



下载APP



11 | Spring Web Body 转化常见错误

2021-05-17 傅健

Spring编程常见错误50例

[进入课程 >](#)**讲述：傅健**

时长 14:01 大小 12.85M



你好，我是傅健。前面几节课我们学习了 Spring Web 开发中绕不开的 URL 和 Header 处理。这一节课，我们接着讲 Body 的处理。


实际上，在 Spring 中，对于 Body 的处理很多是借助第三方编解码器来完成的。例如常见的 JSON 解析，Spring 都是借助于 Jackson、Gson 等常见工具来完成。所以在 Body 处理中，我们遇到的很多错误都是第三方工具使用中的一些问题。

真正对于 Spring 而言，错误并不多，特别是 Spring Boot 的自动包装以及对常见问题不断完善，让我们能犯的错误已经很少了。不过，毕竟不是每个项目都是直接基于 Spring Boot 的，所以还是会存在一些问题，接下来我们就一起梳理下。




案例 1：No converter found for return value of type

在直接用 Spring MVC 而非 Spring Boot 来编写 Web 程序时，我们基本都会遇到 "No converter found for return value of type" 这种错误。实际上，我们编写的代码都非常简单，例如下面这段代码：

 复制代码

```
1 //定义的数据对象
2 @Data
3 @NoArgsConstructor
4 @AllArgsConstructor
5 public class Student {
6     private String name;
7     private Integer age;
8 }
9 //定义的 API 借口
10 @RestController
11 public class HelloController {
12
13     @GetMapping("/hi1")
14     public Student hi1() {
15         return new Student("xiaoming", Integer.valueOf(12));
16     }
17 }
```

然后，我们的 pom.xml 文件也都是最基本的必备项，关键配置如下：

 复制代码

```
1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-webmvc</artifactId>
4     <version>5.2.3.RELEASE</version>
5 </dependency>
```

但是当我们运行起程序，执行测试代码，就会报错如下：

实际上节课我们就贴出过相关代码并分析过，所以这里只是带着你简要分析下上述代码的基本逻辑：

1. 查看请求的头中是否有 ACCET 头，如果没有则可以使用任何类型；
2. 查看当前针对返回类型（即 Student 实例）可以采用的编码类型；
3. 取上面两步获取结果的交集来决定用什么方式返回。

比较代码，我们可以看出，假设第 2 步中就没有找到合适的编码方式，则直接报案例中的错误，具体的关键代码行如下：

[复制代码](#)

```
1 if (body != null && producibleTypes.isEmpty()) {
2     throw new HttpResponseMessageNotWritableException(
3         "No converter found for return value of type: " + valueType);
4 }
```

那么当前可采用的编码类型是怎么决策出来的呢？我们可以进一步查看方法 `AbstractMessageConverterMethodProcessor#getProducibleMediaTypes`：

[复制代码](#)

```
1 protected List<MediaType> getProducibleMediaTypes(
2     HttpServletRequest request, Class<?> valueClass, @Nullable Type targetType
3 )
4     Set<MediaType> mediaTypes =
5         (Set<MediaType>) request.getAttribute(HandlerMapping.PRODUCIBLE_MEDIA
6     if (!CollectionUtils.isEmpty(mediaTypes)) {
7         return new ArrayList<>(mediaTypes);
8     }
9     else if (!this.allSupportedMediaTypes.isEmpty()) {
10         List<MediaType> result = new ArrayList<>();
11         for (HttpMessageConverter<?> converter : this.messageConverters) {
12             if (converter instanceof GenericHttpMessageConverter && targetType !=
13                 if (((GenericHttpMessageConverter<?>) converter).canWrite(targetTy
14                     result.addAll(converter.getSupportedMediaTypes());
15             }
16         }
17         else if (converter.canWrite(valueClass, null)) {
18             result.addAll(converter.getSupportedMediaTypes());
19         }
20     }
21     return result;
```

```
22     }
23     else {
24         return Collections.singletonList(MediaType.ALL);
25     }
26 }
```

假设当前没有显式指定返回类型（例如给 `GetMapping` 指定 `produces` 属性），那么则会遍历所有已经注册的 `HttpMessageConverter` 查看是否支持当前类型，从而最终返回所有支持的类型。那么这些 `MessageConverter` 是怎么注册过来的？

