# Spring的设计理念和整体架构

# Spring Framework的核心：IoC容器的实现

# Spring AOP的实现

## Spring AOP概述

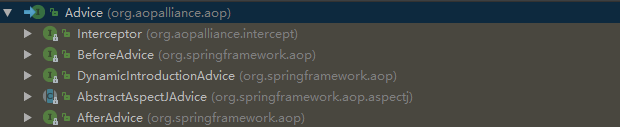
### AOP概念回顾

1、AOP是Aspect-Oriented Pogramming(面向切面编程)的简称

2、模块化

* 初学编程时喜欢把所有代码写进一个main函数里，这种编码方式造成了一个很不好的结果---程序维护性很差
* 于是，开始用子函数来对程序进行模块划分，并对一些基本的功能进行封装
* 后来，为了让代码维护更方便，又把不同的子函数的实现放到了不同的文件中。这样更方便了，不仅不用再一长串的代码文件里查找和维护，还可以让不同的开发人员并行开发和维护，大大提高开发效率
* 面向对象设计其实也是一种模块化的方法，它把相关的数据及其处理方法放在了一起。与单纯使用子函数进行封装相比，面向对象的模块化特性更完备，它体现了计算的一个基本原则---让计算尽可能靠近数据。一个类基本就是一个基本的模块。利用继承关系使得类得到重用，提高开发效率
* 但是程序中总会出现一些重复代码，而且不太方便使用继承的方法把他们重用和管理起来。它们功能重复并且需要用在不同的地方，虽然可以对这些代码左一些简单的封装，使之成为公共函数，但是在这种显式调用中，使用它们不是很方便
* 在使用这些公共函数的时候，往往也需要进行一些逻辑设计，也就是需要代码实现来支持，而且这些逻辑代码也是需要维护的
* 使用AOP后，可以将这些重复的代码抽取出来单独维护，在需要使用时统一调用，还可以为如何使用这些公共代码提供丰富灵活的手段

### Advice增强



1、Advice定义在连接点做什么，为切面增强提供织入接口。Advice是AOP联盟定义的一个接口如下

1. **public** **interface** Advice {
2. }

2、在Spring AOP的实现中，使用了这个统一接口，并通过这个接口，为AOP切面增强的织入功能做了更多的细化和扩展，比如BeforeAdvice、AfterAdvice、ThrowsAdvice等

1. **public** **interface** BeforeAdvice **extends** Advice {
2. }
3. **public** **interface** AfterAdvice **extends** Advice {
4. }

3、在BeforeAdvice的继承关系中，定义了为待增强的目标方法设置的前置增强接口MethodBeforeAdvice，使用这个前置接口需要实现一个回调函数

1. **public** **interface** MethodBeforeAdvice **extends** BeforeAdvice {
2. **void** before(Method var1, Object[] var2, Object var3) **throws** Throwable;
3. }

* 作为回调函数，before方法的实现在Advice中被配置到目标方法后，会在调用目标方法时被回调
* Method对象：这个参数是目标方法的反射对象
* Object[]对象数组：这个对象数组中包含方法的输入参数

4、在Advice的实现体系中，Spring还提供了AfterAdvice这种增强类型，其常用的子接口有AfterReturningAdvice，ThrowsAdvice

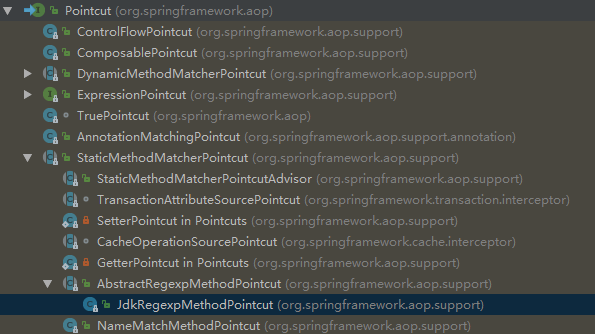
1. **public** **interface** AfterReturningAdvice **extends** AfterAdvice {
2. **void** afterReturning(Object var1, Method var2, Object[] var3, Object var4) **throws** Throwable;
3. }

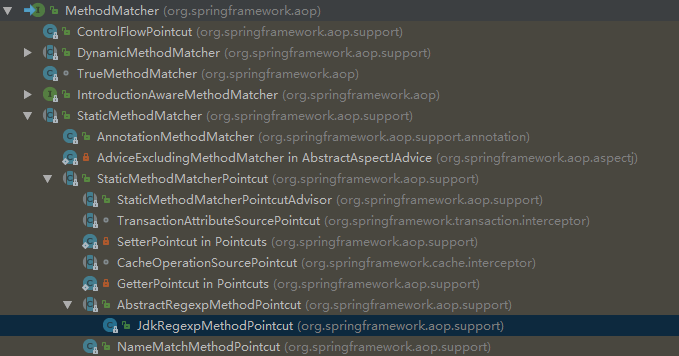
* afterReturning也是一个回调函数，AOP应用需要在这个接口实现中提供切面增强的具体设计，在这个Advice(增强)被正确配置后，在目标方法调用结束并成功返回的时候，接口会被Spring AOP回调

1. **public** **interface** ThrowsAdvice **extends** AfterAdvice {
2. }

* 对于ThrowsAdvice，并没有指定需要实现的接口方法，它在抛出异常时被回调，这个回调是AOP使用反射机制来完成的

### Pointcut切点





* JdkRegexpMethodPointcut作为两条继承体系交集的子类，因此JdkRegexpMethodPointcut就是一个MethodMatcher

1、Pointcut(切点)决定Advice(增强)应该作用于哪个连接点，也就是说通过Pointcut来定义需要增强的方法的集合，这些集合的选取可以按照一定的规则来完成。Pointcut是Spring定义的接口，如下

1. **public** **interface** Pointcut {
2. Pointcut TRUE = TruePointcut.INSTANCE;
4. ClassFilter getClassFilter();
6. MethodMatcher getMethodMatcher();
7. }

* Spring为Pointcut接口定义了一个默认实现，其返回的ClassFilter以及MethodMatcher均为默认实现，下面会介绍

1. **class** TruePointcut **implements** Pointcut, Serializable {
2. **public** **static** **final** TruePointcut INSTANCE = **new** TruePointcut();
4. **private** TruePointcut() {
5. }
7. **public** ClassFilter getClassFilter() {
8. **return** ClassFilter.TRUE;
9. }
11. **public** MethodMatcher getMethodMatcher() {
12. **return** MethodMatcher.TRUE;
13. }
15. **private** Object readResolve() {
16. **return** INSTANCE;
17. }
19. **public** String toString() {
20. **return** "Pointcut.TRUE";
21. }
22. }

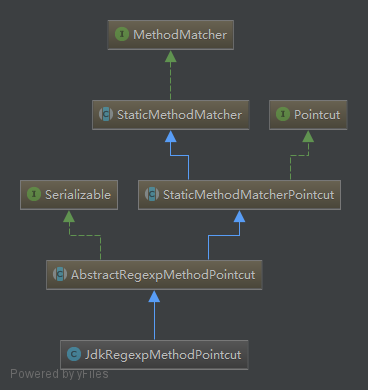
* 这个接口的两个方法很好理解，一个用于过滤类，一个用于过滤对应的方法，两个都返回true才返回成功
* 通过getMethodMatcher返回一个MethodMatcher，对于Point的匹配判断功能，具体是由这个返回的MethodMatcher来完成的。即由这个MethodMatcher来判断是否需要对当前方法调用进行增强，或者是否需要对当前调用方法应用到配置好的Advice(增强)
* ClassFilter与MethodMatcher接口定义如下

1. **public** **interface** ClassFilter {
2. ClassFilter TRUE = TrueClassFilter.INSTANCE;
4. **boolean** matches(Class<?> var1);
5. }
6. **public** **interface** MethodMatcher {
7. MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;
9. **boolean** matches(Method var1, Class<?> var2);
11. **boolean** isRuntime();
13. **boolean** matches(Method var1, Class<?> var2, Object[] var3);
14. }

* Spring为ClassFilter与MethodMatcher接口定义了两个默认实现，作为基本设定，matches方法均返回true。当Pointcut的实现类中，如果没有注入自定义的ClassFilter和MethodMatcher实现类，那么将会使用这两个基本类

1. **class** TrueClassFilter **implements** ClassFilter, Serializable {
2. **public** **static** **final** TrueClassFilter INSTANCE = **new** TrueClassFilter();
4. **private** TrueClassFilter() {
5. }
7. **public** **boolean** matches(Class<?> clazz) {
8. **return** **true**;
9. }
11. **private** Object readResolve() {
12. **return** INSTANCE;
13. }
15. **public** String toString() {
16. **return** "ClassFilter.TRUE";
17. }
18. }
19. **class** TrueMethodMatcher **implements** MethodMatcher, Serializable {
20. **public** **static** **final** TrueMethodMatcher INSTANCE = **new** TrueMethodMatcher();
22. **private** TrueMethodMatcher() {
23. }
25. **public** **boolean** isRuntime() {
26. **return** **false**;
27. }
29. **public** **boolean** matches(Method method, Class<?> targetClass) {
30. **return** **true**;
31. }
33. **public** **boolean** matches(Method method, Class<?> targetClass, Object[] args) {
34. **throw** **new** UnsupportedOperationException();
35. }
37. **private** Object readResolve() {
38. **return** INSTANCE;
39. }
41. **public** String toString() {
42. **return** "MethodMatcher.TRUE";
43. }
44. }

2、在Pointcut的继承关系中，以正则表达式切点JdkRegexpMethodPointcut的实现原理为例，来具体了解切点Pointcut的工作原理。JdkRegexpMethodPointcut类通过正则表达式对方法名进行匹配的功能

