



下载APP



05 | Spring AOP常见错误（上）

2021-04-30 傅健

Spring编程常见错误50例

[进入课程 >](#)**讲述：傅健**

时长 17:22 大小 15.91M



你好，我是傅健。这节课开始，我们聊聊 Spring AOP 使用中常遇到的一些问题。

Spring AOP 是 Spring 中除了依赖注入外（DI）最为核心的功能，顾名思义，AOP 即 Aspect Oriented Programming，翻译为面向切面编程。

而 Spring AOP 则利用 CGLib 和 JDK 动态代理等方式来实现运行期动态方法增强，其目的是将与业务无关的代码单独抽离出来，使其逻辑不再与业务代码耦合，从而降低系统的耦合性，提高程序的可重用性和开发效率。因而 AOP 便成为了日志记录、监控管理、性能统计、异常处理、权限管理、统一认证等各个方面被广泛使用的技术。



追根溯源，我们之所以能无感知地在容器对象方法前后任意添加代码片段，那是由于 Spring 在运行期帮我们把切面中的代码逻辑动态“织入”到了容器对象方法内，所以说

AOP 本质上就是一个代理模式。然而在使用这种代理模式时，我们常常会用不好，那么这节课我们就来解析下有哪些常见的问题，以及背后的原理是什么。

案例 1：this 调用的当前类方法无法被拦截

假设我们正在开发一个宿舍管理系统，这个模块包含一个负责电费充值的类 ElectricService，它含有一个充电方法 charge()：

[复制代码](#)

```
1 @Service
2 public class ElectricService {
3
4     public void charge() throws Exception {
5         System.out.println("Electric charging ...");
6         this.pay();
7     }
8
9     public void pay() throws Exception {
10        System.out.println("Pay with alipay ...");
11        Thread.sleep(1000);
12    }
13
14 }
```

在这个电费充值方法 charge() 中，我们会使用支付宝进行充值。因此在这个方法中，我加入了 pay() 方法。为了模拟 pay() 方法调用耗时，代码执行了休眠 1 秒，并在 charge() 方法里使用 this.pay() 的方式调用这种支付方式。

但是因为支付宝支付是第三方接口，我们需要记录下接口调用时间。这时候我们就引入了一个 @Around 的增强，分别记录在 pay() 方法执行前后的时间，并计算出执行 pay() 方法的耗时。

[复制代码](#)

```
1 @Aspect
2 @Service
3 @Slf4j
4 public class AopConfig {
5     @Around("execution(* com.spring.puzzle.class5.example1.ElectricService.pay
6     public void recordPayPerformance(ProceedingJoinPoint joinPoint) throws Thr
7         long start = System.currentTimeMillis();
8         joinPoint.proceed();
9         long end = System.currentTimeMillis();
```

```
10         System.out.println("Pay method time cost (ms) : " + (end - start));  
11     }  
12 }
```

最后我们再通过定义一个 Controller 来提供电费充值接口，定义如下：

[复制代码](#)

```
1 @RestController  
2 public class HelloWorldController {  
3     @Autowired  
4     ElectricService electricService;  
5     @RequestMapping(path = "charge", method = RequestMethod.GET)  
6     public void charge() throws Exception {  
7         electricService.charge();  
8     };  
9 }
```

完成代码后，我们访问上述接口，会发现这段计算时间的切面并没有执行到，输出日志如下：

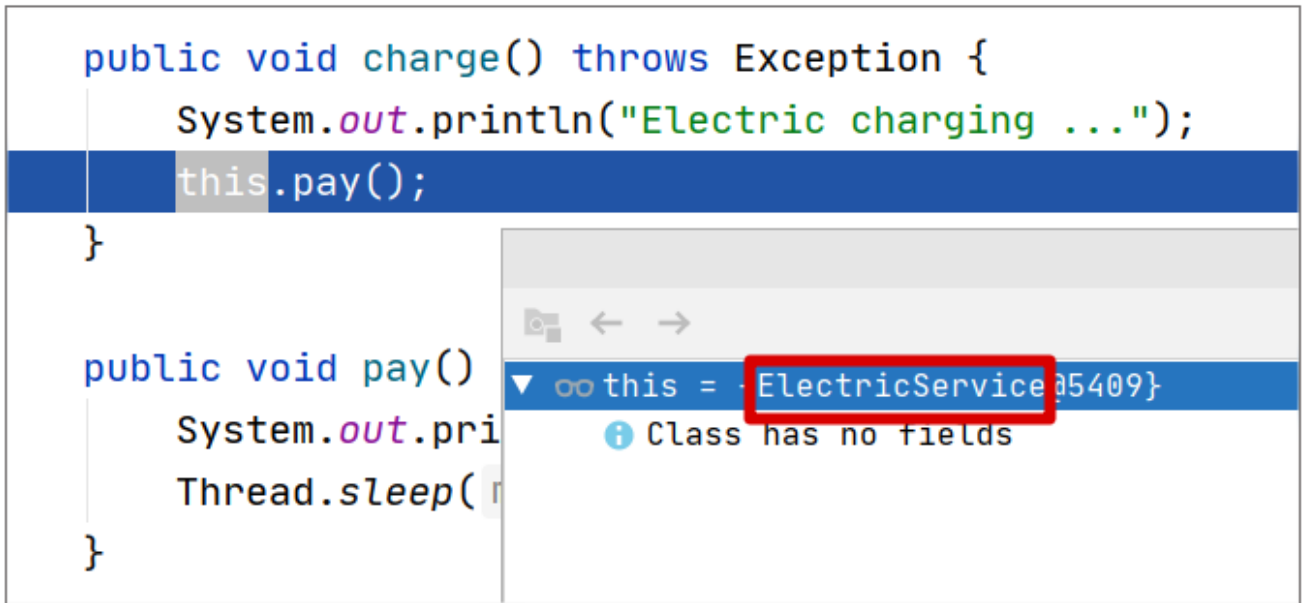
Electric charging ...