在 Spring MVC（非 Spring Boot）启动后，我们都会构建 `RequestMappingHandlerAdapter` 类型的 Bean 来负责路由和处理请求。

具体而言，当我们使用 `<mvc:annotation-driven/>` 时，我们会通过 `AnnotationDrivenBeanDefinitionParser` 来构建这个 Bean。而在它的构建过程中，会决策出以后要使用哪些 `HttpMessageConverter`，相关代码参考 `AnnotationDrivenBeanDefinitionParser#getMessageConverters`：

[复制代码](#)

```
1 messageConverters.add(createConverterDefinition(ByteArrayHttpMessageConverter.
2 RootBeanDefinition stringConverterDef = createConverterDefinition(StringHttpMe
3 stringConverterDef.getPropertyValues().add("writeAcceptCharset", false);
4 messageConverters.add(stringConverterDef);
5 messageConverters.add(createConverterDefinition(ResourceHttpMessageConverter.c
6 //省略其他非关键代码
7 if (jackson2Present) {
8     Class<?> type = MappingJackson2HttpMessageConverter.class;
9     RootBeanDefinition jacksonConverterDef = createConverterDefinition(type, so
10     GenericBeanDefinition jacksonFactoryDef = createObjectMapperFactoryDefiniti
11     jacksonConverterDef.getConstructorArgumentValues().addIndexedArgumentValue(
12     messageConverters.add(jacksonConverterDef);
13 }
14 else if (gsonPresent) { messageConverters.add(createConverterDefinition(GsonHt
15 }
16 //省略其他非关键代码
```

这里我们会默认使用一些编解码器，例如 `StringHttpMessageConverter`，但是像 JSON、XML 等类型，若要加载编解码，则需要 `jackson2Present`、`gsonPresent` 等变量为 `true`。

这里我们可以选取 `gsonPresent` 看下何时为 `true`，参考下面的关键代码行：

```
gsonPresent = ClassUtils.isPresent("com.google.gson.Gson", classLoader);
```

假设我们依赖了 `Gson` 包，我们就可以添加上 `GsonHttpMessageConverter` 这种转换器。但是可惜的是，我们的案例并没有依赖上任何 `JSON` 的库，所以最终在候选的转换器列表里，并不存在 `JSON` 相关的转换器。最终候选列表示例如下：



```
▼ messageConverters = {ArrayList@4364} size = 8
  ▶ 0 = {ByteArrayHttpMessageConverter@4368}
  ▶ 1 = {StringHttpMessageConverter@4369}
  ▶ 2 = {ResourceHttpMessageConverter@4370}
  ▶ 3 = {ResourceRegionHttpMessageConverter@4371}
  ▶ 4 = {SourceHttpMessageConverter@4372}
  ▶ 5 = {AllEncompassingFormHttpMessageConverter@4373}
  ▶ 6 = {Jaxb2RootElementHttpMessageConverter@4374}
```

由此可见，并没有任何 `JSON` 相关的编解码器。而针对 `Student` 类型的返回对象，上面的这些编解码器又不符合要求，所以最终走入了下面的代码行：

```
1 if (body != null && producibleTypes.isEmpty()) {
2     throw new HttpMessageNotWritableException(
3         "No converter found for return value of type: " + valueType);
4 }
```

[复制代码](#)

抛出了 "No converter found for return value of type" 这种错误，结果符合案例中的实际测试情况。

问题修正

针对这个案例，有了源码的剖析，可以看出，**不是每种类型的编码器都会与生俱来，而是根据当前项目的依赖情况决定是否支持**。要解析 `JSON`，我们就要依赖相关的包，所以这里我们可以以 `Gson` 为例修正下这个问题：

```
1 <dependency>
2   <groupId>com.google.code.gson</groupId>
3   <artifactId>gson</artifactId>
4   <version>2.8.6</version>
5 </dependency>
```

[复制代码](#)

我们添加了 Gson 的依赖到 pom.xml。重新运行程序和测试案例，你会发现不再报错了。

另外，这里我们还可以查看下 GsonHttpMessageConverter 这种编码器是如何支持上 Student 这个对象的解析的。

通过这个案例，我们可以知道，Spring 给我们提供了很多好用的功能，但是这些功能交织到一起后，我们就很可能入坑，只有深入了解它的运行方式，才能迅速定位问题并解决问题。

