# 尚马教育 JAVA 高级课程

# SpringMVC高级

文档编号：C06

创建日期： 2017-07-07

最后修改日期：2021-02-22

版 本 号：V3.5

电子版文件名：尚马教育-第三阶段-6.springMVC高级.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Mybatis版本更新 | V2.0 |
| 2019-08-09 | 冯勇涛 | 课件格式以及课程深度加深 | V3.0 |
| 2019-11-13 | 冯勇涛 | 课件完善 | V3.1 |
| 2021-02-22 | 冯勇涛 | 深化内容，优化结构 | V3.5 |

目录

[1. 异常处理 2](#_Toc29514_WPSOffice_Level1)

[1.1. 局部异常 2](#_Toc1767_WPSOffice_Level2)

[1.2. 全局异常 2](#_Toc4140_WPSOffice_Level2)

[2. 静态资源处理 4](#_Toc1767_WPSOffice_Level1)

[2.1. 对静态资源做映射 4](#_Toc9934_WPSOffice_Level2)

[3. 文件上传 5](#_Toc4140_WPSOffice_Level1)

[4. Json支持 6](#_Toc9934_WPSOffice_Level1)

[5. 拦截器 7](#_Toc23045_WPSOffice_Level1)

[5.1. 创建登录拦截器类 7](#_Toc23045_WPSOffice_Level2)

[5.2. 配置拦截器 7](#_Toc28602_WPSOffice_Level2)

[5.3. 拦截器执行流程图 7](#_Toc8835_WPSOffice_Level2)

知识点：

|  |
| --- |
| 掌握SpringMVC的静态资源处理；  掌握SpringMVC的全局统一异常处理；  掌握SpringMVC的文件上传处理；  掌握拦截器；  配置跨域 |

## 静态资源处理

由于springmvc前端控制器的映射路径为/，因此前端的静态资源请求会被sprngmvc处理，进入springmvc处理流程，查找对应的handler，此时会出现404错误，因为后端并没有对应的处理器处理静态资源，需要对静态资源做单独处理。

在tomcat服务器下有默认的两个servlet：JspServlet与DefaultServelt，可在conf/web.xml文件下查看两个serlvet。

JspServlet：处理jsp后缀的请求，由于这个servlet的存在，前端访问jsp文件可以直接访问。如下：

|  |
| --- |
| <servlet>  <servlet-name>jsp</servlet-name>  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>  <init-param>  <param-name>fork</param-name>  <param-value>false</param-value>  </init-param>  <init-param>  <param-name>xpoweredBy</param-name>  <param-value>false</param-value>  </init-param>  <load-on-startup>3</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>jsp</servlet-name>  <url-pattern>\*.jsp</url-pattern>  <url-pattern>\*.jspx</url-pattern>  </servlet-mapping> |

DefaultServlet：该servelt默认的映射路径为/,该路径会被springmvc的DispatcherServlet覆盖进而失效。

|  |
| --- |
| <servlet>  <servlet-name>default</servlet-name>  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>  <init-param>  <param-name>debug</param-name>  <param-value>0</param-value>  </init-param>  <init-param>  <param-name>listings</param-name>  <param-value>false</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |

### 重新配置DefaultServlet

在web.xml中配置DefaultServlet，重新配置该servlet的url映射路径，指定资源资源的后缀。

|  |
| --- |
| <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.html</url-pattern>  <url-pattern>\*.css</url-pattern>  <url-pattern>\*.js</url-pattern>  <url-pattern>\*.jpg</url-pattern>  <url-pattern>\*.png</url-pattern>  <url-pattern>\*.gif</url-pattern>  </servlet-mapping> |

### 使用mvc:default-servlet-handler配置

|  |
| --- |
| <mvc:default-servlet-handler default-servlet-name="default"></mvc:default-servlet-handler> |