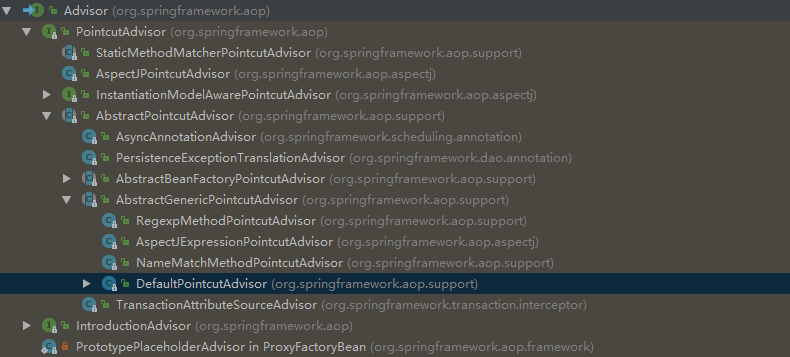


3、StaticMethodMatcherPointcut如下

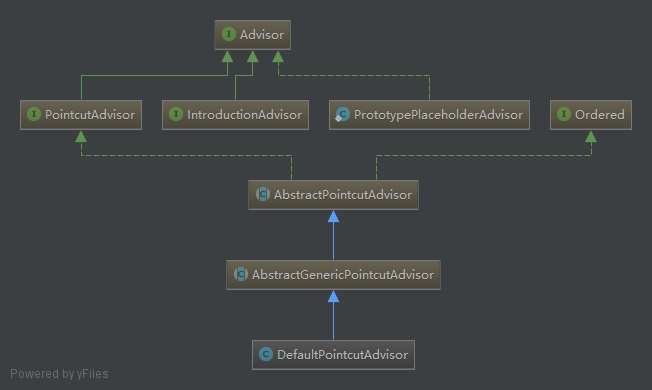
1. **public** **abstract** **class** StaticMethodMatcherPointcut **extends** StaticMethodMatcher **implements** Pointcut {
2. **private** ClassFilter classFilter;
4. **public** StaticMethodMatcherPointcut() {
5. **this**.classFilter = ClassFilter.TRUE;
6. }
8. **public** **void** setClassFilter(ClassFilter classFilter) {
9. **this**.classFilter = classFilter;
10. }
12. **public** ClassFilter getClassFilter() {
13. **return** **this**.classFilter;
14. }
16. **public** **final** MethodMatcher getMethodMatcher() {
17. **return** **this**;  //该类是MethodMatcher的实现，其父类有matches的基本定义(抛出异常，也就是说matches的具体实现交给StaticMethodMatcher的子类，同时StaticMethodMatcherPointcut也是StaticMethodMatcher的子类，但是没有重写matches方法，因此matches的具体工作交给StaticMethodMatcherPointcut的子类完成)该方法final是指返回的MethodMatcher的静态类型就是StaticMethodMatcherPointcut
18. }
19. }

* 返回的MethodMatcher就是StaticMethodMatcherPointcut的实例本身
* classFilter也是一个单例

### Advisor切面



1、完成对目标方法的切面增强设计(Advice)和关注点的设计(Pointcut)以后，需要一个对象把它们接合起来，完成这个作用的就是Advisor(切面)。Advisor是Spring定义的接口，如下



* Advisor如下，一个Advisor包含一个获取Advice的方法

1. **public** **interface** Advisor {
2. Advice getAdvice();
4. **boolean** isPerInstance();
5. }

* PointcutAdvisor接口对Advisor进行了扩展，添加了获取切点的方法，如下

1. **public** **interface** PointcutAdvisor **extends** Advisor {
2. Pointcut getPointcut();
3. }

* 意味着PointcutAdvisor的可用实现类中，必须包含一个Advice以及一个Pointcut

2、通过Advisor，可以定义应该使用哪个增强并在哪个切点使用它

3、以DefaultPointcutAdvisor进行分析

1. **public** **class** DefaultPointcutAdvisor **extends** AbstractGenericPointcutAdvisor **implements** Serializable {
2. **private** Pointcut pointcut;
4. **public** DefaultPointcutAdvisor() {
5. **this**.pointcut = Pointcut.TRUE;  //默认持有TruePointcut的实例
6. }
8. **public** DefaultPointcutAdvisor(Advice advice) {
9. **this**(Pointcut.TRUE, advice);
10. }
12. **public** DefaultPointcutAdvisor(Pointcut pointcut, Advice advice) {
13. **this**.pointcut = Pointcut.TRUE;
14. **this**.pointcut = pointcut;  //更换为自定义的Pointcut
15. **this**.setAdvice(advice);
16. }
18. **public** **void** setPointcut(Pointcut pointcut) {
19. **this**.pointcut = pointcut != **null**?pointcut:Pointcut.TRUE;
20. }
22. **public** Pointcut getPointcut() {
23. **return** **this**.pointcut;
24. }
26. **public** String toString() {
27. **return** **this**.getClass().getName() + ": pointcut [" + **this**.getPointcut() + "]; advice [" + **this**.getAdvice() + "]";
28. }
29. }

* 默认构造方法中pointcut默认被设置为Pointcut.Ture，这个Pointcut.True在Pointcnt接口中被定义
* TruePointcut存在的意义就是提供一个默认的返回true的切点

## Spring AOP的设计与实现

### JVM的动态代理特性

1、在Spring AOP实现中，使用的核心技术是动态代理，而这种动态代理实际上是JDK的一个特性，通过JDK动态代理特性，可以为任意Java对象创建代理对象，对于具体使用来说，这个特性是通过Java Reflection API来完成的

2、调用过程如下图所示



* 在Proxy的调用过程中，如果客户(Client)调用Proxy的request方法，会在调用目标对象的request方法的前后调用一系列处理，而这一系列的处理相对于目标对象来说是透明的，目标对象对这些处理毫不知情，这就是Proxy模式
* 在使用JDK的动态代理时，需要位代理对象设计一个回调方法，这个回调方法起到的作用就是加入额外的处理动作

1. **public** **interface** InvocationHandler {
2. **public** Object invoke(Object proxy, Method method, Object[] args)
3. **throws** Throwable;
4. }

* 该方法第一参数是代理对象实例，第二个参数是Method方法对象，代表的是当前Proxy被调用的方法，最后一个参数是被调用的方法中的参数

3、为什么这个接口需要传入代理对象的实例：

* 产生这个问题的原因是：我把InvocationHandler的对象当成代理对象了，其实这个InvocationHandler的实例只是Advice(增强)而已，代理对象是通过字节码技术动态生成的，并不是InvocationHandler的实例，因此必须传入代理对象
* Proxy.newInstance()在生成具体代理对象时，会传入InvocationHandler的实例，并从这个InvocationHandler的实例中获取被代理对象的实例

### Spring AOP的设计分析

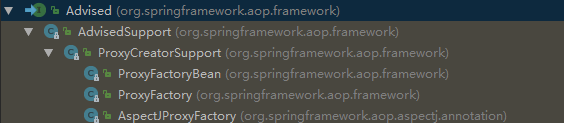
1、对于AOP的使用者来说，只需要配置相关的Bean定义即可，但仔细分析Spring AOP的内部设计可以看到，为了让AOP起作用，需要完成一系列过程：

1. 需要为目标对象建立代理对象，这个代理对象可以通过JDK的Proxy来完成，也可以通过第三方的类生成器CGLib来完成
2. 还需要启动代理对象的拦截器来完成各种横切面的织入，这一系列的织入设计是通过一系列Adapter来实现的，通过一系列的Adapter的设计，可以把AOP的横切面设计和Proxy模式有机结合起来，从而实现在AOP中定义好的各种组织方式

### Spring AOP的应用场景

## 建立AoPProxy代理对象

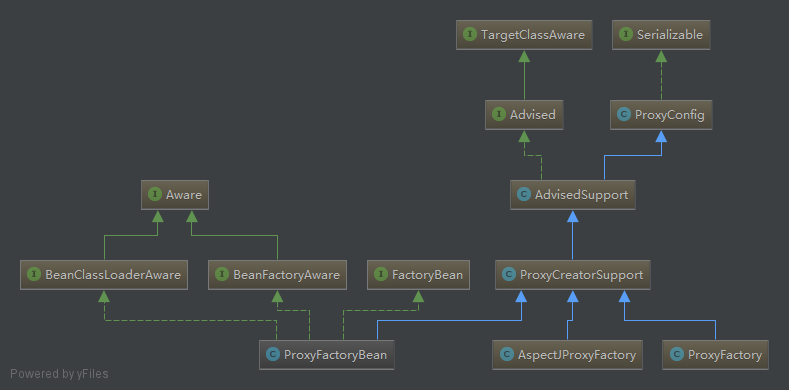
### 设计原理



* 注意区分Advised和Advice和Adivisor，名字差不多！！！
* Advised扮演者类似于Manager的一个功能

1、在Spring的AOP模块中，一个主要的部分是代理对象的生成，而对于Spring应用，是通过配置和调用Spring的ProxyFactoryBean来完成这个任务的。在ProxyFactoryBean，封装了主要代理对象的生成过程。可以使用JDK的Proxy和CGLib两种生成方式

