# 接口解析

## org.aopalliance.aop.Advice接口

org.aopalliance.aop.Advice接口是AOP联盟定义的通知接口。

org.springframework.aop包中提供了一些继承自Advice接口的子接口，相应的类图如图1所示，在程序中可以通过实现这些子接口来进行AOP编程。

其类图如下：

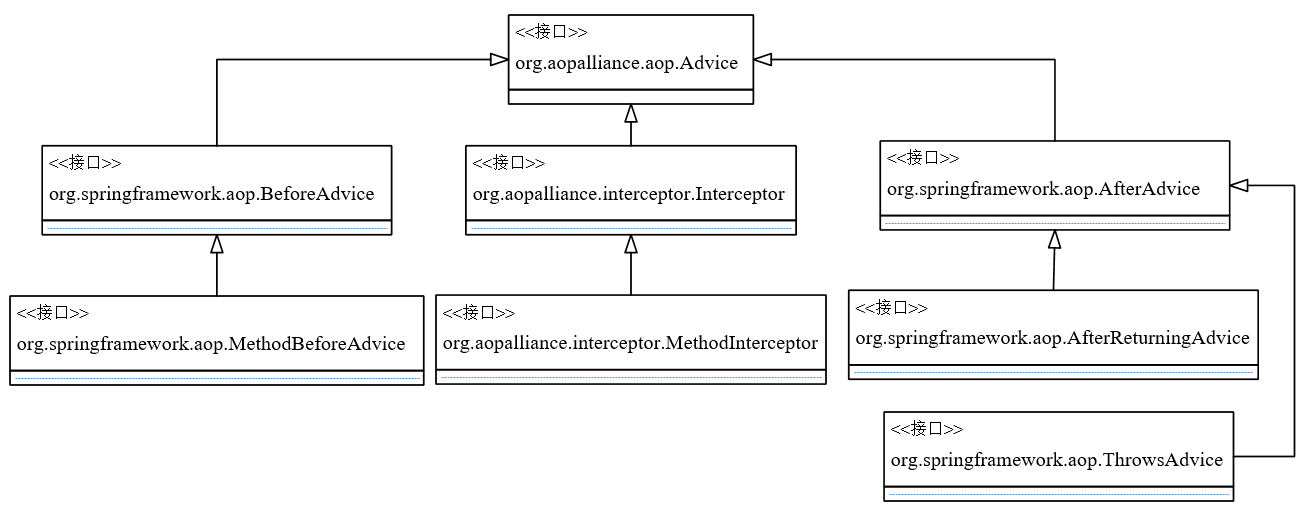


图1 org.springframework.aop包中提供的AOP编程接口

说明几点：

（1）Advice是AOP联盟定义的接口，Spring中的BeforeAdvice和AfterAdvice继承了该接口，并分别提供了相应的实现类，即MethodBeforeAdvice、MethodAfterReturningAdvice和ThrowsAdvice，在程序中可以通过实现以上三个接口来提供前置通知、返回通知和异常通知。

（2）MethodInterceptor也是AOP联盟定义的接口，在程序中可以通过实现该接口来提供环绕通知。

org.springframework.aop.aspectj包中提供了针对AspectJ的包装类，相应的类图如图2所示，在程序中可以通过@Before等注解或者在spring配置文件中通过<aop:config>等标签来进行AOP编程。

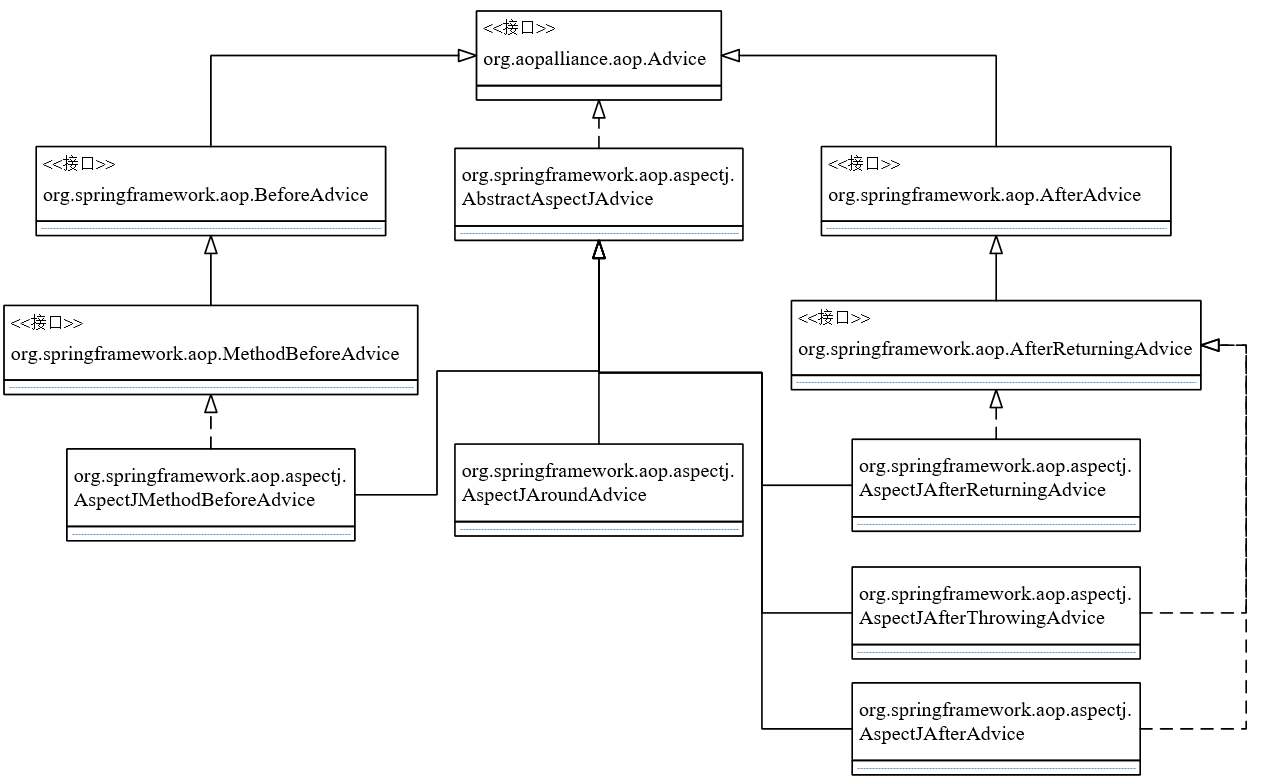


图2 org.springframework.aop.aspectj包中提供的针对AspectJ的包装类

## org.aopalliance.interceptor.MethodInterceptor接口

org.aopalliance.interceptor.MethodInterceptor接口是AOP联盟定义的接口，其类图如图3所示。

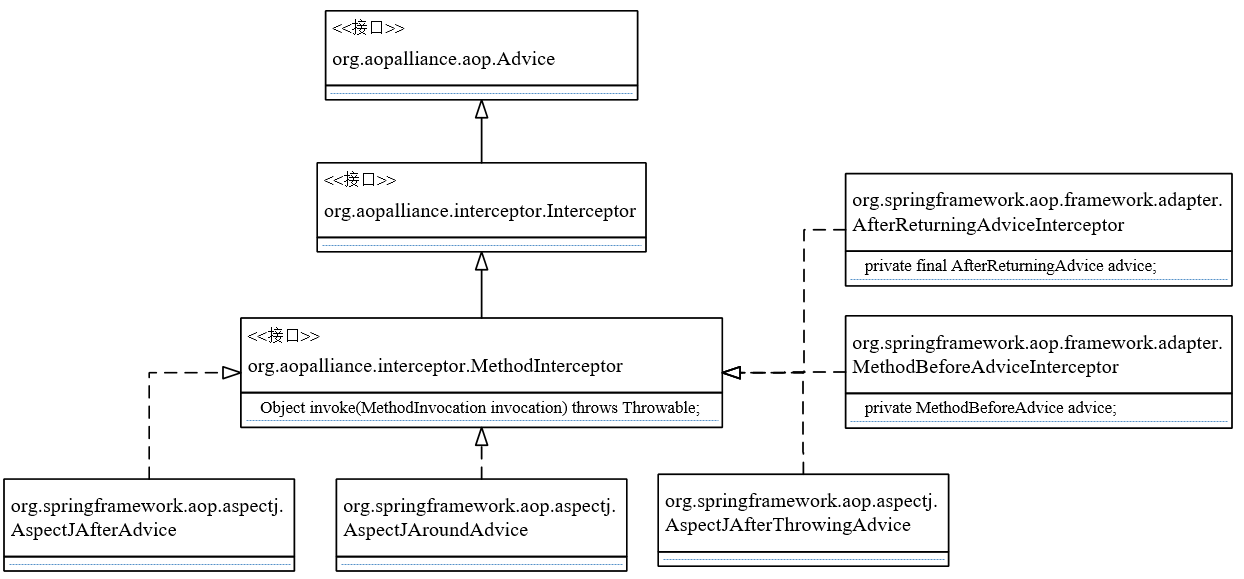


图3 org.aopalliance.interceptor.MethodInteceptor类图

可以看到，所有的Advice最终都会转化成MethodInterceptor，并执行其中的invoke方法。另外，也可以看到，Advice有两种方式转换成MethodInterceptor，一种是直接实现MethodInterceptor接口，例如AspectJAfterAdvice；一种是通过适配器模式来转换成MethodInterceptor。

