



下载APP



## 23 | 答疑现场：Spring 补充篇思考题合集

2021-06-14 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)**讲述：傅健**

时长 00:37 大小 597.17K



你好，我是傅健。

欢迎来到第三次答疑现场，恭喜你，到了这，终点已近在咫尺。到今天为止，我们已经解决了 50 个线上问题，是不是很有成就感了？但要想把学习所得真正为你所用还要努力练习呀，这就像理论与实践之间永远有道鸿沟需要我们去跨越一样。那么接下来，话不多说，我们就开始逐一解答第三章的课后思考题了，有任何想法欢迎到留言区补充。

### 🔗 第 18 课



在案例 1 中使用 Spring Data Redis 时，我们提到了 StringRedisTemplate 和 RedisTemplate。那么它们是如何被创建起来的呢？

实际上, 当我们依赖 `spring-boot-starter` 时, 我们就间接依赖了 `spring-boot-autoconfigure`。

```
▼ org.springframework.boot:spring-boot-starter:2.4.5
  ► org.springframework.boot:spring-boot:2.4.5
  ► org.springframework.boot:spring-boot-autoconfigure:2.4.5
  ► org.springframework.boot:spring-boot-starter-logging:2.4.5
    jakarta.annotation:jakarta.annotation-api:1.3.5
    org.springframework:spring-core:5.3.6 (omitted for duplicate)
    org.yaml:snakeyaml:1.27
```

在这个 JAR 中, 存在下面这样的类, 即 `RedisAutoConfiguration`。

[复制代码](#)

```
1
2 @Configuration(proxyBeanMethods = false)
3 @ConditionalOnClass(RedisOperations.class)
4 @EnableConfigurationProperties(RedisProperties.class)
5 @Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.c
6 public class RedisAutoConfiguration {
7
8     @Bean
9     @ConditionalOnMissingBean(name = "redisTemplate")
10    @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
11    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory r
12        RedisTemplate<Object, Object> template = new RedisTemplate<>();
13        template.setConnectionFactory(redisConnectionFactory);
14        return template;
15    }
16
17    @Bean
18    @ConditionalOnMissingBean
19    @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
20    public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory redis
21        StringRedisTemplate template = new StringRedisTemplate();
22        template.setConnectionFactory(redisConnectionFactory);
23        return template;
24    }
25
26 }
```

从上述代码可以看出, 当存在 `RedisOperations` 这个类时, 就会创建 `StringRedisTemplate` 和 `RedisTemplate` 这两个 Bean。顺便说句, 这个

RedisOperations 是位于 Spring Data Redis 这个 JAR 中。

再回到开头，RedisAutoConfiguration 是如何被发现的呢？实际上，它被配置在

spring-boot-autoconfigure 的 META-INF/spring.factories 中，示例如下：

[复制代码](#)

```
1 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConf
3 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
4 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
5 org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfigu
6 org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
```

那么它是如何被加载进去的呢？我们的应用启动程序标记了 @SpringBootApplication，这个注解继承了下面这个注解：

[复制代码](#)

```
1 //省略其他非关键代码
2 @Import(AutoConfigurationImportSelector.class)
3 public @interface EnableAutoConfiguration {
4     //省略其他非关键代码
5 }
```

当它使用了 AutoConfigurationImportSelector 这个类，这个类就会导入在 META-INF/spring.factories 定义的 RedisAutoConfiguration。那么 import 动作是什么时候执行的呢？实际上是在启动应用程序时触发的，调用堆栈信息如下：

```
process:437, AutoConfigurationImportSelector$AutoConfigurationGroup (org.springframework.boot.autoconfigure)
getImports:879, ConfigurationClassParser$DeferredImportSelectorGrouping (org.springframework.context.annotation)
processGroupImports:809, ConfigurationClassParser$DeferredImportSelectorGroupingHandler (org.springframework.context.annotation)
process:780, ConfigurationClassParser$DeferredImportSelectorHandler (org.springframework.context.annotation)
parse:193, ConfigurationClassParser (org.springframework.context.annotation)
processConfigBeanDefinitions:331, ConfigurationClassPostProcessor (org.springframework.context.annotation)
postProcessBeanDefinitionRegistry:247, ConfigurationClassPostProcessor (org.springframework.context.annotation)
invokeBeanDefinitionRegistryPostProcessors:311, PostProcessorRegistrationDelegate (org.springframework.context.support)
invokeBeanFactoryPostProcessors:112, PostProcessorRegistrationDelegate (org.springframework.context.support)
invokeBeanFactoryPostProcessors:746, AbstractApplicationContext (org.springframework.context.support)
refresh:564, AbstractApplicationContext (org.springframework.context.support)
refresh:782, SpringApplication (org.springframework.boot)
refresh:774, SpringApplication (org.springframework.boot)
refreshContext:439, SpringApplication (org.springframework.boot)
run:339, SpringApplication (org.springframework.boot)
run:1340, SpringApplication (org.springframework.boot)
run:1329, SpringApplication (org.springframework.boot)
```

