第3章

Spring Boot视图技术

本章学习目标

- 掌握 Thymeleaf 模板引擎。
- 掌握 Spring Boot 处理 JSON 数据。
- 掌握 Spring Boot 处理国际化问题。
- 掌握 Spring Boot 文件的上传和下载。
- 掌握 Spring Boot 的异常处理。

根据党的二十大报告，必须坚持“创新是第一动力”。创新是一个国家、一个民族发展进步的不竭动力，是推动人类社会进步的重要力量。

Web 应用开发是现代软件开发中的重要部分。Spring Boot 的 Web 开发内嵌了 Servlet 和服务器，并结合 Spring MVC 来完成开发。Web 开发是一种基于 B/S 架构的应用软件开发技术，分为前端和后端。前端的可视化及用户交互由浏览器实现，即以浏览器作为客户端，实现客户端与服务器远程的数据交互。

本章通过学习基于用户信息的 Web 层技术、集成 Spring MVC 技术和集成静态页面模板技术，学习创新的思维模式，并深刻体会这种时代宝贵的工匠精神。

3.1 创建静态 Web 页面


视频讲解

Spring Boot 项目在不使用任何模板引擎的前提下，想要直接访问 HTML 页面，是如何实现的呢？只需要把静态文件（HTML，JS，CSS）放在 resource 的 static 文件夹里面即可正常访问。下面通过示例讲解 Spring Boot 如何访问静态 Web 页面。

（1）创建 Spring Boot Web 应用 springboot0301。

（2）在项目的 src/main/resource/static 目录下创建 CSS 文件夹和 js 文件，将文件放入对应的文件夹下。

（3）选中 static 目录并右击，选择新建 HTML 文件，命名为 index.html。HTML 文件代码如下：

```
<!DOCTYPE html>
<html lang="zh_CN" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
```

```
    <title>Title</title>
    <link href="css/style.css" rel="stylesheet" type="text/css">
    <script src="js/check.js"></script>
</head>
<body>
<div align="center"><br>
    <a class="title">用户登录</a><br>
</div>
<form name="form1" method="post">
    <table width="398" height="215" border="1" align="center" cellpadding="0"
cellspacing="0">
        <tr>
            <td width="394" height="213">
                <table width="91%" height="80%" border="0" align="center"
cellpadding="1" cellspacing="1">
                    <tr>
                        <td width="120" align="right">用户名:</td>
                        <td width="208">
                            <input name="userid" type="text" id="userid" size="15"
maxlength="20">
                        </td>
                    </tr>
                    <tr>
                        <td width="120" align="right">密码:</td>
                        <td width="208">
                            <input name="password" type="password" id="password" size=
"15" maxlength="20">
                        </td>
                    </tr>
                    <tr>
                        <td width="120" height="23" align="right"> </td>
                        <td width="208">
                            <div align="left">
                                <input type="submit" name="Submit" value="登录"
onclick="javascript:return(checkform());">
                                <input type="reset" name="reset" value="重填">
                            </div>
                        </td>
                    </tr>
                </table>
                <div align="center"></div>
                <div align="center"></div>
                <div align="center"></div>
                <div align="center"></div>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

(4) 运行程序。

index.html是默认的启动页面,在浏览器地址栏中输入http://localhost:8080/后,结果如图3-1所示。

图 3-1 静态 Web 页面的运行效果图

3.2 Spring Boot 对 JSP 的支持


视频讲解

Spring Boot 默认不支持 JSP,因为 JSP 的性能相对较低。官方推荐使用 Thymeleaf,如果想在项目中使用 JSP,需要进行相关初始化工作。

下面通过实例演示 Spring Boot 如何访问 JSP 页面。

(1) 创建 Spring Boot Web 应用 springboot0302。由于 Spring Boot 使用 JSP 时需打包为 WAR 类型,所以在创建项目时修改打包方式,或者在后面的 pom.xml 文件中进行修改。

(2) 自动生成 ServletInitializer 类。

打开该类不难发现它继承了 SpringBootServletInitializer 这个父类,而 SpringBootServletInitializer 类是 Spring Boot 提供的 Web 程序初始化的入口,在使用外部容器(如使用外部 Tomcat)运行项目时会自动加载并且装配。

实现 SpringBootServletInitializer 的子类需要重写一个 configure 方法,方法内自动根据 Springboot0302Application.class 的类型创建一个 SpringApplicationBuilder,将其交付给 Spring Boot 框架来完成初始化运行配置。

(3) 整理脚本样式静态文件。

JS 脚本、CSS 样式、Image 等静态文件默认放置在项目的 src/main/resource/static 目录下。

(4) 配置 Spring Boot 支持 JSP。

打开 pom.xml 文件(Maven 配置文件)可以看到,之前构建项目时已经添加了 Web 模块。因为在 JSP 页面中使用 EL 和 JSTL 标签显示数据,所以除了添加 Servlet 和 Tomcat 依赖外,还需要添加 JSTL 依赖,具体代码如下。

```
<!-- WAR 包部署到外部的 Tomcat 中已经包含了以下依赖,在此需要对其进行添加否则会与内嵌的
Tomcat 容器发生冲突 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
<!-- Servlet 依赖 -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
</dependency>
<!-- 配置 JSP JSTL 的支持 -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
<!-- 引入 Spring Boot 内嵌的 Tomcat 对 JSP 的解析包,否则无法解析 JSP 页面 -->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

(5) 配置视图。

必须按照 JSP 文件之前的文件目录风格配置视图,因此需要在 src/main 目录下创建 webapp/WEB-INF/views 文件夹。修改 application.properties 文件,使得 Spring MVC 支持视图的跳转目录指向为/main/webapp/views,代码如下。

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

(6) 创建 Book 实体类、BookController 控制器及对应的 BookDAO 数据层等,具体代码见源文档。

(7) 在 WEB-INF/views 目录下新建 booklist.jsp 文件,主要展示所有图书的信息,具体代码如下。

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title>网上书店</title>
<link href="css/style.css" rel="stylesheet" type="text/css">
</head>
<body>
<%@ include file="header.jsp" %>
<br><a class="title">在线购书</a><br>
<table width="700" border="1" cellpadding="0" cellspacing="0">
  <tr>
    <td width="43%" height="26"><div align="center">书名</div></td>
    <td width="17%"><div align="center">作者</div></td>
    <td width="17%"><div align="center">出版社</div></td>
    <td width="5%"><div align="center">价格</div></td>
    <td width="9%"><div align="center">操作</div></td>
  </tr>
```

```
  <c:forEach items = " ${bookList}" var = "book">
    <tr>
      <td height = "32"><div align = "center"> ${book.getTitle()} </div></td>
      <td><div align = "center"> ${book.getAuthor()}</div></td>
      <td><div align = "center"> ${book.getPublisher()}></div></td>
      <td><div align = "center"> ${book.getPrice()}></div></td>
      <td><div align = "center"><a href = "bookdetail.jsp?isbn = ${book.get综合业务数字网()}">
<img src = "images/buy.gif" width = "45" height = "16" border = "0"></a></div></td>
    </tr>
  </c:forEach>
</table>
<p> </p>
<%@ include file = "footer.jsp" %>
</body>
</html>
```

(8) 启动测试。

在浏览器地址栏中输入 http://localhost:8080/后,在页面中输入任意用户名和密码,单击登录按钮,就可以进入网上书店的首页,如图 3-2 所示。

图 3-2　JSP 页面的运行效果图

3.3　Thymeleaf 的基本语法

Thymeleaf 是一款用于渲染 XML/XHTML/HTML5 内容的模板引擎。与 JSP、Velocity、FreeMaker 等类似,它也可以轻易地与 Spring MVC 等 Web 框架进行集成,以作为 Web 应用的模板引擎。与其他模板引擎相比,Thymeleaf 最大的特点是能够直接在浏览器中打开并正确显示模板页面,而不需要启动整个 Web 应用。

Thymeleaf 支持 Spring Expression Language 语言作为方言,也就是 SpEL,SpEL 是可

以用于 Spring 中的一种 EL 表达式。

简而言之,与以往使用过的 JSP 不同,Thymeleaf 使用 HTML 的标签来完成逻辑和数据的传入及渲染,而不用像 JSP 一样作为一个 Servlet 被编译再生成。即便是单独用 Thymeleaf 开发的 HTML 文件,依旧可以正确打开并有少量(相对)有价值的信息,并且是可以被浏览器直接打开的。用 Thymeleaf 完全替代 JSP 是可行的。

Thymeleaf 的主要作用是把 model 中的数据渲染到 HTML 中,因此其语法主要是如何解析 model 中的数据。在 HTML 页面中引入 Thymeleaf 命名空间,即在 HTML 模板文件中对动态的属性使用 th:命名空间修饰。

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

这样才可以在其他标签里面使用 th:* 这样的语法,这是下面语法的前提。

3.3.1 变量表达式

通过项目 springboot0303 来学习 Thymeleaf 的语法。

(1) 新建一个实体类:User

```
public class User {
  private String name;
  private Integer age;
  private String sex;
 }
```

(2) 在模型中添加数据:

```
@GetMapping("show1")
public String show2(Model model){
  User user = new User();
  user.setAge(21);
  user.setName("张三");
  user.setSex("男");
  model.addAttribute("user", user);
  return "index";
}
```

语法说明:

Thymeleaf 通过 ${} 来获取 model 中的变量,注意这不是 EL 表达式,而是 OGNL 表达式。

示例:

在页面获取 user 数据,其效果如图 3-3 所示。

图 3-3 利用 OGNL 表达式获取数据

```
<h1>
    欢迎您:<span th:text="${user.name}">请登录</span>
</h1>
```

OGNL 表达式的语法与 EL 表达式几乎是一样的。两者的区别在于,OGNL 表达式写在一个名为 th:text 的标签属性中,该标签即为指令。

Thymeleaf 崇尚自然模板，即模板是纯正的 HTML 代码，脱离模板引擎，在纯静态环境也可以直接运行。如果直接在 HTML 中编写 ${}这样的表达式，显然在静态环境下就会出错，这不符合 Thymeleaf 的理念。

Thymeleaf 中所有的表达式都需要写在指令中，指令是 HTML5 中的自定义属性，在 Thymeleaf 中的所有指令都是以 th:开头的。由于表达式 ${user.name}是写在自定义属性中的，因此在静态环境下，表达式的内容会被当作是普通字符串，浏览器会自动忽略这些指令，从而不会报错。

现在不经过 Spring MVC，而是直接用浏览器打开页面，如图 3-4 所示。

图 3-4　用浏览器打开页面

在静态环境下，th 指令不会被识别，但是浏览器也不会报错，而是把它当作一个普通属性处理。这样，span 的默认值“请登录”就会显示在页面上。如果是在 Thymeleaf 环境下，th 指令就会被识别和解析，而 th:text 的含义就是替换所在标签中的文本内容，于是 user.name 的值就替代了 span 中默认的“请登录”。

指令的设计，正是 Thymeleaf 的高明之处，也是它优于其他模板引擎的原因。动静结合的设计，使得无论是前端开发还是后端开发都可以完美契合。

需要注意的是，如果不支持这种 th:的命名空间写法，那么可以把 th:text 换成 data-th-text，Thymeleaf 同样可以兼容。

另外，出于安全考虑，th:text 指令会把表达式读取到的值进行处理，防止 HTML 的注入。例如，将字符串“你好”存入到 Model 中，在页面中可以使用如下两种方式获得消息文本：

```
<span th:text="${msg}"></span><!-- 不对 HTML 标签解析，输出原始内容 -->
<span th:utext="${msg}"></span><!-- 对 HTML 标签解析，输出加粗的"你好" -->
```

3.3.2　自定义变量

下面这个例子是分别展示获取到的用户的所有信息。

```
<h2>
    <p>姓名:<span th:text="${user.name}">Jack</span></p>
    <p>年龄:<span th:text="${user.age}">21</span></p>
</h2>
```

当数据量比较多时,频繁地写 user. 就会非常麻烦。因此,Thymeleaf 提供了自定义变量来解决该问题。

```
<h2 th:object="${user}">
    <p>姓名:<span th:text="*{name}">Jack</span></p>
    <p>年龄:<span th:text="*{age}">21</span></p>
</h2>
```

首先,在 h2 上用 th:object="${user}"获取 user 的值,并且保存。然后,在 h2 内部的任意元素上,通过 *{属性名}的方式获取 user 中的属性,这样就省去了大量的 user. 前缀。th:object 声明变量一般情况下会与 *{}一起配合使用,达到事半功倍的效果。

3.3.3 方法

Thymeleaf 通过 ${}来获取容器上下文环境中的变量,而表达式是 OGNL 表达式。OGNL(Object-Graph Navigation Language),即对象图形化导航语言,它是一种能够方便地操作对象属性的开源表达式语言。

OGNL 表达式本身支持方法的调用,例如:

