CGLIB proxies should still consider @Transactional annotations on interface methods [SPR-14322] #1

> https://github.com/spring-projects/spring-framework/issues/18894

If an application components implements an interface whose methods carry annotations that are triggering interceptors (e.g. for transactions), enabling target class proxying will result in the interceptors for those annotations not being triggered anymore. Here's a sample:
如果一个组件实现了某一个接口，这个接口的方法上使用了可以出发Interceptor的注解，比如事务。
如果设置了proxyTargetClass为true将会导致 没有触发interceptor

```
interface SomeComponent {

 @Transactional
 void init();
}

@Component
class SomeComponentImpl implements SomeComponent {

 @Override
 public void init() {
 if (!TransactionSynchronizationManager.isActualTransactionActive()) {
 throw new IllegalStateException("Expected transaction to be active!");
 }
 }
}

@Component
class Invoker {

 public Invoker(List<SomeComponent> components) {
 components.forEach(SomeComponent::init);
 }
}

启动类上使用@EnableTransactionManagement(proxyTargetClass = false)
来决定是用JDK动态代理还是cglib代理

```

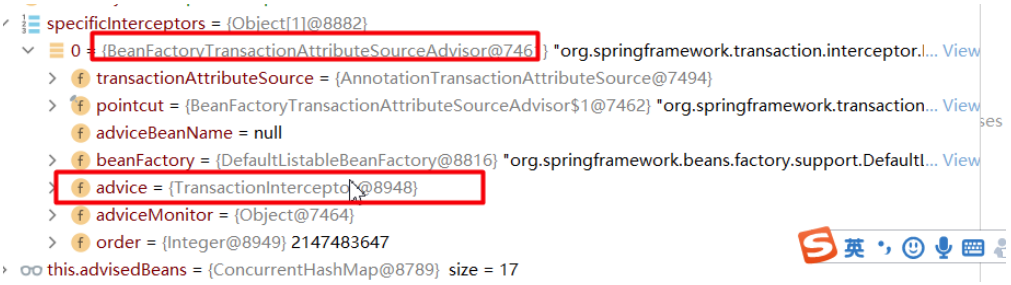
If the above is bootstrapped with standard `@EnableTransactionManagement` the instances handed to the constructor of `Invoker` are JDK proxies and the lookup of the advice chain results in the interceptor for transactions being returned and thus activated. If `proxyTargetClass` is set to `true`, the instances received by the constructor are CGLib proxies and the lookup of the advice chain results in an empty one and thus no transaction is created in the first place.

@Transactional定义在接口上的时候，如果我们使用的是接口代理，那么最终返回的advice中会包含TransactionInterceptor。 但是如果是使用CGLib代理则 返回了一个空的advice。这个该如何理解?

我们知道代理的入口是org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#wrapIfNecessary
在这个wrapIfNecessary中。
首先是执行getAdvicesAndAdvisorsForBean 进行获取interceptors，然后如果获取到的interceptor不是空数组，则就会createProxy。

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
    if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
        return bean;
    } else if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
        return bean;
    } else if (!this.isInfrastructureClass(bean.getClass()) && !this.shouldSkip(bean.getClass(), beanName)) {
        Object[] specificInterceptors = this.getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, (TargetSource)null);
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            Object proxy = this.createProxy(bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        } else {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }
    } else {
        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
}
```

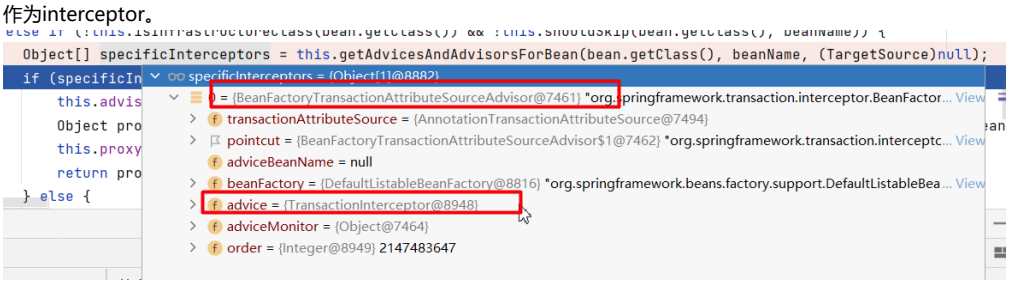
1，当proxyTargetClass为false，使用jdk代理时，返回一个Advisor，advisor中的advice就是TransactionInterceptor



这个BeanFactoryTransactionAttributeSourceAdvisor对象是在ProxyTransactionManagementConfiguration这个配置类中被注入的

```
@Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor() {
  BeanFactoryTransactionAttributeSourceAdvisor advisor = new BeanFactoryTransactionAttributeSourceAdvisor();
  advisor.setTransactionAttributeSource(transactionAttributeSource());
  advisor.setAdvice(transactionInterceptor());
  advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
  return advisor;
}
```

2，当proxyTargetClass为true时候，上面的getAdvicesAndAdvisorsForBean 也会返回一个BeanFactoryTransactionAttributeSourceAdvisor对象作为interceptor。



也就是说 不管是 proxyTargetClass为true 还是false，我们都可以为SomeComponentImpl对象找到 specificInterceptors。

然后我们分析第二个 createProxy，主要考虑proxyTargetClass为true的情况

```
123
124     protected Object createProxy(Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {
125         if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
126             AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)this.beanFactory, beanName, beanClass);
127         }
128
129         ProxyFactory proxyFactory = new ProxyFactory();
130         proxyFactory.copyFrom(this);
131         if (!proxyFactory.isProxyTargetClass()) {
132             if (this.shouldProxyTargetClass(beanClass, beanName)) {
133                 proxyFactory.setProxyTargetClass(true);
134             } else {
135                 this.evaluateProxyInterfaces(beanClass, proxyFactory);
136             }
137         }
138
139         Advisor[] advisors = this.buildAdvisors(beanName, specificInterceptors);
140         Advisor[] var7 = advisors;
141         int var8 = advisors.length;
142
143         for(int var9 = 0; var9 < var8; ++var9) {
144             Advisor advisor = var7[var9];
145             proxyFactory.addAdvisor(advisor);
146         }
147
148         proxyFactory.setTargetSource(targetSource);
149         this.customizeProxyFactory(proxyFactory);
150         proxyFactory.setFrozen(this.freezeProxy);
151         if (this.advisorsPreFiltered()) {
152             proxyFactory.setPreFiltered(true);
```



```
153         }
154
155         return proxyFactory.getProxy(this.getProxyClassLoader());
156     }
157
```

在createProxy方法中，首先我们 会根据proxyFactory中的proxyTargetClass属性，同时会判断BeanDefinition中的preserveTargetClass属性，
第二个就是buildAdvisor，buildAdvisor的参数是前面getAdvicesAndAdvisorsForBean 的返回值，
这里buildAdvisor的返回值才会被真正作为Advisor 创建Proxy
然后在后面的proxyFactory.getProxy中会根据proxyTargetClass来判断是否使用cglib代理

```
public static boolean shouldProxyTargetClass(ConfigurableListableBeanFactory beanFactory, String beanName) {
    if (beanName != null && beanFactory.containsBeanDefinition(beanName)) {
        BeanDefinition bd = beanFactory.getBeanDefinition(beanName);
        return Boolean.TRUE.equals(bd.getAttribute(PRESERVE_TARGET_CLASS_ATTRIBUTE));
    } else {
        return false;
    }
}
```

