第5章 Spring Boot实现Web MVC

广东财经大学信息学院
罗 东 俊 博士
ZSUJONE@126.COM

（内部资料，请勿外传）

目的和要求

- 熟悉Thymeleaf模板引擎基本语法。
- 掌握Spring Boot整合Thymeleaf模板引擎的使用。
- 掌握Spring Boot整合Spring MVC的使用。
- 掌握拦截器的作用和使用方法。
- 掌握Spring Boot中MVC功能的定制。

主要内容

- 5.1 MVC设计概述
- 5.2 使用视图技术Thymeleaf
- 5.3 使用控制器
- 5.4 使用拦截器
- 5.5 自定义Web MVC配置

5.3 使用控制器

- 5.3.1 基本用法
- 5.3.2 数据绑定
- 5.3.3 数据校验

5.3.1 基本用法

- 1.定义控制器
- 2.定义请求映射

1.定义控制器

- 在Spring MVC中，控制器负责处理由DispatcherServlet接收并分发过来的请求，它把用户要请求的数据封装成一个Model，然后把该Model返回给对应的View进行展示。
- 定义控制器的注解：
  - @Controller
  - @RestController

@Controller

- @Controller注解用于将控制层（Controller）的类标识为Spring容器中的Bean。
  - ✧分发处理器会通过Spring的扫描机制找到该Bean实例，并检测其中的方法是否为URL请求映射方法。

@RestController

- 该注解为组合注解，等价于@Controller+@ResponseBody注解，用来标注Rest风格的控制器类。
  - @ResponseBody注解不用再标注在URL请求处理方法上；
  - 控制器类中的URL请求处理方法会直接返回JSON字符串。

示例代码

```
package com.itheima.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloController {
   @GetMapping("/hello")
   public String hello(){
       return "hello Spring Boot";
    }
}
```

该注解为组合注解，等同于Spring中@Controller+@ResponseBody注解

等同于Spring框架中@RequestMapping(RequestMethod.GET)注解

HelloController：web请求处理控制类；

@RestController：将当前类作为控制层组件添加到Spring容器（ApplicationContext）中，同时当前类的处理方法会返回JSON字符串；

@GetMapping：设置方法的web访问路径并限定其访问方式是Get。

2.定义请求映射

- 定义请求映射，即定义URL请求和Controller方法之间的映射，需要考虑以下3方面：
  - ✧1）使用@RequestMapping
  - ✧2）定义方法参数类型
  - ✧3）定义方法返回类型

1）使用@RequestMapping

- @RequestMapping用于映射一个URL请求到一个方法上。
- 使用时，它可标注在一个方法或一个类上。
  - ✧标注在方法上：作为请求处理方法在程序接收到对应的URL请求时被调用。
  - ✧标注在类上：表示类中所有响应请求的方法都以该地址作为父路径。

标注在方法上

```
package com.itheima.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class FirstController{
        @RequestMapping(value="/firstController")
        public ModelAndView handleRequest(HttpServletRequest request,
                                HttpServletResponse response) {
            ...
            return mav;
        }
}
```

此时，可以通过地址：**http://localhost:8080/firstController访问该方法！**

标注在类上

```
package com.itheima.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping(value="/hello")
public class FirstController{
        @RequestMapping(value="/firstController")
        public ModelAndView handleRequest(HttpServletRequest request,
                                HttpServletResponse response) {
            ...
            return mav;
        }
}
```

此时，须通过地址：**http://localhost:8080/hello/firstController**访问该方法！

@RequestMapping的属性

属性名	类型	描述
name	String	可选属性，用于为映射地址指定别名。
value	String[]	可选属性，同时也是默认属性，用于映射一个请求和一种方法，可以标注在一个方法或一个类上。
method	RequestMethod[]	可选属性，用于指定该方法用于处理哪种类型的请求方式，其请求方式包括 GET、POST、HEAD、OPTIONS、PUT、PATCH、DELETE 和 TRACE 例如 method=RequestMethod.GET 表示只支持 GET 请求，如果需要支持多个请求方式则需要通过{}写成数组的形式，并且多个请求方式之间是有英文逗号分隔。
params	String[]	可选属性，用于指定 Request 中必须包含某些参数的值，才可以通过其标注的方法处理。
headers	String[]	可选属性,用于指定 Request 中必须包含某些指定的 header 的值，才可以通过其标注的方法处理。
consumes	String[]	可选属性，用于指定处理请求的提交内容类型（Content-type），比如 application/json，text/html 等。
produces	String[]	可选属性，用于指定返回的内容类型，返回的内容类型必须是 request 请求头（Accept）中所包含的类型。

@RequestMapping的默认属性

- 表中所有属性都是可选的，但其默认属性是**value**。
- 当**value**是其唯一属性时，可以省略属性名，并且默认是**GET**方式。
  - ✧例如，下面两种标注的含义相同：
    - **@RequestMapping(value="/firstController")**
    - **@RequestMapping("/firstController")**

@RequestMapping的组合注解

- Spring框架的4.3版本中，引入了新的组合注解，来帮助简化常用的HTTP请求的映射，并更好的表达被注解方法的语义。
  - ✧@GetMapping：匹配GET方式的请求；
  - ✧@PostMapping：匹配POST方式的请求；
  - ✧@PutMapping：匹配PUT方式的请求；
  - ✧@DeleteMapping：匹配DELETE方式的请求；
  - ✧@PatchMapping：匹配PATCH方式的请求。

代码示例

- 在实际开发中，传统的@RequestMapping注解使用方式如下：

```
@RequestMapping(value="/user/{id}",method=RequestMethod.GET)
public String selectUserById(String id){
    ...
}
```

- 使用@GetMapping注解后的简化代码如下：

```
@GetMapping("/user/{id}")
public String selectUserById(String id){
    ...
}
```

2）定义方法参数类型

- 在控制器类中，每一个请求处理方法可以有多个不同类型的参数，也可以有不同类型的返回结果。

不同参数类型

- javax.servlet.ServletRequest / javax.servlet.http.HttpServletRequest
- javax.servlet.ServletResponse / javax.servlet.http.HttpServletResponse
- javax.servlet.http.HttpSession
- org.springframework.web.context.request.WebRequest或
- org.springframework.web.context.request.NativeWebRequest
- java.util.Locale
- java.util.TimeZone (Java 6+) / java.time.ZoneId (on Java 8)
- java.io.InputStream / java.io.Reader
- java.io.OutputStream / java.io.Writer
- org.springframework.http.HttpMethod
- java.security.Principal

不同参数类型

- @PathVariable、@MatrixVariable、@RequestParam、@RequestHeader、@RequestBody、@RequestPart、@ModelAttribute、 @SessionAttribute、@RequestAttribute注解
- HttpEntity<?>
- java.util.Map / org.springframework.ui.Model /org.springframework.ui.ModelMap
- org.springframework.web.servlet.mvc.support.RedirectAttributes
- org.springframework.validation.Errors /org.springframework.validation.BindingResult
- org.springframework.web.bind.support.SessionStatus
- org.springframework.web.util.UriComponentsBuilder