Pay with alipay ...

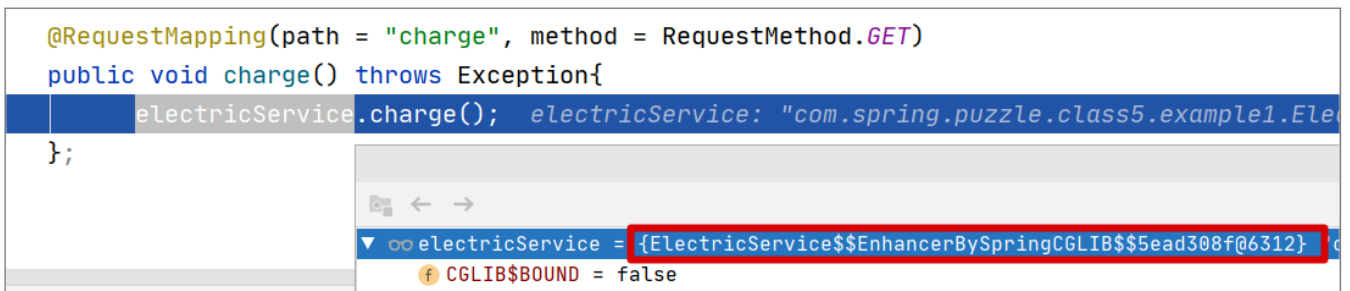
回溯之前的代码可知，在 @Around 的切面类中，我们很清晰地定义了切面对应的方法，但是却没有被执行到。这说明了在类的内部，通过 this 方式调用的方法，是没有被 Spring AOP 增强的。这是为什么呢？我们来分析一下。

案例解析

我们可以从源码中找到真相。首先来设置个断点，调试看看 this 对应的对象是什么样的：



可以看到，`this` 对应的就是一个普通的 `ElectricService` 对象，并没有什么特别的地方。再看看在 Controller 层中自动装配的 `ElectricService` 对象是什么样：