注意点：此种方式仍然是启用DefaultServlet，与1.1方法没有区别，default-servlet-name属性名默认=”default”，可以省略。

### 使用mvc:resources配置

使用<mvc:resources/>标签对特定字符串开头的url进行映射。

|  |
| --- |
| <!--内部提供资源资源处理器。url以/static/开头的请求，拼接/static/hello.css-->  <mvc:resources mapping="/static/\*\*" location="/static/"></mvc:resources>  <mvc:resources mapping="/page/\*\*" location="/page/"></mvc:resources>  <mvc:resources mapping="/upload/\*\*" location="/upload/"></mvc:resources> |

## 异常处理

### 局部异常处理

在某个handler处理器类中进行局部异常处理，使用@ExceptionHandler来定义异常处理方法，仅针对当前类中的接口有效。

|  |
| --- |
| //局部异常  @ExceptionHandler  @ResponseBody  **public** ResultBean doException(Exception e) {  System.***out***.println("进入局部异常处理方法！！！"+e.getMessage());  **return** ResultBean.*status*(Status.***ERROR***);  } |

局部异常能够对不同的异常类型分别进行处理，但仅能够对当前Handler中的方法产生异常有效。不建议使用。

### 全局异常处理

使用HandlerExceptionResolver接口实现全局自定义异常捕获。定义自定义异常处理类,实现HandlerExceptionResolver接口,并实现resolver方法进行自定义异常信息处理及页面跳转。

|  |
| --- |
| @Component **public class** MyGlobalExceptionHandler **implements** HandlerExceptionResolver {  @Override  **public** ModelAndView resolveException(HttpServletRequest req, HttpServletResponse resp, Object o, Exception e) {  HandlerMethod hm =(HandlerMethod)o;  Class<?> beanType = hm.getBeanType();  Method method = hm.getMethod();   ModelAndView nv = **new** ModelAndView();  nv.setViewName(**"error"**);  nv.addObject(**"errorMsg"**,e.toString());  nv.addObject(**"time"**,System.currentTimeMillis());  nv.addObject(**"name"**,beanType.getName()+**"."**+method.getName());  **return** nv;  } } |

此种方法只能对所有请求返回统一的信息，做不到针对不同的异常返回不同的数据给前端，不具有实用性。

### 全局统一异常处理

使用ControllerAdvice控制层增强注解来实现全局异常处理，使用ExceptionHandler注解来实现对不同的异常进行不同的处理。

#### 定义状态枚举对象

|  |
| --- |
| **public enum** StatusEnum {  LOGIN\_SUC(20000,**"登陆成功"**),  LOGIN\_ERROR(50001,**"登陆失败"**),  OPS\_SUC(20002,**"操作成功"**),  OPS\_ERROR(50002,**"操作失败"**),  LOGIN\_PWD\_ERROR(50003,**"用户名密码有误"**),  FILE\_NOT\_EXITS(50004,**"文件不存在"**),  NO\_LOGIN(50005,**"用户未登陆"**);  **private** Integer status;  **private** String msg;  StatusEnum(Integer status, String msg) {  **this**.status = status;  **this**.msg = msg;  }  **public** Integer getStatus() {  **return** status;  }  **public void** setStatus(Integer status) {  **this**.status = status;  }  **public** String getMsg() {  **return** msg;  }  **public void** setMsg(String msg) {  **this**.msg = msg;  } } |

#### 定义响应状态实体

|  |
| --- |
| **public class** StatusBean {  **private** Integer status;  **private** String msg;  **public** StatusBean(StatusEnum se) {  **this**.status = se.getStatus();  **this**.msg = se.getMsg();  }  **public** Integer getStatus() {  **return** status;  }  **public void** setStatus(Integer status) {  **this**.status = status;  }  **public** String getMsg() {  **return** msg;  }  **public void** setMsg(String msg) {  **this**.msg = msg;  } } |

#### 定义返回数据实体对象

|  |
| --- |
| **public class** ResponseBean **extends** StatusBean{  **private** Object data;  **public** ResponseBean(StatusEnum se, Object data) {  **super**(se);  **this**.data = data;  }  **public** Object getData() {  **return** data;  }  **public void** setData(Object data) {  **this**.data = data;  } } |

