Spring MVC

概念

Spring web MVC 框架提供了模型-视图-控制的体系结构和可以用来开发灵活、松散耦合的 web 应用程序的组件。

• M（model-模型）：模型封装了应用程序数据，并且通常它们由 POJO 组成。 • V（view-视图）：视图主要用于呈现模型数据，并且通常它生成客户端的浏览器可以解释的 HTML 输出。 • C（controller-控制器）：控制器主要用于处理用户请求，并且构建合适的模型并将其传递到视图呈现。

特点

（1）Spring MVC拥有强大的灵活性、非入侵性和可配置性。

（2）Spring MVC 提供了一个前端控制器DispatcherServlet，开发者无须额外开发控制器对象。

（3）Spring MVC分工明确，包含控制器、验证器、命令对象、模型对象、处理程序映射视图解析器，等等，每一个功能实现由一个专门的对象负责完成。

（4）Spring MVC可以自动绑定用户输入，并正确地转换数据类型。例如：Spring MVC能自动解析字符串，并将其设置为模型的int或float类型的属性。

（5）Spring MVC使用一个名称/值的Map对象实现更加灵活的模型数据传输。

（6）Spring MVC内置了常见的校验器，可以校验用户输入，如果校验不通过，则重定向回输入表单。输入校验是可选的，并且支持编程方式及声明方式。

（7）Spring MVC支持国际化，支持根据用户区域显示多国语言，并且国际化的配置非常简单。

（8）Spring MVC支持多种视图技术，最常见的有JSP技术以及其他技术，包括Velocity和FreeMarker。

（9）Spring 提供了一个简单而强大的JSP标签库，支持数据绑定功能，使得编写JSP页面更加容易。

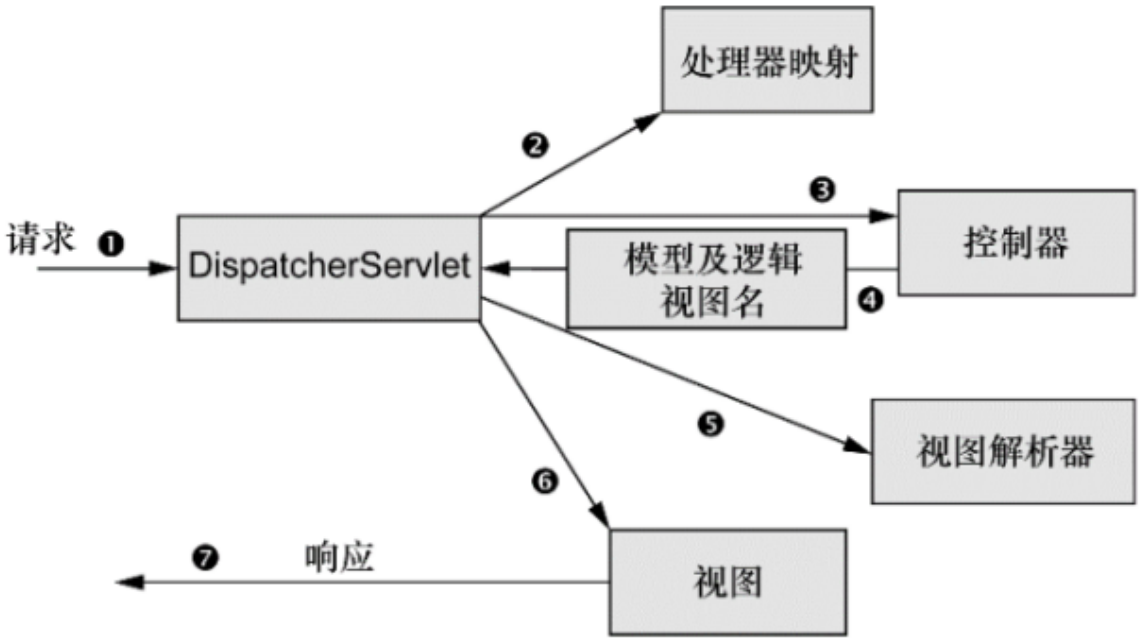
优点

1. 开发灵活
2. 松散耦合

核心

Spring MVC围绕 **DispatcherServlet（前置控制器）** 设计，DispatcherServlet封装了servlet，提供更便捷的api给开发者使用。