```
public abstract class AutoProxyUtils {
    public static final String PRESERVE_TARGET_CLASS_ATTRIBUTE = Conventions.getQualifiedAttributeName(AutoProxyUtils.class, attributeName: "preserveTargetClass");
    public static final String ORIGINAL_TARGET_CLASS_ATTRIBUTE = Conventions.getQualifiedAttributeName(AutoProxyUtils.class, attributeName: "originalTargetClass");
```

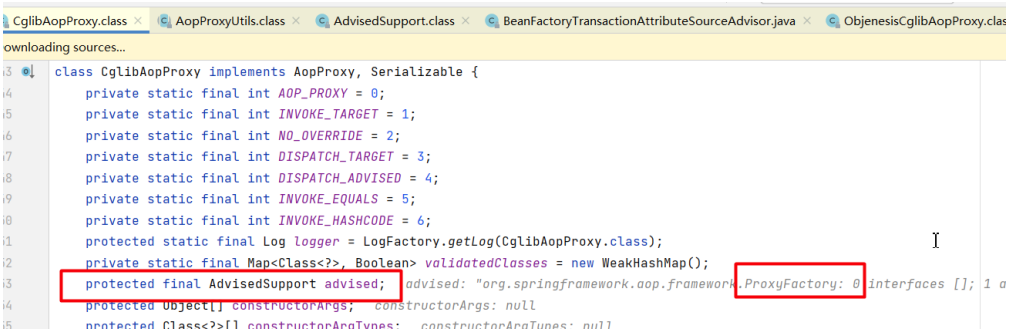
我们现在的疑问是： 不管是proxyTargetClass为true 还是false，在getAdvicesAndAdvisorsForBean 和createProxy方法内的buildAdvisor 中都返回了BeanFactoryTransactionAttributeSourceAdvisor作为Advisor，那么为什么proxyTargetClass为true的时候没有实现事务代理，而仅在proxyTargetClass为false的时候执行SomeComponentImpl 的init方法有事务？

第一：当proxyTargetClass 为false的时候使用jdk动态代理

第二 当proxyTargetClass为true的是偶 使用cglib代理，proxyFactory的getProxy最终会触发下面的getProxy
在org.springframework.aop.framework.CglibAopProxy#getProxy(java.lang.ClassLoader)

其中当前对象是ObjenesisCglibAopProxy（class ObjenesisCglibAopProxy extends CglibAopProxy），proxyFactory.getProxy的过程中会创建一个AopProxy，这个AopProxy可能是JdkDynamicAopProxy 或者ObjenesisCglibAopProxy。在这里是ObjenesisCglibAopProxy对象，同时 ProxyFactory在创建AopProxy的时候会将自身传递给AopProxy，

在CglibAopProxy中的 AdvisedSupport advised;属性就是ProxyFactory

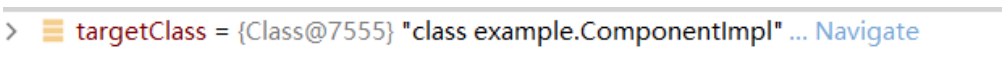


因此之类的this.advised 就是ProxyFactory对象

enhancer.setInterfaces(AopProxyUtils.*completeProxiedInterfaces*(this.advised));

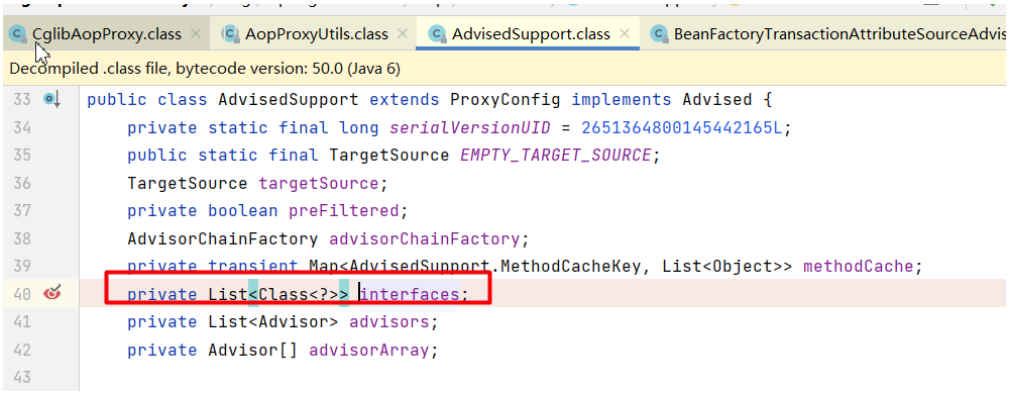
在completeProxiedInterfaces 方法中会advised.getTargetClass(); 获取到当前创建的Bean的 class，

```
public static Class<?>[] completeProxiedInterfaces(AdvisedSupport advised) {
    Class<?>[] specifiedInterfaces = advised.getProxiedInterfaces();
    if (specifiedInterfaces.length == 0) {
        Class<?> targetClass = advised.getTargetClass();
        if (targetClass != null) {
            if (targetClass.isInterface()) {
                advised.setInterfaces(new Class[]{targetClass});
            } else if (Proxy.isProxyClass(targetClass)) {
                advised.setInterfaces(targetClass.getInterfaces());
            }

            specifiedInterfaces = advised.getProxiedInterfaces();
        }
    }
```



ProxyFactory中的targetClass是在上面的wrapIfNecessary方法中 调用了createProxy方法中创建了ProxyFactory，在创建ProxyFactory的时候设置了一些属性。

在上面的completeProxiedInterfaces方法中， getTargetClasss之后 又将这个targetClass设置到 advised的interfaces属性中，也就是advised.setInterfaces(new Class[]{targetClass});
对应着是



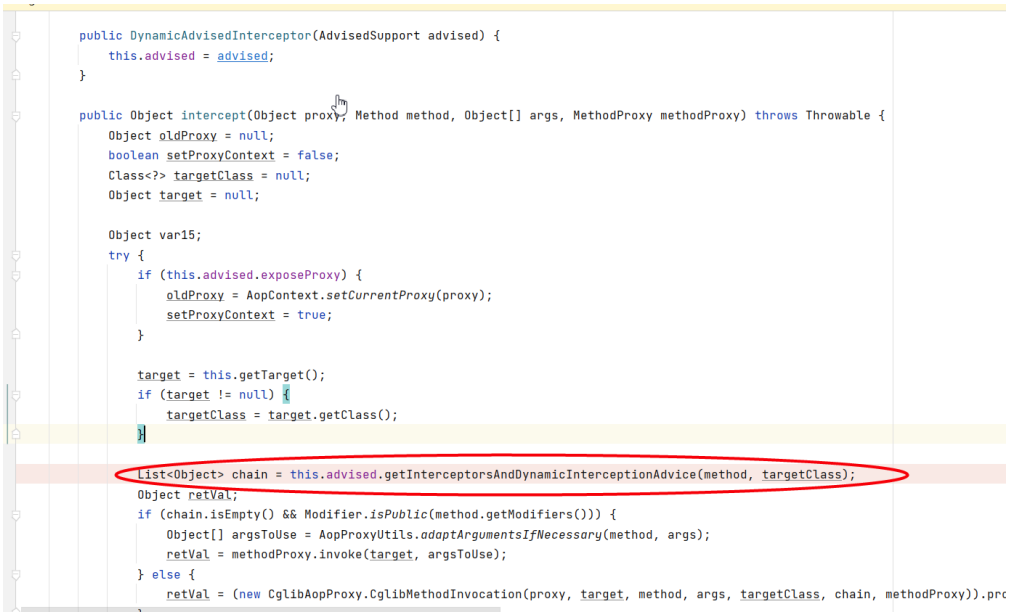
这个interfaces会被作为enhancer的interfaces属性
enhancer.setInterfaces(AopProxyUtils.*completeProxiedInterfaces*(this.advised));

然后又继续配置enhancer的属性

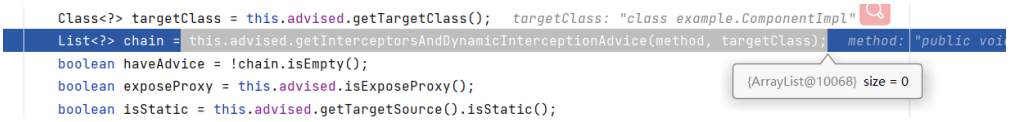
```
Callback[] callbacks = this.getCallbacks(rootClass);
Class<?>[] types = new Class[callbacks.length];

