<http://blog.csdn.net/lchpersonal521/article/details/53112728>

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/lchpersonal521/article/details/53112728)

## 前言

　　Spring mvc框架相信很多人都很熟悉了，关于这方面的资料也是一搜一大把。但是感觉讲的都不是很细致，让很多初学者都云里雾里的。本人也是这样，之前研究过，但是后面一段时间不用发现又忘记了。所以决定写下来，以备后用。   
　　本系列文基于spring-4.3.1，配置方式全部基于java-based方式

## 从配置讲起

先上一段配置的代码：

@EnableWebMvc

@Configuration

public class MvcConfig extends WebMvcConfigurerAdapter {

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.jsp("/WEB-INF/jsp/", ".jsp");

registry.enableContentNegotiation(new MappingJackson2JsonView());

}

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.favorPathExtension(true)

.ignoreAcceptHeader(true)

.parameterName("mediaType")

.defaultContentType(MediaType.TEXT\_HTML)

.mediaType("html", MediaType.TEXT\_HTML)

.mediaType("json", MediaType.APPLICATION\_JSON);

}

@Bean(name = "multipartResolver")

// 文件上传bean

public CommonsMultipartResolver commonsMultipartResolver() {

return new CommonsMultipartResolver();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

　　基于java-based方式的spring mvc配置，需要创建一个**配置**类并实现WebMvcConfigurer 接口，WebMvcConfigurerAdapter 抽象类是对WebMvcConfigurer 接口的简单抽象（增加了一些默认实现），所以上面配置代码选择直接继承WebMvcConfigurerAdapter 。然后根据项目的需要实现接口中特定的方法，最后要注意的是，要在配置类上标注@EnableWebMvc。   
　　到这里可能有人会问，我怎么知道实现哪些方法？具体该怎么配？它们之间的处理流程是怎样的？好的，别急，我们一步步来。   
　　首先第一步，我们需要知道WebMvcConfigurer 接口都提供了哪些回调方法？

## WebMvcConfigurer

package org.springframework.web.servlet.config.annotation;

/\*\*

\* 篇幅原因，我们先只介绍Spring mvc常用的一些方法

\*/

public interface WebMvcConfigurer {

void addFormatters(FormatterRegistry registry);

void configureMessageConverters(List<HttpMessageConverter<?>> converters);

void extendMessageConverters(List<HttpMessageConverter<?>> converters);

Validator getValidator();

/\* 配置内容裁决的一些选项\*/

void configureContentNegotiation(ContentNegotiationConfigurer configurer);

void configureAsyncSupport(AsyncSupportConfigurer configurer);

/\* @since 4.0.3 \*/

void configurePathMatch(PathMatchConfigurer configurer);

/\*参数解析\*/

void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers);

/\*返回值解析\*/

void addReturnValueHandlers(List<HandlerMethodReturnValueHandler> returnValueHandlers);

/\*异常处理\*/

void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> exceptionResolvers);

void extendHandlerExceptionResolvers(List<HandlerExceptionResolver> exceptionResolvers);

void addInterceptors(InterceptorRegistry registry);

MessageCodesResolver getMessageCodesResolver();

void addViewControllers(ViewControllerRegistry registry);

/\*\*

\* 这里配置视图解析器

\*/

void configureViewResolvers(ViewResolverRegistry registry);

/\*\*

\*静态资源处理

\*/

void addResourceHandlers(ResourceHandlerRegistry registry);

void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer);

void addCorsMappings(CorsRegistry registry);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54

下面我们开始着重讲解以下几个常用的方法：

void configureViewResolvers(ViewResolverRegistry registry);

void configureContentNegotiation(ContentNegotiationConfigurer configurer);

void addViewControllers(ViewControllerRegistry registry);

void addResourceHandlers(ResourceHandlerRegistry registry);

void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer);

* 1
* 2
* 3
* 4
* 5

#### 1. configureViewResolvers(ViewResolverRegistry registry)

　　从方法名称我们就能看出这个方法是用来配置视图解析器的，该方法的参数ViewResolverRegistry 是一个注册器，用来注册你想自定义的视图解析器等。ViewResolverRegistry 常用的几个方法：

##### １)．enableContentNegotiation

/\*\* 启用内容裁决视图解析器\*/

public void enableContentNegotiation(View... defaultViews) {

initContentNegotiatingViewResolver(defaultViews);

