通过源码深入理解 Spring 事务的实现原理

你好，首先感谢你能订阅这篇文章，可能篇幅有点长，还希望你能耐心读下去，相信我不会让你失望的。下面我们来开启 Spring 事务的学习之旅吧。

通过源码深入理解 Spring 事务的实现原理
一. 前言
1.1 事务配置
二. 事务代理类的创建
2.1 解析 tx 自定义标签
2.2 事务代理类的创建
2.2.1 取得bean对应的拦截器
2.2.2 创建Proxy代理类
2.2.3 Proxy代理类的执行
三. 创建事务
3.1 doGetTransaction
3.2 handleExistingTransaction
3.2.1 PROPAGATION_NEVER
3.2.2 PROPAGATION_NOT_SUPPORTED
3.2.3 PROPAGATION_REQUIRES_NEW
3.2.4 PROPAGATION_NESTED
3.2.5 PROPAGATION_REQUIRED
3.3 创建新事务
四. 提交事务
4.1 processRollback
4.2 processCommit
4.2.1 cleanupAfterCompletion
五. 回滚事务
5.1 是否回滚
5.2 事务回滚
六. Spring事务中的设计模式
七. 总结

一. 前言

1.1 事务配置

```
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionM
anager">
        <property name="dataSource" ref="partition1" />
    </bean>

    <tx:annotation-driven transaction-
manager="transactionManager" />
```

spring使用的版本为3.2.6.RELEASE。

二. 事务代理类的创建

2.1 解析 tx 自定义标签

spring通过TxNamespaceHandler来解析事务标签，TxNamespaceHandler的init方法如下：


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

也就是说spring通过AnnotationDrivenBeanDefinitionParser来解析annotation-driven，解析的时候会区分mode是aspectj还是proxy，因为我们没有配置，mode为默认的proxy，parse过程如下：

（1）注册InfrastructureAdvisorAutoProxyCreator

InfrastructureAdvisorAutoProxyCreator实现了InstantiationAwareBeanPostProcessor接口，在创建bean后会调用postProcessAfterInstantiation方法，在此方法会根据需要创建AopProxy代理类。

（2）注册TransactionAttributeSource

注册了AnnotationTransactionAttributeSource BeanDefinition。

（3）注册TransactionInterceptor

事务拦截器，在事务代理类执行时会调用invoke方法。TransactionInterceptor依赖TransactionAttributeSource，同时实现MethodInterceptor接口，而Interceptor接口又继承了接口Advice，也就是说TransactionInterceptor为一种Advice通知。

（4）TransactionAttributeSourceAdvisor

注册BeanFactoryTransactionAttributeSourceAdvisor通知器，依赖TransactionAttributeSource，TransactionInterceptor。

2.2 事务代理类的创建

在创建bean之后的initializeBean方法中，会执行如下代码：

```
    if (mbd == null || !mbd.isSynthetic()) {
            wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
    }
```

applyBeanPostProcessorsAfterInitialization方法实现如下：

```
    public Object
applyBeanPostProcessorsAfterInitialization(Object existingBean,
String beanName)
            throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor beanProcessor :
getBeanPostProcessors()) {
            result =
beanProcessor.postProcessAfterInitialization(result, beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```


也就是说此时会执行beanProcessor的postProcessAfterInitialization，而我们在解析事务自定义标签时，正好注册了InfrastructureAdvisorAutoProxyCreator，此bean的postProcessAfterInitialization方法如下：

```
    public Object postProcessAfterInitialization(Object bean,
String beanName) throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(),
beanName);
            if (!this.earlyProxyReferences.containsKey(cacheKey))
```

```
{
                    //如果需要的话创建代理类
                    return wrapIfNecessary(bean, beanName, cacheKey);
                }
            }
            return bean;
        }
```

我们来理解下wrapIfNecessary，如果需要的话创建代理类，也就是说有可能会创建代理类，也有可能不会创建代理类，那么什么情况下会创建代理类，什么情况下又不会创建代理类呢，我们继续来看下此方法的实现：

```
        // 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;
        }
```


此方法主要包括2个核心点：取得bean对应的拦截器，创建代理类。

2.2.1 取得bean对应的拦截器

方法入口为AbstractAdvisorAutoProxyCreator.getAdvicesAndAdvisorsForBean，在此方法中会调用findEligibleAdvisor查找合适的Advisor。

```
        protected List<Advisor> findEligibleAdvisors(Class beanClass,
    String beanName) {
            //查找所有候选的Advisors，获取BeanFactory中所有Advisor.class
    的bean
            List<Advisor> candidateAdvisors =
    findCandidateAdvisors();
            List<Advisor> eligibleAdvisors =
    findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);
            extendAdvisors(eligibleAdvisors);
            if (!eligibleAdvisors.isEmpty()) {
                eligibleAdvisors = sortAdvisors(eligibleAdvisors);
            }
            return eligibleAdvisors;
        }
```

此方法的实现过程为：先从BeanFactory查找所有类型为Advisor.class的bean，然后判断这些Advisor是否可以应用到当前bean上。我们着重来看下findAdvisorsThatCanApply方法

的实现，通过代码层层跟踪可以看到如下代码：

