# 1. 过滤器(Filter)

## 1.1 过滤器简介

开发Web项目时，经常需要对用户的请求进行重复的操作，比如：判断用户是否已经登录（登录了才能展示相关页面）、设置请求的编码格式为UTF-8。

对每个请求这样重复写代码很麻烦，不利于管理。我们想对一些请求进行统一的过滤管理，这时就可考虑使用过滤器。

对请求使用了过滤器后，那么代码的执行流程就为：

（1）用户请求Servlet或JSP资源，该请求会先进入过滤器（即实现了对Servlet/JSP请求的拦截），过滤器中可对请求进行预处理。

（2）过滤器处理完毕后，放行该请求，这时请求才会到达Servlet或JSP。

（3）Servlet或JSP进行自己的处理，完毕后，这次的请求和响应还会返回到过滤器，最后由服务器响应给用户。

## 1.2 过滤器的使用

过滤器的开发步骤如下：

（1）新建一个类，实现Filter接口。该类就是过滤器类。Filter接口中有如下方法：

A：void init(FilterConfig filterConfig)：服务器启动时会执行init方法，可在init()里面写初始化内容。

方法提供了一个filterConfig对象参数，该对象可用于获取过滤器的初始化参数（即在web.xml中也可对过滤器配置参数，就像对Servlet配置参数，使用ServletConfig能拿到参数一样），FilterConfig接口提供了下面方法：

String getInitParameter(String name) ：获得指定名称的参数值

Enumeration getInitParameterNames()：获得所有参数名称的集合。

上述的用法还是和以前的一样。

B：void doFilter(request, response, FilterChain filterChain)：过滤器拦截的处理方法，传递了响应和请求的参数。主要就是写这里面的方法来预处理请求和响应数据。

另外方法参数还提供了一个filterChain对象，该对象用于将当前的请求放行，这样请求才能进入到Servlet/JSP或者下一个过滤器。（因为下面我们会在web.xml中配置过滤器，可在一个应用中配置多个过滤器。如有多个过滤器，则请求还会进入下一个过滤器。进入过滤器的顺序是根据web.xml中过滤器的配置顺序来的。）调用以下方法对当前请求放行：

void doFilter(ServletRequest request, ServletResponse response)

所有的过滤器形成了过滤器链FilterChain，因此在过滤器中必须放行，若不放行，就无法进行下去。

C：void destroy()：服务器关闭时会自动调用destroy()方法来销毁过滤器实例。

现在简单编写一个过滤器如下：

|  |
| --- |
| **public class** MyFilter **implements** Filter {  **public** MyFilter() {  System.***out***.println(**"创建了过滤器实例"**);  }    @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  System.***out***.println(**"初始化过滤器"**);  *// 可获得过滤器中配置的参数。比如配置了名叫encoding的参数* System.***out***.println(filterConfig.getInitParameter(**"encoding"**));  }   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain)  **throws** IOException, ServletException {  System.***out***.println(**"执行过滤器处理方法。"**);  *// 现在不做什么处理，直接放行* filterChain.doFilter(req, resp);  *// Servlet处理完成后，又会回来。* System.***out***.println(**"Servlet/JSP响应完毕。"**);  }    @Override  **public void** destroy() {  System.***out***.println(**"销毁了Filter实例"**);  } } |

（2）在web.xml中配置Filter：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"**>  *<!-- 在filter节点中配置过滤器 -->* <**filter**>  *<!-- Filter的名称 -->* <**filter-name**>myFilter</**filter-name**>  *<!-- 指定过滤器类 -->* <**filter-class**>com.zhang.test.MyFilter</**filter-class**>  *<!-- 配置Filter的初始化参数。可设置多个，即有多个init-param标签 -->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>UTF-8</**param-value**>  </**init-param**>  </**filter**>  *<!-- 过滤器映射配置 -->* <**filter-mapping**>  *<!-- 过滤器名称，和上面一致即可 -->* <**filter-name**>myFilter</**filter-name**>  *<!-- 设置拦截的路径。设置为/\*表示拦截所有资源，包括Servlet/JSP/HTML等。 -->* <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**> </**web-app**> |

可以看出Filter和Servlet的配置方法基本一致。需要注意：

（1）/\*拦截了所有请求，包括静态资源，比如html页面、js文件等。

（2）若在web.xml配置Servlet，则Servlet的配置需要在Filter配置之后。

## 1.3 拦截指定的请求

有时不希望拦截全部请求，只希望拦截一部分请求。可用url-pattern和servlet-name说明拦截路径和要拦截的Servlet名称，并且支持通配符。一个filter-mapping中可配置多个url-pattern和servlet-name节点。示例如下：

|  |
| --- |
| <**filter-mapping**>  *<!-- 拦截指定的jsp -->* <**url-pattern**>/index.jsp</**url-pattern**>  *<!-- 拦截所有的jsp -->* <**url-pattern**>\*.jsp</**url-pattern**>  *<!-- 根据servlet的名称拦截 -->* <**servlet-name**>IndexServlet</**servlet-name**> </**filter-mapping**> |

