MinGRn 5

关于代理：为什么 JDK 动态代理只能为接口生成代理？

java 代理 spring aop cglib 发布于 2月24日

写在前面

相信每当想起有关动态代理的时候大家都会脱口而出的就是：`cglib` 动态 和 `JDK` 动态代理。

再细一点的话也就是 `cglib` 动态代理底层使用的是继承，`JDK` 动态代理使用的实现。

那么，为什么 `JDK` 动态代理一定要是实现接口的形式？使用继承不行吗？或者说为什么 `JDK` 动态代理只能为接口生成代理对象不能为普通类生成代理对象？

不知道诸君有没有想过这个问题?

下面就开始解答该问题：

准备环境

为了解答该问题我们首先来构建一个程序复现该问题。

现在就基于 `spring V5.0.7.RELEASE` 构建一个 `maven` 实例，项目名称随意！

包名：`com.mingrn.proxy`

在 `pom` 文件中引入如下依赖：

```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-framework-bom</artifactId>
                <version>5.0.7.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
        </dependency>
    </dependencies>
```

现在来看下项目结构：

```
.
└── com.mingrn.proxy
    ├── App.java
    ├── aop
    │   └── AspectProxy.java
    ├── config
    │   └── AppConfig.java
    └── service
        ├── UserService.java
        └── impl
            └── UserServiceImpl.java
```

- UserService.java：

```
public interface UserService {
    List find();
}
```

- UserServiceImpl.java：

```
@Service("userService")
public class UserServiceImpl implements UserService {

    @Override
    public List find() {
        System.out.println("find");
        return null;
    }
}
```

- AppConfig.java：

```
@Configuration
@ComponentScan("com.mingrn.proxy")
@EnableAspectJAutoProxy(proxyTargetClass = false)
public class AppConfig {
}
```

- AspectProxy.java：

```
@Component
@Aspect
public class AspectProxy {

    @Pointcut("execution(* *com.mingrn.proxy.service.*.*(..))")
    public void pointCut() {
    }

    @Before("pointCut()")
    public void before() {
        System.out.println("before");
    }
}
```

现在我们再 App.java 中编写测试，保证 AOP 正常运行：

```
public class App {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        UserService userService = (UserService) context.getBean("userService");
        userService.find();
    }
}
```

输出结果如下，表示我们的 AOP 没什么问题！

```
before
find
```

现在开始进行问题复现：

复现问题

我们知道，`SpringAOP` 默认使用的是 JDK 动态代理，那么来看下下面这条输出：

```
UserService userService = (UserService) context.getBean("userService");
System.out.println("userService instanceof UserServiceImpl ? " + (userService instanceof UserServiceImpl));
```

> 为了演示需要，特意在 `UserServiceImpl` 上为 `bean` 定义了一个名称，如果直接使用 `context.getBean(UserServiceImpl.class);` 根据类型获取会出现启动报错问题。

不知道诸君觉得输出的是 `true` 还是 `false`？答案利索当然的是 `false` 了，对不对。因为 `JDK` 动态代理会为目标对象在内存中生成一个新的实现代理类，这个新的代理类跟我们源代码中的 `UserServiceImpl.java` 没有任何关系，唯一的关系是：**都实现了 `UserService` 接口类。**

那么，将 `AppConfig` 的代理设置为 `true` 即 `@EnableAspectJAutoProxy(proxyTargetClass = true)`。

现在再次打印相信都知道结果为 `true` 了，原因就是生成的代理对象与源代码中的 `UserServiceImpl.java` 是父子关系。

现在我们代理还原为 `false` 即 `@EnableAspectJAutoProxy(proxyTargetClass = false)` 再看下如下输出：

```
UserService userService = (UserService) context.getBean("userService");
System.out.println("userService instanceof Proxy ? " + (userService instanceof Proxy));
```

诸君现在知道输出结果是什么吗？如果对 JDK 代理有些了解的话就会知道输出结果为：**TRUE。**那么，为什么会这样？知道该问题后我们自然而然的就知道了：**为什么 JDK 动态代理一定是代理接口类！**

为什么 JDK 动态代理一定是代理接口类！

接着上面，不知道诸君有没有看过 JDK 动态代理的源码，如果看过了就应该知道 `java.lang.reflect.Proxy` 类中有一个内部类 `ProxyClassFactory`。该内部类有如下一个方法：

```
public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {}
```