```
    public static boolean canApply(Advisor advisor, Class<?>
targetClass, boolean hasIntroductions) {
        if (advisor instanceof IntroductionAdvisor) {
            return ((IntroductionAdvisor)
advisor).getClassFilter().matches(targetClass);
        }
        else if (advisor instanceof PointcutAdvisor) {
            PointcutAdvisor pca = (PointcutAdvisor) advisor;
            return canApply(pca.getPointcut(), targetClass,
hasIntroductions);
        }
        else {
            // It doesn't have a pointcut so we assume it
applies.
            return true;
        }
    }
```

因为我们在前面注册的BeanFactoryTransactionAttributeSourceAdvisor为PointcutAdvisor，所以会通过PointCut来判断Advisor是否可以应用到目标类中，如果可以应用的话那么则会创建代理类，否则则不创建代理类（在没有其它满足条件的Advisor的前提下）。canApply方法核心实现如下：


```
    //取得目标类所有继承的类和实现的接口
    Set<Class> classes = new HashSet<Class>
(ClassUtils.getAllInterfacesForClassAsSet(targetClass));
        //添加目标类
        classes.add(targetClass);
        for (Class<?> clazz : classes) {
            //取得类下的所有方法，不包括private，protected
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                if ((introductionAwareMethodMatcher != null &&

introductionAwareMethodMatcher.matches(method, targetClass,
hasIntroductions)) ||
                        methodMatcher.matches(method,
targetClass)) {
                    return true;
                }
            }
        }
```

此时会遍历目标类所有继承的类和实现的接口中的所有方法，只要有一个方法可以和methodMatcher匹配成功，那么就认为此Advisor可以应用到此bean。TransactionAttributeSourcePointcut匹配方法如下：

```
        public boolean matches(Method method, Class targetClass) {
            //为解析自定义标签时注册的
    AnnotationTransactionAttributeSource
            TransactionAttributeSource tas =
    getTransactionAttributeSource();
            return (tas == null ||
    tas.getTransactionAttribute(method, targetClass) != null);
        }
```

通过TransactionAttributeSource 来获取事务属性，在getTransactionAttribute方法中会调用computeTransactionAttribute来计算出事务属性，过程如下：

```
            // 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;
            }
```


（1）获取目标类方法上的事务属性

```
        protected TransactionAttribute
    determineTransactionAttribute(AnnotatedElement ae) {
            for (TransactionAnnotationParser annotationParser :
    this.annotationParsers) {
                TransactionAttribute attr =
    annotationParser.parseTransactionAnnotation(ae);
                if (attr != null) {
                    return attr;
                }
            }
            return null;
        }
```

通过SpringTransactionAnnotationParser来解析方法上的事务属性：

```
        public TransactionAttribute
    parseTransactionAnnotation(AnnotatedElement ae) {
            //判断方法上是否存在Transactional注解
```

```
        Transactional ann = AnnotationUtils.getAnnotation(ae,
Transactional.class);
        if (ann != null) {
            //如果存在的话解析Transactional注解
            return parseTransactionAnnotation(ann);
        }
        else {
            return null;
        }
    }
```

如果存在Transactional注解的话，会解析出propagation，isolation，timeout，readOnly，rollbackFor，rollbackForClassName，noRollbackFor，noRollbackForClassName这些属性，然后创建RuleBasedTransactionAttribute对象返回。

（2）获取目标类上的事务属性

过程同上，只是此时会获取目标类上的Transactional注解。

从上面的代码中可以看出：只要方法或类上存在Transactional注解（可以是目标类、目标类的方法，也可以是目标类父类、父类的方法上存在Transactional注解，但是方法不能是private，protected，必须是public），那么就认为BeanFactoryTransactionAttributeSourceAdvisor可以应用到bean，也就可以创建代理类了。

2.2.2 创建Proxy代理类

调用ProxyFactory.getProxy来创建代理类，实现如下：

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

（1）createAopProxy

```
    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.");
            }
            //如果目标代理类为接口，则创建JdkDynamicAopProxy
```

