

课程内容：

- 1、Spring MVC处理请求的基本流程分析
- 2、四种Handler的作用与源码实现
- 3、三种HandlerMapping的作用与源码实现
- 4、四种HandlerAdapter的作用与源码实现
- 5、方法参数解析器的作用及源码实现
- 6、方法返回值处理器的作用及源码实现

【有道云笔记】19-SpringMVC启动与请求处理流程解析

<https://note.youdao.com/s/CS6qoFMf>

原理流程图：<https://www.processon.com/view/link/63f1d5cc2f69f86c1f96ee9c->

SpringMVC的作用毋庸置疑，虽然我们现在都是用SpringBoot，但是SpringBoot中仍然是在使用SpringMVC来处理请求。

我们在使用SpringMVC时，传统的方式是通过定义web.xml，比如：

```
1  <web-app>
2
3      <servlet>
4          <servlet-name>app</servlet-name>
5          <servlet-
6      class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
7          <init-param>
8              <param-name>contextConfigLocation</param-name>
9              <param-value>/WEB-INF/spring.xml</param-value>
10             </init-param>
11             <load-on-startup>1</load-on-startup>
12         </servlet>
13
14         <servlet-mapping>
15             <servlet-name>app</servlet-name>
16             <url-pattern>/app/*</url-pattern>
17         </servlet-mapping>
18 </web-app>
```

我们只要定义这样的一个web.xml，然后启动Tomcat，那么我们就能正常使用SpringMVC了。

SpringMVC中，最为核心的就是DispatcherServlet，在启动Tomcat的过程中：

1. Tomcat会先创建DispatcherServlet对象
2. 然后调用DispatcherServlet对象的init()

而在init()方法中，会创建一个Spring容器，并且添加一个ContextRefreshListener监听器，该监听器会监听ContextRefreshedEvent事件（Spring容器启动完成后就会发布这个事件），也就是说Spring容器启动完成后，就会执行ContextRefreshListener中的onApplicationEvent()方法，从而最终会执行DispatcherServlet中的initStrategies()，这个方法中会初始化更多内容：

```
1  protected void initStrategies(ApplicationContext context) {  
2      initMultipartResolver(context);  
3      initLocaleResolver(context);  
4      initThemeResolver(context);  
5  
6      initHandlerMappings(context);  
7      initHandlerAdapters(context);  
8  
9      initHandlerExceptionResolvers(context);  
10     initRequestToViewNameTranslator(context);  
11     initViewResolvers(context);  
12     initFlashMapManager(context);  
13 }
```

其中最为核心的就是HandlerMapping和HandlerAdapter。

什么是Handler?

Handler表示请求处理器，在SpringMVC中有四种Handler：

1. 实现了Controller接口的Bean对象
2. 实现了HttpRequestHandler接口的Bean对象
3. 添加了@RequestMapping注解的方法
4. 一个HandlerFunction对象

比如实现了Controller接口的Bean对象：

```
1 @Component("/test")
2 public class ZhouyuBeanNameController implements Controller {
3
4     @Override
5     public ModelAndView handleRequest(HttpServletRequest request,
6     HttpServletResponse response) throws Exception {
7         System.out.println("zhouyu");
8         return new ModelAndView();
9     }
10 }
```

实现了HttpRequestHandler接口的Bean对象：

```
1 @Component("/test")
2 public class ZhouyuBeanNameController implements HttpRequestHandler {
3
4     @Override
5     public void handleRequest(HttpServletRequest request, HttpServletResponse
6     response) throws ServletException, IOException {
7         System.out.println("zhouyu");
8     }
9 }
```

添加了@RequestMapping注解的方法：

```

1 @RequestMapping
2 @Component
3 public class ZhouyuController {
4
5     @Autowired
6     private ZhouyuService zhouyuService;
7
8     @RequestMapping(method = RequestMethod.GET, path = "/test")
9     @ResponseBody
10    public String test(String username) {
11        return "zhouyu";
12    }
13
14 }

```

一个HandlerFunction对象（以下代码中有两个）：

```

1 @ComponentScan("com.zhouyu")
2 @Configuration
3 public class AppConfig {
4
5     @Bean
6     public RouterFunction<ServerResponse> person() {
7         return route()
8             .GET("/app/person", request ->
9                 ServerResponse.status(HttpStatus.OK).body("Hello GET"))
10            .POST("/app/person", request ->
11                ServerResponse.status(HttpStatus.OK).body("Hello POST"))
12            .build();
13    }
14 }

```

什么是HandlerMapping?