```
<h2 th:object="${user}">
    <p>姓:<span th:text="*{name.split(' ')[0]}">张</span></p>
    <p>名:<span th:text="*{name.split(' ')[1]}">三</span></p>
</h2>
```

这里调用了 name(字符串)的 split 方法。

Thymeleaf 提供了一些内置对象,并且在这些对象中提供了一些方法,以方便调用。在获取这些对象后,需要使用“#对象名”来引用。

(1) 一些环境相关对象。

#ctx:获取 Thymeleaf 自己的 Context 对象。

#request:如果是 Web 程序,可以获取 HttpServletRequest 对象。

#response:如果是 Web 程序,可以获取 HttpServletResponse 对象。

#session:如果是 Web 程序,可以获取 HttpSession 对象。

#servletContext:如果是 Web 程序,可以获取 HttpServletContext 对象。

(2) Thymeleaf 提供的全局对象。

#dates:处理 java.util.date 的工具对象。

#calendars:处理 java.util.calendar 的工具对象。

#numbers:用来对数字格式化的方法。

#strings:用来处理字符串的方法。

#bools:用来判断布尔值的方法。

#arrays:用来处理数组的方法。

#lists:用来处理 List 集合的方法。

#sets:用来处理 Set 集合的方法。

#maps：用来处理 Map 集合的方法。

例如，在环境变量中添加日期类型对象：

```
@GetMapping("show2")
public String show2(Model model){
    model.addAttribute("today", new Date());
    return "index";
}
```

在页面中处理如下：

```
<p>
  今天是：<span th:text="${#dates.format(today,
'yyyy-MM-dd')}">2022-01-14</span>
</p>
```

添加日期对象的运行效果如图 3-5 所示。

图 3-5 #dates 全局对象

3.3.4 字面值

有时需要在指令中填写基本类型，如字符串、数值、布尔等，但并不希望被 Thymeleaf 解析为变量，这时将其称为字面值。

（1）字符串字面值。

使用一对单引号引用的内容就是字符串字面值。

```
<p>你正在观看 <span th:text="'Thymeleaf'">template</span> 的字符串常量值.</p>
```

th:text 指令中的 Thymeleaf 并不会被认为是变量，而会被认为是一个字符串。字符串字面值的程序运行效果如图 3-6 所示。

图 3-6 字符串字面值的程序运行效果图

（2）数字字面值。

数字字面值不需要任何特殊语法，直接写入内容即可，而且可以进行算术运算。

```
<p>今年是<span th:text="2022"></span>.</p>
<p>两年后将会是 <span th:text="2022 + 2"></span>.</p>
```

数字字面值的程序运行效果如图 3-7 所示。

（3）布尔字面值。

布尔类型的字面值是 true 或 false。

```
<div th:if="true">
    你填的是 true
</div>
```

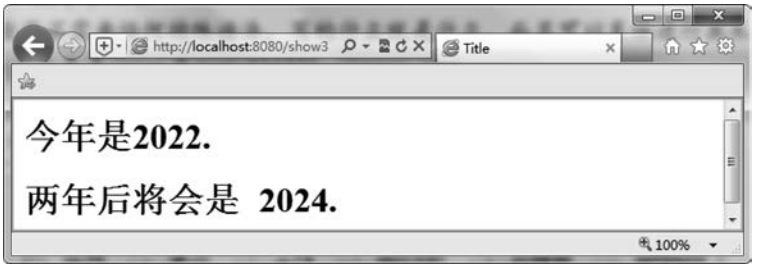


图 3-7 数字字面值的程序运行效果图

这里引用了一个 th:if 指令。

3.3.5 拼接

在指令中经常会遇到普通字符串与表达式拼接的情况：

```
<span th:text="'欢迎您:' + ${user.name} "></span>
```

字符串字面值需要用单引号，拼接起来非常麻烦，Thymeleaf 对此进行了简化，使用一对|即可。

```
<span th:text="|欢迎您: ${user.name}|"></span>
```

这与上面的语法是完全等效的，同时省去了字符串字面值的书写。

3.3.6 运算

需要注意的是，${}内部通过 OGNL 表达式引擎进行解析，而外部通过 Thymeleaf 的引擎进行解析，因此运算符尽量放在${}外。

(1) 算术运算。

支持的算术运算符：+、-、*、/、%。

```
<span th:text=" ${user.age}"></span><span th:text=" ${user.age} % 2 == 0"></span>
```

算术运算符的程序运行效果如图 3-8 所示。

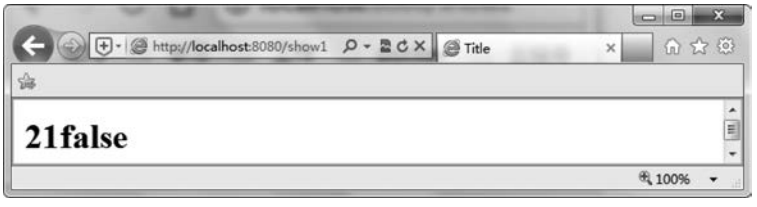


图 3-8 算术运行符的程序运行效果图

(2) 比较运算。

支持的比较运算：>、<、>=、<=、==、!=。需要注意的是，==和!=不仅可以比较数值，而且可以比较对象，与 equals 的功能类似。

除了使用常规的比较运算符外，还可以使用别名：gt(>)、lt(<)、ge(>=)、le(<=)、not(!)、eq(==)、ne(!=)。

```
<span th:text="'你的年龄是否小于 30:'+ ( ${user.age} < 30)"></span>
<span th:text="'你的年龄是否超过 30:'+ ( ${user.age} gt 30)"></span>
```

比较运算符的程序运行效果如图 3-9 所示。

图 3-9 比较运算符的程序运行效果图

(3) 三元运算。

三元运算符的表达式为 condition ? then :else。

condition：条件。

then：条件成立的结果。

else：不成立的结果。

其中的每个部分都可以是 Thymeleaf 中的任意表达式。

```
<span th:text="'你的性别：'+(${user.sex}?'男':'女')"></span>
```

三元运算符的程序运行效果如图 3-10 所示。

图 3-10 三元运算符的程序运行效果图

(4) 默认值。

指令中的取值有时可能为空，这时需要进行非空判断，可以使用表达式“?:默认值”简写。

```
<span th:text="'你的年龄是'+(${user.age}?: 20)"></span>
```

当前面的表达式值为 null 时，就会使用后面的默认值。

注意：“?:”之间没有空格。

默认值的程序运行效果如图 3-11 所示。

图 3-11 默认值的程序运行效果图

3.3.7 循环

当信息页面的数据格式相同时,页面通常对它们进行循环迭代。JSTL 有一个<c:foreach>,同理 Thymeleaf 也有一个 th:each,其作用都是一样的,用于遍历数组、List、Set、Map 等数据。

假如有用户的 java.util.List 集合的 users 在 Context 中。

```
<tr th:each="user : ${users}">
    <td th:text="${user.name}"></td>
    <td th:text="${user.age}"></td>
    <td th:text="${user.sex}"></td>
</tr>
```

java.util.List 类型不是可以在 Thymeleaf 中使用迭代的唯一值类型,下面这些类型的对象都可以通过 th:each 进行迭代。

(1) 任何实现 java.util.Iterable 接口的对象,其值将被迭代器返回,而不需要在内存中缓存所有值。

(2) 任何实现 java.util.Enumeration 接口的对象。

(3) 任何实现 java.util.Map 接口的对象,在迭代映射时,iter 变量将是 java.util.Map.Entry 类。

(4) 任何数组。

(5) 任何将被视为包含对象本身的单值列表。

循环语句的程序运行效果如图 3-12 所示。

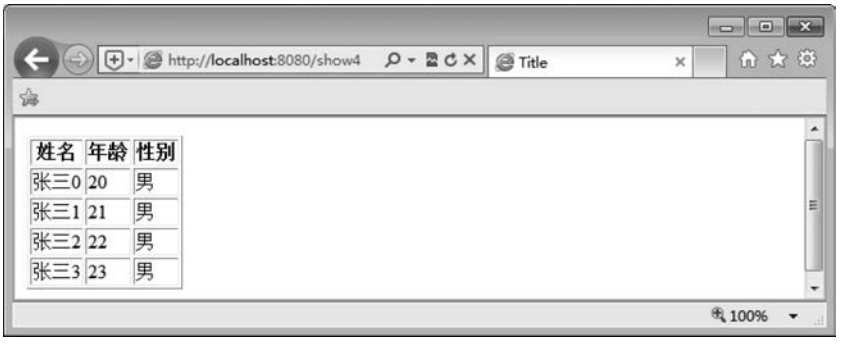


图 3-12 循环语句的程序运行效果图

在迭代过程中,经常会使用到它的一些迭代状态,如当前迭代的索引、迭代变量中的元素的总数、当前迭代的是奇数还是偶数、当前是否为第一个元素、当前是否为最后一个元素等。在 JSP 的 JSTL 中,<c:foreach items="${list}" var ="li" varStatus="status">包含 status 属性。在使用 th:each 时,Thymeleaf 提供了一种用于跟踪迭代状态的机制:状态变量。状态变量在每个 th:each 属性中定义,并包含以下数据。

(1) index 属性:当前迭代索引,从 0 开始。

(2) count 属性:当前迭代计数,从 1 开始。

(3) size 属性:迭代变量中元素的总量。

(4) current 属性:每次迭代的 iter 变量,即当前遍历的元素。

（5）even/odd 布尔属性：当前迭代是偶数还是奇数。

（6）first 布尔属性：当前迭代是否为第一个迭代。

（7）last 布尔属性：当前迭代是否为最后一个迭代。

```
<tr th:each="user,loopStatus: ${users}" th:class="${loopStatus.odd}?odd">
    <td th:text="${loopStatus.count}"></td>
    <td th:text="${user.name}"></td>
    <td th:text="${user.age}"></td>
     <td th:text="${user.sex}"></td>
</tr>
```

状态变量的程序运行结果如图 3-13 所示。

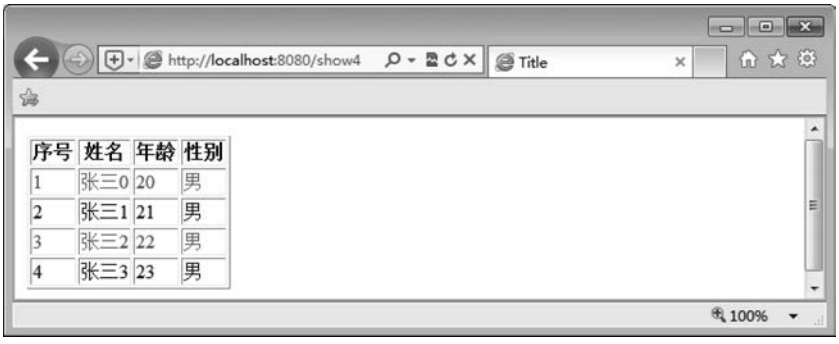


图 3-13　状态变量的程序运行结果图

迭代状态变量(本例中的 loopStatus)在 th:each 属性中,通过在变量 user 后直接写其名称来定义,用逗号分隔。与 iter 变量一样,状态变量的作用范围也是 th:each 属性的标签定义的代码片段中。

如果没有显式地设置状态变量,则 Thymeleaf 将始终创建一个默认的迭代变量,该状态迭代变量名称为：迭代变量＋“Stat”。

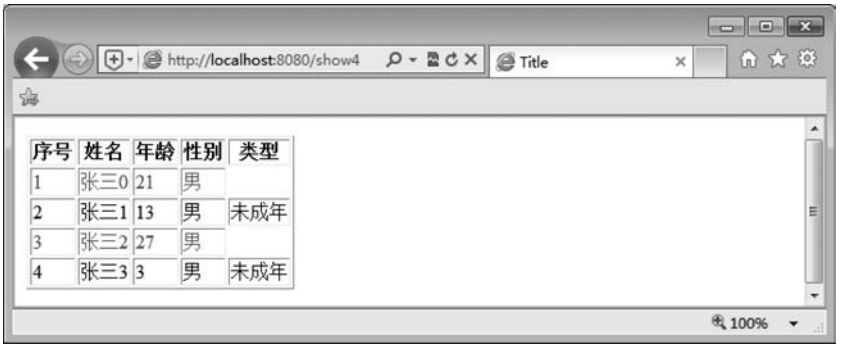
```
<tr th:each="user : ${users}" th:class="${userStat.odd}?odd">
    <td th:text="${userStat.count}"></td>
    <td th:text="${user.name}"></td>
    <td th:text="${user.age}"></td>
    <td th:text="${user.sex}"></td>
</tr>
```

3.3.8　逻辑判断

很多时候只有在满足某个条件时,才将一个模板片段显示在结果中,否则不进行显示。例如,只有当用户年龄小于 18 岁时,才显示为未成年人,否则不显示。th:if 属性用于满足逻辑判断的需求。