```
            if (targetClass.isInterface()) {
                return new JdkDynamicAopProxy(config);
            }
            //目标代理类不为接口的话，创建CglibAopProxy
            return CglibProxyFactory.createCglibProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
```

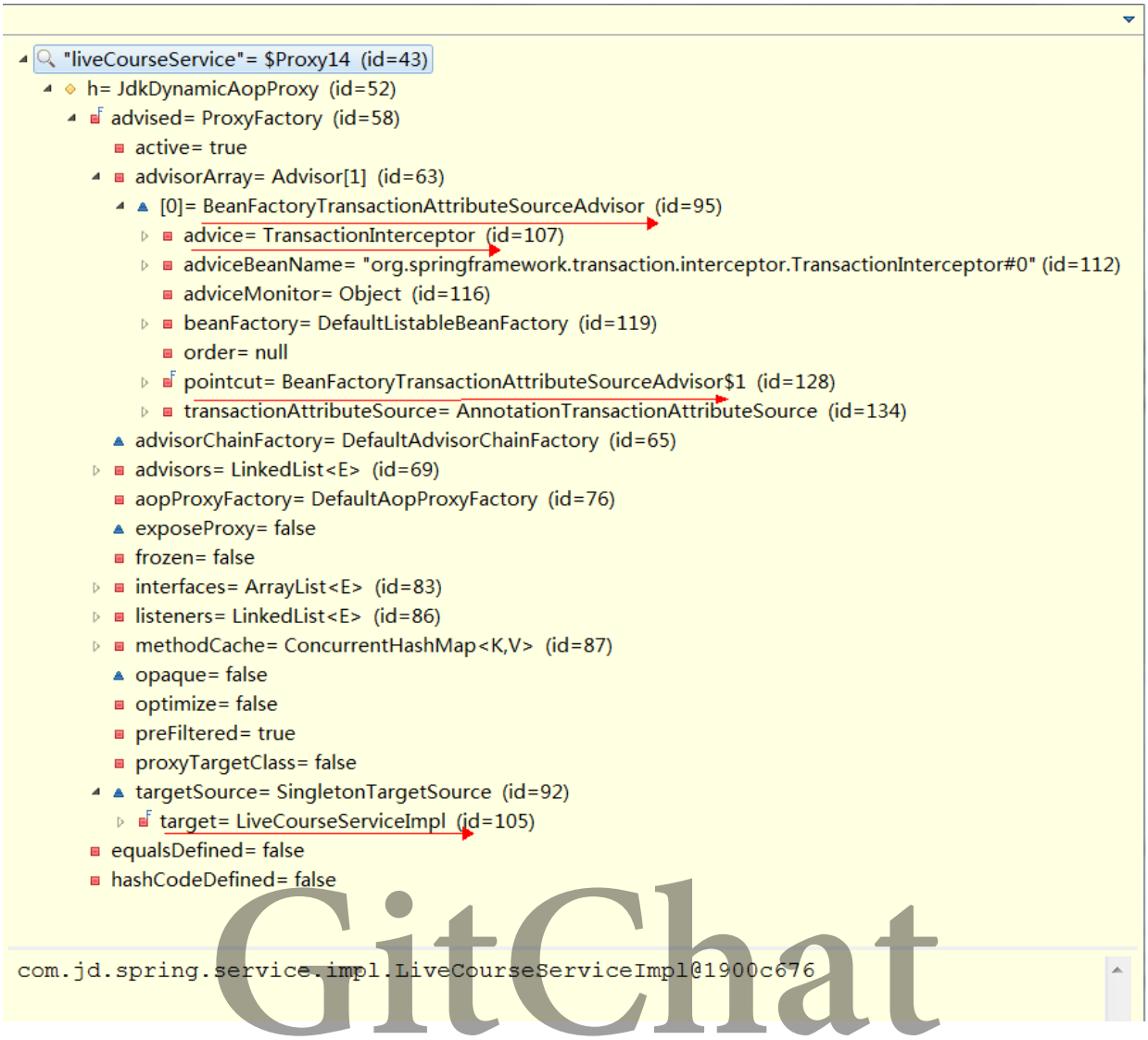
optimize是否使用优化策略，默认false；
proxyTargetClass是否代理目标类,而不是目标类实现的接口，默认false；
hasNoUserSuppliedProxyInterfaces 未配置proxyInterfaces

（2）getProxy

我们以JdkDynamicAopProxy为例来看下获取代理类的实现，

```
    //获取代理的接口类
    Class[] proxiedInterfaces =
AopProxyUtils.completeProxiedInterfaces(this.advised);
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces,
this);
```


最终创建的代理类如下：



可以看到事务代理类对应的Advisor为BeanFactoryTransactionAttributeSourceAdvisor，Advice为TransactionInterceptor，pointCut为BeanFactoryTransactionAttributeSourceAdvisor下的TransactionAttributeSourcePointcut，target为LiveCourseServiceImpl。

2.2.3 Proxy代理类的执行

下面我们来看下JdkDynamicAopProxy代理类的执行，因为JdkDynamicAopProxy实现InvocationHandler方法，所以入口为其invoke方法，核心代码如下：

```
        // Get the interception chain for this method.
        List<Object> chain =
    this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
    targetClass);
        if (chain.isEmpty()) {
            retVal = AopUtils.invokeJoinpointUsingReflection(target,
    method, args);
        }
        else {
            invocation = new ReflectiveMethodInvocation(proxy,
    target, method, args, targetClass, chain);
            // Proceed to the joinpoint through the interceptor
```

```
chain.
            retVal = invocation.proceed();
        }
```

取得method对应的拦截器链，如果存在的话那么执行拦截器链的invoke方法，否则的话直接执行目标类的方法。事务对应的拦截器为TransactionInterceptor，下面我们来看下TransactionInterceptor的Invoke方法：

