# springMVC概述

## 概述

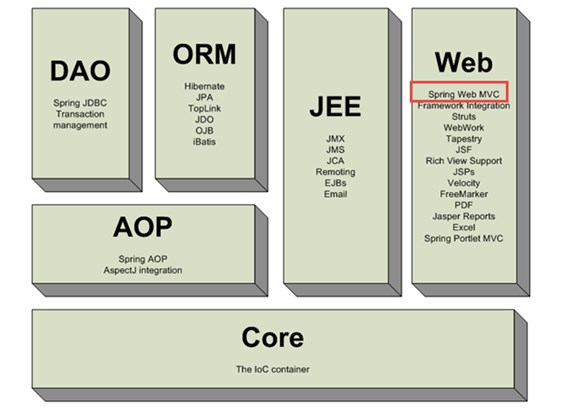
Spring MVC是Spring提供的一个强大而灵活的web框架。springmvc是spring的一个模块，不需要与spring整合。，借助于注解，Spring MVC提供了几乎是POJO的开发模式，使得控制器的开发和测试更加简单。这些控制器一般不直接处理请求，而是将其委托给Spring上下文中的其他bean，通过Spring的依赖注入功能，这些bean被注入到控制器中。

Spring MVC主要由DispatcherServlet、处理器映射、处理器(控制器)、视图解析器、视图组成。他的两个核心是两个核心：

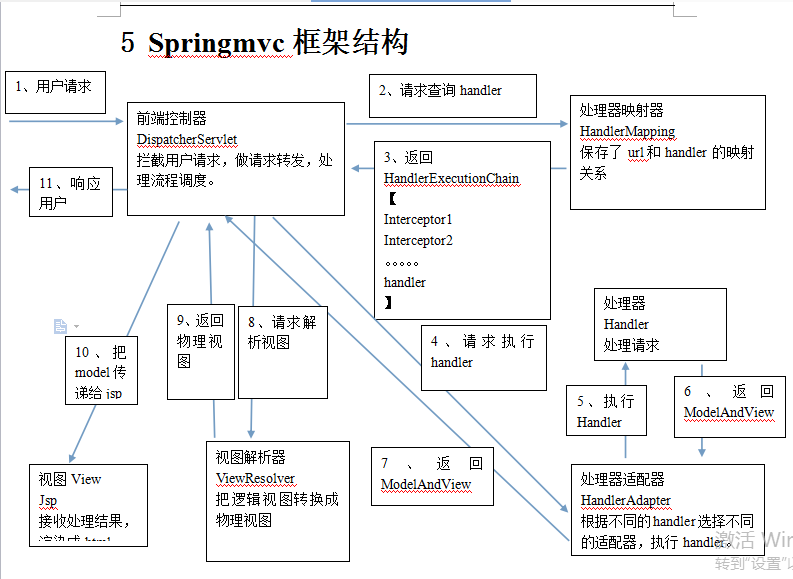
处理器映射：选择使用哪个控制器来处理请求

视图解析器：选择结果应该如何渲染

通过以上两点，Spring MVC保证了如何选择控制处理请求和如何选择视图展现输出之间的松耦合。



## springmvc的原理图



当一个请求到来时，DispatcherServlet 首先通过请求和事先解析好的 HandlerMapping配置，找到对应的处理器（controller)，这样就准备开始运行处理器和拦截器组成的执行链，而运行处理器需要有一个对应的环境，这样它就有了一个处理器的适配器（HandlerAdapter)，通过这个适配器就能运行对应的处理器及其拦截器，这里的处理器包含了控制器的内容和其他增强的功能，在处理器返回模型和视图给DispacherServlet后，DispacherServlet就会把对应的视图信息传递给视图解析器（ViewResolver)。注意，这一步取决于是否使用逻辑视图，如果是逻辑视图，那么视图解析器就会解析它，然后把模型渲染到视图中去，最后响应用户的请求；如果不是逻辑视图，则不会进行处理，而是直接通过视图渲染数据模型。

1、用户向服务器发送请求，请求被SpringMVC的前端控制器DispatcherServlet截获。

2、DispatcherServlet需要知道在哪个处理器（controlller）中执行，此时需要向处理器映射器（handlermapping）请求handler ，如果有，则返回执行链（HandleExecutionchain），顶底层是一个类，依次有拦截器（HandleInterceptor，这里可能有多个拦截器），然后有handle。

DispatcherServlet对请求的URL（统一资源定位符）进行解析，得到URI(请求资源标识符)，然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象，包括Handler对象以及Handler对象对应的拦截器，这些对象都会被封装到一个HandlerExecutionChain对象当中返回。

3、DispatcherServlet根据获得的Handler，选择一个合适的HandlerAdapter。HandlerAdapter的设计符合面向对象中的单一职责原则，代码结构清晰，便于维护，最为重要的是，代码的可复制性高。HandlerAdapter会被用于处理多种Handler，调用Handler实际处理请求的方法。

4、

提取请求中的模型数据，开始执行Handler(Controller)。在填充Handler的入参过程中，根据配置，spring将帮助做一些额外的工作

消息转换：将请求的消息，如json、xml等数据转换成一个对象，将对象转换为指定的响应信息。

数据转换：对请求消息进行数据转换，如String转换成Integer、Double等。

数据格式化：对请求的消息进行数据格式化，如将字符串转换为格式化数字或格式化日期等。

数据验证：验证数据的有效性如长度、格式等，验证结果存储到BindingResult或Error中。

5、Handler执行完成后，向前端控制器DispatcherServlet返回一个ModelAndView对象，ModelAndView对象是springmvc的一个底层对象，其中应该包含视图名或视图模型。

6、根据返回的ModelAndView对象，选择一个合适的ViewResolver(视图解析器)返回给DispatcherServlet。

7、视图的类型有多重多样，DispatcherServlet会请求解析视图，Viewresolve解析完之后返回的是一个物理视图，也就是jsp

8、将模型数据填充到request域中，返回给用户。视图渲染的数据，在ModelAndView中。

9、最后响应给用户。

总结：以上8个步骤，DispatcherServlet、HandlerMapping、HandlerAdapter和ViewResolver等对象协同工作，完成SpringMVC请求—>响应的整个工作流程，这些对象完成的工作对于开发者来说都是不可见的，开发者并不需要关心这些对象是如何工作的，开发者，只需要在Handler(Controller)当中完成对请求的业务处理。

概念：

前端控制器：DispatcherServlet（中央处理器）

作用：作用接收请求，响应结果，相当于转发器。有了该控制器，减少了其他组件的耦合性，不许要程序员开发。

处理器映射器Handlermapping

作用：根据请求的url查找对应的handler，也就是controller

处理器适配器：HandlerAdapter

作用：按照特定的规则（HandlerAdapter要求的规则）去执行Handler，

注：编写Handler是按照HandlerAdapter的要求去做，这样适配器才可以正确的执行handler。

处理器适配执行目标方法：（需要程序开发）编写的规则需要根据HandlerAdapter。

视图解析器：ViewResolver

作用：进行视图解析，根据逻辑视图名，解析成真正的视图（view）

视图view（需要程序员开发jsp）：

view是一个接口，实现类支持不同的View类型（jsp、pdf）

## modelAndView

ModelAndView：是一个对象，该对象中包含了一个model属性和一个view属性。

model：其实是一个ModelMap类型。其实ModelMap是一个LinkedHashMap的子类。

view：包含了一些视图信息。

视图解析器：视图解释器解析ModelAndVIew是，其中model本生就是一个Map的实现类的子类。视图解析器将model中的每个元素都通过request.setAttribute(name, value);添加request请求域中。这样就可以在JSP页面中通过EL表达式来获取对应的值。

1.封装model：在封装成model的过程中，如果我们往modelAndview中添加多个参数时候，其实是把数据都存放在了一个Map中，让后循环Map，把数据从map中取出来在通过setAttribute放入域域对象中，我们在前台通过域对象就能拿到数据。

|  |
| --- |
| //把数据和跳转地址封装进modelAndView  @RequestMapping("/testModelAndView")  public ModelAndView testModelAndView(){  String viewName = SUCCESS;  ModelAndView modelAndView = new ModelAndView(viewName);  //添加模型数据到 ModelAndView 中.  modelAndView.addObject("time", new Date());  return modelAndView;  }  // 页面通过域对象拿到数据  time: ${requestScope.time } |

2.跳转地址：跳转地址是通过SpringMVC配置文件中拿到前缀+返回字符串+springMvc中拿到后缀组成，在解析视图的时候把拼装的地址通过转发的方式跳转到对应的页面

|  |
| --- |
| //我们自己的Map封装了我们自己的数据  @RequestMapping("/testMap")  public String testMap(Map<String, Object> map){  map.put("name", Arrays.asList("小明","小红","小军"));  return SUCCESS;  }  //通过域对象取出我们的模型数据  name: ${requestScope.name } |

## SpringMVC接口解释

（1）DispatcherServlet接口：

Spring提供的前端控制器，所有的请求都有经过它来统一分发。在DispatcherServlet将请求分发给Spring Controller之前，需要借助于Spring提供的HandlerMapping定位到具体的Controller。

（2）HandlerMapping接口：

能够完成客户请求到Controller映射。

（3）Controller接口：

需要为并发用户处理上述请求，因此实现Controller接口时，必须保证线程安全并且可重用。

Controller将处理用户请求，这和Struts Action扮演的角色是一致的。一旦Controller处理完用户请求，则返回ModelAndView对象给DispatcherServlet前端控制器，ModelAndView中包含了模型（Model）和视图（View）。

从宏观角度考虑，DispatcherServlet是整个Web应用的控制器；从微观考虑，Controller是单个Http请求处理过程中的控制器，而ModelAndView是Http请求过程中返回的模型（Model）和视图（View）。

（4）ViewResolver接口：

Spring提供的视图解析器（ViewResolver）在Web应用中查找View对象，从而将相应结果渲染给客户。

# SpringMVC入门程序

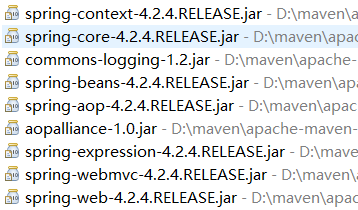
## 环境准备

1，java环境

2，myeclipse

3，springmvc版本 4.3

需要spring的所有jar包（一定要有spring-webmvc-4.3.7.RELEASE.jar）



## 创建springmvc.xml声明

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入头文件 -->  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  </beans> |

## 配置前端控制器web.xml

|  |
| --- |
| <!—1、配置springmvc的前端控制器开始 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servletclass>  org.springframework.web.servlet.DispatcherServlet  </servlet-class>  <!—通过contextConfigLocation配置springmvc需要加载的配置文件  配置文件中包含映射器和适配器 -->  <如果不配置contextConfigLocation，默认加载的是/WEB-INF/serlvet名称-servlet.xml（springmvc-servlet.xml）>  <init-param>//<!—2、创建对象之后，立即加载spring的配置文件，配置文件包含：处理器映射器、处理器配置器、视图解析器、拦截器等配置。  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <!—1、配置当tomcat启动时就创建 DispatcherServlet对象 -->  <load-on-startup>1</load-on-startup>  </servlet>  <!—3、当有请求访问时，会先执行拦截器中的配置，然后在执行对应handler，然后在执行视图解析器。-->  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!-- 所有以.action结局的请求都 交给 DispatcherServlet进行解析。 -->  <url-pattern>\*.action</url-pattern>  <!—所有的请求都交给DispatcherServlet解析，静态文件需要配置不让DispatcherServlet解析，使用此种方式可以实现RESTful的风格url  <url-pattern>/</url-pattern>  3、这样的配置不对，使用这种配置，最终要转发到一个jsp页面，任然后由DispatcherServlet解析jsp不能根据jsp页面找到handler，会报错。  <!-- <url-pattern>/\*</url-pattern> -->  </servlet-mapping>  <!-- 配置springmvc的前端控制器结束 --> |

### url-pattern路径问题

/\*和/都是拦截所有请求；

/：会拦截所有请求，但是不会拦截\*.jsp；能保证jsp访问正常；

/\*的范围更大；还会拦截到\*.Jsp这些请求；一但拦截Jsp页面就不能显示了；

## 创建log4j.properties

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # MyBatis logging configuration...  log4j.logger.org.mybatis.example.BlogMapper=TRACE  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

## 配置springmvc.xml

内容:

