

**Spring AOP**

# AOP概述

AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

目前比较流行的动态代理技术：

1）JDKProxy（interface-based）

2）CGLIB（class-based）

静态织入：Aspectj

动态代理是在对象调用之前动态添加before 或者after事件。静态织入就比较粗暴了，将会改变class的原有结构，生成新的class，这样，在调用时，实际上我们引用的已经不是原来的class了。

Spring可以使用JDKProxy和CGLIB动态代理，也可以使用Aspectj的静态代理。不管使用哪种代理，都需要引入Aspectj的部分jar包，使用了其中的部分注解和技术。一般情况下，使用动态代理就基本满足日常应用了。毕竟静态代理还要使用Aspectj的编译器，复杂度略高。

1. Joinpoint(连接点)

类里面可以被增强的方法，这些方法称为连接点

1. Pointcut(切入点)

所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

在类里边可以有很多方法被增强，比如实际操作中，只是增强了个别方法，则定义实际被增强的某个方法为切入点。

1. Advice(通知/增强)

所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

1. Aspect(切面)

是切入点和通知（引介）的结合

1. Introduction(引介)

引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.

1. Target(目标对象)

代理的目标对象(要增强的类)

1. Weaving(织入)

是把增强应用到目标的过程，把advice 应用到 target的过程

1. Proxy（代理）

一个类被AOP织入增强后，就产生一个结果代理类

# Spring Boot AOP

SpringAOP本身的技术过程是很简单的。

解析我们创建或者配置的Advisor，使用@Aspectj注解的类。将其中的切点pointcut和advice解析后，在初始化Bean完成之后，通过pointcut中的ClassFilter和MethodMatcher去判定，当前类是否需要代理。

如果需要代理的话，使用ClassVisitor和ClassWriter去解析当前Class，并生成一个新的Class，Class中主要有以下几个属性

**private** MethodInterceptor CGLIB$CALLBACK\_0;  
**private** MethodInterceptor CGLIB$CALLBACK\_1;  
**private** NoOp CGLIB$CALLBACK\_2;  
**private** Dispatcher CGLIB$CALLBACK\_3;  
**private** Dispatcher CGLIB$CALLBACK\_4;  
**private** MethodInterceptor CGLIB$CALLBACK\_5;  
**private** MethodInterceptor CGLIB$CALLBACK\_6;

其中0号是：处理业务的拦截器，也就是我们所有需要拦截的函数，是一个DynamicAdvisedInterceptor，在CglibAopProxy Class中声明，当然，在jdk那边，也有一套。

5号处理equals函数

6号处理hashcode函数。

剩下其他的暂时没用上。

DynamicAdvisedInterceptor：我们的所有代理都是通过这个实现的。当我们调用我们的目标函数时，目标函数里会先调用这个Class中的intercept函数，这个函数会先获取我们调用的函数所需的advisor切点，又依次调用获取的通知器。没有的时候，再使用MethodProxy调用真正的函数。MethodProxy有两个函数，一个是invoke，一个是invokeSuper。Invoke调用的是原版的bean，也就是在我们调用的那个函数里，在调用本Class的其他函数时，将不会再次触发调用的代理拦截，因为你调用的Class还是原版的那个class。

调用invokeSuper的时候，所有函数都还是代理类，还是会触发代理拦截，你调用的Class是enhancer新做的Class，当调用其他函数的时候，还是会调用intercept函数，继续调用一系列的切点。

Spring选择了invoke函数，而且不可更改。当然，我们可以自己实现一个类似于CglibAopProxy、JdkDynamicAopProxy，这样的proxy，其中调用的地方改成invokeSuper就可以了。不过，spring似乎认为自己的选择是合理的，不然他们应该会给出配置的。