for(x = 0; x < types.length; ++x) {
 types[x] = callbacks[x].getClass();
}

enhancer.setCallbackFilter(new CglibAopProxy.ProxyCallbackFilter(this.advised.getConfiguratio
OnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));
```

getCallbacks方法中会设置一个特殊的interceptor CglibAopProxy.DynamicAdvisedInterceptor(this.advised);

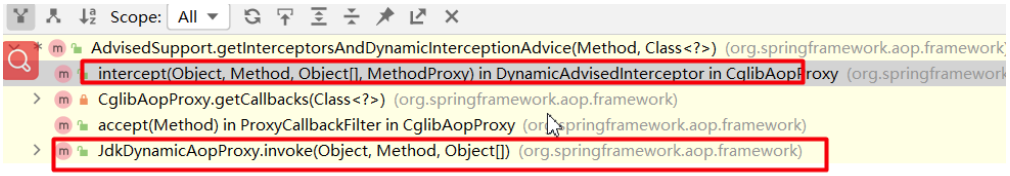
在这个特殊的interceptor中会 构建一个调用链



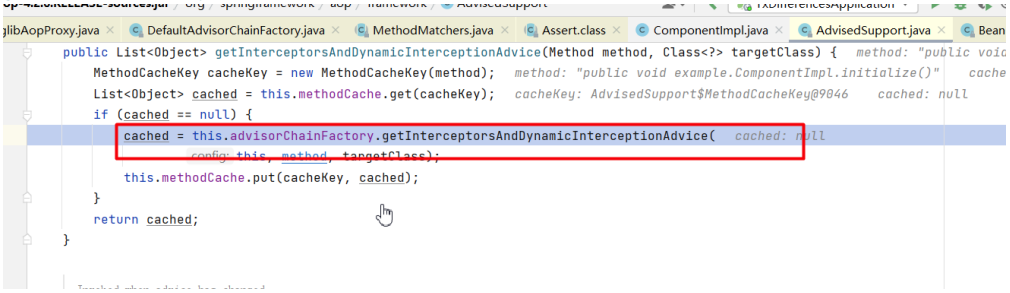
上面的getInterceptorsAndDynamicInterceptionAdvice 的size为0，也就是没有将TransactionInterceptor作为interceptor，这就会导致cglib代理生成的对象调用其方法时 构建的调用链中并没有TransactionInterceptor对象作为Interceptor



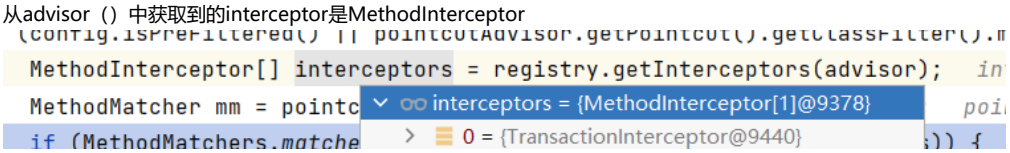
这个getInterceptorsAndDynamicInterceptionAdvice方法很重要，他在两个地方被调用到，分别对应Cglib的AOp实现和JDK动态代理的Aop实现，也就是说



getInterceptorsAndDynamicInterceptionAdvice的实现如下



在上图中我们发现 最终实现的过程中 遍历每一个Advisor，其中config就是ProxyFactory， 其中
从advisor（）中获取到的interceptor是MethodInterceptor



PointCut是

```
/serial/
public class BeanFactoryTransactionAttributeSourceAdvisor extends AbstractBeanFactoryPointcutAdvisor {

    private TransactionAttributeSource transactionAttributeSource;   transactionAttributeSource: null

    private final TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePointcut() {
        @Override
        protected TransactionAttributeSource getTransactionAttributeSource() {
            return transactionAttributeSource;
        }
    };
```

二

插播一段，在wrapIfNecessary 方法内会调用getAdvicesAndAdvisorsForBean

```
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {   bean: ComponentImpl@15073   beanName: "componentImpl"
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);   cacheKey: "componentImpl"
        if (!this.earlyProxyReferences.contains(cacheKey)) {
            return wrapIfNecessary(bean, beanName, cacheKey);   bean: ComponentImpl@15073   beanName: "componentImpl"   cacheKey: "componentImpl"
        }
    }
    return bean;
}
```

Debugger | Console | Actuator

Frames | Threads

"main"@1 in group "main": RUNNING

```
getTransactionAttribute:102, AbstractFallbackTransactionAttributeSource (org.springframework.transa
matches:41, TransactionAttributeSourcePointcut (org.springframework.transaction.interceptor)
canApply:211, AopUtils (org.springframework.aop.support)
canApply:248, AopUtils (org.springframework.aop.support)
findAdvisorsThatCanApply:280, AopUtils (org.springframework.aop.support)
findAdvisorsThatCanApply:118, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framewo
findEligibleAdvisors:88, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autop
getAdvicesAndAdvisorsForBean:69, AbstractAdvisorAutoProxyCreator (org.springframework.aop.fram
wrapIfNecessary:346, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
```

Variables

this = {Annota
beanClass = {
beanName =
targetSource

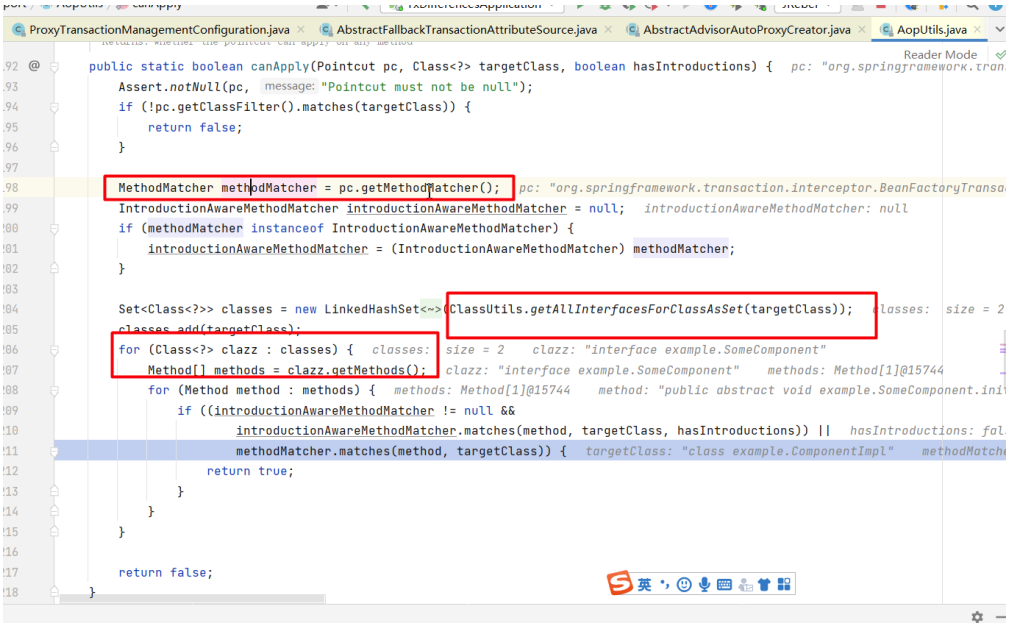
在findAdvisorsThatCanApply方法中会遍历每一个Advisor，判断这个Adisor是否可以应用到当前Bean的类上。

