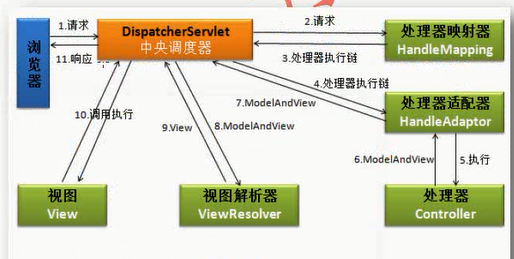
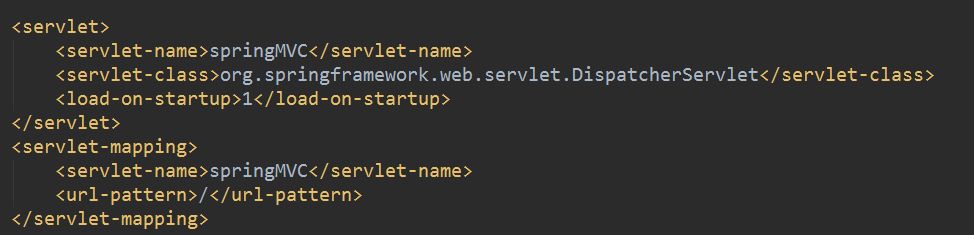
# 执行流程



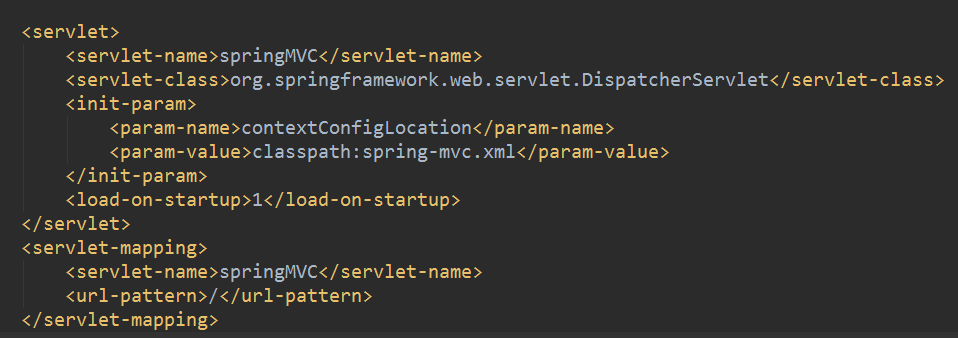
## 基础配置

### web.xml配制



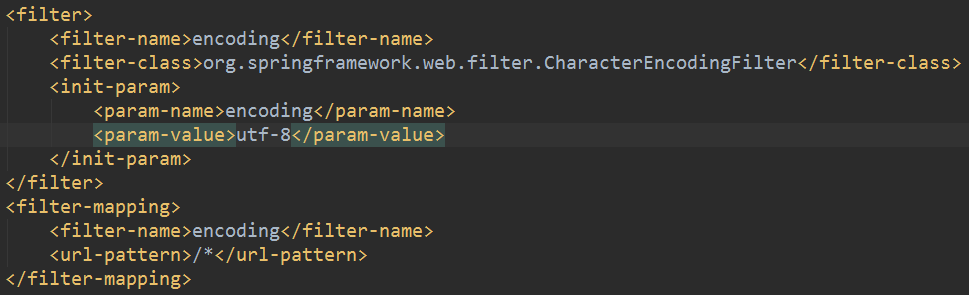
DispatcherServlet是SpringMVC诸多默认的组件，在初始化时，它会寻找在应用程序的WEB-INF目录中的配置文件，该配置文件的命名方式为：servletName-servlet.xml，其中servletName是部署描述符中的Dispatcher servlet的名字：如：我配置的是spirngMVC，则对应的文件在WEB-INF/springMVC.xml。

当然我们可以将配置文件放在应用中的任何地方，只需要配置如下方式即可：



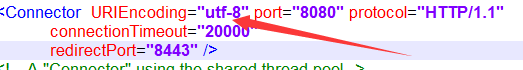
### 乱码问题

解决POST乱码问题：在web.xml添加如下配置



解决GET乱码问题：

1. 修改Tomcat配置文件（server.xml）。



2.对参数进行重新编码



### url-pattern配置问题

* 不能为/\*：url-pattern不能为/\*，因为DispatcherServlet会将所有的动态页面的跳转请求当做一个普通的Controller请求，中央调度器会调用处理器映射器为其寻找相应的处理器，最终寻找不到。
* 最好也不要写成”/”：因为SpringMVC会将静态资源当做普通的Controlle请求

## 路径问题

### 相对路径

### 绝对路径

# 处理器映射器

HandlerMapping接口负责根据request请求寻找 对应的Handler处理器和Interceptor拦截器，并将它们封装到HandlerExecutionChain对象中，返回给中央处理器。

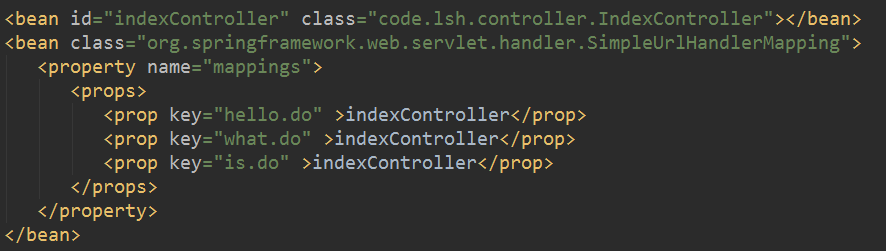
常用的实现类有两种：

1. BeanNameUrlHandlerMapping

不足点：（1）处理器的Bean的id为一个url的请求路径，而不是Bean的名称

（2）处理器Bean的定义与请求的url绑定一起。如出翔多个url使用同一个处理器Bean，就需要在Spring IoC容器配置多次同一个处理器Bean，这将导致创建多个处理器实例。

1. SimpleUrlHandlerMapping



1. 最新的：RequestMappingHandlerMapping



# 处理器适配器

默认注册的：SimpleControllerHandlerAdapter ：handler处理器实现Controller接口

HttpRequestHandlerAdapter：handler处理器实现了HttpRequestHandler接口。

AnnotationMethodHandlerAdapter

最新的适配器：RequestMappingHandlerAdapter



# 处理器

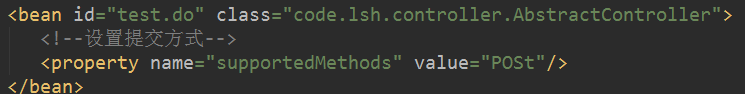
处理器不仅是继承Controller接口，还可以继承一些其他类来完成一些特殊的功能。

