【有道云笔记】14-Spring之AOP底层源码解析（上）.md
https://note.youdao.com/s/2OEFGhEt

动态代理

代理模式的解释：为**其他对象**提供一种**代理**以控制对这个对象的访问，增强一个类中的某个方法，对程序进行扩展。

比如，现在存在一个UserService类：

```
public class UserService  {

 public void test() {
  System.out.println("test...");
 }

}
```

此时，我们new一个UserService对象，然后执行test()方法，结果是显而易见的。

如果我们现在想在**不修改UserService类的源码**前提下，给test()增加额外逻辑，那么就可以使用动态代理机制来创建UserService对象了，比如：

```
UserService target = new UserService();

// 通过cglib技术
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(UserService.class);

// 定义额外逻辑，也就是代理逻辑
enhancer.setCallbacks(new Callback[]{new MethodInterceptor() {
 @Override
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
  System.out.println("before...");
  Object result = methodProxy.invoke(target, objects);
  System.out.println("after...");
  return result;
 }
}});

// 动态代理所创建出来的UserService对象
UserService userService = (UserService) enhancer.create();

// 执行这个userService的test方法时，就会额外会执行一些其他逻辑
userService.test();
```

得到的都是UserService对象，但是执行test()方法时的效果却不一样了，这就是代理所带来的效果。

上面是通过cglib来实现的代理对象的创建，是基于**父子类**的，被代理类（UserService）是父类，代理类是子类，代理对象就是代理类的实例对象，代理类是由cglib创建的，对于程序员来说不用关心。

除开cglib技术，jdk本身也提供了一种创建代理对象的动态代理机制，但是它只能代理接口，也就是UserService得先有一个接口才能利用jdk动态代理机制来生成一个代理对象，比如：

```
public interface UserInterface {
 public void test();
}

public class UserService implements UserInterface {

 public void test() {
  System.out.println("test...");
 }

}
```

利用JDK动态代理来生成一个代理对象：

```
UserService target = new UserService();

// UserInterface接口的代理对象
Object proxy = Proxy.newProxyInstance(UserService.class.getClassLoader(), new Class[]
{UserInterface.class}, new InvocationHandler() {
 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
  System.out.println("before...");
  Object result = method.invoke(target, args);
  System.out.println("after...");
  return result;
 }
});

UserInterface userService = (UserInterface) proxy;
userService.test();
```

如果你把new Class[]{UserInterface.class}，替换成new Class[]{UserService.class}，允许代码会直接报错：

```
Exception in thread "main" java.lang.IllegalArgumentException: com.zhouyu.service.UserService is
not an interface
```

表示一定要是个接口。

由于这个限制，所以产生的代理对象的类型是UserInterface，而不是UserService，这是需要注意的。

ProxyFactory

上面我们介绍了两种动态代理技术，那么在Spring中进行了封装，封装出来的类叫做ProxyFactory，表示是创建代理对象的一个工厂，使用起来会比上面的更加方便，比如：

```
UserService target = new UserService();

ProxyFactory proxyFactory = new ProxyFactory();
proxyFactory.setTarget(target);
proxyFactory.addAdvice(new MethodInterceptor() {
 @Override
 public Object invoke(MethodInvocation invocation) throws Throwable {
  System.out.println("before...");
  Object result = invocation.proceed();
  System.out.println("after...");
  return result;
 }
});

UserInterface userService = (UserInterface) proxyFactory.getProxy();
userService.test();
```

通过ProxyFactory，我们可以不再关系到底是用cglib还是jdk动态代理了，ProxyFactory会帮我们去判断，如果UserService实现了接口，那么ProxyFactory底层就会用jdk动态代理，如果没有实现接口，就会用cglib技术，上面的代码，就是由于UserService实现了UserInterface接口，所以最后产生的代理对象是UserInterface类型。

Advice的分类

1. Before Advice：方法之前执行
2. After returning advice：方法return后执行
3. After throwing advice：方法抛异常后执行
4. After (finally) advice：方法执行完finally之后执行，这是最后的，比return更后
5. Around advice：这是功能最强大的Advice，可以自定义执行顺序