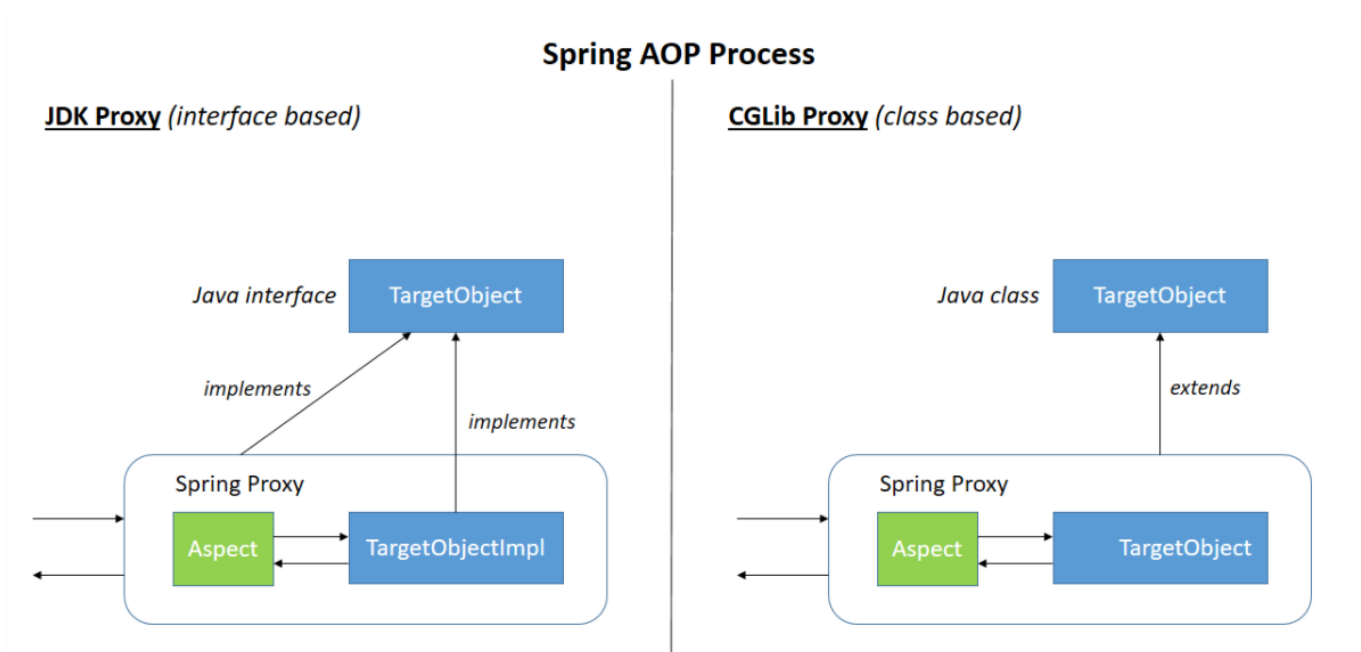


可以看到，这是一个被 Spring 增强过的 Bean，所以执行 `charge()` 方法时，会执行记录接口调用时间的增强操作。而 `this` 对应的对象只是一个普通的对象，并没有做任何额外的增强。

为什么 `this` 引用的对象只是一个普通对象呢？这还要从 Spring AOP 增强对象的过程来看。但在此之前，有些基础我需要在这里强调下。

1. Spring AOP 的实现

Spring AOP 的底层是动态代理。而创建代理的方式有两种，**JDK 的方式**和 **CGLIB 的方式**。JDK 动态代理只能对实现了接口的类生成代理，而不能针对普通类。而 CGLIB 是可以针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，来实现代理对象。具体区别可参考下图：



2. 如何使用 Spring AOP

在 Spring Boot 中，我们一般只要添加以下依赖就可以直接使用 AOP 功能：

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

而对于非 Spring Boot 程序，除了添加相关 AOP 依赖项外，我们还常常会使用 `@EnableAspectJAutoProxy` 来开启 AOP 功能。这个注解类引入（Import）`AspectJAutoProxyRegistrar`，它通过实现 `ImportBeanDefinitionRegistrar` 的接口方法来完成 AOP 相关 Bean 的准备工作。

补充完最基本的 Spring 底层知识和使用知识后，我们具体看下创建代理对象的过程。先来看下调用栈：

```

getProxy:206, CglibAopProxy (org.springframework.aop.framework)
getProxy:110, ProxyFactory (org.springframework.aop.framework)
createProxy:461, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
wrapIfNecessary:340, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
postProcessAfterInitialization:289, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
applyBeanPostProcessorsAfterInitialization:437, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
initializeBean:1790, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
doCreateBean:602, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
createBean:524, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
lambda$doGetBean$0:335, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1, 630074945 (org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$34)
getSingleton:234, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:333, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:208, AbstractBeanFactory (org.springframework.beans.factory.support)
preInstantiateSingletons:944, DefaultListableBeanFactory (org.springframework.beans.factory.support)
finishBeanFactoryInitialization:917, AbstractApplicationContext (org.springframework.context.support)
refresh:582, AbstractApplicationContext (org.springframework.context.support)
<init>:93, AnnotationConfigApplicationContext (org.springframework.context.annotation)

```

创建代理对象的时机就是创建一个 Bean 的时候，而创建的的关键工作其实是由 AnnotationAwareAspectJAutoProxyCreator 完成的。它本质上是一种 BeanPostProcessor。所以它的执行是在完成原始 Bean 构建后的初始化 Bean (initializeBean) 过程中。而它到底完成了什么工作呢？我们可以看下它的 postProcessAfterInitialization 方法：

[复制代码](#)

```

1 public Object postProcessAfterInitialization(@Nullable Object bean, String bea
2     if (bean != null) {
3         Object cacheKey = getCacheKey(bean.getClass(), beanName);
4         if (this.earlyProxyReferences.remove(cacheKey) != bean) {
5             return wrapIfNecessary(bean, beanName, cacheKey);
6         }
7     }
8     return bean;
9 }

```

上述代码中的关键方法是 wrapIfNecessary，顾名思义，在需要使用 AOP 时，它会把创建的原始的 Bean 对象 wrap 成代理对象作为 Bean 返回。具体到这个 wrap 过程，可参考下面的关键代码行：

[复制代码](#)

```

1 protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey
2     // 省略非关键代码
3     Object[] specificInterceptors = getAdvisesAndAdvisorsForBean(bean.getClass(
4     if (specificInterceptors != DO_NOT_PROXY) {
5         this.advisedBeans.put(cacheKey, Boolean.TRUE);
6         Object proxy = createProxy(
7             bean.getClass(), beanName, specificInterceptors, new SingletonTarg
8         this.proxyTypes.put(cacheKey, proxy.getClass());