可以自己定义组合来拦截。如果只想拦截Servlet，那么可以定个规范，比如让所有的Servlet都以Action开头，这样就好拦截了。

## 1.4 过滤器设置编码案例和使用注解

之前，每次请求和响应几乎都要先设置编码，我们可以在过滤器中就实现编码的处理，使得GET和POST请求和响应不再考虑编码的情况。

[同样，我们也能使用@WebFilter注解来对Filter进行配置，这样就不用自行配置web.xml了。缺点](mailto:同样，我们也能使用@WebFilter注解来对Filter进行配置，这样就不用自行配置web.xml了。缺点)是使用注解不能确定过滤器执行的顺序。

代码的实现：

|  |
| --- |
| @WebFilter(urlPatterns = {**"/\*"**}) **public class** EncodingFilter **implements** Filter {   @Override  **public void** init(FilterConfig filterConfig) {}   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain)  **throws** IOException, ServletException {  *// 先设置请求编码* req.setCharacterEncoding(**"UTF-8"**);  *// 预先就设置响应编码。因为等到响应回来再设置的话，就晚了。必须在write()之前设置。* resp.setContentType(**"text/html;charset=UTF-8"**);  *// 放行* filterChain.doFilter(req, resp);  }    @Override  **public void** destroy() {} } |

这样，Servlet中就不再需要考虑编码问题。

## 1.5 登录判断过滤器

对一些管理系统来说，除了主页、注册和登录等页面外，进入其他页面时都要判断用户是否登录了。若没有登录，则跳转到主页。可将用户登录信息保存在session中。

为了简化登录判断，我们写个过滤器来进行判断。过滤器中可设置不需要拦截的页面、拿到session进行判断等。

代码示例：

|  |
| --- |
| @WebFilter(urlPatterns = {**"/\*"**}) **public class** EncodingFilter **implements** Filter {  *// 先定义不需要过滤的页面* String[] **extraPages** = {**"index"**, **"login"**, **"signup"**};  *// 通过该方法判断该请求的地址是否需要拦截* **private boolean** needFilter(HttpServletRequest request) {  *// 得到请求地址* String uri = request.getRequestURI();  *// 截取得到页面名称* String pageName = uri.substring(uri.lastIndexOf(**"/"**) + 1, uri.length());  *// 遍历extraPages，看书否需要拦截* **boolean** flag = **false**;  **for** (String page : **extraPages**) {  **if** (page.equals(pageName)) {  flag = **true**;  **break**;  }  }  **return** flag;  }  @Override  **public void** init(FilterConfig filterConfig) {}   @Override  **public void** doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain)  **throws** IOException, ServletException {  *// 将req和resp转换成HttpServletRequest和HttpServletResponse* HttpServletRequest request = (HttpServletRequest) req;  HttpServletResponse response = (HttpServletResponse) resp;  **if** (needFilter(request)) {  *// 需要拦截。则获得session* HttpSession session = request.getSession();  **if** (session == **null** || session.getAttribute(**"username"**) == **null**) {  *// 没有用户信息，则跳转到首页* request.getRequestDispatcher(**"/index"**).forward(request, response);  } **else** {  *// 放行* filterChain.doFilter(req, resp);  }  } **else** {  *// 不需拦截，直接放行* filterChain.doFilter(req, resp);  }  }   @Override  **public void** destroy() {} } |

# 2. 监听器(Listener)

## 2.1 监听器概述

一、含义：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的情况。

二、监听器监听的对象以及作用：

监听器一般监听的是服务器自行创建的对象，即request、session和servletContext。因为自己创建使用的对象可自己控制，不需要监听。监听上述的服务器对象的创建、销毁和属性的改变，能够让开发者在特定的时机触发执行想要的自定义的操作。例如，如果想要在服务器启动时就执行一些操作，则可监听servletContext，当servletContext创建时就触发执行自定义代码。

三、有哪些监听器？Servlet提供了监听器接口，开发者只要自定义类实现监听器接口，即可创建好一个监听器。

（1）生命周期监听器。

生命周期监听器接口用于监听服务器对象的创建和销毁。Servlet提供了三个接口分别用于监听request、session和servletContext对象的创建和销毁：

1. ServletRequestListener接口：监听request对象的创建和销毁。

2. HttpSessionListener接口：监听session对象的创建和销毁。

3. ServletContextListener接口：监听servletContext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。（具体在下面讲）

（2）属性监听器。

属性监听器用于监听request、session和servletContext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。

1. ServletRequestAttributeListener接口：监听request对象属性的变化；

2. HttpSessionAttributeListener接口：监听session对象属性的变化；

3. ServletContextAttributeListener接口：监听servletContext对象属性的变化。

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。

（2）Session相关监听器。

Servlet还提供了Session相关的监听器：

1. HttpSessionBidingListener：监听对象绑定到session上的事件；

2. HttpSessionActivationListener：监听session序列化及反序列化的事件（了解即可）。

## 2.2 监听器的使用

开发监听器的步骤：

（1）写一个普通Java类，实现相关监听器接口；

（2）在web.xml中配置监听器类，或者直接在监听器类上使用@WebListener注解。

### 2.2.1 生命周期监听器使用示例

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;  @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..."**);  } } |

