结合Spring框架，在进行数据库操作的时候，经常会使用@Transactional注解，本篇将深入源码分析@Transactional注解的工作原理。

首先从<tx:annotation-driven/>说起。配置了<tx:annotation-driven/>，就必定有对应的标签解析器类，查看NamespaceHandler接口的实现类，可以看到一个TxNamespaceHandler，它注册了AnnotationDrivenBeanDefinitionParser对annotation-driven元素进行解析。

```java
public class TxNamespaceHandler extends NamespaceHandlerSupport {

    static final String TRANSACTION_MANAGER_ATTRIBUTE = "transaction-manager";

    static final String DEFAULT_TRANSACTION_MANAGER_BEAN_NAME = "transactionManager";

    static String getTransactionManagerName(Element element) {
        return (element.hasAttribute(TRANSACTION_MANAGER_ATTRIBUTE) ?
                element.getAttribute(TRANSACTION_MANAGER_ATTRIBUTE) : DEFAULT_TRANSACTION_MANAGER_BEAN_NAME);
    }


    @Override
    public void init() {
        registerBeanDefinitionParser("advice", new TxAdviceBeanDefinitionParser());
        registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
        registerBeanDefinitionParser("jta-transaction-manager", new JtaTransactionManagerBeanDefinitionParser());
    }

}
```

进入AnnotationDrivenBeanDefinitionParser类，重点看parse方法。

```java
@Override
public BeanDefinition parse(Element element, ParserContext parserContext) {
    String mode = element.getAttribute("mode");
    if ("aspectj".equals(mode)) {
        // mode="aspectj"
        registerTransactionAspect(element, parserContext);
    }
    else {
        // mode="proxy"
        AopAutoProxyConfigurer.configureAutoProxyCreator(element, parserContext);
    }
    return null;
}
```

从代码中可以看出，如果<tx:annotation-driven/>中没有配置mode参数，则默认使用代理模式进行后续处理；如果配置了mode=aspectj，则使用aspectj代码织入模式进行后续处理。

本篇分析使用代理模式的代码，进入AopAutoProxyConfigurer.configureAutoProxyCreator方法。

```java
private static class AopAutoProxyConfigurer {

    public static void configureAutoProxyCreator(Element element, ParserContext parserContext) {
        AopNamespaceUtils.registerAutoProxyCreatorIfNecessary(parserContext, element);   重要代码

        String txAdvisorBeanName = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME;
        if (!parserContext.getRegistry().containsBeanDefinition(txAdvisorBeanName)) {
            Object eleSource = parserContext.extractSource(element);

            // Create the TransactionAttributeSource definition.
            RootBeanDefinition sourceDef = new RootBeanDefinition(
                    "org.springframework.transaction.annotation.AnnotationTransactionAttributeSource");
            sourceDef.setSource(eleSource);
            sourceDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            String sourceName = parserContext.getReaderContext().registerWithGeneratedName(sourceDef);

            // Create the TransactionInterceptor definition.
            RootBeanDefinition interceptorDef = new RootBeanDefinition(TransactionInterceptor.class);
            interceptorDef.setSource(eleSource);
            interceptorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            registerTransactionManager(element, interceptorDef);
            interceptorDef.getPropertyValues().add("transactionAttributeSource", new RuntimeBeanReference(sourceName));
            String interceptorName = parserContext.getReaderContext().registerWithGeneratedName(interceptorDef);

            // Create the TransactionAttributeSourceAdvisor definition.
            RootBeanDefinition advisorDef = new RootBeanDefinition(BeanFactoryTransactionAttributeSourceAdvisor.class);
            advisorDef.setSource(eleSource);
            advisorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            advisorDef.getPropertyValues().add("transactionAttributeSource", new RuntimeBeanReference(sourceName));
            advisorDef.getPropertyValues().add("adviceBeanName", interceptorName);
            if (element.hasAttribute("order")) {
                advisorDef.getPropertyValues().add("order", element.getAttribute("order"));
            }
            parserContext.getRegistry().registerBeanDefinition(txAdvisorBeanName, advisorDef);

            CompositeComponentDefinition compositeDef = new CompositeComponentDefinition(element.getTagName(), eleSource);
            compositeDef.addNestedComponent(new BeanComponentDefinition(sourceDef, sourceName));
            compositeDef.addNestedComponent(new BeanComponentDefinition(interceptorDef, interceptorName));
            compositeDef.addNestedComponent(new BeanComponentDefinition(advisorDef, txAdvisorBeanName));
            parserContext.registerComponent(compositeDef);
        }
    }
}
```