```
<tr th:each="user,loopStatus: ${users}" th:class="${loopStatus.odd}?odd">
    <td th:text="${loopStatus.count}"></td>
    <td th:text="${user.name}"></td>
    <td th:text="${user.age}"></td>
    <td th:text="${user.sex}"></td>
    <td><span th:if="${user.age}<18">未成年</span></td>
</tr>
```

逻辑判断的程序运行结果如图 3-14 所示。



序号	姓名	年龄	性别	类型
1	张三0	21	男	
2	张三1	13	男	未成年
3	张三2	27	男	
4	张三3	3	男	未成年

图 3-14　逻辑判断的程序运行效果图

th:if 属性不仅只以布尔值作为判断条件,它还将按照如下规则判定指定的表达式结果。

(1) 如果表达式结果为布尔值,则判定为 true 或者 false。

(2) 如果表达式的值为 null,th:if 将判定此表达式为 false。

(3) 如果值是数字且为 0 时,判定为 false; 不为零时则判定为 true。

(4) 如果值是字符串且为“false”“off”“no”时,判定为 false; 否则判定为 true。当字符串为空时,也判定为 true。

(5) 如果值不是布尔值、数字、字符或字符串的其他对象,只要不为 null,则判定为 true。

th:unless 是 th:if 的反向属性,它们判断的规则一致,只是 if 当结果为 true 时进行显示,unless 当结果为 false 进行显示。

3.3.9　分支控制 switch

th:switch/th:case 与 Java 中的 switch 语句等效,有条件地显示匹配的内容。只要其中一个 th:case 的值为 true,则同一个 switch 语句中的其他 th:case 属性都将被视为 false。当有多个 case 的值为 true 时,则只取第一个。

switch 语句的 default 选项指定为 th:case =“*”,即当没有 case 的值为 true 时,将显示 default 的内容,如果有多个 default,则只取第一个。

```
<tr th:each="user,loopStatus: ${users}" th:class="${loopStatus.odd}?odd">
    <td th:text="${loopStatus.count}"></td>
    <td th:text="${user.name}"></td>
    <td th:text="${user.age}"></td>
    <td th:text="${user.sex}"></td>
    <td th:switch="${user.age}<18">
        <span th:case="true">未成年</span>
        <span th:case="*">成年人</span>
    </td>
</tr>
```

分支语句程序的运行效果如图 3-15 所示。

序号	姓名	年龄	性别	类型
1	张三0	1	男	未成年
2	张三1	2	男	未成年
3	张三2	21	男	成年人
4	张三3	6	男	未成年

图 3-15 分支语句程序的运行效果图

3.3.10 Thymeleaf 模板片段

系统中的很多页面都有公共内容，如菜单、页脚等，这些公共内容可以提取并放在一个称为“模板片段”的公共页面中，其他页面可以引用该公共页面。

模板片段可以是 HTML 标签，也可以使用 th:fragment 属性定义片段。

首先定义一个页脚片段 footer.html，代码如下所示。

```
<span th:fragment = "frag1"> frag1 </span>
<span th:fragment = "frag2"> frag2 </span>
<div id = "footer1"> footer1 </div>
<div>
    <div id = "footer2"> footer2 </div>
</div>
<div>
    <span class = "content"> footer3 </span>
    <span class = "content"> footer4 </span>
</div>
<div th:fragment = "welcome(userName)">
    <span th:text = "|hello,| + ${userName}"></span>
</div>
```

接下来就可以在 Web 页面中使用 th:insert、th:replace、th:include 属性来包含页脚片段，这 3 个属性的区别如下。

- th:insert：在当前标签里面插入模板中的标签。
- th:replace：替换当前标签为模板中的标签。
- th:include：在标签里面插入模板的标签内容。

这里以使用 th:insert 属性插入片段为例，介绍其语法规则，其他两个属性类似。

(1) th:insert="~{模板名称}"。

插入模板的整个内容。

(2) th:insert="~{模板名称::选择器}"。

插入模板的指定内容，选择器可以对应 th:fragment 定义的名称，也可以用类似 JQuery 选择器的语法选择部分片段。

片段选择器语法：

① /name：选择子节点中节点名称为 name 的节点。

② //name：选择全部子节点中节点名称为 name 的节点。

③ name[@attr='value']：选择名称为 name 且属性值为 value 的节点，如有多个属性则用 and 连接。

④ //name[@attr='value'][index]：选择名称为 name 且属性值为 value 的节点，并指定节点索引。

片段选择器的简化语法：

① 可以省略@符号。

② 使用#符号代替 id 选择，如 div#id 等价于 div[id='id']。

③ 使用.符号代替 class 选择，如 div.class 等价于 div[class='class']。

④ 使用%代替片段引用，如片段节点使用了 th:ref 或 th:fragment，则可使用 div%ref 来选取节点。

(3) th:insert="~{::选择器}"。

不指定模板名称，则选择器作用于当前页面。

(4) th:insert="~{this::选择器}"。

与"~{::选择器}"类似，不同之处是在本页面找不到片段时，会到模板引擎的 process 方法处理的模板中寻找片段。

(5) 模板片段也支持传入变量，其引用语法为~{footer.html::名称(参数)}。

Thymeleaf 引用模板片段的代码如下所示。

```
<h4>th:insert 引用片段</h4>
引用指定模板的整个内容
<div th:insert="~{footer.html}"></div>
引用指定模板的片段
<div th:insert="~{footer.html::frag1}"></div>
引用本页面的片段
<div th:insert="~{::frag3}"></div>
<div th:insert="~{this::frag3}"></div>
<div th:fragment="frag3">frag3</div>

<h4>th:replace、th:include 与 th:insert 的区别</h4>
<div th:replace="~{footer.html::frag1}"></div>
<div th:include="~{footer.html::frag1}"></div>

<h4>片段选择器的部分用法</h4>
<div th:insert="~{footer.html::/div[@id='footer1']}"></div>
<div th:insert="~{footer.html:://div#footer2}"></div>
<!--<div th:insert="~{footer.html::#footer2}"></div>-->
<div th:insert="~{footer.html::span[class='content']}"></div>
<div th:insert="~{footer.html:://span[class='content'][0]}"></div>
<div th:insert="~{footer.html:://span.content}"></div>
<!--<div th:insert="~{footer.html::.content}"></div>-->
<!--<div th:insert="~{footer.html::.content[0]}"></div>-->
<div th:insert="~{footer.html::span%frag1}"></div>
<!--<div th:insert="~{footer.html::%frag1}"></div>-->
<h4>含有变量的片段引用</h4>
```

```
<div th:insert = "~{footer.html::welcome('小明')}"></div>
```

IDEA 运行后，查看网页源代码，代码如下：

```
<h4>th:insert 引用片段</h4>
引用指定模板的整个内容
<div>
  <span>frag1 </span>
  <span>frag2 </span>
  <div id = "footer1"> footer1 </div>
  <div>
      <div id = "footer2"> footer2 </div>
  </div>
  <div>
      <span class = "content"> footer3 </span>
      <span class = "content"> footer4 </span>
  </div>
  <div>
      <span>hello,null </span>
  </div>
</div>
引用指定模板的片段
<div><span>frag1 </span></div>
引用本页面的片段
<div><div>frag3 </div></div>
<div><div>frag3 </div></div>
<div>frag3 </div>

<h4>th:replace、th:include 与 th:insert 的区别</h4>
<span>frag1 </span>
<div>frag1 </div>

<h4>片段选择器的部分用法</h4>
<div><div id = "footer1"> footer1 </div></div>
<div><div id = "footer2"> footer2 </div></div>
<div><span class = "content"> footer3 </span><span class = "content"> footer4 </span></div>
<div><span class = "content"> footer3 </span></div>
<div><span class = "content"> footer3 </span><span class = "content"> footer4 </span></div>
<div><span>frag1 </span></div>

<h4>含有变量的片段引用</h4>
<div>
  <div>
  <span>hello,小明</span>
  </div>
</div>
```

3.4 实现基于 Thymeleaf 的 Web 应用


视频讲解

3.3 节已经对 Thymeleaf 的基本语法及使用进行了介绍，接下来通过一个项目重点讲解 Spring Boot 与 Thymeleaf 模板引擎的整合使用。

(1) 创建 Spring Boot Web 应用 springboot0304。

(2) 选择相应依赖。

使用 Spring Initializr 方式创建项目,并在 Dependencies 依赖选择中选择 Web 模块下的 Web 场景依赖和 Template Engines 模块下的 Thymeleaf 场景依赖,然后根据提示完成项目的创建。引入的场景依赖效果如图 3-16 所示。

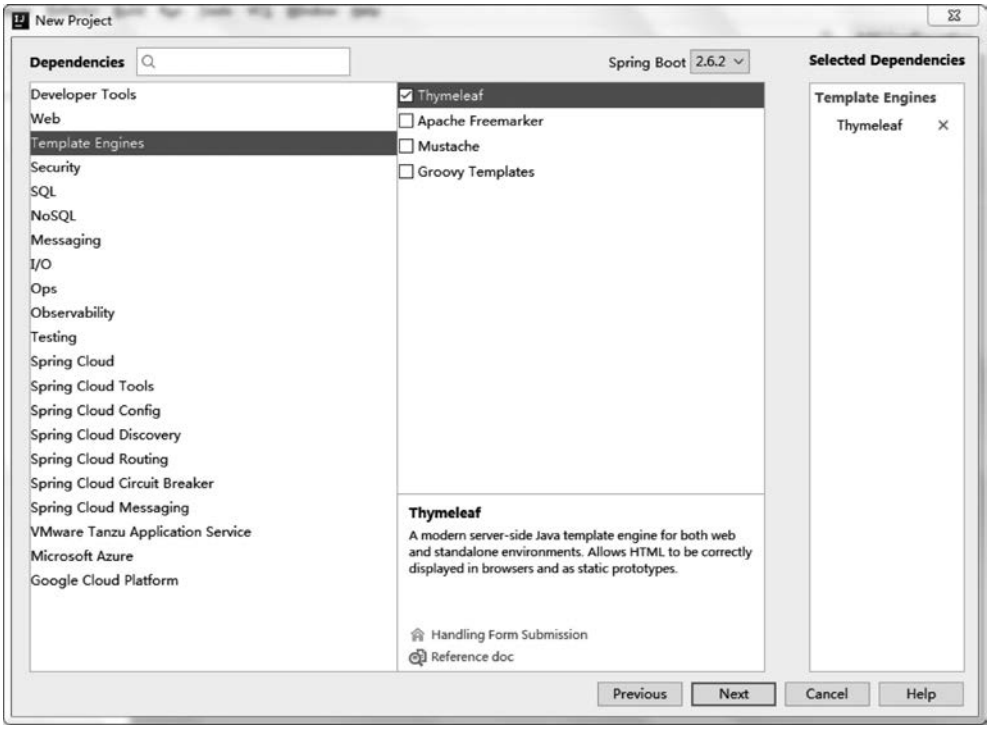


图 3-16 引入的场景依赖效果图

项目创建成功后,在 pom.xml 文件中就添加了 Thymeleaf 依赖,代码如下。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

(3) 添加 Thymeleaf 配置。

在 application.properties 文件中,添加相关的配置项,代码如下。

```
#开发配置为 false,避免修改模板后重启服务器
spring.thymeleaf.cache=false
#模板的模式,支持 HTML、XML、TEXT、JAVASCRIPT
spring.thymeleaf.mode=HTML5
#编码,可不用配置
spring.thymeleaf.encoding=utf-8
#内容类别,可不用配置
spring.thymeleaf.content-type=text/html
```

(4) 整理脚本样式静态文件。

为了保证页面美观效果,需要在项目的静态资源文件夹 static 下创建相应的目录和文件,具体代码见源文档。

（5）创建 Book 实体类、BookController 控制器及对应的 BookDAO 数据层等，具体代码见源文档。

（6）在 resources/templates 目录下创建模板片段 header.html 和 footer.html，在此展示 header.html 代码。footer.html 代码比较简单，具体代码见源文档。

```
<div align="center" th:fragment="head">
<table width="700" border="1" cellpadding="0" cellspacing="0" bordercolor="#CCCCCC">
    <tr>
        <td width="111" height="87" align="center"><img src="images/logo.gif"
width="91" height="72"><br></td>
        <td width="509" align="center"><img src="images/banner.bmp" width="510"
height="80"></td>
        <td align="center" width="70"><a href="">网站管理</a></td>
    </tr>
</table>
<table width="700" border="1" cellpadding="0" cellspacing="0" bordercolor="#CCCCCC">
    <tr>
        <td width="164"><div align="center"><a href="">首页</a></div></td>
        <td width="224"><div align="center"><a href="">在线购书</a></div></td>
        <td width="304" align="right"><div align="center"><a href="">我的购物车
</a></div><div align="center"></div></td>
        <td width="304" align="right"><div align="center"><a href="">我的订单</a>
</div></td>
        <td width="304" align="right"><div align="center"><span th:text="|当前日
期:${dateString}|"/></div></td>
    </tr>
</table>
</div>
```

（7）创建模板页面 booklist.html 文件，并引入 head 和 foot 代码片段，主要展示所有图书的信息，具体代码如下。

