第3章 Spring MVC 的组件开发

CHAPTER 3

本章目标

1. 了解 Spring MVC 框架中的拦截器，掌握拦截器的创建和配置方法。
2. 掌握文件的上传和下载的基本原理和使用方法。
3. 了解 Spring MVC 标签库，掌握几个常用标签的使用方法。

3.1 拦 截 器

Spring MVC 中的拦截器(interceptor)类似于 Servlet 中的过滤器(filter)，主要用于拦截用户请求并作相应的处理。例如，通过拦截器可以进行权限验证、记录请求信息的日志、判断用户是否登录等。使用 Spring MVC 中的拦截器，须对拦截器类进行定义和配置。

3.1.1 Spring MVC 拦截器的设计

在开发拦截器之前，要先了解拦截器的设计结构。拦截器必须实现 HandlerInterceptor 接口。为增强功能，开发了多个拦截器。Spring MVC 拦截器接口与类设计结构如图 3-1 所示。

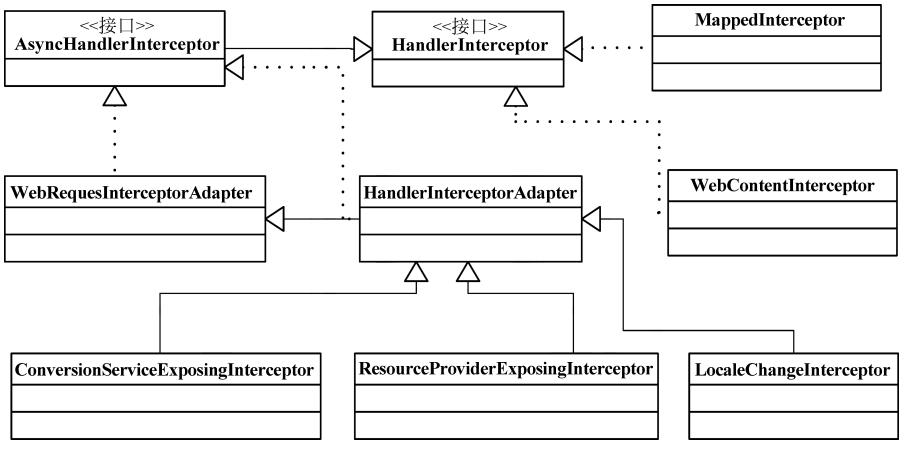

图 3-1 Spring MVC 拦截器接口与类设计结构

从图 3-1 中可以看出，所有的类都直接或间接实现了 HandlerInterceptor 接口，HandlerIntercerptor 的源代码如下。

```
1.  package org.springframework.web.servlet;
2.  public interface HandlerInterceptor {
3.      default boolean preHandle(HttpServletRequest request, HttpServletResponse
        response, Object handler)
4.          throws Exception {
5.      return true;
6.    }
7.    default void postHandle(HttpServletRequest request, HttpServletResponse
      response, Object handler,
8.        @Nullable ModelAndView modelAndView) throws Exception {
9.    }
10.   default void afterCompletion(HttpServletRequest request, HttpServlet-
      Response response, Object handler,
11.           @Nullable Exception ex) throws Exception {
12.   }
13. }
```

HandlerInterceptor 接口中的 3 个方法描述如下。

- preHandle()方法：该方法在控制器方法被调用前执行，其返回值为一个 boolean 值，如果为 true，表示继续向下执行；如果为 false，会中断后面的所有操作。
- postHandle()方法：该方法在控制器方法被调用后执行，且在解析视图之前执行，可以通过此方法对请求域中的模型和视图做进一步修改。
- afterCompletion()方法：该方法在视图渲染结束之后执行，即该方法在整个请求完成后，无论是否产生异常都会执行。

拦截器的执行流程可用如图 3-2 所示的流程图表示。

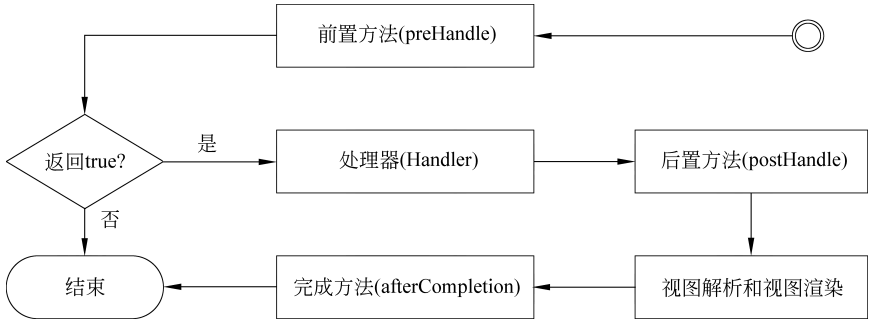

图 3-2　拦截器的执行流程图

3.1.2　单个拦截器的使用

要使用 Spring MVC 中的拦截器，需要对拦截器类进行定义和配置。通常，拦截器类

可以通过两种方式定义。

- 通过实现 HandlerInterceptor 接口。
- 继承 HandlerInterceptor 接口的实现类(如 HandlerInterceptorAdapter)定义。

