



下载APP



09 | Spring Web URL 解析常见错误

2021-05-12 傅健

Spring编程常见错误50例

[进入课程 >](#)**讲述：傅健**

时长 21:43 大小 19.90M



你好，我是傅健。

上一章节我们讲解了各式各样的错误案例，这些案例都是围绕 Spring 的核心功能展开的，例如依赖注入、AOP 等诸多方面。然而，从现实情况来看，在使用上，我们更多地是使用 Spring 来构建一个 Web 服务，所以从这节课开始，我们会重点解析在 Spring Web 开发中经常遇到的一些错误，帮助你规避这些问题。

不言而喻，这里说的 Web 服务就是指使用 HTTP 协议的服务。而对于 HTTP 请求，首先要处理的就是 URL，所以今天我们就先来介绍下，在 URL 的处理上，Spring 都有哪些经典的案例。闲话少叙，下面我们直接开始演示吧。



案例 1：当 @PathVariable 遇到 /

在解析一个 URL 时，我们经常会使用 `@PathVariable` 这个注解。例如我们会经常见到如下风格的代码：

[复制代码](#)

```
1 @RestController
2 @Slf4j
3 public class HelloWorldController {
4     @RequestMapping(path = "/hi1/{name}", method = RequestMethod.GET)
5     public String hello1(@PathVariable("name") String name){
6         return name;
7     }
8 };
9 }
```

当我们使用 <http://localhost:8080/hi1/xiaoming> 访问这个服务时，会返回"xiaoming"，即 Spring 会把 name 设置为 URL 中对应的值。

看起来顺风顺水，但是假设这个 name 中含有特殊字符 / 时（例如 <http://localhost:8080/hi1/xiao/ming>），会如何？如果我们不假思索，或许答案是"xiao/ming"？然而稍微敏锐点的程序员都会判定这个访问是会报错的，具体错误参考：

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Mar 13 11:15:57 CST 2021

There was an unexpected error (type=Not Found, status=404)

No message available

如图所示，当 name 中含有 /，这个接口不会为 name 获取任何值，而是直接报 Not Found 错误。当然这里的“找不到”并不是指 name 找不到，而是指服务于这个特殊请求的接口。


实际上，这里还存在另外一种错误，即当 name 的字符串以 / 结尾时，/ 会被自动去掉。例如我们访问 <http://localhost:8080/hi1/xiaoming/>，Spring 并不会报错，而是返回 xiaoming。

针对这两种类型的错误，应该如何理解并修正呢？

案例解析

实际上，这两种错误都是 URL 匹配执行方法的相关问题，所以我们有必要先了解下 URL 匹配执行方法的大致过程。参考

`AbstractHandlerMethodMapping#lookupHandlerMethod`:

 复制代码

```
1  @Nullable
2  protected HandlerMethod lookupHandlerMethod(String lookupPath, HttpServletRequest request) {
3      List<Match> matches = new ArrayList<>();
4      //尝试按照 URL 进行精准匹配
5      List<T> directPathMatches = this.mappingRegistry.getMappingsByUrl(lookupPath);
6      if (directPathMatches != null) {
7          //精确匹配上，存储匹配结果
8          addMatchingMappings(directPathMatches, matches, request);
9      }
10     if (matches.isEmpty()) {
11         //没有精确匹配上，尝试根据请求来匹配
12         addMatchingMappings(this.mappingRegistry.getMappings().keySet(), matches);
13     }
14
15     if (!matches.isEmpty()) {
16         Comparator<Match> comparator = new MatchComparator(getMappingComparator());
17         matches.sort(comparator);
18         Match bestMatch = matches.get(0);
19         if (matches.size() > 1) {
20             //处理多个匹配的情况
21         }
22         //省略其他非关键代码
23         return bestMatch.handlerMethod;
24     }
25     else {
26         //匹配不上，直接报错
27         return handleNoMatch(this.mappingRegistry.getMappings().keySet(), lookupPath);
28     }
29 }
```

大体分为这样几个基本步骤。

1. 根据 Path 进行精确匹配

这个步骤执行的代码语句是"`this.mappingRegistry.getMappingsByUrl(lookupPath)`", 实际上, 它是查询 `MappingRegistry#urlLookup`, 它的值可以用调试视图查看, 如下图所示:

```
▼ ∞ this.urlLookup = {LinkedMultiValueMap@7857} size = 7
  ▶ 0 = {"/hi5" -> {LinkedList@7872} size = 1}
  ▶ 1 = {"/hi6" -> {LinkedList@7874} size = 1}
  ▶ 2 = {"/hi3" -> {LinkedList@7876} size = 1}
  ▶ 3 = {"/hi41" -> {LinkedList@7878} size = 1}
  ▶ 4 = {"/hi4" -> {LinkedList@7880} size = 1}
  ▶ 5 = {"/hi2" -> {LinkedList@7882} size = 1}
  ▶ 6 = {"/error" -> {LinkedList@7884} size = 2}
```

