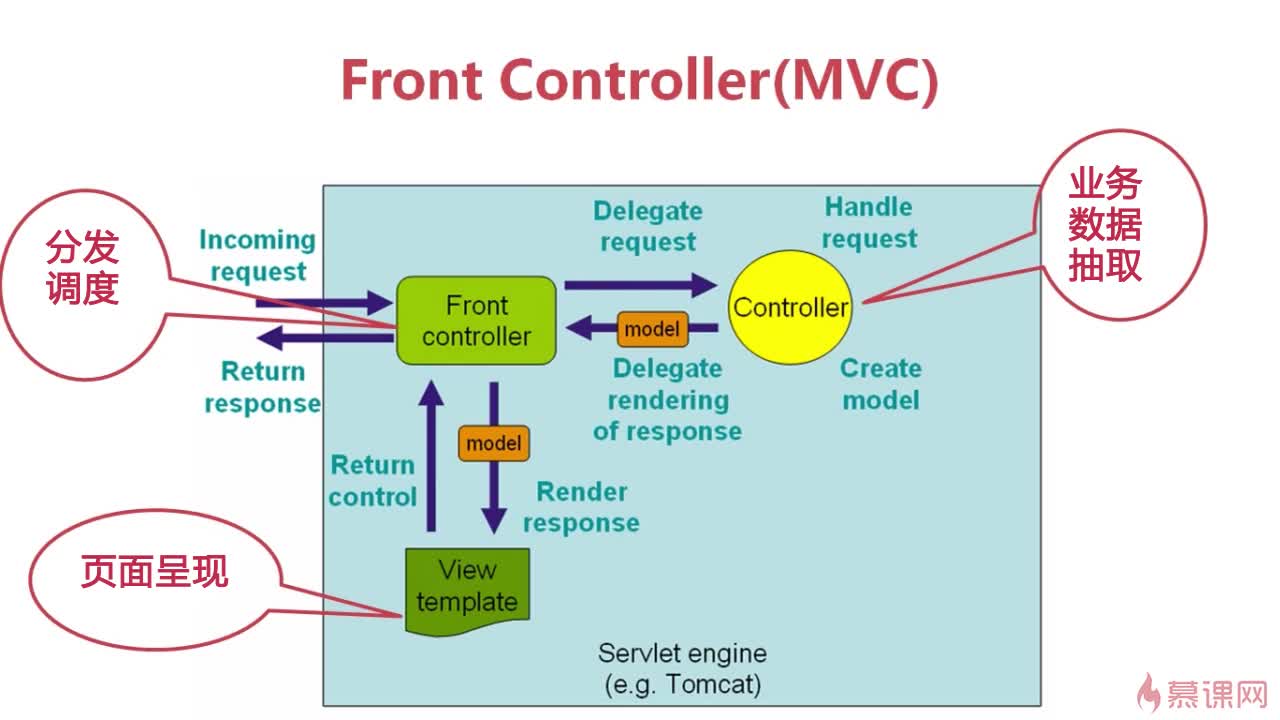
Spring MVC使用详解

Spring为展现层提供的基于MVC设计理念的优秀的Web框架，是目前最主流的MVC框架之一。Spring3.0后全面超越Struts2，成为最优秀的MVC框架Spring MVC通过一套MVC注解，让POJO成为处理请求的控制器，而无须实现任何接口。支持REST风格的URL请求采用了松散耦合可插拔组件结构，比其他MVC框架更具扩展性和灵活性。



MVC--Model-View-Controller

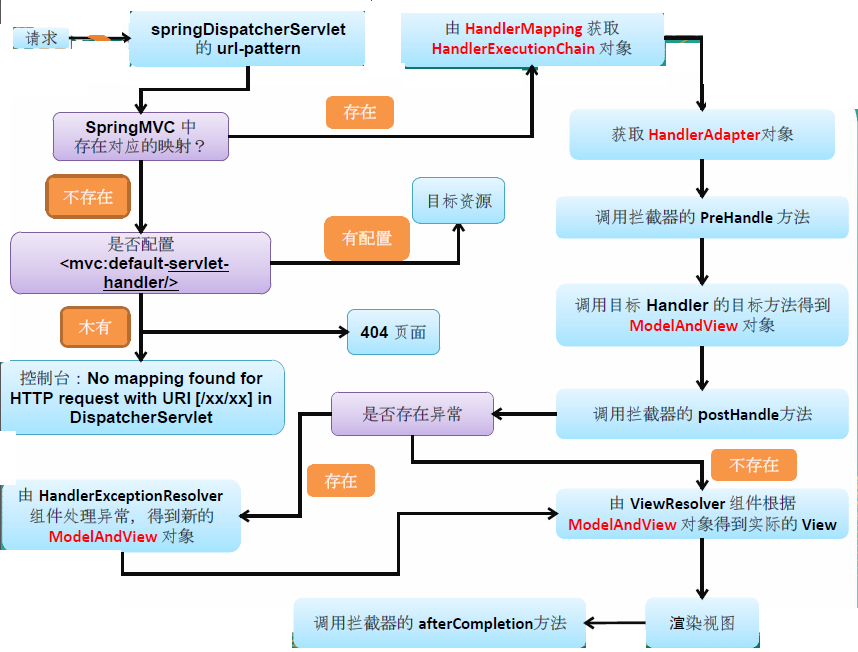
**Model(模型层)**：业务数据的信息表示，关注支撑业务的信息构成，通常是多个业务实体的组合;