以继承 HandlerInterceptorAdapter 类为例,自定义拦截器,拦截对象是所有以 *.do 结尾的请求资源,类如代码清单 3-1 所示。

代码清单 3-1：/chap3/src/com/interceptor/UserInterceptor.java

```
1.  package com.interceptor;
2.  import javax.servlet.http.HttpServletRequest;
3.  import javax.servlet.http.HttpServletResponse;
4.  import org.springframework.lang.Nullable;
5.  import org.springframework.web.servlet.ModelAndView;
6.  import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
7.  public class UserInterceptor extends HandlerInterceptorAdapter {
8.      public boolean preHandle(HttpServletRequest request, HttpServletResponse
        response, Object handler)
9.            throws Exception {
10.       System.out.println("preHandle of UserInterceptor ");
11.       return true;
12.   }
13.   public void postHandle(HttpServletRequest request, HttpServletResponse
      response, Object handler,
14.           @Nullable ModelAndView modelAndView) throws Exception {
15.       System.out.println("postHandle of UserInterceptor ");
16.   }
17.   public void afterCompletion ( HttpServletRequest request,
      HttpServletResponse response, Object handler,
18.       @Nullable Exception ex) throws Exception {
19.     System.out.println("afterCompletion of UserInterceptor ");
20.   }
21. }
```

在这个自定义的拦截器中,每个方法中只是在控制台上输出一个字符串,表示当前方法被调用。

下面是控制器的一个处理请求的方法,如代码清单 3-2 所示。

代码清单 3-2：com/controller/HelloController.java

```
1.  @Controller
2.  public class HelloController {
3.      @RequestMapping("/hello")
4.      public ModelAndView handleRequest ( HttpServletRequest request,
        HttpServletResponse response) throws Exception {
5.      //创建 ModelAndView 对象
```

```
6.        System.out.println("第一个 Hello World 程序");
7.        ModelAndView mv=new ModelAndView();
8.        //将信息保存在 ModelAndView 对象,这个对象中的值可传递到 JSP 页面中
9.        mv.addObject("msg","第一个 Hello World 程序");
10.       //设置要转发的页面
11.       mv.setViewName("helloWorld");
12.       return mv;
13.     }
14.   }
```

在这个请求处理方法中，在控制台上打印了一个字符串，表示当前方法被调用，然后转发到 helloWorld.jsp 页面。

最后，在 Spring MVC 的配置文件中添加拦截器的配置信息，如代码清单 3-3 所示。

代码清单 3-3：/chap3/WebContent/WEB-INF/springmvc-config.xml

```
1.   <?xml version="1.0" encoding="UTF-8"?>
2.   <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:mvc="http://www.springframework.org/schema/mvc"
4.       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.       xmlns:p="http://www.springframework.org/schema/p"
6.       xmlns:context="http://www.springframework.org/schema/context"
7.       xsi:schemaLocation="
8.           http://www.springframework.org/schema/beans
9.           https://www.springframework.org/schema/beans/spring-beans.xsd
10.          http://www.springframework.org/schema/context
11.          https://www.springframework.org/schema/context/spring-context.xsd
12.          http://www.springframework.org/schema/mvc
13.          https://www.springframework.org/schema/mvc/spring-mvc.xsd">
14.      <!--指定需要扫描的包 -->
15.      <context:component-scan base-package="com.controller" />
16.      <!--定义视图解析器 -->
17.      <bean id="viewResolver" class=
18.      "org.springframework.web.servlet.view.InternalResourceViewResolver">
19.          <!--设置前缀 -->
20.          <property name="prefix" value="/WEB-INF/jsp/" />
21.          <!--设置后缀 -->
22.          <property name="suffix" value=".jsp" />
23.    </bean>
24.    <mvc:interceptors>
25.    <mvc:interceptor>
```

```
26.     <mvc:mapping path="/*.do"/>
27.     <bean class="com.interceptor.UserInterceptor"/>
28.     </mvc:interceptor>
29.     </mvc:interceptors>
30.  </beans>
```

第 3、12、13 行黑体字部分是定义 mvc 命名空间和所需的 schema 文件位置信息。第 24 行的 <mvc:interceptors>是一个容器标签，其子标签是<mvc:interceptor>，用这个标签可定义多个拦截器。

在<mvc:interceptor>标签下有两个子标签，第 26 行的<mvc:mapping path="/*.do">，表示拦截后缀为 *.do 的所有请求 uri。第 27 行定义实现这个拦截器的类。

首先，发布此服务，然后在浏览器中访问此服务，其运行结果如图 3-3 所示。

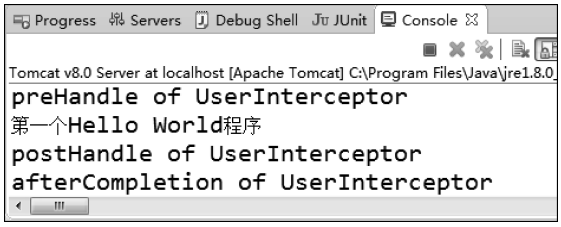

图 3-3　单个拦截器运行结果

从运行结果看，拦截器中的 3 个方法调用次序与前面分析的结果一致。

3.1.3　多个拦截器的使用

当一个项目比较大，业务逻辑复杂，需求也在不断变化的时候，可以添加多个拦截器满足实际工程的需求，此时多个拦截器中的方法执行次序也发生了变化。下面通过有多个拦截器存在时的示例说明拦截器中的方法与处理器的方法之间的先后执行次序。第一个拦截器的定义如代码清单 3-4 所示。