查询 `urlLookup` 是一个精确匹配 `Path` 的过程。很明显, <http://localhost:8080/hi1/xiao/ming> 的 `lookupPath` 是 `"/hi1/xiao/ming"`, 并不能得到任何精确匹配。这里需要补充的是, `"/hi1/{name}"` 这种定义本身也没有出现在 `urlLookup` 中。

2. 假设 Path 没有精确匹配上, 则执行模糊匹配

在步骤 1 匹配失败时, 会根据请求来尝试模糊匹配, 待匹配的匹配方法可参考下图:

```
Expression:
this.mappingRegistry.getMappings().keySet()

Result:
▼ ∞ result = {LinkedHashMap$LinkedKeySet@7966} size = 9
  ▶ 0 = {RequestMappingInfo@7896} "{GET /hi5}"
  ▶ 1 = {RequestMappingInfo@7898} "{GET /hi6}"
  ▶ 2 = {RequestMappingInfo@7900} "{GET /hi3}"
  ▶ 3 = {RequestMappingInfo@7902} "{GET /hi41}"
  ▶ 4 = {RequestMappingInfo@7904} "{GET /hi1/{name}}"  
  ▶ 5 = {RequestMappingInfo@7906} "{GET /hi4}"
  ▶ 6 = {RequestMappingInfo@7908} "{GET /hi2}"
  ▶ 7 = {RequestMappingInfo@7910} "{ /error}"
  ▶ 8 = {RequestMappingInfo@7912} "{ /error, produces [text/html]}"
```

显然, `"/hi1/{name}"` 这个匹配方法已经出现在待匹配候选中了。具体匹配过程可以参考方法 `RequestMappingInfo#getMatchingCondition`:

[复制代码](#)

```
1 public RequestMappingInfo getMatchingCondition(HttpServletRequest request) {
2     RequestMethodsRequestCondition methods = this.methodsCondition.getMatchingC
3     if (methods == null) {
4         return null;
5     }
6     ParamsRequestCondition params = this.paramsCondition.getMatchingCondition(r
7     if (params == null) {
8         return null;
9     }
10    //省略其他匹配条件
11    PatternsRequestCondition patterns = this.patternsCondition.getMatchingCondi
12    if (patterns == null) {
13        return null;
14    }
15    //省略其他匹配条件
16    return new RequestMappingInfo(this.name, patterns,
17        methods, params, headers, consumes, produces, custom.getCondition());
18 }
```

现在我们知道**匹配会查询所有的信息**, 例如 Header、Body 类型以及 URL 等。如果有一项不符合条件, 则不匹配。

在我们的案例中, 当使用 <http://localhost:8080/hi1/xiaoming> 访问时, 其中 `patternsCondition` 是可以匹配上的。实际的匹配方法执行是通过 `AntPathMatcher#match` 来执行, 判断的相关参数可参考以下调试视图:

```
if (this.pathMatcher.match(pattern, lookupPath)) { pathMatcher: AntPathMatcher@7817 lookupPath: "/hi1/xiaoming"
    return pattern; pattern: "/hi1/{name}"
}
```

但是当我们使用 <http://localhost:8080/hi1/xiao/ming> 来访问时, `AntPathMatcher` 执行的结果是 `"/hi1/xiao/ming"` 匹配不上 `"/hi1/{name}"`。

3. 根据匹配情况返回结果

如果找到匹配的方法, 则返回方法; 如果没有, 则返回 `null`。

在本案例中，<http://localhost:8080/hi1/xiao/ming> 因为找不到匹配方法最终报 404 错误。追根溯源就是 AntPathMatcher 匹配不了 `/hi1/xiao/ming` 和 `/hi1/{name}`。

另外，我们再回头思考 <http://localhost:8080/hi1/xiaoming/> 为什么没有报错而是直接去掉了 `/`。这里我直接贴出了负责执行 AntPathMatcher 匹配的 `PatternsRequestCondition#getMatchingPattern` 方法的部分关键代码：

[复制代码](#)

```
1 private String getMatchingPattern(String pattern, String lookupPath) {
2     //省略其他非关键代码
3     if (this.pathMatcher.match(pattern, lookupPath)) {
4         return pattern;
5     }
6     //尝试加一个/来匹配
7     if (this.useTrailingSlashMatch) {
8         if (!pattern.endsWith("/") && this.pathMatcher.match(pattern + "/", look
9             return pattern + "/";
10    }
11 }
12 return null;
13 }
```

