**1.Spring AOP的概述**

Aspect是一种新的模块化机制，用来描述分散在对象、类或函数中的横切关注点

使用AOP后，不仅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。

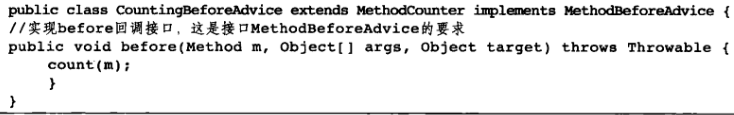
**1.2Advice通知**

定义连接点在做什么，为切面增强提供织入接口，例如BeforeAdvice、AfterAdvice、ThrowsAdvice等；

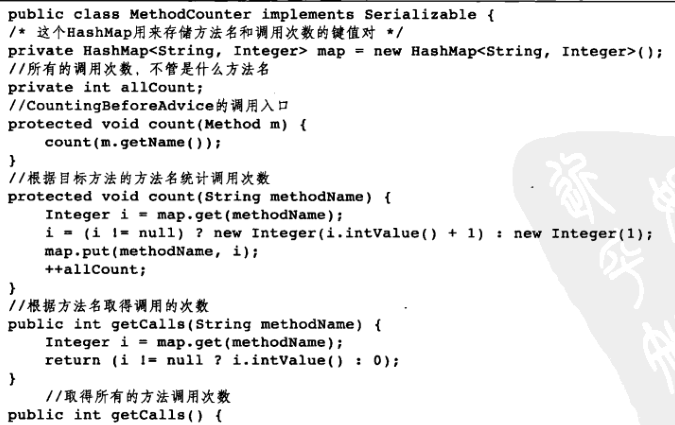
BeforeAdvice继承体系中，定义了为待增强的目标方法设置的前置增强接口MethodBeforeAdvice，使用这个前置接口需要实现一个回调函数



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，具体调用参数：method对象，这个参数是目标方法的反射对象；Object[]对象数组，包含目标方法的形参；实现一个功能统计被调用的方法次数，根据调用方法的方法名进行统计，把统计结果根据方法名，调用次数放入map中；CountingBeforeAdvice是接口MethodBeforeAdvice的实现类；



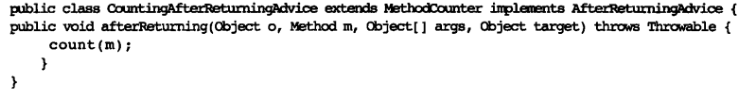
Count是MethodCounter类中的方法；首先通过目标方法的反射对象，得到方法名，然后进行累加，把统计结果放到维护的map中



AfterAdvice通知类型，AfterReturningAdvice是比较常用的一个，接口中定义了一个方法

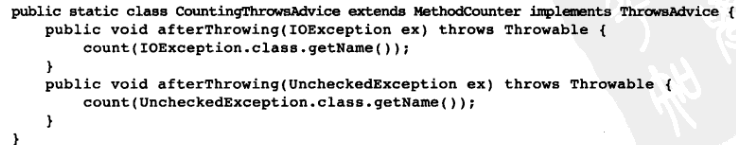


在这个Advice通知被正确配置以后，在目标方法调用结束并成功返回的时候，接口会被AOP回调。回调参数，目标方法的返回结果，方法的反射对象，方法参数，调用target里面的该方法。依然用上面的例子



方法调用和前面一样，只是发生的时间不一样。如果实现不同的aop通知接口，就会被aop编织到不同的调用场合中。一个是在方法调用前实现切面增强，一个是在方法调用返回结果后实现增强，相同代码可以灵活出现在不同的应用场合。

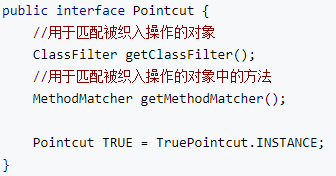
ThrowsAdvice，没有指定需要实现的接口方法，他在抛出异常时被回调，这个回调是aop使用反射机制来完成的。



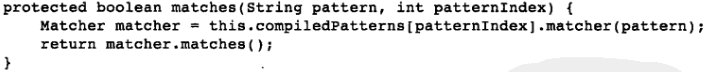
**1.3 Pointcut切点**

**决定Advice通知应该作用于哪个连接点**，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通常意味着标识的方法。

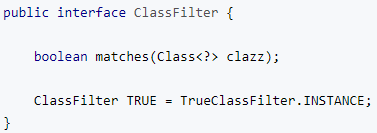
Pointcut接口定义中，需要返回一个MethodMatcher，对于Point的方法匹配判断功能，就是用MethodMatcher实现的、以正则表达式切点JdkRegexMethodPointcut为例；JdkRegexMethodPointcut是StaticMatcherPointcut的子类，同时也是MethodMatcher的子类。



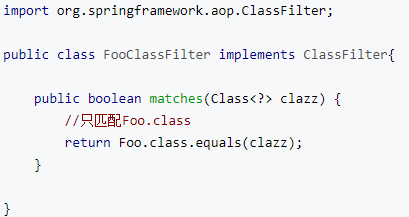
在Pointcut中，MethodMatcher对象实际上可以被配置成JdkRegexMethodPointcut来完成方法的匹配判断的。在JdkRegexMethodPointcut中，可以看到一个matches方法，这个matches方法是MethodMatcher定义的接口方法。JdkRegexMethodPointcut中，这个matches方法就是使用正则表达式来对方法名进行匹配的地方。



