关于对Spring事务管理源码的简单分析

一、Spring 事务实现机制

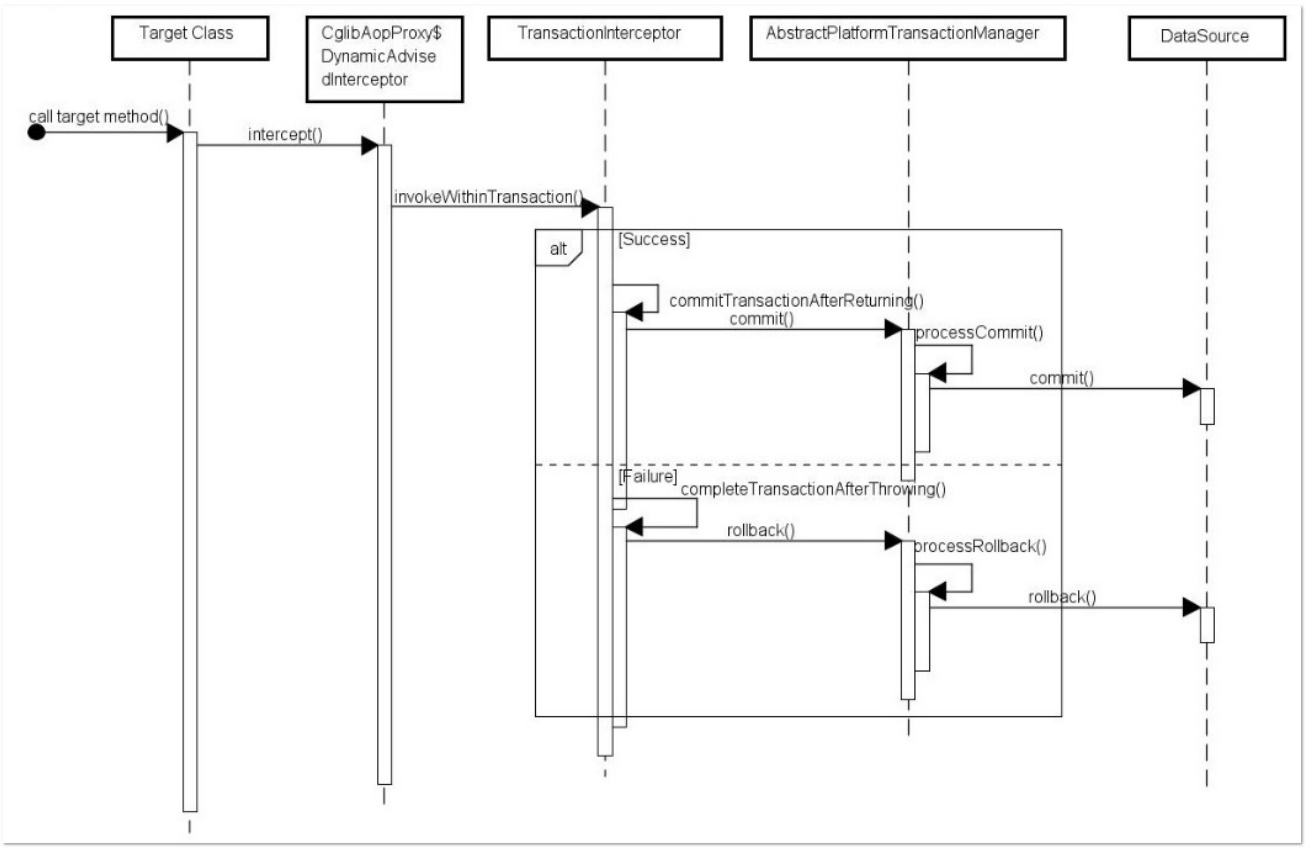
流程：

1、在调用声明@Transactional 的目标方法时，Spring Framework 默认使用 `AOP` 代理，在代码运行时生成一个代理对象；

2、根据@Transactional 的属性配置信息，这个代理对象决定该方法是否由 `TransactionInterceptor` 来使用拦截；

3、在 `TransactionInterceptor` 拦截时，会在目标方法开始执行前创建并开启事务，然后执行目标方法的逻辑，最后根据执行情况是否出现异常来调用 `抽象事物管理器AbstractPlatformTransactionManager`；