1、处理器映射器

2、处理器适配器

3、视图解析器

注：

1、这三个的顺序怎么都可以。

2、最终的目的是去执行对应的处理器。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入头文件 -->  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  <!-- 配置控制器 -->  <!--配置控制器 -->  <bean id=*"userController"* name=*"/userController.action"* class=*"com.sx.controller.UserController"* >  <!-- 配置处理器映射器：有两个处理器映射器，且多个映射器之间可以并存，前端控制器判断，能那个url处理，就让那个url处理。  1:org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping把bean的名字做为url进行查找，也就是handler必须配置name属性  2:org.springframework.web.servlet.handler.SimpleUrlHandlerMapping 里面可以配置属性【后面讲】  -->  <!--1、使用控制器的name属性来匹配 -->  <bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*> </bean>  <!--2、简单url处理器映射。使用控制器的id来匹配控制器 -->  <bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>  <property name=*"mappings"*>  <props>  <!--key为访问路径，值为控制器的id属性，一个控制器可以有多个映射器-->  <prop key=*"/userController1.action"*>userController</prop>  <prop key=*"/userController2.action"*>userController</prop>  <prop key=*"/userController3.action"*>userController</prop>  </props>  </property>  </bean>  <!-- 配置处理器适配器 ：所有的处理器适配器都实现handleradapter接口，框架判断是否实现了该接口，来判断是否为适配器。  有两个处理器适配器  1:org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter写的controller要实现Controller的接口  2:org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter 写controler要实现HttpRequestHandler接口【后面说】    <!--配置适配器，且这两中适配器可以共存，且>  <!--1、这种方法要求handle实现Controller接口 -->  <bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"*></bean>  <!--2、这种方法要求handle实现HttpRequestHandler接口 -->  <bean class=*"org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"*></bean>  <!-- 配置视图解析器 ：解析jsp视图，默认使用jstl标签，classpath有jstl包-->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  </bean>  </beans> |

### 注解实现映射器和适配器

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入头文件 -->  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  <!-- 扫描 -->  <context:component-scan base-package=*"com.sxt.controller"*></context:component-scan>  <!-- 适配器和控制器的配置 -->  <mvc:annotation-driven></mvc:annotation-driven>  <!-- 配置视图解析器 -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  </bean>  <!-- 文件上传的配置 -->  <!-- 拦截器的配置 -->  <!-- 静态文件过滤 -->  <mvc:default-servlet-handler/>  </beans> |

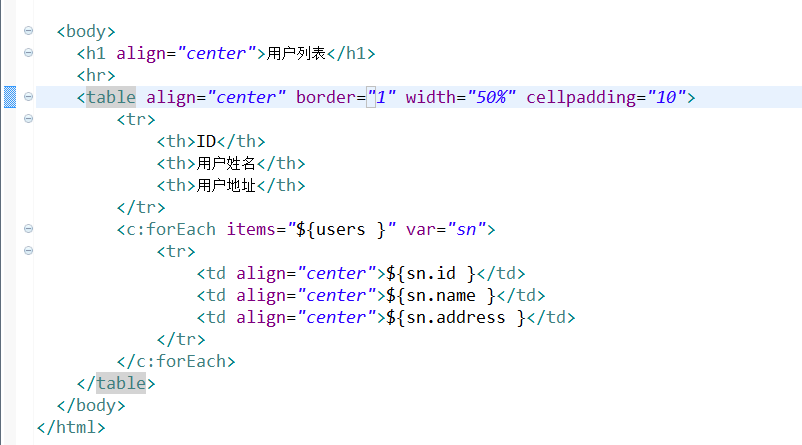
### 注意点

如果

## 开发handler

|  |
| --- |
| //1、实现Controller接口  @Override  **public** ModelAndView handleRequest(HttpServletRequest arg0,  HttpServletResponse arg1) **throws** Exception {  User user = **new** User(1, "sx", "北京");  User user2 = **new** User(2, "lb", "上海");  User user3 = **new** User(3, "lx", "深圳");  List<User> userList = **new** ArrayList<User>();  userList.add(user);  userList.add(user3);  userList.add(user2);  //返回ModelAndView  ModelAndView modelAndView = **new** ModelAndView();  //相当于request的setAttribut，在jsp页面通过userList获取数据  modelAndView.addObject("userList",userList);  //指定视图  modelAndView.setViewName("/WEB-INF/user/list.jsp");  **return** modelAndView;  } |

## 创建页面list.jsp



## 总结

1、客户端点击链接会发送http://1ocalhost:8e8e/springmvc/hello来到tomcat服务器；

2、SpringMVC的前端控制器收到所有请求；

3、来看请求地址和@RequestMappring标注的哪个匹配，来找到到底使用那个类的哪

4、前端控制器找到了目标处理器类和目标方法，直接利用返回执行目标方法；

5、方法执行完成以后会有一个返回值；SpringMVC认为这个返回值就是要去的页面地址

6、拿到方法返回值以后；用视图解析器进行拼串得到完整的页面地址；

7、拿到页面地址，前端控制器帮我们转发到页面；|

# 入门程序总结

通过入门程序理解springmvc前端控制器、处理器映射器、处理器适配器、视图解析器用法。

## 九大组件

全是接口：提供了规范，

|  |
| --- |
| /\*\* 文件上传\*/  **private** MultipartResolver multipartResolver;  /\*\* 区域信息解析器，和国际化有关\*/  **private** LocaleResolver localeResolver;  /\*\* 主题解析器：强大的主题效果跟换\*/  **private** ThemeResolver themeResolver;  /\*\* handler的映射器\*/  **private** List<HandlerMapping> handlerMappings;  /\*\* handler的适配器\*/  **private** List<HandlerAdapter> handlerAdapters;  /\*\* springmvc强大的异常解析功能：异常解析器\*/  **private** List<HandlerExceptionResolver> handlerExceptionResolvers;  /\*\* 如果方法没有返回值，使用请求的地址作为视图名\*/  **private** RequestToViewNameTranslator viewNameTranslator;  /\*\* flashmap +manager：springmvc中运行重定向携带数据的功能\*/  **private** FlashMapManager flashMapManager;  /\*\* 视图解析器\*/  **private** List<ViewResolver> viewResolvers; |

dispatcherServlet九大组件初始化的地方

|  |
| --- |
| **protected** **void** initStrategies(ApplicationContext context) {  initMultipartResolver(context);  initLocaleResolver(context);  initThemeResolver(context);  initHandlerMappings(context);  initHandlerAdapters(context);  initHandlerExceptionResolvers(context);  initRequestToViewNameTranslator(context);  initViewResolvers(context);  initFlashMapManager(context);  } |

model and view

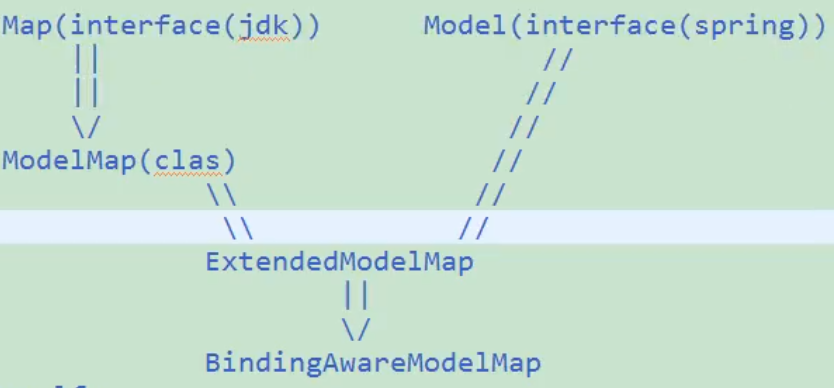
### 方式一

可以在方祛处传入Map、或者Mode1或者ModelMap。给这些参数里面保存的所有数据都会放在域中。可以在页面获取。这些值都会传入请求域中。

Map、Mode1、ModelMap三个接口的实现可都是，最终都是BindingAwareModelMap工作。

class org.springframework.validation.support.BindingAwareModelMap

BindingAwareModelMap绑定的内容都会放在请求与中。



|  |
| --- |
| @RequestMapping("/hello")  **public** String hello(Map<String , String> map) {  map.put("name", "lee");  **return** "index";  } |

model

|  |
| --- |
| @RequestMapping("/hello")  **public** String tesModel(Model model) {  model.addAttribute("gender", "nv");  **return** "index";  } |

### modelAndView

方法的返回值可以变为ModelAndView类型；既包含视图信息（页面地址）也包含模型数据（给页面带的数据）；而且数据是放在请求域中；

|  |
| --- |
| @RequestMapping("/hello4")  **public** ModelAndView tesModel() {  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.setViewName("index"); //设置返回页面  modelAndView.addObject("name", "sx"); //设置返回的数据  **return** modelAndView;  } |

### session中保存数据

SpringMVC提供了一种可以临时给Session域中保存数据的方式；使用一个注解 @SessionAttributes(只能标在类上）

@SessionAttributes(value="msg" ,type=”String.class”):

value:返回是，model中由名字是msg的都方法session中。

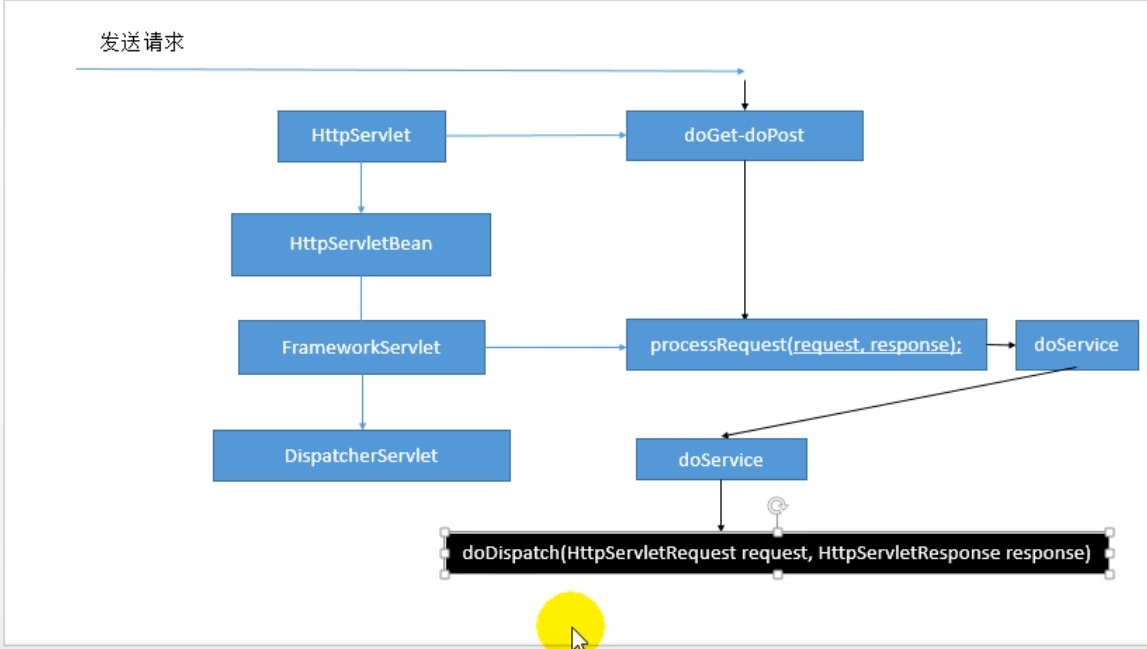
type：返回是，model中的类型是String，添加到session中

给BindingAwareMode1Map中保存的数据，或者是modelandview同时给session中放一份。

|  |
| --- |
| @SessionAttributes(names= {"name","gender"})  @Controller  **public** **class** UserController {    @RequestMapping("/hello4")  **public** ModelAndView tesModel() {  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.setViewName("index");  modelAndView.addObject("name", "sx");  modelAndView.addObject("gender", "male");  **return** modelAndView;  }  } |