```
    public Object invoke(final MethodInvocation invocation)
throws Throwable {
        Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass(invocation.getThis()) : null);
        // Adapt to TransactionAspectSupport's
invokeWithinTransaction...
        return invokeWithinTransaction(invocation.getMethod(),
targetClass, new InvocationCallback() {
            public Object proceedWithInvocation() throws
Throwable {
                return invocation.proceed();
            }
        });
    }
```


invokeWithinTransaction为事务执行的关键方法，重中之中，我把核心代码贴出来，咱们逐一分析下：

```
    // 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);

        if (txAttr == null || !(tm instanceof
CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with
getTransaction and commit/rollback calls.
            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.
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }
```

上面只是我列出了invokeWithinTransaction方法的一个if分支，还有一个分支是判断如果事务管理器为CallbackPreferringPlatformTransactionManager，那么执行另外一段逻辑，因为我们配置的为DataSourceTransactionManager，所以我们就不再分析另外逻辑分支了，感兴趣的可以自己看下。

上面的代码描述了执行事务的大体框架，步骤如下：

（1）获取事务属性

获取事务属性，我们在判断方法或类上是否存在Transactional注解时，已经创建了事务属性RuleBasedTransactionAttribute。

（2）获取事务管理器

为配置文件中配置的DataSourceTransactionManager。

（3）methodIdentification

类名.方法名，如com.jd.spring.service.impl.LiveCourseServiceImpl.insertLiveCourseAndUser

（4）createTransactionIfNecessary

在后面”创建事务”这一章节分析。

（5）目标类执行

通过反射执行目标类的目标方法。

（6）completeTransactionAfterThrowing

当目标方法抛出异常时，执行此方法，有可能会回滚事务，也有可能不会回滚事务，需要看下是否新事务和抛出的异常类型，具体实现参见后面章节”回滚事务”。

（7）cleanupTransactionInfo

无论目标方法执行成功与否，执行完成后需要清除事务信息。

```
transactionInfoHolder.set(this.oldTransactionInfo);
```

（8）commitTransactionAfterReturning

提交事务，具体实现参见后面章节”提交事务”。

三. 创建事务

下面我们来看下创建事务的方法createTransactionIfNecessary，此方法的返回值为TransactionInfo，核心代码如下：

```
        TransactionStatus status = null;
        if (txAttr != null) {
            if (tm != null) {
                //获取TransactionStatus，很关键
                status = tm.getTransaction(txAttr);
            }
        }
        return prepareTransactionInfo(tm, txAttr,
joinpointIdentification, status);
```

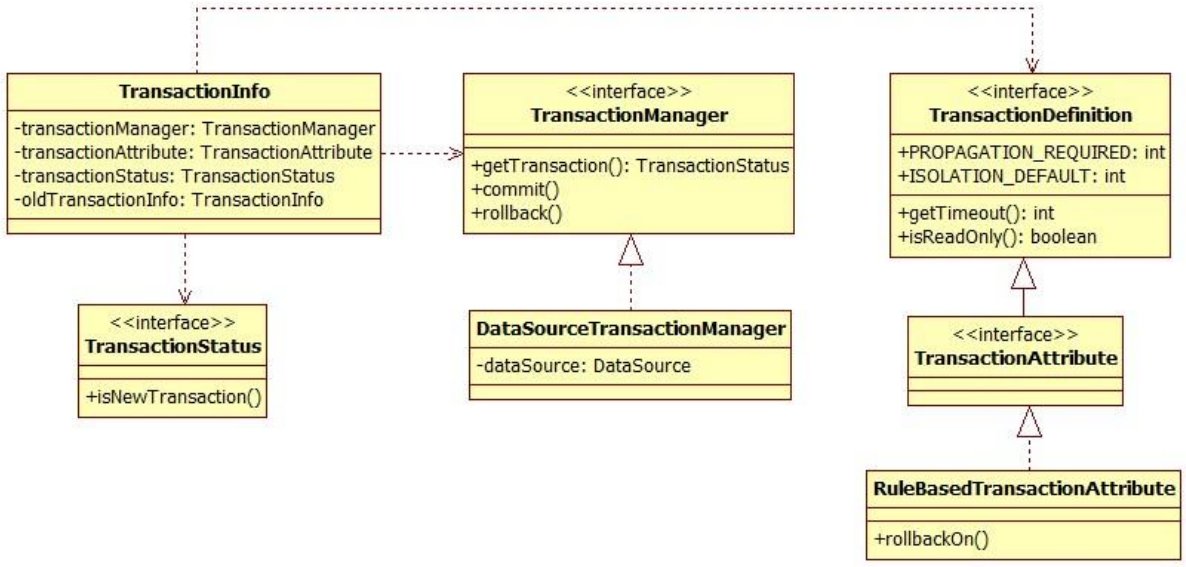

首先获取TransactionStatus，然后创建TransactionInfo对象，并将TransactionInfo对象绑定到当前线程。

```
      protected TransactionInfo
prepareTransactionInfo(PlatformTransactionManager tm,
            TransactionAttribute txAttr, String
joinpointIdentification, TransactionStatus status) {
        //创建TransactionInfo
        TransactionInfo txInfo = new TransactionInfo(tm, txAttr,
joinpointIdentification);
        if (txAttr != null) {
            //设置TransactionStatus
            txInfo.newTransactionStatus(status);
        }
        //绑定到当前线程
        txInfo.bindToThread();
        return txInfo;
    }

        private void bindToThread() {
            //保存旧的事务信息，此处为备忘录模式，当时候再还原回去
            this.oldTransactionInfo =
transactionInfoHolder.get();
```