## org.springframework.aop.Advisor接口

org.springframework.aop.Advisor接口的类图如图4所示。

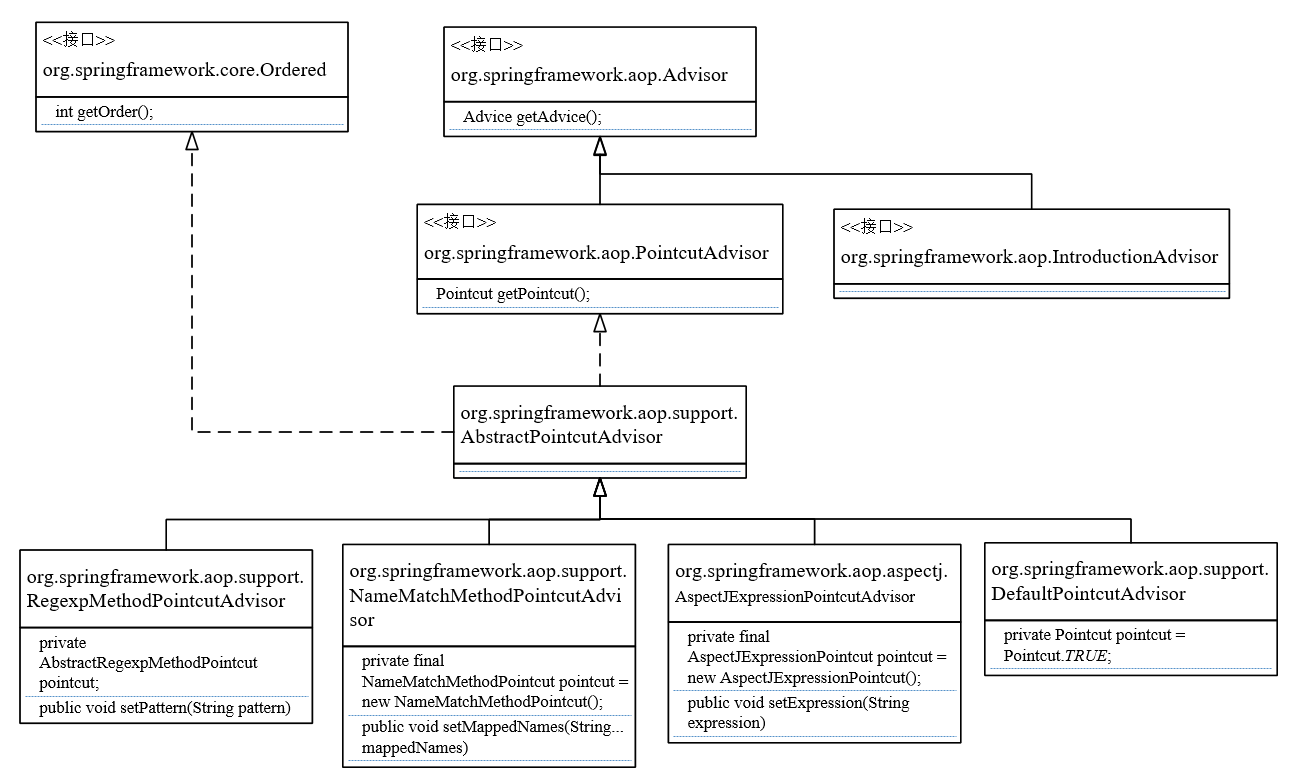


图4 org.springframework.aop.Advisor接口类图

可以看到，org.springframework.aop.Advisor接口本身只包含Advice，而org.springframework.aop.PointcutAdvisor接口则将Advice和Pointcut结合了起来。Spring提供了四种Advisor用来封装Pointcut和Advice，分别是：

org.springframework.aop.support.NameMatchMethodPointcutAdvisor

org.springframework.aop.support.RegexpMethodPointcutAdvisor

org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor

org.springframework.aop.support.DefaultPointcutAdvisor

在程序中可以使用以上四种PointcutAdvisor来配置切点和通知。

## org.springframework.aop.Pointcut接口

org.springframework.aop.Pointcut接口的类图如图5所示。

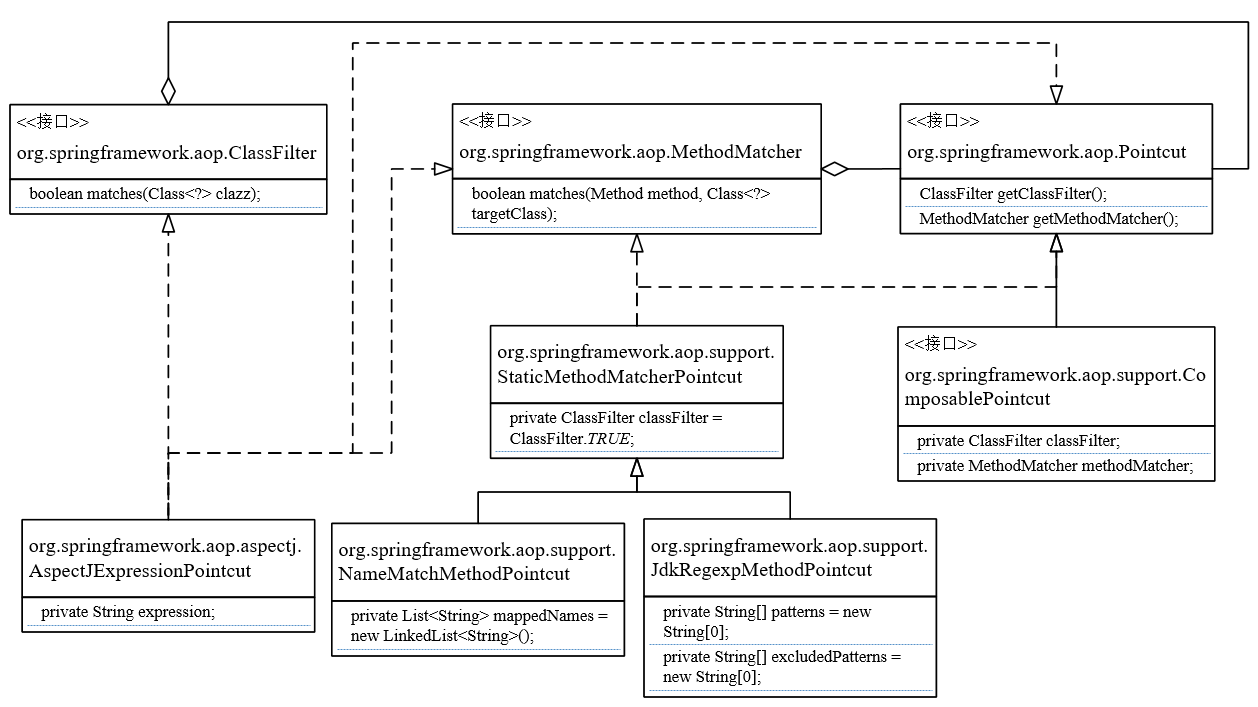


图5 org.springframework.aop.Pointcut接口的类图

org.springframework.aop.Pointcut接口有两个属性，ClassFilter和MethodMatcher，分别用来过滤类和方法。Spring针对Pointcut接口提供了三个实现类，分别是：

org.springframework.aop.support.NameMatchMethodPointcut

org.springframework.aop.support.JdkRegexpMethodPointcut

org.springframework.aop.aspectj.AspectJExpressionPointcut

其中NameMatchMethodPointcut类和JdkRegexpMethodPointcut类均继承自org.springframework.aop.aupport.StaticMethodMatcherPointcut类，该类实现了org.springframework.aop.MethodMatcher接口，并拥有一个值为ClassFilter.TRUE的ClassFilter属性，表示匹配任意类。AspectJExpressionPointcut类既继承了org.springframework.aop.aupport.StaticMethodMatcherPointcut类，又实现了org.springframework.aop.ClassFilter接口。

因为有可能ClassFilter和MethodMatcher并不止一个，因此Spring提供了ComposablePointcut接口来管理多个ClassFilter和MethodMatcher。

虽然ComposablePointcut只提供了一个classFilter属性和一个methodMatcher属性，但却可以管理多个ClassFilter和MethocMatcher，这是通过抽象类org.springframework.aop.support.ClassFilters和抽象类org.springframework.aop.support.MethodMatchers来实现的，以ClassFilters中静态内部类UnionClassFilter为例，其定义如图6所示。

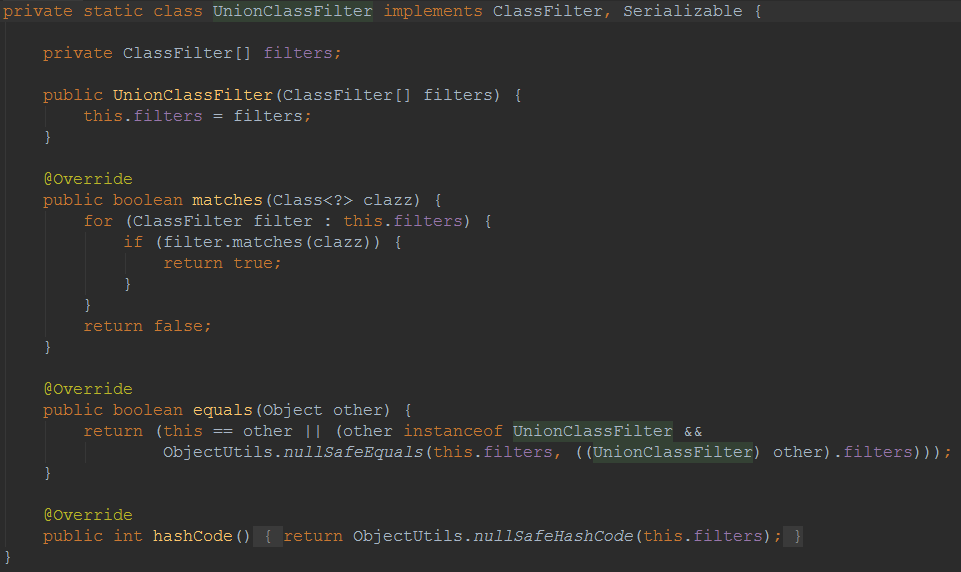


图6 org.springframework.aop.support.ClassFilters.UnionClassFilter定义

可以看到，UnionClassFilter实现了ClassFilter接口，并使用一个ClassFilter[]数组来维护多个ClassFilter。

# Jdk动态代理源码解析

整个Spring AOP可以分为两个大过程：

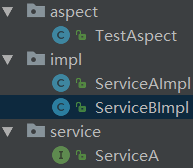
（1）根据xml文件或者注解中配置的拦截信息，生成相应的代理对象，这个代理对象中包含了相应的拦截器。

（2）执行所拦截的方法时，会调用代理对象的执行逻辑，完成各种拦截。

本节先对第二个过程进行源码解析，对一个过程先做一个简单的陈述：如果拦截的类的对应方法是接口方法，则使用动态代理创建代理对象，否则，则使用Cglib创建代理对象。

首先给出案例工程。

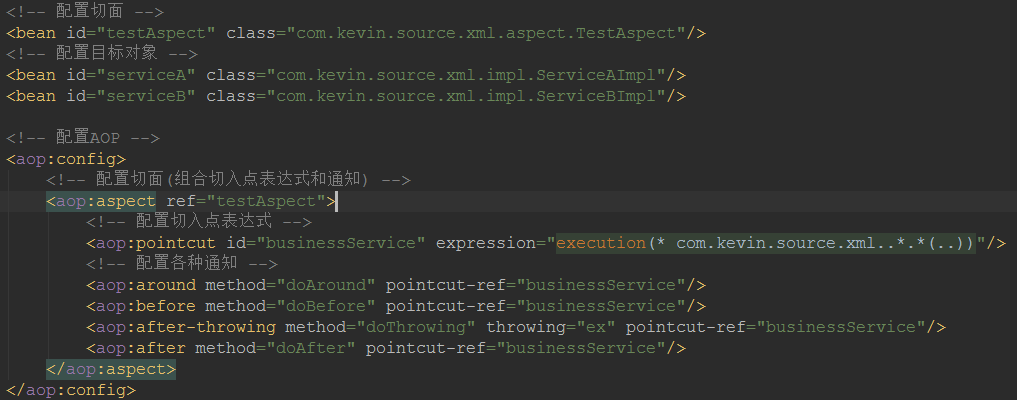
工程目录如下：



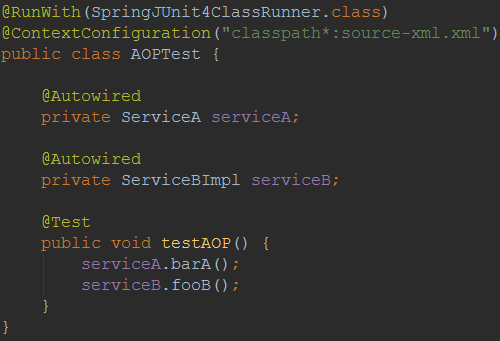
其中，切面TestAspect.java类如下：



xml配置如下；



测试用例如下；



测试结果如下：

log Begining method: com.kevin.source.xml.impl.ServiceAImpl.barA

ServiceAImpl.barA()