#### 自定义异常对象

|  |
| --- |
| **public class** MvcException **extends** Exception {  **private** StatusEnum se;  **public** MvcException(StatusEnum se) {  **this**.se = se;  }  **public** StatusEnum getSe() {  **return** se;  }  **public void** setSe(StatusEnum se) {  **this**.se = se;  } } |

#### 定义全局统一异常处理对象

|  |
| --- |
| @ControllerAdvice **public class** GlobalExceptionHandler {  *//对其他异常处理* @ExceptionHandler(Exception.**class**)  **public** ResponseEntity doException(Exception e){  System.out.println(e.toString());  **return** ResponseEntity.ok(**new** StatusBean(StatusEnum.OPS\_ERROR));  }   *//对MvcException异常处理* @ExceptionHandler(MvcException.**class**)  **public** ResponseEntity doMvcException(MvcException e){  System.out.println(e.toString());  **return** ResponseEntity.ok(**new** StatusBean(e.getSe()));  } } |

#### 测试异常处理，控制层抛出异常信息

当用户名密码有误时，可以直接抛出MvcException异常，交给GlobalExceptionHandler全局统一异常处理对象来处理MvcException异常，返回异常数据给前端。

|  |
| --- |
| @PostMapping(**"login"**) **public** ResponseEntity doLogin(@RequestBody Sysuser user, HttpSession session) **throws** MvcException {  Sysuser loginUser = us.login(user);  **if**(loginUser!=**null**){  session.setAttribute(**"login\_user"**,loginUser);  loginUser.setUpwd(**""**);  **return** ResponseEntity.ok(**new** ResponseBean(StatusEnum.LOGIN\_SUC,loginUser));}**else**{  **throw new** MvcException(StatusEnum.LOGIN\_PWD\_ERROR);  } } |

## 文件上传下载

SpringMVC对apache的commons-fileupload组件进行封装，避免了繁琐的request解析操作。

### 添加commons-filupload依赖

加入commons-fileupload-xxx.jar、commons-io-xxx.jar。

|  |
| --- |
|  |

### 配置文件处理器

在SpringMVC中配置multipart文件处理器。

|  |
| --- |
| <!-- id:必须是multipartResolver -->  <bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>  <property name=*"maxUploadSize"* value=*"50000000"*></property><!-- 最大文件大小 -->  <property name=*"maxInMemorySize"* value=*"10000000"*></property><!-- 临时文件域 -->  <property name=*"defaultEncoding"* value=*"UTF-8"*></property><!-- 中文文件名 -->  <property name=*"uploadTempDir"* value=*"/upload/tmp"*></property><!-- 临时文件存储路径 -->  </bean> |

### 编写文件上传方法

在handler类中添加upload()方法，参数列表添加MultipartFile类型形参接收上传的文件。多文件的话，使用MultipartFile数组。

|  |
| --- |
| @Controller **public class** FileHandler {  @PostMapping(**"upload"**)  **public** ResponseEntity doUpload(MultipartFile userImg, HttpServletRequest req) **throws** IOException {  **byte**[] bytes = userImg.getBytes();*//得到文件字节数组内容 // InputStream inputStream = userImg.getInputStream();//得到文件输入流内容，适合于大文件* String originalFilename = userImg.getOriginalFilename();*//得到文件名* **long** size = userImg.getSize();*//得到文件大小* String realPath = req.getServletContext().getRealPath(**"/"**);  String savepath =**"/upload/"**+originalFilename;  File target = **new** File(realPath+savepath);  FileUtils.writeByteArrayToFile(target,bytes);  Map<String,Object> result = **new** HashMap();  result.put(**"filesize"**,size);  result.put(**"filename"**,originalFilename);  result.put(**"uploadtime"**,System.currentTimeMillis());  result.put(**"savepath"**,savepath)  **return** ResponseEntity.ok(**new** ResponseBean(StatusEnum.OPS\_SUC,result));  } } |

