回首页

Spring AOP 源码解析

创建时间: 2018-07-24 00:00:00

[TOC]

之前写过 IOC 的源码分析，那篇文章真的有点长，看完需要点耐心。很多读者希望能写一写 Spring AOP 的源码分析文章，这样读者看完 IOC + AOP 也就对 Spring 会有比较深的理解了。今天终于成文了，可能很多读者早就不再等待了，不过主要为了后来者吧。

本文不会像 IOC 源码分析那篇文章一样，很具体地分析每一行 Spring AOP 的源码，目标读者是已经知道 Spring IOC 源码是怎么回事的读者，因为 Spring AOP 终归是依赖于 IOC 容器来管理的。

阅读建议：1、先搞懂 IOC 容器的源码，AOP 依赖于 IOC 容器来管理。2、仔细看完 Spring AOP 使用介绍 这篇文章，先搞懂各种使用方式，你才能"猜到"应该怎么实现。

Spring AOP 的源码并不简单，因为它多，所以阅读源码最好就是找到一个分支，追踪下去。**本文定位为走马观花，看个大概，不具体到每一个细节。**

目录：

前言

这一节，我们先来"猜猜" Spring 是怎么实现 AOP 的。

在 Spring 的容器中，我们面向的对象是一个个的 bean 实例，bean 是什么？我们可以简单理解为是 BeanDefinition 的实例，Spring 会根据 BeanDefinition 中的信息为我们生产合适的 bean 实例出来。

当我们需要使用 bean 的时候，通过 IOC 容器的 getBean(...) 方法从容器中获取 bean 实例，只不过大部分的场景下，我们都用了依赖注入，所以很少手动调用 getBean(...) 方法。

Spring AOP 的原理很简单，就是**动态代理**，它和 AspectJ 不一样，AspectJ 是直接修改掉你的字节码。

代理模式很简单，接口 + 真实实现类 + 代理类，其中 真实实现类 和 代理类 都要实现接口，实例化的时候要使用代理类。所以，Spring AOP 需要做的是生成这么一个代理类，然后**替换掉**真实实现类来对外提供服务。

替换的过程怎么理解呢？在 Spring IOC 容器中非常容易实现，就是在 getBean(...) 的时候返回的实际上是代理类的实例，而这个代理类我们自己没写代码，它是 Spring 采用 JDK Proxy 或 CGLIB 动态生成的。

> getBean(...) 方法用于查找或实例化容器中的 bean，这也是为什么 Spring AOP 只能作用于 Spring 容器中的 bean 的原因，对于不是使用 IOC 容器管理的对象，Spring AOP 是无能为力的。

本文使用的调试代码

阅读源码很好用的一个方法就是跑代码来调试，因为自己一行一行地看的话，比较枯燥，而且难免会漏掉一些东西。

下面，我们先准备一些简单的调试用的代码。

首先先定义两个 Service 接口：

```
// OrderService.java
public interface OrderService {

    Order createOrder(String username, String product);

    Order queryOrder(String username);
}
// UserService.java
public interface UserService {

    User createUser(String firstName, String lastName, int age);

    User queryUser();
}
```

然后，分别来一个接口实现类：

```
// OrderServiceImpl.java
public class OrderServiceImpl implements OrderService {

    @Override
```

```
    public Order createOrder(String username, String product) {
        Order order = new Order();
        order.setUsername(username);
        order.setProduct(product);
        return order;
    }

    @Override
    public Order queryOrder(String username) {
        Order order = new Order();
        order.setUsername("test");
        order.setProduct("test");
        return order;
    }
}

// UserServiceImpl.java
public class UserServiceImpl implements UserService {

    @Override
    public User createUser(String firstName, String lastName, int age) {
        User user = new User();
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setAge(age);
        return user;
    }

    @Override
    public User queryUser() {
        User user = new User();
        user.setFirstName("test");
        user.setLastName("test");
        user.setAge(20);
        return user;
    }
}
```

写两个 Advice：

```
public class LogArgsAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Thro
        System.out.println("准备执行方法：" + method.getName() + ", 参数列表：" + A
    }
}

public class LogResultAdvice implements AfterReturningAdvice {

    @Override
    public void afterReturning(Object returnValue, Method method, Object[] args
            throws Throwable {
        System.out.println(method.getName() + "方法返回：" + returnValue);
    }
}
```