2、ProxyFactoryBean的继承关系如下



* ProxyConfig是一个数据基类，这个数据基类为ProxyFactoryBean这样的子类提供了配置属性
* **AdvisedSupport封装了AOP对增强和切面的相关操作(含有List<Advisor>)**，这些操作对于不同AOP代理对象的生成都是一样的，但是对于具体的AOP代理对象的创建，AdvisedSupport把它交给它的子类取完成，**还保留了被代理对象的信息(实例以及类型TargetSource)**
* ProxyCreatorSupport是子类创建AOP代理对象的一个辅助类
* 对于需要使用AspectJ的AOP应用，AspectJProxyFactory起到集成Spring和AspectJ的作用
* 对于使用Spring AOP的应用，ProxyFactoryBean和ProxyFactory都提供了AOP功能的封装，只是使用ProxyFactoryBean，可以在IoC容器中完成声明式配置，而使用ProxyFactory，则需要编程式地使用Spring AOP的功能

### 配置ProxyFactoryBean

1、在基于XML配置Spring Bean时，往往需要一系列的配置步骤来使用ProxyFactoryBean和AOP

1. 定义使用的切面Advisor，这个切面应该作为一个Bean来定义，这个切面需要切点(方法定位条件)以及增强(额外逻辑代码以及方位信息)的定义
2. 定义ProxyFactoryBean，把它作为另一个Bean来定义，他是封装AOP的主要类，在配置ProxyFactoryBean时，需要设定与AOP实现相关的重要属性，比如proxyInterface、interceptorNames和target等

* interceptorNames属性的值往往设置为需要的Advice，因为这些切面在ProxyFactoryBean的AOP配置下，是通过使用代理对象的拦截器机制起作用的，因此沿用了拦截器的名字，旧瓶装新酒

1. 定义target属性，作为target属性注入的Bean，即被代理对象

### ProxyFactoryBean生成AopProxy代理对象

1、AopProxy的生成过程如下图



2、ProxyFactoryBean#getObject

1. **public** Object getObject() **throws** BeansException {
2. //这里初初始化Advisor链
3. **this**.initializeAdvisorChain();
4. //这里对singleton和prototype的类型进行区分，生成对应的proxy
5. **if**(**this**.isSingleton()) {
6. **return** **this**.getSingletonInstance();  //这里生成的是代理对象的单例
7. } **else** {
8. **if**(**this**.targetName == **null**) {
9. **this**.logger.warn("Using non-singleton proxies with singleton targets is often undesirable. Enable prototype proxies by setting the \'targetName\' property.");
10. }
12. **return** **this**.newPrototypeInstance();
13. }
14. }

3、ProxyFactoryBean#initializeAdvisorChain

1. **private** **synchronized** **void** initializeAdvisorChain() **throws** AopConfigException, BeansException {
2. **if**(!**this**.advisorChainInitialized) {
3. **if**(!ObjectUtils.isEmpty(**this**.interceptorNames)) {
4. **if**(**this**.beanFactory == **null**) {
5. **throw** **new** IllegalStateException("No BeanFactory available anymore (probably due to serialization) - cannot resolve interceptor names " + Arrays.asList(**this**.interceptorNames));
6. }
8. **if**(**this**.interceptorNames[**this**.interceptorNames.length - 1].endsWith("\*") && **this**.targetName == **null** && **this**.targetSource == EMPTY\_TARGET\_SOURCE) {
9. **throw** **new** AopConfigException("Target required after globals");
10. }
12. String[] var1 = **this**.interceptorNames;  //interceptorNames属性对应的String[]
13. **int** var2 = var1.length;
15. //这里添加Advisor链的调用，在XML文件中通过interceptorNames属性进行配置
16. **for**(**int** var3 = 0; var3 < var2; ++var3) {
17. String name = var1[var3];
18. **if**(**this**.logger.isTraceEnabled()) {
19. **this**.logger.trace("Configuring advisor or advice \'" + name + "\'");
20. }
22. **if**(name.endsWith("\*")) {
23. **if**(!(**this**.beanFactory **instanceof** ListableBeanFactory)) {
24. **throw** **new** AopConfigException("Can only use global advisors or interceptors with a ListableBeanFactory");
25. }
27. **this**.addGlobalAdvisor((ListableBeanFactory)**this**.beanFactory, name.substring(0, name.length() - "\*".length()));
28. } **else** {
29. //如果程序在这里被调用，那么需要加入命名的拦截器advice，并且需要检查这个Bean是singleton还是prototype类型
30. Object advice;
31. **if**(!**this**.singleton && !**this**.beanFactory.isSingleton(name)) {
32. advice = **new** ProxyFactoryBean.PrototypePlaceholderAdvisor(name);
33. } **else** {
34. //将获取Adivce的过程转交给IoC容器来完成
35. advice = **this**.beanFactory.getBean(name);
36. }
37. //将Advice包装成Advisor后添加到AdvisedSuppot的List<Advisor>字段当中
38. **this**.addAdvisorOnChainCreation(advice, name);
39. }
40. }
41. }
43. **this**.advisorChainInitialized = **true**;
44. }
45. }

* 该过程有一个标志位advisorChainInitialized，用来表示Advisor链是否已经初始化，如果已经初始化，就不再初始化直接返回
* 这个初始化过程发生在应用第一次通过ProxyFactoryBean取获取代理对象的时候
* 取得Advisor的方式也很简单，把Advisor的名字交给容器的getBean方法就可以了，然后把Advisor通过方法addAdvisorOnChainCreation添加到拦截器链中
* 其中ProxyFactoryBean#addAdvisorOnChainCreation中第一句将Advice转换成了Advisor

1. **private** **void** addAdvisorOnChainCreation(Object next, String name) {
2. //这里将Advice包装成了Advisor
3. Advisor advisor = **this**.namedBeanToAdvisor(next);
4. **if**(**this**.logger.isTraceEnabled()) {
5. **this**.logger.trace("Adding advisor with name \'" + name + "\'");
6. }
8. **this**.addAdvisor(advisor);
9. }

* ProxyFactoryBean# namedBeanToAdvisor

1. **private** Advisor namedBeanToAdvisor(Object next) {
2. **try** {
3. //其中advisorAdapterRegistry是GlobalAdvisorAdapterRegistry下的单例
4. **return** **this**.advisorAdapterRegistry.wrap(next);
5. } **catch** (UnknownAdviceTypeException var3) {
6. **throw** **new** AopConfigException("Unknown advisor type " + next.getClass() + "; Can only include Advisor or Advice type beans in interceptorNames chain except for last entry," + "which may also be target or TargetSource", var3);
7. }
8. }

4、接着看ProxyFactoryBean#getObject，生成singleton的代理对象在getSingletonInstance()中完成的，这个方法是ProxyFactoryBean生成AopProxy代理对象的调用入口。代理对象会封装对target目标对象的调用，也就是说针对target对象的方法调用行为会被这里生成的代理对象拦截

* 具体生成过程是
* 读取ProxyFactoryBean中的配置，为生成代理对象做好必要的准备，比如设置代理的方法调用接口等
* Spring通过AopProxy类来具体生成代理对象

5、ProxyFactoryBean#getSingletonInstance()

1. **private** **synchronized** Object getSingletonInstance() {
2. **if**(**this**.singletonInstance == **null**) {
3. **this**.targetSource = **this**.freshTargetSource();
4. **if**(**this**.autodetectInterfaces && **this**.getProxiedInterfaces().length == 0 && !**this**.isProxyTargetClass()) {
5. //根据AOP框架来判断需要代理的接口
6. Class targetClass = **this**.getTargetClass();
7. **if**(targetClass == **null**) {
8. **throw** **new** FactoryBeanNotInitializedException("Cannot determine target class for proxy");
9. }
10. //这里设置代理对象的接口
11. **this**.setInterfaces(ClassUtils.getAllInterfacesForClass(targetClass, **this**.proxyClassLoader));
12. }
14. **super**.setFrozen(**this**.freezeProxy);
15. //这里的方法会使用ProxyFactory来生成需要的Proxy
16. **this**.singletonInstance = **this**.getProxy(**this**.createAopProxy());
17. }
19. **return** **this**.singletonInstance;
20. }
22. //通过createAopProxy返回的AopProxy来得到代理对象
23. **protected** Object getProxy(AopProxy aopProxy) {
24. **return** aopProxy.getProxy(**this**.proxyClassLoader);
25. }

* AopProxy是一个接口，由两个子类实现，一个是Cglib2AopProxy，另一个是JdkDynamicProxy。对这两个AopProxy接口的子类的实现，Spring分别通过CGLIB和JDK来生成需要的Proxy对象

6、具体代理对象的生成，是在ProxyFactoryBean的基类ProxyCreatorSupport的实现中借助AopProxyFactory完成的，这个代理对象要么从JDK中生成，要么借助CGLIB获得。由于ProxyFactoryBean本身就是ProxyCreatorSupport的子类，所以在ProxyFactoryBean中获得AopProxy是很方便的。在ProxyCreatorSupport中看到，具体的AopProxy是通过AopProxyFactory来生成的，至于需要生成什么样的代理对象，所有的信息都封装在AdvisedSupport里， 这个对象也是生成AopProxy的方法输出参数

7、ProxyCreatorSupport#createAopProxy

1. **protected** **final** **synchronized** AopProxy createAopProxy() {
2. **if**(!**this**.active) {
3. **this**.activate();
4. }
5. //通过AopProxyFactory获取AopProxy，这个AopProxyFactory是在初始化函数中定义的，使用的是DefaultAopProxyFactory
6. **return** **this**.getAopProxyFactory().createAopProxy(**this**);
7. }

* 这里使用了AopProxyFactory来创建AopFactory，AopProxyFactory使用的是DefaultAopProxyFactory，该类的作用就根据配置信息(AdvisedSupport)创建JdkDynamicAopProxy的实例或者CglibProxyFactory的实例
* 这个被使用的AopProxyFactory，作为AopProxy的创建工厂对象，是在ProxyFactoryBean的基类ProxyCreatorSupport中被创建的。在创建AopProxyFactory时，它被设置为DefaultAopProxyFactory，于是问题就转换为在DefaultAopProxyFactory中，AopProxy是怎样生成的