```
            //通过ThreadLocal保存当前事务信息，
ThreadLocal<TransactionInfo>
            transactionInfoHolder.set(this);
        }
```

上面这一小部分分析了TransactionInfo的创建，TranactioInfo和TransactionStatus关系如下图所示：



下面我们来着重讲下getTransaction。

3.1 doGetTransaction

获取事务对象，方法如下：

```
    protected Object doGetTransaction() {
        DataSourceTransactionObject txObject = new
DataSourceTransactionObject();
        //设置是否允许嵌套事务，默认为false，
DataSourceTransactionManager为true

txObject.setSavepointAllowed(isNestedTransactionAllowed());
        //从当前线程中获取ConnectionHolder，ConnectionHolder依赖
Connection,存在transactionActive属性，如果之前创建了事务的话，那么此时
conHolder不为null
        ConnectionHolder conHolder =
            (ConnectionHolder)
TransactionSynchronizationManager.getResource(this.dataSource);
        txObject.setConnectionHolder(conHolder, false);
        return txObject;
    }
```

当第一次创建事务对象时，会获取connection连接，然后创建ConnectionHolder对象，并放入到ThreadLocal当前线程中。

3.2 handleExistingTransaction

如果之前已经创建了事务对象，那么将根据不同的传播属性来决定是否创建事务。判断是否已经存在事务的条件为：
txObject.getConnectionHolder() != null && txObject.getConnectionHolder().isTransactionActive()

如果已经存在了事务，handleExistingTransaction处理逻辑如下：

3.2.1 PROPAGATION_NEVER

如果传播属性为PROPAGATION_NEVER，不允许存在事务，抛出异常。

3.2.2 PROPAGATION_NOT_SUPPORTED

如果传播属性为 PROPAGATION_NOT_SUPPORTED ，则挂起当前事务，不开启新的事务，并且创建DefaultTransactionStatus对象返回。


```
    Object suspendedResources = suspend(transaction);
            boolean newSynchronization =
(getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
            return prepareTransactionStatus(
                    definition, null, false, newSynchronization,
debugEnabled, suspendedResources);
```

（1）suspend挂起当前事务

返回值为SuspendedResourcesHolder对象。

```
    protected Object doSuspend(Object transaction) {
        DataSourceTransactionObject txObject =
(DataSourceTransactionObject) transaction;
        //将connectionHolder设置为null
        txObject.setConnectionHolder(null);
        ConnectionHolder conHolder = (ConnectionHolder)

TransactionSynchronizationManager.unbindResource(this.dataSource)
;
        return conHolder;
    }
```

清空当前线程绑定的ConnectionHolder，然后取得当前线程的readOnly，isolationLevel，wasActive，创建SuspendedResourcesHolder返回。

```
        return new SuspendedResourcesHolder(
                suspendedResources, suspendedSynchronizations, name,
    readOnly, isolationLevel, wasActive);
```

（2）prepareTransactionStatus

```
        //创建DefaultTransactionStatus，只有是 newSynchronization &&
    !TransactionSynchronizationManager.isSynchronizationActive();才会
    同步保存事务信息
        //newSynchronization默认为SYNCHRONIZATION_ALWAYS
        DefaultTransactionStatus status = newTransactionStatus(
                    definition, transaction, newTransaction,
    newSynchronization, debug, suspendedResources);

        //只有DefaultTransactionStatus.newSynchronization=true时才会将
    事务状态信息保存到TransactionSynchronizationManager
        prepareSynchronization(status, definition);
```

通过TransactionSynchronizationManager保存当前线程的事务状态、事务定义信息。事务同步管理器通过ThreadLocal来保存当前的事务状态、事务定义信息。


```
        protected void
    prepareSynchronization(DefaultTransactionStatus status,
    TransactionDefinition definition) {
            if (status.isNewSynchronization()) {

    TransactionSynchronizationManager.setActualTransactionActive(stat
    us.hasTransaction());

    TransactionSynchronizationManager.setCurrentTransactionIsolationL
    evel(
                        (definition.getIsolationLevel() !=
    TransactionDefinition.ISOLATION_DEFAULT) ?
                                definition.getIsolationLevel() :
    null);

    TransactionSynchronizationManager.setCurrentTransactionReadOnly(d
    efinition.isReadOnly());

    TransactionSynchronizationManager.setCurrentTransactionName(defin
    ition.getName());

    TransactionSynchronizationManager.initSynchronization();
            }
        }
```

在status.isNewSynchronization时才会同步保存事务状态，事务定义信息，同理在事务提交、回滚时，只有是isNewSynchronization才会进行事务信息的清除和还原。

3.2.3 PROPAGATION_REQUIRES_NEW

如果传播属性为PROPAGATION_REQUIRES_NEW，则每次都创建事务，则挂起当前事务（挂起当前事务同上，主要是创建了SuspendedResourcesHolder对象），开启新事务。