代码清单 3-4：/chap3/src/com/interceptor/Interceptor1.java

```
1.  public class Interceptor1 extends HandlerInterceptorAdapter {
2.    public boolean preHandle (HttpServletRequest request, HttpServletResponse
      response, Object handler)
3.          throws Exception {
4.      System.out.println("preHandle of UserInterceptor1 ");
5.      return true;
6.    }
7.    public void postHandle (HttpServletRequest request, HttpServletResponse
      response, Object handler,
8.        @Nullable ModelAndView modelAndView) throws Exception {
9.      System.out.println("postHandle of UserInterceptor1 ");
10.   }
```

```
11.    public void afterCompletion(HttpServletRequest request,
       HttpServletResponse response, Object handler,
12.        @Nullable Exception ex) throws Exception {
13.      System.out.println("afterCompletion of UserInterceptor1 ");
14.    }
15.  }
```

第二个拦截器的定义如代码清单 3-5 所示。

代码清单 3-5：/chap3/src/com/interceptor/Interceptor2.java

```
1.   public class Interceptor2 extends HandlerInterceptorAdapter {
2.     public boolean preHandle(HttpServletRequest request, HttpServletResponse
       response, Object handler)
3.          throws Exception {
4.       System.out.println("preHandle of UserInterceptor2 ");
5.       return true;
6.       }
7.       public void postHandle(HttpServletRequest request, HttpServletResponse
         response, Object handler,
8.           @Nullable ModelAndView modelAndView) throws Exception {
9.       System.out.println("postHandle of UserInterceptor2 ");
10.    }
11.    public void afterCompletion(HttpServletRequest request,
       HttpServletResponse response, Object handler,
12.         @Nullable Exception ex) throws Exception {
13.      System.out.println("afterCompletion of UserInterceptor2 ");
14.    }
15.  }
```

在 Spring MVC 配置文件中添加拦截器配置，如代码清单 3-6 所示。

代码清单 3-6：/chap3/WebContent/WEB-INF/springmvc-config.xml

```
1.   <mvc:interceptors>
2.        <mvc:interceptor>
3.        <mvc:mapping path="/*.do"/>
4.        <bean class="com.interceptor.UserInterceptor1"/>
5.        </mvc:interceptor>
6.        <mvc:interceptor>
7.        <mvc:mapping path="/*.do"/>
8.        <bean class="com.interceptor.UserInterceptor2"/>
9.        </mvc:interceptor>
10.  </mvc:interceptors>
```

控制器处理请求的方法还是用代码清单 3-2 所示代码，启动服务后，在浏览器中访问

处理请求的映射地址，运行结果如图 3-4 所示。

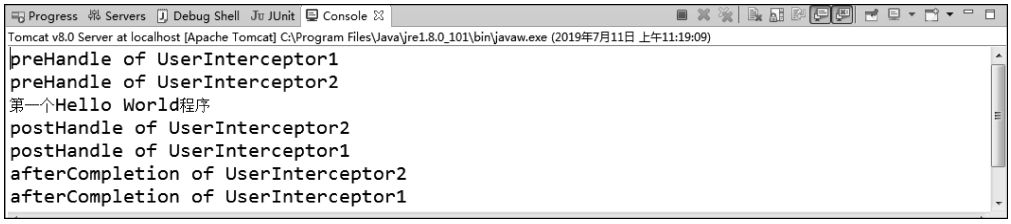

图 3-4　多个拦截器运行结果

从运行结果分析，在执行请求方法 handleRequest()之前，依次执行 Interceptor1 和 Interceptor2 中的 prdHandle()方法，在执行 handleRequest()之后，反序执行拦截器中的另外两个方法。多拦截器各方法之间执行次序如图 3-5 所示。

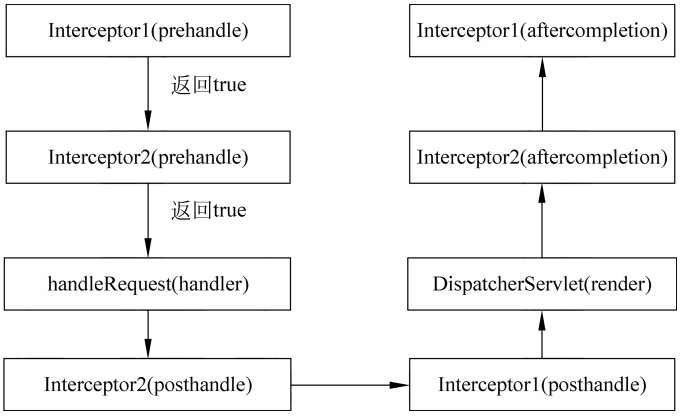

图 3-5　多拦截器各方法之间执行次序

3.1.4　拦截器应用——用户权限验证

拦截器的作用是在请求处理方法执行前，执行拦截器中的 prehandle()方法，这样可以预先处理一些事务，如判断一个用户是否已经登录一个网站，如果已经登录，可直接进入网站；如果没有登录，则转发到登录页面。下面是一个通过拦截器完成用户权限验证的实例。