HandlerMapping负责去寻找Handler，并且保存路径和Handler之间的映射关系。

因为有不同的Handler，所以在SpringMVC中会由不同的HandlerMapping来负责寻找Handler，比如：

1. BeanNameUrlHandlerMapping：负责Controller接口和HttpRequestHandler接口
2. RequestMappingHandlerMapping：负责@RequestMapping的方法
3. RouterFunctionMapping：负责RouterFunction以及其中的HandlerFunction

BeanNameUrlHandlerMapping的寻找流程：

1. 找出Spring容器中所有的beanName
2. 判断beanName是不是以“/”开头
3. 如果是，则把它当作一个Handler，并把beanName作为key，bean对象作为value存入handlerMap中
4. handlerMap就是一个Map

RequestMappingHandlerMapping的寻找流程：

1. 找出Spring容器中所有beanType
2. 判断beanType是不是有@Controller注解，或者是不是有@RequestMapping注解
3. 判断成功则继续找beanType中加了@RequestMapping的Method
4. 并解析@RequestMapping中的内容，比如method、path，封装为一个RequestMappingInfo对象
5. 最后把RequestMappingInfo对象做为key，Method对象封装为HandlerMethod对象后作为value，存入registry中
6. registry就是一个Map

RouterFunctionMapping的寻找流程会有些区别，但是大体是差不多的，相当于是一个path对应一个HandlerFunction。

各个HandlerMapping除开负责寻找Handler并记录映射关系之外，自然还需要根据请求路径找到对应的Handler，在源码中这三个HandlerMapping有一个共同的父类AbstractHandlerMapping

AbstractHandlerMapping实现了HandlerMapping接口，并实现了getHandler(HttpServletRequest request)方法。

AbstractHandlerMapping会负责调用子类的getHandlerInternal(HttpServletRequest request)方法从而找到请求对应的Handler，然后AbstractHandlerMapping负责将Handler和应用中所配置的HandlerInterceptor整合成为一个HandlerExecutionChain对象。

所以寻找Handler的源码实现在各个HandlerMapping子类中的getHandlerInternal()中，根据请求路径找到Handler的过程并不复杂，因为路径和Handler的映射关系已经存在Map中了。

比较困难的点在于，当DispatcherServlet接收到一个请求时，该利用哪个HandlerMapping来寻找Handler呢？看源码：

```
1  protected HandlerExecutionChain getHandler(HttpServletRequest request) throws
   Exception {
2      if (this.handlerMappings != null) {
3          for (HandlerMapping mapping : this.handlerMappings) {
4              HandlerExecutionChain handler = mapping.getHandler(request);
5              if (handler != null) {
6                  return handler;
7              }
8          }
9      }
10     return null;
11 }
```

很简单，就是遍历，找到就返回，默认顺序为：

所以BeanNameUrlHandlerMapping的优先级最高，比如：

```
1  @Component("/test")
2  public class ZhouyuBeanNameController implements Controller {
3
4      @Override
5      public ModelAndView handleRequest(HttpServletRequest request,
   HttpServletRequest response) throws Exception {
6          System.out.println("Hello zhouyu");
7          return new ModelAndView();
8      }
9  }
```

```
1 @RequestMapping(method = RequestMethod.GET, path = "/test")
2 @ResponseBody
3 public String test(String username) {
4     return "Hi zhouyu";
5 }
```

请求路径都是/test，但是最终是Controller接口的会生效。

什么是HandlerAdapter?