看课上给的代码例子将一目了然

Advisor的理解

跟Advice类似的还有一个Advisor的概念，一个Advisor是有一个Pointcut和一个Advice组成的，通过Pointcut可以指定要需要被代理的逻辑，比如一个UserService类中有两个方法，按上面的例子，这两个方法都会被代理，被增强，那么我们现在可以通过Advisor，来控制到具体代理哪一个方法，比如：

```
UserService target = new UserService();

ProxyFactory proxyFactory = new ProxyFactory();
proxyFactory.setTarget(target);
proxyFactory.addAdvisor(new PointcutAdvisor() {
 @Override
 public Pointcut getPointcut() {
  return new StaticMethodMatcherPointcut() {
   @Override
   public boolean matches(Method method, Class<?> targetClass) {
    return method.getName().equals("testAbc");
   }
  };
 }

 @Override
 public Advice getAdvice() {
  return new MethodInterceptor() {
   @Override
   public Object invoke(MethodInvocation invocation) throws Throwable {
    System.out.println("before...");
    Object result = invocation.proceed();
    System.out.println("after...");
    return result;
   }
  };
 }

 @Override
 public boolean isPerInstance() {
  return false;
 }
});

UserInterface userService = (UserInterface) proxyFactory.getProxy();
userService.test();
```

上面代码表示，产生的代理对象，只有在执行testAbc这个方法时才会被增强，会执行额外的逻辑，而在执行其他方法时是不会增强的。

创建代理对象的方式

上面介绍了Spring中所提供了ProxyFactory、Advisor、Advice、PointCut等技术来实现代理对象的创建，但是我们在使用Spring时，我们并不会直接这么去使用ProxyFactory，比如说，我们希望ProxyFactory所产生的代理对象能直接就是Bean，能直接从Spring容器中得到UserSerivce的代理对象，而这些，Spring都是支持的，只不过，作为开发者的我们肯定得告诉Spring，那些类需要被代理，代理逻辑是什么。

ProxyFactoryBean

```
@Bean
public ProxyFactoryBean userServiceProxy(){
 UserService userService = new UserService();

 ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
 proxyFactoryBean.setTarget(userService);
 proxyFactoryBean.addAdvice(new MethodInterceptor() {
  @Override
  public Object invoke(MethodInvocation invocation) throws Throwable {
   System.out.println("before...");
   Object result = invocation.proceed();
   System.out.println("after...");
   return result;
  }
 });
 return proxyFactoryBean;
}
```

通过这种方法来定义一个UserService的Bean，并且是经过了AOP的。但是这种方式**只能针对某一个Bean。**它是一个FactoryBean，所以利用的就是FactoryBean技术，间接的将UserService的代理对象作为了Bean。

ProxyFactoryBean还有额外的功能，比如可以把某个Advise或Advisor定义成为Bean，然后在ProxyFactoryBean中进行设置

```
@Bean
public MethodInterceptor zhouyuAroundAdvise(){
 return new MethodInterceptor() {
  @Override
  public Object invoke(MethodInvocation invocation) throws Throwable {
   System.out.println("before...");
   Object result = invocation.proceed();
   System.out.println("after...");
   return result;
  }
 };
}

@Bean
public ProxyFactoryBean userService(){
 UserService userService = new UserService();

    ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
 proxyFactoryBean.setTarget(userService);
 proxyFactoryBean.setInterceptorNames("zhouyuAroundAdvise");
 return proxyFactoryBean;
}
```

BeanNameAutoProxyCreator

ProxyFactoryBean得自己指定被代理的对象，那么我们可以通过BeanNameAutoProxyCreator来通过指定某个bean的名字，来对该bean进行代理

```
@Bean
public BeanNameAutoProxyCreator beanNameAutoProxyCreator() {
 BeanNameAutoProxyCreator beanNameAutoProxyCreator = new BeanNameAutoProxyCreator();
 beanNameAutoProxyCreator.setBeanNames("userSe*");
 beanNameAutoProxyCreator.setInterceptorNames("zhouyuAroundAdvise");
 beanNameAutoProxyCreator.setProxyTargetClass(true);

    return beanNameAutoProxyCreator;
}
```

通过BeanNameAutoProxyCreator可以对批量的Bean进行AOP，并且指定了代理逻辑，指定了一个InterceptorName，也就是一个Advise，前提条件是这个Advise也得是一个Bean，这样Spring才能找到的，但是BeanNameAutoProxyCreator的缺点很明显，它只能根据beanName来指定想要代理的Bean。

