# 1. 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。

## 1.1 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();  } } |

## 1.2 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Unicode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* } } |

## 1.3 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。* } } |

## 1.4 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**> |

## 1.5 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。

# 2. 文件上传与下载

## 2.1 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。）

（4）表单中提供<input type="file">类型的上传输入域。一个表单中，可以上传多个文件，只要提供多个上传输入域即可。

现在我们演示一下文件上传时，使用“multipart/form-data”时数据传递的格式。下面的JSP表单中上传了一个用户名和两个文件（name都使用note）：

|  |
| --- |
| <%@**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="note"**>  <**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();  } } |

上传时，用户名填写“Tom”，上传的两个文本文件分别是a.txt和b.txt，其中的文件内容是“aaaaa”和“bbbbb”，内容比较简单，便于我们观察。点击上传后，我们通过浏览器的开发者工具就能查看到请求的内容是：

|  |
| --- |
| **请求头：**  POST /WebDemo/upload HTTP/1.1  Host: localhost:8080  Connection: keep-alive  Content-Length: 422  Content-Type: multipart/form-data; boundary=----WebKitFormBoundary5MltSW901hQInmlQ  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.62 Safari/537.36  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8  Referer: http://localhost:8080/WebDemo/1.jsp  Accept-Encoding: gzip, deflate, br  Accept-Language: zh-CN,zh;q=0.9  **请求体：**  ------WebKitFormBoundary5MltSW901hQInmlQ  Content-Disposition: form-data; name="username"  Tom  ------WebKitFormBoundary5MltSW901hQInmlQ  Content-Disposition: form-data; name="note"; filename="a.txt"  Content-Type: text/plain  aaaaa  ------WebKitFormBoundary5MltSW901hQInmlQ  Content-Disposition: form-data; name="note"; filename="b.txt"  Content-Type: text/plain  bbbbb  ------WebKitFormBoundary5MltSW901hQInmlQ-- |

通过请求的数据格式和内容我们做以下总结：

（1）首先请求头中“Content-Type”内容是“multipart/form-data; boundary=----WebKitFormBoundary5MltSW901hQInmlQ”。“multipart/form-data”指明了表单的enctype属性值，其中的boundary指明“边界”是“----WebKitFormBoundary5MltSW901hQInmlQ”，这个边界boundary下面会用作表单数据的分组。boundary是浏览器自行产生的。

（2）请求体中同样包含了表单上传的数据，但这里的数据格式和以前的表单数据（即enctype为application/x-www-form-urlencoded）是不同的，每个input项数据以字符串“--boundary”做分隔（所以会比boundary多出两个“-”），每个input项数据的格式又是“头”和“体”两部分，它们之间有一个空行，例如“Content-Disposition: form-data; name="username"”是input项username的“头”，下面的“Tom”数据是“体”。同理，下面的file输入域数据也是这样的格式，通过其中的内容能拿到上传的数据、参数名和数据类型、上传的文件名filename（如果是文件的话）等信息。最后，用“--boundary--”作为表示主体内容的结束。我们在Tomcat中，也能看到主体内容是这样的数据。

现在我们再来对表单的enctype属性作一下说明：

该属性用于告知服务器请求正文的MIME类型的，相当于设置请求消息头的Content-Type。

（1）当enctype="application/x-www-form-urlencoded"时，相当于请求头的Content-Type设置为application/x-www-form-urlencoded，此时用POST请求的正文格式就类似于：

|  |
| --- |
| username=abc&password=123 |

在Servlet中，request.getParameter(String name)方法只能读取请求正文是application/x-www-form-urlencoded类型的数据，此时获得参数是没有问题的。

（2）当enctype="multipart/form-data"时，请求头的Content-Type就被设置为“multipart/form-data; boundary=XXX”，这时POST请求的正文格式就和上面讲的类似。

在Servlet中，是无法像之前那样通过request.getParameter(String name)方法得到参数数据的。如果想如果想得到表单的数据，需要解析其中的各种内容。虽然我们能够实现解析，但比较麻烦，因此下面我们直接使用第三方组件获得上传的数据。这里，重点在于了解原理。