**ClassFilter**

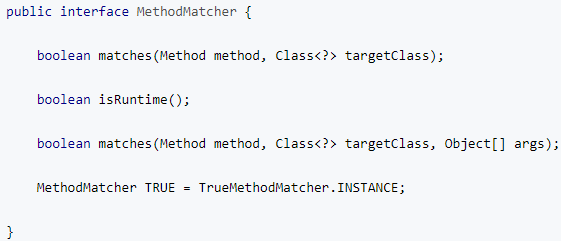


当织入的目标对象和Point指定的类型相同时，返回true，否则返回false，即意味着不会对这个类型的目标对象进行织入操作



如果类型对所捕捉的 Joinpoint 无所谓，那么 Pointcut 中使用的 ClassFilter 可以直接使用ClassFilter TRUE = TrueClassFilter.INSTANCE

**MethodMatcher**



MethodMatcher 通过重载，定义了两个 matches 方法，**而这两个方法的分界线就是 isRuntime 方法**，这里要特别注意！

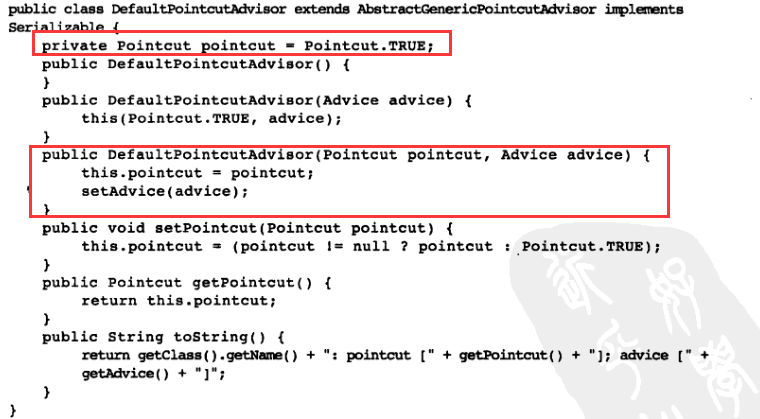
　　注意到三参数的matches方法中，最后一个参数是args，因此也就可以想到：两个 mathcers 方法的区别在于，**在进行方法拦截的时候，是否匹配方法的参数**

　　根据是否对方法的参数进行匹配，Pointcut可以分为StaticMethodMatcher和DynamicMethodMatcher，**当isRuntime()返回false，表明不对参数进行匹配，为StaticMethodMatcher，返回true时，表示要对参数进行匹配，为DynamicMethodMatcher。**

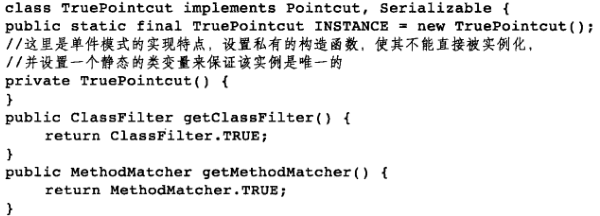
　　一般情况下，DynamicMethodMatcher会影响性能，所以我们一般使用StaticMethodMatcher就行了

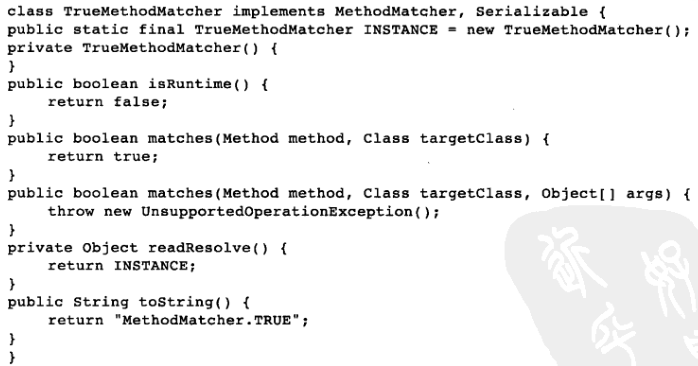
**1.4 Advisor通知器**

Advice 通知和Pointcut切点设计完成后，需要Advisor将他们连接起来。通过Advisor，可以定义应该使用哪个Advice并在哪个Pointcut使用它。以DefaultPointcutAdvisor为例，有两个属性，advice和pointcut，分别配置Advice和Pointcut。



Pointcut.True的值为 TruePointcut.INSTANCE;单例模式





**1.5. JoinPoint连接点**

被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法。AOP中仅支持方法级别的JoinPoint，更确切的说，只支持方法执行 （Method Execution ）类型的 Joinpoint，虽然 Spring AOP 仅提供方法拦截，但是实际的开发过程中，这已经可以满足 80% 的开发需求了。

　　Spring AOP 之所以如此，主要有以下几个原因。

　　1. Spring AOP 要提供一个简单而强大的 AOP 框架，并不想因大而全使得框架本身过于臃肿。能够仅付出 20% 的努力，就能够得到 80% 的回报。否则，事倍功半，并不是想看到的结果。

　　2. 对于类中属性 （Field ）级别的 Joinpoint ，如果提供这个级别的拦截，那么就破坏了面向对象的封装，而且，完全可以通过 setter 和 getter 方法的拦截达到同样的目的。

