课程内容：

1、方法参数解析源码分析
2、文件上传MultipartFile源码解析
3、方法返回值解析源码分析
4、视图解析核心源码分析
5、SpringMVC拦截器源码解析
6、@EnableWebMvc源码解析
7、WebApplicationInitializer使用方式
8、SpringMVC父子容器介绍与源码分析
【有道云笔记】20-SpringMVC重点功能底层源码解析
https://note.youdao.com/s/6F3fGagK

SpringMVC处理请求核心流程图：https://www.processon.com/view/link/63f4cf1176e6143857799c2a

课堂疑问1：

当我们使用@RequestParam，并且没有注册StringToUserEditor时，但是User中提供了一个String类型参数的构造方法时：

```
@RequestMapping(method = RequestMethod.GET, path = "/test")
@ResponseBody
public String test(@RequestParam("name") User user) {
        return user.getName();
}
```

```java
public class User {

        private String name;

        public User(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

SpringMVC在进行把String转成User对象时，会先判断有没有User类型对应的StringToUserEditor，如果有就会利用它来把String转成User对象，如果没有则会找User类中有没有String类型参数的构造方法，如果有则用该构造方法来构造出User对象。

对应的源码方法为：
org.springframework.beans.TypeConverterDelegate#convertIfNecessary(java.lang.String, java.lang.Object, java.lang.Object, java.lang.Class<T>, org.springframework.core.convert.TypeDescriptor)

```java
}
else if (convertedValue instanceof String && !requiredType.isInstance(convertedValue))
    if (conversionAttemptEx == null && !requiredType.isInterface() && !requiredType.isE
        try {
            Constructor<T> strCtor = requiredType.getConstructor(String.class);
            return BeanUtils.instantiateClass(strCtor, convertedValue);
        }
        catch (NoSuchMethodException ex) {
            // proceed with field lookup
            if (logger.isTraceEnabled()) {
```

课堂疑问2：

如果方法返回的是byte[]：

```
@RequestMapping(method = RequestMethod.GET, path = "/test")
@ResponseBody
public byte[] test() {
        byte[] bytes = new byte[1024];
        return bytes;
}
```

这种情况会直接使用ByteArrayHttpMessageConverter来处理，会直接把byte[]写入响应中：

```
public class ByteArrayHttpMessageConverter extends
AbstractHttpMessageConverter<byte[]> {

        /**
         * Create a new instance of the {@code ByteArrayHttpMessageConverter}.
         */
        public ByteArrayHttpMessageConverter() {
                super(MediaType.APPLICATION_OCTET_STREAM, MediaType.ALL);
        }


        @Override
        public boolean supports(Class<?> clazz) {
                return byte[].class == clazz;
        }

        // ...

        @Override
        protected void writeInternal(byte[] bytes, HttpOutputMessage outputMessage)
throws IOException {
                StreamUtils.copy(bytes, outputMessage.getBody());
        }

}
```

SpringMVC父子容器

我们可以在web.xml文件中这么来定义：

```
<web-app>

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

        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>/WEB-INF/spring.xml</param-value>
        </context-param>

        <servlet>
                <servlet-name>app</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>/WEB-INF/spring-mvc.xml</param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>

        <servlet-mapping>
                <servlet-name>app</servlet-name>
                <url-pattern>/app/*</url-pattern>
        </servlet-mapping>

</web-app>


```

在这个web.xml文件中，我们定义了一个listener和servlet。

父容器的创建

ContextLoaderListener的作用是用来创建一个Spring容器，就是我们说的SpringMVC父子容器中的父容器，执行流程为：

1. Tomcat启动，解析web.xml时

2. 发现定义了一个ContextLoaderListener，Tomcat就会执行该listener中的contextInitialized()方法，该方法就会去创建要给Spring容器
3. 从ServletContext中获取contextClass参数值，该参数表示所要创建的Spring容器的类型，可以在web.xml中通过<context-param>来进行配置
4. 如果没有配置该参数，那么则会从ContextLoader.properties文件中读取org.springframework.web.context.WebApplicationContext配置项的值，SpringMVC默认提供了一个ContextLoader.properties文件，内容为org.springframework.web.context.support.XmlWebApplicationContext
5. 所以XmlWebApplicationContext就是要创建的Spring容器类型
6. 确定好类型后，就用反射调用**无参构造方法**创建出来一个XmlWebApplicationContext对象
7. 然后继续从ServletContext中获取contextConfigLocation参数的值，也就是一个spring配置文件的路径
8. 把spring配置文件路径设置给Spring容器，然后调用refresh()，从而启动Spring容器，从而解析spring配置文件，从而扫描生成Bean对象等
9. 这样Spring容器就创建出来了
10. 有了Spring容器后，就会把XmlWebApplicationContext对象作为attribute设置到ServletContext中去，key为WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE
11. 把Spring容器存到ServletContext中的原因，是为了给Servlet创建出来的子容器来作为父容器的

子容器的创建

Tomcat启动过程中，执行完ContextLoaderListener的contextInitialized()之后，就会创建DispatcherServlet了，web.xml中定义DispatcherServlet时，load-on-startup为1，表示在Tomcat启动过程中要把这个DispatcherServlet创建并初始化出来，而这个过程是比较费时间的，所以要把load-on-startup设置为1，如果不为1，会在servlet接收到请求时才来创建和初始化，这样会导致请求处理比较慢。

1. Tomcat启动，解析web.xml时
2. 创建DispatcherServlet对象
3. 调用DispatcherServlet的init()
4. 从而调用initServletBean()
5. 从而调用initWebApplicationContext()，这个方法也会去创建一个Spring容器（就是子容器）
6. initWebApplicationContext()执行过程中，会先从ServletContext拿出ContextLoaderListener所创建的Spring容器（父容器），记为rootContext
7. 然后读取contextClass参数值，可以在servlet中的<init-param>标签来定义想要创建的Spring容器类型，默认为XmlWebApplicationContext
8. 然后创建一个Spring容器对象，也就是子容器
9. 将rootContext作为parent设置给子容器（父子关系的绑定）

10. 然后读取contextConfigLocation参数值，得到所配置的Spring配置文件路径
11. 然后就是调用Spring容器的refresh()方法
12. 从而完成了子容器的创建

SpringMVC初始化

子容器创建完后，还会调用一个DispatcherServlet的onRefresh()方法，这个方法会从Spring容器中获取一些特殊类型的Bean对象，并设置给DispatcherServlet对象中对应的属性，比如HandlerMapping、HandlerAdapter。

流程为：

1. 会先从Spring容器中获取HandlerMapping类型的Bean对象，如果不为空，那么就获取出来的Bean对象赋值给DispatcherServlet的handlerMappings属性
2. 如果没有获取到，则会从DispatcherServlet.properties文件中读取配置，从而得到SpringMVC默认给我们配置的HandlerMapping

DispatcherServlet.properties文件内容为：