配置一下：

```
<bean id="userServiceImpl" class="com.javadoop.springaopsource.service.imple.UserServiceImpl"/>
<bean id="orderServiceImpl" class="com.javadoop.springaopsource.service.imple.OrderServiceImpl"/>

<!--定义两个 advice-->
<bean id="logArgsAdvice" class="com.javadoop.springaopsource.advice.LogArgsAdvice"/>
<bean id="logResultAdvice" class="com.javadoop.springaopsource.advice.LogResultAdvice"/>

<!--定义两个 advisor-->
<!--记录 create* 方法的传参-->
<bean id="logArgsAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice" ref="logArgsAdvice" />
    <property name="pattern" value="com.javadoop.springaopsource.service.*.create.*" />
</bean>
<!--记录 query* 的返回值-->
<bean id="logResultAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice" ref="logResultAdvice" />
    <property name="pattern" value="com.javadoop.springaopsource.service.*.query.*" />
</bean>


<!--定义DefaultAdvisorAutoProxyCreator-->
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```

> 我们这边使用了前面文章介绍的配置 Advisor 的方式，我们回顾一下。

> 每个 advisor 内部持有 advice 实例，advisor 负责匹配，内部的 advice 负责实现拦截处理。配置了各个 advisor 后，配置 DefaultAdvisorAutoProxyCreator 使得所有的 advisor 配置自动生效。

启动：

```
public class SpringAopSourceApplication {

    public static void main(String[] args) {

        // 启动 Spring 的 IOC 容器
        ApplicationContext context = new ClassPathXmlApplicationContext("classpat

        UserService userService = context.getBean(UserService.class);
        OrderService orderService = context.getBean(OrderService.class);

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();

        orderService.createOrder("Leo", "随便买点什么");
        orderService.queryOrder("Leo");
    }
}
```

输出：

```
准备执行方法：createUser, 参数列表：[Tom, Cruise, 55]
queryUser方法返回：User{firstName='test', lastName='test', age=20, address='null
准备执行方法：createOrder, 参数列表：[Leo, 随便买点什么]
queryOrder方法返回：Order{username='test', product='test'}
```

> 从输出结果，我们可以看到：
>
> LogArgsAdvice 作用于 UserService#createUser(...) 和 OrderService#createOrder(...) 两个方法；
>
> LogResultAdvice 作用于 UserService#queryUser() 和 OrderService#queryOrder(...) 两个方法；

下面的代码分析中，我们将基于这个简单的例子来介绍。

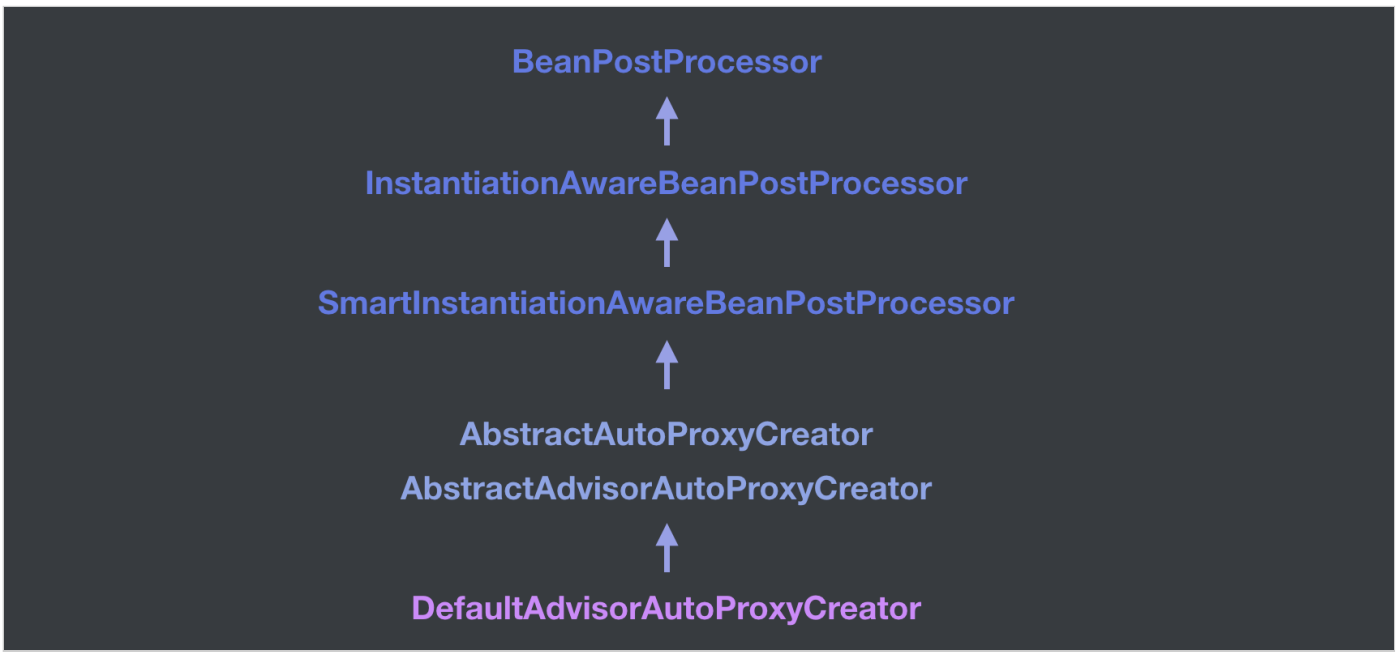
IOC 容器管理 AOP 实例

本节介绍 Spring AOP 是怎么作用于 IOC 容器中的 bean 的。

Spring AOP 的使用介绍 那篇文章已经介绍过 DefaultAdvisorAutoProxyCreator 类了，它能实现自动将所有的 advisor 生效。

我们来追踪下 DefaultAdvisorAutoProxyCreator 类，看看它是怎么一步步实现的动态代理。然后在这个基础上，我们再简单追踪下 @AspectJ 配置方式下的源码实现。

首先，先看下 DefaultAdvisorAutoProxyCreator 的继承结构：



我们可以发现，DefaultAdvisorAutoProxyCreator 最后居然是一个 **BeanPostProcessor**，在 Spring IOC 源码分析的时候说过，BeanPostProcessor 的两个方法，分别在 init-method 的前后得到执行。

