# <mvc:annotation-driven />

<mvc:annotation-driven /> 是一种简写形式，完全可以手动配置替代这种简写形式，简写形式可以让初学都快速应用默认配置方案。<mvc:annotation-driven /> 会自动注册DefaultAnnotationHandlerMapping与AnnotationMethodHandlerAdapter 两个bean,是spring MVC为@Controllers分发请求所必须的。

并提供了：数据绑定支持，@NumberFormatannotation支持，@DateTimeFormat支持，@Valid支持，读写XML的支持（JAXB），读写JSON的支持（Jackson）。

后面，我们处理响应ajax请求时，就使用到了对json的支持。

后面，对action写JUnit单元测试时，要从spring IOC容器中取DefaultAnnotationHandlerMapping与AnnotationMethodHandlerAdapter 两个bean，来完成测试，取的时候要知道是<mvc:annotation-driven />这一句注册的这两个bean。

<context:annotation-config> declares support for general annotations such as @Required, @Autowired, @PostConstruct, and so on.

<mvc:annotation-driven /> is actually rather pointless. It declares explicit support for annotation-driven MVC controllers (i.e.@RequestMapping, @Controller, etc), even though support for those is the default behaviour.

My advice is to always declare <context:annotation-config>, but don't bother with <mvc:annotation-driven /> unless you want JSON support via Jackson.

1:mvc annotation-driven 新增标签

以下为spring mvc 3.1中annotation-driven所支持的全部配置。

Xml代码 收藏代码

<mvc:annotation-driven message-codes-resolver ="bean ref" validator="" conversion-service="">

<mvc:return-value-handlers>

<bean></bean>

</mvc:return-value-handlers>

<mvc:argument-resolvers>

</mvc:argument-resolvers>

<mvc:message-converters>

</mvc:message-converters>[/color]

</mvc:annotation-driven>

其中3.1新增部分如下

return-value-handlers

允许注册实现了HandlerMethodReturnValueHandler接口的bean，来对handler method的特定的返回类型做处理。

HandlerMethodReturnValueHandler接口中定义了两个方法

supportsReturnType 方法用来确定此实现类是否支持对应返回类型。

handleReturnValue 则用来处理具体的返回类型。

例如以下的handlerMethod

Java代码 收藏代码

@RequestMapping("/testReturnHandlers")

public User testHandlerReturnMethod(){

User u = new User();

u.setUserName("test");

return u;

}

所返回的类型为一个pojo，正常情况下spring mvc无法解析，将转由DefaultRequestToViewNameTranslator 解析出一个缺省的view name，转到 testReturnHandlers.jsp，

我们增加以下配置

Xml代码 收藏代码

<mvc:annotation-driven validator="validator">

color=red] <mvc:return-value-handlers>

<bean class="net.zhepu.web.handlers.returnHandler.UserHandlers"></bean>

</mvc:return-value-handlers>[/color]

</mvc:annotation-driven>

及如下实现类

Java代码 收藏代码

public class UserHandlers implements HandlerMethodReturnValueHandler {

Logger logger = LoggerFactory.getLogger(this.getClass());

@Override

public boolean supportsReturnType(MethodParameter returnType) {

Class<?> type = returnType.getParameterType();

if(User.class.equals(type))

{

return true;

}

return false;

}

@Override

public void handleReturnValue(Object returnValue,

MethodParameter returnType, ModelAndViewContainer mavContainer,

NativeWebRequest webRequest) throws Exception {

logger.info("handler for return type users ");

mavContainer.setViewName("helloworld");