案例 2：变动地返回 Body

案例 1 让我们解决了解析问题，那随着不断实践，我们可能还会发现在代码并未改动的情況下，返回结果不再和之前相同了。例如我们看下这段代码：

```
1 @RestController
2 public class HelloController {
3
4     @PostMapping("/hi2")
5     public Student hi2(@RequestBody Student student) {
6         return student;
7     }
8
9 }
```

[复制代码](#)

上述代码接受了一个 Student 对象，然后原样返回。我们使用下面的测试请求进行测试：

```
POST http://localhost:8080/springmvc3_war/app/hi2
Content-Type: application/json
{
  "name": "xiaoming"
}
```

经过测试，我们会得到以下结果：

```
{
  "name": "xiaoming"
}
```

但是随着项目的推进，在代码并未改变时，我们可能会返回以下结果：


```
{
  "name": "xiaoming",
  "age": null
}
```

即当 age 取不到值，开始并没有序列化它作为响应 Body 的一部分，后来又序列化成 null 作为 Body 返回了。

在什么情况下会如此？如何规避这个问题，保证我们的返回始终如一。


案例解析

如果我们发现上述问题，那么很有可能是这样一种情况造成的。即在后续的代码开发中，我们直接依赖或者间接依赖了新的 JSON 解析器，例如下面这种方式就依赖了 Jackson：

 复制代码

```
1 <dependency>
2     <groupId>com.fasterxml.jackson.core</groupId>
3     <artifactId>jackson-databind</artifactId>
4     <version>2.9.6</version>
5 </dependency>
```

当存在多个 Jackson 解析器时，我们的 Spring MVC 会使用哪一种呢？这个决定可以参考

 复制代码

```
1 if (jackson2Present) {
2     Class<?> type = MappingJackson2HttpMessageConverter.class;
3     RootBeanDefinition jacksonConverterDef = createConverterDefinition(type, so
```




```
4     GenericBeanDefinition jacksonFactoryDef = createObjectMapperFactoryDefinitio
5     jacksonConverterDef.getConstructorArgumentValues().addIndexedArgumentValue(
6     messageConverters.add(jacksonConverterDef);
7 }
8 else if (gsonPresent) {
9     messageConverters.add(createConverterDefinition(GsonHttpMessageConverter.cl
10 }
```

从上述代码可以看出，Jackson 是优先于 Gson 的。所以我们的程序不知不觉已经从 Gson 编解码切换成了 Jackson。所以此时，**行为就不见得和之前完全一致了。**

针对本案例中序列化值为 null 的字段的行为而言，我们可以分别看下它们的行为是否一致。

1. 对于 Gson 而言：

GsonHttpMessageConverter 默认使用 new Gson() 来构建 Gson，它的构造器中指明了相关配置：

 复制代码

```
1 public Gson() {
2     this(Excluder.DEFAULT, FieldNamingPolicy.IDENTITY,
3         Collections.<Type, InstanceCreator<?>>emptyMap(), DEFAULT_SERIALIZE_NULL
4         DEFAULT_COMPLEX_MAP_KEYS, DEFAULT_JSON_NON_EXECUTABLE, DEFAULT_ESCAPE_HT
5         DEFAULT_PRETTY_PRINT, DEFAULT_LENIENT, DEFAULT_SPECIALIZE_FLOAT_VALUES,
6         LongSerializationPolicy.DEFAULT, null, DateFormat.DEFAULT, DateFormat.DE
7         Collections.<TypeAdapterFactory>emptyList(), Collections.<TypeAdapterFac
8         Collections.<TypeAdapterFactory>emptyList());
9 }
```

从 DEFAULT_SERIALIZE_NULLS 可以看出，它是默认不序列化 null 的。

2. 对于 Jackson 而言：

MappingJackson2HttpMessageConverter 使用 "Jackson2ObjectMapperBuilder.json().build()" 来构建 ObjectMapper，它默认只显式指定了下面两个配置：

MapperFeature.DEFAULT_VIEW_INCLUSION

DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES

Jackson 默认对于 null 的处理是做序列化的，所以本案例中 age 为 null 时，仍然被序列化了。

通过上面两种 JSON 序列化的分析可以看出，**返回的内容在依赖项改变的情况下确实可能发生变化。**

问题修正

那么针对这个问题，如何修正呢？即保持在 Jackson 依赖项添加的情况下，让它和 Gson 的序列化行为一致吗？这里可以按照以下方式进行修改：

```
1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  @JsonInclude(JsonInclude.Include.NON_NULL)
5  public class Student {
6      private String name;
7      //或直接加在 age 上: @JsonInclude(JsonInclude.Include.NON_NULL)
8      private Integer age;
9  }
10
```

