## [17. Web MVC framework](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/" \l "mvc)

### [17.1. Introduction to Spring Web MVC framework](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-introduction)（关于SpringMVC的介绍）

#### 17.1.1 Features of Spring Web MVC

Spring MVC框架主要围绕着DispatcherServlet设计而来，默认的请求句柄是通过@Controller和@RequestMapping注解，提供多元化的处理方法。通过Spring3.0介绍，@Controller原理允许你RESTful的WEB站点和应用，通过@PathVariable注解和其它的另外属性。

### 17.2 The DispatcherServlet

SpringMVC框架跟其它框架类似，是以请求为驱动而设计，分发请求至不同Controller。

接下来简单描述下SpringMVC DispatcherServlet请求工作流描述。DispatcherServlet的设计模式是以“Front Controller”为设计理念。



上述的请求工作流是站在高角度的。

DispatcherServlet是真正的Servlet，是继承于HttpServlet，被申明在WEB.XML文件中

<web-app>

<servlet>

<servlet-name>example</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

<url-pattern>/example/\*</url-pattern>

</servlet-mapping>

</web-app>

上述的web.xml例子的配置，实际上DispatcherServlet会处理/example开始的请求访问，你也可以不同xml的配置，而是采纳动态方法。

**public** **class** MyWebApplicationInitializer **implements** WebApplicationInitializer {

*@Override*

**public** **void** onStartup(ServletContext container) {

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", **new** DispatcherServlet());

registration.setLoadOnStartup(1);

registration.addMapping("/example/\*");

}

}



在上述的DispatcherServlet初始化中，Spring MVC为寻找一个【servlet-name】-servlet.xml在WEB-INF目录下

<web-app>

<servlet>

<servlet-name>**golfing**</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>**golfing**</servlet-name>

<url-pattern>/golfing/\*</url-pattern>

</servlet-mapping>

</web-app>

这个时候你就需要配置这么一个文件/WEB-INF/golfing-servlet.xml，能够包含所有的Spring WEB MVC的Bean，你也能在外面配置，以参数的形式加载进来。

<web-app>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

</web-app>

#### 17.2.1 Special Bean Types In the WebApplicationContext

介绍下特殊的在WebApplicationContext的Bean

|  |  |
| --- | --- |
| Bean类型 | 说明 |
| [HandlerMapping](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-handlermapping) |  |
| HandlerAdapter |  |
| [HandlerExceptionResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-exceptionhandlers) |  |
| [ViewResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-viewresolver) |  |
| [LocaleResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-localeresolver) &[LocaleContextResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-timezone) |  |
| [ThemeResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-themeresolver) |  |
| [MultipartResolver](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-multipart) |  |
| [FlashMapManager](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#mvc-flash-attributes) |  |
|  |  |

### 17.3 Implementing Controllers

Controllers提供了访问应用的接口，Controllers拦截用户的输入，然后转换为model，通过view展示。Spring2.5介绍了MVC Controller的注解方式， @RequestMapping,@RequestParam, @ModelAttribute。

*@Controller*

**public** **class** HelloWorldController {

*@RequestMapping("/helloWorld")*

**public** String helloWorld(Model model) {

model.addAttribute("message", "Hello World!");

**return** "helloWorld";

}

}

就像你看到的样子，通过@Controller and @RequestMapping能够很灵活地进行方法名的命令。

#### 17.3.1 Defining a controller with @Controller

为了能够自动检测到注解的controller，你需要配置如下信息

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

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">

<context:component-scan base-package="org.springframework.samples.petclinic.web"/>

*<!-- ... -->*

</beans>

#### 17.3.2 Mapping Requests With @RequestMapping

你可以通过RequestMapping 注解，映射到希望的句柄的方法上，通过如下信息配置

*@Controller*

**@RequestMapping("/appointments")**

**public** **class** AppointmentsController {

**private** **final** AppointmentBook appointmentBook;

*@Autowired*

**public** AppointmentsController(AppointmentBook appointmentBook) {

**this**.appointmentBook = appointmentBook;

}

**@RequestMapping(method = RequestMethod.GET)**

**public** Map<String, Appointment> get() {

**return** appointmentBook.getAppointmentsForToday();

}

**@RequestMapping(value="/{day}", method = RequestMethod.GET)**

**public** Map<String, Appointment> getForDay(*@PathVariable* *@DateTimeFormat(iso=ISO.DATE)* Date day, Model model) {

**return** appointmentBook.getAppointmentsForDay(day);

}

**@RequestMapping(value="/new", method = RequestMethod.GET)**

**public** AppointmentForm getNewForm() {

**return** **new** AppointmentForm();

}

**@RequestMapping(method = RequestMethod.POST)**

**public** String add(*@Valid* AppointmentForm appointment, BindingResult result) {

**if** (result.hasErrors()) {

**return** "appointments/new";

}

appointmentBook.addAppointment(appointment);