上述代码中使用了@WebListener注解，并且在代码中测试了能否成功执行操作并拿到request和域对象。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HttpSessionListener和ServletContextListener也是类似的步骤，只是ServletContextListener的创建事件代码会在服务器启动时就执行（因为servletContext对象就是在服务器启动时创建的），而HttpSessionListener中创建事件代码也会稍后执行。

对于HttpSessionListener，需要注意的是：当浏览器关闭时，并不会调用HttpSessionListener中session销毁事件的代码，因为服务器端此时的该session对象并未被销毁。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，之前的SESSIONID会消失，因此导致服务器并不能拿到之前的session对象，但这只是说明session失效了，不是session对象被销毁了。在Tomcat中，session默认30分钟销毁（可自行在web.xml中配置销毁时间），当然也可调用session.invalidate()方法主动销毁session对象，这样HttpSessionListener中的session销毁事件才会触发执行。

使用ServletContextListener进行监听使用较多，因为可在服务器启动时就执行一些操作。比如以后要学习的Spring框架，就是通过配置监听器来完成Spring容器加载的（Spring的ContextLoaderListener就实现了ServletContextListener接口）。

### 2.2.2 属性监听器示例

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 |

### 2.2.3 Session相关监听器示例

HttpSessionBindingListener监听对象绑定/解除绑定到sesison上的事件。通过示例来解释。

我们只要写一个自定义类（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 {   *// 这个类基本上是一个实体类* **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。

需要注意的是，这里我们并没有配置“监听器”，即既没有配置注解，也没有配置web.xml。这是因为Employee这个对象是我们自己定义使用的，而不是服务器对象，因此无需“监听”。

# 3. 国际化

国际化，叫internationalization，软件中常用i18n表示国际化，因为i和n之间有18个字母。

软件国际化就是软件能根据用户电脑所使用的语言，而显示不同的语言，不必为一种语言重新开发一套系统。在进行Web应用国际化之前，需要了解JDK提供的本地化类Locale。

## 3.1 本地化类Locale

本地化对象用于封装当前语言、国家等特征。可以获取当前系统的本地化对象，也可以模拟其他区域的本地化对象。

代码示例：

|  |
| --- |
| **public class** TestLocale{  **public static void** main(String[] args) {  Locale china = Locale.*getDefault*(); *//获得默认的本地环境，这里得到的是中国的环境  //也可以通过Locale.CHINA指定获得中国的环境* System.***out***.println(china.getCountry()); *// CN 获得国家简称* System.***out***.println(china.getDisplayCountry()); *//中国 获得国家名称* System.***out***.println(china.getLanguage()); *//zh 获得语言简称* System.***out***.println(china.getDisplayLanguage()); *//中文 获得语言名称   //模拟美国* Locale us = Locale.***US***;  System.***out***.println(us.getCountry()); *// US* System.***out***.println(us.getDisplayCountry());*// 美国* System.***out***.println(us.getLanguage()); *//en* System.***out***.println(us.getDisplayLanguage()); *//英文* System.***out***.println();  } } |

## 3.2 静态数据的国际化

静态数据就是指网页上静态的、固定的内容，比如大部分文字的描述等。为了实现静态数据的国际化，需要先写好各个“国家”的配置文件。配置文件的名字要满足这样的格式：“基础名\_语言简称\_国家简称.properties”。比如我们需要支持中文和英文，那么我们在src下新建两个文件，分别叫“msg\_zh\_CN.properties”和“msg\_en\_US.properties”。内容为：

|  |
| --- |
| **name**=**用户姓名 age**=**用户年龄 gender**=**用户性别** |

和

|  |
| --- |
| **name**=**User Name age**=**User Age gender**=**User Gender** |

注意：properties中，实际不能存储中文。这里只是为了好看，实际上应该采用URL编码。

在程序中，用ResourceBundle类读取国际化的资源。例如：

|  |
| --- |
| **public class** TestLocale{  **public static void** main(String[] args) {  *//用Locale对象创建ResourceBundle对象* ResourceBundle bundle = ResourceBundle.*getBundle*(**"msg"**, Locale.***CHINA***); *//直接传入配置文件的基础名即可* String name = bundle.getString(**"name"**);  String age = bundle.getString(**"age"**);  String gender = bundle.getString(**"gender"**);  System.***out***.println(name);  System.***out***.println(age);  System.***out***.println(gender);  *//如果换用Locale.US创建对象，则输出是英文的* } } |

## 3.3 动态数据的格式化

动态数据是在程序运行时动态产生的，比如数据库中的数据，需要特殊处理。Java中提供了解决数值，货币，时间，日期等数据国际化问题的类，位于java.util包和java.text包中。主要使用的API有：