```



```
9         return proxy;
10    }
11    // 省略非关键代码
12 }
13
```

上述代码中，第 6 行的 `createProxy` 调用是创建代理对象的关键。具体到执行过程，它首先会创建一个代理工厂，然后将通知器（advisors）、被代理对象等信息加入到代理工厂，最后通过这个代理工厂来获取代理对象。一些关键过程参考下面的方法：

[复制代码](#)

```
1  protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
2      @Nullable Object[] specificInterceptors, TargetSource targetSource) {
3      // 省略非关键代码
4      ProxyFactory proxyFactory = new ProxyFactory();
5      if (!proxyFactory.isProxyTargetClass()) {
6          if (shouldProxyTargetClass(beanClass, beanName)) {
7              proxyFactory.setProxyTargetClass(true);
8          }
9          else {
10             evaluateProxyInterfaces(beanClass, proxyFactory);
11         }
12     }
13     Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
14     proxyFactory.addAdvisors(advisors);
15     proxyFactory.setTargetSource(targetSource);
16     customizeProxyFactory(proxyFactory);
17     // 省略非关键代码
18     return proxyFactory.getProxy(getProxyClassLoader());
19 }
```

经过这样一个过程，一个代理对象就被创建出来了。我们从 Spring 中获取到的对象都是这个代理对象，所以具有 AOP 功能。而之前直接使用 `this` 引用到的只是一个普通对象，自然也就没办法实现 AOP 的功能了。

问题修正

从上述案例解析中，我们知道，**只有引用的是被动态代理创建出来的对象，才会被 Spring 增强，具备 AOP 该有的功能。**那什么样的对象具备这样的条件呢？

有两种。一种是被 @Autowired 注解的，于是我们的代码可以改成这样，即通过 @Autowired 的方式，在类的内部，自己引用自己：

[复制代码](#)

```
1 @Service
2 public class ElectricService {
3     @Autowired
4     ElectricService electricService;
5     public void charge() throws Exception {
6         System.out.println("Electric charging ...");
7         //this.pay();
8         electricService.pay();
9     }
10    public void pay() throws Exception {
11        System.out.println("Pay with alipay ...");
12        Thread.sleep(1000);
13    }
14 }
```

另一种方法就是直接从 AopContext 获取当前的 Proxy。那你可能会问了，AopContext 是什么？简单说，它的核心就是通过一个 ThreadLocal 来将 Proxy 和线程绑定起来，这样就可以随时拿出当前线程绑定的 Proxy。

不过使用这种方法有个小前提，就是需要在 @EnableAspectJAutoProxy 里加一个配置项 exposeProxy = true，表示将代理对象放入到 ThreadLocal，这样才可以直接通过 AopContext.currentProxy() 的方式获取到，否则会报错如下：

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


Sat Apr 10 15:59:14 CST 2021

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

Cannot find current proxy: Set 'exposeProxy' property on Advised to 'true' to make it available.


java.lang.IllegalStateException: Cannot find current proxy: Set 'exposeProxy' property on Advised to 'true' to make it available.
at org.springframework.aop.framework.AopContext.currentProxy(AopContext.java:69)
at com.spring.puzzle.class5.example1.ElectricService.charge(ElectricService.java:20)
at com.spring.puzzle.class5.example1.ElectricService\$\$FastClassBySpringCGLIB\$\$7c2be586.invoke(<generated>)
at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)
at org.springframework.aop.framework.CglibAopProxy\$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:769)
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:163)
at org.springframework.aop.framework.CglibAopProxy\$CglibMethodInvocation.proceed(CglibAopProxy.java:747)
at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:93)
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
at org.springframework.aop.framework.CglibAopProxy\$CglibMethodInvocation.proceed(CglibAopProxy.java:747)
at org.springframework.aop.framework.CglibAopProxy\$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:689)

按这个思路，我们修改下相关代码：

 复制代码

```
1 import org.springframework.aop.framework.AopContext;
2 import org.springframework.stereotype.Service;
3 @Service
4 public class ElectricService {
5     public void charge() throws Exception {
6         System.out.println("Electric charging ...");
7         ElectricService electric = ((ElectricService) AopContext.currentProxy(
8             electric.pay());
9     }
10    public void pay() throws Exception {
11        System.out.println("Pay with alipay ...");
12        Thread.sleep(1000);
13    }
14 }
```

同时，不要忘记修改 EnableAspectJAutoProxy 注解的 exposeProxy 属性，示例如下：

 复制代码

```
1 @SpringBootApplication
2 @EnableAspectJAutoProxy(exposeProxy = true)
3 public class Application {
4     // 省略非关键代码
5 }
```

这两种方法的效果其实是一样的，最终我们打印出了期待的日志，到这，问题顺利解决了。


 复制代码

```
1 Electric charging ...
2 Pay with alipay ...
3 Pay method time cost(ms): 1005
```