```
<div th:insert="~{header.html::head}"></div>
<div align="center"><a class="title">在线购书</a><br>
  <table width="700" border="1" cellpadding="0" cellspacing="0">
   <tr>
    <td width="43%" height="26"><div align="center">书名</div></td>
    <td width="17%"><div align="center">作者</div></td>
    <td width="17%"><div align="center">出版社</div></td>
    <td width="5%"><div align="center">价格</div></td>
    <td width="9%"><div align="center">操作</div></td>
  </tr>
  <div th:each="book:${bookList}">
    <tr>
        <td height="32"><div align="center"><span th:text="${book.getTitle()}">
</span></div></td>
        <td><div align="center"><span th:text="${book.getAuthor()}"></span></div></td>
        <td><div align="center"><span th:text="${book.getPublisher()}"></span>
</div></td>
        <td><div align="center"><span th:text="${book.getPrice()}"></span></div></td>
        <td><div align="center"><a href=""><img src="images/buy.gif" width="45"
height="16" border="0"></a></div></td>
    </tr>
  </div>
```

```
    </table>
  </div>
  <div th:replace="~{footer.html::foot}"></div>
```

(8) 启动测试。

在浏览器地址栏中输入 http://localhost:8080/login 后,就可以进入网上书店的首页,如图 3-17 所示。

图 3-17 运行代码片段效果图


视频讲解

3.5 Spring Boot 中的页面国际化实现

什么是国际化? 例如,dubbo.apache.org 是一个默认英文的网站,单击右上角的中文就会切换成中文网站,这就是国际化。

在 Spring Boot 的 Web 应用中实现页面信息国际化非常简单,下面通过示例讲解国际化的实现过程。

(1) 创建 Spring Boot Web 应用 springboot0305。

(2) 选择 Web 场景依赖和 Thymeleaf 场景依赖。

(3) 编写国际化配置文件。

① 在 resources 资源文件下新建一个 i18n 目录,用于存放国际化配置文件。选中此文件夹并右击,在弹出的命令选择器中选择 New→Resource Bundle 命令,如图 3-18 所示。

Resource Bundle 是一堆前缀名称相同但后缀名称不同的属性文件的集合,且至少包含两个有着相似前缀名称的属性文件,如 book_en_US.properties 和 book_zh_CN.properties。Resource Bundle 从字面上理解其实就是资源包,为了方便统一管理繁多的国际化文件,只是在 IntelliJ IDEA 内显示上多了一层名为 Resources 的 Resource Bundle 目录,但在实际物理目录下的 book_*.properties 等文件仍在 i18n 目录下。

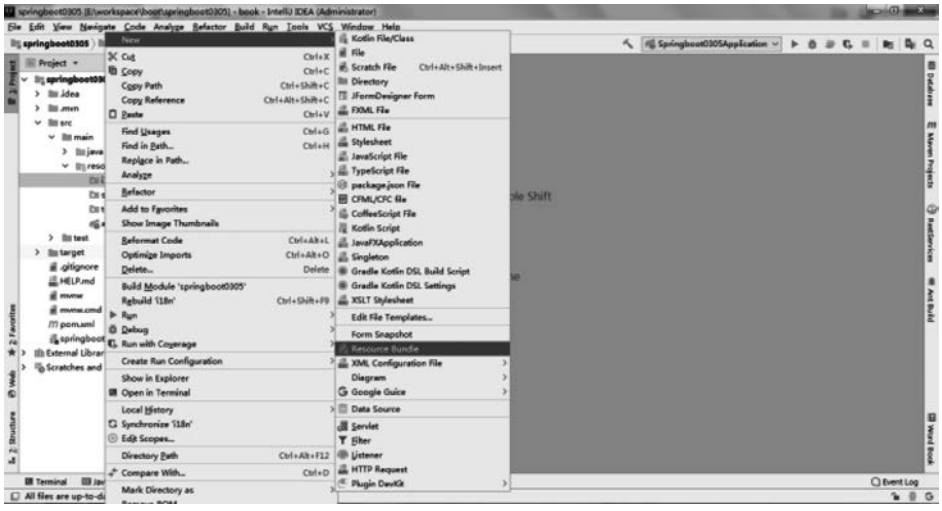


图 3-18　选择 Resource Bundle 命令

② 选择 Resource Bundle 后会弹出如图 3-19 所示的窗口，在窗口中填写 Resource Bundle 的基础名称 book，勾选 User XML-based properties files 则会创建 XML 格式的属性文件。Project locale 表示项目里已经存在的区域，Locales to add 表示添加相应的区域，单击右边的＋号即可添加，多个区域用英文的逗号隔开，如图 3-20 所示。

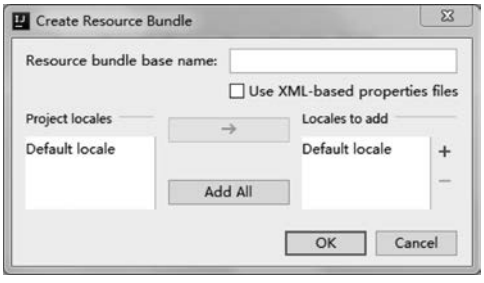


图 3-19　新建 Resource Bundle

图 3-20　添加 Locales

③ 区域添加完成后，可以在 Locales to add 看到已经添加的区域，如图 3-21 所示。单击 OK 按钮，生成 Resource Bundle，如图 3-22 所示。

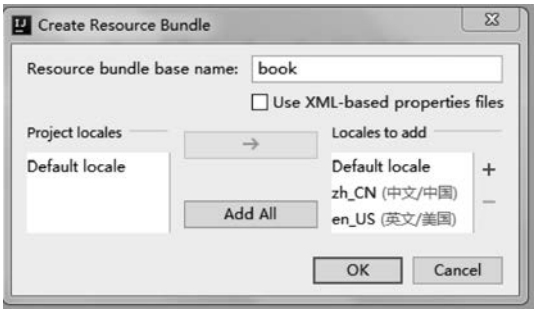


图 3-21　添加完成区域效果

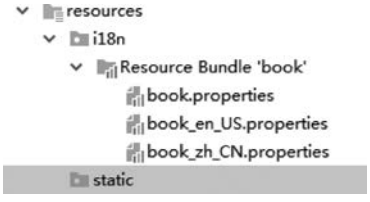


图 3-22　生成 Resource Bundle

book. properties 为自定义默认语言配置文件，book_zh_CN. properties 为自定义中文国际化文件，book_en_US. properties 为自定义英文国际化文件。

需要说明的是，Spring Boot 默认识别的语言配置文件为类路径 resources 下的 messages.properties，其他语言国际化文件的名称必须严格按照“文件前缀名_语言代码_国家代码.properties”的形式命名。

④ 点进 3 个文件中的任意一个，单击左下角的可视化工具，同时对 3 个文件进行编写，如图 3-23 所示。先单击 Resource Bundle 再单击＋号进行属性的添加，只需要添加一个即可，另外两个会自动加上。

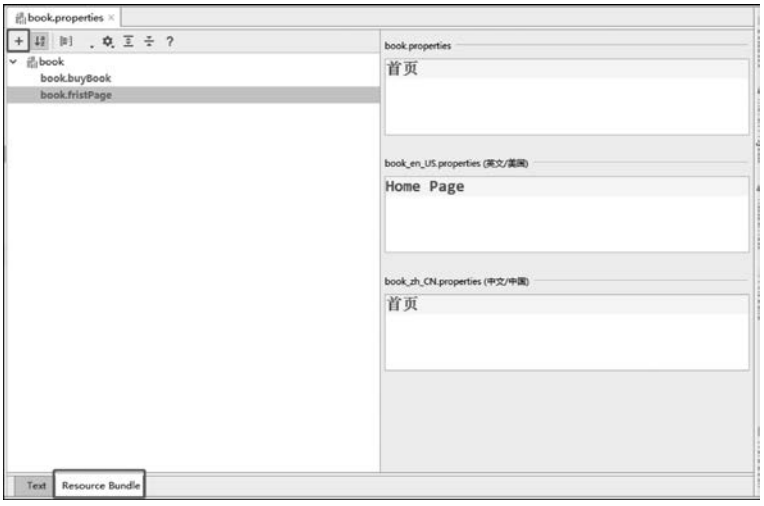


图 3-23　利用可视化工具添加属性

⑤ 查看配置文件。

book.properties 和 book_zh_CN.properties 配置文件：

```
book.author = 作者
book.bookname = 书名
book.buyBook = 在线购书
book.cart = 购物车
book.date = 当前日期
book.development = 科技工作室开发
book.fristPage = 首页
book.management = 网站管理
book.operation = 操作
book.order = 订单
book.price = 价格
book.publisher = 出版社
```

book_en_US.properties 配置文件：

```
book.author = Author
book.bookname = BookName
book.buyBook = Purchase Books
book.cart = hopping Cart
book.date = Date
book.development = Technology Studio Development
book.fristPage = Home Page
book.management = Website Management
book.operation = Operation
```

```
book.order = Order
book.price = Price
book.publisher = Publisher
```

(4) 启用国际化配置。

Spring Boot 对国际化的自动配置涉及 MessageSourceAutoConfiguration 类的方法,这里 Spring Boot 已经自动配置好了管理国际化资源文件的组件 ResourceBundleMessageSource。资源文件 messages 存放在 i18n 目录下,所以要配置 messages 的路径。在 application.properties 文件里配置国际化信息,代码如下。

```
# 关闭 Thymeleaf 模板缓存
spring.thymeleaf.cache = false
# 绑定国际化配置文件
spring.messages.basename = i18n.book
```

注意:如果配置文件可以直接放在类路径下的 messages.properties 中,则不用将配置文件再在 application.properties 里配置。

(5) 以 springboot0304 为基础,创建相应的 JS、CSS、图片、对应的类和页面。修改 head.html 页面,并获得国际化信息,代码如下。

```
<table width = "700" border = "1" cellpadding = "0" cellspacing = "0" bordercolor = "#CCCCCC">
    <tr align = "center">
        <td width = "164"><a href = "">[[#{book.firstPage}]]</a></td>
        <td width = "224"><a href = ""><span th:text = "#{book.buyBook}"></span></a></td>
        <td width = "304"><a href = ""><span th:text = "#{book.cart}"></span></a></td>
        <td width = "304"><a href = ""><span th:text = "#{book.order}"></span></a></td>
        <td width = "304"><span th:text = "#{book.date} + ':' + ${dateString}"/></td>
    </tr>
</table>
```

代码中使用 Thymeleaf 模块的#{}消息表达式设置了国际化展示的一部分信息。当对购物车进行国际化设置时,需要在标签中设置国际化信息。当对首页进行国际化设置时,这里使用了行内表达式[[#{book.firstPage}]]动态获取国际化文件中的信息。

此时国际化配置完成了一半,需要根据浏览器的请求头 Accept-Language 的语言进行页面的回显,但与程序无关,如图 3-24 所示。

图 3-24 Accept-Language 的语言为中文

在浏览器设置中切换需要显示的语言为英语置顶，重新刷新页面就可以显示出英文的信息，如图 3-25 所示。

图 3-25　Accept-Language 的语言为英文

(6) 实现国际化的方式有两种，一种是根据浏览器的语言变化；另一种是单击链接来改变语言。接下来需要手动对页面语言进行自定义切换。修改 foot.html 页面，增加中英文切换选项，代码如下所示。

```
    <div align = "center" th:fragment = "foot">
<table width = "700" border = "0" cellspacing = "2" cellpadding = "2">
    <tr>
        <td align = "center"><span th:text = "#{book.development}"></span><br>
            CopyRight@2004 <br>
            Email:<a href = "mailto:123@163.net">123@163.net</a></td>
    </tr>
    <tr>
        <td align = "center">
            <a th:href = "@{/login(l = 'zh_CN')}">中文</a>    
            <a th:href = "@{/login(l = 'en_us')}">English</a>
        </td>
    </tr>
</table>
    </div>
```

LocaleResolver（获取区域信息对象）解析器位于国际化的 Locale（区域信息对象），在 Web MVC 自动配置文件中可以看到 Spring Boot 的默认配置。

```
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
  //容器中没有就自己配置,有的话就使用已有配置
     if (this.mvcProperties.getLocaleResolver() == WebMvcProperties.LocaleResolver.FIXED)
     {   return new FixedLocaleResolver(this.mvcProperties.getLocale());     }
  //接收头国际化分解
       AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();
       localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
       return localeResolver;
}
```

AcceptHeaderLocaleResolver 类中获取 Locale 的方法如下：