```
# Default implementation classes for DispatcherServlet's strategy interfaces.
# Used as fallback when no matching beans are found in the DispatcherServlet context.
# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping,\
org.springframework.web.servlet.function.support.RouterFunctionMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter,\
org.springframework.web.servlet.function.support.HandlerFunctionAdapter


org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,\
org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager
```

默认提供了3个HandlerMapping，4个HandlerAdapter，这些概念在后续DispatcherServlet处理请求时都是会用到的。

值得注意的是，从配置文件读出这些类后，是会利用Spring容器去创建出来对应的Bean对象，而不是一个普通的Java对象，而如果是Bean对象，那么就会触发Bean的初始化逻辑，比如RequestMappingHandlerAdapter，后续在分析请求处理逻辑时，会发现这个类是非常重要的，而它就实现了InitializingBean接口，从而Bean对象在创建时会执行afterPropertiesSet()方法。

RequestMappingHandlerAdapter初始化

我们先可以简单理解RequestMappingHandlerAdapter，它的作用就是在收到请求时来调用请求对应的方法的，所以它需要去解析方法参数，方法返回值。

在RequestMappingHandlerAdapter的afterPropertiesSet()方法中，又会做以下事情（这些事情大家可能现在看不懂，可以后面回头再来看，我先列在这）：

1. 从Spring容器中找到加了@ControllerAdvice的Bean对象
    a. 解析出Bean对象中加了@ModelAttribute注解的Method对象，并存在modelAttributeAdviceCache这个Map中
    b. 解析出Bean对象中加了@InitBinder注解的Method对象，并存在initBinderAdviceCache这个Map中
    c. 如果Bean对象实现了RequestBodyAdvice接口或者ResponseBodyAdvice接口，那么就把这个Bean对象记录在requestResponseBodyAdvice集合中
1. 从Spring容器中获取用户定义的HandlerMethodArgumentResolver，以及SpringMVC默认提供的，整合为一个HandlerMethodArgumentResolverComposite对象，HandlerMethodArgumentResolver是用来**解析方法参数**的
2. 从Spring容器中获取用户定义的HandlerMethodReturnValueHandler，以及SpringMVC默认提供的，整合为一个HandlerMethodReturnValueHandlerComposite对象，HandlerMethodReturnValueHandler是用来**解析方法返回值**的

以上是RequestMappingHandlerAdapter这个Bean的初始化逻辑。

RequestMappingHandlerMapping初始化

RequestMappingHandlerMapping的作用是，保存我们定义了哪些@RequestMapping方法及对应的访问路径，而RequestMappingHandlerMapping的初始化就是去找到这些映射关系：

1. 找出容器中定义的所有的beanName
2. 根据beanName找出beanType
3. 判断beanType上是否有@Controller注解或@RequestMapping注解，如果有那么就表示这个Bean对象是一个Handler

4. 如果是一个Handler，就通过反射找出加了@RequestMapping注解的Method，并解析@RequestMapping注解上定义的参数信息，得到一个对应的RequestMappingInfo对象，然后结合beanType上@RequestMapping注解所定义的path，以及当前Method上@RequestMapping注解所定义的path，进行整合，则得到了当前这个Method所对应的访问路径，并设置到RequestMappingInfo对象中去
5. 所以，一个RequestMappingInfo对象就对应了一个加了@RequestMapping注解的Method，并且请求返回路径也记录在了RequestMappingInfo对象中
6. 把当前Handler，也就是beanType中的所有RequestMappingInfo都找到后，就会存到MappingRegistry对象中
7. 在存到MappingRegistry对象过程中，会像把Handler，也就是beanType，以及Method，生成一个HandlerMethod对象，其实就是表示一个方法
8. 然后获取RequestMappingInfo对象中的path
9. 把path和HandlerMethod对象存在一个Map中，属性叫做pathLookup
10. 这样在处理请求时，就可以同请求路径找到HandlerMethod，然后找到Method，然后执行了

WebApplicationInitializer的方式

除开使用web.xml外，我们还可以直接定义一个WebApplicationInitializer来使用SpringMVC，比如：

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) {

        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.register(AppConfig.class);

        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(context);
        ServletRegistration.Dynamic registration = servletContext.addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/*");
    }
}
```

```
@ComponentScan("com.zhouyu")
@Configuration
public class AppConfig  {
}
```