找到了Handler之后，接下来就该去执行了，比如执行下面这个test()

```
1 @RequestMapping(method = RequestMethod.GET, path = "/test")
2 @ResponseBody
3 public String test(String username) {
4     return "zhouyu";
5 }
```

但是由于有不同种类的Handler，所以执行方式是不一样的，再来总结一下Handler的类型：

1. 实现了Controller接口的Bean对象，执行的是Bean对象中的handleRequest()
2. 实现了HttpRequestHandler接口的Bean对象，执行的是Bean对象中的handleRequest()
3. 添加了@RequestMapping注解的方法，具体为一个HandlerMethod，执行的就是当前加了注解的方法
4. 一个HandlerFunction对象，执行的是HandlerFunction对象中的handle()

所以，按逻辑来说，找到Handler之后，我们得判断它的类型，比如代码可能是这样的：

```

1 Object handler = mappedHandler.getHandler();
2 if (handler instanceof Controller) {
3     ((Controller)handler).handleRequest(request, response);
4 } else if (handler instanceof HttpRequestHandler) {
5     ((HttpRequestHandler)handler).handleRequest(request, response);
6 } else if (handler instanceof HandlerMethod) {
7     ((HandlerMethod)handler).getMethod().invoke(...);
8 } else if (handler instanceof HandlerFunction) {
9     ((HandlerFunction)handler).handle(...);
10 }

```

但是SpringMVC并不是这么写的，还是采用的**适配模式**，把不同种类的Handler适配成一个HandlerAdapter，后续再执行HandlerAdapter的handle()方法就能执行不同种类Handler对应的方法。

针对不同的Handler，会有不同的适配器：

1. HttpRequestHandlerAdapter
2. SimpleControllerHandlerAdapter
3. RequestMappingHandlerAdapter
4. HandlerFunctionAdapter

适配逻辑为：

```

1 protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
2     if (this.handlerAdapters != null) {
3         for (HandlerAdapter adapter : this.handlerAdapters) {
4             if (adapter.supports(handler)) {
5                 return adapter;
6             }
7         }
8     }
9     throw new ServletException("No adapter for handler [" + handler +
10                                "]: The DispatcherServlet configuration needs to
    include a HandlerAdapter that supports this handler");
11 }

```

传入handler，遍历上面四个Adapter，谁支持就返回谁，比如判断的代码依次为：


```

1 public boolean supports(Object handler) {
2     return (handler instanceof HttpRequestHandler);
3 }
4
5 public boolean supports(Object handler) {
6     return (handler instanceof Controller);
7 }
8
9 public final boolean supports(Object handler) {
10    return (handler instanceof HandlerMethod && supportsInternal((HandlerMethod)
11        handler));
12 }
13
14 public boolean supports(Object handler) {
15    return handler instanceof HandlerFunction;
16 }

```

根据Handler适配出了对应的HandlerAdapter后，就执行具体HandlerAdapter对象的handle()方法了，比如：

HttpRequestHandlerAdapter的handle()：

```

1 public ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
2     Object handler)
3     throws Exception {
4     ((HttpRequestHandler) handler).handleRequest(request, response);
5     return null;
6 }

```

SimpleControllerHandlerAdapter的handle()：

```
1 public ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
   Object handler)
2
   throws Exception {
3     return ((Controller) handler).handleRequest(request, response);
4 }
```

HandlerFunctionAdapter的handle():

```
1 HandlerFunction<?> handlerFunction = (HandlerFunction<?>) handler;
2 serverResponse = handlerFunction.handle(serverRequest);
```

因为这三个接收的直接就是Request对象，不用SpringMVC做额外的解析，所以比较简单，比较复杂的是RequestMappingHandlerAdapter，它执行的是加了@RequestMapping的方法，而这种方法的写法可以是多种多样，SpringMVC需要根据方法的定义去解析Request对象，从请求中获取出对应的数据然后传递给方法，并执行。

@RequestMapping方法参数解析

当SpringMVC接收到请求，并找到了对应的Method之后，就要执行该方法了，不过在执行之前需要根据方法定义参数信息，从请求中获取出对应的数据，然后将数据传给方法并执行。

