

# Day 11

---

## 异步调用

发送异步请求

接受异步请求参数

异步请求接受响应数据

## 异步请求-跨域访问

跨域访问介绍

跨域注解

## 拦截器

作用

拦截器和过滤器对比

自定义拦截器开发过程

配置拦截器

拦截器配置与方法参数

前置处理方法

后置处理方法

完成处理方法

拦截器配置项

多拦截器的情况

异常处理

## 异常处理器

异常分类处理

注解开发异常处理器

使用注解实现异常分类管理

## 异常处理解决方案

## 自定义异常

异常定义格式

异常触发方式

## 实用技术

文件上传下载

上传文件过程分析

文件上传下载实现

页面表单

SpringMVC配置

控制器

文件上传注意事项

Restful风格配置

Rest

Rest 行为约定方式

Restful开发入门

校验框架

表单校验

表单校验分类

表单校验规则

表单校验框架

快速使用

开启校验

设置校验规则

获取错误信息

多规则校验

嵌套校验

分组校验

SSM 整合

整合步骤

项目结构搭建

## 异步调用

### 发送异步请求

```

1 <a href="javascript:void(0);" id="testAjax">访问controller</a>
2 <script type="text/javascript" src="/js/jquery-3.6.0.min.js"></script>
3 <script type="text/javascript">
4     $(function(){
5         $("#testAjax").click(function(){ //为id="testAjax"的组件绑定点击事件
6             $.ajax({ //发送异步调用
7                 type:"POST", //请求方式: POST请求
8                 url:"ajaxController", //请求参数 (也就是请求内容)
9                 data:'ajax message', //请求参数 (也就是请求内容)
10                dataType:"text", //响应正文类型
11                contentType:"application/text", //请求正文的MIME类型
12            });
13        });
14    });
15 </script>

```

## 接受异步请求参数

@RequestBody 将异步提交数据组织成标准请求参数格式，并赋值给形参

```

1 @RequestMapping("/ajaxController")
2 public String ajaxController(@RequestBody String message){
3     System.out.println(message);
4     return "page.jsp";
5 }

```

- 注解添加到 POJO 参数前方时，封装的异步提交数据按照 POJO 的属性格式进行关系映射
- 注解添加到集合参数前方时，封装的异步提交数据按照集合的存储结构进行关系映射

```
1 @RequestMapping("/ajaxPojoToController")
2 // 如果处理参数是 POJO，且页面发送的请求数据格式与 POJO 中的属性对应，@RequestBody 注解
  可以自动映射对应请求数据到POJO中
3 // 注意：POJO中的属性如果请求数据中没有，属性值为null，POJO中没有的属性如果请求数据中有，
  不进行映射
4 public String ajaxPojoToController(@RequestBody User user){
5     System.out.println("controller pojo :"+user);
6     return "page.jsp";
7 }
8
9 @RequestMapping("/ajaxListToController")
10 //如果处理参数是List集合且封装了POJO，且页面发送的数据是JSON格式的对象数组，数据将自动映射
   到集合参数中
11 public String ajaxListToController(@RequestBody List<User> userList){
12     System.out.println("controller list :"+userList);
13     return "page.jsp";
14 }
```

## 异步请求接受响应数据

方法返回值为Pojo时，自动封装数据成json对象数据

```
1 @RequestMapping("/ajaxReturnJson")
2 @ResponseBody
3 public User ajaxReturnJson(){
4     System.out.println("controller return json pojo...");
5     User user = new User();
6     user.setName("Jockme");
7     user.setAge(40);
8     return user;
9 }
```

方法返回值为 List 时，自动封装数据成 json 对象数组数据

```
1 @RequestMapping("/ajaxReturnJsonList")
2 @ResponseBody
3 // 基于 jackson 技术, 使用 @ResponseBody 注解可以将返回的保存 POJO 对象的集合转成json数
  组格式数据
4 public List ajaxReturnJsonList(){
5     System.out.println("controller return json list...");
6     User user1 = new User();
7     user1.setName("Tom");
8     user1.setAge(3);
9
10    User user2 = new User();
11    user2.setName("Jerry");
12    user2.setAge(5);
13
14    ArrayList al = new ArrayList();
15    al.add(user1);
16    al.add(user2);
17
18    return al;
19 }
```