```
public Locale resolveLocale(HttpServletRequest request) {
//默认为根据请求头带来的区域信息获取 Locale 进行国际化
    Locale defaultLocale = this.getDefaultLocale();
    if (defaultLocale != null && request.getHeader("Accept-Language") == null) {
        return defaultLocale;
    } else {
        Locale requestLocale = request.getLocale();
        List<Locale> supportedLocales = this.getSupportedLocales();
        if (!supportedLocales.isEmpty() && !supportedLocales.contains(requestLocale)) {
            Locale supportedLocale = this.findSupportedLocale(request, supportedLocales);
            if (supportedLocale != null) {
                return supportedLocale;
            } else {
                return defaultLocale != null ? defaultLocale : requestLocale;
            }
        } else {
            return requestLocale;
        }
    }
}
```

如果想单击链接让国际化资源生效，就需要让自己的 Locale 生效。此时在 it.com.boot.springboot03_05.config 包下创建一个用于定制国际化功能区域信息解析器的自定义配置类 MyLocaleResolver，通过链接获取携带区域的信息。

```
public class MyLocaleResolver implements LocaleResolver {
//解析请求
    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        String language = request.getParameter("l");      //获取 header 页面传过来的参数
    String head = request.getHeader("Accept-Language");
    //Accept-Language: zh-CN, zh;q=0.9,en-US;q=0.8,en;q=0.7
          Locale locale;
        if (!StringUtils.isEmpty(language)) {              //如果请求链接不为空
              //由于格式为 en_us,所以用"_"分割请求参数
    String[] split = language.split("_");
      //截取后把新的属性(国家,地区)封装给 locale
    locale = new Locale(split[0], split[1]);
        }
        else {
            String[] splits = head.split(",");
            String[] split = splits[0].split("-");
            locale = new Locale(split[0], split[1]);
        }
    return locale;
    }
    @Override
    public void setLocale(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, Locale locale) {
    }
}
```

(7) 为了让区域化信息能够生效，需要再配置一下这个组件。修改 MyLocaleResolver

类为配置类，并在该类下重新注册一个类型为 LocaleResolver 的 Bean 组件，这样就可以覆盖默认的 LocaleResolver 组件。

```
@Configuration
public class MyLocaleResolver implements LocaleResolver {
    //省略解析请求
    @Bean
    public LocaleResolver localeResolver()
    {
        return new MyLocaleResolver();
    }
}
```

（8）重启服务器，发现单击按钮也可以实现成功切换，中英文页面分别如图 3-26 和图 3-27 所示。

图 3-26　中文页面

图 3-27　英文页面

3.6 Spring Boot 集成 Spring MVC

如果想在 Spring Boot 中自己定义一些 Handler、Interceptor、ViewResolver 和 MessageConverter 等，该如何完成呢？在 Spring Boot 1.5 版本中，依靠重写 WebMvcConfigurerAdapter 的方法来添加自定义拦截器和消息转换器等。在 SpringBoot 2.0 版本后，该类被标记为@Deprecated，因此只能靠 WebMvcConfigurer 接口来实现。

WebMvcConfigurer 是一个接口，提供很多自定义的拦截器，如跨域设置、类型转换器等。此接口为开发者预想了很多拦截层面的需求，方便开发者自由选择使用。

Spring Boot 推荐使用 WebMvcConfigurer 接口的实现类来实现代码配置，具体代码如下。

```
@Configuration
public class MyConfigurer implements WebMvcConfigurer {}
```

3.6.1 配置自定义拦截器 Interceptor

Java 里的拦截器是动态拦截 Action 调用的对象，它提供了一种机制，可以使开发者在一个 Action 执行的前后执行一段代码，也可以在一个 Action 执行前阻止其执行。此外，拦截器还提供了一种可以提取 Action 中可重用部分代码的方式。在 Spring AOP 中，拦截器用于在某个方法或字段被访问前进行拦截，然后在其前后加入某些操作。

下面使用 WebMvcConfigurer 接口中的 addInterceptors()方法注册自定义拦截器。

(1) 以项目 springboot0301 和 springboot0305 为基础，创建项目 springboot0306。

(2) 在 config 包下创建一个自定义拦截器类 MyInterceptor，并编写拦截业务代码。实现拦截器需要一个 HandlerInterceptor 接口的实现类，并重写其中的 3 个方法，具体代码如下。

```
@Component
public class MyInterceptor implements HandlerInterceptor {
      //在请求处理前进行调用(Controller 方法调用前)
    @Override
    public boolean preHandle (HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        Object user = request.getSession().getAttribute("user");
        if (user == null) {
            response.sendRedirect("/index");
            return false;
        }
        return true;
    }
    //请求处理后进行调用(Controller 方法调用后),但需要在视图被渲染前
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
    }
    //在整个请求结束后被调用,即在 DispatcherServlet 渲染了对应的视图后执行(主要是用于进
行资源清理工作)
```

```
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
    }
}
```

只有拦截器的 preHandle 方法返回 true,postHandle、afterCompletion 才有可能被执行。如果 preHandle 方法返回 false,则该拦截器的 postHandle、afterCompletion 必然不会被执行。

(3) 在实现拦截器后,还需要将拦截器注册到 Spring 容器中。在 config 包下创建一个实现实现 WebMvcConfigurer 接口的配置类 MyConfigurer,覆盖其 addInterceptors(InterceptorRegistry registry)方法。将 Bean 注册到 Spring 容器中,可以选择@Component 或@Configuration,具体代码如下。

```
@Configuration
public class MyConfigurer implements WebMvcConfigurer {
@Autowired
private MyInterceptor myInterceptor;
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(myInterceptor)
                .addPathPatterns("/**")                    //所有路径都被拦截
                .excludePathPatterns("/main")              //main 请求不被拦截
                .excludePathPatterns("/index")
                .excludePathPatterns("/**/*.js")           //JS 静态资源不被拦截
                .excludePathPatterns("/**/*.css");         //CSS 静态资源不被拦截
    }
}
```

其中,addPathPatterns 用于设置拦截器的过滤路径规则,excludePathPatterns 用于设置不需要拦截的过滤规则。

(4) 修改 BookController,具体代码如下。

```
@Controller
public class BookController {
    @RequestMapping("/login")
    public String findAll(Model model)
    {
        Calendar calendar = Calendar.getInstance();
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        BookDAO bookDao = new BookDAO();
        List bookList = bookDao.queryAllBook();
        model.addAttribute("dateString", dateFormat.format(calendar.getTime()));
        model.addAttribute("bookList", bookList);
        return "booklist";
    }
    @RequestMapping("/index")
    public String toIndex() {
        return "index";
    }
    @RequestMapping("/main")
    public String toMain(HttpSession session) {
        session.setAttribute("user", "userLoginSuccess");
```

```
        return "redirect:login";
    }
}
```

(5) 运行程序。当访问 http://localhost:8080/login 路径时,系统会自动跳转到用户登录页面,这就说明此处定制的自定义拦截器生效了。由于对 CSS 和 JS 进行了不拦截设置,所以在登录页面中可以进行用户名和密码的验证。只有用户信息填入完整,单击“登录”按钮,才能回到图书首页。

3.6.2 跳转指定页面

用传统方式写 Spring MVC 时,如果需要访问一个页面,必须要写 Controller 类,然后再写一个方法跳转到页面,这样会很麻烦。例如,“/index”可以通过重写 WebMvcConfigurer 中的 addViewControllers 方法达到效果。

addViewControllers 方法可以实现将一个无业务逻辑的请求直接映射为视图,不需要编写控制器,从而简化了页面跳转。

修改 3.6.1 节的 MyConfigurer 配置类,在其实现类中重写 addViewControllers 方法,具体代码如下。

```
@Configuration
public class MyConfigurer implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/index").setViewName("index");
        registry.addViewController("/index.html").setViewName("index");
    }
}
```

此处重写 addViewControllers 并不会覆盖 WebMvcAutoConfiguration 中的 addViewControllers(在此方法中,Spring Boot 将“/”映射至 index.html),这意味着自己的配置和 Spring Boot 的自动配置同时有效,这也是推荐该配置方式的原因。

3.7 Spring Boot 处理 JSON 数据


视频讲解

在后台的开发过程中,不可避免地会遇到一系列对 JSON 数据的返回,此时需要提供各种各样的数据。一般情况下数据类型最常用的是 JSON 和 XML,这里主要讲解 Spring Boot 怎样进行 JSON 数据的返回及对一些特殊情况的处理。

JSON 是目前主流的前后端数据传输方式,在 Spring MVC 中使用消息转换器 HttpMessageConverter 对 JSON 的转换提供了很好的支持,在 Spring Boot 中对相关配置做了进一步简化。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

上述依赖中默认加入了 Jackson-databind 作为 JSON 处理器,此时不需要添加额外的 JSON 处理器就可以返回 JSON。这是 Spring Boot 自带的处理方式,如果采用这种方式,对

于字段忽略、日期格式化等都可以使用注解实现。

```
public class Book {
  @JsonIgnore                                          //过滤该属性
  protected Float price;
  @JsonFormat(pattern = "yyyy-MM-dd")                  //格式化输出该属性
  private Date publicationDate;
}
```

Spring Boot 在处理 JSON 数据时,需要用到两个重要的 JSON 格式转换注解,分别是@ResponseBody 和@RequestBody。

(1) @ResponseBody 注解的作用:将 controller 的方法返回的对象通过适当的转换器转换为指定的格式,并将其写入 response 对象的 body 区,通常用来返回 JSON 数据或 XML 数据。

(2) @RequestBody 注解的作用:在形参列表上,将前台发送的固定格式的数据(XML 数据或 JSON 数据等)封装为对应的 JavaBean 对象,封装时使用系统默认配置的 HttpMessageConverter(消息转换器)进行解析,然后封装到形参上。

常见的 JSON 处理器除了 Jackson-databind 外,还有 Gson 和 FastJson 两种,在使用时需要添加相应的依赖。这里以 FastJson 为例,讲解 JSON 处理器的使用方法。

fastjson.jar 是阿里巴巴开发的一款专门用于 Java 开发的包,可以方便地实现 JSON 对象与 JavaBean 对象的转换、JavaBean 对象与 JSON 字符串的转换,以及 JSON 对象与 JSON 字符串的转换。

下面通过示例讲解 JSON 数据的处理过程。

(1) 以项目 springboot0306 为基础,创建项目 springboot0307。

(2) 打开 pox.xml 文件,去除 jackson-databind 依赖,加入 FastJson 依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
       <exclusion>
          <groupId>com.fasterxml.jackson.core</groupId>
          <artifactId>jackson-databind</artifactId>
       </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.75</version>
</dependency>
```

(3) FastJson 需要自己配置 HttpMessageConverter,打开 MyConfigurer 配置类,在其实现类中,重写 configureMessageConverters 方法,具体代码如下。

```
@Override
public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    //创建 FastJson 的消息转换器
    FastJsonHttpMessageConverter convert = new FastJsonHttpMessageConverter();
```

```
    //创建 FastJson 的配置对象
    FastJsonConfig config = new FastJsonConfig();
    //对 JSON 数据进行格式化
    formatting(SerializerFeature.PrettyFormat);
    convert.setFastJsonConfig(config);
    converters.add(convert);
}
```

(4) 修改“/login”请求,使其返回 JSON 格式的数据。

```
@RequestMapping("/login")
@ResponseBody
public String findAll( )
{
    Calendar calendar = Calendar.getInstance();
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    BookDAO bookDao = new BookDAO();
    List bookList = bookDao.queryAllBook();
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("dateString", dateFormat.format(calendar.getTime()));
    jsonObject.put("booklist",bookList );
    return jsonObject.toJSONString();
}
```

(5) 修改 booklist.html 页面,使其在页面中显示 AJAX 请求到的数据。

```
<script type="application/javascript">
  $(function () {
     var tbody = document.getElementById("tbody-result");
     var date = document.getElementById("date");
      $.ajax({
               url: "login",
               type: "get",
  //如果当前为 JSON 格式,需要修改此属性,定义发送请求的数据格式为 JSON 字符串
               contentType: "application/json; charset=UTF-8",
               //定义回调响应的数据格式为 JSON 字符串,此属性可以省略
               dataType: "json",
               //成功响应的结果
               success: function (data) {
                   var inf = "";
                   var bookList = data.booklist
                   for (var i = 0; i < bookList.length; i++) {
                       var book = bookList[i];
                       inf += "<tr><td height='32'><div align='center'>" + book.title +
                           "</div></td><td><div align='center'>" + book.author +
                           "</div></td><td><div align='center'>" + book.publisher +
                           "</div></td><td><div align='center'>" + book.price +
                           "</div></td><td><div align='center'><a href=''><img src=
'images/buy.gif' width='45' height='16' border='0'></a></div></td></tr>"
                   }
                   tbody.innerHTML = inf;
                   date.innerText += data.dateString;
               },
```

```
            error: function () {
                alert(("操作失败"));
            }
        }
    );
})
</script>

<table width = "700" border = "1" cellpadding = "0" cellspacing = "0">
    <tr>省略标题行</tr>
    <tbody id = "tbody - result"></tbody>
</table>
```