一个HttpServletRequest通常有：

1. request parameter
2. request attribute
3. request session
4. request header
5. request body

比如如下几个方法：

```
1 public String test(String username) {
2     return "zhouyu";
3 }
```

表示要从request parameter中获取key为username的value

```
1 public String test(@RequestParam("uname") String username) {  
2     return "zhouyu";  
3 }
```

表示要从request parameter中获取key为uname的value

```
1 public String test(@ModelAttribute String username) {  
2     return "zhouyu";  
3 }
```

表示要从request attribute中获取key为username的value

```
1 public String test(@SessionAttribute String username) {  
2     return "zhouyu";  
3 }
```

表示要从request session中获取key为username的value

```
1 public String test(@RequestHeader String username) {  
2     return "zhouyu";  
3 }
```

表示要从request header中获取key为username的value

```
1 public String test(@RequestBody String username) {  
2     return "zhouyu";  
3 }
```

表示获取整个请求体

所以，我们发现SpringMVC要去解析方法参数，看该参数到底是要获取请求中的哪些信息。

而这个过程，源码中是通过HandlerMethodArgumentResolver来实现的，比如：

1. RequestParamMethodArgumentResolver：负责处理@RequestParam
2. RequestHeaderMethodArgumentResolver：负责处理@RequestHeader
3. SessionAttributeMethodArgumentResolver：负责处理@SessionAttribute
4. RequestAttributeMethodArgumentResolver：负责处理@RequestAttribute
5. RequestResponseBodyMethodProcessor：负责处理@RequestBody
6. 还有很多其他的...

而在判断某个参数该由哪个HandlerMethodArgumentResolver处理时，也是很粗暴：

```
1 private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) {  
2  
3     HandlerMethodArgumentResolver result = this.argumentResolverCache.get(parameter);  
4     if (result == null) {  
5         for (HandlerMethodArgumentResolver resolver : this.argumentResolvers) {  
6             if (resolver.supportsParameter(parameter)) {  
7                 result = resolver;  
8                 this.argumentResolverCache.put(parameter, result);  
9                 break;  
10            }  
11        }  
12    }  
13    return result;  
14  
15 }
```

就是遍历所有的HandlerMethodArgumentResolver，哪个能支持处理当前这个参数就由哪个处理。

比如：

```
1 @RequestMapping(method = RequestMethod.GET, path = "/test")
2 @ResponseBody
3 public String test(@RequestParam @SessionAttribute String username) {
4     System.out.println(username);
5     return "zhouyu";
6 }
```

以上代码的username将对应RequestParam中的username，而不是session中的，因为在源码中RequestParamMethodArgumentResolver更靠前。

当然HandlerMethodArgumentResolver也会负责从request中获取对应的数据，对应的是resolveArgument()方法。

比如RequestParamMethodArgumentResolver：

```

1  protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest
   request) throws Exception {
2      HttpServletRequest servletRequest =
   request.getNativeRequest(HttpServletRequest.class);
3
4      if (servletRequest != null) {
5          Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name,
   parameter, servletRequest);
6          if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
7              return mpArg;
8          }
9      }
10
11     Object arg = null;
12     MultipartRequest multipartRequest =
   request.getNativeRequest(MultipartRequest.class);
13     if (multipartRequest != null) {
14         List<MultipartFile> files = multipartRequest.GetFiles(name);
15         if (!files.isEmpty()) {
16             arg = (files.size() == 1 ? files.get(0) : files);
17         }
18     }
19     if (arg == null) {
20         String[] paramValues = request.getParameterValues(name);
21         if (paramValues != null) {
22             arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
23         }
24     }
25     return arg;
26 }
27

```

核心是：

```
1  if (arg == null) {
2      String[] paramValues = request.getParameterValues(name);
3      if (paramValues != null) {
4          arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
5      }
6  }
```