## 异步请求-跨域访问

### 跨域访问介绍

当通过域名A下的操作访问域名B下的资源时, 称为跨域访问。

跨域访问时, 会出现无法访问的现象。

### 跨域注解

@CrossOrigin设置当前处理器方法/处理器类中所有方法支持跨域访问

```
1 @RequestMapping("/cross")
2 @ResponseBody
3 //使用@CrossOrigin开启跨域访问
4 //标注在处理器方法上方表示该方法支持跨域访问
5 //标注在处理器类上方表示该处理器类中的所有处理器方法均支持跨域访问
6 @CrossOrigin
7 public User cross(HttpServletRequest request){
8     System.out.println("controller cross..." + request.getRequestURL());
9     User user = new User();
10    user.setName("Jockme");
11    user.setAge(39);
12    return user;
13 }
```

## 拦截器

拦截器（Interceptor）是一种动态拦截方法调用的机制

## 作用

作用主要有两个：

1. 在指定的方法调用前后执行预先设定后的代码
2. 阻止原始方法的执行

核心原理还是 AOP 思想

拦截器链：多个拦截器按照一定的顺序，对原始被调用功能进行增强

## 拦截器和过滤器对比

归属不同：Filter 属于 Servlet 技术，Interceptor 属于 SpringMVC 技术

拦截内容不同：Filter 对所有访问进行增强，Interceptor 仅针对 SpringMVC 的访问进行增强

## 自定义拦截器开发过程

实现HandlerInterceptor接口

```

1 // 自定义拦截器需要实现HandlerInterceptor接口
2 public class MyInterceptor implements HandlerInterceptor {
3     //处理器运行之前执行
4     @Override
5     public boolean preHandle(HttpServletRequest request,
6                             HttpServletResponse response,
7                             Object handler) throws Exception {
8         System.out.println("前置运行-----a1");
9         //返回值为false将拦截原始处理器的运行
10        //如果配置多拦截器，返回值为false将终止当前拦截器后面配置的拦截器的运行
11        return true;
12    }
13
14    //处理器运行之后执行
15    @Override
16    public void postHandle(HttpServletRequest request,
17                          HttpServletResponse response,
18                          Object handler,
19                          ModelAndView modelAndView) throws Exception {
20        System.out.println("后置运行-----b1");
21    }
22
23    //所有拦截器的后置执行全部结束后，执行该操作
24    @Override
25    public void afterCompletion(HttpServletRequest request,
26                              HttpServletResponse response,
27                              Object handler,
28                              Exception ex) throws Exception {
29        System.out.println("完成运行-----c1");
30    }
31
32    //三个方法的运行顺序为    preHandle -> postHandle -> afterCompletion
33    //如果preHandle返回值为false，三个方法仅运行preHandle
34 }