　　3. 如果应用需求非常特殊，完全超出了 Spring AOP 提供的那 80% 的需求支持，可以求助于现有其他 AOP 实现产品，如 AspectJ。 目前看来， AspectJ 是 Java 平台对 AOP 支持最完善的产品，同时，Spring AOP 也提供了对 Aspect的支持。

2**.Spring AOP的设计与实现**

**2.1 JVM动态代理特性**

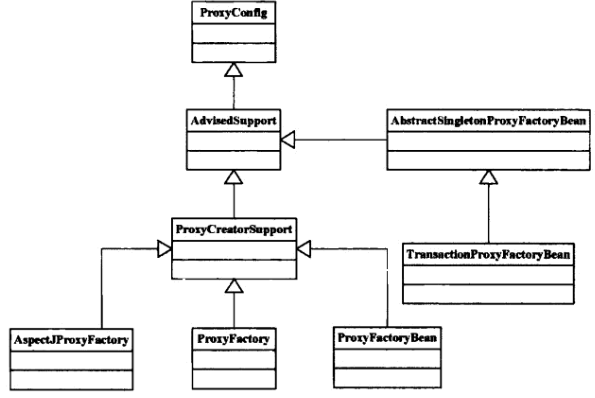
Reflection包中看到的Proxy对象，这个对象生成后，所起的作用就类似于Proxy模式中的Proxy对象，在使用时，还需要为代理对象设计一个回调方法，在其中加入作为代理需要额外处理的操作。Jdk中，需要实现下面所示的InvocationHandler接口



这个invoke方法第一个参数就是代理对象的实例，第二个是Method方法对象，代表的是当前代理对象中被调用的方法对应的接口方法对象；第三个为被调用的方法参数。

**3.建立AopProxy代理对象**

Spring的AOP模块中，一个主要的部分就是代理对象的生成，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。



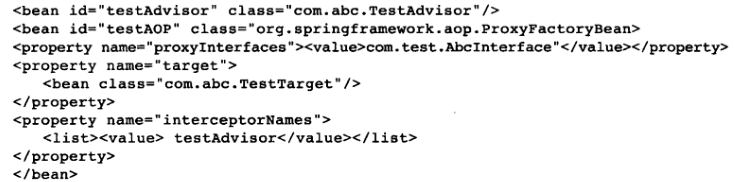
**3.1 配置ProxyFactoryBean**

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

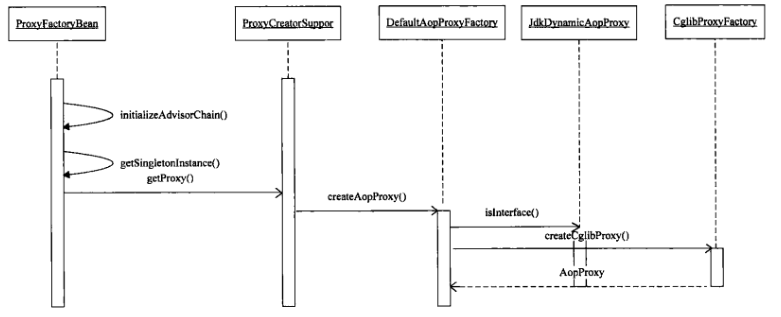
①定义使用的通知器Advisor，这个通知器应该作为一个bean来定义，通知器内部，定义了advice通知

②定义ProxyFactoryBean，把他作为另一个Bean，封装AOP功能的主要类，需要设置相关的属性，proxyInterface、interceptorNames和target，interceptorNames属性的值往往设置为需要**定义的通知器或者通知**，因为这些通知器在ProxyFactoryBean的aop配置下，通过使用代理对象的拦截器起作用。

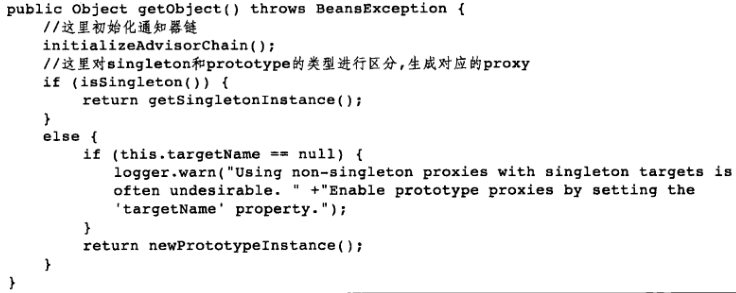
③定义target属性，作为target属性注入的bean，是需要用aop通知器中的切面应用来增强的对象，**接口的实现类**。