按同样的思路，可以找到方法中每个参数所要求的值，从而执行方法，得到方法的返回值。

@RequestMapping方法返回值解析

而方法返回值，也会分为不同的情况。比如有没有加@ResponseBody注解，如果方法返回一个String：

1. 加了@ResponseBody注解：表示直接将这个String返回给浏览器
2. 没有加@ResponseBody注解：表示应该根据这个String找到对应的页面，把页面返回给浏览器

在SpringMVC中，会利用HandlerMethodReturnValueHandler来处理返回值：

1. RequestResponseBodyMethodProcessor：处理加了@ResponseBody注解的情况
2. ViewNameMethodReturnValueHandler：处理没有加@ResponseBody注解并且返回值类型为String的情况
3. ModelMethodProcessor：处理返回值是Model类型的情况
4. 还有很多其他的...

我们这里只讲RequestResponseBodyMethodProcessor，因为它会处理加了@ResponseBody注解的情况，也是目前我们用得最多的情况。

RequestResponseBodyMethodProcessor相当于会把方法返回的对象直接响应给浏览器，如果返回的是一个字符串，那么好说，直接把字符串响应给浏览器，那如果返回的是一个Map呢？是一个User对象呢？该怎么把这些复杂对象响应给浏览器呢？

处理这块，SpringMVC会利用HttpMessageConverter来处理，比如默认情况下，SpringMVC会有4个HttpMessageConverter：

1. ByteArrayHttpMessageConverter：处理返回值为**字节数组**的情况，把字节数组返回给浏览器
2. StringHttpMessageConverter：处理返回值为**字符串**的情况，把字符串按指定的编码序列号后返回给浏览器
3. SourceHttpMessageConverter：处理返回值为**XML对象**的情况，比如把DOMSource对象返回给浏览器
4. AllEncompassingFormHttpMessageConverter：处理返回值为**MultiValueMap对象**的情况

StringHttpMessageConverter的源码也比较简单：

```
1 protected void writeInternal(String str, HttpOutputMessage outputMessage) throws
   IOException {
2     HttpHeaders headers = outputMessage.getHeaders();
3     if (this.writeAcceptCharset && headers.get(HttpHeaders.ACCEPT_CHARSET) == null) {
4         headers.setAcceptCharset(getAcceptedCharsets());
5     }
6     Charset charset = getContentTypeCharset(headers.getContentType());
7     StreamUtils.copy(str, charset, outputMessage.getBody());
8 }
```

先看有没有设置Content-Type，如果没有设置则取默认的，默认为ISO-8859-1，所以默认情况下返回中文会乱码，可以通过以下来中方式来解决：

```
1 @RequestMapping(method = RequestMethod.GET, path = "/test", produces =
   {"application/json;charset=UTF-8"})
2 @ResponseBody
3 public String test() {
4     return "周瑜";
5 }
```



```

1  @ComponentScan("com.zhouyu")
2  @Configuration
3  @EnableWebMvc
4  public class AppConfig implements WebMvcConfigurer {
5
6      @Override
7      public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
8
9          StringHttpMessageConverter messageConverter = new
StringHttpMessageConverter();
10         messageConverter.setDefaultCharset(StandardCharsets.UTF_8);
11         converters.add(messageConverter);
12     }
13 }

```

不过以上四个Converter是不能处理Map对象或User对象的，所以如果返回的是Map或User对象，那么得单独配置一个Converter，比如MappingJackson2HttpMessageConverter，这个Converter比较大，能把String、Map、User对象等等都能转化成JSON格式。

```

1  @ComponentScan("com.zhouyu")
2  @Configuration
3  @EnableWebMvc
4  public class AppConfig implements WebMvcConfigurer {
5
6      @Override
7      public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
8
9          MappingJackson2HttpMessageConverter messageConverter = new
MappingJackson2HttpMessageConverter();
10         messageConverter.setDefaultCharset(StandardCharsets.UTF_8);
11         converters.add(messageConverter);
12     }
13 }

```

具体转化的逻辑就是Jackson2的转化逻辑。

总结

以上就是整个SpringMVC从启动到处理请求，从接收请求到执行方法的整体流程。