```
boolean hasIntroductions = !eligibleAdvisors.isEmpty();   eligibleAdvisor
for (Advisor candidate : candidateAdvisors) {   candidateAdvisors:  size
    if (candidate instanceof IntroductionAdvisor) {
        // already processed
        continue;
    }
    if (canApply(candidate, clazz, hasIntroductions)) {   clazz: "class e
        eligibleAdvisors.add(candidate);
    }
}
return eligibleAdvisors;
```

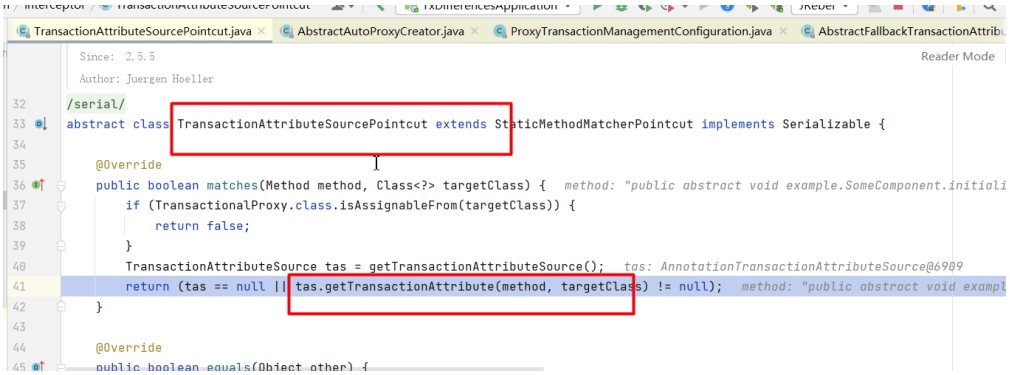
其中有一个Advisor就是BeanFactorytransactionAttributeSourceAdvisor，属性advice是MethodInterceptor，pointcut是TransactionAttributeSourcePointcut

```
275        for (Advisor candidate : candidateAdvisors) {   candidateAdvisors:  size = 1   candidate: "org.springframework.transact
candidate = {BeanFactoryTransactionAttributeSourceAdvisor@6833} "org.springframework.transaction.interceptor.BeanFactoryTransactionAttributeSourceAdvisor: advice bean 'null'"
  transactionAttributeSource = {AnnotationTransactionAttributeSource@6909}
  pointcut = {BeanFactoryTransactionAttributeSourceAdvisor$1@6834} "org.springframework.transaction.interceptor.BeanFactoryTransactionAttributeSourceAdvisor$1: org.spri... View
  adviceBeanName = null
  beanFactory = {DefaultListableBeanFactory@6840} "org.springframework.beans.factory.support.DefaultListableBeanFactory@19db3c2e: defining beans [org.springframework.cc... View
  advice = {TransactionInterceptor@15980}
  adviceMonitor = {Object@6836}
  order = {Integer@15981} 2147483647
```

在canApply方法中会 针对class的所有Interfaces中的每一个方法判断 MethodMatcher是否match

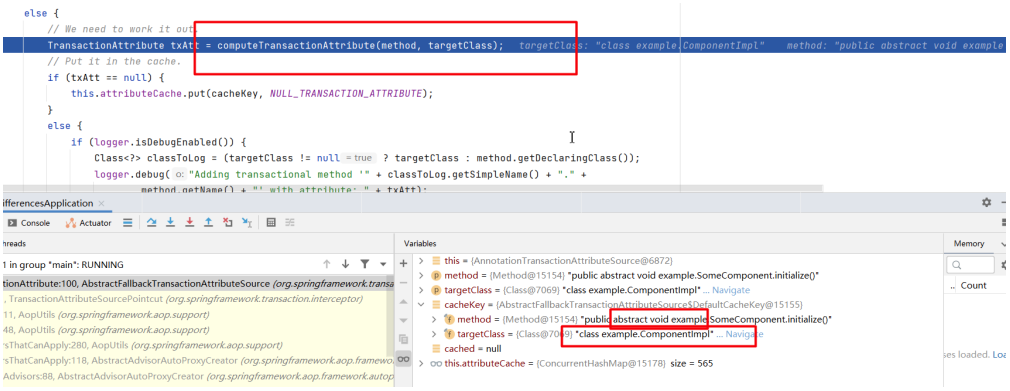


是否match的依据 是判断方法上是否有@Transactional注解，这个methodMatcher如下

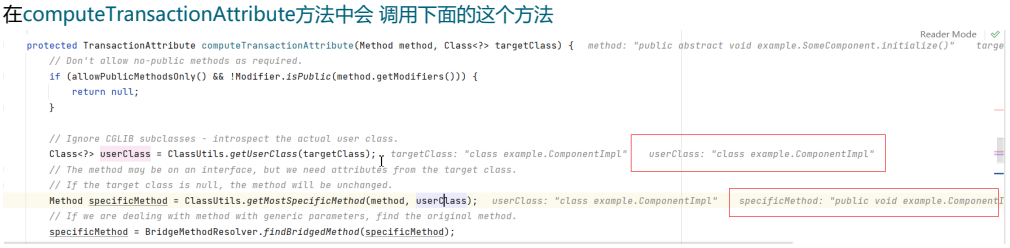


假设我们在接口的方法上使用了@Transactional注解，而且指定使用Cglib代理（proxyTargetClass为true）
那么上面的canApply方法会遍历接口上的每一个方法，然后调用下面的getTransactionAttribute

在遍历的过程中会遍历接口example.SomeComponent#initialize中的initialize方法，然后分析这个方法上@Transactional注解，注意下面的method是存在abstract修饰符

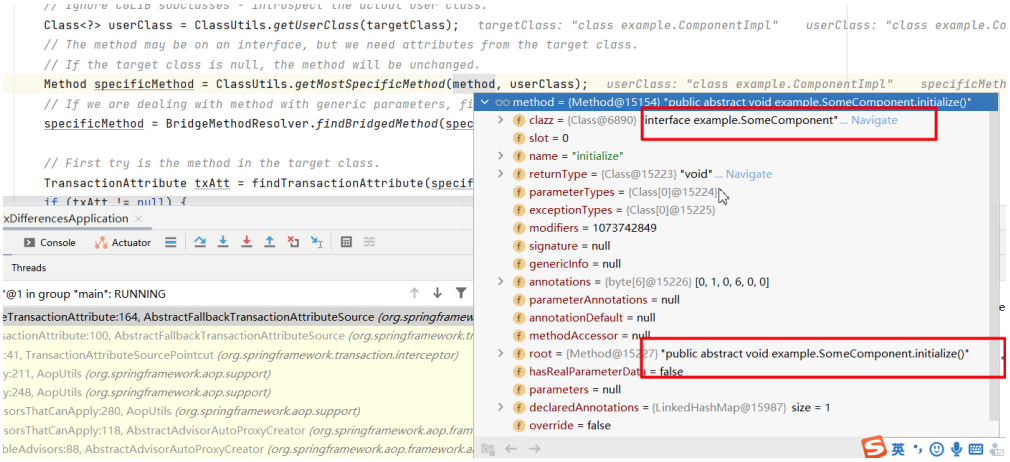


在computeTransactionAttribute方法中会 调用下面的这个方法

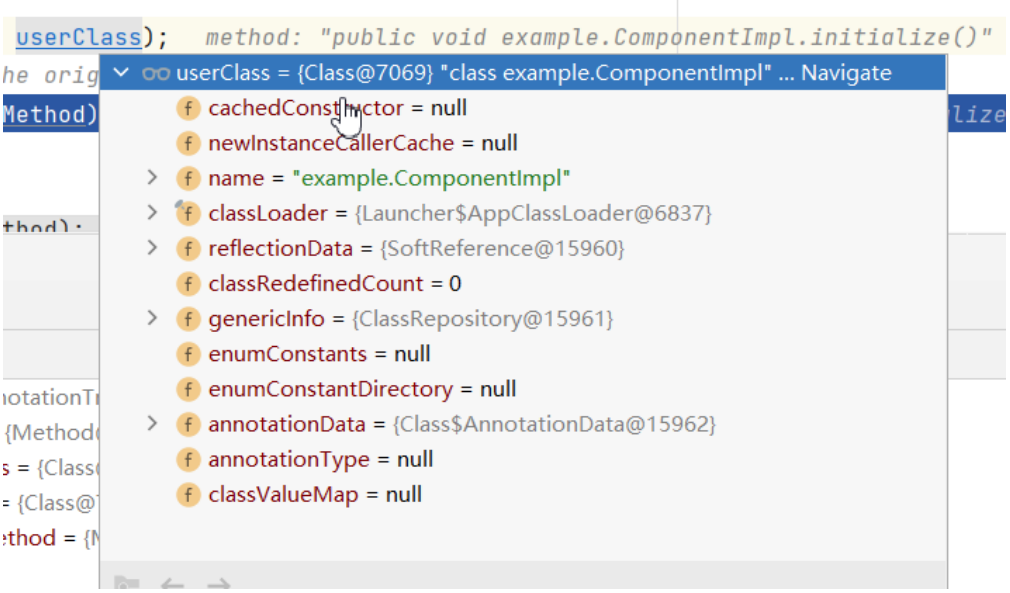


其中ClassUtils.*getUserClass*(targetClass); 这里的targetClass就是 Bean的实现类example.ComponentImpl

然后method是通过接口class 获取到的method对象，

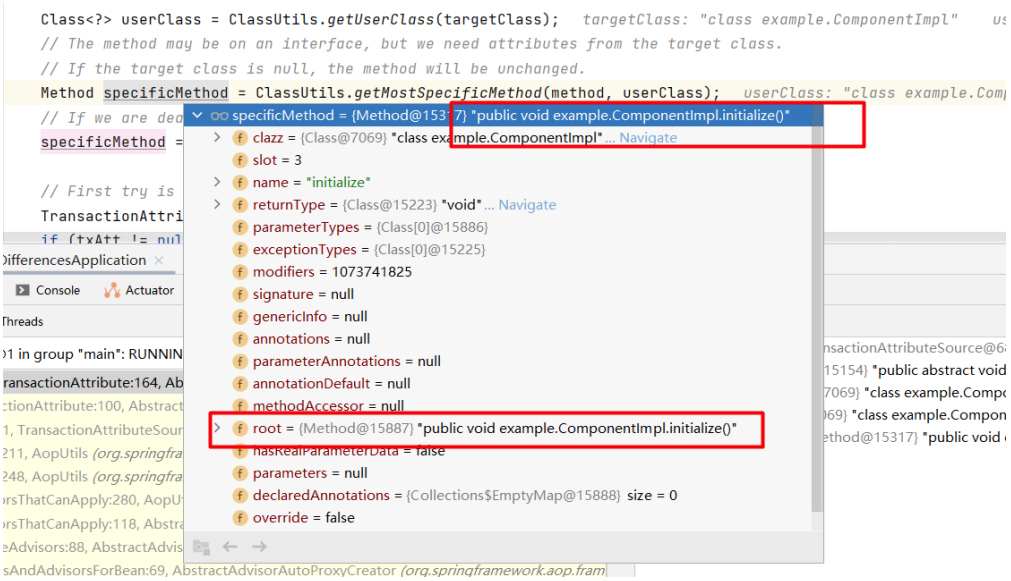


userClass是



ClassUtils.*getMostSpecificMethod*(method, userClass); 方法就是在接口的实现类上获取该方法的实现方法

所以specificMethod就是下面这个 具体的实现类的方法。



然后我们首先在实现类的方法上寻找TransactionAttribute，如果实现类上有Attribute则表示存在事务。