**3.2ProxyFactoryBean生成AopProxy代理对象**

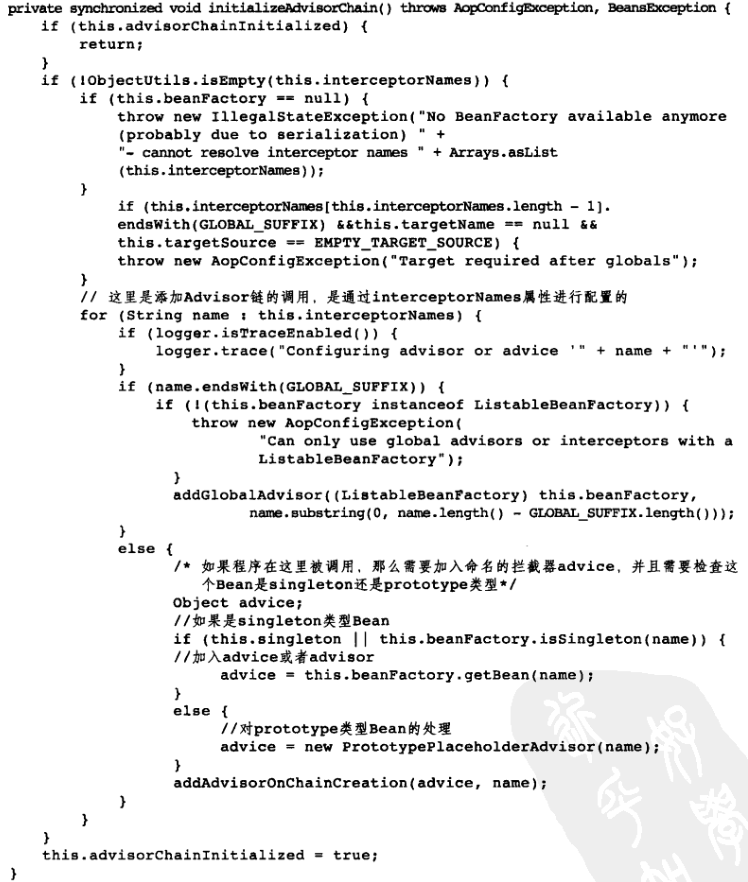


从FactoryBean中获取对象，是以getObject方法作为入口完成的，**ProxyFactoryBean**实现中的getObject方法，是FactoryBean需要实现的接口方法。对ProxyFactoryBean来说，把需要对target目标对象增加的通知处理，都通过getObject方法进行封装。



对通知器链进行初始化，通知器链封装了一系列的拦截器，这些拦截器都要从配置中读取，然后为代理对象的生成做好准备。Spring中有singleton和prototype类型的两种bean，需要区分。

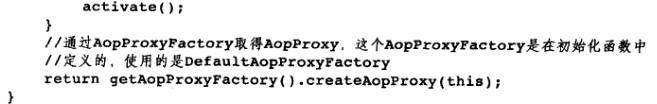
为Proxy代理对象配置Advisor链，是在initializeAdvisorChain方法中完成的，有一个标志位AdvisorChainInitialized，用来表示通知器链是否已经初始化，如果已经初始化，就直接返回。即初始化只发生在应用第一次通过ProxyFactoryBean去获取代理对象的时候。初始化之后，读取配置中出现的所有通知器，使用getBean就行，参数为通知器的名字，然后把从容器中取得通知器加入拦截器中，由addAdvisorOnChainCreation实现。



生成singleton的代理对象在getSingletonInstance代码中完成，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目标对象的方法调用，target对象的方法会被代理对象拦截。首先读取ProxyFactoryBean中的配置，为生成代理对象做好准备，设置方法调用接口等，







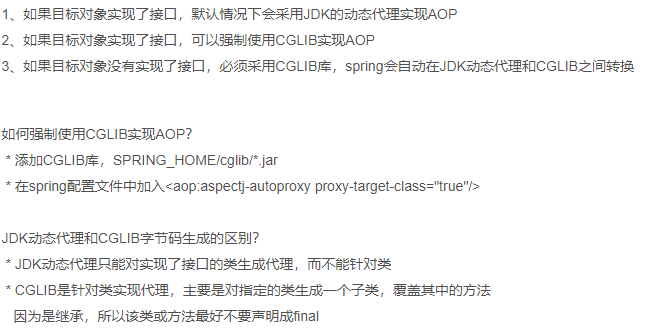
获取目标对象的接口，然后设置接口和classLoader，调用createAopProxy方法，内部使用AopProxyFactory取得AopProxy对象，然后调用getProxy获取得到代理对象；AopProxy是一个接口类，由两个子类实现，一个是cglib2AopProxy，另一个是JdkDynamicProxy；至于要生成什么样的代理对象，所有信息都封装在AdvisedSupport中，ProxyFactoryBean本身也是AdvisedSupport的子类。ProxyCreatorSupport是AdvisedSupport的子类，上面的this，就是ProxyCreatorSupport，也是createAopProxy的参数；

这里默认使用的是DefaultAopProxyFactory，这个AopProxyFactory，作为AopProxy的工厂对象，是在ProxyFactoryBean的基类ProxyCreatorSupport中被创建的，创建AopProxyFactory时，他被设置成了DefaultAopProxyFactory。

关于AopProxy代理对象的生成，如果目标对象时接口类，那么适合使用JDK来生成代理对象，否则Spring会使用CGlib来生成代理对象。具体实现过程交给了JDKDynamicAopProxy和CglibProxyFactory的createCglibProxy方法；