```
    SuspendedResourcesHolder suspendedResources =
suspend(transaction);
    try {
        boolean newSynchronization =
(getTransactionSynchronization() != SYNCHRONIZATION_NEVER);

        //创建新的TransactionStatus对象，在挂起之前线程的时候将
TransactionSynchronizationManager的active属性设置为false，所以此时会
同步新的事务信息
        DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true,
newSynchronization, debugEnabled, suspendedResources);

        //开启事务
        doBegin(transaction, definition);
        //同步事务信息，同上，此时status.isNewSynchronization()为true
        prepareSynchronization(status, definition);
        return status;
    }
```


doBegin交由不同的事务管理器来进行事务的开启，此处是典型的模板方法。doBegin方法实现如下：

```
    protected void doBegin(Object transaction,
TransactionDefinition definition) {
        DataSourceTransactionObject txObject =
(DataSourceTransactionObject) transaction;
        Connection con = null;
        try {
            //如果事务对象对应的connectionHolder为null，首次创建事务或
者之前创建事务，事务传播属性为REQUIRED_NEW，将之前事务挂起，
            //这两种情况connectionHolder都为null
            if (txObject.getConnectionHolder() == null ||

txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                //从数据源中获取连接，然后设置事务对象的
ConnectionHolder
                Connection newCon =
this.dataSource.getConnection();
                //true标明为新的connectionHolder
                txObject.setConnectionHolder(new
```

```
ConnectionHolder(newCon), true);
            }

txObject.getConnectionHolder().setSynchronizedWithTransaction(tru
e);
            con = txObject.getConnectionHolder().getConnection();

            Integer previousIsolationLevel =
DataSourceUtils.prepareConnectionForTransaction(con, definition);
txObject.setPreviousIsolationLevel(previousIsolationLevel);

            //设置自动提交为false
            if (con.getAutoCommit()) {
                txObject.setMustRestoreAutoCommit(true);
                con.setAutoCommit(false);
            }
            //设置事务active为true，此属性用来判断是否已经存在事务

txObject.getConnectionHolder().setTransactionActive(true);

            int timeout = determineTimeout(definition);
            if (timeout != TransactionDefinition.TIMEOUT_DEFAULT)
{

txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
            }

            // txObject记录newConnectionHolder属性
            if (txObject.isNewConnectionHolder()) {
                //绑定connectionHolder到当前线程

TransactionSynchronizationManager.bindResource(getDataSource(),
txObject.getConnectionHolder());
            }
        }
        catch (Throwable ex) {
            //如果创建事务失败的话，释放连接 connection.close
            DataSourceUtils.releaseConnection(con,
this.dataSource);
            throw new CannotCreateTransactionException("Could not
open JDBC Connection for transaction", ex);
        }
    }
```

在新创建事务时会获取数据库连接，然后绑定到当前线程，同理当提交事务、回滚事务时，只有为txObject.isNewConnectionHolder()新创建的事务时，才会关闭数据库连接

3.2.4 PROPAGATION_NESTED

（1）如果事务管理器TransactionManager不支持嵌套事务，那么抛出异常。

（2）如果事务管理器使用保存点来支持嵌套事务，那么创建保存点，实现如下：

```
        if (useSavepointForNestedTransaction()) {
            //同上
            DefaultTransactionStatus status =
                    prepareTransactionStatus(definition, transaction,
    false, false, debugEnabled, null);
            status.createAndHoldSavepoint();
            return status;
        }
```

创建保存点最终还是通过 getConnection().setSavepoint(SAVEPOINT_NAME_PREFIX + this.savepointCounter); //SAVEPOINT_

（3）如果事务管理器不使用保存点（如）来支持嵌套事务，那么开启新事务，过程同上。

3.2.5 PROPAGATION_REQUIRED

```
        boolean newSynchronization = (getTransactionSynchronization()
    != SYNCHRONIZATION_NEVER);
        return prepareTransactionStatus(definition, transaction,
    false, newSynchronization, debugEnabled, null);
```


使用已经存在的事务，不开启新事务。在事务提交时也不会提交事务，只有在外层事务提交时才会提交事务

3.3 创建新事务

如果之前没有创建事务，并且事务传播属性是 PROPAGATION_REQUIRED ，PROPAGATION_REQUIRES_NEW ，PROPAGATION_NESTED ，那么创建新的事务。

```
        boolean newSynchronization = (getTransactionSynchronization()
    != SYNCHRONIZATION_NEVER);
        DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true, newSynchronization,
    debugEnabled, suspendedResources);
        //开启事务，同上
        doBegin(transaction, definition);
        //同步事务状态、事务定义信息，同上
        prepareSynchronization(status, definition);
```

四. 提交事务

上面花费了不少篇幅把createTransactionIfNecessary创建事务这一部分讲完了，下面就是目标方法执行和事务提交。目标方法执行稍微简单，通过反射执行目标方法即可，下面我们着重来看下事务的提交commitTransactionAfterReturning。