## 前端控制器配置：

源码分析



|  |
| --- |
| **protected** **void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  HttpServletRequest processedRequest = request;  HandlerExecutionChain mappedHandler = **null**;  **boolean** multipartRequestParsed = **false**;  WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);  **try** {  ModelAndView mv = **null**;  Exception dispatchException = **null**;  **try** {  //1、检查是否文件上传请求，如果是，对请求包装  processedRequest = checkMultipart(request);  multipartRequestParsed = (processedRequest != request);  //2根据当前的请求，找到那个controller来处理，mappedHandler封装了controler  // Determine handler for the current request.  mappedHandler = getHandler(processedRequest);  **if** (mappedHandler == **null** || mappedHandler.getHandler() == **null**) { //如果没有找到处理器处理这个请求，就404  noHandlerFound(processedRequest, response);  **return**;  }  //拿到能执行这个类所有方法的适配器，相当于反射工具。注解方法适配器。  // Determine handler adapter for the current request.  HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  //获取请求方式  // Process last-modified header, if supported by the handler.  String method = request.getMethod();  **boolean** isGet = "GET".equals(method);  **if** (isGet || "HEAD".equals(method)) {  **long** lastModified = ha.getLastModified(request, mappedHandler.getHandler());  **if** (logger.isDebugEnabled()) {  logger.debug("Last-Modified value for [" + *getRequestUri*(request) + "] is: " + lastModified);  }  **if** (**new** ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {  **return**;  }  }    **if** (!mappedHandler.applyPreHandle(processedRequest, response)) {  **return**;  }  // 控制器(cotnroller)（handler）的方法被调用  ha：适配器，使用适配器执行方法，将执行完后的返回值作为视图名，设置保存到modelandview中，无论目标怎么写，最终适配器都会将适配器的信息封装到modelandview  mv = ha.handle(processedRequest, response, mappedHandler.getHandler());  **if** (asyncManager.isConcurrentHandlingStarted()) {  **return**;  }  applyDefaultViewName(processedRequest, mv);  mappedHandler.applyPostHandle(processedRequest, response, mv);  }  **catch** (Exception ex) {  dispatchException = ex;  }  //这个方法转发到页面，  //6、根据方法最终执行完成后封装的modelandview；转发到对应页面，而且modelandview可以从请求域中获取。  processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);  }  **catch** (Exception ex) {  triggerAfterCompletion(processedRequest, response, mappedHandler, ex);  }  **catch** (Error err) {  triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);  }  **finally** {  **if** (asyncManager.isConcurrentHandlingStarted()) {  // Instead of postHandle and afterCompletion  **if** (mappedHandler != **null**) {  mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);  }  }  **else** {  // Clean up any resources used by a multipart request.  **if** (multipartRequestParsed) {  cleanupMultipart(processedRequest);  }  }  }  } |

1）、所有请求过来DispatcherServlet收到请求，

2）、调用doDispatch（）方法进行处理

1）、getHandler()：根据当前请求地址找到能处理这个请求的目标处理器类（处理器）

2）、getHandlerAdapter()：根据当前处理器类获取到能执行这个处理器方法的适配器；

3）、使用刚才获取到的适配器（AnnotationMethodHandlerAdapter)执行目标方法；

4）、目标方法执行后会返回一个ModelAndView对象

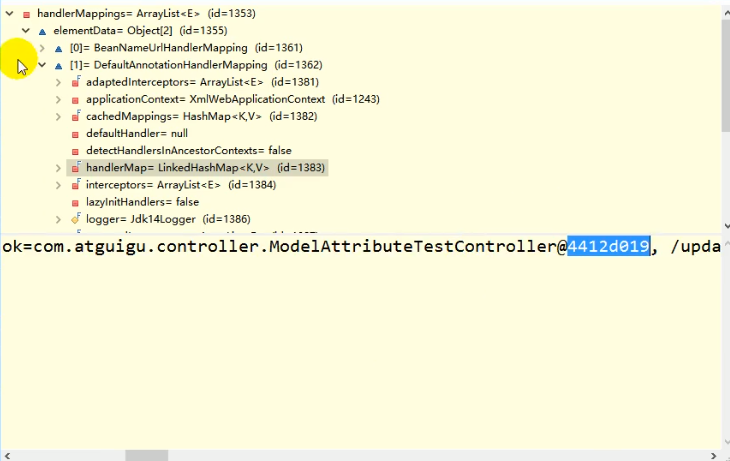
5）、根据Mode1Andview的信息转发到具体的页面，并可以在请求域中取出ModelAndView中的模型数据

### 找到处理器(controller)

怎么根据当前请求就能找到哪个类能来处理,getHandler会返回目标处理器类的执行链；

|  |
| --- |
| **protected** HandlerExecutionChain getHandler(HttpServletRequest request) **throws** Exception {// handlerMappings中保存请求与对应能处理的controller  **for** (HandlerMapping hm : **this**.handlerMappings) {  **if** (logger.isTraceEnabled()) {  logger.trace(  "Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");  }  HandlerExecutionChain handler = hm.getHandler(request);  **if** (handler != **null**) {  **return** handler;  }  }  **return** **null**;  } |

handlerMap:ioc容器启动创建Controller对象的时候扫描每个处理器都能处理什么请求，保存在HandlerMapping的handlerMap属性中；下一次请求过来，就来看哪个HandlerMapping中有这个请求映射信息就行了；注解版和bean版

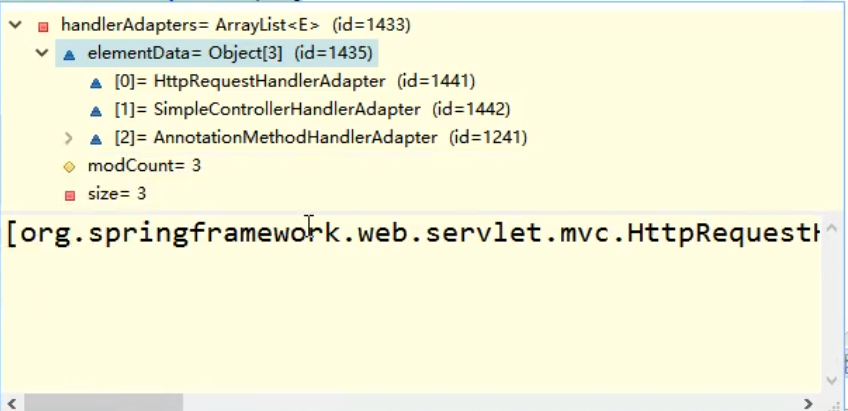


### 找到适配器

适配器目的:让适配器执行目标方法。

|  |
| --- |
| **protected** HandlerAdapter getHandlerAdapter(Object handler) **throws** ServletException {  **for** (HandlerAdapter ha : **this**.handlerAdapters) {  **if** (logger.isTraceEnabled()) {  logger.trace("Testing handler adapter [" + ha + "]");  }  **if** (ha.supports(handler)) {  **return** ha;  }  }  **throw** **new** ServletException("No adapter for handler [" + handler +  "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports this handler");  } |

handlerAdapters：使用注解方法的适配器。适配器就是一个反射工具。解析注解方法的适配器。



AnnotationMethodHandlerAdapter:能解析注解方法的适配器；处理器类中只要有标了注解的这些方法就能用；

### 执行方法

|  |
| --- |
| **public** String updateBook(@RequestParam(value="author")String author,Map<String , String > map,  HttpServletRequest request,@ModelAttribute("hah")User use) {    } |

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

执行该方法return invokeHandlerMethod(request, response, handler);

|  |
| --- |
| **protected** ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  //拿到方法的解析器  ServletHandlerMethodResolver methodResolver = ~~getMethodResolver~~(handler);  //方法解析器根据当前请求地址找到怕正正执行的方法  Method handlerMethod = methodResolver.resolveHandlerMethod(request);  //创建方法执行器  ServletHandlerMethodInvoker methodInvoker = **new** ServletHandlerMethodInvoker(methodResolver);  //包装原生的request和response  ServletWebRequest webRequest = **new** ServletWebRequest(request, response);  //创建了一个隐含模型，BindingAwareModelMap是map下面的。  ExtendedModelMap implicitModel = **new** BindingAwareModelMap();  //正正执行目标方法的，目标方法利用反射执行期间确定参数值，提前执行modelAttribute等所有的操作都在这个里面  Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler, webRequest, implicitModel);  ModelAndView mav =  methodInvoker.getModelAndView(handlerMethod, handler.getClass(), result, implicitModel, webRequest);  methodInvoker.updateModelAttributes(handler, (mav != **null** ? mav.getModel() : **null**), implicitModel, webRequest);  **return** mav;  } |

对方的

|  |
| --- |
| **public** **final** Object invokeHandlerMethod(Method handlerMethod, Object handler,  NativeWebRequest webRequest, ExtendedModelMap implicitModel) **throws** Exception {  Method handlerMethodToInvoke = BridgeMethodResolver.*findBridgedMethod*(handlerMethod);  **try** {  **boolean** debug = ***logger***.isDebugEnabled();  //1、如果sessionAttribute中有值，会放到隐含模型中  **for** (String attrName : **this**.methodResolver.getActualSessionAttributeNames()) {  Object attrValue = **this**.sessionAttributeStore.retrieveAttribute(webRequest, attrName);  **if** (attrValue != **null**) {  implicitModel.addAttribute(attrName, attrValue);  }  }  **for** (Method attributeMethod : **this**.methodResolver.getModelAttributeMethods()) {  Method attributeMethodToInvoke = BridgeMethodResolver.*findBridgedMethod*(attributeMethod);  Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler, webRequest, implicitModel);  **if** (debug) {  ***logger***.debug("Invoking model attribute method: " + attributeMethodToInvoke);  }  String attrName = AnnotationUtils.*findAnnotation*(attributeMethod, ModelAttribute.**class**).value();  **if** (!"".equals(attrName) && implicitModel.containsAttribute(attrName)) {  **continue**;  }  ReflectionUtils.*makeAccessible*(attributeMethodToInvoke);  Object attrValue = attributeMethodToInvoke.invoke(handler, args);  **if** ("".equals(attrName)) {  Class<?> resolvedType = GenericTypeResolver.*resolveReturnType*(attributeMethodToInvoke, handler.getClass());  attrName = Conventions.*getVariableNameForReturnType*(attributeMethodToInvoke, resolvedType, attrValue);  }  **if** (!implicitModel.containsAttribute(attrName)) {  implicitModel.addAttribute(attrName, attrValue);  }  }  Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler, webRequest, implicitModel);  **if** (debug) {  ***logger***.debug("Invoking request handler method: " + handlerMethodToInvoke);  }  ReflectionUtils.*makeAccessible*(handlerMethodToInvoke);  **return** handlerMethodToInvoke.invoke(handler, args);  }  **catch** (IllegalStateException ex) {  // Internal assertion failed (e.g. invalid signature):  // throw exception with full handler method context...  **throw** **new** HandlerMethodInvocationException(handlerMethodToInvoke, ex);  }  **catch** (InvocationTargetException ex) {  // User-defined @ModelAttribute/@InitBinder/@RequestMapping method threw an exception...  ReflectionUtils.*rethrowException*(ex.getTargetException());  **return** **null**;  }  } |

## 处理器映射器：

非注解处理器映射器（了解）

注解的处理器映射器（掌握）

对标记@Controller类中标识有@RequestMapping的方法进行映射。在@RequestMapping里边定义映射的url。使用注解的映射器不用在xml中配置url和Handler的映射关系。

## 处理器适配器：

非注解处理器适配器（了解）

注解的处理器适配器（掌握）

注解处理器适配器和注解的处理器映射器是配对使用。理解为不能使用非注解映射器进行映射。

<mvc:annotation-driven></mvc:annotation-driven>可以代替下边的配置：

<!--注解映射器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>

<!--注解适配器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

实际开发使用：mvc:annotation-driven

## 开发中的springmvc.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入头文件 -->  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  <!-- 扫描 -->  <context:component-scan base-package=*"com.sxt.controller"*></context:component-scan>  <!-- 适配器和控制器的配置 -->  <mvc:annotation-driven></mvc:annotation-driven>  <!-- 配置视图解析器 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  </bean>  <!-- 文件上传的配置 -->      <!-- 拦截器的配置 -->    <!-- 静态文件过滤 -->  <mvc:default-servlet-handler/>  </beans> |

# web.xml

## 配置前端控制器web.xml

|  |
| --- |
| <!—1、配置springmvc的前端控制器开始 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servletclass>  org.springframework.web.servlet.DispatcherServlet  </servlet-class>  <!—通过contextConfigLocation配置springmvc需要加载的配置文件  配置文件中包含映射器和适配器 -->  <如果不配置contextConfigLocation，默认加载的是/WEB-INF/serlvet名称-servlet.xml（springmvc-servlet.xml）>  <init-param>//<!—2、创建对象之后，立即加载spring的配置文件，配置文件包含：处理器映射器、处理器配置器、视图解析器、拦截器等配置。  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <!—1、配置当tomcat启动时就创建 DispatcherServlet对象 -->  <load-on-startup>1</load-on-startup>  </servlet>  <!—3、当有请求访问时，会先执行拦截器中的配置，然后在执行对应handler，然后在执行视图解析器。-->  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!-- 所有以.action结局的请求都 交给 DispatcherServlet进行解析。 -->  <url-pattern>\*.action</url-pattern>  <!—所有的请求都交给DispatcherServlet解析，静态文件需要配置不让DispatcherServlet解析，使用此种方式可以实现RESTful的风格url  <url-pattern>/</url-pattern>  3、这样的配置不对，使用这种配置，最终要转发到一个jsp页面，任然后由DispatcherServlet解析jsp不能根据jsp页面找到handler，会报错。  <!-- <url-pattern>/\*</url-pattern> -->  </servlet-mapping>  <!-- 配置springmvc的前端控制器结束 --> |

### url-pattern路径问题

/\*和/都是拦截所有请求；

/ :会拦截所有请求，但是不会拦截\*.jsp；能保证jsp访问正常；

/\* :的范围更大；还会拦截到\*.Jsp这些请求；一但拦截Jsp页面就不能显示了；

为什么不会拦截jsp页面？

因为处理\*.jsp是tomcat做的事，所有项目的小web.xml都是继承于大web.xml。

这个大的web.xml是tomcat服务器中的web.xml。默认的处理器是DefaultServlet。

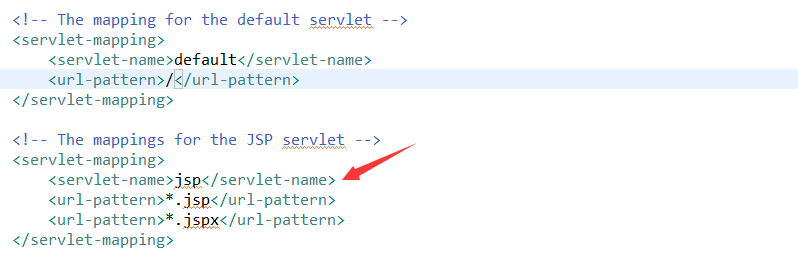
DefaultServlet是Tomcat中处理静态资源的?