例 3-1　当用户访问系统所有资源前进行权限的验证，如果是已登录的用户，可直接进入相应的请求处理方法；如果是未登录的用户，则重定向到登录页面。

（1）编写控制器类，如代码清单 3-7 所示。

代码清单 3-7：/chap3/src/com/controller/UserController1.java

```
1.  package com.controller;
2.  //处理用户请求的控制器
3.  @Controller
4.  //响应所有 userInterceptor 路径下的处理请求
```

```
@RequestMapping(value="/userInterceptor")
public class UserController1 {
    @GetMapping(value="/login")                    //响应 login.do 请求
    public String login() {
        return "/login";                           //转发到 login.jsp 页面
    }
 //这是处理用户登录的方法,响应 userInterceptor/login.do 的处理请求
    @PostMapping(value="/login")
    public String login(User user,Model model,HttpSession session ) {
        String msg;                                //登录是否成功信息
        if(user.getUsername()!=null&&user.getUsername().equals("admin")
            &&user.getPassword()!=null&&user.getPassword().equals(
            "123456"))
        {
            msg="登录成功";
            model.addAttribute("user",user);       //保存数据到 Model 对象中
            session.setAttribute("user",user);     //保存数据到 session 对象中
            return "main";                         //转发到 main.jsp
        }
        else
            msg="登录失败";
        model.addAttribute("msg",msg);             //保存数据到 Model 对象中
        return "login";                            //转发到 login.jsp
    }
    @RequestMapping(value="/main")                 //响应 main.do 请求
    public String main() {
        return "main";                             //转发到 main.jsp 页面
    }
    @GetMapping(value="/logout")                   //响应 logout.do 请求
    public String logout(HttpSession session) {
        session.invalidate();                      //当前的 session 失效
 //重定向到 Login.do 请求
        return "redirect:/userInterceptor/login.do";
    }
}
```

（2）编写拦截器类，决定哪些请求是允许的，哪些请求是非法的并加以拒绝，如代码清单 3-8 所示。

代码清单 3-8：/chap3/src/com/interceptor/LoginInterceptor.java

```
public class LoginInterceptor extends HandlerInterceptorAdapter {
  //进入 Handler()方法之前执行
  //可用于身份认证、身份授权。如果认证没有通过,表示用户没有登录,需要此方法拦
  //截不再往下执行,否则就放行
```

```
4.     @Override
5.     public boolean preHandle(HttpServletRequest request, HttpServletResponse
       response, Object handler)
6.           throws Exception {
7.       //获取请求的 url
8.       String url=request.getRequestURI();
9.       //判断 url 是否公开地址(实际使用时将公开地址配置到配置文件中)
10.      //这里假设公开地址是登录提交的地址
11.      if (url.indexOf("login.do") >0) {
12.          //如果进行登录提交,放行
13.          return true;
14.      }
15.      //判断 session
16.      HttpSession session=request.getSession();
17.      //从 session 中取出用户身份信息
18.      User user=(User) session.getAttribute("user");
19.      if (user !=null) { //如果用户存在,表示已经登录,放行
20.          return true;
21.      }
22.      //执行到这里表示用户身份需要验证,跳转到登录页面
23.       request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward
          (request, response);
24.      return false;
25.    }
26.  }
```

(3) 在 springmvc-config.xml 中配置自定义的拦截器,如代码清单 3-9 所示。

代码清单 3-9：/chap3/WebContent/WEB-INF/springmvc-config.xml

```
1.  <mvc:interceptors>
2.  <mvc:interceptor>
3.  <!-只拦截/userInterceptor 路径下的请求-->
4.  <mvc:mapping path="/userInterceptor/*.do"/>
5.  <!-自定义的拦截器类-->
6.  <bean class="com.interceptor.LoginInterceptor"/>
7.  </mvc:interceptor>
8.  </mvc:interceptors>
```

配置文件中其他省略的代码与代码清单 3-3 相同。

(4) 视图页面,main.jsp 作为主页,login.jsp 是登录页面,在没有登录前,只能显示 login.jsp 页面,其他页面是不允许访问的,只有登录后,才能访问其他页面,如 main.jsp。

main.jsp 页面如代码清单 3-10 所示。

代码清单 3-10：/chap3/WebContent/WEB-INF/jsp/main.jsp

```
1.  <body>
2.  当前用户：${user.username}
3.  <a href="${pageContext.request.contextPath}/userInterceptor/logout.
    do">退出</a>
4.
5.  </body>
```

login.jsp 页面如代码清单 3-11 所示。

代码清单 3-11：/chap3/WebContent/WEB-INF/jsp/login.jsp

```
1.  <form action="${pageContext.request.contextPath}/userInterceptor/
    login.do" method="post" onsubmit="return check()">
2.          <br /><br />
3.  账 号：<input id="usercode" type="text" name="username" />
4.      <br /><br />
5.  密 码：<input id="password" type="password" name="password" />
6.    <br /><br />
7.  <center><input type="submit" value="登录" /></center>
8.  </form>
```

代码清单 3-11 中只给出了 form 部分的代码，其他代码与前面的相同。

（5）发布服务，启动 Tomcat，首先访问 main.do，出现如图 3-6 所示的页面。

图 3-6　login.jsp 页面

由于用户没有登录，此时访问 main.do 时被拦截器拦截，跳转到 login.jsp 页面，此时输入正确的用户名和密码，单击“登录”按钮后进入 main.jsp 页面，如图 3-7 所示。