请求处理过程



(1) 用户发送请求给DispatcerServlet

(2) DispatcerServlet把请求委托给Handler Mapping处理，Handler Mapping根据请求信息（url、get、post等）找到匹配Controller（控制器）并返回给DispatcerServlet

(3) DispatcerServlet将请求委托给相应的Controller处理

(4) Controller调用业务处理逻辑后，返回Model（模型）和逻辑视图名给DispatcerServlet

(5) DispatcerServlet将逻辑视图名等信息委托给View Resoler（视图解析器），View Resoler处理后返回真实的视图地址给DispatcherServlet

(6) DispatcherServlet将Model委托给相应的View渲染

(7) View将结果响应给用户

SpringMVC和Struts2区别

1. Springmvc处理请求方式是基于方法设计，也就是说一个类有多个方法，多个方法可以处理多个请求，简化处理请求逻辑。
2. Servlet和Struts2处理请求方式是基于类设计，一个类只能处理一个请求，除非通过判断参数、地址等方式处理不同请求，处理请求逻辑繁琐。

映射

SpringMVC是通过@RequestMapping注解匹配请求地址，一旦匹配上就会调用该方法，它可以注解在方法、类上面。

@RequestMapping属性

属性	属性值类型	示例	描述
Value	String[]	Value={“/index”, “/home”}	匹配请求映射地址
Method	RequestMethod[]	Method={RequestMethod.GET}	匹配请求的方式：get、post、put、delete
Params	String[]	Params={“username”,”password != 123456”}	匹配请求参数
headers	String[]	Headers={“content-type=text/*”}	匹配请求头部参数
consumes	String[]	consumes = {"text/plain", "application/*"}	匹配请求媒体内容类型
produces	String[]	produces = "application/json; charset=UTF-8"	匹配请求响应的内容类型

@RequestMapping注解在类上代表所有方法上的地址都会加上该地址作为前缀

@RequestMapping模糊匹配

Ant 风格资源地址支持 3 种匹配符：

– ?：匹配文件名中的一个字符

– *：匹配文件名中的任意字符

– **：** 匹配多层路径

@RequestMapping 还支持 Ant 风格的 URL：

/user/*/createUser: 匹配/user/aaa/createUser、/user/bbb/createUser 等 URL

/user/**/createUser: 匹配/user/createUser、/user/aaa/bbb/createUser 等 URL

/user/createUser??: 匹配/user/createUseraa、/user/createUserbb 等 URL

接收参数

1. 单个参数
    - 自动匹配

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc
//自动匹配要求参数名和前端传过来的参数名一样
@RequestMapping("/getParam")
public String getParam(String name){

}
```

    - @RequestParam

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc
//@RequstParam允许参数名和前端传过来的参数名不一样
@RequestMapping("/getParam")
public String getParam(@RequstParam("name") String username){

}
```

- HttpServletRequest

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc
@RequestMapping("/getParam")
public String getParam(HttpServletRequest request){
    String name = request.getParameter("name");
}
```

- @CookieValue

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc
//假设cookie有username=asd
@RequestMapping("/getParam")
public String getParam(@CookieValue("username") String username){

}
```

- @RequestHeader

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc
@RequestMapping("/getParam")
public String getParam(@RequestHeader("Accept-Encoding") String encoding){

}
```

- @SessionAttributes：将返回request域的某个属性和值保存到session中
- @ModelAttribute：通过@ModelAttribute获取session参数

2. 多个参数

- 通过实体对象，要求对象的属性名和传过来的参数名一样

```
//请求地址：http://localhost:8080/springmvc/getParam?name=abc&age=20&sex=1
//要求对象的属性名和传过来的参数名一样
@RequestMapping("/getParam")
public String getParam(User user){
    String name = request.getParameter("name");
}
```

3. 路径参数

- @PathVariable

```
//请求地址：http://localhost:8080/springmvc/getParam/1
@RequestMapping("/getParam/{id}")
public String getParam(@PathVariable("id") int id){

}
```

响应模型和视图

(1) Map

(2) Model

① Mode.addAttribute(Object, Object);

(3) ModelAndView

① 创建ModelAndView对象

② 设置view的名字

③ 绑定Model数据

(4) 特殊格式

① Json