在这段代码中，AntPathMatcher 匹配不了 `/hi1/xiaoming/` 和 `/hi1/{name}`，所以不会直接返回。进而，在 `useTrailingSlashMatch` 这个参数启用时（默认启用），会把 Pattern 结尾加上 `/` 再尝试匹配一次。如果能匹配上，在最终返回 Pattern 时就隐式自动加 `/`。

很明显，我们的案例符合这种情况，等于说我们最终是用了 `/hi1/{name}/` 这个 Pattern，而不再是 `/hi1/{name}`。所以自然 URL 解析 name 结果是去掉 `/` 的。

问题修正

针对这个案例，有了源码的剖析，我们可能会想到可以先用 `/**` 匹配上路径，等进入方法后再尝试去解析，这样就可以万无一失吧。具体修改代码如下：

[复制代码](#)

```
1 @RequestMapping(path = "/hi1/**", method = RequestMethod.GET)
2 public String hi1(HttpServletRequest request){
3     String requestURI = request.getRequestURI();
```



```
4     return requestURI.split("/hi1/")[1];  
5 }
```

但是这种修改方法还是存在漏洞，假设我们路径的 `name` 中刚好又含有 `"/hi1/"`，则 `split` 后返回的值就并不是我们想要的。实际上，更合适的修订代码示例如下：

[复制代码](#)

```
1 private AntPathMatcher antPathMatcher = new AntPathMatcher();  
2  
3 @RequestMapping(path = "/hi1/**", method = RequestMethod.GET)  
4 public String hi1(HttpServletRequest request){  
5     String path = (String) request.getAttribute(HandlerMapping.PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE);  
6     //matchPattern 即为"/hi1/**"  
7     String matchPattern = (String) request.getAttribute(HandlerMapping.BEST_MATCHING_PATTERN_ATTRIBUTE);  
8     return antPathMatcher.extractPathWithinPattern(matchPattern, path);  
9 };
```

经过修改，两个错误都得以解决了。当然也存在一些其他的方案，例如对传递的参数进行 URL 编码以避免出现 `/`，或者干脆直接把这个变量作为请求参数、Header 等，而不是作为 URL 的一部分。你完全可以根据具体情况来选择合适的方案。

案例 2：错误使用 `@RequestParam`、`@PathVariable` 等注解


我们常常使用 `@RequestParam` 和 `@PathVariable` 来获取请求参数（request parameters）以及 `path` 中的部分。但是在频繁使用这些参数时，不知道你有没有觉得它们的使用方式并不友好，例如我们去获取一个请求参数 `name`，我们会定义如下：

```
@RequestParam("name") String name
```

此时，我们会发现变量名称大概率会被定义成 `RequestParam` 值。所以我们是不是可以用下面这种方式来定义：

```
@RequestParam String name
```

这种方式确实是可以的，本地测试也能通过。这里我还给出了完整的代码，你可以感受下这两者的区别。

 复制代码

```
1 @RequestMapping(path = "/hi1", method = RequestMethod.GET)
2 public String hi1(@RequestParam("name") String name){
3     return name;
4 };
5
6 @RequestMapping(path = "/hi2", method = RequestMethod.GET)
7 public String hi2(@RequestParam String name){
8     return name;
9 };
```

很明显，对于喜欢追究极致简洁的同学来说，这个酷炫的功能是一个福音。但当我们换一个项目时，有可能上线后就失效了，然后报错 500，提示匹配不上。

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Mar 13 15:51:20 CST 2021

There was an unexpected error (type=Internal Server Error, status=500).

Name for argument type [java.lang.String] not available, and parameter name information not found in class file either.

java.lang.IllegalArgumentException: Name for argument type [java.lang.String] not available, and parameter name information not found in class file either.

at org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.updateNamedValueInfo(AbstractNamedValueMethodArgumentResolver.java:134)

at org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.getNamedValueInfo(AbstractNamedValueMethodArgumentResolver.java:117)

at org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.resolveArgument(AbstractNamedValueMethodArgumentResolver.java:103)

at org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)

at org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.java:167)

at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:134)

案例解析

要理解这个问题出现的原因，首先我们需要把这个问题复现出来。例如我们可以修改下 pom.xml 来关掉两个选项：

 复制代码

```
1 <plugin>
2     <groupId>org.apache.maven.plugins</groupId>
3     <artifactId>maven-compiler-plugin</artifactId>
4     <configuration>
5         <debug>>false</debug>
6         <parameters>>false</parameters>
7     </configuration>
8 </plugin>
```

上述配置显示关闭了 parameters 和 debug，这 2 个参数的作用你可以参考下面的表格：

<debug>	Set to true to include debugging information in the compiled class files. Default value is: true. User property is: maven.compiler.debug.
<parameters>	Set to true to generate metadata for reflection on method parameters. Default value is: false. User property is: maven.compiler.parameters.