如：

1.继承org.springframework.web.portlet.mvc.AbstractController



注册处理器并设置Http请求方法：



2.继承org.springframework.web.servlet.mvc.multiaction.MultiActionController

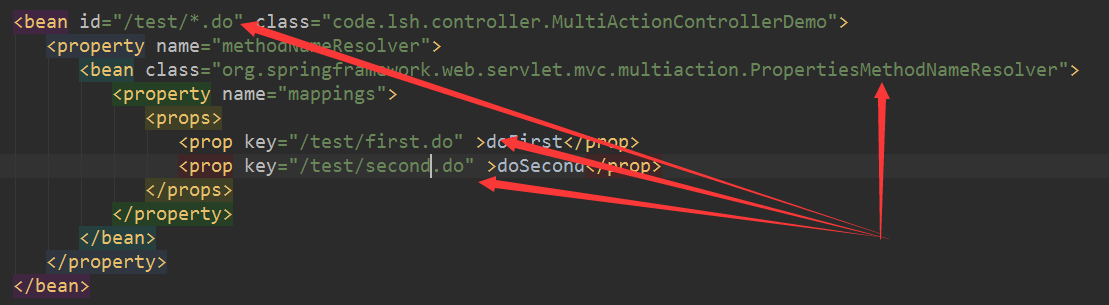
MultiActionController实现了Controller接口，允许多个Http请求被同一个class处理。

继承MultiActionController的类的处理器handler中的方法的返回值可以是：Map，String，void和ModelAnd’View。

MultiActionController类具有一个methodNameResolver属性（方法名解析器），该解析器会将方法名作为资源名称进行解析，这就意味着，在我们提交请求时，要将方法名作为资源名称出现。默认的方法名解析器是InternalPathMethodNameResolver，我们可以通过setMethodNameResolver()方法更改默认的方法名解析器。；

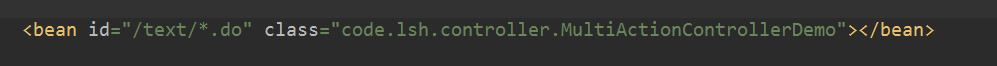
方法名解析器是实现了MethodNameResolver接口，而SpringMVC为我们提供了3个具体的实现：InternalPathMethodNameResolver，ParameterMethodNameResolver和

PropertiesMethodNameResolver。



|  |
| --- |
| import org.springframework.web.servlet.ModelAndView; import org.springframework.web.servlet.mvc.multiaction.MultiActionController;  import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse;  public class MultiActionControllerDemo extends MultiActionController {  public ModelAndView doFirst(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws Exception  {  ModelAndView mav = new ModelAndView() ;  mav.addObject("message","invoke doFirst method") ;  mav.setViewName("/index.jsp");  return mav;  }   public ModelAndView doSecond(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws Exception  {  ModelAndView mav = new ModelAndView() ;  mav.addObject("message","invoke doSecond method") ;  mav.setViewName("/index.jsp");  return mav;  } } |

注册处理器handler：



注意：MultiActionController的所有处理器handler方法必须是public，并且方法不能重载。

## 参数绑定

Controller中方法默认支持的参数类型：HttpServletRquest，HttpServletResponse，

HttpSession，Model。

Controller中方法返回值类型：void，ModelAndView，String，Map。

绑定数组：

## 数据模型

在控制器方法中，可以把ModelAndView，Model，ModelMap作为参数，在SpringMVC运行时，会自动初始化它们，因此可以选择ModelMap或Model作为数据模型。实际上SpringMVC创建的是一个BindingAwareModelMap实例。

### ModelAndView

### Model

主要用于处理器类中方法，用于储存返回给页面的数据，底层是request域传递数据，但是对request域进行了扩展。

## 保存并获取属性参数

* @RequestAttribute获取HTTP的请求对象属性值，用来传递给控制器的参数。
* @SessionAttribute在HTTP的回话对象属性值中，用来传递该控制器的参数。
* @SesssionAttributes，可以给它配置一个字符串数组，这个数组对应的是数据模型对应的键值对，然后讲这些键值对保存到Session中。不过这个注解只能对类进行标注，不能对方法或者参数注解。它可以配置属性名称或者属性类型。
* @Controller  
  @RequestMapping("/attr")  
  @SessionAttributes(names={"id"},types={Person.class})  
  public class AttributeController  
  {  
   @RequestMapping("test1")  
   public ModelAndView attr()  
   {  
   ModelAndView mav = new ModelAndView();  
   Person person = new Person();  
   person.setName("hao");  
   person.setAge(20);  
   mav.addObject("person",person) ;  
   mav.setViewName("/session.jsp");  
   return mav ;  
   }

如果浏览器请求/attr/test1?id=1，那么它就会进入attr方法中，然后数据模型保存了一个Person对象，由于它们满足@SessionAttributes的配置，所以最后它会保存到Session对象中。

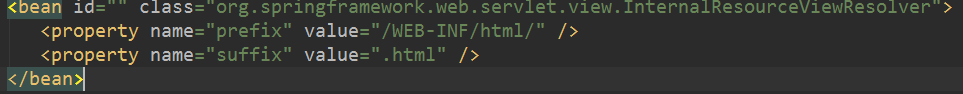
* @CookieValue和@RequestHeader

# 视图解析器

视图解析器是实现了ViewResolver接口的类。

常用的视图解析器：

1. InternalResourceViewResolver，该视图解析器用于web应用内部资源的封装和跳转，对内部资源的查找规则：前缀+视图名称+后缀，拼接成一个web应用内部资源路径。InternalResourceViewResolver解析器会将处理器方法返回的模型属性都存放到对应的request中，让后将请求转发到目标的URL。



1. BeanNameViewResolver是将资源名称封装为Bean，并将封装的Bean纳入到Spring IoC容器中管理，ModelAndView通过设置的视图名称为该Bean的id值来对资源文件的访问，我们可以在配置文件中配置多个View视图Bean。

