69_Spring 事务框架源码初探（一）：事务的开启和开启事务之后再执行业务逻辑

1、开篇

上节课主要介绍 Spring 的事务支持以及传播特性，包括：Spring 对事务的支持，Spring 事务的声明方式，Spring 事务的传播特性三个部分。Spring 对事务的支持中介绍了三个重要的接口，分别负责事务属性、运行状态和事务管理。Spring 事务的声明方式以@Transactional 的注释为主，可以针对类和方法。事务的传播特性在事务嵌套调用的场景会用到，这里列举了 7 个类型供大家参考。本节课带大家看看 Spring 事务框架源码，了解事务的开启和开启事务之后执行的业务逻辑。今天课程的内容包括以下几个部分：

- 事务开启代码解析
- 事务开启大图总结

2、事务开启代码解析

前面的课程提到了 Spring 声明式事务是通过 AOP 增强实现的，所以现在分析 Spring 事务执行过程的时候离不开对 AOP 拦截器执行链的描述，其中比较重要的拦截器就是 TransactionInterceptor 了，于是我们的 AOP 事务开启旅程就从它开始。

如图 1 所示， TransactionInterceptor 继承与 TranscationAspectSupport，同时实现了 MethodInterceptor 和 Serializable 接口。

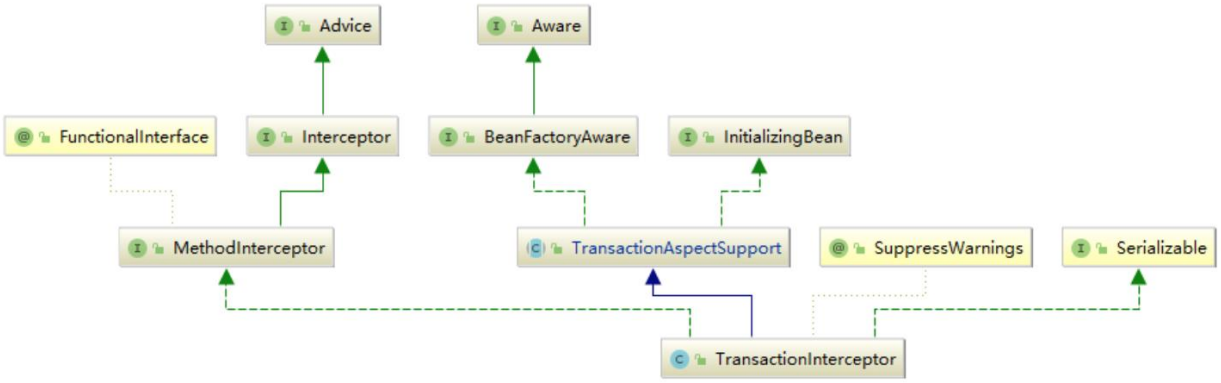

图 1

如图 2 所示，在 TransactionInterceptor 中有一个重要的方法就是 invoke，其中会获取带来对象的类型，同时会调用父类 TranscationAspectSupport 的 invokeWithintransaction 方法传入拦截器执行的方法和目标类，从而返回连接点的执行结果。

```
/**
 * @param invocation 拦截器链的执行器，用于执行下一个拦截器
 * @return 返回连接点的执行结果
 * @throws Throwable
 */
@Override
@Nullable
public Object invoke(MethodInvocation invocation) throws Throwable {
    //获取被代理对象的类类型
    Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);

    //调用父类TransactionAspectSupport方法
    //传入连接点方法对象，代理对象类类型，拦截器执行器MethodInvocation的proceed()方法引用
    return invokeWithinTransaction(invocation.getMethod(), targetClass, invocation::proceed);
}
```

图 2

顺着代码追到 TranscationAspectSupport 的 invokeWithintransaction 方法中去看看，如图 3 所示，该方法传入连接点的方法对象、被代理的对象（用于执行方法）以及执行下一个的拦截器。在方法体里面通过 getTransactionAttributeSource 方法获取事务注解属性源，再通过它获取事务注解属性对象，由事务注解属性对象产生事务管理器，最后的目的是为了生成 TransactionInfo 的事务信息对象，通过这个对象执行并提交事务。需要注意的是，生成对象的方法是 createTransactionIfNecessary。

```java
/** 执行事务逻辑
     * @param method 连接点的方法对象
     * @param targetClass 被代理的对象，用于执行method
     * @param invocation 用于执行下一个拦截器
     * @return the return value of the method, if any
     * @throws Throwable propagated from the target invocation
     */
    @Nullable
    protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
            final InvocationCallback invocation) throws Throwable {
        //事务注解属性源，包含了保存 事务注解属性的缓存 如果为空不执行事务
        TransactionAttributeSource tas = getTransactionAttributeSource();
        //事务注解属性对象
        final TransactionAttribute txAttr = (tas != null ? tas.getTransactionAttribute(method, targetClass) : null);
        //事务管理器
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        //获取连接点唯一标识字符串  com.example.UserService.insertUser()
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);
        //如果是本地事务执行逻辑
        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            //创建事务并返回事务对象信息
            //TransactionInfo 事务信息对象  主要包含 TransactionStatus TransactionAttribute PlatformTransactionManager
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
            try {
                //执行下一个拦截器，如果只有一个拦截器，那么下一个会执行连接点的方法()
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                //事务回滚
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                //清除事务信息
                cleanupTransactionInfo(txInfo);
            }
            //提交事务
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }
```