}

}

此时再访问 http://localhost:8080/springmvc/testReturnHandlers ，将交由 UserHandlers来处理返回类型为User的返回值。

argument-resolvers

允许注册实现了WebArgumentResolver接口的bean，来对handlerMethod中的用户自定义的参数或annotation进行解析

例如

Xml代码 收藏代码

<mvc:annotation-driven validator="validator">

<mvc:argument-resolvers>

<bean

class="net.zhepu.web.handlers.argumentHandler.MyCustomerWebArgumentHandler" />

</mvc:argument-resolvers>

</mvc:annotation-driven>

对应java代码如下

Java代码 收藏代码

public class MyCustomerWebArgumentHandler implements WebArgumentResolver {

@Override

public Object resolveArgument(MethodParameter methodParameter,

NativeWebRequest webRequest) throws Exception {

if (methodParameter.getParameterType().equals(MyArgument.class)) {

MyArgument argu = new MyArgument();

argu.setArgumentName("winzip");

argu.setArgumentValue("123456");

return argu;

}

return UNRESOLVED;

}

}

这里我们定义了一个 customer webArgumentHandler，当handler method中参数类型为 MyArgument时生成对参数的类型绑定操作。

注意新注册的webArgumentHandler的优先级最低，即如果系统缺省注册的ArgumentHandler已经可以解析对应的参数类型时，就不会再调用到新注册的customer ArgumentHandler了。

message-converters

允许注册实现了HttpMessageConverter接口的bean，来对requestbody 或responsebody中的数据进行解析

例如

假设我们使用 text/plain格式发送一串字符串来表示User对象，各个属性值使用”|”来分隔。例如 winzip|123456|13818888888，期望转为user对象，各属性内容为user.username = winzip,user.password=123456;user.mobileNO = 13818888888

以下代码中supports表示此httpmessageConverter实现类针对 User类进行解析。

构造函数中调用 super(new MediaType("text", "plain"));以表示支持 text/plain格式的输入。

Java代码 收藏代码

public class MyCustomerMessageConverter extends

AbstractHttpMessageConverter<Object> {

@Override

protected boolean supports(Class<?> clazz) {

if (clazz.equals(User.class)) {

return true;

}

return false;

}

public MyCustomerMessageConverter() {

super(new MediaType("text", "plain"));

}

@Override

protected Object readInternal(Class<? extends Object> clazz,

HttpInputMessage inputMessage) throws IOException,

HttpMessageNotReadableException {

Charset charset;

MediaType contentType = inputMessage.getHeaders().getContentType();

if (contentType != null && contentType.getCharSet() != null) {

charset = contentType.getCharSet();

} else {

charset = Charset.forName("UTF-8");

}

String input = FileCopyUtils.copyToString(new InputStreamReader(

inputMessage.getBody(), charset));

logger.info(input);

String[] s = input.split("\\|");

User u = new User();

u.setUserName(s[0]);

u.setPassword(s[1]);

u.setMobileNO(s[2]);

return u;

}

@Override

protected void writeInternal(Object t, HttpOutputMessage outputMessage)

throws IOException, HttpMessageNotWritableException {

}

修改servlet context xml配置文件，增加message-converters的相应配置如下。

Xml代码 收藏代码

<mvc:message-converters>

<bean class="net.zhepu.web.handlers.messageConverterHandler.MyCustomerMessageConverter"></bean>

</mvc:message-converters>

message-codes-resolver

先看看spring mvc中对于messageCodeResolver的用法。

spring mvc中使用DefaultMessageCodesResolver作为缺省的MessageCodesResolver的实现类，其作用是对valid errors中的errorcode进行解析。其解析方式如下

当解析error global object注册的errorcode时，errorcode的查找顺序为

1：errorcode.validationobjectname

2：errorcode

例如

以下声明中

Java代码 收藏代码

public String helloWorld2(@ModelAttribute("user") User u,

BindingResult result)

当使用 result.reject("testFlag");来注册一个globat error object时，spring mvc将在messageSource中先查找 testFlag.user这个errorcode,当找不到时再查找testFlag这个errorcode。

当解析fields error时，将按以下顺序生成error code

1.: code + "." + object name + "." + field

2.: code + "." + field

3.: code + "." + field type

4.: code

还是以上面的代码为例，当使用 result.rejectValue("userName", "testFlag");来注册一个针对user.UserName属性的错误描述时，errors对象中将生成以下的error code list,