国际化货币：NumberFormat.getCurrencyInstance()；

国际化数字：NumberFormat.getNumberInstance()；

国际化日期：DateFormat.getDateTimeInstance(dateStyle, timeStyle, aLocale)。

例子：

|  |
| --- |
| **public class** TestLocale{  *//国际化货币* @Test  **public void** testI18N1(){  Locale locale = Locale.***CHINA***; *//模拟语言环境* **double** money = 100; *//数据准备* NumberFormat nf = NumberFormat.*getCurrencyInstance*(locale); *//用指定语言环境创建对象，如果传递参数，则采用系统默认环境  //国际化货币* String moneyStr = nf.format(money);  System.***out***.println(moneyStr);  }  *//解析的例子，这些API每个都有format和parse方法，可以利用它反过来解析。比如把$100解析成100* @Test  **public void** parseCurrency() **throws** Exception{  String moneyStr = **"$100"**;  NumberFormat nf = NumberFormat.*getCurrencyInstance*(Locale.***US***);  Number n = nf.parse(moneyStr);  System.***out***.println(n.intValue());  }  *//数值国际化* @Test  **public void** formatNumber(){  NumberFormat nf = NumberFormat.*getNumberInstance*(Locale.***US***);  String nstr = nf.format(1000000);  System.***out***.println(nstr); *//输出1,000,000  //可以自行测试把字符串"1,000,222,414"解析成数字，用这个API很方便* }  *//日期国际化* @Test  **public void** formatDate(){  *//前两个参数传递的分别是日期格式和时间格式* **int** dateStyle = DateFormat.***SHORT***;  **int** timeStyle = DateFormat.***SHORT***;  DateFormat df = DateFormat.*getDateTimeInstance*(dateStyle, timeStyle, Locale.***CHINA***);  String date = df.format(**new** Date());  System.***out***.println(date);  */\*  \* 日期  \* FULL 2015年3月4日 星期三  \* LONG 2015年3月4日  \* MEDIUM 2015-3-4  \* SHORT 15-3-4  \*  \* 时间  \* FULL 下午04时31分59秒 CST  \* LONG 下午04时32分37秒  \* MEDIUM 16:33:00  \* SHORT 下午4:33  \*/   //如果只想用日期或者时间，则直接使用DateFormat.getDateInstance(int style, Locale alocale)或者  //DateFormat.getTimeInstance(int style, Locale alocale)  //解析的话，自己做。* } } |

## 3.4 JSP国际化案例

（1）首先准备好两种语言的配置文件。即比如上面的msg\_zh\_CN.properties和msg\_en\_US.properties。

（2）JSP页面即可国际化：

|  |
| --- |
| <%@ **page import**="**java.util.ResourceBundle**" %> <%@ **page pageEncoding**="**UTF-8**" **language**="**java**" %> <!DOCTYPE **html**> **<%** *//用Java代码加载资源* ResourceBundle bundle = ResourceBundle.*getBundle*(**"msg"**, request.getLocale()); *//在src文件夹下的不需要包名  //浏览器访问时，会自动带区域信息，服务器把它封装到request里面了，我们可以直接拿到。  //下面一律用Java代码* **%>** <**html**> <**head**>  <**meta charset="UTF-8"**>  <**title**>**<%=**bundle.getString(**"title"**)**%>**</**title**> </**head**> <**body**>  <**form**>  **<%=**bundle.getString(**"userName"**)**%>** ：<**input type="text"**>  **<%=**bundle.getString(**"pwd"**)**%>**：<**input type="password"**>  <**input type="submit" value="<%=**bundle.getString(**"submit"**)**%>"**>  </**form**> </**body**> </**html**> |

## 3.5 使用国际化JSTL标签库

上述在JSP里写Java代码是不规范的，我们可以使用JSTL中提供的“国际化与格式化”标签进行国际化。只要掌握下面三个标签就能基本实现国际化：

<fmt:setLocale value=""/> 设置本地化对象

<fmt:setBundle basename=""/> 设置工具类

<fmt:message></fmt:message> 显示国际化文本

例如这样优化上面的JSP：

|  |
| --- |
| <%@ **page pageEncoding**="**UTF-8**" **language**="**java**" %> *<%-- 引入fmt（即格式化）标签库--%>* <%@ **taglib prefix**="**fmt**" **uri**="**http://java.sun.com/jsp/jstl/fmt**" %> *<%--设置本地化对象--%>* <**fmt:setLocale value="${**pageContext.request.locale**}"**></**fmt:setLocale**> *<%--设置工具类 basename指明配置文件名，var指明该对象名称，等会直接使用--%>* <**fmt:setBundle basename="msg" var="bundle"**></**fmt:setBundle**> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**>  <**title**><**fmt:message key="title" bundle="${**bundle**}"**></**fmt:message**></**title**> </**head**> <**body**>  <**form**>  <**fmt:message key="userName" bundle="${**bundle**}"**></**fmt:message**> ：<**input type="text"**>  <**fmt:message key="pwd" bundle="${**bundle**}"**></**fmt:message**>：<**input type="password"**>  <**input type="submit" value="**<**fmt:message key="submit" bundle="${**bundle**}"**></**fmt:message**>**"**>  </**form**> </**body**> </**html**> |