## 2.2 使用FileUpload组件获得上传数据

### 2.2.1 相关API

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。会使用DiskFileItemFactory创建这个工厂类。

|--ServletFileUpload：可获取所有的FileItem对象

List parseRequest(request) ; 获得所有FileItem对象

boolean isMultipartContent(request) ，这是静态方法，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

isFormField()：是否是普通的表单字段（即不是文件输入域字段）

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

### 2.2.2 第一个例子

现在我们实现获取演示的JSP中上传的表单数据，代码如下：

|  |
| --- |
| **package** com.webdemo;  **import** org.apache.commons.fileupload.FileItem; **import** org.apache.commons.fileupload.FileItemFactory; **import** org.apache.commons.fileupload.FileUploadException; **import** org.apache.commons.fileupload.disk.DiskFileItemFactory; **import** org.apache.commons.fileupload.servlet.ServletFileUpload;  **import** javax.servlet.ServletException; **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** java.io.\*; **import** java.util.List;  @WebServlet(urlPatterns = **"/upload"**) **public class** UploadServlet **extends** HttpServlet {  @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  *// 检测form是否是multipart/form-data类型的* **if** (!ServletFileUpload.*isMultipartContent*(req)) {  **throw new** RuntimeException(**"The form's enctype attribute value must be multipart/form-data"**);  }  *// 解析请求体内容  // 1. 创建FileItemFactory工厂* FileItemFactory factory = **new** DiskFileItemFactory();  *// 2. 利用工厂创建ServletFileUpload对象* ServletFileUpload sfu = **new** ServletFileUpload(factory);  *// 3. 存放FileItem的数组* List<FileItem> fileItemList = **null**;  **try** {  *// 解析请求内容生成FileItem数组* fileItemList = sfu.parseRequest(req);  } **catch** (FileUploadException e) {  **throw new** RuntimeException(e);  }  *// 4. 遍历FileItem处理表单项的数据* **for** (FileItem fileItem : fileItemList) {  **if** (fileItem.isFormField()) {  *// 处理普通字段* processFormField(fileItem);  } **else** {  *// 处理文件上传字段* processFileUploadField(fileItem);  }  }  }   *// 处理普通字段的方法* **protected void** processFormField(FileItem fileItem) {  String name = fileItem.getFieldName(); *// 获得参数名* String value = fileItem.getString(); *// 获得参数值* System.***out***.println(name + **":"** + value);  }   *// 处理文件上传字段的方法* **protected void** processFileUploadField(FileItem fileItem) {  **try** {  InputStream in = fileItem.getInputStream(); *// 获得上传文件的输入流* String fileName = fileItem.getName();*// 获得上传的文件名  // 将文件输出到应用目录的files目录下。* String realPath = getServletContext().getRealPath(**"/files"**);  File storeDir = **new** File(realPath);  **if** (!storeDir.exists()) {  storeDir.mkdirs();  }  OutputStream out = **new** FileOutputStream(**new** File(storeDir, fileName));  *// 保存文件，操作IO流* **int** len = -1;  **byte**[] buff = **new byte**[1024];  **while** ( (len = in.read(buff)) != -1 ) {  out.write(buff, 0, len);  }  *// 关闭流* in.close();  out.close();  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  } } |

上面的代码是有点小问题的：

（1）如果使用IE浏览器上传文件，则会出错。这是因为IE和Edge浏览器上传文件时会上传文件的完整路径名，例如“D:\a.txt”，这样fileItem.getName()拿到的就是“D:\a.txt”。而其他浏览器上传文件时只会上传文件名，不会上传路径，因此拿到的直接就是a.txt。在开发中一般只要拿到文件名即可，因此为了兼容IE和Edge，需要处理文件名，只要截取即可。实际上在FileUpload官方FAQ页面中提供了解决办法，见网站：http://commons.apache.org/proper/commons-fileupload/faq.html中的第4个问题，推荐的方式是：

|  |
| --- |
| String fileName = fileItem.getName(); **if** (fileName != **null**) {  filename = FilenameUtils.getName(filename); } |

也就是说直接使用commons-io中的工具类即可。当然使用截取的方式我们也是能做的。

（2）在保存文件时我们写了一大堆IO流代码，实际上利用fileItem的write()方法可直接将上传的文件写到File对象中，省去了我们获得IO流和操作IO流的步骤。并且write()方法会主动删除上传文件时产生的临时文件，这个下面详细讲。

因此综合以上两点，我们把processFileUploadField方法完善为：

|  |
| --- |
| *// 处理文件上传字段的方法* **protected void** processFileUploadField(FileItem fileItem) {  **try** {  String fileName = fileItem.getName();*// 获得上传的文件名* **if** (fileName != **null**) {  fileName = FilenameUtils.*getName*(fileName); *// 处理文件名，确保只拿到文件名* }  *// 将文件输出到应用目录的files目录下。* String realPath = getServletContext().getRealPath(**"/files"**);  File storeDir = **new** File(realPath);  **if** (!storeDir.exists()) {  storeDir.mkdirs();  }  *// 保存文件* fileItem.write(**new** File(storeDir, fileName));  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } } |

