



下载APP



16 | Spring Exception 常见错误

2021-05-28 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)**讲述：傅健**

时长 14:34 大小 13.35M



你好，我是傅健。

今天，我们来学习 Spring 的异常处理机制。Spring 提供了一套健全的异常处理框架，以便我们在开发应用的时候对异常进行处理。但是，我们也会在使用的时候遇到一些麻烦，接下来我将通过两个典型的错误案例，带着你结合源码进行深入了解。

案例 1：小心过滤器异常

为了方便讲解，我们还是沿用之前在事务处理中用到的学生注册的案例，来讨论异常的问题：



复制代码

```
1 @Controller
2 @Slf4j
3 public class StudentController {
4     public StudentController(){
5         System.out.println("construct");
6     }
7
8
9
10    @PostMapping("/regStudent/{name}")
11    @ResponseBody
12    public String saveUser(String name) throws Exception {
13        System.out.println(".....用户注册成功");
14        return "success";
15    }
16 }
```

为了保证安全，这里需要给请求加一个保护，通过验证 Token 的方式来验证请求的合法性。这个 Token 需要在每次发送请求的时候带在请求的 header 中，header 的 key 是 Token。

为了校验这个 Token，我们引入了一个 Filter 来处理这个校验工作，这里我使用了一个最简单的 Token：111111。

当 Token 校验失败时，就会抛出一个自定义的 NotAllowException，交由 Spring 处理：

[复制代码](#)

```
1 @WebFilter
2 @Component
3 public class PermissionFilter implements Filter {
4     @Override
5     public void doFilter(ServletRequest request, ServletResponse response, Fil
6         HttpServletRequest httpServletRequest = (HttpServletRequest) request;
7         String token = httpServletRequest.getHeader("token");
8
9
10        if (!"111111".equals(token)) {
11            System.out.println("throw NotAllowException");
12            throw new NotAllowException();
13        }
14        chain.doFilter(request, response);
15    }
16
17
18    @Override
19    public void init(FilterConfig filterConfig) throws ServletException {
20    }
```

```
21
22
23     @Override
24     public void destroy() {
25     }
```

NotAllowedException 就是一个简单的 RuntimeException 的子类：

[复制代码](#)

```
1 public class NotAllowException extends RuntimeException {
2     public NotAllowException() {
3         super();
4     }
5 }
```

同时，新增了一个 RestControllerAdvice 来处理这个异常，处理方式也很简单，就是返回一个 403 的 resultCode：

[复制代码](#)

```
1 @RestControllerAdvice
2 public class NotAllowExceptionHandler {
3     @ExceptionHandler(NotAllowedException.class)
4     @ResponseBody
5     public String handle() {
6         System.out.println("403");
7         return "{\"resultCode\": 403}";
8     }
9 }
```

为了验证一下失败的情况，我们模拟了一个请求，在 HTTP 请求头里加上一个 Token，值为 111，这样就会引发错误了，我们可以看看会不会被 NotAllowExceptionHandler 处理掉。