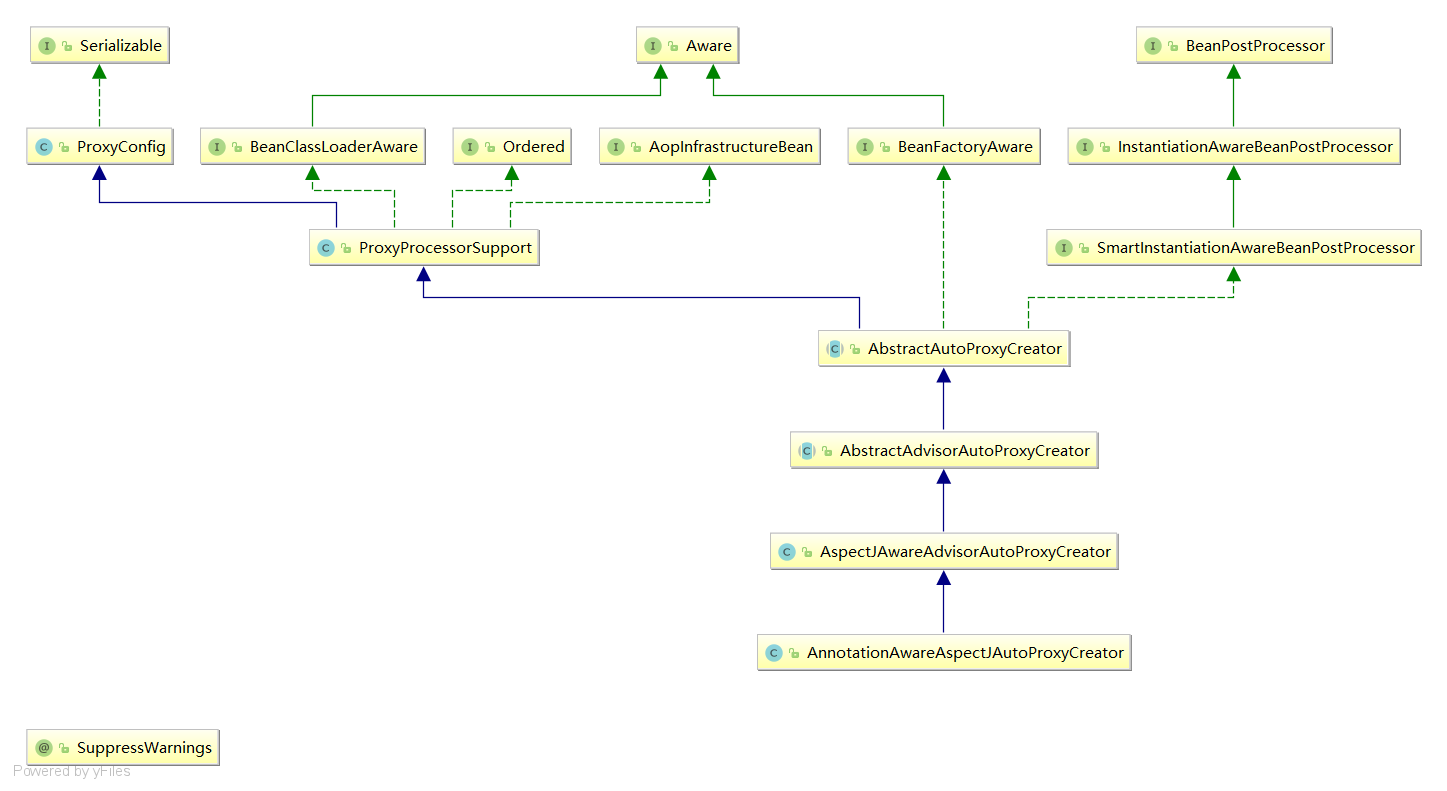
所以使用了代理的功能，都可能存在这个问题。比如：

你调用一个没有使用@Transactional注解的函数，里面调用@Transactional注解的函数，将不会触发第二次的代理拦截，也就不会加入事务。当然，在非必须的时候，我们不建议改成调用invokerSuper函数，毕竟每次函数调用，都要去触发拦截（不管是否需要代理处理），对于性能上还是有所损失的。毕竟，这种问题，我们一般都是通过更明确的研发格式处理。

