25_源码分析：Spring AOP JDK 动态代理实现原理

1、开篇

上节课延续上节课的 Spring AOP 的主题，介绍了 Spring AOP 中的几个核心概念：Aspect、Join Points、Pointcuts 以及 Advice。并且通过代码例子告诉大家它们的基本用法。本节课会通过源码分析介绍 Spring AOP JDK 动态代理实现的原理。内容包括：

- invoke：实现 AOP 中具体的逻辑；
- getProxy：获取产生的代理类；

2、invoke：实现 AOP 中具体的逻辑

从前面的课程我们可以知道，JDK 中 AOP 的实现是基于 java.lang.reflect 包中 Proxy 和 InvocationHandler 两个接口来实现的。对于 InvocationHandler 的创建，需要我们重写三个方法：

- 构造函数：将目标代理对象传入。
- invoke：实现 AOP 中具体的逻辑。
- getProxy：获取产生的代理类。

而在 Spring 框架中，JDK 方式的代理也是实现了上述过程，在源码分析上面我们也将其分为 inovke 和 getProxy 两大部分给大家介绍。

JDK 动态代理中是通过 proxyFactory.getProxy 获取代理的，如下面代码所示，通过 createAopProxy()生成对应的 proxyFactory 然后在调用其中的 getProxy 方法。

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

createAopProxy 中决定的实现类为 JdkDynamicAopProxy，如下面代码所示，它实现了 AopProxy 和 InvocationHandler。

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable
```

由于我们会实现 InvocationHandler，并且 override 其中的 invoke 方法，然后通过 InvocationHandler 中的 getProxy 方法获取代理类，通过对代理类调用 process 方法执行 AOP 的相关操作。

那么首先来看看 override 的 invoke 方法的源码，如图 1 所示，在 invoke 方法中我们需要关注红框标注的部分：

- 在获取目标类以后，通过目标类获取要执行方法的拦截器链，这里的变量为 chain。
- 如果 chain 为空说明没有在方法上进行拦截，那么直接调用切点方法就行了。
- 如果 chain 不为空说明在方法上有拦截，于是将拦截器封装在 ReflectiveMethodInvocation，并执行，这部分也是 AOP 需要关注的。

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    MethodInvocation invocation;
    Object oldProxy = null;
    boolean setProxyContext = false;
    TargetSource targetSource = this.advised.targetSource;
    Class<?> targetClass = null;
    Object target = null;
    try {

        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {…
        }
        if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {…
        }
        if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&…
        }
        Object retVal;
        if (this.advised.exposeProxy) {…
        }
        //获取AOP 目标类
        target = targetSource.getTarget();
        if (target != null) {
            targetClass = target.getClass();
        }
        //获取当前方法的拦截器链chain;
        List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
        //检查是否有advice。 如果没有，需要回退到目标的直接反射调用，从而避免创建 Method Invocation。
        //如果chain为空，直接调用切点方法;
        if (chain.isEmpty()) {
            Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
            retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
        }
        else {
            //chain不为空，将拦截器封装在ReflectiveMethodInvocation，并执行
            invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
            //通过拦截器调用链执行 joinpoint
            retVal = invocation.proceed();
        }
        // Massage return value if necessary.
        Class<?> returnType = method.getReturnType();
        if (retVal != null && retVal == target && returnType.isInstance(proxy) &&
                !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
            retVal = proxy;
```

图 1 invoke 方法

上面通过了 getInterceptorsAndDynamicInterceptionAdvice 获取目标类执行方法的拦截器链 chain，这里对 getInterceptorsAndDynamicInterceptionAdvice 进行解析。如图 2 所示，依旧关注红框的部分：