结合上面的堆栈和相关源码，我们不妨可以总结下 RedisTemplate 被创建的过程。

当 Spring 启动时，会通过 ConfigurationClassPostProcessor 尝试处理所有标记 @Configuration 的类，具体到每个配置类的处理是通过 ConfigurationClassParser 来完成的。

在这个完成过程中，它会使用 ConfigurationClassParser.DeferredImportSelectorHandler 来完成对 Import 的处理。AutoConfigurationImportSelector 就是其中一种 Import，它被 @EnableAutoConfiguration 这个注解间接引用。它会加载"META-INF/spring.factories"中定义的 RedisAutoConfiguration，此时我们就会发现 StringRedisTemplate 和 RedisTemplate 这两个 Bean 了。

## 🔗 第 19 课

RuntimeException 是 Exception 的子类，如果用 rollbackFor=Exception.class，那对 RuntimeException 也会生效。如果我们需要对 Exception 执行回滚操作，但对于 RuntimeException 不执行回滚操作，应该怎么做呢？

我们可以同时为 @Transactional 指定 rollbackFor 和 noRollbackFor 属性，具体代码示例如下：

📄 复制代码

```
1 @Transactional(rollbackFor = Exception.class, noRollbackFor = RuntimeException
2 public void doSaveStudent(Student student) throws Exception {
3     studentMapper.saveStudent(student);
4     if (student.getRealname().equals("小明")) {
5         throw new RuntimeException("该用户已存在");
6     }
7 }
```

## 🔗 第 20 课

结合案例 2，请你思考这样一个问题：在这个案例中，我们在 CardService 类方法上声明了这样的事务传播属性，@Transactional(propagation = Propagation.REQUIRES\_NEW)，如果使用 Spring 的默认声明行不行，为什么？



答案是不行。我们前面说过，Spring 默认的事务传播类型是 REQUIRED，在有外部事务的情况下，内部事务则会加入原有的事务。如果我们声明成 REQUIRED，当我们要操作 card 数据的时候，持有的依然还会是原来的 DataSource。

## 🔗 第 21 课

当我们比较案例 1 和案例 2，你会发现不管使用的是查询（Query）参数还是表单（Form）参数，我们的接口定义并没有什么变化，风格如下：

📄 复制代码

```
1 @RestController
2 public class HelloWorldController {
3     @RequestMapping(path = "hi", method = RequestMethod.GET)
4     public String hi(@RequestParam("para1") String para1){
5         return "helloworld:" + para1;
6     };
7
8 }
```

那是不是 @RequestParam 本身就能处理这两种数据呢？

不考虑实现原理，如果我们仔细看下 @RequestParam 的 API 文档，你就会发现 @RequestParam 不仅能处理表单参数，也能处理查询参数。API 文档如下：

In Spring MVC, "request parameters" map to query parameters, form data, and parts in multipart requests. This is because the Servlet API combines query parameters and form data into a single map called "parameters", and that includes automatic parsing of the request body.

稍微深入一点的话，我们还可以从源码上看看具体实现。

不管是使用 Query 参数还是用 Form 参数来访问，对于案例程序而言，解析的关键逻辑都是类似的，都是通过下面的调用栈完成参数的解析：

Frames	Threads
✓ "http-nio-8080-exec-3"@6,050 in group "main": RUNNING	
parseParameters:3157, Request (org.apache.catalina.connector) getParameterValues:1177, Request (org.apache.catalina.connector) getParameterValues:424, RequestFacade (org.apache.catalina.connector) getParameterValues:153, ServletWebRequest (org.springframework.web.context.request) <b>resolveName:181, RequestParamMethodArgumentResolver (org.springframework.web.method.annotation)</b> resolveArgument:108, AbstractNamedValueMethodArgumentResolver (org.springframework.web.method.annotation) resolveArgument:121, HandlerMethodArgumentResolverComposite (org.springframework.web.method.support) getMethodArgumentValues:167, InvocableHandlerMethod (org.springframework.web.method.support) invokeForRequest:134, InvocableHandlerMethod (org.springframework.web.method.support) invokeAndHandle:106, ServletInvocableHandlerMethod (org.springframework.web.servlet.mvc.method.annotation) invokeHandlerMethod:888, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation) handleInternal:793, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation) handle:87, AbstractHandlerMethodAdapter (org.springframework.web.servlet.mvc.method) doDispatch:1040, DispatcherServlet (org.springframework.web.servlet) doService:943, DispatcherServlet (org.springframework.web.servlet)	

这里可以看出，负责解析的都是 RequestParamMethodArgumentResolver，解析最后的调用也都是一样的方法。在 org.apache.catalina.connector.Request#parseParameters 这个方法中，对于 Form 的解析是这样的：