```
//添加jackson的依赖
<!-- jackson -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>${jackson.version}</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson.version}</version>
</dependency>

//添加默认视图
<!-- 指定默认视图解析器 -->
        <property name="defaultViews">
            <list>
                <!-- json视图解析器 -->
                <bean
                    class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" />
            </list>
        </property>

//编写Controller
    /**
     *
```

```
     * 返回json数据给前端
     *
     * @ResponseBody：将对象通过某种视图输出到响应内容里面
     *
     * @version 2018年3月3日上午10:22:23
     * @author zhuwenbin
     * @param user
     * @return
     */
    @RequestMapping("model/returnDataByJson")
    @ResponseBody
    public User returnDataByJson(User user) {
        System.out.println(user);
        return user;
    }
```

② xml

```
//添加oxm的依赖
<!-- jackson -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>${spring.version}</version>
</dependency>

//添加默认视图
<!-- 指定默认视图解析器 -->
        <property name="defaultViews">
            <list>
                <!-- xml视图解析器，需要增加相应的依赖spring-oxm，并且需要返回的xml对象需要在下面的扫描中添加 -->
                <bean class="org.springframework.web.servlet.view.xml.MarshallingView">
                    <constructor-arg>
                        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
                            <property name="packagesToScan">
                                <list>
                                    <value>com.qhcs.springmvc.entity</value>
                                </list>
                            </property>
                        </bean>
                    </constructor-arg>
                </bean>
            </list>
        </property>

//在实体上加上注解@XmlRootElement
@XmlRootElement
public class User {
}

//编写Controller
```

```java
    /**
     *
     * 返回json数据给前端
     *
     * @ResponseBody：将对象通过某种视图输出到响应内容里面
     *
     * @version 2018年3月3日上午10:22:23
     * @author zhuwenbin
     * @param user
     * @return
     */
    @RequestMapping("model/returnDataByJson")
    @ResponseBody
    public User returnDataByJson(User user) {
        System.out.println(user);
        return user;
    }
```

注意：通过后缀切换json或者xml视图，比如model/returnDataByJson.xml切换到xml视图

上传和下载

上传

(1) 加入common-fileupload依赖

```xml
<!-- fileupload -->

<dependency>

    <groupId>commons-fileupload</groupId>

    <artifactId>commons-fileupload</artifactId>

    <version>${fileupload.version}</version>

</dependency>
```

(2) 配置文件上传解析器

```
<!-- 文件上传解析器 -->
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置默认字符编码 -->
    <property name="defaultEncoding" value="utf-8"></property>
    <!-- 设置最大上传文件大小（byte单位） -->
    <property name="maxUploadSize" value="3000000"/>
</bean>
```

(3) 将上传表单的enctype="multipart/form-data"

```
<!-- enctype指定表单编码类型 -->
<form action="${ctx }/upload/doUpload" method="post" enctype="multipart/form-data">
    <table>
        <tr>
            <td>用户名</td>
            <td>
                <input type="text" name="username">
            </td>
        </tr>
        <tr>
            <td>密码</td>
            <td>
                <input type="password" name="password">
            </td>
        </tr>
        <tr>
            <td>头像</td>
            <td>
                <input type="file" name="file">
            </td>
        </tr>
        <tr>
            <td colspan="2">
                ${msg }
            </td>
        </tr>
        <tr>
            <td>
                <input type="reset" value="重置">
            </td>
            <td>
                <input type="submit" value="提交">
            </td>
        </tr>
    </table>
</form>
```

(4) 通过MultipartFile来获取表单file类型数据项

```
@PostMapping("/upload/doUpload")
    public String doUpload(User user, @RequestParam("file") MultipartFile multipartFile, Model
model) {
        System.out.println(user);
        if (Objects.nonNull(multipartFile)) {
            // 获取该表单项参数名字
            String name = multipartFile.getName();
            // 获取文件名
            String fileName = multipartFile.getOriginalFilename();
            // 获取内容类型
            String contentType = multipartFile.getContentType();
            // 获取文件大小
            long size = multipartFile.getSize();
            System.out.println(
                    "name=" + name + ", fileName=" + fileName + ", contentType=" + contentType +
", size=" + size);
            //保存上传文件
            try {
                multipartFile.transferTo(new File("E:\\工作\\测试\\上传\\" + fileName));
            } catch (IllegalStateException | IOException e) {
                e.printStackTrace();
                model.addAttribute("msg", "文件: " + fileName + "上传失败! ");
            }