process time: 1 ms

log Ending method: com.kevin.source.xml.impl.ServiceAImpl.barA

log Begining method: com.kevin.source.xml.impl.ServiceBImpl.fooB

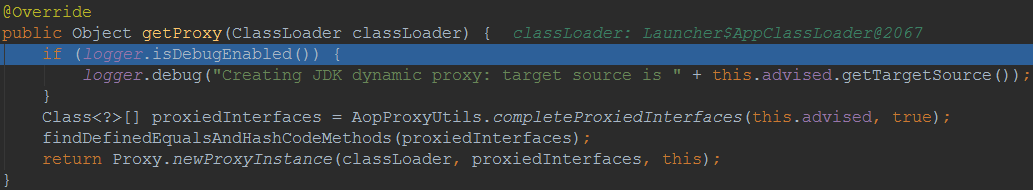
ServiceBImpl.fooB()

process time: 14 ms

log Ending method: com.kevin.source.xml.impl.ServiceBImpl.fooB

接下来分析以上过程。

调试测试用例时，我们可以看到，此时的ServiceA不再是它的实现类ServiceAImpl，而是一个代理对象。由于serviceA.barA()是接口方法，因此会使用动态代理创建代理对象，即通过JdkDynamicAopProxy类中的getProxy()方法获取代理对象，如下：



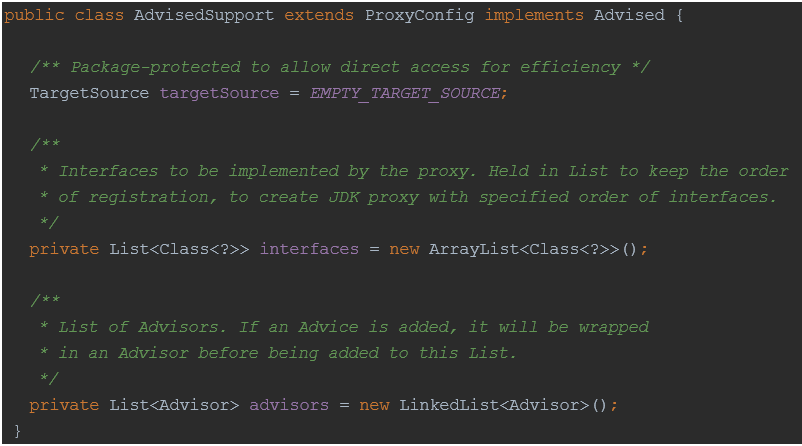
可以看到，在getProxy()方法中，最终是通过Proxy.newInstance()方法创建一个代理对象。由于JdkDynamicAopProxy类实现了InvocatioinHandler接口，所以在调用Proxy.newInstance()方法时，传入的调用处理器就是this。同时，这也意味着当执行代理对象的方法时，就会去执行JdkDynamicAopProxy类中的invoke()方法。

JdkDynamicAopProxy类中有一个成员属性：



这个成员属性advised中包含了一系列信息，例如目标对象、代理对象需要实现的接口、通知。

我们看一下AdvisedSupport类中的一些重要成员属性。



其中targetSource表示目标类对象，这里就是ServiceAImpl；interfaces表示代理对象需要实现的接口，这里就是ServiceA；advisors表示通知，包含了我们在xml配置文件中配置的通知信息，这里有五个通知：ExposeInvocatoinInterceptor和我们在xml配置文件中配置的四个通知，其中ExposeInvocationHandler是Spring AOP默认加入的通知，后面会讲。

另外，我们看到AdvisedSupport类继承了PorxyConfig类，并实现了Advised接口。Advised接口主要包含目标类对象targetSource、代理对象需要实现的接口interfaces和通知advisors；ProxyConfig类则包含了创建代理对象时所需要用到的配置信息，例如proxyTargetClass属性表示是否强制使用Cglib创建代理对象，exposeProxy属性表示是否暴露代理对象从而实现线程内共享该代理对象，以上两个属性分别对应xml配置文件中的如下配置：

<aop:config proxy-target-class="false" expose-proxy="false">  
</aop:config>

AdvisedSupport类中的一系列内容的创建都是为了调用JdkDynamicAopProxy类中的invoke()方法做准备，该方法如下：

@Override  
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 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.  
 return equals(args[0]);  
 }  
 else if (!this.hashCodeDefined && AopUtils.*isHashCodeMethod*(method)) {  
 // The target does not implement the hashCode() method itself.  
 return hashCode();  
 }  
 else if (method.getDeclaringClass() == DecoratingProxy.class) {  
 // There is only getDecoratedClass() declared -> dispatch to proxy config.  
 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, args);  
 }  
  
 Object retVal;

// 关注点1  
 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 target,  
 // in case it comes from a pool.  
 target = targetSource.getTarget();  
 if (target != null) {  
 targetClass = target.getClass();  
 }

// 关注点2  
 // Get the interception chain for this method.  
 List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
  
 // Check whether we have any advice. If we don't, we can fallback on direct  
 // reflective invocation of the target, and avoid creating a MethodInvocation.  
 if (chain.isEmpty()) { // 关注点3  
 // We can skip creating a MethodInvocation: just invoke the target directly  
 // Note that the final invoker must be an InvokerInterceptor so we know it does  
 // nothing but a reflective operation on the target, and no hot swapping or fancy proxying.  
 Object[] argsToUse = AopProxyUtils.*adaptArgumentsIfNecessary*(method, args);  
 retVal = AopUtils.*invokeJoinpointUsingReflection*(target, method, argsToUse);  
 }  
 else { // 关注点4  
 // We need to create a method invocation...  
 invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);  
 // 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.isPrimitive()) {  
 throw new AopInvocationException(  
 "Null return value from advice does not match primitive return type for: " + method);  
 }  
 return retVal;  
 }  
 finally {  
 if (target != null && !targetSource.isStatic()) {  
 // Must have come from TargetSource.  
 targetSource.releaseTarget(target);  
 }  
 if (setProxyContext) {  
 // Restore old proxy.  
 AopContext.*setCurrentProxy*(oldProxy);  
 }  
 }  
}

关注点1：this.advised.exposeProxy就是我们在xml文件中配置的<aop:config expose-proxy="false">。expose-proxy属性默认为false，如果配置为true，则意味着在该线程内将暴露代理对象，实现共享，即在该线程的任何地方都可以取到代理对象，以上过程是通过Threadlocal来实现的，我们可以看下AopContext类是如何实现的，如下：

public abstract class AopContext {  
  
 */\*\*  
 \* ThreadLocal holder for AOP proxy associated with this thread.  
 \* Will contain {****@code*** *null} unless the "exposeProxy" property on  
 \* the controlling proxy configuration has been set to "true".  
 \** ***@see*** *ProxyConfig#setExposeProxy  
 \*/* private static final ThreadLocal<Object> *currentProxy* = new NamedThreadLocal<Object>("Current AOP proxy");  
  
  
 */\*\*  
 \* Try to return the current AOP proxy. This method is usable only if the  
 \* calling method has been invoked via AOP, and the AOP framework has been set  
 \* to expose proxies. Otherwise, this method will throw an IllegalStateException.  
 \** ***@return*** *Object the current AOP proxy (never returns {****@code*** *null})  
 \** ***@throws*** *IllegalStateException if the proxy cannot be found, because the  
 \* method was invoked outside an AOP invocation context, or because the  
 \* AOP framework has not been configured to expose the proxy  
 \*/* public static Object currentProxy() throws IllegalStateException {  
 Object proxy = *currentProxy*.get();  
 if (proxy == null) {  
 throw new IllegalStateException(  
 "Cannot find current proxy: Set 'exposeProxy' property on Advised to 'true' to make it available.");  
 }  
 return proxy;  
 }  
  
 */\*\*  
 \* Make the given proxy available via the {****@code*** *currentProxy()} method.  
 \* <p>Note that the caller should be careful to keep the old value as appropriate.  
 \** ***@param*** *proxy the proxy to expose (or {****@code*** *null} to reset it)  
 \** ***@return*** *the old proxy, which may be {****@code*** *null} if none was bound  
 \** ***@see*** *#currentProxy()  
 \*/* static Object setCurrentProxy(Object proxy) {  
 Object old = *currentProxy*.get();  
 if (proxy != null) {  
 *currentProxy*.set(proxy);  
 }  
 else {  
 *currentProxy*.remove();  
 }  
 return old;  
 }  
  
}

可以看到，AopContext内部定义了一个ThreadLocal<Object> currentProxy属性来实现线程内共享代理对象，这样的话我们就可以在自定义的Advice上通过AopContext获取到当前的代理对象。

关注点2：找到与目标类和目标方法所对应的拦截器链，其内部是通过this.advised.advisorChainFactory来实现的，advisorChainFatory的默认实现类为DefaultAdvisorChainFactory，实现过程如下：

@Override  
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(  
 Advised config, Method method, Class<?> targetClass) {  
  
 // This is somewhat tricky... We have to process introductions first,  
 // but we need to preserve order in the ultimate list.  
 List<Object> interceptorList = new ArrayList<Object>(config.getAdvisors().length);  
 Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaringClass());  
 boolean hasIntroductions = *hasMatchingIntroductions*(config, actualClass);  
 AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.*getInstance*();  
  
 for (Advisor advisor : config.getAdvisors()) {  
 if (advisor instanceof PointcutAdvisor) {  
 // Add it conditionally.  
 PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;  
 if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {  
 MethodInterceptor[] interceptors = registry.getInterceptors(advisor);  
 MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();  
 if (MethodMatchers.*matches*(mm, method, actualClass, hasIntroductions)) {  
 if (mm.isRuntime()) {  
 // Creating a new object instance in the getInterceptors() method  
 // isn't a problem as we normally cache created chains.  
 for (MethodInterceptor interceptor : interceptors) {  
 interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm));  
 }  
 }  
 else {  
 interceptorList.addAll(Arrays.*asList*(interceptors));  
 }  
 }  
 }  
 }  
 else if (advisor instanceof IntroductionAdvisor) {  
 IntroductionAdvisor ia = (IntroductionAdvisor) advisor;  
 if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass)) {  
 Interceptor[] interceptors = registry.getInterceptors(advisor);  
 interceptorList.addAll(Arrays.*asList*(interceptors));  
 }  
 }  
 else {  
 Interceptor[] interceptors = registry.getInterceptors(advisor);  
 interceptorList.addAll(Arrays.*asList*(interceptors));  
 }  
 }  
  
 return interceptorList;  
}

上述过程对advised.getAdvisors()进行遍历，即对我们在xml文件中配置的各种通知进行遍历，然后分三种情况来获取对应的拦截器。

（1）当Advisor为PointcutAdvisor类型时，首先判断该PointcutAdvisor的ClassFilter是否拦截了targetClass，若拦截，则继续判断PointcutAdvisor的MethodMatcher是否拦截targetClass的method方法，如果也拦截，就将PointcutAdvisor中的Advice封装成MethodInterceptor，并将其添加到拦截器链中。在添加MethodInterceptor到拦截器链之前，还需要判断该PointcutAdvisor的MethodMatcher是否是动态变化的，如果是，则需要将该MethodInterceptor和MethodMatcher进一步包装为InterceptorAndDynamicMethodMatcher，然后将其添加到拦截器链中，如果不是，则直接添加MethodInterceptor到拦截器链中；