(6) 运行程序,效果图如图 3-26 所示。


视频讲解

3.8 Spring Boot 实现 RESTful 风格的 Web 应用

RESTful 架构风格是目前最流行的一种架构风格,它结构清晰、符合标准、易于理解、扩展方便,所以在 Web 开发中经常被使用。REST 的全称是 Representational State Transfer,译作“表现层状态转化”。

RESTful 架构是对 MVC 架构改进后所形成的一种架构,通过使用事先定义好的接口与不同的服务联系起来。在 RESTful 架构中,浏览器使用 POST、DELETE、PUT 和 GET 这 4 种请求方式分别对指定的 URL 资源进行增、删、改、查操作。因此,RESTful 通过 URI 实现对资源的管理和访问,具有扩展性强、结构清晰的特点。

RESTful 架构将服务器分成前端服务器和后端服务器两部分,前端服务器为用户提供无模型的视图,后端服务器为前端服务器提供接口。浏览器向前端服务器请求视图,通过视图中包含的 AJAX 函数发起接口请求并获取模型。

RESTful 是一种对 URL 进行规范的编码风格,通常一个网址对应一个资源,访问形式类似 http://xxx.com/xx/{id}/{id}。

举个例子,在某购物网站上买手机时会有很多品牌选择,而每种品牌下又有很多型号,此时 https://mall.com/mobile/iPhone/6 可以代表 iphone6,而 https://mall.com/mobile/iPhone/7 和 https://mall.com/mobile/iPhone/8 分别代表 iphone7 和 iphone8。

在进行 Web 开发的过程中,method 常用的值是 GET 和 POST。可事实上,method 值还可以是 PUT 和 DELETE 等其他值。既然 method 值如此丰富,那么就可以考虑使用同一个 URL,但是约定不同的 method 来实施不同的业务,这就是 RESTful 的基本考虑。

CRUD 是最常见的操作,在使用 RESTful 风格前,通常的增加做法如下:

```
/addCategory?name = xxx
```

可是在使用 RESTful 风格后,增加做法就变为:

```
/categories
```

传统风格和 RESTful 风格的对比如表 3-1 所示,URL 使用相同的"/categories"语句,区别只是在于 method 值不同,服务器根据 method 的不同来判断浏览器期望进行的业务行为。

表 3-1 传统风格和 RESTful 风格的对比

功能	传统风格		RESTful 风格	
	URL	method	URL	method
增加	/addCategory?name=xxx	POST	/categories	POST
删除	/deleteCategory?id=123	GET	/categories/123	DELETE
修改	/updateCategory?id=123&name=yyy	POST	/categories/123	PUT
获取	/getCategory?id=123	GET	/categories/123	GET
查询	/listCategory	GET	/categories	GET

为了实现 RESTful API 接口，Spring Boot 提供了以下注解，对请求参数和返回数据格式进行封装，方便用户快速开发。

(1) @RestController 一般用于 Controller 类上，指定所有接口返回的数据都是 text/json 格式。

(2) @ResponseBody 用于方法上，指定接口返回 text/json 格式，如果使用@RestController 就没有必要使用@ResponseBody。

(3) @GetMapping 用于方法上，是一个组合注解，与@RequestMapping(method=RequestMethod.GET)作用一致。

(4) @PostMapping 用于方法上，是一个组合注解，与@RequestMapping(method=RequestMethod.POST)作用一致。

(5) @PutMapping 用于方法上，是一个组合注解，与@RequestMapping(method=RequestMethod.PUT)作用一致。

(6) @DeleteMapping 用于方法上，是一个组合注解，与@RequestMapping(method=RequestMethod.DELETE)作用一致。

(7) @RequestParam 用于方法上，映射请求参数到 Java 方法的参数，当前端传递的参数与后台自定义的参数不一致时，可以使用 name 属性来标记。

(8) @PathVariable 用于方法上，映射 URL 片段到 Java 方法的参数。

下面通过一个例题来讲解 Spring Boot 如何实现 RESTful。

(1) 以项目 springboot0307 为基础，创建项目 springboot0308。

(2) 创建 Controller 类。

```
@RestController
public class UserController {
    @GetMapping("/user/{username}")
    public String toMain(@PathVariable String username, String password,HttpSession session ) {
         JSONObject jsonObject = new JSONObject();
        if (username.equals("admin")&&password.equals("123456")) {
            session.setAttribute("user", "userLoginSuccess");
            jsonObject.put("flag", true);
        }
        else
            jsonObject.put("flag", false);
            return jsonObject.toJSONString();
    }
    @DeleteMapping("/user/{username}")
    public String deleteUser( @PathVariable String username ,HttpSession session ) {
```

```
        JSONObject jsonObject = new JSONObject();
        System.out.println(username);
        jsonObject.put("flag", "删除成功");
        return jsonObject.toJSONString();
    }
    @PostMapping("/user")
    public String addUser( @RequestBody String json ) {
        JSONObject jsonObject = JSONObject.parseObject(json);
        String username = jsonObject.getString("username");
        String password = jsonObject.getString("password");
        System.out.println(username);
        System.out.println(password);
        jsonObject.put("flag", "增加成功");
        return jsonObject.toJSONString();
    }
    @PutMapping("/user")
    public String updateUser(User user) {
        JSONObject jsonObject = new JSONObject();
        System.out.println(user);
        jsonObject.put("flag", "修改成功");
        return jsonObject.toJSONString();
    }
}
```

(3) 编写相应的 AJAX 提交,这里只列出查询和添加两类,修改和删除类似,具体代码如下。

```
function checkform() {
   var username = document.getElementById("userid").value;
   var password = document.getElementById("password").value;
   var json = {"password": password};
   $.ajax({
          url: "user/" + username,
          type: "get",
          data: json,
          dataType: "json",
          //成功响应的结果
          success: function (data) {
              if (data.flag) {
                  alert("登录成功");
                  window.location = "tologin";
              }
              else
                  alert("登录失败")
          },
          error: function () {
              alert(("操作失败"));
          }
      }
   );
}
  function addform() {
     var username = $("#userid").val();
     var password = $("#password").val() ;
     alert(username)
     var json = {"username":username,"password": password};
```

```
        $.ajax({
              url: "user" ,
              type: "post",
              data: JSON.stringify(json),
              contentType: "application/json; charset = UTF - 8",
              dataType: "json",
              //成功响应的结果
              success: function (data) {
                  if (data.flag) {
                     alert("添加成功");
                  }
                  else
                     alert("添加失败")
              },
              error: function () {
                  alert(("操作失败"));
              }
          }
      );
}
```

(4) 运行程序,效果图如图 3-28～图 3-31 所示。

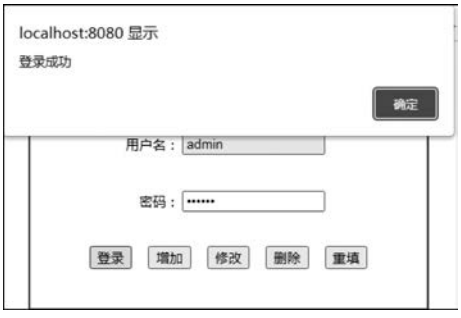


图 3-28　登录成功效果图

图 3-29　添加成功效果图

图 3-30　修改成功效果图

图 3-31　删除成功效果图

3.9　Spring Boot 文件上传和下载


视频讲解

在 Web 应用中,对多媒体文件的操作非常常见,文件的上传与下载尤为如此。本节将通过项目的新增图片和附件的上传、下载模块,带领大家熟悉该类型功能的开发流程。

3.9.1 文件上传

Spring Boot 通常为服务提供者,但是在一些场景中还是要用到文件上传下载这种“非常规”操作。如何在 Spring Boot 中实现文件的上传与下载功能呢? 回顾一下在 Spring MVC 中的操作,需要在 Spring MVC 的配置文件中增加文件上传的 Bean 的配置。

```
<bean id = "multipartResolver"
class = "org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
```

在 Spring MVC 中完成 Bean 配置后,在后台对应的处理方法中就可以直接获取文件的输入流。而对于 Spring Boot 来说,不需要配置文件上传的解析类,因为 Spring Boot 已经注册好了。

Java 中的文件上传涉及两个组件,一个是 CommonsMultipartResolver,另一个是 StandardServletMultipartResolver。CommonsMultipartResolver 使用 commons-fileupload 来处理 multipart 请求,而 StandardServletMultipartResolver 则基于 Servlet 3.0 来处理 multipart 请求。因此,若使用 StandardServletMultipartResolver 处理请求,则不需要添加额外的 JAR 包。Tomcat 7.0 开始支持 Servlet 3.0,而 Spring Boot 2.6.3 内嵌的 Tomcat 为 Tomcat 9.0.56,因此可以直接使用 StandardServletMultipartResolver。在 Spring Boot 提供的文件上传自动化配置类 MultipartAutoConfiguration 中,默认也是采用 StandardServletMultipartResolver。因此,在 Spring Boot 中上传文件甚至可以做到零配置,具体上传过程如下。

(1) 创建 Spring Boot Web 应用 springboot0309,并添加 spring-boot-starter-web 和 spring-boot-starter-thymeleaf 依赖。

(2) 打开 application.properties 文件,添加如下配置,完成对上传文件大小的限制。

```
#设置单个文件大小
spring.servlet.multipart.max-file-size = 5MB
#设置单次请求文件的总大小
spring.servlet.multipart.max-request-size = 500MB
```

(3) 创建 User 类。

```
public class User {
  private String username,password;
private String tupian;                                    //上传头像需要保存相对地址
  //省略 get 和 set 方法
}
```

(4) 创建 MyConfig 配置类。

图片上传无法立即显示而需要重启才能访问,这是因为服务器拥有保护措施,服务器不能对外部暴露真实的资源路径。为了解决该问题,需要配置虚拟路径映射进行访问。

```
@Configuration
public class MyConfig implements WebMvcConfigurer {
  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    //获取文件的真实路径
    String path = System.getProperty("user.dir") + "\\src\\main\\resources\\static\\
upload\\";
```

```
        ///upload/** 是对应 resource 下的工程目录
        registry.addResourceHandler("/upload/**").addResourceLocations("file:" + path);
    }
}
```

其中 System.getProperty("user.dir")是当前项目路径，如图 3-32 所示。

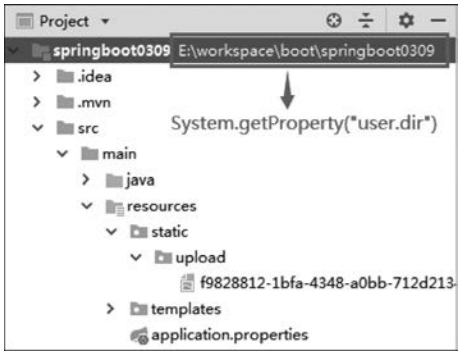


图 3-32　当前项目路径

(5) 创建上传页面。

在 resources 目录下的 static 目录中创建一个 zhuce.html 文件，完成个人信息的注册，代码如下。上传接口是/zhuce，注意请求方法是 POST，enctype 是 multipart/form-data。

```
<form th:action="@{/zhuce}" method="post" enctype="multipart/form-data">
    <label>姓名</label><input type="text" name="username"><br>
    <label>密码</label><input type="password" name="password"><br>
    <label>图片</label><input type="file" name="file"><br>
    <input type="submit" value="上传">
</form>
```

(6) 创建文件上传处理接口。

```
@Controller
public class UserController {
  @GetMapping("/zhuce")
  public String zhuce() {
    return "zhuce";
  }
  @PostMapping("/zhuce")
  public String tijiao(User user, MultipartFile file, Model model) {
    if (!file.isEmpty()) {
        String fileName = UUID.randomUUID() + "_" + file.getOriginalFilename();
        String path = System.getProperty("user.dir");
        File filePath = new File(path, "\\src\\main\\resources\\static\\upload");
        if (!filePath.isDirectory()) {
            filePath.mkdirs();
        }
        try {
            File userFile = new File(filePath, fileName);
            file.transferTo(userFile);
            user.setTupian("/upload/" + fileName);
        } catch (IOException e) {
                e.printStackTrace();
```

```
                }
            }
            model.addAttribute("user", user);
            return "permanager";
        }
    }
```

规划上传文件的保存路径为项目运行目录下的 upload 文件夹,如果该文件夹不存在,则新建 upload 文件夹。为了避免上传文件重名,这里通过随时生成 uid 给上传文件名重新命名。将上传文件保存到目标文件中,把头像的相对地址赋值给当初用户的图片属性上。

(7) 创建 permanager.html,显示上传成功的结果,代码如下。

```
<div th:object = " ${user}">
   <label>姓名</label><span th:text = " * {username}"></span><br>
   <label>密码</label><span th:text = " * {password}"></span><br>
   <label>图片</label><img th:src = "@{ * {tupian}}"><br>
</div>
```