该解析器存在如下的不足：

（1）只可以完成将内部资源封装后的跳转，但无法转向外部资源。

（2）对于内部资源的定义，也只能定义一种格式的资源：存放在同一目录的同一文件类型的资源文件。

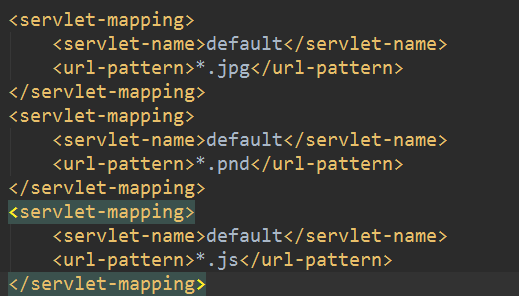




1. XmlViewResolver
2. VelocityViewResolver
3. FreeMarkerViewResolver
4. 定义多个视图解析器

# 解决静态资源

方法一：使用Tomcat中配置的Servlet，进行静态资源处理org.apache.catalina.servlets.DefaultServlet，我们只要在Web项目中的web.xml进行如下配置即可。

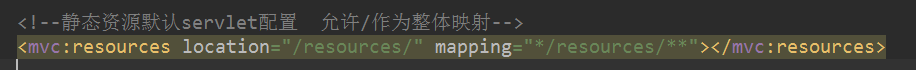


方法二：使用mvc:default-servlet-handler标签。



<mvc:default-servlet-handler/>会对静态资源的访问请求通过HandlerMapping映射到默认的Servlet请求处理器DefaultServletHttpRequestHandler对象，而该处理器调用了Tomcat的DefaultServlet。

第三种：使用mvc:resources标签



在Spring3.x之后，Spring专门定义了处理静态资源访问请求的处理器：ResourceHttpRequestHandler。

location：表示静态资源所处的目录。

mapping：表示对静态资源的请求

# 请求转发和重定向

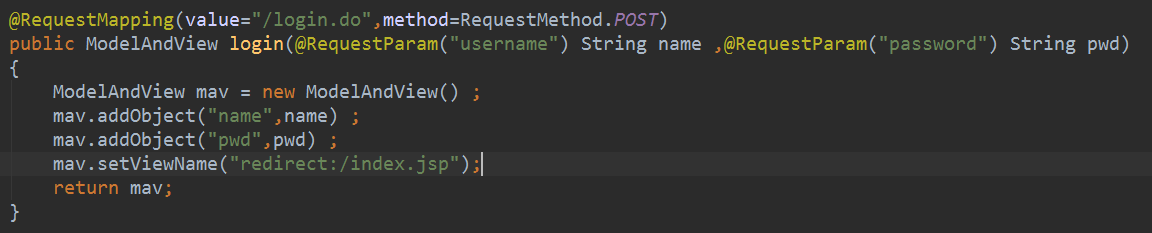
## 返回ModelAndView的请求转发

1. 请求转发到页面
2. 请求转发到其他Controller

## 返回ModelAndView的重定向

1. 重定向到页面

A：通过ModelAndView携带参数





该提交方法为POST，重定向到index.jsp页面，但将参数拼接在地址栏中，我们可以使用${param.paramName}获取参数的值。

如果ModelAndView中存放bean，则bean的属性值无法被传递

**注意：**我们重定向的路径需要加上“/”，因为重定向无法访问WEB-INF目录下的页面，并且加上“/”的参考路径是我们web应用的根:www.localhost:8080/，如果不加上“/”，则参考路径是变为我们请求路径-资源名称：www.localhost:8080/login/

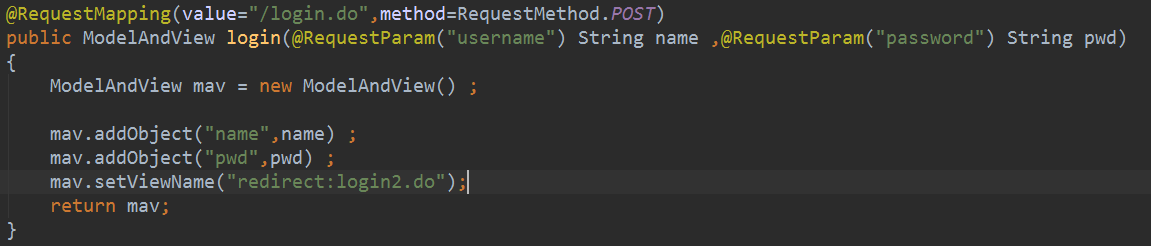
B：通过HttpSession携带参数

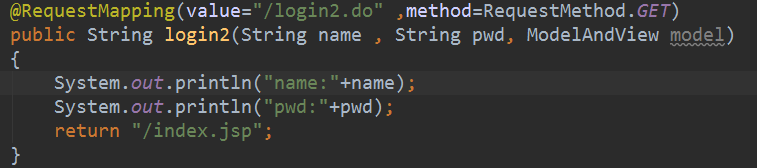
C：通过Model携带参数

（1）重定向到页面，如果Model存放参数，则参数会以URL的形式传递到重定向的页面。

1. 重定向到其他Controller

A：通过ModelAndView携带参数







携带的参数拼接到url上了。对象无法被传递。

我们可以使用RedirectAttributes作为形参，来保存我们需要的对象

@RequestMapping("/test1")  
public ModelAndView test1(RedirectAttributes ra)  
{  
 ModelAndView mav = new ModelAndView() ;  
 Person person = new Person();  
 person.setAge(25);  
 person.setName("sai");  
 ra.addFlashAttribute("person",person) ;  
 mav.setViewName("redirect:/redirect/test4");  
 return mav ;  
}