 复制代码


我们可以直接使用 @JsonInclude 这个注解，让 Jackson 和 Gson 的默认行为对于 null 的处理变成一致。

上述修改方案虽然看起来简单，但是假设有很多对象如此，万一遗漏了怎么办呢？所以可以从全局角度来修改，修改的关键代码如下：

```
//ObjectMapper mapper = new ObjectMapper();
mapper.setSerializationInclusion(Include.NON_NULL);
```

但是如何修改 ObjectMapper 呢？这个对象是由 MappingJackson2HttpMessageConverter 构建的，看似无法插足去修改。实际上，我

们在非 Spring Boot 程序中，可以按照下面这种方式来修改：

 复制代码


```
1 @RestController
2 public class HelloController {
3
4     public HelloController(RequestMappingHandlerAdapter requestMappingHandlerAdapt
5         List<HttpMessageConverter<?>> messageConverters =
6             requestMappingHandlerAdapter.getMessageConverters();
7         for (HttpMessageConverter<?> messageConverter : messageConverters) {
8             if(messageConverter instanceof MappingJackson2HttpMessageConverter ){
9                 (((MappingJackson2HttpMessageConverter)messageConverter).getObject
10             }
11         }
12     }
13     //省略其他非关键代码
14 }
```

我们用自动注入的方式获取到 RequestMappingHandlerAdapter，然后找到 Jackson 解析器，进行配置即可。

通过上述两种修改方案，我们就能做到忽略 null 的 age 字段了。

案例 3：Required request body is missing

通过案例 1，我们已经能够解析 Body 了，但是有时候，我们会有一些很好的想法。例如为了查询问题方便，在请求过来时，自定义一个 Filter 来统一输出具体的请求内容，关键代码如下：

 复制代码

```
1 public class ReadBodyFilter implements Filter {
2
3     //省略其他非关键代码
4     @Override
5     public void doFilter(ServletRequest request,
6         ServletResponse response, FilterChain chain)
7         throws IOException, ServletException {
8         String requestBody = IOUtils.toString(request.getInputStream(), "utf-8
9         System.out.println("print request body in filter:" + requestBody);
10        chain.doFilter(request, response);
11    }
12
13 }
```

然后，我们可以把这个 Filter 添加到 web.xml 并配置如下：

[复制代码](#)

```
1 <filter>
2   <filter-name>myFilter</filter-name>
3   <filter-class>com.puzzles.ReadBodyFilter</filter-class>
4 </filter>
5 <filter-mapping>
6   <filter-name>myFilter</filter-name>
7   <url-pattern>/app/*</url-pattern>
8 </filter-mapping>
```

再测试下 Controller 层中定义的接口：

[复制代码](#)

```
1 @PostMapping("/hi3")
2 public Student hi3(@RequestBody Student student) {
3     return student;
4 }
```

运行测试，我们会发现下面的日志：


```
print request body in filter:{
"name": "xiaoming",
"age": 10
}
25-Mar-2021 11:04:44.906 璀ㄿㄿ [http-nio-8080-exec-5]
org.springframework.web.servlet.handler.AbstractHandlerExceptionResolver.logE
xception Resolved
[org.springframework.http.converter.HttpMessageNotReadableException:
Required request body is missing: public com.puzzles.Student
com.puzzles.HelloController.hi3(com.puzzles.Student)]
```