这种方法我们也能使用SpringMVC，流程为：

1. Tomcat启动过程中就会调用到我们所写的onStartup()
2. 从而创建一个Spring容器
3. 从而创建一个DispatcherServlet对象并初始化
4. 而DispatcherServlet初始化所做的事情和上述是一样的

那为什么Tomcat启动时能调用到MyWebApplicationInitializer中的onStartup()呢?

这个跟Tomcat的提供的扩展机制有关，在SpringMVC中有这样一个类：

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {

        @Override
        public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses,
ServletContext servletContext)
                        throws ServletException {
        // ...
        }

}
```

这个类实现了javax.servlet.ServletContainerInitializer接口，并且在SpringMVC中还有这样一个文件：META-INF/services/Tomcatjavax.servlet.ServletContainerInitializer，文件内容为org.springframework.web.SpringServletContainerInitializer。

很明显，是SPI，所以Tomcat在启动过程中会找到这个SpringServletContainerInitializer，并执行onStartup()，并且还会找到@HandlesTypes注解中所指定的WebApplicationInitializer接口的实现类，并传递给onStartup()方法，这其中就包括了我们自己定义的MyWebApplicationInitializer。

在SpringServletContainerInitializer的onStartup()中就会调用MyWebApplicationInitializer的onStartup()方法了：

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {

        @Override
        public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses,
ServletContext servletContext)
                        throws ServletException {

                List<WebApplicationInitializer> initializers = Collections.emptyList();

                if (webAppInitializerClasses != null) {
                        initializers = new ArrayList<>
(webAppInitializerClasses.size());
                        for (Class<?> waiClass : webAppInitializerClasses) {
                                // 过滤掉接口、抽象类
                                if (!waiClass.isInterface() &&
!Modifier.isAbstract(waiClass.getModifiers()) &&
WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
                                        try {
                                                // 实例化
initializers.add((WebApplicationInitializer)
ReflectionUtils.accessibleConstructor(waiClass).newInstance());
                                        }
                                        catch (Throwable ex) {
                                                throw new ServletException("Failed to
instantiate WebApplicationInitializer class", ex);
                                        }
                                }
                        }
                }

                if (initializers.isEmpty()) {
                        servletContext.log("No Spring WebApplicationInitializer types
detected on classpath");
                        return;
                }

                servletContext.log(initializers.size() + " Spring
WebApplicationInitializers detected on classpath");
```

```
                AnnotationAwareOrderComparator.sort(initializers);
                // 调用initializer.onStartup()
                for (WebApplicationInitializer initializer : initializers) {
                        initializer.onStartup(servletContext);
                }
        }

}
```

方法参数解析

在RequestMappingHandlerAdapter的初始化逻辑中会设置一些默认的HandlerMethodArgumentResolver，他们就是用来解析各种类型的方法参数的。

比如：

1. RequestParamMethodArgumentResolver，用来解析加了@RequestParam注解的参数，或者什么都没加的基本类型参数（非基本类型的会被ServletModelAttributeMethodProcessor处理）
2. PathVariableMethodArgumentResolver，用来解析加了@PathVariable注解的参数
3. RequestHeaderMethodArgumentResolver，用来解析加了@RequestHeader注解的参数

比如RequestParamMethodArgumentResolver中是这么处理的:

```
protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
    HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);

    // ...

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

很简单了，就是把请求中对应的parameterValue拿出来，最为参数值传递给方法。

其他的类似，都是从请求中获取相对应的信息传递给参数。

但是需要注意的是，我们从请求中获取的值可能很多时候都是字符串，那如果参数类型不是String，该怎么办呢？这就需要进行类型转换了，比如代码是这么写的：

```
@RequestMapping(method = RequestMethod.GET, path = "/test")
@ResponseBody
public String test(@RequestParam User user) {
    System.out.println(user.getName());
return "hello zhouyu";
}
```

表示要获取请求中user对应的parameterValue，但是我们发请求时是这么发的：

```
http://localhost:8080/tuling-web/app/test?user=zhouyu
```

那么SpringMVC就需要将字符串zhouyu转换成为User对象，这就需要我们自定义类型转换器了，比如：