```
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName) throws
    Object postProcessAfterInitialization(Object bean, String beanName) throws
}
```

这里再贴一下 IOC 的源码，我们回顾一下：

// AbstractAutowireCapableBeanFactory

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition m
            throws BeanCreationException {

    // Instantiate the bean.
    BeanWrapper instanceWrapper = null;

    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
        // 1. 创建实例
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    ...

    // Initialize the bean instance.
    Object exposedObject = bean;
    try {
        // 2. 装载属性
        populateBean(beanName, mbd, instanceWrapper);
        if (exposedObject != null) {
            // 3. 初始化
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
    }
    ...
}
```

在上面第 3 步 initializeBean(...) 方法中会调用 BeanPostProcessor 中的方法，如下：

```
protected Object initializeBean(final String beanName, final Object bean, RootB
    ...
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        // 1. 执行每一个 BeanPostProcessor 的 postProcessBeforeInitialization 方法
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, be
    }

    try {
```

```
            try {
                // 调用 bean 配置中的 init-method="xxx"
                invokeInitMethods(beanName, wrappedBean, mbd);
            }
            ...
            if (mbd == null || !mbd.isSynthetic()) {
                // 我们关注的重点是这里！！！
                // 2. 执行每一个 BeanPostProcessor 的 postProcessAfterInitialization 方法
                wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, bea
            }
            return wrappedBean;
        }
```

也就是说，Spring AOP 会在 IOC 容器创建 bean 实例的最后对 bean 进行处理。其实就是在这一步进行代理增强。

我们回过头来，DefaultAdvisorAutoProxyCreator 的继承结构中，postProcessAfterInitialization() 方法在其父类 AbstractAutoProxyCreator 这一层被覆写了：

// AbstractAutoProxyCreator

```
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throc
        if (bean != null) {

            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }
```

继续往里看 wrapIfNecessary(...) 方法，这个方法将返回代理类（如果需要的话）：

```
    protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey)
        if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
            return bean;
        }
```

```
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
            return bean;
        }
        if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), be
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }

        // 返回匹配当前 bean 的所有的 advisor、advice、interceptor
        // 对于本文的例子，"userServiceImpl" 和 "OrderServiceImpl" 这两个 bean 创建过程中，
        //   到这边的时候都会返回两个 advisor
        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass()
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            // 创建代理...创建代理...创建代理...
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new SingletonTarge
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
```

这里有两个点提一下：

getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null)，这个方法将得到所有的**可用于拦截当前 bean 的** advisor、advice、interceptor。