1.: testFlag.user.userName

2.: testFlag.userName

3.: testFlag.java.lang.String

4.: testFlag

而mvc:annotation-driven新增的属性message-codes-resolver则提供了注册自定义的MessageCodesResolver的手段。

例如上面想要在所有的error code前增加前缀validation.的话，可以这么来做

Xml代码 收藏代码

<mvc:annotation-driven validator="validator" message-codes-resolver="messageCodeResolver">

</mvc:annotation-driven>

新增messageCodeResolver bean定义如下

Java代码 收藏代码

<bean id="messageCodeResolver" class="org.springframework.validation.DefaultMessageCodesResolver">

<property name="prefix" value="validation."></property>

</bean>

此时，所有的errorcode都会生成缺省前缀 validation.

例如前面的 result.reject("testFlag"); 生成的error code list就变为了

validation.testFlag.user 和 validation.testFlag了。

2: @RequestMapping 新增参数Consumes 和Produces

前面介绍过@RequestMapping的参数中有一个header的参数，来指定handler method能接受的http request 请求的header内容。

而consumes和produces则更进一步，直接指定所能接受或产生的request请求的content type。

例如

Java代码 收藏代码

@RequestMapping(value="/testMsgConverter",consumes="text/plain",produces="application/json")

表示handlermethod接受的请求的header中的 Content-Type为text/plain;

Accept为application/json

3: URI Template 新增功能

这部分的例子直接照抄Spring 3.1 M2: Spring MVC Enhancements中的示例

1: @PathVariable 声明的参数可自动加入到model中。

例如

Java代码 收藏代码

@RequestMapping("/develop/apps/edit/{slug}")

public String editForm(@PathVariable String slug, Model model) {

model.addAttribute("slug", slug);

// ...

}

现在可以写为

Java代码 收藏代码

@RequestMapping("/develop/apps/edit/{slug}")

public String editForm(@PathVariable String slug, Model model) {

// model contains "slug" variable

}

2：handler method中的redirect string可支持url template了

例如

Java代码 收藏代码

@RequestMapping(

value="/groups/{group}/events/{year}/{month}/{slug}/rooms",

method=RequestMethod.POST)

public String createRoom(

@PathVariable String group, @PathVariable Integer year,

@PathVariable Integer month, @PathVariable String slug) {

// ...

return "redirect:/groups/" + group + "/events/" + year + "/" + month + "/" + slug;

}

现在可写为

Java代码 收藏代码

@RequestMapping(

value="/groups/{group}/events/{year}/{month}/{slug}/rooms",

method=RequestMethod.POST)

public String createRoom(

@PathVariable String group, @PathVariable Integer year,

@PathVariable Integer month, @PathVariable String slug) {

// ...

return "redirect:/groups/{group}/events/{year}/{month}/{slug}";

}

3：url template中可支持databinding 了

例如

Java代码 收藏代码

@RequestMapping("/people/{firstName}/{lastName}/SSN")

public String find(Person person,

@PathVariable String firstName,

@PathVariable String lastName) {

person.setFirstName(firstName);

person.setLastName(lastName);

// ...

}

现在可以写成

Java代码 收藏代码

@RequestMapping("/people/{firstName}/{lastName}/SSN")

public String search(Person person) {

// person.getFirstName() and person.getLastName() are populated

// ...