除过Jsp，和servlet外剩下的都是静态资源；

index.html：静态资源，tomcat就会在服务器下找到这个资源并返回；

我们前端控制器的使用dispatcherservet,禁用了tomcat服务器中的DefaultServlet。所以无法访问到静态资源。

为什么已经使DispatcherServlet又能访问jsp；因为我们没有覆盖服务器中的JspServlet的配置。



服务器的大web.xml中有一个DefaultServlet是url-pattern=/。我们的配置中前端控制器url-pattern=/，静态资源会来到DispatcherServlet（前端控制器）看那个方法的RequestMapping是这个index，

/\* 将直接拦截所有的请。

### 不指定配置文件位置

如果不指定也会默认去找一个文件；

/WEB-INF/springDispatcherServlet-servlet.xml

可以在web应用的/WEB-INF下创建一个名叫 前端控制器名-servlet.xml。

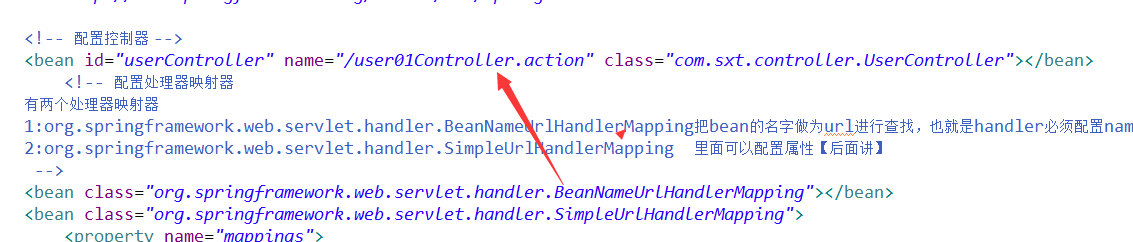
# springmvc.xml详解

## 非注解映射器和适配器

### 映射器

1：org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping

使用controller配置的name属性值作为请求的url-pattern



2：org.springframework.web.servlet.handler.SimpleUrlHandlerMapping

可以使用多个url指向同一个Controller



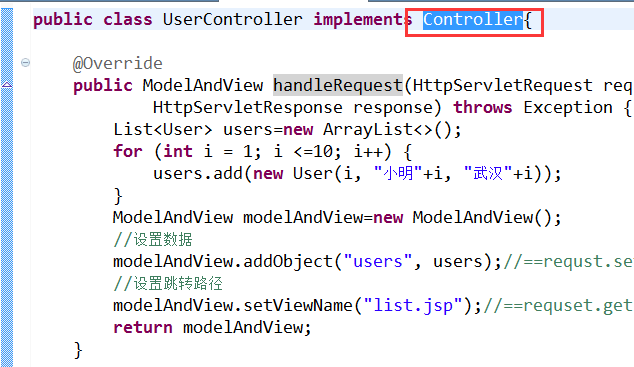
总结

以上两个映射可以共存

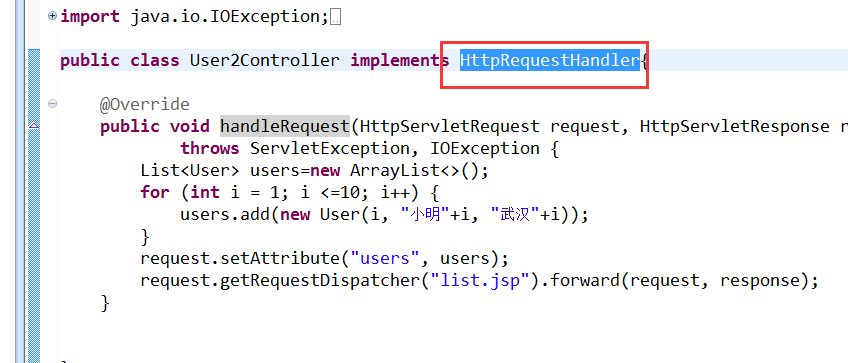
### 适配器

1:org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter

写的controller要实现Controller的接口



2:org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter 写controler要实现HttpRequestHandler接口



总结

两个适配器可以共存

## 注解的映射器和适配器

### 映射器

在spring3.1之前使用org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping注解映射器。

在spring3.1之后使用org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping注解映射器。



### 适配器

在spring3.1之前使用org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter注解适配器。

在spring3.1之后使用org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter注解适配器



### 综合配置



### 注解总结

@Controller 让spring在IOC容器里面创建对象

@RequestMapping("user")

|--作用在类上 给该类方法里面的所有请求加上前缀

|--作用在方法上 该类里面的该方法的请求地址

注意URL不能重置[不能使用相同的URL指向不同的方法]

如下不合法

## 视图解析器

作用：controller中的返回值可以进行拼接。

|  |
| --- |
| <bean id = *"InternalResourceViewResolver"* class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/views/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean> |



# 数据转化

# Controller的参数绑定

Spring MVC提供了诸多的注解来解析参数，其目的在于把控制器从复杂的ServletAPI

中剥离，这样就可以在非Web容器环境中重用控制器，也方便测试人员对其进行有效测试。

## @requestMapping

总结：

1、

### @RequestMapping 简介

在Spring MVC 中使用 @RequestMapping 来映射请求，也就是通过它来指定控制器可以处理哪些URL请求，相当于Servlet中在web.xml中配置。与xml中的映射作用一致。让我们先看一下RequestMapping注解类的源码

|  |
| --- |
| <servlet>  <servlet-name>servletName</servlet-name>  <servlet-class>ServletClass</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>servletName</servlet-name>  <url-pattern>url</url-pattern>  </servlet-mapping> |

源码

|  |
| --- |
| @Target({ElementType.METHOD, ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Mapping  public @interface RequestMapping {  String name() default "";  String[] value() default {};  String[] path() default {};  RequestMethod[] method() default {};  String[] params() default {};  String[] headers() default {};  String[] consumes() default {};  String[] produces() default {};  } |

1）在@Target中有两个属性，分别为 ElementType.METHOD 和 ElementType.TYPE ，也就是说 @RequestMapping 可以在方法和类的声明中使用

2）可以看到注解中的属性除了 name() 返回的字符串，其它的方法均返回数组，也就是可以定义多个属性值，例如 value() 和 path() 都可以同时定义多个字符串值来接收多个URL请求

### 作用

1、url映射

2、窄化请求映射：为了对url进行分类管理，可以在这里定义根路径，最终访问url是根路径+子路径。

3、限制http请求方法：处于一些安全的考虑，对请求的方式进行方法的限制。

### value和path 属性

@RequestMapping 中的 value 和 path 属性（这两个属性作用相同，可以互换，如果仅有这一个属性，则可以省略，下面两个例子均采用省略的方式）

1)将@RequestMapping 注解在login 方法上，而UserController上不添加

|  |
| --- |
| @Controller  public class UserController {  @RequestMapping("/login")  public String login() {  return "success";  }  } |

这时的方法 login() 能处理的 URL 请求路径是基于 Web 应用的，也就是 http://localhost/SpringMVC/login，也就是 index.jsp 页面中的 User Login 链接地址应该是：

|  |
| --- |
| <a href="login">User Login</a> |

2）将 @RequestMapping 注解在 UserController 类上，这时类的注解是相对于 Web 根目录，而方法上的是相对于类上的路径。

|  |
| --- |
| @Controller  @RequestMapping("/user")  public class UserController {  @RequestMapping("/login")  public String login() {  return "success";  }  } |

这时的方法login()能处理的 URL 请求路径则是 http://localhost/SpringMVC/user/login，也就是 index.jsp 页面中的 User Login 链接地址应该是

|  |
| --- |
| <a href="user/login">User Login</a> |

### method 属性

1）简介：@RequestMapping 中的 method 主要用来定义接收浏览器发来的何种请求。在Spring中，使用枚举类

org.springframework.web.bind.annotation.RequestMethod来定义浏览器请求的方式。

Http规范定义了多种请求资源的方式，最基本的有四种，分别为：GET（查）、POST（增）、PUT（改）、DELETE（删），而URL则用于定位网络上的资源相当于地址的作用，配合四种请求方式，可以实现对URL对应的资源的增删改查操作。

在实际应用中，很多人并没有按照这个规范做，因为使用GET/POST同样可以完成PUT和DELETE操作，甚至GET也可以完成POST操作，因为GET不需要用到表单，而POST却需要通过表单来发送。

2）通过@RequestMapping(value="/login",method=RequestMethod.GET) 来指定 login()方法 仅处理通过 GET 方式发来的请求

|  |
| --- |
| @Controller  @RequestMapping(path ="/user")  public class UserController {  @RequestMapping(path = "/login", method=RequestMethod.GET)  public String login() {  return "success";  }  } |

这时，如果浏览器发来的请求不是GET的话，将收到浏览器返回的错误提示，也就是得通过链接的方式而不是表单的方式：

<a href="user/login>User Login</a>

3、通过@RequestMapping(value="/login",method=RequestMethod.POST) 来指定 login()方法仅处理通过 POST 方式发来的请求

|  |
| --- |
| @Controller  @RequestMapping(path = "/user")  public class UserController {  @RequestMapping(path = "/login", method=RequestMethod.POST)  public String login() {  return "success";  }  } |

这时，必须通过表单的方式发送请求，否则将收到浏览器返回的错误提示

|  |
| --- |
| <form action="user/login" method="post">  <input type="submit" value="使用Post发送请求"/>  </form> |

4)由于在 RequestMapping 注解类中 method() 方法返回的是 RequestMethod 数组，所以可以给 method 同时指定多个请求方式，例如：

|  |
| --- |
| @Controller  @RequestMapping(path = "/user")  public class UserController {  // 该方法将同时接收通过GET和POST方式发来的请求  @RequestMapping(path = "/login", method={RequestMethod.POST,RequestMethod.GET})  public String login() {  return "success";  }  } |

### params

作用：规格请求参数

param1：表示请求必须包含名为param1的请求参数

eg:@RequestMaping(name=“hello”,params={"username"})

发送请求的时候必须带上一个名为username的参数；没带都会404

! param1: 表示请求必须不包含名为param1的请求参数

eg:@RequestMaping(name=“hello”, params={!"username"})

发送请求的时候必须不带上一个名为username的参数

eg:@RequestMaping(name=“hello”, params={"username!=sx"})