如果方法中添加了Model参数，则每次调用该请求处理方法时，Spring MVC都会创建Model对象，并将其作为参数传递给方法。

3）定义方法返回类型

- Spring MVC所支持的常见请求处理方法返回类型如下：
  - ✧ModelAndView：可同时返回数据和视图
  - ✧Model
  - ✧Map
  - ✧View
  - ✧String：可以跳转视图，但不能携带数据
  - ✧void：只返回数据，而不会跳转视图
  - ✧HttpEntity<?>或ResponseEntity<?>
  - ✧Callable<?>
  - ✧DeferredResult<?>

通常做法

- 由于**ModelAndView**类型未能实现数据与视图之间的解耦，所以在企业开发时，方法的返回类型通常使用**String**，并通过**Model**参数类型将数据带入视图页面。

```
@GetMapping("/firstController")
public String handleRequest(HttpServletRequest request,
                            HttpServletResponse response, Model model) {
    model.addAttribute("msg", "这是我的第一个Spring Boot Web MVC程序");
    return "first";
}
```

跳转到first.html视图

重定向和请求转发

- return “redirect:/first”
  - ✧重定向到“/first”请求处理方法，并要求目标请求处理方法的method方式为GET。
  - ✧重定向后客户端浏览器的网址是目标URL地址
- return “forward:/first”
  - ✧请求转发到“/first”请求处理方法，并要求目标请求处理方法与当前请求处理方法的method方式一致。
  - ✧转发后客户端浏览器的网址不会发生变化
- 以上重定向（转发）的目标均只能是请求处理方法，不能直接是html视图页面。
  - ✧不带“redirect”或“forward”的跳转目标只能直接是html视图页面，而不能是请求处理方法。

示例代码

■用法一：

```
@PostMapping("/checkUser")
public String checkUser(@ModelAttribute User user) {
    return " results";
}
```

该用法通过@ModelAttribute注解将绑定的数据user直接携带到results.html视图页面。

示例代码

- 用法二：

```
@PostMapping("/checkUser")
public String checkUser(@ModelAttribute User user) {
    return "forward:/toResults";
}
@PostMapping("/toResults")
public String toResults(){
    return "results";
}
```

该用法通过@ModelAttribute注解将绑定的数据user通过请求处理方法携带到results.html视图页面。

示例代码

■用法三：

```
@PostMapping("/checkUser")
public String checkUser(User user, RedirectAttributes attr) {
    // attr.addAttribute(“user”, user); //user作为参数跟在URL后被传递
    attr.addFlashAttribute(“user”, user); //user暂存在session中，而不是作为参数跟在URL后被传递
    return "redirect:/toResults";
}
@GetMapping("/toResults")
public String toResults(){
    return "results";
}
```

该用法须通过RedirectAttributes将绑定的数据user通过请求处理方法携带到results.html视图页面。

5.3.2 数据绑定

- 1.数据绑定介绍
- 2.数据绑定方式
- 3.数据绑定案例

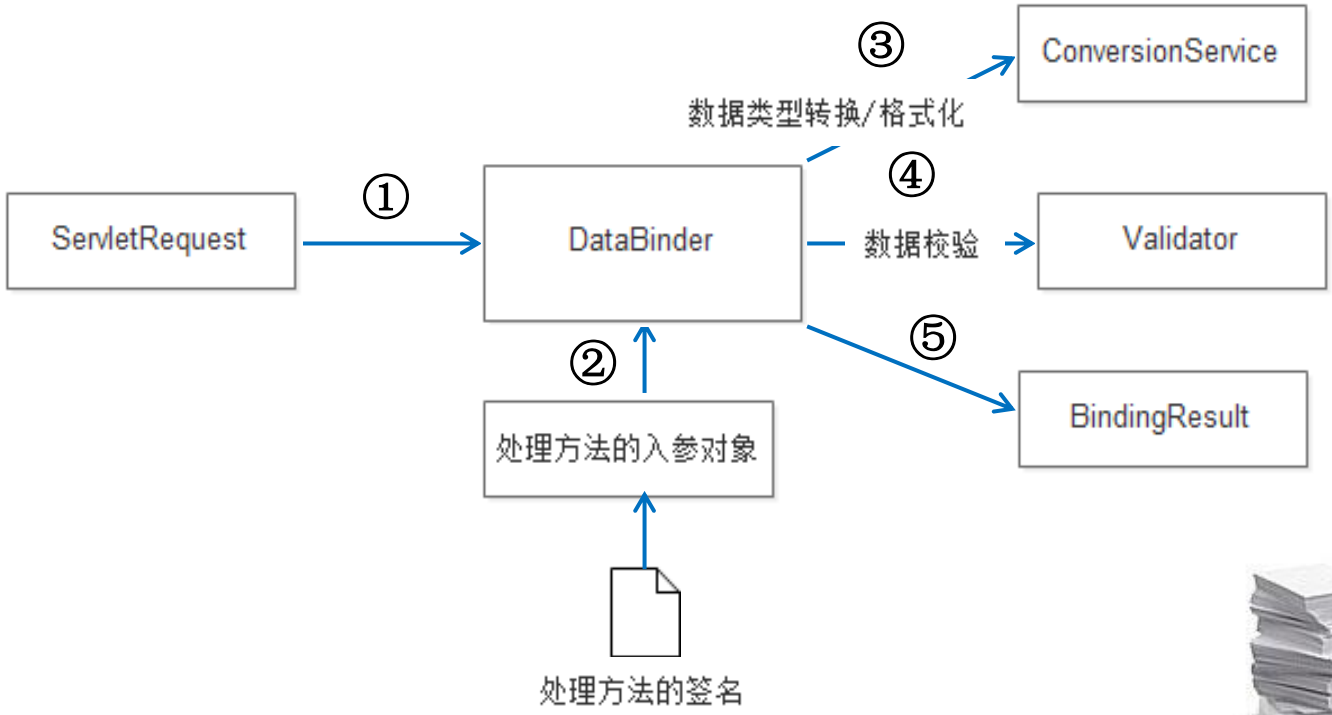
1.数据绑定介绍

- 在执行程序时，Spring Boot会根据客户端请求参数的不同，将请求消息中的信息以一定的方式转换并绑定到控制器类的方法参数中。
- 这种将请求消息数据与后端方法参数建立连接的过程就是Spring Boot中的数据绑定。

数据绑定的过程

- 在数据绑定过程中，**Spring Boot**框架会通过数据绑定组件（**DataBinder**）将请求参数串的内容进行类型转换，然后将转换后的值赋给控制器类中方法的形参，这样后台方法就可正确绑定并获取客户端请求携带的参数（如：**URL**参数、表单参数、**JSON**消息等）。