```

## 配置拦截器

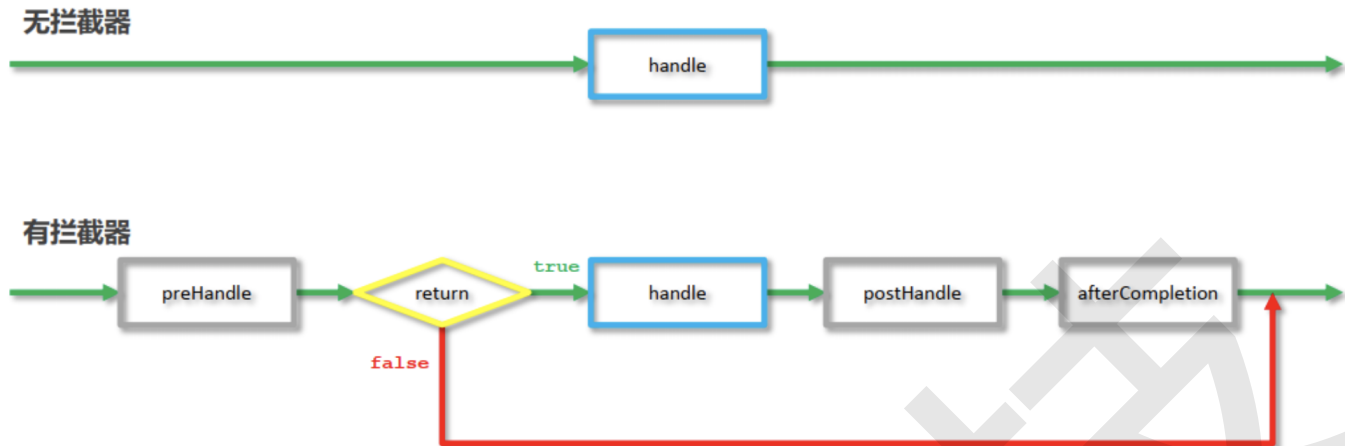
配置拦截器

```

1 <mvc:interceptors>
2     <mvc:interceptor>
3         <mvc:mapping path="/showPage"/>
4         <bean class="com.rushuni.interceptor.MyInterceptor"/>
5     </mvc:interceptor>
6 </mvc:interceptors>

```

注意：配置顺序为先配置执行位置，后配置执行类



## 拦截器配置与方法参数

### 前置处理方法

原始方法之前运行

```
1 public boolean preHandle(HttpServletRequest request,
2                           HttpServletResponse response,
3                           Object handler) throws Exception {
4     System.out.println("preHandle");
5     return true;
6 }
```

Java

复制代码

参数说明：

- request:请求对象
- response:响应对象
- handler:被调用的处理器对象，本质上是一个方法对象，对反射中的Method对象进行了再包装

返回值

- 返回值为 `false`，被拦截的处理器将不执行

### 后置处理方法

原始方法运行后运行，如果原始方法被拦截，则不执行



```
1 public void postHandle(HttpServletRequest request,
2                        HttpServletResponse response,
3                        Object handler,
4                        ModelAndView modelAndView) throws Exception {
5     System.out.println("postHandle");
6 }
```

参数说明：

modelAndView:如果处理器执行完成具有返回结果，可以读取到对应数据与页面信息，并进行调整。

## 完成处理方法

拦截器最后执行的方法，无论原始方法是否执行

```
1 public void afterCompletion(HttpServletRequest request,
2                            HttpServletResponse response,
3                            Object handler,
4                            Exception ex) throws Exception {
5     System.out.println("afterCompletion");
6 }
```

## 拦截器配置项

```

1  <mvc:interceptors>
2      <!--开启具体的拦截器的使用，可以配置多个-->
3      <mvc:interceptor>
4          <!--设置拦截器的拦截路径，支持*通配-->
5          <!--/**          表示拦截所有映射-->
6          <!--/*          表示拦截所有/开头的映射-->
7          <!--/user/*      表示拦截所有/user/开头的映射-->
8          <!--/user/add*   表示拦截所有/user/开头，且具体映射名称以add开头的映射-->
9          <!--/user/*All   表示拦截所有/user/开头，且具体映射名称以All结尾的映射-->
10         <mvc:mapping path="/*"/>
11         <mvc:mapping path="/*"/*"/>
12         <mvc:mapping path="/handleRun*"/*"/>
13         <!--设置拦截排除的路径，配置/**或/*，达到快速配置的目的-->
14         <mvc:exclude-mapping path="/b*"/*"/>
15         <!--指定具体的拦截器类-->
16         <bean class="com.rushuni.MyInterceptor"/>
17     </mvc:interceptor>
18 </mvc:interceptors>

```

## 多拦截器的情况

- 当有多个拦截器时，会形成拦截器链。
- 拦截器的运行顺序参照配置的先后顺序
- 当拦截器中出现对原始处理器的拦截，后面的拦截器终止运行
- 当拦截器运行中断，只运行配置在前面的拦截器的 afterCompletion 操作。