可以看到，请求的 Body 确实在请求中输出了，但是后续的操作直接报错了，错误提示：Required request body is missing。

案例解析

要了解这个错误的根本原因，你得知道这个错误抛出的源头。查阅请求 Body 转化的相关代码，有这样一段关键逻辑（参考

RequestMappingHandlerMethodProcessor#readWithMessageConverters）：

 复制代码

```
1  protected <T> Object readWithMessageConverters(NativeWebRequest webRequest, Me
2      Type paramType) throws IOException, HttpMediaTypeNotSupportedException,
3      HttpServletRequest servletRequest = webRequest.getNativeRequest(HttpServletRequest
4      ServletServerHttpRequest inputMessage = new ServletServerHttpRequest(servle
5      //读取 Body 并进行转化
6      Object arg = readWithMessageConverters(inputMessage, parameter, paramType);
7      if (arg == null && checkRequired(parameter)) {
8          throw new HttpMessageNotReadableException("Required request body is miss
9              parameter.getExecutable().toGenericString(), inputMessage);
10     }
11     return arg;
12 }
13 protected boolean checkRequired(MethodParameter parameter) {
14     RequestBody requestBody = parameter.getParameterAnnotation(RequestBody.class)
15     return (requestBody != null && requestBody.required() && !parameter.isOptio
16 }
```

当使用了 @RequestBody 且是必须时，如果解析出的 Body 为 null，则报错提示 Required request body is missing。

所以我们要继续追踪代码，来查询什么情况下会返回 body 为 null。关键代码参考 AbstractMessageConverterMethodArgumentResolver#readWithMessageConverters：

 复制代码

```
1  protected <T> Object readWithMessageConverters(HttpInputMessage inputMessage,
2      Type targetType){
3      //省略非关键代码
4      Object body = NO_VALUE;
5      EmptyBodyCheckingHttpInputMessage message;
6      try {
7          message = new EmptyBodyCheckingHttpInputMessage(inputMessage);
8          for (HttpMessageConverter<?> converter : this.messageConverters) {
9              Class<HttpMessageConverter<?>> converterType = (Class<HttpMessageConv
10                  GenericHttpMessageConverter<?> genericConverter =
11                      (converter instanceof GenericHttpMessageConverter ? (GenericHttp
```

```
12         if (genericConverter != null ? genericConverter.canRead(targetType, c
13             (targetClass != null && converter.canRead(targetClass, contentT
14         if (message.hasBody()) {
15             //省略非关键代码：读取并转化 body
16         else {
17             //处理没有 body 情况，默认返回 null
18             body = getAdvice().handleEmptyBody(null, message, parameter, ta
19         }
20         break;
21     }
22 }
23 }
24 catch (IOException ex) {
25     throw new HttpMessageNotReadableException("I/O error while reading input
26 }
27 //省略非关键代码
28 return body;
29 }
```

当 message 没有 body 时（message.hasBody() 为 false），则将 body 认为是 null。继续查看 message 本身的定义，它是一种包装了请求 Header 和 Body 流的 EmptyBodyCheckingHttpInputMessage 类型。其代码实现如下：

[复制代码](#)

```
1 public EmptyBodyCheckingHttpInputMessage(HttpInputMessage inputMessage) throws
2     this.headers = inputMessage.getHeaders();
3     InputStream inputStream = inputMessage.getBody();
4     if (inputStream.markSupported()) {
5         //省略其他非关键代码
6     }
7     else {
8         PushbackInputStream pushbackInputStream = new PushbackInputStream(inputS
9         int b = pushbackInputStream.read();
10        if (b == -1) {
11            this.body = null;
12        }
13        else {
14            this.body = pushbackInputStream;
15            pushbackInputStream.unread(b);
16        }
17    }
18 }
19 public InputStream getBody() {
20     return (this.body != null ? this.body : StreamUtils.emptyInput());
21 }
22 }
```


Body 为空的判断是由 `pushbackInputStream.read()` 其值为 -1 来判断出的，即没有数据可以读取。

看到这里，你可能会疑问：假设有 Body，`read()` 的执行不就把数据读取走了一点么？确实如此，所以这里我使用了 `pushbackInputStream.unread(b)` 调用来把读取出来的数据归还回去，这样就完成了是否有 Body 的判断，又保证了 Body 的完整性。

分析到这里，再结合前面的案例，你应该能想到造成 Body 缺失的原因了吧？

1. 本身就没有 Body；
2. 有 Body，但是 Body 本身代表的流已经被前面读取过了。


很明显，我们的案例属于第 2 种情况，即在过滤器中，我们就已经将 Body 读取完了，关键代码如下：

```
//request 是 ServletRequest  
String requestBody = IOUtils.toString(request.getInputStream(), "utf-8");
```