通过上述描述，我们可以看出这 2 个参数控制了一些 debug 信息是否加进 class 文件中。我们可以开启这两个参数来编译，然后使用下面的命令来查看信息：

```
javap -verbose HelloWorldController.class
```

执行完命令后，我们会看到以下 class 信息：

```
public java.lang.String hi3(java.lang.String);
  descriptor: (Ljava/lang/String;)Ljava/lang/String;
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=2, args_size=2
      0: aload_1
      1: areturn
  LineNumberTable:
    line 40: 0
  LocalVariableTable:
    Start Length Slot Name Signature
      0      2     0 this Lcom/spring/puzzle/web/ur1/HelloWorldController;
      0      2     1 name Ljava/lang/String;
  MethodParameters:
    Name Flags
    name
  RuntimeVisibleAnnotations:
    0: #38(#39=[s#51],#41=[e#42.#43])
  RuntimeVisibleParameterAnnotations:
    parameter 0:
      0: #49()
```

debug 参数开启的部分信息就是 LocalVariableTable，而 paramters 参数开启的信息就是 MethodParameters。观察它们的信息，你会发现它们都含有参数名 name。

如果你关闭这两个参数，则 name 这个名称自然就没有了。而这个方法本身在 @RequestParam 中又没有指定名称，那么 Spring 此时还能找到解析的方法么？

答案是否定的，这里我们可以顺带说下 Spring 解析请求参数名称的过程，参考代码 `AbstractNamedValueMethodArgumentResolver#updateNamedValueInfo`：

[复制代码](#)

```
1 private NamedValueInfo updateNamedValueInfo(MethodParameter parameter, NamedVa
2     String name = info.name;
3     if (info.name.isEmpty()) {
4         name = parameter.getParameterName();
5         if (name == null) {
6             throw new IllegalArgumentException(
7                 "Name for argument type [" + parameter.getNestedParameterType()
8                 "] not available, and parameter name information not found in c
9         }
10    }
11    String defaultValue = (ValueConstants.DEFAULT_NONE.equals(info.defaultValue
12    return new NamedValueInfo(name, info.required, defaultValue);
13 }
```

其中 `NamedValueInfo` 的 `name` 为 `@RequestParam` 指定的值。很明显，在本案例中，为 `null`。

所以这里我们就会尝试调用 `parameter.getParameterName()` 来获取参数名作为解析请求参数的名称。但是，很明显，关掉上面两个开关后，就不可能在 class 文件中找到参数名了，这点可以从下面的调试试图中得到验证：



当参数名不存在，`@RequestParam` 也没有指明，自然就无法决定到底要使用什么名称去获取请求参数，所以就会报本案例的错误。

问题修正

模拟出了问题是如何发生的，我们自然可以通过开启这两个参数让其工作起来。但是思考这两个参数的作用，很明显，它可以让我们的程序体积更小，所以很多项目都会青睐去关闭这两个参数。

为了以不变应万变，正确的修正方式是**必须显式在 @RequestParam 中指定请求参数名**。具体修改如下：

```
@RequestParam("name") String name
```


通过这个案例，我们可以看出：很多功能貌似可以永远工作，但是实际上，只是在特定的条件下而已。另外，这里再拓展下，IDE 都喜欢开启相关 debug 参数，所以 IDE 里运行的程序不见得对产线适应，例如针对 parameters 这个参数，IDEA 默认就开启了。

另外，本案例围绕的都是 @RequestParam，其实 @PathVariable 也有一样的问题。这里你要注意。

那么说到这里，我顺带提一个可能出现的小困惑：我们这里讨论的参数，和 @QueryParam、@PathParam 有什么区别？实际上，后者都是 JAX-RS 自身的注解，不需要额外导包。而 @RequestParam 和 @PathVariable 是 Spring 框架中的注解，需要额外导入依赖包。另外不同注解的参数也不完全一致。

案例 3：未考虑参数是否可选

在上面的案例中，我们提到了 @RequestParam 的使用。而对于它的使用，我们常常会遇到另外一个问题。当需要特别多的请求参数时，我们往往会忽略其中一些参数是否可选。例如存在类似这样的代码：

 复制代码

```
1 @RequestMapping(path = "/hi4", method = RequestMethod.GET)
2 public String hi4(@RequestParam("name") String name, @RequestParam("address"))
3     return name + ":" + address;
4 };
```

在访问 <http://localhost:8080/hi4?name=xiaoming&address=beijing> 时并不会出问题，但是一旦用户仅仅使用 name 做请求（即 <http://localhost:8080/hi4?name=xiaoming>）

name=xiaoming) 时, 则会直接报错如下:

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Mar 13 15:23:41 CST 2021

There was an unexpected error (type=Bad Request, status=400).