收到获取此方法的拦截器链，依然将 Advisor 分为了三种类型：

- PointcutAdvisor：当前 class、method 与 Pointcut 匹配时，才获取并保存 MethodInterceptor 的 List。
- IntroductionAdvisor：当前 class 与 Pointcut 匹配时，获取并保存 MethodInterceptor 的 List。
- 其他类型：直接获取并保存 MethodInterceptor 的 List；

```
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
        Advised config, Method method, Class<?> targetClass) {
    List<Object> interceptorList = new ArrayList<Object>(config.getAdvisors().length);
    Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaringClass());
    boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
    AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
    for (Advisor advisor : config.getAdvisors()) {
        if (advisor instanceof PointcutAdvisor) {
            // Add it conditionally.
            PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
            if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {
                MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
                MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
                if (MethodMatchers.matches(mm, method, actualClass, hasIntroductions)) {
                    if (mm.isRuntime()) {
                        // Creating a new object instance in the getInterceptors() method
                        // isn't a problem as we normally cache created chains.
                        for (MethodInterceptor interceptor : interceptors) {
                            interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm));
                        }
                    }
                    else {
                        interceptorList.addAll(Arrays.asList(interceptors));
                    }
                }
            }
        }
        else if (advisor instanceof IntroductionAdvisor) {
            IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
            if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass)) {
                Interceptor[] interceptors = registry.getInterceptors(advisor);
                interceptorList.addAll(Arrays.asList(interceptors));
            }
        }
        else {
            Interceptor[] interceptors = registry.getInterceptors(advisor);
            interceptorList.addAll(Arrays.asList(interceptors));
        }
    }
    return interceptorList;
}
```


图 2 getInterceptorsAndDynamicInterceptionAdvice

正如图 2 中 getInterceptorsAndDynamicInterceptionAdvice 处理 PointcutAdvisor 类型的时候会用到 getInterceptors 方法获取 MethodInterceptor 的数组类型，这里再看看 getInterceptors 内部的实现是如何的。

如图 3 所示，方法体中使用 advisor.getAdvice()获取 Advice。假设注解 Advisor 以 @Before 注解形式存在，那么此时得到的 Advice 应该是 AspectJMethodBeforeAdvice。进入的是 interceptors.add(adapter.getInterceptor(advisor));这行代码，将 AspectJMethodBeforeAdvice 包装成 MethodInterceptor 类型，并返回。

```
public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException {
    List<MethodInterceptor> interceptors = new ArrayList<MethodInterceptor>(3);
    Advice advice = advisor.getAdvice();
    if (advice instanceof MethodInterceptor) {
        interceptors.add((MethodInterceptor) advice);
    }
    for (AdvisorAdapter adapter : this.adapters) {
        if (adapter.supportsAdvice(advice)) {
            interceptors.add(adapter.getInterceptor(advisor));
        }
    }
    if (interceptors.isEmpty()) {
        throw new UnknownAdviceTypeException(advisor.getAdvice());
    }
    return interceptors.toArray(new MethodInterceptor[interceptors.size()]);
}
```

图 3 getInterceptors

上面说了如何获取方法拦截链 chain，下面接着说法拦截链为空和不为空的处理情况。拦截器链为空，如下面代码所示，表示当前的 Method 并没有 Advice 逻辑需要增强。

```
Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
```

拦截器链不为空，如图 4 所示，需要创建 ReflectiveMethodInvocation 对象，对象中对 proxy、target、targetClass、method、arguments 以及 interceptorsAndDynamicMethodMatchers 进行了定义。

```
protected ReflectiveMethodInvocation(
    Object proxy, Object target, Method method, Object[] arguments,
    Class<?> targetClass, List<Object> interceptorsAndDynamicMethodMatchers) {
    this.proxy = proxy;
    this.target = target;
    this.targetClass = targetClass;
    this.method = BridgeMethodResolver.findBridgedMethod(method);
    this.arguments = AopProxyUtils.adaptArgumentsIfNecessary(method, arguments);
    this.interceptorsAndDynamicMethodMatchers = interceptorsAndDynamicMethodMatchers;
}
```

图 4 ReflectiveMethodInvocation

有了 ReflectiveMethodInvocation 对象就要通过 proceed 方法对拦截链上的拦截器进行递归执行，也是一种对当前 Method 的链式增强。如图 5 所示，proceed 方法中会在拦截链上通过 currentInterceptorIndex 获取一个拦截器，然后对其进行动态代理方法的匹配，如果匹配通过了执行拦截器中的方法，否则跳过该拦截器，执行拦截器链上的下一个拦截器。如果是一个拦截器就通过静态的方式执行，不用执行其中的嵌入方法。

```
public Object proceed() throws Throwable {
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
        return invokeJoinpoint();
    }
    //currentInterceptorIndex 用来递归执行 MethodInvocation.proceed,从而进入到下一个拦截器
    Object interceptorOrInterceptionAdvice =
            this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
    if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
        //进行动态代理方法的匹配，如果匹配上了执行加入拦截器以后的方法，否则跳过拦截器执行下一个拦截器链的拦截器。
        InterceptorAndDynamicMethodMatcher dm =
                (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
        if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
            return dm.interceptor.invoke(this);
        }
        else {
            //动态匹配失败，跳过拦截器到拦截链的下一个拦截器
            return proceed();
        }
    }
    else {
        //只是一个拦截器就通过静态的方式执行不用执行其中嵌入的方法
        return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
    }
}
```