            // 提示用户
            model.addAttribute("msg", "文件: " + fileName + "上传成功! ");
        }
        return "upload";
    }
```

(5) 保存

下载

```
/**
     *
     * 下载
     *
     * @version 2018年3月3日下午4:51:32
     * @author zhuwenbin
     * @param fileName
     * @return
     * @throws RuntimeException
     * @throws IOException
     */
    @PostMapping(value = "/download/doDownload", produces =
MediaType.APPLICATION_OCTET_STREAM_VALUE)
    public ResponseEntity<byte[]> doDownload(String fileName) throws RuntimeException,
IOException {
        System.out.println(fileName);
        // 文件保存路径
        String savePath = "E:\\工作\\测试\\上传";
```

```
        // 判断文件是否存在
        File file = new File(savePath, fileName);
        if (!file.exists()) {
            throw new RuntimeException("文件不存在! ");
        }
        if (file.isDirectory()) {
            throw new RuntimeException("请输入正确的文件名! ");
        }
        // 读取文件
        byte[] bs = FileUtils.readFileToByteArray(file);

        // 设置响应头部
        HttpHeaders headers = new HttpHeaders();
        // 设置响应内容类型
        headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
        // 设置下载的文件名
        headers.setContentDispositionFormData("attachment", new String(("下载文件-" +
fileName).getBytes(), "iso-8859-1"));

        return new ResponseEntity<byte[]>(bs, headers, HttpStatus.CREATED);
    }
```

Restful表单

(1) 在web.xml配置过滤器

```
<!-- 将post转换为put、delete请求 -->

<filter>

    <filter-name>HiddenHttpMethodFilter</filter-name>

    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>

</filter>

<filter-mapping>

    <filter-name>HiddenHttpMethodFilter</filter-name>

    <url-pattern>/*</url-pattern>

</filter-mapping>
```

(2) 在表单上加入隐藏域

```
<form action="${ctx }/restful/user/1" method="post">
    <!-- 隐藏参数，name固定为_method，value制定具体的请求方式-->
    <input type="hidden" name="_method" value="delete">
    <input type="submit" value="delete">
</form>
<br>
<form action="${ctx }/restful/user/1" method="post">
    <!-- 隐藏参数，name固定为_method，value制定具体的请求方式-->
    <input type="hidden" name="_method" value="put">
    <input type="submit" value="put">
</form>
```

(3) 在controlelr类上通过设置method来获取相应的表单类型

```
    /**

     *

     * delete：删除某个资源

     *

     * @version 2018年3月3日下午6:17:15

     * @author zhuwenbin

     * @param user

     * @return

     */

    @DeleteMapping(value = "restful/user/{id}")
    public String delete(@PathVariable("id") String id, Model model) {

        model.addAttribute("msg", "删除用户成功！id：" + id);

        return "restful";

    }



    /**

     *

     * put：修改某个资源

     *
```

```
     * @version 2018年3月3日下午6:17:15

     * @author zhuwenbin

     * @param user

     * @return

     */

    @PutMapping(value = "restful/user/{id}")
    public String put(@PathVariable("id") String id, Model model) {

        model.addAttribute("msg", "修改用户成功! id: " + id);

        return "restful";