Required String parameter 'address' is not present

org.springframework.web.bind.MissingServletRequestParameterException: Required String parameter 'address' is not present

at

org.springframework.web.method.annotation.RequestParamMethodArgumentResolver.handleMissingValue(RequestParamMethodArgumentResolver.java:204)

at

org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.resolveArgument(AbstractNamedValueMethodArgumentResolver.java:121)

at

org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)

at org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.java:167)

at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:134)

at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:106)

at

此时, 返回错误码 400, 提示请求格式错误: 此处缺少 address 参数。

实际上, 部分初学者即使面对这个错误, 也会觉得惊讶, 既然不存在 address, address 应该设置为 null, 而不应该是直接报错不是么? 接下来我们就分析下。

案例解析


要了解这个错误出现的根本原因, 你就需要了解请求参数的发生位置。

实际上, 这里我们也能按注解名 (@RequestParam) 来确定解析发生的位置是在 RequestParamMethodArgumentResolver 中。为什么是它?

追根溯源, 针对当前案例, 当根据 URL 匹配上要执行的方法是 hi4 后, 要反射调用它, 必须解析出方法参数 name 和 address 才可以。而它们被 @RequestParam 注解修饰, 所以解析器借助 RequestParamMethodArgumentResolver 就成了很自然的事情。

接下来我们看下 RequestParamMethodArgumentResolver 对参数解析的一些关键操作, 参考其父类方法

AbstractNamedValueMethodArgumentResolver#resolveArgument:

 复制代码

```
1 public final Object resolveArgument(MethodParameter parameter, @Nullable Model
2     NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactor
3     NamedValueInfo namedValueInfo = getNamedValueInfo(parameter);
4     MethodParameter nestedParameter = parameter.nestedIfOptional();
5     //省略其他非关键代码
```

```
6 //获取请求参数
7 Object arg = resolveName(resolvedName.toString(), nestedParameter, webReque
8 if (arg == null) {
9     if (namedValueInfo.defaultValue != null) {
10         arg = resolveStringValue(namedValueInfo.defaultValue);
11     }
12     else if (namedValueInfo.required && !nestedParameter.isOptional()) {
13         handleMissingValue(namedValueInfo.name, nestedParameter, webRequest);
14     }
15     arg = handleNullValue(namedValueInfo.name, arg, nestedParameter.getNeste
16 }
17 //省略后续代码：类型转化等工作
18 return arg;
19 }
```

如代码所示，当缺少请求参数的时候，通常会按照以下几个步骤进行处理。

1. 查看 namedValueInfo 的默认值，如果存在则使用它

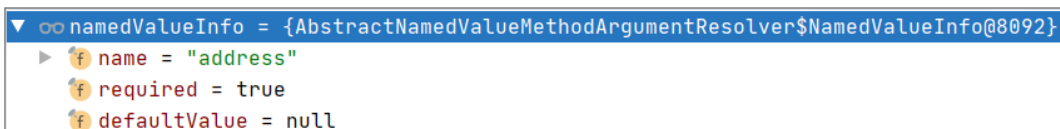
这个变量实际是通过下面的方法来获取的，参考

RequestParamMethodArgumentResolver#createNamedValueInfo:

复制代码

```
1 @Override
2 protected NamedValueInfo createNamedValueInfo(MethodParameter parameter) {
3     RequestParam ann = parameter.getParameterAnnotation(RequestParam.class);
4     return (ann != null ? new RequestParamNamedValueInfo(ann) : new RequestPara
5 }
```

实际上就是 @RequestParam 的相关信息，我们调试下，就可以验证这个结论，具体如下图所示：



```
▼ namedValueInfo = {AbstractNamedValueMethodArgumentResolver$NamedValueInfo@8092}
  ▶ name = "address"
  ▶ required = true
  ▶ defaultValue = null
```

2. 在 @RequestParam 没有指明默认值时，会查看这个参数是否必须，如果必须，则按错误处理

判断参数是否必须的代码即为下述关键代码行：

```
namedValueInfo.required && !nestedParameter.isOptional()
```

很明显，若要判定一个参数是否是必须的，需要同时满足两个条件：条件 1 是 `@RequestParam` 指明了必须（即属性 `required` 为 `true`，实际上它也是默认值），条件 2 是要求 `@RequestParam` 标记的参数本身不是可选的。

我们可以通过 `MethodParameter#isOptional` 方法看下可选的具体含义：

[复制代码](#)

```
1 public boolean isOptional() {
2     return (getParameterType() == Optional.class || hasNullableAnnotation() ||
3             (KotlinDetector.isKotlinReflectPresent() &&
4              KotlinDetector.isKotlinType(getContainingClass()) &&
5              KotlinDelegate.isOptional(this)));
6 }
```