案例 2：直接访问被拦截类的属性抛空指针异常

接上一个案例，在宿舍管理系统中，我们使用了 charge() 方法进行支付。在统一结算的时候我们会用到一个管理员用户付款编号，这时候就用到了几个新的类。

User 类，包含用户的付款编号信息：

 复制代码

```
1 public class User {
2     private String payNum;
3     public User(String payNum) {
4         this.payNum = payNum;
5     }
6     public String getPayNum() {
7         return payNum;
8     }
9     public void setPayNum(String payNum) {
10        this.payNum = payNum;
11    }
12 }
```

AdminUserService 类，包含一个管理员用户（User），其付款编号为 202101166；另外，这个服务类有一个 login() 方法，用来登录系统。

 复制代码

```
1 @Service
2 public class AdminUserService {
3     public final User adminUser = new User("202101166");
4
5     public void login() {
6         System.out.println("admin user login...");
7     }
8 }
```

我们需要修改 ElectricService 类实现这个需求：在电费充值时，需要管理员登录并使用其编号进行结算。完整代码如下：

 复制代码

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Service;
3 @Service
4 public class ElectricService {
5     @Autowired
6     private AdminUserService adminUserService;
7     public void charge() throws Exception {
8         System.out.println("Electric charging ...");
9         this.pay();
10    }
11
12    public void pay() throws Exception {
13        adminUserService.login();
14    }
15 }
```

```
14         String payNum = adminUserService.adminUser.getPayNum();
15         System.out.println("User pay num : " + payNum);
16         System.out.println("Pay with alipay ...");
17         Thread.sleep(1000);
18     }
19 }
```

代码完成后，执行 `charge()` 操作，一切正常：

[复制代码](#)

```
1 Electric charging ...
2 admin user login...
3 User pay num : 202101166
4 Pay with alipay ...
```

这时候，由于安全需要，就需要管理员在登录时，记录一行日志以便于以后审计管理员操作。所以我们添加一个 AOP 相关配置类，具体如下：

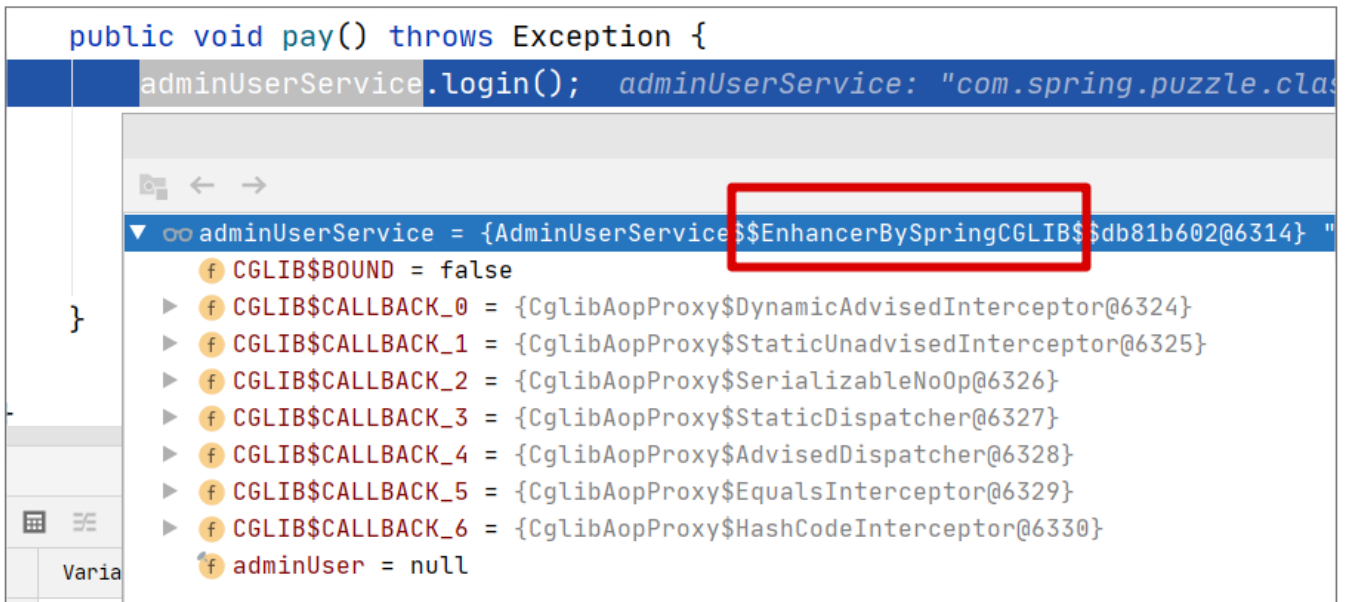
[复制代码](#)