}

* 1
* 2
* 3
* 4

　　该方法会创建一个内容裁决解析器ContentNegotiatingViewResolver ，该解析器不进行具体视图的解析，而是管理你注册的所有视图解析器，所有的视图会先经过它进行解析，然后由它来决定具体使用哪个解析器进行解析。具体的映射规则是根据请求的media types来决定的。

##### 2). 　UrlBasedViewResolverRegistration

public UrlBasedViewResolverRegistration jsp(String prefix, String suffix) {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix(prefix);

resolver.setSuffix(suffix);

this.viewResolvers.add(resolver);

return new UrlBasedViewResolverRegistration(resolver);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

　　该方法会注册一个内部资源视图解析器InternalResourceViewResolver 显然访问的所有jsp都是它进行解析的。该方法参数用来指定路径的前缀和文件后缀，如：

registry.jsp("/WEB-INF/jsp/", ".jsp");

* 1

　　对于以上配置，假如返回的视图名称是example，它会返回/WEB-INF/jsp/example.jsp给前端，找不到则报404。 

##### 3). 　beanName

public void beanName() {

BeanNameViewResolver resolver = new BeanNameViewResolver();

this.viewResolvers.add(resolver);

}

* 1
* 2
* 3
* 4

　　该方法会注册一个BeanNameViewResolver 视图解析器，这个解析器是干嘛的呢？它主要是将视图名称解析成对应的bean。什么意思呢？假如返回的视图名称是example，它会到spring容器中找有没有一个叫example的bean，并且这个bean是View.class类型的？如果有，返回这个bean。 

##### 4). 　viewResolver

public void viewResolver(ViewResolver viewResolver) {

if (viewResolver instanceof ContentNegotiatingViewResolver) {

throw new BeanInitializationException(

"addViewResolver cannot be used to configure a ContentNegotiatingViewResolver. Please use the method enableContentNegotiation instead.");

}

this.viewResolvers.add(viewResolver);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

　　这个方法想必看名字就知道了，它就是用来注册各种各样的视图解析器的，包括自己定义的。 

#### 2. configureContentNegotiation(ContentNegotiationConfigurer configurer)

　　上面一节我们讲了configureViewResolvers 方法，假如在该方法中我们启用了内容裁决解析器，那么configureContentNegotiation(ContentNegotiationConfigurer configurer) 这个方法是专门用来配置内容裁决的一些参数的。这个比较简单，我们直接通过一个例子看： 

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

/\* 是否通过请求Url的扩展名来决定media type \*/

configurer.favorPathExtension(true)

/\* 不检查Accept请求头 \*/

.ignoreAcceptHeader(true)

.parameterName("mediaType")

/\* 设置默认的media yype \*/

.defaultContentType(MediaType.TEXT\_HTML)

/\* 请求以.html结尾的会被当成MediaType.TEXT\_HTML\*/

.mediaType("html", MediaType.TEXT\_HTML)

/\* 请求以.json结尾的会被当成MediaType.APPLICATION\_JSON\*/

.mediaType("json", MediaType.APPLICATION\_JSON);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

　到这里我们就可以举个例子来进一步熟悉下我们上面讲的知识了，假如我们MVC的配置如下：

@EnableWebMvc

@Configuration

public class MvcConfig extends WebMvcConfigurerAdapter {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.jsp("/WEB-INF/jsp/", ".jsp");

registry.enableContentNegotiation(new MappingJackson2JsonView());

}

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.favorPathExtension(true)

.ignoreAcceptHeader(true)

.parameterName("mediaType")

.defaultContentType(MediaType.TEXT\_HTML)

.mediaType("html", MediaType.TEXT\_HTML)

.mediaType("json", MediaType.APPLICATION\_JSON);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

　　controller的代码如下：

@Controller

public class ExampleController {

@RequestMapping("/example1")

public ModelAndView example1() {

Map<String, String> map = new HashMap();

map.put("1", "a");

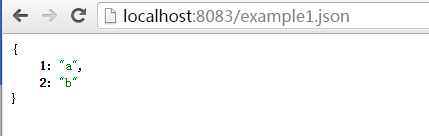
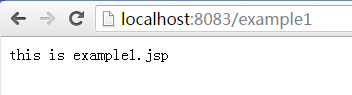
map.put("2", "b");

return new ModelAndView("example1", map);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