```
// First try is the method in the target class.
TransactionAttribute txAtt = findTransactionAttribute(specificMethod);
if (txAtt != null) {
    return txAtt;
}
```

如果实现类上没有，则我们在实现类上寻找TransactionAttribute

```
// Second try is the transaction attribute on the target class.
txAtt = findTransactionAttribute(specificMethod.getDeclaringClass());
if (txAtt != null) {
    return txAtt;
}
```

最后，如果specificMethod不等于method，则就在method上找注解，这里的method就是接口类中的abstract method

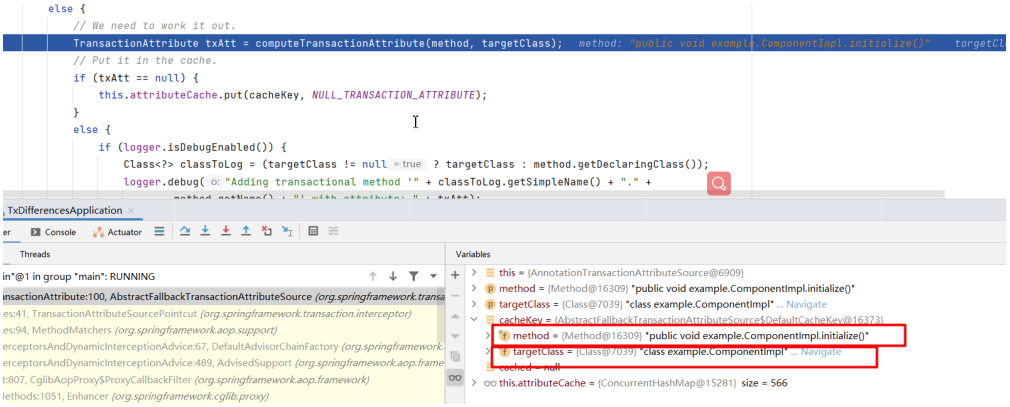
```
if (specificMethod != method) {
    // Fallback is to look at the original method.
    txAtt = findTransactionAttribute(method);
    if (txAtt != null) {
        return txAtt;
    }
    // Last fallback is the class of the original method.
    return findTransactionAttribute(method.getDeclaringClass());
}
```

因此当接口的方法上存在注解是时候，我们以接口中的抽象方法 获取TransactionAttribute的时候是可以获取到TransactionAttribute

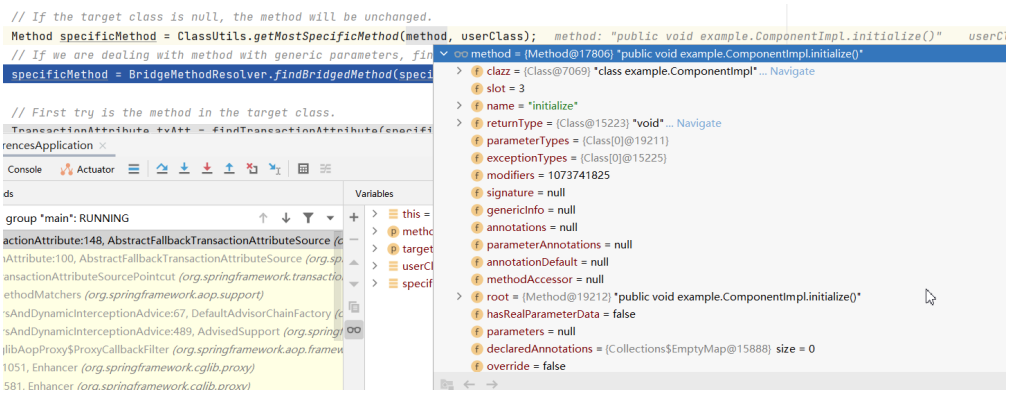
我想表达的是在Bean的创建过程中 会分析Bean的接口 上的每一个方法，这个方法在实现类的方法上是否有注解，在实现了上是否有注解，在接口的抽象方法上是否有注解，最终解析的结果会被放置到org.springframework.transaction.interceptor.AbstractFallbackTransactionAttributeSource#attributeCache属性中，因此每一个方法都被标记位是否支持事务。