（2）当Advisor为IntroductionAdvisor时，因为IntroductionAdvisor是应用在类上的，因此只需要判断IntroductionAdvisor的ClassFiltr是否拦截targetClass，若拦截，则将IntroductionAdvisor中的Advice封装成MethodInterceptor，并将其添加到拦截器链中；

（3）对于其他类型的Advisor，直接将Advisor中的Advice封装成MethodInterceptor，并将其添加到拦截器链中。

上述三种情况中，都需要将Advisor中的Advice封装成MethodInterceptor并添加到拦截器链中，封装的过程是通过DefaultAdvisorChainFactory类的getInterceptors()方法完成的，如下；

@Override  
public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException {  
 List<MethodInterceptor> interceptors = new ArrayList<MethodInterceptor>(3);  
 Advice advice = advisor.getAdvice();  
 if (advice instanceof MethodInterceptor) {  
 interceptors.add((MethodInterceptor) advice);  
 }  
 for (AdvisorAdapter adapter : this.adapters) {  
 if (adapter.supportsAdvice(advice)) {  
 interceptors.add(adapter.getInterceptor(advisor));  
 }  
 }  
 if (interceptors.isEmpty()) {  
 throw new UnknownAdviceTypeException(advisor.getAdvice());  
 }  
 return interceptors.toArray(new MethodInterceptor[interceptors.size()]);  
}

可以看到，在将Advice封装成MethodInterceptor时，首先会判断该Advice是否已经实现了MethodInterceptor接口，如果已经实现的话，则直接将其强转为MethodInterceptor，AspectJAfterAdvice、AspectJAroundAdvice、AspectJAfterThrowingAdvice均实现了MethodInterceptor接口，因此直接强转；如果没有实现的话，则利用适配器模式将该Advice包装成MethodInterceptor，DefaultAdvisorChainFactory类硬编码了三种适配器，如下：

public DefaultAdvisorAdapterRegistry() {  
 registerAdvisorAdapter(new MethodBeforeAdviceAdapter());  
 registerAdvisorAdapter(new AfterReturningAdviceAdapter());  
 registerAdvisorAdapter(new ThrowsAdviceAdapter());  
}

AspectJBeforeAdvice、AspectJAfterReturingAdvice均没有实现MethodInterceptor接口，因此会被包装成MethodInterceptor。

关注点3：在获取拦截器链后，如果拦截器链为空，则通过反射执行目标类的目标方法，如下：

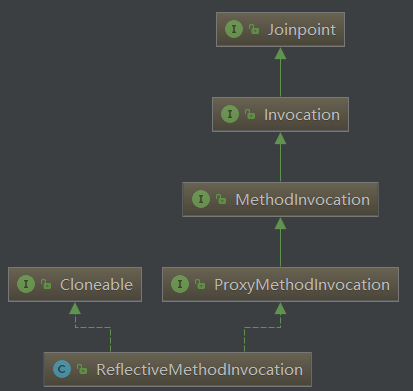
public static Object invokeJoinpointUsingReflection(Object target, Method method, Object[] args)  
 throws Throwable {  
  
 // Use reflection to invoke the method.  
 try {  
 ReflectionUtils.*makeAccessible*(method);  
 return method.invoke(target, args);  
 }

// 略

}

关注点4：在获取拦截器链后，如果拦截器链不为空，就创建一个ReflectiveMethodInvocation对象来执行拦截。

我们首先看一下ReflectiveMethodInvocation类的类图，如下：



其中Joinpoint类中的getThis()方法用来获取目标对象，proceed()方法用来执行拦截；Invocation类中的getArguments()方法用来获取方法参数；MethodInvocation类中的getMethod()方法用来获取方法；ProxyMethodInvocation类中的getPorxy()方法用来获取代理对象。因此，ReflectiveMethodInvocation类即拥有了如下属性：

public class ReflectiveMethodInvocation implements ProxyMethodInvocation, Cloneable {  
  
 protected final Object proxy;  
  
 protected final Object target;  
  
 protected final Method method;  
  
 protected Object[] arguments;  
  
 private final Class<?> targetClass;   
protected final List<?> interceptorsAndDynamicMethodMatchers;  
}

其中interceptorsAndDynamicMethodMatchers是前面获取到的拦截器链。

然后我们看一下ReflectiveMethodInvocation的proceed()方法，如下：

@Override  
public Object proceed() throws Throwable {  
 // We start with an index of -1 and increment early.  
 if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {  
 return invokeJoinpoint();  
 }  
  
 Object interceptorOrInterceptionAdvice =  
 this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);  
 if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {  
 // Evaluate dynamic method matcher here: static part will already have  
 // been evaluated and found to match.  
 InterceptorAndDynamicMethodMatcher dm =  
 (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;  
 if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {  
 return dm.interceptor.invoke(this);  
 }  
 else {  
 // Dynamic matching failed.  
 // Skip this interceptor and invoke the next in the chain.  
 return proceed();  
 }  
 }  
 else {  
 // It's an interceptor, so we just invoke it: The pointcut will have  
 // been evaluated statically before this object was constructed.  
 return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);  
 }  
}

首先看this.currentInterceptorIndex，它是ReflectiveMethodInvocation的一个成员属性，从-1开始，当其值为拦截器链中拦截器的个数减1时，表示拦截器链执行完毕，于是调用invokeJointpoint()方法执行目标对象的目标方法。

然后，从拦截器链中依次取出拦截器并执行拦截，这里会判断它是否是InterceptorAndDynamicMethodMatcher类实例。如果是，因为InterceptorAndDynamicMethodMatcher类中的MethodMatcher是可变的，所以在真正地执行拦截前，还要再判断一次该MethodMatcher是否拦截targetClass的method方法，如果是，则执行本次拦截，否则跳过本次拦截，继续执行下一个拦截；如果不是InterceptorAndDynamicMethodMatcher类实例，则直接执行拦截，由于此时的拦截器链中有5个MethodInterceptor，因此会拦截5次，我们依次看下这5次拦截是如何执行的。

第一个拦截器是ExposeInvocationInterceptor，这是Spring自动为我们添加的，其invoke()方法如下：

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 MethodInvocation oldInvocation = *invocation*.get();  
 *invocation*.set(mi);  
 try {  
 return mi.proceed();  
 }  
 finally {  
 *invocation*.set(oldInvocation);  
 }  
}

其中方法形参MethodInvocation mi始终是前面创建的ReflectiveMethodInvocation对象，invocation是ExposeInvocationInterceptor类的一个成员属性，使用了ThreadLocal将前面创建的ReflectiveMethodInvocation对象存放到当前线程中，实现线程内共享该ReflectiveMethodInvocation对象，这意味着我们可以在自定义的Advice中获取到该ReflectiveMethodInvocation对象。然后，执行mi.proceed()方法，即继续执行下一个拦截。

第二个拦截器是AspectJAfterAdvice，其invoke()方法如下：

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 return mi.proceed();  
 }  
 finally {  
 invokeAdviceMethod(getJoinPointMatch(), null, null);  
 }  
}

即先去执行后面的拦截器，等待其他拦截器都执行完毕后，再执行本次拦截。

第三个拦截器是AspectJAfterThrowingAdvice，其invoke()方法如下：

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 try {  
 return mi.proceed();  
 }  
 catch (Throwable ex) {  
 if (shouldInvokeOnThrowing(ex)) {  
 invokeAdviceMethod(getJoinPointMatch(), null, ex);  
 }  
 throw ex;  
 }  
}

即先去执行后面的拦截器，如果在执行后面的拦截器的过程中抛出了一场，则执行本次拦截。

第四个拦截器是AspectJAroundAdvice，其invoke()方法如下：

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 if (!(mi instanceof ProxyMethodInvocation)) {  
 throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: " + mi);  
 }  
 ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;  
 ProceedingJoinPoint pjp = lazyGetProceedingJoinPoint(pmi);  
 JoinPointMatch jpm = getJoinPointMatch(pmi);  
 return invokeAdviceMethod(pjp, jpm, null, null);  
}

其中lazyGetProceedingJoinPoint(pmi)方法会将ReflectiveMethodInvocation对象封装成ProceedingJoinPoint对象，这是一个十分重要的对象，我们看下在环绕通知中的内容：

public Object doAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  
 long start = System.*currentTimeMillis*();  
 Object result = proceedingJoinPoint.proceed();  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("process time: " + (end - start) + " ms");  
 return result;  
}

在环绕通知中，有个PorceedingJointPoint类型的形参proceedingJoinPoint，调用proceedingJoinPoint.proceed()方法本质上调用的仍然是ReflectiveMethodInvocation的proceed()方法，从而继续执行下一个拦截器，如果我们不调用proceedingJoinPoint.proceed()方法，则拦截器的执行就此终止，不会执行之后的拦截器。这里，我们在程序中调用了proceedingJoinPoint.proceed()方法，因此继续执行下一个拦截器。

最后一个拦截器是MethodBeforeInterceptor，其invoke()函数如下：

@Override  
public Object invoke(MethodInvocation mi) throws Throwable {  
 this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );  
 return mi.proceed();  
}

即先执行通知，然后继续执行下一个拦截器。

此时，所有拦截器执行完毕，于是执行目标对象的目标方法。目标方法执行完毕后，返回到上一个嵌套的proceed()方法，即执行最后一个拦截器MethodBeforeInterceptor的invoke()方法；执行完毕后，返回到上一个嵌套的proceed()方法，即执行上一个拦截器AspectJAroundAdvice的invoke()方法，也就是执行完毕我们编写的环绕通知；执行完毕后，返回到上一个嵌套的proceed()方法，即执行上一个拦截器AspectJAfterThrowingAdvice的invoke()方法，由于在这之后执行的拦截过程中均没有抛出异常，因此继续返回到上一个嵌套的proceed()方法，即执行上一个拦截器AspectJAfterAdvice的invoke()方法，此时会执行后置通知；执行完毕后，返回到上一个嵌套的proceed()方法，即执行上一个拦截器ExposeInvocationInterceptor的invoke()方法。至此，整个拦截过程结束。

# Cglib代理源码解析

Jdk动态代理是通过JdkDynamicAopProxy来生成代理对象的，而Cglib则是由CglibAopProxy来生成代理对象。

JdkDynamicAopProxy和CglibAopPorxy类均实现了AopProxy接口，如下：

public interface AopProxy {  
Object getProxy();  
Object getProxy(ClassLoader classLoader);  
  
