



22 | Spring Test 常见错误

2021-06-11 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)



讲述：傅健

时长 10:35 大小 9.70M



你好，我是傅健。

前面我们介绍了许多 Spring 常用知识点上的常见应用错误。当然或许这些所谓的常用，你仍然没有使用，例如对于 Spring Data 的使用，有的项目确实用不到。那么这一讲，我们聊聊 Spring Test，相信你肯定绕不开对它的使用，除非你不使用 Spring 来开发程序，或者你使用了 Spring 但是你不写测试。但话说回来，后者的情况就算你想如此，你的老板也不会同意吧。

那么在 Spring Test 的应用上，有哪些常见错误呢？这里我给你梳理了两个典型，闲话叙，我们直接进入这一讲的学习。



案例 1：资源文件扫描不到

首先，我们来写一个 HelloWorld 版的 Spring Boot 程序以做测试备用。

先来定义一个 Controller：

[复制代码](#)

```
1 @RestController
2 public class HelloController {
3
4     @Autowired
5     HelloWorldService helloWorldService;
6
7     @RequestMapping(path = "hi", method = RequestMethod.GET)
8     public String hi() throws Exception{
9         return helloWorldService.toString() ;
10    };
11
12 }
```

当访问 <http://localhost:8080/hi> 时，上述接口会打印自动注入的 HelloWorldService 类型的 Bean。而对于这个 Bean 的定义，我们这里使用配置文件的方式进行。

1. 定义 HelloWorldService，具体到 HelloWorldService 的实现并非本讲的重点，所以我们可以简单实现如下：

[复制代码](#)


```
1 public class HelloWorldService {
2 }
```

2. 定义一个 spring.xml，在这个 XML 中定义 HelloWorldService 的 Bean，并把这个 spring.xml 文件放置在 /src/main/resources 中：

[复制代码](#)


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans http://
5       <bean id="helloWorldService" class="com.spring.puzzle.others.test.example1
6       </bean>
7 </beans>
```

3. 定义一个 Configuration 引入上述定义 XML，具体实现方式如下：

 复制代码

```
1 @Configuration
2 @ImportResource(locations = {"spring.xml"})
3 public class Config {
4 }
```

完成上述步骤后，我们就可以使用 main() 启动起来。测试这个接口，一切符合预期。那么接下来，我们来写一个测试：

 复制代码

```
1 @SpringBootTest()
2 class ApplicationTests {
3
4     @Autowired
5     public HelloController helloController;
6
7     @Test
8     public void testController() throws Exception {
9         String response = helloController.hi();
10        Assert.notNull(response, "not null");
11    }
12
13 }
```

当我们运行上述测试的时候，会发现测试失败了，报错如下：

```
Caused by: java.io.FileNotFoundException Create breakpoint : Could not open ServletContext resource [/spring.xml]
    at org.springframework.web.context.support.ServletContextResource.getInputStream(ServletContextResource.java:159) ~[spring-web-5.3.10.jar:5.3.10]
    at org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:331) ~[spring-beans-5.3.10.jar:5.3.10]
    ... 84 common frames omitted
```

为什么单独运行应用程序没有问题，但是运行测试就不行了呢？我们需要研究一下 Spring 的源码，来寻找答案。

案例解析

在了解这个问题的根本原因之前，我们先从调试的角度来对比下启动程序和测试加载 spring.xml 的不同之处。

1. 启动程序加载 spring.xml

首先看下调用栈：

```

✓ "restartedMain"@1,620 in group "main": RUNNING
getInputStream:170, ClassPathResource (org.springframework.core.io)
loadBeanDefinitions:331, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
loadBeanDefinitions:305, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
loadBeanDefinitions:188, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
loadBeanDefinitions:224, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
loadBeanDefinitions:195, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)

```

可以看出，它最终以 ClassPathResource 形式来加载，这个资源的情况如下：

