，可能造成安全问题导致损失。

为了安全性，可以为cookie设置HttpOnly属性，如果该属性设置为true，则该cookie是无法被脚本所访问的，但当服务器访问时可正常发送cookie数据。

Servlet3.0中新增了API以设置Cookie的HttpOnly属性，例如：

|  |
| --- |
| @Override **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  Cookie cookie = **new** Cookie(**"name"**, **"eric"**);  cookie.setHttpOnly(**true**); // 设置HttpOnly属性为true  resp.addCookie(cookie); } |

这样浏览器使用document.cookie是无法获得name=eric这个cookie值的。在Chrome的开发者工具中，可以看到该Cookie的“HTTP”属性一栏被打钩，说明该cookie的HttpOnly属性为true：



## 3.4 WebFragment片段

Servlet3.0提供了WebFragment，WebFragment相当于是项目的插件，只要把WebFragment片段生成的jar包引入到项目中，就能使用该片段的功能。

例如resources文件夹下提供了myfragment.jar这个包，其中META-INF目录下的web-fragment.xml内容如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-fragment  xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  version="3.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-fragment\_3\_0.xsd"  metadata-complete="true">  <servlet>  <servlet-name>CentrollerServlet</servlet-name>  <servlet-class>com.itheima.servlet.CentrollerServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>CentrollerServlet</servlet-name>  <url-pattern>/action/\*</url-pattern>  </servlet-mapping>  </web-fragment> |

其中其实就是配置了一个路径为/action/\*的Servlet。如果将该jar包放入项目中，那么应用启动后，直接访问该路径就可访问该Servlet，无需其他配置。