发送请求可以不带一个名为username的参数，或者=带了username参数，但username的参数值必须不为sx。

测试 @RequestMapping 的 params 属性，该属性表示请求参数，也就是追加在URL上的键值对，多个请求参数以&隔开，例如：

<http://localhost/SpringMVC/user/login?username=kolbe&password=123456>

则这个请求的参数为username=kolbe以及password=123456，@RequestMapping 中可以使用 params 来限制请求参数，来实现进一步的过滤请求，举个例子：

|  |
| --- |
| @Controller  @RequestMapping(path = "/user")  public class UserController {  // 该方法将接收 /user/login 发来的请求，且请求参数必须为 username=kolbe&password=123456  @RequestMapping(path = "/login", params={"username=kolbe","password=123456"})  public String login() {  return "success";  }  } |

该例中则表示 UserController 中的 login() 方法仅处理 /user/login 发来的请求，且必须带有 username=kolbe&password=123456 的请求参数，否则浏览器将返回HTTP 404的错误，对应 index.jsp 中的链接

|  |
| --- |
| <a href="user/login?username=kolbe&password=123456">User Login</a> |

### header参数

headers={}，请求头中的任意字段都可以设置。

1、该请求只能谷歌浏览器访问，其他浏览器无法访问

|  |
| --- |
| @RequestMapping(name="/hello",headers= {"User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.12 Safari/537.36"}) |

### consumes和produces

consumes：只接受内容类型是哪种的请求，规定请求头中的Content-Type

produces：告诉浏览器返回的内容类型是什么，给响应头中加上Content-Type:

### PathVariable带占位符的URL

PathVariable：通过@PathVariable可以捋URL中占位符参数绑定到控制器处理方法的人参中：

（一）带占位符的URL是Spring 3.0 新增的功能，可以通过 @PathVariable 将 URL 中的占位符绑定到控制器的处理方法的参数中，占位符使用{}括起来

使用方法：

1）带占位符的URL示例：

|  |
| --- |
| @Controller  @RequestMapping(path = "/user")  public class UserController {  @RequestMapping(value="/{id}", method=RequestMethod.GET)  public String show(@PathVariable("id") Integer id) {  return "success";  }  } |

在这个控制器中 show() 方法将可以接收 user/1、user/2、user/3等等的路径请求，请求的方法必须为GET，使用 @PathVariable 为应用实现 REST 规范提供了具大的便利条件。

### requestmapping的模糊匹配

URL地址可以写模糊的通配符：

？：能替代任意一个字符，

@RequestMapping(name="/ hello?")

例如请求: hello1、hello2、hello3

注：如果精确和模糊多个匹配情况下，使用精确匹配.

\*：能替代任意多个字符，和一层路径

@RequestMapping(name="/a\*/hello")

例如请求：/abc/hello、/abcs/hello

匹配一次路径

@RequestMapping(name="/a/\*/hello")

例如：/abc/a/hello、/abc/b/hello

注：不能匹配多层路径。/abc/b/c/hello

\*\*：能替代多层路径

@RequestMapping(name="/a/\*\*/hello")

例如：/abc/b/c/hello

## 绑定的过程

处理器适配器调用springmvc提供参数绑定组件将key/value数据转成controller方法的形参参数绑定组件：在spirngmvc.早期版本使用PropertyEditor(只能将字符串传成java对象）后期使用converter（进行任意类型的传换）spirngmvc 提供了很多converter（转换器）

在特殊情况下需要自定义converter。对日期数据绑定需要自定义converterl。

## ModelAndView

从客户端请求key/value数据，经过参数绑定，将key/value的数据绑定到controller的方法的形参上。在springmvc中，接收页面提交的数据是通过方法的形参接收的。而不是controller类定义成员变量接收。

## 默认参数绑定

直接在controller方法形参上定义下边类型的对象，就可以使用这些对象。在参数绑定过程中，如果遇到下边类型。

HttpServletRequest

HttpServletResponse

HttpSession

Model：model是将数据填充到request域中。

|  |
| --- |
| /\*\*  \* 默认参数绑定  \*/  @RequestMapping("addUser4")  **public** ModelAndView addUser4(HttpServletRequest request,HttpServletResponse response,HttpSession session){  System.*out*.println("request:"+request);  System.*out*.println("response:"+response);  System.*out*.println("session:"+session);  System.*out*.println("session-request:"+request.getSession());  System.*out*.println("servletContext-request:"+request.getServletContext());  System.*out*.println("servletContext-session:"+session.getServletContext());  **return** **new** ModelAndView("../success.jsp");  } |

## 普通类型参数绑定

Spring MVC目前也比较智能化，如果传递过来的参数名称和HTTP的保存一致，那么

须任何注解也可以获取参数。

获取参数：通过参数名称和HTTP请求参数的名称保持一致来获取参数，如果不一致是没法获取到的，这样的方式允许参数为空。

### jsp

|  |
| --- |
| <h2>简单类型</h2>  <form action=*"user/addUser1.action"* method=*"post"*>  用户名:<input type=*"text"* name=*"username"*><br>  年龄:<input type=*"text"* name=*"age"*><br>  地址<input type=*"text"* name=*"address"*><br>  <input type=*"submit"* value=*"提交"*>  </form> |

### controller

|  |
| --- |
| /\*\*  \* 添加数据 简单类型的数据  \*/  @RequestMapping("addUser1")  **public** ModelAndView addUser1(String username,Integer age,String address){  System.*out*.println(username);  System.*out*.println(age);  System.*out*.println(address);  **return** **new** ModelAndView("../success.jsp");  } |

### UserController



## 集合类型参数绑定

### index.jsp

|  |
| --- |
| <h2>集合类型</h2>  <form action=*"user/addUser2.action"* method=*"post"*>  用户名:<input type=*"text"* name=*"username"*><br>  爱好:<input type=*"checkbox"* name=*"hobby"* value=*"LOL"*>LOL  <input type=*"checkbox"* name=*"hobby"* value=*"DNF"*>DNF  <input type=*"checkbox"* name=*"hobby"* value=*"CF"*>CF  <input type=*"checkbox"* name=*"hobby"* value=*"DOTA"*>DOTA<br>  <input type=*"submit"* value=*"提交"*>  </form> |

### controller

|  |
| --- |
| /\*\*  \* 集合类型的数据  \*/  @RequestMapping("addUser2")//使用数据来获取表单的值。  **public** ModelAndView addUser2(String[] hobby,String username){  System.*out*.println(Arrays.*toString*(hobby));  **return** **new** ModelAndView("../success.jsp");  } |

## POJO类型参数绑定(重点)

1、将POJO中的每一个属性，从request参数中取出来，并封装。

2、还可以级联封装，属性的属性。

3、请求参数的参数名和对象当中的属性名一一对应。

### user类

|  |
| --- |
| **public** **class** User {  **private** Integer id;  **private** String name;  **private** String address;  } |

### index.jsp

|  |
| --- |
| <h2>对象类型</h2>  <form action=*"user/addUser3.action"* method=*"post"*>  ID:<input type=*"text"* name=*"id"*><br>  用户名:<input type=*"text"* name=*"name"*><br>  地址<input type=*"text"* name=*"address"*><br>  <input type=*"submit"* value=*"提交"*>  </form> |

### controller

|  |
| --- |
| \* 对象类型绑定  @RequestMapping("addUser3")  **public** ModelAndView addUser3(User user){//这里可以接收到页面传过来的参数。  System.*out*.println(user);//调用参数。  **return** **new** ModelAndView("../success.jsp");  } |

## @RequestParam获取请求参数信息

如果前端的参数命名规则和后台的不一致，例如前台穿过来的为name，而后台为uname，springMVC提供了注解来实现这种转换。

### 默认情况

直接给方法入参上写一个和请求参数名相同的变量。这个变量就来接收请求参数的值；

如果链接有值，则会映射到参数上。

如果链接没有值，则会为null

### 使用RequestParam

@RequestParam：获取请求参数的注解，默认请求参数是必带的

value:指定要获取参数的key

required:这个参数是否是必须的。

defaultvalue:参数的默认值，没带时为null

index.jsp

|  |
| --- |
| <h2>对象类型</h2>  <form action=*"user/addUser3.action"* method=*"post"*>  用户名:<input type=*"text"* name=*"name"*><br>  <input type=*"submit"* value=*"提交"*>  </form> |

controller

|  |
| --- |
| \* 对象类型绑定  @RequestMapping("addUser3") //表示该值是从请求参数中获取  **public** ModelAndView addUser3(@RequestParam (“*name*”)Stirng uname){//这里可以接收到页面传过来的参数。  System.*out*.println(uname);//调用参数。  **return** **new** ModelAndView("../success.jsp");  } |

### @RequestHeader

@RequestHeader:获取请求头中某个key的值，如果请求头中有这个，将会报错。

value:指定要获取参数的key

required:这个参数是否是必须的。

defaultvalue:参数的默认值，没带时为null

|  |
| --- |
| @RequestMapping("get")  **public** String getHeader(@RequestHeader(name="User-Agent")String name) {  System.***out***.println(name);  **return** "hello";  } |

结果: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Win64; x64; Trident/7.0)

### @CookieValue

@CookieValue:获取某个cookie值

name:cookie的名字

required:这个参数是否是必须的。

defaultvalue:参数的默认值，没带时为null

|  |
| --- |
| @RequestMapping("getCookie")  **public** String getCookie(@RequestHeader(name="User-Agent")String name  ,@CookieValue("JSESSIONID")String cookie) {  System.***out***.println(cookie);  **return** "hello";  } |

结果;03DEBFEE580D7897F544C40CBACE3803

## restful风格的URL参数绑定

RESTful架构风格规定，数据的元操作，即CRUD(create, read, update和delete,即数据的增删查改)操作，分别对应于HTTP方法：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源，这样就统一了数据操作的接口，仅通过HTTP方法，就可以完成对数据的所有增删查改工作。

GET（SELECT）：从服务器取出资源（一项或多项）。

POST（CREATE）：在服务器新建一个资源。

PUT（UPDATE）：在服务器更新资源（客户端提供完整资源数据）。

PATCH（UPDATE）：在服务器更新资源（客户端提供需要修改的资源数据）。

DELETE（DELETE）：从服务器删除资源。

### 使用get请求来添加一个用户

jsp

|  |
| --- |
| <input type="button" id="btn2" value="sayHelloTo小明">  $("#btn2").on("click",**function**(){  $.ajax({  url:'user/sayHello/小明',//  type:'get', //GET  async:**true**, //或false,是否异步  //dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:**function**(data){  alert(data);  }  });  }); |

controller

使用get请求方式来获取资源

|  |
| --- |
| @RequestMapping(value={"sayHello/{name}"},method={RequestMethod.GET},produces="text/html;charset=utf-8")  @ResponseBody  **@PathVariable**("name")就是把value={"sayHello/{name}"的name的值给形参n  **@param** name  **@return**  **public** String sayHello2(@PathVariable("name")String name){  System.*out*.println(name);  **return** "你好:"+name;  }  注：也可以使用注解GetMapping代替上面的注解，相当于设置映射方式值接收get请求方式。  @GetMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8") |

### Post请求方式添加

jsp

|  |
| --- |
| $("#btn5").on("click",function(){  $.ajax({  url:'user/user/11/小明11/武汉11',//这里传递多个参数给controller  type:'post', //使用post方式  async:true, //或false,是否异步  dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:function(data){  alert(data.msg);  }  });  }); |

controller

|  |
| --- |
| \* 添加一个用户  @PostMapping(value="user/{id}/{name}/{address}")  //注解标明,只有请求为post方式才能访问到该controller，请求接收出来的多个参数。  @ResponseBody  public Map<String,Object> addUser1(@PathVariable("id")Integer id,  @PathVariable("name")String name,  @PathVariable("address") String address){  //这里使用@PathVariable()表名url中参数的名字。  User user=new User(id, name, address);  users.add(user);  print();  Map<String,Object> map=new HashMap<>();  map.put("msg", "添加成功");  return map;  } |

### delete请求方式删除一个资源

jsp

|  |
| --- |
| <input type="button" id="btn4" value="使用delete删除id=1的用户">  $("#btn4").on("click",**function**(){  $.ajax({  <url:'user/user/1>', //使用url传递参数  type:'delete', //使用delete请求方式  async:**true**, //或false,是否异步  dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:**function**(data){  alert(data.msg);  }  });  }); |

controller

|  |
| --- |
| 根据ID查询一个用户  @DeleteMapping(value="user/{id}") //指定method的方式是delete方式才能访问到该资源。  @ResponseBody  **public** Map<String,Object> deleteUser(@PathVariable("id")Integer id){  **for** (User u : *users*) {  **if**(u.getId()==id){  *users*.remove(u);  **break**;  }  }  print();  Map<String,Object> map=**new** HashMap<>();  map.put("msg", "删除成功");  **return** map;  } |