注意：JSTL标签一定要有始有终，一定要有结束标签，不能像HTML那样随意，否则出错。

fmt标签中除了有国际化的标签，还有格式化的标签，比如

（1）格式化数值：

<fmt:formatNumber pattern=" " value="100.99"></fmt:formatNumber>

若pattern="0.00"，则保留两位小数，四舍五入，

若pattern="#.##"，则当小数末尾是0时就不显示。

（2）格式化日期：

<fmt:formatDate pattern="yyyy-MM-dd" value="${date}"/>

pattern表示格式，value表示日期对象。这样实现域对象中的日期就非常方便。

# 4. 文件上传与下载

## 4.1 文件上传概览

使用表单上传文件的前提：

（1）必须是表单上传；

（2）表单提交方式必须是POST提交；

（3）要设置form表单的enctype属性值为“multipart/form-data”。（enctype的默认值是“application/x-www-form-urlencoded”，平时不会写。）

示例的JSP文件（为了在Servlet中查看普通上传和文件上传的区别，这里上传一个用户名和一个文件）：

|  |
| --- |
| <%@**page language**="**java**" **pageEncoding**="**UTF-8**" %> <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"**>  <**title**>文件的上传</**title**> </**head**> <**body**>  <**form action="${**pageContext.request.contextPath**}/upload" method="post" enctype="multipart/form-data"**>  用户名：<**input name="userName" type="text"** >  上传文件：<**input type="file" name="upFile"**>  <**input type="submit"**>  </**form**> </**body**> </**html**> |

相应的Servlet端为：

|  |
| --- |
| @WebServlet(urlPatterns = **"/upload"**) **public class** UploadServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *//对于POST，使用getInputStream获得流来查看所得数据。* InputStream is = req.getInputStream();  *//用字符流输出该InputStream流看是什么信息* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String oneLine = **null**;  **while**((oneLine = br.readLine()) != **null**){  System.***out***.println(oneLine);  }  br.close();  } } |

先上传一个文本文件，名字是a.txt，里面有三行英文。比如a...b...c....。并且用户名输入的是“Tom”，则控制台结果是：

|  |
| --- |
| ------WebKitFormBoundary2fBvBAnpwt5t0ZCL  Content-Disposition: form-data; name="userName"  Tom  ------WebKitFormBoundary2fBvBAnpwt5t0ZCL  Content-Disposition: form-data; name="upFile"; filename="a.txt"  Content-Type: text/plain  aaaaaaaaaaaaaaa  bbbbbbbbbbbbbbb  ccccccccccccccc  ------WebKitFormBoundary2fBvBAnpwt5t0ZCL-- |

观察可知，服务器把上传的数据分组了，每一个上传项都包含在“------WebKitFormBoundary...”里面，比如第一项是普通表单上传，name值就是表单的name值，隔一行是“Tom”；第二项是上传的文件，name值还是对应表单元素的name属性值，filename是文件名，Content-Type指明文件是文本类型，隔一行输出了文件内容。

因此可知，如果我们要获得文件内容和上传文件的信息（比如文件名），就要自己手动解析这些数据。但是这样做太麻烦了，这里就不介绍具体做法了。我们可以使用第三方框架处理上传的这些数据，这就是下面讲的。

## 4.2 使用开源组件FileUpload获得上传的数据

这里使用Apache提供的开源组件FileUpload处理文件上传的数据。

使用FileUpload除了要引入fileupload.jar外，还要引入一个依赖包：commons-io.jar，该包简化了一些IO操作。这两个包都可在Apache上自行搜索下载。

FileUpload就是把每一个上传项封装成一个FileItem对象（注意，即使上传的不是文件，也会被封装成一个FileItem对象）。开发者只要会使用API操作FileItem对象即可。主要的API有：