}

4: Validation For @RequestBody

@RequestBody现在直接支持@valid标注了,如果validation失败，将抛出

RequestBodyNotValidException。

具体处理逻辑可见 spring 中的RequestResponseBodyMethodProcessor中的以下代码。

Java代码 收藏代码

public Object resolveArgument(MethodParameter parameter,

ModelAndViewContainer mavContainer,

NativeWebRequest webRequest,

WebDataBinderFactory binderFactory) throws Exception {

Object arg = readWithMessageConverters(webRequest, parameter, parameter.getParameterType());

if (shouldValidate(parameter, arg)) {

String argName = Conventions.getVariableNameForParameter(parameter);

WebDataBinder binder = binderFactory.createBinder(webRequest, arg, argName);

binder.validate();

Errors errors = binder.getBindingResult();

if (errors.hasErrors()) {

throw new RequestBodyNotValidException(errors);

}

}

return arg;

}

5:annotation-driven缺省注册类的改变

Spring 3.0.x中使用了annotation-driven后，缺省使用DefaultAnnotationHandlerMapping 来注册handler method和request的mapping关系。

AnnotationMethodHandlerAdapter来在实际调用handlermethod前对其参数进行处理。

并在dispatcherServlet中，当用户未注册自定义的ExceptionResolver时，注册AnnotationMethodHandlerExceptionResolver来对使用@ExceptionHandler标注的异常处理函数进行解析处理(这也导致当用户注册了自定义的exeptionResolver时将可能导致无法处理@ExceptionHandler)。

在spring mvc 3.1中，对应变更为

DefaultAnnotationHandlerMapping -> RequestMappingHandlerMapping

AnnotationMethodHandlerAdapter -> RequestMappingHandlerAdapter

AnnotationMethodHandlerExceptionResolver -> ExceptionHandlerExceptionResolver

以上都在使用了annotation-driven后自动注册。

而且对应分别提供了AbstractHandlerMethodMapping , AbstractHandlerMethodAdapter和 AbstractHandlerMethodExceptionResolver以便于让用户更方便的实现自定义的实现类。

# mvc:view-controller

<mvc:view-controller path="/home" />

<mvc:view-controller path="/header" view-name="common/header" />

mvc:view-controller可以在不需要Controller处理request的情况，转向到设置的View

Java代码 收藏代码

@EnableWebMvc

@Configuration

public class WebConfig extends WebMvcConfigurerAdapter {

@Override

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

}

}

或者用xml设置方式

Xml代码 收藏代码

<mvc:view-controller path="/" view-name="home"/>

# MVC的拦截器

经本人在Spring mvc中对方案1和方案2的测试表明，并没有拦截静态资源，所以可以放心使用方案1和方案2，方案3可以放弃，并且可以放心使用<mvc:annotation-driven />注解。

方案一，（近似）总拦截器，拦截所有url

<mvc:interceptors>

<bean class="com.app.mvc.MyInteceptor" />

</mvc:interceptors>

为什么叫“近似”，前面说了，Spring没有总的拦截器。

<mvc:interceptors/>会为每一个HandlerMapping，注入一个拦截器。总有一个HandlerMapping是可以找到处理器的，最多也只找到一个处理器，所以这个拦截器总会被执行的。起到了总拦截器的作用。

如果是REST风格的URL，静态资源也会被拦截。（在4.0上测试并未有此问题）

方案二， （近似） 总拦截器， 拦截匹配的URL。

<mvc:interceptors >

<mvc:interceptor>

<mvc:mapping path="/user/\*" /> <!-- /user/\* -->

<bean class="com.mvc.MyInteceptor"></bean>

</mvc:interceptor>

</mvc:interceptors>

就是比 方案一多了一个URL匹配。

如果是REST风格的URL，静态资源也会被拦截。（在4.0上测试并未有此问题）

方案三,HandlerMappint上的拦截器。

如果是REST风格的URL，静态资源就不会被拦截。因为我们精准的注入了拦截器

复制代码

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="interceptors">

<list>

<bean class="com.mvc.MyInteceptor"></bean>

</list>

</property>

</bean>

复制代码

如果使用了<mvc:annotation-driven />， 它会自动注册DefaultAnnotationHandlerMapping 与AnnotationMethodHandlerAdapter 这两个bean,所以就没有机会再给它注入interceptors属性，就无法指定拦截器。