数据绑定的过程


③
ConversionService
数据类型转换/格式化
④
ServletRequest
①
DataBinder
数据校验
Validator
⑤
②
BindingResult
处理方法的入参对象
处理方法的签名

数据绑定的过程

- ①Spring Boot将ServletRequest对象传递给DataBinder；
- ②将处理方法的参数对象传递给DataBinder；
- ③DataBinder调用ConversionService组件进行数据类型转换、数据格式化等工作，并将ServletRequest对象中的消息填充到参数对象中；
- ④调用Validator组件对已经绑定了请求消息数据的参数对象进行数据合法性校验；
- ⑤校验完成后会生成数据绑定结果BindingResult对象，Spring Boot会将BindingResult对象中的内容赋给处理方法的相应参数。

2.数据绑定方式

- 根据客户端请求参数形式和个数不同，请求消息数据可绑定到以下类型的后端方法形参中：
  - ✧1）默认数据类型
  - ✧2）基本数据类型
  - ✧3）POJO类型
  - ✧4）数组/列表类型
  - ✧5）包装类型
  - ✧6）日期类型

1）默认数据类型

- 当前端请求的参数比较简单时，可以在后台方法的形参中直接使用Spring Boot提供的默认参数类型进行数据绑定。
- 常用默认参数类型：
  - ✧HttpServletRequest：通过request对象获取请求信息；
  - ✧HttpServletResponse：通过response处理响应信息；
  - ✧HttpSession：通过session对象得到session中存放的对象；
  - ✧Model/ModelMap：Model是一个接口，ModelMap是一个接口实现，作用是将model数据填充到request域。

示例代码

后端

```
@Controller
public class UserController {
        @RequestMapping("/selectUser")
         public String selectUser(HttpServletRequest request) {
                String id = request.getParameter("id");
                System.out.println("id="+id);
                return "success";
            }
}
```

前端

```
http://localhost:8080/selectUser?id=1
```

结果

```
id=1
```

2）基本数据类型

- 当前端请求的参数比较简单时，还可在后台方法的形参中使用基本数据类型进行数据绑定，例如**int**、**String**、**Double**等类型。
- 当前端请求中参数名和后台控制器类方法中的形参名不一样时，可以用**@RequestParam**注解进行间接数据绑定。

@RequestParam的属性声明

属性	说明
value	name 属性的别名，这里指参数的名字，即入参的请求参数名字，如 value="item_id"表示请求的参数中名字为 item_id 的参数的值将传入。如果只使用 vaule 属性，则可以省略 value 属性名。
name	指定请求头绑定的名称。
required	用于指定参数是否必须，默认是 true，表示请求中一定要有相应的参数。
defaultValue	默认值，表示如果请求中没有同名参数时的默认值。

示例代码

后端

```
@Controller
public class UserController {
    @RequestMapping("/selectUser")
    public String selectUser(@RequestParam("user_id") Integer id) {
        System.out.println("id="+id);
        return "success";
    }
}
```

先用@RequestParam接收同名参数，后间接绑定到方法形参上

前端

```
http://localhost:8080/selectUser?user_id=1
```

结果

```
id=1
```

示例代码

后端

```
@Controller
public class UserController {
    @RequestMapping("/user/{user_id}")
    public String selectUser(@PathVariable("user_id") Integer id) {
        System.out.println("id="+id);
        return "success";
    }
}
```

先用@PathVariable接收同名参数，后间接绑定到方法形参上

前端

http://localhost:8080/user/1 (RESTful风格URL)

结果

id=1

3）POJO类型

- **POJO类型的数据绑定就是将所有关联请求的参数封装在一个POJO （Plain Ordinary Java Object）中，然后在方法中直接使用该POJO作为形参来完成数据绑定。**
  - ✧在使用**POJO**类型数据绑定时，前端请求的参数名（如**form**表单内各元素的**name**属性值）必须与要绑定的**POJO**类中的属性名一样

示例代码

后端

```
@Controller
public class UserController {
    @PostMapping("/registerUser")
    public String registerUser(User user) {
        String username = user.getUsername();
        Integer password = user.getPassword();
        System.out.println("username="+username);
        System.out.println("password="+password);
        return "success";
    }
}
```

前端

```
<form class="form-signin col-md-3 " th:action="@{/registerUser}"  method="post">
    <input class="form-control" type="text" name="username" th:placeholder="用户名"/>
    <input class="form-control" type="text" name="password" th:placeholder="密码"/>
    <button class="btn btn-lg btn-primary btn-block" type="submit">注册</button>
</form>
```

4）数组/列表类型

- 如果前端请求的参数是批量数据，且都是同类型的基本数据类型时，可在后台方法的形参中使用基本数据类型的数组/列表类型进行数据绑定。

示例代码

后端

```
@Controller
public class UserController {
   @RequestMapping("/deleteUsers")
    public String deleteUsers(Integer[] ids) {
        if(ids !=null){
            for (Integer id : ids) {System.out.println("删除了id为"+id+"的用户！ ");}
         }else{System.out.println("ids=null");}
            return "success";
    }
}
```

前端

```
<form class="form-signin col-md-3 " th:action="@{/deleteUsers}"  method="post">
    <table width="20%" border=1>
            <tr><td><input name="ids" value="1" type="checkbox"></td><td>tom</td></tr>
            <tr><td><input name="ids" value="2" type="checkbox"></td><td>jack</td></tr>
    </table>
    <button class=“btn btn-lg btn-primary btn-block” type=“submit”>删除</button>
</form>
```

5）包装类型

- 所谓包装类型，是指包含有基本数据类型、POJO类型、日期类型、List类型、数组类型等多种类型属性的复杂对象类型。
  - ✧如果属性是基本数据类型或其List/数组类型，则前端请求的参数名直接用对应的属性名；
  - ✧如果属性是POJO类型，则前端请求的参数名必须为【对象.属性】，其中【对象】为该POJO类型对象，【属性】为该POJO类型对象的属性；
  - ✧如果属性是POJO类型的List/数组类型，则前端请求的参数名必须为【对象[i].属性】，其中【对象】为该List/数组对象，【属性】为该POJO类型对象的属性。

示例代码

包装类

```
public class UserVO {
    private User[] users;
    private User user;
    private Integer[] ids;

    public void setIds(Integer[] ids) { this.ids = ids;}
    public Integer[] getIds() {return ids;}
    public void setUser(User user) {this.user = user;}
    public User getUser() {return user;}
    public User[] getUsers() {return users;}
    public void setUsers(User[] users) {this.users = users;}
}
```

示例代码

后端

```
@Controller
public class UserController {
    @RequestMapping("/registerUser")
    public String registerUser(UserVO uservo) {
            User[] users=uservo.getUsers();
            Integer[] ids=uservo.getIds();
            User user=uservo.getUser();
            System.out.println(users[0].getUsername());
            System.out.println(ids[1]);
            System.out.println(user.getUsername());
            return "success";
    }
}
```