在这种情况下，作为一个普通的流，已经没有数据可以供给后面的转化器来读取了。

问题修正

所以我们可以直接在过滤器中去掉 Body 读取的代码，这样后续操作就又能读到数据了。但是这样又不满足我们的需求，如果我们坚持如此怎么办呢？这里我先直接给出答案，即定义一个 `RequestBodyAdviceAdapter` 的 Bean：

 复制代码

```
1 @ControllerAdvice  
2 public class PrintRequestBodyAdviceAdapter extends RequestBodyAdviceAdapter {  
3     @Override  
4     public boolean supports(MethodParameter methodParameter, Type type, Class<  
5         return true;  
6     }  
7     @Override  
8     public Object afterBodyRead(Object body, HttpInputMessage inputMessage, Met  
9         Class<? extends HttpMessageConverter<?>> converterType) {  
10         System.out.println("print request body in advice:" + body);  
11         return super.afterBodyRead(body, inputMessage, parameter, targetType,  
12
```

```
13     }  
    ,
```

我们可以看到方法 `afterBodyRead` 的命名，很明显，这里的 `Body` 已经是从小数据流中转化过的。

那么它是如何工作起来的呢？我们可以查看下面的代码（参考 `AbstractMessageConverterMethodArgumentResolver#readWithMessageConverters`）：

[复制代码](#)

```
1  protected <T> Object readWithMessageConverters(HttpInputMessage inputMessage,  
2      //省略其他非关键代码  
3      if (message.hasBody()) {  
4          HttpInputMessage msgToUse = getAdvice().beforeBodyRead(message, param  
5          body = (genericConverter != null ? genericConverter.read(targetType, conte  
6          body = getAdvice().afterBodyRead(body, msgToUse, parameter, targetType, co  
7          //省略其他非关键代码  
8      }  
9      //省略其他非关键代码  
10     return body;  
11 }
```

当一个 `Body` 被解析出来后，会调用 `getAdvice()` 来获取 `RequestResponseBodyAdviceChain`；然后在这个 `Chain` 中，寻找合适的 `Advice` 并执行。

正好我们前面定义了 `PrintRequestBodyAdviceAdapter`，所以它的相关方法就被执行了。从执行时机来看，此时 `Body` 已经解析完毕了，也就是说，传递给 `PrintRequestBodyAdviceAdapter` 的 `Body` 对象已经是一个解析过的对象，而不再是一个流了。

通过上面的 `Advice` 方案，我们满足了类似的需求，又保证了程序的正确执行。至于其他的一些方案，你可以来思考一下。

重点回顾

通过这节课的学习，相信你对 Spring Web 中关于 Body 解析的常见错误已经有所了解了，这里我们再次回顾下关键知识点：

1. 不同的 Body 需要不同的编解码器，而使用哪一种是协商出来的，协商过程大体如下：

查看请求头中是否有 ACCET 头，如果没有则可以使用任何类型；

查看当前针对返回类型（即 Student 实例）可以采用的编码类型；

取上面两步获取的结果的交集来决定用什么方式返回。

2. 在非 Spring Boot 程序中，JSON 等编解码器不见得是内置好的，需要添加相关的 JAR 才能自动依赖上，而自动依赖的实现是通过检查 Class 是否存在来实现的：当依赖上相关的 JAR 后，关键的 Class 就存在了，响应的编解码器功能也就提供上了。

3. 不同的编解码器的实现（例如 JSON 工具 Jaskson 和 Gson）可能有一些细节上的不同，所以你一定要注意当依赖一个新的 JAR 时，是否会引起默认编解码器的改变，从而影响到一些局部行为的改变。

4. 在尝试读取 HTTP Body 时，你要注意到 Body 本身是一个流对象，不能被多次读取。

以上即为这节课的主要内容，希望能对你有所帮助。

思考题

通过案例 1 的学习，我们知道直接基于 Spring MVC 而非 Spring Boot 时，是需要我们手工添加 JSON 依赖，才能解析出 JSON 的请求或者编码 JSON 响应，那么为什么基于 Spring Boot 就不需要这样做了呢？

期待你的思考，我们留言区见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | Spring Web Header 解析常见错误

下一篇 12 | Spring Web 参数验证常见错误

精选留言 (2)

写留言



正在研读Spring50

2021-05-18

源码果然还是不那么容易啃,看着看着就走神了;铁子们有啥好办法吗

展开



哦吼掉了

2021-05-17

思考题：springboot自动装配了WebMvcAutoConfiguration

展开