当然我们可以通过人工配置上面的两个Bean，不使用 <mvc:annotation-driven />，就可以 给interceptors属性 注入拦截器了。

<mvc:annotation-driven />到底帮我们做了啥

一句 <mvc:annotation-driven />实际做了以下工作：（不包括添加自己定义的拦截器）

我们了解这些之后，对Spring3 MVC的控制力就更强大了，想改哪就改哪里。

spring 3.0.x是下面的配置

复制代码

<!-- 注解请求映射 -->

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="interceptors">

<list>

<ref bean="logNDCInteceptor"/> <!-- 日志拦截器，这是你自定义的拦截器 -->

</list>

</property>

</bean>

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="messageConverters">

<list>

<ref bean="byteArray\_hmc" />

<ref bean="string\_hmc" />

<ref bean="resource\_hmc" />

<ref bean="source\_hmc" />

<ref bean="xmlAwareForm\_hmc" />

<ref bean="jaxb2RootElement\_hmc" />

<ref bean="jackson\_hmc" />

</list>

</property>

</bean>

<bean id="byteArray\_hmc" class="org.springframework.http.converter.ByteArrayHttpMessageConverter" /><!-- 处理.. -->

<bean id="string\_hmc" class="org.springframework.http.converter.StringHttpMessageConverter" /><!-- 处理.. -->

<bean id="resource\_hmc" class="org.springframework.http.converter.ResourceHttpMessageConverter" /><!-- 处理.. -->

<bean id="source\_hmc" class="org.springframework.http.converter.xml.SourceHttpMessageConverter" /><!-- 处理.. -->

<bean id="xmlAwareForm\_hmc" class="org.springframework.http.converter.xml.XmlAwareFormHttpMessageConverter" /><!-- 处理.. -->

<bean id="jaxb2RootElement\_hmc" class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter" /><!-- 处理.. -->

<bean id="jackson\_hmc" class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter" /><!-- 处理json-->

复制代码

转载:http://elf8848.iteye.com/blog/875830

spring 3.1 later：

Spring 3.0.x中使用了annotation-driven后，缺省使用DefaultAnnotationHandlerMapping 来注册handler method和request的mapping关系。

AnnotationMethodHandlerAdapter来在实际调用handlermethod前对其参数进行处理。

并在dispatcherServlet中，当用户未注册自定义的ExceptionResolver时，注册AnnotationMethodHandlerExceptionResolver来对使用@ExceptionHandler标注的异常处理函数进行解析处理(这也导致当用户注册了自定义的exeptionResolver时将可能导致无法处理@ExceptionHandler)。

在spring mvc 3.1中，对应变更为

DefaultAnnotationHandlerMapping -> org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

AnnotationMethodHandlerAdapter -> org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

AnnotationMethodHandlerExceptionResolver -> ExceptionHandlerExceptionResolver

以上都在使用了annotation-driven后自动注册。

而且对应分别提供了AbstractHandlerMethodMapping , AbstractHandlerMethodAdapter和 AbstractHandlerMethodExceptionResolver以便于让用户更方便的实现自定义的实现类

<mvc:annotation-driven />的可选配置

复制代码

<mvc:annotation-driven message-codes-resolver ="bean ref" validator="" conversion-service="">

<mvc:return-value-handlers>

<bean></bean>

</mvc:return-value-handlers>

<mvc:argument-resolvers>

</mvc:argument-resolvers>

<mvc:message-converters>

</mvc:message-converters>[/color]

</mvc:annotation-driven>

复制代码

具体可以参见：http://starscream.iteye.com/blog/1098880

# Spring拦截器的定义：

Spring为我们提供了：

org.springframework.web.servlet.HandlerInterceptor接口，

org.springframework.web.servlet.handler.HandlerInterceptorAdapter适配器，

实现这个接口或继承此类，可以非常方便的实现自己的拦截器。

有以下三个方法：

复制代码

//Action之前执行:

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler);

//如果返回false则中断请求

//生成视图之前执行

public void postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView);

