AOP避坑指南

无论是使用AspectJ语法，还是配合Annotation，使用AOP，实际上就是让Spring自动为我们创建一个Proxy，使得调用方能无感知地调用指定方法，但运行期却动态“织入”了其他逻辑，因此，AOP本质上就是一个代理模式。

因为Spring使用了CGLIB来实现运行期动态创建Proxy，如果我们没能深入理解其运行原理和实现机制，就极有可能遇到各种诡异的问题。

我们来看一个实际的例子。

假设我们定义了一个`UserService`的Bean：

```
@Component
public class UserService {
    // 成员变量:
    public final ZoneId zoneId = ZoneId.systemDefault();

    // 构造方法:
    public UserService() {
        System.out.println("UserService(): init...");
        System.out.println("UserService(): zoneId = " + this.zoneId);
    }

    // public方法:
    public ZoneId getZoneId() {
        return zoneId;
    }
```

再写个`MailService`，并注入`UserService`：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        String dt = ZonedDateTime.now(zoneId).toString();
        return "Hello, it is " + dt;
    }
}
```

最后用`main()`方法测试一下：

```
@Configuration
@ComponentScan
public class AppConfig {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        MailService mailService = context.getBean(MailService.class);
        System.out.println(mailService.sendMail());
    }
}
```

查看输出，一切正常：

```
UserService(): init...
UserService(): zoneId = Asia/Shanghai
Hello, it is 2020-04-12T10:23:22.917721+08:00[Asia/Shanghai]
```

关于作者

关注公众号不定期领红包：

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(public * com..*.UserService.*(..))")
    public void doAccessCheck() {
        System.err.println("[Before] do access check...");
    }
}
```

别忘了在`AppConfig`上加上`@EnableAspectJAutoProxy`。再次运行，不出意外的话，会得到一个`NullPointerException`：

```
Exception in thread "main" java.lang.NullPointerException: zone
    at java.base/java.util.Objects.requireNonNull(Objects.java:246)
    at java.base/java.time.Clock.system(Clock.java:203)
    at java.base/java.time.ZonedDateTime.now(ZonedDateTime.java:216)
    at com.itranswarp.learnjava.service.MailService.sendMail(MailService.java:19)
    at com.itranswarp.learnjava.AppConfig.main(AppConfig.java:21)
```

仔细跟踪代码，会发现`null`值出现在`MailService.sendMail()`内部的这一行代码：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
}
```

```
@Component
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

用`final`标注的成员变量为`null`？逗我呢？

怎么肥四？

为什么加了AOP就报NPE，去了AOP就一切正常？`final`字段不执行，难道JVM有问题？为了解答这个诡异的问题，我们需要深入理解Spring使用CGLIB生成Proxy的原理：

第一步，正常创建一个`UserService`的原始实例，这是通过反射调用构造方法实现的，它的行为和我们预期的完全一致；

第二步，通过CGLIB创建一个`UserService`的子类，并引用了原始实例和`LoggingAspect`：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target;
    LoggingAspect aspect;

    public UserService$$EnhancerBySpringCGLIB() {
    }

    public ZoneId getZoneId() {
        aspect.doAccessCheck();
        return target.getZoneId();
    }
}
```

如果我们观察Spring创建的AOP代理，它的类名总是类似`UserService$$EnhancerBySpringCGLIB$$1c76af9d`（你没

```
UserService original = new UserService();
```

第二个`UserService`实例实际上类型是`UserService$$EnhancerBySpringCGLIB`，它引用了原始的`UserService`实例：

```
UserService$$EnhancerBySpringCGLIB proxy = new UserService$$EnhancerBySpringCGLIB();
proxy.target = original;
proxy.aspect = ...
```

注意到这种情况仅出现在启用了AOP的情况，此刻，从`ApplicationContext`中获取的`UserService`实例是proxy，注入到`MailService`中的`UserService`实例也是proxy。

那么最终的问题来了：proxy实例的成员变量，也就是从`UserService`继承的`zoneId`，它的值是`null`。

原因在于，`UserService`成员变量的初始化：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

在`UserService$$EnhancerBySpringCGLIB`中，并未执行。原因是，没必要初始化proxy的成员变量，因为proxy的目的是代理方法。

实际上，成员变量的初始化是在构造方法中完成的。这是我们看到的代码：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    public UserService() {
    }
}
```

```
public class UserService {
    public final ZoneId zoneId;
    public UserService() {
        super(); // 构造方法的第一行代码总是调用super()
        zoneId = ZoneId.systemDefault(); // 继续初始化成员变量
    }
}
```