### post方式（采用json数据格式）

jsp

|  |
| --- |
| <input type="button" id="btn6" value="使用post添加用户2">  $("#btn6").on("click",function(){  $.ajax({  url:'user/user',  type:'post', //POST  async:true,  data:{//这里使用json传递数据  id:11,  name:'小明11',  address:'武汉11'  },  dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:function(data){  alert(data.msg);  }  });  }); |

controller

|  |
| --- |
| 添加一个用户  @PostMapping(value="user/{id}/{name}/{address}")//虽然传值的方式不同，但是接收值的方式相同  @ResponseBody  public Map<String,Object> addUser1(@PathVariable("id")Integer id,@PathVariable("name")String name,@PathVariable("address")String address){  User user=new User(id, name, address);  users.add(user);  print();  Map<String,Object> map=new HashMap<>();  map.put("msg", "添加成功");  return map;  } |

也可以使用对象作为形参接收json传来的参数

|  |
| --- |
| \* 添加一个用户  @PostMapping(value="user")//这里也不需要写参数  @ResponseBody  public Map<String,Object> addUser2(User user){//使用带对象类型来接收json穿过来的参数。底层会自动将键值对映射为对象。  users.add(user);  print();  Map<String,Object> map=new HashMap<>();  map.put("msg", "添加成功");  return map;  } |

### 使用Put请求方式来更新资源

jsp

|  |
| --- |
| <input type="button" id="btn7" value="使用put更新用户1">  $("#btn7").on("click",function(){  $.ajax({  url:'user/user/1/习大大/大武汉',//使用url传值  type:'put',  async:true, //或false,是否异步  dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:function(data){  alert(data.msg);  }  });  }); |

controller

|  |
| --- |
| \* 更新一个用户  @PutMapping(value="user")  @ResponseBody  public Map<String,Object> updateUser2(User user){  for (User u : users) {  if(u.getId()==user.getId()){  BeanUtils.copyProperties(user, u);//使用BeanUtils.copyProperties()将一个对象的属性赋值到另一对象当中，spring提供的方法。  break;  }  }  print();  Map<String,Object> map=new HashMap<>();  map.put("msg", "更新成功");  return map;  } |

使用对象接收传的json对象。

|  |
| --- |
| @PutMapping(value="user")  @ResponseBody  public Map<String,Object> updateUser2(User user){//这里使用user接收json数据  for (User u : users) {  if(u.getId()==user.getId()){  BeanUtils.copyProperties(user, u);  break;  }  }  print();  Map<String,Object> map=new HashMap<>();  map.put("msg", "更新成功");  return map;  } |

### PATCH方式完成全查詢

jsp

|  |
| --- |
| <input type="button" id="btn9" value="使用patch全查询用户">  $("#btn9").on("click",function(){  $.ajax({  url:'user/user',  type:'patch', //使用patch方式来提交参数。  async:true, //或false,是否异步  dataType:'json', //返回的数据格式：json/xml/html/script/jsonp/text  success:function(data){  alert(data.length);  }  });  }); |

controller

|  |
| --- |
| 全查询PATCH  @PatchMapping("user")//只有Patch方式才能访问到该资源。  @ResponseBody  public List<User> queryAllUser(){  return users;  } |

### put请求无法提交普通key-value的数据解决方法

|  |
| --- |
| <!-- 配置springmvc对于put请求的keyvalue表单数据的接收 -->  <filter>  <filter-name>HttpPutFormContentFilter</filter-name>  <filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>HttpPutFormContentFilter</filter-name>  <servlet-name>springmvc</servlet-name>  </filter-mapping> |

### 8总结

@PathVariable

|-value 从url里面取出对应的值赋值给形式参数

@GetMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8")

|==@RequestMapping(method=HttpMethod.GET)

@PostMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8")

|==@RequestMapping(method=HttpMethod.POST)

@DeleteMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8")

|==@RequestMapping(method=HttpMethod.Delete)

@PutMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8")

|==@RequestMapping(method=HttpMethod.PUT)

@PatchMapping(value={"sayHello/{name}"},produces="text/html;charset=utf-8")

|==@RequestMapping(method=HttpMethod.PATCH)

@RestControlller

|--作用在类==@Contrller+@ResponseBody

如果一个类使用@ RestControlller做为注解里面的所有方法都只能返回json对象或字符串给页面

## HttpServletRequest的几种其它获取方法

SpringMVC可以直接在参数上写原生API；

HttpServletRequest

HttpServletResponse

HttpSession

java.security.Principal 与http相关的对象

Locale 国际化相关。

Inputstream

OutputStream

Reader

Writer

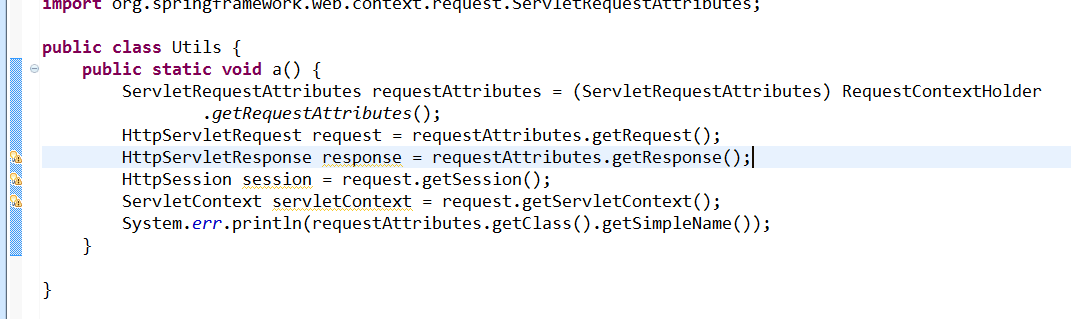
### 使用参数绑定



### 使用注解



### 使用解耦的方法



## 使用ajax传递Json 参数

@ResponseBody这个注解通常使用在控制层（controller）的方法上，其作用是将方法的返回值以特定的格式写入到response的body区域，进而将数据返回给客户端。当方法上面没有写ResponseBody,底层会将方法的返回值封装为ModelAndView对象。

### springmvc返回一个字符串

jsp

|  |
| --- |
| <form action=*"user/addUser1.action"* method=*"post"*>  用户名:<input type=*"text"* name=*"username"* id=*"username"*> <span id=*"usernameSpan"*></span> <br>  <input type=*"submit"* value=*"提交"*>  </form> |

controller

|  |
| --- |
| @Controller  @RequestMapping("user")//作用在类上，相当于在url前面加了一个前缀  **public** **class** UserController{  @RequestMapping(value={"sayHello1","sayHello2"},produces="text/html;charset=utf-8")  @ResponseBody//设定将方法的返回值返回到body中  **public** String sayHello(){  **return** "helloworld 你好";  }  {} |

### springmvc使用ajax返回给一个字符串

jsp

|  |
| --- |
| <form action=*"user/addUser1.action"* method=*"post"*>  用户名:<input type=*"text"* name=*"username"* id=*"username"*> <span id=*"usernameSpan"*></span> <br>  <input type=*"submit"* value=*"提交"*>  </form>  $("#username").blur(**function**(){  **var** username=$(**this**).val();  //使用ajax提交  $.post("user/checkUserName.action",{username:username},**function**(obj){  //alert(obj);  **if**(obj==="false"){  $("#usernameSpan").html("<font color=blue>用户名可用</font>");  }**else**{  $("#usernameSpan").html("<font color=red>用户名不可用</font>");  }  });  }); |

controller

|  |
| --- |
| /\*\*  \* 检查用户名是否存在  \*/  @RequestMapping("checkUserName")  @ResponseBody  **public** String checkUserName(String username){  List<String> names=**new** ArrayList<>();  names.add("admin1");  names.add("admin2");  names.add("admin3");  names.add("admin4");  Boolean flag=**false**;//默认不存在  **for** (String string : names) {  **if**(string.equalsIgnoreCase(username)){  flag=**true**;  **break**;  }  }  **return** String.*valueOf*(flag);  } |

### ajax请求返回一个页面springmvc（即返回一个modeandview）

jsp

|  |
| --- |
| html部分  <input type=*"button"* id=*"getDate"* value=*"请求数据"*>  <div id=*"content"*></div>  js部分  <script type=*"text/javascript"*>  $("#getDate").on("click",**function**(){  $.post("user/queryAllUser.action",**function**(obj){//这里返回一个页面给对象。  $("#content").html(obj);//将页面字符串插入到div中  });  }); |

controller

|  |
| --- |
| @RequestMapping("queryAllUser")  **public** String queryAllUser(Model model){  List<User> users=**new** ArrayList<>();  **for** (**int** i = 1; i <=10; i++) {  users.add(**new** User(i, "小明"+i, "武汉"+i));  }  model.addAttribute("users", users);//使用model添加数据  **return** "list"; //这里返回一个list页面给ajax  } |

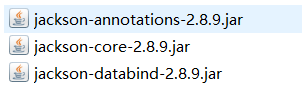
### 返回一个json对象springmvc+fastjson的写法

controller

|  |
| --- |
| 返回一个User对象的字符串给页面  \*/  @RequestMapping(value="getUser",produces="text/html;charset=utf-8")  @ResponseBody  **public** String getUser(){  User user=**new** User(1, "小明", "武汉");  String json=JSON.*toJSONString*(user);  System.*out*.println(json);  **return** json;  }  返回一个user书的字符串给页面  @RequestMapping(value="getUsers",produces="text/html;charset=utf-8")  @ResponseBody  **public** String getUsers(){  List<User> users=**new** ArrayList<>();  **for** (**int** i = 1; i <=10; i++) {  users.add(**new** User(i, "小明"+i, "武汉"+i));  }  //使用fastjson将对象转为json格式的字符串  String json=JSON.*toJSONString*(users);  System.*out*.println(json);  **return** json;  } |

### 返回一个java对象 springmvc+jackson的写法

1、导入jackson的包



2、controller

|  |
| --- |
| /\*\*  \* 返回一个User对象的字符串给页面  \*/  @RequestMapping(value="getUser1")  @ResponseBody  **public** User getUser1(){  User user=**new** User(1, "小明", "武汉");  **return** user;  }  //显示效果  /\*\*  \* 返回一个User对象数组的字符串给页  \*/  @RequestMapping(value="getUsers1")  @ResponseBody  **public** List<User> getUsers1(){  List<User> users=**new** ArrayList<>();  **for** (**int** i = 1; i <=10; i++) {  users.add(**new** User(i, "小明"+i, "武汉"+i));  }  **return** users;  } |

## 重定向

spring MVC框架controller间跳转，需重定向。有几种情况：不带参数跳转，带参数拼接url形式跳转，带参数不拼接参数跳转，页面也能显示。

### 不传递任何的参数

我在后台一个controller跳转到另一个controller，为什么有这种需求呢，是这样的。我有一个列表页面，然后我会进行新增操作，新增在后台完成之后我要跳转到列表页面，不需要传递参数，列表页面默认查询所有的。

方式一：使用ModelAndView

return new ModelAndView("redirect:/toList");

这样可以重定向到toList这个方法

方式二：返回String

return "redirect:/ toList ";

### 重定向带参数拼接方法

列表页面有查询条件，跳转后我的查询条件不能丢掉，这样就需要带参数的了，带参数可以拼接url。

方式一：自己手动拼接url

new ModelAndView("redirect:/toList？param1="+value1+"&param2="+value2); 这样有个弊端，就是传中文可能会有乱码问题。

方式二：用RedirectAttributes，这个是发现的一个比较好用的一个类

这里用它的addAttribute方法，这个实际上重定向过去以后你看url，是它自动给你拼了你的url。

使用方法：

attr.addAttribute("param", value);

return "redirect:/namespace/toController";

这样在toController这个方法中就可以通过获得参数的方式获得这个参数，再传递到页面。过去的url还是和方式一一样的。

### 带参数不拼接url（重点是这个）

一般我估计重定向到都想用这种方式：

|  |
| --- |
| @RequestMapping(“/save”)  **public** String save(@ModelAttribute(“form”) Bean form,RedirectAttributes attr)  **throws** Exception {  String code = service.save(form);  **if**(code.equals(“000”)){  attr.addFlashAttribute(“name”, form.getName());  attr.addFlashAttribute(“success”, “添加成功!”);  **return** “redirect:/index”;  }**else**{  attr.addAttribute(“projectName”, form.getProjectName());  attr.addAttribute(“enviroment”, form.getEnviroment());  attr.addFlashAttribute(“msg”, “添加出错”);  **return** “redirect:/maintenance/toAddConfigCenter”;  }  } |

addFlashAttribute() springMVC3中 该方法将信息放到session中,在页面直接用el表达式就可以获得.session在跳到页面后马上移除对象。所以你刷新一下后这个值就会丢掉。