## 异常处理

### 异常处理器

Spring MVC 提供了 HandlerExceptionResolver 接口作为异常处理器。

```

1  @Component
2  public class ExceptionResolver implements HandlerExceptionResolver {
3      public ModelAndView resolveException(HttpServletRequest request,
4                                          HttpServletResponse response,
5                                          Object handler,
6                                          Exception ex) {
7          System.out.println("异常处理器正在执行中");
8          ModelAndView modelAndView = new ModelAndView();
9          //定义异常现象出现后, 反馈给用户查看的信息
10         modelAndView.addObject("msg", "出错啦! ");
11         //定义异常现象出现后, 反馈给用户查看的页面
12         modelAndView.setViewName("error.jsp");
13         return modelAndView;
14     }
15 }

```

## 异常分类处理

根据异常的种类不同, 进行分门别类的管理, 返回不同的信息

```

1  public class ExceptionResolver implements HandlerExceptionResolver {
2      @Override
3      public ModelAndView resolveException(HttpServletRequest request,
4                                          HttpServletResponse response,
5                                          Object handler,
6                                          Exception ex) {
7          System.out.println("my exception is running ...."+ex);
8          ModelAndView modelAndView = new ModelAndView();
9          if( ex instanceof NullPointerException){
10             modelAndView.addObject("msg", "空指针异常");
11         }else if ( ex instanceof ArithmeticException){
12             modelAndView.addObject("msg", "算术运算异常");
13         }else{
14             modelAndView.addObject("msg", "未知的异常");
15         }
16         modelAndView.setViewName("error.jsp");
17         return modelAndView;
18     }
19 }

```

## 注解开发异常处理器

通过 @ControllerAdvice 设置当前类为异常处理器类, 同时可以实现异常分类管理。

```
1 @Component
2 @ControllerAdvice
3 public class ExceptionAdvice {
4 }
```

## 使用注解实现异常分类管理

@ExceptionHandler 注解，可以让处理器方法可以设定多个异常的处理方式。

```
1 @ExceptionHandler(Exception.class)
2 @ResponseBody
3 public String doOtherException(Exception ex){
4     return "出错啦，请联系管理员! ";
5 }
```

## 异常处理解决方案

异常处理方案

- 业务异常
  - 发送对应消息传递给用户，提醒规范操作
- 系统异常
  - 发送固定消息传递给用户，安抚用户
  - 发送特定消息给运维人员，提醒维护
  - 记录日志
- 其他异常
  - 发送固定消息传递给用户，安抚用户
  - 发送特定消息给编程人员，提醒维护
  - 纳入预期范围内
  - 记录日志

## 自定义异常

### 异常定义格式

```

1 //自定义异常继承RuntimeException, 覆盖父类所有的构造方法
2 public class BusinessException extends RuntimeException {
3     public BusinessException() {
4     }
5
6     public BusinessException(String message) {
7         super(message);
8     }
9
10    public BusinessException(String message, Throwable cause) {
11        super(message, cause);
12    }
13
14    public BusinessException(Throwable cause) {
15        super(cause);
16    }
17
18    public BusinessException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
19        super(message, cause, enableSuppression, writableStackTrace);
20    }
21 }

```

## 异常触发方式

```

1 if(user.getName().trim().length()<4) {
2     throw new BusinessException("用户名长度必须在2-4位之间, 请重新输入! ");
3 }

```

通过自定义异常将所有的异常现象进行分类管理, 以统一的格式对外呈现异常消息

## 实用技术

## 文件上传下载

### 上传文件过程分析

- MultipartResolver 接口
  - MultipartResolver 接口定义了文件上传过程中的相关操作, 并对通用性操作进行了封装。
  - MultipartResolver 接口底层实现类 CommonsMultipartResovler。

- CommonsMultipartResovler并未自主实现文件上传下载对应的功能，而是调用了apache的文件上传下载组件。

XML | 复制代码

```
1 <dependency>
2   <groupId>commons-fileupload</groupId>
3   <artifactId>commons-fileupload</artifactId>
4   <version>1.4</version>
5 </dependency>
```