```
    public final void commit(TransactionStatus status) throws
TransactionException {
        if (status.isCompleted()) {
            throw new IllegalTransactionStateException(
                    "Transaction is already completed - do not
call commit or rollback more than once per transaction");
        }

        DefaultTransactionStatus defStatus =
(DefaultTransactionStatus) status;

        //isLocalRollbackOnly默认false
        if (defStatus.isLocalRollbackOnly()) {
            processRollback(defStatus);
            return;
        }
        //shouldCommitOnGlobalRollbackOnly默认为false
        //isGlobalRollbackOnly判断条件为
ConnectionHolder.rollbackOnly，在抛出异常不提交事务时设置
        if (!shouldCommitOnGlobalRollbackOnly() &&
defStatus.isGlobalRollbackOnly()) {
            processRollback(defStatus);

            //如果是新事务，抛出Transaction rolled back because it
has been marked as rollback-only异常
            if (status.isNewTransaction() ||
isFailEarlyOnGlobalRollbackOnly()) {
                throw new UnexpectedRollbackException(
                        "Transaction rolled back because it has
been marked as rollback-only");
            }
            return;
        }

        processCommit(defStatus);
    }
```


4.1 processRollback

进行事务回滚处理，具体实现见下面章节”回滚事务”。

4.2 processCommit

```
        prepareForCommit(status);
        triggerBeforeCommit(status);
        triggerBeforeCompletion(status);
        beforeCompletionInvoked = true;
        boolean globalRollbackOnly = false;

        //如果是新事务或者isFailEarlyOnGlobalRollbackOnly提早失败，
        if (status.isNewTransaction() ||
    isFailEarlyOnGlobalRollbackOnly()) {
            globalRollbackOnly = status.isGlobalRollbackOnly();
        }
        if (status.hasSavepoint()) {
            status.releaseHeldSavepoint();
        }
        else if (status.isNewTransaction()) {
            if (status.isDebug()) {
                logger.debug("Initiating transaction commit");
            }
            doCommit(status);
        }
        // Throw UnexpectedRollbackException if we have a global
    rollback-only
        // marker but still didn't get a corresponding exception from
    commit.
        if (globalRollbackOnly) {
            throw new UnexpectedRollbackException(
                    "Transaction silently rolled back because it has
    been marked as rollback-only");
        }

        ……
        //提交后清除事务同步信息，事务状态，事务定义
        cleanupAfterCompletion(status);
```

从上面的代码可以看出：

（1）如果有保存点的话释放保存点，即嵌套事务的提交是释放保存点

（2）如果是新事务的话，那么提交事务

```
        Connection con =
    txObject.getConnectionHolder().getConnection();
        con.commit();
```

也就是说只有是新事务的话，才会提交事务。也就是说如果之前已经存在事务，并且传播行为为 PROPAGATION_NOT_SUPPORTED ， PROPAGATION_REQUIRED ，那么此时是不会进行事务提交的。

4.2.1 cleanupAfterCompletion

```
    private void cleanupAfterCompletion(DefaultTransactionStatus
status) {
        status.setCompleted();

        //只有在isNewSynchronization才会执行clear
        if (status.isNewSynchronization()) {

            //清除事务管理器保存的事务状态，定义信息
            TransactionSynchronizationManager.clear();
        }
        //只有在新事务时才会执行
        if (status.isNewTransaction()) {
            doCleanupAfterCompletion(status.getTransaction());
        }
        //如果有挂起资源的话，那么还原挂起的资源
        if (status.getSuspendedResources() != null) {
            resume(status.getTransaction(),
(SuspendedResourcesHolder) status.getSuspendedResources());
        }
    }
```


(1) TransactionSynchronizationManager.clear

```
    public static void clear() {
        clearSynchronization();
        setCurrentTransactionName(null);
        setCurrentTransactionReadOnly(false);
        setCurrentTransactionIsolationLevel(null);
        setActualTransactionActive(false);
    }
```

(2) doCleanupAfterCompletion

```
    // Remove the connection holder from the thread, if exposed.
    if (txObject.isNewConnectionHolder()) {

TransactionSynchronizationManager.unbindResource(this.dataSource)
;
    }

    // Reset connection.
    Connection con =
txObject.getConnectionHolder().getConnection();
    try {
        if (txObject.isMustRestoreAutoCommit()) {
            con.setAutoCommit(true);
```

```
            }
            //重新设置隔离级别（前一事务隔离级别），readOnly=false
            DataSourceUtils.resetConnectionAfterTransaction(con,
    txObject.getPreviousIsolationLevel());
        }

        //如果是新创建的ConnectionHolder，提交事务时需要释放连接的
    conn.close
        if (txObject.isNewConnectionHolder()) {
                DataSourceUtils.releaseConnection(con,
    this.dataSource);
        }

        //this.transactionActive = false;
        //this.savepointCounter = 0;
        //this.synchronizedWithTransaction = false;
        //this.rollbackOnly = false;
        txObject.getConnectionHolder().clear();
```

（3）resume

在上面clear后，通过resume将挂起的事务connectionHolder，事务状态，事务定义信息重新绑定到当前线程