上图代码中标出了一行核心代码，容易被忽略。进入AopNamespaceUtils.registerAutoProxyCreatorIfNecessary方法。

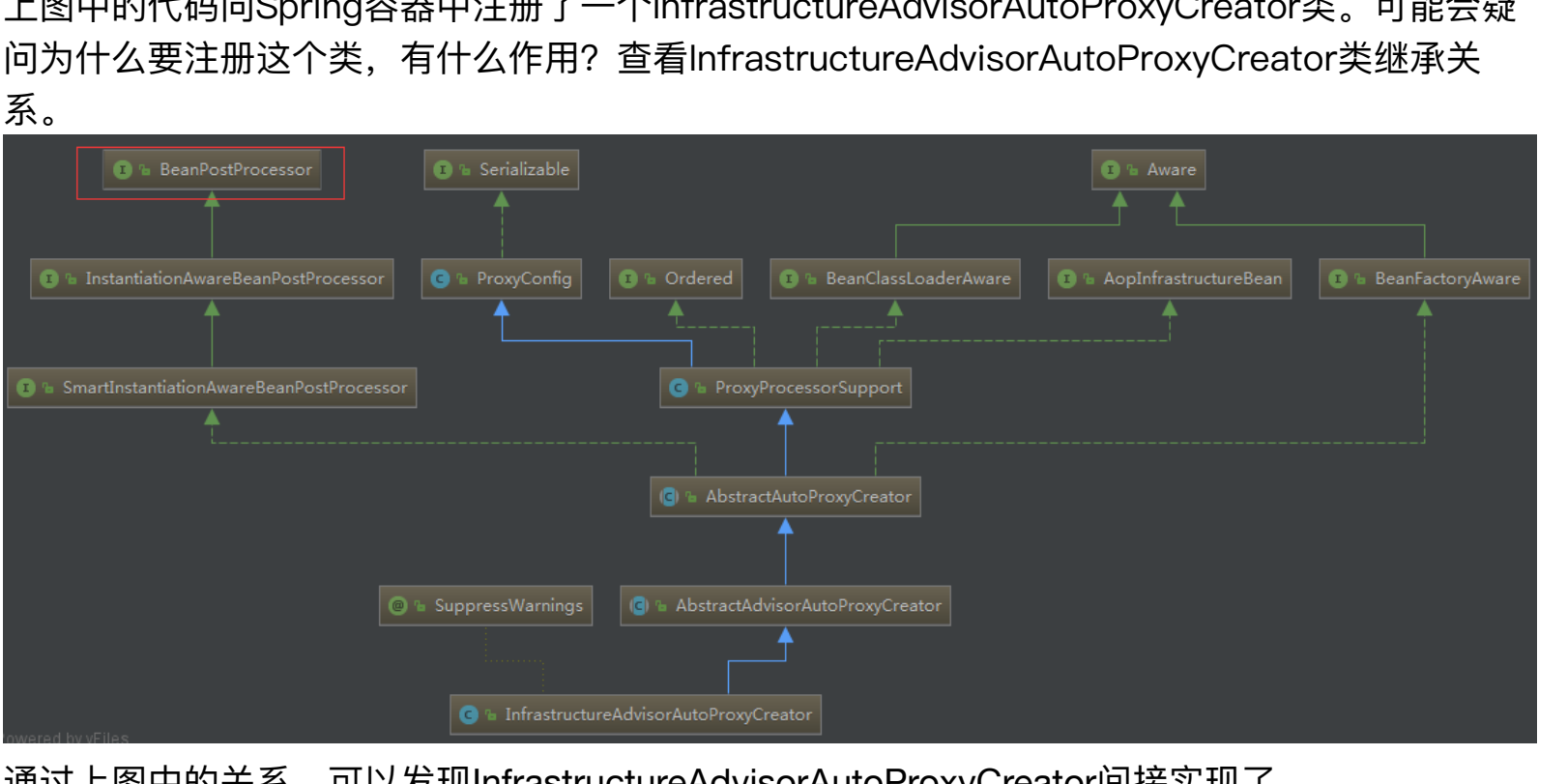
```java
public static void registerAutoProxyCreatorIfNecessary(
        ParserContext parserContext, Element sourceElement) {

    BeanDefinition beanDefinition = AopConfigUtils.registerAutoProxyCreatorIfNecessary(
            parserContext.getRegistry(), parserContext.extractSource(sourceElement));
    useClassProxyingIfNecessary(parserContext.getRegistry(), sourceElement);
    registerComponentIfNecessary(beanDefinition, parserContext);
}
```

重点关注上图中标出的代码，进入AopConfigUtils.registerAutoProxyCreatorIfNecessary方法。

```java
public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
    return registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class, registry, source);
}
```

上图中的代码向Spring容器中注册了一个InfrastructureAdvisorAutoProxyCreator类。可能会疑问为什么要注册这个类，有什么作用？查看InfrastructureAdvisorAutoProxyCreator类继承关系。