　　在WEB-INF/jsp目录下创建一个example1.jsp文件，内容随意。现在启动tomcat，在浏览器输入以下链接：<http://localhost:8080/example1.json>，浏览器返回如下：   
　　   
在浏览器输入<http://localhost:8080/example1> 或者<http://localhost:8080/example1.html>，返回如下：   
   
　　显然，两次使用了不同的视图解析器，那么底层到底发生了什么？在配置里我们注册了两个视图解析器：ContentNegotiatingViewResolver和 InternalResourceViewResolver，还有一个默认视图：MappingJackson2JsonView。controller执行完毕之后返回一个ModelAndView，其中视图的名称为example1。返回首先会交给ContentNegotiatingViewResolver 进行视图解析处理，而ContentNegotiatingViewResolver 会先把视图名example1交给它持有的所有ViewResolver尝试进行解析（本实例中只有InternalResourceViewResolver），然后根据请求的mediaType，再将example1.mediaType（这里是example1.json 和example1.html）作为视图名让所有视图解析器解析一遍，两步解析完毕之后会获得一堆候选的List<View> 再加上默认的MappingJackson2JsonView ，最后根据请求的media type从候选的List<View> 中选择一个最佳的返回，至此视图解析完毕。现在就可以理解上例中为何请求链接加上.json 和不.json 结果会不一样。当加上.json 时，表示请求的media type 为MediaType.APPLICATION\_JSON，而InternalResourceViewResolver 解析出来的视图的ContentType与其不符，而与MappingJackson2JsonView 的ContentType相符，所以选择了MappingJackson2JsonView 作为视图返回。当不加.json 请求时，默认的media type 为MediaType.TEXT\_HTML，所以就使用了InternalResourceViewResolver解析出来的视图作为返回值了。我想看到这里你已经大致可以自定义视图了。

#### 3. addViewControllers(ViewControllerRegistry registry)

　　此方法可以很方便的实现一个请求到视图的映射，而无需书写controller，例如： 

@Override

public void addViewControllers(ViewControllerRegistry registry){

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

}

* 1
* 2
* 3
* 4

　　这是访问${domain}/login时，会直接返回login页面。

#### 4. addResourceHandlers(ResourceHandlerRegistry registry)

　　此方法用来专门注册一个Handler，来处理静态资源的，例如：图片，js，css等。举例： 

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resource/\*\*").addResourceLocations("/WEB-INF/static/");

}

* 1
* 2
* 3
* 4

　　当你请求<http://localhost:8083/resource/1.png>时，会把/WEB-INF/static/1.png返回。**注意**：这里的静态资源是放置在**WEB-INF**目录下的。

#### 5. configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer)

　　用法：

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

* 1
* 2
* 3
* 4

　　此时会注册一个默认的Handler：DefaultServletHttpRequestHandler，这个Handler也是用来处理静态文件的，它会尝试映射/\*。当DispatcherServelt映射/时（/ 和/\* 是有区别的），并且没有找到合适的Handler来处理请求时，就会交给DefaultServletHttpRequestHandler 来处理。**注意**：这里的静态资源是放置在web根目录下，而非**WEB-INF** 下。   
　　可能这里的描述有点不好懂（我自己也这么觉得），所以简单举个例子，例如：在**webroot**目录下有一个图片：1.png 我们知道Servelt规范中web根目录（webroot）下的文件可以直接访问的，但是由于DispatcherServlet配置了映射路径是：/ ，它**几乎**把所有的请求都拦截了，从而导致1.png 访问不到，这时注册一个DefaultServletHttpRequestHandler 就可以解决这个问题。其实可以理解为DispatcherServlet破坏了Servlet的一个特性（根目录下的文件可以直接访问），DefaultServletHttpRequestHandler是帮助回归这个特性的。

## 题外话

　 问：　/和/\*　有什么区别？   
　 答： /会拦截除了**jsp**以外的所有url，/\* 会拦截所有url，包括**jsp**。例如：在webroot下面有一个test.jsp,当DispatcherServlet 配置映射/ 时，浏览器输入：<http://localhost:8083/test.jsp> 这个jsp是可以直接访问的，并且不经过DispatcherServlet ，而当DispatcherServlet配置映射/\* 时，这个请求就会被DispatcherServlet 拦截。