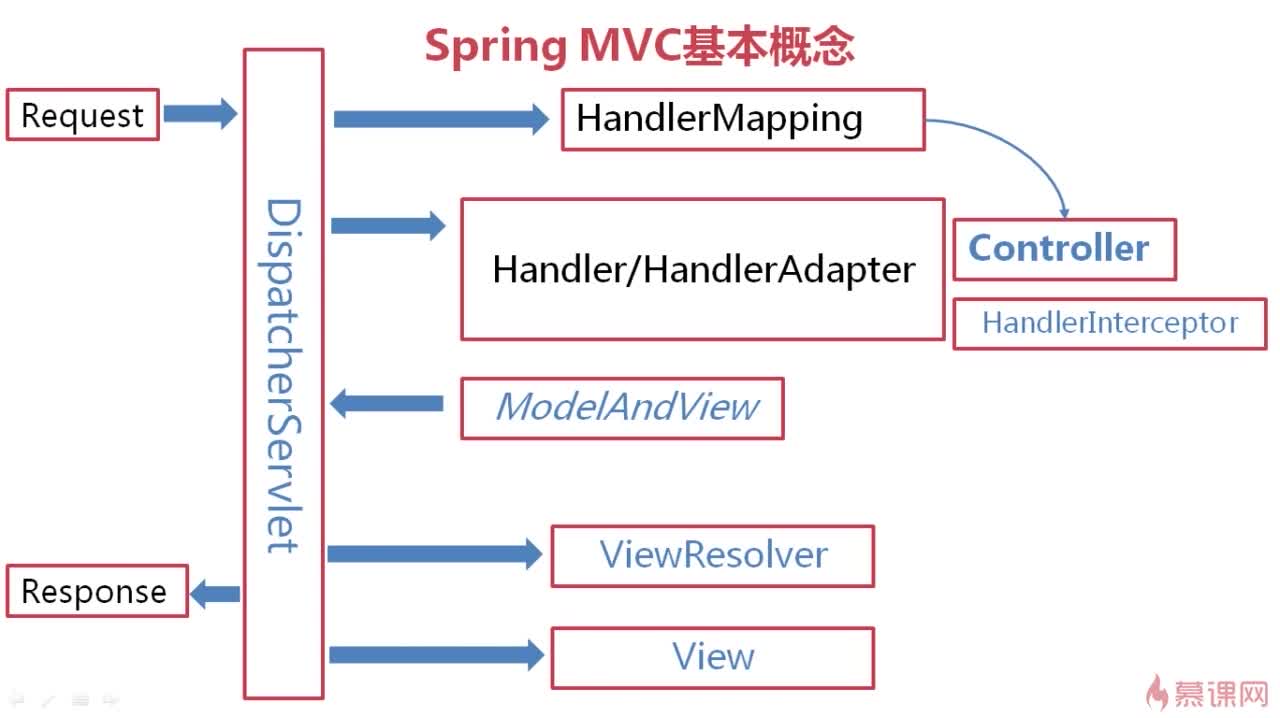
**View(视图层)**：为用户提供UI,重点关注数据的呈现；

**Controller(控制层)**：调用业务逻辑产生合适的数据(Model),传递数据给视图层用于呈现.

MVC是一种架构模式：程序分层，分工合作，既相互独立，又协同工作

# SpringMVC详细运行流程图





第一步：客户端请求提交到DispatcherServlet；

第二步：DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller；

第三步：DispatcherServlet将请求提交到Controller；Controller调用业务逻辑处理后，返回ModelAndView；

第四步：DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图；视图负责将结果显示到客户端。

# SpringMVC配置解析

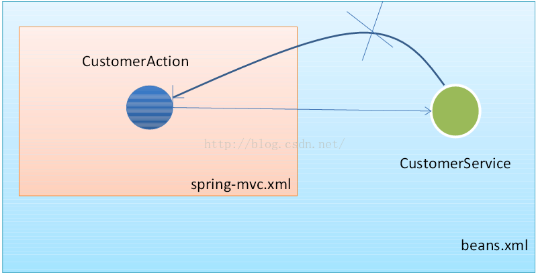
## 配置说明

|  |
| --- |
| **#web.xml的配置说明**  **<!-- configure the springIOC -->**  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  **<!--configure the setting of springmvcDispatcherServlet and configure the mapping-->**  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc-servlet.xml</param-value>  </init-param>  <!-- <load-on-startup>1</load-on-startup> -->  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |
| **#** **Spring IOC配置**  <!--防止冲突，spring IOC中的bean服务于所有的dispatchServlet容器-->  <context:component-scan base-package="test.SpringMVC.integrate">  <context:exclude-filter type="annotation"  expression="org.springframework.stereotype.Controller"/>  <context:exclude-filter type="annotation"  expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan> |
| **#** **Spring MVC配置**  <!-- scan the package and the sub package -->  <context:component-scan base-package=" ">  <context:include-filter type="annotation"  expression="org.springframework.stereotype.Controller"/>  <context:include-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan>  <!-- don't handle the static resource -->  <mvc:default-servlet-handler />  <!-- if you use annotation you must configure following setting -->  <mvc:annotation-driven />  <!-- configure the InternalResourceViewResolver -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"  id="internalResourceViewResolver">  <!-- 前缀 -->  <property name="prefix" value="/WEB-INF/jsp/" />  <!-- 后缀 -->  <property name="suffix" value=".jsp" />  </bean> |

## spring MVC和IOC容器的冲突解决

在Spring MVC 配置文件中引用业务层的Bean

* 多个Spring IOC 容器之间可以设置为父子关系，以实现良好的解耦。
* Spring MVC WEB 层容器可作为"业务层"Spring 容器的子容器：即WEB 层容器可以引用业务层容器的Bean，而业务层容器却访问不到WEB 层容器的Bean



# SpringMVC请求处理(Controller)

## @RequestMapping注解说明

|  |
| --- |
| **@Controller**  public class HelloWorld {  /\*\*  \* 1. 使用 @RequestMapping 注解来映射请求的 URL(这里的请求就是：href="helloWorld")  \* 2. 返回值会通过视图解析器解析为实际的物理视图, 对于 InternalResourceViewResolver 视图解析器, 会做如下的解析:  \* 通过 prefix + returnVal + 后缀 这样的方式得到实际的物理视图, 然会做转发操作  \*  \* /WEB-INF/views/success.jsp  \*/  **@RequestMapping("/helloWorld")**  public String helloWorld(){  System.out.println("springMVC HelloWorld!");  return "success";  }  } |

spring MVC 使用@RequestMapping注解为控制器指定可以处理哪些URL 请求。在控制器的类**定义及方法定义**处都可标注@RequestMapping。DispatcherServlet截获请求后，就通过控制器上@RequestMapping提供的映射信息确定请求所对应的处理方法。

-- **类定义处**：提供初步的请求映射信息。相对于WEB 应用的根目录

-- **方法处**：提供进一步细分映射信息。相对于类定义处的URL。若类定义处未标注@RequestMapping，则方法处标记的URL 相对于WEB 应用的根目录

@RequestMapping除了可以使用请求URL映射请求外，还可以使用请求方法、请求参数及请求头映射请求**@RequestMapping的value、method、params及heads** 分别表示请求URL、请求方法、请求参数及请求头的映射条件，他们之间是与的关系，联合使用多个条件让请求映射更加精确化params和headers支持简单的表达式：

* param：表示请求必须包含名为param的请求参数
* !param：表示请求不能包含名为!param的请求参数
* param!= value： 表示请求包含名为param的请求参数，但其值不能为value
* {"param1=value1", "param2"}：请求必须包含名为param1 和param2 的两个请求参数，且param1 参数的值必须为value1

@RequestMapping映射的url还可以使用通配符：

* ?：匹配文件名中的一个字符
* \*：匹配文件名中的任意字符
* \*\*：\*\* 匹配多层路径

## 静态资源的处理

若将DispatcherServlet请求映射配置为/，则Spring MVC 将捕获WEB 容器的所有请求，包括静态资源的请求，SpringMVC会将他们当成一个普通请求处理，因找不到对应处理器将导致错误。

**采用<mvc:default-servlet-handler />方式**： <mvc:default-servlet-handler/> 将在SpringMVC上下文中定义一个**DefaultServletHttpReq  
uestHandler**，它会对进入DispatcherServlet的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由WEB 应用服务器默认的**Servlet 处理**，如果不是静态资源的请求，才由DispatcherServlet继续处理。