当我们调用addFlashAttribute方法后，SpringMVC会将数据保存到Session中，重定向后就会将其清除，这样就可以将数据传递到下一个地址了。

B：通过HttpSession携带参数

## 返回String的 请求转发

## 返回String的重定向

1. 重定向到其他页面

A：通过ModelAndView携带参数

B：通过形参RedirctAttribute的addAttribute携带参数

1. 重定向到其他Controller

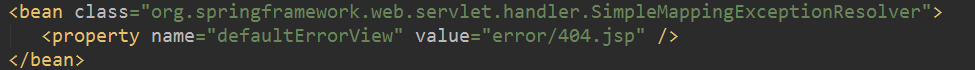
## 返回void的重定向

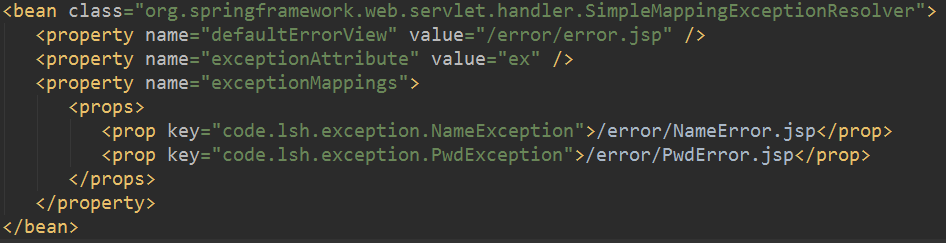
## 返回void的请求转发

# 异常处理

常用的异常处理方法：

1. 使用系统定义好的异常处理类SimpleMappingExceptionResolver



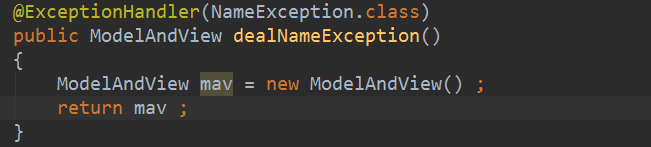


1. 使用自定义异常处理

实现HandlerExceptionResolver接口

1. 使用异常处理注解

使用注解@ExceptionHandler



[4．用@ControllerAdvice](mailto:4.使用@ControllerAdvice)注解

如果使用@ExceptionHandler注解，该注解必须在同一个Controller类中才有效，但在Spring3.x之后，Spring提供了@ControllerAdvice注解，方便我们更好的处理异常，

@ControllerAdvice  
public class GlobalExceptionHandler  
{  
 @ExceptionHandler({MaxUploadSizeExceededException.class})  
 public String dealUploadException(HttpServletRequest request, Exception ex){  
 if(ex instanceof MaxUploadSizeExceededException){  
 return "redirect:/upload.jsp";  
 }  
 return "index.jsp";  
 }  
}

使用@ControllerAdvice标注的类，类中的方法可以处理一种或者多种异常类型，并且异常处理方法的返回值可以是ModelAndView，String

@ExceptionHandler({MaxUploadSizeExceededException.class})  
public String dealUploadException(HttpServletRequest request, Exception ex){  
 FlashMap flashMap = (FlashMap) request.getAttribute(DispatcherServlet.*OUTPUT\_FLASH\_MAP\_ATTRIBUTE*);  
 if(ex instanceof MaxUploadSizeExceededException){  
 flashMap.put("msg","文件过大");  
 return "redirect:/upload/redirect";  
 }  
 return "index.jsp";  
}

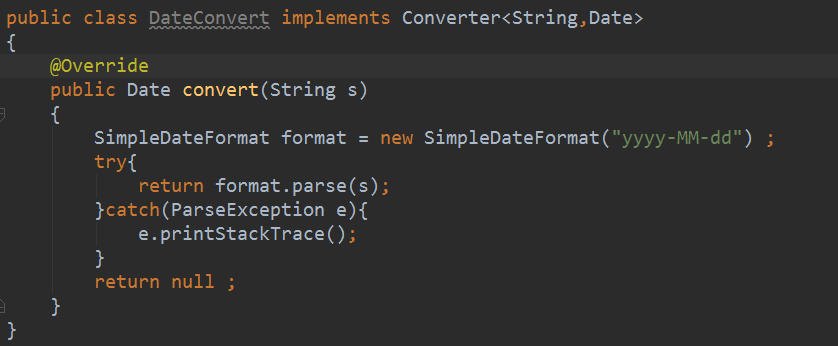
这里使用FlashMap，在重定向时传递参数。

# 类型转换器

用于处理SpringMVC无法处理的类型转换：如，日期类型。

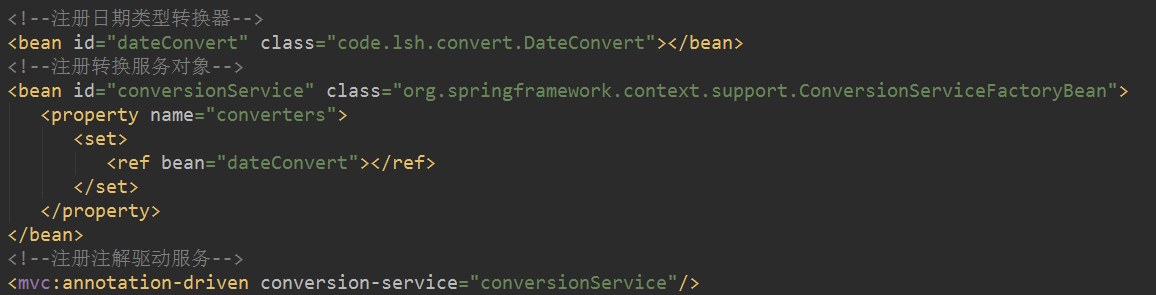
**自定义类型转换器：**日期类型转换器

第一步：需要实现org.springframework.core.convert.converter .Converter接口。



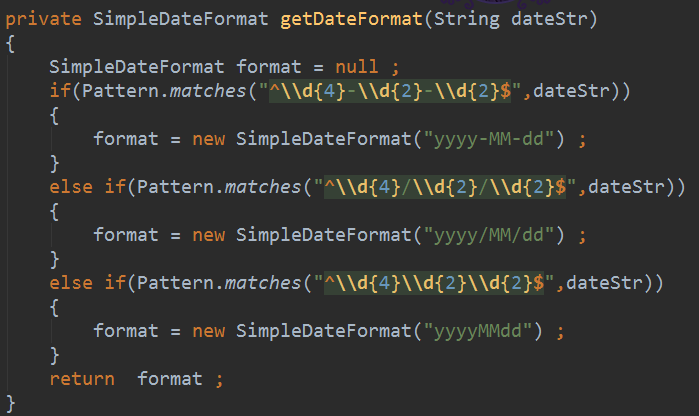
第二步：将该转化器注册到Spring容器内，并被ConversionServiceFactoryBean统一管理。

（或者org.springframework.format.support.FormattingConversionService）



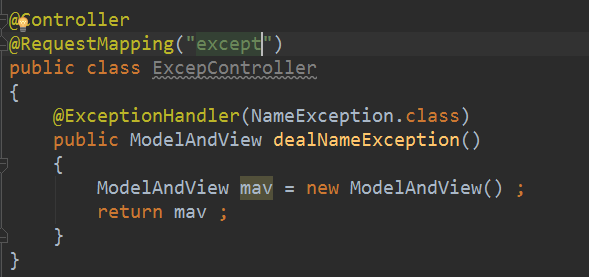
不足：使用这种方式，我们只能处理一种类型的日期格式，并且将SpringMVC支持的日期类型转换覆盖掉了。