注意：多文件上传，upload方法MultipartFile形参改为数组,或多个MultipartFile类型的形参。

### 编写文件下载方法

|  |
| --- |
| @GetMapping(**"down"**) **public** ResponseEntity doDownLoad(String filepath,HttpServletRequest req) **throws** MvcException, IOException {  String realPath = req.getServletContext().getRealPath(**"/"**);  String path = realPath+filepath;  File file = **new** File(path);  **if**(file.exists()){  String filename=file.getName();  **byte**[] bytes = FileUtils.readFileToByteArray(file);  HttpHeaders headers = **new** HttpHeaders();  headers.add(**"Content-Disposition"**,**"attachment;filename="**+URLEncoder.encode(filename,**"UTF-8"**));  **return new** ResponseEntity(bytes,headers,HttpStatus.OK);  }**else**{  **throw new** MvcException(StatusEnum.FILE\_NOT\_EXITS);  } } |

## **拦截器**

拦截器类似于servlet过滤器，属于MVC框架中的概念，起到拦截用户请求进行过滤的作用。

创建自定义拦截器类,从HandlerInterceptor接口派生拦截器类,分别实现preHandle，postHandle，afterCompletion方法，preHandle是最常用的方法，属于前置拦截。

preHandle在handler执行之前调用。

postHandle在ModelAndView返回之前调用。

afterCompletion在handler执行完成之后调用(报错也执行)。

### 创建登录拦截器类

|  |
| --- |
| **public** **class** LoginInterceptor **implements** HandlerInterceptor{  //前拦截，在执行handlerMethod方法之前。  @Override **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  HttpSession session = request.getSession();  Object login\_user = session.getAttribute(**"login\_user"**);  **if**(login\_user!=**null**)  **return true**;  **else**{  **throw new** MvcException(StatusEnum.NO\_LOGIN);  } }  } |

### 配置拦截器

|  |
| --- |
| <mvc:interceptors>  <mvc:interceptor>  <mvc:mapping path=*"/\*\*"*/><!--拦截路径-->  <mvc:exclude-mapping path=*"/static/\*\*"*/><!--不拦截路径-->  <mvc:exclude-mapping path=*"/user/dologin"*/><!--不拦截路径-->  <bean class=*"com.javasm.interceptor.LoginInterceptor"*></bean><!--拦截器对象-->  </mvc:interceptor>  </mvc:interceptors> |

### 拦截器执行流程图

|  |
| --- |
|  |

## 跨域请求

### 什么是跨域：

跨域是指浏览器不允许当前页面的所在的源去请求另一个源的数据。源指协议，端口，域名。只要这个3个中有一个不同就是跨域。 这里列举一个经典的列子：

前后端分离项目产生的跨域请求问题。

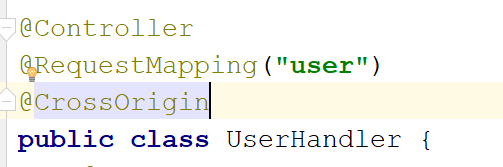
前端独立部署服务器，后端独立部署服务器

两者协议，ip，端口有一个不同，违反浏览器的同源策略，会造成跨域访问失败问题。

解决方案如下：

### 方法1：@CrossOrigin注解

该注解添加到在处理器方法或类上。



缺点：只能够针对当个类处理。

### 方法2：springMVC的全局配置

|  |
| --- |
| <mvc:cors>  <mvc:mapping path="/\*\*"   allowed-origins="\*"   allowed-headers="\*"   allowed-methods="\*"   allow-credentials="true"/> </mvc:cors> |

缺点：方法1与方法2在正常情况下都能够正常处理跨域请求，一旦项目中引入拦截器，则跨域处理失效，因为springmvc的cors配置也是基于拦截器实现的，并且默认作为最后一个拦截器处理，这样会导致在跨域拦截器之前执行了业务拦截器。比如：引入登录拦截器，A请求被登录拦截，则不向下继续执行跨域拦截器，本次请求就会产生跨域异常，该请求则不支持跨域，客户端收到错误信息。