|--FileItemFactory：创建FileItem的工厂类，负责把每个请求项封装成FileItem对象

|--ServletFileUpload：可获取所有的FileItem对象

List parseRequest(request) ; 获得所有FileItem对象

boolean isMultipartContent(request) , 判断上传表单类型是否是文件上传表单

setFileSizeMax 设置单个文件的最大值

setSizeMax 设置总文件最大大小

setHeaderEncoding(“UTF-8”) 设置上传文件的编码，便于处理中文文件名

|--FileItem 封装了上传表单的元素值

getFieldName(); 获得上传表单元素名称，就是name属性值

getString(); 获得上传的数据，就是value值。

getContentType(); 获得上传文件的类型（如果上传的不是文件，该项为null）

getInputSteam(); 获得上传文件的文件流（如果上传的不是文件，该项为null）

getName(); 获得文件名（如果上传的不是文件，该项为null）

write(file); 写文件

delete(); 删除临时文件

例子：

|  |
| --- |
| @WebServlet(urlPatterns = **"/upload"**) **public class** UploadServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  **try**{  *//先创建FileItemFactory工厂* FileItemFactory factory = **new** DiskFileItemFactory();  *//用工厂创建ServletFileUpload，以便获得FileItem* ServletFileUpload upload = **new** ServletFileUpload(factory);  *//可以进行一些设置* upload.setFileSizeMax(100 \* 1024 \* 1024);*//设置单个文件允许的最大大小是100M* upload.setSizeMax(130 \* 1024 \* 1024); *//设置总文件大小不能超过130M，因为有的表单可能上传多个文件* upload.setHeaderEncoding(**"UTF-8"**); *//设置上传编码为UTF-8以便处理中文文件名。  //判断当前的表单是否是上传文件的表单* **if**(upload.*isMultipartContent*(req)){  *//获得FileItem对象* List<FileItem> list = upload.parseRequest(req);  *//遍历FileItem* **for**(FileItem item : list){  **if**(item.isFormField()){  *//如果是普通的表单项，就获得该表单的名字和值即可* String fieldName = item.getFieldName();  String content = item.getString(**"UTF-8"**); *//设置UTF-8，才能正确处理中文* System.***out***.println(**"name="** + fieldName + **",值是："** + content);  }**else**{  *//不是普通的表单项，就说明是文件上传的表单项* String name = item.getFieldName();*//获得表单元素名* String fileName = item.getName(); *//获得文件名  /\*  特别注意：对于Firefox和Chrome，只上传文件名，是不带路径的，即如果我上传D盘下的a.txt文件，则这里fileName直接拿到的就是a.txt  但是，IE和Edge这两微软产品，上传的文件是带路径的，就是D:\a.txt。  为了统一，我们要获取该文件的文件名：  \*/* **if**(fileName.contains(**"\\"**)){  fileName = fileName.substring(fileName.lastIndexOf(**'\\'**) + 1);  }  String content = item.getString(); *//获得内容，这里就是文件内容的编码* String type = item.getContentType(); *//文件类型* InputStream is = item.getInputStream(); *//文件流 ，实际大多在删除临时文件时用到  /\*  解决上传文件同名问题  若很多用户上传同名文件，比如1.jpg，不希望覆盖彼此的文件  则后台处理，给每个文件添加一个唯一标记  \*/* String uid = UUID.*randomUUID*().toString(); *//随机生成一个UUID  //让UUID与文件名拼接* fileName = uid + **"#"** + fileName;   *//将文件保存在当前应用的upload目录下* String path = getServletContext().getRealPath(**"/upload"**);  File file = **new** File(path);  **if**(!file.exists()){  file.mkdir(); *//如果文件夹不存在就创建* }  file = **new** File(file, fileName);  item.write(file); *//保存文件* is.close(); *//关闭这个文件流，下面再删除%Tomcat%/temp目录下的临时文件，这样才能删除成功，否则删除临时文件是失败的* item.delete(); *//删除临时文件* }  }  }**else**{  System.***out***.println(**"当前表单不是文件上传表单"**);  }  }**catch**(Exception e){  e.printStackTrace();  **throw new** RuntimeException();  }  }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  **this**.doGet(req, resp);  } } |

## 4.3 文件下载

文件下载比较简单，首先得到要下载的文件的文件流。

然后要设置content-type的类型，不能是text/html了。这里的类型要用MIME类型。MIME (Multipurpose Internet Mail Extensions) 是描述消息内容类型的因特网标准。比如可以设置\*/\*来下载任意文件，设置image/\*下载任何图片等。

同时，还要设置Content-Disposition响应头，内容为“attachment;filename=文件名”。这时，数据传输使用文件流。

例子：

|  |
| --- |
| @WebServlet(urlPatterns = **"/start"**) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException, ServletException {  *// 得到要下载的文件* File file = **new** File(**"E:/Chrome浏览器.exe"**);  *// 用URL编码文件名，防止中文乱码* String fileName = URLEncoder.*encode*(file.getName());  *// 设置content-type* resp.setContentType(**"\*/\*"**);  *// 设置内容长度，就是下载的文件的大小。一些下载工具以此作为下载进度的依据* resp.setContentLengthLong(file.length());  *// 设置Content-Disposition响应头* resp.setHeader(**"Content-Disposition"**, **"attachment;filename="** + fileName);   *// 下面就是用流输出文件* BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(file));  **byte**[] buffer = **new byte**[1024 \* 1024];  **int** length = 0;  **while** ( (length = bis.read(buffer)) != -1 ) {  resp.getOutputStream().write(buffer, 0, length);  }  *// 关闭流* bis.close();  resp.getOutputStream().close();  } } |

# 5. Java Web开发模式

## 5.1 Java Bean

JavaBean就是一个普通的java类，但需要符合以下规范：

（1）必须提供无参构造器；

（2）类中所有属性要私有化，并提供getter和setter方法。

JavaBean是Java类的规范，主要用JavaBean来封装数据。常用的使用场景是作为实体对象Entity使用。

注意，boolean类型属性的get方法名字以is开头，比如private boolean flag的get方法应该是isFlag()，而不是getFlag()。