另一个就是 TargetSource 这个概念，它用于封装真实实现类的信息，上面用了 SingletonTargetSource 这个实现类，其实我们这里也不太需要关心这个，知道有这么回事就可以了。

我们继续往下看 createProxy(...) 方法：

```
    // 注意看这个方法的几个参数，
    //   第三个参数携带了所有的 advisors
    //   第四个参数 targetSource 携带了真实实现的信息
```

```
protected Object createProxy(
        Class<?> beanClass, String beanName, Object[] specificInterceptors, Targe

    if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
        AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.b
    }

    // 创建 ProxyFactory 实例

    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.copyFrom(this);

    // 在 schema-based 的配置方式中，我们介绍过，如果希望使用 CGLIB 来代理接口，可以配置
    // proxy-target-class="true",这样不管有没有接口，都使用 CGLIB 来生成代理：
    //   <aop:config proxy-target-class="true">......</aop:config>
    if (!proxyFactory.isProxyTargetClass()) {
        if (shouldProxyTargetClass(beanClass, beanName)) {
            proxyFactory.setProxyTargetClass(true);
        }
        else {
            // 点进去稍微看一下代码就知道了，主要就两句：
            // 1. 有接口的，调用一次或多次：proxyFactory.addInterface(ifc);
            // 2. 没有接口的，调用：proxyFactory.setProxyTargetClass(true);
            evaluateProxyInterfaces(beanClass, proxyFactory);
        }
    }

    // 这个方法会返回匹配了当前 bean 的 advisors 数组
    // 对于本文的例子，"userServiceImpl" 和 "OrderServiceImpl" 到这边的时候都会返回两个
    // 注意：如果 specificInterceptors 中有 advice 和 interceptor，它们也会被包装成 ad
    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    for (Advisor advisor : advisors) {
        proxyFactory.addAdvisor(advisor);
    }

    proxyFactory.setTargetSource(targetSource);
    customizeProxyFactory(proxyFactory);

    proxyFactory.setFrozen(this.freezeProxy);
```

```
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }

        return proxyFactory.getProxy(getProxyClassLoader());
    }
```

我们看到，这个方法主要是在内部创建了一个 ProxyFactory 的实例，然后 set 了一大堆内容，剩下的工作就都是这个 ProxyFactory 实例的了，通过这个实例来创建代理：`getProxy(classLoader)`。

ProxyFactory 详解

根据上面的源码，我们走到了 ProxyFactory 这个类了，我们到这个类来一看究竟。

顺着上面的路子，我们首先到 ProxyFactory#getProxy(classLoader) 方法：

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

该方法首先通过 createAopProxy() 创建一个 AopProxy 的实例：

```
    protected final synchronized AopProxy createAopProxy() {
        if (!this.active) {
            activate();
        }
        return getAopProxyFactory().createAopProxy(this);
    }
```

创建 AopProxy 之前，我们需要一个 AopProxyFactory 实例，然后看 ProxyCreatorSupport 的构造方法：

```
    public ProxyCreatorSupport() {
        this.aopProxyFactory = new DefaultAopProxyFactory();
    }
```

这样就将我们导到 DefaultAopProxyFactory 这个类了，我们看它的 createAopProxy(...) 方法：

```
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigExcept
        // (我也没用过这个optimize，默认false) || (proxy-target-class=true) || (没有指
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppli
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target
                        "Either an interface or a target is required for proxy creati
            }
            // 如果要代理的类本身就是接口，也会用 JDK 动态代理
            // 我也没用过这个。。。
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            // 如果有接口，会跑到这个分支
            return new JdkDynamicAopProxy(config);
        }
    }
    // 判断是否有实现自定义的接口
    private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
        Class<?>[] ifcs = config.getProxiedInterfaces();
        return (ifcs.length == 0 || (ifcs.length == 1 && SpringProxy.class.isAssi
    }

}
```

到这里，我们知道 createAopProxy 方法有可能返回 JdkDynamicAopProxy 实例，也有可能返回 ObjenesisCglibAopProxy 实例，这里总结一下：

如果被代理的目标类实现了一个或多个自定义的接口，那么就会使用 JDK 动态代理，如果没有实现任何接口，会使用 CGLIB 实现代理，如果设置了 proxy-target-class="true"，那么都会使用 CGLIB。

JDK 动态代理基于接口，所以只有接口中的方法会被增强，而 CGLIB 基于类继承，需要注意就是如果方法使用了 final 修饰，或者是 private 方法，是不能被增强的。