## 文件上传下载实现

### 页面表单

HTML | 复制代码

```
1 <form action="/fileupload" method="post" enctype="multipart/form-data">
2   上传: <input type="file" name="file"/><br/>
3   <input type="submit" value="上传"/>
4 </form>
```

### SpringMVC配置

HTML | 复制代码

```
1 <bean id="multipartResolver"
2
3   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4 </bean>
```

### 控制器

Java | 复制代码

```
1 @RequestMapping(value = "/fileupload")
2 public void fileupload(MultipartFile file){
3     file.transferTo(new File("file.png"));
4 }
```

## 文件上传注意事项

1. 文件命名问题， 获取上传文件名，并解析文件名与扩展名
2. 文件名过长问题
3. 文件保存路径
4. 重名问题

```

1  @RequestMapping(value = "/fileupload")
2  // 参数中定义MultipartFile参数, 用于接收页面提交的type=file类型的表单, 要求表单名称与参
   数名相同
3  public String fileupload(MultipartFile file,MultipartFile file1,MultipartFile
   file2, HttpServletRequest request) throws IOException {
4      System.out.println("file upload is running ..." + file);
5      //      MultipartFile参数中封装了上传的文件的相关信息
6      //      System.out.println(file.getSize());
7      //      System.out.println(file.getBytes().length);
8      //      System.out.println(file.getContentType());
9      //      System.out.println(file.getName());
10     //      System.out.println(file.getOriginalFilename());
11     //      System.out.println(file.isEmpty());
12     // 首先判断是否是空文件, 也就是存储空间占用为0的文件
13     if(!file.isEmpty()){
14         // 如果大小在范围要求内正常处理, 否则抛出自定义异常告知用户 (未实现)
15         // 获取原始上传的文件名, 可以作为当前文件的真实名称保存到数据库中备用
16         String fileName = file.getOriginalFilename();
17         // 设置保存的路径
18         String realPath = request.getServletContext().getRealPath("/images");
19         // 保存文件的方法, 指定保存的位置和文件名即可, 通常文件名使用随机生成策略产生, 避免
   文件名冲突问题
20         file.transferTo(new File(realPath, file.getOriginalFilename()));
21     }
22     // 测试一次性上传多个文件
23     if(!file1.isEmpty()){
24         String fileName = file1.getOriginalFilename();
25         // 可以根据需要, 对不同种类的文件做不同的存储路径的区分, 修改对应的保存位置即可
26         String realPath = request.getServletContext().getRealPath("/images");
27         file1.transferTo(new File(realPath, file1.getOriginalFilename()));
28     }
29     if(!file2.isEmpty()){
30         String fileName = file2.getOriginalFilename();
31         String realPath = request.getServletContext().getRealPath("/images");
32         file2.transferTo(new File(realPath, file2.getOriginalFilename()));
33     }
34     return "page.jsp";
35 }

```

## Restful风格配置

### Rest

Rest (REpresentational State Transfer) 一种网络资源的访问风格, 定义了网络资源的访问方式

- 传统风格访问路径

- <http://localhost/user/get?id=1>
- <http://localhost/deleteUser?id=1>
- Rest风格访问路径
  - <http://localhost/user/1>