    }
```

Ajax

(1) 加入jackson依赖

```
<!-- jackson -->

<dependency>

    <groupId>com.fasterxml.jackson.core</groupId>

    <artifactId>jackson-core</artifactId>

    <version>${jackson.version}</version>

</dependency>

<dependency>

    <groupId>com.fasterxml.jackson.core</groupId>

    <artifactId>jackson-databind</artifactId>

    <version>${jackson.version}</version>

</dependency>
```

(2) 配置视图解析器

```
    <!-- 内容转发视图解析器，根据内容不同转发到不同的视图解析器 -->
    <bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
        <!-- 视图解析器列表 -->
        <property name="viewResolvers">
            <list>
                <!-- jsp视图解析器 -->
                <bean id="jspViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                    <!-- 支持jstl -->
                    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
                    <!-- 比如逻辑视图名：index，添加前缀变成/WEB-INF/jsp/index，再添加后缀变成/WEB-
INF/jsp/index.jsp -->
                    <!-- 前缀，表示在逻辑视图名前加上路径 -->
                    <property name="prefix" value="/WEB-INF/jsp/"/>
                    <!-- 后缀，表示在逻辑视图名后加上后缀名 -->
                    <property name="suffix" value=".jsp"/>
                    <!-- 指定解析器的优先顺序 -->
                    <property name="order" value="1"/>
                </bean>
                <!-- beanNameResolver -->
                <bean id="beanNameViewResolver"
class="org.springframework.web.servlet.view.BeanNameViewResolver">
                    <!-- 指定解析器的优先顺序 -->
                    <property name="order" value="2"/>
                </bean>
            </list>
        </property>
        <!-- 指定默认视图解析器 -->
        <property name="defaultViews">
            <list>
                <!-- json视图解析器 -->
                <bean
class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
            </list>
        </property>
    </bean>
```

(3) 在controller方法上加入@ResponseBody注解

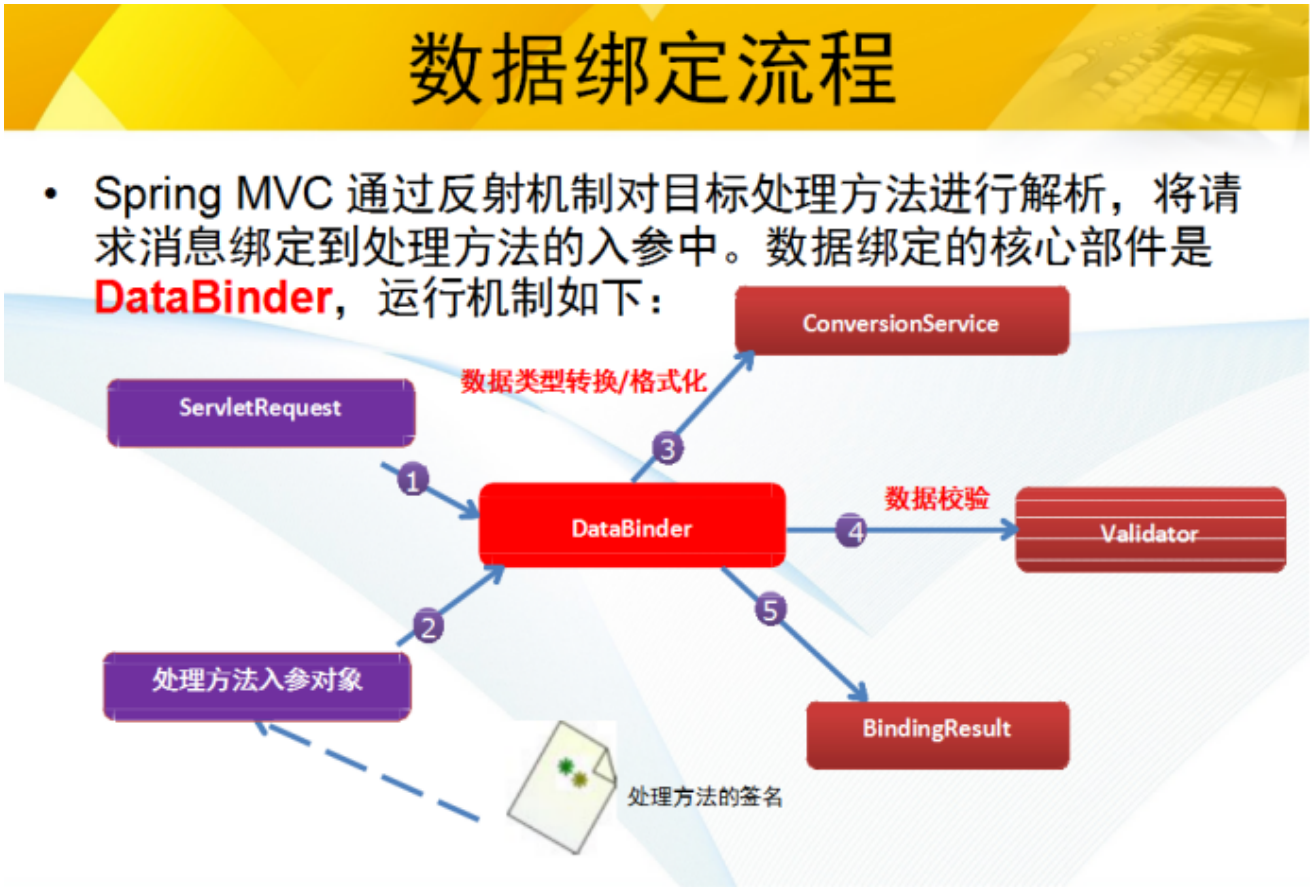
```
    /**
     *
     * 获取用户信息，返回json格式数据
     * @version 2018年3月3日下午12:02:34
     * @author zhuwenbin
     * @param id
     * @return
     */
    @GetMapping("/ajax/user/{id}")
    @ResponseBody
    public User getUser2(@PathVariable("id") String id) {
        return new User(Integer.parseInt(id), "Kobe", "123456");
    }
```

(4) 通过jquery的ajax操作