有了 AopProxy 实例以后，我们就回到这个方法了：

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

我们分别来看下两个 AopProxy 实现类的 getProxy(classLoader) 实现。

JdkDynamicAopProxy 类的源码比较简单，总共两百多行，

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

java.lang.reflect.Proxy.newProxyInstance(...) 方法需要三个参数，第一个是 ClassLoader，第二个参数代表需要实现哪些接口，第三个参数最重要，是 InvocationHandler 实例，我们看到这里传了 this，因为 JdkDynamicAopProxy 本身实现了 InvocationHandler 接口。

InvocationHandler 只有一个方法，当生成的代理类对外提供服务的时候，都会导到这个方法中：

```
public interface InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

下面来看看 JdkDynamicAopProxy 对其的实现：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwab
   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)) {
         // The target does not implement the equals(Object) method itself.
         // 代理的 equals 方法
         return equals(args[0]);
      }
      else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
         // The target does not implement the hashCode() method itself.
         // 代理的 hashCode 方法
         return hashCode();
      }
      else if (method.getDeclaringClass() == DecoratingProxy.class) {
         // There is only getDecoratedClass() declared -> dispatch to proxy con
         //
         return AopProxyUtils.ultimateTargetClass(this.advised);
      }
      else if (!this.advised.opaque && method.getDeclaringClass().isInterface()
            method.getDeclaringClass().isAssignableFrom(Advised.class)) {
         // Service invocations on ProxyConfig with the proxy config...
         return AopUtils.invokeJoinpointUsingReflection(this.advised, method, a
      }

      Object retVal;
```

```
// 如果设置了 exposeProxy，那么将 proxy 放到 ThreadLocal 中
if (this.advised.exposeProxy) {
   // Make invocation available if necessary.
   oldProxy = AopContext.setCurrentProxy(proxy);
   setProxyContext = true;
}

// May be null. Get as late as possible to minimize the time we "own" the
// in case it comes from a pool.
target = targetSource.getTarget();
if (target != null) {
   targetClass = target.getClass();
}

// Get the interception chain for this method.
// 创建一个 chain，包含所有要执行的 advice
List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAd

// Check whether we have any advice. If we don't, we can fallback on dire
// reflective invocation of the target, and avoid creating a MethodInvoca
if (chain.isEmpty()) {
   // We can skip creating a MethodInvocation: just invoke the target dir
   // Note that the final invoker must be an InvokerInterceptor so we knc
   // nothing but a reflective operation on the target, and no hot swappi
   // chain 是空的，说明不需要被增强，这种情况很简单
   Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, a
   retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsT
}
else {
   // We need to create a method invocation...
   // 执行方法，得到返回值
   invocation = new ReflectiveMethodInvocation(proxy, target, method, arg
   // Proceed to the joinpoint through the interceptor chain.
   retVal = invocation.proceed();
}

// Massage return value if necessary.
Class<?> returnType = method.getReturnType();
```

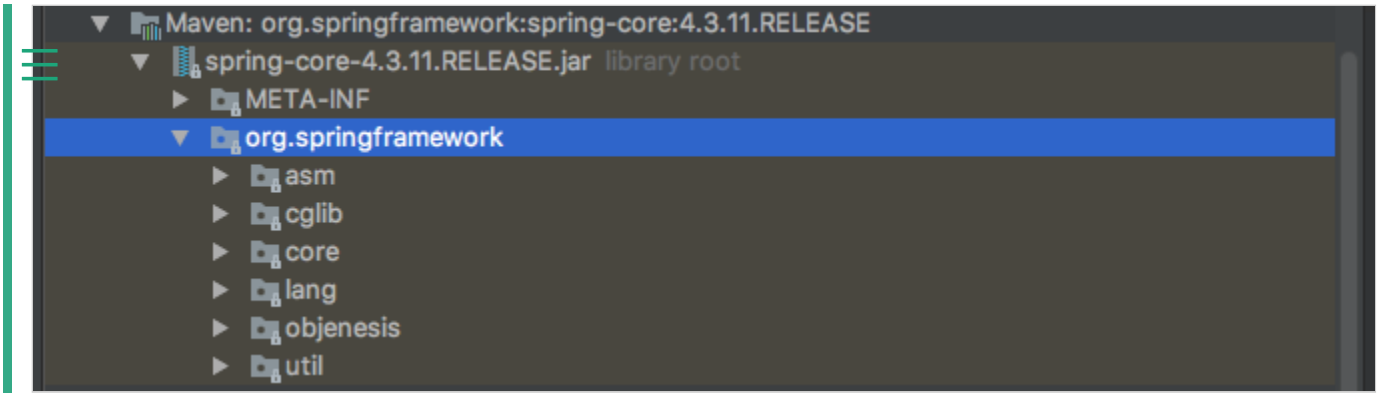
```
            if (retVal != null && retVal == target &&
                    returnType != Object.class && returnType.isInstance(proxy) &&
                    !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())
                // Special case: it returned "this" and the return type of the method
                // is type-compatible. Note that we can't help if the target sets
                // a reference to itself in another returned object.
                retVal = proxy;
            }
            else if (retVal == null && returnType != Void.TYPE && returnType.isPrimit
                throw new AopInvocationException(
                        "Null return value from advice does not match primitive return t
            }
            return retVal;
        }
        finally {
            if (target != null && !targetSource.isStatic()) {
                // Must have come from TargetSource.
                targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
                // Restore old proxy.
                AopContext.setCurrentProxy(oldProxy);
            }
        }
    }
```