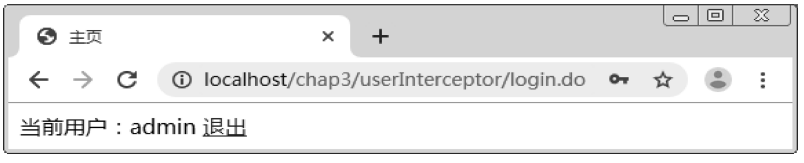

图 3-7　main.jsp 页面

登录成功后，可以随时进入主页面 main.jsp，但当单击“退出”链接后，当前的 session

失效，将不能再直接访问 main.jsp，而是跳转到 index.jsp 页面。

3.2　文件的上传与下载

在实际工程中，文件的上传与下载是必备的功能之一，Spring MVC 对此给出了很好的解决方案，可以方便地实现这个功能。

3.2.1　文件的上传

Spring MVC 为上传文件提供了良好的支持。首先，Spring MVC 的文件上传是通过 MultipartResolver(Multipart 处理器)处理的，对于 MultipartResolver 来说，它只是一个接口，有两个实现类。

- CommonsMultipartResolver 类，依赖于 Apache 下的 jakarta Common FileUpload 项目解析 Multipart 请求，它可以在 Spring 的各个版本中使用，只是它依赖于第三方 JAR 包才能实现。
- StandardServletMultipartResolver 类，是 Spring 3.1 版本后的产物，它依赖于 Servlet 3.0 或者更高版本的实现，但不依赖于第三方 JAR 包。

两个实现类与 MultipartResolver 接口的关系如图 3-8 所示。

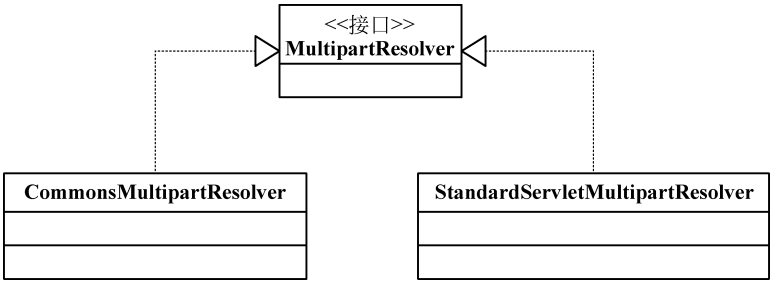

图 3-8　两个实现类与 MultipartResolver 接口的关系

下面以第二种方案实现为例，讲解 Spring MVC 如何实现文件的上传功能。

要实现文件上传，需要前端与后端服务器配合才能实现。首先，在前端页面上要有一个上传文件的表单，表单需要满足下面 3 个条件。

- form 表单的 method 属性设置为 post。
- form 表单的 enctype 属性设置为 multipart/form-data。
- 设置<input>标签的 type="file"。

示例代码如下：

```
1.  <form action="upload.do" enctype="multipart/form-data"
2.        method="post" >
3.  <input type="file" name="file" />
4.  <input type="submit" value="上传文件" multiple="multiple"/>
```

上述代码中，上传文件除了满足 3 个必备条件外，还增加了 multiple 属性，该属性是 HTML 5.0 中的新特性，如果使用了该属性，同时选中多个文件进行上传，即可实现多文件上传。

当客户端 form 表单的 enctype 属性为 multipart/form-data 时，浏览器会采用二进制流的形式处理表单的数据，服务器端就会对文件上传的请求进行解析处理。Spring MVC 为文件上传提供了直接的支持，这是通过 MultipartResolver 接口实现的。

MultipartResolver 接口有两个实现类，前面已经介绍过，这里用 Spring MVC 自己提供的 StandardServletMultipartResolver 类完成文件的上传。由于这个类是基于 Servlet 的，所以要在 Web.xml 中进行配置，示例代码如下。

```
<multipart-config>
            <!--临时文件的目录-->
            <location>e:/mvc/upload</location>
            <!--上传文件最大 2MB -->
            <max-file-size>2097152</max-file-size>
            <!--上传文件整个请求不超过 4MB -->
            <max-request-size>4194304</max-request-size>
</multipart-config>
```

上面的代码中，<multipart-config>作为 DispatcherServlet 的子标签，在此是设置上传文件的相关属性，如默认目录，上传文件大小限制等。接下来还要在 Spring MVC 配置文件中进行配置，如下面的代码所示。

```
<!--配置文件上传解析器 -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.support.StandardServletMult
ipartResolver">
```

因为 MultipartResolver 接口的实现类 StandardServletMultipartResolver，内部是引用 MultipartResolver 字符串获取该实现对象并完成文件解析的，所以在配置这个 Bean 的实例时，必须指定 Bean 的 id 为 multipartResolver。

当完成页面表单和文件上传解析器的配置后，需要在 Controller 中编写文件上传的方法实现文件上传的功能。在 Spring MVC，实现文件上传功能其实很简单，其代码如下。

```
//处理上传文件请求的方法及方法的形参
@RequestMapping(value ="/upload")
    public ModelAndView upload(
@RequestParam("username") String username,
HttpServletRequest request,
MultipartFile file)
```