## 2.3 文件上传注意的几个问题

本章解决的问题是基于2.2.2节的代码的。

### 2.3.1 安全性问题

之前上传文件直接保存在应用路径的files文件夹中了，这样是不安全的。例如用户上传一个“demo.jsp”文件，内容如下：

|  |
| --- |
| <%@ **page pageEncoding**="**UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>示例</**title**> </**head**> <**body**> **<%** Runtime runtime = Runtime.*getRuntime*();  runtime.exec(**"notepad.exe"**); **%>** </**body**> </**html**> |

然后用户通过访问“http://IP:port/项目名/files/demo.jsp”这个地址，就能让JSP中的Java代码执行，会导致严重的安全问题。解决该问题的方法就是把文件上传到用户不能访问的地方，例如保存到WEB-INF目录下，或者不要保存到本应用目录下。

### 2.3.2 文件重名被覆盖的问题

一个目录中的文件不能重名，否则会被覆盖。因此，必须给文件一个唯一的名字。如果是数据库应用，那么文件表一般设计如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 主键 | 文件路径 | 文件名 | 唯一文件名 | 所属用户ID | 上传时间 |
| 1 | /WEB-INF/files | demo.jsp | 1001.jsp | 1 | 2017-12-25 |
| 2 | /WEB-INF/files | demo.jsp | 1002.jsp | 1 | 2017-12-25 |
| ... |  |  |  |  |  |

文件还是存到磁盘中，通过文件表能找到文件所在路径，“唯一文件名”就是文件存储时的名称，通过这样解决文件名重复的问题。当然表中也记录了文件上传时的原始文件名，这样就能让用户知道自己上传的文件名是什么。

自然，这个“唯一文件名”是要由系统生成的唯一的编码，那么如何生成呢？这里我们使用UUID码。UUID是通用唯一识别码（Universally Unique Identifier），依据UUID标准能产生保证唯一的识别码（重复概率极小，可忽略）。在Java中，生成UUID码可使用UUID类，例如：

|  |
| --- |
| String uuid = UUID.*randomUUID*().toString(); System.***out***.println(uuid); |

生成的UUID码类似于（32个16进制数，中间有横杠分隔）：

|  |
| --- |
| 7771bf2c-1c88-4dbe-8c8f-903bfe186bda |

每次运行生成的UUID是不同的，保证是唯一的（包括不同的机器）。

我们这里不采用数据库，所以把上传的文件名以“UUID#文件名”的方式重新命名，这样既使文件名唯一，还能通过分隔符“#”识别原文件名，后续需要显示文件名时，只要截取第一个#后面的内容即可。

因此，这时processFileUploadField方法可写为：

|  |
| --- |
| **protected void** processFileUploadField(FileItem fileItem) {  **try** {  String fileName = fileItem.getName();*// 获得上传的文件名* **if** (fileName != **null**) {  fileName = FilenameUtils.*getName*(fileName);  }  *// 在文件名前面加上UUID和#分隔符* fileName = UUID.*randomUUID*().toString() + **"#"** + fileName;  *// 将文件输出到应用目录的WEB-INF/files目录下。* String realPath = getServletContext().getRealPath(**"/WEB-INF/files/"**);  File storeDir = **new** File(realPath);  **if** (!storeDir.exists()) {  storeDir.mkdirs();  }  *// 保存文件* fileItem.write(**new** File(storeDir, fileName));  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } } |

### 2.3.3 避免一个文件夹中文件过多

如果把上传的文件全部保存在一个文件夹中，则会造成一个文件夹中的文件过多，会导致查看该目录文件变慢。同时为了较好地管理这些文件，最好将这些文件存入不同的文件夹中。常用下面两种方案：

方案1：按照日期分目录存储文件。每天上传的文件会存入到以该天命名的文件夹中，例如2017年12月25日上传的文件将会存入“2017-12-25”文件夹中，同理，2017年12月26日上传的文件将会存入“2017-12-26”文件夹中。