通过上图中的关系，可以发现InfrastructureAdvisorAutoProxyCreator间接实现了BeanPostProcessor接口，从AbstractAutoProxyCreator类中继承了postProcessAfterInitialization方法。Spring容器在初始化每个单例bean的时候，会遍历容器中的所有BeanPostProcessor实现类，并执行其postProcessAfterInitialization方法。

进入AbstractAutoProxyCreator类的postProcessAfterInitialization方法。

```java
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (!this.earlyProxyReferences.contains(cacheKey)) {
            return wrapIfNecessary(bean, beanName, cacheKey);
        }
    }
    return bean;
}
```

其中wrapIfNecessary方法是创建代理对象的核心方法。

```java
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;
}
```

getAdvicesAndAdvisorsForBean方法会遍历容器中所有的切面，查找与当前实例化bean匹配的切面，这里就是获取事务属性切面，查找@Transactional注解及其属性值，具体实现比较复杂，这里暂不深入分析，最终会得到BeanFactoryTransactionAttributeSourceAdvisor实例，然后根据得到的切面进入createProxy方法，创建一个AOP代理。

```java
protected Object createProxy(
        Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {

    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.copyFrom(this);

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

    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);
    }

    return proxyFactory.getProxy(getProxyClassLoader());
}
```

进入ProxyFactory.getProxy方法。

```java
public Object getProxy(ClassLoader classLoader) {
    return createAopProxy().getProxy(classLoader);
}
```

createAopProxy方法决定使用JDK还是Cglib创建代理。

```java
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()) {
            return new JdkDynamicAopProxy(config);
        }
        return new ObjenesisCglibAopProxy(config);
    }
    else {
        return new JdkDynamicAopProxy(config);
    }
}
```

可以看出默认是使用JDK动态代理创建代理，如果目标类是接口，则使用JDK动态代理，否则使用Cglib。

这里分析使用JDK动态代理的方式，进入JdkDynamicAopProxy.getProxy方法。

```java
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
    }
    Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

可以看到很熟悉的创建代理的代码Proxy.newProxyInstance。这里要注意的是，newProxyInstance方法的最后一个参数是JdkDynamicAopProxy类本身，也就是说在对目标类进行调用的时候，会进入JdkDynamicAopProxy的invoke方法。

这里只关注JdkDynamicAopProxy的invoke方法的重点代码。

```java
// 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()) {
    // 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.
    retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
}
else {
    // 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.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;
```

this.advised.getInterceptorsAndDynamicInterceptionAdvice获取的是当前目标方法对应的拦截器，里面是根据之前获取到的切面来获取相对应拦截器，这时候会得到TransactionInterceptor实例。如果获取不到拦截器，则不会创建MethodInvocation，直接调用目标方法。这里使用TransactionInterceptor创建一个ReflectiveMethodInvocation实例，调用的时候进入ReflectiveMethodInvocation的proceed方法。

```java
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);
    }
}
```

代码中的interceptorOrInterceptionAdvice就是TransactionInterceptor的实例，执行invoke方法进入TransactionInterceptor的invoke方法。

```java
public Object invoke(final MethodInvocation invocation) throws Throwable {
    // Work out the target class: may be {@code null}.
    // The TransactionAttributeSource should be passed the target class
    // as well as the method, which may be from an interface.
    Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);

    // Adapt to TransactionAspectSupport's invokeWithinTransaction...
    return invokeWithinTransaction(invocation.getMethod(), targetClass, new InvocationCallback() {
        @Override
        public Object proceedWithInvocation() throws Throwable {
            return invocation.proceed();
        }
    });
}
```

TransactionInterceptor从父类TransactionAspectSupport中继承了invokeWithinTransaction方法。

```java
protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)
        throws Throwable {

    // If the transaction attribute is null, the method is non-transactional.
    final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
    final PlatformTransactionManager tm = determineTransactionManager(txAttr);
    final String joinpointIdentification = methodIdentification(method, targetClass);
                                                                         声明式事务处理
    if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
        // Standard transaction demarcation with getTransaction and commit/rollback calls.
        TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification); 是否开启事务操作
        Object retVal = null;
        try {
            // This is an around advice: Invoke the next interceptor in the chain.
            // This will normally result in a target object being invoked.
            retVal = invocation.proceedWithInvocation(); 调用目标类的目标方法
        }
        catch (Throwable ex) {
            // target invocation exception
            completeTransactionAfterThrowing(txInfo, ex); 事务回滚操作
            throw ex;
        }
        finally {
            cleanupTransactionInfo(txInfo);
        }
        commitTransactionAfterReturning(txInfo); 事务提交操作
        return retVal;
    }
                                                                         编程式事务处理
    else {
        // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in.
        try {
            Object result = ((CallbackPreferringPlatformTransactionManager) tm).execute(txAttr,
                    (status) -> {
                        TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
                        try {
                            return invocation.proceedWithInvocation();
                        }
                        catch (Throwable ex) {
                            if (txAttr.rollbackOn(ex)) {
                                // A RuntimeException: will lead to a rollback.
```

可以看到，在需要进行事务操作的时候，Spring会在调用目标类的目标方法之前进行开启事务、调用异常回滚事务、调用完成会提交事务。

是否需要开启新事务，是根据@Transactional注解上配置的参数值来判断的。如果需要开启新事务，获取Connection连接，然后将连接的自动提交事务改为false，改为手动提交。

当对目标类的目标方法进行调用的时候，若发生异常将会进入completeTransactionAfterThrowing方法。