**采用<mvc:resources />方式：**<mvc:default-servlet-handler />将静态资源的处理经由Spring MVC框架交回Web应用服务器处理。而<mvc:resour  
ces />更进一步，**由Spring MVC框架自己处理静态资源**，并添加一些有用的附加值功能。首先，<mvc:resources />允许静态资源放在任何地方，如WEB-INF目录下、类路径下等。其次，<mvc:resources />依据当前著名的Page Speed、YSlow等浏览器优化原则对静态资源提供优化。你可以通过cacheSeconds属性指定静态资源在浏览器端的缓存时间，一般可将该时间设置为一年，以充分利用浏览器端的缓存。在输出静态资源时，会根据配置设置好响应报文头的Expires 和 Cache-Control值。在接收到静态资源的获取请求时，会检查请求头的Last-Modified值，如果静态资源没有发生变化，则直接返回303相应状态码，提示客户端使用浏览器缓存的数据，而非将静态资源的内容输出到客户端，以充分节省带宽，提高程序性能。

|  |
| --- |
| <mvc:resources location="/,classpath:/META-INF/publicResources/" mapping="/resources/\*\*"/> |

# SpringMVC数据处理(Model)

## 各类参数绑定方式

|  |
| --- |
| **#基本数据类型，包装类型**  表单中的name值或者请求参数变量的key与controller方法中参数名一致  **#对象的绑定**  表单中的name值或者请求参数变量的key与controller方法中参数名一致，如果对象中还包含其他对象需要使用 "属性名(对象类型的属性).属性名"来命名input的name。  <form action="saysth.do" method="post">  <input name="firstName" value="张" /><br>  <input name="lastName" value="三" /><br>  <input name="contactInfo.tel" value="13809908909" /><br>  <input name="contactInfo.address" value="北京海淀" /><br>  <input type="submit" value="Save" />  </form>  **#list集合的绑定**  不能直接绑定集合，需要将集合封装到一个实体类中。通过 **"对象[index].属性"**来命名input的name。  <form action="saysth.do" method="post">  <input name="users[0].firstName" value="aaa" />  <input name="users[0].lastName" value="bbb" />  <input name="users[10].firstName" value="eee" />  <input name="users[10].lastName" value="fff" />  </form>  **#map集合数据的绑定**  不能直接绑定集合，需要将集合封装到一个实体类中。通过 **"对象["key"].属性"**来命名input的name。  private Map<String, User> users;  <form action="saysth.do" method="post">  <input name="users['x'].firstName" value="aaa" />  <input name="users['x'].lastName" value="bbb" />  <input name="users['y'].firstName" value="ccc" />  <input name="users['y'].lastName" value="ddd" />  </form> |

## 处理方法参数绑定

**请求处理方法中可以注入很多参数类型**：HttpServletRequest, HttpServlet  
Response, HttpSession, InputStream, OutputStream, Map, Model等。

**请求处理方法可返回参数**：ModelAndView, Model, Map, View, String(视图)

处理方法参数绑定常用的注解,我们根据处理Request的不同内容部分分为四种常见类型：

1. **处理requet uri 部分**（这里指uri template中variable，不含queryString部分）**的注解**： @PathVariable;
2. **处理request header部分注解：**@RequestHeader, @CookieValue;
3. **处理request body部分的注解**：@RequestParam, @RequestBody;
4. **处理attribute类型是注解**： @SessionAttributes,@ModelAttribute;

### @PathVariable注解

|  |
| --- |
| #使用@RequestMapping URI template 样式映射时， 即 someUrl/{paramId}, 这时的paramId可通过 @Pathvariable注解绑定它传过来的值到方法的参数上。  @Controller  @RequestMapping("/owners")  public class RelativePathUriTemplateController {    @RequestMapping("/pets/{petId}")  public void findPet**(@PathVariable String ownerId, @PathVariable String petId**, Model model) {  // implementation omitted  }  }  上面代码把URI template 中变量 petId的值，绑定到方法的参数上。若方法参数名称和需要绑定的uri template中变量名称不一致，需要在@PathVariable("name")指定uri template中的名称。 |

### @RequestHeader, @CookieValue

|  |
| --- |
| @RequestHeader 注解,可以把Request请求header部分的值绑定到方法参数上。  @RequestMapping("/displayHeaderInfo.do")  public void displayHeaderInfo**(@RequestHeader("Accept-Encoding") String encoding, @RequestHeader("Keep-Alive") long keepAlive**) {  }  @CookieValue 可以把Request header中关于cookie的值绑定到方法参数上。  @RequestMapping("/displayHeaderInfo.do")  public void displayHeaderInfo**(@CookieValue("JSESSIONID") String cookie**) {  } |

### @RequestParam, @RequestBody

**@RequestParam** ：**A)** 常用来处理简单类型的绑定，通过**Request.getPa  
rameter()** 获取的String可直接转换为简单类型的情况（ String--> 简单类型的转换操作由ConversionService配置的转换器来完成）；因为使用request.  
getParameter()方式获取参数，所以可以处理get 方式中queryString的值，也可以处理post方式中 body data的值；**B）**用来处理Content-Type: 为 application/x-www-form-urlencoded编码的内容，提交方式GET、POST；

**C)** 该注解有两个属性： value、required； value用来指定要传入值的name名称，required用来指示参数是否必须绑定；

|  |
| --- |
| @RequestMapping(method = RequestMethod.GET)  public String setupForm(**@RequestParam("petId") int petId**, ModelMap model) {  Pet pet = this.clinic.loadPet(petId);  model.addAttribute("pet", pet);  return "petForm";  } |

**@RequestBody**：该注解常用来处理Content-Type: 不是application/x-www-form-urlencoded编码的内容，例如application/json, application  
/xml等；它是通过使用HandlerAdapter 配置的**HttpMessageConverters**来解析post data body，然后绑定到相应的bean上的。因为配置有Form  
HttpMessageConverter，所以也可以用来处理 application/x-www-form-urlencoded的内容，处理完的结果放在一个MultiValueMap<String, String>里。示例代码：

|  |
| --- |
| @RequestMapping(value = "/something", method = RequestMethod.PUT)  public void handle(**@RequestBody String body**, Writer writer) throws IOException {  writer.write(body);  } |

### @SessionAttributes, @ModelAttribute

**@SessionAttributes**：该注解用来绑定HttpSession中的attribute对象的值，便于在方法中的参数里使用。该注解有value、types两个属性，可以通过名字和类型指定要使用的attribute 对象；

|  |
| --- |
| @Controller  @RequestMapping("/editPet.do")  **@SessionAttributes("pet")**  public class EditPetForm {  // ...  } |

**@ModelAttribute**：SpringMVC在每次调用请求处理方法的时候，都会创建Model类型的一个实例。如果打算使用该实例，只需要在方法中添加一个Model类型的参数。我们也可以使用@ **ModelAttribute注解。**该注解有两个用法，一个是用于方法上，一个是用于参数上；**用于方法上时**： 通常用来在处理@RequestMapping之前，为请求绑定需要从后台查询的model；**用于参数上时**： 用来通过名称对应，把相应名称的值绑定到注解的参数bean上；要绑定的值来源于：