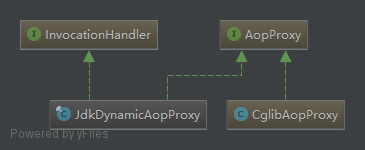
8、关于AopProxy代理对象的生成，需要考虑使用哪种生成方式，如果目标对象是接口类，那么适合使用JDK来生成代理对象，否则Spring会使用CGLIB来生成目标对象的代理对象。对于具体的AopProxy代理对象的生成，最终并没有由DefaultAopProxyFactory来完成，而是由Spring封装的JdkDynamicAopProxy和CglibProxyFactory类来完成的

1. **public** **class** DefaultAopProxyFactory **implements** AopProxyFactory, Serializable {
2. **public** DefaultAopProxyFactory() {
3. }
5. **public** AopProxy createAopProxy(AdvisedSupport config) **throws** AopConfigException {
6. **if**(!config.isOptimize() && !config.isProxyTargetClass() && !**this**.hasNoUserSuppliedProxyInterfaces(config)) {
7. **return** **new** JdkDynamicAopProxy(config);
8. } **else** {
9. Class targetClass = config.getTargetClass();
10. **if**(targetClass == **null**) {
11. **throw** **new** AopConfigException("TargetSource cannot determine target class: Either an interface or a target is required for proxy creation.");
12. } **else** {
13. **return** (AopProxy)(!targetClass.isInterface() && !Proxy.isProxyClass(targetClass)?**new** ObjenesisCglibAopProxy(config):**new** JdkDynamicAopProxy(config));
14. }
15. }
16. }
18. **private** **boolean** hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
19. Class[] ifcs = config.getProxiedInterfaces();
20. **return** ifcs.length == 0 || ifcs.length == 1 && SpringProxy.**class**.isAssignableFrom(ifcs[0]);
21. }
22. }

9、在AopProxy代理对象生成过程中，首先要从AdvisedSupport对象中获得配置的目标对象。在对目标对象配置的检查完成之后，需要根据配置的情况来决定使用什么方式来创建AopProxy代理对象，一般而言默认用JDK来产生AopProxy代理对象，但是如果遇到的配置的目标对象不是接口类的实现，会使用CGLib来产生AopProxy代理对象

### JDK生成AopProxy代理对象

1、AopProxy继承关系图如下



2、JdkDynamicAopProxy#getProxy

1. **public** Object getProxy(ClassLoader classLoader) {
2. **if**(logger.isDebugEnabled()) {
3. logger.debug("Creating JDK dynamic proxy: target source is " + **this**.advised.getTargetSource());
4. }
6. Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(**this**.advised, **true**);
7. **this**.findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
8. **return** Proxy.newProxyInstance(classLoader, proxiedInterfaces, **this**);
9. }

* 在生成Proxy对象之前，首先需要从advised对象中取得代理对象的代理接口配置，然后调用Proxy的newProxyInstance方法，最终得到对应的Proxy代理对象，newProxyInstance方法需要指明三个参数
* 类装载器
* 代理接口
* Proxy回调方法所在对象，这个对象需要实现InvocationHandler接口，通过回调方法invoke，完成了AOP编织实现的封装

### CGLIB生成AopProxy代理对象

1、CglibAopProxy#getProxy

1. **public** Object getProxy(ClassLoader classLoader) {
2. **if**(logger.isDebugEnabled()) {
3. logger.debug("Creating CGLIB proxy: target source is " + **this**.advised.getTargetSource());
4. }
5. //从advised中取得IoC容器中配置的target对象
6. **try** {
7. Class ex = **this**.advised.getTargetClass();
8. Assert.state(ex != **null**, "Target class must be available for creating a CGLIB proxy");
9. Class proxySuperClass = ex;
10. **int** x;
11. **if**(ClassUtils.isCglibProxyClass(ex)) {
12. proxySuperClass = ex.getSuperclass();
13. Class[] enhancer = ex.getInterfaces();
14. Class[] callbacks = enhancer;
15. **int** types = enhancer.length;
17. **for**(x = 0; x < types; ++x) {
18. Class additionalInterface = callbacks[x];
19. **this**.advised.addInterface(additionalInterface);
20. }
21. }
23. **this**.validateClassIfNecessary(proxySuperClass, classLoader);
24. //验证代理对象的接口设置，并创建CGLIB的Enhancer，这个Enhancer对象是CGLIB的主要操作类
25. Enhancer var12 = **this**.createEnhancer();
26. **if**(classLoader != **null**) {
27. var12.setClassLoader(classLoader);
28. **if**(classLoader **instanceof** SmartClassLoader && ((SmartClassLoader)classLoader).isClassReloadable(proxySuperClass)) {
29. var12.setUseCache(**false**);
30. }
31. }
32. //设置Enhancer对象，包括设置代理接口，回调方法
33. //来自advised的IoC配置，比如使用AOP的DynamicAdvisedInterceptor拦截器
34. var12.setSuperclass(proxySuperClass);
35. var12.setInterfaces(AopProxyUtils.completeProxiedInterfaces(**this**.advised));
36. var12.setNamingPolicy(SpringNamingPolicy.INSTANCE);
37. var12.setStrategy(**new** CglibAopProxy.ClassLoaderAwareUndeclaredThrowableStrategy(classLoader));
38. Callback[] var13 = **this**.getCallbacks(ex);
39. Class[] var14 = **new** Class[var13.length];
41. **for**(x = 0; x < var14.length; ++x) {
42. var14[x] = var13[x].getClass();
43. }
45. var12.setCallbackFilter(**new** CglibAopProxy.ProxyCallbackFilter(**this**.advised.getConfigurationOnlyCopy(), **this**.fixedInterceptorMap, **this**.fixedInterceptorOffset));
46. var12.setCallbackTypes(var14);
47. //通过Enhancer生成代理对象
48. **return** **this**.createProxyClassAndInstance(var12, var13);
49. } **catch** (CodeGenerationException var9) {
50. **throw** **new** AopConfigException("Could not generate CGLIB subclass of class [" + **this**.advised.getTargetClass() + "]: " + "Common causes of this problem include using a final class or a non-visible class", var9);
51. } **catch** (IllegalArgumentException var10) {
52. **throw** **new** AopConfigException("Could not generate CGLIB subclass of class [" + **this**.advised.getTargetClass() + "]: " + "Common causes of this problem include using a final class or a non-visible class", var10);
53. } **catch** (Exception var11) {
54. **throw** **new** AopConfigException("Unexpected AOP exception", var11);
55. }
56. }

* 在这个过程中，需要注意的是对Enhancer对象callback回调的设置，正是这些回调封装了Spring AOP的实现，就像JDK的Proxy对象的invoke回调一样，Enhancer的callback回调设置中，实际上是通过设置DynamicAdvisedInterceptor拦截器来完成AOP功能的

### 总结

1、通过使用AopProxy对象封装target目标对象之后，ProxyFactoryBean的getObject方法得到的对象就不是一个普通的Java对象了，而是一个AopProxy代理对象

2、对target目标对象的方法调用首先会被AopProxy代理对象拦截(说白了客户端操作的就是是代理对象)，对于不同的AopProxy代理对象生成方式，会使用不同的拦截回调入口

1. 对于JDK的AopProxy代理对象，使用的是InvocationHandler的invoke回调入口
2. 对于CGLIB的callback回调中，对于AOP实现，是通过DynamicAdvisedInterceptor来完成的，而DynamicAdvisedInterceptor的回调入口是intercept方法

3、可以把AOP的实现部分分成两个部分

1. AopPorxy代理对象的生成：可以看成一个静态的AOP基础设施的建立过程，通过这个准备过程，把代理对象、拦截器这些待调用的部分都准备好，等待着AOP运行过程中对这些基础设施的使用
2. 拦截器调用：拦截器调用的实现原理，和AopProxy代理对象生成一样，也是AOP实现的重要组成部分

## Spring AOP拦截器调用的实现

### 拦截器

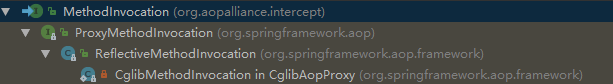
1、MethodInterceptor定义如下(注意区分MethodInvocation与MethodInterceptor)

1. **public** **interface** MethodInterceptor **extends** Interceptor {
2. Object invoke(MethodInvocation var1) **throws** Throwable;
3. }

2、任取一个MethodInterceptor实现类，以MethodBeforeAdviceInterceptor为例进行分析

1. **public** **class** MethodBeforeAdviceInterceptor **implements** MethodInterceptor, Serializable {
2. **private** MethodBeforeAdvice advice;
4. **public** MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
5. Assert.notNull(advice, "Advice must not be null");
6. **this**.advice = advice;
7. }
9. **public** Object invoke(MethodInvocation mi) **throws** Throwable {
10. //执行拦截逻辑
11. **this**.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());
12. //继续执行拦截器调用链，注意，这里用的是传入的参数mi
13. **return** mi.proceed();
14. }
15. }

3、MethodInvocation继承链如下



* 主动进行拦截器链调用的是MethodInvocation的实现类
* MethodInvocation的实现类必须持有一个拦截器链，这样才能依序进行拦截器的调用
* 另外，MethodInvocation的实现类还必须持有被代理对象，来执行真正的业务逻辑
* JdkDynamicAopProxy采用的是ReflectiveMethodInvocation
* CglibAopProxy采用的是CglibMethodInvocation