示例代码

前端

```
<form class="form-signin col-md-3 " th:action="@{/registerUser}"  method="post">
   <input class="form-control" type="text" name="users[0].username" th:placeholder="列表用户名"/>
   <input class="form-control" type="text" name="ids" th:placeholder="ID0"/>
   <input class="form-control" type="text" name="ids" th:placeholder="ID1"/>
   <input class="form-control" type="text" name="user.username" th:placeholder="个体用户名"/>
    <button class=“btn btn-lg btn-primary btn-block” type=“submit”>注册</button>
</form>
```

JSON实现方式

后端

```
@RestController
public class UserController {
    @RequestMapping("/registerUser")
    public User registerUser(@RequestBody UserVO uservo) {
            User[] users=uservo.getUsers();
            Integer[] ids=uservo.getIds();
            User user=uservo.getUser();
            System.out.println(users[0].getUsername());
            System.out.println(ids[1]);
            System.out.println(user.getUsername());
            return user;
    }
}
```

前端

定义发送请求的数据格式为JSON字符串。

关于JSON数据结构

- JSON有如下数据结构：
  - ①对象结构
  - ②数组结构
  - ③组合结构

①对象结构

- 以“{”开始，以“}”结束，中间部分由0个或多个以英文“，”分隔的name:value对构成（注意name和value之间以英文“：”分隔）
- 例如：一个address对象包含城市、街道、邮编等信息，使用JSON的表示形式如下：
  - {"city":"Beijing","street":"Xisanqi","postcode":100096}

②数组结构

- 以“[”开始，以“]”结束，中间部分由0个或多个以英文“，”分隔的值的列表组成。
- 例如，一个数组包含了**String、Number、Boolean、null**类型数据，使用**JSON**的表示形式如下：
  - **["abc",12345,false,null]**
- 如果使用**JSON**存储单个数据（如“**abc**”），一定要使用数组的形式，不要使用**Object**形式，因为**Object**形式必须是“名称：值”的形式。

③组合结构

- 对象、数组数据结构也可以分别组合构成更为复杂的数据结构。
- 例如：一个**person**对象包含**name**、**hobby**和**address**对象，其代码表现形式如下：

```
{
        "name": "zhangsan"
        "hobby":["篮球","羽毛球","游泳"]
        "address":{
                "city":"Beijing"
                "street":"Xisanqi"
                "postcode":100096
        }
 }
```

6）日期类型

- 当前端请求参数是日期字符串，要与后台的Date类型形参进行绑定时，需要开发者自定义转换器——自定义数据绑定。
- Spring框架提供了一个Converter用于将一种类型的对象转换为另一种类型的对象。

自定义数据绑定案例

- 本案例实现将日期格式为“yyy-MM-dd HH:mm:ss”的字符串与后台Date类型形参进行数据绑定。
- 搭建步骤：
  - ①创建转换器
  - ②配置转换器
  - ③创建Web控制器
  - ④效果测试

①创建转换器

- 在chapter05项目中，新建一个com.itheima.convert包，在包中新建一个日期转换类DateConverter，实现将String类型转换成Date类型。

DateConverter类

```
import org.springframework.core.convert.converter.Converter;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class DateConverter implements Converter<String, Date> {
    private String datePattern = "yyyy-MM-dd HH:mm:ss";                    // 定义日期格式
    @Override
    public Date convert(String source) {
        SimpleDateFormat sdf = new SimpleDateFormat(datePattern);          // 格式化日期
        try {
            return sdf.parse(source);
        } catch (ParseException e) {
            throw new IllegalArgumentException(
                    "无效的日期格式，请使用这种格式:" + datePattern);
        }
    }
}
```

目标类型

源类型

②配置转换器

- 在chapter05项目的com.itheima.config包中新建一个自定义配置类ConverterConfig，用于对日期转换器的配置，应用程序会应用配置的转换器对字符串日期进行数据绑定。

ConverterConfig类

```
import com.itheima.convert.DateConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ConverterConfig{
    @Bean
    public DateConverter myDateConverter(){
        return new DateConverter();
    }
}
```

③创建Web控制器

- 在chapter05项目的com.itheima.controller包中，新建一个Web控制类DateController，并在类中编写绑定日期数据的方法。

DateController类

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import java.util.Date;
@Controller
public class DateController {
        //使用自定义转换器绑定日期数据
        @RequestMapping("/customDate")
        public String customDate(Date date) {
            System.out.println("date="+date);
            return "success";
        }
}
```

④效果测试

前端	http://localhost:8080/customDate?date=2021-04-16 15:55:55
结果	Fri Apr 16 15:55:55 CST 2021

3.数据绑定案例

- 本案例通过将表单数据绑定到数组类型形参，实现多文件的上传，同时实现各类型文件的下载。
  - ✧**Spring Boot**将上传文件自动绑定到**MultipartFile**对象，通过**transferTo(File dest)**方法将文件上传到服务器磁盘中
  - ✧通过文件下载工具**FileUtils**的**readFileToByteArray(file)**方法将服务器文件下载到本地磁盘

MultipartFile的常用方法

- byte[] getBytes()：获取文件数据。
- String getContentType()：获取文件MIME类型，如image/jpeg等。
- InputStream getInputStream()：获取文件流。
- String getName()：获取表单中文件组件的名字。
- String getOriginalFilename()：获取上传文件原名。
- long getSize()：获取文件字节大小，单位为byte。
- boolean isEmpty()：是否有（选择）上传文件。
- void transferTo(File dest)：将上传文件保存到一个目标文件中。

文件上传与下载

- 文件上传与下载实现步骤：
  - ✧①引入文件下载工具依赖
  - ✧②设置上传文件大小限制
  - ✧③创建Web控制器
  - ✧④创建文件上传页面
  - ✧⑤创建文件下载页面
  - ✧⑥效果测试

①引入文件下载工具依赖

```
<!--文件下载工具依赖-->
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>
```

②设置上传文件大小限制

- 在全局配置文件application.properties中添加文件上传功能的相关设置

```
# 单个上传文件大小限制（默认1MB）
spring.servlet.multipart.max-file-size=10MB
# 总上传文件大小限制（默认10MB）
spring.servlet.multipart.max-request-size=50MB
```

③创建Web控制器

- 在chapter05项目的com.itheima.controller包中，新建一个Web控制类FileController，实现向文件上传页面跳转和文件上传管理，以及向文件下载页面跳转和文件下载管理。

FileController类

```
import org.springframework.web.multipart.MultipartFile;
import org.apache.commons.io.FileUtils;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import javax.servlet.http.HttpServletRequest;
import java.io.File;
import java.net.URLEncoder;
import java.util.UUID;
@Controller
public class FileController {
   @GetMapping("/toUpload")                                          // 向文件上传页面跳转
    public String toUpload() {
        return "upload";
    }
```

FileController类