我们可以使用正则表达式进行处理：在接受的字符串后，判断字符串的类型以生成对应的SimpleDateFormat对象。

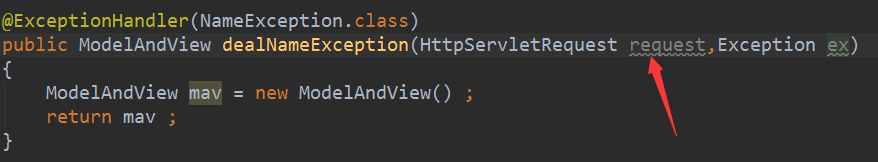


**数据回现：**当数据类型转换发生异常后，需要返回表单页面，让用户重新填写。但在正常情况下，当发生类型转换异常时，SpringMVC会自动跳转到400页面，所以，若想在发生类型转换异常后，跳转到指定的页面，需要将异常进行捕获，然后通过异常处理器跳转到指定的页面。

提示：当请求参数的值与接受该参数的处理器方法形参的类型不匹配时，会抛出类型为TypeMismatchException



在异常处理时，我们通过HttpServletRequest获取用户提交的参数。



在类型转换失败后，该用户的提示信息的展示：



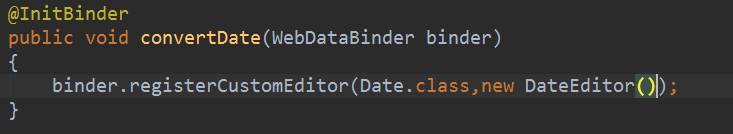
# 初始化参数绑定：

类型转换的另一种方式，使用@InitBinder注解，由数据绑定器WebDataBinder作为参数，即可在请求中所携带的参数被处理器方法执行前先进性类型编译器的绑定操作，实现对特定类型的转换。

**第一步：**自定义属性转换器

|  |
| --- |
| import org.springframework.beans.TypeMismatchException; import org.springframework.beans.propertyeditors.PropertiesEditor;  import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; import java.util.regex.Pattern;  public class DateEditor extends PropertiesEditor {  @Override  public void setAsText(String text) throws IllegalArgumentException  {  SimpleDateFormat format = getDateFormat(text);  try{  setValue(format.parse(text));  }catch(ParseException e){  e.printStackTrace();  }  }   private SimpleDateFormat getDateFormat(String dateStr)  {  SimpleDateFormat format = null ;  if(Pattern.*matches*("^\\d{4}-\\d{2}-\\d{2}$",dateStr))  {  format = new SimpleDateFormat("yyyy-MM-dd") ;  }  else if(Pattern.*matches*("^\\d{4}/\\d{2}/\\d{2}$",dateStr))  {  format = new SimpleDateFormat("yyyy/MM/dd") ;  }  else if(Pattern.*matches*("^\\d{4}\\d{2}\\d{2}$",dateStr))  {  format = new SimpleDateFormat("yyyyMMdd") ;  }  throw new TypeMismatchException("",Date.class);  } } |

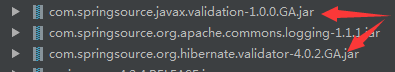
第二步：使用注解。



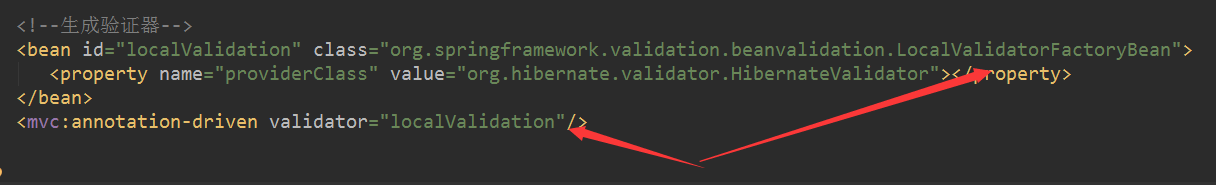
# 数据验证

SpringMVC支持JSR（Java Specification Requests）303-Bean Validation数据验证规范。我们常用Hibernate Validation

第一步：引入jar包



第二步：注册验证器LocalValidatorFactoryBean和配置注解驱动



第三步：使用注解

常用的注解：

@Valid @NotBlack BindingResult

如果在实体类中，如果需要判定实体的一个属性不能为空，我们可以使用@NotBlack注解，

@NotNull:值不能为空。

@Null:值必须为空

@Pattern(regex=""):字符串必须匹配正则表达式

@Size(min=?,max=?):集合的元素数量必须在min和max之间。

@Emial:字符串必须是Email地址

@Length(min=?,max=?):检验字符串的长度

@NotBlack:字符串必须有字符

@NotEmpty:字符串不为null，集合有元素

@Range(min=?,max=?):数字必须大于等于min小于等于max

@URL:字符串是合法的url

@SafeHtml:字符串是合法的html

@AssertFalse:值必须是false

@AssertTrue:值必须是true

@Future:值必须是未来的日期

@Past:值必须是过去的日期

@Max(value=?):数字的最大值

@Min(value=?):数字的最小值

@DecimalMax(max=?,inclusive=?):

@DecimalMin(min=?,inclusive=?):

@Digits(integer=?,fraction=?):数字检验格式，integer指定整数部分的最长长度，fraction指定小数部分的最长长度。

|  |
| --- |
| import javax.validation.constraints.\*;  public class Student {  @NotNull(message = "姓名不能空")  @Size(min = 3,max = 6,message = "姓名长度{min}-{max}")  private String name ;   @Min(value = 0 ,message = "成绩不能小于{value}")  @Max(value = 100,message = "成绩不能大于{value}")  private double score ;   @NotNull(message = "手机号不能为空")  @Pattern(regexp = "^1[34578]\\d{8}$",message = "手机号格式不正确")  private String mobile ;   public String getName(){  return name;  }   public void setName(String name){  this.name = name;  }   public double getScore(){  return score;  }   public void setScore(double score){  this.score = score;  }   public String getMobile(){  return mobile;  }   public void setMobile(String mobile){  this.mobile = mobile;  }   @Override  public String toString()  {  return "Student{" + "name='" + name + '\'' + ", score=" + score + ", mobile='" + mobile + '\'' + '}';  } } |