DefaultAdvisorAutoProxyCreator

```
@Bean
public DefaultPointcutAdvisor defaultPointcutAdvisor(){
 NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
 pointcut.addMethodName("test");

    DefaultPointcutAdvisor defaultPointcutAdvisor = new DefaultPointcutAdvisor();
 defaultPointcutAdvisor.setPointcut(pointcut);
 defaultPointcutAdvisor.setAdvice(new ZhouyuAfterReturningAdvise());

    return defaultPointcutAdvisor;
}

@Bean
public DefaultAdvisorAutoProxyCreator defaultAdvisorAutoProxyCreator() {

    DefaultAdvisorAutoProxyCreator defaultAdvisorAutoProxyCreator = new DefaultAdvisorAutoProxyCre
ator();

 return defaultAdvisorAutoProxyCreator;
}
```

通过DefaultAdvisorAutoProxyCreator会直接去找所有Advisor类型的Bean，根据Advisor中的PointCut和Advice信息，确定要代理的Bean以及代理逻辑。

但是，我们发现，通过这种方式，我们得依靠某一个类来实现定义我们的Advisor，或者Advise，或者Pointcut，那么这个步骤能不能更加简化一点呢？

对的，通过**注解**！

比如我们能不能只定义一个类，然后通过在类中的方法上通过某些注解，来定义PointCut以及Advice，可以的，比如：

```
@Aspect
@Component
public class ZhouyuAspect {

 @Before("execution(public void com.zhouyu.service.UserService.test())")
 public void zhouyuBefore(JoinPoint joinPoint) {
  System.out.println("zhouyuBefore");
 }

}
```

通过上面这个类，我们就直接定义好了所要代理的方法(通过一个表达式)，以及代理逻辑（被@Before修饰的方法），简单明了，这样对于Spring来说，它要做的就是来解析这些注解了，解析之后得到对应的Pointcut对象、Advice对象，生成Advisor对象，扔进ProxyFactory中，进而产生对应的代理对象，具体怎么解析这些注解就是**@EnableAspectJAutoProxy注解**所要做的事情了，后面详细分析。

对Spring AOP的理解

OOP表示面向对象编程，是一种编程思想，AOP表示面向切面编程，也是一种编程思想，而我们上面所描述的就是Spring为了让程序员更加方便的做到面向切面编程所提供的技术支持，换句话说，就是Spring提供了一套机制，可以让我们更加容易的来进行AOP，所以这套机制我们也可以称之为Spring AOP。

但是值得注意的是，上面所提供的注解的方式来定义Pointcut和Advice，Spring并不是首创，首创是AspectJ，而且也不仅仅只有Spring提供了一套机制来支持AOP，还有比如 JBoss 4.0、aspectwerkz等技术都提供了对于AOP的支持。而刚刚说的注解的方式，Spring是依赖了AspectJ的，或者说，Spring是直接把AspectJ中所定义的那些注解直接拿过来用，自己没有再重复定义了，不过也仅仅只是把注解的定义赋值过来了，每个注解具体底层是怎么解析的，还是Spring自己做的，所以我们在用Spring时，如果你想用@Before、@Around等注解，是需要单独引入aspecj相关jar包的，比如：

```
compile group: 'org.aspectj', name: 'aspectjrt', version: '1.9.5'
compile group: 'org.aspectj', name: 'aspectjweaver', version: '1.9.5'
```

值得注意的是：AspectJ是在编译时对字节码进行了修改，是直接在UserService类对应的字节码中进行增强的，也就是可以理解为是在编译时就会去解析@Before这些注解，然后得到代理逻辑，加入到被代理的类中的字节码中去的，所以如果想用AspectJ技术来生成代理对象 ，是需要用单独的AspectJ编译器的。我们在项目中很少这么用，我们仅仅只是用了@Before这些注解，而我们在启动Spring的过程中，Spring会去解析这些注解，然后利用动态代理机制生成代理对象的。

IDEA中使用Aspectj：https://blog.csdn.net/gavin_john/article/details/80156963

AOP中的概念

上面我们已经提到Advisor、Advice、PointCut等概念了，还有一些其他的概念，首先关于AOP中的概念本身是比较难理解的，Spring官网上是这么说的：

> Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring used its own terminology