```
// We need to work it out.
TransactionAttribute txAtt = computeTransactionAttribute(method, targetClass);
// Put it in the cache.
if (txAtt == null) {
    this.attributeCache.put(cacheKey, NULL_TRANSACTION_ATTRIBUTE);
}
```

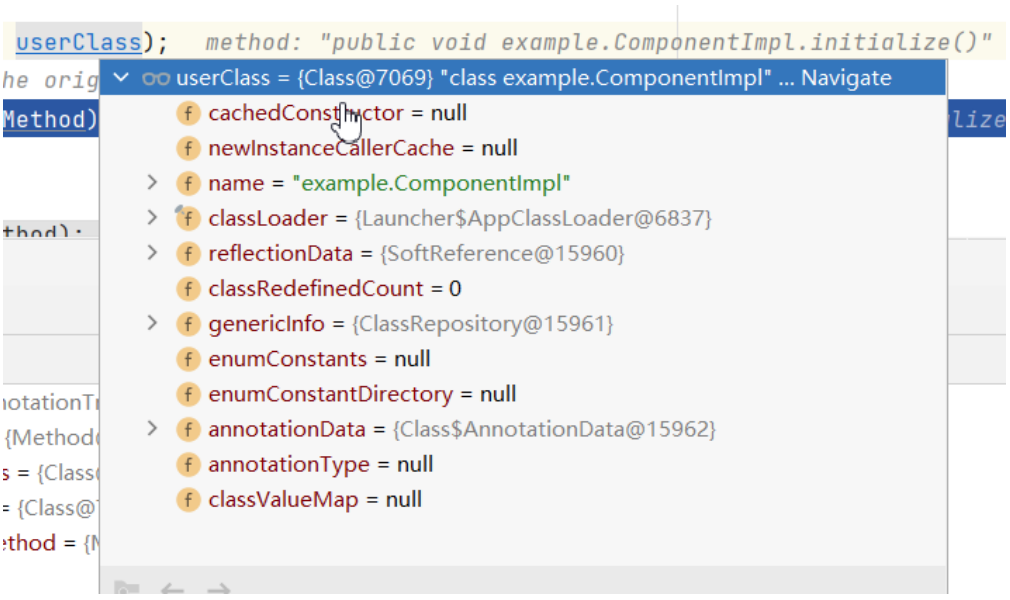
在使用了cglib代理的情况下 如果我们以ComponentImpl类中的initialize方法 去分析是否有TransactionAttribute，会返回null，也就是没有事务，注意下面的method 中没有abstract修饰符



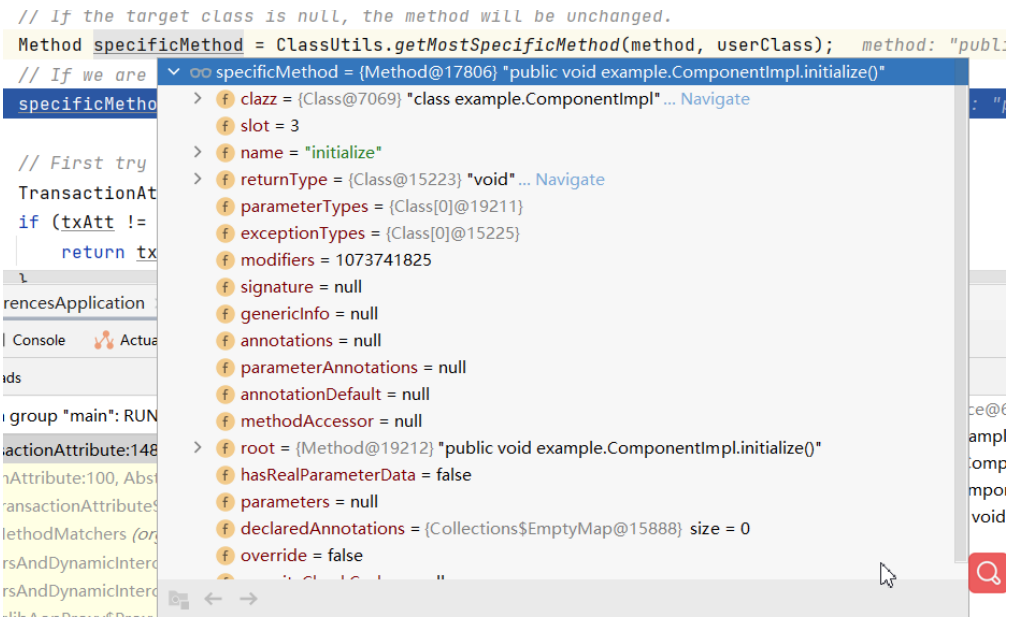
为什么会找不到事务呢？ 这个时候computeTransactionAttribute方法的入参method是



userClass是

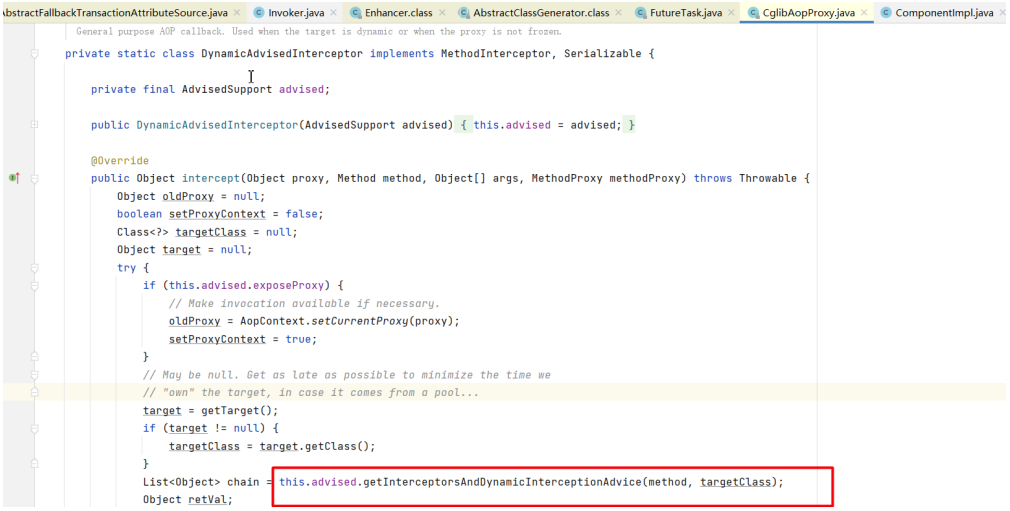


然后ClassUtils.*getMostSpecificMethod方法返回的*specificMethod方法



然后我们就会发现specificMethod等于method，而这个实现类的方法上并没有@Transactional注解，因此找不到TransactionalAttribute

当我们调用Cglib对象的方法的时候会执行DynamicAdvisedInterceptor的intercept方法，在内部会获取匹配的advisor是advice，因为对于事务Advisor来说， 没有解析到方法上的TransactionalAttribute所以 cglibg返回的chain的size为0，也就是没有TransactionalInterceptor，因此Cglib生成的代理对象的方法没有事务。



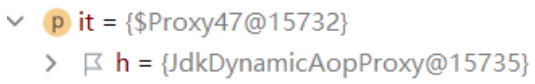
对于Cglib代理，在wrapIfNecessary的过程中会调用getProxy，在getProxy的过程中会使用 enhancer.createClass();,

在生成class的过程中会生成每一个方法，org.springframework.cglib.proxy.Enhancer#getMethods(java.lang.Class, java.lang.Class[], java.util.List, java.util.List, java.util.Set)

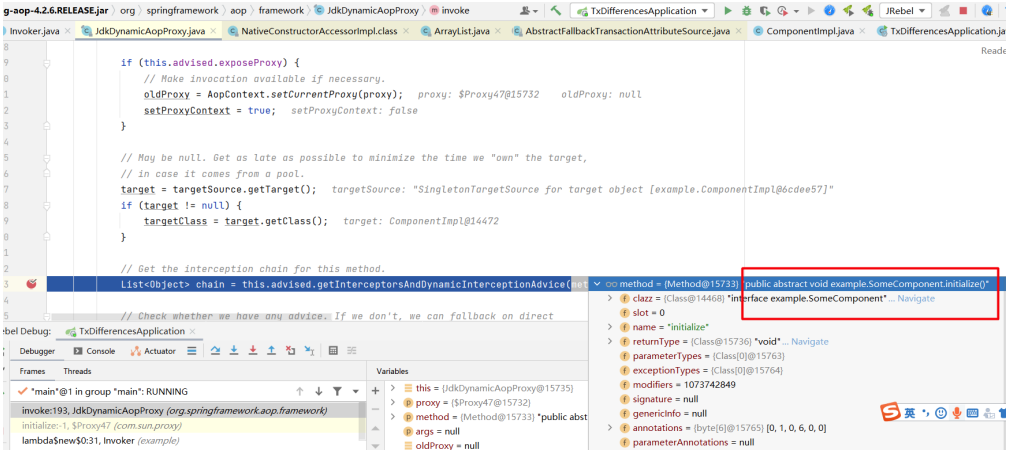
```
1

