【有道云笔记】15-Spring之AOP底层源码解析（下）.md
https://note.youdao.com/s/WldZVRCr

ProxyFactory选择cglib或jdk动态代理原理

ProxyFactory在生成代理对象之前需要决定到底是使用JDK动态代理还是CGLIB技术：

```
// config就是ProxyFactory对象

// optimize为true,或proxyTargetClass为true,或用户没有给ProxyFactory对象添加interface
if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
 Class<?> targetClass = config.getTargetClass();
 if (targetClass == null) {
  throw new AopConfigException("TargetSource cannot determine target class: " +
    "Either an interface or a target is required for proxy creation.");
 }
    // targetClass是接口，直接使用Jdk动态代理
 if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
  return new JdkDynamicAopProxy(config);
 }
    // 使用Cglib
 return new ObjenesisCglibAopProxy(config);
}
else {
    // 使用Jdk动态代理
 return new JdkDynamicAopProxy(config);
}
```

代理对象创建过程

JdkDynamicAopProxy

1. 在构造JdkDynamicAopProxy对象时，会先拿到被代理对象自己所实现的接口，并且额外的增加SpringProxy、Advised、DecoratingProxy三个接口，组合成一个Class[]，并赋值给proxiedInterfaces属性
2. 并且检查这些接口中是否定义了equals()、hashcode()方法
3. 执行`Proxy.newProxyInstance(classLoader, this.proxiedInterfaces, this)`，得到代理对象，**JdkDynamicAopProxy**作为InvocationHandler，代理对象在执行某个方法时，会进入到JdkDynamicAopProxy的**invoke()**方法中

ObjenesisCglibAopProxy

1. 创建Enhancer对象
2. 设置Enhancer的superClass为通过ProxyFactory.setTarget()所设置的对象的类
3. 设置Enhancer的interfaces为通过ProxyFactory.addInterface()所添加的接口，以及SpringProxy、Advised、DecoratingProxy接口
4. 设置Enhancer的Callbacks为DynamicAdvisedInterceptor
5. 最后创建一个代理对象，代理对象在执行某个方法时，会进入到DynamicAdvisedInterceptor的intercept()方法中

代理对象执行过程

1. 在使用ProxyFactory创建代理对象之前，需要往ProxyFactory先添加Advisor
2. 代理对象在执行某个方法时，会把ProxyFactory中的Advisor拿出来和当前正在执行的方法进行匹配筛选
3. 把和方法所匹配的Advisor适配成MethodInterceptor
4. 把和当前方法匹配的MethodInterceptor链，以及被代理对象、代理对象、代理类、当前Method对象、方法参数封装为MethodInvocation对象
5. 调用MethodInvocation的proceed()方法，开始执行各个MethodInterceptor以及被代理对象的对应方法
6. 按顺序调用每个MethodInterceptor的invoke()方法，并且会把MethodInvocation对象传入invoke()方法
7. 直到执行完最后一个MethodInterceptor了，就会调用invokeJoinpoint()方法，从而执行被代理对象的当前方法

各注解对应的MethodInterceptor

- **@Before**对应的是AspectJMethodBeforeAdvice，在进行动态代理时会把AspectJMethodBeforeAdvice转成**MethodBeforeAdviceInterceptor**
  - 先执行advice对应的方法
  - 再执行MethodInvocation的proceed()，会执行下一个Interceptor，如果没有下一个Interceptor了，会执行target对应的方法
- **@After**对应的是AspectJAfterAdvice，直接实现了**MethodInterceptor**
  - 先执行MethodInvocation的proceed()，会执行下一个Interceptor，如果没有下一个Interceptor了，会执行target对应的方法
  - 再执行advice对应的方法
- **@Around**对应的是AspectJAroundAdvice，直接实现了**MethodInterceptor**
  - 直接执行advice对应的方法，由@Around自己决定要不要继续往后面调用
- **@AfterThrowing**对应的是AspectJAfterThrowingAdvice，直接实现了**MethodInterceptor**
  - 先执行MethodInvocation的proceed()，会执行下一个Interceptor，如果没有下一个Interceptor了，会执行target对应的方法
  - 如果上面抛了Throwable，那么则会执行advice对应的方法
- **@AfterReturning**对应的是AspectJAfterReturningAdvice，在进行动态代理时会把AspectJAfterReturningAdvice转成**AfterReturningAdviceInterceptor**
  - 先执行MethodInvocation的proceed()，会执行下一个Interceptor，如果没有下一个Interceptor了，会执行target对应的方法
  - 执行上面的方法后得到最终的方法的返回值
  - 再执行Advice对应的方法