```

▼ this = {DefaultResourceLoader$ClassPathContextResource@4791} "class path resource [spring.xml]"
> f path = "spring.xml"
> f classLoader = {RestartClassLoader@4800}
  f clazz = null

```

而具体到加载实现，它使用的是 ClassPathResource#getInputStream 来加载 spring.xml 文件：

```

public InputStream getInputStream() throws IOException {
    InputStream is;
    if (this.clazz != null) {
        is = this.clazz.getResourceAsStream(this.path);    clazz: null
    } else if (this.classLoader != null) {
        is = this.classLoader.getResourceAsStream(this.path);    path: "spring.xml"    classLoader: RestartClassLoader@4800
    } else {
        is = ClassLoader.getSystemResourceAsStream(this.path);
    }

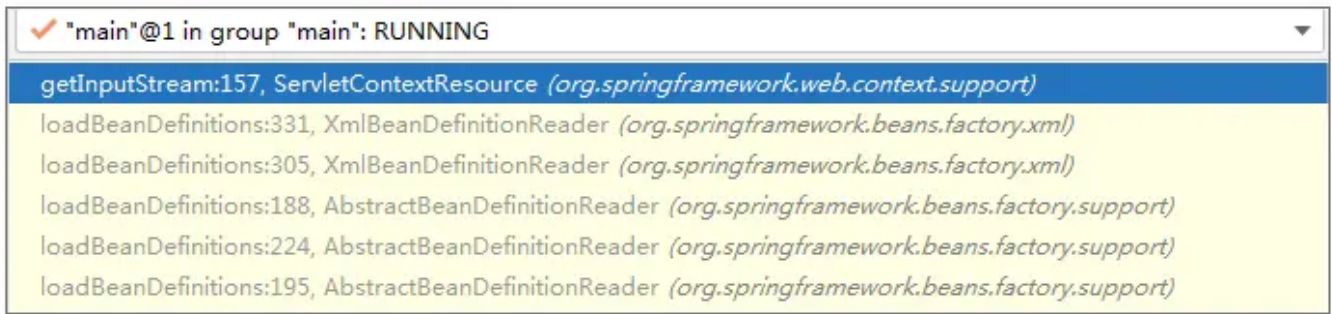
    if (is == null) {
        throw new FileNotFoundException(this.getDescription() + " cannot be opened because it does not exist");
    } else {
        return is;
    }
}

```

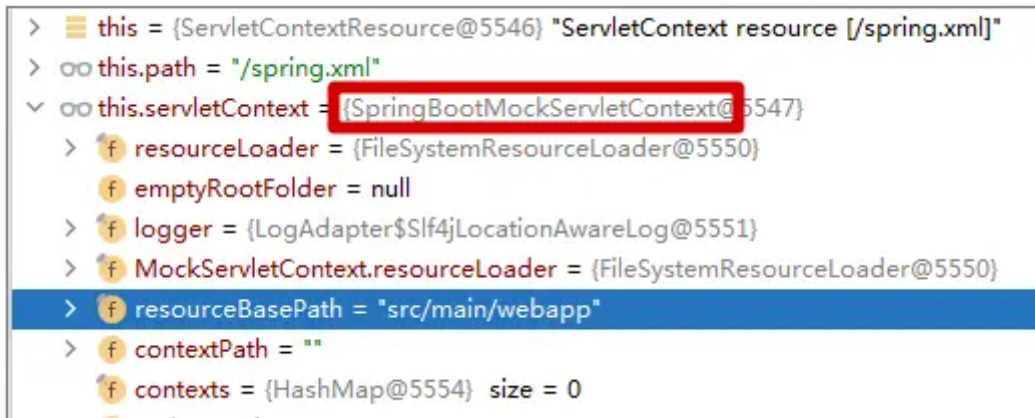
从上述调用及代码实现，可以看出最终是可以加载成功的。

2. 测试加载 spring.xml

首先看下调用栈：



可以看出它是按 ServletContextResource 来加载的，这个资源的情况如下：



具体到实现，它最终使用的是 MockServletContext#getResourceAsStream 来加载文件：


复制代码