[复制代码](#)

```

1  if (!("application/x-www-form-urlencoded".equals(contentType))) {
2      success = true;
3      return;
4  }
5
6  //走到这里，说明是 Form: "application/x-www-form-urlencoded"
7  int len = getContentLength();
8
9  if (len > 0) {
10     int maxPostSize = connector.getMaxPostSize();
11     if ((maxPostSize >= 0) && (len > maxPostSize)) {
12         //省略非关键代码
13     }
14     byte[] formData = null;
15     if (len < CACHED_POST_LEN) {
16         if (postData == null) {
17             postData = new byte[CACHED_POST_LEN];
18         }
19         formData = postData;
20     } else {
21         formData = new byte[len];
22     }
23     try {
24         if (readPostBody(formData, len) != len) {
25             parameters.setParseFailedReason(FailReason.REQUEST_BODY_INCOMPLETE
26             return;
27         }
28     } catch (IOException e) {

```

```
29         //省略非关键代码
30     }
31     //把 Form 数据添加到 parameter 里面去
32     parameters.processParameters(formData, 0, len);
```

Form 的数据最终存储在 Parameters#paramHashValues 中。

而对于查询参数的处理，同样是在 org.apache.catalina.connector.Request#parseParameters 中，不过处理它的代码行在 Form 前面一些，关键调用代码行如下：

```
1 parameters.handleQueryParameters();
```

[复制代码](#)

最终它也是通过 org.apache.tomcat.util.http.Parameters#processParameters 来完成数据的添加。自然，它存储的位置也是 Parameters#paramHashValues 中。

综上所述，虽然使用的是一个固定的注解 @RequestParam，但是它能处理表单和查询参数，因为它们都会存储在同一个位置：Parameters#paramHashValues。

## 🔗 第 22 课


在案例 1 中，我们解释了为什么测试程序加载不到 spring.xml 文件，根源在于当使用下面的语句加载文件时，它们是采用不同的 Resource 形式来加载的：

```
1 @ImportResource(locations = {"spring.xml"})
```

[复制代码](#)


具体而言，应用程序加载使用的是 ClassPathResource，测试加载使用的是 ServletContextResource，那么这是怎么造成的呢？

实际上，以何种类型的 Resource 加载是由 DefaultResourceLoader#getResource 来决定的：

 复制代码


```
1 @Override
2 public Resource getResource(String location) {
3     //省略非关键代码
4     if (location.startsWith("/")) {
5         return getResourceByPath(location);
6     }
7     else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
8         return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.len
9     )
10    }
11    else {
12        try {
13            // Try to parse the location as a URL...
14            URL url = new URL(location);
15            return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new
16        }
17        catch (MalformedURLException ex) {
18            // No URL -> resolve as resource path.
19            return getResourceByPath(location);
20        }
21    }
```

结合上述代码，你可以看出，当使用下面语句时：

 复制代码

```
1 @ImportResource(locations = {"classpath:spring.xml"})
```

走入的分支是：

 复制代码

```
1 //CLASSPATH_URL_PREFIX:classpath
2 else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
3     return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.len
4 }
```

即创建的是 ClassPathResource。

而当使用下面语句时：

 复制代码



```
1 @ImportResource(locations = {"spring.xml"})
```

走入的分支是：

[复制代码](#)

```
1     try {
2         // 按 URL 加载
3         URL url = new URL(location);
4         return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new
5     }
6     catch (MalformedURLException ex) {
7         // 按路径加载
8         return getResourceByPath(location);
9     }
```

先尝试按 URL 加载，很明显这里会失败，因为字符串 `spring.xml` 并非一个 URL。随后使用 `getResourceByPath()` 来加载，它会执行到下面的 `WebApplicationContextResourceLoader#getResourceByPath()`：

[复制代码](#)

```
1 private static class WebApplicationContextResourceLoader extends ClassLoaderF
2     private final WebApplicationContext applicationContext;
3     //省略非关键代码
4     protected Resource getResourceByPath(String path) {
5         return (Resource)(this.applicationContext.getServletContext() != null
6     }
7 }
```

可以看出，这个时候其实已经和 `ApplicationContext` 息息相关了。在我们的案例中，最终返回的是 `ServletContextResource`。

相信看到这里，你就能明白为什么一个小小的改动会导致生成的 `Resource` 不同了。无非还是因为你定义了不同的格式，不同的格式创建的资源不同，加载逻辑也不同。至于后续是如何加载的，你可以回看全文。

以上就是这次答疑的全部内容，我们下节课再见！

分享给需要的人，Ta订阅后你可得 **20元** 现金奖励

 赞 4

 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22 | Spring Test 常见错误

下一篇知识回顾 | 系统梳理Spring编程错误根源

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。