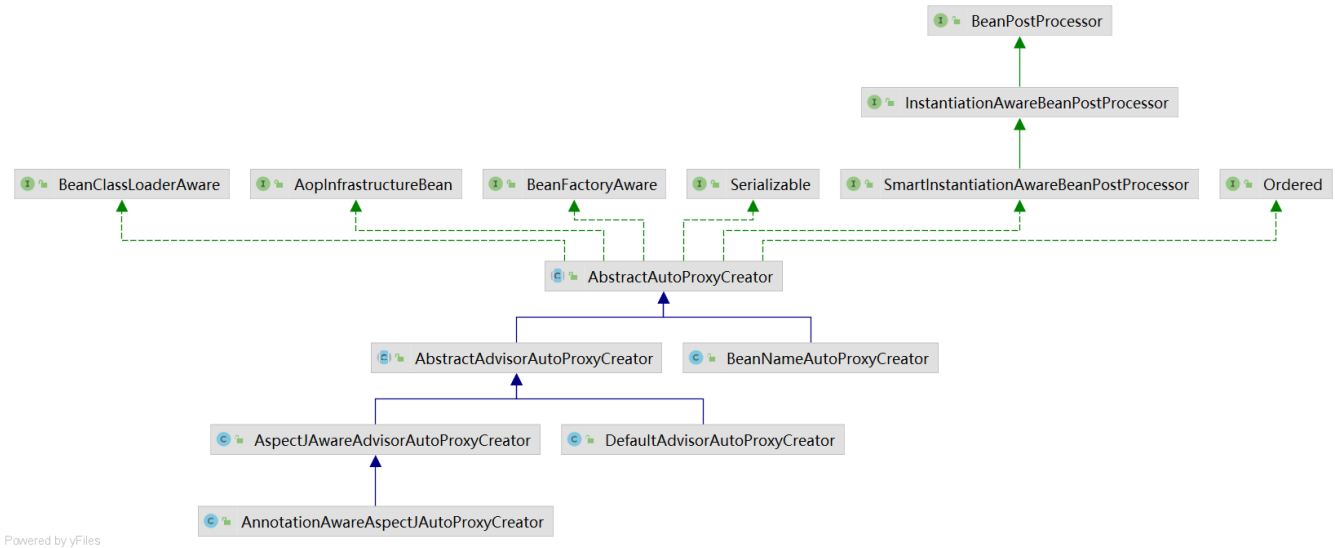
AbstractAdvisorAutoProxyCreator

DefaultAdvisorAutoProxyCreator的父类是AbstractAdvisorAutoProxyCreator。

AbstractAdvisorAutoProxyCreator非常强大以及重要，只要Spring容器中存在这个类型的Bean，就相当于开启了AOP，AbstractAdvisorAutoProxyCreator实际上就是一个BeanPostProcessor，所以在创建某个Bean时，就会进入到它对应的生命周期方法中，比如：在某个Bean**初始化之后**，会调用wrapIfNecessary()方法进行AOP，底层逻辑是，AbstractAdvisorAutoProxyCreator会找到所有的Advisor，然后判断当前这个Bean是否存在某个Advisor与之匹配（根据Pointcut），如果匹配就表示当前这个Bean有对应的切面逻辑，需要进行AOP，需要产生一个代理对象。

@EnableAspectJAutoProxy

这个注解主要就是往Spring容器中添加了一个AnnotationAwareAspectJAutoProxyCreator类型的Bean。



AspectJAwareAdvisorAutoProxyCreator继承了**AbstractAdvisorAutoProxyCreator**，重写了findCandidateAdvisors()方法，**AbstractAdvisorAutoProxyCreator**只能找到所有Advisor类型的Bean对象，但是**AspectJAwareAdvisorAutoProxyCreator**除开可以找到所有Advisor类型的Bean对象，还能把@Aspect注解所标注的Bean中的@Before等注解及方法进行解析，并生成对应的Advisor对象。

所以，我们可以理解@EnableAspectJAutoProxy，其实就是像Spring容器中添加了一个AbstractAdvisorAutoProxyCreator类型的Bean，从而开启了AOP，并且还会解析@Before等注解生成Advisor。

Spring中AOP原理流程图

https://www.processon.com/view/link/5faa4ccce0b34d7a1aa2a9a5