```
        Object suspendedResources =
    resourcesHolder.suspendedResources;
        if (suspendedResources != null) {
            //重新绑定conHolder
            doResume(transaction, suspendedResources);
        }
        List<TransactionSynchronization> suspendedSynchronizations =
    resourcesHolder.suspendedSynchronizations;
        //还原事务状态、定义信息
        if (suspendedSynchronizations != null) {

    TransactionSynchronizationManager.setActualTransactionActive(reso
    urcesHolder.wasActive);

    TransactionSynchronizationManager.setCurrentTransactionIsolationL
    evel(resourcesHolder.isolationLevel);

    TransactionSynchronizationManager.setCurrentTransactionReadOnly(r
    esourcesHolder.readOnly);

    TransactionSynchronizationManager.setCurrentTransactionName(resou
    rcesHolder.name);
            doResumeSynchronization(suspendedSynchronizations);
        }

    doResume:
```

```
    ConnectionHolder conHolder = (ConnectionHolder)
suspendedResources;

TransactionSynchronizationManager.bindResource(this.dataSource,
conHolder);
```

五. 回滚事务

上面我们分析了方法正常执行后的事务提交，下面我们再来看下方法执行失败后的处理。completeTransactionAfterThrowing：

```
    if (txInfo.transactionAttribute.rollbackOn(ex)) {
        try {

txInfo.getTransactionManager().rollback(txInfo.getTransactionStat
us());
        }
    }else {
        try {

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus
());
        }
    }
```


5.1 是否回滚

首先根据抛出的异常来判断是否进行回滚。如果回滚的话执行rollback方法，否则的话即使抛出了异常还是进行commit。

```
    public boolean rollbackOn(Throwable ex) {

        RollbackRuleAttribute winner = null;
        int deepest = Integer.MAX_VALUE;

        //根据定义的回滚规则判断是否回滚事务，回滚规则包括rollbackFor，
rollbackForClassName，noRollbackFor，noRollbackForClassName
        if (this.rollbackRules != null) {
            for (RollbackRuleAttribute rule : this.rollbackRules)
{
                int depth = rule.getDepth(ex);
                if (depth >= 0 && depth < deepest) {
                    deepest = depth;
```

```
                    winner = rule;
                }
            }
        }

        //如果没有回滚规则的话，默认是抛出RuntimeException、error时回滚
        if (winner == null) {
            return super.rollbackOn(ex);
        }

        return !(winner instanceof NoRollbackRuleAttribute);
    }
```

5.2 事务回滚

```
    private void processRollback(DefaultTransactionStatus status)
{
        triggerBeforeCompletion(status);
        if (status.hasSavepoint()) {
            status.rollbackToHeldSavepoint();
        }
        else if (status.isNewTransaction()) {
            doRollback(status);
        }
        else if (status.hasTransaction()) {

        // isLocalRollbackOnly 是否局部回滚，默认为false
        // isGlobalRollbackOnParticipationFailure 当部分失败后全局回
滚，默认为true
        if (status.isLocalRollbackOnly() ||
isGlobalRollbackOnParticipationFailure()) {

            //getConnectionHolder().setRollbackOnly();  设置
ConnectionHolder.rollbackOnly=true
            doSetRollbackOnly(status);
        }

        }

    ......
    //事务回滚后，清除事务信息，同提交cleanupAfterCompletion
    cleanupAfterCompletion(status);
}
```


从上面的代码可以看出：

（1）如果有保存点的话回滚到保存点（嵌套事务）

```
getConnectionHolderForSavepoint().getConnection().rollback((Savepo
int) savepoint);
```

在开启事务时保存了保存点，回滚时直接回滚到此保存点即可。

（2）如果是新事务的话，那么回滚事务

Connection con = txObject.getConnectionHolder().getConnection();
con.rollback();

（3）如果不是新事务的话，标记事务为rollbackOnly，在提交事务时会判断rollbackOnly标记，如果为true，则进行事务回滚

六. Spring事务中的设计模式

（1）工厂模式

ProxyFactory，BeanFactory。

（2）动态代理

创建事务代理类，JdkDynamicAopProxy。

（3）模板方法

PlatformTransactionManager封装了事务commit，rollback，getTransaction的骨架，具体的实现交由不同的子类来实现，如DataSourceTransactionManager，JtaSourceTransactionManager。

（4）责任链模式

aop拦截器的执行。

（5）策略模式

在创建代理类时，如果代理的是接口那么创建JdkDynamicAopProxy，如果代理的是类那么创建CglibAopProxy。

（6）备忘录模式

在挂起事务时，会在当前创建的TransactionInfo记录OldTransactionInfo，这样当在恢复事务时，可以通过OldTransactionInfo进行还原。

上面这几个设计模式只是spring中比较常见的设计模式，spring中肯定还存在其它的设计模式，期待你的发现。

七. 总结

Spring 源码相对来说比较简单，只要抓住主脉络（创建代理类，创建事务，提交事务，回滚事务），然后通过 debug 一步步跟下来，你也可以深入了解和理解 Spring事务的实现原理 。由于个人时间、精力、水平有限，难免有一些遗漏和错误的地方，还请大家多多指教，谢谢。