然而，在控制台上，我们只看到了下面这样的输出，这其实就说明了 NotAllowExceptionHandler 并没有生效。

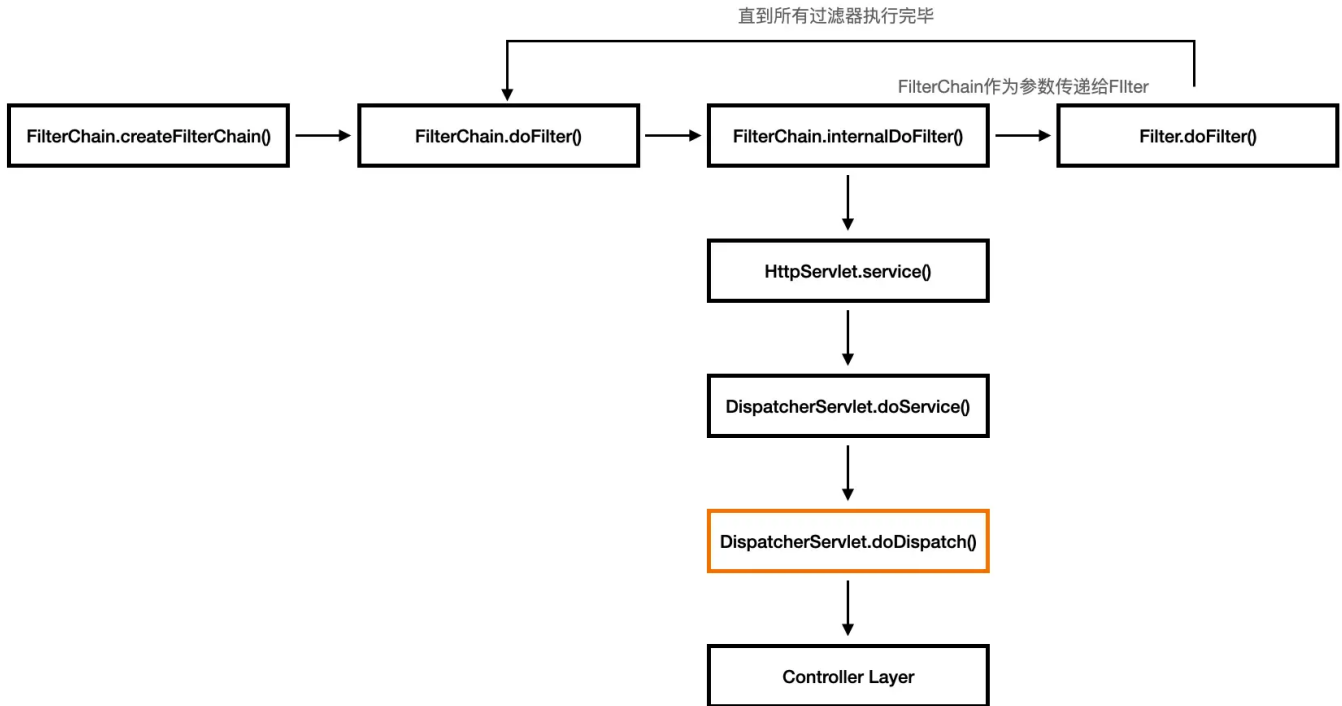
[复制代码](#)

```
1 throw NotAllowException
```

想下问题出在哪呢？我们不妨对 Spring 的异常处理过程先做一个了解。

案例解析

我们先来回顾一下 [第 13 课](#) 讲过的过滤器执行流程图，这里我细化了一下：




从这张图中可以看出，当所有的过滤器被执行完毕以后，Spring 才会进入 Servlet 相关的处理，而 `DispatcherServlet` 才是整个 Servlet 处理的核心，它是前端控制器设计模式的实现，提供 Spring Web MVC 的集中访问点并负责职责的分派。正是在这里，Spring 处理了请求和处理器之间的对应关系，以及这个案例我们所关注的问题——统一异常处理。

其实说到这里，我们已经了解到过滤器内异常无法被统一处理的大致原因，就是因为异常处理发生在上图的红色区域，即 `DispatcherServlet` 中的 `doDispatch()`，而此时，过滤器已经全部执行完毕了。

下面我们将深入分析 Spring Web 对异常统一处理的逻辑，深刻理解其内部原理。

首先我们来了解下 `ControllerAdvice` 是如何被 Spring 加载并对外暴露的。在 Spring Web 的核心配置类 `WebMvcConfigurationSupport` 中，被 `@Bean` 修饰的 `handlerExceptionResolver()`，会调用 `addDefaultHandlerExceptionResolvers()` 来添加默认的异常解析器。

 复制代码

```
1 @Bean
2 public HandlerExceptionResolver handlerExceptionResolver(
3     @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager con
4     List<HandlerExceptionResolver> exceptionResolvers = new ArrayList<>();
5     configureHandlerExceptionResolvers(exceptionResolvers);
6     if (exceptionResolvers.isEmpty()) {
7         addDefaultHandlerExceptionResolvers(exceptionResolvers, contentNegotiati
8     }
9     extendHandlerExceptionResolvers(exceptionResolvers);
10    HandlerExceptionResolverComposite composite = new HandlerExceptionResolverC
11    composite.setOrder(0);
12    composite.setExceptionResolvers(exceptionResolvers);
13    return composite;
14 }
```