**return** "redirect:/appointments";

}

}

##### Matrix Variables

矩阵参数出现在path的变量上，每一个矩阵参数都是通过“;”分割开来，比如"/cars;color=red;year=2012"，多个参数可能被分割成"color=red,green,blue"或者重复"color=red;color=green;color=blue"

比如

*// GET /pets/42;q=11;r=22*

*@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)*

**public** **void** findPet(*@PathVariable* String petId, *@MatrixVariable* **int** q) {

*// petId == 42*

*// q == 11*

}

*// GET /owners/42;q=11/pets/21;q=22*

*@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)*

**public** **void** findPet(

*@MatrixVariable(value="q", pathVar="ownerId")* **int** q1,

*@MatrixVariable(value="q", pathVar="petId")* **int** q2) {

*// q1 == 11*

*// q2 == 22*

}

默认值

*// GET /pets/42*

*@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)*

**public** **void** findPet(*@MatrixVariable(required=false, defaultValue="1")* **int** q) {

*// q == 1*

}

所有矩阵参数可能存放在map中

*// GET /owners/42;q=11;r=12/pets/21;q=22;s=23*

*@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)*

**public** **void** findPet(

*@MatrixVariable* Map<String, String> matrixVars,

*@MatrixVariable(pathVar="petId"")* Map<String, String> petMatrixVars) {

*// matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]*

*// petMatrixVars: ["q" : 11, "s" : 23]*

}

##### Consumable Media Types

请求中的Content-Type对应的media type

*@Controller*

@RequestMapping(value = "/pets", method = RequestMethod.POST, **consumes="application/json"**)

**public** **void** addPet(*@RequestBody* Pet pet, Model model) {

*// implementation omitted*

}

##### Producible Media Types

*@Controller*

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, **produces="application/json"**)

*@ResponseBody*

**public** Pet getPet(*@PathVariable* String petId, Model model) {

*// implementation omitted*

}

##### Request Parameters and Header Values

请求时候的参数获取

*@Controller*

*@RequestMapping("/owners/{ownerId}")*

**public** **class** RelativePathUriTemplateController {

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, **params="myParam=myValue"**)

**public** **void** findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model) {

*// implementation omitted*

}

}

获取head的参数形式

*@Controller*

*@RequestMapping("/owners/{ownerId}")*

**public** **class** RelativePathUriTemplateController {

@RequestMapping(value = "/pets", method = RequestMethod.GET, **headers="myHeader=myValue"**)

**public** **void** findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model) {

*// implementation omitted*

}

}

#### 17.3.3 Defining @RequestMapping handler methods

##### Supported method return types

下面介绍下返回的对象类型

* ModelAndView，通过@ModelAttribute注解关联到数据进入类型的方法
* Model，通过@ModelAttribute注解
* View

##### Binding request parameters to method parameters with @RequestParam

通过使用@RequestParam注解，把请求参数绑定到方法参数上

*@Controller*

*@RequestMapping("/pets")*

*@SessionAttributes("pet")*

**public** **class** EditPetForm {

*// ...*

*@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";

}

*// ...*

}

@RequestParam(value="id", required=false)当请求的时候为required=false时候，表示请求时候不必须携带，相反的话，就必须携带。

##### Mapping the request body with the @RequestBody annotation

###### @RequestBody

作用：

      i) 该注解用于读取Request请求的body部分数据，使用系统默认配置的HttpMessageConverter进行解析，然后把相应的数据绑定到要返回的对象上；

      ii) 再把HttpMessageConverter返回的对象数据绑定到 controller中方法的参数上。

使用时机：

A) GET、POST方式提时， 根据request header Content-Type的值来判断:

* application/x-www-form-urlencoded， 可选（即非必须，因为这种情况的数据@RequestParam, @ModelAttribute也可以处理，当然@RequestBody也能处理）；
* multipart/form-data, 不能处理（即使用@RequestBody不能处理这种格式的数据）；
* 其他格式， 必须（其他格式包括application/json, application/xml等。这些格式的数据，必须使用@RequestBody来处理）；

B) PUT方式提交时， 根据request header Content-Type的值来判断:

* application/x-www-form-urlencoded， 必须；
* multipart/form-data, 不能处理；
* 其他格式， 必须；