}

同JdkDynamicAopProxy类一样，CglibAopProxy类也拥有一个protected final AdvisedSupport advised;属性，其中包含了目标类对象、代理对象需要实现的接口、通知和创建代理对象时所需要用到的配置信息，例如表示是否强制使用Cglib创建代理对象的proxyTargetClass属性，表示是否暴露代理对象从而实现线程内共享该代理对象的exposeProxy属性。这个属性在解析Jdk动态代理源码时已经详细说明，这里不再赘言。

我们看一下CglibAopProxy类中的getProxy()方法，如下：

@Override  
public Object getProxy(ClassLoader classLoader) {  
 if (*logger*.isDebugEnabled()) {  
 *logger*.debug("Creating CGLIB proxy: target source is " + this.advised.getTargetSource());  
 }  
  
 try {

// 此时的rootClass就是class com.kevin.source.xml.impl.ServiceBImpl  
 Class<?> rootClass = this.advised.getTargetClass();  
 Assert.*state*(rootClass != null, "Target class must be available for creating a CGLIB proxy");  
  
 Class<?> proxySuperClass = rootClass;

// 判断rootClass是否是Cglib产生的类（内部判断rootClass的className是否包含$$），对于本工程肯定不符合，跳过  
 if (ClassUtils.*isCglibProxyClass*(rootClass)) {  
 proxySuperClass = rootClass.getSuperclass();  
 Class<?>[] additionalInterfaces = rootClass.getInterfaces();  
 for (Class<?> additionalInterface : additionalInterfaces) {  
 this.advised.addInterface(additionalInterface);  
 }  
 }  
  
 // Validate the class, writing log messages as necessary.

// 验证proxySuperClass中是否有final方法，并对classLoader可见，如果有final方法或者对classLoader不可见，则打

// 印出警告信息，不做任何处理，即不会生成代理对象  
 validateClassIfNecessary(proxySuperClass, classLoader);  
  
 // Configure CGLIB Enhancer...

// 创建一个Enhancer类实例，并设置类加载器、要继承的类、要实现的接口、回调函数，最终调用enhancer.create()返回一个代

// 理对象  
 Enhancer enhancer = createEnhancer();  
 if (classLoader != null) {  
 enhancer.setClassLoader(classLoader);  
 if (classLoader instanceof SmartClassLoader &&  
 ((SmartClassLoader) classLoader).isClassReloadable(proxySuperClass)) {  
 enhancer.setUseCache(false);  
 }  
 }  
 enhancer.setSuperclass(proxySuperClass);  
 enhancer.setInterfaces(AopProxyUtils.*completeProxiedInterfaces*(this.advised));  
 enhancer.setNamingPolicy(SpringNamingPolicy.*INSTANCE*);  
 enhancer.setStrategy(new ClassLoaderAwareUndeclaredThrowableStrategy(classLoader));  
  
 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 call above  
 enhancer.setCallbackFilter(new ProxyCallbackFilter(  
 this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));  
 enhancer.setCallbackTypes(types);  
  
 // Generate the proxy class and create a proxy instance.  
 return createProxyClassAndInstance(enhancer, callbacks);  
 }

// 略

}

可以看到，CglibAopProxy类使用Enhancer类实例创建代理对象，具体而言就是：首先new一个Enhancer类实例，然后为其设置类加载器、要继承的父类、要实现的接口和回调函数，最后调用enhander.create()创建一个代理对象并返回即可。 在设置回调函数时，有一个重要的回调函数是DynamicAdvisedInterceptor，这个回调函数在执行通知时会用到，在其intercept()方法中实现了和Jdk动态代理类似的处理逻辑。

接下来我们看下拦截过程是如何实现的，在DynamicAdvisedInterceptor类的intercept()方法中，如下：

@Override  
public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  
 Object oldProxy = null;  
 boolean setProxyContext = false;  
 Class<?> targetClass = null;  
 Object target = null;  
 try {  
 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 target, in case it comes from a pool...  
 target = getTarget();  
 if (target != null) {  
 targetClass = target.getClass();  
 }  
 List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
 Object retVal;  
 // Check whether we only have one InvokerInterceptor: that is,  
 // no real advice, but just reflective invocation of the target.  
 if (chain.isEmpty() && Modifier.*isPublic*(method.getModifiers())) {  
 // We can skip creating a MethodInvocation: just invoke the target directly.  
 // Note that the final invoker must be an InvokerInterceptor, so we know  
 // it does nothing but a reflective operation on the target, and no hot  
 // swapping or fancy proxying.  
 Object[] argsToUse = AopProxyUtils.*adaptArgumentsIfNecessary*(method, args);  
 retVal = methodProxy.invoke(target, argsToUse);  
 }  
 else {  
 // We need to create a method invocation...  
 retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed();  
 }  
 retVal = *processReturnType*(proxy, target, method, retVal);  
 return retVal;  
 }  
 finally {  
 if (target != null) {  
 releaseTarget(target);  
 }  
 if (setProxyContext) {  
 // Restore old proxy.  
 AopContext.*setCurrentProxy*(oldProxy);  
 }  
 }  
}

这里的拦截过程和Jdk动态代理类一样，也是分为两步，第一步是获取拦截器链，第二步是创建一个MethodInvocation类实例来执行该拦截器链。

第一步：获取拦截器链。这个和Jdk动态代理获取拦截器链的步骤完全相同。

第二步：创建MethodInvocation类实例来执行拦截器链。Cglib创建的MethodInvocation是CglibMethodInvocation，该类继承了ReflectiveMethodInvocation类，并重写了invokeJoinpoint()方法。也就是说，执行拦截时，调用的依旧是MethodInvocation的父类ReflectiveMethodInvocation类中的proceed()方法，当所有拦截执行完毕后，调用目标对象的目标方法，这是通过invokeJoinpoint()方法来完成的，即此时调用的是MethodInvocation类中重写的invokeJoinpoint()方法。

ReflectiveMethodInvocation类中的invokeJoinpoint()方法如下：

protected Object invokeJoinpoint() throws Throwable {  
 return AopUtils.*invokeJoinpointUsingReflection*(this.target, this.method, this.arguments);  
}

可以看到，ReflectiveMethodInvocation一直使用反射执行目标方法。

CglibMethodInvocation类中重写的invokeJoinpoint()方法如下：

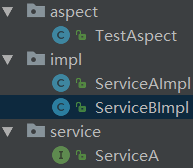
*/\*\*  
 \* Gives a marginal performance improvement versus using reflection to  
 \* invoke the target when invoking public methods.  
 \*/*@Override  
protected Object invokeJoinpoint() throws Throwable {  
 if (this.publicMethod) {  
 return this.methodProxy.invoke(this.target, this.arguments);  
 }  
 else {  
 return super.invokeJoinpoint();  
 }  
}

可以看到，CglibMethodInvocation会判断目标方法是否是public方法，如果不是，则使用反射执行目标方法，否则使用this.methodProxy.invoke(this.target, this.arguments)代理方法来执行，注释里面说到这样做会比反射性能更优。

# 解析xml生成代理对象源码解析

首先给出案例工程。

工程目录如下：



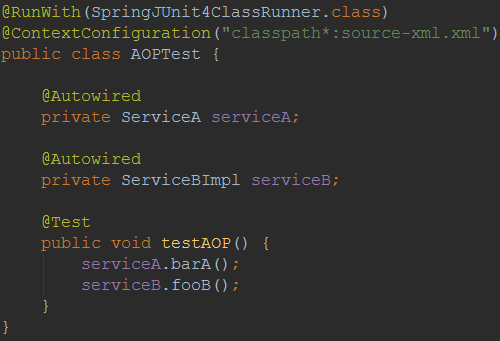
其中，切面TestAspect.java类如下：



xml配置如下；



测试用例如下；



测试结果如下：

log Begining method: com.kevin.source.aop.impl.ServiceAImpl.barA

ServiceAImpl.barA()

ServiceBImpl.fooB()

process time: 44 ms

log Ending method: com.kevin.source.aop.impl.ServiceBImpl.fooB

接下来分析Spring是如何解析该xml文件并生成相应的代理对象的。

Spring解析xml文件并最终生成代理对象分为三步：

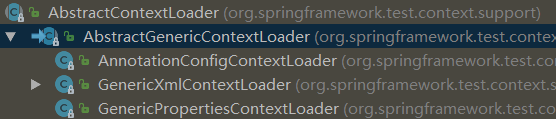
（1）解析xml文件并将在xml文件中配置的bean注册到bean工厂中；

（2）实例化注册到bean工厂中的bean；

（3）创建动态代理。

首先是解析xml文件，并将在xml文件中配置的bean注册到bean工厂中。

调试代码可知，Spring会使用AbstractGenericContextLoader的loadContext()方法加载上下文，AbstractGenericContextLoader类的类图如下：



在loadContext()方法中，Spring会接着调用loadBeanDefinitions()方法来解析xml文件，并将在xml文件中配置的bean注册到bean工厂中。

对于在xml文件中配置的bean，Spring会调用GenericXmlContextLoader. createBeanDefinitionReader(GenericApplicationContext context)方法创建一个XmlBeanDefinitionReader类实例，用来解析xml文件并注册bean。XmlBeanDefinitoin类中有一个成员属性private final BeanDefinitionRegistry registry，属性值就是前面创建XmlBeanDefinitionReader类实例时传递进来的GenericApplicationContext context（注：GenericApplicationContext类实现了BeanDefinitionRegistry接口），该属性的作用是将在xml文件中配置的bean注册到bean工厂中。

在注册bean时，对于<bean>标签，直接用registry将对应的bean注册到bean工厂中即可；对于使用<aop-config>标签，则会先注册几个BeanDifinitionParser，然后由相应的BeanDefinitionParser来解析aop配置，最终还是使用registry将解析得到的advice、pointcut等bean注册到bean工厂中。

下面细讲一下Spring是如何注册bean的。

XmlBeanDefinitionReader类实例在解析xml文件并注册bean时，会创建一个DefaultBeanDefinitionDocumentReader类实例，然后调用该类实例的parseBeanDefinitions()方法完成bean的注册，该方法如下：

protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  
 if (delegate.isDefaultNamespace(root)) {  
 NodeList nl = root.getChildNodes();  
 for (int i = 0; i < nl.getLength(); i++) {  
 Node node = nl.item(i);  
 if (node instanceof Element) {  
 Element ele = (Element) node;  
 if (delegate.isDefaultNamespace(ele)) {

// 对于<bean>标签，直接使用registry将其注册到bean工厂中  
 parseDefaultElement(ele, delegate);  
 }  
 else {

// 对于<aop:config>标签，由相应的BeanDefinitionParser来解析aop配置，最终还是使用registry将解析得到的

// advice、pointcut等bean注册到bean工厂中  
 delegate.parseCustomElement(ele);  
 }  
 }  
 }  
 }  
 else {  
 delegate.parseCustomElement(root);  
 }  
}