### 设计原理

1、在Spring AOP通过JDK的Proxy方式或CGLIB方式生成代理对象的时候，相关的拦截器已经配置到代理对象中去了，拦截器在代理对象中起作用是通过这些方法的回调来完成的

2、如果使用JDK的Proxy来生成代理对象，那么需要通过InvocationHandler来设置拦截器回调；如果使用CGLIB来生成代理对象，就需要根据CGLIB的使用要求，通过DynamicAdvisedInterceptor来完成回调

### JdkDynamicAopProxy的invoke拦截

1、在JdkDynamicAopProxy中生成AopProxy对象时调用如下

1. **return** Proxy.newProxyInstance(classLoader, proxiedInterfaces, **this**);

* 这里的this参数应该是InvocationHandler对象(JkdDynamicAopProxy实现了InvocationHandler接口)，InvocationHandler是JDK定义的反射类的一个接口，该接口定义了invoke方法，这个invoke方法是作为JDK Proxy代理对象进行拦截的回调入口
* 在JdkDynamicAopProxy的invoke方法作为Proxy对象的回调函数被触发，从而通过invoke的具体实现，来完成对目标对象方法调用的拦截或者说功能增强工作

2、JdkDynamicAopProxy#invoke

1. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
2. Object oldProxy = **null**;
3. **boolean** setProxyContext = **false**;
4. TargetSource targetSource = **this**.advised.targetSource;
5. Class targetClass = **null**;
6. Object target = **null**;
8. Class retVal;
9. **try** {
10. **if**(!**this**.equalsDefined && AopUtils.isEqualsMethod(method)) {
11. //如果目标对象没有实现Object类的基本方法:equals
12. Boolean retVal3 = Boolean.valueOf(**this**.equals(args[0]));
13. **return** retVal3;
14. }
16. **if**(!**this**.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
17. //如果目标对象没有实现Object类的基本方法:hashCode
18. Integer retVal2 = Integer.valueOf(**this**.hashCode());
19. **return** retVal2;
20. }
22. **if**(method.getDeclaringClass() != DecoratingProxy.**class**) {
23. Object retVal1;
24. **if**(!**this**.advised.opaque && method.getDeclaringClass().isInterface() && method.getDeclaringClass().isAssignableFrom(Advised.**class**)) {
25. //根据代理对象的配置来调用服务
26. retVal1 = AopUtils.invokeJoinpointUsingReflection(**this**.advised, method, args);
27. **return** retVal1;
28. }
30. **if**(**this**.advised.exposeProxy) {
31. oldProxy = AopContext.setCurrentProxy(proxy);
32. setProxyContext = **true**;
33. }
35. //得到目标对象的地方
36. target = targetSource.getTarget();
37. **if**(target != **null**) {
38. targetClass = target.getClass();
39. }
41. //从这里获得定义好的拦截器,将Advisor转换成MethodInterceptor
42. List chain = **this**.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
43. //如果没有设定拦截器，那么就直接调用target的对应方法
44. **if**(chain.isEmpty()) {
45. Object[] returnType = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
46. retVal1 = AopUtils.invokeJoinpointUsingReflection(target, method, returnType);
47. } **else** {
48. //如果有拦截器的设定，那么需要调用拦截器之后才调用目标对象的响应方法，通过构造一个ReflectiveMethodInvocation来实现
49. ReflectiveMethodInvocation invocation = **new** ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
50. //沿着拦截器链继续前进
51. retVal1 = invocation.proceed();
52. }
54. Class returnType1 = method.getReturnType();
55. **if**(retVal1 != **null** && retVal1 == target && returnType1.isInstance(proxy) && !RawTargetAccess.**class**.isAssignableFrom(method.getDeclaringClass())) {
56. retVal1 = proxy;
57. } **else** **if**(retVal1 == **null** && returnType1 != Void.TYPE && returnType1.isPrimitive()) {
58. **throw** **new** AopInvocationException("Null return value from advice does not match primitive return type for: " + method);
59. }
61. Object var13 = retVal1;
62. **return** var13;
63. }
65. retVal = AopProxyUtils.ultimateTargetClass(**this**.advised);
66. } **finally** {
67. **if**(target != **null** && !targetSource.isStatic()) {
68. targetSource.releaseTarget(target);
69. }
71. **if**(setProxyContext) {
72. AopContext.setCurrentProxy(oldProxy);
73. }
75. }
77. **return** retVal;
78. }

### CglibAopProxy的intercept拦截

1、CglibAopProxy的AopProxy代理对象生成的时候，回调是在DynamicAdvisedInterceptor(CglibAopProxy.java中的静态内部类)对象中实现的。CglibAopProxy的intercept回调方法的实现和JdkDynamicAopProxy的回调实现是非常类似的，只是在CglibAopProxy中构造CglibMethodInvocation对象(ReflectiveMethodInvocation的子类)来完成拦截器链的调用，而在JdkDynamicAopProxy中是通过构造ReflectiveMethodInvocation对象来完成的

* 第一个问题：为什么不像JdkDynamicAopProxy实现InvocationHandler接口一样，直接让CglibAopProxy实现Cglib回调接口MethodInterceptor???，好像无所谓
* 第二个问题：MethodIntercept接口在好多地方都有定义(接口内容都相同)，这里统一换成了Spring的定义

2、DynamicAdvisedInterceptor#intercept

1. **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {
2. Object oldProxy = **null**;
3. **boolean** setProxyContext = **false**;
4. Class targetClass = **null**;
5. Object target = **null**;
7. Object argsToUse1;
8. **try** {
9. **if**(**this**.advised.exposeProxy) {
10. oldProxy = AopContext.setCurrentProxy(proxy);
11. setProxyContext = **true**;
12. }
14. target = **this**.getTarget();
15. **if**(target != **null**) {
16. targetClass = target.getClass();
17. }
19. //从advised中取拦截器，将Advisor转换成MethodInterceptor
20. List chain = **this**.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
21. Object retVal;
22. //如果没有AOP增强配置，那么直接调用target对象的方法
23. **if**(chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
24. Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
25. retVal = methodProxy.invoke(target, argsToUse);
26. } **else** {
27. //通过CglibMethodInvocation来启动advice增强
28. retVal = (**new** CglibAopProxy.CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy)).proceed();
29. }
31. retVal = CglibAopProxy.processReturnType(proxy, target, method, retVal);
32. argsToUse1 = retVal;
33. } **finally** {
34. **if**(target != **null**) {
35. **this**.releaseTarget(target);
36. }
38. **if**(setProxyContext) {
39. AopContext.setCurrentProxy(oldProxy);
40. }
42. }
44. **return** argsToUse1;
45. }

### 目标对象方法的调用

1、如果没有配置拦截器，那么会对目标对象的方法直接进行调用

2、对于JdkDynamicAopProxy代理对象，这个目标对象的方法调用是通过AopUtils使用反射机制在AopUtils.invokeJoinpointUsingReflection的方法中实现的

1. **public** **static** Object invokeJoinpointUsingReflection(Object target, Method method, Object[] args) **throws** Throwable {
2. **try** {
3. ReflectionUtils.makeAccessible(method);
4. **return** method.invoke(target, args);
5. } **catch** (InvocationTargetException var4) {
6. **throw** var4.getTargetException();
7. } **catch** (IllegalArgumentException var5) {
8. **throw** **new** AopInvocationException("AOP configuration seems to be invalid: tried calling method [" + method + "] on target [" + target + "]", var5);
9. } **catch** (IllegalAccessException var6) {
10. **throw** **new** AopInvocationException("Could not access method [" + method + "]", var6);
11. }
12. }

3、对于使用CglibAopProxy的代理对象，它对目标对象的调用是通过CGLIB的MethodProxy对象来直接完成的，这个对象的使用时由CGLIB的设计决定的。具体的调用在DynamicAdvisedInterceptor的intercept方法中可以看到，使用的是CGLIB封装好的功能

### AOP拦截器链的调用

1、AOP怎样完成对目标对象的增强，这些实现封装在AOP拦截器链中，由一个个具体的拦截器来完成

2、JDK和CGLIB会生成不同的AopProxy代理对象，从而构造了不同的回调方法来启动对拦截器链的调用

1. JdkDynamicAopProxy中的invoke方法
2. CglibAopProxy中使用DynamicAdvisedInterceptor的interceptor方法

* 虽然使用了不同的AopProxy代理对象，但最终对AOP拦截的处理殊途同归：它们对拦截器的调用都是从ReflectiveMethodInvocation中通过proceed方法实现的。在运行拦截器的拦截方法之前，需要对代理方法完成一个匹配判断，通过这个匹配判断来决定拦截器是否满足切面增强的要求
* 在Pointcut切点中需要进行matches的匹配过程，即matches调用对方法进行匹配判断，来决定是否需要实行Advice(增强)