```

1 @Nullable
2 public InputStream getResourceAsStream(String path) {
3     String resourceLocation = this.getResourceLocation(path);
4     Resource resource = null;
5
6     try {
7         resource = this.resourceLoader.getResource(resourceLocation);
8         return !resource.exists() ? null : resource.getInputStream();
9     } catch (IOException | InvalidPathException var5) {
10         if (this.logger.isWarnEnabled()) {
11             this.logger.warn("Could not open InputStream for resource " + (res
12         )
13
14         return null;
15     }
16 }

```

你可以继续跟踪它的加载位置相关代码，即 getResourceLocation()：

 复制代码

```
1 protected String getResourceLocation(String path) {
2     if (!path.startsWith("/")) {
3         path = "/" + path;
4     }
5     //加上前缀：/src/main/resources
6     String resourceLocation = this.getResourceBasePathLocation(path);
7     if (this.exists(resourceLocation)) {
8         return resourceLocation;
9     } else {
10        //{"classpath:META-INF/resources", "classpath:resources", "classpath:s
11        String[] var3 = SPRING_BOOT_RESOURCE_LOCATIONS;
12        int var4 = var3.length;
13
14        for(int var5 = 0; var5 < var4; ++var5) {
15            String prefix = var3[var5];
16            resourceLocation = prefix + path;
17            if (this.exists(resourceLocation)) {
18                return resourceLocation;
19            }
20        }
21
22        return super.getResourceLocation(path);
23    }
24 }
```

你会发现，它尝试从下面的一些位置进行加载：

 复制代码

```
1 classpath:META-INF/resources
2 classpath:resources
3 classpath:static
4 classpath:public
5 src/main/webapp
```

如果你仔细看这些目录，你还会发现，这些目录都没有 spring.xml。或许你认为源文件 src/main/resource 下面不是有一个 spring.xml 么？那上述位置中的 classpath:resources 不就能加载了么？

那你肯定是忽略了一点：当程序运行起来后，src/main/resource 下的文件最终是不带什么 resource 的。关于这点，你可以直接查看编译后的目录（本地编译后是 target\classes 目录），示例如下：

名称	修改日期	类型	大小
com	2021/5/29 9:42	文件夹	
META-INF	2021/5/29 7:44	文件夹	
application.properties	2021/5/29 7:49	PROPERTIES 文件	1 KB
spring.xml	2021/5/23 7:57	XML 文档	1 KB

所以，最终我们在所有的目录中都找不到 spring.xml，并且会报错提示加载不了文件。报错的地方位于 ServletContextResource#getInputStream 中：

[复制代码](#)

```
1 @Override
2 public InputStream getInputStream() throws IOException {
3     InputStream is = this.servletContext.getResourceAsStream(this.path);
4     if (is == null) {
5         throw new FileNotFoundException("Could not open " + getDescription());
6     }
7     return is;
8 }
```

问题修正

从上述案例解析中，我们了解到了报错的原因，那么如何修正这个问题？这里我们可以采用两种方式。

1. 在加载目录上放置 spring.xml

就本案例而言，加载目录有很多，所以修正方式也不少，我们可以建立一个 src/main/webapp，然后把 spring.xml 复制一份进去就可以了。也可以在 /src/main/resources 下面再建立一个 resources 目录，然后放置进去也可以。

2. 在 @ImportResource 使用 classpath 加载方式

[复制代码](#)