本质还是两次跳转,spring进行了封装;

## 请求转发

### ViewResolver请求转发

|  |
| --- |
| @RequestMapping(value="/testredirect",method = { RequestMethod.POST, RequestMethod.GET })  public String testredirect(HttpServletResponse response){  return "forward:/index";  } |

带参数

|  |
| --- |
| @RequestMapping(value="/testredirect",method = { RequestMethod.POST, RequestMethod.GET })  public String testredirect(HttpServletRequest request){  request.setAttribute("username", "51gjie"); //把username参数传递到request中  return "forward:/user/index"; //请求转发到页面  } |

### 传统的renquest方式

|  |
| --- |
| request.getRequestDispatcher(" ").forward(request,response);  　response.sendRedirect(" "); |

### 注意点

1、请求转发request是可以共享，在requeust设置setAttribute可以在其他的

## 总结

ModelAndView

|--addObject("users" ,users) 设置数据

|--setViewName("../list.jsp") 设置跳转页面 请求转发

|--|--setViewName("redirect../list.jsp") 重定向页面

String

|--return "list.jsp" 请求转发

|--return " redirect../list.jsp " 请求重定向

void

|--会以方法名类的@RequestMapping("user")的值作为转发的页面

@RequestMapping("user")

public class UserController{

@RequestMapping("addUser")

public void addUser(){}

}

springmvc会找 WebRoot/user/addUser

# 拦截器

## 拦截器概述

## 实现拦截器

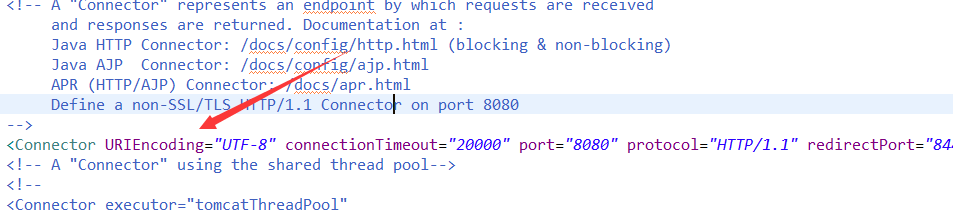
|  |
| --- |
| 1、实现拦截器，首先实现org.springframework. web. servlet.  Hand! er Interceptor，接口。并实现接口中的三个方法  **public** **class** SessionInteceptor **implements** HandlerInterceptor{  /\*\*  \* 当Contrller准备执行时的回调方法  \* 如果返回true代表放行  \* 否则，说明用户没有登陆  \*/  @Override  **public** **boolean** preHandle(HttpServletRequest request,  HttpServletResponse response, Object handler) **throws** Exception {  Object user = request.getSession().getAttribute("user");  System.*out*.println("----preHandle");  **if**(**null**!=user){  **return** **true**;  }**else**{  //跳转到登陆页面  response.sendRedirect("../index.jsp");  **return** **false**;  }  }  /\*\*  \* 进入controller之后，返回modelAndView之前，执行该方法。  \*/  @Override  **public** **void** postHandle(HttpServletRequest request,  HttpServletResponse response, Object handler,  ModelAndView modelAndView) **throws** Exception {  System.*out*.println("----postHandle"+modelAndView.getViewName());  }  /\*\*  \* 当springmvc渲染好页面之后再回调的方法  \*/  @Override  **public** **void** afterCompletion(HttpServletRequest request,  HttpServletResponse response, Object handler, Exception ex)  **throws** Exception {  System.*out*.println("-----afterCompletion");  }  } |

# 中文乱码处理

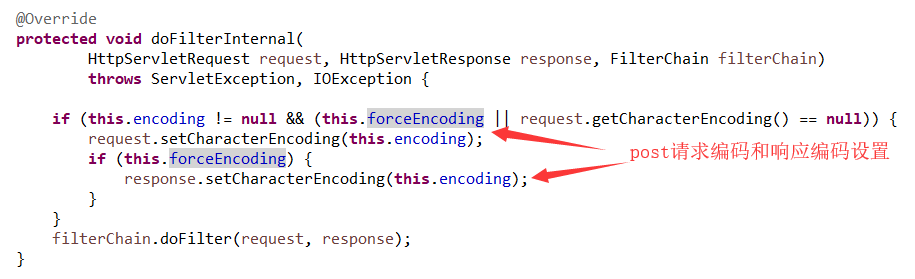
## 请求乱码

### get请求

tomcat的server.xml中设置乱码处理



### POST乱码



使用xml中使用过滤器

|  |
| --- |
| <!--字符编码配置 -->  <filter>  <filter-name>characterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>characterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

## rest风格filter和字符编码的filter

注：字符编码filter，一定要现在其他filter前面！

|  |
| --- |
| <!--字符编码配置 -->  <filter>  <filter-name>characterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>characterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  <!--rest风格的filter -->  <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> |

# 文件上传

serlvet的上传方法

设置form表单的encType

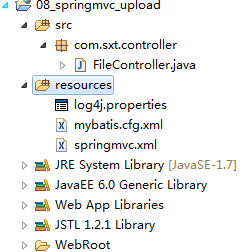
给servlet加注解

Part part=request.getPart("mf");

part.write()

## 准备工作

### 创建项目并加入两个包



https://mvnrepository.com/

http://central.maven.org/maven2/commons-fileupload/commons-fileupload/1.3.3/

http://central.maven.org/maven2/commons-io/commons-io/2.6/



### 配置springmvc对文件上传的支持

|  |
| --- |
| <!-- 文件上传的配置 配置一个二进制的流 注意id的值不能乱写multipartResolver-->  <bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>  <!--指定文件上传的编码 -->  <property name=*"defaultEncoding"* value=*"UTF-8"*></property>  <!-- 临时文件的目录 -->  <property name=*"uploadTempDir"* value=*"/upload/temp"* ></property>  <!-- 设置一次性文件上传的最大值 20M -->  <property name=*"maxUploadSize"* value=*"21474836480"*></property>  </bean> |

## 单文件上传不改名保存到服务器

### 创建index.jsp

|  |
| --- |
| <!-- 上传单个文件 -->  <h1>上传单个文件</h1>  <form action=*"upload/upload01.action"* method=*"post"* enctype=*"multipart/form-data"*>  选择文件：<input type=*"file"* name=*"mf"*><br>  <input type=*"submit"* value=*"上传"*>  </form> |

### 创建FileController

|  |
| --- |
| /\*\*  \* 单文件上传不改名  \*/  @RequestMapping("upload01")  **public** String upload01(MultipartFile mf,HttpServletRequest request){  System.*out*.println("文件类型:"+mf.getContentType());//文件类型  System.*out*.println("得到input标签的name属性值 :"+mf.getName());//得到input标签的name属性值  System.*out*.println("得到文件名:"+mf.getOriginalFilename());//得到文件名  System.*out*.println("文件大小:"+mf.getSize());//得到文件大小  // System.out.println(":"+mf.getInputStream()); 得到文件流  String oldName=mf.getOriginalFilename();  //把文件写到服务器的tomcat/webapps/bjsxt/upload/  //1,得到服务器里面要保存文件的父路径  String realPath = request.getServletContext().getRealPath("/upload");  //2，根据文件名构造文件  File file=**new** File(realPath,oldName);  //3,使用MultipartFile的方法把mf里面的流写入file文件  **try** {  mf.transferTo(file);  } **catch** (Exception e) {  e.printStackTrace();  }  **return** "success";  } |

## 多文件上传不改名保存到服务器

### 修改index.jsp

|  |
| --- |
| <h1>多文件单个文件不改名</h1>  <form action=*"upload/upload2.action"* method=*"POST"* enctype=*"multipart/form-data"*>  选择文件<input type=*"file"* name=*"mf"*><br/>  选择文件<input type=*"file"* name=*"mf"*><br/>  选择文件<input type=*"file"* name=*"mf"*><br/>  选择文件<input type=*"file"* name=*"mf"*><br/>  <input type=*"submit"* value=*"上传"*>  </form> |

### 修改FileController

|  |
| --- |
| @RequestMapping("upload2")  **public** String upload2(MultipartFile[] mf,HttpServletRequest request){  //传多个文件，使用MultipartFile数组来接收  **if**(**null**!=mf&&mf.length>0){  **for**(**int** i = 0;i<mf.length;i++){  //1、获取文件的原名  String oldName =mf[i].getOriginalFilename();  //2、获取存储在服务器的父路径  String realPath = request.getServletContext().getRealPath("/upload");  //3、根据父路径和文件名构造文件  File file = **new** File(realPath, oldName);  //4、使用MutilpartFile的方法将mf里面的流写入file文件中  **try** {  mf[i].transferTo(file);  } **catch** (IllegalStateException | IOException e) {  e.printStackTrace();  }  }  }  **return** "success";  } |

## 产生随机文件名的工具类

### 创建RandomUtils

|  |
| --- |
| **public** **class** RandomUtil {  //1、使用时间格式命名文件  **static** SimpleDateFormat *dateFormat* = **new** SimpleDateFormat("yyyyMMddHHmmssSSS");  **static** Random *random* = **new** Random();    //2、根据文件名生成新名字，根据时间+四位随机数  //例如：名字的格式：oldName aaaaa.png aaaa.aaa.aaa.png  **public** **static** String createFileNameUserTime(String oldName){  //2.1获取文件名字的格式,最后一个点到最后的内容  String suffix = oldName.substring(oldName.lastIndexOf("."), oldName.length());  //2.2创建一个当前的时间  String time = *dateFormat*.format(**new** Date());  //2.3获取一个随机的四位数1000-9999  Integer random4 = *random*.nextInt(9000)+1000;  **return** time+random4+suffix;  }    //3、根据UUID产生一个新的名字  **public** **static** String createFileNameUserUUID(String oldName){  //3.1获取文件名字的格式,最后一个点到最后的内容  String suffix = oldName.substring(oldName.lastIndexOf("."), oldName.length());  //3.2产生一个随机字符串，并将—替换掉,然后变为大写  String uuid = UUID.*randomUUID*().toString().replace("\_", "").toUpperCase();  **return** uuid+suffix;  }  } |

## 单文件上传改名保存到服务器

### 修改index.jsp

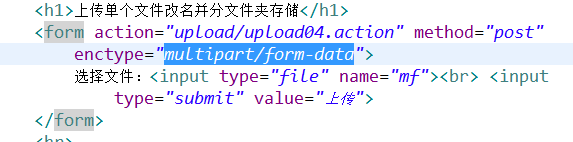
|  |
| --- |
| <h1>上传单个文件改名</h1>  <form action=*"upload/upload03.action"* method=*"post"*  enctype=*"multipart/form-data"*>  选择文件：<input type=*"file"* name=*"mf"*><br>  <input type=*"submit"* value=*"上传"*>  </form> |

### 修改FileController

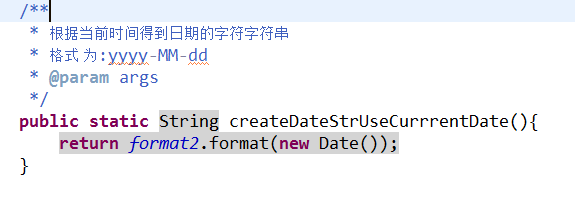
|  |
| --- |
| @RequestMapping("upload3")  **public** String upload3(MultipartFile mf,HttpServletRequest request){  //1、获取上传文件的名字  String oldName = mf.getOriginalFilename();  //2、获取保存在服务器上文件的父路径  String realPath = request.getServletContext().getRealPath("/upload");  //3、获取原文件名的格式，并产生一个和时间相关的随机名字  String newName = RandomUtil.*createFileNameUserTime*(oldName);  //4、根据文件名构造文件  File file = **new** File(realPath, newName);  //5、使用MultipartFile 的方法吧mf里面的流写入file文件  **try** {  mf.transferTo(file);  } **catch** (IllegalStateException | IOException e) {  e.printStackTrace();  }  **return** "success";  } |