最终按照下图的调用栈，Spring 实例化了 `ExceptionHandlerExceptionResolver` 类。

<init>:108, ExceptionHandlerExceptionResolver


createExceptionHandlerExceptionResolver:1022, WebMvcConfigurationSupport

createExceptionHandlerExceptionResolver:465, WebMvcAutoConfiguration\$EnableWebMvcConfiguration

addDefaultHandlerExceptionResolvers:994, WebMvcConfigurationSupport

handlerExceptionResolver:948, WebMvcConfigurationSupport


从源码中我们可以看出，`ExceptionHandlerExceptionResolver` 类实现了 `InitializingBean` 接口，并覆写了 `afterPropertiesSet()`。

 复制代码

```
1 public void afterPropertiesSet() {
2     // Do this first, it may add ResponseBodyAdvice beans
3     initExceptionHandlerAdviceCache();
4     //省略非关键代码
5 }
```

并在 `initExceptionHandlerAdviceCache()` 中完成了所有 `ControllerAdvice` 中的 `ExceptionHandler` 的初始化。其具体操作，就是查找所有 `@ControllerAdvice` 注解的 `Bean`，把它们放到成员变量 `exceptionHandlerAdviceCache` 中。

在我们这个案例里，就是指 `NotAllowedExceptionHandler` 这个异常处理器。

 复制代码

```
1 private void initExceptionHandlerAdviceCache() {
2     //省略非关键代码
3     List<ControllerAdviceBean> adviceBeans = ControllerAdviceBean.findAnnotated
4     for (ControllerAdviceBean adviceBean : adviceBeans) {
5         Class<?> beanType = adviceBean.getBeanType();
6         if (beanType == null) {
7             throw new IllegalStateException("Unresolvable type for ControllerAdvi
8         }
9         ExceptionHandlerMethodResolver resolver = new ExceptionHandlerMethodReso
10        if (resolver.hasExceptionMappings()) {
11            this.exceptionHandlerAdviceCache.put(adviceBean, resolver);
12        }
13    //省略非关键代码
14 }
```

到这，我们可以总结一下，WebMvcConfigurationSupport 中的 handlerExceptionHandlerResolver() 实例化并注册了一个 ExceptionHandlerExceptionHandlerResolver 的实例，而所有被 @ControllerAdvice 注解修饰的异常处理器，都会在 ExceptionHandlerExceptionHandlerResolver 实例化的时候自动扫描并装载在其类成员变量 exceptionHandlerAdviceCache 中。

当第一次请求发生时，DispatcherServlet 中的 initHandlerExceptionResolvers() 将获取所有注册到 Spring 的 HandlerExceptionHandlerResolver 类型的实例，而 ExceptionHandlerExceptionHandlerResolver 恰好实现了 HandlerExceptionHandlerResolver 接口，这些 HandlerExceptionHandlerResolver 类型的实例则会被写入到类成员变量 handlerExceptionHandlerResolvers 中。

 复制代码

```
1 private void initHandlerExceptionResolvers(ApplicationContext context) {
2     this.handlerExceptionResolvers = null;
3
4     if (this.detectAllHandlerExceptionResolvers) {
5         // Find all HandlerExceptionHandlerResolvers in the ApplicationContext, includi
6         Map<String, HandlerExceptionHandlerResolver> matchingBeans = BeanFactoryUtils
7             .beansOfTypeIncludingAncestors(context, HandlerExceptionHandlerResolver.c
8         if (!matchingBeans.isEmpty()) {
9             this.handlerExceptionResolvers = new ArrayList<>(matchingBeans.values
10            // We keep HandlerExceptionHandlerResolvers in sorted order.
11            AnnotationAwareOrderComparator.sort(this.handlerExceptionResolvers);
12        }
13        //省略非关键代码
14    }
15 }
```


接着我们再来了解下 **ControllerAdvice** 是如何被 **Spring** 消费并处理异常的。下文贴出的是核心类 **DispatcherServlet** 中的核心方法 **doDispatch()** 的部分代码：

[复制代码](#)

```
1  protected void doDispatch(HttpServletRequest request, HttpServletResponse resp
2      //省略非关键代码
3
4      try {
5          ModelAndView mv = null;
6          Exception dispatchException = null;
7          try {
8              //省略非关键代码
9              //查找当前请求对应的 handler，并执行
10             //省略非关键代码
11         }
12         catch (Exception ex) {
13             dispatchException = ex;
14         }
15         catch (Throwable err) {
16             dispatchException = new NestedServletException("Handler dispatch fail
17         }
18         processDispatchResult(processedRequest, response, mappedHandler, mv, dis
19     }
20     //省略非关键代码
```