(8) 运行程序,在浏览器地址栏中输入 http://localhost:8080/zhuce,选择要上传的头像,上传界面如图 3-33 所示。单击“上传”按钮,成功上传头像后显示注册结果,如图 3-34 所示。

图 3-33 上传界面

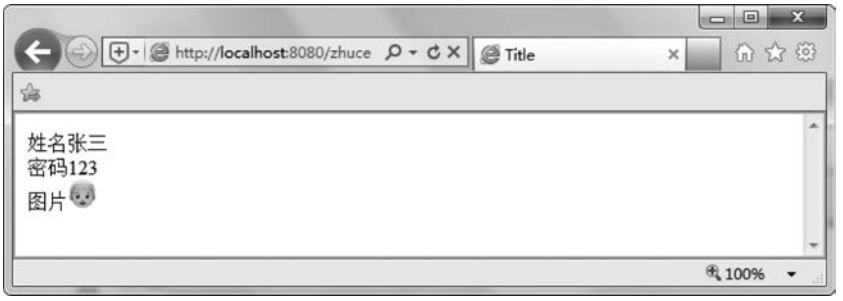


图 3-34 显示上传成功界面

3.9.2 文件下载

在 Spring Boot 项目中,可能会存在让用户下载文档的需求,比如让用户下载 readme 文档来更好地了解该项目的概况或使用方法。因此,需要为用户提供可以下载文件的 API,将用户希望获取的文件作为下载资源返回给前端。

下面以项目 springboot0309 为基础，继续分析下载功能。

(1) 配置 application. xml。

```
file.docDir: /src/main/resources/static/upload
```

该路径就是待下载文件存放在服务器上的目录，即相对路径。

(2) 将属性与实体类自动绑定。

Spring Boot 中的注解@ConfigurationProperties 可以将 application 中定义的属性与 POJO 类自动绑定，为此需要定义一个 POJO 类来进行 application 中 file. docDir 的配置绑定。

```
@ConfigurationProperties(prefix = "file")
@Component
public class FileProperties {
      private String docDir;
      public String getDocDir() {
         return docDir;
      }
      public void setDocDir(String docDir) {
         this.docDir = docDir;
      }
}
```

注解@ConfigurationProperties(prefix = "file")在 Spring Boot 应用启动时将以 file 为前缀的属性与 POJO 类绑定，也就是将 application. xml 中的 file. docDir 与 FileProperties 中的字段 docDir 进行绑定。

(3) 编写自定义配置类。

打开 MyConfig 配置类，生成下载文件所在目录的配置类，其返回值对象返回值会作为组件添加到 Spring 容器中。

```
@Autowired
private FileProperties fileProperties;
@Bean
public File getFile() {
  String path = System.getProperty("user.dir");
  File fileDir = new File(path, fileProperties.getDocDir());
  return fileDir;
}
```

(4) 编写控制器。

```
@Controller
public class UserController {
    @Autowired
    private File fileDir;
    @GetMapping("/showdownload")
    public String showDownLoad(Model model) {
        File[] fileList = fileDir.listFiles();
        model.addAttribute("fileList", fileList);
        return "download";
    }
    @GetMapping("/download")
    public void toDownLoad(String filename, HttpServletResponse response) {
```

```
            try {
                //通过流读取文件
                FileInputStream is = new FileInputStream(new File(fileDir, filename));
        //获得响应流
                ServletOutputStream os = response.getOutputStream();
              //设置响应头信息
                response.setHeader("content-disposition", "attachment;fileName=" + URLEncoder.
    encode(filename, "UTF-8"));
              //通过响应流将文件输入流读取的文件写出
                IOUtils.copy(is, os);
              //关闭流
                IOUtils.closeQuietly(is);
                IOUtils.closeQuietly(os);
            } catch (Exception ex) {
            }
        }
    }
```

(5) 编写文件下载程序。

在前端 download.html 页面中,可以使用 a 标签来下载文件,但要注意在 a 标签中定义 filename 参数来规定这是下载文件。

```
<table>
    <tr>
        <th>序号</th>
        <th>文件</th>
    </tr>
    <tr th:each="file,fileState: ${fileList}">
      <td><span th:text="${fileState.count}"></span></td>
      <td>
        <a th:href="@{/download(filename= ${file.name})}">
         <span th:with="tempFileName= ${file.name.substring(file.name.lastIndexOf('_')+1)}">
         <span th:text="${tempFileName}"></span>
         </span>
        </a>
       </td>
    </tr>
</table>
```

(6) 运行程序,在浏览器地址栏中输入 http://localhost:8080/showdownload,进入下载页面,选择所需要下载的文件进行下载,如图 3-35 所示。

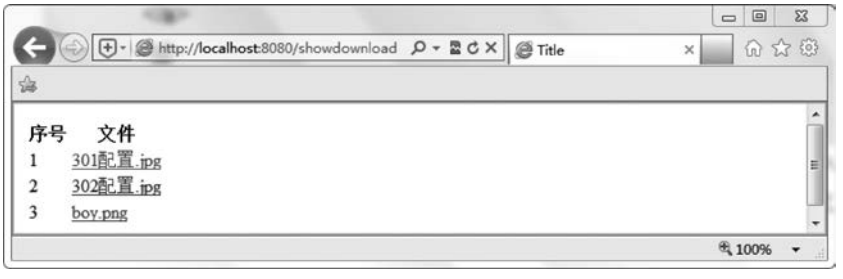


图 3-35 文件下载页面效果

3.10 Spring Boot 的异常统一处理

在程序中出现错误是难以避免的，经验再丰富的程序员编写的程序也会出现错误。在 Java 中，程序出现错误会抛出“不正常信息”(Throwable)。Throwable 又被分为“错误”(Error)和“异常”(Exception)。有别于人为失误造成的“故障”(Bug)，异常在程序中代表的是出现了当前代码无法处理的状况。例如，在一个对象不存在(值为 Null)的情况下，调用该对象的某个方法引发了空指针；用户输入了一段 URL，但并没有找到对应的资源；在一段计算过程中，0 被当作除数，等等。完善的错误处理，使程序不会意外崩溃甚至能友好地提示用户进行正确操作，这是让程序变得越发健壮的重要处理步骤。

在 Java 开发中，异常特别是检查型异常(Checked Exception)，通常需要进行 try/catch 处理。而在基于 Spring Boot 的开发过程中，异常处理有了更多的处理方式。

3.10.1 自定义 error 页面

使用 Web 应用时，在请求处理过程中发生错误是非常常见的情况。Spring Boot 提供了一个默认的映射/error，在抛出异常后，会转到该请求中处理，并且该请求有一个全局的错误页面来展示异常内容。例如，启动某个项目，在浏览器中随便输入一个访问地址，由于地址不存在，Spring Boot 会跳转到错误页面，如图 3-36 所示。

图 3-36 错误页面

虽然 Spring Boot 提供了默认的错误页面映射，但是在实际应用中，图 3-36 所示的错误页面对用户来说并不友好，需要自己实现异常提示。接下来将演示自己如何实现错误提示页面。

(1) 创建 Spring Boot 项目 springboot03_10，并添加相应的依赖。

(2) 创建控制器类，该类即为了演示错误而定制。

```
@Controller
public class TestController {
  @GetMapping("/index")
  public String toIndex()
  {
    int result = 1/0;
    return "index";
  }
}
```

(3) 定制错误页面。

在 Spring Boot 中定制错误页面有 3 种方法，分别如下。

① 使用精确匹配的方式，将错误页面命名为“错误状态码. html”，并放在模板引擎

templates/error 文件夹下。当发生访问错误时,会跳转到对应状态码的页面,如 404.html 和 500.html。也可以使用模糊匹配的方式来定义错误页面的名称,将错误页面命名为 4xx.html 和 5xx.html,以此匹配对应类型的所有错误。精确匹配的查找方式要优先于模糊匹配的方式。

② 如果 templates/error 目录下没有自定义的错误页面,那么需要在 static/error 目录下定义 4xx.html 或 5xx.html 页面。

③ 直接在 templates 目录下创建 error.html 页面,这样当访问错误或异常时,可以自动将该页面作为错误页面。

Spring Boot 为错误页面提供了以下属性。

- timestamp:时间戳。
- status:状态码。
- error:错误提示。
- exception:异常对象。
- message:异常消息。
- errors:JSR303 数据校验的错误。

在 5xx.html 页面中编写如下代码。

```
<body background="500.jpeg">
<div>
      <p><b>错误发生时间:</b>
      <span th:text="${#dates.format(timestamp,'yyyy-MM-dd')}"></span></p>
      <p><b>错误状态码:</b><span th:text="${status}"></span></p>
      <p><b>异常消息:</b>[[${message}]]</p>
      <p><b>错误提示:</b>[[${error}]]</p>
  </div>
</body>
```

(4) 运行程序。

在成功运行项目后,访问 http://localhost:8080/hello。由于服务器找不到请求的网页,Spring Boot 便会找到 src/main/resources/templates/error 目录中的 404.html 页面,运行效果如图 3-37 所示。

图 3-37 404 错误页面

继续访问 http://localhost:8080/index。该请求中计算除法发生了异常，而该方法仅抛出了 exception 异常，并没有处理异常。当 Spring Boot 发现有异常抛出且没有处理时，将在 src/main/resources/templates/error 目录下找到 500.html 页面并显示异常信息，运行效果如图 3-38 所示。

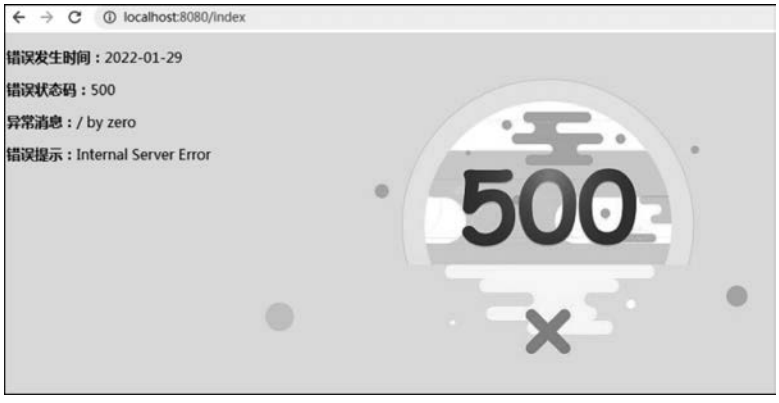


图 3-38 500 错误页面

从上述运行结果可以看出，使用自定义错误页面并没有真正处理异常，只是将异常或错误信息显示给客户端，因为在服务器控制台上同样抛出了异常，如图 3-39 所示。

```
java.lang.ArithmeticException: / by zero
    at it.com.boot.springboot03_10.TestController.toIndex(TestController.java:11)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:655) ~[tomcat-embed
```

图 3-39 异常信息

3.10.2 @ExceptionHandler 注解

不难发现，使用自定义 error 页面并没有真正处理异常，在本节将@ExceptionHandler 注解处理异常。该注解主要用于在 Controller 层面进行相同类型的异常处理，在对应 Controller 类中定义异常处理方法，并为其使用@ExceptionHandler 注解。Spring 会检测到该注解，并将该方法注册为对应异常类及其子类的异常处理程序。异常处理的示例代码如下。

```
@ExceptionHandler()
public String handleException2(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常：默认";
    return resultStr;
}
```

使用该注解的方法可以拥有非常灵活的签名，包括以下类型：

- 异常类型(Throwable)：可以选择一个大概的异常类型。例如，示例里的签名可以改为“Throwable e”或“Exception e”，也可以改为一个具体的异常类型。
- 请求与响应对象(Request/Response)：比如 ServletRequest 和 HttpServletRequest。
- InputStream/Reader：用于访问请求的内容。
- OutputStream/Writer：用户访问响应的内容。

• Model：作为从该方法返回 Model 的替代方案。

在@ExceptionHandler 注解中可以添加参数，参数是某个异常类的 class，代表这个方法专门处理该类异常，示例代码如下：

```
@ExceptionHandler(NumberFormatException.class)
public String handleException(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常: NumberFormatException";
    return resultStr;
}
```

此时注解的参数是 NumberFormatException.class，表示只有方法抛出 NumberFormatException 时，才会调用该方法。

当异常发生时，Spring 会选择最接近抛出异常的处理方法。

例如，NumberFormatException 异常，该异常有父类 RuntimeException 和 Exception，如果分别定义异常处理方法，@ExceptionHandler 将分别使用这 3 个异常作为参数，示例代码如下：

```
@ExceptionHandler(NumberFormatException.class)
public String handleException(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常: NumberFormatException";
    return resultStr;
}

@ExceptionHandler()
public String handleException2(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常: 默认";
    return resultStr;
}

@ExceptionHandler(RuntimeException.class)
public String handleException3(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常: RuntimeException";
    return resultStr;
}
```