```
//json
function getUser(id){
    $.get(
        "${ctx}/ajax/user",
        {
            "id":id
        },
        function(data){
            $("#content").text("id=" + data.id + ", username=" + data.username + ", password=" +
data.password);
        }
    );
}

//xml
function getUser(id){
    $.get(
        "${ctx}/ajax/user.xml",
        {
            "id":id
        },
        function(data){
            $("#content").text("id=" + $(data).find("id").text()
                               + ", username=" + $(data).find("username").text()
                               + ", password=" + $(data).find("password").text());
        }
    );
}
```

验证



1. 数据校验

(1) 永远不要相信接收的数据是正确的

(2) 脏数据比程序错误更严重

(3) 数据校验：

① 前端校验，规范用户输入并友好提示

② 后端校验，前端校验只能校验普通的输入，后端校验才是真正校验

后端验证

(1) 加入Hibernate-validation的依赖

```
<!-- hibernate-validator实现jsr303，实现bean校验 -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.7.Final</version>
</dependency>
```

(2) 在需要校验的实体的属性上加入注解类似于 @NotNull、@Max 等标准的注解指定校验规则

```
// 用户名
@NotBlank(message = "请输入用户名")
@Size(max = 20, min = 6, message = "请输入6-20为用户名")
private String username;


// 密码
@NotBlank(message = "请输入密码")
@Pattern(regexp = "^\d{6}$", message = "请输入6位数字密码")
private String password;
```

(3) 在处理请求的方法的实体参数前加上@Valid注解

(4) 在验证的实体参数后面（紧跟者实体参数）声明BindingResult对象，通过对象得到错误信息

(5) 根据错误信息响应

```
@RequestMapping("/user")
public String login(@Valid User user, BindingResult bindingResult, Model model) {
    // 如果存在错误，获取错误信息
    if (bindingResult.hasErrors()) {
        FieldError fieldError = bindingResult.getFieldError();
        String errorMsg = fieldError.getDefaultMessage();
```

```
        model.addAttribute("errorMsg", errorMsg);
        return "forward:/";
    }

    model.addAttribute("user", user);

    return "home";
}
```

校验注解种类

限制	说明
@Null	限制只能为null
@NotNull	限制必须不为null
@AssertFalse	限制必须为false
@AssertTrue	限制必须为true
@DecimalMax(value)	限制必须为一个不大于指定值的数字
@DecimalMin(value)	限制必须为一个不小于指定值的数字
@Digits(integer,fraction)	限制必须为一个小数，且整数部分的位数不能超过integer，小数部分的位数不能超过fraction
@Future	限制必须是一个将来的日期
@Max(value)	限制必须为一个不大于指定值的数字
@Min(value)	限制必须为一个不小于指定值的数字
@Past	限制必须是一个过去的日期
@Pattern(value)	限制必须符合指定的正则表达式
@Size(max,min)	限制字符长度必须在min到max之间
@Past	验证注解的元素值（日期类型）比当前时间早
@NotEmpty	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）
@NotBlank	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的空格
@Email	验证注解的元素值是Email，也可以通过正则表达式和flag指定自定义的email格式

异常

局部异常-》注解形式的全局异常-》配置形式的全局异常

(1) 在web.xml配置响应异常页面