## 5.2 Java Web开发模式

前面讲的三层架构仍然会在Web项目中采用。其实不管什么类型的项目，都应该考虑分层，这样使层次清晰，结构分明，易于维护。

在Web开发中，常用MVC设计模式。M表示model，即模型层；V表示view，即视图层；C表示control，即控制层。

经过前面的学习，我们清楚的知道：JSP、HTML等就是视图层，专门用于向用户展示内容；Servlet就是控制层，用来控制请求的转发和调用业务方法处理数据；M就是存储数据的模型，就是下面的三层结构实现的业务数据的封装。

因此使用MVC结构进行开发逻辑很清晰。

# 6 扩展：内省的应用

## 6.1 请求的一般处理

Java后端在处理请求数据时，常用的操作是先将请求的数据封装到一个JavaBean对象中，然后进行具体的业务操作。下面就举例说明。

例如addStudent.jsp页面中有个表单需要提交学生信息（姓名name、年龄age和性别gender），则代码如下：

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**meta charset="UTF-8"** />  <**title**>添加学生</**title**> </**head**> <**body**> <**form action="/WebDemo/addStudentServlet" method="POST"**>  姓名：<**input type="text" name="name"** />  年龄：<**input type="number" name="age"** />  性别：<**input type="text" name="gender"** />  <**input type="submit"**> </**form**> </**body**> </**html**> |

Java后端新建一个Student实体类，用于封装学生信息。根据Java Bean规范，编写Student类如下：

|  |
| --- |
| **package** com.bean;  **public class** Student {  **private** String **name**;  **private int age**;  **private** String **gender**;    **public** Student() {}   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return age**;  }   **public void** setAge(**int** age) {  **this**.**age** = age;  }   **public** String getGender() {  **return gender**;  }   **public void** setGender(String gender) {  **this**.**gender** = gender;  }    *// 添加toString()方法* @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **", gender='"** + **gender** + **'\''** +  **'}'**;  } } |

然后在AddStudentServlet这个处理类中，先将各个参数封装到一个Student对象中，接下来再处理业务：

|  |
| --- |
| **package** com.servlet.demo;  @WebServlet(urlPatterns = **"/addStudentServlet"**) **public class** AddStudentServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**);  *// 创建对象* Student student = **new** Student();  student.setName(req.getParameter(**"name"**)); *// 设置name属性* student.setAge(Integer.*parseInt*(req.getParameter(**"age"**))); *// 设置age属性。这里就不处理异常情况了* student.setGender(req.getParameter(**"gender"**));  *// 输出信息查看* System.***out***.println(student);  *// 下面针对Student对象进行业务上的处理......* } } |

那么上述代码有何缺点呢？很明显，上述代码没有通用性，如果现在不是添加学生，而是添加教师，JavaBean的属性不一样了，代码就不能通用了。我们希望能通过一段代码做到：能把请求的参数直接封装成指定类型的对象，只要传递的参数名称和Java类属性一致即可。

那么怎么实现呢？需要用到内省的知识。

## 6.2 内省的应用

在Java SE时也讲过内省，但是应用之处不是很清晰，现在通过Web开发来回顾和应用下内省。

内省可以说是操作Java Bean的一套API。在上面例子中，就是把请求的参数封装进JavaBean的属性中，因此我们主要使用内省操作属性。当然，内省的原理是基于反射的。

如果将6.1中封装Java Bean的代码用内省完成，则为：

|  |
| --- |
| **package** com.servlet.demo;  @WebServlet(urlPatterns = **"/addStudentServlet"**) **public class** AddStudentServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**);  *// 创建对象* Student student = **new** Student();   **try** {  *// 使用内省获得Java Bean的描述信息* BeanInfo beanInfo = Introspector.*getBeanInfo*(Student.**class**);  *// 获得属性信息* PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();  *// 遍历属性信息* **for** (PropertyDescriptor pd : pds) {  Method method = pd.getWriteMethod(); *// 获得属性的写方法对象* String name = pd.getName(); *// 获得属性名称  // 如果属性名是class，就忽略本次循环。因为class属性是Object类的，所有子类都有这个属性* **if** (**"class"**.equals(name)) {  **continue**;  }  Class clazz = pd.getPropertyType(); *// 获得属性的类型  // 接下来，将对应的参数封装进对象。  // 需要注意的是，要判断属性的类型进行处理。现在就进行一个简单的int类型处理，其余当做字符串处理* **if** (**"int"**.equals(clazz.getName())) {  String value = req.getParameter(name);  method.invoke(student, Integer.*parseInt*(value)); *// 封装参数* } **else** {  method.invoke(student, req.getParameter(name)); *// 封装参数* }   }  *// 封装完毕。* System.***out***.println(student);  *// 业务处理代码......* } **catch** (Exception e) {  e.printStackTrace();  }   } } |