```
@PostMapping("/uploadFile")                                          // 文件上传管理
 public String uploadFile(MultipartFile[] fileUpload, HttpServletRequest request, Model model) {
    model.addAttribute("uploadStatus", 1);                           // 默认文件上传成功(返回状态1)
    for (MultipartFile file : fileUpload) {
       String fileName = file.getOriginalFilename();                 //获取文件名以及后缀名
       fileName = UUID.randomUUID() + "_" + fileName;                //重新生成文件名（根据具体情况生成对应文件名）
      String dirPath = request.getServletContext().getRealPath(“/filePool/”);   // 将文件上传到指定的服务器目录filePool
       File filePath = new File(dirPath);
       if (!filePath.exists()) {
          filePath.mkdirs();
       }
       try {
          file.transferTo(new File(dirPath + fileName));             //将文件存储到服务器目标文件
       } catch (Exception e) {
          e.printStackTrace();
          model.addAttribute("uploadStatus", "上传失败： " + e.getMessage());    //上传失败，返回状态信息
       }
    }
    return "upload";
 }
```

FileController类

```
@GetMapping("/toDownload")                                          // 向文件下载页面跳转
public String toDownload(HttpServletRequest request, Model model) {
    String path = request.getServletContext().getRealPath("/filePool/");
    File fileDir = new File(path);
    File filesList[] = fileDir.listFiles();                          //从指定目录获得文件列表
    model.addAttribute("filesList", filesList);
    return "download";
}
@GetMapping("/downloadFile")                                        // 所有类型文件下载管理
public ResponseEntity<byte[]> downloadFile(HttpServletRequest request, String filename) throws Exception {
    String path = request.getServletContext().getRealPath("/filePool/");          //设置下载文件路径
    File file = new File(path + File.separator + filename);                        //构建将要下载的文件对象
    filename = getFilename(request, filename);                                     // 通知浏览器以下载方式打开（下载前对文件名进行转码）
    HttpHeaders headers = new HttpHeaders();                                       // 设置响应头
    headers.setContentDispositionFormData("attachment", filename);
    headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);   // 定义以流的形式下载返回文件数据
    try {
        return new ResponseEntity<>(FileUtils.readFileToByteArray(file), headers, HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        return new ResponseEntity<byte[]>(e.getMessage().getBytes(), HttpStatus.EXPECTATION_FAILED);
    }
}
```

FileController类

```
    // 根据浏览器的不同进行编码设置，返回编码后的文件名
    private String getFilename(HttpServletRequest request, String filename)  throws Exception {
        // IE不同版本User-Agent中出现的关键词
        String[] IEBrowserKeyWords = {"MSIE", "Trident", "Edge"};
        // 获取请求头代理信息
        String userAgent = request.getHeader("User-Agent");
        for (String keyWord : IEBrowserKeyWords) {
            if (userAgent.contains(keyWord)) {
                //IE内核浏览器，统一为UTF-8编码显示，并对转换的+进行更正
                return URLEncoder.encode(filename, "UTF-8").replace("+", " ");
            }
        }
        //火狐等其它浏览器统一为ISO-8859-1编码显示
        return new String(filename.getBytes("UTF-8"), "ISO-8859-1");
    }
}
```

④创建文件上传页面

- 在**chapter05**项目**resources**的**templates**目录下，新建一个选择文件上传视图页面**upload.html**，在页面中使用表单上传文件。

文件上传页面upload.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>文件上传界面</title>
    <link rel="stylesheet" th:href="@{/login/css/bootstrap.min.css}">
</head>
<body class="text-center">
<h3 class="panel-title">文件上传示例</h3>
<div th:if="${uploadStatus}==1">
    <a style="color: red" th:href="@{/toDownload}">上传成功，请去下载</a>
</div>
<div th:unless="${uploadStatus}==1">
    <span style="color: red">[[${uploadStatus}]]</span>
</div>
<form class="form-signin col-md-3 " th:action="@{/uploadFile}" method="post" enctype="multipart/form-data">
    <input class="form-control" type="file" name="fileUpload" th:placeholder="选择文件"/>
    <input class="form-control" type="file" name="fileUpload" th:placeholder="选择文件"/>
    <button class="btn btn-lg btn-primary btn-block" type="submit">上传文件</button>
</form>
</body>
</html>
```

⑤创建文件下载页面

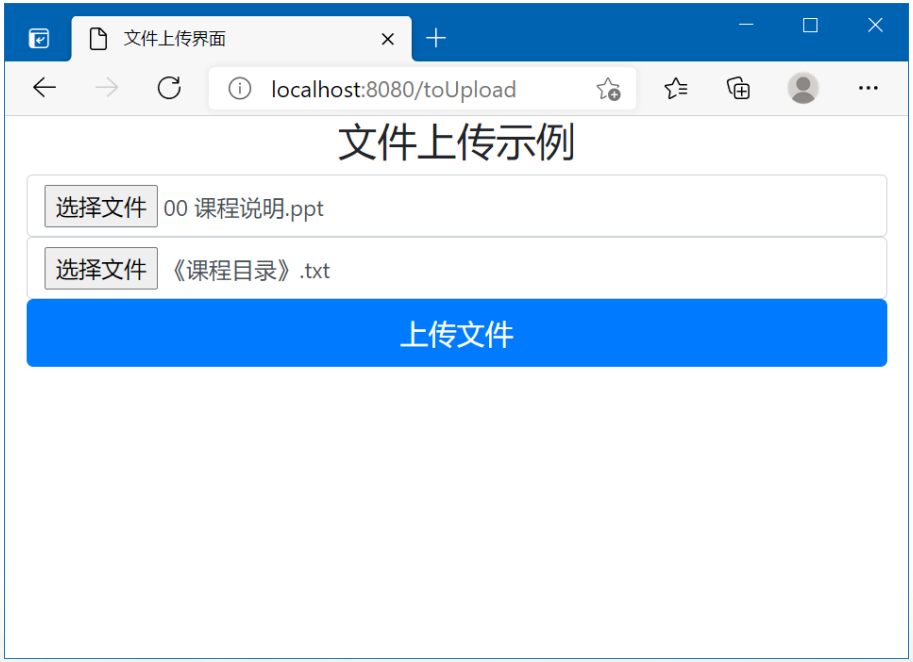
- 在**chapter05**项目**resources**的**templates**目录下，新建一个文件下载视图页面**download.html**，在页面上显示可下载的文件列表。