Spring 在执行用户请求时，当在“查找”和“执行”请求对应的 handler 过程中发生异常，就会把异常赋值给 **dispatchException**，再交给 **processDispatchResult()** 进行处理。

[复制代码](#)

```
1  private void processDispatchResult(HttpServletRequest request, HttpServletResponse
2      @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv
3      @Nullable Exception exception) throws Exception {
4      boolean errorView = false;
5      if (exception != null) {
6          if (exception instanceof ModelAndViewDefiningException) {
7              mv = ((ModelAndViewDefiningException) exception).getModelAndView();
8          }
9          else {
10             Object handler = (mappedHandler != null ? mappedHandler.getHandler()
11             mv = processHandlerException(request, response, handler, exception);
12             errorView = (mv != null);
13         }
```

```
14     }  
15     //省略非关键代码
```

进一步处理后，即当 Exception 不为 null 时，继续交给 processHandlerException 处理。

[复制代码](#)

```
1  protected ModelAndView processHandlerException(HttpServletRequest request, Http  
2      @Nullable Object handler, Exception ex) throws Exception {  
3      //省略非关键代码  
4      ModelAndView exMv = null;  
5      if (this.handlerExceptionResolvers != null) {  
6          for (HandlerExceptionResolver resolver : this.handlerExceptionResolvers)  
7              exMv = resolver.resolveException(request, response, handler, ex);  
8              if (exMv != null) {  
9                  break;  
10             }  
11         }  
12     }  
13     //省略非关键代码  
14 }
```

然后，processHandlerException 会从类成员变量 handlerExceptionResolvers 中获取有效的异常解析器，对异常进行解析。

显然，这里的 handlerExceptionResolvers 一定包含我们声明的 NotAllowExceptionHandler#NotAllowException 的异常处理器的 ExceptionHandlerExceptionResolver 包装类。

问题修正

为了利用 Spring MVC 的异常处理机制，我们需要对 Filter 做一些改造。手动捕获异常，并将异常 HandlerExceptionResolver 进行解析处理。

我们可以这样修改 PermissionFilter，注入 HandlerExceptionResolver：

[复制代码](#)

```
1  @Autowired  
2  @Qualifier("handlerExceptionResolver")  
3  private HandlerExceptionResolver resolver;
```


然后，在 `doFilter` 里捕获异常并交给 `HandlerExceptionResolver` 处理：

[复制代码](#)

```
1 public void doFilter(ServletRequest request, ServletResponse response, Fil
2     HttpServletRequest httpServletRequest = (HttpServletRequest) request;
3     HttpServletResponse httpServletResponse = (HttpServletResponse) respon
4     String token = httpServletRequest.getHeader("token");
5     if (!"111111".equals(token)) {
6         System.out.println("throw NotAllowException");
7         resolver.resolveException(httpServletRequest, httpServletResponse,
8             return;
9     }
10    chain.doFilter(request, response);
11 }
```

当我们尝试用错误的 Token 请求，控制台得到了以下信息：

[复制代码](#)

```
1 throw NotAllowException
2 403
```

返回的 JSON 是：

[复制代码](#)

```
1 {"resultCode": 403}
```


再换成正确的 Token 请求，这些错误信息就都没有了，到这，问题解决了。

案例 2：特殊的 404 异常

继续沿用学生注册的案例，为了防止一些异常的访问，我们需要记录所有 404 状态的访问记录，并返回一个我们的自定义结果。

一般使用 RESTful 接口时我们会统一返回 JSON 数据，返回值格式如下：

```
1 {"resultCode": 404}
```


 复制代码

但是 Spring 对 404 异常是进行了默认资源映射的，并不会返回我们想要的结果，也不会对这种错误做记录。

于是我们添加了一个 `ExceptionHandlerController`，它被声明成 `@RestControllerAdvice` 来全局捕获 Spring MVC 中抛出的异常。


`ExceptionHandler` 的作用正是用来捕获指定的异常：

```
1 @RestControllerAdvice
2 public class MyExceptionHandler {
3     @ResponseStatus(HttpStatus.NOT_FOUND)
4     @ExceptionHandler(Exception.class)
5     @ResponseBody
6     public String handle404() {
7         System.out.println("404");
8         return "{\"resultCode\": 404}";
9     }
10 }
```