```
/**
 * 作者：周瑜大都督
 */
public class StringToUserEditor extends PropertyEditorSupport {

        @Override
        public void setAsText(String text) throws IllegalArgumentException {
                User user = new User();
                user.setName(text);
                this.setValue(user);
        }
}
```

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.registerCustomEditor(User.class, new StringToUserEditor());
}
```

Spring默认提供的Converter：
org.springframework.core.convert.support.DefaultConversionService#addCollectionConverters

MultipartFile解析

文件上传代码如下：

```
@RequestMapping(method = RequestMethod.POST, path = "/test")
@ResponseBody
public String test(MultipartFile file) {
    System.out.println(file.getName());
        return "hello zhouyu";
}
```

要理解SpringMVC的文件上传，我们得先回头看看直接基于Servlet的文件上传，代码如下：

```java
@WebServlet(name = "uploadFileServlet", urlPatterns = "/uploadFile")
@MultipartConfig
public class UploadFileServlet extends HttpServlet {

        public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        Collection<Part> parts = request.getParts();

                for (Part part : parts) {
                        //content-disposition对于的内容为：form-data; name="file"; filename="zhouyu.xlsx"
                        String header = part.getHeader("content-disposition");

                        String fileName = getFileName(header);

                        if (fileName != null) {
                                part.write("D://upload" + File.separator + fileName);
                        } else {
                                System.out.println(part.getName());
                        }

                }

                response.setCharacterEncoding("utf-8");
                response.setContentType("text/html;charset=utf-8");
                PrintWriter out = response.getWriter();
                out.println("上传成功");
                out.flush();
                out.close();
        }

        public String getFileName(String header) {
                String[] arr = header.split(";");
                if (arr.length < 3) return null;
                String[] arr2 = arr[2].split("=");
                String fileName = arr2[1].substring(arr2[1].lastIndexOf("\\") + 1).replaceAll("\"", "");
                return fileName;
        }
```

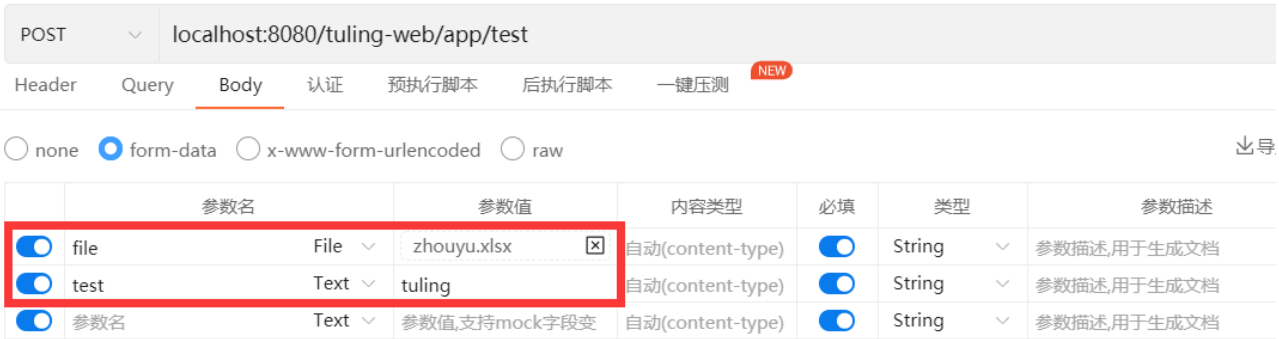
```
38
39 }
```

可以看到第一行代码是：

```
1 Collection<Part> parts = request.getParts();
```

从request中拿到了一个Part集合，而这个集合中Part可以表示一个文件，也可以表示一个字符串。

比如发送这么一个请求：



那么这个请求中就会有两个Part，一个Part表示文件，一个Part表示文本。

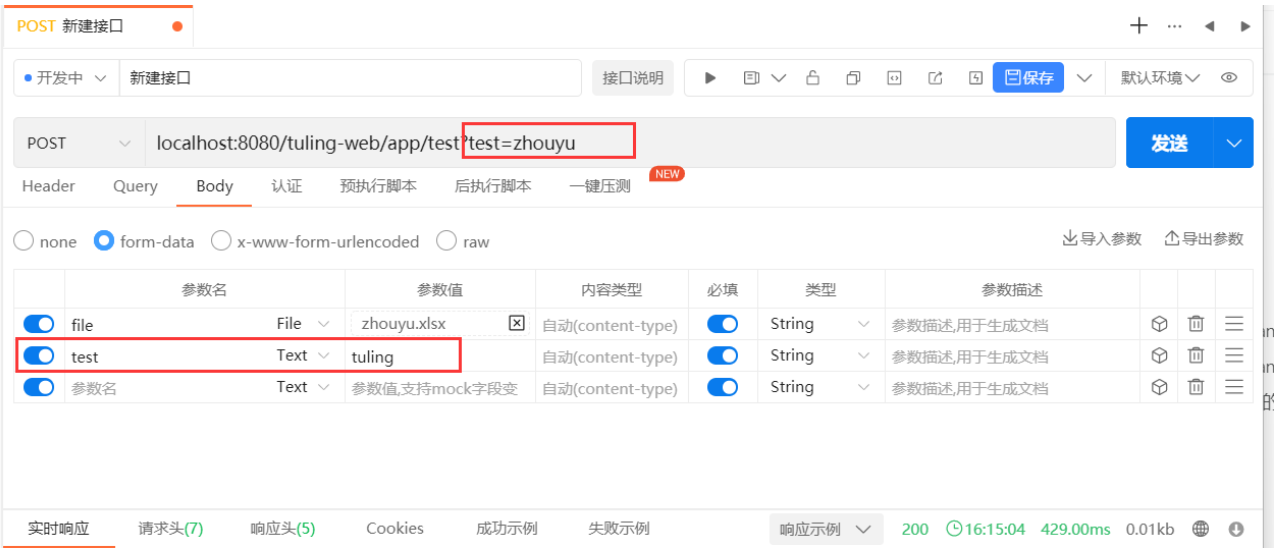
有了这个知识点，我们再来看Controller中的代码：

```
1 @RequestMapping(method = RequestMethod.POST, path = "/test")
2 @ResponseBody
3 public String test(MultipartFile file, String test) {
4     System.out.println(file.getName());
5     System.out.println(test);
6     return "hello zhouyu";
7 }
```