A） @SessionAttributes 启用的attribute 对象上；

B） @ModelAttribute 用于方法上时指定的model对象；

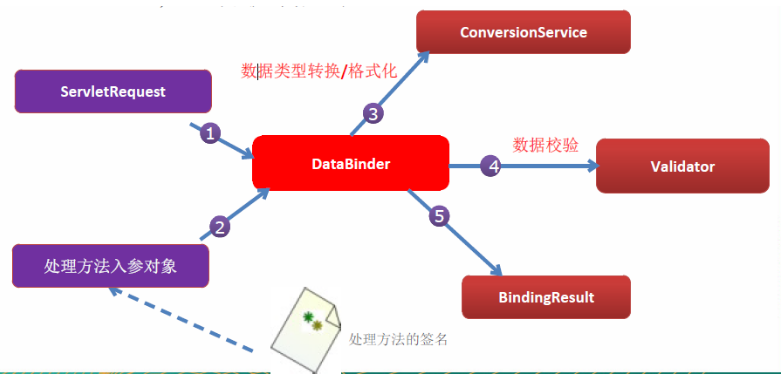
C） 上述两种情况都没有时，new一个需要绑定的bean对象，然后把request中按名称对应的方式把值绑定到bean中。

|  |
| --- |
| **@ModelAttribute**  public Account addAccount(@RequestParam String number) {  return accountManager.findAccount(number);  }  这种方式实际的效果就是在调用@RequestMapping的方法之前，为request对象的model里put（"account"， Account）；**返回值会自动添加到Model中**。  @RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)  public String processSubmit**(@ModelAttribute Pet pet)** {  }  首先查询 @SessionAttributes有无绑定的Pet对象，若没有则查询@ModelAttribute方法层面上是否绑定了Pet对象，若没有则将URI template中的值按对应的名称绑定到Pet对象的各属性上。 |

## 数据转换 & 数据格式化 & 数据校验

**数据绑定流程**：1) Spring MVC 主框架将 ServletRequest 对象及目标方法的入参实例传递给 WebDataBinderFactory 实例，以创建 DataBinder 实例对象 2) DataBinder 调用装配在 Spring MVC 上下文中的 ConversionService 组件进行数据类型转换、数据格式化工作。将 Servlet 中的请求信息填充到入参对象中3) 调用 Validator 组件对已经绑定了请求消息的入参对象进行数据合法性校验，并最终生成数据绑定结果 BindingData 对象 4) Spring MVC 抽取 BindingResult 中入参对象和校验错误对象，将它们赋给处理方法的响应入参。

Spring MVC 通过反射机制对目标处理方法进行解析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 DataBinder，运行机制如下：



### 数据转换

Spring MVC 上下文中内建了很多转换器，可完成大多数 Java 类型转换工作。

**自定义类型转换器：**ConversionService 是 Spring 类型转换体系的核心接口。可以利用 ConversionServiceFactoryBean 在 Spring 的 IOC 容器中定义一个 ConversionService. Spring 将自动识别出IOC 容器中的 ConversionService，并在 Bean 属性配置及Spring MVC 处理方法入参绑定等场合使用它进行数据的转换。可通过 ConversionServiceFactoryBean 的 converters 属性注册自定义的类型转换器

|  |
| --- |
| **//会将自定义的 ConversionService 注册到Spring MVC 的上下文中**  <mvc:annotation-driven conversion-service="conversionService">  </mvc:annotation-driven>    **<!-- 配置 ConversionService -->**  <bean id="conversionService"  class="org.springframework.format.support.FormattingConversionServiceFactoryBean">  <property name="converters">  <set>  <ref bean="employeeConverter"/>  </set>  </property>  </bean> |

Spring 定义了 3 种类型的转换器接口，实现任意一个转换器接口都可以作为自定义转换器注册到ConversionServiceFactroyBean 中：

* **Converter<S,T>**：将 S 类型对象转为 T 类型对象
* **ConverterFactory**：将相同系列多个 "同质" Converter 封装在一起。如果希望将一种类型的对象转换为另一种类型及其子类的对象（例如将 String 转换为 Number 及 Number 子类（Integer、Long、Double 等）对象）可使用该转换器工厂类
* **GenericConverter**：会根据源类对象及目标类对象所在的宿主类中的上下文信息进行类型转换

|  |
| --- |
| **@Component**  public class EmployeeConverter implements Converter<String, Employee> {  @Override  public Employee convert(String source) {  if(source != null){  String [] vals = source.split("-");  //GG-gg@atguigu.com-0-105  if(vals != null && vals.length == 4){  String lastName = vals[0];  String email = vals[1];  Integer gender = Integer.parseInt(vals[2]);  Department department = new Department();  department.setId(Integer.parseInt(vals[3]));    Employee employee = new Employee(null, lastName, email, gender, department);  System.out.println(source + "--convert--" + employee);  return employee;  }  }  return null;  }  } |

**关于 mvc:annotation-driven**

<mvc:annotation-driven /> 会自动注册 RequestMappingHandler  
Mapping、RequestMappingHandlerAdapter 与ExceptionHandler  
ExceptionResolver 三个bean。

还将提供以下支持：

* 支持使用 ConversionService 实例对表单参数进行类型转换
* 支持使用 @NumberFormat annotation、@DateTimeFormat 注解完成数据类型的格式化
* 支持使用 @Valid 注解对 JavaBean 实例进行 JSR 303 验证
* 支持使用 @RequestBody 和 @ResponseBody 注解

### 数据格式化

对属性对象的输入/输出进行格式化，从其本质上讲依然属于 "类型转换" 的范畴。Spring 在格式化模块中定义了一个实现 ConversionService 接口的FormattingConversionService 实现类，该实现类扩展GenericConversion  
Service，因此它既具有类型转换的功能，又具有格式化的功能Formatting  
ConversionService 拥有一个FormattingConversionServiceFactroyBean 工厂类，后者用于在 Spring 上下文中构造前者FormattingConversionServ  
iceFactroyBean 内部已经注册了 :

* **NumberFormatAnnotationFormatterFactroy**：支持对数字类型的属性使用 @NumberFormat 注解
* **JodaDateTimeFormatAnnotationFormatterFactroy**：支持对日期类型的属性使用 @DateTimeFormat 注解

装配了 FormattingConversionServiceFactroyBean 后，就可以在 Spring MVC 入参绑定及模型数据输出时使用注解驱动了。

<mvc:annotation-driven/> 默认创建的ConversionService 实例即为**FormattingConversionServiceFactroyBean**.

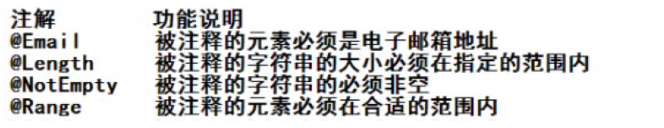
|  |
| --- |
| **-- 日期格式化，**@DateTimeFormat 注解可对java.util.Date、java.util.Calendar、java.long.Long 时间类型进行标注：  @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")  **private** Date createTime;  **--数值格式化，**@NumberFormat • 可对类似数字类型的属性进行标注，它拥有两个互斥的属性：style：类型为 NumberFormat.Style。用于指定样式类型，包括三种：Style.NUMBER（正常数字类型）、Style.CURRENCY（货币类型）、 Style.PERCENT（百分数类型）  pattern：类型为 String，自定义样式， 如patter="#,###"；  **@NumberFormat(pattern="#,###,###.#")**  private Float salary; |