在该方法中调用了 `sun.misc.ProxyGenerator` 类的如下方法：

```
public static byte[] generateProxyClass(final String var0, Class<?>[] var1, int var2){}
```

即代码如下：

```
/**
 * Generate the specified proxy class.
 */
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(proxyName, interfaces, accessFlags);
```

这条代码就是为目标类在内存中生成一个代理类，可以看到返回的类型是 `byte[]`。所以，现在要做的就是利用该语句为 `UserService` 生成一个代理对象，并将二进制数据生成为一个 `class` 文件！我们只需要利用反编译工具查看一个该代码即可一幕了然了。

现在开始：

```
public class App {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        UserService userService = (UserService) context.getBean("userService");

        Class<?>[] interfaces = new Class[]{UserService.class};
        byte[] bytes = ProxyGenerator.generateProxyClass("UserService", interfaces);

        File file = new File("/<path>/UserService.class");
        try {
            OutputStream outputStream = new FileOutputStream(file);
            outputStream.write(bytes);
            outputStream.flush();
            outputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

最后，就会在我们的磁盘中生成一个 `UserService.class` 字节码文件，我们只需要反编译即可（可以直接利用 IDE 查看，如 IntelliJ IDEA）。打开字节码文件（省略无关内容）如下所示：

```
public final class UserService extends Proxy implements com.mingrn.proxy.service.UserService {
    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m0;

    public UserService(InvocationHandler var1) throws  {
        super(var1);
    }

    public final boolean equals(Object var1) throws  {
        // ...
    }

    public final List find() throws  {
        // ...
    }

    public final String toString() throws  {
        // ...
    }

    public final int hashCode() throws  {
        // ...
    }
```

我们需要关心的仅仅是生成的 `UserService` 类的继承与实现关系即可：

```
class UserService extends Proxy implements com.mingrn.proxy.service.UserService {}
```

现在明白为什么 JDK 动态代理一定是只能为接口生成代理类而不是使用继承了吗？

总结一句话

JDK 动态代理是为接口生成代理对象，该代理对象继承了 JAVA 标准类库 `Proxy.java` 类并且实现了目标对象。由于 JAVA 遵循单继承多实现原则所以 JDK 无法利用继承来为目标对象生产代理对象。

最后，以后如果再由面试官问题该问题即可将上面这句话砸他脸上，狠狠的那种！

阅读 1.3k • 发布于 2月24日 举报

赞 1 收藏 1 分享

MinGRn
5
关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

Spring AOP就是这么简单啦

前言 只有光头才能变强 上一篇已经讲解了Spring IOC知识点一网打尽！，这篇主要是讲解Spring的AOP模块~ 之前我已经写过一篇...

Java3y • 阅读 12.7k • 25 赞

【Spring】一次线上@Transational事务注解未生效的原因探究

现象描述 上周同事发现其基于mySql实现的分布式锁的线上代码存在问题，代码简化如下： {代码...} 实际执行test()后发现doInside(...

Jiadong • 阅读 8.6k • 12 赞 • 10 评论

Java设计模式综合运用(动态代理+Spring AOP)

代理(Proxy)是一种提供了对目标对象另外的访问方式，即通过代理对象访问目标对象。这样做的好处是:可以在目标对象实现的基础...

landy8530 • 阅读 1.8k • 9 赞

Spring AOP的实现原理

AOP（Aspect Orient Programming），我们一般称为面向方面（切面）编程，作为面向对象的一种补充，用于处理系统中分布于...

listen • 阅读 5.5k • 8 赞 • 4 评论

探析Spring AOP（二）:Spring AOP的实现机制

Spring AOP 属于第二代 AOP， 采用动态代理机制和字节码生成技术实现 。 与最初的 AspectJ 采用编译器将横切逻辑织入目标...

zhangpc060 • 阅读 820 • 2 赞

Spring入门IOC和AOP学习笔记

Spring有两个核心接口：BeanFactory和ApplicationContext,ApplicationContext是BeanFactory的子接口、它们都可以代表Spring容...

scu酱油仔 • 阅读 1.1k • 2 赞

向您生动地讲解Spring AOP 源码（3）

前言 往期文章： Spring IoC - Spring IoC 的设计 Spring IoC - IoC 容器初始化 源码解析 Spring IoC - 依赖注入 源码解析 Spring AO...