示例代码：

|  |
| --- |
| **protected void** processFileUploadField(FileItem fileItem) {  **try** {  String fileName = fileItem.getName();*// 获得上传的文件名* **if** (fileName != **null**) {  fileName = FilenameUtils.*getName*(fileName);  }  *// 在文件名前面加上UUID和#分隔符* fileName = UUID.*randomUUID*().toString() + **"#"** + fileName;  *// 将文件输出到应用目录的WEB-INF目录下。* String realPath = getServletContext().getRealPath(**"/WEB-INF/files"**);  File storeDir = *generateChildDir*(realPath); *// 获得最终的目录  // 保存文件* fileItem.write(**new** File(storeDir, fileName));  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } }  *// 根据时间生成的“父目录 + 子目录”，并自动创建好目录* **public static** File generateChildDir(String basePath) {  DateFormat df = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  String date = df.format(**new** Date());  File file = **new** File(basePath, date);  **if** (!file.exists()) {  file.mkdirs();  }  **return** file; } |

但是上面的解决办法缺点是：如果某天上传文件过多，还是会导致一个文件夹中有过多文件。下面就用随机的方式来把文件放入不同的文件夹中。

方案2：按照文件名的hashCode随机存储到不同的目录中。先看代码（重写一个generateChildDir方法重载即可，其中要传递basePath和文件名，因为此时是根据文件名选择目录存储。最后processFileUploadField中改用这个方法即可）：

|  |
| --- |
| **public static** File generateChildDir(String basePath, String fileName) {  *// 获得文件名的hashCode* **int** hashCode = fileName.hashCode();  **int** dir1 = hashCode & 0xf; *// 一级目录* **int** dir2 = ( hashCode & 0xf0 ) >>> 4; *// 二级目录* File file = **new** File(basePath, dir1 + **"/"** + dir2);  **if** (!file.exists()) {  file.mkdirs();  }  **return** file; } |

上面生成一级和二级目录的方法分别是“hashCode&0xf”和“(hashCode&0xf0) >>> 4”，这是什么意思呢？

首先hashCode是根据文件名生成的32位int类型数据，可以认为根据文件名是随机生成的。生成一级和二级目录都使用了与操作，hashCode&0xf得到的结果就是hashCode后四位二进制位对应的数据，数据范围是“0-15”，而(hashCode&0xf0)>>>4就是得到hashCode的倒数第八位到倒数第五位的二进制数据结果，其数据范围也是“0-15”，大家可以通过二进制位运算进行了解。最后，一级和二级目录共会生成16\*16（即256）个目录，上传的文件会随机存储到这些目录中。

当然，也可通过其他的策略将文件存到不同的目录中。

### 2.3.4 限制上传文件的类型

在后台限制上传文件的类型一般是通过判断其后缀名或者其MIME类型。通过FileItem对象的getContentType()就能获得其MIME类型。但是通过扩展名和MIME类型也不能完全限制上传文件的类型：因为如果用户可以更改了某文件的扩展名，就无法通过扩展名判断类型了；而且也无法通过MIME判断类型，因为FileUpload直接从浏览器上传的数据中获得MIME类型，而Chrome和Firefox就是通过扩展名得到相应的MIME类型，只有IE浏览器会根据文件内容获得真实的MIME类型。

如果较真的话，也可以在服务器端读取内容判断其真实的MIME类型，但是一般不会这样做，只通过扩展名识别即可，节省服务器资源。

### 2.3.5 限制上传文件大小

很多情况下需要限制用户上传文件的大小以减轻服务器负担。在FileUpload中，可以使用ServletFileUpload的setFileSizeMax(long i)和setSizeMax(long i)方法，分别限制单个文件的上传大小和上传文件的总大小，单位是字节。例如以下代码：

|  |
| --- |
| @Override **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **if** (!ServletFileUpload.*isMultipartContent*(req)) {  **throw new** RuntimeException(**"The form's enctype attribute value must be multipart/form-data"**);  }  FileItemFactory factory = **new** DiskFileItemFactory();  ServletFileUpload sfu = **new** ServletFileUpload(factory);  *// 限制上传文件的大小* sfu.setFileSizeMax(3 \* 1024 \* 1024); *// 设置单个文件的上传大小为3M* sfu.setSizeMax(5 \* 1024 \* 1024); *// 设置上传的总文件的大小为5M  // 存放FileItem的数组* List<FileItem> fileItemList = **null**;  **try** {  fileItemList = sfu.parseRequest(req);  } **catch** (FileUploadException e) {  e.printStackTrace();  }  *// 遍历FileItem* **for** (FileItem fileItem : fileItemList) {  **if** (fileItem.isFormField()) {  processFormField(fileItem);  } **else** {  processFileUploadField(fileItem);  }  } } |