```
1 @Aspect
2 @Service
3 @Slf4j
4 public class AopConfig {
5     @Before("execution(* com.spring.puzzle.class5.example2.AdminUserService.lo
6         public void logAdminLogin(JoinPoint pjp) throws Throwable {
7         System.out.println("! admin login ...");
8     }
9 }
```

添加这段代码后，我们执行 `charge()` 操作，发现不仅没有相关日志，而且在执行下面这一行代码的时候直接抛出了 `NullPointerException`：

```
String payNum = dminUserService.user.getPayNum();
```

本来一切正常的代码，因为引入了一个 AOP 切面，抛出了 `NullPointerException`。这会是什么原因呢？我们先 debug 一下，来看看加入 AOP 后调用的对象是什么样子。



可以看出，加入 AOP 后，我们的对象已经是一个代理对象了，如果你眼尖的话，就会发现在上图中，属性 `adminUser` 确实为 `null`。为什么会这样？为了解答这个诡异的问题，我们需要进一步理解 Spring 使用 CGLIB 生成 Proxy 的原理。

案例解析

我们在上一个案例中解析了创建 Spring Proxy 的大体过程，在这里，我们需要进一步研究一下通过 Proxy 创建出来的是一个什么样的对象。正常情况下，`AdminUserService` 只是一个普通的对象，而 AOP 增强过的则是一个 `AdminUserService$$EnhancerBySpringCGLIB$$xxxx`。

这个类实际上是 `AdminUserService` 的一个子类。它会 `overwrite` 所有 `public` 和 `protected` 方法，并在内部将调用委托给原始的 `AdminUserService` 实例。

从具体实现角度看，CGLIB 中 AOP 的实现是基于 `org.springframework.cglib.proxy` 包中 `Enhancer` 和 `MethodInterceptor` 两个接口来实现的。

整个过程，我们可以概括为三个步骤：

定义自定义的 `MethodInterceptor` 负责委托方法执行；

创建 `Enhancer` 并设置 `Callback` 为上述 `MethodInterceptor`；

`enhancer.create()` 创建代理。

接下来，我们来具体分析一下 Spring 的相关实现源码。

在上个案例分析里，我们简要提及了 Spring 的动态代理对象的初始化机制。在得到 Advisors 之后，会通过 ProxyFactory.getProxy 获取代理对象：

[复制代码](#)

```
1 public Object getProxy(ClassLoader classLoader) {
2     return createAopProxy().getProxy(classLoader);
3 }
```

在这里，我们以 CGLIB 的 Proxy 的实现类 CglibAopProxy 为例，来看看具体的流程：

[复制代码](#)

```
1 public Object getProxy(@Nullable ClassLoader classLoader) {
2     // 省略非关键代码
3     // 创建及配置 Enhancer
4     Enhancer enhancer = createEnhancer();
5     // 省略非关键代码
6     // 获取Callback: 包含DynamicAdvisedInterceptor, 亦是MethodInterceptor
7     Callback[] callbacks = getCallbacks(rootClass);
8     // 省略非关键代码
9     // 生成代理对象并创建代理 (设置 enhancer 的 callback 值)
10    return createProxyClassAndInstance(enhancer, callbacks);
11    // 省略非关键代码
12 }
```

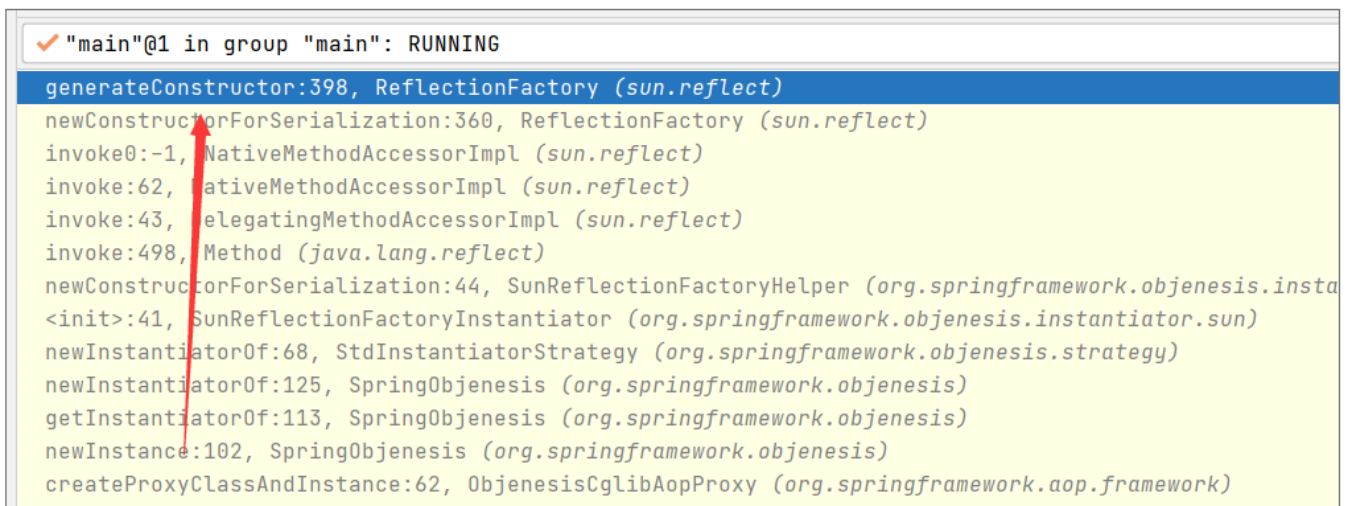
上述代码中的几个关键步骤大体符合之前提及的三个步骤，其中最后一步一般都会执行到 CglibAopProxy 子类 ObjenesisCglibAopProxy 的 createProxyClassAndInstance() 方法：

[复制代码](#)