## 单文件上传改名并分文件夹保存到服务器

### 修改index.jsp



### 修改RandomUtils



### 修改FileController

|  |
| --- |
| @RequestMapping("upload5")  **public** String upload5(MultipartFile mf,HttpServletRequest request){    //1、获取读取的文件名  String oldName = mf.getOriginalFilename();  //2、获取保存在服务器上文件夹的路径  String realPath = request.getServletContext().getRealPath("/upload");  //3、创建一个新的文件夹名字，这个文件夹是根据天来命名的。  String dirName = RandomUtil.*createFileNameUserDate*();  //4、根据路径和文件名创建一个文件,并判断该文件夹是否存在，如果不存在则创建。  File file = **new** File(realPath, dirName);  **if**(!file.exists()){  file.mkdirs();  }  //5、根据原文件名获取一个uuid名字  String newName = RandomUtil.*createFileNameUserUUID*(oldName);  //6、创建一个新的文件,根据父文件夹和文件名创建一个文件  File file1 = **new** File(file, newName);  //7、将mf中的数据输出到file1文件中  **try** {  mf.transferTo(file1);  } **catch** (IllegalStateException | IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **return** "success";  } |

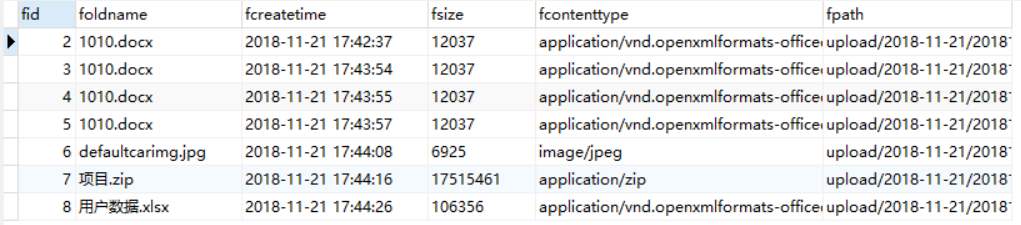
## 文件上传改名并分文件夹保存到服务器和数据库

文件上传改名并分文件夹保存到服务器和数据库

服务器存放的是文件

数据库存放的是上传的文件在服务器的地址

### 创建数据库和表



注：

### 生成Attachment类

### 生成AttachmentMapper

|  |
| --- |
| **public** **class** Attachment {  **private** Integer fid;  **private** String foldname;  **private** Date fcreatetime;  **private** Double fsize;  **private** String fcontenttype;  **private** String fpath;  ….  } |

### mapper接口

|  |
| --- |
| **public** **interface** AttachmentMapper {  //1、插入一条文件信息  **int** insert(Attachment attachment);    } |

### mapper.xml

|  |
| --- |
| <insert id=*"insert"* parameterType=*"com.sxt.domain.Attachment"* >  insert into sys\_attachment (fid, foldname, fcreatetime,  fsize, fcontenttype, fpath  )  values (#{fid,jdbcType=INTEGER}, #{foldname,jdbcType=VARCHAR}, #{fcreatetime,jdbcType=TIMESTAMP},  #{fsize,jdbcType=DOUBLE}, #{fcontenttype,jdbcType=VARCHAR}, #{fpath,jdbcType=VARCHAR}  )  </insert> |

### 创建AttachmentService

|  |
| --- |
| **public** **interface** AttachmentService {  //1、保存文件数据  **public** **void** addAttachment(Attachment attachment);  } |

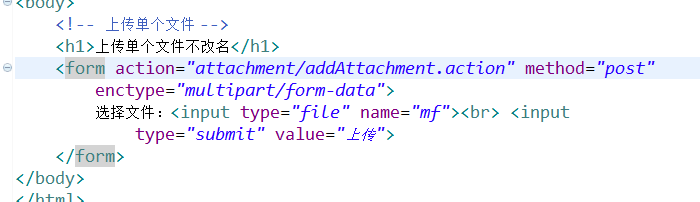
### 创建AttachmentServiceImpl

|  |
| --- |
| @Controller  @Transactional  **public** **class** AttachmentServiceImpl **implements** AttachmentService {  @Autowired  AttachmentMapper attachmentMapper;  //保存文件信息  @Override  **public** **void** addAttachment(Attachment attachment) {  attachmentMapper.insert(attachment);  }  } |

### 创建AttachmentController

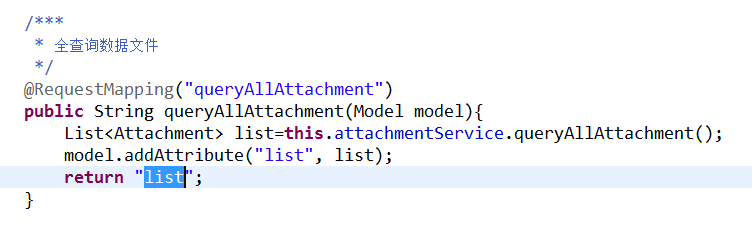
|  |
| --- |
| /\*\*  \* 文件管理控制器  \* **@author** LJH  \*  \*/  @Controller  @RequestMapping("attachment")  **public** **class** AttachmentController {  @Autowired  **private** AttachmentService attachmentService;    /\*\*  \* 上传文件  \*/  @RequestMapping("addAttachment")  **public** String addAttachment(MultipartFile mf,AttachmentVo attachmentVo){  String oldName=mf.getOriginalFilename();  String fcontenttype=mf.getContentType();  Long size=mf.getSize();  //1,得到服务器里面要保存文件的父路径  String realPath = WebUtils.*getServletContext*().getRealPath("/upload");  //2,得到当前时间的文件夹名  String dirName=RandomUtils.*createDateStrUseCurrrentDate*();  //3,根据文件夹对象  File dir=**new** File(realPath,dirName);  //4,判断这个文件夹是否存在 如果不存在 就创建  **if**(!dir.exists()){  dir.mkdirs();  }  String newName=RandomUtils.*createFileNameUseTime*(oldName);  //5，根据文件名构造文件  File file=**new** File(dir,newName);  //6,使用MultipartFile的方法把mf里面的流写入file文件  **try** {  mf.transferTo(file);  } **catch** (Exception e) {  e.printStackTrace();  }  Attachment attachment=**new** Attachment();  attachment.setFoldname(oldName);  attachment.setFcontenttype(fcontenttype);  attachment.setFsize(Double.*valueOf*(size));  attachment.setFcreatetime(**new** Date());  attachment.setFpath("upload/"+dirName+"/"+newName);//注意这里要使用相对路径  //7,把文件信息保存到数据库  attachmentService.addAttachment(attachment);  **return** "success";  }    } |

### 创建addAttachment.jsp

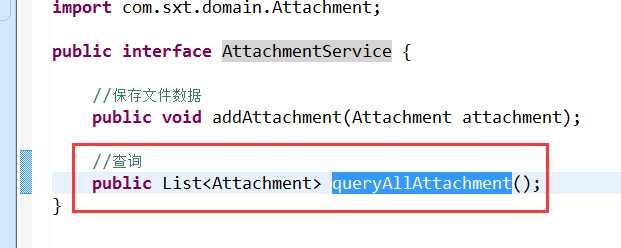


## 查询所有文件并显示

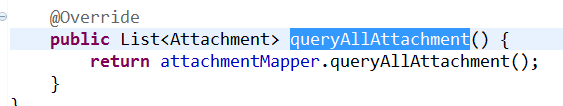
### 修改AttachmentController



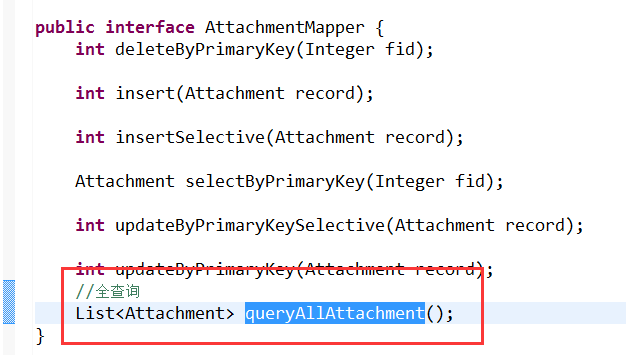
### 修改AttachmentSerivce



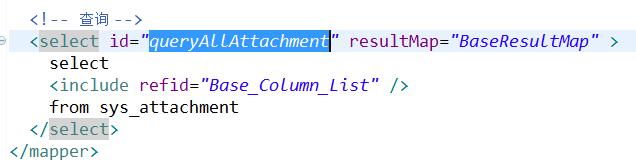
### 修改AttachmentSerivceImpl



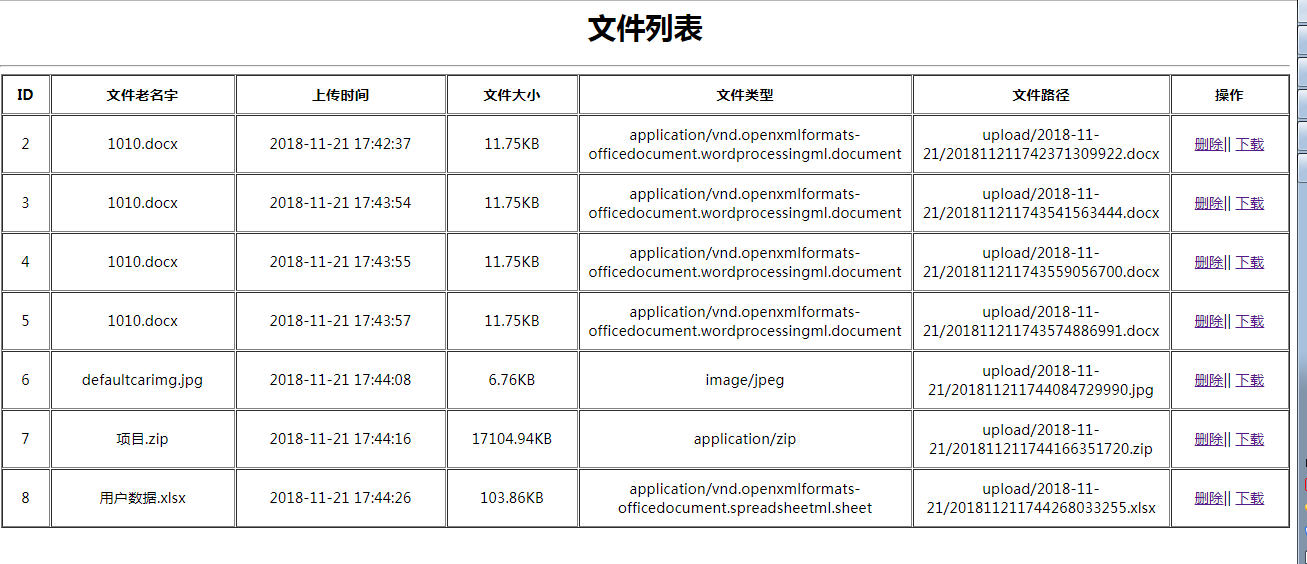
### 修改AttachmentMapper



### 修改AttachmentMapper.xml



### 创建WEB-INF/view/list.jsp



|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <%@taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>  <%@taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  <%  String path = request.getContextPath();  String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";  %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <base href=*"*<%=basePath%>*"*>    <title>文件列表</title>    <meta http-equiv=*"pragma"* content=*"no-cache"*>  <meta http-equiv=*"cache-control"* content=*"no-cache"*>  <meta http-equiv=*"expires"* content=*"0"*>  <meta http-equiv=*"keywords"* content=*"keyword1,keyword2,keyword3"*>  <meta http-equiv=*"description"* content=*"This is my page"*>  <!--  <link rel="stylesheet" type="text/css" href="styles.css">  -->  </head>    <body>  <h1 align=*"center"*>文件列表</h1>  <hr>  <table align=*"center"* border=*"1"* width=*"100%"* cellpadding=*"10"*>  <tr>  <th>ID</th>  <th>文件老名字</th>  <th>上传时间</th>  <th>文件大小</th>  <th>文件类型</th>  <th>文件路径</th>  <th>操作</th>  </tr>  <c:forEach items=*"*${list }*"* var=*"sn"*>  <tr>  <td align=*"center"*>${sn.fid }</td>  <td align=*"center"*>${sn.foldname }</td>  <td align=*"center"*>  <fmt:formatDate value=*"*${sn.fcreatetime }*"* pattern=*"yyyy-MM-dd HH:mm:ss"*/>  </td>  <td align=*"center"*>  <fmt:formatNumber value=*"*${sn.fsize/1024 }*"* pattern=*"#.00KB"*></fmt:formatNumber>  </td>  <td align=*"center"* width=*"20%"*>${sn.fcontenttype }</td>  <td align=*"center"* width=*"20%"*>${sn.fpath }</td>  <td align=*"center"*>  <a href=*""*>删除</a>||  <a href=*""*>下载</a>  </td>  </tr>  </c:forEach>  </table>  </body>  </html> |