说明：request的body部分的数据编码格式由header部分的Content-Type指定；

###### @ResponseBody

作用：

      该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

使用时机：

      返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

###### HttpMessageConverter

**[java]** [view plaincopy](http://blog.csdn.net/walkerjong/article/details/7520896)

1. <span style="font-family:Microsoft YaHei;">/\*\*
2. \* Strategy interface that specifies a converter that can convert from and to HTTP requests and responses.
3. \*
4. \* @author Arjen Poutsma
5. \* @author Juergen Hoeller
6. \* @since 3.0
7. \*/
8. **public** **interface** HttpMessageConverter<T> {
10. /\*\*
11. \* Indicates whether the given class can be read by this converter.
12. \* @param clazz the class to test for readability
13. \* @param mediaType the media type to read, can be {@code null} if not specified.
14. \* Typically the value of a {@code Content-Type} header.
15. \* @return {@code true} if readable; {@code false} otherwise
16. \*/
17. **boolean** canRead(Class<?> clazz, MediaType mediaType);
19. /\*\*
20. \* Indicates whether the given class can be written by this converter.
21. \* @param clazz the class to test for writability
22. \* @param mediaType the media type to write, can be {@code null} if not specified.
23. \* Typically the value of an {@code Accept} header.
24. \* @return {@code true} if writable; {@code false} otherwise
25. \*/
26. **boolean** canWrite(Class<?> clazz, MediaType mediaType);
28. /\*\*
29. \* Return the list of {@link MediaType} objects supported by this converter.
30. \* @return the list of supported media types
31. \*/
32. List<MediaType> getSupportedMediaTypes();
34. /\*\*
35. \* Read an object of the given type form the given input message, and returns it.
36. \* @param clazz the type of object to return. This type must have previously been passed to the
37. \* {@link #canRead canRead} method of this interface, which must have returned {@code true}.
38. \* @param inputMessage the HTTP input message to read from
39. \* @return the converted object
40. \* @throws IOException in case of I/O errors
41. \* @throws HttpMessageNotReadableException in case of conversion errors
42. \*/
43. T read(Class<? **extends** T> clazz, HttpInputMessage inputMessage)
44. **throws** IOException, HttpMessageNotReadableException;
46. /\*\*
47. \* Write an given object to the given output message.
48. \* @param t the object to write to the output message. The type of this object must have previously been
49. \* passed to the {@link #canWrite canWrite} method of this interface, which must have returned {@code true}.
50. \* @param contentType the content type to use when writing. May be {@code null} to indicate that the
51. \* default content type of the converter must be used. If not {@code null}, this media type must have
52. \* previously been passed to the {@link #canWrite canWrite} method of this interface, which must have
53. \* returned {@code true}.
54. \* @param outputMessage the message to write to
55. \* @throws IOException in case of I/O errors
56. \* @throws HttpMessageNotWritableException in case of conversion errors
57. \*/
58. **void** write(T t, MediaType contentType, HttpOutputMessage outputMessage)
59. **throws** IOException, HttpMessageNotWritableException;
61. }
62. </span>

该接口定义了四个方法，分别是读取数据时的 canRead(), read() 和 写入数据时的canWrite(), write()方法。

在使用 <mvc:annotation-driven />标签配置时，默认配置了RequestMappingHandlerAdapter（注意是RequestMappingHandlerAdapter不是AnnotationMethodHandlerAdapter,详情查看Spring 3.1 document “16.14 Configuring Spring MVC”章节），并为他配置了一下默认的HttpMessageConverter：

**[java]** [view plaincopy](http://blog.csdn.net/walkerjong/article/details/7520896)

1. ByteArrayHttpMessageConverter converts **byte** arrays.
3. StringHttpMessageConverter converts strings.
5. ResourceHttpMessageConverter converts to/from org.springframework.core.io.Resource **for** all media types.
7. SourceHttpMessageConverter converts to/from a javax.xml.transform.Source.
9. FormHttpMessageConverter converts form data to/from a MultiValueMap<String, String>.
11. Jaxb2RootElementHttpMessageConverter converts Java objects to/from XML — added **if** JAXB2 is present on the classpath.
13. MappingJacksonHttpMessageConverter converts to/from JSON — added **if** Jackson is present on the classpath.
15. AtomFeedHttpMessageConverter converts Atom feeds — added **if** Rome is present on the classpath.
17. RssChannelHttpMessageConverter converts RSS feeds — added **if** Rome is present on the classpath.

ByteArrayHttpMessageConverter: 负责读取二进制格式的数据和写出二进制格式的数据；

StringHttpMessageConverter：   负责读取字符串格式的数据和写出二进制格式的数据；

ResourceHttpMessageConverter：负责读取资源文件和写出资源文件数据；

FormHttpMessageConverter：       负责读取form提交的数据（能读取的数据格式为 application/x-www-form-urlencoded，不能读取multipart/form-data格式数据）；负责写入application/x-www-from-urlencoded和multipart/form-data格式的数据；

MappingJacksonHttpMessageConverter:  负责读取和写入json格式的数据；

SouceHttpMessageConverter：                   负责读取和写入 xml 中javax.xml.transform.Source定义的数据；