4、 `AbstractPlatformTransactionManager` 操作数据源DataSource 提交或回滚事务。



二、源码分析

1、Cglib2AopProxy.java

```
private static class DynamicAdvisedInterceptor implements MethodInterceptor,
Serializable {
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
```

```
        ...
        List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);//这个方法
会根据注解对应的方法、类等条件进行匹配，会获取到事务拦截TransactionInterceptor（这里有属性
transactionManagerBeanName，是我们自己在xml配置的bean）
        ...
         //判断方法是否为public，如果不是，则事务管理注解失效。
        if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
        }
        ...
        //重点来了..
       retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass,
chain, methodProxy).proceed();
        //##以下为对CglibMethodInvocation()的分析##
        1、首先需要知道的是，CglibMethodInvocatio类继承反射类ReflectiveMethodInvocation，
        private static class CglibMethodInvocation extends ReflectiveMethodInvocation
        相当于调用反射类ReflectiveMethodInvocation
        2、继续看ReflectiveMethodInvocation
        public class ReflectiveMethodInvocation implements ProxyMethodInvocation,
Cloneable
        可以知道ReflectiveMethodInvocation实现动态代理接口
        3、继续往上走，可以知道上面的类或接口都基于public interface Joinpoint，
            而Joinpoint是Spring aop的连接点。
        4、于是我们知道了，CglibMethodInvocation类在此处的作用是：根据动态代理，去反射被注解了
@Transactional的方法对。
         综上所述：
         于是，我们大胆猜测，CglibMethodInvocation类的proceed()方法的作用：
            以AOP切面的方式去控制一组事务。即在执行目标方法前开启事务，执行完后提交或回滚事务。

    }
}
```

此时，我们知道关键点在 ReflectiveMethodInvocation类中的proceed()方法，如果知道了其是如何实现的话，那就知道了其注解方式的事务管理的原理。

于是，看看proceed()。

2、ReflectiveMethodInvocation.proceed()

```
public Object proceed() throws Throwable {
    //首先，获取传过来的拦截器对象TransactionInterceptor。
    Object interceptorOrInterceptionAdvice =

 this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
    currentInterceptorIndex初始值被定义为-1;
    当配置了多个事务管理器时，如果不指定名称，按配置的顺序有先。
    ...
     //然后，执行拦截器
     return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
}
```

于是，我们就去看拦截器TransactionInterceptor的invoke()方法干了些什么。

3、TransactionInterceptor.invoke()

```
public Object invoke(final MethodInvocation invocation) throws Throwable {
    //首先，获取注解了@Transaction传入的属性信息对象TransactionAttribute。如：回滚规则，超时时间等
   final TransactionAttribute txAttr =
    getTransactionAttributeSource().getTransactionAttribute(invocation.getMethod(), targetClass);
    ...
    //然后，根据txAttr定义事务管理器对象PlatformTransactionManager
     final PlatformTransactionManager tm = determineTransactionManager(txAttr);
     ...
    //接下来，开始执行事务管理相关逻辑，几乎和以前手撸事物管理一样，如下：
     if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
        //开启事务
         TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
         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.
             //执行目标方法的逻辑
             retVal = invocation.proceed();
         }
         catch (Throwable ex) {
             //回滚操作，封装了rollback()
             completeTransactionAfterThrowing(txInfo, ex);
             throw ex;
         }
         finally {
             //实质是恢复事务管理TransactionInfo到之前的状态 () ? ? ? ? ?
             cleanupTransactionInfo(txInfo);
         }
         commitTransactionAfterReturning(txInfo);//提交事务
         return retVal;
}
```

上面写法是不是感觉很熟悉，对的，和我们自己以前手写事务管理很相似,即开启事务、业务逻辑、提交/回滚.

至此，Spring事物实现机制大体叙述完了。对Spring事物实现机制应该有了更加清晰的认识了。

此时，如果有空，我们可以看看的是try里面的逻辑----invocation.proceed()。当然，这部分属于反射的知识。

在invocation.proceed();中，invocation指的是之前在第2点说的ReflectiveMethodInvocation.proceed()，此处再次调用它时，会执行以下代码：

```
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
            return invokeJoinpoint();
        }
```

之前提到，currentInterceptorIndex初始为-1，所以当第一次进来时不会执行invokeJoinpoint()，而执行第2点所说的逻辑。此时，我们从之前的分析中能知道invokeJoinpoint()的作用，即执行目标方法的逻辑。

于是，我们看看invokeJoinpoint()到底是什么。

```
        protected Object invokeJoinpoint() throws Throwable {
```

```
            return AopUtils.invokeJoinpointUsingReflection(this.target, this.method,
this.arguments);
        }
```

而 AopUtils工具类中的invokeJoinpointUsingReflection()如下：

```
        public static Object invokeJoinpointUsingReflection(){
            // Use reflection to invoke the method.
            try {
                ReflectionUtils.makeAccessible(method);
                return method.invoke(target, args);
            }
        }
```

于是，终于知道搞了半天，原来只是回来执行目标方法而已。一直以为会有环绕通知等。