> 上面就三言两语说了一下，感兴趣的读者自己去深入探索下，不是很难。简单地说，就是在执行每个方法的时候，判断下该方法是否需要被一次或多次增强（执行一个或多个 advice）。

说完了 JDK 动态代理 JdkDynamicAopProxy#getProxy(classLoader)，我们再来瞄一眼 CGLIB 的代理实现 ObjenesisCglibAopProxy#getProxy(classLoader)。

ObjenesisCglibAopProxy 继承了 CglibAopProxy，而 CglibAopProxy 继承了 AopProxy。

> ObjenesisCglibAopProxy 使用了 Objenesis 这个库，和 cglib 一样，我们不需要在 maven 中进行依赖，因为 spring-core.jar 直接把它的源代码也搞过来了。



通过 CGLIB 生成代理的代码量有点大，我们就不进行深入分析了，我们看下大体的骨架。它的 getProxy(classLoader) 方法在父类 CglibAopProxy 类中：

// CglibAopProxy#getProxy(classLoader)

```
@Override
public Object getProxy(ClassLoader classLoader) {
      ...
      // Configure CGLIB Enhancer...
      Enhancer enhancer = createEnhancer();
      if (classLoader != null) {
         enhancer.setClassLoader(classLoader);
         if (classLoader instanceof SmartClassLoader &&
               ((SmartClassLoader) classLoader).isClassReloadable(proxySuperCla
            enhancer.setUseCache(false);
         }
      }
      enhancer.setSuperclass(proxySuperClass);
      enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advis
      enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
      enhancer.setStrategy(new ClassLoaderAwareUndeclaredThrowableStrategy(clas

      Callback[] callbacks = getCallbacks(rootClass);

      Class<?>[] types = new Class<?>[callbacks.length];
      for (int x = 0; x < types.length; x++) {
         types[x] = callbacks[x].getClass();
      }
      // fixedInterceptorMap only populated at this point, after getCallbacks c
      enhancer.setCallbackFilter(new ProxyCallbackFilter(
            this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap,
```

```
            enhancer.setCallbackTypes(types);

            // Generate the proxy class and create a proxy instance.
            return createProxyClassAndInstance(enhancer, callbacks);
        }
        catch (CodeGenerationException ex) {
            ...
        }
        catch (IllegalArgumentException ex) {
            ...
        }
        catch (Throwable ex) {
            ...
        }
    }
```

CGLIB 生成代理的核心类是 Enhancer 类，这里就不展开说了。

基于注解的 Spring AOP 源码分析

上面我们走马观花地介绍了使用 DefaultAdvisorAutoProxyCreator 来实现 Spring AOP 的源码，这里，我们也同样走马观花地来看下 @AspectJ 的实现原理。

我们之前说过，开启 @AspectJ 的两种方式，一个是 `<aop:aspectj-autoproxy/>`，一个是 `@EnableAspectJAutoProxy`，它们的原理是一样的，都是通过注册一个 bean 来实现的。