2 computeTransactionAttribute:148, AbstractFallbackTransactionAttributeSource (org.springframework.transaction.interceptor)
3 getTransactionAttribute:100, AbstractFallbackTransactionAttributeSource (org.springframework.transaction.interceptor)
4 matches:41, TransactionAttributeSourcePointcut (org.springframework.transaction.interceptor)
5 matches:94, MethodMatchers (org.springframework.aop.support)
```

```
6 getInterceptorsAndDynamicInterceptionAdvice:67, DefaultAdvisorChainFactory (org.springframework.aop.framework)
7 getInterceptorsAndDynamicInterceptionAdvice:489, AdvisedSupport (org.springframework.aop.framework)
8 accept:807, CglibAopProxy$ProxyCallbackFilter (org.springframework.aop.framework)
9 emitMethods:1051, Enhancer (org.springframework.cglib.proxy)
10 generateClass:581, Enhancer (org.springframework.cglib.proxy)
11 generateClass:33, TransformingClassGenerator (org.springframework.cglib.transform)
12 generate:25, DefaultGeneratorStrategy (org.springframework.cglib.core)
13 generate:990, CglibAopProxy$ClassLoaderAwareUndeclaredThrowableStrategy (org.springframework.aop.framework)
14 generate:312, AbstractClassGenerator (org.springframework.cglib.core)
15 generate:445, Enhancer (org.springframework.cglib.proxy)
16 apply:85, AbstractClassGenerator$ClassLoaderData$3 (org.springframework.cglib.core)
17 apply:83, AbstractClassGenerator$ClassLoaderData$3 (org.springframework.cglib.core)
18 call:54, LoadingCache$2 (org.springframework.cglib.core.internal)
19 run:266, FutureTask (java.util.concurrent)
20 createEntry:61, LoadingCache (org.springframework.cglib.core.internal)
21 get:34, LoadingCache (org.springframework.cglib.core.internal)
22 get:105, AbstractClassGenerator$ClassLoaderData (org.springframework.cglib.core)
23 create:278, AbstractClassGenerator (org.springframework.cglib.core)
24 createHelper:433, Enhancer (org.springframework.cglib.proxy)
25 createClass:338, Enhancer (org.springframework.cglib.proxy)
26 createProxyClassAndInstance:55, ObjenesisCglibAopProxy (org.springframework.aop.framework)
27 getProxy:203, CglibAopProxy (org.springframework.aop.framework)
28 getProxy:109, ProxyFactory (org.springframework.aop.framework)
29 createProxy:468, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
30 wrapIfNecessary:349, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
```

当我们proxyTargetClass为false，使用jdk动态代理时，执行代理对象的接口方法，代理对象本身也是接口的实现类对象， it instanceof SomeComponent 接口为ture，



接口对象的方法被拦截，将会执行 InvocationHandler的invoke方法，也就是org.springframework.aop.framework.JdkDynamicAopProxy#invoke



invoke方法接受的参数 Method 的签名是 class是接口， 方法是 abstract，

```
exception.
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwal
    MethodInvocation invocation;
    Object oldProxy = null;   oldProxy: null
```

在invoke方法中 调用了getInterceptorsAndDynamicInterceptionAdvice，

根据这个方法签名我们是可以找打TransactionalAttribute，因为在创建bean 的时候会执行wrapIfNecessary，这个方法内会 遍历每一个Advisor，针对每一个Advisor 遍历对象的接口中的每一个方法，然后根据Advisor中的Pointcut判断该Advisor是否匹配这个方法， 如果匹配那么就取出Advisor中的Advice，对于事务Advisor而言，其pointcut在每一个方法上寻找TransactionAttribute，并将方法和找到的TransactionAttribute缓存起来， 如果找不到就意味着没有事务。

```
for (Advisor candidate : candidateAdvisors) {   candidateAdviso
    if (candidate instanceof IntroductionAdvisor) {
        // already processed
        continue;
    }
    if (canApply(candidate, clazz, hasIntroductions)) {   clazz
        eligibleAdvisors.add(candidate);
    }
}
```

```
Set<Class<?>> classes = new LinkedHashSet<~>(ClassUtils.getAllInterfacesForClassAsSet(targetClass));   classes:  si
classes.add(targetClass);
for (Class<?> clazz : classes) {   classes:  size = 2    clazz: "interface example.SomeComponent"
    Method[] methods = clazz.getMethods();   clazz: "interface example.SomeComponent"    methods: Method[1]@15712
    for (Method method : methods) {   methods: Method[1]@15712    method: "public abstract void example.SomeCompone
        if ((introductionAwareMethodMatcher != null &&
                introductionAwareMethodMatcher.matches(method, targetClass, hasIntroductions)) ||   hasIntroductio
                methodMatcher.matches(method, targetClass)) {   targetClass: "class example.ComponentImpl"    metho
            return true;
        }
    }
}
```

然后在获取TransactionAttribute的过程中AbstractFallbackTransactionAttributeSource的computeTransactionAttribute

```
isor.java  AbstractAutoProxyCreator.java  AbstractFallbackTransactionAttributeSource.java  TransactionAttributeSourcePointcut.java
        else {
            return (TransactionAttribute)cached;   cached: null
        }
    }
    else {
        // We need to work it out.
        TransactionAttribute txAtt = computeTransactionAttribute(method, targetClass);   method: "public abs
        // Put it in the cache.
        if (txAtt == null) {
            this.attributeCache.put(cacheKey, NULL_TRANSACTION_ATTRIBUTE);
        }
```

如果传入的方法Method的签名是 接口中的方法（class为接口类，methodname为 abstract method）

```
method = {Method@14893} "public abstract void example.SomeComponent.initialize()"
clazz = {Class@14444} "interface example.SomeComponent" ... Navigate
```

computeTransactionAttribute方法中首先 获取targetClass，这个targetClass其实就是Bean对应的class，也就是实现类。

```
// Ignore CGLIB subclasses - introspect the actual user class.
Class<?> userClass = ClassUtils.getUserClass(targetClass);
```

然后 根据method获取在实现类上的实现方法specificMethod

```
// The method may be on an interface,
but we need attributes from the target class.
// If the target class is null, the method will be unchanged.
Method specificMethod = ClassUtils.
```

```
5  getMostSpecificMethod(method, userClass);
```

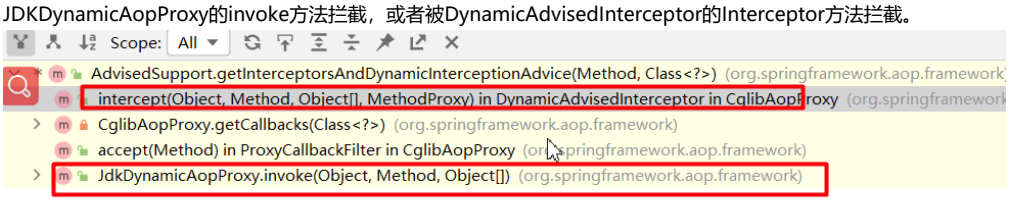
然后在实现类的方法上获取指定方法的TransactionAttribute，如果没有，就在实现类上获取TransactionAttribute，如果还没有，判断当前调用拦截到的方法method和SpecificMethod是否相同，如果不同则获取当前拦截到的方法method上的TransactionAttribute。当在接口的方法上进行注解@Transactional 且使用jdk代理的时候，当前调用的方法method 的签名信息class是接口，method是接口的abstract 方法， 因为jdk代理对象实现了接口；