Restful 就是按照 Rest 风格访问网络资源

优点

- 隐藏资源的访问行为，通过地址无法得知做的是何种操作。
- 书写简化。

## Rest 行为约定方式

GET (查询) <http://localhost/user/1> GET

POST (保存) <http://localhost/user> POST

PUT (更新) <http://localhost/user> PUT

DELETE (删除) <http://localhost/user> DELETE

上述行为是约定方式，约定不是规范，可以打破，所以称 Rest 风格，而不是 Rest 规范

## Restful开发入门



```
1 //设置rest风格的控制器
2 @RestController
3 //设置公共访问路径，配合下方访问路径使用
4 @RequestMapping("/user")
5 public class UserController {
6
7     //rest风格访问路径完整书写方式
8     @RequestMapping("/user/{id}")
9     //使用@PathVariable注解获取路径上配置的具名变量，该配置可以使用多次
10    public String restLocation(@PathVariable Integer id){
11        System.out.println("restful is running ....");
12        return "success.jsp";
13    }
14
15    //rest风格访问路径简化书写方式，配合类注解@RequestMapping使用
16    @RequestMapping("/{id}")
17    public String restLocation2(@PathVariable Integer id){
18        System.out.println("restful is running ....get:"+id);
19        return "success.jsp";
20    }
21
22    //接收GET请求配置方式
23    @RequestMapping(value = "{id}",method = RequestMethod.GET)
24    //接收GET请求简化配置方式
25    @GetMapping("/{id}")
26    public String get(@PathVariable Integer id){
27        System.out.println("restful is running ....get:"+id);
28        return "success.jsp";
29    }
30
31    //接收POST请求配置方式
32    @RequestMapping(value = "{id}",method = RequestMethod.POST)
33    //接收POST请求简化配置方式
34    @PostMapping("/{id}")
35    public String post(@PathVariable Integer id){
36        System.out.println("restful is running ....post:"+id);
37        return "success.jsp";
38    }
39
40    //接收PUT请求简化配置方式
41    @RequestMapping(value = "{id}",method = RequestMethod.PUT)
42    //接收PUT请求简化配置方式
43    @PutMapping("/{id}")
44    public String put(@PathVariable Integer id){
45        System.out.println("restful is running ....put:"+id);
46        return "success.jsp";
47    }
48}
```

```

49 //接收DELETE请求简化配置方式
50 @RequestMapping(value = "{id}",method = RequestMethod.DELETE)
51 //接收DELETE请求简化配置方式
52 @DeleteMapping("{id}")
53 public String delete(@PathVariable Integer id){
54     System.out.println("restful is running ....delete:"+id);
55     return "success.jsp";
56 }
57 }

```

XML | 复制代码

```

1 <!--配置拦截器，解析请求中的参数_method，否则无法发起PUT请求与DELETE请求，配合页面表单使用-->
2 <filter>
3     <filter-name>HiddenHttpMethodFilter</filter-name>
4     <filter-
5 class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
6 </filter>
7 <filter-mapping>
8     <filter-name>HiddenHttpMethodFilter</filter-name>
9     <servlet-name>DispatcherServlet</servlet-name>
10 </filter-mapping>

```

开启SpringMVC对Restful风格的访问支持过滤器，即可通过页面表单提交 PUT 与 DELETE 请求

页面表单使用隐藏域提交请求类型，参数名称固定为\_method，必须配合提交类型 method=post 使用

XML | 复制代码

```

1 <form action="/user/1" method="post">
2     <input type="hidden" name="_method" value="PUT"/>
3     <input type="submit"/>
4 </form>

```

Restful请求路径简化配置方式

Java | 复制代码

```

1 @RestController
2 public class UserController {
3     @RequestMapping(value = "/user/{id}",method = RequestMethod.DELETE)
4     public String restDelete(@PathVariable String id){
5         System.out.println("restful is running ....delete:"+id);
6         return "success.jsp";
7     }
8 }

```

# 校验框架

## 表单校验

表单校验保障了数据有效性、安全性

如果没有表单校验，数据可以随意输入，导致错误的结果。后端表单校验的重要性。

## 表单校验分类

- 校验位置
  - 客户端校验
  - 服务端校验
- 校验内容与对应方式
  - 格式校验
    - 客户端：使用JS技术，利用正则表达式校验
    - 服务端：使用校验框架
  - 逻辑校验
    - 客户端：使用 ajax 发送要校验的数据，在服务端完成逻辑校验，返回校验结果
    - 服务端：接收到完整的请求后，在执行业务操作前，完成逻辑校验

## 表单校验规则

- 长度：例如用户名长度，评论字符数量
- 非法字符：例如用户名组成
- 数据格式：例如 Email 格式、IP 地址格式
- 边界值：例如转账金额上限，年龄上下限
- 重复性：例如用户名是否重复