### 数据校验

**JSR 303(Java验证规范)** ： Java 为 Bean 数据合法性校验提供的标准框架，它已经包含在 JavaEE 6.0 中 。JSR 303 通过在 Bean 属性上标注类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证。常用验证如下：

| **注解** | **描述** |
| --- | --- |
| @AssertFalse | 必须为 Boolean 类型且为 false |
| @AssertTure | 必须为 Boolean 类型且为 true |
| @DecimalMax | 数值必须小于或等于一个给定的 BigDecimalString 值 |
| @DecimalMin | 必须为数字且小于或等于一个给定的 BigDecimalString 的值 |
| @Digits | 必须为数字，且值必须为给定的数值 |
| @Future | 值必须为一个未来的日期 |
| @Max | 必须为数字，且值小于或等于给定的值 |
| @Min | 必须为数字，且值大于或等于给定的值 |
| @NotNull | 不能为 null |
| @Null | 必须为 null |
| @Past | 值必须为一个过去的日期 |
| @Pattern | 值必须满足给定的正则表达式 |
| @Size | 必须为 String，集合或数组的一种，且长度需满足给定的范围 |

**Hibernate Validator** 是 JSR 303 的一个参考实现，除支持所有标准的校验注解外，它还支持以下的扩展注解。



Spring 4.0 拥有自己独立的数据校验框架，同时支持 JSR 303 标准的校验框架。Spring 进行数据绑定时，可同时调用校验框架完成数据校验工作。在 Spring MVC 中，可直接通过注解驱动的方式进行数据校验。Spring 的 Local  
ValidatorFactroyBean 既实现了 Spring Validator 接口，也实现了 JSR 303 的 Validator 接口。只要在 Spring 容器中定义了一个LocalValidatorFactory  
Bean，即可将其注入到需要数据校验的 Bean 中**。Spring 本身并没有提供 JSR303 的实现，所以必须将JSR303 的实现者的 jar 包放到类路径下。**

<mvc:annotation-driven/> 会默认装配好一个 LocalValidatorFactoryBean，通过在处理方法的入参上标注 @valid 注解即可让 Spring MVC 在完成数据绑定后执行数据校验的工作。如何校验 ? 注解 ?

①. 使用 JSR 303 验证标准

②. 加入 hibernate validator 验证框架的 jar 包

③. 在 SpringMVC 配置文件中添加 <mvc:annotation-driven />

④. 需要在 bean 的属性上添加对应的注解

⑤. 在目标方法 bean 类型的前面添加 @Valid 注解

|  |
| --- |
| <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>5.2.2.Final</version>  </dependency> |

**注意**：需校验的 Bean 对象和其绑定结果对象或错误对象时成对出现的，它们之间不允许声明其他的入参。

|  |
| --- |
| @RequestMapping(value="/emp", method=RequestMethod.POST)  public String save(@Valid Employee employee, Errors result,  Map<String, Object> map){  System.out.println("save: " + employee);  if(result.getErrorCount() > 0){  System.out.println("出错了!");  for(FieldError error:result.getFieldErrors()){  System.out.println(error.getField() + ":" + error.getDefaultMessage());  }  //若验证出错, 则转向定制的页面  map.put("departments", departmentDao.getDepartments());  return "input";  }  employeeDao.save(employee);  return "redirect:/emps";  } |

Spring MVC 除了会将表单/命令对象的校验结果保存到对应的 BindingResult 或 Errors 对象中外，还会将所有校验结果保存到 "隐含模型"。隐含模型中的所有数据最终将通过 HttpServletRequest 的属性列表暴露给 JSP 视图对象。

## message国际化信息展示

默认情况下，SpringMVC根据Accept-Language参数判断客户端的本地化类型。当接受到请求时，SpringMVC会在上下文中查找一个本地化解析器（LocalResolver），找到后使用它获取请求所对应的本地化类型信息。Spr  
ingMVC还允许装配一个动态更改本地化类型的拦截器，这样通过指定一个请求参数就可以控制单个请求的本地化类型。

|  |
| --- |
| //读取国际化属性文件（命名：basename\_languageCode\_CountryCode）  //没有找到文件，默认读取basename.properties文件  //languageCode：zh,en ；CountryCode：CN,US  <bean id="messageSource"  class="org.springframework.context.support.ReloadableResourceBundleMessageSource">  <property name="basenames" >  <list>  <value>/WEB-INF/resource/messages</value>  <value>/WEB-INF/resource/labels</value>  </list>  </property>  </bean>  //  <bean id="localeResolver"  class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">  </bean> |
| //使用spring message来展示国际化错误信息  <input id="reset" type="reset" tabindex="4"  value="<spring:message code="button.reset"/>"> |

## 转发和重定向

|  |
| --- |
| **转发**：return "forward:xxx";  **重定向**：return "redirect: xxx";  **//重定向传值**  RequestMapping(value = "/saveProduct", method = RequestMethod.POST)  public String saveProduct(Product product, RedirectAttributes redirectAttributes) {  .....  redirectAttributes.**addFlashAttribute**("product", product);  return "redirect:product\_view";  } |

# SpringMVC视图解析（View）

当我们对SpringMVC控制的资源发起请求时，这些请求都会被Spring  
MVC的DispatcherServlet处理，接着Spring会分析看哪一个HandlerMap  
ping定义的所有请求映射中存在对该请求的最合理的映射。然后通过该Hand  
lerMapping取得其对应的Handler，接着再通过相应的HandlerAdapter处理该Handler。HandlerAdapter在对Handler进行处理之后会返回一个ModelAndView对象。在获得了ModelAndView对象之后，Spring就需要把该View渲染给用户，即返回给浏览器。在这个渲染的过程中，发挥作用的就是ViewResolver和View。当Handler返回的ModelAndView中不包含真正的视图，只返回一个逻辑视图名称的时候，ViewResolver就会把该逻辑视图名称解析为真正的视图View对象。View是真正进行视图渲染，把结果返回给浏览器的。

## ViewResolver和View介绍

SpringMVC用于处理视图最重要的两个接口是ViewResolver和View。**ViewResolver**的主要作用是把一个逻辑上的视图名称**解析为一个真正的视图**，SpringMVC中用于把View对象呈现给客户端的是View对象本身，而**View  
Resolver只是把逻辑视图名称解析为对象的View对象**。**View接口**的主要作用是用于**处理视图，然后返回给客户端**。Spring为我们提供了非常多的视图解析器，下面将列举一些视图解析器。

**AbstractCachingViewResolver**：这是一个抽象类，这种视图解析器会把它曾经解析过的视图保存起来，然后每次要解析视图的时候先从缓存里面找，如果找到了对应的视图就直接返回，如果没有就创建一个新的视图对象，然后把它放到一个用于缓存的map中，接着再把新建的视图返回。使用这种视图缓存的方式可以把解析视图的性能问题降到最低。