```
// First try is the method in the target class.
TransactionAttribute txAtt = findTransactionAttribute(specificMethod);
if (txAtt != null) {
    return txAtt;
}

// Second try is the transaction attribute on the target class.
txAtt = findTransactionAttribute(specificMethod.getDeclaringClass());
if (txAtt != null) {
    return txAtt;
}

if (specificMethod != method) {
    // Fallback is to look at the original method.
    txAtt = findTransactionAttribute(method);
    if (txAtt != null) {
        return txAtt;
    }
    // Last fallback is the class of the original method.
    return findTransactionAttribute(method.getDeclaringClass());
}
```

当在接口方法上进行注解且proxyTargetClass为true使用cglib代理时，方法调用会被DynamicAdvisedInterceptor拦截到，其中method的签名是class为实现类对象的类名，方法为ComponentImpl的 public void initialize()非抽象方法。

```
private static class DynamicAdvisedInterceptor implements MethodInterceptor, Serializable {

    private final AdvisedSupport advised;

    public DynamicAdvisedInterceptor(AdvisedSupport advised) { this.advised = advised; }

    @Override
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        Object oldProxy = null;
        boolean setProxyContext = false;
```

总 的来说不管是cglibg动态代理还是 jdk动态代理，对于代理对象调用其方法的时候 会被InvocationHandler的invoke方法拦截，也就是JDKDynamicAopProxy的invoke方法拦截，或者被DynamicAdvisedInterceptor的Interceptor方法拦截。



在这两个拦截方法中都会调用 AdvisedSupport的getInterceptorsAndDynamicInterceptionAdvice方法 来构建一个调用链。

// Get the interception chain for this method.

List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

构建调用链的逻辑在org.springframework.aop.framework.DefaultAdvisorChainFactory#getInterceptorsAndDynamicInterceptionAdvice方法中，遍历每一个Advisor，用Advisor中的Pointcut 去匹配这个方法

```
// 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));
                }
            }
        }
    }
```

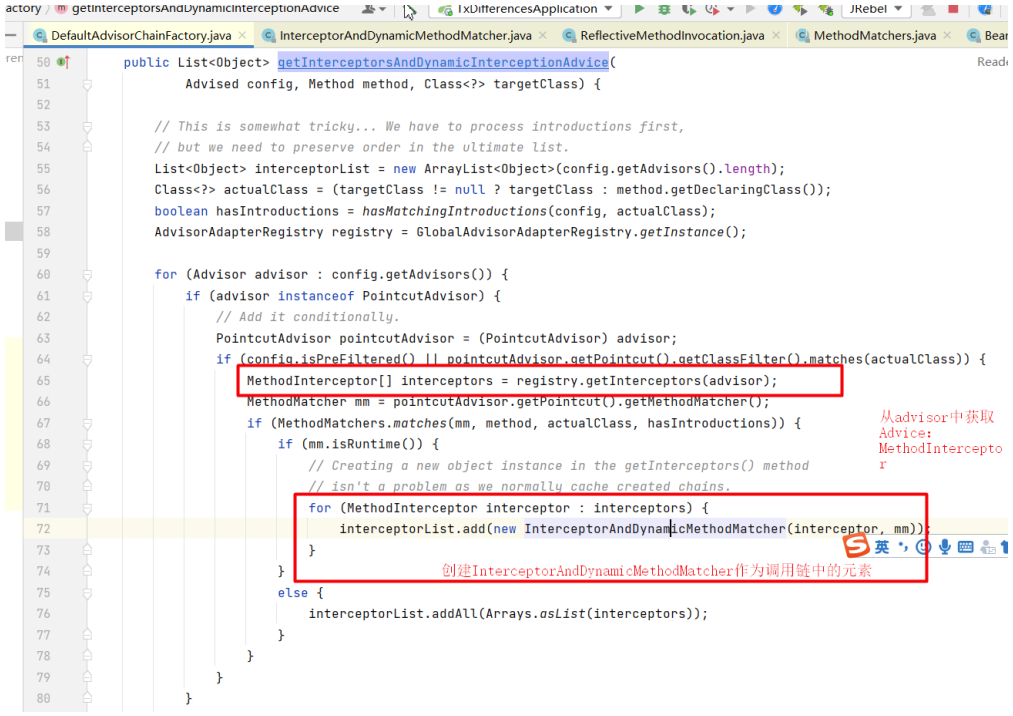
比如BeanFactoryTransactionAttributeSourceAdvisor 这个advisor，

MethodInterceptor[] interceptors = registry.getInterceptors(advisor);将会返回MethodInterceptor 其Pointcut实现match逻辑 会判断方法上是否存在TransactionAttribute属性

```
private final TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePointcut() {
    @Override
    protected TransactionAttributeSource getTransactionAttributeSource() { return transactionAttributeSource; }
};
```

```
/erial/
abstract class TransactionAttributeSourcePointcut extends StaticMethodMatcherPointcut implements Seria

    @Override
    public boolean matches(Method method, Class<?> targetClass) {
        if (TransactionalProxy.class.isAssignableFrom(targetClass)) {
            return false;
        }
        TransactionAttributeSource tas = getTransactionAttributeSource();
        return (tas == null || tas.getTransactionAttribute(method, targetClass) != null);
    }
```

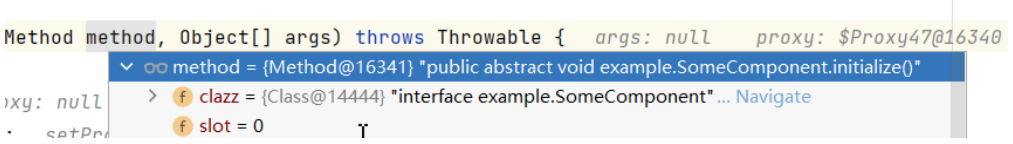
如果根据拦截的方法method对象，推断出这个method上存在注解@Transactional，则这个方法是一个事务方法。否则就是非事务方法，对于事务方法，会返回MethodInterceptor作为拦截器



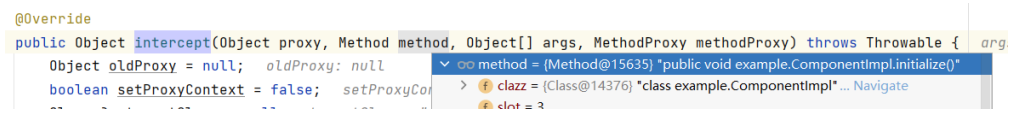
最终 InterceptorAndDynamicMethodMatcher在调用链对象ReflectMethodInvocation中如下被使用。

那么我们来说另外一个问题：

（1）jdk动态代理的 org.springframework.aop.framework.JdkDynamicAopProxy#invoke 方法参数method表示拦截到的方法，这个方法的签名信息是，class为接口，method为接口中的抽象放个发



（2）cglib代理 org.springframework.aop.framework.CglibAopProxy.DynamicAdvisedInterceptor#intercept 的方法的参数 method表示拦截到的方法，这个method的信息如下



另外一个问题，根据拦截到的method确定该方法上是否存在TransactionAttribute，从而判断法方法是否是事务方法，这个逻辑是在哪里实现的？

org.springframework.transaction.interceptor.AbstractFallbackTransactionAttributeSource#getTransactionAttribute方法中会调用computeTransactionAttribute方法

org.springframework.transaction.interceptor.AbstractFallbackTransactionAttributeSource#computeTransactionAttribute