该方法是用来处理上传文件请求的，其中的 file 是一个 MultipartFile 引用变量，用来接收前端页面传递过来的文件，并将文件封装在这个对象中。MultipartFile 接口中提供

了一些获取文件信息的方法，见表 3-1。

表 3-1　MultipartFile 接口中提供的获取文件信息的方法

方　法	说　明
Byte[] getBytes()	以字节数组的形式返回文件的内容
String getContentType()	返回文件的内容
InputStream getInputStream()	读取文件内容，返回一个 InputStream 流
String getName()	获取多部件 form 表单的参数名称
String getOriginalFilename()	获取上传文件的原始文件名
Boolean isEmpty()	判断上传文件的内容是否为空
Void transferTo(File file)	保存上传文件到目标目录中

例 3-2　文件上传示例，要求在一个 JSP 页面单击“浏览”按钮，打开一个文件管理窗口，选中某一个或多个文件后，可以上传到服务器指定的目录下。

具体实现步骤如下。

(1) 创建 Web 工程，添加 Spring MVC 的 JAR 包，修改 web.xml 配置文件，添加 SpringMVC 必需的 listener 和 servlet。

(2) 创建上传文件页面 fileUpload.jsp，如代码清单 3-12 所示。

代码清单 3-12：/chap3/WebContent/WEB-INF/jsp/fileUpload.jsp

```
1.   <form action="${pageContext.request.contextPath}/file/upload.do"
2.       enctype="multipart/form-data"
3.       method="post" onsubmit="return check()">
4.                <br /><br />
5.       上传人: <input id="username" type="text" name="username" />
6.          <br /><br />
7.       请选择文件: <input id="filename" type="file" name="file" />
8.          <br /><br />
9.          <center><input type="submit" value="上传" /></center>
10.  </form>
```

代码第 1 行中的 action 请求的地址是 file/upload.do，对应的是 FileUploadController 控制器类中的 upload()方法，在此处理上传文件的请求。

(3) 创建控制器类 FileUploadController，如代码清单 3-13 所示。

代码清单 3-13：/chap3/src/com/controller/FileUploadController.java

```
1.   @Controller
2.   @RequestMapping("/file")
3.   public class FileUploadController {
4.       @RequestMapping(value ="/toUpload")
5.       public String toUpload() {
6.           return "fileUpload";
```

```
7.        }
8.        //处理上传文件请求的方法
9.      @RequestMapping(value="/upload")
10.     public ModelAndView upload(@RequestParam("username") String username,
              HttpServletRequest request,MultipartFile file) {
11.           ModelAndView mv=new ModelAndView();
12.           //指定要上传的文件所在路径
13.           String path =request.getServletContext().getRealPath("/upload/");
14.           //获取原始文件名
15.           String fileName=file.getOriginalFilename();
16.           fileName=path+File.separator+fileName;
17.           file.getContentType();
18.           //创建目标文件
19.           File dest=new File(fileName);
20.           String msg=null;
21.           try {
22.                 //保存文件
23.                 file.transferTo(dest);
24.                 msg="上传文件成功";
25.           }catch(IllegalStateException|IOException e) {
26.                 msg="上传文件失败";
27.                 e.printStackTrace();
28.         }
29.         mv.addObject("msg", msg);       //提示信息保存在 ModelAndView 对象中
30.   //用户信息保存在 ModelAndView 对象中
31.         mv.addObject("username",username);
32.         mv.setViewName("result");
33.         return mv;
34.     }
35.   }
```

第 10 行的 MultipartFile file 是 MultipartFile 接口的封装对象，其实现类是在 Spring MVC 中配置的 StandardServletMultipartResolver 类，此处的参数对应表单中的上传文件控件提交的文件信息。

第 13 行指定上传文件保存在服务中的路径，如果不指定，将保存在 web.xml 中指定的默认路径。

第 15 行得到上传文件原来的名字，在此也可以根据需要重新对文件进行命名。

第 23 行将上传文件保存到指定的服务器路径下。

（4）创建 springmvc-config.xml 配置文件，如代码清单 3-14 所示。

代码清单 3-14：/chap3/WebContent/WEB-INF/springmvc-config.xml

```
1.  <beans><!-schema 配置信息省略 -->
2.      <!--指定需要扫描的包 -->
3.      <context:component-scan base-package="com.controller" />
4.      <!--定义视图解析器 -->
5.      <bean id="viewResolver" class=
6.     "org.springframework.web.servlet.view.InternalResourceViewResolver">
7.          <!--设置前缀 -->
8.          <property name="prefix" value="/WEB-INF/jsp/" />
9.          <!--设置后缀 -->
10.         <property name="suffix" value=".jsp" />
11.     </bean>
12.     <!--配置文件上传解析器 -->
13.     <bean id="multipartResolver"
14.         class=" org. springframework. web. multipart. support. Standard -
            ServletMultipartResolver">
15.     </bean>
16. </beans>
```