3、ReflectiveMethodInvocation#proceed

1. **public** Object proceed() **throws** Throwable {
2. //从索引为-1的拦截器开始调用，并按序递增，如果拦截器链中的拦截器迭代调用完毕，这里开始调用target的函数，这个函数是通过反射机制完成的，具体实现在AopUtils.invokeJoinpointUsingReflection方法中
3. **if**(**this**.currentInterceptorIndex == **this**.interceptorsAndDynamicMethodMatchers.size() - 1) {
4. **return** **this**.invokeJoinpoint();
5. } **else** {
6. //这里沿着定义好的interceptorOrderInterceptionAdvice链进行处理
7. Object interceptorOrInterceptionAdvice = **this**.interceptorsAndDynamicMethodMatchers.get(++**this**.currentInterceptorIndex);
8. **if**(interceptorOrInterceptionAdvice **instanceof** InterceptorAndDynamicMethodMatcher) {
9. //这里对拦截器进行动态匹配判断，这里是触发匹配的地方，如果和定义的Pointcut匹配，那么这个advice将会得到执行，如果不匹配，那么proceed会被递归调用，直到所有拦截器都被运行过为止
10. InterceptorAndDynamicMethodMatcher dm = (InterceptorAndDynamicMethodMatcher)interceptorOrInterceptionAdvice;
11. **return** dm.methodMatcher.matches(**this**.method, **this**.targetClass, **this**.arguments)?dm.interceptor.invoke(**this**):**this**.proceed();
12. } **else** {
13. //如果是一个interceptor，直接调用这个interceptor对应的方法
14. **return** ((MethodInterceptor)interceptorOrInterceptionAdvice).invoke(**this**);
15. }
16. }
17. }

* ReflectiveMethodInvocation中存了一个字段，该字段可以保存拦截器(MethodInterceptor)也可以保存拦截器(MethodInterceptor)的封装类型InterceptorAndDynamicMethodMatcher(该类型保存了一个拦截器以及一个Matcher)

List<?> interceptorsAndDynamicMethodMatchers

1. **class** InterceptorAndDynamicMethodMatcher {
2. **final** MethodInterceptor interceptor;
3. **final** MethodMatcher methodMatcher;
5. **public** InterceptorAndDynamicMethodMatcher(MethodInterceptor interceptor, MethodMatcher methodMatcher) {
6. **this**.interceptor = interceptor;
7. **this**.methodMatcher = methodMatcher;
8. }
9. }

### 配置Advisor

1、在整个AopProxy代理对象的拦截回调过程中，先回到ReflectiveMethodInvocation#proceed方法。在这个方法里，可以看到得到了配置的interceptorOrInterceptionAdvice

1. Object interceptorOrInterceptionAdvice = **this**.interceptorsAndDynamicMethodMatchers.get(++**this**.currentInterceptorIndex);

* 这个interceptorOrInterceptionAdvice是获得的拦截器，它通过拦截器机制对目标对象的行为增强起作用
* 这个拦截器来自interceptorsAndDynamicMethodMatchers，具体地说，他是interceptorsAndDynamicMethodMatchers持有的List中的一个元素。那么关于如何配置拦截器的问题，就转化为这个List中的拦截器从哪来来，在哪里配置的问题

2、接着回到JdkDynamicAopProxy中的invoke方法，可以看到这个List中的interceptors是在哪个调用中获得的。对于CglibAopProxy，也有类似过程，只不过这个过程是在DynamicAdvisedInterceptor的interceptor回调中实现的

1. List chain = **this**.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

3、AdvisedSupport#getInterceptorsAndDynamicInterceptionAdvice

1. **public** List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, Class<?> targetClass) {
2. //这里使用了cache，利用cache取获取已有的interceptor链，但是第一次还是需要自己手动生成的，这个interceptor链是由advisorChainFactory完成的
3. //在这里使用的是DefaultAdvisorChainFactory
4. AdvisedSupport.MethodCacheKey cacheKey = **new** AdvisedSupport.MethodCacheKey(method);
5. List cached = (List)**this**.methodCache.get(cacheKey);
6. **if**(cached == **null**) {
7. cached = **this**.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(**this**, method, targetClass);
8. **this**.methodCache.put(cacheKey, cached);
9. }
11. **return** cached;
12. }

* 取得拦截器链的工作是由配置好的advisorChainFactory来完成的，它是一个生成Advisor链的工厂
* 在这里advisorChainFactory被配置成DefaultAdvisorChainFactory，实现了interceptor链的获取过程
* 在这个获取过程中，首先设置了一个List，其长度是由配置的Advisor个数来决定的，这个配置就是在XML中对ProxyFactoryBean的interceptNames属性的配置
* AdvisorAdapterRegistry对advice(增强)的织入功能起了很大的作用，后面会分析。有了AdvisorAdapterRegistry注册器，利用它对从ProxyFactoryBean配置中得到的Advice进行适配，从而获得相应的拦截器，再把它加入前面设置好的List中去，完成所谓拦截器注册过程
* 在拦截器适配和注册过程完成后，List中的拦截器会被JDK生成的AopProxy对象的invoke方法或者CGLIB代理对象的intercept拦截方法取得，并启动拦截器的invoke调用，最终触发Advice(增强)

4、DefaultAdvisorChainFactory#getInterceptorsAndDynamicInterceptionAdvice

1. **public** List<Object> getInterceptorsAndDynamicInterceptionAdvice(Advised config, Method method, Class<?> targetClass) {
2. //advisor链已经在config中持有了，这里可以直接使用
3. ArrayList interceptorList = **new** ArrayList(config.getAdvisors().length);
4. Class actualClass = targetClass != **null**?targetClass:method.getDeclaringClass();
5. **boolean** hasIntroductions = hasMatchingIntroductions(config, actualClass);
6. AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
7. Advisor[] var8 = config.getAdvisors();
8. **int** var9 = var8.length;
10. **for**(**int** var10 = 0; var10 < var9; ++var10) {
11. Advisor advisor = var8[var10];
12. MethodInterceptor[] interceptors1;
13. **if**(advisor **instanceof** PointcutAdvisor) {
14. PointcutAdvisor var20 = (PointcutAdvisor)advisor;
15. **if**(config.isPreFiltered() || var20.getPointcut().getClassFilter().matches(actualClass)) {
16. //拦截器链通过AdvisorAdapterRegistry来加入的，这个AdvisorAdapterRegistry对advice织入起了很大作用
17. interceptors1 = registry.getInterceptors(advisor);
18. MethodMatcher mm = var20.getPointcut().getMethodMatcher();
19. //使用MethodMatchers的matches方法进行匹配判断
20. **if**(MethodMatchers.matches(mm, method, actualClass, hasIntroductions)) {
21. **if**(mm.isRuntime()) {
22. MethodInterceptor[] var15 = interceptors1;
23. **int** var16 = interceptors1.length;
25. **for**(**int** var17 = 0; var17 < var16; ++var17) {
26. MethodInterceptor interceptor = var15[var17];
27. interceptorList.add(**new** InterceptorAndDynamicMethodMatcher(interceptor, mm));
28. }
29. } **else** {
30. interceptorList.addAll(Arrays.asList(interceptors1));
31. }
32. }
33. }
34. } **else** **if**(advisor **instanceof** IntroductionAdvisor) {
35. IntroductionAdvisor var19 = (IntroductionAdvisor)advisor;
36. **if**(config.isPreFiltered() || var19.getClassFilter().matches(actualClass)) {
37. interceptors1 = registry.getInterceptors(advisor);
38. interceptorList.addAll(Arrays.asList(interceptors1));
39. }
40. } **else** {
41. MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
42. interceptorList.addAll(Arrays.asList(interceptors));
43. }
44. }
46. **return** interceptorList;
47. }
49. //判断Advisors是否符合配置要求
50. **private** **static** **boolean** hasMatchingIntroductions(Advised config, Class<?> actualClass) {
51. **for**(**int** i = 0; i < config.getAdvisors().length; ++i) {
52. Advisor advisor = config.getAdvisors()[i];
53. **if**(advisor **instanceof** IntroductionAdvisor) {
54. IntroductionAdvisor ia = (IntroductionAdvisor)advisor;
55. **if**(ia.getClassFilter().matches(actualClass)) {
56. **return** **true**;
57. }
58. }
59. }
61. **return** **false**;
62. }

5、ProxyFactoryBean的getObject方法中对advisor进行初始化时，从XML配置中获取了advisor，在初始化advisor中，可以看到对IoC容器的一个getBean回调，通过对这个IoC容器的getBean调用来得到配置好的advisor，下面是ProxyFactoryBean#initializeAdvisorChain

1. **private** **synchronized** **void** initializeAdvisorChain() **throws** AopConfigException, BeansException {
2. **if**(!**this**.advisorChainInitialized) {
3. **if**(!ObjectUtils.isEmpty(**this**.interceptorNames)) {
4. **if**(**this**.beanFactory == **null**) {
5. **throw** **new** IllegalStateException("No BeanFactory available anymore (probably due to serialization) - cannot resolve interceptor names " + Arrays.asList(**this**.interceptorNames));
6. }
8. **if**(**this**.interceptorNames[**this**.interceptorNames.length - 1].endsWith("\*") && **this**.targetName == **null** && **this**.targetSource == EMPTY\_TARGET\_SOURCE) {
9. **throw** **new** AopConfigException("Target required after globals");
10. }
12. String[] var1 = **this**.interceptorNames;
13. **int** var2 = var1.length;
15. **for**(**int** var3 = 0; var3 < var2; ++var3) {
16. String name = var1[var3];
17. **if**(**this**.logger.isTraceEnabled()) {
18. **this**.logger.trace("Configuring advisor or advice \'" + name + "\'");
19. }
21. **if**(name.endsWith("\*")) {
22. **if**(!(**this**.beanFactory **instanceof** ListableBeanFactory)) {
23. **throw** **new** AopConfigException("Can only use global advisors or interceptors with a ListableBeanFactory");
24. }
26. **this**.addGlobalAdvisor((ListableBeanFactory)**this**.beanFactory, name.substring(0, name.length() - "\*".length()));
27. } **else** {
28. //需要对Bean的类型进行判断，是单例类型还是prototype类型
29. Object advice;
30. //这里是取得advisor的地方，是通过beanFactory取得的，把interceptorNames这个List中的interceptor名字交给beanFactory，然后通过调用beanFactory的getBean去获取
31. **if**(!**this**.singleton && !**this**.beanFactory.isSingleton(name)) {
32. advice = **new** ProxyFactoryBean.PrototypePlaceholderAdvisor(name);
33. } **else** {
34. advice = **this**.beanFactory.getBean(name);
35. }
37. **this**.addAdvisorOnChainCreation(advice, name);
38. }
39. }
40. }
42. **this**.advisorChainInitialized = **true**;
43. }
44. }