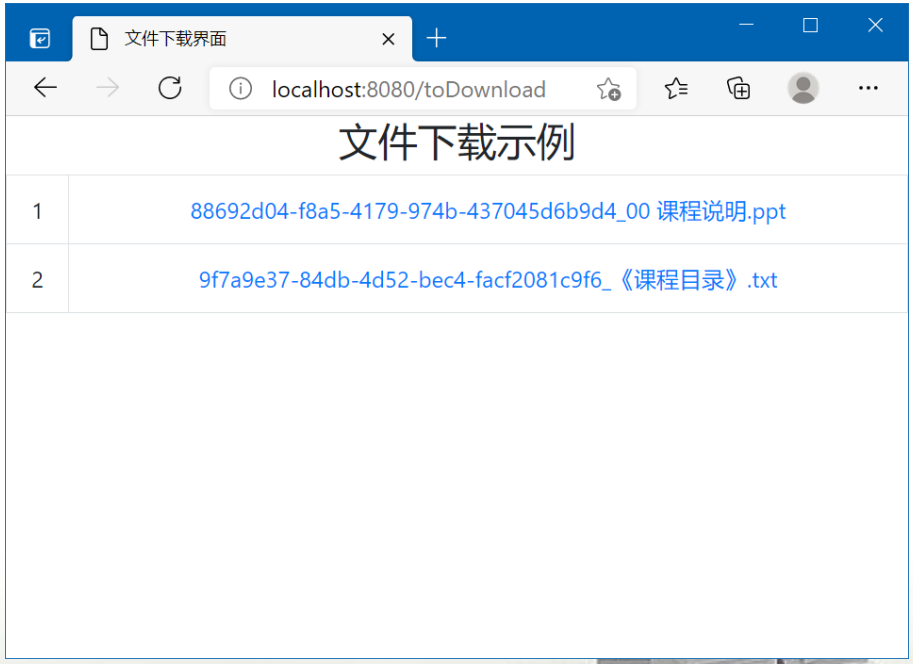
文件下载页面download.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>文件下载界面</title>
    <link rel="stylesheet" th:href="@{/login/css/bootstrap.min.css}">
</head>
<body class="text-center">
<h3 class="panel-title">文件下载示例</h3>
<table class="table table-bordered table-hover">
    <tbody class="text-center">
    <tr th:each="file,fileStat:${filesList}">
        <td>
            <span th:text="${fileStat.count}"></span>
        </td>
        <td>
            <!--file.name相当于调用getName()方法获得文件名称  -->
            <a th:href="@{downloadFile(filename=${file.name})}">
                <span th:text="${file.name}"></span>
            </a>
        </td>
    </tr>
    </tbody>
</table></body></html>
```

⑥效果测试

- 启动项目进行测试，在浏览器上访问 **http://localhost:8080/toUpload**





5.3.3 数据校验

- 用户注册时，提交的表单数据通常需要进行数据校验。
  - ✧前端JavaScript程序预校验（可绕过）
  - ✧后端校验
- **Spring Boot内置的hibernate-validator是JSR（Java Specification Requests）标准的实现，提供了一套常用的校验注解，可直接应用在对象的属性上，从而实现对参数优雅的后端数据校验。**

1.常用数据校验注解

- 常用的数据校验注解可分以下几大类：
  - ✧①空检查注解
  - ✧②booelan检查注解
  - ✧③长度检查注解
  - ✧④日期检查注解
  - ✧⑤数值检查注解

①空检查注解

- **@Null：检查值是否为null。**
- **@NotNull：检查值是否不为null。**
- **@NotBlank：检查字符序列去前后空格后的长度是否大于0。**
- **@NotEmpty：检查约束元素是否为null或是empty（[]）。**
- **@NotEmptyPattern：在字符串不为空的情况下，检查是否匹配正则表达式。**

```
// goods.gname.required为属性文件的错误代码
@NotBlank(message="{goods.gname.required}")
private String gname;
```

②booelan检查注解

- **@AssertTrue：验证boolean属性是否为true。**
- **@AssertFalse：验证boolean属性是否为false。**

```
@AssertTrue
private boolean isLogin;
```

③长度检查注解

- **@Size(min=, max=)：检查对象（Array，Collection，Map，String）的元素个数是否在min（含）和max（含）之间**
- **@Length(min=, max=)：检查字符串长度是否在min（含）和max（含）之间。**

```
@Length(min=1,max=100)
private String gdescription;
```

④日期检查注解

- **@Past：检查日期是否在过去，即小于当前日期。**
- **@PastOrPresent：检查日期是否在过去或现在，即小于等于当前日期。**
- **@Future：检查日期是否在未来，即大于当前日期。**
- **@FutureOrPresent：检查日期是否在现在或将来，即大于等于当前日期。**

```
@Past(message="{gdate.invalid}")
private Date gdate;
```

日期检查注解

- **@DateValidator：验证日期字符串格式是否满足正则表达式，Local为ENGLISH。**
- **@DateFormatCheckPattern：验证日期字符串格式是否满足正则表达式，Local为自己手动指定的。**
- **@Pattern(regex=,flag=)：检查字符串是否与正则表达式 regex 匹配。**
- **@ListStringPattern(regex=,flag=)：验证List中的字符串是否满足正则表达式。**
- **@NotEmptyPattern(regex=)：在字符串不为空的情况下，验证是否匹配正则表达式。**

⑤数值检查注解

- **@Min (value=)：检查值是否大于或等于指定的最小值。**
- **@Max(value=)：检查值是否小于或等于指定的最大值。**
- **@DecimalMax(value=, inclusive=)：检查带注解的值是否小于（inclusive=false）或等于（inclusive=true）指定的最大值（value）。value是一个根据BigDecimal字符串表示的最大值。**
- **@DecimalMin(value=, inclusive=)：检查带注解的值是否大于（inclusive=false）或等于（inclusive=true）指定的最小值（value）。value是一个根据BigDecimal字符串表示的最小值。**

数值检查注解

- **@Negative**：检查元素是否严格为负数。零值被认为无效。
- **@NegativeOrZero**：检查元素是否为负或零。
- **@Positive**：检查元素是否严格为正。零值被视为无效。
- **@PositiveOrZero**：检查元素是否为正或零。

数值检查注解

- **@Digits：验证Number和String的构成是否合法。**
- **@Digits(integer=,fraction=)：验证字符串是否是符合指定格式的数字，integer指定整数精度，fraction指定小数精度。**
- **@Range(min=, max=)：检查数字是否介于min和max之间。**

```
@Range(min=0,max=100,message="{gprice.invalid}")
private double gprice;
```

数值检查注解

- **@Valid**：用于标记对关联对象进行校验，将验证在对象及其属性上定义的约束。如果关联对象是个集合或者数组，那么对其中的元素进行校验，如果是一个**map**，则对其中的值部分进行校验。
- **@CreditCardNumber**：信用卡号码验证。
- **@Email**：检查指定的字符序列是否为有效的电子邮件地址。如果为**null**，不进行验证，通过验证。

2.自定义数据校验注解

- Spring Boot的数据校验功能可以满足大多数的验证需求，但如果在系统内需要实现一些其他校验功能，则可根据规则进行自定义。
- 自定义校验需要提供两个类：
  - ✧①自定义校验注解类
  - ✧②自定义校验业务逻辑类

自定义校验注解类示例

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD})                    // 限定使用范围——只能在字段上使用
@Retention(RetentionPolicy.RUNTIME)      // 表明注解的生命周期，它在代码运行时可以通过反射获取到注解
@Constraint(validatedBy = MyPasswordValidator.class)      // validatedBy属性指定该注解的校验逻辑
public @interface MyPasswordConstraint {
    String message() default "Invalid Password";                    // 定义错误提示
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

自定义校验业务逻辑类示例