在不使用 Kotlin 的情况下，所谓可选，就是参数的类型为 `Optional`，或者任何标记了注解名为 `Nullable` 且 `RetentionPolicy` 为 `RUNTIM` 的注解。

3. 如果不是必须，则按 null 去做具体处理

如果接受类型是 `boolean`，返回 `false`，如果是基本类型则直接报错，这里不做展开。

结合我们的案例，我们的参数符合步骤 2 中判定为必选的条件，所以最终会执行方法 `AbstractNamedValueMethodArgumentResolver#handleMissingValue`：

[复制代码](#)

```
1 protected void handleMissingValue(String name, MethodParameter parameter) thro
2     throw new ServletRequestBindingException("Missing argument '" + name +
3         "' for method parameter of type " + parameter.getNestedParameterType(
4     }
```

问题修正

通过案例解析，我们很容易就能修正这个问题，就是让参数有默认值或为非可选即可，具体方法包含以下几种。

1. 设置 @RequestParam 的默认值

修改代码如下：

```
@RequestParam(value = "address", defaultValue = "no address") String address
```

2. 设置 @RequestParam 的 required 值

修改代码如下：

```
@RequestParam(value = "address", required = false) String address)
```

3. 标记任何名为 Nullable 且 RetentionPolicy 为 RUNTIME 的注解

修改代码如下：

```
//org.springframework.lang.Nullable 可以  
//edu.umd.cs.findbugs.annotations.Nullable 可以  
@RequestParam(value = "address") @Nullable String address
```

4. 修改参数类型为 Optional


修改代码如下：

```
@RequestParam(value = "address") Optionaladdress
```

从这些修正方法不难看出：假设你不学习源码，解决方法就可能只局限于一两种，但是深入源码后，解决方法就变得格外多了。这里要特别强调的是：**在 Spring Web 中，默认情况下，请求参数是必选项。**


案例 4：请求参数格式错误

当我们使用 Spring URL 相关的注解，会发现 Spring 是能够完成自动转化的。例如在下面的代码中，age 可以被直接定义为 int 这种基本类型（Integer 也可以），而不是必须是 String 类型。

 复制代码

```
1 @RequestMapping(path = "/hi5", method = RequestMethod.GET)
2 public String hi5(@RequestParam("name") String name, @RequestParam("age") int
3     return name + " is " + age + " years old";
4 };
```

鉴于 Spring 的强大转化功能，我们断定 Spring 也支持日期类型的转化（也确实如此），于是我们可能会写出类似下面这样的代码：

 复制代码

```
1 @RequestMapping(path = "/hi6", method = RequestMethod.GET)
2 public String hi6(@RequestParam("Date") Date date){
3     return "date is " + date ;
4 };
```

然后，我们使用一些看似明显符合日期格式的 URL 来访问，例如

🔗 <http://localhost:8080/hi6?date=2021-5-1 20:26:53>，我们会发现 Spring 并不能完成转化，而是报错如下：

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Mar 14 08:49:53 CST 2021

There was an unexpected error (type=Bad Request, status=400).

Failed to convert value of type 'java.lang.String' to required type 'java.util.Date'; nested exception is org.springframework.core.convert.ConversionFailedException: Failed to convert from type [java.lang.String] to type [org.springframework.web.bind.annotation.RequestParam java.util.Date] for value '2021-5-1 20:26:53'; nested exception is java.lang.IllegalArgumentException
org.springframework.web.method.annotation.MethodArgumentTypeMismatchException: Failed to convert value of type 'java.lang.String' to required type 'java.util.Date'; nested exception is org.springframework.core.convert.ConversionFailedException: Failed to convert from type [java.lang.String] to type [org.springframework.web.bind.annotation.RequestParam java.util.Date] for value '2021-5-1 20:26:53'; nested exception is java.lang.IllegalArgumentException
at
org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.resolveArgument(AbstractNamedValueMethodArgumentResolver.java:121)
at
org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)

此时，返回错误码 400，错误信息为"Failed to convert value of type 'java.lang.String' to required type 'java.util.Date'".