最终也能封装好对象。为了通用型，可以将上述代码提取成一个工具类方法：

|  |
| --- |
| **package** com.utils;  **public class** BeanHelper {  **public static** <T> T transferParam(HttpServletRequest request, Class<T> clz) **throws** Exception {  *// 创建对象* T t = clz.newInstance();  *// 使用类的描述信息* BeanInfo beanInfo = Introspector.*getBeanInfo*(clz);  *// 获得属性信息* PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();  *// 遍历属性信息* **for** (PropertyDescriptor pd : pds) {  Method method = pd.getWriteMethod(); *// 获得属性的写方法对象* String name = pd.getName(); *// 获得属性名称  // 如果属性名是class，就忽略本次循环。因为class属性是Object类的，所有子类都有这个属性* **if** (**"class"**.equals(name)) {  **continue**;  }  Class clazz = pd.getPropertyType(); *// 获得属性的类型  // 接下来，将对应的参数封装进对象。  // 需要注意的是，要判断属性的类型进行处理。现在就进行一个简单的int类型处理，其余当做字符串处理* **if** (**"int"**.equals(clazz.getName())) {  String value = request.getParameter(name);  method.invoke(t, Integer.*parseInt*(value)); *// 封装参数* } **else** {  method.invoke(t, request.getParameter(name)); *// 封装参数* }  }  **return** t;  } } |

这样Servlet中直接这样写：

|  |
| --- |
| **package** com.servlet.demo;  @WebServlet(urlPatterns = **"/addStudentServlet"**) **public class** AddStudentServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**);  **try** {  Student student = BeanHelper.*transferParam*(req, Student.**class**);  *// 封装完毕。* System.***out***.println(student);  *// 业务处理代码......* } **catch** (Exception e) {  e.printStackTrace();  }   } } |

这样也能封装好Student，并且transferParam方法同样能适应别的Java Bean（读者可以一试）。利用内省实现了通用性。

通过内省的机制，现在再回过头来看看Java Bean规范的意义，解析如下：

（1）必须提供无参构造器。意义：有了无参构造器，才能使用内省（反射）创建出需要的Java Bean对象（newInstance）。

（2）类中所有属性要私有化，并提供getter和setter方法。意义：Java Bean不能脱离封装性的基本特性。内省也会根据getter/setter方法分辨出哪些是属性的操作方法（需要命名规范）。

之前学习过Apache的BeanUtils库能简化内省的操作。这里同样能利用BeanUtils实现请求参数自动封装成Java Bean。我们只要引入BeanUtils库，使用populate方法即可：

|  |
| --- |
| **package** com.servlet.demo;  @WebServlet(urlPatterns = **"/addStudentServlet"**) **public class** AddStudentServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**);  Student student = **new** Student();  **try** {  *// populate方法会自动把map集合数据封装到bean对象中。因此使用req.getParameterMap方法* BeanUtils.*populate*(student, req.getParameterMap());  *// 封装完毕* System.***out***.println(student);   } **catch** (Exception e) {  e.printStackTrace();  }  } } |

正如之前所言，BeanUtils使用非常简单和方便，populate方法就是把Map集合中的数据封装到了Java对象中了。而且BeanUtils还能自动类型转换（比如age这个int类型就直接封装进去了）。BeanUtils自动类型转换能转换Java的基本类型（int long boolean等），如果要想BeanUtils能自动转换其他类型，就需要向BeanUtils注册一个类型转换器。

例如现在Student有个birthday属性，类型是Date，我们想前端传递过来的“yyyy-MM-dd”这样的日期字符串格式能够被BeanUtils自动封装到birthday中，那么就要在调用populate方法之前，向BeanUtils注册Date类型的类型转换器：

|  |
| --- |
| **package** com.servlet.demo;  @WebServlet(urlPatterns = **"/addStudentServlet"**) **public class** AddStudentServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding(**"UTF-8"**);  Student student = **new** Student();  **try** {  *// 注册一个转换器，让BeanUtils知道如何转换。  // register方法的参数：  // 参数1：用于定义如何转换，传递一个Converter对象。  // 参数2：转换的类型字节码，即将原数据进行转换后的类型字节码，这里是Date.class。* ConvertUtils.*register*(**new** Converter() {  @Override  **public** <T> T convert(Class<T> clazz, Object obj) {  *// convert方法的参数：  // 参数1：转换的类型字节码，即转换后的类型。该参数就相当于上面的参数2，这里是Date.class。  // 参数2：要转换的数据。这里就相当于要转换的时间字符串。  // 这里的目的是把obj转换成Date类型返回。* String dateStr = obj.toString(); *// 直接toString()用字符串接收，成为日期字符串。因为请求参数本就是字符串。  // 最后转换成Date返回。* SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  **try** {  **return** (T) sdf.parse(dateStr);  } **catch** (ParseException e) {  e.printStackTrace();  **return null**;  }  }  }, Date.**class**);    BeanUtils.*populate*(student, req.getParameterMap());  *// 封装完毕* System.***out***.println(student);   } **catch** (Exception e) {  e.printStackTrace();  }  } } |

这样日期字符串就能自动转换了（前提是满足yyyy-MM-dd格式）。

最后，BeanUtils中还常用copyProperties(obj1, obj2)方法，它的作用是将一个对象中的属性复制到另一个对象中。

总之，内省在一些框架的底层是很常用的，包括一些JSP标签、EL表达式以及以后要学习的一些框架等。通过本次内省的简单应用，以后使用框架开发时，也能知道其基本的原理。