Jaxb2RootElementHttpMessageConverter:  负责读取和写入xml 标签格式的数据；

AtomFeedHttpMessageConverter:              负责读取和写入Atom格式的数据；

RssChannelHttpMessageConverter:           负责读取和写入RSS格式的数据；

当使用@RequestBody和@ResponseBody注解时，RequestMappingHandlerAdapter就使用它们来进行读取或者写入相应格式的数据。

###### HttpMessageConverter匹配过程：

@RequestBody注解时： 根据Request对象header部分的Content-Type类型，逐一匹配合适的HttpMessageConverter来读取数据；

spring 3.1源代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/walkerjong/article/details/7520896)

1. <span style="font-family:Microsoft YaHei;">**private** Object readWithMessageConverters(MethodParameter methodParam, HttpInputMessage inputMessage, Class paramType)
2. **throws** Exception {
4. MediaType contentType = inputMessage.getHeaders().getContentType();
5. **if** (contentType == **null**) {
6. StringBuilder builder = **new** StringBuilder(ClassUtils.getShortName(methodParam.getParameterType()));
7. String paramName = methodParam.getParameterName();
8. **if** (paramName != **null**) {
9. builder.append(' ');
10. builder.append(paramName);
11. }
12. **throw** **new** HttpMediaTypeNotSupportedException(
13. "Cannot extract parameter (" + builder.toString() + "): no Content-Type found");
14. }
16. List<MediaType> allSupportedMediaTypes = **new** ArrayList<MediaType>();
17. **if** (**this**.messageConverters != **null**) {
18. **for** (HttpMessageConverter<?> messageConverter : **this**.messageConverters) {
19. allSupportedMediaTypes.addAll(messageConverter.getSupportedMediaTypes());
20. **if** (messageConverter.canRead(paramType, contentType)) {
21. **if** (logger.isDebugEnabled()) {
22. logger.debug("Reading [" + paramType.getName() + "] as \"" + contentType
23. +"\" using [" + messageConverter + "]");
24. }
25. **return** messageConverter.read(paramType, inputMessage);
26. }
27. }
28. }
29. **throw** **new** HttpMediaTypeNotSupportedException(contentType, allSupportedMediaTypes);
30. }</span>

@ResponseBody注解时： 根据Request对象header部分的Accept属性（逗号分隔），逐一按accept中的类型，去遍历找到能处理的HttpMessageConverter；

源代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/walkerjong/article/details/7520896)

1. <span style="font-family:Microsoft YaHei;">**private** **void** writeWithMessageConverters(Object returnValue,
2. HttpInputMessage inputMessage, HttpOutputMessage outputMessage)
3. **throws** IOException, HttpMediaTypeNotAcceptableException {
4. List<MediaType> acceptedMediaTypes = inputMessage.getHeaders().getAccept();
5. **if** (acceptedMediaTypes.isEmpty()) {
6. acceptedMediaTypes = Collections.singletonList(MediaType.ALL);
7. }
8. MediaType.sortByQualityValue(acceptedMediaTypes);
9. Class<?> returnValueType = returnValue.getClass();
10. List<MediaType> allSupportedMediaTypes = **new** ArrayList<MediaType>();
11. **if** (getMessageConverters() != **null**) {
12. **for** (MediaType acceptedMediaType : acceptedMediaTypes) {
13. **for** (HttpMessageConverter messageConverter : getMessageConverters()) {
14. **if** (messageConverter.canWrite(returnValueType, acceptedMediaType)) {
15. messageConverter.write(returnValue, acceptedMediaType, outputMessage);
16. **if** (logger.isDebugEnabled()) {
17. MediaType contentType = outputMessage.getHeaders().getContentType();
18. **if** (contentType == **null**) {
19. contentType = acceptedMediaType;
20. }
21. logger.debug("Written [" + returnValue + "] as \"" + contentType +
22. "\" using [" + messageConverter + "]");
23. }
24. **this**.responseArgumentUsed = **true**;
25. **return**;
26. }
27. }
28. }
29. **for** (HttpMessageConverter messageConverter : messageConverters) {
30. allSupportedMediaTypes.addAll(messageConverter.getSupportedMediaTypes());
31. }
32. }
33. **throw** **new** HttpMediaTypeNotAcceptableException(allSupportedMediaTypes);
34. }</span>

MappingJacksonHttpMessageConverter 调用了 objectMapper.writeValue(OutputStream stream, Object)方法，使用@ResponseBody注解返回的对象就传入Object参数内。若返回的对象为已经格式化好的json串时，不使用@RequestBody注解，而应该这样处理：

1、response.setContentType("application/json; charset=UTF-8");

2、response.getWriter().print(jsonStr);