方法中的两个参数分别表示：

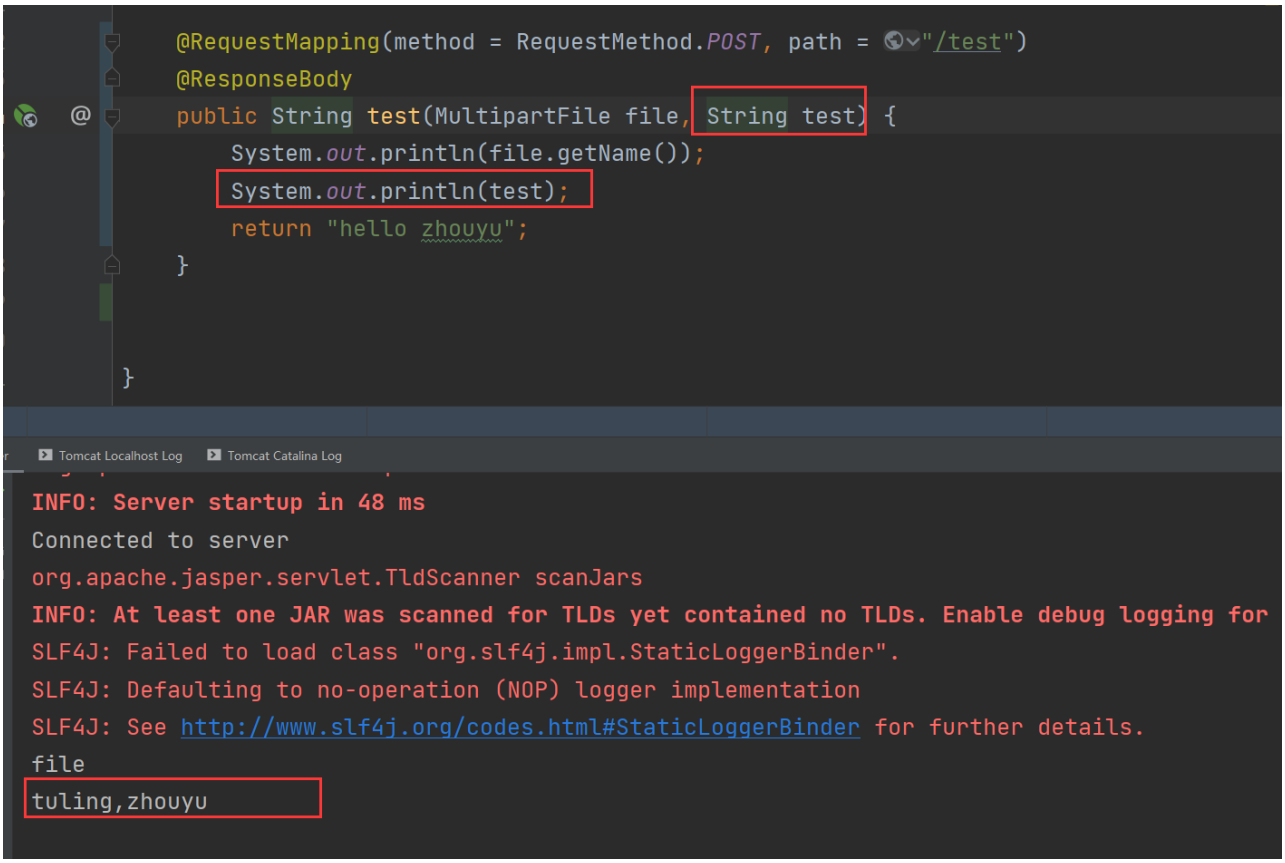
1. file对应的是文件Part
2. test对应的就是文本Part

那有同学可能会有疑问，假如我请求是这么发的呢：

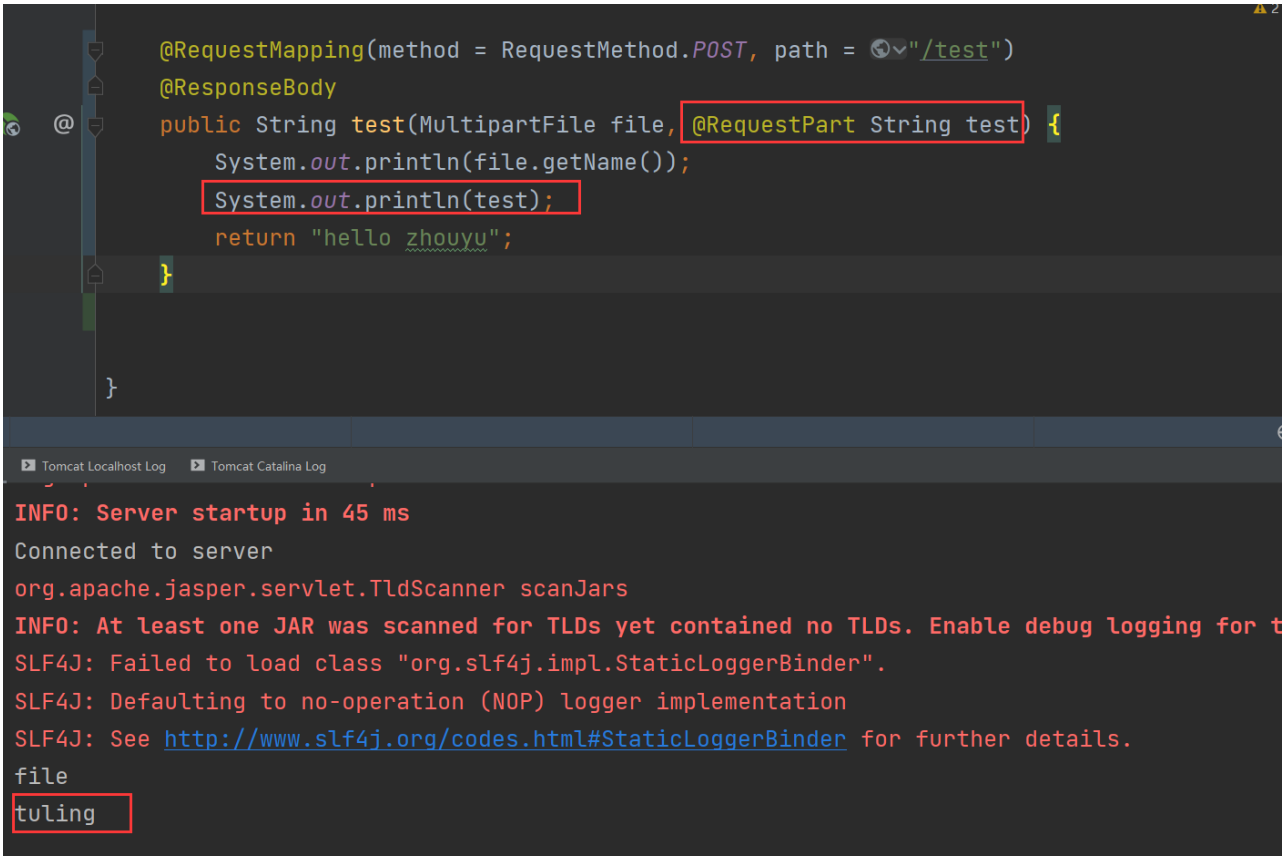


表达里面的test=tuling，请求parameter中的test=zhouyu，那最终test等于哪个呢？

答案是两个：



那如果我只想获取表达里的test呢？可以用@RequestPart注解：



当接收到一个请求后：

1. SpringMVC利用MultipartResolver来判断当前请求是不是一个multipart/form-data请求
2. 如果是会把这个请求封装为StandardMultipartHttpServletRequest对象
3. 并且获取请求中所有的Part，并且遍历每个Part
4. 判断Part是文件还是文本
5. 如果是文件，会把Part封装为一个StandardMultipartFile对象（实现了MultipartFile接口），并且会把StandardMultipartFile对象添加到multipartFiles中
6. 如果是文本，会把Part的名字添加到multipartParameterNames中
7. 然后在解析某个参数时
8. 如果参数类型是MultipartFile，会根据参数名字从multipartFiles中获取出StandardMultipartFile对象，最终把这个对象传给方法

方法返回值解析

在RequestMappingHandlerAdapter的初始化逻辑中会设置一些默认的HandlerMethodReturnValueHandler，他们就是用来解析各种类型的方法返回值的。

比如：

1. ModelAndViewMethodReturnValueHandler，处理的就是返回值类为ModelAndView的情况
2. RequestResponseBodyMethodProcessor，处理的就是方法上或类上加了@ResponseBody的情况