对于<bean>标签，由于delegate.isDefaultNamespace()为true，因此执行delegate.parseDefaultElement()方法，会直接使用registry将bean注册到bean工厂中；对于非<bean>标签，由于delegate. isDefaultNamespace()为false，因此执行delegate.parseCustomElement()方法。

对于<aop:config>标签，Spring会使用AopNamespaceHandler类的init()方法先注册几个BeanDefinitionParser，如下：

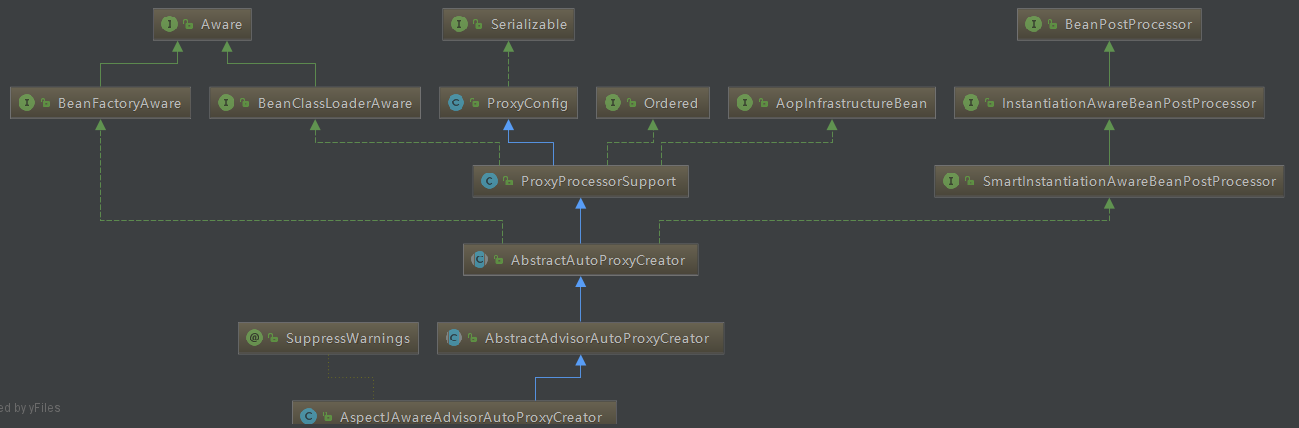
@Override  
public void init() {  
 // In 2.0 XSD as well as in 2.1 XSD.  
 registerBeanDefinitionParser("config", new ConfigBeanDefinitionParser());  
 registerBeanDefinitionParser("aspectj-autoproxy", new AspectJAutoProxyBeanDefinitionParser());  
 registerBeanDefinitionDecorator("scoped-proxy", new ScopedProxyBeanDefinitionDecorator());  
  
 // Only in 2.0 XSD: moved to context namespace as of 2.1  
 registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());  
}

可以看到，Spring会使用ConfigBeanDefinitionParser类来解析<aop:config>标签，使用AspectJAutoProxyBeanDefinitionParser类来解析<aop:aspectj-autoproxy>标签。

我们看下ConfigBeanDefinitionParser是如何解析<aop:config>标签的。

@Override  
public BeanDefinition parse(Element element, ParserContext parserContext) {  
 CompositeComponentDefinition compositeDef =  
 new CompositeComponentDefinition(element.getTagName(), parserContext.extractSource(element));  
 parserContext.pushContainingComponent(compositeDef);  
  
 configureAutoProxyCreator(parserContext, element);  
  
 List<Element> childElts = DomUtils.*getChildElements*(element);  
 for (Element elt: childElts) {  
 String localName = parserContext.getDelegate().getLocalName(elt);  
 if (*POINTCUT*.equals(localName)) {  
 parsePointcut(elt, parserContext);  
 }  
 else if (*ADVISOR*.equals(localName)) {  
 parseAdvisor(elt, parserContext);  
 }  
 else if (*ASPECT*.equals(localName)) {  
 parseAspect(elt, parserContext);  
 }  
 }  
  
 parserContext.popAndRegisterContainingComponent();  
 return null;  
}

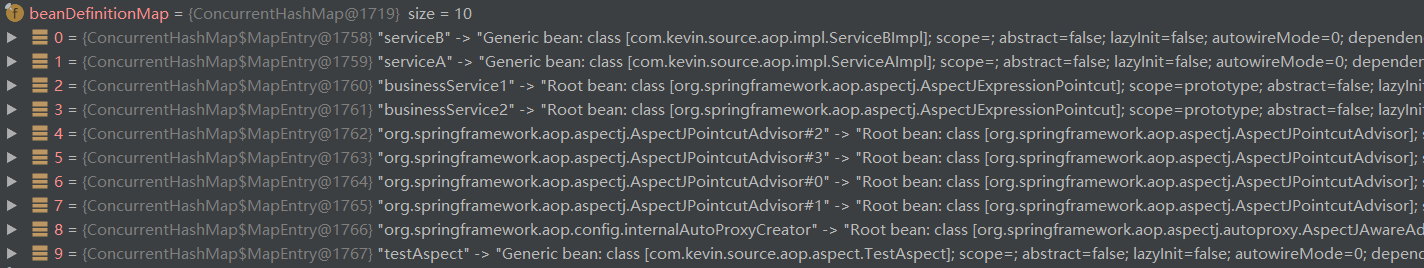
可以看到，ConfigBeanDefinitionParser首先会通过调用configureAutoProxyCreator()方法注册一个AspectJAwareAdvisorAutoProxyCreator，并将在xml文件中配置的proxy-target-class特性和expose-proxy特性到它的属性中，AspectJAwareAdvisorAutoProxyCreator的类图如下：



AspectJAwareAdvisorAutoProxyCreator实现了ProxyConfig类，因此包含了代理对象的一些配置信息，例如是否强制使用Cglib创建代理对象的proxyTargetClass属性，是否在线程内共享代理对象的exposeProxy属性；同时，它还实现了BeanPostProcessor后置处理器，在接下来的分析中，我们会看到，Spring会调用AspectJAwareAdvisorAutoProxyCreator类（此时的作用是后置处理器）中的craeteProxy()方法完成代理对象的创建。

然后，ConfigBeanDefinitionParser会将pointcut、advisor、aspect等信息封装成一个BeanDefinition类实例，并将其注册到bean工厂中。BeanDefinition类中有一个beanClass属性，表示该bean对应的Class。就本工程案例而言，对于封装了pointcut的BeanDefinition类实例，其beanClass属性值为AspectJExpressionPointcut.class；对于封装了aspect的BeanDefinition类实例，其beanClass属性值为AspectJPointcutAdvisor.class。

至此，Spring解析完毕xml文件，并将在xml文件中配置的bean注册到bean工厂中。此时bean工厂中的所有bean如下：



可以看到，我们在xml文件中配置的两个目标对象serviceA、serviceB，切面testAspect，两个切入点表达式businessService1、businessService2，四个Advice，以及ConfigBeanDefinitionParser创建的一个AspectJAwareAdvisorAutoProxyCreator，均被注入到bean工厂中。

然后是实例化注册到bean工厂中的bean。

Spring通过在AbstractGenericContextLoader类的loadContext()方法中调用context.refresh()方法来完成bean的实例化，不过在实例化bean后，会调用AspectJAwareAdvisorAutoProxyCreator后置处理器，即执行以下操作：

（1）调用后置处理器的postProcessBeforeInitialization()方法

（2）初始化bean

（3）调用后置处理器的postProcessAfterInitialization()方法

在调用后置处理器的postProcessBeforeInitialization()方法时，会通过后置处理器的findCandidateAdvisors()找到所有可使用的Advisor，即：List<Advisor> candidateAdvisors。在本案例工程中，可使用的Advisor有4个，即我们在xml文件中配置的4个Advice，Spring会从bean工厂中取出这4个Advice对应的Advisor。

在调用后置处理器的postProcessAfterInitialization()方法时，会判断该bean是否需要被代理，如果需要，则创建代理对象，否则返回该bean。判断bean是否需要被代理是在AspectJAwareAdvisorAutoProxyCreator类的父类AbstractAutoProxyCreator的wrapIfNecessary()中完成的，如下：

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(), beanName)) {  
 this.advisedBeans.put(cacheKey, Boolean.*FALSE*);  
 return bean;  
 }  
  
 // Create proxy if we have advice.  
 Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);  
 if (specificInterceptors != *DO\_NOT\_PROXY*) {  
 this.advisedBeans.put(cacheKey, Boolean.*TRUE*);  
 Object proxy = createProxy(  
 bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));  
 this.proxyTypes.put(cacheKey, proxy.getClass());  
 return proxy;  
 }  
  
 this.advisedBeans.put(cacheKey, Boolean.*FALSE*);  
 return bean;  
}

可以看到，在wrapIfNecessary()方法中，AbstractAutoProxyCreator会继续调用getAdvicesAndAdvisorsForBean()方法获取可以应用到该bean上的所有拦截器，即通过各个Advisor中的Pointcut的ClassFilter和MethodMatcher来判断该bean是否需要被织入通知。

最后是创建动态代理。

在实例化bena之后，Spring会调用后置处理器AspectJAwareAdvisorAutoProxyCreator对实例化的bean进行后续处理。这里会首先判断该bean是否需要被代理，即通通过各个Advisor中的Pointcut的ClassFilter和MethodMatcher来判断该bean是否需要被织入通知，如果不需要，则直接返回该bean；如果需要，则为其创建代理对象。创建代理对象是通过createProxy()方法来完成的，如下：

protected Object createProxy(  
 Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {  
  
 if (this.beanFactory instanceof ConfigurableListableBeanFactory) {  
 AutoProxyUtils.*exposeTargetClass*((ConfigurableListableBeanFactory) this.beanFactory, beanName, beanClass);  
 }  
  
 ProxyFactory proxyFactory = new ProxyFactory();

// 关注点1  
 proxyFactory.copyFrom(this);

// 关注点2  
 if (!proxyFactory.isProxyTargetClass()) {  
 if (shouldProxyTargetClass(beanClass, beanName)) {  
 proxyFactory.setProxyTargetClass(true);  
 }  
 else {  
 evaluateProxyInterfaces(beanClass, proxyFactory);  
 }  
 }

// 关注点3  
 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);  
 }

// 关注点4  
 return proxyFactory.getProxy(getProxyClassLoader());  
}

关注点1：将ProxyConfig信息复制到ProxyFactory中（注：ProxyFactory类也继承了ProxyConfig类），因此ProxyFactory就可以获得代理对象的一些配置信息，例如是否强制使用Cglib创建代理的proxyTargetClass属性，是否在线程内共享代理对象的exposeProxy属性。

关注点2：判断proxyTargetClass属性是否为faslse，如果为false，则还需要判断是否应该将其设为true，若目标类没有实现任何接口，则仍然会将proxyTargetClass属性设为true。

关注点3：将xml文件中配置的各种通知封装为Advisor，这是通过buildAdvisors()方法完成的，在该方法中执行了如下代码：

this.advisorAdapterRegistry.wrap(allInterceptors.get(i));