### 方法3：CrosFilter过滤器

|  |
| --- |
| <!-- 这个配置需要放到Spring的配置文件中，不能放到SpringMVC的配置文件，因为SpringMVC的加载是基于Servlet，它是晚于Filter的 -->  <bean id="corsFilter" class="org.springframework.web.filter.CorsFilter">  <constructor-arg name="configSource">  <bean class="org.springframework.web.cors.UrlBasedCorsConfigurationSource">  <property name="corsConfigurations">  <map>  <entry key="/\*\*">  <bean class="org.springframework.web.cors.CorsConfiguration">  <property name="allowCredentials" value="true"/>  <property name="allowedMethods">  <list>  <value>GET</value>  <value>POST</value>  <value>HEAD</value>  <value>PUT</value>  <value>DELETE</value>  <value>OPTIONS</value>  </list>  </property>  <property name="allowedHeaders" value="\*"/>  <property name="allowedOrigins" value="\*"/>  </bean>  </entry>  </map>  </property>  </bean>  </constructor-arg>  </bean>  <!-- web.xml  由于CorsFilter跟通常的Filter不一样，Spring对其做了很多改造，所以加载的方式要使用DelegatingFilterProxy，通过Spring的方式把它放到容器中  -->  <filter>  <filter-name>myCorsFilter</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <init-param>  <param-name>targetBeanName</param-name>  <param-value>corsFilter</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>myCorsFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

### 方法4：自定义跨域处理过滤器

|  |
| --- |
| public class MyCrosFilter implements Filter {  @Override  public void init(FilterConfig filterConfig) throws ServletException {  }  @Override  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  try {  HttpServletRequest httpRequest = (HttpServletRequest) request;  HttpServletResponse httpResponse = (HttpServletResponse) response;   // 预检请求，请求头Origin是客户端地址，要求跨域头不能是\*  String origin = httpRequest.getHeader("Origin");  if (origin == null) {  httpResponse.addHeader("Access-Control-Allow-Origin", "\*");  } else {  httpResponse.addHeader("Access-Control-Allow-Origin", origin);  }  httpResponse.addHeader("Access-Control-Allow-Headers",  "Origin, x-requested-with, Content-Type, Accept,X-Cookie");  httpResponse.addHeader("Access-Control-Allow-Credentials", "true");  httpResponse.addHeader("Access-Control-Allow-Methods", "GET,POST,PUT,OPTIONS,DELETE");  //预检请求，直接通过。  if (httpRequest.getMethod().equals("OPTIONS")) {  httpResponse.setStatus(HttpServletResponse.SC\_OK);  return;  }  chain.doFilter(request, response);  } catch (Exception e) {  throw e;  }  }  @Override  public void destroy() {  } } |

### 方法5：springboot的解决方式：

|  |
| --- |
| @Configuration  public class MyConfiguration {  //springmvc中 没有FilterRegistrationBean这个类  @Bean  public FilterRegistrationBean这个类 corsFilter() {  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  CorsConfiguration config = new CorsConfiguration();  config.setAllowCredentials(true);  config.addAllowedOrigin("http://domain1.com");  config.addAllowedHeader("\*");  config.addAllowedMethod("\*");  source.registerCorsConfiguration("/\*\*", config);  FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));  bean.setOrder(0);  return bean;  }  } |

### 方法6：其他方案（了解）

基于nginx做url代理；未学习nginx，暂不学习。

基于node的一个包[http-proxy-middleware](https://www.npmjs.com/package/http-proxy-middleware)，基于node组件启动一个后端程序做代理，把所有程序转发到服务器。(属于前端技能)。

## 用户登陆状态处理

#### 服务端HttpSession保存登录状态

采用服务端保存客户端会话状态信息，安全可靠。

缺点：应用服务器搭建集群时，在多个服务器之间需要解决session共享,有成熟解决方案：spring-session。

#### 服务端签发token机制

服务端不保存会话状态信息，当用户登陆，服务端生成token字符串，返回客户端保存localStorage或cookie。每次请求都携带token信息在请求头中，发送服务端，服务端校验是否合法请求。

依赖组件：auto0或jjwt，两者都可以用来生成token。

缺点：token过长，效率低；服务端无法控制token强制失效，安全性稍低；需要自己实现token刷新。