在运行中，如果表单上传的单个文件大小超过代码限制的大小，sfu.parseRequest(req)方法则出现类似如下异常：

|  |
| --- |
| org.apache.commons.fileupload.FileUploadBase$FileSizeLimitExceededException: The field note exceeds its maximum permitted size of 3145728 bytes. |

如果表单上传的文件总大小超过限制的大小，sfu.parseRequest(req)方法则出现类似如下异常：

|  |
| --- |
| org.apache.commons.fileupload.FileUploadBase$SizeLimitExceededException: the request was rejected because its size (29385665) exceeds the configured maximum (5242880) |

异常中说明了文件超过了指定的大小，可以看出，“FileUploadBase$FileSizeLimitExceededException”和“FileUploadBase$SizeLimitExceededException”这两个异常实际是“FileUploadBase”类中的内部类异常，因为它们之间是用“$”分隔的。在程序中，最好处理这个异常以便程序正常运行，因此代码改进如下：

|  |
| --- |
| @Override **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **if** (!ServletFileUpload.*isMultipartContent*(req)) {  **throw new** RuntimeException(**"The form's enctype attribute value must be multipart/form-data"**);  }  FileItemFactory factory = **new** DiskFileItemFactory();  ServletFileUpload sfu = **new** ServletFileUpload(factory);  *// 限制上传文件的大小* sfu.setFileSizeMax(3 \* 1024 \* 1024); *// 设置单个文件的上传大小为3M* sfu.setSizeMax(5 \* 1024 \* 1024); *// 设置上传的总文件的大小为5M  // 存放FileItem的数组* List<FileItem> fileItemList = **null**;  **try** {  fileItemList = sfu.parseRequest(req);  } **catch** (FileUploadBase.FileSizeLimitExceededException e) {  resp.getWriter().write(**"单个文件不能超过3M"**);  } **catch** (FileUploadBase.SizeLimitExceededException e) {  resp.getWriter().write(**"总文件不能超过5M"**);  } **catch** (FileUploadException e) {  e.printStackTrace();  **throw new** RuntimeException(**"解析请求失败"**);  }  *// 由于出现异常可能导致fileItemList为null，因此这里用普通的for循环，并在其中检测null* **for** (**int** i = 0; fileItemList != **null** && i < fileItemList.size(); i++) {  FileItem fileItem = fileItemList.get(i);  **if** (fileItem.isFormField()) {  processFormField(fileItem);  } **else** {  processFileUploadField(fileItem);  }  } } |

上面代码处理了异常，使客户端能看到错误结果。在我使用的新版Tomcat中，上述异常代码虽然能执行，但通过response对象并不能向客户端响应数据和转发请求，直接表现为浏览器无响应！这时，需要修改一下Tomcat的server.xml中的Connector配置，加上“maxSwallowSize="-1"”这个属性值，例如：

|  |
| --- |
| <Connector port="8080" protocol="HTTP/1.1"  connectionTimeout="20000"  redirectPort="8443" maxSwallowSize="-1" /> |

重启服务器运行，这样就能看到响应的结果了。maxSwallowSize貌似用于设置上传出错时，允许客户端发送的请求体的数据大小。官方文档中说，如果不设置足够的长度（默认是2M），则可能无法响应客户端，因此这里设置成-1表示不限制客户端发送的数据大小。

Tomcat中有很多关于文件上传的参数，如果以后上传时需要其他错误，可以从这方面解决。

### 2.3.6 中文乱码问题

在multipart-data表单中，如果上传的参数值或者文件名带有中文，则有乱码现象，通过如下方式解决：

（1）对于表单中的普通字段，只要在调用getString方法获得参数值时，通过getString(String charset)重载设置下数据编码即可。例如页面的数据时UTF-8编码的，那么只要用fileItem.getString("UTF-8")即可得到正确的数据。例如：

|  |
| --- |
| **protected void** processFormField(FileItem fileItem) {  String name = fileItem.getFieldName(); *// 获得参数名* String value = **null**; *// 获得参数值* **try** {  value = fileItem.getString(**"UTF-8"**);  *// 下面是要处理UnsupportedEncodingException异常的* } **catch** (UnsupportedEncodingException e) {  **throw new** RuntimeException(**"不支持的编码类型"**);  }  System.***out***.println(name + **":"** + value); } |

（2）对于上传文件名中的中文乱码，只要设置request.setCharacterEncoding(String charset)即可，因为文件名就是包含在请求体的每项参数数据的“头”中的，而设置request.setCharacterEncoding就是设置“头”的编码的，包括请求头的编码。