直接输出到body区，然后的视图为void。

##### Creating REST Controllers with the @RestController annotation

##### Using HttpEntity

HttpEntity类似于@RequestBody 和 @ResponseBody，通过HttpEntity能够访问到body。

*@RequestMapping("/something")*

**public** ResponseEntity<String> handle(HttpEntity<**byte**[]> requestEntity) **throws** UnsupportedEncodingException {

String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader"));

**byte**[] requestBody = requestEntity.getBody();

*// do something with request header and body*

HttpHeaders responseHeaders = **new** HttpHeaders();

responseHeaders.set("MyResponseHeader", "MyValue");

**return** **new** ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);

}

##### Mapping cookie values with the @CookieValue annotation

*@RequestMapping("/displayHeaderInfo.do")*

**public** **void** displayHeaderInfo(**@CookieValue("JSESSIONID")** String cookie) {

*//...*

}

##### Mapping request header attributes with the @RequestHeader annotation

这是一个请求头

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

*@RequestMapping("/displayHeaderInfo.do")*

**public** **void** displayHeaderInfo(**@RequestHeader("Accept-Encoding")** String encoding,

**@RequestHeader("Keep-Alive")** **long** keepAlive) {

*//...*

}

@ControllerAdvice，是spring3.2提供的新注解，从名字上可以看出大体意思是控制器增强。让我们先看看@ControllerAdvice的实现：

**Java代码  [收藏代码](javascript:void())**

1. @Target(ElementType.TYPE)
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. @Component
5. **public** **@interface** ControllerAdvice {
7. }

 没什么特别之处，该注解使用@Component注解，这样的话当我们使用<context:component-scan>扫描时也能扫描到，具体可参考[【第十二章】零配置 之 12.3 注解实现Bean定义 ——跟我学spring3](http://jinnianshilongnian.iteye.com/blog/1461055" \t "_blank)。

其javadoc定义是：

**写道**

/\*\*  
\* Indicates the annotated class assists a "Controller".  
\*  
\* <p>Serves as a specialization of {@link Component @Component}, allowing for  
\* implementation classes to be autodetected through classpath scanning.  
\*  
\* <p>It is typically used to define {@link ExceptionHandler @ExceptionHandler},  
\* {@link InitBinder @InitBinder}, and {@link ModelAttribute @ModelAttribute}  
\* methods that apply to all {@link RequestMapping @RequestMapping} methods.  
\*  
\* @author Rossen Stoyanchev  
\* @since 3.2  
\*/

即把@ControllerAdvice注解内部使用@ExceptionHandler、@InitBinder、@ModelAttribute注解的方法应用到所有的 @RequestMapping注解的方法。非常简单，不过只有当使用@ExceptionHandler最有用，另外两个用处不大。

接下来看段代码：

**Java代码  [收藏代码](javascript:void())**

1. @ControllerAdvice
2. **public** **class** ControllerAdviceTest {
4. @ModelAttribute
5. **public** User newUser() {
6. System.out.println("============应用到所有@RequestMapping注解方法，在其执行之前把返回值放入Model");
7. **return** **new** User();
8. }
10. @InitBinder
11. **public** **void** initBinder(WebDataBinder binder) {
12. System.out.println("============应用到所有@RequestMapping注解方法，在其执行之前初始化数据绑定器");
13. }
15. @ExceptionHandler(UnauthenticatedException.**class**)
16. @ResponseStatus(HttpStatus.UNAUTHORIZED)
17. **public** String processUnauthenticatedException(NativeWebRequest request, UnauthenticatedException e) {
18. System.out.println("===========应用到所有@RequestMapping注解的方法，在其抛出UnauthenticatedException异常时执行");
19. **return** "viewName"; //返回一个逻辑视图名
20. }
21. }

如果你的spring-mvc配置文件使用如下方式扫描bean

**Java代码  [收藏代码](javascript:void())**

1. <context:component-scan base-**package**="com.sishuok.es" use-**default**-filters="false">
2. <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
3. </context:component-scan>

 需要把@ControllerAdvice包含进来，否则不起作用：

**Java代码  [收藏代码](javascript:void())**

1. <context:component-scan base-**package**="com.sishuok.es" use-**default**-filters="false">
2. <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
3. <context:include-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice"/>
4. </context:component-scan>

1、@ModelAttribute注解的方法作用请参考[SpringMVC强大的数据绑定（2）——第六章 注解式控制器详解——跟着开涛学SpringMVC](http://jinnianshilongnian.iteye.com/blog/1705701" \t "_blank)中的【二、暴露表单引用对象为模型数据】，作用是一样的，只不过此处是对所有的@RequestMapping注解的方法都起作用。当需要设置全局数据时比较有用。

2、@InitBinder注解的方法作用请参考[SpringMVC数据类型转换——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC](http://jinnianshilongnian.iteye.com/blog/1723270" \t "_blank)，同1类似。当需要全局注册时比较有用。