我们看下wrap()函数，如下：

@Override  
public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {  
 if (adviceObject instanceof Advisor) {  
 return (Advisor) adviceObject;  
 }  
 if (!(adviceObject instanceof Advice)) {  
 throw new UnknownAdviceTypeException(adviceObject);  
 }  
 Advice advice = (Advice) adviceObject;  
 if (advice instanceof MethodInterceptor) {  
 // So well-known it doesn't even need an adapter.  
 return new DefaultPointcutAdvisor(advice);  
 }  
 for (AdvisorAdapter adapter : this.adapters) {  
 // Check that it is supported.  
 if (adapter.supportsAdvice(advice)) {  
 return new DefaultPointcutAdvisor(advice);  
 }  
 }  
 throw new UnknownAdviceTypeException(advice);  
}

可以看到，如果是Advisor则直接返回，如果是Advice，则通过MethodInterceptor或者AdvisorAdaptor将其包装为DefaultPointcutAdvisor。

关注点4：使用DefaultAopProxyFactory来创建代理对象，这是通过DefaultAopProxyFactory.createProxy()方法完成的，如下：

@Override  
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  
 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.");  
 }  
 if (targetClass.isInterface() || Proxy.*isProxyClass*(targetClass)) {  
 return new JdkDynamicAopProxy(config);  
 }  
 return new ObjenesisCglibAopProxy(config);  
 }  
 else {  
 return new JdkDynamicAopProxy(config);  
 }  
}

可以看到，这里会决定是采用Jdk动态代理创建代理对象，还是采用Cglib创建代理对象。

如果满足任意以下条件且目标类不是接口，则采用Cglib创建代理对象，否则采用Jdk动态代理创建代理对象：

（1）config.isOptimize()为true，即如果优化，则采用Cglib代理。默认为false；

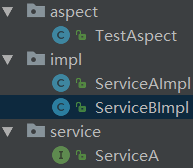
（2）config.isProxyTargetClass()为true，即如果proxyTargetClass属性为true，则强制使用Cglib代理；

（3）hasNoUserSuppliedProxyInterfaces(config)为true，即如果目标类没有实现任何接口，或者仅实现了SpringProxy接口，则使用Cglib代理。

# 解析ApsectJ注解生成代理对象源码解析

首先给出案例工程。

工程目录如下：



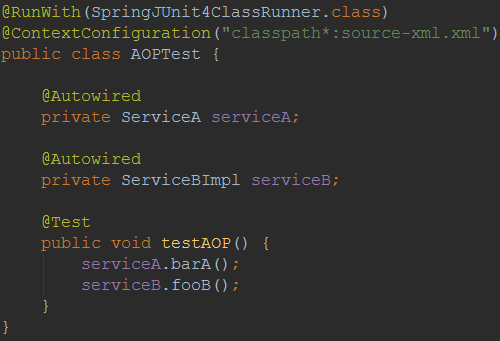
其中，切面TestAspect.java类如下：

@Component  
@Aspect  
public class TestAspect {  
  
 @Pointcut("execution(\* com.kevin.source.annotation..\*.bar\*(..))")  
 public void businessService1() {}  
  
 @Pointcut("execution(\* com.kevin.source.annotation..\*.foo\*(..))")  
 public void businessService2() {}  
  
 @Around("businessService2()")  
 public Object doAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  
 long start = System.*currentTimeMillis*();  
 Object result = proceedingJoinPoint.proceed();  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("process time: " + (end - start) + " ms");  
 return result;  
 }  
  
 @Before("businessService1()")  
 public void doBefore(JoinPoint joinpoint) {  
 System.*out*.println("log Begining method: "  
 + joinpoint.getTarget().getClass().getName() + "."  
 + joinpoint.getSignature().getName());  
 }  
  
 @AfterThrowing(value = "businessService1()", throwing = "ex")  
 public void doThrowing(JoinPoint joinpoint, Throwable ex) {  
 System.*out*.println("method " + joinpoint.getTarget().getClass().getName()  
 + "." + joinpoint.getSignature().getName() + " throw exception");  
 System.*out*.println(ex.getMessage());  
 }  
  
 @After("businessService2()")  
 public void doAfter(JoinPoint joinPoint) {  
 System.*out*.println("log Ending method: "  
 + joinPoint.getTarget().getClass().getName() + "."  
 + joinPoint.getSignature().getName());  
 }  
}

xml配置如下；

<context:component-scan base-package="com.kevin.source.annotation"/>  
  
<aop:aspectj-autoproxy expose-proxy="false" proxy-target-class="false"></aop:aspectj-autoproxy>

测试用例如下；



测试结果如下：

log Begining method: com.kevin.source.aop.impl.ServiceAImpl.barA

ServiceAImpl.barA()

ServiceBImpl.fooB()

process time: 44 ms

log Ending method: com.kevin.source.aop.impl.ServiceBImpl.fooB

接下来分析Spring是如何解析AspectJ注解并生成相应的代理对象的。

Spring解析AspectJ注解并生成相应的代理对象也分为三步：

（1）解析xml文件并将在xml文件中配置的bean注册到bean工厂中；

（2）实例化注册到bean工厂中的bean；

（3）创建动态代理。

首先是解析xml文件，并将在xml文件中配置的bean注册到bean工厂中。这一过程同前一章类似，也是创建一个XmlBeanDefinitionReaader类实例，用来读取xml配置文件，并将在xml文件中配置的bean注册到bean工厂中。对于<bean>标签，会直接使用registry将bean注册到bean工厂中；对于非<bean>标签，会使用相应的BeanDefinitionParser来完成bean的注册。这里细讲一下Spring对于xml文件中的<context:component-scan>标签和<aop:aspecj-autoproxy>标签的解析。

对于xml文件中的<context:component-scan>标签，Spring会使用ContextNamespaceHandler类的init()方法先注册几个BeanDefinitionParser，如下：

@Override  
public void init() {  
 registerBeanDefinitionParser("property-placeholder", new PropertyPlaceholderBeanDefinitionParser());  
 registerBeanDefinitionParser("property-override", new PropertyOverrideBeanDefinitionParser());  
 registerBeanDefinitionParser("annotation-config", new AnnotationConfigBeanDefinitionParser());  
 registerBeanDefinitionParser("component-scan", new ComponentScanBeanDefinitionParser());  
 registerBeanDefinitionParser("load-time-weaver", new LoadTimeWeaverBeanDefinitionParser());  
 registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());  
 registerBeanDefinitionParser("mbean-export", new MBeanExportBeanDefinitionParser());  
 registerBeanDefinitionParser("mbean-server", new MBeanServerBeanDefinitionParser());  
}

可以看到，Spring会使用ComponentScanBeanDefinitionParser类来解析<context:component-scan>标签，ComponentScanBeanDefinitionParser会把扫描到的bean注册到bean工厂中。

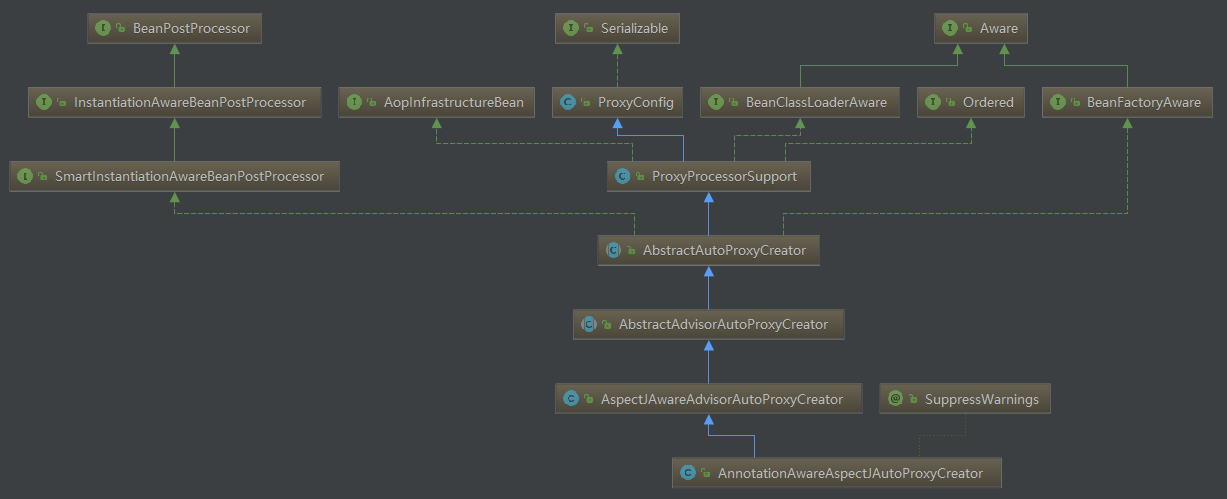
对于xml文件中的<aop:aspectj-autoproxy>标签，Spring会使用AopNameHandler的init()方法先注册几个BeanDefinitionParser，如下：

@Override  
public void init() {  
 // In 2.0 XSD as well as in 2.1 XSD.  
 registerBeanDefinitionParser("config", new ConfigBeanDefinitionParser());  
 registerBeanDefinitionParser("aspectj-autoproxy", new AspectJAutoProxyBeanDefinitionParser());  
 registerBeanDefinitionDecorator("scoped-proxy", new ScopedProxyBeanDefinitionDecorator());  
  
 // Only in 2.0 XSD: moved to context namespace as of 2.1  
 registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());  
}

可以看到，Spring会使用AspectJAutoProxyBeanDefinitionParser类来解析<aop:aspectj-autoproxy>标签，我们看下是如何解析的，如下：

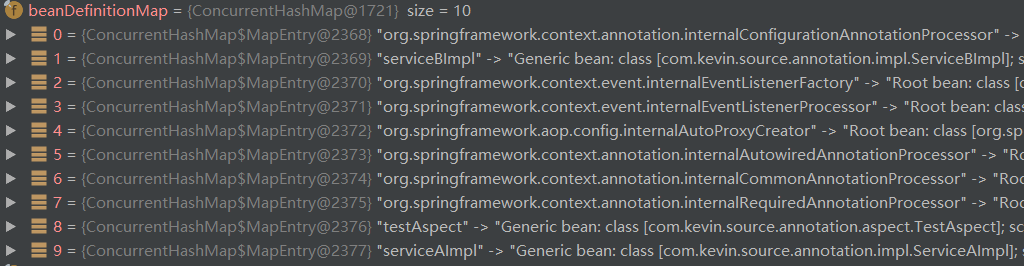
@Override  
public BeanDefinition parse(Element element, ParserContext parserContext) {  
 AopNamespaceUtils.*registerAspectJAnnotationAutoProxyCreatorIfNecessary*(parserContext, element);  
 extendBeanDefinition(element, parserContext);  
 return null;  
}

AspectJAutoProxyBeanDefinitionParser会调用registerAspectJAnnotationAutoProxyCreatorIfNecessary()方法注册一个AnnotationAwareAspectJAutoProxyCreator，并将在xml文件中配置的proxy-target-class特性和expose-proxy特性到它的属性中，AnnotationAwareAspectJAutoProxyCreator的类图如下：