 复制代码

我们尝试发送一个错误的 URL 请求到之前实现过的 `/regStudent` 接口，并把请求地址换成 `/regStudent1`，得到了以下结果：


```
1 {"timestamp":"2021-05-19T22:24:01.559+0000","status":404,"error":"Not Found", "}
```

 复制代码

很显然，这个结果不是我们想要的，看起来应该是 Spring 默认返回的结果。那是什么原因导致 Spring 没有使用我们定义的异常处理器呢？

案例解析

我们可以从异常处理的核心处理代码开始分析，`DispatcherServlet` 中的 `doDispatch()` 核心代码如下：

 复制代码

```
1 protected void doDispatch(HttpServletRequest request, HttpServletResponse resp
2     //省略非关键代码
3     mappedHandler = getHandler(processedRequest);
4     if (mappedHandler == null) {
5         noHandlerFound(processedRequest, response);
6         return;
7     }
8     //省略非关键代码
9 }
```

首先调用 `getHandler()` 获取当前请求的处理器，如果获取不到，则调用 `noHandlerFound()`：

 复制代码

```
1 protected void noHandlerFound(HttpServletRequest request, HttpServletResponse
2     if (this.throwExceptionIfNoHandlerFound) {
3         throw new NoHandlerFoundException(request.getMethod(), getRequestUri(req
4             new ServletServerHttpRequest(request).getHeaders());
5     }
6     else {
7         response.sendError(HttpServletResponse.SC_NOT_FOUND);
8     }
9 }
```

`noHandlerFound()` 的逻辑非常简单，如果 `throwExceptionIfNoHandlerFound` 属性为 `true`，则直接抛出 `NoHandlerFoundException` 异常，反之则会进一步获取到对应的请求处理器执行，并将执行结果返回给客户端。

到这，真相离我们非常近了，我们只需要将 `throwExceptionIfNoHandlerFound` 默认设置为 `true` 即可，这样就会抛出 `NoHandlerFoundException` 异常，从而被 `doDispatch()` 内的 `catch` 俘获。进而就像案例 1 介绍的一样，最终能够执行我们自定义的异常处理器 `MyExceptionHandler`。

于是，我们开始尝试，因为 `throwExceptionIfNoHandlerFound` 对应的 Spring 配置项为 `throw-exception-if-no-handler-found`，我们将其加入到 `application.properties` 配置文件中，设置其值为 `true`。

设置完毕后，重启服务并再次尝试，你会发现结果没有任何变化，这个问题也没有被解决。


实际上这里还存在另一个坑，在 Spring Web 的 `WebMvcAutoConfiguration` 类中，其默认添加的两个 `ResourceHandler`，一个是用来处理请求路径 `/webjars/**`，而另一个是 `/**`。

即便当前请求没有定义任何对应的请求处理器，`getHandler()` 也一定会获取到一个 `Handler` 来处理当前请求，因为第二个匹配 `/**` 路径的 `ResourceHandler` 决定了任何请求路径都会被其处理。`mappedHandler == null` 判断条件永远不会成立，显然就不可能走到 `noHandlerFound()`，那么就不会抛出 `NoHandlerFoundException` 异常，也无法被后续的异常处理器进一步处理。

下面让我们通过源码进一步了解这个默认被添加的 `ResourceHandler` 的详细逻辑。

首先我们来了解下 `ControllerAdvice` 是如何被 Spring 加载并对外暴露的。

同样是在 `WebMvcConfigurationSupport` 类中，被 `@Bean` 修饰的 `resourceHandlerMapping()`，它新建了 `ResourceHandlerRegistry` 类实例，并通过 `addResourceHandlers()` 将 `ResourceHandler` 注册到 `ResourceHandlerRegistry` 类实例中：

 复制代码

```
1 @Bean
2 @Nullable
3 public HandlerMapping resourceHandlerMapping(
4     @Qualifier("mvcUrlPathHelper") UrlPathHelper urlPathHelper,
5     @Qualifier("mvcPathMatcher") PathMatcher pathMatcher,
6     @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager con
7     @Qualifier("mvcConversionService") FormattingConversionService conversio
8     @Qualifier("mvcResourceUrlProvider") ResourceUrlProvider resourceUrlProv
9
10     Assert.state(this.applicationContext != null, "No ApplicationContext set");
11     Assert.state(this.servletContext != null, "No ServletContext set");
12
13     ResourceHandlerRegistry registry = new ResourceHandlerRegistry(this.applica
14         this.servletContext, contentNegotiationManager, urlPathHelper);
15     addResourceHandlers(registry);
16
17     AbstractHandlerMapping handlerMapping = registry.getHandlerMapping();
```

```
18     if (handlerMapping == null) {
19         return null;
20     }
21     handlerMapping.setPathMatcher(pathMatcher);
22     handlerMapping.setUrlPathHelper(urlPathHelper);
23     handlerMapping.setInterceptors(getInterceptors(conversionService, resourceU
24     handlerMapping.setCorsConfigurations(getCorsConfigurations());
25     return handlerMapping;
26 }
```