**在Controller层接受实体参数时，使用@Valid进行验证，并且使用BindingResult**

**类为方法参数，进行错误的接受**

|  |
| --- |
| import code.lsh.bean.Student; import org.springframework.stereotype.Controller; import org.springframework.validation.BindingResult; import org.springframework.validation.FieldError; import org.springframework.validation.annotation.Validated; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.servlet.ModelAndView;  @Controller @RequestMapping("/valida") public class ValidationController {  @RequestMapping(value = "validaStu",method = RequestMethod.*POST*)  public ModelAndView validationStudent(@Validated Student student, BindingResult br)  {  ModelAndView mav = new ModelAndView() ;  int errorCount = br.getErrorCount() ;  if(errorCount >0 )  {  FieldError scoreError = br.getFieldError("score") ;  FieldError nameError = br.getFieldError("name") ;  FieldError mobileError = br.getFieldError("mobile");  if(nameError!=null)  {  mav.addObject("nameErro",nameError.getDefaultMessage()) ;  }  if(scoreError!=null)  {  mav.addObject("scoreError",scoreError.getDefaultMessage()) ;  }  if(mobileError!=null)  {  mav.addObject("mobileError",mobileError.getDefaultMessage()) ;  }  }  mav.addObject("student",student) ;  mav.setViewName("/inputStudent.jsp");  return mav ;  } } |

案例二：

|  |
| --- |
| package code.lsh.bean;  import org.springframework.format.annotation.DateTimeFormat;  import javax.validation.constraints.\*; import java.util.Date;  public class Transaction {  */\*\*  \* 产品编号  \*/* @NotNull(message="不能为空")  private Long productId ;  */\*\*  \* 用户编号  \*/* @NotNull(message="不能为空")  private Long userId ;  */\*\*  \* 交易日期  \*/* @NotNull(message="不能为空")  @Future(message="只能是将来日期")  @DateTimeFormat(pattern="yyyy-MM-dd")  private Date date ;  */\*\*  \* 价格  \*/* @NotNull(message="不能为空")  @DecimalMin(value="0.1",message="最少0.1元")  private Double price ;   */\*\*  \* 数量  \*/* @Min(value=1,message="最少为1")  @Max(value=100,message="最大为100")  @NotNull(message="不能为空")  private Integer quantity ;   */\*\*  \* 交易金额  \*/* @NotNull  @DecimalMax(value="50000.0",message="最多为50000.0元")  @DecimalMin(value="1.0",message="最少为1.0元")  private Double amount ;   */\*\*  \* 邮件  \*/* @Pattern(regexp="^([a-zA-Z0-9]\*[-\_]?[a-zA-Z0-9]+)\*@([a-zA-Z0-9]\*[-\_]?[a-zA-Z0-9]+)+[\\.][A-Za-z]{2,3}([\\.])[a-zA-Z]{2}?$",message="邮件格式不正确")  private String email ;   */\*\*  \* 备注  \*/* @Size(min=0,max=256,message="0到255个字符")  private String note ;   public Long getProductId(){  return productId;  }   public void setProductId(Long productId){  this.productId=productId;  }   public Long getUserId(){  return userId;  }   public void setUserId(Long userId){  this.userId=userId;  }   public Date getDate(){  return date;  }   public void setDate(Date date){  this.date=date;  }   public Double getPrice(){  return price;  }   public void setPrice(Double price){  this.price=price;  }   public Integer getQuantity(){  return quantity;  }   public void setQuantity(Integer quantity){  this.quantity=quantity;  }   public Double getAmount(){  return amount;  }   public void setAmount(Double amount){  this.amount=amount;  }   public String getEmail(){  return email;  }   public void setEmail(String email){  this.email=email;  }   public String getNote(){  return note;  }   public void setNote(String note){  this.note=note;  }   @Override  public String toString(){  return "Transaction{"+"productId="+productId+", userId="+userId+", date="+date+", price="+price+", quantity="+quantity+", amount="+amount+", email='"+email+'\''+", note='"+note+'\''+'}';  } } |

控制器方法：

|  |
| --- |
| @Controller @RequestMapping("/validate") public class ValidateController {  @RequestMapping(value="/test",method=RequestMethod.*POST*)  public ModelAndView annotationValidateion(@Valid Transaction trans, Errors errors)  {  System.*out*.println(trans.toString());  ModelAndView mav = new ModelAndView() ;  if(errors.hasErrors())  {  List<FieldError> errorList = errors.getFieldErrors() ;  for(FieldError error :errorList)  {  System.*out*.println("field:"+error.getField()+"\t"+"msg:"+error.getDefaultMessage());  }  }  mav.setViewName("/index.jsp");  return mav ;  } |

## 自定义校验注解

@Constraint注解：自定义Bean验证约束，自定义Bean验证约束注解必须要引用@Constraint注解，该

## 使用验证器

Spring提供了Validator接口来实现检验，它将在进入控制器逻辑之前对参数的合法性进行校验。

Validator接口时SpringMVC检验表单逻辑核心接口，它的接口定义

第一步：实现Validator接口

mport org.springframework.validation.Errors;  
import org.springframework.validation.Validator;  
  
public class TrasactionValidator implements Validator  
{  
 @Override  
 public boolean supports(Class<?> aClass)  
 {  
 //判断验证其是否为Transaction，如果是则进行校验  
 return Transaction.class.equals(aClass);  
 }  
  
 @Override  
 public void validate(Object obj, Errors errors)  
 {  
 Transaction transaction = (Transaction)obj ;  
 //校验  
 double dis = transaction.getAmount()-(transaction.getPrice()\*transaction.getQuantity());  
 if(Math.*abs*(dis)>0.01)  
 {  
 //加入错误信息  
 errors.rejectValue("amount",null,"交易金额和购买金额与价格不匹配");  
 }  
 }  
}

第二步：绑定到对应的控制器中，使用@InitBinder注解

@InitBinder  
public void initBinder(DataBinder binder)  
{  
 //数据绑定器加入验证器  
 binder.setValidator(new TrasactionValidator());  
}

第三步：启动验证器

# 文件上传和下载

SpringMVC文件的上传是通过MultipartResolver（Mulitipart解析器）处理的，MultipartResolver是一个接口，它有两个实现类：

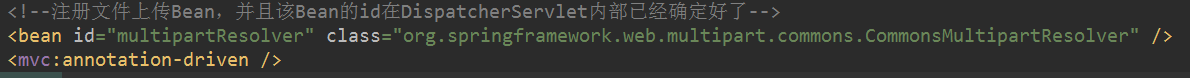
* CommonsMultipartResolver：依赖于Apache下的jakarta Common FileUpload项目解析Multipart请求，可以在Spring的各个版本中使用，只是它要依赖第三方包才得以实现。
* StandardServletMultipartResolver：是Spring3.1版本后的产物，它依赖于Servlet3.0或者更高版本的实现，它不依赖第三方jar包。