如何理解这个案例？如果实现自动转化，我们又需要做什么？

案例解析

不管是使用 @PathVariable 还是 @RequestParam，我们一般解析出的结果都是一个 String 或 String 数组。例如，使用 @RequestParam 解析的关键代码参考

RequestParamMethodArgumentResolver#resolveName 方法:

[复制代码](#)

```
1 @Nullable
2 protected Object resolveName(String name, MethodParameter parameter, NativeWeb
3     //省略其他非关键代码
4     if (arg == null) {
5         String[] paramValues = request.getParameterValues(name);
6         if (paramValues != null) {
7             arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
8         }
9     }
10    return arg;
11 }
```

这里我们调用的"`request.getParameterValues(name)`", 返回的是一个 `String` 数组, 最终给上层调用者返回的是单个 `String` (如果只有一个元素时) 或者 `String` 数组。

所以很明显, 在这个测试程序中, 我们给上层返回的是一个 `String`, 这个 `String` 的值最终是需要做转化才能赋值给其他类型。例如对于案例中的"`int age`"定义, 是需要转化为 `int` 基本类型的。这个基本流程可以通过

`AbstractNamedValueMethodArgumentResolver#resolveArgument` 的关键代码来验证:

[复制代码](#)

```
1 public final Object resolveArgument(MethodParameter parameter, @Nullable Model
2     NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactor
3     //省略其他非关键代码
4     Object arg = resolveName(resolvedName.toString(), nestedParameter, webReque
5     //以此为界, 前面代码为解析请求参数, 后续代码为转化解析出的参数
6     if (binderFactory != null) {
7         WebDataBinder binder = binderFactory.createBinder(webRequest, null, name
8         try {
9             arg = binder.convertIfNecessary(arg, parameter.getParameterType(), pa
10        }
11        //省略其他非关键代码
12    }
13    //省略其他非关键代码
14    return arg;
15 }
```

实际上在前面我们曾经提到过这个转化的基本逻辑，所以这里不再详述它具体是如何发生的。

在这里你只需要回忆出它是需要**根据源类型和目标类型寻找转化器来执行转化的**。在这里，对于 age 而言，最终找出的转化器是 StringToNumberConverterFactory。而对于 Date 型的 Date 变量，在本案例中，最终找到的是 ObjectToObjectConverter。它的转化过程参考下面的代码：

[复制代码](#)

```
1 public Object convert(@Nullable Object source, TypeDescriptor sourceType, Type
2     if (source == null) {
3         return null;
4     }
5     Class<?> sourceClass = sourceType.getType();
6     Class<?> targetClass = targetType.getType();
7     //根据源类型去获取构建出目标类型的方法：可以是工厂方法（例如 valueOf、from 方法）也可以
8     Member member = getValidatedMember(targetClass, sourceClass);
9     try {
10         if (member instanceof Method) {
11             //如果是工厂方法，通过反射创建目标实例
12         }
13         else if (member instanceof Constructor) {
14             //如果是构造器，通过反射创建实例
15             Constructor<?> ctor = (Constructor<?>) member;
16             ReflectionUtils.makeAccessible(ctor);
17             return ctor.newInstance(source);
18         }
19     }
20     catch (InvocationTargetException ex) {
21         throw new ConversionFailedException(sourceType, targetType, source, ex.g
22     }
23     catch (Throwable ex) {
24         throw new ConversionFailedException(sourceType, targetType, source, ex);
25     }
```

当使用 ObjectToObjectConverter 进行转化时，是根据反射机制带着源目标类型来查找可能的构造目标实例方法，例如构造器或者工厂方法，然后再次通过反射机制来创建一个目标对象。所以对于 Date 而言，最终调用的是下面的 Date 构造器：

[复制代码](#)

```
1 public Date(String s) {
2     this(parse(s));
3 }
```

然而，我们传入的 [2021-5-1 20:26:53](#) 虽然确实是一种日期格式，但用来作为 Date 构造器参数是不支持的，最终报错，并被上层捕获，转化为 ConversionFailedException 异常。这就是这个案例背后的故事了。

问题修正

那么怎么解决呢？提供两种方法。

1. 使用 Date 支持的格式

例如下面的测试 URL 就可以工作起来：

<http://localhost:8080/hi6?date=Sat, 12 Aug 1995 13:30:00 GMT>

2. 使用好内置格式转化器

实际上，在 Spring 中，要完成 String 对于 Date 的转化，ObjectToObjectConverter 并不是最好的转化器。我们可以使用更强大的 AnnotationParserConverter。在 Spring 初始化时，会构建一些针对日期型的转化器，即相应的一些 AnnotationParserConverter 的实例。但是为什么有时候用不上呢？

这是因为 AnnotationParserConverter 有目标类型的要求，这点我们可以通过调试角度来看下，参考 FormattingConversionService#addFormatterForFieldAnnotation 方法的调试试图：

```
Set<Class<?>> fieldTypes = annotationFormatterFactory.getFieldTypes(); fieldTypes: size = 3
for (Class<?> fieldType : fieldTypes) { fieldType: "class java.util.Date" fieldTypes: size = 3
    addConverter(new AnnotationPrinterConverter(annotationType, annotationFormatterFactory, fieldType));
    addConverter(new AnnotationParserConverter(annotationType, annotationFormatterFactory, fieldType));
}
+ {Class@2209} "interface org.springframework.format.annotation.DateTimeFormat"
```