```
<!-- 异常页面 -->

<error-page>

    <error-code>500</error-code>

    <location>/WEB-INF/jsp/500.jsp</location>
```

```
</error-page>


<error-page>
    <error-code>404</error-code>
    <location>/WEB-INF/jsp/404.jsp</location>
</error-page>
```

(2) 通过在controller添加异常处理方法，处理本类发生的异常，这是局部异常处理。

```
    /**
     *
     * @ExceptionHandler注解的方法是异常处理方法
     *
     * @version 2018年3月5日下午6:13:29
     * @author zhuwenbin
     * @param exception
     * @param model
     * @return
     */
    @ExceptionHandler
    public String exceptionHandler(Exception exception, Model model) {
        System.out.println("exceptionHandler: " + exception.getMessage());
        model.addAttribute("exception", exception);
        return "exception";
    }


    /**
```

```
     *
     * @ExceptionHandler(ArithmeticException.class)指定处理ArithmeticException异常的方法
     *
     * @version 2018年3月5日下午6:13:29
     * @author zhuwenbin
     * @param exception
     * @param model
     * @return
     */
    @ExceptionHandler(ArithmeticException.class)
    public String exceptionHandler2(Exception exception, Model model) {
        System.out.println("exceptionHandler2: " + exception.getMessage());
        model.addAttribute("exception", exception);
        return "exception";
    }
```

(3) 全局处理异常

① 通过@ControllerAdvice注解统一处理异常

```
@ControllerAdvice
public class CommonExceptionHandler {



    /**
     * 全局异常处理
     *
     * @version 2018年3月5日下午6:26:07
     * @author zhuwenbin
     * @param exception
     * @param model
```

```
     * @return
     */
    @ExceptionHandler
    public String exceptionHandler(Exception exception, Model model) {
        System.out.println("CommonExceptionHandler: " + exception.getMessage());
        model.addAttribute("exception", exception);
        return "500";
    }
}
```

② 通过配置文件的方式

```
    <!-- 全局异常处理解析器 -->
    <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
            <props>
                <!-- key指定异常的全路径，500的视图名称 -->
                <prop key="java.lang.ArithmeticException">404</prop>
                <prop key="java.lang.NullPointerException">exception</prop>
            </props>
        </property>
    </bean>
```

拦截器

> 基于动态代理实现，类似环绕通知

(1) 编写拦截器处理类，实现HandlerInterceptor接口

```
public class LoginInterceptor implements HandlerInterceptor {


    /**
     * 返回true继续执行后续请求操作，false则中断执行请求
     */
```

```
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler)
            throws Exception {
        System.out.println(">>>>>正在执行preHandle方法");
        HttpSession session = request.getSession();
        Object object = session.getAttribute("user");
        // 用户未登陆让它继续登陆
        if (Objects.isNull(object)) {
            System.out.println("用户未登陆，跳到登陆页面");
            // 重定向到登陆页面
            response.sendRedirect(request.getServletContext().getContextPath());
            // 中断执行
            return false;
        }

        // 如果用户已经登陆
        return true;
    }
}
```

(2) 在配置文件配置拦截器

```
    <!-- 拦截器，可以配置多个拦截器 -->
    <mvc:interceptors>
        <mvc:interceptor>
            <!-- 指定拦截的路径 -->
            <mvc:mapping path="/account/**"/>
            <!-- 指定拦截处理类 -->
            <bean class="com.qhcs.springmvc.interceptor.LoginInterceptor"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
```

拦截器的使用场景

处理所有请求共性问题：

1、乱码问题：用request，response参数去设置编码；

2、解决权限验证问题（是否登陆，取session对象查看）；

拦截器与过滤器的区别

1、拦截器Interceptor依赖于框架容器，基于反射机制，只过滤请求；

2、过滤器Filter依赖于Servlet容器，基于回调函数，过滤范围大；

拦截器与AOP的区别

1、拦截器Interceptor依赖于框架容器，基于反射机制，只过滤请求；

2、AOP依赖于框架容器，基于反射机制，拦截Spring管理Bean的访问；