* advisor通知器取得的是委托给IoC容器完成，但是在ProxyFactoryBean是如何获得IoC容器，然后通过回调IoC容器的getBean方法来得到需要的Advisor呢？
* 这涉及到IoC容器实现原理，在使用DefaultListableBeanFactory作为IoC容器时，它的基类是AbstractAutowireCapableBeanFactory，在这个基类中可以看到一个对Bean进行初始化的initializeBean方法
* 在这个Bean的初始化过程中，对IoC容器在Bean中的回调进行了设置，这个设置很简单，判断这个Bean是否实现了BeanFactoryAware接口，如果是，那么它一定实现了BeanFactoryAware定义的接口方法，通过这个接口方法，可以把IoC容器设置到Bean自身定义的一个属性中。这样，在这个Bean的自身实现中，就能够得到它所在的IoC容器，从而调用IoC容器的getBean方法，完成对IoC容器的回调

### Advice的实现

1、在AopProxy代理对象生成时，其拦截器也同样建立起来了，本小结关注的问题是：Spring AOP定义的通知是怎样实现对目标对象的增强

2、在为AopProxy代理对象配置拦截器的实现中，有一个取得拦截器的配置过程，这个过程时由DefaultAdvisorChainFactory实现的，而这个工厂类负责生成拦截器链，在它的getInterceptorsAndDynamicInterceptionAdvice方法中，有一个适配和注册过程，在这个适配和注册过程下，通过配置Spring预先设计好的拦截器，Spring加入了它对AOP实现的处理

3、为了详细了解这个过程，从DefaultAdvisorChainFactory的实现开始，重新回顾getInterceptorsAndDynamicInterceptionAdvice方法

1. **public** List<Object> getInterceptorsAndDynamicInterceptionAdvice(Advised config, Method method, Class<?> targetClass) {
2. ArrayList interceptorList = **new** ArrayList(config.getAdvisors().length);
3. Class actualClass = targetClass != **null**?targetClass:method.getDeclaringClass();
4. **boolean** hasIntroductions = hasMatchingIntroductions(config, actualClass);
6. //得到注册器GlobalAdivosrAdapterRegistry，这是一个单例模式的实现
7. AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
8. Advisor[] var8 = config.getAdvisors();
9. **int** var9 = var8.length;
11. **for**(**int** var10 = 0; var10 < var9; ++var10) {
12. Advisor advisor = var8[var10];
13. MethodInterceptor[] interceptors1;
14. **if**(advisor **instanceof** PointcutAdvisor) {
15. PointcutAdvisor var20 = (PointcutAdvisor)advisor;
16. **if**(config.isPreFiltered() || var20.getPointcut().getClassFilter().matches(actualClass)) {
17. //从GlobalAdvisorAdapterRegistry中取得MethodInterceptor的实现
18. interceptors1 = registry.getInterceptors(advisor);
19. MethodMatcher mm = var20.getPointcut().getMethodMatcher();
20. **if**(MethodMatchers.matches(mm, method, actualClass, hasIntroductions)) {
21. **if**(mm.isRuntime()) {
22. //遍历getInterceptors()方法取得的对象，包装成InterceptorAndDynamicMethodMatcher对象
23. MethodInterceptor[] var15 = interceptors1;
24. **int** var16 = interceptors1.length;
26. **for**(**int** var17 = 0; var17 < var16; ++var17) {
27. MethodInterceptor interceptor = var15[var17];
28. interceptorList.add(**new** InterceptorAndDynamicMethodMatcher(interceptor, mm));
29. }
30. } **else** {
31. interceptorList.addAll(Arrays.asList(interceptors1));
32. }
33. }
34. }
35. } **else** **if**(advisor **instanceof** IntroductionAdvisor) {
36. IntroductionAdvisor var19 = (IntroductionAdvisor)advisor;
37. **if**(config.isPreFiltered() || var19.getClassFilter().matches(actualClass)) {
38. interceptors1 = registry.getInterceptors(advisor);
39. interceptorList.addAll(Arrays.asList(interceptors1));
40. }
41. } **else** {
42. MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
43. interceptorList.addAll(Arrays.asList(interceptors));
44. }
45. }
47. **return** interceptorList;
48. }

* 在GlobalAdvisorAdapterRegistry中隐藏着不少AOP实现的重要细节，它的getInterceptors方法为AOP实现做了很大的贡献，就是这个方法封装着advice织入实现的入口

4、GlobalAdvisorAdapterRegistry实现分析

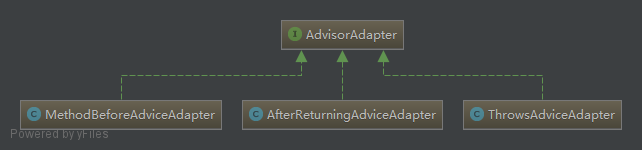
1. **public** **abstract** **class** GlobalAdvisorAdapterRegistry {
2. //单例模式的典型实现(这种实现没有延迟加载功能，如果将字段保留为内部静态类的私有字段，则还可以达到延迟加载的功能)，使用静态类变量来保持唯一一个实例，然后还将类设为抽象类，禁止该类创建实例，但是不妨碍抽象类的静态成员的使用
3. **private** **static** AdvisorAdapterRegistry instance = **new** DefaultAdvisorAdapterRegistry();
5. **public** GlobalAdvisorAdapterRegistry() {
6. }
7. //返回单例的实例
8. **public** **static** AdvisorAdapterRegistry getInstance() {
9. **return** instance;
10. }
12. **static** **void** reset() {
13. instance = **new** DefaultAdvisorAdapterRegistry();
14. }
15. }

* 该类的作用就是提供单例的DefaultAdvisorAdapterRegistry

5、DefaultAdvisorAdapterRegistry设置了一系列adapter适配器，正是这些adapter适配器的实现，为Spring AOP的advice提供编织能力

* 该类中有一系列在AOP应用中与用到的Spring AOP的advice相对应的apater适配实现，并看到了这些adapter的具体使用，具体使用主要体现在以下两个方面

1. 调用adapter的upport方法，通过这个方法来判断取得的advice属于什么类型的advice，从而根据不同的advice类型来注册不同的AdviceInterceptor，也就是前面看到的哪些拦截器
2. 这些AdviceInterceptor都是String AOP框架设计好的，是为实现不同的advice功能提供服务的，有了这些AdviceInceptor，可以方便地使用由Spring提供的各种不同advice来设计AOP应用。也就是说，正是这些AdviceInceptor最终实现了advice在AopProxy代理对象中的织入功能



6、DefaultAdvisorAdapterRegistry实现分析(Important)

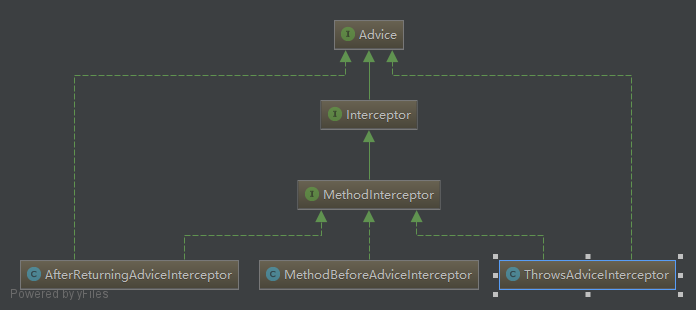
1. **public** **class** DefaultAdvisorAdapterRegistry **implements** AdvisorAdapterRegistry, Serializable {
2. //持有一个AdvisorAdapter的List，这个List中的Adapter是与实现Spring AOP的Advice增强功能相对应的
3. **private** **final** List<AdvisorAdapter> adapters = **new** ArrayList(3);
5. //这里把已有的Advice实现的Adapter加进来，包括MethodBeforeAdvice、AfterReturningAdvice、ThrowsAdvice等AOP的advice封装实现
6. **public** DefaultAdvisorAdapterRegistry() {
7. **this**.registerAdvisorAdapter(**new** MethodBeforeAdviceAdapter());
8. **this**.registerAdvisorAdapter(**new** AfterReturningAdviceAdapter());
9. **this**.registerAdvisorAdapter(**new** ThrowsAdviceAdapter());
10. }
12. **public** Advisor wrap(Object adviceObject) **throws** UnknownAdviceTypeException {
13. **if**(adviceObject **instanceof** Advisor) {
14. **return** (Advisor)adviceObject;
15. } **else** **if**(!(adviceObject **instanceof** Advice)) {
16. **throw** **new** UnknownAdviceTypeException(adviceObject);
17. } **else** {
18. Advice advice = (Advice)adviceObject;
19. **if**(advice **instanceof** MethodInterceptor) {
20. **return** **new** DefaultPointcutAdvisor(advice);
21. } **else** {
22. Iterator var3 = **this**.adapters.iterator();
24. AdvisorAdapter adapter;
25. **do** {
26. **if**(!var3.hasNext()) {
27. **throw** **new** UnknownAdviceTypeException(advice);
28. }
30. adapter = (AdvisorAdapter)var3.next();
31. } **while**(!adapter.supportsAdvice(advice));
33. **return** **new** DefaultPointcutAdvisor(advice);
34. }
35. }
36. }
38. //这里是在DefaultAdvisorChainFactory中调用的getInterceptors方法
39. **public** MethodInterceptor[] getInterceptors(Advisor advisor) **throws** UnknownAdviceTypeException {
40. ArrayList interceptors = **new** ArrayList(3);
41. //从Advisor配置中获得Advice
42. Advice advice = advisor.getAdvice();
43. //如果通知是MethodInterceptor类型的通知，直接加入interceptors的List中，不需要适配
44. **if**(advice **instanceof** MethodInterceptor) {
45. interceptors.add((MethodInterceptor)advice);
46. }
48. //遍历所有适配器，使用已经配置好的Adapter：MethodBeforeAdviceAdapter、AfterReturningAdviceAdapter以及ThrowsAdviceAdapter，然后从对应的adapter中取出封装好AOP编制功能的拦截器(对应给定的Advice，它一定属于某个适配器)
49. Iterator var4 = **this**.adapters.iterator();
51. **while**(var4.hasNext()) {
52. AdvisorAdapter adapter = (AdvisorAdapter)var4.next();
53. //下面的判断在整个while遍历中只会成功一次
54. **if**(adapter.supportsAdvice(advice)) {
55. interceptors.add(adapter.getInterceptor(advisor));
56. }
57. }
59. **if**(interceptors.isEmpty()) {
60. **throw** **new** UnknownAdviceTypeException(advisor.getAdvice());
61. } **else** {
62. **return** (MethodInterceptor[])interceptors.toArray(**new** MethodInterceptor[interceptors.size()]);
63. }
64. }
66. **public** **void** registerAdvisorAdapter(AdvisorAdapter adapter) {
67. **this**.adapters.add(adapter);
68. }
69. }