```
import org.passay.*;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import java.util.Arrays;

//String为校验的类型
public class MyPasswordValidator  implements ConstraintValidator<MyPasswordConstraint, String> {
    @Override
    //初始化校验消息，得到配置的注解内容
    public void initialize(MyPasswordConstraint myPasswordConstraint) {


    }
```

自定义校验业务逻辑类示例

```
@Override
//自定义校验逻辑
public boolean isValid(String password, ConstraintValidatorContext validatorContext) {
    PasswordValidator validator = new PasswordValidator(Arrays.asList(
            new LengthRule(8, 30),                                          //密码长度为8到30位
            new CharacterRule(EnglishCharacterData.UpperCase, 1),           //至少有一个英文的大写字母
            new CharacterRule(EnglishCharacterData.LowerCase, 1),           //至少有一个英文的小写字母
            new CharacterRule(EnglishCharacterData.Special, 1),             //至少有一个英文的特殊字符
            new IllegalSequenceRule(EnglishSequenceData.Alphabetical, 5, false),  //不允许有5个连续的英文字母
            new IllegalSequenceRule(EnglishSequenceData.Numerical, 5, false),     //不允许有5个连续的数字
            new IllegalSequenceRule(EnglishSequenceData.USQwerty, 5, false),      //不允许有5个键盘连续的字母
            new WhitespaceRule()                                            //需要有空格
    ));
    RuleResult result = validator.validate(new PasswordData(password));
    return result.isValid();
}
}
```

3.数据校验案例

- 本案例基于Spring Boot内置的hibernate-validator工具，以Thymeleaf为视图技术，实现对用户注册页面表单数据（用户名、密码、电话号码等）的后端数据校验和前端错误提示功能。
- 搭建步骤：
  - ✧①引入依赖启动器
  - ✧②自定义数据校验注解
  - ✧③创建持久化类
  - ✧④创建Web控制类
  - ✧⑤创建注册页面
  - ✧⑥效果测试

①引入启动器依赖

- 在chapter05项目的pom.xml中引入Passay密码验证框架和hibernate-validator数据校验依赖，提供对密码和其他数据的校验支持。

引入启动器依赖

```
<!--引入Passay密码验证框架-->
<dependency>
    <groupId>org.passay</groupId>
    <artifactId>passay</artifactId>
    <version>1.6.0</version>
</dependency>
<!--引入数据校验依赖-->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.8.Final</version>
</dependency>
```

②自定义数据校验注解

- 为了实现对密码的校验，需要自定义以下两个数据校验注解：
  - ✧**@MyPasswordConstraint**：验证密码是否符合预设的长度、字符等规则要求。
  - ✧**@PasswordMatchConstraint**：验证密码和确认密码是否一致。
- 在**chapter05**项目中新建一个**com.itheima.validation**包，并在包中新建两个数据校验注解类**MyPasswordConstraint**和**PasswordMatchConstraint**及其业务逻辑类**MyPasswordValidator**和**PasswordMatchValidator**。

MyPasswordConstraint注解类

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD})                    // 限定使用范围——只能在字段上使用
@Retention(RetentionPolicy.RUNTIME)      // 表明注解的生命周期，它在代码运行时可以通过反射获取到注解
@Constraint(validatedBy = MyPasswordValidator.class)      // validatedBy属性指定该注解的校验逻辑
public @interface MyPasswordConstraint {
    String message() default "Invalid Password";                    // 定义错误提示
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

MyPasswordValidator逻辑类

```
import org.passay.*;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import java.util.Arrays;
public class MyPasswordValidator  implements ConstraintValidator<MyPasswordConstraint, String> {
    @Override
    public void initialize(MyPasswordConstraint myPasswordConstraint) {  //初始化校验消息，得到配置的注解内容
    }
    @Override
    public boolean isValid(String password, ConstraintValidatorContext validatorContext) {  //自定义校验逻辑
        PasswordValidator validator = new PasswordValidator(Arrays.asList(
                new LengthRule(8, 30)                                                    //密码长度为8到30位
        ));
        RuleResult result = validator.validate(new PasswordData(password));
        return result.isValid();
    }
}
```

PasswordMatchConstraint注解类

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

// 限定使用范围——可在字段、类、注解上使用
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)       // 表明注解的生命周期，它在代码运行时可以通过反射获取到注解
@Constraint(validatedBy = PasswordMatchValidator.class)       // validatedBy属性指定该注解的校验逻辑
public @interface PasswordMatchConstraint  {
    String message() default "Password Not Match";                    // 定义错误提示
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

PasswordMatchValidator逻辑类

```
import com.itheima.domain.User;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class PasswordMatchValidator implements ConstraintValidator<PasswordMatchConstraint, User> {
    @Override
    public void initialize(PasswordMatchConstraint passwordMatchConstraint) {
    }

    @Override
    public boolean isValid(User user, ConstraintValidatorContext constraintValidatorContext) {
        return user.getPassword().equals(user.getMatchingpwd());
    }
}
```

③创建持久化类

- 在chapter05项目com.itheima.domain包中，修改实体类User，为类及其属性添加校验注解。

User类

```
import com.itheima.validation.MyPasswordConstraint;
import com.itheima.validation.PasswordMatchConstraint;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
@PasswordMatchConstraint
public class User {
    private Integer id;
    @NotNull
    @NotBlank
    @Size(min = 4, max = 50, message = "用户名长度必须在4~50个字符之间")
    private String username;
    @NotNull
    @MyPasswordConstraint
    private String password;
    @NotNull
    private String matchingpwd;
    @Pattern(regexp="^1[356789]\\d{9}$", message = "手机号不合法")
    private String phone;
    //省略属性的getXX()和setXX()方法
}
```

④创建Web控制类

- 在chapter05项目的com.itheima.controller包中，新建一个Web控制类RegisterController，实现向前端模板页面及其公共片段动态数据传递。

RegisterController类

```
import com.itheima.domain.User;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import javax.validation.Valid;
import java.util.Calendar;
@Controller
public class RegisterController {
    //跳转到注册页register.html
    @GetMapping("/toRegisterPage")
    public String toRegisterPage(Model model){
        model.addAttribute("user",new User());
        return "register";
    }
```

RegisterController类