### 2.3.7 手动处理上传时的临时文件

为了提高速度，FileUpload在处理上传文件时会默认使用一个10KB大小的缓存，否则一个字节一个字节地读取是很慢的。该缓存是在内存中的，当缓存不够用时（例如上传的文件大小超过10KB），FileUpload会将上传文件的数据保存到一个临时目录中，随后，再将此临时文件的输入流提供给程序操作，由程序决定如何处理上传的文件（如修改、保存等）。

FileUpload默认使用Tomcat的“temp”目录作为文件上传的临时文件目录。在代码中，可以控制UploadFile的缓存大小和临时文件目录，方式1是通过DiskFileItemFactory的构造方法，方式2是调用DiskFileItemFactory的setSizeThreshold和setRepository方法进行控制。示例如下：

|  |
| --- |
| *// 方式1，使用构造方法，。 // 参数1是缓存大小，以字节为单位；参数2是设置临时目录* FileItemFactory factory = **new** DiskFileItemFactory(20 \* 1024, **new** File(**"d:/temp"**)); */\* 方式2是通过DiskFileItemFactory对象调用方法设置。 此时就不能使用FileItemFactory接口了，且传给ServletFileUpload的也是该对象*  *\*/* DiskFileItemFactory f = **new** DiskFileItemFactory(); *// 方式2* f.setSizeThreshold(20 \* 1024); f.setRepository(**new** File(**"d:/temp"**)); ServletFileUpload sfu = **new** ServletFileUpload(f); |

开发中一般使用FileUpload默认的即可，缓存也不要设置大，毕竟是内存空间。

我们之前说fileItem.write()方法会自动删除临时文件，如果我们还是用自己写的IO操作处理的话，临时文件是不会被删除的，大家可以观看Tomcat/temp目录验证一下。如果要手动删除临时文件，只要主动调用fileItem对象的delete()方法即可。例如下面就是手动删除的例子：

|  |
| --- |
| **protected void** processFileUploadField(FileItem fileItem) {  **try** {  String fileName = fileItem.getName();*// 获得上传的文件名* **if** (fileName != **null**) {  fileName = FilenameUtils.*getName*(fileName);  }  *// 在文件名前面加上UUID和#分隔符* fileName = UUID.*randomUUID*().toString() + **"#"** + fileName;  *// 将文件输出到应用目录的WEB-INF目录下。* String realPath = getServletContext().getRealPath(**"/WEB-INF/files"**);  File storeDir = **new** File(realPath); *// 获得最终的目录* **if** (!storeDir.exists()) {  storeDir.mkdirs();  }  *// 保存文件，采用自己的IO流操作* InputStream in = fileItem.getInputStream();  FileOutputStream out = **new** FileOutputStream(**new** File(storeDir, fileName));  **byte**[] buff = **new byte**[1024];  **int** len = -1;  **while** ( (len = in.read(buff)) != -1 ) {  out.write(buff, 0, len);  }  in.close();  out.close();  *// 最后删除临时文件* fileItem.delete();  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } } |

在开发中，还是建议直接使用write()方法。

最后，我们还发现一个问题：如果一个文件输入域中没有上传文件，那么最终服务器会保存一个空的文件，这时，还需要判断一下fileName不为空，不为空才执行下面的保存操作，否则什么都不做。

## 2.4 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。

之前在Servlet中做过文件下载，上述内容也基本知道。唯一需要注意的是：“attachment;filename=文件名”中的“文件名”需要先经过URL编码，尤其是包含中文等字符时，否则浏览器下载时不能正确显示文件名。即使文件名中不包含中文，为了确保文件名正确，也应该对文件名进行URL编码，这是因为JDK官方文档在URLEncoder类中对URL编码进行了解释（这个在之前讲解中提到过），其中说：空格这样的字符经过URL编码后变为“+”。因此为了确保文件名正确，强烈建议对所有文件名进行URL编码。

例子：

|  |
| --- |
| @Override **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  *// 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* 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="** + URLEncoder.*encode*(fileName, **"UTF-8"**));  *// 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(); } |

也可让用户下载之前上传的文件，基本只要按照之前的方式处理路径问题和文件名问题即可，读者可自行实现。一般，文件都是存在文件系统中，而不是存在数据库的BLOB字段中，因为数据库并不擅长存储文件，我们只要在数据库中拿到文件的URL，照样能让用户下载文件。

# 3. Java Web开发模式

## 3.1 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()。

## 3.2 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。

# 4 扩展：内省的应用

## 4.1 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。

## 4.2 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。