AnnotationAwareAspectJAutoProxyCreator实现了ProxyConfig类，因此包含了代理对象的一些配置信息，例如是否强制使用Cglib创建代理对象的proxyTargetClass属性，是否在线程内共享代理对象的exposeProxy属性；同时，它还实现了BeanPostProcessor后置处理器，在接下来的分析中，我们会看到，Spring会调用AspectJAwareAdvisorAutoProxyCreator类（此时的作用是后置处理器）中的craeteProxy()方法完成代理对象的创建。

至此，Spring解析完毕xml文件，并将在xml文件中配置的bean注册到bean工厂中。此时bean工厂中的所有bean如下：



可以看到，Spring会扫描出我们编写的三个bean，分别是：serviceAImpl、serviceBImpl、testAspect。

然后是实例化注册到bean工厂中的bean。这个过程同前一章类似，Spring也是通过在AbstractGenericContextLoader类的loadContext()方法中调用context.refresh()方法来完成bean的实例化，不过在实例化bean后，调用的是AnnotationAwareAspectJAutoProxyCreator后置处理器，即执行以下操作：

（1）调用后置处理器的postProcessBeforeInitialization()方法

（2）初始化bean

（3）调用后置处理器的postProcessAfterInitialization()方法

在调用后置处理器的postProcessBeforeInitialization()方法时，会通过后置处理器的findCandidateAdvisors()找到所有可使用的Advisor，即：List<Advisor> candidateAdvisors，并将其缓存起来。在本案例工程中，此时可使用的Advisor为4个，因为我们在切面TestApect中编写了4个Advice，Spring会为这些Advice分别创建对应的InstantiationModelAwarePointcutAdvisorImpl类实例，InstantiationModelAwarePointcutAdvisorImpl类实现了PointcutAdvisor接口。

在调用后置处理器的postProcessAfterInitialization()方法时，会判断该bean是否需要被代理，如果需要，则创建代理对象，否则返回该bean。判断bean是否需要被代理的过程与前一章相同。

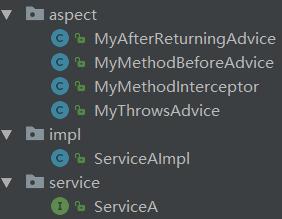
最后是创建动态代理。这一过程与前一章相同。

# ProxyFactoryBean生成代理对象源码解析

前面所有章节讲的都是Spring引入AspectJ的AOP，本章分析Spring自己实现的AOP。

Spring自己的AOP是通过ProxyFactoryBean来完成的，首先给出案例工程。

工程目录如下：



其中，各种通知如下：

public class MyMethodBeforeAdvice implements MethodBeforeAdvice {  
 @Override  
 public void before(Method method, Object[] args, Object target) throws Throwable {  
 System.*out*.println("log Begining method: "  
 + target.getClass().getName() + "."  
 + method.getName());  
 }  
}

public class MyAfterReturningAdvice implements AfterReturningAdvice {  
 @Override  
 public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {  
 System.*out*.println("log After Returning method: "  
 + target.getClass().getName() + "."  
 + method.getName());  
 }  
}

public class MyThrowsAdvice implements ThrowsAdvice {  
  
 public void afterThrowing(Method method, Object[] args, Object target, Exception ex) {  
 System.*out*.println("method " + target.getClass().getName()  
 + "." + method.getName() + " throw exception");  
 System.*out*.println(ex.getMessage());  
 }  
}

public class MyMethodInterceptor implements MethodInterceptor {  
 @Override  
 public Object invoke(MethodInvocation invocation) throws Throwable {  
 long start = System.*currentTimeMillis*();  
 Object result = invocation.proceed();  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("process time: " + (end - start) + " ms");  
 return result;  
 }  
}

xml配置如下；

<!-- 配置目标对象 -->  
<bean id="serviceA" class="com.kevin.source.spring.impl.ServiceAImpl"/>  
  
<!-- 配置前置通知 -->  
<bean id="myMethodBeforeAdvice" class="com.kevin.source.spring.aspect.MyMethodBeforeAdvice"/>  
  
<!-- 配置返回通知 -->  
<bean id="myAfterReturningAdvice" class="com.kevin.source.spring.aspect.MyAfterReturningAdvice"/>  
  
<!-- 配置异常通知 -->  
<bean id="myThrowsAdvice" class="com.kevin.source.spring.aspect.MyThrowsAdvice"/>  
  
<!-- 配置环绕通知 -->  
<bean id="myMethodInterceptor" class="com.kevin.source.spring.aspect.MyMethodInterceptor"/>  
  
<!-- 使用NameMatchMethodPointcutAdvisor配置返回通知的切入点表达式和通知 -->  
<bean id="nameMatchMethodPointcutAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  
 <property name="advice" ref="myAfterReturningAdvice"/>  
 <property name="mappedNames">  
 <list>  
 <value>barA</value>  
 </list>  
 </property>  
</bean>  
  
<!--使用RegexpMethodPointcutAdvisor配置异常通知的切入点表达式和通知 -->  
<bean id="regexpMethodPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  
 <property name="advice" ref="myThrowsAdvice"/>  
 <property name="pattern" value=".\*A"/>  
</bean>  
  
<!-- 使用AspectJExpressionPointcutAdvisor配置前置通知的切入点表达式和通知 -->  
<bean id="aspectJExpressionPointcutAdvisor" class="org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor">  
 <property name="advice" ref="myMethodInterceptor"/>  
 <property name="expression" value="execution(\* barA(..))"/>  
</bean>  
  
<!-- 配置ProxyFactoryBean -->  
<bean id="proxyFactoryBean" class="org.springframework.aop.framework.ProxyFactoryBean">  
 <!-- 配置代理接口集 -->  
 <property name="proxyInterfaces">  
 <list>  
 <value>com.kevin.source.spring.service.ServiceA</value>  
 </list>  
 </property>  
 <!-- 配置切面 -->  
 <property name="interceptorNames">  
 <list>  
 <value>myMethodBeforeAdvice</value>  
 <value>nameMatchMethodPointcutAdvisor</value>  
 <value>regexpMethodPointcutAdvisor</value>  
 <value>aspectJExpressionPointcutAdvisor</value>  
 </list>  
 </property>  
 <!-- 配置目标对象 -->  
 <property name="targetName" value="serviceA"/>  
</bean>

测试用例如下；

@Test  
public void testAop() {  
 ApplicationContext context = new ClassPathXmlApplicationContext("source-spring.xml");  
 ServiceA serviceA = (ServiceA) context.getBean("proxyFactoryBean");  
 serviceA.barA();  
}

测试结果如下：

log Begining method: com.kevin.source.spring.impl.ServiceAImpl.barA

ServiceAImpl.barA()

process time: 0 ms

log After Returning method: com.kevin.source.spring.impl.ServiceAImpl.barA

接下来分析Spring是如何通过ProxyFactoryBean来生成代理对象的。

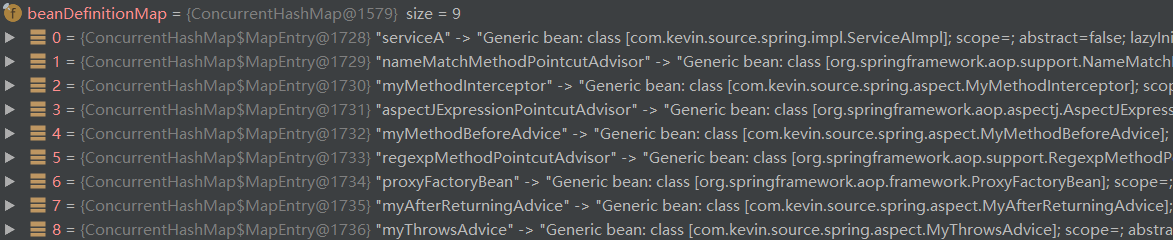
Spring通过ProxyFactoryBean生成代理对象也分为三步：

（1）解析xml文件并将在xml文件中配置的bean注册到bean工厂中；

（2）实例化注册到bean工厂中的bean；

（3）创建动态代理。

首先是解析xml文件，并将在xml文件中配置的bean注册到bean工厂中。这一过程同前一章类似，也是创建一个XmlBeanDefinitionReaader类实例，用来读取xml配置文件，并将在xml文件中配置的bean注册到bean工厂中。对于<bean>标签，会直接使用registry将bean注册到bean工厂中；对于非<bean>标签，会使用相应的BeanDefinitionParser来完成bean的注册。在本案例工程中，由于xml文件中都是通过<bean>标签配置bean，因此会直接使用registry将bean注册到bean工厂中，最终，bean工厂中的所有bean如下：



可以看到，Spring会扫描出我们编写的三个bean，分别是：serviceAImpl、serviceBImpl、testAspect。

然后是实例化注册到bean工厂中的bean。由于此前没有注册任何后置处理器，因此这里就是直接实例化bean工厂中的所有bean。

最后是创建动态代理。在程序中通过context.getBean("proxyFactoryBean")获取代理对象时，实际上是调用ProxyFactoryBean.getObject()方法获取代理对象，该方法如下：

@Override  
public Object getObject() throws BeansException {

// 关注点1  
 initializeAdvisorChain();

// 关注点2  
 if (isSingleton()) {  
 return getSingletonInstance();  
 }  
 else {  
 if (this.targetName == null) {  
 logger.warn("Using non-singleton proxies with singleton targets is often undesirable. " +  
 "Enable prototype proxies by setting the 'targetName' property.");  
 }  
 return newPrototypeInstance();  
 }  
}

关注点1：根据我们配置的interceptorNames来获取拦截器链。实际上还是调用this.advisorAdapterRegistry.wrap()方法将xml文件中配置的Advice包装为Advisor，如下：

@Override  
public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {  
 if (adviceObject instanceof Advisor) {  
 return (Advisor) adviceObject;  
 }  
 if (!(adviceObject instanceof Advice)) {  
 throw new UnknownAdviceTypeException(adviceObject);  
 }  
 Advice advice = (Advice) adviceObject;  
 if (advice instanceof MethodInterceptor) {  
 // So well-known it doesn't even need an adapter.  
 return new DefaultPointcutAdvisor(advice);  
 }  
 for (AdvisorAdapter adapter : this.adapters) {  
 // Check that it is supported.  
 if (adapter.supportsAdvice(advice)) {  
 return new DefaultPointcutAdvisor(advice);  
 }  
 }  
 throw new UnknownAdviceTypeException(advice);  
}

这里需要说明的是，ProxyFactoryBean继承了AdviceSupport类，因此有表示代理对象需要实现的接口的interfaces属性，也有表示通知的advisors属性。在xml文件中配置的Advice被封装为Advisor后，就被放到了ProxyFactoryBean的advisors属性中。

关注点2：创建代理对象。这个过程同前一章节类似，也是县创建一个DefaultAopProxyFactory类实例，然后调用其createProxy()方法来创建相应的代理对象。

通过分析ProxyFactoryBean的源码，我们看到，这种方式实现AOP还是比较麻烦的，并且配置一个ProxyFactoryBean仅能实现对一个目标对象的拦截，要想拦截多个目标对象，需要配置多个ProxyFactoryBean，所以通常我们还是使用Spring结合AspectJ来进行AOP编程。