然而，对于Spring通过CGLIB动态创建的`UserService$$EnhancerBySpringCGLIB`代理类，它的构造方法中，并未调用`super()`，因此，从父类继承的成员变量，包括`final`类型的成员变量，统统都没有初始化。

有的童鞋会问：Java语言规定，任何类的构造方法，第一行必须调用`super()`，如果没有，编译器会自动加上，怎么Spring的CGLIB就可以搞特殊？

这是因为自动加`super()`的功能是Java编译器实现的，它发现你没加，就自动给加上，发现你加错了，就报编译错误。但实际上，如果直接构造字节码，一个类的构造方法中，不一定非要调用`super()`。Spring使用CGLIB构造的Proxy类，是直接生成字节码，并没有源码-编译-字节码这个步骤，因此：

⚠ Spring通过CGLIB创建的代理类，不会初始化代理类自身继承的任何成员变量，包括final类型的成员变量！

再考察`MailService`的代码：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
```

如果启动了AOP，注入的是代理后的`UserService$$EnhancerBySpringCGLIB`实例，那么问题大了：获取的`UserService$$EnhancerBySpringCGLIB`实例的`zoneId`字段，永远为`null`。

那么问题来了：启用了AOP，如何修复？

修复很简单，只需要把直接访问字段的代码，改为通过方法访问：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        // 不要直接访问UserService的字段:
        ZoneId zoneId = userService.getZoneId();
        ...
    }
}
```

无论注入的`UserService`是原始实例还是代理实例，`getZoneId()`都能正常工作，因为代理类会覆写`getZoneId()`方法，并将其委托给原始实例：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target = ...
    ...

    public ZoneId getZoneId() {
        return target.getZoneId();
    }
}
```

注意到我们还给`UserService`添加了一个`public`+`final`的方法：

```
@Component
public class UserService {
    ...
    public final ZoneId getFinalZoneId() {
        return zoneId;
    }
}
```

如果在`MailService`中，调用的不是`getZoneId()`，而是`getFinalZoneId()`，又会出现`NullPointerException`，这是因为，代理类无法覆写`final`方法（这一点绕不过JVM的ClassLoader检查），该方法返回的是代理类的`zoneId`字段，即`null`。

实际上，如果我们加上日志，Spring在启动时会打印一个警告：

```
10:43:09.929 [main] DEBUG org.springframework.aop.framework.CglibAopProxy - Final method [public final java.time.ZoneId xxx.UserService.getFinalZoneId()] cannot get proxied via CGLIB: Calls to this method will NOT be routed to the target instance and might lead to NPEs against uninitialized fields in the proxy instance.
```

上面的日志大意就是，因为被代理的`UserService`有一个`final`方法`getFinalZoneId()`，这会导致其他Bean如果调用此方法，无法将其代理到真正的原始实例，从而可能发生NPE异常。

因此，正确使用AOP，我们需要一个避坑指南：

1. 访问被注入的Bean时，总是调用方法而非直接访问字段；
2. 编写Bean时，如果可能会被代理，就不要编写`public final`方法。

这样才能保证有没有AOP，代码都能正常工作。

思考

为什么Spring刻意不初始化Proxy继承的字段？

小结

由于Spring通过CGLIB实现代理类，我们要避免直接访问Bean的字段，以及由`final`方法带来的“未代理”问题。

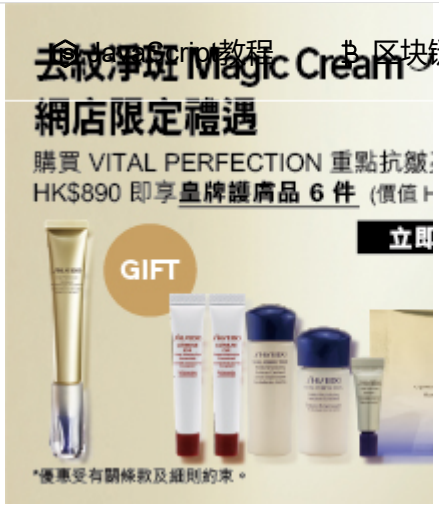
遇到CglibAopProxy的相关日志，务必要仔细检查，防止因为AOP出现NPE异常。

读后有收获可以支付宝请作者喝咖啡：

还可以分享给朋友：

分享到微博

上一页 下一页



评论

发表评论

登录后发表评论