```
1 @Configuration
2 //@ImportResource(locations = {"spring.xml"})
3 @ImportResource(locations = {"classpath:spring.xml"})
4 public class Config {
5 }
```

这里，我们可以通过 Spring 的官方文档简单了解下不同加载方式的区别，参考 <https://docs.spring.io/spring-framework/docs/2.5.x/reference/resources.html>：

Table 4.1. Resource strings

Prefix	Example	Explanation
classpath:	classpath:com/myapp/config.xml	Loaded from the classpath.
file:	file:/data/config.xml	Loaded as a URL, from the filesystem. ^[a]
http:	http://myserver/logo.png	Loaded as a URL.
(none)	/data/config.xml	Depends on the underlying ApplicationContext.

^[a] But see also the section entitled [Section 4.7.3, "FileSystemResource caveats"](#).

很明显，我们一般都不会使用本案例的方式（即 `locations = {"spring.xml"}`，无任何“前缀”的方式），毕竟它已经依赖于使用的 `ApplicationContext`。而 `classpath` 更为普适些，而一旦你按上述方式修正后，你会发现它加载的资源已经不再是 `ServletContextResource`，而是和应用程序一样的 `ClassPathResource`，这样自然可以加载到了。

所以说到底，表面上看，这个问题是关于测试的案例，但是实际上是 `ImportResource` 的使用问题。不过通过这个案例，你也会明白，很多用法真的只能在某个特定场合才能工作起来，你只是比较幸运而已。

案例 2：容易出错的 Mock

接下来，我们再来看一个非功能性的错误案例。有时候，我们会发现 Spring Test 运行起来非常缓慢，寻根溯源之后，你会发现主要是因为很多测试都启动了 Spring Context，示例如下：


```

      .- - - - -      _      - - - - -
    /\ /  _ _ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \
  ( ( ) \ _ _ | ' _ | ' _ | ' _ \ / _ ' _ \ \ \ \
  \ / _ _ ) | | ) | | | | | | | ( | | ) ) ) )
    ' | _ _ | . _ | | | | | | | \ _ , | / / / /
  =====|_|=====|_|_/=//_/_/_/_/
:: Spring Boot ::      (v2.2.2.RELEASE)

```

```

2021-05-30 07:59:20.992 INFO 7396 --- [
2021-05-30 07:59:20.993 INFO 7396 --- [
2021-05-30 07:59:24.025 INFO 7396 --- [
2021-05-30 07:59:24.692 INFO 7396 --- [

```

Using generated security password: 6ab8e7a8-8937-4a


```

      .- - - - -      _      - - - - -
    /\ /  _ _ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \
  ( ( ) \ _ _ | ' _ | ' _ | ' _ \ / _ ' _ \ \ \ \
  \ / _ _ ) | | ) | | | | | | | ( | | ) ) ) )
    ' | _ _ | . _ | | | | | | | \ _ , | / / / /
  =====|_|=====|_|_/=//_/_/_/_/
:: Spring Boot ::      (v2.2.2.RELEASE)

```

那么为什么有的测试会多次启动 Spring Context？在具体解析这个问题之前，我们先模拟写一个案例来复现这个问题。

我们先在 Spring Boot 程序中写几个被测试类：

 复制代码

```

1 @Service
2 public class ServiceOne {
3 }
4 @Service
5 public class ServiceTwo {
6 }
7

```

然后分别写出对应的测试类：

 复制代码

```

1 @SpringBootTest()
2 class ServiceOneTests {


```

```
3
4     @MockBean
5     ServiceOne serviceOne;
6
7     @Test
8     public void test(){
9         System.out.println(serviceOne);
10    }
11 }
12
13 @SpringBootTest()
14 class ServiceTwoTests {
15     @MockBean
16     ServiceTwo serviceTwo;
17     @Test
18     public void test(){
19         System.out.println(serviceTwo);
20     }
21 }
```

在上述测试类中，我们都使用了 @MockBean。写完这些程序，批量运行测试，你会发现 Spring Context 果然会被运行多次。那么如何理解这个现象，是错误还是符合预期？接下来我们具体来解析下。

案例解析

当我们运行一个测试的时候，正常情况是不会重新创建一个 Spring Context 的。这是因为 Spring Test 使用了 Context 的缓存以避免重复创建 Context。那么这个缓存是怎么维护的呢？我们可以通过 DefaultCacheAwareContextLoaderDelegate#loadContext 来看下 Context 的获取和缓存逻辑：

 复制代码