//最后执行，可用于释放资源

public void afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

复制代码

# mvc:resources

spring mvc 的<mvc;resources mapping="\*\*\*" location="\*\*\*">标签是在spring3.0.4出现的，主要是用来进行静态资源的访问。在spring3.0.4出来的时候spring还没有更新其schema所以在配置文件中有可能找不到<mvc:resources >标签，这个问题在spring3.0.5中已经解决，而且网上也有很多其他的解决方案，我在这里就不记录了。   
  
首先使用spring mvc需要配置其使用的servlet.在web.xml中:

**Java代码**

1. <servlet>
2. <servlet-name>springMVC</servlet-name>
3. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
4. <load-on-startup>1</load-on-startup>
5. </servlet>
7. <servlet-mapping>
8. <servlet-name>springMVC</servlet-name>
9. <url-pattern>/</url-pattern>
10. </servlet-mapping>

这里给 servlet-name定义的名称是springMVC,这样的话会在web-inf下spring会自动扫描一个XML文件名叫springMVC-servlet.xml文件，这里都是spring自动扫描的，如果你没有提供，将会报一个文件查找不到的异常。看了下org.springframework.web.servlet.DispatcherServlet加载这个文件的过程，貌似这个文件存放的地址也是可以进行设置的，具体怎么搞我还没有研究。   
  
由于spring mvc拦截了所有请求，所以当你设置

**引用**

<servlet-mapping>   
<servlet-name>springMVC</servlet-name>   
<url-pattern>/</url-pattern>   
</servlet-mapping>

的时候会影响到静态资源文件的获取，这样就需要有这个标签来帮你分类完成获取静态资源的责任。   
  
springMVC-servlet.xml文件

**Java代码  [收藏代码](javascript:void())**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:mvc="http://www.springframework.org/schema/mvc"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6. http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
8. <mvc:resources mapping="/javascript/\*\*" location="/static\_resources/javascript/"/>
9. <mvc:resources mapping="/styles/\*\*" location="/static\_resources/css/"/>
10. <mvc:resources mapping="/images/\*\*" location="/static\_resources/images/"/>
11. <mvc:**default**-servlet-handler />

14. <bean **class**="org.springframework.web.servlet.view.InternalResourceViewResolver">
15. <property name="prefix" value="/WEB-INF/views/"/>
16. <property name="suffix" value=".jsp"/>
17. </bean>
19. </beans>

这里可以看到我所有的页面引用到/styles/\*\*的资源都从/static\_resources/css里面进行查找。   
  
页面的一段静态资源访问的代码。

**Java代码  [收藏代码](javascript:void())**

1. <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"%>
2. <HTML>
3. <HEAD>
4. <TITLE> ABCDEFG </TITLE>
5. <link type="text/css" rel="stylesheet" href="<c:url value='/styles/siteboard.css'/>">
6. ...
7. ...
8. ...

可能这个标签的真谛就是为了引用资源的访问不会类似CONTROLLER一样被拦截，区分出关注的资源的访问，一般我们在springMVC里面的拦截都会配置为"/"，拦截所有的。

1、重定向  
<mvc:view-controller path="/" view-name="redirect:/admin/index"/>  
即如果当前路径是/ 则重定向到/admin/index  
  
2、view name  
<mvc:view-controller path="/" view-name=admin/index"/>  
如果当前路径是/ 则交给相应的视图解析器直接解析为视图  
如  
<bean id="defaultViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver" p:order="2">  
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>  
        <property name="contentType" value="text/html"/>  
        <property name="prefix" value="/WEB-INF/jsp/"/>  
        <property name="suffix" value=".jsp"/>  
    </bean>

则得到的视图时 /WEB-INF/jsp/admin/index.jsp

不想进controller，可以在spring-mvc.xml中配置静态访问路径

Xml代码  收藏代码

1. <!-- 访问静态资源文件  -->
2. **<mvc:resources** mapping="/images/\*\*" location="/images/" cache-period="31556926"**/>**

 像这样，jsp文件若放在静态路径/images下，可以直接访问，而不经过controller。