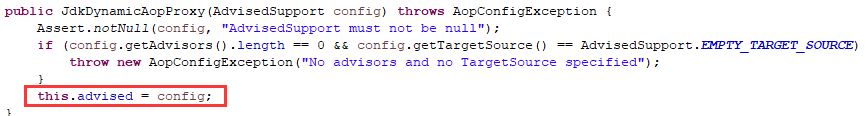


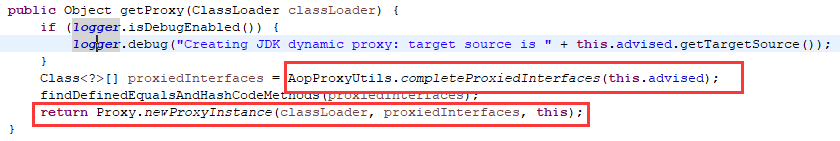
在AopProxy代理对象的生成过程中，首先要从AdvisedSupport对象中取得**配置的目标对象**，这个目标对象是实现AOP的所必须的，因为我们是要对目标对象进行增强。目标对象检查完成后，需要根据配置情况来决定使用什么方式来创建AopProxy代理对象。默认使用Jdk生成AopProxy代理对象，



**3.3 JDK生成AopProxy代理对象**

从ProxyFactoryBean的getObject开始，里面先对通知进行初始化，加入到拦截器中，然后判断singleton还是prototype，分情况讨论，如果是singleton，调用getSingletonInstance方法，这里会得到代理对象接口的类对象targetClass；设置代理对象的接口信息，然后调用getProxy（createAopProxy）方法，createAopProxy返回一个AopProxy对象，然后就会将ProxyFactoryBean对象作为AdvisedSupport参数传入，然后进行生成代理对象，config中包含着target相关信息和接口信息。

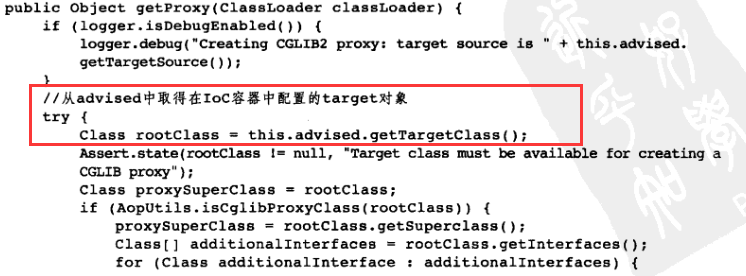




JdkDynamicAopProxy本身实现了AopProxy、InvocationHandler和序列化接口，实现了invoke回调方法，完成了代理对象和InvocationHandler即JdkDynamicAopProxy连接起来；

**3.4 CGLIB生成AopProxy对象**

CglibAopProxy类中的getProxy方法

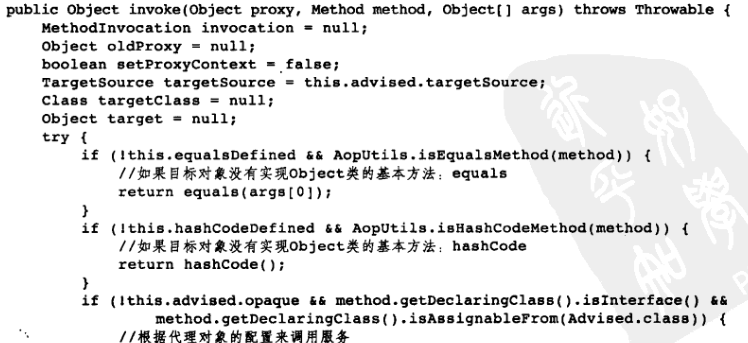




**4.Spring AOP拦截器调用的实现**

JDK需要通过InvocationHandler来设置拦截器回调，Cglib需要根据使用要求，通过DynamicAdvisedInterceptor来完成回调

**4.1JDKDynamicAopProxy的invoke拦截**





Invoke方法中，核心内容是获取拦截器链，如果拦截器链为空，直接调用目标方法，如果拦截器链不为空则先调用拦截器链，再对目标方法进行调用。

首先是获得拦截器链，从List<Object> chain = this.advised.

getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);方法中，可以看出获取interceptors的操作是由advised对象完成的，