3. ViewNameMethodReturnValueHandler，处理的就是返回值为字符串的请求（无@ResponseBody）

我们重点看RequestResponseBodyMethodProcessor。

假如代码如下：

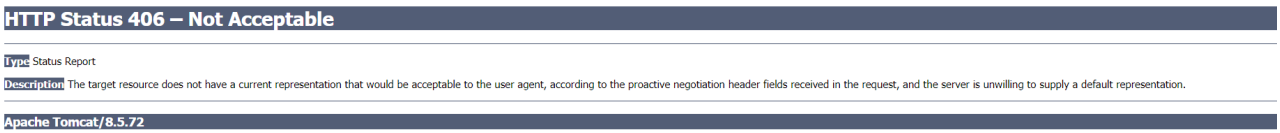
```
@Controller
public class ZhouyuController {

        @RequestMapping(method = RequestMethod.GET, path = "/test")
        @ResponseBody
        public User test() {
                User user = new User();
        user.setName("zhouyu");
                return user;
        }

}
```

方法返回的是User对象，那么怎么把这个User对象返回给浏览器来展示呢？那得看当前请求设置的Accept请求头，比如我用Chrome浏览器发送请求，默认给我设置的就是：Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9

表示当前这个请求接收的内容格式，比如html格式、xml格式、各种图片格式等等。

如果我们的方法返回的是一个字符串，那么就对应html格式，就没问题，而如果我们不是返回的字符串，那我们就转成字符串，通常就是JSON格式的字符串。

所以，我们需要将User对象转换成JSON字符串，默认SpringMVC是不能转换的，此时请求会报错：



而要完成这件事情，我们需要添加一个MappingJackson2HttpMessageConverter，通过它就能把User对象或者Map对象等转成一个JSON字符串。

XML的添加方式：

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
  </mvc:message-converters>
</mvc:annotation-driven>
```

记得要引入Jackson2的依赖：

```
<!--
https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind
<!--
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.2</version>
</dependency>

```

我们看一下MappingJackson2HttpMessageConverter的构造方法：

```
public MappingJackson2HttpMessageConverter() {
    this(Jackson2ObjectMapperBuilder.json().build());
}


public MappingJackson2HttpMessageConverter(ObjectMapper objectMapper) {
    super(objectMapper, MediaType.APPLICATION_JSON, new MediaType("application", "*+json"));
}
```

表示MappingJackson2HttpMessageConverter支持的MediaType
为"application/json"、"application/*+json"。

所以如果我们明确指定方法返回的MediaType为"text/plain"，那么
MappingJackson2HttpMessageConverter就不能处理了，比如：

```
@RequestMapping(method = RequestMethod.GET, path = "/test", produces = "text/plain")
@ResponseBody
public User test() {
    User user = new User();
    user.setName("zhouyu");
    return user;
}
```

以上代码表示，需要把一个User对象转成一个纯文本字符串，默认是没有这种转换器的。

一个HttpMessageConverter中有一个canWrite()方法，表示这个HttpMessageConverter能把什么类型转成什么MediaType返回给浏览器。

比如SpringMVC自带一个StringHttpMessageConverter，它能够把一个String对象返回给浏览器，支持所有的MediaType。

那为了支持把User对象转成纯文本，我们可以自定义ZhouyuHttpMessageConverter：

```
/**
 * 作者：周瑜大都督
 */
public class ZhouyuHttpMessageConverter extends AbstractHttpMessageConverter<User> {

        @Override
        public List<MediaType> getSupportedMediaTypes() {
                ArrayList<MediaType> mediaTypes = new ArrayList<>();
                mediaTypes.add(MediaType.ALL);
                return mediaTypes;
        }

        @Override
        protected boolean supports(Class clazz) {
                return User.class == clazz;
        }

        @Override
        protected User readInternal(Class<? extends User> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException {
                return null;
        }

        @Override
        protected void writeInternal(User user, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException {
                StreamUtils.copy(user.getName(), Charset.defaultCharset(), outputMessage.getBody());
        }
}
```

我定义的这个HttpMessageConverter就能够把User对象转成纯文本。

拦截器解析

我们可以使用HandlerInterceptor来拦截请求：

```
package org.springframework.web.servlet;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.lang.Nullable;
import org.springframework.web.method.HandlerMethod;

public interface HandlerInterceptor {

        default boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
                        throws Exception {

                return true;
        }


        default void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler,
                        @Nullable ModelAndView modelAndView) throws Exception {
        }


        default void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler,
                        @Nullable Exception ex) throws Exception {
        }

}

```

具体执行顺序看下图：

https://www.processon.com/view/link/63e9f3e6234df52a1e9303fb

@EnableWebMvc解析

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
}
```

导入了一个DelegatingWebMvcConfiguration配置类，这个配置类定义了很多个Bean，比如RequestMappingHandlerMapping，后续在创建RequestMappingHandlerMapping这个Bean对象时，会调用DelegatingWebMvcConfiguration的getInterceptors()方法来获取拦截器：

```
@Bean
@SuppressWarnings("deprecation")
public RequestMappingHandlerMapping requestMappingHandlerMapping(...) {

    RequestMappingHandlerMapping mapping = createRequestMappingHandlerMapping();
    mapping.setInterceptors(getInterceptors(conversionService, resourceUrlProvider));
    // ...
    return mapping;
}
```