**UrlBasedViewResolver**：它是对ViewResolver的一种简单实现，而且继承了AbstractCachingViewResolver，主要就是提供的一种拼接URL的方式来解析视图，它可以让我们通过prefix属性指定一个指定的前缀，通过suffix属性指定一个指定的后缀，然后把返回的逻辑视图名称加上指定的前缀和后缀就是指定的视图URL了。如prefix=/WEB-INF/jsps/，suffix=.jsp，返回的视图名称viewName=test/indx，则UrlBasedViewResolver解析出来的视图URL就是/WEB-INF/jsps/test/index.jsp。默认的prefix和suffix都是空串。URLBasedViewResolver支持返回的视图名称中包含redirect:前缀，这样就可以支持URL在客户端的跳转，如当返回的视图名称是"redirect:test.do"的时候，URLBasedViewResolver发现返回的视图名称包含"redirect:"前缀，于是把返回的视图名称前缀"redirect:"去掉，取后面的test.do组成一个Redirect  
View，RedirectView中将把请求返回的模型属性组合成查询参数的形式组合到redirect的URL后面，然后调用HttpServletResponse对象的send  
Redirect方法进行重定向。同样URLBasedViewResolver还支持forword:前缀，对于视图名称中包含forword:前缀的视图名称将会被封装成一个InternalResourceView对象，然后在服务器端利用RequestDispatcher的forword方式跳转到指定的地址。使用UrlBasedViewResolver的时候必须指定属性viewClass，表示解析成哪种视图，一般使用较多的就是InternalResourceView，利用它来展现jsp，但是当我们使用JSTL的时候我们必须使用JstlView。下面是一段UrlBasedViewResolver的定义，根据该定义，当返回的逻辑视图名称是test的时候，UrlBasedViewResolver将把逻辑视图名称加上定义好的前缀和后缀，即"/WEB-INF/test.jsp"，然后新建一个viewClass属性指定的视图类型予以返回，即返回一个url为"/WEB-INF/test  
.jsp"的InternalResourceView对象。

|  |
| --- |
| <bean  class="org.springframework.web.servlet.view.UrlBasedViewResolver">  <property name="prefix" value="/WEB-INF/" />  <property name="suffix" value=".jsp" />  <property name="viewClass" value="org.springframework.web.servlet.view.InternalResourceView"/>  </bean> |

**InternalResourceViewResolver**：它是URLBasedViewResolver的子类，所以URLBasedViewResolver支持的特性它都支持。在实际应用中InternalResourceViewResolver也是使用的最广泛的一个视图解析器。那么InternalResourceViewResolver有什么自己独有的特性呢？单从字面意思来看，我们可以把InternalResourceViewResolver解释为内部资源视图解析器，这就是InternalResourceViewResolver的一个特性。InternalResource  
ViewResolver会把返回的视图名称都解析为InternalResourceView对象，InternalResourceView会把Controller处理器方法返回的模型属性都存放到对应的request属性中，然后通过RequestDispatcher在服务器端把请求forword重定向到目标URL。比如在InternalResourceViewResolver中定义了prefix=/WEB-INF/，suffix=.jsp，然后请求的Controller处理器方法返回的视图名称为test，那么这个时候InternalResourceViewResolver就会把test解析为一个InternalResourceView对象，先把返回的模型属性都存放到对应的HttpServletRequest属性中，然后利用RequestDispatcher在服务器端把请求forword到/WEB-INF/test.jsp。这就是InternalResourceView  
Resolver一个非常重要的特性，我们都知道存放在/WEB-INF/下面的内容是不能直接通过request请求的方式请求到的，为了安全性考虑，我们通常会把jsp文件放在WEB-INF目录下，而InternalResourceView在服务器端跳转的方式可以很好的解决这个问题。下面是一个InternalResourceViewResolver的定义，根据该定义当返回的逻辑视图名称是test的时候，InternalResource  
ViewResolver会给它加上定义好的前缀和后缀，组成"/WEB-INF/test.jsp"的形式，然后把它当做一个InternalResourceView的url新建一个Internal  
ResourceView对象返回。

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

**FreeMarkerViewResolver、VolocityViewResolver**：这两个视图解析器都是UrlBasedViewResolver的子类。FreeMarkerViewResolver会把Con  
troller处理方法返回的逻辑视图解析为FreeMarkerView，而VolocityView  
Resolver会把返回的逻辑视图解析为VolocityView。因为这两个视图解析器类似，所以这里我就只挑FreeMarkerViewResolver来做一个简单的讲解。FreeMarkerViewResolver和VilocityViewResolver都继承了UrlBased  
ViewResolver。对于FreeMarkerViewResolver而言，它会按照UrlBased  
ViewResolver拼接URL的方式进行视图路径的解析。但是使用FreeMarker  
ViewResolver的时候不需要我们指定其viewClass，因为FreeMarkerView  
Resolver中已经把viewClass定死为FreeMarkerView了。我们先在Spring  
MVC的配置文件里面定义一个FreeMarkerViewResolver视图解析器，并定义其解析视图的order顺序为0。

|  |
| --- |
| <bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver"  p:prefix="/" p:suffix=".ftl">  <property name="contentType" value="text/html;charset=UTF-8"></property>  <property name="requestContextAttribute" value="base"></property>  <property name="order" value="0"></property>  </bean> |

那么当我们请求的处理器方法返回一个逻辑视图名称viewName的时候，就会被该视图处理器加上前后缀解析为一个url为"viewName.ftl"的FreeM  
arkerView对象。对于FreeMarkerView**我们需要给定一个FreeMarkerCo  
nfig的bean对象来定义FreeMarker的配置信息**。FreeMarkerConfig是一个接口，Spring已经为我们提供了一个实现，它就是FreeMarkerConfig  
urer。我们可以通过在SpringMVC的配置文件里面定义该bean对象来定义FreeMarker的配置信息，该配置信息将会在FreeMarkerView进行渲染的时候使用到。对于FreeMarkerConfigurer而言，我们最简单的配置就是配置一个templateLoaderPath，告诉Spring应该到哪里寻找FreeMarker的模板文件。这个templateLoaderPath也支持使用"classpath:"和"file:"前缀。当FreeMarker的模板文件放在多个不同的路径下面的时候，我们可以使用templateLoaderPaths属性来指定多个路径。在这里我们指定模板文件是放在"/WEB-INF/freemarker/ "下面的。

|  |
| --- |
| <bean id="freemarkerConfigurer" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">  <property name="templateLoaderPath" value="/WEB-INF/freemarker/" />  <property name="defaultEncoding" value="UTF-8" />  <property name="freemarkerSettings">  <props>  <prop key="template\_update\_delay">0</prop>  <prop key="default\_encoding">UTF-8</prop>  <prop key="number\_format">0.##########</prop>  <prop key="datetime\_format">yyyy-MM-dd HH:mm:ss</prop>  <prop key="classic\_compatible">true</prop>  <prop key="tag\_syntax">auto\_detect</prop>  <prop key="whitespace\_stripping">true</prop>  <prop key="template\_exception\_handler">ignore</prop>  </props>  </property>  <property name="freemarkerVariables">  <map>  <entry key="context" value="#{servletContext.contextPath}" />  <entry key="cusVar" value="${cusVar}" />  </map>  </property>  </bean>  <bean  class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">  <property name="suffix" value=".ftl" />  <property name="contentType" value="text/html;charset=UTF-8" />  <property name="requestContextAttribute" value="base" />  <property name="order" value="1" />  </bean> |