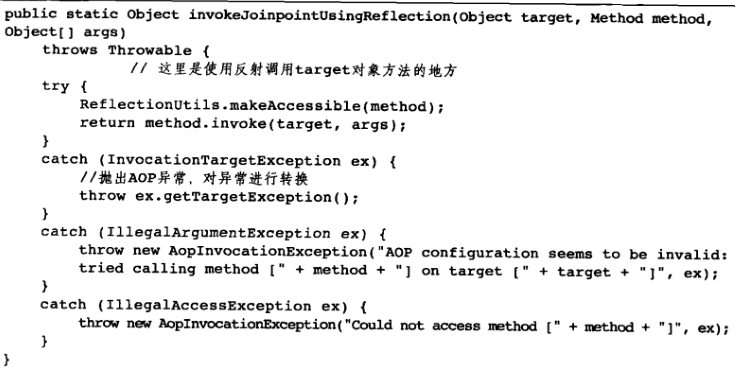
**4.2 cglibAopProxy的intercept拦截**



与JDK的拦截类似，只是拦截器链的处理不一样，这里采用CglibMethodInvocation完成

**4.3 目标对象方法的调用**

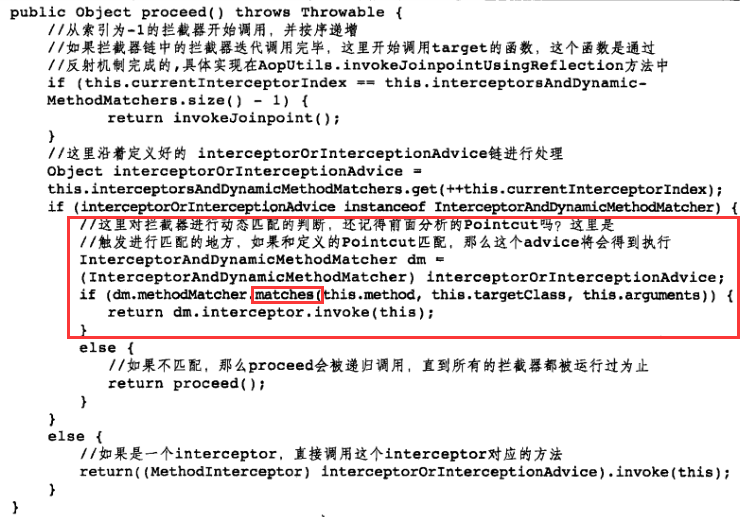
如果没有设置拦截器，会对目标对象的方法直接进行调用。使用AopUtils.invokeJoinpointUsingReflection的方法中实现的。



使用CglibAopProxy直接调用methodProxy.invoke，如果没有拦截器就会调用父类的方法，即target的方法

**4.4 AOP拦截器链的调用**

不同的AopProxy代理对象，但最终对Aop拦截的处理基本类似：他们对拦截器链的调用都是在ReflectiveMethodInvocation中通过proceed方法实现的，cglib的CglibMethodInvocation是ReflectiveMethodInvocation的子类。Proceed中会逐个运行拦截器中的拦截方法。在运行拦截器的方法之前，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要求。通过Pointcut的matches调用对方法进行匹配判断，来决定是否需要进行增强。



Proceed方法中，先进行判断，如果已经到拦截器末尾了，就直接调用目标对象的实现方法；否则沿着拦截器链继续进行，得到下一个拦截器，通过这个拦截器进行matches判断；判断是否适用于增强的场合，如果是，从拦截器中得到interceptor，调用invoke方法进行增强，如果不匹配，那么proceed会被递归调用，直到拦截器都被运行完了。

**4.5 配置通知器**

Proceed方法中得到了interceptorOrInterceptionAdvice拦截器，



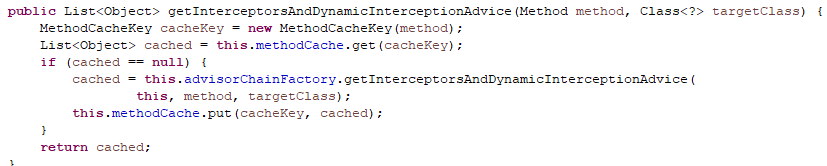
interceptorOrInterceptionAdvice是获得的拦截器，他通过拦截器机制对目标对象的行为增强起作用，这个拦截器来自interceptorsAndDynamicMethodMatchers，具体来说，他是interceptorsAndDynamicMethodMatchers持有的list中的一个元素；这个list是从哪里获得的，jdkDynamicAopProxy的invoke中