3、@ExceptionHandler，异常处理器，此注解的作用是当出现其定义的异常时进行处理的方法，其可以使用springmvc提供的数据绑定，比如注入HttpServletRequest等，还可以接受一个当前抛出的Throwable对象。可以参考javadoc或snowolf的[Spring 注解学习手札（八）补遗——@ExceptionHandler](http://snowolf.iteye.com/blog/1636050" \t "_blank)。

该注解非常简单，大多数时候其实只@ExceptionHandler比较有用，其他两个用到的场景非常少，这样可以把异常处理器应用到所有控制器，而不是@Controller注解的单个控制器。

##### Jackson Serialization View Support

##### Jackson JSONP Support

为了支持JSONP支持@ResponseBody 和 ResponseEntity的方法

*@ControllerAdvice*

**public** **class** JsonpAdvice **extends** AbstractJsonpResponseBodyAdvice {

**public** JsonpAdvice() {

**super**("callback");

}

}

### 17.3.4 Asynchronous Request Processing

*@RequestMapping(method=RequestMethod.POST)*

**public** Callable<String> processUpload(**final** MultipartFile file) {

**return** **new** Callable<String>() {

**public** String call() **throws** Exception {

*// ...*

**return** "someView";

}

};

}

## 17.4 Handler mappings

Spring的拦截器，在我们申请指定的功能点时候，产生非常大的效果。

需要实现拦截功能必选实现来自于org.springframework.web.servlet包的HandlerInterceptor 接口，该接口提供了三种方法，preHandle()：在实际句柄类被处理之前；postHandler()：在实际句柄类被处理之后；afterCompletion()：在整个请求处理被结束后。该三个方法已经足够灵活地处理各种复杂情况。

Interceptors 实际上是责任链模式，Interceptors 属性将在AbstractHandlerMapping配置。

<beans>

<bean id="handlerMapping"

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

<property name="interceptors">

<list>

<ref bean="officeHoursInterceptor"/>

</list>

</property>

</bean>

<bean id="officeHoursInterceptor"

class="samples.TimeBasedAccessInterceptor">

<property name="openingTime" value="9"/>

<property name="closingTime" value="18"/>

</bean>

<beans>

**public** **class** TimeBasedAccessInterceptor **extends** HandlerInterceptorAdapter {

**private** **int** openingTime;

**private** **int** closingTime;

**public** **void** setOpeningTime(**int** openingTime) {

**this**.openingTime = openingTime;

}

**public** **void** setClosingTime(**int** closingTime) {

**this**.closingTime = closingTime;

}

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) **throws** Exception {

Calendar cal = Calendar.getInstance();

**int** hour = cal.get(HOUR\_OF\_DAY);

**if** (openingTime <= hour && hour < closingTime) {

**return** true;

}

response.sendRedirect("http://host.com/outsideOfficeHours.html");

**return** false;

}

}

上述的列子举例说明了下，任何请求处理都会被TimeBasedAccessInterceptor拦截类，如果一旦当前时间在工作之外时间，用户的请求将被重定向到一个静态页面，实际上，有点相当于公务员，工作时间才能回复，不是吗？

## 17.5 Resolving views（解决视图）

Spring提供页面渲染方式，能够满足我们把model渲染至浏览器上而无需特定页面技术。Spring能够整合JSP、velocity、XSTL等多种引擎。

Spring主要依托于两个视图句柄类，ViewResolver和View，ViewResolver提供一种映射把view和实际的view结合起来。View接口主要是用来寻址把请求和请求的页面结合起来。

### 17.5.1 Resolving views with the ViewResolver interface（通过ViewResolver解决视图）

Spring提供了一系列的ViewResolver，下面的表格将罗列些常用的ViewResolver。

|  |  |
| --- | --- |
| ViewResolver | Description |
| AbstractCachingViewResolver | 抽象ViewResolver能够缓存View，需要扩展该ViewResolver |
| XmlViewResolver | 实现ViewResolver接口，接受标准的XML，默认配置/WEB-INFO/views.xml |
| ResourceBundleViewResolver | 实现ViewResolver接口被定义在ResourceBundle，标准的方式是与properties文件捆绑在一起，默认名称是view.properties |
| UrlBasedViewResolver | 简单实现ViewResolver接口，通过view逻辑到URLS上 |
| InternalResourceViewResolver | UrlBaseViewResolver的简单子集，你能够指定靠SetViewClass方法，能够支持JstlView和TileView |
| VelocityViewResolver | UrlBaseViewResolver的简单子集，能够支持Velocity |
| FreeMarkerViewResolver | UrlBaseViewResolver的简单子集，能够支持FreeMarkerView |
| ContentNegotiatingViewResolver | 接口 ViewResolver 的实现，用于根据请求文件的后缀名或请求的 header 中的 accept 字段查找视图 |