解析 `<aop:aspectj-autoproxy/>` 需要用到 AopNamespaceHandler：

```
public class AopNamespaceHandler extends NamespaceHandlerSupport {

    /**
     * Register the {@link BeanDefinitionParser BeanDefinitionParsers} for the
     * '{@code config}', '{@code spring-configured}', '{@code aspectj-autoproxy}'
     * and '{@code scoped-proxy}' tags.
     */
    @Override
    public void init() {
        // In 2.0 XSD as well as in 2.1 XSD.
        registerBeanDefinitionParser( elementName: "config", new ConfigBeanDefinitionParser());
        registerBeanDefinitionParser( elementName: "aspectj-autoproxy", new AspectJAutoProxyBeanDefinitionParser());
        registerBeanDefinitionDecorator( elementName: "scoped-proxy", new ScopedProxyBeanDefinitionDecorator());

        // Only in 2.0 XSD: moved to context namespace as of 2.1
        registerBeanDefinitionParser( elementName: "spring-configured", new SpringConfiguredBeanDefinitionParser());
    }

}
```

然后到类 AspectJAutoProxyBeanDefinitionParser：

```
class AspectJAutoProxyBeanDefinitionParser implements BeanDefinitionParser {

    @Override
    @Nullable

    public BeanDefinition parse(Element element, ParserContext parserContext) {
        AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(pa
        extendBeanDefinition(element, parserContext);
        return null;
    }
    ...
}
```

进去 registerAspectJAnnotationAutoProxyCreatorIfNecessary(...) 方法：

```
public static void registerAspectJAnnotationAutoProxyCreatorIfNecessary(
        ParserContext parserContext, Element sourceElement) {

    BeanDefinition beanDefinition = AopConfigUtils.registerAspectJAnnotationAutc
            parserContext.getRegistry(), parserContext.extractSource(sourceElement
    useClassProxyingIfNecessary(parserContext.getRegistry(), sourceElement);
    registerComponentIfNecessary(beanDefinition, parserContext);
}
```

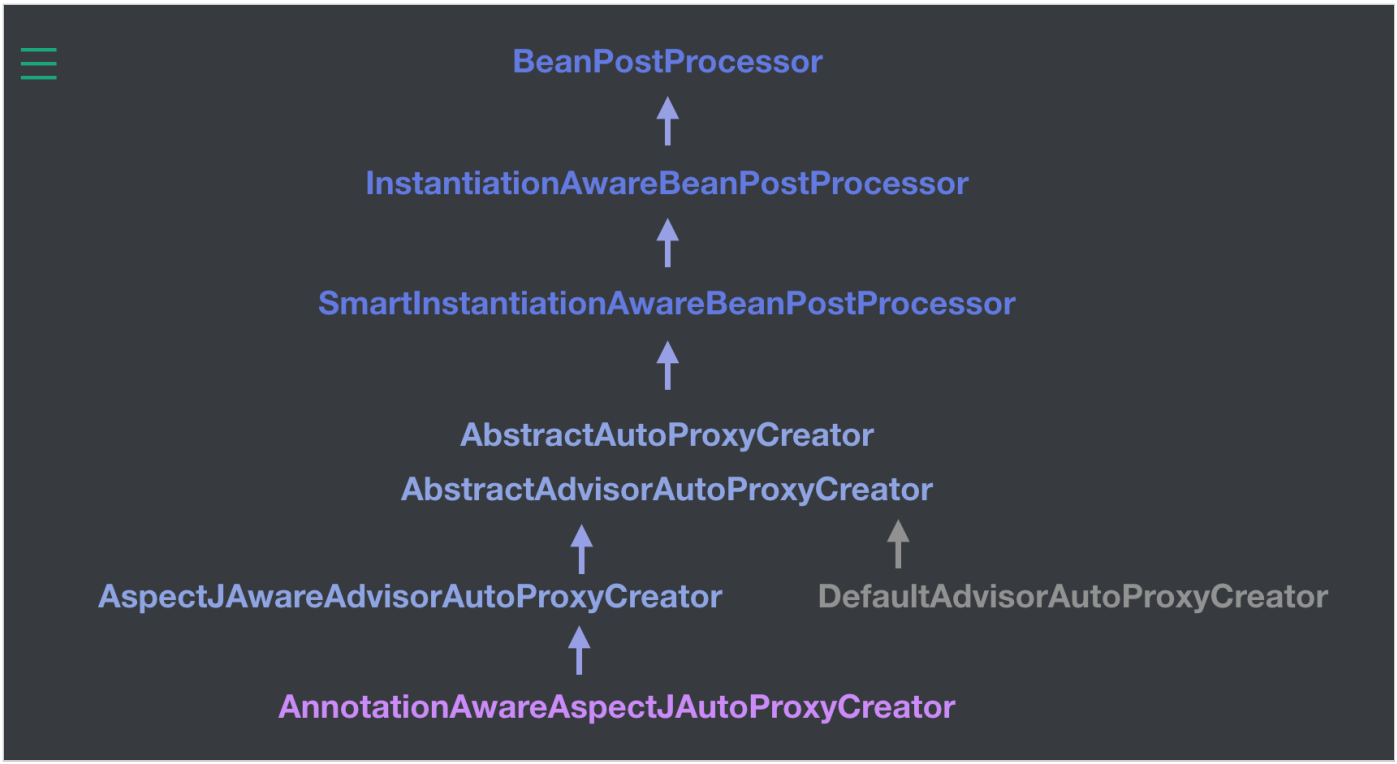
再进去 AopConfigUtils#registerAspectJAnnotationAutoProxyCreatorIfNecessary(...)：

```
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessa
        @Nullable Object source) {

    return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreatc
}
```