```java
protected void completeTransactionAfterThrowing(TransactionInfo txInfo, Throwable ex) {
    if (txInfo != null && txInfo.hasTransaction()) {
        if (logger.isTraceEnabled()) {
            logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +
                    "] after exception: " + ex);
        }
        if (txInfo.transactionAttribute.rollbackOn(ex)) {   针对某些异常才会进行事务回滚
            try {
                txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {
                logger.error("Application exception overridden by rollback exception", ex);
                ex2.initApplicationException(ex);
                throw ex2;
            }
            catch (RuntimeException ex2) {
                logger.error("Application exception overridden by rollback exception", ex);
                throw ex2;
            }
            catch (Error err) {
                logger.error("Application exception overridden by rollback error", ex);
                throw err;
            }
        }
                                                                         不会回滚事务
        else {
            // We don't roll back on this exception.
            // Will still roll back if TransactionStatus.isRollbackOnly() is true.
            try {
                txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {
                logger.error("Application exception overridden by commit exception", ex);
                ex2.initApplicationException(ex);
                throw ex2;
            }
            catch (RuntimeException ex2) {
                logger.error("Application exception overridden by commit exception", ex);
                throw ex2;
            }
            catch (Error err) {
```

Spring并不会对所有类型异常都进行事务回滚操作，默认是只对Unchecked Exception(Error和RuntimeException)进行事务回滚操作。

```java
public boolean rollbackOn(Throwable ex) {
    return (ex instanceof RuntimeException || ex instanceof Error);
}
```

总结

从上面的分析可以看到，Spring使用AOP实现事务的统一管理，为开发者提供了很大的便利。但是，有部分开发人员会误用这个便利，基本都是下面这两种情况：

1.A类的a1方法没有标注@Transactional，a2方法标注@Transactional，在a1里面调用a2；

2.将@Transactional注解标注在非public方法上。

第一种为什么是错误用法，原因很简单，a1方法是目标类A的原生方法，调用a1的时候即直接进入目标类A进行调用，在目标类A里面只有a2的原生方法，在a1里调用a2，即直接执行a2的原生方法，并不通过创建代理对象进行调用，所以并不会进入TransactionInterceptor的invoke方法，不会开启事务。

@Transactional的工作机制是基于AOP实现的，而AOP是使用动态代理实现的，动态代理要么是JDK方式、要么是Cglib方式。如果是JDK动态代理的方式，根据上面的分析可以知道，目标类的目标方法是在接口中定义的，也就是必须是public修饰的方法才可以被代理。如果是Cglib方式，代理类是目标类的子类，理论上可以代理public和protected方法，但是Spring在进行事务增强是否能够应用到当前目标类判断的时候，遍历的是目标类的public方法，所以Cglib方式也只对public方法有效。

```java
public static boolean canApply(Pointcut pc, Class<?> targetClass, boolean hasIntroductions) {
    Assert.notNull(pc, "Pointcut must not be null");
    if (!pc.getClassFilter().matches(targetClass)) {
        return false;
    }

    MethodMatcher methodMatcher = pc.getMethodMatcher();
    IntroductionAwareMethodMatcher introductionAwareMethodMatcher = null;
    if (methodMatcher instanceof IntroductionAwareMethodMatcher) {
        introductionAwareMethodMatcher = (IntroductionAwareMethodMatcher) methodMatcher;
    }

    Set<Class<?>> classes = new LinkedHashSet<>(ClassUtils.getAllInterfacesForClassAsSet(targetClass));
    classes.add(targetClass);
    for (Class<?> clazz : classes) {
        Method[] methods = clazz.getMethods();
        for (Method method : methods) {
            if ((introductionAwareMethodMatcher != null &&
                    introductionAwareMethodMatcher.matches(method, targetClass, hasIntroductions)) ||
                    methodMatcher.matches(method, targetClass)) {
                return true;
            }
        }
    }

    return false;
}
```

深入Class类getMethods方法，可以看到取得是public修饰的方法。

```java
public Method[] getMethods() throws SecurityException {
    // be very careful not to change the stack depth of this
    // checkMemberAccess call for security reasons
    // see java.lang.SecurityManager.checkMemberAccess
    checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), true);
    return copyMethods(privateGetPublicMethods());
}
```