这是适应于 String 到 Date 类型的转化器 AnnotationParserConverter 实例的构造过程，其需要的 annotationType 参数为 DateTimeFormat。

annotationType 的作用正是为了帮助判断是否能用这个转化器，这一点可以参考代码 AnnotationParserConverter#matches:

[复制代码](#)

```
1 @Override
2 public boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType) {
3     return targetType.hasAnnotation(this.annotationType);
4 }
```

最终构建出来的转化器相关信息可以参考下图:



图中构造出的转化器是可以用来转化 String 到 Date，但是它要求我们标记 @DateTimeFormat。很明显，我们的参数 Date 并没有标记这个注解，所以这里为了使用这个转化器，我们可以使用上它并提供合适的格式。这样就可以让原来不工作的 URL 工作起来，具体修改代码如下：

[复制代码](#)

```
1 @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss") Date date
```


以上即为本案例的解决方案。除此之外，我们完全可以制定一个转化器来帮助我们完成转化，这里不再赘述。另外，通过这个案例，我们可以看出：尽管 Spring 给我们提供了很多内置的转化功能，但是我们一定要注意，格式是否符合对应的要求，否则代码就可能会失效。

重点回顾

通过这一讲的学习，我们了解到了在 Spring 解析 URL 中的一些常见错误及其背后的深层原因。这里再次回顾下重点：


1. 当我们使用 @PathVariable 时，一定要注意传递的值是不是含有 / ；

2. 当我们使用 `@RequestParam`、`@PathVariable` 等注解时，一定要意识到一个问题，虽然下面这两种方式（以 `@RequestParam` 使用示例）都可以，但是后者在一些项目中并不能正常工作，因为很多产线的编译配置会去掉不是必须的调试信息。

 复制代码

```
1 @RequestMapping(path = "/hi1", method = RequestMethod.GET)
2 public String hi1(@RequestParam("name") String name){
3     return name;
4 };
5 //方式2: 没有显式指定RequestParam的“name”，这种方式有时候会不行
6 @RequestMapping(path = "/hi2", method = RequestMethod.GET)
7 public String hi2(@RequestParam String name){
8     return name;
9 };
```

3. 任何一个参数，我们都需要考虑它是可选的还是必须的。同时，你一定要想到参数类型的定义到底能不能从请求中自动转化而来。Spring 本身给我们内置了很多转化器，但是我们要以合适的方式使用上它。另外，Spring 对很多类型的转化设计都很贴心，例如使用下面的注解就能解决自定义日期格式参数转化问题。


 复制代码

```
1 @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss") Date date
```

希望这些核心知识点，能帮助你高效解析 URL。

思考题

关于 URL 解析，其实还有许多让我们惊讶的地方，例如案例 2 的部分代码：

 复制代码

```
1 @RequestMapping(path = "/hi2", method = RequestMethod.GET)
2 public String hi2(@RequestParam("name") String name){
3     return name;
4 };
```

在上述代码的应用中，我们可以使用 <http://localhost:8080/hi2?name=xiaoming&name=hanmeimei> 来测试下，结果会返回什么呢？你猜会是

🔗 [xiaoming&name=hanmeimei](#) 么?

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 导读 | 5分钟轻松了解一个HTTP请求的处理过程

下一篇 10 | Spring Web Header 解析常见错误

精选留言 (6)

💬 写留言



panSky

2021-05-13

我运行思考题结果是: xiaoming,hanmeimei
看源码是两个同名请求参数name被放到String[]中, Spring转换器转换String[]->String
时, 用 ", " 分隔符拼接后返回。
看别人运行结果不一样, 很疑惑。
期待正确答案。

展开 ∨

💬 1

👍 1



哦吼掉了

2021-05-12

思考题: 应该是xiaoming
RequestParamMethodArgumentResolver#resolveArgument 134行, 相同参数
只会取第一个参数

有个问题, @RequestBody和@RequestParam区别是不是可以加餐一下? 刚学习的时候...

展开 ∨

💬

👍 1

**程序员人生**

2021-05-18

看到request.getParameterValues(name)，我仿佛回到了十几年前，刚毕业那会
展开 ▾

**望舒**

2021-05-12

结果居然跟想象中的不一样，程序没有识别后面的参数。
展开 ▾

**Yuuuuuu**

2021-05-12

对于RequestParam和RequestBody的使用也有一些疑惑，哪些参数可以被RequestParam获取，哪些可以被RequestBody获取？
展开 ▾

**魔**

2021-05-12

虽然熟悉，但看下来收获很大
展开 ▾