图 3

上面 invokeWithintransaction 方法中会将事务信息对象执行并提交，其中通过 createTransactionIfNecessary 产生这个事务信息对象，接下来看看这个对象的产生方法 createTransactionIfNecessary。如图 4 所示，该方法中通过传入的 PlatformTransactionManager 获取事务管理器，然后调用其中的 getTransaction 方法获取事务。

```
@SuppressWarnings("serial")
protected TransactionInfo createTransactionIfNecessary(@Nullable PlatformTransactionManager tm,
        @Nullable TransactionAttribute txAttr, final String joinpointIdentification) {
    //如果没有执行名称，则使用joinpointIdentification，并返回代理对象
    if (txAttr != null && txAttr.getName() == null) {
        txAttr = new DelegatingTransactionAttribute(txAttr) {
            @Override
            public String getName() {
                return joinpointIdentification;
            }
        };
    }
    TransactionStatus status = null;
    //事务属性不为null
    if (txAttr != null) {
        //事务管理器不为null
        if (tm != null) {
            //获取事务
            status = tm.getTransaction(txAttr);
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Skipping transactional joinpoint [" + joinpointIdentification +
                        "] because no transaction manager has been configured");
            }
        }
    }
    //创建并初始化TransactionInfo
    return prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
}
```

图 4

getTransaction 的方法比较长，主要是根据事务的传播级别对事务进行处理，由于篇幅原因我们截取其中一段代码。如图 5 所示，红框的部分，首先调用 doGetTransaction 方法获取 transaction 对象。然后，经过事务传播级别判断以后，生成 DefaultTransactionStatus 类型，同时 doBegin 方法开启事务、获取链接、设置数据库事务隔离级别等操作。最后，通过运行 prepareSynchronization 方法初始化事务同步器 TransactionSynchronizationManager。

```java
public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException
    //返回DataSourceTransactionObject 持有ConnectionHolder 数据库连接
    //如果当前线程已经存在事务，则会从ThreadLocal的Map 以DataSource为key获取ConnectionHolder
    Object transaction = doGetTransaction();
    // Cache debug flag to avoid repeated checks.
    boolean debugEnabled = logger.isDebugEnabled();
    if (definition == null) {
        // Use defaults if no transaction definition given.
        definition = new DefaultTransactionDefinition();
    }
    //当前线程是否存在事务
    if (isExistingTransaction(transaction)) {…
    }
    //注意：以下处理逻辑为不存在事务
    if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {…
    }
    //事务传播级别为PROPAGATION_MANDATORY，不存在事务抛出异常
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_MANDATORY) {…
    }
    //以下事务传播级别，在当前不存在事务的情况下，统一创建新的事务
    else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
        SuspendedResourcesHolder suspendedResources = suspend(null);
        if (debugEnabled) {
            logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);
        }
        try {
            boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
            //创建status
            DefaultTransactionStatus status = newTransactionStatus(
                    definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
            //开启事务 获取连接 设置数据库事务隔离级别 设置超时时间 绑定连接资源到本地线程
            doBegin(transaction, definition);
            //初始化事务同步器TransactionSynchronizationManager
            prepareSynchronization(status, definition);
            return status;
        }
```

图 5

doBegin 的代码如图 6 所示，它做了如下主要操作：

- 通过 obtainDataSource 中的 getConnection 方法，从数据源获取新的连接
- 通过 prepareConnectionForTransaction 方法设置数据库事务隔离级别
- 通过 prepareTransactionalConnection 方法设置只读事务
- 通过 getConnectionHolder 中的 setTransactionActive 方法设置事务为生效状态
- 通过 determineTimeout 方法设置连接超时时间

```java
@Override
protected void doBegin(Object transaction, TransactionDefinition definition) {
    DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
    Connection con = null;
    try {
        if (!txObject.hasConnectionHolder() ||
                txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
            //从数据源获取新的连接
            Connection newCon = obtainDataSource().getConnection();
            if (logger.isDebugEnabled()) {
                logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");
            }
            txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
        }
        txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
        con = txObject.getConnectionHolder().getConnection();
        //设置 数据库事务隔离级别
        Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);
        txObject.setPreviousIsolationLevel(previousIsolationLevel);
        if (con.getAutoCommit()) {
            txObject.setMustRestoreAutoCommit(true);
            if (logger.isDebugEnabled()) {
                logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
            }
            //设置为 不自动提交事务 即开启事务
            con.setAutoCommit(false);
        }
        //设置 只读事务
        prepareTransactionalConnection(con, definition);
        //设置事务为生效的
        txObject.getConnectionHolder().setTransactionActive(true);
        //设置连接的超时时间
        int timeout = determineTimeout(definition);
        if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
            txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
        }
        // 绑定新的连接到本地线程 key为对应的数据源
        if (txObject.isNewConnectionHolder()) {
            TransactionSynchronizationManager.bindResource(obtainDataSource(), txObject.getConnectionHolder());
        }
    }
```