意思是，AOP中的这些概念不是Spring特有的，不幸的是，AOP中的概念不是特别直观的，但是，如果Spring重新定义自己的那可能会导致更加混乱

1. Aspect：表示切面，比如被@Aspect注解的类就是切面，可以在切面中去定义Pointcut、Advice等等
2. Join point：表示连接点，表示一个程序在执行过程中的一个点，比如一个方法的执行，比如一个异常的处理，在Spring AOP中，一个连接点通常表示一个方法的执行。
3. Advice：表示通知，表示在一个特定连接点上所采取的动作。Advice分为不同的类型，后面详细讨论，在很多AOP框架中，包括Spring，会用Interceptor拦截器来实现Advice，并且在连接点周围维护一个Interceptor链
4. Pointcut：表示切点，用来匹配一个或多个连接点，Advice与切点表达式是关联在一起的，Advice将会执行在和切点表达式所匹配的连接点上
5. Introduction：可以使用@DeclareParents来给所匹配的类添加一个接口，并指定一个默认实现
6. Target object：目标对象，被代理对象
7. AOP proxy：表示代理工厂，用来创建代理对象的，在Spring Framework中，要么是JDK动态代理，要么是CGLIB代理
8. Weaving：表示织入，表示创建代理对象的动作，这个动作可以发生在编译时期（比如Aspejctj），或者运行时，比如Spring AOP

Advice在Spring AOP中对应API

上面说到的Aspject中的注解，其中有五个是用来定义Advice的，表示代理逻辑，以及执行时机：

1. @Before
2. @AfterReturning
3. @AfterThrowing
4. @After
5. @Around

我们前面也提到过，Spring自己也提供了类似的执行实际的实现类：

1. 接口MethodBeforeAdvice，继承了接口BeforeAdvice
2. 接口AfterReturningAdvice
3. 接口ThrowsAdvice
4. 接口AfterAdvice
5. 接口MethodInterceptor

Spring会把五个注解解析为对应的Advice类：

1. @Before：AspectJMethodBeforeAdvice，实际上就是一个MethodBeforeAdvice
2. @AfterReturning：AspectJAfterReturningAdvice，实际上就是一个AfterReturningAdvice
3. @AfterThrowing：AspectJAfterThrowingAdvice，实际上就是一个MethodInterceptor
4. @After：AspectJAfterAdvice，实际上就是一个MethodInterceptor
5. @Around：AspectJAroundAdvice，实际上就是一个MethodInterceptor

TargetSource的使用

在我们日常的AOP中，被代理对象就是Bean对象，是由BeanFactory给我们创建出来的，但是Spring AOP中提供了TargetSource机制，可以让我们用来自定义逻辑来创建**被代理对象。**

比如之前所提到的@Lazy注解，当加在属性上时，会产生一个代理对象赋值给这个属性，产生代理对象的代码为：

```
protected Object buildLazyResolutionProxy(final DependencyDescriptor
descriptor, final @Nullable String beanName) {
  BeanFactory beanFactory = getBeanFactory();
  Assert.state(beanFactory instanceof DefaultListableBeanFactory,
    "BeanFactory needs to be a DefaultListableBeanFactory");
  final DefaultListableBeanFactory dlbf = (DefaultListableBeanFactory) beanFactory;

  TargetSource ts = new TargetSource() {
   @Override
   public Class<?> getTargetClass() {
    return descriptor.getDependencyType();
   }
   @Override
   public boolean isStatic() {
    return false;
   }
   @Override
   public Object getTarget() {
    Set<String> autowiredBeanNames = (beanName != null ? new LinkedHashSet<>(1) : null);
    Object target = dlbf.doResolveDependency(descriptor, beanName, autowiredBeanNames, null);
    if (target == null) {
     Class<?> type = getTargetClass();
     if (Map.class == type) {
      return Collections.emptyMap();
     }
     else if (List.class == type) {
      return Collections.emptyList();
     }
     else if (Set.class == type || Collection.class == type) {
      return Collections.emptySet();
     }
     throw new NoSuchBeanDefinitionException(descriptor.getResolvableType(),
       "Optional dependency not present for lazy injection point");
    }
    if (autowiredBeanNames != null) {
     for (String autowiredBeanName : autowiredBeanNames) {
      if (dlbf.containsBean(autowiredBeanName)) {
       dlbf.registerDependentBean(autowiredBeanName, beanName);
      }
     }
    }
    return target;
   }
   @Override
   public void releaseTarget(Object target) {
   }
  };

  ProxyFactory pf = new ProxyFactory();
  pf.setTargetSource(ts);
  Class<?> dependencyType = descriptor.getDependencyType();
  if (dependencyType.isInterface()) {
   pf.addInterface(dependencyType);
  }
  return pf.getProxy(dlbf.getBeanClassLoader());
 }
```

这段代码就利用了ProxyFactory来生成代理对象，以及使用了TargetSource，以达到代理对象在执行某个方法时，调用TargetSource的getTarget()方法实时得到一个**被代理对象**。

Introduction

https://www.cnblogs.com/powerwu/articles/5170861.html

LoadTimeWeaver

https://www.cnblogs.com/davidwang456/p/5633609.html