最终通过 `ResourceHandlerRegistry` 类实例中的 `getHandlerMapping()` 返回了 `SimpleUrlHandlerMapping` 实例，它装载了所有 `ResourceHandler` 的集合并注册到了 Spring 容器中：

[复制代码](#)

```
1 protected AbstractHandlerMapping getHandlerMapping() {
2     //省略非关键代码
3     Map<String, HttpRequestHandler> urlMap = new LinkedHashMap<>();
4     for (ResourceHandlerRegistration registration : this.registrations) {
5         for (String pathPattern : registration.getPathPatterns()) {
6             ResourceHttpRequestHandler handler = registration.getRequestHandler()
7             //省略非关键代码
8             urlMap.put(pathPattern, handler);
9         }
10    }
11    return new SimpleUrlHandlerMapping(urlMap, this.order);
12 }
```

我们查看以下调用栈截图：

```
addResourceHandlers:300, WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter
addResourceHandlers:95, WebMvcConfigurerComposite
addResourceHandlers:88, DelegatingWebMvcConfiguration
resourceHandlerMapping:537, WebMvcConfigurationSupport
```

可以了解到，当前方法中的 `addResourceHandlers()` 最终执行到了 `WebMvcAutoConfiguration` 类中的 `addResourceHandlers()`，通过这个方法，我们可以知道当前有哪些 `ResourceHandler` 的集合被注册到了 Spring 容器中：

[复制代码](#)

```
1 public void addResourceHandlers(ResourceHandlerRegistry registry) {
```

```
2     if (!this.resourceProperties.isAddMappings()) {
3         logger.debug("Default resource handling disabled");
4         return;
5     }
6     Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
7     CacheControl cacheControl = this.resourceProperties.getCache().getCacheControl();
8     if (!registry.hasMappingForPattern("/webjars/**")) {
9         customizeResourceHandlerRegistration(registry.addHandler("/webjars/**",
10             .addResourceLocations("classpath:/META-INF/resources/webjars/")
11             .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl)
12         ));
13     }
14     String staticPathPattern = this.mvcProperties.getStaticPathPattern();
15     if (!registry.hasMappingForPattern(staticPathPattern)) {
16         customizeResourceHandlerRegistration(registry.addHandler("/static",
17             .addResourceLocations(getResourceLocations(this.resourceProperties.getStaticLocations()))
18             .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl)
19         ));
20     }
```

从而验证我们一开始得出的结论，此处添加了两个 ResourceHandler，一个是用来处理请求路径 /webjars/**，而另一个是 /**。


这里你可以注意一下方法最开始的判断语句，如果 this.resourceProperties.isAddMappings() 为 false，那么会直接返回，后续的两个 ResourceHandler 也不会被添加。

[复制代码](#)

```
1     if (!this.resourceProperties.isAddMappings()) {
2         logger.debug("Default resource handling disabled");
3         return;
4     }
```

至此，有两个 ResourceHandler 被实例化且注册到了 Spring 容器中，一个处理路径为 /webjars/** 的请求，另一个处理路径为 /** 的请求。


同样，当第一次请求发生时，DispatcherServlet 中的 initHandlerMappings() 将会获取所有注册到 Spring 的 HandlerMapping 类型的实例，而 SimpleUrlHandlerMapping 恰好实现了 HandlerMapping 接口，这些 SimpleUrlHandlerMapping 类型的实例则会被写入到类成员变量 handlerMappings 中。