```
    @PostMapping("/checkUser")
    public String checkUser(@Valid @ModelAttribute User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "register";
        }
        return "results";
    }
    @ModelAttribute("currentYear")
    public int getCurrentYear(){
        return Calendar.getInstance().get(Calendar.YEAR);
    }
    @ModelAttribute("institute")
    public String getInstitute(){
        return "广东财经大学";
    }
}
```

启用数据校验

⑤创建注册页面

- 在chapter05项目resources的templates目录下，新建一个用户注册视图页面register.html，在页面中使用th:field、th:errors属性和#fields内置对象获取表单参数错误提示信息，并使用公共片段设计footer。

注册页面register.html

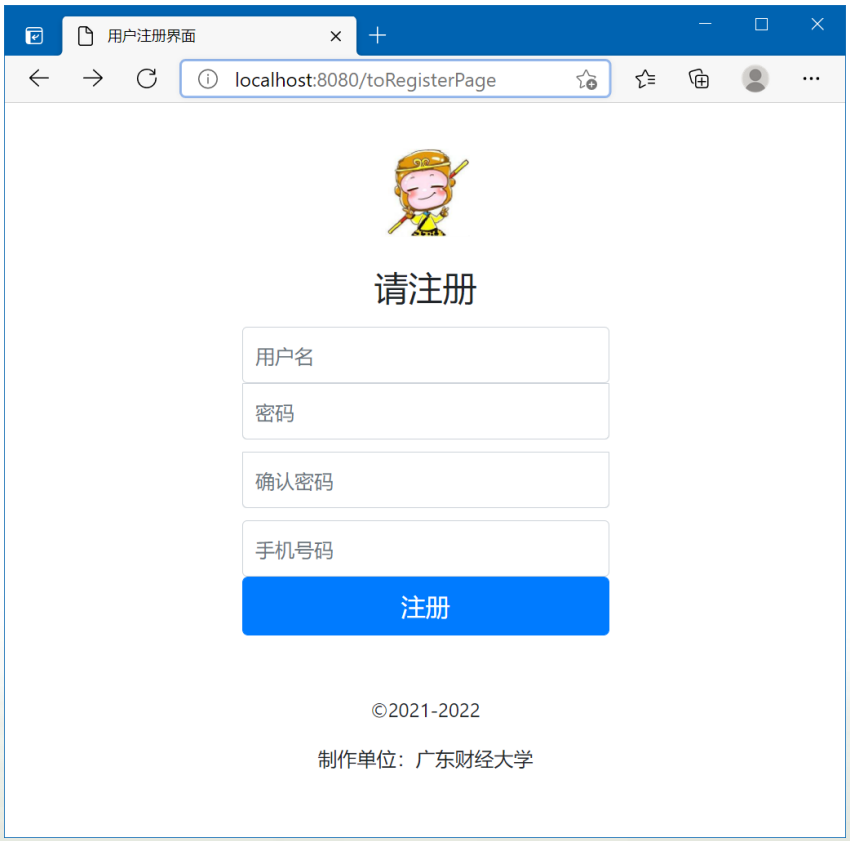
```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,shrink-to-fit=no">
    <title>用户注册界面</title>
    <link th:href="@{/login/css/bootstrap.min.css}" rel="stylesheet">
    <link th:href="@{/login/css/signin.css}" rel="stylesheet">
</head>
<body class="text-center">
<style type="text/css">
    .warn{color:red}
</style>
<!-- 用户注册form表单 -->
```

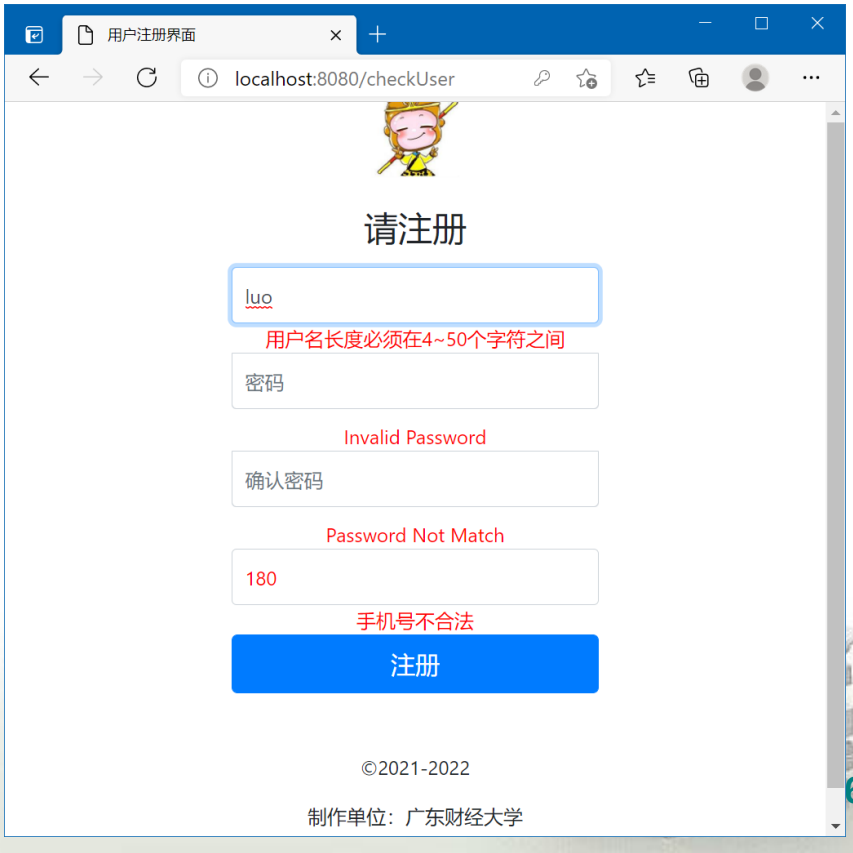
注册页面register.html

```
<form class="form-signin" th:action="@{/checkUser}" th:object="${user}" method="post">
    <img class="mb-4" th:src="@{/login/img/login.jpg}" width="72" height="72">
   <h1 class="h3 mb-3 font-weight-normal" th:text="请注册">欢迎注册</h1>
   <input type="text" th:field="*{username}" class="form-control" th:errorclass="warn“ th:placeholder="用户名"  required=""
autofocus="">
   <span  class="warn" th:if="${#fields.hasErrors('username')}" th:errors="*{username}">用户名错误</span>
   <input type="password"  th:field="*{password}" class="form-control" th:placeholder="密码"  required="">
   <span  class="warn" th:if="${#fields.hasErrors('password')}" th:errors="*{password}">密码错误</span>
   <input type="password" th:field="*{matchingpwd}" class="form-control" th:placeholder="确认密码"  required="">
   <span  class="warn" th:if="${#fields.hasErrors('${user}')}" th:errors="${user}">确认密码错误</span>
   <input type="text" th:field="*{phone}" class="form-control" th:errorclass="warn"  th:placeholder="手机号码"  required="">
   <span  class="warn" th:if="${#fields.hasErrors('phone')}" th:errors="*{phone}">手机号码错误</span>
   <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="注册">注册</button>
   <div th:replace="~{footer::footer(${currentYear},${institute})}"></div>
</form>
</body>
</html>
```

⑥效果测试

- 启动项目进行测试，在浏览器上访问 **http://localhost:8080/toRegisterPage**






聰明等于勤奋
勞力就是幸福