图 5 proceed

在 proceed 方法中无论是否匹配上动态代理方法都会运行 invoke，这里以以前置增强为例，实现类为 MethodBeforeAdviceInterceptor，如图 6 所示，在 override 的 invoke 方法中会执行 advice 中的 before 方法实现增强内容。

```
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, Serializable {

    private MethodBeforeAdvice advice;

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );
        return mi.proceed();
    }

}
```

图 6 invoke

这里把 JDK 的链式增强做一个总结，如图 7 所示，首先根据 Advisors 中的 Advice 创建 MethodInterceptor 拦截器链，这个链中保存对目标类对应方法的所有增强，这些增强以拦截器的方式放到拦截器链中。然后创建 MethodInvocation 用来保存上述的 MethodInterceptor 拦截器链。接着执行 MethodInvocation.proceed 的递归方法，在方法中通过 IterceptorIndex 对拦截器链上的拦截器进行遍历。在拦截器链中的逻辑都执行完毕了，通过 Invoke 执行目标类的原方法。

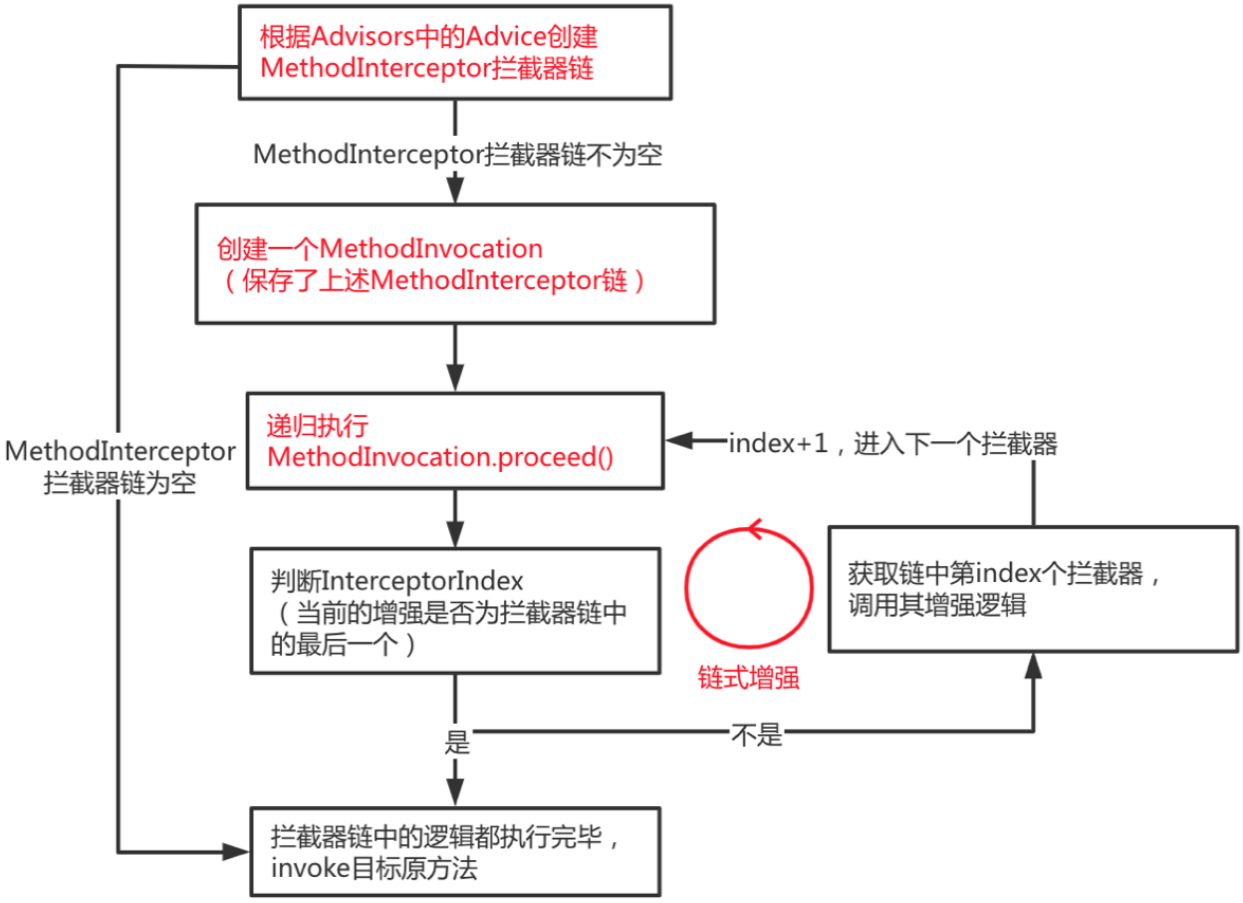

图 7 JDK 链式增强

3、getProxy：获取产生的代理类

上面说完了 invoke 方法中实现添加增强方法到目标类的过程，这里再说一下如果通过 getProxy 获取代理对原方法和增强方法执行的过程。如图 8 所示，getProxy 接受 ClassLoader 作为参数，方法体中调用了 AopProxyUtils 中的 completeProxiedInterfaces 生成代理接口，然后通过 Proxy 中的 newProxyInstance 方法将代理实例化。

```
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
    }
    Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

图 8 getProxy

再来看看 getProxy 中调用的 completeProxiedInterfaces 方法，

completeProxiedInterfaces 方法体中获取 AdvisedSupport 代理配置中目标类中需要被代理的接口

在当前目标类中，如果需要被代理的接口数为 0 的情况下做如下处理：

- 如果目标类它自身就是个接口，将其加到目标 Interface。
- 如果目标类本身已经是一个代理类（根据 Proxy.isProxyClass 判断），是 JDK 动态代理创建的代理对象，那么将其所有的接口都加到目标 Interface。
- 重新获取目标接口。

  addSpringProxy、addAdvised 用于判断目标类是否是特殊接口：SpringProxy、Advised。一般情况下两个 boolean 都是 true；

  将目标代理类、SpringProxy、Advised 三个接口填入 proxiedInterfaces 中，返回此数组。

```
public static Class<?>[] completeProxiedInterfaces(AdvisedSupport advised) {