```
1 public ApplicationContext loadContext(MergedContextConfiguration mergedContext
2     synchronized(this.contextCache) {
3         ApplicationContext context = this.contextCache.get(mergedContextConfig
4         if (context == null) {
5             try {
6                 context = this.loadContextInternal(mergedContextConfiguration)
7                 //省略非关键代码
8                 this.contextCache.put(mergedContextConfiguration, context);
9             } catch (Exception var6) {
10                //省略非关键代码
11            }
12        } else if (logger.isDebugEnabled()) {
13            //省略非关键代码
```

```
14         }
15
16         this.contextCache.logStatistics();
17         return context;
18     }
19 }
```

从上述代码可以看出，缓存的 Key 是 MergedContextConfiguration。所以一个测试要不要启动一个新的 Context，就取决于根据这个测试 Class 构建的 MergedContextConfiguration 是否相同。而是否相同取决于它的 hashCode() 实现：

[复制代码](#)

```
1 public int hashCode() {
2     int result = Arrays.hashCode(this.locations);
3     result = 31 * result + Arrays.hashCode(this.classes);
4     result = 31 * result + this.contextInitializerClasses.hashCode();
5     result = 31 * result + Arrays.hashCode(this.activeProfiles);
6     result = 31 * result + Arrays.hashCode(this.propertySourceLocations);
7     result = 31 * result + Arrays.hashCode(this.propertySourceProperties);
8     result = 31 * result + this.contextCustomizers.hashCode();
9     result = 31 * result + (this.parent != null ? this.parent.hashCode() : 0);
10    result = 31 * result + nullSafeClassName(this.contextLoader).hashCode();
11    return result;
12 }
```

从上述方法，你可以看出只要上述元素中的任何一个不同都会导致一个 Context 会重新创建出来。关于这个缓存机制和 Key 的关键因素你可以参考 Spring 的官方文档，也有所提及，这里我直接给出了链接，你可以对照着去阅读。

点击获取：<https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#testcontext-ctx-management-caching>

现在回到本案例，为什么会创建一个新的 Context 而不是复用？根源在于两个测试的 contextCustomizers 这个元素的不同。如果你不信的话，你可以调试并对比下。

ServiceOneTests 的 MergedContextConfiguration 示例如下：

```

    mergedConfig = {MergedContextConfiguration@2326} "[MergedContextConfiguration@1ba9117e testClass = ServiceOneTests, locations = '{}', class
    > testClass = {Class@1407} "class com.spring.puzzle.others.test.example2.ServiceOneTests" ... Navigate
    > locations = {String[0]@2330} []
    > classes = {Class[0]@2331}
    > contextInitializerClasses = {Collections$UnmodifiableSet@2332} size = 0
    > activeProfiles = {String[0]@2330} []
    > propertySourceLocations = {String[0]@2333} []
    > propertySourceProperties = {String[0]@2334} []
    > contextCustomizers = {Collections$UnmodifiableSet@2335} size = 6
    > 0 = {ExcludeFilterContextCustomizer@2340}
    > 1 = {DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectContextCustomizer@2341}
    > 2 = {MockitoContextCustomizer@2342}
    > definitions = {LinkedHashSet@2346} size = 1
    > 0 = {MockDefinition@2348} "[MockDefinition@732c2a62 name = "", typeToMock = com.spring.puzzle.others.test.example2.ServiceOne,
    > 3 = {TestRestTemplateContextCustomizer@2343}
    > 4 = {PropertyMappingContextCustomizer@2344}
    > 5 = {WebDriverContextCustomizerFactory$Customizer@2345}
    > contextLoader = {SpringBootTestLoader@2336}
    > cacheAwareContextLoaderDelegate = {DefaultCacheAwareContextLoaderDelegate@2337}
    > parent = null

```

ServiceTwoTests 的 MergedContextConfiguration 示例如下：