Springboot是自动识别相关注解和class（AopAutoConfiguration）来配置AOP模块的。相关知识参考SpringBoot的启动相关。

识别后，根据配置（spring.aop.proxyTargetClass）属性配置JDKProxy或者CGLIB。

主要配置Class就像我们之前说的一样（AspectJAutoProxyRegistrar，继承ImportBeanDefinitionRegistrar），在Configuration注解解析完成后，再次添加部分Bean定义。这里会添加AnnotationAwareAspectJAutoProxyCreator，这个类是springAOP的核心，如下所示：



**注：如果图片太小看不清，另存到外面打开即可。**

## AnnotationAwareAspectJAutoProxyCreator

处理所有使用@Aspectj注解的class，还有Spring的advisor（继承自Advisor）spring自己的advisor规则遵循AbstractAdvisorAutoProxyCreator里面的

## AspectJAwareAdvisorAutoProxyCreator

处理Aspectj相关规则

## AbstractAdvisorAutoProxyCreator

主要配置器，自动创建代理类。

## AbstractAutoProxyCreator

自动配置代理类。SmartInstantiationAwareBeanPostProcessor实现了该类，在对象创建事件前后，初始化事件前后多次处理对象。主要的代理实现全在这里。

## ProxyProcessorSupport

主要的处理类，4.1加入，后续会讲到，使用机会不多。

## ProxyConfig

主要信息配置类。

1. AopInfrastructureBean

Aop的结构bean

## AbstractAutoProxyCreator

### postProcessBeforeInstantiation

初始化bean之前调用

判定当前class是否需要代理。

AspectJAwareAdvisorAutoProxyCreator🡪shouldSkip

判定的时候，获取所有的代理类Advisor

先获取实现了Advisor的类，之后获取Aspectj注解的类

将所有的advisor解析成功之后备用。

### postProcessAfterInitialization

在这里创建代理。

判定当前bean是否需要代理，这个地方特别复杂，有时间在详细看，没时间就算了，本身是一个成型的系统，使用的是Aspectj的jar

本质上还是获取切点，切点里有两种过滤器，一个是ClassFilter，一个是MethodMatcher。

系统默认提供了很多切入点，供你使用。

1. AnnotationMatchingPointcut

可承载两个参数，判定使用某Annotation注解的class或者method。

1. AspectJExpressionPointcut

我们使用的execution(\* com.spring.aop.service.impl.UserService\*+.\*(..)) && args()，这种配置就会解析成这个切点。

最终将AspectJExpressionPointcut中的ClassFilter和MethodMatcher赋值给ComposablePointcut，转译一层安全性更好。

1. JdkRegexpMethodPointcut

当然也可以使用JDK自带的regexp

基本也就这样几个重要的。

**Enhancer**

主要的函数，通过将所有的切点、通知信息带入到enhancer中，最终生成proxy

底层使用ClassVisitor和ClassWriter，按照我们的class，再重新造一个class，这个class的模式可以参照

可以很清楚的看到制作的class的代码。如果在反编译工具中看不出该类，可以放到工程里，用idea，idea的反编译器能清楚的看出结构。

## 获取代理类的方式

使用maven

做一个这样的类

maven配置大概这样

里面有个MANIFEST.MF文件

Manifest-Version: 1.0

Archiver-Version: Plexus Archiver

Created-By: Apache Maven

Built-By: himly

Build-Jdk: 1.8.0\_131

Premain-Class: com.spring.enhancer.UserServiceClassFileTransformer

Project-name: java-agent-classFileTransformer

Project-version: 1.0.0

大概是这样的

主要是里面有个Premain-Class，在我们使用主函数启动的时候，这个函数会在main函数之前执行。一般都不怎么会使用这个。某些特殊情况，我们可以使用这个方式，在main函数执行前，做一些事情。

在我们调用时，不管是主函数，或者是springboot（本质也是使用main函数启动），在vm options处添加-javaagent:C:\LZY\javaagent-classFileTransformer\target\java-agent-classFileTransformer-1.0.0.jar（改成你自己的）

这样，就能打出我们想要的class了。