举例，JSP作为视图技术，你能够使用UrlBaseViewResolver，ViewResolver把view到指定URL上，移交请求到RequestDispatcher ，然后渲染view。

<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBasedViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

当return test的view的名称，ViewResolver通过RequestDispatcher 请求处理发送请求到/WEB-INF/jsp/test.jsp

当你想要捆绑不同view技术在web应用中时，可以通过使用ResourceBundleViewResolver

<bean id="viewResolver"

class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

<property name="defaultParentView" value="parentView"/>

</bean>

它和 XmlViewResolver 一样，也是继承自 AbstractCachingViewResolver ，但是它缓存的不是视图，这个会在后面有说到。和 XmlViewResolver 一样它也需要有一个配置文件来定义逻辑视图名称和真正的 View 对象的对应关系，不同的是 ResourceBundleViewResolver 的配置文件是一个属性文件，而且必须是放在 classpath 路径下面的，默认情况下这个配置文件是在 classpath 根目录下的 views.properties 文件，如果不使用默认值的话，则可以通过属性 baseName 或baseNames 来指定。 baseName 只是指定一个基名称， Spring 会在指定的 classpath 根目录下寻找以指定的 baseName 开始的属性文件进行 View 解析，如指定的 baseName 是 base ，那么 base.properties 、 baseabc.properties 等等以 base 开始的属性文件都会被 Spring 当做 ResourceBundleViewResolver 解析视图的资源文件。 ResourceBundleViewResolver 使用的属性配置文件的内容类似于这样：

resourceBundle.(class)=org.springframework.web.servlet.view.InternalResourceView

resourceBundle.url=/index.jsp

test.(class)=org.springframework.web.servlet.view.InternalResourceView

test.url=/test.jsp

在这个配置文件中我们定义了两个 InternalResourceView 对象，一个的名称是 resourceBundle ，对应 URL 是 /index.jsp ，另一个名称是 test ，对应的 URL 是 /test.jsp。从这个定义来看我们可以知道 resourceBundle 是对应的视图名称，使用 resourceBundle.(class) 来指定它对应的视图类型， resourceBundle.url 指定这个视图的 url 属性。会思考的读者看到这里可能会有这样一个问题：为什么 resourceBundle 的 class 属性要用小括号包起来，而它的 url 属性就不需要呢？这就需要从 ResourceBundleViewResolver 进行视图解析的方法来说了。 ResourceBundleViewResolver 还是通过 bean工厂来获得对应视图名称的视图 bean 对象来解析视图的。那么这些 bean 从哪里来呢？就是从我们定义的 properties 属性文件中来。在 ResourceBundleViewResolver 第一次进行视图解析的时候会先 new 一个 BeanFactory 对象，然后把 properties 文件中定义好的属性按照它自身的规则生成一个个的 bean 对象注册到该 BeanFactory 中，之后会把该 BeanFactory 对象保存起来，所以 ResourceBundleViewResolver 缓存的是 BeanFactory ，而不是直接的缓存从 BeanFactory 中取出的视图 bean 。然后会从bean 工厂中取出名称为逻辑视图名称的视图 bean 进行返回。接下来就讲讲 Spring 通过 properties 文件生成 bean 的规则。它会把 properties 文件中定义的属性名称按最后一个点“ . ”进行分割，把点前面的内容当做是 bean 名称，点后面的内容当做是 bean 的属性。这其中有几个特别的属性， Spring 把它们用小括号包起来了，这些特殊的属性一般是对应的 attribute ，但不是 bean 对象所有的 attribute 都可以这样用。其中 (class) 是一个，除了 (class) 之外，还有 (scope) 、 (parent) 、 (abstract) 、 (lazy-init)。而除了这些特殊的属性之外的其他属性， Spring 会把它们当做 bean 对象的一般属性进行处理，就是 bean 对象对应的 property 。所以根据上面的属性配置文件将生成如下两个 bean 对象：

<bean id="resourceBundle" class="org.springframework.web.servlet.view.InternalResourceView">

<property name="url" value="/index.jsp"/>

</bean>

<bean id="test" class="org.springframework.web.servlet.view.InternalResourceView">

<property name="url" value="/test.jsp"/>

</bean>

从 ResourceBundleViewResolver 使用的配置文件我们可以看出，它和 XmlViewResolver 一样可以解析多种不同类型的 View ，因为它们的 View 是通过配置的方式指定的，这也就意味着我们可以指定 A 视图是 InternalResourceView ， B 视图是 JstlView。

来看下面这个一个例子，我在 SpringMVC 的配置文件中定义了一个 ResourceBundleViewResolver 对象，指定其 baseName 为 views ，然后 order 为 1 。