 复制代码

```
1 private void initHandlerMappings(ApplicationContext context) {
2     this.handlerMappings = null;
3     //省略非关键代码
4     if (this.detectAllHandlerMappings) {
5         // Find all HandlerMappings in the ApplicationContext, including ancesto
6         Map<String, HandlerMapping> matchingBeans =
7             BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMap
8         if (!matchingBeans.isEmpty()) {
9             this.handlerMappings = new ArrayList<>(matchingBeans.values());
10            // We keep HandlerMappings in sorted order.
11            AnnotationAwareOrderComparator.sort(this.handlerMappings);
12        }
13    }
14    //省略非关键代码
15 }
```

接着我们再来了解下被包装为 handlerMappings 的 ResourceHandler 是如何被 Spring 消费并处理的。

我们来回顾一下 DispatcherServlet 中的 doDispatch() 核心代码：

 复制代码

```
1 protected void doDispatch(HttpServletRequest request, HttpServletResponse resp
2     //省略非关键代码
3     mappedHandler = getHandler(processedRequest);
4     if (mappedHandler == null) {
5         noHandlerFound(processedRequest, response);
6         return;
7     }
8     //省略非关键代码
9 }
```

这里的 getHandler() 将会遍历成员变量 handlerMappings：

 复制代码

```
1 protected HandlerExecutionChain getHandler(HttpServletRequest request) throws
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     }
```



```
9     }  
10    return null;  
11 }
```

因为此处有一个 SimpleUrlHandlerMapping，它会拦截所有路径的请求：

```
{  
  3 = {SimpleUrlHandlerMapping@7980}  
  v f urlMap = {LinkedHashMap@8006} size = 2  
    > 3 "/webjars/**" -> {ResourceHttpRequestHandler@8023} "  
    > 3 "/**" -> {ResourceHttpRequestHandler@8025} "Resource
```

所以最终在 doDispatch() 的 getHandler() 将会获取到此 handler，从而 mappedHandler==null 条件不能得到满足，因而无法走到 noHandlerFound()，不会抛出 NoHandlerFoundException 异常，进而无法被后续的异常处理器进一步处理。


问题修正

那如何解决这个问题呢？还记得 WebMvcAutoConfiguration 类中 addResourceHandlers() 的前两行代码吗？如果 this.resourceProperties.isAddMappings() 为 false，那么此处直接返回，后续的两个 ResourceHandler 也不会被添加。

 复制代码

```
1 public void addResourceHandlers(ResourceHandlerRegistry registry) {  
2     if (!this.resourceProperties.isAddMappings()) {  
3         logger.debug("Default resource handling disabled");  
4         return;  
5     }  
6     //省略非关键代码  
7 }
```

其调用 ResourceProperties 中的 isAddMappings() 的代码如下：

 复制代码

```
1 public boolean isAddMappings() {  
2     return this.addMappings;  
3 }
```

到这，答案也就呼之欲出了，增加两个配置文件如下：

[复制代码](#)

```
1 spring.resources.add-mappings=false
2 spring.mvc.throwExceptionIfNoHandlerFound=true
```

修改 `MyExceptionHandler` 的 `@ExceptionHandler` 为 `NoHandlerFoundException` 即可：

[复制代码](#)

```
1 @ExceptionHandler({NoHandlerFoundException.class})
```

这个案例在真实的产线环境遇到的概率还是比较大的，知道如何解决是第一步，了解其内部原理则更为重要。而且当你进一步去研读代码后，你会发现这里的解决方案并不会只有这一种，而剩下的就留给你去探索了。

重点回顾

通过以上两个案例的介绍，相信你对 Spring MVC 的异常处理机制，已经有了进一步的了解，这里我们再次回顾下重点：

`DispatcherServlet` 类中的 `doDispatch()` 是整个 Servlet 处理的核心，它不仅实现了请求的分发，也提供了异常统一处理等一系列功能；

`WebMvcConfigurationSupport` 是 Spring Web 中非常核心的一个配置类，无论是异常处理器的包装注册（`HandlerExceptionResolver`），还是资源处理器的包装注册（`SimpleUrlHandlerMapping`），都是依靠这个类来完成的。

思考题

这节课的两个案例，在第一次发送请求的时候，会遍历对应的资源处理器和异常处理器，并注册到 `DispatcherServlet` 对应的类成员变量中，你知道它是如何被触发的吗？

期待你的思考，我们留言区见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

👍 赞 1 💡 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 15 | Spring Security 常见错误

下一篇 17 | 答疑现场：Spring Web 篇思考题合集

更多学习推荐

Java 面试必考 300 题

最新汇总

限时免费领取



精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。