* 将不同的增强接口适配成有统一接口的MethodInterceptor(即拦截器)

7、以MethodBeforeAdviceAdapter的实现分析为例

1. **class** MethodBeforeAdviceAdapter **implements** AdvisorAdapter, Serializable {
2. MethodBeforeAdviceAdapter() {
3. }
5. **public** **boolean** supportsAdvice(Advice advice) {
6. **return** advice **instanceof** MethodBeforeAdvice;
7. }
9. **public** MethodInterceptor getInterceptor(Advisor advisor) {
10. MethodBeforeAdvice advice = (MethodBeforeAdvice)advisor.getAdvice();
11. **return** **new** MethodBeforeAdviceInterceptor(advice);
12. }
13. }

* 它实现了AdvisorAdapter的两个接口方法：一个是supportsAdvice，这个方法对advice的类型进行判断，如果advice是MethodBeforeAdvice的实例，那么返回true
* 另一个是对getInterceptor接口方法的实现，这个方法把Advice从Adivisor中取出来，然后创建一个MethodBeforeAdviceInterceptor对象，通过这个对象把取得的advice包装起来



8、Spring AOP为了实现advice的织入，设计了特定的适配器对这些功能进行了封装，应用虽然不会直接使用这些适配器(非public class)，但确是advice能够发挥作用比不可少的准备，仍然以MethodBeforeAdviceInterceptor为例进行分析

1. **public** **class** MethodBeforeAdviceInterceptor **implements** MethodInterceptor, Serializable {
2. **private** MethodBeforeAdvice advice;
4. //为指定的Advice创建对应的MethodBeforeAdviceInterceptor对象
5. **public** MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
6. Assert.notNull(advice, "Advice must not be null");
7. **this**.advice = advice;
8. }
10. //这个invoke方法是拦截器的回调方法，会在代理对象的方法被调用时触发回调
11. **public** Object invoke(MethodInvocation mi) **throws** Throwable {
12. **this**.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());
13. **return** mi.proceed();
14. }
15. }

* MethodBeforeAdviceInterceptor完成的是对MethodBeforeAdvice通知的封装
* 可以在MethodBeforeAdviceInterceptor设计的invoke回调方法中，看到首先触发了advice的before的回调，然后才是MethodInvocation的proceed方法调用
* 这就与之前的ReflectiveMethodInvocation#proceed分析联系起来，在AopProxy代理对象触发的ReflectiveMethodInvocation的proceed方法中，在取得拦截器以后，启动了对拦截器invoke方法的调用。按照AOP的配置规则，ReflectiveMethodInvocation触发的拦截器invoke方法，最终会根据不同的advice类型，触发Spring对不同的advice的拦截器封装，比如对MethodBeforeAdvice，最终会触发MethodBeforeAdviceInterceptor的invoke方法，该方法会先调用advice的before方法，这就是MethodBeforeAdvice所需要的对目标对象的增强效果：在方法调用之前通知增强

9、其他MethodInterceptor的实现类也类似，例如AfterReturningAdviceInterceptor的实现

1. **public** **class** AfterReturningAdviceInterceptor **implements** MethodInterceptor, AfterAdvice, Serializable {
2. **private** **final** AfterReturningAdvice advice;
4. //为指定advice创建AfterReturningAdviceInterceptor对象
5. **public** AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {
6. Assert.notNull(advice, "Advice must not be null");
7. **this**.advice = advice;
8. }
10. **public** Object invoke(MethodInvocation mi) **throws** Throwable {
11. //这里先传递拦截器链的调用
12. Object retVal = mi.proceed();
13. //然后再回调advice的后置增强，这就是后置增强的体现
14. **this**.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.getThis());
15. **return** retVal;
16. }
17. }

* AfterReturningAdviceInterceptor与MethodBeforeAdviceInterceptor不同的地方在于AfterReturningAdviceInterceptor#invoke方法先完成MethodInterceptor的proceed调用，然后再执行advice的afterReturning回调

10、ThrowsAdviceInterceptor回调的方法调用要复杂一些，它维护了一个exceptionHandlerMap来对应不同的方法调用场景，这个exceptionHandlerMap中handler的取得是与处罚ThrowsAdvice增强的异常相关的

1. **public** **class** ThrowsAdviceInterceptor **implements** MethodInterceptor, AfterAdvice {
2. **private** **static** **final** String AFTER\_THROWING = "afterThrowing";
3. **private** **static** **final** Log logger = LogFactory.getLog(ThrowsAdviceInterceptor.**class**);
4. **private** **final** Object throwsAdvice;
5. **private** **final** Map<Class<?>, Method> exceptionHandlerMap = **new** HashMap();
7. **public** ThrowsAdviceInterceptor(Object throwsAdvice) {
8. Assert.notNull(throwsAdvice, "Advice must not be null");
9. **this**.throwsAdvice = throwsAdvice;
10. //配置ThrowsAdvice的回调方法
11. Method[] methods = throwsAdvice.getClass().getMethods();
12. Method[] var3 = methods;
13. **int** var4 = methods.length;
15. **for**(**int** var5 = 0; var5 < var4; ++var5) {
16. Method method = var3[var5];
17. **if**(method.getName().equals("afterThrowing") && (method.getParameterTypes().length == 1 || method.getParameterTypes().length == 4) && Throwable.**class**.isAssignableFrom(method.getParameterTypes()[method.getParameterTypes().length - 1])) {
18. **this**.exceptionHandlerMap.put(method.getParameterTypes()[method.getParameterTypes().length - 1], method);
19. **if**(logger.isDebugEnabled()) {
20. logger.debug("Found exception handler method: " + method);
21. }
22. }
23. }
25. **if**(**this**.exceptionHandlerMap.isEmpty()) {
26. **throw** **new** IllegalArgumentException("At least one handler method must be found in class [" + throwsAdvice.getClass() + "]");
27. }
28. }
30. **public** **int** getHandlerMethodCount() {
31. **return** **this**.exceptionHandlerMap.size();
32. }
34. **private** Method getExceptionHandler(Throwable exception) {
35. Class exceptionClass = exception.getClass();
36. **if**(logger.isTraceEnabled()) {
37. logger.trace("Trying to find handler for exception of type [" + exceptionClass.getName() + "]");
38. }
40. Method handler;
41. **for**(handler = (Method)**this**.exceptionHandlerMap.get(exceptionClass); handler == **null** && exceptionClass != Throwable.**class**; handler = (Method)**this**.exceptionHandlerMap.get(exceptionClass)) {
42. exceptionClass = exceptionClass.getSuperclass();
43. }
45. **if**(handler != **null** && logger.isDebugEnabled()) {
46. logger.debug("Found handler for exception of type [" + exceptionClass.getName() + "]: " + handler);
47. }
49. **return** handler;
50. }
52. **public** Object invoke(MethodInvocation mi) **throws** Throwable {
53. //把对目标对象的方法调用放入try/catch中，并在catch中触发ThrowsAdvice的回调，把异常接着向外抛出，不做太多处理
54. **try** {
55. **return** mi.proceed();
56. } **catch** (Throwable var4) {
57. Method handlerMethod = **this**.getExceptionHandler(var4);
58. **if**(handlerMethod != **null**) {
59. **this**.invokeHandlerMethod(mi, var4, handlerMethod);
60. }
62. **throw** var4;
63. }
64. }
66. //通过反射启动对ThrowsAdvice回调方法的调用
67. **private** **void** invokeHandlerMethod(MethodInvocation mi, Throwable ex, Method method) **throws** Throwable {
68. Object[] handlerArgs;
69. **if**(method.getParameterTypes().length == 1) {
70. handlerArgs = **new** Object[]{ex};
71. } **else** {
72. handlerArgs = **new** Object[]{mi.getMethod(), mi.getArguments(), mi.getThis(), ex};
73. }
75. **try** {
76. method.invoke(**this**.throwsAdvice, handlerArgs);
77. } **catch** (InvocationTargetException var6) {
78. **throw** var6.getTargetException();
79. }
80. }
81. }