第 14 行配置的 bean 是 MultipartFile 接口的实现类，此处通过 Spring IoC 创建类的实例。

(5) 修改 web.xml 配置文件，修改后的代码如代码清单 3-15 所示。

代码清单 3-15：/chap3/WebContent/WEB-INF/web.xml

```
1.  <servlet-name>app</servlet-name>
2.      <servlet-class>org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
3.      <init-param>
4.          <param-name>contextConfigLocation</param-name>
5.          <param-value>/WEB-INF/springmvc-config.xml</param-value>
6.      </init-param>
7.      <load-on-startup>1</load-on-startup>
8.      <multipart-config>
9.          <!--临时文件的目录-->
10.         <location>e:/mvc/upload</location>
11.         <!--上传文件最大 2MB -->
12.         <max-file-size>2097152</max-file-size>
13.         <!--上传文件整个请求不超过 4MB -->
14.         <max-request-size>4194304</max-request-size>
15.       </multipart-config>
16.
17. </servlet>
```

第 8～15 行配置了 MultipartFile 接口实现类 StandardServletMultipartResolver 的几个属性，这几个属性是上传文件必需的。

（6）发布服务，启动 Tomcat，在浏览器中输入地址
http://localhost/chap3/file/toUpload.do，浏览器中出现如图 3-9 所示的页面。

图 3-9　上传文件页面

单击“选择文件”按钮，打开目录管理窗口，选择要上传的文件，并输入上传人的姓名，单击“上传”按钮后，出现上传文件成功的页面，此时文件被上传到项目工程所在目录的 upload 目录中。

如果工程的名字为 chap3，发布到 Eclipse 的 Tomcat 中，则发布后工程的实际目录为

```
F:\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps
  \chap3\
```

其中 F:\workspace 是 Eclipse 的工作空间所在目录。

3.2.2　文件的下载

文件下载看似简单，其实还是要分几种情况分别进行处理。最简单的情况是在页面给出一个超链接，该链接 href 的属性等于要下载文件的文件名，就可以实现文件下载了。这种情况只适用于非中文文件件名，而且文件不是很大。如果该文件的文件名为中文文件名，在某些早先的浏览器上就会导致下载失败；如果使用最新的 Firefox、Chrome、Opera、Safari，则都可以正常下载文件名为中文的文件。为了保证下载成功，此时要对文件名进行转码处理，避免出现中文乱码。如果要下载的文件很大，而且要随时掌握下载的状态，此时可通过服务器读取本地文件流，然后将文件流输出到客户端，在这个过程中可以轻易获取文件传输过程中的各个参数。下面给出的是利用 ResponseEntity 类型实现的一种文件下载方式。

例 3-3　创建一个下载文件的页面，单击下载链接可下载服务器端指定的文件，要求对汉字文件名也可以正常下载。

（1）创建一个下载页面，如代码清单 3-16 所示。

代码清单 3-16：/chap3/WebContent/WEB-INF/jsp/download.jsp

```
1.  <%@page language="java" contentType="text/html; charset=UTF-8"
2.    pageEncoding="UTF-8"%>
3.  <%@page import="java.net.URLEncoder"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C //DTD HTML 4.01 Transitional //EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>下载页面</title>
</head>
<body>
<a href="${pageContext.request.contextPath}/file/download.do?
filename=第 1 章 spring 的基础知识.pptx">文件下载  </a>
</body>
</html>
</html>
```

此处的下载文件名为“第 1 章 spring 的基础知识.pptx”，是存放在服务器端指定下载目录中的文件，此处是发布工程目录下的 upload 目录。

(2) 在控制器类 FileUploadController 中创建处理下载文件请求的方法 fileDownload()，如代码清单 3-17 所示。

代码清单 3-17：/chap3/src/com/controller/FileUploadController.java

```
//处理文件下载请求的方法
  @RequestMapping("/download")
    public ResponseEntity<byte[]> fileDownload(HttpServletRequest
    request,
    @RequestParam("filename") String filename,
    @RequestHeader("User-Agent") String userAgent) throws Exception{
    //指定要下载的文件所在路径
    String path=request.getServletContext().getRealPath("/upload/");
    System.out.println(path);
    //创建该文件对象
    File file=new File(path+File.separator+filename);
    //该方法返回一个 static ResponseEntity.BodyBuilder 对象
    BodyBuilder builder=ResponseEntity.ok();

    builder.contentType(MediaType.APPLICATION_OCTET_STREAM);
    //对文件名编码，防止中文文件乱码
    filename=URLEncoder.encode(filename,"UTF-8");
  //不同的浏览器，处理方式不同，要根据浏览器版本区别对待
  //如果是 IE，只用 UTF-8 字符集进行 URL 编码即可
    if(userAgent.indexOf("MSIE")>0) {
      builder.header("Content-Disposition","attachment;filename="+
      filename);}
```

```
21.     //而 FireFox、Chrome 等浏览器,则需要说明编码的字符集
22.       else
23.       {builder.header("Content-Disposition","attachment;filename*=UTF
          -8''"+filename);
24.       }
25.
26.      return builder.body(FileUtils.readFileToByteArray(file));
27.   }
```

ResponseEntity 继承了 HttpEntity，是 HttpEntity 的子类，第 12 行的 ResponseEntity.ok() 方法返回一个 BodyBuilde，可用于 RestTemplate 和 Controller 层方法。第 26 行的 builder.body() 方法返回一个 ResponseEntity 对象。

（3）发布服务，并启动 Tomcat，分别在不同的浏览器中进行测试。下面是在 Chrome 浏览器中测试结果。在地址栏中输入网址 http://localhost/chap3/file/toDownload.do，在页面上单击“文件下载”超链接，出现如图 3-10 所示的页面。