```

    mergedConfig = {MergedContextConfiguration@11020} "[MergedContextConfiguration@62b0bf85 testClass = ServiceTwoTests, locations = '{}', class
    > testClass = {Class@1490} "class com.spring.puzzle.others.test.example2.ServiceTwoTests" ... Navigate
    > locations = {String[0]@2330} []
    > classes = {Class[0]@11022}
    > contextInitializerClasses = {Collections$UnmodifiableSet@11023} size = 0
    > activeProfiles = {String[0]@2330} []
    > propertySourceLocations = {String[0]@2333} []
    > propertySourceProperties = {String[0]@2334} []
    > contextCustomizers = {Collections$UnmodifiableSet@11024} size = 6
    > 0 = {ExcludeFilterContextCustomizer@11030}
    > 1 = {DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectContextCustomizer@11031}
    > 2 = {MockitoContextCustomizer@11032}
    > definitions = {LinkedHashSet@11036} size = 1
    > 0 = {MockDefinition@11038} "[MockDefinition@1fcaea93 name = "", typeToMock = com.spring.puzzle.others.test.example2.ServiceTwo,
    > 3 = {TestRestTemplateContextCustomizer@11033}
    > 4 = {PropertyMappingContextCustomizer@11034}
    > 5 = {WebDriverContextCustomizerFactory$Customizer@11035}
    > contextLoader = {SpringBootTestLoader@11025}
    > cacheAwareContextLoaderDelegate = {DefaultCacheAwareContextLoaderDelegate@11026}
    > parent = null

```

很明显，MergedContextConfiguration（即 Context Cache 的 Key）的 ContextCustomizer 是不同的，所以 Context 没有共享起来。而追溯到 ContextCustomizer 的创建，我们可以具体来看下。

当我们运行一个测试（testClass）时，我们会使用 MockitoContextCustomizerFactory#createContextCustomizer 来创建一个 ContextCustomizer，代码示例如下：

复制代码

```
1 class MockitoContextCustomizerFactory implements ContextCustomizerFactory {
```

```
2 MockitoContextCustomizerFactory() {
3 }
4
5 public ContextCustomizer createContextCustomizer(Class<?> testClass, List<
6     DefinitionsParser parser = new DefinitionsParser();
7     parser.parse(testClass);
8     return new MockitoContextCustomizer(parser.getDefinitions());
9 }
10 }
```

创建的过程是由 DefinitionsParser 来解析这个测试 Class (例如案例中的 ServiceOneTests) , 如果这个测试 Class 中包含了 MockBean 或者 SpyBean 标记的情况, 则将对对应标记的情况转化为 MockDefinition , 最终添加到 ContextCustomizer 中。解析的过程参考 DefinitionsParser#parse :

[复制代码](#)


```
1 void parse(Class<?> source) {
2     this.parseElement(source);
3     ReflectionUtils.doWithFields(source, this::parseElement);
4 }
5
6 private void parseElement(AnnotatedElement element) {
7     MergedAnnotations annotations = MergedAnnotations.from(element, SearchStra
8 //MockBean 处理     annotations.stream(MockBean.class).map(MergedAnnotation::syn
9         this.parseMockBeanAnnotation(annotation, element);
10    });
11 //SpyBean 处理     annotations.stream(SpyBean.class).map(MergedAnnotation::synth
12         this.parseSpyBeanAnnotation(annotation, element);
13    });
14 }
15
16 private void parseMockBeanAnnotation(MockBean annotation, AnnotatedElement ele
17     Set<ResolvableType> typesToMock = this.getOrDeduceTypes(element, annotatio
18 //省略非关键代码
19     Iterator var4 = typesToMock.iterator();
20     while(var4.hasNext()) {
21         ResolvableType typeToMock = (ResolvableType)var4.next();
22         MockDefinition definition = new MockDefinition(annotation.name(), type
23 //添加到 DefinitionsParser#definitions
24         this.addDefinition(element, definition, "mock");
25     }
26 }
```

那说了这么多, Spring Context 重新创建的根本原因还是在于使用了 @MockBean 且不同, 从而导致构建的 MergedContextConfiguration 不同, 而

MergedContextConfiguration 正是作为 Cache 的 Key，Key 不同，Context 不能被复用，所以被重新创建了。这就是为什么在案例介绍部分，你会看到多次 Spring Context 的启动过程。而正因为“重启”，测试速度变缓慢了。

问题修正

到这，你会发现其实这种缓慢的根源是使用了 @MockBean 带来的一个正常现象。但是假设你非要去提速下，那么你可以尝试使用 Mockito 去手工实现类似的功能。当然你也可以尝试使用下面的方式来解决，即把相关的 MockBean 都定义到一个地方去。例如针对本案例，修正方案如下：

 复制代码