## 视图解析器链

在SpringMVC中可以同时定义多个ViewResolver视图解析器，然后它们会组成一个ViewResolver链。当Controller处理器方法返回一个逻辑视图名称后，ViewResolver链将根据其中ViewResolver的优先级来进行处理。所有的ViewResolver都实现了Ordered接口，在Spring中实现了这个接口的类都是可以排序的。在ViewResolver中是通过order属性来指定顺序的，默认都是最大值。所以我们可以通过指定ViewResolver的order属性来实现Vie  
wResolver的优先级，order属性是Integer类型，order越小，对应的View  
Resolver将有越高的解析视图的权利，所以第一个进行解析的将是ViewResol  
ver链中order值最小的那个。当一个ViewResolver在进行视图解析后返回的View对象是null的话就表示该ViewResolver不能解析该视图，这个时候如果还存在其他order值比它大的ViewResolver就会调用剩余的ViewResol  
ver中的order值最小的那个来解析该视图，依此类推。当ViewResolver进行视图解析后返回的是一个非空的View对象的时候，就表示该ViewResolver能够解析该视图，那么视图解析这一步就完成了，后续的ViewResolver将不会再用来解析该视图。当定义的所有ViewResolver都不能解析该视图的时候，Spring就会抛出一个异常。

基于Spring支持的这种ViewResolver链模式，我们就可以在Spring  
MVC应用中同时定义多个ViewResolver，给定不同的order值，这样我们就可以对特定的视图特定处理，以此来支持同一应用中有多种视图类型。**注意**：像InternalResourceViewResolver这种能解析所有的视图，即永远能返回一个非空View对象的ViewResolver一定要把它放在ViewResolver链最后。

|  |
| --- |
| <bean id="freemarkerConfigurer"  class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">  <property name="templateLoaderPath" value="/WEB-INF/freemarker/" />  <property name="defaultEncoding" value="UTF-8" />  <property name="freemarkerSettings">  <props>  <prop key="template\_update\_delay">0</prop>  <prop key="default\_encoding">UTF-8</prop>  <prop key="number\_format">0.##########</prop>  <prop key="datetime\_format">yyyy-MM-dd HH:mm:ss</prop>  <prop key="classic\_compatible">true</prop>  <prop key="tag\_syntax">auto\_detect</prop>  <prop key="whitespace\_stripping">true</prop>  <prop key="template\_exception\_handler">ignore</prop>  </props>  </property>  <property name="freemarkerVariables">  <map>  <entry key="context" value="#{servletContext.contextPath}" />  <entry key="cusVar" value="${cusVar}" />  </map>  </property>  </bean>  <bean  class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">  <property name="suffix" value=".ftl" />  <property name="contentType" value="text/html;charset=UTF-8" />  <property name="requestContextAttribute" value="base" />  <property name="order" value="1" />  </bean>  <!-- 配置视图解析器: 如何把 handler 方法返回值解析为实际的物理视图 -->  <bean id="viewResolver"  class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/jsp/" />  <property name="suffix" value=".jsp" />  <property name="order" value="2" />  </bean> |

# SpringMVC异常处理

## 处理局部异常（Controller内）

|  |
| --- |
| **@ExceptionHandler(Exception.class)**  public ModelAndView exceptionHandler(Exception ex){  ModelAndView mv = new ModelAndView("error");  mv.addObject("exception", ex);  return mv;  }  @RequestMapping("/error")  public String error(){  int i = 5/0;  return "hello";  } |

## 处理全局异常（所有Controller）

@ControllerAdvice注解本身已经使用了@Component。

|  |
| --- |
| @ControllerAdvice  public class testControllerAdvice {  @ExceptionHandler  public ModelAndView exceptionHandler(Exception ex){  ModelAndView mv = new ModelAndView("error");  mv.addObject("exception", ex);  return mv;  }  } |

# SpringMVC文件上传和下载

## 文件上传

|  |
| --- |
| #maven依赖  <!-- 文件上传 -->  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.3.2</version>  </dependency> |
| //spring 装配bean multipartResolver  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <property name="maxUploadSize" value="10000000" />  <property name="resolveLazily" value="true" />  </bean> |
| #MultipartFile接口详解  byte[] getBytes()：以字节数组的形式返回文件的内容。  String getContentType()：返回文件内容类型。  InputStream getInputStream()：返回inputStream，从中可以读取内容  String getName()：以多部分形式返回参数名称  String getOriginalFilename()：获取本地初始文件名  long getSize()：返回文件的大小，字节为单位。  boolean isEmpty()：判断上传的文件是否为空。  void transferTo(File destination)：它将上传的文件保存到目标路径下。 |
| @RequestMapping(value = "/save-product")  public String saveProduct(HttpServletRequest servletRequest, @ModelAttribute Product product,  BindingResult bindingResult, Model model) {  List<MultipartFile> files = product.getImages();  List<String> fileNames = new ArrayList<String>();  if (null != files && files.size() > 0) {  for (MultipartFile multipartFile : files) {  String fileName = multipartFile.getOriginalFilename();  fileNames.add(fileName);  File imageFile = new File(servletRequest.getServletContext().getRealPath("/WEB-INF/image"), fileName);  try {  multipartFile.transferTo(imageFile);  } catch (IOException e) {  e.printStackTrace();  }  }  }  } |

## 文件下载

步骤：1) 请求中添加HttpServletResponse-->2)将相应的内容类型设为未见的内容类型（Content-Type）-->3)添加一个名字为Content-Disposition的Http相应头，并赋值attachment; filename=fileName

|  |
| --- |
| @RequestMapping(value = "/export", method = RequestMethod.GET)  public void toExcle(HttpServletRequest request, **HttpServletResponse response**, EmployeeIncomePageDto pageDto)  throws UnsupportedEncodingException {  HSSFWorkbook wb = employeeIncomeService.toExcle(pageDto);  String excelName = "服务收入详情表.xls";  try {  **response.setContentType**("application/vnd.ms-excel;charset=utf-8");  **response.setHeader**("Content-disposition",  "attachment;filename=" + new String(excelName.getBytes("GB2312"), "ISO8859-1"));  **ServletOutputStream outputStream = response.getOutputStream();**  wb.write(outputStream);  outputStream.flush();  } catch (IOException e) {  logger.error(e.getMessage());  }  } |

# SpringMVC拦截器的使用

自定义拦截器：Spring MVC也可以使用拦截器对请求进行拦截处理，用户可以自定义拦截器来实现特定的功能，自定义的拦截器必须实现HandlerInter  
ceptor接口。要实现以下三个方法：

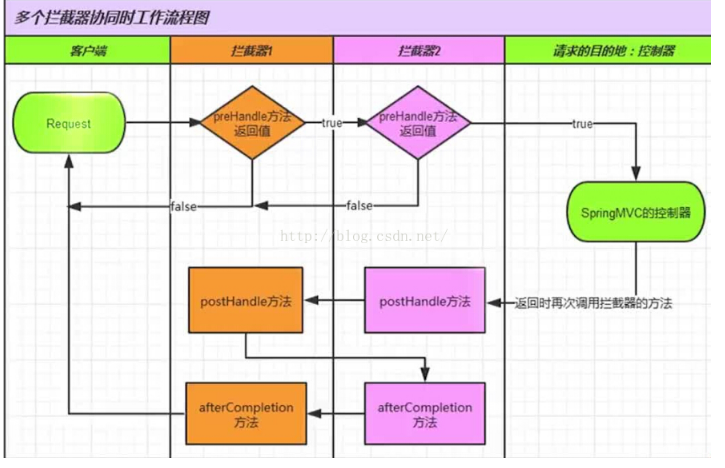
**preHandle()**：这个方法在业务处理器处理请求之前被调用，在该方法中对用户请求request 进行处理。如果程该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去进行处理，则返回true；如果决定不需要再调用其他的组件去处理请求，则返回false。