第一步：导入jar包



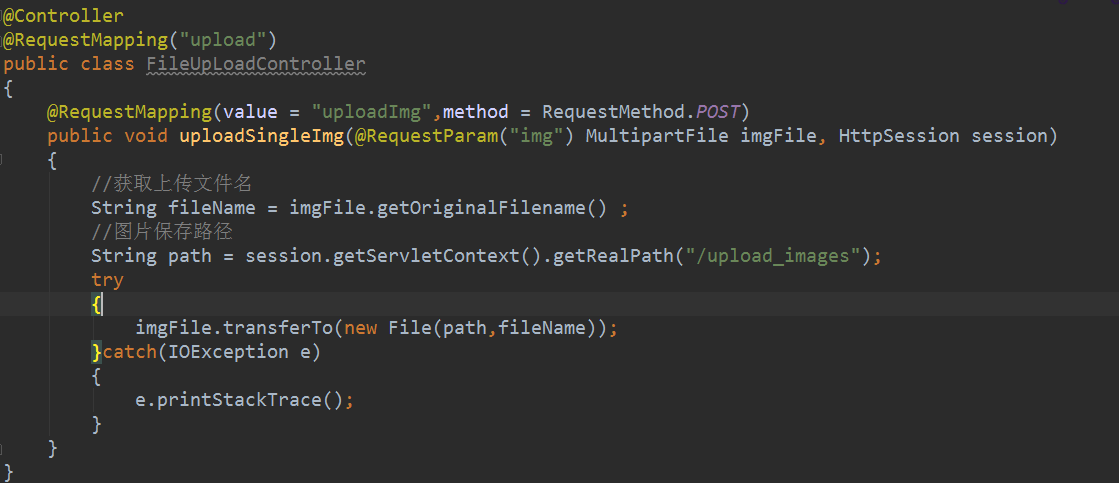
## 上传单个文件：

第二步：注册上传服务Bean：CommonsMutipartResolver



注意：multipartResolver是Spring约定好的Bean name不可以修改。

第三步：定义处理器方法，形参必须是MultipartFile



注意：无论是否上传文件，最终MultipartFile对象都被创建。

### 案例

使用StandardServletMultipartResolve单文件上传：

添加StandardServeltMultipartResolver Bean到容器中，这要Bean的id必须是multipartResolver。

<bean id="multipartResolver"  
 class="org.springframework.web.multipart.support.StandardServletMultipartResolver"></bean>

修改web.xml:

<servlet>  
 <servlet-name>spring-dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:config/applicationContext-\*.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
 <multipart-config>  
 <location>F:\\hao</location>  
 <max-file-size>-1</max-file-size>  
 <max-request-size>-1</max-request-size>  
 <file-size-threshold>0</file-size-threshold>  
 </multipart-config>  
</servlet>  
<servlet-mapping>  
 <servlet-name>spring-dispatcher</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>

Controller类：

@Controller  
@RequestMapping("/upload")  
public class UpLoadController  
{  
 @RequestMapping(value = "/image" ,method=RequestMethod.*POST*)  
 public String upload(MultipartHttpServletRequest request) throws Exception {  
 MultipartFile multipartFile = request.getFile("upload\_file");  
 if(multipartFile!=null){  
 String fileName = multipartFile.getName();  
 String originFileName = multipartFile.getOriginalFilename();  
 System.*out*.println("fileName:"+fileName);  
 System.*out*.println("originFileName:"+originFileName);  
 multipartFile.transferTo(new File("F:\\hao"+ File.*separatorChar*+fileName));  
 }  
 return "test/success";  
 }  
}

或者：修改方法

@RequestMapping(value = "/image" ,method=RequestMethod.*POST*)  
public String upload(@RequestParam("upload\_file") MultipartFile multipartFile) throws Exception {  
 if(multipartFile!=null){  
 String originFileName = multipartFile.getOriginalFilename();  
 System.*out*.println("originFileName:"+originFileName);  
 multipartFile.transferTo(new File("F:\\hao"+ File.*separatorChar*+originFileName));  
 }  
 return "test/success";  
}

upload.jsp:

<body>  
 <form action="upload/image" method="post" enctype="multipart/form-data">  
 Upload Image：<input type="file" name="upload\_file"><br><br>  
 <input type="submit" value="Submit"/>  
 </form>  
 <img src="" style="widht:200px;height:200px;">  
 <script>  
 window.onload=**function**()  
 {  
 **var** x=**new** FileReader;  
 document.forms[0].elements[0].onchange=**function**(){  
 x.readAsDataURL(**this**.files[0]);  
 };  
 x.onloadend=**function**(){  
 document.images[0].src=**this**.result;  
 }  
 }  
 </script>  
</body>

上传多文件：

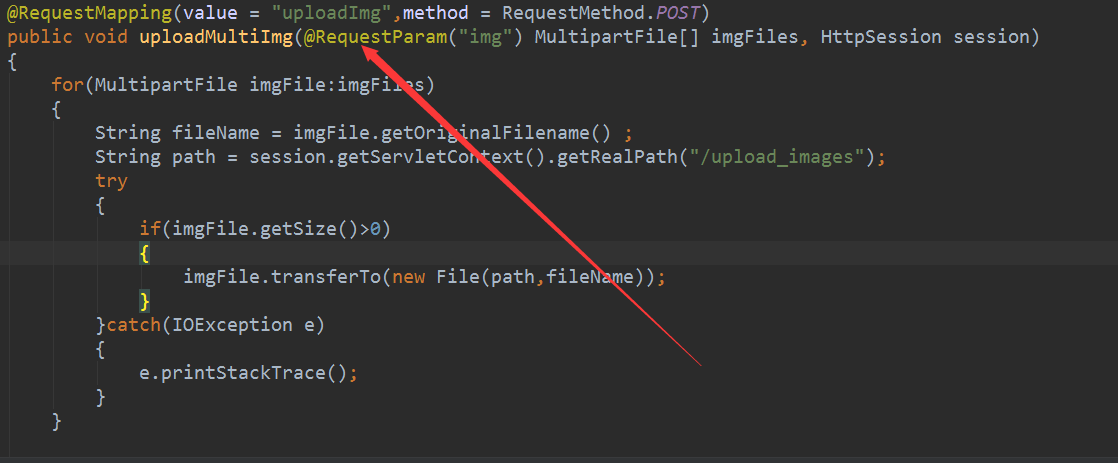
<form action="upload/image" method="post" enctype="multipart/form-data">  
 Upload Image1：<input type="file" name="upload\_file1"><br><br>  
 Upload Image2：<input type="file" name="upload\_file2"><br><br>  
 Upload Image3：<input type="file" name="upload\_file3"><br><br>  
 <input type="submit" value="Submit"/>  
</form>