而在getInterceptors()方法中会调用addInterceptors()方法，从而会调用WebMvcConfigurerComposite的addInterceptors()方法，然后会遍历调用WebMvcConfigurer的addInterceptors()方法来添加拦截器：

```
public void addInterceptors(InterceptorRegistry registry) {
    for (WebMvcConfigurer delegate : this.delegates) {
        delegate.addInterceptors(registry);
        }
}
```

那么delegates集合中的值是哪来的呢？在DelegatingWebMvcConfiguration中进行了一次set注入：

```
@Autowired(required = false)
public void setConfigurers(List<WebMvcConfigurer> configurers) {
    if (!CollectionUtils.isEmpty(configurers)) {
        this.configurers.addWebMvcConfigurers(configurers);
    }
}

public void addWebMvcConfigurers(List<WebMvcConfigurer> configurers) {
    if (!CollectionUtils.isEmpty(configurers)) {
        this.delegates.addAll(configurers);
    }
}
```

所以就是把Spring容器中的WebMvcConfigurer的Bean添加到了delegates集合中。

所以，我们可以配置WebMvcConfigurer类型的Bean，并通过addInterceptors()方法来给SpringMvc添加拦截器。

同理我们可以利用WebMvcConfigurer中的其他方法来对SpringMvc进行配置，比如

```
@ComponentScan("com.zhouyu")
@Configuration
@EnableWebMvc
public class AppConfig implements WebMvcConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer.addPathPrefix("/zhouyu", t ->  t.equals(ZhouyuController.class));
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {

    }
}
```

所以@EnableWebMvc的作用是提供了可以让程序员通过定义WebMvcConfigurer类型的Bean来对SpringMVC进行配置的功能。

另外值得注意的是，如果加了@EnableWebMvc注解，那么Spring容器中会有三个HandlerMapping类型的Bean：

1. RequestMappingHandlerMapping
2. BeanNameUrlHandlerMapping
3. RouterFunctionMapping

如果没有加@EnableWebMvc注解，那么Spring容器中默认也会有三个HandlerMapping类型的Bean：

1. BeanNameUrlHandlerMapping
2. RequestMappingHandlerMapping
3. RouterFunctionMapping

就顺序不一样而已，源码中是根据DispatcherServlet.properties文件来配置有哪些HandlerMapping的。

```
private void initHandlerMappings(ApplicationContext context) {
		this.handlerMappings = null;

		// 默认为true，获取HandlerMapping类型的Bean
		if (this.detectAllHandlerMappings) {
			// Find all HandlerMappings in the ApplicationContext, including ancestor contexts.
			Map<String, HandlerMapping> matchingBeans =
BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);
			if (!matchingBeans.isEmpty()) {
				this.handlerMappings = new ArrayList<>(matchingBeans.values());
				// We keep HandlerMappings in sorted order.
AnnotationAwareOrderComparator.sort(this.handlerMappings);
			}
		}
		// 获取名字叫handlerMapping的Bean
		else {
			try {
				HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMapping.class);
				this.handlerMappings = Collections.singletonList(hm);
			}
			catch (NoSuchBeanDefinitionException ex) {
				// Ignore, we'll add a default HandlerMapping later.
			}
		}

		// 如果从Spring容器中没有找到HandlerMapping类型的Bean
		// 就根据DispatcherServlet.properties配置来创建HandlerMapping类型的Bean
        // 默认就有这么一个文件，会创建出来三个HandlerMapping的Bean
		if (this.handlerMappings == null) {
			this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
			if (logger.isTraceEnabled()) {
				logger.trace("No HandlerMappings declared for servlet '" + getServletName() +
"': using default strategies from DispatcherServlet.properties");
```

```
                        }
                }

                for (HandlerMapping mapping : this.handlerMappings) {
                        if (mapping.usesPathPatterns()) {
                                this.parseRequestPath = true;
                                break;
                        }
                }
        }
```

由于加和不加@EnableWebMvc注解之后的HandlerMapping顺序不一样，可能会导致一些问题（工作中很难遇到）：

```
@Component("/test")
public class BeanNameUrlController implements Controller {
        @Override
        public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {
                System.out.println("BeanNameUrlController");
                return null;
        }
}
```

```java
@RestController
public class ZhouyuController {

        @GetMapping("/test")
        public String test() {
                System.out.println("ZhouyuController");
                return null;
        }

}
```

这两个Controller访问路径是一样的，但是负责处理的HandlerMapping是不一样的，

1. BeanNameUrlController对应的是BeanNameUrlHandlerMapping
2. ZhouyuController对应的是RequestMappingHandlerMapping

如果加了@EnableWebMvc注解，顺序为：

1. RequestMappingHandlerMapping
2. BeanNameUrlHandlerMapping
3. RouterFunctionMapping

会先由RequestMappingHandlerMapping处理/test请求，最终执行的是ZhouyuController中的test

如果没有加@EnableWebMvc注解，顺序为：

1. BeanNameUrlHandlerMapping
2. RequestMappingHandlerMapping
3. RouterFunctionMapping

会先由BeanNameUrlHandlerMapping处理/test请求，最终执行的是BeanNameUrlController中的test

注意，一个HandlerMapping处理完请求后就不会再让其他HandlerMapping来处理请求了。