**postHandle()**：这个方法在业务处理器处理完请求后，但–是DispatcherServlet向客户端返回响应前被调用，在该方法中对用户请求request进行处理。

**afterCompletion()**：这个方法在DispatcherServlet完全处理完请求后被调用，可以在该方法中进行一些资源清理的操作。

## 拦截器的使用步骤

|  |
| --- |
| 第一步：编写拦截器类实现HandlerInterceptor接口，必须实现上面三个方法  第二步：2.将拦截器注册进SpringMVC框架中  <mvc:interceptors> <mvc:interceptor>  <mvc:mapping path="/mvc/\*\*"/> <bean class="test.SpringMVC.Interceptor.MyInterceptor"></bean> </mvc:interceptor> </mvc:interceptors> |



## 使用场景

全局的问题：字符串编码，用户登陆验证。

# Spring MVC创建RESTful API

## REST基础知识

Restful风格的API是一种软件架构风格，设计风格而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。REST的具体含义：

* **Representational(表述性)**：资源表述形式（xml,json,html等）
* **State(状态)**：使用REST，我们关注资源的状态和不是行为。
* **Transfer(转移)**：REST涉及转移资源数据，它以表述形式转移资源到另一个应用

在Restful风格中，用户请求的url使用同一个url而用请求方式：get，post，delete，put...等方式对请求的处理方法进行区分，这样可以在前后台分离式的开发中使得前端开发人员不会对请求的资源地址产生混淆和大量的检查方法名的麻烦，形成一个统一的接口。在Restful风格中，现有规定如下：

* **GET（SELECT）：**从服务器查询，可以在服务器通过请求的参数区分查询的方式。
* **POST（CREATE）：**在服务器新建一个资源，调用insert操作。
* **PUT（UPDATE）：**在服务器更新资源，调用update操作。
* **PATCH（UPDATE）：**在服务器更新资源（客户端提供改变的属性）。(目前jdk7未实现，tomcat7也不行)。
* **DELETE（DELETE）：**从服务器删除资源，调用delete语句。

**注意**：javaWeb部分版本不支持put.delete，统一使用post来请求。

## REST资源表述形式

**内容协商**：选择一个视图，它能够将模型渲染为呈现给客户端的表述形式；（确定请求的媒体类型--> 找到适合请求媒体类型的最佳视图）**基本不用**

**消息转换器**：通过一个消息转换器将控制器所返回的对象转换为呈献给客户端的表述形式（不会有视图）。Spring自带了各种各样的转换器，其中最常用的是json。假如客户端通过请求的Accept头信息表明它能接受"application/  
json"，并且Jackson JSON在类路径下，那么处理方法返回对象就将交给MappingJacksonHttpMessageConverter，并由它转换成返回客户端的JSON表述形式。

|  |
| --- |
| #maven依赖  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-core</artifactId>  <version>2.8.0</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.8.0</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-annotations</artifactId>  <version>2.8.0</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.jaxrs</groupId>  <artifactId>jackson-jaxrs-json-provider</artifactId>  <version>2.8.0</version>  </dependency>  #springMVC中json字符窜转换成日期需要加上注解  @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss",timezone = "GMT+8") |
| **在响应体中返回资源状态**  //使用@ResponseBody，控制器会将返回值转为json文档写到响应体中。  @RequestMapping(value="/getProduct",method=RequestMethod.GET)  public **@ResponseBody** ResultDto<Product> getProduct(){  ...  }  **在请求体中接受资源状态**  @requestBody参数：spring将会查看头部信息，并查找能够将请求主体转换为对应对象的消息转换器。  **为控制器默认设置消息转换**  @RESTController注解包含@Controller和@ResponseBody。spring将会为该控制器的所有处理方法应用消息转换功能。 |

## 异常处理

|  |
| --- |
| **//定义基本返回类型**  **public** **class** ResultDto<T> **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **int** code;  **private** String message;  **private** T data;  //set,get方法 |
| **//返回数据工具类**  **public** **class** ResultDtoFactory {  **public** **static** <T> ResultDto<T> toAck(T data) {  ResultDto<T> dto = **new** ResultDto<T>();  dto.setCode(HttpStatus.***OK***.value());  dto.setMessage("SUCCESS");  dto.setData(data);  **return** dto;  }  **public** **static** <T> ResultDto<T> toNack(**int** code, String msg) {  ResultDto<T> dto = **new** ResultDto<T>();  dto.setCode(code);  dto.setMessage(msg);  **return** dto;  }  } |
| **#全局异常处理，代码里面只管正常情况**  @ControllerAdvice  @ResponseBody  **public** **class** ExceptionAdvice {  **private** Logger logger = LoggerFactory.*getLogger*(ExceptionAdvice.**class**);    /\*\*  \* 404 - NOT\_FOUND  \*/  @ExceptionHandler(ResourceNotFoundException.**class**)  **public** <T> ResultDto<T> handleResourceNotFoundException(ResourceNotFoundException e) {  logger.error("资源未找到", e);  **return** ResultDtoFactory.*toNack*(HttpStatus.***NOT\_FOUND***.value(), "资源未找到");    }  /\*\*  \* 400 - Bad Request  \*/  @ExceptionHandler(HttpMessageNotReadableException.**class**)  **public** <T> ResultDto<T> handleHttpMessageNotReadableException(HttpMessageNotReadableException e) {  logger.error("参数解析失败", e);  **return** ResultDtoFactory.*toNack*(HttpStatus.***BAD\_REQUEST***.value(), "参数解析失败");    }  /\*\*  \* 405 - Method Not Allowed  \*/  @ResponseStatus(HttpStatus.***METHOD\_NOT\_ALLOWED***)  @ExceptionHandler(HttpRequestMethodNotSupportedException.**class**)  **public** <T> ResultDto<T> handleHttpRequestMethodNotSupportedException(HttpRequestMethodNotSupportedException e) {  logger.error("不支持当前请求方法", e);  **return** ResultDtoFactory.*toNack*(HttpStatus.***METHOD\_NOT\_ALLOWED***.value(), "不支持当前请求方法");    }  /\*\*  \* 415 - Unsupported Media Type  \*/  @ExceptionHandler(HttpMediaTypeNotSupportedException.**class**)  **public** <T> ResultDto<T> handleHttpMediaTypeNotSupportedException(Exception e) {  logger.error("不支持当前媒体类型", e);  **return** ResultDtoFactory.*toNack*(HttpStatus.***UNSUPPORTED\_MEDIA\_TYPE***.value(), "不支持当前媒体类型");  }  /\*\*  \* 500 - Internal Server Error  \*/  @ExceptionHandler(Exception.**class**)  **public** <T> ResultDto<T> handleException(Exception e) {  logger.error("服务运行异常", e);  **return** ResultDtoFactory.*toNack*(HttpStatus.***INTERNAL\_SERVER\_ERROR***.value(), "服务运行异常");  }  } |
| **//自定义资源未找到异常**  **public** **class** ResourceNotFoundException **extends** RuntimeException {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** String message;  **public** ResourceNotFoundException() {  }  **public** ResourceNotFoundException(String message) {  **super**(message);  }  } |