@RequestMapping(value = "/image" ,method=RequestMethod.*POST*)  
public String upload(MultipartHttpServletRequest request) throws Exception {  
 Iterator<String> iter = request.getFileNames();  
 while(iter.hasNext()){  
 MultipartFile multipartFile = request.getFile(iter.next());  
 String originName = multipartFile.getOriginalFilename();  
 multipartFile.transferTo(new File("F:\\hao\\"+originName));  
 }  
 return "test/success";  
}

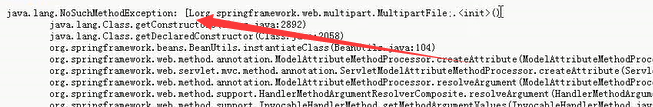
或者：注意，如果使用下面的方法，则页面中的input中的name都要是upload\_file

@RequestMapping(value = "/image" ,method=RequestMethod.*POST*)  
public String upload(@RequestParam("upload\_file") MultipartFile[] multipartFiles) throws Exception {  
 for(MultipartFile multipartFile:multipartFiles){  
 String originName = multipartFile.getOriginalFilename();  
 multipartFile.transferTo(new File("F:\\hao\\" + originName));  
 }  
 return "test/success";  
}

## 上传多个文件



如果不适用@RequestParam，则会发生异常。



说明数组无法没有被初始化。

## 问题

在我们配置MultipartResolver的Bean是，都指定了上传文件的最小文件大小和最大文件大小，而这些配置是否起到作用，不得而知。但是我们可以定义拦截器对上传请求进行拦截，进而判断是否符合我们的要求。

public class FileUploadInterceptor implements HandlerInterceptor  
{  
 private int maxSize ;  
 public void setMaxSize(int maxSize){  
 this.maxSize = maxSize;  
 }  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  
 throws Exception{  
 if(ServletFileUpload.*isMultipartContent*(request)){  
 ServletRequestContext servletRequestContext = new ServletRequestContext(request);  
 long size = servletRequestContext.contentLength();  
 if(size > maxSize){  
 throw new MaxUploadSizeExceededException(maxSize);  
 }  
 }  
 return true;  
 }  
 @Override  
 public void postHandle(HttpServletRequest request,  
 HttpServletResponse response, Object handler,  
 ModelAndView modelAndView) throws Exception{  
 }  
 @Override  
 public void afterCompletion(HttpServletRequest request,  
 HttpServletResponse response,  
 Object handler, Exception ex) throws Exception{  
 }  
}

修改配置文件：

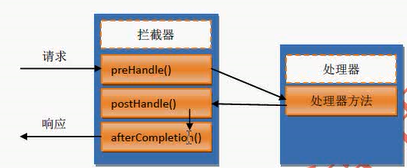
<mvc:interceptors>  
 <mvc:interceptor>  
 <mvc:mapping path="/\*\*"/>  
 <bean class="code.lsh.interceptor.FileUploadInterceptor">  
 //3\*1024\*1024  
 <property name="maxSize" value=""/>  
 </bean>  
 </mvc:interceptor>  
</mvc:interceptors>

# 拦截器

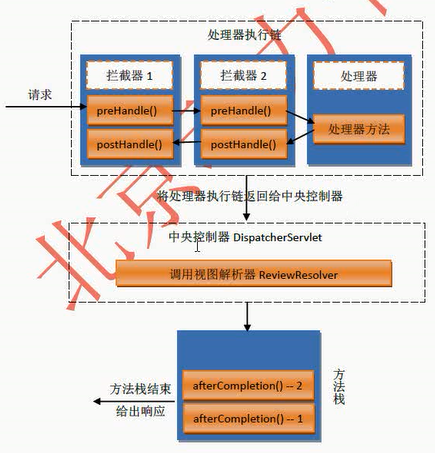
用于拦截用户请求，其拦截时间点：在处理器映射器根据用户提交的请求映射到所要执行的处理器类，并且也找到了要执行处理器类的处理器适配器，在处理器适配器执行处理器类之前

自定义处理器：实现HandlerInterceptor接口，

拦截器执行流程：



多个拦截器和处理器方法执行流程：



# 结合SpringBoot开发RESTful服务

@RestController标明该Controller提供RestAPI

@PathVariable映射url片段到java方法的参数。

@PageableDefault

@JsonView

@RequestBody映射请求体到java方法参数

测试类：

|  |
| --- |
| @Autowired private WebApplicationContext wac ;  private MockMvc mockMvc ;  @Before public void setUp() {  mockMvc = MockMvcBuilders.*webAppContextSetup*(wac).build(); } |

分页查询：

|  |
| --- |
| */\*\*  \* 分页查询  \** ***@param*** *condition  \** ***@param*** *pageable  \** ***@return*** *\*/* @RequestMapping(value="/user4",method = RequestMethod.*GET*) public List<User> query(UserQueryCondition condition,@PageableDefault(page=2,size=10,sort="age,desc") Pageable pageable) {  System.*out*.println(ReflectionToStringBuilder.*toString*(condition, ToStringStyle.*MULTI\_LINE\_STYLE*));  System.*out*.println("size:"+pageable.getPageSize());  System.*out*.println("pageNumber:"+pageable.getPageNumber());  System.*out*.println("sort:"+pageable.getSort());  return getUsers(); } |

JsonView使用：

|  |
| --- |
| */\*\*  \* 测试JsonView用例  \** ***@param*** *id  \** ***@return*** *\*/* @RequestMapping(value="/userInfo/{id:\\d+}",method = RequestMethod.*GET*) public User getUserInfo(@PathVariable() int id) {  User user = new User();  user.setUsrename("LiSaiHao");  return user ; } |
| @Test public void getUserInfoTest() throws Exception {  String result = mockMvc.perform(MockMvcRequestBuilders.*get*("/userInfo/20")  .contentType(MediaType.*APPLICATION\_JSON\_UTF8*))  .andExpect(MockMvcResultMatchers.*status*().isOk())  .andExpect(MockMvcResultMatchers.*jsonPath*("$.length").value(3))  .andReturn().getResponse().getContentAsString();  System.*out*.println("返回结果是："+result); } |