```
1 public class ServiceTests {
2     @MockBean
3     ServiceOne serviceOne;
4     @MockBean
5     ServiceTwo serviceTwo;
6
7 }
8
9 @SpringBootTest()
10 class ServiceOneTests extends ServiceTests{
11
12     @Test
13     public void test(){
14         System.out.println(serviceOne);
15     }
16
17 }
18
19 @SpringBootTest()
20 class ServiceTwoTests extends ServiceTests{
21     @Test
22     public void test(){
23         System.out.println(serviceTwo);
24     }
25 }
```

重新运行测试，你会发现 Context 只会被创建一次，速度也有所提升了。相信，你也明白这么改能工作的原因了，现在每个测试对应的 Context 缓存 Key 已经相同了。

重点回顾

通过以上两个案例，相信你对 Spring Test 已经有了进一步的了解，最后总结下重点。

在使用 Spring Test 的时候，一定要注意资源文件的加载方式是否正确。例如，你使用的是绝对路径，形式如下：

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

[复制代码](#)

那么它可能在不同的场合实现不同，不一定能加载到你想要的文件，所以我并不推荐你在使用 @ImportResource 时，使用绝对路径指定资源。

另外，@MockBean 可能会导致 Spring Context 反复新建，从而让测试变得缓慢，从根源上看，这是属于正常现象。不过你一定要意识到这点，否则，你可能会遇到各种难以理解的现象。

而假设你需要加速，你可以尝试多种方法，例如，你可以把依赖 Mock 的 Bean 声明在一个统一的地方。当然，你要格外注意这样是否还能满足你的测试需求。

思考题

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？

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

© 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 21 | Spring Rest Template 常见错误

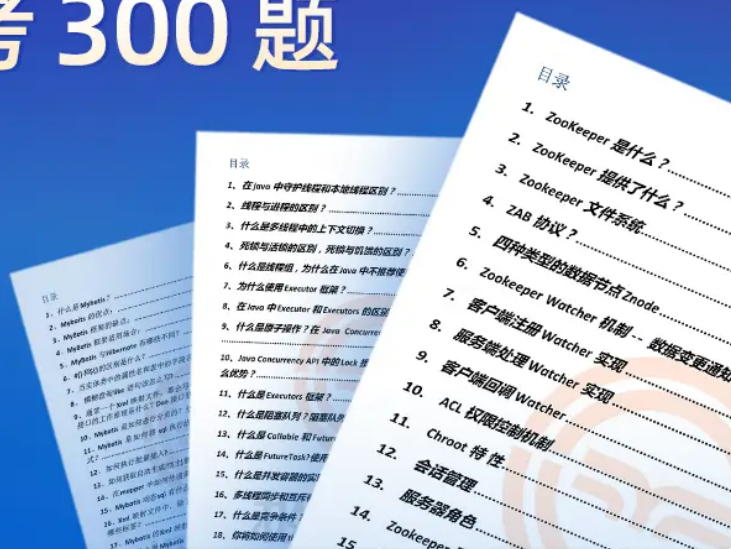
下一篇 23 | 答疑现场：Spring 补充篇思考题合集

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。