当代码抛出 NumberFormatException 时，调用的方法将是注解参数 NumberFormatException.class 的方法，即 handleException()；而当代码抛出 IndexOutOfBoundsException 时，调用的方法将是注解参数 RuntimeException 的方法，即 handleException3()。

标识了@ExceptionHandler 注解的方法，其返回值类型与标识了@RequestMapping 注解的方法是相同的，可参见@RequestMapping 的说明。例如，默认返回 Spring 的 ModelAndView 对象，也可以返回 String，这时的 String 是 ModelAndView 的路径，而不是字符串本身。

有些情况下会给标识了@RequestMapping的方法添加@ResponseBody，比如使用AJAX的场景会直接返回字符串。异常处理类也可以如此操作，在添加@ResponseBody注解后，可以直接返回字符串，示例代码如下：

```
@ExceptionHandler(NumberFormatException.class)
@ResponseBody
public String handleException(Exception ex) {
    System.out.println("抛出异常:" + ex);
    ex.printStackTrace();
    String resultStr = "异常: NumberFormatException";
    return resultStr;
}
```

一个Spring Boot应用中往往存在多个控制器，不适合在每个控制器中添加使用@ExceptionHandler注解修饰的方法进行异常处理。传统的做法是定义一个控制器父类（如BaseController），它包含了执行共同操作的方法，其他的控制器类（如ControllerA和ControllerB）继承这个控制器父类。图3-40显示了控制器父类和控制器子类的关系。

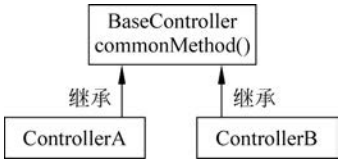


图3-40 控制器父类和控制器子类的关系

3.10.3 @ControllerAdvice注解

继承是提高控制器类的代码可重用性的有效手段，但是它有一个缺陷，那就是由于Java语言不支持多继承，当控制器类继承了一个控制器父类后，就不能再继承其他的类。

Spring MVC框架提供了另一种方式来为多个控制器类提供共同的方法：利用@ControllerAdvice注解定义一个控制器增强类。

控制器增强类并不是控制器类的父类。在程序运行时，Spring MVC框架会把控制器增强类的方法代码块动态注入其他控制器类中，通过这种方式增强控制器类的功能。图3-41显示了控制器增强类（如MyControllerAdvice）和控制器类的关系。

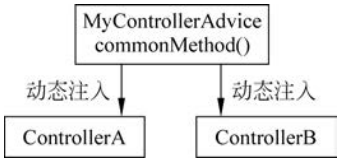


图3-41 控制器增强类和控制器类的关系

@ControllerAdvice注解是Spring 3.2中新增的注解，学名是Controller增强器，作用是给Controller控制器添加统一的操作或处理。

对于@ControllerAdvice，比较熟知的用法是结合@ExceptionHandler用于全局异常的处理，但其作用不止于此。ControllerAdvice拆开来就是Controller Advice，Advice在Spring的AOP中是用来封装一个切面所有属性的，包括切入点和需要织入的切面逻辑。这里ControllerAdvice也可以这么理解，其抽象级别应该是用于对Controller进行切面环绕的，而具体的业务织入方式则是通过结合其他的注解来实现的。@ControllerAdvice是在类上声明的注解，其用法主要有3点，灵活使用这3个功能可以简化很多工作。需要注意的是，这是Spring MVC提供的功能，在Spring Boot中可以直接使用。下面介绍@ControllerAdvice的具体用法。

1. 全局异常处理

使用@ControllerAdvice实现全局异常处理，只需要定义类并添加该注解即可。定义类的方式如下：

```
@ControllerAdvice
public class MyGlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ModelAndView customException(Exception e) {
        ModelAndView mv = new ModelAndView();
        mv.addObject("message", e.getMessage());
        mv.setViewName("myerror");
        return mv;
    }
}
```

在该类中,可以定义多个方法,不同的方法处理不同的异常,如专门处理空指针的方法、专门处理数组越界的方法等,也可以直接像上面代码一样,在一个方法中处理所有的异常信息。

@ExceptionHandler 注解用来指明异常的处理类型,即如果这里指定为NullpointerException,则数组越界异常就不会进入该方法。

2. 全局数据绑定

全局数据绑定功能可以用来做一些初始化的数据操作,可以将一些公共的数据定义在添加了@ControllerAdvice 注解的类中。这样,在每一个 Controller 的接口中,就都能够访问这些数据。

定义全局数据如下:

```
@ControllerAdvice
public class MyControllerAdvice {
  @ModelAttribute(name = "colors")
  public Map<String,String> setColors() {
    HashMap<String, String> colors = new HashMap<String,String>();
    colors.put("RED", "红色");
    colors.put("BLUE", "蓝色");
    colors.put("GREEN", "绿色");
    return colors;
  }}
```

当程序运行时,Spring MVC 框架会把 MyControllerAdvice 类的 setColors()方法动态注入其他控制器类中,因此其他控制器类自动拥有了该方法。例如,在 TestAttributeController 类中可以直接访问 Model 中的 colors 属性。

```
@RequestMapping(value = "/testColor")
public String testColor(@ModelAttribute("colors") Map<String,String> colors, @ModelAttribute
("userName") String name){
  System.out.println(name + "'s favourite color:" + colors.get("RED"));
  return "result";
}
```

通过浏览器访问 http://localhost:8080/helloapp/testColor?name=Tom,testColor()方法会在服务器端打印“TOM's favourite color:红色”。

对控制器添加通知有以下几种方式:

(1) 只对一部分控制器添加通知,如某个包下的控制器。

```
@ControllerAdvice(basePackages = {"com.example"})
//@ControllerAdvice(com.example)
```

```
public class MyControllerAdvice2{ … }
```

basePackages：指定一个或多个包，这些包及其子包下的所有 Controller 都被该@ControllerAdvice 管理。

（2）不固定包名，只想把包里的某个类传进去。

```
@ControllerAdvice(basePackageClasses = MainController.class)
public class MyControllerAdvice2{ … }
```

basePackageClasses：basePackages 的一种变形，指定一个或多个 Controller 类，这些类所属的包及其子包下的所有 Controller 都被该 @ControllerAdvice 管理。

（3）只对某几个控制器添加通知。

```
@ControllerAdvice(assignableTypes = {PersonController.class,MainController.class})
public class MyControllerAdvice2{ … }
```

3. 全局数据预处理

考虑有两个实体类为 Car 和 Driver，分别定义如下：

```
public class Car {
    private String name;
    private String type;
    //省略 get 和 set 方法
}
public class Driver {
    private String name;
    private Integer age;
    //省略 get 和 set 方法
}
```

定义一个数据添加接口如下：

```
@PostMapping("/car")
public void addCar(Car car , Driver driver ) {
    System.out.println(car);
    System.out.println(driver);
}
```

此时，添加操作会产生问题，因为两个实体类都有一个 name 属性，从前端传递时无法区分。通过@ControllerAdvice 的全局数据预处理可以解决该问题，解决步骤如下：

（1）给接口中的变量取别名。

```
@RestController
public class TestController {
  @GetMapping("/index")
  public void index(@ModelAttribute("c") Car car, @ModelAttribute("d") Driver driver) {
    System.out.println(car);
    System.out.println(driver);
    }
}
```

（2）进行请求数据预处理。

在@ControllerAdvice 标记的类中添加如下代码：

```
@ControllerAdvice(value = "com.example.controller")
public class TestControllerAdvice {
```

```
    @InitBinder("c")
    public void b(WebDataBinder binder) {
        binder.setFieldDefaultPrefix("c.");
    }
    @InitBinder("d")
    public void a(WebDataBinder binder) {
        binder.setFieldDefaultPrefix("d.");
    }
}
```

@InitBinder("c") 注解表示该方法用来处理与 Car 相关的参数,在方法中给参数添加一个 c 前缀,即请求参数要有 c 前缀。

(3) 发送请求。

请求发送时,通过给不同对象的参数添加不同的前缀,可以实现参数的区分。

请求地址 http://127.0.0.1:8080/index?c.name=保时捷&c.type=赛跑&d.name=全爷&d.age=35
输出{"Driver.age":35,"Car.type":"赛跑","Driver.name":"全爷","Car.name":"保时捷"}

本章小结

本章首先介绍了 Spring Boot 的 Web 开发支持,然后详细讲述了 Spring Boot 推荐使用的 Thymeleaf 模板引擎,包括 Thymeleaf 的基础语法、常用属性和国际化。同时,本章还介绍了 Spring Boot 对 JSON 数据的处理、文件上传下载、异常统一处理和对 JSP 的支持等 Web 应用开发的常用功能。


在线测试

习题

一、单选题

1. 以下关于 Thymeleaf 模板引擎常用标准表达式的说法,错误的是(　　)。
   A. 变量表达式#{…}主要用于获取上下文中的变量值
   B. 使用 th:text="${#locale.country}"动态获取当前用户所在国家信息
   C. 使用消息表达式#{…}进行国际化设置时,还需要提供一些国际化配置文件
   D. 片段表达式~{…}用来标记一个片段模板,并根据需要移动或传递给其他模板
2. 以下关于 Spring Boot 整合 Thymeleaf 的相关配置说法,正确的是(　　)。
   A. spring.thymeleaf.cache 表示是否开启 Thymeleaf 模板缓存,默认为 false
   B. spring.thymeleaf.prefix 指定了 Thymeleaf 模板页面的存放路径,默认为 resources/
   C. spring.thymeleaf.suffix 指定了 Thymeleaf 模板页面的名称后缀,默认为.html
   D. spring.thymeleaf.encoding 表示模板页面变化格式,默认为 iso8859-1
3. 以下关于 Thymeleaf 模板引擎页面标签的说法,错误的是(　　)。
   A. th:each 用于元素遍历,类似 JSP 中的 c:forEach 标签
   B. th:value 用于属性值修改,指定标签属性值
   C. th:utext 用于指定标签显示的文本内容,对特殊标签进行转义
   D. th:href 用于设定链接地址

4. IE不同版本User-Agent中出现的关键词不同，以下不属于IE User-Agent关键词的是(　　)。

A. MSIE　　B. Mozilla　　C. Edge　　D. Trident

5. Thymeleaf支持处理多种模板视图，不包括(　　)。

A. CSS　　B. XML　　C. JS　　D. EXE

6. 在Spring Boot中使用路径扫描的方式整合内嵌式Servlet三大组件时，不包括的注解或属性是(　　)。

A. @WebServlet注解　　B. @EnableWebMvc注解

C. @ServletComponentScan注解　　D. value属性

二、多选题

1. 以下关于Spring Boot整合Spring MVC框架实现Web开发中文件上传功能的相关说法，错误的是(　　)。

A. 实现文件上传功能，还需要提供文件上传相关依赖

B. 必须在配置文件中对文件上传功能进行配置

C. 多文件上传处理类中的方法中必须由MultipartFile[]类型参数进行多文件接收

D. spring.servlet.multipart.max-file-size用来设置所有上传文件的大小限制，默认值为10MB

2. 以下关于Thymeleaf主要标准表达式语法及说明，正确的是(　　)。

A. Thymeleaf模板页面中的th:text="${#locale.country}"动态获取当前用户所在国家信息

B. ${#object.firstName}使用Thymeleaf模板提供的内置对象object获取当前上下文对象中的firstName属性值

C. <div th:insert="~{thymeleafDemo::title}"></div>中的title为引入的模板名称

D. 使用th:insert或th:replace属性可以插入Thymeleaf模板片段

3. Spring Boot框架整合MVC实现Web开发支持的前端模板引擎包括(　　)。

A. Mustache　　B. FreeMarker　　C. Thymeleaf　　D. Groovy

三、判断题(对的打"√"，错的打"×")

1. 在Spring Boot项目的classpath:/static/目录下编写一个index.html页面，可以作为Spring Boot默认欢迎页。(　　)

2. 在Spring Boot中进行中文名文件下载处理时，如果内核信息是IE，则转码为ISO-8859-1进行处理。(　　)

3. Thymeleaf支持处理6种模板视图，包括HTML、XML、TEXT、JAVASCRIPT、CSS和RAW。(　　)

4. 使用data-th-*属性定制Thymeleaf模板页面时，不需要引入Thymeleaf标签。(　　)

5. Thymeleaf是适用于Web和独立环境的现代服务器端Java模板引擎。(　　)

四、填空题

1. Spring Boot为整合Spring MVC框架实现Web开发，支持静态项目首页_____。

2. Spring Boot 整合 Spring MVC 实现 Web 开发,需要引入依赖启动器_____。

3. 在 Spring Boot 中,使用组件注册方式整合内嵌 Servlet 容器的 Filter 组件时,只需将自定义组件通过_____类注册到容器中即可。

4. Spring Boot 为整合 Spring MVC 框架实现 Web 开发,内置了 ContentNegotiating-ViewResolver 和_____两个视图解析器。

5. 消息表达式_____主要用于 Thymeleaf 模板页面国际化内容的动态替换和展示。