cglibAopProxy也有类似的，

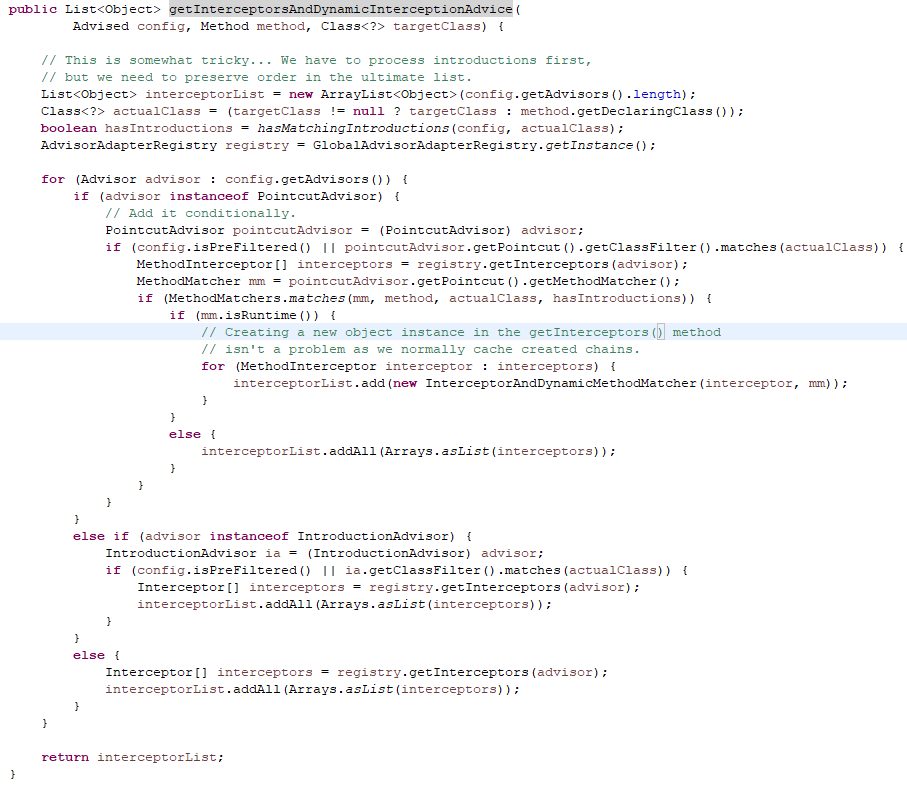


获取interceptors的操作是由advised对象来完成的，这个advise是一个AdvisedSupport对象，从类的继承上来看，这个AdvisedSupport类同时也是ProxyFactoryBean的基类。从AdvisedSupport的代码中可以看到getInterceptorsAndDynamicInterceptionAdvice的实现，代码清单如下。在这个方法中获得了拦截器链，在取得拦截器链的时候，为提高取得拦截器的效率，还为其设置了缓存。



这里使用了cache，利用cache获取已有的interceptor链，但是第一次还是需要自己动手生成。这个interceptor链的生成是由advisorChainFactory完成的，这里使用的是DefaultAdvisorChainFactory。在DefaultAdvisorChainFactory中实现了一个interceptor链的获取过程，代码如下。

在这个获取过程中，首先设置了一个list,其长度是由配置的通知器的个数来决定的，这个配置就是在XML中对ProxyFactoryBean做的interceptorNames属性的配置。然后DefaultAdvisorChainFactory会通过一个AdvisorAdapterRegistry来实现拦截器的注册，AdvisorAdapterRegistry对advice通知的织入功能起了很大的作用。有了AdvisorAdapterRegistry注册器，利用它来对从ProxyFactoryBean配置中得到的通知进行适配，从而获得相应的拦截器，再把他加入到前面设置好的List中去，完成所谓的拦截器注册过程。



获取到拦截器链之后，将目标对象，拦截器链，方法，代理对象等作为参数，创建一个ReflectiveMethodInvocation类，通过这个对象完成AOP功能实现的封装；

ProxyFactoryBean的getObject方法里面对advisor进行初始化，从xml配置中获取advisor通知器。

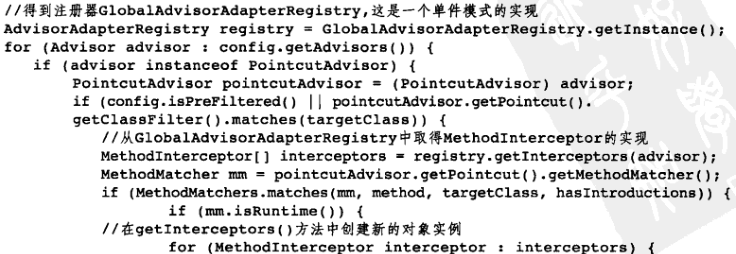


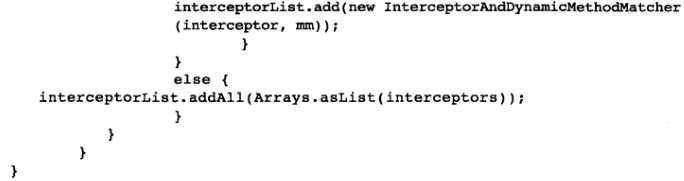
Advisor通知器的取得是委托给IoC容器完成的，但是在ProxyFactoryBean如何获的IoC容器，然后回调IoC容器的getBean方法取得advisor的呢？初始化bean的过程中，对IoC容器在bean 中的回调进行了设置，首先判断这个bean类型是不是实现BeanFactoryAware接口，如果是，那么它一定实现了BeanFactoryAware定义的接口方法，通过合格接口方法，可以把IoC容器设置到Bean自身定义的一个属性中去。那么在bean自身实现中，就能够得到它所在的IoC容器，从而调用容器的getBean方法



4.6 Advice通知的实现

在DefaultAdvisorChainFactory中，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，这个适配和注册过程中，通过配置Spring预先设计好的拦截器，Spring加入它对AOp的实现，可以看到，首先构造一个GlobalAdvisorAdapterRegistry单例，然后对配置的Advisor通知器进行逐个遍历，这些通知器链都是配置在interceptorNames中的，然后封装在Advised类型的参数config中，可以方便的取得配置的通知器，有了这些通知器，接着就是一个由GlobalAdvisorAdapterRegistry来完成拦截器的适配和注册过程。





GlobalAdvisorAdapterRegistry为AOP的实现做出了很大贡献，getInterceptors方法封装这advice织入实现的入口，先了解一下GlobalAdvisorAdapterRegistry单件，起到一个适配器的作用，但同时它也是一个单件模式的应用，为Spring AOP模块提供了一个DefaultAdvisorAdapterRegistry单件，由它完成各种通知的适配和注册工作。