```
1 protected Object createProxyClassAndInstance(Enhancer enhancer, Callback[] cal
2     //创建代理类Class
3     Class<?> proxyClass = enhancer.createClass();
4     Object proxyInstance = null;
5     //spring.objenesis.ignore默认为false
6     //所以objenesis.isWorthTrying()一般为true
7     if (objenesis.isWorthTrying()) {
8         try {
9             // 创建实例
10            proxyInstance = objenesis.newInstance(proxyClass, enhancer.getUseCach
```

```
11     }
12     catch (Throwable ex) {
13         // 省略非关键代码
14     }
15 }
16
17 if (proxyInstance == null) {
18     // 尝试普通反射方式创建实例
19     try {
20         Constructor<?> ctor = (this.constructorArgs != null ?
21             proxyClass.getDeclaredConstructor(this.constructorArgTypes) :
22             proxyClass.getDeclaredConstructor());
23         ReflectionUtils.makeAccessible(ctor);
24         proxyInstance = (this.constructorArgs != null ?
25             ctor.newInstance(this.constructorArgs) : ctor.newInstance());
26         //省略非关键代码
27     }
28 }
29 // 省略非关键代码
30 ((Factory) proxyInstance).setCallbacks(callbacks);
31 return proxyInstance;
32 }
```

这里我们可以了解到，Spring 会默认尝试使用 objenesis 方式实例化对象，如果失败则再次尝试使用常规方式实例化对象。现在，我们可以进一步查看 objenesis 方式实例化对象的流程。



参照上述截图所示调用栈，objenesis 方式最后使用了 JDK 的 `ReflectionFactory.newConstructorForSerialization()` 完成了代理对象的实例化。而如果你稍微研究下这个方法，你会惊讶地发现，这种方式创建出来的对象是不会初始化类成员变量的。

所以说到这里，聪明的你可能已经觉察到真相已经暴露了，我们这个案例的核心是代理类实例的默认构建方式很特别。在这里，我们可以总结和对比下通过反射来实例化对象的方式，包括：

```
java.lang.Class.newInstance()
```

```
java.lang.reflect.Constructor.newInstance()
```

```
sun.reflect.ReflectionFactory.newConstructorForSerialization().newInstance()
```

前两种初始化方式都会同时初始化类成员变量，但是最后一种通过 `ReflectionFactory.newConstructorForSerialization().newInstance()` 实例化类则不会初始化类成员变量，这就是当前问题的最终答案了。

问题修正

了解了问题的根本原因后，修正起来也就不困难了。既然是无法直接访问被拦截类的成员变量，那我们就换个方式，在 `UserService` 里写个 `getUser()` 方法，从内部访问获取变量。

我们在 `AdminUserService` 里加了个 `getUser()` 方法：

```
1 public User getUser() {  
2     return user;  
3 }
```

[复制代码](#)

在 `ElectricService` 里通过 `getUser()` 获取 `User` 对象：

```
// 原来出错的方式：
```

```
//String payNum = adminUserService.adminUser.getPayNum();
```

```
// 修改后的方式：
```

```
String payNum = adminUserService.getAdminUser().getPayNum();
```

运行下来，一切正常，可以看到管理员登录日志了：

```
1 Electric charging ...
2 ! admin login ...
3 admin user login...
4 User pay num : 202101166
5 Pay with alipay ...
```

[复制代码](#)

但你有没有产生另一个困惑呢？既然代理类的类属性不会被初始化，那为什么可以通过在 AdminUserService 里写个 getUser() 方法来获取代理类实例的属性呢？

我们再次回顾 createProxyClassAndInstance 的代码逻辑，创建代理类后，我们会调用 setCallbacks 来设置拦截后需要注入的代码：