    Class<?>[] specifiedInterfaces = advised.getProxiedInterfaces();
    if (specifiedInterfaces.length == 0) {
        // No user-specified interfaces: check whether target class is an interface.
        Class<?> targetClass = advised.getTargetClass();
        if (targetClass != null) {
            if (targetClass.isInterface()) {
                advised.setInterfaces(targetClass);
            }
            else if (Proxy.isProxyClass(targetClass)) {
                advised.setInterfaces(targetClass.getInterfaces());
            }
            specifiedInterfaces = advised.getProxiedInterfaces();
        }
    }
    boolean addSpringProxy = !advised.isInterfaceProxied(SpringProxy.class);
    boolean addAdvised = !advised.isOpaque() && !advised.isInterfaceProxied(Advised.class);
    int nonUserIfcCount = 0;
    if (addSpringProxy) {
        nonUserIfcCount++;
    }
    if (addAdvised) {
        nonUserIfcCount++;
    }
    Class<?>[] proxiedInterfaces = new Class<?>[specifiedInterfaces.length + nonUserIfcCount];
    System.arraycopy(specifiedInterfaces, 0, proxiedInterfaces, 0, specifiedInterfaces.length);
    if (addSpringProxy) {
        proxiedInterfaces[specifiedInterfaces.length] = SpringProxy.class;
    }
    if (addAdvised) {
        proxiedInterfaces[proxiedInterfaces.length - 1] = Advised.class;
    }
    return proxiedInterfaces;
}
```

图 9 completeProxiedInterfaces

4、总结

本节课针对 Spring AOP JDK 的主题，分析对应的源代码。从 invoke 实现 AOP 中具体的逻辑和 getProxy 获取产生的代理类两个方面进行了分析。下节课依旧是源码分析，分析的对象是另一种 Spring AOP 代理方式： CGLIB 动态代理。下期见，拜拜。