最终我们看到，Spring 注册了一个 AnnotationAwareAspectJAutoProxyCreator 的 bean，beanName 为："org.springframework.aop.config.internalAutoProxyCreator"。

我们看下 AnnotationAwareAspectJAutoProxyCreator 的继承结构：



和前面介绍的 DefaultAdvisorAutoProxyCreator 一样，它也是一个 BeanPostProcessor，剩下的我们就不说了，它和它的父类 AspectJAwareAdvisorAutoProxyCreator 都不复杂。

闲聊 InstantiationAwareBeanPostProcessor

为什么要说这个呢？因为我发现，很多人都以为 Spring AOP 是通过这个接口来作用于 bean 生成代理的。

```
public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor

    Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) t

    boolean postProcessAfterInstantiation(Object bean, String beanName) throws E

    PropertyValues postProcessPropertyValues(
            PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String bean

}
```

它和 BeanPostProcessor 的方法非常相似，而且它还继承了 BeanPostProcessor。

不仔细看还真的不好区分，下面是 BeanPostProcessor 中的两个方法：

```
Object postProcessBeforeInitialization(Object bean, String beanName) throws Bea
```

```
Object postProcessAfterInitialization(Object bean, String beanName) throws Bean
```

发现没有，InstantiationAwareBeanPostProcessor 是 `Instantiation`，BeanPostProcessor 是 `Initialization`，它代表的是 bean 在实例化完成并且属性注入完成，在执行 init-method 的前后进行作用的。

而 InstantiationAwareBeanPostProcessor 的执行时机要前面一些，大家需要翻下 IOC 的源码：

```
// AbstractAutowireCapableBeanFactory 447行
protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] a
    ...
    try {
        // 让 InstantiationAwareBeanPostProcessor 在这一步有机会返回代理
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }
    // BeanPostProcessor 是在这里面实例化后才能得到执行
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    ...
    return beanInstance;
}
```

点进去看 resolveBeforeInstantiation(beanName, mbdToUse) 方法，然后就会导到 InstantiationAwareBeanPostProcessor 的 postProcessBeforeInstantiation 方法，对于我们分析的 AOP 来说，该方法的实现在 AbstractAutoProxyCreator 类中：

```
@Override
public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanNam
    ...
    if (beanName != null) {
      TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
      if (targetSource != null) {
          this.targetSourcedBeans.add(beanName);
          Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass
          Object proxy = createProxy(beanClass, beanName, specificInterceptors,
          this.proxyTypes.put(cacheKey, proxy.getClass());
```

```
            return proxy;
        }
    }

    return null;
}
```

我们可以看到，这里也有创建代理的逻辑，以至于很多人会搞错。确实，这里是有可能创建代理的，但前提是对于相应的 bean 我们有自定义的 TargetSource 实现，进到 getCustomTargetSource(...) 方法就清楚了，我们需要配置一个 customTargetSourceCreators，它是一个 TargetSourceCreator 数组。

这里就不再展开说 TargetSource 了，请参考 Spring Reference 中的 Using TargetSources。

小结

本文真的是走马观花，和我之前写的文章有很大的不同，希望读者不会嫌弃。

不过如果读者有看过之前的 Spring IOC 源码分析和 Spring AOP 使用介绍 这两篇文章的话，通过看本文应该能对 Spring AOP 的源码实现有比较好的理解了。

本文说细节说得比较少，如果你在看源码的时候碰到不懂的，欢迎在评论区留言与大家进行交流。

（全文完）