```
1 protected Object createProxyClassAndInstance(Enhancer enhancer, Callback[] cal
2     Class<?> proxyClass = enhancer.createClass();
3     Object proxyInstance = null;
4     if (objenesis.isWorthTrying()) {
5         try {
6             proxyInstance = objenesis.newInstance(proxyClass, enhancer.getUseCach
7         }
8         // 省略非关键代码
9         ((Factory) proxyInstance).setCallbacks(callbacks);
10    return proxyInstance;
11 }
```

[复制代码](#)

通过代码调试和分析，我们可以得知上述的 callbacks 中会存在一种服务于 AOP 的 DynamicAdvisedInterceptor，它的接口是 MethodInterceptor（callback 的子接口），实现了拦截方法 intercept()。我们可以看下它是如何实现这个方法的：

```
1 public Object intercept(Object proxy, Method method, Object[] args, MethodProx
2     // 省略非关键代码
3     TargetSource targetSource = this.advised.getTargetSource();
4     // 省略非关键代码
5     if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
6         Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method,
7         retVal = methodProxy.invoke(target, argsToUse);
8     }
9     else {
10        // We need to create a method invocation...
11        retVal = new CglibMethodInvocation(proxy, target, method, args, targe
12    }
```

[复制代码](#)

```
13     retVal = processReturnType(proxy, target, method, retVal);
14     return retVal;
15 }
16 //省略非关键代码
17 }
```

当代理类方法被调用，会被 Spring 拦截，从而进入此 `intercept()`，并在此方法中获取被代理的原始对象。而在原始对象中，类属性是被实例化过且存在的。因此代理类是可以通过方法拦截获取被代理对象实例的属性。

说到这里，我们已经解决了问题。但如果你看得仔细，就会发现，其实你改变一个属性，也可以让产生的代理对象的属性值不为 `null`。例如修改启动参数 `spring.objenesis.ignore` 如下：



此时再调试程序，你会发现 `adminUser` 已经不为 `null` 了：

```
▼ proxyInstance = {AdminUserService$$EnhancerBySpringCGLIB$$1da65bcb@6613}
  f CGLIB$BOUND = true
  ▶ f CGLIB$CALLBACK_0 = {CglibAopProxy$DynamicAdvisedInterceptor@6435}
  ▶ f CGLIB$CALLBACK_1 = {CglibAopProxy$StaticUnadvisedInterceptor@6449}
  ▶ f CGLIB$CALLBACK_2 = {CglibAopProxy$SerializableNoOp@6466}
  ▶ f CGLIB$CALLBACK_3 = {CglibAopProxy$StaticDispatcher@6457}
  ▶ f CGLIB$CALLBACK_4 = {CglibAopProxy$AdvisedDispatcher@6364}
  ▶ f CGLIB$CALLBACK_5 = {CglibAopProxy$EqualsInterceptor@6467}
  ▶ f CGLIB$CALLBACK_6 = {CglibAopProxy$HashCodeInterceptor@6468}
  ▶ f adminUser = {User@6632}
```

所以这也是解决这个问题的一种方法，相信聪明的你已经能从前文贴出的代码中找出它能够工作起来的原理了。

重点回顾

通过以上两个案例的介绍，相信你对 Spring AOP 动态代理的初始化机制已经有了进一步的了解，这里总结重点如下：

1. 使用 AOP，实际上就是让 Spring 自动为我们创建一个 Proxy，使得调用者能无感知地调用指定方法。而 Spring 有助于我们在运行期里动态织入其它逻辑，因此，AOP 本质上就是一个动态代理。
2. 我们只有访问这些代理对象的方法，才能获得 AOP 实现的功能，所以通过 this 引用是无法正确使用 AOP 功能的。在不能改变代码结果前提下，我们可以通过 @Autowired、AopContext.currentProxy() 等方式获取相应的代理对象来实现所需的功能。
3. 我们一般不能直接从代理类中去拿被代理类的属性，这是因为除非我们显示设置 spring.objenesis.ignore 为 true，否则代理类的属性是会被 Spring 初始化的，我们可以通过在被代理类中增加一个方法来间接获取其属性。

思考题

第二个案例中，我们提到了通过反射来实例化类的三种方式：

```
java.lang.Class.newInstance()
```

```
java.lang.reflect.Constructor.newInstance()
```

```
sun.reflect.ReflectionFactory.newConstructorForSerialization().newInstance()
```

其中第三种方式不会初始化类属性，你能够写一个例子来证明这一点吗？

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 04 | Spring Bean生命周期常见错误

下一篇 06 | Spring AOP常见错误（下）

精选留言 (2)

写留言



哦吼掉了

2021-04-30

System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "./cglib"); //代理类持久化到文件

思考题：

AdminUserServiceSon继承AdminUserService，看起来父类都没有加载。但是为啥还是希望老师解答下...

展开 ∨



Ball

2021-04-30

🧐总结一下，今天以两个 AOP 场景下的问题为线索，深入 Spring 源码探讨了 Spring 的动态代理机制，还分享了 AOP 场景下问题的 debug 技巧。结合问题定位的过程，最终给出了问题的多种解决方案。👍

展开 ∨