<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

<property name="order" value="1"/>

</bean>

我在 classpath 的根目录下有两个属性文件，一个是 views.properties ，一个是 views\_abc.properties ，它们的内容分别如下：

views.properties ：

resourceBundle.(class)=org.springframework.web.servlet.view.InternalResourceView

resourceBundle.url=/index.jsp

test.(class)=org.springframework.web.servlet.view.InternalResourceView

test.url=/test.jsp

views\_abc.properties ：

abc.(class)=org.springframework.web.servlet.view.InternalResourceView

abc.url=/abc.jsp

定义了如下这样一个 Controller ，它有三个处理器方法。

@Controller

@RequestMapping("/mytest")

public class MyController {

@RequestMapping("resourceBundle")

public String resourceBundle() {

return "resourceBundle";

}

@RequestMapping("testResourceBundle")

public String testResourceBundle() {

return "test";

}

@RequestMapping("abc")

public String abc() {

return "abc";

}

}

那么当我们请求 /mytest/resourceBundle.do 的时候， ResourceBundleViewResolver 会首先尝试着来解析该视图，这里 Controller 处理器方法返回的逻辑视图名称是 resourceBundle ， ResourceBundleViewResolver 按照上面提到的解析方法进行解析，这个时候它发现它是可以解析的，然后就返回了一个 url 为 /index.jsp 的 InternalResourceView 对象。同样，请求 /mytest/testResourceBundle.do 返回的逻辑视图 test 和 /mytest/abc.do 返回的逻辑视图 abc 它都可以解析。当我们把 basename 指定为包的形式，如“ com.tiantian.views ”，的时候 Spring 会按照点“ . ”划分为目录的形式，到 classpath 相应目录下去寻找 basename 开始的配置文件，如上面我们指定 basename 为“ com.tiantian.views ”，那么 spring 就会到 classpath 下的 com/tiantian 目录下寻找文件名以 views 开始的 properties 文件作为解析视图的配置文件。

### 17.5.2 Chaining ViewResolvers

Spring提供多种ViewResolver，你能够链条式加入ViewResolver，如果有必要性，你可以通过设置order属性指定顺序，越高的order属性值，ViewResolver越迟被加载到链条中。

下面的例子将介绍下两种ViewResolver。

<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="order" value="1"/>

<property name="location" value="/WEB-INF/views.xml"/>

</bean>

*<!-- in views.xml -->*

<beans>

<bean name="report" class="org.springframework.example.ReportExcelView"/>

</beans>

如果指定的ViewResolver没有渲染到结果集上，Spring会检查上下文。如果额外的ViewResolver存在，Spring会持续检查他们知道view被渲染，如果没有ViewResolver，Spring会丢出一场ServletException。

### 17.5.3 Redirecting to views

主要通过redirect和forward两个关键词的前缀，来代替使用requestDispatcher.redirect()和forward()方法。

### 17.5.4 ContentNegotiatingViewResolver

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<property name="mediaTypes">

<map>

<entry key="atom" value="application/atom+xml"/>

<entry key="html" value="text/html"/>

<entry key="json" value="application/json"/>

</map>

</property>

<property name="viewResolvers">

<list>

<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

</list>

</property>

<property name="defaultViews">

<list>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" />

</list>

</property>

</bean>

<bean id="content" class="com.foo.samples.rest.SampleContentAtomView"/>

根据后缀的扩展性名字，来分流。

## 17.6 Using flash attributes

## Flash 属性 和 RedirectAttribute：通过FlashMap存储一个请求的输出，当进入另一个请求时作为该请求的输入，典型场景如重定向（POST-REDIRECT-GET模式，1、POST时将下一次需要的数据放在FlashMap；2、重定向；3、通过GET访问重定向的地址，此时FlashMap会把1放到FlashMap的数据取出放到请求中，并从FlashMap中删除；从而支持在两次请求之间保存数据并防止了重复表单提交）。 Spring Web MVC提供FlashMapManager用于管理FlashMap，默认使用SessionFlashMapManager，即数据默认存储在session中。

## 17.7 Building URIs

SpringMVC提供通过UriComponentsBuilder& UriComponents构建URI

例如下面的URI模板

UriComponents uriComponents = UriComponentsBuilder.fromUriString(

"http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();

UriComponents是不可变的，通过expand()和encode()方法

UriComponents uriComponents = UriComponentsBuilder.newInstance()

.scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()

.expand("42", "21")

.encode();

在Servlet环境下，通过ServletUriComponentsBuilder 构建静态工厂方法提供有效的URL信息来自于Servlet请求。

HttpServletRequest request = ...

*// Re-use host, scheme, port, path and query string*

*// Replace the "accountId" query param*

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)

.replaceQueryParam("accountId", "{id}").build()

.expand("123")

.encode();

或者

*// Re-use host, port and context path*

*// Append "/accounts" to the path*

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)

.path("/accounts").build()

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)

.path("/accounts").build()

### 17.7.1 Building URIs to Controllers and methods