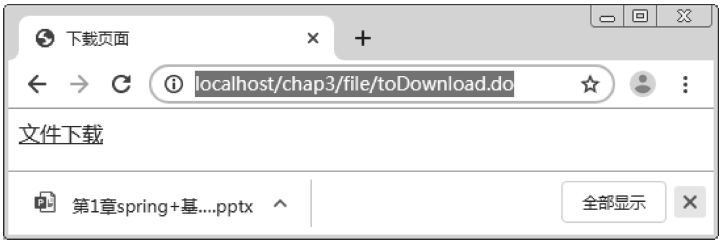

图 3-10　下载文件页面

从运行结果看，中文文件名下载是正常的，对不同的浏览器进行文件下载测试，程序都能正常运行。

3.3　Spring 的表单标签库

从 Spring 2.0 版开始，Spring 提供了一组全面的可绑定数据的标签库，用于在使用 JSP 和 Spring Web MVC 时处理表单元素，每个标签都支持其相应 HTML 标签对应的属性集。与其他表单/输入标签库不同，Spring 的表单标签库与 Spring Web MVC 集成，使标签可以访问控制器处理的命令对象和引用数据。正如我们在以下示例中所示，表单标记使 JSP 更易于开发、读取和维护。

表单标签库实现类在 spring-webmvc.jar，标签库描述符文件是 spring-form.tld。要在页面上使用此标签库中的标签，需要在 JSP 页面的头部添加如下的 JSP 指令：

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

3.3.1　form 标签

使用 Spring 的 form 标签的作用与 HTML 中的 form 功能类似，只是属性有所不同，其中一个重要的属性 modelAttribute 是 Spring 所特有的，它可以绑定模型属性名称，默认值是 command，可自定义修改。示例代码如下：

```
<form:form modelAttribute="user" method="post" action="">
…
</form:form>
```

包含这段代码的 JSP 页面要保证在 Model 中存在 user 这个属性变量名称。

3.3.2　input 标签

Spring MVC 中的 input 标签会被渲染成一个类型为 HTML input 的标签，使用 Spring 标签的好处是可以绑定表单数据，一般是通过 input 标签的 path 属性绑定 Model 中的值。示例代码如下：

```
1.   <form:form method="post" action="addProduct.do" >
2.     <table>
3.       <tr>
4.           <td>产品名称:</td>
5.           <td><form:input path="name"/></td>
6.       </tr>
7.       <tr>
8.           <td>生产日期:</td>
9.           <td><form:input path="pd"/></td>
10.       </tr>
11.     <tr>
12.         <td>价格:</td>
13.         <td><form:input path="price"/></td>
14.       </tr>
15.       <tr>
16.         <td><input type="submit" value="提交"/></td>
17.       </tr>
18.    </table>
19.  </form:form>
```

这里的 path 属性相当于 HTML 中 input 标签的 name 属性，只是这里的 path 属性可以与 Model 中的值进行绑定。除此标签外，如 password、hidden、textarea 等标签都最终渲染为 HTML 对应的标签，主要区别是有一个 path 属性绑定与 Controller 中请求处理方法的 Model 属性对应的属性值。

3.3.3 checkboxes 标签

Spring MVC 的 checkboxes 标签会渲染多个类型为 checkbox 的普通 HTML 标签。checkboxes 常用的属性主要有两个：path 和 items，具体含义如下。

- path 属性：要绑定属性路径，与 Controller 请求方法的参数对应。
- items 属性：用于生成 checkbox 元素的对象的 Collection、Map 或者 Array。

使用 checkboxes 标签时，以上两个属性是必须指定的，items 表示当前要用来显示的项的集合数据，而 path 绑定的表单对象的属性表示当前表单对象拥有的项，即在 items 显示的所有项中，表单对象拥有的项会被设定为选中状态。

例 3-4 以用户的爱好为例演示 checkboxes 标签的使用方法。

(1) 创建 TagController 控制器，生成填充复选框选项的值，如代码清单 3-18 所示。

代码清单 3-18：chap3/src/com/controller/TagController.java

```
1.  package com.controller;
2.  import java.util.ArrayList;
3.  import java.util.List;
4.  import org.springframework.stereotype.Controller;
5.  import org.springframework.ui.Model;
6.  import org.springframework.web.bind.annotation.RequestMapping;
7.  import com.po.User;
8.  @Controller
9.  public class TagController {
10.   //处理选择爱好的请求方法
11.   @RequestMapping("/hobby")
12.   public String hobby(Model model) {
13.       //创建一个 user 对象
14.       User user=new User();
15.       //hobbys 用于存放当前用户的所有爱好
16.       List<String>hobbys=new ArrayList<String>();
17.       hobbys.add("篮球");
18.       hobbys.add("游泳");
19.       //将 hobbys 赋值给 user 对象
20.       user.setHobbys(hobbys);
21.       //用于在页面上显示所有爱好的选项
22.       List<String>hobbyList=new ArrayList<String>();
23.       hobbyList.add("篮球");
24.       hobbyList.add("围棋");
25.       hobbyList.add("游泳");
26.       hobbyList.add("画画");
27.       //将 user 保存在 Model 中
28.       model.addAttribute("user",user);
```