图 6

至此事务开启的主要操作就已经完成了，当然还有一些后续操作，我们在大图总结中进行讲解。

3、事务开启大图总结

上面通过对主要源代码的分析了解了 Spring 事务开启的整个过程，这里通过一张大图梳理一下整个过程。如图 7 所示，我们从上往下看，TransactionInterceptor 作为事务拦截器是本次课程的核心，通过其中的 invoke 方法开启事务。其中会调用父类 TransactionAspectSupport 中的 invokeWithinTransaction 方法，这个方法

有两个分支，左边一个是执行业务逻辑的方法，调用的是 invocation 中的 processdWithinInvacation 完成的。

右边的分支用来开启事务，会调用 TransactionAspectSupport 中的 createTransactionIfNecessary 方法，其中会使用到事务管理器 PlatFormTransactionManager 中的 getTransaction 方法获取事务。此处获取事务的操作有两个部分，第一步通过调用 JpaTransactionManager 中的 doGetTransaction 方法返回 JpaTransactionObejct，第二步通过 JpdaTransactionManager 中的 doBegin 方法调用 HibernateJpaDialect 中的 beginTransaction 完成事务的开启。

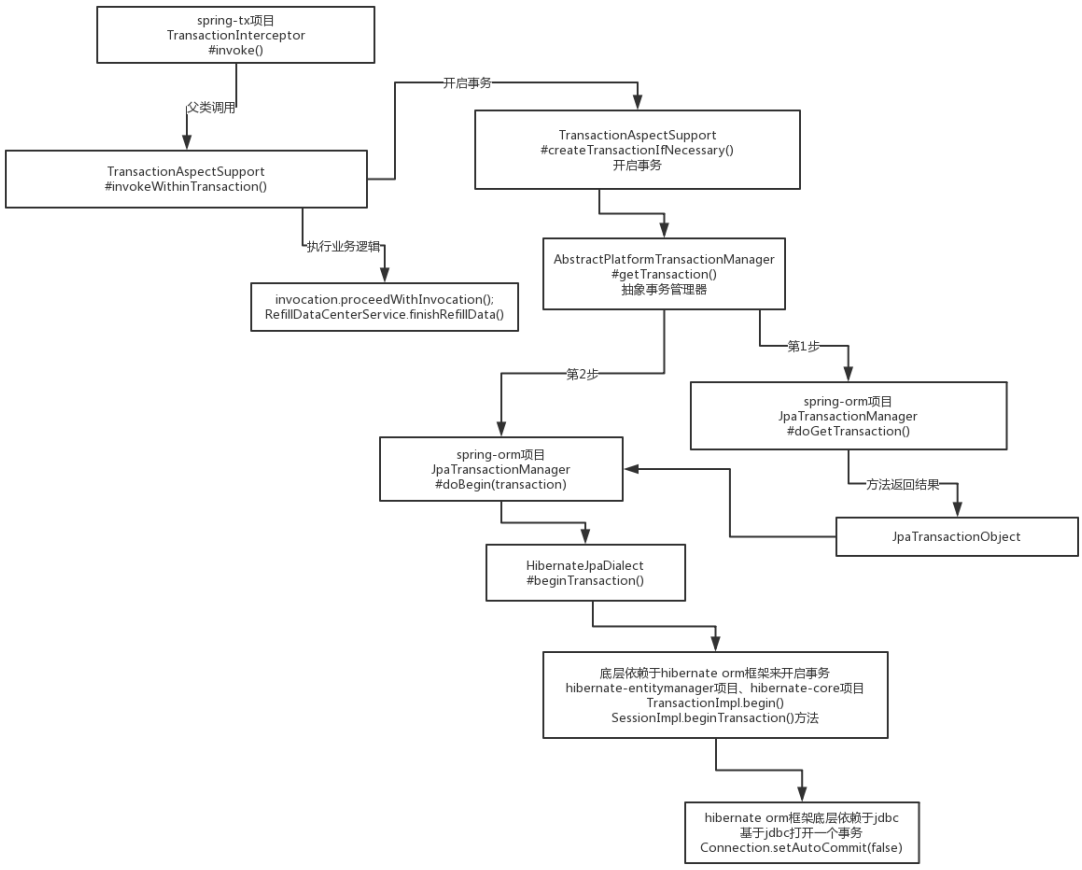

图 7

4、总结

本节课主要围绕 Spring 事务的开启展开，由于 Spring 事务采用了声明事务的方式，用到了 AOP 的拦截器实现，因此 TransactionInterceptor 作为事务拦截器的

核心类就作为了本课程的切入点，通过该类的调用的主要方法将整个过程进行了描述，最后通过一张大图总结事务开启过程。

下节课通过源码分析的方式分析事务提交和回滚的部分。下期见，拜拜。