## 表单校验框架

笼统来说，就是 Java 规定了一套关于验证器的 API，规范先后发了两版，就是 JSR 303 与 JSR 349。

然后提出了基于规范的实现：Bean Validator。

JSR 303 是最早对应了 Bean Validator 1.0 的版本，提供 bean 属性相关校验规则，然后后来扩展了 JSR 349，提出了依赖注入、注解等内容，也就是现在的 Bean Validator。

而 Hibernate 框架中包含一套独立的校验框架 Hibernate Validator


## 快速使用

### 开启校验

@Valid 、 @Validated

```
1 // @Valid 、 @Validated 设定对当前实体类类型参数进行校验
2 @RequestMapping(value = "/addemployee")
3 public String addEmployee(@Valid Employee employee) {
4     System.out.println(employee);
5 }
```


Java

 复制代码

### 设置校验规则

```
1 // @NotNull @NotNull
2 public class Employee{
3     @NotNull(message = "姓名不能为空")
4     private String name;//员工姓名
5 }
```

Java

 复制代码

每个校验规则所携带的参数不同，根据校验规则进行相应的调整。

具体的校验规则查看对应的校验框架进行获取。

### 获取错误信息

```

1  @RequestMapping(value = "/addemployee")
2  public String addEmployee(@Valid Employee employee, Errors errors, Model
    model){
3      System.out.println(employee);
4      if(errors.hasErrors()){
5          for(FieldError error : errors.getFieldErrors()){
6              model.addAttribute(error.getField(),error.getDefaultMessage());
7          }
8          return "addemployee.jsp";
9      }
10     return "success.jsp";
11 }

```

可以通过形参 Errors 获取校验结果数据，通过 Model 接口将数据封装后传递到页面显示

```

1  <form action="/addemployee" method="post">
2      员工姓名: <input type="text" name="name"><span style="color:red">${name}
    </span><br/>
3      员工年龄: <input type="text" name="age"><span style="color:red">${age}
    </span><br/>
4      <input type="submit" value="提交">
5  </form>

```

通过形参Errors获取校验结果数据，通过Model接口将数据封装后传递到页面显示

页面获取后台封装的校验结果信息

## 多规则校验

同一个属性可以添加多个校验器

```

1  @NotNull(message = "请输入您的年龄")
2  @Max(value = 60,message = "年龄最大值不允许超过60岁")
3  @Min(value = 18,message = "年龄最小值不允许低于18岁")
4  private Integer age;//员工年龄

```

3种判定空校验器的区别

表单数据	@NotNull	@NotEmpty	@NotBlank
String s = null;	false	false	false
String s = " " ;	true	false	false
String s = "  ";	true	true	false
String s = "Jock" ;	true	true	true

## 嵌套校验

Java | 复制代码

```

1 public class Employee {
2     //实体类中的引用类型通过标注@Valid注解，设定开启当前引用类型字段中的属性参与校验
3     @Valid
4     private Address address;
5 }

```

注意：开启嵌套校验后，被校验对象内部需要添加对应的校验规则

## 分组校验

- 同一个模块，根据执行的业务不同，需要校验的属性会有不同
  - 新增用户
- 修改用户
- 对不同种类的属性进行分组，在校验时可以指定参与校验的字段所属的组类别
  - 定义组（通用）
  - 为属性设置所属组，可以设置多个
  - 开启组校验

Java | 复制代码

```

1 public interface GroupOne {
2 }
3 public String addEmployee(@Validated({GroupOne.class}) Employee employee){
4 }
5 @NotEmpty(message = "姓名不能为空",groups = {GroupOne.class})
6 private String name;//员工姓名

```

# SSM 整合

## 整合步骤

1. Spring
2. MyBatis
3. Spring整合MyBatis
4. SpringMVC
5. Spring整合SpringMVC

## 项目结构搭建

- 项目基础结构搭建
- 创建项目，组织项目结构，创建包
- 创建表与实体类
- 创建三层架构对应的模块、接口与实体类，建立关联关系
  - 数据层接口（代理自动创建实现类）
  - 业务层接口+业务层实现类
  - 表现层类