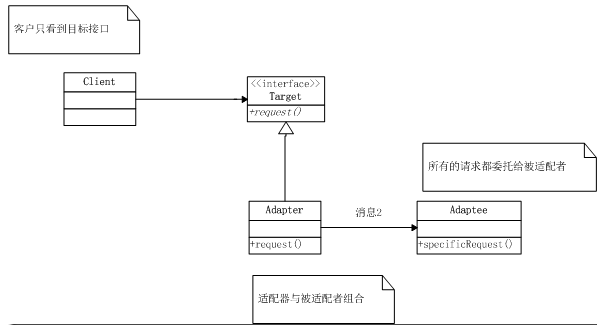


在DefaultAdvisorAdapterRegistry中，设置了一系列的adapter适配器，正是这些adapter的实现，为Spring aop的advice提供编织能力。含有与AOP对应的一系列通知相对应的adapter适配器，这些主要体现在一下两个方面：一是调用adapter的supportsAdvice方法，通过这个方法判断取得的advice是什么类型的advice通知；从而根据不同advice类型来注册不同的adviceInterceptor，加入到interceptors的list中。另一方面，这些AdviceInterceptor都是AOP框架设计好了的，是为实现不同的advice功能提供服务的，有了这些AdviceInterceptor，实现了advice通知在AopProxy代理对象的织入功能。





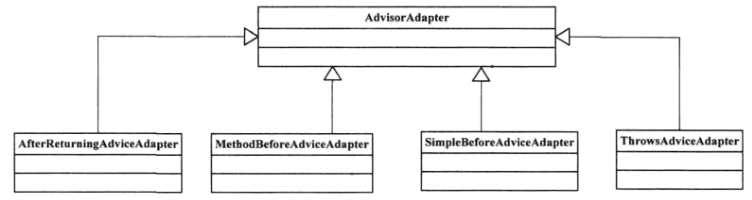
适配器模式，adaptee是一系列的advice



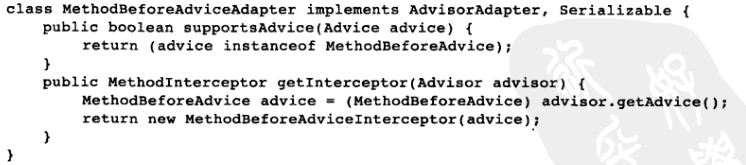
目标(Target)：这就是所期待得到的接口。Interceptor拦截器

源(Adaptee)角色：现在需要适配的接口。Advise通知

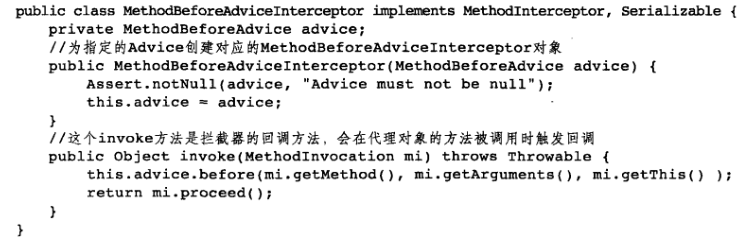
适配器(Adapter)：适配器类是本模式的核心。适配器把源接口转换成目标接口。**显然，这一角色不可以是接口，而必须是具体类。将对应的通知得到对应通知的拦截器**



在DefaultAdvisorAdapterRegistry的getInterceptors调用中，将几个AOP常用的adapter加入到adapter的list中，以MethodBeforeAdviceAdapter为例，实现了AdvisorAdapter的两个接口方法：supportsAdvice，对advice类型进行判断，如果advice是MethodBeforeAdvice的实例，那么返回true；另一个对getInterceptor接口方法的实现，这个方法把advice通知从通知器中取出，创建一个MethodBeforeAdviceInterceptor对象，通过这个对象将advice通知包装起来。

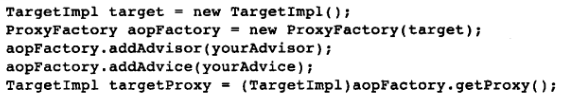


MethodBeforeAdviceInterceptor完成的是对MethodBeforeAdvice的封装，可以在MethodBeforeAdviceInterceptor设计的invoke回调方法中，看到首先出发了advice的before回调，然后才是MethodInvocation的proceed方法调用.这里就是和前面的ReflectiveMethodInvocation的proceed分析中联系起来了，从该类中的interceptorsAndDynamicMethodMatchers成员变量里面封装的都是interceptorsAndDynamicMethodMatcher对象，而对象中中封装着Interceptor和MethodMatcher，因此dm.interceptor.invoke,就是调用对应的拦截器的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MethodBeforeAdvice，最终会触发MethodBeforeAdviceInterceptor的invoke方法。在MethodBeforeAdviceInterceptor方法中，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：在方法调用之前完成通知增强。



**4.7 ProxyFactory实现AOP**

编程式使用ProxyFactory实现AOP





ProxyFactory实现AOP功能，其实现原理与ProxyFactoryBean的实现原理一致，只是在最外层的表现形式不同。ProxyFactory没有使用FactoryBean的IoC封装，而是通过继承ProxyCreatorSupport的功能来完成aop的属性配置。ProxyFactory的getProxy方法取得AopProxy代理对象，getProxy的实现使用了ProxyFactory基类ProxyCreatorSupport的createProxy方法来生成AopProxy代理对象，而AopProxy代理对象的生成是由AopProxyFactory来完成的，使用JDK或者CGLIB的代理对象。